UT-Evolve: AN EVOLUTIONARY AGENT FOR UNIT TEST WRITING

Arshika Lalan*
Nutanix, Inc
San Jose, CA 95134, USA
arshika.lalan@nutanix.com

Rajat Ghosh
Nutanix, Inc
San Jose, CA 95134, USA
rajat.ghosh@nutanix.com

Debojyoti Dutta
Nutanix, Inc
San Jose, CA 95134, USA
debojyoti.dutta@nutanix.com

ABSTRACT

We present **UT-Evolve**, an evolutionary agent that improves the unit test generation capability of large language models through iterative interaction with a testing environment. UT-Evolve orchestrates a stateful agent loop that refines unit tests over multiple iterations using feedback from prior executions, including failures and coverage signals. In contrast to most existing coding agents, which rely on stateless, single-step inference, UT-Evolve maintains persistent state across generations, enabling long-horizon reasoning and adaptive test generation strategies.

We argue that the absence of persistent state fundamentally limits LLM performance on tasks such as unit test generation, where effective solutions require the accumulation of task-specific knowledge and the adaptation of dynamic strategies over time. Rather than producing predominantly surface-level happy-path tests, UT-Evolve prioritizes edge cases and systematically probes latent assumptions, using observed failures to evolve its test generation policy.

In a **preliminary** evaluation on a filtered version of TestGenEval (TestGenEvalMini), UT-Evolve achieves improved test coverage compared to single-step stateless baselines across multiple LLM families. These results suggest that UT-Evolve offers a promising direction for enabling generative models to learn effective behaviors through sustained interaction with complex environments.

1 INTRODUCTION

Difficult problems are rarely solved in a single pass; they require iteration, feedback, and refinement. Nevertheless, most inference-time computation in large language models (LLMs) remains stateless, with each inference call discarding intermediate reasoning unless it is explicitly re-injected into the prompt. While this design choice optimizes deployment throughput, it significantly constrains performance in domains that require deep, multi-stage reasoning—such as program synthesis, theorem proving, multi-hop reasoning, deductive reasoning, and mathematical problem-solving—where intermediate states must be persistently updated and revisited. The fixed computational depth of a transformer forward pass (Vaswani et al., 2017), together with the well-documented decline in reasoning fidelity over long logical chains (Wei et al., 2022; Anil et al., 2022), renders these limitations structural rather than incidental. Autoregressive decoding further restricts exploration by forcing reasoning branches to unfold serially, often necessitating brittle orchestration through multiple model calls (Yao et al., 2023a; Long et al., 2024). Prior works have proposed stateful inference-time mechanisms—including scratchpad prompting (Nye et al., 2021; Wei et al., 2022), tree-structured reasoning (Yao et al., 2023c), and retrieval-augmented agents (Lewis et al., 2020)—yet these approaches continue to elicit reasoning from fixed, opaque model parameters, limiting both steerability (Zhou et al., 2023) and interpretability (Olah et al., 2020; Nanda et al., 2023).

*Internship work at Nutanix

Table 1: Comparison of inference-time reasoning frameworks.

Method	Training	Persistent State	Grounded	Evolution
CoT (Wei et al., 2022)	✗	✗	✗	✗
ReAct (Yao et al., 2023d)	✗	✓	✓	✗
ToT (Yao et al., 2023b)	✗	✗	✗	✗
AlphaEvolve (Novikov et al., 2025b)	✗	✗	✓	✓
UT-Evolve	✗	✓	✓	✓

This lack of reasoning depth becomes particularly acute in logical reasoning tasks such as unit test generation at the repository level (Jain et al., 2025). We address these challenges with a training-free framework that unifies (i) multi-agent reasoning, (ii) adversarial mutation and reward shaping, and (iii) evolutionary preservation with persistent state. An actor proposes candidate edge cases by reasoning over program semantics, an adversary perturbs code to expose latent failure modes, and a critic integrates coverage, exception signals, and mutation feedback to prioritize high-value test cases. A non-Markovian controller maintains memory of prior edge cases and preserves high-fitness candidates across iterations, enabling inference-time policy adaptation and robust exploration. This design allows the system to dynamically adapt to unseen codebases, generate robust edge cases, and achieve higher coverage than existing methods—without gradient-based training or domain-specific fine-tuning.

Our contributions are threefold:

- We introduce **UT-Evolve**, an evolutionary, multi-agent framework for repository-level unit test generation with persistent inference-time state.
- We construct **TestGenEvalMini**, a curated benchmark for evaluating unit test generation under realistic repository-level settings.
- We empirically evaluate UT-Evolve across multiple LLM families and agentic architectures, establishing a preliminary empirical foundation for future in-depth investigation.

2 Related Work

Inference-Time Reasoning with LLMs. A growing line of work improves reasoning in large language models (LLMs) through inference-time prompting and orchestration without modifying model parameters. Chain-of-Thought (CoT) prompting (Wei et al., 2022) elicits intermediate reasoning steps but remains fundamentally stateless, with no mechanism to preserve or reuse reasoning across calls. Tree-of-Thoughts (ToT) (Yao et al., 2023b) improves exploration by branching and evaluating multiple reasoning paths, yet discards the resulting structures after each run. ReAct (Yao et al., 2023d) introduces grounded interaction by interleaving reasoning with environment actions, but maintains only transient state within a single execution trajectory. More broadly, scratchpad prompting (Nye et al., 2021; Wei et al., 2022), structured planning (Yao et al., 2023c), and retrieval-augmented generation (Lewis et al., 2020) partially mitigate depth and exploration limits, but continue to elicit reasoning from fixed, opaque model parameters, constraining steerability (Zhou et al., 2023) and interpretability (Olah et al., 2020; Nanda et al., 2023).

Multi-Agent and Evolutionary Reasoning. Recent multi-agent frameworks demonstrate that collaborative reasoning and role specialization can improve exploration and robustness. Systems such as AI Co-scientist (Gottweis et al., 2025), AlphaEvolve (Novikov et al., 2025b), and GEPA (Agrawal et al., 2025) employ reflective prompt evolution or population-based coordination to enhance problem solving. However, most rely on transient coordination mechanisms—such as prompt-level feedback or heuristic selection—rather than explicitly preserving evaluative knowledge across iterations. Evolutionary and search-based methods address this limitation by retaining high-fitness candidates and explicitly managing the exploration–exploitation tradeoff (Mühlenbein et al., 1988; Burnim & Sen, 2008; Mouret & Clune, 2015; Salimans et al., 2017). Recent work has explored integrating evolutionary search with LLM-based agents (Such et al., 2017; Karten et al.; Leng et al., 2024; Wen et al., 2024), but these approaches often evaluate generations in isolation and lack structured mechanisms for accumulating and reusing evaluation signals over long optimization horizons.

Unit Test Generation. Automated unit test generation has traditionally relied on symbolic execution, fuzzing, and heuristic search to maximize coverage and expose edge cases (Burnim & Sen, 2008). LLM-based test generators have shown promise on small-scale tasks but typically operate in a feed-forward or weakly iterative manner, producing surface-level tests and struggling with repository-level reasoning. Benchmarking efforts such as TestGenEval (Jain et al., 2025) highlight the difficulty of real-world unit test generation, where effective testing requires sustained reasoning over program structure, dependencies, and failure modes.

Positioning of UT-Evolve. Table 1 contrasts UT-Evolve with representative inference-time reasoning frameworks. We reserve *persistent state* for mechanisms that explicitly preserve and reuse evaluative knowledge—such as fitness signals or value estimates—across iterations, rather than transient prompt context or execution traces. Unlike prior approaches, UT-Evolve combines multi-agent reasoning, adversarial mutation, and evolutionary preservation with persistent inference-time state, enabling systematic accumulation of task-specific knowledge and adaptive test generation without gradient-based training or domain-specific fine-tuning.

3 METHODOLOGY

Our central premise is that generating syntactically correct unit tests is largely mechanical once a set of robust edge cases with sufficient coverage are identified. Figure 1 shows the architecture of the unit test generation engine with the proposed *stateful search* for edge case generation. Given source code f, the system first runs the stateful search to extract edge cases iteratively and finally converts those cases into unit tests.

Our stateful multi-agent evolutionary search is an adversarially guided actor-critic (AGAC) system that operates entirely at inference time and does not require gradient-based learning. The **Actor** issues multiple LLM inference calls to propose candidate edge cases, the **Adversary** perturbs the environment to reveal robustness gaps, and the **Critic** assigns scalar rewards used for evolutionary search. The **Executor** is an auxiliary agent that provides an execution environment to execute edge cases, unit tests, and return coverage and robustness feedback. These four agents are orchestrated through the **Controller** which maintains persistent state across stages and orchestrates the search until convergence.

Definition 1 (State)**.** A State is represented in Equation 1,

$$S_{n-1} = \Big(\zeta_{1:n-1}, \mu_{1:n-1}, \kappa_{1:n-1}, c_{1:n-1}, R_{1:n-1}\Big) \tag{1}$$

where $\zeta_{1:n-1}$ denotes the sequence of prior edge cases, $\mu_{1:n-1}$ is the sequence of mutation scores, $\kappa_{1:n-1}$ is the sequence of coverage scores, $c_{1:n-1}$ is the sequence of exception signals, and $R_{1:n-1}$ is the reward history from previous stages.

Definition 2 (Actor)**.** The Actor (A_n) proposes candidate edge cases at each stage. At initialization ($n = 1$), there is no prior feedback or state information to guide generation, so the actor is seeded deterministically (cold-start) using rule-based heuristics such as boundary partition analysis, equivalence classes, and stress conditions. For $n > 1$, the actor generates candidates through large language model in-context learning, conditioned on the persistent state S_{n-1} and the source code f:

$$\zeta_n = \mathcal{A}(f, S_{n-1}) \tag{2}$$

More details about the cold-start can be found in Appendix D.

Definition 3 (Adversary)**.** For each stage, Adversary (D_n) generates a set of mutants $\{f'_{n,j}\}_{j=1}^{M}$ of the source file, and evaluates whether the edge cases ζ_n can kill these mutants (i.e, produce a different output on $f'_{n,j}$ than they did on f). The resulting mutation score is defined in Equation 3 and provides a robustness signal for evaluating the edge case candidates. Mutation testing promotes robustness by checking whether tests can distinguish the true program from systematically perturbed variants, preventing the search from optimizing toward shallow coverage gains.

$$\mu_n = \frac{\text{Number of mutants killed by } \zeta_n}{\text{Total number of generated mutants}} \tag{3}$$

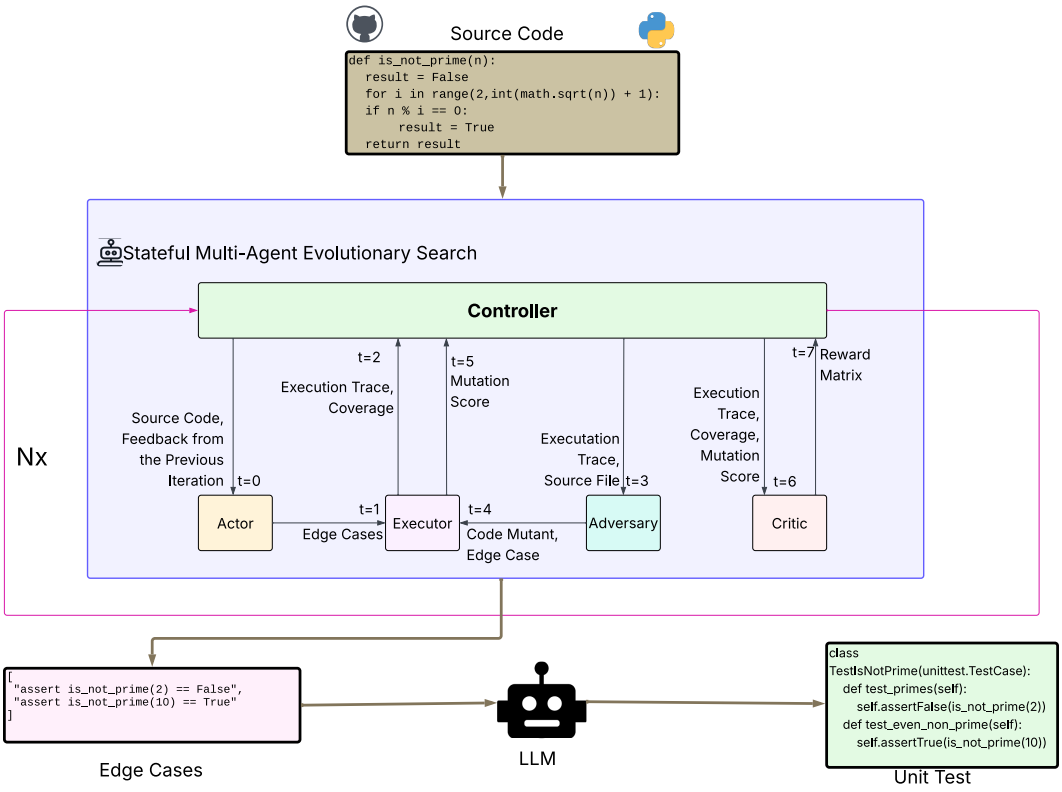

Figure 1: Our architecture for unit test generation decomposes the task into two phases: (i) edge case generation from source code and (ii) unit test construction from those cases. The first phase demands deeper reasoning and is addressed through an evolutionary search (as highlighted in the blue box) executed in a stateful manner over multiple stages ($N\times$) by four agents—Actor, Executor, Adversary, and Critic—coordinated by a Controller that propagates persistent state across N evolutionary stages (as highlighted by the magenta line). Once the edge cases converge to sufficient coverage and robustness, they are translated into a complete unit test file via a single-step inference call.

Definition 4 (Critic)**.** For each stage, Critic ($\mathcal{C}_n$) computes the scalar reward for the edge cases by integrating coverage (κ), mutation robustness (μ), and exception discovery (c), given by Equation 4.

$$R_n^{\text{unnormalized}}(\kappa_n, \mu_n, c_n) = [\alpha \cdot c_n + \beta(\kappa_n + \max(0, (\kappa_n - \theta) \cdot 0.5))] \times \gamma \cdot \mu_n \tag{4}$$

where $\alpha, \beta, \theta, \gamma \in \mathbb{R}_+$ are tunable hyperparameters. All rewards are normalized to $[0, 1]$ using min–max normalization, where $R_{\min}$ and $R_{\max}$ are tracked across all evaluated candidates up to stage n for evolutionary comparison.

The reward combines exception discovery (c_n), structural coverage (κ_n), and mutation robustness (μ_n). The exception term encourages exploration of inputs that expose faults. The coverage term accounts for the proportion of program elements exercised, with an additional bonus once a minimum threshold θ is passed, so that progress beyond trivial coverage is reflected more strongly. Multiplication by the mutation score ensures that high reward is assigned only when the generated tests are also robust to program perturbations. By shaping the critic's reward surface using adversarial perturbations, we ground the actor's responses and reduce the risk of optimizing for shallow coverage gains instead of exploring robust, high-value edge cases. We do not claim optimality of this reward formulation; it serves as a heuristic to guide inference-time exploration.

Definition 5 (Executor)**.** All evaluations for coverage and mutation scoring are executed in a sandboxed Docker environment with a Model-Context Protocol (MCP) server. This provides: (i) **Iso-**

lation: Mutants and edge cases cannot harm the host system; (ii) **Determinism:** Results are reproducible across runs; and (iii) **Bounded resources:** Memory and timeouts prevent unbounded execution. A detailed description of the Executor architecture can be found in Appendix A.5.

Definition 6 (Controller)**.** The controller orchestrates the interplay of Actor, Adversary, and Critic by updating the non-Markovian state information (Equation 1) and checking for the termination criteria as defined in Equation 5.

$$\sum_i R_i \geq \tau \;\text{ or }\; \max_{i \in [n-p+1,\, n]} R_i \;-\; \min_{i \in [n-p+1,\, n]} R_i \leq \delta \tag{5}$$

Here, $\sum_i R_i$ denotes the cumulative reward of elite candidates retained by the controller up to stage n. The controller applies two complementary stopping conditions. The first checks whether the reward has crossed a predefined threshold, indicating that the search has reached a sufficient overall quality level. The second detects a plateau in rewards over the most recent p iterations, suggesting that further search is unlikely to yield substantial improvements. The plateau condition is evaluated only when $n \geq p$. The thresholds (τ, δ) and window size p can be tuned according to task complexity as well as computational budget, allowing the framework to balance thoroughness and efficiency.

A key methodological contribution is that our framework does not require training, fine-tuning, or task-specific adaptation of large language models. Instead, it builds a training-free test-generation agent whose effective behavior emerges from inference-time state management, multi-agent grounding, and evolutionary selection. This positions our framework alongside recent agentic paradigms such as AI Co-Scientist (Gottweis et al., 2025) and AlphaEvolve (Novikov et al., 2025a), while differing in its explicit use of evolutionary preservation and adversarial reward shaping to structure inference-time coordination. Algorithm 1 shows the overall computational framework that we use for multi-stage evolutionary search.

4 Experiments

We evaluate the proposed evolutionary search framework on two benchmarks—HumanEval and TestGenEvalMini—using *three* large language models (LLMs): Llama-70B, GPT-o4-mini, and Gemma-2-27B. The method is compared against *six* inference-time baselines: zero-shot, one-shot, and three-shot prompting, each with and without chain-of-thought (CoT). Performance is assessed using standard structural coverage metrics, including line, branch, and function coverage. All experiments are executed in a sandboxed Docker/MCP environment to ensure isolation, deterministic execution, and reproducibility.

HumanEval (Chen et al., 2021) comprises 164 Python programming tasks with reference implementations and is used to assess the correctness and robustness at the function-level under controlled conditions. To evaluate repository-level reasoning, we curated *TestGenEvalMini*, a 48-task subset of TestGenEvalLite (Zhang et al., 2024), which is derived from SWE-Bench (Jimenez et al., 2024). TestGenEvalLite contains real-world code and test pairs from 11 maintained Python repositories. TestGenEvalMini excludes modules with unstable execution behavior (e.g., repeated rapid tool calls or complex dependencies between modules) to ensure reliable evaluation while preserving key structural characteristics of the full benchmark, including code length, function count, and branching complexity (Appendix, Table 4). We also release curated versions of both datasets augmented with edge-case execution traces and associated coverage, mutation, and exception metadata to support reproducible evaluation and future work on agentic reasoning inference-time for software testing.

5 Results

We conducted a preliminary feasibility study of UT-Evolve on the HumanEval benchmark, with results reported in Appendix (Table 3). Since HumanEval consists of standalone function-level programming tasks, the potential gains from iterative test evolution are inherently limited. Consequently, improvements observed under this setting should be interpreted as conservative estimates of UT-Evolve's effectiveness in more realistic, multi-file or stateful development scenarios.

Algorithm 1 UT-Evolve for Unit Test Generation

Require: Source file f
Ensure: Final Unit Test File UT

Initialize $n \leftarrow 1$, $S_0 \leftarrow \emptyset$, $R_0 \leftarrow 0$
while not ShouldStop($\{R_1, \ldots, R_{n-1}\}, n-1$) **do**
  Actor:

$$\zeta_n = \begin{cases} A(f) & n = 1 \quad \text{(cold start: rule-based heuristics)} \\ A(f, S_{n-1}) & n > 1 \end{cases}$$

  Executor: Run ζ_n on f to obtain execution traces ρ_n, coverage κ_n, and exception signals c_n.
  Adversary: Mutate f into $\{f'_{n,1}, \ldots, f'_{n,M}\}$, execute ζ_n, and compute

$$\mu_n = \frac{K_n}{K_n + U_n}$$

  where K_n and U_n are the numbers of killed and surviving mutants, respectively.
  Executor: Compute exception signals $c_n = \text{ExceptionSignal}(\rho_n)$
  Critic: Compute reward

$$R_n^{\text{unnorm}}(\kappa_n, \mu_n, c_n) = [\alpha \cdot c_n + \beta(\kappa_n + \max(0, (\kappa_n - \theta) \cdot 0.5))] \times \gamma \cdot \mu_n$$

$$R_n = \frac{R_n^{\text{unnorm}} - R_{\min}}{R_{\max} - R_{\min}}$$

  Update archive: retain top-K edge cases from $\zeta_{1..n}$ by reward R_n

$$\zeta_{1:n} \leftarrow \text{top-}K\big(\zeta_{1:n}, \text{sorted by } R_{1:n}\big)$$

  Set $n \leftarrow n + 1$
  Update state:

$$S_n = (\zeta_{1..n}, \mu_{1..n}, \kappa_{1..n}, c_{1..n}, R_{1..n})$$

end while
Synthesis: UT $\leftarrow$ LLM(f, S_n)
return UT

Function ShouldStop($\{R_1, \ldots, R_n\}, m$):

$$\textbf{if } m < p \textbf{ then return } \Big(\sum_{i=1}^{m} R_i \geq \tau\Big)$$

$$\textbf{else return } \Big(\sum_{i=1}^{m} R_i \geq \tau\Big) \ \vee \ \Big(\max_{i \in [m-p+1,\, m]} R_i - \min_{i \in [m-p+1,\, m]} R_i \leq \delta\Big)$$

5.1 TESTGENEVALMINI

Table 2 reports the test coverage statistics achieved by UT-Evolve using three backbone models, compared against six inference-time baselines. Figure 2 further illustrates the final coverage-based edge-case quality obtained through evolutionary search relative to these baselines. From these results, we draw the following observations:

UT-Evolve Demonstrates Consistent Performance Gains. Across models and coverage metrics, UT-Evolve achieves strong performance, securing 7 out of 9 possible podium positions (top-3 rankings). This consistency suggests that iterative evolutionary refinement provides measurable improvements over purely inference-time prompting strategies.

Table 2: Coverage results on TestGenEvalMini across line, branch, and function coverage. Within each metric column, the top three methods are ranked and highlighted using Olympic medal colors (gold = 1st; silver = 2nd; bronze = 3rd.)

TestGenEvalMini	Line Coverage	Branch Coverage	Function Coverage
UT-Evolve SUT Llama 70B	**29.80%**	16.55%	**29.24%**
UT-Evolve SUT o4-mini	28.22%	15.28%	**27.78%**
UT-Evolve SUT Gemma-2-27B	**26.95%**	14.88%	28.05%
Zero Shot LLM	22.59%	15.45%	24.62%
Zero Shot LLM w/ CoT	22.31%	16.02%	22.83%
One Shot LLM	25.22%	14.95%	26.58%
One Shot LLM w/ CoT	25.24%	15.22%	27.28%
Three Shot LLM	25.35%	**17.40%**	26.83%
Three Shot LLM w/ CoT	24.66%	**16.21%**	25.80%

Relative Weakness in Branch Coverage. UT-Evolve exhibits comparatively weaker performance in branch coverage, particularly when paired with o4-mini and Gemma-2-27B. One possible explanation is that the evolutionary search process tends to prioritize exception-triggering or assertion-heavy test cases. Such tests may thoroughly exercise specific control-flow paths while failing to systematically explore complementary branches. Although this behavior may limit branch coverage, it can still uncover deeper semantic or functional edge cases reflected in line and function coverage metrics.

Performance Scaling with Model Capacity. UT-Evolve achieves its strongest results when paired with Llama 70B. A plausible explanation is that larger models provide richer proposal distributions during edge-case generation and more reliable critic feedback during iterative refinement. This suggests that evolutionary search may benefit from higher-capacity backbones, though further controlled studies would be required to isolate scaling effects.

Figure 3 presents the resolution rate (blue, left axis) and average runtime (red, right axis) for two benchmarks. The **resolution rate** is defined as the fraction of generated unit tests that successfully reach convergence. In the left subplot, HUMANEVAL shows that nearly 62% of problems are resolved in a single iteration, with only modest runtime overhead, indicating that the majority of tasks are relatively straightforward. In contrast, the right subplot for TESTGENEVALMINI exhibits a markedly different profile: while the majority of problems require three or more iterations, resolution rates plateau only after extended search, with runtimes rising steeply at higher iteration counts. Together, these results demonstrate that the evolutionary search is efficient on simpler benchmarks while scaling to more complex repository-level tasks with increased inference-time compute. The prompts can be found in Appendix B and an example unit test file generation can be found in Appendix C.

6 CONCLUSION

We introduced UT-Evolve, an evolutionary inference-time framework for unit test generation that departs from stateless prompting by maintaining persistent reasoning state across multiple stages of search. By coordinating an actor for edge-case proposal, an adversary for robustness evaluation, a critic for reward integration, and an executor for sandboxed verification, the system enables structured exploration beyond single-call prompting. Experiments on HumanEval and TestGenEvalMini show that while simple problems are often solved at initialization, stateful evolutionary search yields consistent improvements on more challenging repository-level tasks, particularly in coverage-driven edge-case discovery. UT-Evolve introduces a structured evolutionary feedback loop that enables iterative refinement based on execution signals, moving toward a principled framework for in-task experiential learning at inference time.

Despite these gains, the framework incurs additional inference-time compute and latency costs inherent to multi-stage search. Future work will focus on improving termination criteria for more compute-efficient convergence, extending the executor to handle richer cross-module dependencies, and exploring learned reward models to stabilize scoring. Broader evaluation across multilingual

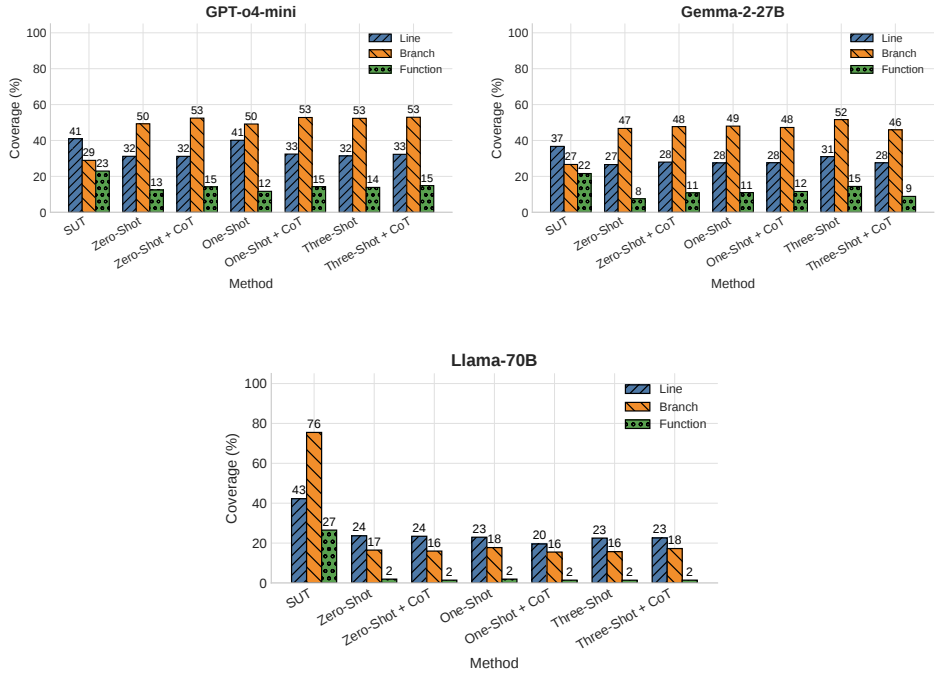

Figure 2: Final edge case quality on TESTGENEVALMINI measured in terms of line, branch, and function coverages across three model families: GEMMA-2-27B (top-left), GPT-O4-MINI (top-right), and LLAMA-70B (bottom). The proposed inference-time evolutionary search (SUT) consistently achieves strong coverage, outperforming few-shot and chain-of-thought baselines in most settings.

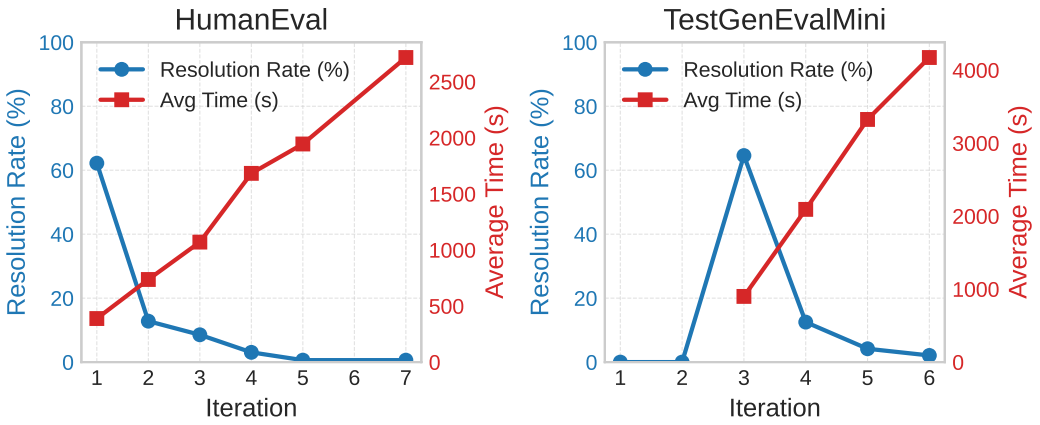

Figure 3: Resolution rate (blue, left axis) and average runtime (red, right axis) per iteration for HumanEval (left) and TestGenEvalMini (right). Higher resolution rate indicates a larger fraction of problems that converge to valid unit tests. Average runtime grows with iteration count as the stateful multi-agent search explores deeper inference-time trajectories.

and industrial-scale repositories will be necessary to assess generalization and practical deployment readiness.

REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. Gepa: Reflective prompt evolution can outperform reinforcement learning, 2025. URL https://arxiv.org/abs/2507.19457.

Cem Anil, James Lucas, and Roger Grosse. Exploring length generalization in large language models. *arXiv preprint arXiv:2207.04901*, 2022.

Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 443–446. IEEE, 2008.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. 2021.

Juraj Gottweis, Wei-Hung Weng, Alexander Daryin, Tao Tu, Anil Palepu, Petar Sirkovic, Artiom Myaskovsky, Felix Weissenberger, Keran Rong, Ryutaro Tanno, Khaled Saab, Dan Popovici, Jacob Blum, Fan Zhang, Katherine Chou, Avinatan Hassidim, Burak Gokturk, Amin Vahdat, Pushmeet Kohli, Yossi Matias, Andrew Carroll, Kavita Kulkarni, Nenad Tomasev, Yuan Guan, Vikram Dhillon, Eeshit Dhaval Vaishnav, Byron Lee, Tiago R D Costa, José R Penadés, Gary Peltz, Yunhan Xu, Annalisa Pawlosky, Alan Karthikesalingam, and Vivek Natarajan. Towards an ai co-scientist, 2025. URL https://arxiv.org/abs/2502.18864.

Kush Jain, Gabriel Synnaeve, and Baptiste Rozière. Testgeneval: A real world unit test generation and test completion benchmark, 2025. URL https://arxiv.org/abs/2410.00752.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024. URL https://arxiv.org/abs/2310.06770.

Seth Karten, Andy Luu Nguyen, and Chi Jin. Pokéchamp: an expert-level minimax language agent for competitive pokémon.

Yan Leng, Hao Wang, and Yuan Yuan. Llm-assisted hypothesis generation and graph-based evaluation. *Available at SSRN 4948029*, 2024.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Marcin Kučer, Sewon Min, Wen-tau Yih, Hannaneh Hajishirzi, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Tao Long, Wei Zheng, Jia Li, Xing Wang, Liang Zhao, Zhiyuan Liu, and Maosong Sun. Trime: Trimming llms for efficient multi-step reasoning. *arXiv preprint arXiv:2402.07644*, 2024.

Jean-Baptiste Mouret and Jeff Clune. Encouraging behavioral diversity in evolutionary robotics: An empirical study. *Evolutionary computation*, 23(3):493–524, 2015.

Heinz Mühlenbein, Martina Gorges-Schleuter, and Ottmar Krämer. Evolution algorithms in combinatorial optimization. *Parallel computing*, 7(1):65–85, 1988.

Neel Nanda, Lawrence Chan, Joseph Smith, Tim Lieberum, Nelson Elhage, James Johnston, Daniel Wang, Marcin Tworkowski, Trenton Bricken, Ethan Perez, et al. Progress measures for grokking via mechanistic interpretability. *arXiv preprint arXiv:2301.05217*, 2023.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025a.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025b. URL https://arxiv.org/abs/2506.13131.

Maxwell Nye, Anders Andreassen, Iddo Gur, Michael Widrich, Melanie Kambadur, Edward Grefenstette, Pushmeet Kohli, Thomas Kipf, and Tim Rocktäschel. Show your work: Scratchpads for intermediate computation with language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

Chris Olah, Nick Cammarata, Ludwig Schubert, Gabriel Goh, Michael Petrov, and Shan Carter. Zoom in: An introduction to circuits. *Distill*, 5(3):e00024, 2020.

Tim Salimans, Jonathan Ho, Xi Chen, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. In *arXiv preprint arXiv:1703.03864*, 2017.

Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. In *arXiv preprint arXiv:1712.06567*, 2017.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chen, Quoc Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.

Jiaxin Wen, Jian Guan, Hongning Wang, Wei Wu, and Minlie Huang. Codeplan: Unlocking reasoning potential in large language models by scaling code-form planning. In *The Thirteenth International Conference on Learning Representations*, 2024.

Shunyu Yao, Maarten Bosma, Zifan Zhao, Dian Yu, Jeffrey Wen, Pranav Kumar, Kevin Luu, Karthik Narasimhan, Melanie Kambadur, Yuan Cao, et al. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2023a.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023b. URL https://arxiv.org/abs/2305.10601.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023c.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023d. URL https://arxiv.org/abs/2210.03629.

Yunhua Zhang, Zhiqiang Zeng, Yujie Luo, Xiangzhe Chen, Zhenchang Liu, Zhewei Wang, Xiang Li, Ge Li, Zhiqiang Zong, Xiaoxing Ma, and Lingming Zhang. Testgeneval: A real world unit test generation and test completion benchmark. *arXiv preprint arXiv:2410.00752*, 2024.

Denny Zhou, Quoc Le, Ed Chen, Jason Wei, et al. We can steer but not explain: When interpretable models are hard to train. *arXiv preprint arXiv:2304.05366*, 2023.

Table 3: Final Edge Case Quality for HumanEval for Llama 70B

HumanEval	Line Coverage	Branch Coverage	Function Coverage
SUT	90.01%	89.76%	91.51%
Zero Shot LLM	82.77%	81.92%	85.36%
Zero Shot LLM with CoT	86.90%	86.73%	87.5%
One Shot LLM	**90.85%**	**90.70%**	**92.07%**
One Shot LLM with CoT	87.21%	87.04%	88.41%
Three Shot LLM	89.94%	89.87%	90.09%
Three Shot LLM with CoT	88.18%	88.13%	89.33%

Table 4: Comparison of structural complexity metrics between TestGenEvalLite and TestGenEvalMini.

Metric	Lite (160 tasks, 11 repositories)	Mini (48 tasks, 6 repositories)
Code LOC	906.57 ± 821.67, median = 584	575.79 ± 600.78, median = 425
Functions	46.27 ± 53.80, median = 31	33.81 ± 37.38, median = 28
Branches	79.87 ± 84.46, median = 52	60.06 ± 70.57, median = 40

A Appendix

A.1 HumanEval

HumanEval consists of standalone, file-level implementations, where the scope for improvement from advanced inference-time strategies is inherently limited. As shown in Table 3, the system under test (SUT) and all six inference-time baselines achieve comparable coverage-based test quality, serving primarily as a non-regression check for our evolutionary search framework. Our evolutionary search method achieves comparable final coverage while requiring zero additional LLM calls in approximately 62% of cases. This highlights the effectiveness of the *cold-start* stage: : seeded by deterministic heuristics such as boundary partitioning and equivalence classes, it often produces high-quality edge cases without requiring iterative refinement. Consequently, most HumanEval problems are resolved at initialization, demonstrating the efficiency of our framework while motivating the use of more challenging benchmarks such as TestGenEvalMini to surface the benefits of stateful, multi-agent evolutionary reasoning.

A.2 TestGenEvalLite vs. TestGenEvalMini

A.3 Computation Cost (FLOPs)

We report floating-point operation counts (FLOPs) for a single evaluation **iteration** of our adaptive pipeline. FLOPs provide a hardware-agnostic measure of computational demand and allow principled comparison across model sizes and ablations.

We decompose the iteration into language-model (LLM) calls and non-LLM procedures (code execution, mutation, bookkeeping). For autoregressive transformer inference, we adopt the standard accounting

$$\text{FLOPS}_{\text{LLM}} \approx 2\mathcal{N}_{\text{params}} \cdot \mathcal{T}$$

where $\mathcal{N}_{\text{params}}$ is the number of model parameters and $\mathcal{T}$ is the total number of tokens processed (prompt + generated). The factor 2 reflects the dominant matrix multiplications in the forward pass. (If back-propagation were involved, a factor $\approx 3x$ the forward cost would be appropriate; our pipeline uses inference only.)

Non-LLM components are counted analytically from primitive operations in the relevant procedures (e.g., parsing, AST transforms, interpreter startup), yielding FLOPs that are negligible relative to LLM usage but included for completeness.

A.4 FLOPs Formulation

We derive the total floating point operations (FLOPs) required per iteration of our Actor–Adversary–Critic loop. Let:

- N_{actor}: number of parameters in the Actor LLM
- N_{ut}: number of parameters in the UnitTest LLM
- L_{src}: source code length (tokens)
- R: number of rule variations (edge cases) generated per iteration
- R_{ut}: maximum number of edge cases to keep for unittest generation
- M: max number of mutants (code mutations) executed per iteration
- T_{others}: average tokens of system prompt, task description etc
- T_{ec}: average tokens per edge case description
- $T_{\text{ut\_out}}$: output length of the generated unit test suite (tokens)
- F_{exec}: FLOPs per code execution
- F_{mut}: FLOPs per mutation generation
- F_{critic}: FLOPs per critic evaluation
- F_{other}: FLOPs for JSON parsing, string processing, and logging

1. Actor FLOPs. The Actor LLM processes both source code and accumulated context to generate new edge cases.

$$T_{\text{actor\_in}} = L_{\text{src}} + (R \cdot T_{\text{ec}}) + T_{\text{others}}$$

$$T_{\text{actor\_out}} = R \cdot T_{\text{ec}}$$

$$T_{\text{actor}} = T_{\text{actor\_in}} + T_{\text{actor\_out}}$$

$$F_{\text{actor}} = 2 \cdot N_{\text{actor}} \cdot T_{\text{actor}}$$

2. Unittest FLOPs. If the system makes use of an LLM to generate the final unittest file as opposed to a Human in the Loop, then these computations also need to be taken into account.
The UnitTest LLM consumes the source and filtered edge cases to produce complete test suites.

$$T_{\text{ut\_in}} = L_{\text{src}} + (R_{\text{ut}} \cdot T_{\text{ec}}) + T_{\text{others}}$$

$$T_{\text{ut}} = T_{\text{ut\_in}} + T_{\text{ut\_out}}$$

$$F_{\text{ut}} = 2 \cdot N_{\text{ut}} \cdot T_{\text{ut}}$$

3. Code Execution FLOPs. Each generated mutant and the original source are executed against all edge cases.

$$E = (M + 1) \cdot R$$

$$F_{\text{exec\_total}} = E \cdot F_{\text{exec}}$$

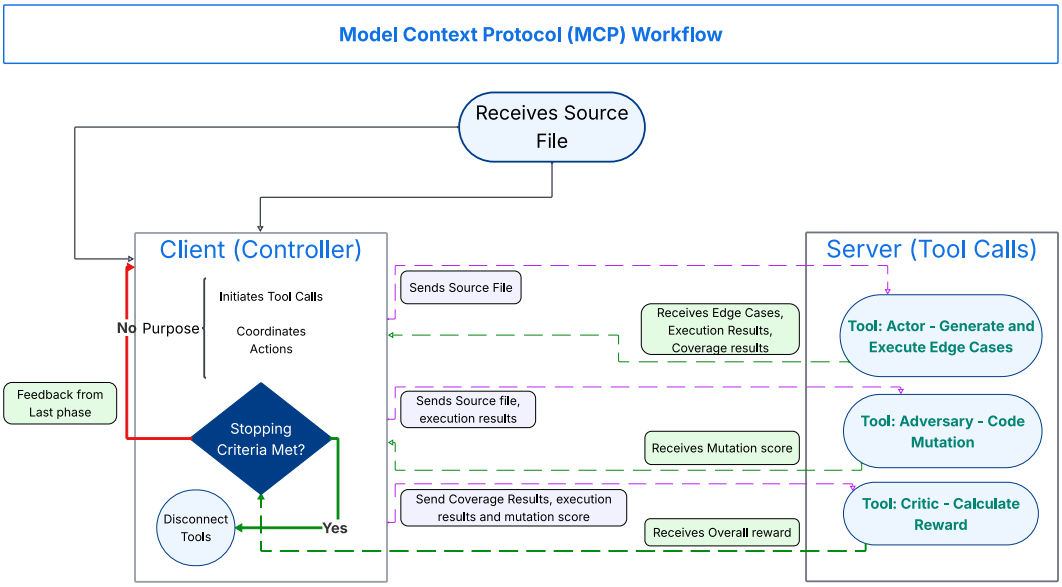

Figure 4: MCP Architecture overview

4. Mutation FLOPs. Considering an average of 30 mutants are generated in every iteration (after which M are randomly sampled for execution).

$$F_{\text{mut\_total}} = 30 \cdot F_{\text{mut}}$$

5. Critic and Other FLOPs.

$$F_{\text{critic\_total}} = R \cdot F_{\text{critic}}$$

$$F_{\text{other\_total}} = F_{\text{other}}$$

6. Total System FLOPs.

$$F_{\text{system}} = F_{\text{actor}} + F_{\text{ut}} + F_{\text{exec\_total}} + F_{\text{mut\_total}} + F_{\text{critic\_total}} + F_{\text{other\_total}}$$

This formulation allows us to compute FLOPs analytically for different evaluation settings, such as TESTGENEVALMINI and HUMANEVAL, by substituting the corresponding parameter values.

Running the system on TestGenEvalMini requires **3584.0 TFLOPs** per LLM Iteration, and an additional **819.2 TFLOPs** for the final unit test file generation. The TFLOPs for all other computation are negligible, including rule-based generation (which only requires an average of **36000 FLOPs**. Running the system on HumanEval requires **812.0 TFLOPs** per LLM Iteration, and an additional **128.0 TFLOPs** for the final unit test file generation. The TFLOPs for all other computation are negligible, including rule-based generation (which only requires an average of **13500 FLOPs**).

Category	TestGenEvalMini (TFLOPs)	HumanEval (TFLOPs)
LLM Iteration	3584.0	812.0
Final Unit Test Generation	819.2	128.0
Rule-based / Other Computation	0.036	0.0135

A.5 EXECUTOR

The ***Executor*** is an integral auxiliary component within our system architecture that facilitates the *Controller* in managing the orchestrated flow of information. It employs a Model Context Protocol (MCP) Client-Server framework to ensure secure and isolated execution of all generated edge

cases and mutated code variants. To maintain strict isolation, all executions on the server side are containerized using Docker, thereby sandboxing them from the host environment.

A.5.1 MCP WORKFLOW

The operational workflow of the Executor is depicted in Fig. 4 and proceeds as follows:

1. The Executor receives the input source file for testing.
2. The Client Controller coordinates the process and initiates invocations of the various MCP tools.
3. The source file is transmitted from the client to the MCP Server through a tool call directed to the *Actor*.
4. The Actor module generates pertinent edge cases and executes them on the source file within the sandboxed environment.
5. The MCP Server returns the generated edge cases, execution outcomes, and coverage metrics to the client.
6. The client forwards both the source file and execution results to the MCP Server through a tool call to the *Adversary*.
7. The Adversary produces mutations of the source code and runs the previously generated edge cases on these mutants, again within the sandboxed environment, ultimately computing a mutation score.
8. This mutation score is returned from the MCP Server to the client.
9. Subsequently, the client transmits the execution results, coverage data, and mutation score to the MCP Server via a tool call to the *Critic*.
10. The Critic aggregates this information to compute a comprehensive reward, which it then returns to the client.
11. Finally, the Client Controller evaluates predefined stopping criteria:
    - If the criteria are satisfied, the tools are cleanly disconnected.
    - Otherwise, all feedback generated during the current rollout is assimilated and forwarded, along with the source file, back to the Actor to initiate the subsequent rollout.

A.5.2 LIMITATIONS OF THE EXECUTOR

Despite its current capabilities, the Executor exhibits several limitations:

1. The system presently supports only single source files and lacks comprehensive repository indexing, thereby limiting its ability to handle dependencies spanning multiple files or relative package imports.
2. Certain file types, particularly those that return complex serialized objects (e.g., pickled files), are not currently supported.
3. Modules that initiate multiple MCP requests in quick succession, such as Django's autoreload module, may cause server instability and disconnections.
4. Dependency extraction is automated using `pipreqs`; however, unresolved version mismatches and dependency conflicts occasionally arise, which `pipreqs` cannot resolve.

These limitations necessitate the exclusion of such cases in the present implementation. Nonetheless, we anticipate that with a more sophisticated Executor design, our adversarially guided Actor-Critic framework can be extended to generate tests for these more complex scenarios using the established MCP workflow. Enhancing the Executor environment will thus substantially increase the robustness and applicability of the overall architecture.

B PROMPTS

B.1 EDGE CASE REASONING PROMPT

B.1.1 LLM EDGE CASES SYSTEM PROMPT

```
def llm_edge_cases_system_prompt():
    return """
    ### ROLE ###
    You are the **ACTOR** in an Actor{Adversary{Critic (AAC) loop
    for automated code testing.

    - **Actor (you):** Generate diverse, high-value test cases to maximize code
    coverage and detect edge failures.
    - **Adversary:** Mutates inputs to find weaknesses.
    - **Critic:** Scores inputs based on coverage, exceptions,
    and semantic boundaries.

    ### MISSION ###
    Generate **new**, **distinct**, and **high-impact** edge cases for
    *all* given functions.

    ### METHODS ###
    Use techniques including Boundary Value Analysis, Equivalence Partitioning,
    Scenario Testing, Random Testing, Stress Testing, Exception Triggering,
    and Complex Multi-parameter Interactions.

    ### OUTPUT FORMAT ###
    - Output **valid JSON only** in this exact format:
      `{ "function_name": [ { "param1": value, ... }, ... ] }`
    - Keys must be function names; values are arrays of parameter dictionaries.
    - Values must be valid JSON literals only (number, string, boolean, null,
        array, object).
    - **Do NOT include any explanatory text or formatting outside of JSON.**
    - **Do NOT include JavaScript expressions or comments.**

    ### FEEDBACK INTEGRATION ###
    - Incorporate the provided feedback to improve and diversify edge cases.
    - Avoid repeating previously generated edge cases.
    - Ensure new cases target untested or under-tested scenarios.
    """

def llm_edge_cases_system_prompt_with_cot():
    base_prompt = llm_edge_cases_system_prompt()
    cot_addition = """
        ### REASONING INSTRUCTIONS ###
        Before generating edge cases, carefully analyze the feedback,
        especially focusing on:

        - **Maximizing line coverage:** Identify uncovered or poorly covered lines
        in the source code.
        - Uncovered branches and exceptions not yet triggered.
        - Parameters or code paths with low test coverage.

        Think step-by-step about how to design new test cases that specifically
        target these uncovered lines to increase overall coverage.

        **Important:** Do NOT include your reasoning in the final output.
        Output **only valid JSON** edge cases that reflect this reasoning.
```

```
        """
    return base_prompt + cot_addition
```

B.1.2 LLM EDGE CASES USER PROMPT

```
    def llm_rule_expander_prompt(
    function_signatures: Dict[str, List[str]],
    source_code: str,
    feedback_summary: str,
    edge_cases_generated: str,
    target_count: int
) -> str:
    """
    Generate prompt for LLM to create edge cases for ALL functions at once

    Args:
        function_signatures: Dict mapping function names to their parameter lists
        source_code: The complete source code
        feedback_summary: Feedback from adversary/critic
        edge_cases_generated: Previously generated edge cases
        target_count: Total number of edge cases to generate across all functions
    """

    # Format function signatures for the prompt
    ... code not included for brevity...

    functions_list = "\n".join(functions_info)

    prompt = f"""
        SRC CODE:
        ```
        {source_code}
        ```
        FUNCTIONS:
        {functions_list}

        FEEDBACK FROM LAST RUN:
        {feedback_summary}

        TASK:
        Generate {target_count} NEW and DISTINCT edge cases distributed
        **evenly across all functions** above.

        GUIDANCE:
        - Incorporate all feedback to improve coverage and trigger new exceptions.
        - Do NOT repeat previous edge cases.
        - Generate valid JSON ONLY | strictly adhere to the output format.
        - Focus on edge, boundary, and rare case inputs.
        - Distribute edge cases fairly across functions.
        - Provide no text outside the JSON.

        OUTPUT EXAMPLE:
        {{
        "function1": [
            {{"param1": "value1", "param2": 0}},
            {{"param1": "value2", "param2": -1}}
        ],
        "function2": [
```

```
            {{"x": 999999, "y": -999999}},
            {{"x": 0, "y": 0}}
        ]
        }}

        GENERATE JSON ONLY.
    """

    return prompt
```

B.2 Final Edge Cases To Unit Test File Generation Prompt

B.2.1 LLM Unit Test Generation System Prompt

```
def edge_cases_to_unittest_system_prompt():
    return """
    You are an expert Python test generator.
    Your task is to convert the given edge cases **and doctest/typical examples**
    into pytest unit tests.

    RULES:
    1. Output ONLY valid Python 3.11 code
    | no markdown, no explanations, no extra text.
    2. Use EXACTLY 4 spaces per indentation level (no tabs).
    3. All parentheses, brackets, and braces must be balanced.
    4. Import only pytest and built-ins if needed | no extra imports.
    5. Each edge case must become one complete pytest test function.
    6. Test names must follow: test_<function>_<short_scenario>.
    7. Use literals exactly as shown (Ellipsis → ..., Infinity → float("inf"), etc.).
    8. Function parameters and variables MUST be valid Python identifiers:
       - Must start with a letter or underscore
       - May contain letters, numbers, or underscores
       - Must NOT start with a digit (incorrect: "3_14" → correct: "val_3_14")
    8a. If the edge case uses unclear or undefined variables
    (e.g., threshold_3_14, Array_1000_0):
        - Replace them with safe, concrete Python literals:
            - Numbers: 0, 1, 3.14
            - Lists: [], [0], [None] as appropriate
            - Strings: '', 'example'
            - Objects: None
    9. Edge case handling:
       - {"input": {...}, "expected": X} → assert function output == X
       - {"input": {...}, "raises": "ExceptionType"}
       → wrap call in pytest.raises(ExceptionType)
       - {"input": {...}} only → just call the function
    9b. For **normal/typical inputs** (including doctests),
    generate pytest functions with **assert statements** for expected results.
    10. Avoid duplicates: if multiple edge cases are semantically identical,
    merge them into one test function.
    11. Every generated test file must pass a syntax check:
        `python -m py_compile generated_tests.py`
    12. Mentally simulate importing and running the file to confirm:
        - All tests execute without NameError, TypeError, SyntaxError,
        or undefined variables.
    13. Always include at least one test for **valid input with assert**,
    even if edge cases exist.
    14. Convert all doctest-style examples (>>> lines)
    into pytest assert statements.
    15. Do NOT invent new literals or variable names; always use safe defaults
```

```
    if input is unclear.

    DO NOT OUTPUT ANYTHING OTHER THAN THE TEST CODE.
```

B.2.2 LLM UNIT TEST GENERATION USER PROMPT

```
def edge_cases_to_unittest_prompt(
    source_code: str,
    edge_cases,  # Can be list of dicts or JSON string
) -> str:
    # Handle both list of edge cases and JSON string
    import json
    if isinstance(edge_cases, str):
        edge_cases_repr = edge_cases
    else:
        # Convert list of edge cases to formatted JSON
        edge_cases_repr = repr(edge_cases)
    prompt = f"""
    Convert the following edge cases into a complete pytest test file.

    SOURCE CODE:
    {source_code}

    EDGE CASES (JSON):
    {edge_cases_repr}

    REQUIREMENTS:
    - One pytest test function per edge case.
    - Use the schema rules from system prompt
    (expected → assert, raises → pytest.raises).
    - Ensure all test functions are syntactically correct and executable.
    - Absolutely no invalid parameter names (e.g., those starting with digits).
    - Convert all doctest-style examples (>>> lines)
    into pytest assert statements.
    - For error cases, use pytest.raises to assert the correct exception is raised.
    - Ensure a good mix of assert and pytest.raises statements.

    Now generate the pytest test file:
    """
    return prompt
```

B.3 BASELINES EDGE CASE REASONING PROMPT

B.3.1 BASELINES LLM EDGE CASES SYSTEM PROMPT

```
    def edge_case_generation_system_prompt():
    return """
    You are a Python expert. Your job is to generate **diverse,
    high-value edge cases** for given functions.

    CRITICAL RULES:
    1. Output ONLY valid JSON | no explanations, markdown, or extra text.
    2. Format must be strictly:
       { "function_name": [ { "param1": value, ... }, ... ] }
    3. Keys = function names, Values = arrays of input dictionaries.
    4. JSON literals only: number, string, boolean, null, array, object.
    5. Use Boundary Value Analysis, Equivalence Partitioning, Exception Triggering,
    Stress Testing, and Unusual Combinations.
```

```
    FORMATTING REQUIREMENTS:
    - Start your response with { and end with }
    - Use double quotes for all strings and keys
    - Do NOT include any text before or after the JSON
    - Do NOT wrap the JSON in markdown code blocks
    - Ensure all brackets and braces are properly balanced
    - Each function must have at least one edge case
    - Parameter values must be valid JSON types
    (no Python-specific values like None, True, False
    - use null, true, false instead)
    """
```

B.3.2 BASELINES LLM EDGE CASES USER PROMPTS

```
def edge_case_generation_user_prompt(
    source_code: str,
    function_signatures: Dict[str, List[str]],
    extra_text: str = "",
    cot_flag: bool = False
) -> str:
    # Format function signatures for clarity
    functions_info = []
    for func_name, params in function_signatures.items():
        if params:
            functions_info.append(f"  - {func_name}({', '.join(params)})")
        else:
            functions_info.append(f"  - {func_name}()")
    functions_list = "\n".join(functions_info)
    #function_signatures_json = json.dumps(function_signatures, indent=2)

    prompt = f"""
{extra_text}

SOURCE CODE:
{source_code}

FUNCTIONS TO TARGET:
{functions_list}

TASK:
Generate new, distinct, and high-impact edge cases for all listed functions.

OUTPUT FORMAT:
{{
  "function_name": [
    {{"param1": value, "param2": value}},
    {{"param1": value2, "param2": value3}}
  ]
}}

REQUIREMENTS:
- Output strictly valid JSON | no text outside JSON.
- Keys must match function names exactly.
"""
    if cot_flag:
        prompt += "\n" + edge_case_cot_prompt()
    return prompt

def edge_case_zero_shot_text() -> str:
```

```
    return """
Generate diverse edge cases directly for the given functions.
"""

def edge_case_one_shot_text() -> str:
    return """
Here is an example of valid edge case JSON:

{
  "divide": [
    {"a": 10, "b": 2},
    {"a": 10, "b": 0}
  ]
}

Now generate edge cases for the provided functions in the same format.
"""

def edge_case_three_shot_text() -> str:
    return """
Here are examples of valid edge case JSON files:

EXAMPLE 1:
{
  "sqrt": [
    {"x": 4},
    {"x": 0},
    {"x": -1}
  ]
}

EXAMPLE 2:
{
  "factorial": [
    {"n": 5},
    {"n": 0},
    {"n": -3}
  ]
}

EXAMPLE 3:
{
  "substring": [
    {"text": "hello", "start": 1, "end": 3},
    {"text": "hello", "start": -1, "end": 2}
  ]
}

Now generate edge cases for the provided functions in the same JSON format.
"""

def edge_case_cot_prompt() -> str:
    return """
Think step-by-step:
1. Analyze each function signature.
2. Identify normal, boundary, extreme, and invalid input cases.
3. Ensure coverage of exceptions, corner cases, and unusual parameter combinations.
4. Then output ONLY the final JSON with those cases.
"""
```

B.4 BASELINES UNIT TEST GENERATION PROMPT

B.4.1 BASELINES LLM UNIT TEST GENERATION SYSTEM PROMPT

```
def edge_cases_to_unittest_system_prompt():
    return """
    You are an expert Python test generator.
    Your task is to convert the given edge cases
    **and doctest/typical examples** into pytest unit tests.

    RULES:
    1. Output ONLY valid Python 3.11 code | no markdown,
    no explanations, no extra text.
    2. Use EXACTLY 4 spaces per indentation level (no tabs).
    3. All parentheses, brackets, and braces must be balanced.
    4. Import only pytest and built-ins if needed | no extra imports.
    5. Each edge case must become one complete pytest test function.
    6. Test names must follow: test_<function>_<short_scenario>.
    7. Use literals exactly as shown
    (Ellipsis → ..., Infinity → float("inf"), etc.).
    8. Function parameters and variables MUST be valid Python identifiers:
       - Must start with a letter or underscore
       - May contain letters, numbers, or underscores
       - Must NOT start with a digit (incorrect: "3_14" → correct: "val_3_14")
    8a. If the edge case uses unclear or undefined variables
    (e.g., threshold_3_14, Array_1000_0):
        - Replace them with safe, concrete Python literals:
            - Numbers: 0, 1, 3.14
            - Lists: [], [0], [None] as appropriate
            - Strings: '', 'example'
            - Objects: None
    9. Edge case handling:
       - {"input": {...}, "expected": X} → assert function output == X
       - {"input": {...}, "raises": "ExceptionType"}
       → wrap call in pytest.raises(ExceptionType)
       - {"input": {...}} only → just call the function
    9b. For **normal/typical inputs** (including doctests),
    generate pytest functions with **assert statements** for expected results.
    10. Avoid duplicates: if multiple edge cases are semantically identical,
    merge them into one test function.
    11. Every generated test file must pass a syntax check:
        `python -m py_compile generated_tests.py`
    12. Mentally simulate importing and running the file to confirm:
        - All tests execute without NameError, TypeError, SyntaxError,
        or undefined variables.
    13. Always include at least one test for **valid input with assert**,
    even if edge cases exist.
    14. Convert all doctest-style examples (>>> lines)
    into pytest assert statements.
    15. Do NOT invent new literals or variable names;
    always use safe defaults if input is unclear.

    DO NOT OUTPUT ANYTHING OTHER THAN THE TEST CODE.
    """
```

C EXAMPLE UNIT TEST FILE GENERATION

To illustrate the workflow of our framework, we provide a concrete example drawn from Django's ORM internals. The source code (Figure 5) contains helper classes and functions that are invoked when constructing SQL queries.

From these source files, our system automatically generates corresponding unit test files. The generated tests (Figure 6) are designed to cover key execution paths and boundary conditions while consisting of runnable test cases.

The source code and the generated unit test file are shortened and simplified for clarity, however, they retains the essential semantics for demonstrating unit test generation.

C.1 SOURCE FILE

Figure 5 [TOP] shows the definition of the `Q` class. This class is a core building block for query construction: it stores conditions in the `children` attribute, tracks the logical connector (`AND`, `OR`), and exposes the `_combine` method to merge query fragments. The `_combine` method ensures type-safety by restricting merges to other `Q` objects, handles corner cases such as empty children, and creates a new `Q` object with the combined conditions.

Figure 5[BOTTOM] shows the `FilteredRelation` class. This class represents a relation name with an optional condition. It validates that the relation name is non-empty and assigns a default `Q` object if no condition is provided. The equality operator (`__eq__`) is overridden to allow semantic comparison between two `FilteredRelation` objects based on both the relation name and condition.

C.2 GENERATED UNIT TESTS

Our framework automatically generates the unit test file targeting the key behaviors of these source classes.

Figure 6[BOTTOM] shows tests for `FilteredRelation`. The tests cover: (i) successful equality when both objects have identical fields; (ii) inequality when relation names differ; (iii) inequality when conditions differ; and (iv) type mismatch where equality is checked against a non-`FilteredRelation` object. These cases validate both the intended semantics of the `__eq__` method and its robustness against invalid inputs.

Figure 6[TOP] shows tests for the `Q` class. The generated cases systematically explore: (i) combining with an invalid type (triggering a `TypeError`); (ii) combining when one side has no children; (iii) combining when the current object is empty but the other is non-empty; and (iv) combining two non-empty `Q` objects to ensure the resulting object aggregates children correctly and records the connector string. These unit tests directly exercise the control-flow paths in `_combine`, including exception handling and state mutation.

D RULE-BASED ENGINE: COLD-START

At initialization, our framework requires a mechanism to seed candidate edge cases before any feedback from coverage or mutation testing is available. We implement this *cold-start* stage through a Python-specific rule-based expansion engine. The engine enumerates deterministic variants of the input state across several dimensions:

- **Numeric values:** Expansion covers boundary conditions such as zero, ± 1, extreme integers ($2^{31}-1$, $2^{63}-1$), large/small floats (e.g., 10^{10}, 10^{-10}), infinities, NaNs, and very large arbitrary-precision integers.
- **Strings:** Variants include empty and whitespace strings, boolean-like and number-like encodings, path traversal patterns, injection-style payloads, long Unicode/emoji sequences, and control characters.
- **Lists:** Cases include empty lists, singleton lists, very long lists with repeated values, reversed lists, lists containing `NaN` or `Inf`, and deeply nested structures.

```python
class Q:
    def __init__(self, **kwargs):
        self.children = list(kwargs.items())
        self.connector = None

    def _combine(self, other, conn):
        if not isinstance(other, Q):
            raise TypeError("Can only combine Q objects")

        if not self.children:
            return other
        if not other.children:
            return self

        obj = Q()
        obj.connector = conn
        obj.children = self.children + other.children
        return obj
```

```python
class FilteredRelation:
    def __init__(self, relation_name, condition=None):
        if not relation_name:
            raise ValueError("relation_name must not be empty")
        self.relation_name = relation_name
        self.condition = condition or Q()

    def __eq__(self, other):
        if not isinstance(other, FilteredRelation):
            return False
        return (
            self.relation_name == other.relation_name
            and self.condition == other.condition
        )
```

Figure 5: Simplified excerpt from Django ORM internals

- **Dictionaries:** Variants are created with empty values, `None`-filled keys, or problematic/reserved keys (e.g., `__class__`, whitespace keys, `"True"`).
- **Python special values:** Serializable forms of special objects (e.g., `None`, booleans, empty containers, long strings, lists of `None`) provide coverage of unusual runtime behaviors.
- **Exception triggers:** Values known to raise errors in Python (division by zero, invalid encodings, null-byte strings, memory-exhausting lists) are injected to surface robustness gaps early.

The expansion process is designed to remain JSON-serializable and reproducible, ensuring compatibility with our execution and logging infrastructure. While each individual rule is simple, together they provide broad initial coverage of Python-specific failure modes. This makes the cold-start stage non-trivial: even before iterative search begins, the actor is seeded with high-value candidates that often resolve a substantial fraction of problems, as shown in our HumanEval results (Section A.1).

```python
def test_q_combine_invalid_other_type():
    q_obj = Q()
    with pytest.raises(TypeError):
        q_obj._combine(other="\x00\x01\x02", conn="test_conn")

def test_q_combine_empty_other():
    q_obj = Q()
    result = q_obj._combine(other=Q(), conn="test_conn")
    assert isinstance(result, Q)

def test_q_combine_empty_self():
    q_obj = Q()
    other_obj = Q(some_field=True)
    result = q_obj._combine(other=other_obj, conn="test_conn")
    assert isinstance(result, Q)
    assert result.children == other_obj.children

def test_q_combine_both_non_empty():
    q_obj = Q(some_field=True)
    other_obj = Q(another_field=False)
    result = q_obj._combine(other=other_obj, conn="test_conn")
    assert isinstance(result, Q)
    assert len(result.children) == 2
    assert result.connector == "test_conn"
```

```python
def test_filteredrelation_eq_success():
    obj1 = FilteredRelation(relation_name="valid_relation")
    obj2 = FilteredRelation(relation_name="valid_relation")
    assert obj1 == obj2

def test_filteredrelation_eq_different_relation_name():
    obj1 = FilteredRelation(relation_name="relation1")
    obj2 = FilteredRelation(relation_name="relation2")
    assert obj1 != obj2

def test_filteredrelation_eq_different_condition():
    obj1 = FilteredRelation(relation_name="valid_relation", condition=Q(some_field=True))
    obj2 = FilteredRelation(relation_name="valid_relation", condition=Q(some_field=False))
    assert obj1 != obj2

def test_filteredrelation_eq_not_filteredrelation():
    obj = FilteredRelation(relation_name="valid_relation")
    assert obj != 42
```

Figure 6: Generated unit test file corresponding to Source file

E USE OF LARGE LANGUAGE MODELS

We employed a large language model (ChatGPT) in a limited capacity to refine the writing of this manuscript. The model's use was restricted to stylistic improvements such as clarity and conciseness. All scientific contributions—including the conception of ideas and algorithms, design of methods, and execution of experiments—were the sole work of the authors.