

RAFT: *Gradual Typing, Invariance Enforcement, and Property Verification in Research Python*

The Secure Program Synthesis Hackathon - Apart Research

Antonin Peronnet - Thomas Wnninger



Abstract. AI researchers prototype in short scripts or Jupyter notebooks and push directly to state-of-the-art frameworks like `transformers` or `vLLM`. They lack the time for rigorous testing or manual verification. Moreover, the nature of their work makes traditional test or spec-driven development difficult to apply. We propose an automated, iterative pipeline to bridge the gap between quick experimental scripts and shareable, verifiable code, utilizing AI and strict static analysis to gradually introduce typing, contracts, and property verification. We validate our results with a backdoor detection experiment. A small reviewer `gemma4 e4b` is more efficient at finding bugs and backdoors after the application of our method. Going from 46% recall on vanilla code, to 95% detection on the `transformers`'s library.

01 | Introduction

Research code often lacks typing, and testing, which contrasts to its quick adoption in state-of-the-art frameworks like `transformers` [1] and `vLLM` [2]. Moreover, the gap between an experimental script and a verifiable artifact is filled with implicit knowledge that currently only lives in the researcher's head, thus, we cannot fully outsource cleanup to AI agents, no external tool knows the underlying specification.

Without a clear method, applied step by step by the agent alongside the researcher, hallucination rate spikes. And clarity can be hard in a world where Python typing paradigms are evolving quickly, leaving standard practices fragmented. Nonetheless, research code needs to stay flexible, thus, heavy verification frameworks are out of scope.

Our solution is a CLI tool that orchestrates existing static tools (`ruff`, `pyrefly`), enforces strict layout conventions, and guides both the researcher and LLM agents in graduating code from raw experiments to robust, verifiable Python.

02 | Method

Input: Jupyter notebooks, experimental Python scripts, and research libraries.

Pipeline:

- **Stage 1 (Static Constraints):** Apply strong, precise typing (`jaxtyping`), `ruff` with strict rules (`bugbear`, `bandit`), `pyrefly`¹, and move all implicit invariants into module/function docstrings.
- **Stage 2 (Runtime Contracts):** Add assertions and runtime contracts following PEP 316. Introduce `crosshair` for bounded verification.
- **Stage 3 (Property Testing):** Implement property-based testing with `Hypothesis` to test both individual functions and end-to-end workflows.

Output: Intermediate Python that is comprehensible, scalable, and mechanically easier to verify.

Kind	UNCONSTRAINED	CONSTRAINED
LINTERS	<code>ruff</code> defaults	<code>ruff</code> (<code>SECURITY</code> , <code>BUGBEAR</code> , <code>BANDIT</code>), <code>pyrefly</code>
RUNTIME CHECKS	NONE	<code>beartype</code> , <code>pydantic</code> STRICT
TYPING	PLAIN <code>Tensor</code>	<code>Float[Tensor, "b d"]</code> (<code>JAXTYPING</code>)
DATACLASS	PLAIN	FROZEN, SLOTTED AND KW-ONLY
COLLECTIONS	<code>list</code> , <code>dict</code>	<code>tuple</code> , <code>Mapping</code>
DISCRIMINATORS	<code>kind: str</code>	<code>Literal[...]</code>
INVARIANTS	IN ARCHITECTURE.md	IN DOCSTRINGS
LAYOUT	FLAT	PURE core/ WITH IMPURE runners/

Table 1: Difference in the substrate.

¹Citations for the tool used are: `Jaxtyping` [3], `Ruff` [4], `Pyrefly` [5], `Beartype` [6], `Hypothesis` [7], `Crosshair` [8].

03 | RAFT: the tool

We made a small CLI that turns the staged method above into a gated pipeline. It walks a codebase through three stages in a fixed order and refuses to advance until the current one passes, so hardening stays incremental: the researcher confronts one class of defect at a time rather than the whole gap between script and artifact at once.

In one sentence: it's a guide combining existing tools and agent skills to facilitate the transition between experimental code and reproducible, trusted code.

■ 1 A gated, three-stage pipeline

We mirror each stage directly: *clarity* realizes Stage 1 (static constraints), *contracts* Stage 2 (runtime contracts), and *properties* Stage 3 (property testing):

Stage	Tools	Gate
clarity	ruff, pyrefly	types, style, hygiene
contracts	annassert, crosshair	every non-trivial public function carries ≥ 1 run-time assertion
properties	pytest + hypothesis	output invariants, not single-input smoke tests

`rafty check` runs the stages in order and stops at the first failure; `rafty stage <name>` runs one in isolation. On failure RAFT does not dump every warning — it aggregates counts across the failing tools and surfaces the single file with the most issues, giving one concrete place to start rather than an intimidating list.

■ 2 Configuration and skills

All configuration lives in `pyproject.toml`. Each stage is a name, a list of shell commands, and a pointer to a *skill* file:

```
1  [tool.rafty.stages.clarity]
2  skill = "clarity.md"
3  tools = ["uv run ruff check src/", "uv run pyrefly check src/"]
4
5  [tool.rafty.stages.contracts]
6  skill = "contracts.md"
7  tools = ["uv run annassert check src/ --require-assertions",
8           "uv run crosshair check src/"]
9
10 [tool.rafty.stages.properties]
11 skill = "properties.md"
12 tools = ["uv run pytest tests/ --cov=src/ --cov-fail-under=10"]
```

toml

Skills are plain-markdown files copied into `.skills/` by `rafty init`, one per stage plus an entrypoint `rafty.md`. Each explains what its stage checks and how to approach it. Crucially they are the *same* context whether the reader is a human collaborator or an LLM agent:

the implicit specification that otherwise “only lives in the researcher’s head” is written down once, next to the gate that enforces it.

■ 3 **annassert: assertions as types**

Talking with researchers, we found that many didn’t use strong typing because of a lack of knowledge, and confusing conventions. They usually preferred to use simple `assert`. So, we designed a prototype called `annassert`, which is a custom linter built for the *contracts* stage. It flags any function above a McCabe complexity threshold that carries no runtime assertion: if you cannot state a single invariant about a non-trivial function, you have not pinned down what it does — or it does too much. It then lifts the structural part of those assertions into the type system as `Annotated[...]` metadata, so a static checker can reject violating *callers* without ever running the code:

```
1  # input
2  def repeat(s, n):
3      assert isinstance(s, str) and len(s) > 0
4      assert isinstance(n, int) and n >= 0
5      return s * n
6
7  # after annassert
8  def repeat(s: Annotated[str, MinLen(1)], n: Annotated[int, Ge(0)]):
9      assert len(s) >= 1
10     assert n >= 0
11     return s * n
```

 Python

This is the bridge between Stage 2 and Stage 1: the `isinstance` checks collapse into the annotation, the value predicates survive as both a (simplified) runtime assertion and static metadata. The “typing resolves bugs for free” effect we report in the experiments is here applied deliberately rather than incidentally.

■ 4 **Invocation**

RAFT exposes a Click CLI (`init`, `check`, `stage`, `extract`, `list`, `skills`) and a mirrored Python API, `CodeBase`, for driving it from a notebook or agent. `rafty extract <function>` is the inspection primitive: it reconstructs a function’s full contract — signature, assertions, and the property tests that cover it — in a single view.

```
1  pyproject.toml
2  └─ load_config() → RaftyConfig
3      └─ run_stages() → StageResult (one per stage)
4          └─ run_tool() → ToolResult (one per shell command)
```

04 | **Experiments: backdoor detection**

To find the set of tools and convention we would use in RAFT, we took existing transformers code, along with synthetic code bases, and applied our method. To grade the different steps, we used the backdoor detection task: A small LLM reviewer is asked

to scan a repository and find bugs, exploits, and hidden backdoors. This way, we can measure both how going from stage to stage naturally reduces bugs — just adding strong typing resolves multiple bugs, “for free” — but also how the remaining work can be done with a LLM. Highlighting how **few good conventions and correct tool use can drastically improve research code, while being cheap and easy to apply**.

■ 1 Methodology

1.1 Dataset

We used a corpus of 7 small repositories (a mix of extracted-and-harnessed HuggingFace transformers functions and self-contained interpretability playgrounds), each carrying calibrated injected backdoors. We then rewritten the repositories using our method to get both the original version (stage 0) and our version (stage 1).

Stage	Substrate
stage0	Basic type hints, <code>Dict[str, Any]</code> , plain docstrings, flat modules. Invariants — when stated at all — live in README/ARCHITECTURE.md, far from the code that must uphold them.
stage1	Frozen dataclasses / pydantic models, jaxtyping tensor shapes with <code>@jaxtyped(typechecker=beartype)</code> , <code>core/-runners/ split</code> , and explicit Invariants: / Conventions: blocks in module docstrings , local to the code.

The backdoors span five difficulty levels: L1 local-syntactic (e.g. `eval` on data), L2 local-semantic (e.g. an always-pass subset), L3 multi-file flow (producer in one file, consumer in another — neither line wrong alone), L4 hidden in-module state (call-count time bomb, `env kill-switch`), L5 stated-contract violation (docstring states a rule, body quietly contradicts it).

We ensured behavioral identity (`stage0` $\equiv$ `stage1`) was verified for every repo on clean inputs and on every trigger, so that any difference in detection is attributable to the substrate, not to the bug being accidentally fixed or made trivial during the rewrite. Indeed, we do not need to prove that using strong typing, contracts and hypothesis catches errors.

1.2 Review strategy

Orthogonal to the substrate, we vary how code is presented to the reviewer:

- **whole** — the entire codebase (every `.py`, line-numbered, plus README/ARCHITECTURE if present) is dumped in one holistic call; the model finds everything itself.
- **narrow** — one bundle per public function (its source + module docstring), reviewed in several focused passes with different questions: “are the types respected?”, “are the types narrow enough?”, “are the invariants respected?”, “is the function doing what is said in the docstring?”, ...

This deliberately confounds substrate with prompting (`stage0` defaults to `whole`, `stage1` to `narrow`); so we ran the full 2×2 experiment grid for ablation.

1.3 Reviewer and scoring

The reviewer is `gemma-4-E4B-it` [9] (Q4_K_M) served locally via `llama.cpp` [10] `llama-server` (OpenAI-compatible, port 8080), driven through `instructor` [11] in JSON mode against a pydantic Findings schema. Decoding is greedy (temperature 0).

The primary metric is **agent-recall** = caught / agent-eligible backdoors. A backdoor is caught when a finding matches its file and (a) lands within a line window of the anchor and (b) — for anchors that carry a distinctive token — mentions that token in the rationale. This strict matching (token required, not just proximity) was added after we found a scoring artifact.

■ 2 Results

	whole	narrow
stage0 (loose)	46% (11/24)	71% (17/24)
stage1 (structured)	60% (12/20)	95% (19/20)

Pooled agent-recall, strict scoring. Denominators differ because stage1's eval backdoors are statically caught and excluded from the agent pool.

Interestingly, `stage1-whole` (60%) < `stage0-narrow` (71%). A one-shot dump of `stage1` code loses to per-function review of `stage0` code. So the dominant lever is narrow per-function decomposition, not the structured substrate. This confirms and quantifies an earlier observation that a naive one-shot dump hurts conventions when context grows and is poorly managed. Thought this result might not be true when using large more capable models.

However, `stage1` does improve clarity and detection when used correctly. Specifically: it enables multi-turn detection, asking the model one question at a time, which helps models to focus and reduces hallucinations.

More importantly, forcing the reviewer to focus on small questions enables quick human verification. In experimental, research code, there might be a large part of the code that we do not want tests on, that we “accept” as experimental. This incremental hardening might be one cheap solution to do that: help the researchers harden the part of the code that needs to be hardened, and keep the experimental code experimental.

05 | Lessons Learned

- **Alignment against tooling:** LLMs trained with RL against linters and test runners learn the *weaknesses* of the tools, not the human intent.
- Common failure modes include aggressively injecting `# type: ignore` to pass checks, writing tautological tests (`assert True`), and generating technically compliant but semantically useless docstrings.

- **Defense in depth:** Relying on a single metric (like unit test coverage) is highly susceptible to LLM “bullshit.” Combining strict structural rules, varied static analyzers, and guided CLI workflows is required to catch evasive bugs.

06 | Non-Goals

- **Trivial Scripts:** Research scripts under 100 lines do not require this overhead.
- **Standard Software Engineering:** Developers building standard web apps can use out-of-the-box mature tooling; RAFT targets the messy middle ground of mathematical/AI research code.
- **Floating-Point Verification:** Formal methods verify the plumbing; property tests validate the numerics; experiments validate the science. Verifying PyTorch numerics is the domain of abstract interpretation, not RAFT.

07 | Limitations

- **Model Constraints:** The framework requires at least a medium-capability LLM to handle the coding syntax.
- **Experiment Scale:** The backdoor corpus relied on backdoors both synthetics and isolated sub-modules rather than massive, globally entangled codebases.
- **No Execution Sandboxing:** The framework does not currently enforce a secure execution environment for the generated artifacts during the transition phases.

08 | Future Work

- Once this is done, we can move on to more rigorous tools like TLA+ [12] / Dafny [13] and even start proving important pieces of the code and experiments with formal provers like Lean [14].
- Using abstract interpretation [15] along property testing. Whilst property testing is sound for refutation and incomplete for proof, abstract interpretation is sound for proof and incomplete for refutation. Tools like auto_LiPRA [16] or CROWN [17], [18] could be used to verify certain properties on AI models.

09 | Bibliography

- [1] T. Wolf *et al.*, "Transformers: State-of-the-Art Natural Language Processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: <https://aclanthology.org/2020.emnlp-demos.6/>
- [2] W. Kwon *et al.*, "Efficient Memory Management for Large Language Model Serving with PagedAttention," in *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.
- [3] P. Kidger and others, "jaxtyping: Type Annotations for Shape and Dtype of Arrays/Tensors."
- [4] Astral Software, "Ruff." [Online]. Available: <https://github.com/astral-sh/ruff>
- [5] Meta Platforms, Inc., "Pyrefly." [Online]. Available: <https://pyrefly.org/>
- [6] C. Curry and Beartype Contributors, "Beartype." [Online]. Available: <https://github.com/beartype/beartype>
- [7] D. MacIver, Z. Hatfield-Dodds, and M. Contributors, "Hypothesis: A new approach to property-based testing," *Journal of Open Source Software*, vol. 4, p. 1891, 2019, doi: 10.21105/joss.01891.
- [8] P. Schanely, "CrossHair: An Analysis Tool for Python Code, Driven by Formal Contracts." 2020.
- [9] Google DeepMind, "Gemma 4 E4B." Hugging Face, 2026.
- [10] G. Gerganov, "llama.cpp: Port of Facebook's LLaMA model in C/C++." GitHub, 2023.
- [11] J. Liu and contributors, "Instructor: Structured Outputs for LLMs." GitHub, 2023.
- [12] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
- [13] K. R. M. Leino, "Dafny: An Automatic Program Verifier for Functional Correctness," in *Proc. Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, 2010.
- [14] L. de Moura and S. Ullrich, "The Lean 4 Theorem Prover and Programming Language," in *Proc. Conf. on Automated Deduction (CADE)*, 2021.
- [15] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *Proc. POPL*, 1977.
- [16] K. Xu *et al.*, "Automatic Perturbation Analysis for Scalable Certified Robustness and Beyond," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

- [17] H. Zhang, T.-W. Weng, P.-Y. Chen, C.-J. Hsieh, and L. Daniel, "Efficient Neural Network Robustness Certification with General Activation Functions," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.
- [18] S. Wang *et al.*, "Beta-CROWN: Efficient Bound Propagation with Per-Neuron Split Constraints for Neural Network Robustness Verification," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.