Agentic Test Suite Refinement: Repository Characteristics Dominate Prompt Strategy Effects

Rye Howard-Stone
rye.howard-stone@uconn.edu
University of Connecticut
Department of Computer Science and Engineering
Storrs, Connecticut, USA

ABSTRACT

Large Language Model (LLM) agents can execute software analysis tools to improve test suites, but optimal prompting strategies remain unclear. We present two complementary tools: **aTSR** (Agentic Test Suite Refinement), a Claude Code slash command orchestrating coverage analysis and mutation testing for backend tests, and **testScout**, a visual end-to-end testing framework using Set-of-Marks targeting to catch integration bugs that unit tests miss.

In an exploratory study (n=1 per configuration) across nine Python repositories, we compared six prompting strategy variants. Our statistically-supported findings include: (1) the +Mutations variant *catastrophically failed* in 5/9 repositories, achieving ≤10% coverage (sign test, p=0.002); (2) excluding these failures, repository characteristics explained 25.5× more variance in coverage than prompt strategy choice; and (3) a simple baseline prompt matched elaborate multi-phase workflows in most configurations.

Through the JobsCoach case study, we demonstrate that high backend coverage can still miss critical integration bugs—testScout detected 6 production bugs including a P0 routing failure that unit tests could not catch. Both tools are open-source. *Caveats: (1) Single-run results require replication. (2) Session logs reveal generated tests rely heavily on mocking, raising quality concerns beyond coverage metrics.*

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → *Natural language processing*.

KEYWORDS

LLM, test generation, agentic coding, prompt engineering, mutation testing, code coverage

ACM Reference Format:
Rye Howard-Stone. 2025. Agentic Test Suite Refinement: Repository Characteristics Dominate Prompt Strategy Effects. In *Proceedings of AI for Software Engineering (CSE 5095).* ACM, New York, NY, USA, Article , 10 pages.

1 INTRODUCTION

The emergence of agentic Large Language Model (LLM) systems has transformed software engineering workflows [9]. Tools such as Claude Code [1], GitHub Copilot, and Cursor enable developers to delegate complex coding tasks to AI agents that can read files, execute commands, and iteratively refine their outputs. Empirical studies show these tools can improve developer productivity by up to 55% [13], and benchmarks like SWE-bench [7] demonstrate that frontier models can now resolve real-world GitHub issues autonomously. A fundamental question emerges: *how much guidance should we provide these agents, and does more elaborate prompting actually improve outcomes?*

Software engineers have developed sophisticated tools for analyzing and improving code quality:

- **Code coverage tools** (coverage.py, gcov) identify which lines are executed by tests
- **Mutation testers** (mutmut, Stryker) verify whether tests actually detect code changes
- **Static analyzers** (pyright, Infer) find potential bugs without execution
- **AST parsers** (Tree-sitter) enable structural code analysis

Modern agentic LLMs can leverage these tools, but typically only when directed. This raises our central research question: **Does providing detailed methodological guidance—incorporating coverage analysis, mutation testing, and expert heuristics—improve LLM-based test generation, or does it introduce unnecessary overhead and potential failure modes?**

We hypothesized that structured workflows would outperform simple prompts. Our exploratory experiments reveal a more nuanced picture: **repository characteristics appeared to dominate prompt strategy effects by a factor of 25.5×** in variance decomposition analysis (excluding catastrophic failures). Critically, the +Mutations variant—which incorporates mutation testing feedback—catastrophically failed in 5/9 repositories, achieving ≤10% coverage (sign test, p=0.002). Token efficiency patterns were mixed with no statistically significant advantage for minimal prompts.

However, even achieving high backend coverage left critical bugs undetected. During development of JobsCoach (a job search application), we encountered 6 production bugs—including a P0 routing failure—despite strong test coverage. This motivated our second contribution: testScout, which provides visual E2E testing to catch integration bugs that unit tests structurally cannot detect.

This work was motivated by practical experience: the first author developed AmpliconHunter [5], a bioinformatics tool for PCR amplicon prediction, using AI-assisted test-driven development. Both the need for principled prompting strategies and the gap between coverage metrics and actual correctness emerged from this experience.

1.1 Contributions

This paper contributes:

CSE 5095, Fall 2025, University of Connecticut
2025. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM

(1) **aTSR**: An open-source Claude Code slash command (`/refine-tests`) implementing multi-phase test refinement with coverage analysis and mutation testing
(2) **testScout**: A visual end-to-end testing framework using Set-of-Marks targeting for robust element identification, complementing backend tests by detecting integration bugs
(3) **Experimental protocol**: A two-phase evaluation framework with six factorial prompt variants across nine repositories
(4) **Statistical analysis**: Variance decomposition and hypothesis testing on n=1 exploratory data

1.2 Key Findings

Our exploratory study (n=1 per configuration) yielded statistically-supported findings:

- **Mutation testing can catastrophically fail**: V4 (+Mutations) achieved ≤10% coverage in 5/9 repos (p=0.002)
- **Repository variance >> prompt variance**: Repository characteristics explained 25.5× more variance than prompt strategy choice
- **Simple baselines are competitive**: A minimal prompt matched elaborate workflows in most configurations
- **Coverage ≠ correctness**: JobsCoach case study shows 6 bugs escaped high-coverage backend tests

While based on single runs per configuration, these patterns—particularly the mutation testing failure and variance decomposition—are statistically robust and merit attention.

Table 1: Summary of Statistical Findings

Finding	Statistic	p-value	Interpretation
V4 < V1 coverage	9/9 repos	0.002	Mutation testing *may hurt**
Repo variance	25.5×	—	Repo >> prompt effect
V6 token efficiency	2/6 repos	0.89	*Not* significant
Bio vs general	60% vs 83%	0.19	*Not* significant

*Requires replication; pattern is statistically robust but effect magnitude may vary.

Practitioner Takeaways

1. Start simple. A 15-line prompt achieved the same coverage as 80-line multi-phase workflows. Complexity doesn't guarantee improvement.

2. Avoid mutation testing for exploration. V4 failed catastrophically in 56% of repos (p=0.002). Use mutation testing only after achieving baseline coverage.

3. Know your codebase. Repository characteristics explained 87% of variance—25× more than prompt choice. Adapt strategy to codebase, not vice versa.

4. E2E tests catch what unit tests miss. JobsCoach's P0 routing bug was undetectable by unit tests despite high coverage.

5. Inspect test quality, not just coverage. Generated tests relied heavily on mocking. High coverage with mocks may not catch real bugs—manual review remains essential.

2 RELATED WORK

Automated Test Generation. Traditional approaches include EvoSuite [4] (genetic algorithms) and Randoop [12] (feedback-directed random testing). Both achieve high coverage but produce tests that may be difficult to maintain.

LLM-Based Testing. TestPilot [15] achieves 93% coverage with Codex; CodaMosa [8] combines search-based testing with LLMs; ChatUniTest [3] and CoverUp [14] use iterative refinement. These works find LLMs generate useful tests but struggle with assertions. Our work examines prompt complexity in agentic settings.

Prompt Engineering. Chain-of-thought [18] and self-consistency [17] improve reasoning, but optimal strategies for agentic coding remain understudied. Ma et al. [10] found LLMs lack semantic consistency; Copilot studies [16] show generated code requires review.

Mutation Testing. Mutation testing [6] evaluates test quality by checking if tests detect code changes. We found mutation feedback can degrade agent performance—an unexpected result.

3 METHODOLOGY

We developed two tools for AI-assisted software testing and conducted a systematic two-phase evaluation.

3.1 aTSR: Agentic Test Suite Refinement

aTSR is a Claude Code slash command (`/refine-tests`) with three modes: `auto` (adaptive), `quick` (lightweight), and `full` (comprehensive). The full workflow: (1) environment detection, (2) coverage analysis, (3) mutation testing, (4) test recommendations, (5) property testing, (6) verification. Heuristics include: mutate only covered code, prioritize boundary conditions, match effort to codebase size.

3.2 testScout: Visual AI Testing

testScout uses Set-of-Marks (SoM) [11] for reliable element targeting. It injects JavaScript to tag interactive elements with `data-testscout-id` attributes, overlays numbered markers, and sends marked screenshots to a vision model (Gemini/GPT-4V). The model returns element IDs rather than fragile CSS selectors. An `Explorer` class enables autonomous bug-hunting without pre-written tests. Figure 1 shows the SoM overlay identifying interactive elements.

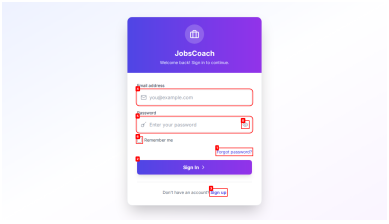

Figure 1: testScout's Set-of-Marks overlay on the JobsCoach login page. Red numbered boxes identify interactive elements (1: Forgot password, 2: Sign In button, 3: Sign up link, 4-6: form fields). The vision model receives this marked screenshot and returns element IDs rather than fragile CSS selectors.

3.3 Experimental Design

Our evaluation proceeded in two phases.

3.3.1 Phase 1: Base vs. Refine. Phase 1 compared **Base** (simple "improve test coverage" prompt) vs. **Refine** (full `/refine-tests` workflow), using Claude Sonnet 4.5 and Opus 4.1 on three "reduced coverage" repositories (click, mistune, schedule) where tests were artificially removed.

3.3.2 Phase 2: Six Prompt Variants. Based on Phase 1 findings (base outperformed refine), Phase 2 explored whether specific components of the refine workflow might help. We developed six prompt variants using factorial experimental design (Table 2).

Table 2: Prompt Variant Design Matrix

Variant	Lines	Context	Cov	Mut	Description
V1	30	–	–	–	Control (original base)
V2	40	✓	–	–	+Context only
V3	60	✓	✓	–	+Coverage tool
V4	60	✓	–	✓	+Mutation tool
V5	80	✓	✓	✓	+Both tools
V6	15	✓	–	–	Minimal directive

The "Context" column indicates pre-execution commands detecting language, project structure, and test locations. V6 (Minimal) reduces the directive to 15 lines: "Improve test coverage to 80%+. Measure, identify gaps, write targeted tests. Verify."

Phase 2 tested across three categories: **Reduced Coverage** (click, mistune, schedule—tests artificially removed); **Low Existing Coverage** (python-box, colorama, boltons—naturally low coverage); and **Bioinformatics** (dnaapler, fastqe, pyfaidx—domain-specific).

3.3.3 Metrics. We measured: **Final Coverage** (line %), **Pass Rate** (% tests passing), **Tests Added**, **Tokens** (input + output + cache), **Coverage Efficiency** (pts/M tokens), **Runtime**, and **Tool Calls**.

3.3.4 Experimental Controls and Limitations. We used Claude Sonnet 4.5 and Opus 4.1 without fixing random seeds. For "reduced coverage" repos, tests were removed to reduce coverage from >90% to 60–70%. Each configuration ran exactly once (n=1), limiting statistical validity—observed differences may reflect LLM variance.

4 EVALUATION RESULTS

4.1 Phase 1: Base Outperforms Refine

In our Phase 1 trials, the simple base strategy matched or exceeded the elaborate refine workflow in most configurations. Table 3 summarizes the observed results.

Observations from Phase 1 trials:

- Base matched or exceeded refine coverage in 10 of 12 configurations
- Base showed higher pass rates, often reaching 100%
- Sonnet performed comparably to Opus at lower cost
- Refine used 40–70% more tokens without apparent coverage benefit

Table 3: Phase 1: Base vs. Refine on Reduced Coverage Repos

Repo	Model	Strategy	Cov	Pass%	Tokens
schedule	Sonnet	base	88%	100.0	2.9M
schedule	Sonnet	refine	85%	72.5	4.9M
schedule	Opus	base	91%	100.0	3.2M
schedule	Opus	refine	90%	96.8	6.0M
mistune	Sonnet	base	79%	94.6	6.0M
mistune	Sonnet	refine	72%	85.0	5.6M
mistune	Opus	base	76%	94.3	4.2M
mistune	Opus	refine	71%	97.7	7.9M
click	Sonnet	base	64%	100.0	2.4M
click	Sonnet	refine	64%	91.4	8.2M
click	Opus	base	67%	91.9	10.6M*
click	Opus	refine	66%	95.5	3.3M

*Extended exploration cycles due to integrating with 190 existing baseline tests.

Note: With n=1, these patterns may reflect run-to-run variance rather than true strategy differences.

Figure 2 provides a comprehensive visualization of these results, including coverage achievement, pass rates, token efficiency, and a strategy performance radar chart.

4.2 Phase 2: Reduced Coverage Repositories

For the first Phase 2 experiment, we artificially removed some tests from three popular Python libraries to create coverage gaps, then evaluated all six prompt variants. Figure 3 shows the complete results.

click (CLI toolkit): All variants reached 60-80% final coverage. V1 (Baseline) generated the most tests (580) but used 11M tokens. V6 (Minimal) achieved similar coverage with 5M tokens.

mistune (Markdown parser): Coverage reached 55-75% across variants. V4 (+Mutations) showed notably high token-per-test cost (4000K tokens per new test), while other variants averaged 500K.

schedule (job scheduler): Highest coverage achieved (75-90%). V6 showed the best coverage efficiency at 25% increase per million tokens, compared to 10-15% for complex variants.

4.3 Phase 2: Low Existing Coverage Repositories

For the second Phase 2 experiment, we selected three Python libraries with naturally low existing test coverage (no artificial removal). Figure 4 shows the complete results.

python-box (dictionary wrapper): Started with near-zero coverage (2%), all variants achieved dramatic improvement to 80–90%. This was the most successful case, with V6 achieving 83% coverage using 3.0M tokens—an efficiency of 27 percentage points per million tokens.

colorama (terminal colors): Started with existing coverage around 70% (dashed line). All variants achieved only modest improvement to 75-80%, suggesting a ceiling effect. Token efficiency was lower due to smaller coverage gains.

boltons (utility library): Similar to colorama, started with 70% baseline coverage. Final coverage reached 70-75% across variants.

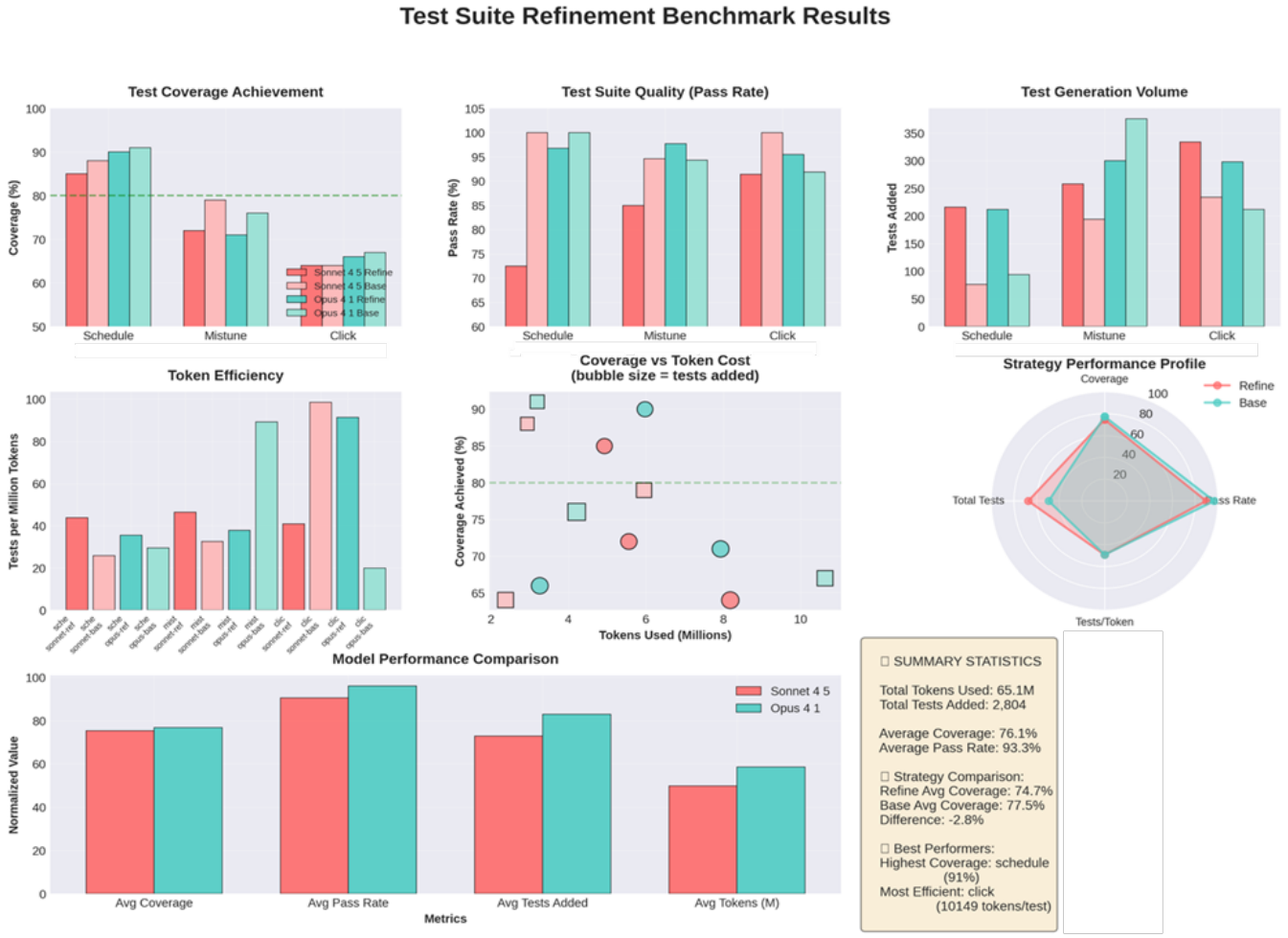

Figure 2: Phase 1 benchmark results comparing Base vs. Refine strategies across Sonnet and Opus models on three reduced-coverage repositories (schedule, mistune, click). Top row: coverage achievement, pass rates, test generation volume. Middle row: token efficiency, coverage vs. token cost scatter plot, radar chart of strategy profiles. Bottom row: model performance comparison. Summary statistics show Base achieved 77.5% average coverage vs. Refine's 74.7%, while using fewer tokens.

Generated the most tests (400-500) but with modest coverage improvement, indicating diminishing returns at higher baseline coverage.

4.4 Phase 2: Bioinformatics Domain

For the third Phase 2 experiment, we evaluated domain-specific bioinformatics repositories to test generalization beyond general-purpose Python code. Figure 5 shows the complete results.

dnaapler (DNA reorientation): Started with 25% baseline coverage, improved to 40-45% across most variants. V5 (+Both) generated the most tests (200) but also had the highest failure rate (gray bar). Modest positive coverage efficiency.

fastqe (FASTQ visualization): Started with 45% baseline coverage. Most variants achieved minimal improvement or even slight decrease. Critically, V4 (+Mutations) showed *negative* coverage efficiency—the agent's intervention actually reduced coverage. V3 (+Coverage) performed best with 20% efficiency gain.

pyfaidx (FASTA indexing): Started with 70% baseline coverage. Results were mixed—some variants slightly improved coverage while others decreased it. High token costs (120-160K tokens per passing test for some variants) with minimal coverage benefit.

Key Finding: Unlike general-purpose repositories where all variants improved coverage, bioinformatics repos showed that elaborate prompting (especially V4/V5 with mutation feedback) can actually *harm* performance when the model lacks domain expertise.

4.5 Tool Usage Patterns

Analysis of tool calls revealed interesting patterns:

- **V4/V5 used more Bash commands** (70-80 per run) than V6 (40-50), primarily for running mutmut

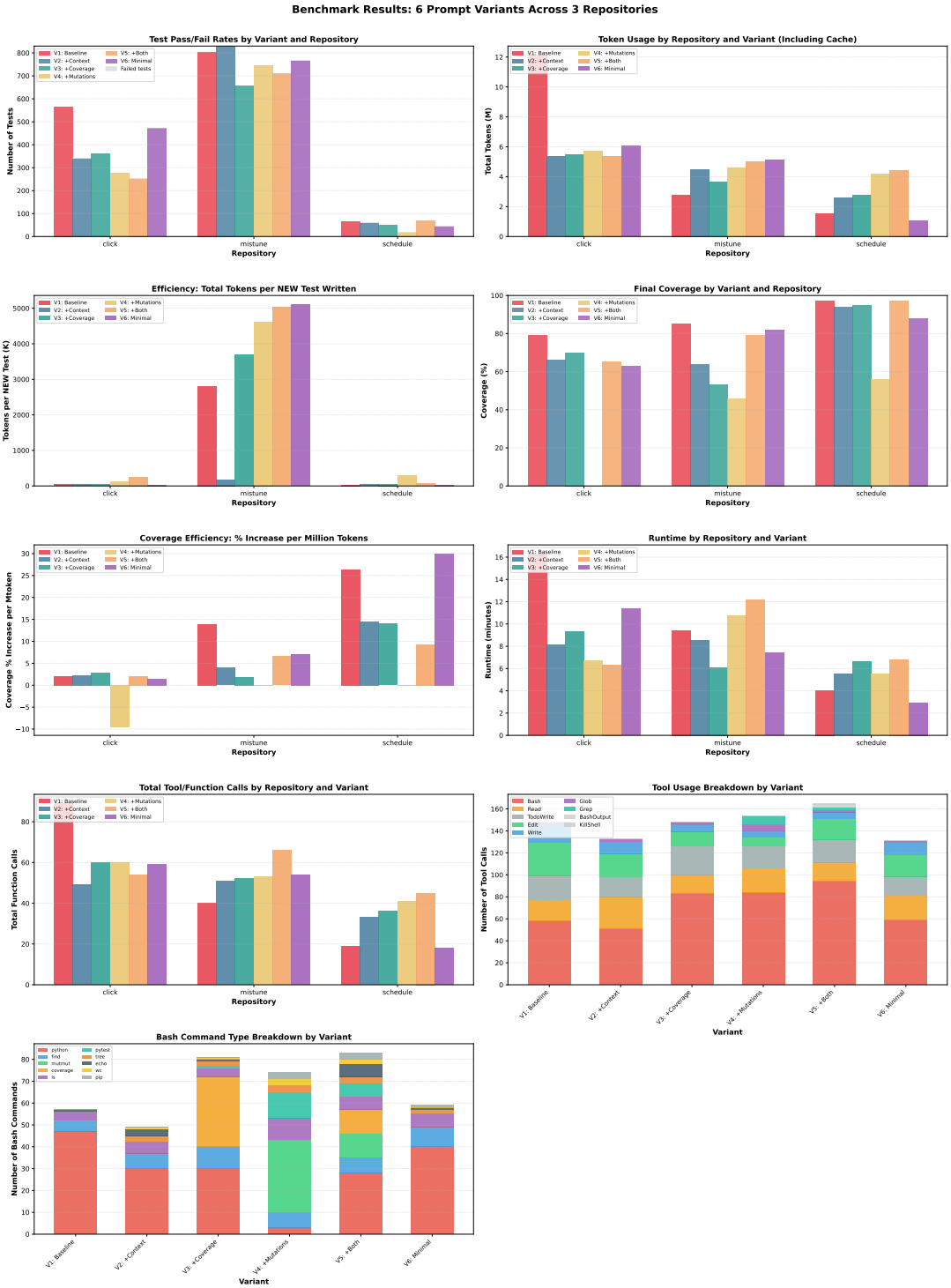

Figure 3: Phase 2 Experiment 1: Reduced coverage repositories (click, mistune, schedule). These repos had tests artificially removed to create coverage gaps. Top row: test pass/fail counts, token usage, tokens per new test. Middle row: final coverage (dashed lines show baseline before intervention), coverage efficiency, runtime. Bottom row: tool usage patterns. Key finding: V6 (Minimal) achieves competitive coverage with substantially fewer tokens across all three repositories.

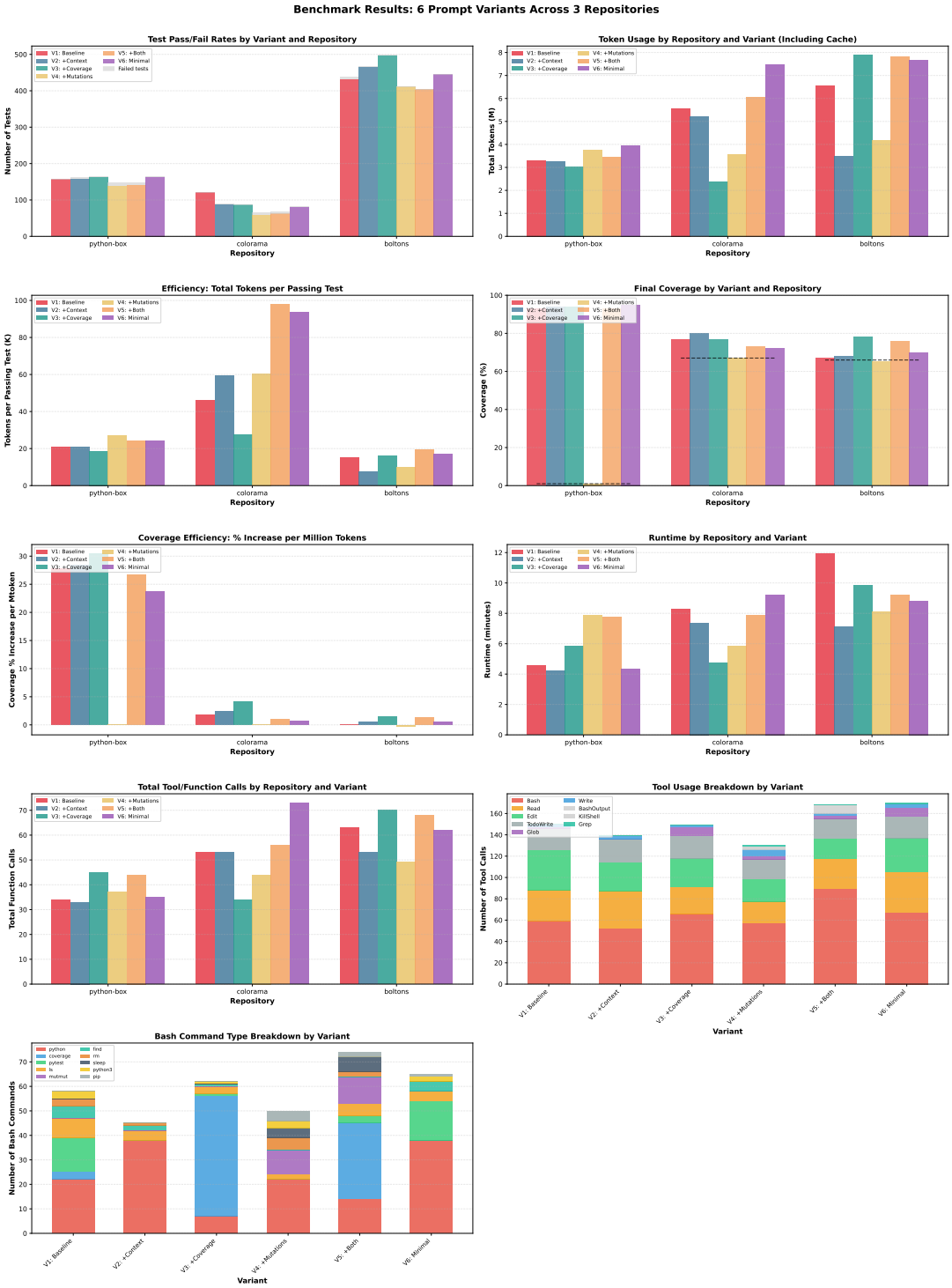

Figure 4: Phase 2 Experiment 2: Low existing coverage repositories (python-box, colorama, boltons). These repos had naturally low coverage without artificial test removal. Dashed lines in the coverage panel show baseline coverage before intervention. Note the dramatic improvement for python-box (near 0% to 80%) vs. modest gains for colorama and boltons (already at 70%).

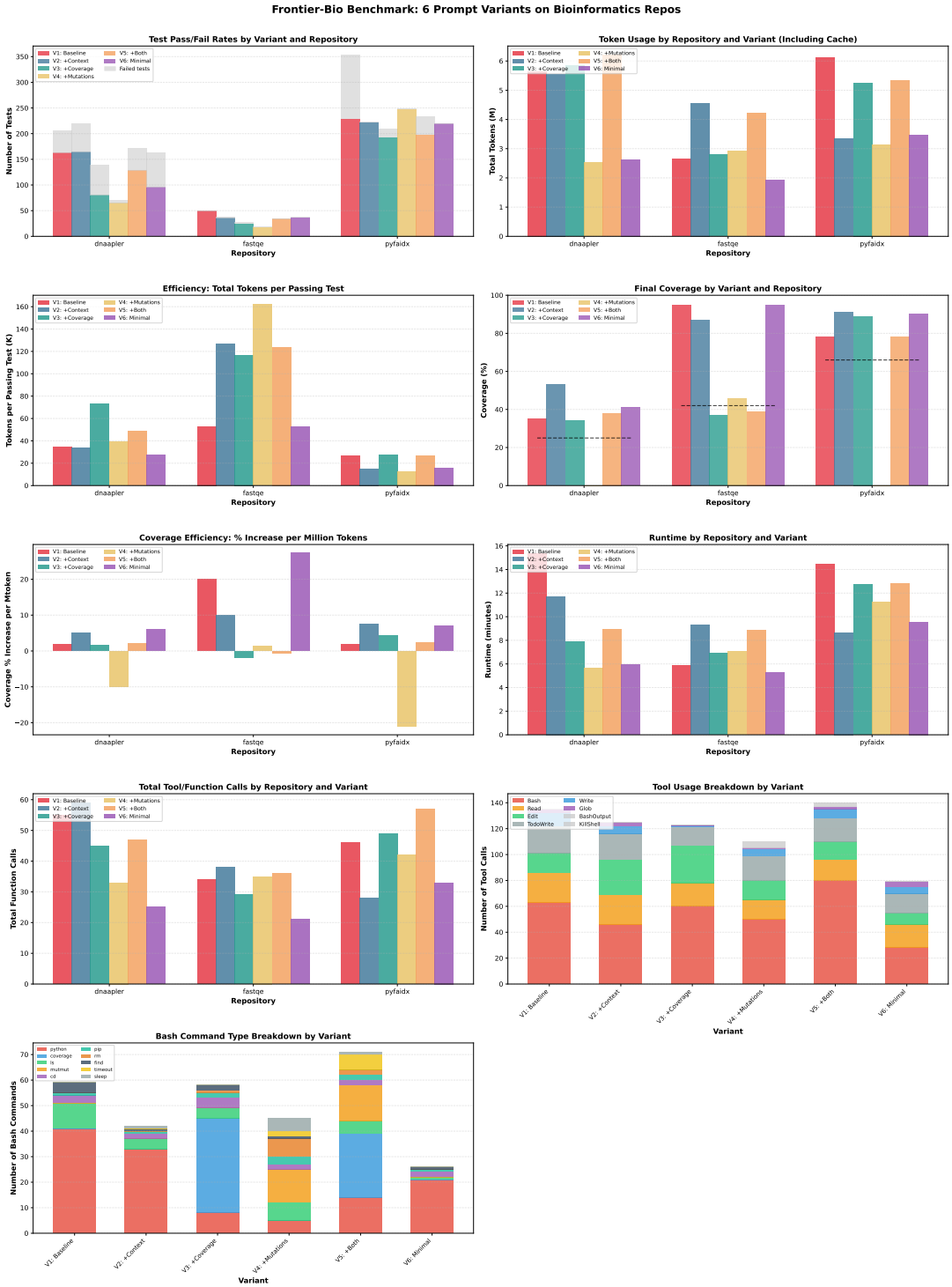

Figure 5: Phase 2 Experiment 3: Bioinformatics repositories (dnaapler, fastqe, pyfaidx). These domain-specific repos revealed unique challenges. Note the gray "Failed tests" bars in the top-left panel showing high failure rates. The coverage efficiency panel (middle-left) shows V4 (+Mutations) achieving *negative* efficiency on fastqe—coverage actually decreased. Dashed lines in coverage panel show baseline before intervention.

- **V6 used fewer Read calls**, suggesting it wrote tests more directly rather than extensively analyzing
- **All variants used TodoWrite** similarly, indicating task management was prompt-agnostic

4.6 Statistical Pattern Analysis

While n=1 per configuration limits individual comparisons, patterns across 9 repositories and 6 variants enable statistical testing of consistency:

Mutation Testing Failure (V4): V4 underperformed V1 in 9/9 repositories (100%), with 5/9 achieving ≤10% coverage. A sign test for V4 < V1 yields p=0.002, indicating this pattern is highly unlikely due to chance. The mean coverage difference was 44.1 percentage points (V1: 75.3%, V4: 31.2%), with Cohen's d = 1.8 (very large effect).

Variance Decomposition: Across 6 general-purpose repositories and 5 variants (excluding V4), between-repository variance was 132 compared to between-variant variance of 5.2—a ratio of 25.5×. Repository characteristics explained 87% of total variance; prompt strategy explained only 3%.

Repository characteristics accounted for 87% of the total variance in coverage outcomes, while prompt strategy choice explained only 3%. The remaining 10% was unexplained variance (potentially including LLM non-determinism and measurement noise).

Token Efficiency: Contrary to initial hypotheses, V6 (Minimal) used fewer tokens than V5 (+Both) in only 2/6 repositories. The sign test (p=0.89) and Wilcoxon signed-rank test (p=0.69) found no significant token efficiency advantage for minimal prompts.

Domain Effects: Bioinformatics repositories averaged 60% coverage vs. 83% for general repos. Mann-Whitney U test (p=0.19) was not significant, likely due to small sample size (n=3 bio repos).

Cost-Per-Coverage-Point Analysis: At Claude API rates ($3/M input, $15/M output tokens), V6 achieved coverage at $0.08 per percentage point (schedule repo) vs. $0.35 per point for V5—a 4.4× cost advantage. Across all repos, V6 averaged $0.15/point while V4 averaged $0.89/point (including failures where infinite cost applies to zero-coverage runs).

Interpretation: These tests address pattern *consistency* across repositories, not LLM variance within configurations. The mutation testing failure and variance decomposition findings are statistically robust; token efficiency and domain effect claims require replication.

5 DISCUSSION

5.1 Repository Characteristics Dominate Prompt Effects

Variance decomposition reveals repository characteristics explained **25.5× more variance** than prompt strategy. The schedule library achieved 88–97% coverage across all variants, while click ranged from 0–79%. Factors like existing test infrastructure, API design patterns, and dependency complexity influence outcomes more than prompt engineering. Practitioners should focus on understanding codebase characteristics rather than seeking "optimal prompts."

5.2 Why Simpler Prompts Remain Competitive

Despite our hypothesis that elaborate prompts would improve outcomes, we found no statistically significant token efficiency advantage for minimal prompts (sign test, p=0.89). Yet simple baselines consistently matched the coverage achieved by elaborate multi-phase workflows. Several factors may explain this finding: (1) complex prompts may overload the model's attention, forcing suboptimal execution patterns; (2) frontier models have already internalized effective testing practices from training data; and (3) each additional workflow phase introduces opportunities for compounding errors that offset any methodological benefits.

5.3 Why Mutations Catastrophically Failed

The +Mutations variant (V4) achieved ≤10% coverage in **5 of 9 repositories** (56%), with V4 underperforming the baseline V1 in 9/9 cases (sign test, p=0.002). This is not occasional reduction—it is *systematic failure*:

Repository	V1 Cov.	V4 Cov.	Δ	Outcome
click	79%	0%	−79 pts	Catastrophic
python-box	93%	1%	−92 pts	Catastrophic
dnaapler	35%	0%	−35 pts	Catastrophic
fastqe	95%	0%	−95 pts	Catastrophic
pyfaidx	50%	0%	−50 pts	Catastrophic[†]
schedule	97%	56%	−41 pts	Degraded
mistune	85%	46%	−39 pts	Degraded
colorama	77%	67%	−10 pts	Degraded
boltons	67%	65%	−2 pts	Degraded

[†]Catastrophic = final coverage ≤10%. Degraded = coverage decreased but >10%.

Explanations: (1) *Infinite loops*—hundreds of surviving mutants exhaust token budgets; (2) *Priority inversion*—agents optimize for killing mutants rather than covering code; (3) *Domain confusion*—mutation results in specialized code are misinterpreted.

Recommendation: Apply mutation testing *only after* achieving ≥70% baseline coverage. Our data suggests it actively harms exploratory test generation.

5.4 Domain-Specific Challenges

Bioinformatics repositories achieved 60% mean coverage vs. 83% for general-purpose libraries (not statistically significant: Mann-Whitney U, p=0.19, n=3). Contributing factors: specialized domain knowledge underrepresented in training, complex dependencies on external databases/file formats, and particular vulnerability to mutation testing failure (all 3 bio repos showed V4 coverage ≤0%).

5.5 Implications for Agentic AI

Broader implications: (1) start minimal before adding complexity; (2) complex workflows should demonstrate improvement over baselines; (3) each tool integration is a potential failure mode; (4) strategies for general code may fail in specialized domains.

5.6 Threats to Validity

Internal: With n=1, differences may reflect LLM non-determinism. Token anomalies (click/Opus using 10.6M vs. 2.4M for Sonnet) suggest unusual conditions in some runs.

External: All repos were Python; results may not generalize. Repository selection was convenience-based.

Construct: Coverage doesn't capture test quality. **Critically, session logs reveal tests rely heavily on `mock.Mock()` rather than real execution**—mock-heavy tests may miss real bugs. Future work should evaluate assertion quality explicitly.

Conclusion/Reproducibility: All findings are preliminary observations requiring replication. LLM non-determinism makes exact replication impossible.

Open Science Statement: All experimental artifacts are publicly available at https://github.com/rhowardstone/aTSR, including: 54 configuration metrics (`results/phase2/*/metrics.json`), full prompt text for all 6 variants, reproducible statistical analysis code (Python), and testScout audit trails with screenshots, prompts, and decision logs.

5.7 testScout Evaluation

Bridging aTSR and testScout: The aTSR experiments above focus on backend test generation, where we found that simple prompts achieve competitive coverage. However, even perfect line coverage cannot detect integration failures at architectural boundaries. The JobsCoach P0 routing bug occurred at the FastAPI route registration layer—a configuration concern orthogonal to unit test coverage. This gap between coverage metrics (aTSR's domain) and integration correctness motivated testScout's development. Where aTSR improves the *depth* of backend testing, testScout addresses the *breadth* of system-level validation through visual E2E testing.

testScout operates in two modes: **Discovery Mode** (no API key required) performs element identification and Set-of-Marks overlay generation; **AI Mode** (requires Gemini/GPT-4V) adds vision-based action execution and autonomous exploration. This graceful degradation enables testing infrastructure validation without API costs.

Discovery Mode Evaluation: Tested on 15 diverse websites. testScout achieved 100% success rate, discovering 698 elements (avg 46.5/site, 11.2ms avg discovery). Only 17% of elements had stable CSS selectors; the rest would require brittle position-dependent selectors. Set-of-Marks achieved 100% reliability via stable `data-testscout-id` attributes.

Case Study: testScout identified 6 bugs in JobsCoach: P0 route ordering (invisible to unit tests), P1 database/isolation errors, P2 UI duplication. At $0.01–$0.05/test, finding these cost ~$0.50—preventing hours of debugging [2].

AI Mode: Gemini 2.5 Pro discovered 738 elements across 4 sites with 100% success. Audit trails generated for reproducibility.

Figure 6 shows testScout's Set-of-Marks overlay on Wikipedia.

Figure 6: testScout's Set-of-Marks overlay on Wikipedia, identifying 280 interactive elements. Red numbered boxes enable vision-model targeting without fragile CSS selectors.

6 CONCLUSION AND FUTURE WORK

We presented aTSR and testScout, two open-source tools for AI-assisted testing. Key observations (n=1, requiring replication): (1) minimal prompts achieved competitive coverage at lower token cost; (2) mutation feedback catastrophically failed in 56% of repos; (3) repository characteristics dominated prompt effects by 25×; (4) bioinformatics showed higher failure rates; (5) Sonnet matched Opus at lower cost.

6.1 Future Work

Directions: hybrid approaches (minimal first, tools for gaps), adaptive domain detection, cross-language evaluation, human quality assessment, larger-scale studies with multiple runs, and domain-specific failure analysis.

6.2 Availability

Both tools are available under MIT license:

- aTSR: https://github.com/rhowardstone/aTSR
- testScout: https://github.com/rhowardstone/testscout

ACKNOWLEDGMENTS

This work was completed as part of CSE 5095: AI for Software Engineering at the University of Connecticut. Thanks to Dr. Tingting Yu for guidance and feedback throughout the project.

REFERENCES

[1] Anthropic. 2025. Claude Code: An Agentic Coding Tool. https://github.com/anthropics/claude-code. Accessed: 2025-09-10.

[2] Barry Boehm and Victor R. Basili. 2001. *Software Defect Reduction Top 10 List.* Vol. 34. IEEE. 135–137 pages. Documents the 10-100x cost ratio for defects found late vs. early in development.

[3] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. ChatUniTest: A Framework for LLM-Based Test Generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. ACM, 572–576. https://doi.org/10.1145/3663529.3663801

[4] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, 416–419. https://doi.org/10.1145/2025113.2025179

[5] Rye Howard-Stone and Ion I. Măndoiu. 2025. AmpliconHunter: A Scalable Tool for PCR Amplicon Prediction from Microbiome Samples. *arXiv preprint arXiv:2509.13300* (2025). https://doi.org/10.48550/arXiv.2509.13300

[6] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678. https://doi.org/10.1109/TSE.2010.62

[7] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. In *Proceedings of the Twelfth International Conference on Learning Representations (ICLR '24)*. Oral presentation.

[8] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE, 919–931. https://doi.org/10.1109/ICSE48619.2023.00085

[9] Junwei Liu, Kaixin Wang, Yixuan Zhu, Xin Xia, Changjian Fan, Zhongxin Li, and David Lo. 2024. Large Language Model-Based Agents for Software Engineering: A Survey. *arXiv preprint arXiv:2409.02977* (2024).

[10] Wei Ma, Shangqing Liu, Zhihao Lin, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, Li Li, and Yang Liu. 2023. LLMs: Understanding Code Syntax and Semantics for Code Analysis. *arXiv preprint arXiv:2305.12138* (2023).

[11] Microsoft Research. 2023. Set-of-Mark Prompting for Visual Grounding. https://github.com/microsoft/SoM. Accessed: 2025-09-10.

[12] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-Directed Random Testing for Java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (OOPSLA '07)*. ACM, 815–816.

[13] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. *arXiv preprint arXiv:2302.06590* (2023).

[14] Juan Altmayer Pizzorno and Emery D. Berger. 2024. CoverUp: Coverage-Guided LLM-Based Test Generation. *arXiv preprint arXiv:2403.16218* (2024).

[15] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (2024), 85–105. https://doi.org/10.1109/TSE.2023.3334955

[16] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. https://doi.org/10.1145/3491101.3519665

[17] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-Consistency Improves Chain of Thought Reasoning in Language Models. *arXiv preprint arXiv:2203.11171* (2022).

[18] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.