International Journal of Scientific Research and Engineering Development-– Volume X Issue X, Year

Available at www.ijsred.com

RESEARCH ARTICLE OPEN ACCESS

Incorporating Developer Activity Signals into AIOps for Improved Incident Prediction

V. Udhayakumar [1] , Amirtha V [2]

[1]Assistant Professor, Department of MCA,, Sri Manakula Vinayagar Engineering College Puducherry 605008, India

Email: udhayakumar.mca@smvec.ac.in

[2]Post Graduate student, Department of MCA,, Sri Manakula Vinayagar Engineering College), Puducherry 605008, India

Email: amirthavijayakumar87@gmail.com

----------**********----------

Abstract:

Software systems fail and if you've spent any time in operations, you already know that most of those failures didn't come out of nowhere. Someone pushed a risky commit. A deployment went out at the wrong time. A configuration change slipped through without a proper review. The signal was there; nobody was watching for it. That is the problem this paper tries to address.

Artificial Intelligence for IT Operations (AIOps) has made genuine progress over the last decade. Modern platforms can parse millions of log lines per second, detect anomalies in metric streams, and correlate alerts across dozens of microservices. But almost all of them share the same blind spot: they watch what the system is doing, not what the people building the system were doing before something went wrong. Developer activity the commits, deployments, pull requests, and rollbacks that precede most production incidents is largely invisible to existing AIOps pipelines.

This paper proposes a framework that closes that gap. We argue that by combining traditional infrastructure telemetry with structured signals drawn directly from developer workflows, it becomes possible to detect incident risk earlier and with greater contextual clarity than infrastructure monitoring alone can provide. We present a layered conceptual architecture for this integration, identify the specific developer signals that carry the most predictive weight based on evidence from the empirical software engineering literature, and discuss the practical challenges and ethical considerations that come with deploying such a system in a real engineering organization.

Keywords — ***AIOps, Incident Prediction, Developer Activity Signals, DevOps, Software Telemetry, Pull Request Analytics, Continuous Deployment, Site Reliability Engineering, Anomaly Detection***

----------**********----------

1. Introduction

There is a moment every on-call engineer knows well. The alert fires at an awkward hour, the dashboards turn red, and the scramble begins. And then almost always somewhere in the post-mortem, someone points to a deployment that went out six hours earlier, or a pull request that merged without enough review, and says: "we should have caught that." They're right. The information was there. It just wasn't being used.

The field of AIOps has focused, understandably, on what is most observable: CPU utilization, memory pressure, request latency, error rates, log volume. These are the signals that infrastructure produces in real time, and they are genuinely useful. Tools built on these signals have reduced mean time to detection and helped operations teams manage systems at a scale that would have been unthinkable manually [1][2]. But they represent only one half of the story.

The other half lives in the developer workflow. Every time an engineer commits code, opens a pull request, triggers a deployment, or rolls back a release, they are generating structured, timestamped data that describes exactly what is changing in a system and how carefully that change was handled. This data is already being collected by version control platforms, CI/CD pipelines, and issue trackers. It is just not being fed into incident prediction systems.

There is a growing body of evidence in empirical software engineering suggesting that this data is not just interesting, it is predictive. Studies have shown that large, high-churn commits correlate with post-release defects [3], that pull requests merged without adequate review are associated with higher fault rates [4], and that deployment frequency and rollback patterns are reliable indicators of operational health [5]. If these signals predict software quality, they should also help predict operational incidents, since most incidents are, at their root, software quality failures that made it to production.

This paper brings these two threads together. We propose a framework called DevSignal-AIOps that treats developer activity as a first-class input to incident prediction, alongside the infrastructure metrics that AIOps systems already monitor. The framework is conceptual, we are not reporting empirical results from a live deployment but it is grounded in real research, designed around practical constraints, and intended to be implementable by teams with access to standard DevOps tooling.

The rest of this paper is organized as follows. Section 2 reviews the relevant literature across AIOps and empirical software engineering. Section 3 presents our proposed framework in detail. Section 4 discusses the signals we consider most valuable and why. Section 5 addresses the practical and ethical challenges of implementation. Section 6 concludes with directions for future work.

2. Literature Review

2.1 The State of AIOps

The term AIOps was coined by Gartner in 2017 to describe platforms that use machine learning and big data to enhance and partially automate IT operations [6]. Since then, the field has developed rapidly. Modern AIOps systems can perform log anomaly detection, metric forecasting, alert correlation, and root cause analysis, often in real time and at the scale of large cloud deployments.

Dang et al. [1] provide one of the most grounded overviews of how AIOps is applied in practice at a major technology company, describing the gap between what academic research produces and what production environments actually need. Their core observation that real-world AIOps must be fast, explainable, and tolerant of noisy data is one we have tried to respect in our framework design.

Deep learning approaches have become common for log parsing and anomaly detection. Du

et al. [7] introduced DeepLog, which models system logs as sequences and uses an LSTM to detect execution anomalies a technique that has since been widely adopted and extended. More recently, transformer-based architectures have been applied to metric forecasting. However, the input to virtually all of these systems remains infrastructure telemetry. The developer layer is absent.

2.2 Developer Activity and Software Quality

The empirical software engineering community has been studying the relationship between development process characteristics and software quality for decades. Nagappan and Ball [3] demonstrated that relative code churn how aggressively a codebase is being modified is a strong predictor of post-release defect density, sometimes more predictive than static code analysis metrics. This finding has been replicated across multiple projects and organizational contexts.

Mockus and Weiss [8] showed that the risk of introducing faults into a software system is related to the size and structural properties of changes, not just their content. Large changes that touch many files or many modules tend to introduce more problems, regardless of how carefully they are reviewed. This insight directly informs our signal taxonomy.

Gousios et al. [4] conducted a large-scale study of pull-based development on GitHub and found that pull requests are far more than a code integration mechanism they encode social and quality information about how a change was received, debated, and ultimately incorporated into a project. The dynamics of the review process (how many people looked at it, how long it sat, how many comments were exchanged) carry real quality signal.

2.3 DevOps and Operational Metrics

The DevOps movement brought operations and development closer together and, in doing so, created a richer vocabulary for talking about the health of a software delivery process [9]. The four key metrics popularized by the DORA research program deployment frequency, lead time for changes, change failure rate, and time to restore service represent exactly the kind of developer-facing signals we believe should be incorporated into incident prediction [5].

Beyer et al. [10] in the Google Site Reliability Engineering book describe the operational philosophy that underpins much of modern cloud operations, including the idea that toil reduction and proactive monitoring are more valuable than reactive firefighting. Our framework is very much in that spirit: we are trying to give operations teams more notice, not better post-mortem tooling.

2.4 The Gap

What is striking, reading across these two bodies of literature, is how little conversation there has been between them. AIOps researchers know that incidents correlate with deployments but tend to treat deployment events as binary triggers rather than rich signals with their own internal structure. Software engineering researchers know that process metrics predict quality but tend to evaluate them against defect databases rather than operational incident records. This paper is an attempt to connect those dots explicitly and propose a framework that takes both literatures seriously.

3. Proposed Framework: DevSignal-AIOps

3.1 Design Philosophy

Before describing the architecture, it is worth saying something about the philosophy behind it. We designed DevSignal-AIOps with three principles in mind. First, it should be additive: organizations should be able to incorporate developer signals into their existing AIOps stack without replacing what already works. Second, it should be explainable: when the system raises an alert, operators should be able to understand why, in terms that connect to concrete developer actions. Third, it should respect privacy: monitoring developer activity at the individual level raises legitimate concerns, and the framework is designed to work at the team and service level, not to attribute risk to individual engineers.

3.2 System Architecture

The framework operates in four stages: data ingestion, feature engineering, risk scoring, and alert generation. Figure 1 gives a high-level view of the pipeline. The first thing any system like this needs to do is listen and there is a lot to listen to. Data Ingestion pulls events from three places that most engineering teams already have running: version control systems like GitHub, GitLab, or Bitbucket, where every commit and pull request leaves a trace; CI/CD platforms like Jenkins, GitHub Actions, or ArgoCD, which record exactly how and when code moves toward production; and operational monitoring tools like Prometheus, Datadog, or the ELK stack, which capture what the system does once that code arrives. None of these sources speak the same language out of the box, so the pipeline normalizes them into a shared event schema before passing them downstream. From there, events flow into a central event bus Apache Kafka is the obvious choice here, mainly because it handles high-throughput streams without breaking a sweat and has a mature ecosystem around it, though teams already invested in other stream processing platforms can adapt the design without much friction.Feature Engineering processes the raw event stream into structured feature vectors. This is the most technically demanding stage of the pipeline and the one where most of the predictive value is created. Features are computed over rolling windows at three time scales: short-term (1 hour), medium-term (6 hours), and long-term (24 hours). The multi-resolution approach is important because different risk patterns manifest at different timescales a sudden spike in commit frequency is a short-term signal, while a sustained decline in code review thoroughness is a longer-term one.

Risk Scoring applies a prediction model to the current feature vector to produce an incident risk score between 0 and 1. We discuss model selection in Section 3.4. The risk score is updated continuously as new events arrive, allowing the system to respond to rapid changes in developer activity in near real time.

Alert Generation translates risk scores into human-facing notifications. Critically, alerts include not just the score but a structured explanation: which signals contributed most to the elevated risk, what developer actions drove those signals, and which services are most likely to be affected. This explainability layer is what separates the system from a black box detector and makes it actionable for on-call engineers.

3.3 Developer Signal Taxonomy

Through review of the empirical software engineering literature and analysis of publicly documented incident post-mortems, we identified four categories of developer signals that carry predictive weight:

Commit Dynamics captures how code is being written and pushed: commit frequency over rolling windows, average commit size in lines of code, the ratio of modified lines to added lines (a measure of how much existing code is being disturbed), and the proportion of commits occurring outside normal working hours. Late-night and weekend commits are not inherently risky, but they become meaningful when combined with other signals.

Pull Request Behavior captures the social and process dimension of code review: time from PR opening to merge, number of reviewers who approved, volume of review comments, whether the PR was merged by its own author, and the proportion of automated checks that passed before merge. Each of these reflects a dimension of review quality that the literature has linked to defect rates.

Deployment Patterns tracks how code moves from repository to production: deployment frequency, the time elapsed between the last merge and the deployment, rollback rate over the past 24 hours, and whether deployments are following standard pipeline paths or bypassing controls. Deployments that arrive faster than usual, or that take unusual pipeline paths, warrant closer attention.

Operational Feedback closes the loop between operations and development by incorporating signals from the production environment that reflect on developer actions: error rate changes in the 30 minutes following a deployment, alert volumes attributed to recent releases, and MTTR for incidents linked to code changes. These signals are retrospective, but they inform the model's understanding of which development patterns have historically led to problems in a given organization.

3.4 Model Selection

The choice of prediction model involves a genuine trade-off between accuracy and explainability. In our framework, we recommend starting with gradient-boosted decision trees (specifically XGBoost or LightGBM) for three reasons. First, they perform well on tabular feature data without requiring large training sets. Second, they are natively explainable through feature importance scores and tools like SHAP values, which align with our design principle of explainability. Third, they are computationally efficient enough to run inference in real time on a standard server.

Deep learning approaches — LSTMs for sequential signal modeling, or transformer-based architectures for capturing long-range dependencies may offer accuracy improvements as the system matures and training data accumulates. However, they are harder to explain, harder to debug when they fail, and require more data to train reliably. We suggest treating them as a future upgrade path rather than a starting point.

Table 1 summarizes the signal categories, their sources, and their expected relationship to incident risk based on existing literature.

Table 1: Developer Signal Categories and Incident Risk Relationship

Signal Category	Source System	Key Features	Risk Relationship
Commit Dynamics	Git / VCS	Frequency, size, churn rate, off-hours ratio	Large, high-churn commits correlate with defect introduction [3]
PR Behavior	GitHub / GitLab	Review depth, merge speed, author self-merge rate	Shallow review linked to higher fault rates [4]
Deployment Patterns	CI/CD Pipeline	Deploy frequency, rollback rate, pipeline bypass	High rollback rate signals unstable releases [5]
Operational Feedback	Monitoring / Alerts	Post-deploy error rate, alert volume, MTTR	Historical failure patterns improve future risk scoring [1]

4. Discussion

4.1 Why This Matters in Practice

The case for incorporating developer signals into incident prediction is ultimately a practical one. Most operations teams already have access to the raw data, it sits in GitHub, in Jenkins, in their deployment logs. What they lack is a systematic way to connect that data to operational risk in real time. DevSignal-AIOps is an attempt to provide that connection in a way that is transparent enough for operators to trust and specific enough to act on.

The most immediate practical benefit is lead time. Infrastructure signals tend to surface risk close to the moment of failure when a service is already misbehaving. Developer signals, by contrast, are available before the change even ships to production. A large, poorly reviewed pull request merging at 11 p.m. is a signal that something might go wrong in the next deployment cycle. Catching that signal hours earlier gives teams genuine options: delay the deployment, add a reviewer, run additional tests, or at least ensure that the right person is on call.

4.2 Limitations of the Framework

We want to be honest about what this framework does not do. It does not address infrastructure failures that have no developer antecedent hardware failures, cloud provider outages, unexpected traffic spikes. These categories of incidents require the infrastructure telemetry that existing AIOps systems already handle well, and DevSignal-AIOps is designed to complement, not replace, those systems.

The framework also assumes a reasonably mature DevOps practice. Organizations that lack structured deployment pipelines, that don't use pull requests, or that have irregular commit practices may find that the developer signals available to them are too sparse or too noisy to be reliably predictive. In those environments, improving the development process itself is probably a prerequisite to making this kind of monitoring worthwhile.

Finally, there is the question of scale. In large organizations with hundreds of teams, the relationship between individual developer actions and specific production services can become diffuse and hard to attribute. The framework handles this better when service ownership is clearly defined and when the codebase has strong modularity. In monolithic or loosely organized systems, the signal-to-noise ratio may be lower.

4.3 Privacy and Ethical Considerations

Monitoring developer activity raises questions that technical teams sometimes prefer to

skip over, but that matter both ethically and practically. If engineers feel that their work habits are being surveilled and used to judge them, trust breaks down, behaviour changes in ways that degrade signal quality, and the best people leave. This is not hypothetical it is documented in organizations that have deployed developer productivity metrics without adequate transparency [11].

Our framework is designed to operate at the team and service level, not the individual level. Risk scores are associated with services and deployments, not with named engineers. The alert output describes what happened (a large, minimally reviewed change was deployed to a critical service) without naming who was responsible. We consider this distinction non-negotiable, not a nice-to-have.

We also recommend that any organization deploying this framework involve engineering teams in the design process, be transparent about what signals are being collected and why, and establish clear policies about how the data can and cannot be used in performance evaluations.

5. Future Work

The most important next step for this framework is empirical validation. The conceptual foundation is solid, but the specific predictive value of different signal categories and the best way to combine them needs to be measured on real incident data. Public datasets from the AIOps Challenge competitions and from organizations like Alibaba Cloud that have released operational data provide a starting point, though ideally this would be validated in partnership with engineering organizations willing to share their commit histories and incident records.

A second direction is adaptive learning. Developer team practices change over time sprint rhythms shift, deployment cadences evolve, new engineers join and change the dynamics of code review. A static model trained on historical data will drift. Future work should explore online learning approaches that allow the model to adapt continuously as team behaviour evolves.

Third, we are interested in the visualization and interaction layer. A risk score is only useful if it is presented in a way that operators can act on quickly. We believe there is real value in a dashboard that shows not just current risk scores, but the trajectory of developer activity signals over time, annotated with deployment events and linked to historical incidents. Building and evaluating such an interface is a natural next step.

6. Conclusion

Production incidents don't just happen to systems. They happen because of the people building those systems and the traces of that human activity are richer with predictive information than the AIOps community has yet fully recognized. This paper has tried to make that case concretely, by proposing a framework that treats developer activity signals as a first-class input to incident prediction alongside the infrastructure telemetry that modern AIOps platforms already consume.

We are not claiming that watching pull requests and deployment logs will prevent all incidents. We are claiming that it can help detect a meaningful class of incidents earlier — the ones that follow from risky changes, rushed reviews, and deployment patterns that experienced engineers already recognize as dangerous. Giving operations teams more time to act on those patterns, rather

than simply better tools for cleaning up after them, seems like a worthwhile goal.

The framework presented here is a starting point. It is grounded in two mature bodies of research — AIOps and empirical software engineering that have been developing in parallel without enough cross-pollination. We hope this paper encourages researchers in both communities to look across the aisle, and to work together toward operational intelligence systems that are as aware of human behavior as they are of system behavior.

References

[1] Dang, Y., Lin, Q., & Huang, P. (2019). AIOps: Real-world challenges and research innovations. In Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp. 4–5. IEEE Press.

[2] Zhao, N., Chen, J., Wang, Z., Peng, X., Wang, G., Wu, Y., Zhang, C., Feng, Y., Zheng, Z., & Pei, D. (2020). Real-time incident prediction for online service systems. In Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 315–326. ACM.

[3] Nagappan, N., & Ball, T. (2005). Use of relative code churn measures to predict system defect density. In Proceedings of the 27th International Conference on Software Engineering (ICSE), pp. 284–292. ACM/IEEE.

[4] Gousios, G., Pinzger, M., & Van Deursen, A. (2014). An exploratory study of the pull-based software development model. In Proceedings of the 36th International Conference on Software Engineering (ICSE), pp. 345–355. ACM.

[5] Forsgren, N., Humble, J., & Kim, G. (2018). Accelerate: The Science of Lean Software and DevOps. IT Revolution Press. ISBN: 978-1942788331.

[6] Gartner Inc. (2017). Market Guide for AIOps Platforms. Gartner Research Report ID: G00325375.

[7] Du, M., Li, F., Zheng, G., & Srikumar, V. (2017). DeepLog: Anomaly detection and diagnosis from system logs through deep learning. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 1285–1298. ACM.

[8] Mockus, A., & Weiss, D. M. (2000). Predicting risk of software changes. Bell Labs Technical Journal, 5(2), 169–180. Wiley.

[9] Kim, G., Humble, J., Debois, P., & Willis, J. (2016). The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. IT Revolution Press. ISBN: 978-1942788003.

[10] Beyer, B., Jones, C., Petoff, J., & Murphy, N. R. (Eds.). (2016). Site Reliability Engineering: How Google Runs Production Systems. O'Reilly Media. ISBN: 978-1491929124.

[11] Kim, M., Zimmermann, T., DeLine, R., & Begel, A. (2016). The emerging role of data scientists on software development teams. In Proceedings of the 38th International Conference on Software Engineering (ICSE), pp. 96–107. ACM.

[12] Humble, J., & Farley, D. (2010). Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley Professional. ISBN: 978-0321601919.

