InfraMind AI: An Evidence-Grounded Multiagent Framework for Software Incident Root Cause Analysis

Miss Snehlata K Choudhari[1], Ruturaj Vijukant Bardapure[2], Yashraj Mukund Dhonde[3], Om Arun Rokade[4], Ayush Arun Vasawade[5], Shubham Ravindra Narute[6]

[1,2,3,4,5,6]*Department of Computer Science and Engineering (AI & ML), D. Y. Patil College of Engineering & Technology, Kolhapur, Maharashtra, India*

***Abstract*—Software incident response frequently requires engineers to manually connect software exceptions, telemetry, source-code modifications, deployment records, technical documentation, and earlier incidents. Log anomaly detectors can become aware of uncommon behavior, but anomaly detection on my own does not reliably give an explanation for why a failure passed off. This paper gives InfraMind AI, a software-primarily based incident-investigation framework designed to connect those evidence resources thru a code-aware ingestion pipeline, retrieval-augmented generation (RAG), and 6 cooperating specialist retailers. The proposed workflow normalizes telemetry and repository findings, correlates artifacts by way of carrier and time, retrieves applicable runbooks and prior incidents with provenance, and synthesizes ranked root-purpose hypotheses with assisting proof, uncertainty, and remediation steerage. The prototype context motivating this design comprises a Next. Js dashboard, a FastAPI carrier layer, a Node/Express utility used for controlled fault injection, and a neighborhood SDK/CLI-orientated crawler workflow. The paper contributes (i) a structure that treats repository evidence and operational telemetry as complementary inputs, (ii) a proof settlement for traceable agent outputs, and (iii) a reproducible assessment protocol based totally on fault injection, baseline comparisons, ablation tests, and metrics for localization accuracy, retrieval fine, quotation assist, latency, and operational cost. No quantitative effectiveness outcomes are claimed in this manuscript due to the fact an independently proven benchmark run has not been supplied. The framework and assessment plan are supposed to support rigorous validation before e-book or operational adoption.**

***Index Terms*—AIOps, incident management, root-purpose evaluation, massive language models, multi-agent systems, retrieval augmented era, code analysis, observability.**

I. INTRODUCTION

Modern cloud-local packages distribute behavior across APIs, offerings, databases, queues, deployment pipelines, and shared infrastructure. When a provider fails, the first observable signal can be an HTTP 500 response, a latency spike, a database-pool timeout, or an exception emitted via a utility. The determined symptom isn't always necessarily the underlying reason. Engineers regularly correlate logs, metrics, lines, latest commits, configuration modifications, deployment occasions, runbooks, and previous incidents earlier than they can shape a defensible diagnosis. This manual move-artifact investigation creates put off and depends heavily on service specific expertise. Research has advanced beneficial techniques for person elements of this system. Anomaly-detection surveys classify methods for coming across deviations from expected conduct [1], [2]. DeepLog fashions log sequences to hit upon deviations [3], whilst LogAnomaly extends log evaluation to semantic and quantitative styles [4]. Retrieval-augmented era dietary supplements a language model with outside evidence [5]. More recent structures use LLMs to collect diagnostic context and reason about cloud incidents [6], [7]. Yet an operational workflow nonetheless wishes to attach heterogeneous runtime signals with the code and trade history of the affected utility, make retrieved evidence auditable, and divulge uncertainty whilst proof is incomplete. InfraMind AI is designed for this integration trouble. It is a software platform for receiving software activities, associating them with a venture and carrier, scanning utility source code thru a neighborhood crawler/CLI, and orchestrating six specialist analytical roles. A RAG aspect retrieves

applicable runbooks, prior incident facts, and technical documents. The ensuing record ought to identify candidate reasons, cite the proof helping each claim, explain self-assurance, and recommend secure next steps. The objective is decision guide for engineers, not independent manufacturing changes. This manuscript is intentionally placed as a gadget-design and evaluation-techniques paper. It describes a structure, a set of testable claims, and an experiment protocol; it does now not assert measured accuracy, decreased mean time to resolution, or superiority over baselines without validated observations.

II. BACKGROUND AND RELATED WORK

2.1. Log & Telemetry Anomaly Detection

Anomaly detection aims to discover observations which differ from the expected or learned normal [1]. Deep learning techniques provide more complex representations and methods of training. However, they come with various assumptions, data requirements and conditions [2]. DeepLog utilises an LSTM network to learn and detect anomalies in log events [3].

LogAnomaly incorporates semantic information within log templates. and addresses both sequential and numerical anomalies [4]. Whilst these are effective at detecting suspicious windows and anomalous events, detection and diagnosis are separate. Simply detecting unusual events within logs does not indicate what caused them, whether it be a change in code, failed dependency, capacity issue or misconfiguration.

2.2. Retrieval Augmented Generation

RAG is used to combine a generator with retrieved pieces of external text [5]. In an incident scenario this can include runbooks, service ownership documentation, known errors, post mortems and verified troubleshooting methods. However, this is only effective if the retrieved content is relevant, up to date, appropriately scoped and accurately included in answers.

Retrieval should be used as a way to collect evidence within InfraMind AI. This means we should make note of the document ID, version or timestamp (if available), and summary of the content. If there is no relevant evidence found then the report should reflect this.

2.3. LLM Incident Diagnosis & Agents

RCACopilot collects runtime diagnostics, predicts category of root causes and explains incidents in the cloud [6]. Roy et al focuses on using LLM agents to perform root cause analysis using tools. They highlight the need for dynamic collection of diagnostics [7]. Studies have shown RCA is challenging with large telemetry corpora. OpenRCA includes 335 failure cases from 3 enterprise systems. However, there were significant issues with the performance of LLMs currently. The success rate was low [8]. RCAEval includes 3 datasets of 735 failure cases. This provides a repeatable environment for microservice RCA [9]. This shows why it is important to test with real examples. A survey of LLM Multi Agent Systems highlights common practices of workflows, interactions, infrastructure and evaluation [10]. Within InfraMind AI, agents will be specialised but this is a design choice, not that having 6 agents will be more effective than 1. This should be tested with controlled ablations to ensure the decomposition has a positive impact. This should be tested with fixed inputs, similar model settings and logged token and latency costs.

2.4. Observability & Operation Standards

OpenTelemetry provides vendor agnostic standards and components to collect and export telemetry such as traces, metrics and logs [11]. These can help standardise aspects of the signals however, the incident platform will still need to have application specific IDs and data governance standards. The NIST GenerativeAI Profile suggests generative AI risk should be managed throughout their lifecycle with governance, measurement and mitigation [12]. These concerns translate directly into incident diagnosis with risks around secrets leaking, mixed evidence between tenants, unsupported recommendations and over confident explanations. These are safety and reliability issues.

III. PROBLEM FORMULATION

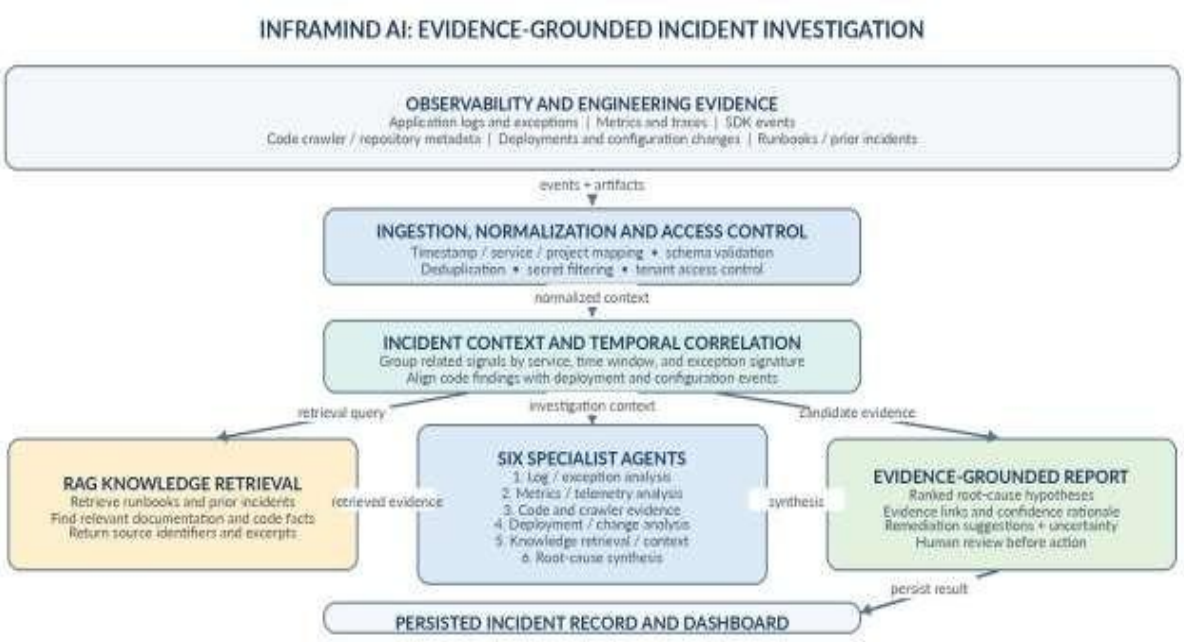
An incident candidate can be defined as a time range [t0, t1], service [s] and set of evidence E. E may include log events (L), metric summaries (M), traces (T), repo findings (C), deployment/configuration changes (D) and knowledge passages (K). Root cause candidates must be prioritised from a collection of candidates (C) where C = {c1, c2, cn}. Justifications

must be generated for each candidate. These justifications must be linked to evidence. The candidate report must include: (1) A brief summary of the incident, (2) Prioritised list of hypotheses, (3) Evidence and time range, (4) Counter evidence or gaps, (5) Rationale for confidence level, (6) Safe remediation ions and (7) Provenance of documents. Hypotheses must differ from observations. For example, if a connection pool timeout occurred at 14:03 then this would be an observation as it will be captured in telemetry. However, if a hypothesis is made that the connection pool was too small then this must be supported by configuration, usage or reproduction evidence.

The key requirement is that all evidence must be traceable. Any complex diagnostic statements must be mapped to at least one piece of evidence or flagged as being inferred. If there is no evidence available or there is contradictory evidence then confidence should be reduced, possible explanations should be provided or no single cause selected.

IV. PROPOSED INFRAMIND AI ARCHITECTURE DIRECTIONS

As seen below (fig 1), the system will take multiple sources of engineering data, process them into consistent incident context through correlation, query knowledge then sends on to experts. This has been designed to be highly modular, if there is no source available this should be taken one who know the information is not available.



Data Sources

There will be a lot of data coming in from the application causing exceptions a structured events through either SDK or HTTP APIs. This includes Telemetry data such as logs, metrics and traces. In addition, repository crawl data will come from a local CLI tool. Derivative data, as well as runs book/incident knowledge will be sourced through known sources. Tools which are involved and could certainly be of interest include repos using git, open telemetry data, vector external DB and backend IPA. Source tool will also need to log how they have actually been deployed in the release config. The data ingestion tool should validate that required fields have been provided, all dates should be normalised and any events mapped to valid tenants/Projects/Applications/Services. Events should be kept unique by having the same source ID's and duplicates filtered out. In order to do this a basic blueprint can be applied containing fields across all event_id, occurred_at, received_at, tenant_id, project_id, application_id, service_name, environment, severity, event_type, trace_id/request_idUser data fields should be

validated on the backend. Client shouldn't be able to validate users. Telemetry can be attached to any project simply by adding a project ID from the desired target.

Code crawl

Repo domain data library can provide supporting data alongside data from runtime events. This will be easy enough to achieve with a local CLI so we can easily inspect any apps source tree during prototyping. Symbolic data regarding the file locations, function/module and dependency declarations, exception handling, any risky configuration and relevant snippets/hash values can be pulled from the repo during this process. The code crawler should track this down, including - Time of crawl, repo revision, the project/app this was from, parameter indicating what was scanned, version of tool ran, warning messages around not being able to view some files. However, by default it should not include any credentials or private keys,.env files within crawl.

This data therefore represents a way of providing context and is not proof of Causality. If there is a function found of interest that fly's in line with the Exception timespan and meaning of the exception & service this will provide more evidence to support why it might be the cause of the exception. When generating the report should explain, if it was discovered exactly from the stack trace, similar symbol name, recent difference, dependencies or less likely general wording match.

Incident Correlation and De-Duplication

Incident correlation will bring together similar events which likely display the same incident or issue. Most possibly correlations could be taken from sharing the same Service/Application ID, Exception fingerprint, request/trace id, timespan, deployment window and dependency. By using a de-duplication approach the same event shouldn't produce more than one incident. Only when an incident policy on the service dictates so then a new occurrence should appear. Any correlations should clearly state the original event ID's so noting down what made the incident become an incident can be traced back.

Ranking of Effort around a potential cause could be using the following formula.

S(c) = w [L orally] + w [M orally] + w [T orally] + w [C orally] + w [D orally] + w [K orally]

Where [L, M, T, C, D or K] is a normalised effort value around logs, metrics, traces, code, deployment or knowledge respectively. As this is just an example, our current prototype tool uses an accurate scoring system. Therefore weights, normalisation and weights should be based on empirical data. A confidence score from a language model shouldn't be considered an accurate probability. Rather the tool should just have a descriptive score based on what they've learned. This shouldn't be done without testing for calibration.

RAG and provenance

For each incident a natural language query will be created using the Incidents summery, exception name, service name, relevant code findings and timeframe. Candidates can be gathered from runbooks, validated incident post-mortem analysis, documentation and previously resolved incidents. Before ranking they should filter on access level per tenant and. Results should have keep a reliable ID, with what version of the document if available, from what section and what time it was created with its content checksum. within the prompt statements should direct the model to help prove service claims using the source data or mark it as an educated guess.

As mention in the previous section, it is important to notice the difference between recall & response quality. Even if you get a useful documented it can still be mis-understood, the response may give a clear and concise response but could include irrelevant references. For this why the experiment should want to record how accurate the recall is (Recall@k/MRR), source relevancy, response fidelity and source justification. If something fails to retrieve in this instance it should be reflected in the report and no fabricated known issues should be made up.

Six-agent investigation workflow

The design will have 6 roles to limit the cognitive load per reasoning step. This can be done by having multiple LLM calls or nodes in a controlled process. For the purpose of evaluating this paper we will just look at how this behaves and not how many prompts are being used. Agent role the orchestrator should provide structured data between roles. The output from an agent should have the following fields: finding_id, role, claim, supporting_evidence_ids, contradicting_evidence_ids, source_refs, confidence_rationale, missing_data and

proposed_follow_up. Synthesis should not create new data to resolve conflicts. It should leave uncertainty in the results and highlight conflicts.

Dashboard/report

A dashboard should be provided showing the timeline of the incident, the severity, impacted service, examples of the events, metadata from the crawler scans, recent changes to consider, source snippets from the sources, findings from each agent, a list of ranked cause for the incident and recommendations to fix the issue. All claims should be clickable to show their source if available and if the user has access. If the investigation takes a long time to run then there should be a status to show what state the investigation is in (queued, running, completed, failed, timed out). The status should also have a reason for failure. The confidence should be shown based on the final structured investigation results.

V. IMPLEMENTATION OF PROTOTYPE

The solution is a browser-based application which ingests data via API and stores it. This also contains Anomalies, RAG and bespoke AI analysis, as well as a dashboard. This project has been built using a Next.js frontend, FastAPI backend. There is also a test application contained within using node/express to produce controlled errors. There is also a local SDK/CLI using node. This is not a fully audited list of all the capabilities included, but shows the implementation context. The specific versions of packages, databases, vector stores, agent names, integration with the production application and commit hash should be checked before submitting. The test application is also important as it will have controlled errors, producing a specific exception, an error thrown and handled, slow requests. Each of these will have truth data associated with them to show what the fault was, when it occurred, what endpoint/service was affected, what caused it to occur, the source version and what should be found. When sending an event to the backend, there should be an acceptance test to ensure that everything works. The event should be sent, the incident created or duplicated, background investigation completed, data retrieved, code evidence available and the same incident displayed on the dashboard.

The system should have enough metadata to be able to reproduce the run. This should include source version, crawler version/arguments, agent/model configuration, retrieval corpus version, prompts/workflow version, telemetry times and test scenario ID. In the output of experiments and paper, do not include any secrets, tokens, environment variables or private keys.

VI. RESEARCH METHODOLOGY AND EVALUATION

The main research claim should be validated using experimentation. A demonstration of deployment is not sufficient to prove accuracy. An example being successfully transporting an exception does not validate that the integration is correct. The below outline will separate the two factors, and is recommended that you include your results of the prototype testing.

6.1 Test Environment and Scenarios

Create a test environment that is not in production. This should have a versioned service, controlled telemetry, fixed knowledge corpus and fault injection manifest.

Start with at least 5 families of faults: (i) application exception/regression, (ii) database connection pool exhaustion, (iii) slow endpoint/timeout, (iv) configuration/dependency mismatches and (v) deployment related failure. If the trigger and expected root cause is known beforehand then create scenarios. Ensure that you keep a clean run with no faults. Repeat the faults, ensuring it occurs under the same conditions. If using public benchmark datasets (like OpenRCA or RCAEval) then detail exactly which version you are using. How was it preprocessed, how was it split and what baseline was used. Any licensing issues should be noted [8] [9].

A practical first experiment would be to use between 30-50 controlled runs within the fault families. This is just a suggestion. Large or public benchmarks should be used before making general statements about reliability in production. Ensure there is no leakage. This means that the ground truth label or the exact change that has been injected should not appear in the prompt or the retrieved corpus in a way the baseline cannot see.

6.2 Baselines and Ablations Testbed and Scenario Design

Ensure that the testbed is within a non-production environment. Ensure that a versioned service is being tested, and that there is some form of telemetry control. There should also be a static knowledge corpus and fault injection manifest, containing at least 5 types of faults.

These could be: (i) Application error/regression (ii) Database connection pool failure (iii) Slow endpoint or timeout (iv) Configuration/dependency error (v) Deployment related failure. Scenarios should only be added when a trigger and expected failure root cause are predefined.

Each scenario should have an ideal run and the fault should be repeated under the same conditions. If using public benchmark datasets then ensure you specify which version you're using, how it was preprocessed, how it was split, what baseline configuration you used and any licensing requirements [8][9]. As a student project, 30-50 controlled tests can be carried out across the different fault types. To begin with, it may be useful to carry out this many tests, however to make general claims about reliability at scale, you should use a larger dataset, or public benchmark. Ensure that there is no data leakage by ensuring that the true label, or exact change applied during injection is not in the prompt or corpus.

Baselines and Ablations

All baselines should be given the same incidents. They should use the same version of the model, decoding parameters, and if there are deterministic controls, they should be run using logged seeds or repeat counts. If the baseline cannot access certain evidence due to its nature, this should be noted.

Metrics

If the sample size is small then the raw count should be provided with confidence intervals. Top-k metrics require an explicit mapping between the injected fault within the incident, and candidate-cause taxonomy. If using human ratings, aim to use at least two independent raters.

Any discrepancies should be noted. If using automated faithfulness metrics, a sample should be manually validated, and the metric/model configuration provided.

Statistical Analysis and Reproducibility

The systems should be run in pairs for the same scenario. This means that all systems will see the same faults. If the results are binary, the difference between them should be shown along with confidence intervals. Latency results should be shown using the median, IQR and p95. If there are enough samples, pair bootstrapping confidence intervals should be used, or a suitable paired test. The incident should be the unit of analysis not the log line. Don't make claims about significance based on a few repeated requests that aren't independent. Exclude any samples based on pre-defined reasons, e.g., failure to trigger, and publish these numbers. The reproducibility package should contain the manifest of the scenario, version of test harness, samples of sanitised events,

labels for expected causes, corpus manifest, version of prompt/workflow, baseline configurations, scoring rubric and raw aggregated results. Ensure you've removed any API keys, personal data, private code or sensitive incident information. Even if your results are negative, e.g., no improvement seen from agent split, as long as it has been measured and reported honestly it is still useful.

VII. THREATS TO VALIDITY AND SAFETY

1. Internal and construct validity

Injecting faults can produce more clean signals than a production failure. Using synthetic stack traces and manipulated functions can help determine root causes easier than cascading dependencies, incomplete or telemetry. The test corpus may have leaked hints through reuse meaning the evaluation should contain scenarios withheld. The information injected during the evaluation must be clearly documented. Known causes must be separated from exploratory incidents. Simply reporting accuracy will not be adequate, if there are no metrics to show how high quality the evidence is and if the remediation is safe.

2. External validity

If I only have one service for processing orders, one language, few exceptions and one provider then the results cannot be generalised to all cloud systems. Different organisations have different logs, schemas, topologies, release processes and runbooks. To generalise results I would need to evaluate using a public benchmark for RCA or more services. The

system should clearly state what the unsupported signal was coming from.

3. Security and Privacy

Secrets, personal details, internal URLs and proprietary code may be found by the repository crawler or incident system. The crawler and incident should have clear scope for what they scan. All secrets should be hidden. Using least privilege credentials. Server-side tenant authentication, retention limits and audit logs. The retrieved documents and comments from the source should be treated as untrusted. Any instruction found in the log/document should not be followed. Un-redacted secrets should not be sent to the externa model. Access controls should be put in place before retrieval.

4. Human in the loop

Recommendations should be suggestions only. The system should not apply any patches, restart any dependencies in production, change access controls or execute any destructive commands based on what the LLM suggests. Remediation can be ordered based on how safe/reversible they are. Requirements should be clearly stated. The report should show what is unknown, conflicting, missing telemetry and alternative hypotheses. Confidence should be shown based on how much evidence is covered and consistent rather than just how certain we are.

VIII. DISCUSSION

The main idea behind the design is to turn incident investigations into data lineage problems. Many systems will give a plausible paragraph about the exception but few will provide a clear lineage from claim to source event, repository finding, deployment or runbook section. By making provenance part of the output schema and ablation testing the different components, InfraMind AI can provide a measurable diagnostic benefit. The crawler should provide some useful contextual information about the code but scanning large areas of the repository may not always be helpful. It could return irrelevant functions or sensitive information. Limiting the indexing to specific files and keeping the findings concise and with links to the source can help reduce noise. Whilst having 6 agents within the workflow may give more evidence coverage it could increase latency, token costs and correlated hallucinations. Structured outputs from the agents should be used with individual traceability. Compare to a single agent baseline. If splitting the agents does not provide improved accuracy or enough evidence to warrant the cost then a simpler workflow may be better. RAG should be treated as a method for retrieving evidence. There is no guarantee the runbooks will be up to date or that the incident will be similar but unrelated. Version aware search, tenant filters, clear references and checking for unsupported claims are required. The architecture should degrade gracefully, if the knowledge index, code crawler or telemetry source are not available then it should state what evidence is missing and proceed with the evidence it has.

The novelty of the research should be clearly defined. This paper does not claim to present anything new such as anomaly detection, RAG, multi-agent orchestration or root cause analysis. The aim is to provide an integration pattern that can easily be tested for combining code crawler evidence and operational evidence with provenance aware multi-agent diagnosis within an incident dashboard. Whether this improves diagnostic quality will be determined by the proposed evaluation.

IX. CONCLUSION AND FUTURE WORK

We have introduced InfraMind AI, a principled analysis system for investigating software incidents that combines application monitoring, repository querying, change correlation, RAG, expert agents, and saved reports. We provide a ranked root cause explanation with justification and uncertainty and define a set of baselines, ablations, metrics, and replication guidelines. We will perform a controlled end to end benchmarking. For each synthetic fault added we will log all events, the crawler output, sources, agent outputs and dashboard states. Future work will involve expanding the testbed and benchmarking using a publicly available benchmark such as OpenRCA or RCAEval. This will help us gain confidence in our system, ensure the methods are isolated between projects, and measure how much citation helps. We can also measure if the multi-agent approach provides better quality per unit of latency and cost. We need to replace the evaluation plan with actual results before submitting this work as an empirical research article.

Data and Artifacts

There were no public datasets, repository links or independently replicable artifacts confirmed for this manuscript. If possible, provide a link to your repository/artifact. If there are any private aspects of your work highlight this. Provide sanitized examples of your testing scenarios, code versions, configurations and measurement scripts. If you cannot publish your code, explain why and how you accessed your artifact.

REFERENCES

[1] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Computing Surveys*, vol. 41, no. 3, Art. no. 15, 2009, doi: 10.1145/1541880.1541882.

[2] G. Pang, C. Shen, L. Cao, and A. van den Hengel, "Deep learning for anomaly detection: A review," *ACM Computing Surveys*, vol. 54, no. 2, Art. no. 38, 2021, doi: 10.1145/3439950.

[3] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly detection and diagnosis from system logs through deep learning," in *Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS)*, 2017, pp. 1285–1298, doi: 10.1145/3133956.3134015.

[4] W. Meng et al., "LogAnomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs," in *Proc. 28th Int. Joint Conf. Artificial Intelligence (IJCAI)*, 2019, pp. 4739–4745, doi: 10.24963/ijcai.2019/658.

[5] P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 9459–9474. [Online]. Available: https://arxiv.org/abs/2005.11401

[6] Y. Chen et al., "Automatic root cause analysis via large language models for cloud incidents," in *Proc. ACM European Conf. Computer Systems (EuroSys)*, 2024. [Online]. Available: https://arxiv.org/abs/2305.15778