Computer Fraud and Security
ISSN (online): 1873-7056

ML-Powered Root Cause Analysis and Automated Remediation for Java Microservices

Tejendra Patel
California State University, Los Angeles (CSULA), CA, USA

Abstract

It is far from adequate to detect performance regressions in production Java microservices without proper attribution and resolution, especially in large, rapidly changing codebases without wide-ranging human involvement. This is challenging in modern continuous delivery environments, where multiple commits are bundled into a release and the root cause is inferred from heterogeneous streams of performance telemetry, version control, and incident history. In this article, let's build an end-to-end system for code change analysis with multi-modal feature engineering, gradient-boosted tree classification, SHAP-based explanations, and large language model code generation. And design an ensemble XGBoost model that learns the non-linear mapping from code change to runtime impact. By using SHAP values in order to give theoretically principled, plain-language relevance explanations that ensure engineer trust that these models are calibrated. A LoRA fine-tuned GPT-4 model then writes production-ready code changes through an AI-orchestrated pull request workflow, with human approval and staged deployment verification remaining mandatory as gates. The automation becomes an accelerant to engineering judgment rather than a substitute for it. The system is continuously retrained based on feedback from engineers to accommodate codebase changes.

Keywords: Performance Regression, Root Cause Analysis, Xgboost, Shap Explainability, Large Language Model Fine-Tuning

1. Introduction

Noticing a performance regression in production is only half the problem. The bigger problem is figuring out what caused the regression and how to fix it. Instrumentation systems generally can identify that a method has suffered a performance regression but require an engineer to analyze when it was last changed, compared to potentially large numbers of other commits to the codebase, and how to fix it. This is harder to do in a modern continuous delivery setup, where multiple changes are batched into a single release to production. With 12 commits, it may take hours of senior engineering time to determine which commit is to blame for which incident (for example, because one commit introduced latency spikes).

Purely manual diagnosis, however, becomes prohibitively difficult as systems grow, as engineers must consider telemetry, version control history, and the domain knowledge of many common runtime antipatterns. These requirements create an overwhelming cognitive load; diagnosing the problem becomes more error-prone, and gradient-boosted trees such as XGBoost are a common solution to the classification problem on structured data. One primary advantage of tree models is producing high-quality models that are also interpretable in terms of the importance of the features involved. LLMs work well because they generalize to conditions instead of just memorizing what they have seen. A machine learning classifier can be used to find the underlying problem in the software system, and the code suggestion can be generated using an LLM model. This essentially closes the gap between detecting a bug and generating code that a human engineer could analyze and rectify. This does mean that the time-saving benefits of machine learning do not remove the need for human review before shipping a bug fix..

2. The Root Cause Analysis Bottleneck

However, determining what changed in the code that caused the problem, from many intervening changes that have been pushed in production, as well as having the performance telemetry at the right level of abstraction and having the right domain knowledge, will still require a fair amount of direct human effort. For example, in modern CI/CD pipelines, developers typically merge multiple changes to create a single release, making it impossible to pinpoint the exact commit that caused the regression after it has been released. Either option (one company-wide rollback or choosing individual rollbacks) is costly to implement and operate. For example, extracting a consistent attribution across incidents and

disambiguating their meaning without tooling may take several hours of work per incident within an active development environment throughout the year. .

A further complication is that the three data streams required for root cause analysis are of different types, and performance telemetry in particular has two sources. This includes metrics of the method that is slow. Examples of such metrics include whether there is an anomaly in its latency due to CPU contention (low CPU time, high latency) or the overhead to execute the method is too much. Code changes metadata are obtained by conducting searches on version control for what changed, how much it changed, or in what manner it changed. The third area is past incident patterns, showing which kinds of code changes tend to cause problems in code paths that run in similar execution environments. To find relationships between these areas is to untangle complex, indirect relationships. String concatenation is not a problem when it is performed in an infrequently executed method, but it can be a problem when performed in a frequently executed method that runs thousands of times for each request.

This is infeasible due to the number and diversity of the combinations of code changes, execution paths, and resulting performance implications. This is also seen in software engineering for machine learning systems, where a true ML and large data system is complex enough in practice that rule-based systems are not effective. More broadly, learning and building methods for prediction performance through data-driven learning and for ML lifecycle processes such as training data quality, feature drifts, and feedback loops. This needs to be considered: whether the main problem will require new tools or an engineering system that differs from customary software systems. It must be able to learn from supervised experience on past incidents, work with any kind of code, and improve over time, as developers teach it.

3. Multi-Modal Feature Engineering and ML Architecture

3.1 Data Integration

Even without tooling, human effort is still required in correlating which code change introduced the bug from potentially thousands of intervening changes that may have been pushed into production. Human effort is required to ensure that performance telemetry is taken at the right level of abstraction that is useful to the engineers, as well as to have domain knowledge and understanding. In modern continuous integration/continuous delivery environments, with several code changes combined together as a single release, it can also be unclear which code change caused the regression. Either a company-wide rollback or an independent rollback of each individual change is expensive or difficult to perform and operate. For example, unambiguously attributing the cause of each incident to an event and disambiguating its meaning, without tooling, could take several hours of work for each incident, within an active development environment year-round.

Another complicating factor is that the three data streams, required for root cause analysis, are not homogeneous: performance telemetry has two sources in particular. One drawback is that this method can be slow. For example, it can have latency because the CPU time that it gets is too small or its code overhead is too high. Metadata is kept around code changes by looking in version control as to what was changed, how much was changed, or the way it was changed. The third area is past incident patterns. It has to do with the kinds of changes that in the past affected code paths running in the same execution environment. Finding correlations and direct/indirect relationships between these areas is difficult. For example, string concatenation in a method that will only be called a few times will not be a problem, but it will be in a method that will be called thousands of times per request.

This is infeasible because of the number and diversity of the code changes, execution paths, and hence performance effects concerned. A similar situation occurs in software engineering for machine learning systems: a true machine learning and big data system is complex enough in practice that rule-based systems are not effective. More generally, needed to learn and build tools to measure prediction performance through data-driven learning and to address issues in the ML lifecycle, including dataset quality, feature drifts, and feedback loops. Asking if the main problem requires new tools or an engineering system different from typical software systems, and whether the system could be trained through supervised learning on past incidents, work on all code types, and be improved over time by developers.

3.2 Model Design

Sub-Model	Role	Primary Inputs
Performance	Estimates the magnitude of CPU change	Method-level CPU features, change magnitude,

Impact Predictor	attributable to a code modification	file category, syntactic patterns
Root Cause Classifier	Identifies the most likely responsible commit from the deployment candidate set	Commit diff features, historical risk score, invocation frequency, loop and string patterns
Severity Estimator	Predicts user-facing impact level across low, medium, high, and critical tiers	Latency shift characteristics, traffic volume exposure, method category

Table 1: Sub-Model Roles and Inputs in the ML Ensemble [1] [5]

The model used to make predictions is an XGBoost. This is a gradient-increased tree, which is known to work well with many types of features and structured tabular data. XGBoost includes an efficient regularization procedure [1]. The model stack is made up of three models. Each model has its own sub-model making predictions for its own task. The online performance effect predictor calculates a regression tree from which it is determined by how much the effect on the CPU will change upon a code change. The root cause classifier gives a yes/no score and an estimate of the likelihood that the code change is the reason for the problem. The severity estimate (low, medium, high, or critical) is based on how much the delay has increased and how many users are affected by the code that is having the problem.

A positive-negative example is, therefore, a case where a failure that had previously been found was fixed by checking manually to find the cause and then logging the result into the tracking system. Since positive examples are defined in terms of the root cause commit, they also carry the full feature vector of the incident. Negative examples are sampled from the successful deployments and inspected for regressions so that the model does not confuse a change that caused a regression with a similar but innocuous change. It is critical to preserve this balance; the model must not regress with any commit that modifies commonly invoked methods.

SHAP (SHapley Additive Explanations) is an example of a game-theoretic approach to feature attribution. SHAP estimates each feature's contribution to a prediction by considering the contribution of each feature to predictions on all possible combinations of feature observations. SHAP values can also be used to calculate global and local feature importance scores to explain what kind of features are the most common in the dataset or why a given commit was flagged. For instance, the most important feature for a regression model was loops inside hot methods and string concatenation. This behavior is expected, since it means that the model has learned to focus on the relevant features of the data rather than noise. Experimenting with different partitions of the dataset through cross-validation yields excellent generalization results and low variance.

4. Real-Time Inference and Explainability

4.1 Inference Pipeline Architecture

Stage	Function	Data Source
Alert Ingestion	Receives regression alert payload containing regressed method, deployment version, and candidate commit list	Automated detection layer
Metric Retrieval	Fetches granular method-level CPU distributions, invocation frequency, and percentile latency profiles for current and previous versions	Data warehouse telemetry
Commit Retrieval	Pulls full code diffs, file change summaries, author metadata, and pull request context for each candidate commit	Version control system
Feature Computation	Constructs structured feature vectors per candidate commit via static analysis and feature engineering pipeline	Combined telemetry and code metadata
Model Scoring	Scores each candidate's feature vector through the root cause classifier to produce ranked regression probabilities	Trained ML ensemble
Confidence Tiering	Routes output to single-commit identification, shortlist presentation, or manual fallback based on probability thresholds	Model output layer
Explanation	Attaches SHAP-based plain-language justification to each ranked	SHAP explainability

Generation	prediction before surfacing to engineer	module

Table 2: Real-Time Inference Pipeline Stages and Their Functions [5] [7]

The inference pipeline will be started right away when there is a regression alert from the automated detection layer. The inference pipeline is triggered by the alert payload and consists of a series of events that happen one after the other in sequence. The alert payload contains the regressed method name, performance metrics, deployment version number, and list of commits included in the deployment. The pipeline, using the seed, will undergo multiple stages until it finds and computes the potential root cause candidates. The candidates will then be ranked in order of priority for the engineer to look at.

In the first step, gather performance traces from the data warehouse and compute statistics for regression and resource utilization. These include a breakdown of CPU time between both versions, usage count of each version, response time breakdown, and how the performance problem degraded over time after the defect was deployed. The second step involves collecting version control metadata about each of the commits to be included in the deployment, including the code diffs, changed files, author history, and other context available in the pull request. The third stage takes this processed data and passes it through a procedure to inspect the code and generate a structured list of features to be input to the trained ensemble model for each candidate commit.

The feature vectors are fed to the root cause classifier simultaneously to produce a probability regression for each of the root cause candidates. This is sorted and presented in bands of confidence to the engineer. A commit will be alerted if the highest probability score on it exceeds a configurable threshold. Otherwise, if there is medium confidence with multiple candidates being near the probability scores, a ranked shortlist of the top three candidates will be presented for manual review by the engineer. This makes it easy in cases where there isn't a possible candidate that meets the minimum necessary criterion to return to the routine of manual analysis, rather than sending a bad forecast into an automated process that creates more problems.

Under load, the model-serving infrastructure can produce predictions in less than one second by batching requests, that is, sending the feature vectors of the candidate changes for a single deployment to the model-serving infrastructure in a single request. The time between the alarm going off and the system running through its causes in priority order needs to be as short as possible. Engineers need the information in seconds rather than minutes, or, worse, it just makes it worse in a busy production environment when the cause has not been determined.

4.2 SHAP-Based Explainability and Engineer Trust

One of the requirements in designing the inference system was that an engineer should never see a prediction without seeing an explanation of why the prediction was made. This principle comes from experience in productionizing these machine learning models and finding that either engineers trust the prediction too much without checking or ignore it because they cannot understand how it was made. In a critical production environment, both of these situations are suboptimal because one will waste time trying to diagnose or fix the problem, or worse still, the solution will only fix one of the issues without addressing the other.

SHAP (SHapley Additive Explanations) is a game-theoretic approach to explain the output of any machine learning model. SHAP is based on cooperative game theory that assigns each feature an importance value for a particular prediction. The importance value is measured by the contribution of each feature to the model output over all possible feature permutations [5]. This formulation gives us local accuracy and consistency. Local accuracy defines the sum of the SHAP values to be the difference between the model's output and the baseline's output, while consistency says that if a feature has a higher SHAP value than another feature, it had a larger impact on the prediction. These properties make SHAP values useful for visualization and as a mathematically sound input to engineering decisions. It is important that the model be interpretable in this way when using it, as engineer acceptance of the model output is what ultimately decides if the automated recommendations are followed. [8]

In practice, every root cause prediction includes an explanation in human-readable text listing the top SHAP features. For example, one flagged commit might be explained by a loop to a frequently called function being added in the commit, the body of the loop containing a string concatenation pattern that is known to create a quadratic complexity problem at scale, and a history of the author regressing performance in that module. In addition to the primary explanation, providing a structured explanation for a prediction allows the engineer to validate or dispute the model hypothesis

without having to find the specific lines of code that are the most important to the model's prediction or the pattern of lines that the model is attending to. Over time, the structured explanation for the prediction can train the engineer to develop trust in the model, which can ultimately allow the engineer to productively interact with the model rather than abandoning the task to inspect the code. However, this trust is critical for the human-in-the-loop validation the system relies on.

5. LLM-Based Fix Generation and Automated Pull Requests

5.1 Fine-Tuned Code Fix Generation

Prompt Component	Role	Content Description
Root Cause Commit Diff	Provides the model with precise visibility into what was added, modified, or removed	Full diff across affected files including syntactic and structural context
Regressed Method Context	Grounds the model's understanding of behavioral constraints the fix must preserve	Class structure, adjacent methods, relevant data types, and call graph position
Historical Fix Examples	Supplies concrete remediation templates based on structurally similar past resolutions	Incident knowledge base entries selected by pattern similarity to current regression

Table 3: LLM Prompt Components and Their Roles [2] [9]

Once the system confidently identifies the root cause commit using the ML ensemble, it moves from figuring out the problem to fixing it by using a fine-tuned large language model to create a specific, ready-to-use code fix. Choosing GPT-4 as the model of choice is for a few reasons: It is better at understanding code; it can reason about code as a whole even when it is spread out over multiple files, and it is better at explaining itself in layman's terms so engineers understand the proposed changes.

The system builds the prompt from three separate input components. The first component is the complete root cause commit diff, which shows the model exactly what lines were added, changed, or removed, along with the context of those changes in terms of syntax and meaning, and which files and methods they affect. The second part includes the surrounding code of the method that has issues, such as the class layout, nearby methods, important data types, and where it fits in the call graph, which helps the model understand the rules that any fix must follow to keep the program working correctly. The third component is a collection of past fix examples taken from a knowledge base, chosen because they are structurally similar to the current problem, which gives the model clear templates for solutions and shows how detailed the output should be.

The output consists of a fixed code block where the change is proposed as idiomatic Java code, followed by an explanation for the proposed code. It contains a one-line explanation of what the main issue was, what was changed and an estimate of the kind of performance difference expected by this change. To ensure the fixes have a consistent structure, prompt engineering and output parsing are used. This allows downstream automation to easily extract and insert the individual parts of the fix.

The fine-tuning is done with the help of a methodology called Low-Rank Adaptation (LoRA), which allows a pretrained model to be adapted to a target task by adding trainable low-rank matrices to the model architecture in the attention layers instead of training the model from scratch. The training dataset consists of past incident-fix pairs from version control history, which link changes labeled with performance regression tags to their later fixes. This paired structure makes sure that the model learns the semantic relationship between patterns that cause regression and their canonical solutions, not just the style of the code. Adjusting this carefully selected dataset leads to clear improvements in how well the generated fixes fit the structure, how accurately the root causes are explained, and how closely the predicted performance effects match the actual results.

Workflow Stage	Responsibility	Automation or Human
Branch Creation	Creates a dedicated git branch scoped to the regression incident for clean audit trail	Automated
Code Application	Applies generated fix programmatically with file path resolution and conflict detection	Automated
CI Test Execution	Runs unit and integration tests against the modified codebase to surface functional regressions	Automated
Pull Request Submission	Opens PR populated with regression summary, root cause analysis, fix diff, test results, and rollback procedure	Automated
Engineer Review and Approval	Validates fix correctness and business logic alignment, and approves or rejects merge	Human (mandatory gate)
Staging Validation	Deploys fix to staging and compares method-level telemetry against pre-regression baselines	Automated with human oversight
Production Deployment	Promotes fix to production only after staging validation confirms expected improvement	Human-approved

Table 4: AI Agent Workflow Stages and Responsibilities [4]

5.2 Automated Pull Request Orchestration and Human-in-the-Loop Safety

After the repair is made, an AI agent handles the entire remediation process, from applying the code to submitting the pull request. Ultimately, the engineer is not required to do anything manually until the final approval gate, and the agent simply creates a Git branch. The change request is kept separate from changes that are already being made, and a link is kept to the place where the patch originated. The automatic code review applies the code changes. It then decides the path names, applies the diffs, checks for conflicts, and finally runs a CI test suite to run unit tests and integration tests on the new code to catch any additional problems that the patch may have introduced before a human reviews the patch. This documentation depth is deliberate: it reduces the cognitive burden on the reviewing engineer to near zero for straightforward cases and provides a structured decision framework for more complex ones.

In this workflow, human approval is still a non-negotiable and architecturally enforced gate. There is no way to automatically merge or deploy a created repair; the pull request system is set up so that an engineer must explicitly approve any merge action before it can happen. After gaining the green light, the repair is initially deployed to a staging environment, where its performance is evaluated by comparing data from certain techniques to earlier benchmarks. Only after staging validation confirms the expected improvement is the change promoted to production. This staged deployment model embodies the overarching principle that automation in critical engineering workflows should serve as an enhancer of human judgment rather than a substitute—a design philosophy well-documented in literature. Human-in-the-loop system design acknowledges that preserving significant human oversight over automated decisions is crucial for system reliability and accountability [10].

Conclusion

Combining multi-modal feature engineering, gradient-increased tree ensembles, SHAP-based explainability, and fine-tuned large language models into a single system to solve performance issues is a meaningful improvement over the speed and accuracy of detecting, diagnosing and fixing production Java microservices. The article moves the ad hoc, time-demanding process of attributing a root cause of an incident accurately from a human-led activity to an automated and heavily structured system providing commit-level attributions in seconds upon an alert. This is accomplished by tracking instrumentation telemetry to version control and historical incidents and by using LoRA-tuned code generation and AI-generated pull requests to reduce mean time to recovery (MTTR) and enable remediation. A change proposal generated by the system is also easily executable and verifiable by a human, who approves it and then verifies that the changes have been made as expected before deploying it into a live state. In this way, it is not an unsafe or irresponsible form of making changes to the production system. For one, performance improvements are done in the context of a

retraining pipeline. Engineers are able to provide feedback on how well the predictions and fixes are working, and the model can therefore increase its accuracy as codebases and antipatterns change over time. This shows that organizations could automatically find and fix these types of issues, end-to-end, without impacting reliability or explainability.

References

[1] Tianqi Chen and Carlos Guestrin, "XGBoost: A Scalable Tree Boosting System," in Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016. [Online]. Available: https://dl.acm.org/doi/epdf/10.1145/2939672.2939785

[2] OpenAI, "GPT-4 Technical Report," arXiv preprint, arXiv:2303.08774, 2024. [Online]. Available: https://arxiv.org/pdf/2303.08774

[3] D. Sculley et al., "Hidden Technical Debt in Machine Learning Systems," in Advances in Neural Information Processing Systems, vol. 28, 2015, pp. 2503–2511. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2015/file/86df7dcfd896fcaf2674f757a2463eba-Paper.pdf

[4] Saleema Amershi et al., "Software Engineering for Machine Learning: A Case Study," in Proc. IEEE/ACM 41st Int. Conf. Software Engineering: Software Engineering in Practice (ICSE-SEIP), Montreal, QC, Canada, 2019, pp. 291–300. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2019/03/amershi-icse-2019_Software_Engineering_for_Machine_Learning.pdf

[5] Scott M. Lundberg and Su-In Lee, "A Unified Approach to Interpreting Model Predictions," in 31st Conference on Neural Information Processing Systems, NIPS, 2017. [Online]. Available: https://arxiv.org/pdf/1705.07874

[6] Jasper Snoek, et al., "Practical Bayesian Optimization of Machine Learning Algorithms," in arXiv:1206.2944v2 [stat.ML] 29 Aug 2012. [Online]. Available: https://arxiv.org/pdf/1206.2944

[7] Muhammad Hanif, et al., "SLA-based Adaptation Schemes in Distributed Stream Processing Engines," Applied Sciences, vol. 9, no. 6, p. 1045, 2019. [Online]. Available: https://www.mdpi.com/2076-3417/9/6/1045

[8] Finale Doshi-Velez and Been Kim, "Towards A Rigorous Science of Interpretable Machine Learning," arXiv:1702.08608v2 [stat.ML] 2 Mar 2017. [Online]. Available: https://arxiv.org/pdf/1702.08608

[9] Edward Hu et al., "LoRA: Low-Rank Adaptation of Large Language Models," in Proc. Int. Conf. Learning Representations (ICLR), 2021. [Online]. Available: https://arxiv.org/pdf/2106.09685

[10] Wei Xu, "Toward Human-Centered AI: A Perspective from Human-Computer Interaction," in Interactions, Volume 26, Issue 4, 2019. [Online]. Available: https://dl.acm.org/doi/epdf/10.1145/3328485