Automated Software Test Generation at Industry Scale Using a Multi-Agent Architecture and Workflow Integration

Matas Rastenis
matas@uber.com
Uber Technologies Inc
Seattle, WA, USA

Ben Chou
ben10@uber.com
Uber Technologies Inc
San Francisco, CA, USA

Shauvik Roy Choudhary
shauvik@uber.com
Uber Technologies Inc
Seattle, WA, USA

René Just*
rjust@cs.washington.edu
University of Washington
Seattle, WA, USA

Abstract

This paper introduces AutoCover, a production system that automatically generates, validates, and repairs software tests using large language models (LLMs). AutoCover now generates about 11% of all new tests that are reviewed and added to Uber's codebase.

AutoCover supports three complementary interaction modes: (1) CLI for local scripting and developer workflows, (2) Headless for generating tests at scale across repository shards and creating merge requests, and (3) IDE for human-in-the-loop generation that captures intent and provides rapid test repair. Together, these modes support both legacy code in existing repositories and newly written code during development, before it is committed to any repository.

AutoCover is implemented as a modular, agentic pipeline built on LangGraph, with sub-graphs for preparation, generation, execution, and validation/repair. Repository adapters supply language- & repository-specific actions and commands, and a code-context retriever feeds only relevant symbols to the LLM to remain within context budgets. To prevent low-quality tests, AutoCover couples intent-aware generation with quality gates and flakiness defenses.

This paper describes AutoCover's end-to-end architecture and evolution, together with an empirical evaluation and considerations for user experience, test quality, and cost.

CCS Concepts

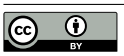
• **Software and its engineering → Software verification and validation**.

Keywords

Artificial Intelligence, Automated Software Testing, Industrial Study

ACM Reference Format:
Matas Rastenis, Ben Chou, Shauvik Roy Choudhary, and René Just. 2026. Automated Software Test Generation at Industry Scale Using a Multi-Agent Architecture and Workflow Integration. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-SEIP '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3786583.3786918

*Work done at Uber.



ICSE-SEIP '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2426-8/2026/04
https://doi.org/10.1145/3786583.3786918

1 Introduction

A comprehensive suite of automated software tests increase confidence in software correctness, prevent regressions, and aid large-scale, automated refactoring and code-migration efforts. At Uber, thousands of engineers contribute to large monorepos with millions of lines of code, and maintaining and increasing test coverage is key to both developer productivity and software reliability. Our goal is to automatically generate effective software tests.

Manually developing and maintaining automated tests is time-consuming, creating tension between development speed and test quality. Achieving high test coverage in modern, large-scale software systems is non-trivial and often requires multiple testing frameworks, analysis techniques, build-system integrations, and careful trade-offs [13]. Examples of such trade-offs include balancing test coverage and maintenance costs, the use of real dependencies and mocks, and test thoroughness and execution time.

Automated test generation has seen renewed interest with the advent of Large Language Models (LLMs), both in purpose-built test generators and in general coding assistants. Empirical studies show that LLM-based generators can achieve competitive coverage and produce tests developers prefer when paired with lightweight analysis or refine-and-retry loops [22, 24, 28]. At the same time, reports caution that generated tests may be incomplete, brittle, or misaligned with project configurations, necessitating careful review and integration with real build and artifact pipelines [5, 9, 11].

Generating effective tests requires context-sensitive choices that a one-size-fits-all test generator cannot make autonomously. First, additional tests increase confidence but also add review and ownership burden [13]. Second, real dependencies tend to be slow and flaky but provide high fidelity, whereas mocks are fast and hermetic but prone to overfitting. Third, thorough concurrency or benchmark tests may catch regressions that lightweight tests miss, but generating and executing them takes longer. Fourth, exercising distinct contract scenarios through the same lines of code can provide value even without increasing line coverage [12]. Fifth, deterministic tests require stable clocks, hermetic I/O, and seeded randomness, which are essential for maintaining trust [23]. Finally, deployment context further constrains computational budgets: IDE usage demands short, predictable latencies, whereas Headless backfills can tolerate longer, parallel runs to expand coverage.

These choices motivate our design for context-sensitive generation (scenario-guided), quality-centric validation (beyond lines covered), and a repair loop that applies minimal, reversible edits—providing fast feedback during development and deeper testing during backfill. We designed, developed, and deployed AutoCover, an LLM-based multi-agent system for automated test generation

at industrial scale. AutoCover supports Go, Java, TypeScript, and Python, and features two core innovations:

(1) **Agentic architecture:** AutoCover orchestrates five specialized agents for test preparation, synthesis, execution, validation, and repair. It tightly integrates with the build system to discover code artifacts and dependencies, including both static and dynamically-generated code.
(2) **Workflow integration:** AutoCover supports three modes of invocation for distinct developer workflows:
   (a) **Headless** runs autonomously across monorepos, generating merge requests with new tests and routing them to owning teams for review.
   (b) **CLI** runs on-demand via explicit command-line invocation, generating tests for specified folders, files, or functions. It modifies the developer's local state with new tests.
   (c) **IDE** runs in the background as developers code, precomputing tests and providing a frictionless experience with the perception of real-time generation.

We refined and gradually deployed AutoCover to all engineers at Uber between September 2024 and September 2025. Our quantitative and qualitative evaluations show:

- AutoCover generates (as of September 2025) on a monthly basis: IDE: hundreds of thousands of tests, Headless: tens of thousands of tests, and CLI: thousands of tests.
- AutoCover's overall success rate of generating viable tests (i.e., passing tests that increase code or scenario coverage) is about 20% for Java, 40% for Go, and 80% for Python.
- AutoCover's viable IDE tests are explicitly accepted by users at a rate of about 44%.
- AutoCover's Validator automatically fixes tens of thousands of developer-written tests. The total number of fixed tests is trending down as the number of automatically generated tests increases, which are valid by design.
- On a sample of production Go code, AutoCover's tests received substantially higher expert quality ratings than those generated by Cursor, while completing the end-to-end generation loop substantially faster.

This paper's organization and contributions are as follows:

- A characterization of challenges to automatically generating high-quality tests in an industrial context (section 2).
- AutoCover, a novel LLM-based multi-agent system for automated test generation (section 3).
- An evaluation of AutoCover (section 4).
- Lessons learned from operationalizing and deploying AutoCover at scale (sections 5 and 6).

2 Motivation

In the summer of 2024, we evaluated the test-generation capabilities of general coding assistants. GitHub Copilot handled simple cases but could not invoke build steps or obtain generated artifacts (coverage, mocks), preventing autonomous iteration in our code repositories. Later, we evaluated Cursor, which improved shell/tool invocation but struggled to persist through multi-step generation and often diverged from required mocking conventions on complex targets. This resulted in outputs that were unacceptable without significant manual intervention. Additionally, Cursor's linear generation led to unacceptably long latency on large files with low existing test coverage.

Why do general coding assistants fail? General prompt-in-IDE tools typically falter for three reasons:

(1) **Build graph and generated-code blindness.** They neither discover the correct build labels nor run codegen, so tests fail to compile with missing mock packages or symbols.
(2) **Transitive dependency discovery.** Without traversing the build graph, assistants miss secondary generated artifacts (e.g., validators' registries, mappers) and attempt ad-hoc stubs that violate real (constructor) constraints.
(3) **Throughput limitations.** Preparation, build, and validation are serialized; there is no fan-out across candidate cases, inflating wall-clock time on large targets.

How AutoCover succeeds. AutoCover couples test synthesis to the build system and parallelizes critical stages:

(1) **Target + artifact plan.** From the file under test, AutoCover resolves the canonical test target and emits an *artifact plan* (transitive dependencies, codegen actions), materializing generated mocks/registries *before* test generation.
(2) **Contract-shaped generation.** A task-aware retriever summarizes control/branch structure (eligibility gate, template validation outcomes, transactional boundary, feed publish, notification branch) so assertions validate observable behavior, not internals.
(3) **Parallelized E2E loop.** Test generation, AST-aware splicing, compilation, and coverage/mutation checks run in pipelined, parallel workers; failures route to *Fixer* for minimal, reversible edits. This improves context retention per step and isolates faults, consistent with results from multi-agent LLM systems [15, 19, 27].
(4) **Quality gates.** *Validator* rejects change-detector tests and enforces best practices (stable oracles, isolation, table-driven structure), admitting tests that raise line coverage *or* satisfy previously uncovered scenarios.

3 AutoCover

We designed and deployed AutoCover to address four key challenges: (1) Scale & complexity: we target at least 85% coverage for new code, but legacy code often lags. Since legacy code evolves slowly, organic test suite growth is limited and would require dedicated engineering effort. (2) Diverse use cases: new code benefits from fast, interactive IDE flows, whereas legacy backfill requires scalable, unattended generation. (3) Quality beyond coverage: flaky or change-detector tests increase maintenance costs [14, 23]. (4) Build-system integration: industrial repos involve complex build rules and generated artifacts; efficient test preparation, retrieval, build, and execution require first-class build integration.

To address these challenges, AutoCover orchestrates five specialized agents (Figure 1) built on LangGraph [15] and multi-agent patterns such as ReAct [27]. AutoCover supports **Headless** (merge requests without intervention), **CLI** (on-demand), and **IDE** (background precompute). It integrates with the build system and parallelizes generation, execution, fixing, and validation.

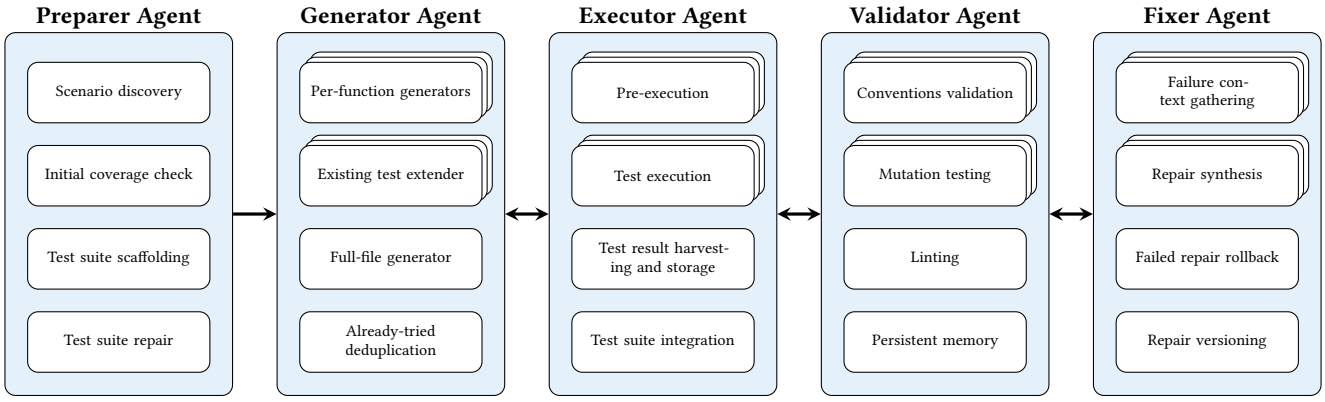


Figure 1: AutoCover's multi-agent architecture that orchestrates five key agents. Double borders indicate parallel tasks.

3.1 Design Rationale

AutoCover's design is guided by the following principles, which together address the key challenges we identified:

- **Specialized agents.** Purpose-built agents split responsibilities, retain focused context, reduce ambiguity, and mitigate hallucinations to improve quality and reduce retries.
- **Reusable nodes.** Encapsulated agents accelerate new flows and enable parallel team development.
- **Language and repository adapters.** Adapters encapsulate language- and repository-specific tooling and policies.
- **Determinism first.** Bounded responsibilities plus deterministic tools improve reliability and debugging.
- **AST-aware splicing.** Structured edits with selective persistence and deterministic rollback.
- **Build-safe parallelism.** Per-test-case target fan-out avoids collisions and reuses caches (Bazel [2]).
- **Table-driven test cases.** Emit table tests by default to avoid brittle post-hoc merges.
- **Durable test cases.** Persist test identity and history to "think longer" on promising test cases and learn from prior failures.
- **Scenario coverage as a signal.** Accept tests that validate previously uncovered behaviors, not just line deltas.
- **Validator as policy gate.** Executable, signal-adding tests enter quality checks; failing tests route to repair.

The *Preparer* → *Generator* → *Executor* → *Validator* → *Fixer* loop runs in parallel where safe, attributes coverage precisely, treats scenarios as first-class acceptance evidence, and steadily upgrades failing candidates into production-quality tests. It preserves determinism and repository hygiene to maintain developer trust.

3.2 Preparer Agent

Role & inputs. Preparer determines *what* to test and *why*, using source context, the build graph, and initial coverage results. In the presence of an initial build failure, it interacts with the Fixer agent to restore build health before resuming. It emits a scenario map and a prioritized target map, and scaffolds a minimal test suite.

Scenario discovery A lightweight reasoning pass summarizes API intent per function and proposes happy-path and edge-case scenarios. This produces a per-function *scenario map* with contract notes, edge conditions, and invariants that later guide acceptance.

Initial coverage check An efficient build+coverage probe establishes a coverage baseline. This information, and the scenario map, is used to create a prioritized *target map* for downstream agents.

Test suite scaffolding Preparer ensures a canonical test file exists and pre-seeds minimal imports and table skeletons where idiomatic. It records adapter hints for downstream agents, including linters, build tools, codegen/mocks, and import conventions.

Test suite repair On initial-build failures only, Preparer invokes the Fixer agent to, e.g., regenerate build files or fix import syntax.

3.3 Generator Agent

Role & inputs. Generator proposes tests that either increase line coverage or satisfy new scenarios, while minimizing downstream repair. It ingests the scenario and target maps, local file context, and feedback from prior attempts.

Per-function generators Generation fans out per source function, guided by scenarios. Workers emit focused candidate tests in parallel, improving throughput without diluting context.

Existing test extender Generator extends existing tests by appending table rows or tightening assertions. This allows it to cover new paths or scenarios maintaining existing test structure.

Full-file generator When function-level parsing fails, Generator operates on the entire file, marking uncovered spans to help recover scenario or line coverage gaps.

Already-tried deduplication A shared store aggregates versioned test cases and de-duplicates candidates by normalized content. This avoids redundant execution and feedback cycles.

3.4 Executor Agent

Role & inputs. Executor compiles and runs candidate tests in parallel with build-safe isolation. It precisely attributes coverage, and splices accepted tests into the final test suite. It consumes the candidate test set, build graph, caches, and adapter policies.

Pre-execution Executor emits an *artifact plan*, from the build graph, and materializes generated code such as mocks concurrently. It pipelines the entire workflow (retrieval → splice → compile → run/coverage → validate) using bounded queues between stages, incremental builds, and caches.

Test execution Executor runs candidate tests in parallel with build-safe isolation. It synthesizes a unified build plan and replicates the test target per test case, thereby avoiding shadow-workspace pitfalls [6] and maximizing cache reuse. Tests execute in isolated Bazel sandboxes with remote/local caching. Determinism is ensured through pinned tool chains, stable seeds, and no global state.
Test result harvesting and storage Executor parses per-target logs and coverage reports, attributes failures to specific test cases, and collects per-(function, scenario) coverage sets.
Test suite integration AST-aware edits (e.g., imports or table-row insertion) with selective persistence retain test cases that compile and contribute a coverage signal. Test cases that regress or add no signal are reverted. Conflicts in helper/test names or import identifiers are detected via AST and resolved via deterministic renames or aliases. Conflicting test cases are re-queued to Fixer with structured feedback.

3.5 Validator Agent

Role & inputs. Validator is the quality gate between execution and persistence. A test case is viable only if it successfully executes and either raises code coverage or scenario coverage.
Conventions validation Validator enforces style, stability, and intent conventions using an LLM-powered best-practices registry. The registry contains machine-readable rules (⟨id, severity, span, rationale, patch, confidence⟩), together with language-scoped, versioned examples, for hermetic IO, seeded randomness, import aliasing, etc., reducing test flakiness [17]. When violations are detected, context-matched examples accompany feedback to guide repairs.
Mutation testing Validator uses mutation testing, with bounded type and count of mutants. Surviving mutants are attributed to specific tests and scenarios, triggering prompts that guide input selection and tighten oracles.
Linting Adapters may skip test executions for obvious violations. For accepted and merged tests, adapters run a final lint pass, where findings are normalized with precise spans and suggested patches.
Persistent memory A stability cache returns prior findings for unchanged files, smoothing LLM variability across runs. Per-target run logs record findings and acceptance reasons for consistent evaluation. They also enable policy-based suppression of repeated low-severity issues. Configuration knobs control re-validation rounds, severity thresholds, and runtime budgets (tools and tokens). Validator emits test case acceptance feedback: accepted tests return to Executor for splicing; non-accepted tests route to Fixer.

3.6 Fixer Agent

Role & inputs. Fixer repairs failing or low-quality test cases through targeted, reversible edits. It operates in parallel (per test case), consuming diagnostics, Validator findings, scenario intent, and edit history. Test cases are prioritized by expected improvement; chronic non-improvers are frozen after repeated failures. This refine-retry approach aligns with evidence that iterative repair improves test compilability and assertion quality [8, 22, 24, 28].
Failure context gathering Fixer aggregates diagnostics and local repository context into a compact per-test-case state to avoid repeating failed strategies.
Repair synthesis Fixer maps Validator findings to concrete edits: improving test structure, strengthening assertions, and fixing environment dependencies. When context is missing, a bounded context crawler retrieves necessary imports, signatures, and examples, using tools such as ls, tree, or read_file. A policy-gated, embedded helper can apply minimal workspace fixes such as regenerating build files or correcting imports.
Failed repair rollback Fixer enforces a "do no harm" policy: patches that degrade passing tests are reverted. All patches are signed and auditable.
Repair versioning Repairs are versioned and merged via Executor's AST-aware splicer. Name and import conflicts are resolved deterministically via renaming or aliasing. Persistent collisions are frozen and annotated with rename instructions. Fixer tracks structured telemetry and respects resource budgets. Successful patterns feed back into same-run prompts.

3.7 Deployment

AutoCover rolled out in a few distinct phases:

(1) **Discovery & team dogfooding (Jun—Jul 2024).** Prototype validated with a small internal group.
(2) **CLI & Headless pilots on remote compute (Aug—Nov 2024).** Early coverage backfills and targeted regression tests at scale.
(3) **Validator-first architecture (Sep—Dec 2024).** Introduced the best-practices registry, early mutation checks, and repository hygiene rules.
(4) **Full parallel pipeline (Jan—Mar 2025).** Multi-target fan-out, parallel fix/execution, deterministic splicing; broader Java/Kotlin support.
(5) **IDE integration & background precompute (Apr—Aug 2025).** Latency-focused with artifact warming, scenario planning, and cost tuning.

Additionally, we continuously improved language support and scalability: (1) *Language expansion & adapters* for Go, Java, TypeScript, and Python, (2) *language-specific mock- and codegen awareness* plus conflict checks, and (3) *operational hardening & scale* through quotas and fallbacks (centralized gateway), prompt caching, anonymization, canaries/SLOs, and direct API client upgrades.

AutoCover was released to developers in distinct mediums over the course of a year. This was a purposeful decision with an initial deployment of AutoCover's CLI, allowing it to run on demand directly on a developer's container or a remote deployment environment. This release aligned with organizational objectives to provide tests for legacy code to prevent future regressions. Releasing an initial version with a rough usage pattern attracted developers who had an operational need for the tool—that is, they needed to work on the aforementioned organizational effort. It also attracted some day-to-day users who integrated the tool into their regular development workflows. This period, before AutoCover was integrated into the IDE extension, was very useful to collect feedback from early adopters to further tune tool performance and efficacy.

A large leap forward in abstraction was made with the IDE extension deployment which effectively targeted all developers using VSCode or a VSCode fork, like Cursor.

4 Evaluation

We evaluated AutoCover using both an intrinsic and an extrinsic evaluation. The intrinsic evaluation uses a curated benchmark to compare AutoCover against baseline coding assistants. Specifically, the evaluation quantifies coverage improvements and tool runtime, and it assesses the quality of the generated tests through an expert review. The extrinsic evaluation uses continuously gathered production data and a user survey to evaluate the efficacy of AutoCover, and AI-assisted test generation in general.

4.1 Benchmark Evaluation of Efficacy

4.1.1 Methodology. We sampled, from Uber's Go code base, 9 subjects that vary in complexity and application domain. Aiming to maximize sample diversity, we chose 3 basic end-to-end programs (**B**), 3 infrastructure components (**I**), and 3 product components (**P**):

- **B1** E2E code implementing a URL parser.
- **B2** E2E code with an internal dependency that requires successful mocking.
- **B3** E2E code for flaky test analysis, determining which test targets need to be run multiple times for assessing flakiness.
- **I1** Implements a hashing system for Bazel targets that generates consistent SHA1 hashes for source files and build rules, supporting both file-based and VCS-based hashing with configurable exclusions and external repository handling.
- **I2** Implements a gateway for managing developer instances with operations like create, update, delete, start, stop, and restart across multiple regions, including name resolution and region validation logic.
- **I3** Implements an NTP metrics collector that monitors network time protocol daemon status, including stratum levels and clock offsets, with graceful handling for unsynchronized states during service startup.
- **P1** Contains mapper functions that convert between RPC protocol and internal objects for handling financial data.
- **P2** Implements a controller for credit card management, handling eligibility, requests, approvals, and notifications.
- **P3** Contains functions for a router service, including slice operations, string conversions, error handling, metrics tagging, data structure transformations, and various helpers.

Pre-deployment, we benchmarked AutoCover and used Cursor (v0.46.8, recent at the time) as a baseline. For each subject, we set a timeout of 60 minutes. *Post-deployment*, we benchmarked two additional versions of Cursor (v1.7.54 and v2.3) and ClaudeCode (v2.0.75) to quantify improvements and variability for these rapidly-changing tools. For a fair comparison, we ran the baselines autonomously with the same information provided to AutoCover. This includes the complete set of rules and detailed descriptions of tools, including the build and test infrastructure.

4.1.2 Results. Figure 2 shows AutoCover's progress over time, broken down by subject type. Each line plot corresponds to one subject and ends at either 100% coverage or the 60-minute timeout. AutoCover reaches full coverage for the basic subjects very quickly (note the different x-axis scale). For infrastructure and product subjects, AutoCover reaches high degrees of coverage quickly for most subjects, followed by a long tail until it reaches full coverage or a plateau. Compared to Cursor-0.46.8, AutoCover achieves higher coverage in a fraction of the time. AutoCover outperforms Cursor on every subject, and Cursor failed to produce runnable tests for three subjects. Note that Cursor's plot for B2 is truncated at 15 minutes, but its coverage remained at 0% for the entire run. More recent versions of Cursor and ClaudeCode perform better than Cursor-0.46.8, yet AutoCover still outperforms these newer baselines without additional tuning. Table 1 summarizes the benchmark results for six distinct time and coverage thresholds.

Table 1: Aggregated benchmark results across all subjects.

(a) Median coverage (%) achieved within time budget.

Tool	Time budget					
	5min	10min	15min	30min	45min	60min
AutoCover	40.0%	93.2%	98.5%	99.4%	100.0%	100.0%
Cursor-0.46.8	0.0%	0.0%	0.0%	40.8%	40.8%	40.8%
Cursor-1.7.54	0.0%	0.0%	97.0%	98.7%	98.7%	98.7%
Cursor-2.3	0.0%	0.0%	0.0%	96.7%	96.7%	96.7%
ClaudeCode-2.0.75	0.0%	0.0%	90.4%	90.4%	98.1%	100.0%

(b) Median time (minutes) to achieve coverage target.

Tool	Coverage target					
	40%	80%	90%	95%	99%	100%
AutoCover	4.7	7.2	8.3	10.8	19.5	41.6
Cursor-0.46.8	30.0	✗	✗	✗	✗	✗
Cursor-1.7.54	11.6	11.6	11.6	11.6	✗	✗
Cursor-2.3	17.0	17.0	17.0	17.0	✗	✗
ClaudeCode-2.0.75	12.8	13.8	13.8	41.4	48.9	48.9

✗: target not achieved for 50% of subjects within 60 minutes.

4.2 Expert Evaluation of Test Quality

4.2.1 Methodology. To evaluate the quality of the generated tests and the extent to which they are useful and acceptable to engineers, we conducted a review with two experts who have a PhD in Computer Science and extensive industry experience. Both experts have deep language expertise and knowledge about Uber's code base. Neither expert is an author of this paper.

We ran AutoCover and Cursor-0.46.8 on each of the nine benchmark subjects and asked the two experts to review the resulting tests according to the following classification:

- *Very good* — **Landable**. Tests are meaningful *and* thorough: exercise contractually important scenarios, use stable behavioral assertions, follow idioms (table-driven where applicable), are lint-clean and deterministic.
- *Good* — **Landable**. Tests are meaningful and reasonably thorough, but could benefit from minor improvements (naming, structure, and minor duplication).
- *OK* — **Landable with light edits**. Enough coverage/intent to justify inclusion, but misses some depth or polish (e.g., incomplete negative paths or overly specific assertions).

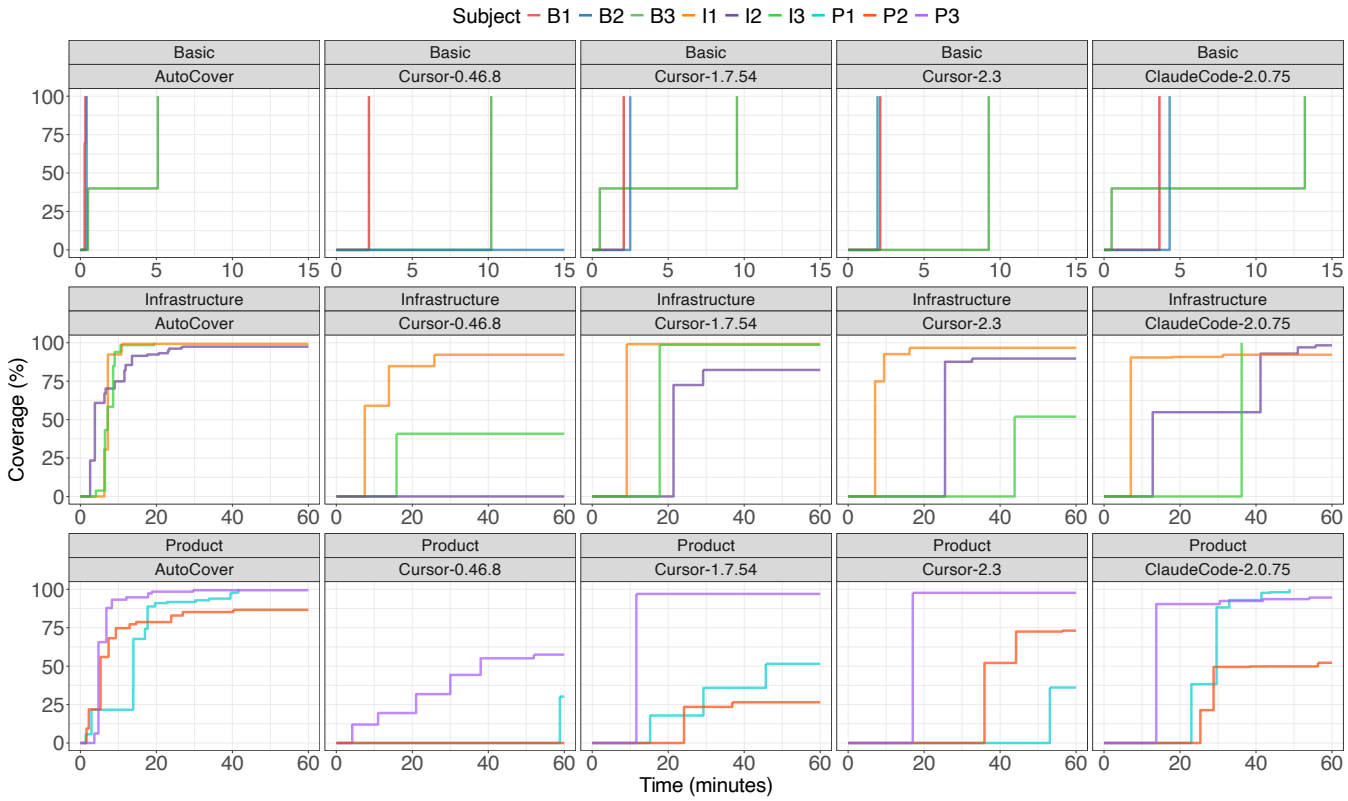


Figure 2: AutoCover's progression for each of the nine benchmark subjects, broken down by subject type.

- *Bad* — **Not landable**. Technically buildable, but low value or brittle: change-detector patterns (asserting internal details), flaky setup/teardown, improper mocking that obscures behavior. Requires substantial revisions to be landable.
- *Very bad* — **Not landable**. Actively harmful to test suite health: non-determinism (time/environment leaks), mutable globals, external I/O without fakes, fragile order-dependence. Must be rejected and regenerated.
- *Broken* — **Not landable**. No output or unbuildable tests (syntax/type/import errors), or zero execution (config errors).

4.2.2 Results. Figure 3 shows the distributions of the expert ratings for AutoCover and Cursor-0.46.8. Both reviews exhibit bi-modal distributions, but with different rating patterns. Reviewer 1 gave stronger scores, suggesting that tests are either acceptable as-is or not at all. In contrast, Reviewer 2 gave more moderate scores, suggesting that some tests may be acceptable after (light) revisions. Overall, the expert reviewers favored AutoCover, which generated no broken tests and more tests that are acceptable as-is.

4.3 Deployment Evaluation of Efficacy

4.3.1 Methodology. AutoCover has evolved over the past year. As such, we report on key characteristics achieved at the time of making the tool generally available to all engineers at Uber. Additionally, we quantify the effect of AutoCover's caching strategies, broken down by agent and token volume.

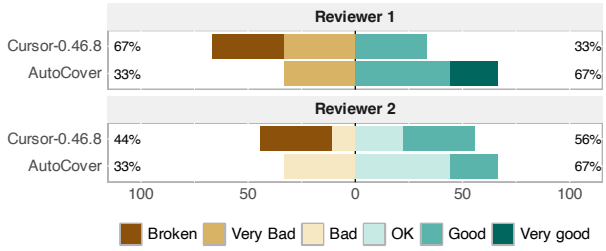


Figure 3: Expert ratings for final test outputs (N=9 subjects).

4.3.2 Results. As of August 2025, AutoCover is available to all engineers at Uber. While adoption is still increasing, we have observed the following results for August and September 2025:

- Per month, AutoCover generates hundreds of thousands of tests for IDE use, tens of thousands of tests for Headless use, and thousands of tests for CLI use.
- The overall success rate of generating (viable) passing tests that increase code or scenario coverage is about 20% for Java, 40% for Go, and 80% for Python.
- AutoCover's viable IDE tests are explicitly accepted by users at a rate of about 44%.
- AutoCover's Validator automatically fixes tens of thousands of developer-written tests.

Table 2 shows the efficacy of AutoCover's caching strategies (see section 5), broken down by agent and underlying model. Overall, the two caching strategies substantially reduce the cost of test generation, compared to a naive implementation.

4.4 User Survey

4.4.1 Methodology. We surveyed engineers with experience using AutoCover across all three interaction modes (CLI, Headless, and IDE). To ensure meaningful exposure, respondents were required to have generated at least 10 tests (CLI or Headless) or accepted at least 10 tests (IDE) in the prior two months. The survey included Likert-scale questions on test quality, usefulness, and integration experience, as well as open-ended questions soliciting improvement suggestions. Given AutoCover's tight integration with the IDE, survey responses reflect experiences with AI-assisted test generation more broadly (e.g., including auto completion and chat-based test generation). Free-text responses clarified which specific improvements users would like to see in AutoCover.

We received 29 responses: 23 active users and 6 inactive users, who self-identified as having significantly reduced tool usage.

4.4.2 Results. Figure 4 summarizes the survey results, comparing responses from active and inactive users. Both groups agree that AI-assisted test generation encourages adding more tests (65 vs. 67% agreement). However, active users report more positive experiences across all other dimensions: 83% (vs. 40%) agree that AI-assisted test generation improves productivity and 61% (vs. 40%) report a positive user experience.

For tool attributes, active users rated usability highest (83%), followed by quality of generated tests (70%) and speed (65%). Inactive users rated all three attributes substantially lower, with none rating test quality as good or excellent.

The free-text responses revealed several recurring themes for desired improvements (some responses mentioned multiple issues):

- **Style (6):** Generated tests should match existing project conventions, even if they diverge from broader conventions and best practices. This includes table-driven tests, consistent assertion patterns, and proper use of established mock libraries. Multiple respondents noted that generated tests ignored idioms already present in the codebase.
- **Bloat (4):** Generated tests are often verbose, with redundant variables, unnecessary assertions, and excessive scenarios that are not always applicable.
- **Speed (4):** Test generation is too slow, in particular for CLI usage. A faster feedback loop would encourage greater adoption.
- **Guidelines (4):** Support for project-specific rules, such as preferring table tests or using `mockgen` instead of manual mocks. This also includes the ability to specify specific mock libraries to use, as generated mocks are sometimes not easily discoverable.
- **Coverage integration (3):** Support for a tighter integration with coverage tooling, including the ability to target only new or changed lines and more visibility into which lines are covered.
- **Interactivity (1):** Ability to interject during generation to guide an agent when it is close but not quite correct.

Table 2: Cache hit rates for AutoCover's individual agents.

Agent (Model)	Tokens	Cache hit rate
OpenAI GPT 5 Thinking		
Preparer	$O(10^8)$	9.5%
OpenAI GPT 4.1		
Validator (CLI & Headless)	$O(10^6)$	30.2%
Validator (IDE)	$O(10^9)$	91.7%
Fixer (Context crawler)	$O(10^9)$	59.5%
Claude 4 Sonnet		
Generator	$O(10^{10})$	53.5%
Fixer (Other)	$O(10^8)$	22.1%

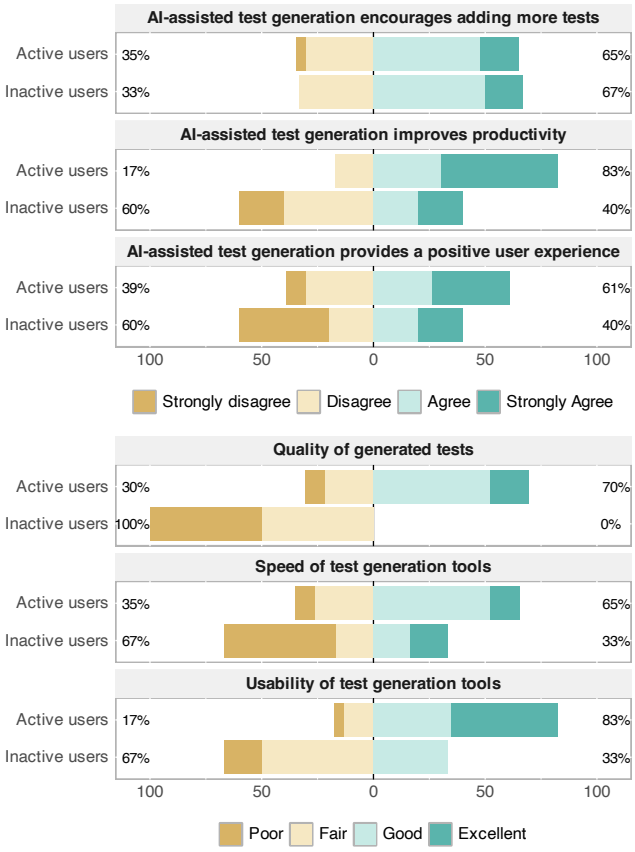


Figure 4: Survey results: overall assessment of AI-assisted test generation across all modalities (N=29 participants).

4.5 Threats to Validity

We ground the discussion of internal, construct, and external validity in AutoCover's design, deployment, and evaluation metrics. We acknowledge that high internal validity is often at odds with real-world deployment. As is true for many industrial contexts, we favor external and ecological validity (realism and generalizability) over internal validity (tight experimental control).

Internal validity Code coverage guidelines/targets and individual team adoption may have influenced developers' behavior when it comes to accepting generated tests. We addressed this threat with a dedicated expert review of test quality, which showed that observed acceptance rates are in line with the expert rating. We tuned in-house LLM gateway anonymization to avoid mangling identifiers. It is unclear whether this decision and potential source of syntax breakages may have an influence on observed efficacy. Multiple improvements landed during the study window (model usage changes, prompt restructuring, parallelization, and Validator rule updates). It is difficult to tease out individual effects, which is one of the reasons we report on overall trends and averages.

Construct validity *Coverage ≠ effectiveness.* Code coverage is an imperfect proxy for test quality. We mitigate this threat via scenario coverage and mutation testing in the Validator. Scenario coverage accounting started later than line coverage. As such, the coverage signals in 2024 are not comparable to those in 2025 and we do not report on direct comparisons. Expert reviews are subjective and it is possible that the experts assessed the quality differently than what our study intended. We mitigated this threat by documenting the quality rubric and clarifying it prior to the review.

External validity The results reported in this paper are tied to the development environment and processes at Uber. However, the development tools and practices used at Uber are similar to those used at other large technology companies. Additionally, AutoCover supports four languages, which provides additional confidence in generalizability of the results. Supporting other languages is also possible due to the extensible language adapter paradigm. Example Patterns and policy rules reflect local idioms, which may not generalize. It is possible that the deployment results are affected, to some degree, by a target selection bias. Developers may have targeted particularly hard-to-test legacy code with CLI or Headless runs, not necessarily reflecting the expected day-to-day experience for newly developed code. For the intrinsic evaluation, we chose a random sample regardless of testedness to mitigate this threat.

5 Operationalization

We hardened AutoCover for always-on, multi-tenant production use. This section details how we manage LLM inference provider quotas and fallbacks, control cost via caching and prompt design, prevent regressions, observe the system with fine-grained telemetry, and preserve privacy with code-aware anonymization.

5.1 Resource Quota Pressure

Centralized routing AutoCover uses Uber's internal LLM routing layer for cost attribution, anonymization, and provider/-model selection. This indirection enables instant failover across providers/models during incidents and provides unified visibility into response codes, throttling, and latency distributions [3].

Right-sizing quotas Estimating steady-state token and request rates proved challenging due to workload variability. While over-provisioned quotas penalize providers, under-provisioned quotas throttle users. We adopted a conservative base quota and engineered the client to absorb variance through:

- *Multi-level model fallbacks*: the scheduler uses two fallbacks (primary → previous-gen → cross-vendor alternatives), ordered by observed capability. Each fallback introduces an incremental delay, so the scheduler prefers waiting a short backoff window for the primary when queues are shallow.
- *Adaptive concurrency control*: per-tenant and global semaphores, token-bucket gating on fan-out bursts, and jittered exponential backoff on 429/5xx responses [1, 20].
- *Priority lanes*: interactive IDE sessions preempt Headless backfill traffic. We enforce fairness with per-user and per-repo ceilings to prevent starvation.
- *Circuit breakers*: rapid demotion of a provider/model when error or latency SLOs are violated. Providers are automatically placed on probation and gradually re-introduced when they meet the SLOs again.

5.2 Cost

Two-tier caching AutoCover uses two complementary caches:

(1) **Literal response caching** at the inference platform (effective for identical prompts).
(2) **Native prompt caching** (e.g., message-level caching) that allows stable prompt segments to be re-used across requests.

Parallel generation/fixing/validation changes file and target context frequently, making literal cache hits uncommon. To maximize cacheability, we restructured prompts into *stable* vs. *volatile* blocks:

- *Stable*: best-practices registry, language adapter guidance, repository conventions, validator rubric, and style.
- *Semi-stable*: target source function contract and API surface (file hash changes invalidate the cache).
- *Volatile*: current test file slice, coverage diffs, failure logs, and per-case history.

We warm the cache with a *pilot* Generator request before fanning out, yielding substantial cache hits for Generator and Fixer (table 2). Further Fixer improvements include normalizing diagnostics with stable IDs for common failures and trimming non-deterministic log fragments. Both techniques raise cache affinity.

Cost shaping We bias model selection by task criticality (interactive IDE sessions vs. Headless backfill). We use smaller reasoning tiers for lint-only checks and cap retries per test case. To reduce redundant work, we deduplicate identical (function, scenario) shards across concurrent users by content hashing. Cache invalidation is TTL-based and keyed by (file hash, function span, imports).

5.3 Regression Prevention

E2E guardrails We maintain a suite of end-to-end scenarios that exercise the full pipeline on representative complex targets. These scenarios run [4]:

- *Pre-merge*: for agent and runtime changes, and prompt-schema updates.
- *Nightly*: in canary environments and on branches that evaluate different model/provider mixes.
- *On-demand*: after provider incidents or gating rule updates.

Signals include time-to-first-coverage, accepted-test count, scenario coverage, Validator pass rate, and flakiness rate.

Alerting and blast-radius control We define SLOs (e.g., P95 generation latency for IDE sessions) and alert on fast drops or slow drifts. Failed canaries automatically hold rollouts (feature flags). We maintain runbooks for common regressions (anonymizer drift, cache stampedes, tooling updates) and provide a one-click rollback for prompt/agent configurations.

5.4 Telemetry

Structured eventing Every stage emits structured telemetry with a shared correlation ID to enable cross-agent timeline reconstruction. Representative counters, timers, and distributions:

- *Throughput*: invocations by mode (IDE, Headless, and CLI), generated vs. inserted tests, acceptance reasons.
- *Quality*: Validator pass/fail by rule, mutation survivors per function, lint errors by class.
- *Coverage*: events/lines per invocation, scenario coverage deltas, attribution by function and scenario.
- *Reliability*: stage latencies, queue times, per-provider/model error codes, fallback counts and chains.
- *Cost*: tokens by stage, prompt-cache hit ratios (by stable vs. volatile blocks), compute per accepted test.

Dashboards & SLOs We track key performance indicators: P50/P95/P99 latencies (time-to-first-coverage), acceptance rates, cache hit rates, and provider health. Seasonality-aware anomaly detection flags sudden shifts (e.g., drop in Generator cache hits). We alert on user-facing SLO breaches (IDE latency, acceptance rate) and upstream degradation (provider/model spikes in 429/5xx responses, anonymizer rewrite errors).

Operability We enrich logs with minimal, privacy-safe context (hashed repo path, language, target size bucket). Distributed traces attach tool I/O digests (e.g., coverage file hash) for reproducibility. Feature flags guard risky changes (e.g., new Validator rules, anonymizer patterns, executor strategies), enabling per-cohort rollouts and quick rollback.

5.5 Anonymization

Generic scrubbers designed to remove personally identifiable information (PII) from prompts misclassify code identifiers (e.g., `User`) as personal information, corrupting syntax and imports.

We tuned the gateway's anonymizer to distinguish code identifiers from actual PII, effectively eliminating invalid anonymizations. To ensure correctness, we validate anonymization with syntax-roundtrip tests (parse before/after) and import-resolution checks, preventing accidental breakage from reaching the model.

6 User Experience

AutoCover's initial deployment focused on CLI and Headless execution, targeting explicit invocation for coverage backfills and regression prevention. While effective, these modes required manual invocation and imposed wait times on developers. This section describes the design and deployment of AutoCover's IDE integration, which aimed at improving user experience for day-to-day development by eliminating invocation friction through automatic background execution and precomputation.

6.1 Design Rationale

AutoCover's multi-agent pipeline satisfied functional requirements and showed promising results in intrinsic evaluations. To enable broader adoption, we explored several alternatives to the IDE integration we eventually settled on: extending the existing CLI with watch modes, deploying background agents on developer workstations, and integrating with AI chat assistants. Each alternative had limitations that rendered it unsuitable for our goals: watch modes still required explicit setup, background agents lacked UI integration points, and chat-based generation required context switching away from code editing.

The IDE integration emerged as the most promising approach, with goals to: (1) eliminate invocation friction by running automatically in the background, (2) reduce perceived latency through precomputation, and (3) enable company-wide deployment via automatic IDE extension installation.

6.2 IDE Integration

There is precedent for adding AI features into the IDE: Copilot Code Completion [25] and Cursor's Tab [7]. Both of these features seem IDE-native which tends to boost general adoption and approval. The IDE serves as a common denominator across developer workflows and provides existing infrastructure for automatic extension deployment. By initializing at session startup and running in parallel to active development, the IDE mode precomputes tests in the background, achieving both goals: eliminating manual invocation and hiding generation latency. The IDE-native presentation increases discoverability and encourages developers to interact with generated tests naturally.

6.3 Implementation Challenges

Deploying AutoCover at scale via IDE integration introduced several implementation challenges related to cost control, shared resources, and operational flexibility.

Automatic background execution Automatic background execution addresses two key adoption barriers: (1) test generation takes time, requiring developers to wait for results, and (2) an explicit invocation requirement creates friction that discourages tool usage. By precomputing results in the background, AutoCover creates the perception of instantaneous generation when developers need tests. It monitors file system events (file open, save, edit counts) with execution debouncing and in-process resource management to deliver results automatically without degrading developer workflows.

Cost management Cost Management is important, in particular when it comes to integrating AI tools and rolling out at scale. This requires fine-tuning what is included in prompts due to costs scaling with input and output tokens. Context and prompt building happens in the core AutoCover pipeline itself, but the IDE extension orchestrates the cadence at which AutoCover is automatically run in the background.

Invocation strategy Balancing quality with cost required extensive experimentation with background execution timing. Our data showed that user engagement with files does not follow a linear trajectory—developers often browse code without actively editing. We needed to trigger AutoCover only when users are

likely actively working on files, avoiding wasted executions during passive browsing. Using line-change count as a heuristic also proved unreliable, adding implementation complexity and producing inconsistent execution behavior for smaller files.

The final invocation pattern that is orchestrated by the extension is to queue up a 5 minute debounced invocation of AutoCover upon file save whose debounce timer resets on each subsequent save. We have found that 5 minutes after a save is conservative enough to adequately reduce costs of background execution at scale. The objective in this configuration is to reduce wasted background execution and costs by only running AutoCover automatically for those who would value the tests most.

Shared workspace Integration of AutoCover into the IDE is inherently constrained by working in a shared workspace. AutoCover was not originally implemented to work in a shared workspace and thus required additional refactoring to ensure a user's workspace is not modified unknowingly through background execution. AutoCover additionally relies on build-system commands which are executed by a singular shared build-system server (AutoCover relies on Bazel for build and test execution). AutoCover's integration in the IDE was meant to be seamless to the user which meant additional refactoring was needed to ensure AutoCover executed build-system commands at a lower priority.

Remote feature flagging and access control Exposing AutoCover through an extension meant we had to adhere to the constraint of extension versioning and our internal release train to roll out updates to all developer machines. Getting a merged change deployed on the actual development machines of the users may take time in the order of days to a week. This release lag was not acceptable to us because we required the ability to tune in real time, in particular if we see performance degradation or cost increases beyond our projected trajectory. We leveraged an internal personnel-based access control infrastructure to tune features in real time. Features included invocation patterns, cadence, display-level feature flagging, and kill switches.

6.4 Refinement

Integrating AI tooling into the IDE is a delicate balancing act: maximizing adoption requires minimizing friction in the user experience, but unconstrained background execution results in unpredictable and unsustainable costs. Company-wide roll out is complete, and we are actively adapting AutoCover to leverage new features, such as those provided by model providers (e.g., message caching). This enables AutoCover to further reduce costs.

AutoCover's pre-computed tests appear as comments [26] that provide top-level information such as test description and code preview, and render command links dynamically to give users the ability to act upon the generated results. The typical actions provided are: (1) insert test case and ensure it passes, (2) insert test case, (3) dismiss test case. Clicking on one of these actions will emit this action to AutoCover and it will automatically start processing a generated test case in real-time. Progress is automatically reported to the user through the same comment pane, where they can view AutoCover's execution and have the ability to cancel and insert test case if AutoCover is taking too long. Test cases are otherwise automatically inserted directly into the file upon completion and a new comment is assigned to the newly added test with info regarding the test case, such as description and last executed actions. We also added support for bypassing background execution for those who want to explicitly trigger AutoCover on demand.

7 Related Work

Industrial IDE assistants Modern AI-enabled IDEs (e.g., GitHub Copilot, Cursor, Windsurf) emphasize in-editor assistance via completions and conversational edits. Copilot's primary surface is code completion with occasional inline fix-ups. Cursor extends this with chat-driven, file-editing agents and a "Tab"/inline action flow that auto-invokes suggested changes and presents diffs inline. These systems focus on developer ergonomics in the editor—summarizing context and rendering suggestions in place—but generally do not execute repository builds, collect coverage, or enforce repository-specific testing conventions as part of an automated, build-integrated loop. In contrast, AutoCover couples test generation with the build, test, and coverage toolchains as well as quality gates, operating in both Headless and IDE.

Commercial test-generation tools Several production tools target automated unit tests. Diffblue synthesizes JUnit tests with an emphasis on regression safety and code coverage. CodiumAI offers multi-language test suggestions inside the IDE with iterative refinement. Symflower integrates static analysis and quick-run loops to propose compilable tests. These tools typically optimize for per-file usefulness and developer handoff inside the IDE, but are not tightly integrated with monorepo build graphs, AST-aware splicing, and per-case coverage attribution/merging at scale. AutoCover 's contributions are complementary: coverage- and scenario-guided planning, parallel case execution via build-target cloning, validator-driven repair (incl. mutation checks), and table-by-default evolution under repository policies.

Academic test-generation and quality signals Search-based and constraint-driven generation has a long history (e.g., EvoSuite [10], Randoop [21], Pynguin [16]). Recent LLM-forward systems (e.g., IBM's ASTER [22]) demonstrate cross-language test synthesis and improved readability with structured prompts. Our work builds on these lines while addressing *operational* gaps (artifact/mocks/codegen integration, AST-aware splicing, deterministic, and per-case execution at monorepo scale) and elevating *scenario coverage* alongside lines/branches, with a validator that combines best-practices checks, linting, and targeted mutation testing.

On using and misusing coverage Prior work cautions that raw code-coverage percentages are an unreliable proxy for test effectiveness and can be gamed or misapplied as a target metric [14, 18]. We echo this guidance and treat coverage as a necessary but insufficient signal, complementing it with scenario evidence and mutation testing outcomes.

8 Conclusions

This paper reports on the design, development, and deployment of AutoCover, an LLM-based multi-agent system for automated test generation at industrial scale. At this point, AutoCover supports Go, Java, TypeScript, and Python. It is available to all engineers at Uber, and it sees increasing adoption. AutoCover now generates about 11% of all new tests reviewed and added to Uber's codebase.

References

[1] Amazon Web Services. 2024. Error Retries and Exponential Backoff. https://docs.aws.amazon.com/general/latest/gr/api-retries.html. Accessed 2025-09-08.

[2] Bazel Team. 2023. Bazel Build System and Remote Caching. Online documentation. https://bazel.build/

[3] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. 2016. *Site Reliability Engineering: How Google Runs Production Systems.* O'Reilly Media.

[4] Betsy Beyer, Chris Jones, Jennifer Petoff, Niall Richard Murphy, et al. 2018. *The Site Reliability Workbook: Practical Ways to Implement SRE.* O'Reilly Media.

[5] Jeremy Clark. 2024. *Trying and Failing with GitHub Copilot.* https://jeremybytes.blogspot.com/2024/12/trying-and-failing-with-github-copilot.html

[6] Cursor. 2024. *Shadow Workspace.* https://cursor.com/blog/shadow-workspace

[7] Cursor. 2026. Tab: Predict your next edit. https://cursor.com/docs/tab/overview Accessed 2026-01-03.

[8] Amirhossein Deljouyi, Roham Koohestani, Maliheh Izadi, and Andy Zaidman. 2024. UTGen: LLM-Guided Understandable Unit Test Generation. arXiv preprint arXiv:2408.11710. https://arxiv.org/abs/2408.11710

[9] Devansh. 2023. *Built for Demos, Not for Devs: The uncomfortable truth about Cursor.* https://machine-learning-made-simple.medium.com/built-for-demos-not-for-devs-05186132116f

[10] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE).* 416–419.

[11] GitHub. 2023. *Writing tests with GitHub Copilot.* https://docs.github.com/en/copilot/tutorials/write-tests

[12] Marko Ivanković, Ivan Budiselić, Luka Rimanić, Goran Petrović, Gordon Fraser, and René Just. 2025. What Types of Automated Tests do Developers Write?. In *Proceedings of the International Conference on Automation of Software Test (AST).*

[13] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. 2019. Code Coverage at Google. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE).* 955–963.

[14] Marko Ivanković, Goran Petrović, Yana Kulizhskaya, Mateusz Lewko, Luka Kalinovčić, René Just, and Gordon Fraser. 2024. Productive Coverage: Improving the Actionability of Code Coverage. 58–68. https://doi.org/10.1145/3639477.3639733

[15] LangChain, Inc. 2024. LangGraph: Building State Machines for LLM Applications. GitHub repository. https://github.com/langchain-ai/langgraph Accessed 2025-09-09.

[16] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated unit test generation for python. In *Proceedings of the International Conference on Automated Software Engineering (ASE).* 168–172.

[17] Qingzhou Luo, Meiyappan Nagappan, Ye Zhang, Kathryn T. Stolee, Tao Wang, and Nachiappan Nagappan. 2014. An Empirical Analysis of Flaky Tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE).* 643–653. doi:10.1145/2635868.2635920

[18] Brian Marick, John Smith, and Mark Jones. 1999. How to misuse code coverage. In *Proceedings of the 16th Interational Conference on Testing Computer Software.* 16–18.

[19] Microsoft Research AI. 2023. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. Project documentation. https://microsoft.github.io/autogen/

[20] Mark Nottingham, Mike Adams, and Erwin Mutz. 2012. Additional HTTP Status Codes. RFC 6585, RFC Editor. https://datatracker.ietf.org/doc/html/rfc6585

[21] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *Proceedings of the International Conference on Software Engineering (ICSE).* IEEE, 75–84.

[22] Rangeet Pan, Myeongsoo Kim, Rahul Krishna, Raju Pavuluri, and Saurabh Sinha. 2025. ASTER: Natural and Multi-Language Unit Test Generation with LLMs. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP).* 413–424. doi:10.1109/ICSE-SEIP66354.2025.00042

[23] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2023. Please fix this mutant: How do developers resolve mutants surfaced during code review? 150–161.

[24] 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. doi:10.1109/TSE.2024.3368208

[25] Visual Studio Code. 2026. Inline suggestions from GitHub Copilot in VS Code. https://code.visualstudio.com/docs/copilot/ai-powered-suggestions Accessed 2026-01-03.

[26] Visual Studio Code. 2026. VS Code API Reference. https://code.visualstudio.com/api/references/vscode-api Accessed 2026-01-03.

[27] Shunyu Yao, Yao Zhao, Dian Yu, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv preprint arXiv:2210.03629. https://arxiv.org/abs/2210.03629

[28] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2024. Evaluating and Improving ChatGPT for Unit Test Generation. *Proceedings of the ACM on Software Engineering* 1, FSE, Article 76 (2024), 1703–1726 pages. doi:10.1145/3660783