Can LLMs Diagnose Performance Bottlenecks from Test Metrics? A Grey-Box Evaluation

Miguel Angelo Bastos Muniz
LESSE/UNIPAMPA
Alegrete, RS, Brazil
miguelmuniz.aluno@unipampa.edu.br

Eduardo Prates Tiadoro
LESSE/UNIPAMPA
Alegrete, RS, Brazil
eduardotiadoro.aluno@unipampa.edu.br

Maicon Bernardino
LESSE/PPGES/UNIPAMPA
Alegrete, RS, Brazil
bernardino@acm.org

Anielle Andrade
LESSE/PPGES/UNIPAMPA
Alegrete, RS, Brazil
anielleandrade@unipampa.edu.br

Elder de Macedo Rodrigues
LESSE/PPGES/UNIPAMPA
Alegrete, RS, Brazil
elderrodrigues@unipampa.edu.br

Abstract

Software performance diagnosis requires engineers to correlate load-test metrics with system telemetry and formulate hypotheses about architectural bottlenecks. This paper investigates whether Large Language Models (LLMs) can support this task without source-code access. We conducted an exploratory Grey-Box evaluation using a Node.js API with two deliberately injected performance problems: simulated N+1 data access and a Memory Leak. Consolidated K6 metrics and server telemetry were analyzed by Gemini 3.5 Flash, Claude Sonnet 4.6, and GPT-5.5 Instant using Zero-Shot Prompting. Three domain experts independently evaluated bottleneck-class diagnostic precision, recommendation actionability, and unsupported-claim severity. Results show scenario-dependent behavior. Memory Leak diagnosis received higher scores when monotonic Heap growth provided a highly discriminative signal, whereas aggregate latency evidence in the N+1 scenario produced broader and less precise diagnoses. Some assertive responses also introduced unsupported causal claims. These findings suggest that LLMs can assist post-test performance diagnosis, but their reliability depends on the diagnostic specificity of the available telemetry and requires human validation.

Keywords

Performance Testing, Root Cause Analysis, Large Language Models, Software Engineering, Node.js.

1 Introduction

The growing digitalization of services and the widespread adoption of distributed architectures have made performance one of the key quality attributes of modern software systems. Web applications, APIs, and digital platforms must meet increasingly stringent scalability, availability, and responsiveness requirements, as performance degradations can compromise user experience, increase operational costs, and directly impact business objectives [6, 14]. In this context, performance testing plays a fundamental role by enabling the early identification of architectural bottlenecks and infrastructure limitations before production deployment [16].

Specialized tools such as K6 (Grafana Labs [4]) enable load tests that reproduce realistic usage scenarios and generate detailed behavioral metrics. However, although data collection has evolved significantly, interpreting results remains a complex activity dependent on specialized knowledge. Analyzing metrics such as latency, throughput, error rates, and response-time percentiles requires correlating different information sources to formulate and validate diagnostic hypotheses [2, 12].

Recent advances in Generative AI have driven the emergence of LLMs capable of interpreting natural language, source code, structured data, and technical documentation [1, 17]. These models have been applied across many Software Engineering activities, including code generation, artifact review, automatic documentation, and incident resolution support [5]. Their capacity to correlate information from different sources suggests a promising potential for assisting engineers in analyzing performance test results.

Despite this progress, prior research has largely focused on automated test generation and RCA from cloud incidents or software bug reports. Recent studies investigate LLM-based RCA from operational diagnostic data [2] and the generation of root-cause explanations from software bug reports [12], while the use of LLMs as assistants for interpreting post-test performance evidence remains less explored. Recent work on Generative AI in performance testing also motivates empirical investigation beyond test preparation [18].

Given this gap, this work investigates the capability of Large Language Models to diagnose performance problems from metrics produced during load testing. To this end, a controlled experimental environment was developed containing a Node.js API with deliberately inserted architectural bottlenecks based on well-known anti-patterns. The system was subjected to load tests using the K6 tool, generating consolidated performance metrics that were subsequently analyzed by different LLMs through the Zero-Shot Prompting technique [9]. Accordingly, the study addresses the following research question (RQ) (Figure 1).

RQ. *To what extent can Large Language Models, operating under a Grey-Box setting without source-code access, infer the likely class of performance bottlenecks from consolidated load-test metrics and server telemetry, while producing actionable and evidence-grounded diagnostic hypotheses?*

Figure 1: Research question

In this study, we distinguish bottleneck-class diagnosis from concrete code-level root-cause localization. Because the evaluated

models do not receive source code, they cannot directly identify the exact statement, variable, or implementation decision responsible for the degradation. Instead, the experiment evaluates whether LLMs can infer the likely architectural class of the bottleneck from performance evidence and provide technically actionable hypotheses about what developers should investigate. Accordingly, RCA is examined here as a diagnostic support activity rather than as definitive code-level fault localization.

This study contributes: **(i)** an exploratory Grey-Box evaluation of LLM-assisted post-test performance diagnosis; **(ii)** a structured expert rubric separating diagnostic precision, recommendation actionability, and unsupported-claim severity; and **(iii)** evidence that diagnostic performance varies with the specificity of available observability signals.

2 Theoretical Background

2.1 Performance Testing and Bottleneck Analysis

Load tests subject the system to simulated concurrent Virtual Users (VUs), measuring behavior under normal and peak conditions [13]. Response time is evaluated not only by its mean, but also by its statistical distribution in percentiles such as *p90* and *p95*. These percentiles characterize tail latency across requests and are commonly used in performance objectives [3].

Performance-threshold violations may stem from inefficient implementation or data-access patterns. A classic example is the N+1 Queries pattern, where a system executes multiple iterative and redundant queries instead of a single optimized request, amplifying data-access latency and resource demand [8]. Root Cause Analysis (RCA) seeks to explain observed failures or degradations by identifying their underlying causal mechanisms.

2.2 LLMs and Prompt Engineering

LLMs are based on deep neural network architectures (Transformers) and are trained on massive data volumes, developing capabilities to interpret source code, structured data, and perform complex logical inferences [1, 17]. Their effectiveness on specific Software Engineering tasks fundamentally depends on the communication technique used, known as Prompt Engineering [20]. Key techniques include Chain-of-Thought Prompting [19] and Few-Shot Prompting [1].

This work adopts Zero-Shot Prompting by providing task instructions and diagnostic data without worked examples of correct diagnoses [9]. Two limitations are particularly relevant to this study. First, the lost-in-the-middle phenomenon: even with large context windows, models may underuse information positioned in intermediate portions of the context [11]. Second, hallucinations may introduce false information or technically incorrect justifications in persuasive language [7]. In performance analysis, incorrect diagnoses formulated with precise technical terminology can mislead engineers into investigating the wrong components. In this study, particular attention is given to claims that either contradict the provided evidence or introduce information unsupported by the available diagnostic data [7].

2.3 Related Work

Ramu [15] proposes *PerfDetectiveAI*, a conceptual framework designed to monitor performance metrics and identify low-throughput areas in software applications. Based on statistical modeling and predictive analytics, the approach focuses on extracting infrastructure data to provide recommendations to engineers. However, the approach relies on predictive analytics rather than LLM-based interpretation and diagnostic report generation.

LLM-based RCA has recently been investigated in contexts such as cloud incidents and software failure reports. Chen *et al.*[2] use LLMs to support RCA from operational diagnostic data, whereas RCEGen [12] generates root-cause explanations from software bug reports. These studies address operational incidents or software-failure reports rather than the interpretation of controlled load-test evidence during development.

Verkholomova and Nazaruka [18] review Generative AI applications in software performance testing and identify open challenges in current adoption. Their analysis motivates empirical investigation of AI support beyond test preparation, including interpretation of performance evidence after execution. In contrast to prior incident-oriented RCA studies, this work contributes grey-box, post-load-test diagnostic evidence and an analysis of unsupported causal claims, rather than LLM-based RCA in general.

3 Methodology

The study adopted an exploratory controlled laboratory evaluation with four steps: **(i)** injecting performance faults into a laboratory application; **(ii)** executing load tests and collecting telemetry; **(iii)** consolidating the data; and **(iv)** submitting the evidence to LLMs for diagnosis.

3.1 Experimental Environment

The test environment comprised an Application Programming Interface (API) developed in Node.js using the Express framework. Node.js was selected because its event-loop execution model provides a suitable environment for observing sequential asynchronous delays and memory retention in API workloads.

To reduce external variability, the database was simulated in memory, avoiding network, disk, and DBMS effects. The simulated dataset consisted of 50 user records and 500 order records, generated programmatically at server startup. A fixed 10 ms delay was applied to each simulated data-access operation, yielding approximately 500 ms of sequential waiting across 50 operations. The value is an experimental parameter rather than a measured database latency.

Server-side health monitoring was implemented directly in the application using Node.js native APIs, via a setInterval with a five-second interval. At each cycle, two indicators were logged: Heap memory usage, obtained via process.memoryUsage().heapUsed, and the one-minute system load average, obtained via os.loadavg()[0]. We emphasize that os.loadavg()[0] represents operating-system load rather than process-level CPU utilization. Therefore, this signal was used only as coarse system-load telemetry and was not interpreted as a direct measurement of Node.js CPU consumption.

All experiments were executed on a Windows host, on which `os.loadavg()[0]` consistently returns `[0, 0, 0]`, explaining the

constant 0.00 values observed and confirming that this signal cannot be used to rule out CPU-related bottlenecks.

3.2 Fault Injection

We built two degradation scenarios as independent API routes.

Scenario 1-Simulated N+1 Data Access (`/api/problem1`): Instead of retrieving users and orders through a single data-access operation, the route sequentially iterated over 50 users and executed one simulated asynchronous access per user. Each operation introduced a 10 ms delay, producing approximately 500 ms of accumulated waiting per request. The scenario emulates the sequential latency amplification of an N+1 pattern [8], but not DBMS-specific effects such as connection-pool contention or query-processing overhead.

Scenario 2-Memory Leak (`/api/problem2`): Each request allocates an array of 100,000 strings and stores it in a global-scope variable, deliberately retaining references so that the allocated data remains reachable and cannot be reclaimed by Node.js's Garbage Collector. Unlike Scenario 1, whose impact is immediate in per-request latency, Scenario 2 causes cumulative degradation: Heap memory grows monotonically throughout the test, potentially causing Out of Memory (OOM) termination[1].

3.3 Load Test Configuration

Both scenarios used 50 VUs for 30 s, with a 1 s sleep between iterations and an experimental 500 ms response-time threshold. The threshold was intentionally defined to establish a reproducible degraded condition and was not derived from a production SLO. Because each VU iteration includes route response time, the number of completed requests differed across scenarios. We therefore used the K6 summaries as the source for request counts and latency distributions. To standardize the evidence submitted to the models and avoid exposing raw request-level traces, we used the native K6 JSON summary (–summary-export), including mean, median, maximum, p90, and p95 statistics.

3.4 AI Analysis Procedure

The prompt was structured by combining two complementary data sources: (i) the K6 JSON summary, including latency percentiles and error rates, and (ii) server telemetry logs containing system load and Heap usage every five seconds. This combination provides external performance metrics and lightweight internal telemetry without exposing source code.

The prompt instructed the model to act as a Senior Performance Engineer, perform a Grey-Box diagnosis correlating load-test metrics with server telemetry, diagnose the architectural nature of the observed bottleneck, and suggest what developers should investigate in the code to mitigate the observed degradation (Figure 2). We characterize this setting as Grey-Box because the model does not access source code, but it receives internal telemetry signals from the server, such as system load and Heap memory usage, in addition to external load-test metrics. The prompt structure was identical across scenarios; only the diagnostic data and route identifier differed.

Analysis Prompt (Zero-Shot – Grey-Box Approach):
Act as a Senior Performance Engineer.
I ran a load test using K6 on the `/api/problem1` route of a Node.js application. The system exceeded the experimental response-time threshold of 500 ms.
Below are the consolidated load test results from K6:
`[K6 JSON SUMMARY INSERTION]`
And here are the server telemetry logs collected during the 30-second test:
`[LOG INSERTION (SYSTEM LOAD AND HEAP MEMORY)]`
Your task: Perform a Grey-Box Root Cause Analysis. Without access to the source code, cross-reference the response-time metrics with server resource consumption to diagnose the architectural nature of this bottleneck. Suggest what developers should investigate in the code to mitigate the observed degradation.

Figure 2: Grey-Box prompt structure submitted to the LLM, omitting source code (English translated representation).

3.5 Model Selection

We evaluated GPT-5.5 Instant (OpenAI), Claude Sonnet 4.6 (Anthropic), and Gemini 3.5 Flash (Google), selected for public availability and relevance to day-to-day software engineering workflows. Prompts were manually submitted through the official Web interfaces using the default user-facing configurations. All executions were conducted between June 4 and June 5, 2026, on free-tier accounts with default session settings and no additional features (file analysis, code execution, browsing, or persistent memory) enabled beyond each interface's default configuration. The study therefore evaluates the model configurations exposed by these interfaces at data-collection time rather than controlled API-level executions.

3.6 Evaluation Protocol

A structured rubric was applied independently by three domain experts with experience in software engineering and performance testing. Criteria and descriptors were defined *a priori*, before collecting model responses, to prevent prior knowledge of results from influencing scoring levels. Final scores represent the arithmetic mean of the three experts' ratings. The rubric is divided into two axes: qualitative (Likert scale [10]) and reliability (unsupported-claim severity).

Criterion A (Table 1) measures precision in inferring the injected bottleneck class from load-test metrics and telemetry. Criterion B (Table 2) measures the technical viability of proposed solutions within the Node.js ecosystem. Criterion C (Table 3) measures the severity of unsupported claims.

A claim was considered unsupported when information not present in the evidence was stated as established fact. Plausible alternatives explicitly framed as hypotheses were not automatically penalized. For Criteria A and B, final scores correspond to the arithmetic mean of the three expert ratings. For Criterion C, we

[1]For consistency with the public replication package, this route is implemented as `/api/problema3` in the source code; the full scenario-to-route mapping is provided in the repository `README`. A third, CPU-blocking route is also implemented but was not evaluated in this study.

report the highest unsupported-claim severity observed among the evaluators, since a single severe unsupported claim may be sufficient to compromise diagnostic reliability.

Table 1: Criterion A: Diagnostic Precision

Score	Description
1	Ignored crucial evidence or inferred an incorrect bottleneck class.
2	Detected degradation, but provided an incorrect technical explanation.
3	Identified a plausible bottleneck class only generically.
4	Identified the injected bottleneck class, but with limited technical justification.
5	Precisely identified the injected bottleneck class and connected the observed evidence to its expected mechanism.

Table 2: Criterion B: Actionability of Recommendations

Score	Description
1	Did not suggest any solution or suggested something completely unfeasible for the architecture.
2	Provided obvious and generic advice (*e.g.*, "Improve the route's code").
3	Technically valid suggestion, but too broad, requiring additional research to be applied.
4	Suggested applicable practices consistent with the stack (*e.g.*, suggested implementing a cache).
5	Provided specific technical investigation or remediation steps within the Node.js/Express context.

Table 3: Criterion C: Unsupported-Claim Severity

Score	Description
0	No unsupported factual claim.
1	Unsupported claim that does not invalidate the main diagnosis.
2	Unsupported or fabricated claim that materially compromises the diagnosis.

4 Results

4.1 Scenario 1 (N+1 Queries)

Gemini 3.5 Flash scored 3.0 on Criterion A, detecting metric anomalies and offering a well-organized diagnostic guide. It scored 3.7 on Criterion B, providing a clear and operational diagnostic guide. However, one evaluator identified a mild unsupported claim (Level 1): the model introduced unverified hypotheses, such as asserting a "connection pool limit of 5", which was not present in the provided evidence.

Claude Sonnet 4.6 received the highest Criterion A score (4.7) and explicitly identified N+1 data access as a high-priority hypothesis. Its Criterion B score was 4.3, indicating technically actionable recommendations. One evaluator identified a mild unsupported claim (Level 1): the model presented additional hypotheses not fully supported by the data, such as an "artificial floor" and "result buffering", without invalidating the core diagnosis.

GPT-5.5 Instant received the lowest scores in this scenario. We considered its diagnosis excessively generic because it identified symptoms without detailing technical resolution steps or addressing Node.js-specific aspects. Notably, it was the only model with no unsupported claims identified by the evaluators (Level 0 on Criterion C), remaining strictly within the provided evidence.

Table 4: LLM Evaluation – Scenario 1: N+1 Queries

Model	Crit. A (1–5)	Crit. B (1–5)	Crit. C (0–2)
Gemini 3.5 Flash (Google)	3.0	3.7	1
Claude Sonnet 4.6 (Anthropic)	4.7	4.3	1
GPT-5.5 Instant (OpenAI)	2.0	2.3	0

Note: Criterion C score 0 = no unsupported claim (best); 1 = mild unsupported claim; 2 = severe unsupported claim. (Worst occurrence verified among evaluators was considered.)

Table 5: LLM Evaluation – Scenario 2: Memory Leak

Model	Crit. A (1–5)	Crit. B (1–5)	Crit. C (0–2)
Gemini 3.5 Flash (Google)	5.0	4.7	0
Claude Sonnet 4.6 (Anthropic)	4.3	4.3	1
GPT-5.5 Instant (OpenAI)	3.7	4.0	1

4.2 Scenario 2 (Memory Leak)

Gemini 3.5 Flash received the highest scores in Scenario 2, including 5.0 on Criterion A. It correctly cross-referenced low K6 latency with continuous Heap growth from 161 MB to over 1.4 GB, identifying a Memory Leak and warning about OOM risk. On Criterion B (4.7), its investigative suggestions directly targeted global accumulators, listeners, and caches and proposed tools such as node –inspect. Evaluators characterized the response as highly didactic and operational, with no unsupported claims identified (Level 0).

Claude Sonnet 4.6 maintained robust performance (4.3 on both Criteria A and B), clearly identifying the active leak and providing high-quality recommendations including heap snapshot guidance with `node –inspect` and identification of per-request growing module variables. However, it again incurred a mild unsupported-claim penalty (Level 1): the model asserted with technical certainty that high `iteration_duration` values derived from Garbage Collector pauses blocking the event loop, when the latency data did not fully sustain this claim (the time could stem from native K6 sleeps).

GPT-5.5 Instant showed substantial improvement relative to Scenario 1 (3.7 on Criterion A, 4.0 on Criterion B), with evaluator scores showing greater dispersion (4, 2, and 5). Part of the panel recognized the model correctly identified a strong leak indicator from Heap growth and provided stack-relevant recommendations (Arrays, Maps, Sets, caches without TTL, listeners without removal, heap snapshots with Clinic.js); another evaluator found the diagnosis insufficiently incisive regarding OOM risk. A mild unsupported claim (Level 1) was noted in one evaluation.

4.3 Analysis and Discussion

The consolidated data indicate that LLM precision and reliability vary substantially according to the class of architectural problem analyzed, directly addressing the RQ posed in this study.

In Scenario 1, in the observed response, **Claude Sonnet 4.6** scored highest in diagnostic precision (4.7 on Criterion A) among the three evaluated responses, through structured reasoning and explicit identification of N+1 data access as a high-priority hypothesis. Evaluators highlighted the clarity of its output, with one noting that *"even a junior developer could understand the problem and resolve it."* In the evaluated N+1 scenario, the model's ability to organize

the available evidence into an actionable diagnostic narrative appeared to complement the correctness of the inferred bottleneck class. Nevertheless, both Gemini and Claude presented mild unsupported claims (Level 1). In Gemini's case, evaluators highlighted the unsupported assumption of a *"small connection pool, with a limit of 5."* Within this scenario, the results reveal a possible tension between diagnostic precision and caution: Claude produced the most precise diagnosis but also introduced mild unsupported extrapolations, whereas GPT-5.5 Instant avoided unsupported claims while remaining diagnostically shallow.

Aggregate latency evidence in Scenario 1 is consistent with, but not uniquely diagnostic of, an N+1 data-access pattern; similar latency profiles could plausibly arise from other sequential access bottlenecks. Criterion A, as applied here, primarily rewards agreement with the injected fault class rather than independent confirmation that the observed evidence uniquely determines that class. We therefore distinguish between a response that matches the hidden injected fault and one that produces an appropriately calibrated, evidence-grounded hypothesis; future evaluations could score these two dimensions separately. Because each model produced a single response per scenario, the comparative differences reported above should be read as differences among the six observed responses rather than as stable performance differences among the underlying models (Section 5).

In Scenario 2, in the observed response, **Gemini 3.5 Flash** received the highest scores among the three evaluated responses, including unanimous maximum ratings of 5.0 on Criterion A by correlating low K6 latency with continuous Heap growth without introducing unsupported claims. Evaluators described its response as *"highly didactic and operational."* **Claude Sonnet 4.6** maintained robust performance but again introduced an unsupported inference by attributing elevated iteration_duration values to Garbage Collector pauses blocking the event loop, although the available latency data did not fully support this conclusion. **GPT-5.5 Instant** showed better diagnostic performance in the evaluated Memory Leak scenario than in the evaluated N+1 scenario, although one evaluator still considered its diagnosis insufficiently incisive regarding the risk of an eventual Out-of-Memory failure.

Taken together, these findings answer the RQ in a nuanced way: LLMs operating under a Grey-Box, Zero-Shot regime can identify the likely class of a performance bottleneck from consolidated load-test metrics combined with lightweight server telemetry. However, diagnostic reliability differed across the evaluated fault scenarios, and more assertive diagnoses were sometimes accompanied by unsupported causal extrapolations. These results indicate that no evaluated model was uniformly reliable across both bottleneck classes and reinforce the need for human validation before acting on AI-generated diagnostic hypotheses.

5 Threats to Validity

We discuss potential threats to the study's validity and the procedures adopted to mitigate them, based on the experimentation principles in software engineering established by Wohlin *et al.* [21].

Construct Validity: The rubric may simplify the complex task of performance diagnosis into fixed ordinal criteria. To reduce this threat, descriptors were defined *a priori* and Criterion A was anchored to the known injected bottleneck class. However, the study evaluates class-level diagnosis and investigation hypotheses, not code-level root-cause localization.

Internal Validity: A significant threat concerns the possibility of human bias during subjective interpretation of the model-generated responses, in addition to the non-deterministic behavior inherent to LLMs (which may generate different outputs for the same prompt). To reduce the risk of human bias, the reading and scoring of responses were conducted independently by multiple evaluators (three domain experts), consolidating results from the average of their assessments. As for non-determinism, the models were kept at their default settings to faithfully simulate organic use by an engineer. A further threat is the absence of a formal inter-rater agreement measure among the three evaluators. Although scores were consolidated through simple arithmetic averaging, this procedure does not quantify the degree of consensus underlying that average, so residual divergences in individual interpretation of the rubric may be masked. Additionally, the Scenario 2 prompt inherited Scenario 1's 500 ms response-time framing, which was not literally accurate for the memory leak scenario (Section 3.4); as this framing was applied uniformly across all three models, it does not bias between-model comparisons within that scenario.

Conclusion Validity: Another limitation concerns sample size and the absence of rigorous statistical repetitions in prompt submissions. This factor reduces the strength of absolute quantitative conclusions. This threat is accepted given the exploratory nature of this research, which aims to provide preliminary evidence on the behavior of Generative AI in Performance Engineering, motivating future large-scale investigations. Accordingly, comparative statements in Section 4 about which model performed best in a given scenario should be read as descriptions of the specific observed responses, not as evidence of stable, generalizable differences in model capability. A measurement limitation concerns the use of the one-minute operating-system load average as a coarse system-load indicator. Because the test duration was 30 seconds and `os.loadavg()` does not measure process-level CPU utilization, the collected values cannot support precise claims regarding CPU consumption by the Node.js process. This limitation restricts the interpretation of CPU-related diagnostic evidence.

External Validity: Finally, a threat to external validity is the possibility that the study's results may not generalize to complex production systems operating with multiple integrations and simultaneous bottlenecks. This limitation stems from the use of an isolated laboratory environment in Node.js with an in-memory simulated database. It was mitigated by basing the experiment's faults on classic anti-patterns widely present in industry, and by providing the models with summarized metrics typical of a real continuous-integration pipeline, ensuring the relevance of the tested scenario to practical software engineering problems.

6 Conclusions and Future Work

We investigated the viability of using LLMs as assistive tools in Root Cause Analysis of software performance problems. Through a controlled Grey-Box experiment, a Node.js API was subjected to K6 load tests, exposing three models (Gemini 3.5 Flash, Claude Sonnet

4.6, and GPT-5.5 Instant) to consolidated execution metrics from two typical architectural faults: N+1 Queries and Memory Leak.

Results show that LLMs can interpret summarized load-test metrics and server telemetry to identify performance anomalies, consistent with prior work on LLM-assisted RCA [2]. However, diagnostic performance and technical reliability differed across the two evaluated fault scenarios, which exposed distinct telemetry signatures. In the observed responses, Claude Sonnet 4.6 produced the most precise and actionable diagnosis in the N+1 scenario, although it introduced mild unsupported claims. Gemini 3.5 Flash produced the strongest evaluated response in the Memory Leak scenario, correlating Heap growth with the observed symptoms and receiving maximum diagnostic-precision scores without unsupported claims.

The use of Zero-Shot Prompting to analyze consolidated load-test evidence appears viable as a support mechanism for performance diagnosis during development. However, unsupported claims were identified in two of the three evaluated responses in each scenario. This result indicates that the evaluated model configurations should be used as diagnostic support tools within a human-in-the-loop process, requiring validation by software engineers before architectural or implementation changes are made.

Future work includes: large-scale experiments with combined concurrent fault injection; quantitative evaluation using multiple prompts in stochastic windows to isolate non-determinism effects; and investigation of Few-Shot Prompting and Retrieval-Augmented Generation (RAG) using source code (White-Box analysis) as a path to mitigating the generic recommendations observed in models such as GPT; and Retrieval-Augmented Generation grounded in architecture documentation, performance baselines, or historical incident records as a complementary strategy to improve grey-box diagnostic precision.

Artifact Availability

Committed to transparency and reproducibility, we provide research data (K6 scripts, logs, and raw model responses) at Zenodo DOI: https://doi.org/10.5281/zenodo.21878273.

Acknowledgments

In the preparation of this manuscript, a generative AI tool (Claude, Anthropic) was used to: (i) assist with English translation and language editing of the text originally drafted by the authors, in a manner analogous to grammar- and style-checking software; and (ii) help generate the JavaScript code for the experimental Node.js API (including the injected N+1 Queries and Memory Leak antipatterns) and the K6 load-testing scripts used in the experiments. All generated code was reviewed, adapted, and validated by the author, who takes full responsibility for the experimental design, implementation, data, and conclusions presented in this paper. The authors acknowledge UNIPAMPA for their support during the development of this work. Maicon Bernardino is financed by CNPq grant #307969/2026-6 and FAPERGS grant #26/2551-9000312-1. Miguel Muniz thanks Internal Call PROPPI No. 05/2025 PROIC/PROPPI/Unipampa), while Eduardo Tiadoro thanks Call No. 137/2025 - PROBIC/FAPERGS/Unipampa for their support.

References

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, Vol. 33. Curran Associates, Inc., 1877–1901.

[2] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, Jun Zeng, Supriyo Ghosh, Xuchao Zhang, Chaoyun Zhang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Tianyin Xu. 2024. Automatic Root Cause Analysis via Large Language Models for Cloud Incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 674–688. doi:10.1145/3627703.3629553

[3] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80. doi:10.1145/2408776.2408794

[4] Grafana Labs. 2024. Grafana K6: Open-Source Load Testing Tool. https://k6.io. Accessed: July 2026.

[5] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Transactions on Software Engineering and Methodology* 33, 8, Article 220 (2024). doi:10.1145/3695988

[6] ISO/IEC. 2011. *ISO/IEC 25010:2011 – Systems and Software Engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE) — System and Software Quality Models.* International Standard ISO/IEC 25010:2011. International Organization for Standardization, Geneva, Switzerland.

[7] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of Hallucination in Natural Language Generation. *ACM Comput. Surv.* 55, 12, Article 248 (March 2023), 38 pages. doi:10.1145/3571730

[8] Bill Karwin. 2013. *SQL Antipatterns: Avoiding the Pitfalls of Database Programming.* Pragmatic Bookshelf.

[9] Kristian Kolthoff, Felix Kretzer, Lennart Fiebig, Christian Bartelt, Alexander Maedche, and Simone Paolo Ponzetto. 2024. Zero-Shot Prompting Approaches for LLM-based Graphical User Interface Generation. arXiv:2412.11328 [cs.SE] https://arxiv.org/abs/2412.11328

[10] R. Likert. 1987. *A Technique for the Measurement of Attitudes.* Archives of Psychology, Columbia University. https://books.google.com.br/books?id=373RzQEACAAJ

[11] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173. doi:10.1162/tacl_a_00638

[12] Rubel Hassan Mollik, Arup Datta, Anamul Haque Mollah, and Wajdi Aljedaani. 2025. RCEGen: A Generative Approach for Automated Root Cause Analysis Using Large Language Models (LLMs). *Software* 4, 4 (2025). doi:10.3390/software4040029

[13] Ian Molyneaux. 2014. *The Art of Application Performance Testing: From Strategy to Tools* (2nd ed.). O'Reilly Media, Inc.

[14] Roger S. Pressman and Bruce R. Maxim. 2021. *Software Engineering: A Practitioner's Approach* (9 ed.). McGraw-Hill Education, New York.

[15] Ramu. 2023. PerfDetectiveAI - Performance Gap Analysis and Recommendation in Software Applications. *arXiv preprint arXiv:2306.06566* (2023).

[16] Ian Sommerville. 2016. *Software Engineering* (10 ed.). Pearson, Boston.

[17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Advances in Neural Information Processing Systems (NeurIPS)*. 5998–6008.

[18] V. Verkholomova and E. Nazaruka. 2025. Overview of the Application of Generative AI in Software Performance Testing. *In Proceedings of the 2025 66th International Scientific Conference on Information Technology and Management Science of Riga Technical University (ITMS 2025)* (2025), 100–105. doi:10.1109/ITMS67030.2025.11236704

[19] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.). http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html

[20] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. arXiv:2302.11382 [cs.SE] https://arxiv.org/abs/2302.11382

[21] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering* (2012 ed.). Springer, Berlin, Germany.