FLAM
DEVELOPED BY LIMES

LIMES DATENTECHNIK GMBH

FLAM - the complete course

Every unit of the training material

limes datentechnik gmbh

FLAMv5.21-42969

Welcome and introductions.
These materials have been compiled for this training and remain yours in full afterwards - including the parts we do not cover for lack of time. So you can carry on working with them and look things up later.

FLAM DEVELOPED BY LIMES

AGENDA

TOPIC	PAGE
FLAM basics	5
Tools: FLCL and FLCC	21
Documentation and support	36
The command language CLE/P	47
The built-in functions	62
Selecting files, forming names	75
Administration	88
Operation and maintenance	99
The three access routes	113
Access: batch utility	124
Access: subsystem and file system	137
Access from inside the program	150

The reading order of the complete material. Every point is one unit, the page numbers refer into this document.
The order is the learning order: the basics, then the access routes, then the functional subjects. A unit assumes what stands before it, and says so on its title slide.

AGENDA

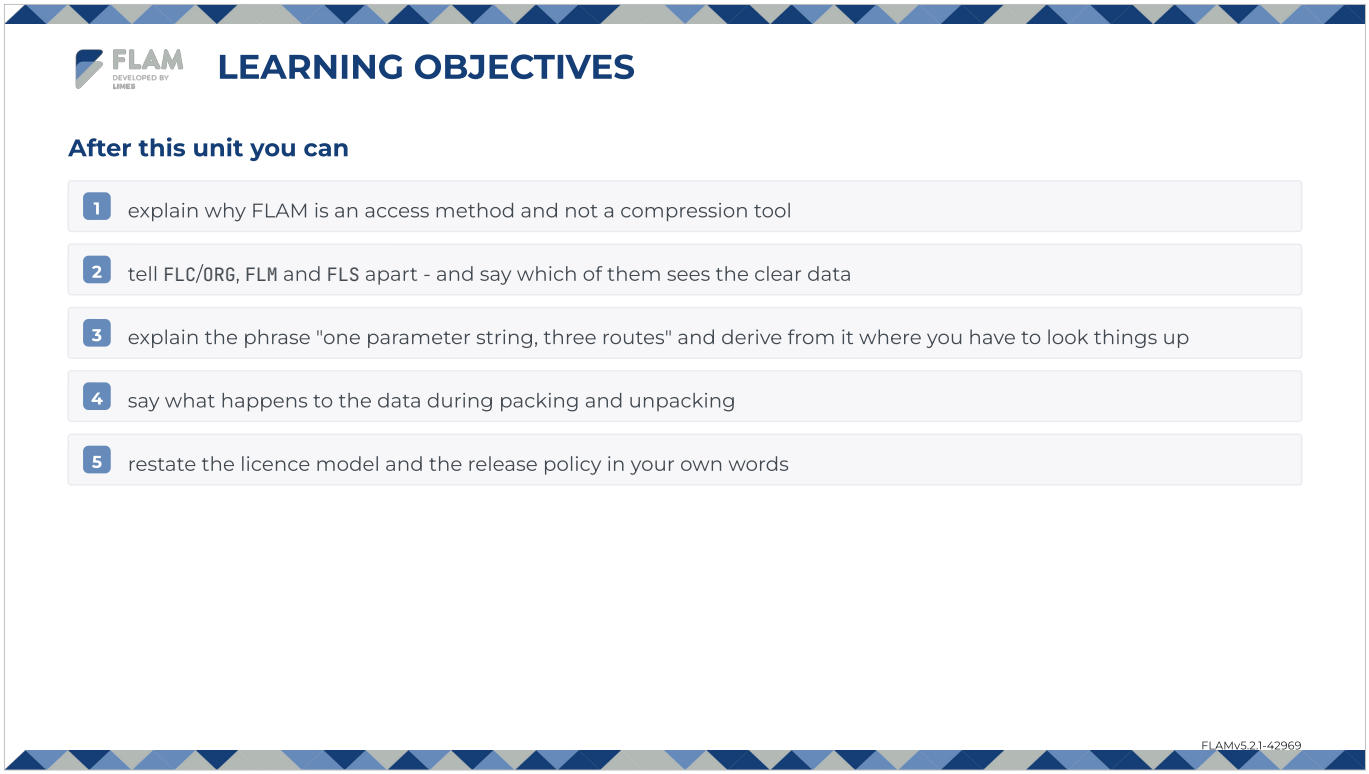
TOPIC	PAGE
Network: SSH and FLIES	163
FLAM4 - architecture and interfaces	176
The FLAM5 archive	189
Indexing, filters and search	205
Managing archived data	221
Pot design	234
Segment store and cloud	250
Conversion with FLUC	264
Compression	283
File formats and records	295
Tables and CSV	309
The format compiler	324

AGENDA

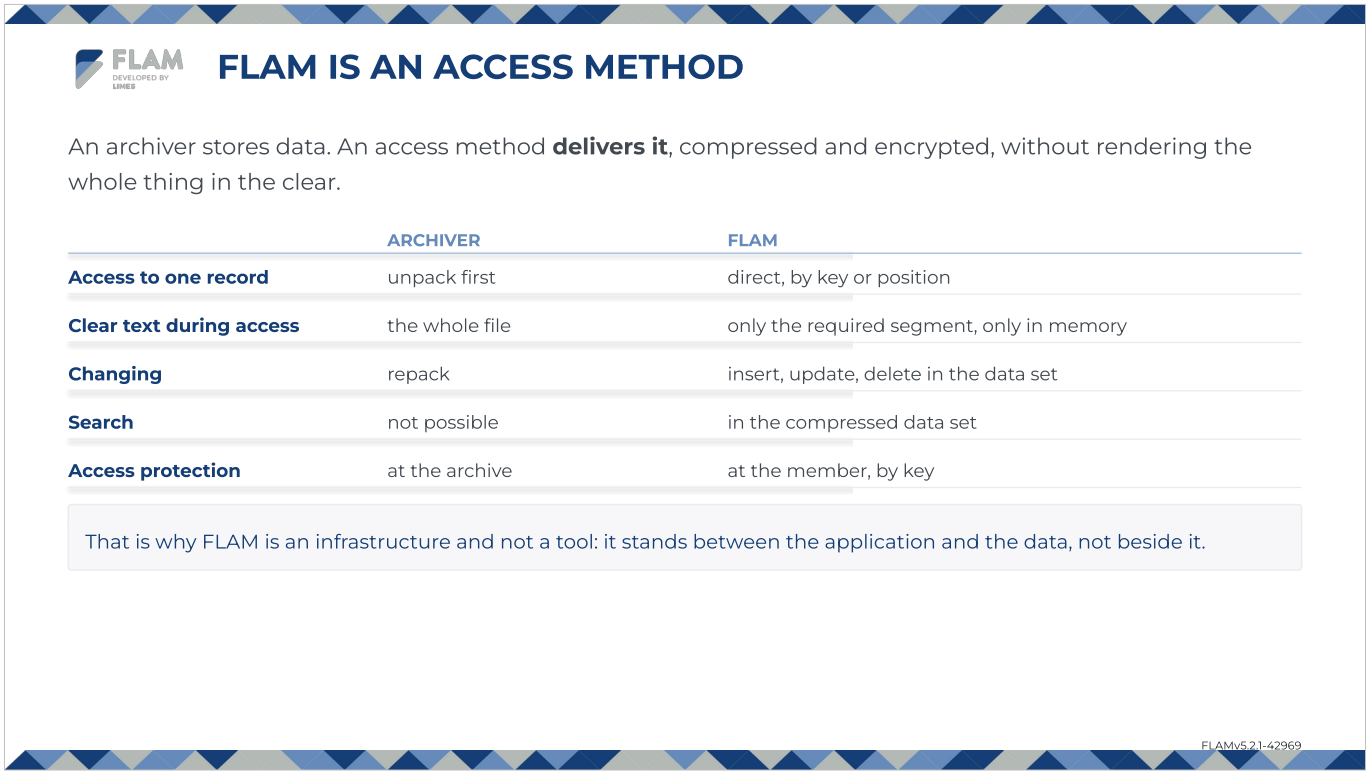
TOPIC	PAGE
Detecting data without a description	339
character sets	355
XML processing	376
Foreign formats and exchange	388
Crypto and keys	399
PGP and signatures	413
Security in operation	426
SAF and RACF	439
Auxiliary functions	452
Embedding into batch jobs	463
Performance and optimisation	479
Accelerators on z/OS	495



First unit of every training. The goal is not completeness but a model that carries: FLAM is an access method, and there are exactly three ways into it. Anyone leaving this unit has to be able to explain the sentence "one parameter string, three ways". Everything else hangs on it.



Objective three is the actual learning objective of the unit; the other four are the vocabulary you need for it.



The most common wrong assumption in the room: "FLAM is like ZIP". Whoever thinks that will look for the wrong solutions later.
The difference is not the compression, it is the access: with an archiver you have to unpack to get at a record. With FLAM you get at the record without unpacking -- and that is exactly what allows archives with search, updates and access control.
If somebody asks "why not just gzip plus openssl": because after that no access is possible any more, only restore.

ONE PARAMETER STRING, THREE ROUTES

The same instruction, packaged three times:

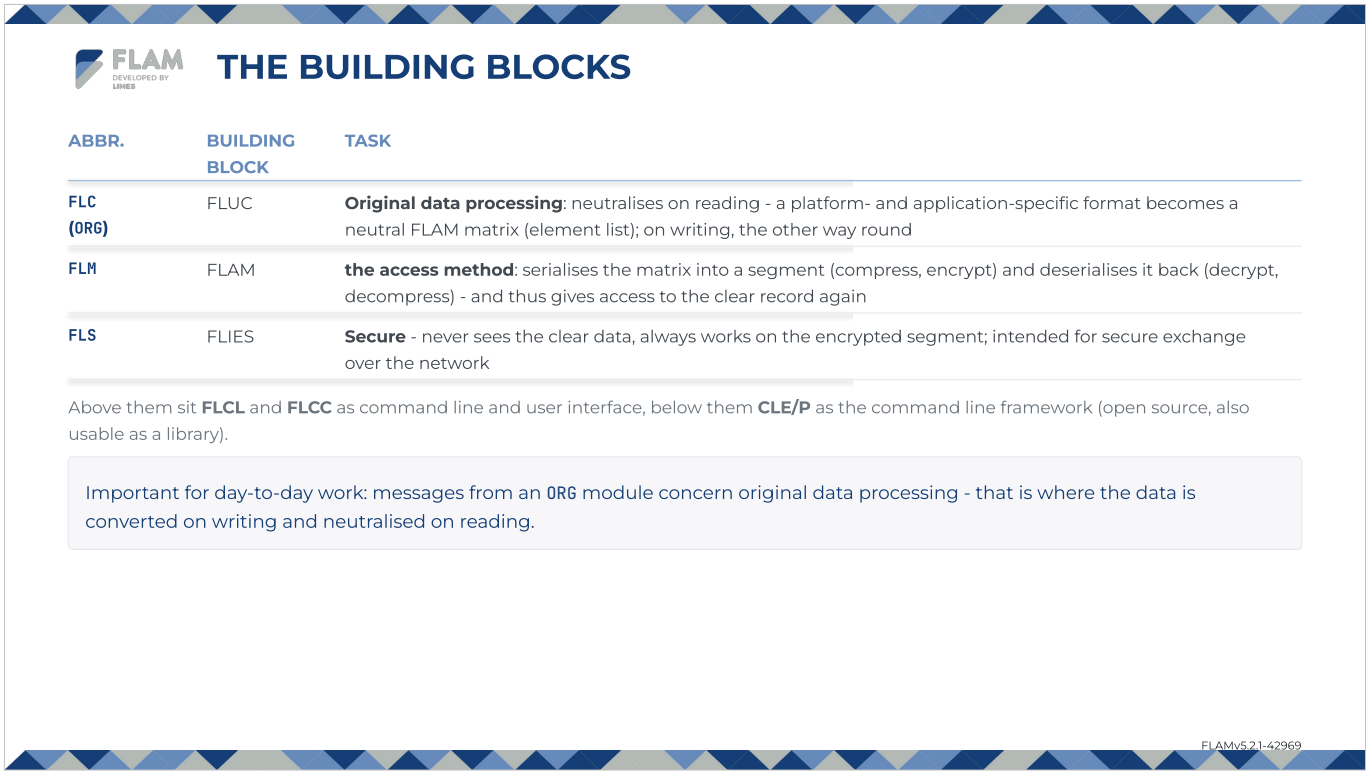
```
read.text(file='eingabe.txt' ccsid='IBM-1141')
write.binary(file='ausgabe.bin')
```

ROUTE	PACKAGING	WHO CHANGES SOMETHING
Batch utility	flcl CONV ... in JCL or shell	Operations
Subsystem, file system	SUBSYS= in the DD statement, mount	Operations
API	parameters of a call (FLCBYT, FLUC-UP, record IF)	Development

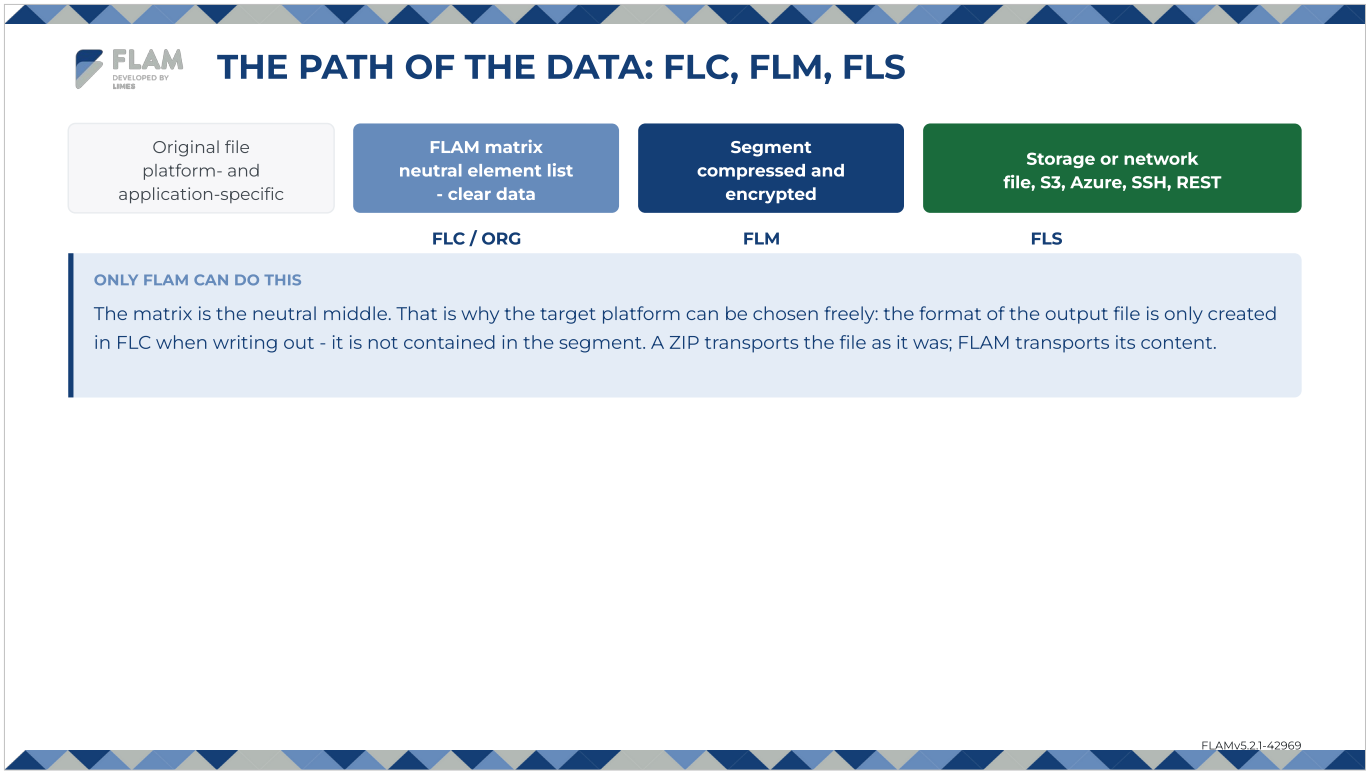
The FLCL manual is therefore also the API manual - it describes the language, not the tool.

FLAMv5.21-42969

This is the load-bearing slide of the whole training, and it appears in this form on flam.de as well: "but since the same parameter strings are also used for the API's and subsystems ... these documents are the basis for the entire FLAM infrastructure".
Consequence for the participants: you learn the language once. Whether the string sits in a JCL, in a DD statement or as a parameter of an API call is a question of packaging, not of content.
Consequence for looking things up: the FLCL manual is also the API manual. That surprises people regularly.



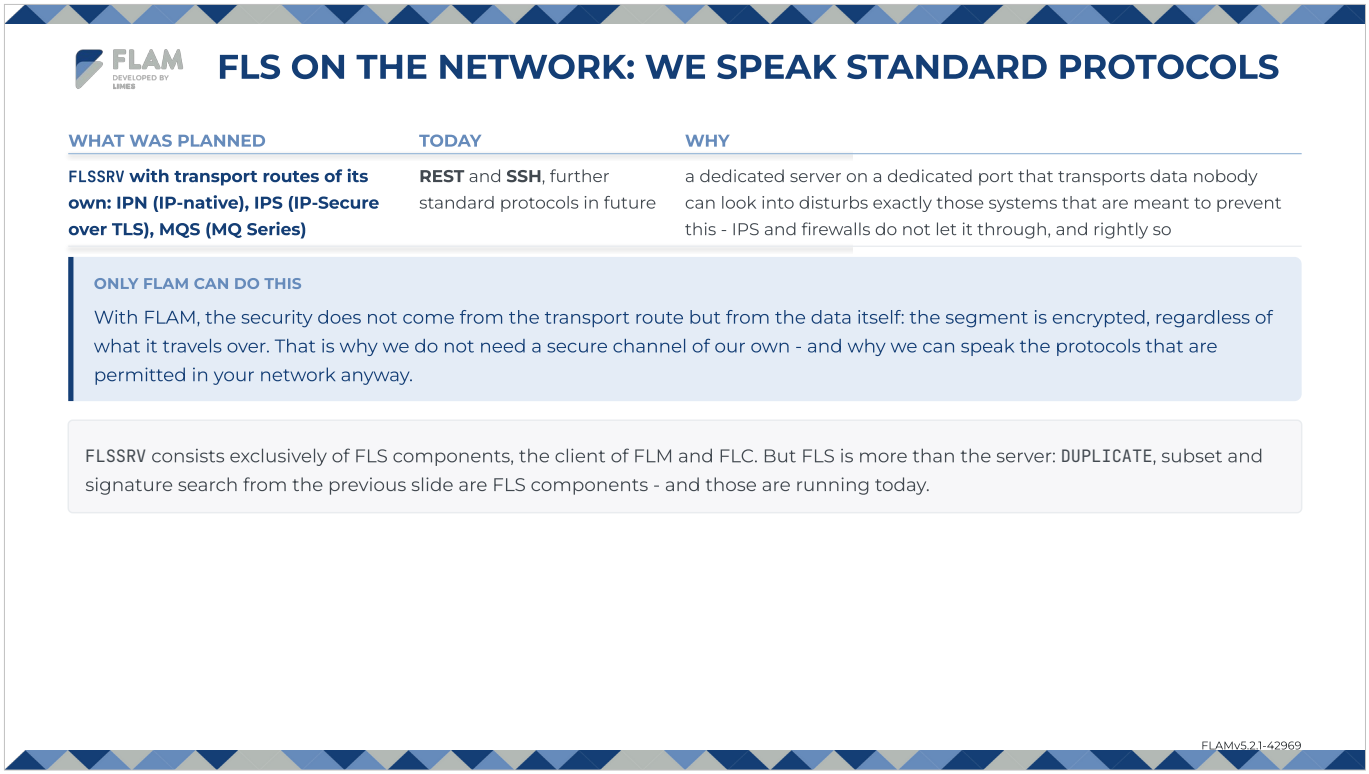
This slide is more important than it looks, because the three abbreviations explain the division of labour -- and because a customer sees the abbreviations in error messages long before knowing the architecture.
The order in the table is the order of the data flow from the outside inwards: FLC takes the original data, FLM turns it into segments, FLS transports them.
FLM is the actual access method. It serialises a matrix into a segment -- that means compress and encrypt -- and deserialises the other way round: decrypt, decompress, and turn the segment back into a matrix on which the clear data record is visible.
FLC is the original data processing. On reading it neutralises: a platform- and application-specific format becomes a neutral FLAM matrix, that is, a member list. On writing exactly the other way round: the neutral member list becomes a concrete format again.
And here is the practical hint that saves time in support: many of these modules are historically called ORG instead of FLC -- ORG for original data processing. If a customer sees an error message from an ORG module, then it is about exactly this layer: the conversion of the original data on writing and the neutralisation on reading.
FLS: the S stands for Secure. FLS components never see the clear data, they always work on the encrypted segment. The next slide spells that out.



The data flow slide. Read it out from left to right and stress where the boundary lies: up to the matrix it is clear data, from the segment on it is not.
On reading: FLC takes the original file in its platform- and application-specific format and neutralises it into a FLAM matrix, that is, a member list. FLM serialises the matrix into a segment -- compress and encrypt. From here on only FLS is at work.
On writing exactly the other way round: FLM deserialises the segment into the matrix, FLC turns it back into a concrete format.
The sentence that should stick: the matrix is the neutral middle. That is why the target platform is freely selectable -- the format of the output file is only created in FLC on writing out, it is not in the segment.



The S in FLS stands for Secure. The core: FLS components never see the clear data. FLS was primarily intended for the secure exchange of data over the network.
Two examples that are already running today.
First, moving. `archive copy(from(...) to.duplicate(...))` moves an archive from S3 to Azure or into a local folder; with a MATCH also only parts of it. Only the segments are moved. Because nothing is unpacked, this route consists exclusively of FLS components -- and that is why for a DUPLICATE there are no compression or encryption parameters to set either.
Second, the search. The process is handed a filter as a signature that reveals nothing about the data. Only the matching segments are collected, as a new FLAM archive. Only at the recipient do FLM and FLC come into play: FLM turns the segments back into matrices, FLC produces the target format.
From that comes the statement that counts in front of a security officer: the archive operator can search and deliver without being able to see the data.
MEMBER access is enough for that, DATA access is only needed by the recipient.
If somebody asks about the FLS server, see the next slide -- that is the honest answer and it belongs here.



This is an honest slide, and it belongs exactly here, because otherwise the question comes: where is the FLS server, the FLSSRV?
The answer: it exists, but we do not ship it, and we will not ship it in the planned form either. FLSSRV was planned for three transport routes of our own -- IPN for IP-native, IPS for IP-Secure over TLS, and MQS for MQ Series from IBM. FLSSRV consists exclusively of FLS components; the client then of FLM and FLC.
The reason for the decision is not a technical shortcoming, but the reality of today's networks: a proprietary server on a port of its own that transports data nobody can look into disturbs exactly those systems that are there to prevent that. IPS and firewalls do not let such a thing through, and rightly so. Anyone who has to push that through against their network department has already lost.
Hence the other route: we speak the protocols that are permitted anyway.
REST and SSH today, further standard protocols later. The security does not come from the transport route but from the data itself -- the segment is encrypted, regardless of what it travels over. That is the point: we do not need a secure channel of our own.
Important for the distinction, otherwise it sounds like a gap: FLS is more than the server. Everything on the previous slide -- DUPLICATE, subset, signature search -- are FLS components and run today. On the network capability of FLAM there is also a deck of its own; the capabilities are there, only the central component is still not finished.

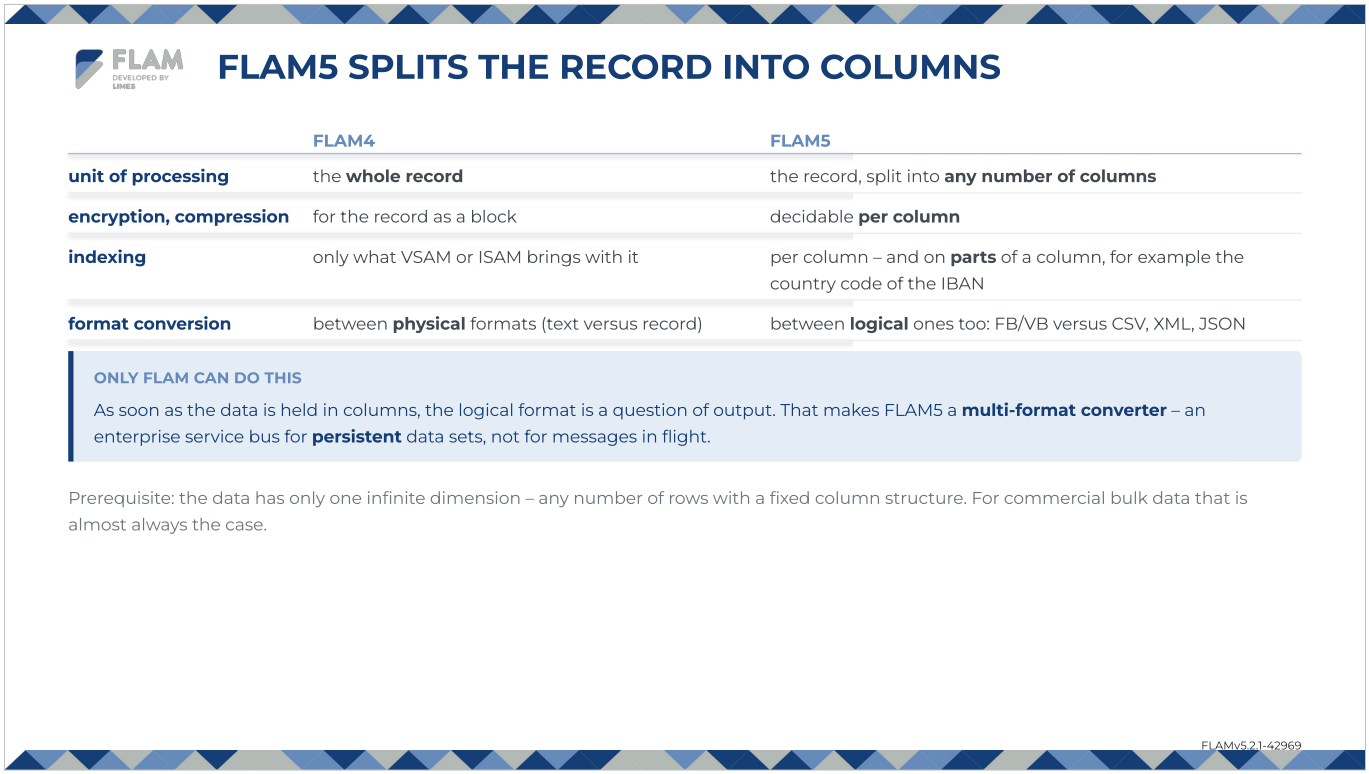
WHAT HAPPENS DURING PACKING AND UNPACKING



- every stage can be switched off individually - you only pay for what you use
- the same path runs backwards: the FLAMFILE is platform-neutral, the target format is only created on writing
- the access decodes only the required segment, not the file
- with **FLAM5** the MACs are **corrected accordingly** on update, insert and delete - completeness and integrity remain protected; with **FLAM4** they become worthless in the process

FLAMv5.21-42969

Four steps, always in this order, and each one can be switched off individually.
What matters is the reversibility: the same route runs backwards, and that is why the target platform is freely selectable -- the FLAMFILE is neutral, the format of the original file is only created on writing out.
The MAC is the point where the pitfall sits later on: record-by-record changes and an integrity proof over the whole file are mutually exclusive.



This is the slide to come back to when somebody later asks what actually distinguishes FLAM5 from FLAM4. Everything else follows from it. **FLAM4 knows only the whole record.** Encryption, compression, integrity -- everything applies to the record as a block. Indexed areas existed only where VSAM or ISAM brought them along. **FLAM5 splits the record into a table with any number of columns.** And from there on every decision is a column decision: which column is encrypted, how it is compressed, whether and how it is indexed. The indexing even goes down to **parts** of a column -- one can index only the country code of an IBAN and search by it without giving away the IBAN itself. Now the conclusion, and that is the real message: once the data lies in columns, the **logical** format is a question of the output. The precondition is that the data has only one infinite dimension -- that is any number of rows with a fixed column structure.
For commercial mass data that is almost always the case. FLAM5 thus converts between **logical** formats: FB and VB on one side, CSV, XML and JSON on the other. Both in both directions. The comparison makes it clear: **FLAM4 converts only between physical file formats** -- text against record. FLAM5 can do that too, and more of it (Unicode and further things), but the difference is the logical level. And from that follows the sentence worth leaving in the room: FLAM5 is thereby a **multi format converter** -- an enterprise service bus for persistent bodies of data. Not for messages in flight, as the usual ESB are, but for the data that is lying on the disk anyway. An addition that matters in the conversation with application development: FLUC can do the same conversion **memory to memory** as well -- without a file being involved. The same technology and the same parameter string therefore apply to the **online business**: a single transaction, an instant payment for instance, is converted in memory. Whoever has described the format logic once as a parameter string uses it in batch and in dialogue -- that is rare, and for a house with both it is an argument.

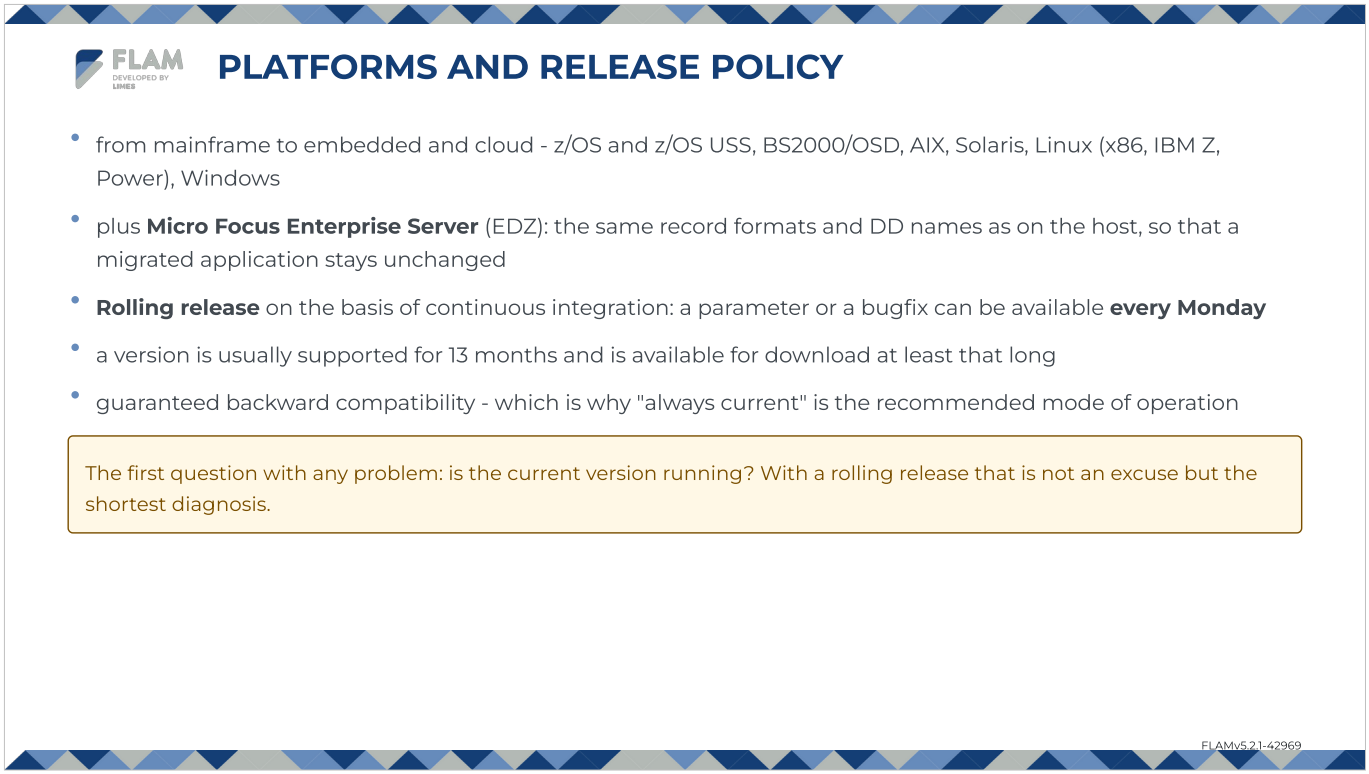
SEGMENT AND MEMBER: WHY ACCESS WORKS

- a **member** is an original file in the archive - with a name, attributes and a key of its own
- a **segment** is a self-contained portion of user data: decodable on its own, without its neighbours
- the access finds the segment and decodes only that one - in memory, never as a file
- this keeps the cost per hit roughly constant, even as the archive grows

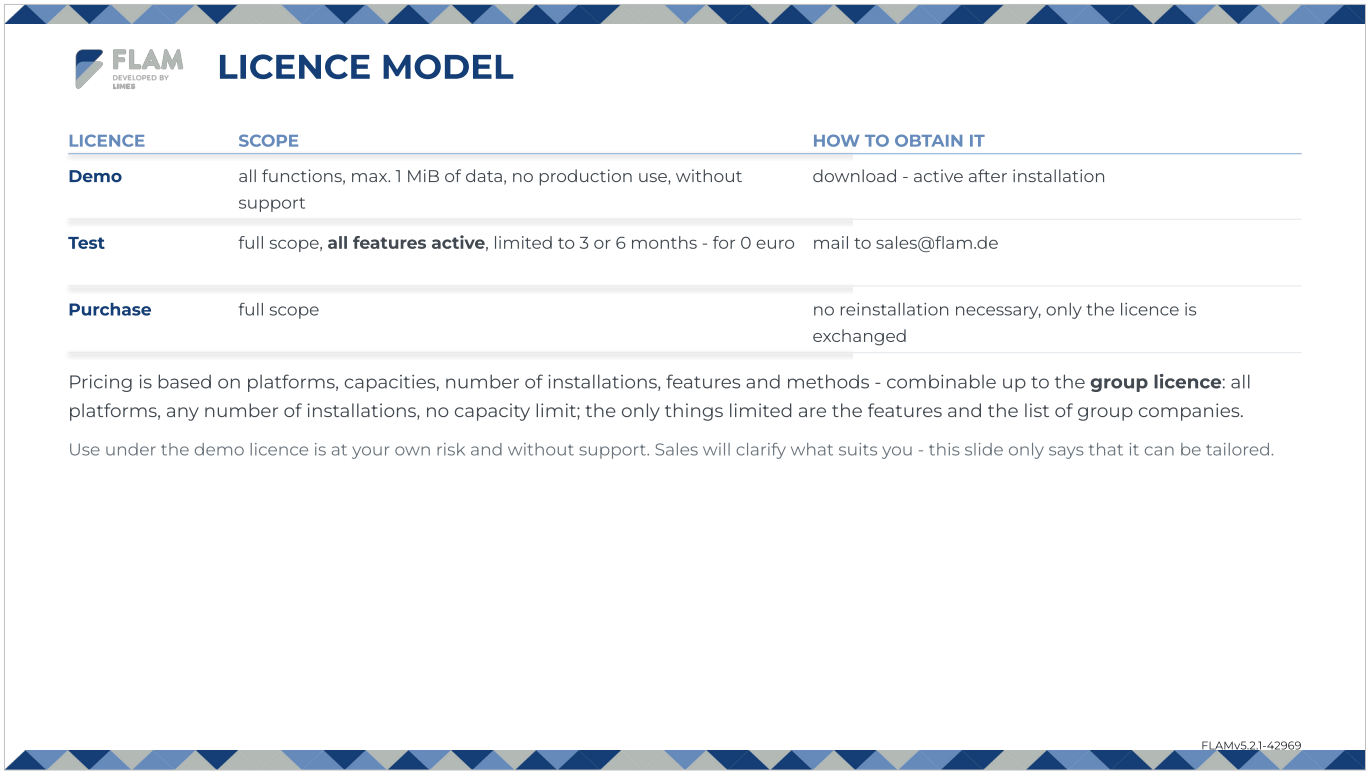
> Only the segment currently required exists in clear text, and only in main memory. That is exactly what several security standards demand.

In depth: decks FLAM5 archive and archive management (matrix, Bloom filter, versioning).

This is the technical justification for the promise of slide 3. Without this slide "access without unpacking" remains an assertion.
The number that should stick: the cost per hit stays roughly constant as the archive grows. That is why a compressed backup becomes a queryable archive. The details come in the decks FLAM5 archive and archive management; only the model here. If somebody asks about the memory requirement -- that belongs in the notes, not on the slide: there are **never more than four matrices in memory at the same time**. With full access these are the directory, the member segment (the index) and **two** data segments: one that travels over the data and carries out updates and deletes, and one at the end for the inserts, that is new records. Whoever only reads (unload) or only writes (load) has three. That number is the answer to "how much main memory does this need" -- it depends on the segment size, but not on the size of the archive.



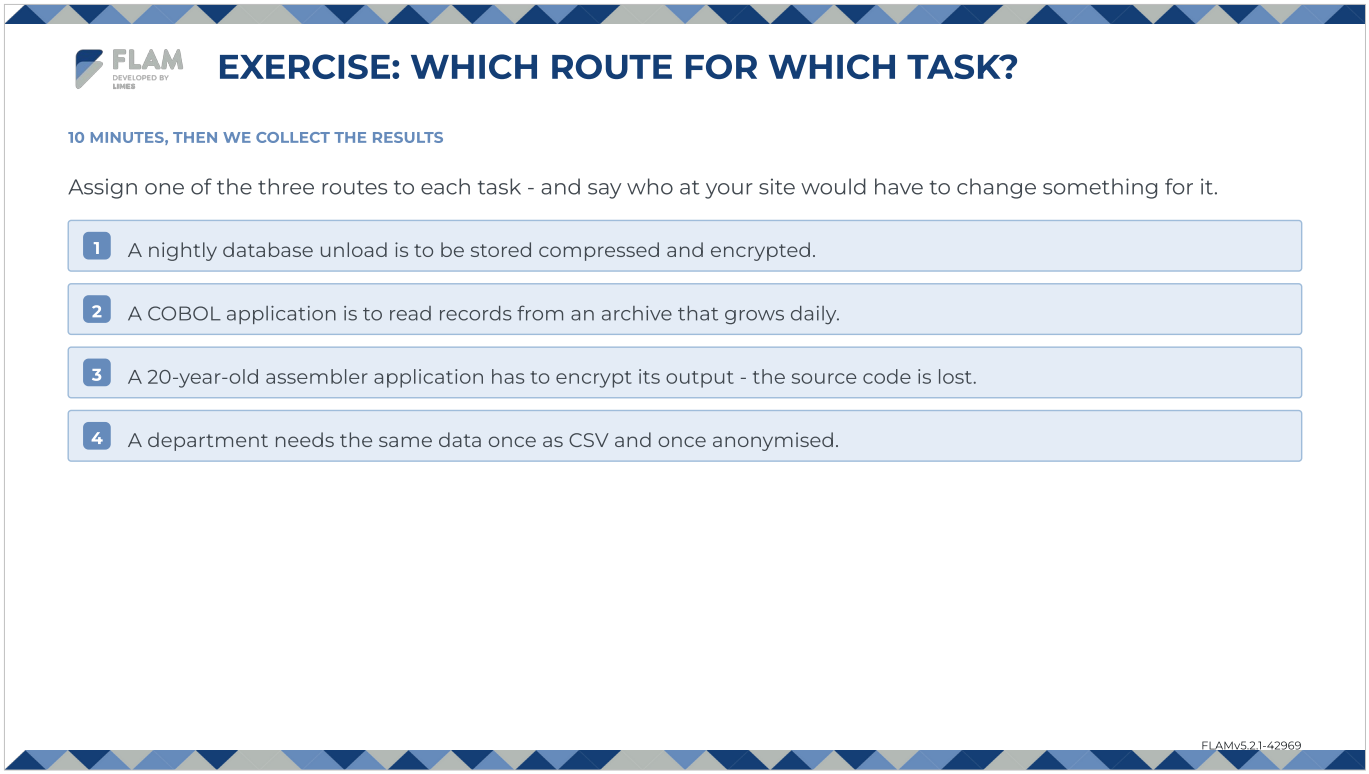
Rolling release is the point that draws questions in production:
there are no fix releases beside the line. An error is rolled out with the next version, and that is why the first support question is always whether the current version is in use.
Backward compatibility is the other half: a FLAMFILE written in 1985 is readable today. That is what allows always running the newest version. One remark on the platform list, because old figures often circulate in the room: **HP-UX, NonStop and z/VSE are no longer supported.** Existing installations keep running and the old manuals stay available, but new versions do not come there any more. Anyone asking usually asks because he still has such a machine himself -- then it is a migration conversation and not a support promise. And the figures on the cadence, because in production they are the real message: we work with **continuous integration**, and from that it follows that a new parameter or a bugfix can be available **every Monday** within the rolling release. Extending an existing component takes a few weeks, a completely new component a few months. That is why "always current" is a recommended mode of operation here and not a risk.



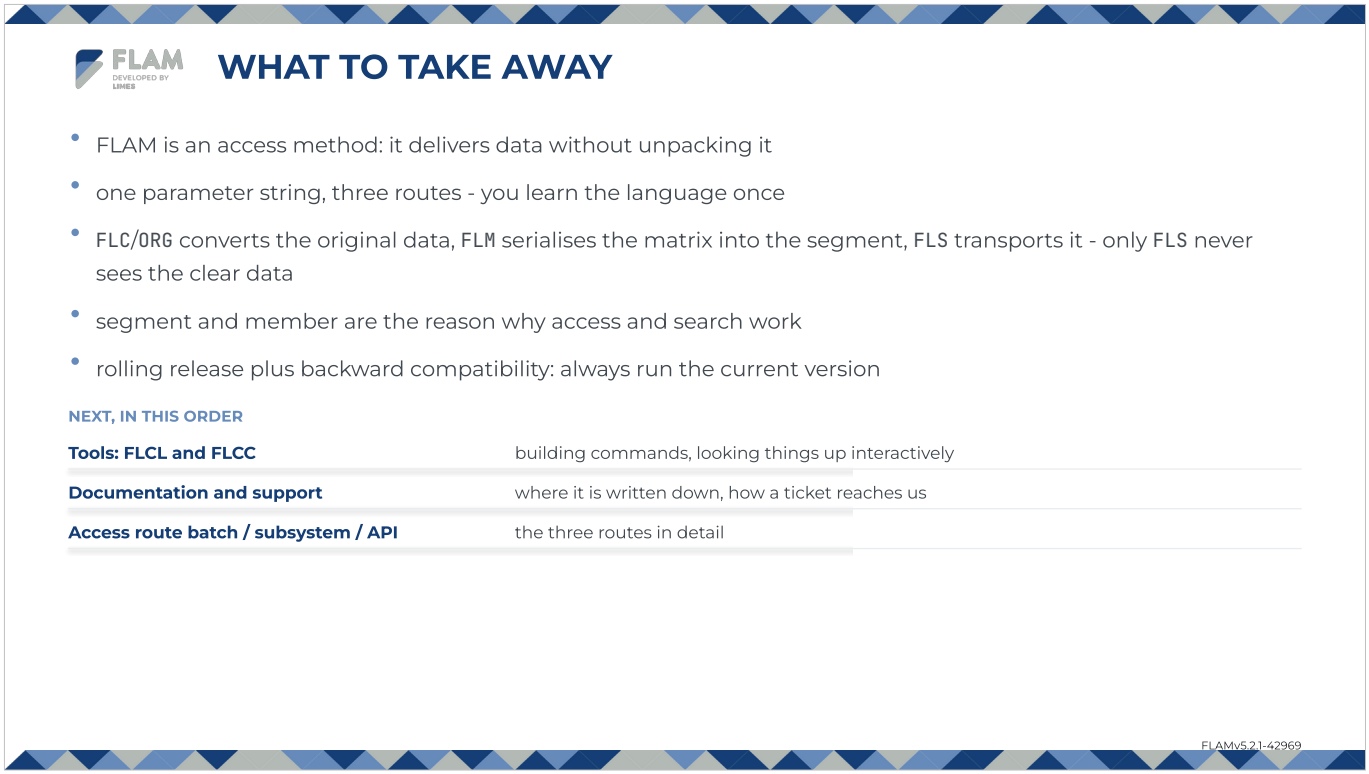
This slide answers the question participants take back to their workplace after the training: how do I try this out.
Important: anybody can download the demo licence and use FLAM with it up to 1 MiB.
If you want to test larger files, you need a test licence -- limited to three or six months, with all features active, and for 0 euro. For that the customer contacts sales@flam.de.
On purchase no reinstallation is necessary, only the licence is swapped.
Do not read out the second table, say the sentence below it instead: the basis of measurement is negotiable, and that is a real difference from standard products. Whoever has a group licence no longer counts installations or capacities -- the only limits are the features and the list of group companies. Details belong in the sales conversation, not in the training; here it is only about the participants knowing that these options exist.



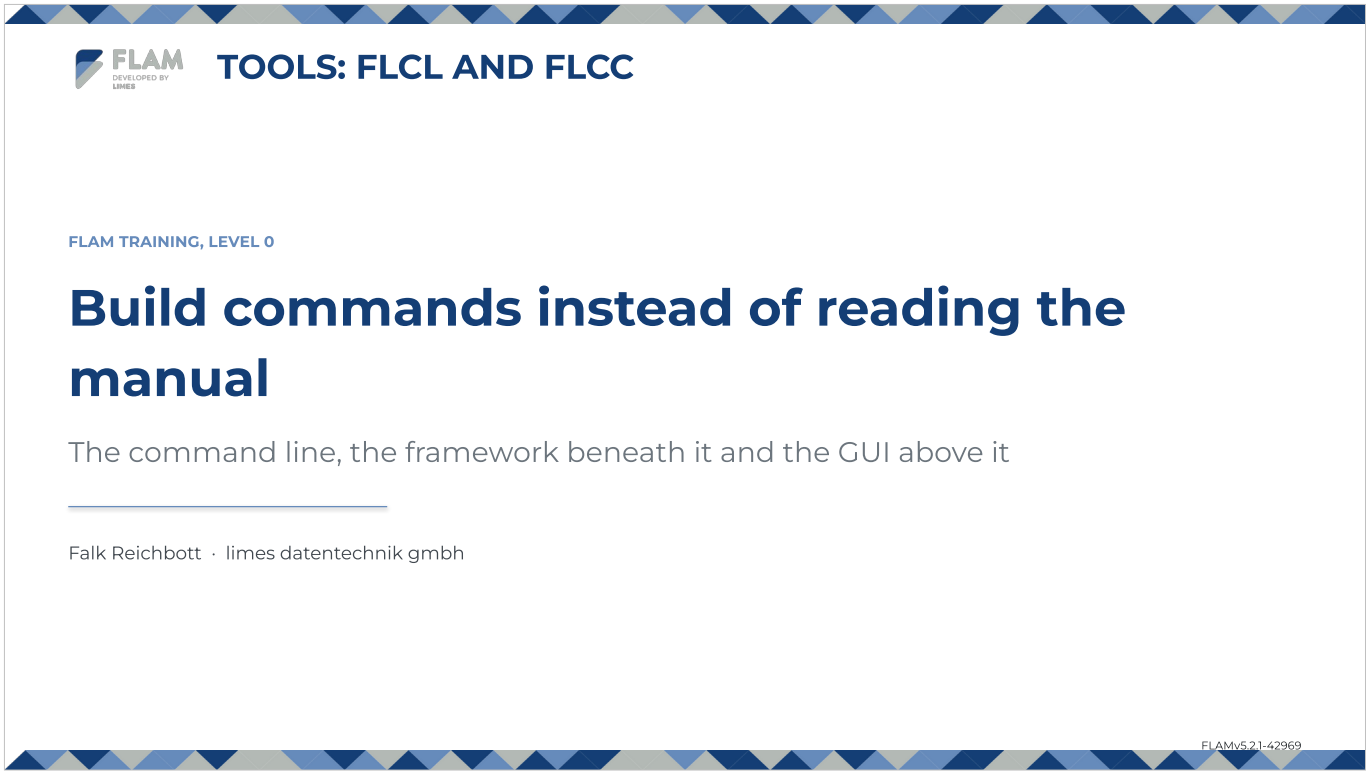
The slide before says **what** is licensed. This one says **how** it is paid for -- and that is the question that comes afterwards from purchasing.
Four models, and they do not differ in the range of functions but in the form of payment:
Purchase is the perpetual right of use, paid once.
Rental is a time-limited right of use and costs one fifth of the purchase price. If a maintenance contract exists, a 50 per cent discount applies on top. That is the route for projects with a foreseeable end.
Hire purchase is the most interesting one for mid-sized companies: a **perpetual** right of use, but paid over a minimum term. At three years a third of the purchase price per year, at four years a quarter, at five a fifth. In the end the right of use belongs to the customer, without having to raise the whole sum at the start.
Consumption is the model that abolishes the basis of measurement entirely: what is paid for are the processed megabytes, measured via our usage log. And then capacities, number of installations, platform and features **no longer matter**. For a customer who does not know how their usage will develop, that is the lowest-risk model -- and for one who runs FLAM on many small machines, often the cheapest.
The pointer to the usage log is doubly worthwhile here: it is the same log that serves for analysis in the performance part. Whoever switches it on sees their consumption before choosing the model.
Do not get into the price discussion. The slide is only meant to make sure that nobody leaves the training with the idea that there is only purchase.



Ten minutes, then collect the answers. The goal is not the right answer, but that the participants ask the question "who has to change something" themselves.
The third line is the interesting one: an application that must not be touched almost always leads to the subsystem -- that is the aha moment.



Read out the learning objectives once more and ask whether all five have been reached.
Next units in the recommended order: tools (FLCL and FLCC), then documentation and support, then the access routes in detail.



Second unit. After the model comes the tool. The sentence that should stick: there are more than 650 manual pages and nobody reads them through -- FLCC is the interactive manual, and it delivers the string you then use everywhere.



Objective four is the working objective of the unit: in the end everybody should be able to build a string in FLCC and take it over into a JCL.

THE 15 COMMANDS OF FLCL

GROUP	COMMANDS	WHAT FOR
Archive	ARCHIVE, FLAM	create, read and manage FLAM5 archives and FLAMFILEs
Conversion	CONV, ICNV, XCNV	convert formats, character sets, encodings
Tables and CSV	BCSV, ACSV, FCSV	binary data and tables to CSV, with and without copybook
Checking and comparing	DIFF, XCHK, HASH	differences, structure and integrity checking
Searching	GREP	in files and archives, also with regex
Keys	KEY	manage keys and keysets
Environment	INFO, UTIL	information about installation, licence and files; `UTIL` handles file tasks via wildcards: search, delete, rename

`flcl ARCHIVE` alone has 15 subfunctions - COMP, DECO, LIST, UPDATE, DELETE, FILTER, PRUNE, REKEY, VERIFY and more.

FLAMv5.21-42969

Do not read out. The participants should see that the user interface is manageable -- the volume lies in the parameters, not in the commands.
If you have to save time: explain only CONV, ARCHIVE, INFO and GREP, and deal with the others as "same language, different verb".

THE LANGUAGE: CLE/P

```
flcl CONV read.text(file='ein.txt' ccsid='IBM-1141')
         write.binary(file='aus.bin')
```

ELEMENT	NOTATION	MEANING
Command	CONV	what is done
Object	read.text(…)	groups parameters that belong together
Overlay	read.text / read.binary	selection of one of several variants
Assignment	file='ein.txt'	string in single quotes
Array	file[a b c]	several values, bracket instead of equals sign
Number	blksize=32760	without quotes

CLE/P is open source and available on our GitHub - you can use the same command line for your own tools.

FLAMv5.21-42969

This slide is the vocabulary list. Whoever understands it can read any string, even one they have never seen before.
The most important thing is the nesting: an object in brackets contains further parameters. That is exactly why the strings are long -- not because the language is complicated but because it is complete.
CLE/P is open source and sits on our GitHub; that surprises many.

FLAM
DEVELOPED BY LIMES

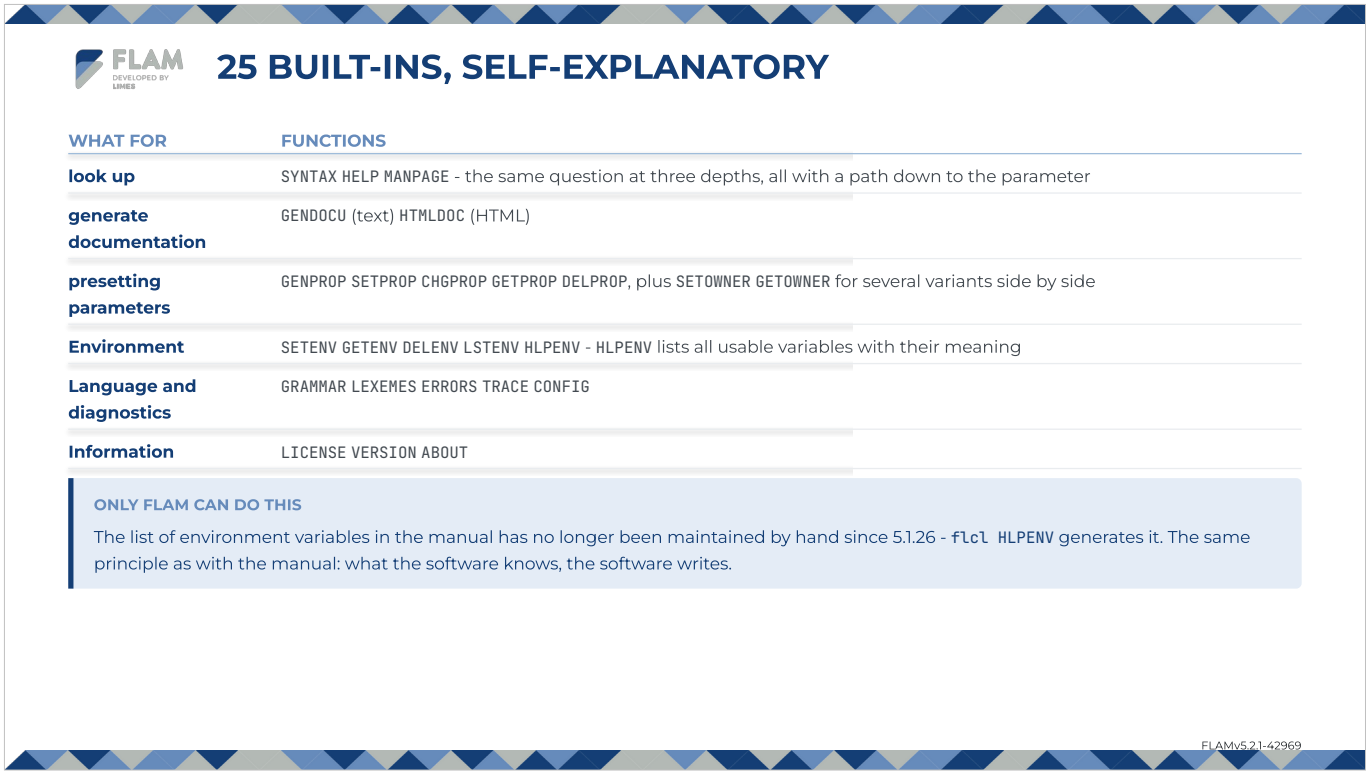
THE BUILT-IN HELP

CALL	WHAT IT DOES
`flcl HELP`	overview of all commands
`flcl HELP CONV`	help on one command
`flcl HELP CONV.read.text.ccsid`	down to the individual parameter
`flcl MANPAGE CONV`	the manual page for it
`flcl GENDOCU CONV=conv.txt`	generate part of the manual
`flcl SYNTAX / GRAMMAR / LEXEMES`	explain the language itself

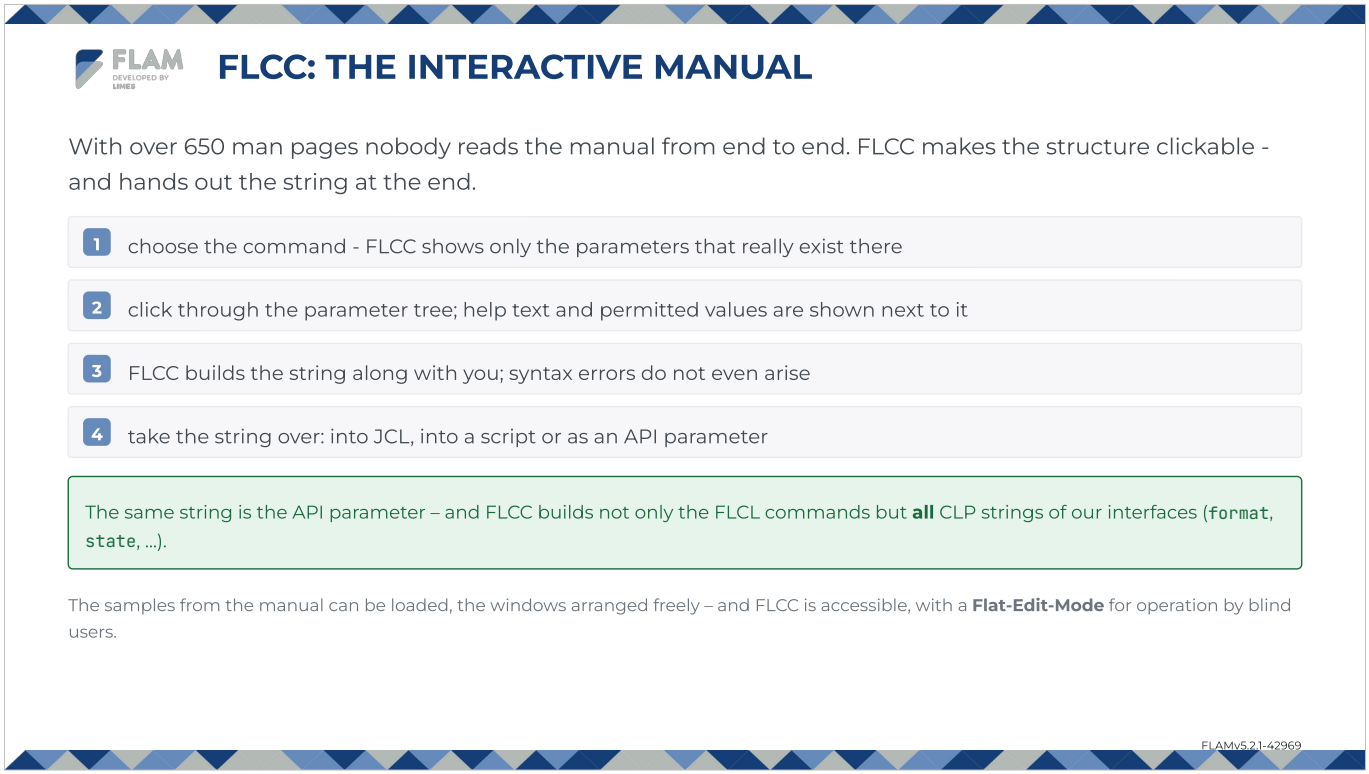
- the software generates its manual itself - so the documentation always matches the installed level
- `HELP` goes right down to the individual parameter, not just to the command
- the whole manual or individual parts can be generated, also for your own in-house documentation

FLAMv5.21-42969

This is the part that saves the most time in operations and is the least known: flcl can generate the manual itself, and exactly for the level that is installed. That is why the documentation cannot go out of date.
In practice: HELP with a path goes all the way down to the individual parameter.
Demonstrate it, do not talk about it.

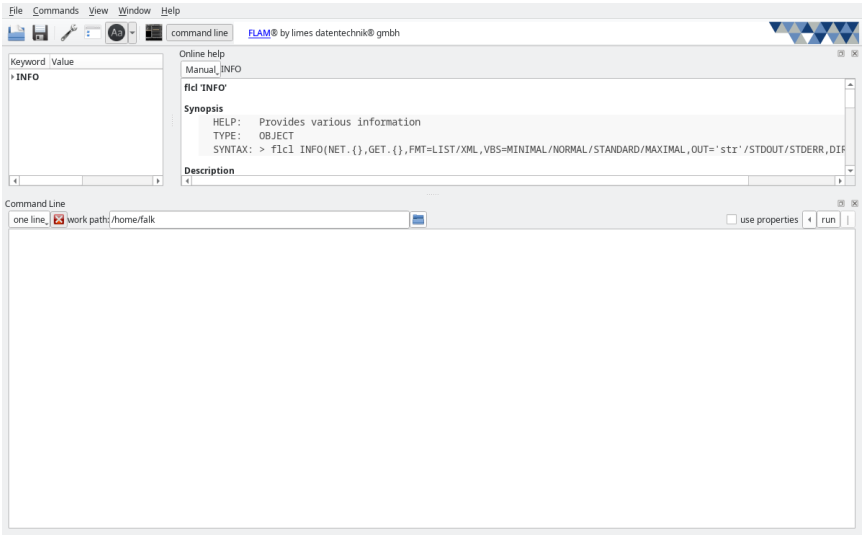


Twenty-five built-in functions, and most users only know HELP.
They are not an accessory: chapter 2 of the manual consists of them, and they are the reason why you do not need a manual for the everyday questions.
Do not read out all of them. Explain the groups and show one representative per group.
Group one, looking things up: SYNTAX, HELP, MANPAGE. The difference is the depth -- SYNTAX gives the one line, HELP the explanation, MANPAGE the whole text. All three take a path and go all the way down to the parameter.
Group two, generating documentation: GENDOCU produces the manual or a part of it as text, HTMLDOC as HTML. That is the route to your own in-house documentation.
Group three, properties -- and that is the operationally most important group.
GENPROP generates a property file with all parameters and their defaults, SETPROP activates it, CHGPROP changes individual values, GETPROP shows the current state, DELPROP withdraws it. With that an organisation sets its standards as defaults once and the commands become short.
With SETOWNER and GETOWNER you can keep several variants side by side -- typically test against production.
Group four, environment: SETENV, GETENV, DELENV, LSTENV and HLPENV. HLPENV is the practical tip, because it lists all usable environment variables together with their meaning -- the manual has not maintained this list by hand since 5.1.26, because HLPENV generates it.
Group five, language and diagnostics: GRAMMAR and LEXEMES output the grammar and the lexemes of the command language, ERRORS the list of completion codes, TRACE switches tracing on, CONFIG shows and changes the configuration.
Group six, information: `LICENSE`, `VERSION`, `ABOUT`. `LICENSE` is the one you really need in operations -- it shows which features are active.



The core of this unit. State the reason openly: the number of parameters is so large that nobody reads the manual through. FLCC makes the structure clickable and delivers a string at the end. Important for developers in the room: the same string is the API parameter. FLCC is therefore also the tool for API calls, not only for the command line.
And that goes further than most assume: **FLCC offers not only the FLCL commands but also the remaining API strings** -- `format`, `state` and the others. So you can build **all** CLP strings for our interfaces with it, not just the ones for the batch utility. Whoever programs against the API clicks their format string together here instead of copying it out of the manual. That covers the subprograms too.
Three things about the handling that always go down well in the room:
FLCC can **load the samples from the manual**. So you do not have to start from zero but take the fitting example and modify it.
The **windows are freely arrangeable** -- the documentation for instance at the top right, so that tree, help and generated string are visible at the same time. That is exactly how the screenshot on the next slide is built up.
And FLCC is **accessible**: there is a **flat edit mode** that considerably simplifies operation by blind users. With public sector clients that is regularly an award criterion and is worth mentioning.
Demonstrate: click a CONV command together in the tree, copy the string, put it into a JCL, run it.

FLCC: PARAMETER TREE

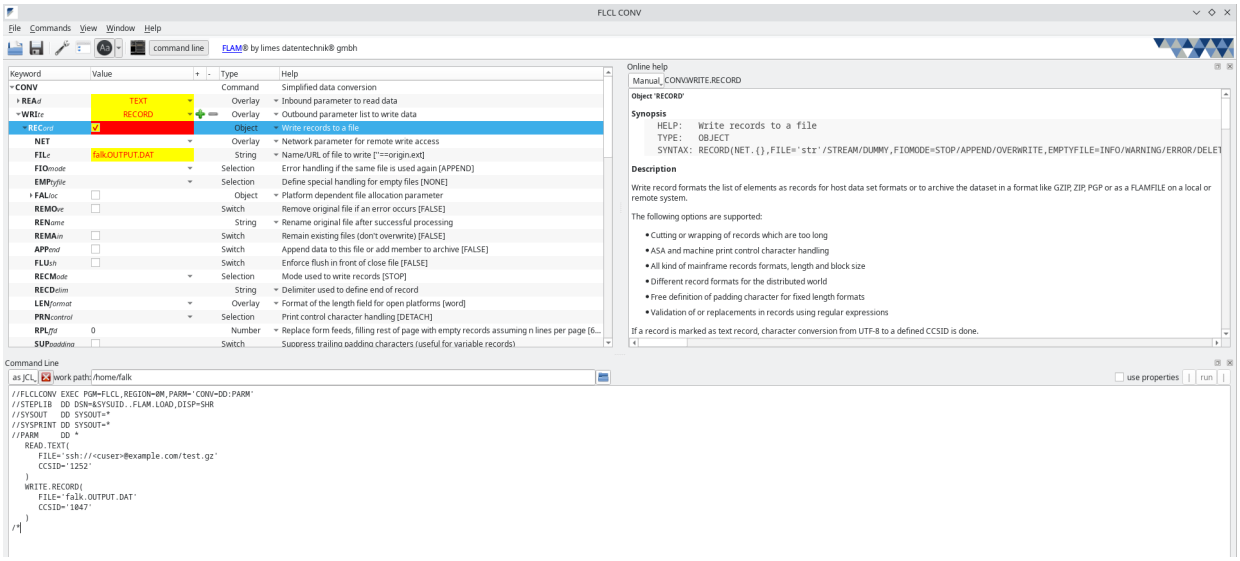


FLCC: on the left the parameter tree, on the right the online help with synopsis and syntax, at the bottom the command line with working path, property switch and execute

FLAMv5.21-42969

Demonstrate this live if a machine is available; the screenshot is the fallback.
Show them: the objects on the left, the values on the right, the growing string at the bottom.
Stress that FLCC only offers what is allowed at that point -- that is the difference from a text editor with the manual open beside it.

FLCC: THE SAME STRING, THREE TARGETS



The expanded parameter tree of a CONV command with the generated string

The bridge to the three-way slide. The string from FLCC travels unchanged into all three wrappers. If there is time: copy it into a JCL once and run it. The effect "is that all?" is exactly the one we want.

THE SAME THING OVER THE THREE ROUTES

BATCH UTILITY

```
//CONV EXEC PGM=FLCL,PARM='CONV'
//FLAMPAR DD *
  read.text(file='ein.txt' ccsid='IBM-1141');
  write.binary(file='aus.bin');
```

SUBSYSTEM

```
//DDNAME DD DSN=datei,SUBSYS=(FLAM,'CRYPTOM=AES')
```

API

```
pvRed = fcbopen("read.text(file='ein.txt' ccsid='IBM-1141')", ...)
```

The content of the brackets is identical. Only the packaging changes.

FLAMv5.21-42969

This slide recurs in the same form in every function deck. This is the first time it appears, so explain it in full once.
The message: the content of the parentheses is identical. What differs is only who passes the string.

PRESETTING PARAMETERS

SOURCE	PURPOSE	TYPICAL USE
Property file	preset parameters permanently	in-house standards: CCSID, block size, report paths
Parameter file	move a long string out of the way	the JCL stays readable, change without changing the JCL
Configuration	several variants side by side	test against production, tenants
Environment	environment and paths	operating-system-dependent items
environment per parameter	preset an individual parameter, without a file	path in upper case, dots to underscores: `CONV_WRITE_TEXT_METHOD=UNIX`
Command line	the individual case	overrides everything before it

With owner and program in front you can narrow this down: `HUGO_FLCL_CONV_READ_TEXT_ENL2LF=ON` applies only to owner `hugo`. `flcl GENPROP` lists what can be preset at all.

> If a parameter "does not work", a later source almost always overrides it. Check the order, not the parameter.

FLAMv5.21-42969

The reason production strings are short: everything recurring lives in a property file or a configuration, not in the command.
The order of precedence is the most common source of error in operations -- if a parameter "has no effect", a later source is almost always overriding it.
That one belongs on the whiteboard.
And there is one route that exists in addition to the property files and that hardly anyone knows: **every parameter can be given a default through its path as an environment variable.**
The rule is mechanical -- take the path, put it all in upper case, replace the dots with underscores. So `CONV.WRITE.TEXT.METHOD` becomes `CONV_WRITE_TEXT_METHOD`.
You can put the program and the owner in front of it to narrow it down: `HUGO_FLCL_CONV_READ_TEXT_ENL2LF=ON` applies only to the owner `hugo`. The owner-specific entry beats the program-specific one, and that beats the global one.
This is the most convenient way to set a default in a test system without creating a file -- and `flcl GENPROP` produces the list of all properties if you want to know what can be given a default at all.



Two built-in functions that hardly anyone knows and that give the tool part of its expressive power: `flcl lexemes` and `flcl grammar`.
`grammar` prints the grammar of the language, `lexemes` the lexemes -- that is, what a number, a string, a comment is. That sounds academic, but two sections of it are needed regularly in production.
First, the predefined constants. They can be used in value expressions, and there are more of them than you would expect: `NOW`, `MINUTE`, `HOUR`, `DAY`, `YEAR` for time arithmetic; `KiB`, `MiB`, `GiB`, `TiB` for sizes; `PI` as a floating-point number; `RND1` through `RND8` for random numbers. And a whole range of timestamps in ready-made formats -- `LCSTAMP`, `LCDATE`, `LCYEAR`, `LCMONTH`, `LCTIME` for local time, the same ones with `GM` in front of them for Greenwich.
And you can **calculate** with them: `bufsiz=4*MiB` *is valid. That is far more readable than* `4194304`*, and that is how it stands in the JCL.*
Second, access to the environment. Inside a string, `<name>` is replaced by the value of the environment variable. On z/OS this also covers exported JCL symbols and system symbols -- which makes a procedure reusable without the JCL being touched.
The distinction matters, because everyone walks into this once: **the constants are values, not text snippets.** `file=LCSTAMP` produces a file with the timestamp as its name.
`file='/pfad/<LCSTAMP>.csv'` does **not** -- inside a string only the environment variable is replaced, and `LCSTAMP` is not one. If you need the timestamp in the file name, use the name tokens (`[base]`, `[indN]`) or set an environment variable beforehand.
Measured, not assumed -- both cases behave that way.

PITFALLS

! **Single quotes are doubled in JCL.**

In a JCL a single quote is written as two. That is not a FLAM peculiarity but JCL - and the most frequent cause of a "syntax error" in a ticket.

! **An array uses brackets instead of an equals sign.**

`file[a b c]`, not `file=[a b c]`. The bracket expression replaces the assignment.

! **Property files take effect even when you have forgotten them.**

With unexpected behaviour first check `flcl INFO GET.SYSTEM` and the active configuration.

! **Quotes are mostly unnecessary.**

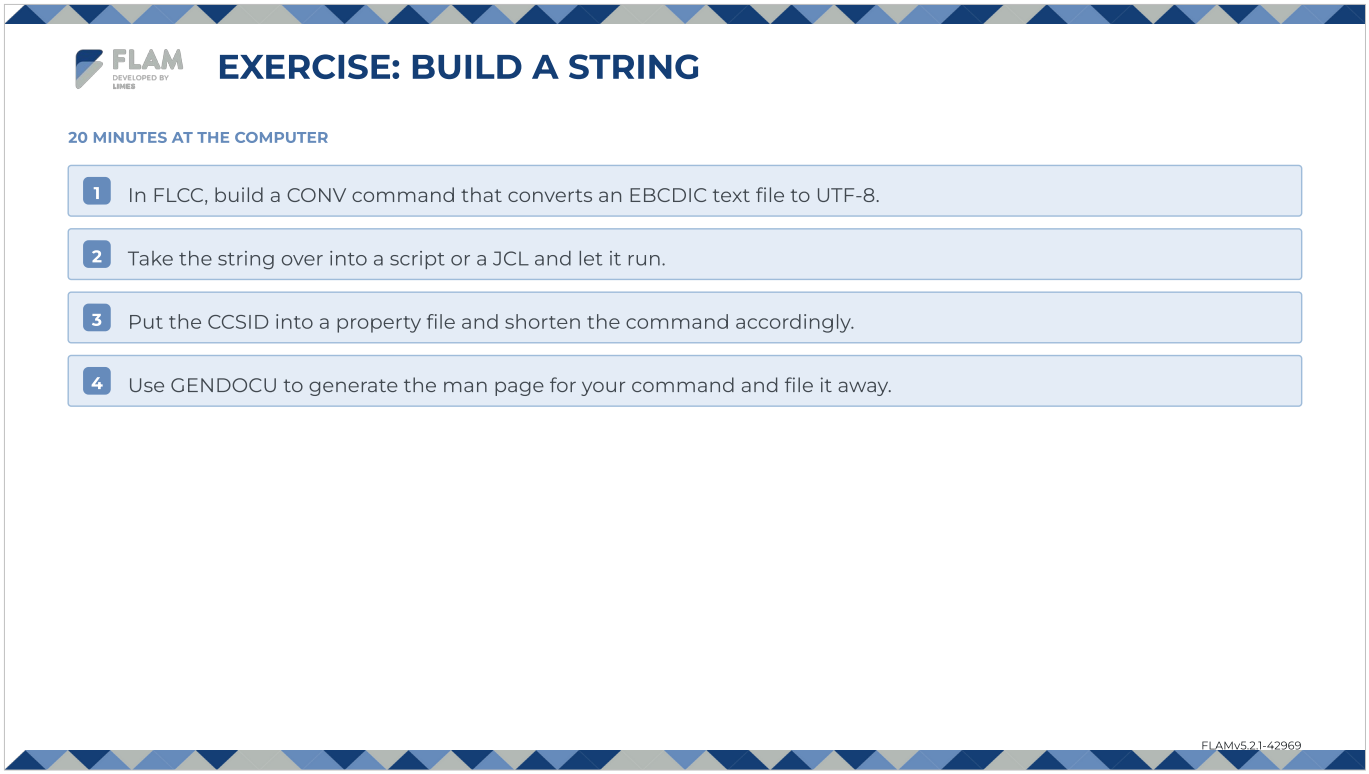
They are only needed if the string contains syntax elements – brackets, spaces, operators. And there are several of them: `'`, `"` and `` ` ``. In shells, take the character that does not interfere there – or leave them out.

! **flcc is the GUI, flcl is the command line.**

For scripts, tests and JCL always use flcl - flcc is for building and looking up.

FLAMv5.21-42969

All of them from support cases. The first one costs the most tickets.
On the quotes, because this comes up in every shell discussion: they are **only needed when the string itself contains syntax elements** -- that is, parentheses, blanks, operators. Otherwise you can leave them out, and a string may start directly after the equals sign; it then ends at the next separator.
And there is not just the single quote: `` `' ``, `` `" `` and the backtick are all valid. That is handy, because in Bash, PowerShell and JCL it is a different character that causes trouble in each -- you simply take the one that does not get in the way there. Look it up in `flcl lexemes`.



Twenty minutes at the machine. Anyone without FLCC works with `flcl HELP`.
The aim is the moment when the string they built themselves runs -- after that the inhibition is gone.
Solution:
1. In FLCC, pick the command, expand the required objects in the parameter tree and set the values. Whatever is inactive does not belong to the chosen overlay -- that is the teaching aid.
2. The command-line view yields the string. Switch to the structured view for reading, to "as JCL" for the mainframe.
3. The same string runs in the shell, in the JCL and as the first parameter of an `fcbopen`. That is the phrase "one parameter string, three routes" in practice -- and the reason why the FLCL manual is also the API manual.
4. If you want to check whether the string is correct without running it: `NORUN` in the command. The parser then checks the syntax and exits.

WHAT TO TAKE AWAY

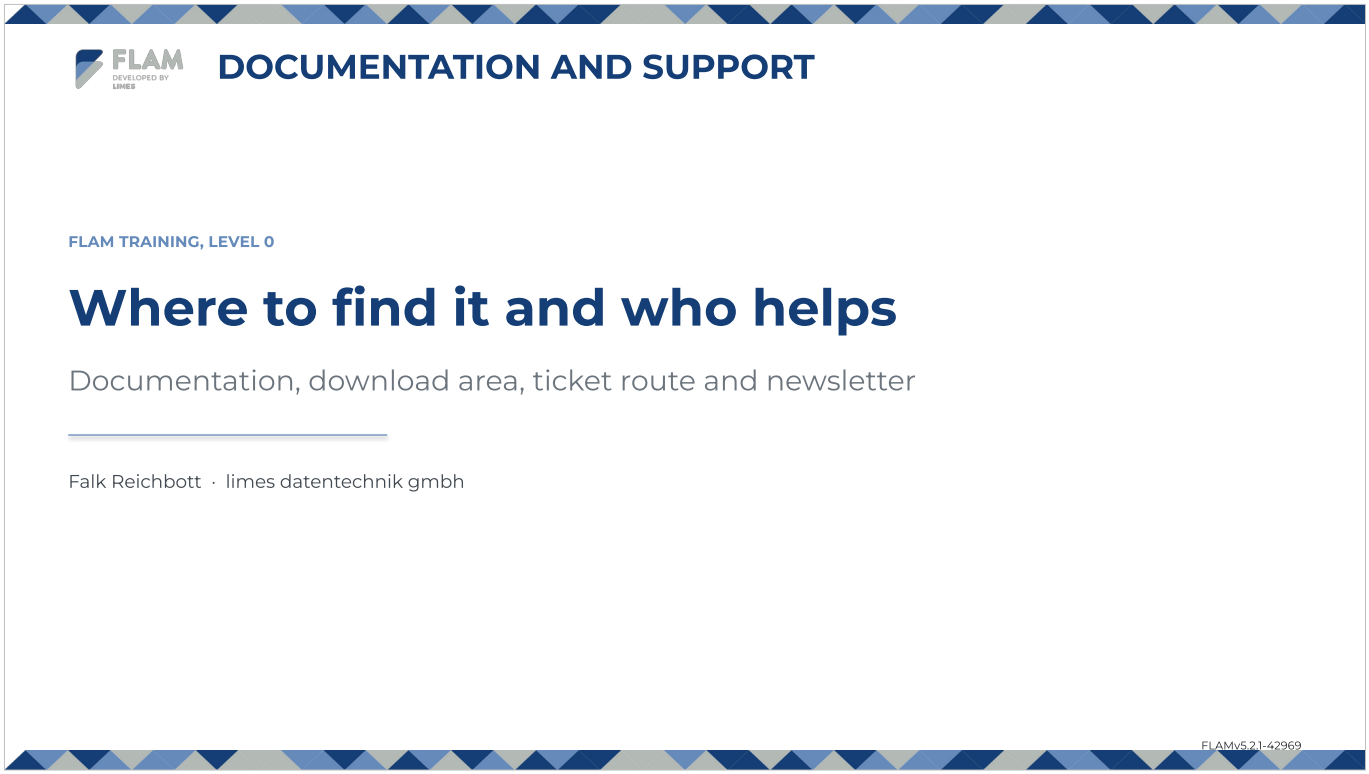
- 15 commands, one language - the volume lies in the parameters
- CLE/P: command, object, overlay, array. Whoever reads that reads every string
- the software generates its manual itself; HELP goes down to the parameter
- FLCC is the interactive manual and supplies the string for all three routes
- presetting has an order - it is the most frequent source of errors

FURTHER READING

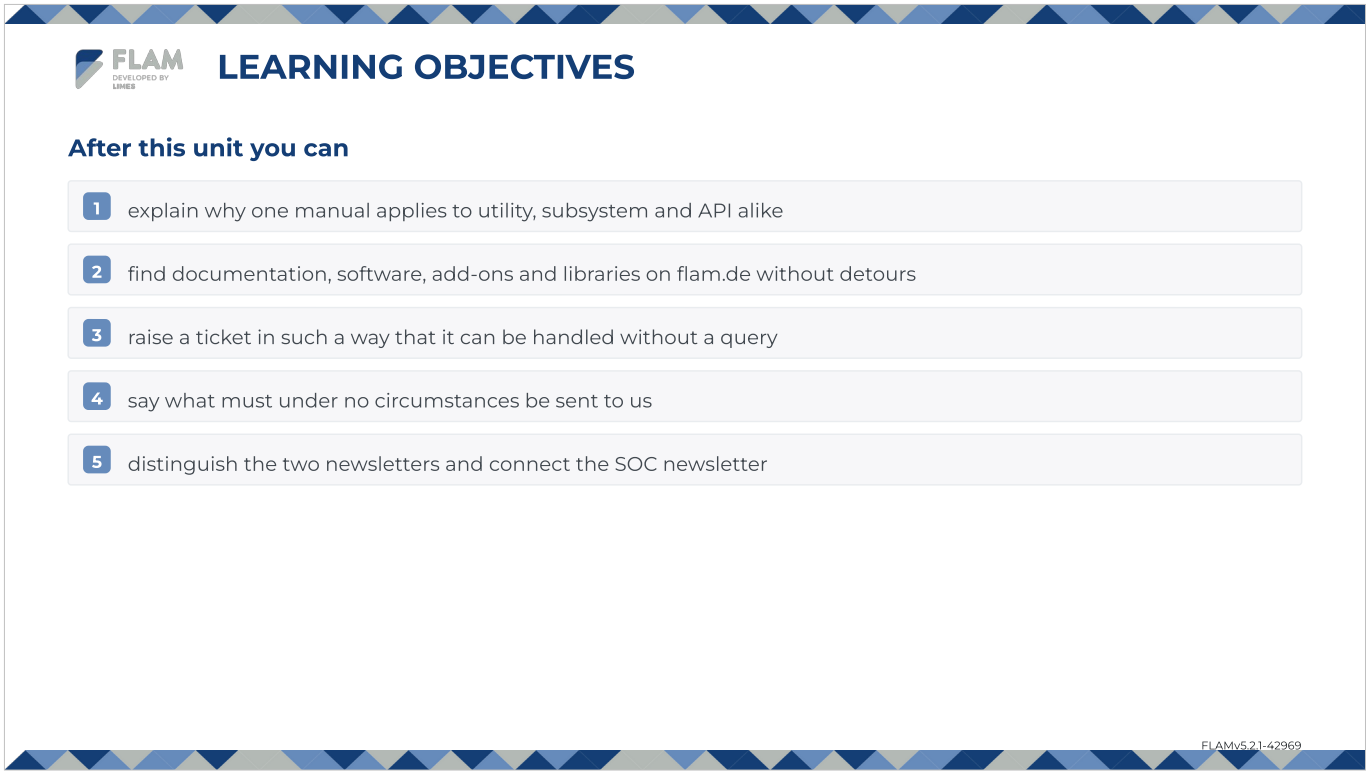
FLCL manual	flam.de/downloads/documentation
CLE/P sources and API	our GitHub area
Help in the tool	flcl HELP, MANPAGE, GENDOCU, SYNTAX, GRAMMAR, LEXEMES

FLAMv5.21-42969

Go through the learning objectives. Next comes documentation and support -- that is, where all of this lives and how a ticket reaches us.



Third unit of level 0. Unspectacular, but the one with the highest practical value after the training: where do I find it, and whom do I ask.
Do not schedule this unit at the end of the day under any circumstances -- it holds the information the participants take away and pass on.



Objective two is the one that counts day to day. Objective four prevents a data protection incident -- that is not a formality.

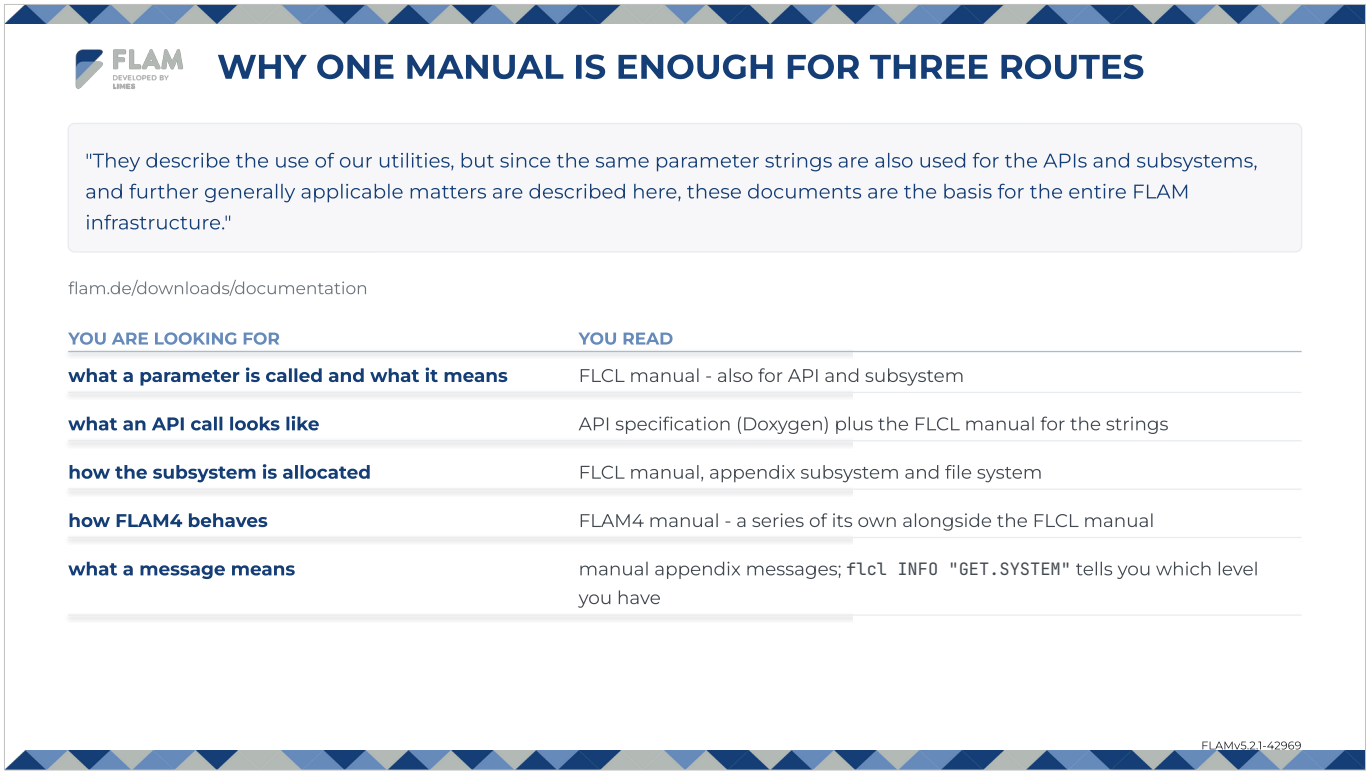
THE SOFTWARE GENERATES ITS MANUAL ITSELF

- the manual is generated from the same parameter table from which the command parser is also built
- so the documentation always agrees with the software level
- the documentation is **part of the installation** - in case of doubt the installed one applies, not the one read online
- online there is always only the newest version
- you can generate parts of it yourself: `flcl GENDOCU`

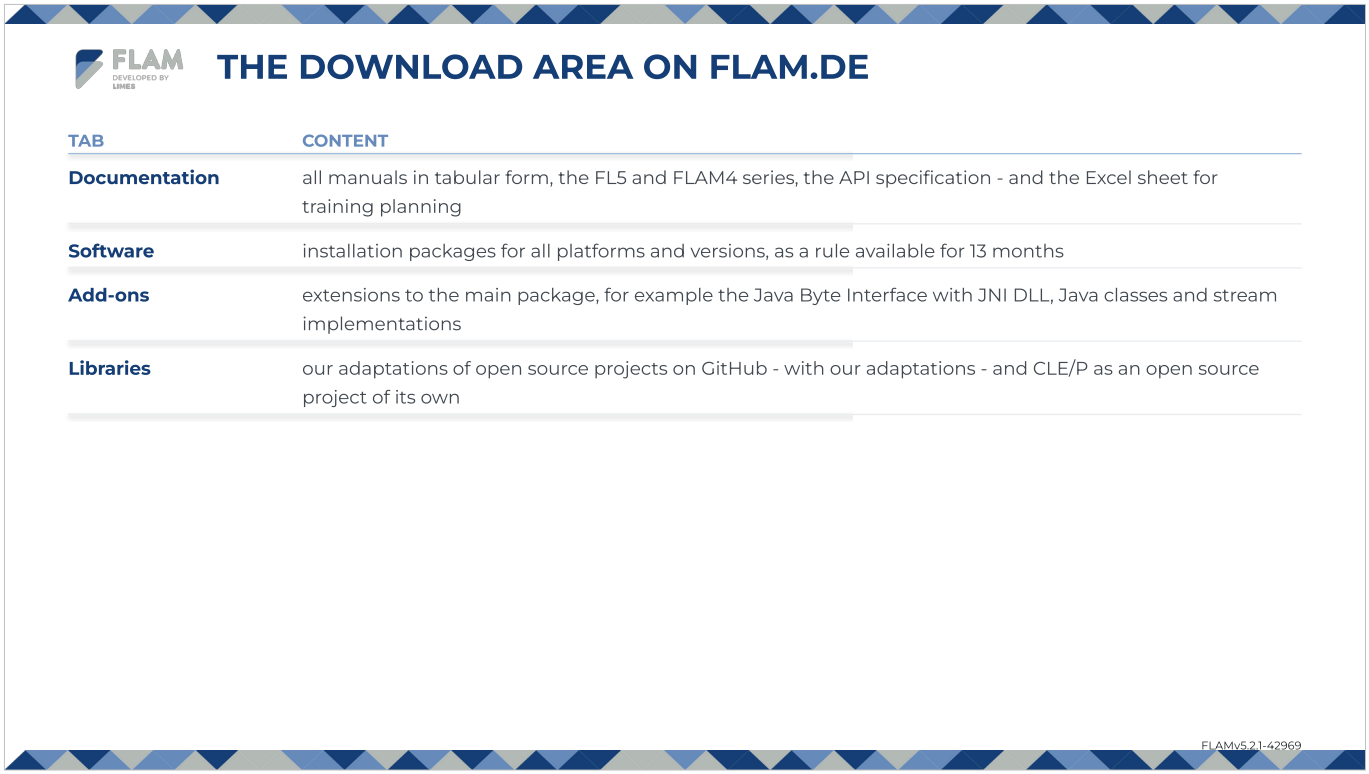
A manual that the software writes itself cannot go out of date - but it describes the language, not your use case. That is what the training is for.

FLAMv5.21-42969

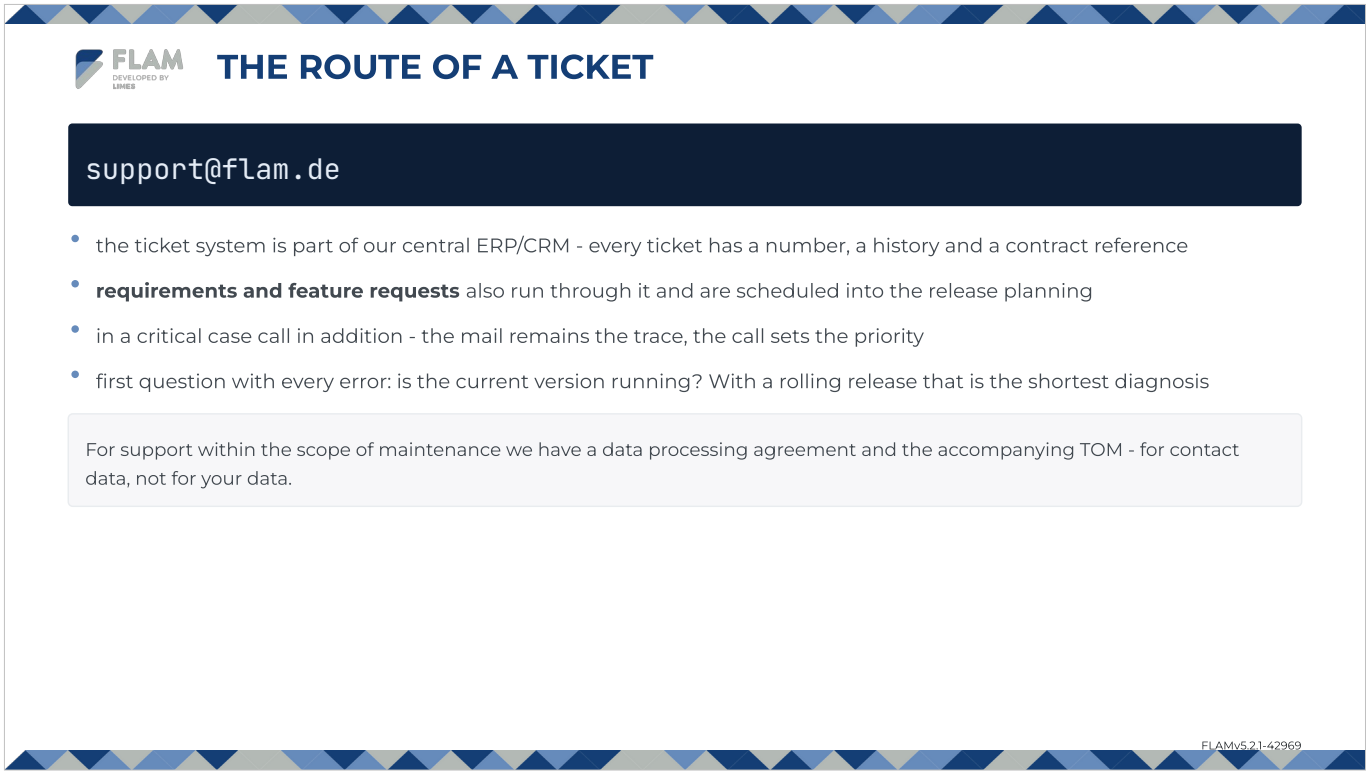
This is the reason why our documentation does not go stale -- and why it sometimes sounds dry: it comes from the same table from which the parser is built.
Practical consequence for operations: the documentation in the installation package is always the right one. Online, only the latest version is available.



Verbatim from flam.de: "They describe the use of our utilities, but since the same parameter strings are also used for the APIs and subsystems ... these documents are the basis for the entire FLAM infrastructure." That is the same statement as in the fundamentals, but here from the point of view of looking things up. Developers in the room, in our experience, look for an "API manual" and do not find it -- because it is the FLCL manual.



Four tabs, and the participants usually know only one. "Libraries" in particular is unknown -- that is where our adaptations to external libraries live, and CLE/P as its own open-source project.
The Excel sheet for training planning is right at the top under Documentation -- point it out, it is the entry point for the next training course.



The central sentence: the ticket system is part of our ERP/CRM, it is not a mailbox on the side. So every ticket has a number, a history and a link to the maintenance contract.
Feature requests go the same route -- that regularly surprises people and is good news: requirements from operations end up in release planning, not on a wish list.

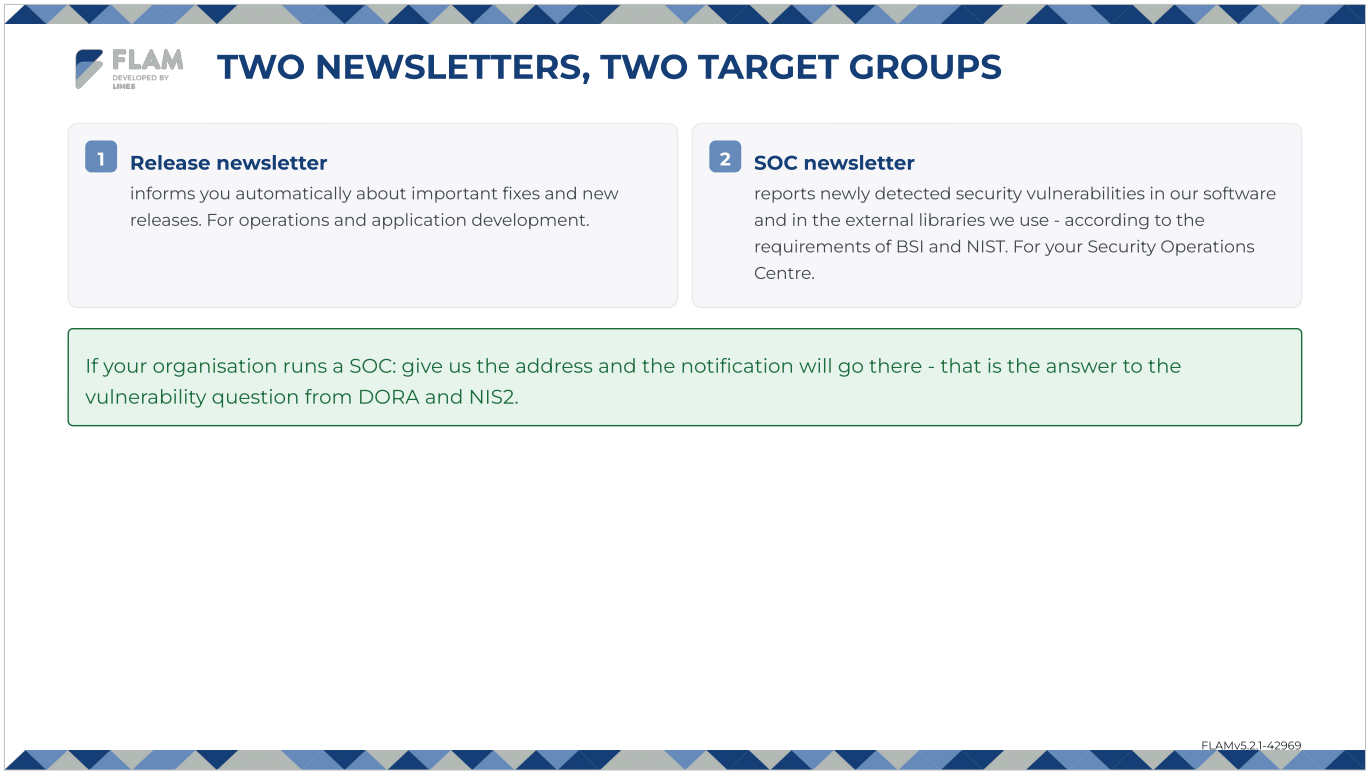
WHAT BELONGS IN A TICKET - AND WHAT DOES NOT

PLEASE SEND WITH IT	PLEASE DO NOT SEND
version and platform (`flcl INFO GET.SYSTEM`)	dumps from production operation
the complete parameter string	production customer data, not even "just an extract"
the message in the original, not typed out again	screenshots instead of text where text is possible
an anonymised minimal example	access credentials, keys, keysets
what you expected and what happened	

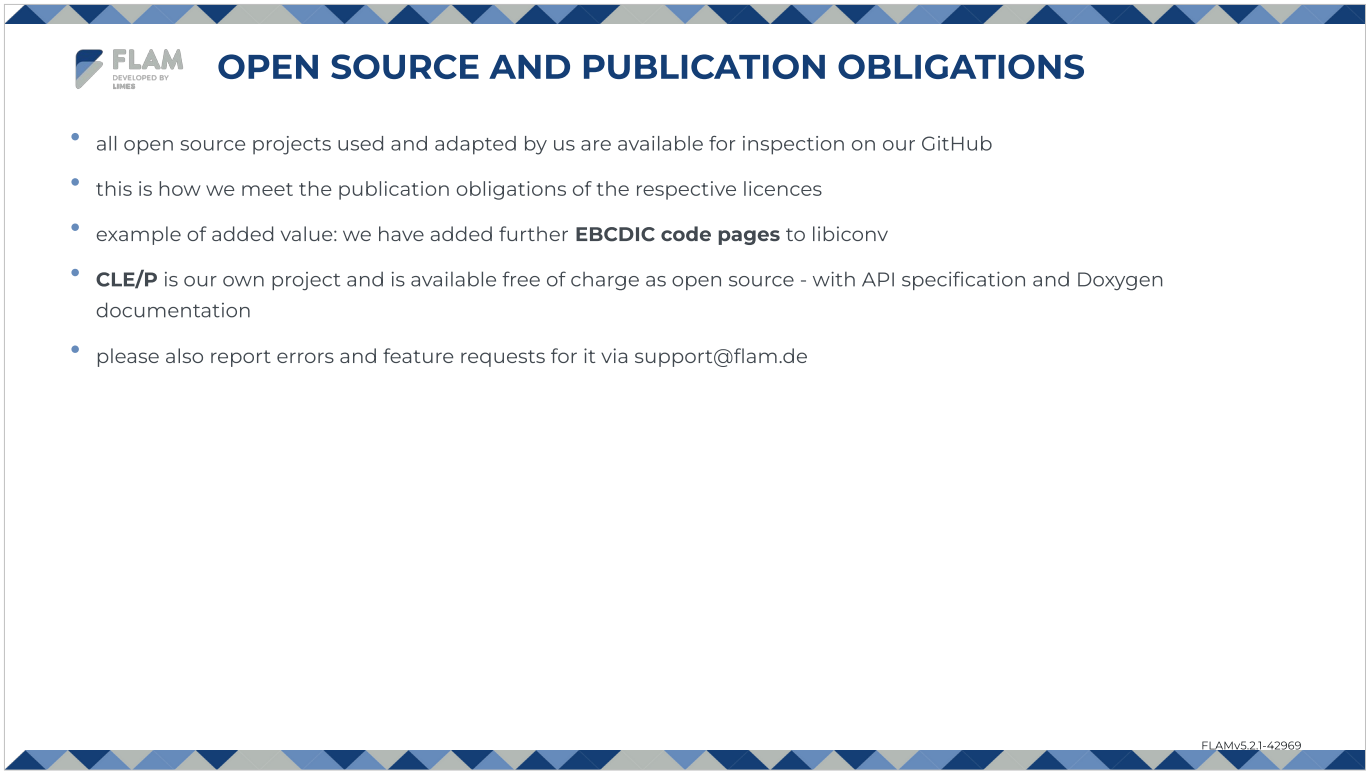
Whoever sends us sensitive data has already compromised it at the moment of sending. Please no dumps and no production customer data.

FLAMv5.21-42969

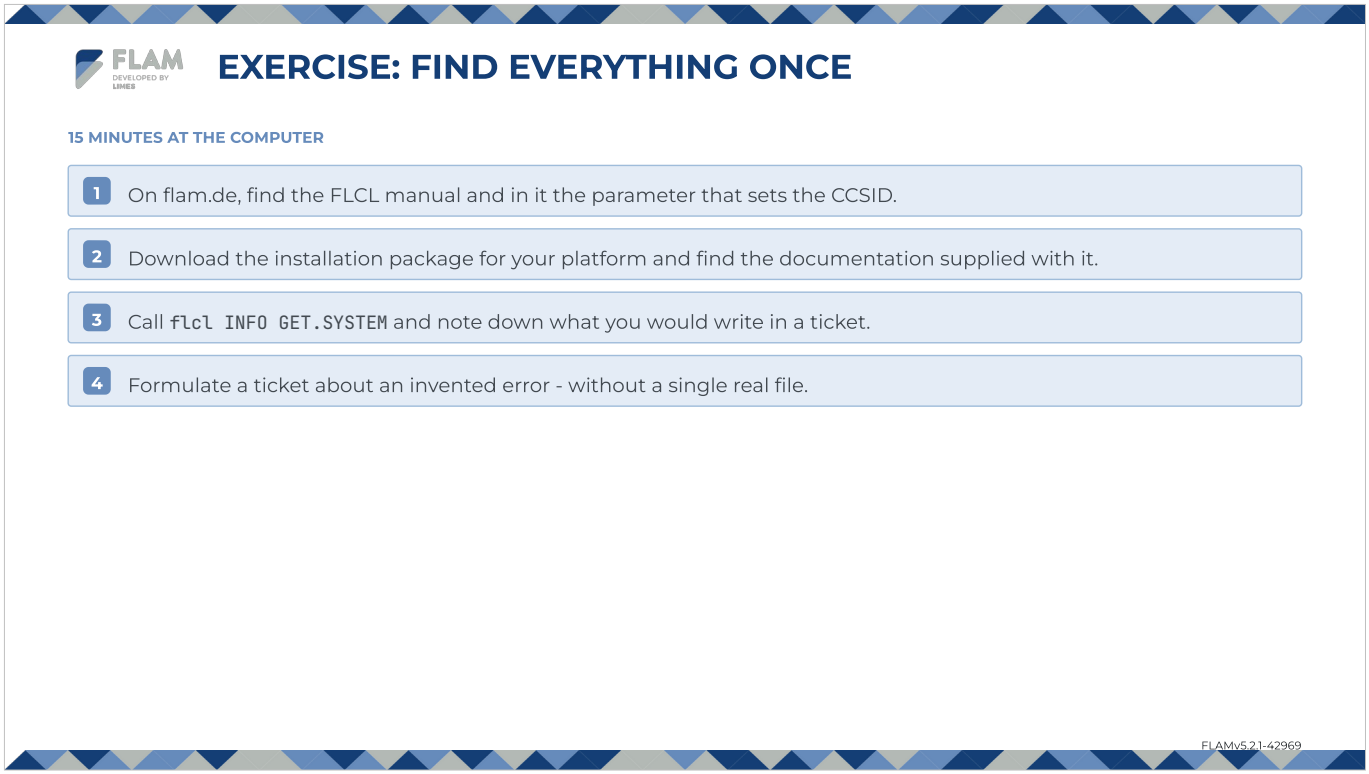
The left column saves a round of follow-up questions, the right one prevents a data protection incident.
Say the last point plainly, it is uncomfortable and important: anyone who sends sensitive data to us has already compromised it the moment they send it. We are not a data centre and have no access to customer systems.
For error reports, the version, the parameter string, the message and an anonymised minimal example are almost always enough.



The second one is the one hardly anyone knows and that every regulated organisation needs: the SOC newsletter reports newly discovered security vulnerabilities in our software **and in the external libraries we use**, following the requirements of BSI and NIST.
That is the answer to the DORA and NIS2 question "how do we learn about vulnerabilities at your end". Offer concretely to add the SOC address.



Two messages. First: we meet the obligations of the open-source licences and disclose our adaptations -- verifiable on GitHub. Second: a part of that is added value for the customer, for instance the EBCDIC code pages in libiconv, which the original there does not have.
CLE/P is our own project and free to use; anyone who needs their own cross-platform command line can take it.



Fifteen minutes at the machine. The effect is large and the effort small: once you have walked the route, you walk it again in daily work.
With exercise 4, make sure that nobody really attaches real data -- that is the learning point.
Solution:
1. The parameter is called `CCSID` and appears in every read and write object.
Faster than the manual is `flcl HELP CONV.read.text.ccsid`.
2. The documentation is in the package, not next to it -- that is the learning point: it therefore matches the installed version, even when a newer one is already online.
3. A ticket needs version and build, platform, the active licence and the effective environment variables -- that is, exactly the output of `flcl INFO GET.SYSTEM`. That saves the first round of follow-up questions.
4. What is expected: what was the intention, what was the call, what came out, what should have come out -- plus the INFO output. And no real data. If you need test data, construct it; anyone who attaches production data has created a data protection problem in order to report a software problem.

WHAT TO TAKE AWAY

- the software writes its manual itself - the installed documentation applies
- one manual for three routes, because the parameter string is the same
- four tabs in the download area: documentation, software, add-ons, libraries
- tickets to support@flam.de - with version, string and message, without production data
- two newsletters: releases for operations, vulnerabilities for the SOC

FURTHER READING

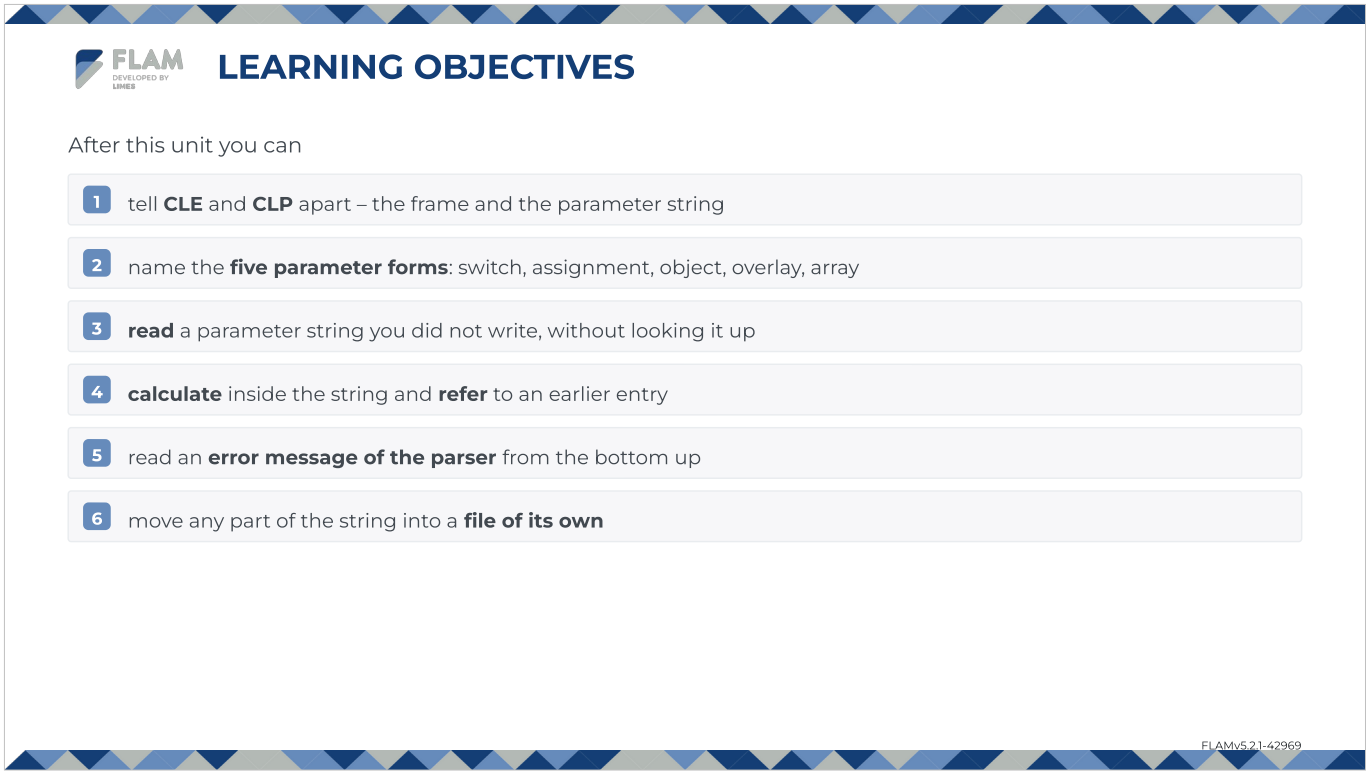
Support and maintenance	flam.de/support
Manuals and training sheet	flam.de/downloads/documentation
Software and licences	flam.de/downloads/software
Open source and CLE/P	flam.de/downloads/libraries

FLAMv5.21-42969

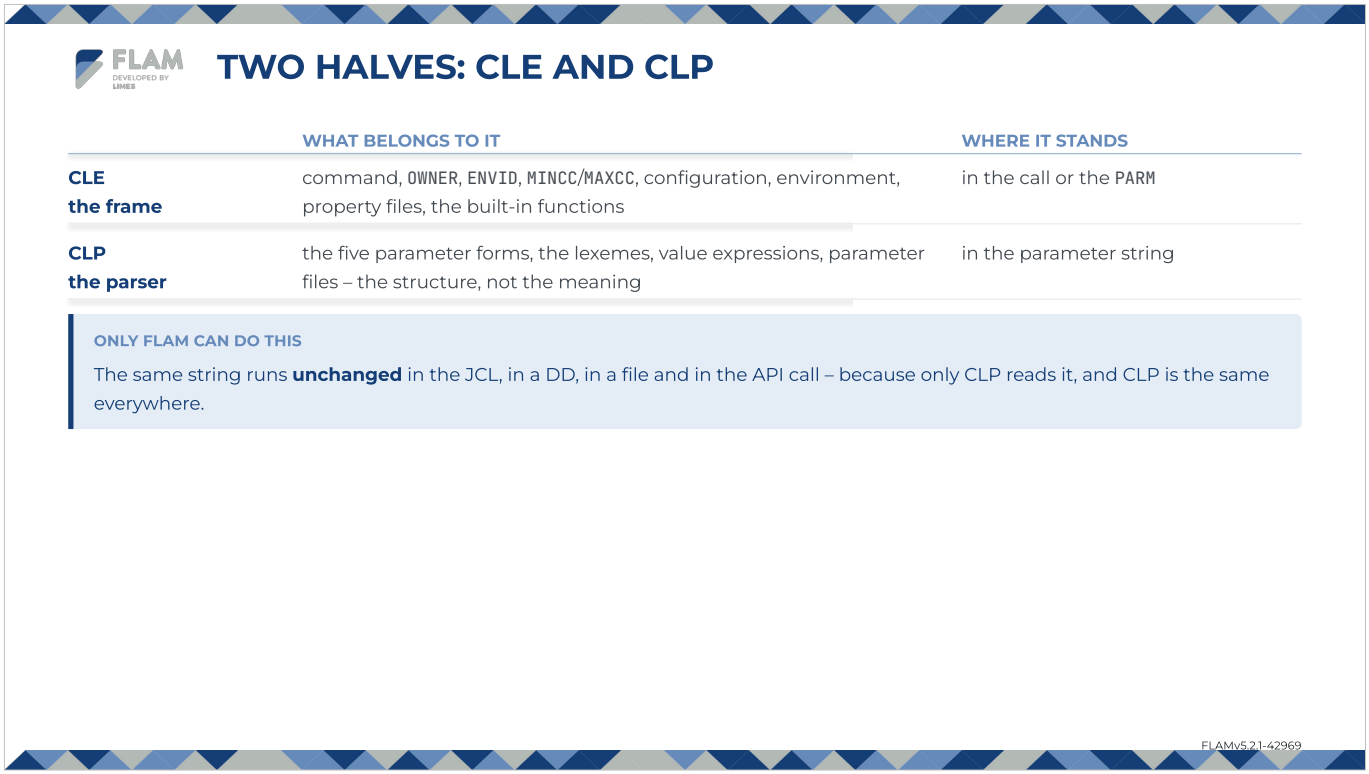
Go through the learning objectives. That completes level 0; next come the three access routes in detail.



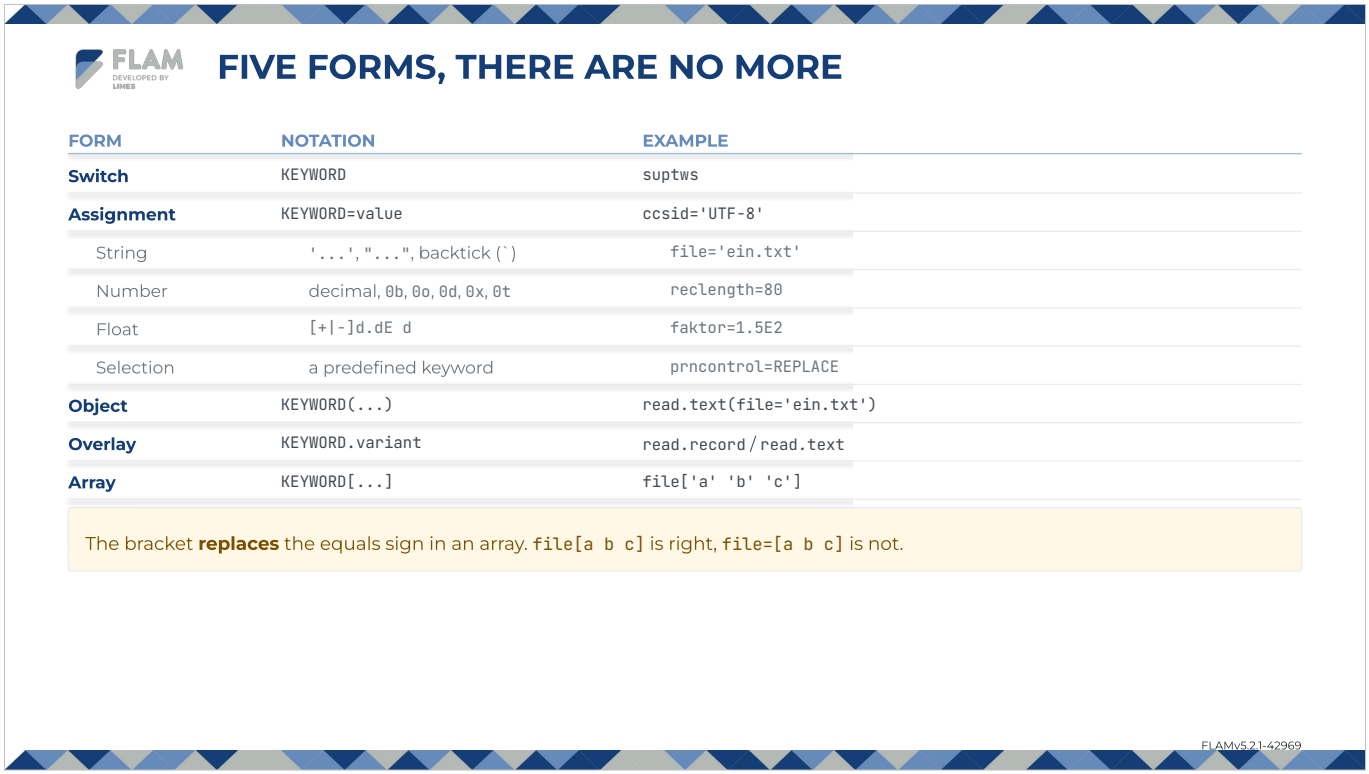
This unit is the reference on the language. It is rarely presented from front to back -- you show it once and look things up in it later. The arc: the two halves CLE and CLP, the five parameter forms, the lexemes, arithmetic inside the string, parameter files, and finally the error output of the parser. The sentence that justifies the unit: **every parameter string in FLAM is written in the same language** -- in the JCL, in the shell, in the subsystem, in the API call. Read it once and you read it everywhere. The unit *Tools: FLCL and FLCC* is assumed; that is where the vocabulary list and the built-in help are.



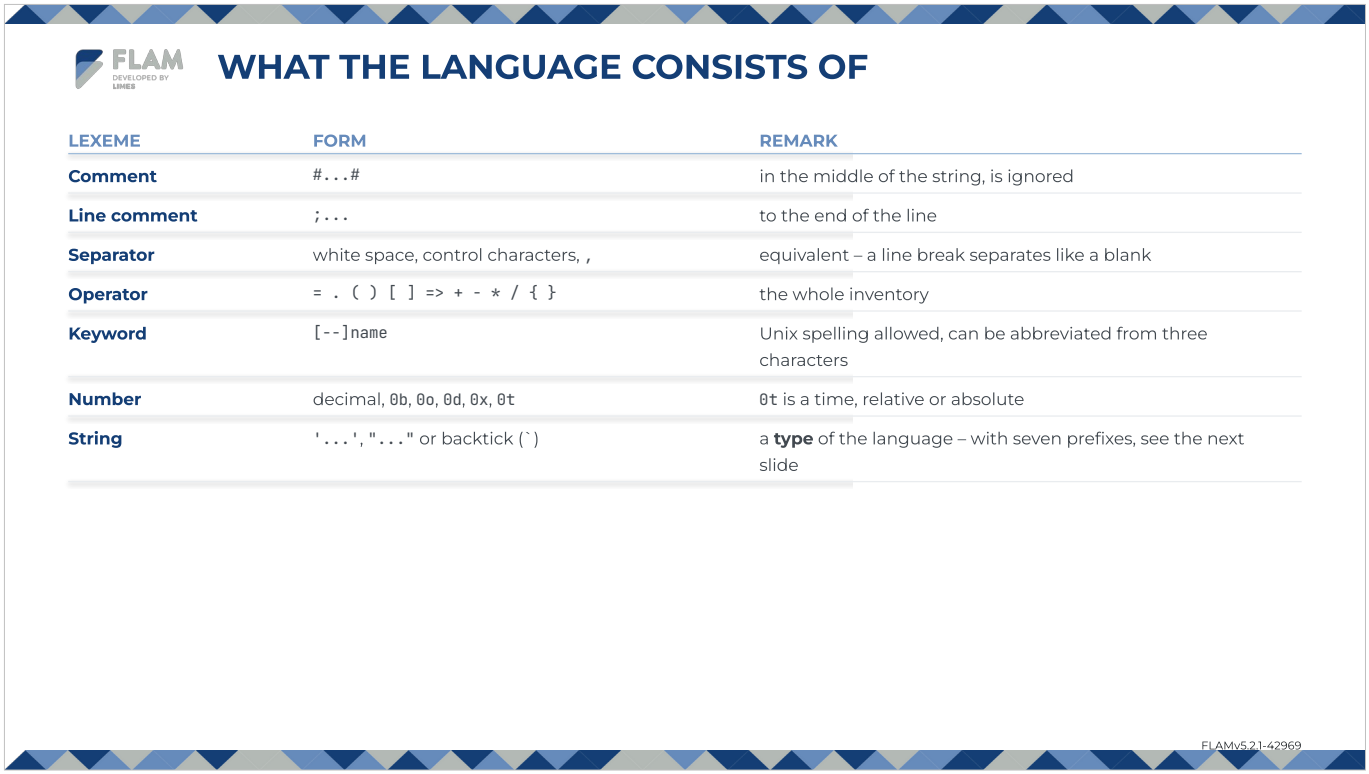
The third objective is the one it is really about: reading a string you did not write, without looking it up. The fifth is the point that saves time in production -- an error message of the parser says exactly where it stopped and what it understood up to there. Whoever has seen that once stops searching.



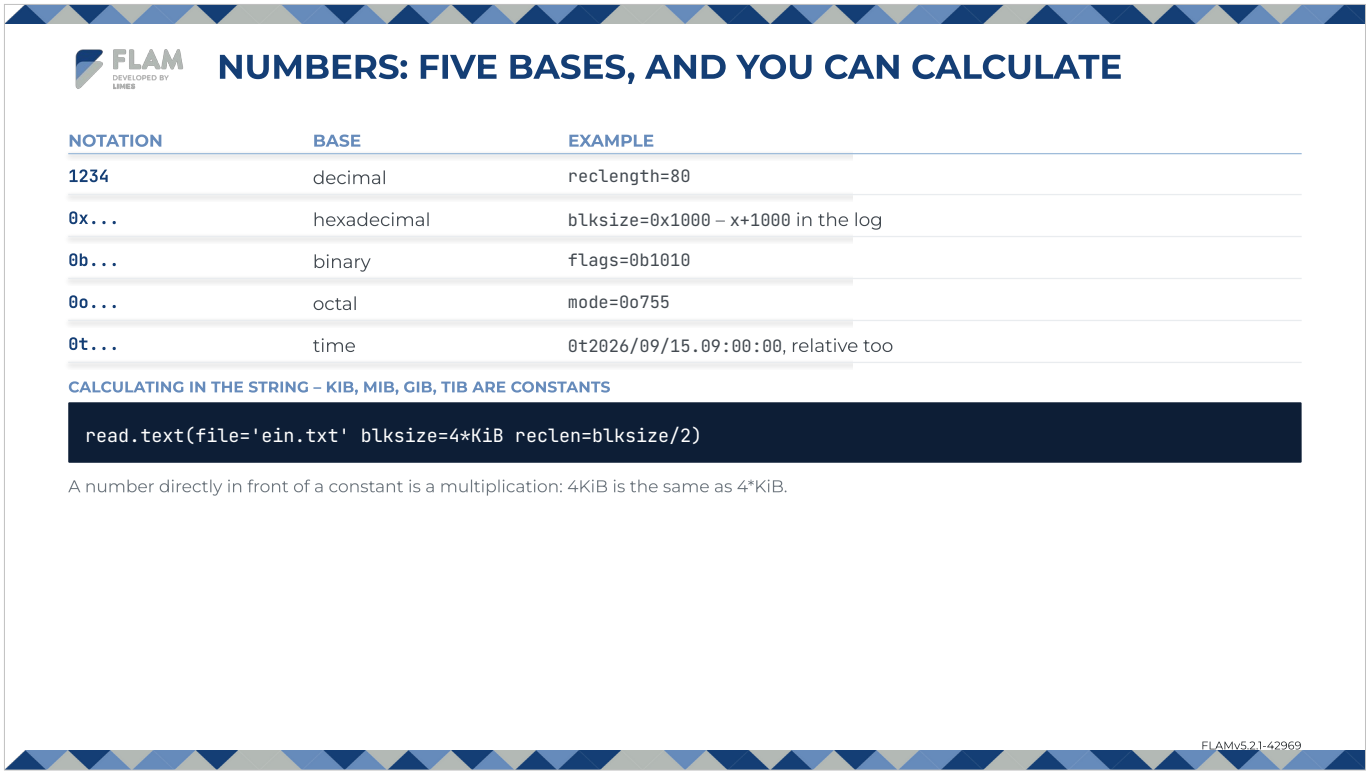
The separation explains why some things stand at the command and others in the string -- a question that comes up in every training course. **CLE** is the frame around the call: which command runs, under which owner, with which environment and which return-code window. The built-in functions belong to it too -- `HELP`, `SYNTAX`, `MANPAGE`, `GENDOCU`, `GENPROP`, `LEXEMES`, `GRAMMAR`. **CLP** is the parser for the parameter string. It knows the forms, the lexemes and the value expressions, and it knows the structure of the commands -- but not their meaning. The practical use of the separation: `MAXCC` and `OWNER` stand in the PARM, because they concern CLE. Everything that describes the processing stands in the string, because it concerns CLP. That is why the same string goes unchanged into the JCL, into a DD, into a file or into an API call. And both are open source -- CLE/P is on our GitHub and is used by other tools.



This is the most important slide of the unit. Whoever knows the five forms can take any string apart -- even one they have never seen. Go through it slowly and show the example on every line: A **switch** is a keyword on its own. It says yes without a value. An **assignment** is keyword, equals sign, value -- and behind it stand the **scalar types** of the language, which have to be kept apart because the parser keeps them apart: a **string** in quotation marks, a **number** in decimal or in one of the other bases, a **float** with a point and an exponent, and the **selection** -- a predefined keyword such as `REPLACE` that looks like a keyword and is a value. The switch is the fifth scalar type, but it has no assignment, which is why it stands in the table above. An **object** groups parameters in parentheses. It is the form that makes the strings long -- not because the language is complicated but because it is complete. An **overlay** is a choice: exactly one of several variants, and the dot separates. `read.text` and `read.record` are two variants of the same overlay. An **array** takes several values. The bracket replaces the equals sign -- `file[a b c]`, not `file=[a b c]`. There is a form without brackets too, but then the end can no longer be recognised for certain; the manual explicitly recommends the enclosed form, and so do we. The rest of the language -- lexemes, arithmetic, files -- hangs on these five alone.



The reading aid for everything that stands between the five forms.
Do not read it out -- showing three lines is enough, the rest is for looking up. The heading is deliberate: what stands here is what the **language** consists of. The string is one of its **types** and appears in the table as a row -- the table does not describe the string, the string is an entry in it. And a small thing about it that is hardly known: a string may also be enclosed in backticks, not only in `'` or `"` -- that is how it stands in `flcl lexemes`, and on a keyboard without quotation marks it is worth gold. That is why the backtick is written out on the slide: as a character on its own it is easily overlooked. Two things out of it are genuinely needed often: **Comments.** `#...#` in the middle of the string, `;` to the end of the line. That is why a parameter file moved out of the JCL stays readable -- you write down why an entry is there. **Separators.** White space, control characters and the comma are equivalent. A line break therefore separates just like a blank, and that is why a string in the JCL may run over any number of lines. And a small thing that pleases newcomers: a keyword may start with `-` or `--`. Anyone coming from the Unix world can write `--file=...`. The short form applies everywhere:
three characters are enough as long as they are unambiguous -- but not in scripts, because a new keyword can remove that uniqueness tomorrow. That concerns not only parameters: **commands, functions and objects may be abbreviated just as well.** `CNV` is `CONV`, `SAV.FIL` is `SAVE.FILE`, `CMP` is `COMPRESS`. Our manual examples use both spellings, and whoever does not know the rule takes that for two different objects. The advice stays the same: write it out in a script, abbreviate on the command line.



The part that makes a JCL readable. A number can be written in five bases, and the log shows the base again afterwards:
`blksize=0x1000` appears as `x+1000`. For sizes that are thought of in powers of two anyway, the hexadecimal spelling is closer to the truth than the decimal number.
More important is the arithmetic: `*4KiB*` *is valid and stands for 4096. That reads markedly better in the JCL, and you see the intention instead of a number you have to work out. The same goes for times --* `*NOW-DAY*` *is yesterday. And the multiplication may be left out: a number directly in front of a constant is a multiplication, so* `*4KiB*` *is the same as* `4KiB`. Measured: `blksize=*4KiB*` *gives 4096.*

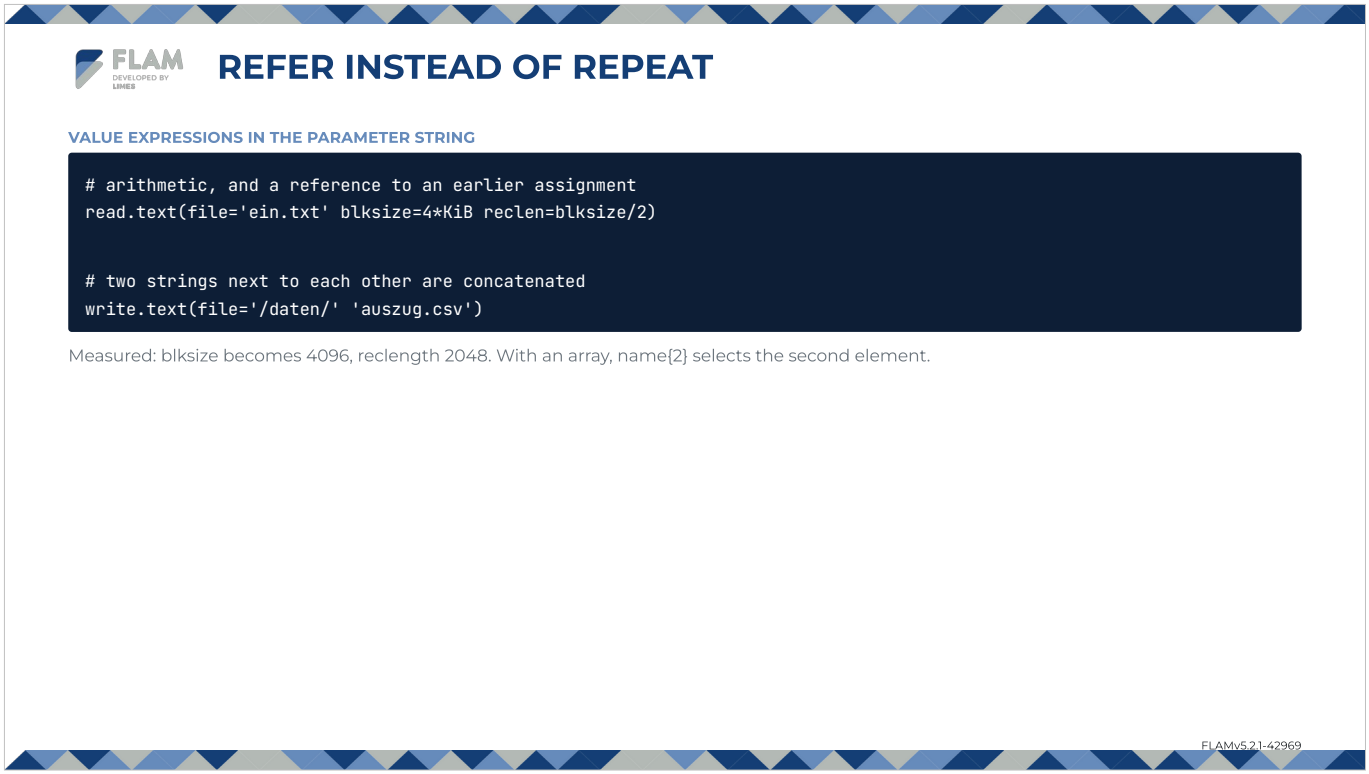
STRINGS: SEVEN PREFIXES

PREFIX	MEANING	WHAT FOR
`'...'`	normal case	text, file names, CCSIDs
`x'...'`	bytes in hexadecimal notation	padding and delimiter characters – `padcharacter=x'40'`
`a'...' e'...'`	binary string in ASCII or EBCDIC	when the encoding of the value has to be fixed
`c'...'`	binary string in the local character set	the default for binary entries
`s'...'`	null-terminated	for interfaces that expect it
`f'...'`	**read the value out of a file**	**passwords** – none of it stands in the call

Two '' are one '. Instead of ' you can use " and the backquote. For a password the log shows `***SECRET***`.

FLAMv5.21-42969

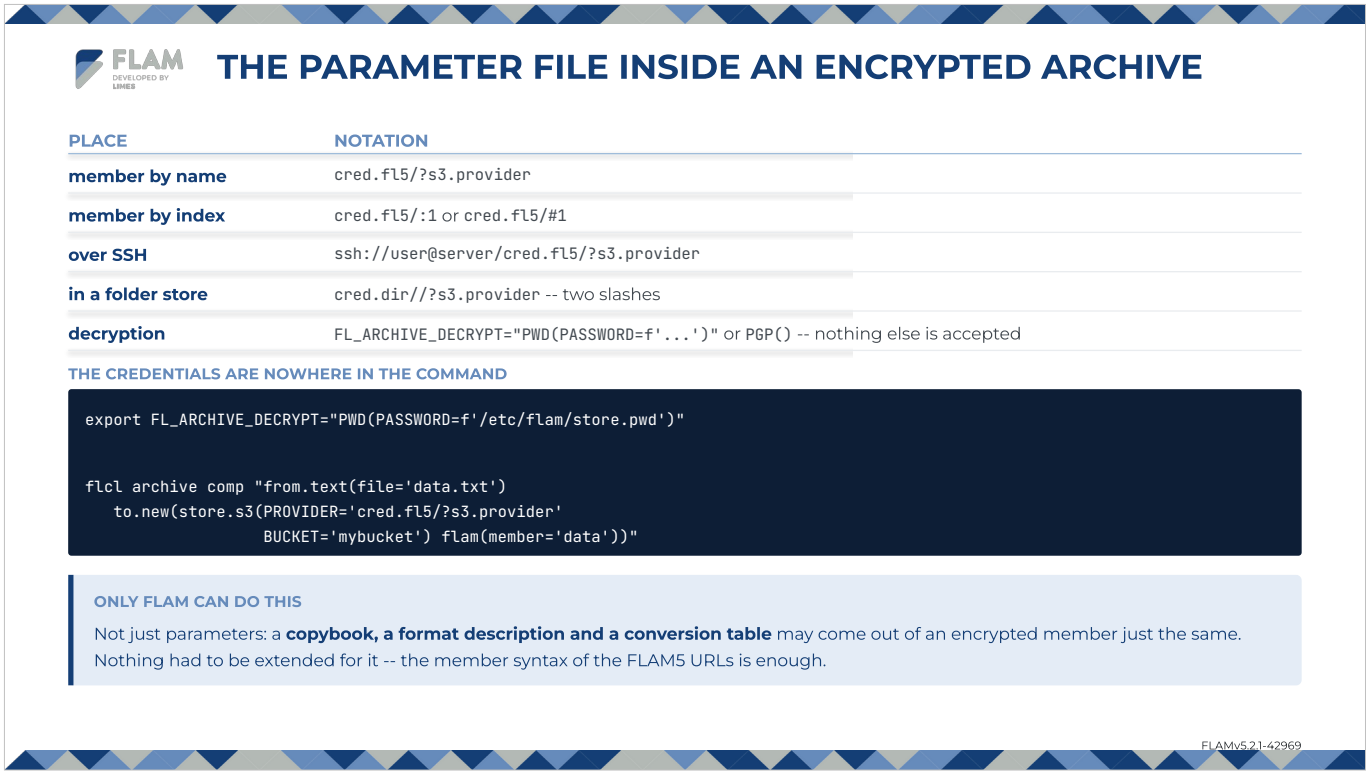
The slide that holds something hardly anybody knows and that is worth gold in payment traffic: **`f'...'` reads the value out of a file.** First the row: without a prefix it is the normal case. `s'...'` is null-terminated, `c'...'` a binary string in the local character set, `a'...'` in ASCII, `e'...'` in EBCDIC, `x'...'` in hexadecimal notation. That is needed when a value is a byte and not text: measured, `padcharacter=x'40'` gives the EBCDIC blank, and the parser reports it as `EBCDIC-BLANK(x'40)`. And then `f'...'`: the content of the file becomes the value. With that a password stands **not** in the parameter string, not in the JCL and not in the process list -- it stands in a file that can be protected with file permissions. The log shows `***SECRET***` *instead of the value.* Measured with `password=f'/pfad/pw.txt'`. Finally the handling:
two apostrophes in a row are one apostrophe. Instead of `'` you can use `"` and the backquote -- handy when the shell has used one kind up already. And directly behind the equals sign a string may stand without quotes; it then ends at the next separator, and keywords take precedence. In the course it is worth saying: readable it is, safe it is not -- whoever has a path with special characters puts quotes around it.



A slide for those who maintain long strings. A value may be an expression: `+ - /` *and parentheses. And -- this is the part almost nobody knows -- a keyword in a value means* **the value of an earlier assignment in the same string**. Measured: `blksize=4*KiB reclen=blksize/2` *gives 4096 and 2048. The halving therefore stands there once, and whoever changes the block size changes it in one place. With an array the index goes in braces:* `name{2}`*. For strings the same holds in a simpler form: two strings next to each other are joined, with or without* `+`*. That lets a file name be assembled from a prefix and a name without writing it twice. The advice for practice: use it sparingly and comment it. A reference the next person does not see is harder to read than a repeated number.*



The slide with which an unreadable JCL is tidied up. Not only the whole parameter string can stand in a file -- **every object, every overlay and every array on its own**. That turns one long line into a short one with references. And now the distinction that really has to be mastered, because it looks small and means three different things: **For an object or an overlay `=` and `=>` are the same** -- both say "the content stands in this file". That is how it stands in the grammar (`flcl grammar`), and that is why the spelling is a matter of taste here. **For a string they are different.** `file='DD:PIN'` assigns the string `DD:PIN`. `file=>'DD:PIN'` reads the **file** DD:PIN and makes an entry out of **every line** -- so with a string array a file list, one line per name or wildcard. Measured: a file with two lines yields two `FILE=` entries. **And `file=f'DD:PIN'` is something else again:** there the **whole content** of the file becomes one single string. That is the form for passphrases -- and the difference to `=>` is exactly the one between "one line per element" and "everything one value". What it brings is three things: The file may hold **comments** -- `;` to the end of the line and `#...#` in the middle of the text. In a JCL line you cannot explain why `PRNCONTROL=REPLACE` is there; in a parameter file you can. The file is **reusable**. The same read side can be used by several jobs, and when the format changes the file changes -- not the JCL. And the file may **lie elsewhere**: a DD name on z/OS, a path, or a URL. With a URL you give the CCSID along, and then the file is re-encoded while it is read -- so a parameter file on a Unix system can be used by a host job. Measured with a file holding comments and three entries.

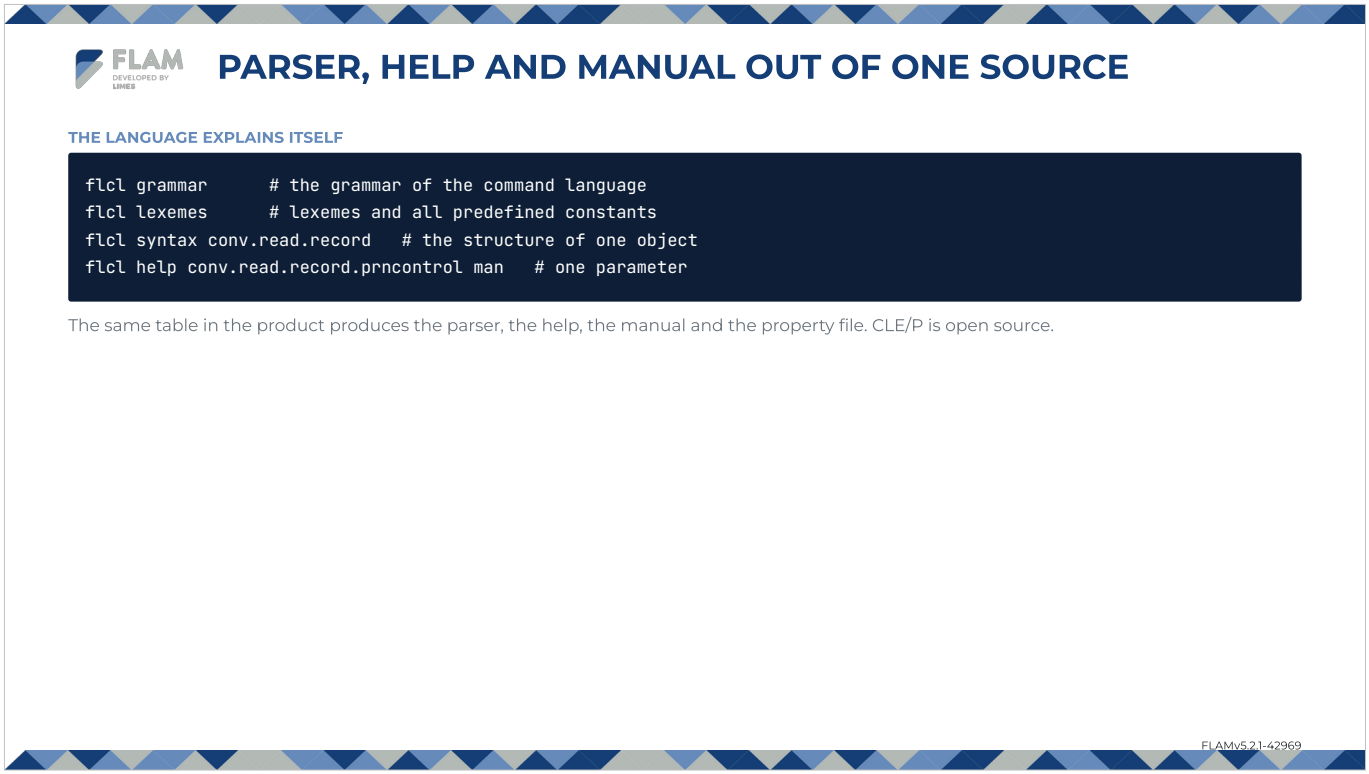


The slide that turns the previous one into a security function -- and it is new. The problem first: a command carries entries that not everyone should read -- the keys of an object store, a password, the address of an internal service. In the command they stand in the job log and in the process list. In an ordinary parameter file they are out of the command, but in the file system. The answer is one sentence: **the parameter file may be a member of a FLAM5 archive, and the archive may be encrypted.** The archive is then a credential store -- the secrets lie encrypted, the command names a member only, and neither the command nor the job log nor the process list shows the content. And the good part: **nothing at the CLP had to be extended for it.** The member syntax that FLAM5 URLs already have is enough -- `archive/?name` by name, `archive/:1` by index. The archive may be remote, over SSH or in a folder store; with a folder store the second slash stands there and the member begins after it. Decryption comes from the environment variable `FL_ARCHIVE_DECRYPT`, on z/OS also as the system symbol `&FLARCDC`. It carries the DECRYPT entry and nothing else: `PWD(PASSWORD=...)` or `PGP()`. **Why only those two?** That is the part worth telling in the room, because it shows how this is thought through: the parameter object around DECRYPT also carries PREPROCESS and POSTPROCESS, and those **execute commands**. A variable that reached as far as those would turn every read of a parameter file into a way of starting programs. For the same reason, naming a crypto engine inside PGP is refused -- an engine is a module that would be loaded and executed. The password does not have to stand in the variable either: `PASSWORD=f'...'` fetches it from a file with its own permissions. And on z/OS one often needs none at all -- if the archive is encrypted for an OpenPGP key, the private key lies in ICSF and there is no password that could stand anywhere. And the part that is often underestimated: **this is not only about parameters.** Reading goes through the file-to-string layer, and that serves every text input -- so a **COBOL copybook**, a PL/I include, an assembler structure, a **format description** or a **conversion table** as well. The description of your data can lie encrypted too, not only the password to it. Measured: a row specification out of an encrypted member, conversion runs through. One boundary that comes up at once in a conversation: **`FL_ARCHIVE_DECRYPT` is not the decryption of the archive a command processes.** A `DECO` of an encrypted archive takes its `DECRYPT()` from the command. The variable applies only to an archive an **input is read from**. One property to know: **the member entry is the decision.** A name without a member is an ordinary file. A name with a member is an archive -- and if it cannot be opened, the member is missing or the decryption fails, **the command stops**. It does not fall back to reading the file raw, because then the bytes of an encrypted archive would land in the parser and the message would be a syntax error somewhere in binary data instead of "member not found" or "password wrong".

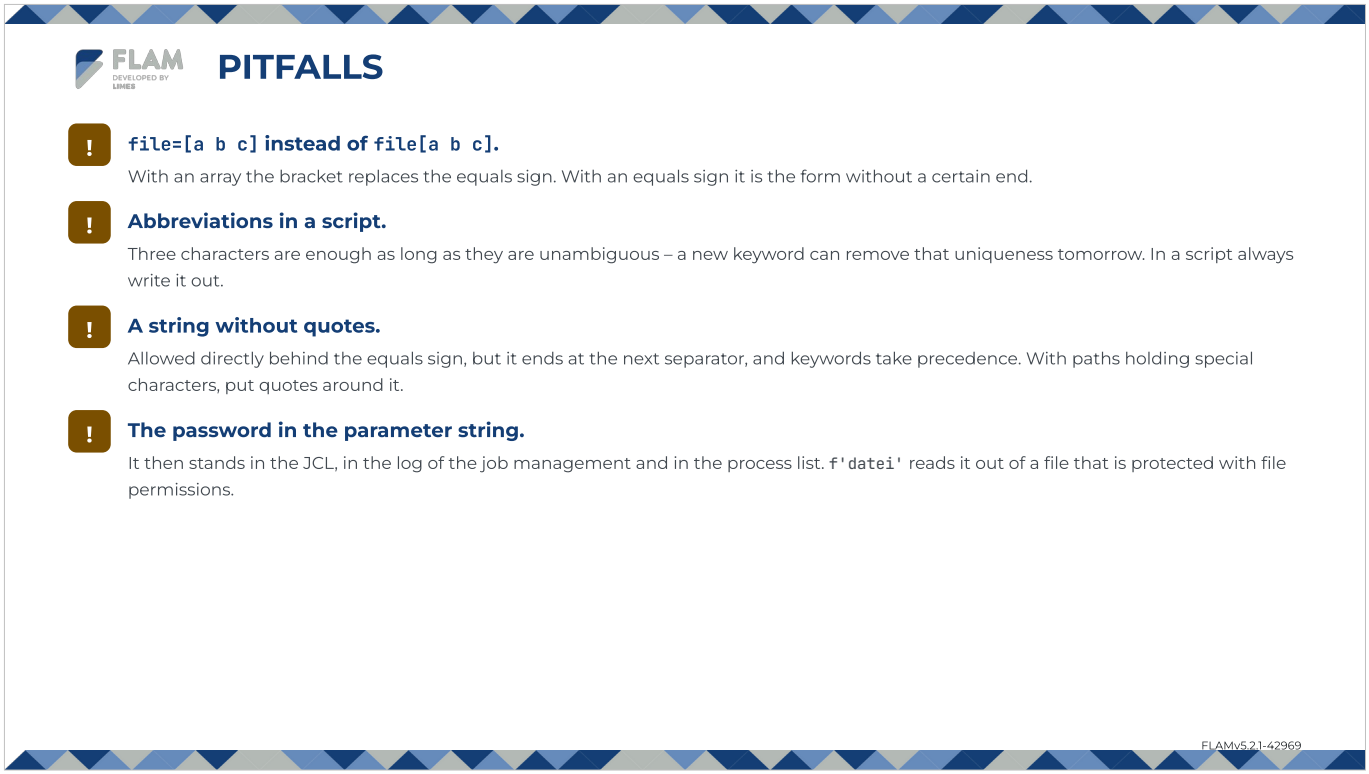
Measured 2026-10-09: store created, the read side stored as a member, a conversion run with it -- it works. With a wrong password "password wrong" comes instead of a parser error. And `read.text(file='archive/?member')` fails with return code 8: a member is an input for the **file-to-string layer**, not a data file.



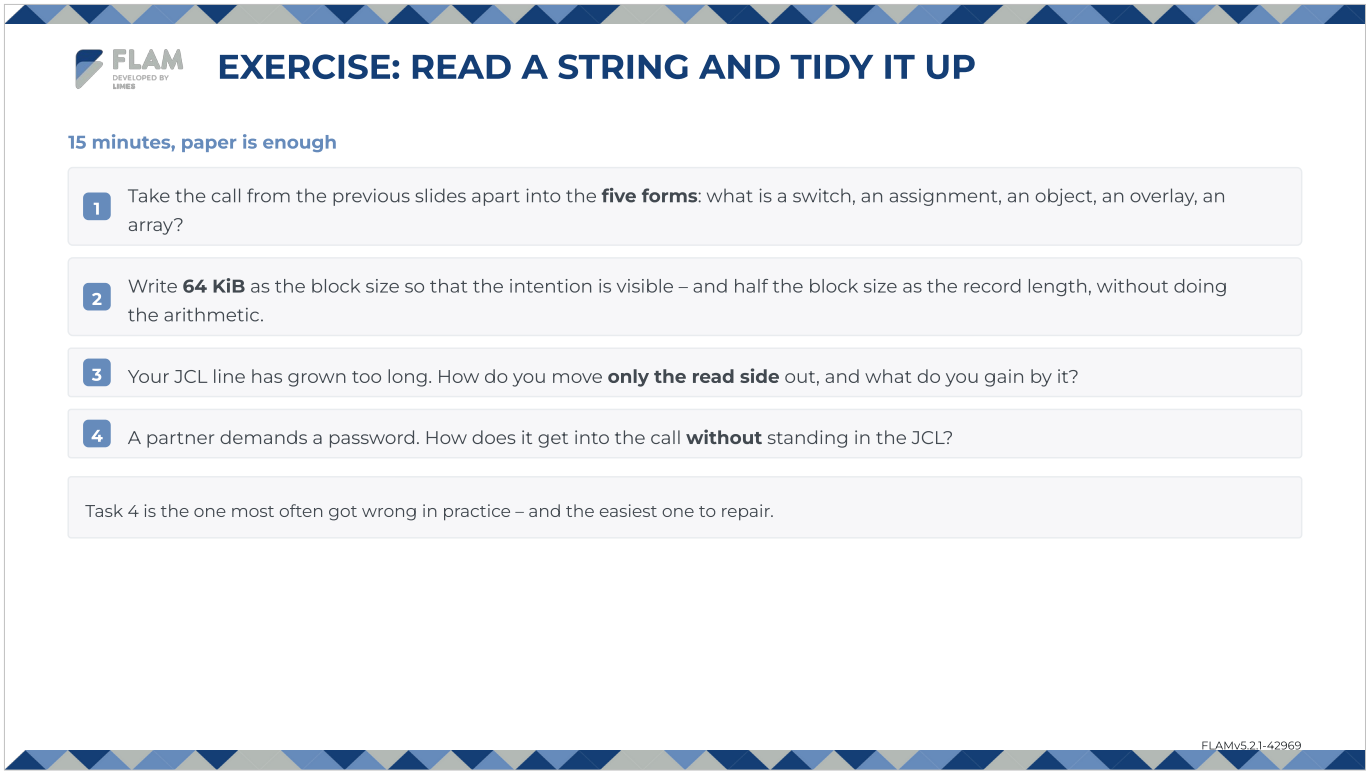
This slide saves more time in production than any other in this unit -- and it is overlooked, because nobody reads an error message. The message has three parts, and you read it **from the bottom up**: At the bottom stands **what was understood successfully**. That is the actual information: up to here everything was in order. In the middle stands the **place** -- row, column, the input itself and below it a marker under the character or the word where it broke off. At the top stands **what was expected**. In the example: a closing parenthesis is missing -- and the parser adds straight away what is probably really the matter, namely a misspelled keyword. The sentence for the room: the parser does not know what you meant, but it knows exactly where it stopped. That is more than most tools offer. Both messages shown are real output.



The slide that explains why the help never goes stale -- keep it short, the documentation unit covers it in detail. The command structure stands **once** in a table in the product.
Out of it come the parser, the built-in help, the manual pages and the property files. So there is no second place holding the description of a parameter -- and therefore none that can be forgotten.
Two built-in functions make that verifiable, and both are worth running once in the room: `flcl grammar` prints the grammar of the command language, `flcl lexemes` the lexemes together with the predefined constants. Everything on the previous slides comes from there. And CLE/P is open source. Whoever builds their own tools can use the same command line -- including the help and the manual generation.



All four out of support cases and out of training courses.



Fifteen minutes. The exercise is deliberately reading work -- there is nothing to type, there is something to understand.
The solutions: **Task 1.** `read.record` is an overlay (one of several variants), `file=...` an assignment, `suptws` a switch, `hash(...)` an object, and `file['a' 'b']` would be an array. **Task 2.** `blksize=64*KiB*` *-- and whoever wants the half as the record length writes `reclen=blksize/2` instead of the number.* **Task 3.** The read side into a file and a reference to it with `read.record=>'DD:RDPAR'`. Comment the entries in the file -- that is the actual gain. **Task 4.** `password=f'/pfad/pw.txt'`. The password then stands in a file with file permissions, not in the call; the log shows `***SECRET***`.

WHAT TO TAKE AWAY

- **CLE** is the frame, **CLP** reads the parameter string – that is why the same string runs on every route
- **five forms**: switch, assignment, object, overlay, array – there are no more
- numbers in five bases, and you can **calculate**: `blksize=4*KiB`
- `f'datei'` fetches the value out of a file – the handle for **passwords**
- every object may come out of a **file of its own** through `=>`, with comments
- the error message names **column and state** – read it from the bottom up

Go through the learning objectives. And the pointer forward: the built-in functions have a unit of their own, and so does the file selection -- both are language, but both are big enough for half an hour of their own.



The sister unit to the command language: there it was about the parameter string, here it is about the frame. Twenty-six functions that have nothing to do with data and still govern the working day: looking things up, presetting, setting the environment, diagnosing. The arc runs from the return value of the command through the property files to what belongs in a ticket. The sentence for the start: **the tool explains and configures itself.** There is no configuration file to guess at, and no parameter list to copy off a web page.



The second objective is the operationally most important one: whoever sets `MAXCC` wrongly loses a real error. The fourth is the handle that keeps a recurring call short without writing a script.

FLAM
DEVELOPED BY LIMES

TWENTY-SIX FUNCTIONS, FIVE GROUPS

GROUP	FUNCTIONS	WHAT FOR
Look up	SYNTAX, HELP, MANPAGE	the structure and the meaning of a parameter
Generating	GENDOCU, HTMLDOC, GRAMMAR, LEXEMES, ERRORS	manual, grammar, return codes
Presetting	GENPROP, SETPROP, CHGPROP, GETPROP, DELPROP	entries that are always the same
Environment	SETENV, GETENV, DELENV, LSTENV, HLPENV, CONFIG, SETOWNER, GETOWNER	what applies without standing in the call
Diagnosis	VERSION, ABOUT, LICENSE, TRACE	what belongs in a ticket

None of these functions processes data. They are the frame – and the frame is the part that costs or saves time in production.

FLAMv5.21-42969

The overview. Do not read it out -- name the groups and say that the unit goes through them in this order. The point worth making: none of these functions processes data. They are the surroundings, and that is exactly why hardly anybody knows them -- until they are needed. One way to place them:
the first two groups replace the manual, the third replaces copy and paste, the fourth replaces scripts around the call, and the fifth is what you write into a ticket.

FLAM
DEVELOPED BY LIMES

WHAT MAY STAND AROUND THE COMMAND

THE CALL LINE, AS THE TOOL DESCRIBES IT ITSELF

```
flcl [OWNER=oid] command "... argument list ..."
     [MAXCC=[max][-min]] [QUIET/SILENT]


flcl [OWNER=oid] command=" parameter file name "
flcl [OWNER=oid] command=>"parameter file name "
```

SETTING	EFFECT
`OWNER=oid`	a configuration and properties of its own per owner
`QUIET`	no log of the frame – for jobs with a protocol of their own
`SILENT`	no error messages either – **with care**

FLAMv5.21-42969

The slide explains the line that stands in every JCL, and the three entries that can be set there. `OWNER=` chooses a configuration of its own. On a machine with several tenants or several applications that is the way to keep their presets apart -- every owner has its own properties and environment entries. `MAXCC=` is the return-code bracket, and it has a slide of its own because it is dangerous. `QUIET` switches off the normal log of the command line frame, `SILENT` the error messages as well. In a job that writes its own protocol, `QUIET` makes sense. `SILENT` should only be set when you are sure -- after that you no longer see why something went wrong.
And the third form is the most important one for long strings:
instead of the argument list a **file name** may stand there.
`command="file"` or `command=>"file"` -- and the whole parameter string is moved out.

MAXCC: THE BRACKET AROUND THE RETURN CODE

SETTING	RUN WITH CC 1	RUN WITH CC 8
without	**1**	**8**
`MAXCC=0`	**0** – the warning disappears	0 – **the error too**
`MAXCC=4`	1	**4** – the error is reported smaller
`MAXCC=8-4`	**0** – below the bound	**8** – stays visible

ONLY FLAM CAN DO THIS

`-min` returns 0 when the code is **smaller** than the bound – small things disappear, errors remain. That is the entry you want.

FLAMv5.21-42969

The most important slide of the unit, and it holds a trap that sits in production jobs. `MAXCC=max` caps the returned code. That is meant to suppress warnings -- a run with completion code 1 returns 0 with `MAXCC=0`, and the job carries on. The trap: the cap applies to **real errors** too. Measured: a run that ends with 8 without the entry ends as **4** with `MAXCC=4`. The error is not gone, it has only become smaller -- and if the job management only reacts from 8 on, it no longer reacts. The second part, `-min`, works differently from what the name suggests: if the code is **smaller** than min, 0 is returned. That is the friendlier variant: `MAXCC=8-4` gives 0 for a run with 1 and 8 for a run with 8. Small things disappear, errors remain. The recommendation for the room: **do not cap, set a lower bound.** `MAXCC=8-4` instead of `MAXCC=4`. All three rows of the table are measured.

LOOKING THINGS UP ON PURPOSE INSTEAD OF BROWSING

THE PATH IS THE QUESTION – DEPTH1..DEPTH9 OR ALL THE DEPTH

```
flcl syntax conv.read                  # what is there
flcl syntax conv.write.text all        # everything below it
flcl help conv.read.record             # what it means
flcl help conv.read.record.prncontrol man   # the manual page
flcl manpage conv.read=read.txt        # write it to a file
```

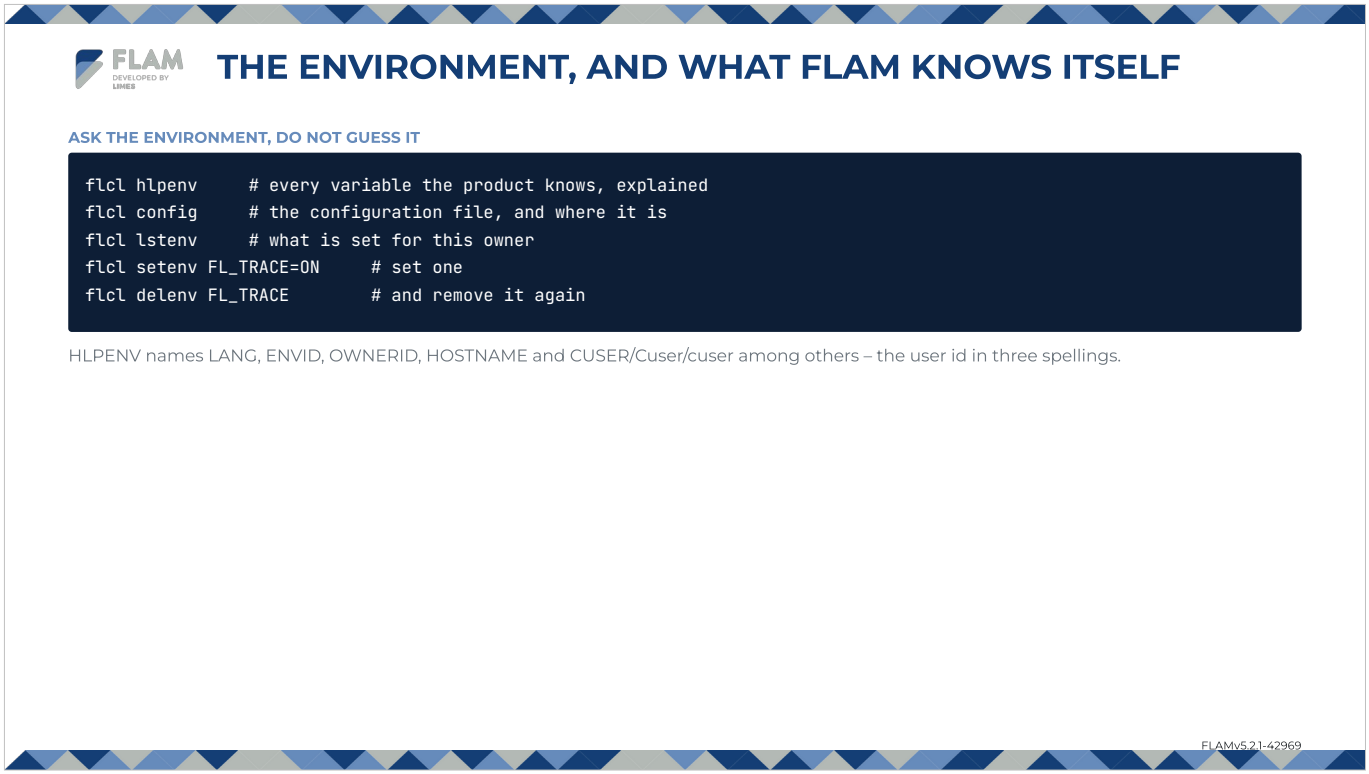
The same table in the product produces the parser, the help and the manual. So the help cannot go stale.

FLAMv5.21-42969

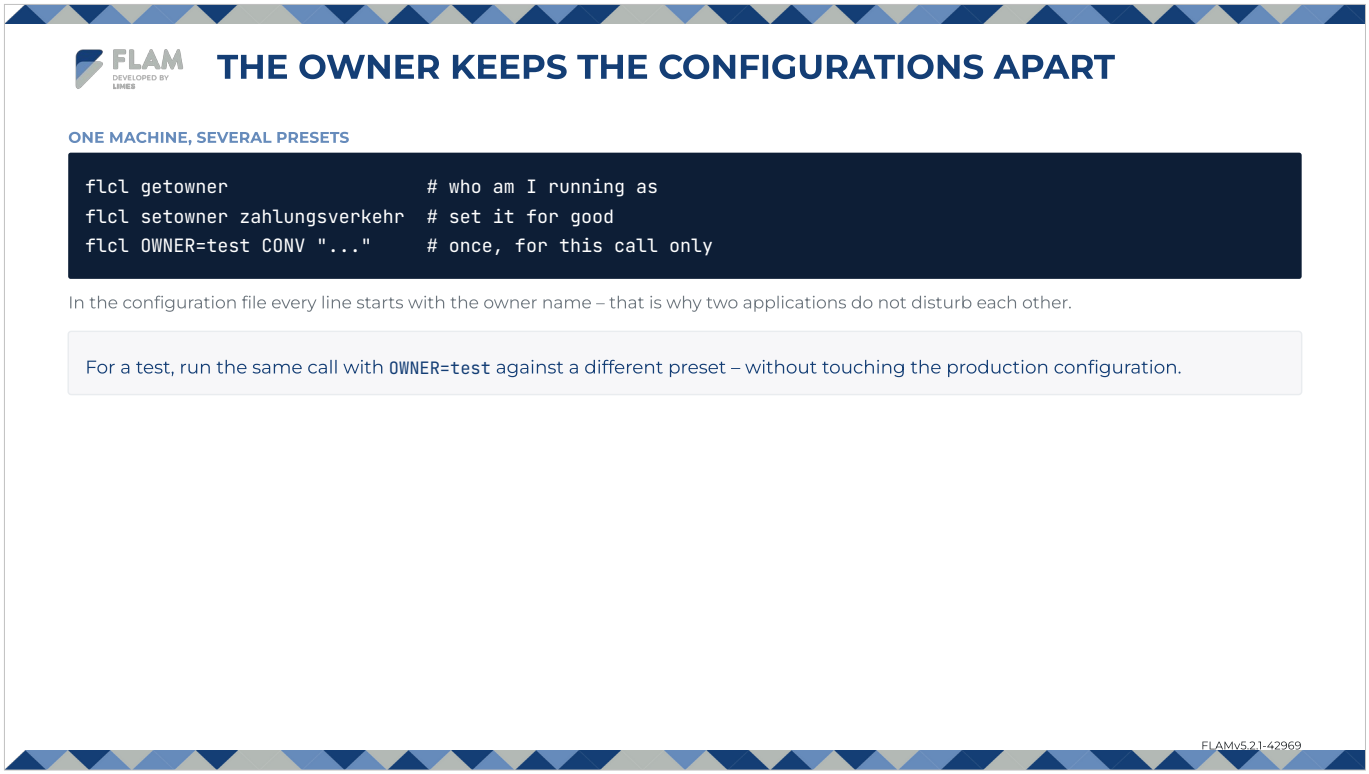
Three functions that approach the same goal from three sides.
The difference is the depth. `SYNTAX` gives the **structure**:
which parameters exist at this place, in one line. That is the function for the question "what was it called again". `HELP` gives the **meaning** -- one line of explanation per parameter. With `MAN` behind it, the whole manual page for exactly this parameter. `MANPAGE` writes the manual page into a file, for a whole command group too. The handle that makes it practical is the **path**. You do not ask the command, you ask the place: `conv.read.record.prncontrol`. And `DEPTH1` to `DEPTH9` or `ALL` decide how many levels come along -- otherwise the answer buries you. Show it once in the course:
`flcl syntax conv.read` shows the variants, `flcl help conv.read.record.prncontrol man` explains a single entry. That is faster than any PDF.



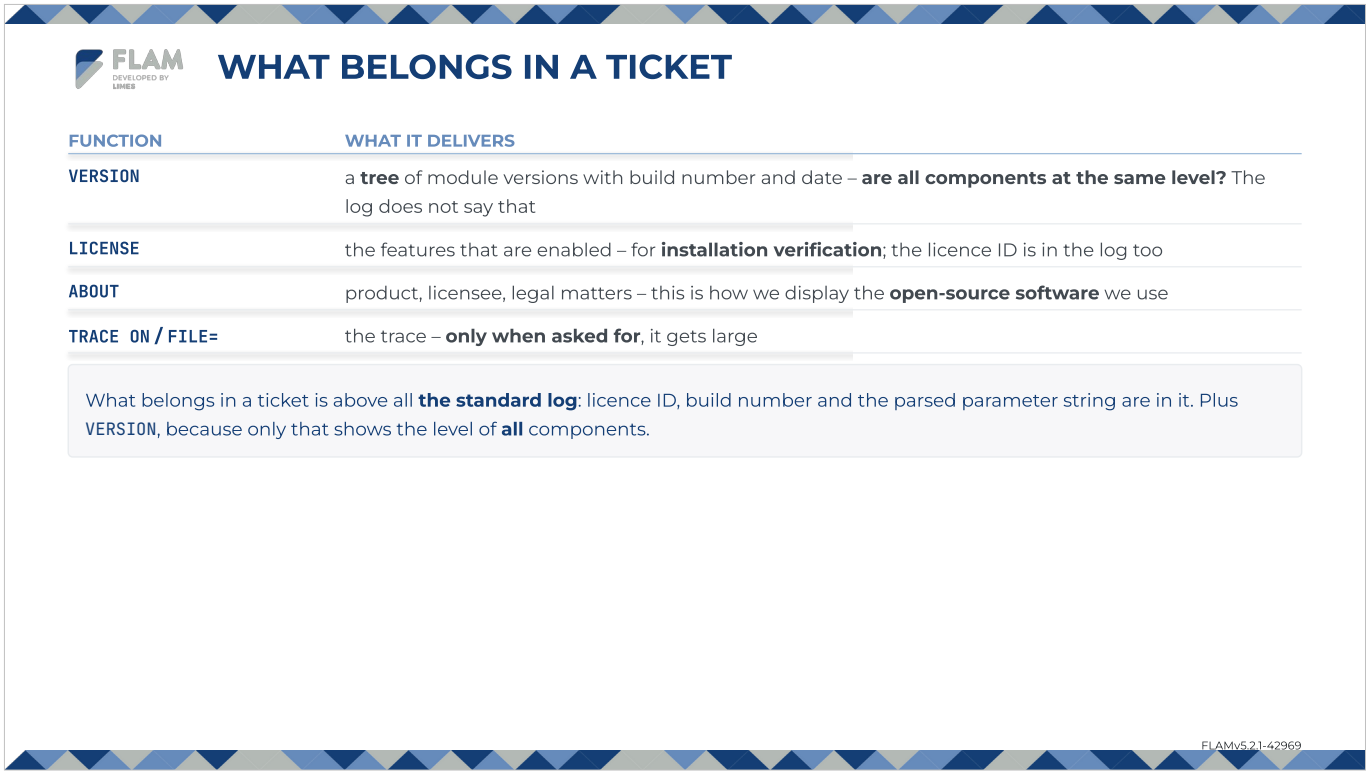
The slide for everybody who repeats the same parameter in twenty jobs. The way has three steps, and `GENPROP` is the one that surprises: `flcl genprop conv=file` writes **every** parameter of the command as a commented-out line into a file, with its type and help text beside it. Measured: for CONV that is **15,972 lines**. You do not read them -- you look for the three you need and take the semicolon away in front of them. `SETPROP` reads the file in and makes it effective. `CHGPROP` changes a single entry without a file. `GETPROP` shows what currently applies, and `DELPROP` clears up. What a property is: a **preset**, not a constraint. If the entry stands in the call, the call wins. That makes it the right place for house standards -- the block size, the CCSID, the virus scanner -- and the wrong one for anything that differs per run. A hint that saves trouble: a property applies per **owner**. Whoever wants it for the whole machine has to know the owner the jobs run under.



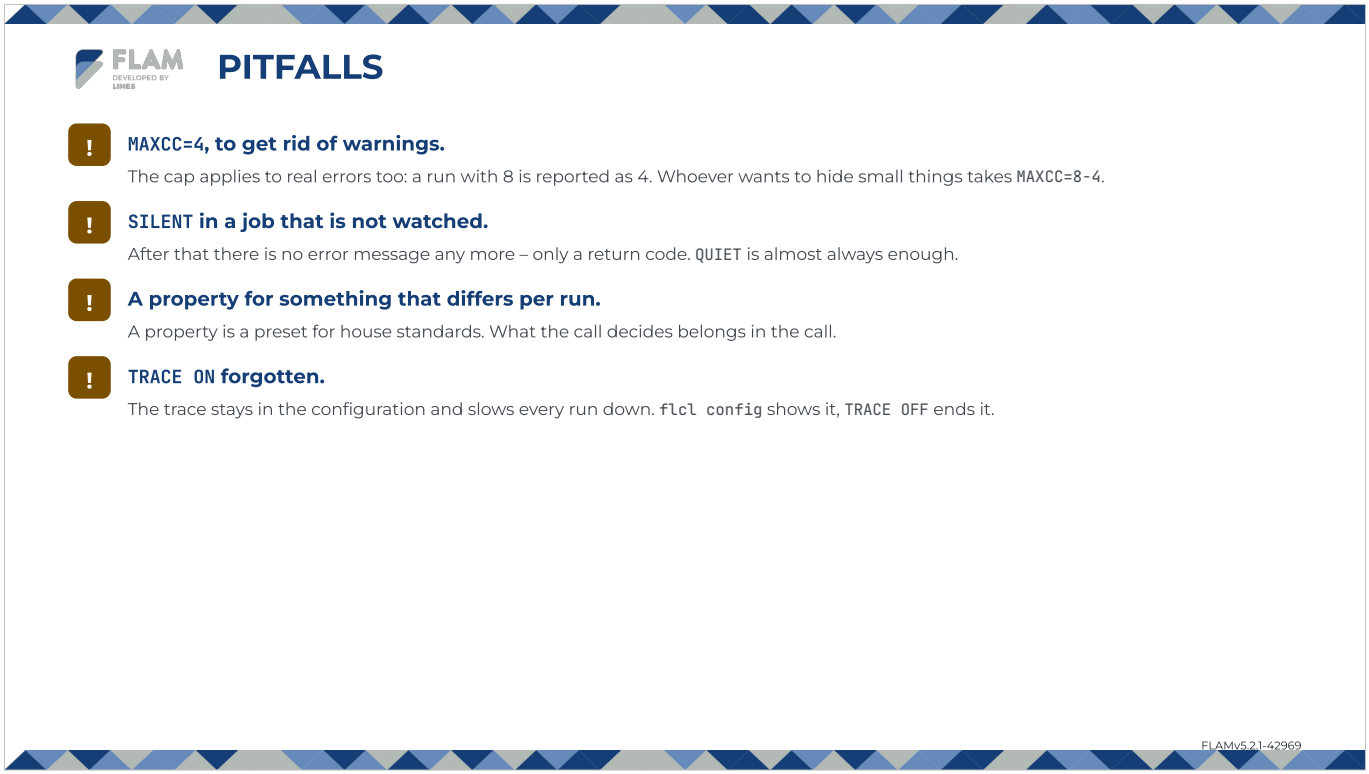
Two things on one slide, and hardly anybody knows the second. First the administration: `SETENV` sets a variable in the configuration of the owner, `GETENV` and `LSTENV` show it, `DELENV` removes it.
`CONFIG` shows the whole configuration file together with its location -- that is the first question with every "it does not work here". Second, and this is the find: **`HLPENV` lists every environment variable the product takes any notice of**, with an explanation. So you do not have to search for which variable exists -- you ask. A few from the list that usually raise a question in the room: `LANG` determines language and CCSID, `ENVID` the environment qualifier for the key label templates, `OWNERID` the owner when none is set, and `CUSER`/`Cuser`/`cuser` give the user id in three spellings -- handy in file names. The advice: the environment belongs in a controlled start environment, not in the user's profile. The security unit says why.



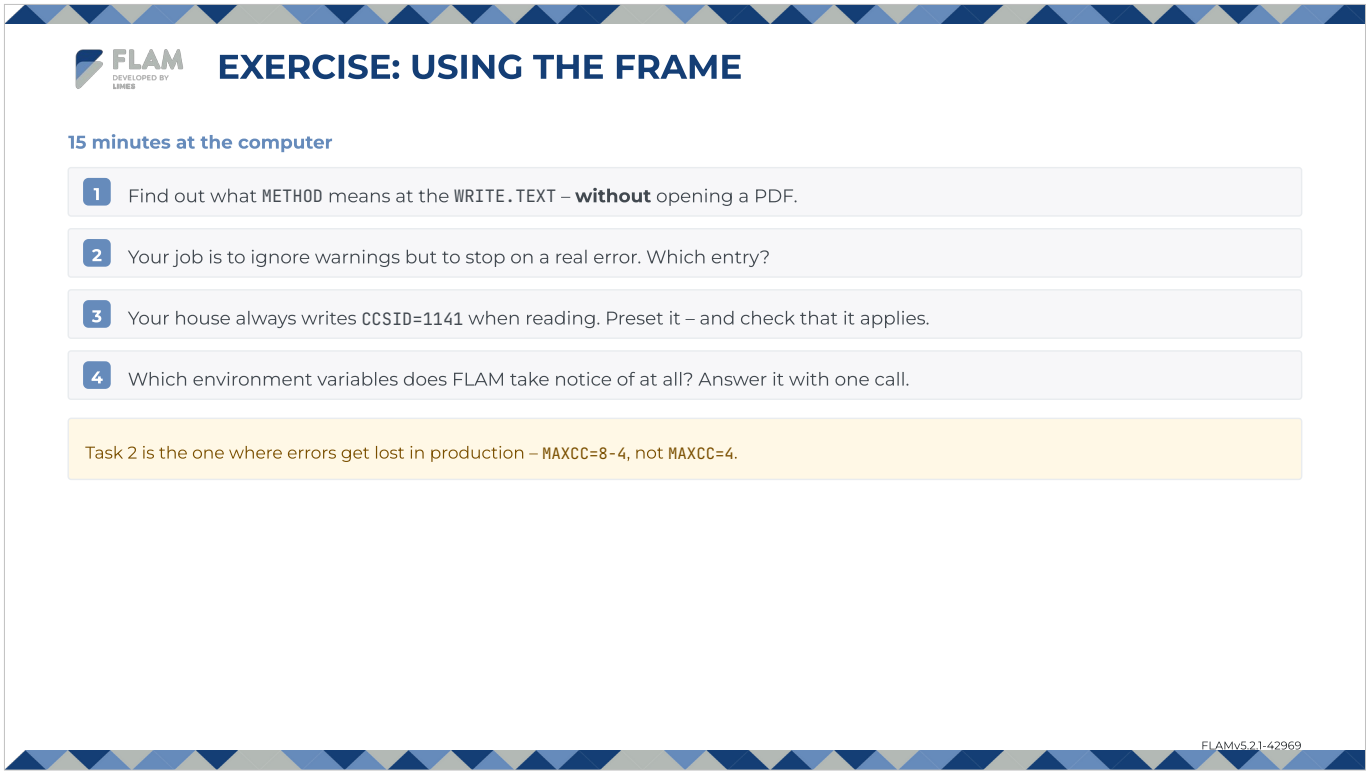
A short slide with an answer to a question that comes up in every data centre: how do I keep two applications apart on one machine? Through the owner. `SETOWNER` sets it, `GETOWNER` shows it, and `OWNER=` at the call overrides it for a single run. Every owner has its own properties and its own environment entries in the same configuration file -- you can see it in the lines `CONFIG` prints: they start with the owner name. What that brings in practice: the payment application gets its presets, the archive gets its own, and nobody changes the other one's by accident. And for a test the same call can be run with `OWNER=test` against a different preset without changing anything.



The slide that saves the support work -- and therefore saves the user waiting time. `VERSION` gives not one line but a **tree**: the version of every module involved with its build number and date. That is the piece of information the support starts with, because it says which code ran. `ABOUT` names the product, the licensee and the legal matters, `LICENSE` the features that are enabled -- the question "is this installation allowed to do that at all" is answered in one line. `TRACE ON` switches the trace on, `TRACE FILE=name` writes it into a file.
That is what the support asks for when the log messages are not enough -- and it should be switched on **only on request**, because it grows large and makes things slow. And now the placing that has to be given so that nobody does the wrong collecting work: **the most important thing for a ticket is our standard log.** It holds the licence ID -- through which we know the enabled features --, the build number and the **parsed parameter string**. Whoever sends the log has sent almost everything already. The only piece of information the log does not give is the **comparison of the build numbers across all components**; for that the output of `VERSION` is needed. So what the three functions are really there for: `VERSION` and `LICENSE` serve the **customer** for installation verification -- are all components at the same level, and am I working with my production licence? And `ABOUT` fulfils our obligation to display the open-source software we use. For a ticket therefore: **the log**, plus `VERSION` for the consistency of the installation, and the parameter string if it is not in the log. Details in the documentation unit.



All four out of production.



Fifteen minutes at the computer. The solutions: **Task 1.** `flcl help conv.write.text.method man` -- the path is the question, `man` gives the manual page. **Task 2.** `MAXCC=8-4`. With that the codes below 4 disappear and a real error stays 8. `MAXCC=4` would be wrong, because it pushes the 8 down to 4. **Task 3.** `flcl genprop conv=conv.prop`, free the line for the CCSID in the editor, `flcl setprop conv=conv.prop`. And check afterwards with `flcl getprop conv.read DEPTH1`. **Task 4.** `flcl hlpenv` -- the question is answered without searching. Whoever has seen the list once asks it first in future.

WHAT TO TAKE AWAY

- `MAXCC=8-4` **instead of** `MAXCC=4` – a cap makes real errors smaller too
- the **path** is the question: `flcl help conv.read.record.prncontrol man`
- `GENPROP` writes **every** entry out commented -- you take the semicolon away in front of the three you need
- a property is a **preset**, the call wins – and it applies per **owner**
- `HLPENV` **names every environment variable the product takes notice of**
- what belongs in a ticket is above all **the log** – licence ID, build number and the parsed parameter string are in it; `VERSION` shows the level of **all** components

Go through the learning objectives. And the note: the order of precedence of the five places an entry can come from is in the unit *Administration* -- here it was about which function you touch them with.

FLAM TRAINING, LEVEL 0

One call instead of a loop

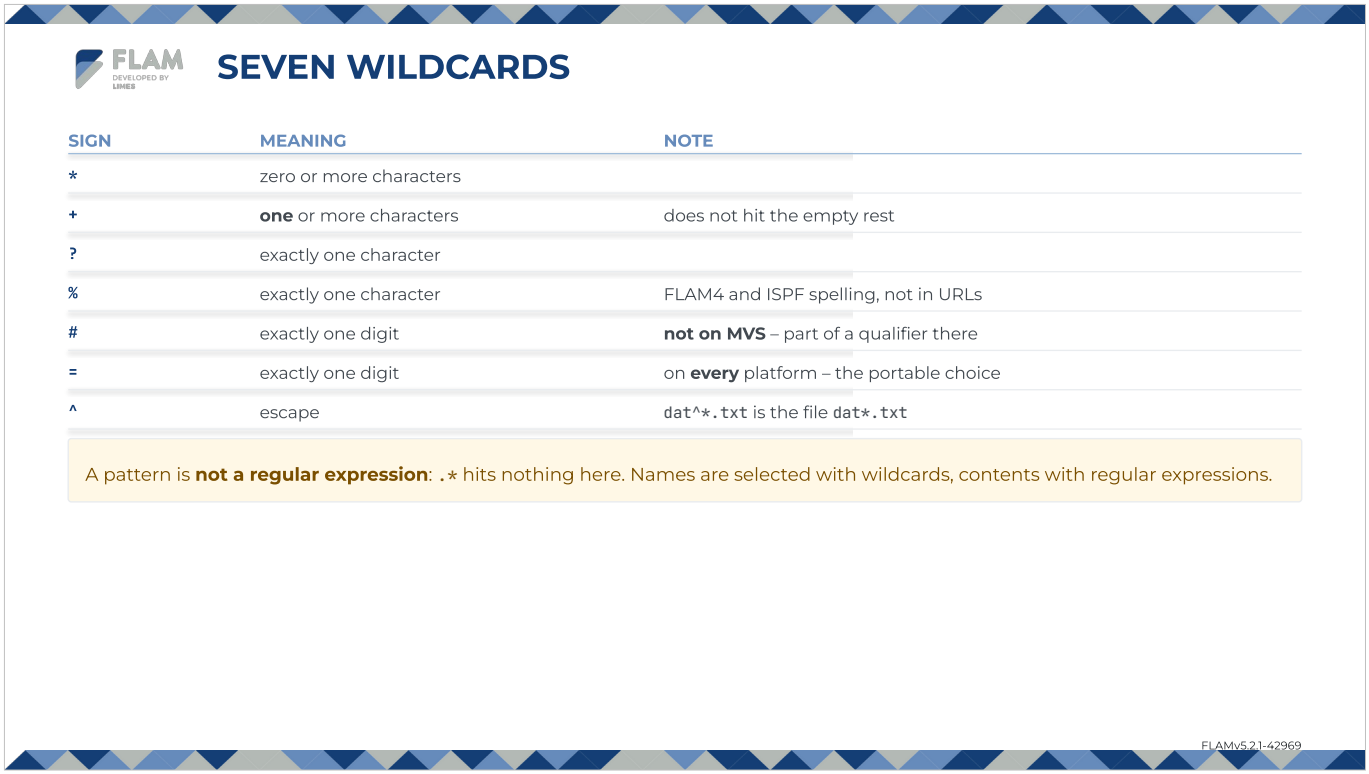
Wildcards, members, directory walk – and how the output name comes about

Falk Reichbott · limes datentechnik gmbh

The third unit on the language, and the one with the most immediate use: it replaces loops. The arc: first the selection -- wildcards, the file name itself, members in containers, the directory walk. Then the other direction: how the output name is formed from the input name. The sentence for the start: **whoever knows this unit writes no more loops.** One call processes a whole folder and names every output by a rule -- in the JCL that is one line instead of a REXX program.



The fourth objective is the one that justifies the unit: a rule instead of a list. Whoever has written one once needs it in every second job.



The vocabulary list of the selection. Showing three lines is enough, the rest is for looking up -- but the last line has to land. Everybody knows the `*` *and the* `?`*. The others are the interesting ones: The* `+` *demands* **at least one** character -- so it hits everything except the empty rest, which `*` *cannot do. The* `%` *is exactly one character and comes from the FLAM4 and ISPF world. Whoever comes from there keeps writing* `%` *and it works* **--** *only not in URLs. The* `#` *is exactly one digit, but does* **not apply on MVS**, because the hash there can be part of a qualifier. Instead there is the `=`: exactly one digit, on every platform. Whoever wants to write portably takes `=`. And the caret `^` is the escape. `dat^*.txt` *means the file that really is called* `dat*.txt`. That is needed rarely and is then the rescue. The sentence that saves a round with support: **a pattern is not a regular expression.** `.` *hits nothing here.*
For contents there are regular expressions, for names there are wildcards -- two languages, and a switch between them.



The slide that makes a procedure portable. Every file name goes through the same mapping before anything is opened. Four rules out of it are genuinely needed: **The tilde** stands for the home directory, on z/OS for the user id as the high-level qualifier. `~/daten/auszug.csv` on Unix and `~.DATEN.AUSZUG` on the host are the same idea. **DD names** may carry the prefix `DD:` but have not needed it since 5.1.22: **if only one qualifier stands there** -- no dot, no slash -- and an allocation of that name exists, then it is a DD name. The prefix only makes it unambiguous, and in a call that does not visibly come out of a JCL, being unambiguous is the argument for it. **On MVS** the shape of the name decides, and in this order: **a slash** makes it a path name, **a dot** a data set name, and **a single qualifier** -- neither dot nor slash -- a DD name. Three examples side by side, and the difference is one character every time: `auszug.csv` is the data set `AUSZUG.CSV`, `auszug` is `DD:AUSZUG` and gets allocated, `./auszug` is the file `auszug` in the current directory. So whoever means a USS file without a suffix **has to** write `./auszug`; without the `./` FLAM tries to allocate `DD:AUSZUG`. On USS a data set name starts with `//`, and with apostrophes behind it it is fully qualified; without apostrophes the user id is put in front. **Environment variables** in angle brackets are replaced -- on z/OS exported JCL symbols and system symbols too. That makes a procedure reusable without touching the JCL. Two warnings: if the variable is not defined it is replaced by **nothing** -- the name is then silently wrong. And whoever really needs a `<` or a `~` in the name writes it **twice**.

SELECTING MEMBERS IN A CONTAINER

THE SLASH OPENS THE CONTAINER

```
# all XML members of a ZIP, written into a FLAM archive
flcl conv "read.xml(file=test.zip/?*.xml)
           write.flam(file=test.adc
                      member=<CUSER>.XMLPDS([base])
                      ccsid=1141)"

# a PDS member on z/OS, with and without a pattern
FILE='~.TEST.XMIT(MYTEXT)'      FILE='~.QUELLE(A*)'
```

FLAMv5.21-42969

The slide that shows an archive feels like a folder. A container -- a ZIP, a PDS/E, a FLAM archive -- is opened with a slash, and behind it stands the pattern for the members. `test.zip/?*.xml` *are all the XML files in the ZIP; the* `?` demands at least one character in front of the extension. On z/OS the member name stands in parentheses, as one expects there: `~.TEST.XMIT(MYTEXT)`, and with wildcards `~.QUELLE(A*)`. *The point for the room: this holds* **for reading and for writing**.
When writing, the same rule forms the member name, and for that there are the name tokens of the next slides -- `member='[base].txt'` writes every element into the archive under its base name. And it holds for all three kinds of container with the same spelling. Whoever can unpack a ZIP can also read a PDS.



Now the other direction, and this is the part that replaces loops. An output name is a **pattern** of text and tokens.
Text stands there as it stands there. A token in square brackets describes a processing rule on the input name. The six tokens of this slide are the ones needed daily: `[path]` is the path without the last slash, `[name]` the file name, `[base]` the file name without the extension, `[ext]` the extension, `[member]` the part in parentheses -- the PDS member name -- and `[copy]` the whole name, which is what you need when you only want to append something. The example below is measured: three files `f1.bin` to `f3.bin`, read through a directory walk, give `f1.txt`, `f2.txt` and `f3.txt`. The sentence for it: the rule stands **once** in the call and holds for every file the walk finds. So you do not have to know how many there are and what they are called.

COUNTING, CUTTING, REWRITING

TOKEN	WHAT IT DOES
`[indN]`	counter **per output file**, N digits with leading zeros
`[midN]`	counter **per member** in an archive – for unique ZIP members
`[rndN]`	random digits – for temporary names
`[upper]` `[lower]` `[title]`	change the case
`[cutX]`	cut off: positive from the front, negative from the back
`[search=replace]`	swap the first occurrence; with * in front of it all of them
`[regex:/pattern/replacement/]`	regular expressions, PCRE compatible
`[AX]` `[AX-]` `[AX-BY]`	cut at the X-th occurrence of a character

FLAMv5.21-42969

The second half of the token list. Here lie the handles with which uniqueness is enforced. `[indN]` is the counter **per output file**: N is the minimum width with leading zeros, and 0 means the shortest possible form. Measured:
`[base|upper]_[ind2]` gives `F1_03`, `F2_02`, `F3_01`.
`[midN]` is the counter **per member inside an archive**.
That is needed when the same record description is written into a ZIP more than once and the ZIP has to have unique member names under ISO 21320-1. `[rndN]` are random digits -- for temporary names. `[upper]`, `[lower]`, `[title]` change the case. That is needed almost always on the way from the host to Unix. `[cutX]` cuts off: positive from the front, negative from the back. With it a member name is kept under eight characters. And then the replacements:
`[search=replace]` swaps the first occurrence, with a star in front of it all of them.
`[regex:/pattern/replacement/]` is the full form with regular expressions -- measured, `[name|regex:/^f([0-9])\.bin$/auszug_$1.csv/]` turns `f1.bin` into the file `auszug_1.csv`. Finally the kit for names with separators: `[AX]`, `[AX-]` and `[AX-BY]` cut at the X-th occurrence of a character. With that a data set name is taken apart at the dots without writing a regular expression.



The slide on which the tokens become a small language.
Several rules may stand in one token, separated by a vertical bar. Every rule gets the result of the previous one -- that is a chain, and it reads from left to right. Both examples are measured. The first turns a base name into upper case and appends a two-digit counter. The second builds a path name out of a data set name: the member name in lower case, an extension behind it. The advice: a chain with two or three rules is readable, one with six is not. When it gets long the call belongs in a parameter file where it can be commented -- the language unit shows that. And a small thing that saves time: the keywords in the tokens are **not** case sensitive, `[BASE]` works just like `[base]`. Whoever needs a square bracket in the name puts a caret in front of it.

ONE CALL FOR A WHOLE FOLDER

DIRECTORY WALK WITH EXCEPTIONS

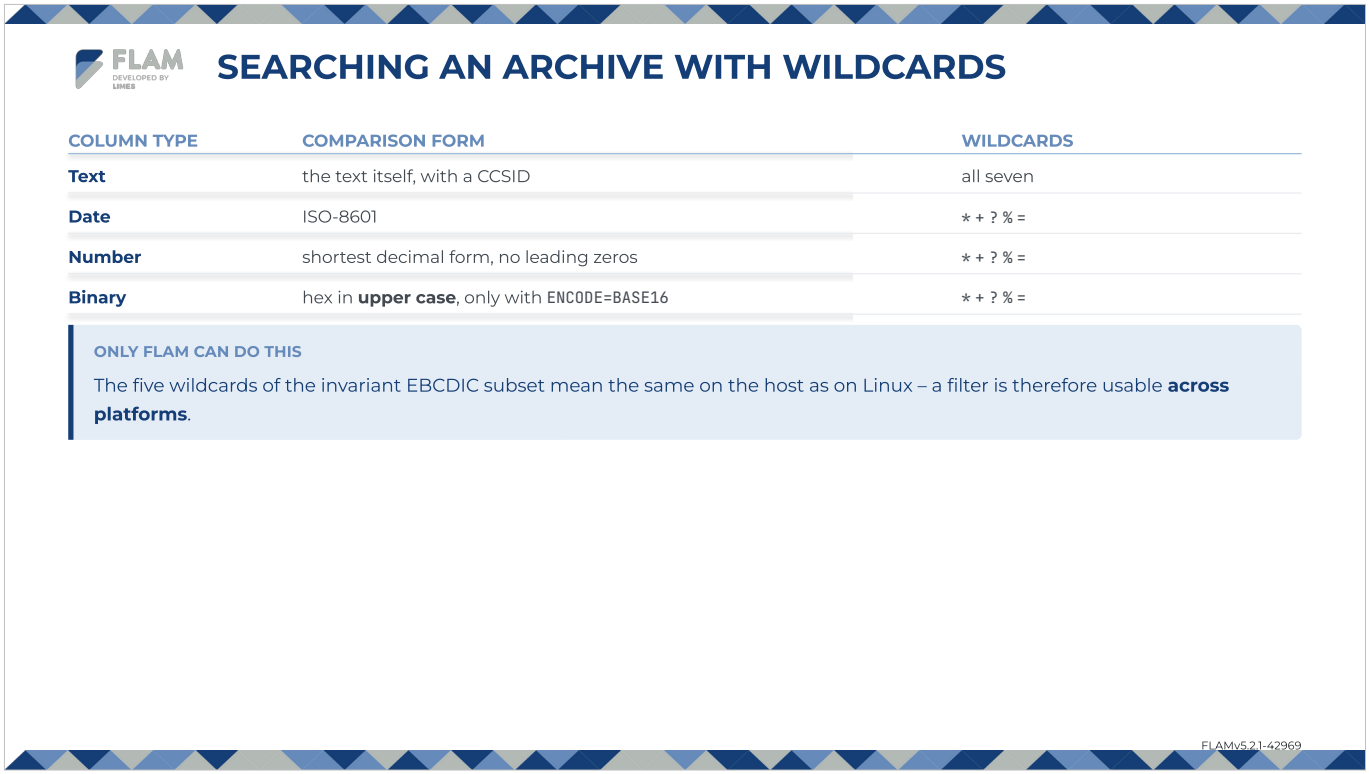
```
flcl CONV "read.record(file='/daten/ein/*.bin'
                       exclude['_test.bin' '.tmp']
                       recformat=FIX reclength=80 ccsid=1141)
           write.text(file='/daten/aus/[base|lower].csv'
                      ccsid='UTF-8' method=UNIX)"

# CC 0 = everything processed
# CC 4 = SOME FILES NOT PROCESSED -- the log names them
```

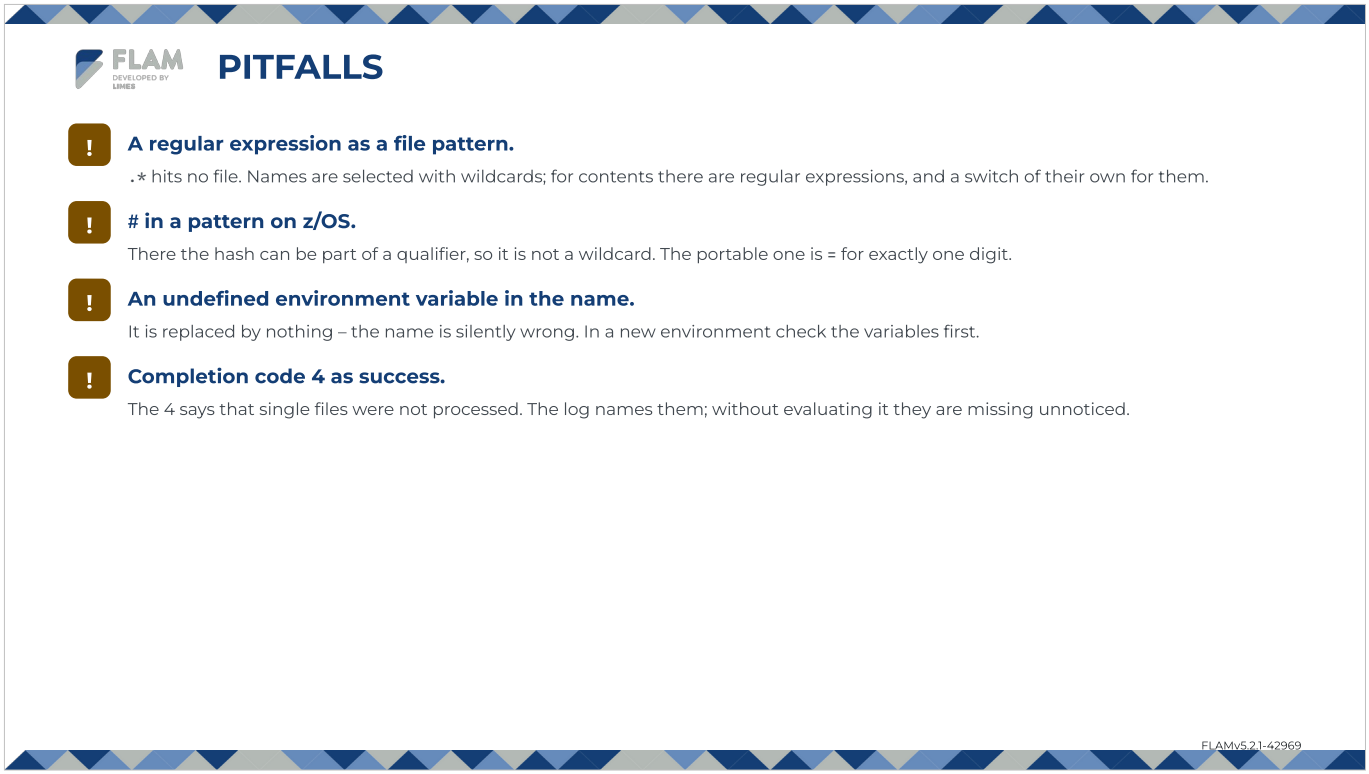
Completion code 4 means: single files were not processed. Whoever treats the 4 as success loses exactly the files that make work.

FLAMv5.21-42969

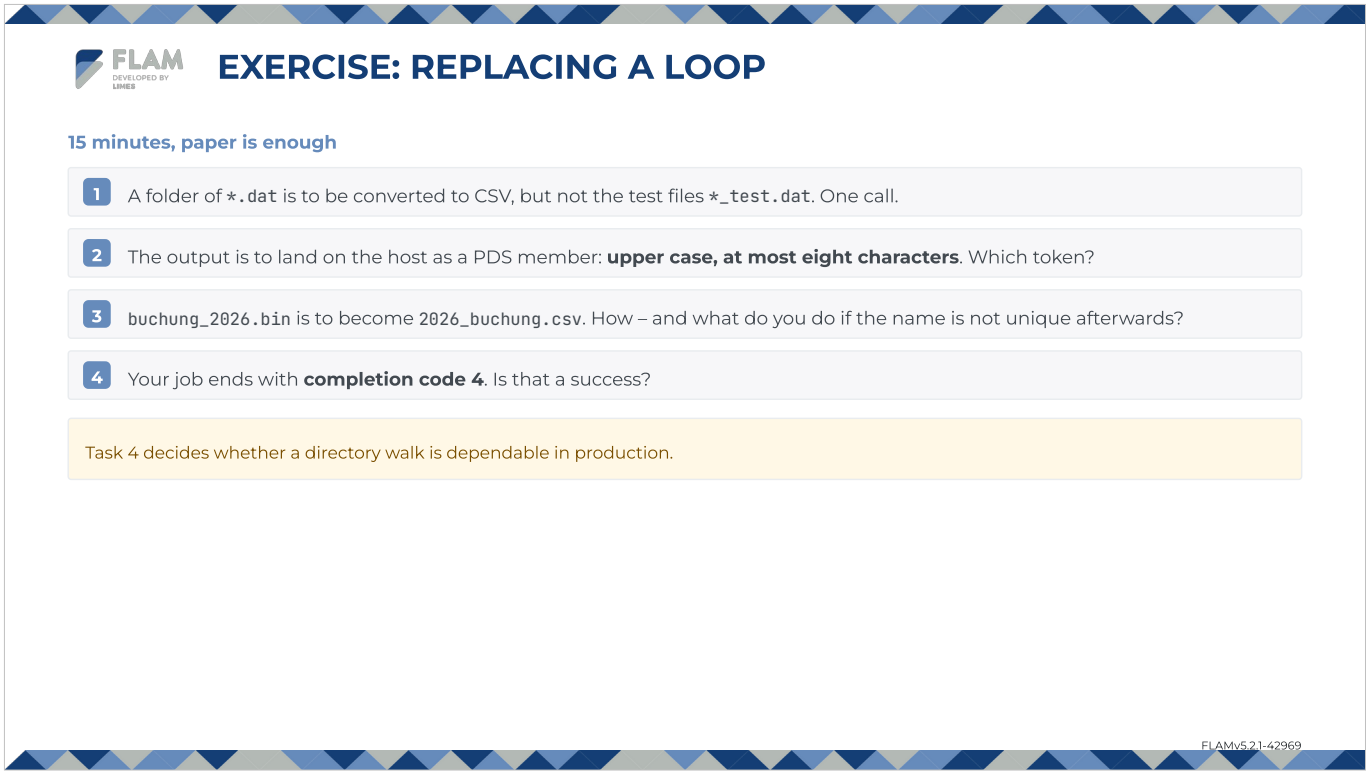
The slide on which the whole thing comes together -- and which holds a trap that sits in the job management. A pattern in the input name triggers a **directory walk**:
FLAM processes every file that matches and forms the output name out of the rule. That replaces the loop. `EXCLUDE` takes single ones out, and it is an array -- several patterns may be given. Measured: with `exclude['*2.bin*']` *two out of three files are processed, and the run ends with* **0**. An excluded file is therefore not a finding, it was never in. And now the trap: **completion code 4 means that single files were not processed** -- because one was locked, because a conversion failed, because a target existed. The run has not failed with that, but it is not complete either. Whoever treats the 4 as success in the job management loses exactly the files that make work. The log line names them. The advice for practice: evaluate the 4 and rework the files it names. The operations unit shows how to set that up in the JCL.



A slide for those who already work with archives -- it connects this unit with the search. Wildcards hold not only for names but also for **values** in a filter with which an archive is searched. And there is a restriction there that has to be known. A column is not stored as text in the archive but in a neutral form. For the comparison it is brought into a fixed text form: a date in ISO-8601, a number as the shortest decimal form without leading zeros, a binary column as hex in upper case. So the pattern has to fit that form, not the one the data was once delivered in.
And the restriction: only a **text column** carries a CCSID. That is why only there the hash and the caret can be recognised reliably -- both are variant characters in EBCDIC and lie differently per code page. For all other types `` `+` ``, `` `?` ``, `` `%` `` *and* `` `=` `` *remain, and those are invariant. The sentence for the room:* **whoever searches across platforms takes the invariant subset.** Then the pattern means the same on the host as on Linux.



All four out of support cases.



Fifteen minutes, paper is enough. The solutions: **Task 1.** `file='/ein/*.dat' *exclude*['_test.dat']`, and the output name `'/aus/[base].csv'`. One line instead of a loop. **Task 2.** `[base|upper|cut8]` -- upper case, then cut to eight characters. The order in the chain matters: rewrite first, then cut. **Task 3.** `[name|regex:/.../.../]`, or more simply with `[search=replace]` when it is only one fixed piece. And the note: if the name is no longer unique afterwards, `[ind0]` helps. **Task 4.** No -- the 4 means that single files were not processed. The job has to take them out of the log and rework them. That is the task that is most expensive in practice when it is solved wrongly.

FLAM
DEVELOPED BY LIMES

WHAT TO TAKE AWAY

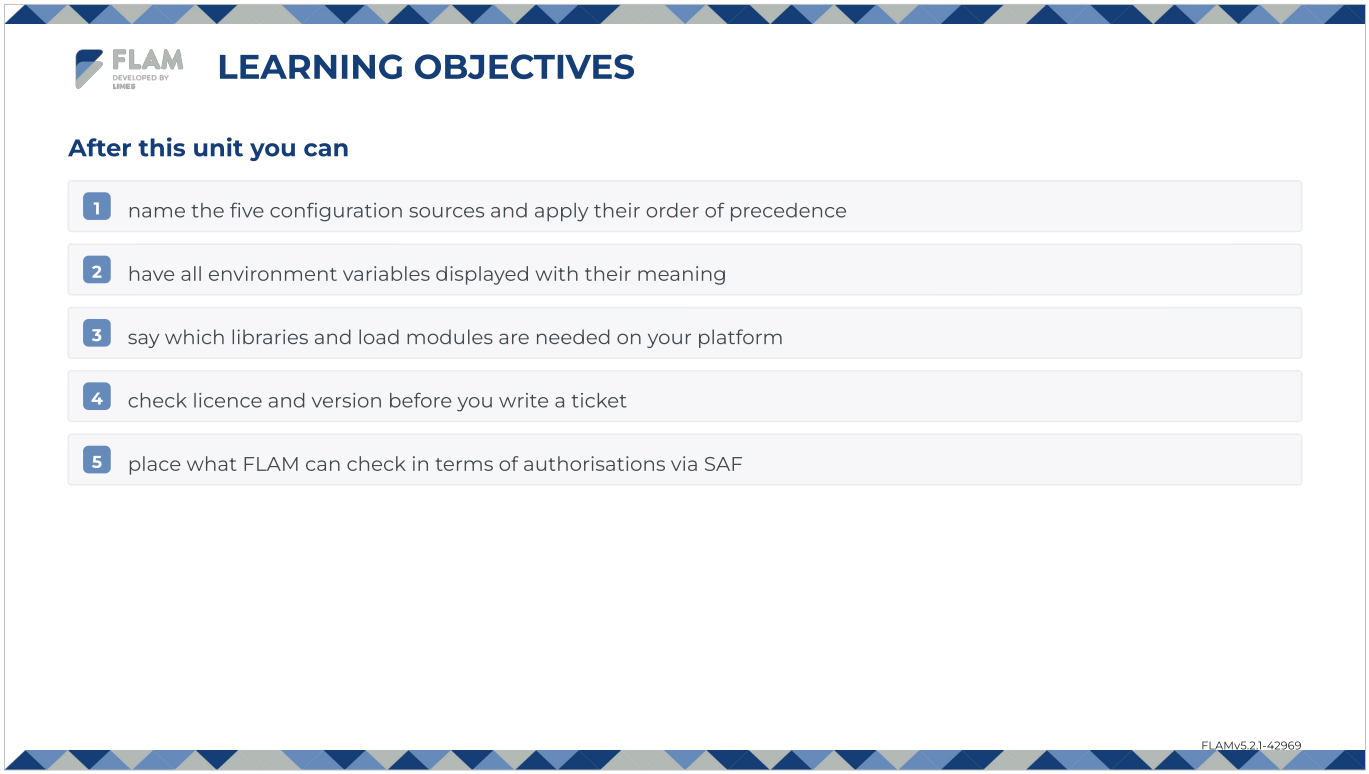
- **seven wildcards**, and = is the portable digit – # does not hold on MVS
- a pattern is **not** a regular expression
- ~, `DD:`, `//''` and `<variable>` make a name fit the platform – an undefined variable becomes **nothing**
- the slash opens a container: ZIP, PDS, FLAM archive
- the output name comes out of a **rule**: `[base]`, `[indN]`, `[regex:/…/…/]`, chained with |
- **completion code 4 means: not all files processed**

To look up in the manual: *1.4 Filename Mapping*, *3.2 Wildcards*, *3.3 Filename Handling*, *3.4 Directory Support* and *3.5 Input to Output Name Mapping*.

Go through the learning objectives. The pointer forward:
the network unit shows that the same name may also be a URL -- the selection then holds over SSH just the same.



Fourth unit of level 0, and the first one aimed at operations. The thread running through it: there are exactly five places where a setting can live, and they have a fixed order of precedence. Anyone who knows it never has to search for long again.



Objective one is the operational one: having the order of precedence in your head saves the most follow-up questions in operations.

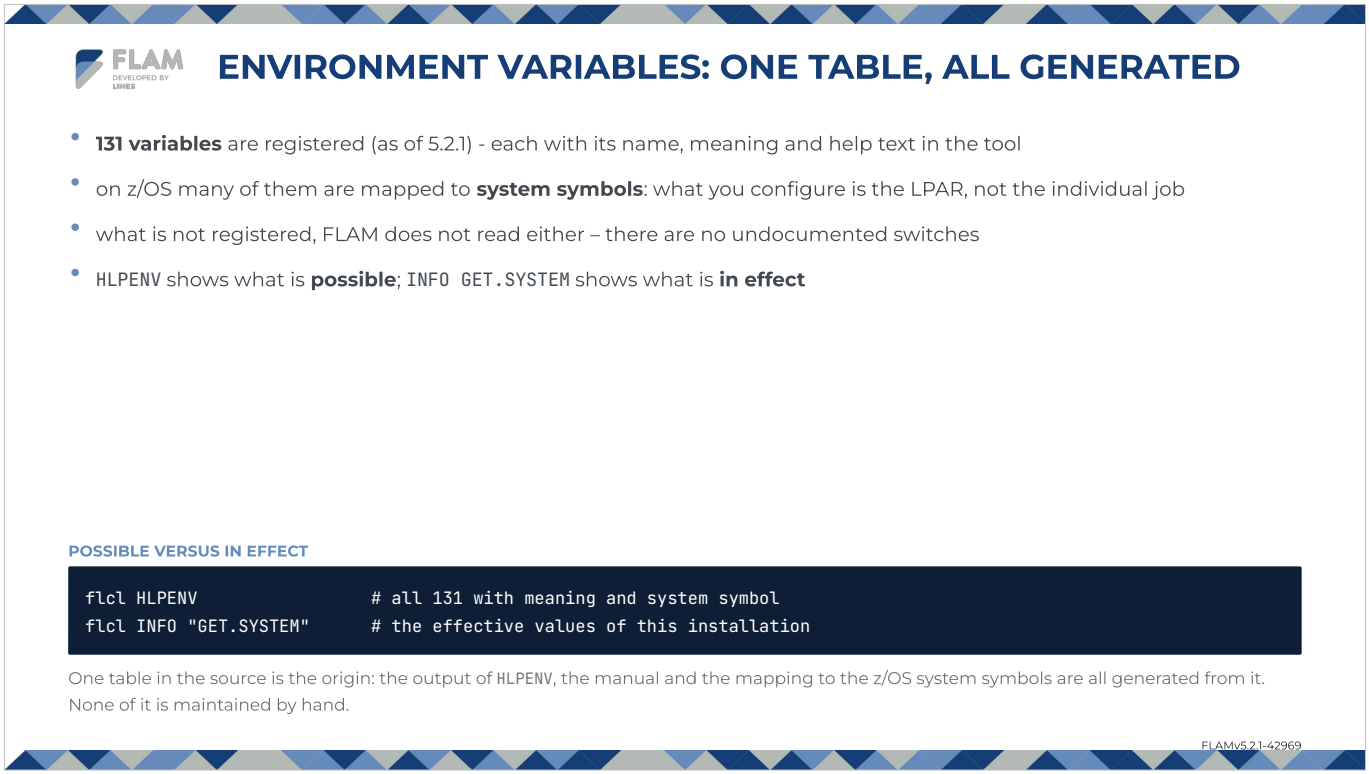
WHERE A SETTING COMES FROM

RANK	SOURCE	INTENDED FOR
1	Property file	in-house standards, permanent: CCSID, block size, report paths
2	Configuration / owner	several variants side by side: test, production, tenant
3	Environment variables	platform- and installation-dependent paths
4	Parameter file	long string, moved out - on z/OS via FLAMPAR
5	Command line	the individual case, wins against everything before it

"The parameter does not work" almost always means: a different one is taking effect. Check the order of precedence first, then the parameter.

FLAMv5.21-42969

The order of precedence onto the whiteboard. From top to bottom, the later entry wins.
This is the cause of the most common operational report "the parameter has no effect": one of them does take effect, just not the one you happened to be looking at.
In practice: when in doubt, have `flcl INFO GET.SYSTEM` print the active configuration instead of guessing.



Name the number, it has an effect: 131 registered variables (as of 5.2.1, counted via `flcl HLPENV`). Nobody has to know them -- but every one of them is described in the tool itself. `flcl HLPENV` is the command operations needs. It lists the name, the meaning and, on z/OS, the assigned system symbol in parentheses.
The point that makes this credible: there is **one** table in the source code, and everything else is generated from it -- the output of `HLPENV`, the manual and, for some of the variables, the mapping to the z/OS system symbols.
Since 5.1.26 the manual no longer maintains the list by hand. An environment variable that is not in this table does not exist -- neither in the tool nor in the documentation.
So name the number as a snapshot and not as a property: it grows with the table.
The peculiarity on z/OS is that many variables are mapped to system symbols -- which means you configure the LPAR and not every job.
The second command alongside it is the one for diagnosis: `INFO GET.SYSTEM` shows not the possible values but the **effective** ones. In a failure case that difference is everything.

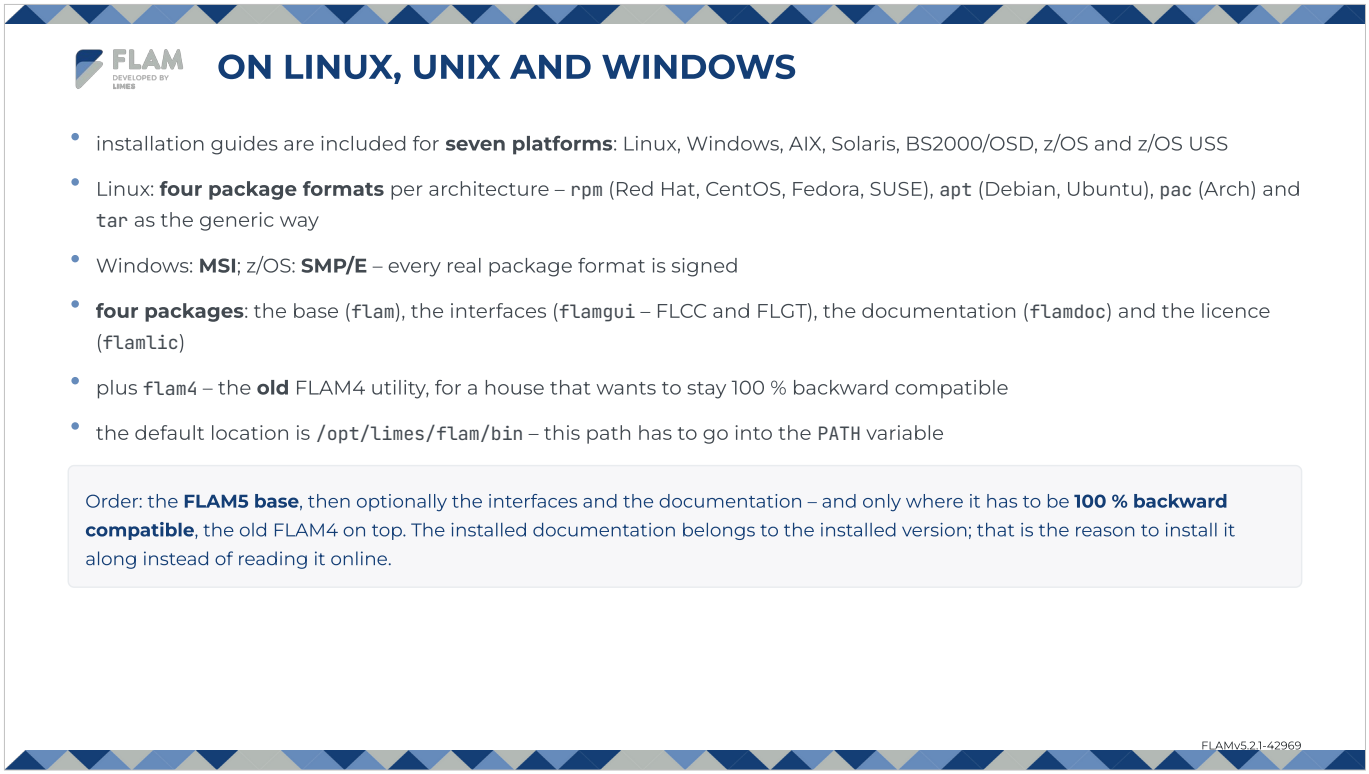
ON Z/OS: LIBRARIES AND LOAD MODULES

WHAT	WHERE	WHAT FOR
Load modules	STEPLIB or LINKLIST	flcl, FLUC-UP, FLAM-UP, record interface
Parameters	FLAMPAR (DD)	the parameter string, moved out of the JCL
Started task	its own procedure	only for the subsystem – without it no SUBSYS=
USS part	LIBPATH	for use from USS and from Java via JNI
Authorisations	RACF class $FLAM	optional, can be enabled from version 5.1.21

The subsystem needs a running service, not just a library. Here operations and application development have to work together.

FLAMv5.2.1-42969

The three lines that are needed in operations. What comes after that depends on which of the three routes is used.
Important: a started task runs for the subsystem. So anyone who wants to use the subsystem needs not just a library but a running service -- that is the point at which operations and application development have to work together.



Keep it short. The only point that regularly draws questions is the PATH -- and that the documentation is a package of its own. On the package formats the full answer, because too little is usually said here: for Linux there are **four** ways per architecture -- `rpm` for Red Hat, CentOS, Fedora and SUSE, `apt` for Debian and Ubuntu, `pac` for Arch, and `tar` as the generic way for everything else. So anyone asking "is there an RPM" gets a yes, and not as an exception. Everybody can look it up: the download area is arranged by platform, architecture and word size, and the package formats sit below that.
On Windows it is MSI, on z/OS SMP/E. The installation guide is included for seven platforms. The split into packages has one reason each, and both come up as a question. **The interfaces are a package of their own**, because not every server has a graphical surface -- on a batch server one does not install FLCC and FLGT at all. **FLAM4 is a package of its own**, laid over FLAM5: FLAM5 brings a new FLAM4 utility with it that in the edge cases does not work a hundred per cent like the old one -- with the input-output name mapping and with member access. Whoever does not want to adjust anything there replaces the new utility with the current old one from the `flam4` package. If somebody finds the HP-UX guide in the package: it is left over from earlier releases. HP-UX is no longer supported.

CHECKING LICENCE AND VERSION

- the delivery contains the **demo licence** – everything runs, but only up to **1 MiB per file**
- the demo licence **forbids commercial and production use**, it comes without warranty and without support and may contain functions in a pre-release state
- for production use a **commercial licence** is acquired: a **package of its own** over the existing installation, no reinstallation
- `flcl license` and `flcl INFO GET.SYSTEM` show version, build, platform, active licence and configuration
- a production licence ID carries the **customer name and the contract number** – if it says DEMO, the demo licence is in effect
- the licence is a singleton in the process – a change only takes effect in the next run

Before every ticket: run `flcl INFO GET.SYSTEM` and send the output with it. That replaces the first round of queries.

FLAMv5.21-42969

This slide is the preparation for every ticket. Whoever runs `flcl INFO GET.SYSTEM` has answered half of the questions back already.
The licence model first, because it is the most frequent commercial question in the room. **What you install runs at once -- with the demo licence.** It is functionally complete: every function, every platform, no time limit. Only the volume is limited, namely 1 MiB per file. And the sentence that goes with it and must not be left out: **the demo licence forbids commercial and production use.** It comes without warranty and without support, and it may contain functions that are still in a pre-release state. It is there for trying out and for testing, not for operation. For production use -- and for compatibility with FLAM4 data -- a **commercial licence** is acquired; it comes as a **package of its own** and is laid over the existing installation.
The software itself is not installed again, and that holds for the renewal too. How to recognise a production licence: the licence ID carries the customer name and the contract number. If it says DEMO, you are still working with the demo licence -- on the host the classic case is that the licence library is not in the STEPLIB. With a licence problem the order is: does a licence take effect at all, which one, and does it fit the platform.

AUTHORISATIONS VIA SAF

- from version **5.1.21** numerous additional checks against the System Authorization Facility can be enabled
- they run through a dedicated RACF class `$FLAM` with resources per functional area
- the checks are **optional** – only switch them on once the resources are defined
- this lets you control who may encrypt, decrypt or access keys

In depth: deck Security – SAF and SKMS.

Just place it in context, do not go deep -- the details are in the deck sicherheit-saf-skms.
The point for operations: the checks are optional and are switched on deliberately. Anyone who switches them on has to define the resources beforehand, otherwise jobs will stall.
The starting point was an exit for virus scanning; out of that came a general way of binding FLAM functions to the security infrastructure.

PITFALLS

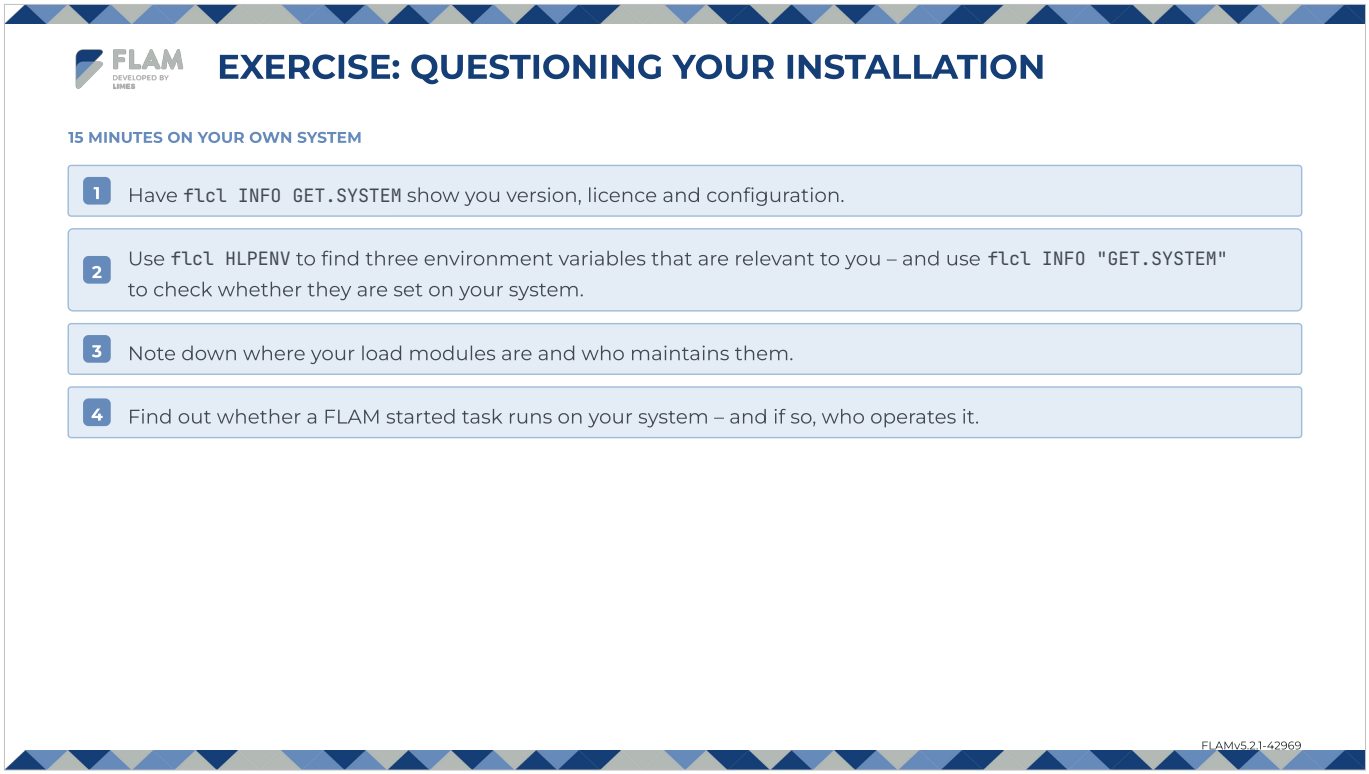
! **A property file takes effect even when nobody knows about it any more.**
With inexplicable behaviour, check `flcl INFO GET.SYSTEM` and the active configuration first, not the parameter.

! **No subsystem without a started task.**
The `SUBSYS=` in the DD statement fails when the service is not running – and the error does not sound like it.

! **The demo licence becomes noticeable late.**
Everything works until a file is larger than 1 MiB. So check the licence before a load test.

! **Environment variables from a shell do not travel into a job.**
On z/OS they belong in the configuration or on system symbols, not in a profile.

All four from operations. The first and the third cost the most time.

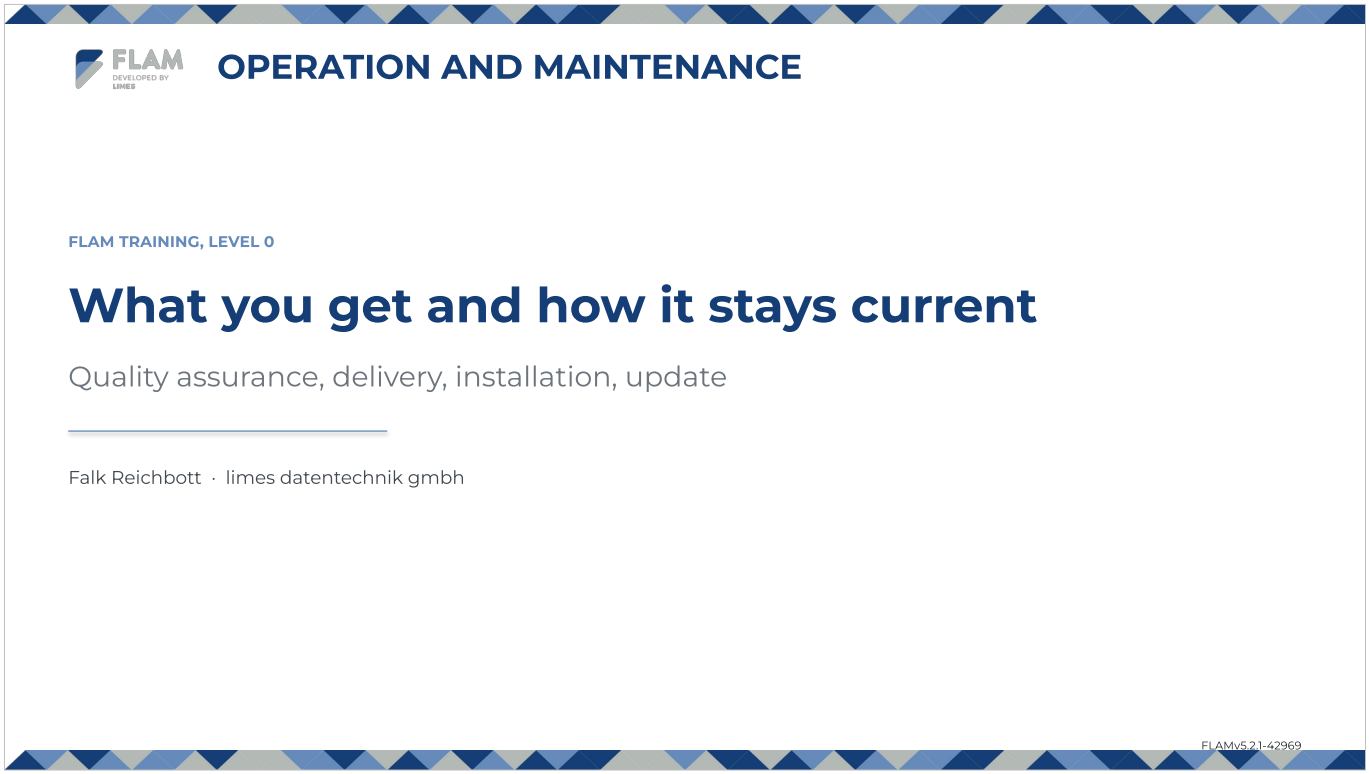


Fifteen minutes on their own system. The aim is that afterwards everyone can describe their installation without having to ask anyone.
Solution, so that nobody gets stuck:
1. `flcl INFO GET.SYSTEM` -- the output names version and build, the licence module, the owner, the configuration file in force and the effective environment variables.
2. `flcl HLPENV` lists all 131. The three that affect almost everyone are `LANG` (the CCSID -- on z/OS from `&FLANGLC`, `&FLANGCC`, `&FLANGCS`), `FL_DEFAULT_CHAR_MODE` (the error behaviour of the character set conversion) and `FL_ZEDC_SUPPORT` on z/OS. The distinction matters: `HLPENV` says what is possible, `INFO GET.SYSTEM` says what is set.
3. On z/OS it is the library in the `STEPLIB`, usually `FLAM.LOAD`; under Unix the installation directory together with `LD_LIBRARY_PATH`. As a rule it is maintained by systems engineering, not by the business department -- and that is exactly what the exercise is meant to clarify.
4. A started task is only needed for the subsystem, the LE-less interfaces and zIIP. Anyone who uses none of those needs none -- that is the most common wrong assumption.

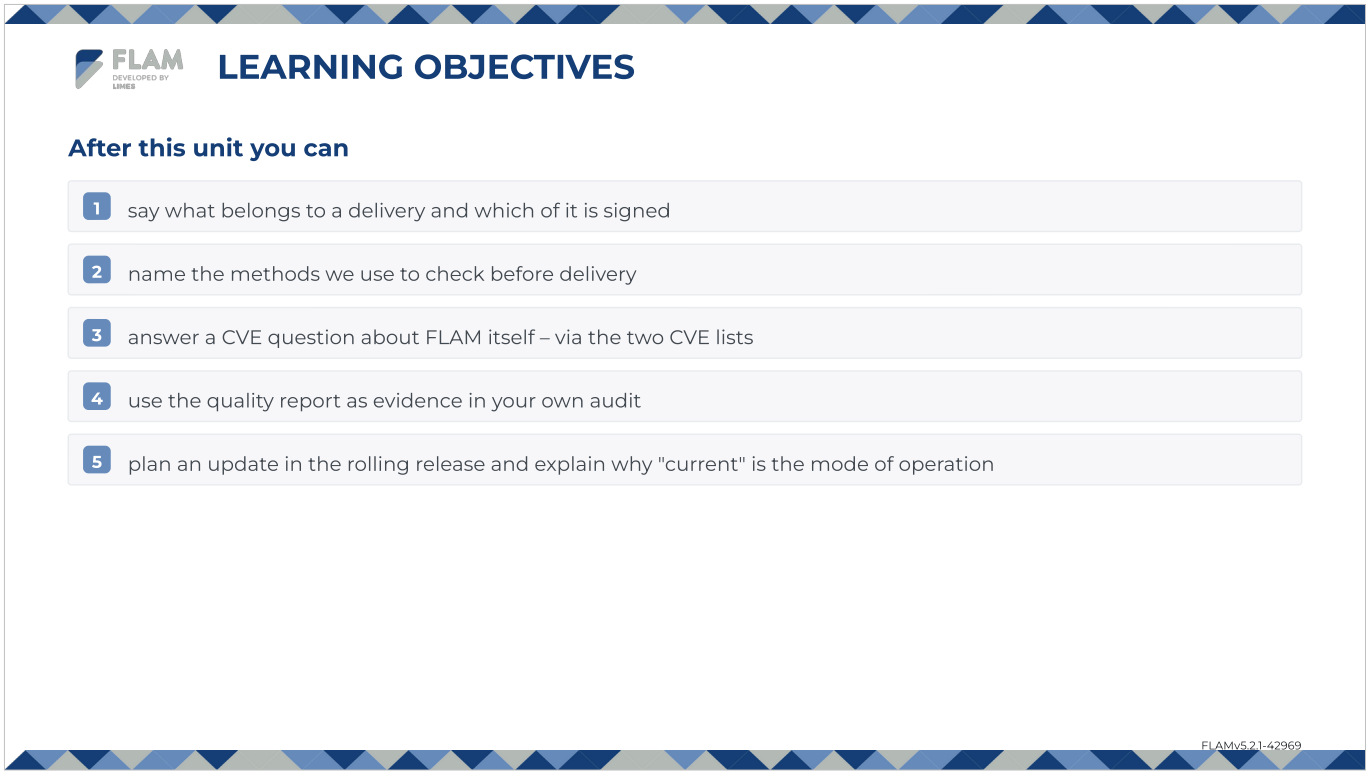
WHAT TO TAKE AWAY

- five configuration sources with a fixed precedence – the later one wins
- 131 environment variables, all described in the tool: `flcl HLPENV` – what is in effect is what `INFO "GET.SYSTEM"` shows
- on z/OS: load modules, FLAMPAR, started task for the subsystem, LIBPATH for USS
- `flcl INFO GET.SYSTEM` before every ticket – version, licence, configuration
- SAF checks are optional and have to be prepared

Go through the learning objectives. Next comes operations and maintenance -- that is, what we deliver, how we verify it and how you update.



Fifth unit of level 0. The audience is operations, not purchasing -- so this is not an advertisement for our QA but the question: what do you get, what is covered by the checks, and how do you update. The sentence that should stick: there is an answer to every CVE, and it is in one of two lists.



Objectives three and four are the ones asked about in audits. Objective five is the operational one.

WHAT BELONGS TO A DELIVERY

- the **installation package** per platform – the result of a single, automated chain; every real package format (MSI, Unix packages, SMP/E) is **signed**
- the **documentation in the package**, produced by the software itself and therefore matching the state – since 5.2.1 **this training material** is in there as well
- the **SBOM** and, since 5.2.1, the **CBOM** – Software and Cryptographic Bill of Materials
- the **quality report** with the findings of the check tools and their assessment
- the **CVE lists** for the external libraries, with a justification per entry

All QA measures lead, as one atomic function, to a signed installation package. Nothing is assembled by hand.

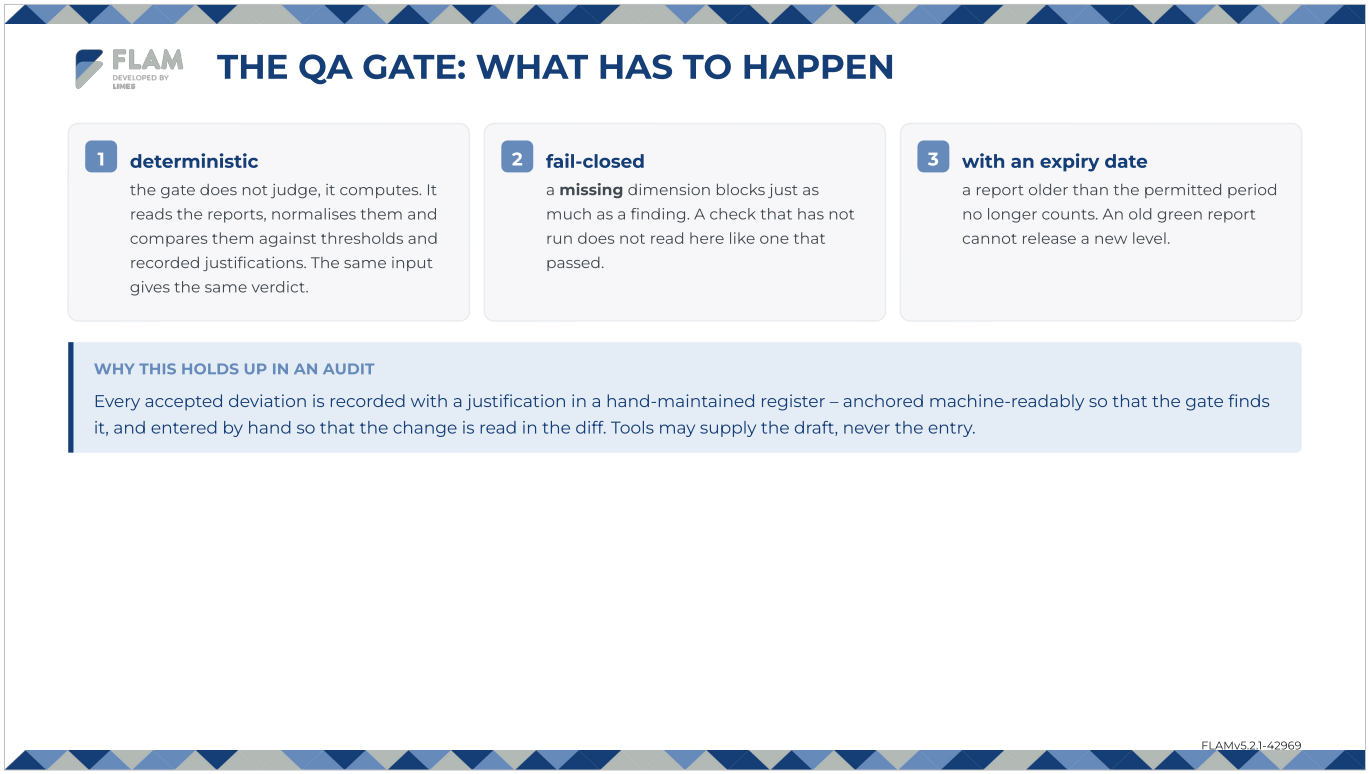
On z/OS the route is SMP/E, so the delivery is signed there too. The only unsigned forms left are the old transfer formats without a package structure - tarballs and XMIT - which we still offer solely because customer processes rely on them.

FLAMv5.2.1-42969

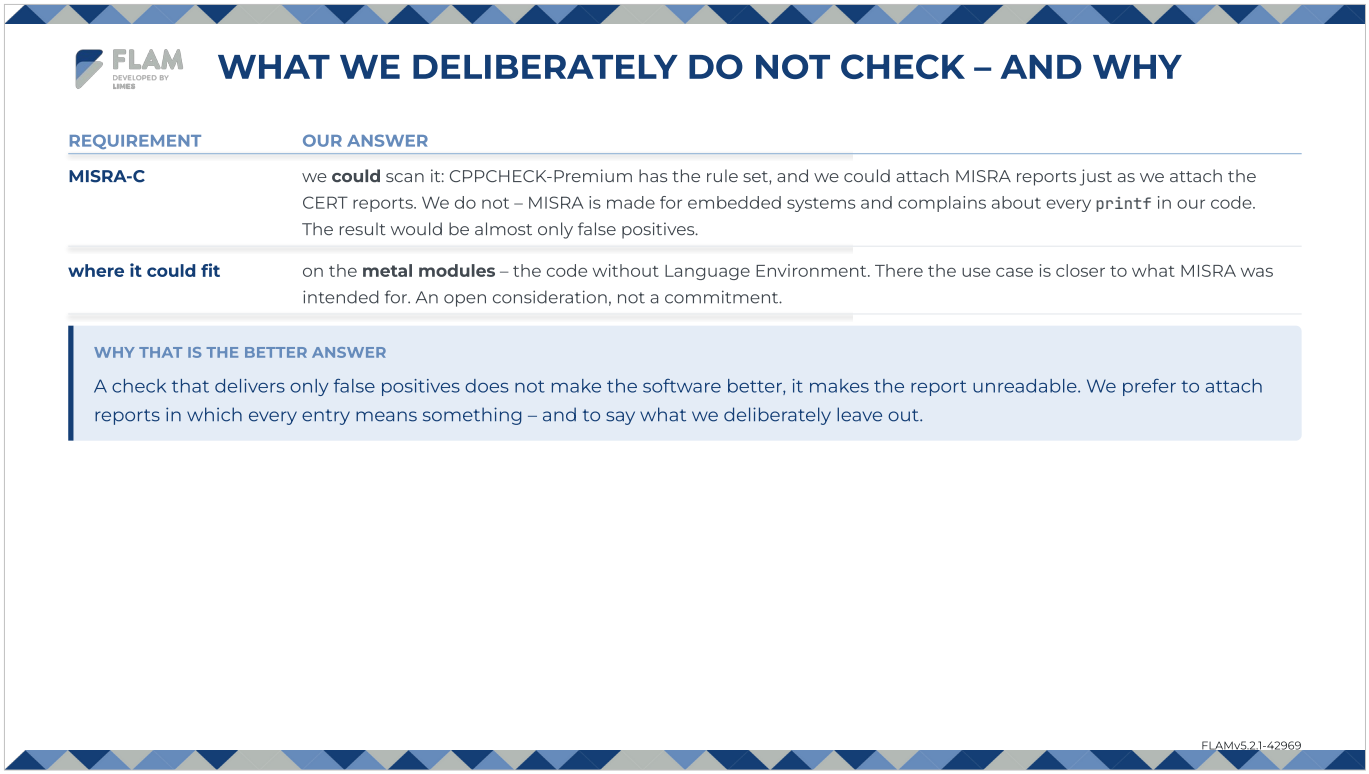
The core: all QA measures have to lead, as **one** function, to **one** signed installation package. That is not a marketing statement but the requirement customers from regulated industries put to us -- and the reason why nothing here is assembled by hand. Important for production: the documentation is in the package. That way it matches the installed version, even when a newer one is already online. And since 5.2.1 that holds for the training material as well: the deck you have in front of you is delivered with it. Anyone reading the training up a year later has the state that belongs to his installed version.



Do not present this as a parade of achievements but as the answer to the question that comes up in the audit: which procedures, and how evidenced.
All the names are in the quality report, with the respective findings and their assessment. Anyone in the room who comes from internal audit will want to photograph this slide.
zACS is the point that convinces a mainframe customer. For that you have to explain briefly what it is actually about -- the room often holds people who have never needed z/OS storage protection.
Storage protection on the mainframe works with keys. An application runs in **key 8**, the system in **key 0**. You write a started task whenever you have to finish writing a piece of the operating system for IBM:
you need authorisation and have to go to key 0 -- that is the mainframe equivalent of root or kernel privileges.
And from that follows the discipline this is about: an STC may only go to key 0 or authorise itself when nothing can go wrong any more. The step has to run atomically, and immediately afterwards you are a normal application again. Anyone who stays authorised and then crashes leaves behind a state in which critical accesses are possible.
That is exactly what zACS targets: it penetrates the started task and tries to break it while it is in the authorised state, in order to take the authorisation over. That forces a style of programming that you need nowhere else -- check every pointer with `VSMLOC`, check every length before anything is written.
To place it in context: this is **not** a static analyser but a dynamic vulnerability scanner. In kind more like Valgrind than like CPPCHECK -- it observes a running system instead of reading source code. And the statement is accordingly: our STC withstood the runs. That is a test result, not a guarantee.
The column "measured against" is more important than the tool names: it says that every dimension has a threshold or a rule it is measured against. A tool without a criterion is worthless in an audit.



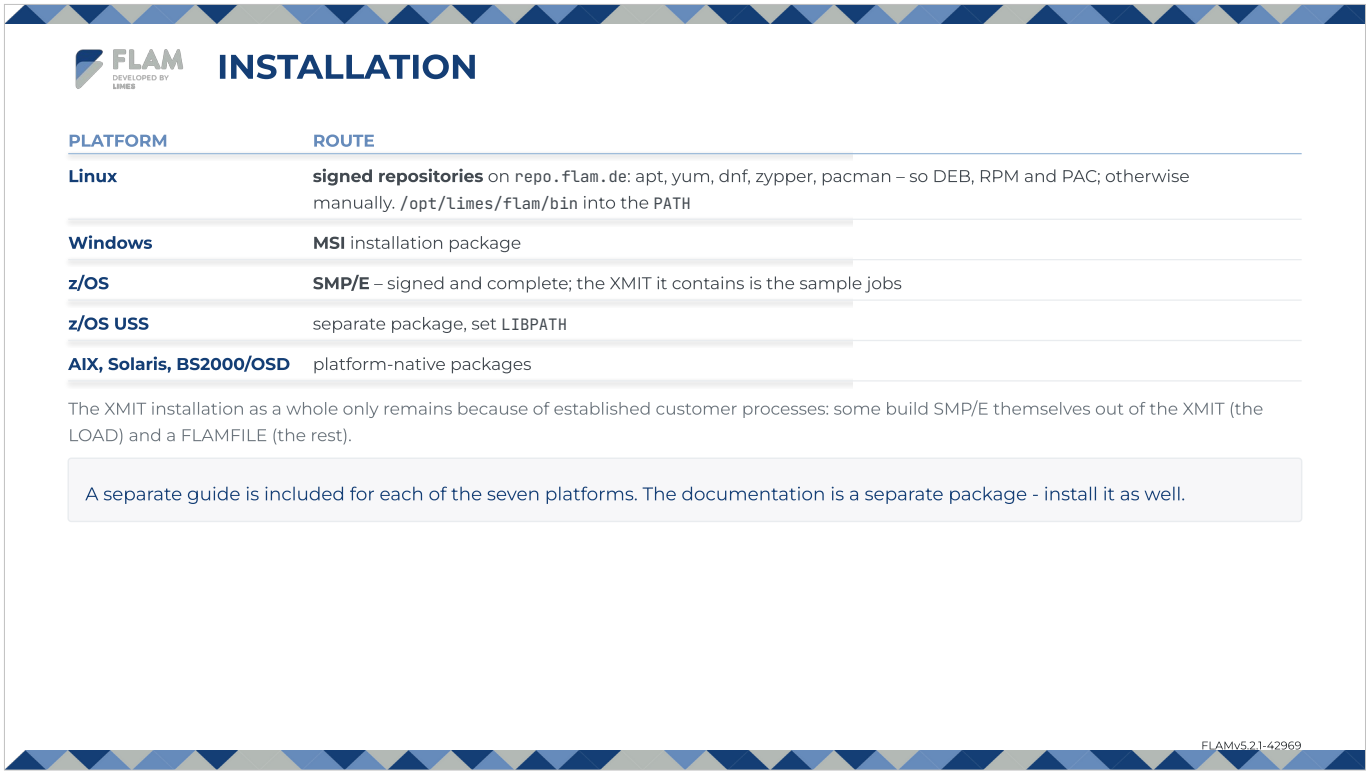
This slide is the difference between "we check a lot" and "we can prove it" -- and it is the slide that actually satisfies an auditor.
The core in one sentence: the delivery is released by a deterministic gate, not by a person reviewing reports.
Stress the three properties one by one.
First, deterministic: the gate itself contains no AI and no judgement. It reads reports, normalises them and compares them against thresholds and recorded justifications. The same input twice gives the same verdict twice.
Second, fail-closed: a missing dimension is an error, not silence.
If a report is not there because a run failed or the test infrastructure was unreachable, then the gate says MISSING and blocks.
That is the point at which most QA chains fail in practice -- a check that did not run otherwise reads like one that passed.
Third, freshness: a report that is older than the permitted period does not count. Otherwise an old green report gives a green verdict for a new state.
And the justifications: every accepted deviation is recorded in a hand-maintained register, with a machine-readable anchor and the justification text beside it. Tools may propose drafts, but the entry is made by hand -- so that the change is visible in the diff and gets read.



This slide comes from a real customer requirement: MISRA-C was on a requirements list, and the question comes up regularly.
The answer is not an excuse but a delimitation: MISRA-C is made for embedded systems. Applied to our code, it complains about every `printf` there is. The rules simply do not fit a tool that works with the operating system and with files -- what comes out of it is practically nothing but false positives.
The technical part of the answer matters, because it shows that it is not a question of effort: we could scan for it. CPPCHECK-Premium has the MISRA rule set, and we could ship MISRA reports just as we ship the CERT reports. We do not do it, because a report made of thousands of false positives is not a statement -- it hides the real findings.
Also say where it could make sense: on the metal modules, that is, the code without Language Environment. There the use case is closer to what MISRA was meant for. That is an open consideration, not a commitment.
The closing sentence is the one that carries in an audit: a check that delivers only false positives does not make the software better, it makes the report unreadable. We would rather deliver reports in which every entry means something.



This is the slide that ends an audit question in two minutes -- and the most common question from regulated institutions.
Every CVE that OSV knows for one of our external libraries lands in exactly one of the two lists. There is no third option and no silence.
Name the example: we link libcrypto but not the TLS layer. A CVE whose fix only touches the TLS layer does not affect us -- and exactly that is in the list, with this justification.



Keep it short, the details are in the instructions. The point for the training is the breadth: seven platforms, and the FLAMFILEs are interchangeable with one another.
On Linux it is more than "a package": there are **signed repositories** on `repo.flam.de` -- apt for Debian-based systems, yum, dnf and zypper for the RPM world, pacman for Arch. Once you have imported the GPG key, you update FLAM afterwards like any other system package. For an operation that manages package sources anyway, that is the most convenient route -- and the signature check comes with it for free.
On z/OS **SMP/E** is the route, and the package is signed and complete.
The XMIT inside it contains only the sample jobs for SMP/E itself.
We keep the XMIT installation as a whole going for one reason only: **the established processes of our customers.** Some build their SMP/E installation themselves out of two parts -- the XMIT contains the LOAD library, a FLAMFILE the rest. Anyone working that way should be able to keep doing so.
The last point: the documentation is a separate package. Anyone who leaves it out reads the wrong version online later.



The part that produces resistance in operations: there is no fix line. A defect is rolled out with the next version.
What you get in return is backward compatibility -- which is why an update is no risk for the existing data. A FLAMFILE from the eighties is readable today.
Practical advice: let a version run ahead in the test system, but do not try to skip versions. 13 months is the window.
And now the qualification of the sentence "there is no fix line", because it often answers the question really on people's minds: **there is a last-stable branch alongside the master.** We can build a fix in there and ship it as a new build for the last revision -- so the customer does get their fix without having to go through a version jump.
Technically this works in both directions: either fix on last-stable and merge into the master, or fix in the master and cherry-pick onto last-stable. We decide what fits case by case.
For the mainframe there is in addition the fast route: **a patch as an XMIT load**, when only one load module has to be swapped. That is the answer to "we cannot get a whole installation through release approval within a week".
The reason this works at all is continuous integration. And from that follows a commitment you can state without hesitation, because it holds: **if a customer wishes for a new parameter on Wednesday or reports a bug, it is available on Monday.** A new component takes about a month. That is the real advantage of the rolling-release line -- not that there are no fix releases, but that you hardly need them.

WHAT OPERATIONS NEED DAY TO DAY

MEANS	WHAT IT DELIVERS	WHEN
return code	the traffic light: good, warning, error	always, in every job
report	what was processed and what was not, per member	when a run has to be explained
logging	the flow in detail, graded	selectively during troubleshooting, not permanently
`flcl INFO GET.SYSTEM`	version, licence, configuration, environment	before every ticket

Logging is powerful and writes a lot. Switch it on selectively, do not run permanently at the highest level.

FLAMv5.21-42969

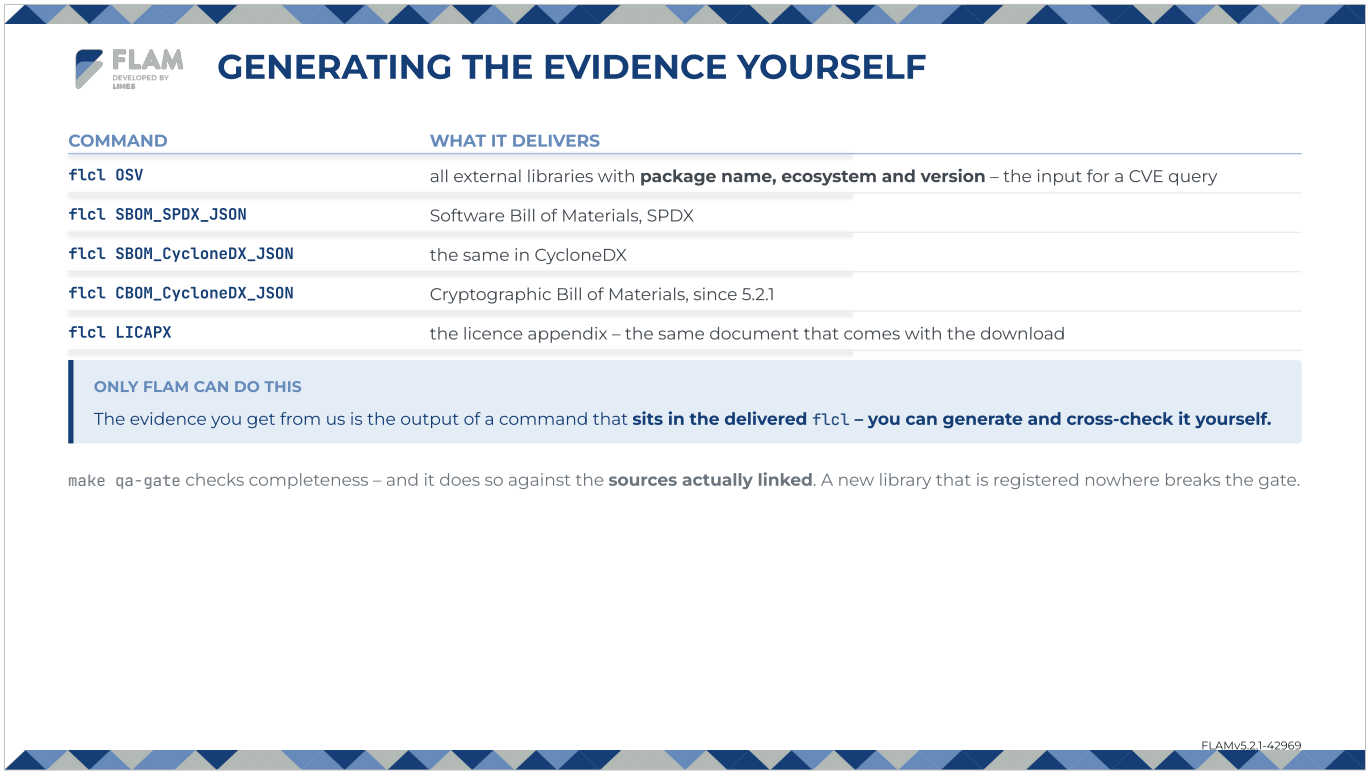
Three things that make the difference between a ticket and solving it yourself.
The return code is the traffic light, the report the justification, the logging the history. Anyone who knows all three needs us less often.
Give a warning about the logging: it is powerful and, set wrongly, can write a lot -- do not run permanently at the highest level.

WHEN YOUR AUDITOR ASKS

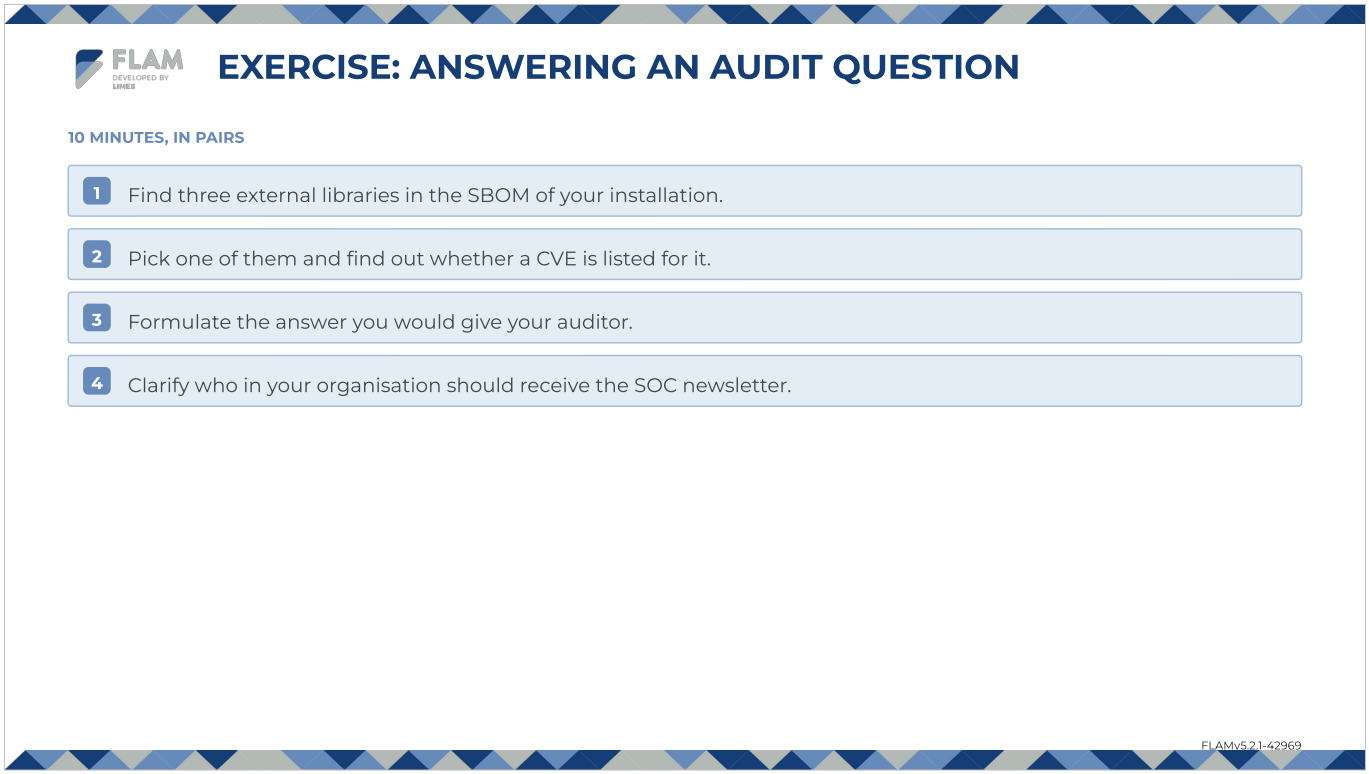
QUESTION IN THE AUDIT	WHAT YOU ANSWER FROM
Which verification methods does the manufacturer use?	quality report, with findings and their assessment
Which components are included?	SBOM per delivery
Is CVE-XXXX-YYYY relevant?	AFFCVE or NAFCVE, with a justification
How do we learn about vulnerabilities?	SOC newsletter following the requirements of BSI and NIST
Who operates the software?	You – we have no access to your systems
Is there a quality management system?	ISO 9001 certified

FLAMv5.21-42969

This slide is a gift to the participants: they can use it to work in their own institution.
Important and honest: we are not a data centre and have no access to customer systems. The customer operates the software on their own responsibility. That is not an excuse, it only changes which controls are applicable.
The SOC newsletter is the answer to the vulnerability question from DORA and NIS2 -- have them add their address.



This slide is a small gift to the auditor in the room, and almost nobody knows about it: **the tools with which we generate our evidence are inside the shipped `flcl`.** They are hidden commands -- they are not in the help, but they work, and customers are allowed to use them.
We use them ourselves to determine SBOM, CBOM, LICAPX, ABOUT, VERSION and the OSV list.
That is the point that creates trust: the evidence you get from us is not handwritten but the output of a command that you can run yourself.
`flcl OSV` prints the list of all external libraries with **package name, ecosystem and version**.
Those three details are exactly what you need for a CVE query against an OSV server -- the ecosystem is the part that most SBOM tools leave out and that is what makes the query usable at all.
The SBOM comes in **two formats**, SPDX and CycloneDX, the CBOM in CycloneDX. Anyone who needs a third format can build it themselves from the OSV list.
`flcl LICAPX` produces the appendix with the open-source licences -- the same document that comes with the download.
And the point that carries the completeness: **`make qa-gate` checks that nothing is missing** -- no licence forgotten, LICAPX complete, OSV lists and SBOM complete. And that check is based on the **actually linked sources**, not on a maintained list. A library that is added and never entered anywhere breaks the gate.



Ten minutes, in pairs. The effect is that the participants realise: they can answer the question themselves without calling us.

WHAT TO TAKE AWAY

- one automated chain, one signed package – documentation, SBOM and quality report are part of it
- we check statically, dynamically, on memory, on coverage and on vulnerabilities
- there is an answer to every CVE: AFFCVE or NAFCVE, with a justification
- rolling release plus backward compatibility – a 13-month planning window
- return code, report, logging and `flcl INFO GET.SYSTEM` are the tools of operations

Go through the learning objectives. That completes level 0; the three access routes follow in detail.

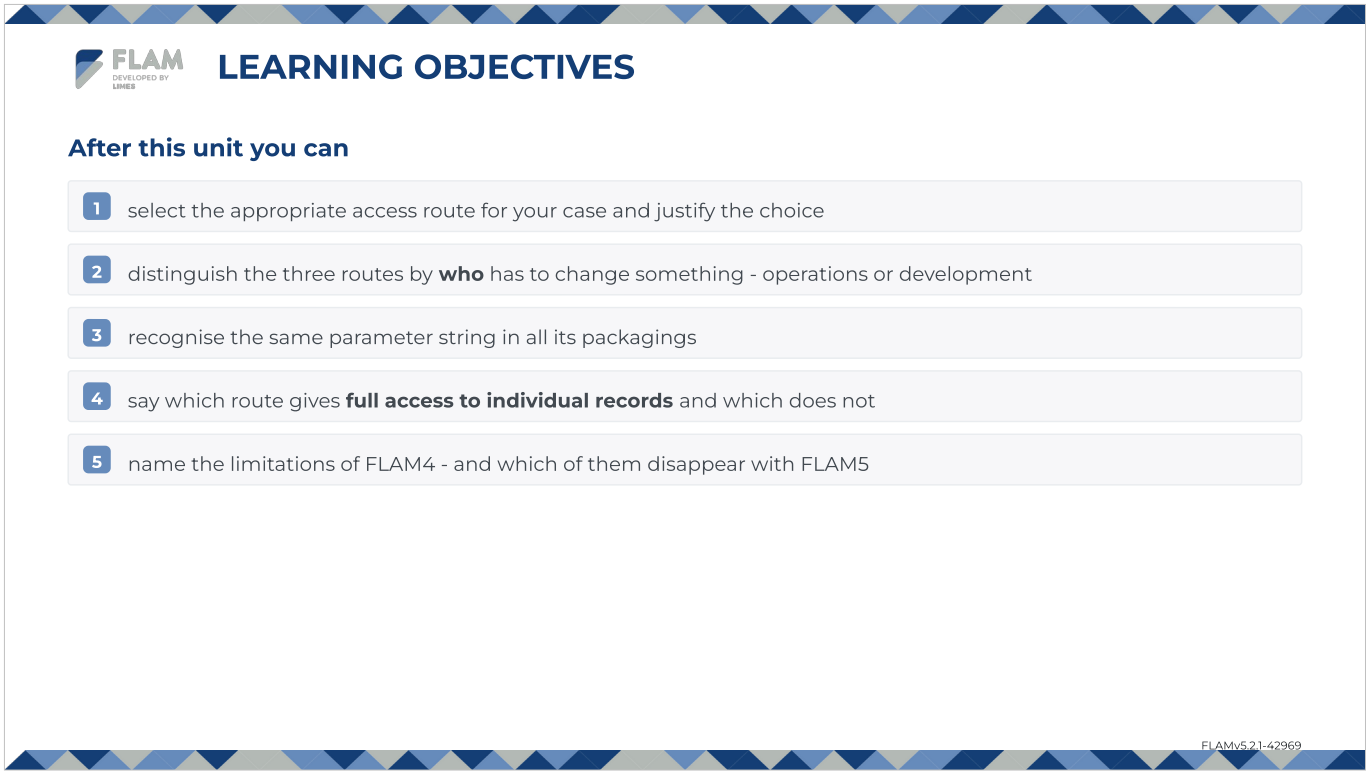
FLAM TRAINING, LEVEL 0/1

One parameter string, three wrappings

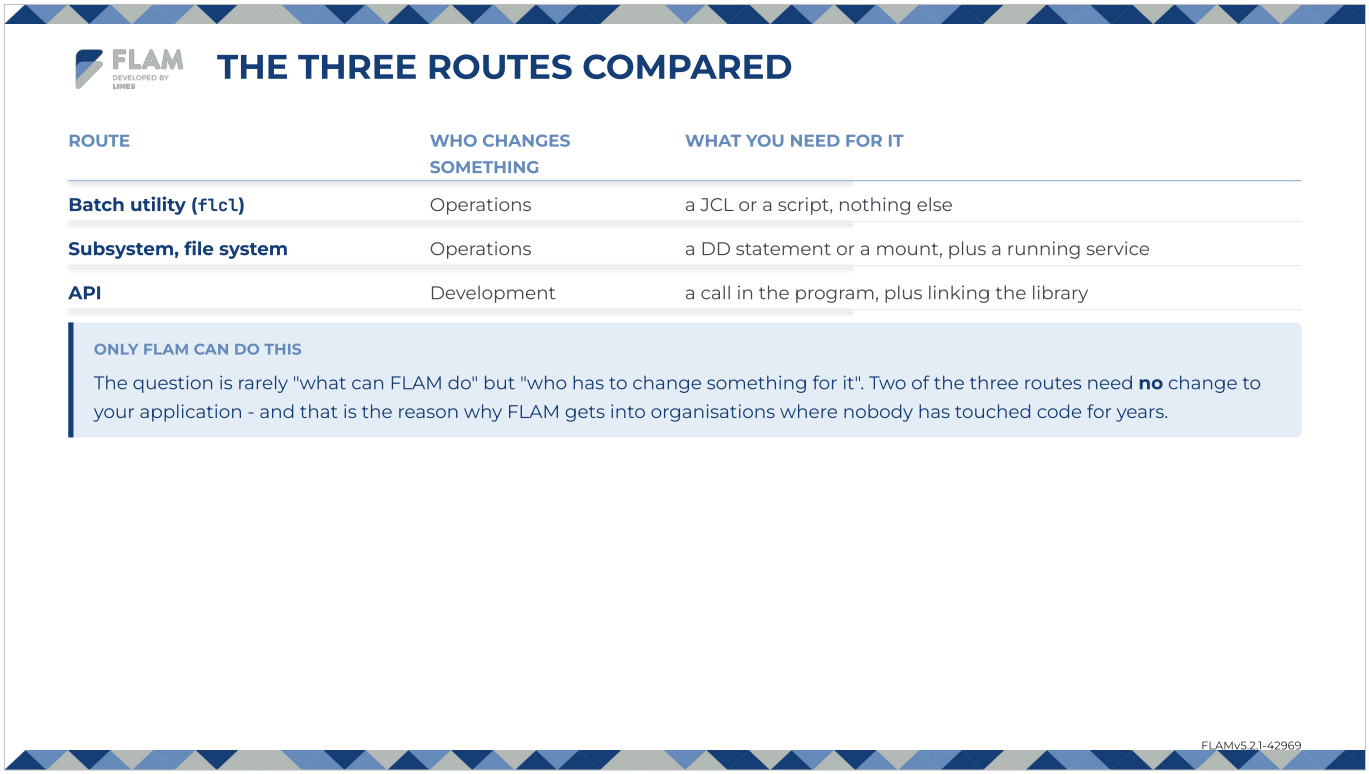
Batch, subsystem and API compared - and what distinguishes FLAM4 from FLAM5

Falk Reichbott · limes datentechnik gmbh

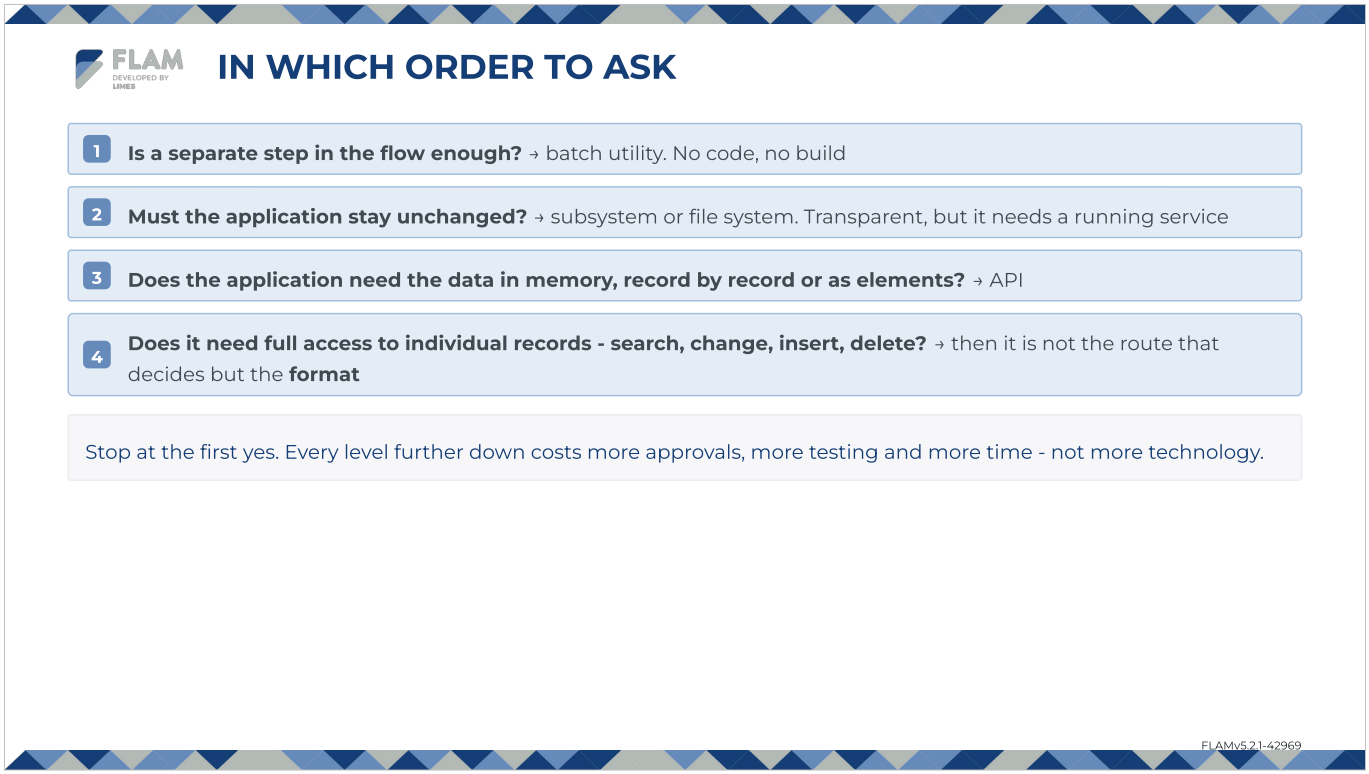
This unit comes straight after the basics and is the link between them:
there the participants learned that there is one parameter string and three ways. Here they see what distinguishes the three ways and how the same case looks on each of them. After this, each of the three following units covers one way in detail. The arc: the criterion, then one case solved four times, then the question that comes up most often in the room -- who can actually change individual records.



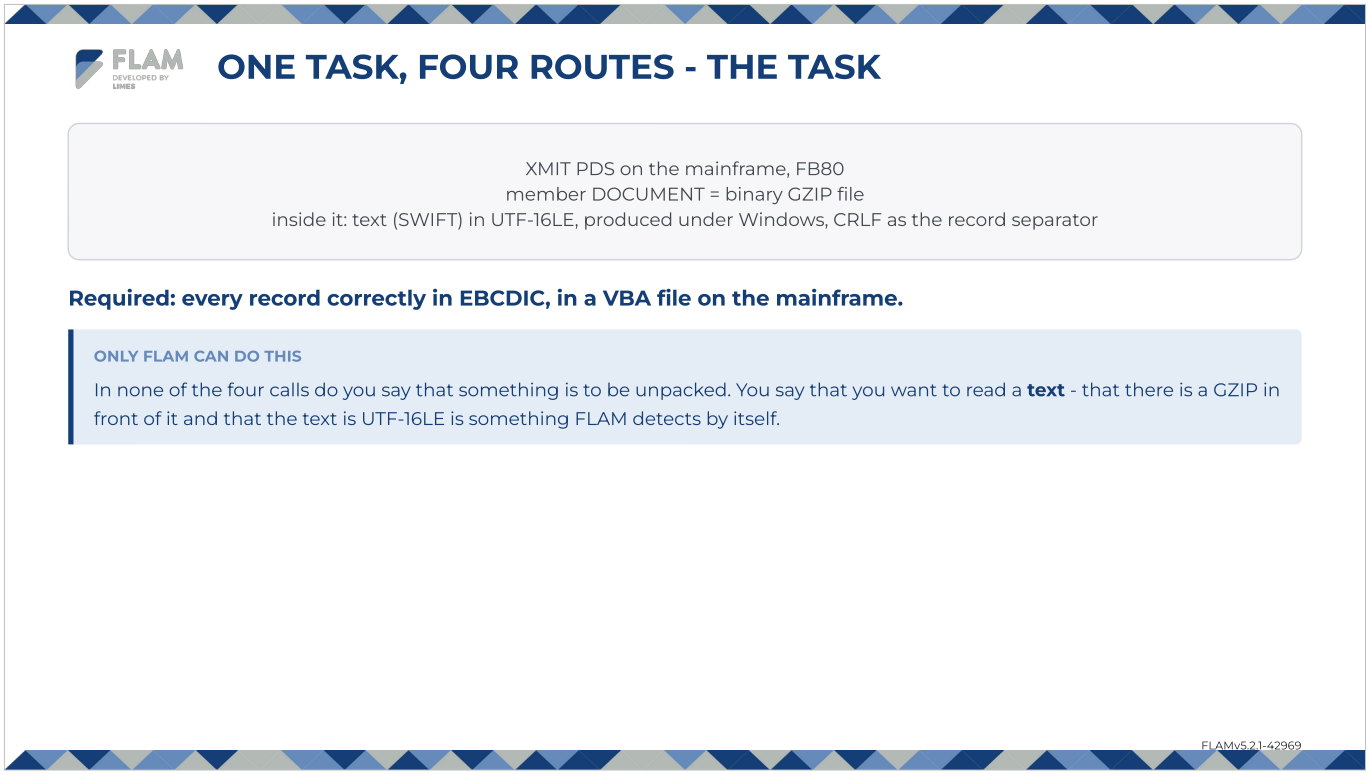
Objective 1 is the working objective: in the end everybody should be able to justify the route for their own case. Objective 4 is the one that prevents the most wrong expectations.



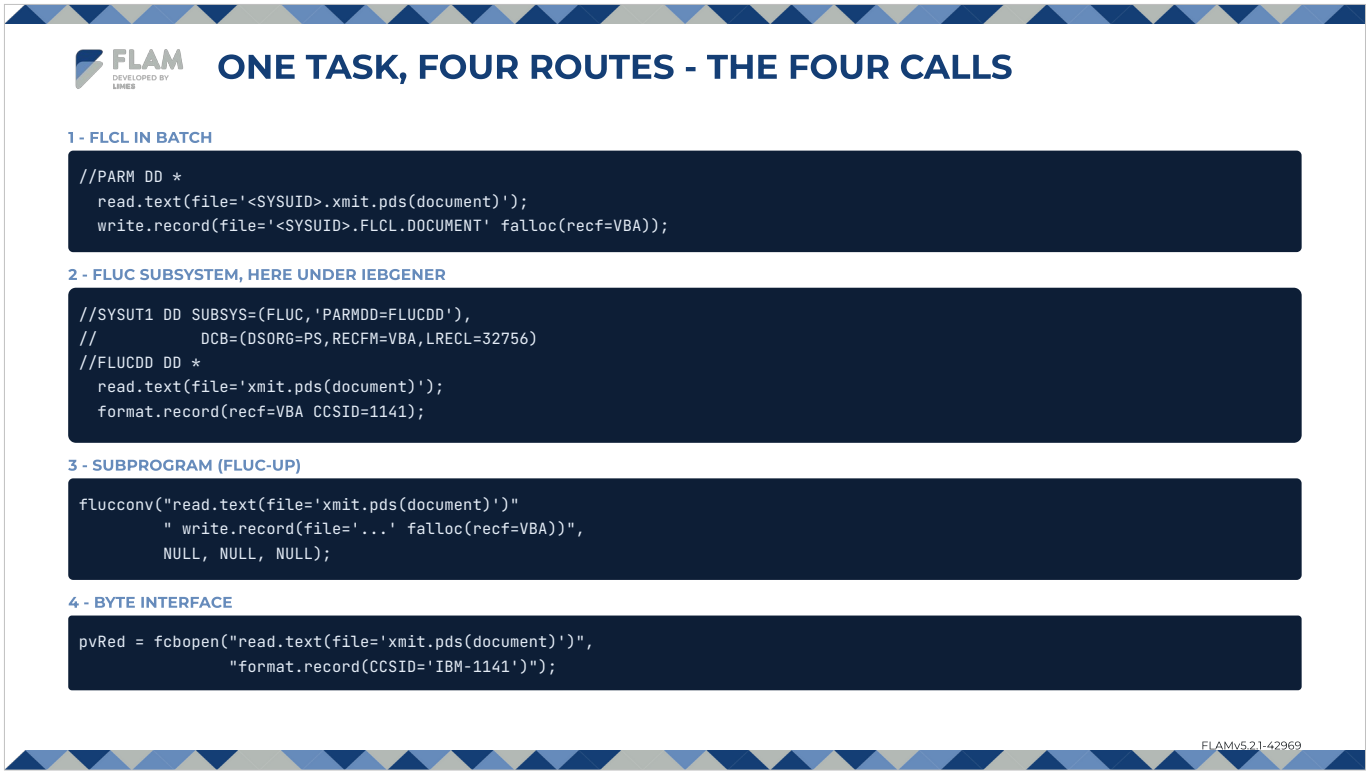
The table everything revolves around. It used to appear once in every level-1 deck; here it appears once, properly.
The decisive column is the second one: **who** has to change something. That is the actual question in almost every project, because it decides effort, approvals and schedule -- not the technology.
Batch and subsystem are operations topics: a JCL, a script, a DD statement, a mount. The API is a development topic: source code, binding, test, approval.
From that follows the order in which you ask in a project: does it work in batch? If not, does it work transparently via the subsystem? If not, then the API. Whoever starts the other way round often builds for weeks in the wrong place.
The third column is the one operations asks about -- what has to be running for it.



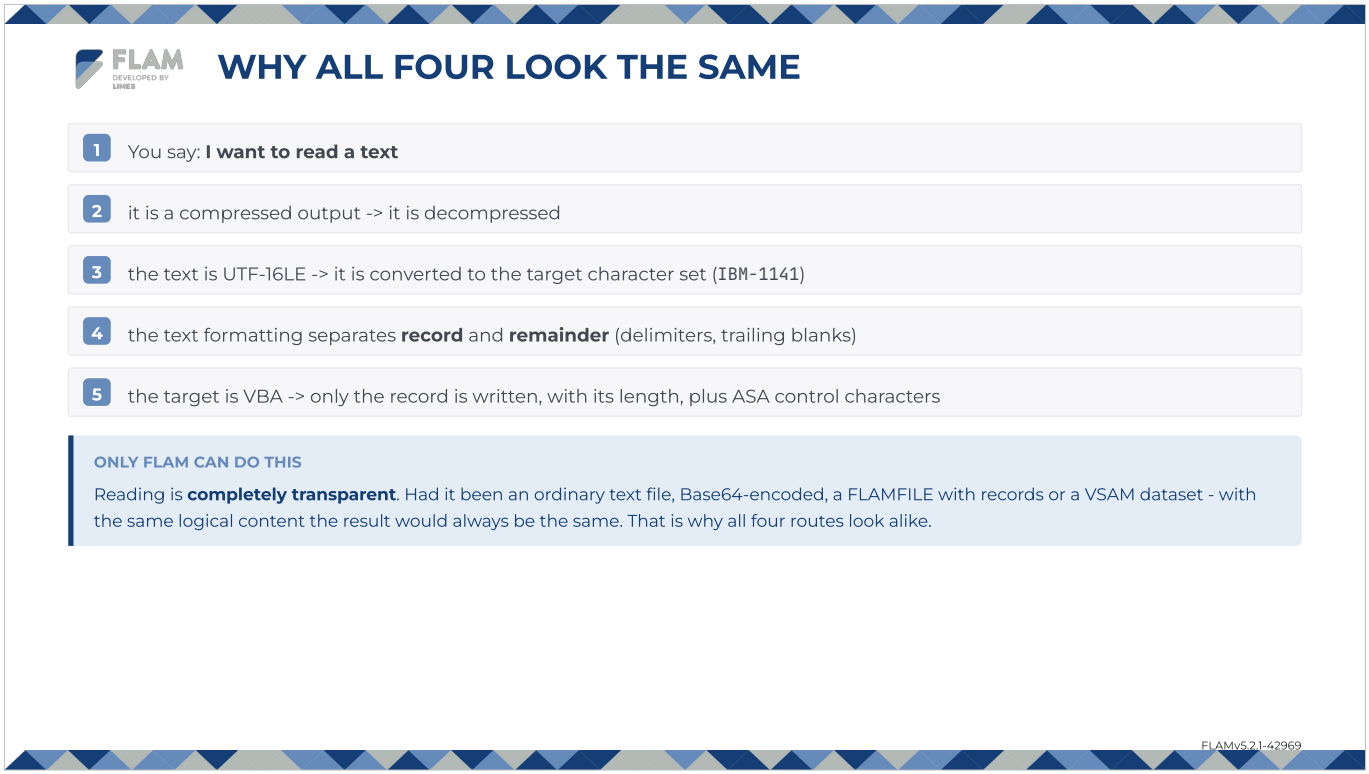
The practical slide for a project conversation. Go through it from top to bottom and stop at the first yes.
The reason for the order is not technical elegance but effort:
every step further down costs more approvals, more testing and more time.
Point four is the exception to the order: whoever needs full access to individual records does not have to start at the top but can jump straight to the last line. See the slide after this one for that.



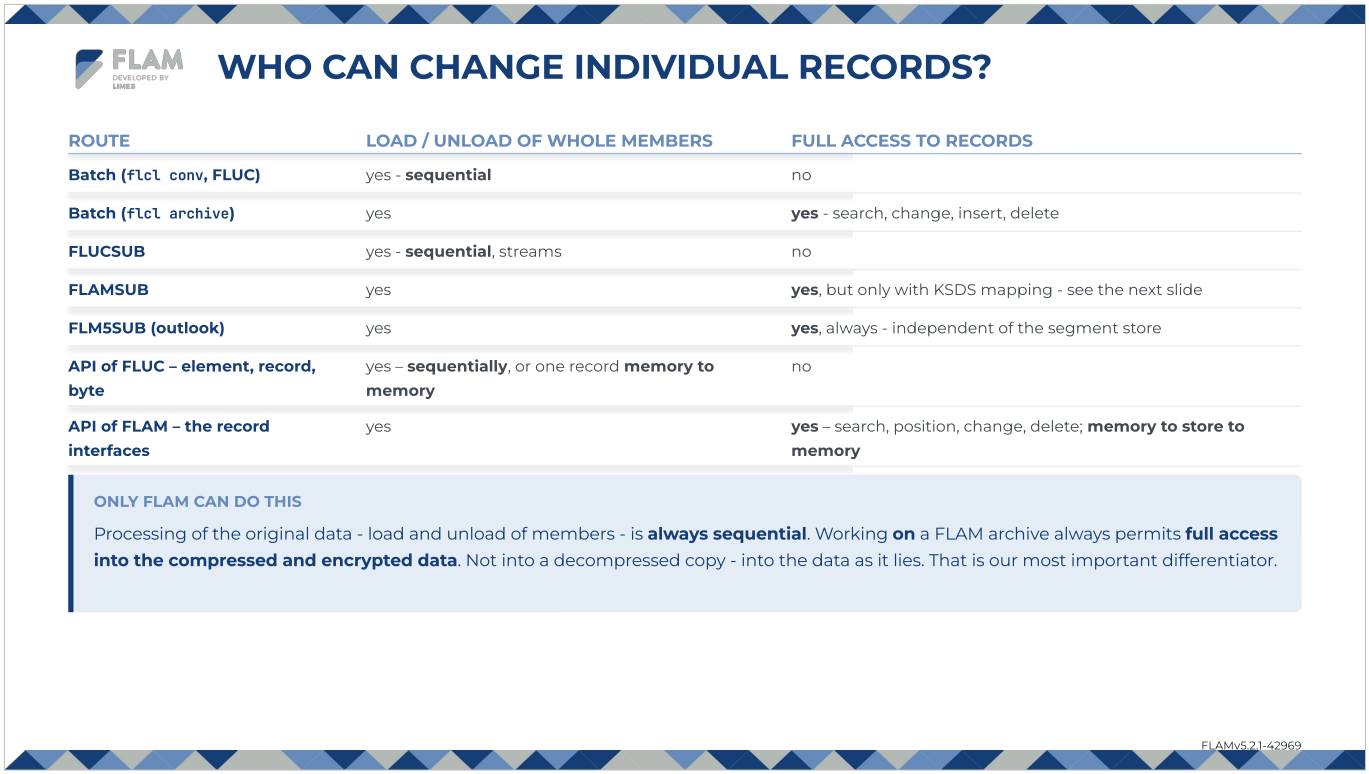
The best example we have: the same case is solved four times, and the solutions differ only in the packaging.
Read the task out slowly, it is deliberately unpleasant: in an XMIT PDS on the mainframe, record format FB80, there is a member DOCUMENT. That is a **binary GZIP file**. Inside it is a text in SWIFT format, encoded in **UTF-16LE**, produced under Windows, with **CRLF** as the record separator.
What is wanted is: every record correctly in EBCDIC in a **VBA file**.
Let the participants think for a moment about how many steps they would need for it. The usual answers are three to five steps.
And then the resolution: it is **one** call, and nowhere in it do you say a single time that anything should be unpacked. You only say that you want to read a text.



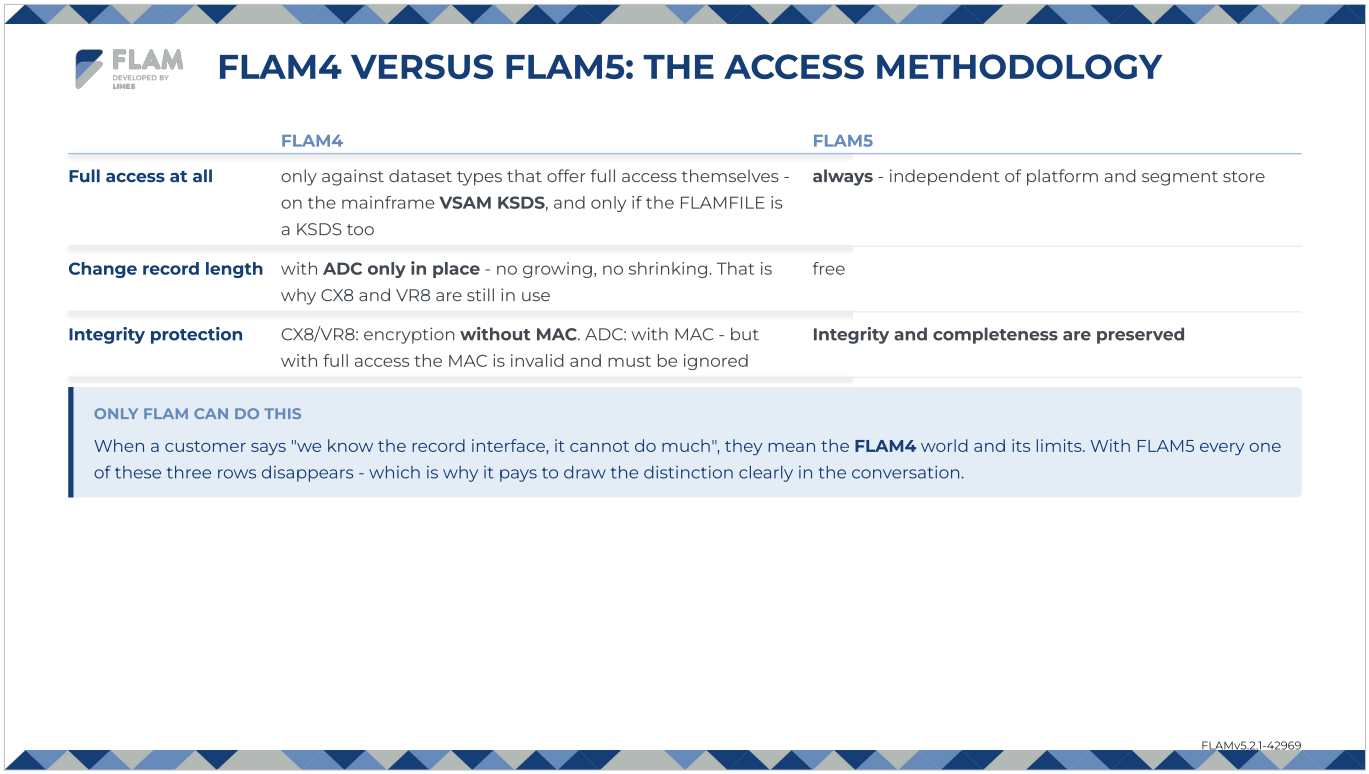
Do not read out all four. Point at what they have in common: in each of the four boxes the same parameter string appears, `read.text(...)`. What differs is exclusively the packaging.
FLCL in batch: one job step, `write.record` with `falloc(recf=VBA)`.
The subsystem: the trick is that here **IEBGENER** does the work -- an IBM utility that knows nothing about FLAM. The `SUBSYS=(FLUC,...)` sits on the input DD, and the parameter string lies in a DD of its own. Instead of `write.record` it says `format.record` here, because the target is not a file but the interface to the utility.
The subprogram: a `flucconv` call with the same string. Available as a DLL for C and C++ and as a load module library for COBOL and PL/I.
The byte interface: `fcbopen` with the two strings from the API unit, then an ordinary read loop.
The question that comes up here: why does it say `format.record` for the subsystem and the byte interface and `write.record` for batch? Because in the first case the data goes into the caller's memory and in the second into a file. That is exactly the distinction between file string and format string.



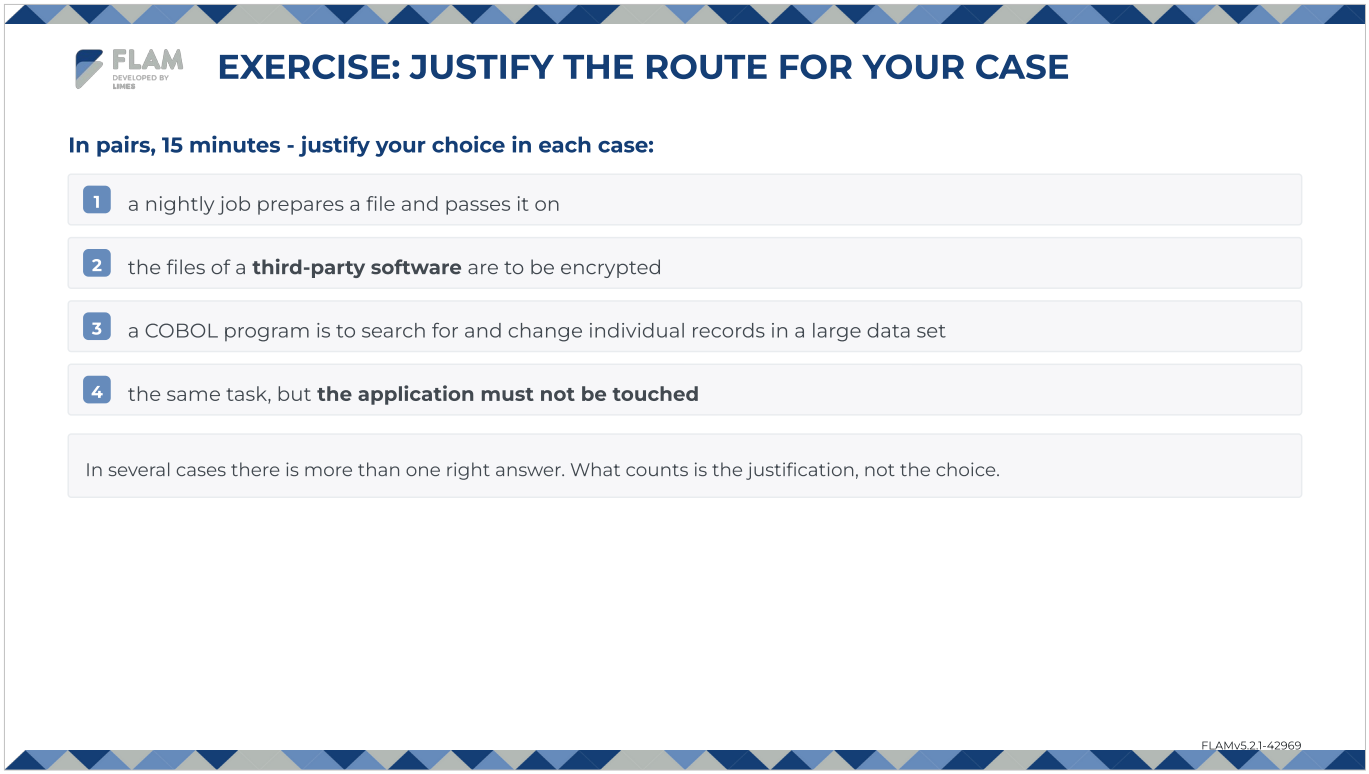
The resolution of the example, and the slide with the actual message of the whole of level 1. Tell the chain once from top to bottom: the user says that he wants to read a **text**. Everything else follows from that by itself.
We come across compressed output, so it is unpacked. We detect that the unpacked text is UTF-16LE, so it is converted -- into the target character set of the platform, here IBM-1141. The text formatting splits the data into pairs of record and remainder, where the remainder is the separator and any trailing blanks. Because writing is record-oriented into a VBA file, we write only the record with its length and put the ASA control character in front of it.
And now the sentence that is to carry the unit: the reading is **completely transparent**. Had the file been an ordinary text file with separators, or Base64-encoded, or a FLAMFILE with records, or a record-oriented VSAM data set -- the result would, with the same logical content, always be the same.
If there is time, bring in the addition from old slide 35: any number of further writes can be hung onto the same call. For instance the same data additionally as UTF-8 with 0x0A as GZIP via SSH onto a Unix server and as ZIP with CP1252 for Windows -- in one call.



The slide that resolves the most frequent wrong expectation -- and I had it wrong myself, which is why it is stated so plainly here now. The wrong short cut is: "full access to individual records is only given by the API". That is not correct. **The subsystem gives full access as well** -- FLAMSUB today under conditions, FLM5SUB in future without. That is the big difference between FLAM and FLUC: FLUC is conversion and therefore sequential, FLAM is an access method. The rule worth remembering, and it holds for everything that follows: First: **processing original data** -- that is loading and unloading members -- is **always sequential**. That is the FLUC route. A whole file in, a whole file out. Second: **working on a FLAM archive** always permits **full access into the compressed and encrypted data**.
Not into an unpacked copy -- into the data as it lies. The second sentence is our most important differentiator. If you take one sentence away from this unit, take that one. And because "the API" is too coarse: there too **FLUC has to be distinguished from FLAM**. The FLUC interfaces -- element, record, byte -- read or write **sequentially**, or they convert a single record **memory to memory** without a file being involved. That is the route for the online business. There is no full access there, and that is not a limitation but the nature of a conversion. The FLAM record interfaces, by contrast, work **against the archive** and can therefore do full access: **memory to store to memory**. So anyone saying "single record access through the API" always means the FLAM side.



Now the limitations of FLAM4, and you have to know them, because the large data sets at the customers are FLAM4 archives. The reflex in the room is often: "record interface, we know that, it cannot do much" -- because they know the old FLAM4 record interface and its limits.
First limit, the data set. Full access only works against data set types that offer full access themselves -- so on the mainframe only VSAM KSDS. And only if the FLAMFILE is a KSDS as well. You can replace RRDS, ESDS and KSDS one for one with a FLAMFILE if the latter is again a KSDS, ESDS or RRDS.
LDS would work too, but nobody needs it.
Second limit, the compression method, and that is the unpleasant one: with **ADC only update in place works**. No growing, no shrinking of a record. That is exactly why CX8 and VR8 are still in use at customers -- not out of habit but because more is possible there.
Third limit, the integrity protection, and here it gets subtle: with CX8 and VR8 there is encryption, but **without a MAC**. With ADC there is encryption **with** a MAC -- but as soon as you do full access, that is, change in place, the MAC is no longer valid and has to be ignored.
And then the contrast the slide is heading for: with FLAM5 all of that falls away. Full access always works, independently of platform and segment store. And integrity and completeness are preserved in the process -- that was the requirement that determined the design of FLAM5.
The aside for those interested in the technology: we do not support VSAM KSDS as a segment store and probably never will. Locally we use USS pathnames, that is, folder and file -- and the zFS underneath physically sits in a VSAM LDS, that is, in the fast, flat form.



15 minutes in pairs, then two cases in plenary.
The exercise is deliberately built so that the justification counts and not the choice -- in several cases there is more than one right answer.
Resolution:
1. A night job that prepares a file and passes it on: **batch**. No code, no build, no approval for an application change.
2. Third-party software whose files are to be encrypted:
subsystem or file system. You are not allowed to and cannot change anything.
3. A COBOL program that searches for and changes individual records in a large data set:
API with a FLAM5 archive. Via FLAMSUB it would only work with KSDS mapping and then without a valid MAC.
4. The same task, but the application must not be touched:
FLAMSUB today under the conditions of the previous slide -- or wait for FLM5SUB. That is the honest answer, and it is better than a false promise.

WHAT TO TAKE AWAY

- the same **parameter string** in all three routes - what differs is the packaging
- the question is **who has to change**: batch and subsystem are operations, the API is development
- stop at the first yes - every level further down costs more approvals and more time
- **Load and unload are always sequential; working on an archive always gives full access**
- the limits customers know from the record interface are **FLAM4** limits

The next three units each take one route in detail: batch utility, subsystem and file system, then the APIs.

The two sentences to remember are the first and the fourth point. After that the three following units each go through one route in detail.



The first of the three access routes, and the one with which eighty percent of customers start and stop. The arc: the call in three environments, moving the parameters out, evaluating the result, automating. The most important part is the evaluation, and that is where the one statement sits that has to land: completion codes 1 and 2 are success. Anyone treating them as errors aborts jobs that have run through.



Objectives three and four are the operationally important ones -- without them a FLAM stage in a chain cannot be monitored.

THE SAME CALL, THREE WORLDS

LINUX, AIX, SOLARIS, Z/OS USS – BASH

```
flcl CONV "read.text(file='ein.txt' ccsid='IBM-1141') \
           write.binary(file='aus.bin')"
```

WINDOWS – POWERSHELL

```
flcl CONV "read.text(file='ein.txt' ccsid='IBM-1141') `
           write.binary(file='aus.bin')"
```

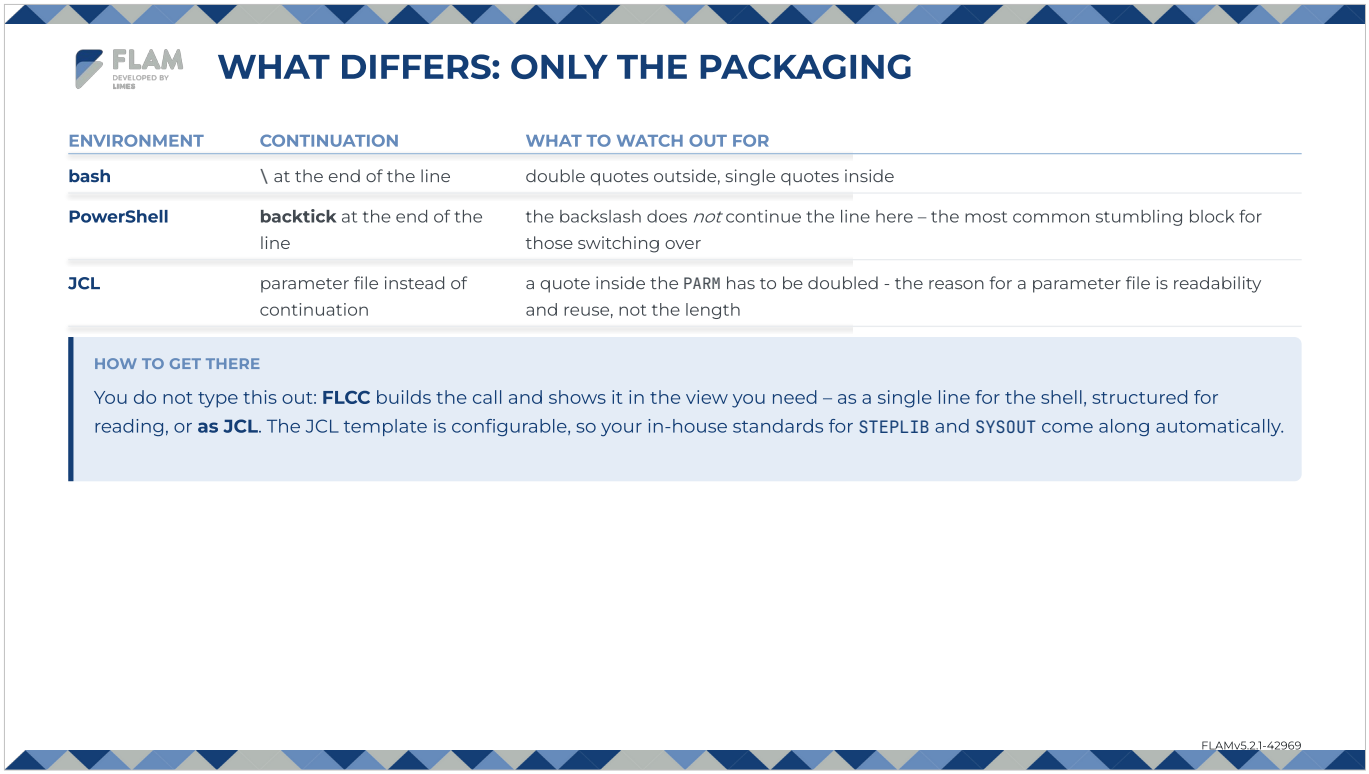
Z/OS – JCL

```
//CONV    EXEC PGM=FLCL,PARM='CONV MAXCC=-2'
//STEPLIB  DD DSN=FLAM.LOAD,DISP=SHR
//SYSPRINT DD SYSOUT=*
//FLAMPAR  DD *
  read.text(file='ein.txt' ccsid='IBM-1141');
  write.binary(file='aus.bin');
```

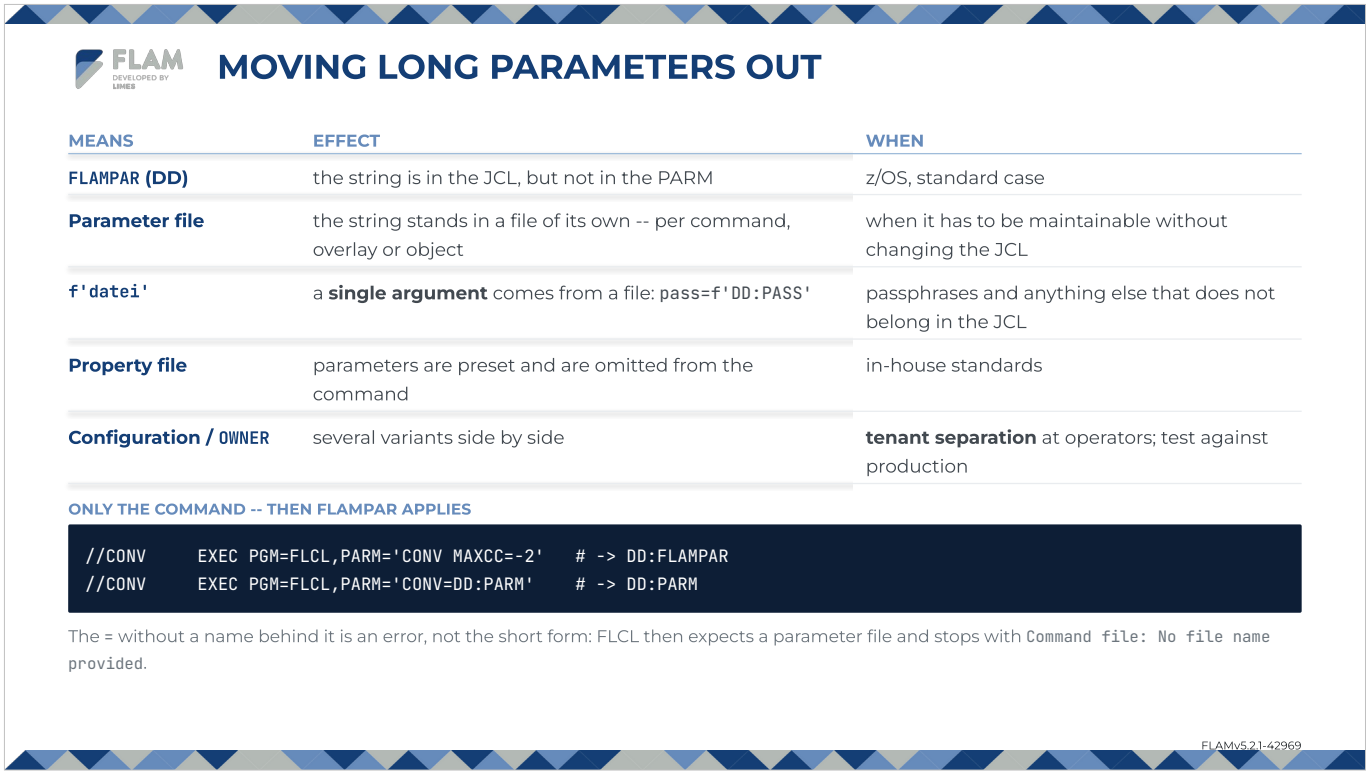
The parameter string is identical in all three cases.

FLAMv5.21-42969

The core slide of the unit. Three environments, one parameter string. Take it slowly and show the participants that all three boxes contain literally the same character sequence.
Everything that differs is the packaging of the respective environment -- not FLAM. Bash: double quotes outside, single ones inside, backslash at the end of the line for the continuation. PowerShell: the same, but the continuation is a backtick. Anyone coming from the Unix world stumbles here -- and anyone who copies the call out of FLCC gets it as a single line anyway and needs no continuation. JCL: the string is not in the PARM but in a parameter file. Point out the doubled apostrophes if the string is to go into the PARM after all -- that is the most frequent cause of alleged syntax errors, and it is JCL, not FLAM. If there is time: run all three once. The effect "this really is the same thing" carries the whole unit.



The reference slide for the previous one. These three lines are the whole difference, and they are the three most frequent support cases for newcomers. Bash: the backslash continues, single quotes inside are literal. PowerShell: the backtick continues, not the backslash. That is the classic one. JCL: a quote inside PARM has to be doubled -- which is why a serious string belongs in a parameter file and not in PARM. On the length, in case the question comes: **there is no limit on our side.** FLAM5 processes a parameter string of any length, and FLAM4 takes whatever z/OS passes at most. The old limit of 100 characters is history, and with `PARMDD` there is a second way anyway of reading the string from a DD card. So the reason for a parameter file is not the length but readability, reuse and the doubled quotes -- `FLAMPAR` or `DD:PARM`. And then the point that saves the participants the most typing: FLCC builds the call and gives it in the view you need. One line for the shell, structured for reading, or directly as JCL. The JCL template is configurable -- anyone with in house standards for STEPLIB, CEEOPTS and SYSOUT stores them in FLCC once and gets them in every generated job from then on. That is the bridge to the unit on the tools: FLCC is our interactive manual, and here it is the generator as well. Three defaults among the DD names that are hardly known and that make a JCL shorter: if no file stands at the `READ`, FLAM looks for `**FLAMIN**`, at the `WRITE` for `**FLAMOUT**`, and for a FLAMFILE additionally `**FLAMFILE**` -- the last one out of compatibility with the old FLAM utility. Whoever names the three allocations that way needs no file name in the command.



Production strings are long. Whoever writes them into the JCL changes the JCL for every little thing -- and needs an approval for it. So: FLAMPAR or a parameter file, and everything recurring into a property file. The JCL then stays stable. Two conveniences hardly anybody knows and that make the JCL shorter: First: **if the `PARM` holds only the command, FLCL reads the allocation `FLAMPAR` by itself.** `PARM='CONV'` is therefore enough when the parameters stand under `//FLAMPAR DD ` -- *that is how the examples of the manual have it too. The equal sign does* **not** belong there: `PARM='CONV='` says "parameter file", and the name behind it is missing. Run with 5.2.1: `Command file: No file name provided`, completion code 36 (Falk suspected it, 2026-09-16). Second: **the `DD:` in front of a DD name is optional.** Anything with only one qualifier is read as a DD name. `PARM='CONV=PARM'` and `PARM='CONV=DD:PARM'` are the same. On the owner the reason, because it is often missing: **it serves above all the separation of tenants at large operators** who work for more than one customer -- IBM and Kyndryl, T-Systems and comparable ones. Every tenant gets its own configuration under its owner, in the same system and without the settings getting in each other's way. Test against production is the simpler case of the same mechanism. And the third stage, which is often missing on this slide and is the most important one in practice: **not only the whole parameter string, a single string argument may come from a file too** -- `pass=f'DD:PASS'`. With that a passphrase stands neither in the JCL nor in the log nor in the process list. The operator `f'...'` applies everywhere a string is expected; `=>` applies to the parameter string of a command, an overlay or an object (Falk, 2026-09-16). Do not forget the order of precedence (see the Administration deck): the command line wins against everything before it. And the sentence to leave the slide with: **the goal is a JCL that does not change any more.** Everything that changes belongs beside it. And the question that belongs to it in operations: **what happens when the target is already there?** `FIOMODE` decides it -- `APPEND` is the default and appends, `OVERWRITE` writes over it, `STOP` aborts. `REMAIN` is the careful variant: an existing file stays untouched. And `falloc(RENEW)` is the other direction: the existing file is deleted so that it can be allocated anew -- the way when the DCB specifications have changed.



The slide that prevents a support case we see regularly. When **reading** it makes no difference whether a file is addressed statically through a DD statement or dynamically through its name. When **writing** it does. A static allocation gives the **DCB** -- record format, record length, block size. FLAM thereby gets no specification of how to shape the data, and then does the obvious thing: it tries to **restore the original**. That is exactly the case customers do not expect. An example: a text matrix lies in the archive. The original was an ordinary text file, so a byte stream with line separators. Writing it into an FB data set gives a byte stream again -- the separators stand in the data, and every record but the last is filled to the brim. Whoever expected records finds one long sausage of bytes. **The way out, when it has to be a static allocation:** give the record format at the `WRITE.RECORD` itself -- `recformat=` and `reclength=`. That is the **logical** entry: it says how the data is cut into records. The `FALLOC()` on the other hand describes the **allocation**, and with a DD statement there is none -- one would be describing something that does not happen. Up to 5.2.0 `falloc(recformat=…)` was the only way and acted as the logical entry as well; that stays valid, but it is no longer the right spelling.
Then FLAM knows that it is to write records -- the net record per record -- and not the shape of the original. If the original was a record data set itself, it fits anyway. **The better way is the dynamic allocation**, and the manual has it that way too: "In all other cases, we recommend using dynamic allocation when writing new files." The reason is not convenience but the space calculation: FLAM works through the first block completely in the open, so it knows the size of the original and the expansion or the compaction the data goes through. The space specifications follow from both. For a PS file `RELEASE` comes on top, so that unused space goes back at close; for a PDS FLAM adds a little and gives nothing back. Whoever allocates statically has to know all of that -- and usually guesses.

READING COMPLETION CODES CORRECTLY

CODE	MEANING	REACTION IN THE CHAIN
`0`	completely successful	continue
`1`	successful, but the log contains a relevant warning – a key that expires soon, for instance	**continue** – evaluate the log
`2`	successful, only the cleanup afterwards failed	**continue** – can be ignored
`4`	warning: some files were not processed	decide on business grounds, evaluate the report
`8`	error during execution	abort, save the report
`12 … 48`	execution was never reached: mapping, syntax, configuration, licence, memory	the call or the environment is wrong – not the data
`64`	fatal error	abort, involve support
`> 64`	Special Condition Code, set deliberately by the command for job control	evaluate as intended

1 and 2 are success. `MAXCC=-2` turns them into `0`. These are the **Completion Codes of the program** – not the return codes of the API calls from `FLMRTC.h`; those come in the unit on the API.

FLAMv5.21-42969

The most important slide of the unit for operations, and the one with the greatest potential for damage if you read it wrongly.
First the distinction that decides everything: these here are the **completion codes of the program**, as they stand in Appendix G of the manual. They are not the same as the return codes of the API calls -- those are in CINC/FLMRTC.h and come up in the unit on the API. Anyone who mixes up the two tables misreads every run. The structure of the table: up to and including 2 the job was successful. 4 means warning, some files were not processed. 8 means an error during the execution. Anything above 8 up to 64 means that it never reached the execution -- mapping, syntax, configuration, licence. 64 is the fatal error, and above 64 are the special condition codes that a command sets deliberately for the job control. That is the real message: **1 and 2 are success.** 1 means there is a relevant warning in the log -- for instance that a key expires soon. 2 means the cleanup after the job went wrong; the task itself is done and it can be ignored. The practical advice:
set `MAXCC=-2`, then 1 and 2 become 0. The message stays in the log -- whoever needs it evaluates the log, not the code. A subtlety that helps in support: relevant for code 1 is only a warning written by a component that was actually used. Warnings from components that dropped out of the processing because the format did not fit are in the log but do not produce a 1.



The report is what makes the difference in a batch chain: it says which member was processed and which was not. Logging is the level below that and belongs in troubleshooting, not in continuous operation -- it writes a lot. Recommendation for the chain: always write the report, switch logging on only when needed. The error trace is the third building block and the one you need in a support case. The example at the bottom is a real completion code 8, and it is worth reading it out in the room **from the bottom up** -- that is how it is built: `uiFlmDelArc` wanted to determine rights, for which `uiFlmArcRights` had to open the segment store, which went via `uiFlmSegOpn` and `uiFlmStoOpn` down to `segstore_objects_open`, and right at the bottom stands the actual reason: "blob not found (key=root)" -- the root object is missing, so the segment store does not exist. The point for the participants: **the topmost message names the symptom, the bottommost one the cause.** Anyone who copies only the first line into the ticket sends "Object not found" -- and that is the most useless information in the whole trace. With `MESSAGE(ERRTRACE)` the whole path is in the log, with source file, function and context. It is exactly this trace we would like to see in the ticket. It saves the first round of follow-up questions entirely.

COMBINING SEVERAL STEPS

- flcl fits into your automation and does not bring one of its own: JCL, BASH, BAT, PERL, scheduler
- what fits into **one** command belongs in one command: one `READ` with several `WRITE`, one `XCNV` with several `OUTPUT`
- **Wildcards, directory traversal and file arrays** are resolved by flcl itself – no loop in the script: `file[a b c]`
- **Pre- and post-processing** runs commands of your own before and after the conversion – per file or per job, and where the file is
- `flcl UTIL run.list` / `run.remove` / `run.files` replaces the cleanup and search steps of the chain

PRE- AND POST-PROCESSING IN A DIRECTORY TRAVERSAL

```
flcl xcnv "input(sav.fil(fio.blk(file='/in/*.txt' \
       prepro(command='touch [name]')))) \
  output(sav.fil(fio.blk(file='[path]/out/[base].gz' comp.gzip() \
       postpro(command='sha1sum [name]' on=success perfile))))"
```

When writing, post-processing runs by default only at the final close – with n inputs to n outputs you need `PERFILE`.

FLAMv5.21-42969

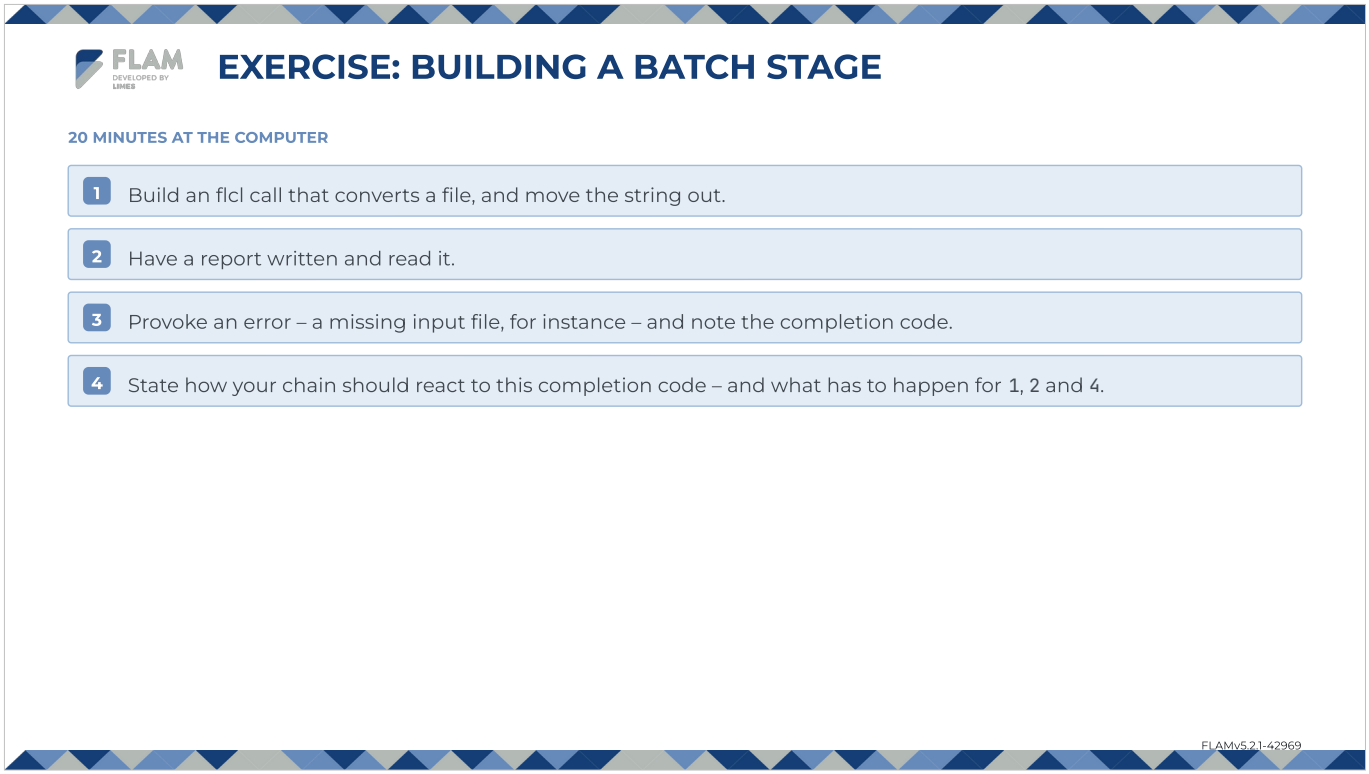
The point is that FLAM fits into existing automation and does not bring its own: JCL, shell, scheduler, whatever is there. There is deliberately no FLAM scheduler and no FLAM scripting language.
And one route that belongs to this access and is only mentioned in the API unit: **the subprograms.** With them a program of your own calls **every command as a function** -- that is, whole-file processing, including the directory walk. Three things therefore belong to the batch access: the **FLCL** for the command line, the **FLCC** for building the strings, and the **load modules with the command functions** for the integration into an application. Anyone who wants to use the utilities in a tool of their own takes the subprogram and not the record interface -- that is the most frequent confusion of all. The order of the points is the order in which you need them. First: whatever can be combined into one command belongs in one command -- a READ with several WRITEs, or an XCNV with several OUTPUTs. That is the big lever and the reason why a chain turns into a single step.
Second: wildcards, directory walk and file arrays. flcl resolves them itself, you need no loop in the script. Third, and almost nobody knows this: pre- and post-processing. Since 5.1.21 you can run your own commands before and after a conversion, per file or per job, on the read side and on the write side. The interesting part: the commands run where the file is -- if you write via SSH, they run on the remote machine. That does away with the small helper steps of the old chain. Name the trap in post-processing:
on writing it runs by default only in the final close, so in a directory walk only once, for the last file. That is right when everything goes into one archive. Anyone who processes n inputs into n outputs and wants to do something per file needs the PERFILE switch (since 5.1.24). Fourth, the cleanup work: `flcl UTIL` handles with `run.list`, `run.remove` and `run.files` exactly what otherwise takes an IDCAMS step on the mainframe.
Important, because I had it wrong here myself: there is **no** `flcl RUN` command. `RUN` is an overlay of `UTIL`, and `command=>'datei'` is a parameter file, not a command sequence.



The slide that is asked about most frequently in operations -- and the one that replaces a heap of hand-written loops. **The file specification is an array.** So you can give several patterns side by side, so-called file lists, and exclude things from the combined hit set again with `EXCLUDE`. `EXCLUDE` can likewise be given more than once. Go through the wildcards briefly: `` *and* `?` *as usual,* `+` *requires at least one further character. On MVS the dot is a qualifier separator, which is why a plain* `` only matches within one qualifier -- across qualifiers it takes `` **or `++`. A wildcard in the name itself is escaped with the circumflex `^`. GDGs can be addressed deliberately: `file='~.**(-1)'` reads the previous generation from all GDGs under your own SYSUID. You can also reach into archives: `member=` or a `?` in the URL selects members by pattern, a `#` or `:` by index, including ranges -- `member=#3,5,8-10,15`. The `DIR` object holds the switches for it: `RECURSIVE` for subdirectories, `ARCHIVE` to go into archives, `HIDDEN`, `LINK`, `TAPE`, `ONEFILESYSTEM`. And now the two warnings that really hurt on the mainframe -- they stand in the manual in this form too: A **wildcard in the first qualifier** triggers a search through the Catalog Search Interface (CSI). That compares *every* data set of the catalog against the pattern and can take a very long time.
And: a pattern can hit a file that **is on tape or migrated**.
Then the run waits for a console reply. Tapes are excluded by default, but that does not prevent the console request in every case -- be careful with wildcards at the start of the pattern.



All from support cases in batch use. On the first one: with FB80 data the ISPF editor can carry line numbers along in columns 73-80.
The CLP reads the parameter file freely, so those numbers stand in the middle of the parameter string and it reports a syntax error at a place that looks perfectly correct in the editor. Two ways out:
set `NUMBER=OFF`, or **put a semicolon at the end of the line** -- then the rest of the line is a comment and the number no longer gets in the way. The semicolon is the more robust way, because it still holds when somebody else touches the file later with `NUMBER=ON`. On the logging trap: the difference between a single file and a directory walk is large here. The default logging writes several lines per file -- with one file that is intended, with ten thousand it fills the file system and makes the run slower. In a directory walk `MESSAGE(MINIMAL)` or `MESSAGE(ERRONLY)` therefore belongs in the call.



Twenty minutes. The goal is a stage you could hang into a real chain -- so with a report and with a reaction to the completion code. On the fourth point, watch out that they do not test for "not equal to zero": 1 and 2 are success.
Solution:
1. The call is the same in bash, PowerShell and JCL -- only the packaging differs: backslash, backtick, parameter file.
2. If you provoke an error with a missing input file, `8` comes back: the execution ran, but with an error. What matters is what does *not* come back -- a `1` or a `2` would have been success.
3. So the chain has to distinguish three cases: `0` to `2` carry on, `4` decide on business grounds (some files were not processed), from `8` on abort.
Anyone who tests for "not equal to zero" aborts at `1` and `2` for no reason.
4. `MAXCC=-2` turns `1` and `2` into a `0`. The message stays in the log -- so evaluate the log, not the code.

WHAT TO TAKE AWAY

- shell and JCL differ in the environment, not in the language
- the goal is a JCL that does not change – everything variable sits next to it
- Completion Codes are graded: 1 and 2 are success, 4 is a warning – “not equal to zero” is not an evaluation
- the report explains the run, the logging explains the cause
- flcl fits into your automation and resolves wildcards itself

FLAMv5.21-42969

Go through the learning objectives. After that the same content follows via subsystem and file system -- transparently, without a call.

FLAM TRAINING, LEVEL 1

The application notices nothing

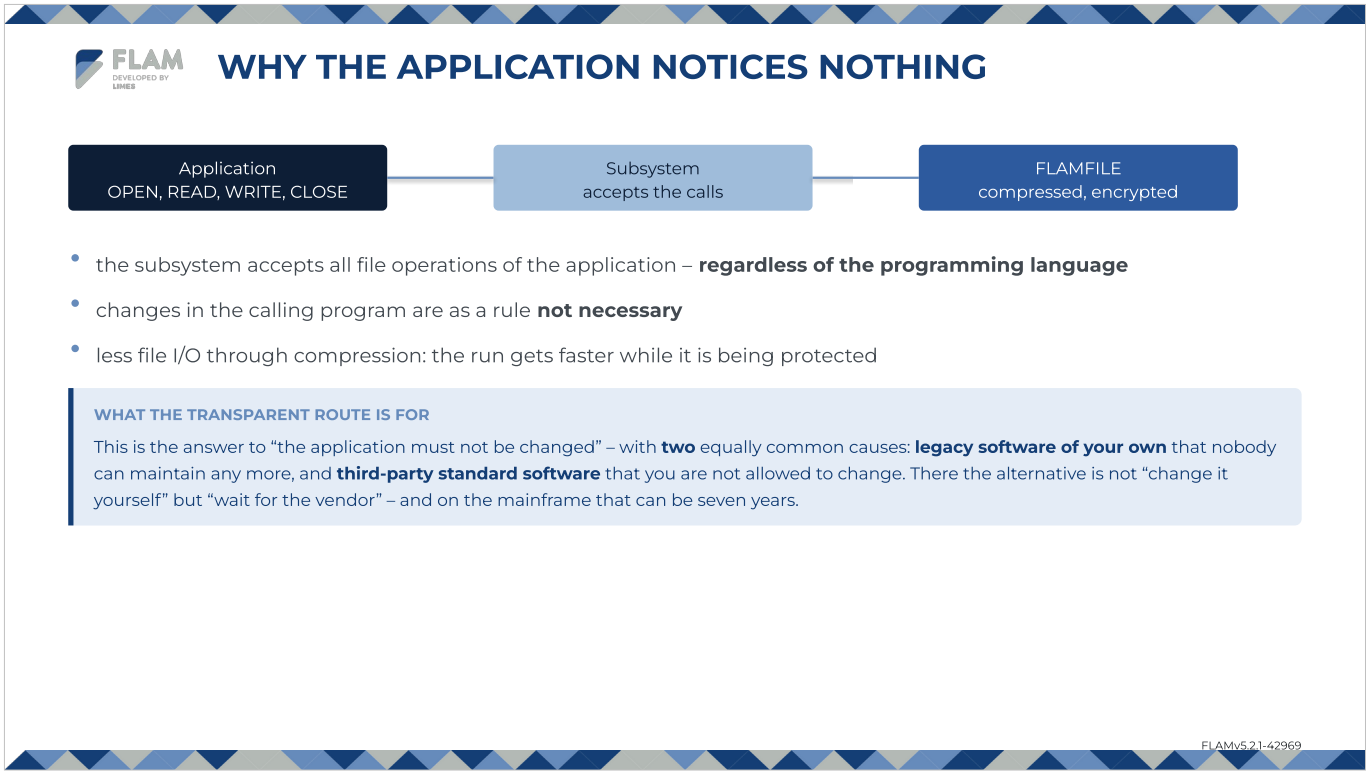
One DD statement on z/OS, one mount on Linux and Unix

Falk Reichbott · limes datentechnik gmbh

Second access route, and the one with the greatest moment of insight: the application is not touched and notices nothing.
This is the answer to "the application must not be changed" -- a sentence that comes up in every legacy project.



Objective five is the one that makes a deployment fail if you do not know it beforehand.



Literally from the manual: all calls of the application program for file operations -- open, close, read and write records -- are taken over and processed by the subsystem. Changes in the calling program are as a rule not necessary. That holds regardless of the language: COBOL, PL/I, C, Java, assembler. The reason is that the redirection happens below the language. A second effect that often tips the decision: compression means less file I/O, so the run gets faster while it is additionally protected. And now the most important part of this slide, because it doubles the field of application. "The application must not be changed" has two quite different causes, and both are equally frequent. The first is the well-known one: your own legacy software. An assembler program from the eighties that runs, that nobody wants to touch any more and for which there is nobody left who could. That is where the subsystem is the only place you can get a grip. The second is the one that is easily overlooked and that is at least as important today: **third-party standard software**. There you may not change anything, not because it would be too risky, but because you do not have the source code. The alternative is not "change it yourself" but "wait for the vendor". Two examples that should be told, because they are understood immediately. One customer wanted all SAP files to be encrypted. Solved with flucFS -- without touching SAP, without filing a requirement with SAP, without waiting for it. And on the mainframe: at IBM seven years can quite well pass until a customer request is fulfilled. In the meantime customers as a rule map the desired functionality transparently to the IBM software via the subsystems. That is not a stopgap, that is the practicable way. Anyone in the room who looks after standard software stops at this point thinking of this as a mainframe legacy topic.



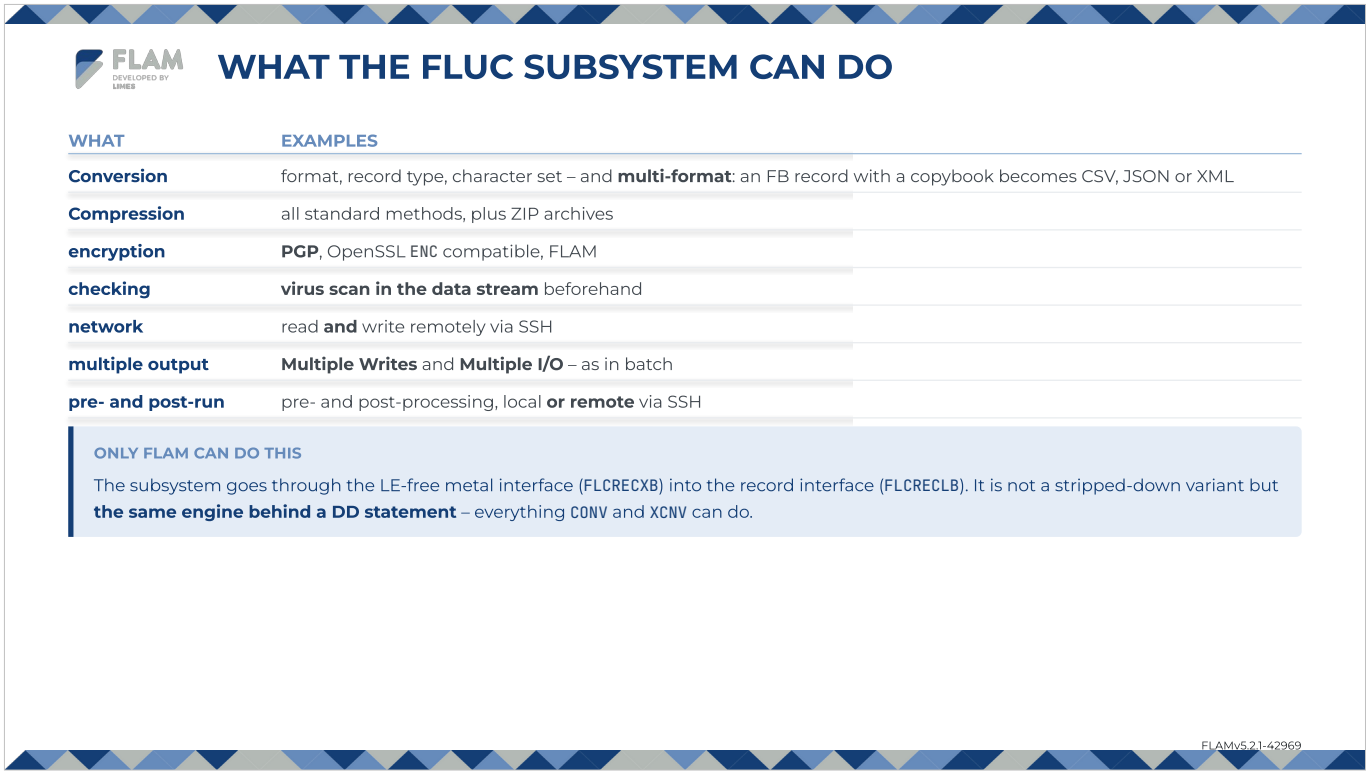
Anyone who speaks of "the subsystem" means two different things depending on their background -- and then expects from the one what only the other can do. That is why the names stand here separately.

Go through it from top to bottom, and stress the timeline while doing so -- two of them exist today, two are coming. FLAMSUB is the old one, and it can do exactly one thing: read and write FLAMFILEs.

That is not a criticism, that was the task. FLUCSUB is the new one and can do everything `CONV` and `XCNV` can -- the next slide spells that out, and the list surprises most people. The wording is chosen deliberately: a customer hears "everything the record interface can do" as the **FLAM4** record interface and thinks of its limits. What is meant is the full scope of the conversion.

FLM5SUB for the FLAM5 archives is planned and will come in one of the next releases. That matters for managing expectations: anyone who wants to address a FLAM5 archive transparently through a subsystem today cannot do so yet. The same picture on the Unix side: flucFS exists, flm5FS is the planned counterpart. If somebody asks about the names -- and the question does come, because at first glance they look illogical -- the answer is instructive, because it explains the architecture. Why not FLAMSUB for the new subsystem? Because FLAMSUB already exists, the old FLAM4 subsystem, and it will continue to exist. We are not replacing it. A customer with FLAM4 data sets keeps his FLAMSUB and gets FLM5SUB alongside it if he wants to address FLAM5 archives transparently. That is why the new one needs a name of its own. And on the Unix side: there flamFS would be free, because there never was a FUSE file system for the old FLAM4FILE. We call it flm5FS all the same -- so that the same name stands for the same thing on both platforms. The price is a name that looks unfamiliar seen in isolation; the gain is that you do not have to keep two naming schemes in your head.

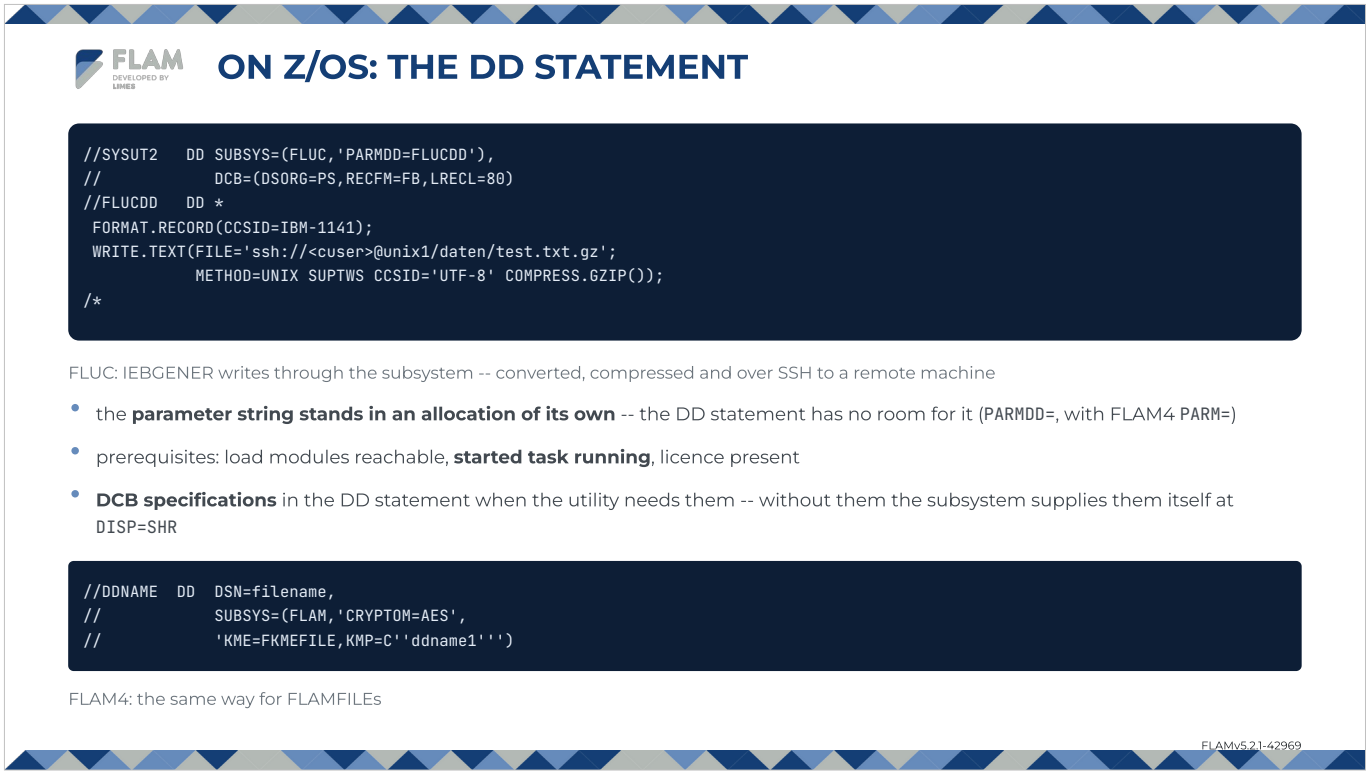
Both lines are an outlook. As long as they are not delivered, they belong to be presented as an outlook and not as a commitment. If somebody from a mid-sized company is sitting in the room, the strategic addition pays off: flm5FS is not only the Unix counterpart to FLM5SUB, it is the entry phase into the cloud. These customers often still sit on client-server. Via a file system they can move their data out into cloud storages safely, **without that having much impact in the company** -- no application rebuild, one mount. That is why it comes before the standard APIs for cloud applications in our cloud planning: the order does not follow the technical difficulty but the question of which customer can come on board next. The sentence that holds the slide together: FLAM archives, FLUC converts -- and the names follow this separation, not the chronology.



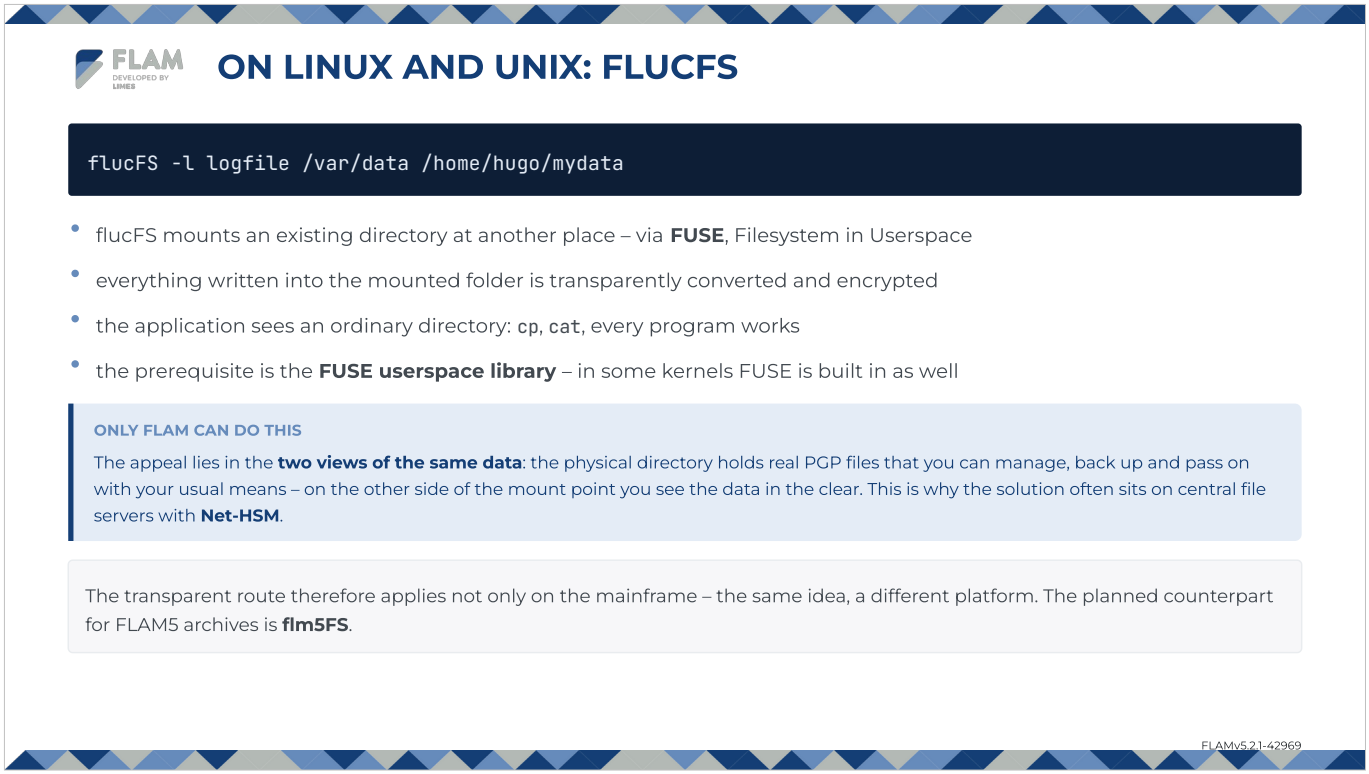
This slide is the actual content of the unit, and it was not in there before -- the deck had reduced FLUCSUB to "convert format and character set". That is considerably too little. The reason the list is so long is an architectural one and is worth the sentence: the subsystem goes through the LE-free metal interface (FLCRECXB) into the record interface (FLCRECLB). That makes everything available in the subsystem that `CONV` and `XCNV` deliver as well -- that is, reading from memory with format-dot-something, and after that any number of writes with the full scope of the conversion. It is not a stripped-down variant but the same engine behind a DD statement. Go through the list once, but do not read it out. The three that trigger reactions in the room are: virus scanning in the data stream, PGP, and reading and writing remotely via SSH. Make the multi-format conversion concrete: out of an FB record with a copybook belonging to it comes a CSV file -- or JSON, or XML.
Without touching the application. FLAMSUB by contrast can only read and write FLAMFILEs. The side remark matters here, because it often causes confusion: the FLUC **understands** FLAM4 as an original datum -- just like a PGP file or a ZIP archive. So FLUCSUB can do FLAMFILEs too, for it that is just one input among many.



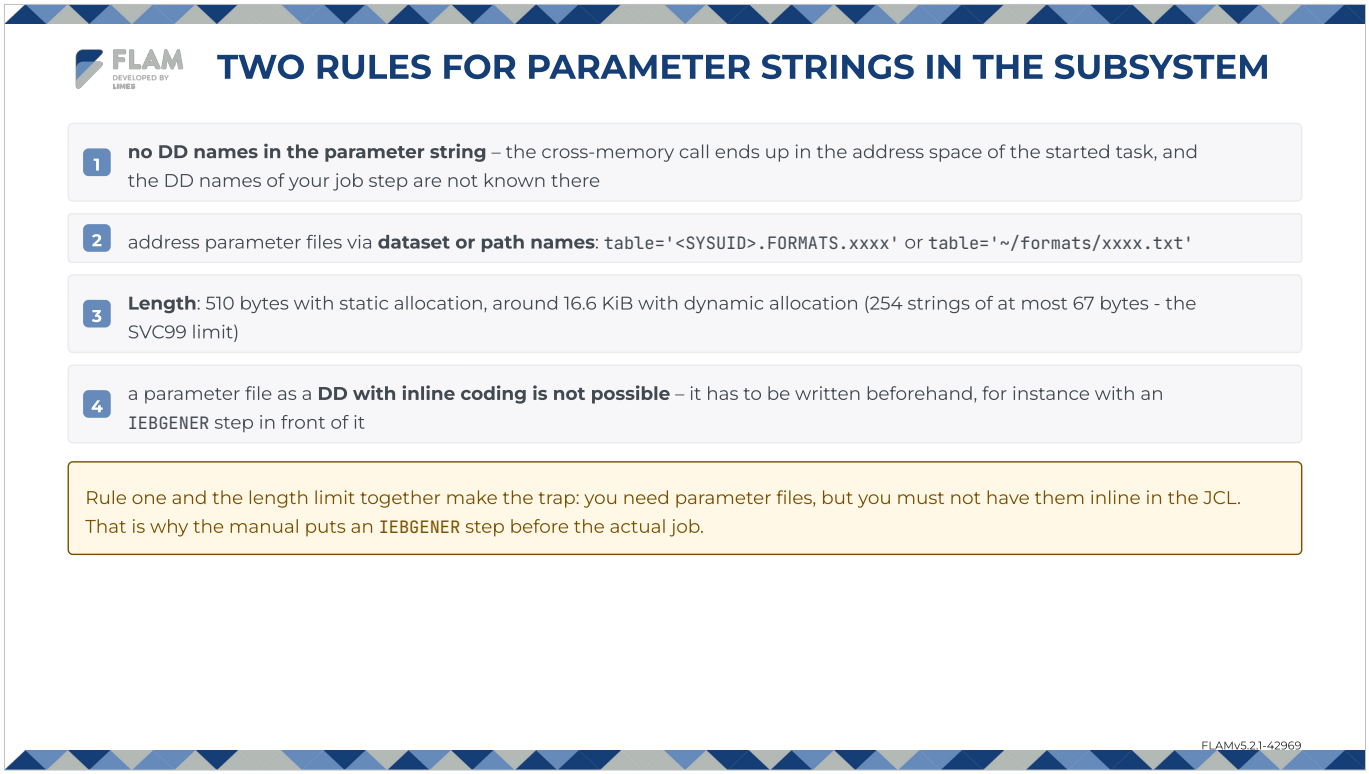
The example that explains the benefit of FLUCSUB in one picture, and it comes from practice. The starting point: for security reasons an old dataset is no longer to lie there in the clear. It is replaced by a classic FLAMFILE -- the application notices nothing of it, because the subsystem sits in front of it. And then comes the actual trick: in the same call you branch the data off.
Out of the same read operation a CSV file is additionally produced, in UTF-8 with 0x0A as the separator, packed as GZIP, deposited via SSH directly at the data warehouse -- and as post-processing the import is started there. Point out the timeline, that is the point:
all of that happens in one read operation, not in a chain. The application writes once. The three lines at the bottom are the pattern: `format.record()` says how the application delivers the data, after that two writes -- one secures, one delivers.



The slide on which the subsystem becomes concrete -- and it deliberately shows **FLUC** first, because the units before it speak of FLUC. On top a real write step out of our delivery (`JOBLIB/FLUCSUB1.jcl`): IEBGENER writes to `SYSUT2`, and because `SUBSYS=(FLUC,'PARMDD=FLUCDD')` stands there, the record stream goes through FLUC. The parameter string lies in an allocation of its own -- that is the rule in the subsystem, because the DD statement has no room for a long string. What the step does is the sentence that should stick: **an unchanged utility writes a compressed file to a remote Unix machine.** Character set conversion from 1141 to UTF-8, line endings the Unix way, trailing blanks gone, gzip, and the transfer over SSH -- all in the DD statement, no program changed. Name the URL rule along with it, because it is visible here: **one slash behind the host means relative to the home directory**, two slashes mean absolute. Below the same pattern for **FLAM4** -- the syntax verbatim from `doc/man_flam4_appendix_flamsub.txt`, section "Encryption using FKMEFILE". Do not improvise: the doubled quotes are the part everybody gets wrong. Name the three prerequisites clearly: load modules reachable, started task running, licence present. If one of them is missing, the error is not self-explanatory. And the property that saves trouble in operations: **invalid parameters FLAM rejects as a JCL error**, not only at run time -- so the job does not come back after two hours of CPU time. The semicolon at the end of the line is the comment character of the command language. With `NUMBER=ON` in the editor the line numbers behind it become a comment -- which is why it stands in all our examples.



The counterpart outside z/OS, and it is pleasingly simple: a directory is mounted at another place, and everything that is written there is transparently converted and encrypted. The example literally from the manual. Demonstrate it if there is a Linux to hand -- the effect "I write with cat and it is encrypted" convinces immediately. Name the dependency: the FUSE library in user space has to be present. The real appeal is the two views of the same data. In the physical directory there are **real PGP files** -- the customer can manage, back up, write to tape and pass those on with his usual means. On the other side of the mount point the application sees the data in the clear. No special format that nobody but us can read. That is why the solution often sits on central file servers with **Net-HSM**: the key is in the HSM, the files lie in the file system as PGP, and whoever does not have the mount sees nothing. And the timeline: for the FLAM5 archives **flm5FS** is the planned counterpart, and it does not exist yet.



This slide was missing, and it is the reason for a good part of the support cases around the subsystem. Both points stand in the manual under "Restrictions", but nobody reads appendices. Rule one, no DD names in the parameter string. The reason follows from the architecture: the subsystem uses the LE-free interface of FLAMSTC, and a cross-memory call passes the conversion on to a server whose address space is controlled by the started task. In that address space the DD names of the job step are simply not known. Only such DD names work as were allocated statically via FLAMSTC -- which is why it is pointless to work with DD names in the subsystem. The consequence for practice: address parameter files by data set name or path name, not by DD. Rule two, the length. With static allocation it is 510 bytes. With dynamic allocation using FLAM5 components it is around 16.6 KiB, and the number comes about like this: 254 parameter strings of at most 67 bytes each, which is the limit of SVC99. It follows that parameter files are even more important in the subsystem than in batch. And here is the trap that ties the two together: a parameter file as a DD with inline coding does not work, because of rule one. You have to write the file beforehand -- typically with an IEBGENER step in front of it, which is how the example stands in the manual too. Anyone who knows these three sentences saves themselves the two most frequent tickets.

WHAT THIS ROUTE CANNOT DO

- the FLUC subsystem works on **streams**, that is on purely sequential I/O – **whole-file processing**, no change to individual parts
- the route needs **a running service**: a started task on z/OS, a FUSE mount on Linux – operations, then, not just a library
- it is **not** a way to search in the archive or to find individual records via a key – that is the record interface
- the application cannot control how encryption is done – the allocation decides that

For FLAM4FILEs only: there a modifying access removes the integrity protection over the whole file. For **FLAM5 archives this no longer applies** – there integrity and completeness are preserved with full access (see the unit *The FLAM5 archive*). A subsystem for it (**FLM5SUB**) is planned.

FLAMv5.21-42969

This slide is the most important one of the unit, because it saves projects from a wrong assumption. Two things have to be kept cleanly apart here, because they often get thrown into one pot.
First: the statement about integrity protection applies only to the **old FLAM4FILEs**. There a modifying access breaks the MAC over the whole thing, because it cannot be recomputed. For the new **FLAM5 archives this no longer applies** -- there integrity and completeness are preserved under full access. So anyone who takes away from this slide that "FLAM cannot do an update with integrity protection" has learnt the opposite. The cross-reference points to the unit on the FLAM5 archive.
Second, the actual limit of the FLUC subsystem, and it is a different one from the MAC: the subsystem works on **streams**, that is purely sequential I/O. That means whole-file processing -- you cannot change parts of it. That is not a crypto topic, it is the nature of the subsystem interface.
Third, organisational and often underestimated: this route needs a running service, so it needs operations -- not just a library.
And the outlook, so that nobody takes the limit to be final: FLM5SUB for the FLAM5 archives is planned.

PITFALLS

! **No subsystem without a started task.**

The `SUBSYS=` fails when the service is not running – and the error does not sound like it.

! **A FLAMFILE read without `SUBSYS=` delivers the compressed bytes.**

That is not corruption, that is a missing allocation.

! **Missing DCB information hits utilities.**

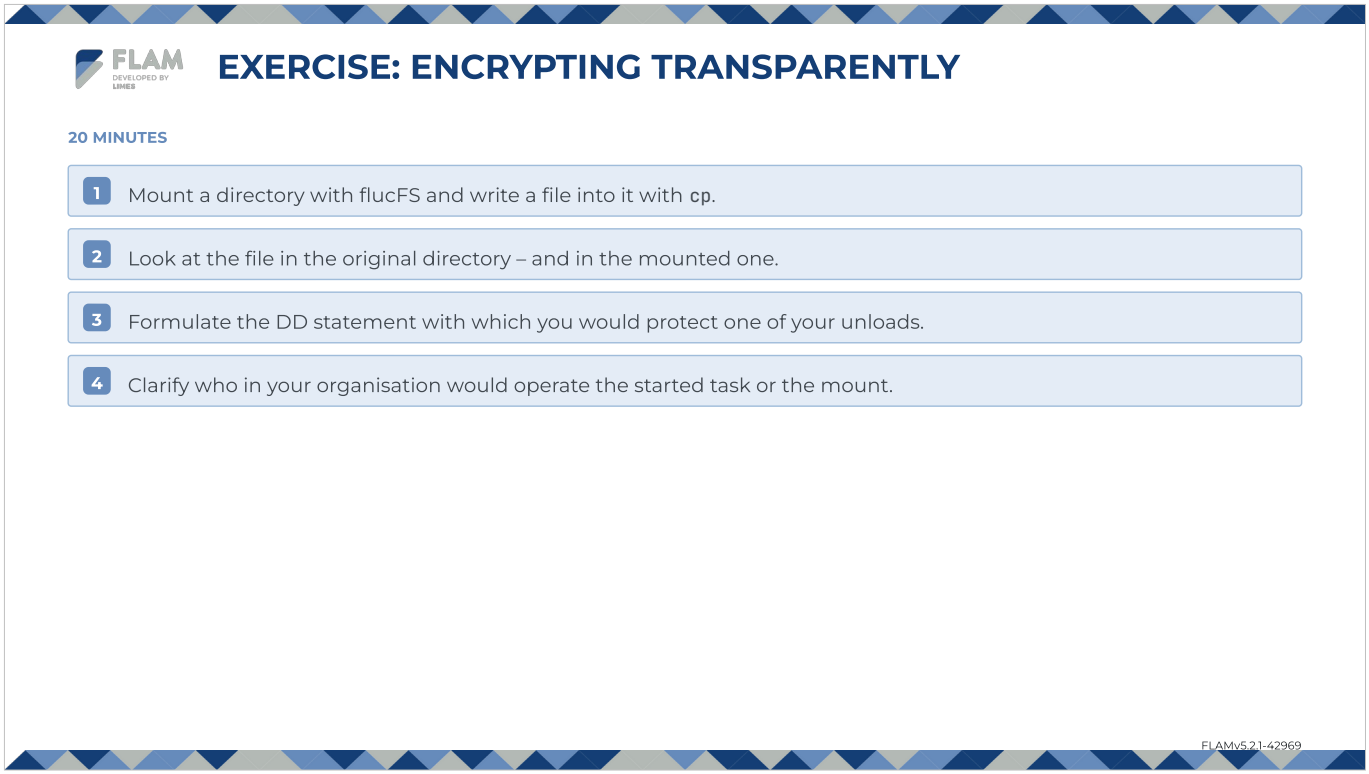
IEBGENER, SORT and assembler routines of your own occasionally need it in the DD statement; without it the subsystem supplies it itself with `DISP=SHR`.

! **The job user needs access to the key.**

With ICSF that means label and service. Otherwise the job fails at the open, not at the write.

FLAMv5.21-42969

All four come from support cases. The first one costs the most tickets, the second one the most time.



Twenty minutes. On Linux with flucFS, on z/OS using the example of a DD statement on paper if no system is available.
Exercise 4 is the important one: it clarifies who in your own organisation would operate the service.
Solution:
1. The `SUBSYS=(FLAM,...)` in the DD statement is the entire change. The application is not touched and not re-linked.
2. Without a running started task the `SUBSYS=` fails -- and the message does not sound like "service missing". That is the most frequent cause of tickets on this topic.
3. If you read the file without `SUBSYS=`, you get the compressed bytes. That is not corruption, it is a missing allocation -- the learning point of the exercise.
4. The application cannot control how encryption is done; the allocation decides that. And with a **FLAM4FILE**, random modification voids the integrity protection over the whole file -- with a FLAM5 archive it does not, there integrity and completeness are preserved. So whoever needs both takes the record interface and a FLAM5 archive.
And do not forget the two rules: no DD names in the parameter string, and the length limit of 510 bytes with static allocation.

WHAT TO TAKE AWAY

- the application is not touched and notices no difference
- **FLAMSUB** (old) can only handle FLAMFILEs - **FLUCSUB** (new) can do everything `CONV` and `XCNV` can
- on Unix **flucFS** via FUSE, with real PGP files in the physical directory
- the subsystem works on **streams**: whole-file processing, no change to individual parts
- the route needs a running service – operations, not just a library

FLM5SUB and **flm5FS** are planned for the **FLAM5 archives**. Today the transparent route applies to FLAMFILEs and to any conversion.

Go through the learning objectives. Name the four names once more; that is the point one can take away from this unit wrongly.
Then the third unit: the same thing inside the program, with access to individual records.

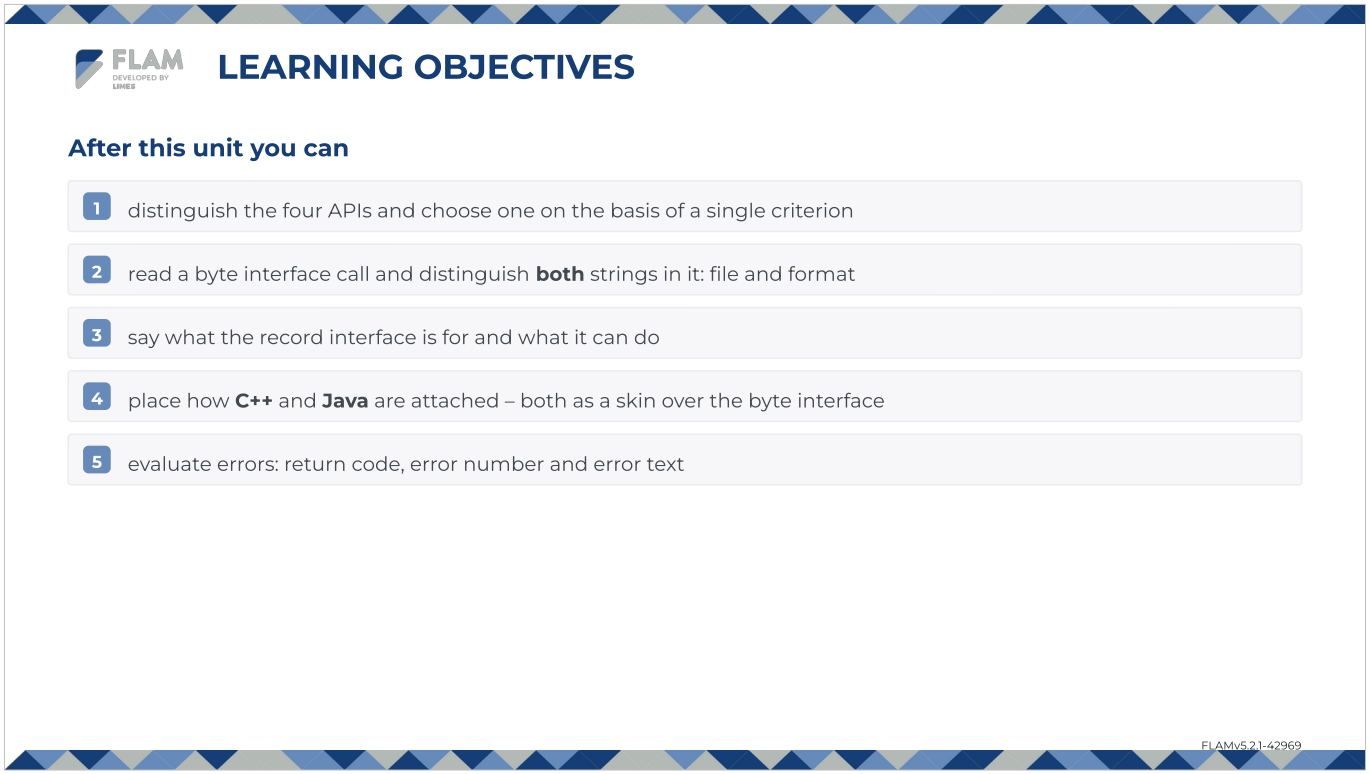
FLAM TRAINING, LEVEL 1

Record by record access from inside the program

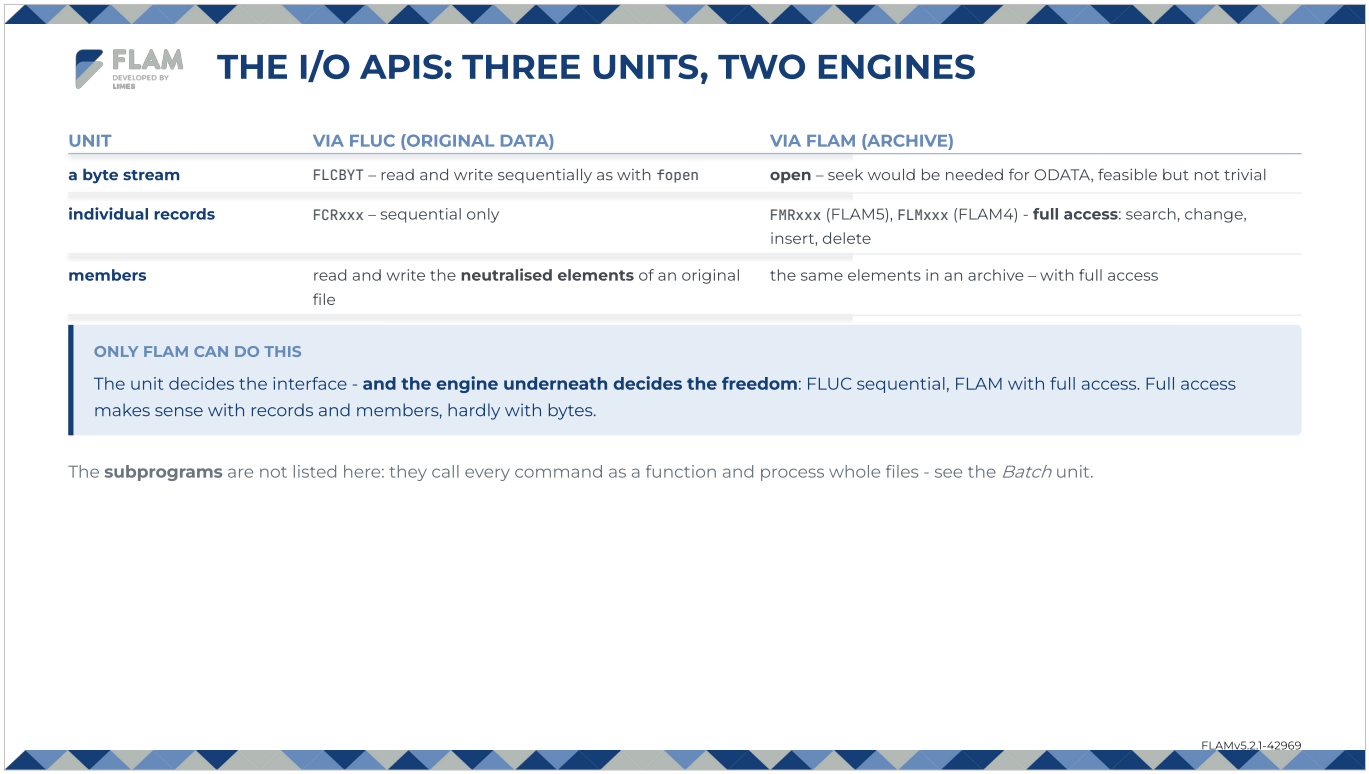
Byte, record, element, subprogram - plus C++ and Java

Falk Reichbott · limes datentechnik gmbh

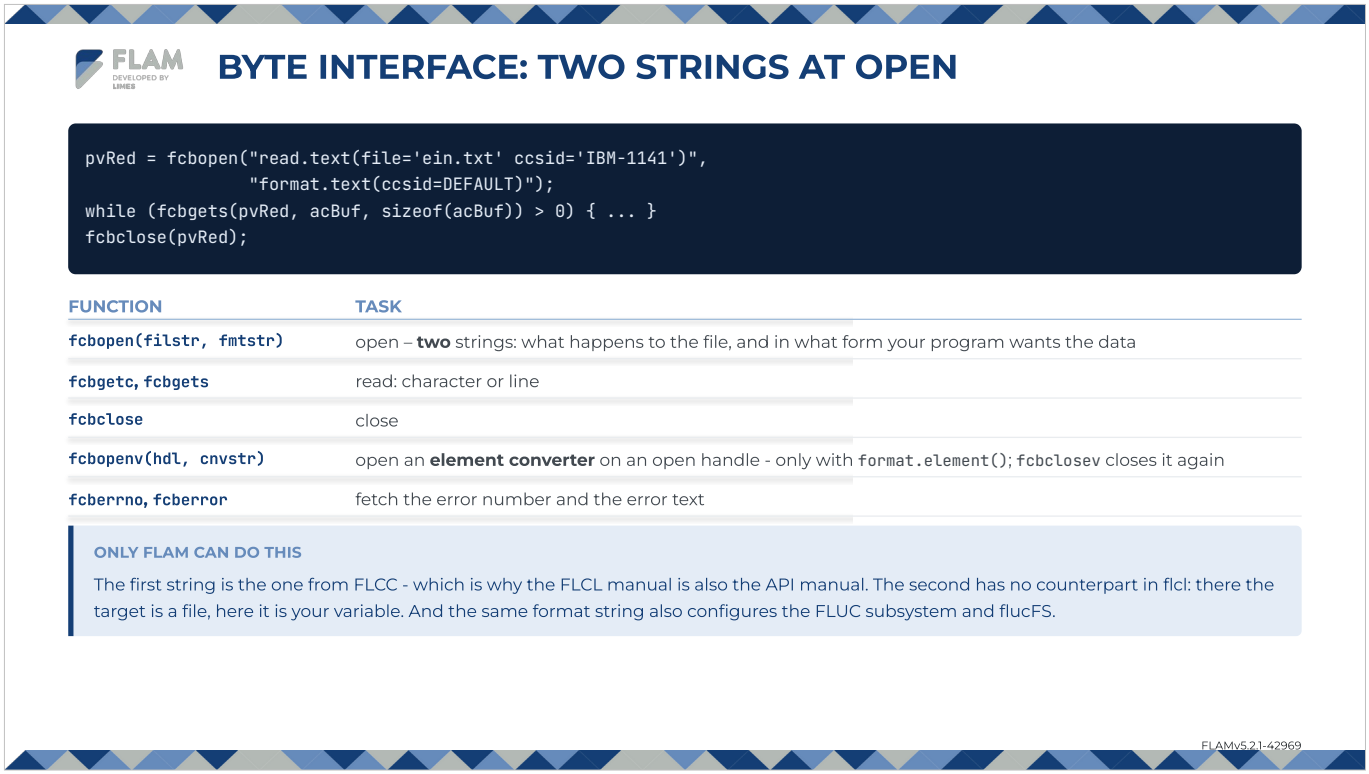
Third access route, and the only one where the application source code is touched. The return is direct access to individual records from inside the application. Important for the boundary, because the short cut is tempting: the API is **not** the only way to full access. `flcl archive` can search and change in batch, and `FLAMSUB` gives full access transparently as well, though with FLAM4 only with KSDS mapping.
What distinguishes the API is access **inside the program** -- the application decides per record what happens. The overview on that is in the unit *The three access routes*. The arc: four APIs, one selection criterion, then each one individually -- and at the end the error handling, because that is what makes the difference in practice.



Objective one is the decisive one: the choice of the right API. Whoever gets it wrong builds for weeks in the wrong place.



The order matters here, otherwise the slide confuses more than it explains. Two axes, and both have to be in mind. **First axis: the unit** your program thinks in -- a byte stream, a record or an element. All three work on an **opened file**:
open, read or write, close. That is the character of an I/O interface. **Second axis, and it is missing from most explanations: FLUC or FLAM.** With the record and the element interface that makes the decisive difference: Over **FLUC** only **sequential** reading or writing is possible. FLUC processes original data, and a stream has no position to jump to. Over **FLAM** there is **full access** -- search, change, insert, delete. Because there an archive with a directory and an index lies underneath. On the byte interface the honest answer, because it is asked in the room: **at present it exists on a FLUC basis only**, that is for streams of original data, and therefore only sequentially. Whether we will offer a byte interface with full access, that is with seek, for FLAM as well is **not yet decided**. For ODATA it might become necessary. It is feasible, not trivial. And the thought behind it, worth passing on: **full access makes sense with records or single elements, with bytes hardly at all.** A byte offset in compressed and encrypted data has little to do with the rows of a table and with indexing. One sentence belongs to the element interface to explain its purpose, because otherwise it looks like a third variant without a reason: **it exists for FLUC and for FLAM, and it works on the neutralised elements** -- that is on the data after FLAM has released it from its format and before it is cast into a target format. The benefit becomes clear in a case everybody knows who relocates data:
read an original file with FLUC and write it into a FLAM archive. Without the element interface, a copybook record would first be built from the neutral elements, and from that record neutral elements again -- a conversion into memory and back out of it, for nothing. With it the **elements are passed through**: from reading the original file straight into the archive. That is not a convenience, that is one working step per record saved. With mass data that is the difference you see in the CPU time. What does **not** stand on this slide and where it belongs: the **subprograms**. They are not an I/O interface but the possibility of calling **every command as a function** -- that is whole-file processing, including the directory walk. That is a quite different character and is therefore in the unit *Batch*. Two slides further on they appear once more, so that they are listed as a programming interface.



The entry point for everything that is not mainframe. Whoever knows `fopen` and `fgets` knows the pattern -- the difference sits in the open.
And here is the point that carries the whole unit and that one skips over on a first reading of the handbook: **`fcbopen` takes two strings**, not one.
The first one is the file or conversion string -- exactly the one from the FLCL handbook and from the FLCC, so `read....` or `write....` or `input(...)`/`output(...)` of XCNV. It describes what happens with the file.
The second one is the **format string**. It describes in which form your program wants to have the data: `format.text()`, `format.record()`, `format.binary()`, `format.table()`, `format.element()`. That is the interface to your memory, and it does not exist with flcl -- there the target is a file, here it is your variable.
So the two questions kept apart: what happens with the file, and how do I want to see the data.
And here the confusion that almost inevitably happens when reading the function list, because the names are so similar: `fcbopenv` is **not** a variant of `fcbopen`. It opens a **member converter** on an already open handle, and it only makes sense with `format.element()`. You can have several of them open at the same time, and the read and write functions with `v` at the end take such a converter handle along in addition. With it you convert member by member -- for instance a number from an XML into a packed BCD representation.
Now the classification that explains why there are exactly two strings and not one -- it applies to **all** interfaces and is worth doing once in detail:
The **read and write strings for file I/O correspond one to one to the strings of the utilities.** That is why the FLCL handbook is the API handbook as well, and that is why the FLCC builds them too.
It is different when reading **out of** memory and writing **into** memory:
there only the sensible part of the FLUC is offered. That is exactly what the keyword `format` is there for. And that is independent of whether at the other end there is a FLAM archive or original data.
The example above is the **FLUC byte interface**. It reads from a file -- that is the original-data processing of the FLUC -- and writes through the format string into memory. That is FLUC as well: a `WRITE.TEXT()` with FIODAT, so to memory.
And this is how it fits together in general, this is the sentence for the whiteboard: **FLUC reads the original and produces a neutral matrix in memory from it. Either FLUC writes from this matrix again -- into a file or into memory -- or FLAM writes it away as compressed and encrypted segments.** Backwards just the same: FLAM reads the segments and makes the neutral matrix, and the original-data writing of the FLUC turns this neutral member list into the format that is needed in memory -- copybook, C structure, JSON.
Whoever has understood this once understands all interfaces at the same time:
it is always the same middle. What differs is only who fills the matrix and who empties it.
The closing sentence: the format strings also configure the FLUC subsystem on z/OS and flucFS on Unix. That is the bridge to the previous unit.

THE FORMAT STRING: YOUR PROGRAM'S VIEW

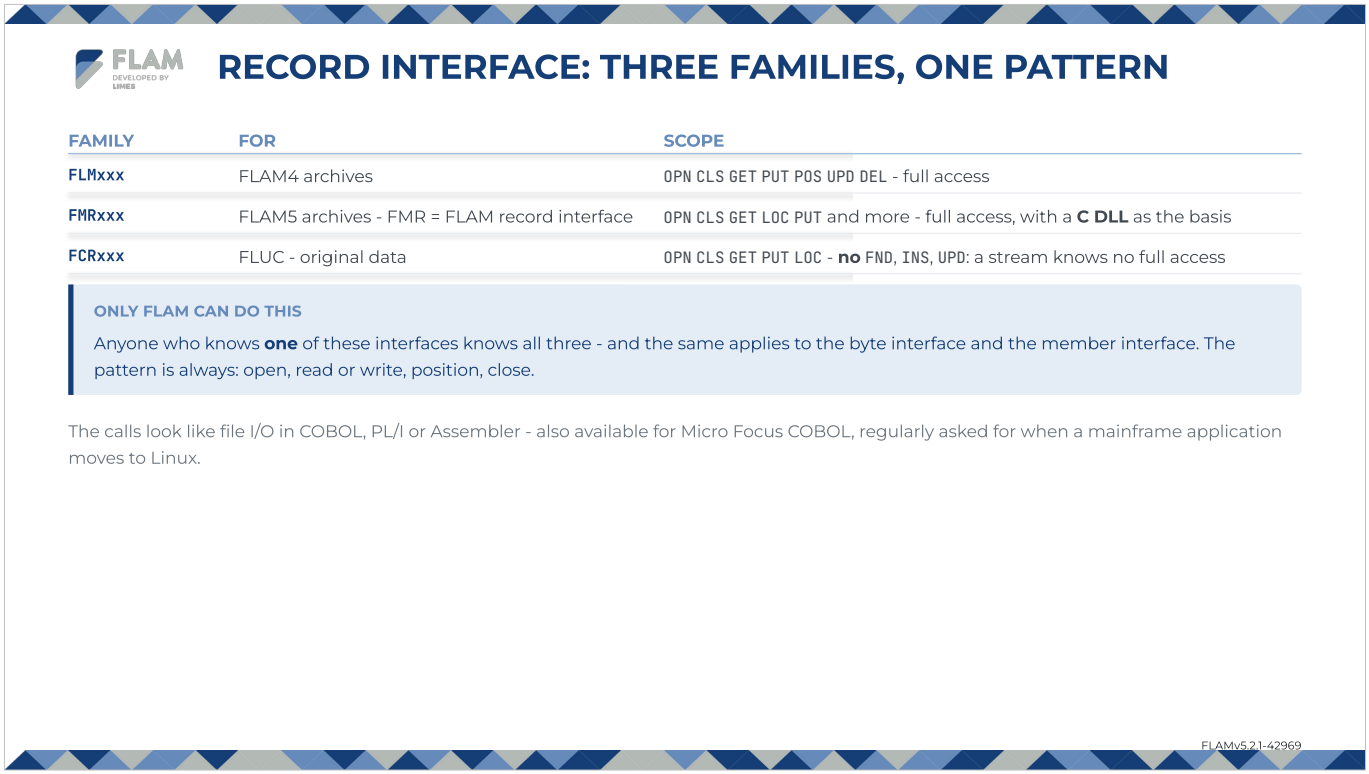
FORMAT STRING	WHAT YOUR PROGRAM RECEIVES OR SUPPLIES
`format.binary()`	bytes, unchanged
`format.char()`	characters, with character set conversion
`format.text()`	lines without separators
`format.record()`	records - the case for COBOL and PL/I
`format.xml()`	XML events
`format.table()`	rows and columns according to the row spec
`format.element()`	individual elements in the **neutral FLAM5 representation** - the basis for element converters

CONVERT PER ELEMENT - CHARACTER STRING FROM XML TO PACKED DECIMAL

```
void* f = fcbopen("read.xml(file='/daten/ein.xml')",
                  "format.element()");
void* c = fcbopenv(f, "conv.float(from(format.str(chrset=UTF-8))"
                  " to(format.bcd(SIGNED TYPE=ZONED"
                  " INTDIGITS=8 FRACDIGITS=2)))");
```

FLAMv5.21-42969

The reference slide for developers, and the slide that decides the choice of interface within the byte interface.
Go through the seven formats once, but not all of them equally deeply. The top three are the everyday business, `TABLE` and `ELEMENT` are the interesting ones.
`format.element()` is the point at which the byte interface goes beyond an ordinary file API: the application gets individual members in the neutral FLAM5 representation, and with a member converter you convert member by member into the representation the application needs.
The example at the bottom is real and taken from Appendix A: a number that sits in an XML as a character string is converted on reading into a packed decimal number with eight integer and two fractional digits. That is exactly what everybody otherwise writes themselves -- and makes mistakes doing it.
The state string at the end: on reading the open returns additional metadata, on writing you can pass it in. With it you transfer the file state from one side to the other without describing it yourself.

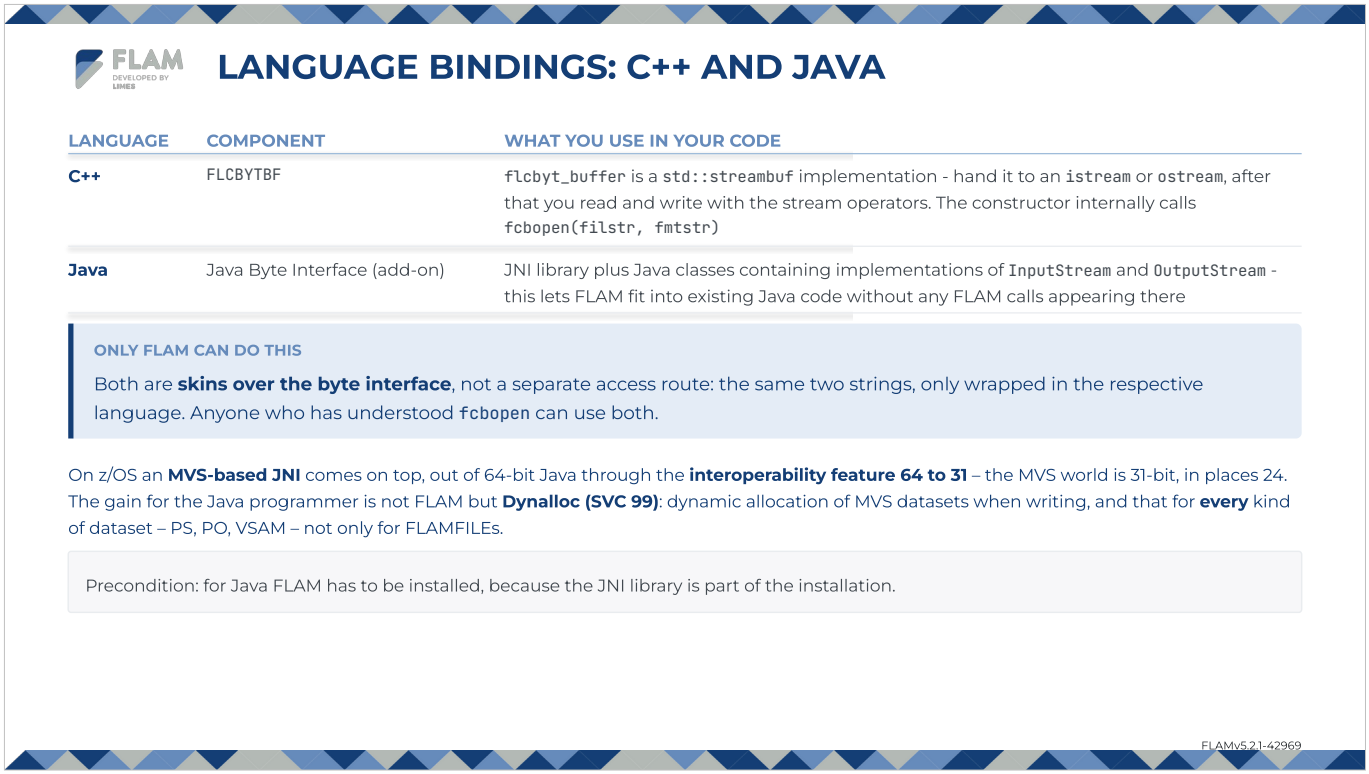


The core of FLAM on the mainframe. Few entry points, and that is the whole interface -- which surprises people, because you expect something big from an access method.
And now the order that counts on this slide: the record interface exists **three times**, for three targets, and the names say which one is meant:
`**FLMxxx**` is the old record interface for the **FLAM4 archives**. It has the full scope: `FLMOPN`, `FLMCLS`, `FLMGET`, `FLMPUT`, `FLMPOS`, `FLMUPD`, `FLMDEL`.
`**FMRxxx**` is the new record interface for the **FLAM5 archives** -- FMR stands for FLAM record interface. `FMROPN`, `FMRCLS`, `FMRGET`, `FMRLOC`, `FMRPUT` and more. For FLAM5 the same applies in principle as for FLAM4, except that here in addition a **DLL with a C interface** provides the basis -- that also makes it usable outside the classic host languages.
`**FCRxxx**` is the record interface of the **FLUC**, so for original data. And here is the difference that fits the previous slide: it has `FCROPN`, `FCRCLS`, `FCRGET`, `FCRPUT` and `FCRLOC` -- but **no** `FND`, `INS` or `UPD`. Because FLUC processes a stream and a stream knows no full access.
The sentence that holds the slide together and takes the fear away from the participants: **whoever knows one of these interfaces knows all three.** The pattern is the same -- open, read or write, position, close.
The same holds for the byte and the member interface. You learn the pattern once.
The calls deliberately look like file I/O in COBOL, PL/I or assembler.
The change in the program stays small -- often it is a search and replace of the I/O verbs.
Do not forget the security argument: only the segment that is needed lies in the clear in memory, no temporary clear-text file arises.

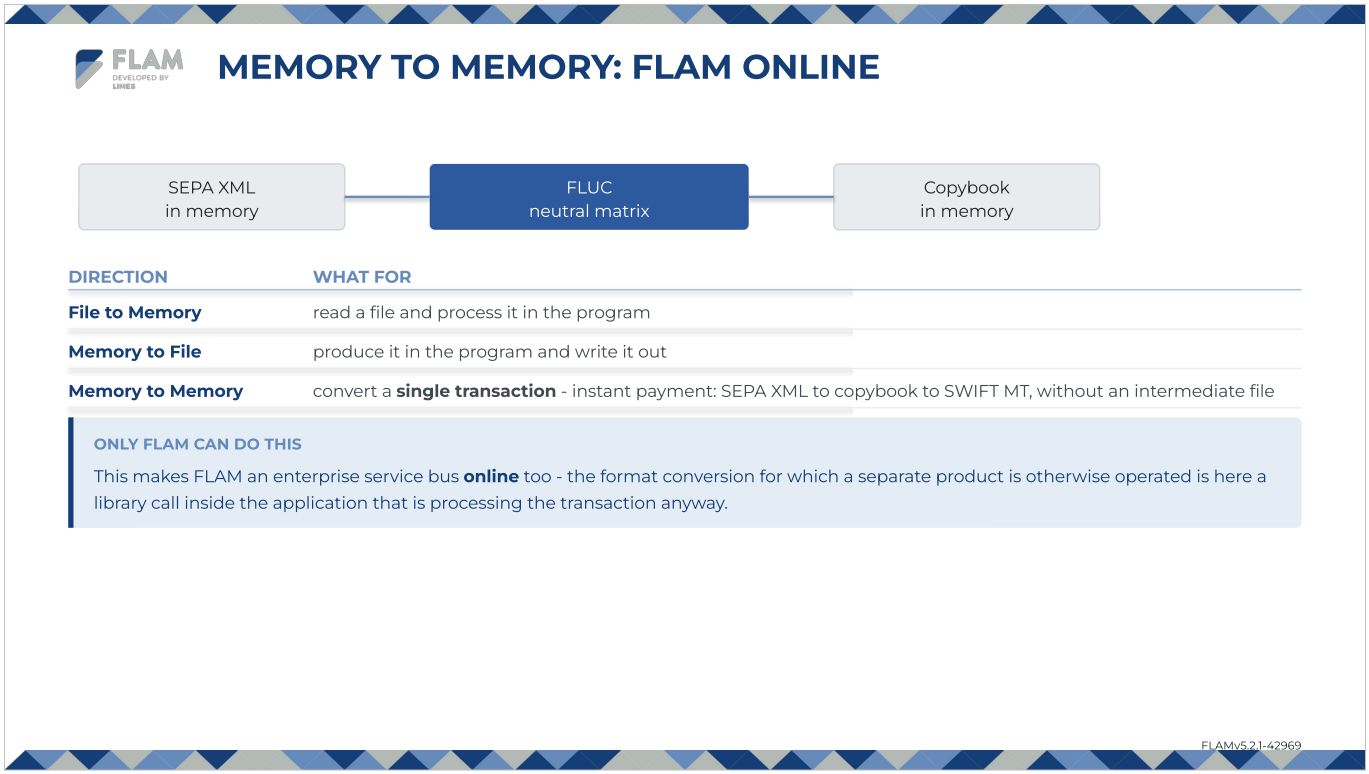
SUBPROGRAM: WHOLE FILES

- platform-neutral load module: **FLUC-UP** for the conversion, **FLAM-UP** for the archive
- statically bindable or as a shared library - callable from any language
- it is the same module that the utility uses as well: **whatever flcl can do, your program can do too**
- optional **exits** between physical I/O and data hand-over - the lever for your own protocols and your own I/O
- a release change does not force the key management to be bound again
- it is **not an I/O interface**: it calls a command as a function, with a directory walk - see the *Batch* unit

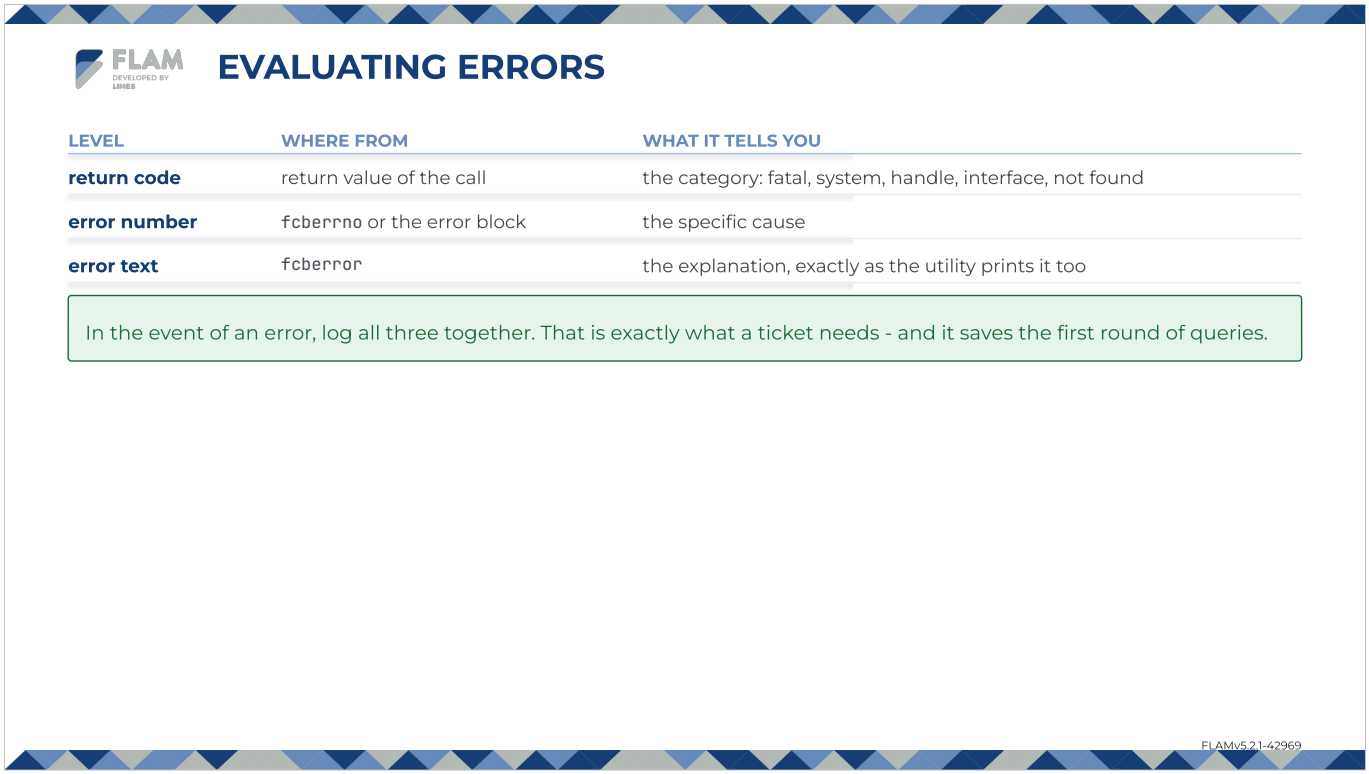
The answer to "we have a tool of our own and want to use FLAM inside it".
Important: the subprogram is the same module that the utility uses as well. Whatever flcl can do, your program can do with it too -- there is no second, weaker interface.
The exits are the lever for your own logging and your own I/O; point that out when in-house standards for audit trails apply in the room.
For classification, so that the slide does not stand apart from the others: the subprogram is **not an I/O interface**. It does not work on an opened file, it calls **a command as a function** -- with everything that goes with it, including the directory walk. So it is whole-file processing, and that is why it actually belongs to the unit *Batch*, where FLCL, FLCC and the load modules with command functions are covered together.
It is here nevertheless, and for one reason: **it is a programming interface**, and whoever hears this unit should know that it exists.
Whoever wants to integrate the utilities into an application needs exactly that.



This slide was about Java only before. The C++ interface was missing entirely -- it exists as a component of its own, `FLCBYTBF`, with its own Doxygen documentation, and it is the more convenient way for everybody writing C++. Keep the two names apart briefly, otherwise somebody asks: `FLCBYTBF` is the header `CPPINC/FLCBYTBF.hpp` and thus the component name in our 8-character convention; `flcbyt_buffer` is the class inside it. One thing, two levels. C++: `flcbyt_buffer` is a `std::streambuf` implementation. You hand it to an `istream` or an `ostream`, and after that the code reads and writes quite normally with the stream operators. The constructor internally calls `fcbopen(filstr, fmtstr)` -- the same two strings from the previous slide. Nothing new to learn, only a different wrapping. Java: the same pattern, but over the JNI. An add-on with a JNI library plus Java classes, holding implementations of `InputStream` and `OutputStream`. The common point that holds the slide together: **both are skins over the byte interface**, not an access route of their own. Whoever has understood the two strings can use both. Name the precondition, because it is the most frequent question: for Java FLAM has to be installed, because the JNI library is part of the installation. And then the part that turns a language binding into an argument on the host -- it should not pass as a footnote: On z/OS there is an **MVS-based JNI**, and out of a 64-bit JVM it is reached through the **interoperability feature 64 to 31**. The reason is plain: the MVS world is 31-bit, in places even 24. But the gain is not the binding itself, it is what becomes reachable with it: the Java programmer has **Dynalloc, that is SVC 99**, fully available -- the dynamic allocation of MVS datasets when writing. That is finally full control over the MVS side, and **not only for FLAMFILEs**: for every kind of dataset a Java program wants to write on the host, so PS, PO and VSAM as well. Anyone writing Java on the host knows the problem -- and has solved it so far with JCL preparation or helper processes. This point is worth leaving in the room.



The slide that lifts this unit beyond the batch world -- and the one that was missing from the material entirely so far.
With the byte, the record and the member interface of the FLUC you can work not only **File to Memory** and **Memory to File**, but also **Memory to Memory**.
And from that follows something you only recognise on a second look: **all conversions of the FLUC can be applied to individual transactions in memory.** Not to a file with millions of records, but to a single operation -- online, within the response-time window.
The example that lands immediately in payments: **Instant Payment.** A transaction comes in as SEPA XML, has to go into a copybook format for the host application, and goes out again as SWIFT MT. Three formats, one transaction, everything in memory, without a single intermediate file.
That is exactly what an enterprise service bus does. The difference: an ESB is an additional system with its own runtime, its own operations and its own licence. Here it is a library call inside the application that processes the transaction anyway.
And the arc back to the fundamentals unit: there it was said that FLAM5 is an enterprise service bus for **persistent** data sets. With Memory to Memory that holds **online** as well -- for messages in flight. That is the complete sentence.
If somebody from architecture is sitting in the room, that is the point at which they listen: the format conversion for which they operate a product of their own is a function of the access method here.



The part that makes the difference in practice and is usually missing from examples.
Three levels: the return code states the category, the error number the concrete cause, the error text the explanation. Whoever logs only the return code has nothing in hand later.
A recommendation worth writing down: in the event of an error always log the return code, the error number and the error text together -- that is exactly what a ticket needs too.

PITFALLS

! **The subprogram and the record interface get confused.**
The subprogram takes whole files, the record interface individual records. The wrong choice costs weeks.

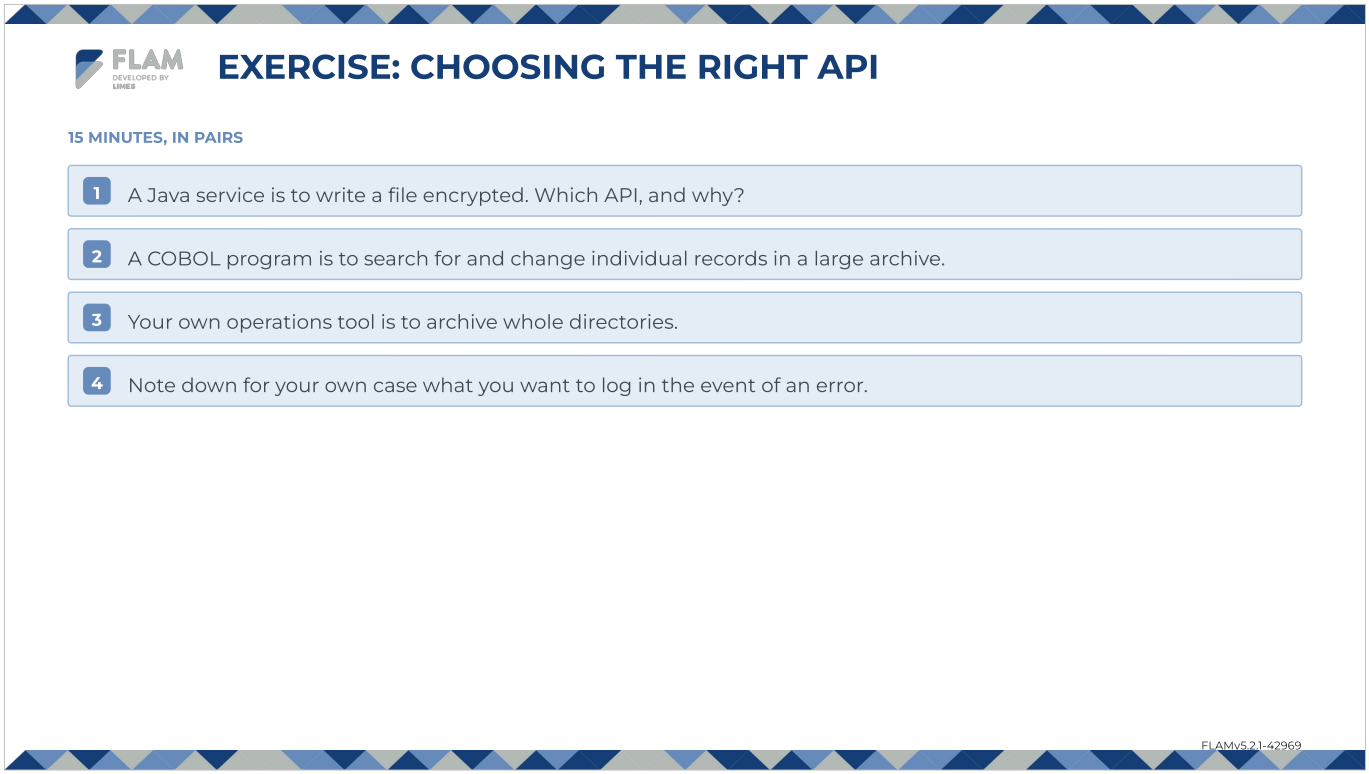
! **The parameter string is assembled inside the program.**
That produces quoting errors at run time. Build it in FLCC, store it as a constant, insert only the values.

! **Only the return code is logged.**
Without the error number and the error text the cause can no longer be determined later - not even by us.

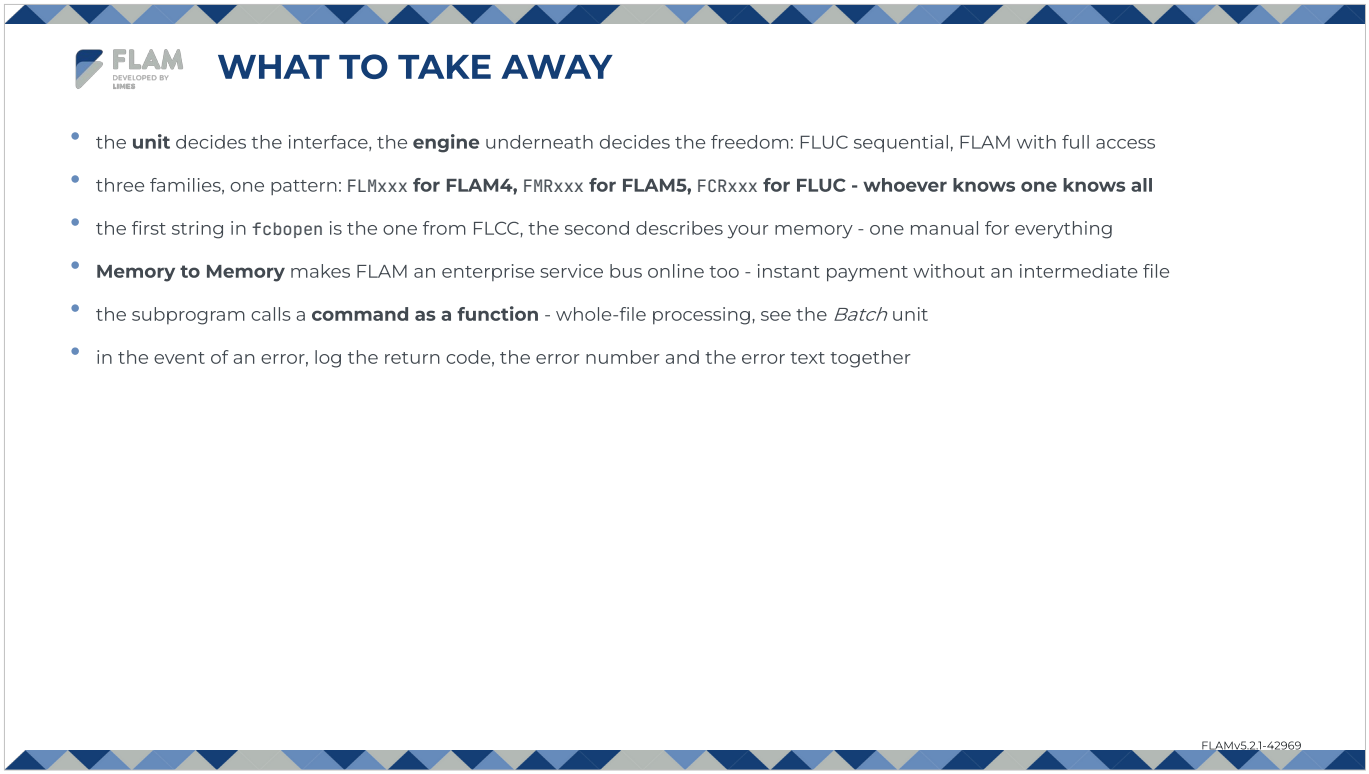
! **Java without a FLAM installation.**
The JNI library is part of the installation. Without it the error only shows up when the class is loaded.

FLAMv5.21-42969

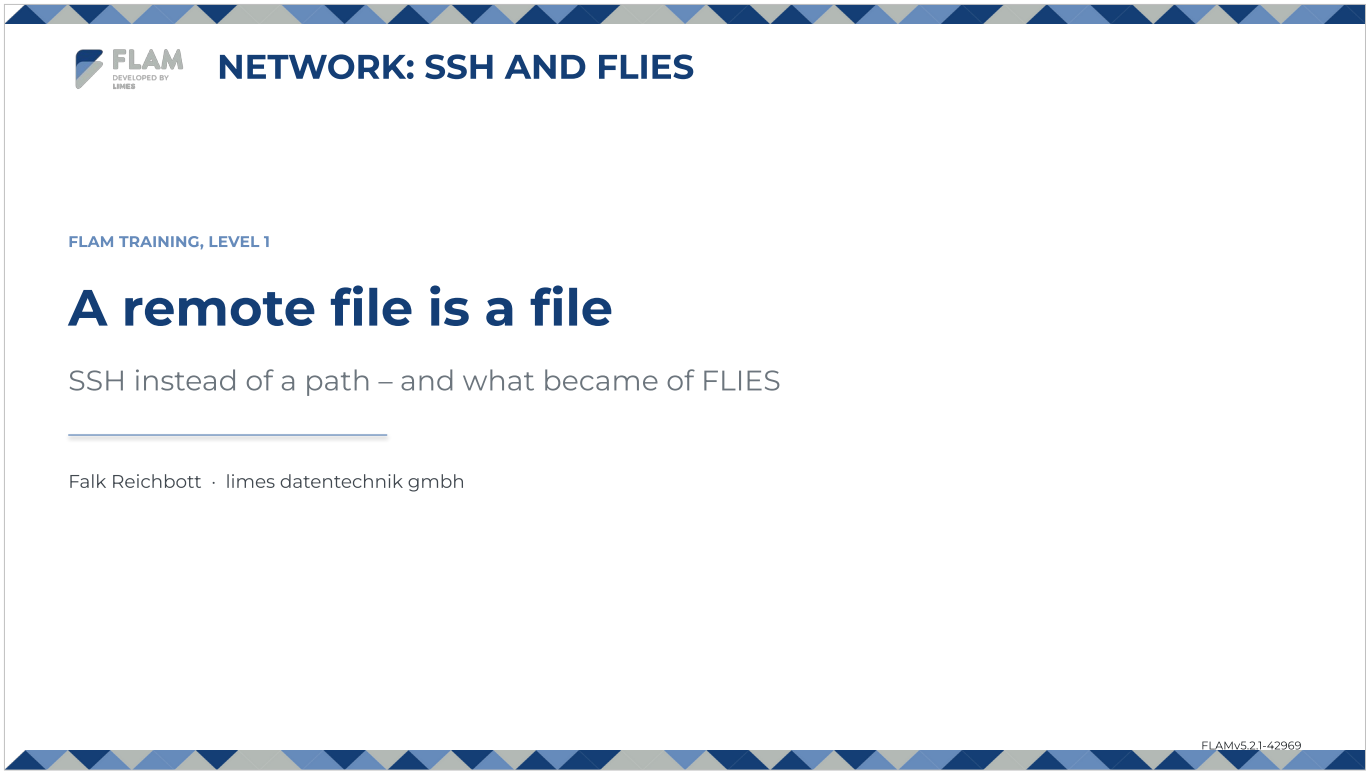
All four come from support cases of API use.



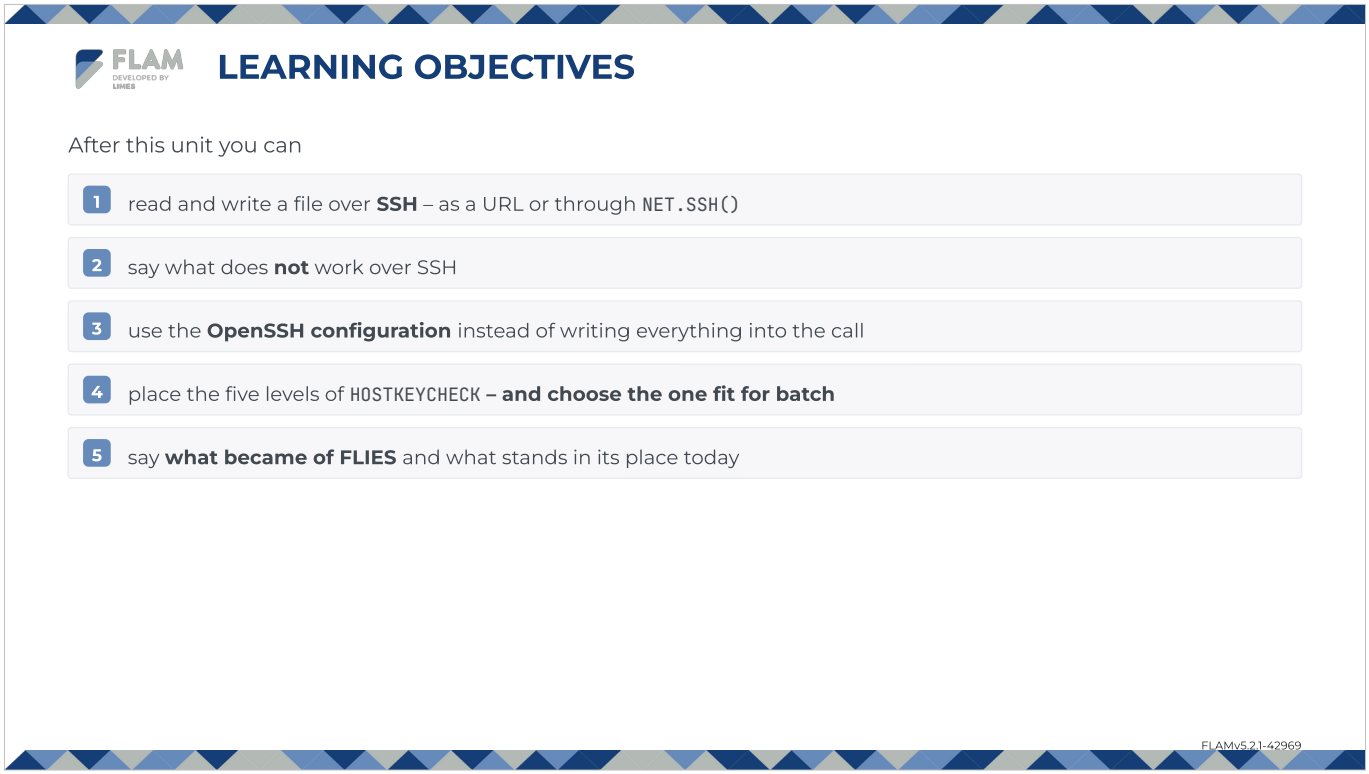
Fifteen minutes, in pairs. The goal is the reasoning, not the answer -- and the reasoning is always the unit in which the program thinks.



Go through the learning objectives. That completes level 1: all three routes have been shown once on the same parameter string.
The module families side by side once more, because the names come up in everyday work: `**FMRxxx**` are the load modules for the FLAM5 archives, `**FLMxxx**` those for the FLAM4 archives, and `**FCRxxx**` those for the FLUC record interface -- the latter only with `PUT`, `GET` and `LOC`, without `FND`, `INS` and `UPD`.
The functional topics follow -- archive, conversion, tables, crypto.



The fourth unit of level 1, and the shortest -- because the network access is deliberately unspectacular. The arc:
a URL instead of a path, what comes along with it and what does not, the authentication, the checking of the host key, and finally the question of what became of FLIES. The sentence for the start: **a remote file is a file.** Everything the previous units said about selection, conversion and name forming holds over SSH just the same -- you swap the path for a URL and nothing else.



The fourth objective is the one that gets asked in an audit -- and the one that is answered wrongly without these slides. The fifth is the reason this unit carries FLIES in its title at all:
the question comes up because the manual pages still describe it.

TWO SPELLINGS FOR THE SAME CONNECTION

THE URL WINS WHEN BOTH ARE THERE

```
# the short form: a URL where a path would stand
#   one slash: relative to the remote home directory
write.text(file='ssh://<cuser>@example.com/daten/aus.csv')


# two slashes: absolute, from the root of the remote host
write.text(file='ssh://<cuser>@example.com//srv/daten/aus.csv')


# the long form: everything a URL has no room for
write.text(file='daten/aus.csv'
           net.ssh(host='example.com' user='<cuser>'
                   prikeyfile='~/.ssh/id_ed25519'
                   hostkeycheck=STRICT))
```

On EBCDIC systems an ampersand is allowed instead of the @ as the separator – then the CCSID of the string need not be known.

FLAMv5.21-42969

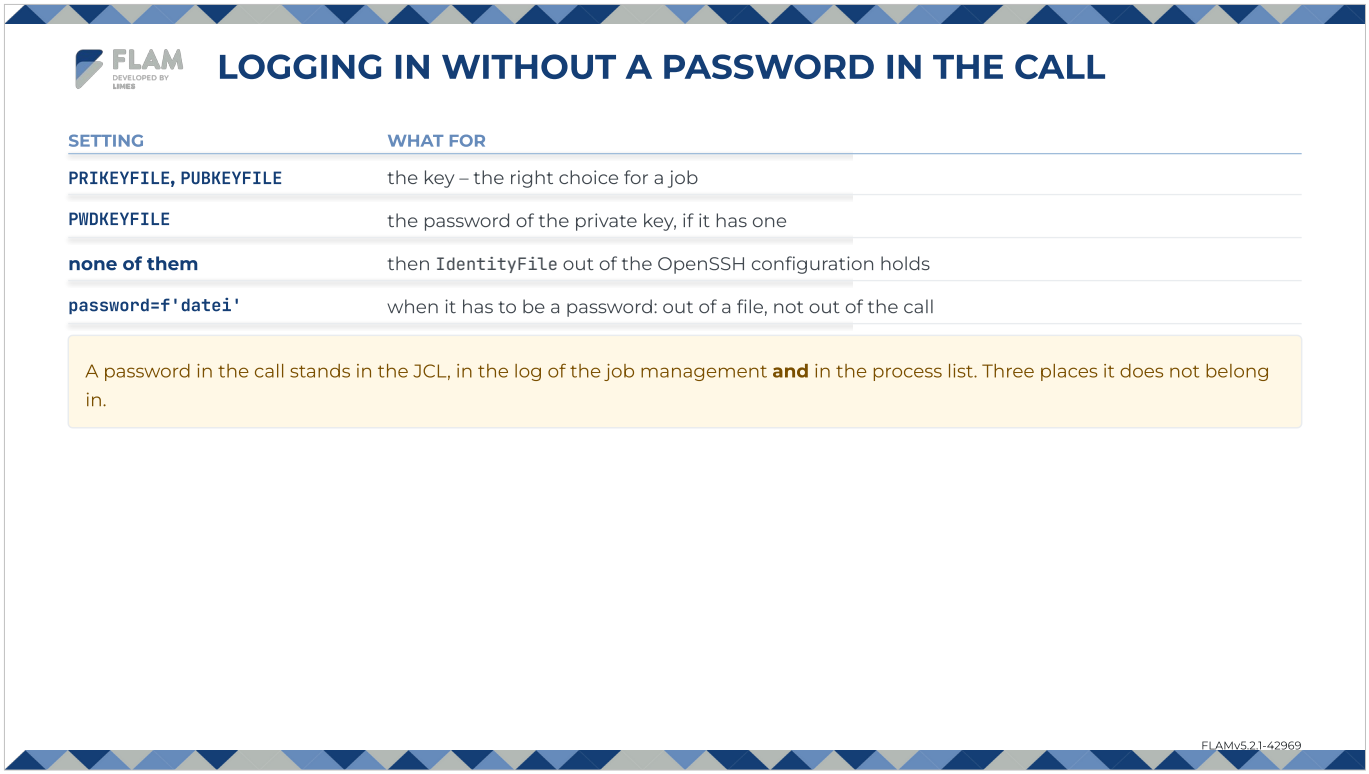
The slide you start with, because it settles the unit in two lines. The **URL** is the short form: user, host, optionally port and path. It stands where the path would otherwise stand -- at the `READ` as at the `WRITE`, and in an archive name too. The **`NET.SSH()` object** is the long form. It can do everything the URL can and in addition the things a URL has no room for: key files, known_hosts, the checking level of the host key, a time limit, algorithms. The rule when both are there: **the URL wins.** That is how the manual has it, and it is meant practically -- the connection parameters stand in the object, and the job overrides the host or the user on purpose. **The path in the URL counts the slashes.** One slash behind the host means: relative to the **home directory of the user** FLAM logs in with. Two slashes mean:
absolute, from the root of the remote host. A tilde does not belong in there -- it would be the local abbreviation and on EBCDIC systems it depends on the code page as well (Falk, 2026-09-16). And a detail for the host that comes up in every z/OS course: the at sign lies differently per EBCDIC code page. That is why an **ampersand** is allowed in a URL as the separator between user and server -- then the CCSID of the CLP string need not be known.



The slide that heads off a disappointment. What works: almost everything. The same kinds of file as locally, **wildcards** and **recursion into subdirectories**. A directory walk over SSH is therefore the same call as locally, and the name building holds just the same. The state of things is better by now than the old passage in the manual: **FLAMFILEs work over SSH**, and through our client **MVS data sets work directly** too -- in both directions.
`read.record(file='INPUT')` and `write.text(file='ssh://...')` is a valid call, and the other way round one reads remotely and writes straight into a PS or a PO member. The reason: on our side stands FLAM, and it knows records. The restriction lies elsewhere, and it matters because it comes up in the customer conversation: **when IBM's standard SSH daemon stands on the host side**, no MVS data set can be written with it -- SFTP knows files as byte streams, no records and no record formats. One then writes into USS first and copies the file into the MVS data set with a **post-processing step**, and deletes the USS file afterwards. What customers take for it: FLCL on the host as a post-process with "remove at read" -- then the host data set is written properly and the intermediate file disappears. A `cp` with a following `rm` does the job as well. And the real message of this slide: the encryption of the transfer is **always** there, even when the data itself is unencrypted. A recording on the line is of use to nobody. A practical piece of advice that stands on the slide and is often forgotten: **compress before transferring.** Our SFTP does not compress, and a `COMP.ZSTD()` in the same call costs no second step but saves the line -- with a text file by a factor of four. And a switch that belongs exactly here, because over SSH it means something different from locally: **`FLUSH`**. It stands in every write object -- `write.text`, `write.binary`, `write.record` -- and the help describes it with "Enforce flush in front of close file". What is meant: before closing, the file is really written out and not merely handed to the buffer. Locally that is `fflush` and then `fsync`, under Windows `FlushFileBuffers`. Over SSH FLAM does the same -- but only if the other side offers it: the SFTP server has to support the extension `fsync@openssh.com`. If it can, it is called, and after the close everything really is written. If it cannot, FLAM writes a line into the log ("fsync() extension not supported by server.
Output data might not be persisted, yet.") and the data is transferred but not guaranteed to be persistent. The practical case:
a follow-on job on the far side that processes the file immediately.
There one sets `FLUSH` and checks in the log once whether the SSHD plays along -- otherwise one is relying on a promise nobody gave.



The slide that makes a long call short -- and that hardly anybody expects. FLAM reads the OpenSSH configuration files: the user's (`~/.ssh/config`) and the system's (`/etc/ssh/ssh_config`). What stands there holds for FLAM too -- Ciphers, Compression, ConnectTimeout, HostKeyAlgorithms, IdentityFile, KexAlgorithms, MACs, Port and the usual others, in OpenSSH syntax. In practice that means: if the connection works with `sftp`, it works with FLAM too, without writing anything into the call.
The order of precedence is the one you expect, and the manual has it: the call, then a configuration file of your own through `CONFIGFILE`, then the user's, then the system's. Whoever does not want the two defaults switches them off with `NODEFAULTCONFIG` -- that is the handle for a production job that should not depend on what stands in a user's home. The advice for operations: put the connection data into a configuration file of its own and point at it with `CONFIGFILE`. Then only the host stands in the call, and the operations team maintains the rest in one place.



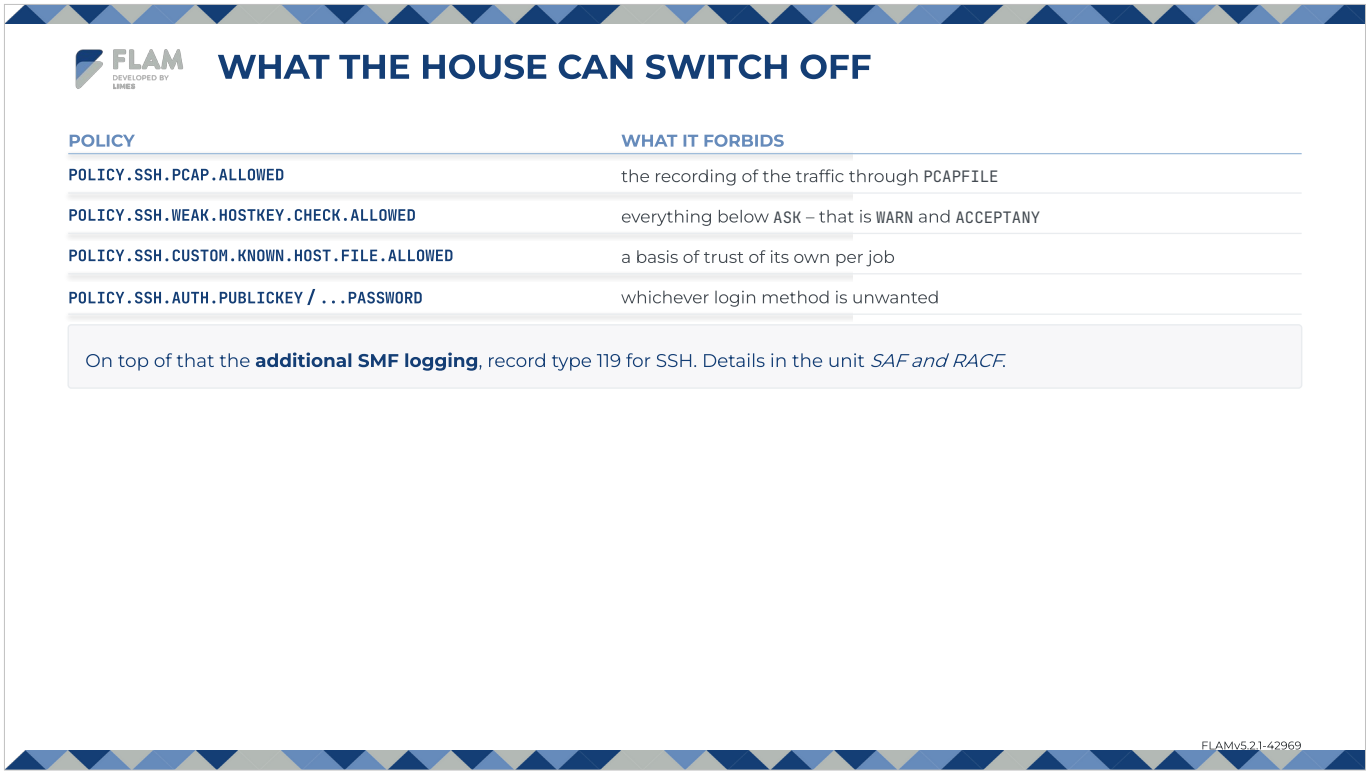
The slide with the security statement that is the easiest one to act on. There are two methods: password and public key. For a production job the key is the right choice -- no secret in the call, no expiry, no change when a password rule changes.
The entries for it: `PRIKEYFILE` the private key, `PUBKEYFILE` the public one, `PWDKEYFILE` the password of the private key if it has one. Or none of them -- then FLAM takes what stands as `IdentityFile` in the OpenSSH configuration. And if it has to be a password after all: **not into the parameter string.** `password=f'/pfad/pw.txt'` reads it out of a file that is protected with file permissions; the log shows `***SECRET***`.
That is the entry from the language unit, and here is its most important use. The sentence for the room: a password in the call stands in the JCL, in the log of the job management and in the process list. Three places it does not belong in.



The most important slide of the unit. It decides whether the connection is really secure -- the encryption is of no use if you encrypt with the wrong counterpart. The server identifies itself with its public key. Known keys stand in `~/.ssh/known_hosts`, in the format of OpenSSH. `HOSTKEYCHECK` says how strictly it is checked, and the manual lists the levels **in descending security**: At the top stands the **fingerprint** itself. You give it, and it has to match -- otherwise the connection is ended. If the key is not in the file yet, it is added. That is the entry for a **batch job**: nobody can answer a question there, and you still accept no arbitrary key. The fingerprint is printed in the log at every connect. `STRICT` demands an entry in the file, `ASK` asks the first time, `WARN` connects even with an unknown key and writes a warning, and `ACCEPTANY` even adds any key to the file -- which makes it trusted in future strict runs. The manual warns explicitly about everything below `ASK`. The sentence for the room: **whoever uses `ACCEPTANY` once has given the protection up for good** -- not only for this run.



A short slide with a statement that saves work. Pre- and post-processing is executed on the machine the file lies on.
If you write over SSH, the commands that belong to it run **on the other side** -- so you need no second job and no second SSH session to start something there. Two forms: with `XCNV` the commands exist **per file**, and there the file name is available through the bracket patterns. With `CONV` they exist globally -- once before and once after the run. Whoever needs something per file takes `XCNV`. And the specification needed for it: the `POSTPROCESS` holds a `NET.SSH()` of its own, which says explicitly where the command runs. The conversion unit shows the case completely -- check the checksum, then unpack. The advice: the command runs **without a shell**. Whoever needs a chain calls a script on the other side. From that follows a small thing that otherwise costs time:
without a shell no tilde is resolved either. The command starts in the home directory of the user, so a relative path like `bin/uebernahme.sh` hits -- a tilde in front of it would be a character in the file name and the call fails.



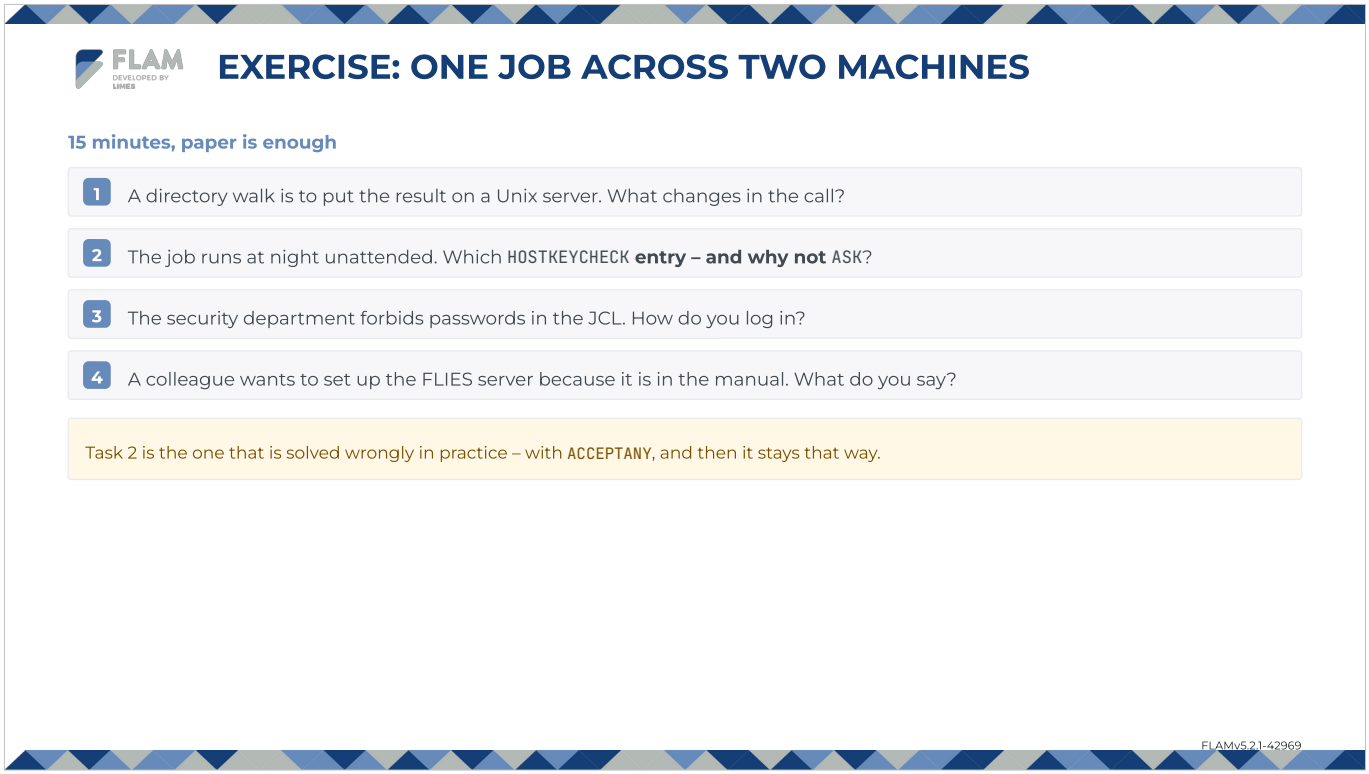
The slide that connects this unit with the security group -- keep it short, it is there in detail. On z/OS a RACF administrator can tighten the screws shown here:
The **packet capturing** through `PCAPFILE` records the traffic -- indispensable for a fault search and an open door in production. `POLICY.SSH.PCAP.ALLOWED` forbids it. The **weak host key check** -- everything below `ASK` -- can be forbidden through `POLICY.SSH.WEAK.HOSTKEY.CHECK.ALLOWED`. After that a job can write `ACCEPTANY` as often as it likes. A **known_hosts file of its own** can be forbidden through `POLICY.SSH.CUSTOM.KNOWN.HOST.FILE.ALLOWED` -- otherwise a job brings its own basis of trust along. On top of that come `POLICY.SSH.AUTH.PUBLICKEY` and `...PASSWORD`, with which the house decides which login method is allowed, and the **additional SMF logging** with record type 119 for SSH. The sentence that makes the connection: an entry in the job is a wish, a policy is a rule. The SAF unit says how a rule is made out of it.



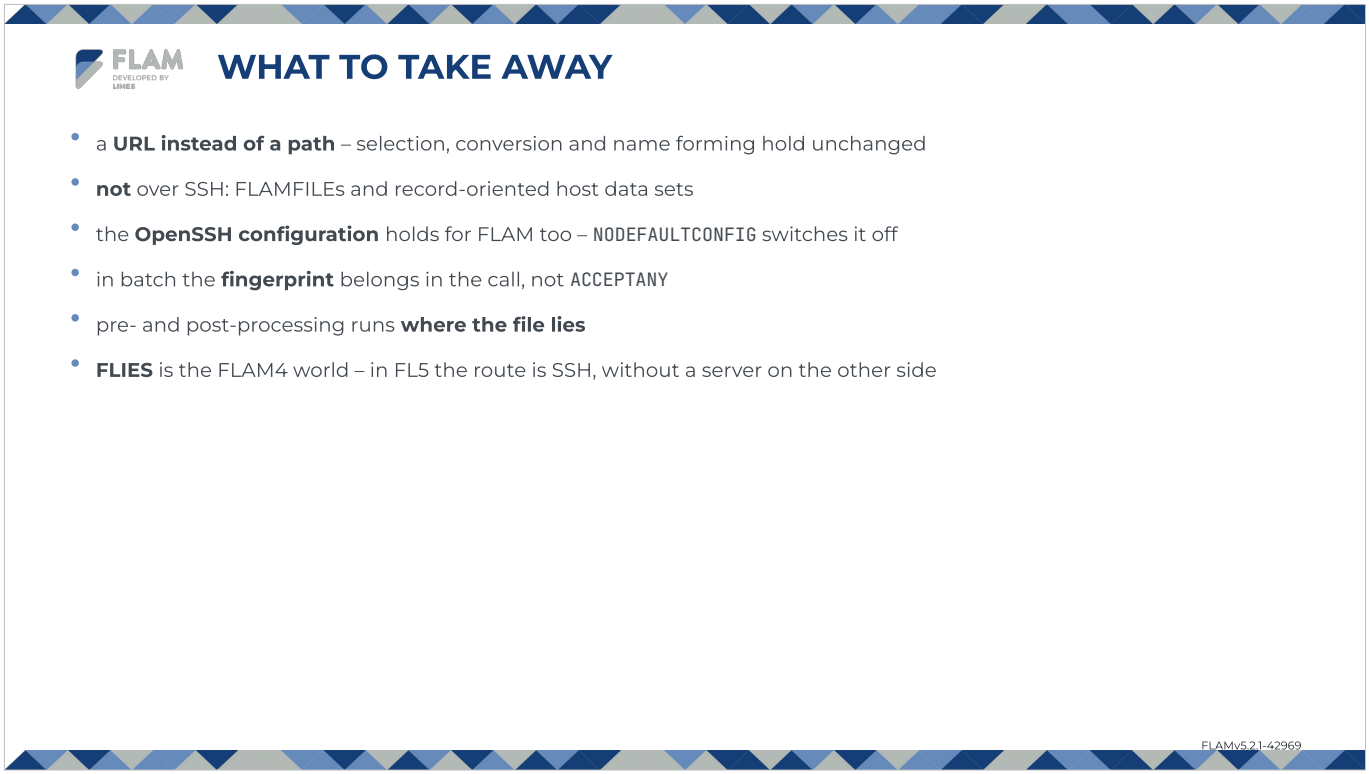
The slide that answers a question coming out of the manual pages -- and it answers it honestly. **FLIES** stands for Frankenstein Limes Integrated Extended Security. In the FLAM4 world it is the server remote accesses ran through, with three methods: **IPN** as native IP on port 17996, **IPS** the same with SSL/TLS, and **MQS** over IBM MQ-Series. For all three a FLIES server had to run on the other side. And today? Measured: the three stand in the parameter table, but with a flag that does **not** put them into the command language. The parser rejects `net.ipn(...)` -- "parameter not valid". So what really is there in FL5 is **SSH**, and for archives in an object store the cloud attachment. Why that is no loss, and it is worth saying this way: SSH needs **no FLAM server** on the other side, only an SSH service -- and that runs everywhere. A server service of your own is operations, monitoring and a port to be opened; an SSH service is there already. If the question about the manual pages comes: they describe the FLAM4 world. That is a documentation finding, and we have noted it. And there is a second way over the line that plays into this and should be named, because it shows the direction:
HTTPS. Through **FLREST** we talk to the cloud object stores, and the attachment of IBM CDA runs over it as well. When we finish the **FLSERV**, it will transfer its segments over HTTPS with FLREST too -- so out of IPN, IPS and MQS becomes **RST**. A protocol of our own on a port of its own is not coming back.



All four out of support cases.



Fifteen minutes, paper is enough. The solutions: **Task 1.** `file='ssh://<cuser>@server/daten/[base].csv'` -- the URL stands where the path would otherwise stand, and the name building applies unchanged. **Task 2.** Give the fingerprint. `ASK` nobody can answer in batch, `STRICT` fails on the first connection, and `ACCEPTANY` gives the protection up for good. The fingerprint stands in the log of every connection setup. **Task 3.** Use a key: `prikeyfile=...`. If it has to be a password, `password=f'datei'` with file permissions. **Task 4.** No. FLIES is the FLAM4 world; in FL5 the command language does not accept the three methods -- measured. The way is SSH, and that one needs no FLAM server there.



Go through the learning objectives. The pointer forward: where an **archive** physically lies -- object store, cloud -- is in the unit *Segment store and cloud*. This unit was the access to **files**.

FLAM TRAINING

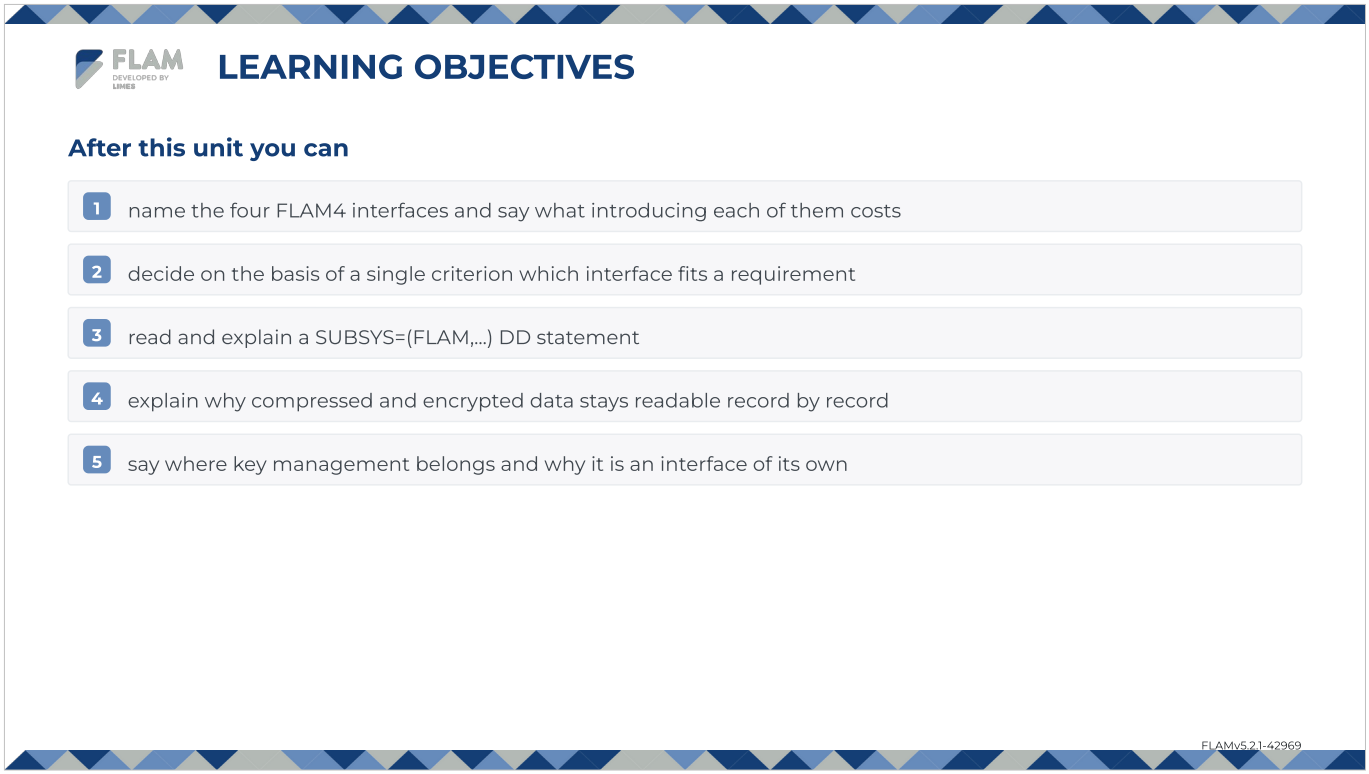
Four ways into the same method

and how to choose the right one

Falk Reichbott · limes datentechnik gmbh

limes: performance at the limits of what is feasible.

Set the frame: FLAM4 is not a compression tool with an API bolted on, it is an access method with four entrances. The whole unit is about choosing between those four entrances.
Ask first which of them are in use in the room today -- most sites use the utility and the subsystem and do not know that the record interface exists at all.



The five objectives are the contract for this unit; come back to them in the summary. The one criterion in objective two is the responsibility for the change: who has to touch something -- operations, build or source code.

ONE ACCESS METHOD, FOUR ENTRANCES

The interface decides **who has to change something** – not what comes out at the end.

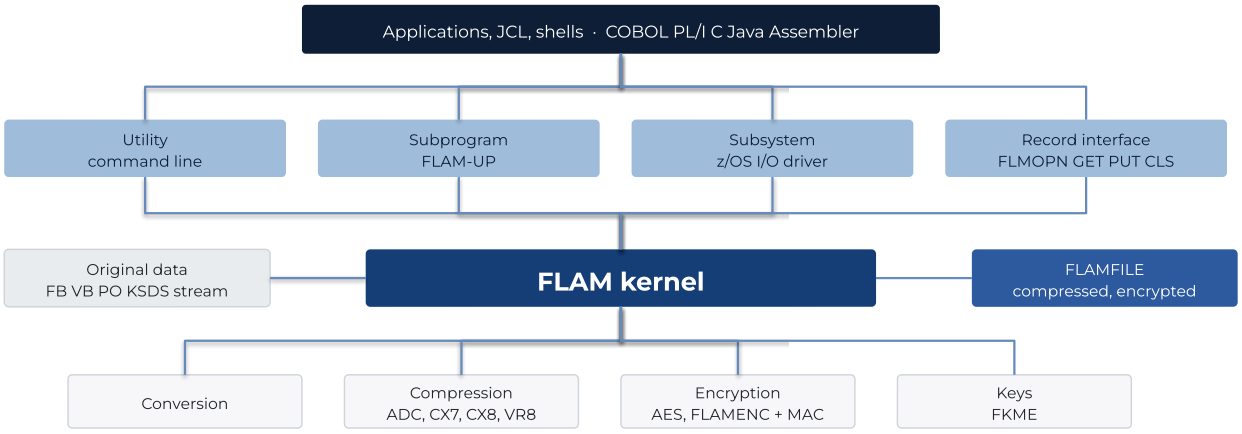
INTERFACE	WHO CHANGES SOMETHING	TYPICAL TRIGGER
Utility	operations, in JCL or a script	migration, exchange, batch
Subprogram	the build, binding a module	your own tool, your own utilities
Subsystem	operations, one DD statement	the application must not be touched
Record interface	the application source code	the program itself works record by record

The FLAMFILE is the same in every row – a decision here is therefore never final.

FLAMv5.21-42969

This slide carries the whole unit. The FLAMFILE is identical in all four cases and interchangeable between them -- which is why the decision is purely organisational.
The old slides presented the four interfaces as four product data sheets; that obscures the only question a customer has to answer.

FLAM4 IN THE PICTURE

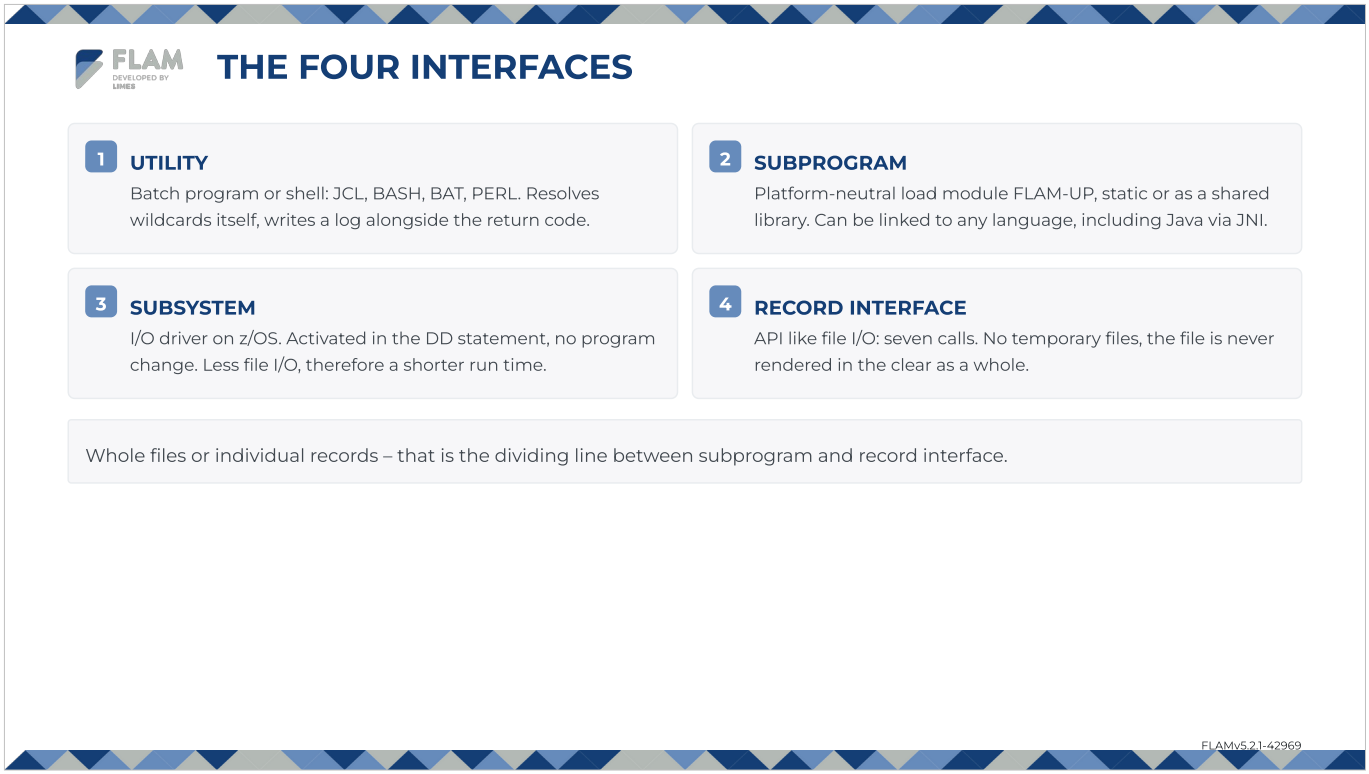


Every path also runs backwards – that is what distinguishes an access method from a converter.

`MODE=` selects both in one parameter: `ADC` `CX7` `CX8` `VR8` for compression, `AES` and `FLAMENC` for encryption, `NDC` for "do not compress". Note: with `CX7`, `CX8` or `VR8` **no** encryption is possible.

FLAMv5.2.1-42969

Work through it from top to bottom. The four interfaces sit on the same kernel; the kernel converts, compresses, encrypts and computes the MAC, and it fetches key material via the key management exit.
Stress that all the routes also run backwards: that is what makes FLAM an access method and not a converter.



The distinction between subprogram and record interface is often confused:
the subprogram takes whole files, the record interface takes records.
When someone says "we want to call FLAM from our own tool", the answer is almost always the subprogram, not the record interface.

SUBSYSTEM: THE DD STATEMENT DOES THE WORK



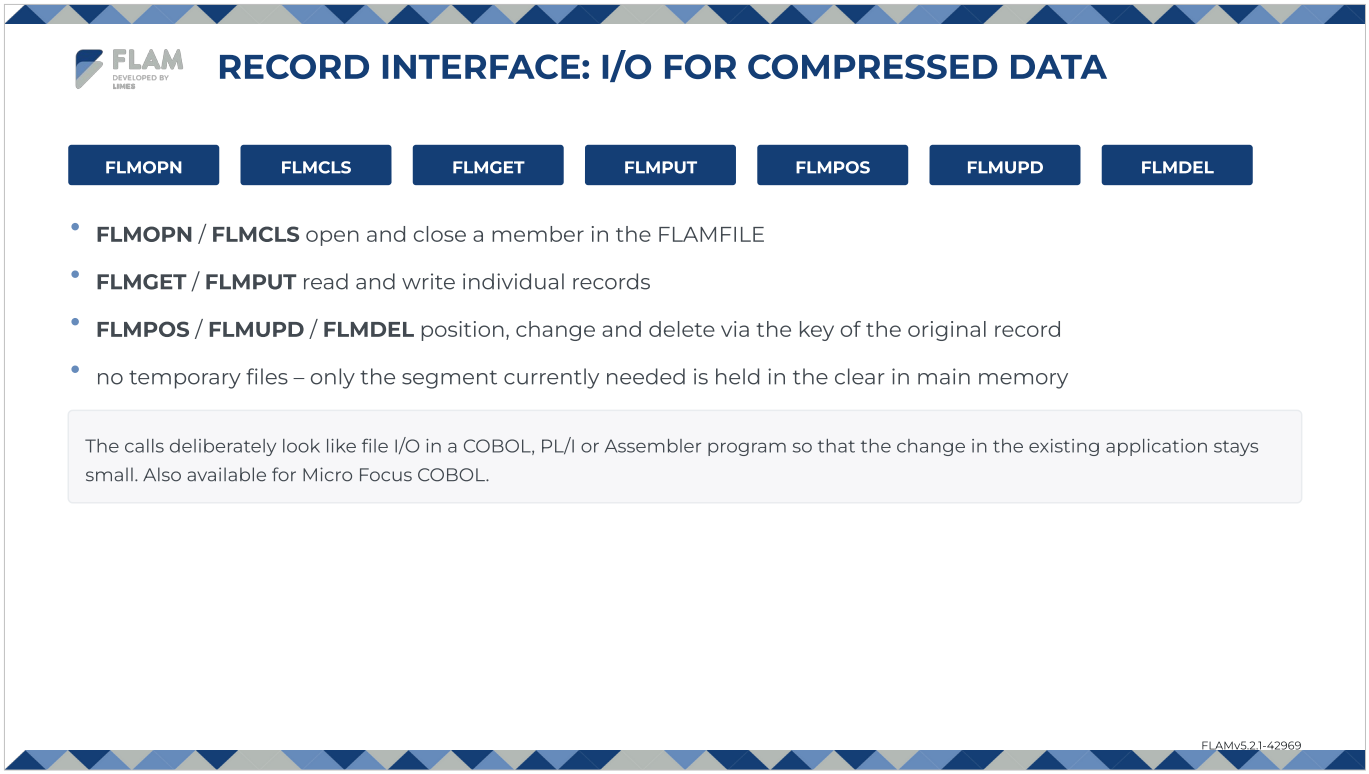
The application keeps issuing ordinary QSAM or VSAM calls and never learns that the data is compressed and encrypted.

```
//DDNAME  DD  DSN=filename,
//            SUBSYS=(FLAM,'CRYPTOM=AES',
//            'KME=FKMEFILE,KMP=C''ddname1''')
```

Source: **doc/man_flam4_appendix_flamsub.txt** – the double quotation marks are the trap.

FLAMv5.21-42969

Source of the example: doc/man_flam4_appendix_flamsub.txt, section "Encryption using FKMEFILE". Do not write this syntax onto a slide from memory -- the doubled quotation marks are the part everyone gets wrong.
The application keeps issuing ordinary QSAM or VSAM calls and never learns that the data on disk is compressed and encrypted.
A second effect that often sells: compression means less file I/O, so the unload gets faster while it is additionally protected.



The seven entry points are the whole interface; that is the point, because you expect something big from an access method.
Security argument: only the segment currently needed exists in the clear, and only in main storage. That is exactly what several security standards demand, and that is why no temporary clear-text file is ever created.
Also available for Micro Focus COBOL -- comes up regularly when a customer moves a mainframe application to Linux.

FINDING ONE RECORD IN A LARGE ARCHIVE

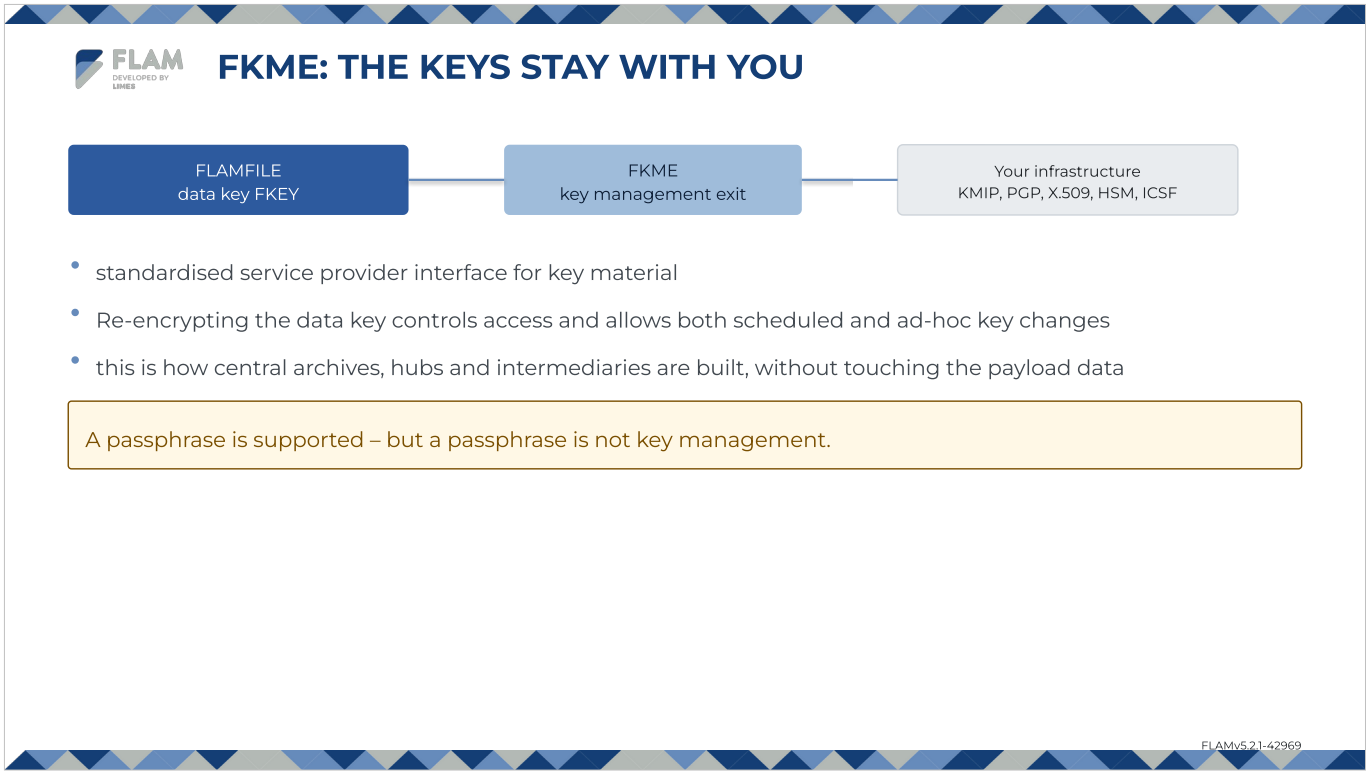
- index-sequential FLAMFILEs: read, write, change, insert, delete – via the key of the original record
- FLAM takes the key at the record interface and uses it to find the right segment
- the search does not decompress the rest of the archive
- on z/OS transparently through the subsystem with VSAM KSDS

This makes an archive with real search affordable: the result set stays encrypted until someone is authorised to see it.

Continued in the deck FLAM5 archive: there matrix, segments and Bloom filters replace the mechanism – the promise stays.

FLAMv5.21-42969

The mechanism is the segment: the key selects a segment, and only that one is decoded. The cost per hit stays roughly constant as the archive grows -- this property is what turns a compressed backup into a queryable archive.
This slide is the bridge to the FLAM5 deck: FLAM5 replaces the mechanism (matrix, segments, Bloom filter), but keeps the promise.



The purpose of the interface is that the customer keeps his own key processes and authorisation rules; FLAM does not become a second key store.
Concretely: the data key of a FLAMFILE (FKEY) is protected by a master key (FMKY) that resides in the customer infrastructure. Re-keying FKEY hands the archive over to a different recipient without touching the payload -- that is how central archives, hubs and intermediaries are built.
The last sentence is the one to remember; write it on the whiteboard.

ONE QUESTION, FOUR ANSWERS

IF THE REQUIREMENT IS	THEN	BECAUSE
exchange or migrate whole files	Utility	no code, no build, scriptable
our own tool is to do it	Subprogram	bind in a module, file level
this application must not be changed	Subsystem	JCL only, transparent
the program processes individual records	Record interface	record level, small change
the key must come from our HSM	FKME, in addition to any of the above	key management is orthogonal

A site can start with the utility and switch to the subsystem later, without converting data.

FLAMv5.21-42969

Hand this slide out. It is the one that gets photographed.
Stress the last line: because the container is identical, a house can start with the utility and later switch to the subsystem without converting data.

PITFALLS FROM THE SUPPORT CASES

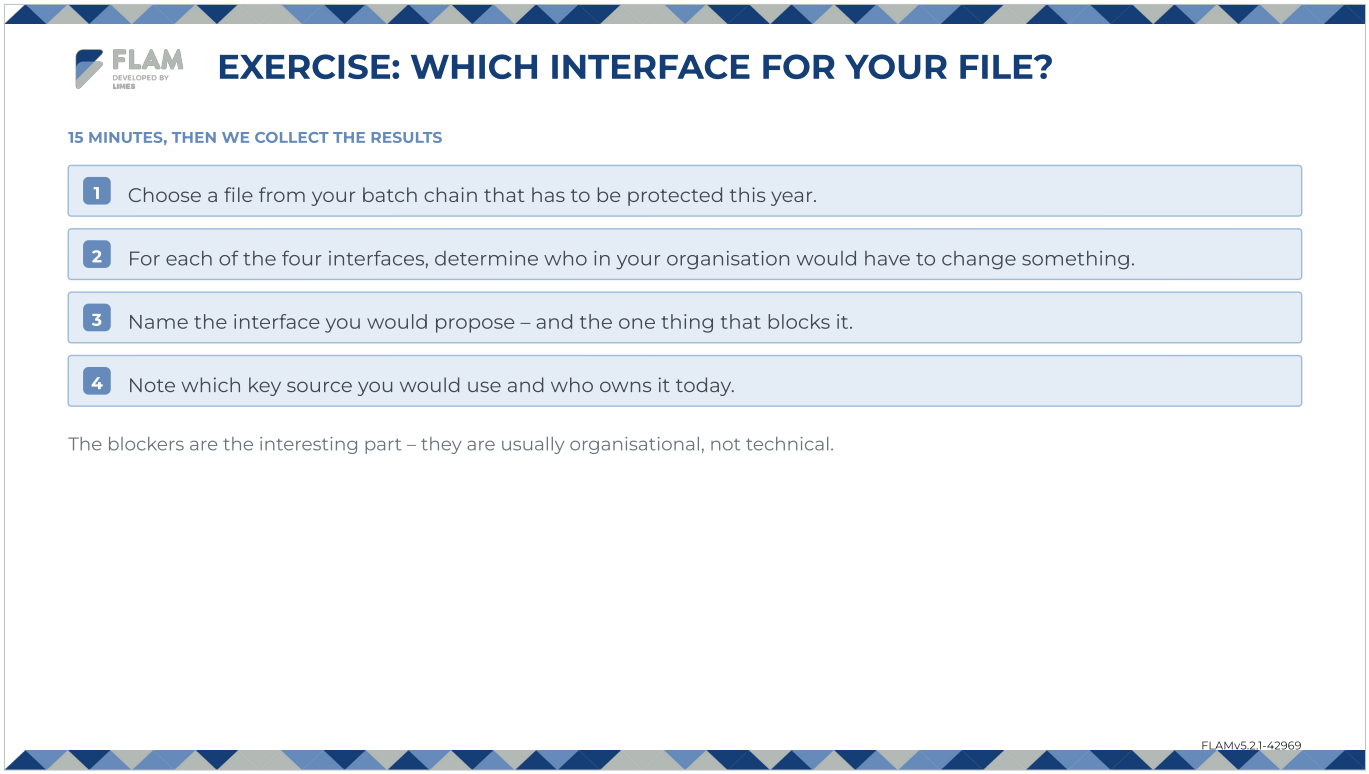
! **Random updates via VSAM void the integrity protection.**
A MAC over the whole file cannot survive record-wise write access. A design trade-off, not a defect.

! **The double quotation marks in SUBSYS= are a JCL trap.**
FLAM rejects invalid parameters as a JCL error, not only at run time.

! **The job user needs access to the key, not only to the data.**
With ICSF that means label and service, otherwise the job fails at open.

! **A FLAMFILE read without SUBSYS= returns the compressed bytes.**
That is not corruption, that is a missing allocation.

All four come from support cases, and all four are cheap to avoid.
The first one is a genuine design trade-off, not a defect: the customer chooses between record-level write access and integrity evidence at file level. Say that openly, it builds trust.



Collect in four columns, one per interface. The blockers are the interesting part -- they are mostly organisational, not technical, and they tell you what the follow-up meeting has to be about. Keep the results, they fill the requirements list at the end of the workshop.
Solution: the criterion is always which unit the application thinks in and who is allowed to change anything.
- A legacy application that must not be touched: subsystem. The application notices nothing, what changes is the allocation.
- An in-house tool that processes whole files: subprogram.
- COBOL or PL/I with record processing: record interface.
- A job that prepares a file and passes it on: batch utility -- no code, no build.
The most frequent wrong choice in the room is the subprogram when the subsystem is actually meant: anyone who does not want to change any code wants the subsystem.

FLAM
DEVELOPED BY LIMES

WHAT TO TAKE AWAY

- one access method, four entry points, one identical FLAMFILE
- the choice affects who is responsible for the change, not the result
- the subsystem is the answer when the application must not be touched
- the record interface is I/O with seven calls, not a framework
- key management is deliberately a separate interface

FURTHER READING

Subsystem, DD syntax, limits	doc/man_flam4_appendix_flamsub.txt
Key management exit	doc/man_flam4_appendix_kmexit.txt
User I/O and user exits	doc/man_flam4_appendix_userio.txt
What FLAM5 changes	Deck FLAM5 archive

FLAMv5.21-42969

At the end read the learning objectives out once more and ask whether all five have been reached. The next deck in the recommended order is FLAM5 archive; the comparison of FLAM4 against FLAM5 is there, not here.

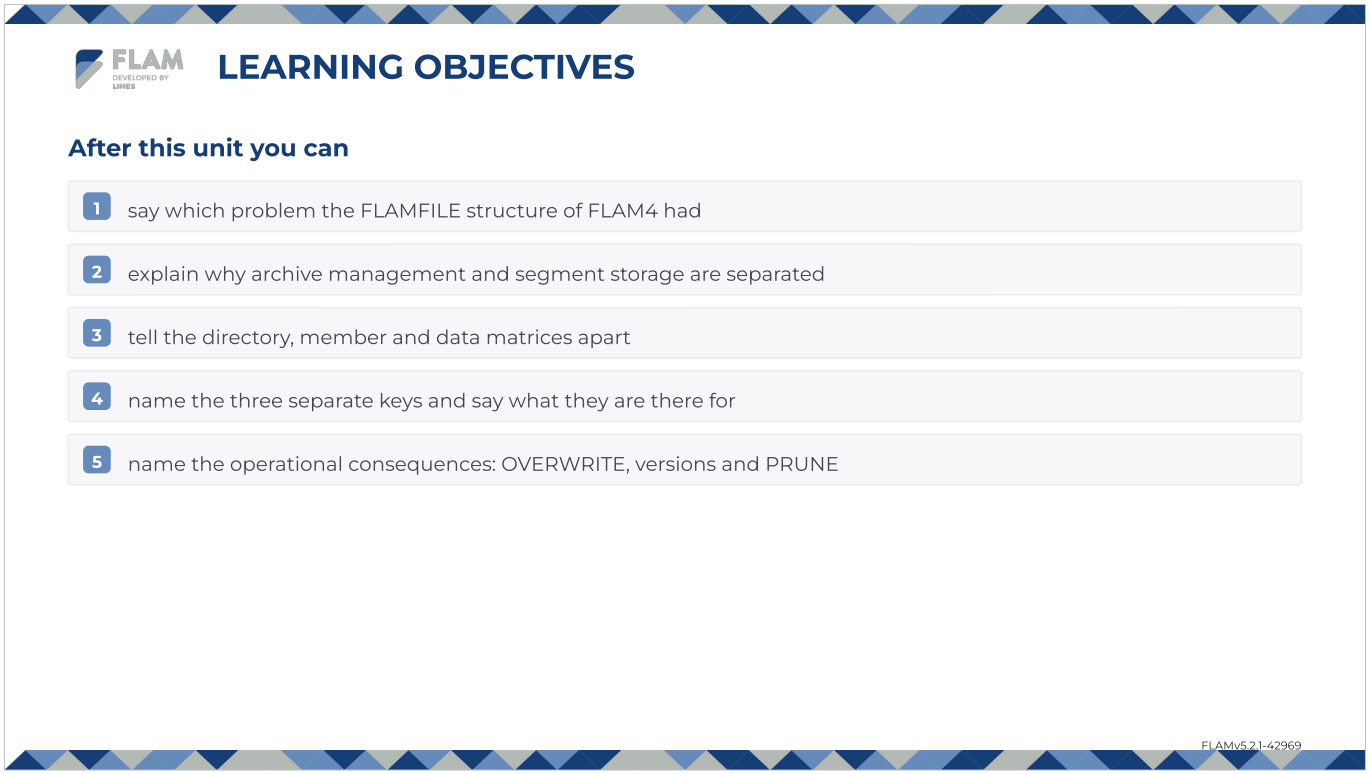
FLAM TRAINING, LEVEL 2

From a compressed file to a file system

Structure, keys, versioning and storage

Falk Reichbott · limes datentechnik gmbh

First functional topic. The unit explains why there was a new archive format -- and the answer is not "faster" but "manageable".
The arc: what FLAM1 to FLAM4 could do and where they hit the wall, what became a requirement out of that, and how FLAM5 is built.



Objective two is the core: the separation of logical archive management and physical segment storage is the idea from which everything else follows.

FLAM DEVELOPED BY LIMES

LOOKING BACK: FLAM1 TO FLAM4

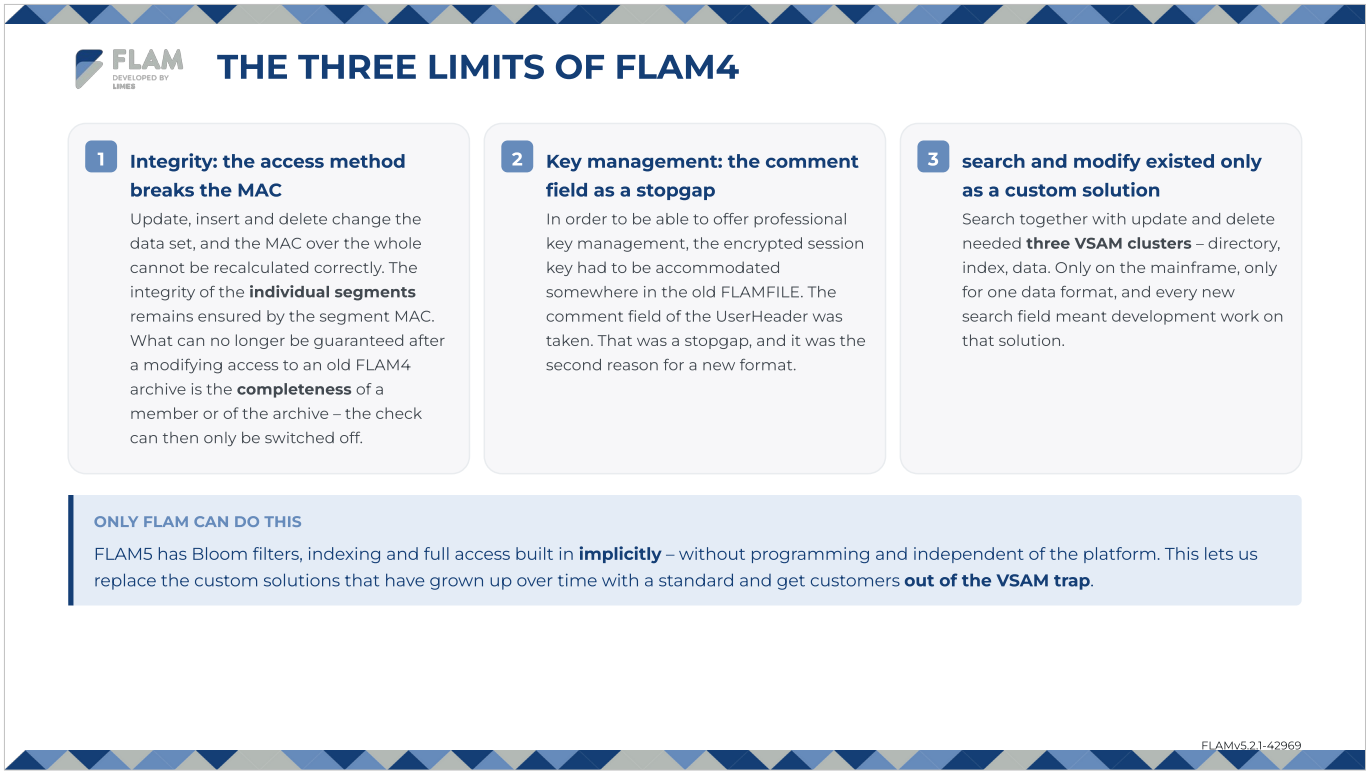
STAGE	WHAT WAS ADDED	WHAT REMAINED
FLAM1	self-contained segments, matrix in memory, the segment header describes the structure	one file, one compressed output
FLAM2	member header, and with it the first archive; FLAM-VSAM with a signature file for the search	customers concatenate FLAMFILEs
FLAM3	several files in one compressed output, the member list can be listed	concatenation still works without problems
FLAM4	AES, integrity protection via MAC, and FKME for HSM-based key management	**the structure reaches its limit**

ONLY FLAM CAN DO THIS

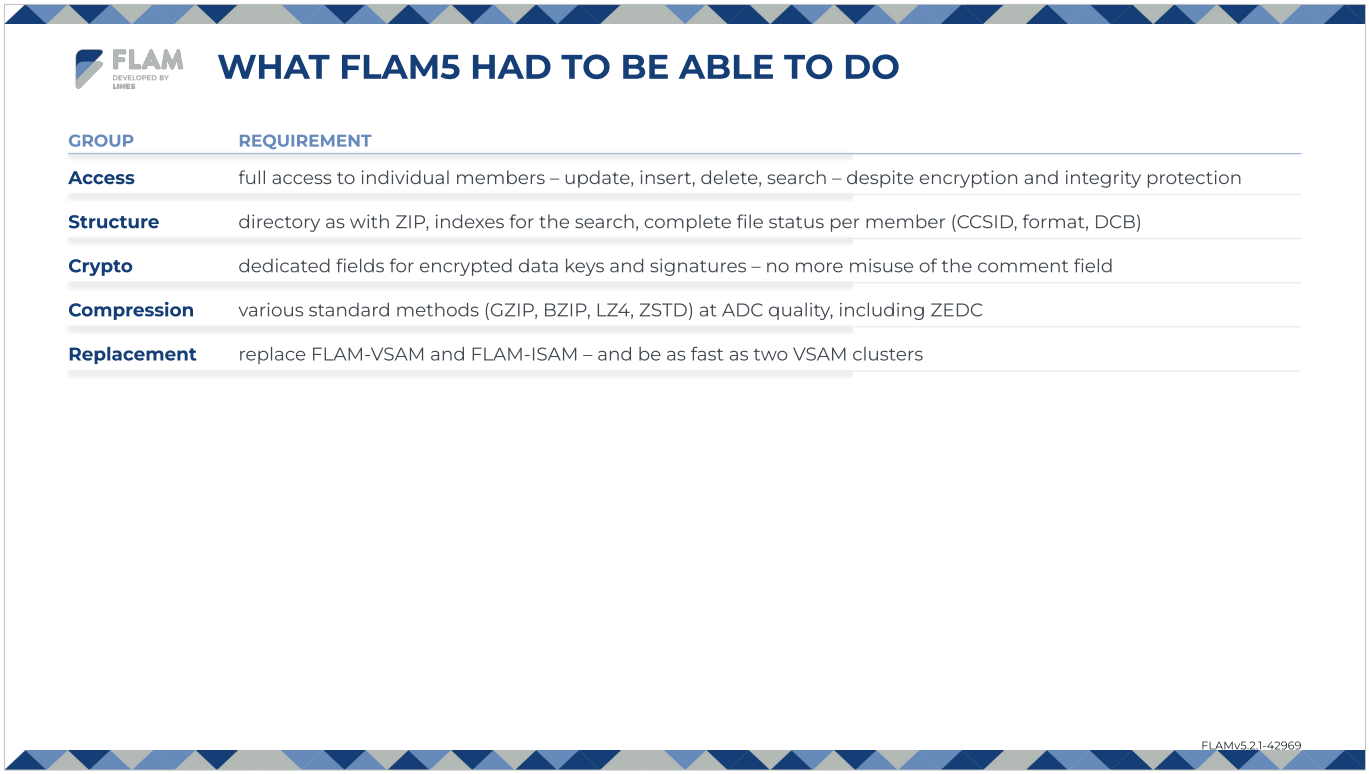
Every stage was backward compatible. FLAM still reads a FLAMFILE from the early nineties today – and that is the reason why the large data sets in the data centres are FLAM4 archives.

FLAMv5.21-42969

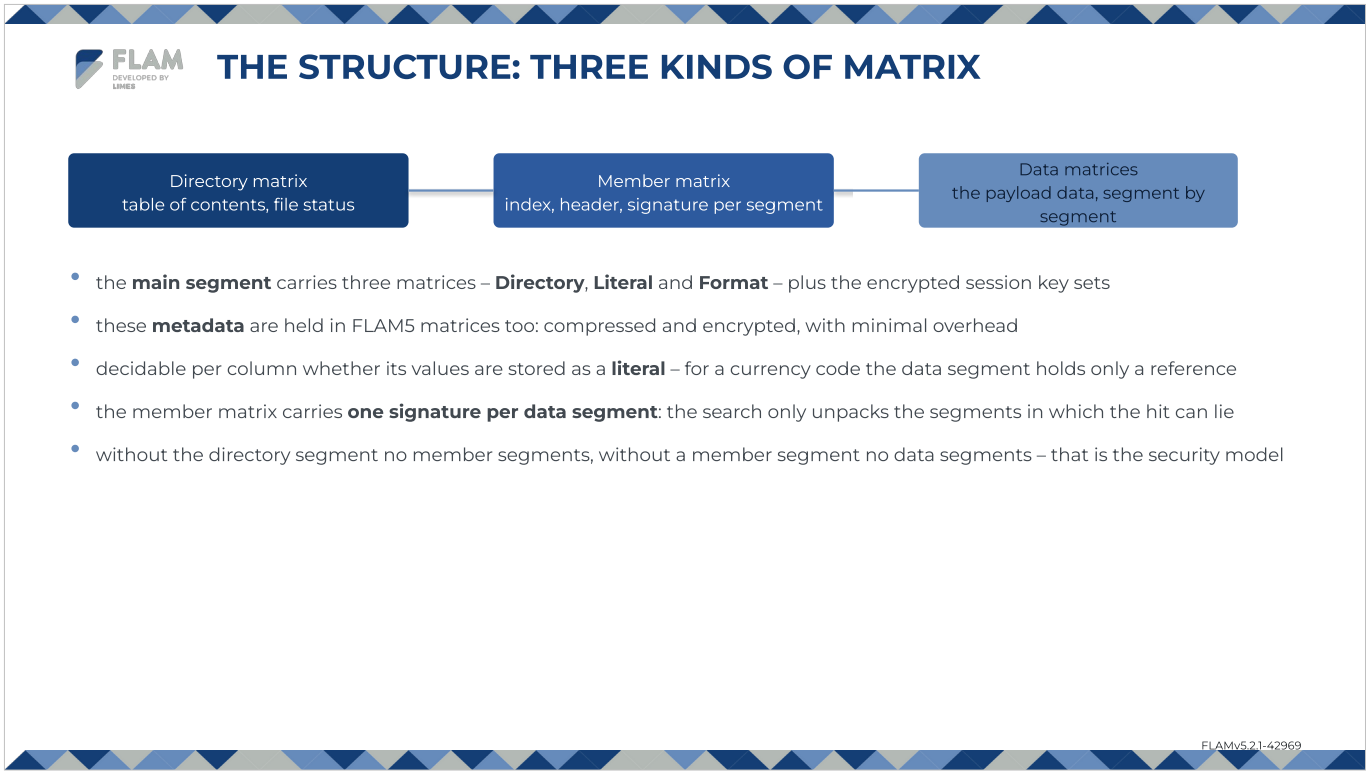
This slide explains why there was a new format -- without it FLAM5 sounds like an end in itself. Every stage was backward compatible, and every stage brought exactly one new capability. That is the real achievement of this line: 30 years of format without a break.
The last line is the decisive one, and it leads on to the next slide:
with FLAM4 came professional key management, and with it the design reached its limit in two places. Only announce it here, do not elaborate.



This is the slide that justifies why FLAM5 was built. Name both points honestly -- anyone who knows them understands the architecture of FLAM5 immediately, and in a house with FLAM4 data sets that is information of practical relevance, not self-criticism.
First limit, the integrity protection. In the FLAM4 world the access method itself breaks the MAC: update, insert and delete change the data set, and the MAC over the whole cannot be recomputed correctly. What matters is the precise distinction, because it saves the statement: the integrity of the individual segments remains guaranteed via the segment MAC.
What can no longer be guaranteed after a modifying access to an old FLAM4 archive is the completeness of a member or of the archive. The only option then is to switch the check off -- that is, to do without the completeness check.
Second limit, the key management. In order to be able to offer professional key management, the encrypted session key had to be accommodated somewhere in the old FLAMFILE. The comment field of the UserHeader was taken. That was a stopgap, and we call it that too.
Third limit, and in practice it is the most expensive one: **searching and finding together with update and delete existed in FLAM4 only as a special solution.** In this project it is called FLM4SEPA. There **three VSAM clusters** work together -- one for the directory, one for the index, one for the data. That ran only on the mainframe, was implemented for **one** particular data format, and every extension of the search fields was again development work on that solution.
That is exactly what the FLAM5 archives have built in **implicitly**: Bloom filters, indexing, full access -- without anyone having to program, and platform independent.
And that is the statement that counts in a house with such solutions: **we are replacing these special solutions with a platform-independent standard and thereby getting the customers out of the VSAM trap.** Anyone who today maintains three clusters in order to be able to search in his archives swaps them for an archive format -- and takes it along at the next platform change.
The closing sentence: these are exactly the points that the FLAM5 archives were meant to fix -- full access, search without a special solution, and integrity and completeness protection are preserved while doing it.



Do not read out all the points; the list was long and is in the old slides.
Group them and explain the four groups.
The most important line is the second one: full access -- update, insert, delete, search -- on individual members, with professional encryption. That is the requirement that determines the whole design.
The replacement for FLAM-VSAM and FLAM-ISAM is the reason why it was urgent at all, and the benchmark was: as fast as two VSAM clusters.



Explain from the bottom up: data matrices hold the payload, member matrices the indexes and signatures per segment, the directory matrix the table of contents. Now in more detail, because the roles always get mixed up in conversation: The **directory matrix** describes which **members** the archive consists of. The **member matrices** describe which **data segments** a member consists of -- and they carry **one signature per data segment**. This signature is the reason why the search is cheap: for a search you read only the member segment, build the matrix from it, check under which signatures the data sought **can** lie at all, and only those data segments are unpacked. Everything else stays compressed and encrypted where it is. The **data matrices** hold the actual net data of the records. And then the **main segment**, which holds more than you think. It consists of three matrices plus the encrypted session key sets: the **directory matrix**, the **literal matrix**, which holds the literals, and the **format matrix**, which holds the format descriptions of the data in the archive.
Why that is clever: because the literals are collected **centrally** in one matrix and the format descriptions **centrally** in another one -- and because these matrices are in turn compressed and encrypted. That makes the entire metadata secure **and** causes extremely little overhead. You do not pay for the structure with space. And then the trick that you should explain on the whiteboard, because it makes immediate sense:
the user decides per column whether the values of that column go into the literal matrix as literals. Take a column that only ever carries the same five keywords -- a currency code, `EUR`, `USD`, `CHF` and so on. Then you map that whole column via the literal matrix: the data segment then holds only a reference. With millions of records that is the difference between three characters per line and three bits. Exactly these tricks together are what make FLAM so effective in the end -- not a better compressor. And the access chain: no member segments without the directory segment, no data segments without the member segment. That is at the same time the security model.
Two names that turn up in the log and in error messages and that one should be able to place: the directory matrix is called **`FL5DIR`** internally, the literal matrix **`FL5LIT`**. And when a text file goes into the archive, `read.text()` internally creates a matrix named **`TXTRST`** -- it holds the remainders that arise when the lines are split: delimiters and trailing whitespace. That is why a text looks **byte for byte** the way it did before after the round trip, although FLAM computed in records. Whoever wants to know why an archive holds a row list with `TXTRST`: exactly for that.

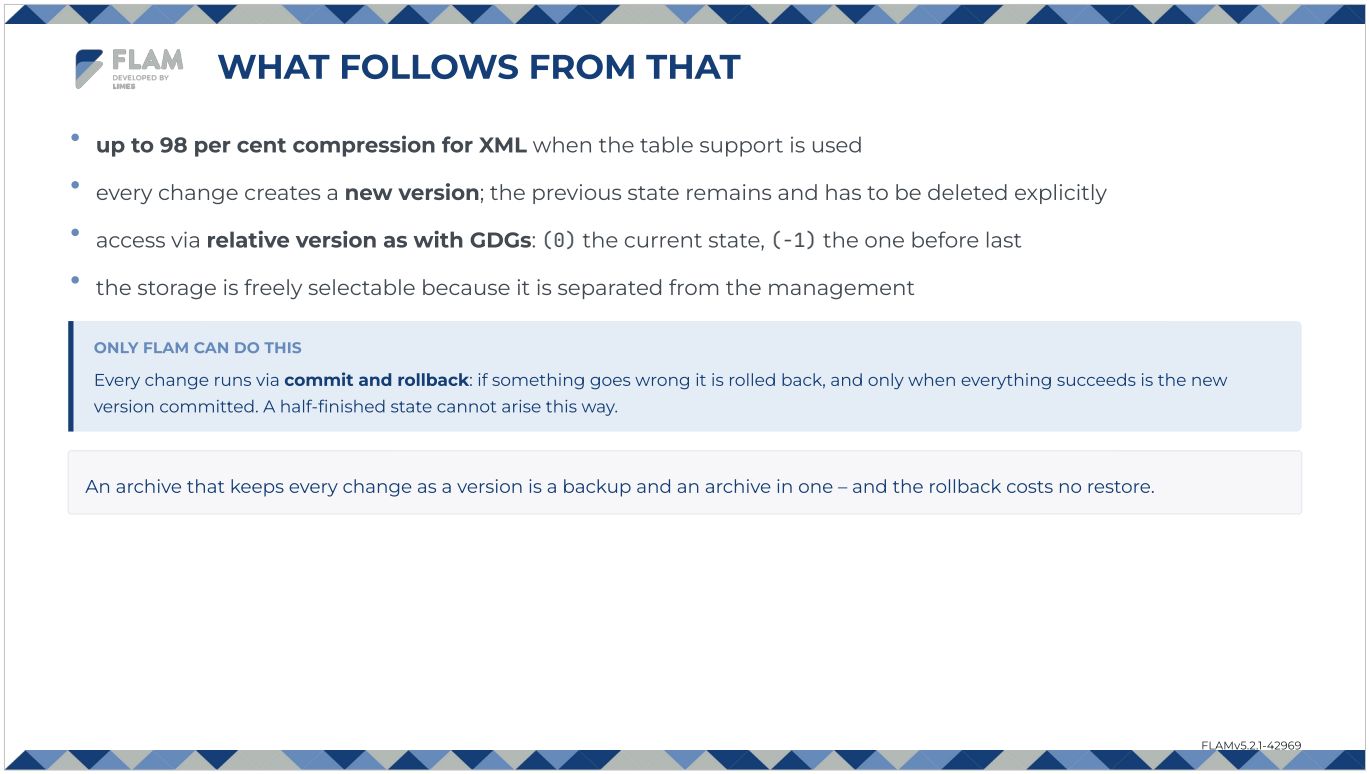
THREE KEYS, THREE VIEWS

KEY	WHAT IT OPENS	WHO TYPICALLY HAS IT
Directory	the table of contents: which members exist, with status	operator, archive management
Index	the search: which segment contains a hit	search service, intermediary
Data	the payload data in clear	only the authorised party at the end of the chain

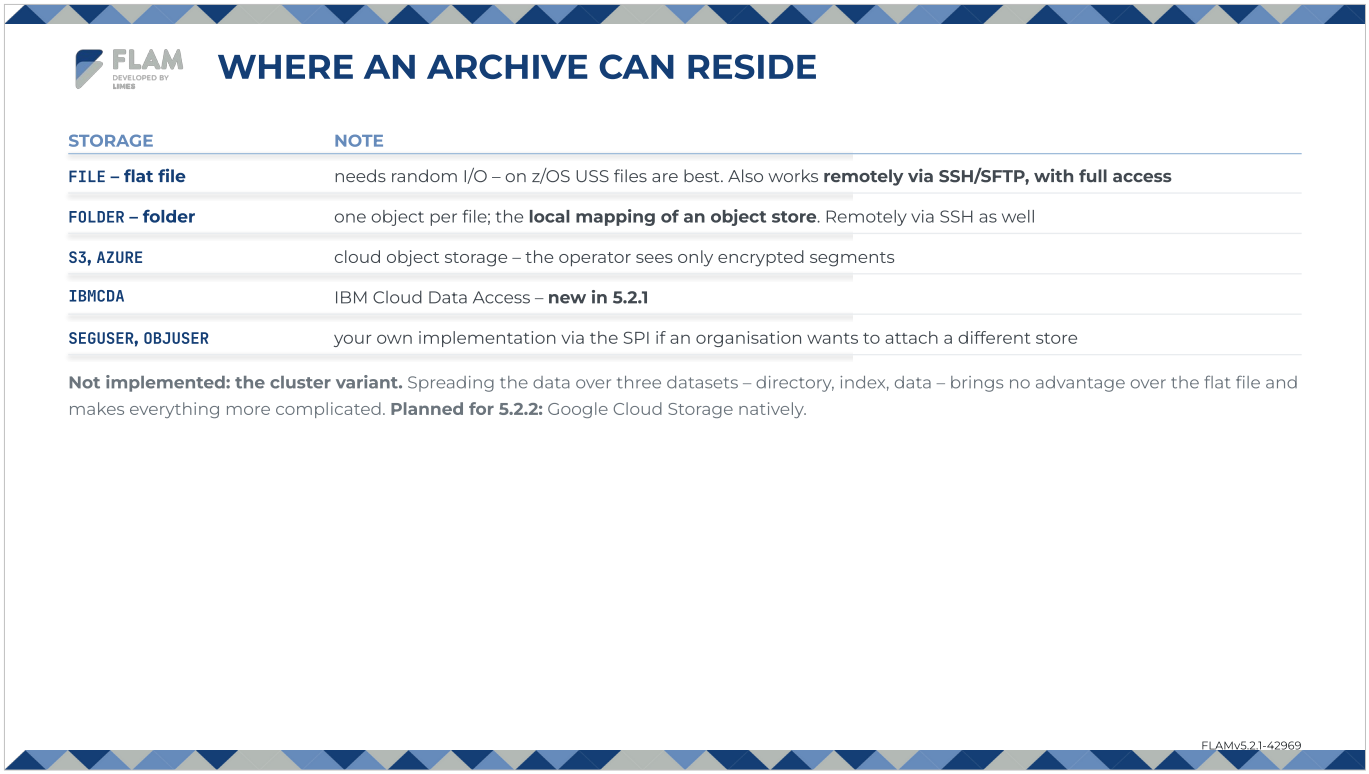
An operator can manage an archive and have it searched without being able to see the data. That is the basis for central archives, hubs and cloud.

FLAMv5.21-42969

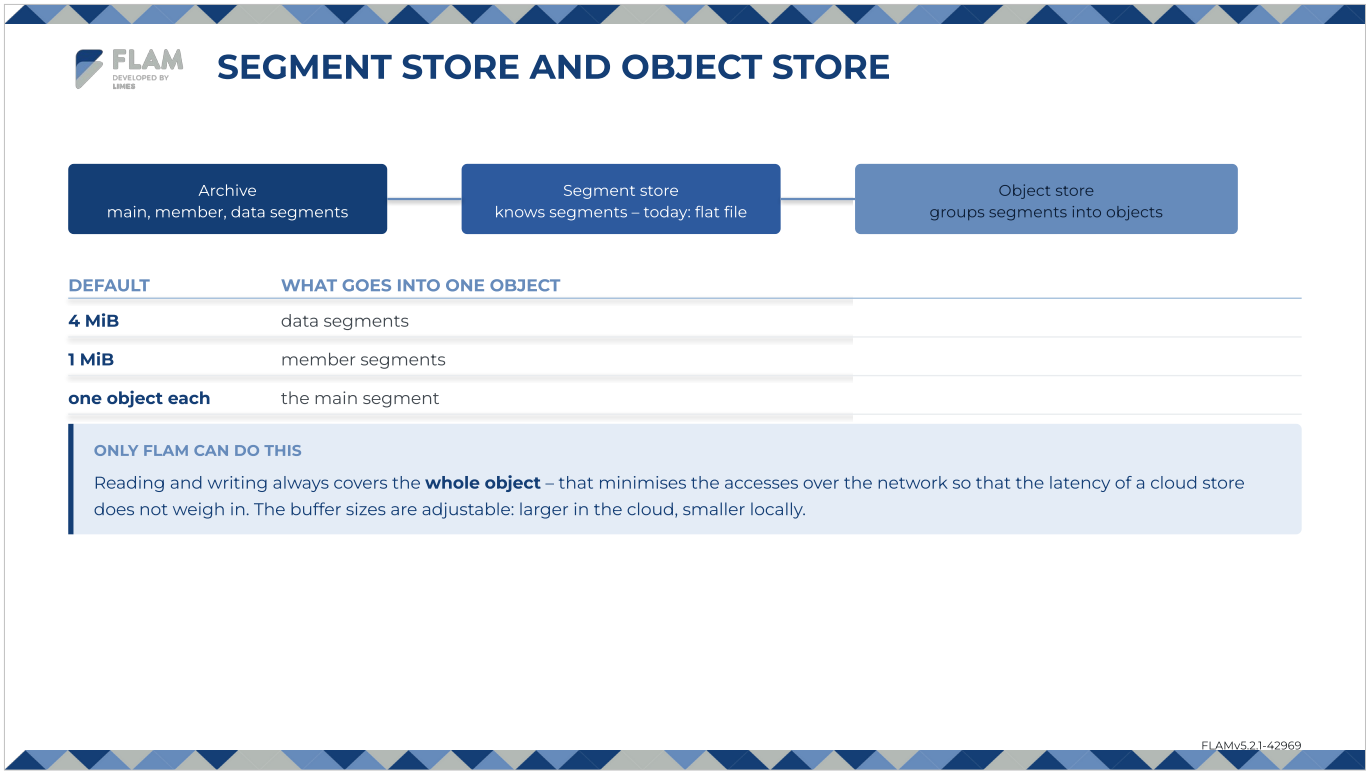
This is the slide that decides cloud and outsourcing conversations.
Anyone who only has the directory key sees the table of contents and nothing else. Anyone who has the index key can search, but not read. Only the data key gives clear text.
Practical consequence: a cloud operator can manage an archive without being able to see the data -- and without being able to derive anything from the search.
The connection with the matrices of the previous slide, because it is what makes the table understandable in the first place: the directory key opens the directory matrix, that is, the list of members. The index key opens the member matrices -- with them you see which data segments a member consists of, and you can search via the segment signatures. Only the data key opens the data matrices with the net data.
So the key separation is not an added ingredient, it falls out of the design itself: every layer has its own matrix, and therefore it can have its own key.



Name the figure: with table support up to 98 percent compression on XML. That is the figure a business department understands.
Versioning is the second surprise: every change produces a new version, the previous state is retained and has to be deleted explicitly -- like an incremental backup, and addressable like a GDG.
And the point operations asks about as soon as changes to an encrypted archive come up: **there is commit and rollback.** If there is an error during the change, it is rolled back. Only if everything runs through is the new version committed. That means no half-finished state can arise -- no archive that is half old and half new and where nobody knows which version applies.



This is the consequence of separating management and storage: the store is exchangeable without anything changing in the archive.
Important for operations: with the flat file random I/O is needed, so on z/OS USS files are the best choice. Anyone who puts an archive into a sequential data set loses random access.
Two things that come as a surprise and that you should therefore say:
The flat file also works remotely. We can read, write and change an archive remotely via SSH or SFTP -- that is full access over the network. The same applies to the folder variant.
And the **folder is basically the local image of an object store.** Anyone who understands it thereby also understands how the cloud storage locations work -- and can develop and test locally before going into the cloud.
What we have **not** implemented, and why: the **cluster variant**. Distributing the data over three data sets -- directory, index, data -- brings no great advantages over the flat file and only makes everything more complicated. That is why for local storage there is only `FILE` or `FOLDER`. One could implement VSAM KSDS as a segment store, but who wants to build in such a z/OS dependency these days.
On the current state: besides S3 and Azure we have supported **IBM CDA since 5.2.1** (Cloud Data Access). **Google Cloud Storage natively is planned and comes with 5.2.2.** And anyone who has a different store can attach it via the SPI himself -- `SEGUSER` and `OBJUSER` are there for exactly that.
The cloud variants are the reason why the three keys matter.



This slide answers the question that comes after the storage slide: if an archive consists of many small segments -- does that not mean thousands of accesses in the cloud?
The answer is: no, and the reason is an intermediate layer. There are two terms, and they are not the same:
The **segment store** is the layer that knows segments. At present it is implemented only in the form of the flat file -- there every segment lies at its position in a file, and you jump to it with offsets.
The **object store** is the layer below it, for everything that cannot do offsets: cloud storage and the folder storage. And at its core it is a **buffering**. Several segments are combined into one object, and reading and writing is always the whole object.
The defaults: **4 MiB of data segments in one object, 1 MiB of member segments in one object** -- and the main segment becomes an object of its own. The buffer sizes are adjustable.
There is exactly one purpose: **to minimise the number of accesses over the network so that the latency does not matter.** A cloud object store is good on throughput and bad on latency.
Anyone who fetches every segment individually there pays the latency once per segment.
Anyone who fetches 4 MiB in one go pays it once per object.
And from that follows an operating rule that you should pass on: **in the cloud larger buffers pay off, locally they do not.** Locally the flat file with offsets is faster anyway, and a large buffer then only reads data that nobody needs.
The folder, by the way, is the local image of exactly this object store.
That is why you can develop and test a cloud layout locally -- the same division, just without the network.

WHAT OPERATIONS NEEDS TO KNOW

- an existing archive is not overwritten silently – either delete it beforehand or set `OVERWRITE`
- **deleted segments physically remain** and only disappear from the directory
- `PRUNE` **removes the segments that are no longer used – that is the cleanup run that operations has to plan for**
- that is the price of versioning: nothing is lost until it is explicitly removed

> An archive in which a lot is deleted grows – until `PRUNE` runs. Plan this run in, like a reorg.

This slide prevents three surprises in production operation.
First: an archive is not simply overwritten, it has to be deleted beforehand -- or you set OVERWRITE.
Second and more important: deleted segments disappear only from the directory, physically they stay where they are. So the archive grows even though something was deleted. PRUNE cleans up.
Third: that is not a defect, it is the price of the versioning.

THE SAME THING OVER THE THREE ROUTES

ROUTE	WHAT WORKS WITH IT	WHAT DOES NOT
Batch utility	`flcl ARCHIVE` – create, read, list, update, check, clean up	no record-wise access in the program
Subsystem, file system	read and write transparently	no search, no targeted update
API	full access: search, position, change, delete	the source code has to be touched

FLAMv5.21-42969

The recurring slide for this topic, here tailored to the question of what works with an archive on which route.
The distinction that often gets too little attention here: record-level access **from within the program** is the domain of the API. Searching and changing itself can also be done by `flcl archive` in batch, and via `FLAMSUB` full access even works transparently -- with FLAM4 under conditions.
Anyone who wants to weigh the routes against each other will find that in the unit *The three access routes*.

PITFALLS

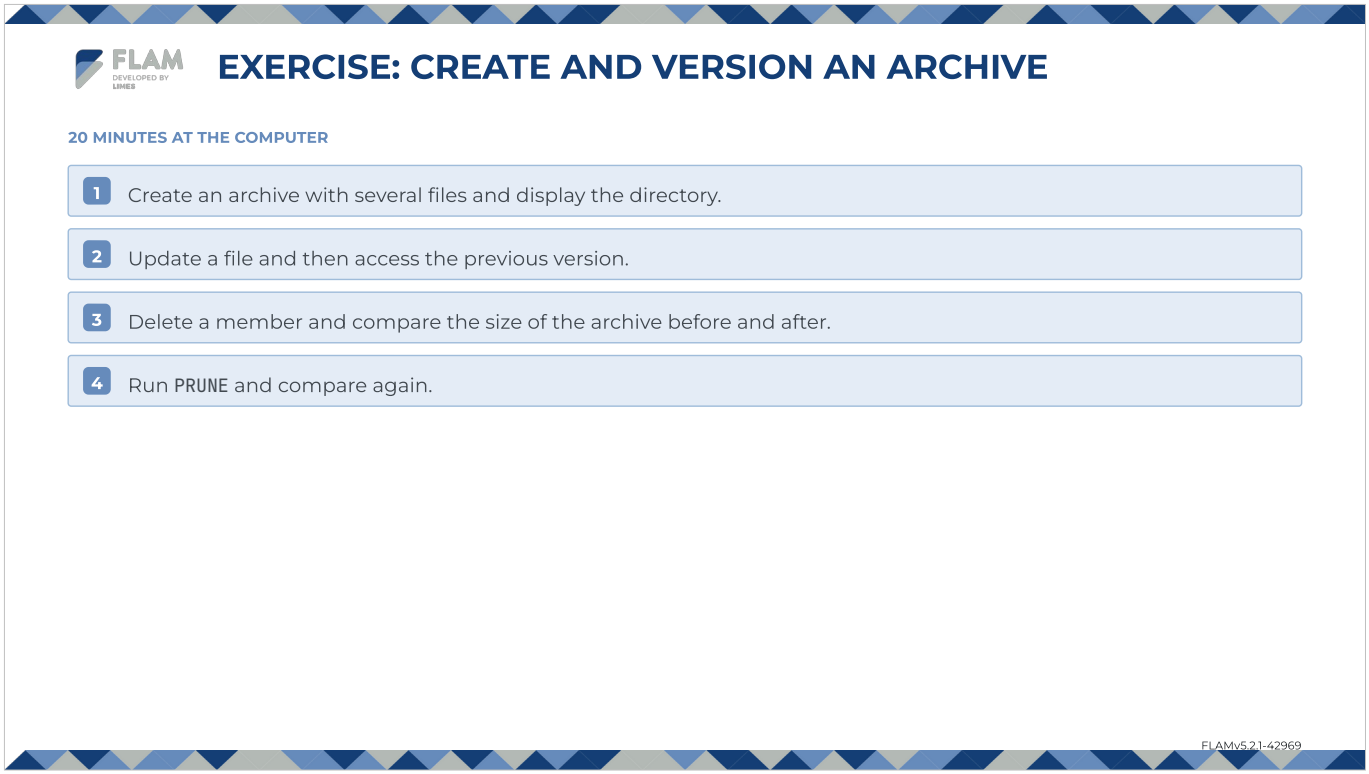
! **The archive grows despite deleting.**
Deleted segments physically remain. Plan `PRUNE` in.

! **An archive in a sequential file loses random access.**
The flat file needs `fseek`. On z/OS use USS files.

! **Concatenation as with FLAM4 no longer works here.**
FLAM5 is a managed archive with a directory, not a concatenation of members.

! **Without the matching key you see less, not nothing.**
Directory, index and data are encrypted separately – that is intended and has to be reflected in the assignment of rights.

All four from practice with FLAM5 archives.



Twenty minutes at the machine. The learning effect lies in step three: after the deletion the archive is not smaller. That is something you have to have seen for yourself once.
Solution:
1. `ARCHIVE.COMP` with `STORE.FILE`, then `ARCHIVE.LIST`.
2. Every change produces a new version. `ARCHIVE.LIST` shows them; the old versions remain readable -- that is intended and the reason why an archive grows.
3. The archive does not get smaller on deletion. Only `ARCHIVE.PRUNE` removes the segments that no longer belong to any version. With file-based archives the whole archive is rewritten in the process -- so plan it like a reorg, do not start it in the lunch break.
4. The expected insight: versioning costs space, and `PRUNE` belongs in the operations calendar. Anyone who does not plan for it is surprised after a year.



The collecting slide, and the one you show to a decision maker. Five points, and not one of them is available with an ordinary archiver.
First, the pot design. Every column gets a pot, and for every pot it is defined individually how it is compressed, encrypted and signed.
You can work globally -- one pot for everything -- or decide per column. And thereby minimise: a column that contains nothing worth protecting does not have to be encrypted, and one that is already compressed does not have to be compressed again. That is the handle with which you save CPU without giving up security.
Second, indexing, primary key and search with update, insert and delete. That is too big for this unit -- it has its own deck, and that is also where you find how Bloom filters and pot formation interact.
Third, the access rights. There are **limited Key Rights**: a key can carry restricted rights. And if someone has no access to a column, the **anonymisation methods** take effect -- he gets the row, but the column anonymised or masked, not an error.
Fourth, the separation of directory, member and data key, and what follows from it: **rekeying**, granting of rights and the restriction of rights. With that you set up **different views on the same data**.
Two examples that are understood immediately in the room: the statistics authority gets no personal data. And for the world outside the PCI DSS scope the credit card numbers are masked -- the same data, the same storage location, a different key.
Fifth, the crypto infrastructure. Today the new archive runs via **PGP or a passphrase**. But one could just as well speak **KMIP** or attach any other cryptographic infrastructure -- the interface is built for that.
If you take one sentence away from this unit: with FLAM5 security is not a property of the archive, but **a property of every single column**.

WHAT TO TAKE AWAY

- FLAM4 had three limits: the MAC over the whole thing broke, the data key sat in the comment field, and searching combined with changing needed three VSAM clusters
- FLAM5 separates archive management from segment storage – that is why the storage is freely selectable
- directory, member and data are separate matrices, themselves compressed and encrypted
- three keys, three views – managing, searching and reading can be separated
- every change is a version with **commit and rollback**; `PRUNE` is the cleanup run
- one **pot** per column: compression, encryption and signature are column decisions

FLAMv5.21-42969

Go through the learning objectives. The next unit shows how access to individual records in a large archive works in practice -- Bloom filters and result sets.

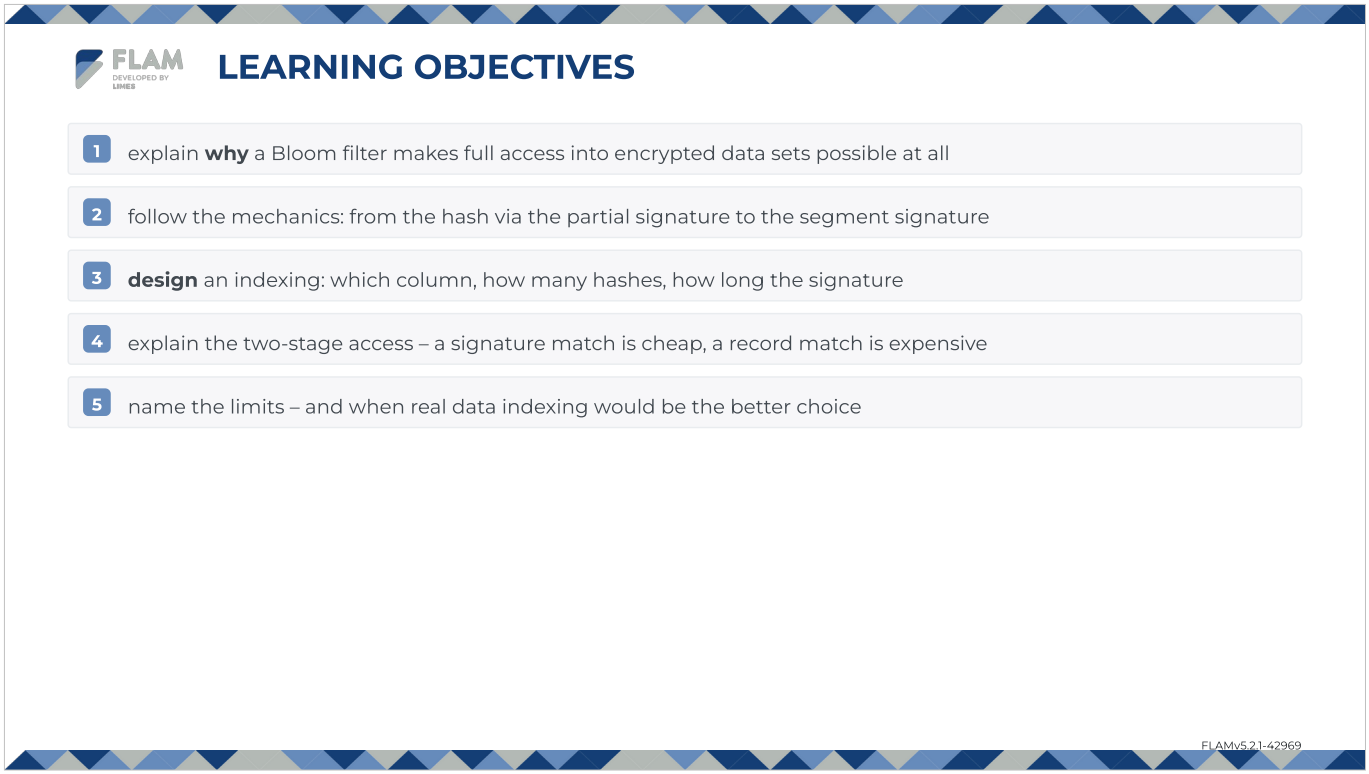
LEVEL 2 – FUNCTION TOPICS: ARCHIVE

Indexing, filters and search

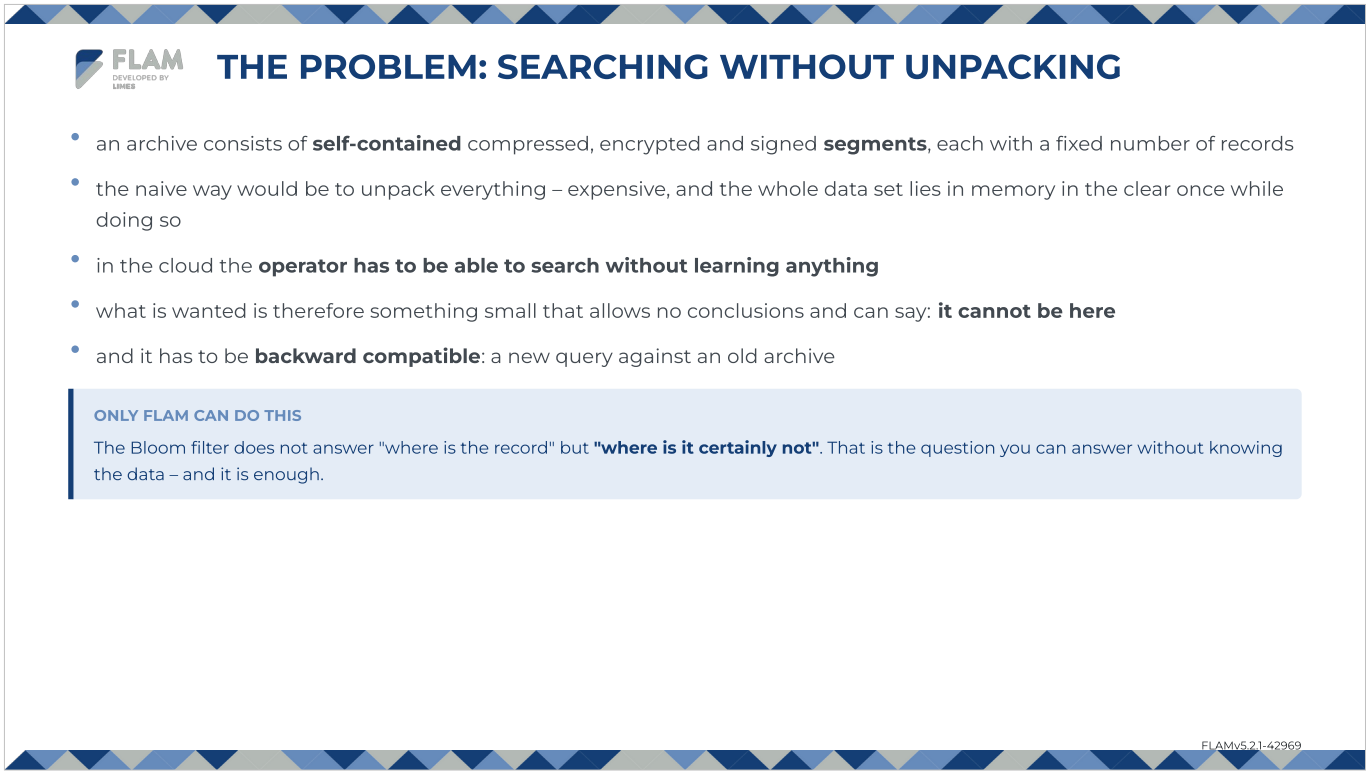
How FLAM5 searches in encrypted data sets – and why that costs little

FLAMv5.21-42969

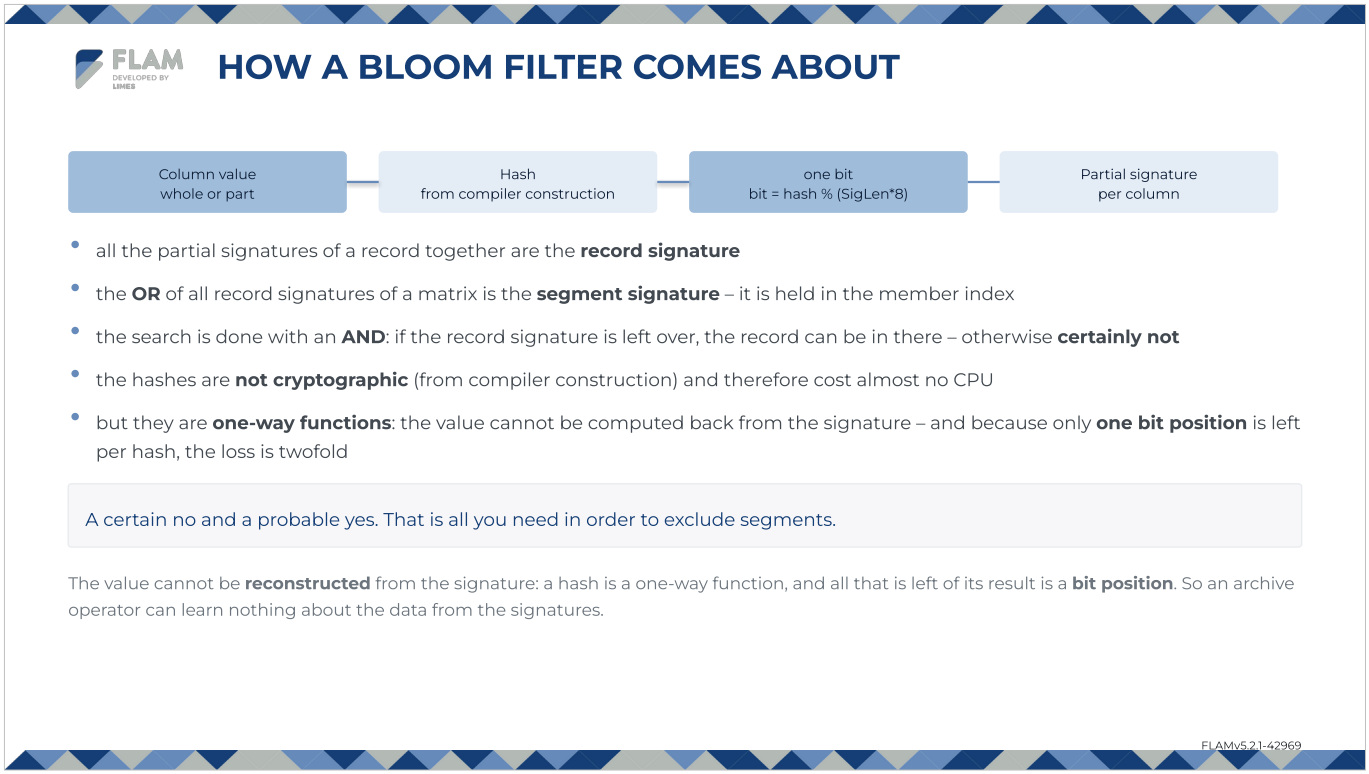
The unit that explains **how** FLAM5 can do what the archive unit promised: search, change, insert and delete in compressed, encrypted and signed data sets -- without rendering the file in the clear. The answer is the **Bloom filter**: an index that does not know the values and still says where they are not. Anyone who has understood this unit can design an archive in such a way that the search is cheap. That is design work, not configuration -- and it pays off over the lifetime of the data set. What *happens* to which column -- compression, encryption and signature -- is the unit *Pot design* of its own. Here it is about how to find a column again. The unit *The FLAM5 archive* is assumed: matrices, segments, the three keys.



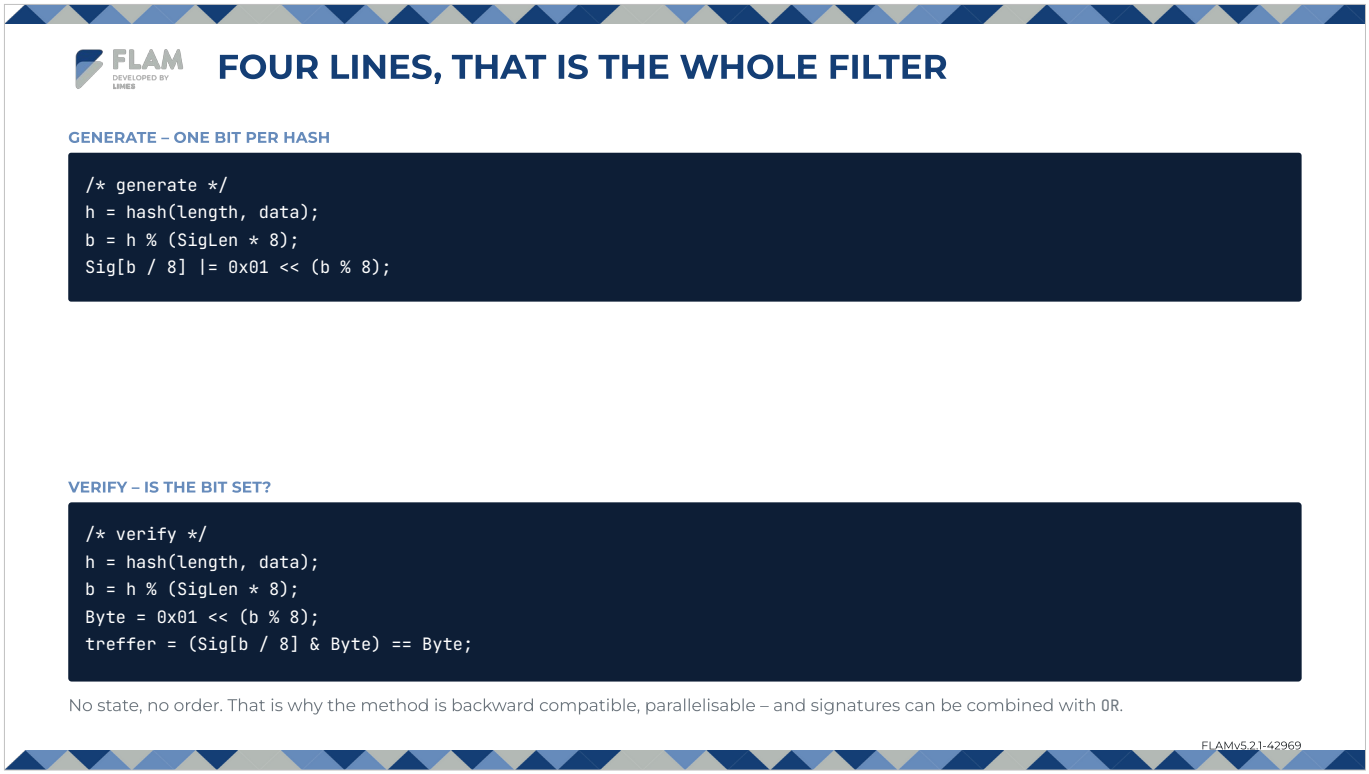
Read the learning objectives as a contract. By the end of the unit everyone should be able to make the design decisions -- which column is indexed and how long its signature has to be.
The fourth objective is the most important one and is readily underestimated: understanding why a hit in the filter is not yet a hit in the record.
Anyone who has not understood that takes the false positives for a defect.



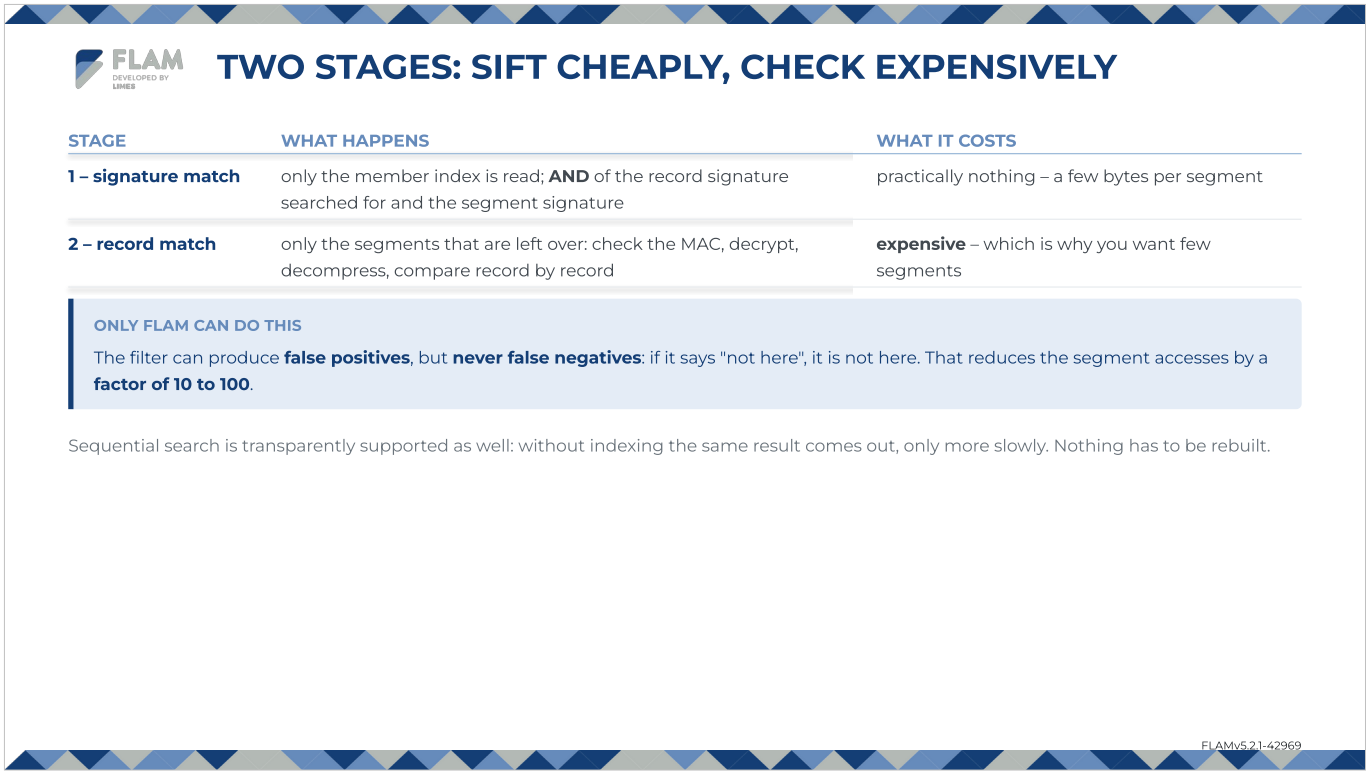
The problem first, then the solution -- otherwise the Bloom filter sounds like a gimmick. The task: an archive consists of self-contained compressed, encrypted and signed segments. "Self-contained" means each segment can be unpacked **on its own** -- each carries a fixed number of records. Someone is looking for a record. What do you do? The naive way is to unpack everything: decrypt, decompress, check the MAC, compare, pack it up again. With an archive holding millions of records that is not only expensive -- it is also **insecure**, because in the process the whole data set lay in the clear in memory once. And in the cloud it does not work at all: there the operator is to run the search without being handed anything that allows conclusions about the data. So you need something that has three properties at once: it has to be small, it must allow **no** conclusions about the data, and it has to be able to say "the record being sought cannot be in this segment". That is exactly what a Bloom filter is. And that last formulation is chosen deliberately -- more on that shortly. The second reason, which counts just as much in practice: **backward compatibility.** A new query has to be able to run against an old archive that is indexed differently. It does -- only with more computing effort. See the slide at the end for that.



The mechanics, and they are deliberately plain -- that is their strength. The basis is **one-way hash functions**, and explicitly **not cryptographic** ones. They are hashes from compiler construction. That matters, because it means they cost almost nothing -- a cryptographic hash function over every column of every record would be unaffordable. The flow from the bottom up, and it is worth writing it along on the board:
For an indexed column a hash is computed over its content -- or over a part of it, see the next slide. The hash value is computed modulo the signature length in bits. The result is a bit position, and this **one bit** is set in the partial signature of that column. You can use several hashes per column -- then several bits are set. All partial signatures over all indexed columns together form the **signature of a record**. The sum -- a simple OR -- of all record signatures of a self-contained matrix forms the **segment signature**.
And these segment signatures are stored as part of the **member index**. That is all. Four lines of code, as the next slide shows. And now the search, that is the nice part: if you look for records in which one or more columns have a particular value, you compute the record signature for those columns -- and combine it with the segment signature by **AND**. If the record signature comes out again, the record being sought is in this segment with high probability. If something else comes out, it **definitely cannot** be in it, and the segment is skipped.
That is the whole idea: a definite no and a probable yes.



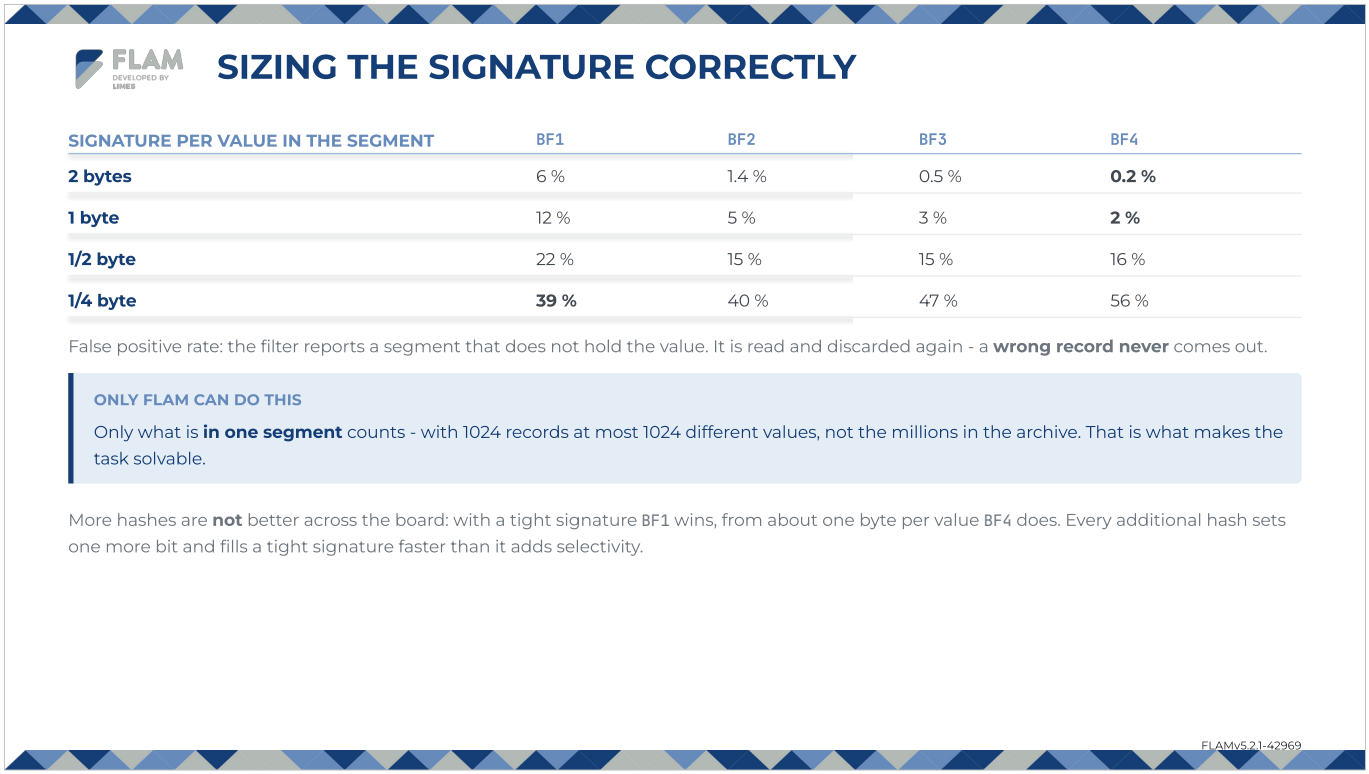
This slide is for the developers in the room, and it works: the Bloom filter is not a method with a secret, it is four lines. On the left the generation: compute the hash, modulo the signature length in bits, from that the byte and bit offset, and set the bit. It is initialised with zeros. On the right the check: the same path, only at the end it asks whether the bit is set. If not -- done, the record is not in this segment. The point to make here: **there is no state and no ordering.** That is why the method is backward compatible, why it can be parallelised, and why signatures can simply be combined with OR. Anyone who has seen this once also believes that the overhead is small.
And because it is always asked afterwards: the signatures themselves compress well. A bit mask with few bits set compresses excellently -- the overhead in the archive is therefore very small.



The slide that makes the difference between understanding and misunderstanding. Access runs in **two stages**, and they cost completely different amounts: **Stage one, the signature match.** You read only the member index, take the segment signatures and combine them with the record signature being sought. That is an AND operation on a few bytes -- practically free. The result is the list of the segments that **could contain** the record. **Stage two, the record match.** Only these segments are opened: check the MAC, decrypt, decompress, and then compare record by record. That is the expensive operation, and that is exactly why you want as few segments for it as possible. And now the sentence you have to say explicitly, because otherwise it is understood as a defect: **a hit in stage one is not yet a hit in the record.** A Bloom filter can have false positives -- two different values can set the same bits. But it cannot have **any false negatives**. When it says "not here", it is not here. That is not a weakness, that is the design. And it is the reason why stage two exists at all. The effect in figures, and it is worth naming them: **the Bloom filter reduces the segment accesses by a factor of 10 to 100.** With an expensive decoding per segment that is the difference between a search that runs and one that nobody pays for. And one more thing that makes integration easy: the method **transparently supports the sequential search as well.** Anyone who does not or cannot use indexing gets the same result -- only more slowly. So you do not have to rebuild anything to switch it on.



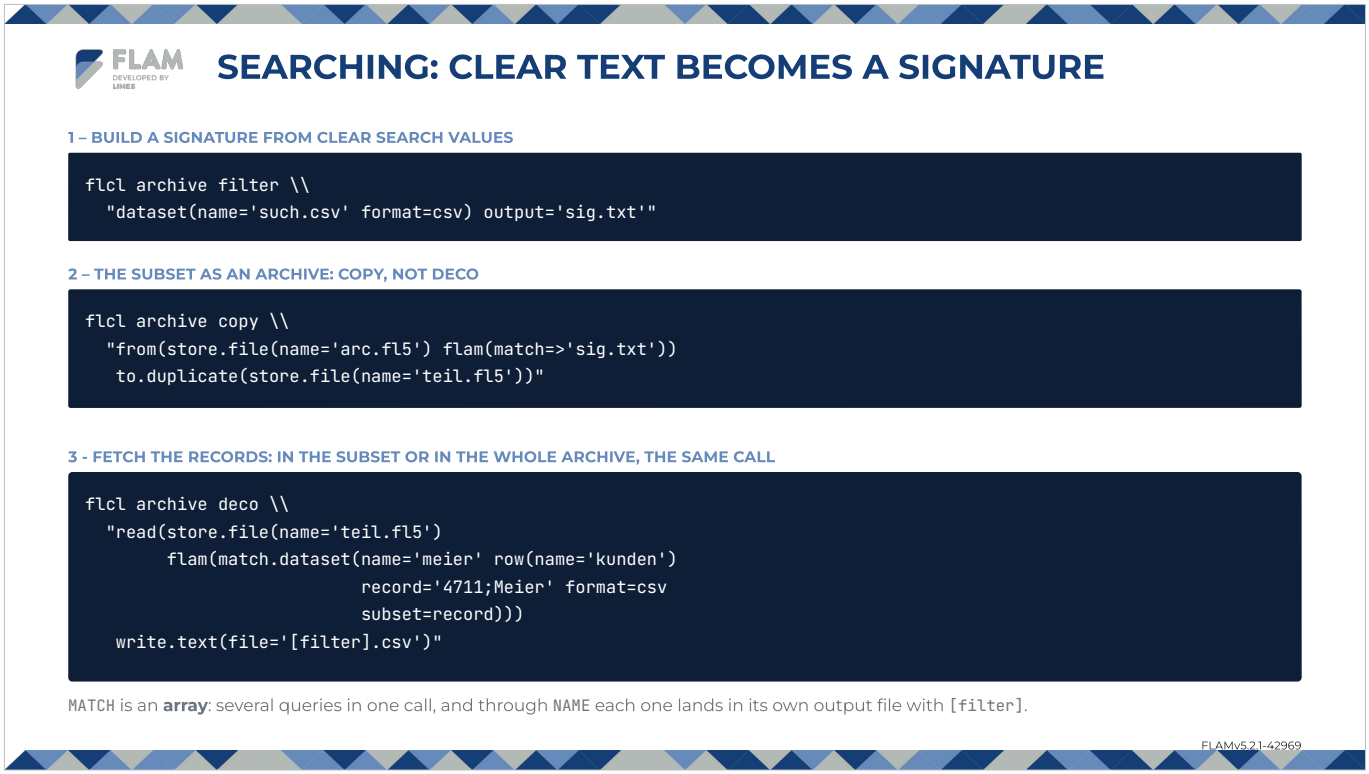
Now the practical part. Indexing is defined **per column** in the format description, in the `INDEX` object of the `COLUMN`.
First the good news: **`INDEX()` on its own is enough.** Since 5.2.1 FLAM sizes the signature itself when it is not given -- from the records per segment, the number of different values and the accepted false positive rate. So no Bloom filter arithmetic is needed to start. `METHOD` is the number of hashes: `BF1` to `BF4`. Given nothing, FLAM takes `BF4` and the length that fits it; given only one of the two, the other is chosen to match. More hashes means more bits set per value and therefore a better hit rate, but also a signature that fills up faster. `SIGLEN` is the length of the partial signature for this column, **in bytes**. There are ready-made values from `SL1` to `SL8192`, but a plain number is allowed as well -- rounded up to the next power of two, at least 4 bytes. `DISTINCT` is the one entry worth making in practice:
the upper bound of the different values of this column.
Without it FLAM assumes that **every** value can differ and sizes accordingly large. Measured with 3000 payments and 1024 records per segment: the IBAN column gets `BF4` with 1024 bytes, the currency column with `DISTINCT=3` only 4 bytes.
`FPRATE` is the accepted false positive rate in per mille, default 50, so 5 per cent. A member with many segments needs a smaller rate: with 100000 segments, 50 per mille read 5000 segments per searched value in vain. `DATOFS` and `DATLEN` narrow down **which part** of the column is hashed. That is partial indexing, and it has a slide of its own. `PRIKEY` declares the column to be part of the **primary key**. More on that shortly as well. The design sentence for the room: **the number of different values per segment determines how many hashes and how long the partial signature should be.** The "per segment" is what matters: a signature applies to one segment, and by default that holds 1024 records. Millions of different IBANs in the archive are therefore not the basis for the calculation -- at most 1024 different ones per segment are. A column with five possible values needs one byte. A column in which every record has its own value needs a thousand times that. Exactly how much is on the next slide -- and that is the slide to take away, because a signature that is too small is the most common design mistake.



The slide that prevents a design mistake which cannot be repaired afterwards -- a signature that is too small. The mechanics in one sentence: the signature has **8 times SIGLEN bit positions**, and every value sets as many bits as the method has hashes. If more values are written into it than there are bit positions, all bits fill up and the filter answers "maybe" to every search. It is not wrong then -- it is simply of no use any more, and the search costs as much as it would without an index. Hence the rule of thumb in the table: **one byte of signature per different value in the segment gives around 2 percent false positives, two bytes around 0.3 percent, half a byte already 16 percent.** That is the whole sizing exercise; the formula behind it is the standard Bloom filter formula and is documented in our manual at the `INDEX` object. Two things about it are surprising: **First, only what is in the segment counts.** Not the millions of IBANs in the archive, but the 1024 records of one segment -- so at most 1024 different values. That is what makes the task solvable in the first place. **Second, more hashes are not better across the board**, and that is the statement which contradicts the gut feeling. With a tight signature `BF1` beats `BF4`: half a byte per value gives 22 percent with `BF1` but 16 with `BF4` -- and at a quarter of a byte it tips over, there `BF1` with 39 percent beats `BF4` with 56. The reason: every additional hash sets one more bit, and when the signature is tight you fill it up faster than you gain selectivity. From about one byte per value upwards `BF4` is clearly the right choice. And the cost, so that the number is in the room: 1024 IBANs with `BF4` and `SL2048` are 2 KiB of signature per segment against some 34 KiB of payload in that column -- around 6 percent. With several indexed columns that adds up, because every column has its own partial signature. The sentence to take away: **the number to know is not the signature length but the number of different values per segment.** FLAM calculates the length from it; it is fixed when the first segment of the row is written and then stands in the segment header -- `LIST` and `INFO` show it, and it can be copied into the format description to freeze it. Every further member of the same row takes the indexing of the archive, even with another number of records per segment: a search works with one indexing per format description. And the limit that shows up in operation: 8192 bytes. If that is not enough for the assumed number of different values, **FLAM writes a warning** and names the three ways out -- index a part of the value (`DATOFS`/`DATLEN`), write fewer records per segment (`MAXREC`) or give `DISTINCT`. Measured:
without `MAXREC` FLAM determined 20343 records per segment, both columns ended at 8192 bytes and exactly that warning came; with `MAXREC=1024` and `DISTINCT=3` for the currency it was 1024 and 4 bytes without a warning.



The slide with the aha effect, and it is at the same time the finest example of the data protection aspect. With `DATOFS` and `DATLEN` you determine which part of a column is hashed over. And that allows something that a classic index cannot do: **search for a characteristic without indexing the value itself.** The example: an IBAN. The first two characters are the country code. If you set `DATOFS=0 DATLEN=2`, you index only this country code. After that you can find all payments to Germany -- without the IBAN itself appearing in the index in any form. And you can have both at once: index one column on the country code and lay a second indexing of the same data over the whole IBAN. Then you search via the one or the other depending on the question.
Now the limit, and it has to be named honestly because it hurts in the design: **only whole indexed parts can be used.** If you search for a fraction of an indexed column -- that is, for something that does not exactly match the indexed range -- then this column becomes a **wildcard** for you. The partial signature is then binary zero, and binary zero is always a hit. That is correct, but it costs the preselection.
That is exactly why at design time you have to know **what will be searched for later.** That is the most unpleasant property of the method, and it appears once more on the disadvantages slide. The same applies to upper and lower case: anyone who later wants to compare regardless of upper and lower case has to know that **beforehand** and set the indexing up accordingly.



How a search works in practice, in three steps -- and the first one is the interesting one. **`ARCHIVE FILTER`** turns a clear data record -- CSV for preference -- into a **signature that allows no conclusions about the data.** That is the command with which the query is prepared. You enter the values you are looking for in the clear and get a bit mask out. And now the point why this is a command of its own and not just a parameter: **the signature can be sent.** The query is built where the data may be known, and only the signature goes to the operator of the archive. He searches with it without knowing what for. For the query to match the indexing of the archive it needs its description. That is what `ROWLIST` is for: **the list of format descriptions of an archive can be produced with `LIST`** and then given as `ROWLIST`. So the indexing need not be known -- it can be fetched. State the indexing in the `FILTER` object instead and it is checked for plausibility. **Second step: `MATCH`.** When reading, the signature is passed with `MATCH.FILTER` -- most simply by reading in the parameter file the `FILTER` command produced: `match=>'sig.txt'`. Or the clear search record is given directly with `MATCH.DATASET`; then FLAM computes the signature itself. Important for the second call: **a `DECO` always delivers clear data.** So whoever wants a subset as an **archive** takes `COPY` with `TO.DUPLICATE` -- then only the matched segments are copied, and nobody has read the data. And `MATCH` is an **array**: any number of queries can be given, `FILTER` and `DATASET` mixed. Give them different names through `NAME` and the name can be used in the input-output name mapping with `[filter]` -- then one call writes the hits of each query into its own file. **Third step: `SUBSET`.** With it you say what you want back: `SEGMENT` delivers the complete segments, `RECORD` only the matching records. `AUTO` decides by command -- segments for `COPY`, records for `DECO`.
For a result set that is passed on, `SEGMENT` is often right:
the recipient then gets closed, integrity protected units. And the connection with the keys from the archive unit: for stage one only the index key is needed. Only stage two needs the data key. That is why an operator can search without being able to read. The middle call is the case that makes the method comprehensible in the first place: **a `COPY` with the filter that copies the matched segments into a new archive.** The result is a smaller archive -- closed, integrity protected segments, without anybody having read the data. And now the point the participants should take away: the third step is **the same call**, whether it goes against the filtered subset or against the whole archive. The search knows no difference.
So the subset is not another route, only a smaller body of data -- the preselection can be made once, passed on, and the recipient works with it exactly as with the original.



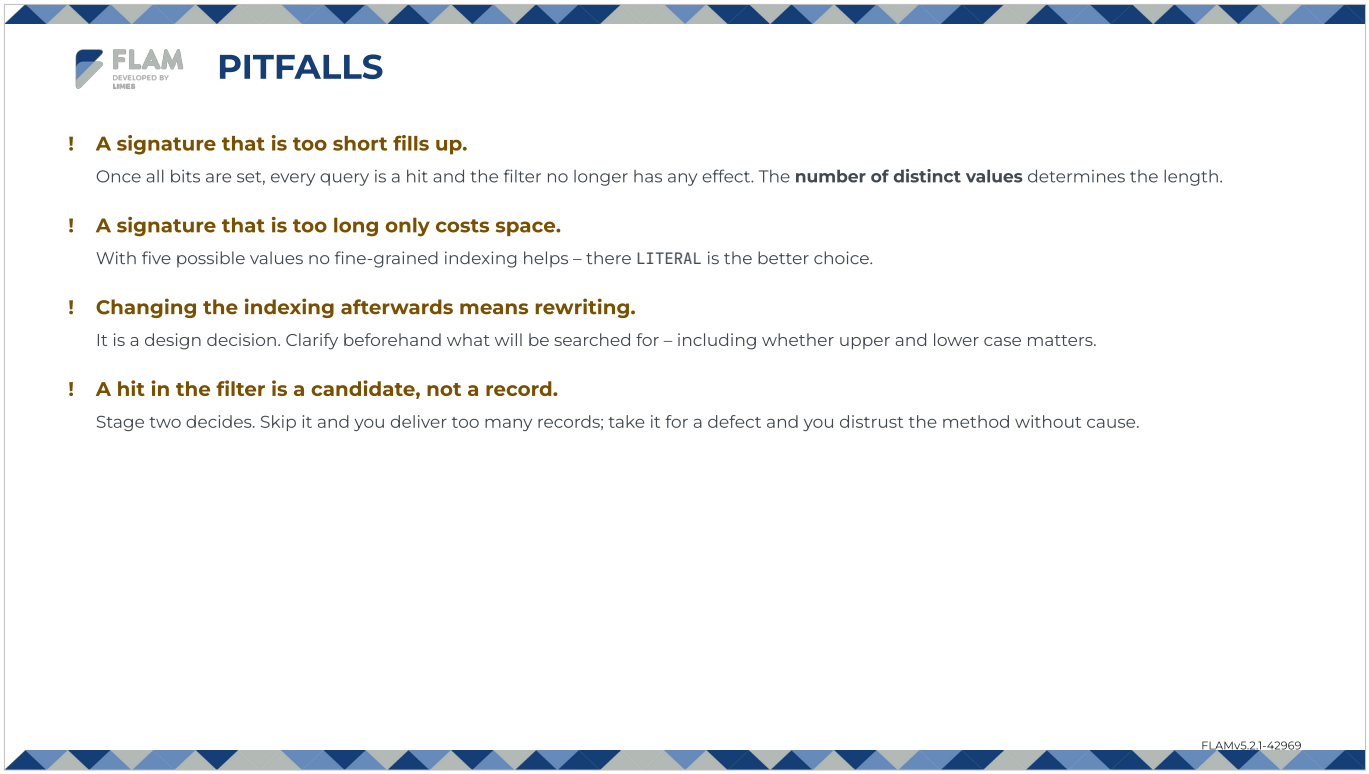
Three things that sit on top of the indexing and are decided along with it in the design. **The primary key.** With `PRIKEY` an indexed column is declared part of the primary key. Several columns can be that together. The archive then has a business key through which `UPDATE` and `DELETE` work precisely -- not through a position but through the content. **Uniqueness.** `LOOKUP` builds the hash table and makes sure in doing so that a record is unique. Anyone who has a business key and wants to rule out duplicates sets it. And the placing that should be said with it:
`LOOKUP` is mostly needed by ourselves -- for the directory matrix. For a customer it only pays off when he searches **repeatedly in the same matrix**, and in practice that means: the content of a whole member lies in one matrix. There are applications for that, but they are rare. A user notices nothing of the matrix boundaries otherwise, computing the table costs, and with the next matrix it is discarded. The uniqueness check is the reason to set it anyway -- not the search speed. **The rights.** And here comes the arc to the anonymisation from the archive unit: with `NOTMTD` you lay down **what happens when somebody has no access to a column.** Four methods: `ZERO` delivers zeros -- the default. `MASK` masks. `RAND` delivers random values. And `IBAN` produces a formally valid IBAN which is not the real one. The last one is the interesting case and it is worth explaining: a test environment or a statistical evaluation needs data that **survives the format check**. Zeros do not -- an application that checks an IBAN aborts. A generated, formally valid IBAN passes. This sets up different views of the same data without copying it: the statistics office gets no personal reference, the test environment gets plausible values, and the authorised application gets the original. One body of data, three views, decided through the key.
Two switches are in the table for completeness, because they belong to the column decision: `LITERAL` moves the values into the **global literal matrix** instead of putting them into the assigned pot. That pays off for columns with few keywords. But take care with security: the literal matrix is protected with the **directory key**. Anyone who wants to distinguish directory access from data access in security terms must **not** set the switch -- not even when the values are literals. `NOCASE` compares without regard to upper and lower case. That too is a decision when the archive is **created**, not when it is searched.



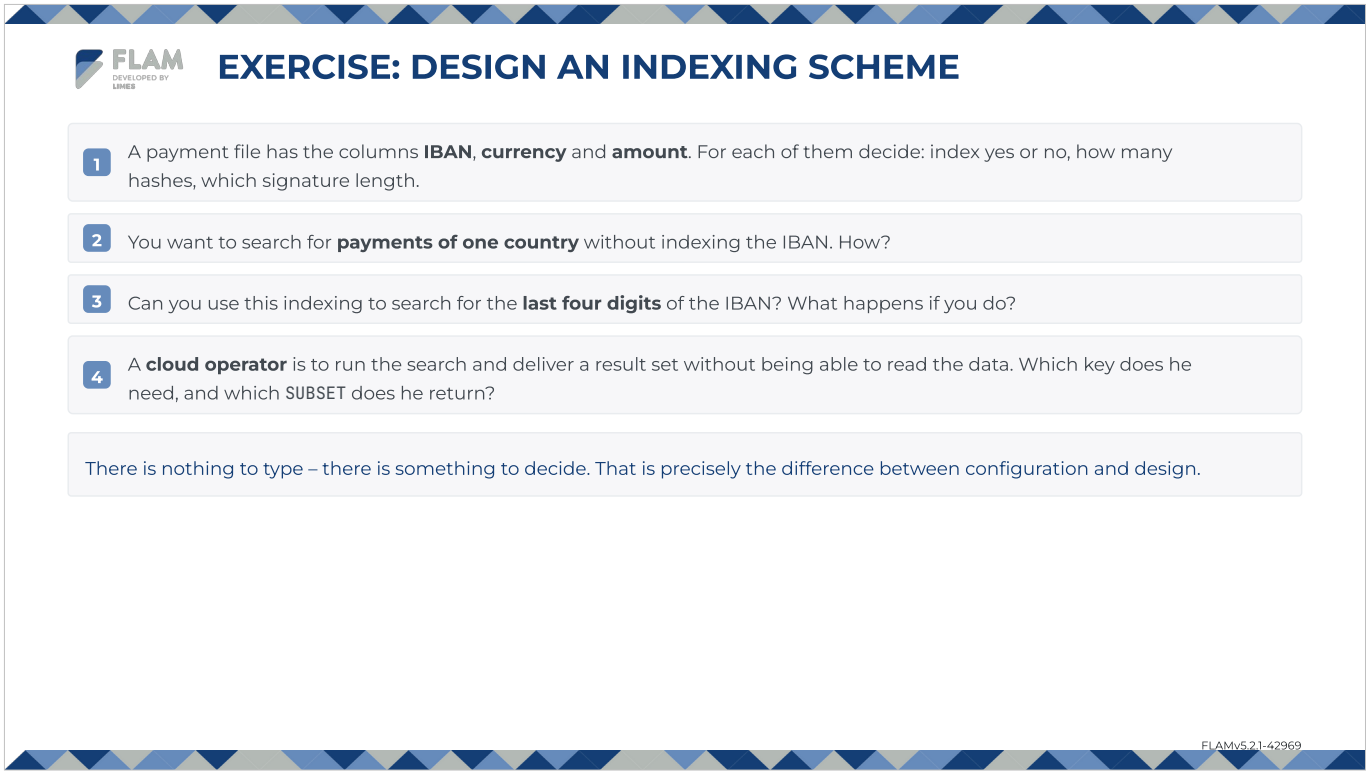
The slide that can cut hours off a load job -- and it belongs here because inserting, too, goes through the **primary key**. The question is always the same: **do I know that these records do not yet exist in the member? If yes, load sequentially.** `RECMODE=SEQUENTIAL` is the default, and the help calls it what it is: "no verification of keys (simple and fast load)". It appends, nothing is checked. That is the normal case for a daily feed whose source is disjoint anyway -- the day's postings, which did not exist before. **If no, you have to insert.** `RECMODE=INSERT` inserts only new records and reports an error when the record is already there.
`KEYED` is what other systems call "put": insert if new, update if present. Beside them there are `UPDATE` and `DELETE`, and for each the variants with a warning (`INSWRN`) or with ignoring (`INSIGN`) instead of an error -- the error handling sits in the keyword itself. And now the price, and it is the reason for this slide: **an insert is expensive.** For every record to be inserted, all segments whose signature matches have to be **unpacked** and every record in them checked for whether the primary key is already there. Which means: **the bigger the member, the more expensive a mass insert.** The Bloom filter helps, but it only reduces the number of candidates -- it does not save the unpacking. The consequence for the design:
anyone loading millions of records a day should make sure the source is **guaranteed new** and load sequentially. A mass insert "to be on the safe side" is the most frequent reason for a load job that gets slower month by month.



The honest balance sheet, and it belongs here. A method whose limits you name is believed. The advantages one by one: the signatures compress well, the overhead is therefore very small. They allow **no conclusions about the data** -- anyone who only has the signature learns nothing. If the indexing does not fit the question, the column becomes a wildcard and the search works anyway. The segment accesses drop by a factor of 10 to 100. The sequential search is transparently supported as well. And backward compatibility: **the same query against differently indexed archives always leads to the same result** -- only the computing effort differs, and it is the lower the more indexes can be used. That is the property that makes long-term archiving possible in the first place. The disadvantages, and they are all of the same type -- you have to think beforehand:
The signatures themselves cannot be indexed, they are compared sequentially. The indexing has to be defined **up front**. Upper and lower case has to be decided beforehand. Anyone who later searches only by columns that are not indexed has to unpack everything. And only whole indexed parts can be used. And then the alternative you should know in order to understand the choice: **real data indexing.** Its advantage is free-text search -- you can search for anything, including for what you did not think of when creating the archive. But its disadvantages are exactly the ones for which we did not take it: the data volume of the indexed columns **doubles**, the index holds the **clear data** -- insecure, that is, and with a subset the sensitive net data is visible -- and the memory requirement for holding the index is much larger. The sentence that weighs it up: **anyone who needs free-text search needs a search engine. Anyone who wants to search a protected data set needs Bloom filters.** And the limit that really bites in operations stands at the foot of the slide: **the size of a member is limited by the index.** The member matrix with the segment signatures has to fit into memory beside the directory. A customer of ours works with 4 KiB signatures, keeps 80 million records in one member and allows only 128 records per matrix -- with that the index no longer fits into the 31-bit address space on z/OS, and that is the reason why we are bringing the 64-bit support on z/OS. The case can be used to show what designing means: 128 records with a 4 KiB signature are far too few. Either the signature length is reduced or -- better -- the number of records per matrix is raised to 1024. With really large members it can still happen that the index does not go into memory.



All of them from the design, not from operations -- this unit makes mistakes that are no longer cheap to correct later. On the first: a partial signature that is too short for a column with many different values fills up. Once all bits are set, every query is a hit and the filter no longer has any effect -- it then only costs. That is why the number of distinct values is the quantity you align the signature length to. On the second: conversely, a very long signature is of no help for a column with five values -- it only costs space. For a currency code `LITERAL` is the better choice than a fine-grained indexing. On the third: the indexing is a design decision. Changing it afterwards means rewriting the archive. That is why it is worth talking to the business people beforehand about what will be searched for. On the fourth: a hit is a candidate. Anyone who takes the false positives for defects distrusts the method without cause -- and anyone who ignores them builds an application that delivers too many records.



Twenty minutes, and the task is deliberately a design task -- there is nothing to type, there is something to decide.
On the solution, once the group is ready: **Task 1.** The IBAN has millions of different values -- it needs a long partial signature and more than one hash, so about `SL64` and `BF2`. The currency code has three to five values: `SL8` and `BF1` suffice, or better still `LITERAL` right away without indexing. The amount is rarely searched for -- no index. **Task 2.** The country code: `DATOFS=0 DATLEN=2` on an indexed column of its own. With that you find all payments of one country without indexing the IBAN. **Task 3.** No -- and that is the crux. Anyone who searches for the last four digits of the IBAN does not hit the indexed range. The column becomes a wildcard, the search works anyway, but without preselection:
all segments have to be unpacked. So the answer is: it works, but it costs the whole advantage. Anyone who needs that indexes the last four digits in addition. **Task 4.** The operator needs the **index key**, not the data key. He can then run stage one and give back the candidates -- the result set as segments with `SUBSET=SEGMENT`. He cannot read them.
That is exactly the cloud scenario from the archive unit.

WHAT TO TAKE AWAY

- the Bloom filter says **definitely no** and **probably yes** – that is why it suffices without knowing the data
- two stages: the signature match is practically free, the record match is expensive – **factor 10 to 100** fewer segments
- `INDEX()` is enough -- FLAM sizes the signature itself; the entry that counts is `DISTINCT`
- per column the design goes on: `DATOFS`/`DATLEN` for the part of the value, `PRIKEY` for the key, `FPRATE` for the rate
- **Partial indexing** allows a search for a characteristic without indexing the value – the country code without the IBAN
- `LITERAL` for columns with few values, `LOOKUP` for the uniqueness, `NOTMTD` for the view without an access right
- the same query delivers the same result against every archive – only the effort differs

FLAMv5.21-42969

Go through the learning objectives. The sentence that should stick is the first one: a definite no and a probable yes.
Next comes the unit *Managing archived data*, in which exactly this mechanism is used in `ARCHIVE` commands -- search, update, insert, delete and the check depths of `VERIFY`.



Second functional topic, and the unit that turns an archive into a body of data: searching, inserting, changing, deleting -- inside the compressed and encrypted holdings. The core is the two-stage access: a cheap preselection over signatures, and only then the expensive decoding. Whoever has understood that understands why this is affordable at all.



Objective two is the actual learning objective. Objective four is what counts in cloud conversations.

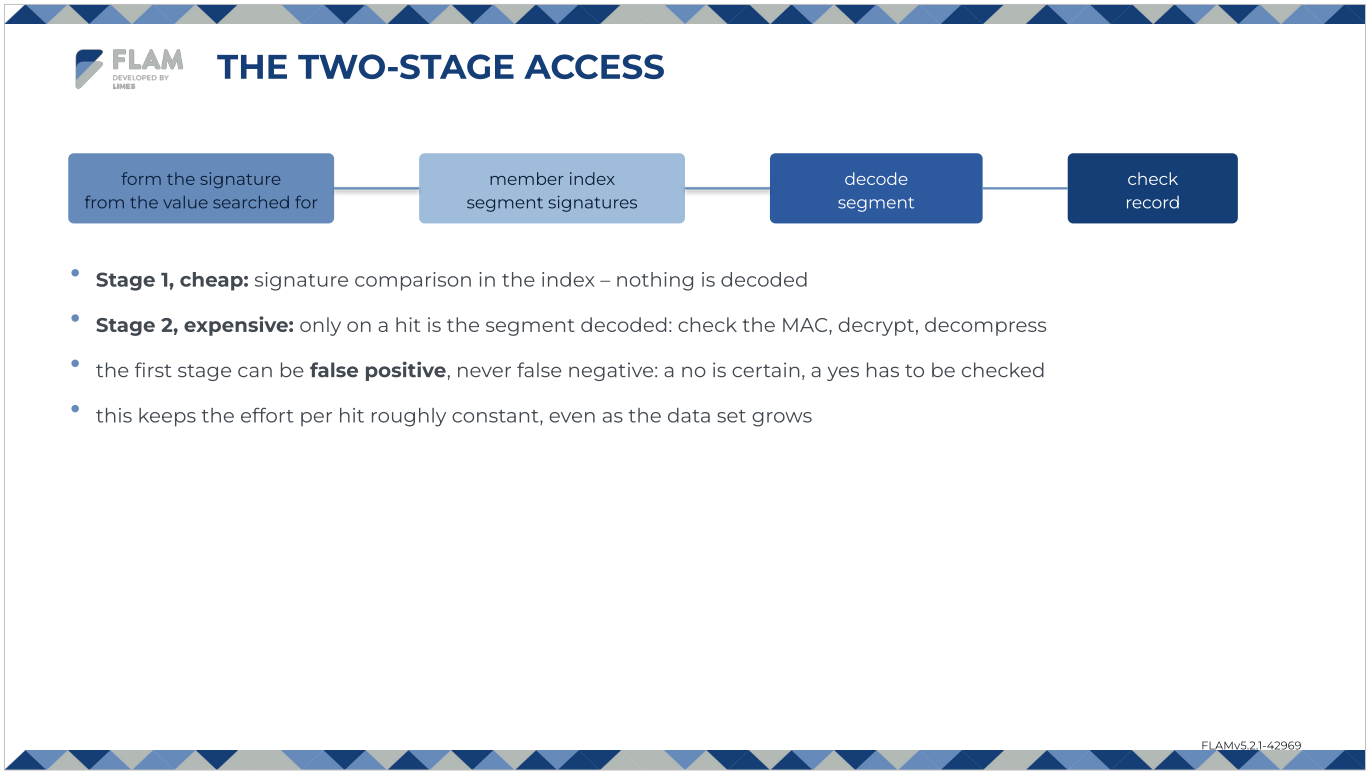
WHAT IS POSSIBLE ON AN ARCHIVE

OPERATION	GRANULARITY	NOTE
Searching	columns of a record	returns a result set, not a file
Read	individual record	by key or by position
Changing	columns of a record	the record is replaced in place
Insert, delete	whole record	delete removes it from the directory, not physically

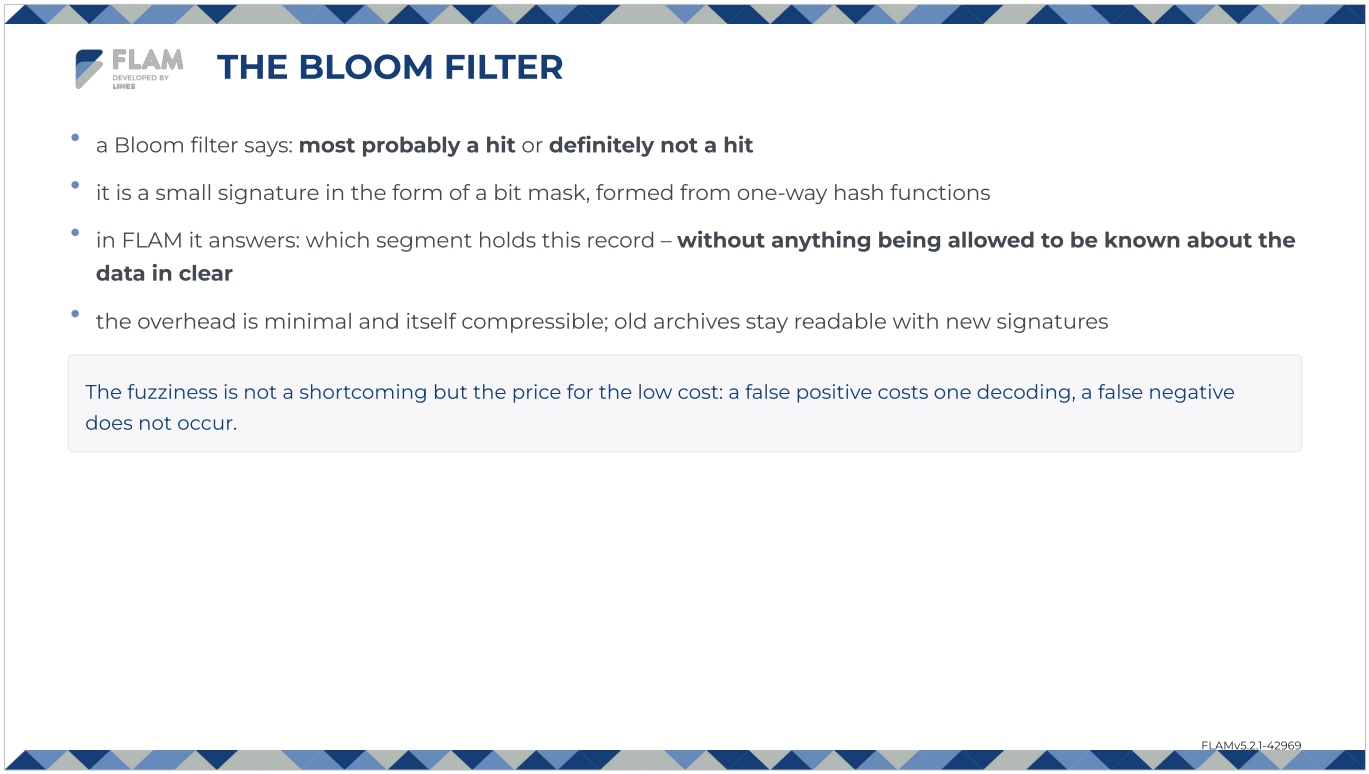
All of that **without decoding the entire file into clear and packing it up again** – and the data sets stay addressable through the access methods your application already uses (on the mainframe VSAM RRDS, KSDS).

FLAMv5.21-42969

The four operations with their respective granularity. The difference between record and column matters: insert and delete hit a record, change and search hit columns.
The sentence that should stick: all of that without rendering the entire file in the clear and packing it up again. That is the difference to any archiver.
And the migration path for existing applications: the data sets remain addressable via the access methods that the application already uses -- on the mainframe VSAM RRDS and KSDS.
If somebody asks about the cloud, here is the clean distinction, because it is often blurred. What works today is the segment store in S3, Azure and IBM CDA -- the storage location is in the cloud, the access comes from here. What does not work yet is the reverse route: a cloud application that accesses an opened member via a standard API. **ODATA** is intended for that, and that is an outlook, not a commitment -- phase 3 of CloudFLAM.
The connection is worth the sentence, because it shows why the record interface is strategic: ODATA is in the cloud what VSAM was on the mainframe. Once full access to an opened FLAM member is in place, all that remains to be built for ODATA are the REST stubs against our record interface. Then this access method can be replaced transparently by us in the cloud.



This slide is the core of the unit. Explain it from left to right.
First stage: a signature is formed from the value sought and checked against the segment signatures in the member index. That is cheap, because nothing is decoded.
Second stage: only if the signature matches is the segment decoded -- check the MAC, decrypt, decompress -- and the record searched for inside it. That is expensive, but happens rarely.
Say the fuzziness clearly: the first stage can be false positive, never false negative. A no is certain, a yes has to be verified.



First in general, then for FLAM. A Bloom filter is an abstract selection mechanism: most probably a hit or definitely not a hit.
It is in many operating systems, databases and file systems, in network routing and in spell checking.
For FLAM it is the answer to the question: in which self-contained compressed, encrypted and signed segment does which record lie -- without anything being allowed to be known about the clear data.
The overhead is minimal and itself compressible.

RESULT SETS WITHOUT CLEAR TEXT

- selecting the segments needs only the **index key**, not the data key
- a result set can therefore be assembled **without reading a single record**
- the result set stays encrypted until someone with the data key opens it
- an operator is not handed anything that allows conclusions about the data

That is the technical basis for central archives, hubs and cloud – and for the automatic anonymisation of result sets.

FLAMv5.21-42969

The slide that decides outsourcing and cloud conversations, and it follows directly from the three keys of the archive.
The searcher needs the index key in order to select segments. He does not need the data key -- so he can assemble a result set without reading a single record.
Wording from the old slides: this makes it possible to realise a search and the assembly of subsets over the compressed, encrypted and signed data without the cloud operator getting anything into his hands that allows conclusions to be drawn about the data.

VERSIONS IN OPERATIONS

- every change creates a new version – like an incremental backup, without an additional run
- access via relative version as with GDGs: `(0)` current, `(-1)` the one before last
- the previous state is retained and has to be deleted **explicitly**
- **deleting does not make the archive smaller** – only `PRUNE` does that

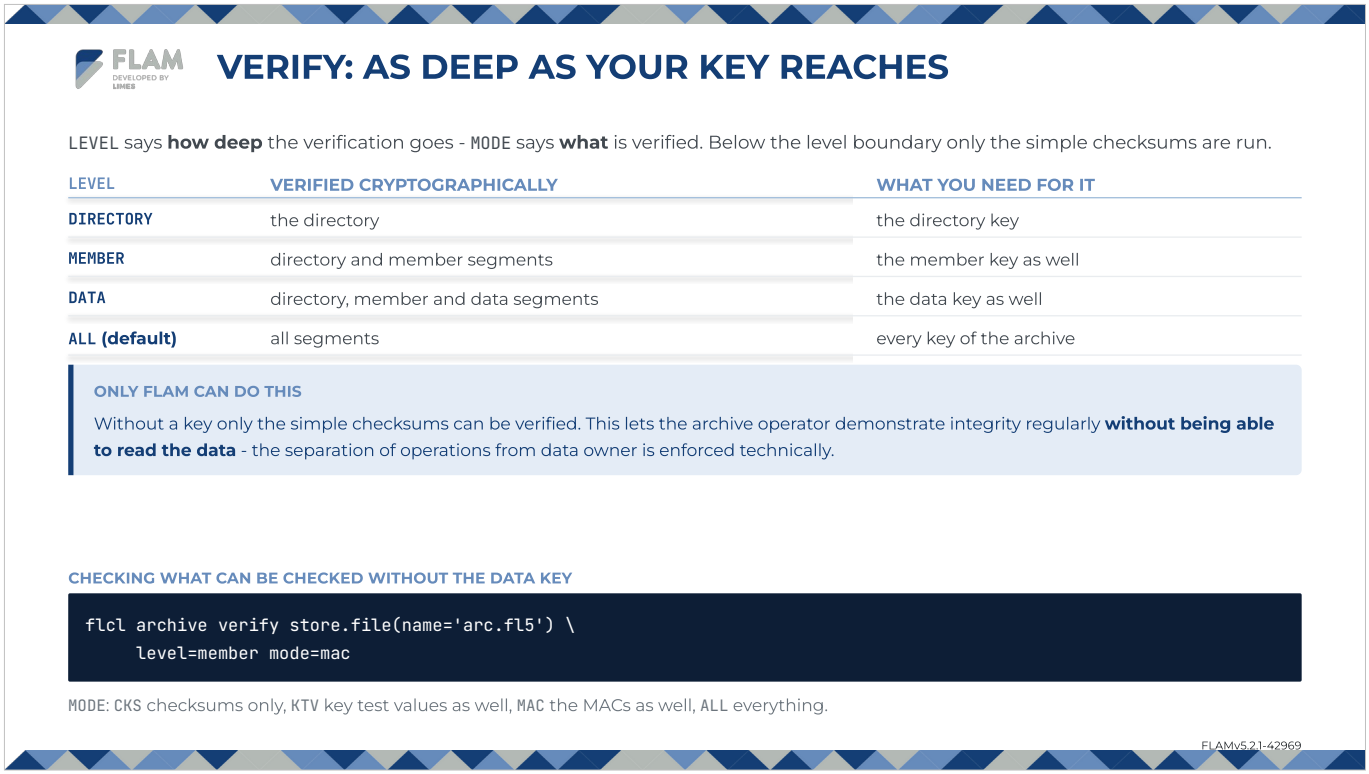
> For capacity planning: an archive with many changes grows steadily. `PRUNE` belongs in the operations calendar, like a reorg.

FLAMv5.21-42969

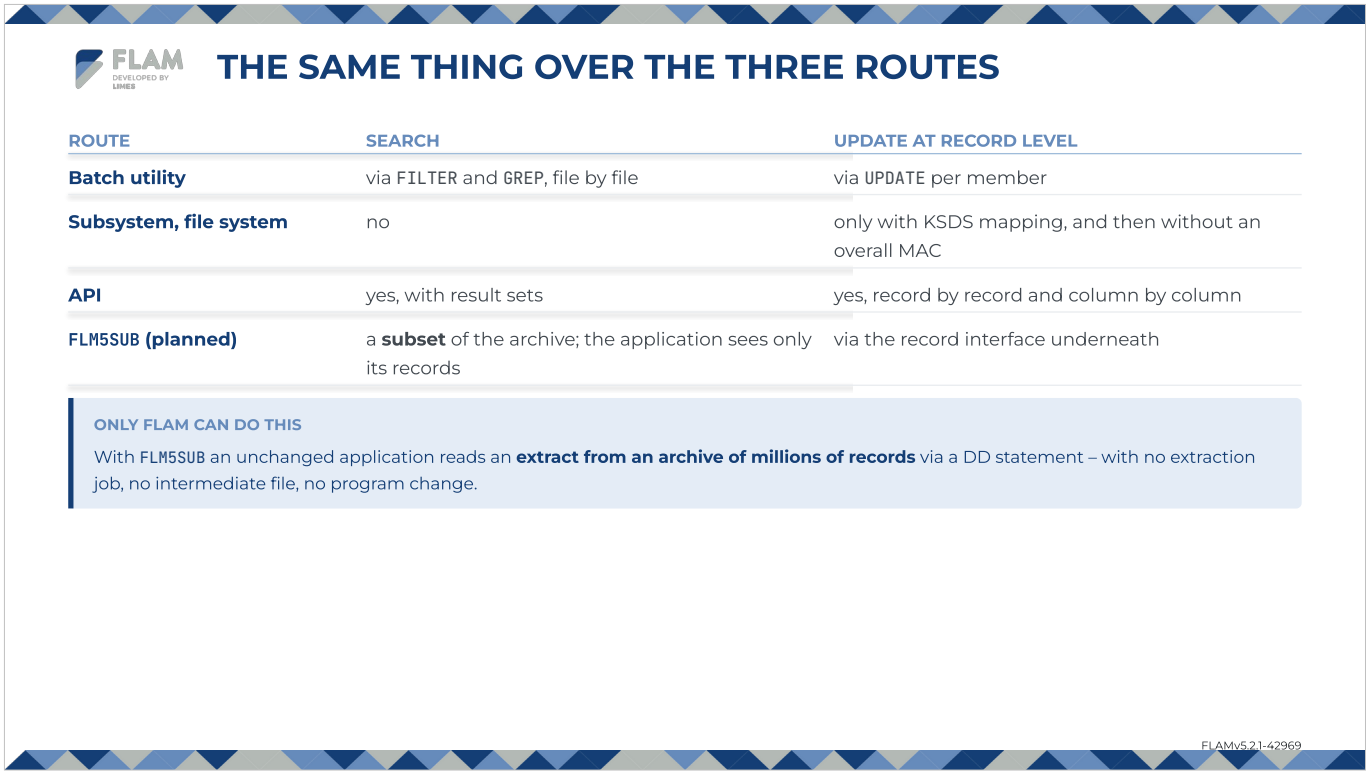
Repetition from the archive deck, here from the operations point of view -- and with the cleanup run that has to be planned for.
The comparison with GDGs works immediately with mainframe people: (0) is the current state, (-1) the one before last.
Important for capacity planning: deleting does not make the archive smaller.
Only PRUNE does that.



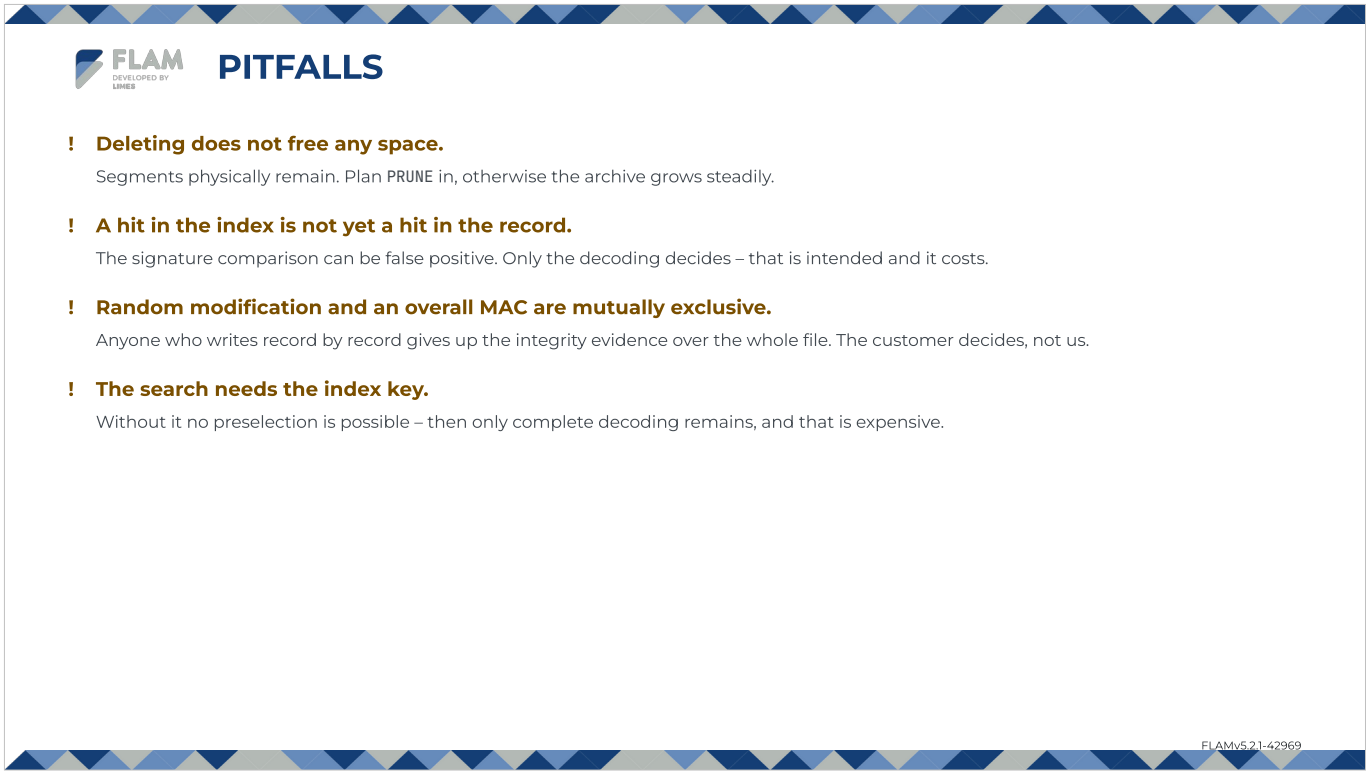
Do not read out. The message is: this is a management facility, not a packer -- and every bit of it can be scripted. The three most often needed in production are LIST, VERIFY and PRUNE.
`COPY` is deliberately in second place, right behind COMP and DECO, because it is both together -- and it has **two variants** that have to be kept apart. The route over the clear data unpacks and packs again; that allows the target to be built differently from the source, with different compression or different keys for instance, but it needs the keys.
`TO.DUPLICATE` moves the compressed and encrypted **segments** unchanged; that is cheap and works without the data key, but it delivers a copy with the same characteristics. For moving an archive the second variant is almost always the right one.
`LIST` outputs more than the name suggests: with `ROWLIST` the **formats** come along, that is the format descriptions of the tables in the archive. That shows how the data lying there is structured -- and it gives exactly the description needed for a search. `FILTER` and `DIFF` are two different things and therefore stand apart: `FILTER` builds a signature from clear values and forms a subset with it, `DIFF` compares -- and not only archive against archive but also **original data against archive content**. That is the case needed after a migration.
REKEY is the function that counts in the crypto conversation:
the data key is re-encrypted without touching the payload.



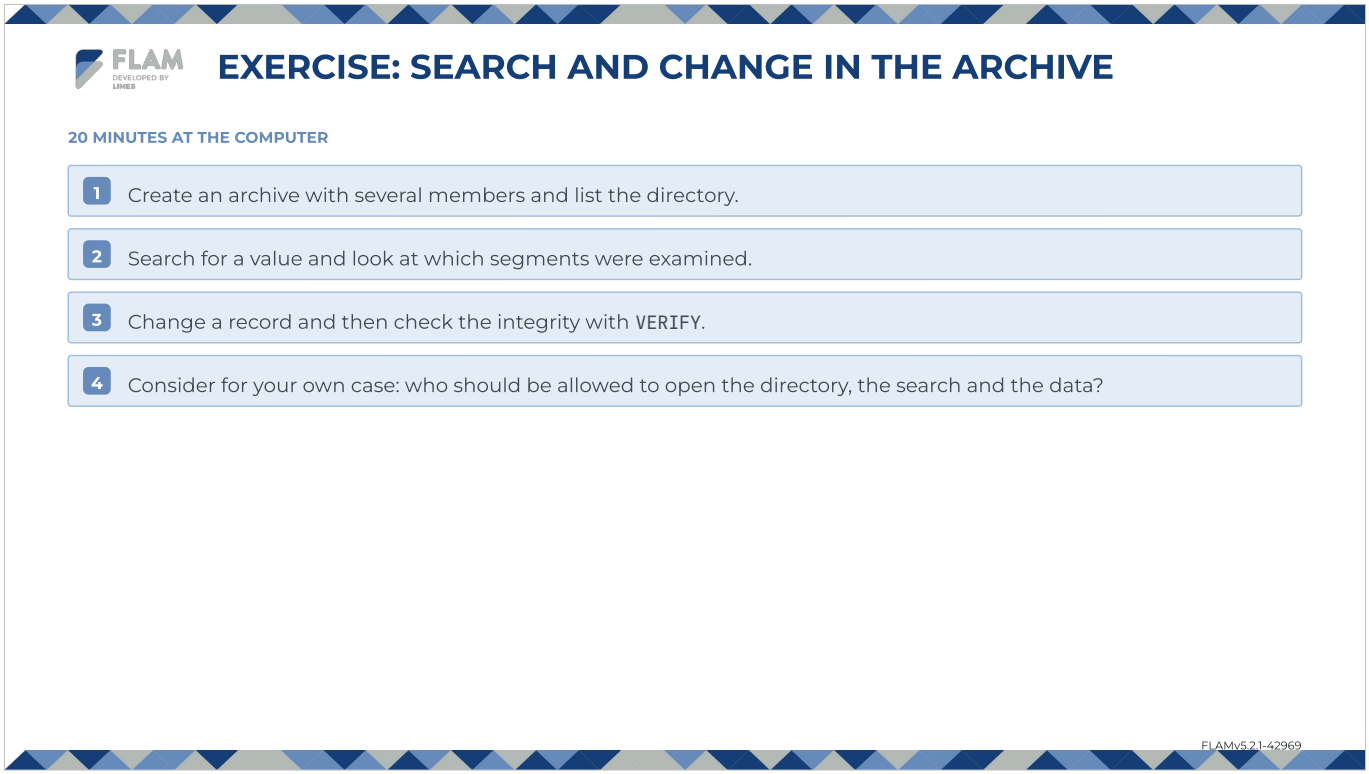
This slide links two topics that stood separately before: the three keys and the integrity check. The connection is the interesting part and was written down nowhere.
Two parameters, and the difference is the whole point of the slide:
`LEVEL` says how deep in the hierarchy the check goes -- `MODE` says what is checked.
Anyone who confuses the two does not understand the output.
The hierarchy is directory, member segments, data segments. `LEVEL` determines up to where the check is cryptographic; below that boundary only the simple checksums run. `DIRECTORY` checks the directory cryptographically, `MEMBER` additionally the member segments, `DATA` also the data segments. `ALL` is the default and checks all segments.
`MODE` determines the what: `CKS` only the simple checksums, `KTV` additionally the key test values, `MAC` additionally the message authentication codes, `ALL` everything.
And now the connection with the keys, which is written nowhere: **without keys you only get at the simple checksums.** Only with key material are key test values and MACs checkable -- and how deep you can go in doing so depends on which of the three keys you have.
Only the directory key: `LEVEL=DIRECTORY`. Directory and member: `LEVEL=MEMBER`. All three: `LEVEL=DATA`.
So `VERIFY` does not check "as deep as possible", but as deep as the key material of the checker reaches. That is a property, not a restriction.
And now the statement that counts in a conversation with an auditor: it lets the archive operator prove the integrity of the data set regularly without being able to read the data. That is exactly the separation of operations and data owner that is otherwise claimed organisationally and here technically enforced.



For this topic the distribution is particularly clear: managing works in batch, transparent reading via the subsystem, but search and targeted changing only via the API.
That is also the answer to the question why the record interface exists at all.
The last line is an outlook, and it is the reason why the slide does not stop at "no" for the subsystem: **with FLM5SUB this route will also work for FLAM5 archives** -- and then in a way that exists nowhere today.
The pattern that becomes possible with it is worth the explanation, because it is the actual benefit: you fetch a **subset** out of a huge archive and give the application only the records from that subset. The application reads via a DD statement and sees an ordinary file -- it does not notice that behind it lies an archive with millions of records, out of which exactly its selection is filtered. No extraction job, no intermediate file, no program change.
For expectation management: FLM5SUB is coming in one of the next releases.
Anyone who wants to address a FLAM5 archive transparently via a subsystem today cannot do that yet.



All four from practice. The first is the one that most often leads to a ticket, the second the one that produces the most misunderstanding.



Twenty minutes. Task four is the interesting one: it leads to the question who in your own house should have which key -- and that is mostly an organisational discussion, not a technical one.
Solution:
1. `ARCHIVE.COMP` creates it, `ARCHIVE.LIST` shows the directory. The output holds members, status and versions; with `VERBOSE=MAXIMAL` also the file status per member.
2. In the log you see how many segments were considered. The point of the task is that there are few of them -- the Bloom filters rule the rest out without unpacking it.
3. After the change a new version arises, and `VERIFY` stays green. That is exactly the difference to FLAM4, where a modifying access destroys the completeness check. If somebody asks about it: see the unit on the FLAM5 archive, slide "The two limits of FLAM4".
4. The expected answer is a division of roles, not a technical answer:
the archive operator gets the directory key, the searching application the index key, and only the data owner the data key. Anyone who puts all three in one hand has given up the separation -- and should do that deliberately.

WHAT TO TAKE AWAY

- Search, read, change, insert, delete – without unpacking the archive
- two-stage: cheap signature comparison, then expensive decoding only on a hit
- the Bloom filter is errs on the side of yes, never no – a no is certain
- result sets are produced without clear text: searching and reading can be separated
- versions remain until `PRUNE` runs – that belongs in the operations calendar

FLAMv5.21-42969

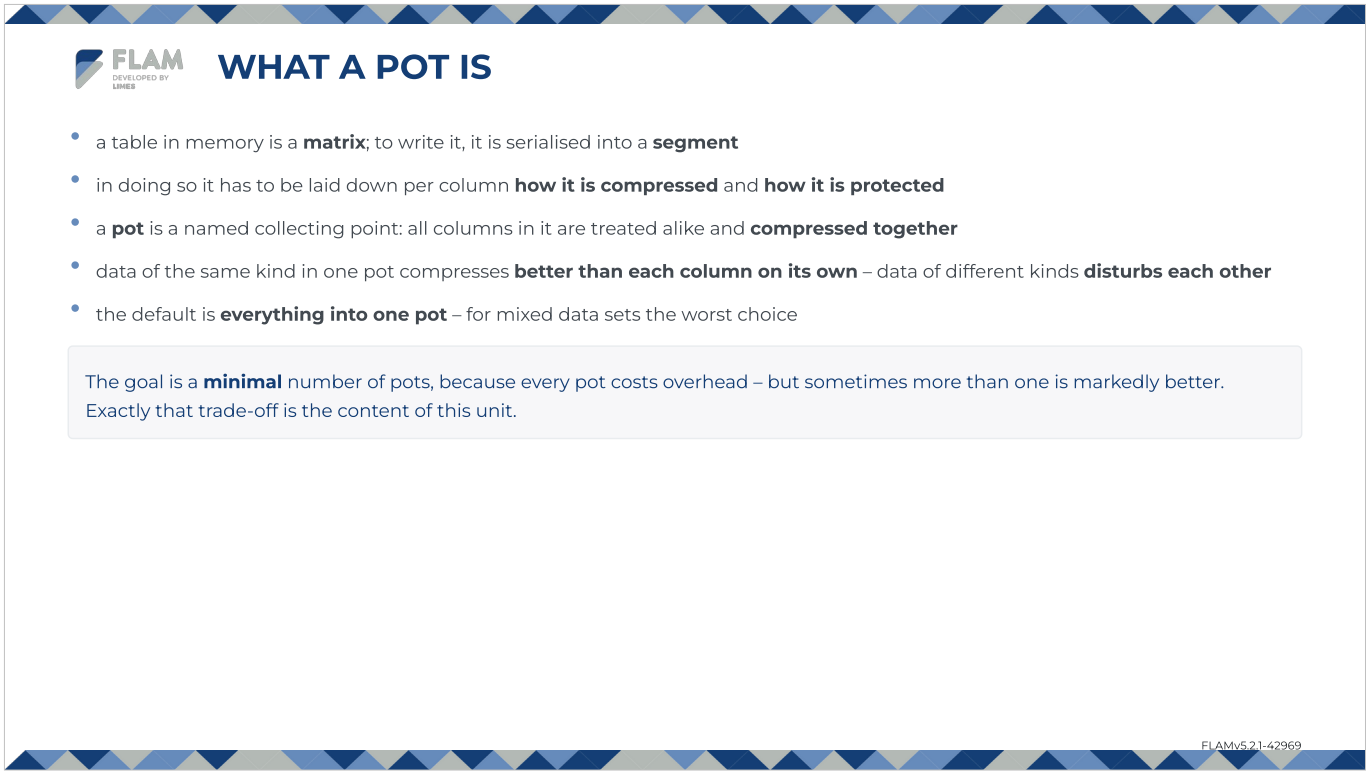
Walk through the learning objectives. That completes the archive block. The next functional topics are conversion and tables.



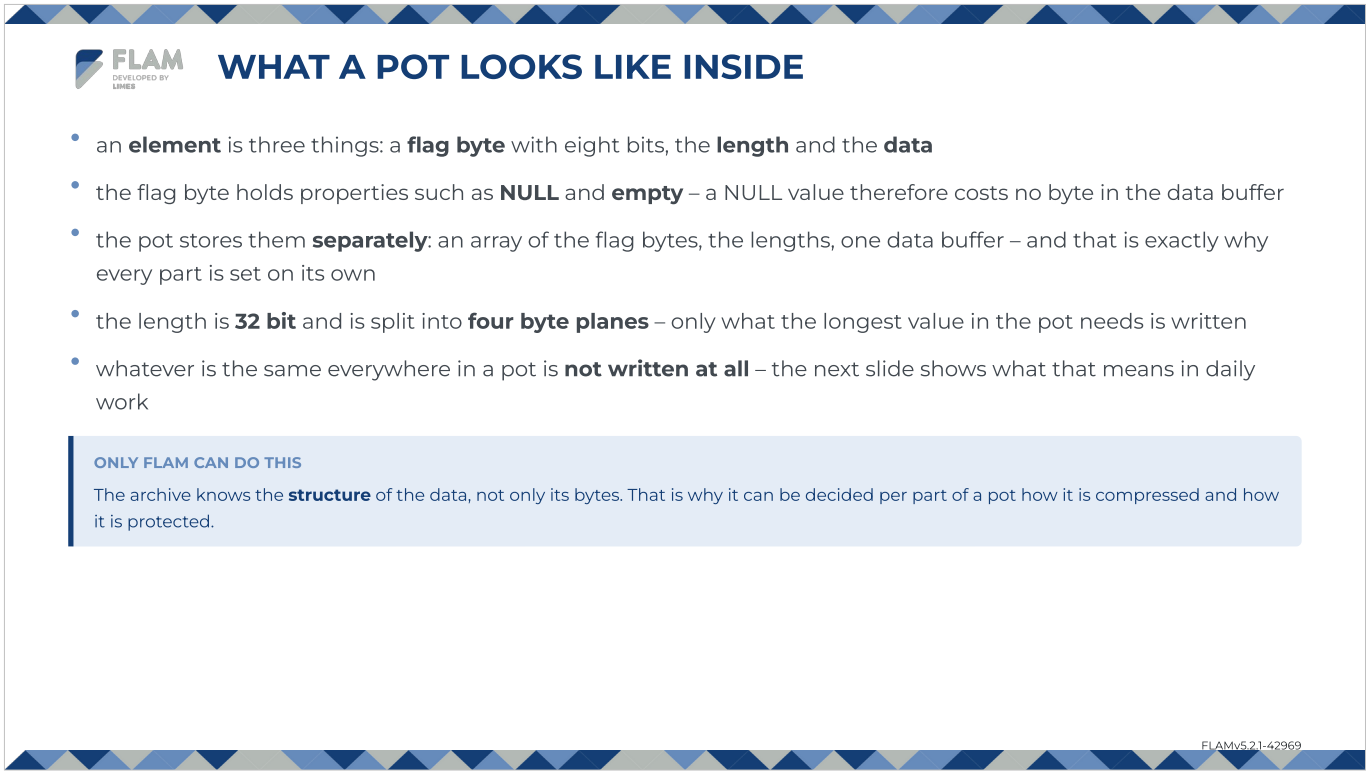
This unit has nothing to do with indexing, even though both are column decisions -- that is why it sat in the indexing deck for a long time and belongs here. What it is about: when a table is written into a segment, it has to be laid down for **every column** how it is compressed and how it is protected. A pot is the collecting point into which columns with data of the same kind are put. And why that deserves a unit of its own: it decides three things at once -- **CPU effort, security and compression ratio**.
Those are exactly the three quantities that matter with a large archive. The sentence from our homepage sums up the goal: only what really is worth protecting is encrypted, and only what pays off is compressed.



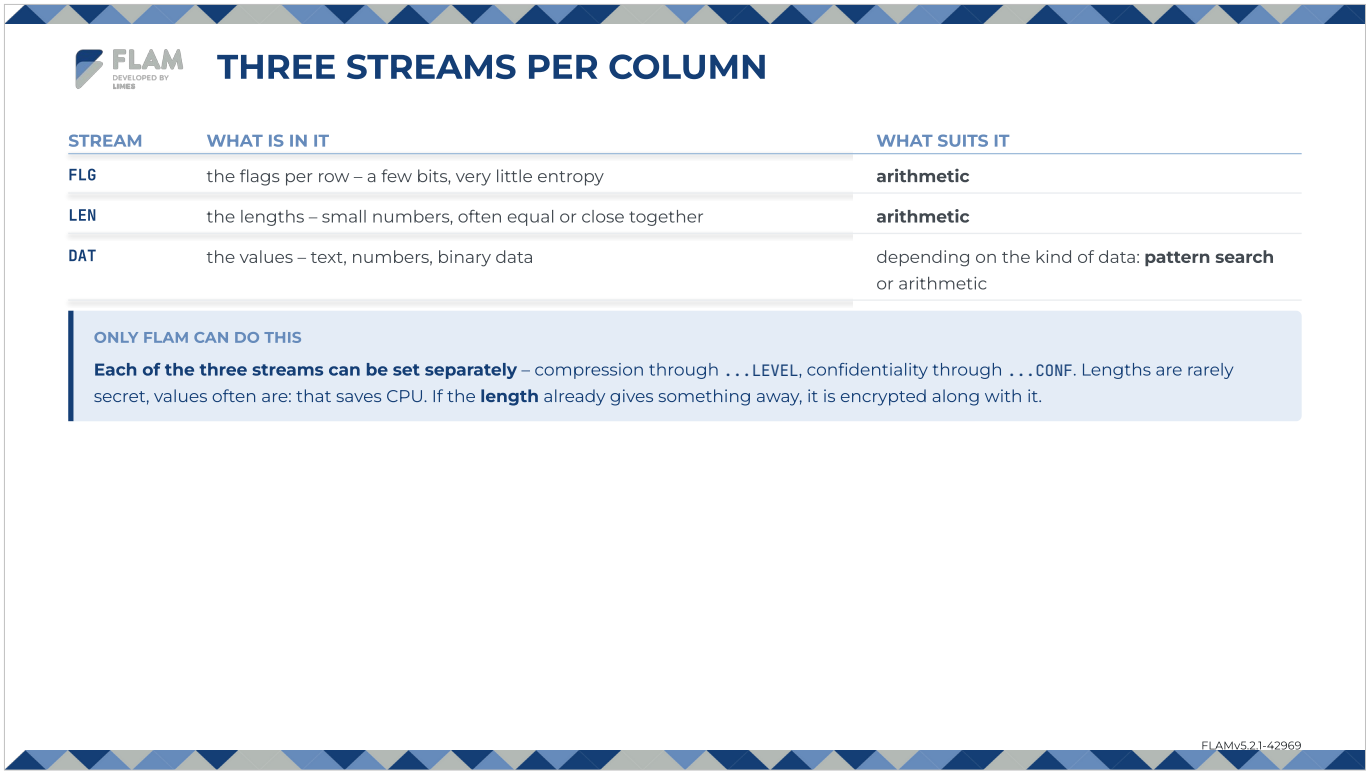
The learning objectives. The second is the core of the unit -- two questions per column, and the design follows. The fifth is the point where you see money: one pot per column is as wrong as one pot for everything, and both occur in practice.



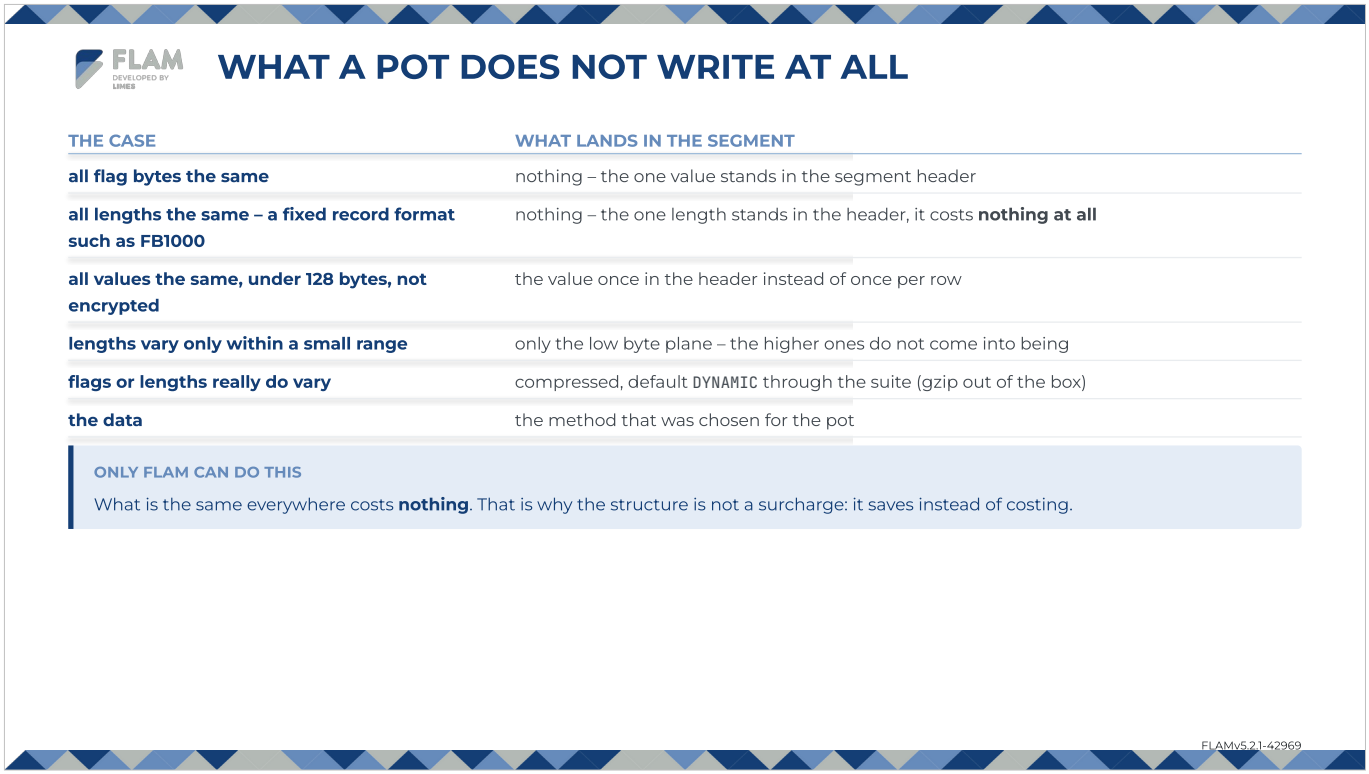
First the mechanism, so that the decisions afterwards make sense.
A table in memory is a **matrix**: columns with typed values, any number of rows. To write it, this matrix has to be serialised into a **segment** -- and in doing so the question arises for every column how it is compressed and how it is protected. A **pot** is the answer to that: you define pots with names, and every column is assigned a pot. All columns of one pot are treated alike and compressed **together**. And that is exactly where the gain and the trap lie. The gain: columns with data of the same kind in one pot compress **better than each one on its own**, because the compressor sees more of the same pattern. The trap: columns with data of different kinds in the same pot **disturb each other**. The sentence from our manual on this, and it is honest: the goal should be a **minimal number** of pots, because every pot costs overhead -- but sometimes more than one is markedly better. What the default does: **everything into one pot.** That is right for many data sets and for mixed data sets the worst choice. That is exactly what this unit is about.



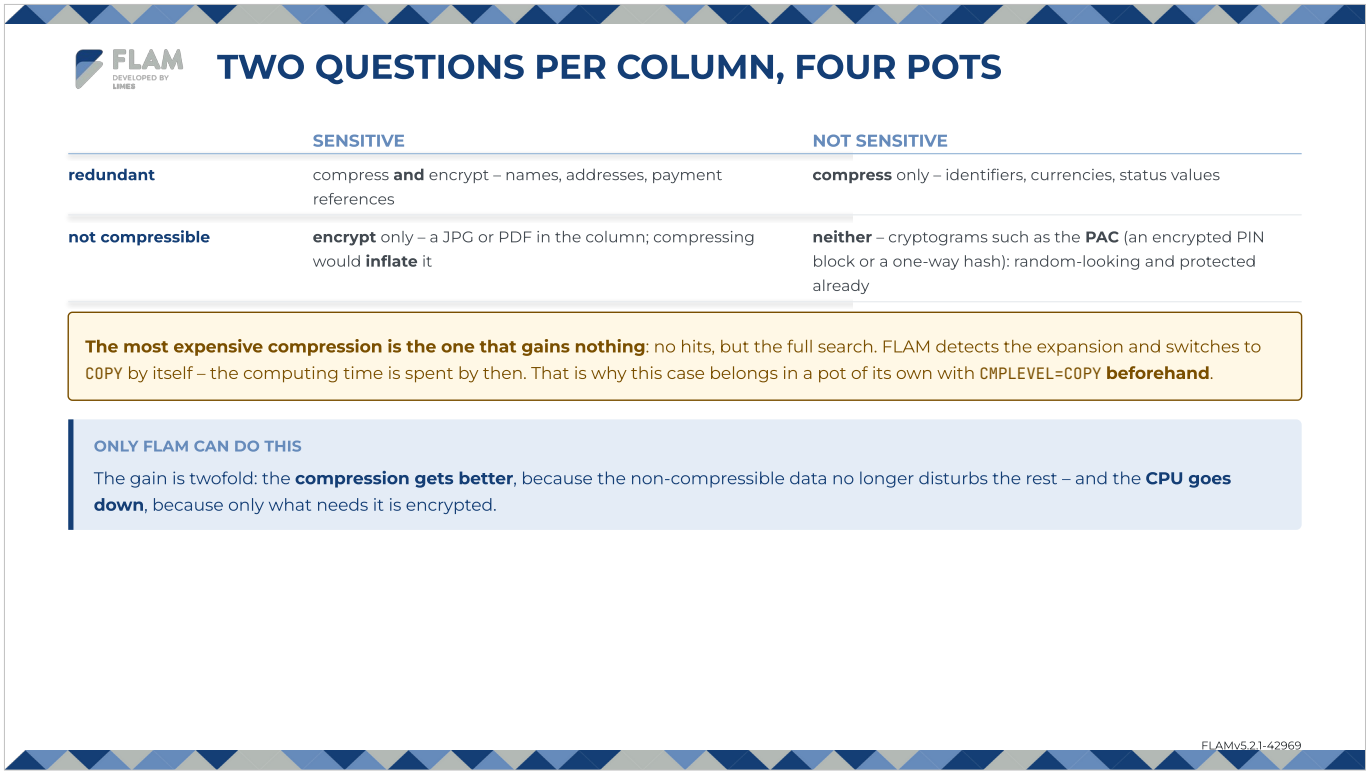
The slide explains why the three streams of the next slide exist at all -- they do not come from a design whim but from the structure of an element.
An element is not just its value. It consists of **one flag byte, one length and the data**. The flag byte holds eight bits that are written along; occupied today are, among others, "the value is NULL" and "the value is empty", the rest is reserve. That is the reason a NULL value in a FLAM archive costs **no** byte of data: it stands in the flag byte.
The pot does not collect these parts element by element but **separately**:
an array of flag bytes, the lengths, and a buffer with the data. That is exactly why a compression method and a protection of its own can be set per part -- which is the next slide.
Two subtleties nobody has to explain but everybody should be able to name when asked. The lengths are split **byte by byte**: if every value is shorter than 256 bytes, only one length buffer is written; a long value costs the second one. And if all flag bytes, all lengths or all values are identical, the whole thing stands **once** in the segment instead of a thousand times -- a column with a constant value is practically free.
For whoever asks about the effect: that is the difference between an archive that compresses a table as a byte stream and one that knows its structure.



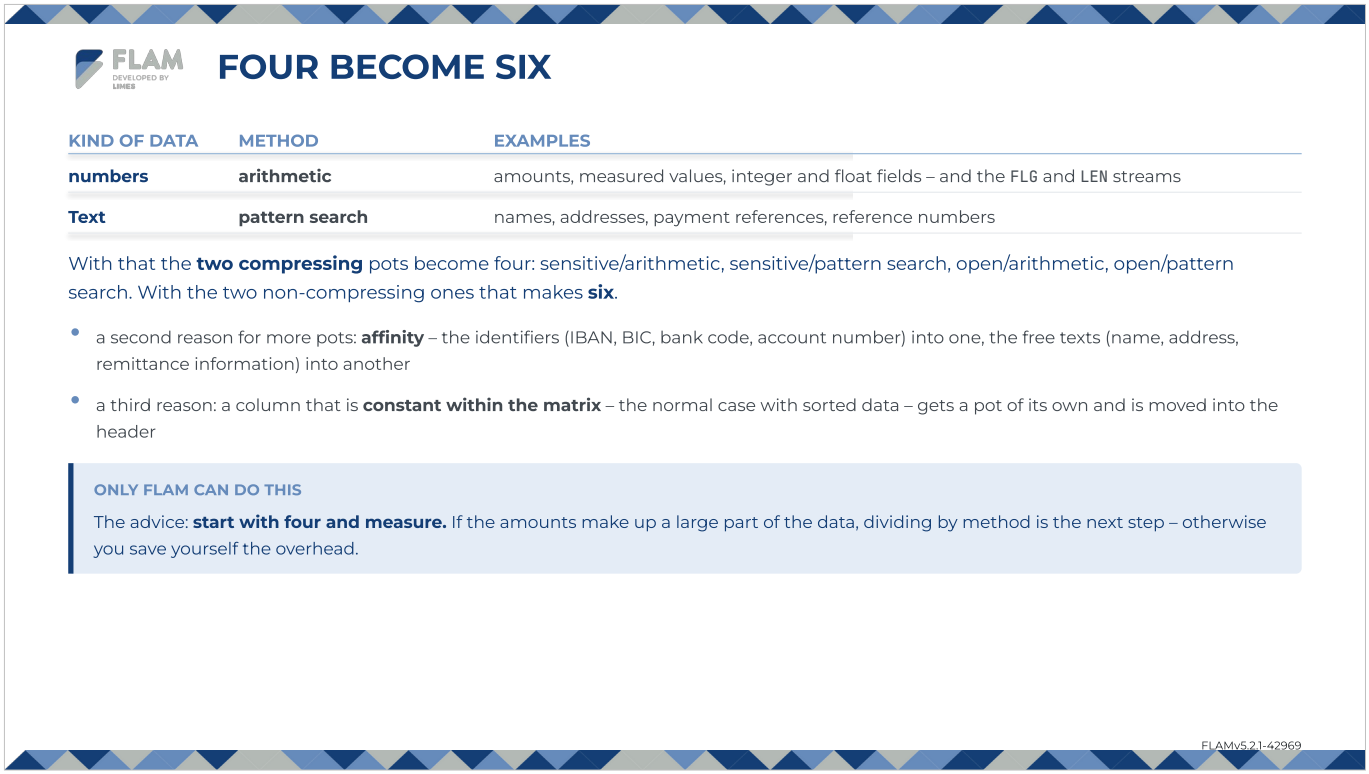
A slide that surprises, and that explains why FLAM reaches such good ratios. A column is not one data stream, but **three**: the flags, the lengths and the values. That is not bookkeeping, it is a compression decision -- because the three are entirely different in kind. **The flags** are a few bits per row: is the value present, is it empty, is it the same as the one above.
Over a thousand rows that is a bit sequence with very little entropy -- arithmetically compressible down to almost nothing.
The lengths are small numbers, often equal or close together. That too is arithmetically far more compressible than with pattern search. **The values** are the actual material -- text, numbers, binary data. And now the point: **each of the three streams can be set separately**, in compression and in confidentiality. In `POTPAR` they are called `FLG`, `LEN` and `DAT`, each with `...LEVEL` for the compression and `...CONF` for the protection. What this is needed for: the lengths of a column are rarely secret, the values often are. So the values can be encrypted and the lengths left in the clear -- that saves CPU. The other way round: if the **length** of a value already gives something away, it is encrypted along with it. And our own practice, because it shows the principle: in our archives we compress the **length fields arithmetically** and the **data through pattern search**. That is the same thing this slide recommends -- only built that way internally already.



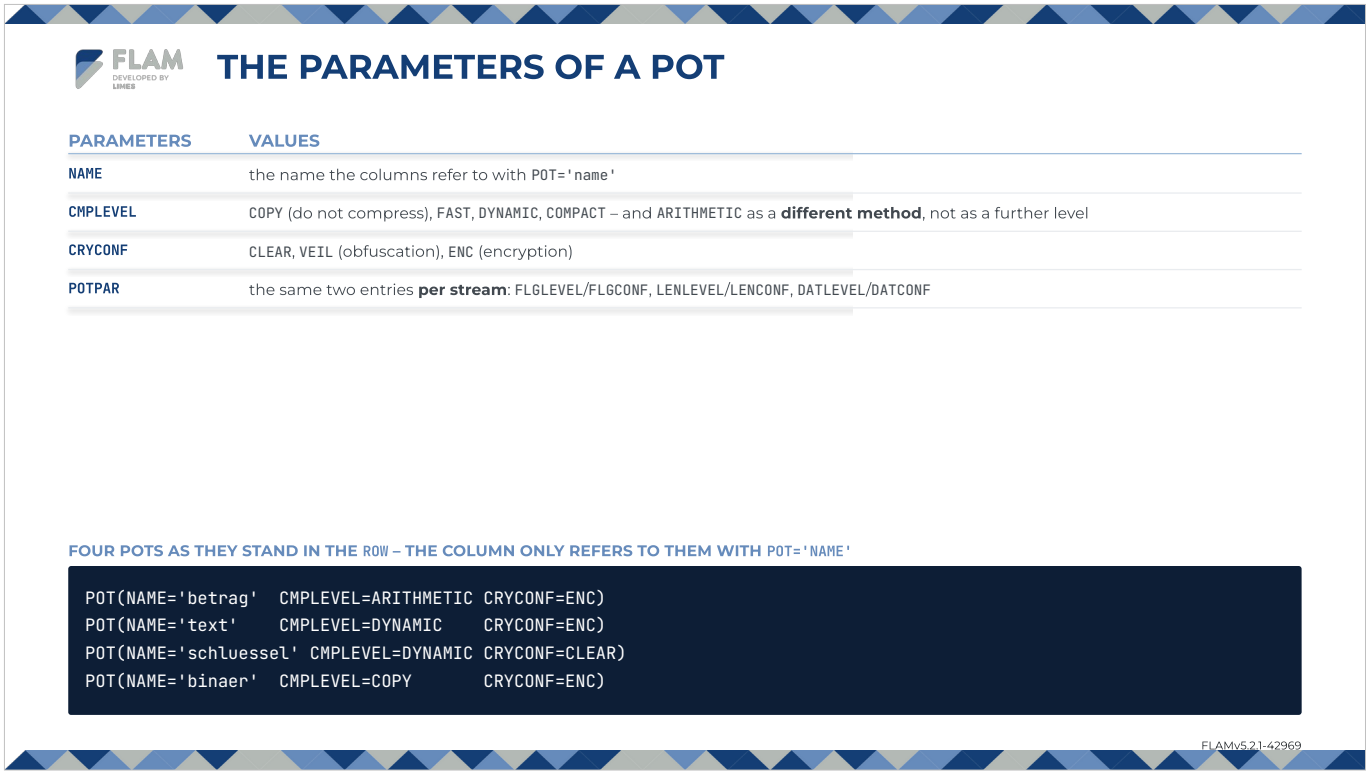
The slide that explains why a FLAM archive becomes so small with fixed record lengths – and it is the answer to the question "does the structure not cost space?".
While collecting, the pot compares every element with the first one and keeps three things in mind: are all flag bytes the same, are all lengths the same, are all values the same. If one of them holds, the whole buffer is **not written**; the one value stands in the segment header instead.
That is not rare but the normal case: flags are the same everywhere most of the time, and with a fixed record format – a PS with FB1000 for instance – all lengths are the same too. Then the length costs **nothing**.
For the values the same rule holds with two limits: the value must be shorter than 128 bytes, and the pot must not be encrypted – the segment header lies under the member key, and an encrypted value has no business there.
If the lengths do vary, the second saving arrives: the length is 32 bit and is split into four byte planes. Only what the longest value in the pot needs is written – under 256 bytes one plane, under 64 KiB two.
The higher planes, which would be 0x00 everywhere, do not come into being.
And only when something really varies is it compressed: flags and lengths with the default DYNAMIC, the method behind it comes from the compression suite, and that is **gzip** out of the box.
The data get the method that was chosen for the pot. The level for flags and lengths can be changed -- **it should not be**, the default is the right choice for this kind of data.
A last point for the question back: if the compression gains nothing, it is discarded and the raw buffer is written. A pot therefore never grows through compression.



The core slide, and it is deliberately kept so simple that you remember it. Per column you ask **two** questions: **Is it redundant?** Do values occur more than once, are there patterns, does compression pay off at all? **Is it sensitive?** Is there something in there that has to be protected? Two questions with two answers each give four cases, and every case is a pot: **Redundant and sensitive** -- compress **and** encrypt. That is the normal case for names, addresses, payment references. **Sensitive, but not compressible** -- encrypt only. The classic example is a JPG or a PDF in a column: it is compressed already, a second pass would only **inflate** it and cost CPU. **Redundant, but not sensitive** -- compress only.
Identifiers, currencies, status values, posting keys.
Neither -- neither compress nor encrypt. The typical case is **cryptograms**: data that is encrypted already. It is not compressible, because it looks like randomness, and it needs no protection, because it has it already. So that this does not stay abstract, the example from payment traffic: the **PAC**, the personal authentication code -- that is the encrypted PIN block with a freely chosen PIN, or a randomised one-way hash for verifying the customer's PIN. Something like that stands as a column in the file, and it is the model case for "neither":
random-looking and protected long since. The gain of this division is twofold: the compression gets better, because the non-compressible data no longer disturbs the rest, and the CPU goes down, because only what needs it is encrypted. Two points belong here, and they are the real reason for the second row of the table. **The less redundant the data is, the more computing time the compressor costs.** That is against the intuition: the most expensive case is exactly the one in which nothing comes out. The compressor finds no hits but has to search through everything all the same -- a JPG in a column is the most compute-intensive thing you can do to a pot. **FLAM detects the expansion and switches to `COPY` by itself.** If the compressed output grows bigger than the raw buffer, we write the raw buffer -- that holds for a single segment and a single buffer too, so for the case where a large JPG suddenly turns up in a pot with everything in it. Inflated data therefore never exists, and the decoding saves itself the work as well. **The computing time is burnt, though.** It can only be recognised once it has been tried. That is exactly why this case belongs on `CMPLEVEL=COPY` from the start through the pot design -- not because the result would be wrong otherwise but because one saves oneself the attempt. That is the basic pattern. **As a rule these four are what you need.** Whoever starts with one and takes the JPGs out already has the largest part of the gain.



The refinement, and it pays off markedly with large data sets.
The two pots in which compression happens can be **divided once more** -- according to the method that suits the kind of data.
Because the methods work on different principles: **Arithmetic compression** is good for **numbers**: amounts, measured values, integer and float fields, and of course the length and flag streams from slide 4. **Pattern search** is good for **text**:
names, addresses, payment references, reference numbers. So if you have a table in which both occur -- and in payment traffic that is always the case --, then the two compressing pots become four: sensitive/arithmetic, sensitive/pattern search, open/arithmetic, open/pattern search. With the two non-compressing ones that makes six. Six pots is already a lot, though -- and with that we are at the trade-off of slide 9. For a large data set with many rows it pays off; for a table with a thousand rows and twenty columns it is too fine-grained. And there is a second reason for more pots that has nothing to do with the method: **the affinity of the data.** Measurements at a central bank showed that the **identifiers** belong together in one pot -- IBAN, BIC, bank code, account numbers --, and the **free texts** in another: names, addresses, payment references.
Identifiers are built similarly to each other, free texts are too, and mixed they disturb one another.
The third case is the most rewarding and the easiest to overlook: **a column whose value is the same throughout the matrix** gets a pot of its own.
Then the shortcut of slide 6 takes effect -- the value is moved into the header and the column costs practically nothing. That happens in practice when the data is **sorted**: within one matrix the tenant, the date or the currency is then constant.
Often the `LITERAL` switch does that as well, but through a pot of its own one has it in hand. The practical advice: **start with four and measure.** If the amounts make up a large part of the data, dividing by method is the next sensible step. If not, you save yourself the overhead.



The reference slide. A pot has a name, a compression level, a confidentiality level -- and optionally `POTPAR` for the three streams individually. With `CMPLEVEL` the order is a scale: `COPY` does not compress, `FAST` is fast with a lower ratio, `DYNAMIC` is the balanced way, `COMPACT` is slow with a high ratio. And `ARITHMETIC` is not a point on this scale, it is a **different method** -- for the numbers of slide 6. With `CRYCONF` there are three levels: `CLEAR` is no protection, `ENC` is encryption. In between lies `VEIL`, confidentiality protection through obfuscation -- the parameter help says exactly that, and more is not documented about it. Whoever has a protection requirement takes `ENC`; `VEIL` is the intermediate level for data that should not lie open but is not worth a key either. And the assignment: on the column there is `POT='name'`. A column without an entry ends up in the default pot. A note on the order in the parameter string, because it is a frequent question: the pots are defined **in the `ROW`**, next to the columns, not inside a column. The column only refers to the name.



A switch that belongs to pot design although it does not stand at the pot -- and that has a security consequence you have to know. The mechanism: some columns contain only a handful of different values -- a status indicator, a posting key, a country code. For such columns there is the **global literal matrix**: with `LITERAL` at the column the values are moved there instead of going into the assigned pot. The gain:
the values stand there **once** and are referenced, not again in every segment. With a column holding ten different values across millions of rows that is considerable. The parameter help explicitly names selections as the purpose -- a selection over a few literals becomes cheap with it.
And now the security consequence, and it is the reason this slide exists: **the literal matrix is protected with the directory key**, not with the data key. What follows from that: whoever -- as described in the unit *The FLAM5 archive* -- wants to distinguish **directory access** from **data access** in security terms must **not** set the switch for sensitive columns. Otherwise somebody with the directory key can read these values although they are not allowed to read the data.
In practice literals are mostly uncritical -- a posting key is no secret. But it is a decision, not an automatism. And it belongs made deliberately, because it falls when the archive is created and cannot be changed later.



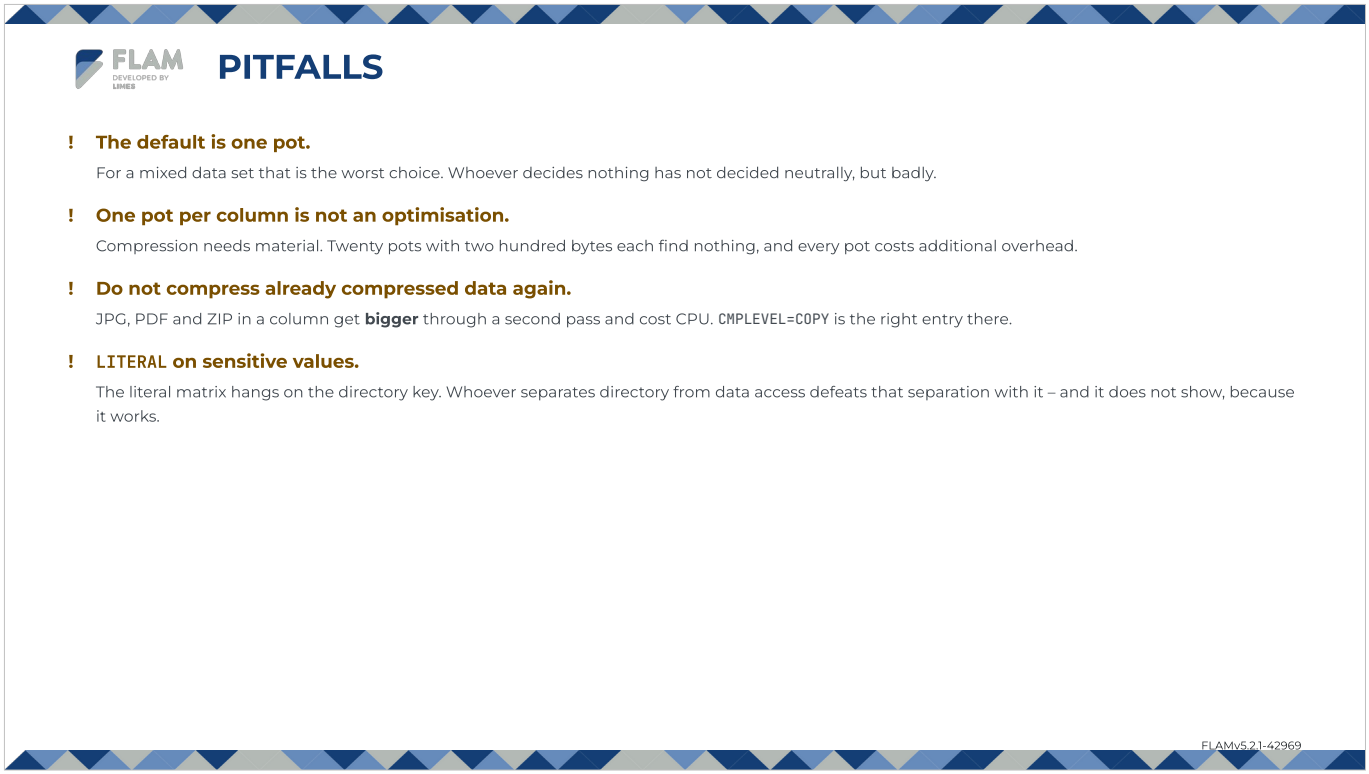
The slide that stands between the pot and the archive -- and the answer to the question that always comes in a security discussion: "and what is the integrity protection?" What a pot settles for itself, `DEFAULTS()` presets for **all** pots. Two specifications we know already (`CMPLEVEL`, `CRYCONF`), two are new and belong explained together. `**INTPROT**` is the integrity level per segment, and it has four steps: `NONE` without protection, `CHKS` with a checksum, `AUTH` with an **encrypted** checksum and `MAC` with a message authentication code -- concretely an **AES-CMAC**, computed over the compressed data. The default is the most honest place in the whole parameter help: **`MAC` when a key is there, otherwise `CHKS`**. Whoever encrypts thus gets the full proof automatically; whoever only compresses gets a checksum.
Exactly these steps appear again as the checking depth of `VERIFY`. `**SERASE**` erases data from storage as soon as it is no longer needed -- default `AUTO`, which means `YES` when a key is in play. For a protection requirement analysis that is the line one shows. `MAXREC` and `MINSIZ` are the adjusting screws of the matrix size; they stand in the performance unit.
And one level higher lie the two **suites**: `CMPSUITE` chooses the compression method (`GZIP`, `LZ4`, `BZIP2`, `ZSTD`), `CRYSUITE` the crypto combination -- `V06` is the default and means AES-256 for derivation and encryption with CRC32 as the checksum. The six suites differ in the key length and in whether CRC32 or FNV32 is computed.



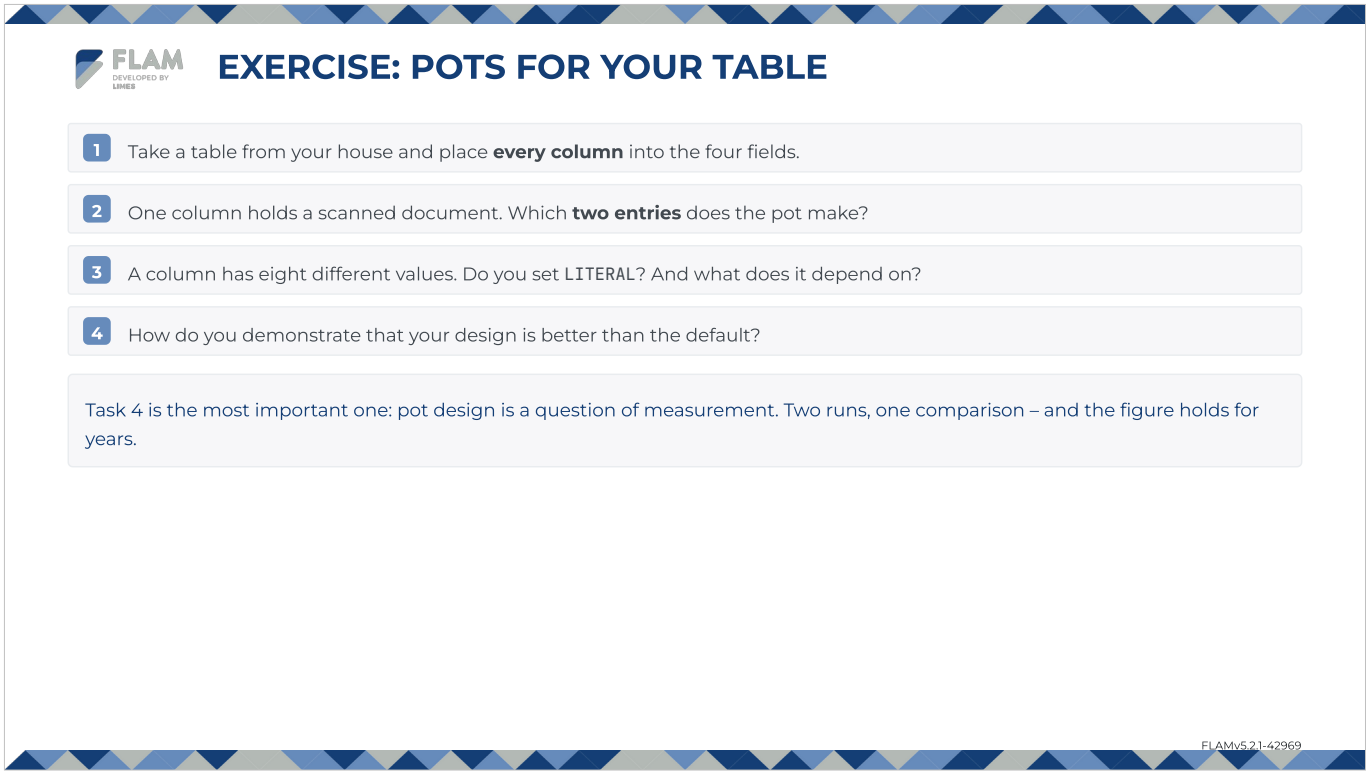
The slide that saves you from the two obvious mistakes -- and both occur in practice. **One extreme: everything into one pot.** That is the default. The mistake in it is not the quantity, it is the mixture: a JPG and a cryptogram in the same pot as the names **disturb the compression** of the names. The compressor keeps seeing random data between usable patterns and therefore finds less. **The other extreme: one pot per column.** Sounds like maximum optimisation and is not, because now **every pot has too little data**. Compression needs material: a method that is allowed to run over two hundred bytes finds nothing. On top of that comes the overhead per pot, which our manual explicitly names. The optimum lies between the two, and it hangs on the amount of data per column in the segment. The order in which you approach it: **First** separate out the non-compressible columns -- JPGs, cryptograms. That is the largest single gain and costs one pot. **Second** separate by protection requirement, so that only what needs it is encrypted. That saves CPU. **Third**, if the data set is large enough, separate by method -- numbers arithmetically, text through patterns. And after that **measure**. That is the sentence to pass on: pot design is a question of measurement, not a question of belief. Writing the same data set with two designs and comparing the sizes costs an afternoon and then holds for years.



The slide in which the participants should recognise their own data. A posting table as every bank has one. The columns in order: **IBAN and name** are sensitive and redundant -- the same accounts and the same names occur again and again.
Text, so pattern search, and encrypted. **The amount and the date** are sensitive and redundant, but they are **numbers**. Arithmetic, and encrypted. **Currency and posting key** are not sensitive and highly redundant -- a handful of values across millions of rows. Compress only.
And here `LITERAL` is the right choice, because it is exactly the case the literal matrix was built for. **The document scan**, a JPG or PDF, is sensitive and not compressible. Encrypt only, `CMPLEVEL=COPY`. **The cryptogram** -- a field encrypted by an upstream stage, say -- is neither. No pot effort. That is four pots and one literal case. And the comparison that justifies the unit: if everything lay in one pot, the document scan would spoil the compression of names and payment references, and the amount would be compressed with a method that is the wrong one for numbers. If the question about figures comes up: our homepage names ratios up to 50 to 1 -- those are data sets where structure and pot design fit together. A concrete figure for your own data you get only by measuring, and that is exactly what the participants should be urged to do.



Four. On the first: the default is one pot, and for a mixed data set that is the worst choice. Whoever decides nothing has decided badly. On the second: one pot per column sounds like optimisation and is the opposite. On the third:
compressing already compressed data a second time costs CPU and makes it bigger. That is not a borderline case, that is the rule with JPG, PDF and ZIP in columns. On the fourth:
`LITERAL` on sensitive values defeats the separation between directory and data access. Nobody notices it, because it works.



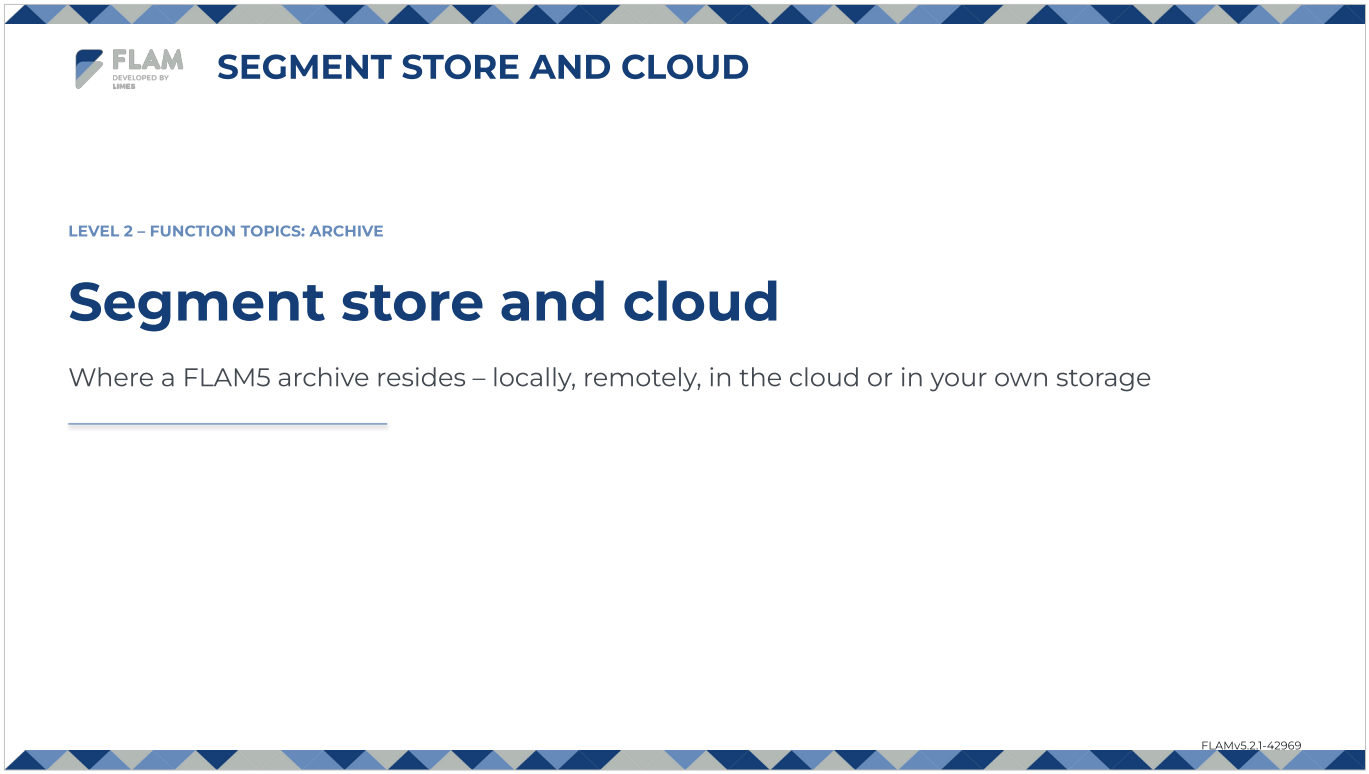
Twenty minutes, and this exercise should be done with a real table of the participants if there is one in the room. The solutions: **Task 1.** Two questions per column -- redundant, sensitive -- and then enter them into the four fields. Pick out the columns with binary contents and cryptograms first, that is the largest gain. **Task 2.** `CMPLEVEL=COPY` and `CRYCONF=ENC`. Compressing would inflate it, and it has to be protected. **Task 3.** Yes, if it is only a handful of values -- but not if these values are themselves worth protecting and the house separates directory from data access. **Task 4.** Write the same data set twice, once with one pot and once with the design, and compare the sizes and CPU times. That is the only sound answer -- everything else is guesswork.

WHAT TO TAKE AWAY

- a **pot** is a named collecting point; columns in it are treated alike and **compressed together**
- **three streams** per column – `FLG`, `LEN`, `DAT` –, each one settable separately in compression and confidentiality
- **two questions** give four pots: redundant and sensitive, sensitive alone, redundant alone, neither
- **arithmetic for numbers, pattern search for text** – that makes six pots, if the data set is large enough
- **one pot disturbs, too many starve** – and `LITERAL` hangs on the directory key
- it is a question of **measurement**: two runs, one comparison, and the decision holds for years

FLAMv5.21-42969

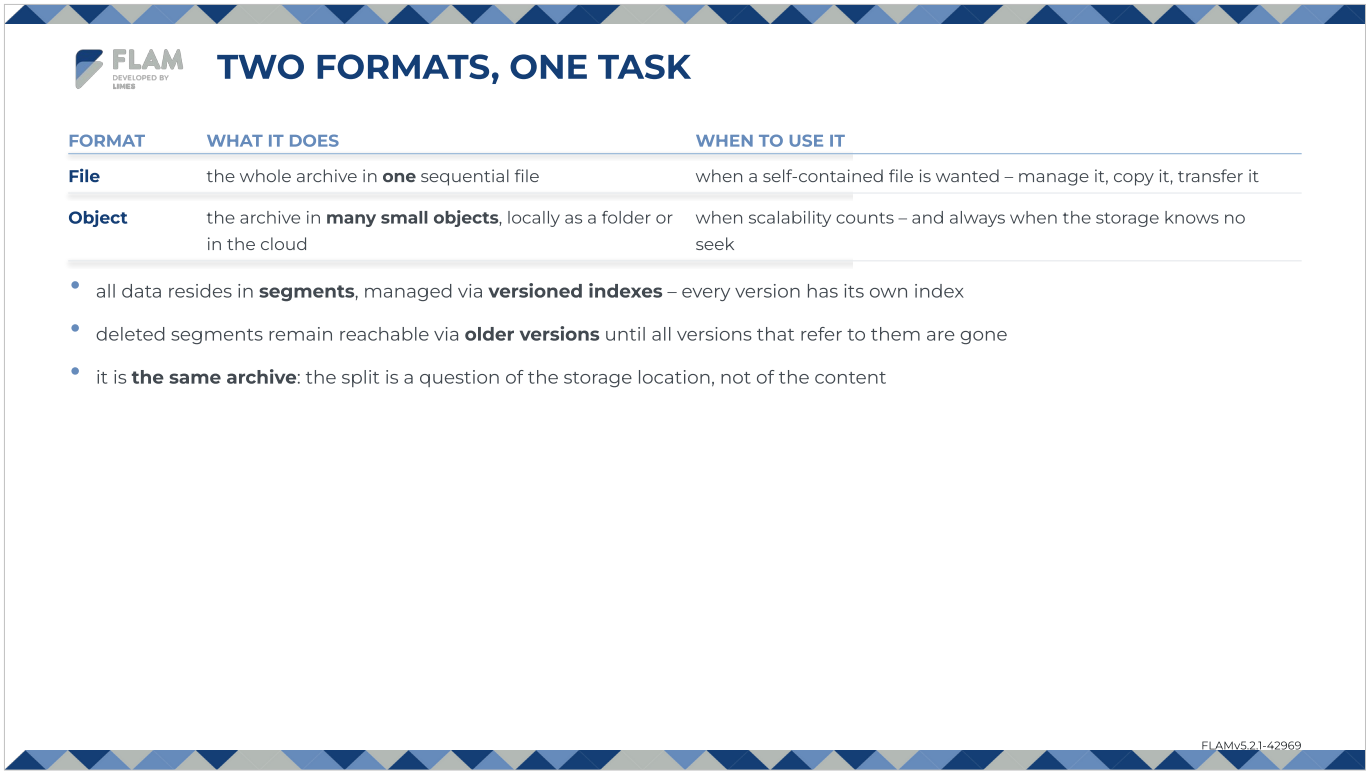
Go through the learning objectives. The closing sentence is the one from the title slide: encrypt only what is worth protecting, and compress only what pays off. And the pointer to the neighbouring unit: indexing is the **other** column decision, and it falls at the same time -- when the archive is created.



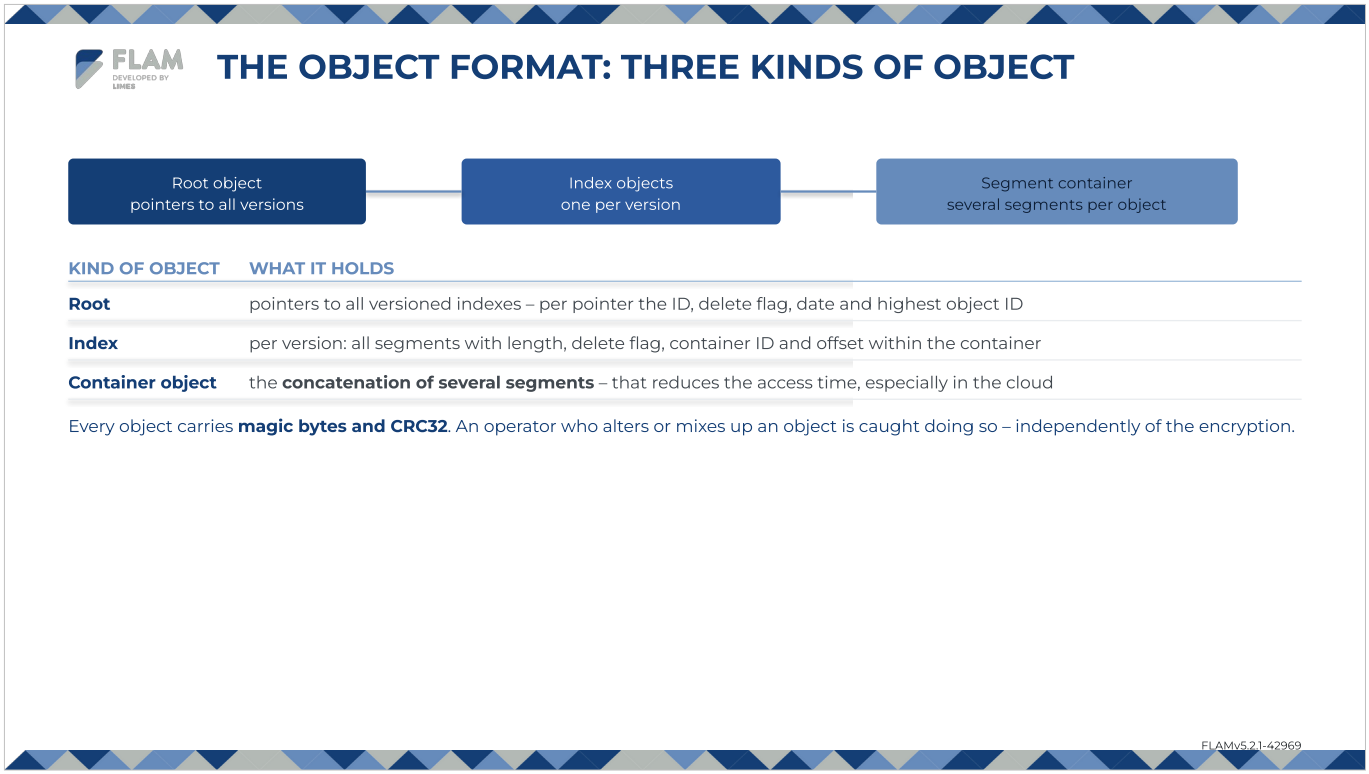
The unit that answers **where a FLAM5 archive physically resides** -- and why that question is separated from the question of what is in the archive. That is the separation that was named as an architectural principle in the archive unit: **archive management** and **segment storage** are two layers. This unit fills in the lower one. In practice it answers three questions that come up in every cloud conversation: what actually lies with the operator? How often does FLAM reach across the network? And what do we do when the customer has a store we do not know? The unit *The FLAM5 archive* is assumed. The unit *Indexing, filters and pots* is useful, but not necessary.



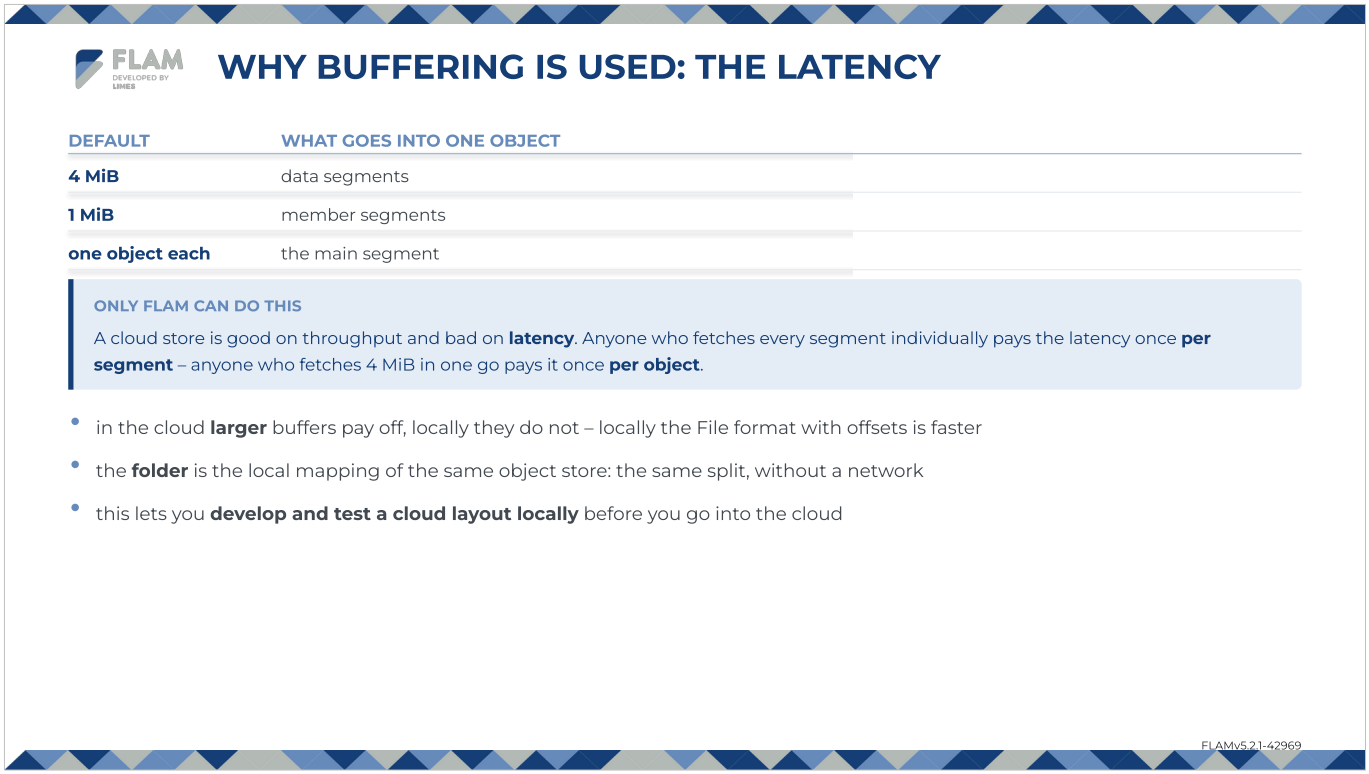
The learning objectives as a contract. The third is the one you are grateful for in the end: understanding why the buffering exists. Without it an archive in the cloud would be unusably slow, and with the wrong setting it is that anyway.
The fifth objective is aimed at the systems people: that you can attach any store you like, and on two levels at that.



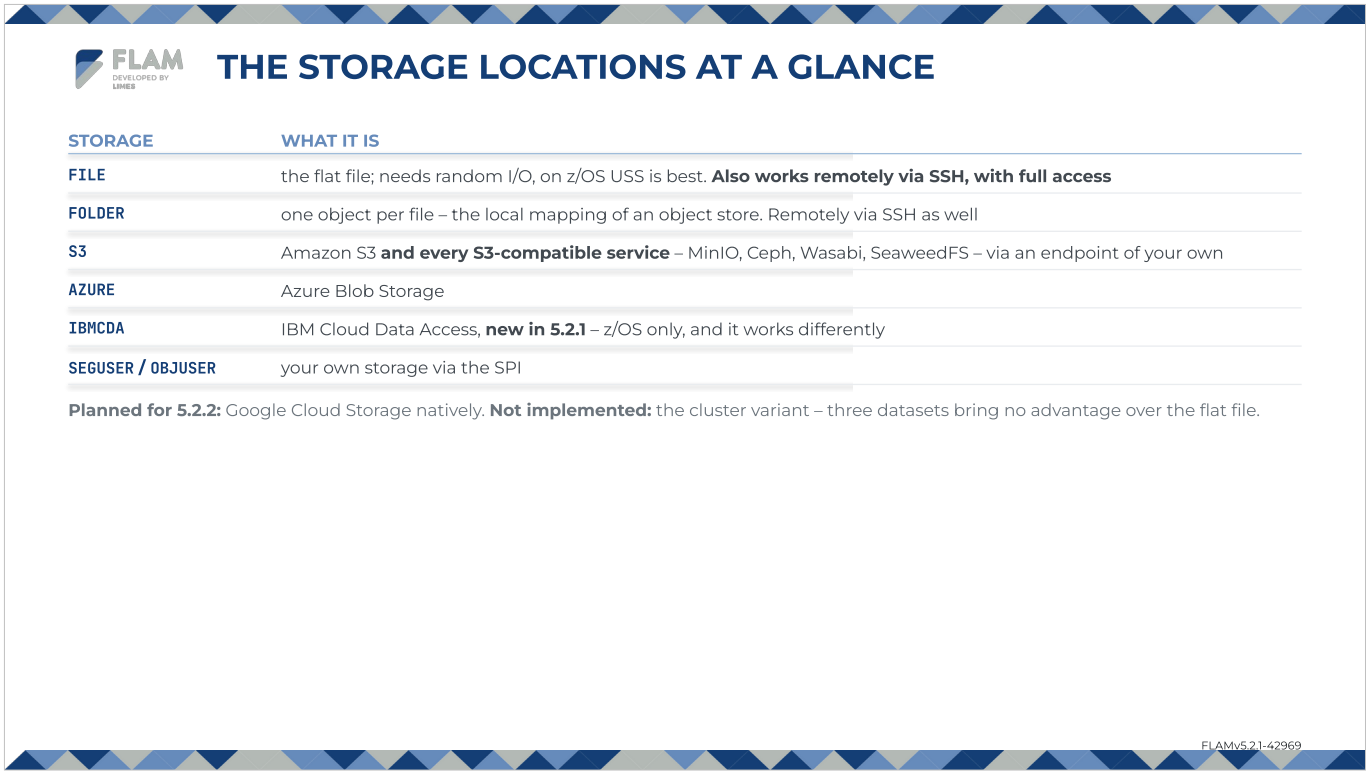
The basic distinction, and it is worth the whole slide, because everything else hangs on it. The segment store is the basis for the storage of FLAM5 archives. All archived data is broken down into **segments**, and the segments are managed via **versioned indexes**. Every version of the archive has its own index, which lists all segments with their metadata. From that follows a property you have to know in operations: **when segments are deleted, they remain reachable via the older versions** -- until all the versions that refer to them are deleted. That is the mechanics behind what turned up in the archive unit as versioning, and the reason why `PRUNE` exists. And there are **two formats**, for two different worlds: the **File format** puts the whole archive into **one** sequential file. That is the right choice when you want a closed file -- for managing, for copying, for transferring. One archive, one file. The **Object format** puts the archive into **many small objects**. They can reside locally in a folder structure or in a cloud store. That is the choice when scalability counts -- and the only choice when the store knows no seek. The sentence for the room: **it is the same archive.** The split is a question of storage, not of content. You can transfer an archive from one format into the other with `ARCHIVE COPY`, without unpacking it.



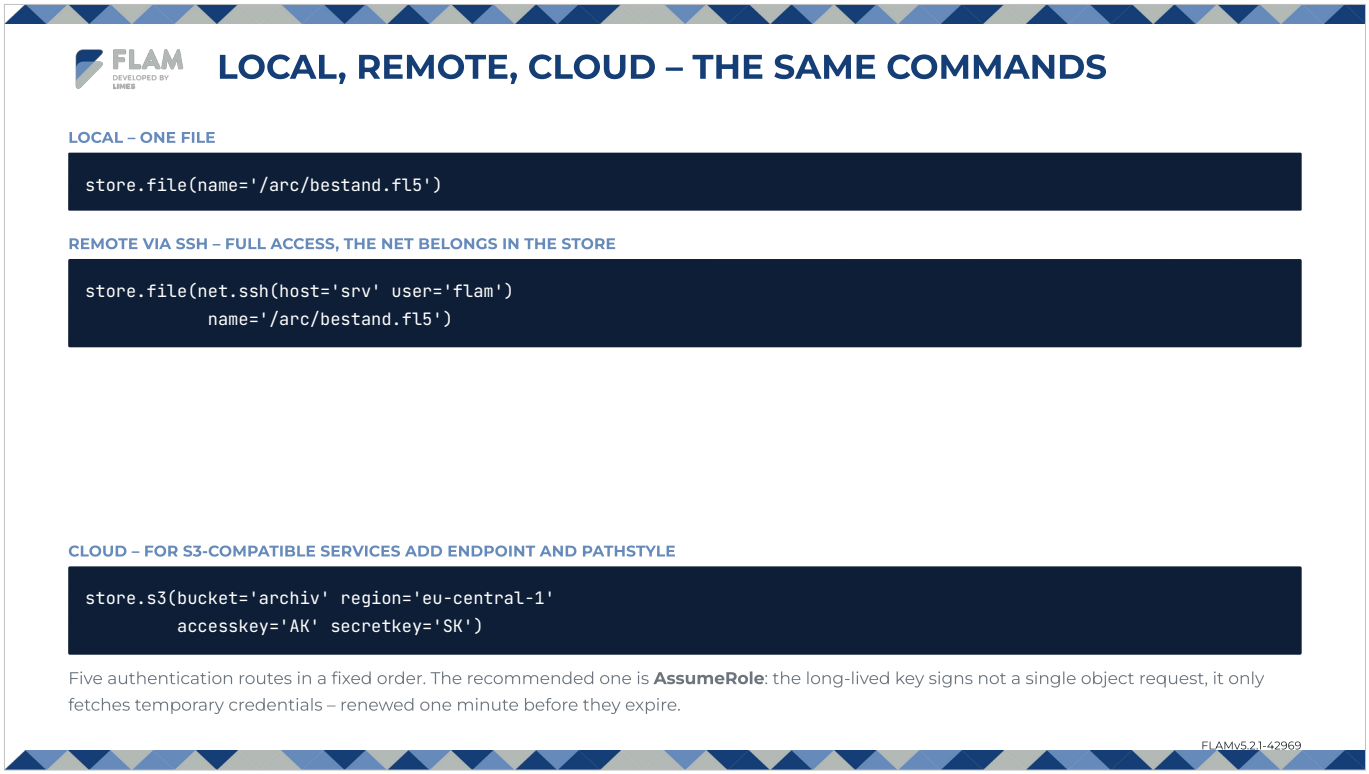
Now more precisely, because anyone going into the cloud should know what actually arrives there as objects. There are exactly three kinds: The **root object** is the entry point. It holds the pointers to all versioned indexes -- per pointer an ID, a deletion flag, the creation date and the highest object ID in use. Whoever reads the root object knows which versions exist.
The **index objects** are one per version. An index holds the list of all data segments of that version, and for each segment: length, deletion flag, the object ID of the container it lies in, and the offset inside that container. The **segment containers** are the interesting part: they are **concatenations of several segments.** And the reason is stated in the manual: combining them reduces the access time, especially with cloud storage. That is the buffering explained on the next slide -- here it is first of all visible as a data structure. A detail that counts for security and helps in the cloud conversation: **every one of these objects carries CRC32 and magic bytes.** An operator who alters or mixes up an object is caught doing it -- independently of the encryption.
And on the question about the overhead, which usually comes up here: the metadata of the segment and object management is **compressed internally with ZSTD**. So the management too costs hardly any space -- that is not an aside but the reason why segments can be kept small without the management weighing anything.



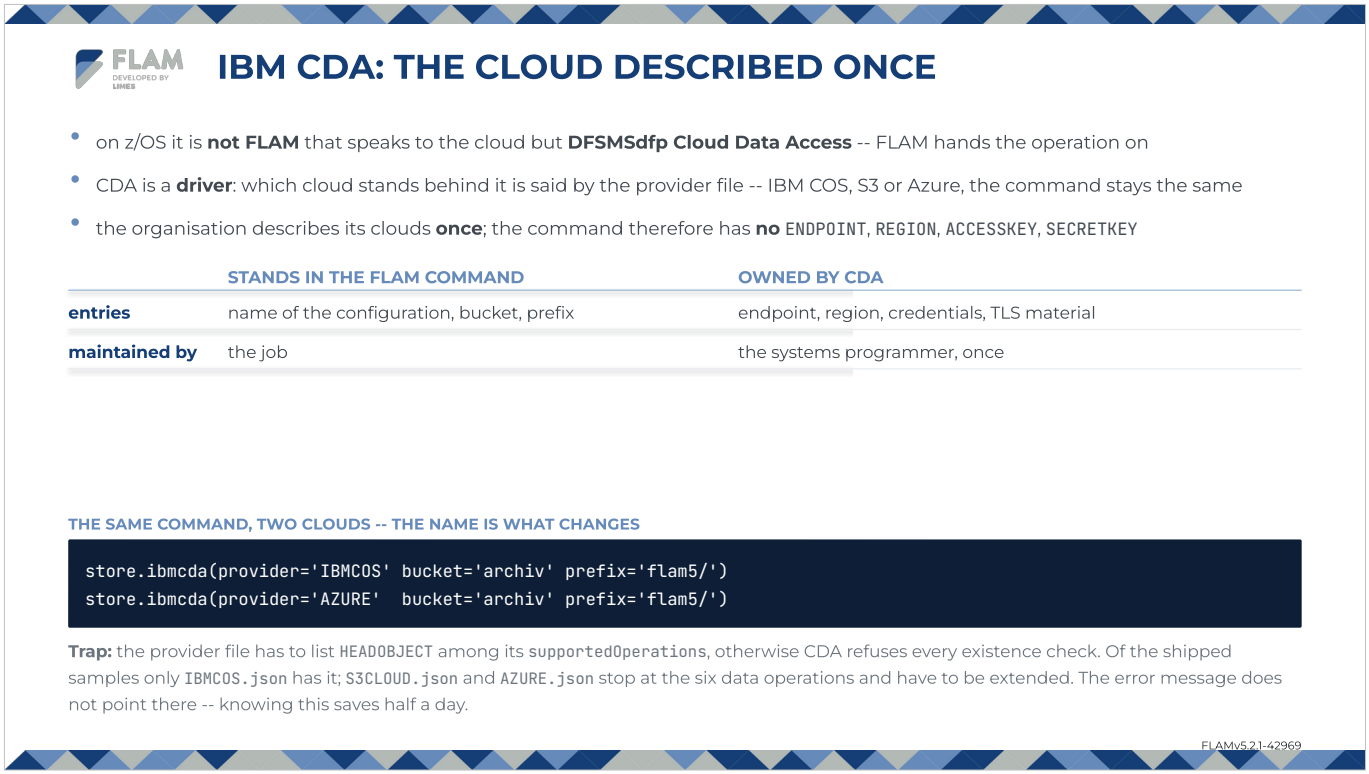
The slide that explains why an archive in the cloud is usable at all. And at the same time it explains an operating rule. The starting position: an archive consists of many segments. A cloud object store is good on **throughput** and bad on **latency** -- every single access costs a network round trip, and you pay for that even for a small object. So anyone who fetches every segment as an object of its own pays the latency **once per segment**. With thousands of segments the archive is unusable -- independently of how wide the line is. That is why the object store combines several segments into one object and always reads or writes **the whole object**. The defaults are in the source code: **4 MiB for data segments, 1 MiB for member segments**, and the main segment becomes an object of its own. With that you pay the latency once per object instead of once per segment. And now the operating rule to pass on:
in the cloud larger buffers pay off, locally they do not. Locally the File format with offsets is faster anyway; a large buffer there only reads data that nobody needs. The sizes are adjustable -- that is a matter of measurement, not a matter of belief. The usage log from the performance unit shows how many accesses really take place. And the arc back to the folder: **the folder is the local mapping of exactly this object store.** The same split into objects, only without a network. That is why you can develop and test a cloud layout locally before you go into the cloud -- and that is the practical advice for anyone starting out.



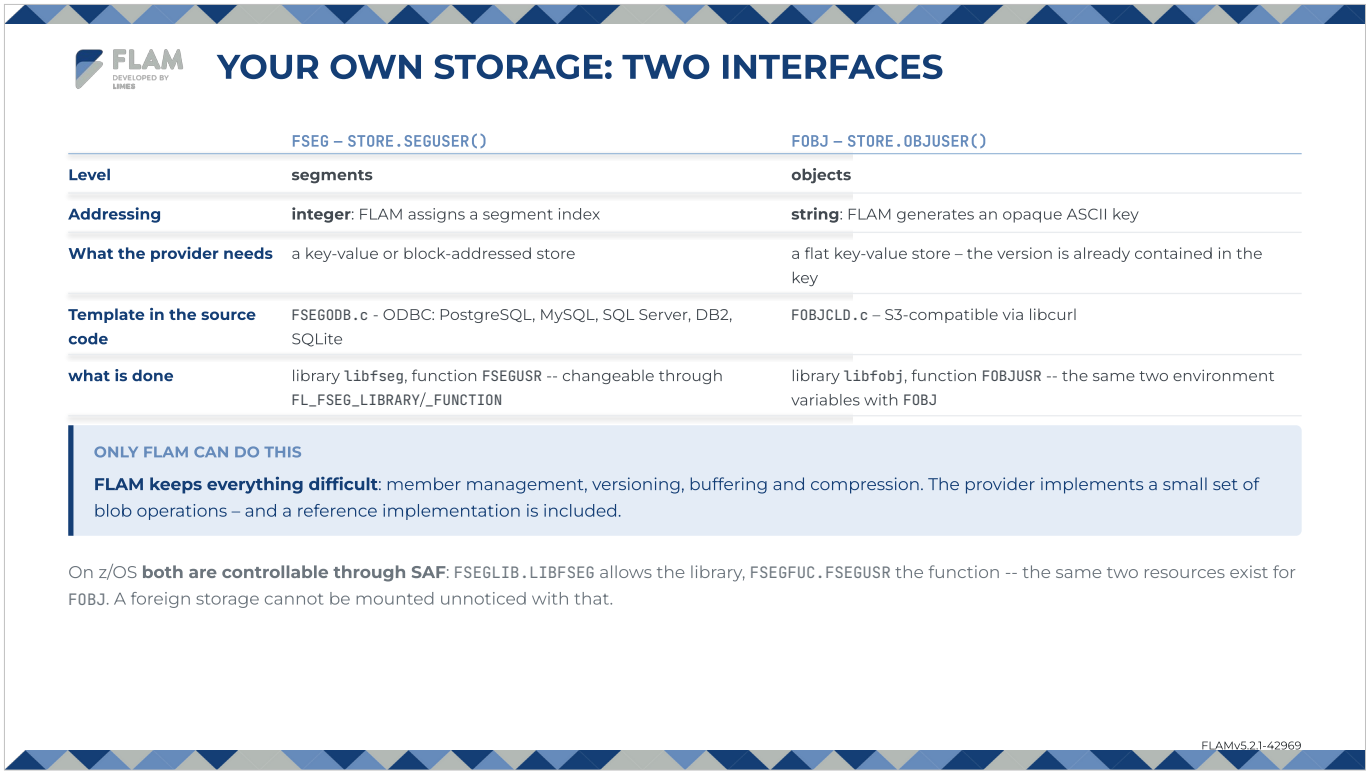
The reference slide. Do not read it out -- go through it and emphasise the two rows that surprise. `FILE` is the flat file. It needs random I/O, so on z/OS USS files are best. And the first surprise: **it also works remotely.** Via `NET.SSH` we read, write and change an archive on a remote system -- with full access, over SFTP. `FOLDER` is the folder variant, one object per file. It too works remotely over SSH. `S3` speaks Amazon S3 **and any S3-compatible service** -- MinIO, Ceph, Wasabi, SeaweedFS.
For that you specify an endpoint of your own. That is the second surprise for many: you do not need an Amazon cloud in order to use S3. Many organisations have an S3-compatible object store in their own data centre.
`AZURE` is Azure Blob Storage. `IBMCDA` is new with 5.2.1 and the z/OS route -- see the next slide for that, because it works differently from the other two. `SEGUSER` and `OBJUSER` are the two SPIs: the store of your own. There is a slide of its own on that too. And on the status: **Google Cloud Storage natively is planned and comes with 5.2.2.** Until then GCS goes over the S3-compatible interface, if the customer activates it. What there is **not**, and why:
the **cluster variant**. Distributing the data over three datasets -- directory, index, data -- brings no advantage over the flat file and makes everything more complicated.



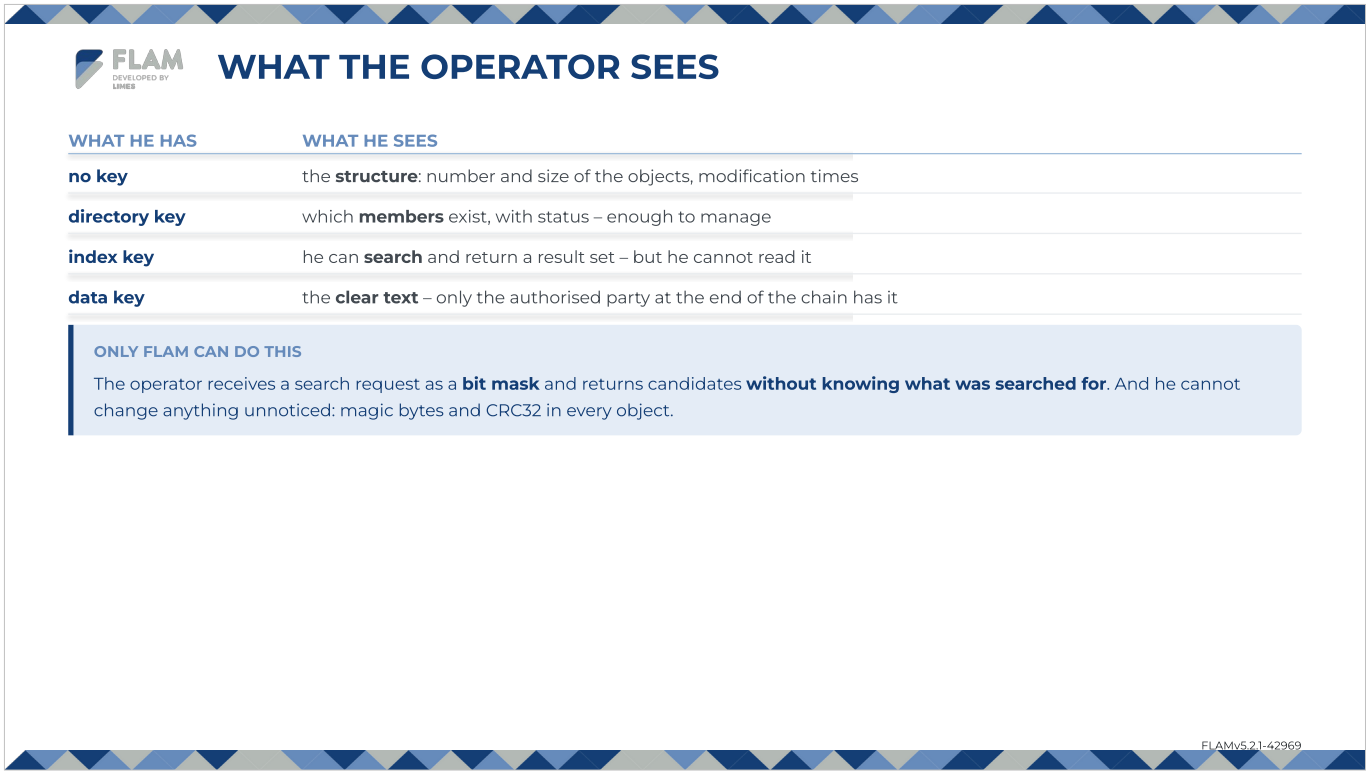
The slide with the calls, and the point about it is the uniformity: **only the `STORE` object changes.** The command, the members, the keys, the search -- everything stays as it is.
That is the practical side of the separation of management and storage. An organisation that starts locally and later goes into the cloud changes one line in its jobs. On the SSH example: the `NET` object belongs inside the `STORE` object, not next to it.
It looks wrong at first and is right -- the network is a property of the storage location. On the S3 example: for Amazon, bucket, region and the keys are enough; the endpoint is derived from them. For an S3-compatible service you specify `ENDPOINT` and switch `PATHSTYLE` on. And a word on authentication, because it is the most frequent follow-up question: there are **five routes**, and they are tried in a fixed order -- AssumeRole, temporary STS credentials, static IAM keys, web identity via an OIDC token, and instance credentials without any key at all.
AssumeRole is recommended, and the reasoning is worth having: with it the long-lived key signs **not a single** object request. It only calls AssumeRole, and the temporary credentials returned sign everything else -- renewed one minute before expiry. Withdrawing access then means changing the role, and not swapping a secret in all jobs.



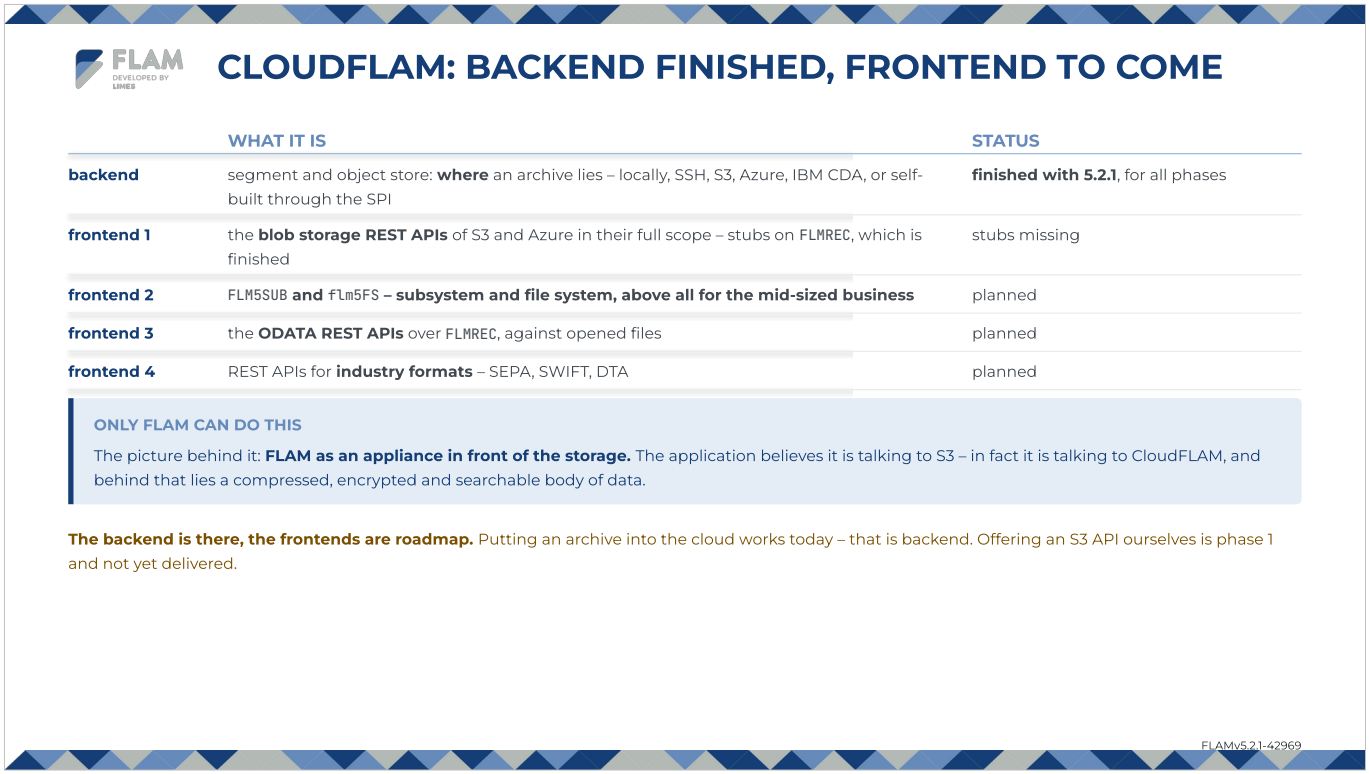
The z/OS slide, and it is different from the two native cloud routes -- hence one of its own. In one sentence: **on z/OS FLAM does not speak to the cloud itself, it hands the operation on to the operating system.** The component in charge is **Cloud Data Access** of DFSMSdfp. It owns the endpoint, the TLS material and the credentials; FLAM names only the name of a configuration and the bucket. Important for the understanding: CDA is not a provider but a **driver**. Which cloud stands behind it is said by the provider file -- for IBM Cloud Object Storage just as for S3 or Azure. The command looks the same in every case, only the name changes. And that is exactly the point: **an organisation that has set up CDA describes its clouds once** -- in a provider file that the systems programmer writes -- and does not describe them again in every FLAM command. That is why `IBMCDA` deliberately has **no** `ENDPOINT`, `REGION`, `ACCESSKEY` or `SECRETKEY`. Those live in the provider file and in the encrypted credential store of CDA. And anyone who specifies them anyway gets an error instead of a value that is silently ignored -- that is intentional. For the cooperation with the systems programmer, a hint that saves a lot of time, because the error message does not point at the cause: **the provider file has to list `HEADOBJECT` among its `supportedOperations`.** FLAM asks via HEAD whether an object exists -- in order to distinguish an object from a directory.
If the operation is missing, CDA rejects the call. Of the examples supplied, only `IBMCOS.json` defines this operation; `S3CLOUD.json` and `AZURE.json` stop at the six data operations. So an S3 provider is best derived from `IBMCOS.json`. New since 5.2.1 and the number that matters for operations: where the installed CDA level offers it, FLAM holds **one session open for the whole archive** instead of logging on again for every single object. That saves a provider file read, a key file read and a TLS handshake per object. An older level answers X'F02', and then every operation stands on its own again -- as before, without anybody having to change anything. So FLAM asks and falls back rather than requiring. It can be switched off with `FL_CDA_SESSION=OFF`; that is a tuning screw, not a feature to talk about. The sentence for the conversation with operations: **we do not bring a second cloud configuration into the organisation.** Anyone who has CDA uses it.



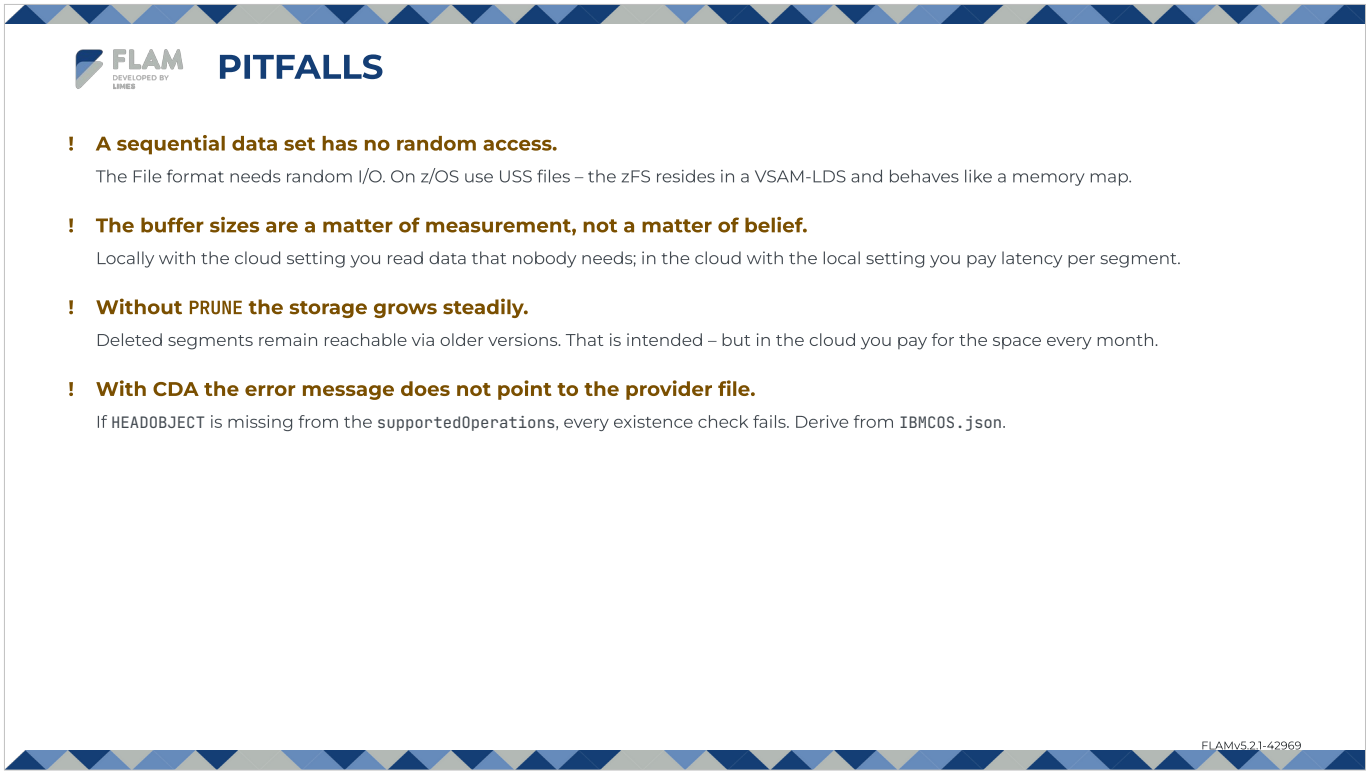
The slide for the organisation that has a store we do not know -- or that does not even exist yet. There are **two** service provider interfaces, and the difference is a single question: **at which level do you start? `FSEG`** starts at the **segment level** and is **integer-addressed**: FLAM assigns a segment index, and the provider maps this index onto a persistent blob.
That fits any key-value or block-addressed store. Activated via `store.seguser()`. **`FOBJ`** starts at the **object level** and is **string-addressed**: FLAM generates an opaque ASCII key for every object and passes it on every call. The provider needs only a flat key-value store. Activated via `store.objuser()`. The point that makes the work small: **FLAM keeps everything difficult.** Member management, versioning, buffering and compression stay with us. The provider implements a **small set of blob operations**.
And with `FOBJ` the versioning is already done before the provider sees it: the version is contained as a prefix in the object key, the provider sees only a flat namespace. To get started there is both: a **reference implementation** (`libfsse`) that wraps the built-in flat file store or folder store respectively, and **templates** in the source code -- `CSRC/SAMPLE/FSEGODB.c` shows an attachment via ODBC to PostgreSQL, MySQL, SQL Server, DB2 or SQLite; `CSRC/SAMPLE/FOBJCLD.c` shows an S3-compatible store via libcurl. The sentence for the room: **anyone who wants a database as the archive medium can have it.** We do not think that is the best idea, but we do not stand in the way -- and the template is included. Two names that are asked about in operations: what is loaded is the library `libfseg` with the function `FSEGUSR`, or `libfobj` with `FOBJUSR`. Both can be changed through `FL_FSEG_LIBRARY`, `FL_FSEG_FUNCTION` and the `FOBJ` twins -- and on z/OS allowed or forbidden through SAF (`FSEGLIB.`, `*FSEGFUC.*`). In between sits a bridge in our code each, which appears in the version tree as `SEGSSE` and `OBJSSE`; whoever debugs a provider sees these names in the messages.



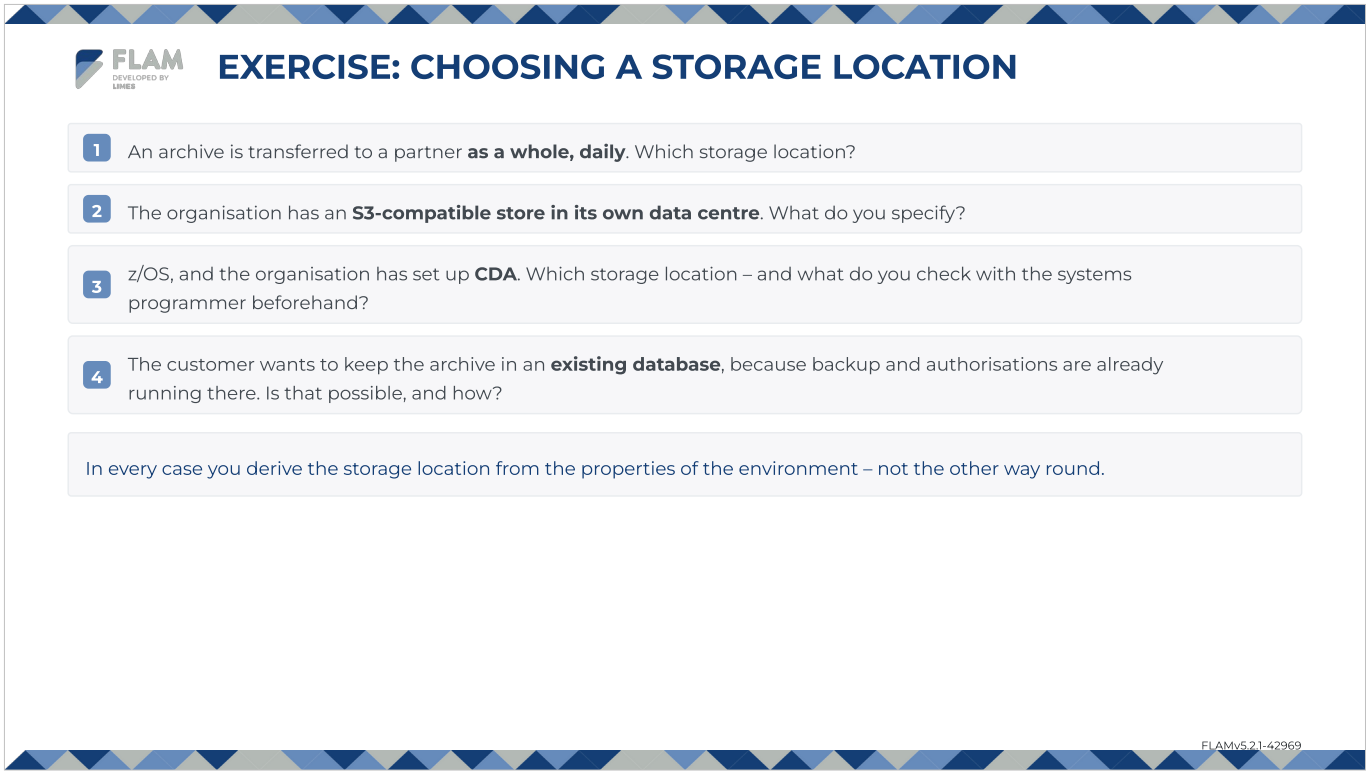
The slide that decides in the cloud conversation, and it brings three threads together: the three keys from the archive unit, the signatures from the indexing unit and the object structure from earlier. What lies with the operator is a set of objects with opaque keys. In them are compressed and encrypted segments. What he can **see** depends on which key he has -- and he usually has none or only the directory key: without a key: the structure. How many objects, how large, when changed. With the directory key: which members exist, with status. He needs that if he is to manage the archive. With the index key: he can **search** -- that is, carry out stage one of the two-stage access and assemble a result set. He cannot read it. Only the data key gives clear text, and only the authorised party at the end of the chain has it. And the piece that makes the statement hard: **the signatures allow no conclusions about the data.** The operator receives a search request as a bit mask and returns candidates without knowing what was searched for. What he **cannot** do, and that is just as important: he cannot change anything unnoticed. Magic bytes and CRC32 in every object, plus the integrity protection of the segments themselves. That is the slide you show to a data protection officer.



The outlook -- and here **two sides have to be kept apart**, otherwise exactly the misunderstanding arises that this slide itself produced before. **The backend is our storage side**: the segment and the object store, that is the question where an archive physically lies. Everything this unit has shown is backend. And that is **finished with 5.2.1** -- for the important cloud storages, more will follow, and through the two SPI a customer can build his own connections. **For all four phases the backend is therefore in place. CloudFLAM is the frontend**: the interface that serves a **foreign application** without it knowing that FLAM is behind it. And it is exactly these frontends that are still missing. **Phase 1: the blob storage REST APIs.** Not "put an archive into S3" -- that is backend and works today. Phase 1 means:
FLAM **itself offers** the REST APIs of S3 and Azure, and in their full scope. An application outside believes it is working against S3, in fact it is CloudFLAM: we compress and encrypt and put the data locally, into the same or into a different cloud storage.
Technically these are stubs on our record interface `FLMREC`, which is finished -- the stubs are missing. **Phase 2: `FLM5SUB` and `flm5FS`.** Subsystem and file system for FLAM5 archives.
The addressee is expressly the **mid-sized business** on client-server:
an unchanged application reads through a DD statement or a mounted directory. **Phase 3: the ODATA REST APIs.** They use `FLMREC` as well and work against **opened files** which are compressed, encrypted and held in a freely chosen storage in the background.
The archive thereby becomes a service that is queried. **Phase 4:**
REST APIs for industry formats **-- SEPA, SWIFT, DTA for the banks.**
That is the last step in the frontend: the application speaks its business format, no longer a storage API. The sentence for managing expectations: **the backend is there, the frontends are roadmap.** Anyone who wants to go into the cloud today can -- as storage.
Anyone expecting an S3 API from us is planning on an announcement.
Saying that difference openly costs nothing and saves a disappointment.



Four, and all three of the first ones have the same root:
the cloud does not behave like a disk. On the first: anyone who puts an archive into a sequential dataset loses random access. On z/OS USS files are the right choice -- the zFS resides physically in a VSAM-LDS, and an LDS behaves like a memory map. On the second: the buffer sizes are a matter of measurement. Anyone who runs the cloud setting locally reads data that nobody needs; anyone who runs the local setting in the cloud pays latency per segment. On the third: plan for `PRUNE`. Deleted segments remain reachable via older versions -- that is intended, but without a cleanup run the storage grows steadily, and in the cloud you pay for it every month. On the fourth: the provider file with CDA. The error message does not point at it.



Twenty minutes. Four cases, and in each of them it is about deriving the storage location from the properties of the environment. On the solutions, once the group is ready:
Case 1 -- an archive that is transferred to a partner as a whole every day: `FILE`. One file, one transfer. The Object format would only be a nuisance here. **Case 2** -- an archive in the organisation's own data centre on an S3-compatible store: `S3` with `ENDPOINT` and `PATHSTYLE`. No Amazon cloud needed. **Case 3** -- z/OS, and the organisation has set up CDA: `IBMCDA`. The systems programmer has the provider file; check whether `HEADOBJECT` is in it. **Case 4** -- the customer wants to keep his archive in an existing database, because his backup and his authorisation concept are already running there: `SEGUSER` via the FSEG SPI, template `FSEGODB.c`. The hint belongs with it: technically it works, but you lose the advantages of an object store and should measure the throughput beforehand.

WHAT TO TAKE AWAY

- two formats: **File** for one file, **Object** for many objects – the same archive, just stored differently
- the Object format has **Root**, **an index per version** and **segment containers**; every object with magic bytes and CRC32
- buffering works against the **latency**: 4 MiB of data segments, 1 MiB of member segments per object, adjustable
- `FILE` and `FOLDER` also work **remotely via SSH**, with full access; `S3` speaks to any S3-compatible service
- `IBMCDA` leaves endpoint and credentials to the **CDA component** – the organisation describes its cloud once
- a **store of your own** can be attached: `FSEG` at segment level, `FOBJ` at object level, with templates in the source code

FLAMv5.21-42969

Go through the learning objectives. The sentence that should stick: the storage location is interchangeable because it is separated from the management -- and in practice that means that an organisation can start locally and later go into the cloud without touching its archives.

FLAM TRAINING, LEVEL 2

One chain instead of a job chain

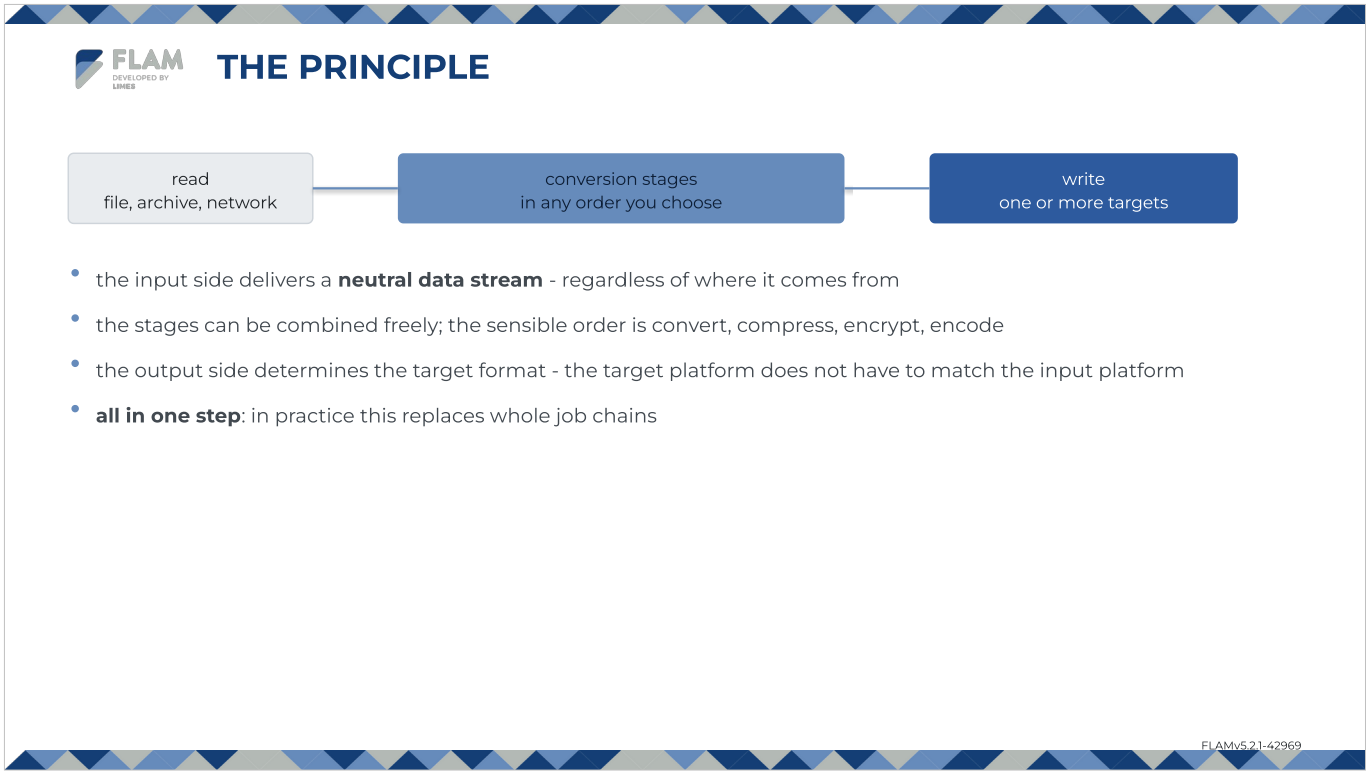
Read, convert, write more than once

Falk Reichbott · limes datentechnik gmbh

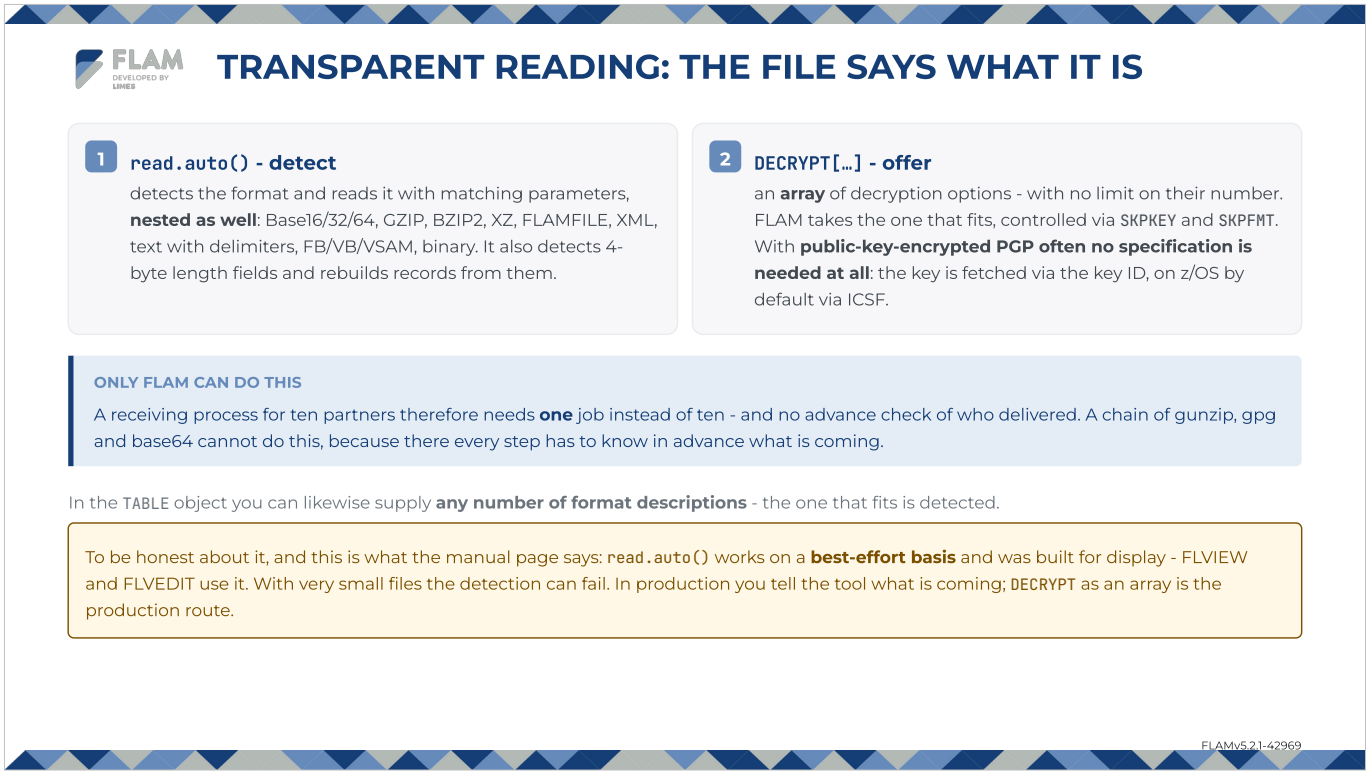
Third functional topic, and the one with the widest use:
most customers earn their money with FLUC, not with the archive. The core is a chain: read, convert in any number of stages, write. And the special part is the multiple outputs -- one run, several targets in several formats.



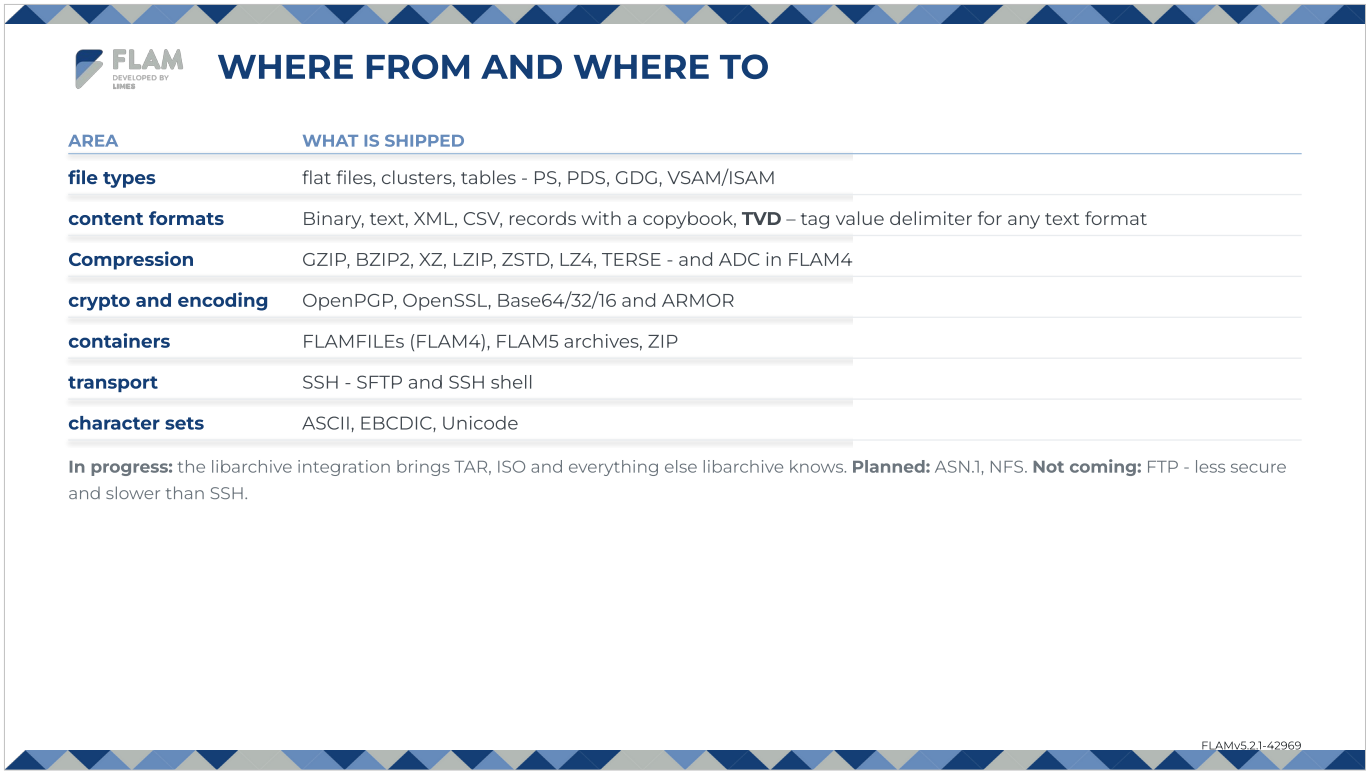
Objective four is the one that replaces the most job chains and therefore delivers the greatest benefit.



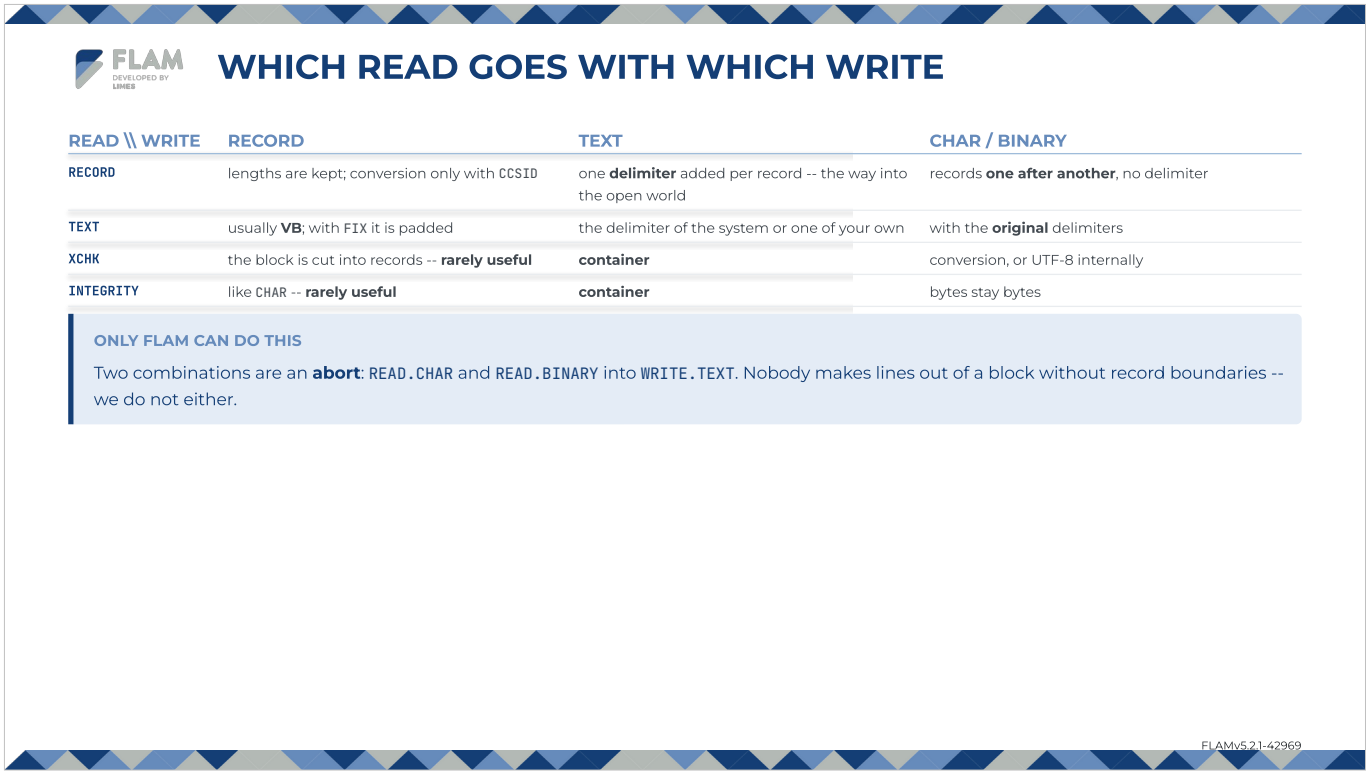
From left to right. The input delivers a neutral stream, then come the conversion stages in any order you like, and at the end there is one or several outputs.
If someone asks about an upper limit: the `CNV` array of the command language has none. In practice it is common sense that limits it, not the tool. The one documented number in this area belongs to something else and comes on the next slide: the decryptions for transparent reading.
Important: the stages are not prescribed. Compressing after encrypting is possible but pointless -- the order is your responsibility, and when in doubt the rule is: convert, compress, encrypt, encode.



This solves a problem that everyone knows who receives data from outside: you do not know exactly beforehand what is in it. The partner says "CSV", and what arrives is a ZIP with a BOM inside.
Two mechanisms, and they are different.
First `read.auto()`. FLAM detects the format and reads it with the matching parameters. And nested, too: a BASE64-encoded, PGP- encrypted, GZIP-compressed FLAMFILE with records inside is unwrapped in one go. Plus text files with delimiters, XML, Base16/32/64, GZIP, BZIP2, XZ, record-oriented datasets FB/VB/VSAM, FLAMFILEs and binary data.
The practical case worth mentioning, because it turns up in support often:
a USS file created with FILEDATA=RECORD and now allocated with FILEDATA=BINARY -- `read.auto()` still reads it correctly, because it recognises the 4-byte length fields and rebuilds the records from them.
Honesty is part of it, and the manual page says exactly that: the detection works on a best-effort basis and you should not rely on it. It was originally built for display -- FLVIEW and FLVEDIT under ISPF use exactly this. With very small files it can fail.
So the rule: `read.auto()` for display, analysis and one-off cases. In production you tell the tool what is coming.
Second, and this is the production case: `DECRYPT` is an **array**. You give several decryption options, and FLAM takes the one that fits.
This is controlled by SKPKEY -- skip if the key does not fit -- and SKPFMT -- skip if the format does not fit. There is no limit on the number.
The case that saves the most work and that is worth mentioning: with public-key-encrypted PGP files, when reading you often need **no DECRYPT entry at all**. The key is pulled automatically via the key ID; you only tell the environment which cryptographic infrastructure FKM5 is to use. On the mainframe that is ICSF by default -- so there you have to specify nothing.
The same principle applies to formats: in the TABLE object you can supply any number of format descriptions, and the one that fits is detected.
Why this counts: a receiving process that accepts files from ten partners needs no ten jobs for it and no upfront check of which partner delivered.



The list is deliberately kept complete so that the participants see the range -- not to be read out. The point:
source and target are orthogonal to the conversion. A file from the host can end up as a member in a ZIP on a remote Linux without anything changing in the conversion.
`read.auto` is the way in for the unknown: FLUC recognises the format itself. Important for credibility: **the table names only what is delivered.** What is in progress is below it and is said to be so. A customer who plans a feature in and then does not find it costs more trust than a shorter list. On the libarchive integration: it brings everything libarchive knows in one go -- TAR and ISO are only the two most often asked about. And on FTP the plain statement, when asked: **FTP is not coming.** It is less secure and slower than SSH, and we have SSH. That is a decision, not a backlog.
One content format deserves a sentence of its own because it is underestimated: **TVD**, tag value delimiter. With it **every** text format can be described that consists of a tag, a value and a separator. It was intended primarily for **SWIFT-MT** -- and that is exactly what it is used for. But it carries more: through TVD we also read and write **JSON**.
For JSON a dedicated, specialised support is additionally planned, along the lines of the one for XML; until then TVD is the way, and it works. So anyone asking about a format that is not in the table should first be asked whether it can be described as tag, value and separator. Mostly it can.



The reference slide that stands in the manual as a matrix and that one has to have seen once -- because two combinations do **not** work and one silently does something else than the customer expects. First the read side in one sentence per object. `READ.RECORD` keeps the record boundaries and converts only when a `CCSID` is there. `READ.TEXT` **throws the record length away**, splits at every usual line delimiter and converts automatically. `READ.CHAR` reads all characters, the delimiters belong to the data, the result is blocks. `READ.BINARY` reads bytes as they are. The line to remember: **`READ.CHAR` or `READ.BINARY` into `WRITE.TEXT` is an error** -- not a silent detour but an abort. The reason is logical: out of a block without record boundaries `WRITE.TEXT` cannot make lines, and it is not supposed to guess. The second line: **block into record is allowed but rarely sensible.** The block is cut into records, by default into variable records with a shorter remainder at the end. Whoever needs that knows it; whoever does it by accident wonders about the record lengths. And the combination one needs most often is the least conspicuous:
`READ.RECORD` into `WRITE.TEXT` adds one delimiter per record -- that is the way from the host into the open world, and the delimiter can be chosen. The other way round (`READ.TEXT` into `WRITE.RECORD`) usually gives variable records; with fixed records it is padded.

SPECIFYING CHARACTER SETS CORRECTLY

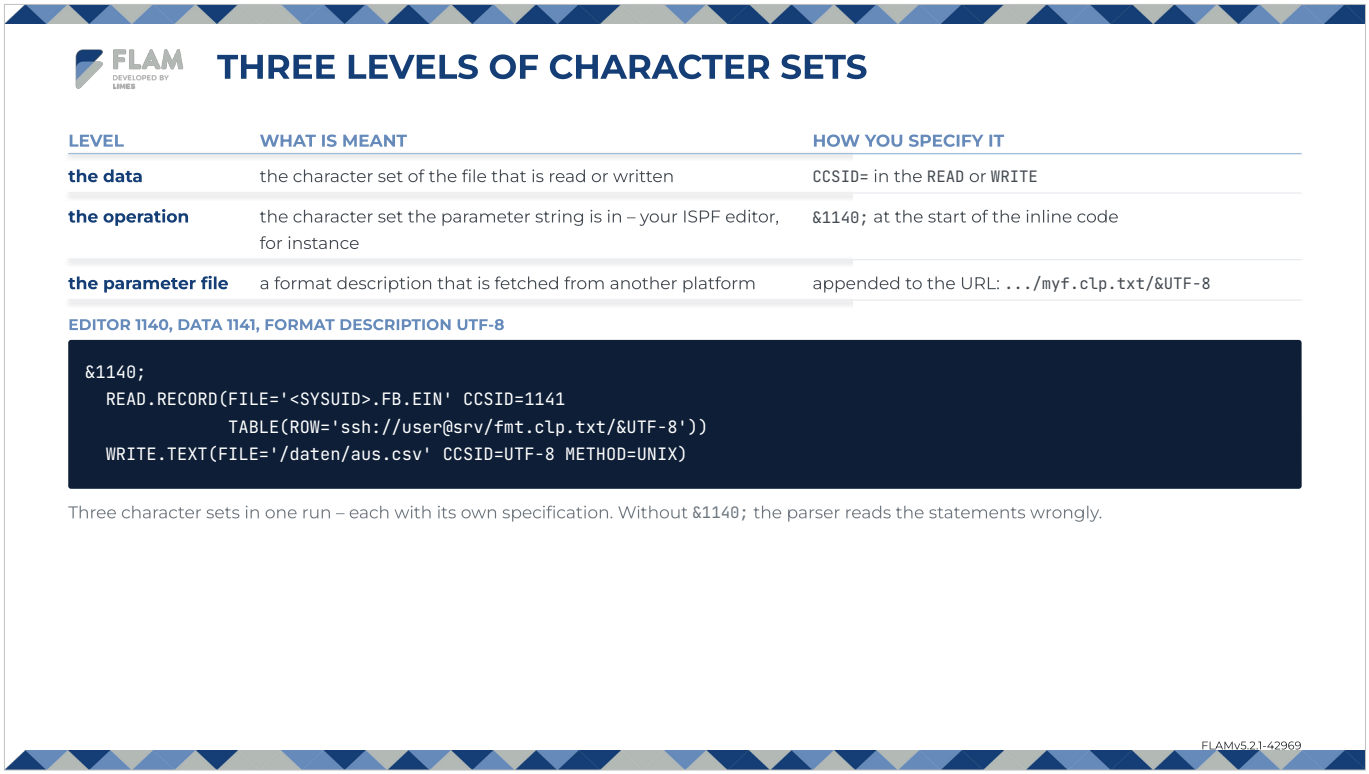
```
read.text(file='ein.txt' ccsid='IBM-1141')
write.text(file='aus.csv' ccsid='UTF-8')
```

- the CCSID describes the **file**, not the platform - a mainframe file stays EBCDIC even if you read it under Linux
- reading and writing each have **their own** CCSID; specifying only one is the most common cause of broken umlauts
- the German classic: `IBM-1141` instead of `IBM-1047` - the difference is precisely the umlauts and the euro sign
- the conversion is done by our own module (`ICV`) - not by libiconv; it handles normalisation, subsets, best-fit mapping, reporting and casing

If after a conversion "only the umlauts" are wrong, it is almost always the CCSID - and almost always the one on the input side.

FLAMv5.21-42969

The most frequent source of error in practice, and it is trivial to avoid:
the CCSID must match the file, not the platform.
The classic case is a mainframe file in 1141 (the German EBCDIC code page with the Euro sign) that is read as 1047 -- then exactly the umlauts are wrong, and it is only noticed in the business department.
Second classic: specifying only one side. Reading and writing each have their own CCSID.
If you say nothing, FLAM does the following -- and it is worth knowing, because it treats the two directions differently:
When writing we always take the local character set of the machine.
When reading we detect. That costs CPU, and with single-byte character sets it is not unambiguous. EBCDIC can be told apart from ASCII and UTF-8 well -- but whether it is German or American EBCDIC is not in the bytes. So we decide that from the **country setting of the machine**: on a system set to German, detected EBCDIC becomes 1141, on one set to American it becomes 1140.
Hence the recommendation for production: **specify the CCSID.** Then nothing has to be detected, it costs no CPU, and the run behaves the same on every machine -- regardless of how someone has set it up.



This slide clears up a mix-up that regularly costs half a day on the mainframe. Character sets do not exist only **in the data**.
There are three levels, and all three can be different:
First the **data** that is processed. Second the character set in which we are **operated** -- that is, the one the parameter string is in.
Third the **parameter files** that are fetched from other platforms.
The example is real and is worth going through slowly: you are sitting on the mainframe writing JCL, but your editor is set to 1140. The data you read is in 1141. And the parameter file that describes the data format sits on a remote Linux server in UTF-8.
Three character sets in one run. Each needs its own specification:
In the JCL inline you write `&1140;` at the start -- with that we read the statements correctly, namely in the character set in which your editor wrote them.
In the URL of the remote parameter file you append the character set: `ssh://user@server/format/myf.clp.txt/&UTF-8`.
In the CONV statement you set the CCSID of the data -- so that nothing has to be detected, and so that the output does not end up in the local character set when it should not.
Anyone who has kept these three levels apart once also understands why "the umlauts are broken" can have five different causes.



This is the function that surprises customers and that delivers the greatest saving.
Multiple writes means: several conversions of the same input -- once CSV in UTF-8 as a PGP file, once EBCDIC in the clear into an FB data set.
Multiple I/O means: the same output to several targets -- the PGP file by SSH to the partner, the same one straight to disk, and additionally as a member in a ZIP for the backup.
Both exist in the batch utility and through the APIs.

AN EXAMPLE FROM PRACTICE

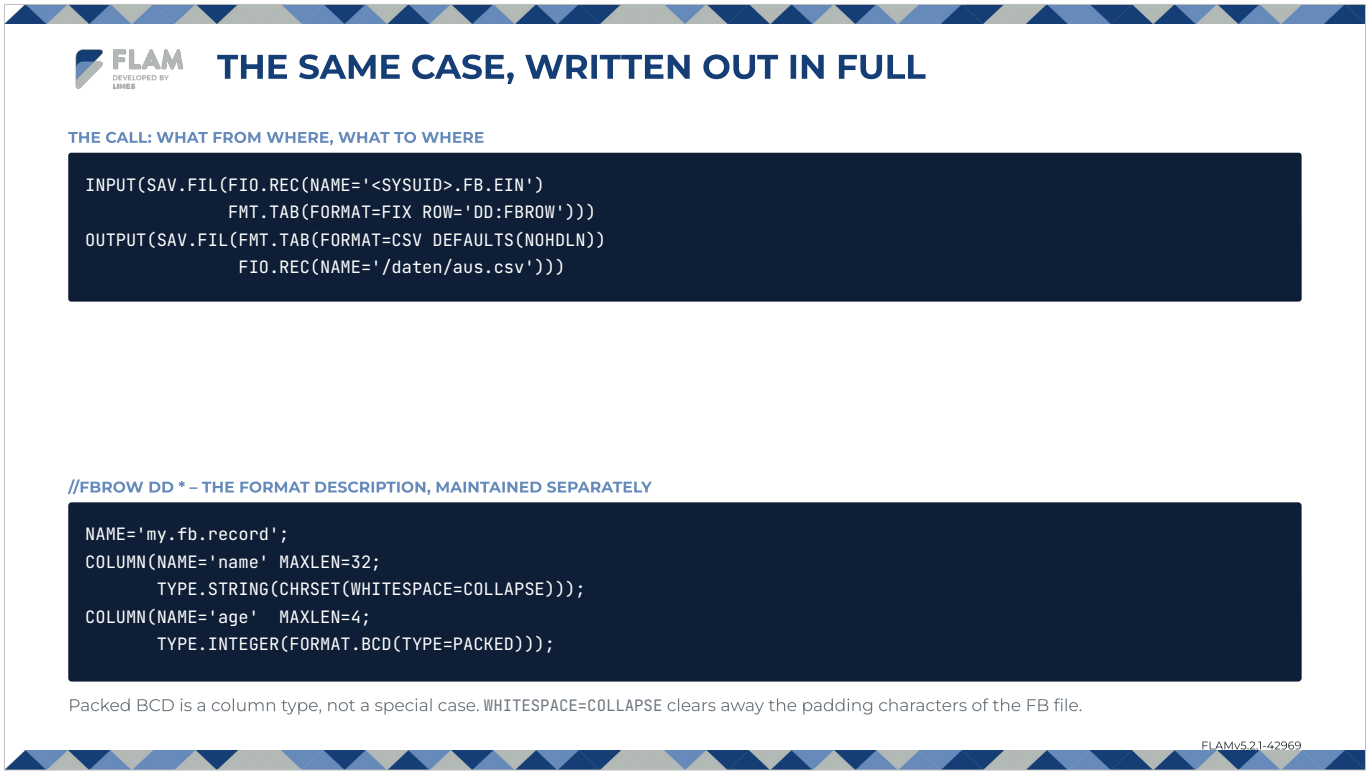
A legacy application writes FB or VB on the mainframe, EBCDIC strings and BCD numbers. It must not be touched.

- table support splits the record into columns and assigns types to them
- the numbers are converted from BCD into their external representation, the strings from EBCDIC to UTF-8
- the output is produced as **CSV with Unix line endings**
- then **OpenPGP-encrypted**, transferred via **SSH** and stored as a **member of a ZIP archive**

One step. And if the route runs through the FLUC subsystem or flucFS, the legacy application notices nothing of it.

FLAMv5.21-42969

The example appears like this in the old slides and is the best selling point, because it describes a whole modernisation in one step.
Starting point: a legacy application writes an FB or VB data set on the mainframe with EBCDIC strings and BCD numbers. Nobody may touch it.
Result: a CSV stream in UTF-8 with Unix line endings, OpenPGP-encrypted, by SSH as a member into a ZIP archive on a remote Linux system.
And the legacy application notices nothing of it when the route goes through the subsystem.



The previous slide describes the case, this one shows it in full. It is example 15 from the manual, and it is deliberately kept small -- two columns, so that you see the structure and not the volume. The task: an FB dataset with two columns becomes a CSV file without a header line. The first column is a name padded with blanks in EBCDIC, up to 32 bytes. The second is a **packed BCD number** over 4 bytes holding the age. For the CSV the blanks in the name are to be collapsed.
The structure to take away: **the call says only what comes from where and goes where -- the format description sits next to it.** The `ROW` object is read via `ROW='DD:FBROW'` from a DD card of its own. That way the same description can be used by several jobs, and the JCL does not change when the format changes.
Two details that usually trigger a question in the room:
`TYPE.INTEGER(FORMAT.BCD(TYPE=PACKED))` -- with that, packed BCD is a column type and not a special case. Nobody has to write a conversion.
`TYPE.STRING(CHRSET(WHITESPACE=COLLAPSE))` -- leading, trailing and repeated blanks drop out. That is the handle needed with FB data almost always, because data is padded with blanks there.
And the character-set question does not even arise here: writing is done in the local character set, that is in EBCDIC, because nothing else is said. If you want UTF-8, say so at the `WRITE`.

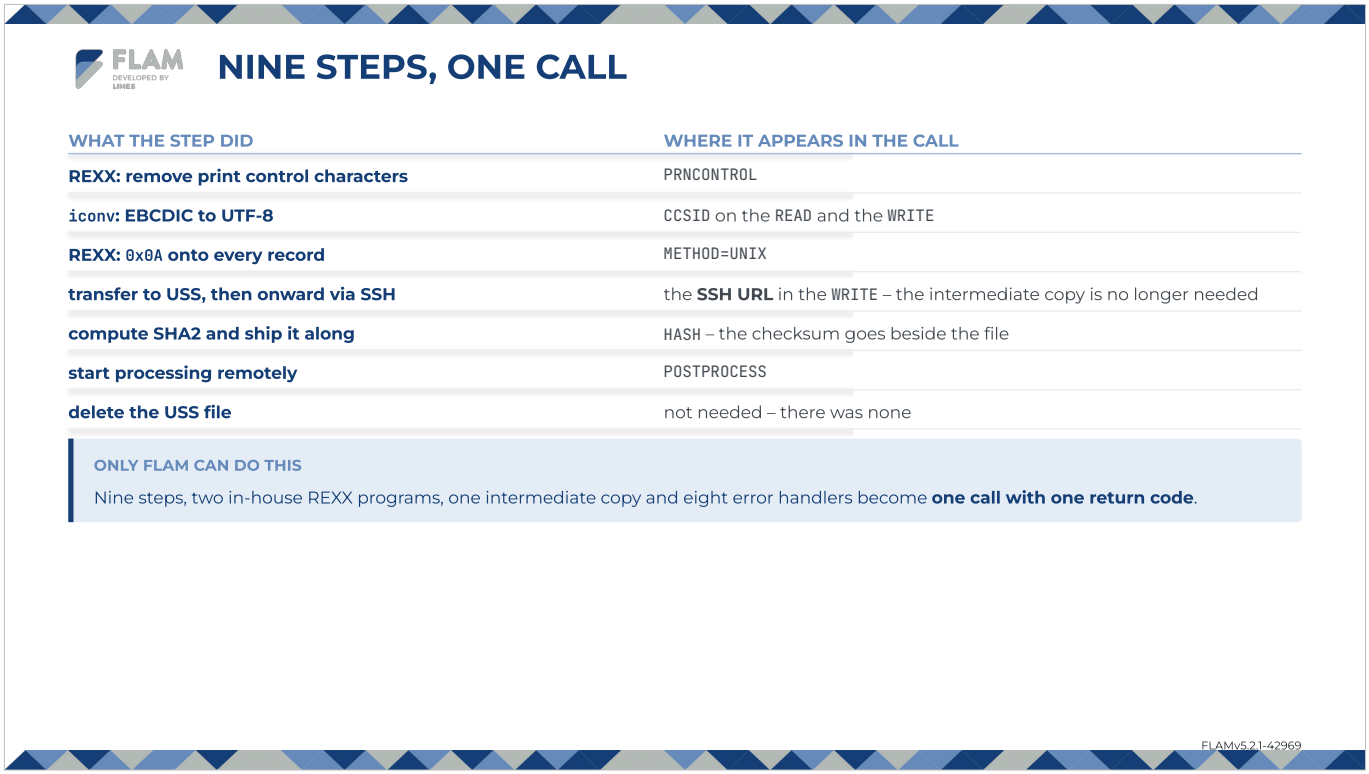
THE SAME THING OVER THE THREE ROUTES

ROUTE	HOW	WHEN
Batch utility	`flcl CONV read...(...) write...(...)`	scheduled processing, job chains
Subsystem, file system	FLUC subsystem on z/OS, flucFS on Linux	when the application must not notice anything
API	byte interface or FLUC-UP in the program	when your program processes the stream itself

For conversion all three routes are equivalent - the choice is purely organisational.

FLAMv5.21-42969

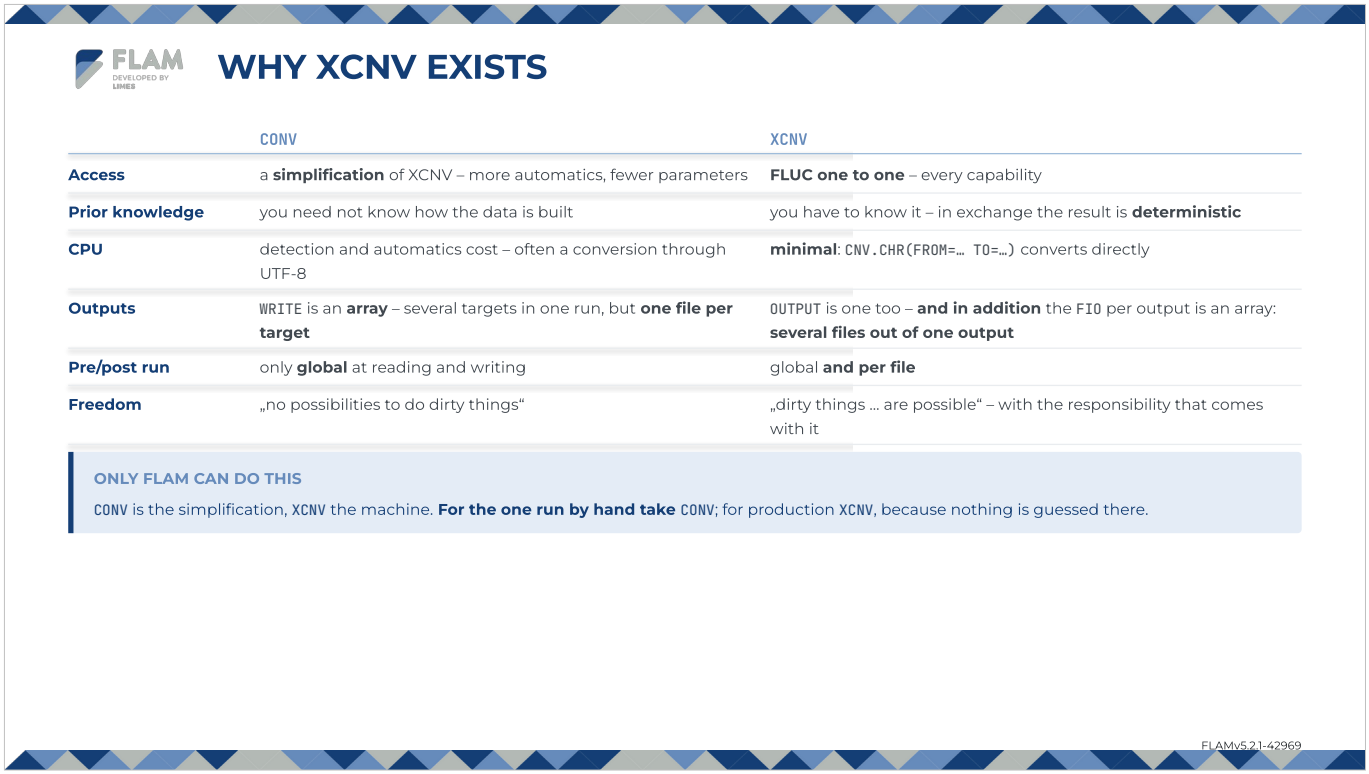
With conversion all three routes are equivalent -- that is the difference from the archive topic, where only the API gives full access.
So the choice here is purely organisational: who is allowed to change what.



This is the slide where someone in the room nods -- because he knows this chain. It is the classic one: an FBA file in EBCDIC is to sit as a text file on a Unix machine, verified.
Read the nine steps out slowly and count along. Every single one is sensible, none is badly built -- they are as they are because each tool can do exactly one thing:
First a REXX that removes or replaces the printer control characters. Then `iconv` from EBCDIC to UTF-8. Then another REXX that appends a `0x0A` to every record. Then transfer the file to USS. Then compute a SHA2 checksum. Then transfer it by SSH to the Unix machine. Then verify the checksum there by SSH. Then start the processing there.
And finally delete the USS file.
Nine steps, two hand-written REXX programs, an intermediate copy in USS and two SSH sessions. And each of the nine can fail on its own, each with its own error handling.
What of that sits in **one** FLCL call: the print preparation is `PRNCONTROL`, the character-set conversion is `CCSID`, the line ending is `METHOD=UNIX`, the transfer is the SSH URL in the `WRITE`, the checksum is `HASH`, and starting the processing is a `POSTPROCESS`. The intermediate copy in USS disappears entirely, because writing is direct.
The point is not the typing. The point is that eight error handlers and one intermediate copy disappear -- and with them the question of which of the nine steps the job got stuck in last night.



The slide before says where each step lands -- this one shows the result. There is no need to read it out, showing three things is enough. First: **there is no FLAM knowledge at the top.** The step is ordinary JCL -- STEPLIB, the input file through a DD, the parameter string in a DD rather than in the PARM, because there it may be of any length. Second, the reading side: `PRNCONTROL=REPLACE` turns the FBA file into **print-prepared records** -- the page break becomes a blank line and the control character disappears from the data. That was the first REXX. `CCSID` says what comes in. The reading side needs no more than that: from 1141 to UTF-8 every character can be mapped, so a substitution rule is not an issue.
Third, the writing side: `SUPTWS` takes the **FB padding** off -- without it every line would be filled up with blanks to the record length. `METHOD=UNIX` puts the `0x0A` on every record, that was the second REXX. `COMP.GZIP()` compresses **before** the transfer; that is not an extra but the normal case -- uncompressed over the line is expensive. The `ssh://` URL replaces the transfer and the intermediate copy in USS. Two details about the URL that get asked about: the `~` stands for the user's home directory. An absolute path would need **two** slashes after the server. And `<cuser>` is a placeholder here for the user id on the other side. Then the checksum: `HASH` computes it and puts it beside the file over the same connection. `MEMBER` is the entry that is easily overlooked -- it determines which **file name stands in the checksum line**.
Without it the name of the target stands there, that is the whole URL, and a `sha256sum -c` on the other side then finds nothing. With a name relative to the home directory it fits, because that is where the command runs. Finally the two `POSTPROCESS` on the other side, only on success: verify the checksum, then unpack. It is an array, they run in the order given. And the command runs **without a shell** -- `cd /daten && ...` therefore does not work; whoever needs a chain calls a script. One note for those rebuilding it on 5.2.1:
the checksum file was written there only after the post-processing, so the `sha256sum -c` in the same call found it empty. From 5.2.2 it is flushed out beforehand. And the semicolon at the end of every line is deliberate: it is the comment character of the command language. With `NUMBER=ON` in the ISPF editor the line numbers land in columns 73 to 80 -- behind the semicolon they are a comment, and the job runs all the same. Without the semicolon the same job stops with a syntax error in column 73 (run through).



The slide that answers the question which comes after the job step: **why are there two conversion commands?** The answer is a ranking, not a choice: **`CONV` is a simplification of `XCNV`.** `XCNV` makes FLUC available one to one to the outside -- every capability the conversion machine has is reachable there. `CONV` lays a comfortable layer over it and takes decisions for you.
What the manual contrasts in section 3.14, and it is worth reading out: `CONV` is "simple to use, more automatisms, fewer parameters", the user "must not know how the data is formatted or where it came from", the conversion runs "on a best-fit basis" -- and: "there are no possibilities to do dirty things". With `XCNV` it says instead "gives access to all capabilities of FLUC", "the user must know how the input data is formatted", "conversions are limited to a minimum of CPU usage" and "dirty things ... are possible". Three things follow from that for production. **First:** determinism. **`XCNV` has fewer automatisms, and every single one has to be switched on explicitly. With that the result does not depend on what was recognisable in today's file. That is exactly what one wants in a night job.** Second: CPU. **The automatics of `CONV` like to convert through an intermediate step -- from X to UTF-8 and then to Y. In `XCNV` one writes `CNV.CHR(FROM=1141 TO='UTF-8')` in the place where it belongs and saves a conversion. The manual states it as a property: minimal CPU consumption.** Third: capabilities that exist only there. **And here one has to be precise:** several outputs `CONV` can do too **-- `WRITE` is an array, which corresponds to the several `OUTPUT` of `XCNV`. The difference lies one level deeper: with `XCNV`** the `FIO` per output is an array too**. One output can therefore write into several files at once -- with `CONV` that is not reachable, there every `WRITE` has exactly one file. On top of that comes** pre- and post-processing per file **instead of only globally. Whoever needs that needs `XCNV`. And the price that has to be named: you have to know what you are doing. "Dirty things" are possible, and the parameter names are the internal ones -- `PRNCONTROL` becomes `PRNCTR`, because that is the name inside the machine.**



The same task as two slides before, only in `XCNV` -- and the two slides should be shown side by side, because then one sees the difference instead of explaining it. The structure is that of FLUC itself, and there is a mnemonic for it worth pinning to the wall: **FLUC is `FIO` `CNV`…`CNV` `FMT` -- `FMT` `CNV`…`CNV` `FIO`.** When reading, the file comes first, then the conversions, then the formatter that makes elements out of it; when writing the same chain runs backwards. The stages stand in `SAVE.FILE` in exactly that order -- `PRE`/`POST` come on top for the environment. Reading: `FIO.REC` with `PRNCTR=REPLACE` -- the same handle as in the `CONV`, only with the internal name. Then `CNV.CHR(FROM=1141 TO='UTF-8')`: **one** conversion, explicitly stated, no detour. And at the end stands `#FMT.BIN()#` -- commented out, because `BIN` is the standard formatter of the input. It stands there all the same so that the chain can be seen complete; `#…#` is the comment of the command language. Writing, from top to bottom in the order of the processing:
`FMT.TXT(METHOD=UNIX SUPTWS)` makes text out of the elements with `0x0A` as the separator and takes the FB padding away -- formatting belongs in the `FMT` stage, not in a conversion. `CNV.GZP()` compresses. `CNV.HSH()` computes the checksum and puts it beside it over the same connection. `FIO.BLK` writes the file through `ssh://`. And `POST` runs the `sha256sum -c` on the far side. The gain over the `CONV` version is not brevity -- this version is longer. The gain is that **every stage stands there**:
no detection, no default that takes effect differently tomorrow.
Whoever needs a second recipient appends a second `OUTPUT`, and the read side still runs only once.

PITFALLS

! **Only one CCSID specified.**

Reading and writing each have their own. If one is missing, a default applies - and it rarely fits.

! **Compressing after encrypting.**

Technically permitted, pointless in practice: encrypted data is no longer compressible. Check the order.

! **Text mode for binary data.**

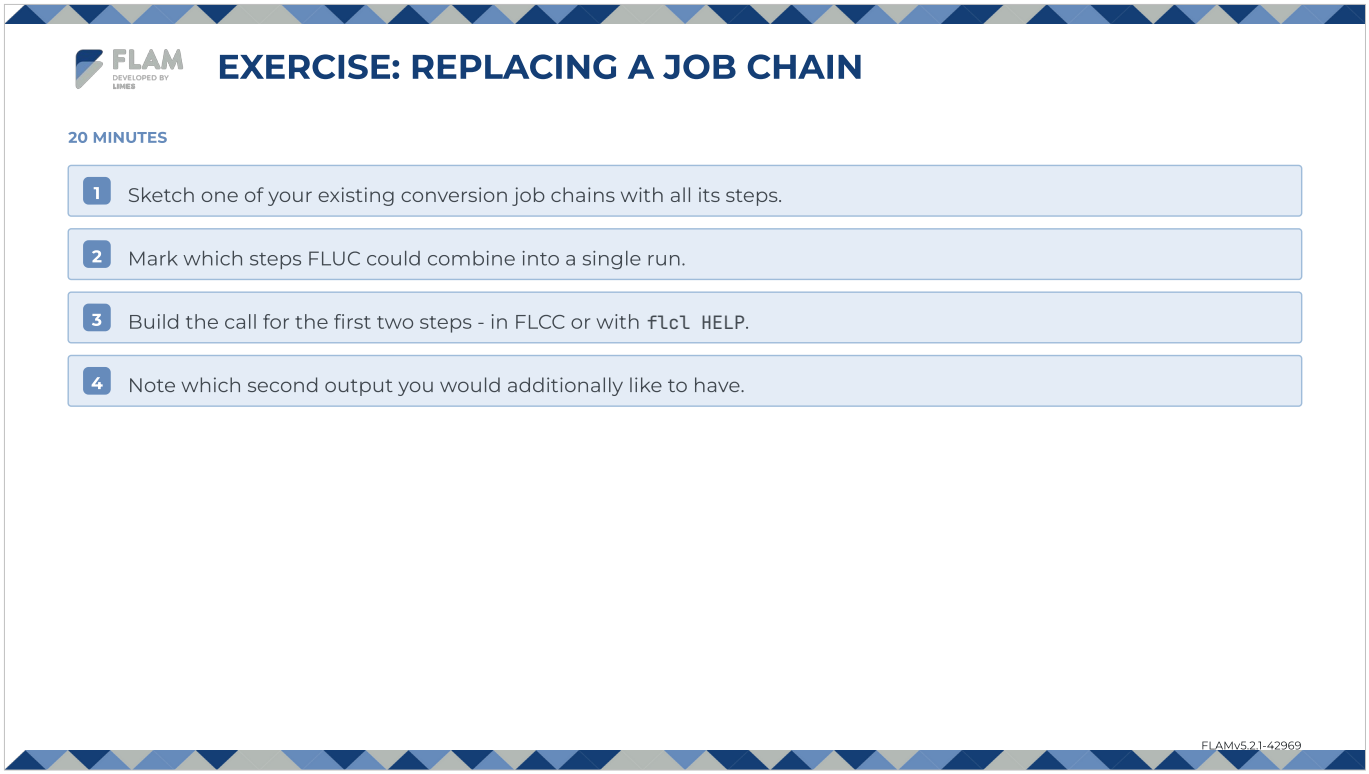
`read.text` interprets line endings. For binary data use `read.binary` - otherwise bytes are lost.

! **Several outputs, but one error path.**

With Multiple Writes it has to be clear what applies if one target fails. That belongs in the chain logic, not in an assumption.

FLAMv5.21-42969

All four from conversion support cases.

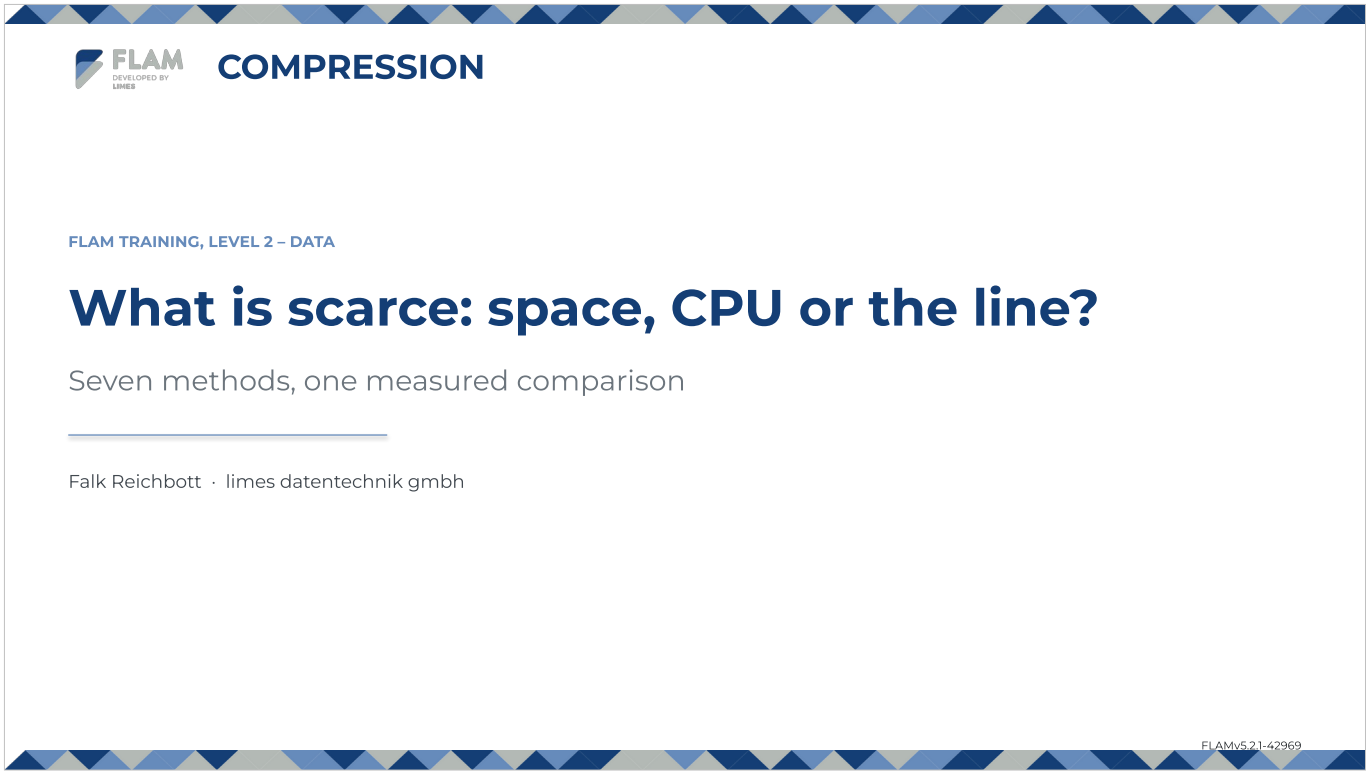


Twenty minutes. The goal is not the perfect chain but the insight into how many steps collapse into one.
Solution:
1. The input is a `READ` object matching the source: `read.record` for datasets, `read.text` for text files, `read.binary` for binary data.
2. The stages in the sensible order: convert, compress, encrypt, encode. Whoever compresses after encrypting compresses randomness -- that costs CPU and gains nothing.
3. The output determines the target format, and the target platform need not match the source platform.
4. If two targets are needed: two `WRITE`, not two runs. Reading and the shared conversion then happen only once.
The expected insight: the old chain had one step per tool, the new one has one step for everything -- and the most expensive item of the old chain was not the CPU time but the I/O for the temporary files.

WHAT TO TAKE AWAY

- read, convert, write - any number of stages, all in one step
- source and target are orthogonal to the conversion
- the CCSID describes the file, not the platform - and there are two of them
- Multiple Writes and Multiple I/O: read once, output several times
- in practice this replaces whole job chains - that is where the benefit lies

Walk through the learning objectives. The next unit goes one level deeper: the table support that turns a record into columns.



The unit in which a number is worth more than an opinion. The arc:
the seven methods, the levels, a measured comparison, what a header stores and what is lost without one, TERSE for the host world, and the three cases in which you do **not** compress. The sentence for the start: the question is never "which is the best one" but **"what is scarce here"** -- space, CPU time or the line. For each of the three answers there is a different method, and the difference is large: between the fastest and the strongest our measurement has **a factor of 90 in CPU time** and a factor of 2.5 in size.

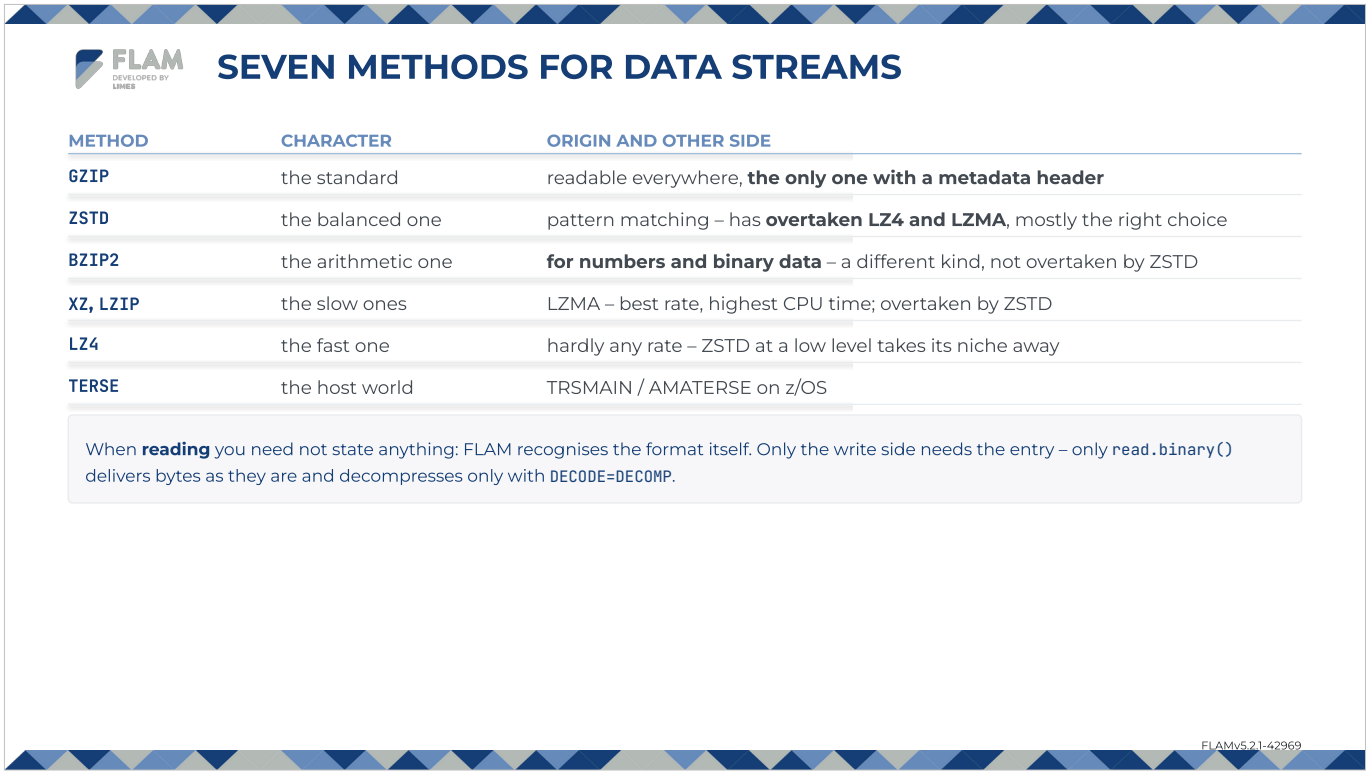
LEARNING OBJECTIVES

After this unit you can

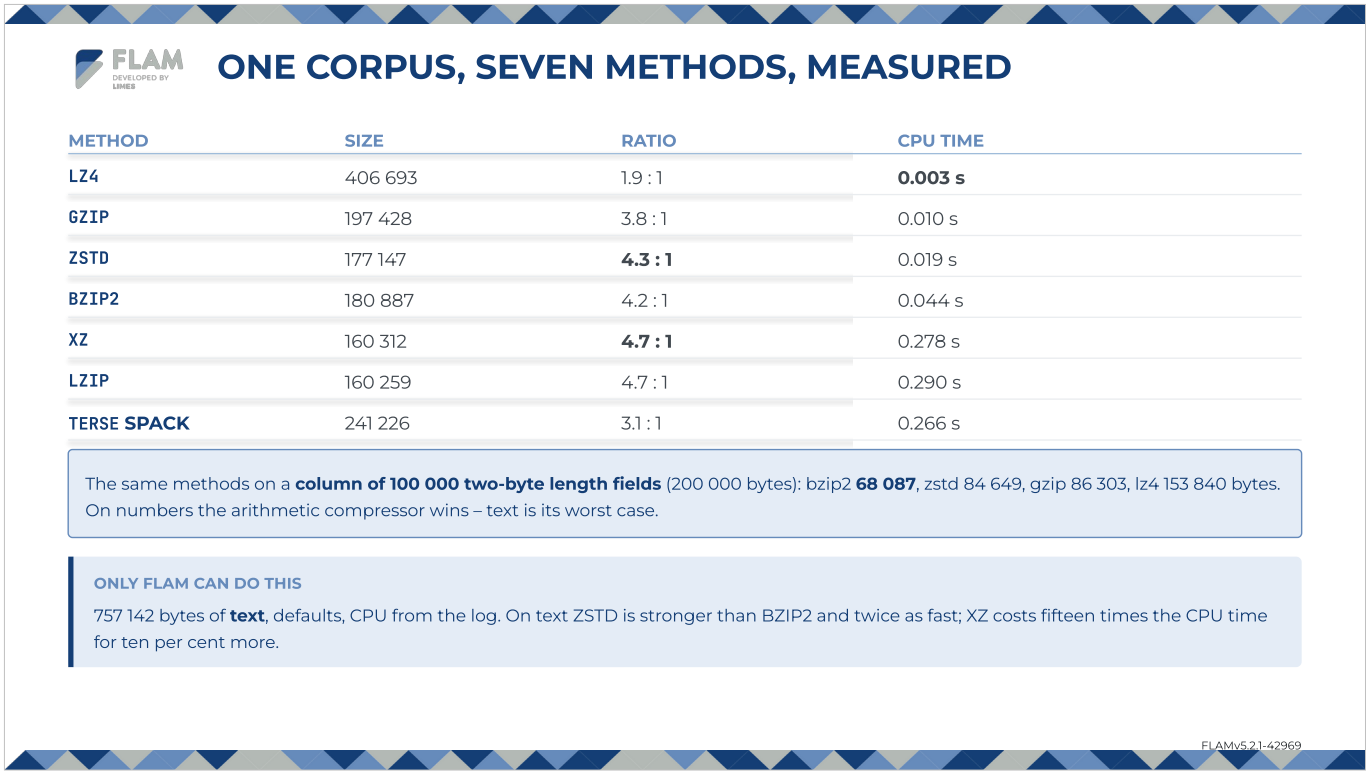
1. name the **seven methods** and place where they come from
2. choose one **with reasons** – by space, CPU or the line
3. say what a **GZIP header** takes along and what is lost without one
4. place **TERSE** – and when it is needed
5. name the three cases in which you do **not** compress
6. explain why compression loses the **record boundaries**

FLAMv5.21-42969

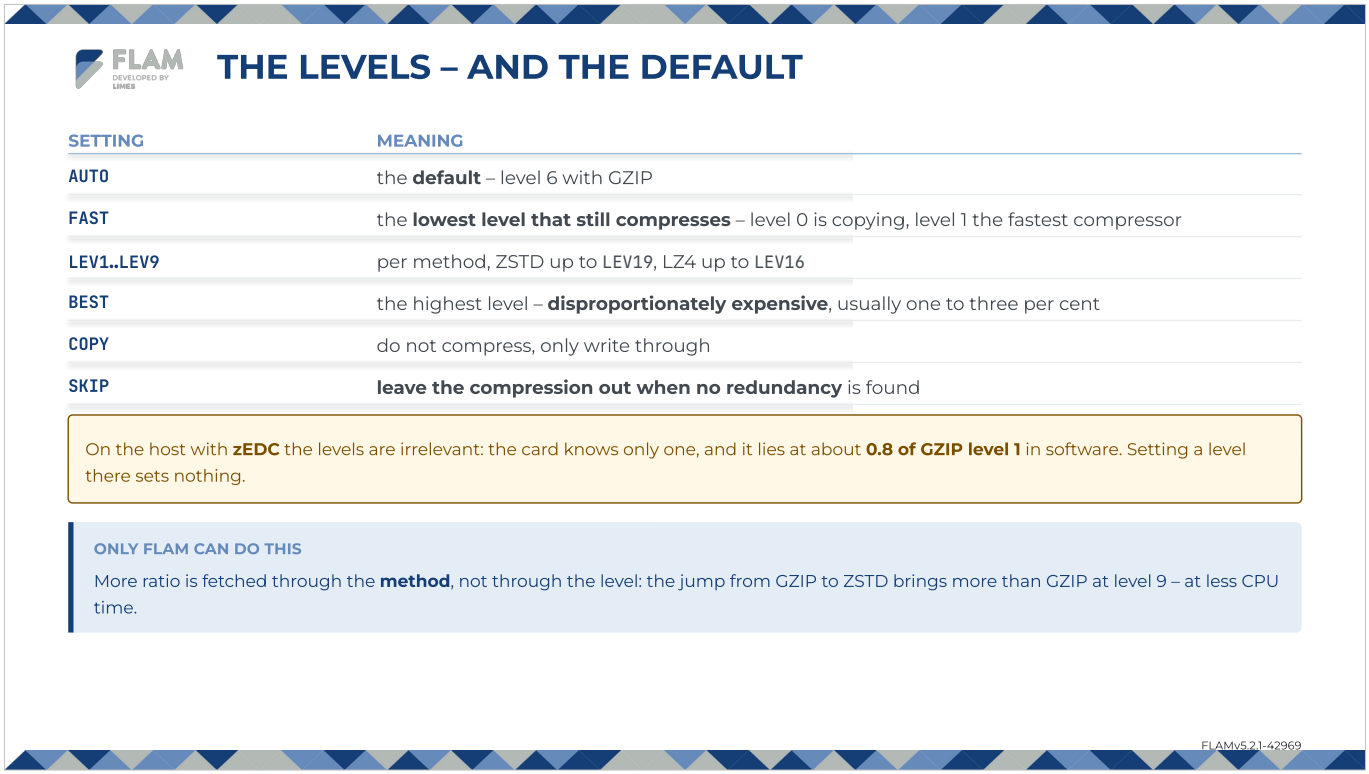
The second objective is the reason the unit exists -- the answer hangs on the question of what is scarce, and hardly anybody asks it. The fifth is the point where money is lost: whoever compresses already compressed data again pays CPU for more bytes.



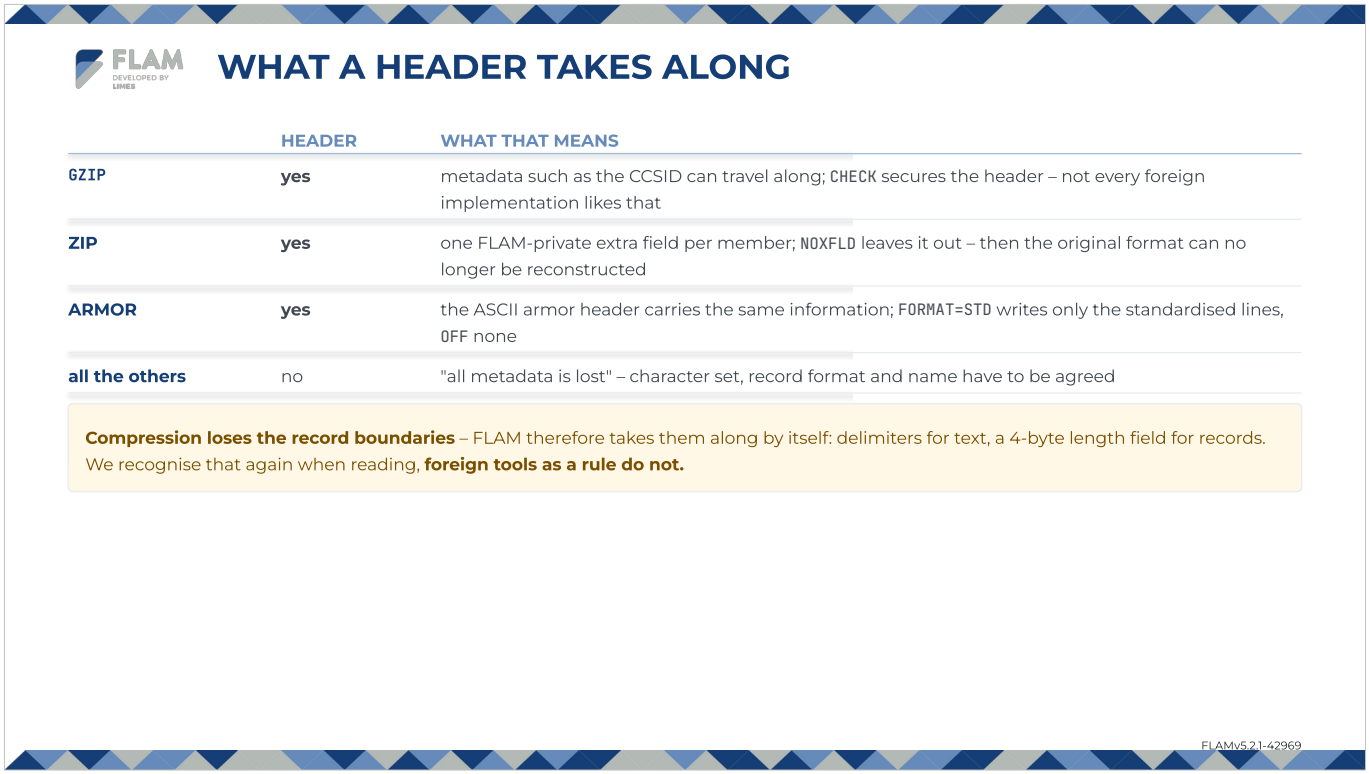
The stocktaking. The column on the right is the important one -- it says where the method comes from and therefore who can read it on the other side. `GZIP` is the standard for the exchange: every system can do it, and it is the only one with a **header for metadata**. `ZSTD` is the new one and in practice mostly the right choice: almost the rate of XZ at a fifteenth of the CPU time. **It has overtaken LZ4 and LZMA** -- at a low level it is as fast as LZ4 and better, at a high one as good as LZMA and needs considerably less memory and CPU for it. `BZIP2` is the special case, and it has to be placed correctly: it is **not a pattern matcher but an arithmetic compressor** -- the best one known. So it is **not** overtaken by ZSTD but often the better choice for **numbers and binary data**. On text, the corpus of the next slide, it is the worst of its class. `XZ` and `LZIP` are built on LZMA, equally strong in our measurement and by far the slowest -- overtaken by ZSTD today. `LZ4` is the fast one. It hardly compresses, and ZSTD at a low level takes this niche away from it too. For completeness, if somebody asks about older methods: the tree-based ones with a thesaurus -- ADC, LZW -- likewise find only repetitions, but need far longer when decoding, devour memory, and building the thesaurus is slow. They are inferior to the pattern matchers and therefore superseded; the arithmetic branch is not. And `TERSE` is the host world: the format of TRSMAIN and AMATERSE, with which data sets have been sent on z/OS for decades. The hint that should be given with it:
all seven are **transparent when reading**. You do not have to say what it is when reading -- FLAM recognises it. That is in the conversion unit. One exception belongs with it, because it otherwise produces a support call: **`read.binary()` does not decompress by itself.** There `DECODE=NONE` is the default, and rightly so -- whoever reads binary wants the bytes as they are.
With `DECODE=DECOMP` (or `ALWAYS`) `read.binary()` unpacks as well. `read.text()`, `read.record()` and `read.auto()` do it by themselves; measured on a ZSTD file.



The slide that carries the unit. Do not read it out -- show the three groups and then name the two numbers worth keeping. The corpus: our manual as a text file, 757 142 bytes. One run per method, each in its default, the CPU time out of the log of `flcl`. The three groups read from top to bottom: **LZ4** is in a class of its own: 1.9 to 1 -- but in three milliseconds.
Whoever has a slow line and a fast CPU takes something else; whoever has a fast line and scarce CPU takes this. **GZIP, ZSTD and BZIP2** are the middle: **ZSTD delivers 4.3 to 1 in 19 milliseconds**, BZIP2 only 4.2 to 1 in 44 -- on **text** ZSTD is therefore better and twice as fast. And here belongs the sentence that saves the slide from a wrong conclusion: **that holds for this corpus.** Text is the worst case for an arithmetic compressor. Take a column of numbers instead -- 100 000 two-byte length fields -- and the picture turns around:
bzip2 68 087 bytes, zstd 84 649, gzip 86 303, lz4 153 840. BZIP2 wins there by about 20 per cent. That is exactly why a method of its own can be chosen per stream in the pot design. **XZ and LZIP** fetch 4.7 to 1 but need 278 and 290 milliseconds for it.
That is **fifteen times** the CPU time of ZSTD for ten per cent more ratio. And `TERSE` stands beside them on purpose: PACK and SPACK are weaker than everything else and still right when the other side expects AMATERSE. Not in the table is **ADC**, the classic FLAMFILE method, and that has a reason: the C implementation is not comparable with the assembler version on the host in its CPU consumption. A number from here would be misleading. The warning that belongs with it: **this is one corpus.** Text compresses differently from numbers, and a payment file differently from a manual. The order of the methods stays, the numbers do not -- whoever wants to know exactly measures their own data. That costs an afternoon.



The slide that stops somebody from writing `BEST` blindly.
Every method has levels, and the spellings are deliberately the same: a number, `FAST`, `LEV1` to `LEV9` (with ZSTD up to `LEV19`, with LZ4 up to `LEV16`), `BEST` and `AUTO`. `AUTO` is the default and with GZIP that is level 6. Two numbers about it, because they mean the same everywhere: **level 0 is as a rule the copy** -- not compressing -- and **level 1 the fastest compressor**. `FAST` is exactly that: the lowest level that still compresses at all. What has to be known about it: the levels cost **disproportionately**. Going from the default to `BEST` doubles or triples the CPU time and often brings one to three per cent. Whoever needs more ratio is better off changing the method than the level -- the jump from GZIP to ZSTD brings more than GZIP at level 9. And then the switch that is clever: **`SKIP` leaves the compression out when no redundancy is found.** With it an already compressed JPG is not inflated without having to know it beforehand. With GZIP it goes further: if the data is marked as free of redundancy and no level is set, GZIP takes level 0 by itself, that is copying. And for the host the statement that ends every consideration of levels there: **with zEDC there is only one level.** The card is designed to reach GZIP level 1 and actually lies at about 0.8 of that. Whoever sets a level on a system with an active zEDC sets nothing -- the compressed output is the same. `COPY` is the explicit form of it -- no compressing, only writing through. That is needed when the chain is to stay but the computing time is not.



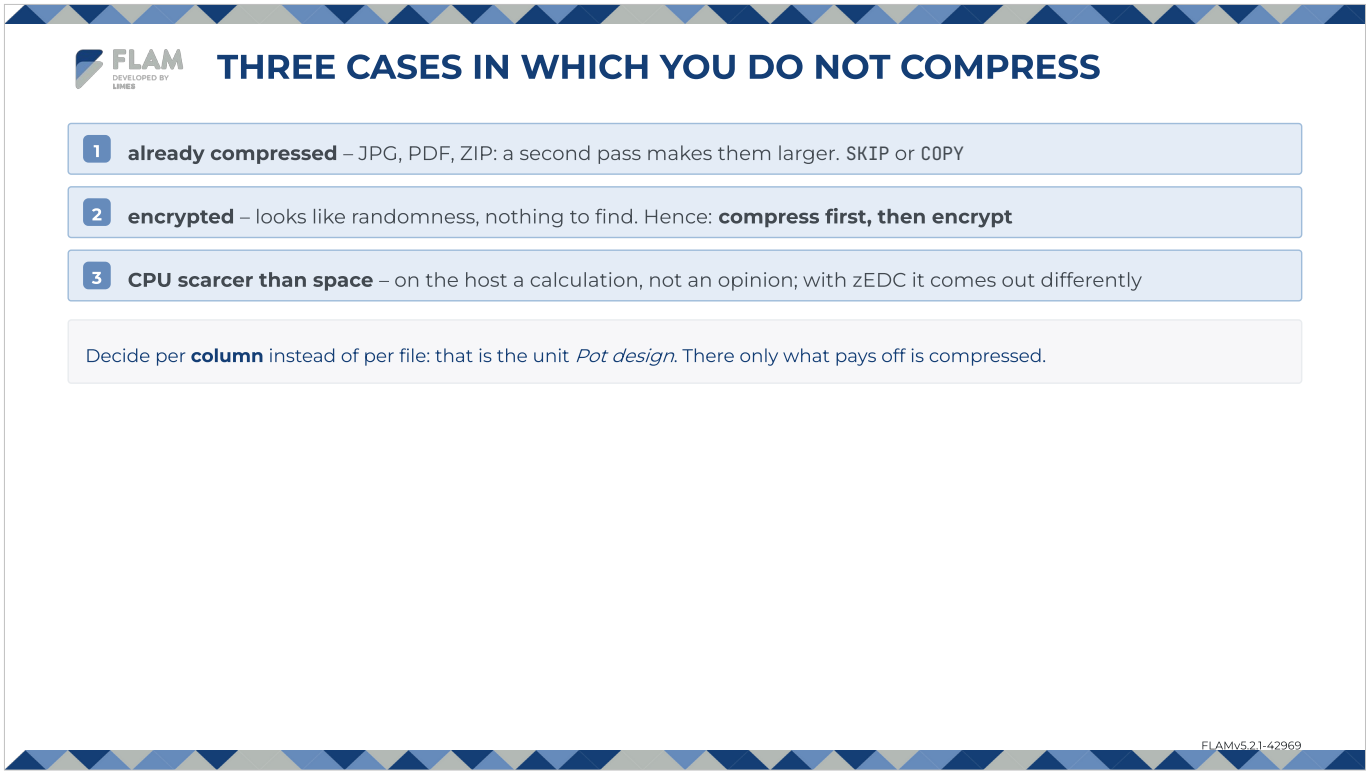
The slide that rescues a data exchange -- and that has something to say only about GZIP. The formats other than GZIP have **no header for metadata**. Our manual pages say so: "there is no header to store meta information about the data, so all metadata is lost".
What is lost is exactly what the other side has to know: the character set, the CCSID, the record format, the original name.
GZIP has a header, and FLAM can put meta information into it. The switch for it is `CHECK`: it computes a checksum over the header.
That is not the default, because **not every GZIP implementation** can deal with it -- whoever sets the switch has to know that FLAM is on the other side. The gain when both sides use FLAM: the CCSID travels along. That does away with the agreement that otherwise runs by e-mail and is wrong at some point. The same holds for **ZIP** and for the **ASCII armor**: there too we write along what a host needs -- record format, record length, character set. With ZIP that is a **private extra field** per member, with the armor it is additional header lines. And here the warning that comes out of customer cases: **some foreign tools warn about them** instead of ignoring them, as would be right. That is why the extensions can be switched off -- with ZIP through `NOXFLD`, with the armor through `FORMAT=STD` or `OFF`. The price stands in the manual page: without the extra field FLAM can no longer reconstruct the original format.
And the second point of this slide, which holds for every method:
compression loses the record boundaries. The result is a block of bytes. FLAM therefore takes the record boundaries along by itself: for text as a delimiter, for records as a **4-byte length field in front of every record** -- that is how it stands in `man_compress_gzip.txt`, and with `write.record()` all file attributes additionally travel into the GZIP header. When reading we recognise that automatically again. **Most foreign tools do not recognise the length fields** and can then do nothing with the data -- that is the most frequent beginner's mistake on the way from the host to Unix.



A short slide for the host -- the accelerator unit has the details, here stands the one statement you have to know when compressing. If a zEDC card is available on the machine, FLAM uses it for GZIP and saves CPU time with it. That is the normal case on a modern z system, and it is the reason why GZIP is often the right choice on the host although ZSTD does better in software. And now the sentence you have to know, because it turns up in an audit or with a partner: **the compression then does not conform to RFC 1951.** The compressed data is encoded according to RFC 1950 and the file format according to RFC 1952 -- readable therefore for any GZIP reader, but whoever checks the bit sequence against RFC 1951 finds a difference. Our manual page says so. In practice that means: interchangeable yes, bit-identical with a software GZIP no. Whoever compares checksums over the **compressed result** instead of over the data sees a difference that is not one. The checksum belongs over the data in the clear. And the point that belongs in the business case, because it is often missing: **the card compresses worse than software.** It is designed for GZIP level 1 and reaches about 0.8 of that; the level cannot be influenced.
With statically structured data the compressed output can therefore be **up to a factor of 10 bigger** than with ZSTD or ADC. The saved MSU are not free, then -- they are paid for in disk space, tapes and line bandwidth. The question a house has to answer is therefore not "does zEDC save CPU" (it does) but **from which factor more storage the saved MSU pay off.**



The slide for the question "and what about TRSMAIN?" -- it comes up in every host course. TERSE is the format with which data sets have been transported on z/OS for decades: PTFs from IBM, dumps to the manufacturer, customer data to a service provider.
There it is produced by TRSMAIN or AMATERSE. What FLAM can do with it, and this is the good news: **it can unpack all four variants** IBM has shipped over time. It can produce the two methods the IBM unpacker accepts -- `PACK` and `SPACK`. On top of that come two header layouts, `NATIVE` and `COMPAT`. What it does **not** do, and this is important: it converts no character sets. Whoever gets an EBCDIC data set tersed and needs text unpacks first and converts afterwards with the character set stage. The other way round just the same. The numbers from our measurement place it: SPACK brings 3.1 to 1, PACK 2.4 to 1 -- so less than GZIP at more CPU time. You do not take TERSE because it compresses well but **because the other side expects it.**



The slide that saves money -- and the one needed most often in practice. **First: already compressed data.** A JPG, a PDF, a ZIP, a cryptogram. A second pass makes them **larger** and costs CPU. That is not a borderline case, that is the rule. The handle against it is `SKIP` or `CMPLEVEL=COPY`.
Second: encrypted data. It looks like randomness, so the compressor finds nothing. And out of that follows the order to remember: **compress first, then encrypt.** The other way round is technically allowed and practically pointless.
Third: when the CPU is scarcer than the space. On a host where the CPU second is charged, it can be cheaper to write a few megabytes more. That is a calculation, not an opinion -- and it comes out differently with a zEDC card than without.
The sentence for the room: **compression is not an end in itself.** It costs CPU, and with the wrong data it brings nothing. The pot design unit shows how to decide that per column instead of for the whole file.

PITFALLS

BEST, because it sounds best.

The highest level costs disproportionate CPU for one to three per cent. Whoever needs more ratio changes the method.

Encrypting and then compressing.

Encrypted data is not compressible. The order is always: compress, then encrypt.

Sending records through a compressor.

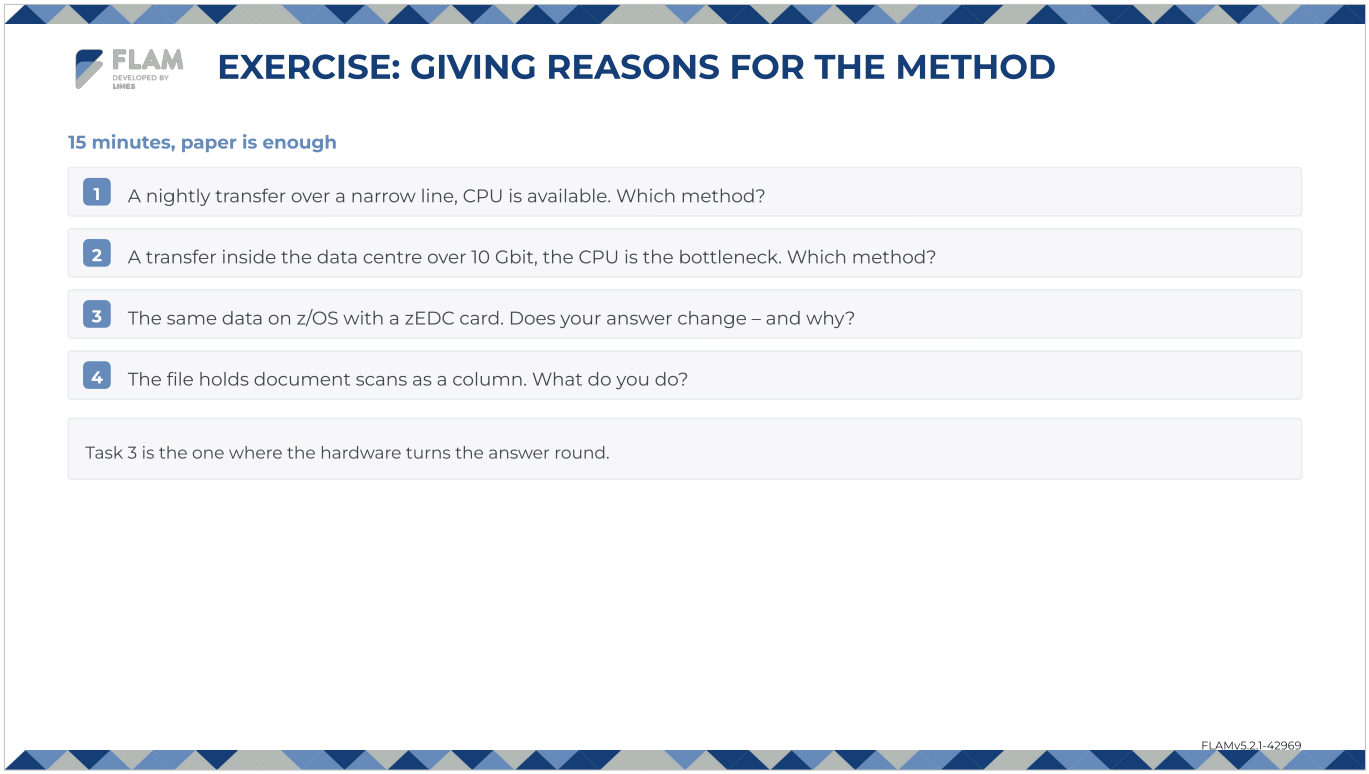
The record boundaries are gone afterwards – the result is a block of bytes. Agree a delimiter, a length field or a fixed length.

Comparing a checksum over the compressed result.

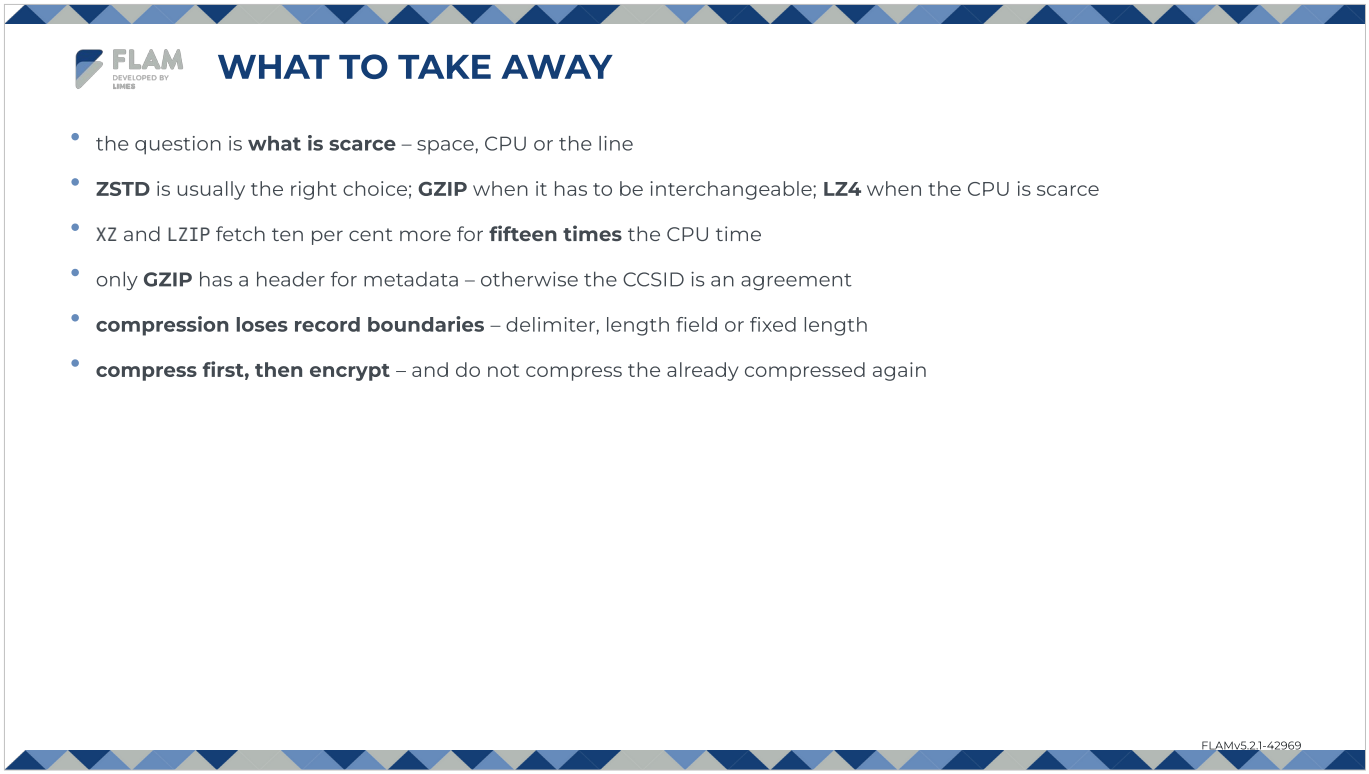
With zEDC the GZIP result is not bit-identical with a software GZIP. The checksum belongs over the data in the clear.

FLAMv5.21-42969

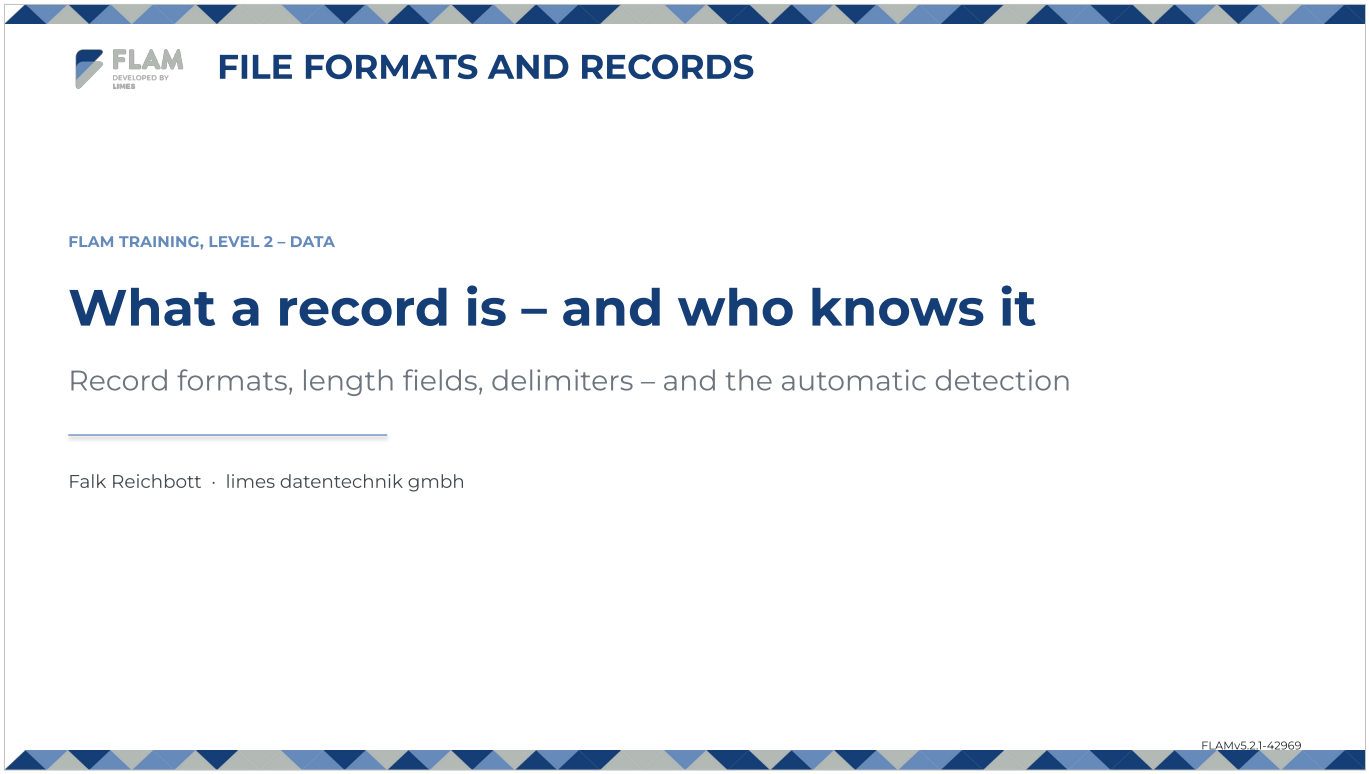
All four out of support cases.



Fifteen minutes. The exercise is deliberately a decision exercise -- there is nothing to type. The solutions: **Task 1.** ZSTD. The line is the bottleneck, the CPU is not -- and ZSTD delivers nearly the ratio of XZ at a fifteenth of the time. GZIP would be the answer if the other side cannot do ZSTD. **Task 2.** LZ4 or nothing at all. If the line is faster than the compressor, every bit of ratio costs more time than it saves. **Task 3.** GZIP -- because of the zEDC card. In software ZSTD would be better, in hardware GZIP wins, because the card takes the CPU time over. **Task 4.** Set `SKIP`, or better: put the column with the document scan into a pot of its own with `CMPLEVEL=COPY`. Then the other columns stay compressible -- that is the pot design unit.



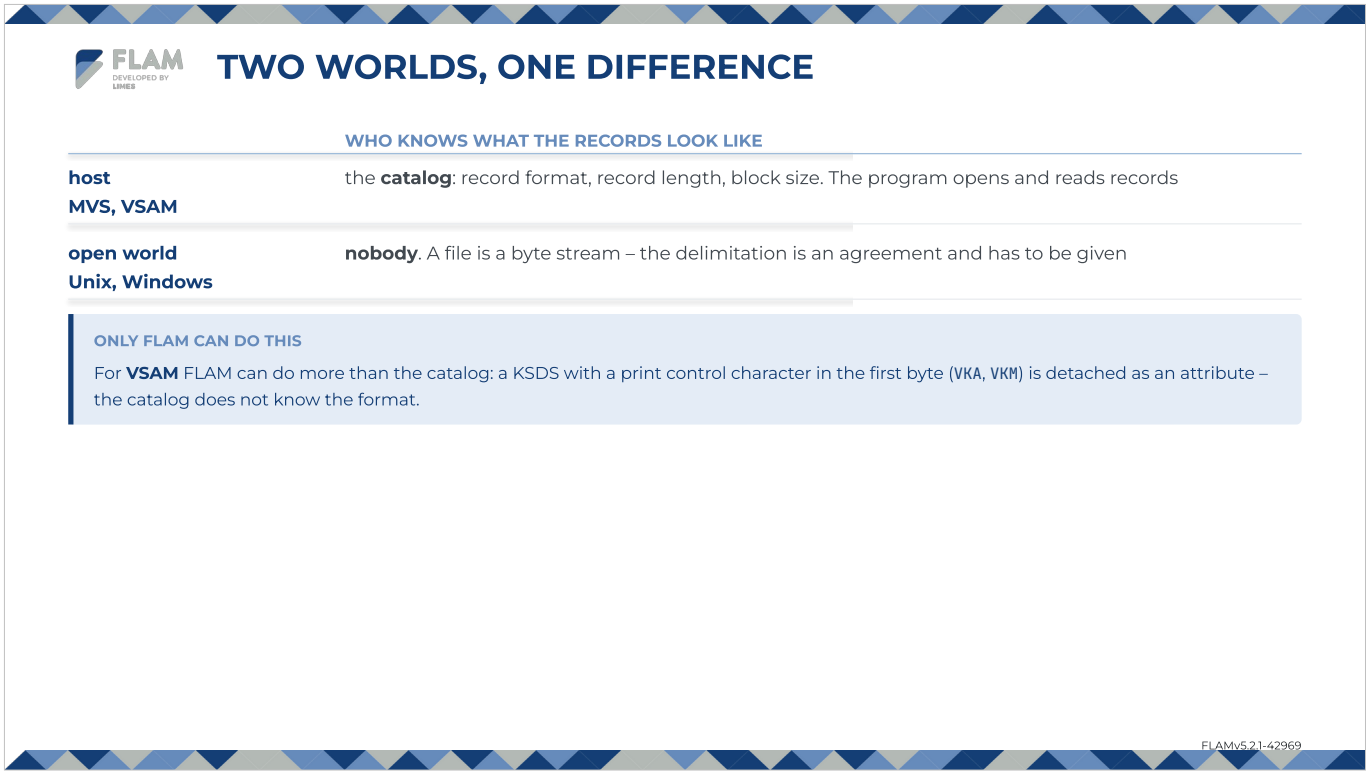
Go through the learning objectives. The pointer forward: how to decide per column instead of for the whole file is in *Pot design* -- and what the hardware takes over on the host is in *Accelerators on z/OS*.



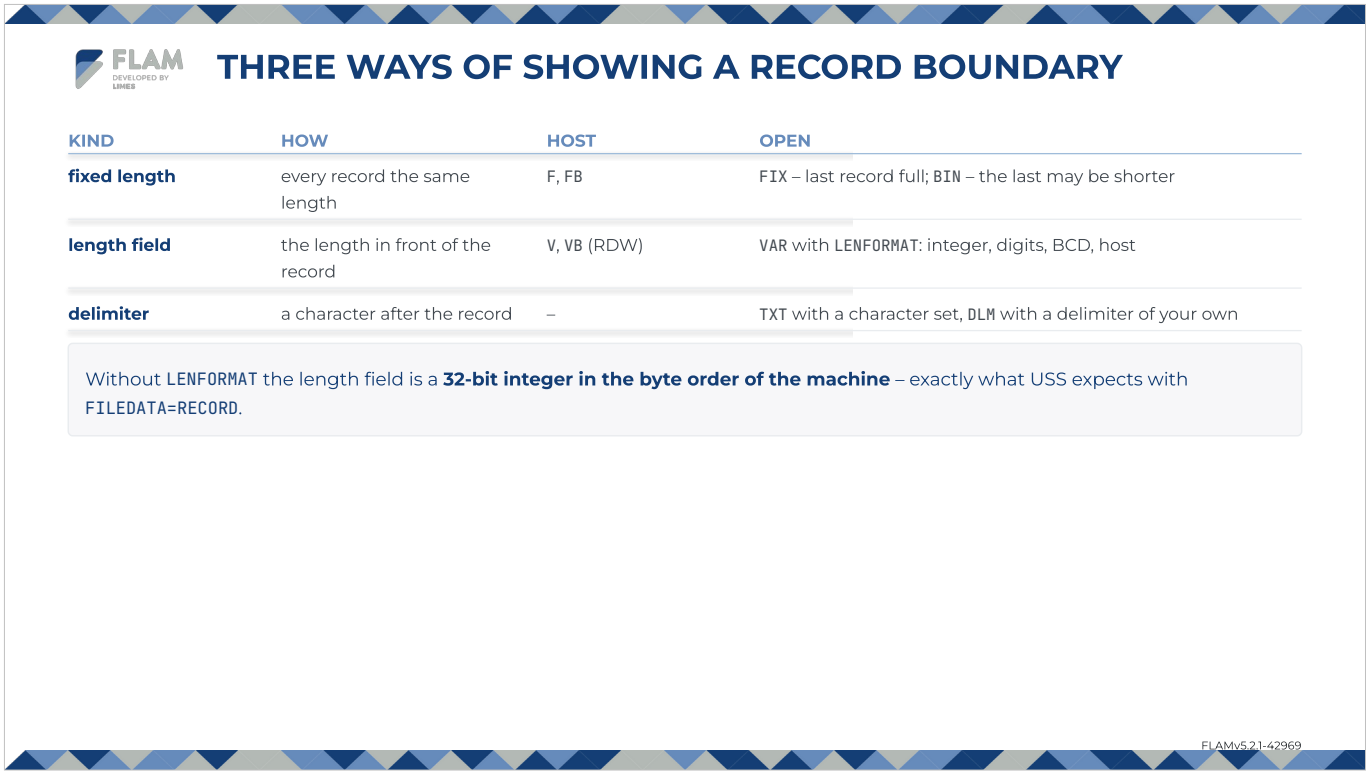
The unit that explains the difference between two worlds -- and it is the one after which most support cases stop. The arc: why a record on the host is something different from one on Unix, the three ways of expressing a record boundary, the format names, the entries around them, the automatic detection, and finally the pitfalls. The sentence for the start: **on the host the file system knows what the records look like -- on Unix and Windows it does not.** That is why it has to be said there. This unit says how.



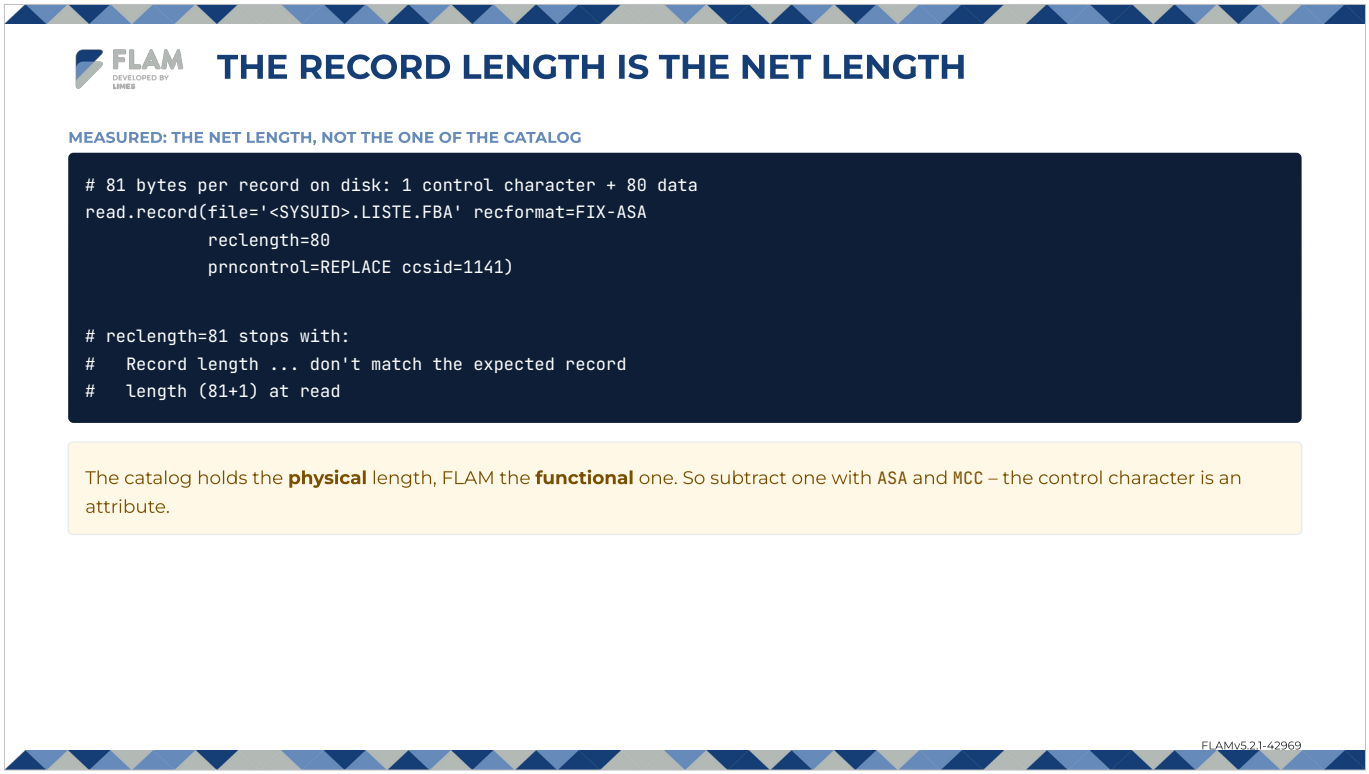
The third objective is the one that produces the support cases: the record length is the **net** length. Whoever takes the length out of the catalog is off by one with print files. The fifth is the handle that saves half the entries.



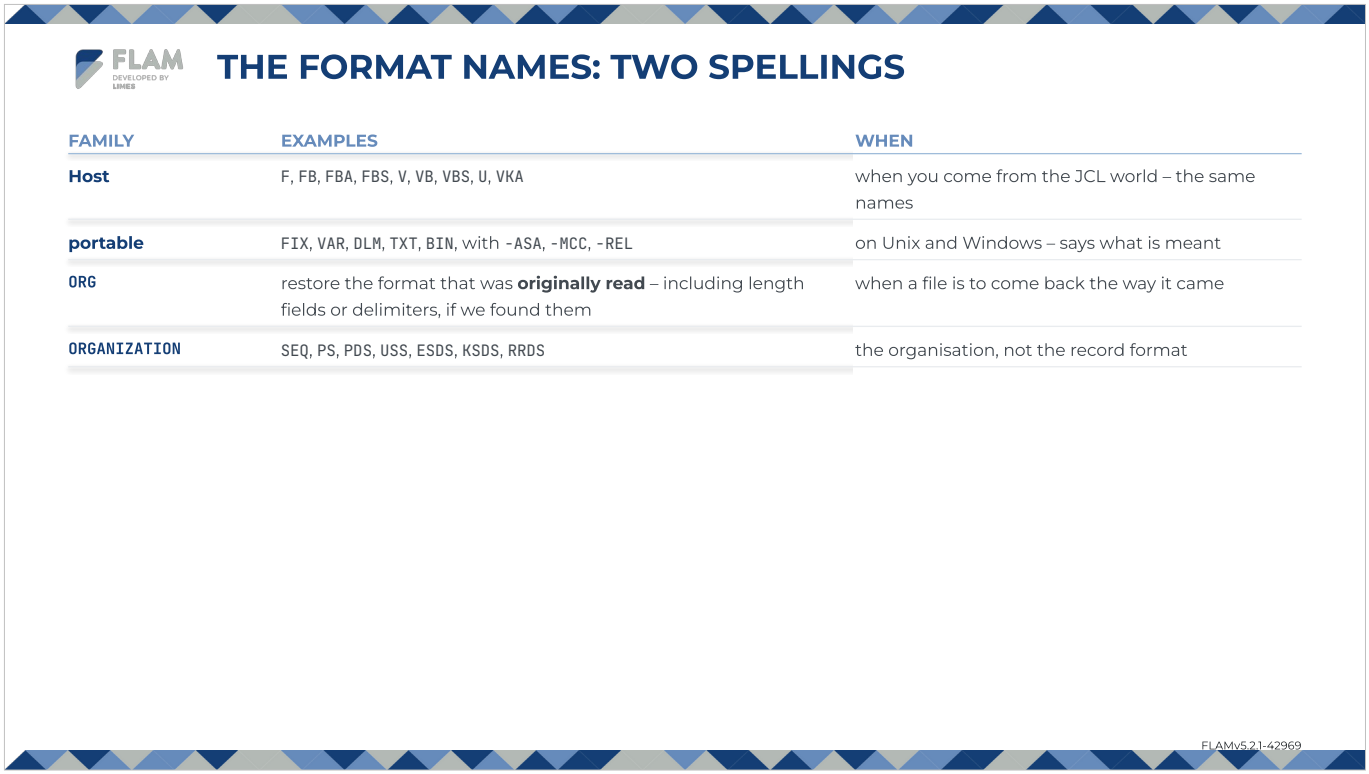
The slide that explains everything else. **On the host** the catalog holds how the file is built: record format, record length, block size. A program opens the file and gets records -- it has to know nothing and give nothing. **On Unix and Windows** there is no catalog. A file is a byte stream. Whoever wants to see records in it has to say how they are delimited -- otherwise there are only bytes. Out of that follows the whole unit: FLAM has to know what the records look like when reading, and what they are to look like when writing. On the host it fetches that out of the catalog, on the open side out of the parameter string. And there is one case in which FLAM can do **more** than the catalog: for VSAM, record formats with print control characters are not provided for. If you know that the first byte in a KSDS is a control character, you can give the format `VKA` or `VKM` -- then the character is detached as an attribute. Our manual page says so, and it is an example of the format entry being useful on the host too.



The core slide. There are no more ways, and each has its world.
Fixed length. Every record is the same length, the end follows from counting. That is the host world (`F`, `FB`) and the portable form is called `FIX`. With `FIX` the last record is of full length too -- with `BIN`, which cuts a byte stream into records, it may be shorter. That is the difference between the two, and it is easy to overlook. **Length field.** In front of every record stands its length. That is the host world of variable records (`V`, `VB`) and on the open side the entry `LENFORMAT`. Four forms: as an integer, as a string of digits, as BCD or in the host format -- that is, as an RDW, the way MVS writes it. **Delimiter.** After every record stands a character or a sequence. That is the Unix world: `TXT` with a character set, or `DLM` for a delimiter of your own in binary data. The important sentence for the open side: without `LENFORMAT` a length field is written as a **32-bit integer in the byte order of the machine**. That is exactly what USS expects with an allocation using `FILEDATA=RECORD` -- so an MVS application can read the file FLAM wrote on USS, without an entry. That is chosen deliberately.



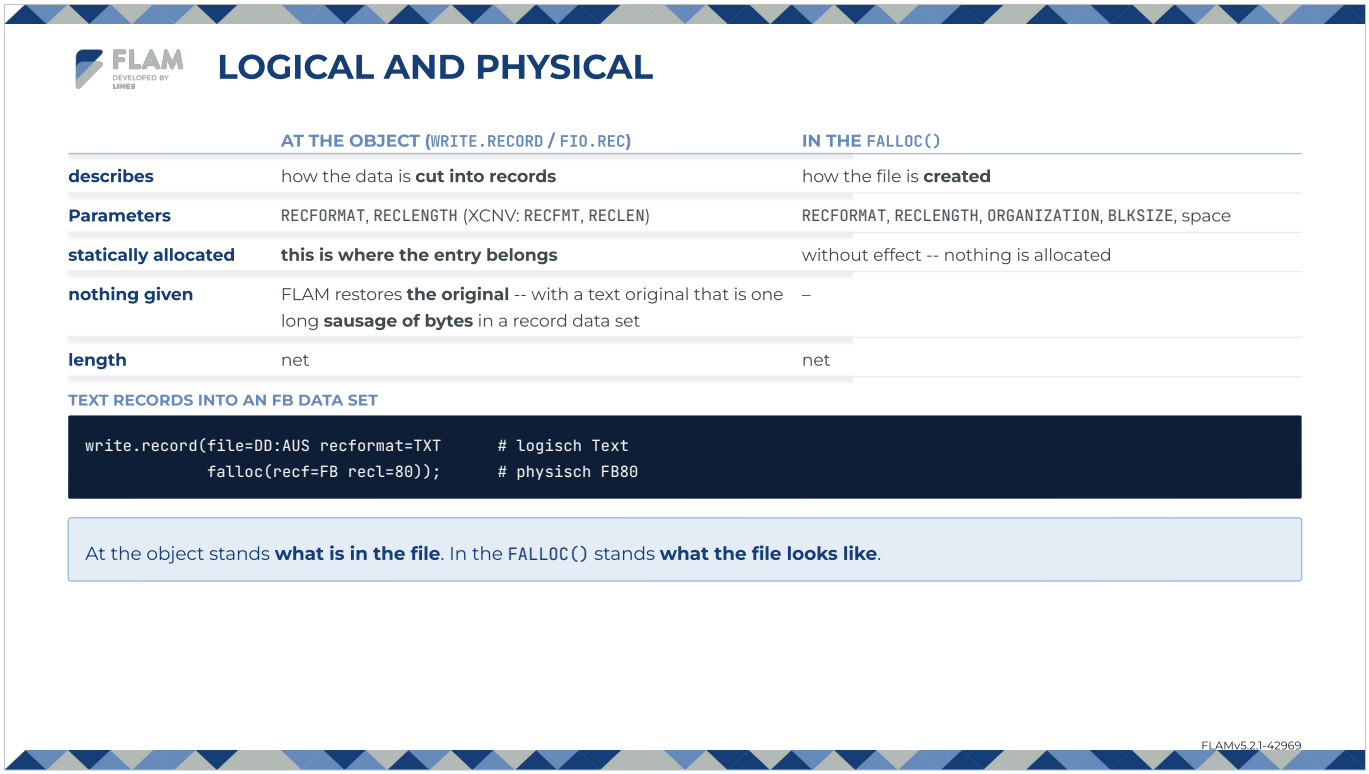
The slide to show before anybody touches a print file -- it saves a support case. The entry `RECLENGTH` in FLAM is **always the pure data length, after all deductions**. The manual says so, and it is the place where host experience and FLAM differ. The example out of the manual: an FBA file with LRECL 121 has the record length **120** in FLAM. The print control character is an attribute, not part of the record. And because it is so easy to confuse, measured here: a file with 81 bytes per record -- one control character and 80 data bytes -- needs `RECFORMAT=FIX-ASA RECLENGTH=80`. Writing 81, the reader says "expected record length (81+1)" and stops. So the error message names both numbers, which makes the search short. The sentence to remember for the room: **the catalog holds the physical length, FLAM the functional one.** Whoever takes the number out of a listing has to subtract one with ASA and MCC.



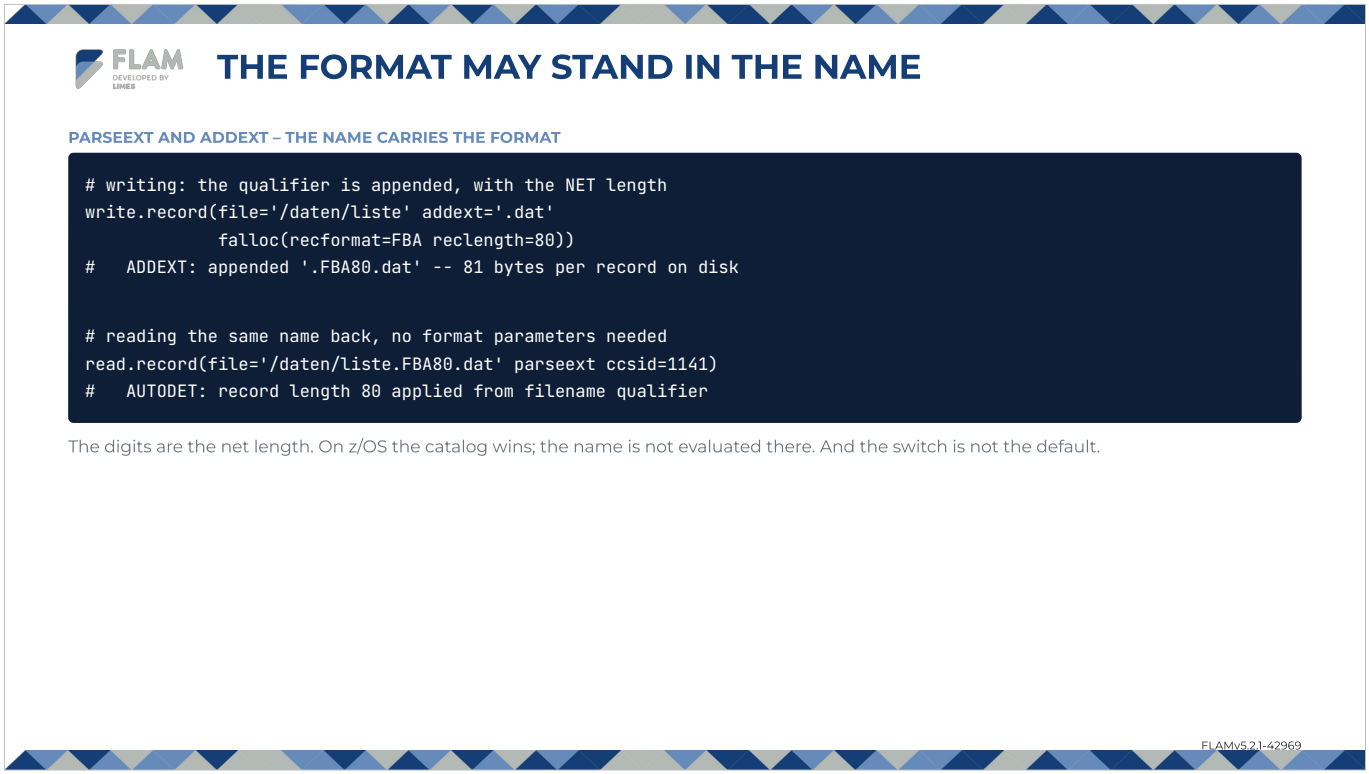
The reference slide. Do not read it out -- explain how it is built, and then the participants know how to read every name. There are two families. The **host spelling** is the one from the JCL world: `F` fixed, `V` variable, `U` undefined, plus `B` for blocked, `S` for spanned, and as a suffix `A` for ASA control characters or `M` for machine control characters. So `FBA` is fixed, blocked, with ASA control characters. Whoever knows JCL knows these names. The **portable spelling** describes the same without host terms: `FIX`, `VAR`, `DLM`, `TXT`, `BIN`, each optionally with `-ASA` or `-MCC`, and `-REL` for the record number with an RRDS. On top of that comes `ORG`, and that is not a catalog lookup but a promise: **restore the format we found when reading.** If we read real records there and found neither length fields nor delimiters, that is exactly the original record format. If on the other hand we found length fields or delimiters, then **`ORG` builds them back in** -- that too is the original. The catalog is another matter: it is interesting for **us**, to know the record format of a file that lies on disk on the host. It says what the system knows about the file -- not what `ORG` gives back. And the organisation is an entry of its own: `ORGANIZATION` says whether it is sequential, a PDS, a USS path or VSAM in one of its forms -- `ESDS`, `KSDS`, `RRDS`. On the open side it is not needed. The practical advice: on the host give **nothing** and let the system do the work -- it knows the format of the file. On the open side take the portable spelling, because it says what is meant instead of what it once was called.



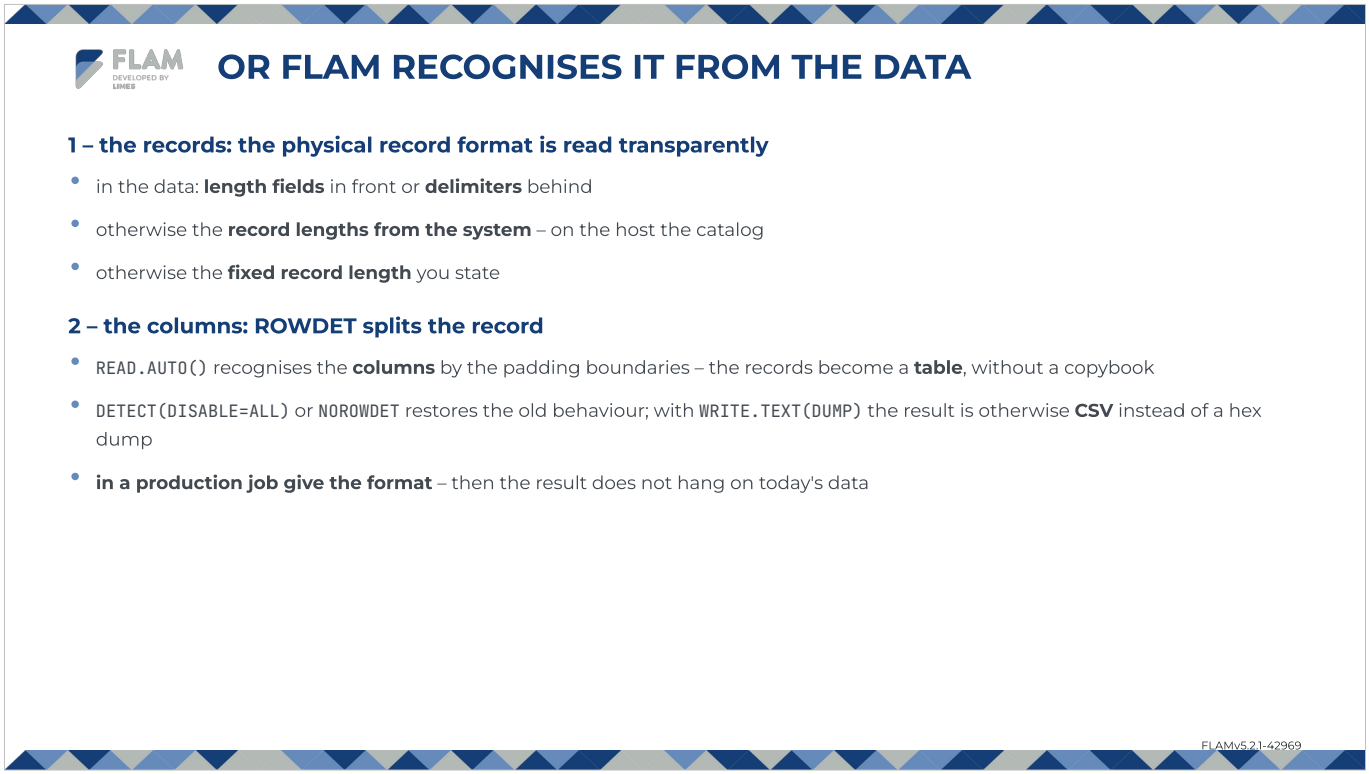
The second reference slide -- and the four entries on it cover ninety per cent of the cases. `PRNCONTROL` decides what happens to a print control character: `DETACH` treats it as an attribute, `RETAIN` leaves it in the record, `ERASE` deletes it. `REPLACE` is more than "a blank line": it **prepares the print completely** and executes the control characters on the data in memory. If an ASA `3` (`0xF3`) stands there, the three line feeds become two additional blank lines behind it. For the page break FLAM fills the page up -- with **60 lines per page** as the default, adjustable through `RPLFFD` (where `0` again means 60 and `1` means: ignore page breaks).
What does not work has to be said, otherwise the participant asks:
overprinting cannot be represented in a file. Underlining becomes a line of hyphens, and text printed in bold simply stands there twice -- it does not get any thicker for it. The result looks as if it came from a daisy-wheel printer, with the limits of a daisy-wheel printer. And the small thing that costs time in support: **set `RPLFFD` and not `PRNCONTROL`, and `REPLACE` is the default** -- no longer `DETACH`. That is how it stands in the code, and it makes sense too: whoever states the lines per page wants to print. `SUPPADDING` removes the padding characters at the end of the record, and `PADCHARACTER` says which character that is. Measured: `x'40'` is recognised as `EBCDIC-BLANK` -- so the byte may be written instead of looking the name up.
`RECMODE` decides what happens with a record that is **too long**: `STOP` is the default and stops, `CUT` cuts it off, `WRAP` makes a second record out of it, `SKIP` skips it. The choice is a functional one: a truncated amount is worse than an abort. One limit to know about: as soon as a **record delimiter** (`RECDELIM`), a key disposition or retained print control is in play, the record has to be buffered, and the mode acts before that. Then only `SKIP` and `CUT` help -- `CUT` shortens the record so that the delimiter still fits into it. `WRAP` and `STOP` are refused with a message instead of aborting. Without a delimiter, a key and print control every mode works for every record length.
`RECCOUNT` and `BLKSIZE` are quantities for the processing, not format entries -- they decide how much is read at once.



The slide that answers a question which had no good answer before:
what if the records are to look different from the file they are written into? Since 5.2.1 those are **two separate entries**.
`RECFORMAT` and `RECLENGTH` at the write object itself say how the data is cut into records before it is handed to the file layer -- that is the **logical** side. The same two parameters inside the `FALLOC()` say how the file or the data set is **created** -- the physical side. Both lengths are net, so data without print control characters and without length fields. The sentence that matters for the customer: **both may differ, and that is the point.** A text matrix may go as text records into an FB data set, and fixed records into a VB data set. And now the part that causes the support cases in practice -- the **default behaviour when writing**: say nothing and FLAM tries to **restore the original**. With a dynamic allocation that is right and goes unnoticed. With a **static** allocation through a DD statement it becomes a trap: the data set gives the DCB, FLAM gets no specification of how to shape the data, and restores the shape of the original. If the original was an ordinary text file -- a byte stream with line separators -- then exactly that byte stream stands in the FB data set: the separators stand in the data, every record but the last is filled to the brim.
Whoever expected records finds one long sausage of bytes. The way out is this very slide: **give the record format you want, logically**, at the write object. `recformat=FIX reclength=80` says how the data is to be cut into records -- regardless of the fact that with a static allocation there is nothing to allocate. If the original was a record data set itself, it fits anyway and nothing has to be given. And the case that caused the change: with a static allocation through a DD statement there is no allocation to describe at all. There the entry belongs at the object -- the `FALLOC()` runs into nothing. Up to 5.2.0 it was the only way and acted as the logical entry as well; that stays valid, but it is no longer the right spelling. What does not go together is refused, and the message names the parameter that resolves it -- `PRNCONTROL`, `RECMODE` or `FALLOC(ORGANIZATION=)`. So nobody has to guess.



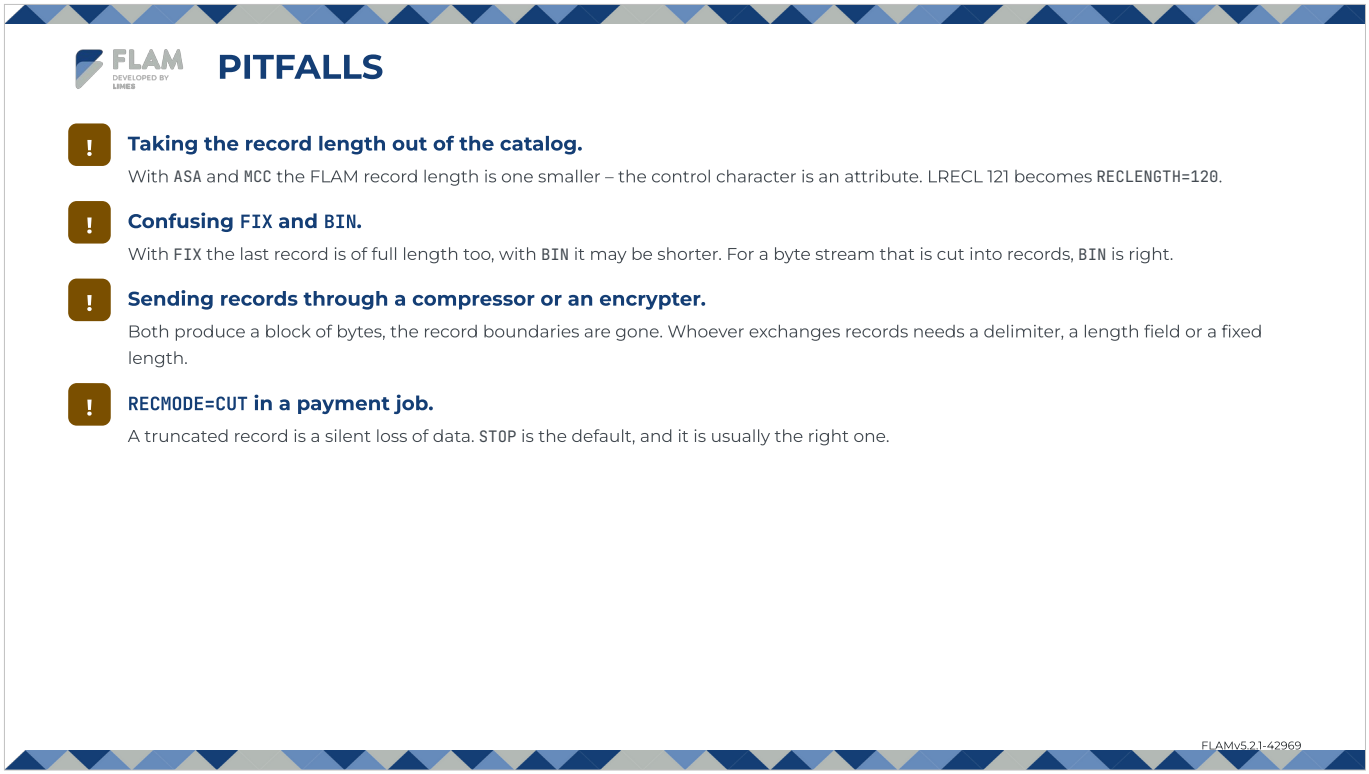
The slide with the handle hardly anybody knows and that simplifies a chain of jobs. On the open side the record format can stand **in the file name**, as a section between dots or underscores. With the switch `PARSEEXT` FLAM reads it and applies format and length without them being repeated in the call. Measured with a file `plain.FB80.bin`: "AUTODET: record length 80 applied from filename qualifier". The digits are the **net length**, not the one of the catalog. `.FB80.` and `.FBA80.` both mean 80 bytes of user data and differ only in the format -- with `.FBA80.` there are 81 bytes per record on the disk, because the control character comes on top. Whoever takes the control character into the data and writes 81 bytes gets `.FB81.`. What that is good for: when a file travels from the host to a Unix system and is processed there by several jobs, it carries its format itself. No job has to know what is in it -- and when the format changes, the name changes, not the JCL. The other direction exists too: when writing, FLAM can **append** the qualifier to the name so the next stage can read it. Two limits to state: on z/OS the catalog always wins -- the name is not evaluated there. And the switch is **not** the default; without `PARSEEXT` the name stays a name.



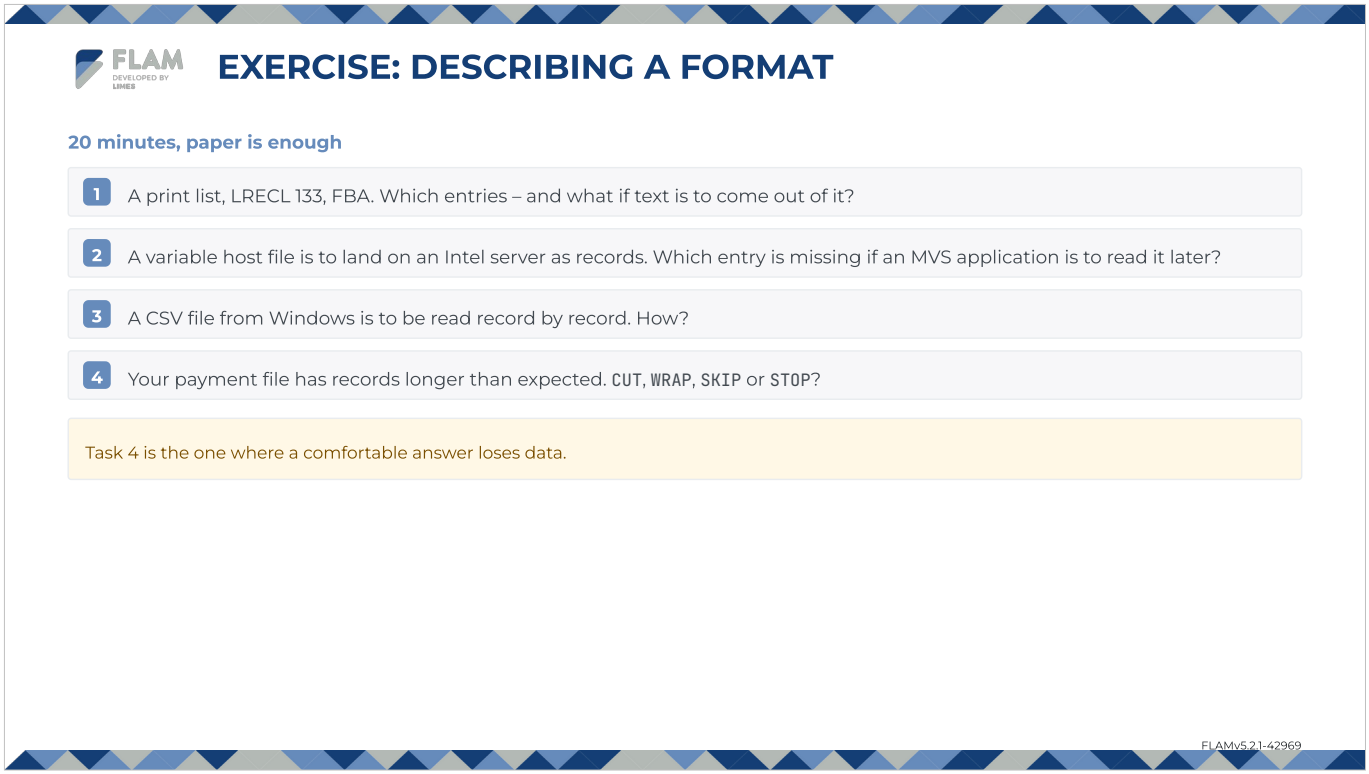
This slide has two halves, and separating them is half the understanding to be gained: **above the records, below the columns.** They are two different detections, and only the lower one is called ROWDET. **Above: the physical record format.** When reading, FLAM looks in the data first -- are there **length fields** in front or **delimiters** behind? If it finds none, it takes the **record lengths from the system**, that is whatever the catalog or the file organisation gives. And if it does not have those either, the **fixed record length** that can be stated remains. With that FLAM reads every one of the four physical record formats transparently, without anybody having to announce it -- and after that one has records. **Below: the columns.** Only now does **ROWDET** come into play: `READ.AUTO()` splits the records into columns by the padding boundaries and the kinds of data and makes a **table** out of them. That is the detection which is new and which changes the behaviour -- which is why it stands as an explicit note in the release notes. What it brings: a file whose structure nobody knows any more becomes readable without having a copybook. That is the case which occurs in every modernisation. What you have to know because it changes existing behaviour: whoever wants the old treatment switches it off with `DETECT(DISABLE=ALL)` or `NOROWDET`.
And with `WRITE.TEXT(DUMP)` a detected table produces **CSV** instead of a hex dump -- whoever wants the dump sets `NOROWDET` inside `READ.AUTO()`. The advice for practice: the detection is a tool for the analysis and for the single case. In a production job that reads the same format every night the format is stated -- then the result does not depend on what is recognisable in today's file.



Two edge cases that each produce one support call a year -- and both stand in the manual and in no course. **The empty record.** On z/OS the language environment does **not write** an empty record of variable length at all. So that it does not vanish, FLAM writes a padding character: `0x00` for binary data, the blank `0x40` for text. When reading, a record of length 1 with exactly that character becomes an empty record again -- provided `_EDC_ZERO_RECLEN` is set to `Y`, which is the default. Whoever switches that variable off in the house gets a blank out of an empty record, and nobody knows why any more. The sentence to take away: **an empty record survives the round trip, but only because FLAM slips a character underneath. The large record.** FLAM5 handles **spanned records** -- the limit of 1 MiB is not a technical one but a memory decision: we allocate `RECCOUNT` times the record length in advance, and more than 1 MiB per record would make that unjustifiably large. **On request the length can be extended**, and the manual says so too. The restriction lies with **FLAM4**, not with FLAM5. Whoever wants to write a large record into an old FLAMFILE turns the matrix into a block with length fields -- FLAM5 therefore gets around the FLAM4 limit by **partitioning**, and with that long records fit into an old FLAMFILE as well. The FLAM4 components see an undefined byte stream while doing so. When reading, the same method has to be stated, otherwise the records do not come back. In `CONV` there is the switch **`LARGER`** at `READ.FLAM4()` and `WRITE.FLAM4()` for it; it uses the method `B4X`. Automatically it does not happen, and that is a deliberate decision. Two limits that go with it: the **automatic record length detection stops working at 32768 bytes** -- there the format has to be stated. And the FLAM5 I/O reads large V(B)S records by itself; `NOSPAN` switches that off.



All four out of support cases, and the first is the most frequent of all.



Twenty minutes, paper is enough. The tasks are real cases.
The solutions: **Task 1.** `recformat=FIX-ASA reclength=132` -- the LRECL is 133, the control character comes off. Plus `prncontrol=REPLACE` if text is to come out of it. **Task 2.** `recformat=VAR lenformat.host()` -- the RDW, the way MVS writes it. Without the entry it would be a 32-bit integer in the byte order of the machine, which is the wrong order on an Intel machine. **Task 3.** `recformat=TXT` with the CCSID of the file, or `DLM` with a delimiter of your own if it is not text. **Task 4.** `STOP`. A truncated or skipped record is a silent loss, and in a payment file that is not negotiable. Whoever wants to see the long records runs one pass with `SKIP` and looks at the log -- but not as a permanent state.

WHAT TO TAKE AWAY

- on the host the **catalog** knows it, in the open world it is an **agreement**
- three ways of a record boundary: **fixed length, length field, delimiter**
- `RECLENGTH` is the **net length** – with `ASA` and `MCC` one less than in the catalog
- without `LENFORMAT` the length field is the one of **USS with** `FILEDATA=RECORD`
- `PARSEEXT` reads the format **out of the name**, `ADDEXT` writes it in – the digits are the **net length**
- `RECMODE` is a functional decision – `STOP` is the default

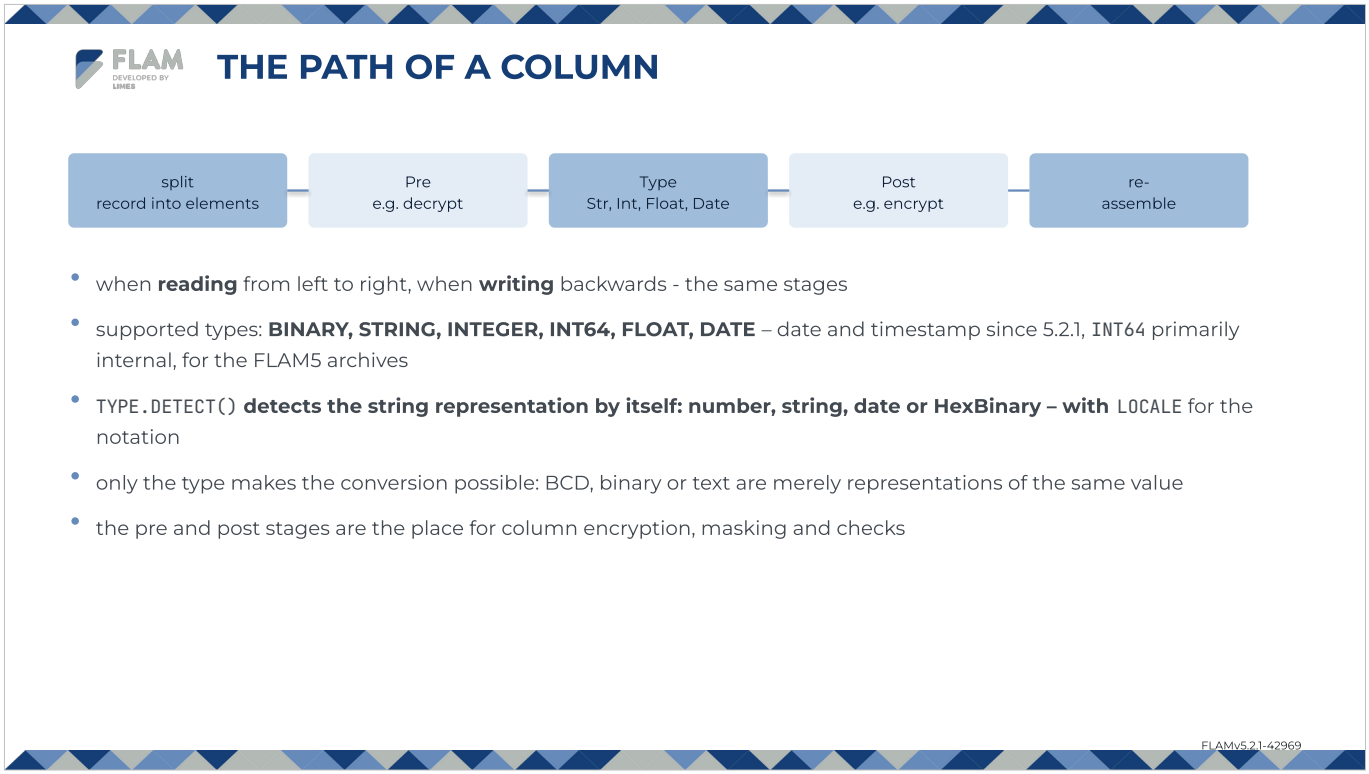
Go through the learning objectives. The pointer forward:
how records become **columns** -- types, copybooks, CSV -- is in *Tables and CSV*. This unit ended at the record.



Fourth functional topic and the most powerful one. Up to here a file was a stream of bytes or records -- now it becomes a table with typed columns. The origin is payment traffic: processing DTA, SWIFT and SEPA uniformly, as records in COBOL or PL/I and as a stream in Java or C++. But it helps with almost every commercial data format.



Objective three is the one that halves the effort: when writing you usually do not have to describe the format.



Five stages, and they are symmetric: left to right when reading, backwards when writing.
The core is the middle: the neutralised element with a **type**. Only the type turns bytes into a number or a character string -- and only that lets FLAM convert between BCD, binary and text.
The pre- and post-processors are the place for column encryption and masking. They deliberately sit between decomposition and typing.
On the state of the types, because something was added here recently: besides `BINARY`, `STRING`, `INTEGER` and `FLOAT` there is **`DATE` since 5.2.1** for date and timestamp, and **`INT64`**, which is used primarily internally for the FLAM5 archives.
And one type that is not really one: **`TYPE.DETECT()`**. With it we detect ourselves which string representation is present -- number, string, date or HexBinary. Via `LOCALE` you say which notation is to be expected: `DE` and `US` differ in the decimal separator and in the date format, and that is exactly the place where an automatic detection otherwise takes a wrong turn.

WHICH TABLE FORMATS

KIND	FORMATS	WHERE THE STRUCTURE COMES FROM
flat	FIX (fixed record length)	column widths, optionally with alignment
flat	VAR - DLM or LFD	delimiter or length field
flat	CSV	delimiter, optionally with headline
hierarchical	XML	path specification within the hierarchy
hierarchical	TVD - tag-value-delimiter, such as SWIFT	path specification
hierarchical	TLV - tag-length-value, such as ASN.1	path specification

Headline, header and trailer separate tables in unload and backup files - but a table change is detected even without them.

Only on the format side, that is in memory: `VARFMT` delivers the columns as an **array of pointer and length**, optionally with null indication – for every column or only for the populated ones. For the populated ones a pointer to the null-terminated column name is added, so that you can correlate. On the file side this does not exist.

FLAMv5.21-42969

Go through the list once, but without details. What matters is the split in two:
flat formats and hierarchical ones.
With the flat ones the structure comes from record length or delimiters, with the hierarchical ones from a path. The path is the reason why XML and SWIFT sit in the same mechanism as a CSV file.
Headline, header and trailer are the markers by which a table change is recognisable in unload and backup files -- but the change is detected even without them.
One format is missing from the table, because it does not describe a file but memory: **the pointer formats.** Whoever writes into memory can get the columns as an **array of pointer and length** -- or as pointer, length and flag, if a null indication is needed.
There are two variants: **for every column** or **only for the populated ones**. With the second, a further pointer comes along that points to the null-terminated column name -- otherwise the caller would not know which value belongs to which column.
Important for expectations: **this works only on the format side**, that is in memory, not on the file side. A file made of pointers does not exist, naturally.
For developers this is the most convenient way to fill a structure without computing offsets -- and exactly the point where the record interface wins over a hand-written decomposition.

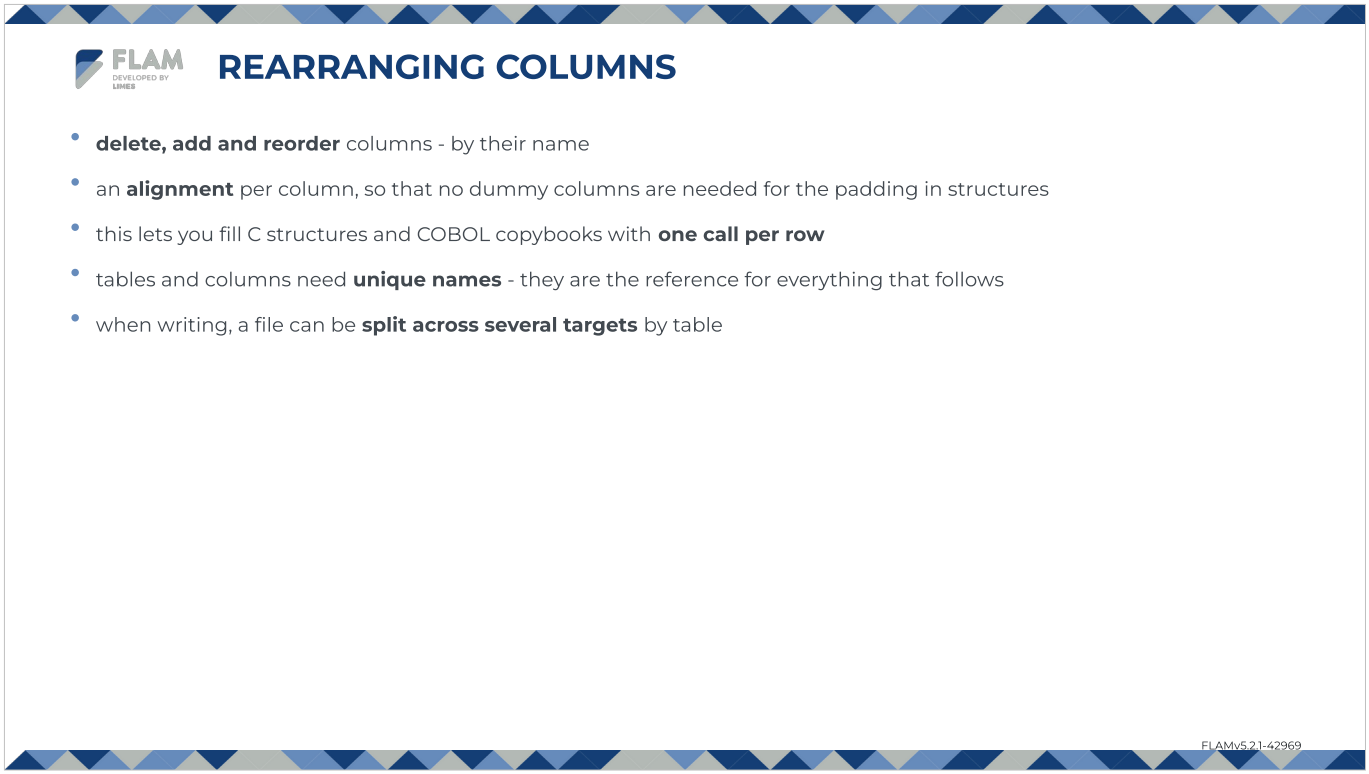
DETECT WHEN READING, ADOPT WHEN WRITING

- when reading, the format information for each table is recorded in a **header member**
- when writing, a **format selection** is then enough - the format does not have to be described
- you only have to describe the input, so that FLAM understands the external format
- **several tables in one file** are detected - even with simply concatenated FB datasets

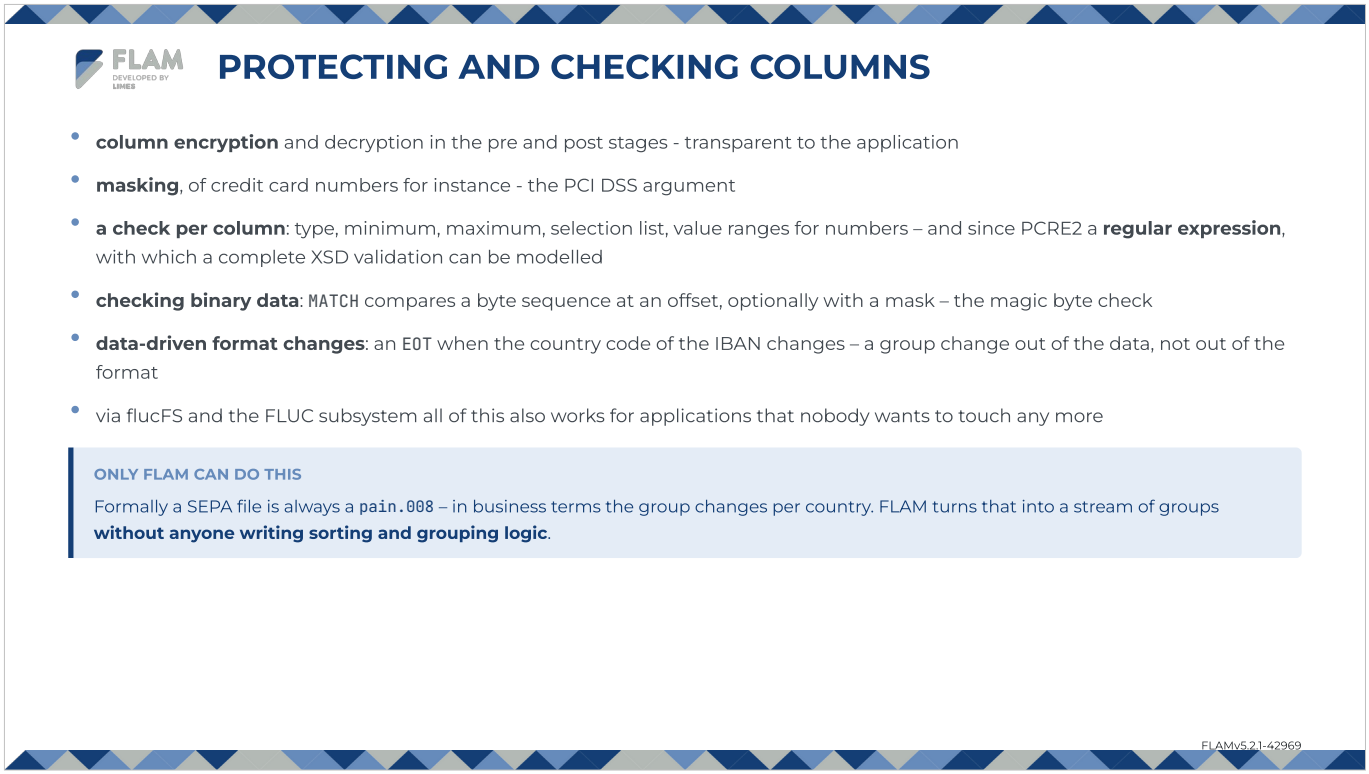
The application developer always gets the same fixed structure per row - regardless of the format the data was delivered in. Until the end of table is reported.

FLAMv5.21-42969

This is the slide that halves the effort, and it is readily overlooked.
When reading, FLUC collects the format information per table in a header element. When writing, a format selection is then enough -- the format does not have to be described again.
You only have to describe the **input**, so that FLAM understands the external format. And even that only if columns are to be deleted or reordered.
The practical effect: different platforms deliver different formats, and the application always gets the same fixed structure per row.



The four operations and the alignment. The alignment is the point that developers appreciate: a compiler places every structure element on its **natural boundary** -- a 4-byte integer on an address divisible by 4, an 8-byte value on one divisible by 8. Between the fields this creates padding bytes that are not in the copybook. Without an alignment specification you would have to create dummy columns for that air.
How large the boundary is depends on the compiler and on the platform -- and that is exactly why it is a specification and not a fixed rule.
With it, C structures and COBOL copybooks can be filled on different platforms and against different compilers with **one call per row**.
That is the argument for the record interface.
Names are mandatory -- without unique table and column names you can reorder nothing and query nothing.



The part that counts in compliance discussions, because it sits below the application: column encryption and masking happen in the pre- and post-stages, that is without the application knowing anything about it.
The example that always works: masking credit card numbers. Anyone who has PCI DSS in the house understands at once why this has to happen below the application.
The per-column check gained RegEx -- type, minima, maxima and selection lists were already there, the pattern is new. With it an XSD pattern check without a separate XML validator is possible; with regular expressions a **complete XSD validation** can be modelled.
Three things that are often overlooked here:
With **numbers** you can restrict value ranges -- not just length, but the permitted range.
With **binary data** you can validate: `MATCH` checks a byte sequence at an offset, optionally with a mask. That is the magic-byte check -- with it you recognise whether a column really holds a JPEG and not something else of matching length.
And the trick that saves the most time in practice: **you can force artificial format changes from the data.** That is, get an `EOT` when the country code of the IBAN changes. A SEPA file is formally always a `pain.008`, but functionally a group change happens per country -- and that is nowhere in the format, it is in the data. With it a file becomes a stream of groups, without anyone writing sorting and grouping logic.

THE COMMANDS FOR CSV

COMMAND	STARTING POINT	RESULT
BCSV	binary file **with** a description: COBOL copybook, PL/I, HLASM	deterministic, fit for production
ACSV	binary file **without** a description	detected automatically, schema can optionally be saved
FCSV	existing CSV file	normalised: delimiter, encoding, structure

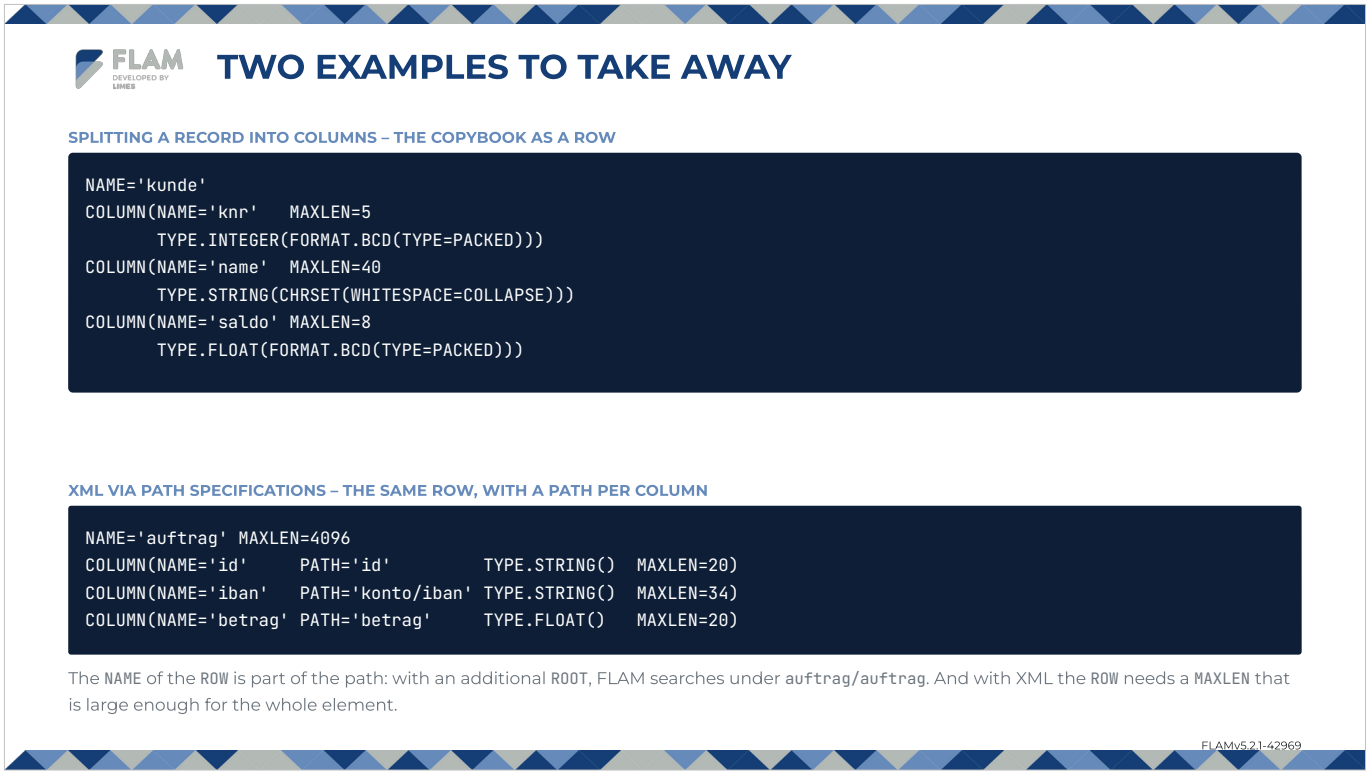
If you have a description, use BCSV - the result is reproducible. ACSV is for exploration and for cases where there is no description.

FLAMv5.21-42969

Three commands for three starting situations. The distinction matters, because otherwise customers pick the wrong one.
BCSV needs a description -- a copybook, a PL/I DECLARE, an HLASM DSECT. ACSV needs none and detects by itself. FCSV normalises existing CSV files.
Whoever has a description should take BCSV: the result is deterministic.
ACSV is for exploration and for cases without a description.



Two points that were on no slide and that are both needed regularly in conversations.
First: the three commands also exist **as standalone programs** -- `flbcsv`, `flacsv` and `flfcsv`, each with its own handbook. Anyone who has only this one task does not need to licence the whole of FLAM. For a conversation with a business department that has exactly one problem, that is often the way in -- and moving up to the full FLAM afterwards costs no change to the calls, because it is the same language.
Second `UNLOAD`, and this is the point for a data centre with IMS or DB2. A database unload is not a flat FB file: every logical record has a prefix block, and at the end there is a sentinel record. With `UNLOAD=`, every FORMAT schema is wrapped in exactly this shell -- a prefix block before every data row and a catch-all row at the end for the sentinel. Without that, someone would have to compute the prefixes away themselves.
Name the formats if someone from the database world is in the room:
DFSURGU0 in its variants LONG, SHORT and HD unload, each also in the flavours with and without a length field, plus the DB2 unload formats. Otherwise just say that they exist.



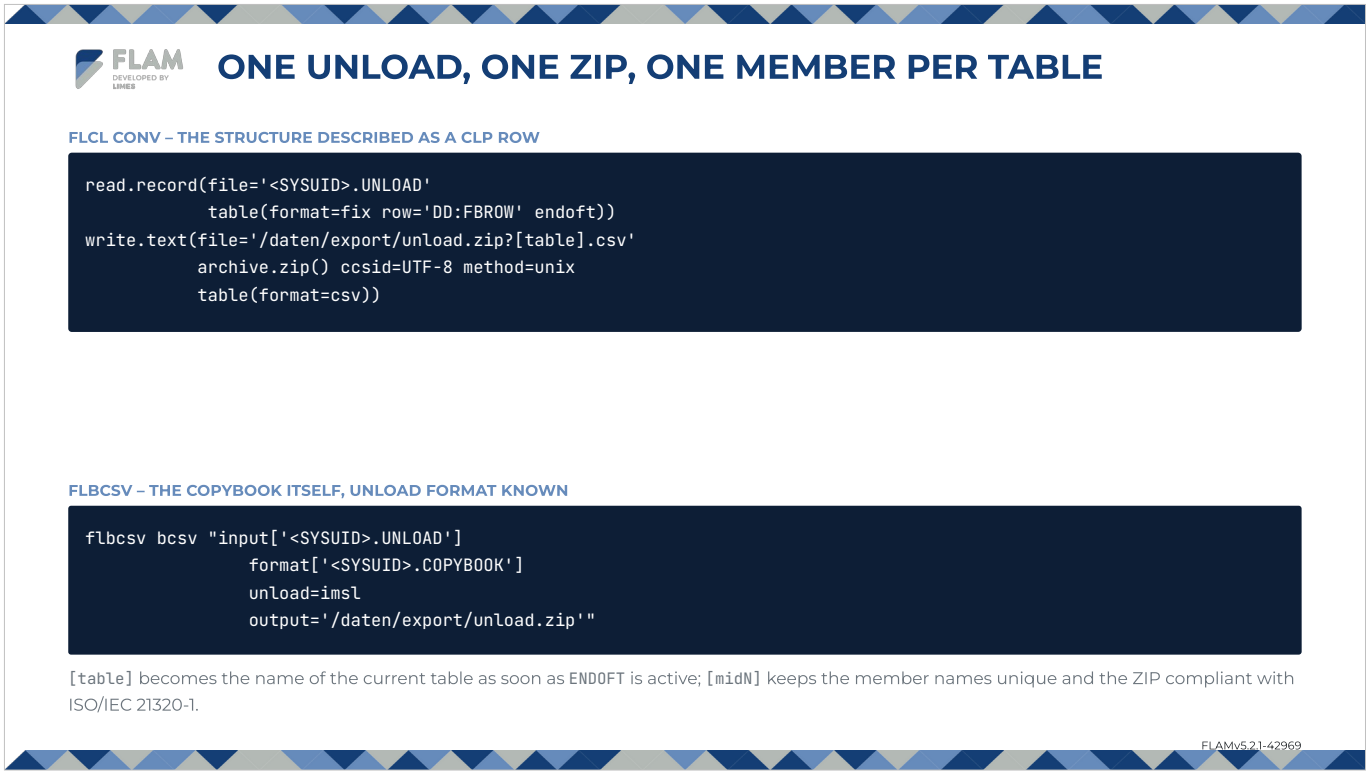
The concrete examples that are always asked for afterwards -- both kept small, so that you see the structure and not the volume. Both have been run through like this, this is not pseudocode.
Left: decomposing a record into columns. That is the copybook question. The `ROW` object describes the row, each `COLUMN` a column with name, type and length. Packed BCD, a character string with `WHITESPACE=COLLAPSE` against the padding characters, a floating-point number -- those are the three cases that make up 90 per cent of a mainframe dataset.
What matters is the separation: the `ROW` sits in a **file of its own** and is fetched with `row='...'`. That way several jobs use the same description, and the description can change without any job being touched.
Right: reading XML via path specifications. The same `ROW` object, but the columns now carry a `PATH`. With that, an XML document becomes a table -- without XSLT, without a parser in your own code.
And two traps in it that I found myself while running it through, so please pass them on:
First: **the name of the `ROW` is part of the path.** Whoever additionally specifies a `ROOT` gets both one after the other -- `NAME='auftrag'` and `ROOT='/auftrag'` become `auftrag/auftrag`, and then it finds nothing. So either the one or the other.
Second: with XML the `ROW` must carry a **`MAXLEN`** that suffices for the whole element. The default is too small, and the message says it exactly -- "row is longer than the provided maximum length".

THE SAME THING OVER THE THREE ROUTES

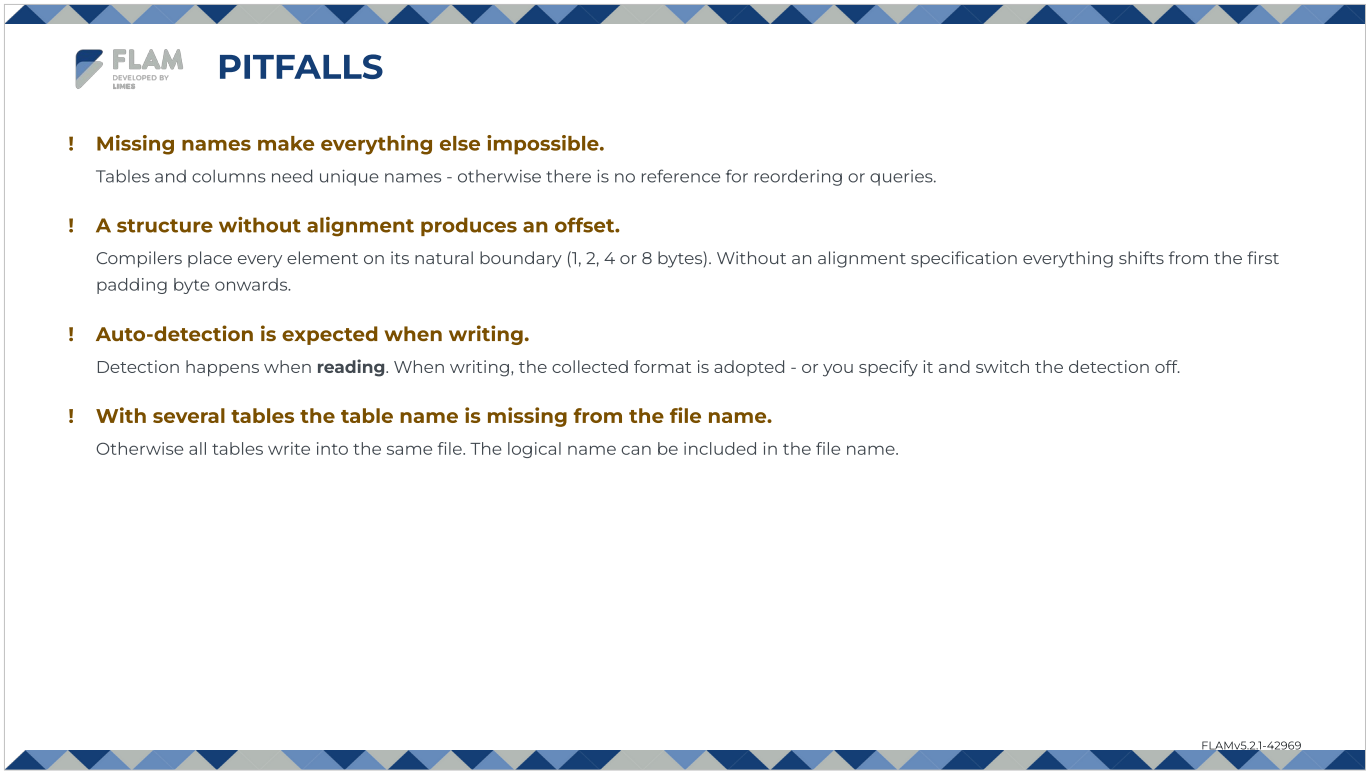
ROUTE	WHAT YOU GET OUT OF IT
Batch utility	`flcl BCSV`, `ACSV`, `FCSV` or `CONV` with a table specification - planned conversion of whole files
Subsystem, file system	column encryption and masking for applications that must not notice any of it
API	a fixed structure per row, one call - regardless of the format it was delivered in

FLAMv5.21-42969

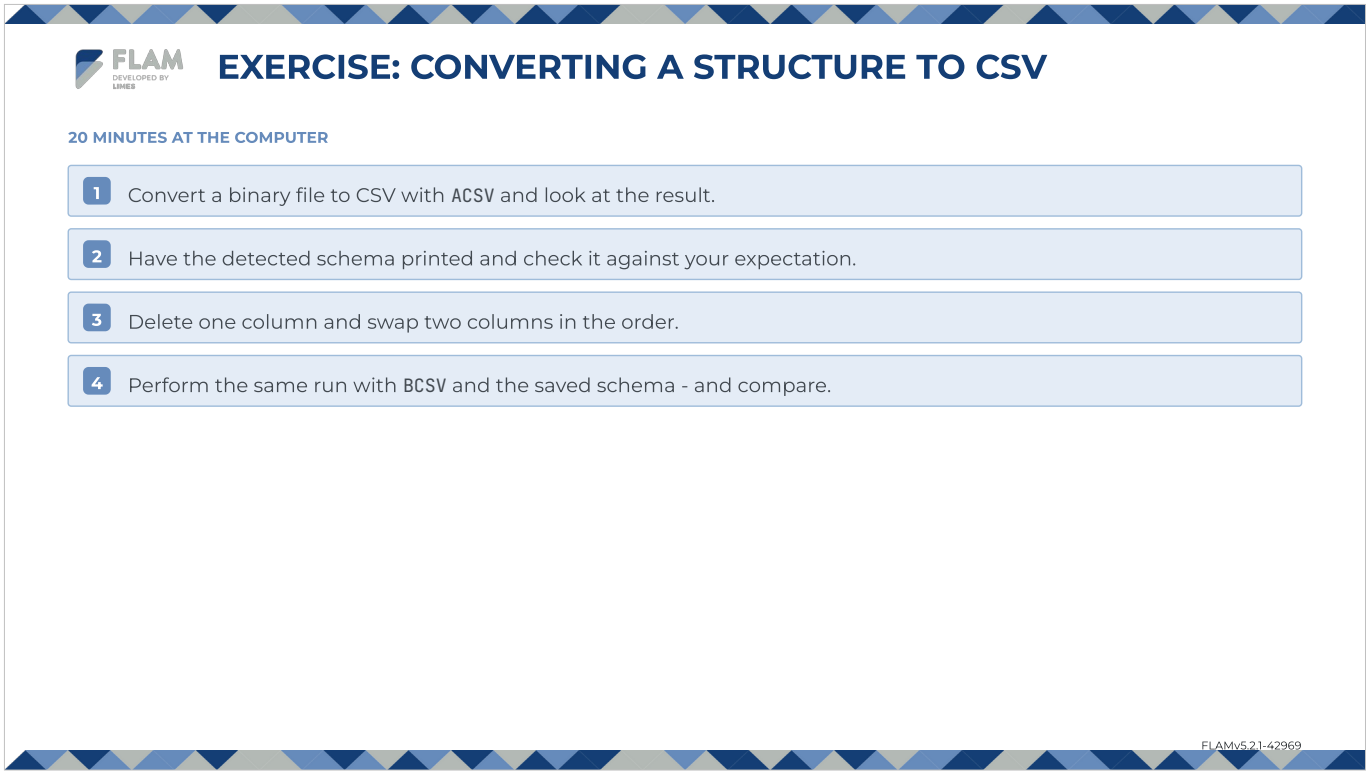
With tables the API route is particularly attractive, because the application gets the same fixed structure per row, independently of the external format.
And the subsystem route is the one with which you secure off-the-shelf software that you are not allowed to touch.



The batch example asked about most often: an unload with **many tables** is to become a ZIP, with **one CSV per table**, in UTF-8 and with Unix line endings -- ready for the Linux side. The key is a single token in the output name: **`[table]`**. It is evaluated to the name of the current table as soon as the end-of-table handling is active (`ENDOFT`). A member of its own thus arises per table automatically -- without a loop, without an analysis beforehand, without knowing how many tables are in there. If the same row specification writes into the same ZIP more than once, there is additionally **`[midN]`**: a counter local to the archive, so that the member names stay unique and the ZIP is **ISO/IEC 21320-1** conformant. That is the standard authorities and banks demand for long-term storage. And now the comparison, because it explains the choice of tool: **`flcl CONV`** takes a **CLP description** of the row -- the `ROW` object from the previous slide. That is the way when you describe the structure yourself or assemble it from several sources.
`flbcsv` takes **the copybook itself** and translates it. For that it knows the unload formats directly: `UNLOAD=IMSL`, `IMSS`, `IHDU` for IMS, `DB2U` and `DB2S` for DB2. Anyone who has an IMS unload and a COBOL copybook needs exactly one line and has to describe nothing. The rule for the room: **if you have a copybook, take `flbcsv`. If you describe the structure yourself, take `CONV`.** Both write the same ZIP. The `flbcsv` call is deliberately that short -- it shows how little is necessary. A `ZIP()` object is not needed: the defaults fit here, and that a ZIP is to be produced is already said by the output name. And a `CCSID` does **not** belong here, for a reason worth knowing: `CCSID` in `flbcsv` is the character set of the **input**, that is of the binary data from the unload -- with a host unload `UTF-8` would simply be wrong there. The output side does not need the statement, because an ISO conformant CSV member in an ISO conformant ZIP is written in UTF-8 anyway.



All four from practical experience with the table support.



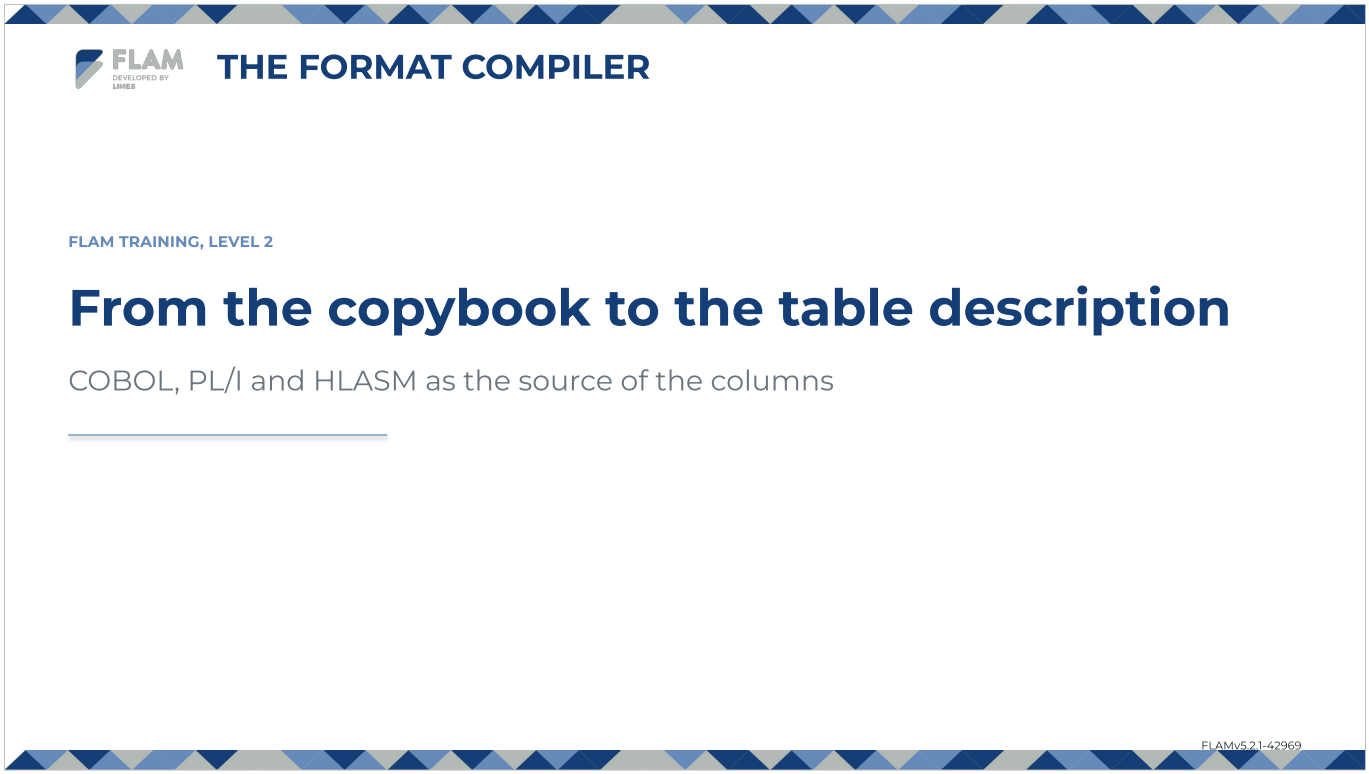
Twenty minutes at the machine. Step four is the learning effect: saving the schema and making it reproducible with BCSV.
Solution:
1. `flcl ACSV` detects by itself and needs no description. The result is good enough to see what is in the file -- but it is exploration, not production.
2. The detected schema can be written out via `ROWOUT`. That gives you a description that you can check and put under version control.
3. `flcl BCSV` with this schema via `FORMAT` delivers the same result -- but reproducibly, because nothing is guessed any more.
The expected insight, and it is the message of the unit: `ACSV` for understanding, `BCSV` for producing. Whoever puts `ACSV` into production has a detection in the critical path -- and it changes its verdict when the data change.

WHAT TO TAKE AWAY

- five stages per column: split, pre-process, type, post-process, reassemble
- flat formats from a length or a delimiter, hierarchical ones from a path
- detected when reading, adopted when writing - that halves the effort
- column encryption, masking and pattern checking sit below the application
- BCSV with a description, ACSV without, FCSV for existing CSV files

FLAMv5.21-42969

Walk through the learning objectives. That completes the conversion block; next come the utility functions.



This unit and the next belong together and answer the same question from two sides: who has the description of the data?
Here is the answer: you have it, in your own sources. A copybook lies in every host operation, a DSECT does too – only nobody has used it as a data description so far, but as a compilation unit. That is exactly what this unit turns around.



Goal two is what surprises participants most: the compiler puts bytes into the description that are not in the copybook.



The opening question, and it has exactly three answers. The middle one is this unit, the right-hand one is the next.
The left one – somebody writes the description by hand – is not wrong, but it is work, and it goes out of date. When a column in the copybook grows longer, the handwritten description is silently wrong.
That is why the order on the slide is a recommendation too: take the source if there is one. Have it detected if there is none. By hand only where neither works.

THREE LANGUAGES, ONE RESULT

LANGUAGE	SOURCE	WHAT THE TRANSLATOR RECOGNISES THEM BY
COBOL	Copybook, fixed or free format	by the level numbers and PIC clauses
PL/I	DCL/DECLARE, level 1 to 15	by the declaration itself
Assembler	HLASM DSECT with named DS fields	by DSECT and the DS fields

- **the file name plays no role** – the translators are tried one after another until one accepts the file
- on the host the descriptions are members of a PDS library and have no extension at all
- **C structures are foreseen but not yet included** – that is the honest answer today

A DSECT AS THE DATA DESCRIPTION – THE CALL SAYS NO MORE

```
flcl BCSV "input='KUNDEN.DAT' format='CUSTREC.asm' \
           reclen=37 output='kunden.zip'"
```

FLAMv5.21-42969

The table is the facts slide, and the most important sentence is below it:
the file name plays no role. FLAM simply tries the translators one after another and asks each of them:
do you recognise this file? The first one that says yes delivers the columns.
That is not convenience but consideration for the host. There copybooks and macros lie as members of a PDS library, and a member has no extension.
A recognition that hung on an extension would be useless on the host.
The order is COBOL, PL/I, assembler – COBOL first, because it is the most frequent case. Which one takes which file is normally not noticeable.
On C: structures from C headers are foreseen but not there. If somebody asks, that is the honest answer – and the outlook slide says why one more translator is not a large undertaking.

FLAM
DEVELOPED BY LIMES

FIELDS BECOME COLUMNS

THE DSECT AS IT LIES IN THE MACRO LIBRARY

```
CUSTREC  DSECT
CUSTID   DS    CL8
CUSTNAM  DS    CL20
CUSTBAL  DS    PL5
CUSTCNT  DS    H
         DS    XL1
```

WHAT THE COMPILER MAKES OF IT

```
col(name='CUSTID'  format.fix(length=8)  type.string())
col(name='CUSTNAM' format.fix(length=20) type.string())
col(name='CUSTBAL' format.fix(length=5)
    type.integer(format.bcd(signed type=PACKED)))
col(name='_ALIGN_33_' format.fix(length=1) drain skip)
col(name='CUSTCNT' format.fix(length=2)
    type.integer(format.bin(signed endian=BIG)))
col(name='_FIL_36_'  format.fix(length=1) drain skip)
```

`_ALIGN_33_` is not in the DSECT: the assembler puts the halfword on an even address. The record length is therefore 37 and not 36.

FLAMv5.21-42969

This is the core of the unit, and it is deliberately a real run: on the left the DSECT, on the right the description FLAM made from it.
Go through it field by field: CL8 becomes a string, PL5 a signed packed decimal, H a two-byte binary number in big-endian – the way the host writes it.
And then the point that sticks: between amount and counter there is a column that does not appear in the DSECT. The assembler puts a halfword on an even address, so a filler byte arises. FLAM knows that rule and enters it as a skipped column.
Whoever writes the description by hand has to think of that byte. Here the compiler thinks of it.

WHAT THE COMPILER BRINGS ALONG

- **Alignment** – filler bytes the compiler of the source language inserts, as skipped columns
- **Filler fields** – FILLER and unnamed fields stay in the description but not in the output
- **Repetitions** – `OCCURS` and `DIMENSION(n)` are unfolded into numbered columns, `MAXOCCURS` limits that
- **Alternatives** – `REDEFINES` and `UNION` each yield a row description of their own, `ORG` in assembler likewise
- **the record length** – it falls out of the description and is the number a flat file needs as `RECLEN`

> **ONLY FLAM CAN DO THIS**
>
> A structure that has lain in a library for thirty years becomes a data description **without one line of programming** – with exactly the filler bytes the compiler of that time produced.

FLAMv5.21-42969

The generalisation of the previous slide, and the slide people come back to later when a record length does not fit.
Alignment is a subject in all three languages: in PL/I it is called ALIGNED, in C it would be the same, in assembler it is not written down at all but follows from the type.
Filler fields: an unnamed DS, a FILLER in the copybook. They stay in the description so that the offsets are right – but they are not in the output. For COBOL that can be changed: FILLER=COLUMN, ALIGN or SKIP.
And the record length: it falls out of the description. That is the number to pass as RECLEN for a flat file. If it is wrong the first record overruns – and the error message looks exactly like that, it names both lengths.

COMPILE: THE DESCRIPTION IN THE CALL

FLCL CONV – THE SAME DESCRIPTION IN THE CONVERSION PATH

```
read.record(file='KUNDEN.DAT' recformat=FIX reclength=37
            table(compile(format='CUSTREC.asm' ccsid='IBM-1141')))
write.text(file='kunden.csv' table(format=csv))
```

SETTING	WHAT IT IS THERE FOR
CCSID	character set of the data; EBCDIC source, UTF-8 output
ENDIAN	byte order for COMP-5 – BIG reads host data on x86
MAXOCCURS	upper limit when unfolding repetitions
COLPATH / COLSEP	column names: whole path or the leaf only
TRIM / FILLER	padding characters and filler fields in the output
DETECT	do not fix character columns, have them detected

FLAMv5.21-42969

Now the classification: COMPILE is a sub-object of TABLE and stands there beside the handwritten ROW entries. Both together are allowed, and the order is fixed: first your own rows, then the compiled ones.
Several COMPILE objects are allowed, each with its own settings. That is needed when two copybooks come from different character sets.
The parameters in the list are the ones asked about in practice. CCSID is the most frequent: the data comes in EBCDIC, the output is to be UTF-8 – that is the one specification almost always needed.
ENDIAN concerns COMP-5 only. Regular binary fields are big-endian by COBOL standard, there the parameter changes nothing.

A CHARACTER FIELD IS NOT ALWAYS TEXT

- `CL20`, `PIC X(20)`, `CHAR(20)` are **byte fields** – the source gives the width, not the content
- without a specification it becomes `TYPE.STRING()`: safe, but a number then stays text
- with the switch it becomes `TYPE.DETECT()` **– number, string, date or hex binary, decided per value**
- `LOCALE` gives the notation: DE and US separate decimals and dates differently

IN CONV A SWITCH IN COMPILE, IN BCSV THE DETECT OBJECT

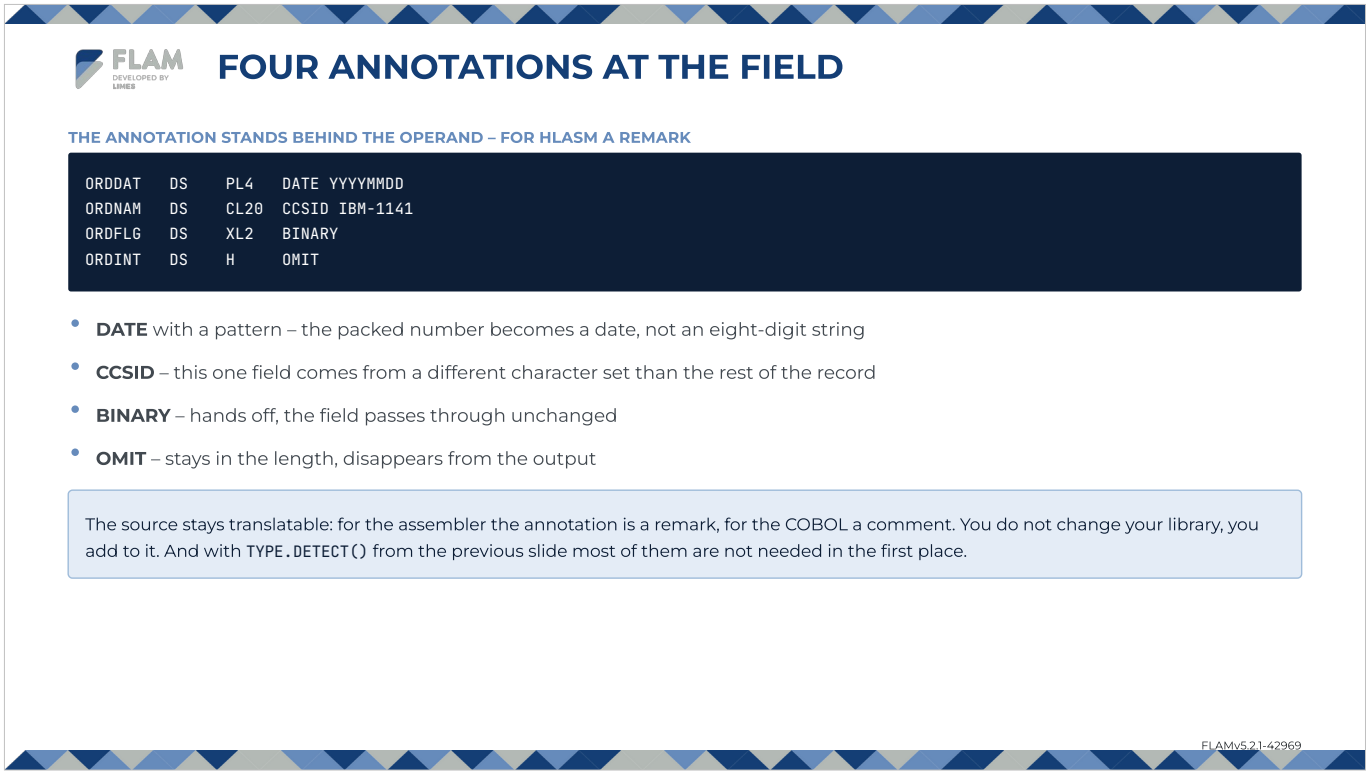
```
read.record(... table(compile(format='CUSTREC.asm' detect)))


flcl BCSV "input='KUNDEN.DAT' format='CUSTREC.asm' \
           reclen=37 detect() output='kunden.zip'"
```

In BCSV an empty `detect()` suffices; the hints inside the object then steer what may be read as a number, a date or binary.

FLAMv5.21-42969

The slide that saves the most trouble in practice, and it deliberately stands before the annotations – because it makes half of them unnecessary.
A `CL20` in assembler, a `PIC X(20)` in a copybook, a `CHAR(20)` in PL/I:
these are **byte fields**, not text fields. The source says only how wide the field is. What is in it, it does not say – and in holdings that have grown, everything is in there: a number as a digit string, a date, an identifier, sometimes really binary data.
Without a further specification the translator has to decide and takes `TYPE.STRING()`. That is the safe choice and often the worse one: a number that arrives as text stays text, and whoever wants to calculate or search with it afterwards starts over.
With the switch it becomes `TYPE.DETECT()`: FLAM looks at the **value** and decides per column whether a number, a string, a date or hex binary is present. `LOCALE` says in which notation to expect it – DE and US differ in the decimal separator and in the date, and that is exactly where a detection otherwise takes the wrong turn.
On the call: in the conversion path it is a switch in COMPILE. In BCSV the **specification of the DETECT object itself** switches it on – `detect()` alone suffices, and the hints inside it then steer the detection.



The slide that brings the most questions in a talk – because it solves a problem everybody knows. A copybook says PIC 9(8) or PL4. It does not say that a date is in there.
Nor does it say that this one text field comes from a different character set than the rest of the record. That information sits in the developer's head – or in the comment beside it.
That is exactly where FLAM reads it. In assembler it is the text behind the operand, in the copybook the comment. The source stays translatable:
for the assembler that is a remark, and remarks it ignores.
OMIT is the inconspicuous fourth and the most useful: the field stays in the length calculation but disappears from the output. That is how a field that must not be handed out is taken out without touching the record structure.
And the hint that takes the weight off this slide: whoever switches on `TYPE.DETECT()` no longer needs DATE and CCSID in most cases – the detection sees for itself that the value is a date. The annotations remain for the cases where it cannot be seen in the value, and for OMIT, which is a decision and not a detection.

IMS: ONE COPYBOOK PER SEGMENT

UNLOAD	WHAT IT IS	WHERE THE SEGMENT NAME COMES FROM
IMSL / IMSS	DFSURGU0, long or short	it is in the envelope – FLAM assigns it itself
IHDU	HD unload, DFSURGL0	a code; `* SEG-CODE X'NNNN'` above the 01 level
RDW / NONE	plain records of variable or fixed length	there is only one structure

ONE COPYBOOK PER SEGMENT, THE ENVELOPE OVERRIDABLE PER COPYBOOK

```
flcl BCSV "input['IMS.UNLOAD'] unload=IMSL \
           format['PART.cpy' 'SUB.cpy/%imss'] \
           output='segmente.zip'"
```

Without `UNLOAD=` somebody would have to take the prefixes of the logical records off by hand. The closing line FLAM appends itself.

FLAMv5.21-42969

The database unload is the case that makes it worthwhile, because here nobody has a description of the file – but everybody has the description of the segments.
An IMS unload is not one structure but many: one per segment type, all in the same file, each with an envelope in front. UNLOAD says which envelope it is.
For the two DFSURGU0 formats the segment name is in the envelope, so FLAM finds by itself which copybook belongs to which record. In the HD unload there is no name there but a code – hence the SEG-CODE annotation above the 01 level.
The last point is the one that counts in mixed holdings: the envelope can be overridden per copybook. A file in which one part comes from the database and another lies flat beside it goes through in one run.
And the closing line FLAM appends itself: a catch-all entry so that the file is read cleanly to the end.

COMPILE ONCE, USE OFTEN

THE FIRST RUN COMPILES, EVERY FURTHER ONE READS THE DESCRIPTION

```
flcl BCSV "input='KUNDEN.DAT' format='CUSTREC.asm' \
           rowout='custrec.row' output='kunden.zip'"

flcl BCSV "input='KUNDEN.DAT' rowin='custrec.row' \
           output='kunden.zip'"
```

- if `ROWIN` is set, `FORMAT` is not looked at – the compiler does not run
- the file is readable text: **look at it before ten million records hit it**
- it belongs to the processing and can be versioned, not to the single run
- **it ages silently** – if the copybook changes it has to be produced again

FLAMv5.21-42969

The operations slide. Compiling costs time, and in daily processing it costs it anew every day, although the copybook perhaps changes twice a year.
ROWOUT writes the compiled description out, ROWIN reads it back in. If ROWIN is set, FORMAT is not looked at at all.
The second benefit is the more important one and is at the bottom: the file is readable. You can look into it before letting ten million records loose on it – and you can touch it up when the compiler resolved something differently than you wanted.
That also makes the description something that can go into version control. It belongs to the processing, not to the run.
The catch, which belongs in the pitfalls: the file goes out of date silently when the copybook changes. Hence the last point.

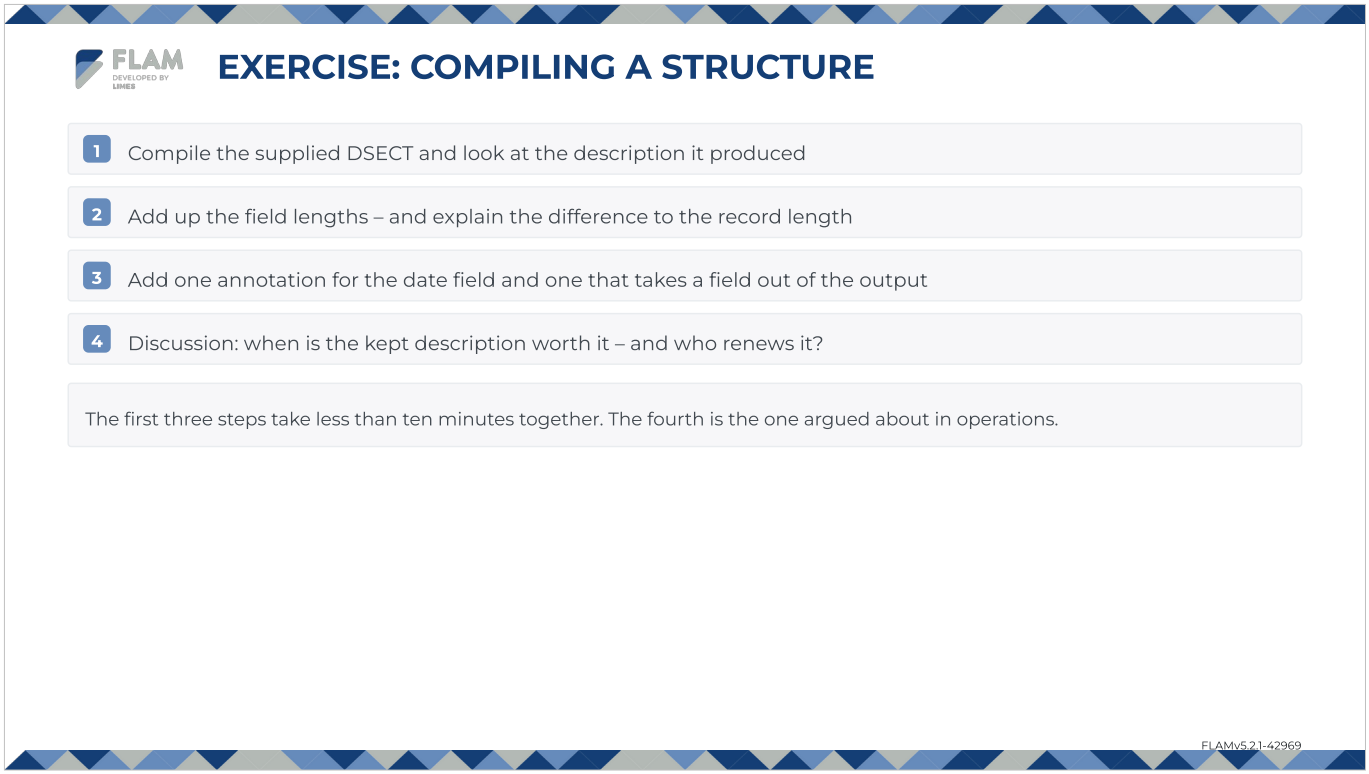
PITFALLS

SYMPTOM	CAUSE AND REMEDY
Record too long, on the first record	record length calculated without the alignment bytes – the message names both numbers
Syntax error although the command is right	no translator accepts the file – the description stays empty
Umlauts and signs wrong	`CCSID` missing – without it the character set of the environment applies
Numbers are wrong on x86	COMP-5 is platform dependent; `ENDIAN=BIG` for host data
DSECT does not go through	macros, conditional assembly, COPY and DC are not resolved

The compiler **reads** the source, it does not translate it: what only the assembler or the COBOL compiler would produce is not available to it.

FLAMv5.21-42969

The five that really occur in support cases.
The record length is the most common: the description yields 37, the file is read with 36, and the first record overruns. The message names both numbers – that is the fastest diagnosis in this whole area.
The second is the one people search longest for: if no translator accepts the file, the description stays empty and the error reads like a syntax error in the command. The question is then not the command but the file – usually a source that a macro or a COPY would first have to complete.
The character set is the silent error: without CCSID the character set of the environment is assumed. On the host that fits by chance, on the Linux machine beside it no longer.
And the last one, because with HLASM it is different from what is expected: the compiler reads the source, it does not assemble it.
Conditional assembly, COPY, USING and DC values are not resolved. A DSECT that assembles its fields from macros does not go through.



Solution:
First: `flcl BCSV "input='CUSTOMER.DAT' format='CUSTREC.asm' rowout='custrec.row' output='customer.zip'"` – and then look into custrec.row.
Second: the sum of the field lengths is 36, the description says 37. The difference is the alignment byte before the halfword; it turns up as _ALIGN_33_.
Third: write `DATE YYYYMMDD` behind the date field and `OMIT` behind the internal one. Then the same run again, and compare the two CSV files – one column less, one column as a date.
Fourth, the discussion question: ROWIN, when the structure is stable and processed daily. But then with a rule about who produces the file again when the copybook changes. Without that rule the kept copy is more dangerous than the lost run.



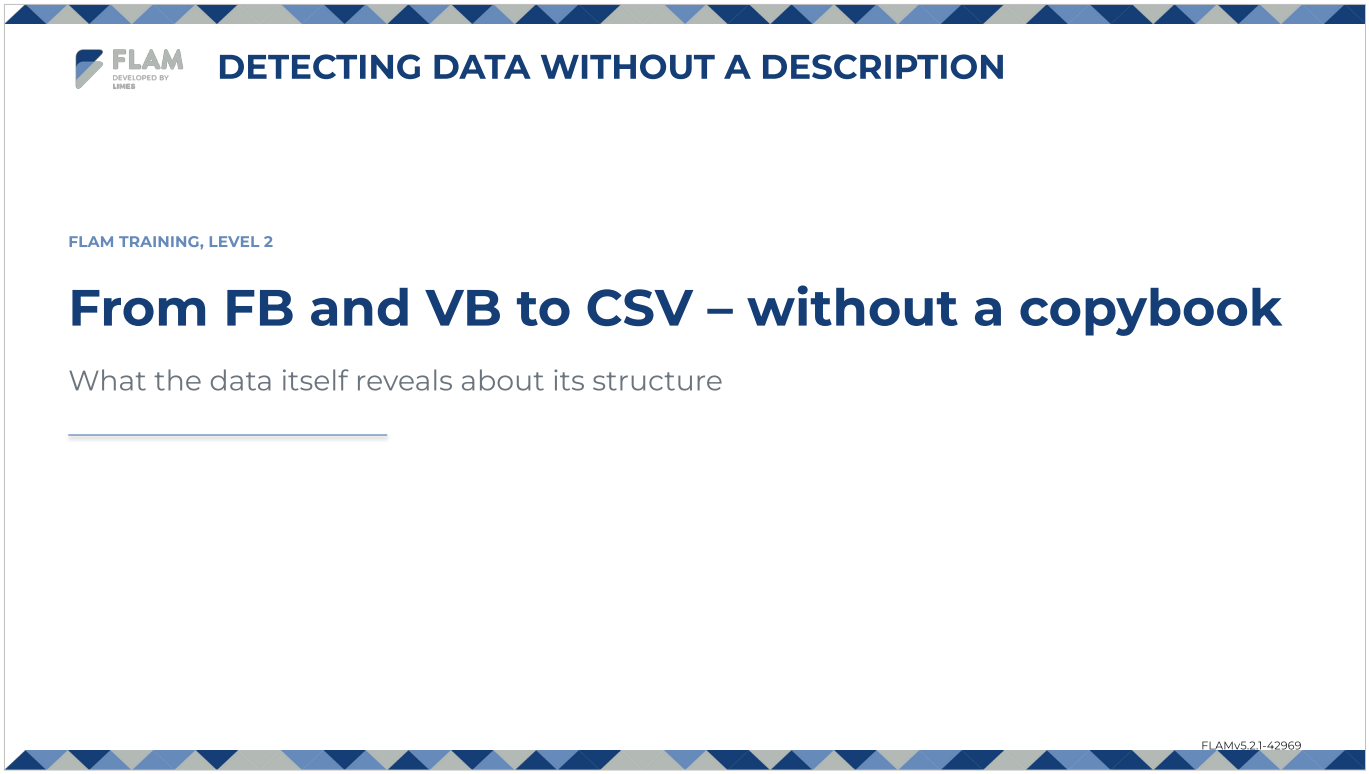
The slide that shows no special case was built here but a place to dock on to.
Technically it is a list of translators asked one after another: "do you recognise this file?" Whoever says yes delivers rows and columns.
Everything after that – types, restructuring, CSV, XML, archive – is the same for all of them. A new translator is therefore exactly one function and not a new product.
Hence the status column: C and C++ is the next one, its slot is already in the list. XSD is decided and is more than a structure description: an XML schema also describes the **permitted values**, down to the regular expression. It is intended above all for payments – SEPA, SWIFT MX and whatever else comes in this form. That makes it not only a compiler but also a check.
DFDL and ASN.1 are under consideration, not promised. DFDL is interesting because it was invented for exactly this purpose: a language to describe foreign formats. ASN.1 comes from telecommunications and the certificate world and is the description there.
If somebody in the talk asks about a format that is not listed: that is the right question, and the honest answer is that it is a matter of effort, not of architecture.

WHAT TO TAKE AWAY

- three languages, tried one after another – the file name plays no role
- character fields are byte fields: with `TYPE.DETECT()` the value decides, not a default
- the compiler brings along what is not in the source: alignment, filler fields, record length
- four annotations at the field steer it without making the source untranslatable
- IMS unloads get one copybook per segment, the envelope is overridable
- compiled once, kept and used again – with a rule for renewing it

And if there is no source any more? Then FLAM detects the structure from the data – the next unit.

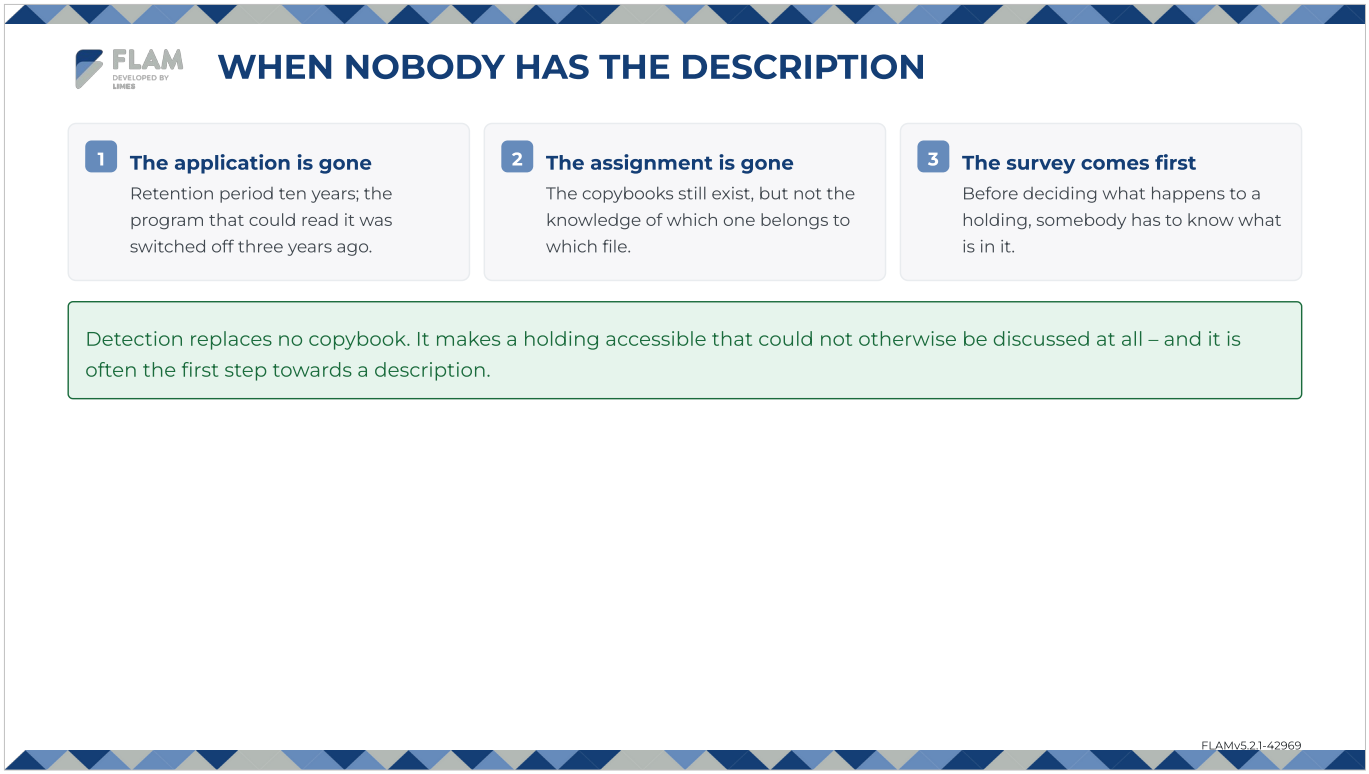
Summary and transition. The last point is the bridge to the next unit: there is no source there any more, and a table arises all the same.



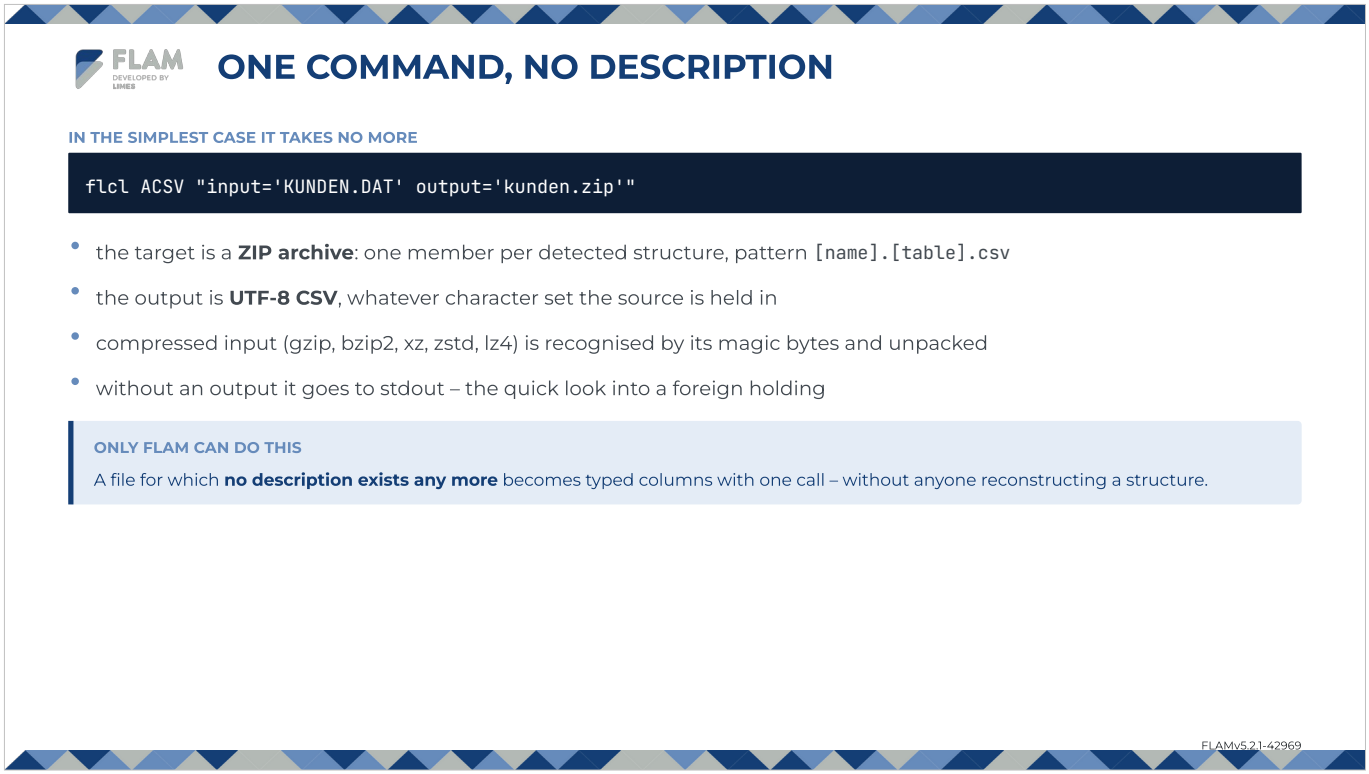
The sequel to the previous unit, with the third answer to the same question: nobody has the description.
This is not an edge case. The application has been switched off, the library holding the copybooks did not travel with it, the developer has retired – and the file is still there, because it has to be kept for ten years.
Important for expectations, and I would rather say it at the start than when asked: this is estimated, not known. The unit shows just as plainly where the estimate ends.



Goal three is the honest one: whoever knows the limit sells the result properly. Whoever does not will be caught out by it.



The problem slide. Three cases, and all three come from real customer conversations – without naming anyone.
The first is the most common and the most annoying: the data has to be kept, the program that could read it does not.
The second is the handover case. The copybooks still exist, but not the knowledge of which one belongs to which file – and trying them out costs more than detecting.
The third stands at the beginning of a project: before deciding what happens to a holding, somebody has to know what is in it.
The point at the bottom is the one to hold on to: detection does not replace the copybook. It makes the file accessible.



The simplest slide of the unit, and in a talk it works best when kept really short: input, output, done.
The target is a ZIP archive, because one file may hold several tables – one member per detected structure. The pattern for the member name can be set.
That without an output the result goes to stdout is handy for a quick look: run it over a few records and read them before touching the whole holding.



The slide that settles the scope before going into depth – otherwise half the room thinks this is only about host files of fixed record length.
There are three detectors under one command. Which one applies is decided by the data; but it can also be said when it is known.
The fixed one is the hardest and therefore dominates the rest of the unit.
It also delivers several record types out of one file – one row description per record type, which is exactly what an overlay in a copybook would be.
The character-separated one is the most comfortable, and the example shows it: here the column boundary lies open, and if there is a header line even the **names** are in the file. On top of that comes the type per column – in the example the detector found the semicolon as the separator, the comma as the decimal separator and the date pattern by itself.
The third reads XML, where the structure sits in the path.
If anybody asks why these are separate at all: because the signals differ.
In the character-separated case the boundary is a character, in the fixed case it has to be inferred from the bytes. That is a different trade, even though the output is the same.

WHAT MARKS A COLUMN BOUNDARY

SIGNAL	WHAT IT REVEALS
Type change	text beside a packed number beside a binary field – the byte classes differ
delimiter	with character-separated data the boundary lies open
Constant filling	a field carrying the same value in every record
Padding	trailing blanks, leading zeros – **one single** record that pads is enough

A record that pads **proposes a boundary** – it becomes valid only once the split works out for every record. Evidence and verification, not a majority.

FLAMv5.21-42969

Now the how, and this is the slide after which the unit has been understood or it has not.
The detector reads bytes. It can place a column boundary only where the data itself reveals something. Four things reveal something, and they are in the table.
The type change is the strongest: between text and a packed number there is a boundary, because the byte classes differ.
The padding characters are the subtle case and the most frequent gain: a text field that is not completely filled ends in blanks, a digit string begins with leading zeros.
And now the point that matters and is often expected the other way round:
one single record is enough for that. Seven records that fill every field to the brim say nothing about where the boundary is – but they do not contradict it either. The one record that pads knows more than all the others together, and if the split it proposes works out for every record, then it holds.
That is a decision on evidence, not a vote. A majority would answer the wrong question here.
The character set is the fourth, and it works twice over: it decides how the content is read, and for fixed-length records also where the boundary lies – because the detector separates text from non-text byte by byte.



The comparison, and it is a real run over the same 37-byte file – on the left with the DSECT from the previous unit, on the right without any description at all.
What went well: the packed amount was recognised as a signed packed decimal, with the right value and the right sign. The record length is correct.
What could not go well: the names. They are nowhere in the file. And the customer number was split, because its first seven digits were identical in every record looked at – a constant becomes a column of its own.
This is the point to pause at in a talk: both are right. The right-hand side is not the worse solution, it is a solution for a different case.
Whoever has the DSECT takes the DSECT.

WHERE DETECTION ENDS

- several **numeric fields** side by side, same character set, no separator – become one wide number
- several **text fields** side by side, filled completely in **every** record – become one wide text field
- **packed or binary fields** with no mark of their own – indistinguishable from one wider field of the same type
- **column names** do not exist: they are in no record file, only in the description

That is not the limit of the method but the limit of the information in the file. For a guaranteed correct column structure the way leads through the description – see the previous unit.

FLAMv5.21-42969

The limits slide, and it deliberately comes early – not at the end, where it would look like small print.
Three cases, and all three have the same cause: two neighbouring fields of the same type without a separator cannot be told apart in the bytes. Five numeric fields in a row become one wide number, two text fields become one wide text field.
The word that matters in the second line is **every**: one single record that does not fill a field completely is enough to make the boundary visible. Only when not a single record pads is it really not there.
That is not a weakness of the implementation, that is the information missing from the file. A human being with the same bytes and no copybook cannot decide it either.
The value of the slide in a sales conversation is exactly this: we say in advance what does not work. That is what makes the promise for the rest credible.
And the consequence is at the bottom: for a guaranteed correct column structure the way leads through the description. Detection is the way there, not the substitute for it.

HOW MANY RECORDS ARE LOOKED AT

SETTING	DEFAULT	WHAT FOR
MAXREC	8	how many records are looked at; FULL = the whole first block
MINREC	2	how many of them have to agree on the same structure

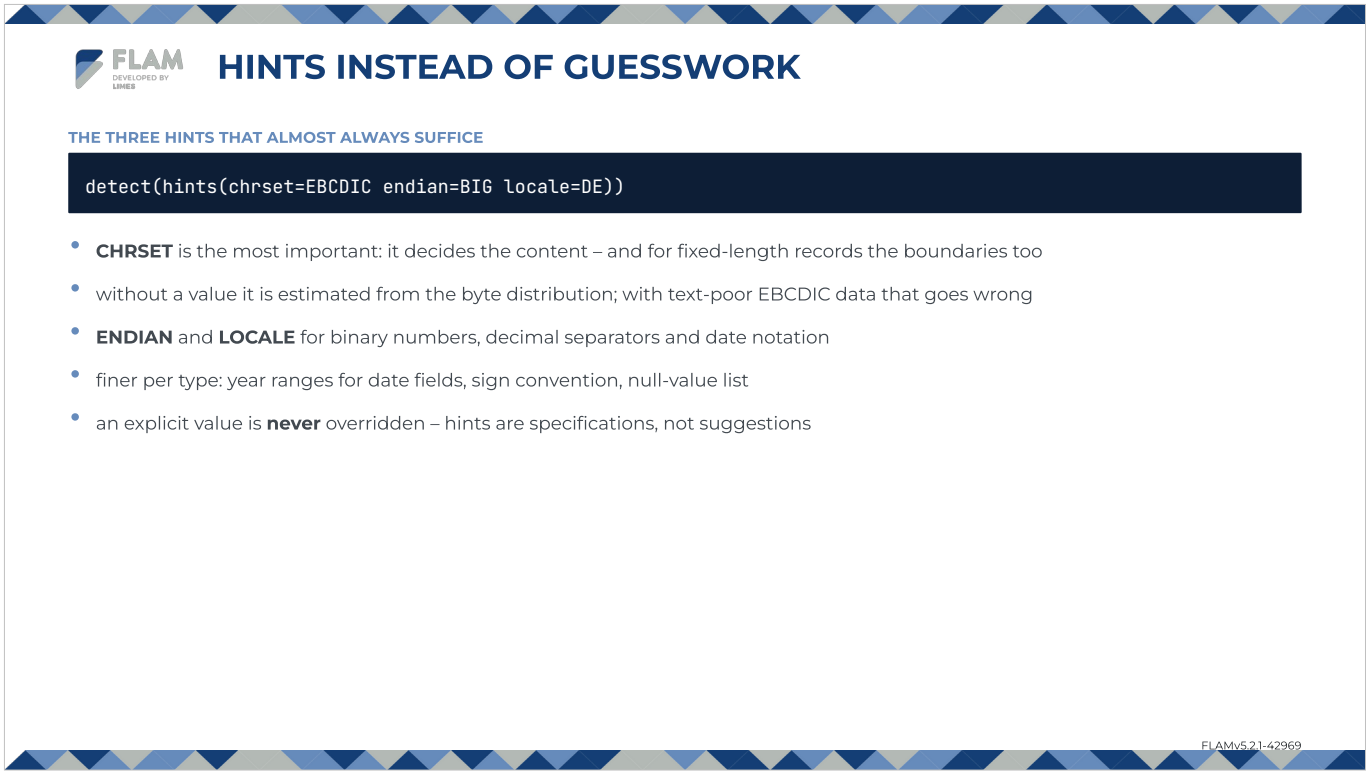
LOOK AT MORE RECORDS, DEMAND A STRICTER AGREEMENT

```
flcl ACSV "input='KUNDEN.DAT' output='kunden.zip' \
           detect(maxrec=FULL minrec=4)"
```

More records also means **more chances** here: the larger the sample, the more likely one record does not fill a field completely – and shows where it ends.

FLAMv5.21-42969

The adjustment that achieves the most and is known the least.
The default is eight records. For a uniform file that is right and costs nothing. For a file with several record types it is too few – and you notice it because one record type does not appear at all.
The second number is the confidence threshold: how many of the records looked at have to agree. The default is two. That is why a header record that looks different from the rest does not break the detection.
The example from the manual is good for telling: eight records looked at, seven of length 80, one of length 20. Detected is 80 – the header record does not interfere.
And the connection to the previous slide: more records do not hurt the padding detection. On the contrary – the more records are looked at, the more likely one of them does not fill a field completely and shows where it ends.



The slide for the person who already knows something about their data – and that is almost always somebody in the room.
The character set is the most important hint, for a reason worth explaining: it decides not only how the content is read but, for fixed-length records, also where the boundaries lie. The detector separates text from non-text byte by byte, and which bytes are text depends on the character set. Without a value it is estimated from the byte distribution, and with EBCDIC data that holds little text that readily goes wrong – because in EBCDIC nearly everything from 0x40 up looks printable, so binary fields fall apart like text.
Hence the rule: if you know it, say it. An explicit value is never overridden.
The remaining hints are finer – per type it can be narrowed what comes into question at all, down to year ranges for date fields and the list of strings that count as a null value.

VARIABLE RECORDS AND IMS UNLOADS

NOTE	WHAT IS READ WITH IT
NONE	default – the envelope is recognised by itself
RDW	plain variable-length records with a length field
IMSL / IMSS	IMS DFSURGU0, long or short envelope
IHDU	IMS HD unload, DFSURGL0 – the physical image

AN IMS UNLOAD WITHOUT A SINGLE SEGMENT DESCRIPTION

```
flcl ACSV "input='IMS.UNLOAD' output='segmente.zip' \
           detect(format=IMSS hints(chrset=EBCDIC))"
```

Without the hint the envelope is usually recognised by itself. The short DFSURGU0 envelope is the case where the marker often does not fall into the sample.

FLAMv5.21-42969

The part that makes the difference in a host environment.
A file with variable records carries a length field before each record, and an IMS unload carries an envelope with a segment name or segment code on top. Whoever does not take that off detects nonsense: the envelope looks like the first columns.
FLAM normally recognises these envelopes by itself. The hint is for the cases where the sample is too thin – the manual names the short DFSURGU0 envelope, whose marker rarely turns up in the first block.
Practical advice for operations: if you know which unload format you have, write it down. The detection is then faster and certain, and afterwards nobody argues about a run that cannot be repeated.

DETECT, LOOK, WRITE DOWN

DETECT ONCE – THEN WORK WITH THE CHECKED DESCRIPTION

```
flcl ACSV "input='KUNDEN.DAT' output='kunden.zip' \
           detect(maxrec=FULL rowout='kunden.row')"

flcl BCSV "input='KUNDEN.DAT' rowin='kunden.row' \
           output='kunden.zip'"
```

1. let it detect and write out the description it found
2. look in and correct: name columns, separate what was merged, put types right
3. use the written-down description from then on – nothing is estimated any more

The file holds **more than one** description: whatever else came into question stands below it as a comment, each variant with its rating. Correcting therefore often means no more than choosing.

FLAMv5.21-42969

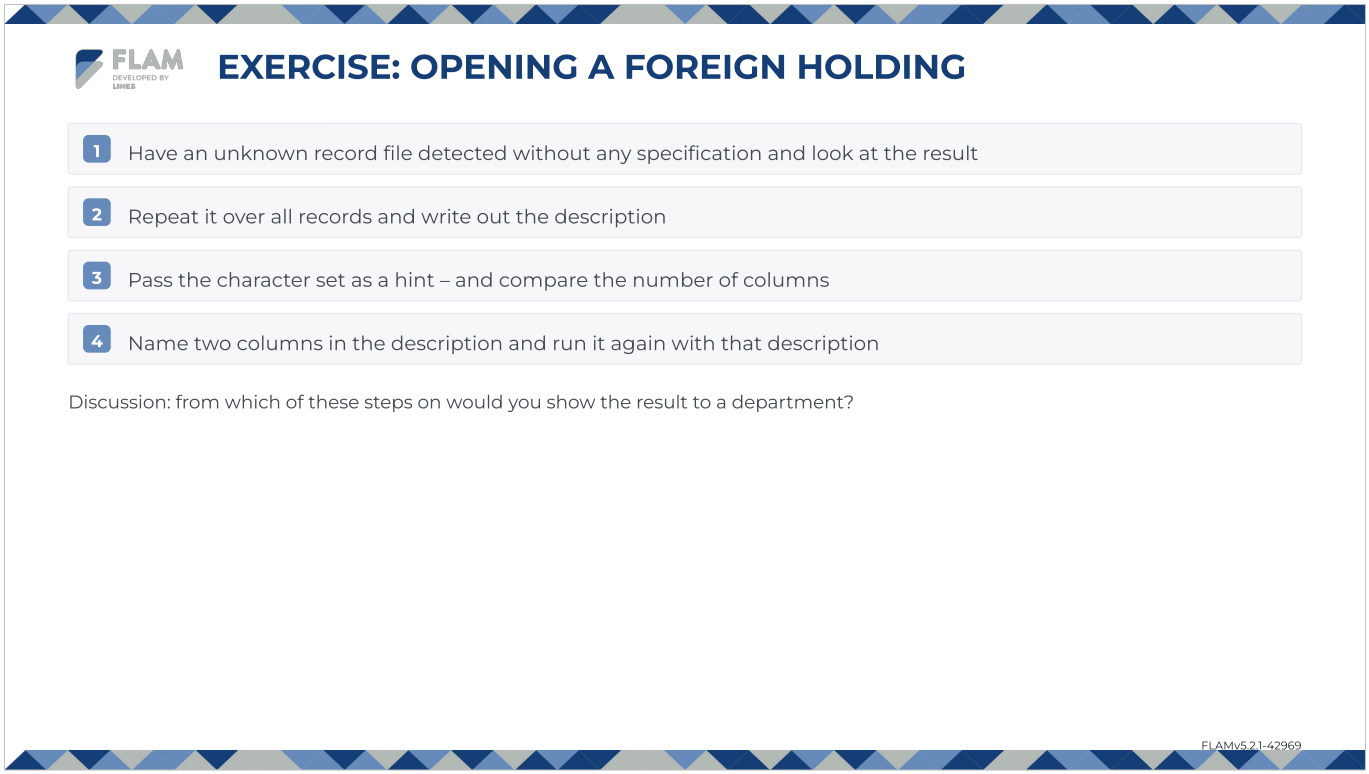
The slide that ties the two units together, and the advice to give for operations.
On request the detection writes out the description it found. That file is readable text and exactly the format the previous unit knows as a kept description.
This turns estimating into a procedure with three steps: let it detect once, look in and correct – name columns, separate two merged fields, put a type right – and work with the written-down description from then on.
The gain is twofold. The daily processing no longer estimates, it reads a fixed description. And there is a description of the data again, one that can go into version control – for a holding that had none.
That is the sentence this unit is sold with.
And the point that makes correcting practicable in the first place: the file holds more than one description. Whatever else came into question stands below it as a comment, each variant with its rating. Whoever corrects therefore does not guess but chooses – and sees how close the decision was. For a file whose structure is unambiguous there is only one line.

FLAM DEVELOPED BY LIMES

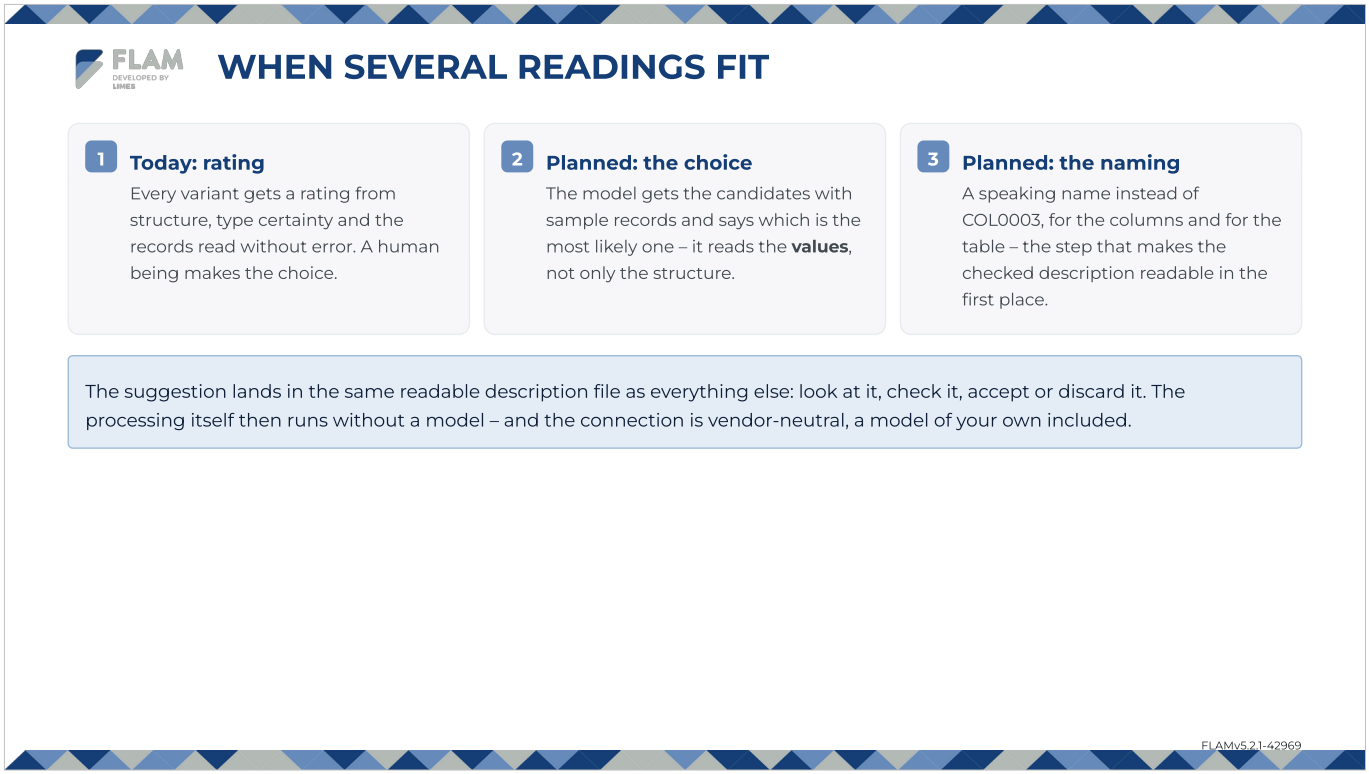
PITFALLS

SYMPTOM	CAUSE AND REMEDY
Structure looks consistent, values are nonsense	record length assumed wrongly – the records lie shifted against each other
Binary fields fall apart into many text columns	character set estimated as EBCDIC or the other way round – set `HINTS.CHRSET`
A record type is missing from the output	sample too small – raise `MAXREC`, `FULL` if need be
Columns with a fixed value and a condition	constant field in too small a sample – look at more records
A different number of columns every day	detected anew every day – detect once, check, write it down

The five from practice.
The record length is the most common and the least conspicuous: if it is assumed wrongly, the detected structure is consistent in itself and still completely wrong, because the records lie shifted against each other. You see it only by looking at the output.
The character set is the second and works twice over, as discussed.
The third is the underestimated one: several record types in one file, and the sample sees only the first. Recognisable because a member is missing.
The fourth we know from the comparison: constant fields become columns of their own with a condition on them. In a small sample every value is easily constant.
And the fifth is the procedural trap: whoever detects anew every day may get a different number of columns every day – because the day's data carries different signals. Daily processing belongs on a written-down description.



Solution:
First: `flcl ACSV "input='UNKNOWN.DAT' output='x.zip'"` and look into the members. Note the number of columns.
Second: the same run with `detect(maxrec=FULL rowout='x.row')`. As a rule the number of columns changes, because more records show more cases. Put both descriptions side by side.
Third: repeat it with `detect(hints(chrset=EBCDIC))`. With EBCDIC data the number of columns usually drops markedly, because binary zones no longer fall apart as text.
Fourth: name two columns in x.row that you recognise and repeat the run with `rowin`. Now there are names in the header line.
The discussion question at the end: from which step on would you show this result to a department? The expected answer is – after step four, not after step one.



The future slide, and it picks up exactly what was already there two slides earlier: the alternatives in the ROWOUT.
Today it is an arithmetic judgement. The detector rates every variant by structure, type certainty and by how many records pass through it without error – and lists them in order. A human being makes the decision.
That is precisely where a language model can do something a formula cannot: it sees not only the structure but the **values**. Whether column three holds account numbers or quantities, whether the number in column five could be a date, whether a split makes sense in business terms – that is reading, not calculating.
Hence the two planned steps. First: the model gets all candidates together with a few records as a table and says which is the most likely. Second, and in daily work perhaps the more valuable one: it proposes **speaking names**. COL0003 becomes ACCOUNT_NUMBER, the record a name like POSTING.
What matters here, and it belongs in the talk: the suggestion goes into the same readable description file as everything else. So nobody decides behind the back of the operation – the suggestion is seen, checked, accepted or not. The processing itself then runs without a model.
The connection is deliberately vendor-neutral and can also run against a locally operated model. With customer data that is regularly the first question asked.

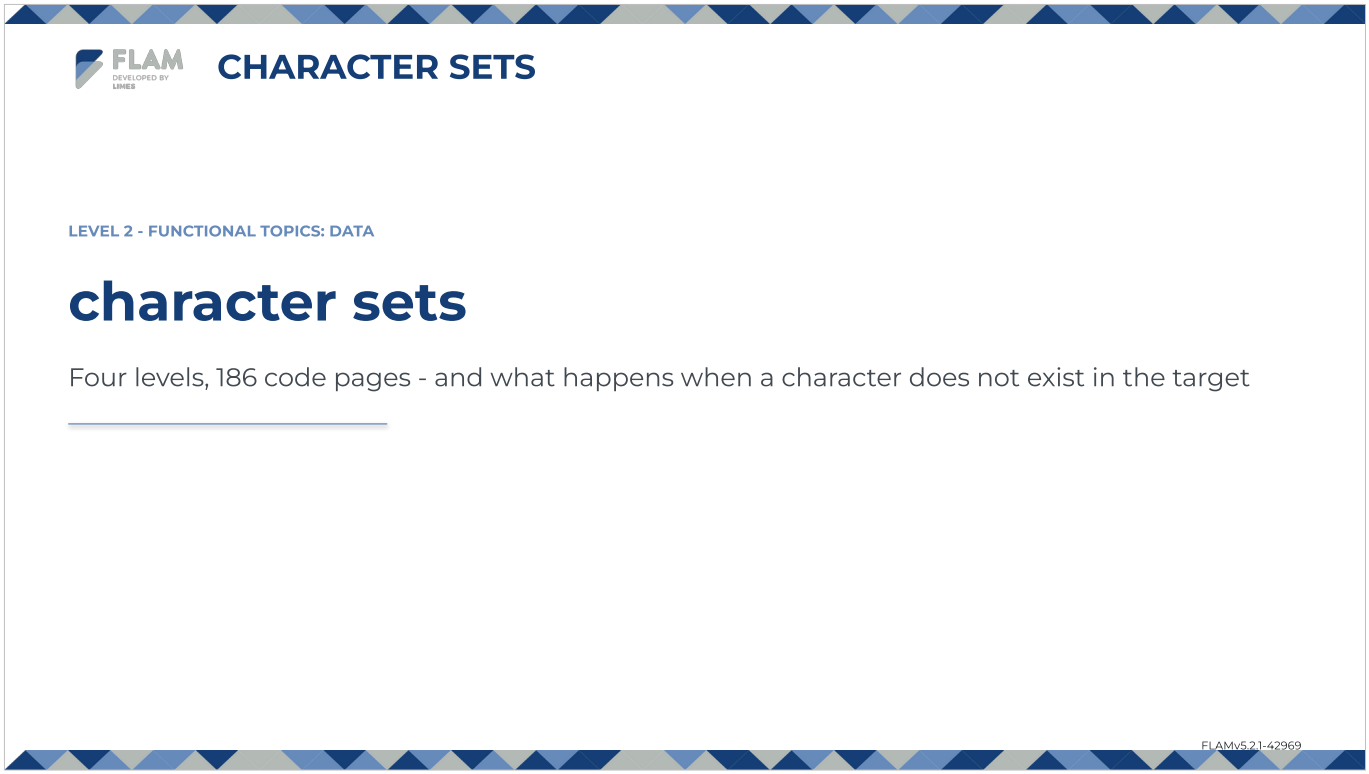
WHAT TO TAKE AWAY

- one call brings an undescribed record file to CSV – an IMS unload too
- three detectors under one command: fixed-length records, character-separated data, XML
- boundaries arise where the data carries a signal: type change, separator, filling, padding
- what carries no signal merges – that is the limit of the information, not of the method
- hints narrow the search space; the character set is the most effective one
- sample size and agreement threshold decide what is found at all
- ambiguous cases yield several rated readings – in future a model will propose the most likely one and speaking names

Detect once, check, write it down – after that nothing is estimated any more. A holding without a description ends up having one again.

FLAMv5.21-42969

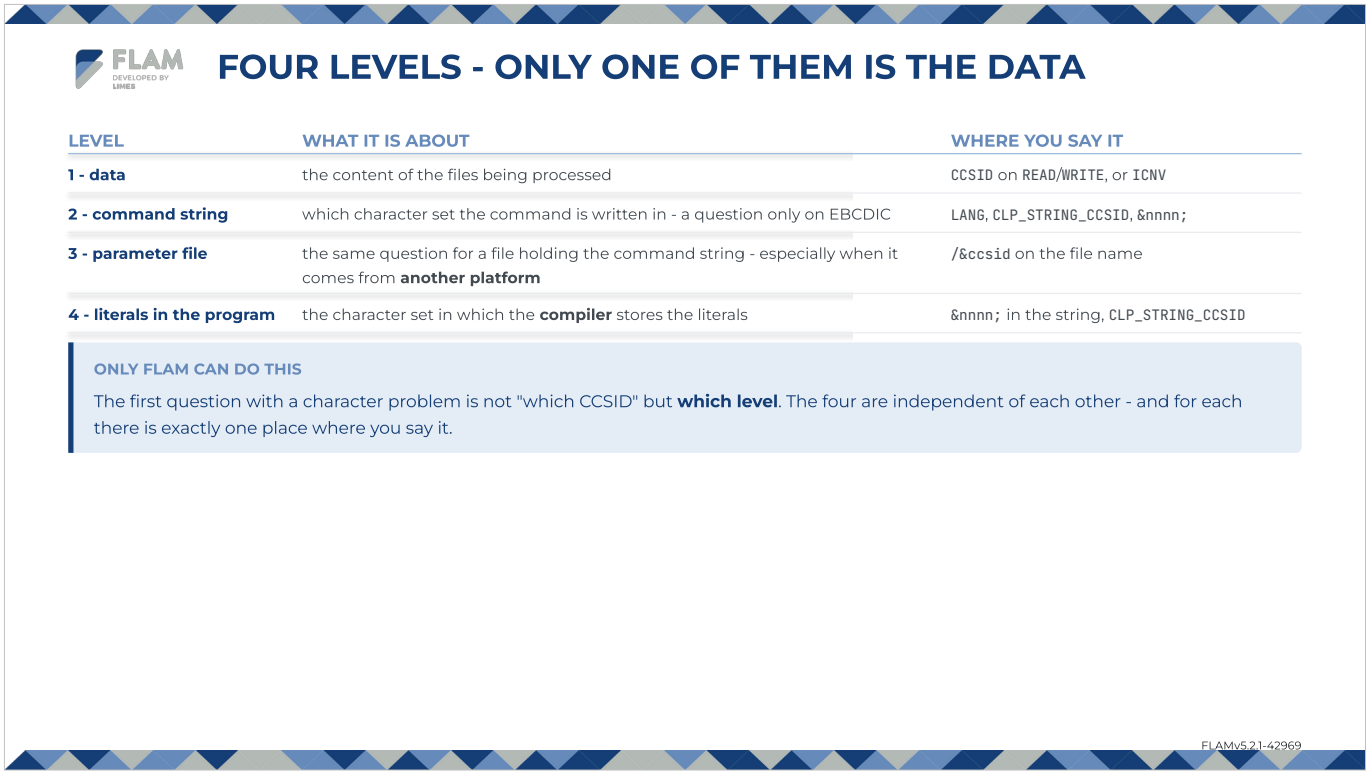
Summary. The last point is the one that ties the two units together and should be remembered: detection is the way to a description, not the substitute for one.



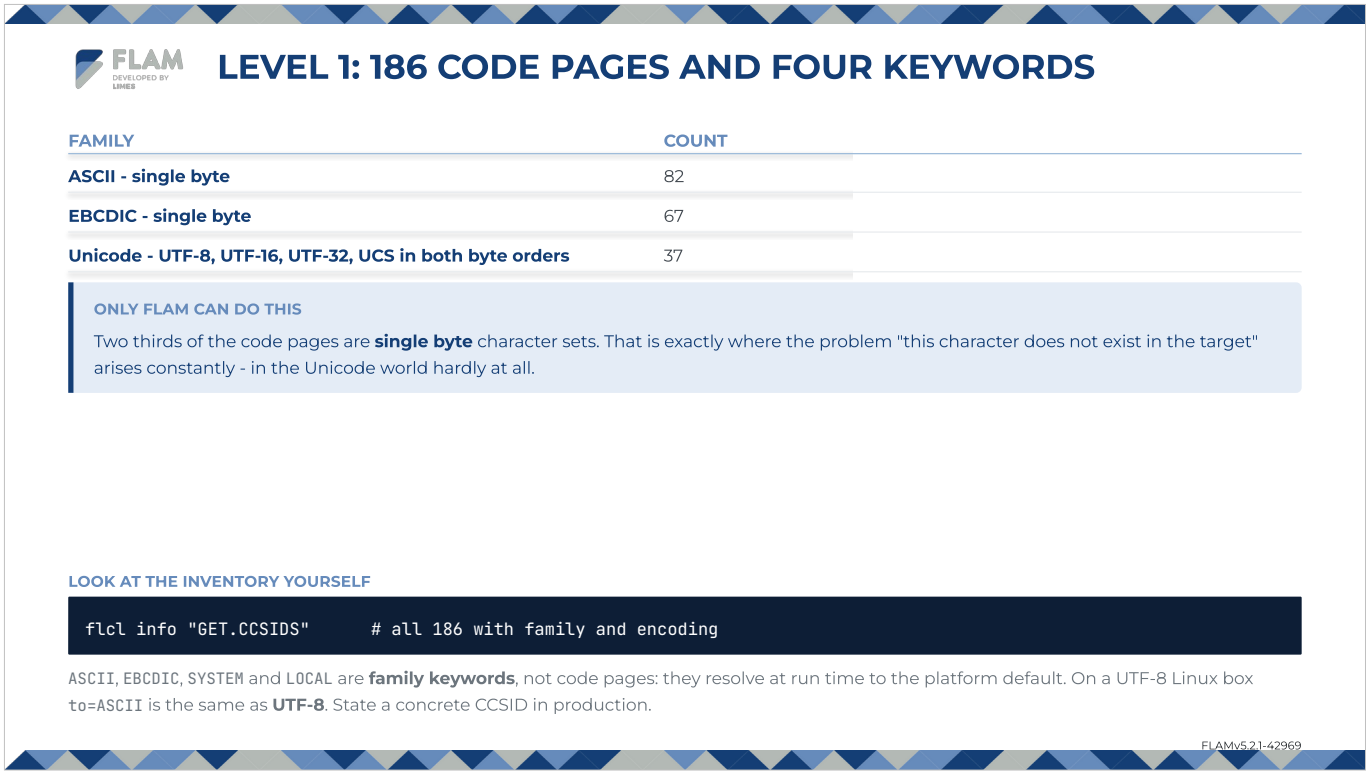
The topic that causes the most data errors in practice -- and the only one where an error is often noticed first by the recipient. This unit goes considerably beyond what was in the conversion unit. There it was about the CCSID as a statement:
which file is in which character set. Here there are two more things: what happens **in between** when a character does not exist in the target -- and the fact that the character set is not stated in **one** place but in **four**. Important right at the start: **we stopped using libiconv long ago and work with our own module.** That is not a detail, because the capabilities of this unit follow from it: normalisation, subsets, best fit mapping, custom mappings, reporting and casing. A library that only re-encodes cannot do that. The unit *Conversion with FLUC* is a prerequisite.



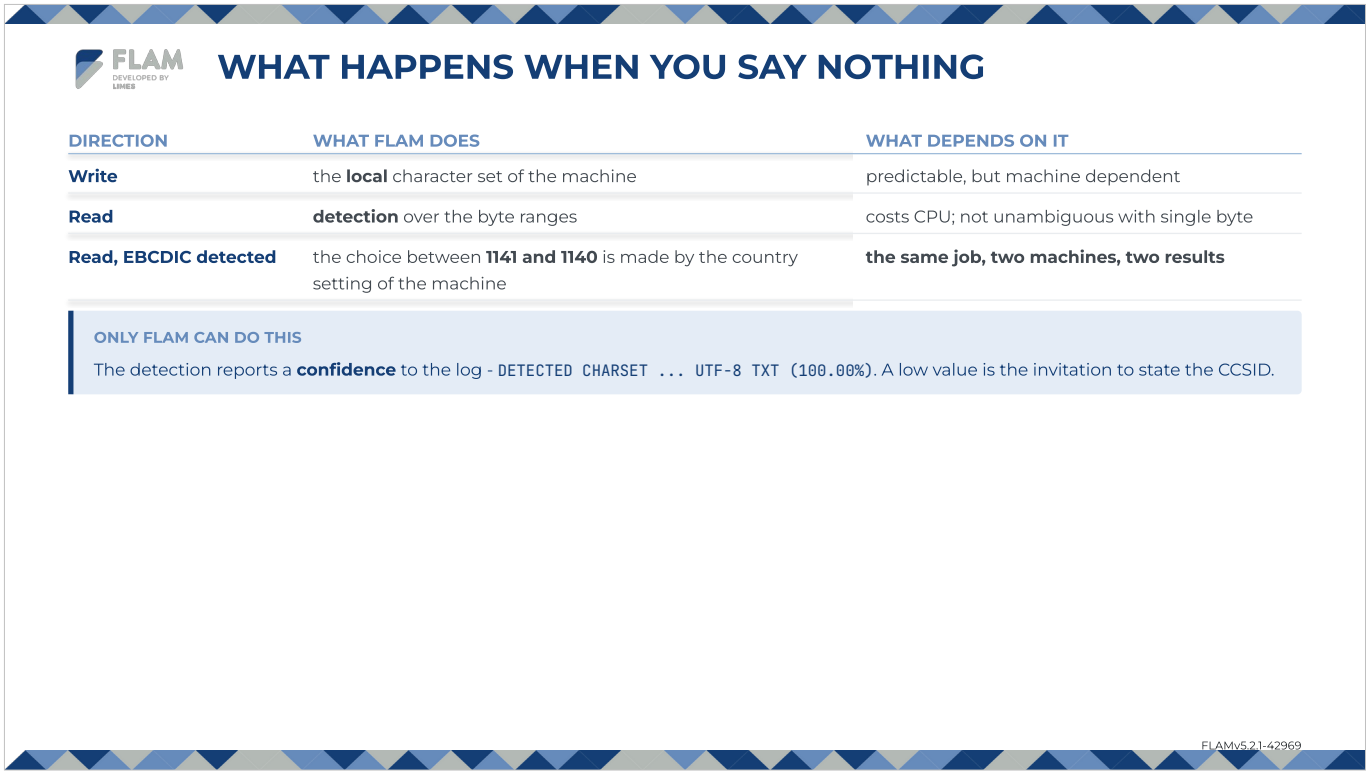
The learning objectives as a contract. The first one is what holds the unit together: whoever keeps the four levels apart asks the right question when something goes wrong -- namely **which** level is affected. The fourth is what the data part leads up to: there is not **one** correct re-encoding, there is a business decision, and the tool has a switch for each of them. The sixth is the most important one for a house in payment traffic -- and it is measured on a slide of its own, not asserted.



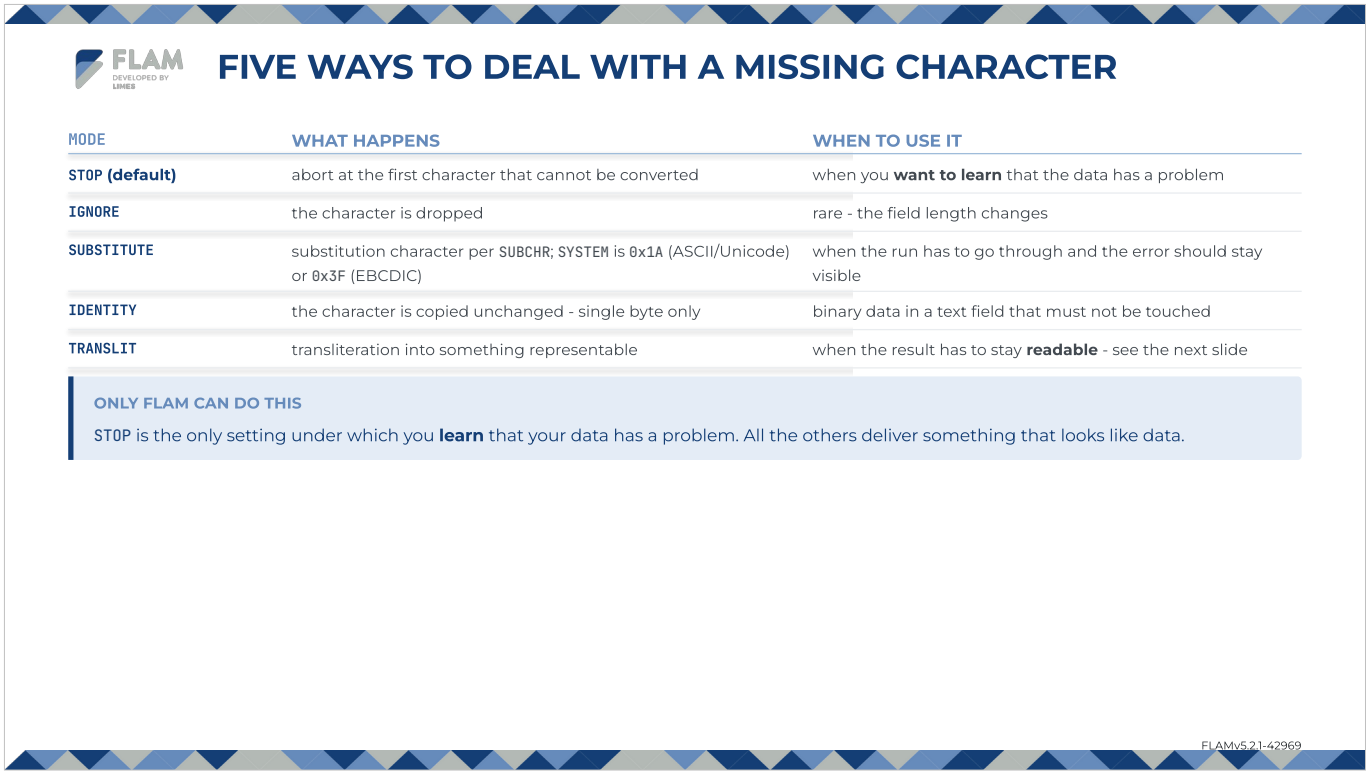
The map for the whole unit, and the slide to open when something goes wrong with characters in production. The first question is never "which CCSID" but **which level**. **Level 1, the data.** The content of the files we process. That is what one thinks of first, and the largest part of this unit is about it: detection, error behaviour, transliteration, subsets, casing, normalisation.
Level 2, the command string. Which character set is the `flcl` command itself written in? On ASCII and UTF-8 platforms that is not a question. On EBCDIC it is one, because brackets, tilde and dollar sit in different places depending on the code page. **Level 3, the parameter file.** The same question for a file that holds the command string -- and it gets sharper as soon as the file comes from **another platform**, fetched from the host over SSH for instance. **Level 4, the literals in the program.** Anyone programming against our APIs has a character set in which the **compiler** stores its literals -- and it can be a different one from that of the variables at run time. These four are independent of each other, and that is exactly what makes the topic complex.
The good news: for each level there is exactly one place where you say it. They are on the following slides.



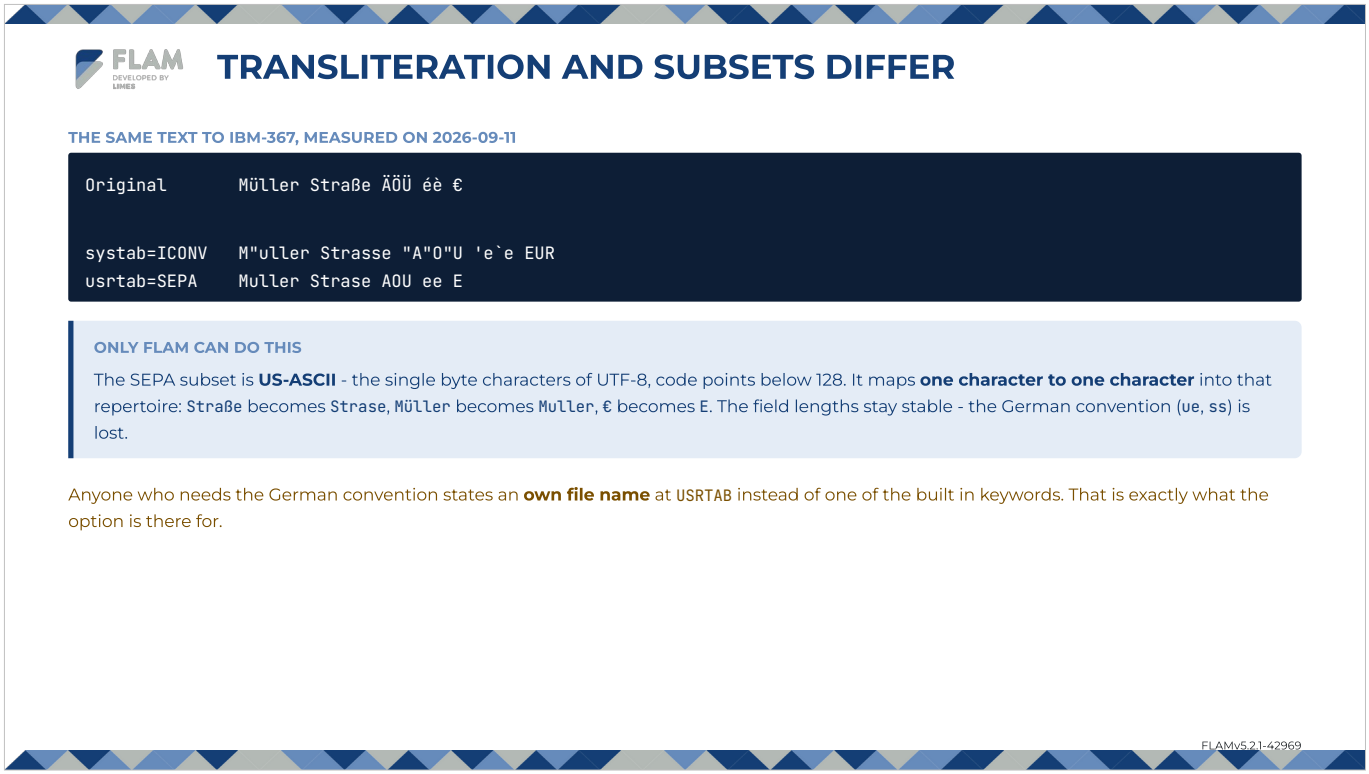
From here on, the data. First the inventory, then the trap. FLAM knows **186 code pages**, and you can look at them yourself:
`flcl INFO "GET.CCSIDS"` lists them with number, character set family and encoding string. The distribution is telling: **82 ASCII, 67 EBCDIC**, the rest UTF-8, UTF-16, UTF-32 and the UCS variants in both byte orders. That two thirds of the code pages are single byte character sets explains why this topic is so large in the first place: in the Unicode world the problem "this character does not exist in the target" hardly arises, in the single byte world it arises constantly. Now the trap, and it regularly costs half a day: besides the concrete code pages there are **family keywords** -- `ASCII`, `EBCDIC`, `SYSTEM` and `LOCAL`. They are convenient, but they are **not** a code page; they resolve at run time to the default of the platform.
Concretely: on a UTF-8 Linux box `to=ASCII` resolves to **UTF-8** -- not to 7 bit ASCII. I measured that myself, and at first I thought the tool was ignoring the parameter. It does the right thing, it simply means something other than one assumes. The rule that follows: **state a concrete CCSID in production.** If you want 7 bit, say `IBM-367` and not `ASCII`. If you want 1141, say `1141` and not `EBCDIC`.



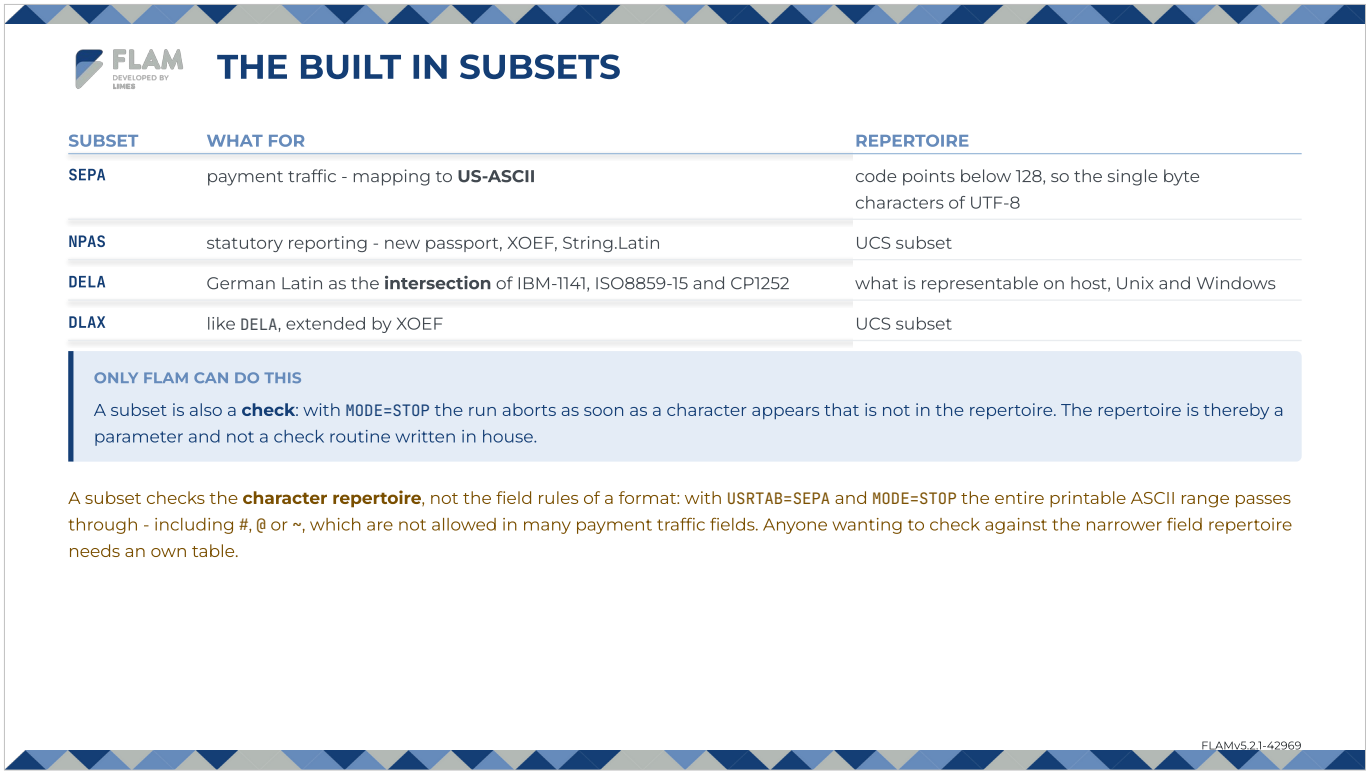
The slide that anticipates a frequent question: "I did not state a CCSID and it worked -- so why should I state one?" **On writing** FLAM takes the local character set of the machine. That is mostly right and always predictable. **On reading** it is detected. And the detection is good, but it is neither free nor unambiguous: it costs CPU -- every byte is examined. With single byte character sets it is **not unambiguous**. EBCDIC can be told apart from ASCII and UTF-8 well, because the byte ranges differ. But whether it is German or American EBCDIC is not in the bytes -- 1141 and 1140 differ in only a few positions. FLAM therefore decides that from the **country setting of the machine**: on a system set to German, detected EBCDIC becomes 1141, on one set to American it becomes 1140. And that is the point where it gets dangerous: **the same job delivers two results on two machines** when they are set differently. You do not find that in a test, you find it in production. A detail you can demonstrate: the detection reports a **confidence**. The log says something like "DETECTED CHARSET FOR CHARACTER CONVERSION: UTF-8 TXT (100.00%)". A file with umlauts makes UTF-8 unambiguous; a file of pure 7 bit text can be anything, and then a lower value appears there. Anyone who sees the value understands at once why stating it is better.



The core of the data part. A character does not exist in the target -- and now there are five answers. None of them is the right one; it is a **business** decision. `STOP` aborts at the first character that cannot be converted. That is the default, and it is expensive because every character has to be checked -- but it **finds wrongly encoded data**. That is our differentiator against tools that substitute silently. `IGNORE` drops the character. The length of the field changes with it, and with fixed record lengths that is a problem. `SUBSTITUTE` replaces it with a substitution character. Which one is said by `SUBCHR` -- `SYSTEM` takes the platform convention, `0x1A` for ASCII and Unicode, `0x3F` for EBCDIC. A list can be given as well.
`IDENTITY` copies the character unchanged. That works only for single byte conversions, and it is the way to go when you have binary data in a text field and do not want to touch it.
`TRANSLIT` switches transliteration on -- `ü` becomes something that can be represented in the target. That is the next slide, because the real decision lies there. The sentence to pass on: **`STOP` is the only setting under which you learn that your data has a problem.** All the others deliver something that looks like data. Anyone switching for CPU reasons should switch `REPORT` on -- see the slide on reporting.



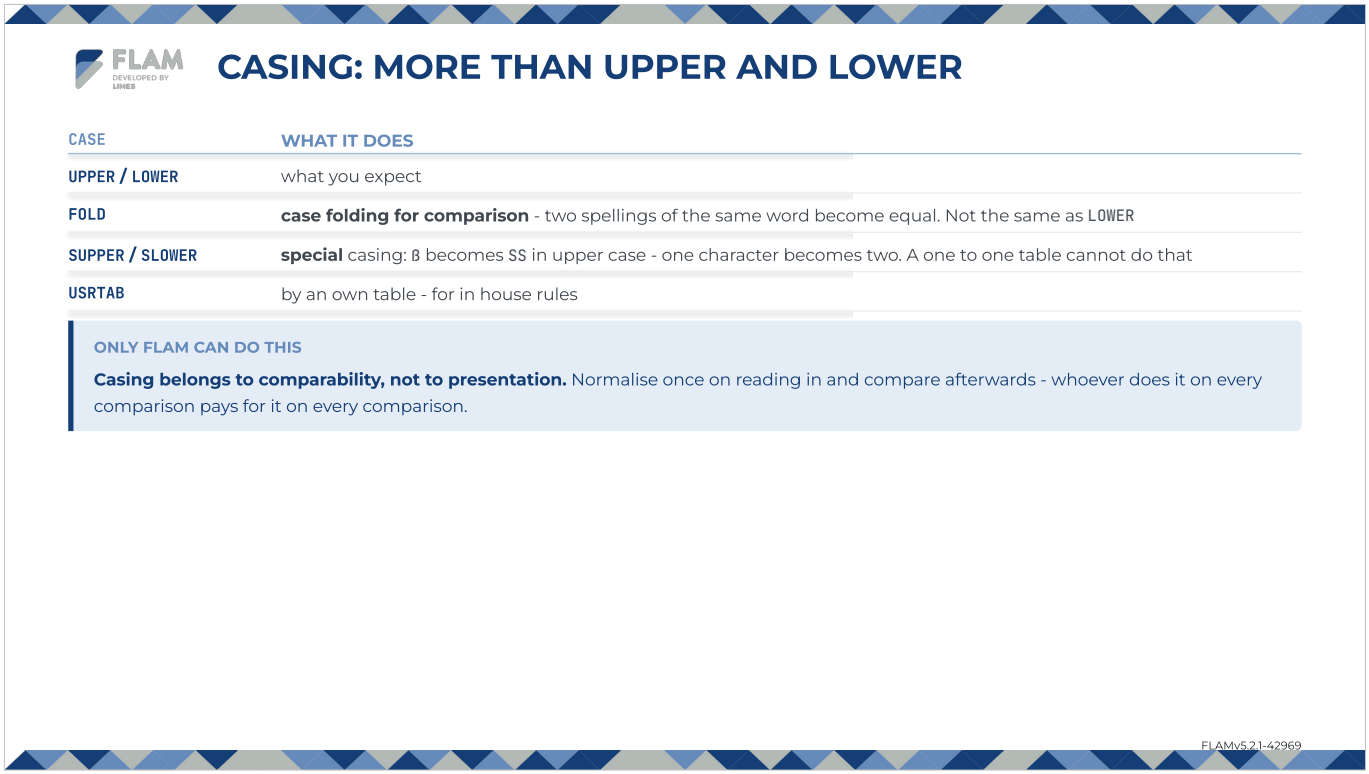
The most important slide of the unit, and the numbers on it are **measured**, not examples. The same input text, the same target code page (`IBM-367`, that is 7 bit US-ASCII), once with the system table and once with the SEPA subset. The system table is the classic ICONV transliteration. It wants to be **readable without loss** and uses marker characters for that: `ü` becomes `"u`, `é` becomes `'e`.
And it knows rules: `ß` becomes `ss`, the euro sign becomes `EUR`.
For a human being the text stays readable. The SEPA subset does something else. Its repertoire is **US-ASCII** -- the single byte characters of UTF-8, so everything below code point 128. And the mapping into it is **one character to one character**: `ü` becomes `u`, `Ä` becomes `A`, `é` becomes `e`. No marker character, no change of length. And now the two results at which you should pause briefly in the room, because they count for German names and addresses: **`Straße` becomes `Strase`** -- with one `s`. The system table produces `Strasse`. `Strase` is not a word, and in an address field that is a data quality problem. **`Müller` becomes `Muller`** -- not `Mueller`. That is a different name. **The euro sign becomes `E`** -- not `EUR`. This is not a criticism of the subset, and it should be said that way: the subset defines a **character repertoire**, not a spelling convention. It maps to US-ASCII, and a one to one mapping keeps the field lengths stable -- which is exactly right in a format with fixed lengths. The German convention `ue`/`ss` is a business rule that sits beside it. But you have to **know** it before applying it to a body of names.
The practical recommendation: anyone moving names and addresses to SEPA should decide beforehand whether he wants the German convention (`ue`, `ss`). Then he needs his **own table** through `USRTAB` with a file name instead of one of the built in keywords -- that is exactly what the option to state your own is for.



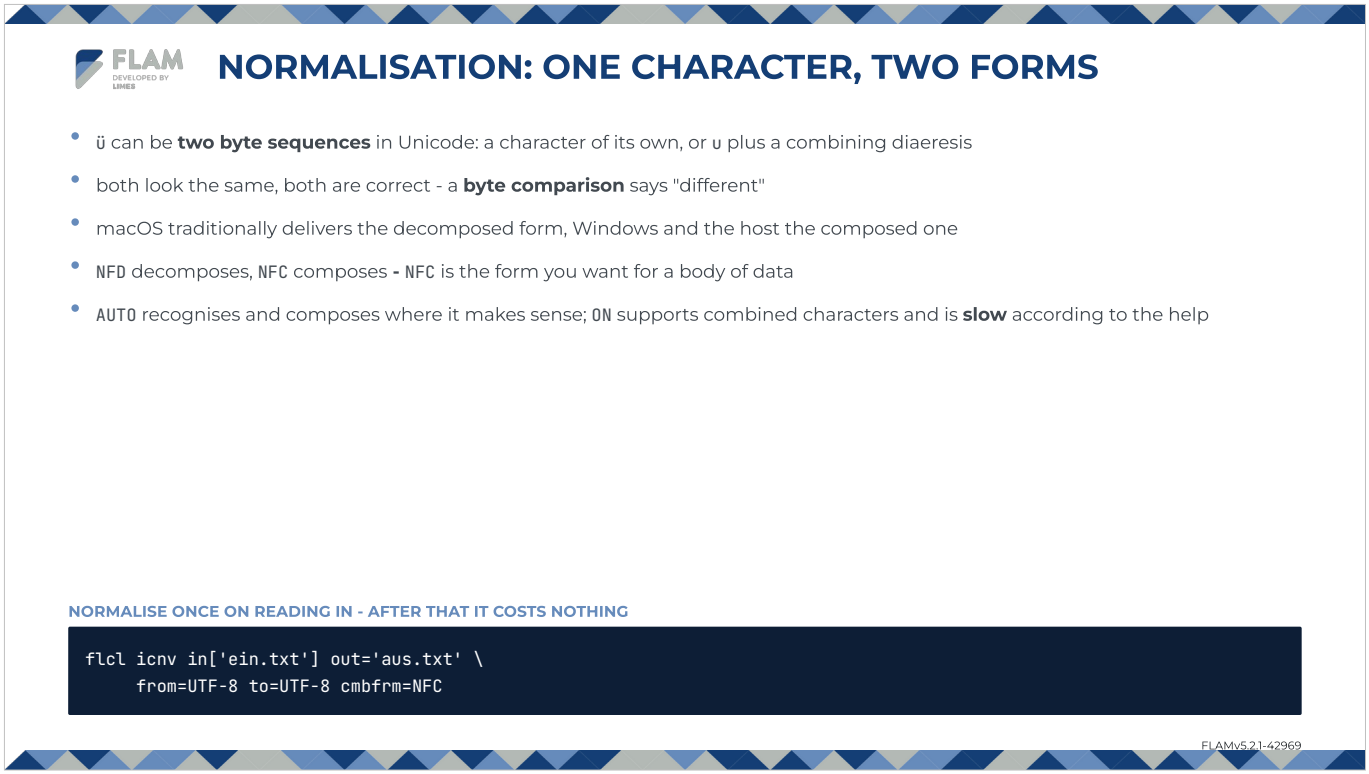
Four subsets are built in, and they are not chosen at random -- each corresponds to a specification a German house knows. `SEPA` maps to **US-ASCII**: everything below code point 128, so the single byte characters of UTF-8. That is the one payment traffic is about. And here an important note, so that nobody promises more than the tool delivers: the subset checks the **character repertoire**, not the field rules of a payment format. Measured:
with `USRTAB=SEPA` and `MODE=STOP` the entire printable ASCII range passes through unchanged -- including the hash sign, the at sign, curly brackets and the tilde, which are not allowed in many payment traffic fields. What the subset does deliver is the assurance: **there is no multi byte character left in here.** Anyone wanting to check against a narrower field repertoire states an own table at `USRTAB`. `NPAS` is the repertoire for statutory reporting: new passport, XOEF, String.Latin. Anyone delivering reporting data needs this one. `DELA` is German Latin as the intersection of IBM-1141, ISO8859-15 and CP1252. That is the practical repertoire for holdings that travel between host, Unix and Windows -- everything representable in all three. `DLAX` is `DELA` extended by XOEF. The benefit is not only the conversion. A subset can also be used as a **check**: with `MODE=STOP` and a subset the run aborts as soon as a character appears that the specification does not allow. That checks a body of data **before** delivery instead of waiting for the recipient to reject it. That is the sentence a compliance person wants to hear: the specification is not in a check routine somebody wrote and has to maintain, it is a parameter.



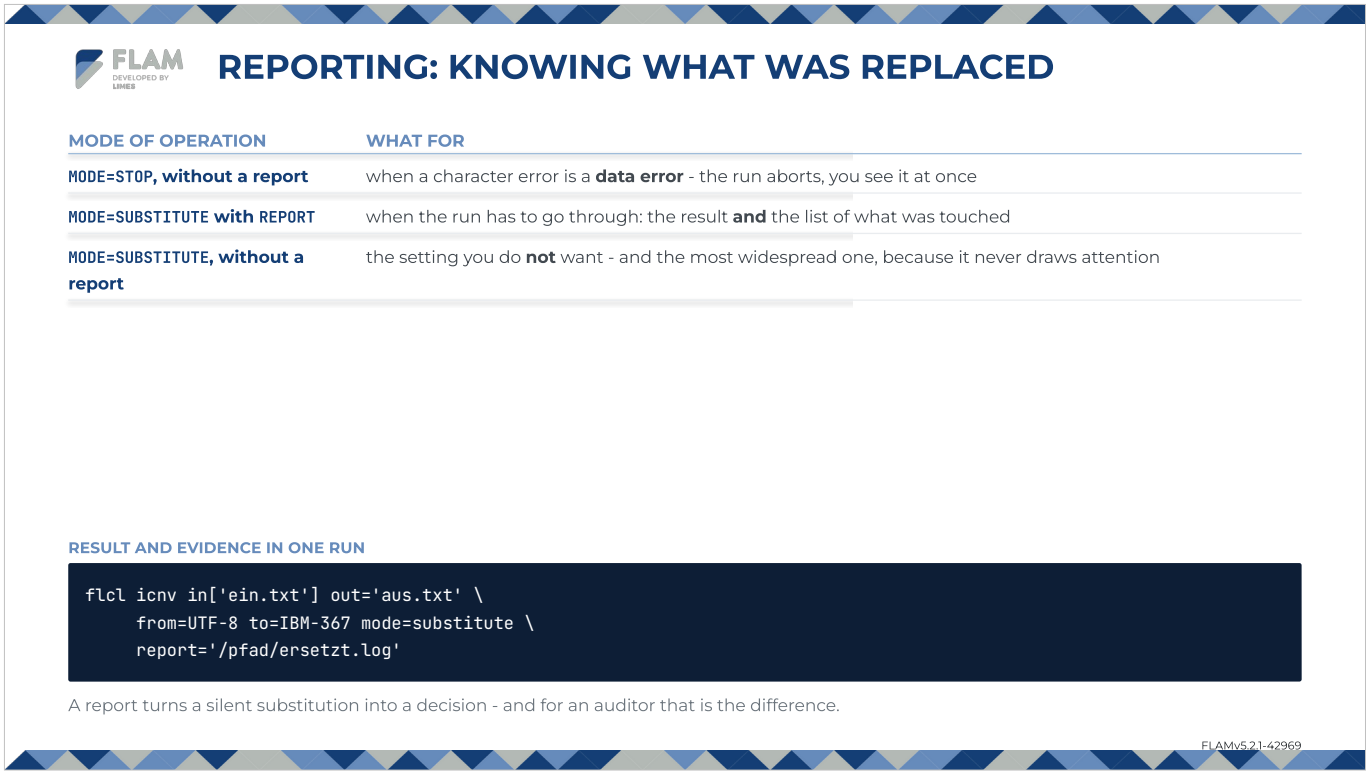
The slide that turns a parameter into a tool -- and that shows something which does not otherwise exist. `USRTAB` does not only take the four built in keywords. It takes **a table of your own**, and in it the customer defines four different things. A sign distinguishes the four kinds, and the difference matters:
Without a sign or with `+` it is a transliteration -- it takes effect **only** when the character does not exist in the target. That is the best fit mapping: for the normal case nothing changes. **With `` *it is a mapping*** -- it is **always** carried out, whether the character exists in the target or not. **With `^` it is a casing** -- your own upper, lower or folding rules, and they take effect only when `CASE=USRTAB` is set. **With `-` a code point is switched off** -- and that is exactly how a **subset of your own** is defined. As a range as well: `(-00-7F)` switches off the whole of US-ASCII, `(+39-30)` switches the digits back on. Now the point to speak slowly at, because it is the real reason for this slide: with a **mapping** a list may stand on the **left** as well. Transliteration is one character onto up to eight; a mapping is **several onto several**. And that turns a re-encoding table into an **input aid**. The case from practice: a house works on the 3270 in EBCDIC but has to support NPAS -- that is, deliver characters which are not on the keyboard at all. Instead of another terminal it defines itself abbreviations: `&OM` becomes the Greek capital omega. The abbreviations can be kept to two characters throughout or terminated with a semicolon, whatever fits the site's own scheme.
The clerk types `&OM`, and the archive holds UTF-8 with the right character. **Nobody else can do that** -- and it costs one line in a table. Two more things that make the table practical: it **extends or overrides** a system table, so not everything has to be defined afresh -- `SYSTAB=ICONV` plus your own lines for your own cases. And the transliteration works **recursively**: if a replacement character is itself not representable in the target, its replacement is applied. The table may be a file -- with all the rules of file names, so `ssh://` from another system as well -- or a string in the parameter that begins with a colon. For short tables and for tables an application produces itself, the string is the more convenient way. For going deeper there is a detailed manual page; it is at `doc/man_cnvchr_usrtab.txt` and in the manual at the parameter.



Casing looks like a minor matter and is one, as long as you only process English. Six settings, and three of them are needed only outside English: `UPPER` and `LOWER` are what you expect. `FOLD` is **case folding for comparison**. That is not the same as lower casing: folding maps onto a form in which two spellings of the same word are equal. Anyone wanting to compare names takes `FOLD` and not `LOWER` -- that is the difference between "looks the same" and "is the same". `SUPPER` and `SLOWER` are the **special** casing. That is needed because the simple rule is wrong in some languages: the German `ß` becomes `SS` in upper case -- one character becomes two. And in Turkish the `i` has an upper case behaviour of its own. A one to one table cannot do that, which is why the special variant exists.
`USRTAB` finally is casing by an own table -- for in house rules. The practical hint: **casing belongs to comparability, not to presentation.** Anyone wanting to make a body of data searchable normalises upper and lower case once on reading in and compares afterwards. Whoever does it on every comparison pays for it on every comparison. And the arc to the indexing unit: anyone wanting to search later without regard to upper and lower case has to decide that **before** the archive is created. Indexing knows no change after the fact.



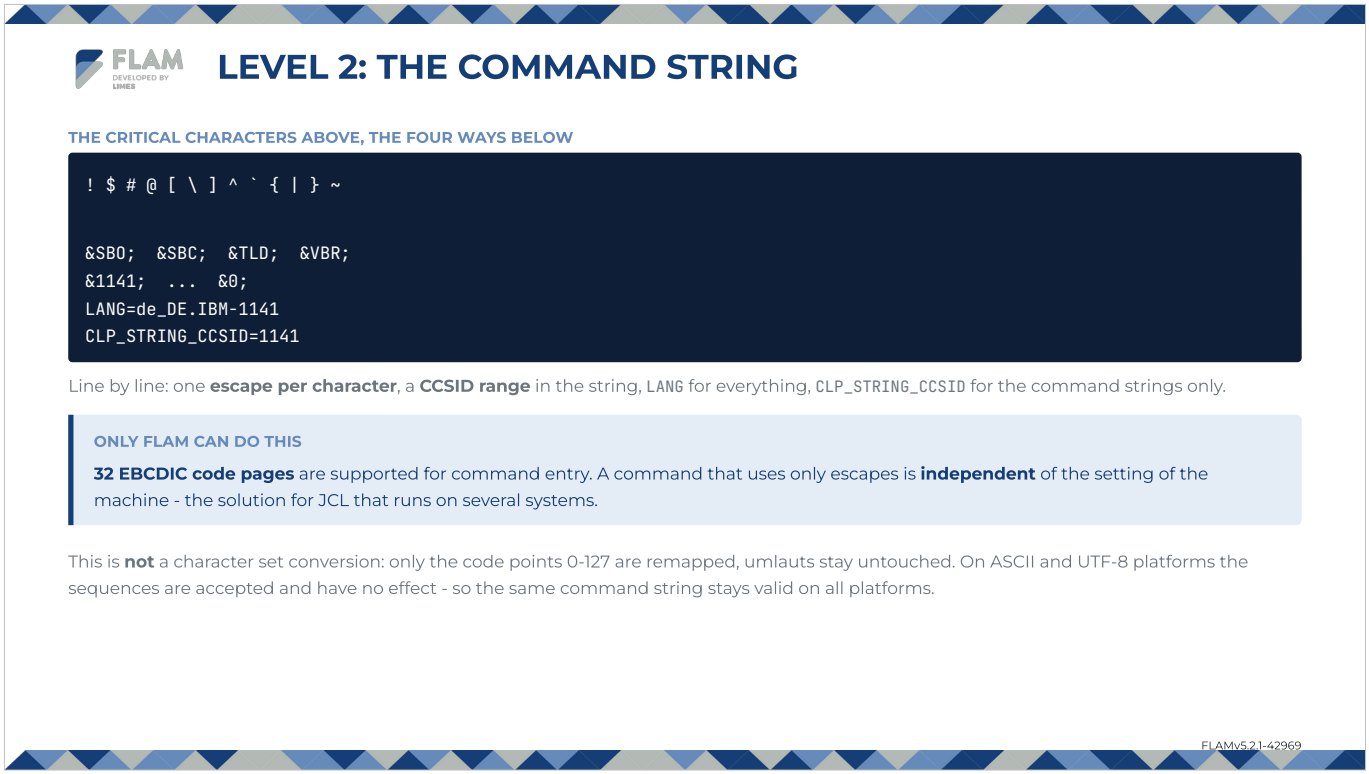
The topic that surprises most people, and one that is unavoidable in a Unicode world. The fact of the matter:
`ü` can be **two different byte sequences** in Unicode.
Once as a character of its own, once as `u` plus a combining diaeresis. Both look the same. Both are correct. And a byte comparison says "different". This is not an academic problem: macOS traditionally delivers the decomposed form, Windows and the host the composed one.
Anyone putting files from both worlds into one body of data has both in there -- and then does not find his records. The answer is normalisation, and `CMBFRM` has it: `NFD` is the canonical **decomposition** -- everything is decomposed into base characters plus combining characters. `NFC` is decomposition **and** subsequent canonical composition -- everything becomes the compact form. That is the form you want for a body of data. `AUTO` recognises combined characters and composes them where it makes sense. `ON` switches support for combined characters on, and the help says explicitly "slow" -- that is an honest statement and it should be passed on. `OFF` switches it off. The recommendation for a body of data that is to be searchable: **normalise to `NFC` on reading in.** Then every character is represented once and the comparison works. After that it costs nothing more.



The slide that turns a setting into traceable processing -- and the one you need as soon as you leave `STOP`. The situation: `SUBSTITUTE` or `TRANSLIT` deliver a file that looks like a good file. The recipient does not see that three characters were replaced. Nor do you. `REPORT` solves that: it logs the **ignored and replaced characters**. The target is `STDOUT`, `STDERR` or a file -- so also a file you archive. That gives three modes of operation, and they should be chosen deliberately:
`MODE=STOP` without a report -- for everything where a character error is a data error. The run aborts, you see it at once. **`MODE=SUBSTITUTE` with a report** -- for everything where the run has to go through. You get the result **and** the list of what was touched. That is the compromise most houses want. **`MODE=SUBSTITUTE` without a report** -- that is the setting you do not want, and it is exactly the most widespread one, because it never draws attention. The sentence for the room: **a report turns a silent substitution into a decision.** And for an auditor the difference is precisely the point.



Two things that occur only at the transition between the worlds. **The line endings.** EBCDIC historically has two characters that occur as a line ending: `0x15` (new line) and `0x25` (line feed). Which one a program produces depends on the compiler and on the age of the code. `ENL2LF` replaces `0x15` with `0x25` on reading, `ELF2NL` replaces `0x25` with `0x15` on writing. Anyone receiving a file in which the line endings are not recognised usually needs exactly one of these two switches. **The BOM.** When changing between Unicode forms there is a byte order mark at the start. `BOM` writes one, `KEPBOM` keeps an existing one. Anyone handing a file to a program that does not expect a BOM has to switch it off -- and anyone handing one to Windows usually wants it. That completes level 1. From the next slide on it is no longer about the data but about the command, the parameter file and the program. And the read side that goes with it, because it is needed just as often: as a `CCSID` one can give **`BOMUTF`** or **`BOMUCS`**. Then the byte order mark of the file decides which Unicode form is read -- `BOMUTF` takes UTF-8 as the fallback, `BOMUCS` UTF-16. That is the answer to "I do not know which Unicode form the file arrives in".



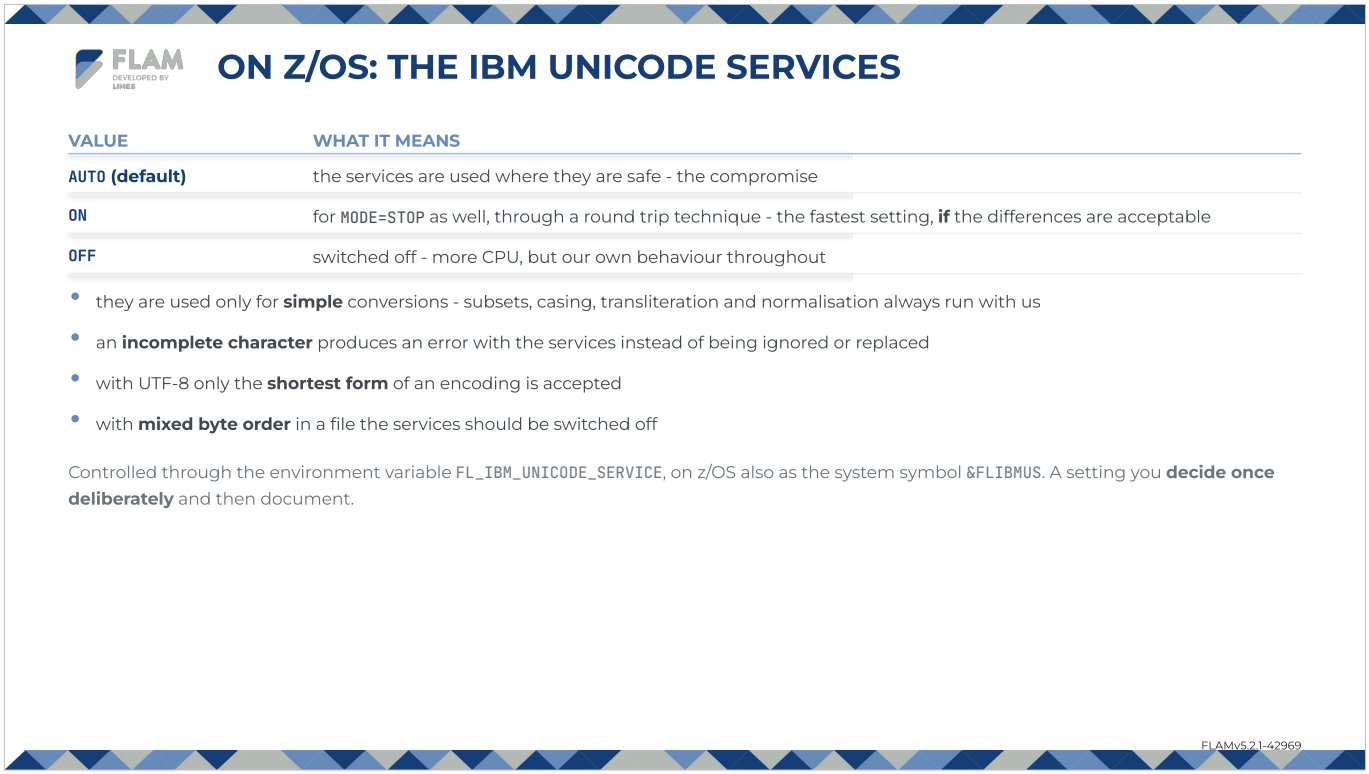
Now the level nobody expects -- and the one that regularly strikes on the host. The fact of the matter: fourteen punctuation characters do **not** sit at fixed positions in the EBCDIC code pages. Exclamation mark, dollar, hash, at sign, the square and curly brackets, backslash, circumflex, backtick, pipe and tilde.
But those are exactly the characters that make up the syntax of our command language. A command with a square bracket typed on a 1141 machine and read on a 1047 machine is a different command.
How FLAM solves that -- and there is a lot of work in the product behind it: **First through `LANG`.** The parser evaluates the environment variable. `LANG=de_DE.IBM-1141` is explicit, `LANG=de_DE` is enough as well -- then the CCSID is derived from the language code: DE is 1141, US is 1140, C is 1047. Without `LANG`, 1047 applies. **32 EBCDIC code pages** are supported for command entry. **Second through escapes.** Every critical character can be written as a sequence: `&SBO;` for the opening square bracket, `&TLD;` for the tilde, `&VBR;` for the vertical bar. A command that uses only escapes is **independent** of the setting of the machine. That is the solution for JCL meant to run on several systems. **Third through CCSID sequences in the string.** `&1141;` switches the interpretation to 1141 in the middle of the string, `&0;` switches back to the system code page. That is level 4 and comes shortly. **Fourth through `CLP_STRING_CCSID`** -- the CCSID for the CLP strings only, independent of `LANG`. Two more things that have to be said: this is **not a character set conversion**, only the code points 0 to 127 are remapped. Umlauts are not touched. And on ASCII and UTF-8 platforms the problem does not exist -- the escapes are accepted there all the same and have no effect. That is deliberate: **you can write platform independent command strings.**



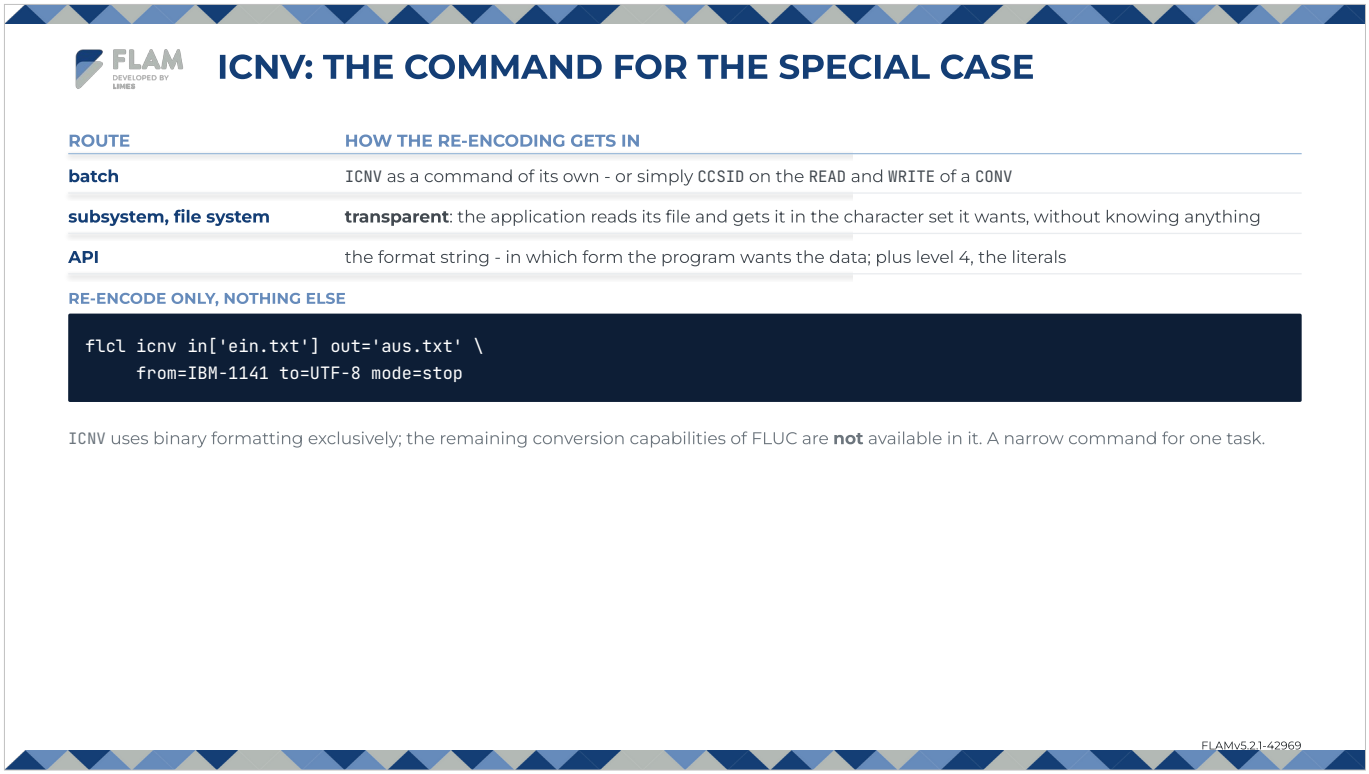
The same question one level further on, and here it becomes practical. A command does not have to be in the JCL or on the command line. For objects, overlays and arrays a **parameter file** can be given, with `=` or `=>`, for ordinary arguments with `=>`.
Its content is a command string in the same syntax. Now the decisive point: a file does not carry its character set inside itself. As long as it sits next to the program on the same machine that does not matter -- it is in the local character set. But the file can come over a URL from **another platform**: `ssh://` fetches it from a remote system, and a Linux server delivers UTF-8 to a z/OS job that expects EBCDIC. That is what the CCSID on the file name is for.
The URL syntax knows three additions: `/?` for a member name, `/:` or `/#` for a member index -- and `/&` for the **CCSID the content of the file is encoded in**. And then the sentence from our own manual, which is worth reading out: "The filename is the only way to specify the CCSID for this type of files." For parameter files there is **no other way**. There is no parameter for it, because the parameter would be inside the file you have to read first. That is not a shortcoming, it is the resolution of a chicken and egg problem -- but you have to know it, otherwise you look in the wrong place.
In practice: anyone keeping parameter files centrally on a server and reading them from host and Unix attaches the CCSID to the URL once and is done with the topic.



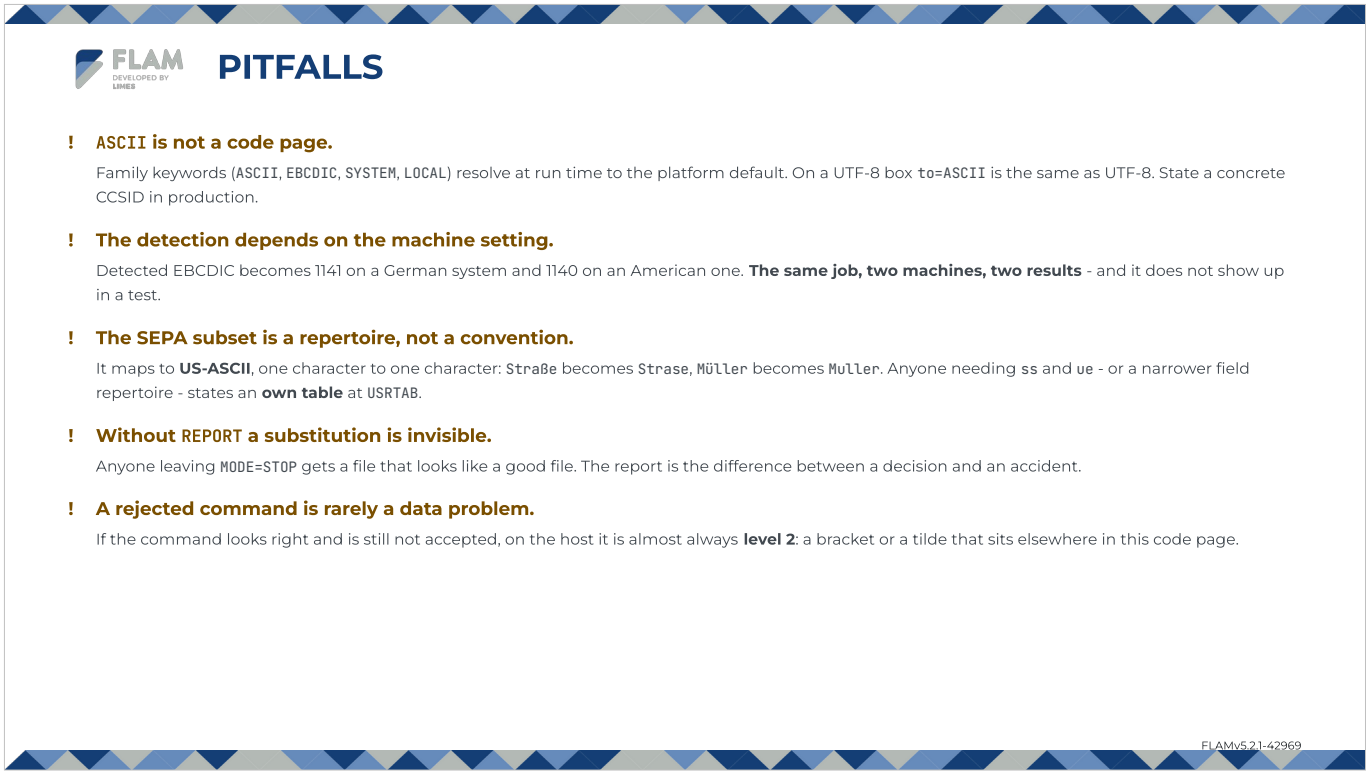
The fourth level, and it concerns everybody who programs against our APIs -- so the participants from application development.
The situation on the host: a program assembles its command string. Part of it is a **literal** in the source, another part comes from a variable at run time -- a file name for instance.
And now the point: the character set of the literals is fixed **at compile time**, that of the variables applies **at run time**. Those can be two different ones. Concretely: C source is normally in 1047, and if nothing else is set the literals are in 1047 as well. In COBOL the default for literals is 1140. The file name the program receives, however, is in the local character set -- so 1141 for instance. A command string assembled from both is therefore **in two character sets at the same time**. No parser can guess that. The solution is the CCSID sequence in the middle of the string, and the example is in our manual exactly like this: the literal part is marked with `&1047;`, before the variable `&0;` switches back to the system code page, afterwards back to 1047. Every part of the string is then in the character set it actually is in. And for the case where you cannot recompile -- old code, somebody else's program:
then you set `CLP_STRING_CCSID` from outside. That is exactly what it is there for. A detail that brings peace of mind: on ASCII and UTF-8 platforms these sequences have no effect. So they can be left in shared source and the code does not have to branch by platform.



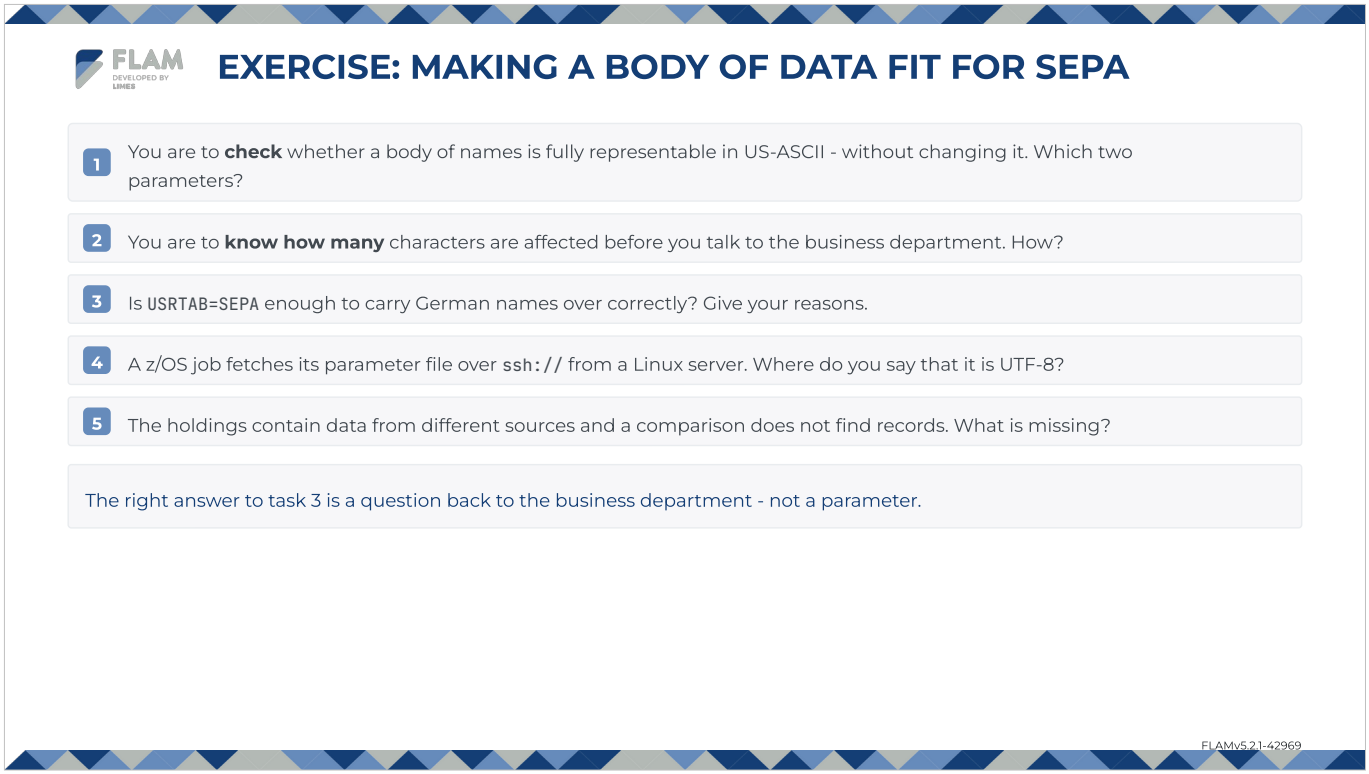
A slide for operation on the host, and one where we deliver an honest trade-off instead of a recommendation. Since version 5.1.27 FLAM can use the **IBM Unicode Services** on System Z instead of its own implementation. That saves CPU and memory. But only for **simple** conversions: as soon as an own table, transliteration, casing, a subset, whitespace handling, reporting or normalisation is involved, our own module does the work anyway. So everything that was interesting in this unit still runs with us. It is controlled through `FL_IBM_UNICODE_SERVICE`, on z/OS also as the system symbol `&FLIBMUS`. Three values: `AUTO` is the default and the compromise: the services are used where they are safe.
`ON` uses them for `MODE=STOP` as well, through a round trip technique. That is the fastest setting, and we recommend it explicitly **if** the differences in behaviour are acceptable. `OFF` switches them off and costs more CPU. The differences, and they have to be named because they show up in practice: an incomplete character produces an error with the services where our module would ignore or replace it.
With UTF-8 the services accept only the **shortest** form of an encoding. Our module decodes first and checks the code point afterwards -- so it accepts overlong forms as well.
With the BOM the behaviour depends on the APAR version. And with **mixed byte order** in a file the services should be switched off -- a change from LE to BE in the middle of the data is not detected. The sentence for the room: this is a setting you **decide once deliberately** and then document.
Not one you try out.



To place it, because this unit has said a lot about `ICNV`:
character set conversion is **normally part of a conversion**, not a run of its own. Anyone converting a file and re-encoding it on the way simply states `CCSID` on the `READ` and on the `WRITE` -- done. `ICNV` is for the case where you want to re-encode **only**.
It runs as part of the read operation of FLUC, works block oriented -- record oriented with `RECORD` -- and uses binary formatting exclusively. All the other conversion capabilities of FLUC are **not** available in this command. That is deliberate: a narrow command for one task. And the placing into the three ways, because it is particularly clear here: in **batch** `ICNV` is the command, or `CCSID` on a `CONV` statement. In the **subsystem and the file system** the re-encoding happens transparently -- the application reads its file and gets it in the character set it wants, without knowing anything. That is the way for standard software. Through the **API** it is the format string: you say in which form the program wants the data -- and observe level 4 while doing so.
The closing sentence: **the choice of character set is the same decision, whichever way you go.** Only the wrapping differs -- as with everything in FLAM.



Five, and the first one caught me out myself. On the first: I set `to=ASCII` and was surprised that the umlauts stayed. The log said "Character conversion to = UTF-8" -- and at first I took it for a defect. `ASCII` is a family keyword, and on a UTF-8 Linux box it resolves to UTF-8. The tool did the right thing. On the second: the detection depends on the country setting of the machine. Two machines, two results, and it does not show up in a test, because the same setting applies there. On the third: the SEPA subset is a one to one mapping. For names and addresses the result is not the German convention. On the fourth: anyone leaving `STOP` should switch `REPORT` on. Otherwise the information that anything was replaced at all disappears. On the fifth, and this is the level trap: when a command is not accepted on the host although it looks right, it is almost never the data. Then it is level 2 -- a bracket or a tilde that sits elsewhere on this machine. Whoever checks the data first is looking at the wrong end.



Twenty minutes, and the tasks are chosen so that three of them require keeping the levels apart. The solutions:
Task 1. `MODE=STOP` together with `USRTAB=SEPA`. The run aborts at the first character that is not in US-ASCII -- without a line of checking code. And please stay precise here if the question comes from the room: this is the check on the **character repertoire**, not on the field rules of a payment format. Hash, at sign and tilde are US-ASCII and pass through. **Task 2.** With `MODE=SUBSTITUTE` or `TRANSLIT` and `REPORT` to a file. Then it runs through and you get the list of affected characters. That is the number you take into the conversation with the business department. **Task 3.** No -- and that is the core of it.
The SEPA subset produces `Muller` and `Strase`. Anyone needing `Mueller` and `Strasse` needs an own table through `USRTAB` with a file name. Which variant is correct in business terms cannot be decided by IT: `Muller` is a different name, and whether the recipient accepts it is in his specification. **Task 4.** Attach the CCSID to the file name: `/&1141` on the `ssh://` URL. For parameter files the file name is the **only** way -- there is no parameter for it. **Task 5.** Normalise to `NFC` on reading in. Otherwise two representations of the same character sit in the holdings and the comparison fails -- especially with data that has passed through macOS.

WHAT TO TAKE AWAY

- **four levels**: the data, the command string, the parameter file, the literals in the program - the first question is **which one**
- **186 code pages**, two thirds of them single byte; `ASCII` and `EBCDIC` are **family keywords**, not code pages
- without a statement: **local on writing, detected on reading** - and the detection depends on the machine setting
- five error behaviours; `STOP` is the only one under which you learn that the data has a problem
- **transliteration** uses marker characters and knows rules (`ß` to `ss`); a **subset** is a character repertoire - `SEPA` is US-ASCII - and maps one character to one character
- `FOLD` for the comparison, `NFC` for the holdings, `REPORT` for the evidence, `/&ccsid` for the parameter file

FLAMv5.21-42969

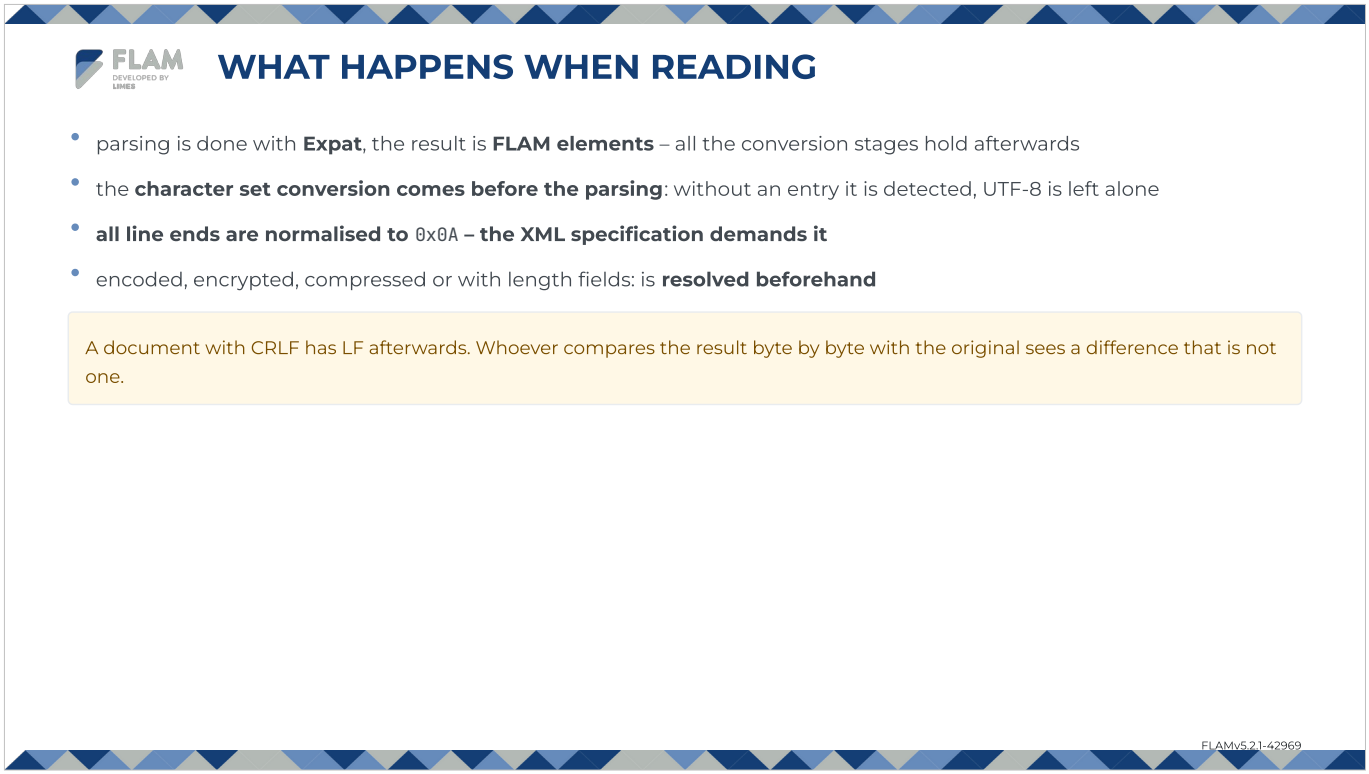
Go through the learning objectives. Two sentences should stick:
the character set is stated in four places, not one -- and the re-encoding of the data is a business decision for which the tool has every switch, but which it does not make for you.



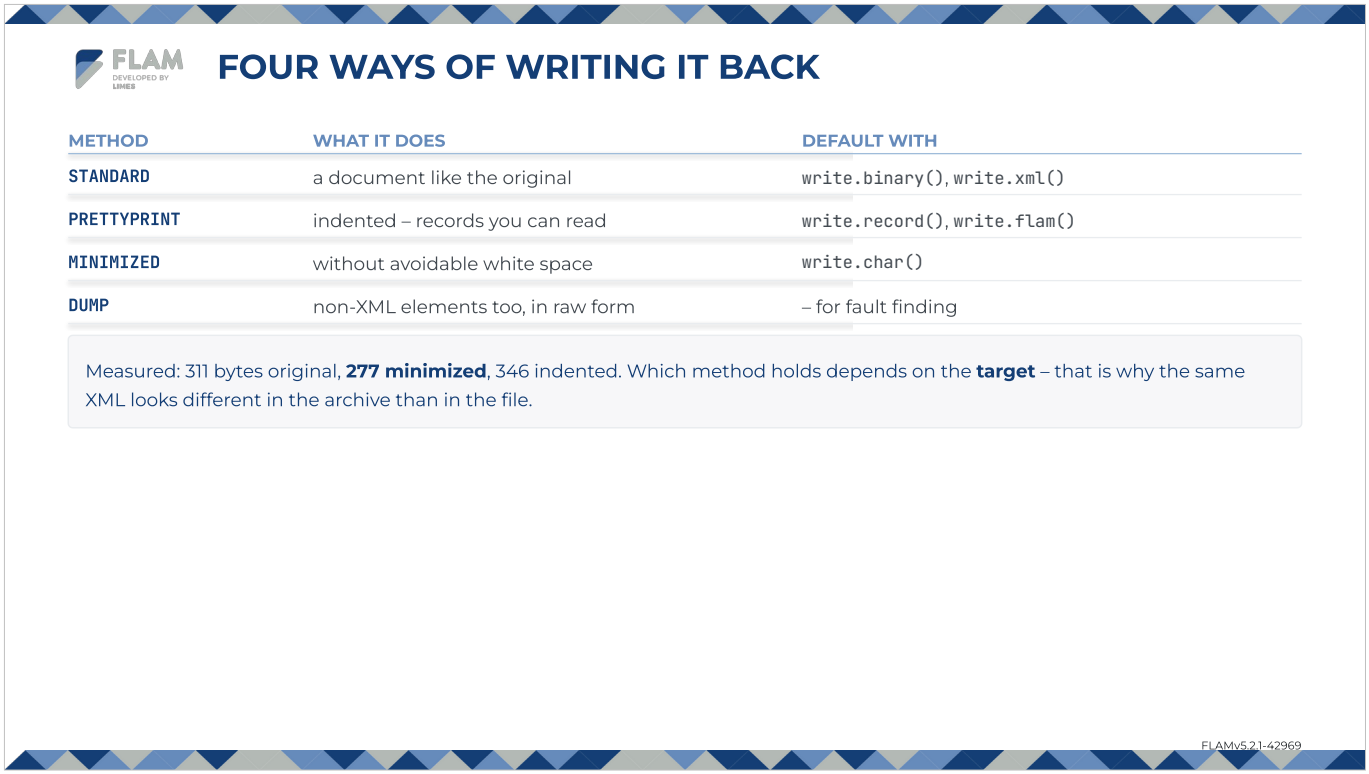
The unit that joins two worlds: the hierarchical one of XML and the flat one of records and columns. The arc: what FLAM makes out of XML when reading, the four ways of writing it back, the character set questions, and then the actual reason for this unit -- XML as a table, that is, the route to CSV and into an archive. The sentence for the start, and it is a warning and a promise at once: **XML is more powerful than a table.** So not every XML goes into a table -- but what occurs in payment traffic does, and then in one call.



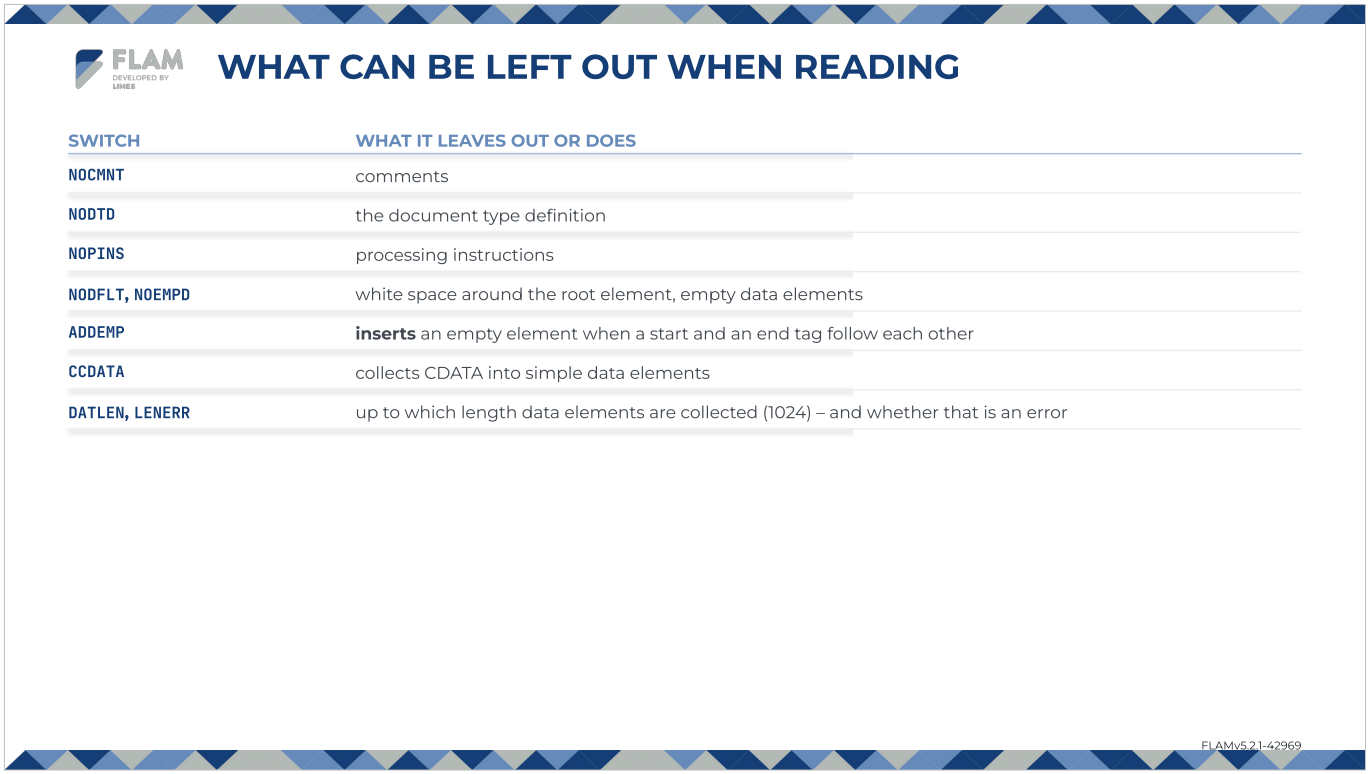
The fourth objective is the reason the unit exists. The fifth is the one that saves a project from a wrong promise: the table form has a clear limit, and it is better known beforehand.



The slide that explains the mechanism -- and it clears up a misconception. FLAM parses the document with **Expat** and makes **FLAM elements** out of it. That is the point: no text copy and no DOM in the application sense comes about, but the usual internal form -- the same one records and columns work with. That is why all the conversion stages can be applied afterwards. What happens before is important for the result: **the character set conversion comes before the parsing.** Without an entry the encoding is detected; if it is UTF-8, nothing is converted. With `CCSID` it is converted from there to UTF-8, and the entry wins against the declaration in the document. And something else happens that has to be known: **all line ends are normalised to 0x0A**.
That is no liberty of FLAM's, the XML specification demands it. A document that comes in with CRLF has LF afterwards -- whoever compares byte by byte finds a difference that is not one. And the chain before holds here too: if the input is encoded, encrypted, compressed or carries length fields, that is resolved automatically before parsing.



The slide after which nobody asks any more why the result looks different from the input. `STANDARD` writes a document that corresponds to the original -- the default with `write.binary()` and `write.xml()`. `PRETTYPRINT` indents.
That is the default with `write.record()` and `write.flam()`, and there is a reason: in a FLAM archive XML is record- oriented, and an indented document has records you can read.
`MINIMIZED` throws away every avoidable bit of white space -- the default with `write.char()`. For a transfer that is the right choice. `DUMP` writes non-XML elements into an XML document too, in raw form. That is the tool for fault finding. Measured on a small payment file: 311 bytes in the original become 277 bytes minimized and 346 bytes indented.
With real documents the difference is larger, because there is more structure and more indentation there. The sentence that belongs with it: **which method holds depends on the target.** Whoever does not know that wonders why the same XML looks different in the archive than in the file.



A reference slide with a practical core. An XML document holds much that is not needed for the processing, and every piece of it costs memory and time. The switches say what you do not want to see: `NOCMNT` the comments, `NODTD` the document type definition, `NOPINS` the processing instructions, `NODFLT` the white space before and after the root element, `NOEMPD` empty data elements. `ADDEMP` is the other direction: it inserts an empty data element when an end tag follows a start tag directly. That is needed when the further processing expects a value for every tag -- otherwise the column is missing. `CCDATA` collects CDATA sections into simple data elements so they can be treated like normal text.
And `DATLEN` with `LENERR` is the place for large text nodes:
`DATLEN` says up to which length data elements are collected -- the default is 1024 -- and `LENERR` turns that into an error instead of a silent split. The advice: when reading for the table transformation, throw away **everything** that is not data. The table transformation ignores formatting, comments and the DTD anyway.

XML AS A TABLE: ROOT AND PATH

ONE CALL: XML TO CSV

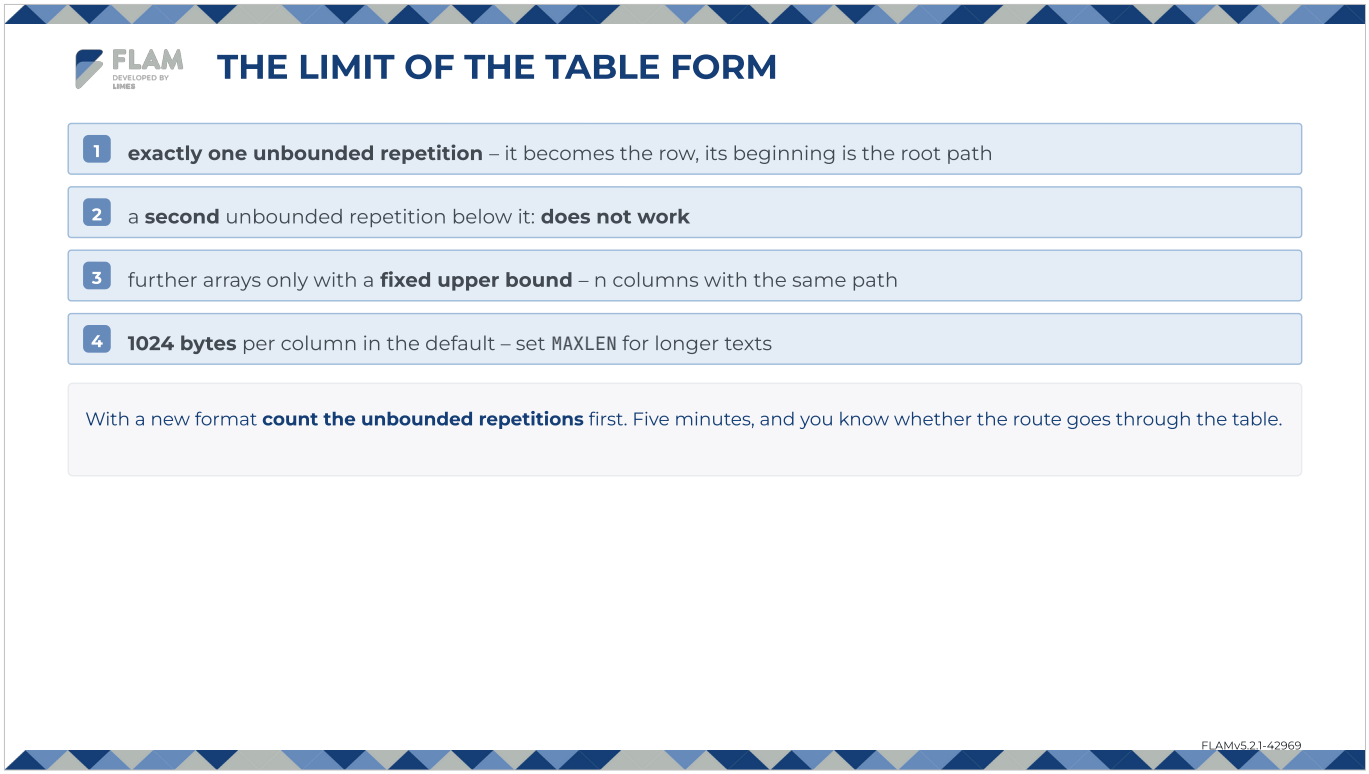
```
# row spec in a file of its own -- ROW=> reads it
NAME='zahlung'
COLUMN(NAME='id'   FORMAT.XML(ROOT='<zahlungen/zahlung>'
                              PATH='<@id>')  TYPE.STRING())
COLUMN(NAME='iban' FORMAT.XML(ROOT='<zahlungen/zahlung>'
                              PATH='<iban>') TYPE.STRING())


flcl XCNV "input(sav.fil(fio.blk(file='zahlung.xml')
                         fmt.tab(format=XML row=>'zrow.txt')))
           output(sav.fil(fmt.tab(format=CSV)
                         fio.blk(file='zahlung.csv')))"

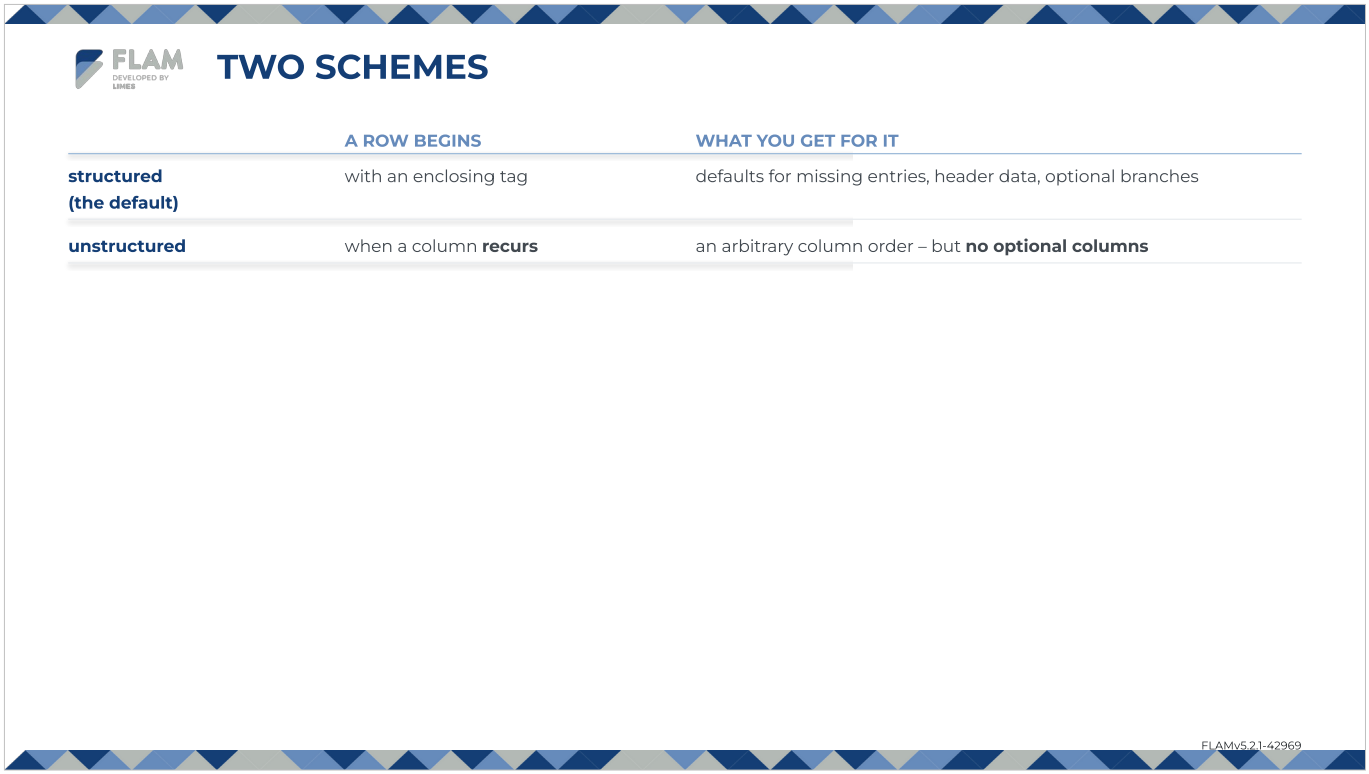

# measured result:
# "id","iban","betrag","währung"
# "1","DE02120300000000202051","1234.56","EUR"
```

FLAMv5.21-42969

The core slide of the unit, and the example is a real run. Two entries suffice: `**ROOT**` is the path to a row, `**PATH**` the path within the row. Both stand in angle brackets, both are case sensitive, and the depth may go up to 64 levels. An attribute is reached with an `@` in front of the last element -- and because the at sign lies differently per EBCDIC code page, an `&` works too.
That is the same precaution as in the SSH URL. In the example: `ROOT='<zahlungen/zahlung>'` makes every `zahlung` element a row, and the four columns fetch the attribute and text nodes. The row specification stands in a file of its own and is read with `ROW=>` -- exactly the handle from the language unit, and it is particularly useful here, because a row specification gets long. It was run with `XCNV`: `FMT.TAB(FORMAT=XML)` on the input, `FMT.TAB(FORMAT=CSV)` on the output. Out come the three lines below, with a header line. The sentence for the room: **this is one call.** No XSLT, no program, no intermediate file. And because it is a conversion chain, a FLAM archive can stand at the output just as well as a CSV file.



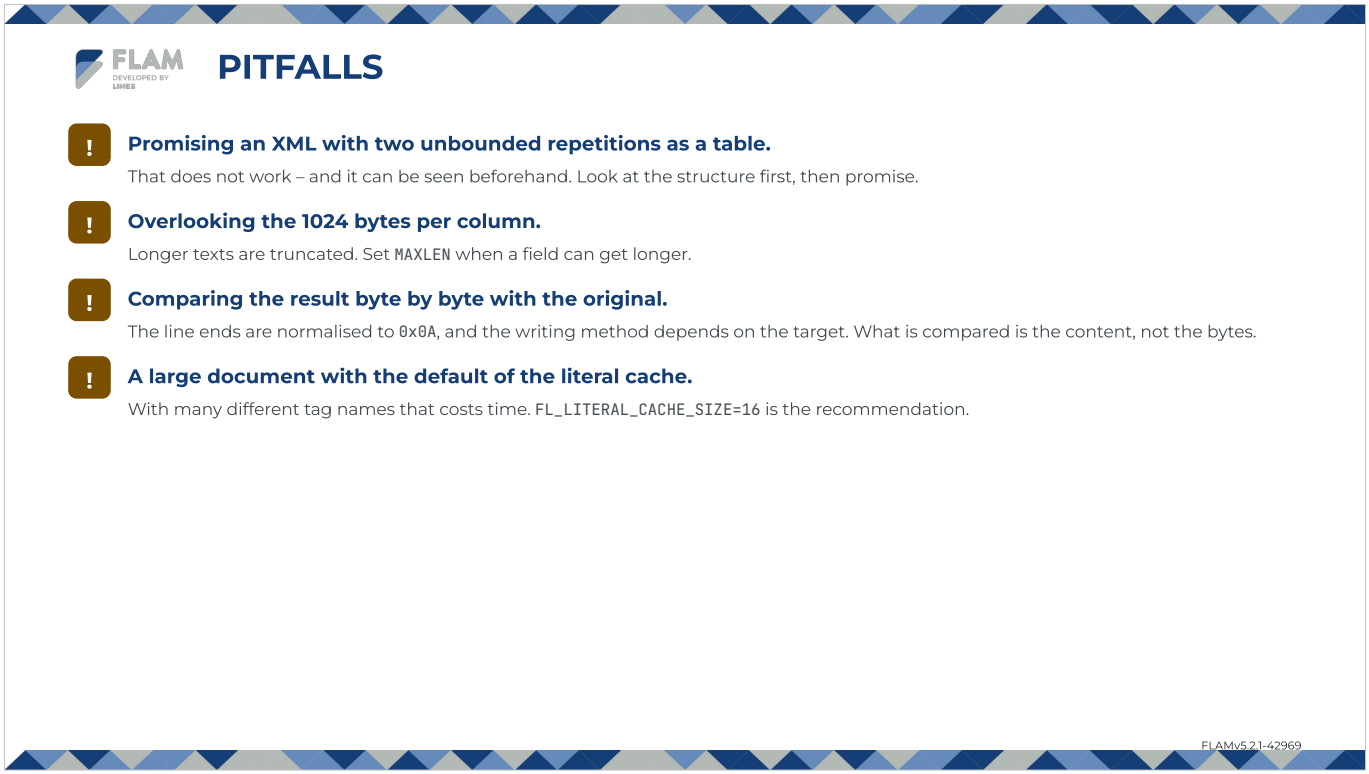
The slide that saves a project from a wrong promise. It stands deliberately right behind the success example. Our manual says it itself: XML is a very powerful data format, and it is translated here into a **less powerful** form. Not every XML can be turned into a table. The rule by which it can be seen beforehand: the table form takes exactly **one unbounded repetition**. Each of these repetitions becomes a row, and its beginning is the root path. If a **second** unbounded repetition occurs below it, it does not work. Further arrays are possible, but only with a **fixed upper bound**: you give n columns with the same path, and with that the number of entries is capped. There is no unbounded number of columns per row. And a number that has to be known: the default for the maximum data length of a column is **1024 bytes**. Whoever has longer texts sets `MAXLEN` -- otherwise the value is truncated. The practical advice: with a new format look at the structure first and count the unbounded repetitions. That is a question of five minutes and decides whether the route goes through the table or through single documents.



The slide for the case where the document is not built the way one expects. The **structured** scheme is the default:
every row is enclosed by a tag. That is the normal case, and only that way are there defaults for missing entries, header data, optional branches and the checking of them. The **unstructured** scheme needs no enclosing brackets: a new row begins when a column occurs that has already been read for the current row. With that the order of the columns within a row is arbitrary -- but there are **no optional columns**: every column has to occur in every row. Since 5.1.25 the structured scheme also takes documents with only one level -- then the whole thing is a table with one row.
That sounds academic but is the case when you want to fetch the header data of a document into a row. The advice: when there is a choice, take the structured one. The unstructured one is for documents that cannot be changed.



Two things that turn up with real documents. **Name spaces** have been supported since 5.1.25. That is the prerequisite for an ISO 20022 document being processable at all -- there is a prefix on every tag there. And with the same change came an optimisation that has to be known, because it has a setting: tag and attribute names are put **once** into a literal cache and reused afterwards. That saves memory and time -- but only if the hash table is large enough. The default is 12, that is 4096 entries. Through the environment variable `FL_LITERAL_CACHE_SIZE` it can be set between 8 (256) and 20 (1,048,576), and our manual **recommends the value 16 for large documents**. That is the kind of entry you write into the start environment once and forget afterwards -- and that makes the difference when a document has a hundred thousand different tag names. Two small things to close: use no diacritics in tag names, and if the UTF-8 holds decomposed combined characters, normalise to NFC beforehand. The character set unit says how.



All four out of projects.



Twenty minutes at the computer, with a document of your own if there is one. The solutions: **Task 1.** Count the unbounded repetitions. Exactly one: it works. Two:
it does not, and another route is needed -- one document per repetition, for instance. **Task 2.** `ROOT` is the path to the row, `PATH` the path in the row; an attribute with an `@` in front. And the row specification into a file of its own, because it gets long. **Task 3.** Set `MAXLEN` to the expected length.
Without the entry it is 1024 bytes, and the rest is missing silently. **Task 4.** Nothing in the call -- `FL_LITERAL_CACHE_SIZE=16` into the start environment.
That is an operations entry, not a processing entry.

WHAT TO TAKE AWAY

- parsing is done with **Expat**, afterwards they are FLAM elements – all the conversion stages hold
- **four writing methods**, and the default depends on the **target**
- line ends are **normalised** to `0x0A` – the specification demands it
- `ROOT` **is the row,** `PATH` **the column,** `@` an attribute – one call to CSV or into the archive
- the table form takes **one** unbounded repetition, and **1024 bytes** per column
- for large documents `FL_LITERAL_CACHE_SIZE=16`

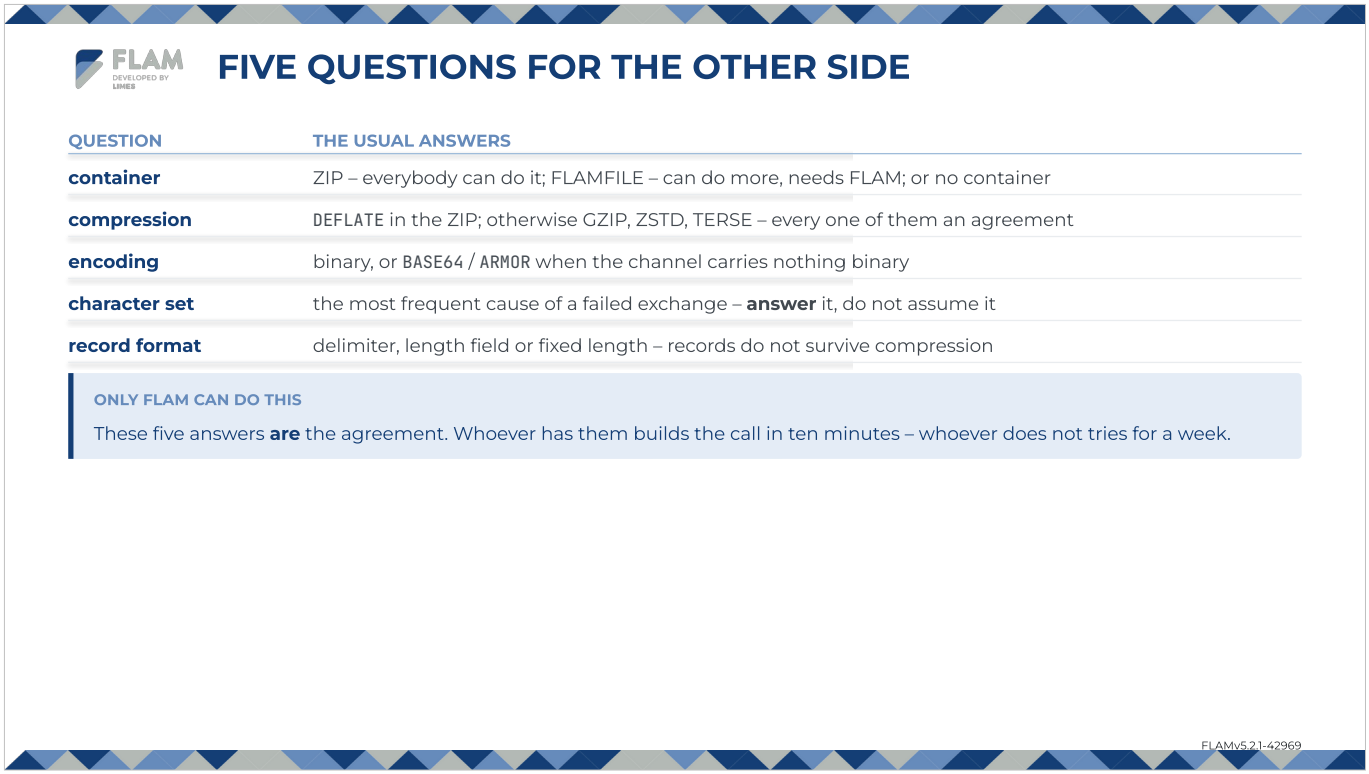
Go through the learning objectives. The pointer forward:
what happens with the columns afterwards -- types, checks, CSV details -- is in *Tables and CSV*.
And the formats of other manufacturers are in *Foreign formats and exchange*.



The last unit of the data group, and the one with the most bearing on the working day: almost every data exchange goes to somebody who has no FLAM. The arc: the question that has to be asked first, then ZIP as the format everybody can read, the FLAM formats of our own, the encodings for channels that carry nothing binary, and the transparent reading. The sentence for the start: **the question is not what we can do but what the other side can read.** Everything else follows from it -- and this unit gives the list to go through with the partner.



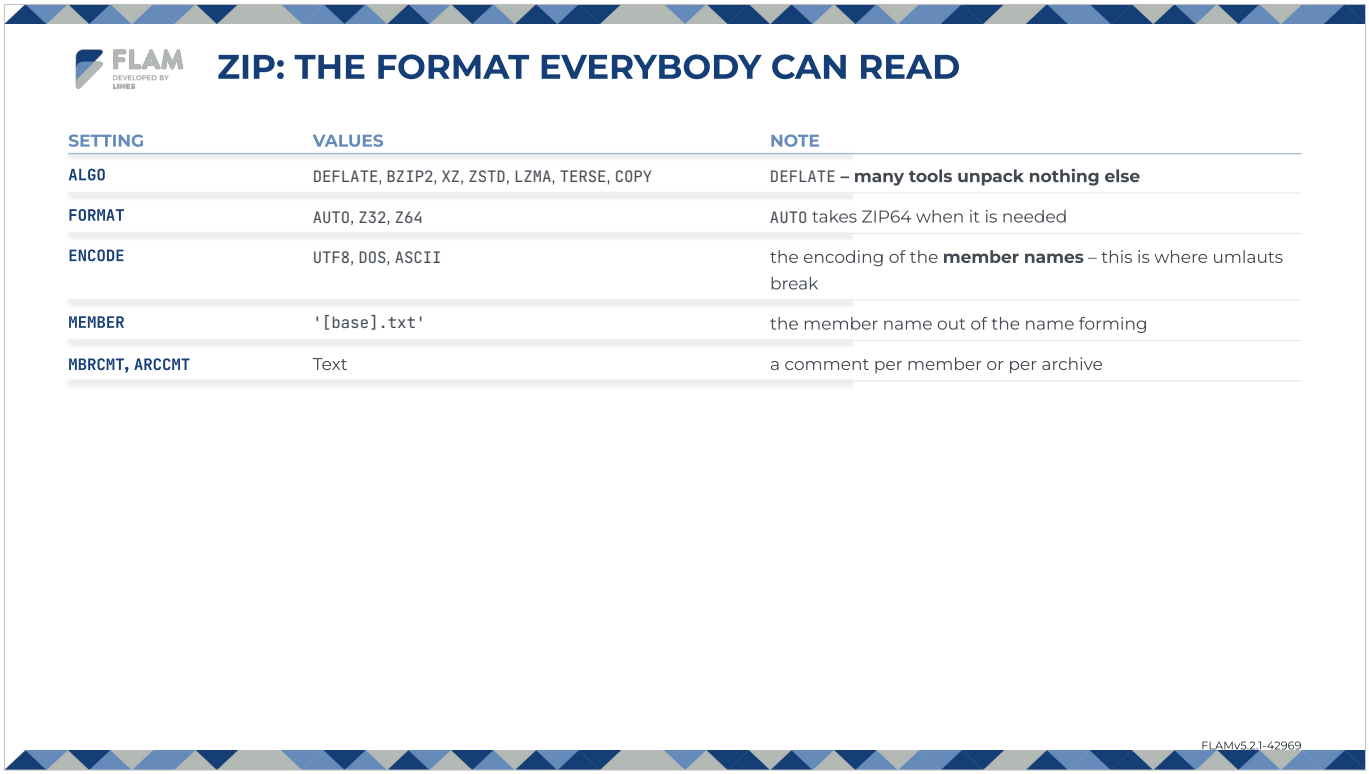
The first objective is the checklist to take into a partner conversation. The others are the answers to it. The fifth is the warning: transparent reading is comfortable and explicitly not dependable -- our manual page says so.



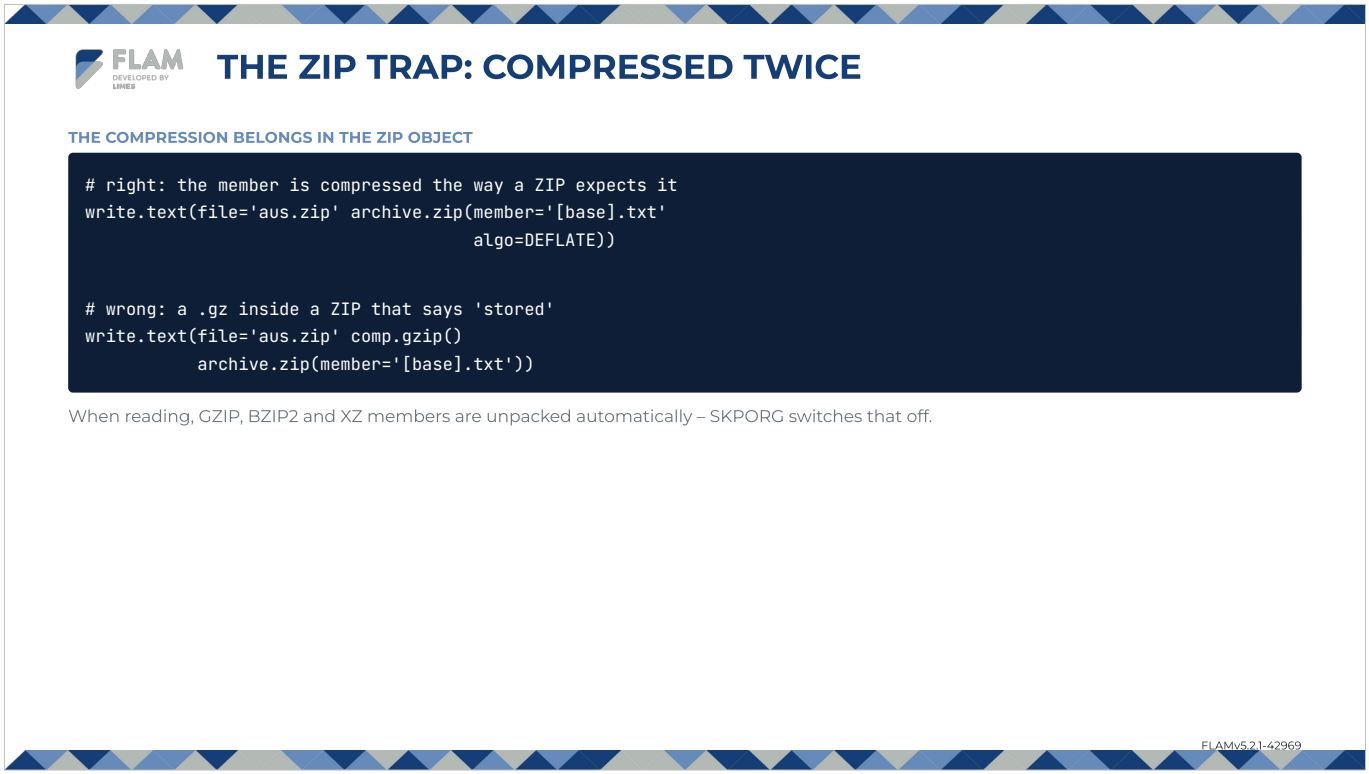
The slide to print out and take into the conversation. Every line has a wrong answer that is expensive. **The container.** Everybody can do ZIP. A FLAMFILE can only be done by somebody who has FLAM -- in exchange it can do more. A single stream needs no container. **The compression.** Inside a ZIP, DEFLATE is the answer every tool understands. Everything else is an agreement. **The encoding.** Does the channel carry binary data? E-mail, form fields and some interfaces do not -- then Base64 or ASCII armor, and you pay a third more space. **The character set.** The question most exchanges fail at, and it has a unit of its own. Only one thing counts here: it has to be **answered**, not assumed. **The record format.** Records survive neither compression nor encryption.

If the other side needs records, it has to be agreed how they are delimited -- delimiter, length field or fixed length. The sentence for the room: **these five answers are the agreement.** Whoever has them builds the call in ten minutes.

Whoever does not have them tries for a week.



The slide about the workhorse of the exchange. A ZIP comes about as part of the chain: `ARCHIVE.ZIP()` at the `WRITE`, and the member name comes out of the name forming -- `member='[base].txt'` writes every input file into it under its base name. Measured: a directory walk over three files gives a ZIP with `f1.txt` to `f3.txt`, readable with the ordinary `unzip`. The entries you really need: `ALGO` -- **`DEFLATE` is the default and the right choice**, because many tools unpack nothing else. `BZIP2`, `XZ`, `ZSTD`, `LZMA` and `TERSE` work too, but then the other side has to play along. `COPY` stores uncompressed.
`FORMAT` decides on ZIP32 or ZIP64. `AUTO` chooses what is needed -- ZIP64 from four gigabytes or 65,535 members.
`ENCODE` determines the **encoding of the member names**:
`UTF8`, `DOS` or `ASCII`. That is the place where umlauts in file names break, and it is nearly always overlooked.
And comments: `MBRCMT` per member, `ARCCMT` for the archive. FLAM writes a member note with its version by itself -- you see it in the `unzip` listing.



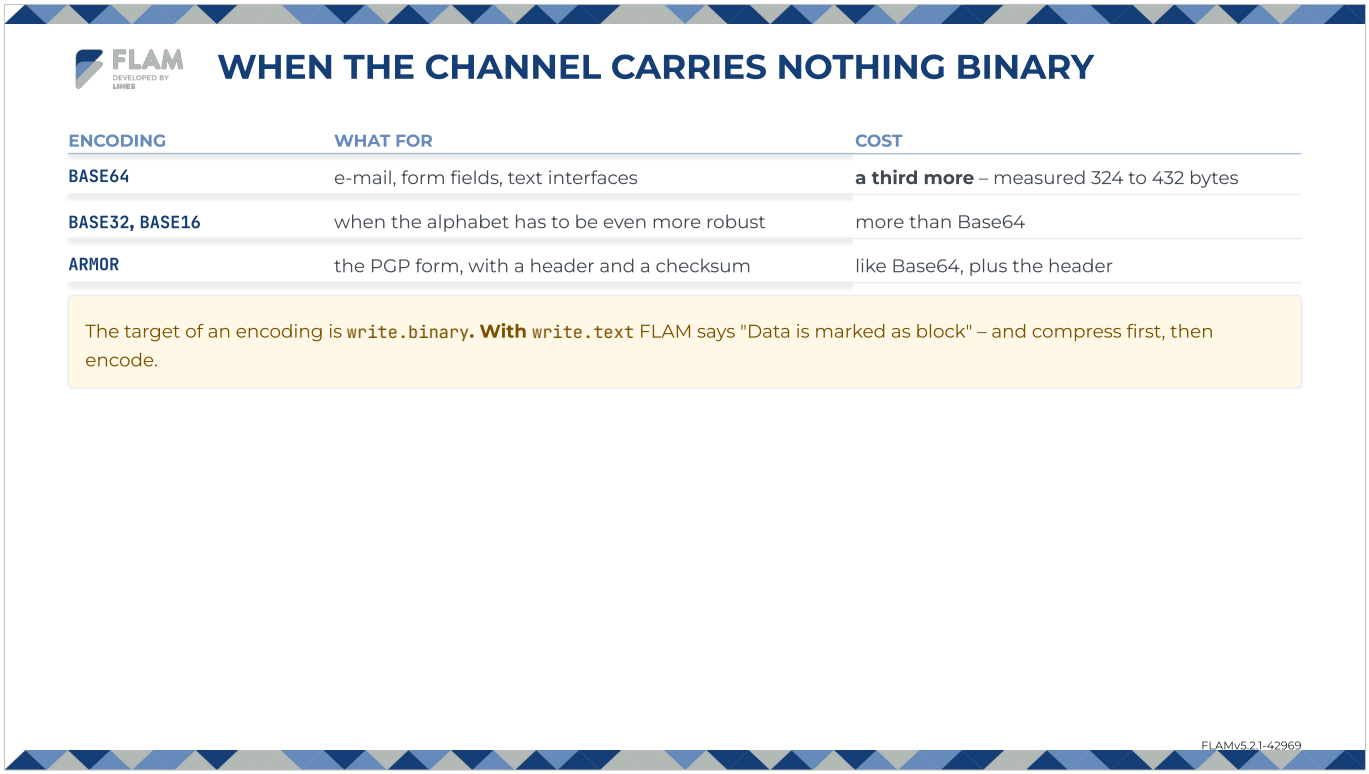
A short slide about a mistake that makes a ZIP useless without reporting an error. The difference is finer than it sounds: `**ARCHIVE.ZIP(ALGO=...)**` **compresses the members the way a ZIP provides for.** Another tool unpacks them quite normally with that. Taking a **compression stage of your own** in the chain instead -- that is, `COMP.GZIP()` and the ZIP after it -- then a file lies in the ZIP that is **uncompressed** from the ZIP's point of view and in truth is a GZIP stream. Our manual page says so. The result: the ZIP can be opened, the member can be unpacked -- and what comes out is a `.gz` nobody expected. The other direction is friendlier: when **reading** out of a ZIP, members compressed with GZIP, BZIP2 or XZ are unpacked automatically. Whoever does not want that sets `SKPORG`. The sentence to remember: **the compression of a ZIP belongs in the ZIP object**, not in the chain in front of it.

WHEN A FLAMFILE IS THE RIGHT FORMAT

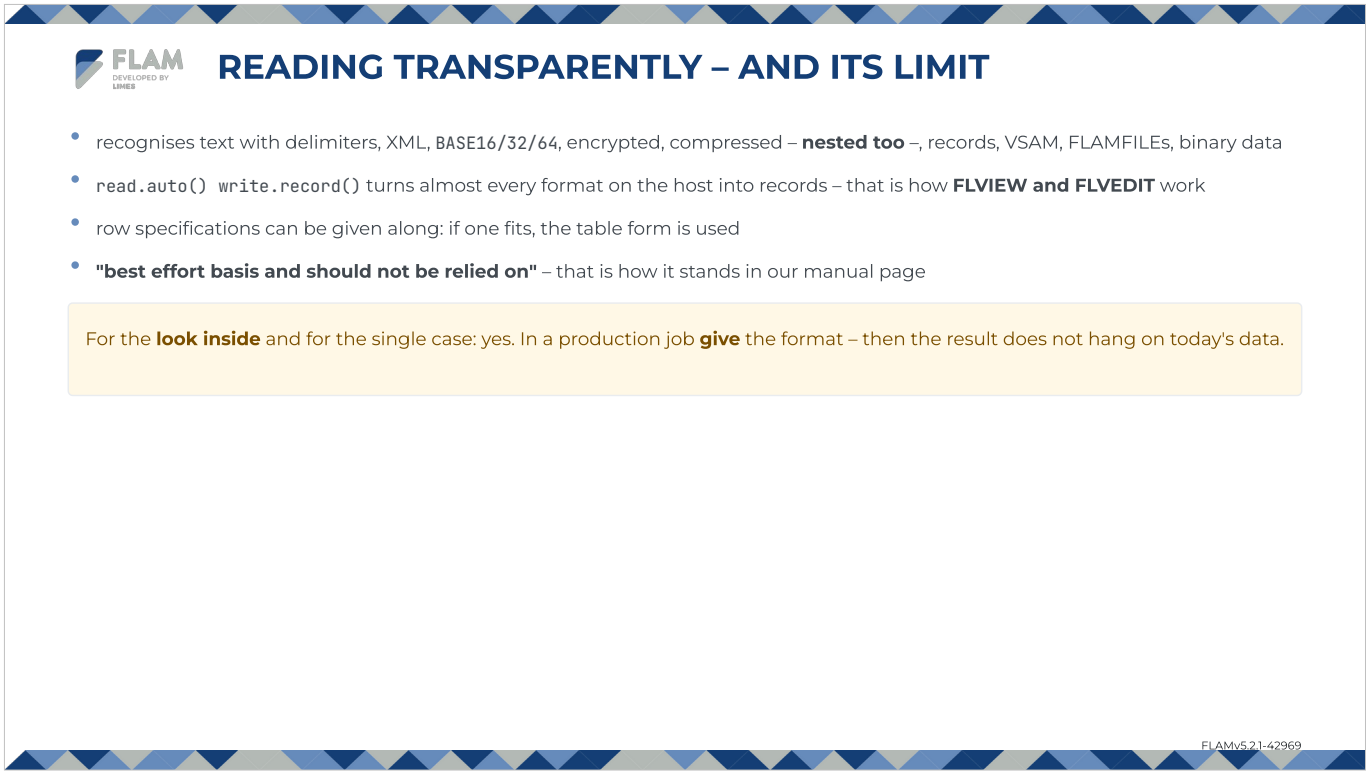
	CAN	NEEDS
ZIP	every tool unpacks it	nothing – the normal case for strangers
FLAMFILE	records with attributes, character set entries, encryption and signature, access per member	**FLAM on the other side**
FLAM5 archive	indexing, pots, two-stage access on top	FLAM5 – the route for new things

FLAMv5.21-42969

The slide that places our own world -- honestly, with both sides. A **FLAMFILE** is the FLAM4 exchange format and is reached through `ARCHIVE.FL4()` or `FLAM4()`. It can do things a ZIP cannot: record-oriented data with its attributes, character set entries, encryption and signature in one container, and the access to single members without unpacking. And it has a condition that has to be said out loud: **the other side needs FLAM.** That makes it the right format for the exchange inside one house, between two sites or with a service provider that has FLAM too -- and the wrong one for an arbitrary recipient. For new undertakings the note from the archive unit holds: the **FLAM5 archive** is the successor, with indexing, pots and the two-stage access. You take a FLAMFILE when the other side is on FLAM4. The sentence for the room: the choice is not a matter of taste but a question of what is installed on the other side.



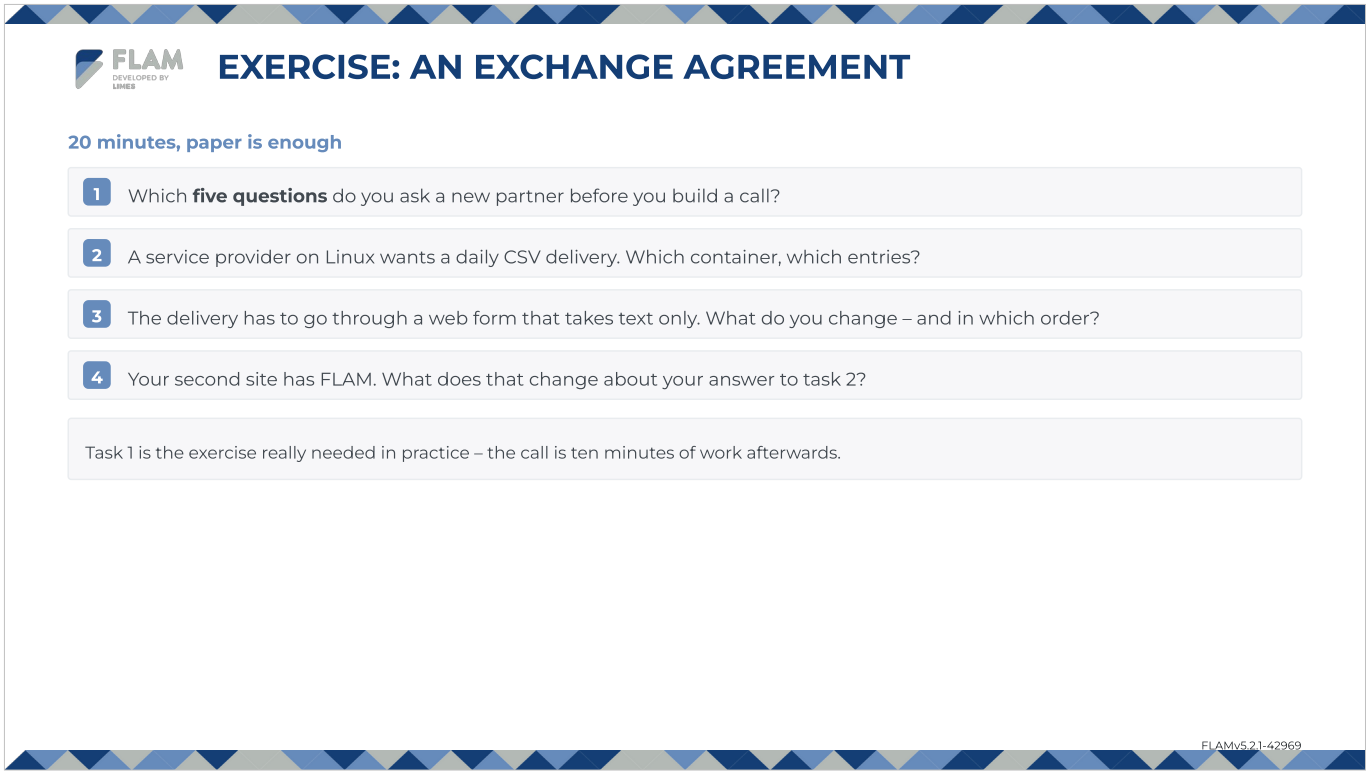
The slide for the cases where data is pushed through a bottleneck.
There are channels that carry no binary: an e-mail, a form field, an interface that expects text, or a route with character set conversion. For that there are the encodings: `BASE64` is the usual one, `BASE32` and `BASE16` are the variants with an even smaller alphabet, and `ARMOR` is the PGP form with a header and a checksum. The price is known and measured: **Base64 costs exactly a third more space** -- 324 bytes become 432. Base32 and Base16 cost more, in exchange their alphabet is even more robust. And a small thing that costs time if you do not know it: the target of an encoding is `write.binary`, not `write.text`. Measured, FLAM otherwise says "Data is marked as block, text formatting is not possible" -- the message is precise, but you have to have seen it once. The advice: compress first, then encode. The encoding inflates, so it belongs at the end of the chain -- and then it inflates the compressed result, not the raw data.



The slide with the most comfortable tool of the whole course and the clearest warning. `READ.AUTO()` tries to recognise the format itself, and it can do a lot: text with delimiters, XML, encoded data (Base16/32/64), encrypted or compressed files -- **nested too** --, record-oriented data sets, VSAM, FLAMFILEs and binary data. With that, `read.auto() write.record()` turns practically every supported format on the host into records; that is exactly what the ISPF commands FLVIEW and FLVEDIT use. And now the sentence out of our own manual page, word for word: **"Detection works on a best effort basis and should not be relied on."** The detection can fail, especially with very small files. Out of that follows the rule to pass on: `read.auto()` is the tool for the **look inside** and for the single case. In a production job that reads the same format every night, give the format -- then the result does not hang on what was recognisable today. An intermediate route is in the manual page too: `read.auto()` can be given a list of row specifications. If one fits, the table form is used; if none fits, the normal detection runs.



All four out of real exchanges.

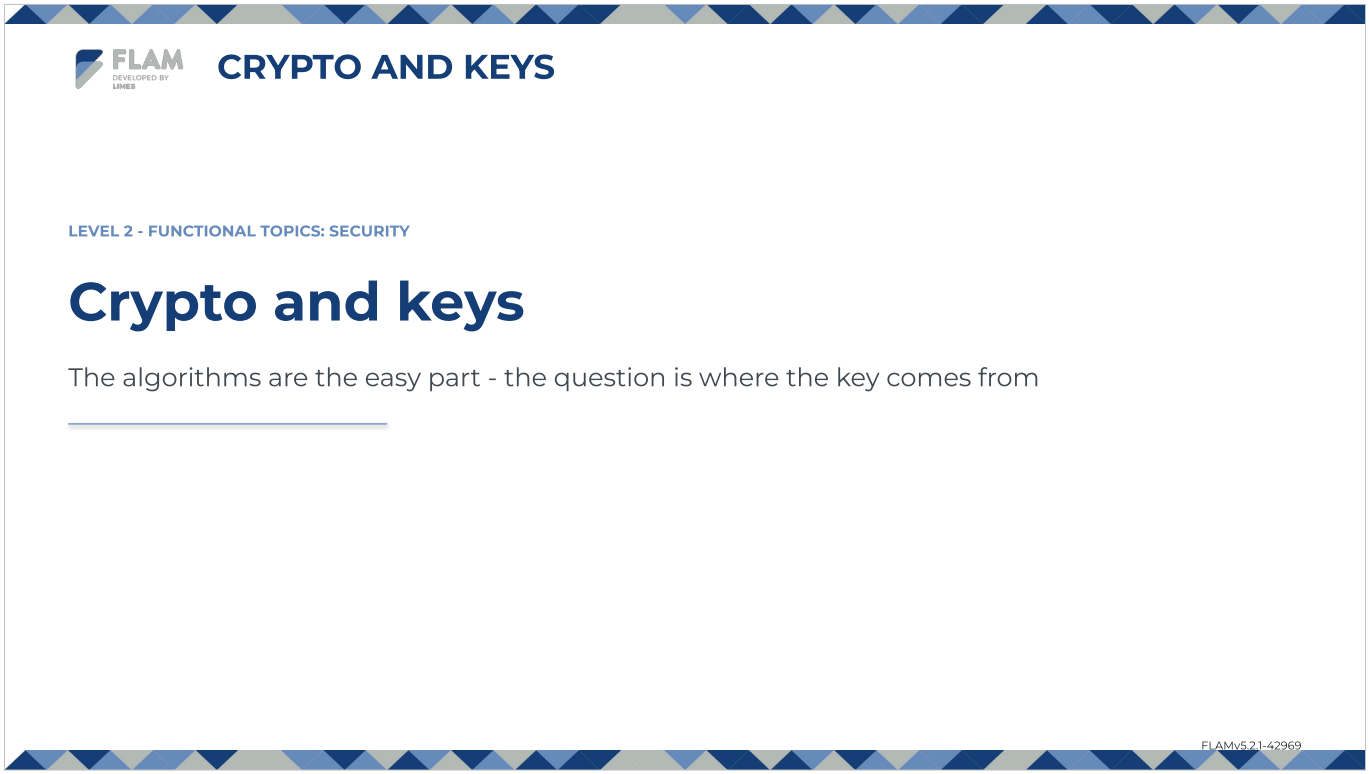


Twenty minutes. The exercise is a conversation, not a call -- and that is exactly what the work is in practice. The solutions: **Task 1.** The five questions:
container, compression, encoding, character set, record format. Whoever has answered all five has the agreement.
Task 2. ZIP with `ALGO=DEFLATE`, `FORMAT=AUTO`, `ENCODE=UTF8` for the member names -- and no umlauts in the names when in doubt. The data itself in UTF-8 with Unix line ends, because the recipient sits on Linux.
Task 3. Base64, and the target is `write.binary`.
Compress beforehand, so that the third of overhead goes on the compressed result and not on the raw data. **Task 4.** A FLAMFILE or a FLAM5 archive -- but only because FLAM stands on both sides. The gain is the access per member and the encryption in the container; with a stranger it would be the wrong answer.

WHAT TO TAKE AWAY

- **five questions** are the agreement: container, compression, encoding, character set, record format
- **ZIP with** `DEFLATE` **for strangers – and** `ENCODE` for the member names
- the compression belongs **in** the ZIP object, not in the chain in front of it
- a **FLAMFILE** can do more but needs FLAM on the other side – for new things the FLAM5 archive
- `BASE64` costs **a third** and belongs at the **end** of the chain; the target is `write.binary`
- `READ.AUTO()` is **best effort** – for the look inside, not for production

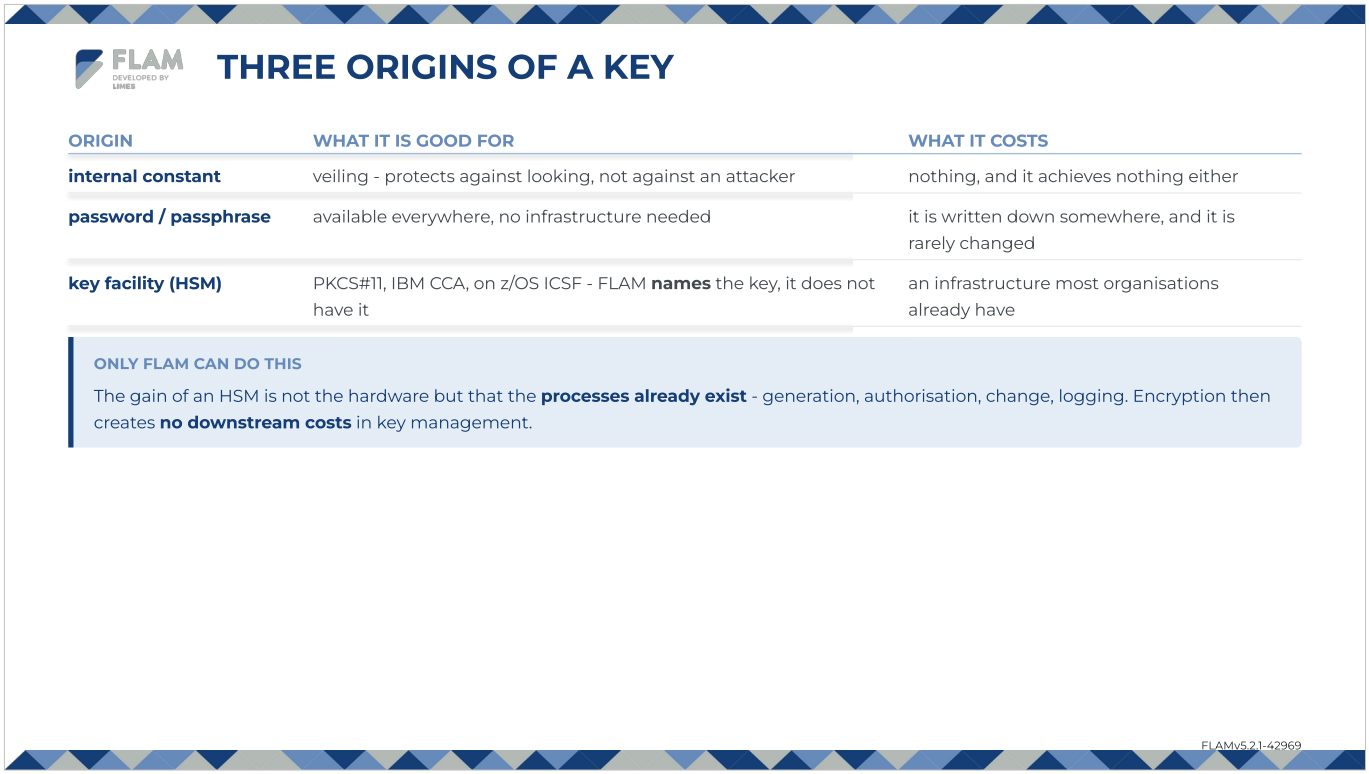
Go through the learning objectives. And with that the data group ends: conversion, tables, character sets, compression, record formats, XML and the exchange. The arc ran from the chain through the structure to the question of who can read it in the end.



First unit of the security group. And the first sentence should put the expectation right: **the algorithms are the easy part.** AES, SHA, RSA -- much is said about them, and they are rarely the problem. The problem is the key: where does it come from, who has it, how is it changed, and what happens when it goes missing.
This unit answers exactly that, and in the order an auditor asks.
What is **not** covered here: the three keys of a FLAM5 archive and the pots -- those are in the units *The FLAM5 archive* and *Pot design*. Here it is about the crypto itself.



The learning objectives as a contract. The second is what the unit leads up to: a password in the parameter string is the worst of three routes, and the other two are there. The fourth is the most important one for an auditor -- he wants to know that the implementation is checked, not that it exists.

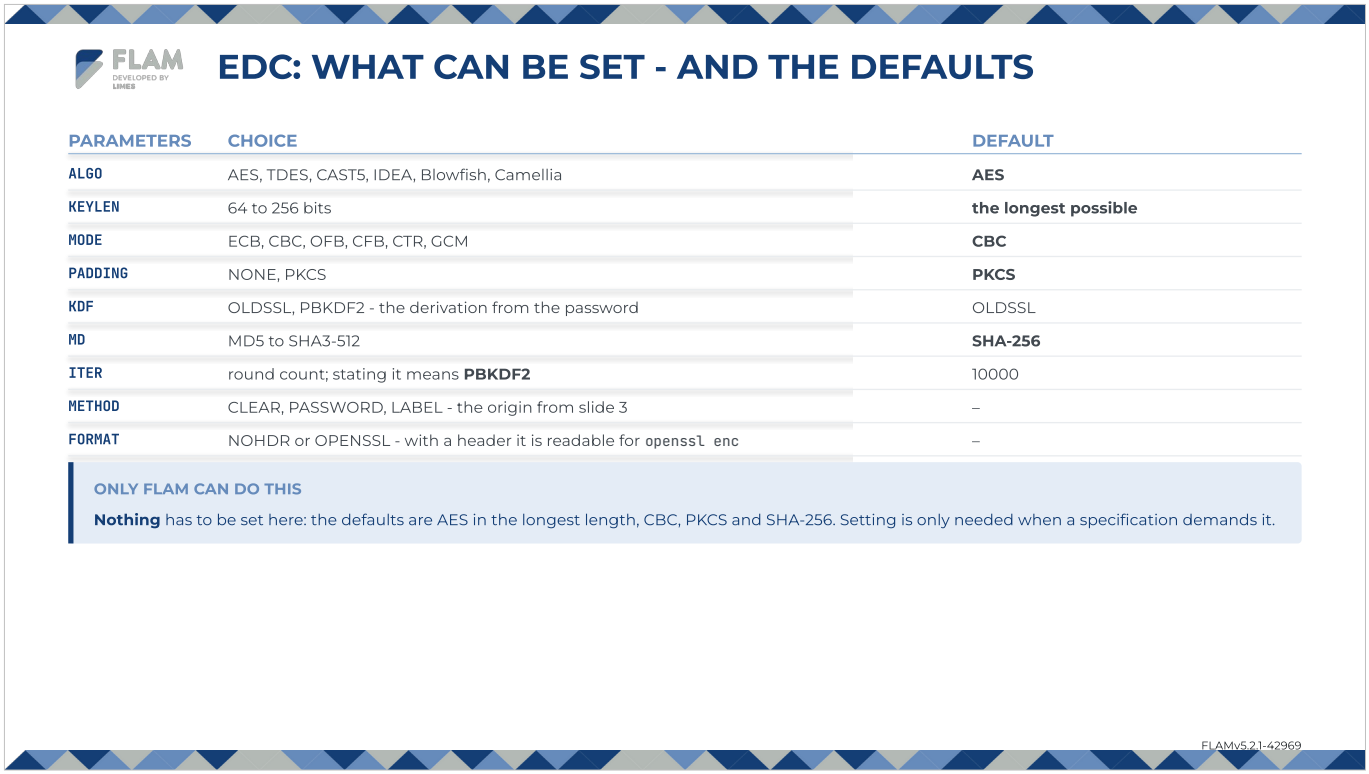


The map of the unit, and the order is a ranking -- from the bottom up it gets better. **The internal constant.** FLAM can protect data with a built in key. That is no secret and is not meant to be one: it protects against looking, not against an attacker. Its honest purpose is veiling -- so that a file is not readable by accident, for instance. **The password or passphrase.** Works everywhere, needs no infrastructure, and has two weaknesses: it is written down somewhere, and it is rarely changed. If a password, then in a parameter file, not in the command string -- see the unit *Security in operation*.

The key facility. That is the route we recommend: the key lies in the infrastructure the organisation already has -- an HSM over PKCS#11, IBM CCA or, on z/OS, ICSF -- and FLAM only names it. The advantage is not the hardware but that **the processes already exist**: generation, authorisation, change, logging. That is the sentence from our manual, and it is the real argument: "no downstream costs caused by encryption (reuse of key management)". The point for the room: in practice encryption hardly ever fails on the mathematics. It fails because nobody organised the key change.



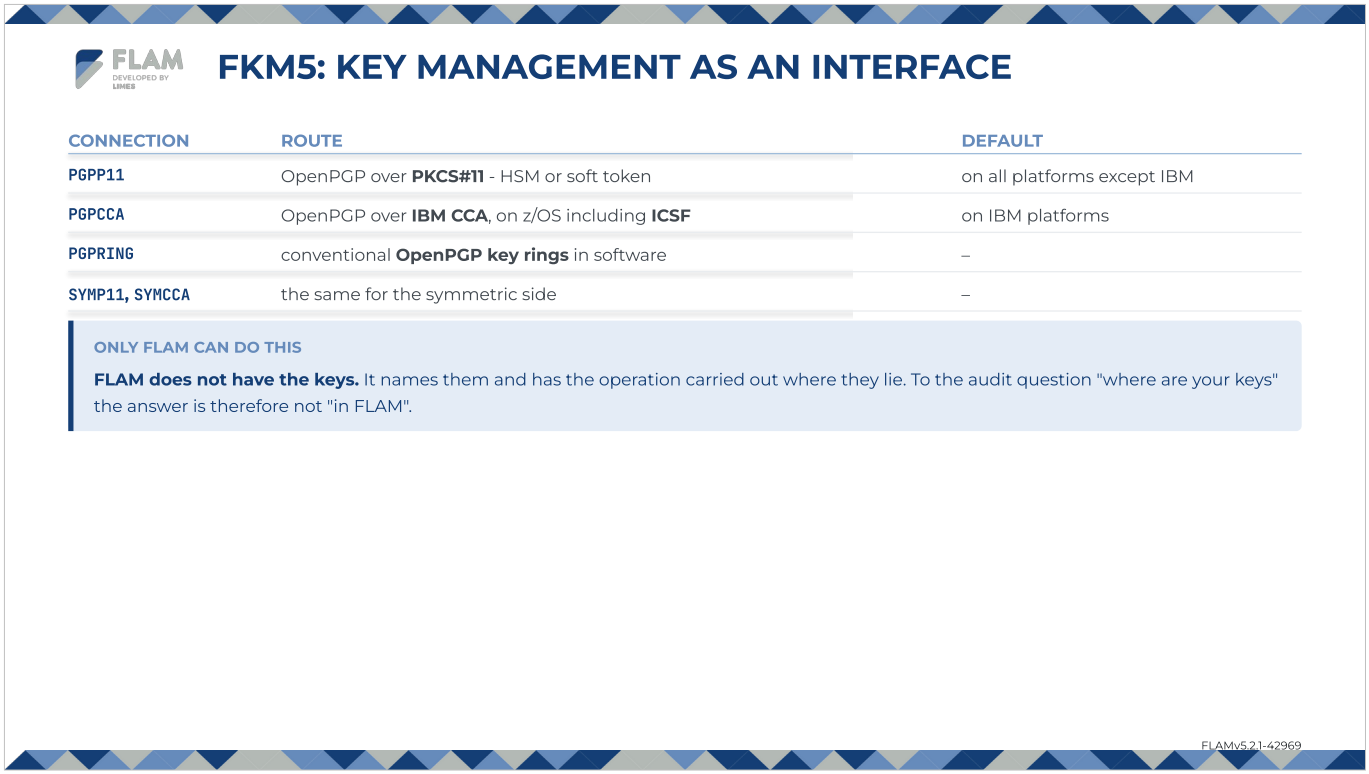
Now the tools. Exactly three objects hang on `ENCRYPT`, and the choice is not a matter of taste but of the occasion. `**PGP**` is OpenPGP to RFC 4880 -- the route when data **leaves the organisation**. The recipient needs to know nothing about FLAM, he only needs an OpenPGP client. There is a unit of its own for it.
`**PWD**` is password based encryption in the FLAM format. The route when data stays in house and it has to be quick. `**EDC**` is raw encryption -- "Raw encryption" is what the help says. Here everything is set by hand: method, key length, mode of operation, padding, derivation function. The occasion is almost always a **specification from outside**: a partner or a standard prescribes exactly one combination, and it has to be matched. The rule for it:
take `EDC` when you must, not when you can. Whoever sets everything himself can set everything wrong himself -- ECB without padding is syntactically allowed and cryptographically a mistake.



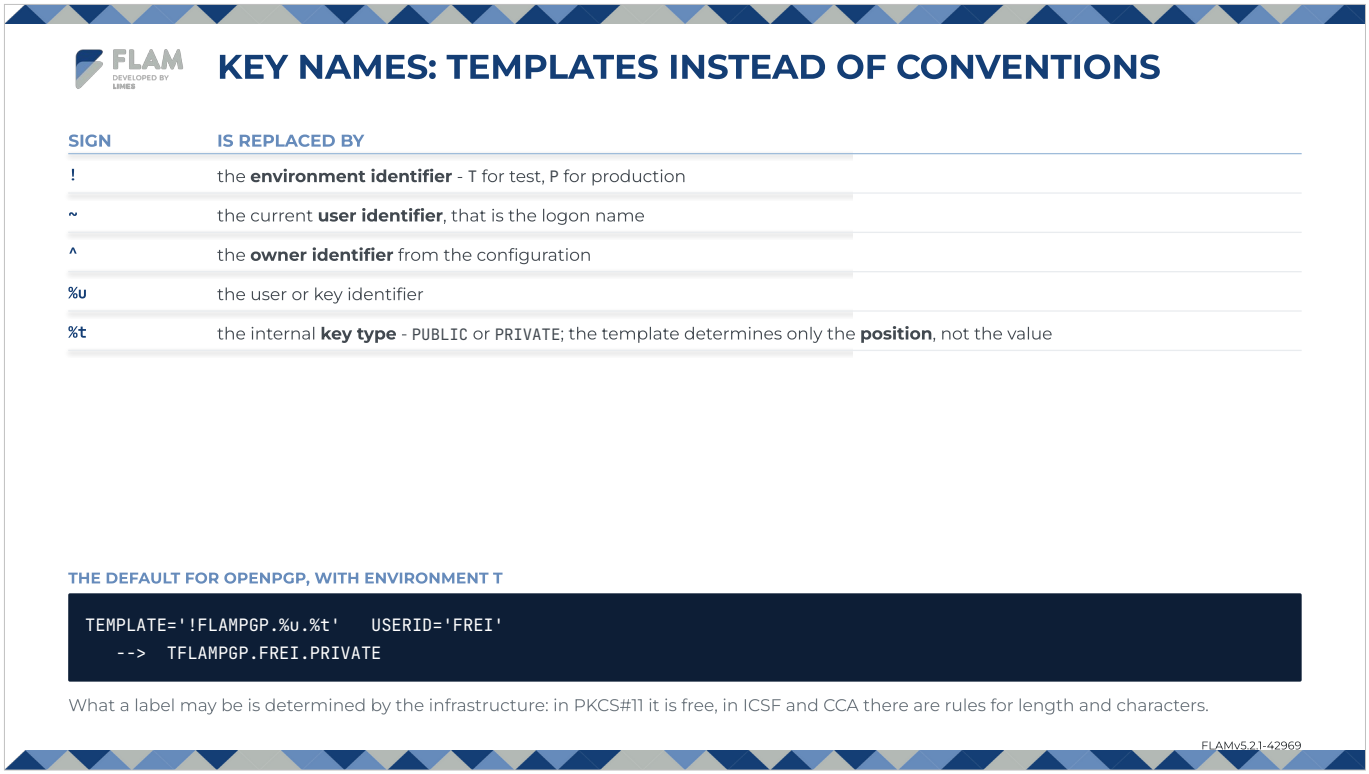
This slide is the reference slide, and the right hand column is the important one: **the defaults are sensible.** State nothing and you get AES in the longest possible key length, CBC, PKCS padding and SHA-256 for the derivation. That is exactly the combination one would choose by hand. Hence the sentence to introduce the slide with: **nothing has to be set here.** It can be, when a specification demands it. On the values one by one, as far as they are asked about: `METHOD` says where the key comes from -- `CLEAR` for the internal constant, `PASSWORD` for the passphrase, `LABEL` for the key facility. That is slide 3 in one parameter. `FORMAT` decides whether a header is written: `OPENSSL` produces the format `openssl enc` reads -- handy when the other side has no FLAM.
`NOHDR` writes the cryptogram only, and then the other side has to know every parameter.
Among the algorithms there are old ones too -- TDES, CAST5, IDEA, Blowfish. They are not there as a recommendation but because a partner can prescribe them. Anyone building anew takes AES.



The slide that clears up a common misconception: a password **is** not a key. It has to be made into one first, and how that happens decides how attackable it is. `KDF=OLDSSL` is the old OpenSSL method: a hash over password and salt, a few rounds. Fast -- and that is exactly the problem, because it is fast for an attacker too. `KDF=PBKDF2` turns it into a computing task: `ITER` rounds, ten thousand by default. For the authorised user that costs a few milliseconds once. For an attacker trying millions of passwords it costs ten thousand times as much. And the convenient part: **whoever states `ITER` gets PBKDF2 automatically.** Both need not be set. The sentence for the room: **a long password with a fast derivation is worse than a short one with a slow derivation.** If passwords are used in the organisation, `ITER` is the one statement not to leave out. Whoever wants to avoid the discussion entirely takes the key facility -- then there is no password to derive anything from.



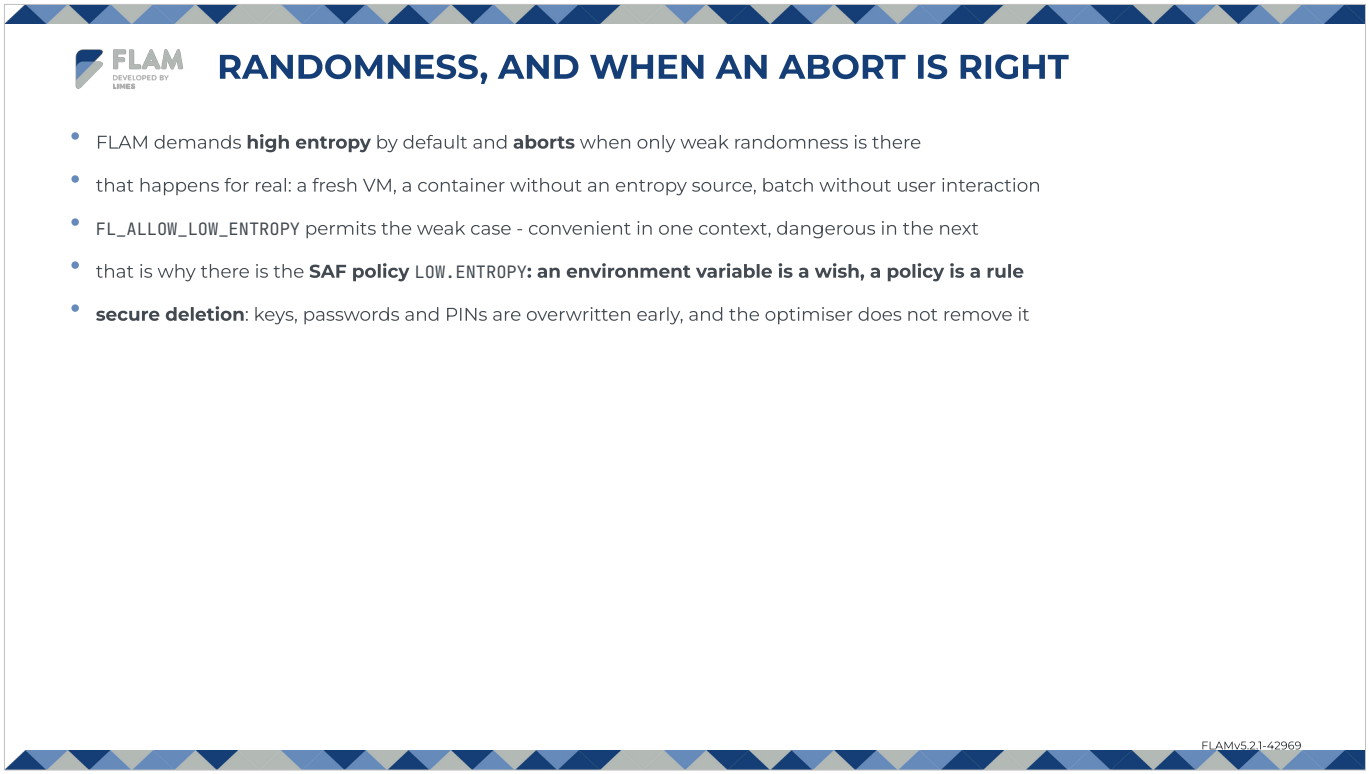
Now the key facility, and this is the slide that counts in the conversation with the security department. **FKM5 is an interface, not an implementation.** FLAM brings connections with it, but the interface is open -- an organisation can connect its own key management, and FLAM is then only the caller. Three routes are delivered, and the naming is simple: the function name says which route. `PGPP11` is OpenPGP over PKCS#11, `PGPCCA` is OpenPGP over IBM CCA including ICSF on z/OS, `PGPRING` is the software route over conventional OpenPGP key rings. For the symmetric side there are `SYMP11` and `SYMCCA`. The default is platform dependent and sensible: `PGPCCA` on IBM platforms, `PGPP11` otherwise.
State nothing and you get the route that fits the machine.
The benefit, and this is the list from our manual that can be read out: use existing infrastructure, reuse existing processes for key management and authorisation, employ hardware modules -- and **no downstream costs caused by the encryption**, because the key management is paid for already. Plus conformance with PCI DSS. One more sentence on the boundary, because it avoids misunderstandings: FLAM does **not have** the keys. It names them and has the operation carried out where the key lies. Anyone asked in an audit where the keys are does not answer "in FLAM".



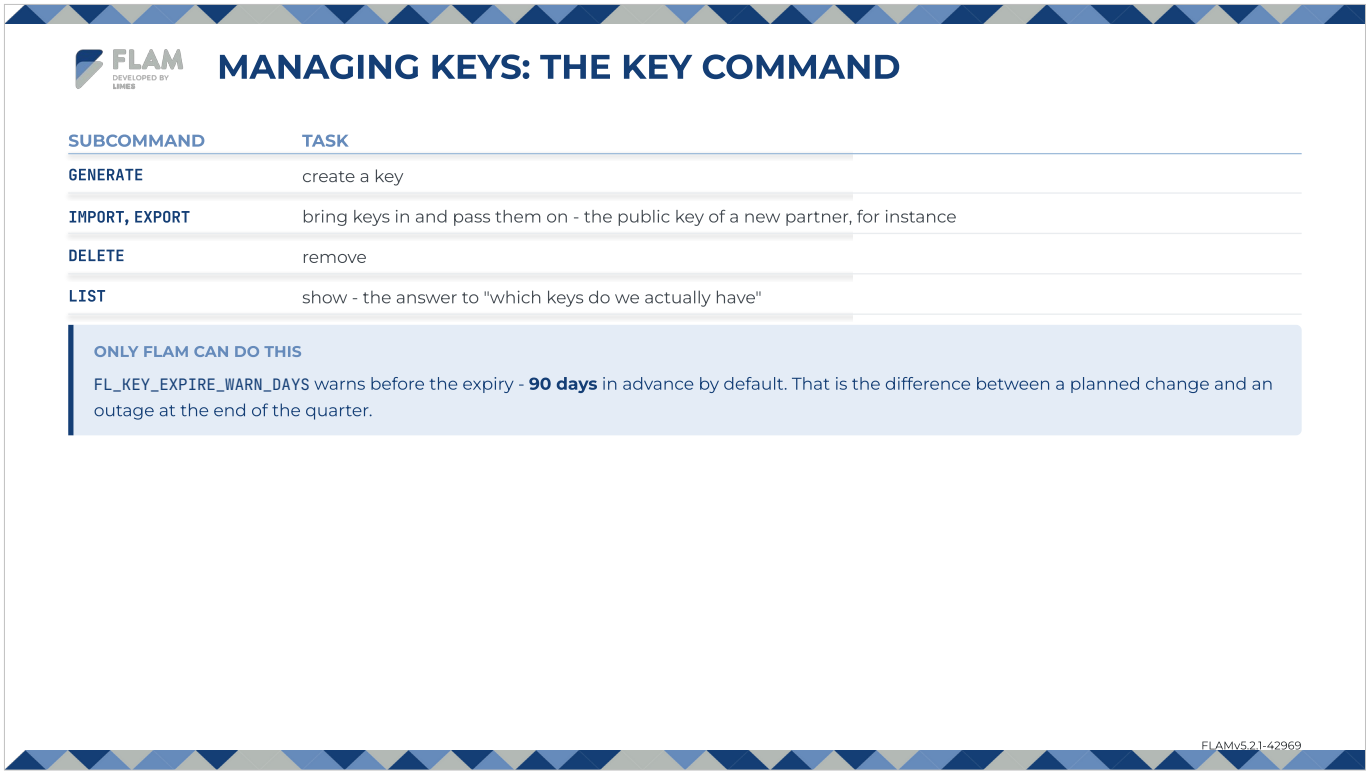
This slide solves a problem most people do not know is one -- until they put FLAM into an organisation with naming conventions. The situation: a key in an HSM is addressed by a **name**, the key label.
And every organisation has its own convention for it -- with environment, owner, application, type, in its own order. FLAM does not solve that with a convention of its own but with a **template**. Three characters for the environment and the identity: `!` is the environment identifier -- T for test, P for production. `~` is the current user identifier, that is the logon name. `^` is the owner identifier from the configuration. Plus two placeholders the application fills: `%u` becomes the user or key identifier, `%t` the internal key type -- PUBLIC or PRIVATE. **Important for the expectation:** the template determines only the **position** of the internal type, not its value. FLAM supplies that. The default for OpenPGP is `!FLAMPGP.%u.%t`. With environment T and the user FREI that becomes `TFLAMPGP.FREI.PRIVATE`. And the limit worth naming: what a label may be is determined by the infrastructure, not by FLAM. In PKCS#11 a label is freely chosen, in ICSF and CCA there are rules for length and characters. So the template cannot do everything -- it can only adapt to the convention of the organisation.



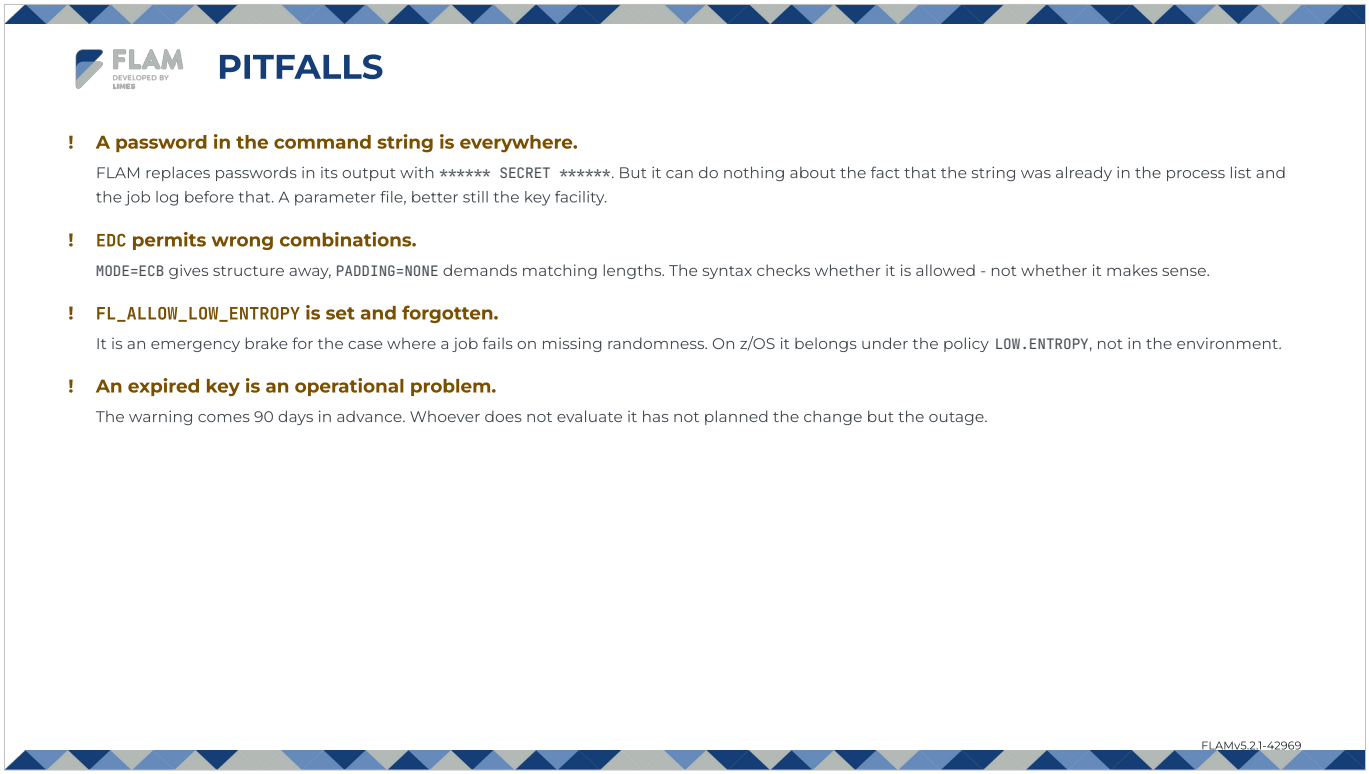
The slide for the auditor, and it holds a statement that is rare in this form. All software crypto runs through **one** central kernel, CryCore. It is a singleton and chooses the best implementation for the machine it runs on -- with hardware acceleration where there is any. On IBM mainframes those are essentially the CPACF functions, on all other platforms the `libcrypto` of OpenSSL. And now the point: **before an algorithm is used for the first time, a self test runs against the NIST test vectors.** So it is not believed that the loaded function implements AES correctly -- it is checked. That is the difference between "we use AES" and "we have looked to see that this AES is AES". Why that is more than diligence: the implementation can be a different one per machine. A wrongly detected hardware capability or a foreign `libcrypto` would otherwise compute wrongly in silence -- and wrongly encrypted data is noticed only on decryption, in the worst case years later. When an auditor asks about the crypto qualification, that is the answer. The environment variable `FL_DEACTIVATE_HW_CRYPTO_ACCELERATION` switches the hardware off and forces the software implementation.
That is a diagnostic tool, not an operational switch.



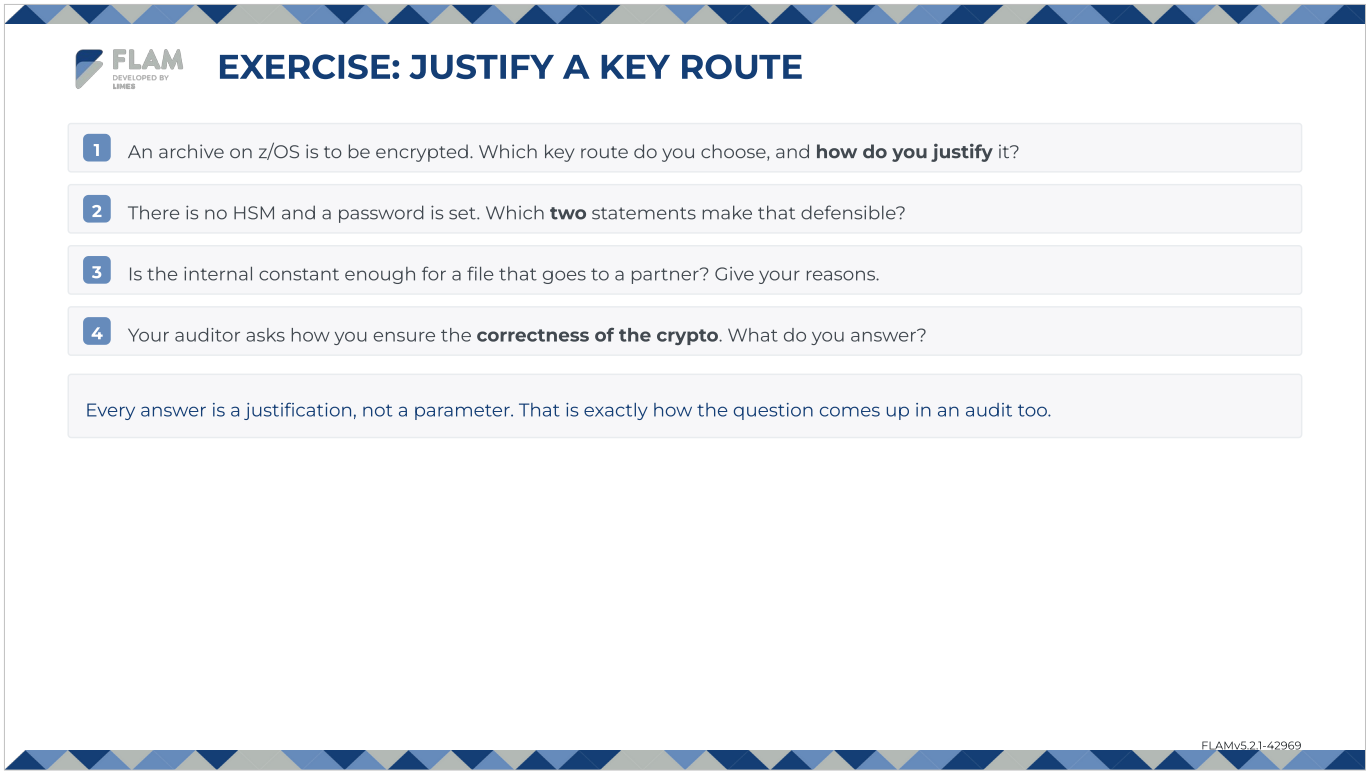
A small topic with a decision in it that should be taken deliberately. A key is only as good as the randomness it comes from. FLAM therefore demands **high entropy** by default -- and aborts when only weak randomness is available. That does happen in practice: a freshly started virtual machine, a container without an entropy source, a batch system without user interaction. The abort is the right default, because a key from weak randomness is a weak key -- and nobody notices that. `FL_ALLOW_LOW_ENTROPY` permits the weak case. And here is the trade-off that has to be made on site: there are contexts in which the quality of the randomness hardly counts -- there the variable is convenient. And there are contexts in which it is everything -- there it is dangerous. That is why there is a **SAF policy** `LOW.ENTROPY` for it: on z/OS the security officer can prevent a user from switching the weak case on. That is the pattern which recurs in the unit *SAF and RACF*: an environment variable is a wish, a policy is a rule. And the neighbouring point, because it belongs here: FLAM **deletes sensitive data securely** -- keys, passwords and PINs are overwritten as early as possible, and the overwriting is not optimised away. That is a small thing which is asked about regularly in audits.



Short, but it belongs here: FLAM can not only use keys but also manage them -- through `flcl KEY` with five subcommands.
`GENERATE` creates, `IMPORT` and `EXPORT` move, `DELETE` removes, `LIST` shows. All scriptable, so all automatable.
The purpose is not to replace the key management of the organisation -- it is meant to be reused, after all. The purpose is that the recurring steps can be put into a job chain: create a key for a new partner, import a public key, pull an inventory. And the point that counts in operation:
`LIST` is the answer to "which keys do we actually have". That question comes up in every audit, and without an inventory it cannot be answered. `FL_KEY_EXPIRE_WARN_DAYS` fits with it: FLAM warns before a key expires -- **90 days** in advance by default. That is the difference between a planned change and an outage at the end of the quarter.



Four, all from practice. On the first: a password in the command string ends up in the process list and often in the job log.
FLAM replaces it in its own output, but it can do nothing about the fact that it was already somewhere before that. On the second: `EDC` permits combinations that are cryptographically wrong. ECB gives structure away, `PADDING=NONE` demands matching lengths. Anyone using `EDC` should have a specification he keeps to. On the third: `FL_ALLOW_LOW_ENTROPY` is an emergency brake that is readily set when a job aborts -- and then forgotten. On z/OS it belongs under the policy. On the fourth: an expired key is not a crypto problem but an operational one. The warning period is there to be used.



Fifteen minutes. The tasks are built so that the answer demands a **justification** every time, not a parameter. The solutions:
Task 1. The key facility, that is `METHOD=LABEL` with FKM5 -- on z/OS over ICSF. Justification: the processes for generation, authorisation and change already exist, and the key does not leave the HSM. **Task 2.** `KDF=PBKDF2` with a high round count, and the password belongs in a parameter file with tight rights. The key facility would be better -- and if the argument "we have no HSM" comes up: PKCS#11 exists as a soft token too. **Task 3.** No. The internal constant is veiling, not protection. For a file that leaves the organisation it is unsuitable -- OpenPGP belongs there, because the recipient has a key of his own. **Task 4.** The self test against the NIST vectors before the first use, the central crypto kernel and -- on z/OS -- CPACF. And the answer to "where are the keys": not in FLAM.

WHAT TO TAKE AWAY

- three origins: **internal constant** (veiling), **password** (everywhere), **key facility** (recommended)
- the gain of an HSM is the **existing processes** - encryption then creates no downstream costs
- `PGP` for leaving the organisation, `PWD` for in house, `EDC` only when a **specification** demands it
- a password is not a key: `ITER` is the statement you do not leave out
- a **self test against the NIST vectors** runs before the first use of every algorithm
- **FLAM does not have the keys** - it names them through a template and has the computing done where they lie

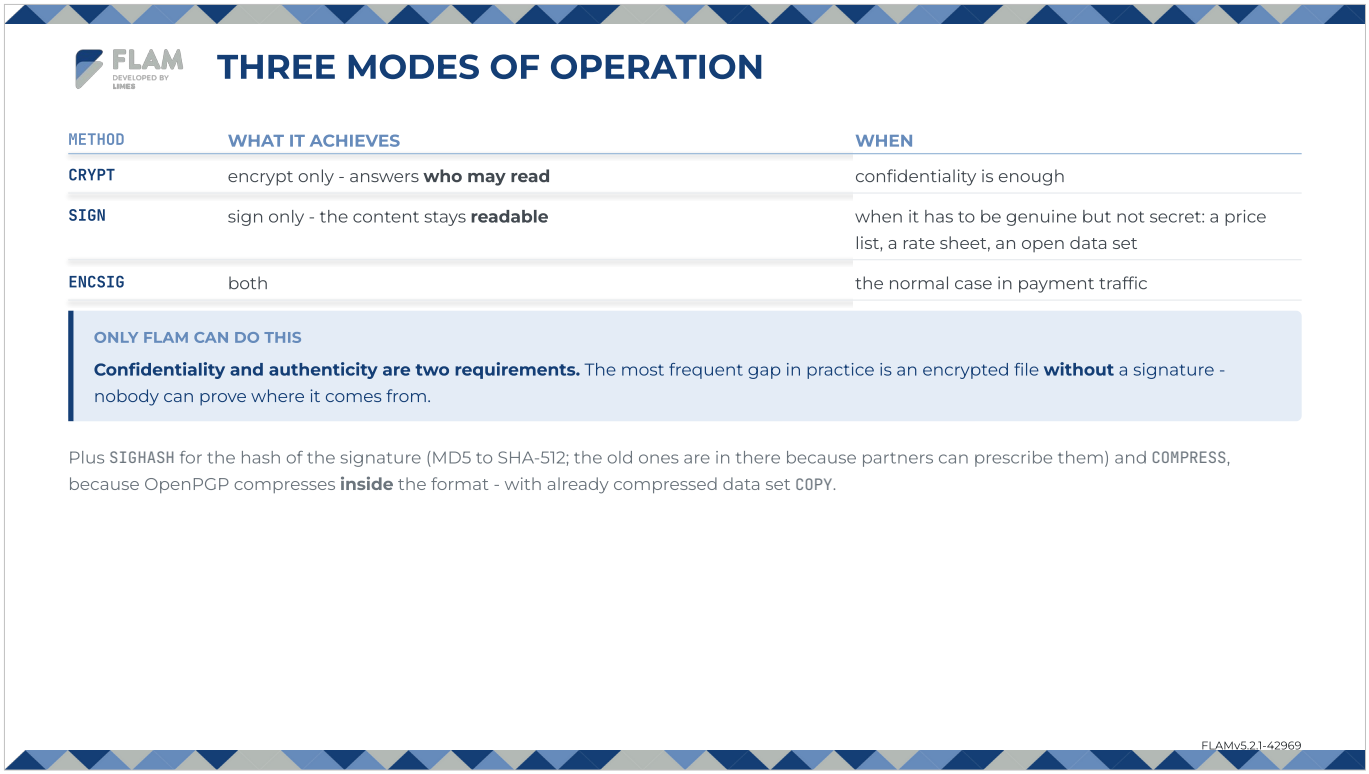
Go through the learning objectives. The closing sentence is the first sentence of the unit, and it should be proven by now: the algorithms are the easy part.



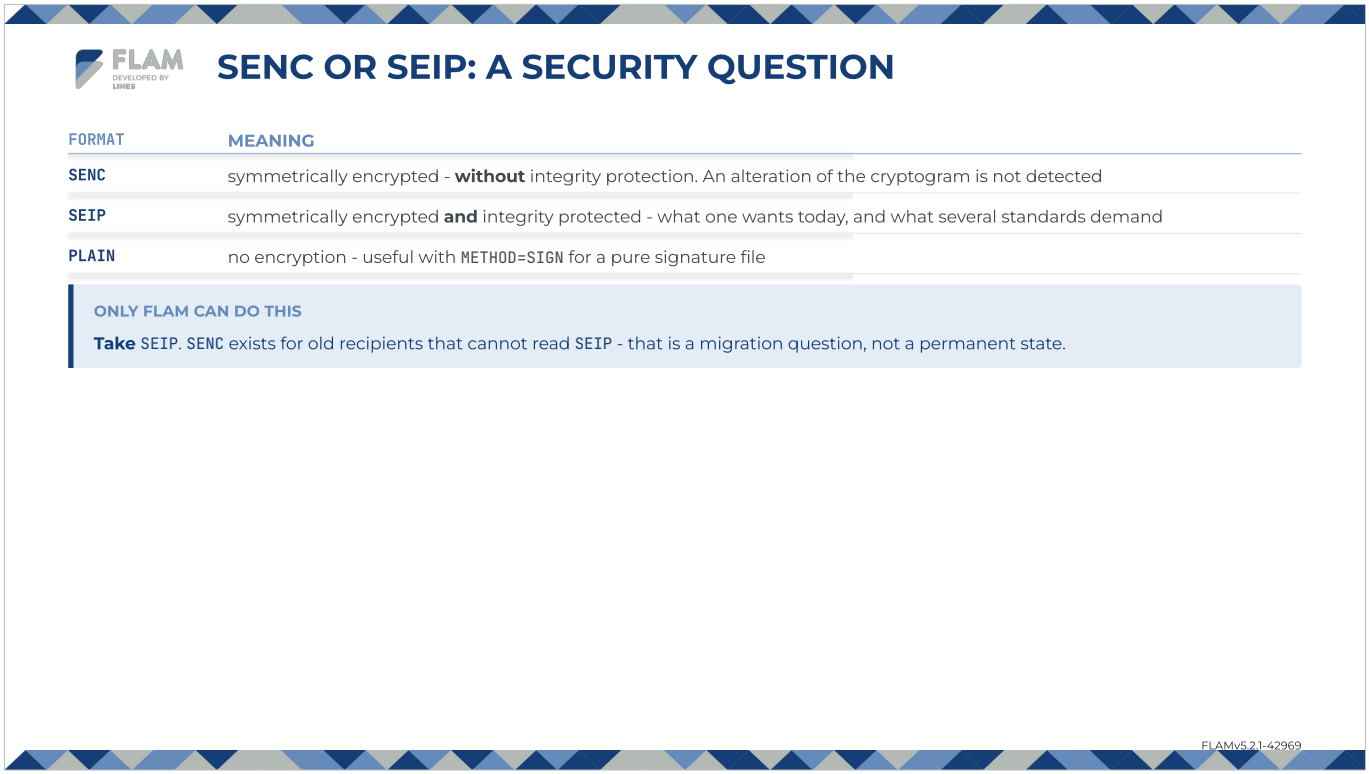
Second unit of the security group, and it continues the previous one: there it was about keys, here about the format **with which data leaves the organisation**. The frame in one sentence: OpenPGP is the only route where the recipient needs to know nothing about FLAM. He needs an OpenPGP client, and those exist everywhere. That is why PGP is the default in data exchange, not the FLAM format. And the second half of the unit, which is often overlooked: **signatures without encryption.** The question "was the data changed on the way" is a different one from "did anybody read it", and it is asked far more often.



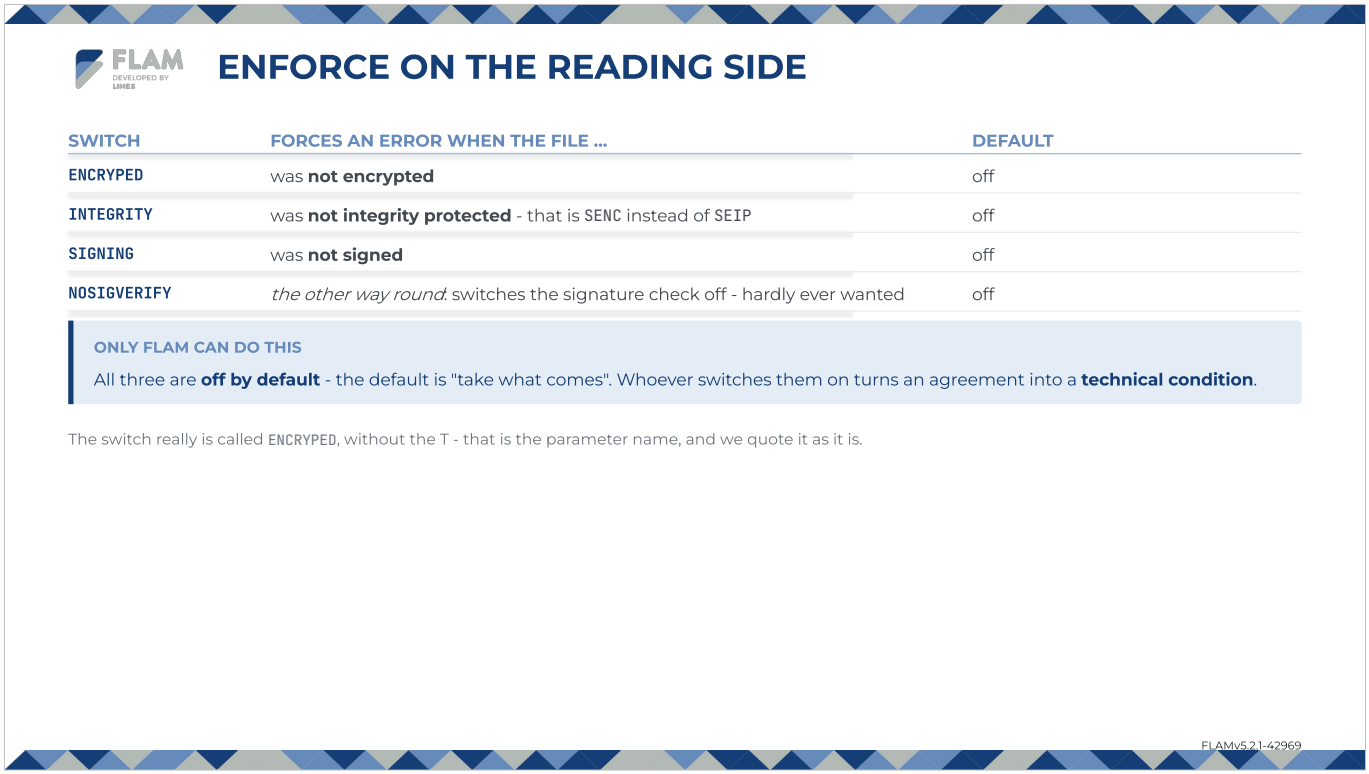
The learning objectives. The third is what distinguishes this unit from a product presentation: on the **reading side** it can be enforced what a file has to satisfy. That is the difference between "we accept PGP" and a policy. The fifth is the point hardly anybody expects -- the session key can be re-encrypted for a further recipient without touching the data.



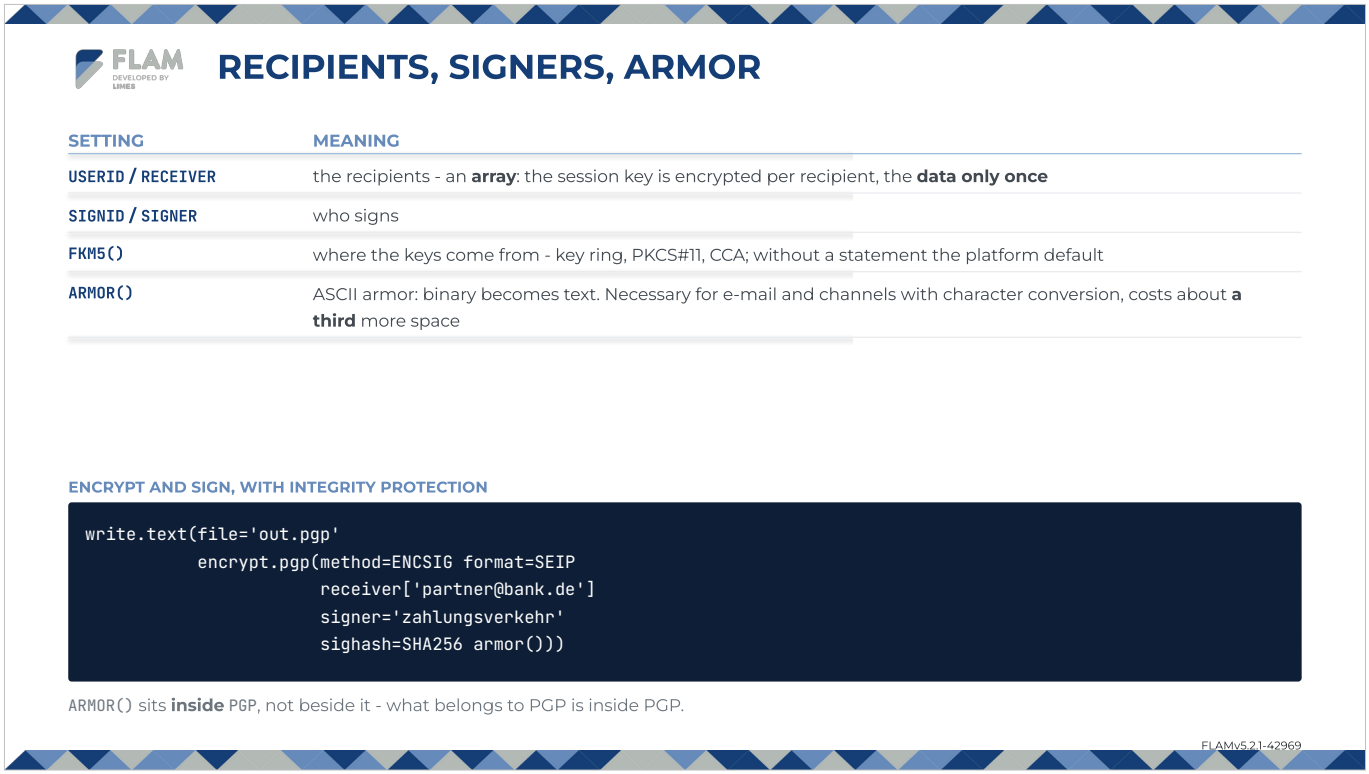
The first decision, and it is often not taken but inherited -- somebody wrote `ENCSIG` years ago, and that is how it has run since. `METHOD=CRYPT` encrypts only. The question "who may read it" is answered, the question "is it unchanged and from whom" is not. `METHOD=SIGN` signs only. The content stays readable -- and that is often exactly right: a price list, a rate sheet, a public data set need not be secret, but they have to be genuine. `METHOD=ENCSIG` does both. The normal case in payment traffic. The sentence to pass on: **confidentiality and authenticity are two different requirements**, and one should know which of them is currently being met. The most frequent gap in practice is an encrypted file without a signature -- nobody can prove where it comes from. `SIGHASH` belongs to the signing, the hash over the content. The choice runs from MD5 to SHA-512; MD5 and SHA-1 are in there because partners can prescribe them, not because they should be chosen. And `COMPRESS` deserves a word here: OpenPGP compresses **inside** the format. Anyone who has already compressed the data should set `COPY` here -- double compression costs CPU and gains nothing. `AUTO` is the convenient choice.



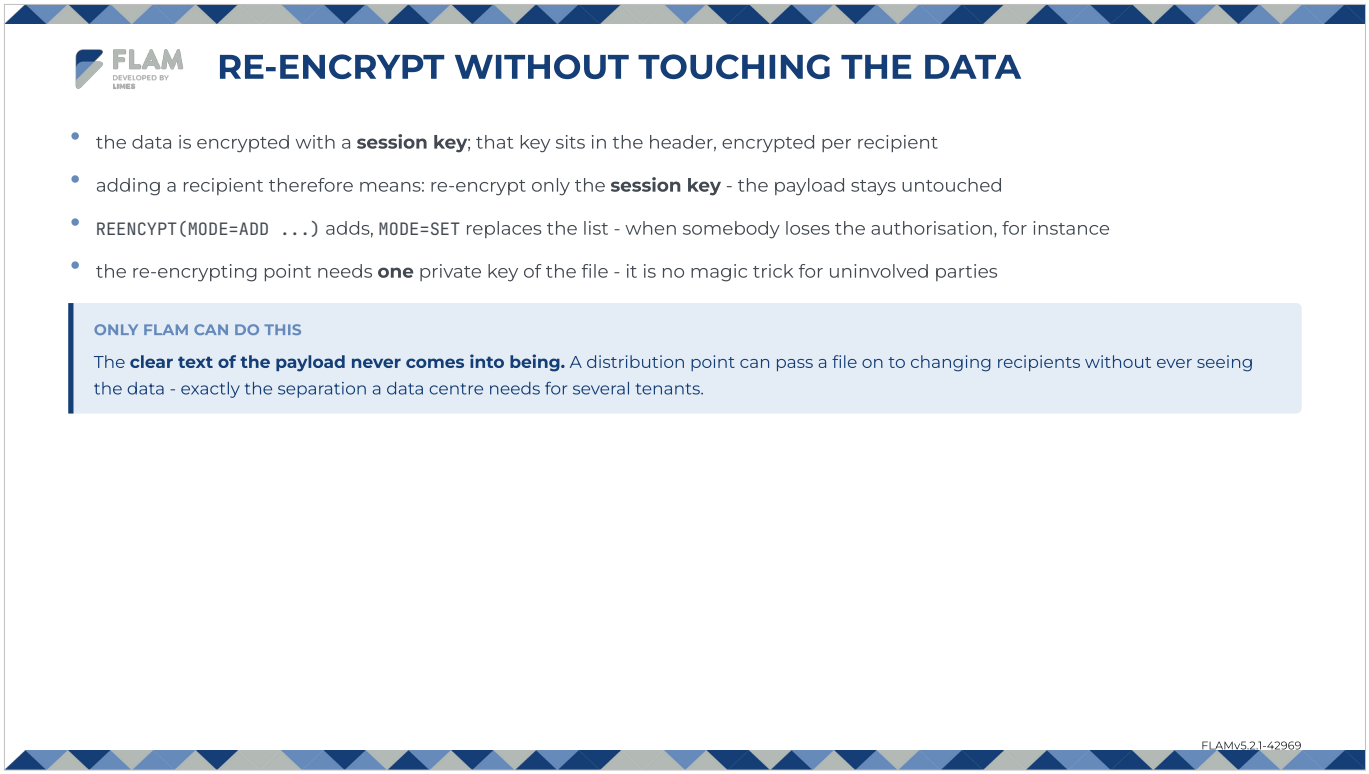
A small slide with a statement one has to know, because it turns up in standards. `FORMAT` has three values, and two of them encrypt: `SENC` is symmetrically encrypted. The content is secret -- but there is **no integrity protection**. An attacker who alters the cryptogram is not caught doing it; decryption then delivers rubbish or, in unlucky cases, altered data.
`SEIP` is symmetrically encrypted **and integrity protected**.
That is what one wants today, and several standards demand it explicitly. `PLAIN` does not encrypt -- useful together with `METHOD=SIGN` when a pure signature file in PGP format is to be produced. The practical advice: **take SEIP**, and enforce on the reading side that incoming files are integrity protected.
See the next slide for that. If the question comes up why `SENC` still exists at all: old recipients. A partner with a very old client cannot read SEIP. That is a migration question, not a permanent state.



This is the strongest slide of the unit, and it answers a question that comes up in every compliance conversation:
how do you make sure an incoming file satisfies the requirements? The situation: a PGP file may be encrypted or not, signed or not, integrity protected or not. Whoever simply decrypts it accepts everything -- including a file that was only signed and not encrypted, or one without a signature. On `DECRYPT.PGP` there are three switches for this, and they are the whole point: `ENCRYPED` forces an error when the file was **not encrypted**. `INTEGRITY` forces an error when it was **not integrity protected** -- that is SENC instead of SEIP. `SIGNING` forces an error when it was **not signed**. All three are **off** by default. And that is the trap: the default is "take what comes". Anyone who has a policy has to switch them on -- then it is no longer an agreement but a technical condition. Two further switches, so that the slide is complete: `NOSIGVERIFY` switches the signature check off -- one hardly ever wants that, and if so, with a reason.
`INTERIM` permits interim results that are not yet clear text. And on the spelling, because it stands out: the switch really is called `ENCRYPED`, without the T. That is the parameter name; we quote it as it is.



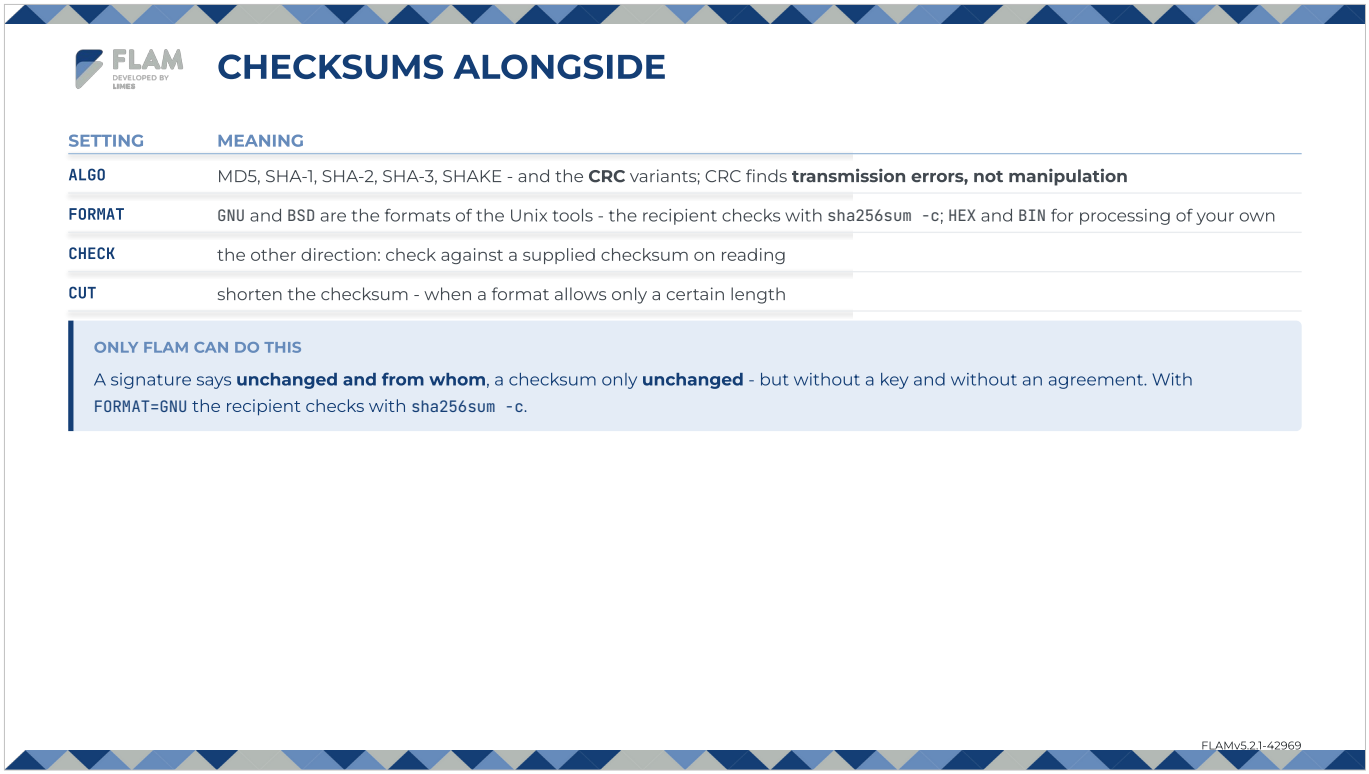
The reference slide for the call. `USERID` or `RECEIVER` names the recipients -- and it is an **array**. Several recipients means: the session key is encrypted for each of them separately, the data only once. That is why a file for ten recipients is not ten times as big. `SIGNID` or `SIGNER` names who signs. The keys themselves come through `FKM5()` -- key ring, PKCS#11 or CCA, see the previous unit. Without `FKM5` the platform dependent default applies. `ARMOR()` switches the ASCII armor encoding on:
the binary format becomes text. That is necessary when the transport cannot carry binary -- an e-mail, a form field, a channel with character set conversion. **It costs about a third more space**, so switch it on only when it is needed. The most frequent mistake with it: `ARMOR` sits **inside** `PGP`, not beside it. That is the structure of the language -- what belongs to PGP is inside PGP.



The slide that surprises -- and that solves a real operational case. The case: an encrypted file exists, and a **further** recipient is to be able to read it. The naive solution is: decrypt, encrypt again. In doing so the clear text exists once, and that is exactly what a relay point does not want. `REENCYPT` solves it differently. The data is encrypted with a **session key**, and it is this session key that sits in the header, encrypted per recipient. To add a recipient one therefore only has to encrypt the **session key** for him -- the payload stays untouched. `MODE=ADD` adds recipients, `MODE=SET` replaces the list. The latter is the route when a recipient is to lose the authorisation.
What it does **not** do, and that has to be said: the point that re-encrypts needs access to the session key -- that is a private key the file was encrypted for. It is not a magic trick with which an uninvolved party can add recipients. But the **clear text of the payload never comes into being**.
The practical benefit: a distribution point can pass a file on to changing recipients without ever seeing the data in the clear. For a data centre working for several tenants that is exactly the separation that is demanded.



The second half of the unit, and it is independent of PGP as a container format. The fact of the matter from our manual, word for word: a signature file holds **only the signature and not the file it signs**. And: the signed documents are **not changed** during generation and verification. Why that counts: the file stays exactly what it was. No header, no wrapper, no different format. A recipient who does not want to or cannot check the signature processes the file as always. And a check is possible afterwards -- years later too, on an archive medium. That is the difference from a PGP signed file: there the signature is **in** the file, and the file is thereby a PGP file. The call is symmetric:
`SIGNATURE.GENERATION` on the `WRITE`, `SIGNATURE.VERIFICATION` on the `READ`. Both hold a file name for the signature file, a format and the key origin through `FKM5()`. The format is currently **OpenPGP only** -- that should be said so that nobody asks for CMS or XMLDSig and is disappointed. The structure is built as an overlay, so it is extensible. The typical use: a file is transferred, and the signature lies beside it. The recipient checks it and thereby has the evidence of integrity and origin -- without the file itself having been touched.



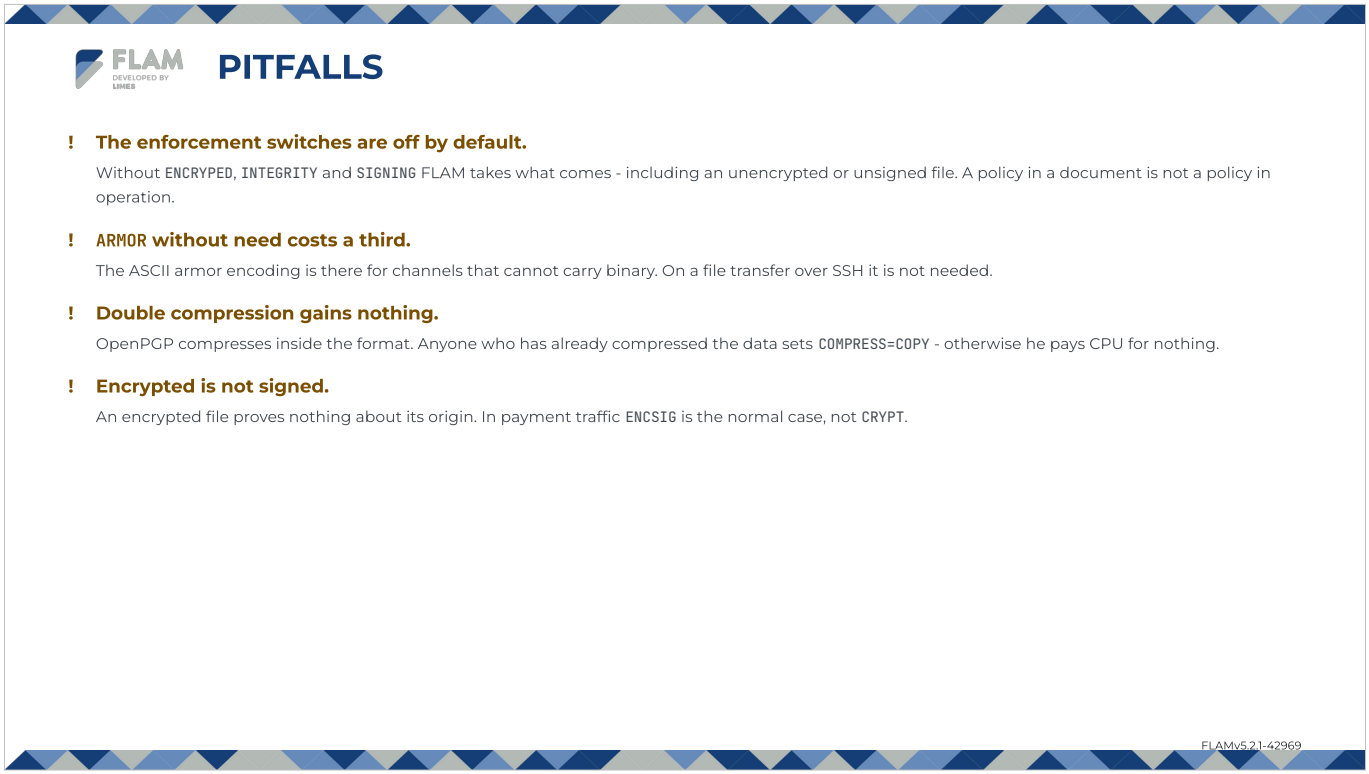
The little sister of the signature, and in practice often what is really needed. A signature answers two questions:
unchanged **and from whom**. A checksum answers only the first -- but it needs no key, no infrastructure and no agreement about formats. The `HASH` object on the `WRITE` produces it along the way, in the same run. The choice is broad: from MD5 and SHA-1 through the SHA-2 and SHA-3 families to SHAKE, plus the CRC variants. The CRCs are not crypto -- they find transmission errors, not manipulation.
The difference should be made clear in the room, because it gets blurred regularly. `FORMAT` determines how the checksum is written: `GNU` and `BSD` are the formats of the Unix tools -- with them the recipient can check with `sha256sum -c` without knowing anything about FLAM. `HEX` and `BIN` are for further processing of your own. And `CHECK` is the other direction: check against a supplied checksum on reading. The sentence for the room: **if you only want to know whether the transmission was clean, a checksum is the right tool** -- and `FORMAT=GNU` makes it checkable by anybody.

KEYS EXPIRE

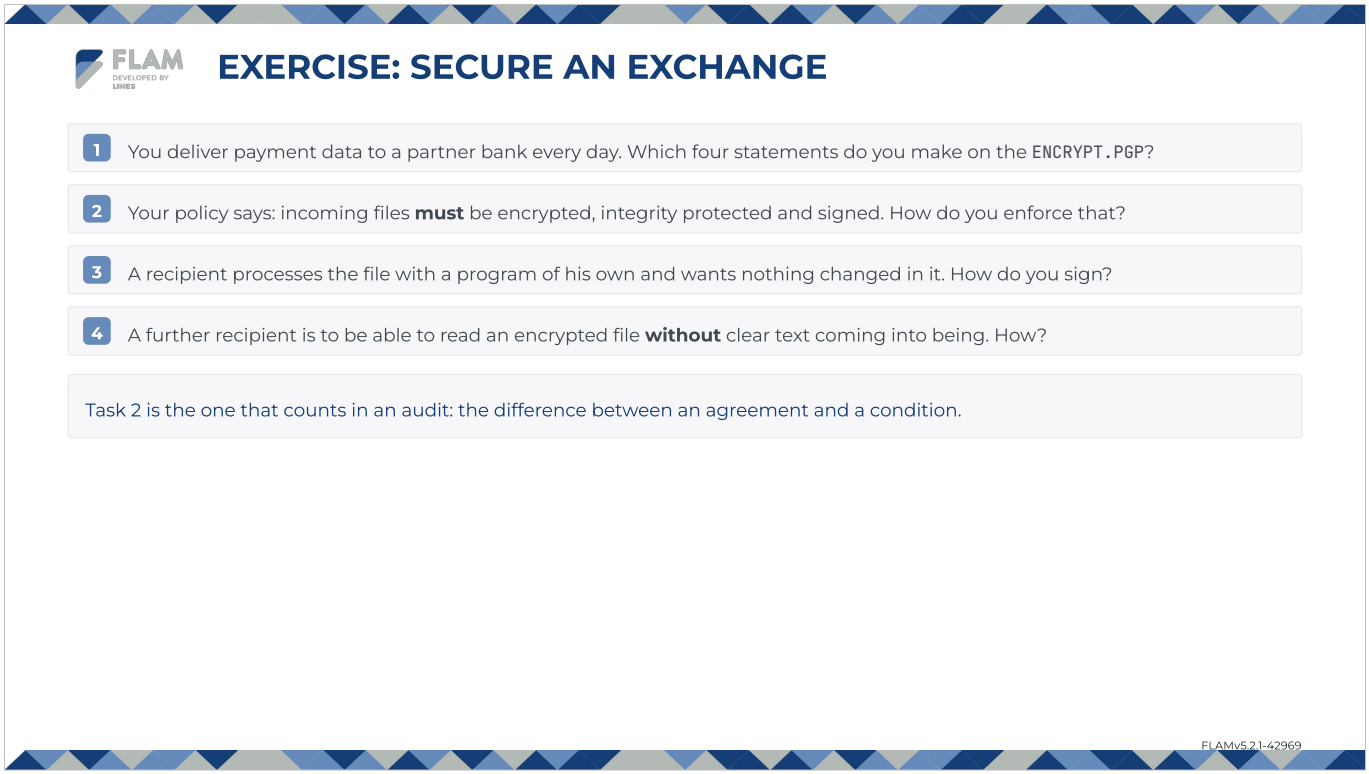
- OpenPGP keys have an **expiry date** - that is intended, and it is the most frequent cause of an exchange that suddenly stops
- `FL_KEY_EXPIRE_WARN_DAYS` sets the warning period, **default 90 days**
- three months are enough to exchange a key with a partner - even across two security departments
- **somebody has to read the warning**: it is in the log, and whoever looks at the log only on errors never sees it
- `flcl KEY LIST` pulls the inventory - all expiry dates at once instead of one by one in the log

FLAMv5.21-42969

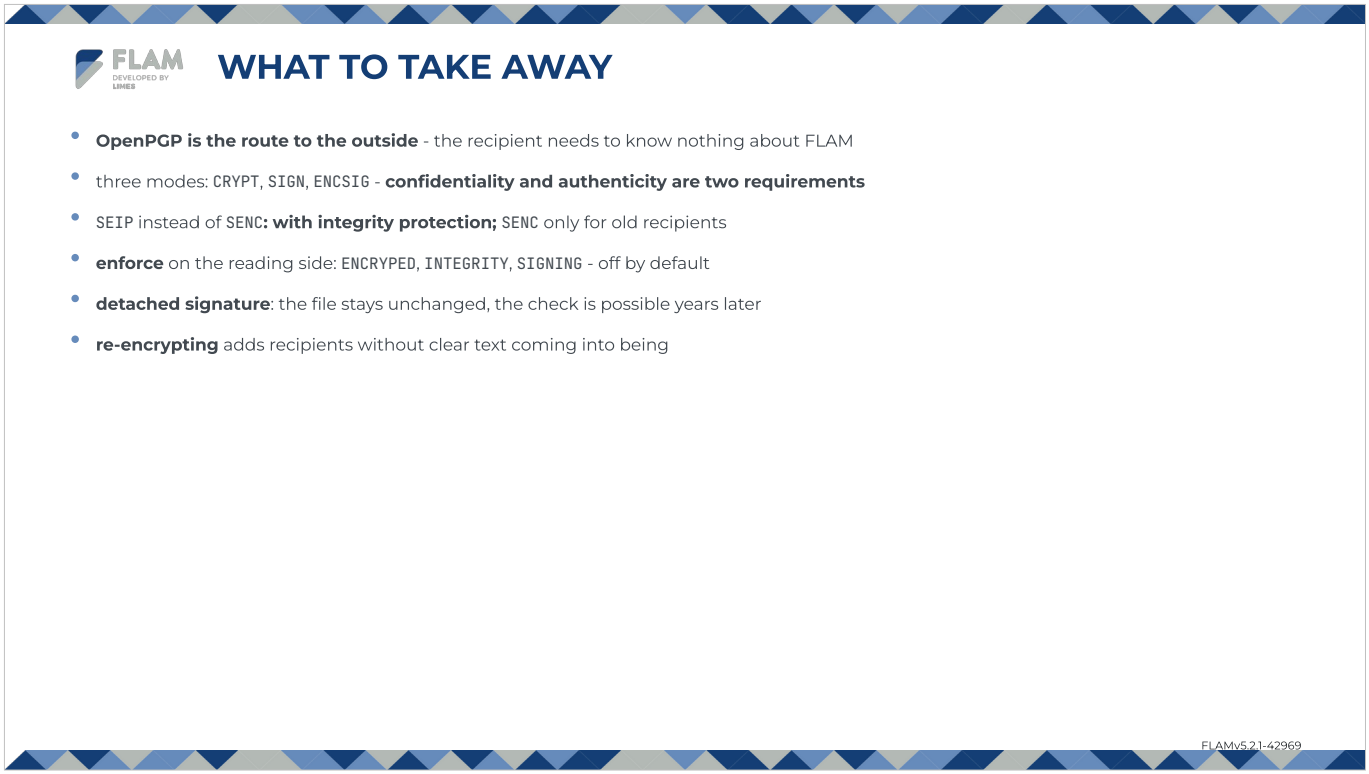
An operational slide, short, but it prevents an outage.
OpenPGP keys have an expiry date -- that is intended and good. In practice it is the most frequent cause of a data exchange that suddenly stops: the partner's key has expired and nobody was watching. FLAM warns beforehand.
`FL_KEY_EXPIRE_WARN_DAYS` sets how early -- **90 days** by default. Three months are enough to exchange a new key with a partner, even when it goes through two security departments. The point that follows: **somebody has to read the warning.** It is in the log. Whoever looks at the log only on errors never sees it -- and instead experiences the abort on the day of expiry. Practical advice: include the warning in the evaluation that runs anyway. And pull the key inventory through `flcl KEY LIST` regularly -- then the expiry dates of all keys are visible at once instead of one by one in the log.



Four, all from support cases. On the first: the enforcement switches are off. Anyone who believes he has implemented a policy because it is written in a document has not implemented it. On the second: `ARMOR` without need costs a third of the space and, with large files, transmission time accordingly. On the third: double compression. PGP compresses itself; anyone who has already compressed should switch it off. On the fourth: an encrypted file without a signature proves nothing about the origin. That is the gap most often found in an audit.



Fifteen minutes. The solutions: **Task 1.** `METHOD=ENCSIG`, `FORMAT=SEIP`, the recipients as an array, the signer named, `SIGHASH=SHA256`. `ARMOR` only when the channel demands it. **Task 2.** Set `ENCRYPED`, `INTEGRITY` and `SIGNING` on the reading side. The requirement is then technically enforced; a file that does not satisfy it leads to an abort and not to processing. **Task 3.** A detached signature:
the file stays unchanged, the recipient can process it without knowing about the signature, and the check is still possible later. A PGP signed file would be a PGP file and thus a different format. **Task 4.** `REENCYPT(MODE=ADD)` -- encrypt the session key for the new recipient. The payload is not touched and the clear text does not come into being. Precondition: the point has a private key of the file.



Go through the learning objectives. The closing sentence:
OpenPGP is the route to the outside, and the enforcement switches are the difference between an agreement and a policy.



Third unit of the security group, and the least spectacular -- but the one an auditor asks about first. Not "which algorithms" but **how is it operated**. The frame in one sentence, and it should be said plainly: FLAM can help to meet many requirements -- PCI DSS, ISO 27001 and others. **But there are points the operator has to observe**, otherwise the encryption is only decoration. Those points are the content of the unit. A second sentence for honesty: some capabilities of FLAM are **backdoors on purpose** -- the virus scan sees clear text, the pre- processing runs programs, the packet capturing records network traffic. All of that is intended and useful. One only has to know that it is there, and close it where it is not needed.



The learning objectives. The second is the core:
there are four places where clear text or a command leaves FLAM, and all four are useful. Whoever knows them can close them. The fourth is the practical benefit for operations -- an installation check that costs two minutes and rules out a class of errors.



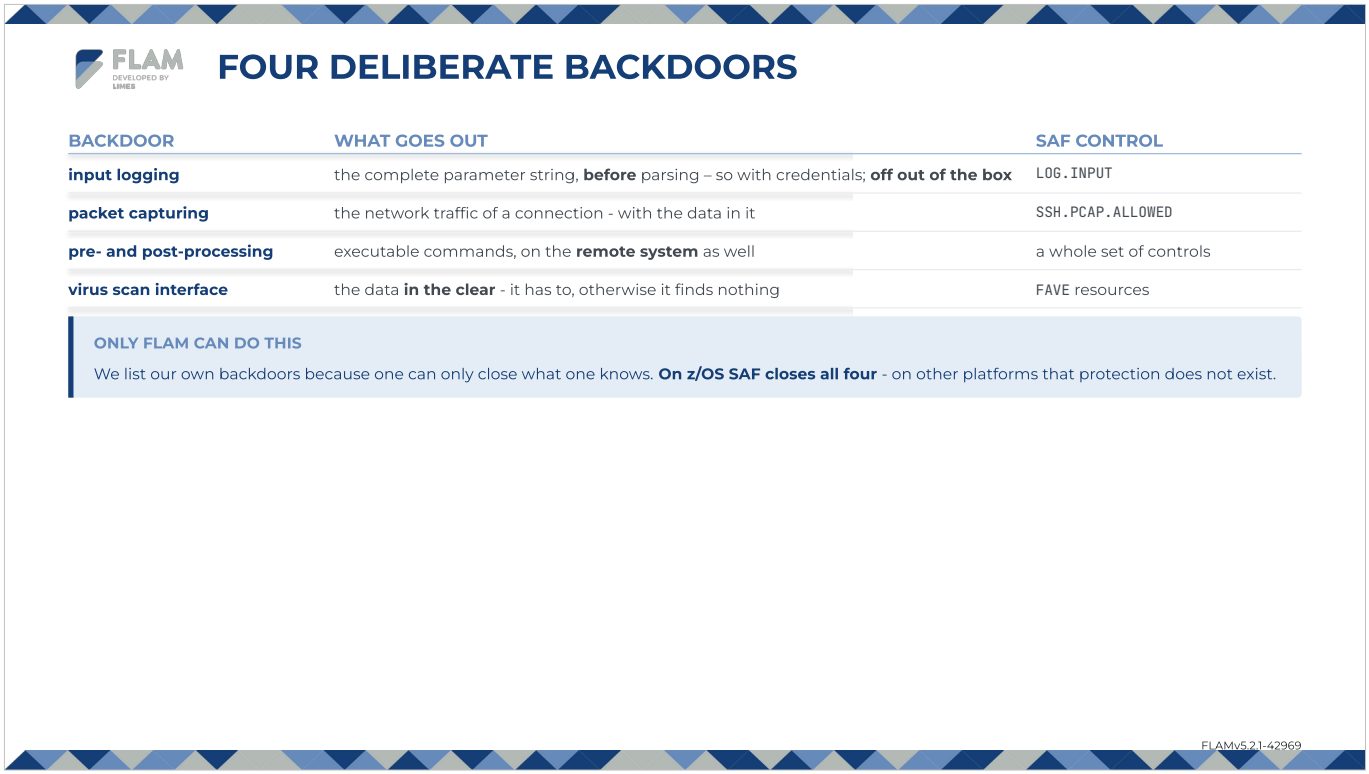
The slide that draws the line of responsibility, and it should not be skipped: it prevents the misunderstanding "we have FLAM, so we are secure". On the left, what FLAM brings: encryption over OpenPGP and SSH, hardware modules over PKCS#11 and CCA or ICSF for key exchange and signing, strict access control over SAF, checksums, virus scan. On the right, what the operator has to achieve -- and that is the longer column: manage and change the keys, check the environment variables, close the backdoors, protect the error trace, ensure the coherence of the installation, and read the warnings. The sentence that should stick: **FLAM provides tools, not states.** A file is not encrypted because FLAM is installed but because somebody ordered it and checked it.



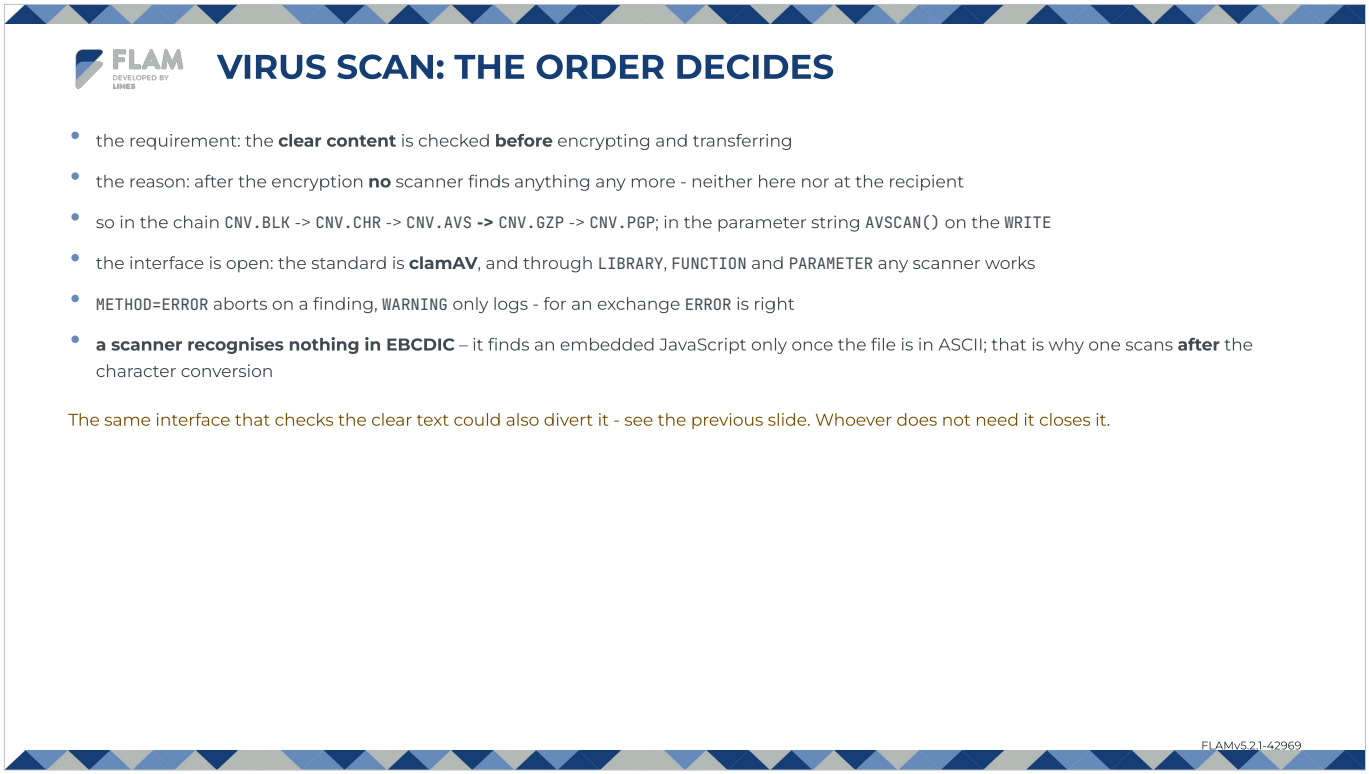
An operational slide that costs two minutes and rules out a whole class of errors. The fact of the matter: FLAM consists of many components, and some are linked dynamically. After an incomplete installation or half an update, components of different states can come together. That shows up as errors which cannot be assigned to any cause. `flcl version` writes a version for **every** component. And the decisive number is not the version number but the **build number** -- the last number after the hyphen. In a coherent installation it is **the same everywhere**. In the example below it is: 41973 in every line. The version numbers, by contrast, may differ -- a component that has not changed since 5.2.0 still carries 5.2.0. That is normal and not a finding. The recommendation from our manual, and it is concrete: **set up an installation check** that calls `VERSION` and verifies that all build numbers are the same. That is one line in a script and belongs after every installation and every update. It fits with this:
we sign modules and installation packages where the platform allows it. Where it does not, we supply secure checksums so that the downloaded package can be checked.



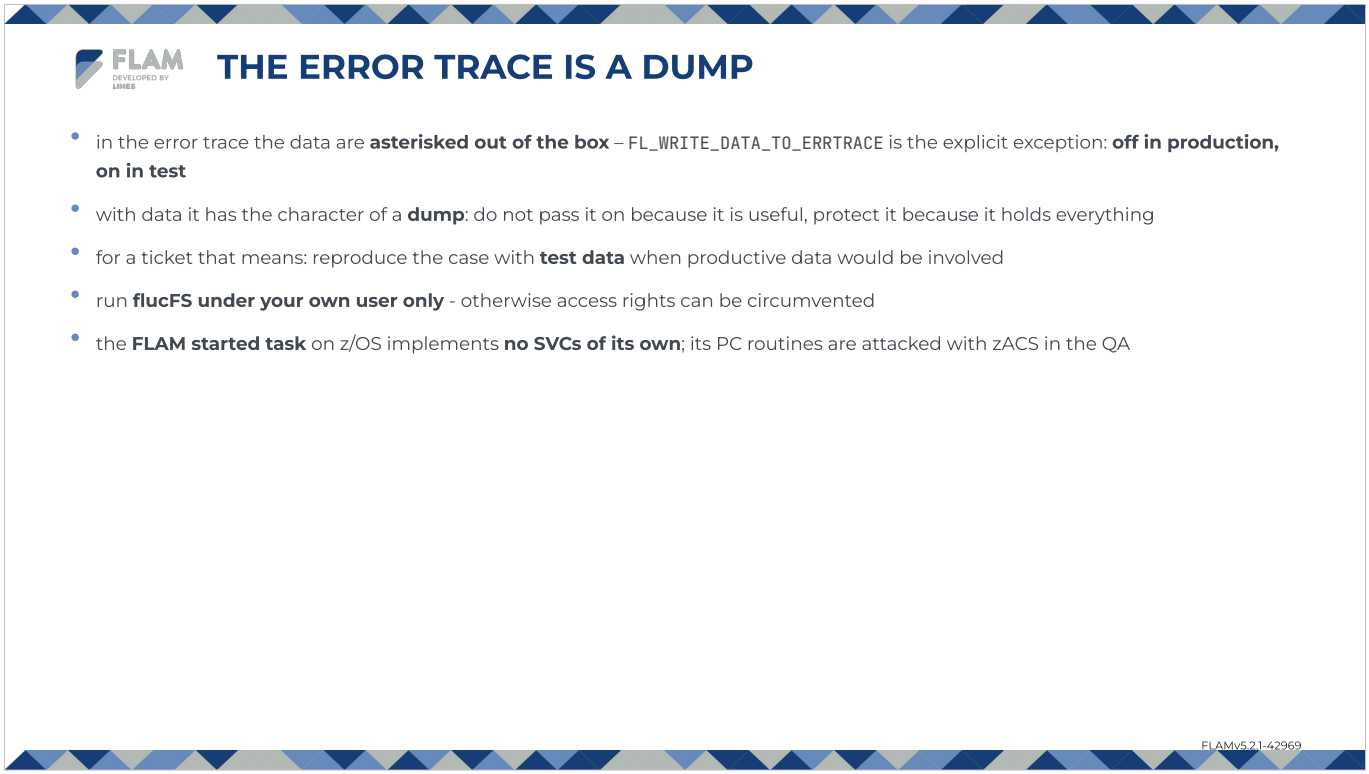
The slide on the most frequent mistake of all. First, what FLAM itself does: passwords, PINs and keys that come in through the command string are **marked as protected** and not written out in the clear -- the output says ` **SECRET** `. The parsing recognises them, and after that FLAM knows which values are critical. And now the limitation one has to know: **when the input logging is switched on, the complete command string is written** -- before the parsing, so with everything in it. That is exactly the backdoor from slide 6. More important still: FLAM can do nothing about the fact that a password in the command string was already everywhere **beforehand** -- in the process list, in the job log, in the history of the shell, in the change log of the JCL. Hence the four levels, from bad to good: **In the command string** -- it works, and it is the worst level. **In a parameter file** -- the file can be protected with rights, and it is not in the process list. That is the level every organisation can reach at once. **In an encrypted member** -- new, and the level reachable without extra hardware: the parameter file lies as a member in an encrypted FLAM5 archive, the command names the member only (`cred.fl5/?s3.provider`), and decryption comes from `FL_ARCHIVE_DECRYPT`. What protects the secret is now cryptography and no longer the file system -- and the archive may even live in version control. The unit on the command language shows the mechanics. **In the hardware module** -- then there is no password worth protecting. That is the recommendation from our manual, and it is unambiguous: an HSM instead of clear passwords for key stores. Two notes for z/OS that operations needs: there is **no pipe** there, and in-stream JCL is the **worst** choice, not the equivalent one -- the credential then stands in the job log and in the spool.
Take a data set that RACF protects instead, or encrypt the archive for an OpenPGP key: then the private key lies in ICSF and there is no password at all. And for Cloud Data Access:
nothing belongs in there, CDA is a credential store itself.
And the sentence that should stay in the room: the question is not whether the password is strong. The question is **how many places know it**.



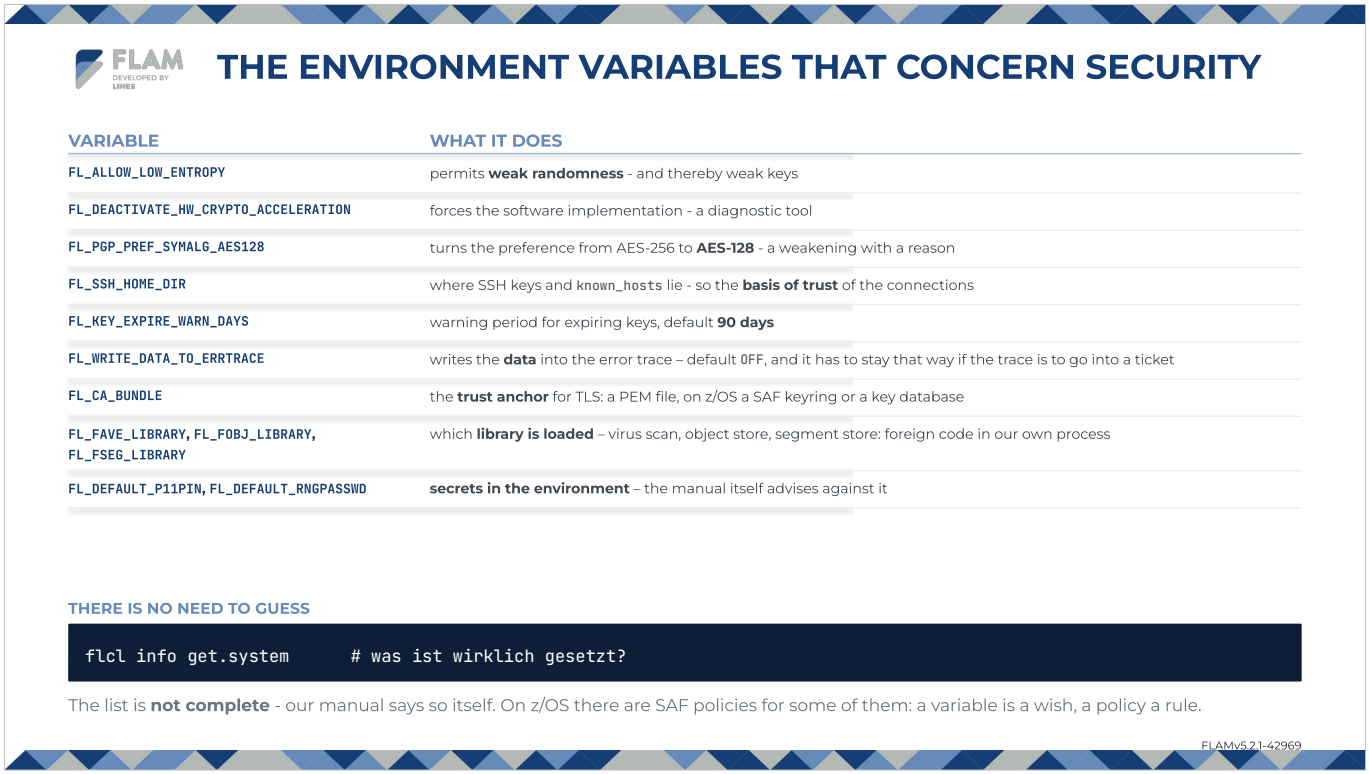
The most important slide of the unit, and it is unusual for a manufacturer: we list ourselves where clear text or executable code leaves the system. All four are useful and all four are intended. The point is not that they are bad but that they have to be **known and closed deliberately**. And the sentence that belongs in front of the list: **every one of these doors is closed out of the box.** The input logging is off, the data logging is off, and in the error trace the data are asterisked as long as an environment variable does not expressly say otherwise. The reason is not caution for its own sake but a concrete case: **so that a customer does not accidentally send us sensitive data into the support.** A parameter string does now and then hold a passphrase or a PIN, and the trace of a character set error holds the bytes before and after it. **The input logging** writes the complete parameter string into the log -- before the parsing, so with passwords. It is off by default and meant for fault finding only. The SAF policy `LOG.INPUT` prevents anybody from switching it on. **The packet capturing** records network traffic. Indispensable when an SSH connection does not work -- and the packets contain the data. Policy: `SSH.PCAP.ALLOWED`. **The pre- and post-processing** runs commands, and on the **remote system** as well. It is built for process automation and it is powerful. It is thereby also the most direct way of running foreign code. There is a complete set of SAF controls for it.
The virus scan interface gets the data **in the clear** -- it has to, otherwise it could find nothing. It is thereby at the same time a way of diverting clear text. Our manual says so word for word: this interface is at the same time a backdoor and should be closed for all other cases. The sentence for the room: **on z/OS all four can be closed over SAF.** On other platforms that protection does not exist -- there one has to work through file rights and configuration.
That is an honest answer and it belongs here.



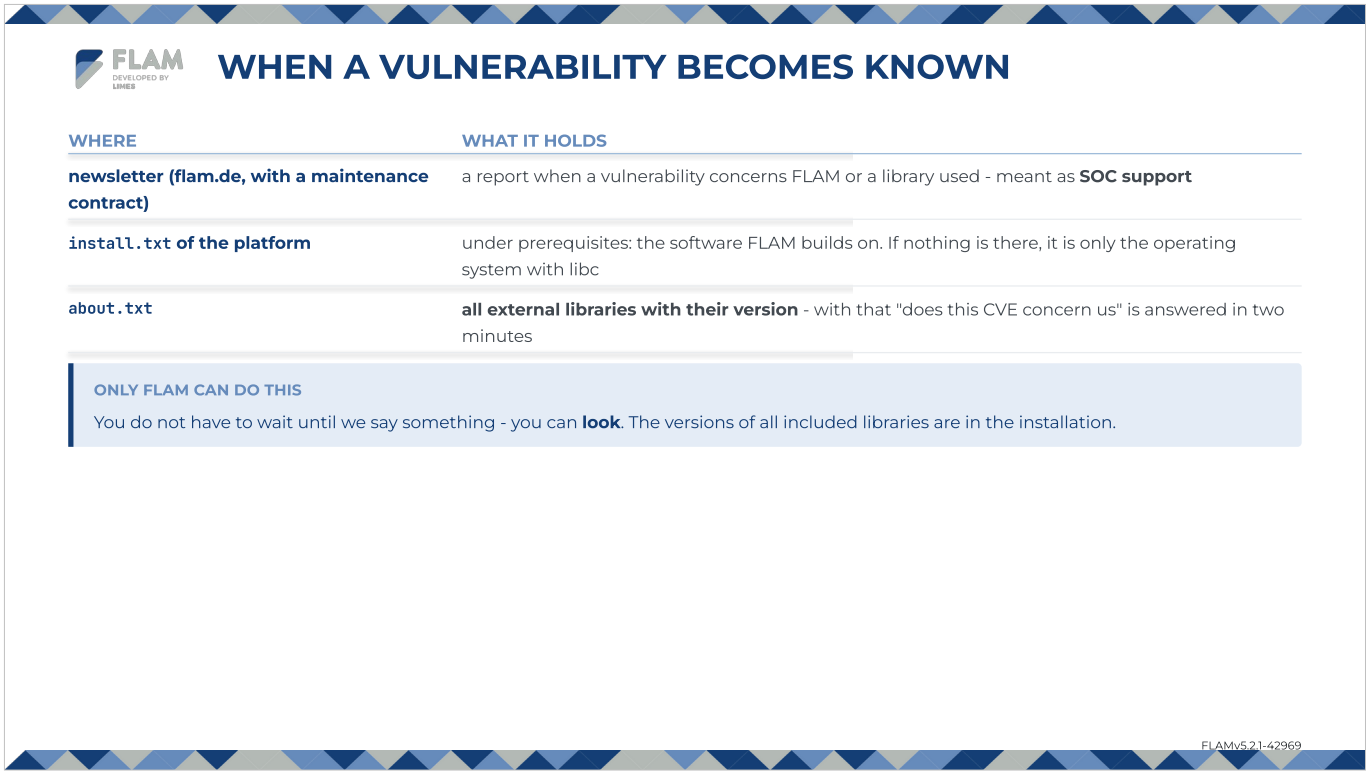
A slide of its own, because the order is the only point that counts here -- and because it is readily got wrong. The requirement several security standards make: the **clear content** of a file has to be checked **before** it is encrypted and written to a remote system. The reason is simple and is overlooked all the same: after the encryption **no** scanner can find anything any more. Neither here nor at the recipient. Whoever encrypts first and scans afterwards has a scanner that checks random bytes. That is why the scan sits in the conversion chain **before** the encryption -- in the unit *Batch embedding* that is the chain `CNV.BLK`, `CNV.CHR`, `**CNV.AVS**`, then `CNV.GZP` and `CNV.PGP`. In the parameter string it is `AVSCAN()` on the `WRITE`. The interface is open: our standard implementation uses clamAV, but any scanner can be connected -- through `LIBRARY`, `FUNCTION` and `PARAMETER`.
`METHOD` decides what a finding means: `WARNING` logs and carries on, `ERROR` aborts. For a data exchange `ERROR` is right -- an infected file should not arrive at the partner.
The second reason for the order is the one nobody expects, and it belongs said: **a virus scanner recognises nothing in EBCDIC.** An embedded JavaScript is recognised only once the file is in ASCII -- the same file in EBCDIC passes through. So the scan has to run **after** the character conversion, and that also means: clear ASCII has to go over the line for the check to take effect at all. And the honest placing to give when somebody asks about the value: **virus scanning makes nothing better.** Only what is known already is recognised, and having to build a backdoor into an encryption product for it was a farce. What we made of it: the interface is open so that there is a solution in the open world as well -- and one can **switch FLAC/FAVE off** and close this little back door.
The note from slide 6 belongs here: the same interface that checks the clear text could also divert it. Whoever does not need it should close it over SAF.



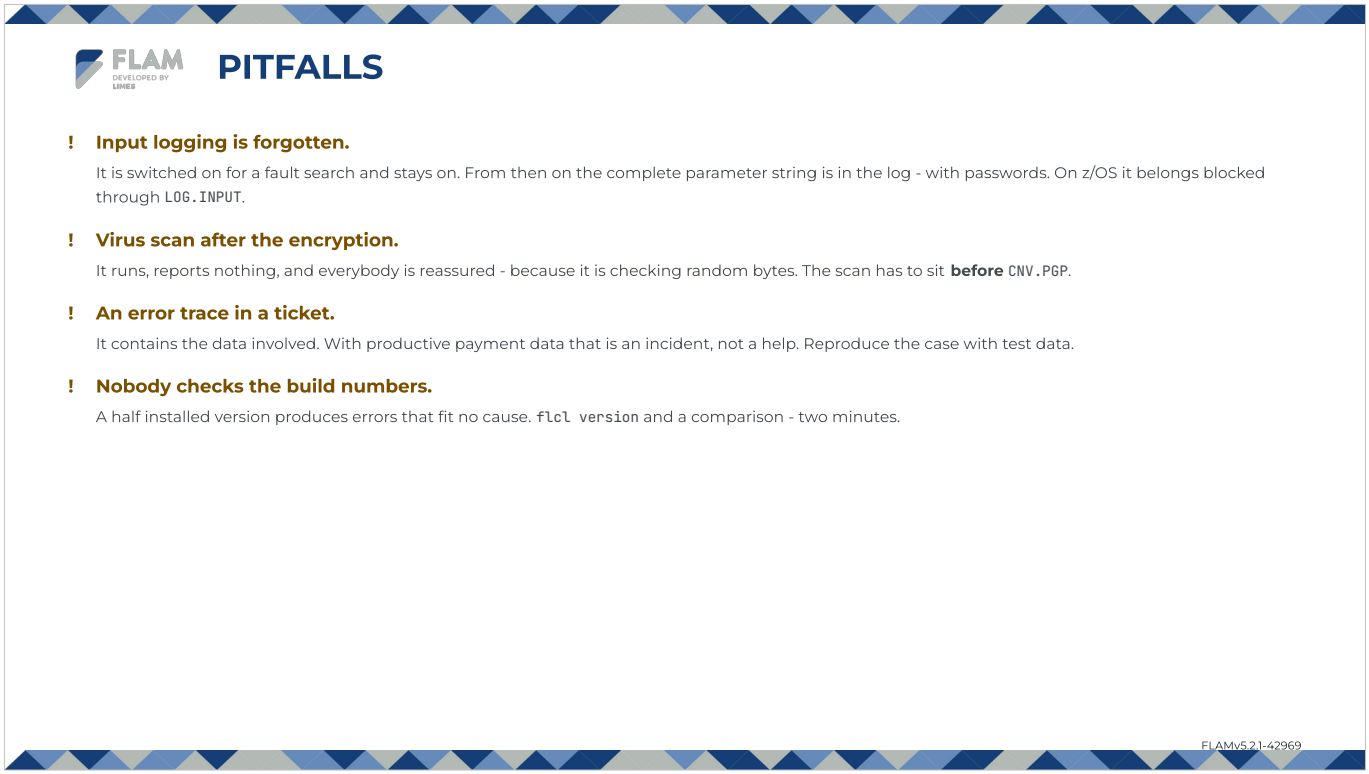
A short slide with a statement one has to hear only once and then never forgets. The error trace is one of the most useful tools of FLAM. For a usable root cause analysis it **could** record the data involved -- and that is why it does **not** do so out of the box:
the data are **asterisked** as long as `FL_WRITE_DATA_TO_ERRTRACE` does not expressly say otherwise. The help of that variable says it itself: "must be OFF if error trace is to be part of the support ticket". In our own test environment the FTINIT switches it **on**, because there one wants to see the bytes at the point of failure.
The rule for a house is thereby simple: **off in production, on in test.** And the practical consequence for a ticket: either the variable was off -- then the trace can be sent -- or the case is reproduced with test data. With data it has the same character as a dump. A dump is not passed on because it is handy either, it is protected because it contains everything. The practical consequence for a ticket: a trace may come to us -- but somebody has to have decided that it may. With productive payment data the answer is usually: reproduce the case with test data. The rule about what belongs in a ticket, from the unit *Documentation and support*, fits with this. And from our own QA talk the sentence that sums it up: whoever sends sensitive data has compromised it at the moment of sending. Two neighbours belong on the same slide because they follow the same logic: **flucFS should run under your own user only** -- otherwise rights can be circumvented through the file system. And the **FLAM started task** on z/OS is an active component in the background; it implements no supervisor calls of its own, and its PC routines are attacked with zACS in our QA. The only finding there are regular terminations with 0C4.



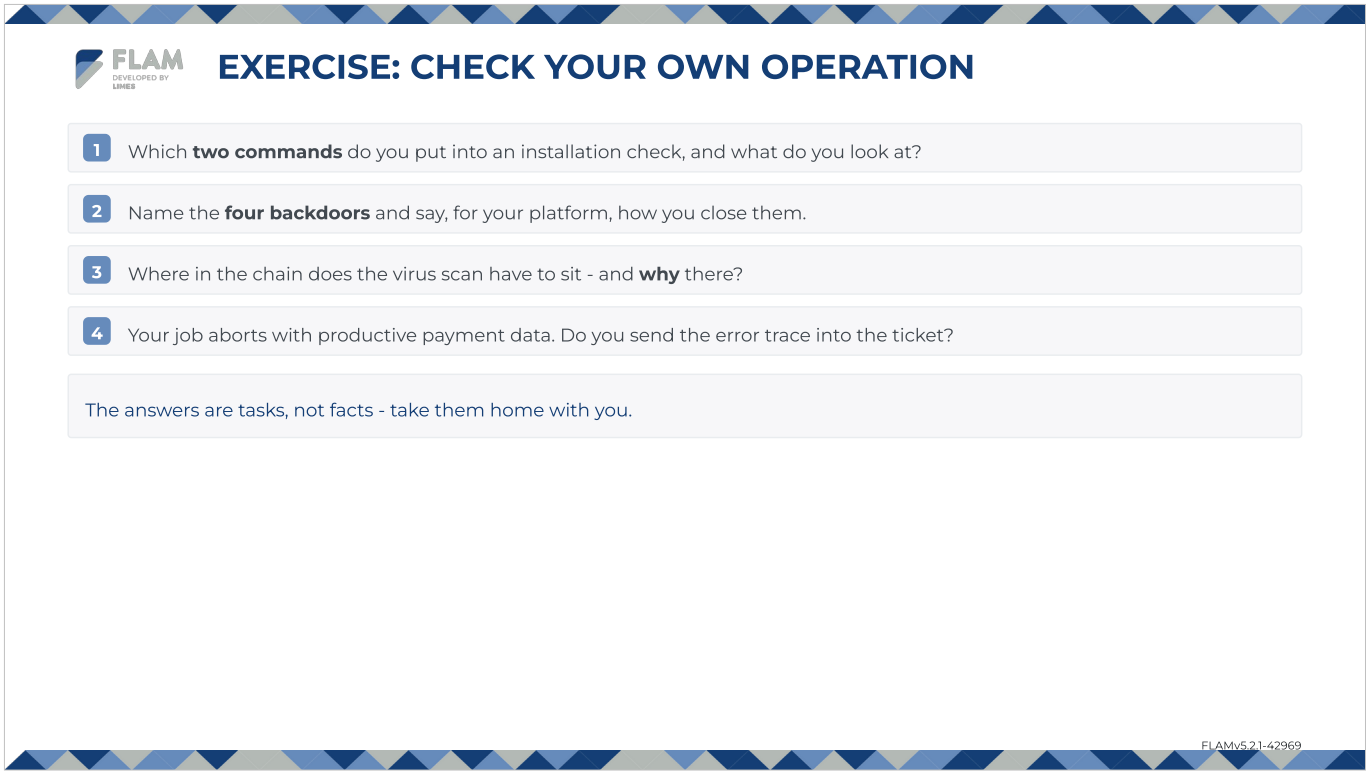
The reference slide, and the list is expressly **not complete** -- our manual says so itself. It holds the ones you should know. `FL_ALLOW_LOW_ENTROPY` allows weak randomness. Convenient when a job aborts -- and thereby a way to weak keys.
`FL_DEACTIVATE_HW_CRYPTO_ACCELERATION` switches the hardware acceleration off and forces software. A diagnostic tool.
`FL_PGP_PREF_SYMALG_AES128` turns the order of the preferred symmetric algorithms round -- AES-128 first instead of AES-256.
There are reasons for it (a partner who cannot do 256), but it is a weakening. `FL_SSH_HOME_DIR` determines where the SSH keys and the `known_hosts` lie. Whoever redirects that redirects the basis of trust of the connections. `FL_KEY_EXPIRE_WARN_DAYS` is the warning period for expiring keys, 90 days by default. Four belong with them which we added while going through `asFl5EnvVarTab` (Falk, review 2026-09-18):
`FL_WRITE_DATA_TO_ERRTRACE` is the variable from the previous slide -- off in production, on in test. `FL_CA_BUNDLE` determines **whom we trust with TLS**: a PEM file, on z/OS a SAF keyring or a key database. Whoever redirects that redirects the basis of trust -- like `FL_SSH_HOME_DIR`, only for HTTPS.
`FL_FAVE_LIBRARY`, `FL_FOBJ_LIBRARY` and `FL_FSEG_LIBRARY` say **which library is loaded**. That is foreign code in our own process, and thereby the most effective variable of this list.
`FL_DEFAULT_P11PIN` and `FL_DEFAULT_RNGPASSWD` put secrets into the environment. The manual advises against it itself, and the environment of a job is not a store for secrets. The complete list is to be had without searching, by the way: `flcl HLPENV` names every variable the product takes notice of, with its meaning. And the practical part: **you need not guess what is set.** `flcl info get.system` shows the current environment.
That belongs in the installation check of slide 4 -- not only "are the build numbers the same" but also "is something set that should not be set". On z/OS there are SAF policies for some of them. That is the difference the next unit works out:
an environment variable is a wish of the user, a policy is a rule of the house.



The slide for the security operations centre, and it answers a question every regulated organisation asks: **how do we learn that it concerns us?** First: there is a **newsletter about security vulnerabilities**. With a maintenance contract one can register for it on flam.de. It is sent when a vulnerability concerns FLAM or one of the libraries used. That is expressly meant as SOC support. Second, and this is the part that counts in an emergency: **one can establish oneself whether a report is relevant.** Two places: in the `install.txt` of the platform, under prerequisites, is the software FLAM builds on. If nothing is there, only the operating system with the language environment and the corresponding libc is concerned. In the `about.txt` are **all external libraries with their versions** that FLAM includes. With that the question "is this CVE relevant for us" can be answered in two minutes -- without asking us. The sentence for the conversation: you do not have to wait until we say something. You can look. The full treatment -- the two CVE lists with a justification per entry -- is in the unit *Operation and maintenance*. All that belongs here is where to look.



Four, all from practice. On the first: the input logging is switched on for a fault search and then forgotten. From then on every password is in the log. On the second: the virus scan after the encryption. It runs, it reports nothing, and everybody is reassured -- because it is checking random bytes. On the third: a trace in a ticket. Well meant, and with productive data an incident. On the fourth: the installation check nobody does. Inconsistent build numbers produce errors that are then looked for in the wrong place.

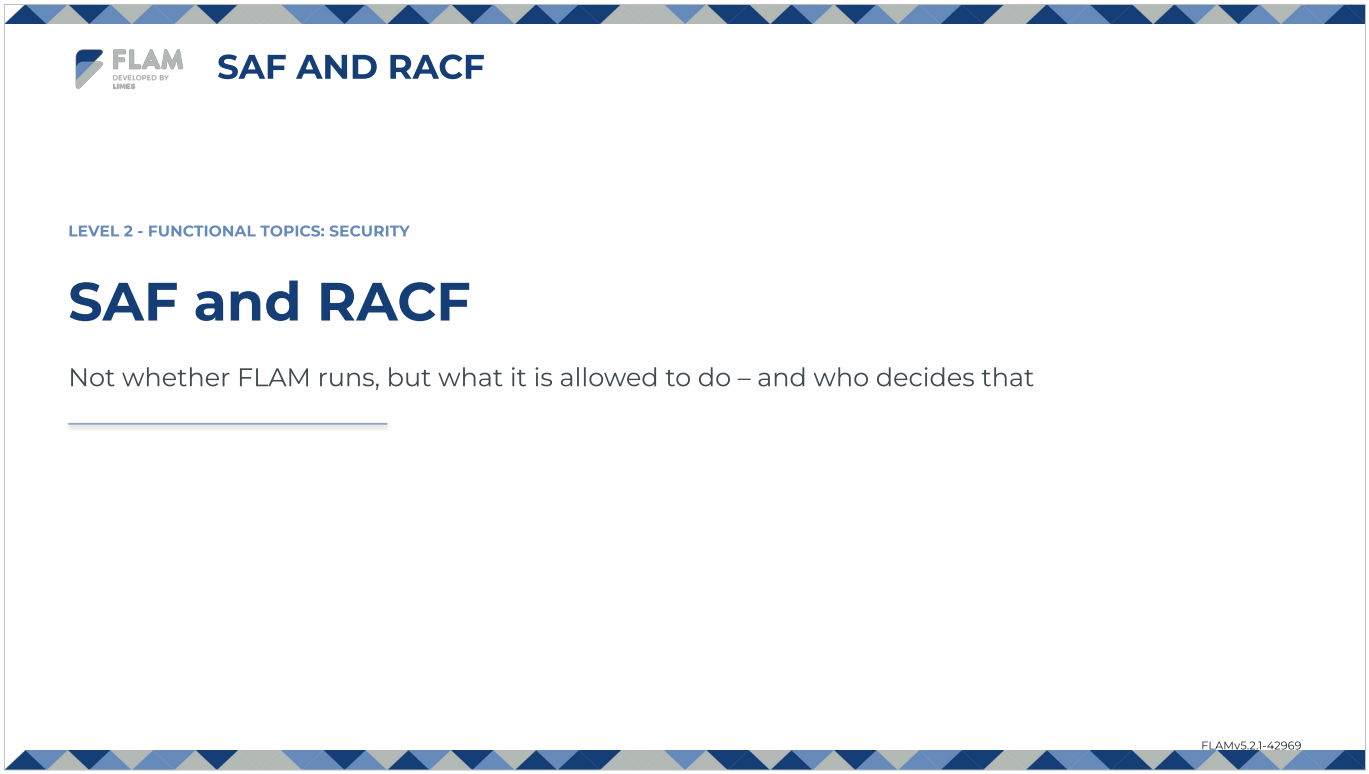


Fifteen minutes, and this exercise is deliberately aimed at the participants' own organisation -- the answers are tasks they take home. The solutions: **Task 1.** `flcl version`, compare all build numbers; plus `flcl info get.system` and check whether any of the variables from slide 9 is set. Both into a script that runs after every installation. **Task 2.** The four: input logging, packet capturing, pre- and post-processing, virus scan interface. Close them over SAF on z/OS, over rights and configuration on other platforms.
Task 3. Before the encryption -- because afterwards nobody finds anything any more. In practice: `AVSCAN()` on the `WRITE`, `METHOD=ERROR` for an exchange. **Task 4.** No.
The trace contains the data involved. Either reproduce it with test data, or somebody with the authority decides and the trace is transferred under protection.

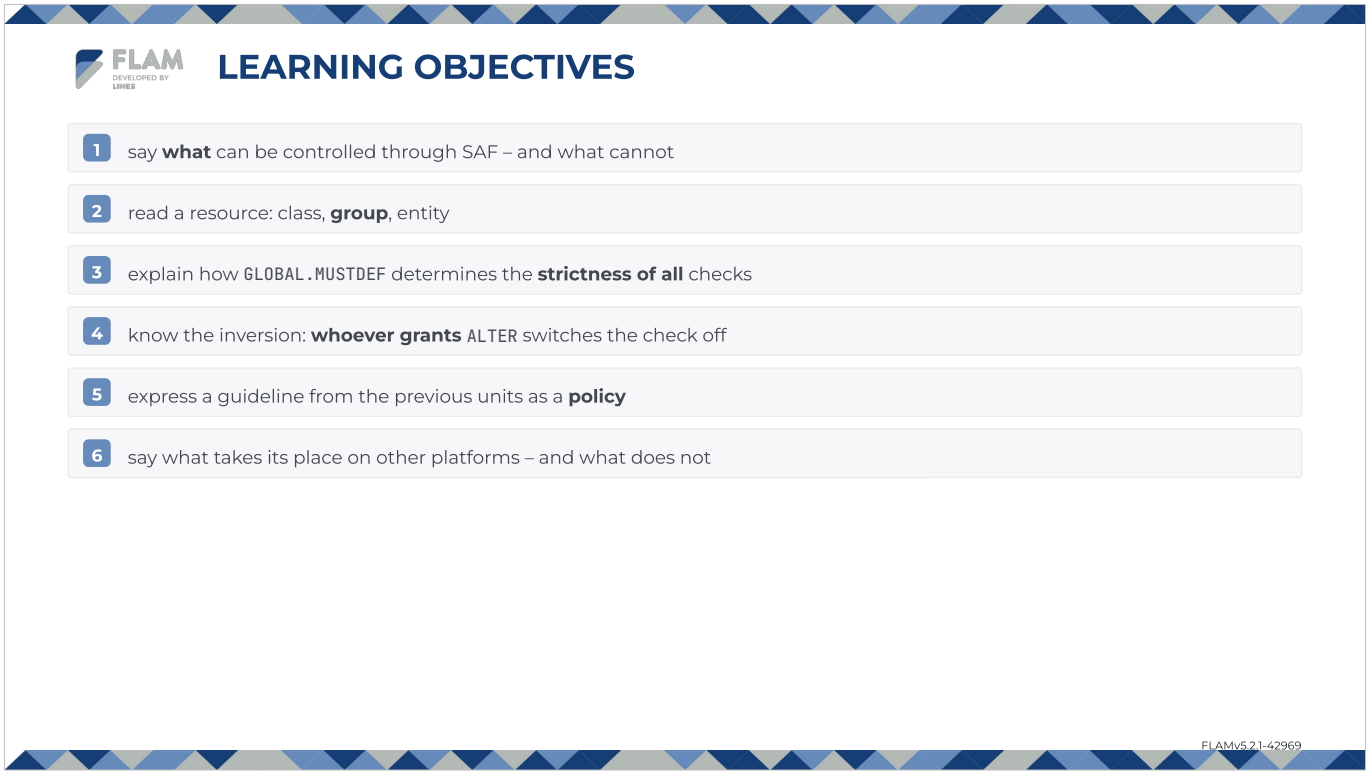
WHAT TO TAKE AWAY

- **FLAM provides tools, not states** - the order and the checking are done by operations
- **four deliberate backdoors**: input logging, packet capturing, pre-/post-processing, virus scan - closable over SAF on z/OS
- the **virus scan has to go before the encryption**, afterwards no scanner finds anything any more
- credentials in three levels: command string **bad**, parameter file reachable at once, **HSM recommended**
- the **error trace is a dump** - reproduce with test data instead of passing it on
- **identical build numbers** in `flcl version` are the shortest installation check

Go through the learning objectives. The closing sentence is the one from slide 3: FLAM provides tools, not states.



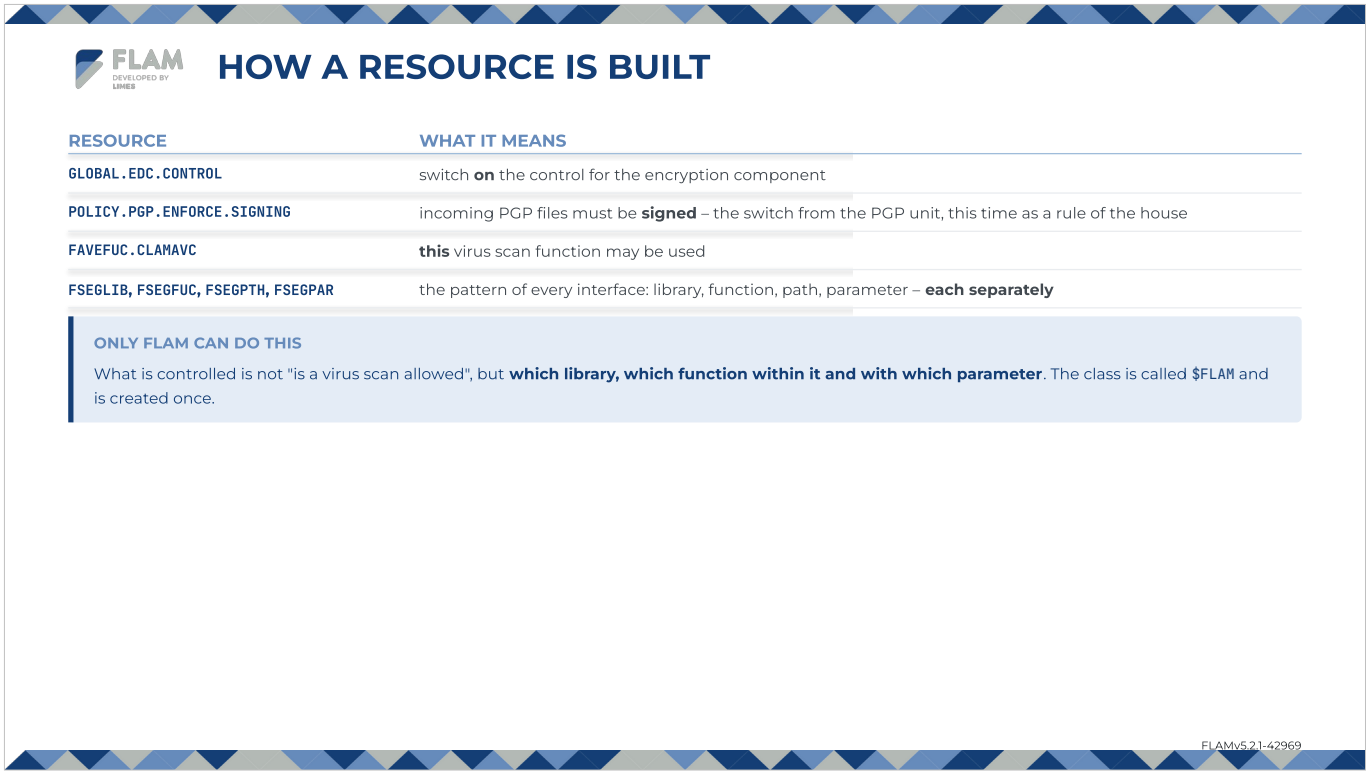
Last unit of the security group, and the only one tied to a platform. It addresses the **security officer**, not the user -- that should be said at the start so the developers in the room know why they are listening: they learn what can be forbidden to them. The frame in one sentence: since version 5.1.21 a RACF administrator on z/OS can activate security policies and control a large part of what FLAM does. **Not whether FLAM runs, but what it is allowed to do.** And the sentence that holds the unit together -- it already came up in the previous units: an environment variable is a **wish of the user**, a policy is a **rule of the house**. Here we learn how to make rules.



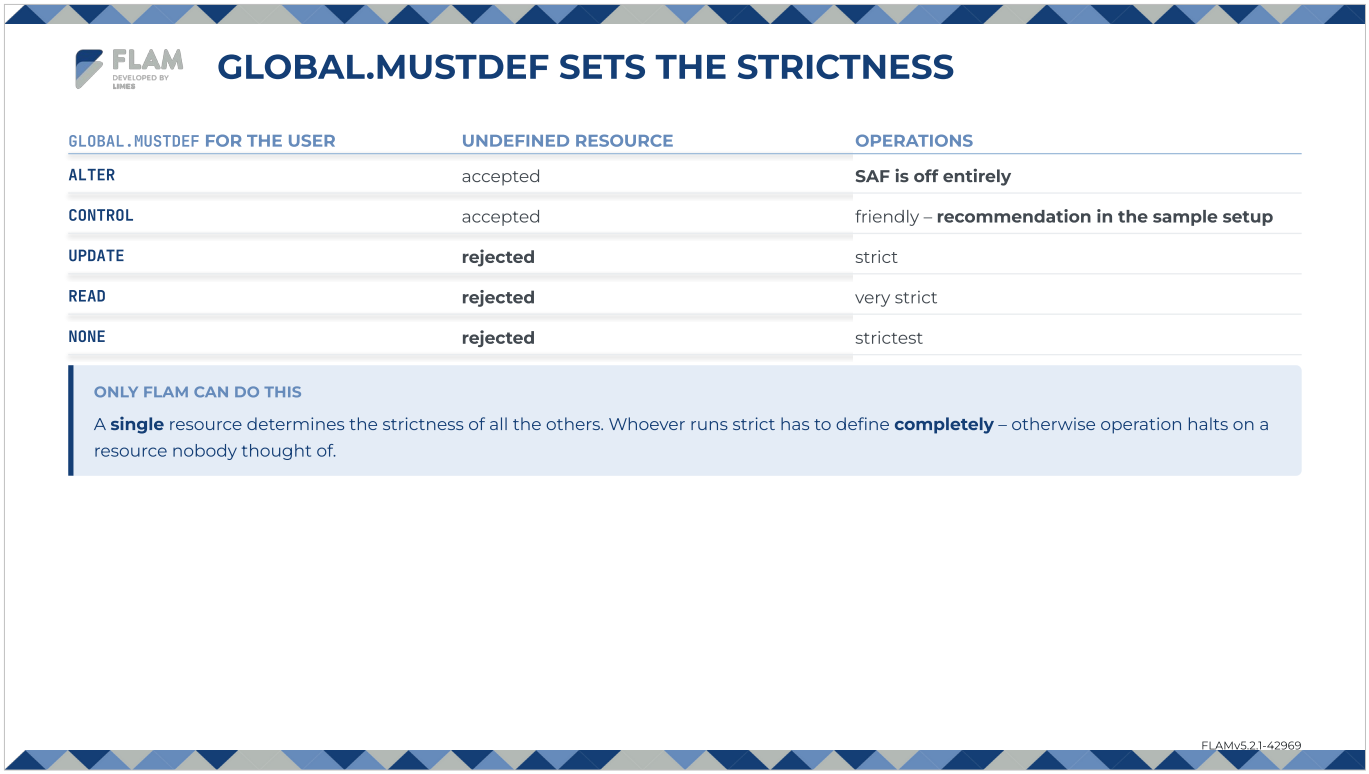
The learning objectives. The third is the reason the unit exists: the mechanics around `GLOBAL.MUSTDEF` cannot be guessed, and whoever gets them wrong sets up a protection that does nothing. The fourth is the sentence that saves a RACF administrator work -- and the mistake they would otherwise make.



The inventory, and it is longer than most expect. Do not read it out -- pick two or three points that matter in the room. The list from our manual, grouped:
Policies: switch PCAP off, enforce fingerprints, minimum lengths for keys and passwords. **The parameter strings themselves** -- that is the most far-reaching point. Through `CLPAR` it can be controlled which parts of the command language a user may use at all. That makes the functionality itself controllable, not just its result. **Pre- and post- processing**: which commands may be executed, locally and remotely. **All interfaces**: FAVE (virus scan), FCPE (pre-/post-processing), FKME (keys), FSEG and FOBJ (storage). Including the question **which virus scanner** the clear data may be handed to. **Complete URLs** -- connection, path, file name and member. With that even single members of a PDS/E, a ZIP or a FLAM archive can be protected.
Connections: SSH with user, server and port; HTTP/REST at method level and protection against TLS bypass. **PGP key management**:
which user identities may be imported, exported, generated or deleted -- and which user and key IDs may be used for signing and for key exchange. **Additional SMF logging**, for instance record type 119 for SSH.
All feature codes of the licence module. The sentence for the room: this is not access control on files -- z/OS already has that. This is control over the **functions of a program**, and that is unusual.



The reading aid, short and necessary, because everything that follows builds on it. There is a **resource class of its own**, `$FLAM`. It is created once through `RDEFINE CDT`; that is in the sample setup in the manual. Within it every resource has the form **group.entity**. The group says what it is about, the entity says which concrete thing. Three examples in which the pattern becomes visible: `GLOBAL.EDC.CONTROL` -- group `GLOBAL`, and the entity says: switch on the control for the encryption component.
`POLICY.PGP.ENFORCE.SIGNING` -- group `POLICY`, and the entity is the policy: incoming PGP files must be signed. That is exactly the switch from the PGP unit, only this time as a rule of the house.
`FAVEFUC.CLAMAVC` -- group `FAVEFUC`, entity `CLAMAVC`: this virus scan function may be used.
The interfaces have three groups each, and the pattern is the same for all of them: `...LIB` for the library, `...FUC` for the function, `...PAR` for the parameter.
For FSEG and FOBJ `...PTH` for the path is added. So what is controlled is not "is a virus scan allowed", but **which library, which function within it and with which parameter**.



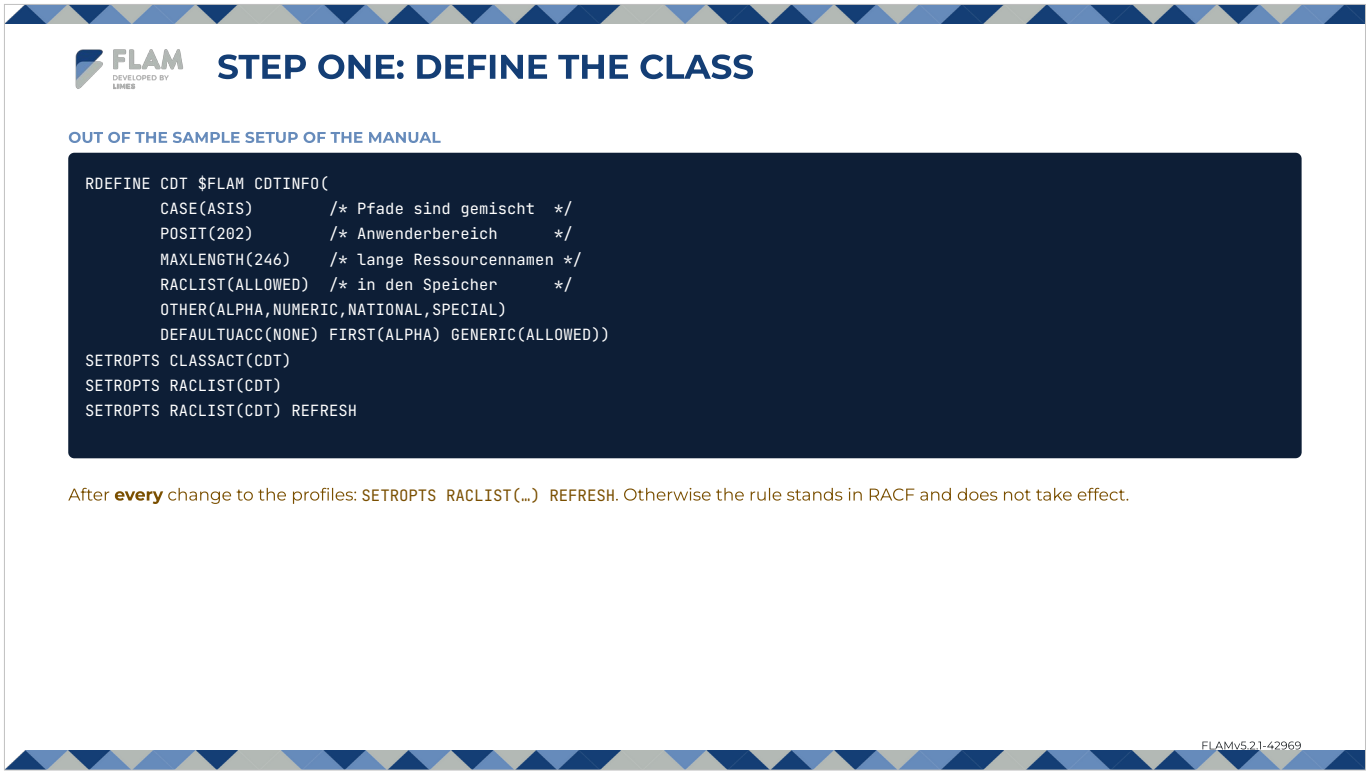
The central slide of the unit. Whoever understands it understands the whole procedure behind it; whoever skips it sets up a protection that does not do what they believe it does. The starting point is a question every authorisation system has to answer: **what happens to a resource that is not defined?** RACF gives three answers for that: 0 means authorised, 4 means **resource not defined**, 8 means not authorised. The 4 is the interesting case, because in a large setup something is always undefined. FLAM does not decide that itself, it lets itself be told -- through **one** resource:
`GLOBAL.MUSTDEF`. And the way it is done is elegant: the **access class** given to the FLAM user on it determines the threshold for **all** further checks. The table reads like this: with `CONTROL` -- that is the recommendation in the sample setup -- undefined resources are accepted. That is the friendly operation: you define what you want to control, and everything else runs. With `UPDATE`, `READ` or `NONE` undefined resources are **rejected**. That is the strict operation: what is not explicitly allowed is forbidden.
Whoever switches that on has to define completely -- otherwise operation comes to a halt. And `ALTER` switches SAF **off entirely**. A technical detail for those who want to know exactly: below `CONTROL` FLAM additionally sets a strong flag that is passed to the storage interfaces -- so a provider learns that it is running in a strictly controlled operation.



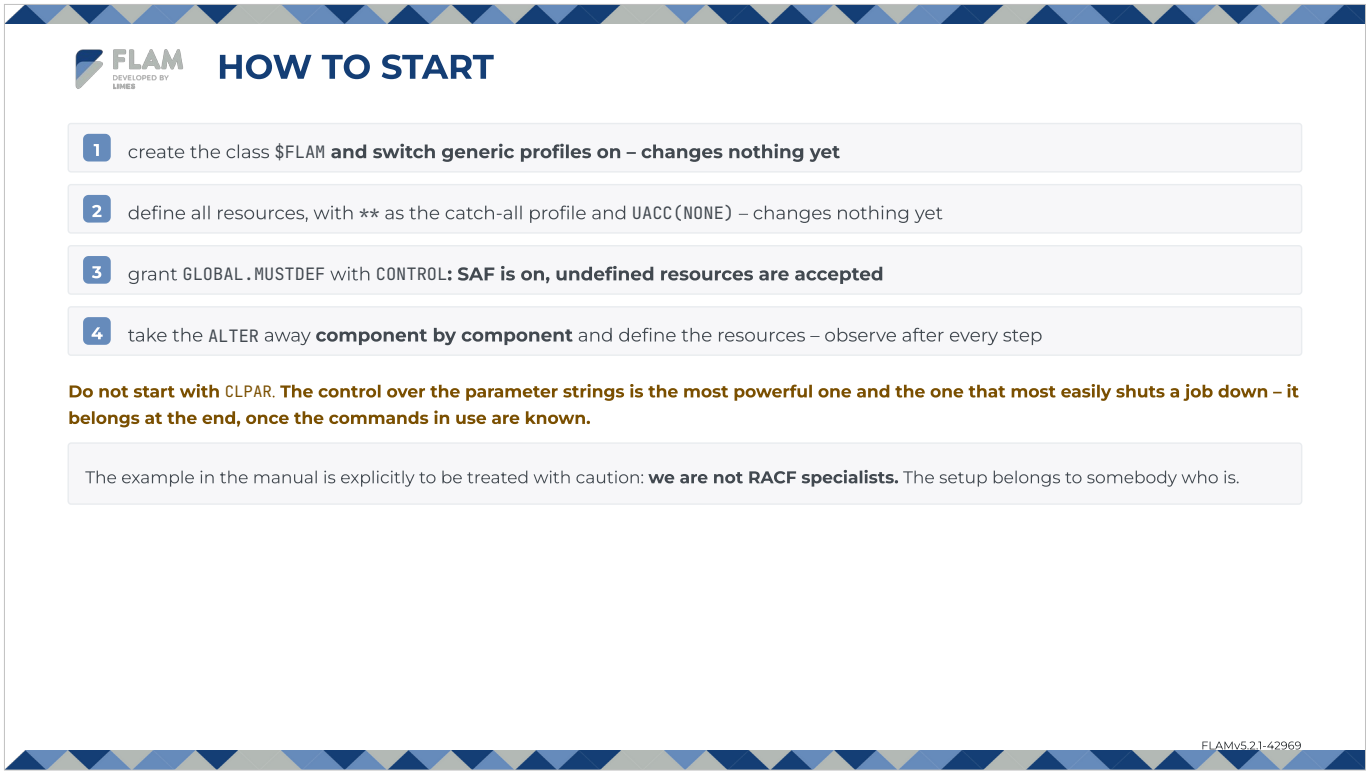
This slide is the most important of the unit, and it contradicts the gut feeling of every RACF administrator. The expectation:
more authorisation means more possibilities for the user. So `ALTER` is granted generously, in order not to hinder anybody.
With the `GLOBAL..*CONTROL*` *resources it is the* **other way round**: the control for a component is **active** exactly when the user does **not** have `ALTER`. Whoever grants `ALTER` is saying: for this user this component is not checked. That is also how it stands in the sample setup of our manual, and it can be seen there: behind the `PERMIT` lines with `ACC(ALTER)` there is the comment `*/INACTIV/*`, and the commented-out lines are marked `ACTIVE`. The file says it itself. Why it is built that way: so that a house can switch the control on **one at a time** without having to define everything. In the initial state everybody has `ALTER`, nothing is checked, FLAM runs as it does without SAF. Then component by component the `ALTER` is taken away and the resources are defined instead. The sentence for the room, and it should be said slowly: **a generous authorisation switches the protection off, it does not extend it.** Whoever checks after the setup whether things still run and grants `ALTER` at the first problem has removed the protection again -- and nobody notices, because nothing fails.



Now the benefit, and this slide is the answer to "what do we get out of it". The selection is chosen so that every line ties back to one of the previous units.
`POLICY.PGP.ENFORCE.ENCRYPTION`, `...INTEGRITY` and `...SIGNING` are exactly the three switches from the PGP slide -- only not as the wish of a job, but as a rule of the house. **That is the difference this group is about**: a job can set them, but it can also leave them out. A policy it cannot leave out. `POLICY.SYM.KEY.LEN.ENC` and `POLICY.SYM.PWD.LEN.ENC` enforce **minimum lengths** for keys and passwords when encrypting -- and they exist in the `.DEC` variant for decrypting too, so that old data is not accidentally read with weak keys. `POLICY.LOW.ENTROPY` and `POLICY.LOG.INPUT` are the counterparts to the environment variables from the previous unit.
`POLICY.SSH.PCAP.ALLOWED` closes packet capturing, `POLICY.SSH.WEAK.HOSTKEY.CHECK.ALLOWED` and `POLICY.SSH.CUSTOM.KNOWN.HOST.FILE.ALLOWED` prevent anybody from working around the trust basis of the SSH connections.
`POLICY.KEY.GENERATE.PGP.EXPIRD.REQIRED` demands that newly generated keys have an expiry date. And `POLICY.KEY.GENERATE.PGP.ALGO.RSA.KEYLEN` enforces a minimum length when generating. A detail on handling: with the policies the **access class carries the value**. In the sample setup `POLICY.SSH.AUTH.PUBLICKEY` stands with `READ` and `POLICY.SSH.AUTH.PASSWORD` with `UPDATE` -- the class says how far something is allowed, not only whether. And the spelling `EXPIRD.REQIRED` is the product's own. We quote it as it is.



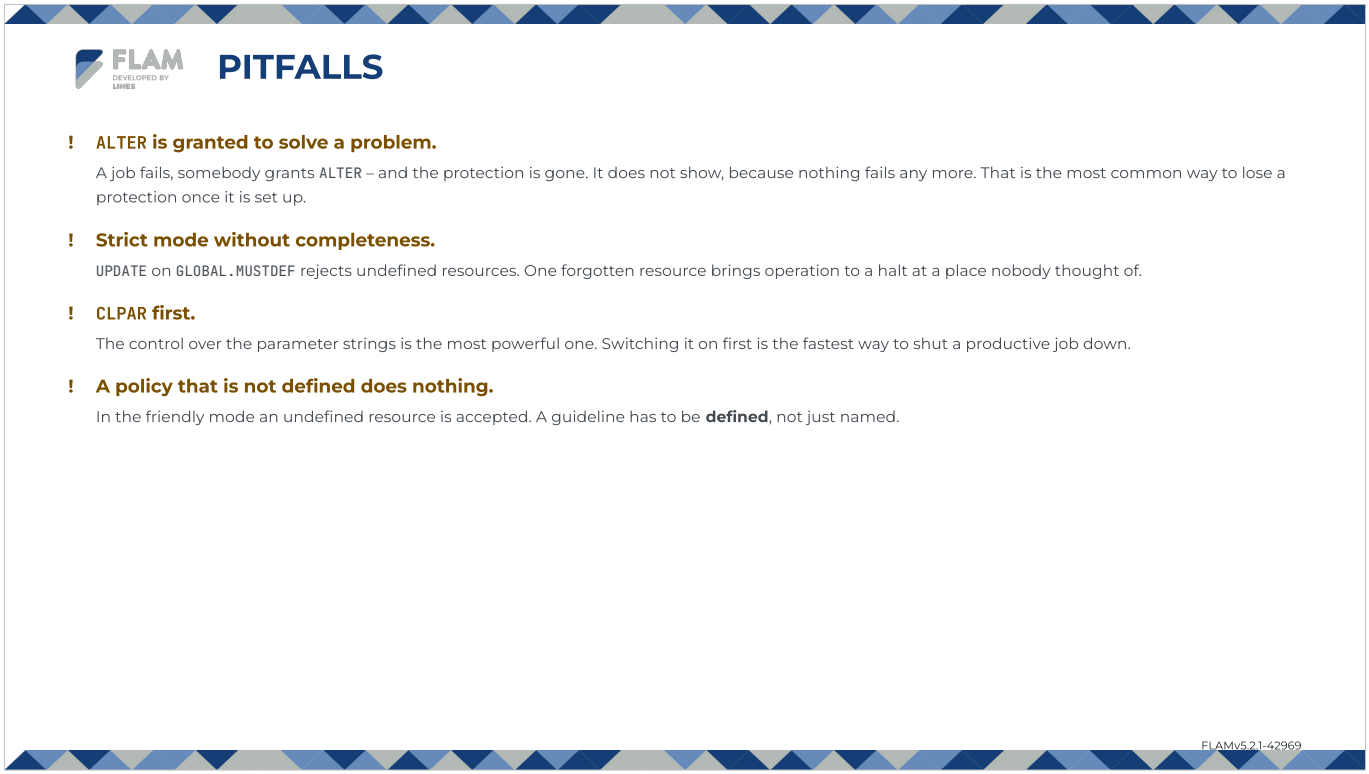
The slide with the commands the manual prescribes -- and without which nothing else takes effect. It is the answer to the question "and how does the system programmer start?". `$FLAM` is **not** an existing RACF class. It is defined in the **class descriptor table**, and four specifications decide: `**POSIT(202)**` lies in the user range 128 to 527 -- the number is free as long as it lies there and is not taken. `**MAXLENGTH(246)**`, because our resource names grow long: connection, path, file name and member stand in them. `**CASE(ASIS)**` and `OTHER(ALPHA,NUMERIC,NATIONAL,SPECIAL)`, because a path holds lower case and special characters. Without `ASIS` no USS path name fits any more. `**RACLIST(ALLOWED)**`, so that the profiles may be put into storage -- that is the prerequisite for the next two lines and for the run time.
Then `SETROPTS CLASSACT` (activate the class), `SETROPTS RACLIST` (profiles into storage) and after every change `REFRESH`. **The `REFRESH` is the line that is forgotten in operations**: the rule stands in RACF but takes effect only after the refresh. And once more the hint from the slide before: the sample comes out of our manual, and it should be done by somebody who does RACF for a living.



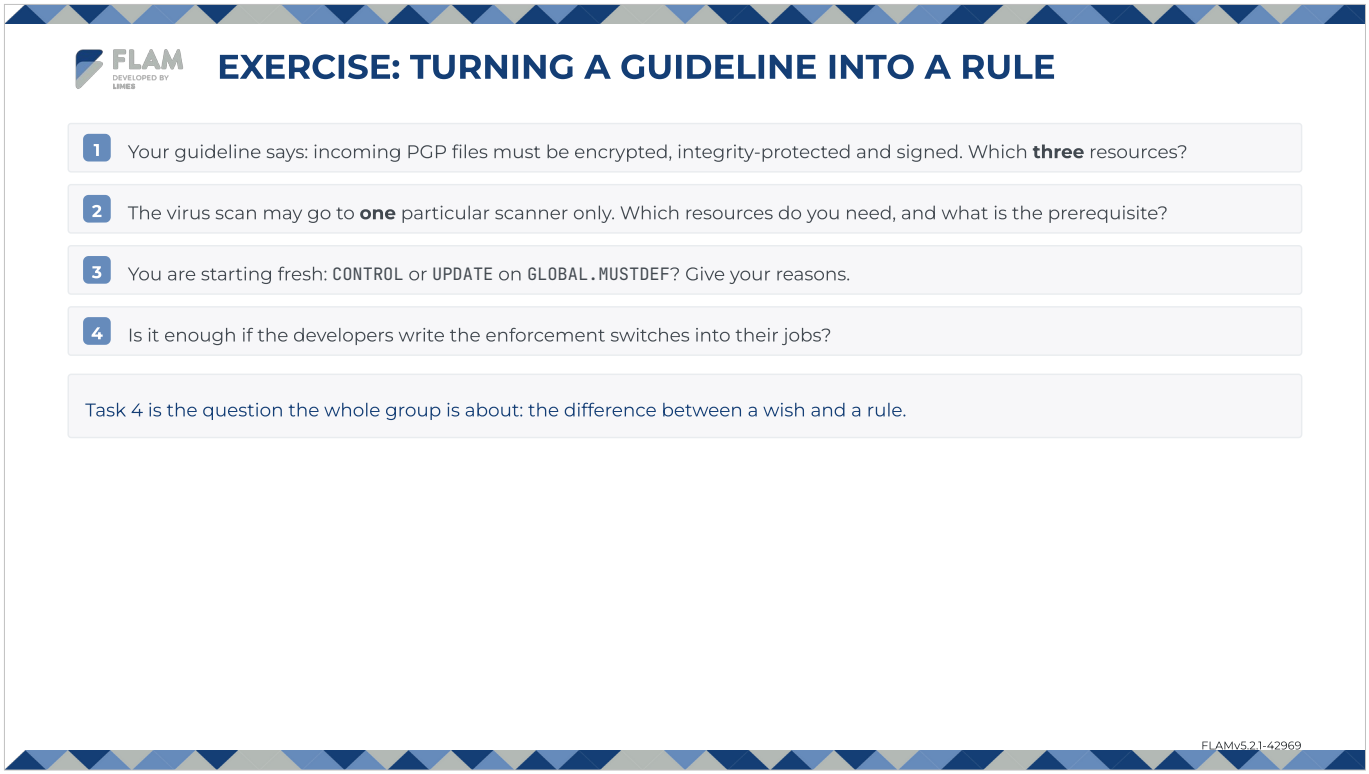
The practical slide: how does a house approach this without bringing operation to a halt. The way our manual suggests, and which has proven itself, has four steps.
First create the class `$FLAM` and switch generic profiles on. That is one-off work and changes nothing yet. **Second** define all resources -- with `` **as the catch-all profile and `UACC(NONE)`.**
That too changes nothing as long as the authorisations are generous.
Third grant `GLOBAL.MUSTDEF` with `CONTROL`. With that SAF is on, but undefined resources are accepted -- operation carries on. **Fourth**, and this is the actual work: take the `ALTER` away **component by component** and define the resources belonging to it. First the virus scan, then the connections, then the parameter strings.
Observe after every step. The advice to give with it:
do not start with `CLPAR`. The control over the parameter strings is the most powerful one and the one that most easily shuts a job down. It belongs at the end, once the commands in use are known. And a note on honesty which is in our manual too: the example there is explicitly to be treated with caution -- **we are not RACF specialists.** The setup belongs to somebody who is.



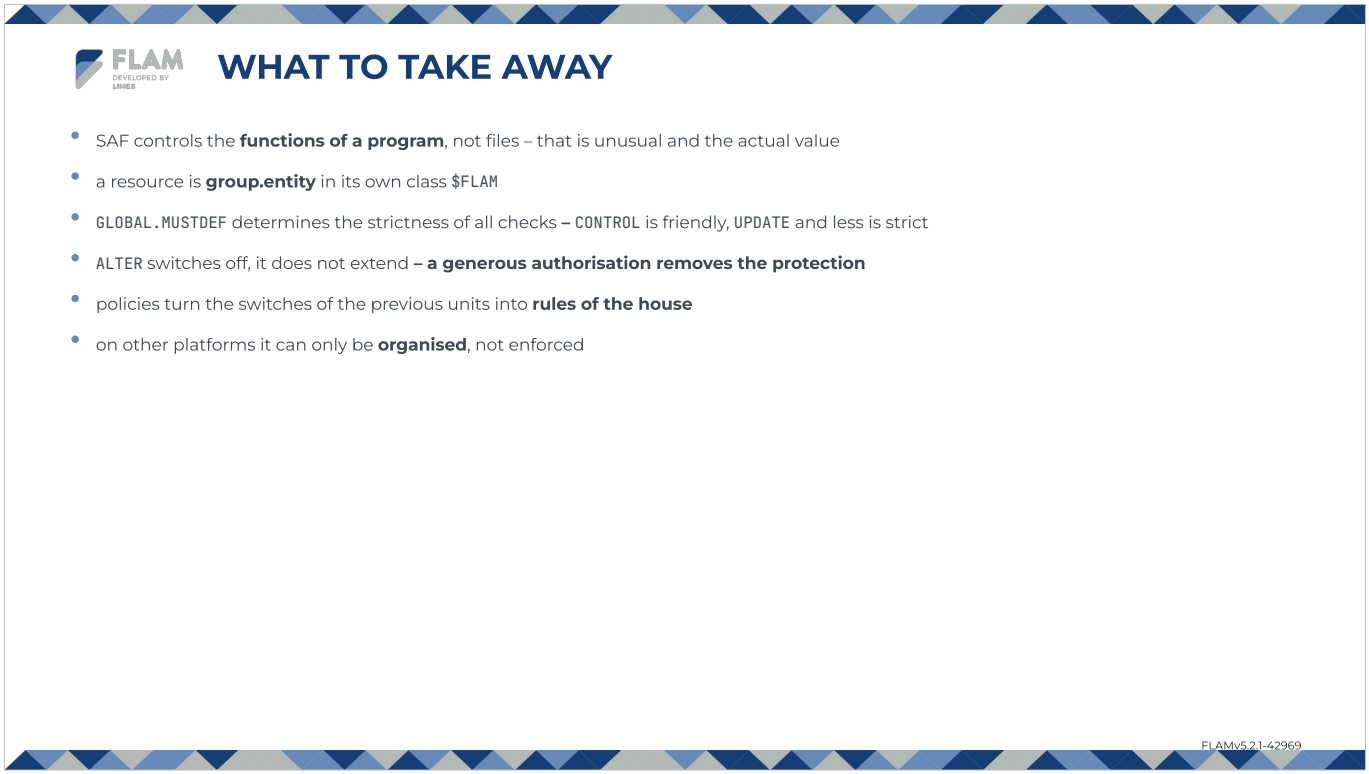
The slide that states an uncomfortable truth instead of avoiding it -- and thereby creates trust. Our manual puts it that way itself: there is **no** comparable protection of application-specific resources on other operating systems. Linux, Unix and Windows have no SAF interface into which a program could hook itself in order to have its own functions controlled. What takes its place there is weaker, and that should be said plainly: **file permissions** on the configuration, the parameter files and the key stores. Whoever may not write the parameter file cannot change the call. **Separate users** for separate tasks -- and flucFS only under its own user, as in the previous unit. **Control the environment**:
whoever can set the environment variables can switch protection off. So they belong in a controlled start environment, not in the profile of the user. **And do not install what is not needed**: the attachment to a virus scanner that nobody uses is a back door without benefit.
The sentence for the room: on z/OS the security officer can **enforce** it. On the other platforms they can only **organise** it. That is a real difference, and it is an argument for the host where payment traffic is concerned.



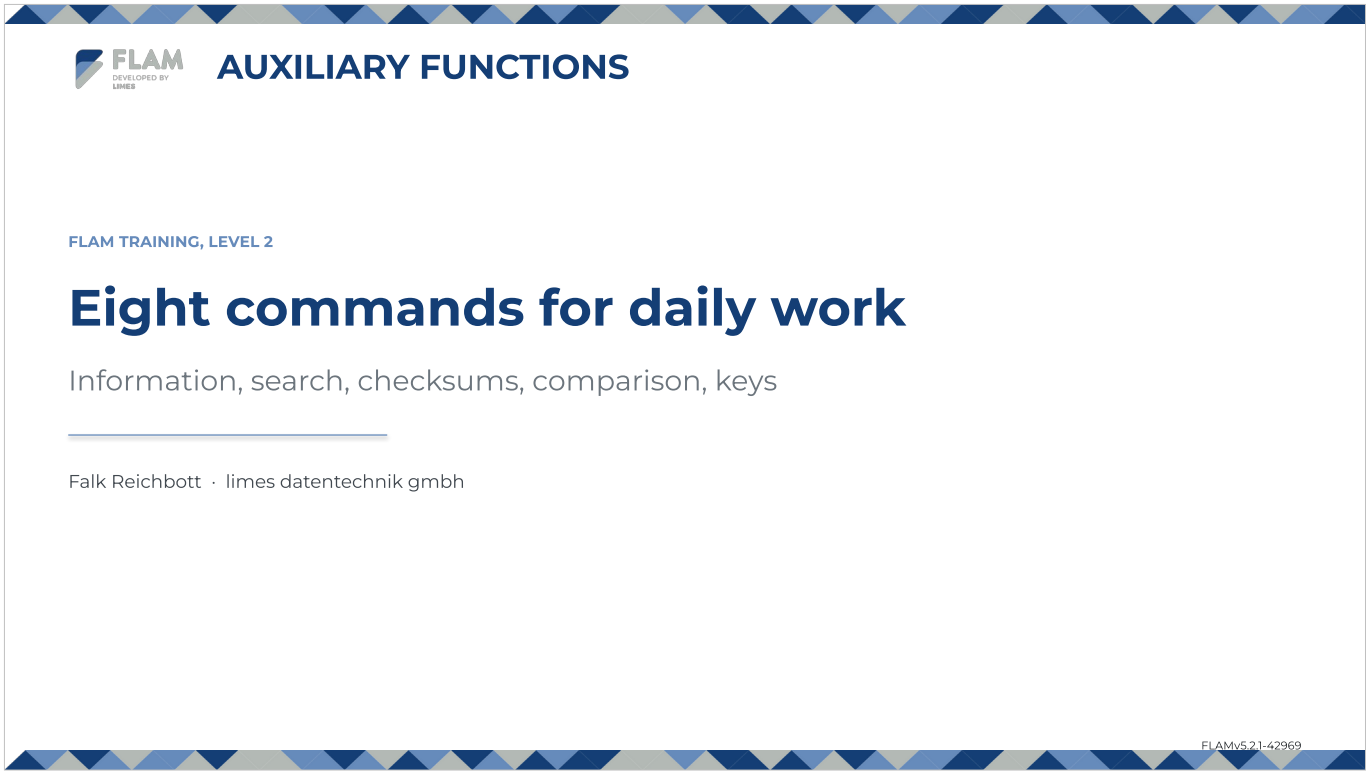
Four, and the first is the one that really happens in practice. On the first: somebody sets SAF up, a job fails, and to get it running again `ALTER` is granted. With that the protection is gone -- and it does not show, because nothing fails. That is exactly why the inversion has a slide of its own. On the second: the strict mode demands completeness. Whoever gives `UPDATE` on `MUSTDEF` and then forgets a resource brings operation to a halt at a place nobody thought of. On the third: switching `CLPAR` on first is the fastest way to shut a productive job down. On the fourth: a policy is a rule only if it is also defined.
In the friendly mode a **not defined** policy is accepted -- so it does nothing. Whoever introduces a guideline has to define it, not just name it.



Fifteen minutes, and the exercise is deliberately the bridge to the previous units. The solutions: **Task 1.** Define `POLICY.PGP.ENFORCE.ENCRYPTION`, `...INTEGRITY` and `...SIGNING` and authorise them so that they take effect.
With that the guideline no longer depends on whether a job sets the switches. **Task 2.** Activate `GLOBAL.AVE.CONTROL` -- that is, **no** `ALTER` on it -- and then define `FAVELIB`, `FAVEFUC` and `FAVEPAR` only for the library, the function and the scanner that are to be used. Everything else is thereby excluded. **Task 3.** Start with `CONTROL`. The strict mode demands a complete setup, and that is what one does not have at the beginning.
Switching over is possible later -- and should be done once the setup stands. **Task 4.** No, and this is the core of the unit: the switch in the job is a wish. It can be left out, and nobody notices. The policy is the rule.



Go through the learning objectives. And with that the security group ends: crypto, OpenPGP, operation, enforcement. The arc ran from the algorithm through the key to operation and to the rule -- and the rule is the only part that cannot be left out.



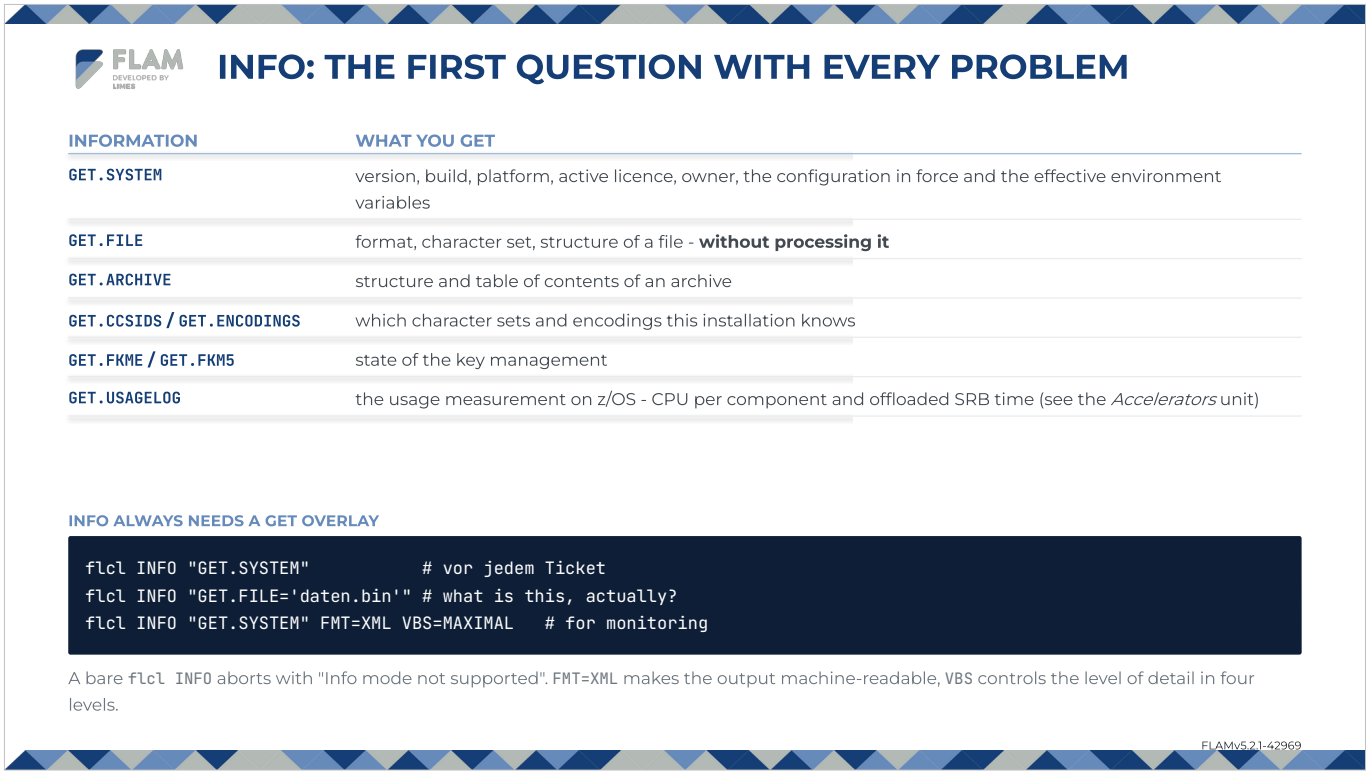
Fifth functional topic, and the one with the best ratio of effort to benefit:
eight commands that save time in daily work and that hardly anybody knows. Do not cover all of them to the same depth. GREP and INFO are the ones everybody needs.



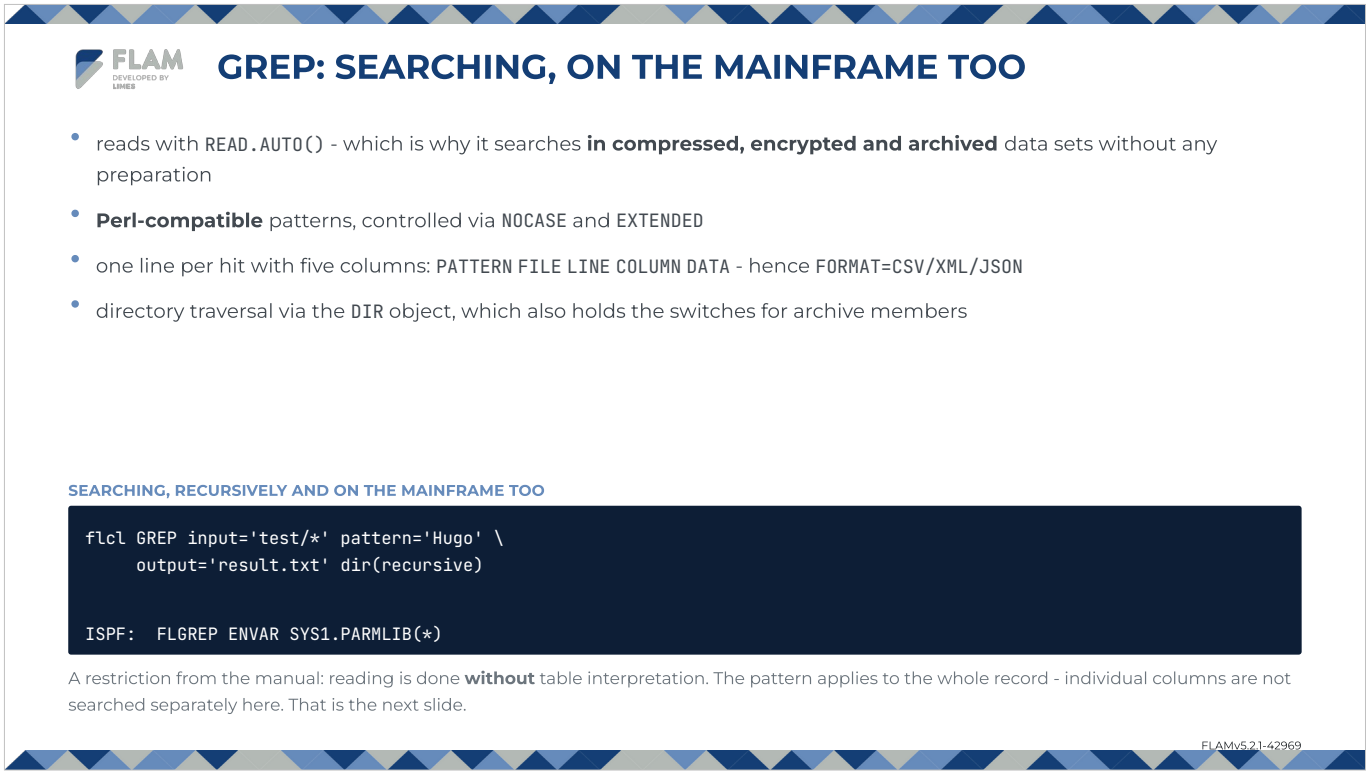
Objective two is the one that saves the most time on the mainframe -- and that many do not know, because they do not expect GREP under z/OS.



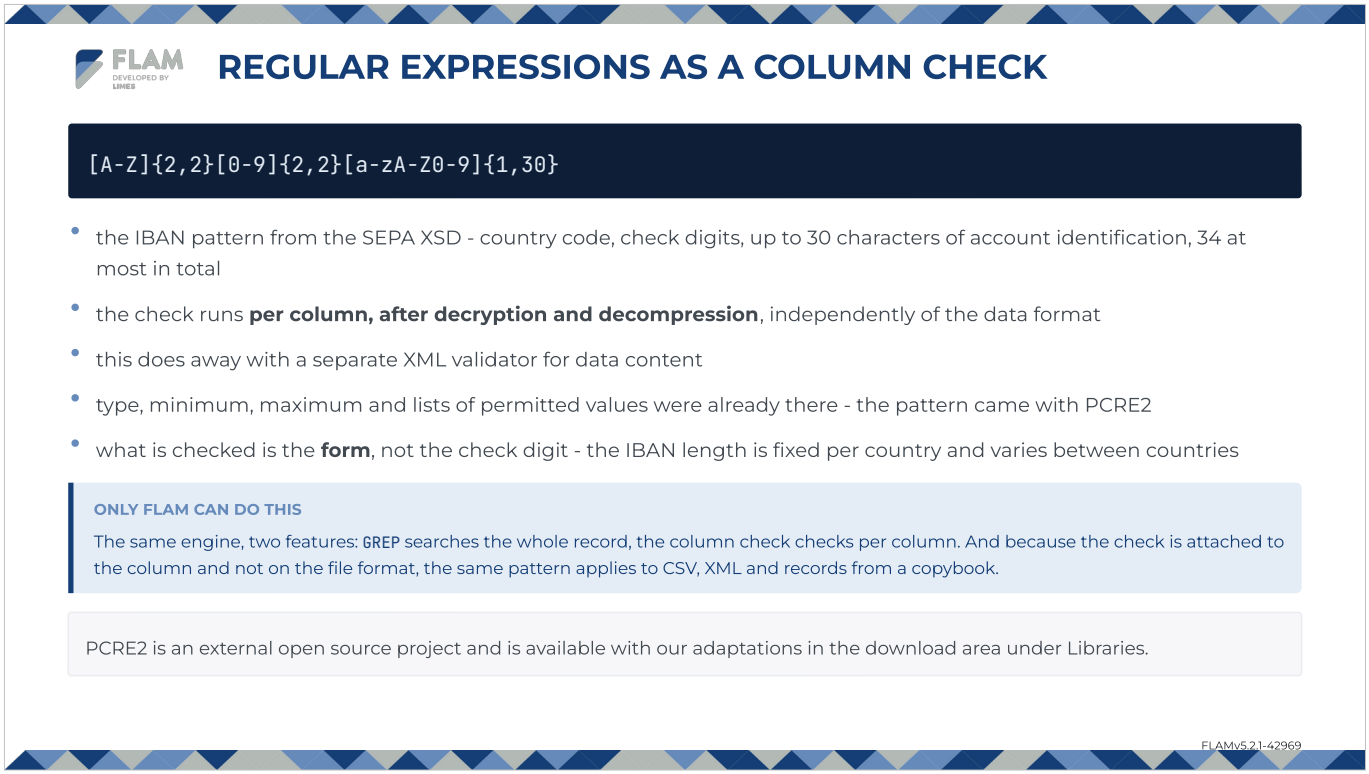
Go through it once, do not read it out. The purpose of the slide is that the participants know that these commands exist -- they get the details with `flcl HELP`.
If time is short: cover only INFO and GREP, leave the others standing as a list.



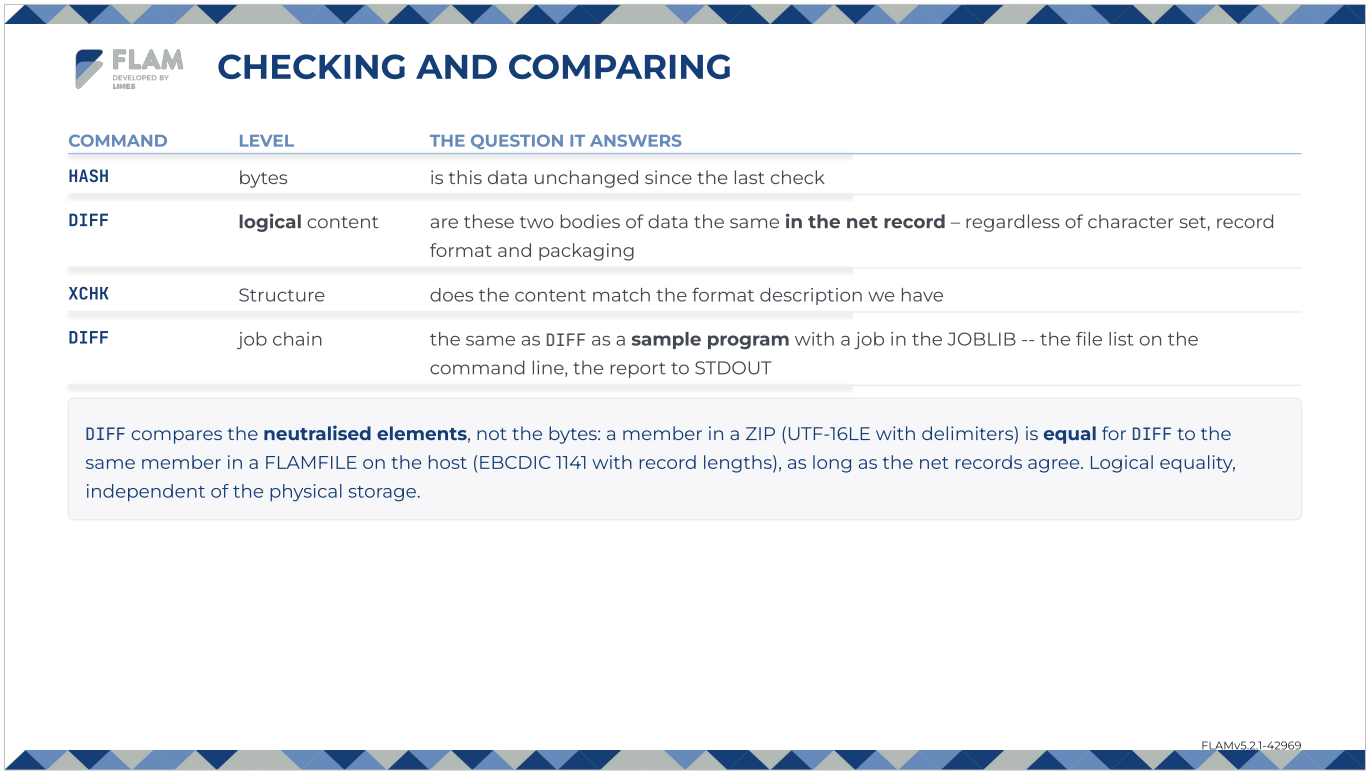
The command that spares you the first round of follow-up questions in every ticket.
Important for operation, because otherwise it goes wrong immediately: `INFO` always needs a `GET` overlay. A bare `flcl INFO` aborts with "Info mode not supported". That is not a defect but the language: `GET` is an overlay, and an overlay has to be selected.
Go through the seven pieces of information once. The two that surprise people in daily work are `GET.FILE` and `GET.ARCHIVE` -- you can query a file or an archive without processing it.
Mention `GET.USAGELOG` only on the mainframe and draw the arc to the unit on the accelerators: that is where the measured values sit with which you decide whether a zIIP offload is worth anything.
The output is controlled by two parameters worth knowing: `FMT=XML` makes it machine-readable, `VBS` governs the verbosity in four levels.
That makes `INFO` suitable for monitoring too.
And the sentence for operations: before every ticket run `flcl INFO GET.SYSTEM` and send the output along.



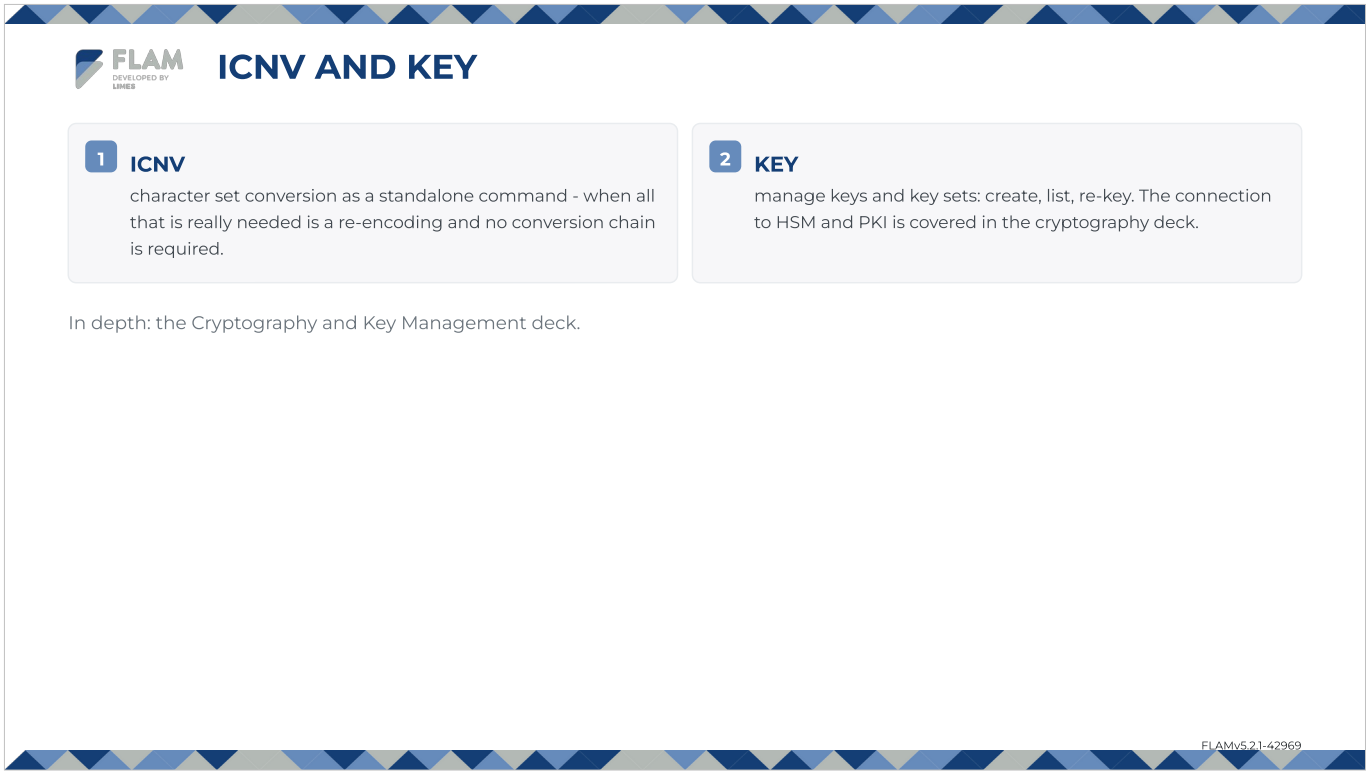
The surprise of this unit: there is grep under z/OS, and it also searches inside archives.
The mechanism behind it explains everything else, and it is worth it: `GREP` is a simplification of the conversion flow. Reading is done with `READ.AUTO()` -- that is why compressed files are unpacked, encrypted ones decrypted, and members of ZIP and FLAM archives are searchable via the switches in the `DIR` object. None of that had to be built specially; it falls out of the architecture.
The second thing that almost nobody knows and that distinguishes `GREP` from `grep`:
per hit a **row with five columns** arises -- PATTERN, FILE, LINE, COLUMN, DATA -- and writing is done with `WRITE.TEXT()`. That is why there are `FORMAT=CSV`, `XML` and `JSON`: the search result is machine-readable, without anyone having to parse the output. That is exactly what `grep` cannot do.
The patterns are Perl-compatible, controlled via `NOCASE` and `EXTENDED`.
And now the restriction that belongs on the slide, because the manual states it explicitly: reading is done **without** table interpretation. The pattern is always checked against the whole record, a row spec cannot be specified here, and individual columns are **not** searched separately. The column-wise check is the next slide -- a different feature, just the same engine.
Small things that turn up in support: comparison is done on the basis of UTF-8 in the neutral string format, trailing blanks are removed, and the delimiter is not part of the compared data.
On z/OS under ISPF the command is called `FLGREP pattern dsn`.



First the distinction from the previous slide, otherwise it blurs: `GREP` searches, and it searches the whole record. **Here** it is about validation, and specifically **per column**. Same engine, two different features -- anyone who mixes them up looks for the pattern in the wrong place. The reason PCRE2 was built in at all was not searching but exactly this validation: XML documents in payment traffic, SEPA and SWIFT-MX, were to become verifiable without a separate XML validator. The example is the IBAN pattern from the SEPA XSD -- country code, two check digits, then up to thirty characters of account identification, so at most 34 in total. A question to the room pays off here, because almost everyone misremembers it: **the IBAN has no fixed length.** It is fixed per country -- Germany 22, Malta 31 -- but variable across the countries. That is exactly why the XSD gives an upper limit and not a fixed number. And name the limit of the method straight away, otherwise the slide promises too much: the pattern checks the **form**, not the check digits. An IBAN with a wrong mod-97 check digit matches the pattern. For the form validation in the data stream that is exactly right -- the business validation remains the job of the application. The important point for security: the validation happens per column, after decryption and decompression, and independently of the data format. So it takes effect where the data lies in the clear -- in memory. And because it is attached to the column and not to the file format, the same validation applies to CSV, to XML and to records from a copybook.



Three commands that belong together because they all answer a question about equality -- on different levels. HASH is the byte level, DIFF the content level, XCHK the structure level. And here is the statement that makes `DIFF` valuable in the first place: **we compare the neutralised elements, not the bytes.** So the question `DIFF` answers is: are these two bodies of data **logically equal**? The example where it clicks: a member in a ZIP archive, stored there in UTF-16LE with delimiters, and the same member in a FLAMFILE on the host, stored there in EBCDIC 1141 with record lengths. Byte by byte the two have nothing to do with each other. For `DIFF` they are **equal** as long as the net records agree -- the physical representation is a matter of indifference to it. That is exactly what `HASH` cannot do, and exactly why these are three commands and not one. XCHK is what counts in a migration: does the content fit the description we have at all. And a hint for the host that stands in the manual page and is rarely known: there is **`SCFCUDIF`** as a sample program in the delivery, with a job in the JOBLIB. It takes a list of file names from the command line, compares the first against all the others and writes the report to STDOUT -- what is compared is the **net data** after decoding, decryption, decompression and character conversion, not the bytes. Four options: `-f` compare completely, `-o` byte offsets into the report, `-m` less protocol, `-r` the first name is the report file. For a house that needs a SuperC-like comparison in a job chain, that is the answer without learning a command.



Two commands for special cases, but both are needed regularly. ICNV is the character set conversion as a command of its own -- for the cases where you really only want to re-encode and do not need a chain. KEY is the tool for key management outside a run. The details are in the crypto deck; here only that the command exists.

PITFALLS

! **A pattern is not a wildcard.**

A regular expression and a file name wildcard are two different languages. There is a switch for it - check first which one applies.

! **A HASH without a reference is worthless.**

A checksum only proves something if the comparison value is stored separately and in a trustworthy place.

! **DIFF across different character sets.**

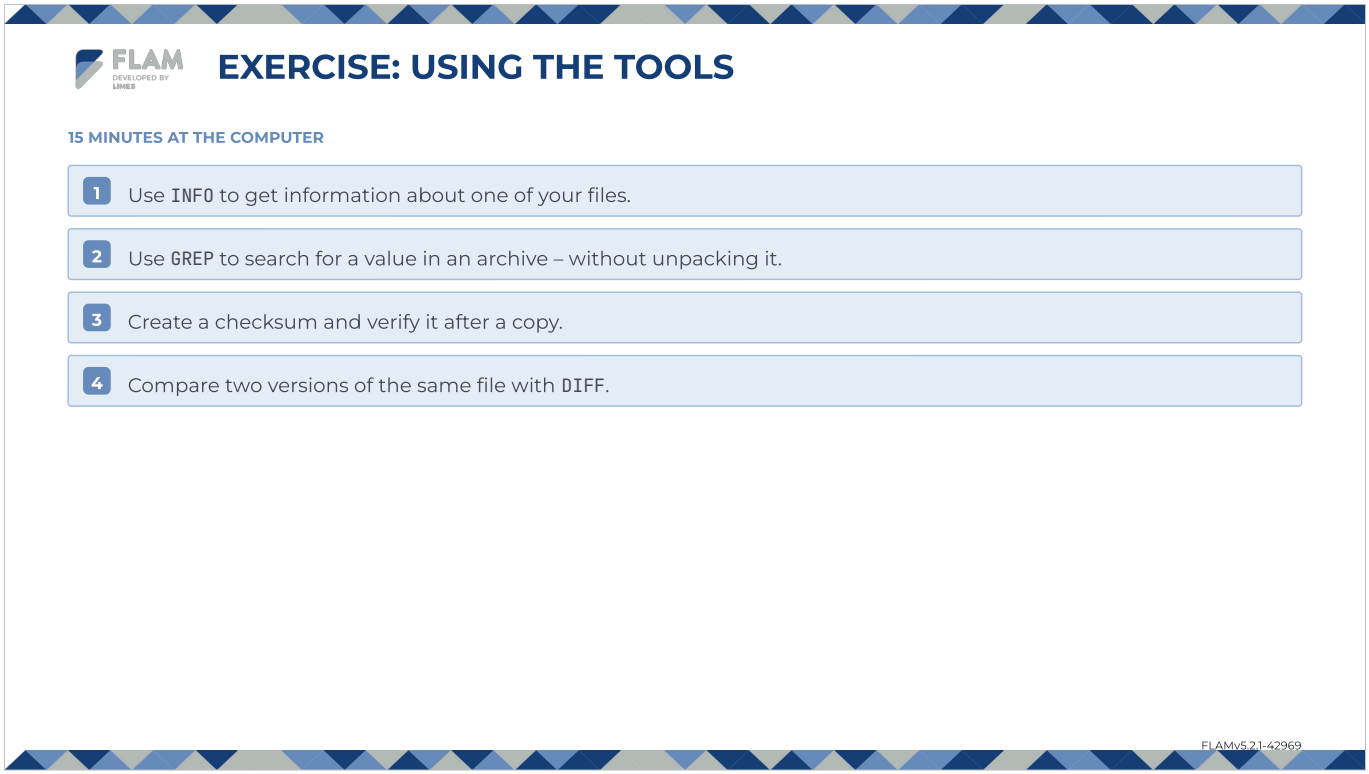
A comparison between EBCDIC and UTF-8 always finds differences. First bring both to one character set, then compare.

! **On the mainframe, grep is POSIX BRE.**

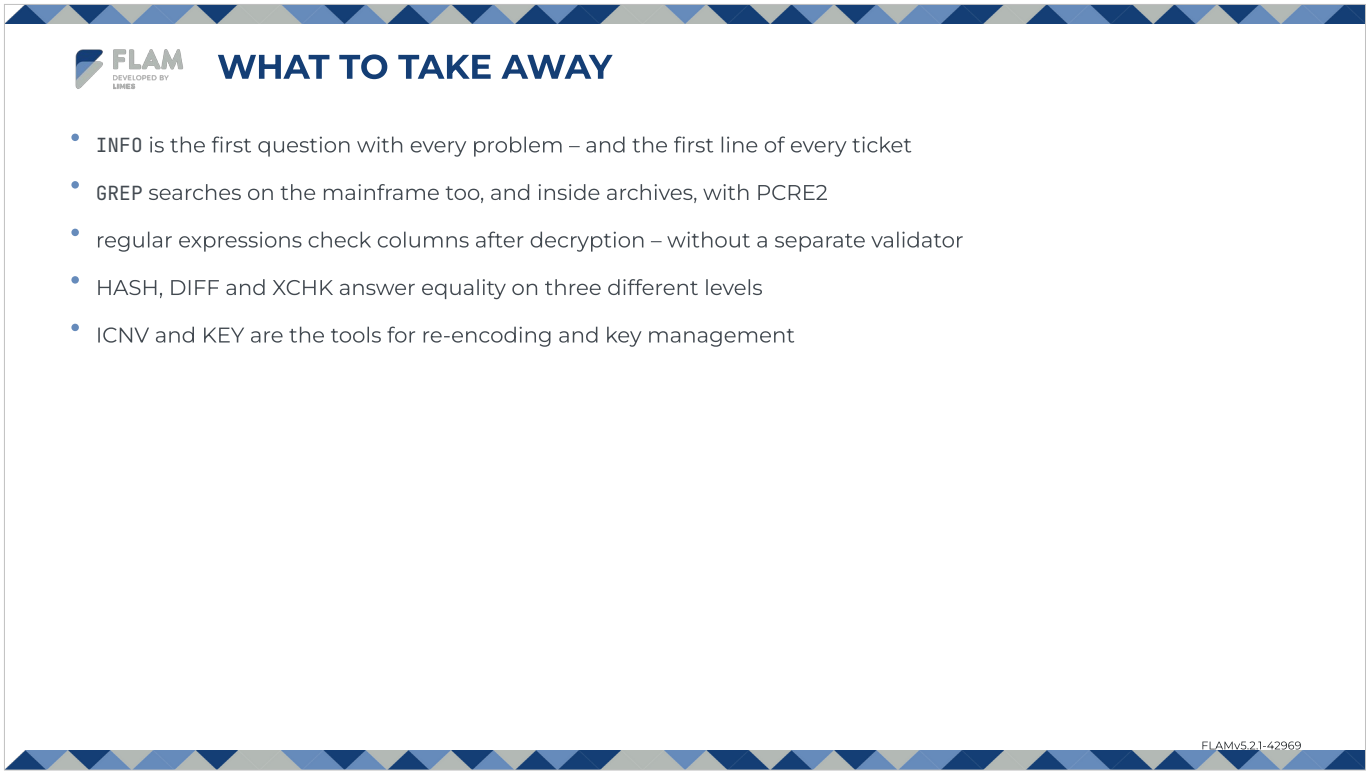
The syntax of the system tool is not that of PCRE2. If a pattern finds nothing and reports nothing, the engine is usually what is meant.

FLAMv5.21-42969

Three from support cases, one from experience with regular expressions.



Fifteen minutes. Exercise two is the one with the biggest aha moment: searching inside an archive without unpacking it.
Solution:
1. `flcl INFO GET.FILE='...'` tells you what a file is without processing it -- format, character set, structure.
2. `flcl GREP input='...' pattern='...'` finds the hits. With `FORMAT=CSV` the result becomes machine-readable; that is the answer to "and how do I evaluate that".
3. `flcl HASH` with `ALGO=SHA256` produces the checksum, `CHECK` verifies it against a stored one. `FORMAT=GNU` yields a file that `sha256sum -c` reads -- handy when the other side has no FLAM.
4. Keep the levels apart: `HASH` answers "are these the same bytes", `DIFF` answers "where do the contents differ", `XCHK` answers "does the content fit the format description". Two files can be identical in content and different byte by byte -- for instance after a character set conversion. That is exactly the trap you run into without this distinction.



Go through the learning objectives. That covers the functional topics for this training.



First unit on Thursday. This is the unit from which the participants take the greatest immediate benefit, because every one of them looks after job chains that look exactly like the one on slide 3. Opening: do not start with FLAM, start with their job chain. Ask how many steps their longest chain has -- the answer is the motivation for the whole unit.



Objective 3 is the real aha moment of this unit. Objective 5 is the one that makes the difference in operations between a clean job and a disruptive one.



Read this chain out slowly and count along while doing it. Eight steps, seven temporary files. Exactly this chain is the starting point for everything that comes afterwards. Whoever recognises it in their own organisation -- and everybody does -- listens to the rest of the unit differently. Important: this is not a badly built chain. It is built that way because each tool can do exactly one thing.

FLAM
DEVELOPED BY LIMES

WHAT THIS CHAIN COSTS

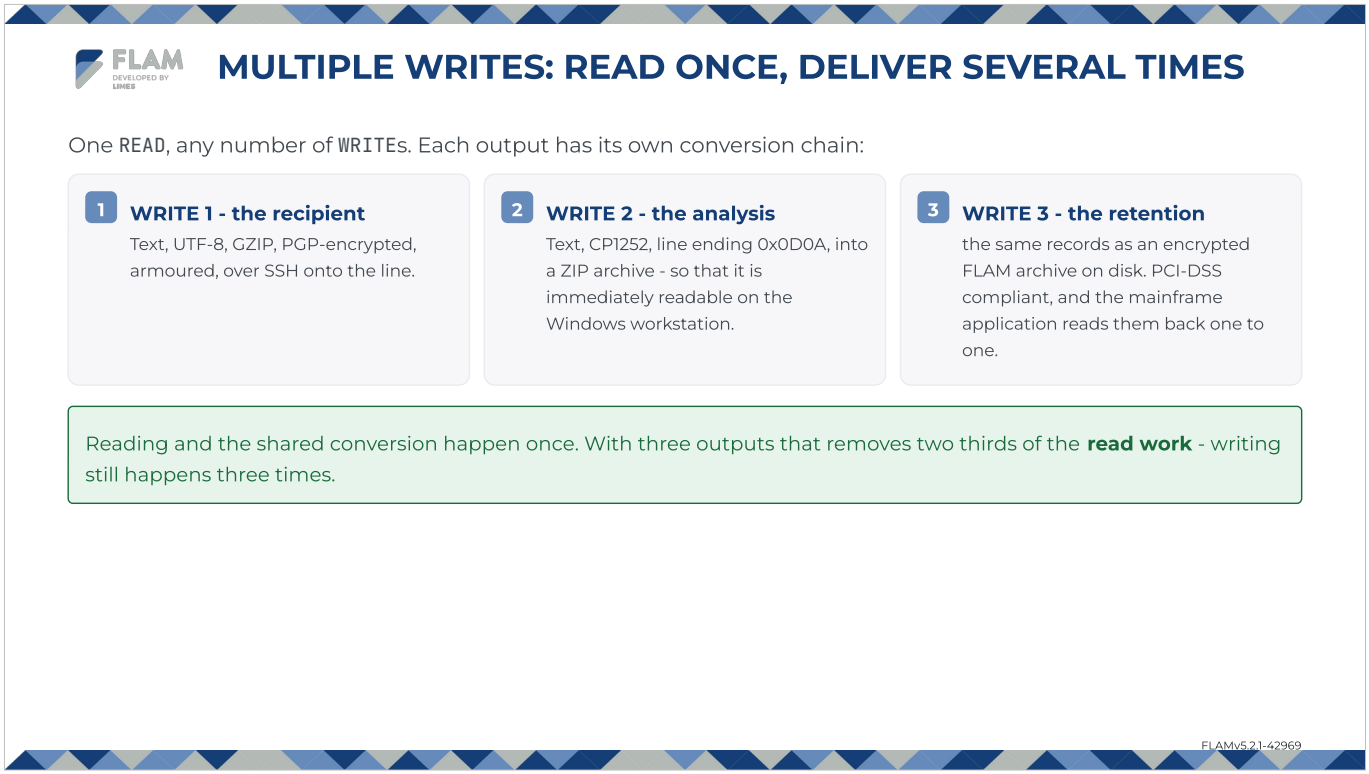
ITEM	WHY IT ARISES
I/O (EXCP)	seven temporary files, each written once and read once - the largest single item of the chain
tool overhead	eight programs are loaded, initialised and terminated; each one reads and writes its own buffers
interpreted code	delimiter and field reformatting as a REXX routine - the most expensive variant when run against bulk data
plain text on disk	the intermediate results sit unencrypted in the file system until step 8 deletes them - if it runs
operational effort	eight return codes, eight sources of error, eight places for the restart

FLAMv5.21-42969

The main item is the I/O, not the compute time of the individual tools. Every temporary file is written once and read once -- those are EXCPs, and on the mainframe they are the most expensive thing about the chain.
The second item is the REXX routines. Setting delimiters, reshaping fields, telling record types apart:
that is typically hand-written REXX, and interpreted code against bulk data is the most expensive way of doing it.
The third item is not in the bill, it is in the audit report.



This is the core slide. Do not talk about the syntax, talk about the structure: one READ, one WRITE, and everything in between are conversion stages in memory. The important sentence: there is no temporary file any more. The data leaves memory exactly once -- at the end, on the wire. Point back to the conversion unit: these are the same conversion stages, only this time in JCL. Two parameters in the example deserve a sentence, because they are often forgotten and the recipient then gets an unusable file.
`CCSID=1208` is UTF-8. And `METHOD=UNIX` sets `0x0A` as the separator -- without it FLAM writes the separator of the source platform, and on the host that is not `0x0A`. Anyone delivering to Unix and forgetting `METHOD` delivers lines nobody there can separate. On the read side there is deliberately no `CCSID`: the detector works there. It decides ASCII or EBCDIC, and which code page that is in the end is said by the environment -- the default CCSID for ASCII or for EBCDIC respectively. It is set either directly or through `LANG`; out of `de_DE` comes IBM-1141 on the host. That is the reason why `LANG` stands in the environment of the step on slide 10. The encryption sits in `ENCRYPT.PGP()`, and `ARMOR()` is an object **inside it** -- not beside it. That is the structure of the language: what belongs to PGP is inside PGP.
One parameter in there is mandatory, and in examples it is often missing: **`USERID` -- the user ID of the recipient.** PGP encrypts for a recipient, and that one has no default; one's own user ID is expressly **not** the standard. Whoever wants to work symmetrically gives `PASSWORD` instead, and whoever wants to sign as well gives `SIGNID` -- which is then one's own user ID. The chain above is deliberately complete so that the order becomes visible. `CNV.BLK` turns the records into blocks with separators -- that has to happen **before** the character conversion. Then `CNV.CHR` for the character set. Then `CNV.AVS`, the virus scan:
it runs on the **readable** text, that is before compressing and encrypting. After that no scanner can find anything any more, neither here nor at the recipient. In the parameter string that is `AVSCAN()` on the `WRITE`. Only then `CNV.GZP`, `CNV.PGP` and `CNV.BAS`. What can be shown on request without overloading the slide: the run can additionally produce a **checksum** or a **detached signature**, and through post-processing (`POST`) something can be triggered on the far side -- either the processing directly, or simply a `RENAME` so that a cron job there picks the file up. The chain then does not end with the transfer but with the start of processing on the target system.



This is the point at which the bill comes out in FLAM's favour once and for all -- and the point that almost nobody knows.
The read operation and everything that is common to all outputs happens exactly once. Only after that does it branch. The arithmetic is simple and you can do it in front of the room: three separate runs read three times, one run with three writes reads once -- that is two thirds of the read work that falls away.
The restriction is important, otherwise you promise too much: writing still happens three times, and the conversion only falls away to the extent that it is common to all outputs. How much that gains in elapsed time depends on the ratio of read work to write work -- on the mainframe, where the EXCP is the expensive item, it is a lot.
The example is real: the same data set goes to the recipient as UTF-8, as CP1252 with CRLF into a ZIP for the Windows evaluation, and as an encrypted FLAMFILE onto disk, so that the host application can read the records again one to one later. The last one is the PCI DSS point.
Available in all three access routes: batch, API and subsystem.

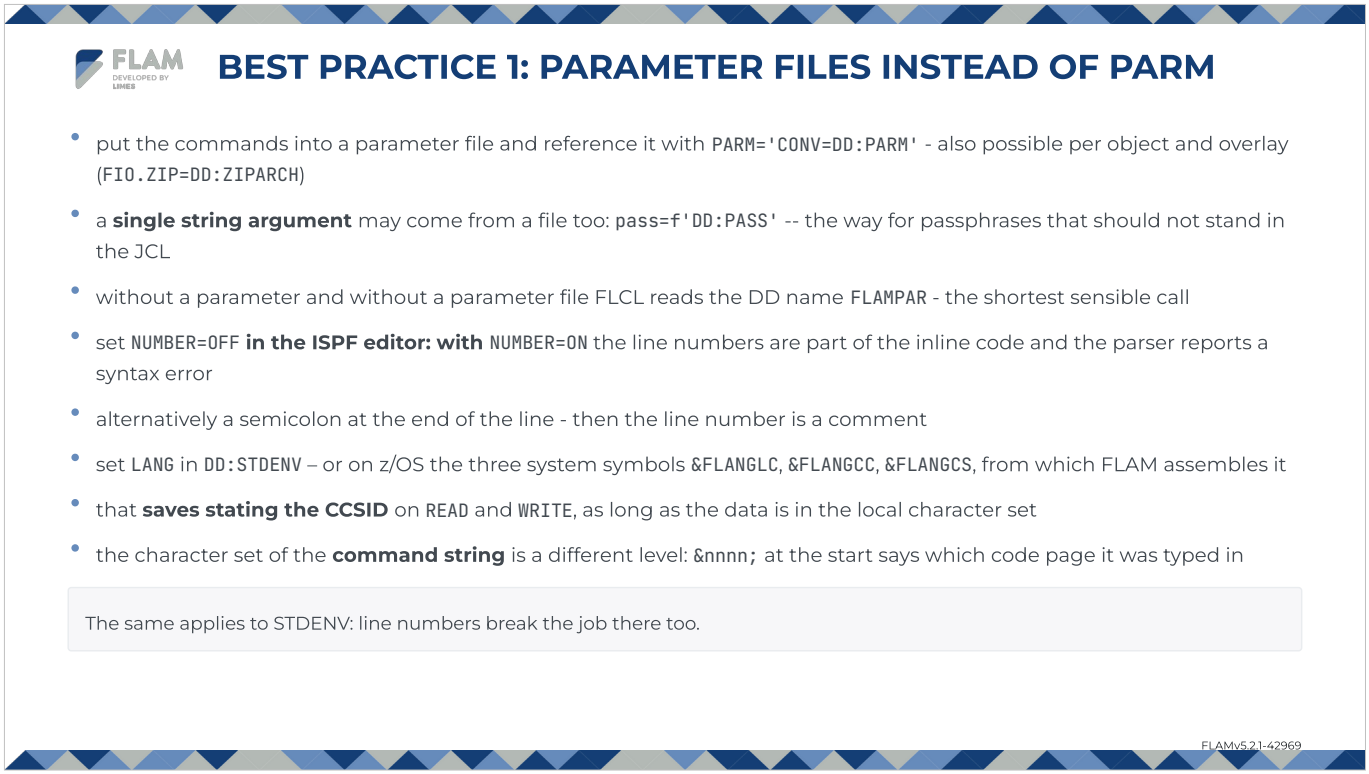
MULTIPLE I/O: ONE RUN, SEVERAL TARGETS

	MULTIPLE WRITES	MULTIPLE I/O
what differs	the conversion per output	only the target
how often conversion happens	once per output	exactly once
typical case	UTF-8 for the partner, CP1252 for the analysis	the same delivery to two recipients, or disk plus a tape copy

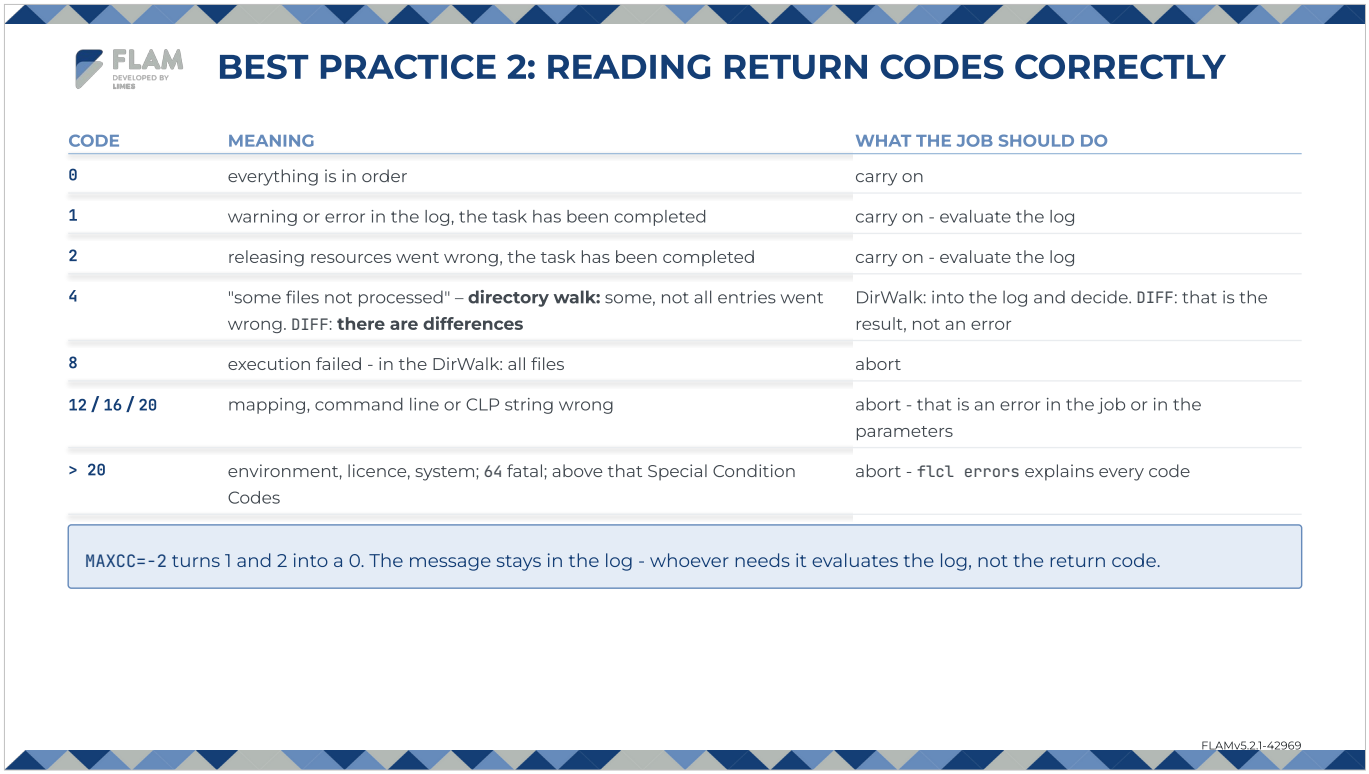
The two can be combined: several writes, and individual ones among them write their result to several targets. Whoever uses all the options saves the most.

FLAMv5.21-42969

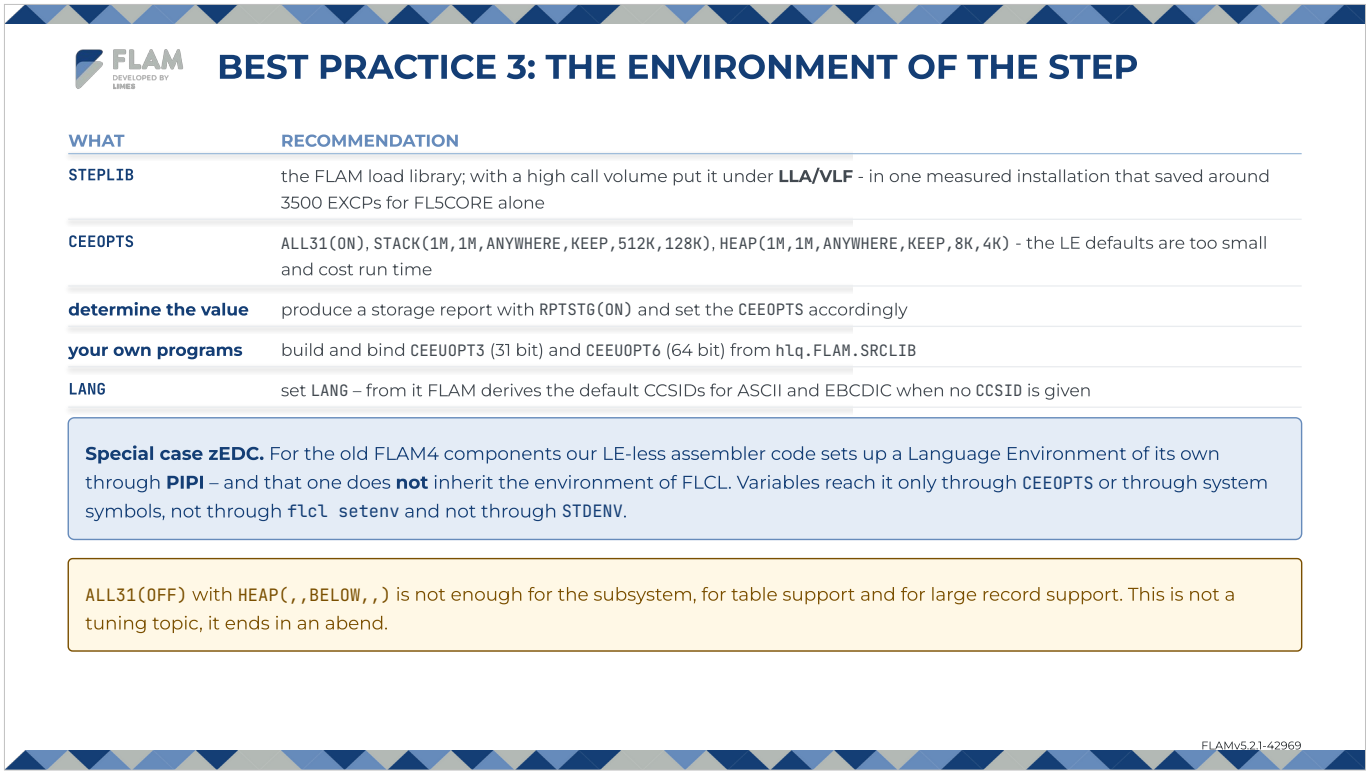
Work out the difference from Multiple Writes clearly -- that is the frequent confusion.
Multiple Writes = different results, each with its own conversion.
Multiple I/O = one result, several targets; conversion happens exactly once.
And you can combine the two: three writes, and the second of them goes to two targets.



The practical part, and first a clarification, because the question about the length of the parameter string comes up regularly:
there is no limit on our side. FLAM5 processes a parameter string of any length, FLAM4 takes whatever z/OS passes at most. The old limit of 100 characters is history, and with `PARMDD` there is a second way of reading the string from a DD card. The reason for a parameter file is therefore not the length, but: readability -- a call spread over several lines with indentation can be read in a review, a PARM string cannot. Reuse -- the same file from several jobs. And the quotes, which have to be doubled inside PARM. The three traps in the order in which they occur: NUMBER=ON in the ISPF editor, a missing LANG, and forgetting MAXCC. NUMBER=ON is by far the most frequent support request on this topic. The line numbers then end up inside the inline code and the parser reports a syntax error at a place that looks perfectly correct in the editor. Two things belong to `LANG` and have to be kept apart -- the same separation of levels as in the unit *Character sets*: **`LANG` applies to the data.** It says which character set the mass data is in locally, and thereby saves stating the CCSID on `READ` and `WRITE`. On z/OS it is set through `DD:STDENV` or through the three system symbols `&FLANGLC`, `&FLANGCC` and `&FLANGCS`, from which FLAM assembles the value. **The command string is a different level.** The misinterpretation of the tilde and the square brackets concerns the parameter string we read in -- not the data. Anyone editing in a 3270 can say with `&nnnn;` at the start which code page was typed in; without that statement `LANG` applies. And the reason why the two can come apart on the host at all: **JCL actually has to be written in 1140, because JES only understands 1140** -- but the data on a German system is often 1141. In practice that mostly works, and it is worth saying why: most German workplaces have 1141 in the 3270, and as long as the JCL itself manages without critical characters -- which normal JCL does -- the submit works, the inline code is written in 1141 and is read as 1141. So it only works because JCL does not need the diacritical characters. As soon as a bracket or a tilde appears in the parameter string, `&nnnn;` is the clean way.



This is the slide that prevents an unnecessary night-time alarm. Return code 1 means: there is a warning or an error in the log, but the task is done. Return code 2 means: something went wrong while releasing resources, the task is done. In both cases the follow-on processing may run. The `2` needs a placing in case it does turn up: it means that we did not get through the finishing part -- `FLUC`/`FLAMFIN` -- without an error, so possibly **not all resources were released**, a file not closed for instance. Little is to be expected in the log about it, because the logging is no longer reliably available in the finishing part. The result of the task is not affected by it, and in practice the `2` has **not occurred** so far. MAXCC=-2 turns that into a 0. That is not whitewashing -- the log still holds the message, and anyone who wants to evaluate it evaluates the log, not the return code. Now the point that causes the most trouble in practice -- the `4`. According to `flcl errors` it literally means "some files not processed", and that presupposes more than one file. In the code there are exactly two places that produce it: **First the directory walk or a member list**: some, but not all entries went wrong -- "There was an error at n of m members". Typical are missing access rights on a single file. If **all** went wrong, an `8` comes back. That includes the case where the directory walk itself warns, for an unreadable subdirectory for instance.
Second `DIFF`: if the comparison finds differences, the result is a `4`. Here the `4` is not a defect but **the answer** -- a job that treats 4 as an error across the board cannot work with `DIFF`. Processing **one** file yields no `4`:
a warning in the log gives `1`, a real error `8`. Anyone seeing a `4` on a single file either has a directory walk (wildcards in the name) or a `DIFF`. Above that it gets systematic: `12` means the mapping of the parameters failed -- something was not there in the system. `16` means a syntax error on the command line, `20` a syntax error in the CLP string itself. From `24` upwards they are environment and system errors, `40` is access control or licence, `64` is the fatal error. Above `64` lie the special condition codes which a command sets itself for the job control. The complete list is fetched by anybody with `flcl errors` -- it is in the manual as well. On the slide are the ones that really occur in a job chain.



This is the slide for the systems programmer in the room. CEEOPTS: the LE default increments are too small for FLAM. With a 128-byte increment the run time and the CPU time rise considerably, because the Language Environment keeps having to load more. 1 MiB for stack and heap is the starting value; RPTSTG(ON) delivers the value that really fits. Since 5.1.25 FLAM brings its own defaults along via CEEUOPT -- CEEUOPT3 for 31 bit, CEEUOPT6 for 64 bit, both in hlq.FLAM.SRCLIB. Whoever builds their own applications against our APIs should build and link the samples along with them. ALL31(OFF) with HEAP BELOW is not enough for the subsystem, the table and the large-record support -- that is not tuning, that is an abend. A special case that costs time when you do not know it: **the old FLAM4 components with zEDC.** Our LE-less assembler code sets up a Language Environment of its own through **PIPI** for the zEDC, or the IBM zlib respectively -- and the environment of FLCL is **not** inherited into that environment. Only `CEEOPTS` is read there. So whoever wants to set variables for the old FLAM4 components while they work with the LE through PIPI (currently only with zEDC) gets there only through `CEEOPTS` or through system symbols -- not through `flcl setenv` and not through `STDENV`. Loading the modules is the one item that a system can remove completely: the FLAM load libraries under LLA/VLF. In one measured installation that was around 3500 EXCPs for FL5CORE alone. But: a library taken in with FREEZE delivers the copy cached by LLA -- before maintenance it has to come out of LLA control and go back in afterwards.

BEST PRACTICE 4: JCL SYMBOLS IN THE COMMANDS

```
//E0       EXPORT SYMLIST=(CHAROUT,HASHALGO)
//S1       SET    CHAROUT=IBM1141
//CONV     EXEC   PGM=FLCL,PARM='CONV=DD:PARM'
//PARM     DD     *,SYMBOLS=JCLONLY
   WRITE.TEXT(FILE='/u/<cuser>/export.gz' COMP.GZIP();
              HASH(CCSID=<CHAROUT> ALGO=&HASHALGO.));
```

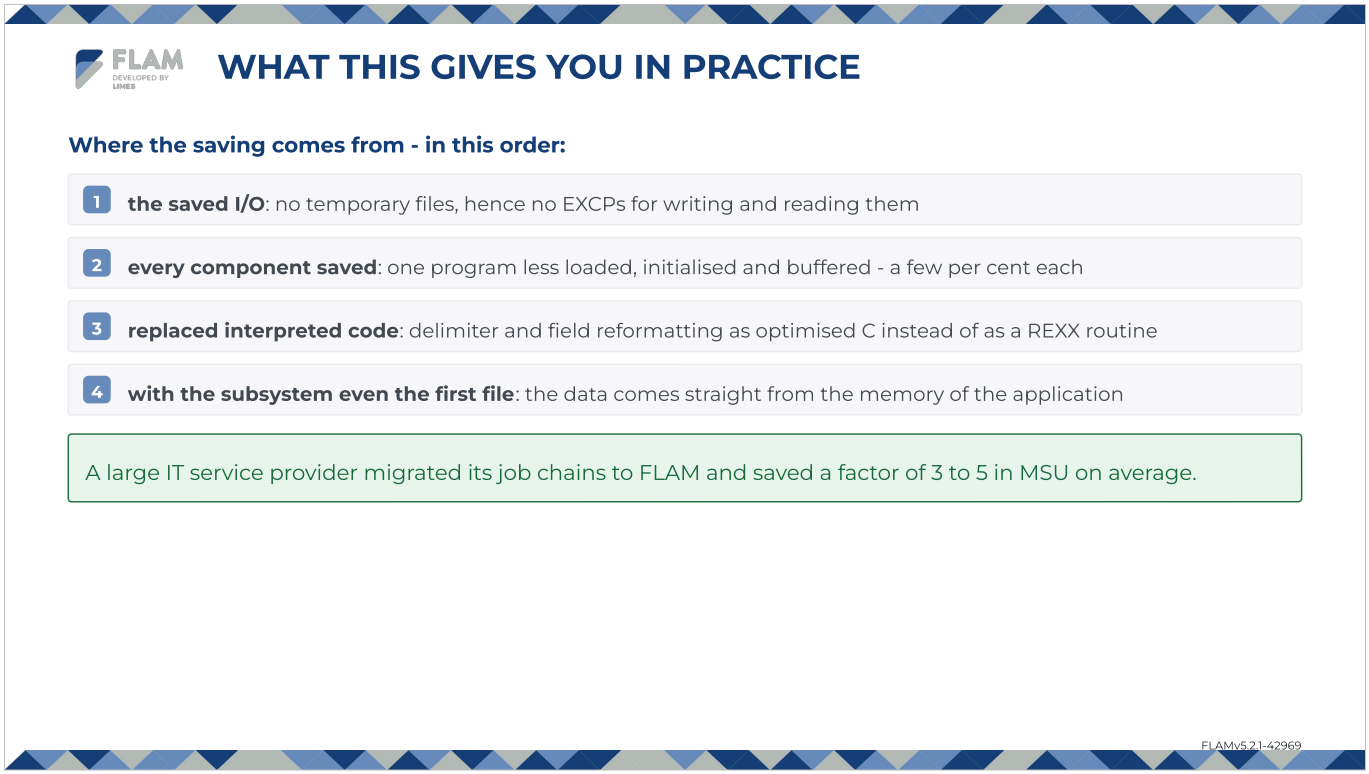
- `SYMLIST=*` exports all symbols - that is what makes JCL procedures genuinely reusable
- priority: **environment variable** before **JCL symbol** before **system symbol** (`<&LDAY>`)
- in an **external** parameter file only `<name>` applies; `&name.` is a JES substitution and works only inline
- if the Language Environment is not in the `STEPLIB`, `CEEGTJS` is not available and the symbols are not substituted

FLAMv5.21-42969

The route to reusable procedures. Since z/OS 2.1 FLAM can determine the values of exported JCL symbols via CEEGTJS and use them like environment variables. For that the Language Environment has to be in the STEPLIB concatenation.
The priority matters, because it makes the difference when debugging:
environment variable beats JCL symbol beats system symbol.
And the syntax trap: in an external parameter file only the angle-bracket form applies. The ampersand with a dot is a substitution by JES and only works with inline coding.



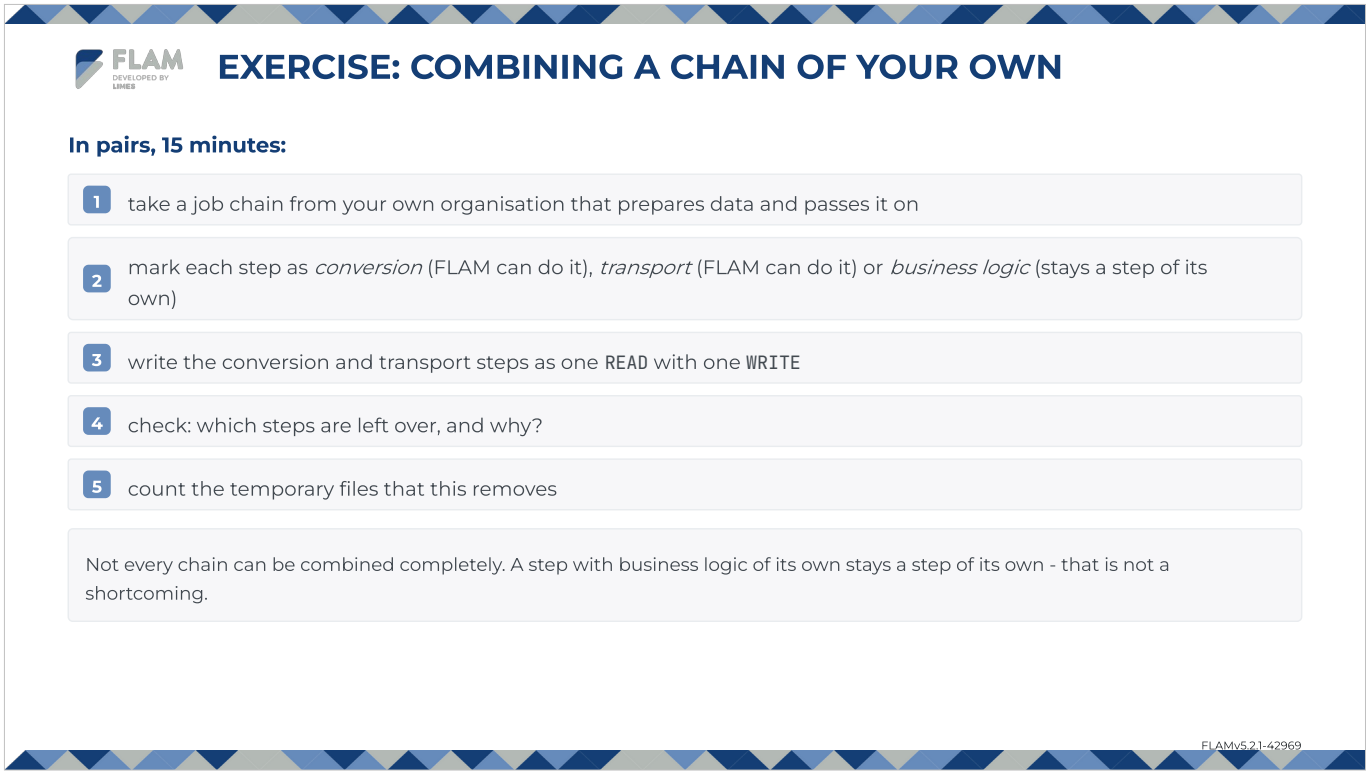
This slide answers a question that comes up in every introduction: who clears the intermediate and source files away?
The four switches look similar but do different things -- and one of them is dangerous if you do not think of the directory walk as well.
`REMOVE` **on the read side** deletes the source file if it was processed successfully. That is exactly what you want when collecting from a handover directory: what has been fetched is gone, and a restart does not fetch it twice.
And here is the trap: **in the directory walk, with a return code 4 the original of the successful files has already been deleted.** Whoever then repeats the run does not transfer exactly those again -- which is correct, but you have to know it, otherwise you go looking for the files. So the restart works off the remainder, not everything.
`REMOVE` **on the write side** is the opposite: it deletes the written file **if there was an error** -- so that no incomplete file is left standing that a follow-on job takes to be finished.
`RENAME` exists on both sides and only ever takes effect on success. That is the usual pattern for handover directories: first write under a working name, then rename -- the recipient only sees the file once it is complete.
`EMPTYFILE` governs what happens with empty results: log only (`INFO`), as a warning (`WARNING`), as an error with a condition code of its own (`ERROR`) or not created at all (`DELETE`).
The last point is the most important one for practice and the reason why the slide exists: **`REMOVE` and `RENAME` go over SFTP.** Pre- and postprocessing execute a command and therefore need the SSH shell. Many customers have SFTP rights on the partner system, but no shell rights. Whoever knows that beforehand builds the flow with `RENAME` instead of with a postprocessing script -- and saves themselves a rights request that often takes weeks and is sometimes refused.
One more thing about the target name, because it goes wrong on the first attempt: the name is formed from **tokens in square brackets** -- `[base]` is the name without the extension, `[name]` with it, `[ext]` the extension only, `[path]` the directory, `[indN]` a running index. There is a handbook page of its own on it.
And the trap: `rename='[base].done'` puts the file in the **working directory**, not next to the source -- for that you need `rename='[path]/[base].done'`.



Here are the numbers, but with the right reasoning. Whoever quotes the number without the reasoning is rightly not believed.
The main reason is the I/O that is saved: the temporary files fall away, so their EXCPs fall away. On top of that comes the fact that every component saved brings another few per cent, and that replaced REXX routines for delimiters bring considerably more than a few per cent.
The special case of the subsystem: there even the first file falls away, because we read the data directly out of the application's memory.
The number itself: a large IT service provider converted its job chains to FLAM and saved on average a factor of 3 to 5 in MSU. Do not name the customer.
The point that is not measured in MSU but counts in the audit: the intermediate states do not exist any more.



Keep it short, but do not leave it out. In a bank this is often the stronger argument than the MSU figure.
The chain on slide 3 writes seven intermediate states to disk unencrypted.
Anyone with access to the file system has access to the clear text -- regardless of how good the encryption is at the end. And step 8, the one that cleans up, is exactly what does not run after an abort.
With the subsystem there is not even a copy of the input file.



15 minutes in pairs, then walk through two chains with the whole room.
Step 4 is the important one: not every chain can be summarised completely. A step that calls a foreign program with its own logic stays a step of its own -- and that is fine. Anyone who says so is believed.
Solution:
2. A rule of thumb for the classification: setting delimiters, converting the character set, compressing, encrypting, armouring, transferring and cleaning up are all conversion or transport -- so FLAM. Business logic is everything that checks values, derives them or merges them from several sources.
3. The result is a `READ` of the input plus a `WRITE` with the chain of conversion stages. If you need two different target formats, you write two `WRITE` statements -- that is the multiple-write case.
4. What typically remains: the business-logic step, a scheduler step and possibly a release or check step. Three out of eight is a good result, not a bad one.
5. The number of temporary files that disappear is always the number of merged steps minus one.

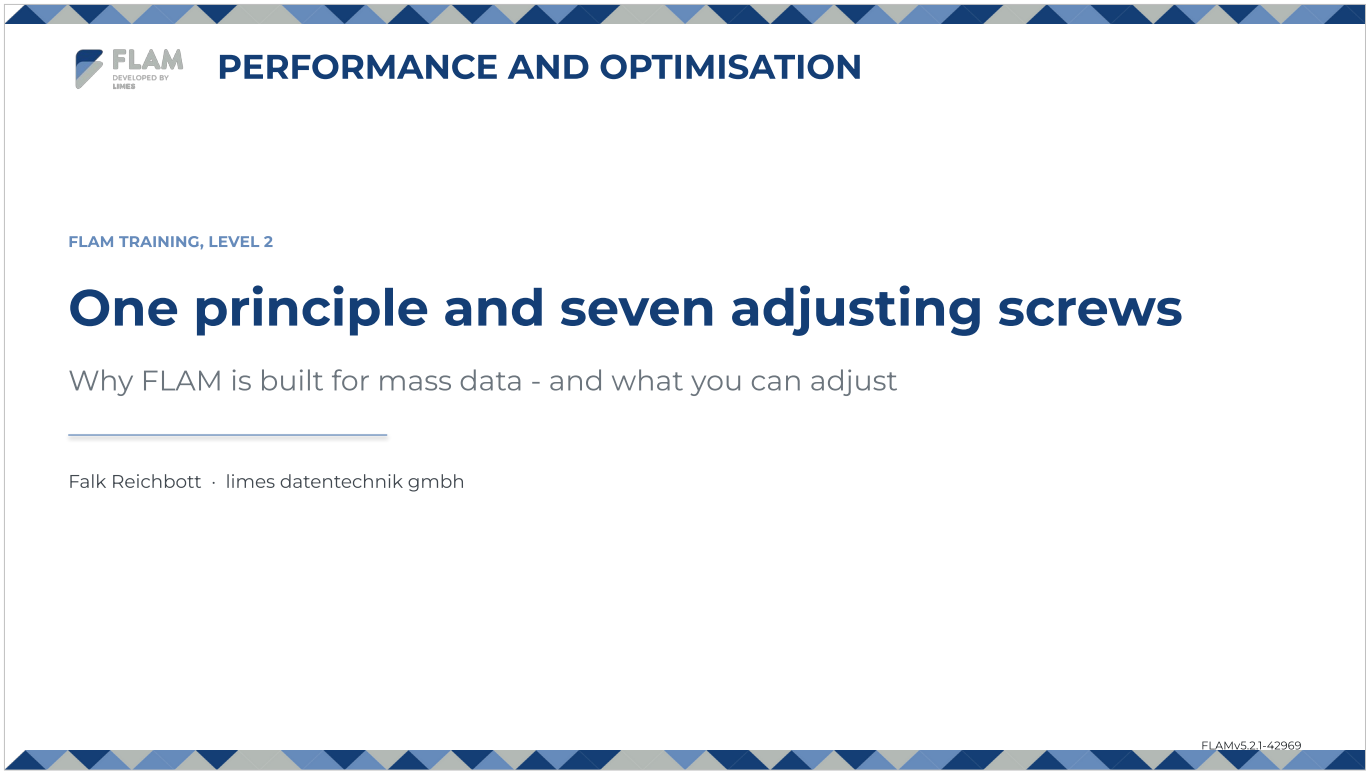
WHAT TO TAKE AWAY

- a job chain becomes **one step**, because the conversion stages run one after another in memory
- the largest item is the **saved I/O** - the temporary files disappear
- **Multiple Writes** and **Multiple I/O** deliver several results from one read
- `MAXCC=-2` for the codes 1 and 2, `NUMBER=OFF` in the editor, `LANG` in `STDENV` - the three most common stumbling blocks
- `CEEOPTS` and `LLA/VLF` are the two tuning knobs outside FLAM

Next unit: what else you can set on this one step - block size, record count, character set conversion, logging.

FLAMv5.21-42969

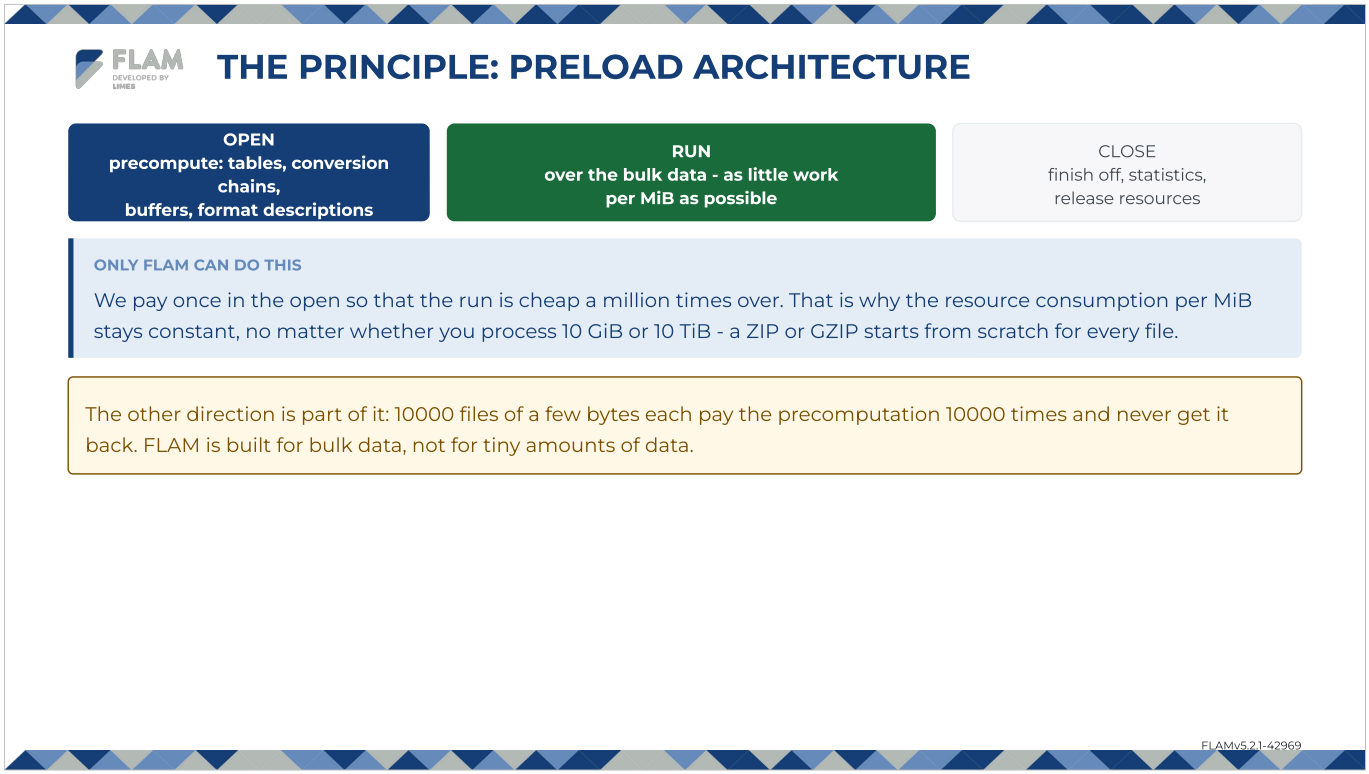
The last point is the bridge to the next unit: why it is worth optimising the remaining step further.



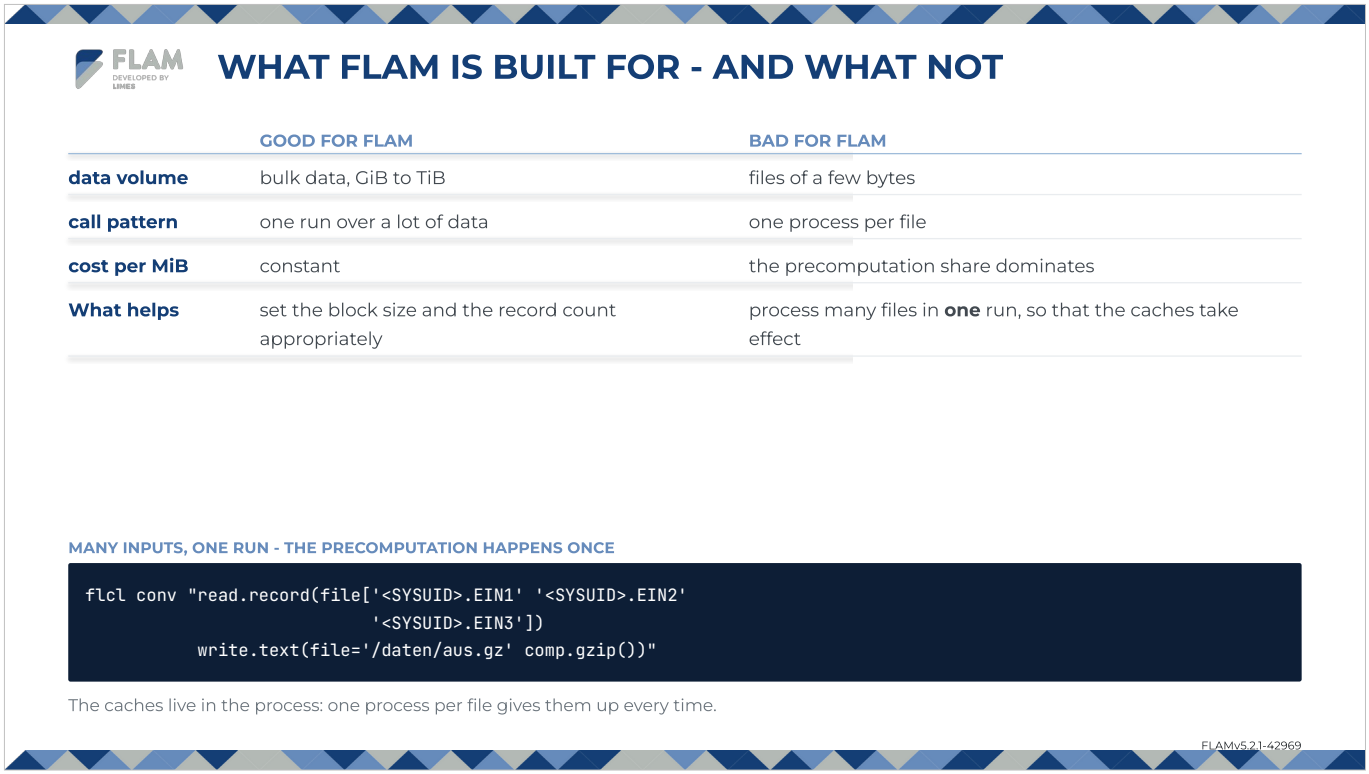
Second unit on Thursday, straight after the batch embedding. The order is deliberate: first turn the chain into one step, then tune that one step. Open with the question: which of you has ever experienced FLAM as slow?
The answers are almost always the same case -- many small files. That is exactly what slide 3 leads up to.



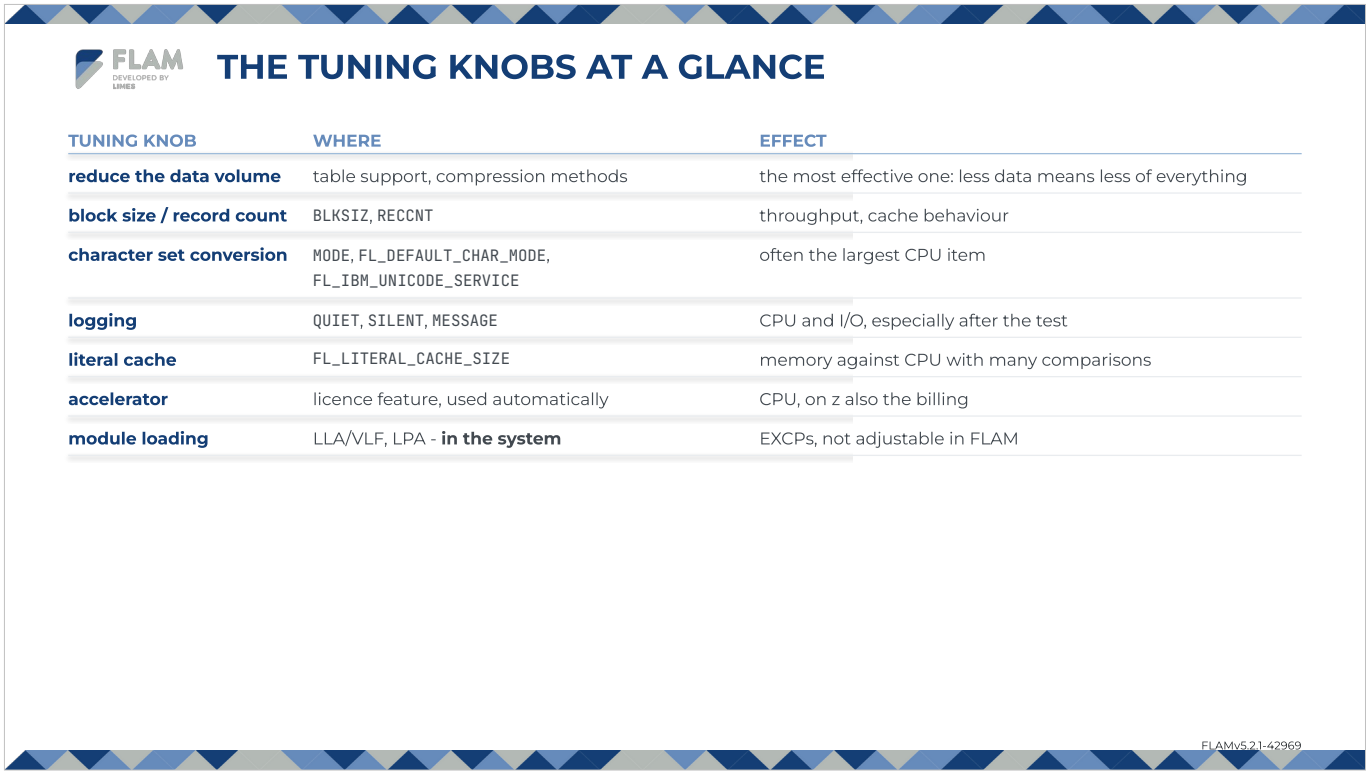
Objective 1 is the most important one. Anyone who understands the preload architecture can decide independently whether FLAM is the right tool for a given use case -- and does not need to ask us.



This is the central slide of the unit. Take it slowly.
The core: in the open, FLAM precomputes as much as possible -- tables, conversion chains, buffers, format descriptions. That costs memory and it costs CPU once. After that, the run over the bulk data is as cheap as it can get.
The phrasing that sticks: we pay once so that we are cheap a million times over.
Do not keep quiet about the other direction -- that is why this slide builds trust: anyone processing 10000 files of a few bytes each pays the precomputation 10000 times and never gets it back. CPU consumption goes through the roof there, and that is not a defect, it is the design.
Mention the caches, but do not sell them as the solution: they dampen the effect, they do not remove it.



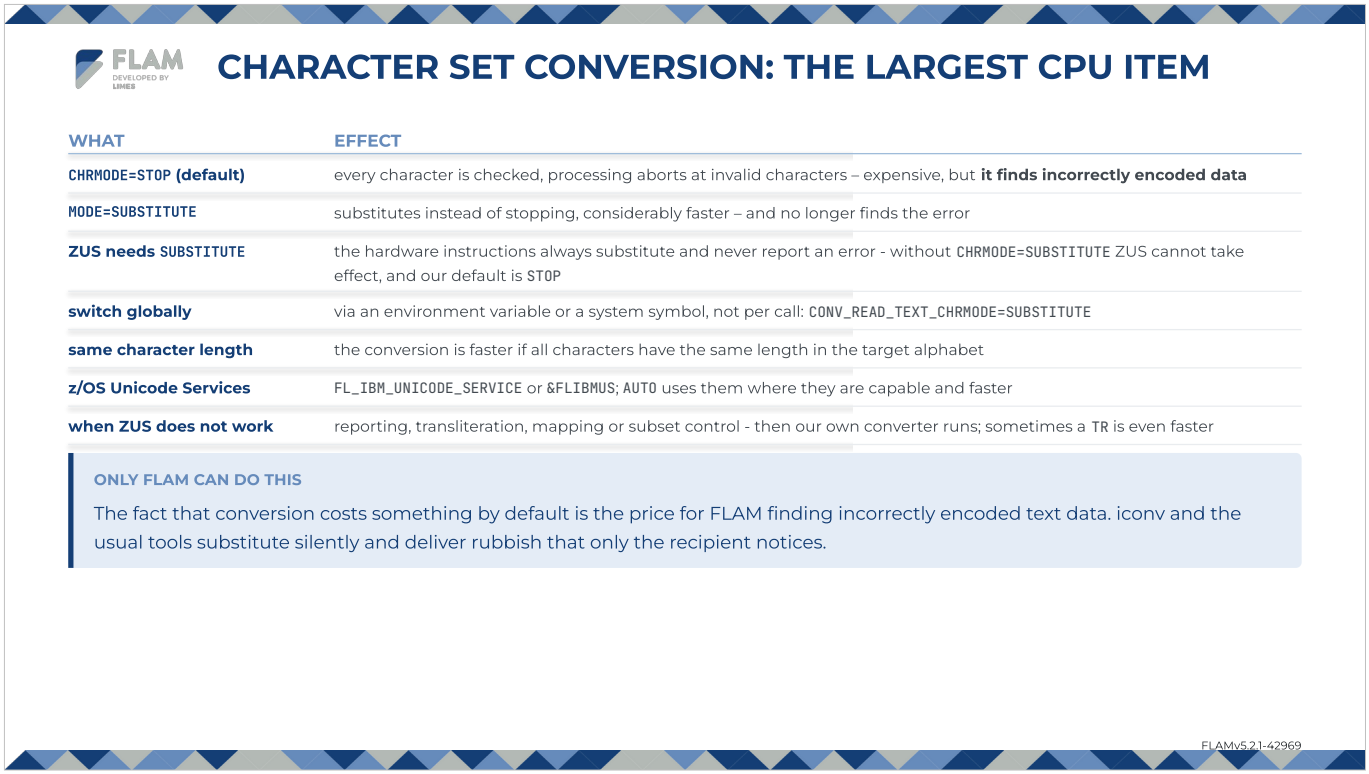
This slide answers a question that keeps coming up in operations, and it answers it honestly. That earns trust, it is not an admission of weakness.
Name the caches concretely, because they show that we know the problem and have addressed it: the character set conversion tables are cached, the literals are cached. Anyone processing many small files in one run benefits from that. Anyone starting a separate process per file does not benefit -- the cache lives in the process.
Hence the practical recommendation: process many small files in one run, not in many runs. The file array and the directory traversal exist for exactly that purpose.



This slide is the map for the rest of the unit and at the same time the slide people photograph. In the order of effect, not in the order of convenience.
The last two lines are the ones that are not set in FLAM but in the system -- that is the point for the systems programmer.



The mechanics first, then the numbers -- and with FLAM5 the numbers are different from the ones known from the FLAM4 days.
FLAM processes in self-contained segments. Every segment carries an overhead, so the segments should not be too small. If there is too much in one segment, processing constantly runs outside the caches -- and throughput drops again. There is an optimum, and it depends on the platform.
On the **64 KiB**, because the number gets passed around: that is a **historical value** we still use for blocking for backward compatibility reasons. It is not an optimum.
For the FLAM5 archives we ran extensive **measurement series** and realigned the defaults accordingly: **a matrix should carry around 1 MiB**, so that the compressed output ends up at 100 to 200 KiB. That is the size at which compression ratio and cache behaviour meet best.
On the read side FLUC follows the medium: **from an SSD it reads 1 MiB by default, from old disks with 4 KiB sectors 64 KiB.** On the mainframe under MVS it reads block by block, naturally.
The difference is largest in the record count per matrix: **FLAM4 has a fixed record count of 128. FLAM5 makes it dynamic** -- minimum 4, typically 1024, and with many small records more than that.
And now the point that explains why this matters at all.
The processing of the original data follows the system. Archiving, by contrast, runs **completely autonomously and asynchronously**. So one block of original data can lead to several matrices or fill only part of a single matrix -- the two sides are decoupled.
Only one thing matters here: **the number of records per matrix has to be defined so that positioning is possible.** Position divided by the record count gives the matrix index. If the number per matrix is not constant, access to a particular record takes longer, because you have to feel your way towards it.
Hence a design decision that regularly shows up in operations:
deleted records are only marked as deleted. Only when the segments are rewritten are the matrices reorganised. If you wonder why the archive does not get smaller after deleting -- that is the reason, and `PRUNE` is the answer.



In practice this is where most of the potential lies, and it is at the same time the most dangerous short cut. Put both honestly side by side.
The default is MODE=STOP: every character is checked for validity and processing aborts on an invalid or unmappable character.
That costs CPU, and it is the reason why FLAM finds wrongly encoded text data where other tools silently produce garbage. That is a differentiator, not overhead.
Setting SUBSTITUTE makes you faster and loses exactly that detection.
That is a business decision, not a technical one.
Second point: the conversion is faster if all characters of the input alphabet end up with the same length in the output alphabet.
Third point, z/OS only, and it is the most important one for the CPU sum:
the **z/OS Unicode Services**, ZUS for short. We use them as long as they can do the requested conversion and are faster than our implementation.
But there is a condition here that you have to know, because otherwise it silently prevents ZUS from being used at all: **the hardware instructions always substitute and never report an error.** So for ZUS to be usable, `CHRMODE=SUBSTITUTE` has to be set.
And our default is `STOP`, for historical reasons.
Hence our recommendation for customers who want to save CPU on z/OS:
set `CHRMODE` to `SUBSTITUTE` by default -- via the environment variable or the system symbol, not in every single call.
Then ZUS can always take effect.
And the honest flip side: if you need **reporting** of all changes, or **transliteration**, or a **mapping**, or the control of a **subset** -- then we cannot use ZUS for you. Then our own converter runs. There are also cases in which a simple `TR` is faster than ZUS; then we take that too, and not the special instructions.
The decision is therefore a business one, not a technical one: error detection versus CPU. We do not make it for the customer.

LOGGING: THE FORGOTTEN ITEM

- By default FLAM writes **many** messages – this costs CPU and I/O
- `QUIET` and `SILENT` reduce the chattiness
- the `MESSAGE` object controls the volume finely – in both directions
- the most frequent case: when moving from test to production, resetting it is forgotten
- `MESSAGE` is the **selection** of the messages, `LOGGING` the **destination** - two separate objects that often get confused

QUIET IN PRODUCTION, SELECTIVELY LOUDER WHEN SOMETHING FAILS

```
//CONV     EXEC PGM=FLCL,PARM='CONV MAXCC=-2 SILENT'
//FLAMPAR  DD *
  read.record(file=DD:IN);
  write.text(file='/daten/aus.txt');
  message(minimal);
  logging.file(name='/pfad/flam.log');
```

`MESSAGE` is a **list of switches** - some are groups (`MINIMAL`, `DEFAULT`, `ERRONLY`), some are individual message types (`INFO`, `WARNING`, `DEBUG`, `NOTICE`). Separately from that, `LOGGING` determines **where** the output is written.

FLAMv5.21-42969

The slide with the best effort-to-benefit ratio of the whole unit, because the change takes five minutes.
By default FLAM writes a lot of messages. That costs CPU and I/O. In the move from test to production, turning the verbosity back down is regularly forgotten -- and then it costs disk space and compute time in production.
Name the other direction too: in an error case you want more, not less.
The MESSAGE object can go both ways.

LITERAL CACHE

- a **global** literal cache saves memory and CPU when comparing character strings
- behind it is a hash table; its size is specified as a **power of two**
- `FL_LITERAL_CACHE_SIZE` or `&FLITCS`, values 8 to 20 – that is 256 to around 1,048,576 entries
- the **debug log** contains the corresponding statistics: you do not have to guess whether the table is too small

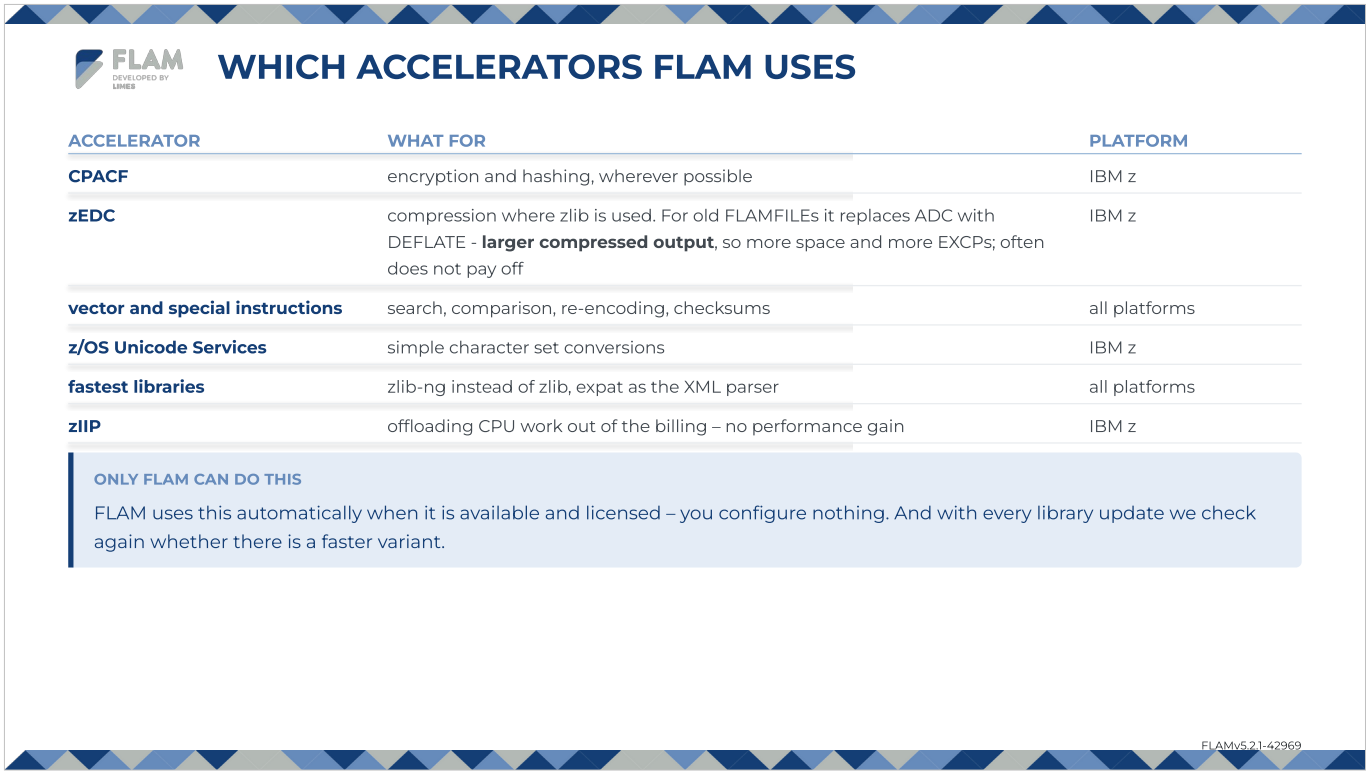
SET IT AND CHECK IT IN THE DEBUG LOG

```
export FL_LITERAL_CACHE_SIZE=16     # 65536 entries
flcl info get.system
```

On z/OS as the system variable `&FLITCS`. The cache lives in the process – it takes effect in one long run, not across many short ones.

FLAMv5.21-42969

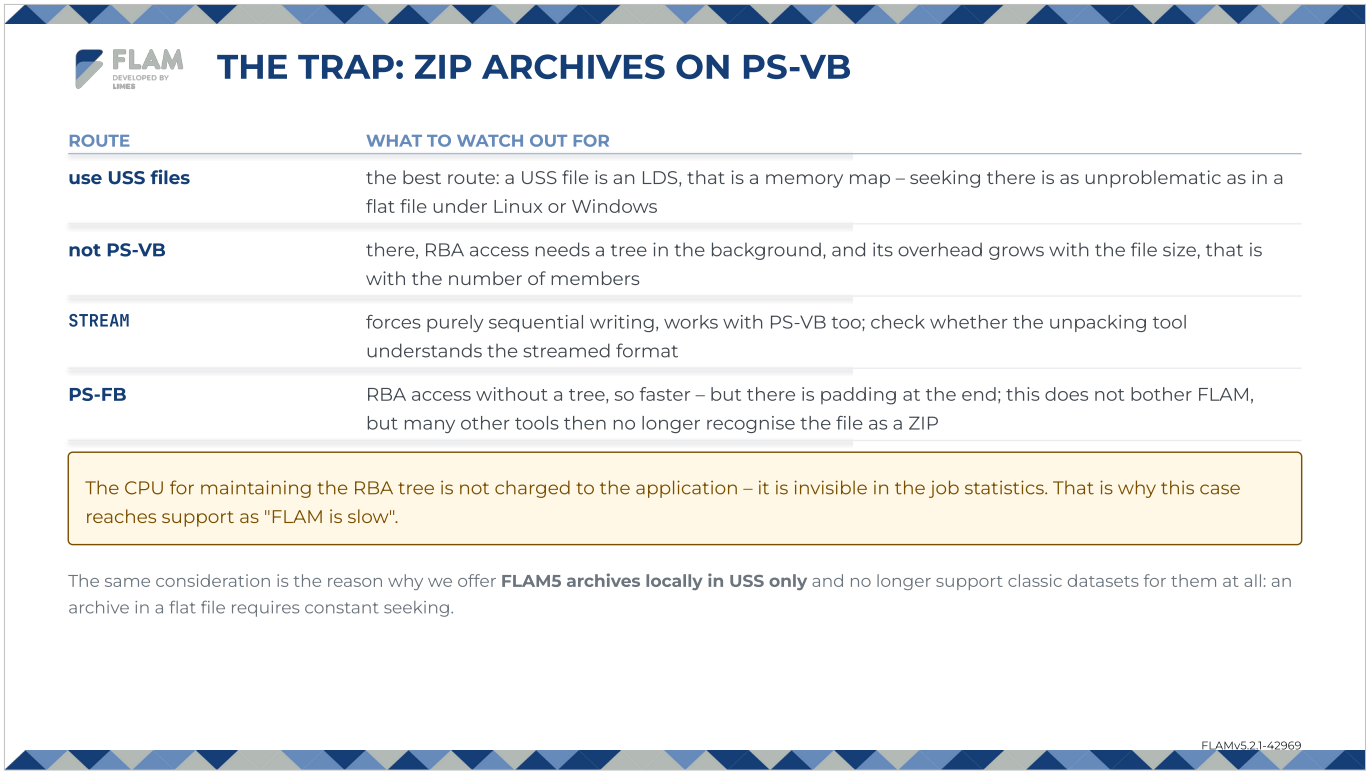
Keep it short -- this is a fine adjustment, not a lever for everyday use.
FLAM uses a global literal cache to save memory and CPU on string comparison. Behind it sits a hash table. Its size is given as a power of two, 8 to 20, which is 256 up to roughly one million entries.
The important sentence: the statistics for it are in the debug log. So you do not have to guess whether the table is too small.



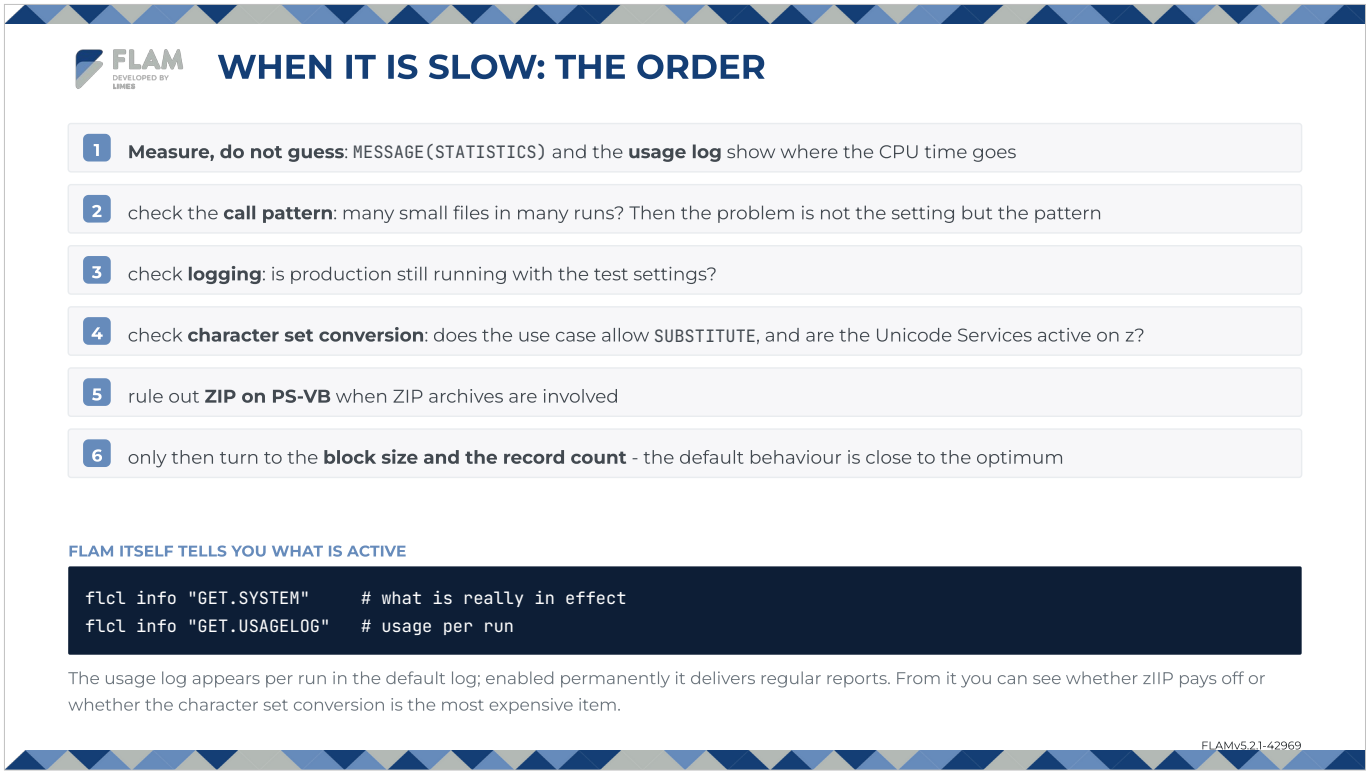
This slide is the point that is deliberately absent from our homepage -- it is a real argument and at the same time an overview you do not get anywhere else.
Important: FLAM uses most of this automatically when it is available and licensed. It is not a configuration task.
CPACF for encryption and hashing, zEDC where zlib is used, special instructions including vector arithmetic on all platforms -- not only on the mainframe.
And the point hardly anyone expects: wherever we can, we link the fastest available libraries. zlib-ng instead of zlib, expat as the XML parser. That is a deliberate decision and it is reviewed at every library update.
One more point on zEDC that pre-empts a question: **zEDC support exists for the old FLAMFILEs as well.** There the ADC is replaced by DEFLATE.
But you have to do the whole sum, otherwise you recommend the wrong thing: DEFLATE produces **considerably larger compressed output** than the ADC. You then no longer pay for the compression in CPU -- but in disk space and in additional EXCPs, that is I/O operations. Across the whole chain that often does **not** pay off.
So: with FLAM5, zEDC is a good thing where zlib works. With the FLAM4 ADC it is a trade-off, and it often comes out against zEDC.
Point to the next unit: zIIP deliberately appears here in one line only, because it is not a performance topic but a billing topic.



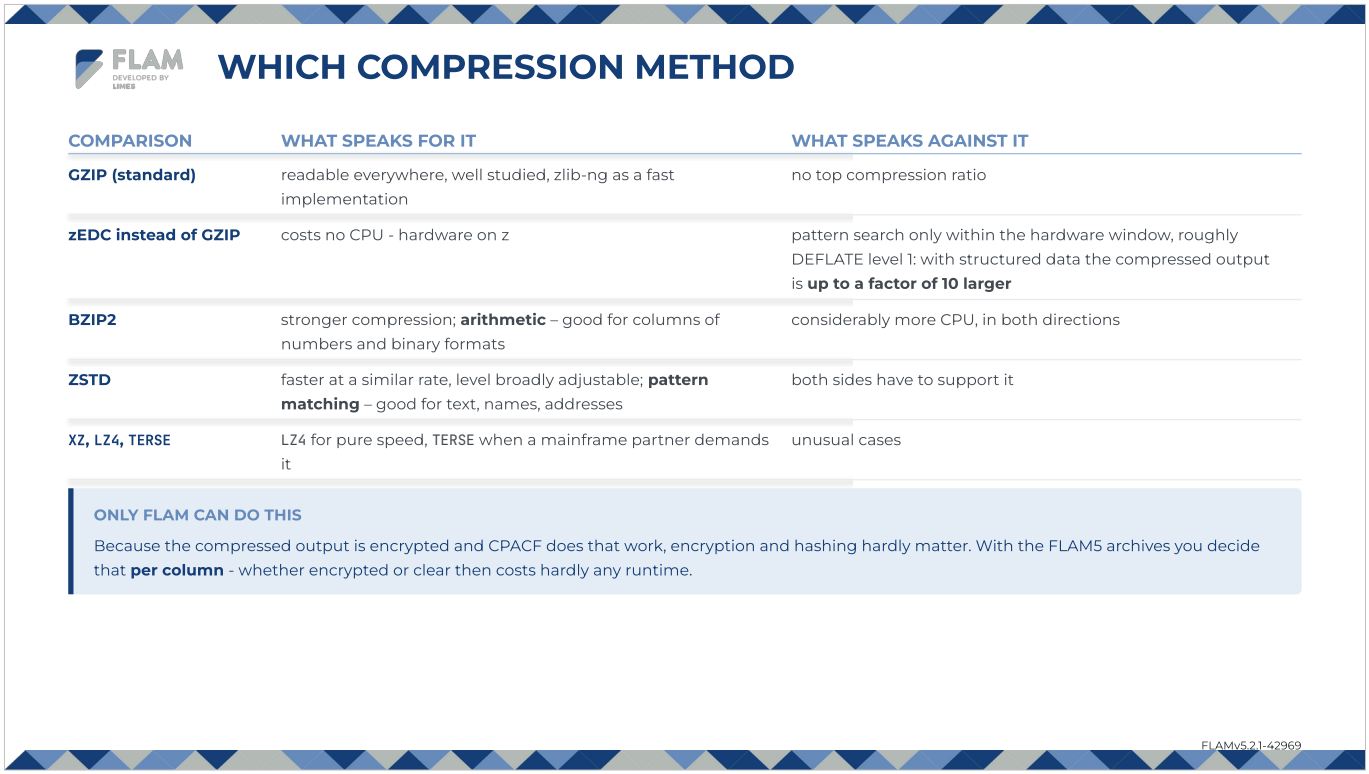
This is the slide that makes the difference from ZIP and GZIP clearest -- and it comes late, because it only works after the mechanics.
First point: the table support reduces the net data before anything is compressed at all. With XML usually by a factor of 3, because the structure is described once and not repeated for every row. That is exactly what a ZIP cannot do in principle: it sees a byte stream, no columns.
Second point: segment-wise access. If you search an archive for a transaction, you do not unpack the archive but the segments in which the hits lie. In practice that is often 1 to 5 segments of 128 records each -- instead of hundreds of thousands. That is the biggest saving there is here, and with ZIP or GZIP it is fundamentally unreachable.
Do not name a customer, and not the daily volumes either. The mechanics are enough.
Third point: the numbers we state publicly. 100 MiB down to 2 to 10 MiB, with SEPA XML up to 98 per cent. That is on the homepage and is therefore quotable.



Very concrete, very mainframe-specific, and a case that regularly reaches support as "FLAM is slow".
The mechanism: when a ZIP archive is created, offsets and statistics values have to be written into the member headers at the end. That requires repositioning within the file. Under z/OS this goes via RBA access. In a PS-VB file the operating system maintains a tree in the background for that, which remembers which relative byte address lies in which block, in which record, at which offset. That is extremely expensive, and with many members the runtime rises considerably. Maintaining this tree also costs CPU that is not charged to the application -- so you do not see it in the job statistics.
The three ways out, in this order: use a file type suitable for RBA access -- and that means **USS files** --, or use the STREAM keyword, or PS-FB.
Why USS files: the zFS is physically mapped into a VSAM LDS. An LDS is the precursor of what is called a memory map today -- a flat, unstructured address space accessed with offsets. That is why seeking is cheap there, and that is why it is exactly the flavour of flat file we need. The file itself is not an LDS, the file system underneath it is one -- experience says somebody will ask about that difference.
For STREAM and for PS-FB, name the respective limitation honestly -- that is the reason why the slide is credible.



The practical slide. Proceed in this order, not in the order of assumptions.
Point 1 first, because it is the most frequent cause and because the answer is then: that is the design, change the call pattern.
Point 2, because logging is changed over in five minutes.
Point 6 is the one you most want to do first and are best off doing last.
The two calls at the bottom belong before every measurement, for one reason:
they separate assumption from fact. `INFO GET.SYSTEM` shows the effective environment variables -- that is, whether `SUBSTITUTE` and the Unicode Services really are set and not merely supposed to be. `LICENSE` shows the active features, among them the accelerators: if you expect CPACF and have no licence for it, you measure for nothing.
The most important advice comes first, which is why it is point one:
measure, do not guess. The tool of choice for that is the **usage log** together with the statistics information. You can look at it per run in the default log, or activate usage logging permanently and pull regular reports.
Then you can see in black and white whether zIIP would be worth it, or whether character set conversion is the most expensive item -- and you do not have to argue about it.
And a sentence for managing expectations that we can say with a clear conscience, because it is true: **our default behaviour is close to the optimum.** Only in special cases do you have to adjust block sizes and record counts -- particularly with very large files. Anyone turning the screws on every job is wasting their time.
One more tool for the special case, because it is barely known: the **force switches**. By default we preserve record lengths and read transparently. But if you know from the outset that this buys nothing -- for instance because a record-oriented mainframe data set has to be read on delimiters anyway and the record lengths do not matter -- then `FRCBLK` switches off the handling of record lengths and forces block orientation. You thereby give up the universality of transparent reading and gain speed.



The question comes up in every training, and it has no answer in a single number -- but it has an order in which to answer it. The four main contenders are: **GZIP** is the standard and the right answer as long as nothing argues against it. Readable everywhere, well studied, and we link zlib-ng instead of zlib, that is the faster implementation. **GZIP versus zEDC**: on z/OS zEDC does the same method in hardware. That costs no CPU. But the hardware looks for patterns only in a limited window -- roughly DEFLATE level 1. With structured data the compressed result can therefore be **up to a factor of 10 bigger** than with our table support.
You trade CPU for space and I/O. **GZIP versus BZIP2**: BZIP2 compresses more strongly but costs considerably more CPU -- and in both directions. For an archive written once and read rarely that can be right. For a daily exchange almost never. And now the difference that makes the choice a **business** one instead of a gut feeling, because the methods do not work on the same principle: **BZIP2 is an arithmetic method, GZIP and ZSTD do pattern matching.** From that follows a rule worth remembering:
BZIP2 is good for columns of numbers and binary formats -- amounts, measured values, integer and float fields. **Pattern matching is good for text** -- names, addresses, payment references. We do exactly that in our own archives: the **length fields** we compress with BZIP2, the **data** with pattern matching. That is also the bridge to the pot design: once columns are put into different pots, the method fitting each pot can be chosen -- amounts arithmetically, texts by pattern. Whoever puts everything into one pot has to decide for one and loses on the other side.
GZIP versus ZSTD: ZSTD is the more modern compromise -- faster at a comparable rate, and broadly adjustable through the level. If both sides can do ZSTD it is mostly the better choice. If the recipient can only do GZIP, the discussion is over. Then the level, and a practical rule for it: **the upper levels cost a lot of CPU for a few percent.** Anyone raising the level should measure beforehand what he gets for it. `XZ` or LZMA, `LZ4` and `TERSE` we support as well, but those are unusual cases -- `LZ4` when only speed matters, `TERSE` when a host partner demands it.
And the point that puts the whole discussion into perspective: **because we usually encrypt the compressed data and the encryption runs over CPACF, encryption and hashing hardly weigh anything.** With the FLAM5 archives there is also the fact that this is decided **per pot and therefore per column**: only what is sensitive gets encrypted. Whether a column is encrypted or kept in the clear then hardly matters for the run time -- the decision is a business one, not a technical one.

EXERCISE: DETERMINE A BLOCK SIZE

Individually or in pairs, 10 minutes:

1. Your records are 200 bytes long on average. Which record count belongs to a block size of 256 KiB?
2. What happens if you increase the block size but leave the record count at 512?
3. Take a job of your own and estimate which of the six points from slide 13 would apply to it

THE CALCULATION AS A CALL

```
flcl conv "read.record(file='<SYSUID>.EIN')
           write.fl5(file='<SYSUID>.ARCHIV.FL5'
                     fmt.blk(blksiz=256k reccnt=1300))" \
     SILENT MAXCC=-2
```

Measure, do not estimate: run the same job with the defaults and with your values and compare the CPU times.

FLAMv5.21-42969

10 minutes. A calculation, deliberately simple, so that the formula sticks.
The solution: 256 KiB divided by 200 bytes is around 1300 records. If you leave 512 in place, the segment is only filled to a good third -- close to two thirds stay unused.
The second part is about them measuring the effect themselves -- not about the number being right.

WHAT TO TAKE AWAY

- **Preload**: expensive in the open, cheap in the run – hence constant cost per MiB with bulk data
- **many small files** are the one case in which this works against you – then change the call pattern, not the setting
- the most effective lever is **less data**: table support and segment-wise access
- the two most frequent CPU items are **character set conversion** and **logging**
- FLAM uses **accelerators** automatically – `flcl LICENSE` shows which ones are licensed

Next unit: zIIP, zEDC and CPACF working together – and why zIIP is not a performance topic but a billing topic.

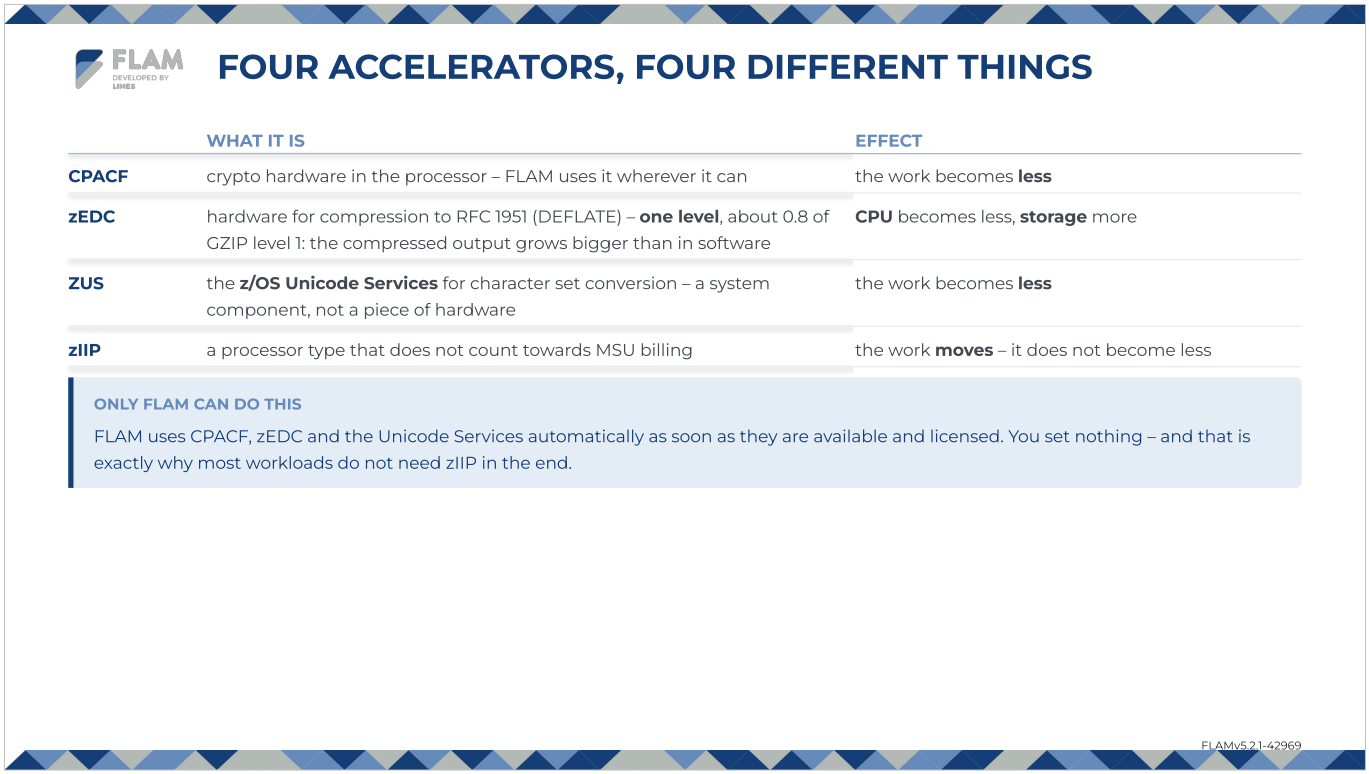
The first and the last point are the ones that should stick.
Transition to the next unit: it is about zIIP, and the first sentence there is that zIIP is not a performance topic.



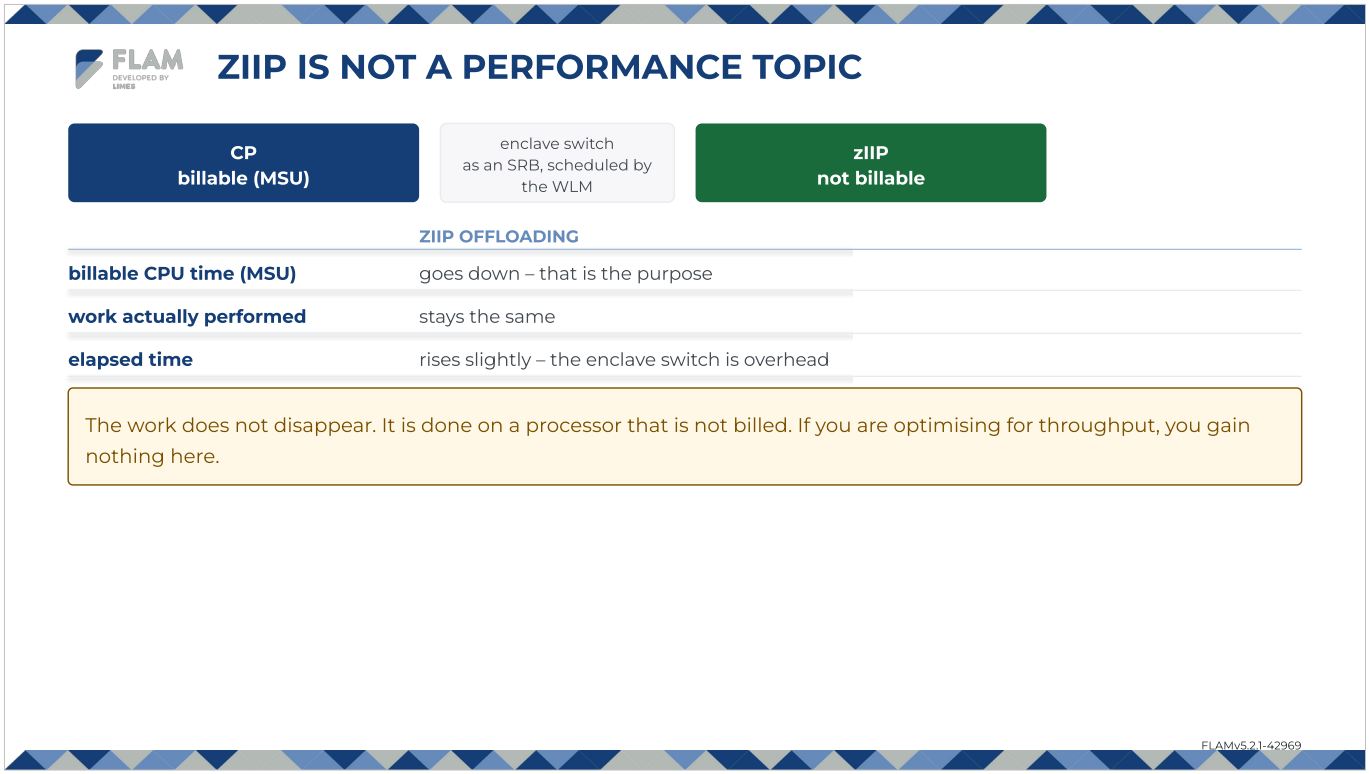
Last unit on Thursday. This is the unit that answers the specific question from the customer -- which is why it is deliberately kept general and without customer names. Tell the specific case verbally.
The first sentence of the unit has to land: zIIP is not a performance topic. After that the conversation is a different one.



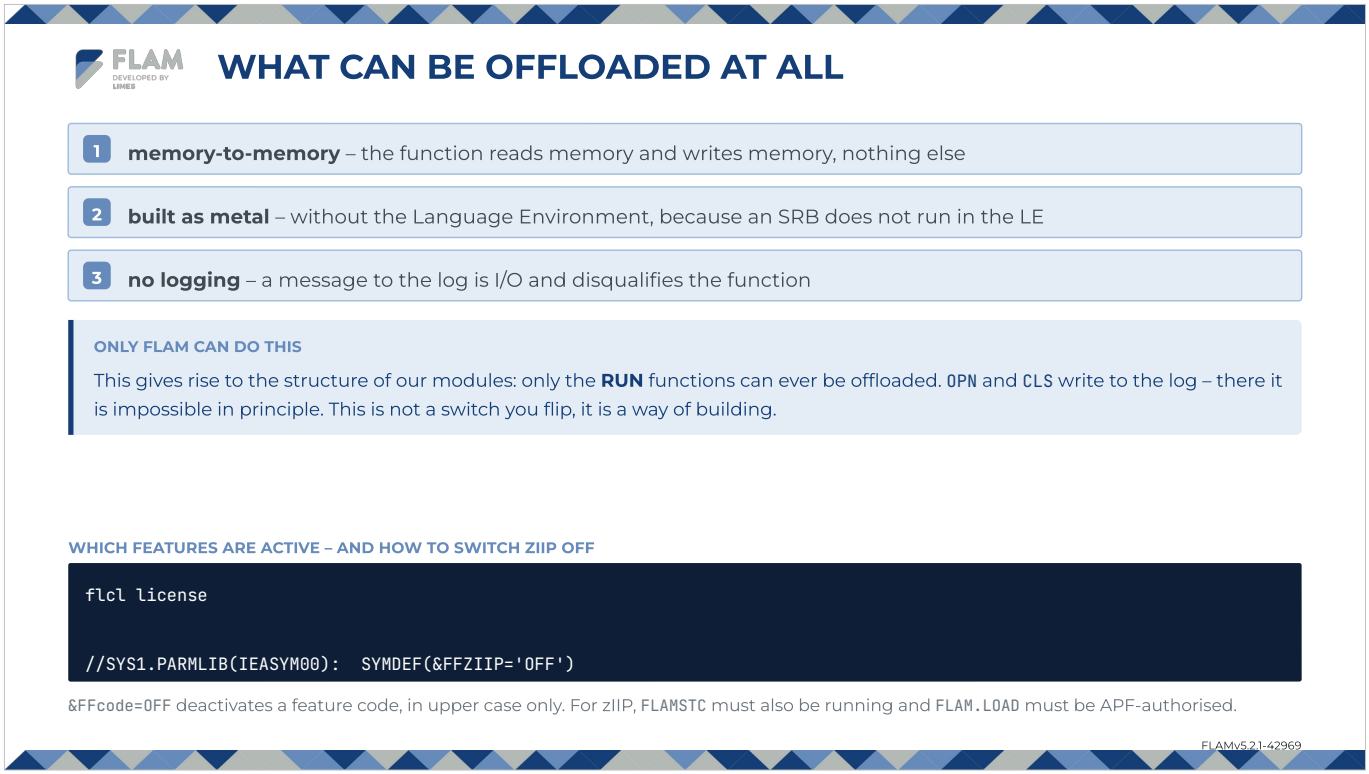
Objective 1 is the core message, objective 4 is the answer to the question the participants came with, and objective 5 is what they can then do themselves.



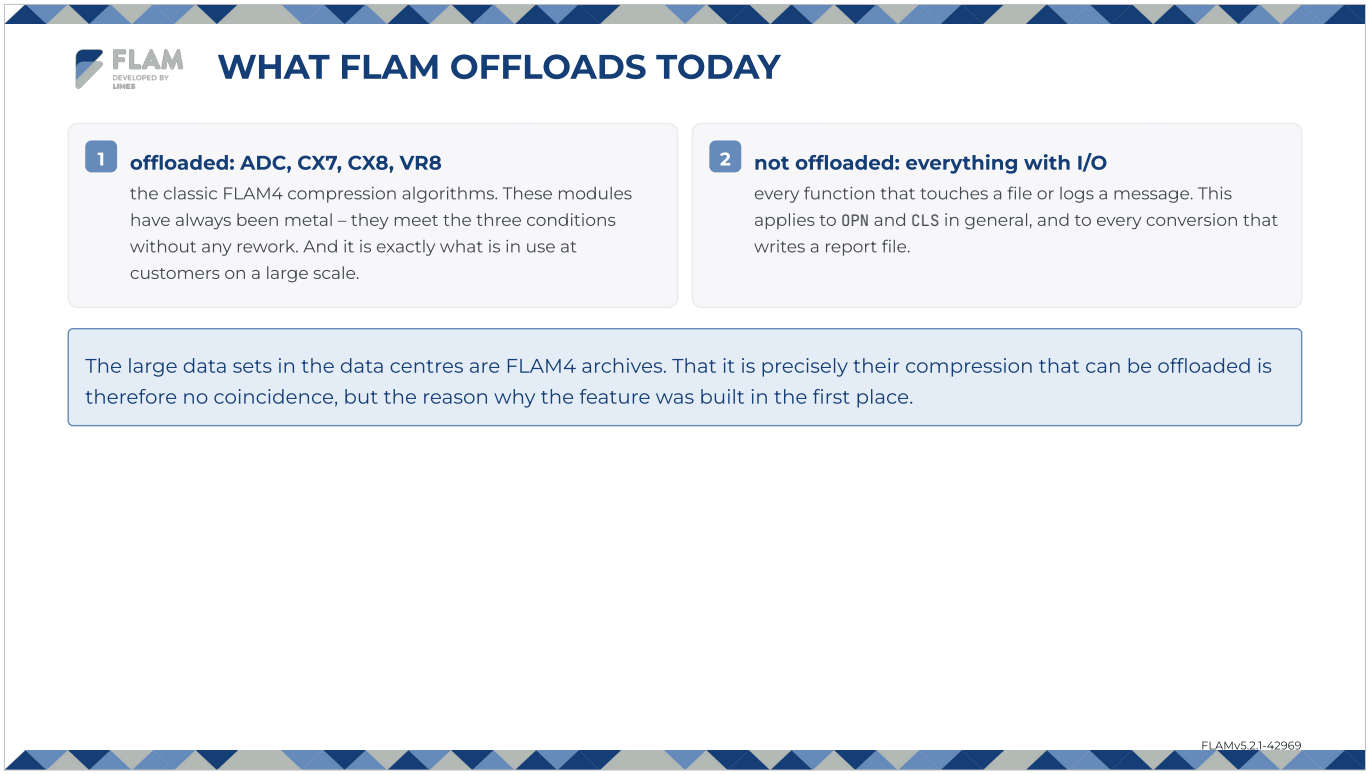
This slide sorts out the terms that regularly get mixed up in conversation -- and the muddle is the real cause of most misunderstandings on this topic. CPACF, zEDC and the z/OS Unicode Services are genuine accelerators: they make the work faster and cheaper. zIIP is not an accelerator but a different processor type which does not enter the MSU billing. The last column is the important one: with the first three the work goes down, with zIIP it only moves. The **z/OS Unicode Services** are the one most easily overlooked, and on a host with a lot of character conversion they are the most effective: FLAM uses them for simple conversions instead of its own implementation. That is not a hardware accelerator but a system component -- controlled through `FL_IBM_UNICODE_SERVICE` or the system symbol `&FLIBMUS`. The details and the differences in behaviour are in the unit *Character sets*; all that belongs here is that they exist and that they count. CPACF and zEDC depend on the accelerator feature of the licence -- that is the point slide 10 comes back to. One restriction belongs with the zEDC straight away, so that nobody takes it for free: **the card compresses worse than software.** It is designed to reach GZIP level 1 and actually lies at about 0.8 of that; the level cannot be influenced. ZSTD compresses about twice as well as DEFLATE in software, ADC with structured data even better -- so whoever switches to zEDC may need **considerably more disk space or more tapes** than with GZIP in software, ZSTD or ADC. The effect is real and it is paid for: CPU is saved, storage and bandwidth are spent.



The core slide. If only one slide of this unit sticks, let it be this one.
The mechanism: a memory-to-memory function is dispatched as an SRB to a zIIP. The Workload Manager schedules it. That costs an enclave switch, and this switch is overhead -- not a gain.
So the sum is: the chargeable CPU time goes down, the elapsed time rises slightly. If you optimise for throughput you gain nothing here. If you optimise for the bill, you gain.
This is not a criticism of zIIP. It is the precondition for asking the right question: what am I actually trying to optimise?
Important against a frequent misunderstanding: the work does not disappear.
It is done on a processor that is not billed.



Now the technical limit, and it is hard. Three conditions, all three of which have to be met.
First, memory-to-memory: the function may only read memory and write memory. No file access, no data set access.
Second, metal, that is without the Language Environment. An SRB does not run in LE.
Third, no logging. A function that writes a message to the log does I/O -- and with that it is disqualified.
Hence the sentence that explains the structure: only the RUN functions can ever be offloaded. OPN and CLS write to the log, so there it is impossible in principle.
That is also the reason why this is not a switch you flip: the modules have to be built for it.



Here comes the point that often surprises: what gets offloaded is the old stuff.
ADC, CX7, CX8 and VR8 are the classic FLAM4 compression algorithms.
These modules have always been metal -- which is why they meet the three conditions from slide 5 without anything having to be rebuilt.
And that is not a footnote: exactly these methods are what is in use at customer sites on a large scale. The big data sets are FLAM4 archives.
The second half of the slide is the honest statement of what is not offloaded and why. Do not phrase it as a shortcoming -- as a consequence of slide 5.

FLAM DEVELOPED BY LIMES

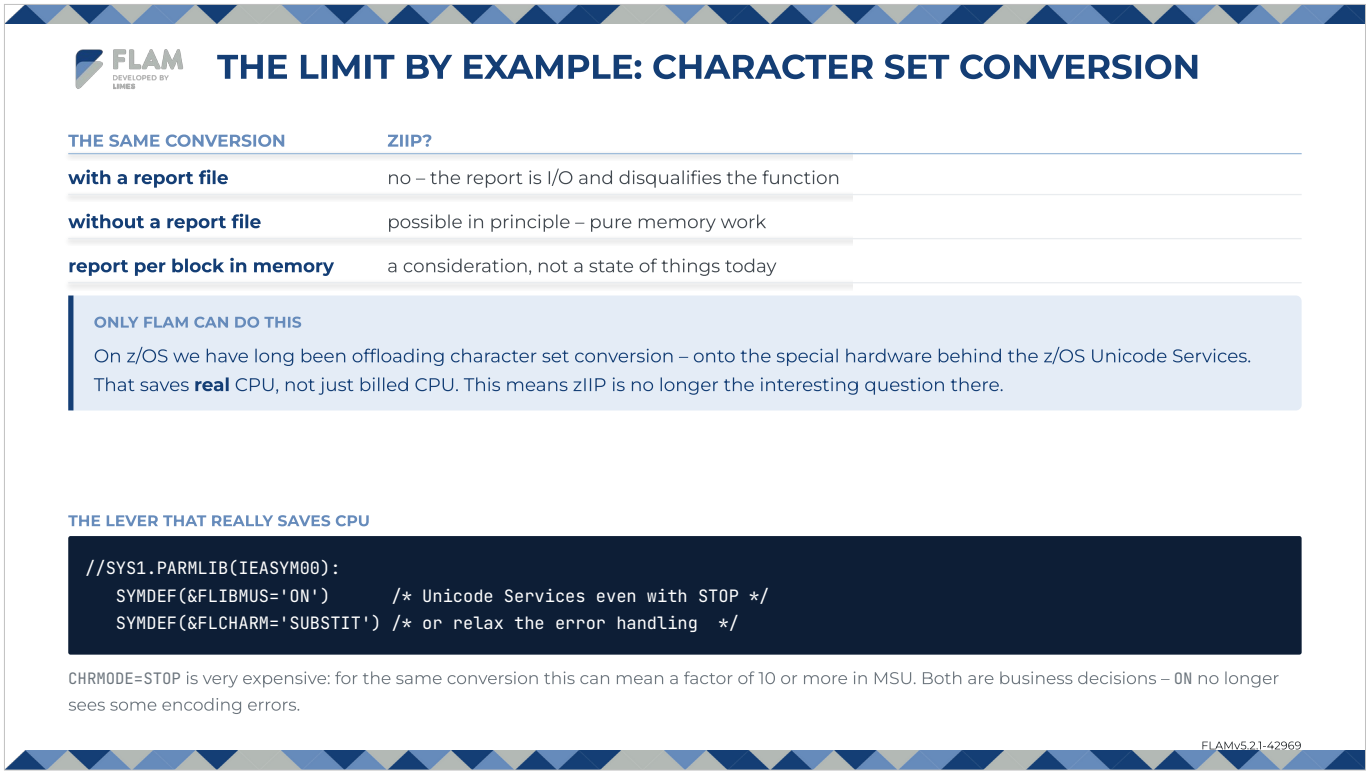
WHAT WOULD ADDITIONALLY BE POSSIBLE

CANDIDATE	WHY IT IS A CANDIDATE
CNV modules	the conversion stages work memory-to-memory – character set conversion, compression, encryption
FMT modules	formatting and record preparation, likewise pure memory work
archive except FLMSEG	only `FLMSEG` really reads and writes the segments; everything above it computes in memory
ZSTD and modern compressors	the same design as ADC, only newer
table support (FMTTAB)	column-wise preparation, compute-intensive and without I/O

An outlook, not a commitment: these modules would have to be built as metal for that – and even then it still holds that only the `RUN` functions are candidates.

FLAMv5.21-42969

This slide is an outlook and is to be marked as such. Do not phrase it as a commitment.
The idea: all of our CNV and FMT modules work memory-to-memory. In the archive that holds for everything except FLMSEG, because FLMSEG actually reads and writes the segments. These modules could be built as metal and thus be offloaded.
Concretely that would be character set conversion, ZSTD and the other modern compressors, and the table support.
Always with the same restriction: only the RUN functions.
If anyone asks why it is not already like that: it is effort, and the benefit depends on whether there is any free zIIP capacity at the customer at all. That is the bridge to slide 11.



The concrete example in which the condition from slide 5 becomes visible -- and it is a good example, because it shows that the limit does not depend on the module but on the configuration. The very same character set conversion is offloadable or not, depending on whether a report file is written. With a report: I/O, so no zIIP. Without a report: pure memory work, so possible in principle. The way out one could build: hold the report per block in memory and write it out only later. Mark that as a consideration too.
And then the point that puts the whole consideration for character set conversion in perspective: on z/OS we already offload it -- namely to the special hardware behind the Unicode Services. That saves real CPU, not just billed CPU. So zIIP is no longer the interesting question there at all.
That is the transition to the next slide.

FLAM DEVELOPED BY LIMES

WHERE HARDWARE MAKES ZIIP UNNECESSARY

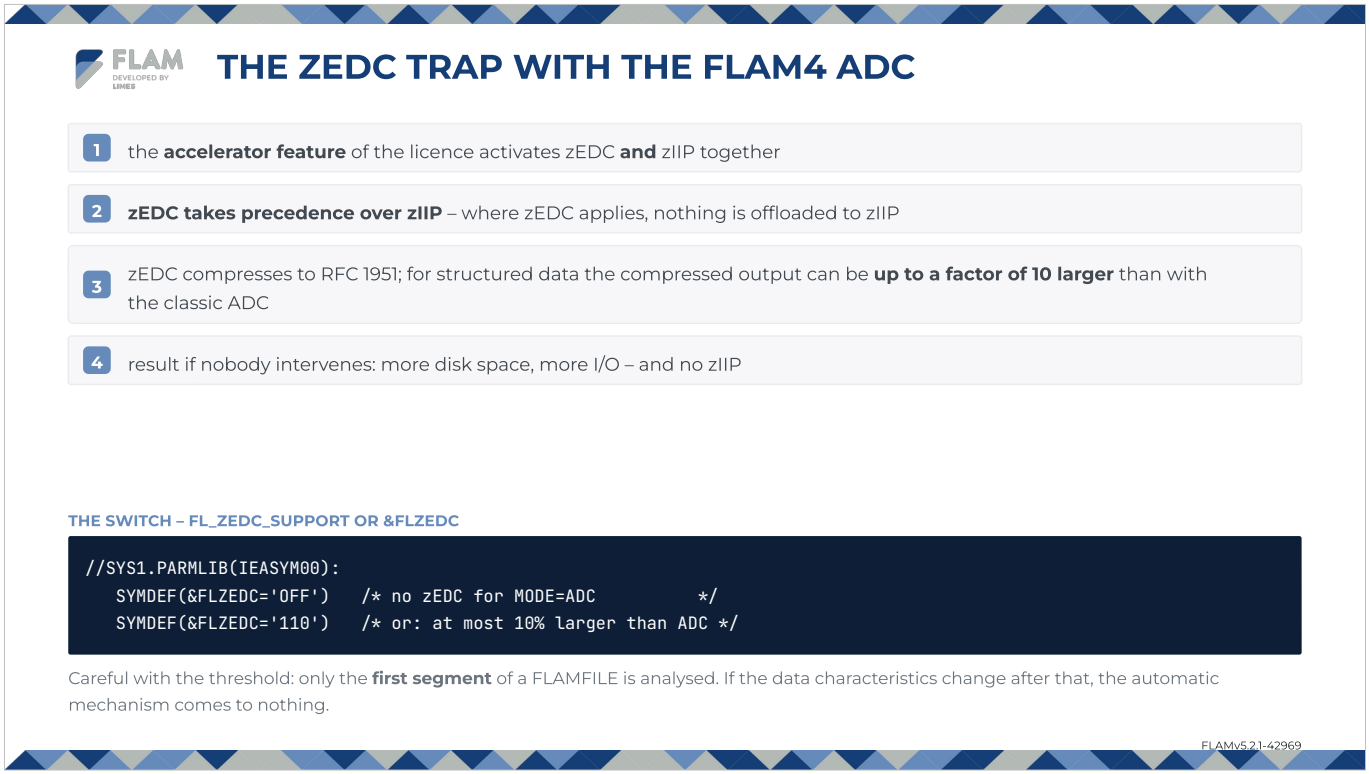
AREA	WHAT ALREADY RUNS IN HARDWARE	ZIIP FOR IT?
encryption and hashing	**CPACF**, wherever it is possible	not needed
character set conversion	**z/OS Unicode Services** on special hardware	not needed
compression to RFC 1951	**zEDC** – in FLAM5 automatically wherever zlib works	not needed
classic FLAM4 compression	no hardware support	**yes, that is what it is there for**

ONLY FLAM CAN DO THIS

The order is hardware first, zIIP last. zIIP is the answer for what has no hardware support – not the first thing to reach for.

FLAMv5.21-42969

The summary of the previous thought, now for all three areas.
Encryption: we always take CPACF when it is possible. So there is no reason to hand encryption to zIIP.
Character set conversion: the Unicode Services use the special hardware.
Compression: zEDC does RFC 1951 in hardware; in FLAM5 we use that automatically wherever zlib is used.
The sentence that should remain: the order is hardware first, zIIP last. zIIP is the answer for whatever has no hardware support.



This is the slide that answers the actual question -- and it answers it with a mechanism, not with an opinion.
The flow: the accelerator feature of the licence activates zEDC and zIIP together. In FLAM5 zEDC is unproblematic, there it is used wherever zlib works. For FLAM4, customers asked for zEDC transparently for the ADC -- and that is where the problem begins.
zEDC is preferred over zIIP. And zEDC compresses according to RFC 1951, which for structured data is considerably worse than the classic ADC: the compressed output can become up to a factor of 10 larger. So anyone who does not switch this off has more disk space, more I/O and no zIIP -- and then "odd effects" that nobody can attribute.
Leave the solution up and read it out slowly: FL_ZEDC_SUPPORT or &FLZEDC.
OFF, or a threshold in per cent.
Be sure to name the limitation of the threshold, otherwise the recommendation is dangerous: only the first segment of a FLAMFILE is analysed. If the data characteristics change after that, the automatic mechanism misses.

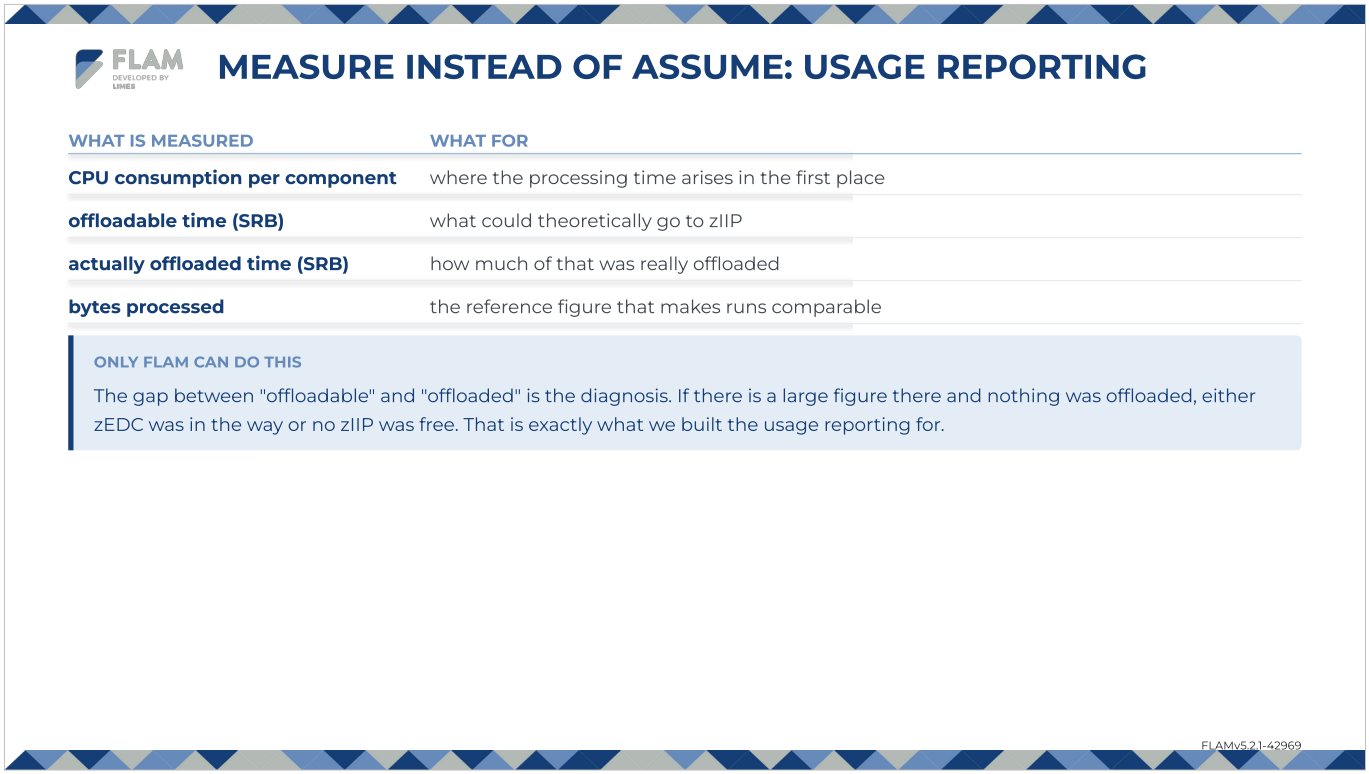
AND EVEN THEN: IS A ZIIP FREE?

- offloading needs **free zIIP capacity** – the WLM schedules the SRB
- if no zIIP is available, the work runs on the CP – that is, exactly where it would have run anyway
- on today's mainframes zIIP capacity is often scarce because **Java** occupies it
- this is not a FLAM question and not a configuration question, but a **capacity question**

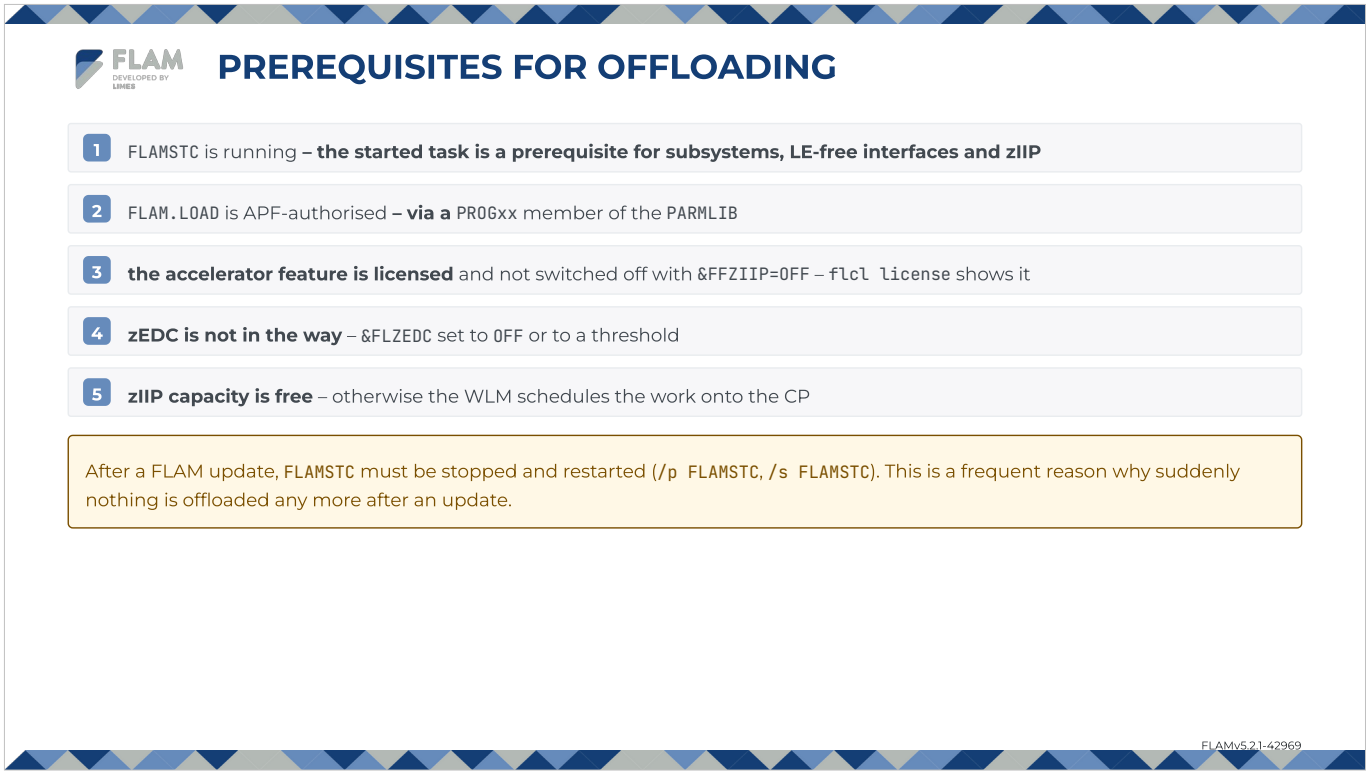
The honest answer to "why is there no zIIP offloading at our site?" is almost always: because in that setup it would not have gained anything. That can be measured instead of assumed.

FLAMv5.21-42969

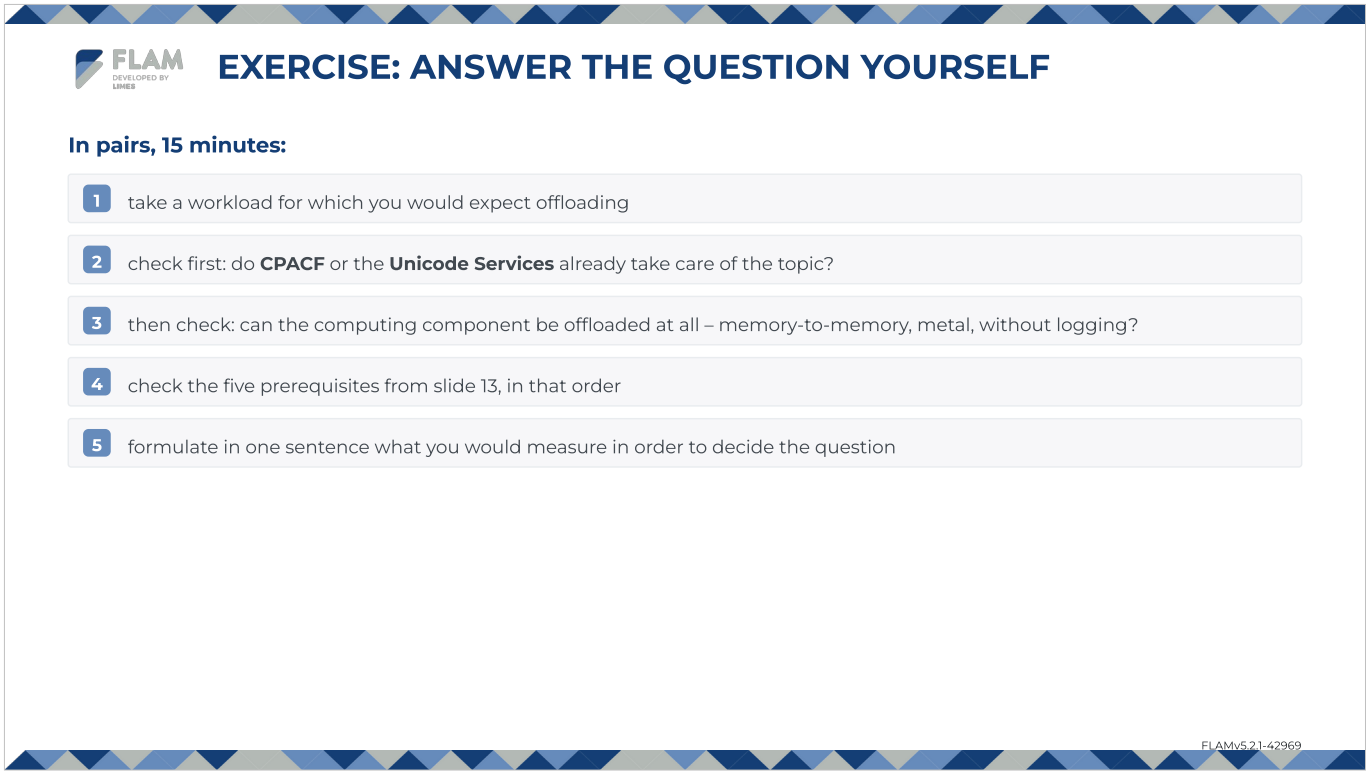
The last building block of the answer, and the one you cannot set in FLAM.
An offload needs free zIIP capacity. The Workload Manager schedules the SRB; if no zIIP is available, the work runs on the CP -- that is, exactly where it would have run anyway.
On today's mainframes zIIP capacity is often scarce because Java occupies it. That is not a FLAM question and not a configuration question, but a capacity question for the capacity planner.
That is why the honest answer to "why no zIIP with us" almost always ends with: because in this constellation it would not have gained anything. And that is why the slide on measuring comes now.



The constructive slide, and the reason why this unit does not end with an "it depends".
Since 5.1.13 FLAM measures CPU consumption per component and logs three values for it: the time that is offloadable as an SRB, the time actually offloaded as an SRB, and the bytes processed. That is exactly the answer to the question. The first number says what would theoretically be possible. The second says what actually happened.
The gap between the two is the diagnosis -- and in that gap sits either zEDC or an occupied zIIP.
Name the effort honestly: it needs the system variables FLOGQL1 to 3, a log stream for the SYSPLEX or the LPAR, a running started task LOGR and the matching RACF settings. That is a job for the systems programmer, not a trifle.
But it is the only basis on which this discussion can be held with numbers instead of assumptions. That is exactly what we built it for.



The checklist to take away. If you expect an offload and see none, work through these five points from top to bottom.
Points 1 and 2 are installation questions, point 3 is the licence, point 4 is the zEDC trap from slide 10, point 5 is the capacity question from slide 11.
After changing FLAM, FLAMSTC has to be stopped and restarted -- that is a frequent reason why nothing is offloaded any more after an update.



15 minutes, pairs. The exercise is deliberately the original question -- so that the participants notice they can now answer it themselves.
When going through the solution, make sure the order is right: first clarify whether anything is offloadable at all, then whether it is licensed, then whether zEDC is in the way, then whether capacity is free. And right at the start, the question of whether CPACF or ZUS has not already settled the topic.

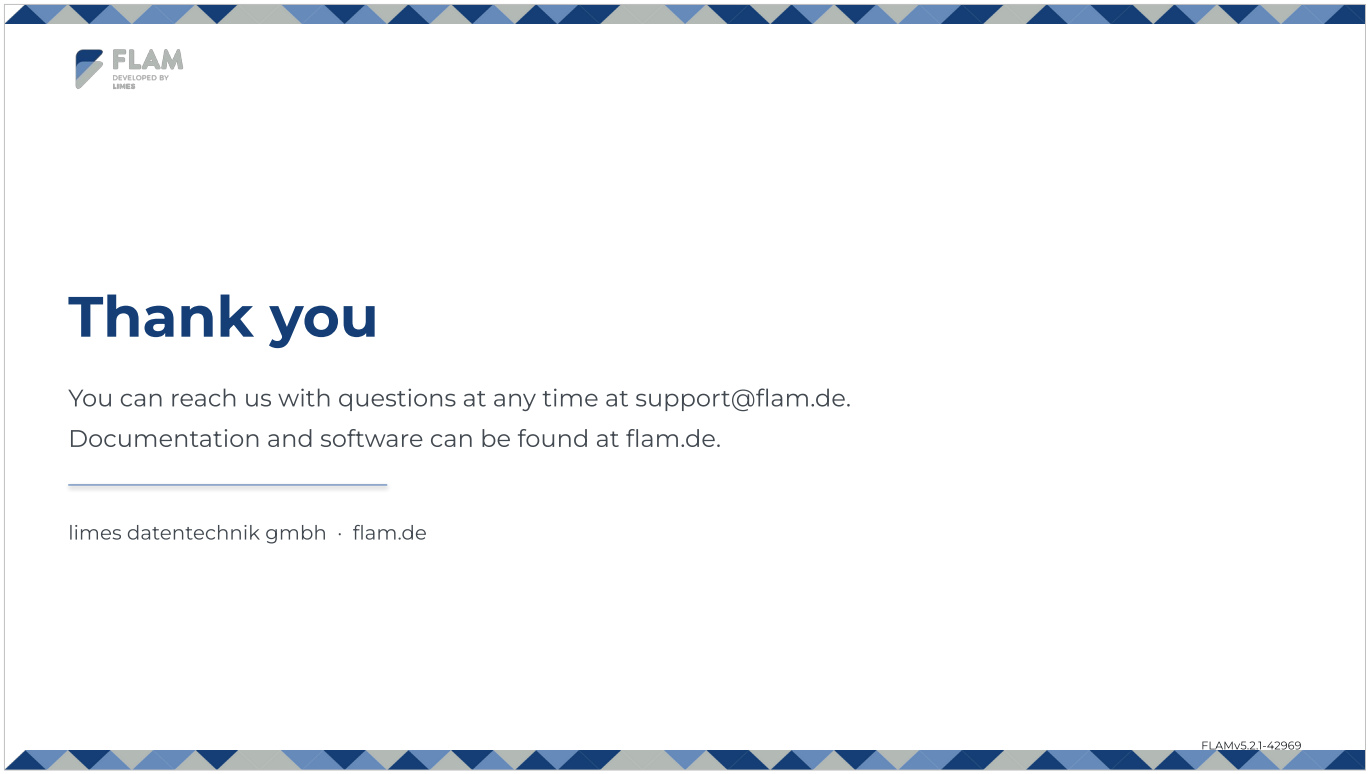
WHAT TO TAKE AWAY

- **zIIP offloads, it does not accelerate** – the MSU go down, the elapsed time rises slightly
- only what works **memory-to-memory**, **metal** and **without logging** can be offloaded – that is, the `RUN` functions
- today these are the classic FLAM4 algorithms **ADC, CX7, CX8, VR8** – exactly what is in use on a large scale
- where **CPACF**, **zEDC** or the **z/OS Unicode Services (ZUS)** take effect, zIIP becomes unnecessary – there the real work goes down
- the most frequent cause of "nothing is being offloaded" is **zEDC before zIIP** or an occupied zIIP

> **ONLY FLAM CAN DO THIS**
>
> And the sentence for the discussion in your own organisation: this can be measured. The **usage reporting** says what was offloadable and what was offloaded – the gap between the two is the answer.

FLAMv5.21-42969

The first and the last point are the ones that should stick. That brings Thursday to an end.



Closing.
Support also answers questions that only come up weeks later during implementation – that is what it is there for, and you are already paying for it. If you come across a case in your environment that we have not covered here, send it to us: as a rule it becomes an example for the next training.