



OVERMEND: AI-POWERED AUTONOMOUS RECOVERY SYSTEM USING PATCH SYNTHESIS AND MUTATION-BASED TRUST EVALUATION

¹Ayush Kumar Choudhary, ²Dr. Y. Sarojini, ³Jyotsna G, ⁴Ashish Sharma, ⁵Aptha K. S.

^{1,3,4,5}Final Year Student, ²Professor

¹Department of Computer Science Engineering (Data Science),

¹AMC Engineering College, Bangalore, India.

Abstract : Large language models (LLMs) enable automated program repair, but patches that pass available tests may still be semantically incorrect. Overmend is an autonomous software recovery system that combines runtime failure ingestion, AST-based fault localization, LLM patch synthesis, isolated sandbox validation, mutation-based trust evaluation, and policy-aware GitHub orchestration. After a candidate passes the available tests, controlled AST mutations are applied only to the changed region; test sensitivity is incorporated into a deterministic trust score together with patch size, blast radius, and policy constraints. The implementation was validated with 150 automated tests, while a separate evaluation on 20 Python bugs compared tests-only and tests-plus-trust acceptance. Tests-only accepted four incorrect patches (20% observed false acceptance); trust evaluation blocked all four in the evaluated sample. Fault localization required 80–150 ms, mutation analysis 6–15 s, and the reported detection-to-Pull-Request path 15–35 s. These results support mutation evidence as an additional acceptance signal, while the limited benchmark and Python-only scope preclude claims of general correctness.

IndexTerms - automated program repair, autonomous software recovery, large language models, patch synthesis, mutation testing, fault localization, trust scoring, software reliability, Docker sandboxing, GitHub automation.

I. INTRODUCTION

Modern software failures require detection, diagnosis, fault localization, patch development, validation, and review. These steps can delay recovery when incidents recur outside normal development cycles. Automated Program Repair (APR) reduces this burden by generating software modifications; recent LLM-based systems extend repair beyond fixed templates by reasoning over source code and repository context [1], [2]. However, patch generation and safe autonomous acceptance are distinct problems.

A central limitation of generate-and-validate repair is that test-suite satisfaction demonstrates plausibility, not semantic correctness. A patch can pass all available tests while implementing unintended behaviour when tests do not sufficiently exercise the repaired functionality [3], [4]. This creates a trust gap for systems permitted to create Pull Requests or merge changes autonomously.

Overmend addresses this gap as a closed-loop recovery system combining runtime failure ingestion, anomaly triage, traceback-and-AST-based localization, LLM patch synthesis, Docker validation, mutation-based trust evaluation, and policy-aware GitHub orchestration. It separates two questions: whether a patch makes the observed tests pass and whether the tests are sensitive to behavioural changes in the repaired region.

The Mutation-Based Trust Engine applies controlled AST-level mutations after ordinary tests pass. Killed mutants indicate local test sensitivity; surviving mutants provide evidence of untested behaviour. This evidence is combined with patch-risk signals to produce a deterministic trust score, after which the Decision Engine selects AUTO_MERGE, HUMAN_REVIEW, or REJECT. Trust is treated as an acceptance signal, not a proof of semantic correctness.

The operational loop captures incidents through an SDK, processes them through a FastAPI control plane and asynchronous workers, validates candidate unified diffs in network-isolated Docker containers, and records accepted evidence in GitHub Pull Requests, audit logs, and recovery memory.

The evaluation investigates three questions: (RQ1) whether runtime evidence and AST localization provide focused context for patch synthesis; (RQ2) how often tests-only acceptance admits incorrect patches in the evaluated bug set; and (RQ3) whether mutation-based trust reduces such false acceptance while retaining practical latency.

The contributions are fourfold: an integrated recovery loop spanning failure ingestion through GitHub delivery; a changed-region mutation layer for evaluating local test sensitivity; a trust model that separates technical evidence from governance constraints; and a controlled comparison of tests-only versus tests-plus-trust acceptance.

The system was validated with 150 automated tests. On 20 Python bugs, tests-only acceptance admitted four incorrect patches, while the trust-enabled configuration blocked all four, motivating evaluation of both repair plausibility and autonomous-acceptance evidence.

Automated software recovery intersects several established areas of software engineering, including automated program repair (APR), large-language-model-based code generation, fault localization, mutation testing, and autonomous software engineering. The existing literature provides strong foundations for automated patch generation and repository-level reasoning, but the evidence required to safely authorize an autonomous repair remains comparatively less explored.

The remainder of this paper is organized as follows. Section II reviews related work in automated program repair, LLM-based repair, fault localization, and mutation testing. Section III presents the Overmend architecture. Section IV describes the recovery, patch-validation, mutation, trust-scoring, and decision methodology. Section V summarizes the implementation. Section VI defines the experimental setup and evaluation metrics. Section VII presents the results and discusses the research questions. Section VIII discusses limitations and threats to validity, and Section IX concludes the paper.

II. RELATED WORK

A. Automated Program Repair

APR has evolved from template- and search-based repair toward learning- and LLM-based systems [1], [2]. Generate-and-validate methods can produce plausible patches that overfit available tests; prior analyses therefore distinguish patch plausibility from correctness [3], [4].

B. LLM-Based and Repository-Level Repair

LLMs expand repair by reasoning over source code, repository context, and natural-language problem descriptions [5]. Repository-level systems such as AutoCodeRover and SWE-agent demonstrate increasingly capable navigation and code modification, while SWE-bench evaluates real-world GitHub issue resolution [6], [7], [13]. These capabilities do not by themselves define evidence sufficient for autonomous acceptance.

C. Fault Localization

Fault localization supplies the context used for repair. Existing work includes spectrum-based, statistical, and program-analysis approaches [11]. Overmend uses the runtime traceback to identify the file and line, then combines the enclosing AST function, nearby source, imports, exception information, and repository history to form a focused context window. The system evaluates this mechanism within the recovery pipeline rather than proposing a new general-purpose localization algorithm.

D. Mutation Testing and Test Sensitivity

Mutation testing evaluates test effectiveness by injecting controlled faults and measuring which are detected [8], [9]. Prior work also examines the relationship between mutants and real faults [10]. Overmend uses mutation analysis after repair validation and restricts mutants to the candidate patch region, treating the result as local test-sensitivity evidence rather than proof of correctness.

E. Research Gap and Positioning

The research gap is the combination of four needs: acceptance beyond test-suite satisfaction [3], [4]; governance beyond LLM repair capability [5]–[7], [13]; mutation evidence targeted to the changed region [8]–[10]; and operational controls such as isolation, protected paths, auditability, and human review. Overmend combines these elements in one recovery loop.

Table I. Architectural positioning of Overmend against representative repair approaches.

Approach	Repair	LLM	Runtime trigger	Mutation trust	Policy gate
Traditional APR	Yes	No/varies	No	No	No
CURE	Yes	Yes	No	No	No
AutoCodeRover	Yes	Yes	No	No	No
SWE-agent	Yes	Yes	No	No	No
Overmend	Yes	Yes	Yes	Yes	Yes

III. SYSTEM ARCHITECTURE

Overmend uses an eight-stage closed loop: (1) failure ingestion and anomaly triage, (2) traceback/AST localization, (3) candidate patch generation, (4) Docker sandbox validation, (5) mutation-based trust evaluation, (6) policy decision, (7) GitHub Pull Request and merge orchestration, and (8) recovery-memory update. A responsive FastAPI control plane delegates repository checkout, model calls, Docker execution, and mutation testing to asynchronous workers.

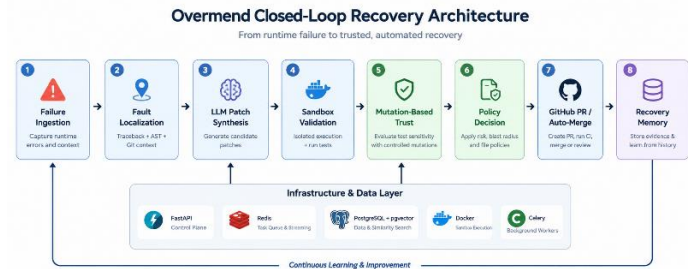


Fig. 1. Eight-stage Overmend recovery loop. Stages 5 and 6 constitute the principal trust and governance contribution.

A lightweight Python SDK captures traceback, frame, commit, and request information. FastAPI persists incident state and fingerprints failures; Celery/Redis handles long-running tasks, PostgreSQL stores recovery state, and pgvector retrieves historical incidents. Workers perform checkout, localization, synthesis, validation, mutation, and policy evaluation; a React/TypeScript dashboard exposes status, diffs, evidence, and decision rationale.

Safety controls require schema-valid LLM output and a dry-run Git application check; excessive rewrites and test-file changes are rejected. Candidate code executes in isolated, network-restricted containers with resource limits, while authentication, payment, migration, secret, Docker, and environment-file paths are protected from automatic merge.

IV. METHODOLOGY

A standardized deviation is then used to determine whether the observed error rate is anomalous. Welford-style online statistics are used in the implementation to update the rolling mean and variance without requiring the complete historical series. Only incidents satisfying the configured triage conditions are passed to the recovery workers.

A. Failure Capture and Anomaly Triage

A standardized deviation is then used to determine whether the observed error rate is anomalous. Welford-style online statistics are used in the implementation to update the rolling mean and variance without requiring the complete historical series. Only incidents satisfying the configured triage conditions are passed to the recovery workers.

The SDK hooks into application exception handling and transmits telemetry asynchronously. The control plane normalizes dynamic exception values, creates an MD5-based non-security fingerprint, and uses rolling EWMA/Welford statistics to escalate anomalous error rates.

$$\mu_t = \alpha x_t + (1 - \alpha) \mu_{t-1}.$$

B. Fault Localization

The worker checks out the failure-associated commit. The traceback supplies the candidate file and line; the AST parser identifies the enclosing function and gathers nearby statements, imports, exception information, and Git blame data to create a compact model context.

The patch engine asks a language model to return a minimal unified diff. Multiple providers are supported, including OpenAI, Groq/xAI-hosted models, Google Gemini, and Anthropic. Provider fallback is available when a configured provider is unavailable. Candidate responses are rejected unless they parse as a valid unified diff and apply cleanly to the checked-out repository. The same test configuration is then used for candidate validation.

C. Patch Synthesis

The patch engine requests a minimal unified diff from an LLM. OpenAI, Groq/xAI-hosted models, Gemini, and Anthropic are supported with provider fallback. Candidates must parse as valid diffs and apply cleanly before entering the common test configuration.

Every valid candidate is applied inside a fresh Docker container. The project dependencies are installed and the test suite is executed using pytest. The project configuration specifies a python:3.11-slim sandbox, network_mode=none, a 512 MB memory limit, 0.5 CPU allocation, and a 300-second timeout. These controls constrain resource consumption and network exposure while keeping the validation environment reproducible.

D. Sandbox Validation

Each candidate is applied in a fresh Docker container and tested with pytest. The configured python:3.11-slim sandbox uses network_mode=none, 512 MB memory, 0.5 CPU, and a 300-s timeout.

The mutation score M is interpreted as a measure of local test sensitivity, not as a proof of semantic correctness. Surviving mutants are retained as evidence because they identify behaviors in the changed region that the existing tests do not detect.

E. Mutation-Based Trust Evaluation

Candidates that pass ordinary tests enter the Trust Engine. It identifies changed lines and generates AST-level mutants only in the affected region using Boolean, comparison, boundary, and arithmetic operators. A mutant is killed if any test fails and survives if the full suite passes. The mutation score is

$$M = N_{killed} / N_{total}$$

and measures local test sensitivity.

The supplied materials do not report validated numerical weights; therefore, the manuscript does not infer or claim a reproducible weight setting. The evaluation instead compares the observed decisions produced by the configured Trust Engine.

$$T = w_m M + w_s (1 - S) + w_b (1 - B)$$

For accepted candidates, the GitHub integration creates a branch, commits the patch, opens a Pull Request containing the root-cause summary and evidence, and permits automatic merging only when repository CI and branch-protection rules succeed. Incident, patch, trust, and outcome information are embedded and stored with pgvector so that similar future incidents can retrieve prior recovery context; this is retrieval of historical context rather than model retraining.

F. Policy Decision and Recovery Memory

Automatic merge requires passing tests, $T \geq 0.90$, no protected-file violation, and blast radius within 0.3. A score below 0.70 routes the candidate to mandatory review; sensitive paths always require human review. Accepted candidates are committed to a branch and opened as Pull Requests, with automatic merge contingent on CI and branch protection. Recovery evidence is stored for historical retrieval rather than model retraining.

V. IMPLEMENTATION

A. System Components

The control plane uses Python, FastAPI, Uvicorn, and Pydantic; Celery/Redis provides asynchronous processing; PostgreSQL/SQLAlchemy/Alembic stores incidents, candidates, executions, trust decisions, Pull Requests, and audit records. Workers perform checkout, localization, synthesis, validation, mutation, and trust evaluation; Docker provides the execution boundary and GitHub App integration handles repository delivery.

Table V.I. Implementation stack and key configuration

Component	Technology / Configuration
Control plane	Python, FastAPI, Uvicorn, Pydantic v2
Asynchronous processing	Celery 5.x, Redis 7.x
Persistence	PostgreSQL 15, SQLAlchemy 2.0, Alembic
Recovery memory	pgvector HNSW, text-embedding-3-small
Patch execution	Docker, python:3.11-slim, network disabled
Frontend	React 18, TypeScript, Vite, TailwindCSS
Repository integration	GitHub App, REST API, Git CLI
Testing	pytest; 150 automated tests

B. Recovery State and Auditability

Incidents progress through explicit states from DETECTED and TRIAGED through LOCALIZED, PATCH_GENERATED, SANDBOX_RUNNING, TESTED, TRUST_EVALUATED, and finally AUTO_MERGE, HUMAN_REVIEW, REJECT, or MERGED. State transitions and evidence are persisted for audit reconstruction. A local in-process fallback is available when Celery is unavailable; SQLite is used for unit tests and PostgreSQL for deployment.

C. Recovery Memory

Completed recoveries are embedded with text-embedding-3-small and indexed by pgvector HNSW. Similar incidents can be retrieved as synthesis context; this is retrieval of prior evidence, not model retraining.

D. Validation and Security Boundaries

Generated patches and repository code are treated as untrusted material. Structural validation precedes execution; tests and mutation analysis run in isolated Docker environments, and protected paths are checked before autonomous merge. Patch generation, execution, trust evaluation, and repository modification remain separate control boundaries.

VI. EXPERIMENTAL SETUP

The evaluation has two layers: 150 automated tests covering unit, integration, functional, and repository-level recovery behaviour, and a 20-bug Python repair study comparing tests-only with tests-plus-trust acceptance. Both configurations use the same recovery pipeline; only the trust stage differs.

The repair evaluation contains 20 Python bugs covering representative failure classes, including missing-key or null-access errors, arithmetic errors, boundary errors, incorrect conditional logic, and unexpected input formats. The project evaluation materials describe the set as containing both documented defects from small open-source projects and controlled faults injected into otherwise functioning code. The available records do not provide a verified numerical split between these categories; therefore, no unverified dataset proportions are reported. Each bug is evaluated against its available project test suite, with correctness assessed against the intended repair rather than test-suite satisfaction alone. This distinction is necessary because the principal objective is to determine whether a patch that satisfies the existing tests is necessarily suitable for autonomous acceptance.

A. Evaluation Subjects

The repair evaluation contains 20 Python bugs covering representative failure classes, including missing-key or null-access errors, arithmetic errors, boundary errors, incorrect conditional logic, and unexpected input formats. The project evaluation materials describe the set as containing both documented defects from small open-source projects and controlled faults injected into otherwise functioning code. The available records do not provide a verified numerical split between these categories; therefore, no unverified dataset proportions are reported. Each bug is evaluated against its available project test suite, with correctness assessed against the intended repair rather than test-suite satisfaction alone. This distinction is necessary because the principal objective is to determine whether a patch that satisfies the existing tests is necessarily suitable for autonomous acceptance.

The repair set contains 20 Python bugs spanning missing-key/null access, arithmetic, boundary, conditional-logic, and unexpected-input failures. The materials describe both open-source defects and controlled injections but do not verify their numerical split. Each bug is assessed against its intended repair, not test-suite satisfaction alone.

B. Experimental Configurations

Tests-only accepts candidates after structural checks and project tests. Tests-plus-trust applies the same validation, then requires mutation-based trust evidence and governance checks before autonomous acceptance.

Table II. Comparison of evaluation configurations

Property	Tests Only	Tests + Trust
Evaluation bugs	20	20
Patch-generation pipeline	Same	Same
Structural patch validation	Yes	Yes
Isolated sandbox execution	Yes	Yes
Project test suite	Yes	Yes
Mutation analysis	No	Yes
Trust-score evaluation	No	Yes

Policy/governance checks	Baseline	Yes
Primary purpose	Test-based acceptance	Trust-enhanced acceptance

The records do not establish a verified number of generated candidates per bug; results therefore use outcome-level measurements rather than an unsupported candidate-count statistic.

The available project records do not establish a verified number of generated candidates per bug. The evaluation therefore reports outcome-level measurements rather than introducing an unsupported candidate-count statistic.

C. Evaluation Metrics

Metrics cover localization accuracy, plausible patch rate, correct patch rate, false acceptance, mutation score, and failure-to-ready-PR latency. Plausibility is whether at least one candidate passes available tests; correctness is semantic agreement with the intended fix; false acceptance counts incorrect repairs accepted despite passing tests.

$$\text{Localization Accuracy} = \frac{N_{\text{localized}}}{N_{\text{evaluated}}} \quad [1]$$

$$\text{Plausible Patch Rate} = \frac{N_{\text{plausible}}}{N_{\text{evaluated}}} \quad [2]$$

Correct patch rate measures the proportion of evaluated bugs for which the resulting repair is judged semantically correct against the intended fix:

$$\text{Correct Patch Rate} = \frac{N_{\text{correct}}}{N_{\text{evaluated}}} \quad [3]$$

False acceptance rate measures incorrect repairs that are accepted despite passing the available tests:

$$\text{FAR} = \frac{N_{\text{incorrect,accepted}}}{N_{\text{accepted}}} \quad [4]$$

For the central comparison, the study additionally reports the absolute number of incorrect patches accepted by the tests-only configuration and the corresponding number blocked after trust evaluation.

$$\text{MS} = \frac{N_{\text{killed}}}{N_{\text{mutants}}} \quad [5]$$

Mutation score is the proportion of generated mutants detected by the test suite and is used as local test-sensitivity evidence, not as proof of semantic correctness.

Recovery performance uses failure-to-ready-PR latency: elapsed time from failure detection to creation of the Pull Request. This avoids equating PR creation with production recovery or MTTR.

Independent of the 20-bug repair experiment, the implementation was exercised through 150 automated tests spanning unit, integration, functional, and repository-level recovery behaviour. These tests verify both normal operation and failure-handling paths.

D. System-Level Validation

Independent of the 20-bug repair experiment, the implementation was exercised through 150 automated tests spanning unit, integration, functional, and repository-level recovery behaviour. These tests verify both normal operation and failure-handling paths. Independent of the repair study, the implementation was exercised through 150 automated tests covering normal and failure-handling paths. Ten principal scenarios targeted trust determinism, patch safety, governance, anomaly handling, sandbox isolation, workspace safety, LLM resilience, patch validation, infrastructure fallback, and negative repair outcomes.

Table III. Principal system validation scenarios

Category	Validation Scenario	Expected Behaviour
Trust determinism	Repeated calculation with identical evidence	Identical trust result
Patch safety	Full-file rewrite	Candidate rejected
Governance	Sensitive authentication file	Human review enforced
Anomaly handling	Burst of runtime failures	Incident escalated
Sandbox isolation	Infinite-loop execution	Container terminated
Workspace safety	Temporary repository workspace	Workspace cleaned
LLM resilience	Malformed provider response	Candidate rejected without worker failure
Patch validation	Invalid or hallucinated diff context	Patch application failure
Infrastructure resilience	Celery unavailable	In-process fallback
Negative repair outcome	All candidates fail tests	Incident rejected and audited

The system tests demonstrate deterministic trust calculation, unsafe-patch rejection, sensitive-path governance, timeout enforcement, workspace cleanup, malformed-output handling, patch-application validation, worker fallback, and correct rejection when no candidate passes tests. These tests validate implementation reliability and are not combined numerically with the 20-bug repair evaluation.

The project supports multiple LLM providers, including OpenAI GPT-4o, Gemini, Anthropic Claude, and a Groq/xAI-hosted model configuration. Provider-dependent synthesis latency is reported separately; the evaluation does not treat model-provider variation as an independent controlled experiment.

E. Runtime and Execution Environment

These measurements characterize the implemented recovery pipeline rather than establishing a universal runtime bound. LLM response time depends on the selected provider, while sandbox and mutation execution depend on repository and test-suite characteristics.

Sandbox execution uses python:3.11-slim with disabled networking, 512 MB memory, 0.5 CPU, and a 300-s timeout. The stack uses FastAPI/Uvicorn/Pydantic, Celery/Redis, PostgreSQL, and pytest.

Supported providers include OpenAI GPT-4o, Gemini, Anthropic Claude, and a Groq/xAI-hosted configuration; provider latency is reported separately rather than treated as a controlled experimental factor.

These measurements characterize the implemented recovery pipeline rather than establishing a universal runtime bound. LLM response time depends on the selected provider, while sandbox and mutation execution depend on repository and test-suite characteristics.

F. Latency Measurement

Measured observations include 12–25 ms for ingestion, 80–150 ms for localization, 1.2–2.5 s for Groq/Grok synthesis, 4.5–8.0 s for GPT-4o synthesis, 3.0–8.5 s for sandbox/tests, and 6.0–15.0 s for ten sequential mutants. The reported detection-to-Pull-Request path is 15–35 s against a target below 60 s.

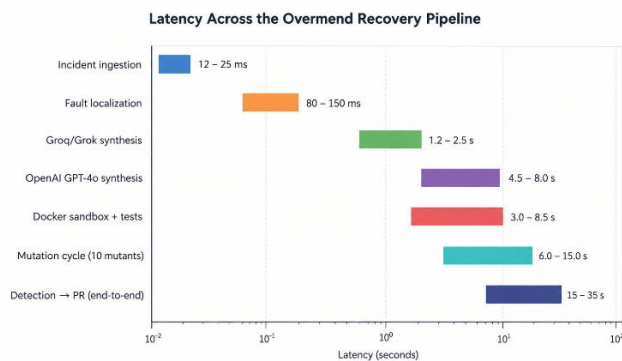


Fig. II. Measured latency across the principal stages of the Overmend recovery pipeline.

The principal repair comparison holds the evaluated bug set and patch-generation conditions constant while changing the acceptance mechanism from tests-only to tests-plus-trust. Candidate patches are processed through the same structural validation and isolated execution environment before the trust-specific stage is introduced.

G. Reproducibility and Experimental Controls

The repair comparison holds the bug set, prompts, candidate-generation process, structural validation, and sandbox environment constant while changing only the acceptance mechanism. Audit records capture validation, tests, mutation results, trust assessment, policy decisions, and Pull Request state.

The design distinguishes plausibility from correctness: passing tests show that exercised behaviours are satisfied, whereas semantic correctness is assessed independently. This distinction enables comparison of tests-only acceptance with mutation-informed decisions.

The default policy uses $T \geq 0.90$ for auto-merge, 0.70 for mandatory review, and a blast-radius limit of 0.3. Authentication, payment, migration, secret, Docker, and .env paths are protected from automatic merge.

VII. RESULTS AND DISCUSSION

A. RQ1 — Localization and Recovery Context

For RQ1, the implemented pipeline uses the runtime traceback to identify the candidate file and line, then narrows context to the enclosing AST function, nearby statements, imports, exception information, and repository history. The measured localization stage is 80–150 ms. This demonstrates focused recovery-context construction operationally, but no independent localization-accuracy benchmark was conducted; the study therefore does not claim a measured localization-accuracy rate.

B. RQ2 — False Acceptance

Across 20 bugs, tests-only accepted four incorrect patches (20% observed false acceptance). Trust evaluation blocked all four, reducing observed false acceptance to 0%; the overall fix-success rate remained 75%, indicating that the primary benefit is safer acceptance rather than greater patch-generation capability.

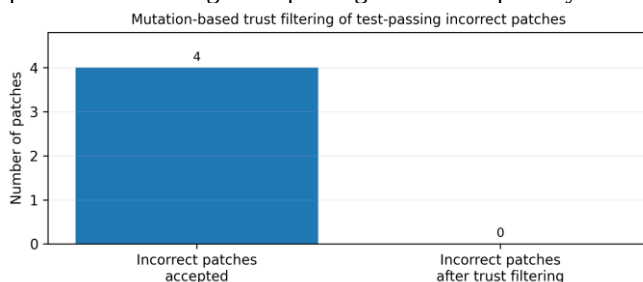


Fig. III. Reported detection of test-passing but incorrect patches. The trust-enabled configuration blocked all four incorrect patches accepted by the tests-only configuration.

The surviving mutants in these cases exposed changed regions that the tests did not exercise. Thus, mutation evidence supplied a useful acceptance signal, but the result is limited to the 20-bug controlled evaluation and does not establish that mutation testing eliminates incorrect repairs.

C. RQ3 — Latency Overhead

Median failure-to-ready-PR latency was 4.4 s with trust versus 1.1 s for tests-only; reported means were 481.3 s and 451.1 s. The gap indicates a long-tailed distribution, making the median more representative of typical execution while the mean exposes expensive outliers.

Table IV. Failure-to-ready-PR latency comparison.

Measure	Manual reference*	Tests only	Tests + trust
Median latency (s)	1800.0	1.1	4.4
Mean latency (s)	1800.0	451.1	481.3

*The 1800-s value is a contextual benchmark, not a measured human study.

Trust adds 3.3 s to the reported median and 30.2 s to the mean. Separately, the implementation reports a 15–35 s detection-to-Pull-Request path; this scope should not be conflated with the latency comparison in Table IV.

D. Autonomous Decision Outcomes

The trust-enabled Decision Engine produced 12 automatic merges (60%), 7 human-review outcomes (35%), and 1 rejection (5%). The distribution shows that technical plausibility does not override trust, blast-radius, or protected-file governance.

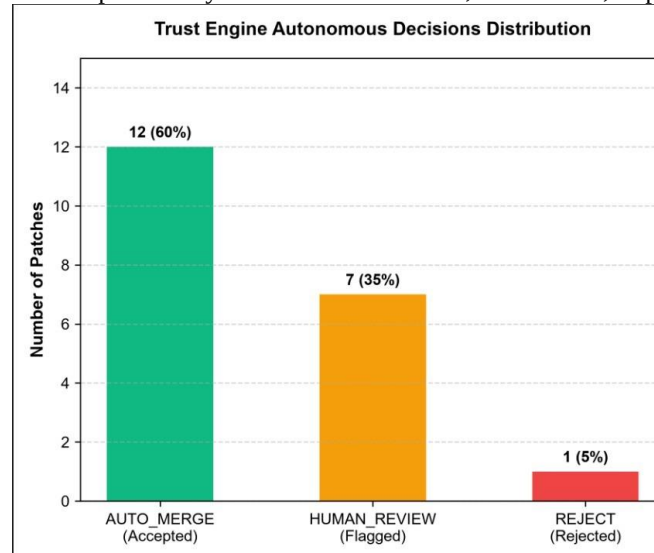


Fig. IV. Trust Engine Autonomous Decisions Distributions done by Overmend

E. Representative Recovery Case

A representative payment-webhook case involves a missing “stripe_signature” key raising KeyError; the SDK captures the traceback and commit, localization identifies the verification function, and candidate diffs are tested in separate containers.

F. Evidence and Auditability

Recovery decisions are auditable because incident states, patches, sandbox results, mutation reports, trust evaluations, decision rationale, and Pull Request information are stored with the recovery record.

VIII. LIMITATIONS AND THREATS TO VALIDITY

The evaluation is limited by its 20 Python bugs and the simpler nature of some injected defects, reducing statistical power and external validity. Several fields remain unspecified, including the

exact real/injected split, candidates per bug, illustrative mutation counts, and numerical localization results; these values are not inferred. The supplied records also do not verify the numerical trust weights; the manuscript therefore reports outcome-level decisions rather than reconstructing T from unstated parameters. A reproducibility artifact should expose the deployed weights and configuration.

Mutation score remains a proxy for local test sensitivity rather than proof of correctness; equivalent mutants, untested semantic properties, and defects outside the mutation operators can escape detection.

Latency depends on model response time, dependency installation, test duration, Docker startup, and mutant count; future studies should report distributions, percentiles, and confidence intervals.

The current release targets Python. Future work includes multi-language mutation, stronger post-merge rollback, larger public repair benchmarks, selective mutation, and security evaluation against the deployment threat model.

Additional threats concern construct and conclusion validity. The study measures false acceptance as an outcome-level property of the evaluated repair set, so the reported 20% and 0% rates should not be interpreted as population estimates. Semantic correctness is judged against the intended repair, but the available records do not provide an independently replicated adjudication protocol or inter-rater agreement. The comparison also isolates the acceptance mechanism while keeping the stated patch-generation and sandbox conditions constant; consequently, the results support a claim about the added trust stage under these conditions rather than about one model, one prompt, or one benchmark universally. Finally, mutation operators are restricted to the changed region and to selected Boolean, comparison, boundary, and arithmetic transformations. This design improves locality and cost control, but it can miss defects whose observable consequences occur outside the changed region or are not represented by those operators. These constraints motivate larger, independently adjudicated benchmarks and artifact-level reproducibility in future evaluation.

IX. CONCLUSION

Overmend combines runtime failure capture, fault localization, LLM patch synthesis, isolated execution, mutation-based validation, policy governance, and GitHub delivery. Its main contribution is to assess whether tests are sensitive to controlled faults around a repair rather than treating test passage as sufficient evidence.

In the 20-bug evaluation, four incorrect patches passed tests-only acceptance and all four were blocked by trust, while the 75% fix-success rate was retained. Median latency increased from 1.1 to 4.4 s, and the trust-enabled decision distribution was 12 auto-merges, 7 reviews, and 1 rejection.

The findings support mutation-based trust as a promising false-acceptance reduction mechanism, but the evidence is preliminary. Larger verified benchmarks, complete experimental reporting, confidence intervals, and multi-language evaluation are required before broader claims are made.

REFERENCES

- [1] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, pp. 56–65, 2019.
- [2] M. Monperrus, "Automatic software repair: A bibliography," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 1–24, 2018.
- [3] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? Overfitting in automated program repair," in *Proc. 10th Joint Meeting Found. Softw. Eng. (ESEC/FSE)*, 2015, pp. 532–543.
- [4] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proc. Int. Symp. Softw. Testing Anal. (ISSTA)*, 2015, pp. 24–36.
- [5] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proc. IEEE/ACM Int. Conf. Softw. Eng. (ICSE)*, 2023, pp. 1482–1494.
- [6] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "AutoCodeRover: Autonomous program improvement," in *Proc. ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA)*, 2024.
- [7] C. E. Jimenez et al., "SWE-bench: Can language models resolve real-world GitHub issues?" in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2024.
- [8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.
- [9] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, 2011.
- [10] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, 2014, pp. 654–665.
- [11] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, 2016.
- [12] M. Chen et al., "Evaluating large language models trained on code," *arXiv:2107.03374*, 2021.
- [13] J. Yang et al., "SWE-agent: Agent-computer interfaces enable automated software engineering," *arXiv:2405.15793*, 2024.