

Reforge

Legacy code, re-forged.

An autonomous code-migration and vulnerability-remediation agent. It upgrades Python repositories across breaking dependency releases, patches critical CVEs, verifies every change in an isolated sandbox, self-heals failing tests, and proves behavioural parity.

01 · CORE PURPOSE

Core Purpose

Upgrade legacy Python code across breaking dependency releases, patch the critical CVEs that come with old pins, and prove that behaviour did not change.

Major releases change dozens of APIs at once. The fix is mechanical but wide, so upgrades get postponed for years and vulnerable versions stay pinned in production. Generic AI coding tools rewrite code but do not prove the result behaves the same, and sending every edit to the largest model is slow and costly.

Reforge is an autonomous migration agent that closes that gap: it finds every affected call site, breaks the code under the new release on purpose to see what really fails, grounds each fix in the official migration guide, patches with a cost-aware router of NVIDIA Nemotron models, self-heals failing tests and hands back an audit a reviewer can sign off on.

What goes in, what comes out

Input	Output
A public GitHub repository URL (or one picked via <i>Connect GitHub</i>)	The refactored source files, viewable and downloadable
Pydantic v1 code pasted into the chat bar	<code>reforge.patch</code> — one git-apply-ready unified diff
A <code>.py</code> file upload	An audit PDF: CVEs resolved, docs used, performance matrix, AI cost ledger, side-by-side diffs

02 · TARGET

Target

First migration target: Pydantic v1 → v2 — one of the most widely used libraries in Python, and one of its most disruptive upgrades.

What gets migrated

Pydantic v1	Pydantic v2
<code>@validator · @root_validator</code>	<code>@field_validator · @model_validator</code>
<code>class Config</code>	<code>model_config = ConfigDict(...)</code>
<code>__root__ models</code>	<code>RootModel[...]</code>
<code>parse_obj · parse_obj_as</code>	<code>model_validate · TypeAdapter</code>
<code>.dict() · .json() · .copy() · .schema()</code>	<code>model_dump · model_dump_json · model_copy · model_json_schema</code>
<code>Field(regex=, max_items=)</code>	<code>Field(pattern=, max_length=)</code>
Pinned <code>pydantic==1.x</code> with known advisories	Pinned to the current 2.x release, CVEs re-checked on OSV

Who it is for

- Teams holding a codebase on Pydantic v1 because the upgrade is too wide to do by hand.
- Security owners who need a known-vulnerable pin removed with evidence that nothing broke.
- Reviewers who want every change traced to documentation, tests and a cost ledger.

Scope check before any work

Reforge inspects the input first. If nothing imports Pydantic, or the code is already on Pydantic v2, it stops immediately with a clear message — no tests are run and no model credits are spent. The pipeline itself is library-agnostic and built to take on further dependency upgrades.

03 · RELIABILITY

Reliability

Nothing is trusted on faith: every change is verified by running the code, before and after.

- **Baseline first.** The test suite runs under v1 to record what passing looks like.
- **Break on purpose.** The new release is installed in an isolated sandbox and the real tests show what actually fails.
- **Grounded changes.** Each plan step cites the official migration documentation it came from.
- **Safe edits.** Hunks must match exactly once, must parse (AST gate) and are written atomically with a full backup and one-command restore.
- **Self-healing.** Failing tests after patching trigger a Super → Ultra heal ladder until they pass or the budget stops it.
- **Parity proof.** A separate verifier runs the original and refactored trees side by side: test outcomes, return values, model schemas and endpoint output.
- **Tests are sacred.** Original tests are snapshotted and restored for verification; any edit to them is flagged.
- **No code retention.** Workspaces, audit output and LLM caches are purged on server start and shutdown; telemetry stores token counts only.

Result on the reference repository

legacy-system-repo · pydantic 1.10.12 → 2.14.1

6/6

tests passing after refactor,
no regressions

100%

output parity, original vs
refactored

1/1

known advisory resolved

12/12

plan steps applied

04 · ARCHITECTURE

Architecture

Six stages, one request — streamed live to the web app as an execution trace.

#	Stage	What happens	Module
0	Ingest	Normalise the input; clone or refresh the repository, or stage the snippet with a generated golden-output test; scope check.	endpoints/endpoint.py · web.py
1	Scan & baseline	Detect the pinned dependency, query OSV for CVEs, run the test suite under v1.	extract/scanner.py · harness.py
2	Break & slice	Run the tests under v2 in a sandbox venv; merge with a static AST scan into a prioritised, import-graph-ordered payload.	sandbox/sandbox.py · slicer.py
3	Ground & plan	Retrieve official docs per breaking symbol (Tavily); one Nemotron 3 Ultra call writes a validated, cached refactor plan.	planner/grounder.py · planner.py
4	Patch & self-heal	SEARCH/REPLACE hunks with Lightning → Super → Ultra escalation, then a v2 test run with a heal ladder.	route_patch/patcher.py · switchyard.py
5	Prove & report	Parity verifier, CVE re-check and the audit PDF.	audit/verifier.py · report.py

Stack

Layer	Technology
Models	NVIDIA Nemotron 3 Ultra, Nemotron 3.5 Super VL, Nemotron 3.5 Lightning via Nebius Token Factory (OpenAI-compatible API)
Backend	Python 3.12 · FastAPI · uvicorn · uv-managed sandboxes · pytest + json-report · OSV.dev · Tavily · ReportLab
Web app	Zero-build HTML/CSS/JS served by the same FastAPI app; NDJSON streaming of the execution trace

05 · EFFICIENCY

Efficiency

The largest model plans once; the smallest model does most of the typing.

Tier	Model on Nebius Token Factory	\$ / 1M in · out	Role
Ultra	NVIDIA Nemotron 3 Ultra · 550B MoE	\$1.00 · \$3.00	Writes the refactor plan; last resort for patches and heals
Super	NVIDIA Nemotron 3.5 Super VL · 120B MoE	\$0.30 · \$0.90	Escalation when Lightning misses; first rung of the heal ladder
Lightning	NVIDIA Nemotron 3.5 Lightning · 30B MoE	\$0.06 · \$0.24	Primary patcher — batched per file, thinking off for speed

- Every dispatch is logged to a cost ledger next to its all-Ultra counterfactual.
- A per-run Stage-4 budget (default \$0.50) caps spend; unavailable tiers are skipped for the rest of the run.
- Plan and patch caches make reruns deterministic and free; files are patched concurrently.
- The scope check stops runs with nothing to migrate before any credits are spent.

Measured on the reference repository

\$0.12

full run, vs \$0.32 on Ultra alone

62%

cheaper than Ultra-only

88.5%

saved in the patch stage alone

5.28×

faster validation hot path under v2

Reforge — legacy code, re-forged.