Tampere University

Safwane Benbba

Streamlining CI/CD Processes: A Comparative Analysis and Migration Strategy from Jenkins to Bitbucket Pipelines

Faculty of Information Technology and Communication Sciences (ITC)
Master's thesis
November 2023

Abstract

Safwane Benbba: Streamlining CI/CD Processes: A Comparative Analysis and Migration Strategy from Jenkins to Bitbucket Pipelines
Master's thesis
Tampere University
Master's Degree Programme in Software Development
November 2023

This thesis presents a case study that conducts a comparative analysis between Jenkins and Bitbucket Pipelines by replicating and examining a CI/CD deployment pipeline for an IoT platform on AWS. The technical features, cost implications, and performance metrics are assessed. It is found that Jenkins offers lower direct computational costs but incurs significant personnel expenses due to the requirement of infrastructure maintenance. In contrast, Bitbucket Pipelines, though higher in computational cost, is observed to remove maintenance overhead by virtue of its managed service structure.

In terms of features, it is highlighted that Bitbucket Pipelines lacks support for dynamic step execution and possesses limited parallelization capabilities, potentially impacting the scalability and flexibility in more complex deployment processes. The performance evaluation indicates that Bitbucket Pipelines has an advantage in deployment times but also faces limitations due to memory constraints for specific tasks.

It is concluded in the thesis that Bitbucket Pipelines offers reduced operational complexity and quicker deployment times, yet organizations are advised to consider the trade-offs in features and memory requirements against the cost benefits when contemplating a shift from Jenkins to Bitbucket Pipelines for their cloud-based CI/CD operations.

Keywords: DevOps, CI/CD, Jenkins, Bitbucket Pipelines

Preface

I would like my colleagues at Kalmar for their assistance with the practical work of this thesis. Special thanks are owed to Juha Uola for his support and expertise during the entire process. My appreciation also extends to professor Kari Systä for his guidance on thesis writing and his swift reviews of my drafts.

I am equally thankful to my friends and family for their unwavering support, particularly during the most challenging phases of writing this thesis.

Lastly, I wish to acknowledge ChatGPT for its assistance in organizing my thoughts and refining the text to its present form.

Tampere, 9th November 2023

Safwane Benbba

Contents

1 Introduction

Continuous Integration (CI) and Continuous Deployment/Delivery (CD) have become fundamental components of modern software development. Their primary objective is to streamline the process of integrating code changes and ensuring that software is efficiently deployed. With numerous CI/CD tools available in the market, making an informed decision becomes imperative. This thesis presents an in-depth comparison of two such prevalent platforms: Jenkins and Bitbucket Pipelines.

The comparison is multi-faceted, evaluating both platforms according to their features, benefits, and limitations. Emphasis is placed on the real-world implementation and challenges that developers might encounter during the integration and deployment processes.

Beyond the functional and operational differences between Jenkins and Bitbucket Pipelines, the economic implications of adopting one over the other are also examined. A preliminary cost analysis is provided, breaking down the expenditures associated with infrastructure as well as the potential overhead introduced by developer interactions and maintenance of the CI/CD systems.

At a time when CI/CD practices are being rapidly adopted and integrated into software development workflows, having clarity on the functionalities and cost structures of these tools is necessary. This thesis seeks to bridge the knowledge gap, offering a comprehensive understanding of Jenkins and Bitbucket Pipelines, and shedding light on the associated operational and financial considerations.

1.1 Background and Motivation

In recent years, there has been a significant evolution in the tools and platforms designed to support and enhance the Continuous Integration and Continuous Deployment (CI/CD) processes in software development. Jenkins, a stalwart in this domain, has been extensively used at Kalmar SST Digital Solutions unit in Cargotec (SST Digi) for its flexibility and wide range of plugins. It offers an open-source platform, enabling developers to tailor their CI/CD pipelines to precise requirements.

However, as with any technology, the landscape is continuously changing. More cloud-native, managed solutions have emerged, offering potential benefits in terms of reduced overhead, scalability, and integration capabilities. Bitbucket Pipelines represents one of these cloud-native solutions. As part of the Atlassian suite, it promises seamless integration with other Atlassian products and presents a different, more streamlined paradigm for CI/CD compared to traditional platforms like Jenkins.

The primary motivation behind this thesis was to dissect the practical and financial implications of transitioning from Jenkins to Bitbucket Pipelines, using the CI/CD setup at SST Digi as a case study. The objective was not just to contrast their functionalities but also to understand the challenges, costs, and potential advantages involved in such a migration. With organizations constantly seeking ways to optimize their development pipelines, both in terms of efficiency and cost, this analysis aims to shed light on whether such a transition is justifiable and what pitfalls and considerations should be anticipated.

1.2 Research question and objectives

This thesis sets out clear research objectives to support its core motivation: to examine the differences between Jenkins and BitBucket Pipelines and to develop a migration plan for transitioning the deployment pipelines of Kalmar's IoT platform, which powers Kalmar Insight [1]. The research objectives are as follows:

1. Conduct a comprehensive comparison between Jenkins and BitBucket Pipelines, considering various aspects such as features, functionality, scalability, performance, security, compliance, ease of use, maintenance, integration, ecosystem support, and cost. This comparison will help organisations make informed decisions when choosing a CI/CD tool that best suits their requirements. [2]

2. Develop a migration strategy and plan for transitioning an IoT platform's deployment pipeline from Jenkins to BitBucket Pipelines. This objective includes identifying and addressing potential challenges associated with the migration process, such as configuration management, compatibility issues, and integration with existing tools and platforms. It is essential to develop a comprehensive migration plan that minimises disruptions and ensures a smooth transition between the two CI/CD platforms. [3]

3. Identify and analyse the potential benefits and challenges associated with migrating deployment pipelines from Jenkins to BitBucket Pipelines. This analysis will help the organisation better understand the implications of such a transition and prepare for potential obstacles. Additionally, the identification of best practices will enable organizations to adopt strategies that improve the efficiency and effectiveness of their CI/CD processes [4].

4. Evaluate the migration process in terms of performance improvements, cost reductions, security enhancements, and usability and maintenance improvements. This objective aims to assess the success of the migration strategy and plan, and to identify areas of improvement for future migrations [5].

5. Contribute to the body of knowledge on CI/CD tools and migration processes, providing valuable insights and recommendations for other organizations considering a similar transition from Jenkins to BitBucket Pipelines. This research will help enrich the existing literature on CI/CD practices and tools, and provide practical guidance for organizations facing similar challenges.

1.3 Thesis structure

This thesis begins with a literature review in Chapter 2, providing a foundational understanding of Continuous Integration and Continuous Deployment (CI/CD) principles, with a focus on Jenkins and Bitbucket Pipelines. After a brief explanation of the methodology in chapter 3, chapter 4 lays the groundwork for the comparison, evaluating both systems from a broad perspective. Chapter 5 details the existing Jenkins architecture in use at SST Digi, setting the stage for a later contrast with Bitbucket Pipelines. The migration process itself is outlined in Chapter 6, documenting the transition of the IoT platform's deployment pipeline to Bitbucket Pipelines and discussing the challenges and solutions found along the way. Chapter 7 provides an overarching evaluation of the migration's outcome, analyzing costs, features, and performance in relation to the original Jenkins setup. The thesis concludes with Chapter 8, reflecting on the findings and suggesting avenues for future work.

2 Literature Review

This chapter presents a focused overview of Continuous Integration and Continuous Deployment (CI/CD) within the framework of DevOps. It presents the foundational concepts, traces their practical applications, and introduces Jenkins and Bitbucket Pipelines—two significant CI/CD tools. This chapter examines the features, usability, and integration capabilities of both tools, drawing from a range of academic studies and online documentation. The aim is to establish a clear understanding of each platform's capabilities to inform the later discussion on migrating from Jenkins to Bitbucket Pipelines.

2.1 Continuous Integration and Continuous Deployment (CI/CD)

Continuous Integration (CI) and Continuous Deployment (CD) are fundamental methodologies in modern software development, enhancing the speed, reliability, and resilience of the software development lifecycle.

CI is a practice in which developers integrate their code changes into a central repository, often multiple times per day. This integration is then validated by an automated build and test process, which facilitates the early identification of integration issues, reducing their occurrence and supporting rapid software development [4]. Research from Vasilescu et al. provides evidence that teams incorporating CI practices show marked improvements in merging pull requests, particularly those originating from core members. Their research suggests a positive correlation between the utilization of CI and a reduction in rejected pull requests from external contributors [6]. Additionally, core developers in these CI-utilizing teams have been observed to identify a significantly greater number of bugs compared to teams without CI. Interestingly, this increased bug detection does not compromise external software quality, as there's no corresponding increase in defects identified by external contributors [6]. While CI focuses on early integration and validation, Continuous Deployment (CD) extends this process to automate the deployment of changes into production environments.

Continuous Integration (CI) involves the regular merging of code changes into a shared repository, which is not limited to automated testing but also includes building and integrating new features. Continuous Deployment (CD), building upon CI, ensures the automated deployment of these changes into production environments. CD offers immediate feedback on the functionality and performance of the software in live settings. By enabling the frequent rollout of updates and enhancements, CD helps to improve customer satisfaction and maintain a competitive advantage

[2]. Chen's work [7] adds to this understanding, linking CD with increased software reliability and fewer late-stage defects. Further insights from Chen's study reveal substantial improvements in the CD process, leading to benefits such as significant enhancements in productivity and efficiency, with developers and testers spending less time on environment setup. Operations engineers are now able to release applications to production with a simple click, and the automated CD pipeline has minimized issues related to old release practices. The CD pipeline has also reduced the number of code changes per release, aiding in easier identification and resolution of problems, and even providing an automatic rollback feature to reduce the risk of release failure. Engineers have experienced reduced stress on release day, transforming it into just another normal day. Furthermore, the transition to CD has fostered trust between the users' department and software development teams, mending relationships that were previously strained due to quality and release issues [7].

However, implementing CI/CD is not without its challenges. Assurance, the trade-off between speed and certainty, reflects the difficulty of balancing improved code validation with the potential slow down from additional tests. Security, concerning both access and information protection, underlines the conflict between maintaining the CI pipeline's integrity and providing developers the necessary access for debugging. Finally, the Flexibility versus Simplicity trade-off emphasizes the tension between a desire for highly configurable CI systems and the corresponding increase in complexity, which may hinder usability. These trade-offs, relevant to both CI and CD, do not diminish the value of these methodologies but highlight the nuance required for effective implementation [8].

The interplay between CI/CD and software architecture further adds to the complexity, with architectural challenges emerging as key considerations in CD/DevOps implementation. The adoption of Continuous Delivery (CD) and DevOps practices brings to light particular software architectural challenges and beneficial characteristics, as identified in a review of recent literature [9]. Ten architectural issues were found, including challenges related to monolithic architectures and complex logging and monitoring. On the other hand, 17 beneficial characteristics were pinpointed, with Deployability, Testability, Automation, Loosely Coupled, and Modifiability being the Top-5 most frequently discussed. Micro-services emerged as the dominant architecture style suitable for CD/DevOps, addressing many of the beneficial characteristics but potentially introducing complexities in areas like traceability and monitoring.

A particular aspect of CI/CD that has drawn attention is the maturity of test automation, especially within the context of open source projects. In the context of Continuous Integration/Continuous Deployment (CI/CD) within open source

projects, test automation maturity has shown significant correlations with product quality and release cycles. Wang et al.'s quantitative study found that elevated levels of test automation maturity - as determined by recognized best practices - are associated with enhanced product quality and condensed release cycles. Notably, these improvements do not necessitate an increase in test automation effort [10]. Furthermore, the study unveiled new insights into the influence of product complexity and team size on test automation effort, and the reasons behind lower quality in older projects. In the CI/CD framework of open source projects, these results emphasize the importance of adhering to standard best practices in test automation to enhance product quality and accelerate release cycles without additional effort. Therefore, Wang et al.'s study supports the value of investment in test automation maturity as a strategic approach to optimizing product quality and development efficiency in the CI/CD process.

2.2 Jenkins

Jenkins is a prominent open-source automation server, automating various phases of the software development process, including compilation, testing, packaging, and deployment. Jenkins Pipeline (or simply "Pipeline") is a suite of plugins that support the integration of continuous delivery pipelines into Jenkins, enabling a more comprehensive way to integrate the entire chain of build, test, and deployment tools using Pipeline scripts, stages, commands, and jobs [11, 12, 13]. The Pipeline, defined in a text file known as a Jenkinsfile, is considered a foundation of "Pipeline-as-code," treated as part of the application and providing benefits like code review on the Pipeline and a single source of truth [13].

Pipelines in Jenkins are defined as automated sequences that transport software from version control through successive stages of the development lifecycle, including building, testing, and deployment [13]. These processes can be executed across different systems, potentially altering the state of the software and producing artifacts when necessary [14]. While Jenkins does not fully negate the requirement to develop scripts for each step, it does offer a comprehensive toolset for modeling pipelines ranging from simple to complex, all via its Pipeline domain-specific language (DSL) syntax. This syntax includes two forms - Declarative and Scripted, with the former providing more advanced syntactical elements [12, 14, 13].

Introduction and Features

Jenkins's history dates back to 2004 when Kohsuke Kawaguchi, a Java developer at Sun, built Hudson to prevent build failures by testing changes before committing

them to the repository. This open-source project gained popularity and was eventually renamed Jenkins in 2011, with Hudson's eventual shutdown in January 2020 [15, 12]. Jenkins is noted for several valuable features:

- **Ease of Use:** Accessible to both technical and non-technical users, including an administrative user setup and easy initial plugin selection [15, 12].
- **Customizability:** High adaptability through declarative or scripted pipelines, offering enhanced workflows defined by scripts. Stages can be named sequences of commands, used by visualization tools, and each job marks an execution instance of a pipeline [15, 12, 14].
- **Plugin Support:** A large index of community-contributed plugins, including the Pipeline plugin that supports custom extensions to its DSL as well as options for integration with other plugins [15, 12, 16, 13].
- **Self-Hosting Ability:** Attractive to companies requiring full control over CI automation, featuring benefits such as pausability, versatility, and extensibility of pipelines [15, 12, 16, 13].

Challenges and Limitations

Despite its popularity, Jenkins has faced challenges:

- **Difficult to Configure:** Reported difficulties in configuration and some complexities in scripting [16, 12].
- **Slowness:** Slower runner speed mentioned as a shortcoming [16].
- **Lack of Scalability:** Issues with scalability, including longer build times and limitations in processing power [16].
- **Plugin Problems:** Challenges due to the need for many plugins from the start [16].

Jenkins in the CI/CD landscape

Jenkins continues to play a significant role in shaping the CI/CD landscape, supporting various languages and tools, including C/C++, PHP, Python, Ruby, and being tightly integrated with Docker [11, 15, 12]. Its Pipeline features, such as durability, fork/join ability, and making chaining tasks a first-class citizen, provide added advantages over its competitors [13]. It often gets compared to Atlassian Bamboo, with key differences being its open-source nature, a larger community, and more plugins [12]. Jenkins's popularity, flexibility, and ease of use have made it a preferred

choice for many projects, and continuous research and user feedback contribute to understanding its real-world applications, challenges, and ongoing evolution in the CI/CD domain [15, 16].

2.3 BitBucket Pipelines

Bitbucket Pipelines is an integrated CI/CD service built into the Bitbucket cloud-based repository service, designed to build, test, and deploy code directly from Bitbucket repositories. It replaces the traditional model of having separate platforms for source control and continuous integration by offering a unified pipeline within Bitbucket itself [17]. Bitbucket Pipelines leverages Docker containers to build and run pipelines, allowing for environment customization and ensuring that software will run the same way in development as it does in production [18].

In Bitbucket Pipelines, pipelines are defined using a bitbucket-pipelines.yml configuration file stored at the root of the repository. This approach aligns with the "Infrastructure-as-Code" paradigm, where CI/CD pipelines can be version-controlled and reviewed just like application code [19, 20]. The YAML-based configuration simplifies the process of setting up complex workflows and tasks, making it easier to configure, maintain, and understand.

Introduction and Features

Bitbucket Pipelines, a component of Atlassian's Bitbucket cloud offering, was introduced in 2016 with the objective of creating a more integrated and streamlined CI/CD experience [21]. One of the key advantages of Bitbucket Pipelines is its tight integration with other Bitbucket features such as code reviews and issue tracking. As noted by Sean Regan, Atlassian's head of marketing for Jira, Bitbucket, and Devtools, this integration extends to the realm of infrastructure as code, allowing source code and infrastructure configurations to coexist and evolve within the same environment [21]. This synergy allows tests to be executed directly where the code resides and facilitates seamless deployment to cloud platforms like Amazon. Such a level of integration serves to liberate developers from the complexities often associated with managing Jenkins infrastructure, a scenario Regan refers to as "Jenkins jail" [21]. By reducing the overhead associated with pipeline management, Bitbucket Pipelines allows developers to focus more on coding, thereby potentially reducing personnel costs related to pipeline maintenance. Bitbucket Pipelines offers several noteworthy features:

- **Simplicity:** A straightforward setup with quick start templates and an intuitive user interface [17].

- **Flexibility:** YAML-based configuration files allow for flexible pipeline definitions, with support for multi-step and parallel builds [20, 22].
- **Docker Support:** Native Docker integration for environment customization, enabling developers to build, tag, and push Docker images [18].
- **Branch Workflows:** Built-in support for branch-specific pipelines, allowing different tasks to run based on the branch being built [20].

Challenges and Limitations

Although Bitbucket Pipelines provides a streamlined and integrated CI/CD process, it also has limitations:

- **Limited Language Support:** Being container-based, it naturally supports languages that run inside Docker containers, potentially limiting older technologies [18].
- **Resource Constraints:** Restrictions on build minutes and the number of concurrent builds can become a bottleneck for larger teams [23].
- **Less Extensibility:** While it supports many third-party integrations, the selection is not as extensive as Jenkins' plugin ecosystem [24].

Bitbucket Pipelines in the CI/CD landscape

The main advantage of Bitbucket Pipelines is its integration with Bitbucket repositories and its straightforward, YAML-based approach to pipeline configuration [16]. It offers a simplified, integrated solution, particularly appealing to smaller teams and projects that can benefit from its cost-effective model and reduced operational overhead. Compared to Jenkins, Bitbucket Pipelines offers less flexibility but is easier to set up and manage, making it a strong alternative for organizations that prioritize ease-of-use and quick setup. Continuous advancements and feature rollouts signify its growing importance in the CI/CD landscape, backed by Atlassian's reputation and focus on integrated software development tools.

3 Methodology

This chapter outlines the research methods employed to evaluate Jenkins and Bitbucket Pipelines as CI/CD platforms. The methodology is rooted in empirical data collection, feature comparison, and cost analysis. Data sources primarily include operational metrics from SST Digi's deployments and relevant technical literature, such as documentation and whitepapers.

A comprehensive review of existing literature and technical documentation furnished the theoretical framework, enabling a nuanced understanding of the CI/CD tools in focus, namely Jenkins and Bitbucket Pipelines. Concurrently, empirical data were gathered from SST Digi's live operational environment. These metrics provide tangible insights into real-world performance and resource utilization.

In addition to cost metrics, a direct feature and architecture comparison between Jenkins and Bitbucket Pipelines was performed. Criteria for evaluation encompassed key functionalities such as adaptability, customizability, and parallel execution capabilities. Additionally, the architecture of each platform, especially in its interaction with AWS services, deployment strategies, and security measures, were critically evaluated. The study however is not without limitations. Certain assumptions, like the build execution time for Bitbucket Pipelines, were made and require further empirical validation.

In summary, the methodology employed in this research represents a hybrid approach, amalgamating theoretical insights from existing literature with empirical data from Jenkins and Bitbucket Pipelines deployments within SST Digi's operational context. This dual-method approach forms the basis for the subsequent chapters that delve into a detailed comparative analysis of the two CI/CD platforms.

4 Jenkins and BitBucket Pipelines Comparison

This chapter provides an examination of Jenkins and Bitbucket Pipelines, focusing on their core architectural principles, setup processes, and the practicalities of pipeline management within CI/CD practices. It contrasts the open-source, customizable nature of Jenkins with the integrated, service-oriented architecture of Bitbucket Pipelines, highlighting the trade-offs between control and maintenance. The review also explores the user experience and financial implications associated with each tool. Through this analysis, the chapter aims to offer insights into the suitability of each CI/CD tool for different organizational needs.

4.1 Architecture and Infrastructure

Jenkins is characterized by its open-source, self-hosted nature, requiring individual setup and maintenance of build servers [11, 12]. Operating in a Java Virtual Machine (JVM), it is platform-independent and functions on various types of hardware and different operating system environments [15]. The self-hosted model affords a high degree of control over the environment, essential for organizations with specific constraints, such as data privacy or usage of certain hardware [25]. However, this also brings a maintenance overhead, encompassing updates to the Jenkins server and its plugins, and addressing hardware or networking issues [12]. A study highlights that this maintenance burden can be so significant that some organizations allocate almost one full-time role to maintain Jenkins [16].

Contrastingly, Bitbucket Pipelines is an integrated, cloud-based CI/CD service within the Bitbucket version control system (VCS) [26]. The need for server infrastructure management is eliminated as builds are executed in Docker containers managed by Atlassian [18]. The diminished maintenance and the simplified initiation of CI/CD processes are evident benefits, fostering a more unified user experience due to the integration with Bitbucket's VCS [26]. However, a trade-off exists in terms of diminished control over the build environment, potentially leading to compliance or security issues depending on organizational requirements. Build resources are also confined by the options offered by Bitbucket Pipelines, potentially affecting build times for larger projects [27].

A trend is noted towards the migration from self-hosted CI tools like Jenkins to cloud-based solutions such as Bitbucket Pipelines. This transition is driven by the reduced maintenance burden and the availability of free tier cloud services, which are particularly advantageous for small teams and projects in terms of personnel and hardware resources [16].

4.2 Ease of Setup and Use

Setting up Jenkins, a self-hosted CI/CD tool, involves installing the Jenkins server, arranging the build environment, and configuring necessary plugins [11, 12, 15]. Jenkins Pipeline's domain-specific language (DSL) is based on Groovy, a dynamic, object-oriented programming language that runs on the Java Virtual Machine (JVM). The DSL leverages Groovy's syntax and runtime capabilities to define complex Continuous Integration and Continuous Deployment (CI/CD) workflows. This integration allows for the incorporation of conditional logic, loops, and exception handling within the pipeline definitions [28]. This may introduce a learning curve for users unfamiliar with the language. Despite the initial complexity, Jenkins offers a high degree of customization and control, enabling the creation of complex pipeline structures [15].

Ease of use in CI tools is critical, with several factors contributing to a positive user experience. This includes a straightforward user interface, detailed documentation, and straightforward configuration processes. Stability in configuration options also plays a crucial role in enhancing usability. However, it is noteworthy that Jenkins' user interface has been characterized as outdated, which could impact its overall user experience negatively [16].

In contrast, Bitbucket Pipelines is a cloud-based service built into Bitbucket's VCS, eliminating the need for server installation and setup [26]. Users configure pipelines by creating a 'bitbucket-pipelines.yml' file in the repository root. Bitbucket automatically detects this file and sets up the build configuration [20]. The simplified setup process and the easy-to-use YAML syntax enhance the user experience. However, while this simplicity is beneficial for smaller projects, it may not provide the same level of flexibility and control that Jenkins offers for more complex pipeline structures as evidenced by the following section.

4.3 Pipelines

The concept of pipelines is central to the continuous integration and continuous delivery (CI/CD) process. Pipelines orchestrate the steps that code changes undergo, from the initial development stages through to deployment in production environments. They automate this workflow, ensuring consistency and speed. In this section, the pipeline methodologies and organizational structures provided by Jenkins and Bitbucket Pipelines are explored and compared. By understanding their respective strengths, complexities, flexibility, and ease of use, valuable insights can be obtained for teams and organizations considering these tools for their CI/CD implementations.

4.3.1 Pipeline Methodologies in Jenkins

Jenkins offers two methodologies for creating pipelines: Declarative and Scripted. Declarative pipelines, a more recent addition, use a structured and simplified syntax, improving both readability and ease of writing for users who need uncomplicated pipeline configurations [28]. On the other hand, Scripted pipelines, based on Groovy DSL, grant extensive control over the pipeline creation process. They are, however, more complex and require a deep understanding of Groovy, which can pose a steep learning curve for unfamiliar users [15]. Notably, Jenkins' Scripted pipelines enable advanced control flow syntax, such as loops or conditional statements that are not directly supported in declarative pipelines or YAML-based pipelines like those used in Bitbucket. Both methodologies support splitting a pipeline into stages, which in turn can be split into steps, each representing a distinct phase of the build process.

4.3.2 Pipeline Configuration in Bitbucket

Bitbucket Pipelines employs YAML for pipeline configuration, which offers an intuitive syntax that is straightforward to understand. This characteristic makes Bitbucket Pipelines user-friendly, particularly for individuals new to CI/CD processes [26]. While the simplicity of YAML promotes quick adoption and readability, it does not provide the flexibility that Jenkins' Scripted pipelines do. As a data serialization language rather than a programming one, YAML does not natively support complex programming constructs like loops or error and exception handling [29].

Bitbucket Pipelines requires the inclusion of script commands directly within the YAML configuration file, which is an added advantage [30]. However, these script commands are executed at the step level, and they lack the capability to interact with the pipeline's execution beyond stopping it. Thus, while this offers some level of control and complexity within individual steps, it does not match the level of comprehensive control that Jenkins offers.

The introduction of stages in Bitbucket Pipelines, currently in Beta, offers additional organization and control over the pipeline process. These stages enable the logical grouping of pipeline steps with shared properties. For example, steps for the same deployment environment can be grouped, preventing interaction from other Pipeline runs, and sharing deployment variables across multiple sequential steps. This feature enhances the structure of deployments, aids in quickly identifying failed deployment parts, allows rerunning of only the failed parts instead of full redeployment, and enables sharing of deployment variables and maintaining a lock over a Deployment Environment across multiple steps. [31]

However, the stage feature in Bitbucket Pipelines has some limitations. Parallel stages are not supported, and a stage cannot include parallel steps. Moreover, while

a stage cannot contain manually triggered or conditional steps, the entire stage can be configured to be manually triggered or conditional [31]. This addition is an important step towards enhancing the control and complexity Bitbucket Pipelines can handle, but it still lacks some of the features available in Jenkins.

4.3.3 Comparative Analysis of Pipeline Organization

Jenkins and Bitbucket Pipelines share a lot of similarities in their approach to pipeline organization. A pipeline in either tools can broken down into different stages, which can represent various environments or different phases in the development process (like 'build', 'test', and 'deploy'), and these stages can be further split into steps, providing a granular level of management, visualization, and control for each stage independently [15]. However, Jenkins also proposes the possibility to trigger another Jenkins job from a pipeline. This hierarchical organization provides a high degree of control and flexibility, especially beneficial for managing complex or large-scale projects.

In the following example, the Jenkins pipeline is depicted with distinct deployment and testing stages. However, instead of embedding the logic directly within the pipeline, each step calls a separate Jenkins job with different parameters. These jobs are responsible for the actual execution of the deployment and testing processes, and different invocations of the same job do different things depending on what parameters they receive. This modular approach allows for better maintainability and reusability of the jobs, as they can be developed, updated, and managed independently from the main pipeline's flow.

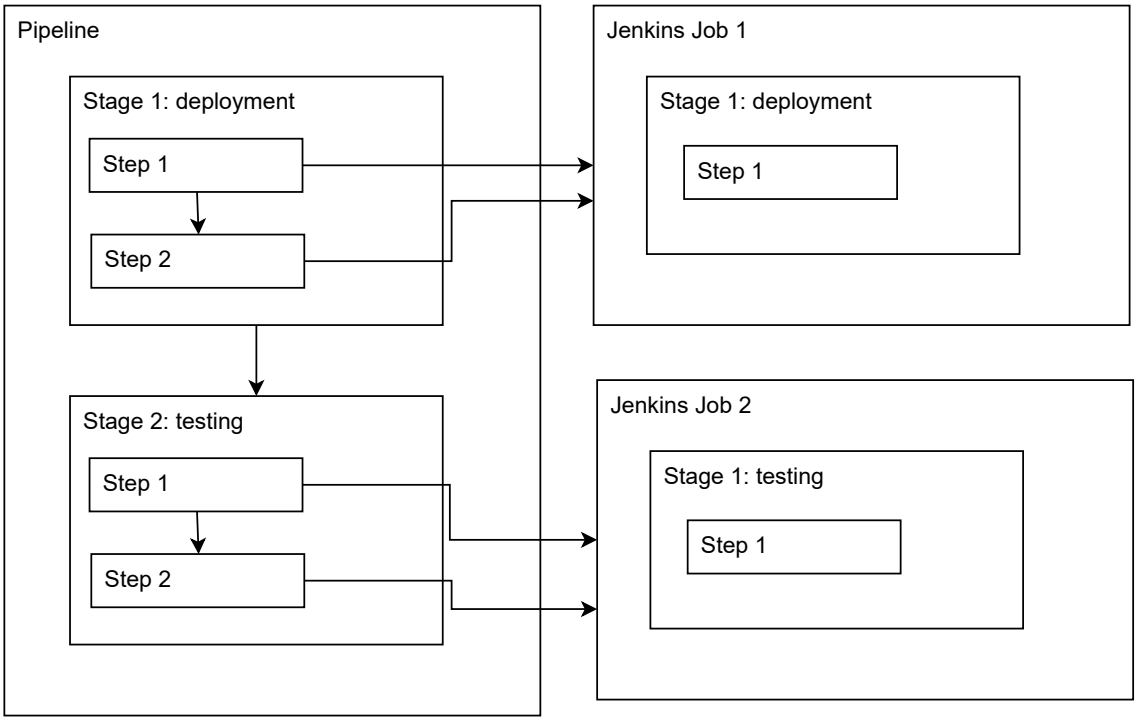

Figure 4.1 *Example of a modular Jenkins pipeline*

On the other hand, Bitbucket Pipelines uses a more simplified structure where a pipeline directly comprises steps, which can be grouped into stages, executed in individual Docker containers [30, 31]. The following diagram illustrates a pipeline where the deployment and testing logic is embedded directly within it.

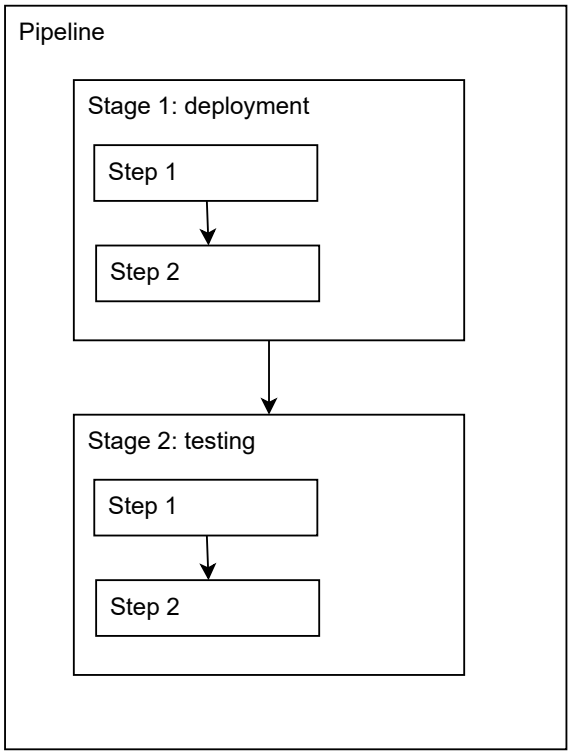

Figure 4.2 *Example of a typical Bitbucket pipeline*

While this approach promotes simplicity and easy understanding, it may lack the granularity and control provided by Jenkins for intricate requirements. For instance, while stages of a development process can still be represented as separate steps within a Bitbucket Pipeline, the user might not have the same degree of control or flexibility as Jenkins' additional levels of separation offer. It is important to note that it also common in Jenkins to define a pipeline in the same manner. In that regard, what is possible to do in Bitbucket Pipelines is also possible in Jenkins, but not the other way around. For teams valuing simplicity and swift setup, Bitbucket's approach might be sufficient. Conversely, for larger, complex projects requiring more granular control over their CI/CD processes, the added separation provided by Jenkins could be beneficial [16].

Conditional Execution

Jenkins' pipeline configuration supports conditional execution of jobs or stages based on various conditions, including the success or failure of previous stages, changes in specific parts of the codebase, or the result of a script. This feature, available through its Scripted and Declarative Pipeline syntax, provides an enhanced level of control and flexibility in managing the build, test, and deploy process. Furthermore, Jenkins' Groovy-based pipeline configuration permits dynamic pipeline creation,

where steps or job calls can be generated dynamically based on certain conditions or parameters. This feature is particularly useful for complex projects requiring a high degree of adaptability and dynamism in their CI/CD processes. [32]

In contrast, Bitbucket Pipelines, with its YAML-based pipeline configuration, does not natively support conditional execution of steps or dynamic pipeline creation, with the exception of "changesets" condition. This condition allows a step or stage to execute only if one of the modified files matches the expression in includePaths, providing a certain level of conditional control [30, 31]. Despite this, the overall lack of support for a wider range of conditions or dynamic pipeline creation can limit its adaptability and control, particularly in complex projects with specific pipeline execution needs.

Parallel execution

Jenkins provides robust support for running jobs or stages in parallel through its Declarative and Scripted Pipeline syntax. In the Declarative syntax, the "parallel" directive can be used to run multiple stages concurrently. This is particularly useful in scenarios where stages are independent and do not rely on the output of one another. A job defined in a Jenkins Pipeline can, for example, build an application, run tests, and deploy to a test environment simultaneously, given these tasks are independent and do not conflict with each other. In the more flexible Scripted Pipeline syntax, parallel execution can be achieved using the "parallel" function. This function accepts a Map of "branch" closures that should be executed in parallel. Each closure corresponds to a set of tasks that constitute a branch of the parallel execution. These branches run concurrently and independently, further emphasizing the power of Jenkins' parallel execution capabilities. [28]

Bitbucket Pipelines also supports parallel execution of steps, albeit with a slightly more simplistic approach. To implement parallel execution, Bitbucket Pipelines uses the 'parallel' keyword in the pipeline configuration. This keyword is followed by a list of steps that are to be run concurrently. These parallel steps are completely isolated from each other, running in separate Docker containers, and they do not share a working directory or environment variables. Despite its straightforward implementation, Bitbucket Pipelines has its limitations. Specifically, Bitbucket Pipelines does not support running sequences of steps (essentially, "stages" in Jenkins terminology) in parallel to each other. [22]

4.4 Cost Structure Comparison

The cost structure of a CI/CD platform is a significant factor to consider when making a choice between different options. It is not only the direct expenses that need

to be accounted for, but also the indirect costs such as maintenance, scalability, and hardware resources. Jenkins and Bitbucket Pipelines present fundamentally different cost structures due to their different deployment models, which can influence the choice between these two options.

Jenkins

Jenkins is an open-source platform, which means it is free to use [11]. However, being a self-hosted solution, there can be indirect costs associated with its use. The necessary infrastructure such as servers to host and run Jenkins can add to the expenses, whether hosted on-premise or in the cloud. Moreover, as one participant reported in a study, maintaining Jenkins could potentially require significant time and resources, to the point of needing a full-time person dedicated to its upkeep [16].

Furthermore, the regular updates and maintenance required for plugins and the core system add to the total cost of ownership. In light of these considerations, while Jenkins itself is free, the indirect costs can add up and must be considered when evaluating the cost-effectiveness of implementing Jenkins.

Bitbucket Pipelines

Bitbucket Pipelines, on the other hand, offers a cloud-based CI/CD service integrated into the Bitbucket platform. This model of CI as a service is increasingly appealing to teams looking to reduce cost and hardware resources, as it eliminates the need for self-hosting and associated maintenance costs [16]. Bitbucket Pipelines operates on a pay-as-you-go model, primarily based on build minutes, i.e., the time your pipelines run [33]. The standard Bitbucket plan includes 2,500 build minutes, with a charge of $10 for every additional 1,000 minutes.

This pricing model provides scalability and cost-effectiveness, especially for smaller projects or teams just starting with CI/CD practices. However, for larger projects with extensive build needs, the costs can escalate quickly, particularly if parallel builds or computationally intensive jobs are frequent.

4.5 Scalability Comparison

Scalability is a crucial attribute to consider when choosing a CI/CD platform. It determines how effectively a tool can adapt to increasing workloads, and whether it can maintain its performance and efficiency as demand grows. Jenkins and Bitbucket Pipelines present different scalability features due to their inherent architectural differences.

Jenkins

Jenkins offers significant scalability options due to its distributed nature [11]. It can distribute tasks across multiple nodes or ”agents”, which can be set up on different machines. This distributed architecture allows Jenkins to manage a large number of tasks and heavy workloads effectively by leveraging the computational resources of multiple machines. Whether the nodes are hosted on-premises or in the cloud, Jenkins can scale horizontally to match the growth of the project [34].

Moreover, the ability to add or remove nodes as needed means Jenkins can handle fluctuating demand efficiently. It also allows for parallel execution of jobs across different nodes, thus reducing build times. However, managing a distributed Jenkins system can be complex and might require dedicated resources, which should be factored into scalability considerations [15].

Bitbucket Pipelines

Bitbucket Pipelines, being a cloud-based CI/CD service, inherently offers scalability by leveraging cloud resources. However, its scalability potential is somewhat tied to its pricing model, which is based on build minutes. Teams can scale up their usage of Bitbucket Pipelines to accommodate larger workloads, but this will directly increase the cost. Unlike Jenkins, Bitbucket Pipelines does not require managing multiple nodes or servers, which simplifies the scaling process. Yet, the lack of control over the underlying infrastructure might be a limitation for specific use-cases or for projects that need more than just straightforward scalability. The parallel execution of steps in Bitbucket Pipelines is possible but uses more build minutes and might increase costs. [22, 33]

4.6 Side-by-side Comparison

Feature/Aspect	Jenkins	Bitbucket Pipelines
Dynamic Job Execution	Supports dynamic execution of jobs or steps based on conditions or parameters.	Jobs are statically set in the configuration file without conditional execution.
Parallel Execution	Can run sequences of linear steps in parallel.	Only supports running steps either sequentially or fully in parallel. Cannot run sequences of steps in parallel with each other.
Tooltips	Each job can have its own description, aiding in clarity and organization. Parameters can be further explained with a tooltip.	Pipelines lack a distinct description feature.
Pipeline Continuity	Jobs triggered as part of a pipeline are inherently considered part of that pipeline.	A pipeline triggered by another does not maintain continuity and is not considered "part" of the original pipeline.
Customizability	High level of customizability with plugins and user-defined configurations.	Limited to the features and configurations provided by Bitbucket.
Integration with Tools	Wide range of integration capabilities with third-party tools.	More streamlined and specific to Bitbucket's ecosystem.
Pricing	Open-source but costs can arise from infrastructure, plugins, and maintenance.	Pricing is based on build minutes and number of users.
Scalability	Can be scaled using various plugins and cloud resources but may require manual configurations.	Built-in scalability but limited to the confines of Bitbucket's ecosystem.
Memory	Limited only by the resources provided by cloud service providers such as AWS EC2 or the specifications of on-premises servers.	Restricted to a maximum of 8GB of memory shared between build containers and services.

Table 4.1 *Side-by-side feature comparison of Jenkins and Bitbucket Pipelines.*

5 Deployment Background and Current Setup

In this chapter, the current deployment process, tools, and methodologies used in the infrastructure setup are discussed. A comprehensive understanding of the existing environment helps in appreciating the migration challenges and strategies discussed in subsequent chapters.

5.1 Infrastructure Management and Terminology

The current deployment setup adopts an infrastructure-as-code model. Within this model, AWS cloud infrastructure and its associated configurations are defined as code and maintained under version control. Collectively this infrastructure makes up the IoT platform that powers Kalmar Insight.

Understanding the terminology used in the deployment process is crucial for comprehending the overall structure:

- **Environment:** This refers to a specific cloud deployment that is able to pass system tests. It might be composed of one or more projects and can be deployed to multiple AWS accounts. Often, resources within an environment have names prefixed with the environment's name for ease of identification. In deployment commands, this name usually acts as a unique parameter.
- **Project:** A project groups resources and is deployable as a singular unit to an AWS account. Some projects function independently, while others may rely on preceding deployments due to interdependencies. Every deployed project is tied to a specific environment.

5.2 Deployment Tools and Strategies

The deployment process utilizes a suite of tools including Ansible [35], boto (AWS SDK for Python) [36], AWS CLI [37], AWS CDK [38], and Flyway [39]. Ansible is particularly notable for its support of Jinja2 templating and its role in handling variables, overseeing diverse tasks, and pre-processing CloudFormation templates. While both CloudFormation and CDK are entrusted with managing the majority of AWS resources, CDK, being the newer addition, is favored over Jinja2+CloudFormation. Regarding deployment strategies, while most projects have flexibility in terms of their AWS account deployment, certain projects are designed to be deployed solely to a specific account, custom-tailored to accommodate them.

While it is feasible to deploy the entire IoT platform directly from a developer's computer, the use of Jenkins jobs introduces an additional layer of cloud-based

support. This setup enables the performance of various operations, ranging from initializing or deleting an entire development environment to executing system tests, all without requiring any local configurations. These automation tools significantly streamline processes, ensuring efficiency and consistency in deployment and testing across different environments.

5.3 Jenkins Architecture

SST Digi uses Jenkins as its primary automation server for continuous integration and continuous deployment processes. Hosted within the Amazon Web Services (AWS) cloud infrastructure, this Jenkins setup offers a scalable and resilient environment tailored to SST Digi's requirements. By exploring the detailed aspects of how Jenkins is configured, the aim is to highlight the comparative effort and complexity involved in its deployment versus using a fully managed service like Bitbucket Pipelines.

5.3.1 Infrastructure and Networking

The infrastructure and networking section describes the foundational components that support Jenkins operations. These include storage solutions for persistent data, security measures for traffic control, and systems to distribute workload efficiently:

- **Elastic File System (EFS)**: Jenkins requires persistent storage for configuration, job histories, and workspace data. An Amazon EFS filesystem addresses this need, optimized for high-performance read/write operations.
- **Security Groups**: Numerous security groups regulate both inbound and outbound traffic. Special attention is given to the Jenkins master, ensuring access is restricted to authorized traffic sources.
- **Load Balancing**: An Application Load Balancer (ALB) uniformly distributes incoming traffic.

5.3.2 Compute Infrastructure

The computational backbone of the Jenkins in AWS setup is designed for reliability and flexibility. Below are the core compute resources employed:

- **EC2**: Jenkins is deployed directly on Amazon EC2 instances. Configurations such as EC2 instance types.
- **Auto Scaling Group (ASG)**: The Jenkins master within an ASG to ensure availability in the likely scenario of the master EC2 getting terminated.

Similarly, there is a separate ASG for the Jenkins agents with the additional responsibility of spinning up more instances if the Jenkins job queue gets too long, or removing them when the queue has been handled.

5.3.3 Monitoring, Logging, and Security

Ensuring the reliability and security of the Jenkins architecture involves comprehensive monitoring, logging, and adherence to stringent security practices. The tools and implemented are as follows:

- **CloudWatch**: Enables proactive monitoring with alarms on key metrics and centralized logging for efficient troubleshooting.
- **AWS Backup**: Ensures data integrity with periodic backups of the EFS filesystem.
- **IAM Roles and Policies**: Specific IAM roles and policies grant Jenkins permissions, enhancing security for various operations, such as SSH interactions. This includes an OIDC role that enables token-based authentication to manage AWS services, eliminating the need for long-term credentials.

5.4 Jenkins Deployment Pipeline Architecture

The Jenkins deployment pipeline at SST Digi is specifically tailored for deploying select projects on Amazon Web Services (AWS). While there are numerous Jenkins jobs employed at SST Digi for various tasks, this section emphasizes the architectural design and operation of the project deployment pipeline.

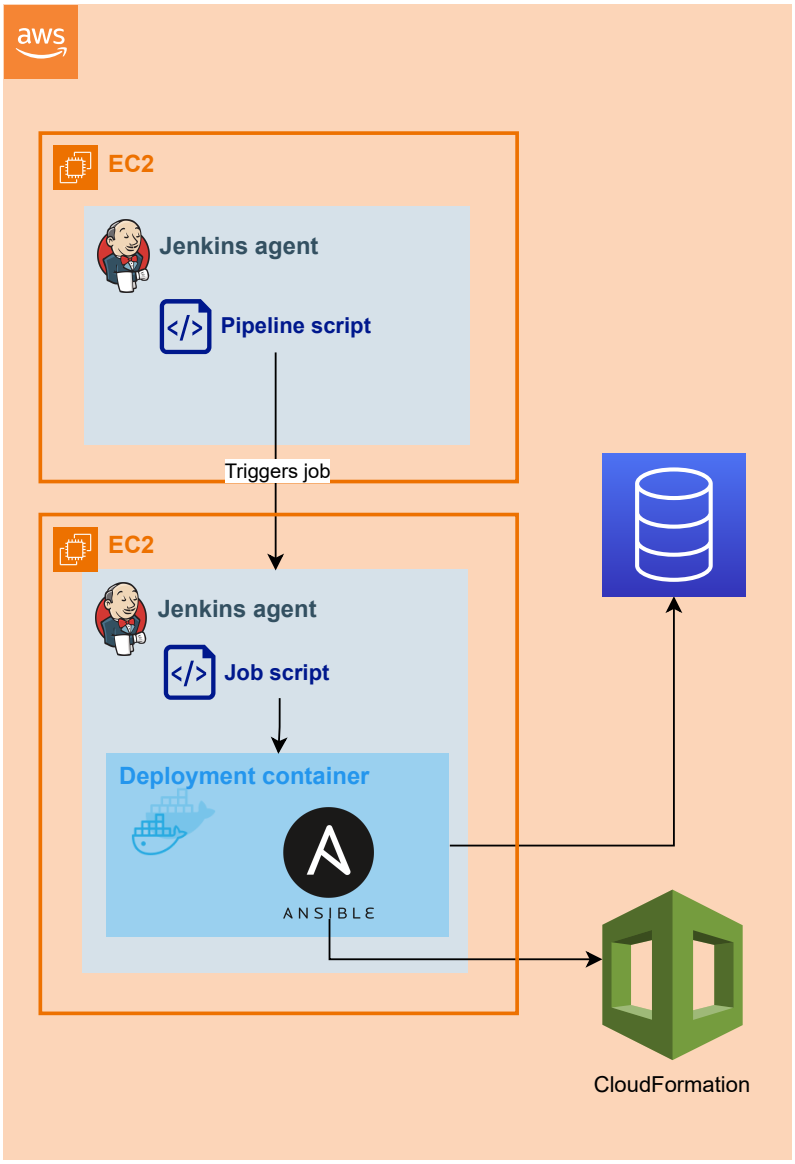

Figure 5.1 *Jenkins project deployment architecture.*

The described Jenkins pipeline serves as an orchestrator for deploying multiple projects to specified AWS accounts, also known as "inventory." The pipeline is parameterized, allowing users to specify crucial information such as the list of projects to deploy and the target AWS inventory. It operates in three distinct stages:

1. **Initialization Stage**: This stage prepares the deployment process by categorizing projects into two lists—those to be deployed in parallel and those requiring sequential deployment.

2. **Sequential Deployment Stage**: Projects listed for sequential deployment are processed in this stage. Each project's deployment is managed by calling a dedicated Jenkins job tailored for individual project deployments.

3. **Parallel Deployment Stage**: This stage manages the deployment of projects or sequences of projects that can be deployed in parallel. For example, project A and the sequence of projects [B, C, D] could be deployed simultaneously, but the projects B, C, and D would be deployed sequentially within their group.

Each time a project is deployed, a separate Jenkins job is triggered. This subordinate job consists of a single step that utilizes two external Groovy scripts. The first script configures AWS profiles based on the given inventory, and the second script executes the deployment, passing along the user-provided parameters.

5.4.1 Modularity and Design Principles

The deployment pipeline is structured with a modular approach, partitioning the process into distinct scripts. This modularity ensures that each segment of the pipeline serves a specific function, simplifying maintenance and modifications.

- **Pipeline Configuration and Setup**: The logic of the deployment pipeline is written into a Jenkinsfile. It is responsible for initializing parameters as well as structuring the deployment stages and sequencing. By analyzing parameters and project lists, it determines the order of deployments, grouping projects into sequences for sequential deployment and identifying which sequences can be deployed in parallel. Based on the provided projects list, the pipeline dynamically triggers another Jenkins job responsible for the deployment of individual projects.
- **AWS Profile Configuration**: Temporary AWS CLI profiles are dynamically configured by a separate script written in Groovy, allowing the pipeline to interact with AWS without relying on permanent credentials.
- **Deployment**: A third essential script, also written in Groovy, handles the actual deployment of a single project and is called by the pipeline as needed.

5.4.2 Deployment Strategy and Execution

A noteworthy aspect of the pipeline is its parameter-driven approach. Parameters such as inventory, environment, and projects allow for deployments tailored to different AWS accounts, environments, or specific projects using the same pipeline. The pipeline's code determines the sequence of deployment, deciding which components can be deployed in parallel and which need to be done sequentially. Security is systematically integrated into the pipeline's design. By dynamically configuring temporary AWS CLI profiles based on the specified inventory, the pipeline securely interacts with AWS resources.

6 Migration Strategy and Challenges

Following the comprehensive analysis of the existing Jenkins setup in the previous chapter, the focus shifts towards the migration process of an IoT platform deployment pipeline from Jenkins to Bitbucket Pipelines. The purpose of this case study is to achieve the deployment of the entire IoT platform from within Bitbucket Pipelines.

This chapter aims to provide a detailed walkthrough of the entire migration process, highlighting the challenges encountered, as well as the solutions and workarounds that were implemented to overcome these obstacles. Details included in this part of the thesis will also inform the practical differences between Bitbucket Pipelines and Jenkins.

6.1 Bitbucket Pipelines Deployment Architecture

Bitbucket Pipelines operates differently from Jenkins in terms of orchestrating CI/CD processes. It adopts a standardized approach that emphasizes clarity and simplicity. Utilizing a YAML-based configuration, it offers a compact and structured solution to build and deploy projects. While certain advantages emerge from this approach, there are also inherent limitations. This section details the architecture of the IoT platform deployment pipeline in Bitbucket Pipelines and how it interfaces with AWS to realize deployment objectives.

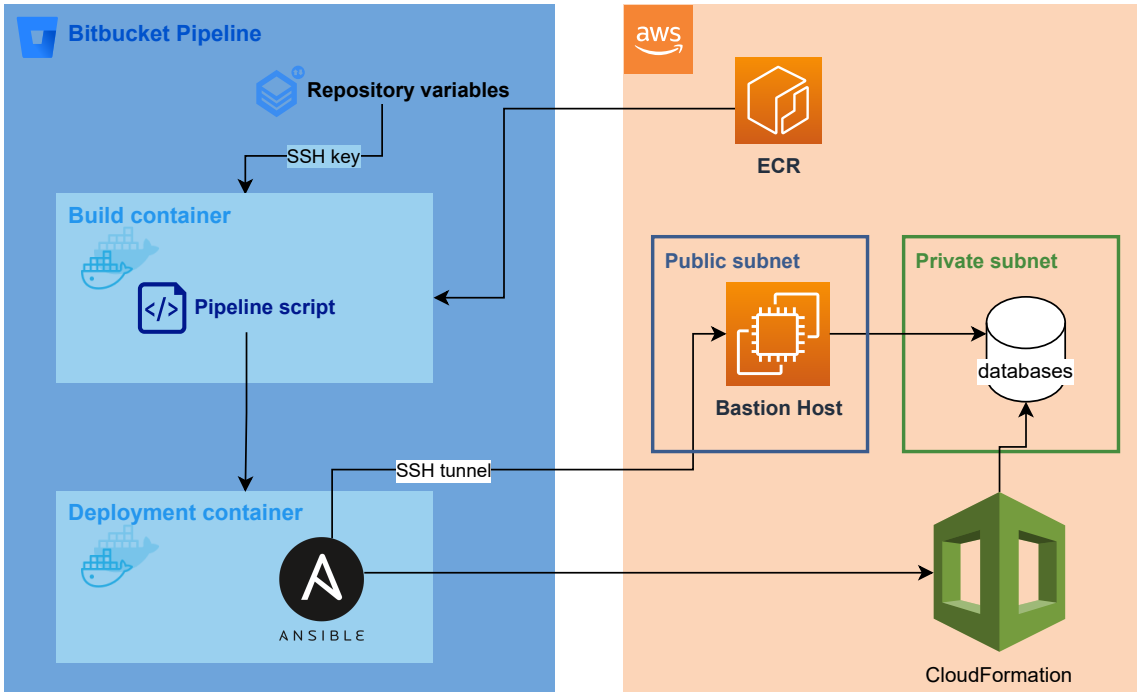

Figure 6.1 *Bitbucket Pipelines deployment architecture.*

Bitbucket Pipelines' mode of operation can be summed up as executing a script within a build container. In this particular case, the executed script is responsible for spinning a new container, here called the deployment container, whose responsibility is to run the Ansible playbook that will interface with AWS to deploy the resources that collectively constitute the IoT platform. The pipeline script also has other responsibilities that will be detailed later in this chapter.

The pipeline is designed to accept user-defined parameters, including a list of projects targeted for deployment and a specified inventory. In contrast to Jenkins, Bitbucket Pipelines relies on a declarative configuration defined in the bitbucket-pipelines.yml file, which means that steps for all potential dependencies are predefined. The pipeline consists of parallel steps, each designated for deploying either a single project or a sequence of projects. Projects are deployed sequentially within each step if that step is responsible for multiple projects. User-driven selectivity in deployment is achieved by simply skipping any project not listed in the user-defined list of projects for deployment. This makes the deployment sequence inherently inflexible, governed by the static step definitions in the configuration file. Each step in the pipeline utilizes two external scripts: the first configures AWS profiles based on the inventory specified by the user, and the second initiates the deployment of the project(s) in accordance with the user-provided parameters.

YAML-based Configuration and Principles

- **Pipeline Configuration**: The pipeline is described in its entirety in the bitbucket-pipelines.yml which orchestrates the deployment steps, service and variable definitions.
- **Docker Integration**: Each step in the pipeline operates within a Docker container, ensuring consistency across deployments. The build containers are derived from a Docker image stored in a private ECR, which encompasses all necessary build dependencies such as git, openssl and AWS CLI.
- **Parameter-Driven Approach**: The pipeline, similar to Jenkins, is driven by user-provided parameters. However, there's a distinction in how each system processes these parameters. Jenkins, being a scripted pipeline, has the flexibility to dynamically run jobs on the fly based on the provided project list, allowing it to deploy an arbitrary set of projects. On the other hand, Bitbucket Pipelines is bound to a predefined setup due to its static configuration nature. Consequently, in the current implementation, it can only recognize and deploy a hardcoded set of projects.

Deployment Strategy and Execution

- **Sequential and Parallel Deployments**: Projects are either deployed sequentially or in parallel, as defined in the YAML. This contrasts with Jenkins, where the deployment order is determined dynamically based on project dependencies.
- **Dynamic vs. Static Deployment**: In Jenkins, the deployment is dynamic, deploying any project based on the given list. In Bitbucket Pipelines, only recognized projects are deployed, limiting flexibility.
- **AWS and Script Integration**: The pipeline heavily relies on shell scripts to interact with AWS and set up deployment environments. These scripts perform tasks similar to their Jenkins counterparts, such as setting up AWS credentials and preparing the deployment environment.
- **Security and AWS Profile Configuration**: A crucial aspect of the pipeline's security is the way it interacts with AWS resources. Instead of using static AWS credentials, the pipeline employs OpenID Connect (OIDC) roles for AWS authentication. This modern method generates temporary credentials for each deployment session, ensuring a reduced risk of credential leakage or misuse. While this dynamic approach to authentication is inherently secure, the Ansible deployment necessitates the use of AWS profiles. To bridge this gap,

a Python script is invoked. This script effectively translates the temporary OIDC credentials into AWS profiles, allowing seamless interaction with AWS for the deployment process. By converting the OIDC authentication data into AWS profiles, the pipeline ensures that Ansible deployments can proceed without requiring any modifications to the way Ansible interacts with AWS.

Strengths and Weaknesses of Bitbucket Pipelines

- **Structured Configuration**: The YAML-based setup offers clarity, making it easier to understand the deployment flow. The structured nature of the format allows for a clear hierarchy and organization of the deployment steps, making the pipeline's flow intuitive. The demarcation between steps, services, and variables in the YAML configuration also facilitates troubleshooting and modifications.

- **Consistency**: The integration of Docker into the pipeline brings about a significant advantage in ensuring consistent environments across all deployment steps. The containers are spawned from Docker images that encapsulate all necessary dependencies and configurations. This means that the build environment can be versioned and easily replicated. The use of a centrally-maintained image further ensures that updates or changes to the environment can be rolled out uniformly, reducing discrepancies between individual deployments.

- **Security**: OIDC roles for AWS authentication provide a secure way to interact with AWS resources without static credentials. Enabling OIDC in Bitbucket Pipelines is straightforward. After configuring AWS or other cloud providers to work with OIDC using unique identifiers provided by Bitbuicket, adding a single line "oidc: true" to the pipeline configuration activates OIDC authentication for the pipeline.

- **Limited Flexibility**: The pipeline can only deploy projects that it is explicitly aware of, limiting adaptability to new or unexpected deployment scenarios.

- **Static Deployment Strategy**: The decision on which projects to deploy in parallel or sequentially is hardcoded and lacks the dynamic adaptability seen in Jenkins.

6.2 Pipeline Code Overview

Bitbucket Pipelines' YAML-based configuration follows a set of rules to create a pipeline. In this section, the configuration and associated scripts driving the Bitbucket Pipelines process will be examined. The aim is to clarify how the pipeline

works at the code level and to showcase integration with services such as Docker and AWS, thus providing a concrete understanding of how the discussed architectural principles and deployment strategies are implemented in practice.

```
options:
  docker: true
  size: 2x

definitions:
    services:
      docker:
        memory: XXXX

pipelines:
  custom:
    deploy:
      - variables:
          - name: variable_name
          ...
      - step:
          name: Deploy project_1
          size: 2x
          image:
            name: some_ecr_url:latest
            aws:
              oidc-role: some_aws_role
          oidc: true
          script:
              - |
                if [[ $projects =~ condition_for_project_1 ]]; then
                    ...
                    source scripts/setup.sh
                    source scripts/deployment.sh
                else
                    echo "Skipping project_1..."
                fi
          services:
          - docker
      - parallel:
          steps:
```

```
- step:
    name: Deploy project_2 and project_2.1
    image:
      name: some_ecr_url:latest
      aws:
        oidc-role: some_aws_role
    oidc: true
    script:
        - |
          if [[ $projects =~ condition_for_project_2 ]]; then
              ...
              source scripts/setup.sh
              source scripts/deployment.sh
          else
              echo "Skipping project_2..."
          fi
        - |
          if [[ $projects =~ condition_for_project_2.1 ]]; then
              ...
              source scripts/setup.sh
              source scripts/deployment.sh
          else
              echo "Skipping project_2.1..."
          fi
    services:
    - docker
- step:
    name: Deploy project_3
    ... % similar setup and conditions as above
- step:
    name: Deploy project_4
    ... % similar setup and conditions as above
```

Options & Definitions: The configuration begins by setting global options. Docker is enabled, and a specific size is allocated for the entire pipeline. Memory allocation for docker services is explicitly defined.

Deployment Structure: The pipeline is organized into steps. These steps are sequential, with some steps containing conditions to determine whether they should be executed. A parallel block allows multiple projects to be deployed simultaneously.

In the `pipelines` section, the main deployment actions occur. The pipeline

uses a custom approach for deployment. The build containers' image is sourced from a private ECR (Elastic Container Registry) repository, as indicated by the `image:name:some_ecr_url:latest` entry. This means the image is not publicly accessible but is stored securely in a private AWS service. The chosen image is designed to come pre-equipped with all the necessary dependencies required for project deployment.

The deployment process is designed to handle dependencies and parallelization. Initially, a foundational project, one on which several others depend, is deployed. Once this foundation is established, the system uses parallel deployment for subsequent projects. This improves efficiency, but it also introduces a challenge. Projects that are interdependent cannot be deployed in isolated steps within parallel blocks because Bitbucket Pipelines does not supported nested steps within parallel steps. A workaround is to deploy these dependent projects within the same step, one after the other, ensuring that dependencies are addressed in the correct sequence.

```
Step 1: [Project A] - foundational project

Parallel Steps:
Step 2: [Project B]          -> A-dependant project
        [Project C -> Project D] -> C is deployed, then D,
            because D depends on C
        [Project E]          -> Another independent project
```

In this layout, `Project A` is the foundational project. After it is deployed, `Project B`, `Project C and D`, and `Project E` are all deployed concurrently. However, within the parallel step for `Project C and D`, `Project C` is deployed first, immediately followed by `Project D` because of the dependency.

The `deployment.sh` script is a one liner whose only purpose is to run the Ansible playbook responsible for deploying a project. The reason for using it is to make the Bitbucket pipeline code cleaner and easier to read since the playbook command is verbose but only uses variables already available to the build container. The other shell script in use `setup.sh` manages setting up the environment and handling credentials. It fetches AWS credentials using OIDC and then processes them with a Python script to create a suitable credentials file, which is a requirement for the Ansible deployment since the playbook cannot connect to AWS using OIDC directly. Another task that the script is responsible for is transferring the SSH key from the build container to an environment variable, which the deployment container needs for successful deployment as explained in the next section.

6.3 Secure access to AWS resources from Bitbucket Pipelines

SST Digi's deployment infrastructure is located in AWS, where databases are hosted within private networks. Jenkins is also hosted in SST Digi's AWS environment but does not have direct access to these private networks. Instead, access is configured through AWS Transit Gateway, which establishes the necessary routing between Jenkins and the target Virtual Private Clouds (VPCs). AWS security groups further refine this network communication, allowing Jenkins to interact with other resources within the AWS private networks.

In contrast, Bitbucket Pipelines is an external service and is not hosted within SST Digi's AWS environment. As a result, it cannot be readily configured to access the private databases hosted in AWS. This sets the stage for unique challenges in establishing secure and efficient network communication between Bitbucket Pipelines and SST Digi's AWS-based resources.

To address the connectivity challenge, a Bastion host was introduced as a secure, intermediary server to bridge the communication between external services and internal AWS resources. A Bastion host is a server that provides controlled access from an external network to internal networks, acting as a single entry point and thereby minimizing potential security risks [40]. The use of Bastion hosts in cloud computing has a well-established precedent in both academic literature and industry best practices. Bastion hosts serve as a critical component for enhancing security in cloud-based deployment architectures by limiting direct access to internal resources, thereby adding an additional layer of security [41].

The Bastion host operates as a "jump server," an intermediary server that mediates access between internal and external networks. In this role, it establishes an SSH tunnel, which is a secure, encrypted channel for communication over an unsecured network. This SSH tunnel is initiated from the Docker container in Bitbucket Pipelines and terminates at the database host within the private network. Specifically within the context of AWS, this strategy allows for the adherence to the principle of least privilege, where the Bastion host is granted only the minimal access rights necessary to perform its role.

While setting up the SSH tunnel, an initial strategy involved opening the SSH connection before deploying. This approach would use the SSH key defined in repository variables, which would be available inside a file within Bitbucket Pipelines' build container. However, this approach encountered a limitation: the SSH tunnel's destination, or target, must be specified when the tunnel is initiated. Given the multiple targets corresponding to the various private databases, this method was not feasible. The tunnel would need to be established before each attempt to access a specific private database. This meant the tunnel had to be created within the

deployment container, not the build container, necessitating the transfer of the SSH key between containers. A viable solution involved reading the SSH key from its file into an environment variable and then passing it to the deployment container using the same mechanism. In the deployment container, an Ansible task was configured to generate an SSH key file and initiate an SSH tunnel, targeting a specified database. This task was invoked before each access attempt to a private database.

Overall, the process can be summarized into the following steps:

1. An AWS Bastion host was created in the same VPC as the private networks and was configured to allow SSH access.
2. An SSH key pair was generated. The private key was stored within BitBucket Pipelines' environment variables, and the public key was added to the authorized_keys file on the AWS Bastion host.
3. The SSH key was read from its file into an environment variable in BitBucket Pipelines' build container.
4. This environment variable was passed to the deployment container.
5. Inside the deployment container, an Ansible task was configured. This task creates an SSH key file using the environment variable and initiates an SSH tunnel to a specified database target.
6. This Ansible task was set to run before each attempt to access a private database.

By following these steps and adapting the deployment scripts accordingly, the SSH tunnel was successfully established from within the Docker container in BitBucket Pipelines though the AWS Bastion host, enabling secure access to private databases for deployment purposes. This solution can be generalized to enable access to any private resources from Bitbucket Pipelines.

6.4 Strategy

The section on strategy outlines the approach taken to transition from Jenkins to Bitbucket Pipelines, detailing the principles that guided the migration to ensure a comprehensive understanding of the process.

Understanding the existing system

The migration process began by analyzing the current Jenkins-based CI/CD setup. This analysis concerned both the underlying infrastructure of Jenkins and the actual

pipeline code. One primary concern was the infrastructure in which Jenkins operated. The resources and capabilities that Jenkins drew from its environment needed to be identified and understood to assess whether Bitbucket Pipelines could match them. Similarly, studying the existing pipeline code used in Jenkins was necessary to determine whether Bitbucket Pipelines could replicate the same functionalities. A significant part of the Jenkins setup is the usage of reusable scripts and functions in the Groovy language tailored specifically for Jenkins. In transitioning to the Bitbucket environment, it was imperative to either replicate these scripts or identify alternative methods to maintain their functionalities

Infrastructure alignment

Moving to Bitbucket Pipelines, which operates in a more containerized environment, necessitated adjustments in the infrastructure setup. It was first essential to identify the precise infrastructure required for project deployment in Bitbucket Pipelines. While Atlassian manages the foundational infrastructure of Bitbucket, three components needed to be created to replicate Jenkins' capabilities: a Docker image, containing all the necessary tools and dependencies for deployment; a bastion host in AWS, to ensure secure communication between Bitbucket and AWS-hosted private databases; and OIDC identity providers, which integrate seamlessly with Bitbucket Pipelines to allow authentication without storing credentials.

Adapting Deployment Scripts and Configurations

Recognizing that a simple transfer approach was not feasible, existing scripts and configurations underwent thorough revisions. The Jenkins setup had certain scripts, configurations, and processes that needed rethinking for Bitbucket Pipelines. As such, deployment scripts were adapted and modified. This step ensured that the modularity principle in the Jenkins implementation was preserved as much as possible, though some loss in that aspect was inevitable due to the declarative style of a Bitbucket pipeline. Incorporating external scripts into the "script" section of a Bitbucket pipeline emulates the scripted pipeline style currently in use in Jenkins and allows for code re-use across different pipelines and steps.

Iterative Testing and Validation

Once the scripts and configurations were in place, the migrated pipeline underwent iterative testing. Each cycle aimed to identify gaps, misconfigurations, or inefficiencies. Feedback from these tests was used to refine the setup further, ensuring that by the end of this phase, the Bitbucket Pipelines setup was robust and ready for full-scale deployment.

Cost Evaluation

Parallel to the technical strategy, there was a financial angle to consider. A cost analysis between Jenkins and Bitbucket Pipelines was conducted. The goal here was not just to identify the cheaper option, but to understand the long-term financial implications, factoring in indirect costs like personnel time.

7 Evaluation and Results

This chapter evaluates the costs, features, and performance of Jenkins and Bitbucket Pipelines in the context of the case study. It compares the computational expenses and personnel involvement required for running CI/CD pipelines and analyzes how each platform's characteristics affect overall efficiency and maintenance demands. A feature analysis examines the flexibility and control provided by each tool, while performance metrics are assessed through deployment times for the IoT platform on AWS. The findings from this chapter aim to guide decisions on whether to migrate from Jenkins to the more streamlined, cloud-based approach of Bitbucket Pipelines.

7.1 Cost comparison

The chapter aims to provide an in-depth cost comparison between Jenkins and Bitbucket Pipelines for running a specific CI/CD pipeline. The evaluation encompasses both the direct costs associated with infrastructure and the indirect costs related to personnel. The analysis reveals substantial differences in cost structures, despite the functional similarities between the two platforms. The focus will be on two principal categories of expenditure: computational costs, which are directly related to the infrastructure supporting the execution of the builds, and personnel costs, which encompass the developer time allocated for maintenance and oversight of the pipeline and its underlying infrastructure.

Computational costs in Jenkins are largely determined by the infrastructure that supports the Jenkins agent. In the context of this analysis, it is notable that the Jenkins agent operates on a *t3.xlarge* Amazon EC2 instance. This EC2 instance incurs a charge of $0.1664 per hour. The pipeline in question runs for an average of 4637.65 minutes per month, which translates to approximately 77.2942 hours per month. This data was obtained through empirical analysis of the last 400 builds executed on the Jenkins platform. Thus, the computational cost for running the Jenkins pipeline can be calculated as 77.2942×0.1664, which amounts to approximately $12.86 per month. It should be noted that the cost for the Jenkins master is not considered in this comparison as it manages multiple pipelines, making its per-pipeline cost negligible.

Bitbucket Pipelines, on the other hand, adopts a different pricing strategy. Costs are incurred based on the total number of build minutes consumed. What sets Bitbucket Pipelines apart is the concept of "effective build minutes." This term refers to the total build time when accounting for parallel steps within the pipeline. For instance, if a build has two parallel steps, each lasting 10 minutes, Bitbucket

Pipelines would bill for 20 minutes. For the Jenkins pipeline under analysis, the effective build minutes, when parallel steps are considered, amount to 6326.29594 minutes per month. This represents an increase of approximately 34% over the parallelized build minutes. Bitbucket Pipelines charges $10 per 1000 build minutes, leading to a computational cost of $\frac{6326.29594}{1000} \times 10$, or approximately $63.26 per month. It is important to note that this calculation assumes that the Bitbucket pipeline would execute for the same duration as its Jenkins counterpart, an assumption based on preliminary data but that requires further empirical validation.

When it comes to personnel costs, the assumption is that Jenkins and Bitbucket Pipelines are nearly on par with regard to the time developers need to allocate for the maintenance of the pipeline logic. However, Jenkins introduces an additional layer of complexity by necessitating the maintenance of the underlying infrastructure, a responsibility that typically falls on the developers.

Typical Jenkins infrastructure maintenance tasks include maintaining security by updating operating systems and installed software such as Jenkins itself and Jenkins plugins, monitoring the health of the infrastructure acting should any problems arise, managing access to Jenkins, and handling issues with badly behaving builds.

This additional overhead can be conservatively estimated to cost half a developer's salary per month. In contrast, Bitbucket Pipelines, as a fully managed service, eliminates the need for infrastructure maintenance, thus incurring no such personnel costs.

The summary of computational and personnel costs for the two platforms is tabulated as follows:

Table 7.1 *Cost Comparison Between Jenkins and Bitbucket Pipelines*

Metrics	Jenkins	Bitbucket Pipelines
Average Build Minutes/Month	4637.65	6326.29594
Computational Cost/Month ($)	12.86	63.26
Personnel Cost/Month	Half of Developer's Salary	N/A

The data reveals that although Jenkins offers a lower computational cost at $12.86 per month, it incurs a significant personnel cost due to infrastructure maintenance. Bitbucket Pipelines, with a computational cost of $63.26 per month but no additional personnel costs, emerges as a more cost-efficient alternative. Therefore, the findings of this analysis support the hypothesis that Bitbucket Pipelines offers financial advantages, particularly for organizations considering a migration to a cloud-based CI/CD service.

The main purpose of this comparison is to find the factor by which Bitbucket Pipelines is more expensive than Jenkins from an infrastructure perspective. Eventually, the Bitbucket pipeline would assume the roles of several other pipelines cur-

rently managed by Jenkins. While the difference in absolute costs would increase, as long as it does not overtake the personnel cost incurred due to Jenkins' maintenance overhead, Bitbucket Pipelines remains advantageous.

The assumptions and calculations provided herein represent a preliminary step in understanding the complex dynamics of CI/CD pipeline costs. Further empirical data and real-world usage metrics would be invaluable for validating these findings and offering a more nuanced perspective.

7.2 Features

This thesis presents a comprehensive analysis of Bitbucket Pipelines and its capability to replace the current Jenkins deployment pipelines. While Bitbucket Pipelines covers most essential features, there are some significant differences that SST Digi, and other organizations considering the switch, should be aware of. This section goes over the most important features that were found to be missing or lacking in Bitbucket Pipelines.

Dynamism

One of the standout limitations of Bitbucket Pipelines is its lack of support for dynamic step executions, a feature that is readily available in Jenkins. Jenkins provides the flexibility to dynamically generate steps or stages, allowing the deployment of projects unknown to the pipeline code by utilizing user-provided parameters to create the necessary deployment steps on-the-fly. Below is an example of a generalized scripted pipeline code in Jenkins where steps are dynamically generated based on user input:

```
node {
  stage('Build') {
    // Build steps here...
  }

  for (project in projectsToDeploy) {
    stage("Deploy ${project}") {
      // Deployment steps that use the 'project' variable
      // to determine what to deploy
      sh "deploy_project.sh ${project}"
    }
  }

  stage('Post-deployment') {
```

```
    // Post-deployment steps here...
  }
}
```

Unfortunately, this level of dynamism is absent in Bitbucket Pipelines, as it requires all potential steps to be predefined in the pipeline configuration. Although this dynamic feature in Jenkins is infrequently used in the current deployment scenario at SST Digi, its absence in Bitbucket Pipelines is worth mentioning.

Parallelization

Bitbucket Pipelines introduces constraints in terms of parallelization. Unlike Jenkins, which allows for parallel stages and nested parallel steps, Bitbucket Pipelines does not support these features. This limitation complicates the task of orchestrating complex deployment sequences and necessitates assigning multiple projects to a single pipeline step, a task that is more straightforwardly handled in Jenkins due to its support for nested parallelism.

Modularity

The approach to modular design also differs significantly between Bitbucket Pipelines and Jenkins. In Jenkins, a pipeline can invoke another job, and this invoked job seamlessly integrates with the primary pipeline. This paradigm encourages modularity and is taken advantage of in the current Jenkins setup at SST Digi. Bitbucket Pipelines, on the other hand, does not naturally align with this design philosophy. While it is technically possible for one Bitbucket pipeline to trigger another, this is not the recommended practice for achieving modularity. Instead, Bitbucket Pipelines encourages the use of pipes [42], Docker images, and external scripts to build modular and reusable components. This shift in approach represents a significant departure from Jenkins and is an important consideration for organizations looking to migrate their deployment pipelines to Bitbucket Pipelines.

7.3 Performance

In the course of evaluating the performance of Bitbucket Pipelines against Jenkins, the efficiency of each tool was assessed through a practical deployment exercise. This involved the replication of a pre-existing Jenkins deployment pipeline within Bitbucket Pipelines as explained in the 6th chapter, with the objective of deploying a nine-project IoT platform onto AWS. The comparative analysis is based on empirical data gathered from ten iterations of a fresh deployment by both Jenkins and Bitbucket Pipelines.

The results are indicative of a performance edge for Bitbucket Pipelines, which completed the deployment process in an average time of 3 hours and 10 minutes. This contrasts with Jenkins, which averaged at 4 hours for the same task. However, attributing this performance differential to the intrinsic efficiency of Bitbucket Pipelines alone would be an oversimplification. The absence of disclosed specifications regarding the computational resources allocated by Atlassian for Bitbucket Pipelines introduces a variable that is beyond the scope of this study to quantify. Conversely, the Jenkins environment is self-hosted, thus making it possible to directly correlate its performance with the specifics of its hosting infrastructure.

This comparison also brings to light the divergent methodologies each tool employs to achieve the same end — the full deployment of the IoT platform. Bitbucket Pipelines adopts an approach where multiple projects may be deployed concurrently within a single build container. Jenkins, on the other hand, dedicates a distinct job for each project. While both strategies ultimately realize the deployment objective, they represent fundamentally different orchestration philosophies that could contribute to the observed disparities in deployment times.

The findings suggest that for the specific deployment requirements and current infrastructure, Bitbucket Pipelines offers a time efficiency advantage over Jenkins. However, It must be acknowledged that these findings are as much a reflection of the execution environment and the specific pipeline configuration and programming as they are of the tools in question. The advantage observed here may not universally apply to all deployment scenarios and should be weighed against other factors such as ease of use, cost, and organizational needs before making an informed decision on the migration from Jenkins to Bitbucket Pipelines.

In addition to the time efficiency considerations, memory allocation is another critical aspect of performance that must be factored into the comparative analysis. As detailed in table 4.1, Bitbucket Pipelines permits a maximum of 8GB of memory for its build containers. This stands in contrast to Jenkins, which is not bound by such limitations and instead is only restricted by the resources allocated to it. Such a disparity in memory availability became pertinent during an extensive flyway migration task; Bitbucket Pipelines encountered limitations, leading to process termination due to insufficient memory. Jenkins, with access to 32GB of memory provided by the AWS infrastructure in use, completed the same task without complication. While this instance highlights a potential bottleneck in Bitbucket Pipelines, it is worth noting that builds requiring more than 8GB of memory are the exception rather than the rule, and for the majority of tasks, Bitbucket Pipelines' memory provision has proven to be adequate. Nevertheless, this does emphasize the importance of understanding the memory demands of specific deployment processes when considering a transition between these tools.

8 Conclusion and Future Work

The analysis presented in this thesis offers a comprehensive evaluation of two widely utilized CI/CD platforms: Jenkins and Bitbucket Pipelines. By replicating an existing deployment pipeline for an IoT platform on AWS from Jenkins to Bitbucket Pipelines, this case study provides insight into the technical, financial, and performance-related aspects of both systems.

The investigation finds that while Jenkins boasts a more cost-effective computational model, it also requires considerable personnel involvement for infrastructure maintenance. This stands in contrast to Bitbucket Pipiles, which, albeit at a higher computational cost, offers a reduction in operational complexity due to its managed service nature, thus eliminating the personnel overhead associated with maintenance tasks. The technical comparison reveals certain limitations in Bitbucket Pipelines, such as a lack of dynamic step execution and constrained parallelization, which could impact complex deployment workflows. Despite these constraints, Bitbucket Pipelines generally offers faster deployment times, though it can be limited by memory capacity in certain scenarios.

In conclusion, the switch from Jenkins to Bitbucket Pipelines presents a trade-off between the adaptability and resource control offered by Jenkins and the simplified management and potential cost savings with Bitbucket Pipelines. Organizations must therefore weigh these factors carefully against their specific requirements and constraints.

For future work, enhancing security through the implementation of a private bastion host for SSH tunneling from Bitbucket Pipelines to AWS is proposed. This would mitigate certain risks by establishing a secure and controlled point of access between the CI/CD pipeline and the cloud environment. Additionally, further research could be directed toward optimizing pipeline configurations to exploit the pipes feature of Bitbucket Pipelines, as well as extending the evaluation to include other containerized CI/CD tools, providing a broader context for the current findings.

The ongoing evolution of both Jenkins and Bitbucket Pipelines suggests that any long-term CI/CD strategy should be flexible and adaptable to change. With the continuous introduction of new features and optimizations, the dynamic landscape of CI/CD tools will require organizations to remain vigilant and responsive to technological advancements. Thus, this thesis lays the foundation for ongoing assessment and decision-making in the selection and implementation of CI/CD solutions.

References

[1] *Kalmar Insight*. Kalmar Global. 2023. URL: https://www.kalmarglobal.com/equipment-services/kalmar-insight/ (visited on 10/27/2023).

[2] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. eng. Hoboken: Pearson Education, Limited, 2010. ISBN: 9780321601919.

[3] Brent Laster. *Jenkins 2: Up and Running: Evolve Your Deployment Pipeline for Next Generation Automation*. eng. First edition. Beijing: O'Reilly, 2018. ISBN: 1-4919-7958-5.

[4] Paul M Duvall, Andrew Glover, and Steve Matyas. *Continuous Integration: Improving Software Quality and Reducing Risk*. eng. Addison-Wesley Signature Series. Addison-Wesley Professional, 2007. ISBN: 0321336380.

[5] Nicole Forsgren, Jez Humble, and Gene Kim. *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*. eng. 1st. Portland, OR: IT Revolution Press, 2018. ISBN: 1942788339.

[6] Bogdan Vasilescu et al. "Quality and productivity outcomes relating to continuous integration in GitHub". In: Aug. 2015, pp. 805–816. DOI: 10.1145/2786805.2786850.

[7] Lianping Chen. "Continuous Delivery: Huge Benefits, but Challenges Too". In: *IEEE Software* 32.2 (2015), pp. 50–54. ISSN: 0740-7459.

[8] Michael Hilton et al. "Trade-Offs in Continuous Integration: Assurance, Security, and Flexibility". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017, pp. 197–207. ISBN: 9781450351058. DOI: 10.1145/3106237.3106270. URL: https://doi.org/10.1145/3106237.3106270.

[9] Robin Bolscher and Maya Daneva. "Designing Software Architecture to Support Continuous Delivery and DevOps: A Systematic Literature Review". In: May 2019. DOI: 10.5220/0007837000270039.

[10] Yuqing Wang et al. "Test automation maturity improves product quality—Quantitative study of open source projects using continuous integration". In: *Journal of Systems and Software* 188 (2022), p. 111259. ISSN: 0164-1212. DOI: 10.1016/j.jss.2022.111259. URL: https://www.sciencedirect.com/science/article/pii/S0164121222000280.

[11] *Jenkins*. n.d. URL: https://www.jenkins.io/ (visited on 08/01/2023).

[12] Martin Heller. “What is Jenkins? The CI server explained”. In: *InfoWorld.com* (2020).

[13] Jenkins Developers. *Jenkins Pipeline Documentation*. n.d. URL: https://www.jenkins.io/doc/book/pipeline/ (visited on 07/27/2023).

[14] Ádám Révész and Norbert Pataki. “Visualisation of Jenkins Pipelines”. In: *Acta cybernetica (Szeged)* 25.2 (2021), pp. 877–895. ISSN: 0324-721X.

[15] Pranoday Pramod Dingare. *CI/CD pipeline using Jenkins Unleashed : solutions while setting up CI/CD processes*. Berkeley, California: Apress, 2022. ISBN: 1-4842-7508-X.

[16] Pooya Rostami Mazrae et al. “On the usage, co-usage and migration of CI/CD tools: A qualitative analysis”. In: *Empirical Software Engineering* 28 (Mar. 2023). DOI: 10.1007/s10664-022-10285-5.

[17] *Bitbucket Pipelines - Continuous Delivery*. URL: https://bitbucket.org/product/features/pipelines (visited on 09/26/2023).

[18] Use Docker Images as Build Environments. *Use Docker Images as Build Environments*. n.d. URL: https://support.atlassian.com/bitbucket-cloud/docs/use-docker-images-as-build-environments/ (visited on 07/26/2023).

[19] Sneh Pandya and Riya Guha Thakurta. *Introduction to Infrastructure as Code: A Brief Guide to the Future of DevOps*. English. New York, New York: Apress L. P., 2022. ISBN: 1-4842-8689-8.

[20] Bitbucket Pipelines Configuration Reference. *Bitbucket Pipelines Configuration Reference*. n.d. URL: https://support.atlassian.com/bitbucket-cloud/docs/bitbucket-pipelines-configuration-reference/ (visited on 07/29/2023).

[21] Paul Krill. “Bitbucket Pipelines: Continuous Delivery in the Cloud”. English. In: *InfoWorld.com* (2016).

[22] Parallel Step Options. *Parallel Step Options*. n.d. URL: https://support.atlassian.com/bitbucket-cloud/docs/parallel-step-options/ (visited on 07/30/2023).

[23] Atlassian. *Databases and service containers*. n.d. URL: https://support.atlassian.com/bitbucket-cloud/docs/databases-and-service-containers/ (visited on 05/09/2022).

[24] *Bitbucket Pipelines Integrations*. URL: https://bitbucket.org/product/features/pipelines/integrations (visited on 09/26/2023).

[25] Justin Ellingwood. *CI/CD Comparison: Using Managed Providers vs. Self-Hosting*. June 2018. URL: https://www.digitalocean.com/community/tutorials/ci-cd-comparison-using-managed-providers-vs-self-hosting (visited on 07/26/2023).

[26] Bitbucket Pipelines. *Bitbucket Pipelines*. n.d. URL: https://bitbucket.org/product/features/pipelines (visited on 07/26/2023).

[27] Databases and Service Containers. *Databases and Service Containers*. n.d. URL: https://support.atlassian.com/bitbucket-cloud/docs/databases-and-service-containers/ (visited on 07/26/2023).

[28] Pipeline Syntax. *Pipeline Syntax*. n.d. URL: https://www.jenkins.io/doc/book/pipeline/syntax/ (visited on 07/29/2023).

[29] YAML.org. *YAML*. n.d. URL: https://yaml.org/ (visited on 08/02/2023).

[30] Step Options. *Step Options*. n.d. URL: https://support.atlassian.com/bitbucket-cloud/docs/step-options/ (visited on 08/01/2023).

[31] *Bitbucket Cloud Documentation - Stage Options*. n.d. URL: https://support.atlassian.com/bitbucket-cloud/docs/stage-options/ (visited on 07/30/2023).

[32] Liam Newman. *Converting Conditional to Pipeline*. Jan. 2017. URL: https://www.jenkins.io/blog/2017/01/19/converting-conditional-to-pipeline/ (visited on 07/27/2023).

[33] Bitbucket Pricing. *Bitbucket Pricing*. n.d. URL: https://www.atlassian.com/software/bitbucket/pricing (visited on 07/24/2023).

[34] Scaling. *Scaling*. n.d. URL: https://www.jenkins.io/doc/book/scaling/ (visited on 07/24/2023).

[35] Inc. Red Hat. *Ansible*. 2023. URL: https://www.ansible.com (visited on 11/07/2023).

[36] Inc. Amazon Web Services. *AWS SDK for Python (Boto3)*. 2023. URL: https://aws.amazon.com/sdk-for-python/ (visited on 11/07/2023).

[37] Inc. Amazon Web Services. *AWS Command Line Interface*. 2023. URL: https://aws.amazon.com/cli/ (visited on 11/07/2023).

[38] Inc. Amazon Web Services. *AWS Cloud Development Kit*. 2023. URL: https://aws.amazon.com/cdk/ (visited on 11/07/2023).

[39] Redgate Software. *Flyway*. 2023. URL: https://flywaydb.org (visited on 11/07/2023).

[40] Inc. Amazon Web Services. *Access a bastion host by using Session Manager and Amazon EC2 Instance Connect - AWS Prescriptive Guidance*. 2023. URL: https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/access-a-bastion-host-by-using-session-manager-and-amazon-ec2-instance-connect.html (visited on 11/07/2023).

[41] G. Vijayababu, D. Haritha, and R. Satya Prasad. "An effective utilization of bastion host services in cloud environment". eng. In: *International journal of innovative technology and exploring engineering* 8.7 (2019), pp. 2215–2220. ISSN: 2278-3075.

[42] Bitbucket. *Bitbucket Pipelines Integrations*. Bitbucket. Accessed 2023. URL: https://bitbucket.org/product/features/pipelines/integrations (visited on 10/23/2023).