FORMULACODE: Evaluating Agentic Superoptimization on Large Codebases

Atharva Sehgal [1 *] **James Hou** [2 *] **Swarat Chaudhuri** [1] **Jennifer J. Sun** [3] **Yisong Yue** [2]

Abstract

Rapid advances in LLM agents have demonstrated the ability to optimize code using continuous objective functions – a significant leap beyond traditional code generation. There is an urgent need for novel benchmarks that measure this ability to drive impact in real-world use cases. Existing code benchmarks, often relying on binary pass/fail outcomes, offer a constrained evaluation landscape compared to these emerging capabilities. To bridge this gap, we introduce FORMULACODE, a novel benchmark designed for evaluating agentic superoptimization on large codebases, with a focus on real-world performance optimization. Constructed from a dataset of 451 real-world performance bottlenecks automatically mined from Github, FORMULACODE enables comprehensive testing of an agent's ability to triage, diagnose, and resolve inefficiencies in realistic software environments. FORMULACODE proves to be a challenging benchmark for frontier LLMs and agentic frameworks, with unrestricted repository exploration emerging as a principal component for finding performance inefficiencies. By introducing FORMULACODE, our goal is to drive the development of next-generation optimization algorithms that meet the rigorous demands of real-world software projects.

1. Introduction

Recent results in using LLMs for code generation (Novikov et al., 2025; Romera-Paredes et al., 2024) highlight that pretrained LLMs, given sufficient compute and scale, can be leveraged to tackle complex problems in science and engineering. Code generated by such agents can even outperform that from human experts, demonstrating the potential to use agents for code superoptimization (Mankowitz et al., 2023a; Lin et al., 2025; Shypula et al., 2024), an idea dating back to the 1980s (Massalin, 1987) to iteratively improve an initial program using execution feedback. These *agentic superoptimization* approaches have the potential to transform the way domain experts and software engineers interact with code.

With this rising interest in agentic superoptimization, comes the need for rigorous benchmarking, in order to ensure scientific progress. However, existing coding benchmarks often focus binary pass/fail tests, rather than tasks such as improving compute or memory costs while passing all tests. Moreover, we are interested in moving towards evaluating system-level performance, rather than individual functions.

We identify several requirements for a useful superoptimization benchmark: (1) it must provide fine-grained evaluation metrics to capture nuanced performance changes; (2) agent performance should be assessed against a reliable human performance baseline to provide a meaningful standard; and (3) such a benchmark must also engage agents with large, real-world codebases, mirroring the challenges faced by human developers.

We introduce FORMULACODE[1] , a novel benchmark designed for advancing agentic superoptimization on large, evolving software ecosystems. FORMULACODE is constructed from real-world GitHub Issues and Pull Requests, drawn from six prominent open-source Python packages including Astropy, NumPy, and SciPy, all of which explicitly address performance optimization challenges. Each instance in FORMULACODE is paired with a human-written evaluation function, derived using the airspeed-velocity (asv) framework – a widely adopted tool for fine-grained performance tracking in the scientific Python community – and a ground-truth human-authored patch. This unique construction allows FORMULACODE to rigorously test an agent's ability to perform the complete optimization lifecycle—triage, diagnosis, and resolution—within authentic, complex software environments, and drive the creation of

[*]Equal contribution [1]The University of Texas at Austin [2]California Institute of Technology [3]Cornell University. Correspondence to: Atharva Sehgal <atharvas@utexas.edu>.

ICML 2025 Workshop on Programmatic Representations for Agent Learning, Vancouver, Canada.

[1]FORMULACODE draws inspiration from Formula 1, where constructors must optimize entire systems—not just individual components—to achieve peak performance on the track. Similarly, FORMULACODE challenges code agents to perform holistic, codebase-level optimizations, reflecting the complexity and interdependence found in real-world software.

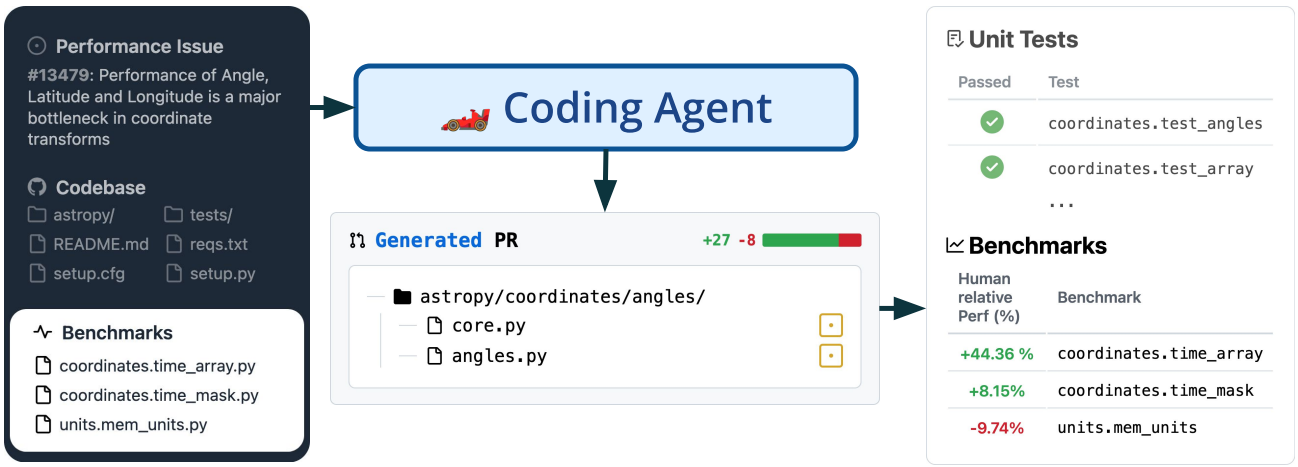

Figure 1: Test cases streamline performance evaluation but constrain coding agents (e.g., AlphaEvolve (Novikov et al., 2025)) to a pass/fail reward – a signal too sparse for fostering iterative optimizations. FORMULACODE introduces a live repository-level benchmark that *complements* existing work (In gray (Jimenez et al., 2024)) by challenging agents to optimize 451 real-world performance bottlenecks against human solutions drawn from community-maintained benchmarks (in light blue). These benchmarks provide evaluation functions that capture fine-grained performance insights, are less susceptible to data leakage, and expose a larger optimization surface to coding agents.

new algorithmic approaches for AI-driven optimization.

2. Related Work

Algorithms for Superoptimization. Code superoptimizers aim to solve programming tasks at a level beyond what is practically attainable by human experts by iteratively improving an initial program using execution feedback. Numerous systems now exceed human-level performance in specialized coding tasks. AlphaCode solves competition-level programming problems (Li et al., 2022). AlphaTensor and AlphaDev produce super-optimized matrix multiplication and sorting routines, respectively (Fawzi et al., 2022; Mankowitz et al., 2023b). These systems combine large, publicly sourced pretraining datasets with carefully chosen inductive biases to make optimization faster. Classical superoptimizers leverage stochastic search and constraint solving techniques to find efficient programs, primarily in low-level languages such as C++, C, and x86 (Schkufza et al., 2013; Sasnauskas et al., 2018). These systems often rewrite code while preserving high-level structure, optimizing memory use, cache performance, and data pipelining.

Recently, two algorithmic directions for iteratively optimizing programs with execution feedback have emerged. *Agentic Optimization* workflows attempt to help LLMs overcome their inability to predict the output of their generated code (Ni et al., 2024) by iteratively running the generated code, evaluating the output, and feeding the output back to the model (Shypula et al., 2024). SWEAgent offers one such implementation that benefits from iterative feedback (Yao, 2024; Yang et al., 2024). *Evolutionary Optimization* algorithms maintain a candidate pool of programs that are sorted in descending order of their performance. Old / inefficient implementations are discarded and new implementations are generated by mutating the best candidates. Funsearch and AlphaEvolve (Romera-Paredes et al., 2024; Novikov et al., 2025) demonstrate that an evolutionary coding agents equipped with pretrained LLMs can efficiently discover and refine novel, high-performance code-based heuristics across diverse scientific domains. Such evolutionary algorithms are extremely scalable but require high quality evaluation functions to penalize degenerate solutions. FORMULACODE is the first benchmark purpose built to assess the superoptimization ability of such agentic and evolutionary AI algorithms in real-world codebases and provides the fine-grained evaluation functions needed for iterative optimization.

Model	Task Horizon	Evaluation
Coding LLM	Single File	Test cases
Coding Agent	Codebase	Test cases
Optimization Agent	Codebase	Evaluation Functions

Table 1: FORMULACODE introduces a novel axis for benchmarking repository-level optimization agents (Novikov et al., 2025; Romera-Paredes et al., 2024) through the use of fine-grained evaluation functions.

Code Generation Benchmarks. Coding benchmarks can be differentiated by their synthesis scope. For a complete list of differences, consult Table 5.

Package	Stars	Citations	Coverage	Live ASV dashboard
Astropy	4 700	13 640	91.89%	astropy-benchmarks
pandas	45 500	12 474	85.25%	asv-runner
scikit-learn	62 100	112 578	98.96%	scikit-learn-benchmarks
SciPy	13 700	38 023	77.96%	scipy-bench
ArcticDB	1 900	–	–	man-group
NumPy	29 500	23 800	81.29%	numpy-bench

Table 2: An overview of the six code repositories included in FORMULACODE along with their public-facing Airspeed-Velocity dashboards and code coverage. We scrape asv-compatible active benchmarks using Google BigQuery (§3.2). The ASV dashboards are used to validate and interpolate any commits that weren't benchmarked locally.

Function and file level. Recent years have seen a proliferation of LLM-coding benchmarks. HumanEval (Chen et al., 2021), and MBPP (Austin et al., 2021) present hand-written programming problems in Python with corresponding unit tests. Many contributions extend these benchmarks to have more testing (Liu et al., 2023), broader scope (Yin et al., 2022; Yang et al., 2023), and more task diversity (Muennighoff et al., 2023; Lai et al., 2022; Zan et al., 2022). CruxEval (Gu et al., 2024) benchmarks the code execution and reasoning ability of LLMs more deeply. LiveCodeBench (Jain et al., 2024a) attempts to mitigate data-leakage by annotating problems with release dates. All these benchmarking efforts utilize unit testing suites to gauge program correctness. FORMULACODE presents a novel, yet orthogonal, axis for benchmarking code LLMs by using community-maintained evaluation functions that continually update with each commit to *supplement* the test cases provided by the above datasets.

Repository level. Function and file level benchmarks evaluate coding ability on self-contained coding tasks. However, real software issues typically span multiple modules and files. Repository level benchmarks (Jimenez et al., 2024; Tang et al., 2024; Jain et al., 2024b) aim to preserve the inherent challenges in real-world software engineering beyond text completion, such as finding relevant files, capturing relationships between modules, tracing information flow, etc. SWE-Bench (Jimenez et al., 2024) collects GitHub issues from popular repositories and evaluates coding agents' ability to resolve the issues. Follow-up efforts benchmark agents on repository-conditioned code synthesis (Tang et al., 2024) and scale-up benchmarking by admitting smaller codebases with LLM-generated unit tests (Jain et al., 2024b). These extensions still is susceptible to data leakage, is hard to directly optimize, and hard to generate. Instead, FORMULACODE *supplements* these benchmarks by assessing agents on community-maintained evaluation functions that present a smoother optimization landscape and, by design, scale substantially better than unit tests in terms of code coverage per testing instance.

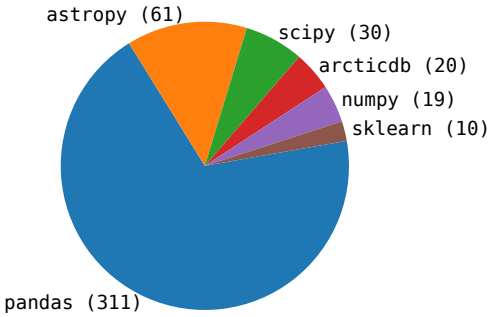

Figure 2: Distribution of FORMULACODE tasks across five open source GitHub repositories. These repositories have a combined 157,000+ GitHub stars and 200,000+ academic citations and each repository uses *Airspeed Velocity* for regression testing. We collect 451 filtered tasks for our preliminary dataset consisting of 500,000+ measurements.

Efficiency Benchmarks. Many benchmarks from the software engineering community study efficient code synthesis on function and file-level efficiency tasks. COFFE (Peng et al., 2025) samples tasks from HumanEval, MBPP, CodeContests, and APPS (Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021) and auto-generates stress tests while ECCO (Waghjale et al., 2024) curates a function and file-level efficient synthesis dataset from IBM CodeNet (Puri et al., 2021) with data-mined test cases. FORMULACODE supersedes these efforts by (1) using community-maintained benchmarks specifically designed to profile performance inefficiencies instead of using hand-curated stress tests and (2) benchmarking on repository-level codebases, which better capture the natural challenges with real-world code optimization.

3. Methodology

FORMULACODE introduces a data-mining pipeline to *automatically generate, maintain, and continually update* a benchmark for code generation given fine-grained evaluation functions. In this section, we first describe the library purpose-build for benchmarking on commodity hardware (§3.1). Then, we describe the scalable process of collecting such benchmarks automatically (§3.2) and the process of executing such benchmarks to obtain performance measures (§3.3). Finally, we showcase two strategies to find performance improving code dits (§3.4). Appendix §A.2 outlines lower-level details.

3.1. Airspeed Velocity.

Airspeed Velocity (`asv`) is a command-line tool that automatically times, memory-profiles, and regression-tests every commit in a Python package against a benchmark

suite defined by domain experts, surfacing the results in a zero-dependency HTML dashboard.[2]Because the tool integrates easily with existing test suites, it has become the backbone of performance testing for many open-source projects, including many repositories in the PyData ecosystem (NumPy, SciPy, scikit-learn, Astropy, Napari, Pandas, and others). Although primarily used with Python, `asv` is multi-lingual via plugins for C++, Julia, Go, etc. An overview of `asv`'s dashboard is presented in Figure 3. More information about `asv` is presented in Appendix § A.2.1.

3.2. Benchmark Discovery via Google BigQuery

Because `asv` dashboards are self-hosted, there is no central registry of regression-testing benchmarks. Moreover, many projects prefer contributors run `asv` locally, so benchmarking scripts are scattered across countless repositories. Naively collecting such benchmarks presents a significant challenge as it requires potentially scraping all of GitHub for repositories with asv installed.

Instead, we query the GitHub Public Dataset on Google BigQuery (GitHub & Google Cloud Platform, 2025), which snapshots ∼2.8 M open-source repositories and more than 2 B code files. To find GitHub repositories that benchmark with asv, we notice that every `asv` benchmark must include an `asv.conf.json` file at the root of its benchmark suite. This anchor file serves as the search key for a commonSQL program to look for asv-compatible code repositories. We include additional filtering steps to ensure forked, archived, and reuploaded repositories are excluded from the search. Overall, this naive query scanning all content would require processing 2.69 TB of data. Due to budget constraints of around 0.25 TB, we restricted the search to configuration files located at the repository root. While this excludes many repositories, we were able to retrieve roughly 400 actively maintained `asv` benchmarks. FORMULACODE focuses on the six with the most GitHub stars: *Astropy*, *NumPy*, *Pandas*, *ArcticDB*, *SciPy*, and *scikit-learn*.

3.3. Benchmark Execution Overview

Running every benchmark on every commit can be accomplished either by *locally* or by *scraping* precomputed dashboards. The two complementary workflows are sketched below; full implementation details, hardware specifications, and edge-case handling appear in Appendix A.2.2.

Local execution. We orchestrate `asv` inside Docker containers pinned to dedicated CPU cores, sharding the commit history across containers to achieve near-linear speed-ups.

Dashboard scraping. For projects that already host public `asv` sites, we develop an ethical HTML crawler that scrapes the precomputed results directly from these websites. An `asv` dashboard is a self-contained static site and has a uniform file hierarchy across repositories; making automated scraping straightforward.

Intuitively, local execution is always preferable to dashboard scraping. However, many historical commits can no longer be replicated due to missing dependencies that are no longer publicly accessible via PyPI. In such instances, the scraping workflow yields immediate historical traces and helps validate the results of our locally running benchmark suite.

3.4. Finding performance improving code edits

Each benchmark produces two chronologically ordered vectors: a sequence of commit hashes $h_{1:n}$ and a sequence of runtime measurements $t_{1:n}$, where n is the number of commits. We frame the search for performance-improving commits as an *offline step-detection* problem and employ PELT with an RBF kernel loss (Killick et al., 2012; Truong et al., 2020) to generate a change-point set $\{c_1, \ldots, c_k\}$ More discussion about the choice of change-point detection algorithms is presented in Appendix A.2.3. We subsequently compute the immediate percentage change in runtime at each change-point boundary

$$\Delta_{\%}(c_i) = \frac{t_{c_i} - t_{c_i-1}}{t_{c_i-1}},$$

and retain only indices with $\Delta_{\%}(c_i) < 0$. For every retained change point, we record:

- the commit *before* the improvement, h_{c_i-1};
- the improving commit itself, h_{c_i}, together with all linked hyperlinks (pull requests, issues, comments) up to two levels deep;
- the precise patch obtained from `git diff` (h_{c_i-1}, h_{c_i});
- other relevant performance metadata such as the benchmark name, instantaneous improvement $\Delta_{\%}(c_i)$, and the aggregated `asv` reported runtimes t_{c_i-1} and t_{c_i}.

4. Case Study: Gold standard optimization of coordinate transformations in `astropy`

This case study traces how human contributors to a large open-source scientific library identified, isolated, and resolved a major performance bottleneck in Astropy. We highlight the chain of reasoning and patch iteration that ultimately yielded a significant enough speedup to warrant a major version update – and contrast this with how a state-of-the-art code-optimization agent performs on the same task.

Background. Astropy is the canonical Python package for astronomy, supporting a range of mission-critical tasks from calibrating exposures on the *James Webb Space Tele-*

[2] `https://asv.readthedocs.io`

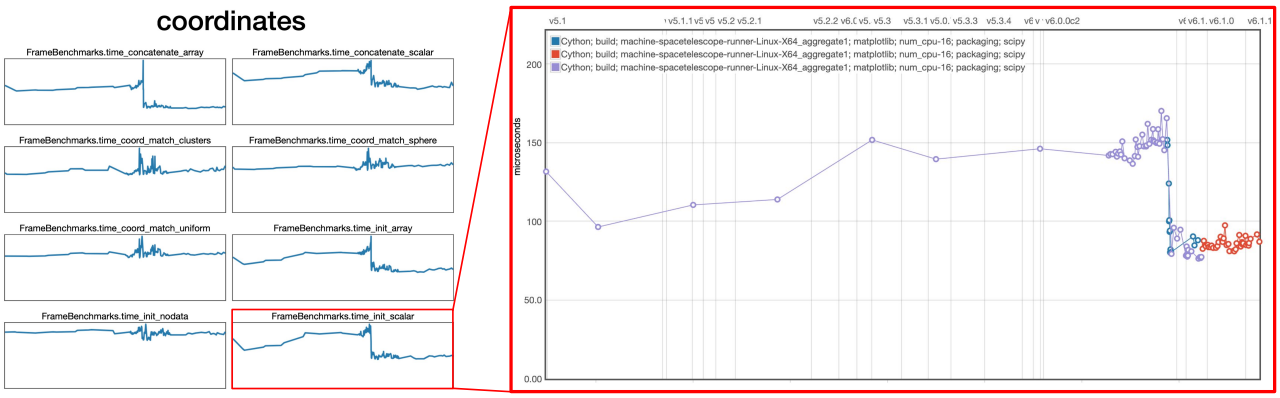

Figure 3: **Airspeed Velocity dashboard for astropy.** Airspeed Velocity (ASV) is the most widely used framework for continuous benchmarking and performance-regression detection in Python packages. **Left**: overview panel summarising wall-time trends for 8 (of 390) community-maintained benchmarks across the entire *astropy* commit history. **Right**: Detailed view for the `time_init_scalar` benchmark. Points represent individual commits; the x-axis shows commit dates (release tags highlighted) and the y-axis the benchmark runtime in microseconds. Color is used to indicate different test bench machines. Hovering over a point reveals commit metadata. The pronounced speed-up in the `time_init_scalar` benchmark around release `v6.0` – mirrored in other benchmarks – results from the fix to Issue #13479 discussed in §4.

scope (Bushouse et al., 2025) to real-time localization of gravitational wave events with the *Laser Interferometer Gravitatinal Wave Observatory* (Singer & Price, 2016). With over 13000 citations and widespread adoption across major astronomical surveys, its correctness and efficiency are critical to scientific workflows. The project employs multiple tools to maintain high standards, including continuous integration (via CircleCI), test coverage tracking (via Codecov), and fine-grained benchmarking (via Airspeed Velocity).

Despite this infrastructure, performance regressions can still arise – especially in numerical subroutines used extensively throughout the codebase.

Manual Discovery and Fix. In Issue #13479, a core contributor noticed a major inefficiency in astropy's coordinates submodule. Using `%timeit`, they profiled the instantiation of three central abstractions in the submodule (`Longitude`, `Latitude`, `Angle`) and discovered that these classes were markedly slower – sometimes nearly 14× slower – than other classes with the same numerical backbone. Manual inspection of the class hierarchy isolated two major bottlenecks. (1) A subroutine redundantly executed expensive NumPy reduction operations on already-validated inputs and (2) All three classes invoked `_wrap_at` (which confines angles to $[0°, 360°)$) even when wrapping was unnecessary.

Further analysis yielded deeper insights. First, they confirmed that operations such as NumPy's `any` and `isfinite` were particularly costly, even when applied to well-formed inputs. Next, they showed that even naive changes (e.g., skipping validation) yielded 5–10× speedups, but were not acceptable due to correctness concerns. Eventually, the issue was resolved by short-circuiting domain checks when inputs were guaranteed to be valid, replacing costly NumPy comparisons with faster identity checks (e.g., using `unit is u.hour` rather than `unit == u.hour`), and unconditionally applying angle wrapping when benchmarks showed it was faster than conditional checks.

These changes were incrementally introduced across multiple commits, each benchmarked and validated against the full test suite. The outcome was a 2–5× speedup in coordinate class instantiation and an approximate 15% improvement in downstream transformation performance. Crucially, these gains required no external static analyzers or compiler-level optimizations – only domain expertise and careful reasoning about low-level numerical behavior.

Takeaways. This example underscores the difficulty of finding and resolving performance bugs in scientific software: they often reside deep inside well-tested abstractions, far from user code or top-level benchmarks. Addressing them necessitates (1) localized knowledge about domain-specific requirements (astronomy numerical constraints), awareness of language-specific performance costs (NumPy reductions' overhead, Python's == vs. `is`), and determining when speed takes priority over generality in performance-critical paths.

Section 5.2 takes a deeper look at how current state-of-the-art optimization agents perform on this task.

5. Experiments

Below, we will detail how code/optimizing agents are evaluated upon FORMULACODE, including the information about the human optimization accessible to the agent and the metrics we measure the agent performance by. We also perform a series of experiments on various code agent works to demonstrate the importance of a codebase-level optimization benchmarks like FORMULACODE.

Evaluation Metrics. Each FORMULACODE instance is centered around an `asv` benchmark test that exhibited a significant runtime improvement from one commit of a codebase – the base commit – to a subsequent improvement commit. Given these two commits and the associated benchmark, we compute the human optimization percentage as the relative runtime reduction between the base and improvement commits:

$$\Delta\% = \frac{t_{\text{improved}} - t_{\text{base}}}{t_{\text{base}}}$$

This value serves as a reference target for what human developers achieved through manual optimization.

To evaluate code agents, we provide them with the codebase versioned at the base commit, the relevant `asv` benchmark test code (as defined in the project's own benchmarking collection), and the human-obtained reduction percentage. This setup gives the agent a well-scoped optimization task with a clear performance objective. Though there are no strict guardrails against modifying the benchmark suite, the separation between the benchmark repository and the actual benchmarked code has prevented such cheating and to our best knowledge, we have not seen behavior of agents changing test cases.

Once the code agent performs the super-optimizing code edits (model-specific details expanded upon below), we apply `asv`'s tooling to evaluate the modified codebase on the relevant benchmark test again to measure its runtime improvement percentage upon the base commit, which we can use to compare against the human baseline.

Importantly, this evaluation pipeline requires minimal setup overhead. Because `asv` is already integrated into many scientific and performance-sensitive codebases, we are able to reuse the codebase's native benchmarking configuration without modification. This makes FORMULACODE a frictionless, realistic, and reproducible framework for evaluating code optimization performance at the commit level.

Another metric we consider is **Mean-Improvement-Percentage (MIP)**. As we are evaluating codebase-level optimization, we must consider the impact of an optimization not only at modification but also its impact at a codebase level on all the code dependent on it. To measure this, we leverage the hierarchy present within `asv` benchmarks, where each benchmark test belongs to a benchmark suite (`coordinates.FrameBenchmarks.time_concatenate_array` → `coordinates`) that encompasses tests within the same submodule in the codebase. We can evaluate the modified codebase (optimized for a specific benchmark test) upon all other tests within the same suite and measure their individual improvement percentages and take the mean to obtain the **Mean-Improvement-Percentage** metric, essentially capturing the ripple effect the change has on its dependencies. Due to time limitations and the metric's requirement to benchmark dozens of tests beyond the relevant test, we only deploy and experiment with this metric in our case study as reference for future works.

Evaluated Baselines. In our experiments, we primarily consider two superoptimization approaches: 1. software agents tasked with optimization issue to tackle and 2. evolutionary algorithms to iteratively improve upon a given initial program.

For the first approach, we employ SWE-Agent (Yang et al., 2024), an agentic method facilitating tool-use for software engineering, as the guiding framework for how the LLM interacts with the codebase and evaluate widely applied models such as GPT-4o (OpenAI et al., 2024) and Claude 3.7 (Anthropic, 2024) Sonnet as baselines.

With the recent release of AlphaEvolve (Novikov et al., 2025) and rising interest in evolutionary methods powered by LLMs, we also consider OpenEvolve (Sharma, 2025), a popular open source implementation of AlphaEvolve. Given that AlphaEvolve has already demonstrated impressive ability to optimize complex algorithms, this baseline serves as a representative gauge on the effectiveness of evolutionary algorithms in the context of code optimization.

5.1. Pilot Evaluations

To highlight the need for a fine-grained evaluation metric – optimization runtime improvement – beyond the test-case correctness emphasized in past benchmarks, we evaluate established tools on a curated dataset from FORMULACODE. This dataset consists of commit pairs extracted from the Astropy (Astropy Collaboration et al., 2022) library that demonstrate meaningful performance improvements.

Before evaluating, we identify key challenges that program synthesizers must address to succeed in a codebase-level optimization benchmark like FORMULACODE:

- The code agent should be capable of performing its own profiling to evaluate and iteratively refine its program variants, implying the need for tool integration and active feedback.

Benchmark Suite	# Instances	GPT-4o		Sonnet 3.7		GPT-4o Oracle		Sonnet 3.7 Oracle	
		Δ%	#Valid	Δ%	#Valid	Δ%	#Valid	Δ%	#Valid
coordinates	15	-32.11	8	**8.91**	12	-36.68	11	5.18	12
imports	10	-9.26	5	13.87	5	2.18	4	**14.94**	3
io_ascii	7	0.13	2	-23.96	3	-4.37	4	**15.01**	4
io_fits	3	-2.37	1	-56.86	1	**21.18**	1	–	0
modeling	7	-3.08	3	**18.15**	6	-20.58	5	0.68	4
stats	2	-10.20	2	-2.21	2	-1.29	1	**-1.09**	2
table	7	3.53	3	**23.58**	5	-5.31	3	-1.98	6
units	10	-11.87	6	**13.61**	8	-10.54	5	-4.46	8
Overall	61	-13.19	30	**9.02**	**42**	-16.58	34	3.08	39

Table 3: Agent performance on 61 optimization instances from the FORMULACODE benchmark, grouped by benchmark suite. The "# Instances" column indicates the number of benchmark tests in each suite used as optimization targets. Δ% reports the average performance gap between the human-optimized and model-optimized runtimes, where negative values indicate underperformance relative to human edits. "#Valid" reports the number of instances in which a model produced a valid optimization patch. Oracle settings provide models with access to files modified in the human patch as additional context. Sonnet 3.7 demonstrates strong promise for agentic optimization §5.1.1, while additional oracle information appears to limit optimization search space and correspondingly performance §5.1.2.

- It must trace and reason about the call graph originating from the benchmark test to locate effective optimization points and anticipate downstream effects of local edits.
- It should manage context across multiple files to identify non-local performance bottlenecks and optimize in a globally coherent fashion, rather than pursuing greedy, locally optimal edits that harm overall performance.

To investigate these properties, we conduct the following targeted experiments:

- **Baseline Comparison.** We benchmark the performance of two leading models, GPT-4o and Claude Sonnet, against human improvements, in effort to establish a performance baseline and capture the gap between current LLM capabilities and expert human optimization.
- **Retrieval Ablation.** We test models with and without oracle guidance (identifying edited files) to evaluate the impact of targeted retrieval on optimization quality.

5.1.1. BASELINE COMPARISON

Experiment Setup In this experiment, we deploy popular models GPT-4o and Claude Sonnet 3.7 onto FORMULACODE via SWE-Agent. To interface the SWE-Agent with FORMULACODE problem instances, we make light modifications to structure our optimization problem as a pull-request (PR) statement, mirroring human-opened PRs as shown in Section 4 to address a potential optimization. We design the PR to contain the benchmark test name and code to optimize for and the reference percentage improvement achieved by the human improvement commit for the agent to reach for. Then, we perform batch deployment of SWE-Agent, GPT-4o or Sonnet 3.7 as the LLM, upon these constructed PRs (with 100 iterations) to obtain predicted patches to the codebase at their respective base commits. These predicted patches are then applied to the codebases and we again measure their runtime improvement on their corresponding benchmark tests via `asv`. In some instances, the predicted patch fails to apply or contains errors. We record such cases accordingly.

Agentic Optimization Can Rival Human Performance on Local Tasks We report the results of our baseline runs with GPT-4o and Claude Sonnet 3.7 in Table 3. Over all passing tests, GPT-4o predicted optimizations on average reduced **13.19%** less runtime than the ground truth human optimization on the same test, while Claude Sonnet 3.7 achieved **9.02%** more runtime reduction than the ground truth human optimization on passing instances. Moreover, Claude Sonnet 3.7 passed on 42 of the 61 identified instances, while GPT-4o passed on 30.

The promising results of Claude Sonnet 3.7 in an agentic framework give reason for excitement for the potential of LLM-powered agentic superoptimization. It is important to note that SWE-Agent was developed around Claude Sonnet 3.7 and that Sonnet itself is more tailored for code generation than GPT-4o. Moreover, as we'll see in the case study, though Claude may perform well in optimizing for a single benchmark test in this experiment, when we consider a more comprehensive metric like MIP, the challenge of codebase-level superoptimization becomes much more challenging.

Benchmark	Human	GPT-4o	Sonnet 3.7	OpenEvolve	Composition
objective benchmark	46.91	59.39	-0.61	44.36	**70.91**
coordinates.FrameBenchmarks	16.60	18.61	9.80	8.15	**23.75**
coordinates.RepresentationBenchmarks	**17.71**	17.63	23.96	3.88	9.04
coordinates.SkyCoordBenchmarks	**21.28**	13.37	3.40	13.95	16.05
coordinates (core)	2.92	**22.91**	-5.75	9.74	-4.51
imports	-0.25	0.00	**0.25**	-0.25	**0.25**
Mean Improvement Percentage	**16.77**	15.88	5.24	10.72	14.71

Table 4: Runtime improvement percentages for the case study instance from Astropy Issue #13479, including the objective benchmark (`coordinates.FrameBenchmarks.time_concatenate_array`) and the broader `coordinates` benchmark suite. Each column represents the performance of a different optimization method: the original human patch, GPT-4o, Claude Sonnet 3.7, OpenEvolve, and a composed patch combining GPT-4o and the human edit (§5.2.3). The final row reports the Mean Improvement Percentage (MIP) across the suite. This metric informs our analysis in §5.2.2 that corroborate the need for a codebase level optimization benchmark like FORMULACODE.

5.1.2. RETRIEVAL ABLATION

Experiment Setup An important component of code agents operating on codebase-scale problems is the ability to retrieve the relevant pieces – in our case, bottlenecks to runtime – of the codebase. To study the ability of LLM superoptimization given perfect retrieval, we design the following retrieval oracle experiment. For each base-improvement commit pair, we record the files modified in the human optimization and provide their paths to the agents as additional information for the agent to replicate.

We present the results in Table 3 and report that the GPT-4o Oracle evaluation successfully passed in $34/61$ tests and on average achieved **16.58%** less runtime improvement than the reference human commit, while Claude Oracle successfully passed in $39/61$ tests and on average achieved **3.08%** better performance than the human optimization.

Oracle Retrieval Can Mislead LLMs It is interesting to note that despite the increased access to ground truth information, the oracle agents were less effective at optimization. We hypothesize that by conditioning the agents to the ground truth files, we constrain the agent's search space to strictly within those files, missing out other potential bottlenecks that the agent would have identified. In this case, if the agent is unable to replicate the optimization of the ground truth, improvement is highly unlikely, leading to the reduced optimization for the oracle agents. This also suggests that tasked with tackling FORMULACODE, agents are not just memorizing solution patches to the same test but instead finding orthogonal and even complementary optimizations to the original improvement based on our fine-grained evaluation metric. We explore this idea further in our case study, analyzing the trace of agent actions on a specific instance and the result of merging human and agent patches.

5.2. Case Study

In this case study, we take a deeper dive into the human optimization presented in Section 4 and compare it with the adjacent solutions predicted by various code optimization agents. Specifically, in Astropy Issue $\#13479$, leading from base commit 96cc7fbe to improvement commit 1ff8068, a human optimization was discovered to avoid redundant angle wrapping, which led to a $\sim 47\%$ reduction in runtime on the `coordinates.FrameBenchmarks.time concatenate_array` benchmark test. Moreover, during our data collection, our screening process identified multiple other benchmark tests that also benefitted from the improvement commit, reflecting that speedups often have global effects in the codebase (Angle is used in many places). Thus, this inspires us to perform a case study on this specific instance to evaluate whether the optimizations introduced by code agents are equally as impactful beyond the objective benchmark (`time_ concatenate_array`) when given the same problem.

5.2.1. CASE STUDY SETUP

For our baseline agents – SWE-Agent with GPT-4o and Claude Sonnet 3.7 – we deploy them on the problem instance in the manner described above to obtain prediction patches that we apply to the codebase at the base commit. In addition to these two baseline agents, we also experiment with the evolutionary program synthesizer OpenEvolve. As the state-of-the-art AlphaEvolve program is yet to be released, we opt for an open-source implementation of AlphaEvolve in OpenEvolve. At the time of our experiments, OpenEvolve only supported single-file evolution. To work around this, we aggregated all relevant code between the benchmarked function and the optimized `_wrap_at` into one file, made executable via dynamic patching. OpenEvolve then evolved intermediate function

bodies while preserving their signatures, allowing us to simulate the optimizations by dynamically patching the original codebase at evaluation. As for the evolutionary objective, we define it as the Δ improvement percentage achieved by the current iteration with respect to the human optimization improvement percentage. For consistency to the other baseline agents, we evolved the codebase for 100 iterations with Gemini 2.0 Flash. Then, at evaluation time, we translate the evolved modifications to the codebase. Since the file-level limitation of OpenEvolve demanded a lot of engineering, we only evaluate it on the case study instance.

At evaluation, for each of the aforementioned code agents, instead of evaluating their predicted patch on solely the objective benchmark test `time_concatenate_array`, we also evaluate the modified codebase on all other tests within the same benchmark suite `coordinates` and take the mean of their respective improvement percentages to compute the MIP metric.

5.2.2. Local vs. Global Effects of Optimization

Table 4 summarizes the runtime improvements achieved by various code agents on both the objective benchmark – `time_concatenate_array` – and the broader `coordinates` suite via the Mean Improvement Percentage (MIP).

Agentic Workflows At first glance, GPT-4o demonstrates strong performance on the specific benchmark, outperforming the human baseline with a **59.39%** reduction versus **46.91%**. However, its overall MIP **15.88%** actually underperforms in comparison to the human MIP, revealing that this improvement does not generalize well across the suite. In other words, GPT-4o's patch is more overfit to the benchmark test rather than optimizing code structure globally.

Surprisingly, Claude achieves its largest gain on `coordinates.RepresentationBenchmarks` with a **23.96%** improvement, despite not optimizing for this benchmark, suggesting that the performance change may not be the result of deliberate optimization, but rather a byproduct of structurally untargeted code edits that may be unreliable. Yet, overall, Claude is still significantly less effective in comparison to all other baselines, demonstrating that an unguided approach is not sustainable.

Evolutionary Strategies Lag Behind OpenEvolve also suffers from similar generalization issues as GPT-4o. Although it reaches **44.36%** – comparable to the human baseline – its MIP degrades significantly in comparison to the human baseline, reflecting a similar local-over-global trend. Its implementation, which limits its visibility to the specification file we compiled, restricts its ability to take codebase-level context and produce coherent codebase-wide improvements, which we believe leads to both the underperformance in the *objective benchmark* and the MIP. However, with new works like AlphaEvolve coming out that have agentic abilities like file exploration, evolutionary algorithms provide a consistent and iterative optimization mechanism.

5.2.3. Complementary Patching

A particularly intriguing finding in our case study is the synergy between human and model-generated optimizations. In the benchmark instance, `time_concatenate_array`, we observed that the patch predicted by GPT-4o targeted a different layer of the performance stack. While the human optimization addressed the redundant operations at the angle level (angles are elements of the array), the GPT-4o agent identified inefficiencies in the array construction logic surrounding those elements. Thus, as these patches operated in different regimes of the codebase, we became interested in whether they were complementary when combined as a single patch.

Composition Results As reported in Table 4, the merged patch yields a runtime reduction of **70.91%** on the benchmark test, compared to **46.91%** for the human edit and **59.39%** for GPT-4o alone. This result underscores the potential for synergistic collaboration between human developers and code agents. This experiment suggests they can serve as complementary tools – surfacing orthogonal optimization opportunities that may not be immediately apparent even to experienced maintainers.

However, the success of this merged patch must still be taken with caution. Though our principal benchmark registered a performance surge, the MIP metric actually declined. We hypothesize that by stacking array-level optimizations on element-level optimizations, the changes place a stronger prior on array operations, which are not as effective when applied to scalar data (the underperforming benchmarks were largely scalar benchmarks).

6. Conclusion

We have presented FormulaCode, which is, to our knowledge, the most comprehensive coding benchmark for agentic superoptimization. In this benchmark, coding agents must not only write code that passes standard correctness tests, but also improve runtime, often of large codebases (i.e., with many function calls). FormulaCode enables the rigorous development and evaluation of coding agents for superoptimization, which is becoming an increasingly important direction in the LLM coding research area. Our evaluations show that FormulaCode is a challenging benchmark for frontier LLMs and agentic frameworks, leaving open significant room for future agent development.

References

Anthropic. The Claude 3 Model Family: Opus, Sonnet, Haiku. https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf, March 2024. Model card v1.0. Accessed 31 May 2025.

Astropy Collaboration, Price-Whelan, A. M., Lim, P. L., Earl, N., Starkman, N., Bradley, L., Shupe, D. L., Patil, A. A., Corrales, L., Brasseur, C. E., Nöthe, M., Donath, A., Tollerud, E., Morris, B. M., Ginsburg, A., Vaher, E., Weaver, B. A., Tocknell, J., Jamieson, W., van Kerkwijk, M. H., Robitaille, T. P., Merry, B., Bachetti, M., Günther, H. M., Aldcroft, T. L., Alvarado-Montes, J. A., Archibald, A. M., Bódi, A., Bapat, S., Barentsen, G., Bazán, J., Biswas, M., Boquien, M., Burke, D. J., Cara, D., Cara, M., Conroy, K. E., Conseil, S., Craig, M. W., Cross, R. M., Cruz, K. L., D'Eugenio, F., Dencheva, N., Devillepoix, H. A. R., Dietrich, J. P., Eigenbrot, A. D., Erben, T., Ferreira, L., Foreman-Mackey, D., Fox, R., Freij, N., Garg, S., Geda, R., Glattly, L., Gondhalekar, Y., Gordon, K. D., Grant, D., Greenfield, P., Groener, A. M., Guest, S., Gurovich, S., Handberg, R., Hart, A., Hatfield-Dodds, Z., Homeier, D., Hosseinzadeh, G., Jenness, T., Jones, C. K., Joseph, P., Kalmbach, J. B., Karamehmetoglu, E., Kałuszyński, M., Kelley, M. S. P., Kern, N., Kerzendorf, W. E., Koch, E. W., Kulumani, S., Lee, A., Ly, C., Ma, Z., MacBride, C., Maljaars, J. M., Muna, D., Murphy, N. A., Norman, H., O'Steen, R., Oman, K. A., Pacifici, C., Pascual, S., Pascual-Granado, J., Patil, R. R., Perren, G. I., Pickering, T. E., Rastogi, T., Roulston, B. R., Ryan, D. F., Rykoff, E. S., Sabater, J., Sakurikar, P., Salgado, J., Sanghi, A., Saunders, N., Savchenko, V., Schwardt, L., Seifert-Eckert, M., Shih, A. Y., Jain, A. S., Shukla, G., Sick, J., Simpson, C., Singanamalla, S., Singer, L. P., Singhal, J., Sinha, M., Sipőcz, B. M., Spitler, L. R., Stansby, D., Streicher, O., Šumak, J., Swinbank, J. D., Taranu, D. S., Tewary, N., Tremblay, G. R., de Val-Borro, M., Van Kooten, S. J., Vasović, Z., Verma, S., de Miranda Cardoso, J. V., Williams, P. K. G., Wilson, T. J., Winkel, B., Wood-Vasey, W. M., Xue, R., Yoachim, P., Zhang, C., Zonca, A., and Astropy Project Contributors. The Astropy Project: Sustaining and Growing a Community-oriented Open-source Project and the Latest Major Release (v5.0) of the Core Package. *The Astrophysical Journal*, 935(2):167, August 2022. doi: 10.3847/1538-4357/ac7c74.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., and Sutton, C. Program synthesis with large language models, 2021.

Bushouse, H., Eisenhamer, J., Dencheva, N., Davies, J., Greenfield, P., Morrison, J., Hodge, P., Simon, B., Grumm, D., Droettboom, M., Slavich, E., Sosey, M., Pauly, T., Miller, T., Jedrzejewski, R., Hack, W., Davis, D., Crawford, S., Law, D., Gordon, K., Regan, M., Cara, M., MacDonald, K., Bradley, L., Shanahan, C., Jamieson, W., Teodoro, M., Williams, T., Pena-Guerrero, M., Graham, B., Molter, E., Brandt, T., Hayes, C., Cooper, R., Clarke, M., and Filippazzo, J. JWST Calibration Pipeline, April 2025. URL https://zenodo.org/records/15178003.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., and et. al, J. K. Evaluating large language models trained on code, 2021.

Fawzi, A., Balog, M., Huang, A., Hubert, T., Romera-Paredes, B., Barekatain, M., Novikov, A., R. Ruiz, F. J., Schrittwieser, J., Swirszcz, G., et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.

GitHub and Google Cloud Platform. bigquery-public-data.github_repos – github public repository dataset. https://console.cloud.google.com/marketplace/details/github/github-repos, 2025. Queried via Google BigQuery on 30 May 2025.

Gu, A., Rozière, B., Leather, H., Solar-Lezama, A., Synnaeve, G., and Wang, S. I. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*, 2024.

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., and Steinhardt, J. Measuring coding challenge competence with apps, 2021.

Jain, N., Han, K., Gu, A., Li, W.-D., Yan, F., Zhang, T., Wang, S., Solar-Lezama, A., Sen, K., and Stoica, I. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024a.

Jain, N., Shetty, M., Zhang, T., Han, K., Sen, K., and Stoica, I. R2E: Turning any github repository into a programming agent environment. In Salakhutdinov, R., Kolter, Z., Heller, K., Weller, A., Oliver, N., Scarlett, J., and Berkenkamp, F. (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 21196–21224. PMLR, 21–27 Jul 2024b. URL https://proceedings.mlr.press/v235/jain24c.html.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. Swe-bench: Can language models resolve real-world github issues?, 2024. URL https://arxiv.org/abs/2310.06770.

Killick, R., Fearnhead, P., and Eckley, I. A. Optimal detection of changepoints with a linear computational cost. *Journal of the American Statistical Association*, 107 (500):1590–1598, October 2012. ISSN 1537-274X. doi: 10.1080/01621459.2012.737745. URL http://dx.doi.org/10.1080/01621459.2012.737745.

Lai, Y., Li, C., Wang, Y., Zhang, T., Zhong, R., Zettlemoyer, L., tau Yih, S. W., Fried, D., Wang, S., and Yu, T. Ds-1000: A natural and reliable benchmark for data science code generation, 2022.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Lin, H., Maas, M., Roquemore, M., Hasanzadeh, A., Lewis, F., Simonson, Y., Yang, T.-W., Yazdanbakhsh, A., Altinbüken, D., Papa, F., Edmonds, M. N., Patil, A., Schwarz, D., Chandra, S., Kennelly, C., Hashemi, M., and Ranganathan, P. Eco: An llm-driven efficient code optimizer for warehouse scale computers, 2025. URL https://arxiv.org/abs/2503.15669.

Liu, J., Xia, C. S., Wang, Y., and Zhang, L. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*, 2023.

Mankowitz, D. J., Michi, A., Zhernov, A., Gelmi, M., Selvi, M., Paduraru, C., Leurent, E., Iqbal, S., Lespiau, J., Ahern, A., Köppe, T., Millikin, K., Gaffney, S., Elster, S., Broshear, J., Gamble, C., Milan, K., Tung, R., Hwang, M., Cemgil, T., Barekatain, M., Li, Y., Mandhane, A., Hubert, T., Schrittwieser, J., Hassabis, D., Kohli, P., Riedmiller, M., Vinyals, O., and Silver, D. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964):257–263, June 2023a. ISSN 1476-4687. doi: 10.1038/s41586-023-06004-9. URL https://doi.org/10.1038/s41586-023-06004-9.

Mankowitz, D. J., Michi, A., Zhernov, A., Gelmi, M., Selvi, M., Paduraru, C., Leurent, E., Iqbal, S., Lespiau, J.-B., Ahern, A., et al. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964): 257–263, 2023b.

Massalin, H. Superoptimizer: a look at the smallest program. *SIGARCH Comput. Archit. News*, 15(5): 122–126, October 1987. ISSN 0163-5964. doi: 10.1145/36177.36194. URL https://doi.org/10.1145/36177.36194.

Muennighoff, N., Liu, Q., Zebaze, A., Zheng, Q., Hui, B., Zhuo, T. Y., Singh, S., Tang, X., von Werra, L., and Longpre, S. Octopack: Instruction tuning code large language models, 2023.

Ni, A., Allamanis, M., Cohan, A., Deng, Y., Shi, K., Sutton, C., and Yin, P. Next: Teaching large language models to reason about code execution, 2024. URL https://arxiv.org/abs/2404.14662.

Novikov, A., Vũ, N., Eisenberger, M., Dupont, E., Huang, P.-S., Wagner, A. Z., Shirobokov, S., Kozlovskii, B., Ruiz, F. J. R., Mehrabian, A., Kumar, M. P., See, A., Chaudhuri, S., Holland, G., Davies, A., Nowozin, S., Kohli, P., and Balog, M. Alphaevolve: A coding agent for scientific and algorithmic discovery. *Google DeepMind White Paper*, May 2025.

OpenAI, Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., Avila, R., Babuschkin, I., Balaji, S., Balcom, V., Baltescu, P., Bao, H., Bavarian, M., Belgum, J., Bello, I., Berdine, J., Bernadett-Shapiro, G., Berner, C., Bogdonoff, L., Boiko, O., Boyd, M., Brakman, A.-L., Brockman, G., Brooks, T., Brundage, M., Button, K., Cai, T., Campbell, R., Cann, A., Carey, B., Carlson, C., Carmichael, R., Chan, B., Chang, C., Chantzis, F., Chen, D., Chen, S., Chen, R., Chen, J., Chen, M., Chess, B., Cho, C., Chu, C., Chung, H. W., Cummings, D., Currier, J., Dai, Y., Decareaux, C., Degry, T., Deutsch, N., Deville, D., Dhar, A., Dohan, D., Dowling, S., Dunning, S., Ecoffet, A., Eleti, A., Eloundou, T., Farhi, D., Fedus, L., Felix, N., Fishman, S. P., Forte, J., Fulford, I., Gao, L., Georges, E., Gibson, C., Goel, V., Gogineni, T., Goh, G., Gontijo-Lopes, R., Gordon, J., Grafstein, M., Gray, S., Greene, R., Gross, J., Gu, S. S., Guo, Y., Hallacy, C., Han, J., Harris, J., He, Y., Heaton, M., Heidecke, J., Hesse, C., Hickey, A., Hickey, W., Hoeschele, P., Houghton, B., Hsu, K., Hu, S., Hu, X., Huizinga, J., Jain, S., Jain, S., Jang, J., Jiang, A., Jiang, R., Jin, H., Jin, D., Jomoto, S., Jonn, B., Jun, H., Kaftan, T., Łukasz Kaiser, Kamali, A., Kanitscheider, I., Keskar, N. S., Khan, T., Kilpatrick, L., Kim, J. W., Kim, C., Kim, Y., Kirchner, J. H., Kiros, J., Knight, M., Kokotajlo, D., Łukasz Kondraciuk, Kondrich, A., Konstantinidis, A., Kosic, K., Krueger, G., Kuo, V., Lampe, M., Lan, I., Lee, T., Leike, J., Leung, J., Levy, D., Li, C. M., Lim, R., Lin, M., Lin, S., Litwin, M., Lopez, T., Lowe, R., Lue, P., Makanju, A., Malfacini, K., Manning, S., Markov, T., Markovski, Y., Martin, B., Mayer, K., Mayne, A., McGrew, B., McKinney, S. M., McLeavey, C., McMillan, P., McNeil, J., Medina, D., Mehta, A., Menick, J., Metz, L., Mishchenko, A., Mishkin, P., Monaco, V., Morikawa, E., Mossing, D., Mu, T., Murati, M., Murk, O., Mély, D., Nair, A., Nakano, R., Nayak, R., Neelakantan, A., Ngo, R., Noh, H., Ouyang, L., O'Keefe, C., Pachocki, J., Paino, A., Palermo, J., Pantuliano, A., Parascandolo, G., Parish, J., Parparita, E., Passos, A., Pavlov, M., Peng, A., Perelman, A., de Avila Belbute Peres, F., Petrov, M., de Oliveira Pinto, H. P., Michael, Pokorny, Pokrass, M., Pong, V. H., Powell, T., Power, A., Power, B., Proehl, E.,

Puri, R., Radford, A., Rae, J., Ramesh, A., Raymond, C., Real, F., Rimbach, K., Ross, C., Rotsted, B., Roussez, H., Ryder, N., Saltarelli, M., Sanders, T., Santurkar, S., Sastry, G., Schmidt, H., Schnurr, D., Schulman, J., Selsam, D., Sheppard, K., Sherbakov, T., Shieh, J., Shoker, S., Shyam, P., Sidor, S., Sigler, E., Simens, M., Sitkin, J., Slama, K., Sohl, I., Sokolowsky, B., Song, Y., Staudacher, N., Such, F. P., Summers, N., Sutskever, I., Tang, J., Tezak, N., Thompson, M. B., Tillet, P., Tootoonchian, A., Tseng, E., Tuggle, P., Turley, N., Tworek, J., Uribe, J. F. C., Vallone, A., Vijayvergiya, A., Voss, C., Wainwright, C., Wang, J. J., Wang, A., Wang, B., Ward, J., Wei, J., Weinmann, C., Welihinda, A., Welinder, P., Weng, J., Weng, L., Wiethoff, M., Willner, D., Winter, C., Wolrich, S., Wong, H., Workman, L., Wu, S., Wu, J., Wu, M., Xiao, K., Xu, T., Yoo, S., Yu, K., Yuan, Q., Zaremba, W., Zellers, R., Zhang, C., Zhang, M., Zhao, S., Zheng, T., Zhuang, J., Zhuk, W., and Zoph, B. Gpt-4 technical report, 2024. URL https://arxiv.org/abs/2303.08774.

Peng, Y., Wan, J., Li, Y., and Ren, X. Coffe: A code efficiency benchmark for code generation, 2025. URL https://arxiv.org/abs/2502.02827.

Puri, R., Kung, D. S., Janssen, G., Zhang, W., Domeniconi, G., Zolotov, V., Dolby, J., Chen, J., Choudhury, M., Decker, L., Thost, V., Buratti, L., Pujar, S., Ramji, S., Finkler, U., Malaika, S., and Reiss, F. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks, 2021. URL https://arxiv.org/abs/2105.12655.

Romera-Paredes, B., Barekatain, M., Novikov, A., Balog, M., Kumar, M. P., Dupont, E., Ruiz, F. J., Ellenberg, J. S., Wang, P., Fawzi, O., et al. Mathematical discoveries from program search with large language models. *Nature*, 625 (7995):468–475, 2024.

Sasnauskas, R., Chen, Y., Collingbourne, P., Ketema, J., Lup, G., Taneja, J., and Regehr, J. Souper: A synthesizing superoptimizer, 2018. URL https://arxiv.org/abs/1711.04422.

Schkufza, E., Sharma, R., and Aiken, A. Stochastic superoptimization. In *ACM SIGARCH Computer Architecture News*, volume 41, pp. 305–316. ACM, 2013.

Sharma, A. Openevolve: Open-source implementation of alphaevolve. https://github.com/codelion/openevolve, 2025. Software, version 1.0.0.

Shypula, A., Madaan, A., Zeng, Y., Alon, U., Gardner, J., Hashemi, M., Neubig, G., Ranganathan, P., Bastani, O., and Yazdanbakhsh, A. Learning performance-improving code edits, 2024. URL https://arxiv.org/abs/2302.07867.

Singer, L. P. and Price, L. R. Rapid bayesian position reconstruction for gravitational-wave transients. *Phys. Rev. D*, 93:024013, Jan 2016. doi: 10.1103/PhysRevD.93.024013. URL https://link.aps.org/doi/10.1103/PhysRevD.93.024013.

Tang, X., Liu, Y., Cai, Z., Shao, Y., Lu, J., Zhang, Y., Deng, Z., Hu, H., An, K., Huang, R., Si, S., Chen, S., Zhao, H., Chen, L., Wang, Y., Liu, T., Jiang, Z., Chang, B., Fang, Y., Qin, Y., Zhou, W., Zhao, Y., Cohan, A., and Gerstein, M. Ml-bench: Evaluating large language models and agents for machine learning tasks on repository-level code, 2024. URL https://arxiv.org/abs/2311.09835.

Truong, C., Oudre, L., and Vayatis, N. Selective review of offline change point detection methods. *Signal Processing*, 167:107299, 2020. ISSN 0165-1684. doi: https://doi.org/10.1016/j.sigpro.2019.107299. URL https://www.sciencedirect.com/science/article/pii/S0165168419303494.

Waghjale, S., Veerendranath, V., Wang, Z., and Fried, D. ECCO: Can we improve model-generated code efficiency without sacrificing functional correctness? In Al-Onaizan, Y., Bansal, M., and Chen, Y.-N. (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 15362–15376, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.859. URL https://aclanthology.org/2024.emnlp-main.859/.

Yang, J., Prabhakar, A., Narasimhan, K., and Yao, S. Intercode: Standardizing and benchmarking interactive coding with execution feedback, 2023.

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., and Press, O. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024. URL https://arxiv.org/abs/2405.15793.

Yao, S. *Language Agents: From Next-Token Prediction to Digital Automation*. PhD thesis, Princeton University, 2024.

Yin, P., Li, W.-D., Xiao, K., Rao, A., Wen, Y., Shi, K., Howland, J., Bailey, P., Catasta, M., Michalewski, H., Polozov, A., and Sutton, C. Natural language to code generation in interactive data science notebooks, 2022.

Zan, D., Chen, B., Yang, D., Lin, Z., Kim, M., Guan, B., Wang, Y., Chen, W., and Lou, J.-G. Cert: Continual pre-training on sketches for library-oriented code generation, 2022.

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.