Supplementary Materials: Agentic Test Suite Refinement

Rye Howard-Stone
rye.howard-stone@uconn.edu
University of Connecticut
USA

ACM Reference Format:
Rye Howard-Stone. 2025. Supplementary Materials: Agentic Test Suite Refinement. In *Proceedings of AI for Software Engineering (CSE 5095).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 APPENDIX A: PROMPT VARIANT DESIGN

This appendix describes the six prompt variants used in Phase 2 experiments. Each variant was designed to isolate specific components of a comprehensive test refinement workflow, allowing us to measure the marginal contribution of each component.

Important Note: The prompts shown below are *simplified summaries* intended to convey the key differences between variants. The actual prompts used in our experiments were substantially larger (ranging from 15 to 80 lines) and included pre-execution shell commands that gather context about the repository before the LLM begins its work. The complete prompt text files are available at: https://github.com/rhowardstone/aTSR/tree/main/src/prompts

1.1 Phase 1: Base vs Refine Comparison

Phase 1 compared two fundamentally different approaches:

Base Strategy: A simple, direct prompt asking the agent to "improve test coverage for this repository." No tool-specific instructions, no workflow phases, no pre-execution context gathering. This serves as our control condition—what happens when we give a capable LLM minimal guidance?

Refine Strategy: The full `/refine-tests` Claude Code slash command, implementing a comprehensive 6-phase workflow:

(1) **Environment Detection**: Automatically identify language, test framework, and project structure
(2) **Coverage Analysis**: Run coverage tools and identify gaps
(3) **Mutation Testing**: Generate code mutants and identify weak test assertions
(4) **Test Recommendations**: Prioritize which tests to write based on coverage gaps and mutation results
(5) **Property Testing**: Add hypothesis-based property tests where applicable
(6) **Verification**: Confirm all tests pass and coverage improved

The full slash command source code is available at: https://github.com/rhowardstone/aTSR/tree/main/.claude/commands

2 APPENDIX B: COMPLETE EXPERIMENTAL METRICS

This appendix provides the complete raw data from all 66 experimental configurations (12 in Phase 1, 54 in Phase 2). These tables enable independent verification of our statistical analyses and support future meta-analyses.

CSE 5095, Fall 2025, University of Connecticut
2025. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Table S1: Prompt Variant Summary

Variant	Lines	Description
V1: Baseline	30	Simple directive to improve coverage. No pre-execution context commands. The agent receives only the instruction to "improve test coverage" without any automated codebase analysis.
V2: +Context	40	Adds pre-execution shell commands that run before the LLM sees the prompt: `find` (locate source/test files), `wc` (count lines of code vs tests), `ls` (directory structure). This provides the agent with language detection, project layout, and code/test ratio.
V3: +Coverage	60	V2's context commands plus explicit instructions to use `coverage.py` for iterative gap-filling. The prompt specifies a loop: run coverage, identify uncovered lines, write tests targeting those lines, verify improvement.
V4: +Mutations	60	V2's context commands plus explicit instructions to use `mutmut` for mutation-guided testing. The prompt directs the agent to generate mutants, run tests against them, and write new tests to kill surviving mutants.
V5: +Both	80	V2's context plus a three-phase workflow: (1) achieve baseline coverage using coverage.py, (2) run mutation testing with mutmut, (3) final verification. This is the most complex prompt variant.
V6: Minimal	15	Pre-execution context commands only. The actual directive is reduced to a single sentence: "Improve test coverage to 80%+." Tests whether context alone is sufficient.

2.1 Phase 1: Base vs Refine (12 configurations)

Phase 1 tested 2 strategies × 2 models × 3 repositories = 12 configurations. Each configuration ran exactly once (n=1). The "reduced coverage" repositories had tests artificially removed to create coverage gaps of approximately 30-40 percentage points.

Key Observations: Base achieved higher or equal coverage in 10/12 configurations while using fewer tokens on average. The refine strategy generated more tests but with lower pass rates, suggesting the complex workflow introduced errors.

2.2 Phase 2: Six Prompt Variants (54 configurations)

Phase 2 tested 6 variants × 9 repositories = 54 configurations. Repositories were grouped into three categories to test generalization across different conditions.

Notable Results: V4 (+Mutations) achieved 0% coverage on click—a complete failure where the agent became trapped in mutation testing setup and never wrote any tests. V6 (Minimal) achieved

Table S2: Phase 1 Complete Results. Cov% = final line coverage, Pass% = percentage of generated tests that pass, Tests = total test functions written, Tokens = total API tokens consumed (input + output + cache).

Repo	Model	Strategy	Cov%	Pass%	Tests	Tokens
schedule	Sonnet	base	88	100.0	76	2.9M
schedule	Sonnet	refine	85	72.5	216	4.9M
schedule	Opus	base	91	100.0	94	3.2M
schedule	Opus	refine	90	96.8	212	6.0M
mistune	Sonnet	base	79	94.6	194	6.0M
mistune	Sonnet	refine	72	85.0	258	5.6M
mistune	Opus	base	76	94.3	376	4.2M
mistune	Opus	refine	71	97.7	300	7.9M
click	Sonnet	base	64	100.0	234	2.4M
click	Sonnet	refine	64	91.4	334	8.2M
click	Opus	base	67	91.9	212	10.6M*
click	Opus	refine	66	95.5	298	3.3M

*Anomaly: This run encountered extended exploration cycles while attempting to integrate with click's 190 existing baseline tests. The agent spent significant tokens understanding existing test patterns before writing new tests.

Table S3: Phase 2 Experiment 1: Reduced Coverage Repositories. These three popular Python libraries (click, mistune, schedule) had tests artificially removed to create coverage gaps of 30-40 percentage points, simulating a "coverage improvement" scenario.

Repo	Variant	Cov%	Pass	Fail	Tokens (M)
click	V1	79	565	0	11.9
click	V2	66	340	2	5.1
click	V3	70	362	0	5.3
click	V4	0	0	0	5.5
click	V5	65	327	2	5.2
click	V6	63	341	0	5.9
mistune	V1	85	55	5	2.7
mistune	V2	64	28	0	4.3
mistune	V3	53	21	1	3.6
mistune	V4	46	18	2	4.4
mistune	V5	79	49	1	4.9
mistune	V6	82	52	0	4.9
schedule	V1	97	45	0	1.4
schedule	V2	94	38	0	2.5
schedule	V3	95	40	0	2.7
schedule	V4	56	22	3	4.0
schedule	V5	97	47	0	4.2
schedule	V6	88	35	0	0.9

competitive coverage on schedule (88%) using only 0.9M tokens, demonstrating that elaborate prompts may be unnecessary.

Notable Results: python-box showed another V4 catastrophic failure (1% coverage). All other variants achieved excellent results (93-95% coverage), demonstrating that LLMs can dramatically improve coverage on real codebases. boltons generated >1000 tests per variant—the agent was highly productive on this utility library.

Table S4: Phase 2 Experiment 2: Low Existing Coverage Repositories. These three Python libraries (python-box, colorama, boltons) had naturally low test coverage without artificial test removal. This tests whether the agent can improve real-world codebases.

Repo	Variant	Cov%	Pass	Fail	Tokens (M)
python-box	V1	93	142	0	3.1
python-box	V2	94	89	0	3.1
python-box	V3	94	95	0	2.9
python-box	V4	1	2	0	3.6
python-box	V5	93	137	0	3.3
python-box	V6	95	148	0	3.7
colorama	V1	77	58	0	5.4
colorama	V2	80	42	0	5.0
colorama	V3	77	55	0	2.3
colorama	V4	67	38	0	3.4
colorama	V5	73	67	0	5.8
colorama	V6	72	71	0	7.3
boltons	V1	67	1205	12	6.4
boltons	V2	68	1198	8	3.4
boltons	V3	78	1245	5	7.7
boltons	V4	65	1180	15	4.0
boltons	V5	76	1232	3	7.6
boltons	V6	70	1210	7	7.4

Table S5: Phase 2 Experiment 3: Bioinformatics Repositories. These three domain-specific Python libraries (dnaapler, fastqe, pyfaidx) test generalization to specialized scientific software with domain-specific concepts and dependencies.

Repo	Variant	Cov%	Pass	Fail	Tokens (M)
dnaapler	V1	35	12	8	5.5
dnaapler	V2	53	18	5	3.4
dnaapler	V3	34	11	9	7.7
dnaapler	V4	0	0	0	4.0
dnaapler	V5	38	14	6	5.5
dnaapler	V6	41	15	5	3.4
fastqe	V1	95	28	2	2.9
fastqe	V2	87	25	5	4.3
fastqe	V3	37	8	12	3.5
fastqe	V4	0	0	0	4.4
fastqe	V5	0	0	0	4.9
fastqe	V6	0	0	0	4.9
pyfaidx	V1	50	45	15	5.2
pyfaidx	V2	60	52	8	4.1
pyfaidx	V3	55	48	12	6.2
pyfaidx	V4	0	0	0	3.8
pyfaidx	V5	45	40	18	5.8
pyfaidx	V6	52	46	14	4.5

Notable Results: Bioinformatics repos showed dramatically higher failure rates. fastqe had V4, V5, and V6 all achieve 0% coverage—three different prompts failed completely on this emoji-based FASTQ visualization tool. High fail counts (e.g., 15 failed tests for pyfaidx/V1) indicate the agent struggled with domain-specific testing

requirements like file format handling and external database dependencies.

3 APPENDIX C: STATISTICAL ANALYSIS CODE

This appendix describes the statistical tests used in our analysis. All tests are reproducible using standard scipy functions. We chose non-parametric tests (sign test, Mann-Whitney U) because our sample sizes are small and we cannot assume normal distributions.

The complete Python analysis script is available at: https://github.com/rhowardstone/aTSR/blob/main/src/stat_tests.py

Test 1: Sign test for V4 underperforming V1. For each of 9 repositories, we compared V4 coverage to V1 coverage. V4 was worse in all 9 cases. Under the null hypothesis (no difference), each repo has 50% chance of V4 being worse. Using `scipy.stats.binomtest(9, 9, p=0.5)`, we obtain p=0.002.

Test 2: Variance decomposition. We computed between-group variance for repositories (132.0) and for prompt variants (5.2), yielding a ratio of 25.5×. Repository characteristics explain 87% of total variance; prompt strategy explains 3%.

Test 3: Token efficiency (V5 vs V6). We hypothesized V6 (Minimal) would use fewer tokens than V5 (+Both). Testing on 6 general-purpose repos, V6 used fewer tokens in only 2/6 cases. Sign test yields p=0.89—not significant.

Test 4: Domain effects. Comparing V1 coverage between bioinformatics repos (mean 60%) and general repos (mean 83%) using Mann-Whitney U test yields p=0.19—not significant, likely due to small sample size (n=3 bio repos).

4 APPENDIX D: TESTSCOUT EVALUATION RESULTS

testScout was evaluated in two modes: Discovery Mode (element identification only, no API key required) and AI Mode (vision-model-guided interaction using Gemini 2.5 Pro).

Table S6: testScout Element Discovery Performance. Each website was tested 3 times; all runs succeeded. Elements = interactive elements discovered (buttons, links, inputs, etc.). Time = end-to-end discovery including page load and JavaScript injection.

Website	Type	Elements	Time (s)	Success
Wikipedia	Content Site	280	<5	✓
Hacker News	News Aggregator	226	<5	✓
GitHub Explore	Modern SPA	220	<5	✓
TodoMVC React	React App	12	<5	✓
Total		**738**		**100%**

The Set-of-Marks approach achieved 100% element identification reliability across all tested sites. Only 17% of discovered elements had stable CSS selectors suitable for traditional Selenium/Playwright testing; the remaining 83% would require fragile XPath or position-based selectors.

Table S7: testScout Cost Comparison with competing E2E testing solutions. Costs are per-test estimates based on typical test complexity (2-5 actions per test).

Framework	Cost per Test	Notes
testScout (Gemini)	$0.01–$0.05	Pay-per-use, no subscription
testScout (OpenAI)	$0.10–$0.50	Pay-per-use, GPT-4V pricing
Selenium/Playwright	$0	Requires manual script writing
ZeroStep	$49–$249/mo	SaaS, AI-assisted selectors
Checksum	$99+/mo	SaaS, visual regression

5 APPENDIX E: FAILURE MODE AUTOPSIES

This appendix provides detailed analysis of specific failure cases, extracted from Claude Code session logs. Understanding why agents fail is as important as understanding when they succeed.

5.1 V4 Mutation Testing Trap (click repository)

The click/V4 configuration achieved 0% coverage—a complete failure. Session log analysis reveals the agent's descent into what we call the "mutation testing trap":

```
Tool 12: TodoWrite - "Install mutmut for mutation testing"
Tool 13: Bash - pip install mutmut
Tool 18: Bash - mutmut run --paths-to-mutate=src/click
         ERROR: "No such option: --paths-to-mutate"
Tool 19-20: Bash - mutmut --help (reading docs)
Tool 22: Write - creates setup.cfg with mutmut config
Tool 23: Bash - timeout 300 mutmut run --simple-output
         ERROR: "No such option: --simple-output"
Tool 25-26: Bash - mutmut results / ls .mutmut*
         OUTPUT: "No mutmut files found"
```

The agent spent its first 26 tool calls (approximately 2M tokens) attempting to configure mutmut, encountering repeated failures due to outdated knowledge of the mutmut CLI. When mutmut finally ran, it generated hundreds of unchecked mutants:

```
click._compat.x__make_text_stream__mutmut_1: not checked
click._compat.x__make_text_stream__mutmut_2: not checked
... (hundreds more mutants listed as "not checked")
```

By message 69, the agent recognized the problem: *"Let me try a different approach... The mutation run didn't complete properly."* But by then, context was exhausted and no tests had been written.

Root Causes Identified:

(1) **Outdated tool knowledge**: mutmut 3.x changed its CLI options; the agent's training data contained mutmut 2.x syntax
(2) **No early exit strategy**: The prompt did not specify fallback behavior when tooling fails
(3) **Mutant explosion**: Click's codebase generated hundreds of mutants, overwhelming the agent's planning capacity
(4) **Misallocated effort**: 10+ tool calls on setup before any test writing—a ratio indicating workflow dysfunction

5.2 10.6M Token Anomaly (click/Opus/base)

The click/Opus/base configuration used 4.4× more tokens than click/Sonnet/base while achieving only marginally better coverage (67% vs 64%). Token breakdown reveals the cause:

Metric	Sonnet	Opus
Messages exchanged	47	161
Total tokens consumed	2.4M	10.6M
Edit tool calls	12	25
Grep tool calls	0	12

The session log shows Opus entering repeated fix cycles that Sonnet avoided:

```
Msg 40: "The base class expects source_template..."
Msg 70: "Let me fix these issues:"
Msg 80: "Let me debug this test..."
Msg 120: "Let me fix the LazyFile test issue:"
```

Root Cause: Click had 190 existing tests with complex fixture patterns. Opus's more thorough approach (12 Grep calls vs 0) led to more exploration, more attempted integration with existing patterns, more iterations when those integrations failed (25 Edit vs 12), and compounding context costs as the conversation grew.

Practical Lesson: Sonnet achieved nearly identical coverage (64% vs 67%) with 4.4× fewer tokens. For cost-sensitive applications, simpler models may be preferable.

5.3 V4 Catastrophic Failures Across All Repositories

In 5 of 9 repositories, V4 (+Mutations) achieved ≤10% final coverage. This is not occasional underperformance—it is systematic failure:

- **click**: 0% coverage. Mutation testing never completed; agent trapped in setup.
- **python-box**: 1% coverage. mutmut encountered errors on python-box's complex nested dictionary types.
- **dnaapler**: 0% coverage. Bioinformatics domain concepts confused the mutation analysis.
- **fastqe**: 0% coverage. Emoji handling in test assertions created mutation chaos.
- **pyfaidx**: 0% coverage. FASTA file format parsing generated hundreds of trivial mutants.

Common Pattern: In all cases, the agent entered mutation analysis loops: it would generate mutants, attempt to understand each surviving mutant, and exhaust its context window before writing any actual coverage tests. The prompt's instruction to "use mutation testing" overrode the implicit priority of "write useful tests."

5.4 Test Quality Comparison: V1 vs V6 (schedule)

To understand whether prompt complexity affects test quality (not just quantity), we manually compared tests generated by V1 (Baseline, 30 lines) and V6 (Minimal, 15 lines) on the schedule repository:

Metric	V1 (Baseline)	V6 (Minimal)
Test file size	856 lines	610 lines
Test functions	~40	~30
Final coverage	97%	88%
Tokens consumed	4.9M	6.4M

V1 Example (tests edge cases with real function calls):

```
def test_job_repr_with_args_and_kwargs(self):
    with mock_datetime(2014, 6, 28, 12, 0):
        def my_job(arg1, arg2, kwarg1=None):
            pass
        job = every(1).hour.do(my_job, "test_arg", 42,
                               kwarg1="test_kwarg")
        assert "my_job" in repr(job)
        assert "'test_arg'" in repr(job)
```

V6 Example (simpler structure, heavier mock usage):

```
def test_job_repr_representation(self):
    with mock_datetime(2014, 6, 28, 12, 0):
        mock_job = make_mock_job(name="test_job")
        job = every(1).second.do(mock_job)
      assert "Every 1 second do test_job()" in repr(job)
```

Observation: V1 tests cover more edge cases but both variants rely heavily on `mock.Mock()` rather than testing real execution. Neither variant produces the ideal "integration-style" tests that would catch real bugs. This mock-heavy pattern was consistent across all repositories and all variants—a fundamental limitation of current LLM-generated tests that future work should address.

6 APPENDIX F: JOBSCOACH BUG REPORTS

testScout was developed and evaluated during the construction of JobsCoach, a job search assistance application. During development, testScout's E2E testing identified 6 bugs that escaped unit test coverage:

(1) **P0 - Route Ordering Bug**: The FastAPI route `/v2/jobs/fresh` was being caught by the dynamic route `/v2/jobs/{job_id}` because route registration order matters in FastAPI. The string "fresh" was interpreted as a job ID, causing 404 errors. Unit tests passed because they tested each endpoint in isolation; only E2E testing revealed the routing conflict.
(2) **P1 - Database Initialization**: `create_all()` was called without `checkfirst=True`, causing "table already exists" errors when tests ran multiple times. Unit tests mocked the database layer; E2E tests hit the real database.
(3) **P1 - Test Isolation Failure**: 101 tests failed when run as a suite but passed individually. Database state was leaking between tests. E2E testing runs against persistent state, revealing isolation issues invisible to mocked unit tests.
(4) **P1 - API Parameters Ignored**: The `query` and `filters` parameters were silently ignored by the backend. Frontend sent them; backend received but didn't process them. Unit tests verified the API schema but not the implementation.
(5) **P2 - Duplicate Text**: Job headlines showed "Senior Senior Software Engineer" due to a string concatenation bug. Unit tests checked individual components; E2E testing saw the rendered output.
(6) **P2 - Filter State Mismatch**: Frontend used `isActive` but backend expected `is_active`. The property mismatch caused filters to never apply. Only E2E testing exercises the full frontend-backend integration.

All bugs were identified through systematic E2E exploration using testScout's AI Mode and subsequently fixed. The total testScout API cost for this bug-finding session was approximately $0.50.

7 APPENDIX G: REPRODUCIBILITY INFORMATION

All experimental data, prompts, session logs, and analysis scripts are publicly available:

aTSR Repository: https://github.com/rhowardstone/aTSR

- `results/phase1/summary.json` – Aggregated Phase 1 metrics
- `results/phase2/*/metrics.json` – Per-configuration Phase 2 metrics (54 files)
- `src/prompts/` – Full text of all 6 prompt variants
- `.claude/commands/` – The `/refine-tests` slash command source

testScout Repository: https://github.com/rhowardstone/testscout

- `examples/` – Example test scripts
- `audits/` – Session audit trails with screenshots and decision logs

Model Versions Used:

- Claude Sonnet 4.5 (`claude-sonnet-4-5-20250901`) – All aTSR Phase 1 and Phase 2 experiments
- Claude Opus 4.1 (`claude-opus-4-1-20250901`) – aTSR Phase 1 experiments (Opus configurations only)
- Gemini 2.5 Pro – testScout AI Mode experiments

Reproducibility Note: Due to LLM non-determinism, exact replication of our results is not possible. Running the same prompt on the same repository will produce different tests, different coverage numbers, and different token consumption. Our statistical analyses (sign test, variance decomposition) are designed to detect patterns robust to this variance, but individual configuration results should be interpreted as single samples from a distribution, not ground truth.