Using AI to port, develop, debug and test VMS applications

On OpenVMS x86, with an AI agent at the keyboard

Eight products, AI-assisted

Four applications ported

Debugging and testing on t86

VSI OpenVMS Boot Camp 2026 | Sector7 | 110 minutes

Overview of the talk

Six parts, 110 minutes, every claim backed by a log, a commit or a screen capture

1	**Products Sector7 has created with AI assistance**	VX/EDT, VX/TPU, VX/SQL, VX/RDML, VX/CDD, VX/TUXEDO, VX/DECFORMS for TDMS, ACMS for Linux
2	**Applications ported, or proof-of-concept ports**	four codebases: what they are made of, and the steps the AI took to analyse, build and package them
3	**Using AI to create an EDT editor**	one prompt, a Ralph loop, and a real VMS host to measure against
4	**Using AI to port AVERTZ to x86 OpenVMS**	HP's ACMS verification program: COBOL, embedded SQL, DECforms, CDD, DECdtm, rebuilt on x86 against PostgreSQL
5	**Using AI to debug an application**	from a line in a log to a fix with evidence before and after
6	**Using AI to test an application**	batch matrices, screen robots, PTY drivers, lockstep against a real system, gates that must fail first

How the AI talks to OpenVMS: ssh, scp and sftp

The agent runs on a Mac or Linux PC; the VMS host is just another machine it can shell into

```
$ ssh t86 'WRITE SYS$OUTPUT "hello-t86"'
 Welcome to OpenVMS (TM) x86_64 Operating System, Version V9.2-3
hello-t86

$ scp BUILD_SQLPRE_NOW.COM t86:/DKA200/JONPOWER/BUILD_SQLPRE_NOW.COM
$ ssh t86 '@DKA200:[JONPOWER]BUILD_SQLPRE_NOW.COM'

$ printf '%s\n' 'get /DKA200/CDD/CDD_LOGGING.TXT /tmp/cdd_logging.txt' 'quit' | sftp
-b - cdd@t86

$ ssh t86 'SUBMIT/QUEUE=SYS$BATCH/LOG=DKA200:[APP.LOGS]JOB_017.LOG -
    DKA200:[APP.SHELLS]JOB_017.COM'
```

Real commands from the session transcripts (July to October 2026); the batch example is the shape used for every job in Part 6

- One DCL command per ssh call. Anything longer is written as a .COM file, copied up, and run with @
- The ssh server presents the disk as a POSIX path: /DKA200/JONPOWER/... maps to DKA200:[JONPOWER]
- Record format matters: IFDL and COBOL sources go up as STMLF; an application refused fixed-length data uploads, VAR/CR through CONVERT worked
- Long jobs go to SYS$BATCH; the agent polls, then fetches the log with TYPE, never through a busy ssh multiplexer
- Key-based login only. The agent never sees a password; it reads the ssh config for the host alias
- Every command and its output is kept in the transcript, so a claim can be traced to the line that proves it

The host does not know an AI is driving it. It sees an interactive user with a slow typing speed and a perfect memory for what it typed last week.

How the agent reports back: a session report, verbatim

After a day on an attendee's two COBOL programs: what ran, what changed, how secrets were handled, what was wrong in our tool

```
What runs (work copies in DKA200:[...], your originals in [...] untouched)
  CUSTSQL_T86.EXE      your CUSTSQL.SCO through VX/SQL, VSI COBOL, link
  CUSTINQ_RDML.EXE     your CUSTINQ.RCO through VX/RDML (--targetdb=rdb, emits
                       embedded SQL) then the same VX/SQL chain
  CUSTINQ_LIBCALL.EXE  your CUSTINQ.RCO through VX/RDML --mode=libcall
                       --targetdb=postgres (COBOL with direct runtime calls), VSI COBOL, link
  All three pass Q 1, L BELL, U 1 with a new memo, Q 1 again, X; each program read
  back the memo the previous one committed, and psql on rh8x saw it too.

What changed in your sources: one line in each working copy, the COPY ... FROM
  DICTIONARY path, from the Integrity CDD path to AVERTZ_CDD_WKSP:VR_CUSTOMERS_WKSP.
  Nothing else: DECLARE ... ALIAS, the transactions, cursors and UPDATE ... WHERE
  CURRENT OF are as you wrote them.

How the database is reached: ... the logical AVERTZ_DB as pgsql://host:port/avertz_demo
  (host and port taken from the protected service file, never displayed), plus PGUSER
  and PGPASSFILE so libpq supplies the user and password; nothing secret is written anywhere.

Two VX/SQL defects your programs exposed, fixed in the vxrdb repo and rebuilt on t86
  1. DECLARE TRANSACTION among the data declarations had its run-time calls emitted
     inside WORKING-STORAGE (%COBOL-F-SYN17); now written after the first paragraph
     of the PROCEDURE DIVISION (1e70d80e6).
  2. UPDATE ... WHERE CURRENT OF carried the file-qualified cursor name, which
     PostgreSQL rejected; it now names the cursor literal the runtime registers (4c0effd3b).
```

the agent's own end-of-session report, 12 October 2026; account and directory names replaced by [...]

What ran, where

three executables, the toolchain each went through, one scripted session all three pass, and the originals untouched

What changed

one line per working copy, named exactly, and the statement that everything else is as the author wrote it

How secrets were handled

host and port from a protected file, never displayed; credentials supplied by libpq; nothing secret written anywhere. Reported without being asked

What was wrong in our tool

two precompiler defects the customer's programs exposed, each with its fix commit. The report does not hide them

Every claim names a file, a commit or a host. That is the shape of the interface: not a chat, a report you can check.

What the model knows, and the rules it works under

Broad and usually right about the shape, wrong about details until it measures; six clauses from the rules file keep it honest

Knows from training

- DCL, lexical functions, logical names, command procedures
- RMS: FAB, RAB, XAB, record formats, indexed files, FDL
- SYS$, LIB$, STR$, SMG$ families and the condition value format
- VAX BASIC, COBOL, FORTRAN, Pascal, MACRO-32, C and their VMS extensions
- Rdb SQL, RDML, CDD, ACMS, DECforms, TDMS and FMS at the concept level

Does not know, or knows wrongly

- Exact qualifier spellings and defaults on a given version
- Item codes, status values, structure layouts: plausible and wrong
- Behaviour only a run shows: what EDT prints after EXIT in a command file
- Your application: it has never seen the source until it reads it
- Its own confidence: a guess sounds like a fact unless told otherwise

Never silently skip

every unhandled input leaves a trace: a counter, a stderr line, a non-zero exit

No stubs, no guessing

an empty body is 'stubbed', never 'done'; claims only after reading the body or running the command

Verify before claiming

a fresh run from clean state, command and output quoted

Measure, do not recall

run it on the real system and keep the capture; no API details from memory

Honest self-appraisal

implemented, stubbed and deferred listed separately; a documented gap is not closed

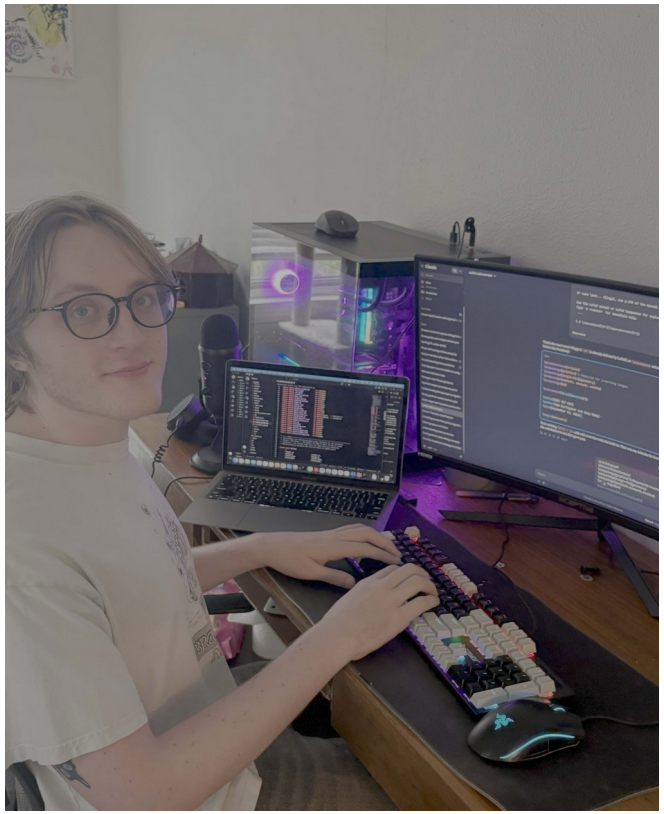
Match the model to the task

mechanical extraction to a smaller model; design, debugging and audit to the larger one

Two remedies, both in this talk: a real host to measure against, and a rules file every session reads first. Each rule exists because it was broken at least once.

Meet Sam: no OpenVMS before May 2026

A physics degree, a master's in AI, and two of the four application ports in this talk



Background

A degree in physics and a master's in AI. No OpenVMS, DCL, RMS, ACMS or DECforms before this year. First commit on an OpenVMS project: 25 May 2026.

What Sam shipped

ACMS for Linux: 423 of its 552 commits between May and August 2026, the 336 numbered iterations and the lockstep proof against a real ia64 system. The accounting application in VAX BASIC and FMS: 258 batch jobs graded, 551 screens walked, 53 defects registered, the update kit delivered, in fifteen days. AVERTZ, HP's own transaction-processing sample, running on x86.

What that proves

The VMS knowledge did not have to be in the engineer's head. It sat in the real system, in the rules the agent works under, and in the evidence every step left behind. A capable engineer with no OpenVMS background can develop, debug and test on VMS from the first week, provided every claim is measured on the host.

The scarce skill is no longer thirty years of VMS. It is the discipline to make the agent prove what it says, and the patience to read what it measured.

Eight products, one method

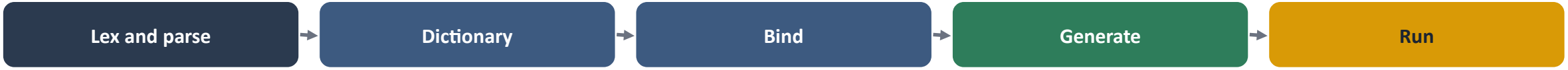
A gate per change, the real OpenVMS product as the source of truth; sizes from the repositories on 11 October 2026

Product	What it is	Written in, size	Started	How it was proven
VX/EDT	the EDT editor for Linux, macOS, Windows	C++17, one file, 6,905 lines	July 2026	measured against EDT V3.13 on t86 and ia64; 4 suites
VX/TPU	DECTPU with an EVE layer	C++17, one file, 9,054 lines	July 2026	129 DECTPU built-ins inventoried; compare run on t86's DECTPU
VX/SQL	SQL$PRE-compatible precompiler and multi-database runtime; unit, system and performance tested with AI	C++ and C, 107,481 lines (361 files); 54,997 lines of tests	May 2026 (import)	SQL$PRE V7.3 parity suites; a 3,184-program customer corpus: 99.0% translate, 96.9% compile clean
VX/RDML	RDML to embedded SQL translator; unit, system and performance tested with AI	C++, 63,864 lines (137 files); 41,605 lines of tests	2026, in the VX/SQL repo	1,124 customer modules; 1,466 of 1,467 compile on IA64
VX/CDD	CDD/Repository replacement with a callable shareable image	C++, 78,126 lines (153 files); 28,170 lines of tests	1 May 2026	callable parity 40/40 against the real image; symbol vector measured from VSI VDD on t86
VX/TUXEDO	wire-compatible Tuxedo for x86 OpenVMS and Linux	C++, 42,492 lines (131 files); 40,032 lines of tests	August 2026	byte-identical to Oracle 22.1.1 captures; 10/10 live scenarios
VX/DECFORMS for TDMS	DECforms runtime and compiler extended to run TDMS forms and syntax	DECforms: 93,172 lines since 2016; the TDMS compiler and runtime: 2026	June 2026	TDMS IVP forms, requests and records compile and render; 105 of 105 customer forms converted
ACMS for Linux	a from-scratch ACMS: EXC, SP, CP, QTI, ADU, audit, operator	C++20, about 37,500 lines; 39,000 lines of tests	17 May 2026	AVERTZ on the real form; lockstep vs ia64, 0 differences

Six of the eight have no history before May 2026 in their repositories; DECforms has ten years of it. The common thread is not the language or the size; it is that each one has a written specification, a test suite that outnumbers or rivals its product code, and a comparison against the original on a real OpenVMS system.

VX/SQL: the SQL$PRE-compatible precompiler and its runtime

Rdb embedded SQL in COBOL, FORTRAN and Pascal, compiled unchanged on x86 against PostgreSQL, SQL Server, MySQL or Oracle



host language and EXEC SQL blocks, SQL$PRE V7.3 command line and qualifiers

INCLUDE FROM DICTIONARY resolved through VX/CDD or native VDD

host variables, indicators, quadword dates, COMP-3 to NUMERIC

calls to the S7 runtime API in the host language; SQLCA, RDB$MESSAGE_VECTOR, WHENEVER

driver bound at link time; SET TRANSACTION lowered per database; DECdtm XA adapter

- Sector7's precompiler, unit, system and performance tested with AI: 252 commits between 25 May and 18 September 2026. It parses the host language and the SQL, resolves INCLUDE FROM DICTIONARY, and generates calls to a runtime API; SQLCA, SQLCODE and RDB$MESSAGE_VECTOR behave as Rdb documents them
- Tests outnumber hope: 127 COBOL .SCO programs and 146 SQL module files in the suite; SQL$PRE command-line parity 42 cases, include parity 7, CLI 25; the XA code generator 87 passed, 0 failed; each non-PostgreSQL driver suite 63 passed
- On a customer corpus of 3,184 embedded-SQL COBOL programs: 3,153 translate (99.0%) and 3,086 compile clean with VSI COBOL on x86 (96.9%); the rest are listed by file and reason in the hand-off notes
- The XA adapter (September 2026): each ACMS server's PostgreSQL connection binds to DECdtm as a resource manager; the global transaction id travels inside the PREPARE TRANSACTION name; recovery resolves in-doubt branches from pg_prepared_xacts
- Proven on AVERTZ: 18 procedures precompiled with /XA_COMMIT, 25 context calls generated, a real two-phase commit witnessed by three logs (Part 6)

```
$ SCOB :== $VXSQL_ROOT:[OPENVMS.BIN]VXSQL.EXE
$ SCOB /COBOL /NOWARN /DEBUG /XA_COMMIT -
     /OBJECT=AVERTZ_OBJ:VR_FIND_CU_PROC.OBJ -
     AVERTZ_SOURCE:VR_FIND_CU_PROC.SCO

*> generated:
    CALL "S7SetContextXaV1" USING BY VALUE 32
         BY REFERENCE CONTEXT-STRUCTURE END-CALL
    CALL "S7SetTransaction" USING BY REFERENCE SQLCA
...
    CALL "S7ExecSelectIntoOne" USING ...
```

the one line that changes in a build: the symbol SCOB

VX/CDD: the dictionary, and the symbol vector for its shareable image

CDD/Repository re-implemented: CDO, DMU, DDL, and a callable image with the CDD$ entry points at the indexes consumers expect

- What it is: a re-implementation of CDD/Repository with the CDO and DMU command languages, DDL, CDD$EXTRACT for the compilers, and a callable CDDSHR.EXE on x86 OpenVMS backed by SQLite. 78,126 lines of C++, 128 commits from 1 May to 23 September 2026, the AI co-author trailer on 66 of them; it grew from a 9,898-line predecessor inside the VX/SQL repository
- Why the symbol vector matters: on OpenVMS a shareable image exports its routines and condition values by index in a symbol vector. ACMS, DECforms and the compilers were linked against the real CDDSHR.EXE, so every CDD$ routine and every CDD$_ condition must sit at the same index, or a native consumer imports the wrong value
- How the table was made: a 78-line table (index SYMBOL=PROCEDURE or DATA, first written 22 June 2026) drives BUILD_CDDSHR.COM, which writes the SYMBOL_VECTOR into the options file at build time, pads gaps with SPARE, adds 45 CDD$K_ constants from slot 1495, then the conditions and library entries, and refuses to link unless the count is exactly 1,692. A test checks the built image with objdump: the vector section must be 0x69c0 bytes, 1,692 entries of 16
- How it was verified: in September the agent compared the installed VSI VDD image and the VX/CDD image on t86 with ANALYZE/IMAGE, readelf and objdump. All 112 common procedure exports matched; five condition values did not. CDD$_ENDFIND lives at slot 519 in VDD; VX/CDD had left 519 zero and exported it at an invented tail slot 1680, which is why ACMSADU crashed on the first CDD walk. The vector was corrected to the measured indexes; the old slots are kept as private data so already-linked VX/CDD consumers still activate

```
$! BUILD_CDDSHR.COM: link vxcdd as CDDSHR.EXE with condition
$! values at the fixed CDD V7.3 symbol-vector indexes verified
$! against the real IA64 SYS$SHARE:CDDSHR.EXE image.
...
$ write opt "SYMBOL_VECTOR=(-"
$ ... one line per index from CDDSHR_VECTOR_613.DAT:
$ write opt " SPARE,-"           ! gaps
$ vector_cddk_base = 1495        ! CDD$K_* constants
...
$ if vector_count .ne. 1692 then goto bad_vector_count
$ link/shareable/executable=[.BIN]CDDSHR.EXE -
$ write sys$output "Built [.BIN]CDDSHR.EXE with 1692 vector
entries."

CDDSHR_VECTOR_613.DAT:
0 CDD$CLEAR_CELL=PROCEDURE
1 CDD$CREATE_ACL_ENTRY=PROCEDURE
2 CDD$CREATE_DIR=PROCEDURE
...
519 CDD$_ENDFIND=DATA        ! was 1680
```

vxtools85/cdd/openvms: the build procedure and the vector table

The lesson for any shareable-image replacement: the ABI is the index, not the name. Measure the real image; do not trust a header file or a listing.

VX/TUXEDO: compatibility lives on the wire, not on paper

Oracle publishes no wire specification, so every byte was measured from a real Oracle Tuxedo 22.1.1 and reproduced

1. Capture → **2. Decode** → **3. Implement** → **4. Gold, compare**

a TCP relay between a real Oracle client or GWTDOMAIN and the Oracle server records every byte both ways: exact frame bytes and lengths

every frame decoded to the byte: a 750-line field-by-field map of the wire, 37 captured Oracle frames checked in as evidence

VX/TUXEDO built to the captured wire: TDOMAIN handshake, WSL redirect, WSH session, STRING and FML32 buffers

capture a gold sample, run VX on the same input, byte-compare. Any Oracle quirk that survives is reproduced deliberately

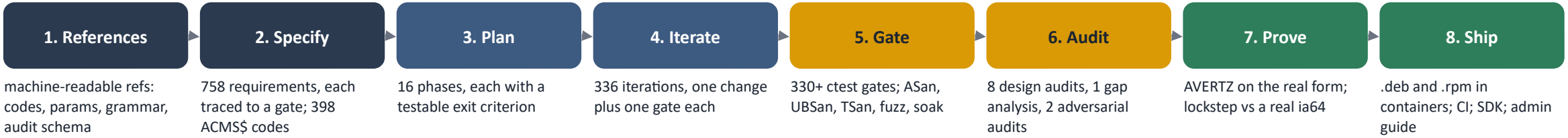
```
ORACLE gold capture          VX/Tuxedo re-encode
00000094 00001000 00000008   00000094 00001000 00000008
0000000a 000003ec fffffb2e   0000000a 000003ec fffffb2e
02000003 eb000679 32...      02000003 eb000679 32...
                             diff = 0 bytes   MATCH
```

an 8-field FML32 reply decoded and re-encoded: 140 bytes out, identical to Oracle's 140 bytes in

- Scope set by inspecting the customer's source and configuration: STRING and FML32 buffers, WSL and WSH with SECURITY=NONE, GWTDOMAIN peers, NO_XA
- 80 automated differential tests under Release and ASan/UBSan; a regression gate that boots Oracle Tuxedo 22.1.1 and exchanges live traffic: 10 of 10 scenarios, both directions, including TPFAIL, empty-buffer replies, reconnect and dual-endpoint failover
- 42,492 lines of C++ with 40,032 lines of tests; 7 large commits from 29 August to 1 October 2026, the AI co-author trailer on 5; 80 tests registered with ctest, the live-Oracle differential 13 of 13
- Out of scope and said so: CARRAY and FML16, XA, LLE/TLS, byte-identical server-side emit

ACMS for Linux: specification in, a proven system out

A from-scratch C++20 ACMS for Linux and macOS, 17 May to 14 August 2026, built to match the real ACMS behaviour by behaviour



521
commits in ninety days

336
iterations, each with a gate and a change-log entry

39k / 37k
lines of test code versus product code

~1M
task executions in the 24-hour soak, zero failures

15+
upstream DECforms runtime fixes

0
real differences from the ia64 over 166 lockstep steps

31
gates under ThreadSanitizer; all under ASan/UBSan

7 of 7
AVERTZ tasks on the real form

Two decisions paid for everything: reuse the real DECforms runtime and the real CDD rather than reimplement them, and accept a gate only after it has been seen to fail. What is out of scope is enforced in code as a loud refusal.

Four codebases, side by side

Named only by what they are; every count is from a scan, an inventory file or a build log in the repository

	A: accounting system	B: banking application	C: manufacturing suite	AVERTZ: HP's ACMS IVP
Era, origin	1990s	production estate moving from Itanium to x86	1990s to today, 5.06M lines scanned and fingerprinted	1991 HP sample, shipped with ACMS
Languages	VAX BASIC, MACRO-32	COBOL, C, DCL COBOL with embedded SQL, COBOL, C, DCL, MMS	62% VAX Pascal (2,555 files, 3.23M); COBOL and C (165 files, 311K); FORTRAN; MACRO-32	COBOL with embedded SQL, C, TDL
Screens	FMS	DECforms DECforms (196 IFDL, 428K lines)	TDMS (105 forms) and DECforms (414 IFDL forms, 783K lines)	DECforms (VR_FORM, 4,296 lines)
Data	RMS indexed files (fixed length, mapped), 400+ logical names	Oracle Rdb Oracle Rdb: EXEC SQL in 3,230 files; 286 SQL and 56 SQL module files	Oracle Rdb through RDML (1,124 modules, 35,916 statements); RMS (147 FDL definitions)	two Rdb databases, now PostgreSQL schemas
Middleware	DCL batch and an overnight controller (27 procedures)	Oracle Tuxedo (FML32, WSL, TDOMAIN), ACMS Oracle Tuxedo (FML32, WSL, TDOMAIN); ACMS (527 TDFs, 70 ADFs, 13 GDFs)	DECmessageQ (1,115 modules on the PAMS API), DECnet to 25 site nodes	ACMS tasks, DECdtm two-phase commit, queued tasks
Dictionary	record layouts and FDLs	CDD CDD: 2,002 CDO files; FROM DICTIONARY in 4,214 files	CDD: 3,460 records, 65,108 elements	CDD: 27 workspace records
Size	about 800,000 lines of BASIC; 263 programs plus a 1,415-module menu library; 258 batch jobs; 551 menu functions	12,262 source files, 6.53M lines; 3,184 embedded-SQL COBOL programs (3.25M lines); 29 VMSINSTAL kit procedures	6,981 source files; 3,870 modules; 297 VMS APIs at 48,198 call sites; 120 application images	28 procedures, 9 TDFs, 2 forms, 62 build procedures
Status	running on x86 t86, 208 of 258 jobs proven, kit delivered	middleware proven on the wire; ACMS definitions parse 612/612; 99.0% of the COBOL translates; the IA64 build campaign is not complete	converters run on the whole corpus; 1,466 of 1,467 translated modules compile on the IA64 reference	runs on x86 against PostgreSQL with XA; 13-stage proof

The steps the AI takes on any codebase

Four stages, the same for a 6,000-line sample and a 5-million-line estate; each stage leaves a file the next can be checked against

1. Analyse the code base

- scan every file: language, size, SHA fingerprint; cluster by pattern
- inventory the dictionary, forms, SQL and RDML, DCL, FDLs, message files
- call-site census of every VMS API: SYS$, LIB$, STR$, SMG$, FORMS$, TSS$, PAMS
- build tree and orphans: which sources reach an image, which are referenced by nothing
- what is missing: include files, CDD records, form sources, libraries

2. Analyse the build

- read the .COM and MMS files: compile qualifiers, link order, options files, logicals
- list every executable and the sources that feed it
- find the reference host that can still build it: IA64 with Rdb, x86 with VX runtimes
- write the build as a manifest-driven procedure with one command and a log per step

3. Build the executables

- compile on the reference first, so the only moving part is the converter or the runtime
- grade every compile and link log; a warning is a row, not noise
- float formats, record sizes and shareable-image vectors are decided here, not at run time
- the first run of anything is a batch job with its log kept

4. Create the installable kit

- BACKUP save sets of runtime, build tree and startup; verify pass; read-back listing; SHA-256
- or PCSI / .deb / .rpm with the admin guide's flags resolved against the binaries
- transfer over the customer's tunnel, hash again on arrival
- CHECK pass before APPLY; every old version kept; two independent rollbacks

What the AI contributes at each stage is reading speed and patience, not judgement about the business. The converters stay deterministic and the reference host stays the source of truth.

A: an accounting system in VAX BASIC and FMS, to x86

About 800,000 lines of BASIC and MACRO-32; 208 of 258 batch jobs and 433 of 551 menu functions proven on t86 in fifteen days

Analyse	Source, includes, record layouts, FDLs, the product documentation and a damaged dataset. 400+ logical names set by a chain of start-up procedures. Fixed-length mapped indexed files for data. Two historical builds found: one VAX floating point, one IEEE.
Build environment	OpenVMS x86-64 V9.2-3 on a VM, FMS, TCP/IP, one private tree so the product's demo data is never touched. The agent reached it over ssh with one account.
Build	263 programs plus 20 supplement modules compiled, linked and analysed in three minutes from a manifest. The link order was the thing to get right: a 1,415-module menu library placed first broke account loading in a dozen reports.
Kit	BACKUP writes three save sets; verify pass, read-back listing, SHA-256; 4.95 GB over the customer's tunnel and hashed again; CHECK pass then APPLY into a clone of the host, every old version kept.

VAX floating point to IEEE

- The recovered data was written by an IEEE build; the historical VAX-float build read every value as the wrong format: FLTINV, 'Illegal number', 36 runs repeated
- Decision: /REAL_SIZE=TFLOAT everywhere; a LONG passed where a TFLOAT is expected is 4 bytes short (Part 5, case 1)

The product documentation as the source of truth

- 258 jobs came from the documentation, the menu dispatch tables and the nightly kit; 149 sources read for what each job asks on stdin
- The documentation says what a job was meant to do: a fault, or a row the dataset never had. A log without 'job terminated' is inconclusive, never a pass

C: a manufacturing suite in Pascal, RDML, TDMS and DECmessageQ

Five million lines; the converters were run on the customer's own code before a proposal was written

Analyse

CodeScanner fingerprinted all 6,981 files and 5.06M lines; LegacyMap produced the call and caller graph; inventories list 3,870 modules, the sources behind 120 images, the build tree and its orphans, 12 missing form sources and the missing CDD records. 297 VMS entry points at 48,198 call sites, every one mapped to a VX product.

Build environment

An IA64 reference host with Rdb, where the original .RPA Pascal still builds: the baseline. The converted .SPA is built beside it, so the only moving part is RDML to SQL. Linux and Oracle come after the two IA64 builds agree.

Convert and build

All 105 TDMS forms through VX/TDMS; all 1,124 RDML modules machine-translated to embedded SQL (90,999 EXEC SQL blocks) and 1,466 of 1,467 compile to verified objects on the IA64 reference; the whole CDD (3,460 records, 65,108 elements) emitted as C++ and IFDL includes.

Test and kit

A 126-page unit-test plan, one table per image: inputs, relations, RMS endpoints, forms, queues, assertions, an automation score. An anonymisation program for 520 fields so real data never leaves the customer. AI generates and expands the tests and triages failures; expected behaviour comes from the reference, never from the AI.

```
START_STREAM GET_PRODUCT_NUMBER USING TRQR IN TEST_RPT_QUALITATIVE_REQ WITH
  TRQR.DUNS_NUMBER = DUNS_NUMBER AND TRQR.PRODUCT_NUMBER CONTAINING "*"
  REDUCED TO TRQR.PRODUCT_NUMBER;                  -- RDML in
```

```
EXEC SQL DECLARE RDML_C_3_GET_PRODUCT_NUMBER CURSOR FOR
  SELECT DISTINCT TRQR.PRODUCT_NUMBER FROM TEST_RPT_QUALITATIVE_REQ TRQR
  WHERE TRQR.DUNS_NUMBER=:VXH_1 AND TRQR.PRODUCT_NUMBER LIKE '%*%';    -- SQL
out
```

The brief, as it was typed

The whole specification was a target and a bar: functionally identical to the real thing

```
... perform a gap analysis on our version tpu.cpp,
Ralph loop until all gaps are fixed. implement a versioning
scheme and add a change log pdf with all the changes from one release to
another, similar to the EDT project.

You can run real TPU on machine (ssh) t86 probably with the command EDIT/TPU
and verify that our TPU is functionally identical to real openvms TPU.

...

Ralph loop until all gaps are closed and all of your regression tests pass
and you have validated that real openvms TPU and our TPU are functionally
identical.
```

Cursor session 2026-07-15, the TPU sibling of the EDT project; spelling tidied, opening clause elided

- A Ralph loop: the agent re-reads its own gap list after every fix and keeps going until the list is empty and the tests are green
- The real editor is the specification. Nothing was written from memory of how EDT behaves
- The real editor on t86 is the source of truth. Where a behaviour is in doubt, the agent runs the same command on both and compares
- Versioning and a changelog were asked for up front, so every release names what changed and against which VMS measurement

Why EDT first: it is small enough to finish, old enough that its behaviour is settled, and every OpenVMS user in the room can tell in ten seconds whether the keypad is right. It is also the editor a migration team asks for on day one of a move to Linux.

Measured on real OpenVMS, then reproduced: the /COMMAND release

A customer field request became V1.11-0; every behaviour was run on EDT V3.13 on ia64 first, then matched

```
$ edit/edt/command=daily daily.lis   ! daily.edt: INCLUDE, ranged SUBSTITUTE, EXIT

Measured on OpenVMS (ia64, EDT V3.13), reproduced on the same fixture files:
  /COMMAND=file replaces EDT.INI for that run (EDTINI.EDT not read)
  default type .edt; a missing command file aborts with exit status 4:
     Command file does not exist / File name:daily
     %RMS-E-FNF, file not found / %EDT-F-EDITORABO, Editor aborted
  EXIT inside the file saves, prints the exit message, then [EOB]
  INCLUDE prints nothing on success; the current line stays where it was
  S/old/new/range replaces on every line of the range; a trailing number
     is a line number, not a repeat count
  TYPE END (and T E) types [EOB], not the last line
  range abbreviations E, EN, B, BE, WH, WHO, R, RE, LA, LAS; a lone L rejected

Result: the customer's file gives the identical 9-line result and output.
```

- No written specification for this one. The agent ran the customer's command on the real system, captured the output, and made the port match
- The changelog also lists what still diverges, measured in the same session: positional line numbers, the error-caret format, SET QUIET semantics. 'Listed so nobody mistakes them for done'
- The DCL layer of VX/RT was fixed in the same release: EDIT/EDT was silently dropping every qualifier when the editor was configured as edt.exe

EDT_CHANGELOG.md, V1.11-0 entry; 35 new checks in edt_regression.sh section 33

The pattern for the whole talk: when a behaviour is observable, do not reason about it. Run it on the real system, keep the capture, and make the port reproduce the capture.

What AVERTZ is

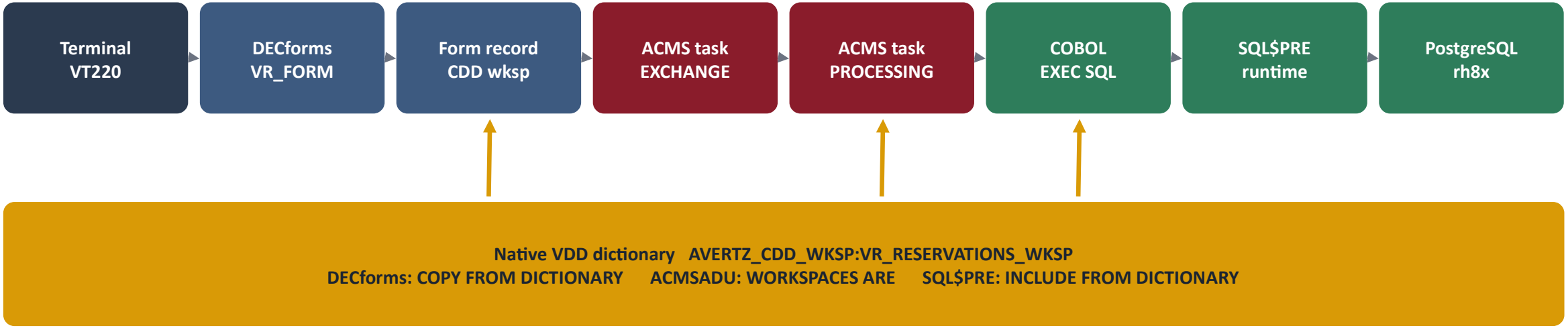
The vehicle-rental IVP shipped with ACMS: every OpenVMS transaction-processing component in one application

ACMS application	VR_APPL_XA: one EXC, five procedure servers (read, update, customer update, log, queue), eight tasks
DECforms	VR_FORM: 24x80 panels for reserve, checkout, check-in, customer and vehicle lists; the RESVLIST form we added
COBOL procedures	18 embedded-SQL procedures (.SCO) precompiled with SQL$PRE /XA_COMMIT; 25 USING CONTEXT statements
Two databases	VEHICLE_RENTALS and VEHICLE_HISTORY, once two Rdb databases, now two schemas in avertz_demo on PostgreSQL
Task definitions	TDFs with BLOCK WITH FORM I/O for screens and BLOCK WITH TRANSACTION for the distributed write
Dictionary	23 CDD workspace records plus field definitions, loaded into native VDD; shared by ACMS, DECforms and the COBOL
C agents	The menu agent (ACMS Systems Interface with DECforms exchange I/O) and the acceptance agent that forces commit or abort
Queued tasks	VR_FAST_CHECKIN_TASK enqueued for the composite fast check-in: QTI starts the DECdtm transaction, the task joins it

Original 1991 HP source. Business logic, screens and task definitions unchanged; the database underneath is PostgreSQL.

How data flows through AVERTZ

One dictionary record is the screen field, the task workspace and the SQL host variable



Operator types BELL, MANCHESTER, class 3. DECforms validates against the form record and transfers vr_control_wksp, vr_customers_wksp and vr_reservations_wksp to the task's EXCHANGE step.

The PROCESSING step calls vr_write_rs_proc IN vr_update_server USING the same workspaces. The COBOL INCLUDEd them FROM DICTIONARY; SQL$PRE binds their fields as $1..$14 of INSERT INTO RESERVATIONS.

Three consumers, one definition: change a field in the dictionary and the screen, the task workspace and the SQL host structure all rebuild against it.

Dates cross every boundary as the same quadword: DECforms date picture, workspace field, COBOL host variable, and only at the runtime edge become a TIMESTAMP.

Where the transaction lives: the task definition

ACMS opens the DECdtm transaction in the TDF and ends it in the block action; the COBOL joins it with USING CONTEXT

```
BLOCK WITH TRANSACTION
    IF ( (vr_customers_shadow_wksp.rec_status = "1") OR ... ) THEN
        PROCESSING
            CALL PROCEDURE vr_store_cu_proc IN vr_cu_update_server
            USING vr_control_wksp, vr_customers_wksp, vr_trans_wksp;
        ACTION IS
            IF (ACMS$T_STATUS_TYPE = "B") THEN ... EXIT BLOCK; ...
    END IF;
store_resv:
    PROCESSING
        CALL PROCEDURE vr_write_rs_proc IN vr_update_server
        USING vr_control_wksp, vr_reservations_wksp, vr_sites_wksp,
              vr_rental_classes_wksp, vr_customers_wksp;
    ACTION
        MOVE "RESERVE " TO vr_hist_wksp.trans_type;
    PROCESSING
        CALL PROCEDURE vr_write_hist_record_proc IN vr_log_server
        USING vr_hist_wksp, vr_reservations_wksp;
END BLOCK;
    ACTION IS
        IF (ACMS$T_STATUS_TYPE = "G") THEN
            COMMIT TRANSACTION; ...
        ELSE
            ROLLBACK TRANSACTION;
            GOTO STEP display_cust_info;
        END IF;
```

VR_RESERVE_TASK.TDF (HP, unchanged)

```
EXEC SQL USING CONTEXT :CONTEXT-STRUCTURE
    SET TRANSACTION READ WRITE
    RESERVING RESERVATIONS FOR SHARED WRITE, ...
END-EXEC.
...
EXEC SQL USING CONTEXT :CONTEXT-STRUCTURE
    INSERT INTO RESERVATIONS ... END-EXEC.
```

VR_WRITE_RS_PROC.SCO (HP, unchanged)

```
*> VX/SQL: USING CONTEXT :CONTEXT-STRUCTURE -- strict DECdtm XA
    CALL "S7SetContextXaV1" USING
        BY VALUE 32
        BY REFERENCE CONTEXT-STRUCTURE
    END-CALL
    CALL "S7SetTransaction" USING ...
```

generated by SQL$PRE /XA_COMMIT: the TID in CONTEXT-STRUCTURE enlists this server's PostgreSQL branch

- BLOCK WITH TRANSACTION: the EXC calls $START_TRANS
- COMMIT TRANSACTION: $END_TRANS; DECdtm prepares and commits every enlisted branch
- ROLLBACK TRANSACTION and CANCEL TASK: $ABORT_TRANS

Building AVERTZ from scratch on x86

One command, thirteen gated steps, every output verified non-empty; the AI wrote and drove every procedure

1. SETUP → **2. CLEAN** → **3. MESSAGE FILE** → **4. COMPILE TASKS** → **5. TASK GROUP**

isolated roots, 16 sources, VDD repository, credentials INI

delete prior objects, images, forms, TDB, ADB

VRMSG.OBJ / VRMSG.EXE

ACMSADU: 8 TDFs into native VDD

VR_TASK_GROUP_XA.TDB, 5 server objects

6. FORMS → **7. MENU AGENT** → **8. SQL$PRE** → **9. LINK SERVERS** → **10. APPLICATION**

FORMS TRANSLATE / EXTRACT / LINK: VR_FORM, VR_RSLIST_FORM

cc + link VR_FORMS_MENU_XA.EXE

18 .SCO through SCOB /COBOL /XA_COMMIT; generated COBOL audited for 25 XA context calls

5 server images + 2 agents against VXSQL_PGSQL_XA_RUNTIME_SHR

ACMSADU: VR_APPL_XA.ADB

11. PREFLIGHT → **12. DEPLOY** → **13. RUN**

runtime, VDD, DECdtm image, two recovery identities, PostgreSQL services

copy ADB, provision 33 ACEs for 4 accounts, ACMS/START, verify 5 servers

RUN_AVERTZ_XA: menu, RESERVE, CHECKOUT, CHECKIN, LIST

```
$ @DKA200:[JONPOWER.AVERTZ_XA_20260910]BUILD_AVERTZ_VDD_PG_XA
AVERTZ-XA >> Build complete: native_vdd=1 sql_procedures=18 xa_commit_preprocessed=18 xa_context_calls=25 forms=2 tasks=8
```

Every bug starts as a line in a log

The method the agent follows, whether the fault is in BASIC from 1994 or in a TCP handshake

What the log says

The program, the error and the line number. The grader collects every such line across all logs, so the starting point is never a feeling.

What the source says

Read the lines around the number, then the routine it calls, then the include that declares the record. The cause is usually two files away from the failing line: in the called routine, or in the include it depends on.

What the data says

Small read-only listers written on the spot: show an asset's fields, list an account's transactions, dump a control table. They settle in minutes what the symptom only hints at.

What the documentation says

The application's own documentation says what a job was meant to do. That separates a fault from a missing fixture.

Reproduce → **Checkpoint** → **Patch** → **Rebuild, rerun** → **Regress, record**

- Reproduce: same shell, same inputs, fresh log. If it needs data, the smallest fixture that triggers it
- Checkpoint: a BACKUP save set of the data tree before anything that writes. Taken as a batch job: a checkpoint from a plain ssh session writes zero blocks
- Patch: a private copy of the source named after the module, never the recovered original. Rebuild with the one command, link in the one order
- Regress and record: rerun the neighbours; register entry with the log before and the log after; commit

Case 1: a silent type mismatch in nine BASIC reports

Application A (VAX BASIC, FMS): 'Illegal I/O channel' right after the heading printed

Step	What the agent did	What it found
Log	Collected the error triplet from nine report logs: program, 'Illegal I/O channel', line number	All nine failed at the same point: just after the page heading
Source	Read the caller around the line, then SUB Heading, then the include declaring the parameters	The caller declares a line counter as LONG and passes it to a SUB whose parameter is TFLOAT
Mechanism	Traced the 8-byte write against the 4-byte variable in the caller's storage	The write overruns into the adjacent variable holding the report channel number
Why nine	Compared the nine callers	All cloned from one template; the fault was present in every copy
Fix	Declared the counter TFLOAT in the nine callers; private copies under the module names; FULL09 rebuild	All nine complete; logs before and after in the register as A1

What made this an AI job: 263 programs, 1,415 library modules and no one alive who wrote them. The agent read nine callers and one SUB in minutes, and had already learned from the build that this codebase was compiled with REAL_SIZE=TFLOAT, which is why a LONG passed where a TFLOAT is expected is 4 bytes short.

Case 2: the XA recovery process that failed only at boot

AVERTZ on t86: the first start of the PostgreSQL recovery process after a reboot exited with no message; a retry a minute later worked

Observe

Three reboots: attempt 1 fails on some boots, never on a warm start. Each failed process lived 5.05 s, used 0.07 s CPU, exit status %X10000002

Decode

%X10000002 is the C run-time library's documented translation of exit(EXIT_FAILURE). The recovery source returns EXIT_FAILURE only when PQconnectdb fails; connect_timeout=5

Capture

tcpdump on rh8x, port 5432 from t86, across reboots 3, 4 and 5. The boot procedure changed to keep a failed attempt's output

Explain

t86 sends SYN from port 49168; rh8x answers with a bare ACK carrying an old connection's sequence numbers (RFC 5961 challenge ACK); t86 sends RST; t86 does not retransmit the SYN within 5 s

Root cause

t86 allocates ephemeral ports from 49166 upward in the same order after every boot. The recovery processes run as SYSTEM; SHUTDOWN.COM skips UIC group 1, so they die with the system and never close their sockets. The stale backend on rh8x still owns the four-tuple

Fix and prove

Retry loop in the recovery procedure (next ephemeral port), starter polls the verifier. Proven warm with iptables dropping every second SYN, then at boot: three attempts, then READY

'Nothing in it is inferred from code alone.' Stage 12 of the run record, 9 October 2026; evidence files 42 to 64 in the repository.

Every batch job: one shell, one log, one grade

Application A: 258 jobs from the product documentation, the menu dispatch tables and the nightly kit

1. Inventory → **2. Author** → **3. Run** → **4. Grade**

258 jobs. 149 sources read for what each job writes and what it asks on stdin

a DCL shell per job: runtime logicals, the image, its answers as stdin lines. Kit shells reused where they existed

upload, SUBMIT to SYS$BATCH, wait, fetch the log. The runner deletes its own job if it hangs

a script reads every log: termination line plus observed output is 'works'; a diagnostic line is a failure to chase; no termination line is inconclusive, never a pass

208
of 258 jobs work

38
run, nothing to do on this dataset

12
not testable: no source, or not a batch job

5
rounds, each unlocking the next

2,000
batch entries, every log kept

Round	What ran	What it needed first
1	read-only reports and extracts	a correct build and the runtime logicals
2	mutating jobs: updates, purges, posts	a BACKUP checkpoint before, a restore after
3	jobs that stopped on missing rows	loaders for control-table rows, pricing services, factor dates
4	vendor feed chains	feeds fabricated from the record maps; VAR/CR files through CONVERT
5	the overnight controller and its 27 procedures	installed procedures, condition rules, arrivals stamped first

Every menu function: a robot walked all 551 screens

An expect script logs in, follows a key path from the main menu, captures each screen with the escape codes stripped, and backs out

- With prompts turned on it answers each field from a rule table: the test account, the test asset, dates inside the dataset's window, N to every yes/no, cancel at every store
- Read-only, 130: walked twice, the second time with prompts answered. Mutating, 221: reviewed first; 73 that act on selection excluded; the rest walked against a fresh checkpoint, restored afterwards. Other, 200: 45 excluded, 155 walked
- A grader reads the captures: SUBMITTED, REACHED, NO_DATA, PROMPT, NOT_REACHED, ERROR
- Forms are not line input: fixed-width fields auto-advance, dates are six digits with no slash, one-character defaults must be deleted before typing. Every convention is a rule in the harness, matched on the latest screen change only

433
functions walked

67
queued a report

87
reached a result or form

62
need a human's values

7
errors, all explained

2
screens that saved on a bare Return, now excluded

The host fought back: one login per function, 130 times in a row, stopped port 22 answering while ping still worked. Now: many functions per login, 15 seconds between logins, a rest after each burst, never kill a connecting session. The seven errors were three screens crashing on a missing lot-editor file, one on an empty locations file (both fixed), and three on a subsystem that is switched off by design.

Proving two-phase commit: three independent witnesses

AVERTZ RESERVE on x86: the PostgreSQL log, DECdtm's log and a pg_prepared_xacts observer agree on the same four milliseconds

```
-- forced DECdtm abort after the task ran (SYS$ABORT_TRANSW)
17:33:36.265 vehicle_rentals  BEGIN
17:33:36.886 vehicle_history  BEGIN
17:33:36.895 vehicle_history  ROLLBACK     <- no prepare: aborted before phase one
17:33:36.895 vehicle_rentals  ROLLBACK

-- DECdtm commit (SYS$END_TRANSW)
17:33:40.172 vehicle_rentals  BEGIN
17:33:40.789 vehicle_history  BEGIN
17:33:40.801 vehicle_history  PREPARE TRANSACTION 'VX1.VH.AQ1To9sQ...'
17:33:40.808 vehicle_rentals  PREPARE TRANSACTION 'VX1.VR.AQ1To9sQ...'
17:33:40.809 vehicle_rentals  COMMIT PREPARED      'VX1.VR.AQ1To9sQ...'
17:33:40.809 vehicle_history  COMMIT PREPARED      'VX1.VH.AQ1To9sQ...'
```

rh8x PostgreSQL server log, log_statement=all, acceptance run 14-SEP-2026

```
$ MCR LMCP DUMP SYSTEM$X86923.LM$JOURNAL/FORMAT
Transaction state (2):  COMMITTED
Transaction ID: F4502BAA-ADF5-11F1-8FD7-AA0004001304
Type  ( 3): LOCAL RM   Name (24): "XA$VXSQL_VR_RENTALS"
Type  ( 3): LOCAL RM   Name (24): "XA$VXSQL_VR_HISTORY"
```

DECdtm's own log: one committed transaction, two resource managers

- Same task, same input, opposite outcomes: the only difference is the DECdtm verb the acceptance agent calls, so the outcome is owned by the coordinator, not the COBOL
- A third witness: a psql observer polling pg_prepared_xacts during the run saw both VX1 branches appear prepared and disappear committed
- ACMSATR, the ACMS audit trail, records the task and its transaction id; the id in LMCP's dump matches it
- Enabling the server-side log fenced the XA servers (the runtime fingerprints pg_settings and refuses to run after a change). The run record keeps the incident: restart the application on both sides of any PostgreSQL configuration change

Lockstep: every keystroke compared with a real ia64 ACMS system

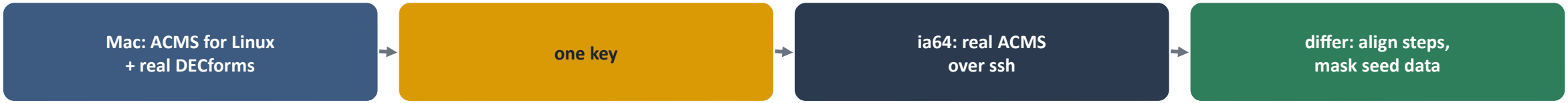
ACMS for Linux against the original: one key per step, screen text, video attributes and cursor position captured on both sides

Before

- 132 real mismatches over 166 steps on 2026-08-14
- All explained by 14 distinct differences: numeric cursor rest off by one everywhere, the entry panel advancing on RETURN instead of TAB's field ring, an unstyled menu banner, a missing '?Invalid Selection' row
- The measurement pass fixed nothing. Capture artefacts were identified and excluded: the emulator models erase-with-attributes, so blank cells showed phantom bold bands

After two fix waves

- All 14 fixed in the runtime and the front-end, never in the form source
- Full re-probe against the archived ia64 frames: 29 identical, 137 seed-only (reservation ids, dates, amounts masked)
- 0 real differences; nine AVERTZ regression gates still green



Three flows: reserve, check out, check in. Four dimensions per step: screen text, video attributes, cursor position and tab order, protected-field behaviour. The bar was raised on purpose from 'it runs' to 'indistinguishable, per keystroke, measured'.

A gate is accepted only after it has been seen to fail

The discipline behind the other numbers: 330 ctest gates for ACMS for Linux, 39,000 lines of test against 37,500 of product

Non-tautology

Revert the fix. The gate must go red. Restore. Green. The change-log entry records the exact failure observed. A gate once passed because its PASS token matched a line printed by a helper binary, not the assertion

Positive controls

A failing case is paired with a case that must still succeed, so the gate proves the mechanism and not a blanket deny: a failed-COMMIT gate keeps a valid child that still commits

Discriminating counts

Fixtures shaped so the right answer is a number no other path yields: five elements where the filter returns exactly three; exactly one server ever spawned

Real paths, not mocks

A commit failure is a real deferred foreign-key constraint. A crash is a real SIGKILL mid-call. A forged identity is a real second socket

Instrumented build	What it found
ASan + UBSan, whole suite	a use-after-dlclose in server shutdown, a task-argument array underflow in queue dispatch, a stack overflow in the queue builders, a 7 KB scratch array overflowing an HTTP thread's stack
ThreadSanitizer, 31 labelled gates	a real data race on its first run: the cancel flag was a volatile written under one lock and polled lock-free by the engine
libFuzzer over four grammars	truncated keywords, a 200 KB token, NUL garbage, real definitions cut at every 4-byte boundary
24-hour mixed soak	268,584 waves, about a million task executions, zero agent failures, resident memory flat. The 30-second version found the SQLite threading corruption first

What changed, and what did not

- The AI reaches OpenVMS the way a contractor does: ssh, a key, DCL one command at a time, batch jobs and logs
- Its VMS knowledge is broad and inexact; a real host to measure against makes it exact
- Eight products in five months, each with a written specification, a gate per change and a comparison against the real thing
- Four application codebases analysed, built and packaged; the converters stay deterministic, the AI calibrates, runs and triages
- Debugging and testing are where the hours went: every claim points at a log, a capture or a commit
- What did not change: measurement wins, nothing is skipped silently, and a fix is not done until it has been seen to work from clean state