penweir

PENWEIR RESEARCH · TECHNICAL REPORT 01

Protection *holds* under pressure

A stress test of the Penweir AI gateway: 123,600 requests, ten workloads and deliberately unreliable providers.

Robin Hana · Penweir

23 October 2026 Penweir 1.3.0

0
raw protected values reached a provider

≈123,600
requests across ten workloads and three baselines

0.4ms
median latency added at normal load (p95 1.0ms)

0
wrong restorations in completed answers, streamed or not

ABSTRACT

Penweir is a self-hosted gateway between company systems and language models. It detects structured identifiers such as email addresses, IBANs, Dutch citizen service numbers (BSN) and card numbers, and redacts, masks, hashes, tokenizes or blocks them according to a policy before a request leaves the network. It restores tokens in the model's answer, including streamed answers, and it logs metadata only.

We sent about 123,600 requests through ten isolated Penweir instances, against simulated language-model providers that failed on purpose. Workloads ranged from a three-step development pipeline to sixteen pipelines of 30–40 steps with 64 KB messages. An oracle at the provider side recorded every raw sensitive value it received.

No protected value reached a provider, and no completed answer contained a wrong restoration. At normal load Penweir adds a median of 0.4 ms and a 95th percentile of 1.0 ms to a 4 KB request, 25 times below the project's own target. When providers failed, fallback kept most of those faults away from clients. When the gateway ran out of capacity, it failed closed: requests were refused, never passed through unprotected. The test also located the limits of the current single-process design, and those set the priorities for the next release.

Key findings

0 leaks

Protection is correct at every load

No raw protected value reached a provider in about 123,600 requests: all five actions, field steps on tool results, custom detectors, long histories, 64 KB payloads and unreliable providers.

0 errors

Restoration is exact, also when streamed

Every tokenized value came back exactly in completed answers, including tokens split across uneven streamed chunks. 43,966 of the requests were streamed.

1.0 ms

Overhead you will not notice

At normal load Penweir adds 0.4 ms at the median and 1.0 ms at p95 to a 4 KB request with the reference policy. The target was 25 ms.

Fails closed

Overload never opens the gate

When capacity runs out, requests end in an error. None bypassed the filters. Provider fault rates of 2–3% at levels 2 and 3 led to 0% client failures, thanks to fallback.

CONTENTS

01 Why we tested
02 How Penweir handles a request
03 Research questions
04 Method
05 Results: protection under load
06 Results: overhead, resilience and resources
07 Capacity and sizing
08 What we are improving
09 Scope and limitations
10 Conclusion
A Appendix

01 Why we tested

Every AI integration is a new way out for your data.

An invoice summary, a support reply, a CRM export: each one can carry names, IBANs and phone numbers to a model the company does not run. Penweir sits in that path and holds those values back. For a gateway like this, three properties matter, in this order. It must never let a protected value through. It must not make every request noticeably slower. And it must keep working when the providers behind it do not.

Functional tests show that a filter works on one request. They do not show what happens with hundreds of concurrent clients, conversations of twenty turns, tool results full of customer fields, and providers that time out or return garbage. That is where shortcuts in a gateway show up: a detector skipped under load, a token restored into the wrong answer, a fallback that resends the unfiltered original.

So we pushed Penweir until it broke, on purpose, and watched what broke first. The short answer: protection never did. Capacity did, and it broke safely.

02 How Penweir handles a request

Penweir is an OpenAI-compatible gateway (`POST /v1/chat/completions`, streamed and not). Applications point at Penweir instead of at a provider. Every request then passes through eight stages (Figure 1).

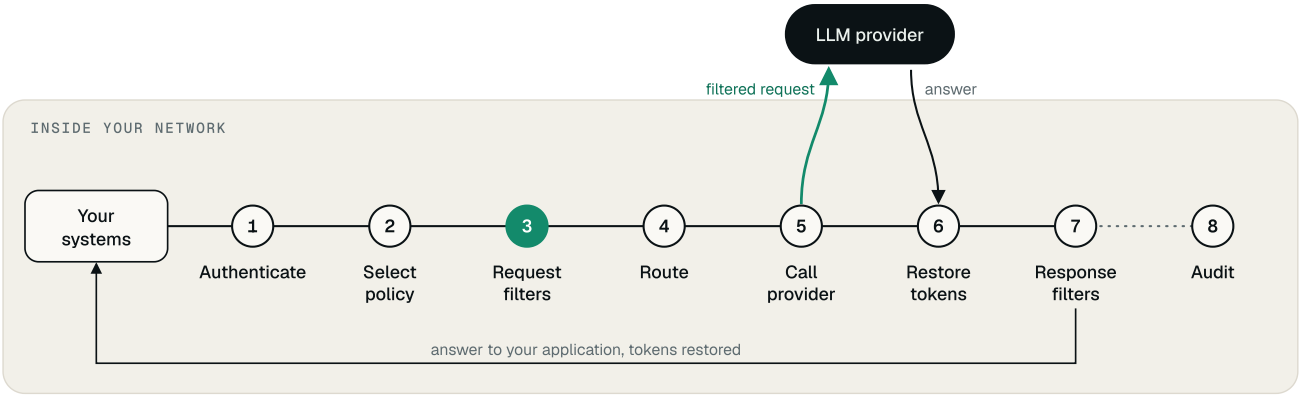


Figure 1. The eight stages of a request. Detection and the chosen action happen in stage 3, before anything leaves the network; only the filtered request crosses to the provider. Stage 6 restores tokens in the answer, also mid-stream, and stage 8 writes one metadata-only audit event per request, off the request path.

1. **Authentication.** The API key is mapped to an application.
2. **Policy selection.** The application's pipeline is chosen, or the default pipeline.
3. **Request filters.** Detectors run and apply one of five actions (Figure 2).
4. **Router.** Picks a provider from the allow-list: the preferred provider first, then the fallback order.
5. **Provider adapter.** Calls the provider, OpenAI-compatible or Anthropic.
6. **Stream de-tokenization.** Restores tokens in the answer, even when a token is split across server-sent event (SSE) chunks.
7. **Response filters.** For example, blocking a card number in the answer.
8. **Audit.** One event per request with counts and timings, never content.

ACTION	WHAT IT DOES	EXAMPLE OUTPUT
`Tokenize`	Replaces the value with a reversible placeholder, held in memory for one request and restored in the answer.	`[IBAN_1#a3f]`
`Mask`	Shows only part of the value, for example the last digits.	`•••• •••• •••• •••• 43 00`
`Redact`	Removes the value.	`[removed]`
`Hash`	Replaces the value with a stable keyed hash, so the same input always maps to the same output.	`h:9f2c…41ad`
`Block`	Refuses the request, or the answer, when a value must never leave.	`request refused`

Figure 2. The five actions a pipeline step can take on a detected value. Example input: the IBAN `NL91 ABNA 0417 1643 00`, a well-known test number. Outputs are illustrative.

The design choices that matter for a stress test:

- **One deployable.** A single async Python process (FastAPI on uvicorn), easy to run inside a company network.
- **Embedded database by default.** SQLite in WAL mode through SQLAlchemy 2, with the default async connection pool. A Postgres backend exists but was not part of this test.
- **Audit off the request path.** A background task writes audit events in batches of 200 from a bounded queue of 10,000 events.
- **Key cache.** A verified API key is cached in memory for 5 seconds; `last_used_at` is written at most once a minute per key.
- **Rate limits.** By default 600 requests per minute per key and 6,000 per minute per gateway. An optional concurrency limit per key is off by default.

03 Research questions

QUESTION	WHAT WE ASKED
RQ1 Overhead	How much latency does the gateway add to a typical request?
RQ2 Scalability	How do throughput and latency behave as concurrency and pipeline complexity grow?
RQ3 Resilience	How do fallback and retry behave when providers fail, rate-limit, time out or send unreadable responses?
RQ4 Protection	Does redaction stay correct under load and failure? No raw protected value may reach a provider, and tokenized values must be restored exactly.
RQ5 Audit	Does every request leave exactly one audit event?

04 Method

4.1 Environment

All runs used one Apple M3 Pro laptop (11 cores, 18 GB RAM, macOS 27.0.1). In the parallel phase, ten Penweir instances ran at the same time on this machine, each with its own load generator and simulated providers. Afterwards we ran three baselines on the idle machine, to separate Penweir's own limits from those of the shared machine. Software versions are listed in the appendix.

4.2 Test harness

The harness was written for this test. No real language model was called and no real personal data was used.

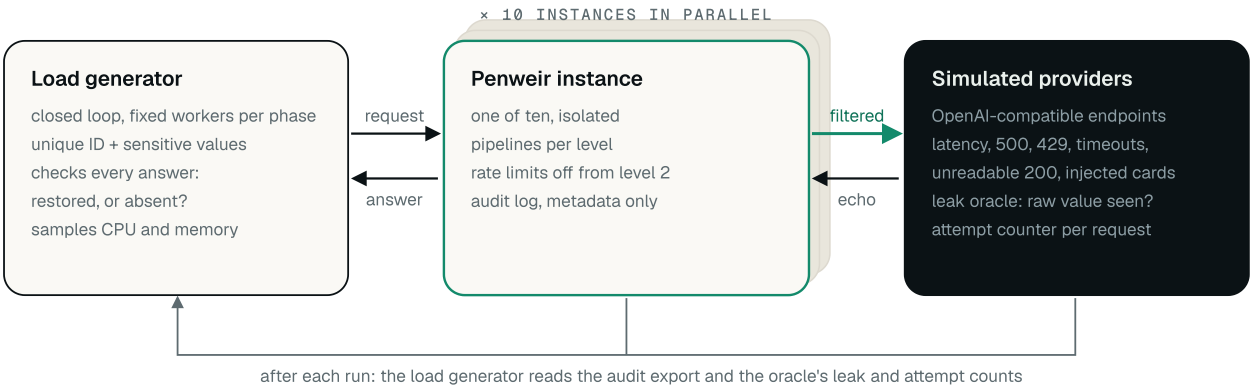


Figure 3. The test harness, repeated for each of the ten instances. The simulated providers echo the user's last message, so restoration can be checked on every answer, and they stream in uneven 3–11 character pieces so tokens are split across chunks. Their leak oracle records any literal raw test value they receive.

Test values were synthetic but valid wherever a checksum exists: email addresses, IBANs (mod-97), Dutch citizen service numbers (eleven test), payment cards (Luhn), phone numbers, Dutch VAT and Chamber of Commerce numbers, postcodes, IP addresses, credentials in a URL, an API-key-like secret, a customer name for field steps, and a custom project code.

4.3 Workloads

Ten levels, each more complex than the last (Figure 4). Each level was configured by its own automated test agent within given parameters, and each sent 10,000 requests. Faults were injected at the providers from level 2 onwards: server errors, 429 rate limits, timeouts, unreadable responses and card numbers slipped into answers.

LEVEL	PIPELINES × STEPS	STEPS PER PIPELINE	PROVIDERS AND FAULTS	PAYLOAD	STREAMED	CONCURRENCY PER PHASE
1	1 × 3 email tokenize, IBAN mask, BSN block		1 healthy	1–8 KB	30%	25 / 100 / 300
2	2 × ~5 1 response step		2 (2% errors on primary)	≤1 KB	30%	50 / 150 / 300
3	3 × ~8 field steps		3 (3% 429, 2% errors)	2 KB, tool results	40%	100 / 250 / 400
4	4 × ~10 all five actions		3 (1% timeouts at 2 s)	4 KB	40%	100 / 300 / 500
5	5 × ~12 2 custom detectors		3 (5% errors, 1% unreadable)	4 KB, 5-turn history	~31%	150 / 400 / 600
6	6 × ~15		4, fallback chains of 3	1–8 KB, 3-turn history	50%	200 / 500 / 800
7	8 × ~18 3 response steps each		4 (5% errors, 5% 429, 2% unreadable)	16 KB, 6 tool fields	50%	250 / 600 / 1,000
8	10 × ~22 3 custom detectors		5 (8% errors, 4% 429, 1% timeouts)	32 KB, 20-turn history	60%	300 / 700 / 1,000
9	12 × ~26		5, all faulty (10% errors, 5% 429, 2% timeouts, 2% unreadable)	48 KB, 10-turn history	60%	400 / 800 / 1,200
10	16 × 30–40 4 custom detectors		6, faulty as in level 9	up to 64 KB, 20-turn history	70%	500 / 1,000 / 1,500

Figure 4. The ten workloads. Each level sent 10,000 requests in three phases of rising concurrency. Level 1 is an exact copy of the development pipeline with the default rate limits; levels 2–10 turn rate limits off to measure raw capacity. The bar shows request steps per pipeline (level 10: 30–40).

4.4 What we measured

METRIC	DEFINITION
Leaks	Occurrences of a raw value, of a kind the pipeline protects, in what a provider received. Target: 0.
Content failures	Completed answers where a tokenized value was not restored, or a masked, redacted or hashed value was present.
Added latency	Penweir's own share of a request: `latency_ms - upstream_latency_ms` from the audit log, for non-streamed requests.[1]
Throughput and latency	Completed requests per second of wall time; end-to-end p50, p95 and p99 at the client.
Failure and retry rate	HTTP 5xx and connection errors as a share of requests sent; extra provider attempts per request.
Audit completeness	Audit events divided by requests sent.

05 Results: protection under load

This is the result Penweir exists for. It held at every level, under every fault, at every concurrency we tried.

0

LEAK ORACLE · ALL LEVELS AND BASELINES

No raw protected value reached a provider.

The simulated providers received 54,323 calls for 52,664 distinct requests and inspected every one of them. They found no protected value in its raw form, including on fallback attempts to a second, third or fourth provider.

MEASURE	RESULT
Leaks: raw protected value received by a provider	**0 across all levels and baselines**
Content failures on completed answers	0[2]
Detections recorded in the audit log	`182,932 (lower bound)`
Actions recorded in the audit log	`180,686`
Card numbers injected into answers	Blocked every time they reached a response filter, for example 37 of 37 at level 9 and 408 at level 5
Streamed answers	Tokens split across SSE chunks were restored exactly

5.1 What Penweir caught

Across the ten levels, the audit log recorded 182,932 detections of eleven named kinds plus several smaller ones (Figure 5). Email addresses, phone numbers and IBANs together made up nearly half. These counts are a lower bound: requests that were never audited, which we come back to in section 8, are not in them.

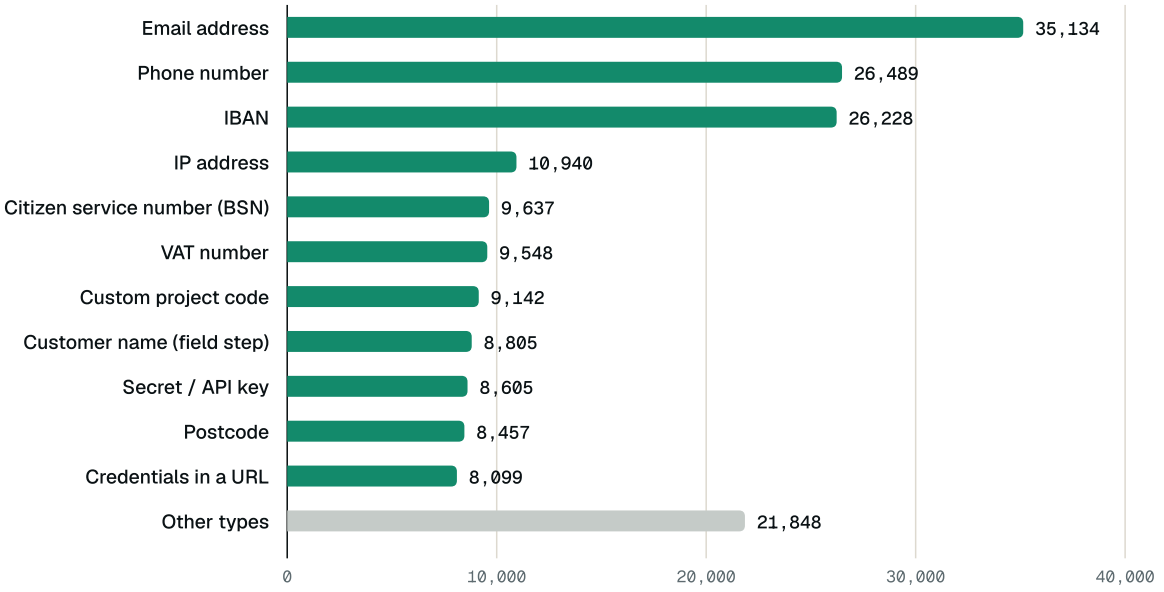


Figure 5. Detections by type, summed over levels 1–10. “Other types” (21,848) groups the remaining identifiers, such as card numbers and Chamber of Commerce numbers.

Each detection was followed by the action its pipeline prescribed (Figure 6). Tokenization was the most common action, which matters: a tokenized value has to come back exactly in the answer, and it did every time.

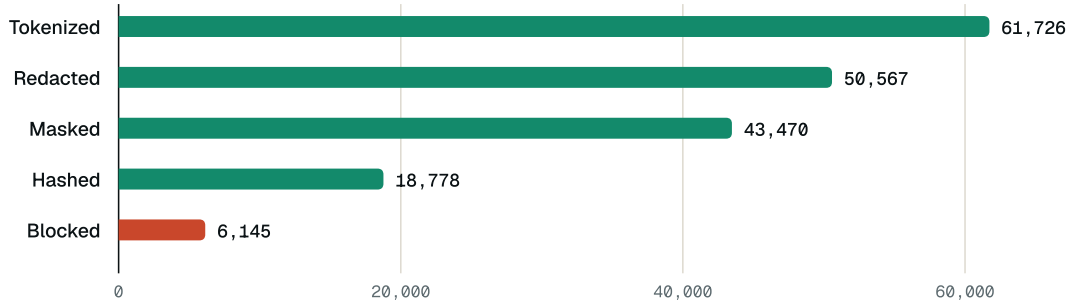


Figure 6. Actions taken on detected values. Blocked (6,145) is shown in vermilion, Penweir's colour for held values. In addition, 5,509 requests were refused by policy and 831 answers were stopped by a response filter.

5.2 Streaming without seams

Streaming is where token restoration usually gets hard. A model sends its answer in small pieces, and a placeholder such as `[EMAIL_1#a3f]` can be cut in two or three places. The simulated providers did exactly that, in uneven pieces of 3 to 11 characters, for 43,966 streamed requests. Penweir restored every split token exactly (Figure 7). When a response filter blocks content in the middle of a stream, the client receives an error event inside the open stream.

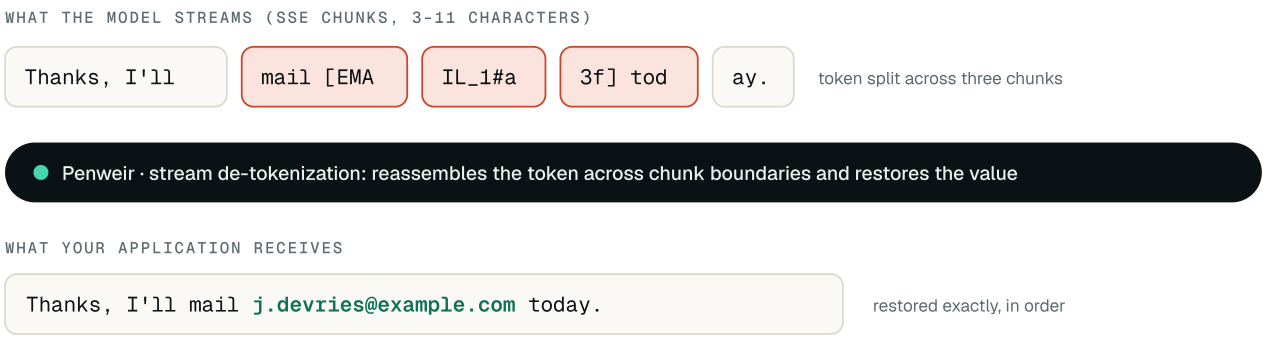


Figure 7. Illustration of stream de-tokenization with a synthetic address. The placeholder arrives split across three chunks; the application receives the original value, in place.

06 Results: overhead, resilience and resources

6.1 Overhead: about a millisecond

To measure Penweir's own cost we used the reference workload from the project's definition of done: a 4 KB prompt and the example policy (IBAN and email tokenized, BSN blocked, a customer-name field redacted, card numbers blocked in the answer), two providers answering in 5–10 ms, no streaming. Over 7,000 requests, Penweir added 0.4 ms at the median and 1.0 ms at p95 (Figure 8). That is 25 times below the 25 ms target. There were no failures, every request was audited, and nothing leaked.



Figure 8. Penweir's added latency per request (solo baseline B1, 7,000 non-streamed requests) on the same axis as the 25 ms target. The longest single request took 91.5 ms. The development pipeline of level 1 added even less: 0.2 ms at p50 and 0.3 ms at p95.

CONCURRENT CLIENTS	REQUESTS	THROUGHPUT	CLIENT P95
1	1,000	86 rps	14.5 ms
10	3,000	742 rps	21.3 ms
50	3,000	440 rps	308 ms

Baseline B1 by concurrency. Client p95 includes the simulated provider's 5–10 ms.

6.2 Resilience: provider faults stay invisible

Providers fail. They return server errors, rate-limit with 429, or time out. Penweir treats these as retryable and moves the request to the next provider in the pipeline's order, with the same filtered content. At levels 2 and 3, providers failed on 2–3% of calls and clients saw 0% failures. At level 9, where every provider was faulty, about 141 requests succeeded only through the chain, some after four attempts.

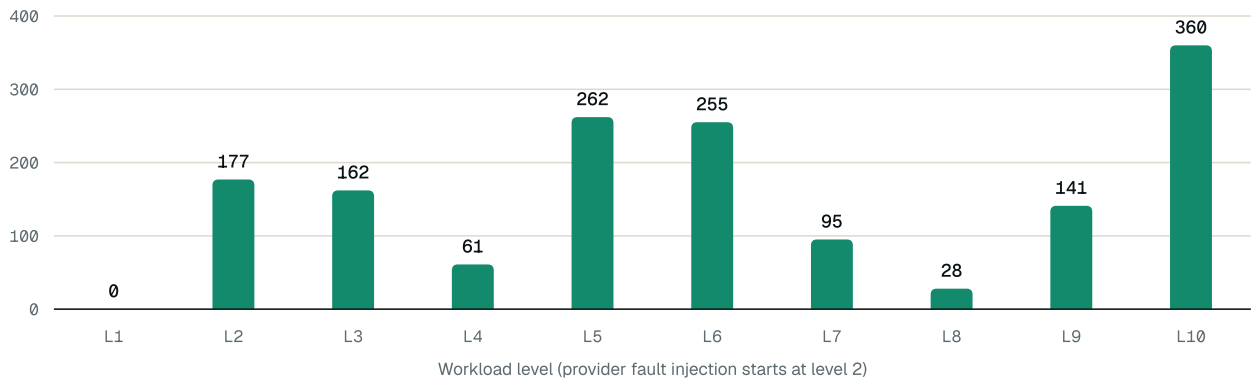


Figure 9. Requests that were sent to more than one provider after a retryable fault, per level. In total 1,541 requests used fallback, which accounts for the 1,659 extra provider calls (54,323 calls for 52,664 distinct requests).

0%
client failures at levels 2–3, with 2–3% of provider calls failing

1,541
requests were sent on to a second provider or further

≈4,700 rps
rate at which Penweir refuses rate-limited requests, at negligible cost

Refusing is cheap. At level 1, with default limits, Penweir refused 7,600 requests above the per-key limit at about 4,700 per second, without spending capacity on them.

6.3 Failing closed

The most important behaviour of a gateway shows when it is overwhelmed. We drove levels 6 to 10 far beyond the capacity of one process. Requests then queued, timed out or were refused. But every request ended in one of two states: a protected answer, or an error. Not one was passed to a provider unfiltered (Figure 10).

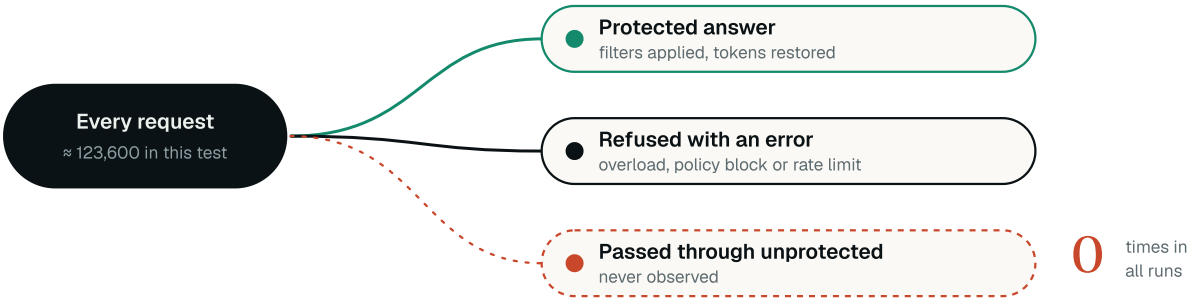


Figure 10. Possible outcomes of a request. Under overload Penweir returns errors; it never falls back to passing a request through without its filters.

6.4 Resources: small and steady

Memory stayed between 116 and 206 MB at every level, with no growth over a run, even with 64 KB messages and 1,500 concurrent clients. The process used one CPU core, which is the subject of the next section.

116–206 MB
memory at every load tested, no upward trend

1 core
86–96% CPU on average: one process, one core

64 KB
largest messages, with 20-turn histories, still filtered correctly

07 Capacity and sizing

Penweir 1.3.0 runs as one async process. Detection is CPU work, and every detector scans every message in the conversation, so cost grows with context length times the number of steps. Measured alone on the test machine, one process handled about 740 requests per second with the reference policy, about 340 with the level 2 pipelines, and 23–37 with 18 steps on 16 KB messages with history (Figure 11).

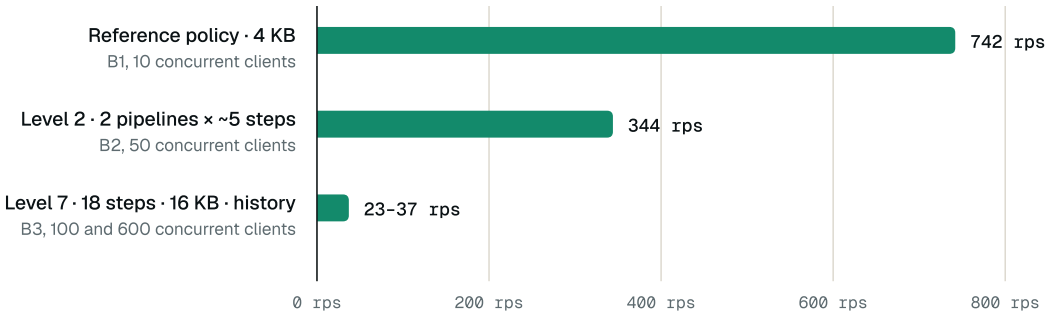


Figure 11. Throughput of one Penweir process on one CPU core, by pipeline cost, from the solo baselines. These are measurements on a laptop shared with nothing else; a server core and the planned multi-worker mode change the absolute numbers, not the shape.

● SIZING GUIDANCE FOR 1.3.0

Penweir is designed for one container per company, serving internal applications. For that deployment, one instance serves on the order of tens to hundreds of requests per second, depending on pipeline cost, at about a millisecond of added latency. Bursts of several hundred concurrent clients against one instance call for the improvements in section 8.

08 What we are improving

A stress test is only useful if it also tells you where to work. Penweir is in development, and this one gave a clear list. All of it concerns capacity and record-keeping under overload. None of it concerns the correctness of protection.

From about 250 to 500 concurrent clients, depending on pipeline cost, requests began to fail with internal errors. The cause is a cache stampede: when a cached API key expires, every concurrent request with that key queries the database at once, and SQLAlchemy's default pool of 5 connections plus 10 overflow runs dry behind SQLite's single writer. Below that threshold the audit log held exactly one event per request (levels 1–3: 10,000 of 10,000). Above it, two kinds of events went missing: requests that failed during authentication, and audit batches that failed to write. For a product whose audit log is a compliance feature, that gap comes first.

#	WHAT WE FOUND	WHAT CHANGES
1	Key-cache stampede exhausts the database pool from a few hundred concurrent clients	Only one request reloads an expired key; the pool is sized explicitly
2	Internal errors during authentication, and failed audit batches, leave no audit event	Audit every request with a valid key, including internal errors; count failed batches as dropped
3	No load shedding: past the knee, requests queue for up to ~30 s instead of being refused	A concurrency cap that answers fast with `503` and `Retry-After`
4	One process on one core sets the throughput ceiling	Several worker processes per instance

We will also document three deliberate behaviours: an unreadable HTTP 200 from a provider is not retried on another provider (143 times in this test); a response blocked mid-stream arrives as an error event inside an HTTP 200 stream; and the default limit of 600 requests per minute per key is easy to reach for a single busy application.

09 Scope and limitations

- **One shared laptop.** Ten instances, their providers and load generators shared 11 cores, so absolute throughput in the parallel phase is pessimistic. Solo baselines showed the same limits and failure modes.
- **macOS networking.** A listen backlog of 128 inflates connection resets compared with a typical Linux server (often 4,096).[3]
- **Simulated providers.** They echo the input with uniform latency and independent random faults. Real providers fail in bursts and send longer answers.
- **The oracle checks literal values.** It catches exact raw values, the password part of a URL and the digits of a Chamber of Commerce number, but would not catch a value altered by a faulty partial transformation. Detector recall on realistic free text was out of scope.
- **Closed-loop load, one run per level.** No confidence intervals; per-phase figures rest on 1,000–6,000 requests each.
- **SQLite only.** The Postgres backend, with its own pooling and concurrent writes, was not tested.

10 Conclusion

Across about 123,600 requests, ten workloads and providers that failed on purpose, Penweir never let a protected value through and never restored a token into the wrong place.

It added about a millisecond to a normal request. It hid most provider failures behind fallback. And when we pushed one process far past what one core can handle, it failed closed. For the deployment Penweir is built for, one container per company serving internal applications, that is the guarantee that matters: the data you decided to hold back stays back, whatever the load.

The same test drew a precise map of what to build next: a stampede-proof key cache, an audit log that records every request even under overload, fast refusals instead of long queues, and more than one worker per instance. Those are engineering problems with known solutions, and they are next on the list.

PENWEIR · IN DEVELOPMENT · EARLY ACCESS OPEN

Hold the line.

Penweir is a self-hosted AI gateway. It runs inside your network, keeps no content and logs metadata only. Early access is open now.

Visit penweir.com

stored: 0 bytes

NOTES

1. For streamed requests `upstream_latency_ms` runs only to the first byte, so added latency is reported for non-streamed requests. For requests that used fallback it includes the failed attempts. The figure in section 6.1 uses non-streamed requests without fallback. ↑
2. At level 3 the harness reported 148 content failures. They came from a wrong expectation in the test: the pipeline hashes email addresses, while the scenario expected them to be restored. They were excluded. ↑
3. Connection errors totalled 22,052 over levels 1–10, mostly resets of connections beyond the listen backlog, past the capacity knee. Those requests never reached the gateway, so they carry no protection risk. ↑

A Appendix

A.1 Volume

SOURCE	REQUESTS
Levels 1–10 (parallel)	100,000 (43,966 streamed)
Agent smoke runs and one discarded run	~1,300
Harness check and three solo baselines	22,300
Total	**≈ 123,600**

A.2 Glossary

Tokenize	Replace a value with a reversible placeholder, for example `[EMAIL_1#a3f]`, held in memory for one request and restored in the answer.
Mask	Show part of a value, for example the last digits.
Redact	Remove the value.
Hash	Replace the value with a stable keyed hash.
Block	Refuse the request, or the answer.
Field step	A rule on a JSON key, such as `customer_name` in a tool result.
Fallback	Sending the request to the next provider in the pipeline's routing order after a retryable failure.
Knee	The concurrency at which throughput stops rising and latency starts to grow sharply.

A.3 Versions and reproducibility

- Penweir 1.3.0, commit `0dc4a63`.
- Python 3.13.15, FastAPI 0.143.0, uvicorn 0.54.0, SQLAlchemy 2.1.4, httpx 0.28.1; Node 26.8.2 for the harness.
- Per-level reports and raw metrics live in the Penweir repository under `docs/reviews/2026-10-23-stress/`. The harness (`fake-llm.mjs`, `loadgen.mjs`, `setup.mjs`, `start-stress.sh`, `values.mjs` and per-level configurations) will be published under `scripts/stress/`.
- The reports contain no request or answer text and no API keys: only counts, timings and status codes.