International Journal of Artificial Intelligence, Data Science, and Machine Learning
Grace Horizon Publication | Volume 6, Issue 2, 225-228, 2025
ISSN: 3050-9262 | https://doi.org/10.63282/3050-9262.IJAIDSML-V6I2P125

Original Article

AI-Powered Bug Triage Using Retrieval-Augmented Generation: A Weighted Confidence Scoring Approach with AWS Bedrock and Vector Search

Rajasekhar, Sunkara
Independent Researcher, USA.

Received On: 22/04/2025 ***Revised On: 02/05/2025*** ***Accepted On: 17/05/2025*** ***Published on: 08/06/2025***

Abstract - *Manual triage of incoming bug reports in a graphics engineering organization is expensive. A senior engineer reads the report, retrieves prior issues that look similar, consults architecture documentation, searches the source code for the relevant components, and produces an opinion on what the root cause is likely to be and which team should own the fix. This work commonly takes hours per report. This paper describes an AI bug triage agent that performs an equivalent analysis automatically and produces a structured root cause hypothesis with a confidence score. The agent is built on AWS Bedrock using Anthropic Claude as the language model. It uses a Retrieval-Augmented Generation pipeline grounded in a curated knowledge base of one thousand one hundred or more previously resolved issues, together with architecture documentation and source code search. It analyzes crash logs attached to the bug report when crash logs are present. The output is a root cause analysis with a confidence score derived from a weighted combination of five signals: historical pattern match against the knowledge base of resolved issues, source code match against the components implicated by the report, crash stack analysis, log evidence, and fix ownership. The weights adjust dynamically based on which signals are available for a given report. A Flask web dashboard exposes real-time triage status, analytics, filterable history, and routing views for issues that fall outside the team scope. Deployment of the agent reduced manual triage latency from hours to minutes for the cases the agent handles end to end, while preserving analyst trust through transparent and inspectable confidence scoring. The paper describes the architecture, the scoring algorithm, the dashboard, and the operational discipline that keeps the agent useful.*

Keywords - *Bug Triage, Retrieval-Augmented Generation, RAG, Large Language Model, AWS Bedrock, Anthropic Claude, Vector Search, Confidence Scoring, Knowledge Base, Graphics Engineering, Flask Dashboard, Root Cause Analysis.*

1. Introduction

Software engineering organizations that ship hardware-coupled software at scale receive a steady stream of bug reports. A graphics engineering team that owns the graphics rendering stack on consumer streaming and smart display devices is a representative case. Reports come from internal engineers, from quality engineering test runs, from device telemetry, and from external customers. The first decision the team has to make on each report is whether the report belongs to the team at all, and if so, what the likely root cause is and which sub-team should own the fix.

This decision is what bug triage is. It sounds simple, but in practice it requires reading the report, retrieving prior issues that look similar, consulting architecture documentation to remember how the affected components fit together, searching the source code for the components implicated by the report, and reading any attached crash logs to extract evidence about what went wrong. A senior engineer who does this work well can produce a credible root cause hypothesis in tens of minutes for a simple report and in hours for a complex one. The cost of doing this manually at the volume of incoming reports is significant, and it concentrates the cost on the engineers who are best equipped to do it, which is the same set of engineers who are also best equipped to fix the bugs.

This paper describes an AI bug triage agent that performs an equivalent analysis automatically. The agent is built on AWS Bedrock and uses Anthropic Claude as the language model. The architecture is Retrieval-Augmented Generation grounded in a curated knowledge base of one thousand one hundred or more previously resolved issues, with additional retrieval against architecture documentation and source code search. For reports that include crash logs, the agent also performs crash log parsing. The output is a structured root cause hypothesis with a weighted confidence score. The agent reduced manual triage latency from hours to minutes on the cases it handles end to end.

The contribution of the paper is not RAG itself, which is by now a standard technique for grounding language models in proprietary knowledge. The contribution is the combination of RAG with a weighted confidence scoring

algorithm that uses five distinct signals, the dynamic weight adjustment that handles reports for which some signals are unavailable, and the operational design that keeps the agent's outputs transparent enough that human analysts continue to trust them. The paper also describes the supporting dashboard and routing infrastructure that make the agent usable in production. The rest of the paper is organized as follows. Section 2 describes the prior manual triage process. Section 3 describes the agent architecture. Section 4 covers the weighted confidence scoring algorithm. Section 5 describes the Flask dashboard. Section 6 discusses the operational discipline that keeps the agent trustworthy. Section 7 covers limitations and future work. Section 8 concludes.

2. The Prior Manual Triage Process

Before the agent was deployed, the team triaged incoming reports manually. The process had four broad steps. The first step was reading the report, including any attached logs or telemetry. The second was retrieving prior issues that looked similar, primarily through keyword search in the issue tracker. The third was consulting architecture documentation when the components implicated by the report were not in the engineer's working memory. The fourth was searching the source code for the components implicated by the report, to see whether recent changes might have introduced the defect.

Performed well, the manual process produced credible root cause hypotheses and routed reports to the correct sub-team on the first attempt most of the time. Performed under time pressure, it produced misroutings that the receiving sub-team would then bounce back to the triager, which added latency and frustration. The cost in engineer time was the dominant cost. The cost in latency was the secondary cost. The agent described in this paper addresses both.

3. Agent Architecture

The agent is built on AWS Bedrock and uses Anthropic Claude as the language model. The retrieval substrate is a vector store containing embeddings of three classes of content: a knowledge base of previously resolved issues, architecture documentation, and source code. The retrieval is performed using semantic search against the vector store. The retrieved content is provided to the language model along with the bug report itself, and the model produces a structured output that includes the root cause hypothesis, the supporting evidence, and the routing recommendation. The confidence score is computed separately from the language model output, by the scoring component described in Section 4.

3.1. Knowledge Base of Resolved Issues

The knowledge base contains one thousand one hundred or more previously resolved issues. Each issue is represented by its title, its description, the resolution that was eventually applied, and the metadata that links it to the relevant components and owners. The knowledge base is the most valuable retrieval source for the agent, because it directly answers the question of how the team has historically resolved issues that look like the current one.

3.2. Architecture Documentation

The architecture documentation describes the components of the graphics rendering stack and how they fit together. It is retrieved when the bug report implicates components whose relationships matter for the root cause analysis. The documentation is the second most valuable retrieval source because it provides the connective tissue that lets the language model reason about how a symptom in one component might originate in another.

3.3. Source Code Search

Source code is searched in two modes. The first mode is semantic search against embeddings of the code, to find functions and modules that look related to the report. The second mode is keyword search against the recent commit history, to find commits that touched the implicated components in the period preceding the report. The two modes complement each other. The semantic search is good at finding the relevant code regardless of how the report describes it. The keyword search is good at finding the specific recent change that might have introduced the defect.

3.4. Crash Log Parsing

When the bug report includes a crash log, the agent parses the log to extract the crashing thread, the call stack, the relevant register state when available, and any error messages that immediately precede the crash. The parsed crash log is then used both as direct evidence in the root cause analysis and as an input to the source code search, because the call stack identifies the specific functions whose source code is most relevant.

3.5. Language Model Inference

The retrieved content and the parsed bug report are assembled into a prompt for the language model. The prompt instructs the model to produce a structured output that includes the root cause hypothesis, the supporting evidence drawn from the retrieved content, the implicated components, and the recommended routing. The structured output is parsed by the agent and combined with the confidence score to produce the final triage record.

4. Weighted Confidence Scoring

The confidence score is the agent's estimate of how much weight a human reader should give to the agent's root cause hypothesis. The score is computed from five signals, each of which captures a different facet of the evidence available for the report.

4.1. The Five Signals

The first signal is historical pattern match. This signal measures how closely the report matches a previously resolved issue in the knowledge base. A close match raises the signal. A weak or absent match lowers it. The second signal is source code match. This signal measures whether the source code search produced functions or modules that the language model's reasoning relies on for the hypothesis.

A strong match raises the signal. A weak or absent match lowers it. The third signal is crash stack analysis. This signal is only available when the report includes a crash log. It measures whether the parsed crash stack points to a specific component and call path that the hypothesis explains. A clean match raises the signal substantially, because crash stacks are high-quality evidence when present.

The fourth signal is log evidence. This signal measures whether non-crash log content in the report supports the hypothesis. Log evidence is less specific than a crash stack, but it can be present even when no crash occurred. The fifth signal is fix ownership. This signal measures whether the hypothesis points to a component whose ownership is unambiguous within the team. A confident routing increases the value of the triage record to its recipient, because the recipient does not have to re-decide who owns the fix.

4.2. Dynamic Weight Adjustment

Not every signal is available for every report. A report without a crash log carries no crash stack signal. A report about a brand-new component carries no historical pattern signal and a weak source code signal. The weight assigned to each signal is therefore dynamic. When a signal is unavailable, its weight is redistributed across the remaining signals in proportion to their base weights. When a signal is available but weak, its weight is reduced and the freed weight is similarly redistributed. The intent is that the confidence score remains comparable across reports with different signal availability, rather than artificially low for reports that simply have fewer signals.

4.3. Why Scoring Is Separate from the Language Model

The confidence score is computed by the scoring component, not by the language model. This separation is deliberate. The language model is good at producing structured prose hypotheses grounded in retrieved content. It is less reliable at producing a calibrated confidence number on its own output. The scoring component, by contrast, has direct access to the evidence the language model used and can produce a score that is reproducible and inspectable. A human reader who disagrees with the score can look at the signal values that fed into it and see exactly why the score came out the way it did.

4.4. Calibration

The weights were initially set based on engineering judgment about the relative reliability of each signal. They were then adjusted by reviewing the agreement between the agent's score and the eventual outcome of the triage. The calibration is ongoing rather than one-time, because the relative reliability of each signal can shift as the knowledge base grows and as the team's coding practices evolve.

5. The Flask Web Dashboard

The agent is exposed to users through a Flask web dashboard. The dashboard is the interface through which engineers consume the agent's output and through which the triage team monitors agent operations.

5.1. Real-Time Status

The dashboard shows real-time status of the agent's current activity. Reports that are being triaged appear with their stage in the pipeline. Completed triages appear with the resulting hypothesis, the score, and the routing recommendation. Failed triages appear with the failure reason so that the operations team can investigate.

5.2. Analytics

The dashboard exposes analytics over the agent's outputs. Average confidence by component, time-to-triage distributions, agreement rates between agent recommendations and eventual outcomes, and the distribution of root cause categories over time are all available. The analytics give the triage team a view into how the agent is performing as a whole, not just on individual reports.

5.3. Filterable History

Every triage the agent has performed is retained in a filterable history. The history is the audit trail for the agent's operation and is the source of truth for debugging cases in which the agent's recommendation disagreed with the eventual outcome. Filters allow the user to narrow the history by component, by time range, by confidence band, or by outcome category.

5.4. Routing for Non-Team Issues

Some reports submitted to the team are not in fact about components the team owns. The agent identifies such reports and surfaces them in a separate routing view, with a recommendation for which team should receive them instead. The routing view is the mechanism by which the agent unblocks the team from the cost of triaging issues that are not the team's responsibility.

6. Operational Discipline

An AI triage agent is only useful if its outputs are trusted. Trust is not granted by the technology; it is earned by the agent's track record and by the operational discipline that keeps the agent honest.

6.1. Transparent Outputs

Every triage output includes the evidence that the language model used. The retrieved knowledge base entries, the architecture documentation passages, and the source code snippets are all attached to the output. A human reader can verify the agent's reasoning by reading the same evidence the agent used. This transparency is what allows engineers to disagree with the agent on specific cases without losing trust in the agent as a whole.

6.2. Human Review of Low-Confidence Triages

Triages with confidence scores below a configurable threshold are flagged for human review. The threshold is set so that the volume of human review is sustainable for the triage team. Above the threshold, the agent's recommendation is acted on directly. Below it, a human looks at the agent's output before any action is taken. The threshold is recalibrated as the agent's track record evolves.

6.3. Continuous Knowledge Base Curation

The knowledge base is curated continuously. New resolved issues are added. Issues whose resolution turned out to be incorrect are flagged or removed. The curation is what keeps the historical pattern match signal meaningful over time, because a knowledge base that drifts away from the team's current practices will produce matches that are no longer relevant.

6.4. Audit of Routing Recommendations

Routing recommendations are audited periodically. If the agent is routing reports to a sub-team that disagrees with the routing, the audit surfaces the disagreement and the routing rules are updated. This is a guardrail against the agent drifting away from the team's ownership model.

7. Limitations and Future Work

7.1. Limitations of the Knowledge Base

The agent's quality depends on the quality of the knowledge base. Reports about issues for which no historical match exists are intrinsically harder for the agent, because one of the five signals is absent. The dynamic weight adjustment compensates for this in the scoring, but the underlying analysis is still less well-supported. The knowledge base is being grown over time, but it cannot anticipate genuinely novel issues.

7.2. Limitations of Crash Log Parsing

Crash log parsing is the strongest single signal when it is available. It is not always available, and when it is available, it is sometimes degraded by truncation, by missing symbols, or by crashes that occurred in components for which symbols are not collected. The agent handles these cases by lowering the crash stack signal, but the underlying analysis quality drops with it.

7.3. Forward Migration to Architecture Improvements

The next-generation architecture for the agent, which uses ECS Fargate, OpenSearch with k-NN HNSW indexing, Amazon Titan embeddings, and DynamoDB for workflow state, is described in companion work. The migration to that architecture is intended to remove some of the current operational constraints while preserving the scoring algorithm and the dashboard described here.

8. Conclusion

AI-powered bug triage with RAG is feasible in production for a graphics engineering organization, and it reduces manual triage latency from hours to minutes on the cases that the agent handles end to end. The combination of a curated knowledge base, retrieval against architecture documentation and source code, crash log parsing, and a weighted confidence scoring algorithm with dynamic weight adjustment produces outputs that are accurate often enough and transparent enough to earn analyst trust. The Flask dashboard makes the outputs consumable in production. The operational discipline around transparent outputs, human review of low-confidence triages, knowledge base curation, and routing audits is what keeps the agent useful over time. Teams considering similar deployments can apply the same architecture and discipline, with attention to the specifics of their own knowledge base and their own confidence calibration.

Acknowledgments

This work was performed in the context of bug triage for a graphics engineering team that owns the graphics rendering stack on consumer streaming and smart display devices. The author thanks the engineers who contributed to the knowledge base curation and the dashboard, and the sub-team owners who provided routing feedback during agent calibration.

References

[1] Lewis, P. et al. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. NeurIPS 2020.
[2] Anthropic. Claude model family documentation.
[3] Amazon Web Services. Amazon OpenSearch Service documentation, including vector search and k-NN.
[4] Amazon Web Services. Amazon Titan embeddings documentation.
[5] Flask project documentation.
[6] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention Is All You Need. NeurIPS, 2017.
[7] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. NAACL-HLT, 2019.
[8] Brown, T. B. et al. Language Models are Few-Shot Learners. NeurIPS, 2020.
[9] Karpukhin, V., Oguz, B., Min, S., Lewis, P., Wu, L., Edunov, S., Chen, D., and Yih, W. Dense Passage Retrieval for Open-Domain Question Answering. EMNLP, 2020.
[10] Reimers, N. and Gurevych, I. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. EMNLP-IJCNLP, 2019.
[11] Johnson, J., Douze, M., and Jegou, H. Billion-Scale Similarity Search with GPUs. IEEE Transactions on Big Data, 2021.
[12] Izacard, G. and Grave, E. Leveraging Passage Retrieval with Generative Models for Open Domain Question Answering. EACL, 2021.
[13] Touvron, H. et al. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971, 2023.