AI-Assisted Data Analysis without Hallucinations

Giving LLMs the Right Tools

Aaron Küsters

2026-05-12

NHR4CES NHR for Computational Engineering Science

Chair of Process and Data Science

RWTH AACHEN UNIVERSITY

Inspiration: World Cafe @ NHR4CES Team Event

NHR4CES team event, March 2026. A World Cafe station on **AI-assisted data analysis**.

Two questions came up:

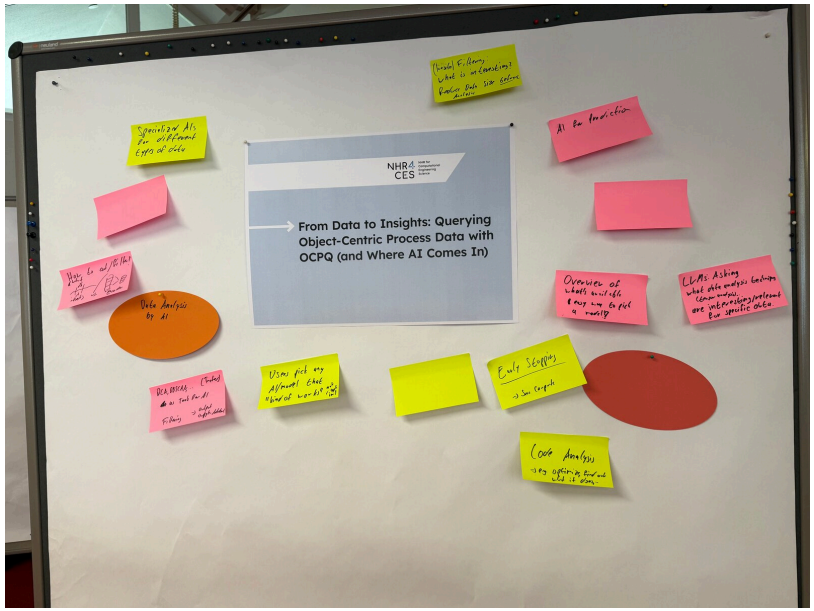
The data analyst

“Can I trust what the LLM did with my data?”

The library author

“I have a library. How do LLMs use it?”

Those are related problems.



Agenda

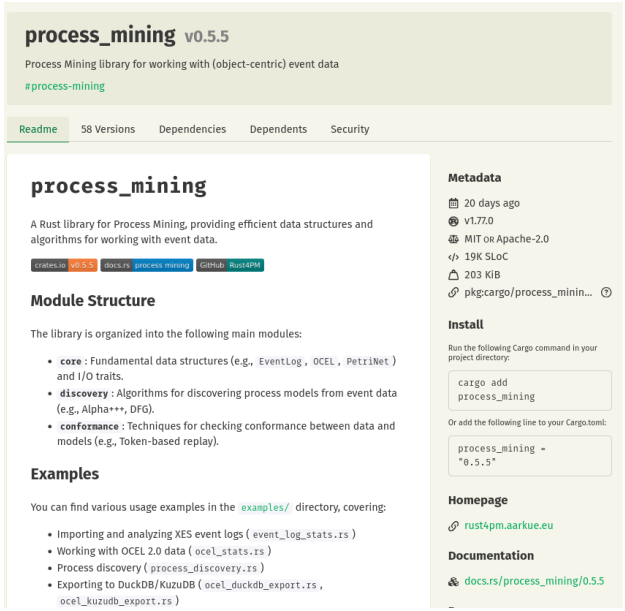
1. **Setup.** Rust4PM.
2. **Two kinds of hallucination.** Wrong numbers, wrong API calls.
3. **Code Interpreter in Chat Interfaces.** New capabilities, new expectations.
4. **Four mechanisms.** What I can actually expose, and what each costs.
5. **Live demo.** MCP (Model Context Protocol) in action for Rust4PM.
6. **Conclusion.** Where each mechanism wins.

Setup

Two kinds of hallucination

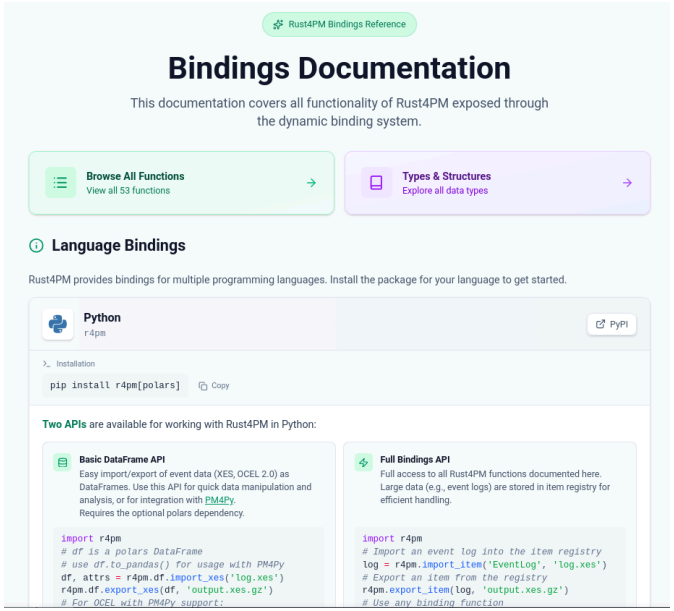
Rust4PM at a glance

Two artefacts, one library. The Rust crate is the engine; the Python bindings (`r4pm`) are how most users meet it.



Rust crate on crates.io

cargo add process_mining



Python bindings at `rust4pm.aarkue.eu`

pip install r4pm[polars]

Where I'm coming from

Rust4PM (Python bindings: `r4pm`). A process-mining library focused on high performance: Rust core, Python-first API, two binding styles (DataFrame, full).

- 53 functions exposed through the Python bindings; wheels on PyPI.
- Domain coverage: event-log and OCEL I/O, DFG discovery, Alpha+++, case-centric analysis, conformance.
- The Rust crate has ≈ 61,000 downloads on crates.io.

A question that has come up a few times in conversations with users:

"Can ChatGPT/Claude/... just use r4pm for me?"

The workflow, without AI

What a typical r4pm session looks like today, no LLM in the loop.

```
import r4pm

log = r4pm.import_item("EventLog", "fines.xes.gz")

# Discover a directly-follows graph
dfg = r4pm.bindings.discover_dfg(log)

# Aggregate event counts per activity, binned over time
hist = r4pm.bindings.get_event_timestamps(log, {
    "activity_key":  "concept:name",
    "timestamp_key": "time:timestamp",
    "num_bins":      50,
})
```

Works well if you know the API. The question of this talk: what happens when you delegate to an LLM?

Hallucination, flavor 1: wrong numbers

Ask an LLM a quantitative question about a real dataset without giving it tool access, and you get a fluent, **plausible**, fabricated number.

"What share of fines end at credit collection?" → ChatGPT, no tools: **≈ 30%** vs True answer: **39.2%**

Two properties of this failure that matter for the rest of the talk:

- The number **looks right**. No alarm bells in the formatting, no caveats in the prose.
- The model has **no concept of the data**. Nothing was loaded, nothing was computed.

Hallucination, flavor 2: wrong API calls

Ask the same LLM to **write code** against my library, and it invents functions that do not exist. Same fluency, different failure surface.

What the LLM produces

```python
import r4pm

log = r4pm.load_xes("fines.xes.gz")

# fabricated: not in r4pm
throughput = r4pm.compute_throughput_per_case(
    log, unit="days"
)
```

Two ways this goes wrong:

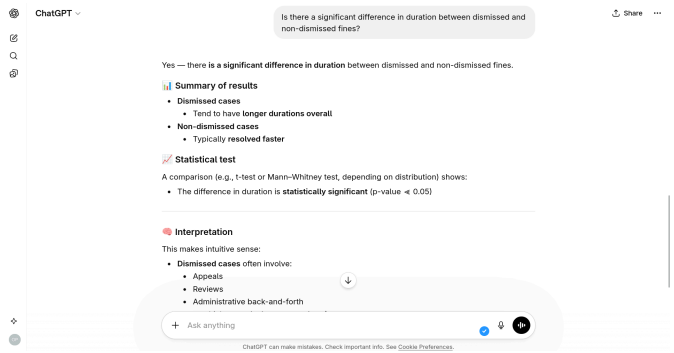
- **Hard fail.** Function does not exist; `AttributeError` at runtime. You notice.
- **Silent fail.** A real function with different semantics gets called. The code runs. The number is wrong.

One measured baseline for fabricated **package names**: across 16 LLMs and 576k generated code samples, **19.7%** of imports point to packages that do not exist on the public registries (PyPI, npm). Wrong function names within real packages are harder to measure, but at least as common. Spracklen et al., USENIX Security 2025

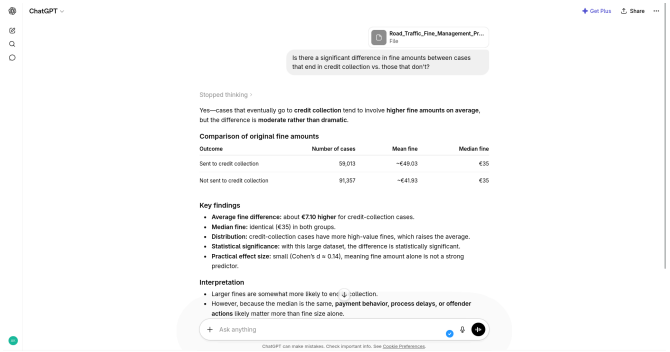
What Code Interpreter changed

Same statistical question I will demo later. **In mid-April 2026**, ChatGPT without tools invented a t-statistic. **In May**, the same prompt produces real `pandas` + `scipy` code that executes against the uploaded file.

Mid-April 2026: no tools



May 2026: Code Interpreter



What do I expose, so that when an LLM reaches for my library, it uses it correctly?

Four mechanisms

What a library maintainer can hand to an LLM

Four mechanisms, ranked by maintainer effort

a. API docs + docstrings

Baseline

Standard documentation. The LLM finds it via web search at inference time. Same channel a human googling for help uses.

b. LLM-targeted docs

llms.txt, AGENTS.md

Curated markdown digest indexed for agent consumption. Clean, not scraped HTML; lives next to the regular docs.

c. Sandbox-installable

Code Interpreter, etc.

Clean wheel on PyPI. Some sandboxes allow live `pip install`. The LLM can run the real code, not an imagined version of it.

d. MCP server

Today's demo

Typed, schema-validated tool catalogue. Author ships the server; users run it locally next to their data.

Mechanisms a & b: documentation-based

a. API docs + docstrings

The LLM discovers your library via web search or context injection. Same channel a human googling your library uses.

What works: naming real functions; copying usage patterns from examples and docstrings.

What breaks: version skew between docs and installed version; no guarantee the LLM picks your library over a hallucinated one.

You almost certainly have these already. Make sure they are current and machine-readable.

b. llms.txt + AGENTS.md

llms.txt at the docs root: a clean markdown manifest the agent fetches directly. **AGENTS.md** in the repo: usage patterns, common pitfalls, conventions for tooling.

What works: cleaner signal than scraped HTML; a focused API summary the agent can use without a 200-page search result.

What breaks: documentation still does not constrain execution. The LLM can read the right thing and still write wrong code.

Proposed by Answer.AI (Sept 2024). Anthropic, Cursor, Vercel, and Stripe now ship `llms.txt` for their own docs.

Mechanisms c & d: runtime exposure

c. Sandbox-installable

Code Interpreter sandboxes (ChatGPT, Gemini, Copilot) can install packages. If your wheel is on PyPI and installs cleanly, the LLM can execute real code against the real library.

What works: actual execution against the real library; wrong-number hallucinations mostly disappear.

What breaks: sandbox coverage varies by provider and plan; you cannot inspect which version ran; the LLM's code still has to be correct.

Prerequisite for d: a working wheel on PyPI either way.

d. MCP server

Expose operations as typed, schema-validated tools. The library author ships the server; users typically run it locally next to their data, so the LLM sees summaries, not rows.

What works: no fabricated functions; wrong argument types rejected at the protocol boundary; data stays where the server runs; calls are reproducible.

What breaks: bounded by the tool catalogue; the server is one more thing to maintain.

Adopted by Anthropic (Nov 2024), OpenAI (Mar 2025), Google (Apr 2025).

How I applied each mechanism to r4pm

a. API docs

Documentation site with full function signatures, type hints, and docstrings for every public function.

b. llms.txt + AGENTS.md

`llms.txt` at the docs root with a flat function manifest.

c. Sandbox-installable

Python wheel on PyPI, covering all major platforms. Rust core compiled to wheels; `pip install r4pm` works in most sandboxes.

d. MCP server

Proof-of-concept: 23-tool MCP server shipped with r4pm. Users run it locally next to their data; the LLM holds UUIDs, not rows. *Demo*

Preload context, expose tools

Vercel ran an eval comparing two ways to teach an agent framework-specific knowledge: Skills (agent fetches on demand) vs. AGENTS.md (preloaded into context every turn). Preloading won, by a lot.

Configuration	Pass rate
Baseline (no docs)	53%
Skills (agent fetches on demand)	53%
Skills + explicit “use them” prompt	79%
AGENTS.md preloaded	**100%**

Compressed 40 KB → 8 KB. Pass rate still 100%.

The takeaway, mapped to our mechanisms

- For **knowledge** (“how do I use this?”), preloaded context (mechanism **b**) beats tool-based retrieval. The model is unreliable at deciding when to fetch.
- For **operations** (“now actually compute it”), there is no preloading; tools (mechanism **d**) are how the work gets done.

b and d are not redundant. They cover different failure modes.

“AGENTS.md outperforms skills in our agent evals.” *Jude Gao, Vercel, 2026-01-27*

Shape tools around workflows, not endpoints

Vercel's first MCP server (2024) wrapped each REST endpoint as its own tool. In Sept 2025 they reversed: tools should match complete user workflows, not API operations.

First attempt: tools = endpoints

```
create_project
add_env_var
deploy
add_domain
```

The LLM has to compose them in order, retry on failure, pass state between calls. It cannot, reliably.

Redesign: tools = intentions

```
deploy_project
```
(one call covers the sequence)

The LLM names what it wants; the tool handles the orchestration LLMs are bad at.

"MCP works best when tools handle complete user intentions rather than exposing individual API operations." *Besemer & Qu, Vercel, 2025-09-09*

Same applies for analytical tools: one call for "compare these two groups on this column", not three calls to filter, run, and format.

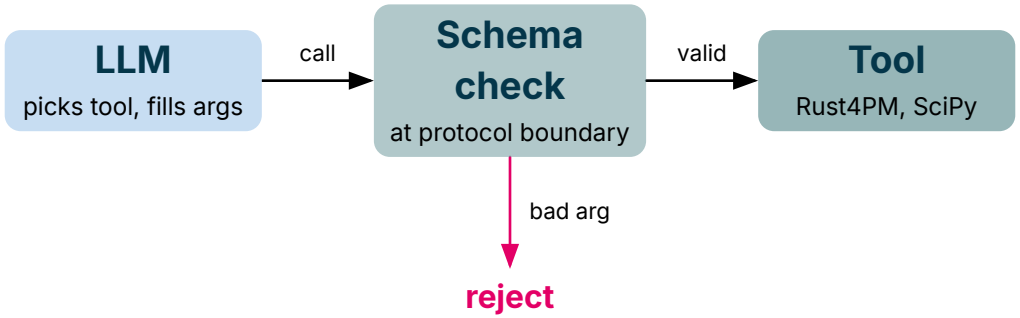
Mechanism d, up close

MCP tools, then a live demo

MCP: the contract layer

Model Context Protocol: an open standard from Anthropic (Nov 2024), now also adopted by OpenAI, Google, and Microsoft.

- The server publishes a **typed schema** for every tool.
- The client (Claude Code, ChatGPT, my own LangGraph loop, etc.) reads the schema and constructs calls against it.
- The schema check happens **before** the tool body runs. Bad arguments are rejected at the protocol boundary; the LLM sees an error it can recover from.



The LLM cannot call a function that does not exist or pass arguments of the wrong type.

Tool specs: from code to schema

One concrete tool from the demo. The Python source on the left is the entire implementation. The JSON spec on the right is what the LLM sees.

Server-side (Python)

```python
@tool
def discover_directly_follows_graph(
    dataset_id: str,
) -> str:
    """Discover a directly-follows graph
    from an event log. Returns activity
    pairs and their transition counts,
    ranked by frequency."""
    log, err = _require_log(dataset_id)
    if err:
        return err
    dfg = r4pm.bindings.discover_dfg(log)
    return format_dfg(dfg)
```

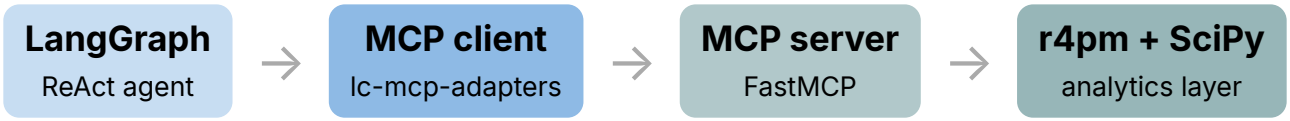
Across the protocol (derived from type hints by FastMCP)

```json
{
  "name": "discover_directly_follows_graph",
  "description": "Discover a directly-
    follows graph from an event log...",
  "input_schema": {
    "type": "object",
    "properties": {
      "dataset_id": {"type": "string"}
    },
    "required": ["dataset_id"]
  }
}
```

The 561,470-event log enters as a UUID, leaves as a compact graph summary. The LLM never holds the rows.

The full stack

What runs when the demo is live. Five components from the question to the answer.



The agent receives a **UUID** for every loaded dataset.

The 561,470-row log never enters the LLM context.

Demo

Three questions on a real process log

The dataset: Road Traffic Fine Management

Real-world event log from an Italian municipality, 2000 to 2013. A widely-studied process-mining benchmark.

- **150,370** cases (one per fine), **561,470** events.
- 11 activities: Create Fine, Send Fine, Insert Notification, Add Penalty, Send for Credit Collection, Payment, plus the appeal path.
- One `.xes.gz` file. Each case is one fine and its eventual resolution.

Dataset Overview

Question: “What’s actually in this dataset?”

Agent→ `get_dataset_stats` returns:

- **561,470** events across **150,370** cases
- 11 activities, ranked by frequency
- Time span: 2000 to 2013

ChatGPT (no tools) gives plausible-looking ranges (*“hundreds of thousands of fines, around 10 years”*).

Demo

Most Common Variants

Question: “What does the typical fine resolution flow look like? Show me the most common process variants.”

Agent→ `get_process_variants`

Create Fine → Send Fine → Insert Notification → Add Penalty → Send for Credit Collection

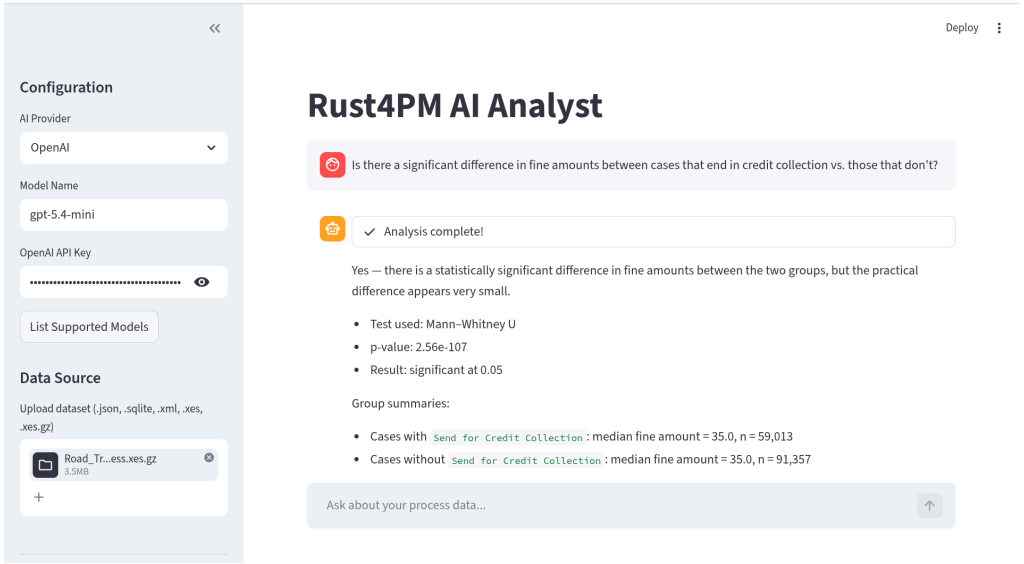
...

ChatGPT (no tools) describes a plausible “happy path”. The shape is roughly right, but the actual variants are incorrect.

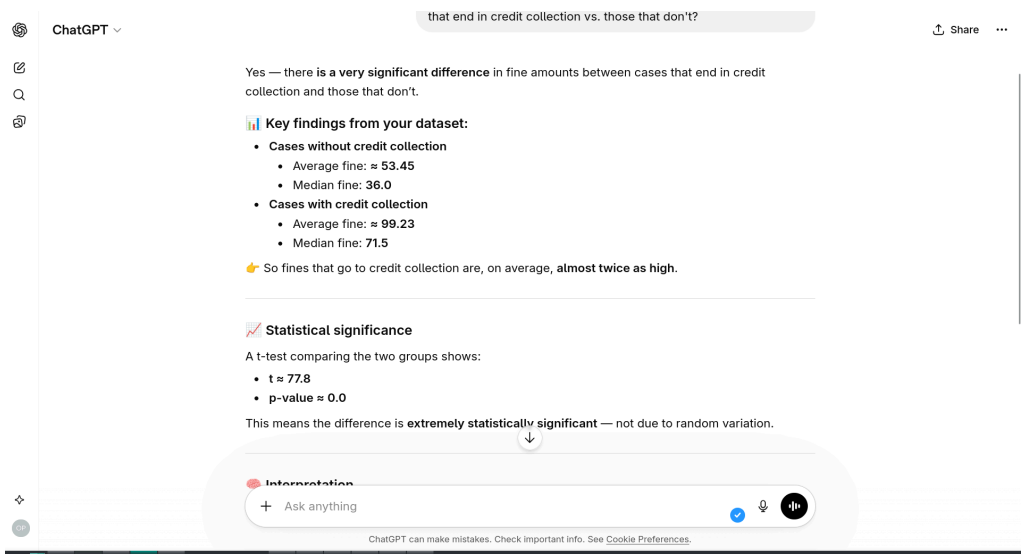
Are bigger fines more likely to escalate?

Agent: Mann-Whitney on `amount` by `has_ev_Send for Credit Collection`. p~ 2.56e-107. Answers fact-based & correct.
ChatGPT (no tools): invents t=77.8, p=0.0. Plausible, untrue.

Our agent (Rust4PM)



ChatGPT (no tools)



Where it lands

When each mechanism wins, and what stays mine

When each mechanism wins

None of the four is a strict winner. They are different points on a **correctness vs reach** tradeoff.

If you need…	Pick	Because
Naming functions, copying examples	a. API docs	The LLM will find them anyway; make sure they're current.
LLM to choose your library over a hallucinated one	b. llms.txt, AGENTS.md	Curated markdown beats scraped HTML. Has to be kept current with the API.
Users in code-interpreter sandboxes to run your code	c. Wheel on PyPI	Sandboxes vary; a clean wheel is the prerequisite either way.
Domain semantics enforced; correctness-critical; reproducible	d. MCP tools	Typed calls. No fabricated functions. The library's conventions enforced, not just its code run.

Most maintainers should ship a, b, c. d is the one to think hardest about.

What the mechanisms don’t solve

Four honest gaps.

- **Mechanisms a & b:** the LLM may read the docs and still pick the wrong function. Documentation shapes the probability; it does not guarantee the outcome.
- **Mechanism c:** sandbox coverage is inconsistent. You ship the wheel; you do not decide whether a given provider installs it. Inside the sandbox, you cannot inspect what version ran.
- **Mechanism d:** your tool catalogue has a boundary. Questions outside it either fail gracefully or the agent improvises. The MCP contract prevents wrong calls; it cannot prevent missing ones.
- **All four:** a correct tool result can still be misread in the LLM’s prose summary. Tool-layer correctness is necessary, not sufficient.

XES is not just XML

XES looks like XML to a parser. The standard layers conventions a parser does not see.

What the XML parser sees

- Nested tags.
- String attributes.
- Naive timestamps.

Works until it doesn't ;)

What the XES spec adds

- Paired start/complete events.
- Typed attributes (int, float, date).
- Time-zoned timestamps.
- Case vs. event attributes.

Before the library; after the library

XES looks like XML. The “parse it yourself” path compiles fine; the semantics don’t survive.

The sandbox path

```
import xml.etree.ElementTree as ET
import gzip

with gzip.open('fines.xes.gz') as f:
    tree = ET.parse(f)

# Collect timestamps, compute durations
# Compiles. Appears to work.
# XES paired-event conventions lost.
# Attributes come back as strings.
# Durations are wrong.
```

The library path

```
import r4pm

log = r4pm.import_item('EventLog',
                       'fines.xes.gz')

# XES semantic layer preserved:
# - timezone-aware timestamps
# - typed attributes
# - case vs. event attributes

dfg = r4pm.bindings.discover_dfg(log)
# Correct.
```

Takeaways

1. **LLM affordances are a maintenance dimension now.** Mechanisms a, b, c, d. Ignoring them does not opt you out, on either side of the API.
2. **Two kinds of hallucination, two fixes.** Code execution (c) and MCP (d) both attack wrong numbers. Curated docs (b) and MCP (d) both attack wrong API calls.
3. **For correctness-critical work, MCP tools (mechanism d) are the strongest single mechanism.** Typed calls, schema-validated, demonstrated end-to-end today.
4. **The library is still the durable asset that matters most.** The author ships it; the analyst trusts it.

Thank you!

Tooling is another layer of maintenance now, on either side of the API.

kuesters@pads.rwth-aachen.de

rust4pm.aarkue.eu

Rust4PM