WCRG Global Research
WORLD COMET RESEARCH GROUP

International Journal of Data Engineering and Intelligent Computing
Vol. 9, No. 3, 2026
Doi: 10.67228/30715717/IJDEIC-V9I3P103
PP. 47-63

Original Article

Can LLM-Generated Backend Services Meet Performance Slos? A Controlled Study of Explicit Requirements and Measured-Feedback Repair

Priyank Agrawal

Master of Science in Information Systems, Northeastern University: Boston, Massachusetts, US.

Received: 07-08-2026 Revised: 07-29-2026 Accepted: 08-09-2026 Published: 08-12-2026

ABSTRACT

Production backend services must satisfy functional contracts and operational objectives simultaneously, yet most evaluations of code-generating language models still emphasize compilation, unit-test success, or repository issue resolution. This study asks whether a coding-agent workflow can generate small backend services that satisfy explicit service-level objectives (SLOs), and whether measured runtime feedback changes the result. We built SLOBench, a reproducible local harness that combines versioned prompts, deterministic functional validation, process-isolated FastAPI execution, concurrent HTTP load, CPU and memory sampling, provenance hashes, and implementation-level analysis. The controlled pilot evaluated three services, metadata lookup, a bounded cache-backed API, and concurrent aggregation, under three conditions: functional-only prompting, functional prompting plus an explicit SLO, and one-step repair after measured feedback. Five generation attempts per task produced 45 implementations; each implementation was measured three times, yielding 135 final runs. All implementations passed the functional oracle and every final load run had zero observed request errors. Adding SLO language alone reduced P95 latency by a median 2.2% across 15 paired generation-task comparisons (bootstrap 95% interval -2.7% to 10.7%; 9/15 improved). Measured-feedback repair improved all 15 pairs, with a median 16.8% reduction relative to the SLO-prompted version (95% interval 15.7% to 20.2%). The two strictest latency targets were nevertheless never reached. Source review further showed that all repaired implementations bypassed portions of the ordinary framework path and three hard-coded the disclosed hot request. Measured feedback therefore improved performance on the declared workload consistently, but the evidence does not establish production readiness. The findings motivate hidden acceptance workloads, calibrated baselines, stronger post-repair contract testing, and multi-model replication in performance-oriented coding-agent evaluation.

KEYWORDS

Large Language Models, Coding Agents, Software Performance, Service-Level Objectives, Backend Services, Execution Feedback, Benchmarking, Reproducibility.

ISSN: 3071-5717
PP: 47-63.

1. INTRODUCTION

Large language models (LLMs) are increasingly used to generate functions, patches, tests, and repository-level changes. Code-generation evaluation has consequently progressed from small executable programming tasks toward broader correctness checks, competitive programming benchmarks, and real-world issue resolution (Hendrycks et al., 2021; Li et al., 2022; Liu et al., 2023; Jimenez et al., 2024). Agent-oriented systems further show that tool access, execution, and interface design can materially affect software-engineering performance (Fan et al., 2023; Yang et al., 2024). These developments strengthen functional evaluation, but they do not remove a central production requirement: a service can return the correct response and still fail because it is too slow, unstable under concurrency, or operationally inefficient.

Operational quality is multidimensional. Tail latency can dominate user-visible performance even when median latency is low, and service behavior depends on workload intensity, software layers, scheduling, and resource contention (Dean & Barroso, 2013; Gan et al., 2019a). Cloud and microservice studies likewise show that quality-of-service violations can emerge from interactions across request paths and platform layers rather than from the application algorithm alone (Gan et al., 2019b; Leitner & Cito, 2016; Rzadca et al., 2020). For an HTTP service, an SLO is therefore meaningful only relative to a declared workload, environment, measurement protocol, and acceptance threshold.

Recent benchmarks have begun to evaluate efficiency in LLM-generated code rather than correctness alone. EffiBench, Mercury, ENAMEL, and ECCO measure runtime efficiency or correctness-efficiency trade-offs using executable tasks and controlled baselines (Du et al., 2024; Huang et al., 2024; Qiu et al., 2025; Waghjale et al., 2024). Other work shows that execution-derived feedback can guide models toward faster or more reliable solutions (Le et al., 2022; Madaan et al., 2023; Peng et al., 2025; Shinn et al., 2023; Zheng et al., 2024). However, most such evaluations remain function-level or algorithmic. Live backend services add framework routing, parsing, serialization, asynchronous scheduling, transport behavior, process lifecycle, and tail latency to the measurement object.

This paper presents SLOBench, a controlled local benchmark for evaluating LLM-generated FastAPI services under explicit operational targets. The study separates three interventions: (1) specifying only the functional contract, (2) adding explicit P95-latency, throughput, and error-rate targets, and (3) supplying measured diagnostic feedback for one repair step. It also separates independent generation attempts from repeated runtime measurements, an important distinction because repeated timing observations are not additional LLM samples. The design follows long-standing systems guidance that benchmark conclusions can be distorted by runtime state, ordering, environmental noise, and apparently minor setup choices (Curtsinger & Berger, 2013; Georges et al., 2007; Kalibera & Jones, 2013; Mytkowicz et al., 2009).

ISSN: 3071-5717
PP: 47-63.

The study makes four contributions:

- An executable benchmark design that couples functional oracles with P50/P95/P99 latency, throughput, error rate, CPU, and resident-memory measurements for generated backend services.
- A paired 3 × 3 × 5 experiment comprising 45 generated implementations and 135 final benchmark runs, with prompts, source hashes, platform metadata, and raw run records retained for audit.
- Evidence that measured feedback produced a consistent P95 improvement on the disclosed workload, whereas an explicit SLO prompt alone produced a small and inconsistent change.
- A qualitative mechanism audit showing that measured gains often came from lower-level response paths and, in three cases, exact specialization to the benchmark request, distinguishing benchmark success from production intent.

Because the pilot uses one coding-agent environment, three intentionally small services, five generation attempts, and one host, it does not claim that coding agents generally meet production SLOs. The narrower methodological question is whether a controlled live-service experiment can reveal a measurable difference between stating a performance requirement and closing the loop with runtime evidence.

2. RELATED WORK

2.1. Correctness-centered code-generation evaluation

Executable code benchmarks established test-based correctness as a core evaluation method for generated programs. APPS broadened natural-language-to-code evaluation to 10,000 programming problems, while AlphaCode demonstrated competitive-programming performance at larger scale (Hendrycks et al., 2021; Li et al., 2022). CodeGen explored open code models and multi-turn program synthesis, and CodeT showed that model-generated tests can improve the selection of candidate solutions (Chen et al., 2023; Nijkamp et al., 2023). EvalPlus subsequently demonstrated that weak test suites can overstate functional correctness and alter apparent model rankings (Liu et al., 2023). LiveCodeBench addresses contamination and saturation by continuously collecting newer problems and evaluating generation, execution, self-repair, and test-output prediction (Jain et al., 2025). These works motivate executable verification, stronger test coverage, and explicit attention to benchmark leakage.

2.2. Repository-level agents and feedback-driven refinement

Repository-level evaluation changes the unit of analysis from a short function to a software-maintenance task. SWE-bench evaluates whether models can resolve real GitHub issues, while SWE-agent shows that an agent-computer interface designed for navigation, editing, and execution can substantially affect outcomes (Jimenez et al., 2024; Yang et al., 2024). More generally, the software-engineering literature identifies LLM-based repair, refactoring, testing, and performance

ISSN: 3071-5717
PP: 47-63.

improvement as promising but reliability-sensitive applications (Fan et al., 2023). Feedback-based methods provide one route to higher reliability. CodeRL uses test and critic signals to guide generation, Self-Refine iterates on model-produced feedback, Reflexion incorporates verbal feedback into agent trajectories, and OpenCodeInterpreter combines execution with iterative code refinement (Le et al., 2022; Madaan et al., 2023; Shinn et al., 2023; Zheng et al., 2024). SLOBench adopts the closed-loop principle but uses measured service performance as the feedback signal.

2.3. Efficiency benchmarks and performance-oriented generation

Efficiency-oriented code benchmarks expose a gap between functional success and performant implementations. EffiBench evaluates efficiency-critical coding problems, Mercury models the runtime distribution of multiple acceptable solutions, ENAMEL introduces an efficiency-focused metric and expert-designed baselines, and ECCO studies whether efficiency can be improved without sacrificing correctness (Du et al., 2024; Huang et al., 2024; Qiu et al., 2025; Waghjale et al., 2024). Learning Performance-Improving Code Edits demonstrates that LLMs can learn higher-level optimizations from human edits under a controlled performance simulator, while PerfCodeGen uses execution feedback during self-refinement to improve runtime efficiency (Peng et al., 2025; Shypula et al., 2024). SLOBench extends this line of work from self-contained programs to a process-isolated HTTP service whose measured performance includes framework and transport overhead.

2.4. Benchmark validity for live services

Performance evaluation is unusually sensitive to experimental design. Repeated runs, warm-up behavior, nondeterminism, benchmark order, layout effects, and host variability can all change conclusions (Curtsinger & Berger, 2013; Georges et al., 2007; Kalibera & Jones, 2013; Mytkowicz et al., 2009). Cloud measurements add further variability from platform and resource-sharing effects (Leitner & Cito, 2016). At service scale, tail latency and microservice interactions make single central-tendency measures insufficient (Dean & Barroso, 2013; Gan et al., 2019a). SLOBench therefore uses fresh processes, randomized final-run order, multiple runtime repetitions, implementation-level aggregation, tail-latency reporting, and full provenance capture. The design also treats workload disclosure as a validity concern because optimization against a fully visible request can reward specialization rather than robust engineering, a concern aligned with broader calls for holistic, multi-metric evaluation (Bommasani et al., 2023).

3. RESEARCH QUESTIONS

- RQ1. Do generated services satisfy the functional contracts and remain error-free under the declared final workload?
- RQ2. Does adding an explicit SLO to the initial prompt improve P95 latency or SLO compliance relative to functional-only prompting?
- RQ3. Does a one-step repair using measured feedback improve P95 latency relative to the SLO-prompted implementation?

WCRG Global Research
ISSN: 3071-5717
PP: 47-63.

- RQ4. What implementation strategies account for measured improvements, and do those strategies suggest general optimization or benchmark specialization?

4. METHOD

4.1. Benchmark system

SLOBench is implemented in Python and runs entirely on one local machine. Each task provides a specification, three condition-specific prompt files, and a deterministic functional validator. Each candidate application is started with Uvicorn in a new subprocess on a free loopback port. After readiness and warm-up, an asynchronous HTTPX client launches a fixed number of workers. Each worker repeatedly issues the declared request until the measurement window closes. A psutil sampler records process CPU usage and resident set size (RSS) every 100 ms. The runner then evaluates task thresholds and writes one self-describing JSON record. This process-isolated design reduces cross-run state carryover and makes each final record traceable to a specific prompt and implementation hash.

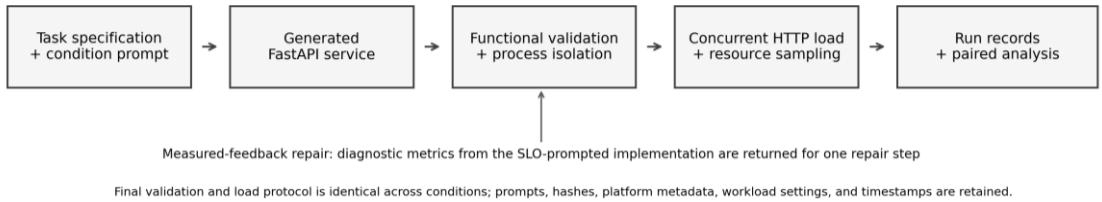

Fig 1 - Revised Slobench Execution and Feedback Pipeline. The Repair Condition Is Derived From the SLO-Prompted Implementation after a Short Diagnostic Measurement; All Three Conditions Then Enter the Same Final Validation and Load Protocol

4.2. Backend tasks and SLOs

The task suite was intentionally small and synthetic so the experiment could run without cloud infrastructure, external databases, or proprietary data. All logical records are generated deterministically from integer identifiers at runtime. The three tasks were selected to exercise distinct service behaviors while remaining simple enough for implementation-level mechanism auditing.

Table 1: Backend Tasks, Fixed Final Workloads, and Declared Slos

Task	Functional contract	Final hot request	Declared SLO
Metadata lookup	GET /metadata/{id}; 100,000 deterministic records; HTTP 404 outside range	GET /metadata/4242	P95 < 20 ms; ≥250 RPS; <1% errors
Cache API	GET /items/{id}; bounded, thread-safe LRU ≤1,000; GET /stats	GET /items/4242	P95 < 25 ms; ≥200 RPS; <1% errors
Aggregation	GET /aggregate?ids=…; 1–50 integers; return count, sum, and average	ids=1,2,3,4,5,6,7,8	P95 < 75 ms; ≥80 RPS; <1% errors

ISSN: 3071-5717
PP: 47-63.

4.3. Prompting conditions

Table 2: Controlled Prompting Conditions and Comparison Roles

Condition	Input to the coding agent	Role in comparison
Functional only	Task specification and instruction to meet the API contract	Baseline generation
Functional + SLO	Same functional contract plus task-specific P95, throughput, and error thresholds	Tests whether stating the objective is sufficient
Measured-feedback repair	Prior SLO implementation, valid diagnostic metrics, functional outcome, and the same thresholds	Tests one closed-loop repair step

The repair prompt requested a complete replacement app.py without changing the API or response contract. Feedback consisted of one valid 5 s diagnostic with a 1 s warm-up and 20 concurrent clients for each task-generation SLO implementation. Three initial diagnostic attempts in the first generation encountered a local process-permission error and were rerun successfully; those diagnostics were never included in the 135-run final dataset.

4.4. Experimental design and generation protocol

Five generation attempts were produced for each task. Within each attempt, the functional-only and functional-plus-SLO artifacts were generated from their respective prompts, and the repaired artifact was derived from that attempt's SLO version. This created 5 generations × 3 tasks × 3 conditions = 45 candidate implementations. The generation-task artifact, rather than an individual runtime repetition, is treated as the independent implementation-level observation. This avoids pseudoreplication from repeated measurements while acknowledging that artifacts produced in one coding-agent environment may still share latent dependencies.

The coding-agent environment was held constant during generation, but the exact hosted backend model build and sampling configuration were not exposed by the execution interface and therefore were not captured. The manuscript does not infer those missing details. This limitation prevents vendor- or model-specific claims and is carried through the external-validity discussion.

4.5. Functional validation

Before performance measurement, each application was imported in isolation with FastAPI's TestClient. The validator checked health, one representative successful response, and task-specific failure behavior. Cache validation additionally exercised repeated lookup and the statistics endpoint; aggregation validation included malformed input. A candidate with a functional failure would have been retained as a failed generation and excluded from load interpretation, but none of the 45 final artifacts failed this oracle. The oracle is intentionally a gate for the pilot rather than a claim of exhaustive contract coverage.

ISSN: 3071-5717
PP: 47-63.

4.6. Final load protocol and metrics

Each of the 45 implementations was measured three times. Every run launched a fresh service process, waited for health, warmed it for 3 s with up to four clients, and then measured for 20 s using 20 concurrent clients. The 135 task-condition-generation-repetition cells were executed in randomized order using seed 2026. The load generator used one asynchronous HTTP client and 20 closed-loop workers; each worker issued its next request after the prior response completed. Reported latency therefore includes loopback transport, HTTP client/server processing, routing, and serialization. Throughput and latency are coupled under this closed-loop design and should not be extrapolated to an open-loop arrival process.

For each run, SLOBench recorded request count, successful count, elapsed time, RPS, error rate, status and exception distributions, mean and P50/P95/P99 latency, mean process CPU, and peak RSS. A run passed only when all three task-specific thresholds were satisfied. Prompt and implementation SHA-256 hashes, Python version, platform, workload settings, and timestamps were stored with the metrics. Reporting P95 and P99 is particularly important because central latency can conceal service-level tail behavior (Dean & Barroso, 2013).

4.7. Statistical analysis

The primary analysis first took the median of the three final repetitions for each generated implementation. This choice follows systems-benchmarking guidance to distinguish measurement variation from treatment-level replication (Georges et al., 2007; Kalibera & Jones, 2013). For each of the 15 matched generation-task sets, percentage improvement was computed as (P95_baseline − P95_comparison) / P95_baseline × 100. We report the median paired improvement, the number of positive pairs, and a nonparametric bootstrap 95% interval for the median using 20,000 resamples and seed 2026. Cell summaries use the median across five implementation medians. SLO pass rate is reported at run level (15 runs per task-condition cell) because threshold compliance is a property of each measured run. Given the exploratory sample size and within-generation dependence, no null-hypothesis significance tests were performed.

4.8. Execution environment

Final measurements ran on a MacBook Air with an Apple M5 processor (10 CPU cores: 4 performance and 6 efficiency) and 24 GB memory, using macOS 26.5.1 on ARM64 and Python 3.12.14. The pinned software stack was FastAPI 0.115.6, Uvicorn 0.34.0, HTTPX 0.28.1, psutil 6.1.1, and pytest 8.3.4. No Docker resource limits were used for the reported dataset. The host is treated as a fixed experimental platform; absolute latency should not be directly compared with measurements collected on different hardware or under different background-load conditions (Leitner & Cito, 2016).

ISSN: 3071-5717
PP: 47-63.

5. Results

5.1. Dataset completeness and functional validity

The final directory contained 135 records, and all 135 contained metrics. Every task-condition cell had 15 runs: five generated implementations measured three times. All 45 implementations passed functional validation, every final HTTP load run recorded a 0% error rate, and implementation hashes were stable across repetitions. No final run had a fatal error.

RQ1 answer: Under the tested requests, the generated services satisfied the limited functional oracle and served the final workload without observed HTTP errors. This establishes validity for the measured pilot paths, not general correctness across the full API input domain.

5.2. P95 latency, throughput, and SLO compliance

Table 3: Final Cell Summaries. P95, Throughput, and RSS are Medians across Five Implementation-Level Medians; Brackets Show Bootstrap 95% Intervals for Median P95. SLO Pass Is the Proportion of 15 Final Runtime Repetitions Satisfying All Thresholds

Task	Condition	P95 ms [95% interval]	RPS	Peak RSS (MB)	SLO pass
Metadata lookup	Functional only	74.8 [69.2, 82.4]	974.4	44.7	0.0%
Metadata lookup	Functional + SLO	73.3 [70.8, 74.1]	1011.3	44.6	0.0%
Metadata lookup	Measured-feedback repair	61.9 [57.7, 66.7]	1030.4	44.6	0.0%
Cache API	Functional only	76.0 [70.1, 82.2]	951.1	44.7	0.0%
Cache API	Functional + SLO	72.0 [68.7, 74.4]	1020.7	44.6	0.0%
Cache API	Measured-feedback repair	59.4 [52.0, 60.7]	1031.7	44.6	0.0%
Aggregation	Functional only	74.6 [72.6, 82.7]	974.7	44.7	53.3%
Aggregation	Functional + SLO	74.0 [72.9, 76.9]	985.8	44.7	46.7%
Aggregation	Measured-feedback repair	61.5 [57.5, 62.8]	1010.3	44.6	100.0%

ISSN: 3071-5717
PP: 47-63.

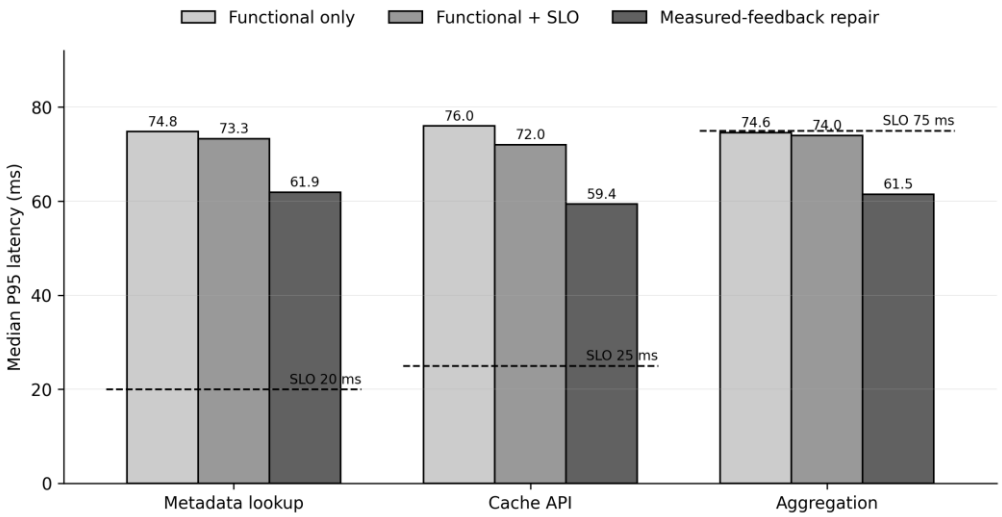

Fig 2 - Revised Median Implementation-Level P95 Latency By Task And Prompting Condition. Each Bar Summarizes Five Generated Implementations After Taking The Median Of Three Runs Per Implementation. Dashed Task-Local Segments Denote the Declared P95 Thresholds

Metadata and cache services exceeded their P95 targets in every final run, even after repair. Aggregation was close to its 75 ms limit before repair and passed 53.3% of functional-only runs and 46.7% of SLO-prompted runs. All 15 repaired aggregation runs passed. Throughput exceeded every task's minimum threshold by a wide margin, so P95 latency was the binding criterion whenever a run failed.

5.3. Paired effects of prompting and measured repair

Table 4: Paired Percentage Changes in Implementation-Level P95 Latency. Positive Values Indicate Lower Latency in the Comparison Condition

Comparison	Pairs	Median improvement	Bootstrap 95% interval	Improved pairs
Functional + SLO vs functional only	15	2.2%	[-2.7%, 10.7%]	9/15
Measured repair vs functional + SLO	15	16.8%	[15.7%, 20.2%]	15/15
Measured repair vs functional only	15	19.1%	[14.5%, 27.4%]	15/15

ISSN: 3071-5717
PP: 47-63.

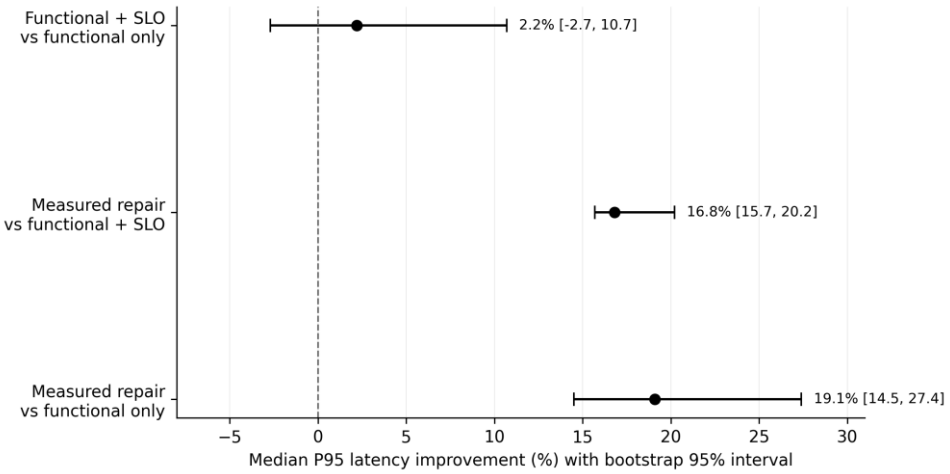

Fig 3 - Revised Paired-Effect Summary for Implementation-Level P95 Latency. Points Show the Median Percentage Improvement and Horizontal Bars Show the Bootstrap 95% Interval for Each Comparison. Values To the Right of Zero Favor the Comparison Condition

RQ2 answer: Explicit SLO wording alone had a small, inconsistent effect: median P95 improvement was 2.2%, the bootstrap interval crossed zero, and only 9 of 15 pairs improved. It did not improve overall aggregation SLO compliance.

RQ3 answer: Measured-feedback repair improved P95 in all 15 matched pairs. The median improvement over the SLO-prompted artifact was 16.8% (95% interval 15.7% to 20.2%); relative to functional-only code, the median was 19.1% (95% interval 14.5% to 27.4%). The improvement was sufficient for universal aggregation compliance but not for the stricter metadata or cache latency targets.

5.4. Tail shape, resource use, and binding constraints

Across the nine task-condition cells, median P50 ranged from 8.6 to 9.3 ms, whereas median P99 ranged from 108.5 to 136.8 ms. The wide separation between central and tail latency indicates that P95 failure was driven more by intermittent delay than by the typical request, a pattern consistent with why tail percentiles matter for service-level analysis (Dean & Barroso, 2013). Cell-median throughput ranged from 951.1 to 1031.7 RPS, and peak RSS medians ranged only from 44.6 to 44.7 MB. Error rate was zero in every cell. On this host, the operative failure mode was therefore tail latency rather than basic capacity, memory consumption, or observed request correctness.

5.5. Qualitative audit of repaired implementations

The 15 repaired source files were manually reviewed to identify the mechanism behind the measured change. Every repair reduced or bypassed part of FastAPI's ordinary dependency-resolution, routing, or JSON-serialization path by using direct ASGI handling, Starlette routes with pre-encoded responses, custom routers, or cached response bodies. Three repairs, the first generation

ISSN: 3071-5717
PP: 47-63.

for each task, contained exact hot-path checks for the disclosed ID or aggregation query and returned precomputed bytes. Other repairs generalized the lower-level path to arbitrary valid IDs or queries, although repeated-workload caching still benefits the declared request distribution.

Table 5: Qualitative Implementation Patterns in the 15 Repaired Artifacts

Pattern	Observed count	Interpretation
Lower-level ASGI/response path	15/15	Reduced framework routing, validation, or serialization overhead on the measured path
Exact disclosed request hard-coded	3/15	Clear benchmark specialization; no comparable benefit is expected for shifted requests
General custom route/router	12/15	Optimization applies across a broader valid input class, although contract coverage remained lightly tested

RQ4 answer: The measured gain was largely explained by moving below FastAPI's conventional endpoint machinery rather than by changing the small task algorithms. Some repairs were general framework-level optimizations; three were direct workload memorization. The final workload cannot distinguish robust engineering from benchmark specialization, so the repaired latency result should be interpreted as performance on the declared request, not as proof of generalized service quality.

6. DISCUSSION

6.1. Why an explicit SLO prompt was insufficient

A numeric performance target tells an agent what outcome is desired but does not reveal which layer dominates latency on a particular host. The functional and SLO-prompted services often used similar FastAPI abstractions, and both easily exceeded throughput requirements. Without measurement, the model had little evidence that routing, response construction, or serialization overhead was more important than data-structure changes. The small median effect and mixed pair directions are therefore consistent with an under-specified optimization problem. This finding complements efficiency benchmarks showing that correct code generation and performance optimization remain distinct capabilities (Du et al., 2024; Huang et al., 2024; Qiu et al., 2025).

6.2. What measured feedback changed

A scalar failure report made the performance gap concrete and triggered a qualitatively different class of edits. Every repaired artifact reduced P95 relative to its own SLO version, which aligns with prior evidence that execution feedback can improve code-selection or optimization behavior (Chen et al., 2023; Peng et al., 2025; Zheng et al., 2024). At the same time, P50 changed little compared with the P95 shift. One plausible explanation is that lower-level response paths reduced enough per-request framework work or queueing pressure to compress the tail. Because SLOBench did not collect event-loop traces or per-layer profiles, this remains an inference from source review

ISSN: 3071-5717
PP: 47-63.

and aggregate metrics rather than a causal decomposition. Future variants could pair the harness with tracing or performance-debugging techniques similar in spirit to Seer (Gan et al., 2019b).

6.3. Meeting a benchmark is not meeting production intent

The repair condition exposes an optimizer's loophole: when the exact workload is fully disclosed, the easiest route to a lower score may be to specialize to that workload. Hard-coding the hot ID was not prohibited by the functional tests, and direct routers were only lightly tested outside the measured path. Such code can satisfy the literal benchmark while weakening maintainability, observability, framework guarantees, or behavior on unseen inputs. The issue parallels broader concerns about benchmark overfitting, contamination, and insufficient test coverage in code-generation evaluation (Jain et al., 2025; Liu et al., 2023). A production-oriented benchmark should therefore separate a visible optimization workload from hidden acceptance workloads and revalidate contract breadth after every performance edit.

6.4. Implications for coding-agent evaluation

The pilot suggests several design requirements for future evaluations:

- Calibrate each SLO against an expert, hand-optimized, or minimal-framework baseline on the same host before asking whether generated code can meet it.
- Keep a performance-feedback workload visible to support optimization, but evaluate generalization on hidden IDs, mixed request distributions, concurrency shifts, and adverse inputs.
- Re-run the full functional and security contract after optimization, especially when fast paths bypass framework-level validation or serialization behavior.
- Record the exact model identifier, version, agent scaffold, sampling settings, token usage, and generation transcript whenever the execution interface exposes them.
- Use multiple models, more independent generation attempts, and hierarchical analysis that respects dependencies among artifacts produced in one agent environment.
- Measure operational dimensions beyond steady-state closed-loop latency, including cold start, sustained load, open-loop overload, CPU saturation, memory growth, multi-worker behavior, and observability overhead.

These recommendations also support a broader view of LLM evaluation in which a single score is insufficient. Holistic evaluation argues for explicit scenarios and multiple metrics rather than treating one capability as a proxy for overall quality (Bommasani et al., 2023). For generated services, functional correctness, latency, throughput, resource use, robustness, security, and maintainability should be reported as related but non-interchangeable dimensions.

ISSN: 3071-5717
PP: 47-63.

7. THREATS TO VALIDITY

7.1. Construct validity

The three services are intentionally small and use deterministic synthetic data. They do not represent database access, remote dependencies, authentication, persistence, large payloads, or heterogeneous production traffic. The functional oracle samples a limited set of inputs. The repeated hot request rewards caching and specialization. The SLOs were selected as experimental targets but were not calibrated against a lower-bound expert baseline on the host; the 20 ms and 25 ms targets may have been infeasible for the chosen client/server stack under 20 closed-loop clients. A future design should establish feasible target bands through reference implementations before model evaluation.

7.2. Internal validity

Randomized order, warm-up, fresh subprocesses, and stable code hashes reduce several sources of bias, but the host was not dedicated laboratory hardware and background activity was not recorded. The load generator and service shared one machine, so client scheduling competed with server scheduling. CPU percentage sampled only the service process, not total system load. These caveats matter because systems experiments can produce misleading differences when environmental variation or setup effects are not controlled (Curtsinger & Berger, 2013; Mytkowicz et al., 2009). The repair prompt also exposed the exact measured request and thresholds, making benchmark awareness part of the treatment. Three early permission failures occurred only in diagnostic runs and were rerun; the final dataset itself is complete.

7.3. External validity

Only one coding-agent environment, one Python/FastAPI stack, one ARM64 Mac, and five generation attempts were studied. The hosted model build was not exposed. Results cannot be generalized to other models, sampling settings, languages, frameworks, operating systems, cloud deployments, or service architectures. Public-cloud and microservice research further demonstrates that platform variation and service composition can materially affect performance behavior (Gan et al., 2019a; Leitner & Cito, 2016). Multi-host and multi-model replication is therefore required before making claims about coding agents as a class.

7.4. Statistical conclusion validity

The bootstrap interval treats 15 generation-task pairs as observational units even though three tasks within a generation may be dependent. Runtime repetitions quantify measurement variability but do not enlarge the independent generation sample. The intervals are descriptive rather than confirmatory, and no preregistered hypothesis test or multiple-model replication was conducted. Reporting pair directions, raw cell outcomes, and implementation-level aggregation is intended to make the limited evidential strength visible. Larger future studies should use more independent generations and a hierarchical model that separates task, generation, model, and runtime variance.

ISSN: 3071-5717
PP: 47-63.

8. REPRODUCIBILITY AND PUBLIC ARTIFACT

The complete sanitized study artifact is publicly available in the SLOBench repository at https://github.com/agrawal-priyank/slobench. It includes task specifications, condition prompts, 45 generated app.py artifacts, functional-validation outputs, 135 final JSON records, aggregation and analysis code, pinned dependencies, documentation, and an MIT license. Each final record binds the prompt and implementation through SHA-256 hashes, and the public manifest supplies checksums for exported JSON files. Synthetic task data are generated at runtime, so no external dataset is required.

The analyzed public snapshot is commit 3d9ba40. The export process replaced machine-specific absolute paths with repository-relative provenance paths while retaining the measurements, hashes, run identifiers, timestamps, workload settings, and platform metadata used in the analysis. A DOI-backed archive of the frozen submission version should be deposited before journal submission so that the cited artifact is immutable and independently retrievable. Persistent, version-specific software citation supports reproducibility and scholarly attribution (Smith et al., 2016).

A minimal reproduction requires Python 3.12, installation of the pinned requirements, functional validation, and execution of the benchmark runner with three repetitions, a 20 s duration, 3 s warm-up, 20 clients, and seed 2026. Absolute latency should be re-baselined for each new host; relative comparisons should be made within the same randomized experiment.

9. CONCLUSION

This controlled pilot separates three ideas that are often conflated: generating functionally correct service code, stating a performance objective, and using measurements to revise code. Across 135 final runs, functional correctness and error-free load behavior did not imply latency compliance. Explicit SLO wording alone produced a small and inconsistent P95 change. One-step measured-feedback repair reduced P95 for every matched generation-task pair, with a median 16.8% improvement over the SLO-prompted implementation, but only the aggregation task consistently crossed its latency target.

The mechanism matters as much as the score. Repairs consistently bypassed higher-level framework paths, and three exactly specialized to the disclosed request. The evidence therefore supports a narrow conclusion: measured feedback can guide a coding agent toward materially faster behavior on a declared backend workload. It does not support the broader claim that the resulting services meet production SLOs. Hidden acceptance workloads, stronger post-repair contracts, calibrated targets, deeper profiling, and multi-model replication are the necessary next steps for a benchmark that measures robust performance engineering rather than optimization to a visible test.

ISSN: 3071-5717
PP: 47-63.

REFERENCES

[1] Bommasani, R., Liang, P., & Lee, T. (2023). Holistic evaluation of language models. Annals of the New York Academy of Sciences, 1525(1), 140–146. https://doi.org/10.1111/nyas.15007

[2] Chen, B., Zhang, F., Nguyen, A., Zan, D., Lin, Z., Lou, J.-G., & Chen, W. (2023). CodeT: Code generation with generated tests. International Conference on Learning Representations. https://openreview.net/forum?id=ktrw68Cmu9c

[3] Curtsinger, C., & Berger, E. D. (2013). STABILIZER: Statistically sound performance evaluation. In Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (pp. 219–228). Association for Computing Machinery. https://doi.org/10.1145/2451116.2451141

[4] Dean, J., & Barroso, L. A. (2013). The tail at scale. Communications of the ACM, 56(2), 74–80. https://doi.org/10.1145/2408776.2408794

[5] Du, M., Tuan, L. A., Ji, B., Liu, Q., & Ng, S.-K. (2024). Mercury: A code efficiency benchmark for code large language models. Advances in Neural Information Processing Systems, 37, 16601–16622. https://doi.org/10.52202/079017-0529

[6] Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., & Zhang, J. M. (2023). Large language models for software engineering: Survey and open problems. In 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE) (pp. 31–53). IEEE. https://doi.org/10.1109/ICSE-FoSE59343.2023.00008

[7] Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B., Hu, K., Pancholi, M., He, Y., Clancy, B., Colen, C., Wen, F., Leung, C., Wang, S., Zaruvinsky, L., ... Delimitrou, C. (2019a). An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (pp. 3–18). Association for Computing Machinery. https://doi.org/10.1145/3297858.3304013

[8] Gan, Y., Zhang, Y., Hu, K., Cheng, D., He, Y., Pancholi, M., & Delimitrou, C. (2019b). Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (pp. 19–33). Association for Computing Machinery. https://doi.org/10.1145/3297858.3304004

[9] Georges, A., Buytaert, D., & Eeckhout, L. (2007). Statistically rigorous Java performance evaluation. In Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (pp. 57–76). Association for Computing Machinery. https://doi.org/10.1145/1297027.1297033

[10] Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., & Steinhardt, J. (2021). Measuring coding challenge competence with APPS. Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks, 1.

[11] Huang, D., Qing, Y., Shang, W., Cui, H., & Zhang, J. M. (2024). EffiBench: Benchmarking the efficiency of automatically generated code. Advances in Neural Information Processing Systems, 37. https://doi.org/10.52202/079017-0367

[12] Jain, N., Han, K., Gu, A., Li, W.-D., Yan, F., Zhang, T., Wang, S., Solar-Lezama, A., Sen, K., & Stoica, I. (2025). LiveCodeBench: Holistic and contamination free evaluation of large language models for code. International Conference on Learning Representations. https://proceedings.iclr.cc/paper_files/paper/2025/hash/94074dd5a072d28ff75a76dabed43767-Abstract-Conference.html

[13] Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. R. (2024). SWE-bench: Can language models resolve real-world GitHub issues? International Conference on Learning Representations. https://openreview.net/forum?id=VTF8yNQM66

[14] Kalibera, T., & Jones, R. E. (2013). Rigorous benchmarking in reasonable time. In Proceedings of the 2013 International Symposium on Memory Management (pp. 63–74). Association for Computing Machinery. https://doi.org/10.1145/2464157.2464160

[15] Le, H., Wang, Y., Gotmare, A. D., Savarese, S., & Hoi, S. C. H. (2022). CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. Advances in Neural Information Processing Systems, 35, 21314–21328. https://doi.org/10.52202/068431-1549

ISSN: 3071-5717
PP: 47-63.

[16] Leitner, P., & Cito, J. (2016). Patterns in the chaos: A study of performance variation and predictability in public IaaS clouds. ACM Transactions on Internet Technology, 16(3), 1–23. https://doi.org/10.1145/2885497

[17] Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., Hubert, T., Choy, P., de Masson d'Autume, C., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Cherepanov, A., ... Vinyals, O. (2022). Competition-level code generation with AlphaCode. Science, 378(6624), 1092–1097. https://doi.org/10.1126/science.abq1158

[18] Liu, J., Xia, C. S., Wang, Y., & Zhang, L. (2023). Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. Advances in Neural Information Processing Systems, 36, 21558–21572.

[19] Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., Gupta, S., Majumder, B. P., Hermann, K., Welleck, S., Yazdanbakhsh, A., & Clark, P. (2023). Self-Refine: Iterative refinement with self-feedback. Advances in Neural Information Processing Systems, 36. https://doi.org/10.52202/075280-2019

[20] Mytkowicz, T., Diwan, A., Hauswirth, M., & Sweeney, P. F. (2009). Producing wrong data without doing anything obviously wrong! In Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (pp. 265–276). Association for Computing Machinery. https://doi.org/10.1145/1508244.1508275

[21] Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., & Xiong, C. (2023). CodeGen: An open large language model for code with multi-turn program synthesis. International Conference on Learning Representations. https://openreview.net/forum?id=iaYcJKpY2B_

[22] Peng, Y., Gotmare, A. D., Lyu, M., Xiong, C., Savarese, S., & Sahoo, D. (2025). PerfCodeGen: Improving performance of LLM generated code with execution feedback. In 2025 IEEE/ACM 2nd International Conference on AI Foundation Models and Software Engineering (FORGE) (pp. 1–13). IEEE. https://doi.org/10.1109/FORGE66646.2025.00008

[23] Qiu, R., Zeng, W. W., Ezick, J., Lott, C., & Tong, H. (2025). How efficient is LLM-generated code? A rigorous and high-standard benchmark. International Conference on Learning Representations. https://proceedings.iclr.cc/paper_files/paper/2025/hash/06694da057cb15fef11542270a592627-Abstract-Conference.html

[24] Rzadca, K., Findeisen, P., Swiderski, J., Zych, P., Broniek, P., Kusmierek, J., Nowak, P., Strack, B., Witusowski, P., Hand, S., & Wilkes, J. (2020). Autopilot: Workload autoscaling at Google. In Proceedings of the Fifteenth European Conference on Computer Systems (Article 16, pp. 1–16). Association for Computing Machinery. https://doi.org/10.1145/3342195.3387524

[25] Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., & Yao, S. (2023). Reflexion: Language agents with verbal reinforcement learning. Advances in Neural Information Processing Systems, 36.

[26] Shypula, A., Madaan, A., Zeng, Y., Alon, U., Gardner, J., Yang, Y., Hashemi, M., Neubig, G., Ranganathan, P., Bastani, O., & Yazdanbakhsh, A. (2024). Learning performance-improving code edits. International Conference on Learning Representations. https://proceedings.iclr.cc/paper_files/paper/2024/hash/ce65173b994cf7c925c71b482ee14a8d-Abstract-Conference.html

[27] Smith, A. M., Katz, D. S., Niemeyer, K. E., & FORCE11 Software Citation Working Group. (2016). Software citation principles. PeerJ Computer Science, 2, e86. https://doi.org/10.7717/peerj-cs.86

[28] Waghjale, S., Veerendranath, V., Wang, Z. Z., & Fried, D. (2024). ECCO: Can we improve model-generated code efficiency without sacrificing functional correctness? In Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (pp. 15362–15376). Association for Computational Linguistics. https://doi.org/10.18653/v1/2024.emnlp-main.859

[29] Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., & Press, O. (2024). SWE-agent: Agent-computer interfaces enable automated software engineering. Advances in Neural Information Processing Systems, 37. https://doi.org/10.52202/079017-1601

[30] Zheng, T., Zhang, G., Shen, T., Liu, X., Lin, B. Y., Fu, J., Chen, W., & Yue, X. (2024). OpenCodeInterpreter: Integrating code generation with execution and refinement. In Findings of the Association for Computational Linguistics: ACL 2024 (pp. 12834–12859). Association for Computational Linguistics. https://doi.org/10.18653/v1/2024.findings-acl.762

ISSN: 3071-5717
PP: 47-63.

[31] Begimher, D., Leo, C., Huang, J., Gaw, P., & Zheng, B. (2026). SIR-Bench: Evaluating Investigation Depth in Security Incident Response Agents. *arXiv preprint arXiv:2604.12040*.

[32] Leo, C., Dykyi, A., Cortegaca, D., Begimher, D., & Jha, P. (2026). ThreatForest: Multi-Agent Attack Tree Generation with Pluggable TTP Framework Mapping. *arXiv preprint arXiv:2607.27528*.