blueoptima™

Software Incident Risk Signatures: A Large-Scale Analysis of Maintainability Metrics and Stability

Understanding the factors and contexts associated with code-related software incidents

Whitepaper

© BlueOptima Limited 2005–2026. All Rights Reserved

blueoptima™

Software Incident Risk Signatures: A Large-Scale Analysis of Maintainability Metrics and Stability

Abstract

This whitepaper demonstrates that software incidents are not random events but predictable outcomes driven by measurable declines in code maintainability. As the adoption of Generative AI accelerates coding output, it has simultaneously introduced a "productivity-quality tradeoff" where increased productivity is coupled with a measurable drop in code quality. We substantiate this tradeoff through empirical evidence from 666,000 code revisions across thirteen major enterprises showing that incidents rose by 111% in Q2 2025 and 58% in Q3 2025 year-over-year, while average resolution times increased from 2 to 22 hours and from 7 to 31 hours, respectively, coinciding with a broader global decline in maintainability. The research presents a five-tier framework that combines static source code analysis with behavioral signals, such as developer tenure and code review friction. This approach identifies high-risk regions as the intersection of stagnant legacy code, poor maintainability, and prolonged review cycles, where the likelihood of incidents is elevated. We employ a Gradient Boosted Decision Tree model (XGBoost) to identify historical risk signatures associated with incident-linked commits. Within the historical dataset used for model fitting, the classifier identifies 74% of incident-linked commits, indicating that these revisions exhibit distinguishable maintainability, and review-process patterns. We demonstrate that software incidents become highly probable events when developers are required to modify unmaintainable code under high-pressure process conditions. Data shows that when an incident occurs in the most unmaintainable quartile of the codebase, the median **PR Time to Resolve** an incident reaches **65.2 hours**, compared to just 1.7 hours in maintainable sections; a 38-fold increase in resolution time. To mitigate these risks, the paper recommends implementing automated maintainability gates within CI/CD pipelines to help engineering teams identify and remediate high-risk code changes earlier in the delivery lifecycle. By catching structural defects at an early stage, organizations can prevent the slow accumulation of maintainability debt that eventually leads to software incidents and downtime.

"Good news, the code is shipping faster than ever."

Source: Image generated by prompting a Generative AI image generator (DALL·E) with the contents of this paper with some adjustment.

Contents

Abstract 2
Introduction 5
Background and Motivation 5
Research Objectives 7
Method 8
Target Definition & Event Isolation (The Pre-Incident State) 8
Foundational Metrics 8
A Five-Tier Feature Engineering Framework 9
Modelling Approach and Statistical Assumptions 11
Results 13
Technology Stack Composition in Incidents 13
Commits to Resolve vs. Files Modified 13
Commit Volume, Review Friction, and Incident Recovery Time 15
Incident Resolution Time: Top vs Bottom Quartile Maintainability 17
The GenAI Period (2023–Present): Quarterly Code-Level Incident Volume vs. Resolution Time 17
Global Risk Model Results 19
Global Risk Drivers 20
The Risk-Prevalence Matrix 20
Discussion 22
Risk Drivers 22
The Risk-Prevalence Matrix 25
1. High-Impact, High-Prevalence Risk Quadrant 25
2. High-Impact, Low-Prevalence Risk Quadrant 26
3. Low-Impact, High-Prevalence Risk Quadrant 26
4. Low-Impact, Low-Prevalence Risk Quadrant 26
Synthesis 26
Conclusion 28
Recommendations 29
1. Implement Automated Maintainability Gates in CI/CD 29
2. Expand Shared Ownership of High-Risk Code Areas 29
3. Require Touch-and-Clean Controls for Aged Files 29
4. Audit Low-Scrutiny Review Approval 30
5. Monitor the GenAI Productivity-Maintainability Trade-Off 30
References 31
Notices and Disclosures 32
About the Research Team 32
BlueOptima's Software Products 32
Appendix A: Mathematical Foundations of the Five-Tier Risk Framework 33
Appendix B: Model Design & Evaluation Methodologies 35

Appendix C: Coding Effort 37
What is Coding Effort? 37
How does it work? 37
Actual Coding Effort (ACE) 37
Billable Coding Effort (BCE) 37
Worked Example 38
Appendix D: Analysis of Relative Thresholds 39

 © BlueOptima Limited 2005–2026. All Rights Reserved

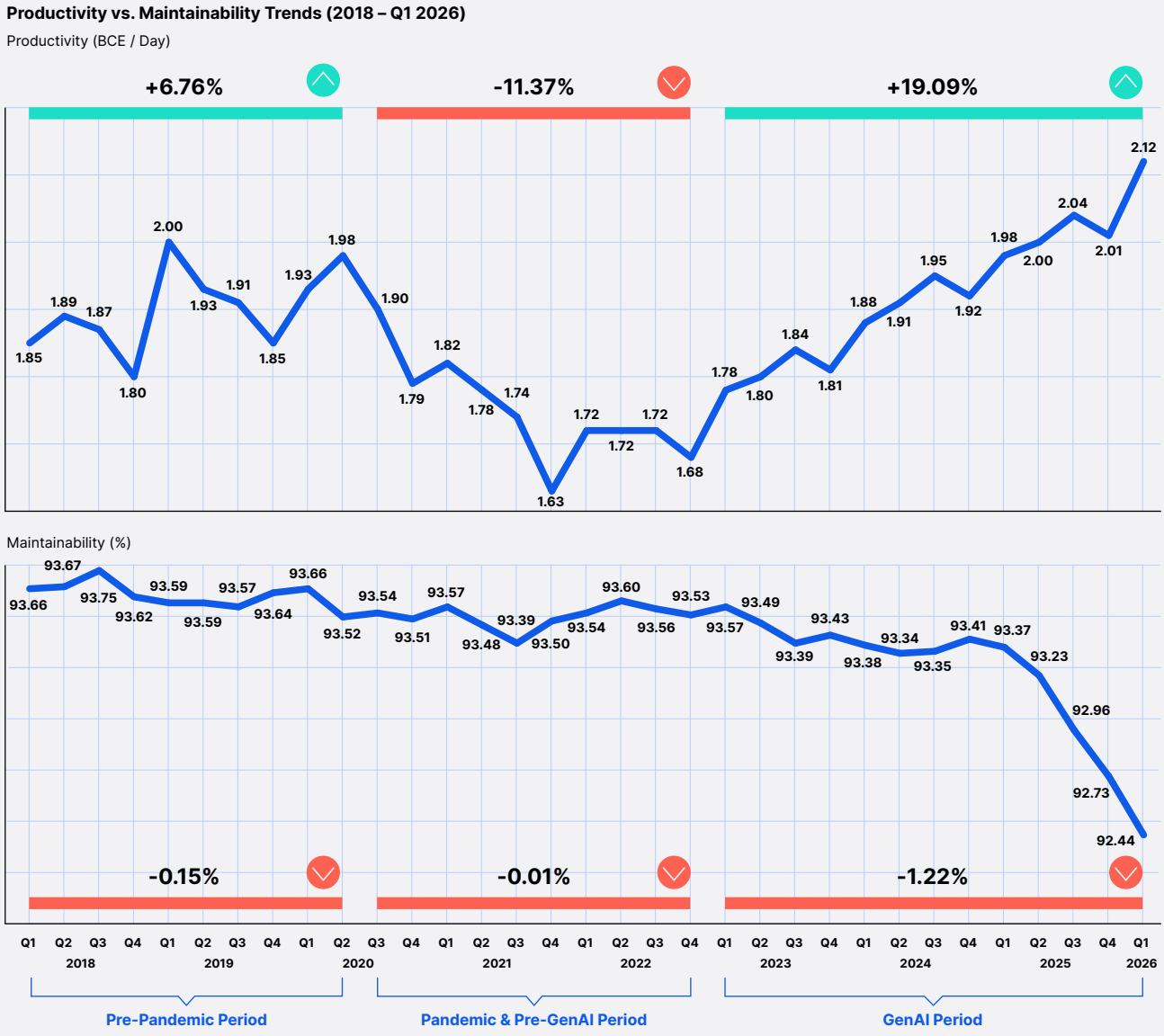
Introduction

Background and Motivation

The software development industry has experienced a turbulent decade, transitioning through distinct periods of stability, pandemic-induced disruptions, and the current era of Artificial Intelligence adoption.

The global COVID-19 pandemic triggered an abrupt transition to remote work in 2020, disrupting established collaboration workflows that impacted developer focus time and degraded the shared vision and understanding of software development initiatives (BlueOptima, 2023). This preceded the 2021 **'Great Resignation,'** a period of significant labor market churn that occurred alongside a notable decline in productivity (BlueOptima, 2022). During this period, industry-wide developer productivity declined by **11.37%**.

Figure 1
Global Productivity and Maintainability Trend from BlueOptima's Benchmark



 | © BlueOptima Limited 2005–2026. All Rights Reserved

Most recently, the widespread adoption of Generative AI (GenAI) from 2023 has introduced a disruptive new technology to the production of software. While GenAI has broadly offset pandemic-era decline through a **19.09%** increase in productivity, it has simultaneously introduced a **productivity-maintainability tradeoff:** as developer productivity increases, the codebase **maintainability has declined by 1.22%.** Notwithstanding the assistance provided by Generative AI in the development process, the quality of codebases has deteriorated. This has impaired maintainability, consequently making the codebases more difficult to understand.

Prior BlueOptima research provides the GenAI context for this study. A longitudinal analysis of software engineering performance from 2018 to 2025 identifies a GenAI-era productivity-maintainability tradeoff, with productivity rebounding by 14.29% from 2023 to 2025 while code quality declined by 0.26 percentage points (BlueOptima, 2025). A separate GenAI license-based study provides more direct adoption-level evidence, finding that active GenAI-licensed developers increased productivity by 4.74%, while aberrant code increased by 4.21%, compared with a 1.70% increase among active developers without GenAI licenses (BlueOptima, 2026). This paper builds on that context by examining whether maintainability decline, repository fragility, and review-process friction are associated with incident-linked revisions.

The primary motivation of this research is to quantify the downstream consequences of this tradeoff. This research statistically demonstrates that declining maintainability is a reliable antecedent of Production software incidents.

The Financial and Operational Imperative of Maintainability

To understand why it is critical to statistically demonstrate that declining maintainability is predictive of Production software incidents, organizations must first understand the remarkable economic toll of reactive engineering practices.

According to Google's Site Reliability Engineering (SRE) framework (Beyer et al., 2016), maintainability is a critical component of a system's **Error Budget**.

Poor software maintainability has repeatedly translated into material compliance failures, as illustrated by the cases of Knight Capital Group and Royal Bank of Scotland. In 2012, Knight Capital suffered a catastrophic trading incident when obsolete "dead code" and inconsistent deployment practices allowed a dormant feature to execute in production, generating erroneous trades and a loss of over $440 million within minutes; the Securities and Exchange Commission subsequently imposed a $12 million penalty, citing failures in software controls and risk management (Securities and Exchange Commission, 2013).

Similarly, the Royal Bank of Scotland experienced a major IT incident the same year due to a failed update in a fragile legacy batch-processing system that was difficult to maintain and insufficiently understood internally, leaving millions of customers unable to access banking services. This incident led to a £56 million fine from the Financial Conduct Authority for breaching operational resilience obligations (Financial Conduct Authority, 2014). A more recent example is TSB Bank, whose 2018 core banking migration failure further illustrates how maintainability deficiencies translate into compliance risk. The migration exposed deeply intertwined legacy systems and insufficiently testable integrations, resulting in prolonged outages, failed transactions, and instances of customers accessing incorrect data. Critically, the bank's inability to rapidly diagnose and remediate the issues highlighted low system transparency and weak change resilience, key indicators of poor maintainability. Regulators, including the Financial Conduct Authority, concluded that these failures reflected broader deficiencies in governance, risk management, and IT controls, leading to a £48.65 million fine (Financial Conduct Authority, 2022). Such examples of Production software incidents resulting in significant commercial losses for organizations are chronically underreported as organizations will avoid sharing the, potentially embarrassing, details of incidents. The examples above are well reported because transparency has been enforced by regulators.

Together, these cases demonstrate that poor maintainability, manifested in unmanaged legacy code, weak change controls, and limited system transparency, can directly undermine regulatory compliance by increasing the likelihood of uncontrolled failures and weakening an organization's ability to evidence effective governance over critical systems.

1. **The Macro-Economic Cost of Poor Quality**
   The financial implications of ignoring these early warning signals are significant. The cost of poor software quality in the U.S. alone was estimated at $2.41 trillion in 2022. Of that total, the burden of accumulated technical debt accounted for approximately $1.52 trillion (Krasner, 2022).

© BlueOptima Limited 2005–2026. All Rights Reserved

Industry analyses estimate that technical debt can consume 20% to 40% of an organization's entire technology estate value. Consequently, technical debt can divert software development resources away from new feature innovation, as developers lose a substantial share of their working time to debt-related activities such as additional testing, source-code analysis, and refactoring (Besker et al., 2019).

2. **The Escalation of AI-Generated Technical Debt (AITD)** This economic drain is now being hyper-accelerated by the Generative AI era. Recent research has extended the traditional concept of technical debt to **AI Technical Debt (AITD)**, recognizing that AI-assisted development introduces severe new forms of architectural and systemic instability (Recupito et al., 2024). The unmanaged adoption of GenAI risks "industrializing" the creation of this debt, as code is produced at a velocity that outpaces human review and structural validation. If organizations continue to rely on reactive management(i.e. waiting for a system to break before identifying the bad code), they face a future where the majority of engineering resources are consumed entirely by servicing AI-generated issues (Krasner, 2022), thereby stifling the very innovation GenAI promised to unlock.
3. **From Reactive Troubleshooting to Risk-Aware Engineering:** By showing that declining maintainability is strongly associated with historical software incidents, this research supports a shift from purely reactive troubleshooting toward risk-aware engineering management. The observed decline in source code maintainability must be interpreted as a measurable signal of accumulating technical debt and potential operational fragility.

If engineering leaders can identify the code, repository, and process conditions historically associated with incident-linked revisions, they can introduce targeted controls before risk accumulates further. These controls in CI/CD pipelines may include maintainability checks, additional review scrutiny for high-risk changes, and focused remediation of fragile legacy areas.

Research Objectives

The primary objective of this research is to empirically test the following core hypothesis: High code quality (i.e., high maintainability) correlates with reduced incidents. Conversely, lower code quality (i.e., lower maintainability) correlates with increased incidents.

This research combines static code maintainability data with human behavioral metadata, including developer tenure and peer review friction, to identify the conditions most strongly associated with software incidents. Ultimately, by statistically evidencing how declining maintainability triggers software incidents, this research provides engineering leadership a framework to proactively identify and remediate high-risk code changes before they are deployed into production

© BlueOptima Limited 2005–2026. All Rights Reserved

Method

To investigate the hypothesis that codebase maintainability inversely correlates with service incidents, we employ a quantitative, longitudinal machine learning approach. The methodology is structured into five sequential phases: target event isolation, metric extraction, behavioural feature engineering, algorithmic modeling, and actionable explanation.

Target Definition & Event Isolation (The Pre-Incident State)

A **Software Incident** is an incident recorded in an incident management system whose remediation required intervention by one or more software developers and resulted in a change to the application source code, evidenced by a commit linked to the incident record.

To accurately isolate software-driven incidents from general IT or network outages, we engineered a deterministic data linkage pipeline starting at the software development workflow layer.

BlueOptima integrates with several version control and task tracking tools such as GitHub, Jira and Azure DevOps to automatically capture development activities and provide performance insights.

To establish a definitive linkage for included records, the analysis scanned task-tracking and version-control metadata for known enterprise-specific incident identifier patterns. These were not generic keyword searches; they reflected the incident-key formats commonly used by organizations, including standard sequential identifiers such as INC1234, hyphenated project-key formats such as INC-1234, metadata-tagged variants such as INC-ABC-001234, and ITSM-originated references from systems such as ServiceNow, Jira Service Management, and BMC Helix ITSM. A revision was treated as incident-linked only where the identifier matched a known pattern and could be resolved to a corresponding incident or task record. This provides a deterministic link between the included revisions and incident-remediation activity, while recognising that commits without recorded incident identifiers fall outside the linked dataset.

By isolating only those incidents that required developer intervention and a subsequent repository merge, we created a canonical service incident dataset that strips away the noise of non-software related service incidents. This mapping provides a direct account of the resolution of the incidents. This links the incurred service downtime directly to the developer's exact code modifications and the pre-existing risk profile of the affected files, associated files, and the wider codebase, transforming the often qualitative accounts of service incidents described in incident tickets into precise, commit-level engineering evidence.

Crucially, to accurately measure systemic risk, our analysis isolates the exact files modified during an incident's remediation, but we strictly evaluate the state of those files and source code *immediately prior* to the incident, rather than their post-fix state. By rolling back the analysis to capture the code's condition right before it broke, we ensure the model estimates associations from the pre-incident state, rather than from post-fix code.

Foundational Metrics

To establish an objective statistical baseline, this study utilized two proprietary metrics developed by BlueOptima:

- **Coding Effort (CE):** Coding Effort is a measure of the average intellectual effort being delivered by a population of programmers delivering change into the source code of an application. This "effort" is a measure of work that reflects the relative difficulty of working on different source and configuration files, while taking into account the aspects of the programmer's work that are not under their control (i.e. the context within which the change is delivered). By evaluating and benchmarking every change made by all developers as they contribute their code through the software development process, Coding Effort accurately reflects intellectual effort invested in coding. For further details, please refer to Appendix – C: Coding Effort.
- **Analysis of Relative Thresholds (ART):** The Analysis of Relative Thresholds (ART) is a metric that evaluates the maintainability of source code and its ease of modification. This metric quantifies the level of difficulty associated with working on unfamiliar code. The ART enables the determination of the thresholds for each maintainability, readability, and accessibility static metric of a coding language based on data from enterprise codebases. The calculation and appropriate maintainability score are then calculated based on the severity of deviations

beyond these thresholds. For further details, please refer to Appendix – D: Analysys of Relative Thresholds.

A Five-Tier Feature Engineering Framework

To identify historical conditions associated with incident-linked commits, this research introduces a Five-Tier feature engineering framework.

This framework is built on the premise that a software incident does not happen in a vacuum; it is an emergent property of complex human behaviors interacting with degrading structural architectures. By leveraging metrics like Coding Effort (CE) to measure engineering output and Analysis of Relative Thresholds (ART) to quantify codebase maintainability, the framework layers 23 distinct variables into a hierarchical risk profile.

Below is a detailed breakdown of what each tier measures and the empirical justification for its importance.

Tier	Name	What it measures	Why it is important
Tier 1	**File-Level Maintainability Degradation (Localized Code Quality)**	It measures the immediate structural impact of the code files modified as part of an incident resolution commit. In practical terms, it assesses how the developer's changes affect the maintainability of the specific files involved, compared against the state of those same files before the incident date. This tier focuses on localized code maintainability signals, including the lifetime maintainability debt accumulated within the affected files. It helps distinguish whether a fix improved, preserved, or further degraded the maintainability of the code directly touched during remediation.	It is important because localized code quality is often where systemic risk first becomes visible. A single change in a critical file can either reduce future risk or quietly create the conditions for the next failure. There are two forms of risk this tier is designed to capture: **Active Technical Debt Injection:** If a developer pushes code that is highly unmaintainable into a previously maintainable file, they are actively injecting new technical debt into the system. This code may become a trap for the next developer who attempts to alter it. **Legacy Technical Debt Existence:** Conversely, if a developer pushes a relatively maintainable change, but the file itself has historically been the victim of highly unmaintainable changes (possessing a high lifetime maintainability deficit), the developer is forced to work within a very difficult, fragile codebase. Even a careful, well-intentioned change inside a highly degraded file carries a massive likelihood of shattering the surrounding logic.
Tier 2	**Systemic Repository Health**	Tier 2 expands the lens from the individual files modified during incident resolution to the broader health of the parent repository. While Tier 1 measures localized code quality, Tier 2 evaluates the structural condition of the repository as a whole, including its accumulated maintainability debt, the velocity at which maintainability is degrading, and the overall resilience of the codebase. This tier considers indicators such as Repository Lifetime Maintainability Debt, Legacy Decay Index, and overall Repository Maintainability. Together, these metrics show whether the repository is improving, stabilizing, or becoming increasingly fragile over time.	This tier contextualizes the environment receiving the code. A perfectly written for-loop pushed into a pristine microservice carries a fundamentally different risk profile than that same for-loop forced into an exhausted, heavily decayed legacy monolith. Poor repository-level metrics indicate that the repository's foundation is so brittle that even minor, well-intentioned code drops can trigger cascading, systemic fractures. It proves that perfect code in a broken system will still cause an incident.

Tier	Name	What it measures	Why it is important
Tier 3	File-Repository Maintainability Divergence	This tier calculates the statistical variance between the quality of the files that required fixes and the historical baseline of the repository it lives in (e.g., Maintainability GAP – File vs Repo). It isolates files that drastically diverge from the established architectural norms of their local environment.	If a repository is generally messy, reviewers may adjust their expectations. However, when a developer pushes a "rogue" file that is vastly more complex or unmaintainable than the repository's accepted norm, it creates a severe cognitive blind spot. This tier acts as an anomaly detection layer, highlighting code that violates local architectural standards and is therefore highly likely to behave unpredictably in production.
Tier 4	Cohort Relatives (Fairness Benchmarking)	This tier evaluates a repository's maintainability against a cohort of active peer codebases within the enterprise that are of similar size and complexity. Rather than assessing a project in isolation, it measures the project's relative deviation from the company's own baseline for that specific "weight class" of software. Because the cohort is drawn from active repositories, the benchmark should be interpreted as an operational comparison against systems currently under development or maintenance, rather than as a census of all repositories historically created by the enterprise. Repositories that have been retired, archived, or rewritten may not contribute to the active peer baseline.	Cohort Benchmarking answers the ultimate question: "Is this system hard to maintain because the business logic requires it, or is the team working on it simply producing worse code than their peers?"
Tier 5	Human & Process Control (Behavioral Metadata)	While Tiers 1 through 4 analyze the mathematics of the source code, Tier 5 measures the human delivery mechanism. It evaluates who is making the changes (tenure and experience), what they are being forced to touch (ancient vs. modern files), and the actual rigor of the organization's CI/CD security checkpoints (how long Pull Requests sit open, and whether code is being genuinely scrutinized or merely rubber-stamped). For clarity, PR duration metrics are separated into two distinct constructs. Pre-Incident Repository Pull Request Time to Close measures the review duration of code changes delivered to the same repository before the incident fix, capturing recent repository-level process friction. Incident Recovery Pull Request Time to Close measures the PR cycle associated with resolving the incident after it has occurred. Only the former is used as a pre-incident contextual signal in the risk model.	This is the most critical tier because it tracks the delivery mechanism of the incident. A structural defect only becomes an incident if human review processes fail to stop it. Tenure metrics indicate whether organizations are concentrating high-risk legacy modifications among a small group of senior engineers. Process Metrics (like PR Time to Close) expose "architectural drift," where stale branches sit open for weeks, silently injecting defects when finally merged into an evolved branch.

Modelling Approach and Statistical Assumptions

Primary Data: The risk models and insights detailed in this research are trained on a cross-industry dataset sample from the **BlueOptima Global Benchmark**. The scope encompasses thirteen enterprises across the Financials, Technology, Healthcare, and Consumer Cyclicals sectors, as classified by The Refinitiv Business Classification[1]. The modeling dataset comprises more than 666,000 source code revisions drawn from thirteen enterprises, including incident-linked revisions and sampled non-incident development activity. The global risk drivers model was trained using data spanning Q1 2023 through Q4 2025. Enterprise-level contribution is not expected to be uniform, since participating organizations differ in software-estate size, repository count, developer population, and development throughput. Higher contribution volume from larger enterprises therefore reflects a greater volume of observable engineering activity, rather than implying greater intrinsic incident vulnerability. To assess representativeness, the dataset was reviewed across enterprise, and temporal dimensions. This review indicated that the modeling sample was not concentrated in a small number of contributors, although contribution volume varied naturally with enterprise scale. Aggregate file-type composition is reported in the results section, and temporal coverage is reflected in the longitudinal analyses.

Outlier Handling (Winsorization): Software engineering telemetry, particularly behavioral metrics like review lifecycles and file age in review, frequently suffers from extreme, right-skewed outliers. To process this data without discarding valuable anomaly signals, the Five-Tier Feature Engineering Framework applies 1st and 99th percentile Winsorization across all contextual features. Unlike data trimming, which deletes extreme edge cases that may correlate with incidents, Winsorization caps extreme outliers at a statistical ceiling. This preserves the 'extreme risk' signal for the algorithm while preventing severe statistical distortion in the SHAP log-odds calculations.

Model Selection for Historical Risk Classification: Initial iterations of this research explored using Logistic Regression to calculate linear Odds Ratios. However, it became clear that linear models fundamentally failed to capture the variation explained by human behaviors. We transitioned the modelling to a gradient-boosted decision tree framework. This was chosen because Gradient Boosting Tree-based algorithm utilizes a leaf-wise tree growth strategy that excels at isolating non-linear feature interactions (e.g., "If PR Scrutiny is Low AND Tenure is High AND Repo Fatigue is High → Risk is Critical"). It resolves cultural contradictions by finding localized, highly specific decision boundaries.

Because incident-linked commits are rare relative to ordinary development activity, the modeling dataset was constructed using negative-class down-sampling. All incident-linked commits were retained as positive observations. Non-incident-linked commits were then randomly sampled, using a fixed random seed, up to a maximum of ten non-incident observations for each incident-linked commit. This produces a maximum 1:10 positive-to-negative class ratio, corresponding to an approximate 9.1% positive-class prevalence in the modeling sample where sufficient non-incident observations are available. This design preserves substantial class imbalance while ensuring that incident-linked revisions are evaluated against a broad sample of ordinary repository activity.

Model Evaluation: Evaluating a machine learning model for software incidents requires methodologies specifically designed for highly imbalanced datasets. Because commits causing incidents represent a low proportion of all commits made, reliance on standard accuracy measures becomes subject to an accuracy tradeoff (He & Garcia, 2009). A baseline model could simply approve every commit, achieve 99.9% accuracy, and completely fail its operational purpose. To evaluate the model as a historical risk-classification diagnostic, we report four metrics commonly used in Software Defect Prediction (SDP) and rare-event classification: Recall, Precision, F1-Score, and AUC-ROC.

Recall is particularly important in this setting because false negatives represent incident-linked revisions that the model fails to identify. From an operational risk perspective, the cost of missing a potentially high-risk change is materially greater than the cost of subjecting an additional change to review. Precision, however, must be interpreted in light of the sampled modeling design. The model is evaluated on a historical classification dataset in which all incident-linked commits are retained and non-incident-linked commits are down-sampled to a maximum 1:10 positive-to-negative ratio. As a result, precision reflects the proportion of model-flagged commits that are incident-linked under this sampled class balance.

1 The classification of organizations into Economic Sectors, Industry Groups, and Industries is done using The Refinitiv Business Classification (TRBC).

For further details, please refer to Appendix B: Model Design & Evaluation Methodologies.

Model Output Interpretation: Before examining the specific drivers of system incidents, we must first establish how the model quantifies risk. To ensure these findings are immediately actionable for engineering leadership, the XGBoost model's outputs have been translated from raw statistical probabilities into Risk Multipliers. Risk multipliers describe modeled historical associations with incident-linked revisions. Examples of how these multipliers are used to aid interpretation are set out below:

The Neutral Baseline (1.0X): Every code deployment inherently carries a baseline probability of causing an incident. For this analysis, an average, routine code change is considered our neutral baseline, represented as a 1.0X risk multiplier. At 1.0X, a specific code characteristic adds zero additional risk to the deployment.

The Risk Multiplier (e.g., 1.24X): When the model identifies a structural or behavioral anti-pattern, it calculates how much that trait compounds the baseline risk. For example, if a characteristic is assigned a 1.24X Risk Multiplier, it indicates that a commit exhibiting this trait is 24% more likely to trigger an outage compared to our neutral baseline.

By framing technical debt as a risk multiplier, we shift the conversation to quantifiable operational threats. A multiplier of 1.24X, 1.50X, or 2.0X acts as a glaring warning sign, explicitly measuring the elevated probability of incident introduced by specific development practices.

© BlueOptima Limited 2005–2026. All Rights Reserved

Results

First, an exploratory analysis of the observed incidents is conducted. This exploratory analysis establishes the baseline operational dynamics of the portfolio, revealing the raw distributions of code changes for delivering a fix.

Technology Stack Composition in Incidents

This analysis isolates application-level software incidents. It includes only incidents for which the root cause was traced directly to the codebase and remediation required a code revision. It excludes external disruptions, including standalone network latency, hardware degradation, and third-party upstream or downstream failures.

Core backend logic, represented by Java, C#, and Kotlin, accounts for the largest share of incident-resolution effort: 51% of the total, equivalent to more than 1.07 million Coding Effort (CE) units. This concentration is consistent with the central role of backend logic in application execution, where defects can propagate across dependent services and produce wider system incidents.

Infrastructure, configuration, and build artefacts, including Terraform, JSON, and YAML, account for 22% of total effort, or nearly 474,000 CE units. This indicates that deployment and configuration issues constitute a material source of application-level incident remediation, alongside conventional code defects.

User-facing technologies, including TypeScript, JavaScript, and HTML, account for 17% of remedial effort, equivalent to 367,000 CE units. Database scripting, represented by SQL, accounts for the remaining 10%, or more than 204,000 CE units, indicating a smaller but concentrated area of incident-resolution effort.

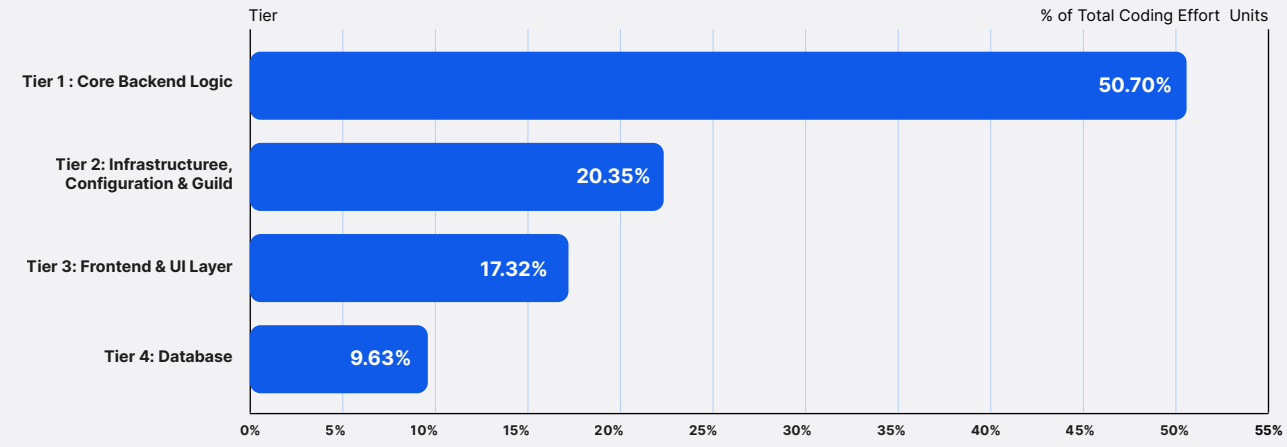
At the file-type level, remediation effort is highly concentrated. Four extensions, Java, C#, TypeScript, and SQL, account for nearly 70% of the total global effort required to resolve application-level incidents.

Commits to Resolve vs. Files Modified

Figure 4 shows the distribution of distinct code commits, defined as source code revisions, associated with individual incident tickets through to final resolution. The distribution is strongly concentrated at the lower end, with more than 66% of incident tickets resolved in four commits or fewer. However, the extended tail of the distribution indicates that a smaller subset of incidents requires substantially greater remediation activity.

In these cases, individual incident tickets require from 10, to more than 40 distinct commits before resolution. This pattern suggests that, while most incidents are resolved through a limited number of revisions, a material minority involves prolonged and iterative remediation.

Figure 2
Percentage of incident resolution effort attributable to the technology stack



 © BlueOptima Limited 2005–2026. All Rights Reserved

Figure 3
Total incident resolution effort by file type

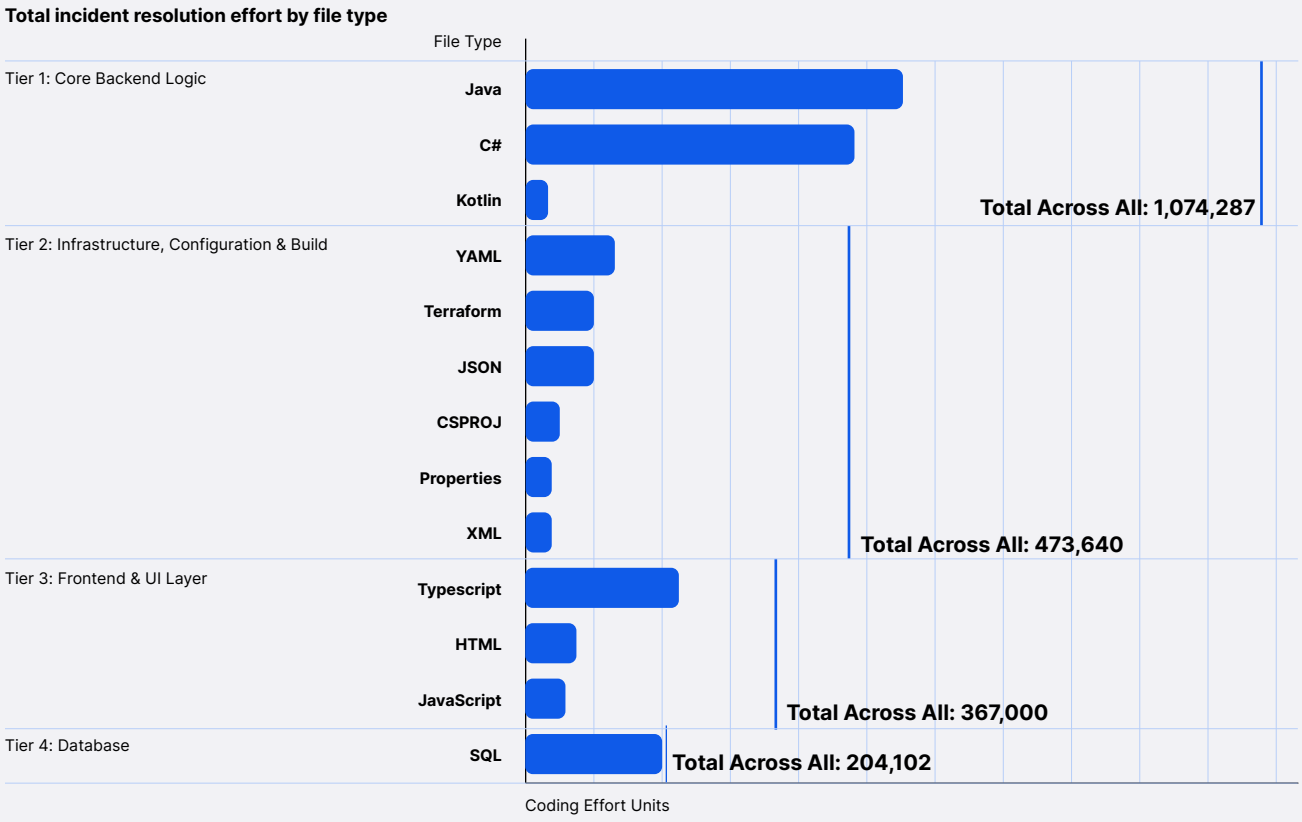
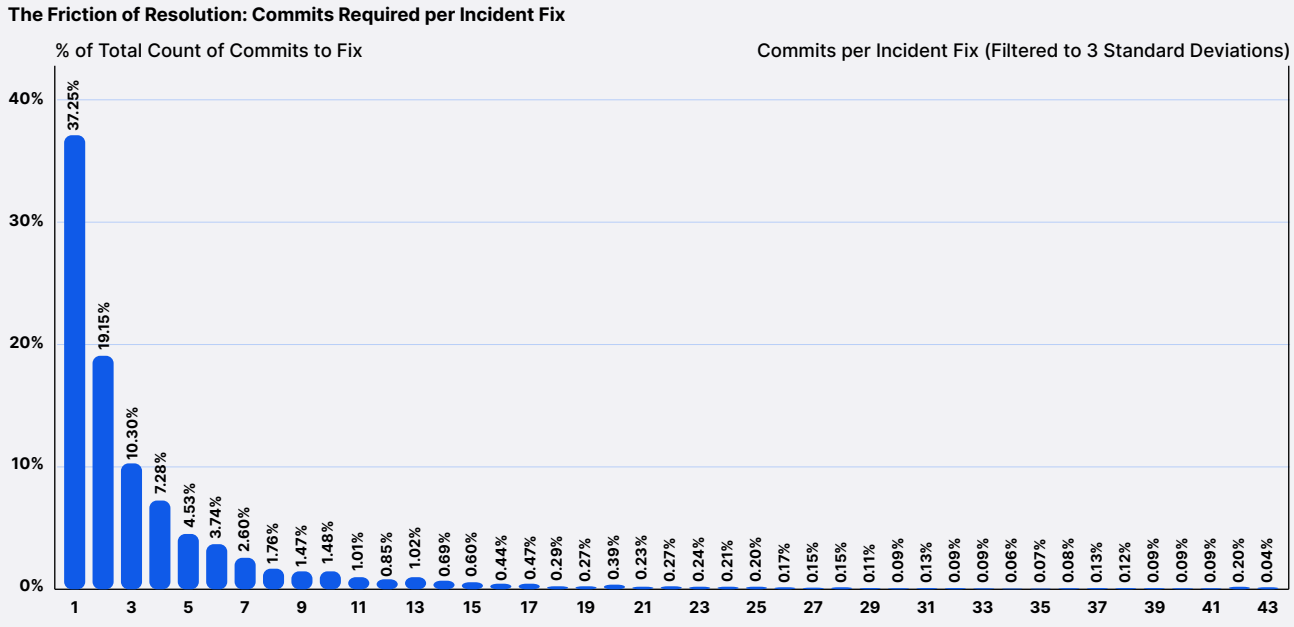


Figure 4
Distribution of commits required per incident fix, showing the percentage of total fixes by commit count



 © BlueOptima Limited 2005–2026. All Rights Reserved

Figure 5 shows that the file footprint of most incident fixes is highly constrained. Nearly 73% of all incidents are resolved by modifying only one or two files. This indicates that many application-level failures are remediated through changes to a limited area of the codebase rather than through broad structural modifications across the repository.

This pattern suggests that incidents are frequently associated with localised defects, such as flawed logic, configuration errors, or database query issues. When considered alongside the commit-volume distribution in Figure 4, the data also indicates a source of operational friction: some incidents require multiple revision attempts despite involving only a small number of files. This is consistent with iterative debugging and may indicate constraints in code maintainability, diagnostic clarity, or change validation.

Commit Volume, Review Friction, and Incident Recovery Time

Figure 6 evaluates operational friction during incident recovery by comparing Pull Request (PR) closure time with the number of commits required to resolve each incident ticket. The methodology first identifies the maximum PR time to close for each unique incident. This captures the longest repair cycle associated with the incident, measured from the start of the fix to the final stabilising merge. The median of these maximum values is then calculated to establish a representative baseline for recovery performance. The Incident Recovery PR Duration here is distinct from the pre-incident repository-level PR duration used in the risk model. Recovery PR duration captures the time required to review and merge code changes that resolve an incident after it has occurred; it is an outcome of the remediation process.

The data shows a positive relationship between commit volume and incident resolution time. As the number of commits required for resolution increases, the median PR closure time also rises. In the context of an active incident, this pattern indicates that repeated revisions can extend recovery duration and increase operational exposure.

Higher commit volumes are unlikely to reflect an intended remediation strategy in most cases. They are more plausibly interpreted as evidence of operational friction during the recovery process. Three mechanisms are particularly relevant.

First, complex or difficult-to-read code can limit diagnostic clarity. When engineers cannot rapidly identify the root cause or determine the correct remediation path, the fix process may shift towards iterative debugging.

Second, multi-commit fixes are harder to review and validate. They can generate additional feedback cycles, changes in scope, and reviewer scrutiny, particularly where reviewers must ensure that a rapid fix does not introduce secondary failure modes.

Third, in systems with weaker maintainability characteristics, restoring normal operation may require a sequence of adjustments across the affected logic rather than a single, targeted correction.

Figure 7 extends this analysis by measuring the median coding effort associated with incident fixes across commit-count bands. The data shows a non-linear increase in effort as commit volume rises, with a particularly marked escalation once fixes exceed 20 commits.

Between the 21 to 40 commit band and the 40+ commit category, median coding effort increases by 244%, rising from 37.62 hours to 129.40 hours. This increase indicates that high-commit incident fixes are not simply larger versions of ordinary fixes. They represent a materially different remediation profile, characterised by substantially greater engineering effort and likely higher diagnostic and validation complexity.

At this level of commit volume, the remediation process is less consistent with a narrow, localised repair and more consistent with repeated correction, regression management, or broader changes to stabilise the affected system behaviour. The data does not establish the specific technical cause in each case, but it does indicate that fixes requiring 40 or more commits impose a disproportionate engineering burden.

Taken together, Figures 6 and 7 show two related dimensions of incident cost. Figure 6 captures recovery exposure, measured by the time required to close the PR and complete the stabilising merge. Figure 7 captures engineering effort, measured by the coding effort hours required to deliver the fix. In lower-commit incidents, both dimensions remain comparatively constrained, which is consistent with a more maintainable remediation environment.

As commit count increases, however, both resolution time and engineering effort escalate. This indicates that the cost of an incident is not limited to service disruption. It also includes the engineering capacity absorbed by diagnosis, revision, review, validation, and stabilisation. In the 40+ commit category, the median effort of 129.40 hours

© BlueOptima Limited 2005–2026. All Rights Reserved

Figure 5
Distribution of files modified per incident fix, showing the percentage of occurrences by file count

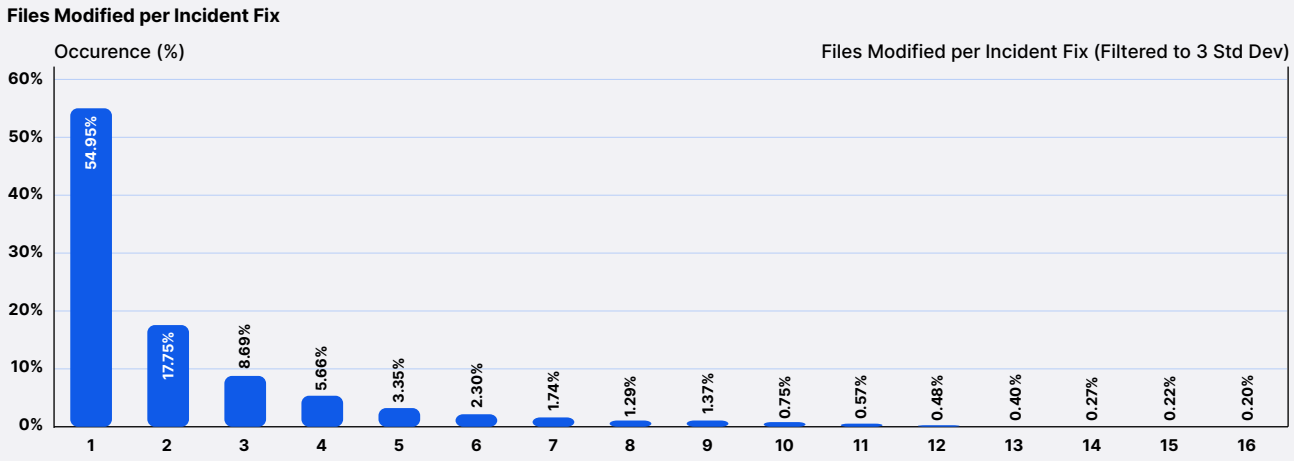


Figure 6
Median incident resolution time (PR time to close) by number of commits per fix

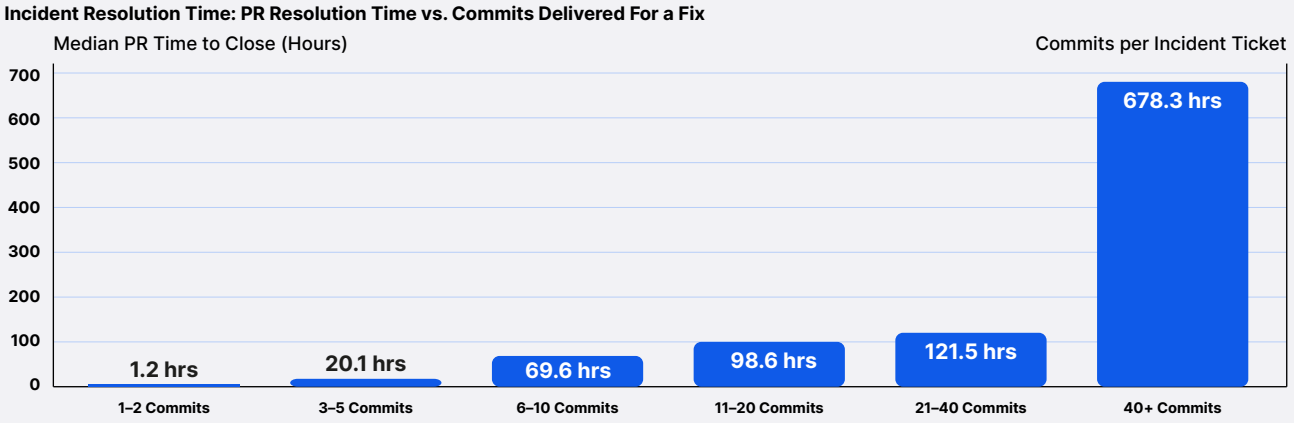
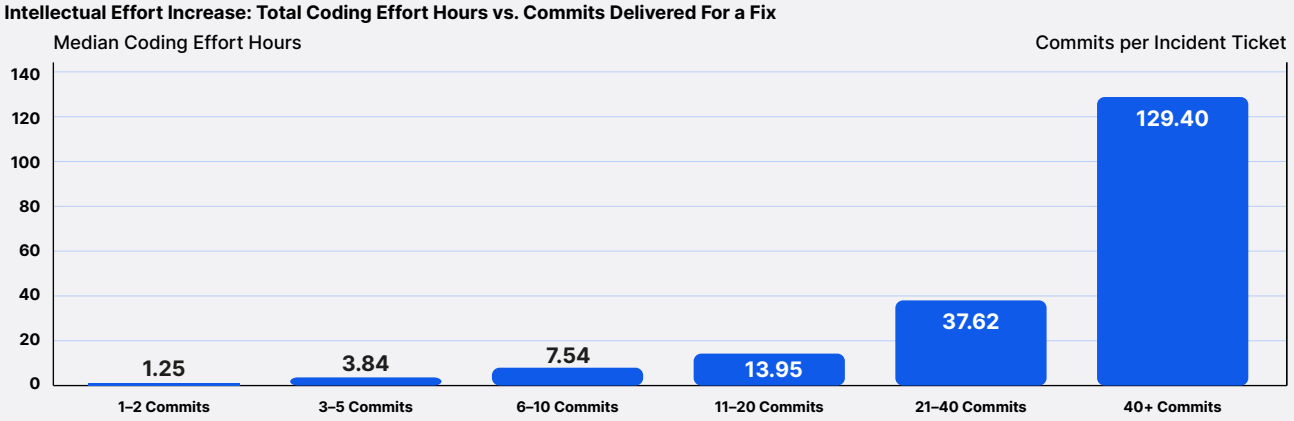


Figure 7
Median coding effort, in hours, required per incident fix by number of commits. Effort increases non-linearly as commit count rises, with a sharp increase beyond 20 commits



illustrates the scale of this burden relative to lower-complexity incident fixes.

Incident Resolution Time: Top vs Bottom Quartile Maintainability

Figure 8 compares the median incident resolution time[2] for incidents occurring in the highest and lowest maintainability quartiles of the codebase. Specifically, it compares incidents associated with the top 25% of code by maintainability against those associated with the bottom 25%.

The results indicate a substantial difference in recovery performance. Incidents located in the least maintainable quartile experience materially longer Pull Request (PR) closure times, suggesting that low maintainability introduces friction during diagnosis, remediation, review, and approval.

Incidents involving the least maintainable file quartile have a median Incident Recovery PR Duration of 65.2 hours, compared with 1.7 hours for the most maintainable quartile, a 38-fold observed difference. This should be interpreted as a maintainability-stratified recovery result rather than a single-variable causal estimate. Low-maintainability files often embody the very conditions that make recovery difficult: accumulated complexity, architectural coupling, reduced readability, historical change burden, and concentrated expertise. The value of maintainability as a measure is precisely that it aggregates these structural impediments into an operationally meaningful signal. The result therefore supports the conclusion that incidents in low-maintainability areas are associated with materially higher recovery friction and operational fragility.

The GenAI Period (2023-Present): Quarterly Code-Level Incident Volume vs. Resolution Time

Figures 9a and 9b examine code-level incident trends across the GenAI period, comparing incident volume[3], resolution time, and remediation effort from early 2023 through Q1 2026. Figure 9a presents the full quarter-year sequence, enabling a chronological view of changes in incident frequency, recovery duration, and coding effort. Figure 9b reorganises the data by calendar quarter, with colour-coded bars representing incident volume for each quarter-year and the dashed line representing resolution time. Together, the two views distinguish the underlying time-series trend from year-over-year quarterly variation.

The data shows a divergence between incident frequency, recovery duration, and the engineering effort required to produce remediation code. Incident volumes increased materially over the observed period. Q1 baseline volumes rose by 201%, from 119 incidents in 2023 to 359 incidents in 2026. The increase was also pronounced in 2025, with incident volume rising by 111% in Q2 2025 compared with Q2 2024, and by 58% in Q3 2025 compared with Q3 2024.

Resolution times increased in parallel. In 2023, resolution times were generally low after an elevated first quarter, falling from 13 hours in Q1 to 1, 2, and 3 hours in the subsequent quarters. In later years, however, resolution times rose substantially. By early 2025, average resolution time had reached a peak of 44 hours. Q2 2025 rose to 22 hours, compared with 2 hours in Q2 2024, while Q3 2025 reached 31 hours, compared with 7 hours in Q3 2024.

At the same time, the effort required to write the code changes associated with incident remediation declined. This creates a notable operational divergence: engineers appear to be spending less time producing the patch code, while the end-to-end time required to deploy and close those fixes has increased.

The 2025 increase in incident frequency coincides with a decline in codebase maintainability, indicating that systems became harder to maintain during the same period. The data does not by itself establish causality, but it is consistent with a shift in the principal recovery constraint. The bottleneck appears less likely to be code production speed and more likely to be the downstream process of review, validation, and approval.

This interpretation is consistent with a maintainability-driven review burden. As codebases become more complex or more tightly coupled, reviewers may require additional time to assess whether a proposed fix is safe, complete, and unlikely to introduce secondary failures. In this context, longer resolution times may reflect increased review friction rather than slower developer implementation.

2 In calculating recovery timelines, we measure the **total elapsed clock time** without adjusting for standard working hours or weekends. This approach is intentional: as long as an incident remains unresolved, the application remains in a state of **active risk exposure**.

3 To mitigate the risk of instrumentation artefacts, incident records were identified using a consistent linkage methodology throughout the observation period. Accordingly, the analysis should be interpreted as a time-series of linked code-level incidents.

Figure 8
Comparison of median incident resolution time (PR time to close) between the least and most maintainable codebases

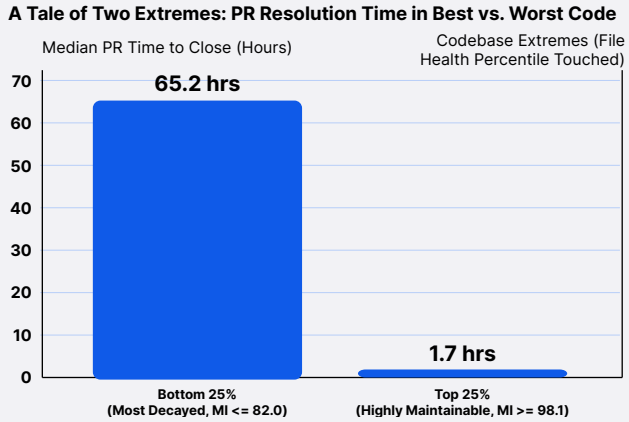


Figure 9a
Quarterly code-level incident volume, resolution time, and remediation effort by quarter-year

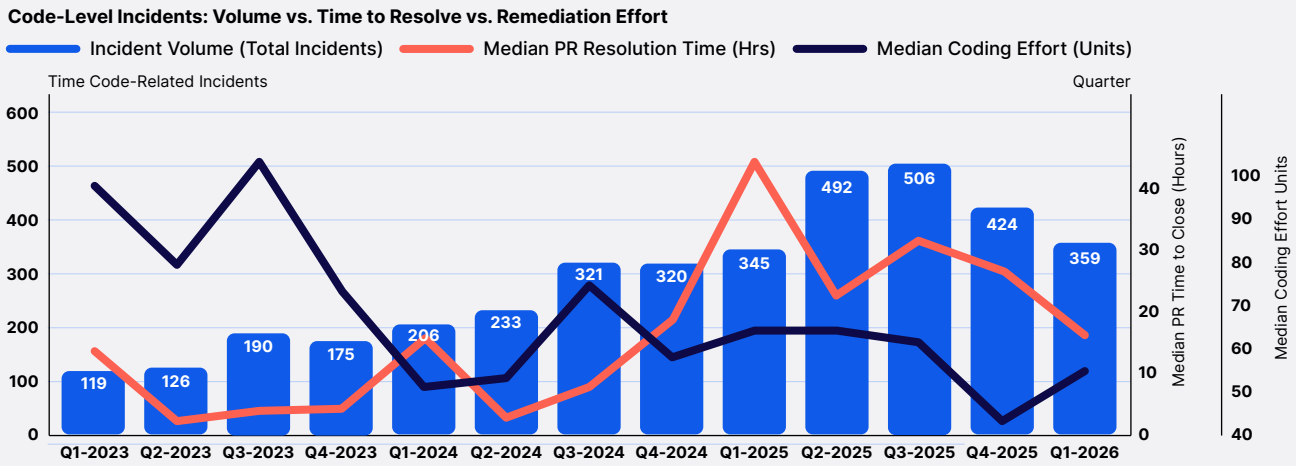
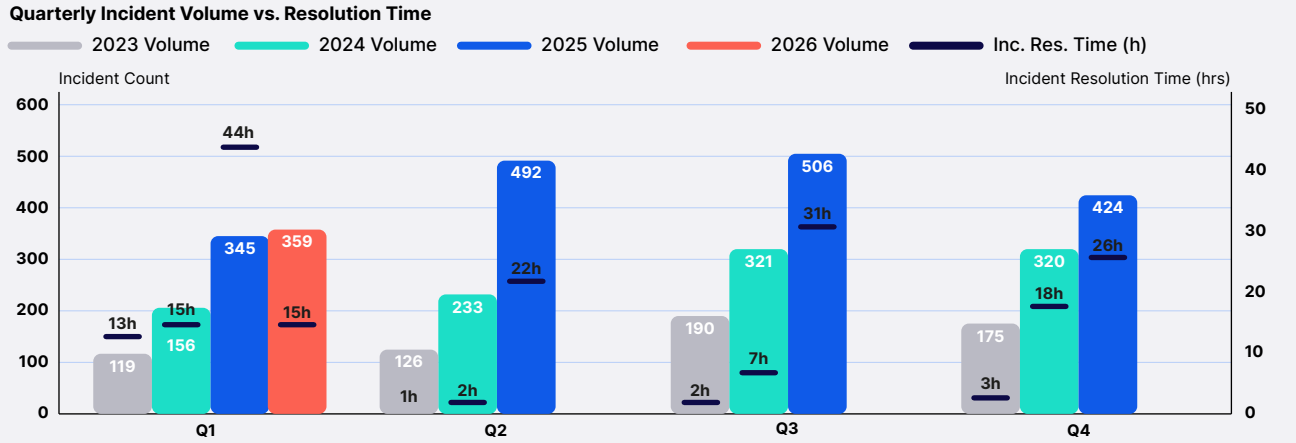


Figure 9b
Quarter-level comparison of incident volume and resolution time across years



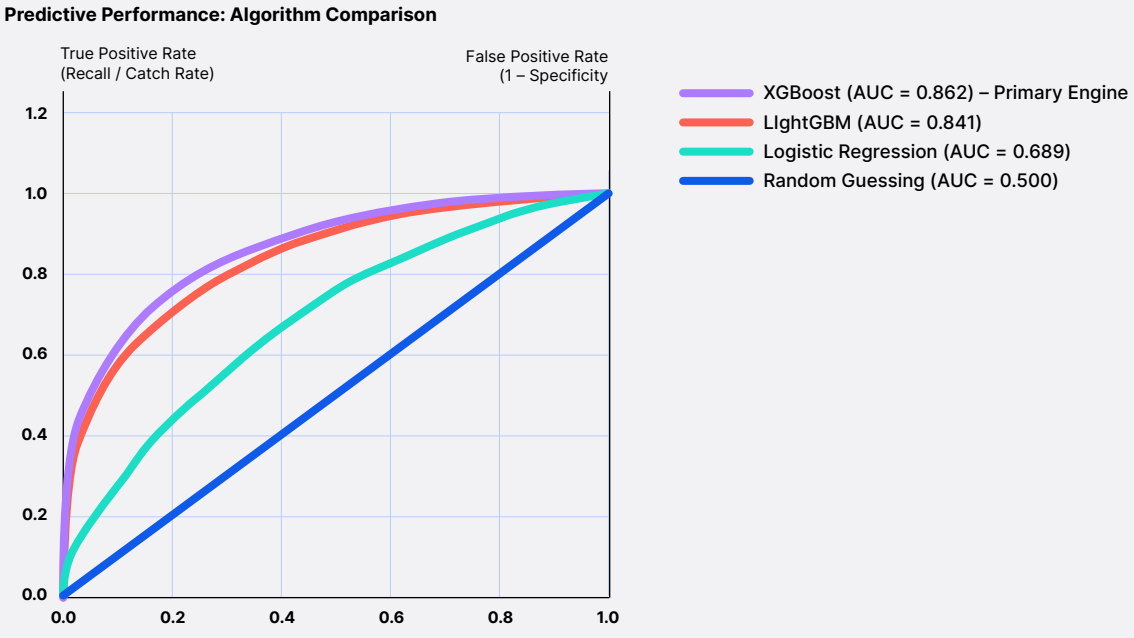
Global Risk Model Results

The risk model was trained on a historical dataset comprising more than 666,000 source-code revisions associated with incident remediation across the enterprise portfolio from Q1 2023 through Q4 2025. Because incident-inducing revisions represent an infrequent but operationally significant positive class, three classification algorithms were trained and evaluated: XGBoost, LightGBM, and Logistic Regression.

XGBoost produced the strongest overall performance and was selected as the preferred model for identifying and ranking code-level incident risk.

Model	AUC-ROC	F1-Score	Recall	Precision
XGBoost	**0.86**	0.40	**0.74**	0.28
LightGBM	0.84	0.38	0.72	0.26
Logistic Regression	0.69	0.25	0.57	0.16

Figure 10
Receiver Operating Characteristic (ROC) curves comparing predictive performance across algorithms



 | © BlueOptima Limited 2005–2026. All Rights Reserved

The modelling objective prioritised recall, reflecting the operational asymmetry between false negatives and false positives. Missing a true incident-inducing change can result in material production impact, while reviewing a false positive typically imposes a comparatively lower cost on engineering teams. The objective, therefore, was to maximise detection coverage while maintaining a level of alert volume that remains usable for code review.

The XGBoost model produced the following performance metrics[4]:

Recall: 0.74.
The model identifies 74% of true incident-inducing revisions. Operationally, this means that approximately three out of every four critical code changes associated with future incidents would be flagged before deployment.

Precision: 0.28.
Of the revisions flagged as high risk, 28% are true positives. In practical terms, approximately one in four flagged changes corresponds to an actual incident-inducing revision.

AUC-ROC: 0.86.
The model demonstrates strong discriminatory performance in ranking risky and non-risky revisions. An AUC-ROC of 0.86 indicates that the model is effective at prioritising higher-risk changes for review relative to lower-risk changes.

Taken together, these results indicate a model design suited to operational risk screening rather than definitive fault classification. The model provides broad detection coverage, identifying approximately three out of every four incident-inducing revisions, while requiring reviewers to examine roughly four high-risk flags to identify one true positive. This represents a practical trade-off for environments in which the cost of an undetected production incident is substantially higher than the cost of additional review.

Global Risk Drivers

SHAP analysis was used to quantify the relative contribution of pre-incident features to the global incident risk model. Figure 11 reports these features as relative risk multipliers against a neutral baseline of 1.00x. The features are grouped into five tiers: file-level maintainability degradation, systemic repository health, structural divergence, cohort relatives, and human and process factors.

Maintainability-related variables constitute a central component of the model's risk signal. At the local file level, maintainability on the fix files and maintainability lifetime deficit on the fix files are each associated with a 1.17x risk multiplier, while maintainability deficit from the previous commit is associated with a 1.06x multiplier. At the repository level, repository lifetime maintainability deficit shows a 1.24x multiplier, the largest maintainability-related effect in the model.

Additional repository-level maintainability features also show positive associations with incident risk. Repository legacy decay index is associated with 1.12x multiplier, while repository maintainability slope and repository maintainability are each associated with a 1.02x multiplier.

The largest overall risk is associated with PR time to close, at 1.34x. Here, PR time to close measures recent repository-level review duration before the incident fix, not the duration of the remediation PR. It should be interpreted as a process-friction signal associated with incident-linked revisions, not as evidence that longer PR duration causes incidents. Other prominent risk-increasing features include tenure at 1.26x and raw file age at 1.22x. These process-related variables sit alongside the maintainability measures, indicating that the model's risk signal is distributed across both code-state and delivery process features.

The Risk-Prevalence Matrix

The risk-prevalence matrix maps each factor's impact, measured by its risk multiplier, against its observed prevalence, measured as percentage presence in historical incident linked revisions. This provides a two-dimensional view of the top incident-associated metrics, separating factors by both relative risk and frequency of occurrence.

The matrix is constructed using portfolio-wide mean values as reference points for both dimensions. This matrix is restricted to the ten most impactful metrics. Their distribution across the matrix shows whether the leading risk-associated factors are primarily frequent portfolio-wide conditions, less frequent but higher-impact conditions, or lower-impact factors with broader occurrence. This provides a descriptive view of how incident-associated risk is distributed across impact and prevalence.

4 These results are reported against a sampled historical classification dataset in which all incident-linked commits are retained and non-incident-linked commits are down-sampled to a maximum 1:10 positive-to-negative ratio. The resulting positive-class prevalence is approximately 9.1% where sufficient non-incident commits are available. The reported precision of 0.28 should therefore be read against this sampled base rate.

Figure 11
Global incident risk drivers segmented by tier, showing relative risk multipliers against a 1.00x baseline. Factors above 1.00x increase observed incident risk, while factors below 1.00x reduce observed incident risk

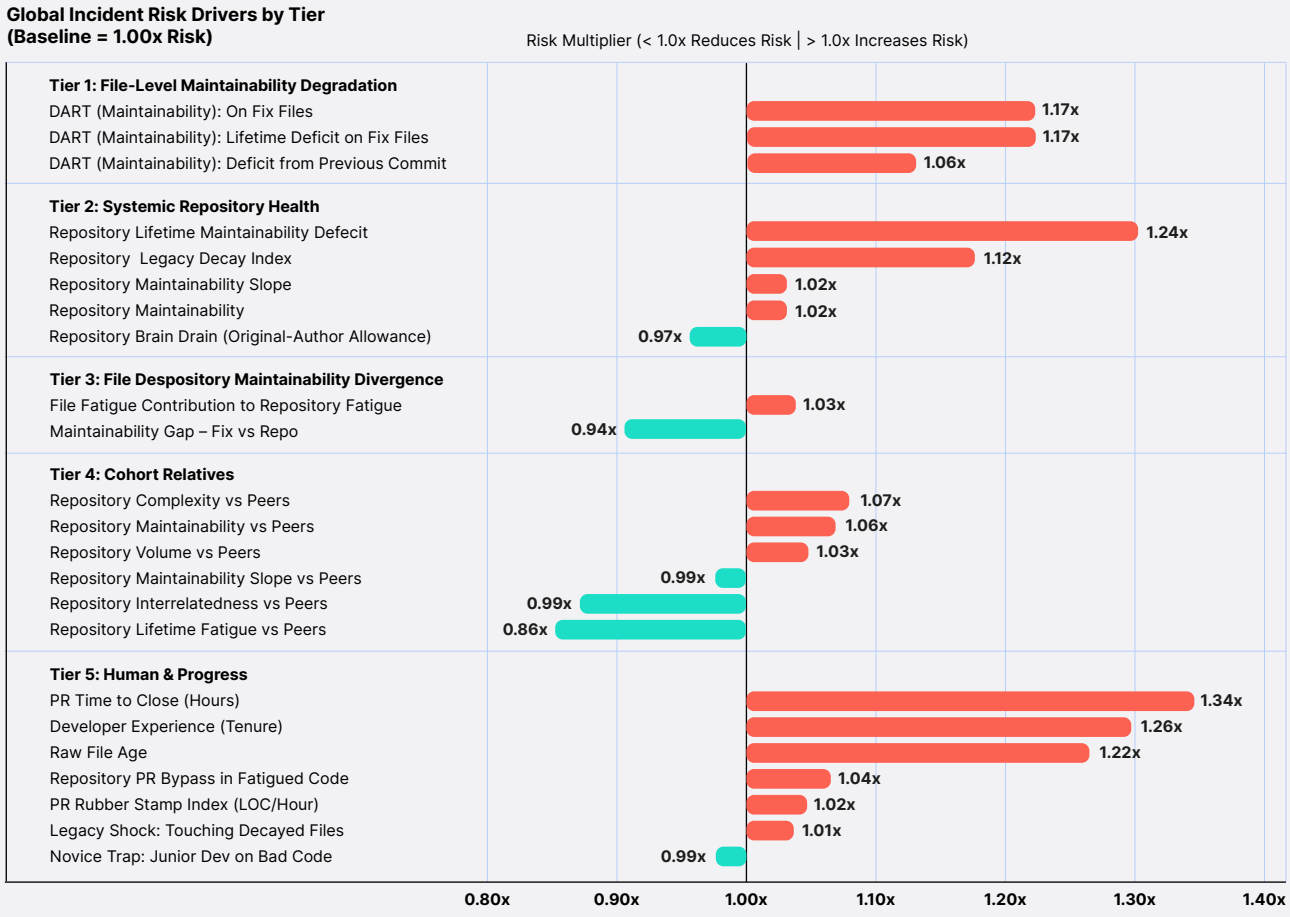
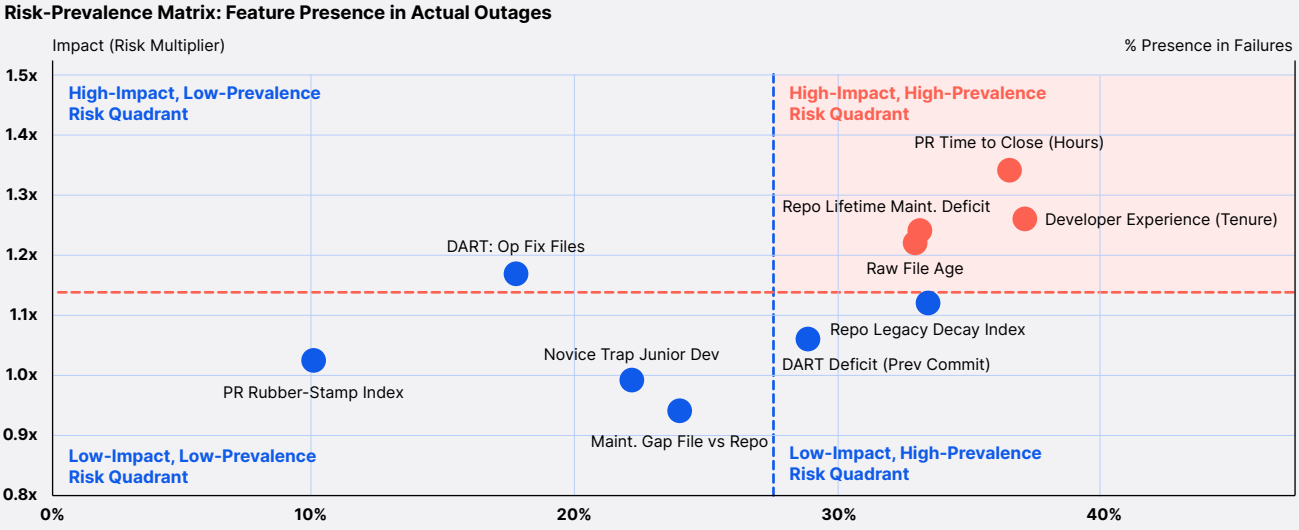


Figure 12
Risk matrix mapping feature presence in actual incidents (x-axis) against impact on incident risk (y-axis)



Discussion

Risk Drivers

The model identifies incident risk across five related feature tiers: file-level maintainability degradation, systemic repository health, file-repository maintainability divergence, cohort relatives, and human and process controls. Each tier captures a different layer of the software delivery environment, from local code maintainability to repository condition, comparative position, and review process dynamics. The following subsections summarise each tier using a consistent structure: feature, risk multiplier, what it measures, and result and interpretation.

Tier 1: File-Level Maintainability Degradation

Tier 1 captures localised maintainability degradation at the file level. These features measure whether the file being modified has recently deteriorated, has accumulated historical maintainability deficits, or is degraded during the incident fix itself. The results indicate that file-level maintainability loss is associated with elevated incident risk.

Feature	Risk Multiplier	What it measures	Result Interpretation
DART on the Fix Files *Maintainability degradation during fixes*	1.17x	This metric determines whether the incident fix reduced maintainability in the files it modified.	Fixes that reduce maintainability are associated with elevated incident risk. Remediation that restores service while degrading code maintainability may leave the affected file more fragile for subsequent changes.
DART Lifetime Deficit *Cumulative File-Level Maintainability Deficit*	1.17x	This metric captures the cumulative history of maintainability degradation within a file.	Files with repeated maintainability deficits show elevated incident risk, indicating that historical degradation can create persistent local complexity and increase susceptibility to regression.
DART Deficit from Previous Commit *Recent Maintainability Decline*	1.06x	The immediate decline in file maintainability relative to the previous commit.	Recent maintainability decline is associated with measurable downstream risk, indicating that even incremental degradation can affect future change stability.

Taken together, Tier 1 indicates that maintainability degradation at the file level is an immediate operational risk signal. Both cumulative degradation and recent quality decline are associated with higher incident probability. This suggests that small maintainability losses should not be treated as isolated code-quality issues. When they recur across commits, they can reduce the reliability margin of individual files and increase the likelihood that future changes will be more difficult to understand, validate, and stabilise.

Tier 2: Systemic Repository Health

Tier 2 characterises the repository-level environment in which individual code changes occur. Whereas Tier 1 focuses on local file condition, Tier 2 captures the broader structural state of the codebase. These metrics are central to the maintainability analysis because they measure accumulated repository-level degradation rather than individual file-level effects.

© BlueOptima Limited 2005–2026. All Rights Reserved

Feature	Risk Multiplier	What it measures	Result Interpretation
Repository Lifetime Maintainability Deficit *Cumulative Repository-Level Maintainability Deficit*	1.24x	The accumulated maintainability deficit across the repository over its lifetime.	This is one of the strongest technical risk signals in the model. Repositories with persistent maintainability debt appear to operate from a higher baseline of structural fragility.
Repository Legacy Decay Index *Repository Age and Modernisation Deficit*	1.12x	The extent to which an older repository has aged without modernisation or refactoring.	Older repositories that have not been modernised or refactored show higher observed incident risk, likely reflecting divergence from current tooling, dependencies, and engineering practices.
Repository Brain Drain *Original-Author Absence*	0.97x	The degree to which original repository authors are no longer present or active.	The effect is slightly below the neutral risk baseline. This may reflect greater caution when current maintainers work in code whose original authors are no longer available.

The strongest maintainability-related effect appears at the repository level. Repository Maintainability Lifetime Deficit carries a 1.24x multiplier, indicating that incident risk is shaped not only by the immediate file being changed, but also by the accumulated condition of the wider repository. A local maintainability decline can create near-term risk, but repeated local declines can become embedded as a persistent repository-level deficit. Once this occurs, future changes are made in a more brittle environment, where hidden coupling, reduced clarity, and weaker modular boundaries can increase the probability that a local modification produces broader operational effects.

Tier 3: File-Repository Maintainability Divergence

Tier 3 captures divergence between the maintainability of the modified file and the maintainability profile of its parent repository. These features identify whether risk is concentrated locally within a file or distributed more broadly across the repository environment.

Feature	Risk Multiplier	What it measures	Result Interpretation
Maintainability Gap, File versus Repository *File-Repository Maintainability Gap*	0.94x	The maintainability difference between the modified file and its parent repository.	This measure is associated with lower observed risk. Where the modified file is more maintainable than its repository context, developers may preserve its quality more carefully, or reviewers may apply stronger quality expectations.
File Fatigue Contribution *File-Level Maintainability Deficit Contribution*	1.03x	The extent to which a single file contributes disproportionately to repository maintainability deficit.	Localised file-level fatigue contributes modestly to incident risk. A single degraded file can create a concentrated risk point even inside a repository that is otherwise comparatively stable.

Tier 3 shows that risk is influenced by the relationship between the local change area and the surrounding repository. A comparatively maintainable file within a degraded repository may receive more careful treatment, which is consistent with the observed 0.94x multiplier. Conversely, when one file contributes disproportionately to the repository maintainability deficit, it creates a localised concentration of risk. This indicates that maintainability risk is not always evenly distributed across a repository. Some files may act as concentrated sources of fragility even when the broader repository does not appear uniformly degraded.

Tier 4: Cohort Relatives (The Peer Comparison)

Tier 4 evaluates repositories relative to comparable systems in the wider enterprise portfolio. These features provide context on whether a repository is unusually complex, unusually fatigued, or otherwise distinct from its peers.

Feature	Risk Multiplier	What it measures	Result Interpretation
Repository Complexity versus Peers	1.07x	The repository's complexity relative to comparable repositories.	Repositories that are more complex than comparable systems show higher observed incident risk. This may reflect broader responsibility concentration, architectural overreach, or greater internal dependency density.
Repository Lifetime Fatigue versus Peers	0.86x	The repository's lifetime fatigue relative to comparable repositories.	This is the strongest modeled risk-reducing feature. The result should not be interpreted as evidence that fatigue is inherently protective. It may reflect lower change frequency in visibly fatigued repositories, stronger controls around known-risk systems, or overlap with other maintainability variables that absorb part of the risk signal.

The peer-relative results should be read alongside the earlier maintainability findings. In absolute terms, accumulated repository maintainability deficit is associated with higher incident risk, consistent with the broader conclusion that long-lived maintainability debt increases operational fragility. However, when lifetime fatigue is expressed relative to comparable peer repositories, the modeled association becomes risk-reducing at 0.86x. This does not mean that maintainability deficit is protective. Rather, it suggests that peer-relative fatigue is capturing a different contextual signal once repository scale, complexity, and related maintainability variables are accounted for. Possible explanations include lower change volume in visibly degraded repositories, more conservative controls around known maintainability-debt hotspots, or overlap with other maintainability features that absorb part of the absolute risk signal.

Tier 5: Human & Process Control (Behavioral Metadata)

Tier 5 captures the execution layer of software delivery, including review duration, developer context, and file history. This tier contains the highest observed risk multipliers in the model, indicating that incident risk is shaped not only by code state, but also by the process through which changes are reviewed and merged.

© BlueOptima Limited 2005–2026. All Rights Reserved

Feature	Risk Multiplier	What it measures	Result Interpretation
PR Time to Close *Pull Request Closure Duration*	1.34x	The time required to close the pull request associated with the change.	This is the strongest overall risk driver in the model. Prolonged review cycles are associated with elevated incident risk and may signal review difficulty, implementation ambiguity, or architectural uncertainty.
Developer Experience or Tenure *Senior Developer Exposure*	1.26x	The experience or tenure associated with the developer making the change.	Developer tenure is associated with elevated incident risk. This likely reflects exposure bias, where senior engineers are assigned to older, more complex, or more fragile systems. This is best interpreted as a concentration-of-risk indicator, not as evidence of individual developer risk.
Raw File Age *File Age and Technical Context Drift*	1.22x	The age of the file being modified.	Older files are associated with higher incident risk. File age may reflect technical context drift, including outdated dependencies, obsolete assumptions, and reduced institutional knowledge.

Pull Request Closure Duration is the strongest global risk driver, with a 1.34x multiplier. In this context, the metric captures the review duration of recent code changes delivered to the same repository before the incident fix, rather than the duration of the incident-remediation PR itself. This suggests that review duration is not merely an administrative workflow measure. It is a process-level risk signal. A pull request that remains open for an extended period may indicate that reviewers are finding the change difficult to validate, difficult to reason about, or potentially unsafe to merge.

Senior Developer Exposure carries a 1.26x risk multiplier. This should not be interpreted as evidence that senior developers are inherently more likely to introduce incidents. A more cautious interpretation is that incident-linked revisions are more common in contexts where experienced developers are involved, which may reflect several mechanisms: concentration of senior engineers on complex or fragile systems, assignment of larger or higher-risk changes to experienced staff, or uneven ownership of critical code areas. The model identifies the association, but does not distinguish among these explanations.

Similarly, File Age and Technical Context Drift, at 1.22x, is consistent with technical context drift. Older files may rely on outdated libraries, patterns, or assumptions, and their original design intent may no longer be well understood. Together, these results show that incident risk increases when aged code, accumulated maintainability deficits, and extended review cycles coincide.

The Risk-Prevalence Matrix

The risk-prevalence matrix complements the risk-driver analysis by comparing each factor's estimated impact with its observed prevalence in historical incident-linked revisions. This separates frequently occurring systemic factors from less frequent but higher-impact contributors.

1. High-Impact, High-Prevalence Risk Quadrant

The high-impact, high-prevalence quadrant contains the most operationally significant combination of risk factors. These features are both strongly associated with incident risk and frequently present in historical incident-linked revisions.

This quadrant includes Pull Request Closure Duration at 1.34x, Senior Developer Exposure at 1.26x, Cumulative Repository-Level Maintainability Deficit at 1.24x, and File Age at 1.22x. The pattern indicates that the most significant risk profile combines accumulated repository maintainability

 © BlueOptima Limited 2005–2026. All Rights Reserved

debt, older files, senior engineer exposure, and extended review cycles.

The placement of Pull Request Closure Duration in this quadrant is particularly important. It is both the strongest observed risk multiplier and sufficiently prevalent to function as a systemic process signal. In this context, slow review is not simply elapsed time. It is associated with higher cognitive load, greater validation difficulty, and increased uncertainty around the safety of the change.

2. High-Impact, Low-Prevalence Risk Quadrant

The high-impact, low-prevalence quadrant contains one top-10 metric: **Maintainability Degradation During Fixes**, with a 1.17x risk multiplier. Although this factor appears less frequently than the high-prevalence drivers, its effect size indicates that it is not a marginal quality issue. When present, it is associated with a material increase in incident risk.

This finding is important because it describes a likely mechanism through which repository-level maintainability deficits accumulate over time. A single low-maintainability change made during incident remediation may appear limited when reviewed in isolation, particularly under time pressure. However, once merged, it becomes part of the repository's permanent maintainability state. Repeated instances of this pattern can accumulate into **Repository-Lifetime Maintainability Deficit**, which carries a 1.24x multiplier and is one of the strongest technical risk signals in the model.

The low prevalence of this metric may reduce its perceived urgency. Teams may treat such changes as exceptional compromises rather than as contributors to long-term structural risk. The matrix suggests the opposite. Even if these events are less frequent, they represent high-impact entry points for maintainability degradation. Over time, accepting low-maintainability fixes and changes increases the probability that the repository will develop a higher baseline maintainability deficit, making future changes more difficult to understand, review, and stabilise.

3. Low-Impact, High-Prevalence Risk Quadrant

The low-impact, high-prevalence quadrant contains factors with comparatively smaller individual multipliers but broader occurrence. Examples include Repo Legacy Decay Index at 1.12x and Recent Maintainability Decline at 1.06x.

These features represent recurring sources of engineering friction. A single instance may not be sufficient to produce an incident, but repeated occurrence across a large portfolio can raise the baseline level of system instability. This is especially relevant for maintainability measures, where small reductions can compound over time into larger repository-level deficits.

The significance of this quadrant is cumulative. Lower-magnitude risk factors can become material when they are frequent, persistent, and distributed across many repositories.

4. Low-Impact, Low-Prevalence Risk Quadrant

The low-impact, low-prevalence quadrant contains factors that are neither frequent nor strongly risk-increasing in isolation. The junior developer on the bad code metric, at 0.99x, appears in this quadrant. This placement does not support the common assumption that junior developers introduced changes may be a primary source for incident risk. The observed multiplier suggests that supervision, review practices, or task allocation may be limiting this risk.

The PR Rubber-Stamp Index, at 1.02x, also appears in this quadrant. Its direct measured association with incident occurrence is limited relative to the dominant drivers. However, this does not imply that low-scrutiny review is irrelevant. Its significance may be indirect. If weak review allows maintainability degradation to pass through the pipeline, it can enable high-impact, low-prevalence risks to accumulate into repository-level maintainability deficits.

Synthesis

The model indicates that incidents are associated with the interaction of the technical environment and delivery behaviour. The strongest observed pattern occurs when changes involve aged files, repositories with accumulated maintainability deficits, and prolonged pull request closure cycles. In these conditions, incident risk is not attributable to a single isolated feature. It reflects the combined effect of repository health, file history, developer exposure, and review complexity.

The central finding is that maintainability functions as an operational risk signal. At the file level, maintainability degradation increases the risk associated with subsequent changes. At the repository level, accumulated maintainability deficit raises the baseline fragility of the system. Delivery

© BlueOptima Limited 2005–2026. All Rights Reserved

factors, especially Pull Request Closure Duration, then appear to reveal or amplify this risk during review and deployment.

This interpretation is consistent with the broader structure of the results. Maintainability-related features appear across multiple tiers, including file-level degradation, repository-level deficit, file-repository divergence, and peer-relative comparison. The model therefore does not treat maintainability as a narrow code-quality metric. It positions maintainability as a cross-layer condition that shapes how difficult software is to understand, change, review, and stabilise under operational pressure.

Conclusion

This research shows that software incidents are not purely random operational events. They are statistically associated with the interaction between technical environment and delivery behaviour. Based on an analysis of more than 666,000 code revisions, the study finds that declining maintainability, measured through BlueOptima's proprietary metrics, functions as an early warning signal for software incidents.

The central finding is that maintainability should be treated as an operational risk control, not only as a code-quality attribute. Local maintainability degradation increases the risk associated with subsequent changes, while accumulated repository-level maintainability deficit raises the baseline fragility of the system. This risk is further amplified when changes involve aged files, prolonged pull request closure cycles, and code areas that are difficult to understand, review, and stabilise.

The model's performance reinforces the practical relevance of this finding. With a recall of 0.74, the model identifies approximately three out of every four incident-inducing revisions before deployment. This level of detection indicates that maintainability and related delivery signals can support proactive risk screening within the software delivery pipeline. The objective is not to increase manual review indiscriminately, but to apply review effort more precisely at the point where risky changes enter the codebase.

The results also indicate that recent productivity gains associated with GenAI may be accompanied by measurable pressure on codebase quality. If machine-assisted code generation accelerates output while reducing maintainability, organizations may face a growing stock of code that is harder to interpret, validate, and safely modify over time. This risk is particularly important where human teams lose contextual knowledge of machine-authored logic or where AI-generated contributions accumulate without sufficient maintainability control.

The implication is clear: maintainability must be embedded into the CI/CD pipeline as a measurable and enforceable risk signal. Stability is not achieved only through faster incident response. It is preserved by preventing low-maintainability changes from accumulating into repository-level structural fragility.

© BlueOptima Limited 2005–2026. All Rights Reserved

Recommendations

To mitigate the systemic risks identified in this report, engineering leadership should adopt targeted controls that treat maintainability as an operational risk signal. The following recommendations preserve the structure of insight and action, linking each recommendation to the empirical findings.

1

Implement Automated Maintainability Gates in CI/CD

Insight: Repository Lifetime Maintainability Deficit carries a 1.24x risk multiplier and is one of the strongest technical risk signals in the model. This deficit is likely formed through repeated lower-level maintainability declines, including Maintainability Degradation During Fixes, which carries a 1.17x multiplier.

Action: Integrate Developer ART (Analysis of Relative Thresholds) maintainability gates directly into CI/CD pipelines. These gates should identify and block changes that introduce material maintainability degradation, especially in high-impact, low-prevalence risk conditions. The objective is to prevent local quality declines from accumulating into persistent repository-level structural fragility.

2

Expand Shared Ownership of High-Risk Code Areas

Insight: Senior developer exposure is associated with a 1.26x risk multiplier. This should not be interpreted as evidence that experienced developers are less reliable. The result is better understood as a concentration-of-risk signal: incident-linked revisions are more common in contexts where senior engineers are involved, which may reflect several overlapping mechanisms, including assignment to older and more complex systems, larger or higher-risk changes, or reliance on a small number of engineers with critical institutional knowledge. The model identifies this association, but does not distinguish among these explanations.

Action: Establish shared ownership for high-risk files and repositories. Use maintainability metrics to identify aged files, repositories with high cumulative maintainability deficit, and code areas with prolonged pull request closure cycles. Prioritise these areas for refactoring, documentation, pairing, and knowledge transfer so that critical operational knowledge and responsibility are distributed across a broader engineering group.

3

Require Touch-and-Clean Controls for Aged Files

Insight: File Age and Technical Context Drift is associated with a 1.22x risk multiplier. Older files may carry higher risk because they can diverge from current tooling, dependency expectations, security standards, and architectural practices.

Action: Adopt a touch-and-clean policy for aged files. When a file above a defined age threshold, for example older than two years, is modified, the change should include a proportionate maintainability review. Where appropriate, this should include refactoring, dependency updates, documentation improvement, test reinforcement, or alignment with current architectural and security standards.

© BlueOptima Limited 2005–2026. All Rights Reserved

4

Audit Low-Scrutiny Review Approval

Insight: The PR Rubber-Stamp Index has a lower direct multiplier, at 1.02x, but may still represent an indirect process risk. Low-scrutiny approvals can allow maintainability degradation to enter the codebase without adequate challenge, contributing over time to cumulative repository-level maintainability deficit.

Action: Use pull request reporting data (using BlueOptima TLD) to identify unusually rapid approvals in high-risk code areas. Pull requests that close very quickly while touching aged files, low-maintainability files, or repositories with high historical risk multipliers should trigger secondary review. This control should focus on review adequacy rather than review duration alone.

5

Monitor the GenAI Productivity-Maintainability Trade-Off

Insight: GenAI-enabled development may increase code production velocity while placing measurable pressure on maintainability. A 1.22 percentage point decline in overall code quality indicates that productivity gains should be evaluated alongside their effect on long-term codebase health.

Action: Do not assess GenAI success solely through delivery speed, output volume, or developer throughput. Incorporate maintainability monitoring into GenAI adoption metrics, including file-level degradation, repository-level maintainability deficit, PR closure duration, and review quality indicators. This ensures that productivity gains do not create a growing stock of code that is harder to understand, review, and safely modify over time.

© BlueOptima Limited 2005–2026. All Rights Reserved

References

Beyer, B., Jones, C., Petoff, J., & Murphy, N. R. (Eds.). (2016). *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly. https://research.google/pubs/site-reliability-engineering-how-google-runs-production-systems/

BlueOptima. (2022, December 04). *The Great Resignation: Exploring the Global Impact and Changing Behaviours*. BlueOptima. https://www.blueoptima.com/resource/the-great-resignation-exploring-the-global-impact-and-changing-behaviours

BlueOptima. (2023, October 6). *Remote Work: Impact on Software Developer Productivity*. BlueOptima. https://www.blueoptima.com/resource/remote-work-impact-on-software-developer-productivity

Financial Conduct Authority. (2014, 11 11). *Final Notice: The Royal Bank of Scotland plc*. Financial Conduct Authority. https://www.fca.org.uk/publication/final-notices/final-notice-rbs.pdf

Financial Conduct Authority. (2022). *Final Notice: TSB Bank plc*. Financial Conduct Authority. https://www.fca.org.uk/publication/final-notices/tsb-bank-plc-2022.pdf

He, H., & Garcia, E. A. (2009, September). Learning from Imbalanced Data. *IEEE Transactions on Knowledge and Data Engineering, 21*(9), 1263–1284. https://doi.org/10.1109/TKDE.2008.239

Krasner, H. (2022). *Cost of Poor Software Quality in the U.S.: A 2022 Report*. Consortium for Information & Software Quality. https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2022-report/

Lessmann, S., Baesens, B., Mues, C., & Pietsch, S. (2008, July/August). Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 34(5), 485–495. https://doi.org/10.1109/TSE.2008.35

Menzies, T., Greenwald, J., & Frank, A. (2007, January 1). Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Transactions on Software Engineering, 33*(1), 2–13. https://dl.acm.org/doi/10.5555/1248721.1248734

Recupito, G., Pecorelli, F., Catolino, G., Lenarduzzi, V., Taibi, D., Nucci, D. D., & Palomba, F. (2024, October). Technical debt in AI-enabled systems: On the prevalence, severity, impact, and management strategies for code and architecture. *Journal of Systems and Software, 216*, 21. https://doi.org/10.1016/j.jss.2024.112151

Securities and Exchange Commission. (2013, October 16). *Knight Capital Americas LLC*. Headlands Technologies LLC. Retrieved April 23, 2026, from https://www.sec.gov/files/litigation/admin/2013/34-70694.pdf

Shmueli, G. (2010, August). To Explain or to Predict? *Statistical Science, 25*(3), 289–310. https://doi.org/10.1214/10-STS330

Besker, T., Martini, A., & Bosch, J. (2019). Software developer productivity loss due to technical debt—A replication and extension study examining developers' development work. *Journal of Systems and Software, 156*, 41–61. https://doi.org/10.1016/j.jss.2019.06.004

Beyer, B., Jones, C., Petoff, J., & Murphy, N. R. (2016). *Site reliability engineering book Google index*. Google SRE. https://sre.google/sre-book/table-of-contents/

BlueOptima. (2025, 10 10). *Stability Plague then AI*. BlueOptima. https://www.blueoptima.com/resource/stability-plague-then-ai

BlueOptima. (2026, 02 24). *GenAI License-Based Usage Impact on Software Development Productivity*. BlueOptima. https://www.blueoptima.com/resource/genai-license-based-usage-impact-on-software-development-productivity

Notices and Disclosures

About the Research Team

This research was conducted by BlueOptima's Data Science team, which specializes in providing DevOps and DevSecOps consultancy services to organizations developing enterprise-ready software at scale. The team combines extensive experience in analyzing software development practices with a deep understanding of performance metrics across diverse technical environments. Leveraging cutting-edge analytics, they deliver actionable insights to optimize software maintainability, performance, and security. For inquiries about consultancy services or to discuss how these findings might benefit your organization, please contact Consulting@BlueOptima.com.

Research Collaboration Opportunities

BlueOptima invites postdoctoral researchers and tenured academics to explore the industry-leading BlueOptima Global Benchmark dataset. This unique dataset offers unparalleled visibility into the software development operations of large-scale enterprise teams, delivering granular metrics and insights into real-world development practices. With industrial-grade data protection and anonymization protocols, the dataset ensures robust privacy while enabling groundbreaking research into software development at scale. Qualified researchers interested in collaboration opportunities are encouraged to contact Research@BlueOptima.com to discuss research proposals and data access protocols.

BlueOptima's Software Products

BlueOptima provides a suite of Enterprise SaaS solutions that empower organizations to deliver software development at scale with greater efficiency and reliability. The company's product portfolio includes Developer Analytics, which delivers actionable insights into developer productivity and code quality; Code Insights, a powerful tool for assessing and mitigating risks in software codebases; and Predictive Assessment, which streamlines and improves the identification, selection, and recruitment of the talent that drives an enterprise's software estate into the future. These solutions optimize performance, improve collaboration, and enhance decision-making for enterprise software teams. For more information, please contact Sales@BlueOptima.com.

Appendix A: Mathematical Foundations of the Five–Tier Risk Framework

To make the High-Impact, High-Prevalence Risk Quadrant reproducible, its thresholds are anchored in two observable dimensions: feature prevalence and modeled risk association.

1. Feature Volatility: Standard Deviation

We define the "Physical Unit" of a risk factor by its Standard Deviation within the global dataset of 666,000 revisions. This represents the natural variance of engineering behavior. For a feature X:

$$\sigma_x = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

2. Model Sensitivity: Mean Absolute SHAP

The impact of a feature is determined by the Mean Absolute SHAP value. This represents the average magnitude of change in the model's log-odds output attributable to that feature:

$$\overline{|\phi_x|} = \frac{1}{N}\sum_{i=1}^{N}|\phi_{i,x}|$$

Where $\phi_{i,x}$ is the SHAP value for feature X in revision i. Because the XGBoost model uses a logistic link function, ϕ is measured in **log-odds**.

3. The Global Risk Multiplier

The Global Multiplier represents the relative increase in incident probability associated with 1 standard deviation from the mean. It is the exponential transformation of the Mean Absolute SHAP value:

$$M_x = e^{\overline{|\phi_x|}}$$

Interpretation: An $M_x = 1.24$ implies that a typical (one standard deviation) increase in feature results in a **24% increase** in the probability of a system-wise outage.

 © BlueOptima Limited 2005–2026. All Rights Reserved

4. The 1% Risk Unit

To flag riskier commits, we solve for the exact physical change (U) required to increase risk by exactly 1%. This is derived from the **Log-Odds Slope** (S), which is the ratio of risk gravity to physical volatility:

The Slope (S):

$$S_x = \frac{\overline{|\phi_x|}}{\sigma_x}$$

The 1% Unit ($U_{1\%}$):

$$U_{1\%} = \frac{1\mathrm{n}(1.01)}{S_x}$$

5. Dynamic Risk Scoring

For any new Pull Request or commit, the **Total Risk Multiplier** (M_{total}) can be calculated by aggregating the deltas (Δ) of all 23 contextual features:

$$M_{total} = \exp\left(\sum_{j=1}^{27} \Delta x_j \cdot S_j\right)$$

Where Δx_j is the observed change in feature j (e.g., a drop of 5 poins in maintainability).
If $M_{total} > 1.25$, the revision is automatically flagged as a **high risk event**.

© BlueOptima Limited 2005–2026. All Rights Reserved

Appendix B: Model Design & Evaluation Methodologies

To ensure the model acts as a valid, objective early warning system, we evaluate it using four metrics established as the standard in Software Defect Prediction (SDP) and anomaly detection:

- **Recall (The "Catch Rate"):** Measures the percentage of actual historical incidents model successfully flagged. The model predicts a threat is absent, but reality says it is present. Research by Menzies et al., (2007), establishes that in software defect prediction, maximizing Recall is the primary economic driver, as the cost of a false negative (a catastrophic defect escaping to production) exponentially outweighs the cost of reviewing a false positive.

  Recall measures your defense against **False Negatives**. (High recall = very few missed threats).

- **Precision (The "False Alarm" Filter):** Measures the signal-to-noise ratio when a model flags a code change. The model predicts a threat is present, but reality says it is absent.

  Precision measures your defense against **False Positives**. (High precision = very few false alarms).

- **The F1-Score (The Operational Trade-Off):** The harmonic mean of Precision and Recall. In domains predicting extreme minority classes, such as credit card fraud, medical anomalies, or software incidents, performance metrics are statistically bounded by the extreme rarity of the event (the base rate). Scientific consensus in imbalanced learning (e.g., He & Garcia, 2009) establishes that standard 0-to-100 grading scales do not apply here. A valid model is measured by how massively it multiplies the base rate to find the needle in the haystack, proving it has found a profitable operational balance between catching threats and maintaining CI/CD pipeline velocity.

- **AUC-ROC (The Automated Triage):** Evaluates the model's fundamental ability to sort and rank risk independently of a fixed threshold. Lessmann et al., 2008, established AUC-ROC as the premier benchmark metric for software defect prediction. It validates the model's utility as an automated triage system, proving it can reliably push the most dangerous changes to the top of the queue for human architectural review. An AUC-ROC score, ranging from 0.5 (random guessing) to 1.0 (perfect foresight), represents the exact probability that the algorithm will correctly rank a hidden threat higher than a safe, normal event.

Methodological Design Choices:

This model is designed as an explanatory risk diagnostic. It identifies historical signatures associated with software incidents, rather than forecasting outcomes for unseen commits in an out-of-sample setting.

Incidents are low-frequency, high-impact events. For this reason, the risk multipliers are calibrated against the full historical incident record. The purpose of this design choice is to preserve the complete observed risk range, including the most severe historical cases, rather than excluding rare but operationally significant events from the estimation process.

This approach is appropriate because the objective is explanatory parameter extraction rather than estimation of out-of-sample generalisation error. In rare-event settings, withholding incident observations for a test set can destabilise the estimated feature effects and reduce the reliability of the maximum observed risk ceiling. The full dataset therefore provides the most stable basis for estimating SHAP coefficients and calibrating risk multipliers.

Future model iterations intended for real-time commit tagging may incorporate advanced K-fold cross-validation, bootstrapping, or other validation procedures. However, the foundational risk multipliers applied to those commits should remain anchored to the full global dataset to preserve calibration against the complete historical incident distribution.

Class Balance and Negative-Class Sampling

Software incidents represent a rare-event classification problem: the number of ordinary developer commits substantially exceeds the number of commits linked to incident fixes. If the full negative class is used without adjustment, model estimation and evaluation can become dominated by the majority class, obscuring the distinguishing structure of incident-linked revisions.

To address this, the modeling dataset was constructed using negative-class down-sampling.

© BlueOptima Limited 2005–2026. All Rights Reserved

All incident-linked commits were retained as positive observations. Non-incident-linked commits were randomly sampled, using a fixed random seed for reproducibility, up to a maximum of ten non-incident observations for each incident-linked commit. This produces a maximum 1:10 positive-to-negative ratio, or an approximate 9.1% positive-class prevalence in the modeling sample where sufficient non-incident observations are available.

The 1:10 design was selected to preserve the rare-event character of software incidents while providing sufficient contrast between incident-linked and ordinary development activity. A highly balanced design, such as 1:1, would overstate the prevalence of incident-linked commits and move the modeling sample too far away from the operational rarity of incidents. Conversely, using the full negative class without sampling would allow the majority class to dominate the learning problem, reducing sensitivity to the comparatively rare incident-linked signal. The selected design therefore maintains meaningful imbalance while ensuring that each incident-linked revision is evaluated against a broad reference set of non-incident activity.

Reported recall, precision, F1-score, and AUC-ROC should therefore be interpreted as historical classification statistics under the sampled class balance.

Cohort-Relative Benchmark Limitation

The cohort-relative benchmarks are based on active repositories. This improves operational relevance, because engineering leaders primarily need to compare systems that remain under development and maintenance. However, it also means the peer baseline may be affected by survivorship bias: repositories that were retired, archived, or rewritten because of severe maintainability or stability problems may be absent from the comparison set. As a result, cohort-relative scores should be interpreted as comparisons against surviving active systems, not against the full historical population of enterprise repositories.

© BlueOptima Limited 2005–2026. All Rights Reserved

Appendix C: Coding Effort

What is Coding Effort?

Coding Effort is a measure of the average intellectual effort being delivered by a population of programmers delivering change into the source code of an application. This "effort" is a measure of work that reflects the relative difficulty of working on different source and configuration files, while taking into account the aspects of the programmer's work that are not under their control (i.e. the context within which the change is delivered). By evaluating every change made by all developers as they contribute their code through the software development process, Coding Effort accurately reflects time spent coding.

How does it work?

BlueOptima deltas every revision made by every developer to every source code file, by integrating into source code repositories (e.g. Subversion, Git, ClearCase etc.) and task trackers (e.g. HP ALM, Atlassian Jira, Bugzilla, etc.). Every change is evaluated in terms of an array of 36 static source code metrics covering more than 70 source code types. The change delivered by developers is then benchmarked based on its Volume, Complexity, and Interrelatedness. Big data analysis techniques then enable every subsequent change to the codebase to be quantified as a Coding Effort score. Proprietary algorithms are used to calculate Coding Effort in terms of BlueOptima model units. Those model units are based on benchmark calculations across hundreds of millions of static metrics gathered over tens of millions of developer interactions with source code.

Actual Coding Effort (ACE) & Billable Coding Effort (BCE) are measures of developers' total output and productivity.

Actual Coding Effort (ACE)

The Actual Coding Effort metric measures the relative intellectual effort delivered into source code change by a developer or group of developers. BlueOptima calculates Actual Coding Effort by evaluating every change developers make to source code using 36 static metrics and benchmarking every commit based on Volume, Complexity, and Interrelatedness.

These static metrics are gathered via integration with source code repositories such as Bitbucket, AzureDevOps, GitHub, and GitLab.

Once all the commit data is gathered across all repositories, BlueOptima filters out the non-meaningful changes that could not be fairly attributed to an individual's effort, such as:

- Copy-paste of code
- Autogenerated code from the IDE or other tooling
- Very small or trivial changes
- Merging and branching activity

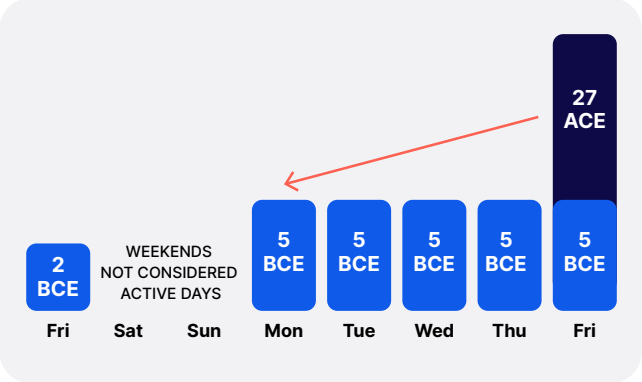
Once filtered, the ACE algorithm only considers activity attributable to an individual developer's intellectual effort, including new code but refactoring existing code and code removal, to ensure developers improving and maintaining codebases are fairly recognized.

When the historical analysis is completed, the benchmarking process computes a statistical understanding of the average code contribution per developer per day across these 36 dimensions. This benchmarking process compares every commit to the history of all other changes to quantify the relative intellectual effort per commit.

Billable Coding Effort (BCE)

Billable Coding Effort (BCE) addresses the inconsistency in how developers commit their code, as some commit frequently while others hold changes for extended periods. BCE accurately reflects work timing by prorating Actual Coding Effort (ACE) across a developer's active workdays, regardless of when the code was actually committed. This proration results in a smoother distribution of effort, enabling more reliable analysis of performance trends over time.

BCE is capped at a maximum of 5 units per day, aligned with findings from academic and enterprise time and motion studies.



 © BlueOptima Limited 2005–2026. All Rights Reserved

The proration model distributes effort over a maximum of 10 working days, excluding weekends and public holidays unless the commit occurred on a holiday. Public holidays are determined based on the city and country of each developer, as provided by the customer organization.

Worked Example

1. If a commit's Actual Coding Effort (ACE) exceeds 5 hours, it will be prorated backward to the previous 10 working days.
2. If a commit was made within the previous 10 working days, the proration will continue until ACE is exhausted (or) the 10 days criteria is met.
3. Example: For a commit on 10/10/2024 with an ACE of 8.8021 hours, 5 hours is assigned as BCE on the same day. The remaining 3.8021 hours is prorated backward over up to 10 preceding working days, regardless of commit frequency.
4. If the same developer has another commit on the previous day i.e. 09/10/2024, worth 9.12 hours of ACE, the breakout of BCE allocation is shown below:

Date	Source Commit	Actual Coding Effort (ACE)	Billable Coding Effort (BCE)	Residual Remaining After Allocation
10/10/2024	Commit on 10/10/2024	8.8021	5.0000	3.8021
09/10/2024	Commit on 09/10/2024	9.1200	5.0000	4.1200
08/10/2024	Residual from 10/10/2024 + residual from 09/10/2024	3.8021 + 4.1200 = 7.9221	5.0000	2.9221
07/10/2024	Remaining residual from 09/10/2024	2.9221	2.9221	0.0000

5. Because BCE is capped at 5 hours per day, 09/10/2024 is already fully utilized by the 09/10 commit itself. As a result, the 3.8021 residual hours from 10/10/2024 cannot be placed on 09/10/2024 and must continue backward to the next available working day.
6. On 08/10/2024, the residual BCE from both dates are eligible for allocation. Since only 5.0000 BCE hours can be assigned to that day, the available daily capacity is filled there, and any remaining effort continues backward. The final 2.9221 hours are then assigned to 07/10/2024.
7. This example shows how the presence of another commit within the backward window does not stop proration. Instead, all excess ACE continues to be distributed across earlier working days, subject to the 5-hour daily BCE cap and the 10-working-day maximum proration window.

Appendix D: Analysis of Relative Thresholds

The Analysis of Relative Thresholds (ART) is a metric that evaluates the maintainability of source code and its ease of modification. This metric assists developers in determining the level of difficulty associated with working on unfamiliar code.

ART works by establishing what "healthy" code looks like for a given organisation and then measuring every commit against that standard. BlueOptima extracts static source-code metrics, covering dimensions like volume, complexity, and inter-relatedness, and groups them into logical collections. Thresholds are then derived relative to the organisation's own codebase, accounting for file type, application, and system context. There is no universal pass/fail rule applied identically to every organisation, as the thresholds reflect what normal, maintainable code looks like in your specific environment.

When a developer makes a source code change, BlueOptima evaluates where that commit sits relative to those thresholds. Code that stays within healthy ranges contributes to maintainable Coding Effort. Code that breaches the thresholds is flagged as aberrant. The static metrics inform where the line is drawn. In other words, ART tells you how much of the team's output stayed on the right side of it.

To establish static metric thresholds, we analyze comparable source code repositories and formulate guidelines based on the quantity, intricacy, and interdependence of the files in that specific language.

For example, 60% of Java files in an application should have between 6 and 10 methods, 70% between 4 and 13, 80% between 3 and 19, and 90% between 2 and 31.

ART provides two types of metrics:

- Dynamic ART (DART)
- File-level ART (FLART)

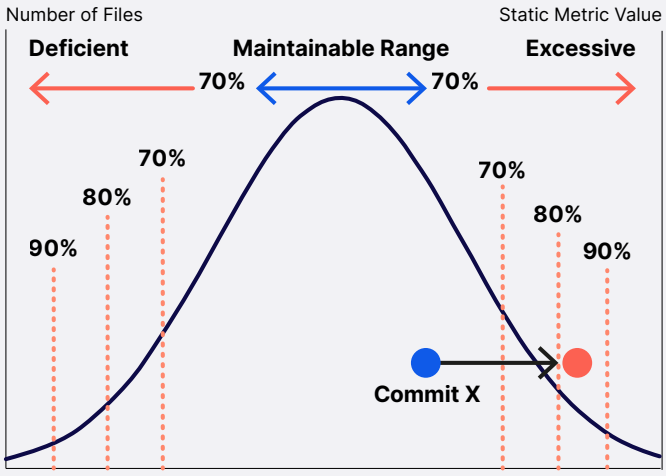
Dynamic ART (DART)

Dynamic ART is a quality metric measuring the maintainability of the changes developers make to source code. It measures how readily a developer can pick an unfamiliar source file and implement changes.

DART is displayed in BlueOptima's user interface as the percentage of Aberrant Coding Effort.

DART offers a reliable approach to identifying developers or teams producing subpar quality codes. It provides a detailed analysis of the underlying factors contributing to ART scores. Every modification made by a developer undergoes evaluation based on quality threshold rules and is then measured against the level of Coding Effort needed for that alteration. DART provides a consistent means of highlighting developers or teams that deliver codes of questionable quality and granular detail into the underlying causes of the ART scores. Each revision made by a developer is scored against the quality threshold rules and then scaled against the amount of Coding Effort required to make that change.

A score of 0% means that changes made were easy to maintain, or that the maintainability of the code was improved. A higher percentage indicates that more effort was put into making changes that are difficult to maintain.



 © BlueOptima Limited 2005–2026. All Rights Reserved

The DART scoring approach differs from others using "red or green" or "on/off" thresholds. Instead, DART offers a range of aberrancy scores based on the severity of the deviation outside the mantaintainable range, and number of static metrics that exceed the thresholds.

For example, a change breaching the 70% threshold for one metric will get a low aberrancy score, whilst one breaching 90%+ thresholds for many metrics will have a much higher score to prioritize code review and fixes to those changes.

File-Level ART (FLART)

The FLART metrics offer a quick view of file maintainability at a static point. FLART is available in the Maintainability Report and Team Lead Maintainability dashboards and is labelled Maintainability. FLART uses the same family of static source code metrics and threshold violation principles but considers the scoring of the file(s) as a whole, rather than the changes in isolation. It helps the managers distinguish the high-risk applications and areas that require improvement and prioritize technical debt tasks to reduce the risk of production issues.

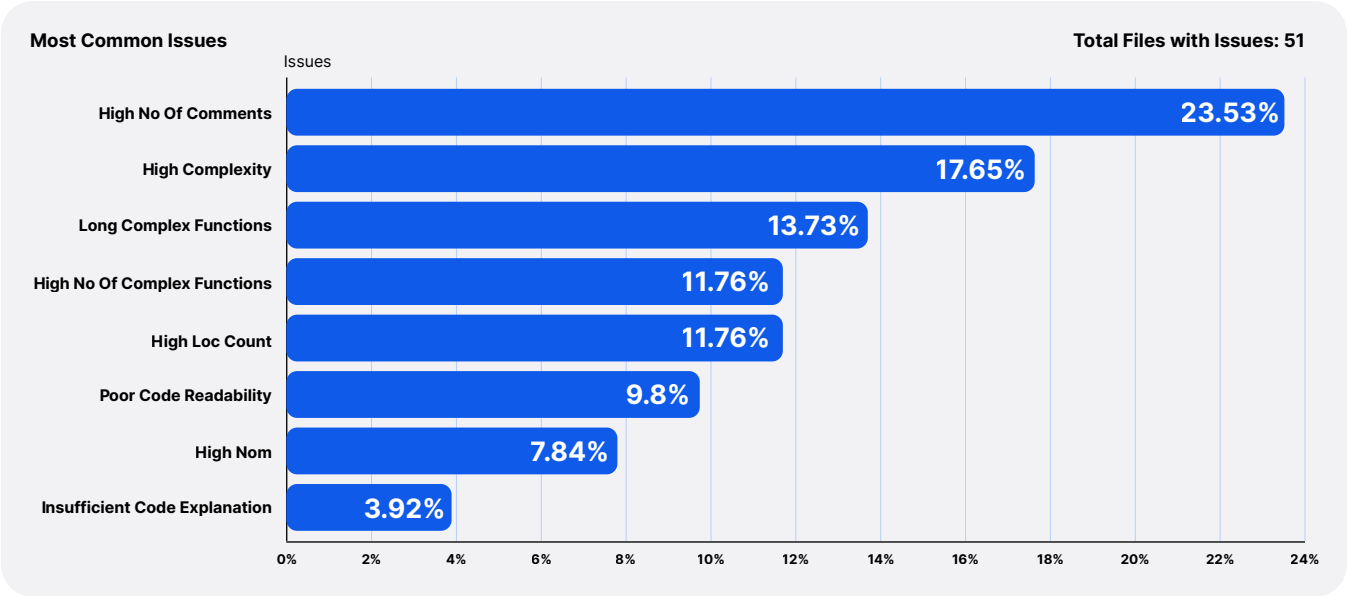
What's the Best Approach to Balance Speed and Quality?

BlueOptima provides metrics across three dimensions: productivity, quality, and cost. We recommend using the ART (Analysis of Relative Threshold) maintainability metric as a control measure, alongside BCE/Day, to ensure that quality is not sacrificed for output.

In this sense, you will be able to inform the team what the maximum level of acceptable aberrance should be while optimizing for productivity.

Use the Maintainability Report and Team Lead Dashboard to identify issues in the codebase your teams work in. This helps you understand the factors or habits affecting your maintainability, increase team awareness of good practices, and take action to fix them.

To ensure team success, productivity must increase without increasing aberrance or compromising maintainability, or vice versa. Your CSM can help you set appropriate targets.



blueoptima™

p +44 207 100 8740
e enquiries@blueoptima.com
w www.blueoptima.com