

Master's Programme in Computer, Communication and Information Sciences

Enhancing Software Release Velocity

Through Integrated Developer Tooling and Workflow Notifications

Maria Syed

Master's thesis
2025

Copyright ©2025 Maria Syed

Author	Maria Syed	
Title of thesis	Enhancing Software Release Velocity	
Programme	Computer, Communication and Information Sciences	
Major	Computer Science	
Thesis supervisor	Prof. Casper Lassenius	
Thesis advisor(s)	Aleksandr Dzhioev, MSc	
Date	Number of pages	Language
18.11.2025	99	English

Abstract

This thesis investigates critical software delivery latency at a large fintech organization, where a modern micro-application architecture was severely bottlenecked by a manual, ticket based approval system. This hybrid environment created an acute organizational bottleneck, imposing high coordination burdens and unpredictable delays on globally distributed feature teams.

Using an Action Research (AR) methodology, the study first established a high-friction baseline, measuring the median Lead Time for Changes (LTC) at 20.2 hours. The core intervention involved replacing the mandatory manual approval gate with a fully automated, self-service deployment model integrated directly into the Continuous Integration/Continuous Delivery (CI/CD) pipeline.

The intervention successfully drove significant organizational efficiency, yielding a 69% reduction in LTC, dropping the median time from 20.2 hours to 6.2 hours. Concurrently, Deployment Frequency (DF) increased by 213% (from 47 to 100 releases per week). This improvement solidified the organization's position within the DORA elite performance tier.

The primary practical guidance derived from this case study is that sustained software acceleration requires prioritizing the decentralization of control over the deployment trigger. This is achieved not merely through technical automation, but by deliberately eliminating all mandatory human coordination steps via external systems (e.g., tickets), relying instead on real-time visibility tooling integrated into the developer workflow. Additionally, and more importantly, this required a complementary organizational culture shift, which involved transferring accountability for production stability directly from administrative roles, such as the Program Manager, to the autonomous development teams.

Keywords Software Release Management, Release Automation, Continuous Delivery, DevOps, Software Delivery, Deployment Notification System

Table of contents

Preface and acknowledgements	6
Symbols and abbreviations	7
Symbols.....	7
Abbreviations.....	7
1 Introduction	9
1.1 Goals and Objectives of the Study.....	10
1.2 Research Questions	10
1.3 Scope and Limitations of the Study	11
1.4 Significance of the Study	11
1.5 Structure of Thesis	11
2 Literature review.....	13
2.1 Foundations of Modern Software Delivery.....	13
2.2 Continuous Delivery.....	17
2.2.1 Core Principles of Continuous Delivery	18
2.3 Software Delivery Performance	20
2.3.1 Throughput Metrics	22
2.3.2 Stability Metrics	22
2.4 Architectural Evolution	23
2.4.1 Monolithic Release Architecture	23
2.4.2 Decomposed Micro-frontend Architecture	24
2.5 Comparative Case Studies.....	26
2.5.1 Benchmark Quantitative Data.....	27
2.5.2 Procedural Comparisons	28
2.5.3 Organizational Alignment.....	28
3 Methodology	30
3.1 Research Design	30
3.2 Data Collection	31
3.2.1 Quantitative Data.....	31
3.2.2 Qualitative Data	32
4 Results.....	35
4.1 Current State Analysis.....	35

4.1.1	Frontend Architecture Evolution.....	35
4.1.2	Software Delivery Evolution	38
4.1.3	Baseline Quantitative Analysis	42
4.1.4	Baseline Qualitative Analysis.....	50
4.2	Technical Implementation	55
4.2.1	Core Design Requirements	56
4.2.2	Release Tracker Dashboard	60
4.2.3	Real-Time Notification System.....	70
4.3	Addressing Challenges	78
4.4	Organizational Rollout	79
5	Validation.....	81
5.1	Quantitative Analysis	81
5.1.1	Deployment Frequency	81
5.1.2	Lead Time for Changes	83
5.1.3	Comparison of Performance Metrics.....	87
5.2	Qualitative Analysis.....	88
6	Discussion	91
6.1	Summary	91
6.1.1	Cultural Shift	92
6.2	Comparison with Industry Benchmarks and Literature	93
6.2.1	Relationship to Continuous Delivery Principles	94
6.3	Addressing Research Questions.....	95
6.4	Implications.....	96
6.5	Conclusion	96
	References	97

Preface and acknowledgements

I want to thank Professor Casper Lassenius for his guidance and sustained support. I also want to thank my advisors, Maciej and AJ, for providing the opportunity to conduct this research within the organization, and for their practical assistance.

I also want to thank my wise fiancée, Henri Eskelinen, for his encouragement and motivation.

Otaniemi, 18 November 2025

Maria Syed

Symbols and abbreviations

Symbols

L1	Median time elapsed between Pull Request approval and merge.
L2	Median time elapsed between Pull Request merge and production deployment.
L2a	Median time elapsed between Pull Request merge and ready for production deployment.
L2b	Median time elapsed between Pull Request ready for deployment and deploy to production.

Abbreviations

DIP	Data Intelligence Provider
PBSD	Plan Based Software Development
ASD	Agile Software Development
FTP	File Transfer Protocol
DevOps	Development Operations
ITSM	IT service management
CI	Continuous Integration
CD	Continuous Delivery
DORA	DevOps Research and Assessment
DF	Deployment Frequency
LTC	Lead Time to Changes
CFR	Change Failure Rate
MTTR	Mean Time To Recovery
HTTP	Hypertext Transfer Protocol
SRP	Single Responsibility Principle
AR	Action Research
PR	Pull Request
QA	Quality Assurance
UI	User Interface
RC	Release Candidate
VCS	Version Control System
IQR	Interquartile Range
DB	Database
NPM	Node Package Manager
FaaS	Functions as a Service
AWS	Amazon Web Services
REST	Representational State Transfer
RegEx	Regular Expression
SSO	Single Sign On

UUID Universal Unique Identifier
YAML Yet Another Markup Language

1 Introduction

Software release management has undergone a significant transformation, evolving from manual processes with granular controls to modern, automated continuous deployment models. The necessity for speed and agility is paramount since, the ability to deploy software quickly and reliably has become a key competitive advantage, enabling companies to respond to market demands, customer feedback, and emerging opportunities with agility [1].

However, as software systems become more complex and interconnected, managing releases has grown increasingly challenging. These inefficiencies in the release processes lead to delays, deployment failures, and miscommunication, which can negatively impact productivity, operational costs, and customer satisfaction [1]. Many organizations that initially deployed updates at a slow and controlled pace have found themselves struggling to scale to managing hundreds or even thousands of deployments per week, exposing the failure of manual and tedious release practices [1].

This structural disparity between the required organizational velocity and the constraints of tedious release processes creates a critical friction point. This thesis seeks to directly address this systemic challenge by investigating and intervening within a specific, high criticality operational environment.

Therefore, this research investigates the frontend release workflow at a large, global enterprise and proposes targeted refinements to accelerate its software delivery. To protect confidentiality, the company is referenced throughout this thesis by the pseudonym, Data Intelligence Provider Limited (DIP).

DIP is a top provider of AI-powered market intelligence, serving institutional investors and financial analysts in a highly competitive environment. Founded in early 2010's, the company successfully anticipated much of the modern demand for data driven decision making, but it now faces intensified rivalry with the rapid rise of generative AI solutions. In this context even small delays in delivering new features or fixes can erode customer trust and weaken DIP's position in the market. As software only delivers value when it reaches end users [1], the journey from idea to production must therefore remain as frictionless as possible.

One point of friction to achieving a seamless delivery to production, which is concurrently the final step, is the production release process. Although this process has been iteratively refined as the company has scaled globally, it still retains vestigial elements of earlier, less efficient practices that may continue to impede its velocity.

Initially, the release process at DIP had followed a cadence inspired by the waterfall methodology, where several updates were bundled within one release artefact. This allowed predictable and carefully controlled releases as each release was tracked through a project management ticketing system and required the necessary approvals from Release Managers.

As the organization grew, it gradually shifted to a more agile delivery cadence with updates released independently. However, in its present state, it continues to remain subject to the ticketing system to record and monitor release progress while maintaining the formal approval step for a designated subset of releases. This thesis will explore these dynamics in detail and identify incremental improvements to enhance overall release velocity while ensuring the capabilities provided by the release ticketing system continue to be satisfied.

1.1 Goals and Objectives of the Study

The primary goal of this thesis is to enhance the release management process of DIP, for specifically the frontend repository, by leveraging automation tools and integrating real-time tracking systems. The research aims to develop a comprehensive understanding of the challenges faced in managing software releases and to propose practical solutions to address these challenges. The specific objectives of the study are as follows:

- To analyse the current frontend release management practices within the company and identify key pain points.
- To design and implement a solution for tracking frontend releases, integrating with existing pipeline tools.
- To incorporate notifications for real-time updates on release statuses in order to reduce the need for manual tracking.
- To evaluate the impact of these tools on release efficiency and team collaboration.

1.2 Research Questions

The study addresses the following key research questions:

1. How can the process of shipping features be streamlined without relying on release tickets?
2. What are the key challenges and benefits of automating the release process within a micro-application architecture?

These questions guide the research, focusing on practical solutions to enhance release management and contribute to the broader field of software engineering.

1.3 Scope and Limitations of the Study

The scope of this study is limited to the frontend release process of a company transitioning to an agile cadence of software delivery. The research focuses on the development and implementation of tools to streamline release management, including a dashboard for tracking releases and workflow integrations for real-time notifications. While the study provides valuable insights into the benefits of automation and tool integration, it is constrained by the specific context of the case study company.

Limitations of the study include the reliance on data from a single organization, which may affect the generalizability of the findings. Additionally, the study focuses primarily on frontend release processes, and the results may not fully capture the complexities of backend or full-stack release management. Future research could expand on these findings by exploring similar solutions in different organizational contexts and across various components of the software stack.

1.4 Significance of the Study

The significance of this study lies in its practical contribution to both DIP and similar software enterprises. By examining a transition from a centralized waterfall model for the release process to a hybrid micro-application architecture, this research surfaces the specific coordination challenges and inefficiencies that emerge during organizational growth and architectural change. The incremental refinements proposed here, ranging from streamlined release workflow to an asynchronous notification system, offer actionable guidance for Release Managers and engineering leaders seeking to reduce cycle times, increase deployment frequency, and preserve developer autonomy.

Together, these contributions show that even in complex, high cadence environments, carefully designed process improvements can unlock significant gains in throughput and quality. The insights generated here will be of value to practitioners grappling with similar scaling challenges and to researchers investigating the mechanisms by which release practices influence organizational agility and competitive advantage.

1.5 Structure of Thesis

The structure of this thesis is designed to provide a comprehensive exploration of software release management and the proposed solutions to enhance it. The thesis is organized into the following chapters:

- Chapter 1 provides an overview of the study, including the background, goals, research questions, scope, and significance.
- Chapter 2 systematically reviews and synthesizes academic and industry literature on the foundations of software delivery, decomposed architectures, software delivery performance metrics and comparative studies.
- Chapter 3 outlines the methodology employed in this study, describing the research design, and data collection procedures.
- Chapter 4 describes the case background, presents the baseline performance analysis, and details the technical design and implementation of the proposed solution.
- Chapter 5 conducts a quantitative evaluation of the intervention's effectiveness, presenting the empirical results by systematically comparing the established baseline metrics with the post-intervention performance data.
- Chapter 6 provides a comprehensive synthesis of the study's findings, critically contrasting the empirical results with the theoretical frameworks established in the literature review and definitively addressing the research questions.

By addressing the challenges of release management, this thesis aims to provide actionable insights and practical solutions that contribute to the broader field of software engineering. The implemented solution has the potential to benefit development teams by improving workflow efficiency, fostering better collaboration, and enabling data driven decision making through the tracking of key performance metrics.

2 Literature review

This chapter systematically reviews and synthesizes the academic and industry literature concerning modern software delivery, establishing the theoretical and empirical context for the analysis of the frontend release process at DIP. The review is specifically tailored to investigate the intersection of high-velocity practices and architectural complexity, which are central to the challenges outlined in Chapter 1.

The goal of this synthesis is to first validate the strategic business imperative of continuous, frictionless software flow, and second, to identify the theoretical basis for eliminating vestigial manual governance steps, such as release ticketing, within an agile, decomposed environment.

This chapter is organized to follow a logical progression, first defining the ideal environment and then scrutinizing the key point of friction that is the focus of this thesis. It is organized as follows:

- Section 2.1 defines the foundations of software delivery, tracing the philosophical shift to the modern, autonomous delivery model.
- Section 2.2 reviews the principles for continuous delivery, establishing the core prescriptive model for release procedures.
- Section 2.3 reviews the software delivery performance metrics, defining the industry standard framework for measurement.
- Section 2.4 reviews architectural evolution, detailing the industry transition from monolithic to micro-service systems.
- Section 2.5 presents comparative studies to provide external benchmarks and define the research gap this thesis aims to address.

2.1 Foundations of Modern Software Delivery

The progression of software engineering methodologies provides the essential context for modern delivery practices, charting a movement away from rigid, sequential processes towards adaptive, collaborative systems. This foundational shift is characterized by the transition from the “Waterfall model”, often referred to as Plan Based Software Development (PBSD), to Agile Software Development (ASD). While the Waterfall model relies on creating a rigid, upfront plan that strictly sequenced phases, ASD emphasizes adaptability, collaboration, and iterative development [2].

The foundation for understanding modern software delivery lies in analysing the traditional approaches it sought to replace, primarily the Waterfall model.

As early as the 1970s, a research paper published by Winston Royce, a computer scientist, described a release process that relies on rigid project management system where software is delivered linearly [3]. Eventually, this approach would be formally recognized as the "Waterfall Methodology", which involves unidirectionally cascading down through stages of development, that begin with specifying the requirements to developing the design solutions to eventually planning and executing the implementation, after which meticulously verifying the quality of the produced software and then finally reaching the last stage which is providing ongoing maintenance as required [4]. The "Waterfall Methodology" strongly cemented itself as a traditional project management workflow which mainly suited the needs of projects with few complicated requirements, predictable outputs and with little likelihood of requirements shifting in later stages of development [4].

By the late 1990s, this methodology remained a standard in software companies with several configuration management tools assisting in facilitating this process, such as Aide de Camp, ClearCase, and Continuous helping to track and organize software versions [5]. At that time, once a release was finalized, its components were locked, labelled, and stored within the configuration management system. Nevertheless, this distribution process had several pitfalls towards seamlessly discovering software versions, since users had to be informed of a release through announcements, advertisements, or via direct communication with the development organization so they could request access to the software [5].

Due the widespread adoption of the Internet in the 1990s, the path through which software was released and distributed had radically shifted. Instead of relying on manual distribution methods, organizations began hosting their software on File Transfer Protocol (FTP) servers and web platforms, allowing users to access and download updates directly. These online platforms made it easier for companies to share product details, distribute both free trials and licensed versions, and streamline software availability on a global scale [5]. Additionally, the rise of search engines, software repositories, and indexing services made it easier for users to discover and retrieve software efficiently.

Although there have been major strides in mitigating pitfalls with the traditional methodology of software release processes, this process had critical shortcomings due to the limited flexibility of the process itself. It dictates that each phase is required to be completed with all issues resolved before initiating the subsequent phase, in addition, faults and oversights that arise after a phase had begun could not be addressed immediately [4]. The traditional approaches often struggled to meet project deadlines and deliver software aligned with customer expectations [6].

As software development expanded, so did the demand for faster and more reliable release cycles. Despite the critical role software delivery plays in generating business value, many organizations still rely on manual release processes which are prone to errors and are tedious. The increasing complexity of modern software systems, combined with the pressure to release frequently and reliably, has made traditional software release practices less effective in the current rapidly changing technology landscape [1].

By 2005, although the traditional methodology had still been widely in use, a new set of methodologies called the agile development methodology had started to spread through the software development sphere. The new methodology claimed to mitigate several shortcomings of the traditional method. While the traditional method is heavily reliant on processes as it entails incrementally developing software through meticulous planning in advance, ensuring maximal control through measuring, predicting and refining the processes [2], the new agile development methodology relies on smaller incremental updates, great collaboration within the team and adaptability to shifting specifications.

This transformation introduced new methodologies, such as Scrum and Kanban, aimed at accelerating software delivery and ensuring more regular updates to customers. The Scrum framework requires complex projects to be split into smaller, incremental batches called sprints, and introduces specific roles within the team, for example the Scrum Master and Product Owner, to maximize efficiency and collaboration. In contrast, the Kanban methodology mandates that work items be approached as a continuous flow of smaller tasks, visualized on a board with strict limits on the maximum number of tasks in progress. The successful application of these methodologies enabled companies to increase productivity and reduce costs, as software was released in small increments over time. Furthermore, this approach increased employee engagement and satisfaction since the engineers developing the software were more involved with handling releases, thereby decentralizing the release management control [7].

This methodology shift gained significant traction as organizations sought to leverage the advantages of these Agile methodologies. Companies were able to increase productivity across their engineering organization due to increased efficiency and transparency, which in turn resulted in reducing waste and saving costs. At the same time, companies witnessed an increase in employee engagement and job satisfaction due to the deeper involvement of engineers in the development lifecycle and increased collaboration. Owing to the practice of shipping software in smaller increments and continuous feedback from stakeholders, the deliverables were of higher quality and quicker to reach the market at the same time [8].

Despite the myriad of advantages that the Agile methodologies offer over the traditional Waterfall model, companies with practices already cemented in the traditional style faced several hurdles while transitioning their release

management system to comply with the Agile standard. Most of these challenges were due to a resistance in reorienting cultures to the new set of principles. Although employees were required to undergo intensive training, integrating Agile successfully required a vital dramatic shift in mindset towards collaboration, transparency and setting smaller milestones [9].

This literature underscores this as a significant challenge, which is the requirement for a mindset change away from the structured nature inherent in the Waterfall model. Organizations focusing exclusively on implementing Agile practices often fail to reap the full benefits if the underlying mindset, which emphasizes trust, responsibility, and decentralized decision making, remains plan driven and centralized, constituting a complex change management process [9]. Furthermore, prevalent obstacles hindering adoption included organizational resistance, existing rigid frameworks, and critically, concerns about losing management control and changing the mindset of project managers [10]. This highlights that top management support emerges as a critical success factor for Agile transformations, stressing its commonality in successful change efforts [11].

The shift to Agile also involves complex organizational adjustments, including the redefinition of roles and the transition to empowered, autonomous teams. Unlike the Waterfall model, which relies on extensive documentation and lengthy development cycles, Agile prioritizes working software and in person communication [2], advocating for shorter timeframes and incremental delivery [8]. This culture also promotes a mentality towards failing quickly and frequently, requiring an increased level of confidence and psychological safety within the team to foster continuous experimentation and innovation [7].

To further build on the challenges of integrating Agile into companies, a survey conducted by A. S. Campanelli et al among expert practitioners, reports on major impediments to implementing a model for measuring release statistics, and project managers being resistant to change due to the loss of control [10]. Furthermore, another survey conducted by S. Obrutsky resulted in similar findings, where organizations reported resistance to change, the inability to easily restructure existing frameworks and the lack of employee training, among other frictions [9].

Considering these key cultural challenges, strategies to successfully integrate Agile methodologies have been extensively studied through researching companies. The conclusion of one of these studies states that small companies benefit from an incremental evolutionary shift into Agile mindset. This involves introducing Agile practices in phases through transparent coherent communication and strong support from management while ensuring alignment with organizational norms. With this style, the organization would have sufficient time to adapt to the new Agile methodology and mindset while slowly shifting existing systems [7].

Ultimately, the limitations encountered in even advanced Agile environments, particularly the organizational and technological silos between development and operations, led to the emergence of the Development Operations (DevOps) methodology [12].

DevOps is widely considered an evolution of the Agile movement [12], taking the principle of continuous and iterative delivery and extending its scope. While the Agile method focuses specifically on software development [13], DevOps represents a behavioural evolution aimed at integrating software development with implementation and operation.

This integration enhances the stability of the development cycle by focusing on checking the level of effectiveness of what is produced in the work cycles [14] through continuous monitoring [15]. In the professional community, DevOps is often conceptualized as the combination of existing practices, where DevOps is a combination of Agile, Lean and Information Technology Service Management (ITSM) [16].

The union of these methodologies accelerates the software release process and increases software quality, specifically in terms of reliability and maintainability [17]. This acceleration and standardization are achieved through the technical implementation of Continuous Integration (CI) and Continuous Delivery (CD) pipelines [18].

CI incorporates automating the merging of code changes with the rest of the system through pipelines to ensure each change is compatible with the entirety of the application ecosystem. While CD is the process of automating the process leading up to the deployment of code to the production environment. These processes are supported by creating cross-functional teams, promoting collaboration that extends beyond traditional Scrum roles to include Release Engineers and System Architects [18].

The following sections will explore the core themes of CD and solidify the theoretical basis upon which this thesis is grounded on.

2.2 Continuous Delivery

CD is the engineering practice that turns the theoretical promise of DevOps culture into a functional reality. It is defined as a set of capabilities that enable software to be developed in short cycles and reliably released at any time. Unlike Continuous Deployment, which automatically pushes every change to production, CD ensures that code is always in a deployable state, allowing the business to choose when to initiate the final release to users [18].

The core of CD is the deployment pipeline, which automates every step of the process from code commit to release preparation, ensuring repeatable, high-quality builds. This pipeline is built on the foundation of CI, where developers merge their code frequently into a shared repository. CI immediately triggers automated testing to provide rapid feedback, making problems visible and fixable within minutes [1].

By ensuring that the release process is fully automated and reliable, CD significantly reduces deployment risk, enables faster feedback loops, and makes releases routine rather than a rare occurrence.

2.2.1 Core Principles of Continuous Delivery

The successful implementation of CD, and the realization of its corresponding organizational benefits, requires adherence to a set of prescriptive principles that govern both technical practice and organizational culture [1].

These core tenets, published by Humble et al [1], were established through the rigorous analysis and synthesis of empirical data drawn from numerous successful software delivery projects over an extended duration. These principles serve as the essential guidelines for optimizing the speed and reliability of the release process [1].

1. Create a Repeatable, Reliable Process for Releasing Software

The paramount objective of CD is to transform software release into an everyday event by establishing a process so thoroughly tested and automated that it is repeatable and reliable. Achieving this state demands prioritization across three deployment tasks: provisioning and managing the target environment, installing the precise application version, and accurately configuring the application's state and data. The aim must be to automate this entire lifecycle, grounding all configurations in version control and utilizing modern virtualization tools for infrastructure provisioning [1].

2. Automate Almost Everything

To enable a repeatable process, this principle states that organizations must commit to automating almost everything within the delivery pipeline. While human judgment remains necessary for certain activities, such as exploratory testing or regulatory compliance approvals, the vast majority of tasks, including build processes, acceptance tests, database schema management, and network configurations, can and should be fully automated. Automation is considered the foundational prerequisite for the deployment pipeline, guaranteeing consistent outcomes. Teams are advised to adopt automation gradually, beginning with the phase of the delivery process that currently constitutes the greatest bottleneck [1].

3. Keep Everything in Version Control

The principle of keeping everything in version control is essential for ensuring full traceability and reproducibility across the delivery

lifecycle. This mandates that all components necessary to build, deploy, test, and release the application, including requirements, automated test cases, and configuration and deployment scripts, must reside in versioned storage. Critically, every element contributing to a specific build must be immediately identifiable via a single change set or build number, allowing immediate correlation between the deployed application, its environment, and its exact source materials [1].

4. **If It Hurts, Do It More Frequently, and Bring the Pain Forward**

Perhaps the most critical heuristic is to address painful processes as early and frequently as possible. This practice encourages teams to move typically deferred activities such as integration, comprehensive testing, and releasing, to the start of the project and execute them continuously. By minimizing the time interval between these activities, teams prevent the accumulation of errors and complexity, thereby reducing the pain associated with each event. This approach strongly encourages the pursuit of increasingly frequent releases [1].

5. **Build Quality In**

This principle mandates that teams must prioritize preventing defects and be disciplined about fixing defects immediately as they are identified by automated testing. Furthermore, quality assurance is not treated as a discrete testing phase at the end of development, nor is it the exclusive domain of testers rather, responsibility for the application's quality should be shared across every member of the delivery team [1].

6. **Done Means Released**

A feature only yields value when it is actively utilized by the users. This principle states that the minimum acceptable standard for the definition of done is successful demonstration and usage by user representatives from a production-like environment [1].

7. **Everybody Is Responsible for the Delivery Process**

Realizing this principle necessitates continuous, cross-functional communication and the implementation of transparent, visible systems. These systems must display the real-time status of the application, including its health, build status, and deployment environment state, thereby enabling collective responsibility and fostering greater collaboration [1].

8. Continuous Improvement

The final principle is Continuous Improvement, recognizing that the delivery process, like the application itself, must evolve throughout its lifecycle. This requires the entire team, transcending organizational silos, to regularly engage in formal retrospectives in order to continuously apply incremental improvements [1].

Ultimately, the adoption of these eight principles fundamentally transforms the delivery capability. Through these automations and the adoption of complementary good practices, the software delivery workflow across all environments becomes verifiable and reproducible, drastically eliminating the ingress of production errors. This systemic approach ensures changes can be deployed rapidly, allowing business value to be realized without friction [1].

As demonstrated, the success of DevOps relies heavily on shifting to a shared DevOps culture rather than merely implementing tools. This culture advocates for development and infrastructure teams working together towards the same goal [19], fostering empathy and trust, which in turn increases the freedom for professionals to experiment and innovate [20].

However, adopting these DevOps methodologies remain a significant challenge since it lacks a simple roadmap or standardized approach for its implementation [21]. This necessitates that organizations define their own integration processes and standards based on their unique level of maturity, justifying the need for empirical research on specific organizational transitions. DevOps leverages principles of automation, measurement, and shared responsibility to achieve the highest standards of continuous delivery.

As organizations incorporate DevOps tools and practices to leverage the speed and efficiency of releases, it remains crucial to be able to quantifiably measure the performance of software delivery to ensure the systems are simultaneously evolving and maintaining stability. The next section will explore the software delivery performance in more detail and provide the key success metrics to measure them.

2.3 Software Delivery Performance

With the emergence of practices and tools for incremental and frequent software updates, companies quickly adopting these practices have concurrently observed a proportional increase in organizational performance. The ability to deploy software quickly and reliably directly translates into competitive advantage since it enables organizations to pursue an experimental approach to product development by rapidly validating assumptions via techniques like A/B testing [1].

This is evidenced by the transformation at Amazon in 2014, which illustrates the direct link between architectural change and process

automation. When Amazon restructured into small, autonomous teams managing independent services, their manual release process became an immediate bottleneck. The implementation of their internal deployment engine, Apollo, removed this friction by reliably coordinating updates across vast fleets of hosts without downtime. This system was engineered to manage complexity, handling rolling updates, monitoring failure thresholds, and balancing deployments across multiple data center, which collectively enabled the company to achieve an average of over one deployment per second and fuel a constant stream of feature improvements [22]. Similarly, Etsy successfully transitioned from infrequent, manual releases to deploying code hundreds of times per week, dramatically reducing their lead time for features and critical bug fixes [23].

These cases highlight how optimizing release processes directly enhances organizational performance by accelerating the delivery of new features, enabling rapid course corrections, and ensuring faster resolution of defects or outages [24].

Despite the demonstrable qualitative benefits of optimizing the delivery pipeline, a critical requirement for establishing process maturity is the quantitative validation of performance improvements. To ensure that organizational initiatives are driving tangible business value, a standardized set of metrics must be employed to provide an objective north star for continuous improvement.

With this goal, Google Cloud founded a program in 2014, called DevOps Research and Assessment (DORA), dedicated to deciphering the practices that spearhead delivery of software and the performance of organizational operations. The program achieves this by conducting surveys annually across a diverse set of engineering teams in the industry, gathering information about the performance of software delivery of their principal service or application [25].

The responses are analyzed through cluster analysis, which is a statistical method to classify responses which share similarities to create contrasting clusters of data, then subsequently, published on the yearly State of Devops Report [25].

The DORA program additionally introduced a set of metrics to measure DevOps performance in organizations called the DORA metrics. This framework divides performance into two complementary dimensions, which are throughput and stability. These are measured by four key metrics to benchmark teams into categories ranging from low to elite performers, providing clear targets for improvement [26].

The DORA metrics have since become widely regarded as the industry standard for measuring DevOps performance since they provide an empirical link between engineering practices and organizational outcomes, such as profitability and market share [24].

2.3.1 Throughput Metrics

Throughput metrics measure the speed and volume of code delivery, quantifying how quickly work moves from conception to production. There are two throughput metrics defined as part of the DORA metrics [24]. These are as follows:

1. **Deployment Frequency (DF)**

This metric measures how often an organization successfully releases code to production for its primary application or service. A high deployment frequency is a proxy for small batch sizes, which reduces risk and shortens feedback loops, thus being strongly correlated with high performance [24].

2. **Lead Time for Changes (LTC)**

This is the time elapsed from a code commit being checked into version control to that code successfully running in a production environment, serving end users. A shorter lead time indicates lower risk and faster ability to respond to market demands or correct defects. Elite performers typically achieve a lead time of less than one hour [24].

2.3.2 Stability Metrics

Stability metrics measure the quality of the software delivery process and the resiliency of the system when failures occur. These metrics ensure that gains in speed, i.e. throughput, are not achieved at the expense of reliability. Similarly, there are two stability metrics, these are as follows:

1. **Change Failure Rate (CFR)**

This metric measures the percentage of deployments to production that result in a failure, requiring immediate remediation (e.g., a rollback, hotfix, or patch). A low CFR indicates high quality and confidence in the CI/CD pipeline and testing practices.

2. **Mean Time to Restore (MTTR)**

This metric measures the time it takes to restore service after a service incident or unplanned outage. MTTR is a key indicator of organizational resilience and ability to respond to emergencies. A shorter MTTR suggests mature incident response and operational capabilities.

A key finding of the DORA research is the rejection of the traditional belief that speed must be traded for stability. The data consistently demonstrates

that high performing teams excel across all four metrics simultaneously, achieving both high throughput and high stability [26].

The yearly “State of DevOps” report is composed from a cluster analysis performed on these four key metrics to maintain consistency across reports published each year. The research program strictly gathers clusters of data as they emerge in the absence of pre-defined levels. This ensures the report maintains a holistic overview across all respondents participating in the year [27].

Based on the findings from the reports, efficient release management has emerged as a vital determinant of organizational performance. Industry data from the DORA 2024 State of DevOps report reveals that elite performing companies deploy new code 182 times more often and achieve a mean time to recovery that is 2293 times faster than that of their low performing peers [27].

Increased deployment frequency enhances organizational performance by accelerating the feedback loop on ongoing development, facilitating timely adjustments, and expediting the delivery of new features and bug fixes to end users. Likewise, minimizing product delivery lead times is advantageous since it accelerates insight into the solution under development, supports more agile course corrections, and when defects or outages occur, it enables the rapid, reliable deployment of corrective patches [24].

Given that the core objective of this thesis is the acceleration of the software delivery process, the subsequent analysis will focus exclusively on the throughput metrics, i.e. DF and LTC. The success of the automated governance artifact will be quantified entirely by its measurable impact on increasing DF and decreasing LTC. While the Stability Metrics (CFR and MTTR) are vital for holistic performance, they will serve as essential contextual monitoring metrics to confirm that speed gains were not achieved through the introduction of significant, unacceptable instability, but will not be the primary variables of this research’s optimization.

2.4 Architectural Evolution

Software architecture fundamentally dictates the speed and reliability with which an organization can deliver value. This section reviews the transition from traditional monolithic architectures, which constrain the velocity of releases, to modern decomposed architectures, which enable the independent deployment necessary for high throughput.

2.4.1 Monolithic Release Architecture

A monolithic application is constructed as a single logical executable where all primary components, including the client side user interface, database interactions, and server side domain logic, are tightly coupled and deployed

together. This architecture is a natural starting point for many systems, allowing for easy division into classes and functions within a single process and simple testing on a developer's machine [28].

However, as applications scale and teams grow, this unified structure imposes significant constraints on the release process, directly conflicting with the goal of achieving high throughput as measured by the DORA metrics. There are several consequences of this approach.

Firstly, the defining constraint of the monolith is that any change, even to a small part of the application, requires the entire monolith to be rebuilt and deployed [28]. This necessity for a single, tightly coordinated release event prevents the rapid flow of small changes through the deployment pipeline [1]. The time elapsed from a code commit to production is artificially extended because work must wait for this synchronized release train, thereby increasing the LTC [23].

Secondly, monolithic systems naturally suffer from poor encapsulation over time, making it difficult to maintain clear boundaries between logical components. This tight coupling between logically independent structures means that changes intended for one part of the application often have unintended ripple effects across the whole system. This lack of clear separation prevents the efficient collaboration necessary for large teams [1].

Finally, a single codebase and deployment process often create centralized dependencies. As Newman in 2015 notes, if a single change breaks the shared build, no other changes can proceed, and trying to manage multiple teams sharing this giant build raises the question of who is in charge. Organizations using this approach often fall back to just deploying everything together, creating bottlenecks that throttle the independent work of development teams [29].

In summary, the monolithic architecture, while simple for small projects, becomes increasingly expensive and increased friction as systems evolve. It fundamentally opposes the principles of continuous flow and independent deployment, necessitating the adoption of decomposed patterns to achieve modern velocity targets.

2.4.2 Decomposed Micro-frontend Architecture

To overcome the release constraints of the monolithic approach, organizations transition to decomposed architectures, which began with the micro-service architectural style. Microservices are defined as an approach to developing a single application as a suite of small services, each running in its own process and communicating via lightweight mechanisms, often a Hypertext Transfer Protocol (HTTP) resource API [28]. Critically, these services are built around business capabilities and are independently deployable by fully automated deployment machinery [28]. The principle of a micro-frontend is

the necessary extension of this style to the user interface, aiming to achieve the same independence at the presentation layer.

The theoretical justification for this decomposed structure, and the reason it enables higher throughput, is rooted in the following core benefits:

1. **Decoupled and Scalable Development**

The decomposed architecture provides significant advantages in managing complexity and scaling teams. Microservices align with the Single Responsibility Principle (SRP) which is the focus of service boundaries on specific business capabilities. This ensures that related code is consistently grouped, preventing the common monolithic problem where functionality becomes spread across the codebase [29].

Furthermore, decomposed architectures are essential when a codebase grows too large for a single team to manage effectively [1]. By dividing the problem into smaller, more expressive chunks, microservices allow organizations to align their architecture with their teams, enabling small, highly productive teams to manage their codebases autonomously [1, 30].

This structure also accounts for the fact that components often represent differences in the rates of change, or have different lifecycles, allowing development teams to manage these cycles independently and facilitating faster evolution [1].

2. **Enhanced Agility and Faster Delivery**

The paramount benefit of decomposition is the ability to deploy independently, which directly addresses the lengthy change cycles of the monolith. By separating the codebase, teams eliminate centralized friction and reduce the need for monolithic build artifacts [1].

With microservices, a change to a single service can be deployed independently of the rest of the system [29], allowing code to be deployed faster. This supports the goal of CD, which requires the application to always be in a releasable state, achieved by making all changes incrementally as a series of smaller, releasable changes [1].

This practice of releasing small batches is solely enabled by independent deployment. The final benefit to agility is the ability to optimize for replaceability. Individual services are small, lowering the cost and risk of replacement, which makes teams comfortable with rewriting or removing services entirely when required, as the small codebase size mitigates both emotional attachment and replacement risk [29].

3. **Technology, Resilience, and Scaling**

Beyond development velocity, decomposition provides benefits in system qualities. Technology heterogeneity is enabled by having multiple, collaborating services, allowing teams to choose different technology stacks, for instance, a graph-oriented database for user interactions and a document-oriented store for posts. This capacity to quickly adopt new technologies provides a competitive advantage [29].

In terms of resilience, service boundaries isolate failures in the system. This means if one service fails, the failure does not cascade, allowing the rest of the system to operate, which contrasts sharply with a monolithic service where a single failure stops the entire system [29].

Finally, decomposition supports targeted scaling. Instead of scaling the entire monolithic application to satisfy a constraint in one component, only those components that require greater resources are scaled. This is more cost effective and provides superior performance by allowing other parts of the system to run on smaller, less powerful hardware [29].

4. **Independent Deployments and Pipelines**

The implementation of a decomposed architecture necessitates a corresponding shift in the build and release process. To manage the overhead of multiple services, the system must be split into several different deployment pipelines for parts that have a different lifecycle or are functionally separate [1].

This dedicated pipeline approach is required to assert the viability of each change for release. The build for each component follows the same pattern as a whole system, including compiling the code, assembling binaries, and running unit and acceptance tests [1].

This process, by ensuring the viability of each component, ensures the application is always in a releasable state, thereby making the decomposed micro-frontend architecture the critical enabler for high throughput software delivery.

2.5 **Comparative Case Studies**

The preceding sections established the theoretical necessity of the DevOps culture and the decomposed micro-frontend architecture for modern software organizations. This section now provides the empirical validation for these principles through comparative case studies, demonstrating that organizations adopting streamlined delivery systems achieve superior business

and operational outcomes. This comparative analysis is essential for establishing the definitive performance benchmarks that illuminate the inefficiency of DIP’s current process and justify the proposed intervention.

2.5.1 Benchmark Quantitative Data

The most compelling comparative studies are derived from the research conducted by Nicole Forsgren, Jez Humble, and Gene Kim [24], as part of the DORA program which analyzed thousands of organizations over several years. As discussed earlier, this research empirically categorized organizations based on the four key DORA metrics (DF, LTC, CFR and MTTR) into four distinct performance tiers, i.e., low, medium, high, and elite [24].

The comparative analysis across these tiers reveals a performance gap so substantial that it directly mandates architectural and process interventions. Elite performers, having mastered the streamlined delivery pipeline, demonstrate capabilities that eclipse their counterparts by orders of magnitude:

- **DF:** High and elite performing teams can deploy multiple times per day, often on demand, contrasting sharply with low performers who deploy only once per month or less. This ability to deploy in small batches is directly enabled by the ability for independent deployments of decomposed systems and automated pipelines [24, 27].
- **LTC:** According to the State of DevOps report from 2024, elite performers achieve an LTC of less than one day, compared to low performers, whose LTC is between one month and six months. There are 127 orders of difference in speed between elite and low performers, which demonstrates the dramatic friction caused by manual processes and monolithic architectures [27].

Table 1. Benchmark DORA metrics from State of DevOps 2024 [27].

Performance level	DF	LTC	CFR	MTTR
Elite	Several per week	< 1 day	5%	< 1 hour
High	1 to 7 per week	1 to 7 days	20%	< 1 day
Medium	Once a week to once a month	7 to 30 days	10%	< 1 day
Low	Once per month to once every 6 months	30 to 180 days	40%	1 week to 1 month

Crucially, the DORA research established that this high throughput is not achieved at the expense of stability. Instead, according to the 2024 State of DevOps report shown in Table 1, high performers achieve higher stability simultaneously, with a CFR 8 times lower and a MTTR of about 2300 times faster than low performers [27].

This finding refutes the legacy belief that organizations must choose between speed and stability, providing the quantitative foundation for the thesis goal, which is to increase throughput without compromising quality.

2.5.2 Procedural Comparisons

To understand how the massive performance differential observed in the DORA studies is achieved, a procedural comparison of release models is necessary. The foundational work on CD [1] contrasts the two antithetical approaches to deployment.

The traditional model is characterized by a centralized release process with several points of friction, i.e., infrequent and large batches of code, extended stabilization phases, and manual approval gates. This process involved extended stabilization phases, the creation of release branches in version control, and often resulted in cycles of weeks or months between releases. This manual approach created deployment risk because the changes were batched into large, infrequent releases, where failure investigation was complex and time consuming [1].

In sharp contrast, the comparative process mandates a deployment pipeline. This pipeline transforms the release process from an infrequent, vulnerable event into an automated, routine activity. This pipeline serves as the technical mechanism that allows an application to remain in a perpetually releasable state. By automating every step, from commit to production readiness, and enforcing small, incremental changes, the pipeline minimizes human error and reduces risk. This procedural shift is the necessary condition that permits elite performers to achieve LTC of under a day [1].

While the DIP organization possessed the underlying CI and CD pipelines, the thesis intervention's focus on automating governance and establishing real-time visibility is the key mechanism required to translate that technical capability into measurable organizational throughput, directly enabling the quantitative results detailed in the DORA research.

2.5.3 Organizational Alignment

While technical automation is necessary, descriptive case studies confirm that high performance can only be sustained when technical architecture is aligned with organizational structure, which is a socio-technical alignment. The DevOps Handbook details the transitions of organizations like Amazon

and Netflix, which successfully moved from monolithic systems and siloed teams to service oriented architectures [23].

These organizational case studies provide critical context that success is not merely about writing automation scripts but about removing the organizational friction that prevents the flow of value [23]. Key insights confirm that organizations must treat the entire value stream as a single system to be optimized, thereby forming small, autonomous teams capable of managing the full lifecycle of their services. This structural change aligns the organization with the ability to make independent deployments as afforded by microservices discussed earlier, thereby eliminating the functional need for a centralized, manual release authority.

This architectural decision to adopt a decomposed approach allows the organization to overcome the constraints of a single, centralized release authority. The move to independently deployable microservices aligns the architecture with the organization, enabling the technical mechanism of the deployment pipeline to function effectively and yielding the profound performance differences observed by the DORA metrics.

3 Methodology

This chapter details the methodological approach employed to systematically address the research questions outlined in Chapter 1. This study adopts the Action Research (AR) framework to optimize the software release process within the DIP organization.

AR is a pragmatic, iterative methodology designed to solve a real-world problem within a specific organizational setting while contributing to general academic knowledge [30]. This framework is ideally suited for organizational change initiatives as it involves a cyclical process of planning a change, acting on the plan, observing the results of the action, and reflecting on those results to inform the next iteration [31]. This approach provides a robust mechanism for intervening directly in DIP's release process (i.e., introducing new automation and tracking tools) and then empirically measuring the impact of that intervention on performance.

The following sections detail the specific phases of the AR cycle as applied to the DIP environment and define the data sources.

3.1 Research Design

The research design is structured around a single, comprehensive AR cycle that systematically progressed through the fundamental phases of planning, action, observation, and reflection [31]. This deliberate bounding of the study allows for a deep, empirical evaluation of the full automation artifact's impact within the research timeline. The cycle phases were defined as follows:

1. **Planning:** This phase (see Section 4.1) involved the initial analysis of the existing manual release process, stakeholder consultation with engineering and release management, and the definition of the key intervention goals e.g., eliminating manual ticketing, improving visibility, and enhancing communication. The initial research plan was formulated based on collecting throughout metrics as baseline measures prior to the intervention.
2. **Action:** This critical phase (see Section 4.2) involved the unified development and deployment of the complete automation artifact. This included building and integrating the automated release dashboard for a centralized display, and the asynchronous notification component. This combined intervention ensured all key deployment events triggered actionable and asynchronous updates to relevant stakeholder channels, thereby addressing both visibility and communication gaps simultaneously.

3. **Observation:** Comprehensive data collection was performed (see Chapter 5) following the deployment of the complete automation solution. This included gathering the final quantitative data and collecting qualitative data through semi-structured interviews with key personnel to understand the practical and cultural impact of the solution.
4. **Reflection:** The final analysis phase (see Chapter 6) synthesized all collected quantitative and qualitative data. This reflection stage evaluated the overall effectiveness of the intervention against the research questions and allowed for the formulation of generalized academic conclusions regarding the automation of governance in decomposed architectures.

3.2 Data Collection

The execution of this methodology relied on two approaches to data collection, which were quantitative and qualitative.

3.2.1 Quantitative Data

Quantitative analysis utilized the throughput indicators from the DORA metrics, i.e., DF and LTC to provide objective measures of release velocity before and after intervention.

The quantitative data was collected by measuring the entire lifecycle of code changes, from the moment a developer opened a Pull Request (PR) until the code was successfully deployed to production. To achieve the required level of detail and accuracy, a custom data aggregation system was implemented.

The backend system responsible for consolidating release metrics combined static PR metadata from Bitbucket, which included PR ID, title and merge commit hash, with the dynamic PR lifecycle events sent from the CI pipelines, which included timestamps for the most significant milestones in the continuous delivery process, e.g., deployment times, PR creation and merge times.

The aggregation was managed by an AWS Lambda function fronted by a secured webhook URL. Each pipeline event issued an HTTP POST to this webhook, allowing the lambda function to update a central DynamoDB database. This DynamoDB table used the merge commit hash as its primary key, accumulating a chronological history of the release process for every change, ensuring data integrity and freshness without manual intervention.

The consolidated dataset residing in DynamoDB was accessed and processed within a Jupyter Notebook environment. The analysis was performed using the Pandas library for data manipulation and structuring. The resulting

data frames were then passed to the Seaborn library to generate statistical visualizations, enabling the identification of trends and correlations.

The analysis focused on comparing the performance of the DORA throughput metrics, DF and LTC, across the baseline and the validation phase.

DF was calculated as the average number of successful production deployments per week, derived from the total count of production deployment events in the frontend repository over the specified observation period, whether baseline or validation. This metric provides a simple and direct measure of the overall organizational cadence.

LTC was calculated as the sum of two distinct time segments, designed to isolate the organizational bottleneck from development work time. The median value for each segment was used to mitigate the impact of statistical outliers caused by infrequent events (e.g., abandoned or significantly delayed releases), thus providing a representative measure of the typical team throughput.

To accurately pinpoint the source of friction, this metric is broken down into two parts, where the total LTC is the sum of these time segments:

1. **PR Approval to Merge (L1):** The median time elapsed from a PR receiving final approval, signifying it is code readiness, until it is merged into the main branch.
2. **PR Merge to Deployment (L2):** The median time required from the moment the code is merged until it is successfully deployed to production. This segment encompasses all subsequent technical processing (CI/CD pipeline execution) and manual deployment waiting time.

For visualization, the data was passed to the Seaborn library to generate box plots. These plots provided a compact visual summary of the data distribution, clearly illustrating the median, lower quartile, and upper quartile for both the baseline and validation periods to empirically demonstrate the effect of the intervention.

3.2.2 Qualitative Data

Qualitative data was collected through semi-structured interviews with key stakeholders across development, operations, and product roles within the organization. This data is necessary to illuminate the lived experience of the teams that drive each deployment.

Accordingly, these interviews and contextual inquiries were conducted with the participants described in Table 2.

Table 2. Qualitative analysis participant title and roles in release process.

Title	Role in release process
Feature team developer	Initiates PRs, drafts release tickets, and monitors pipelines
Quality Assurance (QA) Engineer	Validates build stability and confirms release readiness
Scrum Master	Facilitates day-to-day coordination, surfaces risks, and maintains visibility across teams
Program Manager	Authorizes formal release milestones and communicates status to executive leadership
Release Manager	Co-owns release documentation and executes production rollouts
Product Owner	Ensures alignment between release content and customer-facing objectives

The qualitative data gathered through these semi-structured interviews and contextual inquiries, underwent a rigorous, systematic analysis aimed at extracting core conceptual themes directly from the dataset. This approach ensured that the findings and derived themes emerged inductively from the participants' insights and experiences.

The analysis proceeded through three distinct, iterative phases:

1. **Labeling:** The first phase involved familiarization and labeling of the raw data. The comprehensive interview notes were subjected to multiple readings, and every significant statement, observation, pain point, or suggestion was extracted and assigned a descriptive code or label.
2. **Clustering:** Following this labeling process, the data was clustered and categorized. Related codes were manually grouped together into initial categories based on conceptual overlap. For example, labels such as "lack of visibility," "difficult to find status," and "manual communication" were grouped into a higher level category related to "information friction."
3. **Thematic categorization:** Finally, these categories were refined, validated against the original data transcripts, and consolidated into overarching themes. These themes represent the most prevalent and

relevant patterns of experience and organizational friction identified across all participants.

The final thematic output directly informed the design of the technical solution, ensuring the proposed changes addressed the documented needs of the users and organizational bottlenecks.

4 Results

This chapter presents the empirical results of the AR cycle's observation phase, detailing the data collected before and after the deployment of the automated artifacts. The chapter follows the systematic progression of the AR methodology by following the case background and establishing the baseline performance (see Section 4.1) then documents the technical intervention (see Section 4.2).

The analysis focuses on the quantitative changes in the DORA throughput Metrics (LTC and DF) and triangulates these findings with qualitative data gathered from key stakeholders.

4.1 Current State Analysis

In this section we review the evolution of DIP's frontend architecture and release practices, setting the stage for the enhancements described later.

The rapid growth and competitive pressures in the AI-driven market research space, created an imperative for DIP to adopt a more agile, reliable software delivery. To understand the origins of the current workflow challenges, we begin with the legacy system architecture and trace the key shifts that have shaped the current hybrid release pipeline.

4.1.1 Frontend Architecture Evolution

Initially, the entire frontend was contained within one monolithic application which produced a new build artifact as a result of any updates to a part of the User Interface (UI). This approach soon revealed significant drawbacks as the development team expanded. Every feature change, bug fix or styling update triggered a full re-build and re-deployment, leading to longer build times, an increased the risk of merge conflicts and a heavily orchestrated release schedule. Furthermore, to assure product quality, each bundled release required an explicit approval for deployment to the Release Candidate (RC) environment.

This RC environment is a staging instance that closely mirrors the production system in configuration and data. It serves as the final validation layer where fully integrated builds undergo final testing before any code is promoted to live production.

Following the deployment to the RC environment, developers were required to request an additional approval for deployment to the live production environment. This approval process ensured all code has been thoroughly manually tested before they reach the hands of the customers. In addition, it ensured that deployments were always shipped to production in

the correct sequence, i.e. the build artifact created first should be promoted to production first.

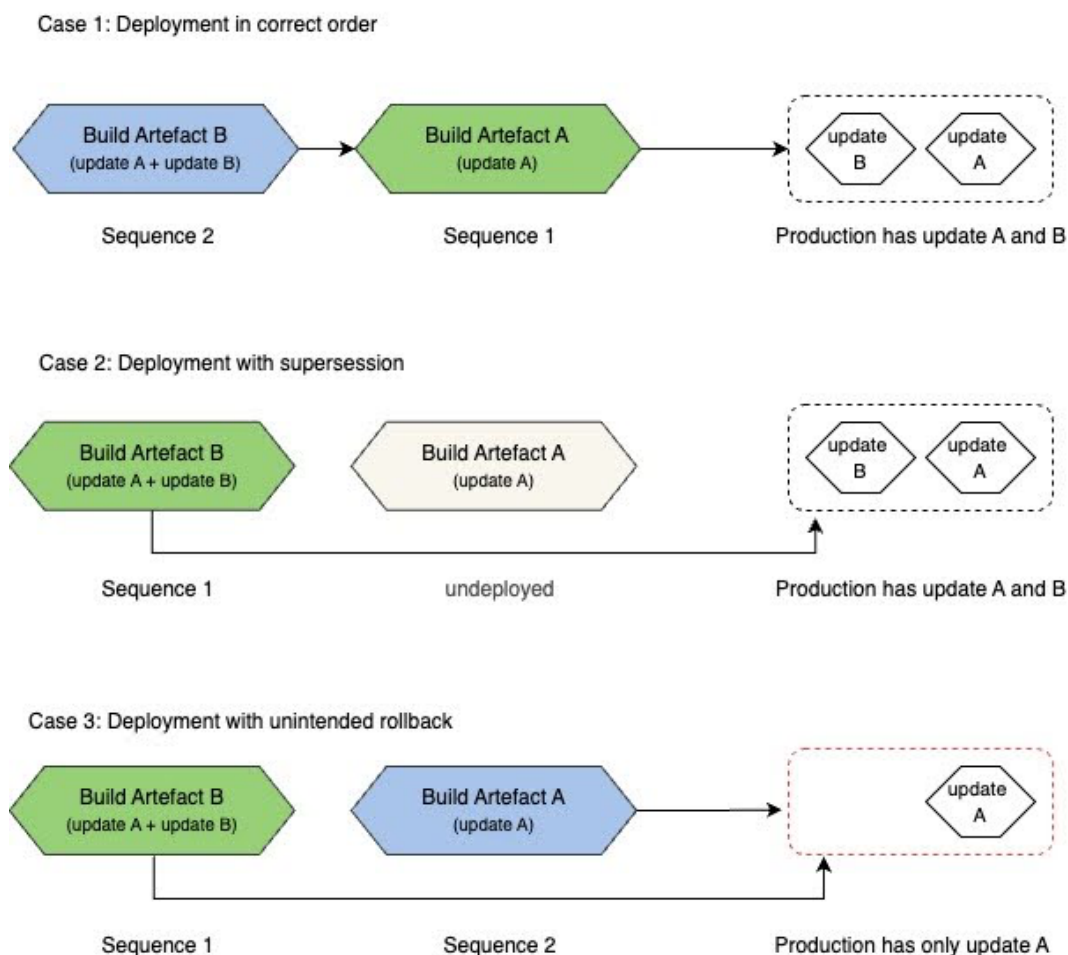


Figure 1: Flow diagram depicting the consequences of not maintaining deployment order.

Ensuring that production builds are deployed in their intended sequence is essential. Misordering artifacts can lead to unintended rollbacks. As depicted in Figure 1, if a newer build is released before an older one, subsequently deploying that older build will omit the changes introduced by the prior release. In effect the system will cause an unintentional rollback and may trigger service disruptions or other production incidents.

To manage the release process and necessary approvals, DIP adopted the use of a project management ticket system provided by Atlassian called Jira to track the status and progress of releases. Each ticket served as the primary mechanism for tracking release status deployment frequency, rollback events and approval stage. This system required each developer to create a ticket on the Jira system called a “release ticket” with all the necessary information about the release and manually progress it through its release lifecycle stages.

While the frontend monolithic application suited the needs for several smaller teams, it soon began posing challenges as the development organization grew to span multiple teams across multiple time zones. Since all deployments to the frontend application shared the same deployment queue, a growing number of contributors resulted in the deployment queue becoming long and difficult to manage.

Recognising that a single large application had become a release bottleneck, the company embarked on a modularisation effort. The frontend codebase had begun to be decomposed into several micro-applications each focused on a distinct feature area of the user interface.

Each micro-application unit encapsulates a discrete segment of the user interface or functional capability and consequently is able to maintain an independent deployment pipeline. This means, when a micro-application update is prepared for release, only its specific build artifact is compiled, packaged, and version labelled. This artifact then replaces its predecessor in the target environment without affecting other components. As a result, the deployment of micro-applications can proceed in isolation, leaving all other micro-application artifacts in production unchanged.

Concurrently, the organization strengthened its quality assurance practices by mandating higher automated test coverage thresholds and delivering targeted training on testing methodologies to development teams. This investment in end-to-end and integration testing fostered confidence that individual micro-applications could be released safely without introducing regressions elsewhere in the user interface. Accordingly, any micro-application that met a predefined benchmark for comprehensive test coverage was granted permission to deploy directly to production, bypassing the need for centralized Release Manager approval.

Despite the ongoing migration to a micro-application architecture, roughly half of the frontend codebase still resides in the original monolith. Consequently, any changes to this remaining monolithic portion must continue to follow the established approval workflow. Furthermore, micro-applications that have yet to meet the organisation's automated testing standards remain subject to centralized release manager approval. Releases to these micro-applications will be referred to as those which require "RC approval". DIP's release workflow has consequently evolved into a hybrid model that retains some of the original coordination overhead and limits the full potential of the micro-application approach.

The following section will provide a detailed account of the current state of that workflow, examining each step from code merge through production rollout. The key actors, artifacts, and decision points involved will be mapped, in order to establish a baseline against which the proposed refinements will later be evaluated.

4.1.2 Software Delivery Evolution

To contextualize the scale and urgency of these processes, it is useful to consider recent deployment volumes. Between January 1 and October 7, 2024, DIP deployed 4010 total releases, of which approximately 1400, which is about 35% of release, were frontend updates. Notably, 42% of those frontend deployments were delivered entirely via the new micro-application pipeline. Figure 2 illustrates both the throughput demands placed on the current system and the potential leverage offered by targeted workflow improvements.

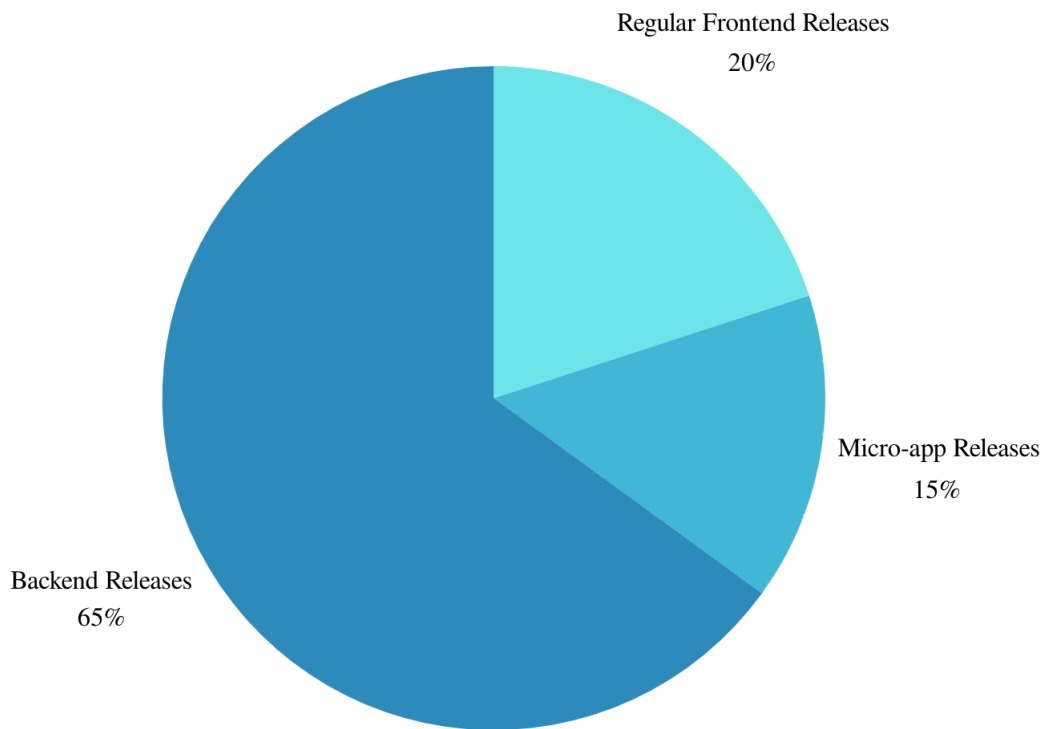


Figure 2: Distribution of releases between 1st Jan 2024 - 7th Oct 2024.

As noted in the previous section, DIP’s frontend release workflow now combines legacy gated deployments with the newer non-gated micro-application pipelines. Figure 3 illustrates the granular steps of the gated release process pathway, from ticket creation through ticket closure.

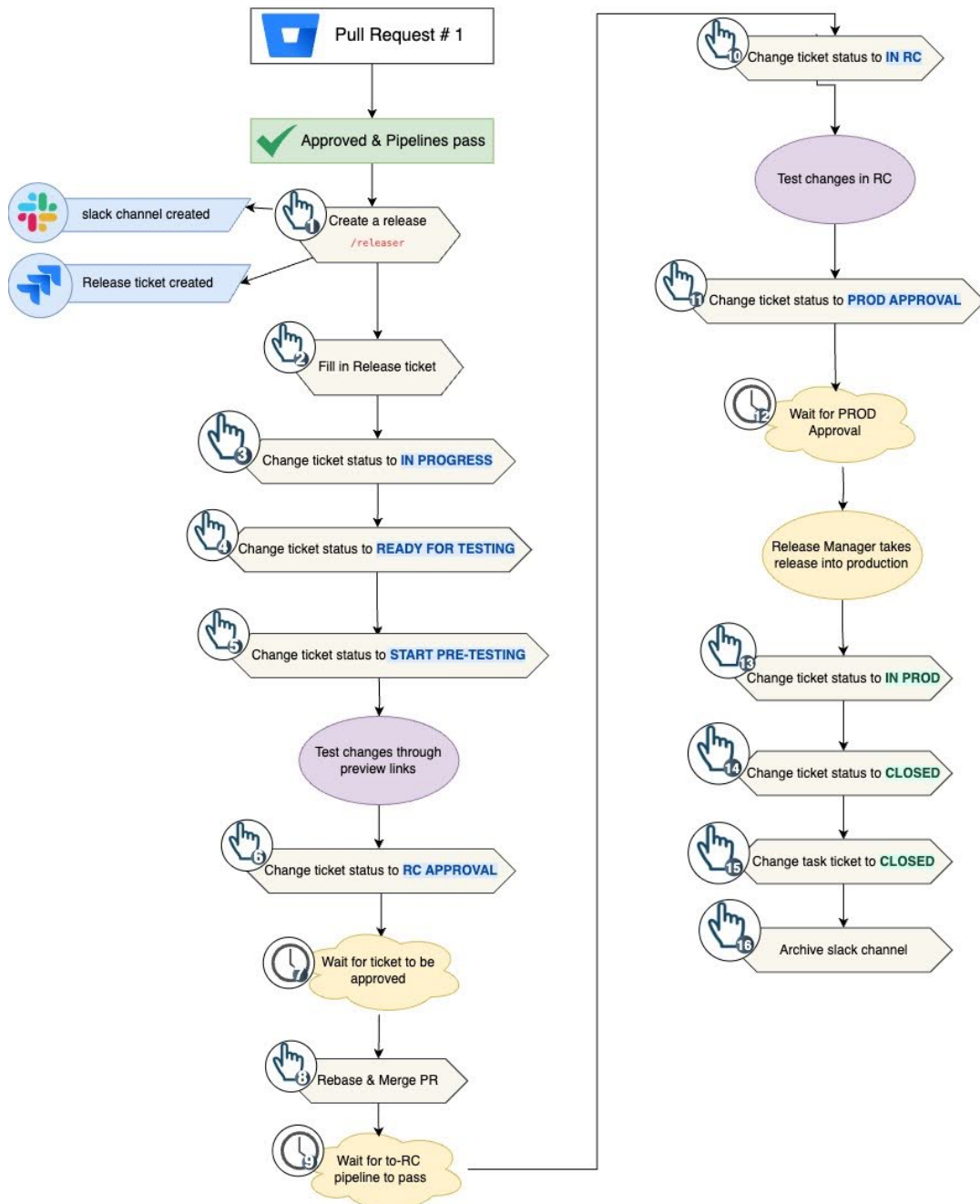


Figure 3: Illustration of the 16 step gated frontend release process.

As illustrated in Figure 3, once a PR is approved, step 1 mandates the developer to create a dedicated Slack channel and open a Jira release ticket to track the deployment. Within that ticket they document the full scope of the release, i.e. code changes, feature flags, configuration updates, and any dependencies (step 2). This setup is essential for enhancing visibility across distributed teams, however the manual labour with drafting release notes,

tagging stakeholders, and linking related task tickets introduces notable overhead.

Once the ticket is populated, the developer advances its status through a series of preparatory stages, i.e. “In Progress” (step 3), “Ready for Testing” (step 4), and “Start Pre-testing” (step 5). Each status change generates notifications to the QA engineers and relevant stakeholders following the slack channel.

After validations via preview links, the developer marks the ticket with the status of “RC Approval” (step 6) and awaits authorization to merge (step 7). This approval can take a few hours to a few days and it relies on the release queue being clear. Upon receiving a sign-off from the Release Manager, the PR is rebased and merged into the release branch (step 8). The developer then monitors the RC pipelines until they complete successfully (step 9) and updates the ticket to the “In RC” status (step 10). The developer has to carefully monitor the release pipelines to ensure the pipeline has completed successfully. Any failures would require re-running the pipeline or alerting the relevant teams to investigate the cause of a failure to release to the RC environment.

Once a successful RC build has been generated, the developer is required to transition the ticket status to “Prod Approval” (step 11) which signals the Release Manager to review and promote the candidate to production (step 12). The Release Manager promotes a release to production only when the release is the first in the release queue to be deployed. This ensures deployments are not released in an incorrect order.

Once production deployment is confirmed, the developer updates the ticket status to “In Prod” (step 13). This phase carries high operational risk, as misconfigurations or overlooked dependencies can lead to customer facing incidents.

Finally, the developer closes the release ticket (step 14), closes the associated task ticket (step 15), and archives the Slack channel created for the release (step 16). Although these administrative steps require minimal effort, they are often deferred, resulting in discrepancies between actual and recorded release states in Jira.

Analysis of historical data reveals that the median elapsed time from PR merge (step 8) through final ticket closure and Slack channel archival (step 16) is 19 hours, underscoring the cumulative impact of manual hand-offs and environment coordination on release velocity.

As outlined earlier, the micro-application pipeline represents DIP’s modernized release path. Figure 4 illustrates its streamlined sequence. The following narrative describes each phase with reference to the corresponding step numbers.

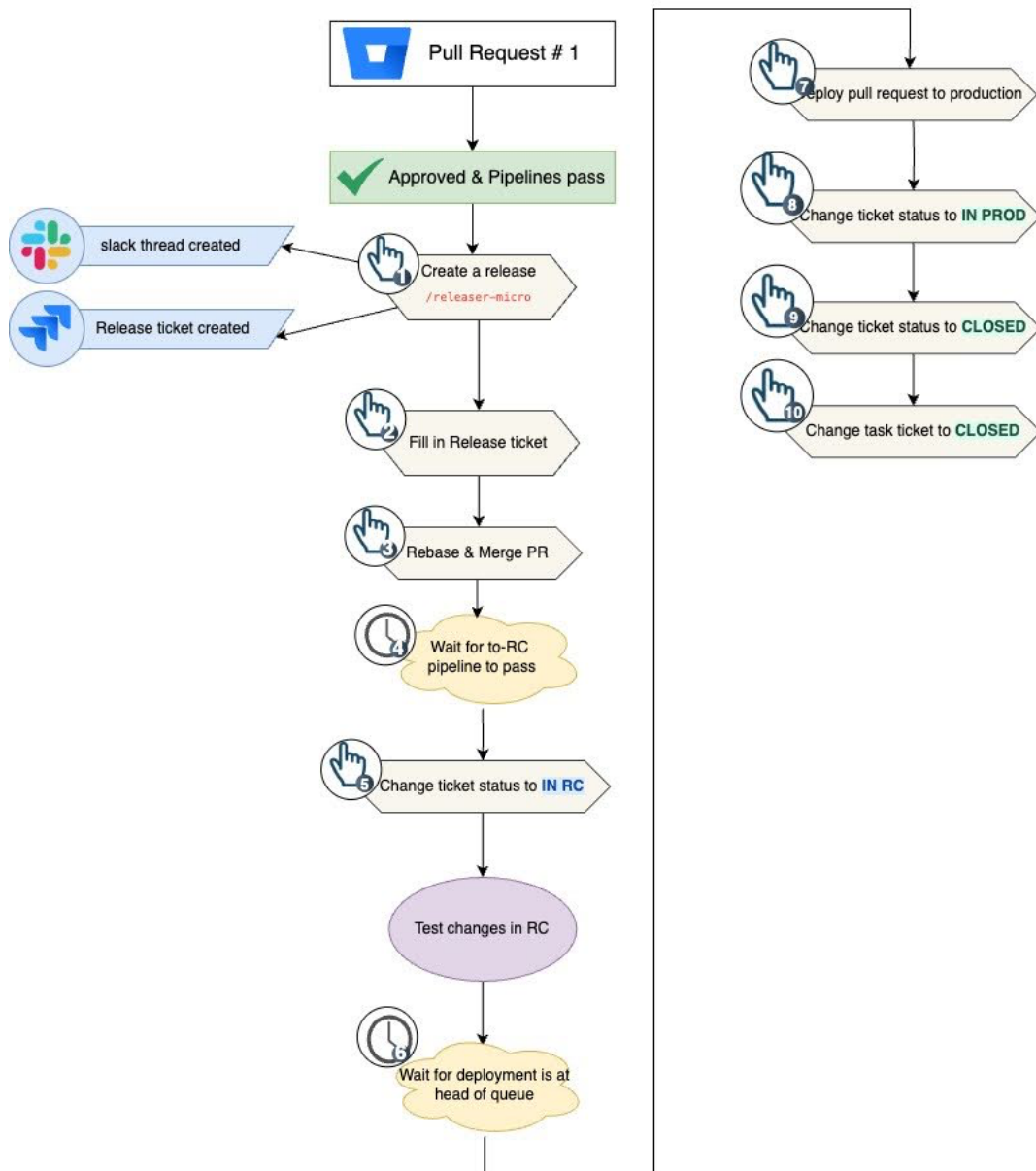


Figure 4: Illustration of the 10 step non-gated frontend release process.

As visualized in Figure 4, upon PR approval, the developer is required to open a thread a dedicated Slack channel and continue the practice of creating a Jira release ticket to track the deployment (step 1). The ticket is then populated with all necessary release metadata, i.e. feature descriptions, configuration parameters, and dependency notes (step 2). This initial phase parallels the gated release process.

With the ticket prepared, the developer rebases and merges the PR directly into the micro-app’s release branch, without requiring an approval from the Program Manager (step 3). The merged code automatically triggers the RC pipelines, which the developer monitors until they complete

successfully (step 4). Upon passing all checks, the ticket status is updated to “In RC” (step 5), indicating readiness for production deployment.

After verifying the RC build in its staging context, the developer waits for the release to reach the head of the deployment queue (step 6). Once at the front, the developer executes the production rollout (step 7) and then marks the Jira ticket “In Prod” (step 8).

Finally, the developer closes the release ticket (step 9) and its associated task ticket (step 10). Unlike the legacy workflow, no separate archival of a Slack channel is necessary since only a thread is created within a channel.

Empirical analysis shows that the median elapsed time from pull-request merge (step 3) through final ticket closure (step 10) is five hours, reflecting the efficiency gains achieved by eliminating several manual stages and consolidating environment management.

Occasionally, micro-application deployments occur out of sequence due to human error, prompting Scrum Masters or Program Managers to intervene by coordinating rollbacks and reordering releases. Such incidents reveal opportunities to eliminate procedural overhead and further streamline the release pipeline.

Having mapped the current release workflows for both the legacy gated and the non-gated micro-application pipelines, we now turn to analysing the root causes of velocity constraints within these processes utilizing the quantitative and qualitative data collection methods for this research.

4.1.3 Baseline Quantitative Analysis

The quantitative analysis for the baseline state will measure the throughput metrics, which is the DF and the LTC.

Deployment Frequency (DF)

There are two types of frontend releases, the gated release that requires RC approval and micro-application release which does not require RC approval. To quantify these workflows over the six month window from 24 April 2024 to 24 October 2024, release ticket data from Jira is visualized in Table 3.

Table 3. Total and average baseline number of release tickets per release type.

Jira Release tickets	Total releases	Average releases per week
All	1124	46
RC approval required	589	24
RC approval not required	535	22

These figures were obtained using JQL filters scoped to the DIP project, selecting tickets of type “Release (simple)” or “Release-Microapp” with statuses “In PROD” or “Closed” and production dates between December 21 2023 and October 20 2024.

Another metric to track the number of releases is to use the information retrieved directly from Bitbucket pipelines, which is the Version Control System (VCS) used in DIP. Each PR would be regarded as one release. However, each PR merge does not warrant a release to production, since changes which do not modify production artifacts do not need to be released to the production environment. Examples of these changes include changes to test files, changes to deployment scripts, changes to Bitbucket pipelines, and so on.

Therefore, it is important to differentiate between PRs that were merged and those that were deployed to production. Metrics retrieved from Bitbucket PRs over a six month window from 24 April 2024 to 20 October 2024 is presented in Table 4.

Table 4. Baseline DF per release type.

Metric	Overall	RC approval required	No RC approval required	Overall average per week
PR merges	1855	800	1055	71
PR deployments	1230	624	606	47

A noticeable differential exists between the total number of PR merges and the count of PRs deployed to production over the sampled time period. This difference in throughput is visually presented for comparison in Figure 5.

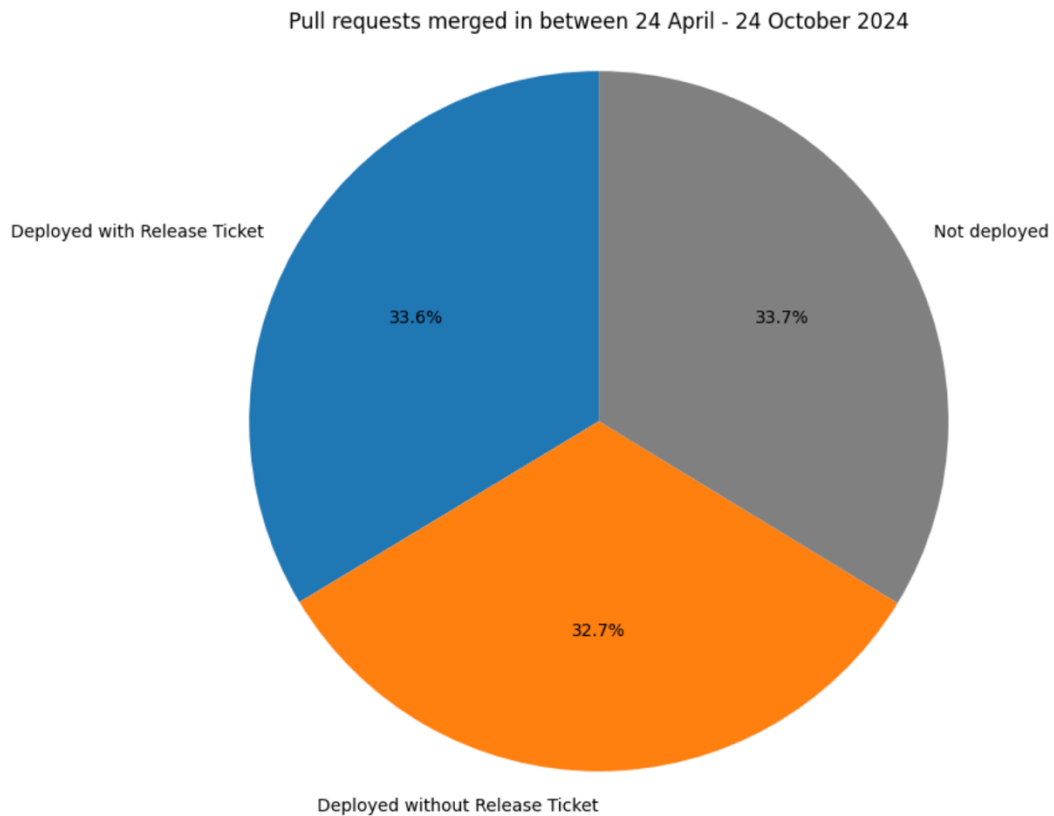


Figure 5: Segmentation of deployed PR merges from 24 April to 24 October 2024.

The data indicates that, of all frontend PRs merged during the six month interval between 24 April and 24 October 2024, approximately one third resulted in no production deployments. These merges typically involve ancillary changes, such as updates to test suites, build scripts, pipeline definitions, or configuration files that do not require formal release procedures.

Moreover, nearly 6% of PRs that reached production were not associated with a dedicated Jira release ticket. This discrepancy suggests that developers either aggregated multiple pull-request merges under a single release ticket or omitted the creation of release tickets entirely. Consequently, analyses based solely on Jira ticket records will undercount a substantial portion of repository activity and deployed changes.

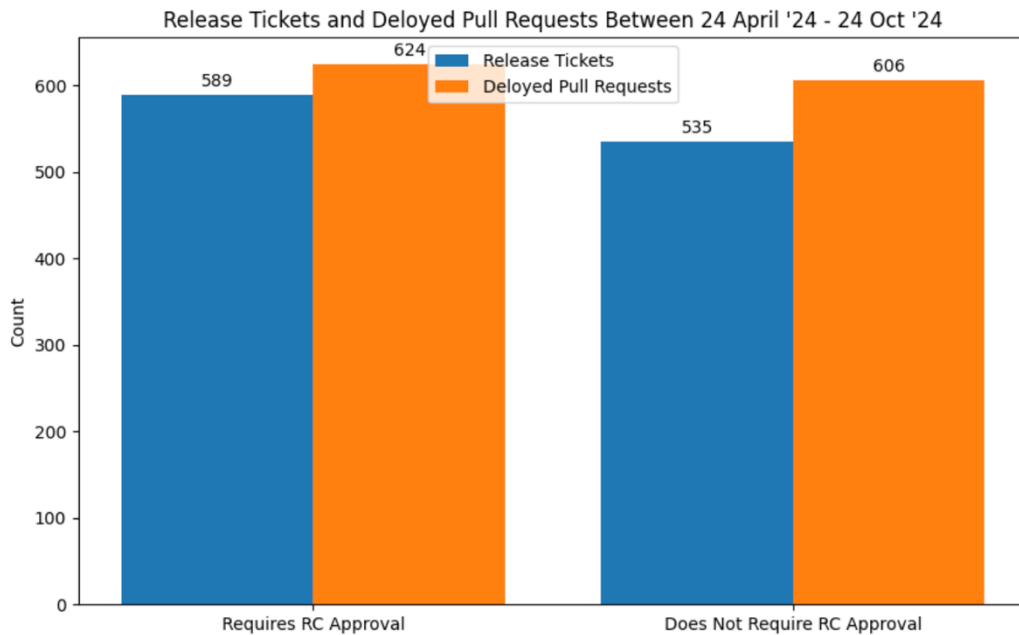


Figure 6: Bar chart comparing deployment frequency for each release process between 24 April 2024 and 24 October 2024.

When comparing deployment counts drawn from Jira against those recorded in Bitbucket as depicted in Figure 6, we observe a modest 6% variance for releases that require RC approval, indicating reasonable alignment between ticket based tracking and actual production deployments. However, for independent micro-application deployments, the divergence grows to approximately 12%. This larger gap implies that developers are more inclined either to consolidate multiple micro-application changes within a single Jira ticket or to omit ticket creation for these releases altogether.

All in all, for the purposes of calculating true deployment cadence, we rely exclusively on Bitbucket’s production deployment logs as the authoritative source. Therefore, the DF for the frontend repository is 47 deployments per week.

Lead Time for Changes (LTC)

This section evaluates the lead time from code readiness to production deployment, with particular attention to the portion of that interval attributable to manual deployment triggers. As detailed in the Methodology (see Section 3.2.1), LTC measures the total elapsed time from code readiness to production deployment. This evaluation is critical for quantifying the latency introduced by manual governance and elucidating the potential gains from further automation by isolating waiting times.

Consistent with the methodological design, the total LTC is segmented into two sequential intervals: L1 (PR Approval to Merge) and L2 (PR Merge to Deployment). The total LTC is the sum of the median values of these two segments.

The distribution of the PR approval to merge segment (L1) is depicted in Figure 7. The box plots compare data across all PRs, distinguishing between those that required RC approval and those that did not, based on data gathered during the six month observation window between 24 April and 24 October 2024.

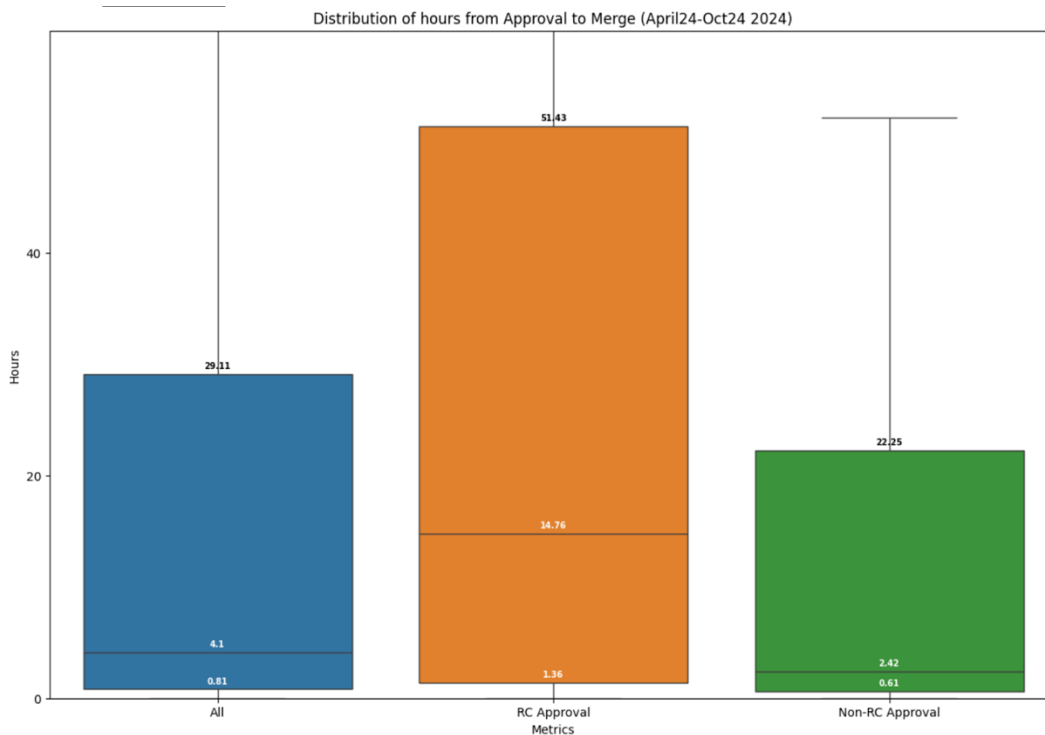


Figure 7: Baseline distribution of hours between PR merge and approval.

As illustrated in Figure 7, PRs that required RC approval experienced a distinct waiting period before merge, while PRs which did not require RC approval were typically merged promptly upon PR approval. This shows that PRs that required approval may have stalled in the queue awaiting formal approval.

The data from Figure 7 indicates that PRs that required RC approval exhibited a broad spread, with a 75th percentile marginally above 50 hours, whereas PRs that did not require RC approval reached a 75th percentile of 22 hours and a median of only 2.4 hours.

Thus, RC gating imposed a 6 fold increase in PR approval to PR merge waiting time. Overall, the first segment of the LTC equation is established at 4.1 hours median.

The baseline performance of the PR merge to production segment (L2), which includes automated processing time and the production deployment wait time, over the same six month time period is illustrated in Figure 8.

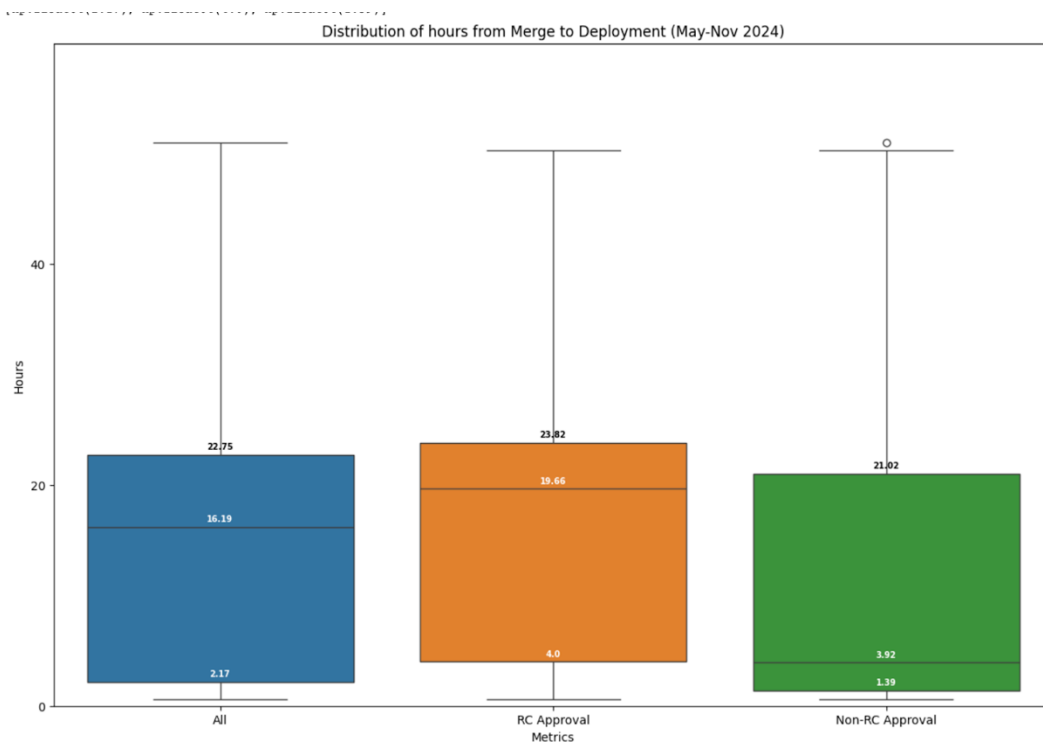


Figure 8: Baseline distribution of time between PR merge and deployment to production.

Over the baseline observation period, the overall median time from merge to production was 16.1 hours. Deployments requiring RC approval took a median of 19.7 hours, which stands in stark contrast to the 3.9 hours required for deployments without RC approval. This significant performance differential suggests that the friction caused by the manual RC process frequently defers deployment to the subsequent business day, while PRs which do not require RC approval are deployed within the same working day.

To determine the proportion of L2 time consumed by automated processes versus human waiting time, the segment is further broken down into pipeline build time (L2a) and production deployment wait time (L2b). To begin with, the time elapsed between PR merge and the code becoming ready for the final manual trigger step is analyzed in Figure 9.

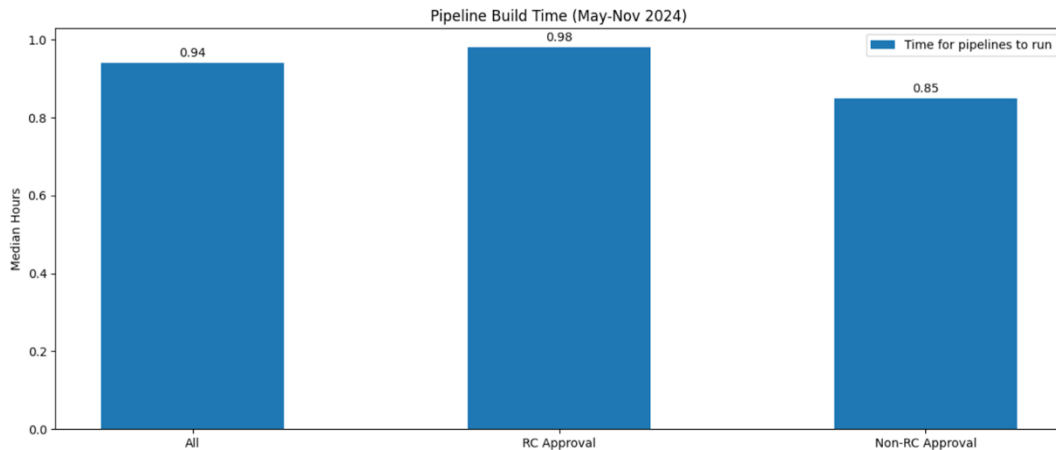


Figure 9: Baseline comparison of pipeline run time across release types.

As Figure 9 illustrates, the median time required for CI/CD pipelines to run is consistent across release types, resting at approximately 1 hour, i.e. 0.98 hours for those that required RC approval and 0.85 hours for the rest. This indicates that the automated pipeline infrastructure does not represent a significant bottleneck. Consequently, a major proportion of the total time within the L2 segment must be attributed to the waiting period preceding the manual deployment trigger, reported in Figure 10.

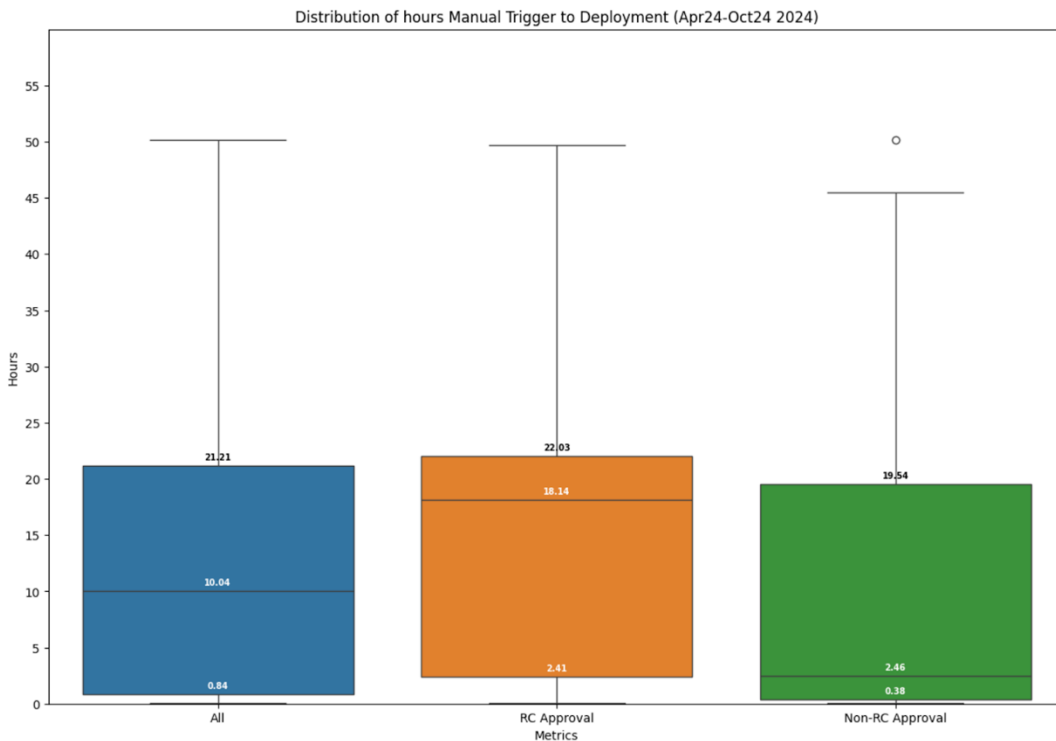


Figure 10: Distribution of time between manual trigger and production deployment across release types.

As reported in Figure 10, a large portion of the 16.1 hours L2 median is attributable to this final wait time for production deployment. The overall median waiting time is 10.0 hours. Deployments requiring RC approval faced a median latency of 18.1 hours, whereas deployments without RC approval waited only 2.5 hours. This pronounced skew toward manual latency suggests that, even after automated build and test pipelines complete, deployments frequently await human intervention.

The Interquartile Range (IQR) for deployments that required RC approval spanned 2.4 to 22 hours, with a median of 18.1 hours. This distribution indicates a bimodal pattern, in which deployments either occur rapidly within the same working day, or are deferred to the next business day, inflating the median.

To contrast, the IQR for deployments that did not require RC approval lied between 0.4 and 19 hours, with a median of only 2.5 hours. This indicates that while a small fraction of these deployments also slipped to the subsequent day, the majority completed within hours of merge.

The overall weighted median LTC for the baseline period is 20.2 hours. The complete comparison of the two operational release paths, utilizing these defined segments, is presented in Table 5.

Table 5. Baseline LTC breakdown per release type.

Metric	Overall (hours)	RC approval required (hours)	No RC approval required (hours)
L1: Approval to Merge	4.10	14.76	2.42
L2: Merge to Deployment	16.10	19.12	2.95
L2b: Production Deployment Wait Time	(10.0)	(18.1)	(2.5)
Total LTC	20.2	34.8	5.4

These findings demonstrate that manual approval and deployment steps constitute the principal bottleneck in DIP’s current release process and validate the thesis premise that further automation of manual triggers and gating mechanisms can substantially reduce total lead time.

While quantitative data illustrates the speed at which updates are coded and shipped into production, qualitative data is necessary to complement the experiences of various stakeholders involved with the release process. The

next section will cover the qualitative data gathered through interviewing several candidates within the company.

4.1.4 Baseline Qualitative Analysis

Qualitative data was collected across several roles within the organization through semi-structured interviews. The responses from these interviews could be consolidated within two themes, the importance of JIRA based release tickets and the challenges associated with the release process.

Importance of Release Tickets

To begin with the importance of these release tickets, the analysis of stakeholder interviews revealed that Jira release tickets served two principal functions within DIP's frontend delivery ecosystem:

- 1. Real-Time Tracking and Management of Release Progress**

Release tickets operate as dynamic coordination artifacts that reflect the status of a given deployment as it advances through the development, RC, and production environments. By centralizing environment-specific statuses and planned deployment dates, these tickets enable multiple roles to synchronize their activities.

For instance, Scrum Masters leverage the ticket queue to prioritize work. They identify critical fixes that must be promoted ahead of less urgent releases and coordinate with developers to adjust deployment order accordingly.

Program Managers, meanwhile, consult release tickets to assess queue availability and preempt potential conflicts among parallel delivery streams.

QA engineers depend on ticket statuses to determine which builds require immediate regression testing, ensuring that high-risk changes receive appropriate validation.

Developers refer to ticket approvals to ascertain when they may safely proceed with merges and deployments, reducing uncertainty around gating criteria.

Finally, Product Owners extract tentative timelines from ticket metadata to communicate expected delivery windows to stakeholders. In this way, release tickets function as a shared “source of truth,” facilitating cross-functional alignment and minimizing ad hoc communication overhead.

2. **Recording Release History**

Beyond their role in live coordination, release tickets constitute an auditable ledger of past deployments, capturing both technical and contextual information that is indispensable for incident resolution, retrospective reporting, and strategic planning.

Each ticket records the nature of included changes (whether new features or bug fixes), the teams responsible for their implementation, the individuals who executed the deployments, and a concise “user/platform value” description that articulates the end-user impact. For releases involving user-interface modifications, the ticket’s metadata triggers updates in a dedicated Slack channel, ensuring that downstream stakeholders remain informed of UX alterations.

QA engineers and Scrum Masters draw upon this historical record when investigating production incidents, tracing the provenance of a regression to its originating release and thereby accelerating root-cause analysis.

Program Managers and Product Owners rely on ticket archives to compile upward-facing reports, such as quarterly “Kick-off” and “Wrap-up” communications, to senior leadership and cross-organizational audiences, substantiating claims of delivery throughput and stability.

Product Operations teams use the same information to draft formal release notes for enterprise customers, ensuring that external documentation aligns precisely with what was deployed.

Challenges of Release Process

Interviews with key stakeholders at DIP reveal 5 overarching categories of friction that impede release velocity. These pain points span procedural overhead, inter-team coordination challenges, and technical reliability issues:

1. **Manual Tracking and Tedious Processes**

Stakeholders consistently report that the current process relies heavily on manual status updates across multiple platforms, i.e. Jira, Confluence, Slack, and even ad-hoc Google Sheets trackers.

Scrum Masters describe spending significant portions of their workday manually advancing ticket statuses, reconciling information across disparate tools, and notifying team members of changes.

Program Managers echo this sentiment, noting that maintaining Gantt charts and status dashboards in Google Sheets for upward reporting to leadership is both error-prone and laborious.

QA Engineers further emphasize that incomplete or outdated ticket information forces them to cross-reference Slack threads or Confluence pages to determine which builds require regression testing.

Developers, too, observe that redundant ticket updates divert time away from coding and code review, while Product Owners struggle to extract reliable delivery timelines from inconsistent ticket metadata.

2. Queue Contention and Release Delays

The release queue for both gated and non-gated micro-applications frequently becomes congested when multiple teams seek to deploy concurrently.

Scrum Masters must negotiate sequence order by personally coordinating with their peers, a process that can stall when teams have equal urgency.

QA engineers raised concerns about critical hotfixes getting delayed if another release is accidentally scheduled ahead of them, necessitating emergency rollbacks or re-booked slots.

From a Program Manager's perspective, this unpredictability complicates planning for backend dependencies and alignment with broader product launches.

In the most acute cases, releases requiring on-call support or cross-team collaboration incur multi-day delays simply because the queue lacks fine-grained prioritization.

3. Inconsistent Usage of Release Tickets

Although Jira release tickets are intended as the single source of truth, stakeholders report wide variation in how thoroughly tickets are completed.

QA engineers note that developers sometimes neglect to update statuses after pipeline failures, making it difficult to discern which releases are genuinely pending versus stalled.

Product Operations teams highlight that fields such as “user/platform value” are often left blank, hampering the generation of coherent release notes for enterprise customers.

This uneven documentation complicates post-release incident investigations, as it obscures the linkage between a deployed change and subsequent production behavior.

4. **Flaky Automated Tests**

Unreliable CI/CD pipelines and intermittent test failures introduce significant latency into the release process.

Program Managers cite flaky end-to-end and integration tests as a recurrent blocker, forcing manual test execution or multiple pipeline re-runs.

QA engineers confirm that such instability degrades confidence in automated regression suites, causing them to revert to manual verification, a time-consuming fallback that undermines the very purpose of continuous delivery.

The cascading effect is that downstream stages in the workflow must await manual sign-off, inflating both lead time and human effort.

5. **Challenges in Triaging Incidents with micro-applications**

While the micro-application architecture was designed to decentralize releases, QA and Product Operations observe that the speed and frequency of these deployments can outpace their ability to monitor and triage incidents.

Because micro-applications bypass RC approval, QA is not systematically alerted to new production releases, leading to blind spots when diagnosing customer-reported issues.

Product Operations recount episodes in which a series of rapid micro-app deployments caused an unforeseen cross-module regression, yet the team responsible for initial remediation lacked sufficient context on the release that triggered the incident.

This gap in oversight increases mean time to recovery and risks customer satisfaction.

Proposed Enhancements

In response to the identified pain points, interview participants suggested a suite of enhancements aimed at reducing manual burden, improving reliability, and consolidating communication channels. These proposals fall into four thematic areas:

1. Automation in Release Process

Stakeholders advocate for automating routine status transitions and metadata population within Jira release tickets.

Scrum Masters envision workflows in which pull-request merges automatically trigger ticket updates to “In RC” or “In Prod,” alleviating the need for manual gating.

QA proposes tighter integration between Bitbucket and Jira so that pull-request metadata, including test results and branch-level comments, feeds directly into ticket fields.

Program Managers further recommend event-driven automation that flags ticket owners through Slack when a release is ready for their action or when a downstream dependency completes, thereby reducing manual notifications.

2. Slack Bot Integration for Real-Time Alerts

To centralize notifications and minimize context switching, several stakeholders suggest deploying custom Slack bots. These bots would automatically post concise summaries of pull-request statuses, pipeline outcomes, and upcoming deployment windows into dedicated channels.

Scrum Masters imagine a workflow where bot-driven reminders cue developers to trigger production releases at appropriate times, obviating the need to continuously monitor CI dashboards.

Product Operations also see value in archiving these bot messages for historical auditing, replacing the current practice of manually saving Slack threads.

This would reduce inefficiencies and workload for developers so they would not need to monitor the pipelines constantly. Furthermore, it

ensures developers are deploying their micro-application changes in the correct order. Often times, developers may fail to read the deployment queue file and may deploy micro-application releases in the wrong order.

3. **Streamlined Testing Environments**

Interviewees recommend reducing the number of mandatory test stages, specifically by eliminating redundant DEV-only environments. In addition, they suggest consolidating testing efforts within a single, robust RC pipeline.

Program Managers note that this would expedite deployment by shortening the path from merge to deployable artifact. Complementing this, QA and Program Managers alike emphasize the importance of investing in test flakiness remediation.

These measures aim to restore confidence in automation and reclaim developer and QA time now lost to manual retesting.

4. **Consolidated Release Tracking**

To address fragmented information sources, stakeholders call for a unified tracking system that consolidates release metadata, criticality ratings, deployment dates, and ownership into a single interface.

QA engineers suggest augmenting existing task tickets with additional fields, i.e. planned dates, risk level, and team ownership thus reducing the necessity for separate release tickets.

Similarly, Product Operations recommends codifying release notes directly within Jira, pulling from structured ticket data to generate user-friendly documentation.

Scrum Masters support the creation of a central dashboard, perhaps embedded within Confluence or a bespoke web portal, that aggregates real-time status across both gated and non-gated micro-application pipelines.

4.2 **Technical Implementation**

This section details the design and implementation of the automated governance artifact, developed in response to the challenges and opportunities identified during the AR Planning and observation phase. The primary objective of this design is to eliminate the bottlenecks caused by

manual release ticketing and enhance the flow of information across the software delivery pipeline, thereby improving the target DORA metrics.

The quantitative and qualitative analyses conducted in Chapter 4 revealed three principal pain points within DIP's release process: a lack of centralized, real-time visibility into deployment status; reliance on error-prone manual ticketing for release gating; and insufficient asynchronous communication regarding production changes. Based on these empirical findings, the design of the automation artifact was decomposed into a set of requirements necessary for a successful organizational intervention.

4.2.1 Core Design Requirements

The following requirements informed the technical architecture and user experience of the integrated solution, which combined a release dashboard with a real-time notification system:

1. Changelog for Frontend PRs

Enforce the author of PRs to include a short log of what has been changed in non-technical terms. This would be included within merge checks to ensure the developer has filled in this field before merging and deploying. Reviewers would also be encouraged to review the change log to ensure the data is understandable and usable by other stakeholders.

The purpose of this change log is to:

- Report upwards to marketing, upper management, company wide audience in “Kickoff” or “Wrap up” emails.
- Help the support engineers handling an incident understand what has been released in a non-technical way.
- Generate release notes for products
- Allow Product Managers to follow up on feature progress without asking the team
- Create a visualization of the data where all the changes to a service would be easy to read and look up.

2. Consolidated Release Tracking

As revealed by our quantitative and qualitative analyses, the existing reliance on Jira tickets fails to capture a significant portion of PR merges and production deployments. Moreover, stakeholders consistently describe the manual creation and maintenance of release tickets as unduly burdensome and prone to inconsistency. To address these issues, we propose the implementation of an automated tracking

system that directly ingests data from front-end pull-request pipelines and outputs a real-time, comprehensive release dashboard.

This system will programmatically collect the following attributes for each pull request and deployment event:

- PR author: The individual responsible for the code change
- Team responsible: The feature team creating the change
- Affected micro-applications: The micro-applications to be deployed
- RC approval requirement: Whether the change affects a micro-application requiring RC approval
- User/platform value: a concise description of the end-user impact
- Merge timestamp: The date and time of the PR merge
- RC deployment timestamp: The date and time the build reached the RC environment
- Production deployment timestamp: The date and time the build reached the production environment.
- Rollback or hotfix indicator: Whether the deployment represents a corrective action

By sourcing this data directly from pipeline logs and version-control events, we eliminate the need for developers to manually enter or update fields in Jira. This approach ensures that the data remains accurate, up-to-date, and free from human error. To facilitate downstream analysis and reporting, stakeholders will be able to export the consolidated dataset into spreadsheet formats such as Excel. Proper access controls and encryption protocols will be enforced to protect sensitive information and comply with organizational security policies.

In practice, the automated tracker will operate as an independent micro-service that subscribes to pipeline webhooks and version-control events. Each time a pipeline completes, or a pull request merges, the service will append a structured record to a centralized database. A lightweight web interface will allow Release Managers and Program Managers to filter, sort, and visualize key performance indicators, such as deployment frequency and lead time, across teams and environments. This unified tracking system not only reduces the overhead of ticket maintenance but also provides the transparency required for continuous improvement and executive reporting.

3. **Slack-bot Integration with Release Pipelines**

A second intervention emerged from stakeholder feedback on the repetitive, error-prone task of monitoring pipeline status and deployment queues. Developers currently must manually check Bitbucket for pipeline outcomes and track when their release reaches the head of the queue in order to trigger production deployment. To streamline this process, we have designed a Slack bot integration that delivers event-driven notifications at pivotal stages of the release pipeline.

The bot will issue the following notifications to the PR author:

- **Deployment Queue Head Notification:** When a release reaches the front of the deployment queue, the bot will send an immediate alert, indicating that the build is ready for production promotion. This removes the need for developers to refresh queue dashboards or coordinate with Release Managers.
- **Supersession Alert:** If a queued deployment is superseded by another change, the bot will notify the original author that their build has been displaced. In such cases, the author is instructed to refrain from manually triggering the superseded release, preventing unintended overwrites of another team's changes.
- **Pipeline Failure and Progress Updates:** The bot will subscribe to CI/CD events and post succinct summaries of pipeline failures identifying failed test suites or error code as well as incremental updates such as build success, artifact publication, and RC deployment completion. These real-time messages enable developers to react promptly to failures and avoid repeated polling of the build system.
- **Manual Trigger Reminders:** Upon successful completion of all automated checks, the bot will prompt the developer to initiate the production deployment. This reminder will include a direct link to the deployment action and a timestamp of when the build became eligible for promotion.

Under the hood, the Slack bot will integrate with the CI/CD platform's webhook infrastructure and the consolidated release tracking database. Message templates will be standardized to ensure clarity and consistency and the notifications will be directed to a dedicated channel.

A pilot deployment of the Slack-bot will be conducted with one feature team. Success metrics will include a reduction in manual wait time, a decrease in superseded deployment incidents, and positive feedback in developer experience surveys. Upon validation, the integration will be extended to all frontend teams, establishing a responsive, event-driven notification layer that significantly reduces cognitive load and accelerates the end-to-end release process.

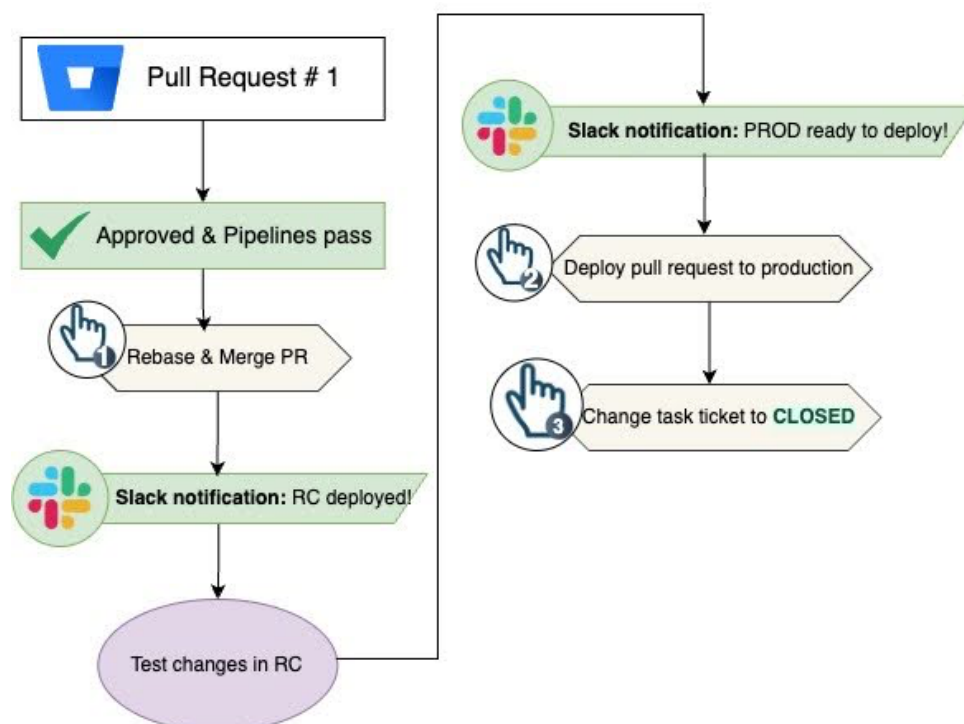


Figure 11: Flow chart showing the streamlined process for releases.

In stark contrast to the legacy workflows described in Section 4.1.2, where gated releases required 16 discrete developer actions and non-gated releases required 10, the new flow consolidates the developer’s responsibilities into just three key steps:

1. **Merge and Notify**

Once a PR that does not require RC approval has been approved and all CI/CD pipelines have passed, the developer rebases and merges the change. No Jira release ticket is required. Immediately upon deployment to the release-candidate environment, the integrated Slack bot alerts the author, who may then perform any necessary validation in RC.

2. **Production Trigger and Confirmation**

When the build reaches the head of the deployment queue and is ready for manual promotion, the Slack bot issues a “ready for production” notification. The developer then initiates the production rollout. Upon successful deployment, a final Slack message confirms that the change is live.

3. **Task Closure**

With the release complete (or if the build has been superseded by another change, as indicated by a supersession alert), the developer simply closes the original Jira task ticket. No additional ticket updates or environment specific status changes are required.

This condensed process eliminates redundant ticket maintenance, reduces waiting and monitoring overhead, and leverages real-time notifications to ensure that developers can focus on feature work rather than release logistics. By reducing the number of manual steps, the new flow is designed to minimize lead time and maximize throughput across DIP’s front-end delivery pipeline.

Building on the analysis findings and the agreed implementation plan, the technical implementation will deliver two complementary automation tools.

The first tool is a unified release-tracker dashboard that consolidates metadata directly from CI/CD pipelines and the VCS, providing a single dashboard to track all frontend deployment events. The second tool is a Slack bot integration that issues real-time notifications to developers at key milestones, such as when their build is ready to be deployed to production or superseded by a newer release.

To capture the user or platform value without relying on Jira tickets, the PR template will be extended to include a mandatory “User/Platform Value” field. The CI pipeline will enforce this requirement by failing any build in which the field is left blank. Once populated, this text will be extracted automatically into the release tracker dashboard, enabling the generation of complete reports on frontend releases and their end-user impact. The following section details the technical architecture and implementation of the unified release tracker dashboard.

4.2.2 **Release Tracker Dashboard**

The release tracker dashboard’s primary functionality is to provide a user interface which allows tracking the status of releases and generating reports for upward reporting. This requires collecting the necessary data into a centralized database and displaying the data on an easy-to-use interface for relevant stakeholders to access.

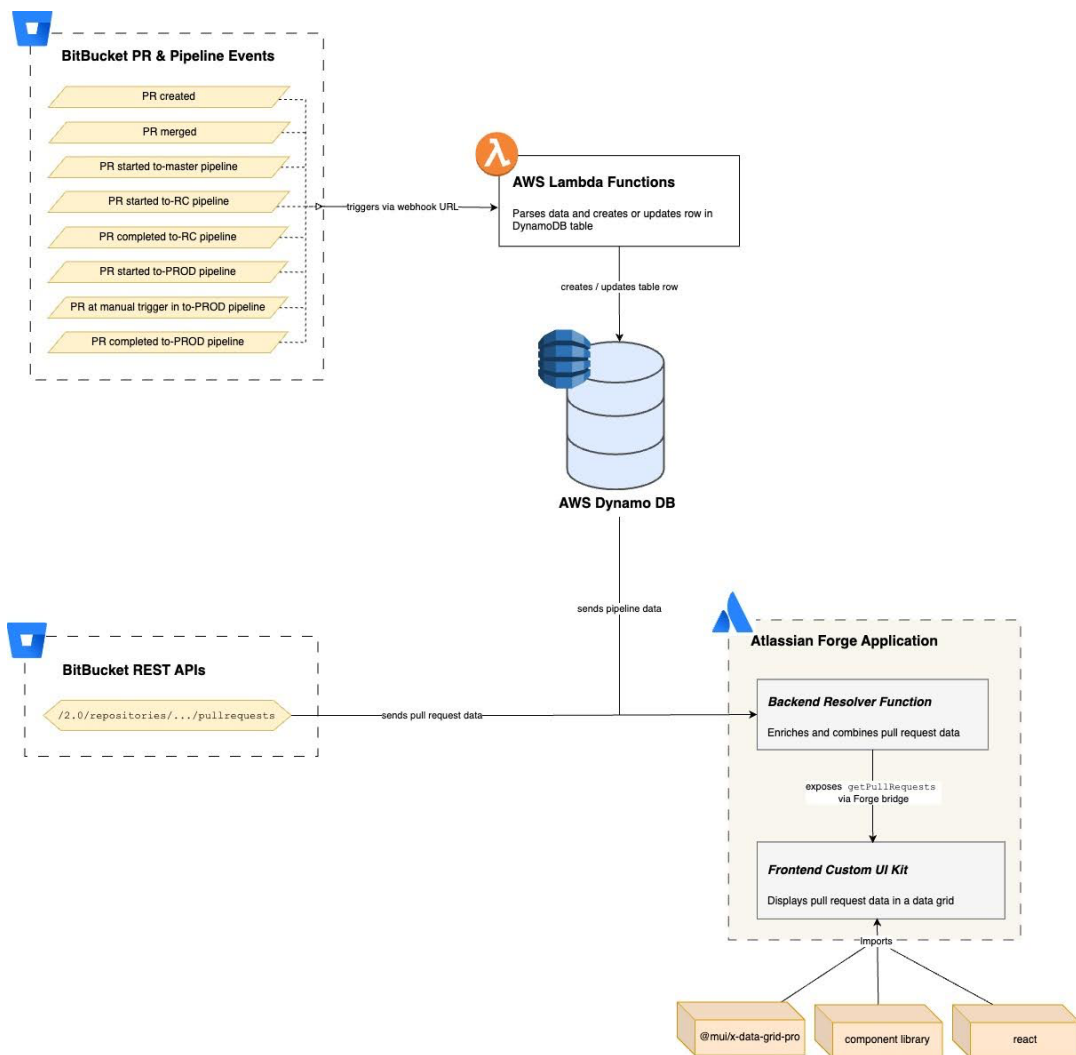


Figure 12: Architecture diagram detailing the frontend release tracker.

As depicted in

Figure 12, when a PR enters certain stages in its lifecycle or pipeline steps, it sends a request to a webhook URL with information about the PR stage which triggers an AWS Lambda function. This function parses the information received and either creates a new record in an AWS Dynamo Data Base (DB) or updates an existing record with the provided state.

The release tracker application backend retrieves data about the pipelines from the table in Dynamo DB and enriches it with the pull request data retrieved from Bitbucket APIs. The application frontend requests the pull request data from the backend and displays it on a data grid built with custom components on React.

This application is hosted and built using the Forge platform, a cloud-native extension framework provided by Atlassian, which seamlessly integrates with Bitbucket, the VCS for the frontend codebase.

The following section will explore how the Forge platform works, and how it was used for the technical development of the release tracker dashboard.

The Forge Platform

Forge is a cloud application development platform which serves developers an infrastructure for hosting applications. The Forge infrastructure is automatically managed, provisioned, monitored and scaled by Atlassian, allowing developers to focus efforts solely on application development and business logic [32].

Forge applications are written in JavaScript and supports the runtime execution environment of Node.js, and consequently all Node modules, whether they are built-in modules, local modules or third-party modules can be imported within the application. It signifies that all Node libraries and Node Package Manager (NPM) packages are all compatible, and so, the entirety of the JavaScript ecosystem can be leveraged.

At the core of the Forge platform is a serverless Functions as a Service (FaaS) platform, which utilizes Amazon Web Services (AWS) Lambda. Consequently, applications developed with Forge are executed within a security layer that invokes data egress restrictions by default. This means outbound network requests are confined by an egress-control proxy, which enforces the exact permissions declared in the app's manifest.

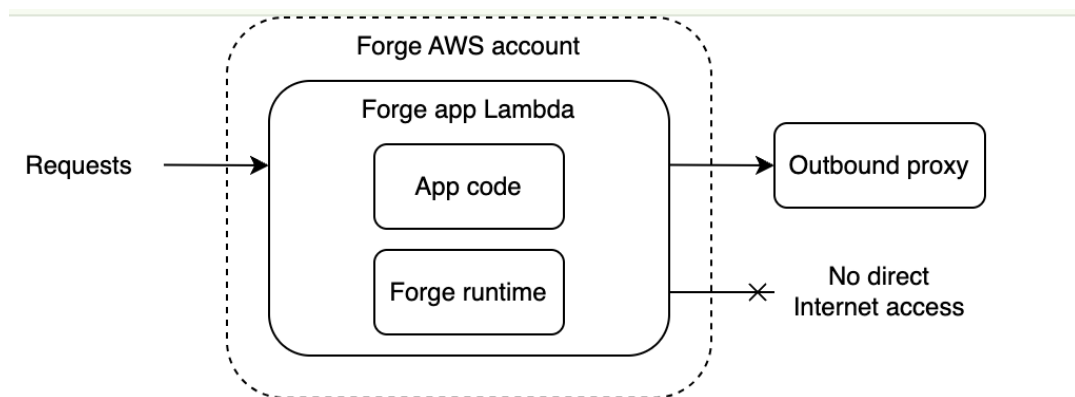


Figure 13: Diagram depicting the security model of the Forge platform [32].

The security model of the Forge platform is founded on strict isolation of application execution and controlled access to data and external systems. Since each Forge function runs within a dedicated AWS Lambda instance with sandboxing for each application, it prevents one application from interfering with or modifying the execution of another.

By separating functions in separate AWS accounts, Forge further minimizes privilege scope, i.e., even in the event of a vulnerability that allows

sandbox escape, the compromised account holds only the minimal entitlements necessary for app operation.

Data management within Forge adheres to a shared-responsibility model. Although the platform ensures that no in-memory or on-disk state persists beyond the lifecycle of a function invocation, developers must also partition any global caches by tenant identifier to prevent cross-customer data leakage. This approach is essential in a multi-tenant environment, where accidentally shared object, such as issue-key cache, could otherwise conflate data between distinct customers.

Forge supports three distinct environments, i.e., development, staging and production, with each maintaining its own code bundle, manifest configuration, module registry and set of environment variables.

The development environment permits log inspection and local tunnelling for iterative debugging. The staging environment mirrors production restrictions but serves continuous deployment pipelines rather than ad hoc development. The production environment enforces the highest security i.e., tunnelling is disabled, and site administrators retain the authority to restrict access to logs and other sensitive diagnostics.

Forge clients additionally automatically attach the correct OAuth-2.0 scope declared in the app manifest to each request, ensuring that applications request only the permissions they require.

By shifting credential handling to the Forge infrastructure, the platform eliminates a common source of security vulnerabilities and simplifies the developer experience, while preserving a least-privilege model that creates end-user trust.

Forge runtime consists of a group of APIs which allow interaction with Representational State Transfer (REST) endpoints. Forge automatically provides the context the function is executing within to these APIs. For instance, when the Forge function is run within the Bitbucket service, it automatically injects details of the current repository and workspace with the request. This assures that functions are executed on the proper tenant.

Forge provides two tools for building user interfaces for applications. The first of these is the UI Kit, which provides a set of UI libraries and components with similar visual appearance as Atlassian products. This option is viable for a developer to get up to speed with developing the UI without the need to build their own custom UI components.

On the other hand, the Custom UI toolkit provided by Forge gives full freedom to the developer to integrate any UI libraries they require. This application is run within an iframe, which then isolates the environment the UI is served from.

The implementation of the release tracker dashboard employed the Custom UI toolkit since it provided flexibility with the visual appearance of the UI components, thereby allowing the use of DIP component library, built on top of Google's Material UI.

When building applications with a frontend, whether it uses the UI Kit or Custom UI toolkit, Forge provides resolvers. Resolvers are essentially functions which enable development of a backend for the application. Within resolvers, it is possible to fetch external data and perform server-side operations.

All in all, utilizing Forge confers several benefits, i.e., seamless integration with the Atlassian ecosystem, enterprise-grade security enforced by default, and a serverless execution paradigm that scales elastically in response to user demand. This architecture accelerates feature development, reduces maintenance overhead, and delivers real-time visibility into frontend deployment activity with minimal infrastructural complexity.

The next section will explain how data was retrieved and enriched within the backend implementation of the application.

Application Backend

The backend layer of the release tracker dashboard is responsible for assembling a comprehensive, real-time view of frontend delivery activity.

To achieve this, it integrates two complementary data streams, static pull request metadata from Bitbucket’s REST API and dynamic pull request lifecycle events emitted by the pipeline CI configuration. As depicted in

Figure 12, the pull request lifecycle events are consolidated in a DynamoDB table where each record is keyed by the merge commit hash and includes timestamps for each significant milestone from pull request creation to production deployment. The backend defines one Forge resolver function named “*getPullRequests*” with the following pseudo code:

```
resolver.define("getPullRequests", async (req) => {
  const { days = DEFAULT_DAYS_OFFSET, context } = req.payload;

  try {
    const prLifecycles = await getPullRequestLifecycles(days);

    const prs = await getPullRequests(days, context);

    const enrichedPrs = await enrichPrs(prLifecycles, prs, context);

    return enrichedPrs;
  } catch (error) {

    console.error(error);
    throw error;
  }
});
```

The application backend programmatically retrieves pull request details via a GET request to the Bitbucket 2.0 API endpoint “/2.0/repositories/{workspace}/{repo_slug}/pullrequests?q={query}&fields={fields}”

The endpoint requires passing the workspace and repository identifiers via URL parameters, while it supports additional filtering and sorting through the query parameters q and fields. By default, this endpoint returns all open pull requests on the specified repository. Therefore, the application backend supplies filters for retrieving only all merged pull requests within the desired time range defined by the application client.

The endpoint returns several key information about the pull request including the title, summary provided by the PR author, state, author, source branch, destination branch, creation timestamp, last updated timestamp etc. [33].

In order to limit the data returned, only relevant fields are specified in the request to the endpoint. The relevant attributes are:

- PR ID
- Title
- Summary
- Merge commit hash

The PR ID and title are relevant to display to the user on the UI for report creation and visual purposes. The PR ID is additionally important for generating links to the PR.

Furthermore, the summary is important for extracting useful information, i.e., the user entered “User/Platform Value” for reporting purposes. Every PR summary includes a subsection titled “User/Platform Value” which prompts the author to write a short description of what user or platform value the PR provides. This would be extracted and included for reporting the added value every PR provides in non-technical terms. It is also important for quickly triaging incidents by engineers with little context of the changes introduced in the PR. This field is extracted by the backend using the Regular Expression (RegEx): */User Platform Value.*?\n\n(?:\n|\$)/*

Finally, the merge commit hash is important since it is the key identifier for records in the Dynamo DB table. This is the field used for linking the pull request data with the pipeline events data.

Since the Bitbucket REST API for retrieving pull request information does not expose enough details for building a comprehensive release tracker dashboard, these details are manually collected via scripts instrumented at pre-defined hooks within the Bitbucket pipeline configurations.

These hooks correspond to lifecycle events including pull request creation, master branch pipeline start, RC pipeline start, RC deployment, production pipeline start, pending production trigger, production deployment, and supersession by a subsequent release.

Each script invocation issues an HTTP POST to a secured webhook URL, transmitting the merge commit hash and a timestamp. An AWS Lambda function, fronted by this webhook, processes each incoming event by updating or inserting a DynamoDB item whose primary key is the merge commit hash. The table schema allocates dedicated attributes for each stage's timestamp and for flags such as *isHotfix* or *supersededBy*. Over time, each DynamoDB record accumulates a complete history of its associated pull request: when it was opened, merged, built, validated in RC, queued and promoted to production, or superseded.

By joining the pull request metadata with the event driven pipeline records, the backend service can serve the frontend dashboard with enriched release documents. These documents include user value annotations, deployment timelines, and supersession alert, all without manual ticket entry. This architecture not only ensures data accuracy and freshness but also provides the foundation for actionable metrics on lead time, deployment frequency, and risk reduction.

Application Frontend

The release tracker dashboard is surfaced as a Forge hosted application within the primary Bitbucket repository's side navigation, ensuring immediate access for any member of the DIP Bitbucket organization. By embedding directly into the existing developer workflow, the dashboard requires no separate login and leverages the platform's Single Sign On (SSO) capabilities.

This is achieved by defining the module within the Forge application manifest file as below:

```
modules:
  bitbucket:repoMainMenuPage:
    - key: fe-release-tracker-repo-main-menu-page
      resource: main
      resolver:
        function: resolver
      title: Frontend Releases
    function:
      - key: resolver
        handler: index.handler
```

Data is fetched via Forge resolver functions, which invoke a GraphQL endpoint backed by AWS AppSync. The AppSync layer routes requests to Lambda resolvers that query the DynamoDB table. The pseudo code for the frontend react hook that calls invokes the resolver function is displayed below:

```

import { invoke, view } from "@forge/bridge";

const usePullRequests = (days) => {
  const [pullRequests, setPullRequests] = useState([]);
  const loadPullRequests = useCallback(async () => {
    try {
      // Set error & loading states
      const context = await view.getContext();
      const response = await invoke("getPullRequests", {
        context,
        days,
      });
      setPullRequests(response);
    }, [days]);

    useEffect(() => {
      loadPullRequests();
    }, [days]);
    return { loadPullRequests, pullRequests, ... };
  });

  export default usePullRequests;

```

To maintain visual consistency with the rest of the DIP product suite, the user interface was constructed using the internal DIP component library. This provided a set of pre-styled form controls, buttons and layout primitives that align with corporate branding and interaction patterns. Material UI's grid component was customised to display deployment records in a tabular format.

Frontend Releases

TIME PERIOD: Last week | ENVIRONMENT: All | APPS: Select apps | TEAMS: Select teams

RC Approval Required | No RC Approval

172 pull requests found | Last updated: 17 Feb 2025, 08:00 | Refresh | Search... | Export

Summary	Task Ticket	Team	Affected Apps	Merge Date	RC Deployed	Prod Deployed
Add data-processing module Created by Jane Doe #11792	JIR-1234	team-a	app 1	17 Feb 2025, 07:05	17 Feb 2025, 07:22	pending
Implement new visualization component Created by John Doe #11791	JIR-567	team-b	app 1	17 Feb 2025, 07:05	17 Feb 2025, 07:22	pending
Update API integration layer Created by John Doe #11788	JIR-890	team-b	app 2	17 Feb 2025, 07:04	17 Feb 2025, 07:22	pending
Refactor notification service Created by Jane Doe #11783	JIR-1112	team-a	app 3	14 Feb 2025, 23:51	15 Feb 2025, 00:08	pending
Add configuration options for filters Created by Jill Doe #11786		team-c	app 1 app 2	14 Feb 2025, 23:42	15 Feb 2025, 00:00	pending
Improve performance of search endpoint Created by Jack Doe #11805	JIR-1314	team-c	app 3	14 Feb 2025, 22:24	14 Feb 2025, 22:39	pending
Implement caching for dashboard data Created by James Doe #11782		team-d	app 1	14 Feb 2025, 20:10	14 Feb 2025, 20:30	pending
Add authentication middleware Created by John Doe #11760	JIR-1516	team-b	app 3	14 Feb 2025, 19:58	14 Feb 2025, 20:16	14 Feb 2025, 21:35
Enhance error handling in client app Created by Jill Doe #11788		team-c	app 1	14 Feb 2025, 19:06	14 Feb 2025, 19:24	pending
Update release automation workflow Created by Jack Doe #11802		team-c	No applications found	14 Feb 2025, 18:43	stopped	stopped
Add analytics tracking events Created by Jane Doe #11798	JIR-1718	team-a	app 2	14 Feb 2025, 16:49	14 Feb 2025, 17:07	pending

Figure 14: Screenshot of the release tracker dashboard frontend displaying all the merged pull requests and relevant information. For confidentiality, data shown in this screenshot has been anonymized.

As shown in Figure 14, the data grid supports multi-dimensional filtering to help stakeholders isolate relevant deployments. Available filter dimensions include:

- Time period: Selectable range up to six months of historical data.
- Environment: DEV, RC or PROD deployments.
- Applications: Specific micro-application names as affected by the pull request.
- Teams: Filtered by the feature team responsible, inferred from the branch name convention requiring each branch to begin with the author's team or username.
- RC approval requirement: Boolean flag indicating whether approval was required to merge this pull request.

Columns may be sorted in ascending or descending order. By default, the grid presents the most recent production deployment at the top. A CSV export function enables users to download the filtered dataset with extended attributes for offline analysis and upward reporting. The export includes:

- Hotfix indicator: yes or no flag
- Rollback indicator: yes or no flag.

- User/Platform value: Text extracted from the pull request summary.
- Pull request author
- Supersession timestamp: Time when build was superseded.

Superseded deployments are visually highlighted in the grid with a yellow status dot. This feature aids rapid identification of changes that were later overridden.

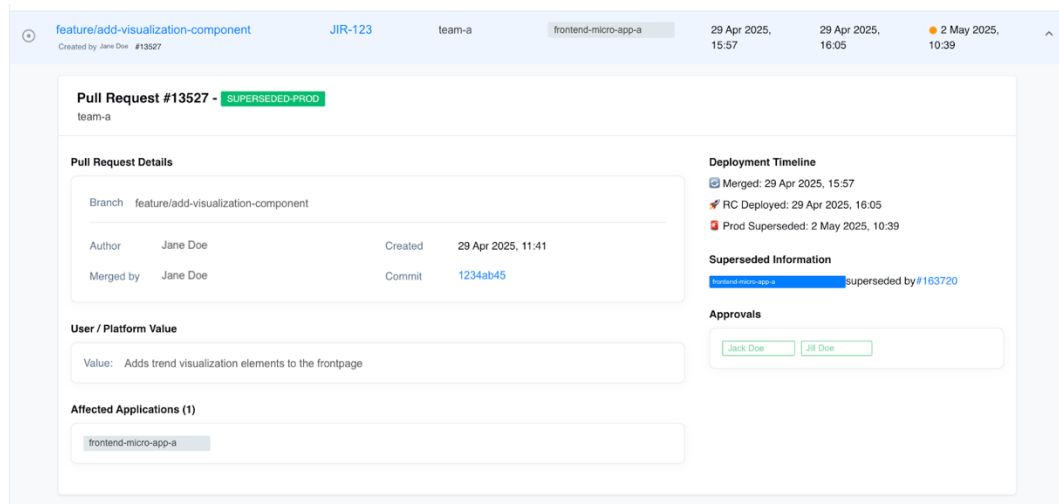


Figure 15: Screenshot of the expanded state displaying more information about a deployment. For confidentiality, data shown in this screenshot has been anonymized.

Each table row may be expanded to reveal a detailed view of the deployment lifecycle. In the expanded pane the following information is displayed:

- Approvers: List of users who approved the pull request.
- Timestamps for all pipeline stages, from PR creation through production deployment.
- Supersession history: Identifier of the release that displaced this deployment, if applicable.

This custom-built application implements all the tracking requirements defined during the AR planning phase. It provides a unified, real-time interface for monitoring frontend deployments, supporting both high level reporting and detailed forensics without manual ticket upkeep.

While this release dashboard helps to alleviate the challenges with gathering reporting data and tracking release real-time without manual ticket creation, a real-time notification system is necessary to ensure developers are

informed about key milestones in the lifecycle of their releases without manually checking release pipeline statuses and frontend deployment queues.

The next section will explore this notification system and the details of the technical implementation to achieve a seamless developer flow when deploying pull requests.

4.2.3 Real-Time Notification System

To enhance the centralized visibility offered by the dashboard and eliminate the overhead associated with manual status checks, a real-time notification system was integrated directly within the Bitbucket deployment pipelines. This notification mechanism provides immediate feedback to developers and stakeholders at critical stages in the deployment lifecycle, thereby enabling timely interventions and streamlined deployment workflows.

Real-time notifications are triggered at various significant pipeline events: upon merging a pull request into the master branch, following successful or failed deployments to the RC or production environments, when deployments are ready for manual promotion to production, and when a deployment has been superseded by a newer release.

Deployment Initiation Notifications

The initial notification event occurs immediately after a pull request is merged into the master branch. Within the Bitbucket pipeline, a script is executed at this merge step, which posts a notification message directly into a designated Slack channel. This notification explicitly mentions the author of the pull request, facilitating prompt acknowledgment and response, as depicted in Figure 16.

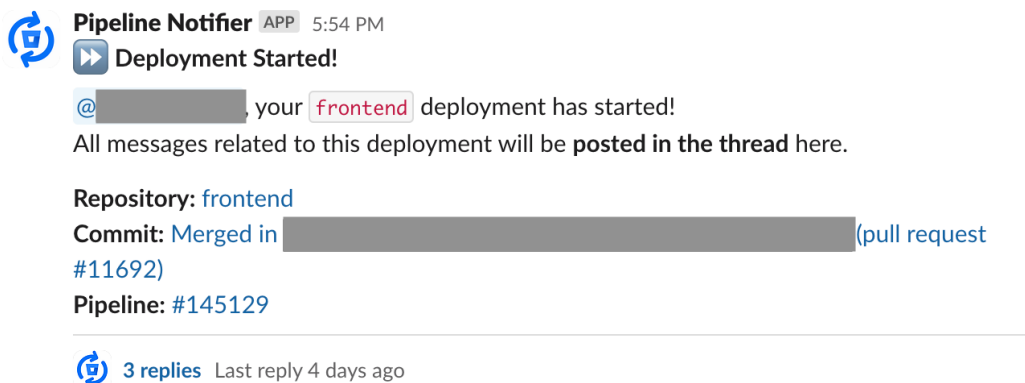


Figure 16: Slack notification screenshot of a deployment that has started.

Due to privacy restrictions, Bitbucket's API does not provide the email addresses of PR authors directly, instead returning only a unique identifier

known as the Bitbucket Universally Unique Identifier (UUID). However, to send targeted Slack notifications to individual authors, it is necessary to translate this Bitbucket UUID into a corresponding Slack user ID.

To solve this, a mapping between Bitbucket UUIDs, Bitbucket email addresses, and Slack user IDs is maintained. This mapping is programmatically created by fetching and cross-referencing user data from both Slack and Bitbucket APIs.

First, the Slack API provides a list of all Slack users, including their email addresses and Slack IDs. Next, the Bitbucket API returns user data that includes email addresses and their associated UUIDs. By matching users based on their email addresses from these two datasets, a reliable map linking Bitbucket UUIDs to Slack user IDs is generated.

This mapping is then stored as a Yet Another Markup Language (YAML) file and cached within the pipeline infrastructure. To ensure accuracy and keep the map updated as new users are added or existing user information changes, this process is executed automatically on a weekly basis.

The following is an example of the code responsible for generating this mapping:

```
const createSlackBitbucketUserMapping = async () => {
  const slackUsers = await getSlackUsers();

  const emailToSlackId = slackUsers.reduce((acc, user) => {
    const email = user.profile.email?.toLowerCase();
    if (email && user.id) {
      acc[email] = user.id;
    }
    return acc;
  }, {});

  const emails = Object.keys(emailToSlackId);

  const bitbucketUsers = await getBitbucketUsers(emails);

  const users = bitbucketUsers
    .reduce<UserIds[]>((acc, account) => {
      const email = account.user.email.toLowerCase();
      const slackId = emailToSlackId[email];
      if (!slackId) {
        return acc;
      }
      acc.push({email, slackId, bitbucketUuid: account.user.uuid});
      return acc;
    }, [])
    .sort((a, b) => a.email.localeCompare(b.email));
  const yamlData = yaml.dump(users);
  await fs.writeFile(SLACK_BITBUCKET_USER_MAPPING_PATH, yamlData,
    'utf8');
};
```

This approach ensures consistent, accurate, and privacy compliant communication between the Bitbucket pipelines and Slack notifications.

Deployment Success and Failure Notifications

Similar scripts are invoked within Bitbucket pipeline steps to notify stakeholders regarding successful or failed deployments to the RC and production environments. As illustrated in Figure 17, these notifications include pertinent details such as a direct link to the pull request, precise timestamps of deployment events, and any additional context required to promptly address issues or acknowledge successes.

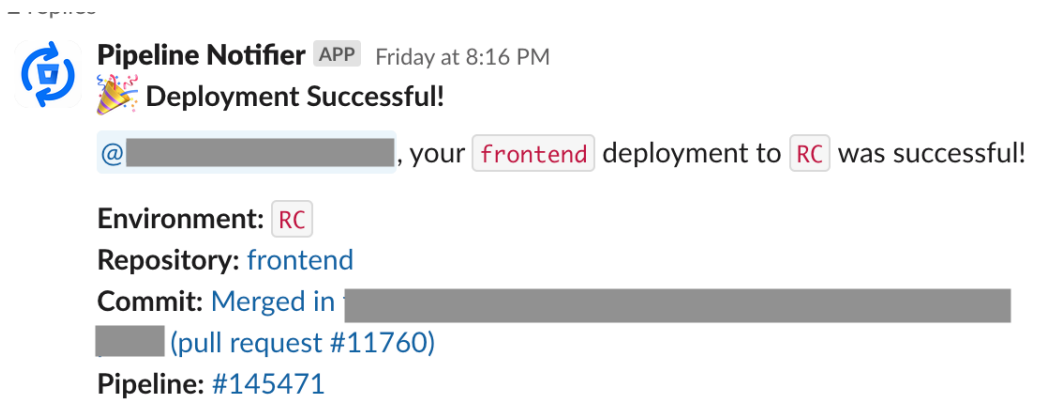


Figure 17: Slack notification screenshot of a RC build that was deployed.

Deployment Queue Based Notifications

Notifications for deployments that are ready to move into production or have become superseded due to newer builds require more sophisticated logic involving deployment queue analysis.

The deployment queue determines the order in which frontend pull requests are deployed to production, and it is computed based on a detailed analysis of the deployment graph. This graph is dynamically constructed during specific steps within the Bitbucket pipeline, using custom pipeline scripts designed to capture and visualize dependencies among deployments.

The deployment graph is generated at two critical points within the production deployment pipeline (i.e., to-PROD):

- 1. Before Manual Deployment Trigger**

When the pipeline reaches the stage immediately before the manual production deployment trigger, a script is executed with an argument labelled "ready-to-deploy".

2. After Manual Deployment Trigger

Upon successful manual triggering of the deployment to production, the script runs again, this time with the argument "deployed".

This two-step approach ensures that the deployment graph consistently reflects the current pipeline status and captures accurate data at pivotal milestones in the deployment lifecycle. Upon execution, the pipeline script first retrieves all production related pipelines from the previous two weeks via Bitbucket's REST API. Pipelines are sorted by their commit date and build number, ordering them from newest to oldest. Each pipeline entry is then enriched with critical metadata statuses, as follows:

- *isCurrentPipeline*: Boolean value set to true if the pipeline's build number matches the current pipeline invoking the script.
- *failed*: Indicates whether the pipeline has failed.
- *deployed*: Determined based on the pipeline state; if the pipeline is currently executing the script, the "deployed" status is inferred from the script argument. Otherwise, this status is retrieved from Bitbucket's API state field (i.e., status = "DEPLOYED").
- *readyToDeploy*: Determined based on the pipeline state; if the pipeline is currently executing the script, the "ready-to-deploy" is inferred from script argument. Otherwise, this status is retrieved from Bitbucket's API state field (i.e., status = "PAUSED").

At this stage, pipelines that do not include changes to any micro-applications are excluded from further analysis to maintain clarity and relevance within the deployment queue.

With the enriched pipeline data, the deployment graph is constructed using "GraphLib", a JavaScript based graph library capable of representing complex dependency structures. Each pipeline is represented as a node within the graph, identified uniquely by its pipeline build number.

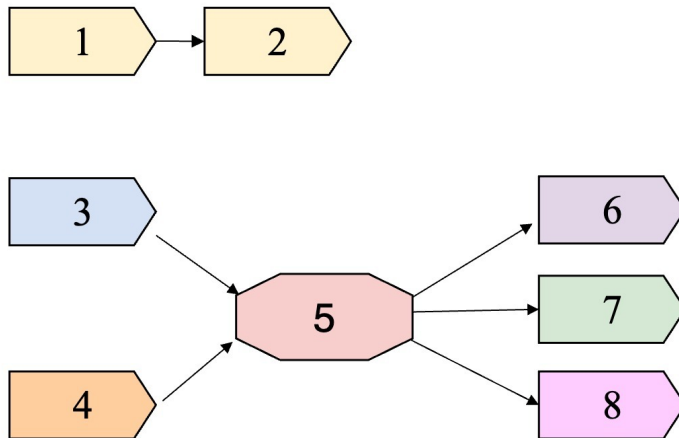
Dependencies among deployments, i.e., pipelines that affect the same micro-application, are established via directed edges, ordered according to their insertion sequence into the graph (newest to oldest).

The final constructed graph contains three primary components:

1. *Pipelines graph*: A comprehensive dependency graph where each node is labelled with its unique build number and interconnected by an edge with the nodes affecting the same micro-application.

2. *Pipelines per app*: A mapping from each micro-application name to an array of pipelines affecting it.
3. *Superseded pipelines*: A mapping from pipeline build numbers to metadata indicating which subsequent pipeline has superseded it.

pipelinesGraph



pipelinesPerApp

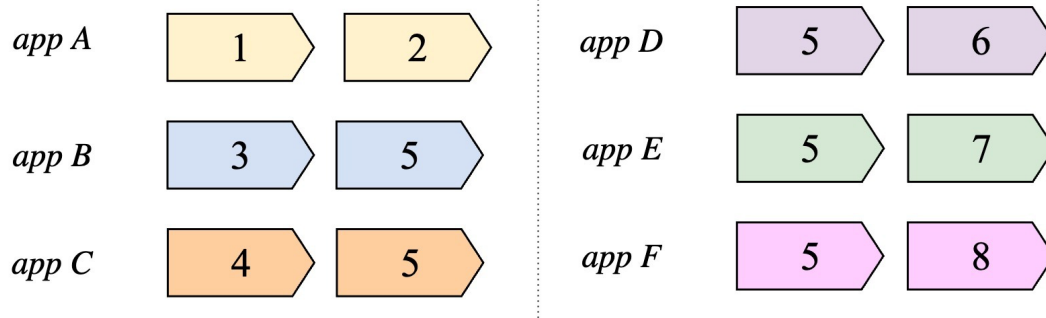


Figure 18: Illustration of generated pipelines graph and pipelines per application.

Once the complete graph is established, it undergoes several transformation steps to accurately represent the actionable deployment queues:

1. **Pruning Deployed or Superseded Pipelines**

Nodes representing pipelines already marked as "deployed" or "superseded" are removed, leaving only those deployments pending action.

2. Graph Segmentation

The resulting pruned graph is decomposed into independent subgraphs. Each subgraph represents a distinct group of deployments interconnected by shared micro-application dependencies, with no dependencies linking them to other subgraphs. For example, an original unified graph containing two sequences $A \rightarrow B$ and $C \rightarrow D \rightarrow E$ would be segmented into two independent graphs: one containing $A \rightarrow B$, and the other $C \rightarrow D \rightarrow E$.

3. Topological Sorting

Each subgraph is topologically sorted to produce a linear ordering of deployments that respects all identified dependencies. The resulting ordered array defines the precise sequence in which deployments must occur to maintain functional integrity and avoid accidental over-rides or rollbacks.

4. Queue Representation

The sorted subgraphs are encapsulated into structured deployment queue objects, each containing:

```
{
  "deployment": "subgraph reference",
  "order": ["sorted build numbers"]
}
```

5. Queue Aggregation

Each independent deployment queue object is subsequently aggregated into an array representing all active deployment queues awaiting action.

6. Queue Visualization and Reporting

To provide stakeholders with immediate visual context, an image representation of the deployment queue graph is dynamically generated from the aggregated subgraphs as shown in Figure 19. This visualization is cached for rapid retrieval. Additionally, a structured deployment queue report is created for each queue and stored locally. These resources aid Release Managers, developers, and program managers in making informed decisions and identifying deployments requiring immediate attention.

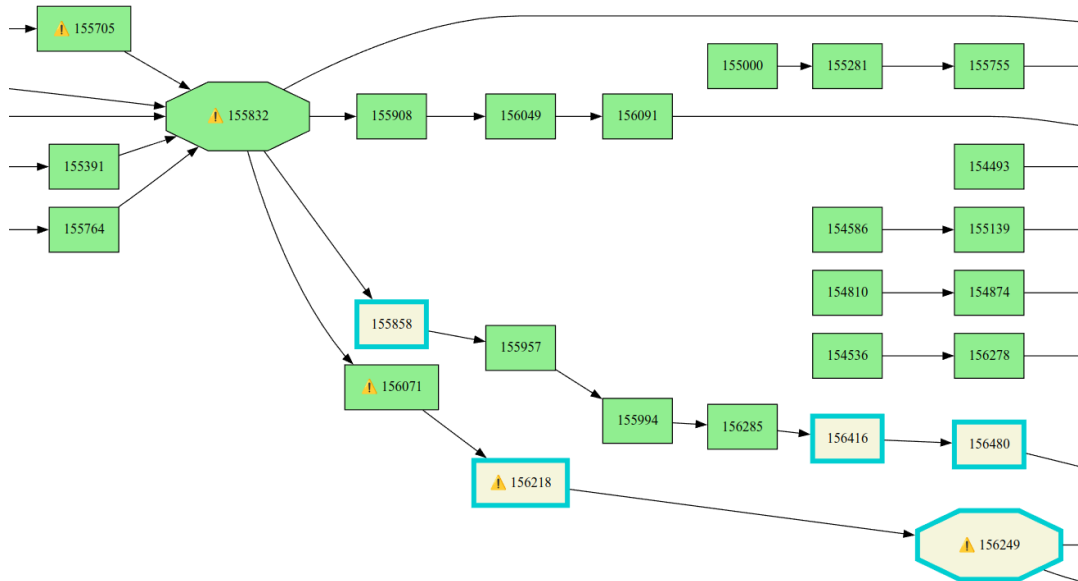


Figure 19: Deployment graph depicting multiple interconnected deployment queues.

In the visual representation of the deployment queue depicted in Figure 19, each box represents an individual production deployment build and the order in which they should be deployed. Builds which include multiple applications in a single deployment are represented as hexagons, while builds which require RC approval before proceeding to deploy are denoted with a warning icon.

These deployment graphs are rebuilt and updated after a pipeline completes building the production deployment artifact, ensuring the graph represents the current state of the production builds at any given moment.

Finally, the system proactively informs developers of deployment queue updates through real-time Slack notifications. Each deployment queue is analysed for nodes without incoming edges, representing deployments that can safely proceed to production immediately. For each of these ready-to-deploy nodes, a Slack notification is dispatched via webhook, alerting the responsible developer that their deployment has reached the front of the queue, as shown in Figure 20.

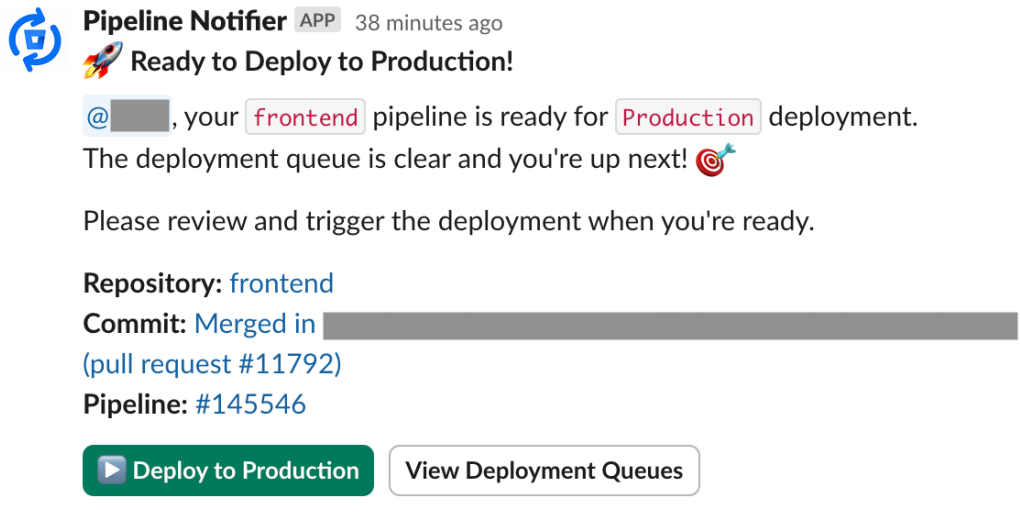


Figure 20: Notification for deployment ready to be deployed to production.

The system tracks notification timestamps and ensures duplicate alerts are not sent within a 24-hour period for the same pipeline. Similarly, notifications are sent when pipelines become superseded due to subsequent deployments, again with built in duplicate message suppression.

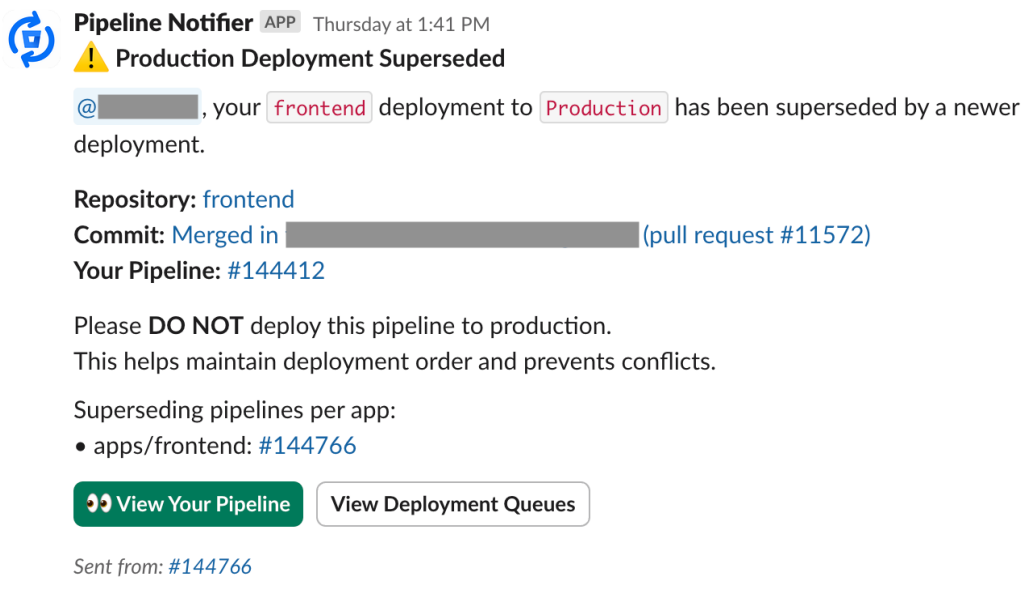


Figure 21: Notification for deployment superseded by a newer build.

To maintain organizational clarity, notification metadata, which includes Slack channel identifiers and thread reference, is cached using Amazon S3, indexed by commit hash and commit date. Subsequent notifications related

to the same deployment are posted within the original Slack thread, ensuring cohesive and context rich communication.

In summary, the notification workflow comprises the following sequential steps:

1. Pipeline scripts trigger notifications upon specific events e.g., merge to master, deployment success/failure etc.
2. Bitbucket UUIDs are mapped to Slack IDs to accurately identify notification recipients.
3. Deployment queue statuses are dynamically analysed to determine readiness and supersession scenarios.
4. Notifications are dispatched via Slack webhooks, delivering immediate visibility to relevant stakeholders.
5. Metadata regarding notifications is cached to maintain coherent Slack messaging and to prevent redundant notifications.

The comprehensive integration of real-time notifications within the deployment pipeline significantly streamlines communication, reduces manual overhead, and enhances developer responsiveness during the deployment lifecycle.

4.3 Addressing Challenges

Within the first six months following the technical rollout of the automation artifact, several organisational and cultural factors impeded the broader adoption of the release dashboard within the company. These challenges underscore the necessity of addressing socio-technical dimensions alongside purely technical implementations to effectively achieve widespread adoption of streamlined deployment processes.

A primary barrier was cultural resistance, largely stemming from previous high-profile incidents involving inadvertent rollbacks and unexpected production issues. These events had significantly diminished stakeholder confidence in allowing developers to independently manage deployment processes without explicit manual oversight. Consequently, stakeholders, particularly program managers, demonstrated reluctance towards transitioning from established manual approval processes, even for micro-application releases verified through extensive testing.

In addition, internal accountability structures contributed further to these adoption challenges. Specifically, Program Managers bore direct

responsibility for the frequency and severity of deployment incidents, creating a strong incentive towards maintaining conservative deployment controls. This responsibility structure diverted attention away from addressing underlying causes of incidents and process inefficiencies, reinforcing preferences for existing risk averse practices.

Program Managers also consistently emphasised the importance of uniformity across deployment workflows for both frontend and backend services. The introduction of differing, more autonomous workflows exclusively for frontend deployments was viewed as potentially adding complexity, despite clear evidence demonstrating increased deployment efficiency and responsiveness.

Lastly, stakeholders initially exhibited low levels of trust in the newly introduced automated release tracker dashboard. Feedback indicated that incremental improvements, along with sustained exposure and familiarity with the tools, would be necessary to increase confidence and enable full integration into existing deployment practices. Together, these impediments underline the necessity of addressing organisational and cultural dimensions alongside technical implementations to effectively achieve widespread adoption of automated and streamlined deployment processes.

4.4 Organizational Rollout

While the release dashboard initially faced resistance, the accompanying Slack notification system was widely adopted within the first month of rollout. The immediate success of the asynchronous notifications in providing real-time deployment visibility validated the underlying organizational need for transparency without adding mandatory friction. This component's rapid success provided the essential confidence foundation for the subsequent, mandatory organizational transition.

The final organizational rollout was driven by a conjunction of three strategic factors:

- 1. Structural Change and Accountability Shift:**

The organizational decision to remove the Scrum Master, Product Owner, and QA roles structurally eliminated several key manual gatekeepers. Simultaneously, the responsibility for maintaining stability was formally shifted entirely to the feature development teams.

- 2. Addressing Uniformity Concerns:**

The resistance from Program Managers regarding the lack of uniformity between frontend and backend processes was addressed by extending the governance principles of the automation artifact to backend services. While full enforcement of deployment order was initially limited on certain platform, e.g., Bitbucket, the commitment to

unifying the visibility layer via the Deployment Registry App satisfied the demand for a consistent, cross platform release audit trail.

3. The Automation Workflow:

The combination of the real-time notification system and the comprehensive audit logging of the Deployment Registry App provided a robust, data driven alternative to the subjective manual approval process. This enabled leadership to make a data backed decision that the fully automated process had reached sufficient maturity to replace the manual gate.

These converging pressures culminated in a formal, organization wide announcement in November 2025, establishing a non-negotiable deadline for the adoption of the new tooling and the simultaneous decommissioning of the legacy systems.

5 Validation

With the full deployment of the automated governance artifact described in the previous chapter, the final phase of the AR cycle focused on objective measurement and impact assessment over the final observation period. The primary objective of this evaluation is to move beyond the high-level summary of results and perform the critical step of triangulation, where the objective performance metrics are interpreted against the subjective experiences of key personnel.

This process isolates the specific causal link between the deployment of the automated governance artifact and the observed organizational change. The analysis is divided into the quantitative and qualitative analyses against the metrics measured during the initial state assessment.

5.1 Quantitative Analysis

The quantitative analysis post-intervention focuses on the DORA throughput metrics, i.e., DF and LTC. While the baseline period was selected to capture a full six month cycle for robust statistical averaging, the post-intervention data was collected over a three month period from 5th August 2025 to 5th November 2025, immediately following the final deployment of the automated solution.

5.1.1 Deployment Frequency

As established in the methodology, DF is measured based on the average number of deployments from the frontend continuous integration pipelines. It is essential to note that not every PR merge necessarily correlates with a deployment, as some changes may produce artifacts that are deemed undeployable (e.g., updates to test infrastructure or pipeline configurations).

The following analysis reports the deployment statistics for the frontend micro-frontend over a three month final observation period, specifically between August 5th and November 5th, 2025. This timeframe was selected to capture data from a period characterized by stabilized process flow and maximal adoption of the automation artifact.

Table 6. Post-intervention DF per release type.

Metric	Overall	RC approval required	No RC approval required	Overall average per week
PR merges	1595	76	1519	116
PR deployments	1308	72	1236	100

As reported in Table 6, the average number of PR merges during the measurement period was 116 per week, resulting in an average of 100 PR deployments per week.

A crucial observation relates to the volume of deployments requiring the former RC gating approval step, which was the focus of the organizational change. Only 4.8% (76 out of 1,595) of all PR merges required RC approval, and this resulted in 5.5% (72 out of 1,308) of the total deployments. This demonstrates that an overwhelming majority of PR merges and deployments have bypassed the centralized approval gate.

The observed trend validates the major organizational shift toward removing the RC gating bottleneck, thereby confirming that the automation artifact successfully facilitated the shift to a decentralized, continuous flow model.

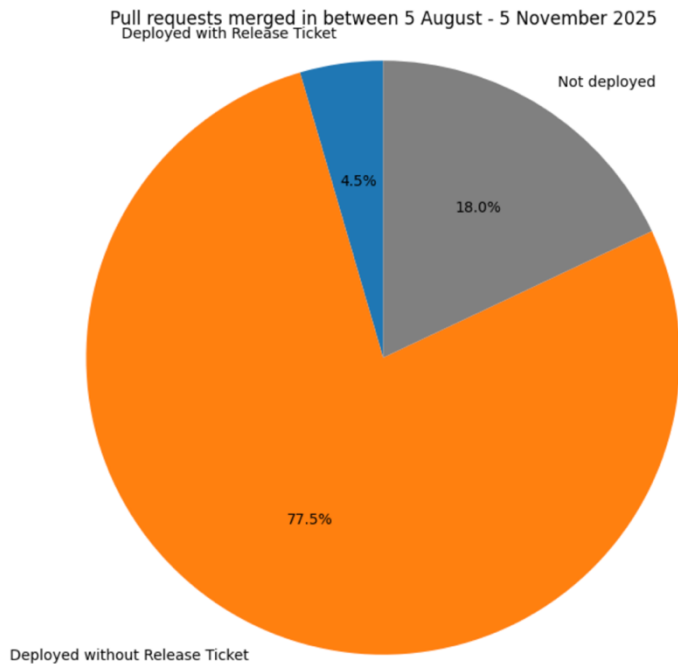


Figure 22: Segmentation of PR merges deployed from 5th August 2025 to 5th November 2025.

The composition of the throughput metrics is further illustrated in Figure 22, which visualizes the distribution of PR merges based on deployment outcome.

This visual representation highlights two key compositional findings from the data presented in Table 6. First, 18% of all PR merges did not result in a deployable artifact, primarily due to changes related to testing infrastructure or pipeline configurations. This characteristic is common in continuous integration pipelines and is filtered out when calculating the DORA LTC. Second, the overwhelming majority of merges and resulting deployments (95% and 94.5% respectively) did not require the RC approval.

To maintain methodological consistency with the DORA definition of DF, which is the rate of successful releases to production, this paper utilizes the frequency of successful deployments per week.

Consequently, the post-intervention DF is established at 100 deployments per week, with only 5.5% of these successful deployments requiring the vestigial RC approval gate. This concentration of deployments outside of the legacy gate underscores the success of the organizational change in decentralizing governance

5.1.2 Lead Time for Changes

The post-intervention LTC data were collected using the identical tools and data aggregation pipeline established in the Methodology (see Section 3.2.1). To accurately isolate residual bottlenecks, the metric retains the same breakdown of two segments: PR Approval to Merge (L1) and PR Merge to Deployment (L2). Furthermore, L2 is similarly sub-segmented to identify the specific period of “Production Deployment Wait Time” (L2b).

These metrics were measured over the three-month observation period, from 5 August to 5 November 2025. The results are subsequently compared against the baseline quantitative metrics reported in the Results (see Section 4.1.3).

To begin with, Figure 23 illustrates the post-intervention distribution of the L1 metric, representing the time from PR approval to merge.

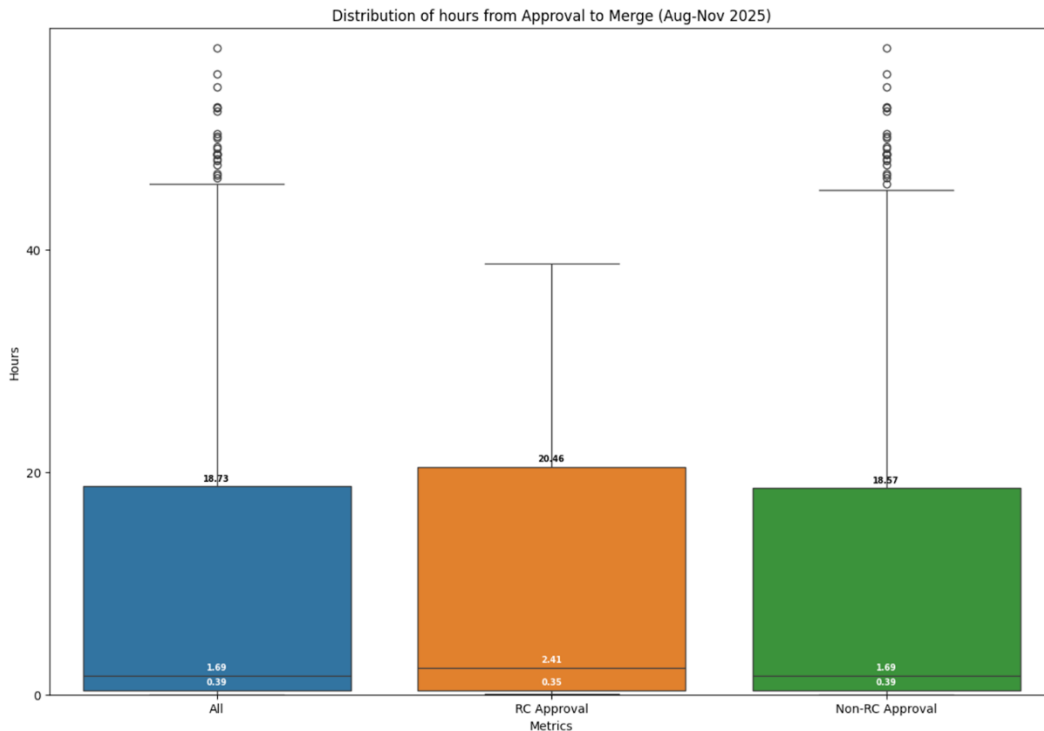


Figure 23: Observation period distribution of hours between PR merge and approval.

As Figure 23 illustrates, the overall median time for all approved PRs to be merged was 1.7 hours. PRs that required RC approval were merged within 2.4 hours, and PRs that required no RC approval were merged in 1.7 hours. This wait time may be attributed to discretionary factors, such as developers synchronizing deployments with customer release announcements, adhering to established deployment windows, or addressing minor supplementary changes after initial approval.

Nevertheless, the post-intervention median of 1.7 hours is sufficiently small, suggesting that this stage no longer represents a significant performance bottleneck requiring further investigation at this performance tier. Therefore, the first segment of the LTC equation is established at 1.7 hours.

The post-intervention performance of the PR merge to deployment segment (L2), which includes automated processing time and the production deployment wait time, is illustrated in Figure 24.

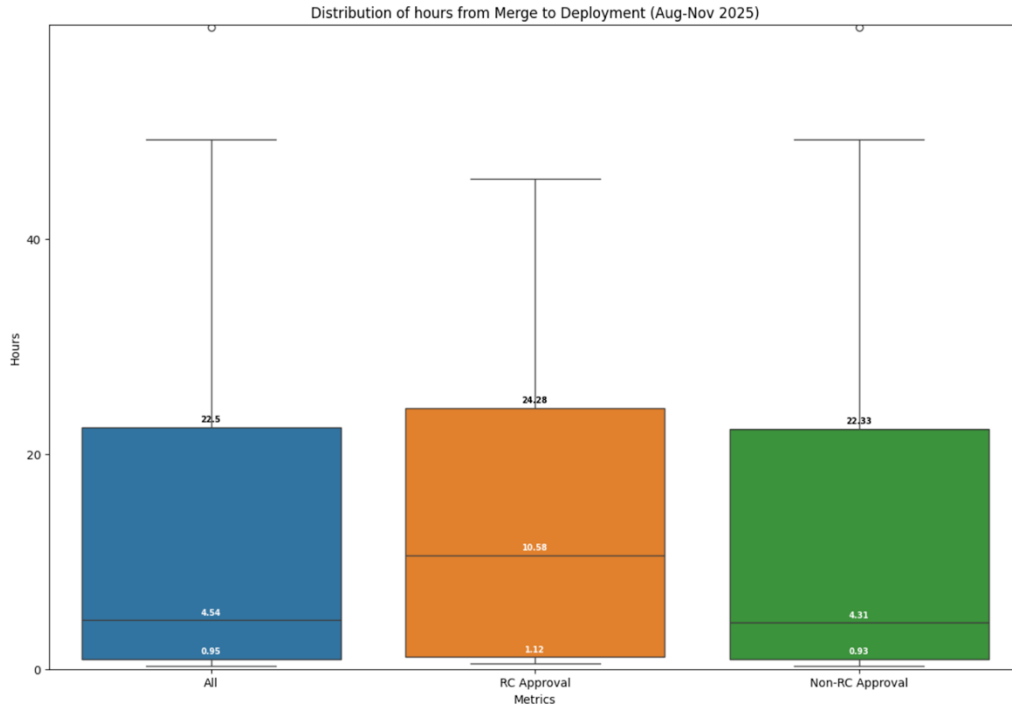


Figure 24: Observation period distribution of time between PR merge and deployment to production.

Over the observation period, the overall median time from merge to production was 4.5 hours. Deployments requiring RC approval took 10.6 hours, and deployments without RC approval took 4.3 hours. Consistent with the baseline analysis, it is critical to determine the extent to which the L2 duration is composed of automated pipeline execution time versus the time spent awaiting the final manual production deployment trigger. The median pipeline run time is illustrated across all release types in Figure 25.

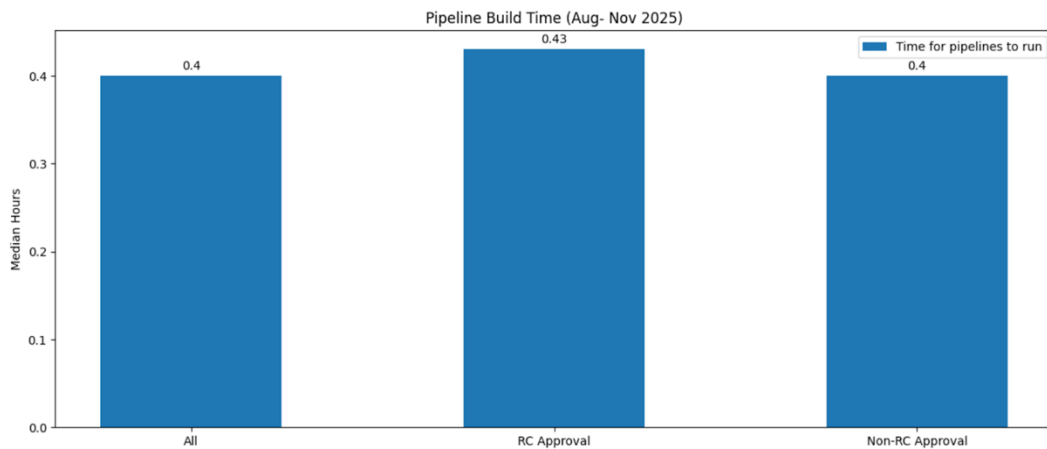


Figure 25: Observation period comparison of pipeline run time across release types.

Similar to the baseline metrics, the pipelines took less than an hour to run across all PRs. This demonstrates that the pipeline execution time constitutes a negligibly small portion of the overall time between PR merge and deployment.

Delving deeper into the final component of the *L2* segment, the wait time for production deployment (*L2b*), Figure 26 illustrates this breakdown across the different release types.

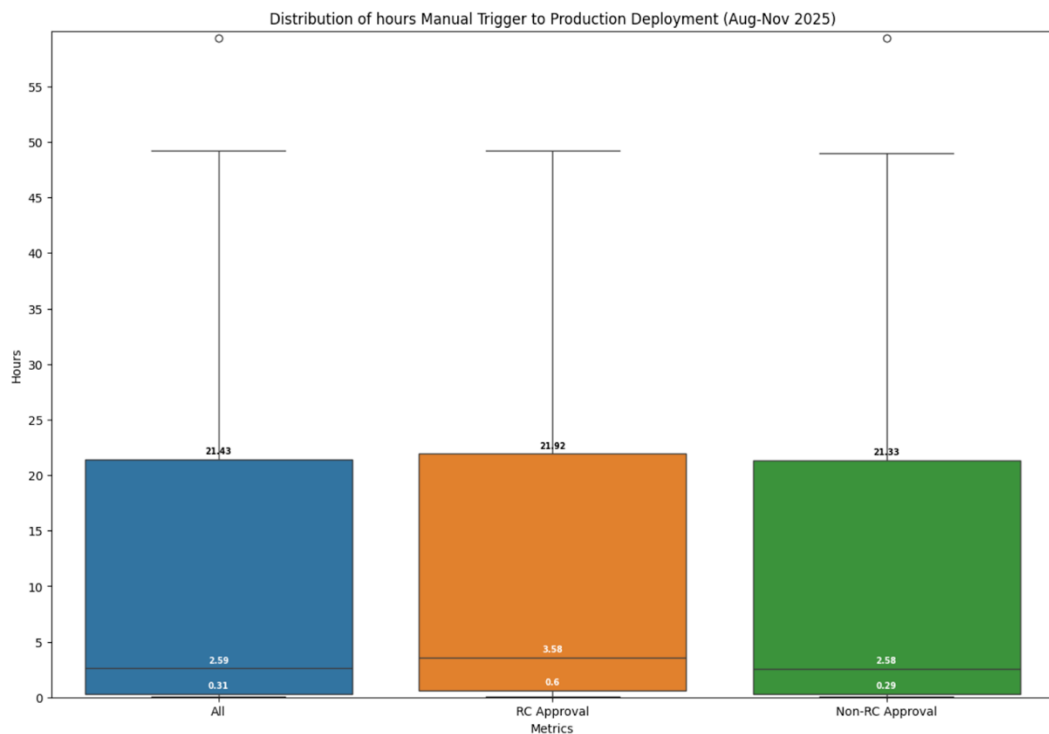


Figure 26: Observation period distribution of time between manual trigger and production deployment across release types.

As shown in Figure 26, PRs not requiring RC approval showed a marginally shorter wait time for production deployment (2.6 hours) compared to RC-required PRs (3.6 hours). This minor difference suggests that the vestigial RC gating process, though infrequent, still introduces a small residual queuing delay compared to the fully autonomous path.

Table 7. Observation period LTC breakdown per release type.

	Overall (hours)	RC approval required (hours)	No RC approval required (hours)
L1: Approval to Merge	1.7	2.4	1.7
L2: Merge to Deployment	4.5	10.6	4.3
L2b: Production Deployment Wait Time	(2.6)	(3.6)	(2.6)
Total LTC	6.2	13	6.0

The LTC distribution is reported in Table 7, where the total LTC for all PRs combined was 6.2 hours in the three-month time window between August 5th and November 5th, 2025.

5.1.3 Comparison of Performance Metrics

The intervention resulted in a statistically significant shift in the organization's delivery performance, moving the capabilities closer to the industry's high and elite performance tiers. The data presented in Table 8 illustrates the dramatic reduction in change delivery time and the corresponding increase in operational cadence.

Table 8. Comparison between baseline and validation metrics.

Metric	Baseline	Validation	Improvement
DF	47 releases/week	100 releases/week	+ 213%
LTC	20.2 hours	6.2 hours	- 69%
Production Deployment Wait Time	10.0 hours	2.6 hours	- 74%

The LTC experienced a significant 69% reduction, decreasing from 20.2 hours to just 6.2 hours. Crucially, the production deployment wait time (L2b) saw an even more pronounced decrease of 74%, dropping from a median of 10.0 hours to 2.6 hours. This drastic elimination of latency in the L2b segment confirms the successful removal of the manual handoffs and

organizational queue time, validating the core hypothesis that the manual approval gate was the primary bottleneck to delivery speed.

Furthermore, the Deployment Frequency (DF) more than doubled (213% improvement), increasing from 47 to 100 releases per week, confirming that the removal of this centralized friction points enabled teams to adopt smaller batch sizes and deploy their independent micro-frontends with greater agility.

While quantitative data illustrates the speed at which updates are coded and shipped into production, qualitative data is necessary to complement and contextualize the experiences of various stakeholders involved with the release process. The next section will cover the qualitative data gathered through interviewing several candidates within the company.

5.2 Qualitative Analysis

The qualitative analysis sought to understand the experiential and cultural shift resulting from the intervention, thereby providing the cultural context for the observed quantitative improvements.

Data was gathered through semi structured interviews with key organizational stakeholders. The original set of participants considered for observation period analysis mirrored the six roles interviewed for the baseline assessment (see Section 3.2.2), which were the feature team developer, Scrum Master, Program Manager, Release Manager, Product Owner and QA Engineer.

However, due to broader organizational realignment during the study period, the Scrum Master, Product Owner, and QA Engineer roles were eliminated and thus required no further interviews. Accordingly, the final qualitative data was collected solely from the feature team developer, Release Manager, and Program Manager, as these roles retained direct operational or governance oversight of the new process.

The findings gathered from these interviews confirm that the new artifact successfully addressed organizational pain points related to friction, visibility, and organizational dependency.

1. Elimination of Process Friction

The most significant qualitative finding across the developer demographic was the total elimination of redundant, nonvalue add work, directly supporting the 69% reduction in LTC.

Interviews with a feature developer confirmed that prior practices, such as "filling out release fields by hand" and "checking pipelines manually before updating Jira," were highly time consuming and introduced unnecessary overhead.

The intervention fundamentally addressed this friction by eliminating the manual ticketing gate entirely. The new process shifts the developer's focus back to code delivery, enabling them to "avoid manually creating and updating Jira release tickets" and effectively moving away from a complex Jira based system. This transfer of responsibility from a human gatekeeper to automated checks fosters the enhanced autonomy necessary for high performing teams.

2. **Real Time Visibility and Operational Stability**

The qualitative data confirmed that the interventions dramatically improved the utility of release information for incident response and operational monitoring. Before the intervention, engineers and stakeholders were forced to rely on "back and forth communication" with PMs or manual checking to confirm the latest deployed code.

The new deployments registry and notification system resolved this by providing real time, asynchronous communication. As a developer noted, "Slack alerts show every RC/Prod deploy in real time," eliminating the need to manually check the pipeline status.

The new release board provides the necessary audit trail to "pinpoint the release that triggered an incident without chasing the PM." The Release Manager confirmed that their primary intended use of the new application is to monitor the queue for any "blocked" releases and "checking releases that went to Prod or RC in case of incidents," validating the tool's use case for operational stability and release management.

3. **Organizational Alignment and Scope Restriction**

The interventions' impact on reducing organizational friction aligned with broader strategic shifts in the department's structure. While the organizational removal of the Scrum Master, Product Owner, and QA roles was a separate, pre-determined strategic change that occurred during the study period, the successful implementation of the artifact complemented this change by immediately reducing the administrative workload for remaining stakeholders.

Specifically, the data confirms that roles such as Product Manager and Technical Program Manager no longer need to use the new release process for upward reporting. This outcome is a validation of the study's success, as it eliminated the need for these roles to be manually involved in or track the flow of every individual micro-frontend deployment.

The observation that Program Managers do not track releases anymore confirms that the necessary information is now disseminated automatically and asynchronously, thereby removing a centralized dependency and allowing these roles to focus on high value strategic work rather than process administration.

6 Discussion

This chapter interprets the results collected during the Observation phase of the Action Research cycle (Chapter 4) against the theoretical and empirical foundation established in the Literature Review (Chapter 2). The primary purpose is to assess the degree to which the automated governance artifact succeeded in addressing the core research problem: the acceleration of the software delivery process by eliminating vestigial manual governance gates within a decomposed micro-frontend architecture.

The discussion systematically synthesizes the quantitative performance metrics with the contextual insights gathered through qualitative interviews, demonstrating the direct relationship between the architectural intervention and subsequent organizational performance gains. The full data analysis is presented in Chapter 5.

6.1 Summary

The implementation of the automated governance artifact yielded statistically significant and substantial improvements across the core DORA throughput metrics at the DIP organization, directly confirming the study's hypothesis.

The intervention resulted in dramatic performance improvements that align the organization's delivery capabilities in the higher band of the elite performance tier, as presented in Section 5.1.3.

The reduction of LTC from 20.2 hours to 6.2 hours signifies the successful removal of the manual ticketing gate, further validated by the drop in production deployment wait time from 10.0 hours to just 2.6 hours, which was previously identified as the central bottleneck preventing continuous flow. Simultaneously, the 213% increase of DF from 47 to 100 releases per week validates that the elimination of friction enabled teams to adopt smaller batch sizes and deploy their independent micro-frontends more routinely, fulfilling the core promise of the CD model [1]. The qualitative data confirmed the mechanisms behind these gains:

1. **Increased Visibility:** The automated release dashboard and real time Slack notifications provided a single source of truth for deployment status, eliminating the need for synchronous status checks and increasing organizational confidence in the process.
2. **Enhanced Autonomy:** Development teams reported a significant increase in ownership and autonomy, as they could initiate production releases immediately upon passing automated checks, rather than

waiting for an external, centralized approval process.

3. **Process Compliance through Automation:** The enforcement of change log capture directly in the PR workflow ensured that the governance requirement was met programmatically, substituting the unreliable and high friction manual check with a low friction, high compliance automated check.

These findings collectively support the conclusion that the automated artifact successfully translated the technical benefits of the micro-frontend architecture into measurable organizational performance gains.

6.1.1 Cultural Shift

The deployment of the technical artifact alone did not guarantee success. Its organizational wide adoption required a significant, high-level organizational realignment, confirming that process improvement is inherently a socio-technical challenge.

Initial adoption was hindered by cultural resistance from Program Managers and stakeholders who preferred established, risk-averse, manual approval processes. This reluctance was rooted in an organizational structure where accountability for production stability rested with administrative roles (Product Managers and QA Engineers) that lacked the authority to fix or deploy quickly.

The eventual, organization-wide rollout was driven by three strategic, structural changes:

1. The formal removal of the Scrum Master, Product Owner, and QA roles structurally dismantled the manual gatekeeping apparatus. Production stability responsibility was formally shifted to feature development teams.
2. This shift created a critical impedance mismatch where the newly accountable development teams were denied the necessary operational autonomy by the lingering manual approval requirement.
3. The automated governance artifact became the mandatory tool required to align high accountability with high autonomy, replacing manual approval with automated checks, real-time visibility, and an immutable audit trail. This was the key reason for the final organizational mandate.

This transition effectively moved the organization from governance by friction to governance by transparency. Stakeholder trust was successfully

shifted from reliance on human gatekeepers to reliance on the objective, real-time data provided by the Deployment Registry App and the asynchronous Slack notification system.

The qualitative data strongly supports this interpretation, as interviewed feature developers explicitly noted the elimination of "redundant, nonvalue add work" such as "filling out release fields by hand" and avoiding a "complex Jira based system."

Furthermore, the observation that roles like Program Manager, Technical Program Manager, and Engineering Manager no longer needed to track individual deployments for upward reporting confirms the successful elimination of a centralized administrative dependency, allowing these roles to focus on high-value strategic work.

6.2 Comparison with Industry Benchmarks and Literature

The measured improvements position the DIP organization firmly within the elite performance tier, demonstrating a successful and immediate alignment with the empirical findings of the DORA research [27].

The observed changes confirm that the automated governance artifact functioned precisely as a mechanism for translating architectural independence (see Section 2.4) into demonstrable delivery speed, fulfilling the requirements of CD [1].

Deployment Frequency (DF)

The 213% increase of the DF from 47 to 100 releases per week provides empirical validation for the procedural necessity of small, decoupled batch sizes [1]. The original DF was suppressed by the centralized release approval bottleneck, which forced development teams to accumulate changes before submitting a manual request.

By removing this centralized constraint, the automated artifact allowed the organization to:

- Reduce batch size since teams no longer felt penalized for submitting a small change, thereby deploying their individual components more frequently.
- Increase throughput, as evident by the 213% increase in LTC which confirms the organizational capacity to consume the efficiency of the underlying micro-frontend architecture.

This outcome demonstrates the direct link between removing organizational friction and achieving high performance. The organization's ability to deploy 100 times per week now aligns it with the operational tempo of an

elite performer, confirming the theory that high DF is a consequence of reducing friction, not just a goal.

Lead Time for Changes (LTC)

The baseline median LTC of 20.2 hours formally positioned the organization within the DORA elite performance threshold of less than one day [27]. However, this duration, which required over half a working day, often resulted in code readiness occurring on one day and production deployment being deferred until the subsequent day, which placed the organization within the high performance tier LTC of between 1 to 7 days [27].

Following the intervention, the LTC realized a significant 69.0% reduction, decreasing to 6.2 hours. This reduction signifies a qualitative shift from operating at the high performer tier to establishing a band deep within the elite status, effectively ensuring continuous same day delivery of all changes. This improvement directly correlates with the effective elimination of approximately 13.8 hours of organizational waiting time.

By achieving an LTC of 6.2 hours, the organization has validated the theoretical premise that removing organizational friction directly translates to superior delivery speed. Consequently, the residual time is now predominantly accounted for by the fixed, necessary duration of automated processes, including compilation, automated testing, and execution within defined deployment windows, successfully shifting the primary operational bottleneck from a manually coordinated process to technical efficiency within the CI/CD pipeline.

6.2.1 Relationship to Continuous Delivery Principles

The findings of this study directly support the core prescriptive principles outlined in the foundational literature on CD [1]. The automated governance artifact functioned as the technical mechanism required to enforce these principles within a legacy organizational structure.

Firstly, the “Automate Almost Everything” principle was addressed by automating the release ticketing, approval, and notification processes. This specifically eliminated the redundant manual work that contributed to the 13.8 hour queue time reducing human intervention to a minimum.

Secondly, the “Create a Repeatable, Reliable Process for Releasing Software” principle was addressed by replacing the variable, high friction manual gate with a programmatic, data driven system, i.e., the frontend releases dashboard, the thesis ensured the release process became predictable and repeatable. The system’s governance model shifted from subjective human approval to objective, automated data checks.

Thirdly, the “If It Hurts, Do It More Frequently, and Bring the Pain Forward” principle adoption was empirically validated with the resulting 213%

increase in DF. The removal of the central approval bottleneck allowed teams to adopt smaller, less painful batch sizes, making the act of deploying a routine, frequent event rather than a deferred, risky one.

Fourthly, the “Keep Everything in Version Control” principle was reinforced by the artifact's non-functional requirement to mandate the 'user platform value', i.e., the human readable release note, within the PR description. This approach linked the essential release metadata to the immutable merge commit hash via the PR metadata. This provided the necessary traceability and eliminated reliance on external release documentation, upholding the spirit of version control for all release artifacts.

Finally, the “Everybody Is Responsible for the Delivery Process” principle was enacted through the strategic organizational shift. By structurally eliminating dedicated gatekeeping roles such as, Scrum Master and QA, and transferring full accountability to the feature development teams, the organization embraced the concept that deployment responsibilities should be shared by everyone. The artifact provided the teams with autonomous tooling necessary to manage this new level of accountability.

6.3 Addressing Research Questions

The findings from this study directly address the primary research questions concerning the implementation of automated software delivery within a micro-application architecture. The results are summarized across the three key research questions presented at the beginning of this thesis.

RQ1. How can the process of shipping features be streamlined without relying on release tickets?

The implemented solution, which was a centralized, automated release tracking dashboard, mechanized the visibility layer of the delivery process. This system served as the single source of truth for deployment and status information across the organization, effectively eliminating the need for manual ticketing. The implementation demonstrated that in high velocity micro-application environments, real-time automation is a viable and superior alternative to sequential manual tracking methods, significantly reducing inherent process latency and risk.

RQ2. What are the key challenges and benefits of automating the release process within a micro-application architecture?

Automation yielded significant and measurable benefits, including streamlined workflows and a demonstrable improvement in overall efficiency and delivery speed. Conversely, the transition exposed a major critical challenge, organizational adoption. Achieving sustained success necessitated dedicated

effort in change management to ensure consistent stakeholder adoption and to overcome resistance to altering established, manual processes. These findings suggest that while the technical benefits of automation are readily achieved, the organizational and integration complexities present the primary barriers to successful rapid transformation.

6.4 Implications

The primary implication of this study is that investment in automated governance tools is most impactful when coupled with strategic organizational restructuring that aligns accountability and autonomy.

The organizational decision to remove certain roles created the necessary pressure for adoption, while the artifact provided the practical solution. The organization has successfully shifted its focus from managing process compliance to optimizing technical efficiency, as residual LTC is now dominated by technical pipeline runtime and timing decisions driven by developers. Future efforts should therefore focus on improving build pipeline speed to address the remaining bottleneck.

A key practical implication for organizations transitioning from manual processes to streamlined automation is the mandatory socio-technical alignment. Any technical solution must be explicitly coupled with strategic organizational changes that grant development teams complete autonomy and ownership over the quality and reliability of their work. This involves fostering cultural changes to increase ownership within the development teams. Furthermore, the technical automation must be designed to simplify the release owner's responsibilities, utilizing reliable automated data streams directly from the build pipelines and instrumenting a comprehensive notification system to ensure continuous visibility and reduce cognitive load.

6.5 Conclusion

The implementation of the automated governance artifact, timed with a fundamental organizational shift in accountability, resulted in a rapid and substantial improvement in software delivery performance.

The study demonstrates that successfully moving an organization from a high performing to an elite performing throughput tier requires not only the introduction of technical automation but also the decisive dismantling of organizational structures that enforce manual governance. Replacing organizational gatekeepers with objective, real-time data and an asynchronous notification system, allowed the organization to shift its focus from managing compliance to optimizing technical flow. The resulting process governed by transparency empowered development teams, dramatically reduced lead time, and positioned the organization to sustain a high cadence CD flow.

References

- [1] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, 2011.
- [2] S. Nerur, R. Mahapatra and G. Mangalaraj, Challenges of migrating to agile methodologies. *Communications of the ACM.*, 2005.
- [3] W. Royce, Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th international conference on Software Engineering*, 1987, pp. 328-338.
- [4] M. Pace, “A Correlational Study on Project Management Methodology and Project Success,” *EPPM-Journal*, 2019.
- [5] Hoek, d. André van, R. Hall, D. Heimbigner, Wolf and L. Alexander, “Software release management,” *ESEC '97/FSE-5*, 1997.
- [6] T. J. Gandomani and M. Z. Nafchi, “Agile transition and adoption humanrelated challenges and issues: A grounded theory approach,” *Comput. Hum. Behav.*, vol. 62, p. 257–266, 2016.
- [7] T. Natarajan and S. Pichai, “Transition From Waterfall to Agile Methodology: An Action Research Study,” *IEEE Access*, vol. 12, pp. 49341-49362, 2024.
- [8] S. Nerur and V. Balijepally, “Theoretical reflections on agile development methodologies,” *Commun. ACM*, vol. 50, no. 3, p. 79–83, March 2007.
- [9] S. Obrutsky and E. Erturk, “The agile transition in software development companies: The most common barriers and how to overcome them,” *Bus. Manage. Res.*, vol. 6, no. 4, p. 40, November 2017.
- [10] S. Campanelli, D. Bassi and F. S. Parreiras, “Agile transformation success factors: A practitioner’s survey,” *Proc. Adv. Inf. Syst. Eng. Cham*, p. 364–379., 2017.
- [11] D. Naslund and R. Kale, “Is agile the latest management fad? A review of success factors of agile transformations,” *Int. J. Quality Service Sci.*, vol. 12, no. 4, p. 489–504, Oct 2020.
- [12] L. Lwakatare, P. Kuvaja and M. Oivo, “Relationship of DevOps to Agile, Lean and Continuous Deployment. In Product-Focused Software Process Improvement,” *Lecture Notes in Computer Science*, p. 399–415, 2016.

- [13] C. Wang and C. Liu, “Adopting DevOps in Agile: Challenges and Solutions. Adopting DevOps in Agile: Challenges and Solutions.,” 29 June 2018. [Online]. Available: <http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1228684&dswid=5071>. [Accessed 28 September 2025].
- [14] E. Dörnenburg, “The Path to DevOps,” *IEEE Soft*, no. 35, p. 71–75, 2018.
- [15] J. Faustino, R. Pereira, B. Alturas and M. Silva, “Agile Information Technology Service Management with DevOps: An Incident Management Case Study,” *Int. J. Agile Syst. Manag.*, no. 13, p. 339–389, 2020.
- [16] S. Galup, R. Dattero and J. Quan, “What Do Agile, Lean, and ITIL Mean to DevOps?,” *Com. ACM*, no. 63, p. 48–53, 2020.
- [17] D. Céspedes, P. Angeleri, K. Melendez and A. Dávila, “Software Product Quality in DevOps Contexts: A Systematic Literature Review. In Trends and Applications in Software Engineering,” *Advances in Intelligent Systems and Computing*, p. 51–64, 2020.
- [18] Almeida, Fernando, J. Simões and S. Lopes, “Exploring the Benefits of Combining DevOps and Agile,” *Future Internet 14*, no. 2, p. 63, 2022.
- [19] W. Luz, G. Pinto and R. Bonifácio, “Adopting DevOps in the real world: A theory, a model, and a case study,” *J. Syst. Soft*, no. 157, 2019.
- [20] D. Venugopal, “DevOps: Driving Innovation with Old Habits.,” 1 September 2020. [Online]. Available: <https://devops.com/devops-driving-innovation-with-old-habits/>. [Accessed 28 September 2025].
- [21] K. Nybom, J. Smeds and I. Porres, “On the Impact of Mixing Responsibilities Between Devs and Ops. In Agile Processes, in Software Engineering, and Extreme Programming,” *Lecture Notes in Business Information Processing*, p. 131–143, 2016.
- [22] Amazon, “The Story of Apollo - Amazon’s Deployment Engine. All Things Distributed,” 2014. [Online]. Available: <https://www.allthingsdistributed.com/2014/11/apollo-amazon-deployment-engine.html>. [Accessed 28 September 2025].

- [23] G. Kim, J. Humble, P. Debois and J. Willis, *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*, IT Revolution Press, 2016.
- [24] N. Forsgren, J. Humble and G. Kim, *Accelerate: the science behind DevOps: building and scaling high performing technology organizations*, Portland, Oregon: IT Revolution, 2018.
- [25] Google, “DORA | Research.,” [Online]. Available: <https://dora.dev/research>. [Accessed 8 November 2025].
- [26] Puppet & (DORA) DevOps Research and Assessment. “State of DevOps Report,” Puppet, Inc., Portland, OR, 2017.
- [27] Puppet & (DORA) DevOps Research and Assessment. “State of DevOps Report 2024.,” Puppet, Inc. , Portland, OR, 2024.
- [28] J. Lewis and M. Fowler, “Microservices in a Nutshell,” 27 June 2014. [Online]. Available: <https://www.thoughtworks.com/insights/blog/microservices-nutshell..> [Accessed 3 November 2025].
- [29] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, O'Reilly Media, 2015.
- [30] D. Coghlan and T. Brannick, *Doing Action Research in Your Own Organization* (4th ed.), Sage Publications, 2014.
- [31] K. Lewin, “Action Research and Minority Problems,” *Journal of Social Issues*, p. 34–46, 1946.
- [32] Atlassian, “The Forge platform,” [Online]. Available: <https://developer.atlassian.com/platform/forge/introduction/the-forge-platform/>. [Accessed 8 November 2025].
- [33] Atlassian, “The BitBucket Cloud REST API.,” [Online]. Available: <https://developer.atlassian.com/cloud/bitbucket/rest/api-group-pullrequests>. [Accessed 08 November 2025].