MetricSynth: Framework for Aggregating DORA and KPI Metrics Across Multi-Platform Engineering

Pallav Jain, Yuvraj Agrawal, Ashutosh Nigam, Pushpak Patil
Adobe Inc.

***Abstract*—In modern, large-scale software development, engineering leaders face the significant challenge of gaining a holistic and data-driven view of team performance and system health. Data is often siloed across numerous disparate tools, making manual report generation time-consuming and prone to inconsistencies. This paper presents the architecture and implementation of a centralized framework designed to provide near-real-time visibility into developer experience (DevEx) and Key Performance Indicator (KPI) metrics for a software ecosystem. By aggregating data from various internal tools and platforms, the system computes and visualizes metrics across key areas such as Developer Productivity, Quality, and Operational Efficiency. The architecture features a cron-based data ingestion layer, a dual-schema data storage approach, a processing engine for metric pre-computation, a proactive alerting system, and utilizes the open-source BI tool Metabase for visualization, all secured with role-based access control (RBAC). The implementation resulted in a significant reduction in manual reporting efforts, saving an estimated 20 person-hours per week, and enabled faster, data-driven bottleneck identification. Finally, we evaluate the system's scalability and discuss its trade-offs, positioning it as a valuable contribution to engineering intelligence platforms.**

***Index Terms*—Engineering Metrics, DORA Metrics, KPI Metrics, Software Development Analytics, Business Intelligence, Developer Productivity, System Architecture**

I. Introduction

The complexity of managing software engineering teams for a software product which spans multiple platforms, presents a formidable challenge. Engineering leaders and stakeholders require timely and accurate insights to make informed, data-driven decisions. However, the necessary data is typically fragmented across a multitude of specialized tools, including source control systems, project management platforms, continuous integration services, log analysis tools, and application performance monitoring solutions. This fragmentation necessitates significant manual effort to collect, aggregate, and analyze data, a process that is not only inefficient but also susceptible to error and inconsistency.

Prior to the implementation of our unified framework, the process of generating performance reports was a significant operational burden. Engineering and program managers would spend hours manually sourcing data from disparate systems and creating complex pivot tables. This manual process lacked data interconnectedness, making it nearly impossible to draw correlations between, for instance, a spike in CI/CD build failures and a subsequent slowdown in feature delivery.

To address these challenges, we have developed a centralized framework that provides a unified, near-real-time view into developer experience (DevEx) and key performance indicators (KPIs). Our work is heavily influenced by industry standards such as the DevOps Research and Assessment (DORA) metrics, which are strong indicators of high-performing teams [2], [9]. We track these alongside a curated set of internal KPIs tailored to our organizational goals across five key areas: Developer Productivity, Quality & Reliability, Automation Tooling, Engagement, and Operational Efficiency.

The main contributions of this paper are:

- The design of a scalable, multi-layered architecture for aggregating engineering data, featuring a dual-schema storage strategy and a pre-computation layer for performance.
- A curated framework of DORA-aligned and internal KPI metrics tailored for a large-scale, multi-platform software product.
- An empirical evaluation of the framework's impact, including quantitative time savings and qualitative case studies on improving development workflows.
- A discussion of the system's unique aspects, including its extensibility via a secure API and its foundation on an open-source stack, offering a blueprint for similar initiatives.

II. Related Work

The field of engineering intelligence has seen significant growth, with various tools and research aiming to provide visibility into the software development lifecycle. Our work is positioned within this landscape.

Commercial Engineering Intelligence Platforms: Tools like Hatica, LinearB, and Jellyfish [7] offer sophisticated platforms by integrating with common development tools. While powerful, they often come with significant licensing costs and may offer limited customizability for organization-specific metrics or data sources. Our approach, utilizing an open-source core (Metabase, MongoDB), provides a high degree of flexibility and extensibility at a lower operational cost.

Platform-Integrated Analytics: Tools like GitLab [4] and GitHub offer built-in analytics, including DORA metrics. These are valuable but are often confined to the data within their own ecosystem. Our framework's primary contribution is its ability to unify data from a heterogeneous, multi-vendor toolchain (Jira, Jenkins, Splunk, Firebase, etc.), providing a holistic view that platform-specific tools cannot.

Academic Research: Research in software analytics has long focused on mining software repositories to understand developer productivity and software quality. Our system builds on these principles, applying them in a large-scale industrial context. Our unique contribution is the detailed architectural blueprint for a multi-platform system that balances industry-standard metrics (DORA) with deeply contextual internal KPIs, and a discussion of practical challenges like scalability and Goodhart's Law.

III. System Architecture

The architecture is a modular, multi-layered system designed for scalability and maintainability. It comprises six core layers: Data Ingestion, Data Storage, Data Processing, Presentation, a secure API endpoint, and a cross-cutting Security layer, as illustrated in Fig. 1.

A. Data Ingestion Layer

The foundation is a robust and resilient data ingestion layer responsible for fetching raw data from a wide array of sources using a series of cron jobs orchestrated by Jenkins. Each data source has a dedicated job tailored to its volatility. The granular implementation details of the data sources and the system's error handling mechanisms are described in Appendix A.

B. Data Storage Layer

All data is stored in a central MongoDB database, chosen for its flexible, document-oriented data model suitable for semi-structured data from API responses and log files [6]. We employ a dual-schema strategy:

- **Raw Data Collections:** Store unprocessed data, providing a complete historical record for auditing and reprocessing [3].
- **Processed Data Collections:** Store computed, time-series data for each metric, optimized for efficient visualization and trend analysis.

C. Data Processing Layer

This layer consists of scripts that run in near-parallel to the ingestion jobs. Its primary responsibilities are to:

1) **Transform and Clean:** Convert raw data into a structured format suitable for analysis (e.g., parsing Git logs, standardizing user IDs).
2) **Compute Metrics:** Pre-calculate the key metrics defined in Section IV. This pre-computation is critical for application performance, acting as a caching layer.
3) **Update Timestamps:** Record a "last updated" timestamp for each metric, providing users with clear data freshness visibility.

D. Alerting Layer

To enable proactive monitoring, an alert layer continuously checks computed metrics against predefined thresholds. As seen in the mockups (Fig. 3), metrics like 'Crash Rate' have clear target ranges. If a metric breaches its threshold (e.g., 'Visitor Crash Rate' exceeds 5%), the alert layer automatically generates and sends a notification to designated Slack channels or email distribution lists. This allows teams to respond to issues like a sudden drop in quality or a pipeline failure promptly.

E. Secure Ingestion API

To enhance the platform's extensibility, a secure API endpoint (authenticated via tokens) is exposed for internal teams. This API allows other services to dump data in a predefined, structured format directly into specific MongoDB collections. By doing so, other teams are empowered to generate their own visualizations in Metabase without needing to build separate data ingestion and processing pipelines, while ensuring data sanctity, structure, and controlled access.

F. Presentation Layer (Metabase)

For data visualization and exploration, we selected Metabase, an open-source BI tool. The decision was made after evaluating several alternatives, a comparison of which is shown in Table I. Metabase offers a seamless user interface, direct connectivity to a wide range of data sources including MongoDB, and native support for writing queries. [1] Its robust support for role-based access control and integration with our single sign-on (SSO) provider, Okta, were critical factors. A key benefit is that it empowers both technical and non-technical users to explore data and create visualizations without extensive training.

TABLE I: Comparison of BI Tools

Feature	**Metabase**	**Grafana**	**Tableau Public**
Cost	Open-Source	Open-Source	Free (Public)
Ease of Use	High	Medium	High
MongoDB Connector	Native	Plugin	Limited
RBAC	Advanced	Advanced	N/A
Self-Hosting	Yes	Yes	No
SQL Support	Native GUI & SQL	Requires specific data source support	Yes

G. Security Layer: Role-Based Access Control (RBAC)

Security and data privacy are paramount. We implemented a custom authentication layer that integrates with existing SSO infrastructure. This layer manages user roles and permissions, ensuring that sensitive metrics are only visible to authorized personnel. The authentication process leverages SAML, and once authenticated, Metabase generates a session cookie that enforces the user's permissions.

IV. Key Metrics for Engineering Excellence

The selection of metrics was a deliberate process, combining industry-standard frameworks like DORA with internal goals tailored to the organization's specific needs. To establish these internal KPIs, we conducted a series of structured workshops

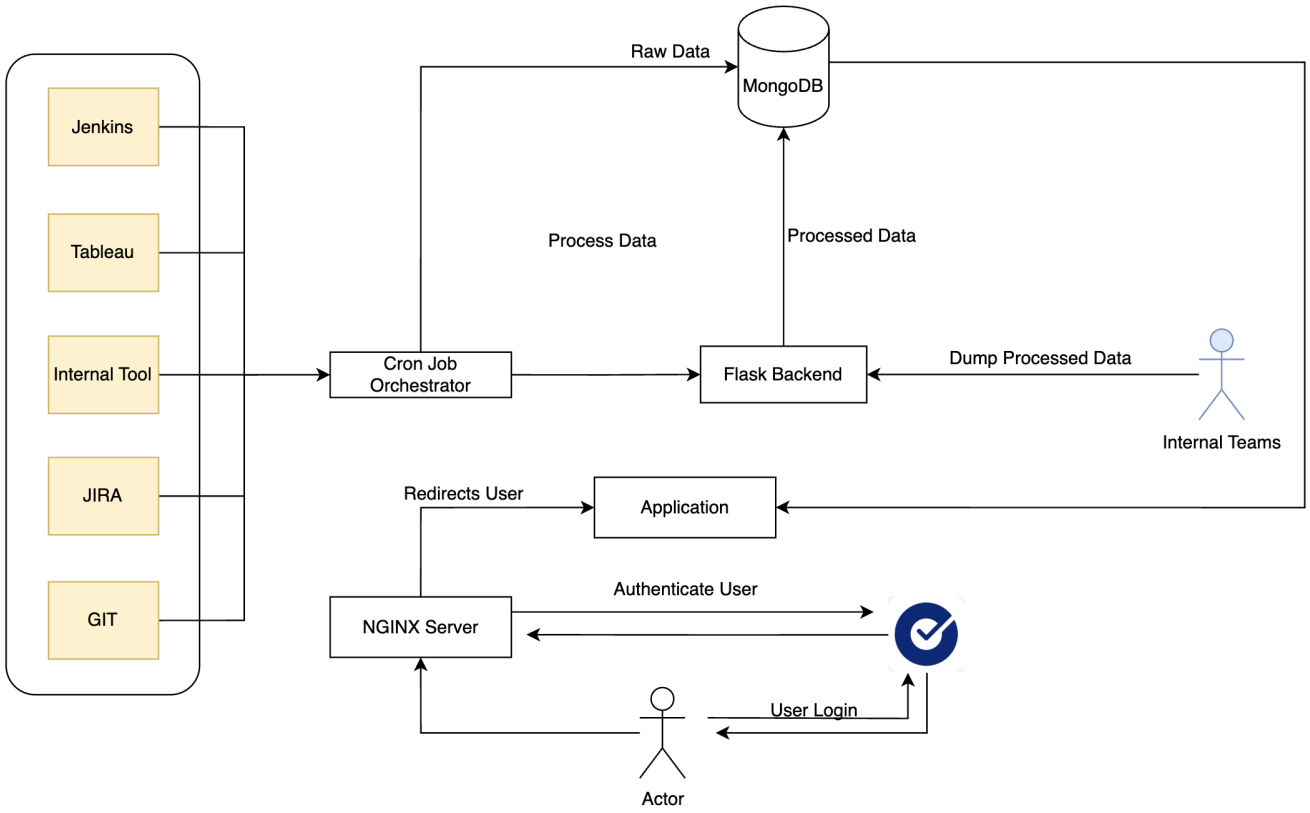

Fig. 1: High-Level System Architecture. The diagram illustrates the flow of data from various sources through the ingestion, storage, and processing layers to the Application presentation layer, with security and alerting as integral components.

with engineering leads and product managers from each platform, followed by a survey to validate the most impactful measures of developer friction and product quality. This ensured the final metric set was aligned with both strategic objectives and the day-to-day realities of our development teams. The metrics provide a balanced view across different facets of engineering, ensuring that speed is not pursued at the expense of quality. The metrics are grouped into five distinct ”Developer Areas.”

A. Developer Productivity

These metrics focus on the efficiency and velocity of the development process, from the initial act of writing code to its final integration and deployment. They are designed to identify friction points and measure the direct impact of tooling and process improvements on the developer workflow.

- **Lead Time for Changes:** A core DORA metric, this measures the median time elapsed from a code commit to that code being successfully deployed into production. This is a holistic indicator of the entire delivery pipeline’s efficiency, encompassing not just coding and review but also testing, integration, and deployment processes. A shorter lead time signifies a highly automated and efficient path to delivering value. Data is sourced by correlating commit timestamps from Git with deployment timestamps from Jenkins.
- **PR Cycle Time:** Defined as the median time taken from when a Pull Request (PR) is created to when it is merged into the main branch. It is a key indicator of development velocity and highlights potential bottlenecks in the code review and local validation processes. A consistently low PR Cycle Time suggests a healthy and efficient code review culture. Data is sourced from Git.
- **Code Commit Frequency:** This tracks the average number of commits per developer per day to the main development branch. A higher frequency often indicates smaller, more manageable batch sizes, which are a hallmark of continuous integration. It reflects a development team’s rhythm and flow, with frequent commits reducing the risk of large, complex merges. Data is sourced directly from Git history.
- **Build Induced Latency:** This metric quantifies the ”wait time” developers experience by measuring the median time taken for a continuous integration build to provide feedback after a PR is submitted or updated. This is a direct measure of friction in the inner development loop; long build times interrupt a developer’s flow, increase context switching, and slow down iteration. Data is sourced from Jenkins and Splunk.

- **PR Throughput:** The average number of Pull Requests merged per team per week. This metric helps in understanding team capacity and output over time. When analyzed as a trend, it can reveal the impact of process changes or highlight teams that may be under-resourced. Data is sourced from Git.
- **Co-pilot Assistance Metrics:** This modern metric suite is crucial for understanding the impact of AI-assisted coding tools. [5] It captures metrics such as the number of suggestions provided, acceptance and decline rates, and overall lines of code generated. A high acceptance rate can be an indicator of improved developer productivity and efficiency.

B. Quality and Reliability

This area measures the stability and quality of the product experienced by the end-users.

- **Stability / User Crash Rate:** The percentage of users who have *not* experienced a crash (Stability) or the percentage of sessions that end in a crash (User Crash Rate). This is a critical user-facing metric. Monitoring this helps in quickly identifying releases with regressions that negatively impact user experience. Data is collected from mobile services for native applications and from analytics tools for web applications.
- **Blocker/Critical Bugs:** The number of open bugs with a priority of "Blocker" or "Critical" in Jira. This provides a direct, unfiltered view of high-priority issues that may be preventing users from completing key workflows or causing significant user frustration.
- **Bug Mix Overall:** The distribution of bugs across different severity levels (Blocker, Critical, Major, Minor, Normal). As shown in the weekly and monthly graphs (see Fig. 3), a healthy trend is a downward slope for high-severity bugs and a stable or declining number of lower-severity bugs.

C. Operational Efficiency (DORA Alignment)

These metrics are aligned with DORA principles and measure the efficiency and reliability of the delivery pipeline.

- **Deployment Frequency:** The number of deployments to production per week. A core DORA metric that indicates the team's ability to deliver value to customers frequently. [2] A high frequency is indicative of a mature and automated CI/CD pipeline. Data is sourced from an internal deployment tool.
- **Main Fail Rate:** The percentage of builds that fail on the primary (main) branch over a given period. This DORA-inspired metric reflects the stability of the CI/CD pipeline and is sourced from Splunk. [4]
- **Average Turnaround Time:** The time taken from triggering a build to its successful completion, indicating the efficiency of the build and test process. Data is sourced from Splunk.

D. Automation Tooling

This area focuses on the health, effectiveness, and efficiency of our automated testing infrastructure. The goal of these metrics is to ensure that the automation suite is not just a collection of tests, but a reliable, fast, and trustworthy system that enables developers to release code with high confidence and minimal friction. A healthy automation ecosystem is a direct enabler of high-frequency, low-risk deployments.

- **Automation Code Coverage:** The percentage of code statements covered by automated tests (primarily unit and integration tests). While not a direct measure of test quality, it serves as a crucial guardrail to prevent untested code from entering the main branch. Tracking this trend over time helps ensure that test debt is not accumulating. Data is sourced from code coverage tools integrated into the CI/CD pipeline.
- **Automation Health:** The percentage of automated test suites that passed in a given build or release cycle. This is a high-level indicator of the entire test suite's reliability. A consistently high health score (e.g., 95%) builds developer trust in the CI/CD pipeline's feedback.
- **Test Automation Status:** This provides a clear view of the automation backlog and progress. It breaks down the total number of testable issues (e.g., from Jira) into distinct categories: *To be automated*, *Automated*, and *Can't be automated*. This metric is crucial for program management to understand automation velocity and resource allocation.

E. Engagement

Engagement metrics help understand how users are interacting with the product.

- **MAU (Monthly Active Users):** The number of unique users who actively engage with the product in a month.
- **Rolling MAU % (RMAU):** A trended view of user engagement over time, smoothing out short-term fluctuations to understand long-term growth or decline in the active user base. Data for both is sourced from Tableau.

V. Evaluation and Results

The application introduction has had a significant and measurable impact. We evaluate its success based on quantitative gains, qualitative improvements, and stakeholder feedback.

A. Quantitative Impact

The most direct quantitative benefit of the framework was the significant reduction in manual effort required for performance reporting across engineering, program management, and leadership. The system automated the time-consuming tasks of data gathering from disparate sources, curating weekly and monthly reports, and, most importantly, identifying cross-platform linkages that were previously obscured.

- **Time Savings:** We estimate a saving of **40 person-hours per week** across the organization. This figure is primarily based on surveys with 15 engineering and program managers who no longer need to perform manual

data collection and reporting. This automation has a cascading benefit: engineers face fewer interruptions for ad-hoc data requests, and leadership can access near-real-time insights directly, eliminating the latency of manual report generation cycles.

B. Qualitative Impact: Case Studies

The framework's true power lies in its ability to correlate data from different sources, revealing previously obscured insights.

Bottleneck Identification: In one instance, the application highlighted a sudden 30% increase in 'PR Cycle Time' for a specific platform. By cross-referencing this with other metrics, managers noticed a simultaneous spike in the 'Main Fail Rate'. This correlation, visualized in Fig. 2, immediately suggested that CI pipeline failures were blocking PR merges. A drill-down into Splunk data revealed a flaky integration test. The test was quarantined, and 'PR Cycle Time' returned to normal within **48 hours**. Without the unified view, this root cause analysis could have taken days.

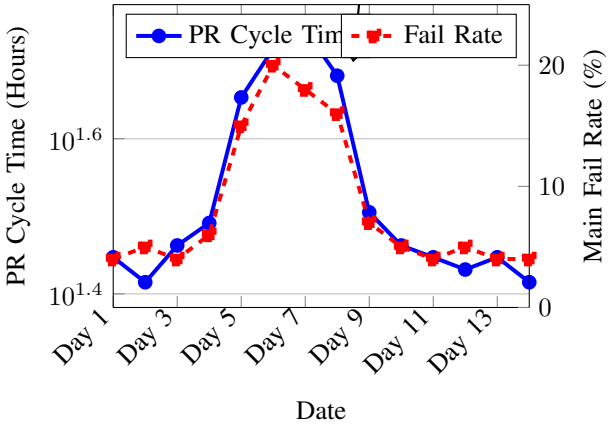

Fig. 2: Correlation of PR Cycle Time and Main Fail Rate during a CI/CD incident, illustrating the application's diagnostic power.

Productivity Tool Impact Analysis: After integrating GitHub Copilot, the application showed a clear uptick trend in 'PR Throughput' for early adopter teams, providing a strong data-driven justification for expanding its use across the organization.

C. Stakeholder Feedback

The reception has been overwhelmingly positive. A Senior Engineering Manager noted, *"For the first time, we have a single pane of glass to understand the health of our engineering organization. The ability to go from a high-level trend to a specific failing build in two clicks is a game changer."*

D. Application Visualizations

The figures below show example visualizations from the application. Fig. 3 shows a dashboard consolidating key quality and reliability metrics, while Fig. 4 focuses on operational efficiency metrics. The combination of single value metrics for at-a-glance health checks and time-series graphs provides a comprehensive view.

VI. Discussion

The implementation surfaced important considerations regarding scalability, performance, platform governance, and the responsible use of metrics.

A. Scalability

The current architecture is designed to be scalable. Onboarding a new metric or data source is a streamlined process. However, we have identified two potential scalability bottlenecks: the cron job orchestrator and the DB instance. As data volume grows, running an excessive number of serial cron jobs could lead to delays. Similarly, the database could become bloated if historical data is not managed properly. To mitigate this, the architecture can be scaled by parallelizing cron jobs. Metabase's ability to connect to multiple data sources simultaneously also means we can federate queries across different databases if needed, reducing the load on a single instance.

B. Performance Optimization: The Role of Caching

Application's latency is a critical factor for user experience. Our architecture emphasizes pre-computation. The data processing layer acts as a caching mechanism, calculating complex metrics and storing the final values in MongoDB. Metabase then simply queries this pre-computed value, resulting in near-instantaneous load times. Furthermore, Metabase's open-source edition provides a built-in caching feature. We leverage this to cache the results of frequently run queries for a specified time-to-live (TTL).

C. Threats to Validity

We acknowledge several potential threats to the validity of our evaluation.

- **Internal Validity:** Our case studies are observational and do not represent controlled experiments. While the correlation between the CI failure and the increase in 'PR Cycle Time' is strong, other unobserved factors could have contributed to this effect.
- **External Validity:** The framework was designed within the context of a large, multi-platform software organization. The specific metric choices, data sources, and architectural decisions may not be directly generalizable to smaller organizations or different software domains (e.g., embedded systems vs. web services) without adaptation.
- **Construct Validity:** We measure abstract concepts like "productivity" using proxy metrics (e.g., 'PR Throughput'). These proxies are imperfect and can be influenced by external factors. This is closely related to the risk of Goodhart's Law ("when a measure becomes a target, it ceases to be a good measure"), which we actively mitigate by focusing on a balanced set of metrics rather than optimizing for any single one.

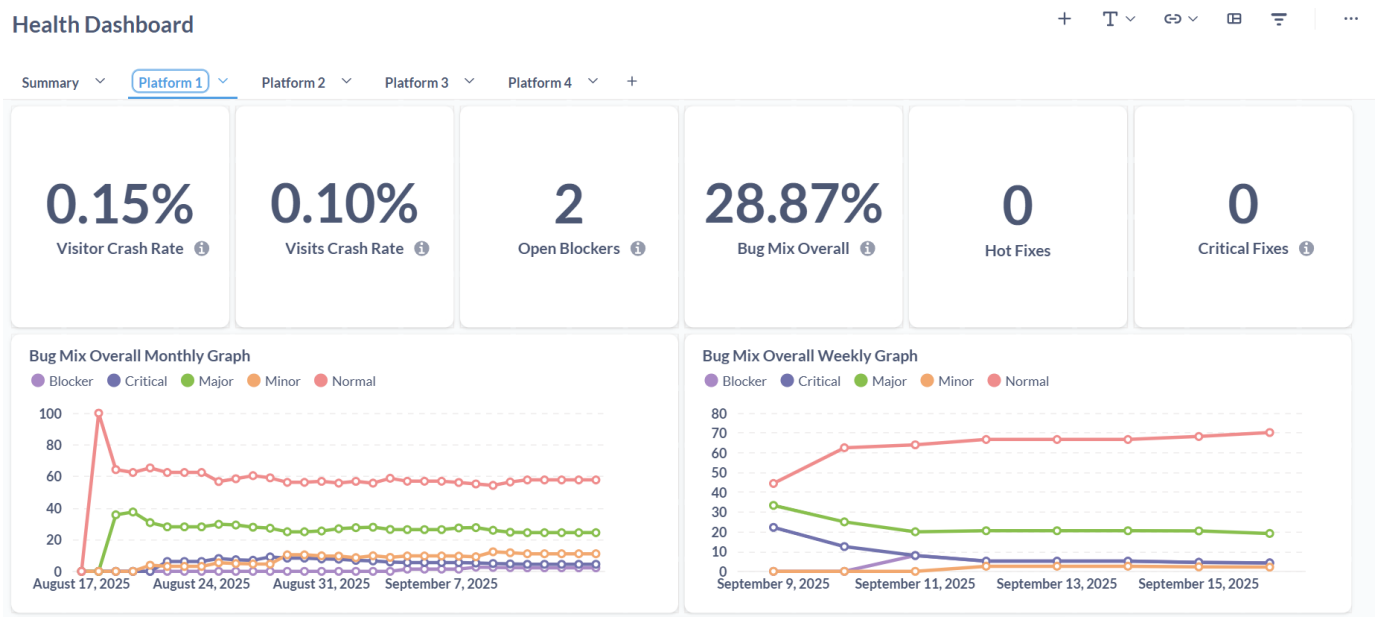

Fig. 3: Application View for a Digital Platform. This view consolidates key quality and reliability metrics, including Blockers, Fixes, Crash Rate, and the overall Bug Mix, providing an at-a-glance health assessment.

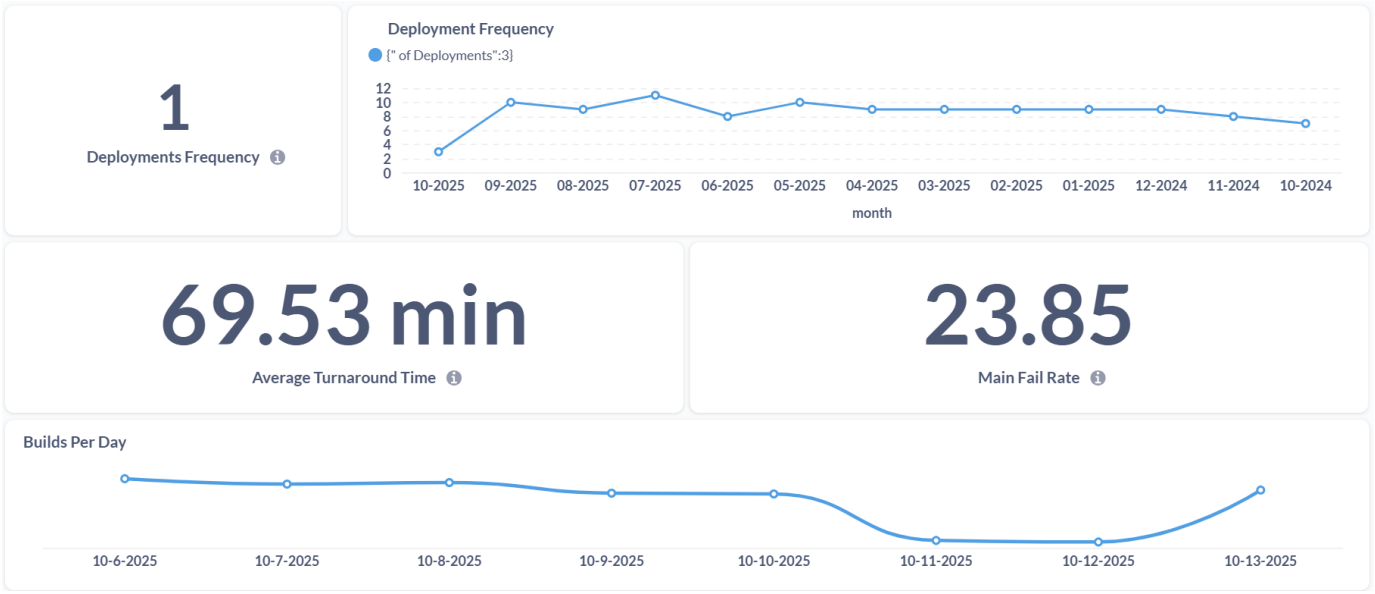

Fig. 4: Application View for a Digital Platform. This visualization focuses on operational efficienct metrics such as Deployment Frequency and Fail Rate.

D. Ethical Considerations and Metric Misuse

Implementing any system that measures developer activity requires careful consideration of its cultural impact. Our guiding principle was that this framework must be a tool for empowerment and process improvement, not for individual performance evaluation. We mitigate the risk of metric misuse in several ways:

1) **Team-Level Aggregation:** Metrics related to individual output, such as 'Code Commit Frequency,' are always aggregated at the team or organization level, never displayed for individuals.
2) **Leadership Training:** We conducted sessions with engineering managers to frame the metrics as diagnostic tools for identifying systemic bottlenecks, explicitly discouraging their use in performance reviews.
3) **Focus on Trends over Absolutes:** We encourage stakeholders to focus on trends and relative changes for a team over time, rather than comparing absolute values between teams with different contexts and charters.

E. Limitations of Metabase

While Metabase is a powerful open-source tool, its visualization options are not as extensive as some commercial BI tools. Performance can also degrade when non-technical users attempt ad-hoc queries on very large raw datasets, sometimes leading to query timeouts or browser performance issues. [8]

VII. Future Work

Our future research agenda is focused on enhancing the intelligence of the platform.

- **Anomaly Detection:** We plan to move beyond static, threshold-based alerting by implementing machine learning models (e.g., ARIMA or LSTM networks) to automatically detect statistically significant anomalies in key metrics. This builds on established research in time-series anomaly detection [10].
- **Predictive Analytics:** We will leverage historical data to build predictive models that can forecast project delays or identify teams at risk of burnout based on trends in PR throughput and cycle time. The validity of these models will be tested against qualitative data from team surveys.
- **Deeper Integration:** We aim to create more interactive UI that allow users to take action directly from the view, such as clicking a "stale PR" metric to trigger a Slack reminder to the assigned reviewers, closing the loop between insight and action.

VIII. Conclusion

The centralized engineering performance framework has successfully transformed how the organization monitors and improves its software development lifecycle. By consolidating data from disparate sources into a single, cohesive view, the platform has eliminated hours of manual report generation, enabled faster, data-driven decision-making, and fostered a culture of shared ownership over key performance indicators. The architecture, built on a foundation of resilient data ingestion, pre-computation, and a flexible open-source BI tool, has proven to be both scalable and performant. The thoughtful selection of DORA-aligned and internally defined metrics provides a comprehensive understanding of developer productivity, product quality, and operational efficiency. This work demonstrates that a well-architected analytics platform is an indispensable tool for any large-scale engineering organization aiming for continuous improvement and engineering excellence.

Appendix

A. Data Sources and Fetching Schedules

- **Git:** Daily cron jobs extracts commit history, branch information, and pull request metadata.
- **Jira:** Fetched daily to gather data on issue types (bugs, stories), priorities, statuses, and resolution times.
- **Splunk:** High-frequency jobs (e.g., hourly) query Splunk for application logs, build failures, and performance data.
- **Crash reporting service:** Fetched daily for detailed, symbolicated crash reports and stability metrics.
- **Mobile app store console:** Daily jobs connect via API to fetch Android-specific quality metrics like ANR rates.
- **Tableau:** Daily jobs pull pre-aggregated business metrics, such as Monthly Active Users (MAU).
- **Other Sources:** Data from internal deployment tools, web analytics, and custom APIs are pulled at corresponding frequencies.

B. Resiliency and Error Handling

The ingestion layer is designed for fault tolerance. If a Jenkins cron job fails, an automated retry mechanism is triggered up to five times with increasing backoff intervals. Should the failure persist, an alert is generated on a dedicated developer Slack channel, and the system uses the last successfully fetched data to prevent application downtime. To handle API rate limits, we respect 'Retry-After' headers and distribute requests evenly. The system also monitors credential expiration and sends proactive alerts to administrators to ensure uninterrupted data flow.

References

[1] RisingWave, "Understanding Metabase's Core Features," 2024. [Online]. Available: https://www.risingwave.com/blog/understanding-metabases-core-features/

[2] Wikipedia, "DevOps Research and Assessment," 2024. [Online]. Available: https://en.wikipedia.org/wiki/DevOps_Research_and_Assessment

[3] MongoDB, "Analytics Driven By MongoDB," [Online]. Available: https://www.mongodb.com/analytics

[4] GitLab, "DevOps Research and Assessment (DORA) metrics," [Online]. Available: https://docs.gitlab.com/ee/user/analytics/dora_metrics.html

[5] McKinsey & Company, "Yes, you can measure software developer productivity," 2023. [Online]. Available: https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/yes-you-can-measure-software-developer-productivity

[6] GeeksforGeeks, "MongoDB Analytics for Big-Data," 2025. [Online]. Available: https://www.geeksforgeeks.org/mongodb-analytics-for-big-data/

[7] Jellyfish, "Goodhart's Law in Software Engineering and How to Avoid Gaming Your Metrics," Sep. 07, 2022. [Online]. Available: https://jellyfish.co/blog/goodharts-law-in-software-engineering/

[8] DashboardFox, "Metabase: Pros and Cons (Straight Talk Review)," Oct. 08, 2021. [Online]. Available: https://dashboardfox.com/blog/metabase-pros-and-cons-straight-talk-review/

[9] N. Forsgren, J. Humble, and G. Kim, *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*. IT Revolution Press, 2018.

[10] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM computing surveys (CSUR)*, vol. 41, no. 3, pp. 1-58, 2009.