A. Appendix

A.1. Extended Related Work

Code generation benchmarks. We present an extended comparison of various code benchmarks in Table 5.

A.2. Methodological Details

A.2.1. Airspeed Velocity Methodology

To benchmark a new function with Airspeed Velocity, a developer supplies a `setup(...)` routine and one or more time profiling functions (e.g. `time_foo(...)`, `time_bar(...)`) and memory profiling functions (e.g. `mem_foo(...)`, `mem_bar(...)`). `asv` then clones the repository, creates an isolated virtual environment, and records the performance characteristics for *all* commits. The tool ships with best-practice safeguards (CPU affinity, warm-ups, repeated trials, etc.) to control system variance.

Airspeed velocity offers many advantages towards our goal of making a benchmark for code optimization:

- **Low barrier to entry.** The minimalist interface means developers routinely add new benchmarks, expanding coverage over time. Asv ships with a robust regression-detection functionality which further motivates developers to ensure that the asv benchmarks maximally cover all performance critical parts of their software. We harvested XXX fine-grained timing and memory benchmarks across X repositories, averaging X benchmarks per project. Across XXX total commits, Superbench logs XXX timed runs.
- **Maturity and reliability.** First released on 1 May 2015, `asv` encapsulates nearly a decade of community experience in timing and memory profiling code on commodity hardware. Most common pitfalls have documented solutions and work-arounds, and platform-specific best practices (for Windows, macOS, and Linux) are well established, ensuring results are both accurate and precise.
- **CI integration.** `asv` co-exists naturally with other continuous-integration tools, so each commit carries both performance *and* correctness metadata.

A.2.2. Executing asv Benchmarks

Once we have collected a list of Python packages that ship with asv benchmarks, we now need the per-commit timing and memory profiles those benchmarks generate for each commit. In practice, we encounter two deployment patterns for benchmarking new commits: (1) a private benchmarking server that exposes results via a public web interface (Refer to Figure 3), (2) a benchmark directory committed with the source tree and runnable locally. The remainder of this section outlines two complementary workflows for collecting these measurements

Running all benchmark scripts locally. One could in principle run `asv` on the entire commit history of the main branch, collecting performance data with a one-line command. `asv` encapsulates nearly a decade of community experience in reliable timing and memory profiling on commodity hardware, which builds confidence in `asv`'s measurements. However, two practical concerns make this naive method impractical. First, benchmarks executed on our local machine may fail to expose regressions that are characteristics of certain operating systems and microarchitectures (e.g: performance characteristics of an x86 Linux node will be different than that of an ARM MacOS node). Second, running `asv` from scratch incurs considerable upfront cost, requiring the sequential construction of a new environment for each commit. With an average runtime of approximately 66 seconds per commit, a project such as `astropy` – which has 39514 commits – would require about 30 days of continuous execution, rendering the method infeasible.

To accelerate benchmarking and amortize this cost, we launch `asv` inside Docker containers. Each container is pinned to a dedicated CPU core and a fixed amount of RAM, and is given a subset of all the commits we wish to benchmark. Such sharded sandboxing gives us uniform runtime conditions across different test bench setups and also enables us to scale benchmarking horizontally as we can run as many simultaneous containers as CPU cores. This allows us to collect results much faster than a serial workflow and enables running `asv` on the entire commit history practical.

However, despite these changes, we faced two recurring challenges while collecting benchmarking results for all repositories. First, very old commits depend on packages that are no longer publicly accessible via PyPI. These commits cannot be replicated and are omitted from our benchmark. Second, many older packages are incompatible with newer versions of Python, and visa-versa, which makes benchmarking all commits with a homogeneous environment setup extremely challenging. We can mitigate the second issue by running multiple containers with different versions of Python.

Dashboard scraping. Because many projects host their `asv` results as a self-contained HTML site, in cases where local execution is not feasible, we can scrape precomputed results directly from the website. The `asv` dashboard is a self-contained static site and has a uniform file hierarchy across installations; making automated scraping straightforward. We maintain a curated list of publicly available dashboards in (Table 2). The time to scrape such webpage is almost negligible; however, in practice, we throttle our requests to respect host bandwidth which raises the collection time to around one hour for all the datasets. This workflow yields immediate historical performance traces and offers a sanity-check for the results of our locally running benchmark suite.

Benchmark	# Tasks	Data Source	Does data leakage help?	Live updates	Synthesis scope	High-fidelity Evaluation function
Ours	440^{++}	GitHub	Doesn't help; leaderboard is relative to humans.	Yes	Repository level	Yes
SWE-Bench	2292	GitHub	Helps; no relative-performance eval.	No	Repository level	No
LiveCodeBench	300^{++}	LeetCode, CodeForces / AtCoder	Yes; old tasks must be removed or scores inflate.	Yes	Function level	No
CruxEval	800^{++}	Custom	No; tasks are procedurally generated and adversarial.	No	Function level	No
ECCO	∼50 000	IBM CodeNet	Yes; many frontier models trained on CodeNet.	No	Function / File level	No

Table 5: An extended comparison of code-optimization benchmarks (From top: SWE-Bench (Jimenez et al., 2024), LiveCodeBench (Jain et al., 2024a), CruxEval (Gu et al., 2024), ECCO (Waghjale et al., 2024). "++" refers to continually updating benchmarks.

A.2.3. Step-Detection Considerations

In the current iteration of our dataset, the runtime measurement $t_{1:n}$, where n is the number of commits, is a scalar quantity which is averaged across multiple test bench runs. Our goal is to discover commits that noticeably improve performance, i.e., create an instantaneous yet persistent drop in runtime. Because CI noise, kernel scheduling, thermal throttling and other non-deterministic system behavior injects high-frequency variance, simply calculating the pairwise difference yields too many false positives.

We therefore cast the task as an offline step-detection problem. Conceptually, we can model our task as an offline step-detection problem where our data is assumed to be piecewise constant with added random noise – which reflects the common scenario that efficiency improvements are often facilitated by a subset of all the commits and each measurement carries some noise. Offline step detection is a well studied problem in signal processing (Truong et al., 2020) and many algorithms exist that balance the efficiency-optimality tradeoff. In this work, we chose to use the PELT algorithm with an RBF kernel loss (Killick et al., 2012) for robustness as implemented in the `ruptures` offline change point detection library (Truong et al., 2020). This algorithm is attractive as it makes no strong parametric assumptions about the underlying data, guarantees an optimal segmentation under an additive cost, and scales linearly in the size of the sequence n. We set the model regularization penalty to $3\log(n)$, which is the Bayesian information criterion; $k = 3$ is a hyper-parameter that empirically worked best but can be changed depending on the desired sensitivity.

A.3. Additional Results

In addition to the earlier reported results for the case study (§5.2, we also performed the same case-study evaluations for the Oracle versions of the tested baseline agents. These results in Table 6 reflect a similar result as in §5.1.1, where a restricted search space from the oracle file paths hampers full exploration for bottlenecks and forces the agent to unproductively brute force for what the human optimization was.

Though evolutionary algorithms OpenEvolve and AlphaEvolve were natively developed for use with Gemini, we also wanted to ensure that the LLM backbone was not a significant factor for the performance difference. To this, we splice in GPT-4o as the backbone and obtain results in Table 6 that highly resemble those obtained with Gemini.

Benchmark	GPT-4o Oracle	Sonnet 3.7 Oracle	OpenEvolve (GPT-4o)
objective benchmark	0.61	0.00	46.36
coordinates.FrameBenchmarks	0.16	0.17	9.88
coordinates.RepresentationBenchmarks	-0.01	0.24	28.12
coordinates.SkyCoordBenchmarks	-1.05	0.16	18.26
coordinates (core)	30.71	0.19	25.66
imports	0.00	0.00	0.25
Mean Improvement Percentage	3.46	0.17	**10.70**

Table 6: Runtime improvement percentages for oracle and OpenEvolve patches on Astropy Issue #13479. We compare GPT-4o Oracle, Sonnet 3.7 Oracle, and OpenEvolve (GPT-4o) across individual benchmarks and the Mean Improvement Percentage (MIP) in the last row.