DEPARTMENT OF
COMPUTER SCIENCE

ALEXANDRE RODRIGUES OLIVEIRA
BSc in Computer Science and Engineering

DUAL-MODELING APPROACHES IN CI/CD

AN AGNOSTIC MODELING FRAMEWORK TO STREAMLINE DEVOPS

MASTER IN STUDY PROGRAM NAME

NOVA University Lisbon
month, year

DEPARTMENT OF
COMPUTER SCIENCE

DUAL-MODELING APPROACHES IN CI/CD

AN AGNOSTIC MODELING FRAMEWORK TO STREAMLINE DEVOPS

ALEXANDRE RODRIGUES OLIVEIRA

BSc in Computer Science and Engineering

Adviser: Vasco Amaral
Associate Professor, NOVA University Lisbon

Co-adviser: Jácome Cunha
Associate Professor, Faculty of Engineering - University of Porto

Examination Committee

Chair: Name of the committee chairperson
Full Professor, FCT-NOVA

Rapporteur: Name of a rapporteur
Associate Professor, Another University

Members: Another member of the committee
Full Professor, Another University

Yet another member of the committee
Assistant Professor, Another University

MASTER IN STUDY PROGRAM NAME

NOVA University Lisbon
month, year

Dual-Modeling Approaches in CI/CD

An Agnostic Modeling Framework to Streamline DevOps

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.1.27) [52].

To my family and friends, the core of my life.

Acknowledgements

I want to express my gratitude to Professor Vasco and Professor Jácome for their dedication, availability, and guidance in helping me prepare and elaborate this thesis. Their knowledge and experience in technological advances and academic research were essential in completing this project. I would also like to thank the FCT and the Department of Computer Science for five years of personal and academic development, which will undoubtedly continue to shape future generations. Finally, I extend a heartfelt thank you to my parents, family, and friends who have always supported me. Without them, I would not have been able to complete this chapter.

"Inveniam viam aut faciam"

Abstract

The widespread adoption of Continuous Integration and Continuous Deployment (CI/CD) practices within DevOps has revolutionized software delivery. However, the lack of standardized tools for CI/CD environments frequently leads to a cumbersome experience, where each pipeline is a kind of complex setup that frequently leads to bottlenecks, delays, and errors, hindering technical and non-technical users. Platform-specific tools further complicate migration between CI/CD technologies. This research addresses these challenges by developing a dual modeling solution based on a generic metamodel, designed to streamline the setup, evolution, and migration of CI/CD pipelines. Through the integration of graphical and textual modeling methods, this study closes the skill gap between technical and non-technical users.

The experimental evaluation revealed that our Textual tool demonstrated superior usability with an 85.2 SUS score, outperforming visual tools like Buddy, scoring 79.8. Our Visual tool, valuable for users favoring a graphical interface, scored 75.6 and exhibited mixed user experiences, particularly among those unfamiliar with visual modeling. The study identified an inverse correlation between usability and workload, highlighting the trade-offs between ease of use and cognitive load.

These findings underscore the importance of offering both approaches in CI/CD tools to cater to diverse user preferences and enhance software delivery efficiency.

Keywords: Continuous integration, Continuous deployment, CI/CD Pipelines, DevOps, Modeling solution, Standardization, Software engineering

Resumo

A adoção generalizada das práticas de Integração Contínua e Entrega Contínua (CI/CD) no âmbito do DevOps revolucionou a entrega de software. No entanto, a falta de ferramentas padronizadas para ambientes de CI/CD conduz frequentemente a uma experiência complicada, em que cada pipeline é uma espécie de configuração complexa que conduz frequentemente a estrangulamentos, atrasos e erros, prejudicando os utilizadores técnicos e não técnicos. As ferramentas específicas de cada plataforma complicam ainda mais a migração entre tecnologias de CI/CD. Esta investigação aborda estes desafios através do desenvolvimento de uma solução de modelação dupla baseada num metamodelo genérico, concebida para simplificar a configuração, evolução e migração de condutas de CI/CD. Através da integração de métodos de modelação gráfica e textual, este estudo colmata a lacuna de competências entre utilizadores técnicos e não técnicos.

A avaliação experimental revelou que a nossa ferramenta textual demonstrou uma usabilidade superior com uma pontuação SUS de 85,2, superando ferramentas visuais como o Buddy, com uma pontuação de 79,8. A nossa ferramenta Visual, valiosa para os utilizadores que preferem uma interface gráfica, obteve uma pontuação de 75,6 e apresentou experiências de utilização mistas, particularmente entre os que não estão familiarizados com a modelação visual. O estudo identificou uma correlação inversa entre a usabilidade e a carga de trabalho, destacando as compensações entre a facilidade de utilização e a carga cognitiva.

Estas conclusões sublinham a importância de oferecer ambas as abordagens nas ferramentas de CI/CD para satisfazer as diversas preferências dos utilizadores e melhorar a eficiência da entrega de software.

Palavras-chave: Integração contínua, Implementação contínua, Pipelines CI/CD, DevOps, Modelação de soluções, Padronizaçao, Engenharia de software

Contents

List of Figures

List of Tables

Listings

Acronyms

AQL	Acceleo Query Language *(p. 16)*
ATL	Atlas Transformation Language *(p. 16)*
CI/CD	Continuous Integration and Continuous Deployment *(p. 1)*
CSE	Continuous Software Engineering *(p. 1)*
DSLs	Domain-specific modelling languages *(p. 11)*
DSR	Design science research *(p. 3)*
EMF	Eclipse Modelling Framework *(p. 15)*
ETL	Extract, Transform, and Load *(p. 15)*
FEUP	Faculty of Engineering of the University of Porto *(p. 4)*
GPLs	General-purpose modelling languages *(p. 11)*
IFML	Interaction Flow Modeling Language *(p. 23)*
IME	Integrated modeling Environment *(p. 22)*
M2M	Model-to-model *(p. 12)*
M2T	Model-to-text *(p. 12)*
MDD	Model-Driven Development *(p. 23)*
MDE	Model-Driven Engineering *(p. 8)*
ML	Machine Learning *(p. 23)*
MLSs	Modelling Language Suites *(p. 11)*
OCL	Object Constraint Language *(p. 34)*
PI	Platform-independent *(p. 28)*

PS	Platform-specific *(p. 28)*
PSL	Purpose-Specific Language *(p. 22)*
QVT	Query/View/Transformation *(p. 16)*
SUS	System Usability Scale *(p. 60)*
TLX	NASA-Task Load Index *(p. 61)*
UML	Unified Modeling Language *(p. 11)*
XMI	XML Metadata Interchange *(p. 44)*

1 Introduction

This chapter provides an introduction to the research thesis, starting with its context and following with motivation, contributions and objectives.

1.1 Context

In the ever-evolving landscape of software development, the integration of Continuous Integration and Continuous Deployment (CI/CD) practices have emerged as a fundamental cornerstone, revolutionizing how organizations deliver software [20, 76, 64, 33, 77, 32, 69, 45]. In the dynamic realm of DevOps, where agility and collaboration between development and operations teams are paramount, CI/CD has become an essential paradigm, enabling faster, more reliable, and more efficient software delivery processes [76, 33, 77, 32, 17, 4, 69, 45, 6, 8, 50], in contrast to the traditional approaches where those teams and phases were separated in the project lifecycle and to achieve the same quality of product, it would be a much more lengthy process. The magnitude of this significance in the contemporary technological sphere cannot be overstated, as it accelerates the pace of development and enhances the overall quality of software products. This widespread adoption of DevOps has led to a proliferation of technological solutions to meet the massive market demands and support Continuous Software Engineering (CSE) processes [20, 45].

The absence of standardized tools for CI/CD pipelines has brought about substantial challenges for organizations. This lack of standardization has entangled organizations in the complexities of managing and monitoring their CI/CD pipelines, resulting in bottlenecks, delays, and errors in software delivery. The IT market, valued at nearly $9.4 billion worldwide [31], is grappling with maintenance costs, with a significant portion attributed to emergencies, unplanned work, and changes. These issues often stem from conflicts between the goals of IT and development teams [31].

In this context, bridging the gap between the standardization of modeling methodologies and tools designed expressly for CI/CD pipelines is crucial as the development of new tools and their adoption is becoming increasingly diverse and fast.

1.2 Motivation

The market's lack of standardization in the CI/CD tools market creates substantial obstacles to effective CI/CD pipeline implementation and management. This is because the existing situation encourages technology silos, with professionals specializing in one CI/CD platform and not knowing how to utilize another due to their inherent complexity. As a result, CI/CD technology migration becomes challenging when they need to use different technologies. Furthermore, those familiar with one platform are more likely to use it for future projects as a kind of customer fidelity, regardless of the benefits or drawbacks that may emerge from unfamiliarity with other platforms.

The vast options of tools with distinct syntaxes and integrations within the same environment also hinder the efficiency, adaptability, and scalability of CI/CD pipelines. With the constant evolution of this environment, organizations require a framework to automate this migration and cover most technologies. This disconnect in shared goals and the vast array of technologies adopted impedes the team's collaboration, leading to software delays [45].

Some effort has been made to achieve this modeling, but it is still in the development phase, and many solutions oversee technological collaboration [35, 1]. Reaching this standardization and easing migration between diverse CI/CD platforms remains a complex challenge for organizations aspiring to thrive in the ever-evolving landscape of software engineering.

Our main challenge of this work is to **develop a unified modeling solution that addresses the lack of standardization in CI/CD tools, simplifies technology migration, and caters to the diverse skill sets of technical and non-technical users, ultimately enhancing software delivery efficiency**.

1.3 Objectives

In pursuit of this research aim, the subsequent research objectives are delineated, providing a clear roadmap to systematically address specific facets of the CI/CD modeling solution development:

- Develop a modeling solution for CI/CD pipelines, capturing the key stages and elements involved in the software delivery pipelines.
- Explore non-uniformities in existing CI/CD tooling and develop strategies to ensure integration within the modeling framework.
- Examine the impact of different configurations of the proposed modeling solution on efficiency and user usability.
- Evaluate the influence of errors or inconsistencies during model transformation configurations and implementation on the reliability of the modeling solution.

Overall, this research will contribute significantly to the field of CI/CD modeling by providing valuable insights and practical solutions that will lead to faster delivery of high-quality software products.

1.4 Contributions

To achieve the defined objectives, this thesis is aimed at addressing existing challenges in CI/CD processes and improving the overall efficiency and interoperability of pipeline configuration. The specific contributions of this work are outlined as follows:

- Development of a Unified Modeling Framework directed towards standardizing CI/CD pipeline modeling.
- Design and implementation of a CI/CD technology independent Metamodel.
- Building of a Graphical and Textual Modeling solution to bridge the gap between technical and non-technical users.
- Facilitation of Platform Migration by providing platform transformations and platform-specific code generation.
- Usability evaluation from industry practitioners.

1.5 Methodology

This work follows Design science research (DSR) [34], a rigorous and iterative methodology for developing and validating a practical problem. The procedure consists of the following stages:

Problem Awareness - The growing adoption of CI/CD practices has highlighted the need for standardized modeling techniques and specific tools tailored for this domain. Current modeling approaches, often specific or inadequate, pose challenges such as collaboration, heterogeneous tools and skill gaps between Development and Operation teams.

Suggestion - The fundamental basis of the proposed solution is a textual and graphical approach to mitigate the skill gap between technical and non-technical users. This way, the modeling solution will utilize graphical notations and symbols to create visually appealing and easy-to-understand models, enabling clear communication and collaboration among stakeholders.

Development - The pipeline elements for the graphical approach and the custom DSL for the textual approach will be connected using the already tested pipeline metamodel. The next step is to generate deployable code on various DevOps platforms with existing transformation tools. Practitioners from the industry will be involved in experiments to ensure that the modeling solution meets real-world requirements.

Evaluation - Through a series of usability experiments derived from the requirements analyzed, together with documentation, the researcher will test and validate the solution.

1.6 Institutional Context

This study involves a collaboration between two entities: the Faculty of Engineering of the University of Porto (FEUP) and Nova LINCS, a Computer Science research laboratory located in NOVA FCT.

1.7 Document Structure

The remainder of this document is organized as follows:

- Chapter 2 - **Background**: Introduction to some of the key concepts of this thesis and their relevance and challenges encountered;
- Chapter 3 - **Related Work**: Review existing CI/CD pipeline modeling literature and analysis of available modeling tools and their limitations;
- Chapter 4 - **Model-driven Languages**: Architecture overview, identification of requirements, design principles, and best practices for building the modeling solution;
- Chapter 5 - **Evaluation**: Experiments to evaluate usability and effectiveness of the developed tool, alongside results interpretations and their implications with threats to validity;
- Chapter 6 - **Conclusion**: Presentation of this study's conclusions, its limitations and recommendations for future research;

2

Background

This chapter introduces the main concepts applied throughout this document. The first section introduces DevOps and the notion of pipelines, followed by Model-Driven Engineering and its fundamental components, modeling methodologies, and tools. Finally, we discuss multiple challenges in modeling CI/CD workflows.

2.1 DevOps: A Collaborative Approach for Agile Software Delivery

The DevOps definition in software development and IT operations is evolving to meet new technological and organizational needs, making it a challenge to define. According to Willis, Humble, and Kim in [46], the notion of DevOps is continually changing due to various interpretations across industries and organizational cultures [5].

DevOps emerged in the early 2000s alongside the Agile Manifesto, recognizing the need for closer collaboration between development and operations teams to address software development complexities and meet demanding release cycles while maintaining quality [6]. The term itself can be stated as a combination of Development and Operations teams [41].

DevOps is a set of tools and practices that automate and integrate software development and IT operations teams [8]. It also provides a solution by breaking down silos and encouraging a cultural shift emphasizing collaboration, automation, and shared responsibility for software delivery.

According to Puppet and CircleCI's 2021 State of DevOps report [58], more than 90% institutions reported employing DevOps practices to some level. Furthermore, the survey stated that high-performing DevOps teams deploy code 208 times more frequently than low-performing teams, with change lead times being 106 times faster.

The Upskilling 2021 Enterprise DevOps Capabilities Report [40] also emphasized the growing relevance of DevOps capabilities across organizations. Over 50% of respondents said DevOps skills are critical for their transformation activities, and roughly 80% said there is a shortage of these capabilities within their teams.

2.1.1 Orchestrating Software Delivery: The Role of CI/CD Pipelines

To achieve maximum agility in the DevOps culture, the DevOps pipeline needs available IT infrastructure to run and test existing code. This requires an automated CI/CD workflow [6]. A DevOps pipeline consists of automatic processes that enable teams to automate and streamline the software development and deployment process [5]. It is the backbone of the DevOps philosophy [6], allowing for a seamless flow of code changes from development to production. We will only consider the DevOps pipeline core, CI and CD, for the scope of this thesis since our goal is to address only these challenges.

Fowler introduced the concept of CI [29], which refers to the automated and frequent integration of code changes into a shared repository. This includes creating and testing changes in a version control system, identifying integration issues, and reducing conflicts between developers' code [72, 73, 4, 25, 6]. This continuous improvement cycle produces a stable, high-quality codebase while decreasing the likelihood of issues surfacing later on [5, 69].

CD is an extension of CI that was later introduced by Humble and Farley [36]. It automates code delivery and deployment to production environments after successful CI testing [36, 6]. Skelton and O'Dell [71] suggest it ensures a deployable state throughout the process, from code, to commit to production, allowing for frequent and reliable releases with minimal manual intervention and risk. Additionally, CD fosters continuous feedback loops with users, enabling early issue identification and continuous improvement, ultimately reducing production costs [5, 69].

Together, CI/CD creates a seamless flow in the DevOps pipeline [4], ensuring extensive testing, validation, and easy deployment of code changes. This feedback loop allows faster iterations and improvements while maintaining software stability, dependability, and productivity.

Figure 2.1 depicts a high-level CI/CD pipeline perspective, where CI and CD are represented as an interconnected loop. The left side of the loop depicts the CI process, which involves stages like Plan, Code, Build, and Test, ensuring continuous validation and integration of new code into the main branch without causing functionality breakage. The CD process (right side) involves stages like Release, Deploy, Operate, and Monitor. After integration, code is automatically or semi-automatically deployed to production, and monitored for performance and potential issues. Continuous feedback from these phases is used to improve the development process.

Figure 2.2 outlines our proposed Feature Model for each CI/CD phase based on widely recognized best practices and industry standards [5, 18, 19, 78, 19, 84, 81, 82, 83, 78, 17]. In the CI phase, Figure 2.2 introduces extra steps like Code Analysis, where automated tools evaluate the code quality before moving to the next stages. There is also an optional focus on Unit Testing and Code Coverage, ensuring that individual components are thoroughly tested and total test coverage is measured. These steps go beyond the fundamental "Test" phase mentioned in Figure 2.1, providing more granular

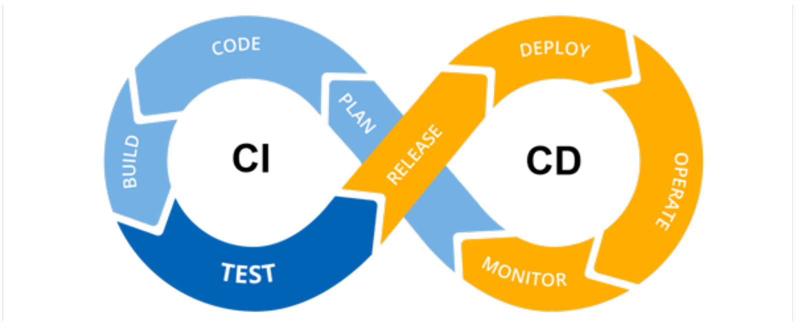


Figure 2.1: CI CD pipeline

code validation. Additionally, Compile Dependencies and Parallel Deployments are introduced, allowing for better management of external resources and testing the code in multiple environments simultaneously.

On the CD side, Figure 2.2 breaks down the deployment process with stages like Deployment Automation, ensuring that the release process is fully automated, and Quality Assurance, where additional checks verify the reliability of the deployment. Strategies like Blue-Green Deployment are also introduced to minimize downtime by maintaining two environments. One environment (e.g., Blue) runs the current application version, while the other (e.g., Green) is updated. When the updated environment is confirmed to work correctly, traffic is switched to it, minimizing disruption. In the Staging phase, the model introduces 1 Box step, testing the deployment in a smaller, isolated environment before being rolled out to production. After deployment, Analysis is introduced as an additional post-production step to assess performance and gather insights, beyond just monitoring the system for issues. The features with the highest presence in enterprise pipelines are labeled as "Mandatory" and the remaining as "Optional".

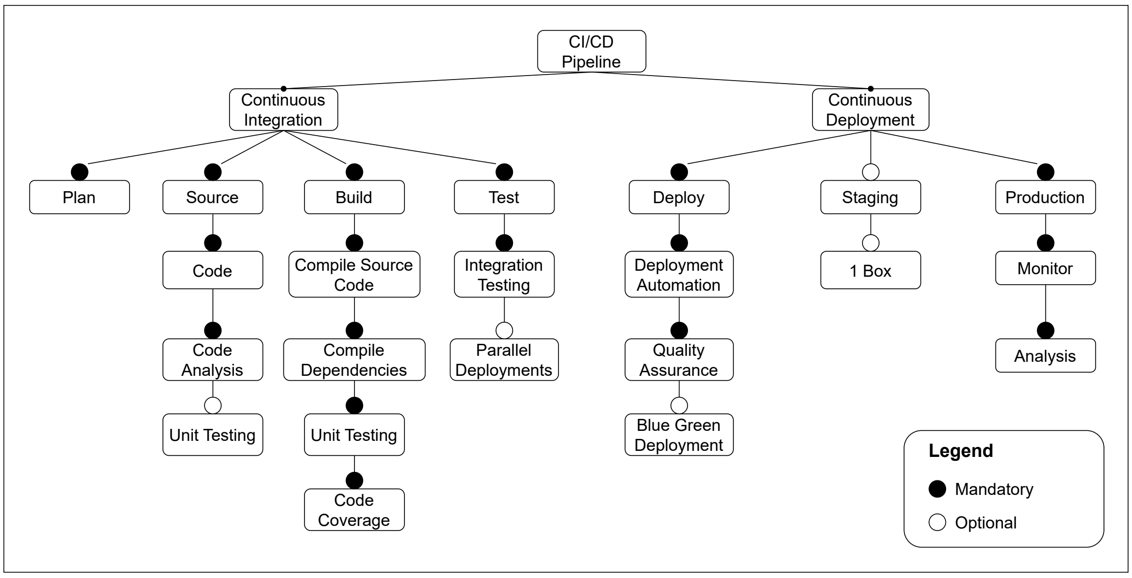


Figure 2.2: CI/CD Pipeline Feature Model

Table 2.1 compares the technologies widely embraced by the community and organizations. Tools with high scalability, parallel processing capabilities, flexible configuration options, and strong management features are most widely embraced by both the developer community and organizations, catering to the diverse needs of different project sizes and complexities. Information was gathered via the community, official documentation, and a survey of published literature [47, 3, 75, 49, 70]. Some notes regarding table features: (1) 'Management' refers to the management of roles and permissions; (2) 'Parallelism' means that tasks can be run concurrently on the same machine.

2.2 Model-Driven Engineering

Unlike traditional approaches that rely primarily on code, Model-Driven Engineering (MDE) places models as fundamental artifacts to capture the system's requirements, design, and behavior, and transform them into the driving force behind the entire development lifecycle. MDE seeks to increase productivity and reduce time-to-market by allowing for greater abstraction and leveraging concepts closer to the issue area rather than programming languages [68]. This shift in paradigm aims to achieve significant benefits, including:

- **Improved Quality:** MDE promotes early and continuous analysis of models, catching errors and inconsistencies early on, leading to a more robust and maintainable system.
- **Enhanced Reusability:** Models can be reused across different projects, reducing development time and ensuring consistency in system architecture.
- **Increased Automation:** MDE leverages transformations and automated processes to convert models, enabling efficient code generation and validation.

The following subsections delve into key aspects of MDE.

2.2.1 Abstraction: Distilling the Essence of Complexity

Abstraction in software engineering is the process of simplified system representations that capture the essential structure, behavior, and system relationships without getting overwhelmed by the complexity of implementation details. The human mind is naturally oriented to abstraction, as it allows us to generalize complex concepts clearly and concisely and also communicate them to others [61], even if they do not have a deep understanding of the underlying details [10]. These abstractions, known as models, serve as blueprints for understanding and building software systems.

CI/CD Tool	**Support/SLA**	**Distributed Builds**	**Scalability**	**Reports**	**Integration Ecosystem**	**Pipeline Configuration**	**Management**	**Build Pipelines**	**Parallelism**	**Migration**
TeamCity	Yes	N/A	High	Yes	High	Groovy, Kotlin, YAML	Yes	Yes	Yes	Yes
Bamboo	Yes	Yes	Moderate	Yes	Moderate	Groovy	Yes	Yes	Yes	Yes
Azure DevOps	Yes	Yes	High	Yes	High	YAML, C#, TypeScript	Yes	Yes	Yes	Yes
AWS CodePipeline	Yes	N/A	Moderate	Yes	Moderate	YAML	No	Yes	N/A	No
GitHub Actions	Yes	N/A	High	N/A	High	YAML, JavaScript, Python	N/A	Yes	Yes	Yes
Travis CI	Yes	N/A	Moderate	Yes (partial)	Moderate	YAML	N/A	No	Yes	No
AppVeyor	Yes	Yes	Moderate	Yes	High	YAML, PowerShell	Yes	Yes	Yes	Yes
CircleCI	Yes	Yes	Moderate	Yes	Low	YAML, Python, Go	No	Yes	Yes	Yes
Concourse	Yes	Yes	Moderate	Yes	Low	YAML	Yes	Yes	Yes	Yes
Drone	N/A	Yes	High	N/A	High	YAML	Yes	Yes	Yes	Yes
GitLab	Yes	Yes	High	Yes	High	YAML, .gitlab-ci.yml	Yes	Yes	Yes	Yes
Jenkins	No	Yes (partial)	High	Yes	High	Groovy	No	Yes	Yes (partial)	Yes
GoCD	Yes	Yes	High	Yes	High	YAML, JSON	Yes	Yes	Yes	Yes

Table 2.1: CI/CD Tools Comparison

2.2.2 Models: Capturing the Essence in a Unified Language

According to [65], "a model is an abstraction of a system often used to replace the system under study, representing a partial and simplified view of a system, making it a design artifact of a simplified system representation [7, 54]. In software engineering, models are language-based entities specified by DSLs with precise syntax and semantics [42]. They evolve as the system design progresses and incorporate fundamental properties or behaviors of the subject they represent.

Models may reveal a system's static and dynamic dimensions, with static models focusing on static components and dynamic models depicting system behavior through operations, algorithms, collaborations, and internal state changes [10]. To better understand the roles and types of models, they can be classified into the following categories [10]:

- **Descriptive:** Describe an existing system behavior
- **Prescriptive:** Arbitrate scope and future system details
- **Definition:** Define system implementation

Additionally, models can be further abstracted based on their focus on the following aspects:

- **Reduction:** Only represent a subset of the original object's attributes
- **Mapping:** Abstract and generalize an original object that represents a category of individuals

These concepts help technical and non-technical stakeholders share a common vision and understanding, improving communication. Furthermore, models assist project planning by providing a more realistic picture of the system to be built and enabling project control based on objective criteria.

2.2.3 Modeling Languages: UML as a Lingua Franca of MDE

MDE and modeling languages are inextricably linked, forming the foundation for a structured and efficient approach to software development. As previously stated, MDE elevates models to the central role in the development process while modeling languages provide the tools to create and manipulate these models effectively.

To achieve this, modeling languages often include a set of modeling concepts and rules that define the syntax and semantics of the language, providing a standardized approach for creating models that can be used across several software development tools and platforms [10]. Based on [61], the success of a modeling language depends on its capacity to offer users relevant domain concepts through primitives that can directly express these notions.

There are two main types of modeling languages: General-purpose modelling languages (GPLs) and Domain-specific modelling languages (DSLs) [10]. GPLs, such as the Unified Modeling Language (UML) and SysML, are standardized languages that model a wide variety of systems or domains and capture different aspects of the system, including requirements, behavior, and structure. DSLs provide a more focused set of modeling elements developed to describe the domain's common concepts and relationships. DSLs can generate compact and expressive models with graphical editors, text editors, and code generators, leading to increased productivity and shorter development periods.

Modeling languages can also differ depending on their abstraction level. Multi-viewpoint modeling is significant, which entails creating many models of the same system. These models can communicate in various languages, but coordinated notations that complement one another are more convenient. These coordinated languages, examples of GPLs, are known as Modelling Language Suites (MLSs), popularized in the MDE community by the open-source project Eclipse. UML is the best-known example of an MLS.

2.2.4 Metamodels: Templates for Model Cohesion and Interoperability

Metamodels emerged to normalize model representation and language, tracing its roots to the 1980s and 1990s when efforts were made to define notations and languages for varied modeling purposes, such as consistency and interoperability.

Metamodels act as building templates for models, ensuring that all models in a domain share common characteristics and can interact seamlessly. It specifies the permissible elements, their properties, and potential interactions within a specific domain or modeling language [57]. Therefore, a model is said to conform to its metamodel [7, 10, 57].

Metamodels establish the DSL, regulations, and restrictions that models in a particular domain must follow and are defined by many as a model that describes other models (Figure 2.3). Any relationship or concept not present in the metamodel will be ignored when building the model representing the system [7]. This standardization promotes consistency and interoperability across different models, facilitating collaboration and reducing the risk of errors.

2.2.5 Transformations: Orchestrating Model Evolution

Model transformations are the foundation of MDE [68], alongside models, enabling the automatic conversion of a target model from a source model alongside a set of predefined constraints [54, 42, 85, 53]. Transformations propagate model changes to other related models, ensuring the system remains consistent and up-to-date [85]. Models also include transformations from abstract to concrete models, such as design to code, and vice-versa with reverse engineering, crucial in code generators and parsers [54].

This automation eliminates the need for manual data migration and reduces the risk of errors, streamlining the development process and enabling rapid adaptation to

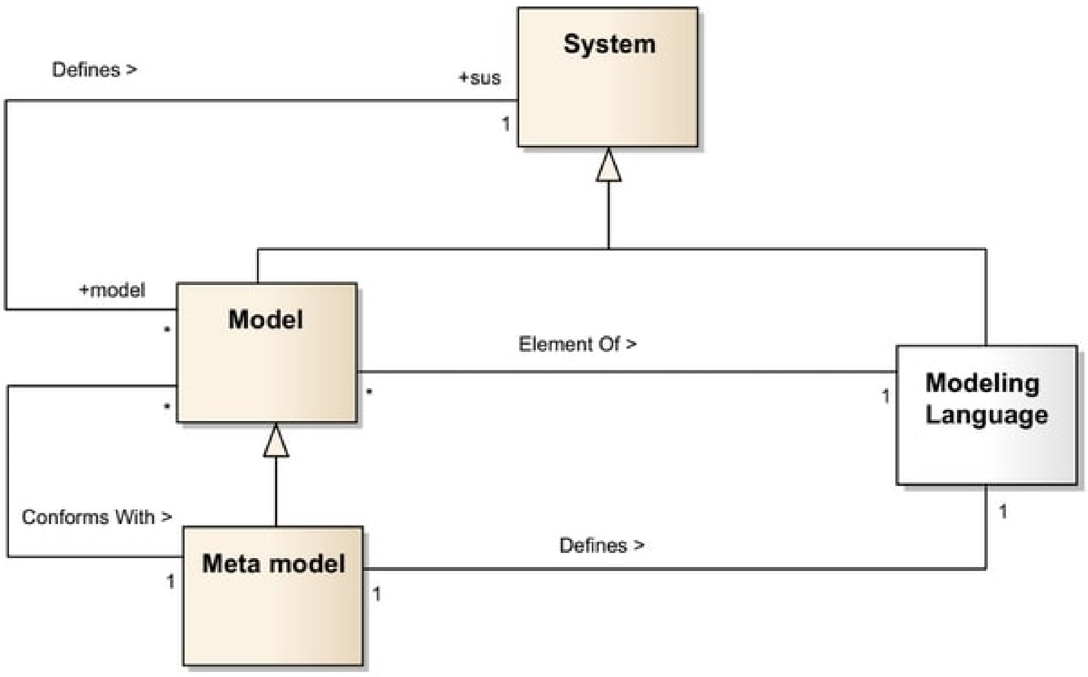


Figure 2.3: The metamodel definition: relationships between metamodel and model (Taken from [65])

evolving requirements. Figure 2.4 depicts the key components of MDE, including software products, platforms, transformations, and models.

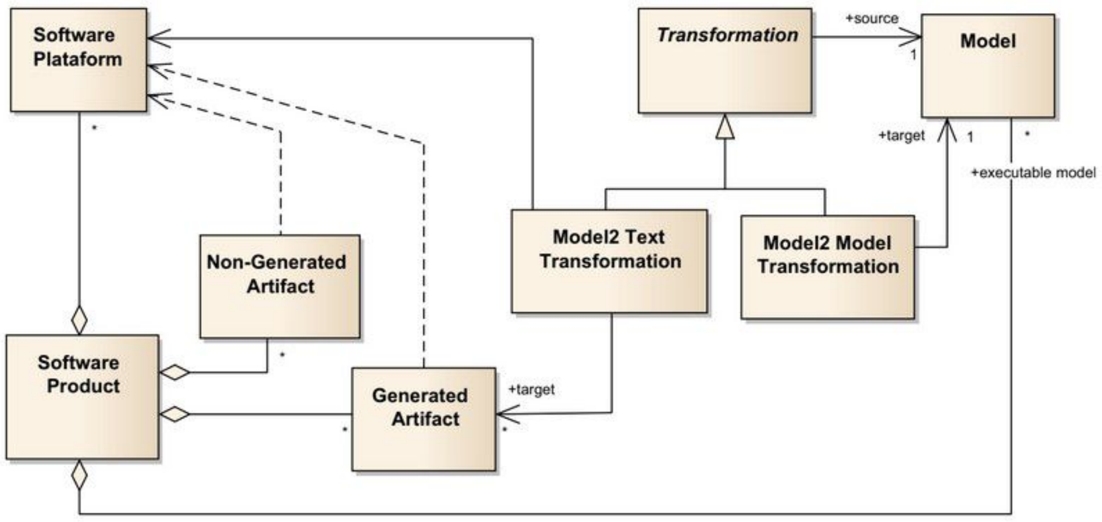


Figure 2.4: Software product, platforms, transformations, and models (Taken from [65])

MDE involves two main types of transformations: Model-to-model (M2M) and Model-to-text (M2T) transformations. M2M transformations convert one model into another by analyzing and extracting data from the source model before creating a new one with a different structure, syntax, or semantics. In this transformation, the models may use the same or different modeling languages. M2M transformations can be classified into endogenous (rephrasing) when the source and destination metamodels are identical and exogenous (translation) when they diverge [54, 10]. Models can be translated between multiple representations and refined and simplified to be closer to the solution. This approach offers seamless interoperability between diverse modeling tools, allowing for system integration and information flow across different stages of software development or platforms.

Figure 2.5 illustrates a transformation process where an input model, conforming to a source metamodel, is transformed into an output model that conforms to a target metamodel. The Transformation Specification defines the rules and operations for the transformation, serving as the connection between source and target metamodels. This specification outlines the rules and operations needed to execute the transformation between models. Both metamodels adhere to a higher-level standard called the Meta-metamodel (often represented by Ecore), which ensures structural consistency across the modeling environment. To execute a transformation, the system uses a Transformation Language, which conforms to the meta-metamodel, allowing the creation of precise transformation specifications. These are then executed by the Transformation Execution process, which reads the input model and applies the transformation rules, producing an output model that adheres to the target metamodel.

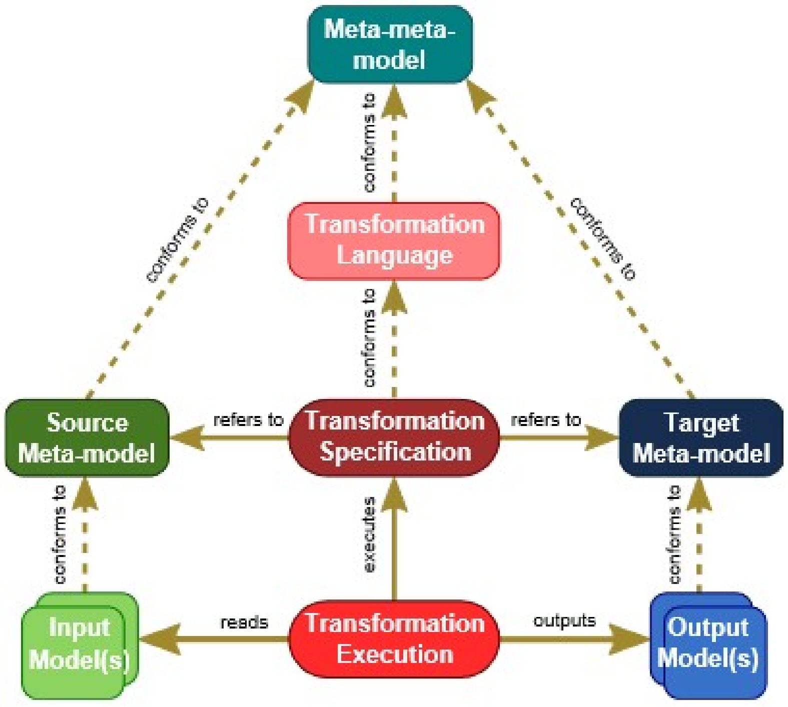


Figure 2.5: Model Transformation

M2T transformations operate on models as their primary input and generate corresponding text as their output, such as code [54]. In practice, these transformations are called code generation when text is in the form of source code [85]. They employ a set of transformation rules that map model elements and their relationships to specific textual constructs. These rules define the syntax and semantics of the generated text, ensuring that it accurately represents the underlying model. In addition to code generation, M2T transformations can generate documentation from models to keep up with an evolving code base.

2.2.6 Modeling Approaches

In software engineering, modeling has become an essential part of the development process. Modeling involves creating a representation of a system or process using a specific language, usually employing a textual or graphical approach.

Graphical representation is a method of visually describing information in two dimensions, as it displays the spatial structures of the parts of a system or process [44]. This approach is often used by developers who prefer a more intuitive and user-friendly way of designing and configuring their systems. It promotes a quick understanding of data and facilitates communication between stakeholders and requirements engineers [51] by effectively capturing relationships and dependencies between components [51, 79, 12].

The growing complexity of contemporary systems has resulted in a considerable amount of model requirements and scalability problems with graphical modeling. Moreover, users need to know the model and its notations before modeling, which might be challenging [51]. Graphical modeling languages like BPMN, UML and Ecore are commonly used in this approach.

On the other hand, the textual approach mitigates some previous modeling challenges mentioned. Textual modeling offers a more compact and structured approach using a textual template to represent model elements [51]. It is well-suited for developers familiar with scripting languages and those seeking greater control over the model's configuration and implementation. Textual modeling languages like YAML and JSON are commonly used in this approach. While the initial learning curve for textual modeling languages can be steeper, this approach can save time and effort, especially when dealing with complex algorithmic model behaviors [1, 65, 51].

2.2.7 Modeling Tools

Modeling tools are crucial in software development as they offer a systematic approach to creating and evolving software systems. These tools allow software engineers to design and build software systems using models as the primary artifacts for the development process. They can be graphical or textual editors. Transformation tools help convert models between different formats, and code generation tools streamline model translation into executable code, making MDE a powerful approach for complex software development.

Graphical Editors

Sirius is an Eclipse project that focuses on developing domain-specific modeling tools, offering a graphical modeling environment for users to define their tools. It supports the creation of graphical editors for Eclipse Modelling Framework (EMF) models, enabling developers to define data models and produce Java code. Users can visualize and validate metamodel structures through diagrams and visualizations, ensuring easy inspection and confirmation of relationships and structures [28, 42]. This approach is advised for creating primarily graphical DSLs.

MPS is an open-source workbench that also allows domain-specific language creation. A model is projected onto a particular graphical representation in the MPS editing approach. It includes a code generator that generates code from metamodels or existing code [43]. This approach is advised when developing hybrid text and graphical DSLs.

MetaEdit+ is a domain-specific modeling tool designed for domain experts to create, edit, and maintain models accurately reflecting their domain. It offers a graphical editor for creating diagrams and a text editor to edit model code. MetaEdit+ is highly customizable, allowing users to create modeling languages and generators.

Textual Editors

Xtext is an Eclipse-based textual editor framework designed for creating and editing DSLs. It allows grammar definition for textual languages and generates parsers, editors, and other tools to work with these languages [42]. Xtext complements Sirius by enabling textual representations of models defined by the metamodel. Combining Xtext's capabilities with Sirius, developers can verify the correctness and completeness of the metamodel grammar through textual representations. Any inconsistencies or errors in the grammar can be identified and rectified within the textual representation, which ensures consistency between the graphical and textual views of the metamodel.

EMFText is another Eclipse-based textual editor framework that allows defining the textual syntax for an Ecore metamodel. It provides a subset of Xtext's features but is more lightweight and easier to learn, capable of generating non-dependable EMFText code. Additionally, EMFText can automatically generate and analyze default syntax to identify potential issues. This makes it a popular choice for creating small to medium-sized DSLs [24, 42].

M2M Tools

In MDE, Extract, Transform, and Load (ETL) from Eclipse retrieves models stored in various formats or abstractions. Once retrieved, these models are transformed to conform to different metamodels or specific transformation rules, assuring compatibility with the target system. The transformation is critical in modifying the models for use in various phases of development. Finally, the transformed models are loaded into a target model or metamodel [10].

Atlas Transformation Language (ATL) is an EMF metamodel transformation language that allows the creation of multiple target models from a set of source models, using rules that specify how source model elements are matched and initialized [27]. It can be applied in imperative style to streamline the codification of intricate transformations and in declarative style to express relationships between source and target model elements [42].

Query/View/Transformation (QVT) is a language family to define model transformations. It consists of three modules: QVT-operational, QVT-relational, and QVT-core, each addressing a distinct element of model transformation. QVT is a robust and versatile language that may be applied to several activities, including code creation [42].

Code Generation

Acceleo Query Language (AQL) is a code-generation tool for MDE that automates software development processes by defining templates and generating code from models. It offers a flexible solution, customizable structure and content, and can be easily integrated into the software development workflow [26, 42].

Xpand is a declarative language for the EMF, enabling the development of domain-specific languages. It is built on Xtend, a Java superset with metaprogramming features, and defines DSL syntax and semantics.

EGL is a metalanguage derived from the QVT family, specializing in model transformations and code creation. Despite its expressive nature, it can be more challenging to understand and apply than other techniques discussed.

MTL is a formal language for specifying transformations between metamodels. It is based on first-order logic and defines the transformations semantics, often used to verify the correctness of transformations.

Xtend is an EMF code generation language that generates templates that specify the format of the generated code. Variables and expressions are then used to tailor the generated code to meet particular needs, simplifying the generation of intricate and repetitious code.

2.3 Challenges in Modeling Solutions for CI/CD

Software development has become increasingly complex with the integration of CI/CD practices into the DevOps landscape, raising gaps and challenges in the area. Some of them are outlined below:

The Lack of Standardization - Varying methods across organizations create a lack of consensus on a uniform modeling framework, hampering team collaboration and interoperability [8, 69]. The absence of standardization impacts CI/CD pipeline tools and integration [8], leading to inconsistencies in modeling techniques and hindering effective model sharing and reuse. The lack of tools capable of handling various stages, interactions, and automation in CI/CD workflows leads to inefficiencies and bottlenecks in the modeling process [69].

Collaboration and Skill Gaps - The identified deficiencies have a major impact on team cooperation and productivity, making them key obstacles to adopting DevOps [45, 69, 22]. Organizations face a significant challenge in finding skilled employees with both development and operational skills and knowledge [45, 69], due to distinct DevOps practices in organizations [45].

Complex Automation - Modeling CI/CD pipelines effectively requires a balance between capturing the intricate details of automated workflows involved in code integration, testing, deployment, and monitoring while remaining simple and adaptable. Additionally, modeling solutions must cater to the varying needs of different stakeholders.

The Limitations of MDE - MDE, while beneficial, also presents several complexities that require addressing. Higher abstraction levels may not necessarily lead to better software, and the abundance of MDE techniques impedes business decision-making since MDE effectiveness is very contextual [37]. Another factor is the dependency on EMF, the common foundation for most MDE offerings, implying that most tool sets have some dependency on it [74]. Furthermore, MDE can lead to lock-in in the abstractions and generator languages adopted, as it requires several dimensions of evolution [23].

Model Evolution Pitfalls - Model evolution in CI/CD pipelines involves updates to meet changes in requirements and software architecture. This MDE process faces challenges like scalability, automation management, heterogeneous dependencies, empirical studies, tool usability, syntax, run-time, hybrid migration approaches, external interfaces, language semantics, and metamodels. Addressing these is crucial for long-term maintainability and effectiveness [57, 14, 37].

Beyond Technical Challenges - In addition to the technical challenges, MDE faces social and community challenges such as empowering non-technical stakeholders to build tools using MDE, companies reluctant to change due to its high adoption risks [38], significant initial investment and a steep learning curve [48], addressing Intellectual Property concerns, and considering domain-specific EAW adapted for different sectors and stakeholders [14].

2.4 Summary

This research highlights the necessity for specific modeling solutions for CI/CD in the DevOps scenario. Achieving this goal will require innovative approaches, collaborative efforts across teams, and the development of specialized tools encompassing the intricacies of CI/CD workflows [5]. The MDE community must consider DevOps and CI/CD as first-class concerns to ensure practical industrial relevance and operate more Agile. Additionally, horizontal reuse of all MDE tooling within programmed workflows is critical for practical MDE deployment, paving the way for efficient and successful CI/CD workflows that help organizations remain competitive in the fast-paced world of software development.

3 Related Work

This chapter investigates current trends and established mechanisms in DevOps modeling, centering on a specific application domain within the DevOps framework. It highlights the limitations encountered by existing modeling frameworks and tools while introducing recent advancements.

3.1 Literature Review

This section provides an overview of the planning and execution of the literature review for this thesis on modeling CI/CD pipelines.

3.1.1 Planning

The planning phase involved defining the research questions, search strategy, selection criteria, and sources. The research questions were formulated to guide the literature review and were as follows:

- What is the current state-of-the-art in modeling CI/CD pipelines?
- What are the challenges and limitations of existing approaches?
- What are the opportunities for improving the modeling of CI/CD pipelines?

3.1.2 Search Strategy

To find relevant articles for this literature review, the search strategy was designed to include keywords related to CI/CD pipeline modeling. With this in mind, the resulting search query was the following: (("CI/CD" OR "DevOps" OR "software delivery" OR "continuous integration" OR "continuous delivery" OR "continuous deployment") AND "pipeline" AND ("model" OR "modeling")). Based on the research questions and sub-questions, the search terms consisted of combining the terms "CI/CD", "pipeline", and "model" or "modeling". Additionally, the search terms were expanded to include related terms such as "DevOps", "software delivery", "continuous integration", "continuous delivery", and "continuous deployment".

The search strategy was designed to ensure that all relevant articles were included in the literature review. A wide range of pertinent subjects were covered by the keywords that discussed modeling approaches and tools for CI/CD pipelines. These search terms were used to find articles in online databases, including ACM, IEEE Xplore, Google Scholar, ResearchGate, and ScienceDirect. The search strings were adapted to fit the syntax and conventions of each database, and the publication date of the articles was not a restriction on the search, but more recent publications were prioritized.

3.1.3 Selection Criteria

The selection criteria were based on the research questions and formulated to ensure that the selected articles were relevant and contributed to the knowledge and development of the proposed solution. The criteria were as follows:

- Relevance to the modeling of DevOps pipelines
- Potential to contribute to the knowledge and development of the proposed solution
- Use of model-driven approaches
- Use of graphical or textual modeling approaches

3.1.4 Execution

After searching, some papers were selected using the previously stated selection criteria to ensure that only relevant publications were included in the final analysis. The screening process was divided into two stages: initially evaluating the titles and abstracts, followed by a comprehensive review of the full text. Articles that did not align with the inclusion requirements were omitted.

The final selection of papers was based on their relevance to the research topics, research quality, and potential contribution to knowledge and the development of the proposed solution. The majority of the selected publications used case studies to validate their offered solutions, which was discovered to be a strong approach for gauging and comprehending the impact and adaptability of the proposed solution across varied organizational contexts.

3.2 Existing Modeling Mechanisms for DevOps

This section explores several approaches proposed in the literature, alongside their contributions and shortcomings to address the challenges of modeling, configuring, and automating DevOps processes.

Colantoni et al. introduced DevOpsML [20], a framework aimed at modeling and configuring DevOps engineering processes and platforms. The framework integrates process and platform models, providing a comprehensive view of the DevOps environment. By leveraging MDE technologies like EMF and Epsilon, DevOpsML comprises process models, platform models, libraries of reusable platform elements, and link models. However, while DevOpsML addresses the need to incorporate new requirements into low-code engineering platforms, it focuses on modeling DevOps as a whole with no code generation and is primarily intended for documentation.

Building upon the DevOpsML framework, ongoing research by Colantoni et al. introduces blended modeling and scenario simulation for continuous delivery pipelines using executable JSON-based models [21]. By employing JSON-SchemaDSL and GEMOC Studio, the authors develop a modeling workbench for JSON-based DSLs, enabling blended modeling of CD pipelines. Their approach combines textual and graphical notations, promoting scenario simulation and evaluation of CD pipelines. However, unlike our approach, their methodology primarily focuses on a single CI/CD technology, limiting its applicability to diverse toolchains.

Sandobalin presents ARGON [67], a model-driven approach for automating the continuous delivery of cloud-based DevOps. ARGON uses a DSL to model cloud infrastructure and its requirements defined through an Infrastructure metamodel. The metamodel provides abstract syntax but lacks concrete notation for ARGON's graphical language. EuGENia is used in the graphical notation in modeling editors. ARGON is coupled with a transformation engine for generating scripts to manage configuration management tools (CMTs).

Brabra et al. propose a model-driven approach for cloud resource orchestration [9], providing systematic mapping and translation support between TOSCA and DevOps tools. To achieve this, they adopt the TOSCA standard for designing resource-related artifacts, regardless of the DevOps tool. They also propose a model-driven translation technique that translates the designed artifacts using TOSCA into DevOps-specific artifacts. Finally, Connectors are provided to establish the bridge between DevOps-specific artifacts and DevOps tools. The authors demonstrate the effectiveness of their approach by applying it to a range of DevOps tools, including Docker, Kubernetes, Terraform, and Juju. Although there are similar approaches to our solution, their approach focuses on DevOps artifact generation.

Furthermore, Tegeler et al. introduce CINCO [76], a model-driven CI/CD framework for modern cloud-based applications, offering a graphical modeling language that enables the rapid development of graphical DSLs and their corresponding IDEs. The solution simplifies CI/CD pipeline setup and supports tools through a graphical representation of pipeline models. It allows automatic code generation, job dependency visualization, and scheduling and execution order, eliminating typical errors in setup and scheduling. While their method improves readability and reduces errors in the pipeline setup, it primarily targets modern cloud-based applications and is currently applied to GitLab.

Wettinger et al. present a TOSCA-based framework [80] for integrating diverse DevOps artifacts within a modular and cloud-agnostic architecture. It discusses the transformation of DevOps artifacts such as Chef cookbooks and Juju charms into standardized TOSCA artifacts, making them interoperable across different platforms. Additionally, it outlines the APIfication phase, where APIs are generated to interact with these artifacts, enabling deployment on different platforms without specifying each one individually. Leveraging Winery, an open-source TOSCA modeling tool, provides a graphical editor for crafting plain topology models. Moreover, the framework incorporates textual modeling for artifact APIfication, generating API specifications based on the functionalities of the artifacts.

In [32], Hugo et al. propose a block-based approach for defining CI/CD pipelines, aiming to overcome the challenges of non-interoperable languages in existing technologies such as GitHub Actions, GitLab CI/CD, and Jenkins. By offering a visual and interactive environment with error detection and helpful suggestions, this approach streamlines pipeline creation and promotes switching between providers. While not implemented, this concept may serve as a basis for our graphical approach.

Finally, Rig [77] is a graphical modeling tool for CI/CD workflows designed to help developers maintain complex workflows. It offers a Purpose-Specific Language (PSL) that abstracts current CI/CD implementation complexities and generates YAML configuration files automatically, ensuring syntactic correctness and compatibility with platforms such as GitLab. Rig uses the Eclipse Rich Client Platform and the Cinco framework to provide an Integrated modeling Environment (IME) for the graphical modeling of CI/CD workflows. It provides an easy-to-use interface for creating, editing, and visualizing CI/CD pipelines with its node and edges representation of jobs and dependencies. The model-driven technique generates entire code from graphical models, assuring consistency and correctness in the resulting YAML configuration files and optimizing CI/CD processes while reducing manual effort and errors.

Table 3.1 highlights and compares the core features of each of the previously mentioned proposals. While several of the solutions support both graphical and textual modeling, none facilitates migration between DevOps platforms.

Characteristic/Approach	DevOpsML	JSON-SchemaDSL	ARGON	Brabra	Tegeler	TOSCA-based	Block-based	Rig	Our Solution
Graphical Modeling	X	X	X		X	X	X	X	X
Textual Modeling		X	X	X		X			X
Code Generation		X	X	X	X	X	X	X	X
Platform-Agnostic				X		X	X	X	X
Platform-Migration									X

Table 3.1: Comparison of Related Approaches

3.3 Ongoing Trends

The environment of DevOps modeling evolves alongside the DevOps philosophy, which is required since modern DevOps environments require efficient, agile, and robust software development and delivery processes. Organizations are exploring emerging patterns, modeling solutions to these problems, and implementing them. We conducted research to identify trends and forthcoming initiatives being investigated and developed.

3.3.1 AI/ML-driven DevOps

AI/ML integration in DevOps has become increasingly popular, with ML-driven DevOps technologies being employed in Machine Learning (ML) projects [66]. According to Dhia et al., using DevOps technologies improves the success rate of ML initiatives. This highlights the potential benefits of using AI/ML to boost DevOps efficiency for future dynamic and distributed data-intensive systems [2, 14].

AI/ML is also expected to improve modeling automation and effectiveness, with additional applications such as modeling bots that give advice and even create source code [16, 13, 74] and model recommenders being incorporated into IDEs. The main reason is that many MDE techniques heavily rely on knowledge and data, as well as the repetitive source code generation for MDE template processes [13, 74].

However, integrating AI/ML into DevOps can present challenges, including the availability of real-world datasets, continuous model evaluation and tuning, and the difficulty of maintaining modularity in large-scale DevOps projects [2].

In light of these challenges, Moin et al. propose adopting Automated ML to enable Model-Driven AI Engineering. This empowers software engineers without deep AI knowledge to develop AI-intensive systems by selecting the most appropriate ML model, algorithm, and hyper-parameters for the task at hand [55].

Raedler et al. [62] used MDE techniques to develop AI algorithms, while Planas et al. presented a Model-Driven Development (MDD) approach to multi-experience user interfaces, integrating a new DSL with the Interaction Flow Modeling Language (IFML) to provide seamless user experiences across several devices and modalities [60].

3.3.2 Low-Code

Merging DevOps and Low-code approaches to improve software development processes is an emerging trend [15, 63, 13]. Low code is a subset of MDD and not a novel concept [15]. Rafi et al.'s work highlights the benefits of adopting low-code platforms in DevOps processes [63]: One of the primary advantages of low-code platforms is bridging the skill gap by allowing non-technical developers to create software and increase participation in software development processes.

Low-code platforms offer a visual and user-friendly approach to application development, expediting and reducing the development time to market. Low-code platforms can enhance DevOps teams' development processes by integrating with other technologies, reducing errors, and enhancing software quality.

3.3.3 Cloud-Based DevOps

Cloud solutions have a strong presence in today's community, and new ways are emerging in favor of business efficiencies, cost savings, agility, and so on. One trend mentioned was modeling cloud-based environments to reduce complexity and extend the accessible capabilities. According to Bucchiarone [13], versioning tools are expected to become an essential feature of modeling environments in the future.

In [74], Süß et al. discuss the use of Kubernetes in cloud-based MDE workflows with interactive IDEs hosted on the cloud provider. The authors believe that this approach can assist in mitigating some of the issues associated with model-driven technology, such as slow interpretation times and limited access to resources.

3.3.4 Honorable Mentions

Other trends have been mentioned that are worth considering [13, 16]:

- **Multiparadigm modeling** - Integrates different modeling paradigms into a single framework, supporting cross-disciplinary communication and collaboration;
- **Adoption models** - Help organizations assess, evaluate, and improve their modeling practices;
- **Model inferers** - Extract patterns from unstructured data and generate corresponding models;
- **Smart code generators** - Generate code from models, taking into account the style and best practices of a particular organization;
- **Real-time model reviewers** - Provide continuous feedback on the quality of models, identifying potential problems and suggesting solutions;
- **Advanced self-morphing and collaborative modeling tools** - Adapt to the needs of different users and collaborate with them in real-time;
- **Semantic reasoning platforms, explainability, and storification** - These tools can make models more self-explanatory and easier to understand, and can also store and manage models in a centralized repository.

3.4 Summary

The reviewed DevOps modeling approaches offer valuable contributions to the field, providing structured representations of complex DevOps processes, enabling optimization, and enhancing collaboration. By leveraging these insights, organizations can effectively manage their DevOps pipelines and deliver high-quality software with greater agility and efficiency. The proposed solution builds upon the strengths of existing approaches and addresses identified research gaps, aiming to advance DevOps modeling practices further.

4 Model-driven Languages

This chapter presents the proposed solution aimed at addressing the challenges in configuring and migrating DevOps pipelines. The solution's technology, procedures, and implementation process will be presented and justified.

4.1 Requirements Analysis

A thorough examination of requirements is required to develop a novel approach to addressing the issues found in the literature review. This section examines the functional and non-functional requirements that guide the solution's design and implementation.

4.1.1 Functional Requirements (FR)

- **Textual Modeling (FR-1) -** The solution shall offer a textual modeling interface using Xtext that enables the configuration of scripts using syntax and grammar tailored to the supported CI/CD platforms.
- **Visual Modeling (FR-2) -** The solution shall provide a visual modeling interface using Sirius, allowing users to configure and visualize models through graphical editors, ensuring alignment with the supported CI/CD platforms.
- **Tool Agnosticism (FR-3) -** The modeling approach should be agnostic of any specific DevOps platform. This means the generated code and the modeling approach should be adaptable to different CI/CD tools and platforms. This prevents vendor lock-in and allows for future migration between these platforms. Developers can apply the solution to projects that use a variety of CI/CD tools without requiring major adjustments.
- **Automated Code Generation (FR-4) -** The solution must be capable of automatically generating executable code (such as configuration files and scripts) based on the CI/CD pipeline's provided textual and visual models. This generated code should be interoperable with common CI/CD tools and platforms, avoiding the need for manual configuration and the risk of errors associated with human coding.

- **Model Transformations (FR-5) -** The solution should facilitate transformations between different types of models used for representing CI/CD pipeline requirements, as this feature still has a high prevalence. This allows users to switch between different model views for better understanding or communication.

- **Error Handling and Reporting (FR-6) -** The solution should include functionalities for handling errors that occur during model instance configurations. This will allow the user to understand and correct mistakes at any configuration step, reducing their chances of frustration from being unable to correct their issues. This includes capturing error messages, logging them for troubleshooting, and potentially notifying users of failures.

4.1.2 Non-Functional Requirements (NFR)

- **User-Friendly Textual Modeling (NFR-1) -** The modeling tool shall provide a user-friendly interface for defining CI/CD pipelines using a clear and concise textual syntax. This syntax should be readily understandable by developers with varying levels of CI/CD expertise. It should allow for the specification of various stages within the pipeline.

- **User-Friendly Visual Modeling (NFR-2) -** The solution shall offer a complementary visual representation of the CI/CD pipeline structure and its modeling. This visual representation should directly correspond to the textual syntax, enabling users to easily grasp the pipeline's flow and dependencies. This visual component fosters better communication and collaboration within development teams regarding CI/CD pipelines. It also enables less technically inclined or visually oriented users to configure their instances more effectively.

  Both NFR-1 and NFR-2 requirements directly address the skill gap identified in the research, where 56% of published articles highlight a lack of knowledge and understanding of CI/CD concepts [45] and its difficult learning curve. By prioritizing a user-friendly textual syntax with a low learning curve, the tool empowers users with varying levels of experience to effectively model CI/CD pipelines.

- **Intuitive User Interface (NFR-3) -** Both textual and visual solution interfaces should be intuitive to learn and use. Catering to users with varying levels of experience is crucial.

- **Scalability for Complex Pipelines (NFR-4) -** The solution should be able to handle complex CI/CD pipelines with numerous stages and intricate dependencies. This scalability ensures the solution remains effective as projects mature and pipeline complexity increases.

- **Maintaining Clean Code (NFR-5) -** The solution should generate clean and well-documented code that is easy to understand and modify. This promotes the maintainability of the CI/CD pipelines over time. Clean code allows developers to easily troubleshoot issues and adapt the pipelines in the future.

4.2 Solution Architecture

The proposed solution employs a dual modeling approach to accommodate diverse user preferences and skill levels. Textual and graphical modeling approaches enable users to configure CI/CD scripts for several DevOps platforms using a generic metamodel. The graphical modeling approach uses Sirius to model the pipeline, while the textual modeling approach uses Xtext to create DSLs for pipeline modeling. Both approaches rely on ATL to convert between platform-independent and platform-specific models, and Acceleo to generate scripts for deployment in the target DevOps platforms.

The 4+1 view model of software architecture best describes our solution's architecture, providing a comprehensive viewpoint on different aspects of the system. This thesis focuses on two specific views: the Process View and the Development View.

The Process View offers a dynamic view of the system's architecture, focused on runtime behavior and interactions among components. These interactions are outlined in Figure 4.1 as an Activity Diagram for both configurations, the textual approach is depicted on the left and the visual on the right.

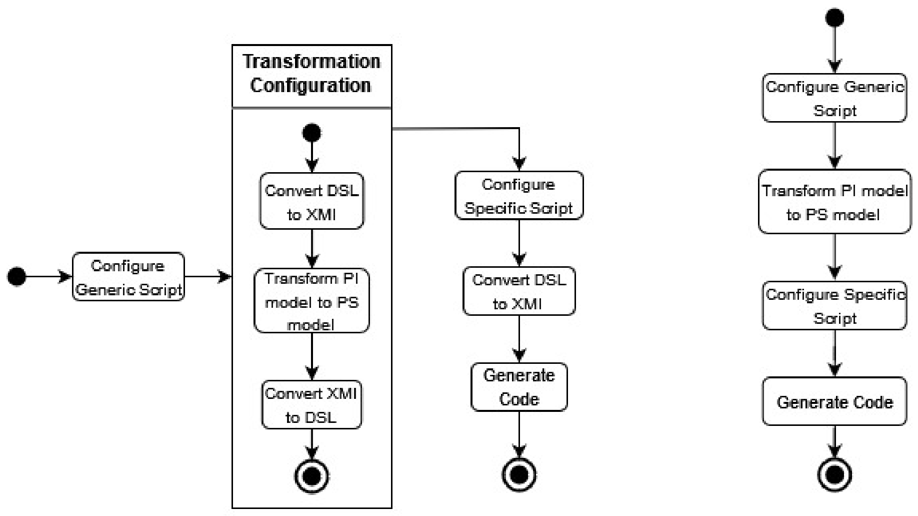

Configure Generic Script
Transformation Configuration
Convert DSL to XMI
Transform PI model to PS model
Convert XMI to DSL
Configure Specific Script
Convert DSL to XMI
Generate Code
Configure Generic Script
Transform PI model to PS model
Configure Specific Script
Generate Code

Figure 4.1: Process View

The textual approach (shown on the left) begins by configuring a generic script in the developed CICD DSL. This is followed by the 'Transformation configuration' sub-process, which includes several steps. First, the DSL is converted into XMI format. Then, the Platform-independent (PI) model is translated into a Platform-specific (PS) model using ATL. Finally, the XMI is converted back into DSL. After these transformations, the new

XMI is configured for the selected target platform during the ATL phase. Then, the DSL is converted back to XMI format for use in the code generation process. The process concludes with the generation of code in a YAML file.

Similar to the textual approach, the visual approach begins with configuring a generic script on the developed Sirius CICD diagram. However, the difference lies in the direct transformation of the PI model into the PS model using ATL. The new XMI is configured for the selected target platform in the ATL phase and then used to generate the code in a YAML file.

Figure 4.2 and Figure 4.3 presents the Development View, also known as the Implementation View, which provides a static perspective of the system's architecture from a software development standpoint.

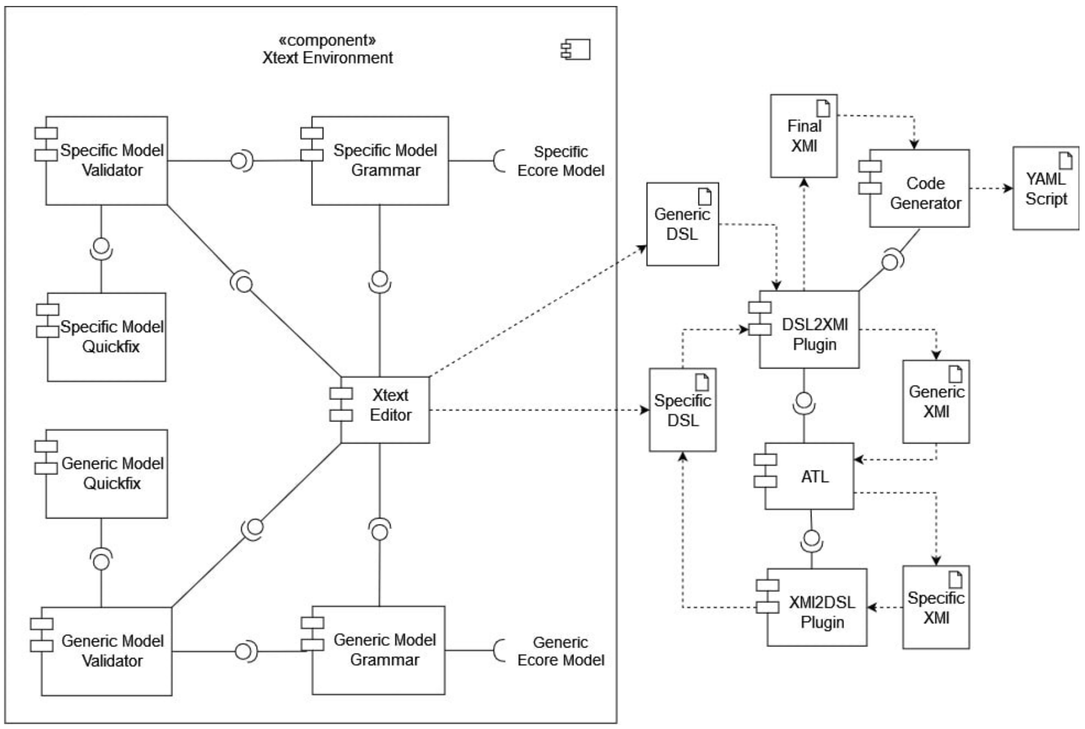


Figure 4.2: Textual Development View

The Xtext Environment development view focuses on the textual representation of models using grammar and quick fixes. The Specific Model Validator checks if the specific configuration follows the constraints and rules outlined in the Specific Model Grammar and the Specific Ecore Model. On the other hand, the Generic Model Validator validates the generic configuration using the constraints and rules described in the Generic Model Grammar and the Generic Ecore Model. The Specific Model Grammar specifies the syntax and structure of a specific configuration, while the Generic Model Grammar provides a generic syntax framework for generating generic configurations. Specific Model Quick fixes automate the correction of common validation errors in specific configurations, while Generic Model Quick fixes offer similar solutions for generic ones. The Xtext Editor is the main interface for writing and editing DSLs, and it supports both generic and specific model grammar, as well as quick fixes.

The DSL2XMI Plugin converts DSL files to XMI format, whereas the XMI2DSL Plugin turns XMI files back to DSL format for use in the Code Generator module. ATL enables the conversion of generic and specific XMI formats. The Final XMI is the outcome of these transformations, and it is then used by the Code Generator to produce the corresponding code, in this case, a YAML script.

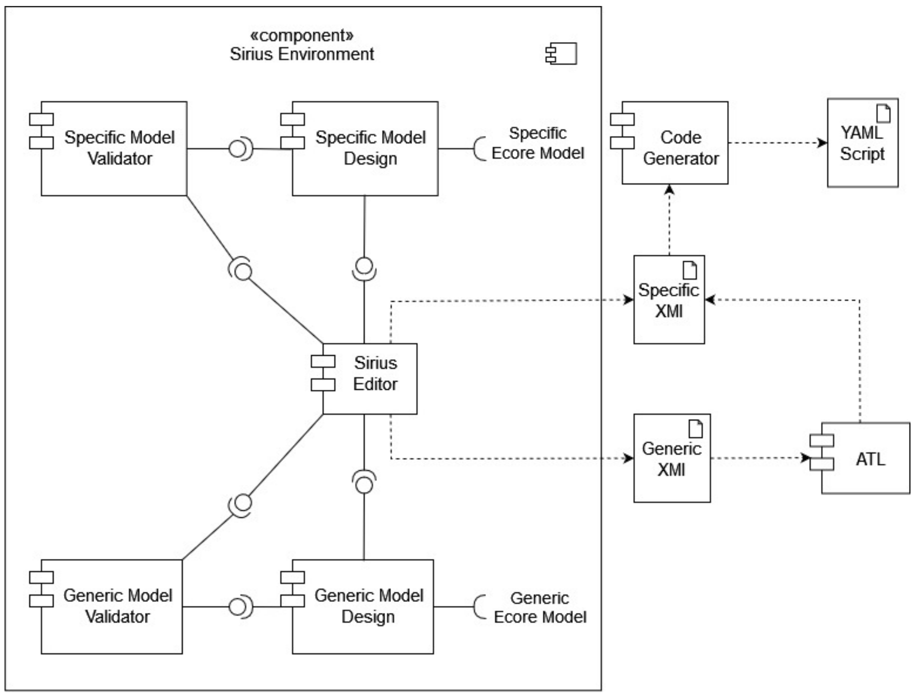


Figure 4.3: Visual Development View

The development view of the Sirius environment includes various components and their interactions to support model validation, editing and code generation. In this approach, the Specific Model Validator validates the specific Ecore model for correctness and consistency, while the Generic Model Validator validates the generic Ecore model. The Sirius Editor acts as the central editor for creating and modifying model configurations available within the features defined in Generic and Specific Model Design, interacting with the specific and generic model validators to ensure model integrity.

Based on the configuration nature, ATL is used to transform the Generic XMI configuration into a specific XMI format, while the Specific XMI file is the final configuration output. The Code Generator (Acceleo) uses the Specific XMI to generate code, in this case, a YAML Script.

Figure 4.4 shows the process of the artifacts generated for both approaches, with Textual configuring the artifacts in Xtext based on the Domain Model and grammar defined and Graphical configuring them using the editor implemented. Both approaches generate the final artifacts with the Acceleo templates.

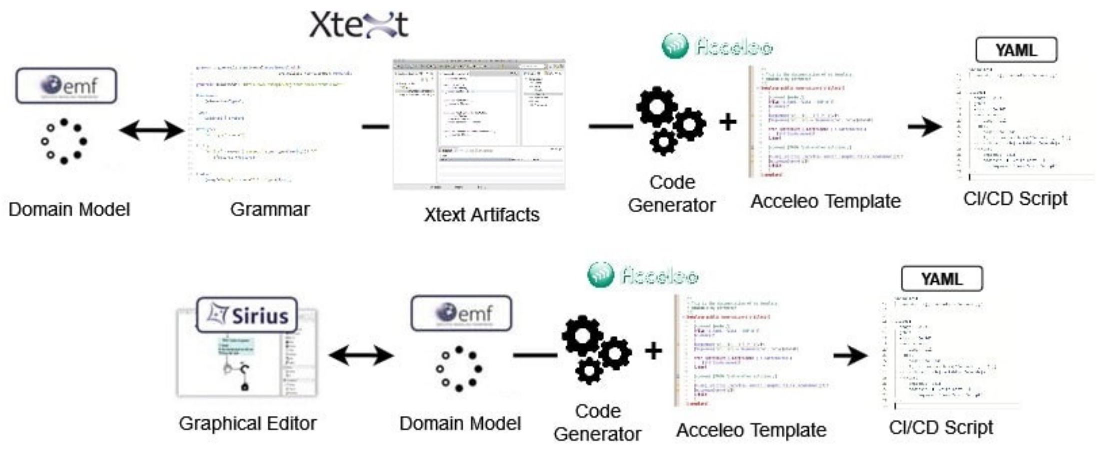


Figure 4.4: Generated Artifacts Process

4.3 Core Metamodel Design

The core of our CI/CD modeling tool is built around a generic metamodel (Figure 4.5), which serves as the foundational framework for both the textual and visual modeling approaches. This generic metamodel is designed to be platform-independent, allowing it to serve as the basis for developing platform-specific metamodels tailored to various DevOps environments. By maintaining a high level of abstraction, the generic metamodel ensures that the tool remains flexible and adaptable to different CI/CD platforms, addressing the diverse needs of development teams.

4.3.1 Generic Metamodel

We departed from an existing metamodel made available by the authors that work at FEUP that had been modified to interact with various DevOps platforms. This metamodel has been improved to accommodate common features present in popular community platforms while remaining at a high level of abstraction. According to [31], GitHub Actions is the dominant DevOps platform by a significant margin. As a result, more emphasis was placed on the functionalities of GitHubActions while developing the final generic metamodel. We included GitHubActions features that were also present on at least two additional platforms specified in [31]. For the remaining platforms, features had to be present in at least four. Currently, the tool implementation is available for three popular ones. According to [59, 39, 30, 31] these platforms are GitHub Actions, Jenkins and CircleCI.

4.3.2 Platform-Specific Metamodels

Building on the foundation of the generic metamodel, platform-specific metamodels are crafted to address the unique requirements and features of various CI/CD platforms.

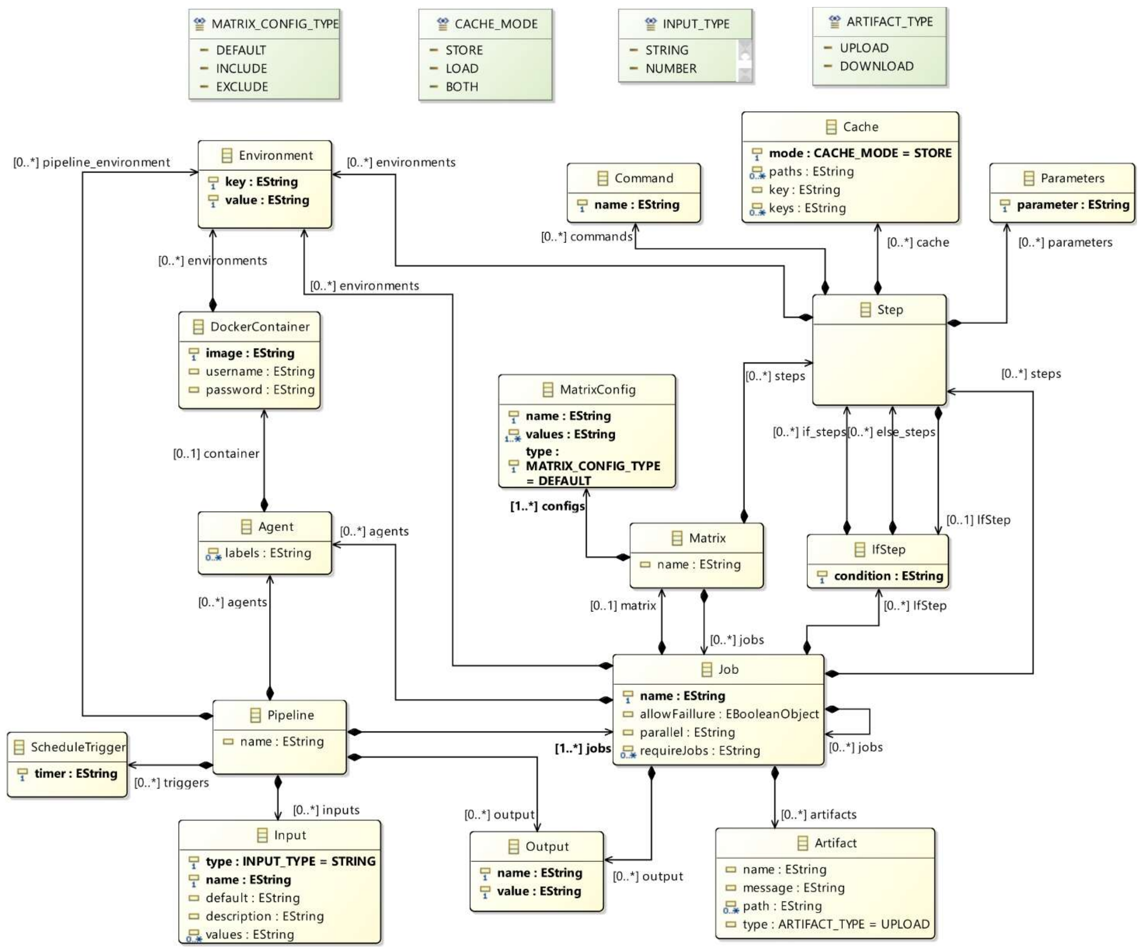


Figure 4.5: CI/CD Independent Metamodel

These metamodels retain the core structure of the generic metamodel but extend and customize it to align with the syntax, features, and workflows of specific CI/CD tools. This approach ensures that the modeling tool can generate accurate and functional pipeline configurations for different platforms while maintaining a high level of abstraction and reusability. As the platform-specific metamodels are very large and complex, they are available in Annex I.

GitHub Actions Metamodel

The GitHub Actions metamodel is designed for GitHub Actions, an integrated CI/CD service within GitHub uses workflow files defined in YAML. Key components of the GitHub Actions metamodel include:

- **Triggers -** GitHub Actions workflows can be triggered by a wide array of events beyond just code push and pull requests. These triggers include webhooks for issue comments, repository dispatches, and schedule-based triggers (cron jobs).

- **Runners -** GitHub Actions supports different types of runners, such as GitHub-hosted runners and self-hosted runners define the environment in which jobs run.
- **Actions -** Actions are reusable units of code that can be combined to build complex workflows. These can be pre-built actions available in the GitHub Marketplace or custom actions defined in repositories.
- **Permissions -** GitHub Actions offers detailed permission settings for workflows, enabling users to restrict access to specific areas, read-only or read-write jobs, and authenticate with GitHub-provided tokens for further control.
- **Secrets -** GitHub Actions support encrypted secrets that can be used for sensitive data such as API keys and credentials.

Jenkins Metamodel

The Jenkins metamodel is designed to cater to the extensive and flexible nature of Jenkins. Jenkins pipelines are defined using a Groovy-based syntax within Jenkinsfiles, which can be either declarative or scripted. Key components of the Jenkins metamodel include:

- **Pipeline Configuration -** This element represents the overall structure of the Jenkins pipeline, including both declarative and scripted pipeline syntax. Our tool does not support scripted pipeline syntax.
- **Agents -** Jenkins agents specify where the pipeline or specific stages should run, whether on any available node or specifically labeled nodes.
- **Triggers -** Jenkins supports a variety of triggers, such as SCM changes, schedule-based triggers (cron), and manual triggers via the Jenkins UI.
- **Post Actions -** Post-build actions specify tasks to be executed after the pipeline stages, such as sending notifications or archiving artifacts.

CircleCI Metamodel

The CircleCI metamodel is tailored for CircleCI, a popular CI/CD tool known for its ease of use and powerful features. CircleCI pipelines are defined using a YAML-based configuration file. Key components of the CircleCI metamodel include:

- **Pipeline Configuration -** Represents the overall structure of the CircleCI configuration file, including version and setup directives.
- **Workflows -** Workflows orchestrate the execution of multiple jobs, allowing for parallel job execution and defining dependencies between jobs.
- **Executors -** CircleCI uses executors to define the environment in which jobs run, including machine executors, Docker containers, and remote Docker.

- **Caching -** CircleCI supports caching mechanisms to speed up builds by reusing dependencies and build artifacts.
- **Artifacts -** Jobs can produce artifacts that are stored and accessible for later stages or for download.
- **Triggers -** Triggers in CircleCI include branch and tag filters, scheduled workflows, and manual approvals.

4.3.3 OCL Invariants for Metamodel Validation

The core metamodel design uses Object Constraint Language (OCL) invariants to ensure data integrity and consistency in CI/CD pipeline definitions. OCL is a declarative language that specifies constraints that must hold for elements and relationships within the metamodel. These invariants are defined within the metamodel and prevent invalid or semantically incorrect pipeline configurations. These restrictions are relevant due to the limited set of rules in a metamodel domain, such as the cardinality of attributes and references. By using OCL, the metamodel domain is enriched as it extends the set of restrictions.

Invariants specific to each platform were developed to maintain consistency across various DevOps platforms. Listing 4.1 contains a snippet of an invariant for the Generic metamodel defined in the OCLinEcore Editor, with the full content in GitHub. Platform-specific invariants can be found in Annex II. This invariant ensures that if the type attribute of an Input instance is of the Boolean type, then the default attribute must be set to either "true" or "false".

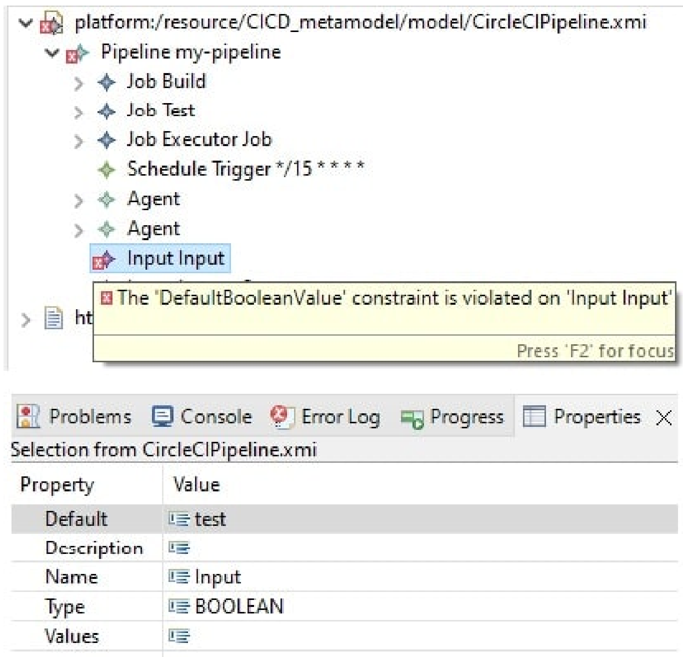
Listing 4.1: CICD OCL Invariants

```
package cICD_metamodel : cICD_metamodel = 'http://.../cICD_metamodel' {
    class Input
    {
        attribute type : INPUT_TYPE[1];
        attribute default : String[?];
        ...
        invariantDefaultBooleanValue('Boolean type must have default value
            set to "true" or "false"'):
            self.type = INPUT_TYPE::BOOLEAN implies
                (self.default = 'true' or self.default = 'false');
    }}
```

Integration with Tree View

Although OCL invariants are defined in the metamodel, the Tree View feature in the visual modeling environment offers a useful tool for interacting with them. The Tree View allows users to navigate the metamodel's elements and relationships, and whenever a change is made, the EMF validation framework checks the relevant OCL invariants. If any invariant is violated, the framework generates validation messages that pinpoint the specific issues and provide guidance on how to fix them.

An example of this mechanism can be found in Figure 4.6 with a CICD metamodel invariant.



```
platform:/resource/CICD_metamodel/model/CircleCIPipeline.xmi
  Pipeline my-pipeline
    Job Build
    Job Test
    Job Executor Job
    Schedule Trigger */15 * * * *
    Agent
    Agent
    Input Input
```

The 'DefaultBooleanValue' constraint is violated on 'Input Input'

Press 'F2' for focus

Property	Value
Default	test
Description	
Name	Input
Type	BOOLEAN
Values	

Figure 4.6: CICD OCL Invariant

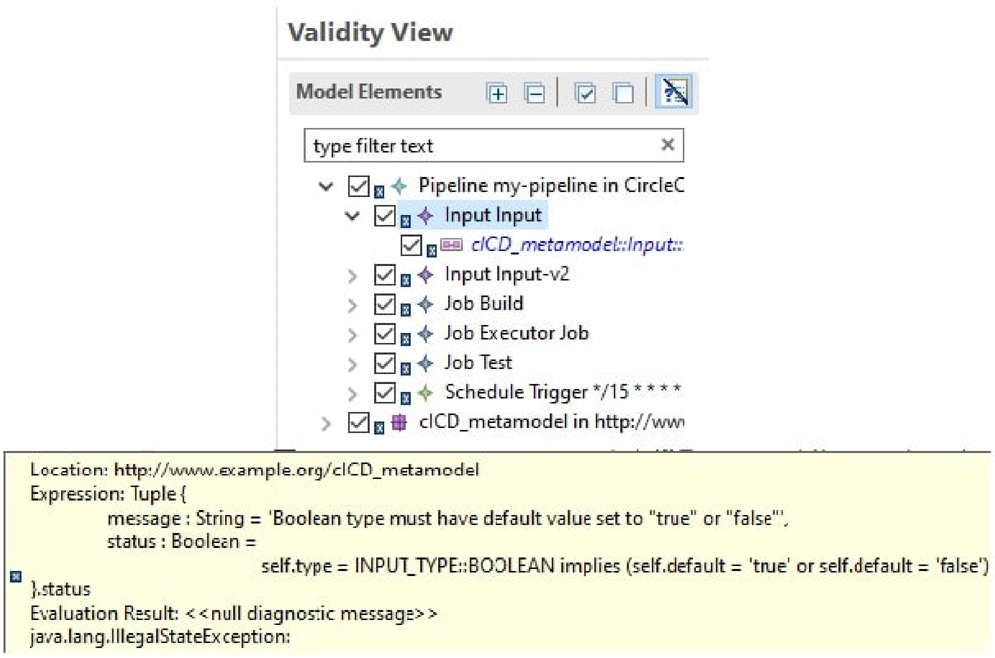
As we can see, a Boolean-type Input has been defined with the 'default' attribute set to 'test'. However, the OCL will produce an error when evaluating this configuration due to the restriction shown in Listing 4.1. In Figure 4.7, we can see the configuration error and the message guiding its correction, in this case by changing the 'default' attribute to 'true' or 'false'.

4.4 Textual Modeling

The textual modeling approach is a critical component of the solution, providing a powerful way to define and delimit the rules and restrictions of CI/CD pipeline configurations through a DSL developed using Xtext. This section elaborates on the grammar definition process, the DSL's syntax and semantics, and the tools and plugins that support this method.

4.4.1 Grammar Definition

The process of defining the grammar for the textual modeling language using Xtext involves several steps. Initially, an Xtext project was created for each existing Ecore model mentioned in Subsection 4.3.2. Xtext automatically generated the initial grammar based



```
Pipeline my-pipeline in CircleC
  Input Input
    cICD_metamodel::Input::
  Input Input-v2
  Job Build
  Job Executor Job
  Job Test
  Schedule Trigger */15 * * * *
cICD_metamodel in http://ww
```

```
Location: http://www.example.org/cICD_metamodel
Expression: Tuple {
        message : String = 'Boolean type must have default value set to "true" or "false"',
        status : Boolean =
                        self.type = INPUT_TYPE::BOOLEAN implies (self.default = 'true' or self.default = 'false')
}.status
Evaluation Result: <<null diagnostic message>>
java.lang.IllegalStateException:
```

Figure 4.7: CICD OCL Invariant Error Message

on this model, providing a starting point that closely reflected the underlying metamodel. However, to better meet the specific requirements of usability, extensibility, maintainability, and user friendliness, this grammar was further refined and customized:

- **Usability -** Designed to be intuitive and easy to understand, this involves using natural language style, eliminating special characters (common in YAML syntax of CI/CD platforms), and organized structure. This aligns with typical DevOps practices, making it accessible even to users who are not experts in modeling or programming, and providing a streamlined configuration.

- **Extensibility -** Allows for easy extension and customization by modifying the metamodel and consequently its grammar, accommodating different DevOps platforms and their evolving requirements. This flexibility ensures that the DSL can adapt to various organizational needs over time.

- **Maintainability -** Can be easily updated and enhanced as new features are introduced or existing ones are modified. This is crucial for the longevity and sustainability of the modeling solution.

- **User Friendliness -** It leverages existing concepts and terms for each platform that users are already familiar with, such as YAML-based configurations, to reduce the learning curve and increase adoption rates.

In Listing 4.2, we have a short grammar extract defined for the generic metamodel (CICD). While we removed some special characters introduced by Xtext for formatting and parsing purposes, such as curly brackets, we retained indentation for readability. Curly braces previously handled indentation, but to maintain structure after their removal, we introduced dedicated BEGIN and END terminal rules within the grammar.

Listing 4.2: CICD grammar definition

```
grammar org.xtext.example.cicd.CICD with org.eclipse.xtext.
common.Terminals

import "http://www.example.org/cICD_metamodel"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

Pipeline returns Pipeline:
        'Pipeline'
        ((BEGIN'name' name=EString END))?
        ((jobs+=Job)+ NEWLINE?)
        ((pipeline_environment+=Environment)+ NEWLINE?)?
        ((triggers+=ScheduleTrigger)+ NEWLINE?)?
        ((agents+=Agent)+ NEWLINE?)?
        ((inputs+=Input)+ NEWLINE?)?
        ((output+=Output)+ NEWLINE?)?
;

Environment returns Environment:
    'Environment'
    (BEGIN
        'key' key=EString
        'value' value=EString
    END)
;

terminal NEWLINE:
// New line on DOS or Unix
    '\r'? '\n';

terminal BEGIN: 'synthetic:BEGIN';  // increase indentation
terminal END: 'synthetic:END';      // decrease indentation
```

An important consideration is the synchronization between the metamodel and the grammar. This means that when changes are made to the metamodel, the grammar alerts the developer to any encountered incompatibilities. This synchronization is achieved through the strong integration within the Eclipse environment, as indicated in the grammar by the *import "http://www.example.org/cICD_metamodel"* line. Despite the synchronization of these structures, manual grammatical adjustments are still necessary. The rationale and implementation provided so far have been applied to the remaining model's grammar.

Some additional excerpts from the platform-specific grammars can be found in Annex III. For the complete grammar, please visit GitHub.

4.4.2 Syntax

The syntax of the textual modeling language is designed to be intuitive and expressive, allowing users to define CI/CD pipeline configurations clearly and concisely. As previously stated, it follows a natural language style with YAML-based conventions to improve readability and usability. The syntax's natural language elements map directly to matching entities and attributes in the underlying metamodel. Listing 4.3 provides an example of a program written for the grammar described in Listing 4.2, with mappings to the CircleCI platform.

Listing 4.3: CICD DSL

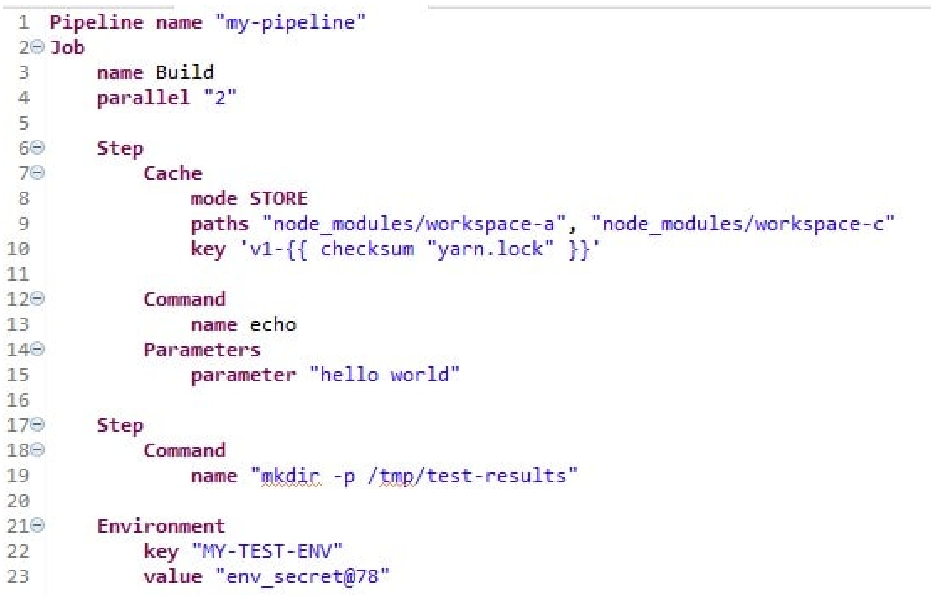
```
Pipeline

Job
    name "build"
    Step
        Cache
            mode LOAD
            keys 'v1-{{ checksum "yarn.lock" }}'
        Cache
            mode STORE
            paths "node_modules"
            key 'v1-{{ checksum "yarn.lock" }}'
        Command
            name "yarn install --frozen-lockfile"

Agent
    labels "default"
    DockerContainer
        image "circleci/node:10"
```

4.4.3 Tool Support

The success of the textual modeling approach depends heavily on the availability of robust tools and plugins that improve the user experience, simplify the development process, and guarantee the correctness of configurations. Leveraging the robust capabilities of Xtext, the following suite of tools was used:

Syntax Highlighting - This feature visually distinguishes different elements of the textual language by rendering keywords, identifiers, literals, and comments in distinct colors

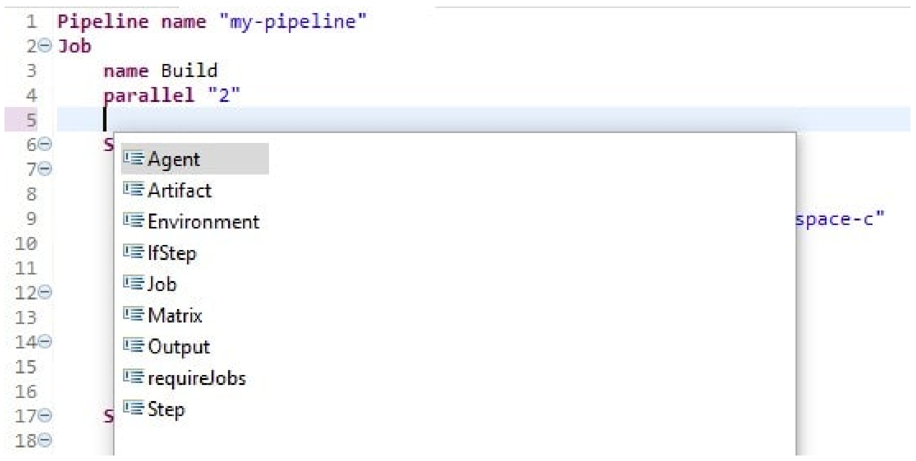
(Figure 4.8). It improves readability, comprehension, and aids in code navigation and error identification.



```
Pipeline name "my-pipeline"
Job
    name Build
    parallel "2"

    Step
        Cache
            mode STORE
            paths "node_modules/workspace-a", "node_modules/workspace-c"
            key 'v1-{{ checksum "yarn.lock" }}'

        Command
            name echo
        Parameters
            parameter "hello world"

    Step
        Command
            name "mkdir -p /tmp/test-results"

    Environment
        key "MY-TEST-ENV"
        value "env_secret@78"
```

Figure 4.8: Syntax Highlighting

Code Completion - Also known as content assist, this feature accelerates the modeling process by providing intelligent suggestions and auto-completion options as users type. This feature leverages the underlying grammar and metamodel to offer contextually relevant proposals, including keywords, attribute names, and valid syntax patterns. It reduces manual input, minimizes errors, and promotes productivity and accuracy, especially for users less familiar with the language syntax. Figure 4.9 displays relevant proposals that are accepted after the 'parallel' keyword, which is available in Xtext with the CTRL+SPACE shortcut, and Listing 4.4 demonstrates Content Assist by clicking on the proposed suggestion.



```
Pipeline name "my-pipeline"
Job
    name Build
    parallel "2"

    S
                                                                   space-c"
    S
```

Figure 4.9: Code completion suggestions

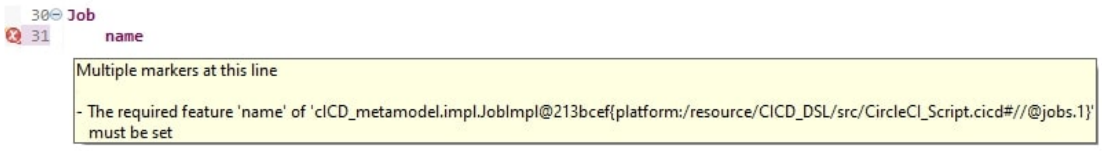
Listing 4.4: Content Assist

```
Pipeline name "my-pipeline"                 Pipeline name "my-pipeline"
Job                                         Job
    name Build                                  name Build
    parallel "2"                                parallel "2"
    Ag                             ->           Agent
```

Error Checking - Error checking with custom validators ensures the correctness and integrity of the modeling artifacts by performing real-time validation against predefined rules and constraints, such as potential syntax errors, semantic inconsistencies, and model violations. Xtext's built-in validation features are relatively minimal, offering two forms of validation:

- **Domain-Specific Constraints -** Enforce the data model's mandatory fields, data point references, and the number of times each value can appear (cardinality).
- **Data Type Verification -** Verify data types associated with properties. For instance, a property expecting an integer value can be checked to ensure it receives a valid numeric input.

Taking the CICD model as an example, the 'Job' name is required to be define (Domain-Specific Constraint). If not defined in the DSL script, the Xtext editor will detect the problem, as illustrated in Figure 4.10.



```
Job
    name
```

Multiple markers at this line

- The required feature 'name' of 'cICD_metamodel.impl.JobImpl@213bcef{platform:/resource/CICD_DSL/src/CircleCI_Script.cicd#//@jobs.1}' must be set

Figure 4.10: Xtext CICD Validator

Building upon Xtext's built-in validation capabilities, we further developed custom validators to address specific requirements of the CI/CD pipeline DSL, increasing the likelihood of detecting mistakes as early as possible before deploying to the target pipelines. These validators can be split into the following categories:

Non-Mandatory Attribute Validation - While some attributes within the pipeline configuration might be optional, it is crucial to prevent empty or invalid values that could lead to unexpected behavior. Custom validators were implemented to specifically handle non-mandatory string and string list attributes. For non-mandatory string attributes, a custom validator verifies if the provided value is empty (''). If an empty string is encountered, the validator raises a warning or error, prompting the developer to provide a valid value. Similar to individual strings, custom validators ensure that string lists, even if non-mandatory, do not contain empty string values (''). The validator iterates through the list elements and flags any empty strings, notifying the user to provide a valid value. To reuse code, the **checkMandatoryStringNotEmpty** method was implemented for Empty String Validation and **checkMandatoryListNotEmpty** for Empty String in List Validation. An example of both these validations can be found in Listing 4.5. In this scenario, the custom validator (annotated with @Check) ensures that the MatrixConfig's 'name' attribute and the 'values' list do not contain empty strings. If the 'name' validation fails, the error message 'MatrixConfig name cannot be empty' (specified in MANDATORY_STRING_EMPTY) will appear.

Listing 4.5: Non-Mandatory Attribute Validation

```
@Check
public void checkAttributeNotEmpty(MatrixConfig matrixConfig) {
    checkMandatoryStringNotEmpty(matrixConfig.getName(),
        String.format(MANDATORY_STRING_EMPTY, "MatrixConfig name"),
        matrixConfig,
        "name",
        MANDATORY_MATRIX_CONFIG_NAME_EMPTY_ERRORCODE);

    checkMandatoryListNotEmpty(matrixConfig.getValues(),
        String.format(MANDATORY_STRING_EMPTY, "MatrixConfig values"),
        matrixConfig,
        "values",
        MANDATORY_MATRIX_CONFIG_VALUES_EMPTY_ERRORCODE);
}

//Auxiliary methods

private void checkMandatoryStringNotEmpty(String value,
    String errorMessage,
    Object object,
    String featureName,
    String errorCode) {

    if (value == null || value.trim().isEmpty()) {
        EStructuralFeature feature = ((EObject) object).eClass()
            .getEStructuralFeature(featureName);
        error(errorMessage, (EObject) object, feature, errorCode);
    }
}
```

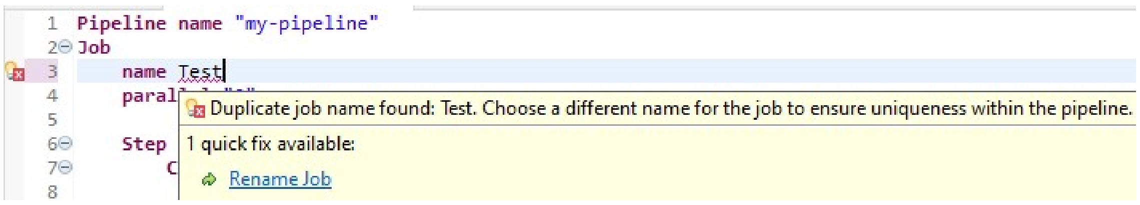
```
private void checkMandatoryListNotEmpty(List<String> values,
    String errorMessage,
    Object object,
    String featureName,
    String errorCode) {

    if (values.isEmpty() || values.stream().anyMatch(
        value -> value.equals(""))) {

        EStructuralFeature feature = ((EObject) object).eClass()
            .getEStructuralFeature(featureName);
        error(errorMessage, (EObject) object, feature, errorCode);
    }
}
```

Platform-Specific Validation - Pipeline setups are frequently subject to particular requirements and limitations specific to CI/CD platforms. To ensure compatibility and prevent platform-specific errors, custom validators were created after a thorough examination of each platform's documentation. Listing 4.6 includes a custom validator that detects duplicate jobs in the pipeline configuration and displays the error message 'Duplicate job name found' (given in DUPLICATE_JOB_NAME). This is a straightforward example of a restriction that, if not implemented, could not be checked using domain-specific constraints.

Listing 4.6: Platform-Specific Validation

```
@Check
public void checkNonDuplicateJobName(Job job) {
    if (job.eContainer() instanceof Pipeline) {
        Pipeline pipeline = (Pipeline) job.eContainer();
        for (Job otherJob : pipeline.getJobs()) {
            if (otherJob != job && otherJob.getName().equals(
                    job.getName())) {

                EStructuralFeature nameFeature = job.eClass()
                    .getEStructuralFeature("name");
                error(String.format(DUPLICATE_JOB_NAME,job.getName()),
                        job,
                        nameFeature,
                        DUPLICATE_JOB_NAME_ERRORCODE);
            }
        }
    }
}
```

Figure 4.11 shows an example of this error occurring, where two Jobs exist with ‘Test’ name. The following Xtext feature demonstrates how to handle this issue.



```
Pipeline name "my-pipeline"
Job
    name Test
    paral
    Step
        C
```

Duplicate job name found: Test. Choose a different name for the job to ensure uniqueness within the pipeline.

1 quick fix available:

Rename Job

Figure 4.11: Duplicate Name Validator

For the complete validation methods of all platform-specific models, visit GitHub.

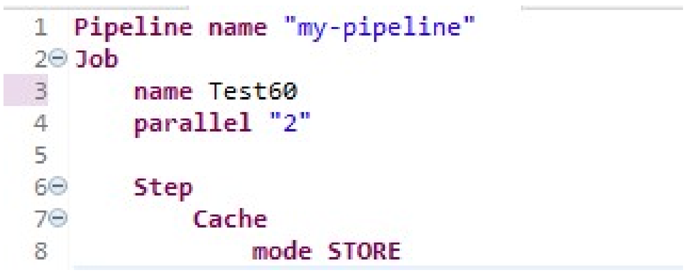
QuickFix - The quick-fix feature helps users resolve detected issues quickly and efficiently by providing automated solutions and corrective actions. When the previously stated validators identify faults or warnings during validation, Xtext provides a suitable quick-fix suggestion suited to each unique situation. In our case, these solutions involve code changes directly within the editor that address the underlying issues. Users can apply quick adjustments with little effort, shortening the iteration cycle and fostering a more seamless modeling experience. Xtext’s strength lies in its ability to link validators and quick-fix functionalities. This integration is possible by defining unique error codes for each customizable validator implemented, and when any inconsistencies are identified in the editor, the code is mapped to a quick-fix procedure that produces the same issue. The Xtext editor uses this association to give the user a contextualized suggestion for the situation at hand. Listing 4.7 provides a quick-fix (@Fix annotation) for the DUPLICATE_JOB_NAME error detected by the validator in Listing 4.6. This is made possible by each validator assigning a unique error code, in this case, DUPLICATE_JOB_NAME_ERRORCODE.

Listing 4.7: Quickfix

```
@Fix(CICDValidator.DUPLICATE_JOB_NAME_ERRORCODE)
public void fixDuplicateJobName(Issue issue,
    IssueResolutionAcceptor acceptor) {

    acceptor.accept(issue, "Rename Job",
    "Rename the job to ensure uniqueness.",
    null,
    new IModification() {
        public void apply(IModificationContext context)
            throws BadLocationException {

            IXtextDocument xtextDocument = context.getXtextDocument();
            Integer offset = issue.getOffset();
            Integer length = issue.getLength();
```

```
                String originalName = xtextDocument.get(offset, length);
                int randomInt = (int) (Math.random() * 100);
                String newName = originalName + randomInt;
                xtextDocument.replace(offset, length, newName);
            }
        });
}
```

We can observe from Figure 4.12 that the editor has resolved the issue of having two identical job names when selecting the 'Rename Job' option, and changed the name to 'Test60'. For the complete quickfix methods, visit GitHub.



```
Pipeline name "my-pipeline"
Job
    name Test60
    parallel "2"

    Step
        Cache
            mode STORE
```

Figure 4.12: Quickfix Duplicate name

4.4.4 Challenges

Several challenges arose throughout the solution's development, particularly when transforming models from various source formats using ATL. The graphical modeling component used XML Metadata Interchange (XMI) for its input models, while the textual component used Xtext files. Since ATL does not natively support text files as input, it was necessary to create two custom plugins to bridge this gap and ensure these model transformations.

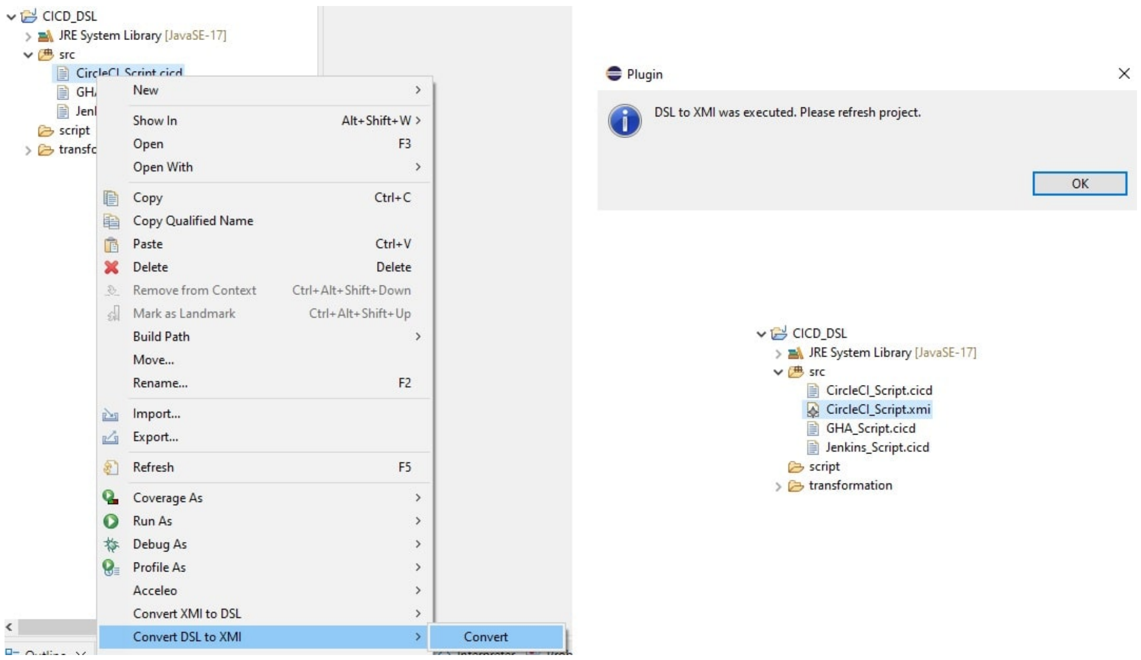
Convert Xtext Files to XMI - One of the primary challenges faced was the incompatibility between Xtext files and ATL, as ATL is designed to work with models serialized in XMI format. To address this, a custom plugin was developed to convert Xtext files into XMI (Listing 4.8). This plugin operates by parsing the Xtext files, interpreting their structure and content according to the defined grammar, and producing corresponding XMI representations. The conversion process involves:

- **Parsing Xtext Files -** Use the Xtext parser to read the textual syntax and semantics specified in the Xtext file
- **Model Resolution -** Resolve all references within the loaded DSL model
- **Serializing to XMI -** Convert Xtext contents into XMI format and add it to the new resource

Listing 4.8: DSL2XMI plugin

```java
public class DSLReader {

    public static void convertXText2XMI (String filePath, Shell shell) {

        ResourceSet resourceSet = new XtextResourceSet();
        URI uri = URI.createFileURI(filePath);
        Resource resource = resourceSet.getResource(uri, true);

        EcoreUtil.resolveAll(resourceSet);

        URI xmiUri = URI.createURI(uri.trimFileExtension().toString()+".xmi");
        Resource xmiResource = resourceSet.createResource(xmiUri);
        xmiResource.getContents().addAll(resource.getContents());

        try { xmiResource.save(null); }
        catch (IOException e) { e.printStackTrace(); }
    }
}
```

Figure 4.13 shows the editor's actions when running this transformation. After a successful execution, the editor displays a pop-up to refresh the project.



DSL to XMI was executed. Please refresh project.

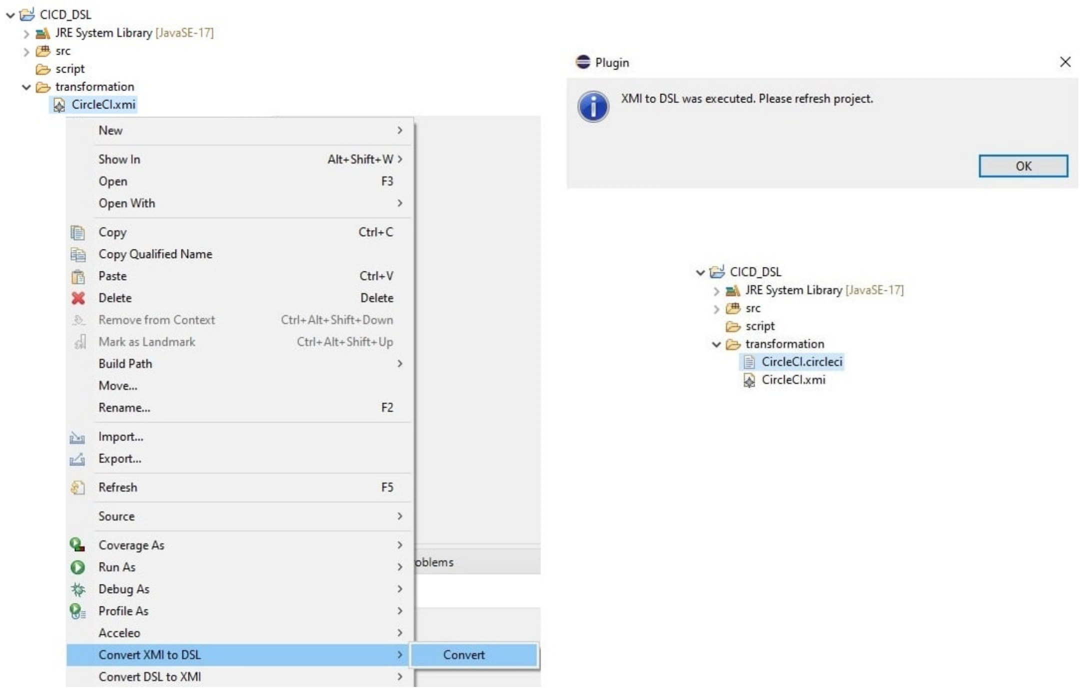
Figure 4.13: DSL2XMI Editor Action

Convert XMI to Xtext Files - After transforming models with ATL, the transformed XMI models must be converted back into Xtext files. This reverse conversion enables users to configure pipelines for specific DevOps platforms using the textual modeling approach.

Due to how the source files (.xmi files) were interpreted, the implementation logic of both plugins differed significantly. These files required further formatting since the .xmi file to DSL conversion compacted the configuration contents onto a single line in the target file. As a result, each platform required its formatter to handle the unique configuration while maintaining the correct pipeline semantics with the established language. Another issue examined was the final file extension name, which differed depending on the platform. We developed a simple mechanism (presented in Annex IV) to determine the platform configured in the .xmi file and append the appropriate extension. The conversion process involves:

- **Determine Extension -** Read the XMI files generated from the ATL transformations and interpreting their structure and content
- **Generate DSL Content -** Map and format the XMI with the appropriate formatter to generate DSL content
- **Serializing to Xtext Files -** Write the generated DSL content to a new file with the determined extension

Figure 4.14 shows the editor's actions when running this transformation. After successful execution, the editor displays a pop-up to refresh the project and a new file is generated with the corresponding configuration platform (.circleci).



XMI to DSL was executed. Please refresh project.

Figure 4.14: XMI2DSL Editor Action

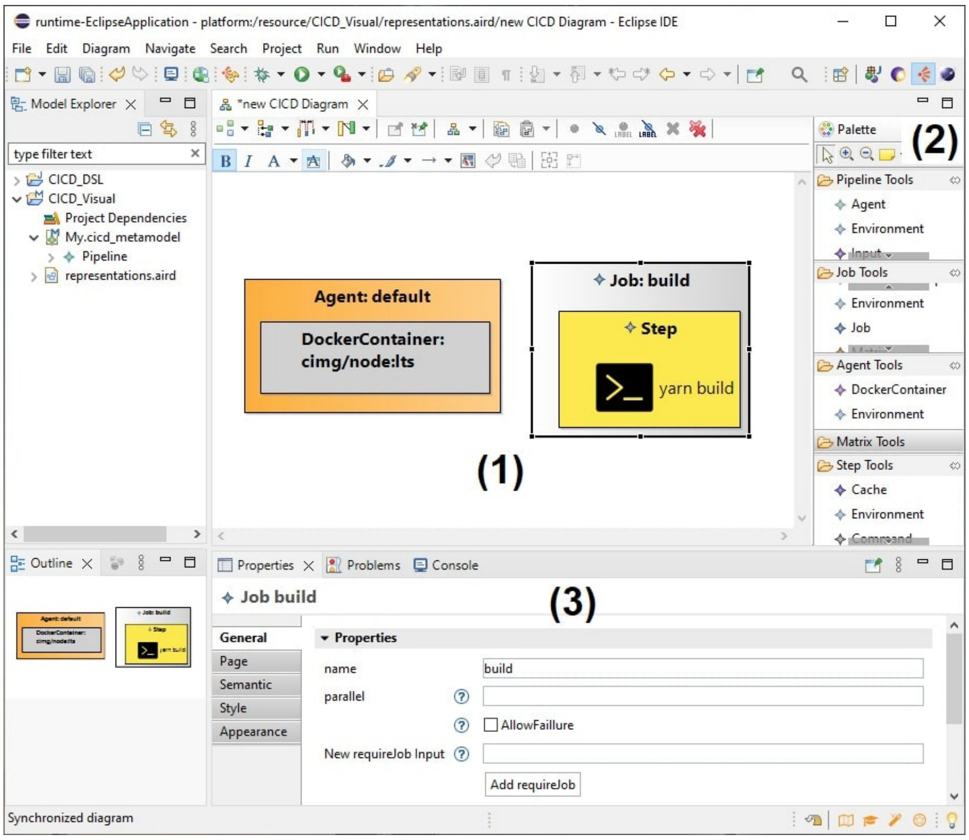
To visualize extracts of each platform formatter, check Annex V. For the complete formatters, visit GitHub.

4.5 Graphical Modeling

The ever-changing DevOps landscape demands adaptable and user-centric solutions for creating and managing CI/CD pipelines. Traditionally, developers have been limited to text-based configurations. While this approach provides a high level of control, it can be burdensome for complex pipelines and requires familiarity with specific syntax rules. Moreover, some developers, particularly those with less programming experience, might find it easier to visualize and configure pipelines through a graphical interface. Understanding the importance of flexibility and user-centric design, we embarked on a multi-faceted approach that accommodated both textual and graphical modeling paradigms, empowering users with the freedom to choose the modeling approach that best suited their individual needs and preferences. We now present the graphical approach.

4.5.1 Visual Language Design

In Eclipse Sirius, designing the visual language involves defining the main visual elements and their representations that illustrate the flow and structure of pipeline configurations. Figure 4.15 shows an overview of the Sirius modeling environment with a numbering of the main views.



name: build
parallel:
AllowFaillure
New requireJob Input:
Add requireJob

Figure 4.15: Sirius Environment

The design focuses on simplicity and clarity and includes two types of mapping: nodes, representing the primary elements of a CI/CD pipeline, and containers, used to group related nodes such as steps in a job. This layout helps organize the visual representation, ensuring that related elements are grouped visually to improve readability. In Figure 4.16, containers are shown as large labeled boxes that encompass the related nodes, while the nodes are represented by icons chosen to depict their purpose in the pipeline. The unique colors and icons for each node type enable users to easily distinguish between the different components of the pipeline.

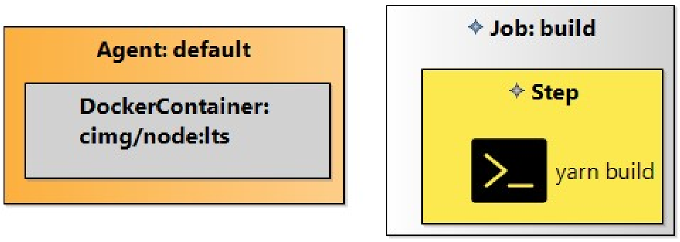
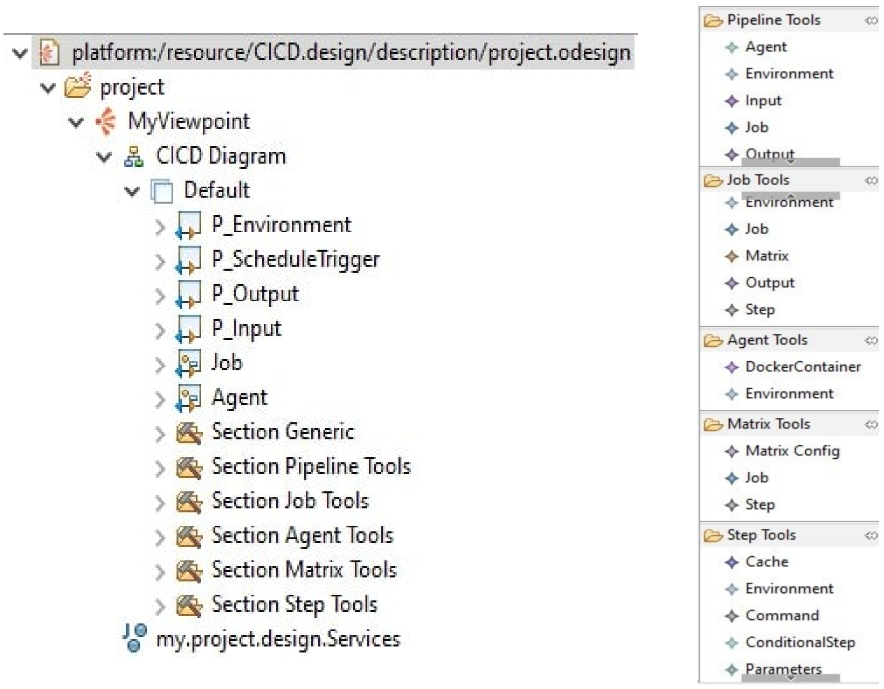


Figure 4.16: Sirius Diagram (1)

The mappings defined for each platform allow the configuration to be visualized using the diagram. However, to create it, sections with specific tools were created. Each section was designed to group the features of the pipeline's main components, with nomenclatures and an intuitive breakdown of what each component can do. This logic was applied to the remaining platforms. On the left side of Figure 4.17, it shows each node and container mappings implementation structure, and sections that possibilities their creation in the diagram, structured by the pallet on the right side.



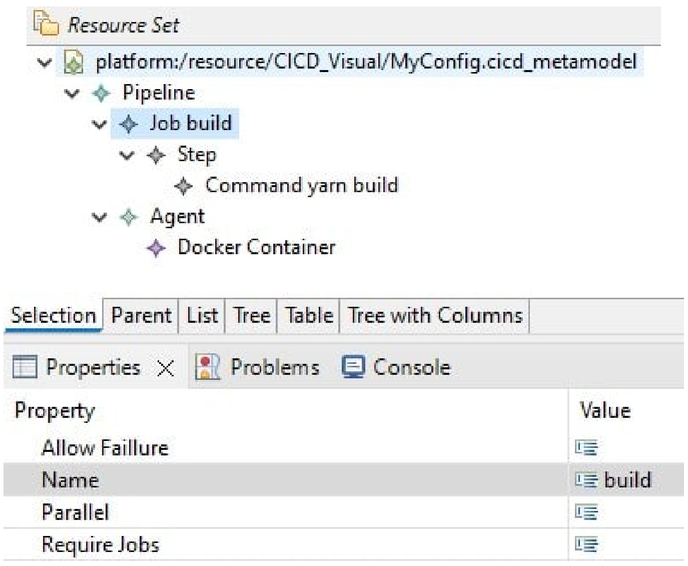
```
platform:/resource/CICD.design/description/project.odesign
  project
    MyViewpoint
      CICD Diagram
        Default
          P_Environment
          P_ScheduleTrigger
          P_Output
          P_Input
          Job
          Agent
          Section Generic
          Section Pipeline Tools
          Section Job Tools
          Section Agent Tools
          Section Matrix Tools
          Section Step Tools
      my.project.design.Services
```

```
Pipeline Tools
  Agent
  Environment
  Input
  Job
  Output
Job Tools
  Environment
  Job
  Matrix
  Output
  Step
Agent Tools
  DockerContainer
  Environment
Matrix Tools
  Matrix Config
  Job
  Step
Step Tools
  Cache
  Environment
  Command
  ConditionalStep
  Parameters
```

Figure 4.17: Diagram Design and Tools Section (2)

4.5.2 Tree View

Unlike the textual approach where configurations reside directly in code files, visual modeling in Eclipse Sirius leverages a different approach. Diagrams within Sirius are built upon XMI files, a standard format for exchanging metamodel information and act as the data storage for the diagram's visual elements. The Tree View is the default editor for XMI files (Figure 4.18), providing a hierarchical representation of the model elements and allow users to interact with and manipulate the underlying model data.



```
Resource Set
  platform:/resource/CICD_Visual/MyConfig.cicd_metamodel
    Pipeline
      Job build
        Step
          Command yarn build
      Agent
        Docker Container
```

Property	Value
Allow Faillure	
Name	build
Parallel	
Require Jobs	

Figure 4.18: Tree View

4.5.3 Tool Support

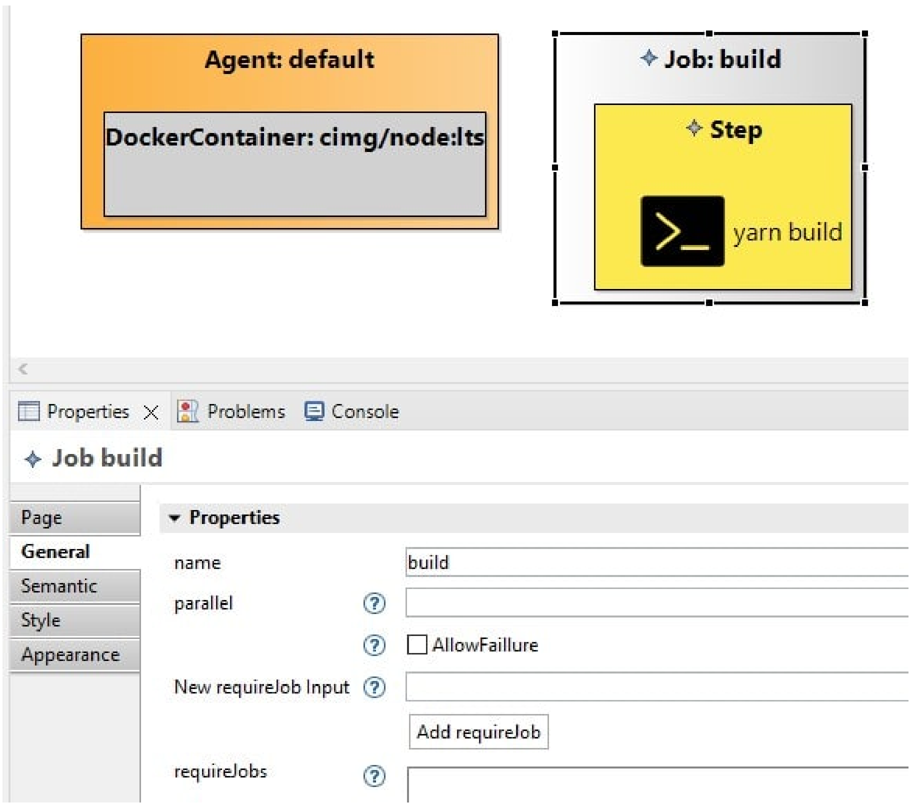
The visual modeling approach is supported by tools that improve the user experience, streamline development, and ensure configuration accuracy. This technique employs a set of features designed to assist users throughout the modeling process, including:

Visual Editing Environment - The tool provides an intuitive and user-friendly interface, allowing users to create, edit, and organize pipeline elements using drag-and-drop interaction on the graphical canvas.

Automatic Layout and Alignment - The tool incorporates automatic layout and alignment tools, facilitating the creation of clean and organized diagrams.

Error Checking - Real-time validation checks validate the configured models against predefined rules and constraints from the associated metamodels.

Contextual Editing Options - Context menus and property views provide editing choices for selected elements, alongside help expressions to help users manipulate pipeline components as per their specific requirements. Figure 4.19 shows an example of the Properties View for the selected 'build' Job to edit them accordingly.

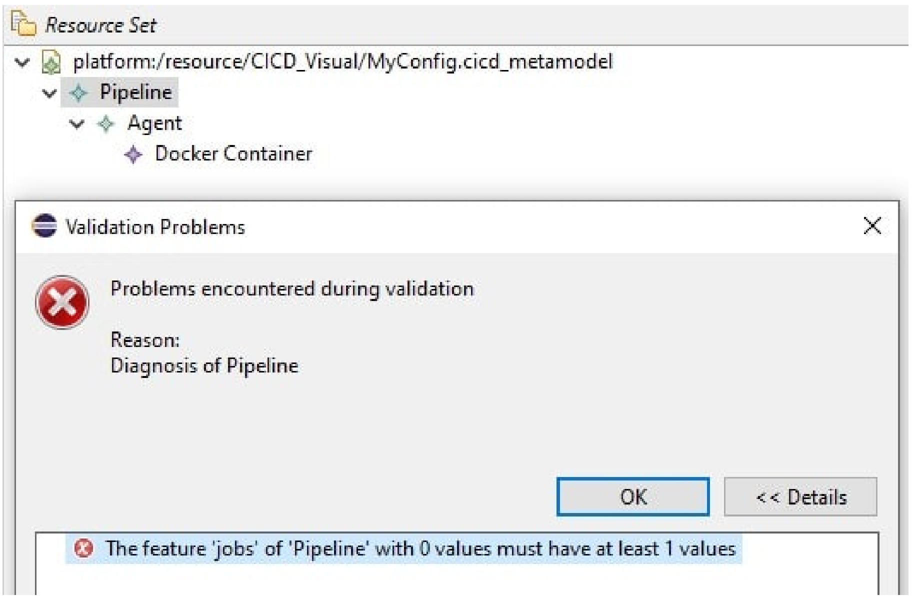


name: build
parallel:
AllowFaillure: ☐
New requireJob Input:
requireJobs:

Figure 4.19: Properties View (3)

Unlike Xtext, Sirius itself does not provide built-in validations in the diagram. This is because it specializes in visual modeling, emphasizing data representation and manipulation using graphics. However, the Tree View functionality comes in handy as it allows the validations inherited from the associated metamodels to be performed. Essentially, the Tree View serves as a bridge between the visual representation and the underlying data, enabling to refinement of pipeline configurations and benefit from the inherited validations from the metamodels. These validations have two types, as in the textual approach: Domain-Specific Constraints and Data Type Verification.

Continuing with the CICD model as an example, validating the configuration depicted in Figure 4.20 results in an error based on a Domain-Specific Constraint, where the 'jobs' reference should have at least one element.


Resource Set
platform:/resource/CICD_Visual/MyConfig.cicd_metamodel
Pipeline
Agent
Docker Container

Validation Problems

Problems encountered during validation

Reason:
Diagnosis of Pipeline

OK
<< Details

The feature 'jobs' of 'Pipeline' with 0 values must have at least 1 values

Figure 4.20: Tree View Validation

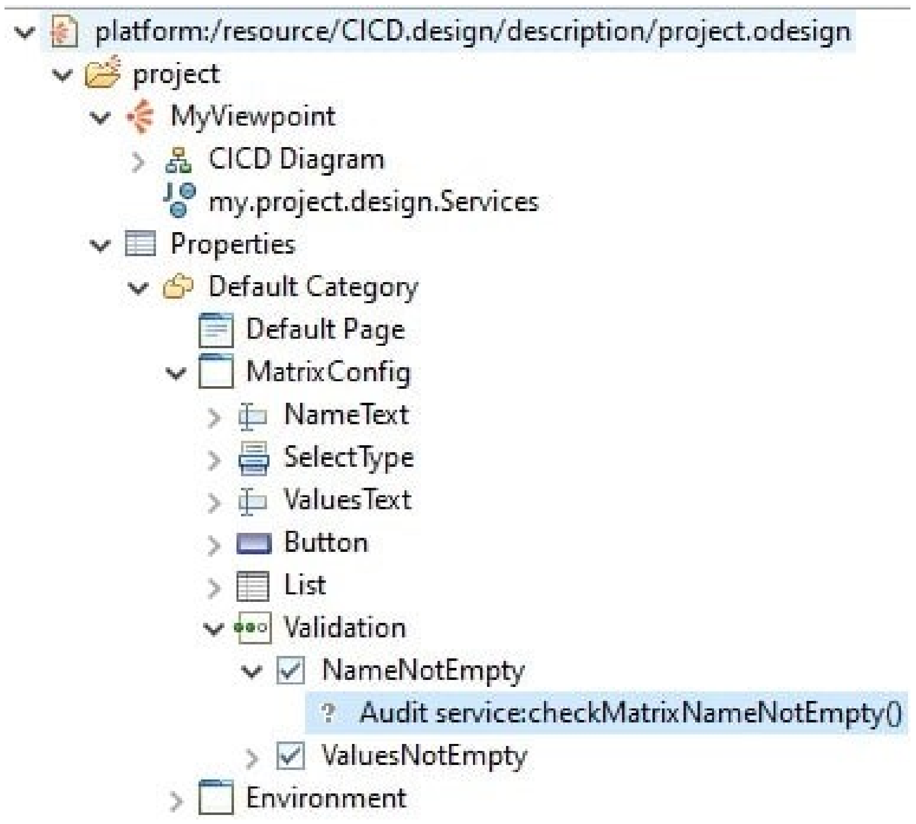
Similar to the textual modeling approach, custom validators were implemented to ensure the integrity and validity of configurations.

Non-Mandatory String Attributes - Just as with the textual approach, certain attributes within the pipeline configuration might be optional, yet it is vital to prevent empty or invalid values that could lead to unexpected behavior. For non-mandatory string attributes, a custom validator checks if the provided value is empty. If an empty string is encountered, the validator raises a warning or error, prompting the user to provide a valid value. Similarly, custom validators were implemented for non-mandatory string list attributes to ensure they do not contain empty string values. The validator iterates through each element in the list and flags any empty strings, notifying the user to provide a valid value.

Listing 4.9: Non-Mandatory Attribute Validation

```
public class Services {
    public boolean checkMatrixNameNotEmpty(MatrixConfig matrixConfig){
        return checkMandatoryStringNotEmpty(matrixConfig.getName());
    }

    public boolean checkMatrixValNotEmpty(MatrixConfig matrixConfig) {
        return checkMandatoryListNotEmpty(matrixConfig.getValues());
    }

    private boolean checkMandatoryStringNotEmpty(String value) {
        if (value == null || value.trim().isEmpty()) {
            return false;
        }
        return true;
    }

    private boolean checkMandatoryListNotEmpty(List<String> values) {
        if (values.isEmpty() ||
                values.stream().anyMatch(value -> value.equals(""))) {
            return false;
        }
        return true;
    }
}
```

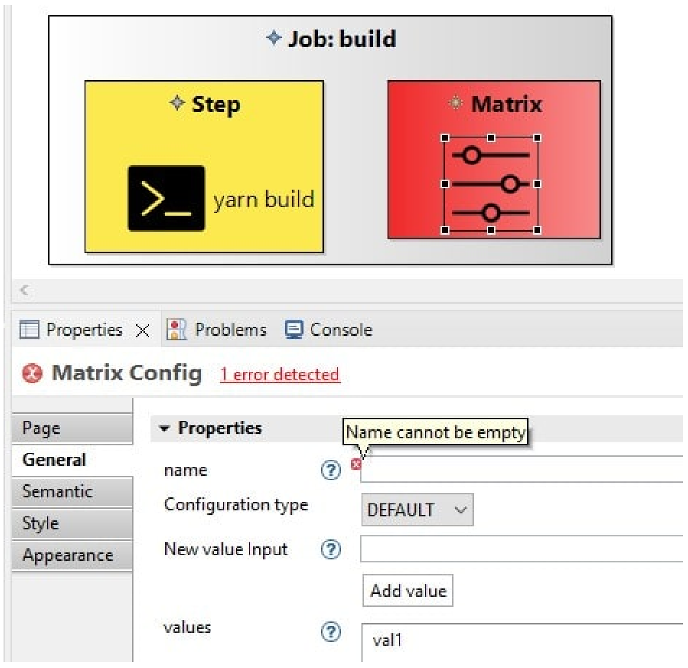
To link these methods to Sirius' validation process, it is necessary to establish validation rules for each element in the Properties View. These annotations serve as instructions for Sirius, indicating which validation method in Services.java should be called when a specific element is encountered in the pipeline. Figure 4.21 demonstrates this connection by creating a validation (NameNotEmpty) for the 'name' attribute of the MatrixConfig element and calling the 'checkAttributeNotEmpty' method from Listing 4.9 to proceed with runtime evaluations of this restriction.



```
platform:/resource/CICD.design/description/project.odesign
  project
    MyViewpoint
      CICD Diagram
      my.project.design.Services
    Properties
      Default Category
        Default Page
        MatrixConfig
          NameText
          SelectType
          ValuesText
          Button
          List
          Validation
            NameNotEmpty
              ? Audit service:checkMatrixNameNotEmpty()
            ValuesNotEmpty
        Environment
```

Figure 4.21: Sirius Validation

When a user selects an element during runtime, Sirius invokes the designated validation methods based on the annotations and executes the specified validation logic, examining the element's data according to the established rules. If any issues are detected by the custom validators, Sirius presents them in the Validation Page of the Properties View. This offers users immediate feedback on potential errors or inconsistencies in their pipeline configuration. Figure 4.22 demonstrates this case where the user defined a MatrixConfig with an empty 'name', resulting in a 'Name cannot be empty' error.



Matrix Config 1 error detected

Name cannot be empty

name

Configuration type DEFAULT

New value Input

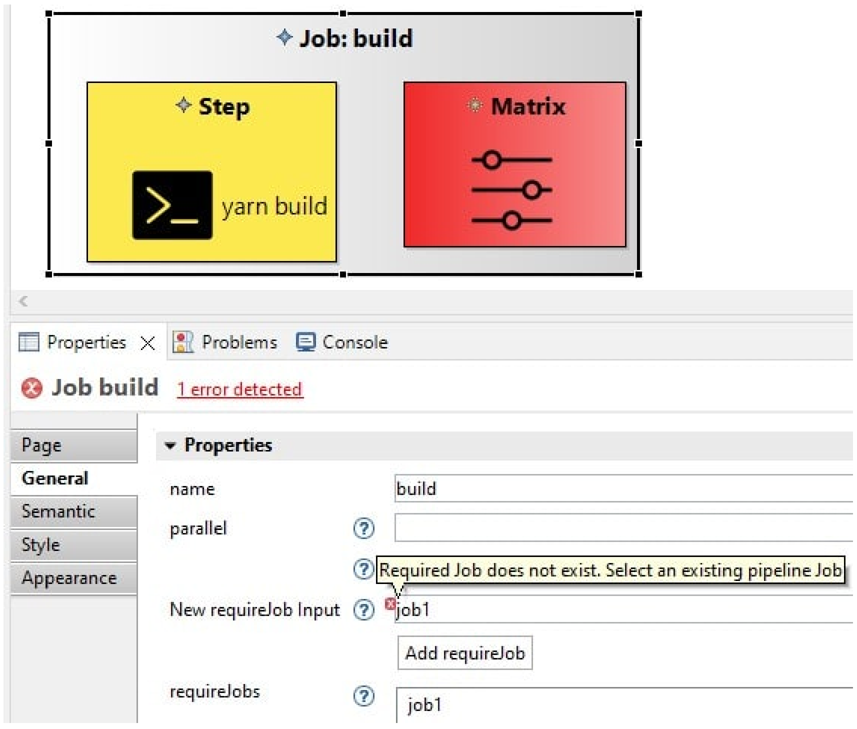
values val1

Figure 4.22: Sirius Validation Error Message

Platform-Specific Validation - Specific validations were needed to address the unique requirements and limitations of different CI/CD platforms. Each platform has its own set of rules and constraints that must be followed for successful deployment and execution of pipelines. Listing 4.10 shows a custom method in the 'Services.java' class to verify if all Jobs in the pipeline configuration exist in the 'requireJobs' attribute, and Figure 4.23 the corresponding error.

Listing 4.10: Platform-Specific Validation

```
public class Services {
    public boolean checkRequiredJobExists(Job job) {
        if (job.eContainer() instanceof Pipeline &&
                !job.getRequireJobs().isEmpty()) {

            for (String requireJob : job.getRequireJobs()) {
                boolean jobExists = false;
                for (Job j : (Pipeline) job.eContainer().getJobs()) {
                    if (requireJob.equals(j.getName())) {
                            jobExists = true;
                            break;
                        }
                    }
                    if (!jobExists) { return false; }
                }
            }
            return true;
        }
}
```



name: build
parallel:
Required Job does not exist. Select an existing pipeline Job
New requireJob Input: job1
requireJobs: job1

Figure 4.23: Required Job Exists Validator

4.6 Platform-Independent to Platform-Specific Transformations

Previously, we discussed the importance of being able to convert pipeline configurations between different models. This capability is essential for creating a flexible and adaptable CI/CD pipeline configuration tool that can work with multiple DevOps platforms without being limited by their characteristics. In this context, ATL [27] was used to facilitate PI to PS transformations.

4.6.1 ATL Overview

ATL is a declarative, rule-based language that is specifically designed for specifying model transformations within the MDE framework. It allows for the definition of mappings between a PI model and a PS model. The PIM captures the essence of the system without being tied to a specific platform, while the PSM translates this essence into the specific constructs and syntax of a target platform.

4.6.2 Transformation Workflow

The transformation process using ATL involves several key steps to ensure that generic pipeline models are accurately converted into the specific syntax and semantics required by different DevOps platforms. The workflow can be divided into the following stages:

- **Model Loading -** The source models, defined using the generic metamodel, are loaded into the ATL transformation engine. These models represent the abstract configuration of CI/CD pipelines, independent of any specific platform.
- **Transformation Execution -** The core of the ATL process involves executing transformation rules. These rules are defined to map elements from the platform-independent metamodel to corresponding elements in the platform-specific metamodel. The rules encapsulate the logic required to translate generic pipeline configurations into the specific constructs and configurations expected by the target platform.
- **Model Validation and Verification -** Post-transformation, the generated platform-specific models are validated to ensure they conform to the target platform's metamodel. This step is crucial to catch any inconsistencies or errors that might have arisen during the transformation process.

To illustrate the transformation process from a generic CI/CD pipeline model to a specific model for CircleCI, consider the following ATL transformation rules (Listing 4.11).

Listing 4.11: CICD2CircleCI ATL

```
-- @path CICD=/CICD_metamodel/model/cICD_metamodel.ecore
-- @path CircleCI=/CircleCI_metamodel/model/circleCI_metamodel.ecore

module CICD2CircleCI;
create OUT : CircleCI from IN : CICD;

rule Pipeline2Pipeline {
    from
        s : CICD!Pipeline
    to
        t : CircleCI!Pipeline(
            version <- '2.1',
            setup <- false,
            jobs <- s.jobs->collect(job | thisModule.Job2Job(job,
                if s.agents->notEmpty() then
                    if not s.agents->first().labels->notEmpty() then
                        s.agents->first().labels->first()
                    else
```

```
                        OclUndefined
                    endif
                else
                    OclUndefined
                endif)),
            commands <- s.inputs ->collect(i | thisModule.Input2Command(i)),
            workflows <- thisModule.ScheduleTrigger2Workflow(s),
            executors <- s.agents ->collect(agent |
                thisModule.DockerContainer2Docker(agent.container,
                    agent.labels->first()))
        )
}

lazy rule Input2Command {
    from
        s : CICD!Input
    to
        t : CircleCI!Command (
            name <- 'Com' + s.name,
            description <- OclUndefined,
            parameters <- thisModule.Input2Parameter(s),
            steps <- thisModule.DummyStep(s)
        )
}
```

In this example, the Pipeline2Pipeline rule maps a generic pipeline to a CircleCI pipeline. It transforms the pipeline's jobs, inputs, schedule triggers, and agents into their CircleCI-specific counterparts. Similarly, the Input2Command rule transforms each generic input into a CircleCI command, and the DummyStep rule maps generic inputs to CircleCI run steps. The DummyStep rule illustrates how some platform-specific functions are required, leading to the need to create indirect mapping rules using the source model to satisfy certain limitations. The remaining platforms snippets for ATL transformations can be found in Annex VI. See GitHub for the complete transformation rules.

4.7 Code Generation

While ATL excels in M2M transformations within MDE, another crucial part of modern software development is generating executable code for deployment through DevOps platforms. Acceleo [26] is an effective tool in this domain since it automates the development of code from textual and visual models.

Acceleo provides numerous advantages as it seamlessly integrates with EMF which allows for easy access and manipulation of model elements, improving efficiency in navigating and retrieving data from the models. Furthermore, the use of control flow components and logic, such as conditional statements and loops, allows developers to design complex code structures from model data. Additionally, the support for various expressions enables dynamic content generation within code templates, giving developers the freedom to do calculations, string operations, and conditional checks.

4.7.1 Code Generation Mechanism

The code generation process leverages AQL templates to translate the abstract representations of CI/CD pipelines into platform-specific scripts. These templates define the rules for transforming model elements into executable code.

The process begins with input models created using either a textual DSL (developed in Xtext) or a graphical model (developed using Sirius). These models are instances of a platform-independent metamodel. Acceleo templates containing AQL queries are then executed against these input models to generate corresponding code snippets. The final output of this process is a set of platform-specific YAML scripts, which can be directly deployed on the respective DevOps platforms.

Listing 4.12 shows a template snippet of a CircleCI configuration file based on the input pipeline model. The **generateElement** template is the primary entry point, which creates a YAML file using the pipeline's name and checks if it has version and setup properties. If present, these fields are included in the generated file. The template then calls other templates, such as **generateOrbs**, to make the reusable "orbs" section of this configuration. It checks if the orbs collection in the Pipeline object is empty and iterates through each orb, retrieving its key-value pairs. These pairs are then inserted into the generated YAML file as part of the CircleCI configuration. Annex VII contains snippets of the remaining paltforms. See GitHub for the complete templates.

Listing 4.12: CircleCI Template

```
[comment encoding = UTF-8 /]
[module generate('http://www.example.org/circleCI_metamodel')]

[template public generateElement(aPipeline : Pipeline)]
[file ('CircleCI' + '.yml', false, 'UTF-8')]
version: [aPipeline.version/]
[if (aPipeline.setup) ]
setup: [aPipeline.setup/]
[/if]
[generateOrbs(aPipeline)/]
[generateCommands(aPipeline)/]
[generateExecutors(aPipeline)/]
[generateJobs(aPipeline)/]
[generateWorkflows(aPipeline)/]
[/file]
[/template]

[template public generateOrbs(aPipeline : Pipeline)]
[if (aPipeline.orbs->notEmpty()) ]
orbs:
[for (o: Orb | aPipeline.orbs)]
    [o.key/]: [o.value/]
[/for]
[/if]
[/template]
```

5 Evaluation

This chapter describes the usability evaluation conducted to determine the effectiveness, efficiency, and user satisfaction of the developed solution, along with its conclusions. We also assessed the effort required for the prototype's usage.

5.1 Planning

A cornerstone of successful human-computer interaction lies in user-centered design. To ensure the proposed CI/CD pipeline definition solution caters effectively to user needs, a comprehensive usability evaluation was conducted. The following structured phases comprise the experiment planning and serve as a guide for assessing the experiment's usability.

5.1.1 Objectives Definition

The main goal of this usability evaluation is to assess the efficiency, effectiveness, and user satisfaction of both the textual and graphical modeling approaches. Additionally, the evaluation aims to determine if the developed tools improve the use and accuracy of scripts in the pipeline configuration process compared to the traditional CircleCI approach. The planning phase laid the groundwork for the evaluation process. The primary objectives were fourfold:

Evaluation Goal	**Question**	**Metric**
Prototype Usability	How usable is the Prototype?	SUS score
Prototype Task Load	How much effort is required a user to use the Prototype?	NASA-TLX score
Prototype Usage Precision	How accurately can users perform tasks using the Prototype?	Precision score
Prototype Usage Recall	How effectively can users recall when using the Prototype?	Recall score

Table 5.1: Goal-Question-Metric

5.1.2 Participants' Demography and Selection

Participants in the experiment include technical and non-technical users with different levels of experience in managing CI/CD pipelines to ensure practical, real-world feedback. This includes professional developers, DevOps engineers and computer science students who are completing or have already obtained their degrees. Demographic data such as age, gender, job title and years of experience with CI/CD tools were collected to better understand the sample's diversity. Participants were recruited through professional networks and social media platforms, ensuring a balanced representation in terms of age, gender, and professional experience. This approach aimed to produce evaluation results that reflected a wide range of users.

5.1.3 Experiment Materials

A selection of materials was used to collect data and insights while evaluating the usability of the proposed CI/CD modeling solution. All experimental material is available in Zenodo.

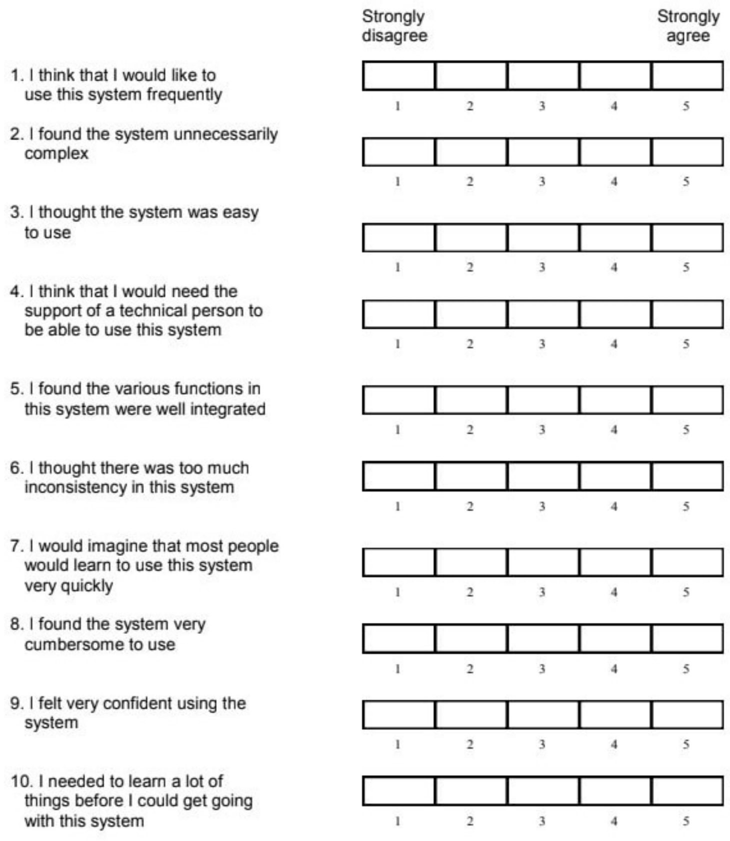
- **Demographic Questionnaire -** The Demographic Questionnaire was used to gather information about the participants' background, such as age, gender, education level, and experience in relevant fields. This information is valuable for understanding the characteristics of the participants and how their backgrounds may influence their perceptions and experiences during the evaluation process. By collecting demographic data, we can gain insights into how different demographic factors may impact the usability of the CI/CD modeling solution, allowing for a more comprehensive analysis of the evaluation results.

- **Tasks -** Participants were instructed to complete pre-defined tasks involving CI/CD pipeline configuration at different levels of complexity, reflecting real-world development scenarios. They started with a tutorial to guide them on using CircleCI syntax to configure a CircleCI script according to the instructions and examples provided in the given platform. They then progress to a more intricate configuration, aiming to evaluate participants' understanding of the tutorial and their ability to interpret and rectify error messages. Intentional errors were introduced for this purpose, and participants were informed about their number. Each participant tested one approach provided by the researcher, either textual or visual, and used similar industry tools as a comparison, such as VS Code for the textual, and Buddy for the visual approach.

  All the scripts were previously validated using the CircleCI extension in VSCode and through the API in PowerShell to validate syntactically and semantically when deploying the script in a forked project on the CircleCI platform. To ensure the correctness and reliability of the generated code, several validation and testing steps were carried out:

- **Model Validation -** Before code generation, the input models were validated against the metamodel to ensure they conform to all defined constraints
- **Template Testing -** Acceleo templates were thoroughly tested to verify that they produce the correct output for a wide range of input models. This involves unit tests that compare the generated scripts against expected outputs
- **Syntax Checking -** The generated scripts were checked for syntax errors specific to the target CI/CD platform. For instance, YAML linting tools and static checking were used to ensure the generated YAML files were syntactically correct

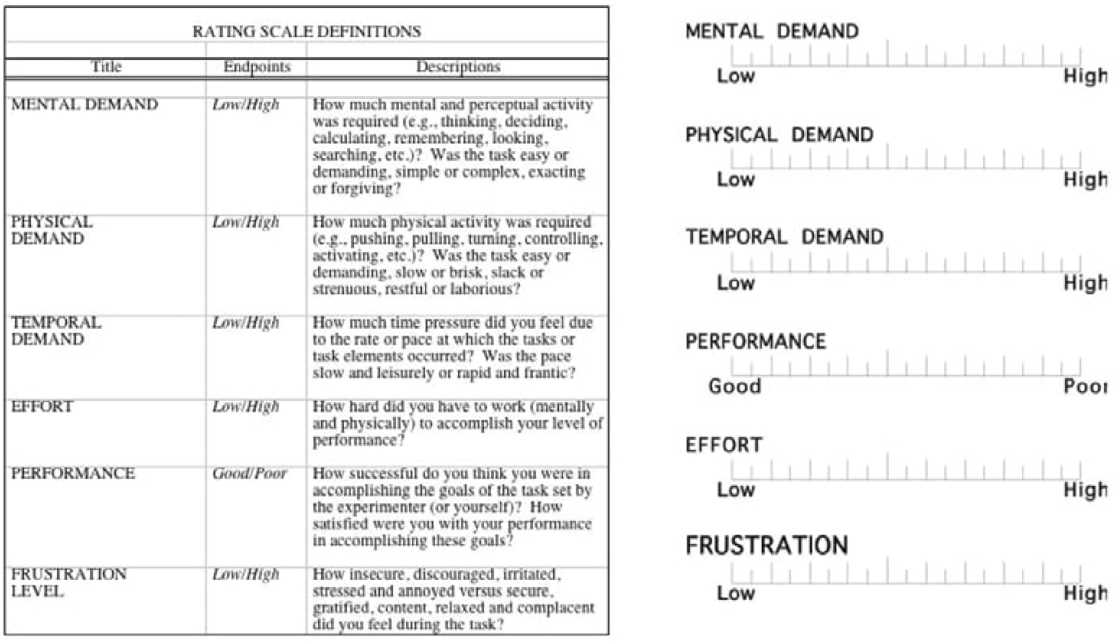
- **System Usability Scale (SUS) -** SUS is a questionnaire-based method for assessing the usability of a wide variety of systems [11]. It consists of 10 statements graded on a scale from 1 to 5, with higher scores indicating better usability. Positive statements, such as 1,3,5,7 and 9, have a score contribution of the scale position minus one. For negative statements 2, 4, 6, 8 and 10, the contribution is 5 minus the scale position. The SUS score is then calculated by summing the score contributions from each statement and multiplying the sum by 2.5. This score ranges from 0 to 100, with higher scores corresponding to theoretically higher usability.



	Strongly disagree				Strongly agree
1. I think that I would like to use this system frequently	1	2	3	4	5
2. I found the system unnecessarily complex	1	2	3	4	5
3. I thought the system was easy to use	1	2	3	4	5
4. I think that I would need the support of a technical person to be able to use this system	1	2	3	4	5
5. I found the various functions in this system were well integrated	1	2	3	4	5
6. I thought there was too much inconsistency in this system	1	2	3	4	5
7. I would imagine that most people would learn to use this system very quickly	1	2	3	4	5
8. I found the system very cumbersome to use	1	2	3	4	5
9. I felt very confident using the system	1	2	3	4	5
10. I needed to learn a lot of things before I could get going with this system	1	2	3	4	5

Figure 5.1: System Usability Scale

- **NASA-Task Load Index (TLX) -** The NASA-TLX [56] questionnaire evaluates a tester's workload in six dimensions: mental demand, physical demand, temporal demand, performance, effort, and frustration. Each dimension is scored on a scale of 1 to 100 with 20 possible values (5, 10, 15, up to 100). The final NASA-TLX score is determined as the average of these scores, with higher scores indicating more effort required. The score can be weighted or unweighted. The tester selects the dimension that contributes the most to the burden from 15 pairings of the six dimensions. The weight of each dimension is determined by how many times it is selected as more impactful. In this study, all dimensions have equal weight, and the tester selects values ranging from 1 to 10 for each dimension. This questionnaire offers valuable insights into the cognitive, physical, and temporal demands of a task, as well as the subjective experience of effort and frustration.



RATING SCALE DEFINITIONS

Title	Endpoints	Descriptions
MENTAL DEMAND	*Low/High*	How much mental and perceptual activity was required (e.g., thinking, deciding, calculating, remembering, looking, searching, etc.)? Was the task easy or demanding, simple or complex, exacting or forgiving?
PHYSICAL DEMAND	*Low/High*	How much physical activity was required (e.g., pushing, pulling, turning, controlling, activating, etc.)? Was the task easy or demanding, slow or brisk, slack or strenuous, restful or laborious?
TEMPORAL DEMAND	*Low/High*	How much time pressure did you feel due to the rate or pace at which the tasks or task elements occurred? Was the pace slow and leisurely or rapid and frantic?
EFFORT	*Low/High*	How hard did you have to work (mentally and physically) to accomplish your level of performance?
PERFORMANCE	*Good/Poor*	How successful do you think you were in accomplishing the goals of the task set by the experimenter (or yourself)? How satisfied were you with your performance in accomplishing these goals?
FRUSTRATION LEVEL	*Low/High*	How insecure, discouraged, irritated, stressed and annoyed versus secure, gratified, content, relaxed and complacent did you feel during the task?

Figure 5.2: NASA Task Load Index

5.1.4 Hypothesis Formulation

The evaluation was guided by the following hypotheses:

Usability of Dual Modeling Approach

The first set of hypotheses aims to evaluate the effectiveness of the dual modeling approach in enabling users to configure CI/CD pipelines efficiently, regardless of their level of expertise in DevOps.

$H_{\mathbf{0DualModeling}}$: The dual modeling approach is not effective or usable from the user's perspective for configuring and managing CI/CD pipelines.

$H_{\mathbf{1DualModeling}}$: The dual modeling approach is effective and usable from the user's perspective for configuring and managing CI/CD pipelines.

User Experience and Perceived Value of Features and Implementations
The second set of hypotheses is centered on the general user experience and the perceived value of the features and implementations, assessing whether they facilitate the use and correctness of the scripts in the pipeline configuration process.

$H_{\mathbf{0FeaturesValue}}$: The participants consider that the features and implementations do not provide any added value in terms of facilitating the use and correctness of the scripts in the pipeline configuration process.

$H_{\mathbf{1FeaturesValue}}$: The participants consider that the features and implementations provide added value in terms of facilitating the use and correctness of the scripts in the pipeline configuration process.

Comparison with CircleCI Approach
The third set of hypotheses aims to compare the usability of the dual modeling approach with CircleCI's approach, evaluating whether the proposed solution helps users to better configure and maintain the correctness of the scripts.

$H_{\mathbf{0ComparisonCircleCI}}$: The dual modeling approach, alongside its features, suggestions, and implementations, does not help users better configure and maintain the correctness of the scripts compared to CircleCI's approach.

$H_{\mathbf{1ComparisonCircleCI}}$: The dual modeling approach, alongside its features, suggestions, and implementations, helps users better configure and maintain the correctness of the scripts compared to CircleCI's approach.

Consequently, this study seeks to reject the null hypotheses (H0) and provide evidence that supports the effectiveness, usability, and added value of the proposed solution. The results will be analyzed to determine whether the proposed solution meets the intended usability and effectiveness criteria, and how it compares to existing solutions such as CircleCI.

5.1.5 Experiment Session Plan

The design of our experiment session plan was a critical aspect in ensuring the validity and reliability of our usability evaluation. Initially, we considered two primary options for structuring our evaluation:

- **Option 1 -** Participants would perform 2-3 tasks, including one simpler task and one more complex task, that was consistent across all participants. This approach would allow us to analyze performance across different task complexities and identify if our tool performs better for certain types of tasks.

- **Option 2 -** A within-subjects design where each participant would engage with both the graphical and textual modeling approaches. This would involve participants performing different tasks that are equivalent in terms of difficulty but vary in nature to avoid redundancy. Some participants would start with one approach and then switch to the other to counterbalance any learning effects.

While the initial plan was to leverage a full within-subjects design where participants would test both textual and visual approaches, practical considerations regarding participant burden and evaluation duration led to an alternative approach. Participants were exposed to tutorials for the approach provided by the researcher to establish a baseline understanding.

However, the actual evaluation tasks involved completing tasks for only one assigned approach (textual or visual), alongside a relevant industry-standard tool (VS Code for textual and Buddy for visual). This modified approach offered a balance between capturing some of the benefits of a within-subjects design (reduced bias, smaller sample size potential) while addressing practical limitations of time and participant burden. Additionally, it allowed for the inclusion of industry-standard tools (VS Code and Buddy) as a point of comparison for user experience evaluation.

Task Script Development

The task script was developed through an iterative process. Pilot tests were conducted to ensure clarity, appropriate complexity levels, and effectiveness in evaluating user interaction with the solution and industry-standard tools. Based on pilot test results, the script was continuously refined to guarantee a robust and informative evaluation experience.

To ensure the tasks reflected real-world scenarios, all scripts were designed as CircleCI configurations. This choice aligns with the study's goal of comparing our solution to a market platform like CircleCI. Notably, CircleCI provided access to a visual and textual approach, although very feature-limited [CircleCI Visual Editor].

Unfortunately, the CircleCI visual editor became unavailable during the final stages of task development. To maintain consistency within the evaluation, the CircleCI editor was replaced with industry-standard tools offering similar functionalities. Buddy was chosen for the visual approach, and VS Code for the textual approach. Importantly, all scripts retained their target configuration for CircleCI, ensuring a valid comparison despite the tool substitution.

Counterbalancing Order

To further strengthen the evaluation and control for potential learning effects, a counterbalanced order was implemented. Here's the breakdown of the four scenarios:

- **Scenario 1 -** Visual Approach and Buddy

- **Scenario 2 -** VS Code and Textual Approach
- **Scenario 3 -** Buddy and Visual Approach
- **Scenario 4 -** Textual Approach and VS Code

By implementing this counterbalanced order, we aimed to minimize the influence of the order in which participants encountered the approaches on their performance. Each scenario appears an equal number of times at the beginning of the evaluation sequence, ensuring that any order effects are evenly distributed across the participant pool. This approach helps to ensure that any observed differences in user experience or performance can be attributed to the specific approach and industry-standard tool combination rather than the order of exposure.

5.2 Execution

The execution phase involved conducting the usability evaluation sessions based on the plan developed during the planning phase. Each session followed a structured procedure to ensure consistency and reliability in data collection.

5.2.1 Sessions Procedure

The usability evaluation started with the researcher briefing participants about the information and objectives. The participant was then given a Zoom session with a remote control and assigned an approach to test, which began with reading the document and answering some demographic questions. After reading, participants performed predefined tasks using the assigned approach and tool. The researcher was available to address technical questions or unexpected technical issues. After completing the tasks, participants filled out SUS and NASA-TLX questionnaires and provided additional feedback through open-ended questions. The entire study was conducted using Google Forms and lasted between 40 and 60 minutes, with all participants answering within this range.

5.3 Results

This section delves into the data collected during the usability evaluation to understand user experience, task performance, and overall solution effectiveness. Both quantitative and qualitative methods were employed to gain a comprehensive understanding of the results. Additionally, participants' demographics are also presented to contextualize and ensure a comprehensive understanding of the diversity of the user base.

5.3.1 Demographic Data

The demographic data collected from 24 respondents reveals that 14 participants are 23 years old, 7 are 22 years old, and the remaining fall within the ages of 24, 26, and 27. All respondents are male. Regarding their primary field of study, 79.2% of respondents are from Computer Science, with the remaining 20.8% coming from Engineering, and all have Master's Degree as their highest degree obtained.

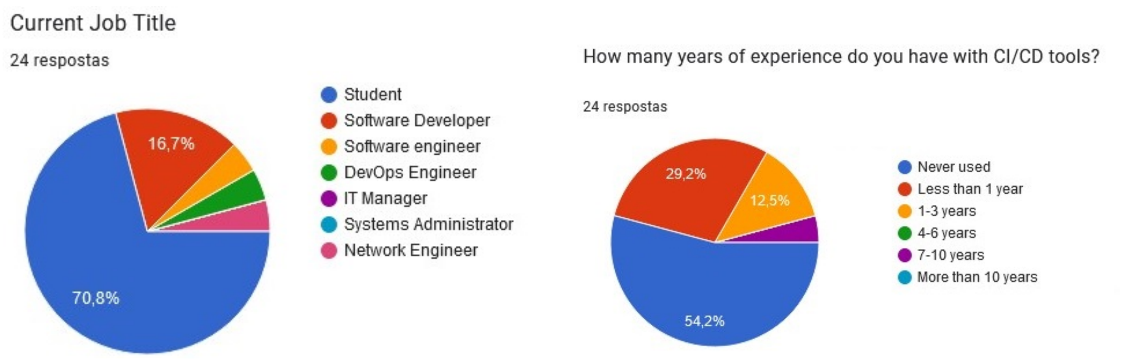


Figure 5.5: Demographic Data

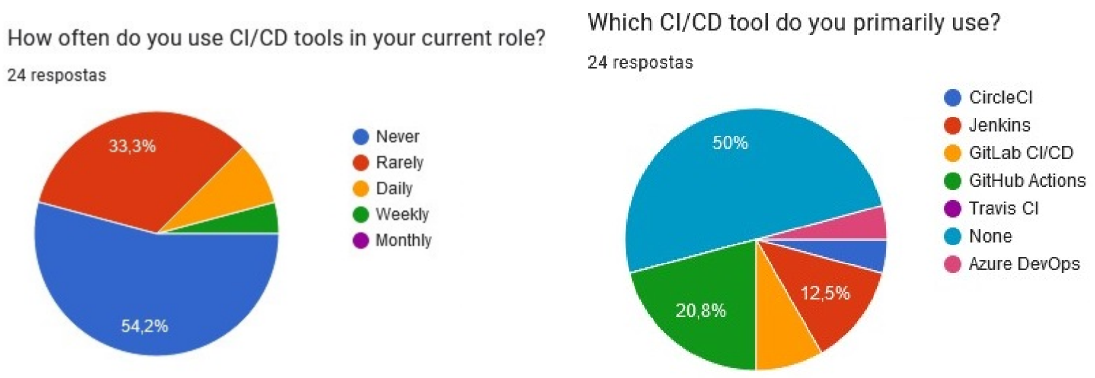


Figure 5.8: Demographic Data

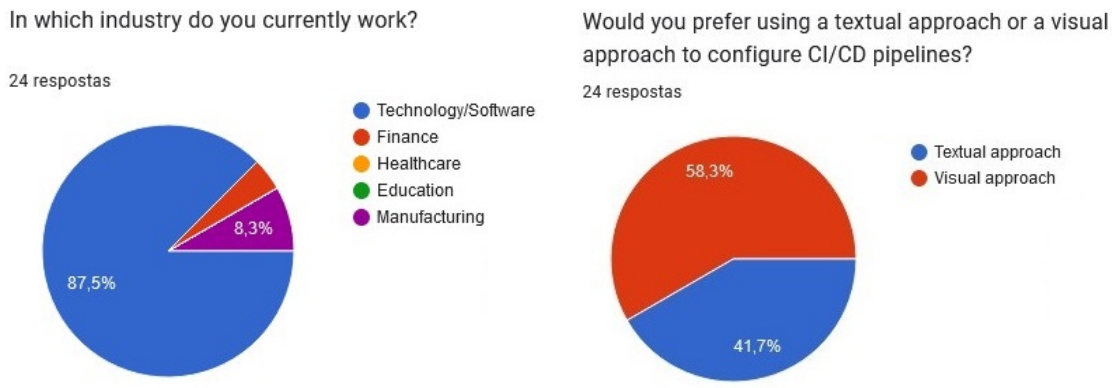


Figure 5.11: Demographic Data

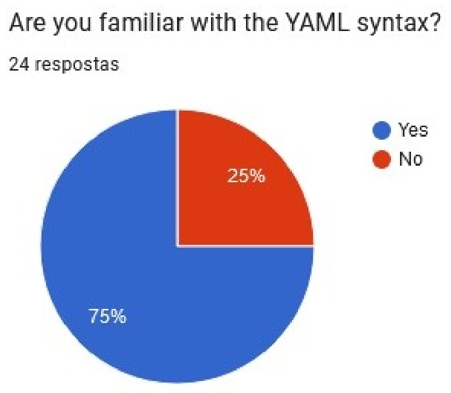


Figure 5.12: Modeling Preferences

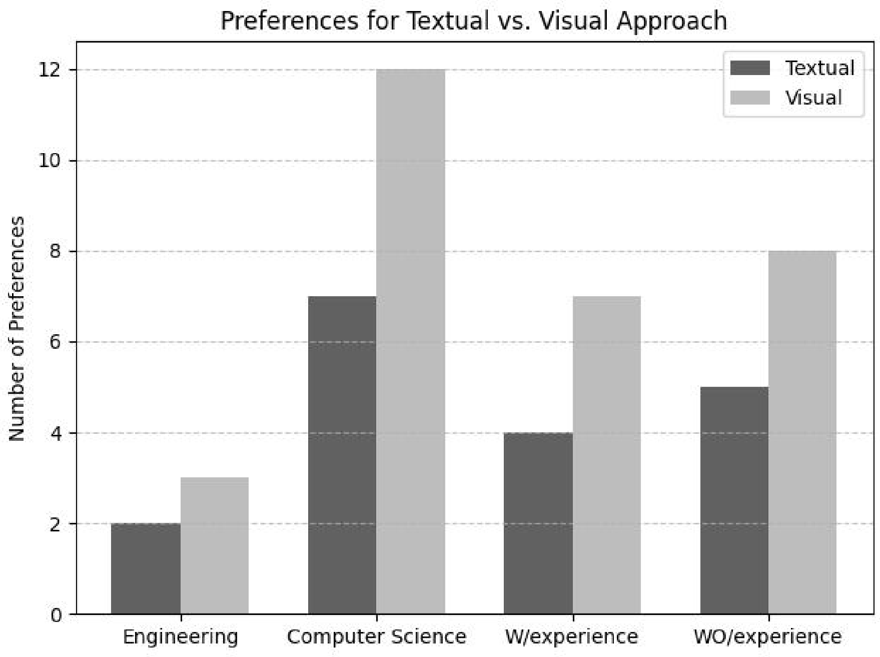


Figure 5.13: Modeling Preferences

Figure 5.13 illustrates the preferences of users from different backgrounds and experience levels between textual and visual tools.

5.3.2 Quantitative Data

We wanted to make statistically sound conclusions about observed trends. To achieve this, we conducted inferential statistical tests by assessing data normality using the Shapiro-Wilk, which refers to whether the data distribution resembles a bell curve. If the p-value is greater than the chosen significance level (usually 0.05), the null hypothesis of normality is rejected, suggesting the data may be reasonably close to a normal distribution. If the p-value is less than 0.05, the null hypothesis is not rejected, indicating the data distribution is likely not normal.

Comparing means required testing for homogeneity of variance with Levene's test, assuming equal variances across groups. If the p-value is less than the chosen significance level (usually 0.05), the null hypothesis is rejected, indicating heterogeneity. After checking normality, we selected either a t-test (for normal data with equal variances) or a Mann-Whitney U test (for non-normal data or unequal variances) to compare group means.

Table 5.2 and 5.3 presents the summary statistics of user test results for usability (measured by SUS) and task load (measured by NASA-TLX). The scores are derived from the participants' scores. Other usability graphs can be found at Annex VIII.

Table 5.2: Usability Scores

Approach	**Mean**	**Std Deviation**	**Min**	**25%**	**50%**	**75%**	**Max**
Textual	85.2	7.93	67.5	81.875	86.25	89.375	95.0
VS Code	56.04	15.49	30.0	43.75	62.5	62.5	85.0
Visual	75.6	10.9	60.0	71.25	75.0	80.0	100
Buddy	79.8	13.3	50.0	71.25	82.5	90.0	97.5

Table 5.3: Workload Scores

Approach	**Mean**	**Std Deviation**	**Min**	**25%**	**50%**	**75%**	**Max**
Textual	36.14	8.02	20.37	32.9	35.4	42.6	50.0
VS Code	43.52	12.86	18.52	37.04	42.6	53.24	64.81
Visual	34.5	9.7	16.7	27.3	33.3	45.8	48.2
Buddy	30.6	9.6	16.7	23.1	27.8	37.0	46.3

Usability
Figure 5.14 illustrates the distribution of SUS scores across the tools studied.

The SUS scores for the Textual approach show a median of 85 with an interquartile range (IQR) from 81 to 89, and whiskers extending from 75 to 95, with one outlier at 67.5. The mean is slightly below the median, causing a minor skew towards lower scores. For VS Code, the median score is about 63, with an IQR from 44 to 62, and whiskers ranging from 30 to 85. There is a high skew toward lower scores with the mean below the median. Textual display higher median SUS scores compared to the VS Code tool, with a smaller spread of scores representing a more consistent usability rating.

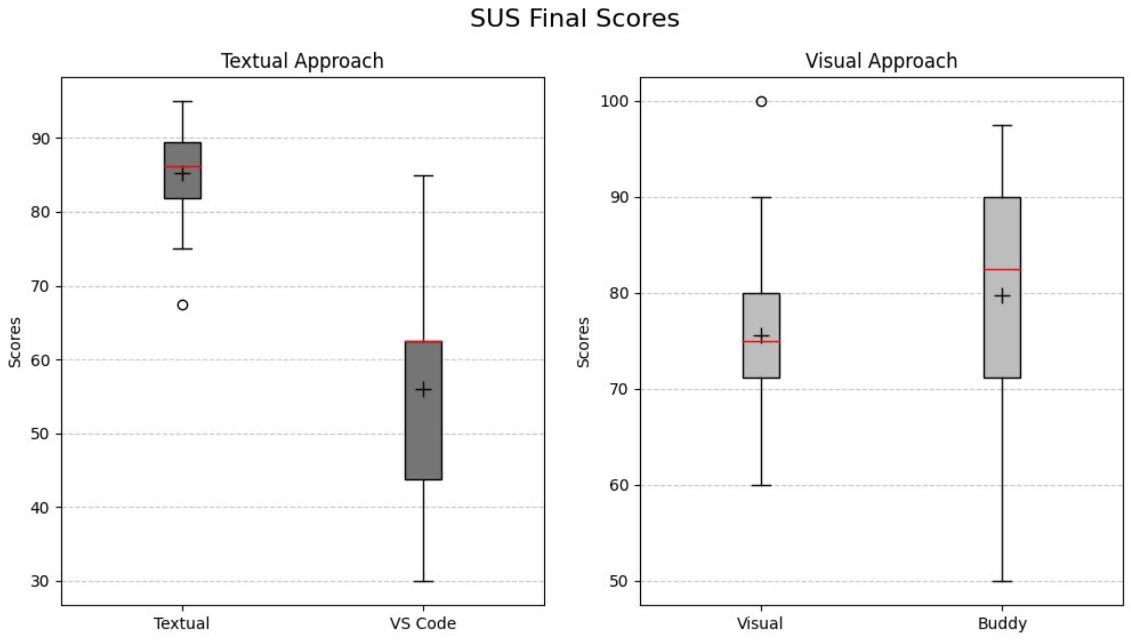


Figure 5.14: SUS Final Scores

In Visual, the median score is around 75, with an IQR from 70 to 80, and whiskers extending from 60 to 90, with one outlier at 100. The mean is slightly above the median, indicating a slight skew for higher scores. However, this skew may be influenced by the outlier. Buddy's median score is about 82, with an IQR from 71 to 90, and whiskers ranging from 50 to 97, and the mean is below the median. Both tools have similar median scores, but Buddy exhibits a slightly larger spread, representing more variability in user rating.

Table 5.4: SUS Textual vs VS Code

Statistic	Result	P-value
T-Test	5.56	1.38e-05

Table 5.5: SUS Visual vs Buddy

Statistic	Result	P-value
T-Test	-0.81	0.43

The statistical analysis (Table 5.4 and 5.5) reveals a significant difference in SUS scores between the Textual and VS Code approaches, with the Textual interface achieving higher usability ratings. Conversely, the Visual and Buddy interfaces show similar SUS scores, but the difference in scores is not statistically significant, indicating comparable levels of usability.

Workload
Figure 5.15 shows the distribution of NASA-TLX scores for the four tools, highlighting perceived workload levels among users.

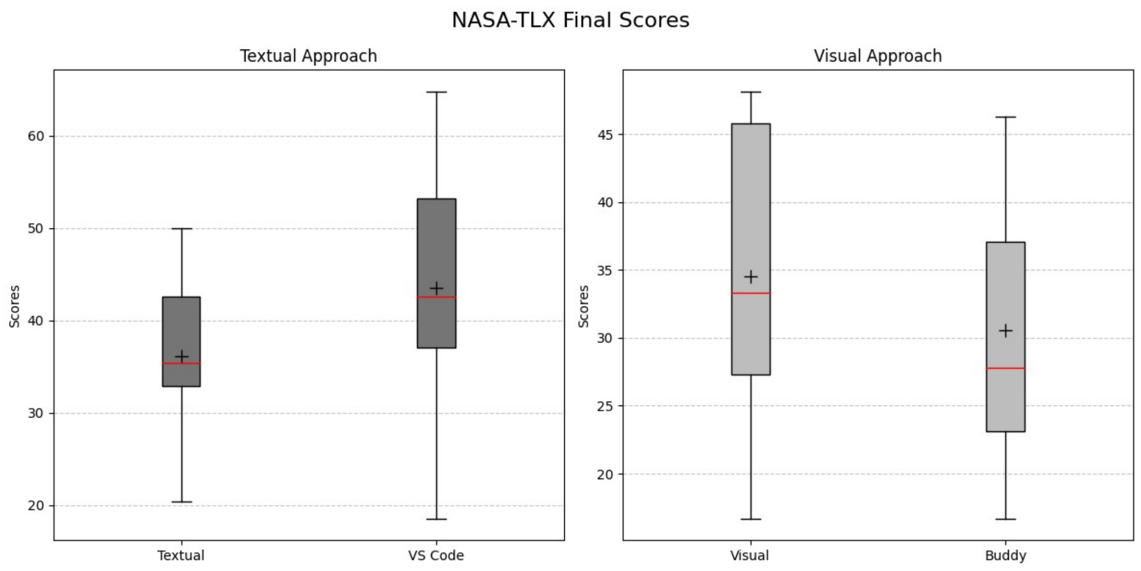


Figure 5.15: NASA-TLX Final Scores

The Textual median score is approximately 38, with an IQR from around 33 to 42. Whiskers extend from about 20 to 50. The mean is slightly above the median, indicating a slight skew towards higher scores. For VS Code, the median score is around 43, with an IQR ranging from 37 to 53. Whiskers extend from 18 to about 65, showing greater variability in the workload. The mean is slightly above the median, suggesting a slight skew towards higher scores.

Table 5.6: NASA-TLX Textual vs VS Code

Statistic	**Result**	**P-value**
T-Test	-1.614	0.12

The Visual median score is roughly 35, with a moderate variability IQR from 30 to 42. Whiskers extend from about 20 to 50, with a mean above the median, indicating a slight skew towards higher scores. For Buddy, the median score is around 35, with an IQR from 25 to 42. Whiskers extend from approximately 20 to 50, with no outliers. The mean is slightly above the median, suggesting a skew towards higher scores.

Table 5.7: NASA-TLX Visual vs Buddy

Statistic	**Result**	**P-value**
T-Test	0.96	0.347

The statistical analysis (Table 5.6 and 5.7) reveals moderate to strong correlations in NASA-TLX scores between the compared tools. However, the T-Test results indicate that these differences are not statistically significant, suggesting that the perceived workload among the tools is comparable.

Figure 5.16 and 5.17 compare NASA-TLX scores across six workload dimensions, identifying areas where one interface induces significantly higher workloads. Both charts indicate minimal Physical Demand scores for all interfaces, with values close to zero. This aligns with the nature of the tasks, which do not require any physical demands.

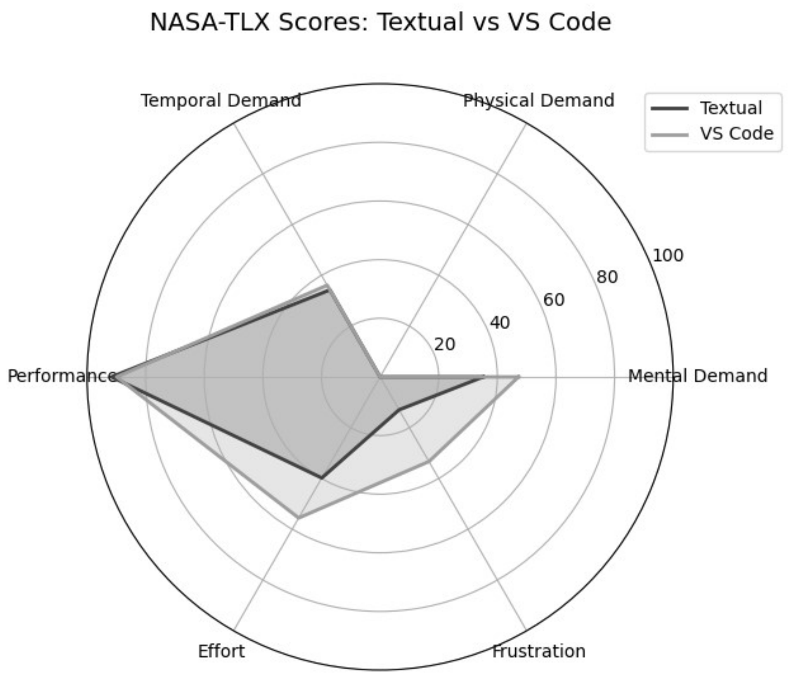


Figure 5.16: NASA-TLX Dimensions Scores Textual vs VS Code

In Figure 5.16, the Temporal, Physical, and Performance dimensions have identical scores for both interfaces. However, there are some disparities in other dimensions, such as the Mental Demand dimension, with VS Code showing a slightly higher mental demand than Textual, being 1.3 times higher. The effort required is somewhat higher for VS Code compared to Textual, at almost twice the effort. Frustration levels are also disparate, with values for VS Code being 3.5 times higher. To statistically analyze these differences, the following tests were conducted:

Table 5.8: NASA-TLX per Dimension Textual vs VS Code

Statistic	Result	P-value
T-Test	-0.578	0.574

The statistical analysis supports the observation that there is no significant difference between Textual and VS Code in terms of usability as measured by NASA-TLX scores, as indicated by the high p-value of 0.574.

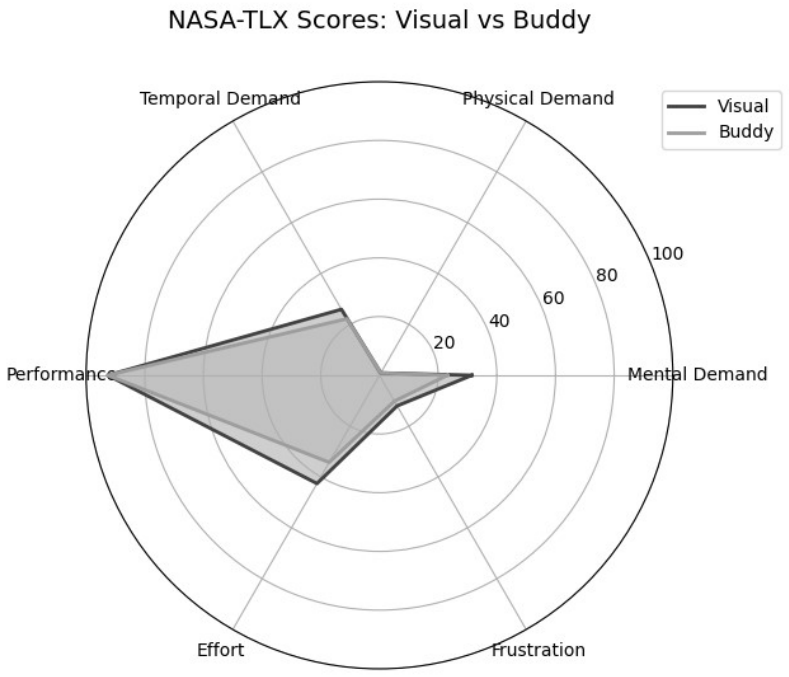


Figure 5.17: NASA-TLX Dimensions Scores Visual vs Buddy

Visual interfaces show less variation and lower values than textual interfaces. In this graph, the most significant variation is in the Effort dimension, with the visual interface showing a slightly higher mental demand than the buddy interface (1.25 times higher). Regarding Frustration, Mental Demand and Temporal Demand, the Visual interface shows slightly higher values than the Buddy interface, but with less disparity.

Table 5.9: NASA-TLX per Dimension Visual vs Buddy

Statistic	Result	P-value
Mann-Whitney U Test	30	0.521

Similar to the Textual vs. VS Code comparison, the statistical analysis for Visual vs. Buddy shows no significant difference in usability scores. This reinforce that both Visual and Buddy provide a comparable level of usability and user satisfaction.

SUS/NASA-TLX

Figure 5.18 shows the relationship between SUS and NASA-TLX scores for each tool, indicate the correlation between usability and perceived workload. The best outcomes are found in the bottom right corner, maximizing usability and reducing workload. In contrast, the poorest outcomes are found in the top left corner with a reversal of the results.

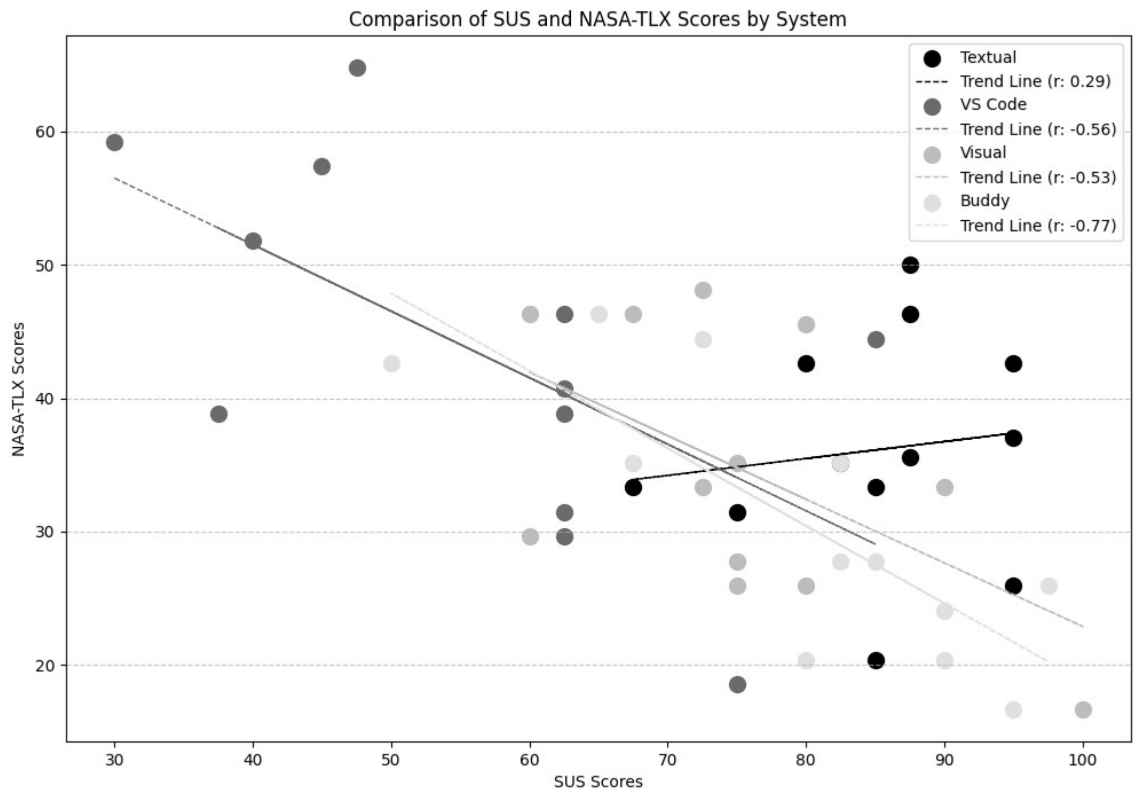


Figure 5.18: Comparison of SUS and NASA-TLX Scores by System

The textual scatter plot shows a slight positive correlation (r = 0.29) between these variables, where as usability increases, the workload slightly increases as well. The data points are clustered around medium NASA-TLX and high SUS scores. The statistical analysis indicates a non-significant p-value of 0.360, meaning that improvements in perceived usability do not have a statistically significant impact on the task load for the Textual tool, contrary to the scatter plot.

For VS Code, the Spearman correlation indicates a moderate to strong negative relationship (r=-0.56) between usability and task load, with a p-value of 0.059. The data points are more concentrated along the negative trend line, with higher usability scores generally associated with lower perceived task load.

For the Visual tool, Spearman (r=-0.53) correlation shows a moderate inverse relationship between usability and task load, with a p-value of 0.075. While these trends are not statistically significant, they indicate that higher usability could be associated with lower perceived task load, though this relationship is not definitive. The points on the graph are moderately scattered but generally follow the negative trend indicated by the trend line.

Lastly, Buddy shows the strongest negative correlation, with Spearman (r=-0.77) correlation being statistically significant (p-value of 0.004). This indicates a significant relationship where higher usability is associated with a reduced task load for the Buddy tool. The graph reflects this strong negative correlation with points closely aligned along the trend line.

Table 5.10: Comparison of SUS and NASA-TLX Scores by System

Tool	Pearson Correlation (P)	Spearman Correlation (S)	P-value (P / S)
Textual	0.13	0.29	0.697 / 0.360
VS Code	-0.60	-0.56	0.039 / 0.059
Visual	-0.53	-0.53	0.073 / 0.075
Buddy	-0.80	-0.77	0.002 / 0.004

Participants Experience Level

Figure 5.19 examines the impact of user experience, classified as novice or experienced, on perceived usability.

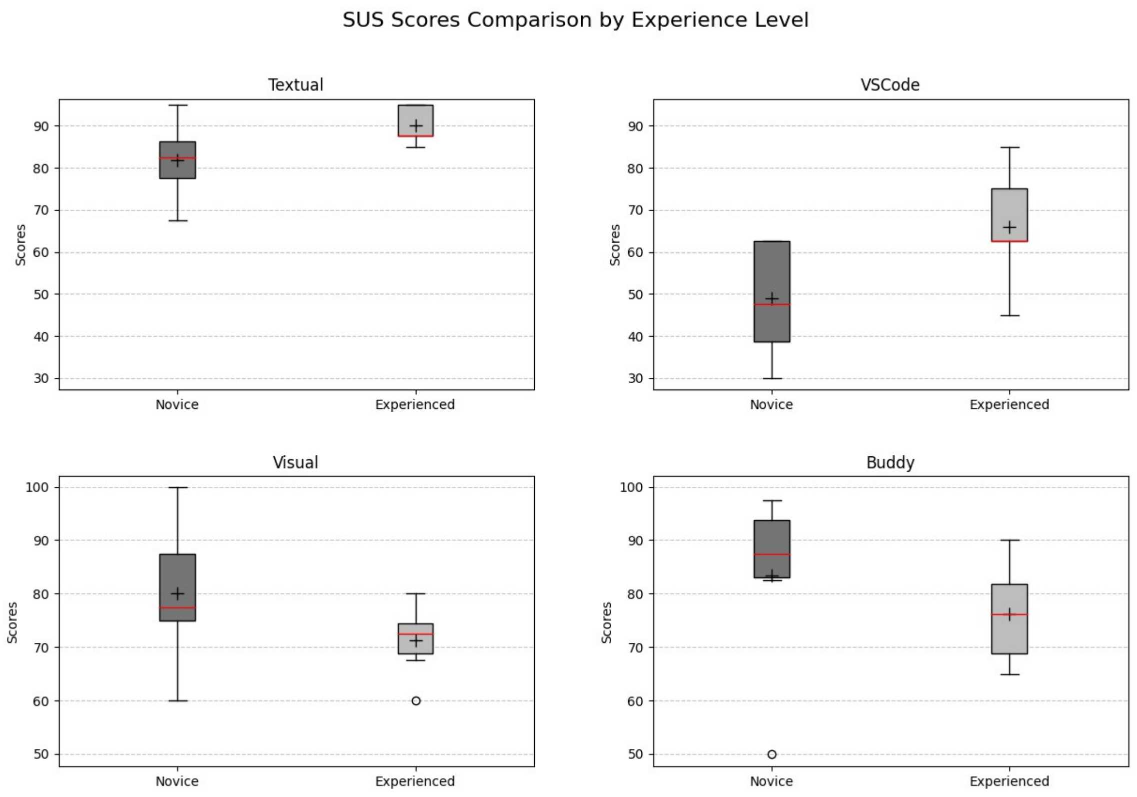


Figure 5.19: SUS Scores Comparison by Experience Level

Novice users of Textual present a median SUS score of around 82.5, with an IQR of approximately 78 to 90. The range of usability experiences among novices varied broadly, indicated by the whiskers extending from about 70 to 95. The mean is close to the median, suggesting a relatively symmetrical distribution. Experienced users' median SUS score increases to about 90 and a narrower IQR ranging from roughly 87.5 to 95. The whiskers span from 85 to 95 indicating a more consistent perception of usability. The mean is slightly above the median, suggesting a minor skew towards higher scores.

For VS Code, novice users have a median SUS score of around 50, with the IQR extending from about 40 to 62. The whiskers range broadly from 30 to 62, reflecting significant variability in usability experiences among novices. The mean score, slightly above the median, indicates a slight skew towards higher ratings. For experienced users, the median SUS score increases to around 62, with an IQR from 62 to 77.5. The whiskers extend from 45 to almost 90, showing less variability compared to novices. The mean is above the median, indicating a minor skew towards lower scores.

Novice users of Visual have a median SUS score of around 77.5, with the IQR spanning from 75 to about 87.5. The whiskers extend from 60 to 100, indicating some variability but generally favorable usability perceptions. The mean is higher than the median, suggesting a skew towards higher scores. Experienced users, however, show a lower median score of about 75, with the IQR ranging from approximately 70 to 75. The whiskers extend from about 70 to 80, indicating less variability but a generally lower perception of usability compared to novices. The mean, slightly below the median, suggests a minor skew towards lower scores.

For Buddy, novice users have a high median SUS score of about 87.5, with an IQR of roughly 85 to 95. The whiskers extend from about 85 to roughly 100, generally high usability perceptions. The mean is below the median, suggesting a minor skew towards lower ratings. Experienced users show a lower median score of around 77.5, with the IQR ranging from approximately 70 to 82. The whiskers extend from 65 to 90, indicating more consistent usability perceptions compared to novices. The mean is the same as the median.

Textual modeling indicates higher usability scores and is more concentrated for experienced users, with VS Code showing the most significant difference between novices and experts, while still presenting lower SUS scores. Visual modeling approaches differed more between novice and experienced users, whereas the Visual tool clearly showed this difference among these users with more concentrated scores for experienced ones. Nevertheless, Buddy demonstrate better uusability results for both user categories.

Table 5.11: SUS Scores Comparison by Experience Level

Tool	**T-Test Statistic**	**P-value**
Textual	-1.876	0.09
VS Code	-2.046	0.068
Visual	1.393	0.194
Buddy	0.877	0.401

The statistical analyses (Table 5.11) of all interfaces show that although there are noticeable differences in SUS scores between novice and experienced users, these differences are not statistically significant. Therefore, the level of user experience does not significantly impact the perceived usability of the interfaces examined.

Similar to this chart, Figure 5.20 explores the influence of user experience on perceived workload.

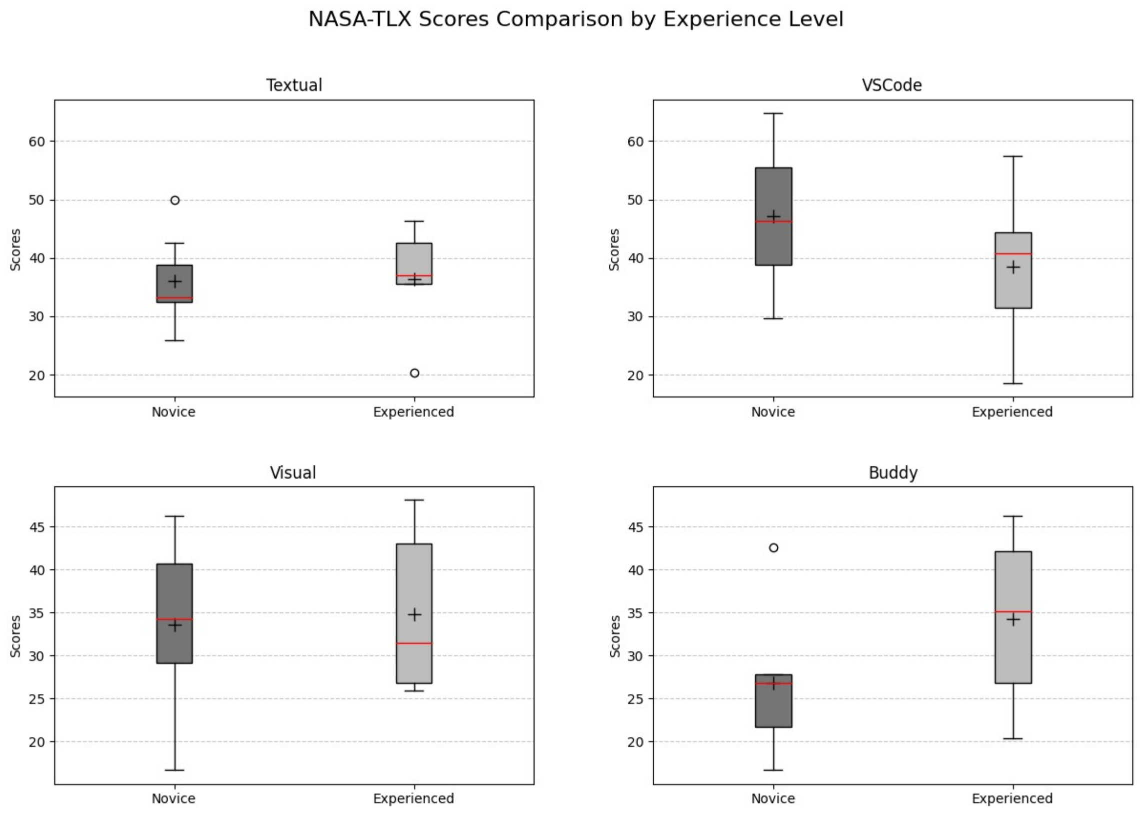


Figure 5.20: NASA-TLX Scores Comparison by Experience Level

The perception of workload, as measured by NASA-TLX scores, varied significantly among different tools and user experience levels. For novice users of Textual, the median score was approximately 34, with an IQR spanning from around 33 to 39. The range of task load experiences varied broadly, as indicated by the whiskers extending from about 25 to 43, with one outlier at 50. The mean score is higher than the median, suggesting a skew towards higher scores. Experienced users of Textual rated their task load higher, with a median score of about 37 and an IQR ranging from roughly 36 to 43. The whiskers, spanning from approximately 36 to 47, indicated a wider range of task load perceptions among experienced users. The mean score was slightly below the median, suggesting a minor skew towards lower scores.

Novice users of VS Code had a median score of around 45, with the IQR extending from about 40 to 55. The whiskers ranged broadly from 30 to 65, reflecting significant variability in task load experiences among novices. The mean score, slightly above the median, indicated a slight skew towards higher ratings. For experienced users, the median score was lower at around 40, with an IQR from approximately 30 to 45. The whiskers extended from about 20 to 60, showing a wide range of task load perceptions. The mean, slightly below the median, indicated a minor skew towards lower scores.

For Visual, novice users had a median score of around 35, with the IQR spanning from about 30 to 40. The whiskers extended from roughly 15 to 45, indicating a wide variability but generally moderate task load perceptions. The mean score was close to the median, suggesting a balanced distribution. However, experienced users showed a lower median score of about 32.5, with the IQR ranging from approximately 27.5 to 37.5. The whiskers extended from about 25 to 48, indicating more variability but a generally lower perception of task load compared to novices. The mean, above the median, suggested a skew towards higher scores.

For Buddy, novice users had a low median score of about 27.5, with an IQR of roughly 22.5 to 27.5. The whiskers extended from about 15 to 27.5, indicating a narrow range of task load perceptions. The mean score is the same as the median. Experienced users showed a higher median score of around 35, with the IQR ranging from approximately 27.5 to 42.5. The whiskers extended from 20 to 47.5, indicating more consistent task load perceptions compared to novices. The mean score, slightly below the median, suggested a minor skew towards lower scores.

Comparing Textual with VS Code, Textual displayed a more concentrated distribution of workload scores for experienced users, while VS Code showed a more significant difference between novices and experienced users. Novices using VS Code reported much higher workload scores with greater variability compared to those using Textual. Experienced users of VS Code had lower, more consistent scores, suggesting they found the tool less taxing than novices did.

Regarding Visual and Buddy comparison, experienced users reported higher workload scores than novices for both Visual and Buddy tools, but with varying levels of consistency. Visual showed similar median workload scores between the two groups, though experienced users exhibited more variability. In contrast, Buddy's experienced users reported a broader range of workload scores, indicating more varied experiences, while novices reported lower and more consistent workload scores.

Table 5.12: NASA-TLX Scores Comparison by Experience Level

Tool	**T-Test Statistic**	**P-value**
Textual	-0.077	0.94
VS Code	1.10	0.297
Visual	-0.208	0.840
Buddy	-1.321	0.216

The statistical analyses (Table 5.12) across all interfaces indicate that there are observable differences in NASA-TLX scores between novice and experienced users, but these differences are not statistically significant. This aligns with the findings from the SUS scores, reinforcing that experience level does not substantially affect users' perceptions of usability and workload for the tools analyzed.

Figure 5.21 compares the SUS/TLX ratios between novice and experienced users for each tool, highlighting the relationship between usability and workload across different experience levels.

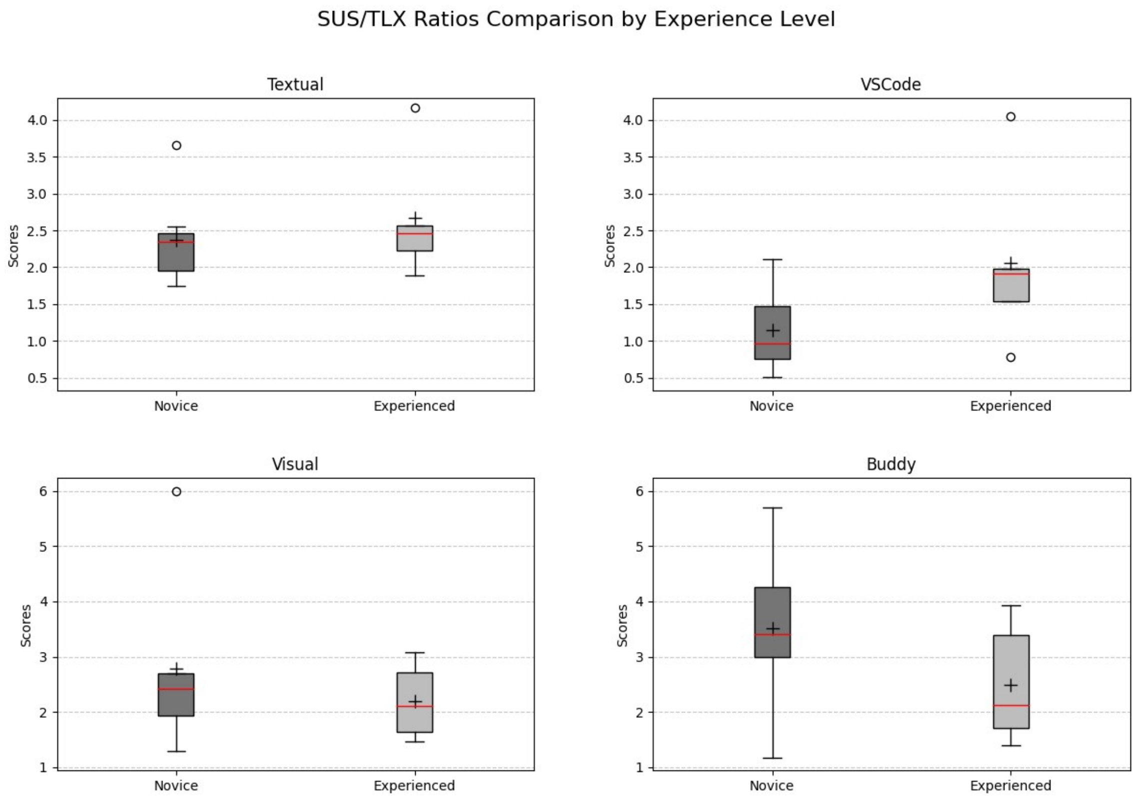


Figure 5.21: SUS-TLX Ratios Comparison by Experience Level

Novice users demonstrated a higher preference for Textual and Buddy, as indicated by their respective median ratios of around 2.5 and 3.5. Textual had an interquartile range spanning from 2 to 2.5 with one outlier observed at 3.6, while Buddy had an interquartile range from 3 to 4.2. In contrast, VS Code presented a lower median ratio of 1, accompanied by an IQR extending from 0.5 to 2. Visual exhibited a median ratio of 2.5 and an IQR between 2 and 2.8, with an outlier at 5.

When examining experienced users, Textual displayed a median of 2.5 and an IQR spanning from 2.25 to 2.5, with an outlier, this time at 4.25. VS Code increased the median ratio to 2, with an interquartile range of 1.5 to 2, and with one outlier at 0.75 and another at 4. Visual presented a similar median ratio of 2 and an IQR ranging from 1.5 to almost 3. Buddy also exhibited a median ratio of 2, accompanied by an IQR ranging from 1.8 and 3.5. Regarding experience plots, both textual modeling approaches have a narrow interquartile range compared to novices.

When comparing Textual with VS Code, both user categories show higher and similar ratio scores with less variability in Textual than in VS Code. Both Textual and VS Code reported outliers, with more significant ones among experienced users. For VS Code, experienced users are more comfortable with the tool than novice ones, as shown in the plot. Regarding Visual and Buddy, both tools exhibit higher scores than Textual modeling approaches. Visual's experienced users report a workload score with a similar median to that of novices but with greater variability. In contrast, Buddy's experienced users report a broader range of scores, indicating varied experiences. Novices, however, report higher and more consistent SUS/TLX ratios for Buddy, suggesting it is easier and more efficient for those with less experience compared to Visual, which is steadier but not notably efficient for either group.

Table 5.13: SUS-TLX Ratios Comparison by Experience Level

Tool	Statistical Test	Statistic	P-value
Textual	T-Test	-0.67	0.52
VS Code	T-Test	-1.75	0.11
Visual	Mann-Whitney U Test	19.0	0.937
Buddy	T-Test	1.334	0.212

The statistical tests in Table 5.13 confirm no significant differences in the SUS/TLX ratios between novice and experienced users across all four tools. However, the boxplots may visually suggest slight differences, particularly in Buddy and VS Code.

NASA-TLX Dimensions

The following scatter plots explore how workload dimensions impacted perceived usability for all tools studied and quantified the strength and direction of these associations.

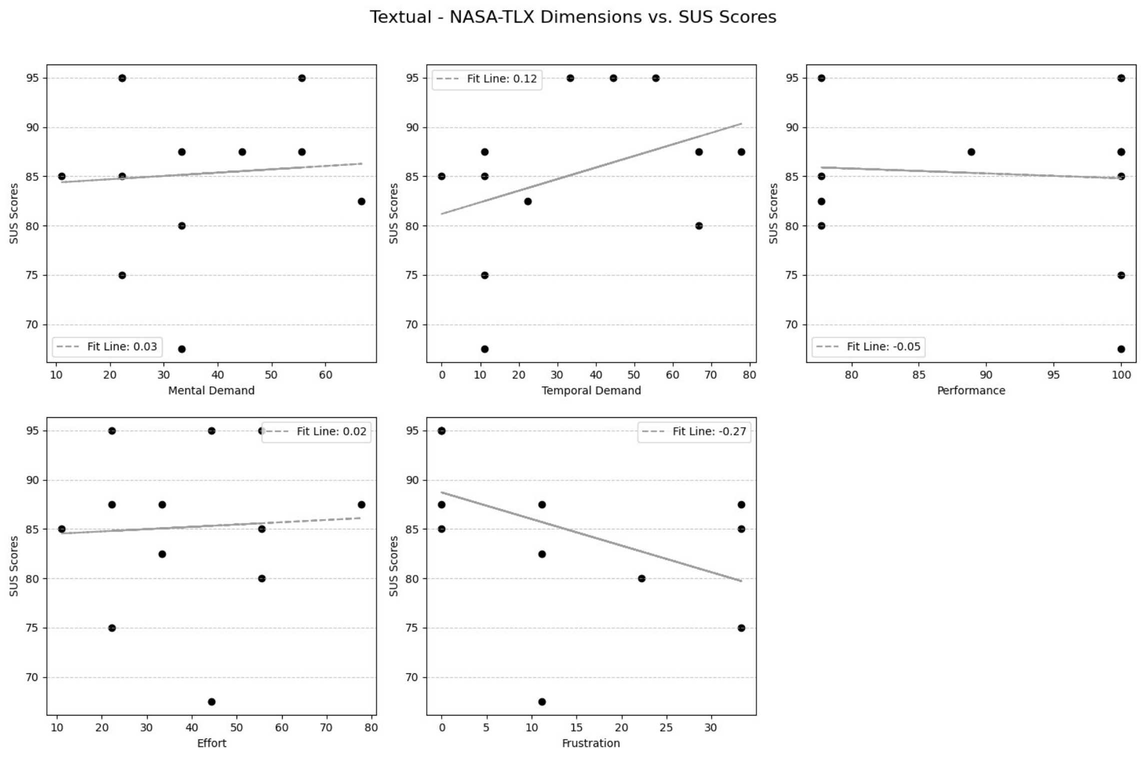


Figure 5.22: Textual - NASA-TLX Dimensions vs. SUS Scores

For the Textual tool, temporal Demand exhibits a weak positive correlation with SUS scores (r=0.12), indicating that as temporal demand increases, perceived usability tends to improve, although the effect is modest. Frustration presents a moderate negative correlation (r=-0.27). Performance shows a very weak negative correlation with SUS scores (r=-0.05). Mental Demand and Effort have almost negligible correlations with SUS scores (r=0.03 and r=0.02, respectively), indicating little to no linear relationship between these dimensions and perceived usability. Table 5.14 presents the statistical results:

Table 5.14: Textual - NASA-TLX Dimensions vs. SUS Scores

Dimension	**Statistical Test**	**Statistic**	**P-value**	**Pearson**	**Spearman**
Mental Demand	T-Test	-9.173	15.66e-09	0.07	X
Temporal Demand	Mann-Whitney U Test	2.0	5.57e-05	X	0.40
Performance	Mann-Whitney U Test	101	0.094	X	0.08
Effort	Mann-Whitney U Test	2.0	5.595e-05	X	0.06
Frustration	Mann-Whitney U Test	0	3.2e-05	X	-0.59

The analysis shows that among the NASA-TLX dimensions, Temporal Demand and Frustration have the most significant impact on perceived usability for the Textual tool, as indicated by their moderate correlations with SUS scores. However, despite being statistically significant, Mental Demand and Effort show weak correlations with SUS scores and thus do not meaningfully influence perceived usability. The Performance dimension shows a very weak and statistically insignificant correlation, indicating minimal impact on usability perception.

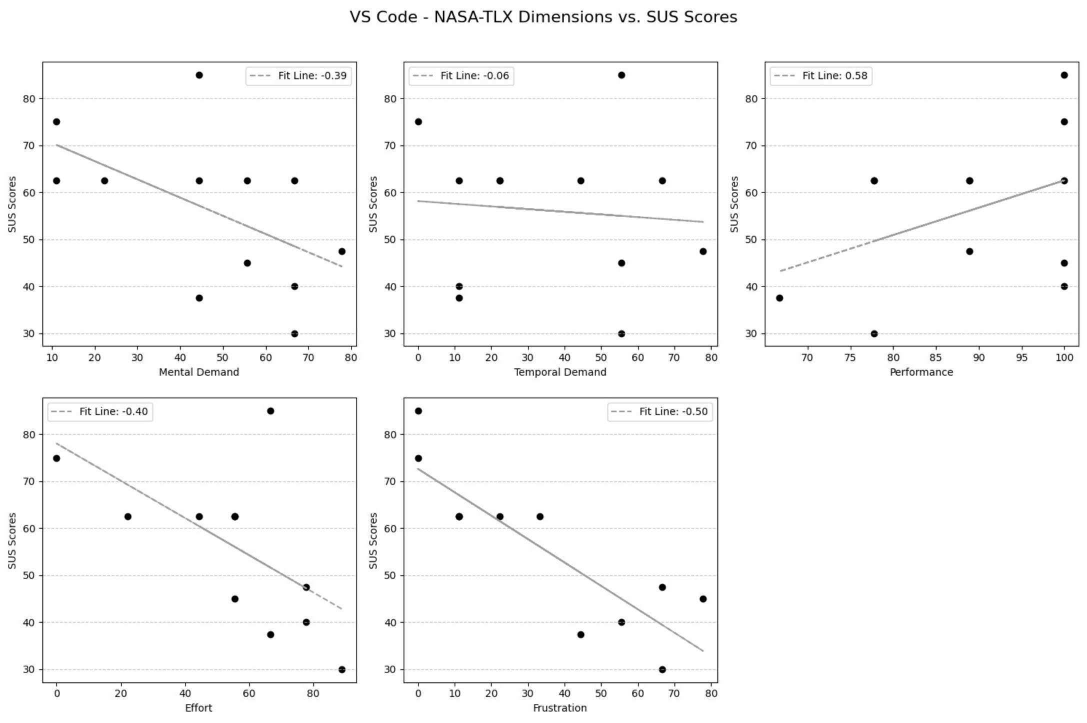


Figure 5.23: VS Code - NASA-TLX Dimensions vs. SUS Scores

Regarding VS Code, Frustration, Effort and Mental Demand demonstrate a strong negative correlation with SUS scores (r=-0.50, r=-0.40 and r=-0.39), indicating that as these factors increase, perceived usability decreases significantly. Performance exhibits a moderate positive correlation with SUS scores (r=0.58). Temporal Demand has a weak negative correlation with SUS scores (r=-0.06), with a minimal impact of temporal demand on perceived usability.

Table 5.15: VS Code - NASA-TLX Dimensions vs. SUS Scores

Dimension	**Statistical Test**	**Statistic**	**P-value**	**Pearson**	**Spearman**
Mental Demand	T-Test	-1.109	0.279	-0.53	X
Temporal Demand	Mann-Whitney U Test	39	0.0589	X	-0.09
Performance	Mann-Whitney U Test	139	0.0001	X	0.40
Effort	T-Test	-0.0569	0.955	-0.60	X
Frustration	Mann-Whitney U Test	40	0.279	X	0.067

The analysis from Table 5.15 reveals that Performance is the only dimension with a statistically significant and meaningful impact on perceived usability, where better performance directly correlates with higher SUS scores. Mental Demand and Effort show moderate negative correlations with usability, meaning that increased cognitive load and effort can make the tool-less user-friendly, but these effects are not statistically significant, meaning they could be due to random variation. Frustration and Temporal Demand exhibit very weak correlations, indicating that these dimensions do not have a substantial impact on usability in the context of this study.

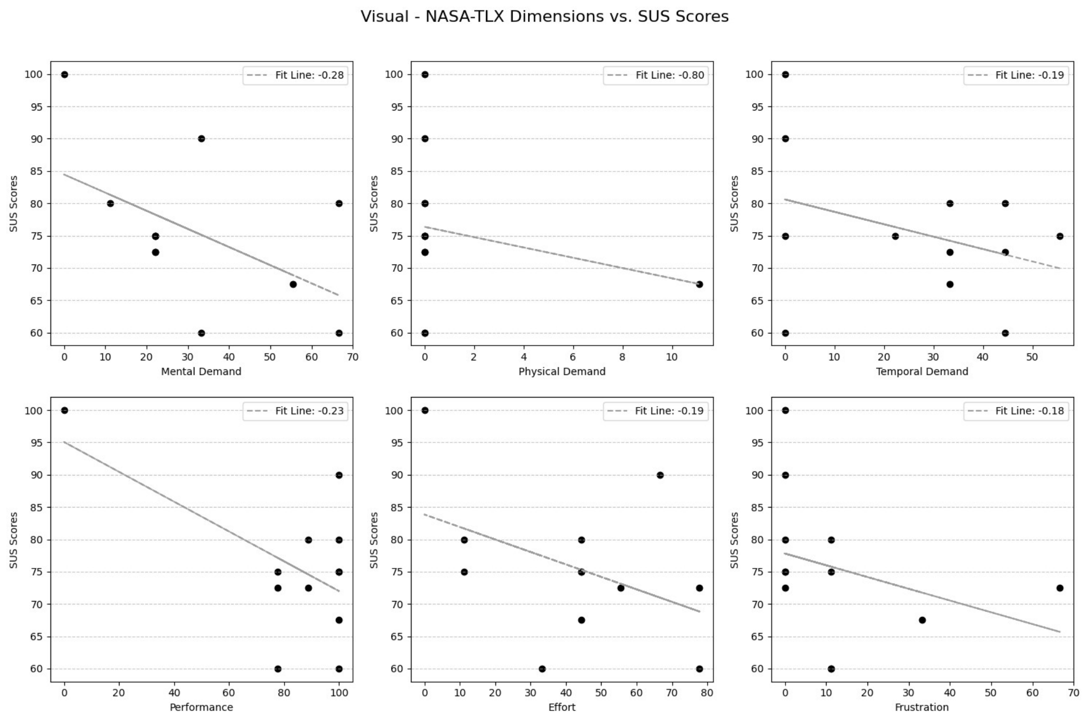


Figure 5.24: Visual - NASA-TLX Dimensions vs. SUS Scores

For the Visual tool, the analysis reveals notable trends in how NASA-TLX dimensions correlate with perceived usability, as measured by SUS scores. Mental Demand and Performance show a moderate negative correlation (r=-0.28 and r=-0.23), indicating that higher cognitive effort and better performance outcomes are linked with lower usability perceptions. Physical Demand also negatively correlates with SUS scores (r=-0.80), representing a slight decline in usability as physical effort increases. This relationship is supported by a significant difference in usability scores across varying levels of physical demand. The remaining dimensions display a similar pattern with a weak negative correlation (r=-0.19 for Effort and Temporal Demand, and r=-0.18 for Frustration).

Table 5.16: Visual - NASA-TLX Dimensions vs. SUS Scores

Dimension	Statistical Test	Statistic	P-value	Pearson	Spearman
Mental Demand	T-Test	-6.384	2.009e-06	-0.52	X
Physical Demand	Mann-Whitney U Test	0	1.373e-05	X	-0.31
Temporal Demand	Mann-Whitney U Test	0	3.315e-05	X	-0.26
Performance	Mann-Whitney U Test	113.0	0.0176	X	-0.06
Effort	T-Test	-4.108	0.00046	-0.43	X
Frustration	Mann-Whitney U Test	2.0	4.934e-05	X	-0.58

This analysis from Table 5.16 reveals that the most significant dimensions influencing perceived usability are Mental Demand, Effort, and Frustration. These dimensions have moderate to strong negative correlations with SUS scores, as supported by highly significant statistical tests. This indicates that reducing mental effort, physical demand, and frustration would increase users' perceived usability of the Visual tool.

Physical Demand and Temporal Demand also exhibit significant correlations with SUS scores, albeit with slightly weaker impacts than other dimensions and thus, their effects on usability may be subtle or inconsistent.

Interestingly, Performance has a statistically significant correlation, but the low correlation affirms that it may not be as crucial to perceived usability as other dimensions. This could imply that, while performance has a role in usability perception, it is overshadowed by the negative impacts of effort and frustration, making these areas more critical for usability improvements in the Visual tool.

Regarding Buddy (Figure 5.25), Mental Demand shows a strong negative correlation with SUS scores (r = -0.62), alongside Frustration (r = -0.53). This indicates that higher mental demands and increased frustration are associated with lower usability perceptions. Physical Demand has a negative correlation (r = -1.45), suggesting that increased physical demand slightly reduces perceived usability. Temporal Demand exhibits a moderately negative correlation (r = -0.45), indicating that time pressure negatively affects usability, though not as strongly as mental demand. Performance displays an almost neutral correlation with SUS scores (r = -0.08), implying that performance success is not strongly linked to usability perceptions. Effort is negatively correlated with SUS scores (r = -0.31), signifying that higher effort leads to lower usability scores.

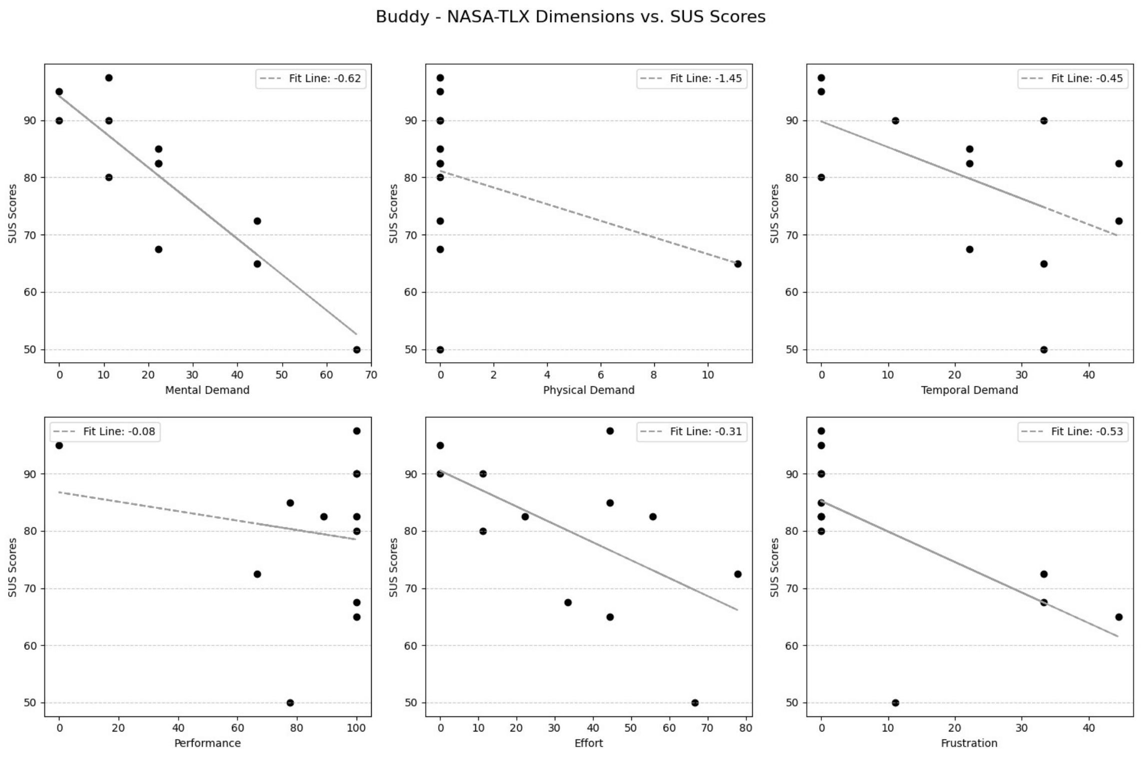


Figure 5.25: Buddy - NASA-TLX Dimensions vs. SUS Scores

Table 5.17: Buddy - NASA-TLX Dimensions vs. SUS Scores

Dimension	Statistical Test	Statistic	P-value	Pearson	Spearman
Mental Demand	T-Test	-8.12	4.56	-0.89	X
Physical Demand	Mann-Whitney U Test	0	1.41	X	-0.39
Temporal Demand	T-Test	-9.29	4.52	-0.53	X
Performance	Mann-Whitney U Test	102.0	0.08	X	0.03
Effort	Mann-Whitney U Test	7.0	0.0002	X	-0.53
Frustration	Mann-Whitney U Test	0	2.57	X	-0.79

The statistical tests from Table 5.17 show that Mental Demand, Effort, and Frustration strongly impact the usability of the Buddy system. These aspects have strong negative correlations with SUS scores, indicating their high statistical significance.

Physical Demand and Temporal Demand also play a role in usability perceptions, but their impact is not as strong as the factors mentioned above. These aspects still have significant negative correlations, so they are important to consider for improving usability.

On the other hand, Performance has minimal influence on usability, with a very weak and non-significant correlation with SUS scores, meaning that performance may not significantly affect users' perception of the Buddy tool's usability.

Task Completion Time on Usability

Figure 5.26 shows the correlation between SUS scores and task completion time for each tool, indicating how usability ratings relate to the time taken to complete tasks.

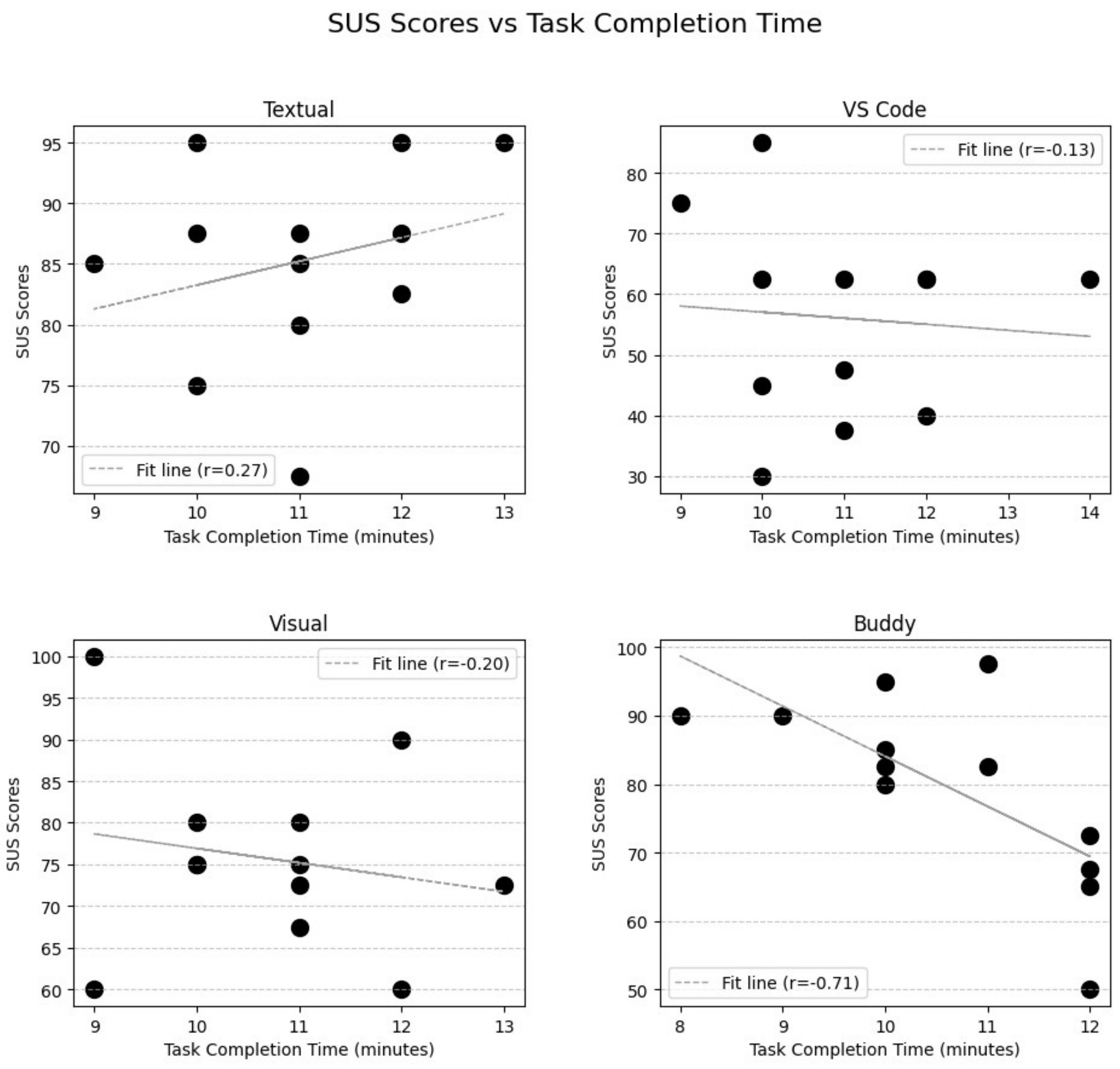


Figure 5.26: SUS Scores vs Task Completion Time

The Textual plot reveals a slight positive correlation between SUS scores and task completion time, with a Spearman correlation coefficient of 0.27. Most data points are clustered around 75 to 90 in SUS scores and between 9.5 to 12 minutes in task completion time. The fit line indicates that as task completion time increases, SUS scores tend to increase slightly, although this effect is minimal.

In contrast, the VS Code plot demonstrates a weak negative correlation, with a Spearman correlation coefficient of -0.13. The data points are widely scattered, with SUS scores ranging from 30 to 80 and task completion times varying from around 9 to 13 minutes. The trend line points that as task completion time increases, SUS scores slightly decrease.

For Visual, the plot demonstrates a weak negative correlation, with a Spearman correlation coefficient of -0.20. The majority of data points are clustered around 9.5 to 12.5 minutes for task completion time, with SUS scores mostly between 65 and 95. The fit line indicates a slight decrease in SUS scores as task completion time increases, meaning that longer task durations might negatively affect perceived usability, although this relationship is not strong.

The Buddy plot shows the most substantial negative correlation among the four tools, with a Spearman correlation coefficient of -0.71. The data points are more widely spread, with task completion times ranging from 8 to 12 minutes and SUS scores varying between 50 and 100. The fit line clearly shows a significant decrease in SUS scores as task completion time increases, indicating that users perceive the tool as less usable when tasks take longer to complete.

Table 5.18: SUS Scores vs Task Completion Time

Tool	**Statistical Test**	**Statistic**	**P-value**	**Spearman**
Textual	Mann-Whitney U Test	0	3.289e-05	0.27
VS Code	Mann-Whitney U Test	0	3.147e-05	-0.13
Visual	Mann-Whitney U Test	0	3.34e-05	-0.20
Buddy	Mann-Whitney U Test	0	3.34e-05	-0.71

The results indicate that Buddy exhibits the strongest negative correlation, confirmed by both the strength of the correlation and its statistical significance. This shows that users perceive the tool as less usable when it takes longer to complete tasks. In contrast, Textual and VS Code show weak correlations, indicating that while task completion time does have a statistically significant effect on usability, the impact is relatively minor. Visual also shows a weak negative correlation, suggesting a minor but statistically significant effect on usability based on task duration.

5.3.3 Qualitative Data

The feedback collected from open-ended questionnaire questions and linear scale questions is categorized into textual and visual approaches. This feedback helps us understand the features that made script configuration easier and provides insights for our platform improvements.

Textual Modeling

The content assist was one of the most mentioned features that accelerated the configuration process and reduced errors, with 11 out of 12 participants classifying it with 5 out of 5 when questioned about the following questions:

- Did the content assist feature cover all the necessary elements and options you needed?
- How much did the content assist feature improve your overall usability of the DSL?

Its implementation was notable for "providing guidance on available attributes in each workflow component", "being case insensitive", and the "ability to be used mid-word". This improved the user experience and sped up the configuration process.

Another notable feature was quickfix, which "efficiently handled difficult tasks" and "offered clear and effective suggestions for the necessary solutions". This feature was classified with 5 out of 5 by 66.7% of participants when questioned about the following questions:

- How useful were the quickfix suggestions in correcting the detected errors?
- How much did the quickfix feature improve your efficiency in correcting errors?

In addition, the elimination of redundant punctuation, such as simple indentation, no dashes, and colons, was emphasized. Furthermore, using an easy-to-understand syntax, including a distinction between components and their attributes with capital letters, enhanced workflow definition simplicity. This not only simplifies the process but also reduces errors.

Improvements

- **Quickfix:** Some participants suggested changing the naming convention when using quickfix to duplicate component names, to make the new name more explicit or to indicate what it will be renamed to. Another suggestion was to display the names of defined but undeclared components and allow them to be changed. However, this solution would only work for specific cases.
- **Content Assist:** The mandatory fields of a component could be implicitly defined by defining the selected component with the content assist functionality with the respective fields. The mandatory fields of a component could be implicitly defined by using the content assist to select the component along with its respective fields. This suggestion is relevant because it removes the responsibility from the user to know/not forget all the mandatory attributes, thereby reducing the likelihood of configuration errors.

  It was suggested that when defining components, the content assist should open directly after pressing the tab instead of having to press the tab and then open the content assist. It was also suggested that a space should be automatically provided when selecting the content assist option. These two suggestions were the participant's preferences, although it was noted that they are not the most relevant points to highlight.

  One participant suggested that it would be interesting to have a feature similar to GitHub Copilot that could suggest a relevant keyword when you're writing. This idea would be to integrate the existing content assist, which is currently triggered by pressing Ctrl+Space, with an AI engine that tries to predict what I'm writing.

Visual Modeling

As observed, the visual approach is more visually complex than the textual one and therefore requires more time to learn the platform than the study provides. Based on this statement, 58.3% of participants rated the ease of navigation and interaction of the approach at 4 out of 5 when asked: How easy was it to interact with and navigate the visual diagram in Sirius Eclipse?

Some feedback worth mentioning is that the diagram visualization significantly enhanced the participant's understanding of the workflow and its components, with 25% of the participants classifying it with 5 out of 5 and 50% with 4 out of 5, when questioned about: How would you rate the visual representation of the CI/CD pipeline in terms of clarity and comprehensibility?. This statement was justified because the "diagram was organized in boxes inside other boxes and components as icons to establish some belonging".

The structured palette from Sirius Eclipse was also another mention that simplified workflow management by offering a clear and organized layout for all components, and making it easier to comprehend the workflow structure and attribute placement, while the validation feature effectively describes specific errors, contributing to a smoother configuration process.

Improvements

- **Error's detection feedback:** Participants' most mentioned suggestion was the difficulty in locating the errors identified by the custom validations since error feedback is only given when the user clicks on the component in question.
- **Diagram:** The diagram should have a more structured layout, with a clear order or hierarchy for its components.
- **Properties View:** For component attributes that require referencing component names defined in the workflow, instead of the user typing in those names, provide a dropdown list of existing component names relevant to the attribute. This would clarify the options available to the user and reduce configuration errors.

5.4 Results Discussion

This section summarizes the key findings from the quantitative data analysis, focusing on the SUS and NASA-TLX score's implications for user experience and task performance, alongside task completion times.

5.4.1 Interpretation of Findings

The usability and workload scores from Tables 5.2 and 5.3 are crucial starting points for our discussion.

Usability

Our solution yielded positive usability results, with Textual's usability score of 85.2 and Visual's score of 75.6 surpassing the average SUS usability score of 68. However, VS Code scored 56.04, significantly lower than the threshold, highlighting the challenges users face with this tool. Despite this, Buddy, with a score of 79.8, remains the preferred visual pipeline modeling tool, while our Visual tool is not far behind.

The findings from Figure 5.14 further reinforce these observations. Textual and Buddy demonstrate high median scores and narrow IQRs, indicating consistent user satisfaction. On the other hand, the Visual tool shows a broader score distribution and presence of outliers, pointing to mixed user experiences, likely due to varying familiarity with visual modeling interfaces. VS Code's low median score and wide score range highlight significant usability challenges, likely worsened by participants' varying experience levels with CI/CD tools and the complexities of CircleCI syntax. Figures VIII.1 and VIII.2 highlight the usability aspects impacting the presented results, where most discrepancies are presented in the textual modeling approach.

Participants who tested the textual approach expressed a clear preference for Textual over VS Code, finding it easier to use and learn. This preference is likely due to the implemented features and syntax guidance, contributing to its ease of configuration.

In contrast, Visual and Buddy exhibit similar high usability scores, and participants stated a preference for visual modeling noted by 14 out of 24 participants, as it allowed them to see different workflow states and their components. This statement was somewhat due to the initial proposal in which more experienced users preferred a textual approach due to the simplicity of the visual interface and faster configuration by only stating the syntax needed to configure the scripts. Nonetheless, our visual approach still holds significant value, particularly for users who prefer a more graphical interface for complex workflows and want an observable state of the workflow.

Buddy's effectiveness can be attributed to its ability to simplify complex processes into visual components, making it easier for a wider range of users to manage, including those with less technical knowledge. It is worth noting that an initial assumption was that the developed tool would be less appealing, given that Eclipse and Sirius environments do not prioritize this visual aspect.

The correlation between task completion time and SUS scores, as shown in Figure 5.26, supports these observations. The Textual approach stands out for its quick task completion speed. As participants use the platform more, it leads to increased usability due to the help of the implemented features. In contrast, VS Code shows a slightly negative correlation due to the lack of assistance. Users are required to understand the CircleCI syntax to correctly configure the scripts, which can generate frustration and lead to lower perceived usability over time. Overall, visual modeling has a stronger negative correlation between SUS scores and task completion times, whereas the Visual tool shows slightly longer task completion times on average. This may stem from the inherently exploratory nature of visual interfaces, where users interact with graphical elements to construct or modify workflows. While this approach provides an intuitive and accessible entry point, the navigation and manipulation of visual elements can introduce additional steps that extend the time required to complete tasks. However, this does not necessarily detract from the usability of the Visual approach; rather, it highlights the trade-off between ease of use and the time investment required for task completion.

Workload

All platforms exhibit positive workload perceptions, with the visual modeling approach tasks demanding the least effort, as indicated by scores of 34.5 for Visual and 30.6 for Buddy. Textual tool again stood out from the textual modeling approaches with a lower workload score of 36.14, reinforcing that experienced users find this method less cognitively taxing for identical task efforts. The notably higher workload associated with VS Code (43.52) is corroborated by its consistently higher scores across most NASA-TLX components, as Figure 5.16 indicates that VS Code's higher workload is driven by increased mental demands, effort, and frustration. Participants' perception of complexity may be influenced by unfamiliarity with the tool and CircleCI syntax, and a lack of configuration guides/help, as evidenced by their feedback. This may be reflected in the results of the dimensions with the highest discrepancy.

Contrastly, Visual and Buddy tools fall between these extremes. Visual shows more variability in workload perception, as it is more mentally demanding and requires more effort, while Buddy's workload perception remains relatively stable. These lower scores may reflect the ease with which users can understand and interact with CI/CD pipelines in a visual format, which reduces mental effort compared to more text-heavy approaches. The slightly higher Visual workload score compared to Buddy's may be attributed to Visual's interface being more demanding in visual information, such as navigating through the diagram or understanding the relationships between different components in the interface. While it simplifies complex workflows, translating them into visual elements might introduce additional cognitive steps, such as interpreting icons or adjusting the interface layout.

SUS/NASA-TLX Impact

The relationship between usability and perceived workload, examined in Figure 5.18, reveals an inverse correlation, where higher usability aligns with a lower workload. This is not a definitive statement for every case, but rather a study-specific affirmation that this study demonstrated an inverse correlation since there are cases where high usability does not imply low workload scores.

Textual reveals as the outlier tool, presenting a weak positive correlation, suggesting that as users find it easier to use, they perceive slightly more workload. This positive correlation reflects the trade-off for experienced users between high usability and the cognitive load involved in efficiently using the tool. While users find Textual easier and faster, their mental demands (as measured by NASA-TLX) remain high due to the nature of working with textual models, especially when performing complex tasks.

VS Code contradicts this trend with a moderate negative correlation, where low usability (SUS = 56.04) corresponds to a higher perceived workload. Configuring difficulties with the scripts in this tool can also be explained with data points concentration being more on the low usability side of the chart, alongside five lower usability scores and a very high workload.

Visual also exhibits a moderate negative correlation with a broader range of scores, indicating mixed user satisfaction, while Buddy's trend is less distinct, indicating less consistency in the relationship between usability and workload.

These observations suggest that for Textual and VS Code, usability improvements have a more pronounced effect on reducing workload, whereas for Visual and Buddy, the relationship is more complex. This complexity may be due to specific features or tasks within these tools affecting perceived workload differently.

NASA-TLX Dimensions Impact

The results of the correlation analysis between NASA-TLX dimensions and SUS scores provide further insights, revealing that Mental demand, Effort, and Frustration are the most influential dimensions across all platforms, with strong negative correlations indicating that higher levels in these dimensions significantly reduce usability scores. In contrast, temporal demand shows a weak correlation, suggesting it has a lesser, but still notable, impact on usability perceptions. Physical demand, as expected, has no impact on tool usability due to the nature of the tasks performed by the participants. Performance generally has a weak correlation with usability scores, indicating that users' perceptions of how well they perform tasks with the tool do not strongly influence their overall usability ratings. The results depicted in Figures 5.22, 5.23, 5.24, and 5.25 generally align with the initially formulated hypotheses, with these dimensions having a more significant impact on the perceived tools usability.

Participants Experience Level Impact

Upon conducting a thorough analysis of usability and workload scores for users with different experience levels, the data presented in Figures 5.19, 5.20, and 5.21 reveals interesting patterns. Notably, Textual and Buddy consistently exhibit high median SUS scores across both novice and experienced users, with experienced users of Textual reporting slightly higher median scores. This suggests that familiarity contributes to its perceived usability. On the other hand, VS Code demonstrates a significant difference between novice and experienced users, with novices rating its usability much lower, indicating a steep learning curve.

When examining NASA-TLX scores, Textual once again stands out, with experienced users reporting lower workload scores, affirming earlier findings that increased usability corresponds to reduced perceived effort. The results for Visual and Buddy are less conclusive, implying that while these tools are generally user-friendly, certain tasks or features may pose challenges for less experienced users.

For VS Code, the data emphasizes its complexity, particularly for novice users, who report low usability and high workload. Even among experienced users, the workload remains substantial, indicating that the tool is mentally demanding irrespective of skill level. This reinforces the perception of VS Code as a tool with a steep learning curve and significant usability challenges.

Visual and Buddy show a different pattern. Novice users find Buddy highly usable with low cognitive demand, suggesting that its visual interface is effective in simplifying complex processes. However, experienced users exhibit more variability in their workload scores for Buddy, potentially indicating that its simplicity may not fully meet the needs of advanced users. Visual, on the other hand, presents a more variable experience for both novice and experienced users, with moderate workload scores across the board. This variability likely stems from the exploratory nature of visual interfaces, which can introduce cognitive challenges regardless of experience level.

Demographic Impact

Interestingly, the study found that user preferences were not significantly associated with demographic factors, suggesting that the choice between textual and visual approaches is more closely related to individual work habits and personal comfort rather than specific background characteristics. This finding underscores the importance of offering both approaches in CI/CD tools, allowing users to choose the method that best suits their workflow and cognitive style.

5.4.2 Inferences

The evaluation of the dual modeling approach and the perceived value of features and implementations is essential to ascertain the effectiveness and usability of the proposed solution. The observations from the usability and workload scores align with the hypotheses formulated and shed light on the tangible impact of the dual modeling approach on user experience and perceived value.

The usability scores reveal that both Textual and Visual demonstrate higher than average SUS usability scores, with Textual scoring 85 and Visual scoring 75, compared to the average SUS usability score of 68. This suggests that our solution, particularly the textual approach, offers above-average usability, supporting the hypothesis $H_{\mathbf{1DualModeling}}$ - that the dual modeling approach, comprising both textual and graphical representations, is effective and usable in configuring and managing CI/CD pipelines.

The results support $H_{\mathbf{1FeaturesValue}}$, indicating that the features and implementations add significant value. Participants' feedback highlighted a preference for Textual over VS Code, finding it easier to use and learn due to the implemented features and syntax, which facilitated script configuration. This preference is evident from the high usability scores and positive feedback regarding the features' utility. In contrast, Visual and Buddy tools also show high usability scores, with a notable preference for visual modeling by 12 out of 20 participants. These findings confirm that the features and implementations in both textual and graphical approaches significantly enhance user experience and script correctness, thereby supporting the hypothesis.

The data provide mixed support for $H_{\mathbf{1ComparisonCircleCI}}$. While Textual and Buddy tools received high usability and low workload scores, indicating effective script configuration and management, the variability in Visual's workload perceptions suggests areas needing improvement. Textual's clear negative correlation between high usability and low workload suggests it facilitates easier script configuration than VS Code. Visual and Buddy tools also show favorable usability scores but with a more complex relationship between usability and workload. This complexity could be attributed to specific features or tasks within these tools affecting user perceptions differently. While the dual modeling approach appears generally effective, certain tools and features need refinement to fully surpass CircleCI's approach in usability and efficiency, suggesting partial support for the hypothesis.

The alignment between our hypothesis-driven evaluation and the functional and non-functional requirements demonstrates that the dual modeling approach, along with its specific features, not only fulfills the expected criteria but also provides tangible benefits in terms of usability and efficiency, as evidenced by the high SUS scores and positive user feedback (**FR-1, FR-2, NFR-1, NFR-2, NFR-3**).

The Textual modeling tool, with a strong usability score of 85.2, clearly demonstrates that the interface we developed for configuring CI/CD scripts not only fulfills the need for a user-friendly textual approach but also exceeds expectations. Users found it intuitive and efficient, confirming that our design choices hit the mark. Although with less strength, the same can be said about the Visual modeling tool's intuitive and user-friendly interface, scoring 75.6. It's worth noting that the tool's capacity to streamline complex workflows was acknowledged and valued, demonstrating that we effectively met the need for visual modeling.

Tool agnosticism was another critical requirement (**FR-3**), and our solution proved its adaptability by maintaining compatibility with various CI/CD platforms. This was reinforced by the automated code generation feature, which seamlessly generated interoperable code across different tools, reducing the need for manual configurations and minimizing errors (**FR-4**).

Furthermore, our approach to model transformations enabled users to switch between different model views, facilitating a more comprehensive understanding of CI/CD pipelines. This flexibility in navigating between textual and visual models confirms that we effectively addressed the need for smooth model transitions.

Our error handling and reporting features, particularly within the Textual tool, played a significant role in reducing user frustration during configuration with the interface's guidance to understand and correct mistakes at any configuration step (**FR-6**).

Scalability was another key consideration, and our solution demonstrated its ability to handle complex CI/CD pipelines effectively (**NFR-4**). While the Visual tool showed slightly longer task completion times, these were manageable and indicative of the tool's ability to support intricate workflows without compromising overall efficiency.

Although the participants did not test it, the requirements of model transformations (FR-5) and maintaining clean code (NFR-5) were met in its implementation and testing iteration.

In conclusion, the dual modeling approach, comprising Textual and Visual interfaces, presents a solid solution for configuring CI/CD pipelines, effectively catering to users with varying preferences and expertise levels. The high usability scores and the inverse relationship between usability and workload underscore its effectiveness and user-centric design. However, to further enhance the usability and address the identified challenges, targeted improvements in both interfaces are necessary. By refining the visual elements, streamlining the textual syntax, and providing more robust user support, our approach can achieve even higher levels of user satisfaction and effectiveness, solidifying its position as a versatile and efficient tool for CI/CD pipeline configuration.

5.4.3 Lessons Learned

During the user testing and usability evaluation process, we gained valuable insights that have significantly contributed to our understanding of user preferences and challenges. One of the key lessons learned is the importance of conducting diverse and representative user testing to capture a wide range of perspectives and experiences. This approach allowed us to identify varying user preferences and expertise levels, which in turn, informed the design of our dual modeling approach.

Furthermore, we learned that continuous user feedback is essential for refining and improving the usability of our interfaces. The feedback we received during iterative pilot tests and the evaluation process highlighted the need for targeted improvements in textual and visual interfaces. It also emphasized the significance of providing robust user support to address the identified challenges effectively.

Additionally, we recognized the value of prioritizing user-centric design, as evidenced by the inverse relationship between usability and workload. This reinforced our commitment to creating interfaces that not only facilitate CI/CD pipeline configuration but also ensure a seamless and satisfying user experience.

5.5 Threats to Validity

When considering the threats to the validity of this study, several potential biases come to light. Participant bias is a significant concern, as the sample consists mainly of Computer Science students from the same university, potentially not representing the broader population of CI/CD pipeline users. Participants' prior experience with CI/CD tools and familiarity with specific platforms could bias their perceptions and performance. Additionally, the potential for learning effects exists, as participants used multiple tools in the study, and although efforts were made to minimize this effect by presenting tools in a randomized order, it cannot be entirely ruled out.

The use of Buddy as a competing platform for the visual approach poses a significant threat to the study. Since Buddy does not configure pipelines for a specific CI/CD platform, its functionalities may limit the fair comparison and evaluation of the visual approach.

Another important concern is the sample size, which consists of 24 participants. A larger sample size would provide stronger and more precise results. However, this sample size is enough to identify potential improvements for both approaches. Nielsen has indicated that 5 users can uncover most issues in an application.

Furthermore, the study's design and methodology, particularly the specific tasks evaluated, may also introduce potential sources of bias or limitations to the findings. Focusing on specific tasks may not fully capture the broader range of activities that users perform in real-world CI/CD pipeline configurations.

6 Conclusion

This chapter provides an overview of the key findings, contributions, limitations, and future work arising from this research.

6.1 Overview

In this thesis, we developed a framework for CI/CD pipelines, using both textual and visual methods. The framework aimed to simplify the complexities of CI/CD solutions and enable their use across various platforms. By integrating textual and visual approaches, we aimed to create a solution suitable for users with different experience levels in CI/CD processes and would aid in migrating between them. It was crucial to implement features that allow users to model configurations in both textual and visual formats, providing flexibility based on their preferences. We utilized M2M transformations with ATL to ensure model coherence and interoperability.

Additionally, we used Acceleo for code generation to automatically create CI/CD pipeline scripts from the models. We also included plugins focused on improving usability through the metamodels definition, grammar, and interfaces. Finally, we conducted a usability study to evaluate the quality and usability of our framework by comparing it with commercial tools.

6.2 Limitations

Despite its strengths, the framework has some limitations that should be acknowledged. Firstly, it is dependent on the Eclipse platform, which may not be preferable or convenient for all users, especially those who are used to other IDEs. This dependency can limit the adoption and use of the framework in various development environments. However, development in Eclipse has enabled exceptional integration of the workflow developed by integrating MDE methodologies with DevOps, something difficult or even unsupported in most of the IDEs available.

The Jenkins and CircleCI solutions, as implemented in this framework, do not support scripted pipelines, which restricts their flexibility and limits the scenarios in which they can be effectively used.

In the visual approach, a problem arises when performing M2M transformations with ATL, where the Eclipse IDE wizard does not recognize the transformed XMI files, requiring users to manually enter their path.

In addition, the framework supports most cron expressions but struggles with complex ones, which can affect the accuracy and completeness of tasks in CI/CD pipelines.

A relevant challenge worth mentioning is the evolution problem inherent to MDE. When a CI/CD platform updates its functionalities, the framework needs to be re-engineered to introduce the new features. This evolution problem presents a two-edged sword. On one hand, it requires effort to adapt the framework to the latest platform updates, posing a maintenance challenge. On the other hand, the use of MDE provides a powerful abstraction that facilitates platform migration and adaptation. This abstraction enables the framework to support multiple CI/CD platforms, simplifies the process of migrating pipeline configurations across different tools, and supports a seamless plugin integration that provides this solution to be made. Thus, while the evolution problem is a limitation, it also underscores the fundamental advantage of MDE in creating a flexible and adaptable system capable of addressing the diverse and changing needs of CI/CD pipeline management.

Furthermore, the study's design and methodology, particularly the specific tasks evaluated, may also introduce potential sources of bias or limitations to the findings. Focusing on specific tasks may not fully capture the broader range of activities that users perform in real-world CI/CD pipeline configurations.

6.3 Future Work

There are other directions to continue this work with future work, drawn from the feedback from the usability study and the analysis of the most important requirements for a robust solution that meets users' needs. Such improvements include enhancing certain features of both approaches and even creating/integrating new functionalities as mentioned in the participants' feedback.

A web-based solution would allow users to access and use the board from any device with an Internet connection, considerably increasing its convenience and usability. Alongside real-time collaboration and peer modeling, the configuration would be more efficient and reduce configuration errors.

One important aspect is the possibility of interoperability between textual and visual modeling. Ensuring that changes made in one format are automatically reflected in the other not only simplifies the user experience by allowing them to start in one approach and continue in another but also avoids inconsistencies in configurations. Even with experienced participants in the field, the vast majority cited that they preferred to have a visualization of the pipeline configuration, even if they wrote it down verbatim.

Expanding the framework to support more CI/CD platforms would also increase its versatility and applicability. By incorporating additional platforms, the framework can cater to a wider range of users and use cases, increasing its relevance and usefulness in the CI/CD ecosystem.

Finally, it would be relevant to redo a usability study with participants with a greater weight of experience and a more diverse background, and with CircleCI's visual editor as the competing tool.

Bibliography

[1] L. Addazi and F. Ciccozzi. "Blended graphical and textual modelling for UML profiles: A proof-of-concept implementation and experiment". In: *Journal of Systems and Software* 175 (2021), p. 110912. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2021.110912. URL: https://www.sciencedirect.com/science/article/pii/S0164121221000091 (cit. on pp. 2, 14).

[2] A. Alnafessah et al. "Quality-Aware DevOps Research: Where Do We Stand?" In: *IEEE Access* 9 (2021), pp. 44476–44489. DOI: 10.1109/ACCESS.2021.3064867 (cit. on p. 23).

[3] *AltexSoft*. https://www.altexsoft.com/blog/cicd-tools-comparison/. Accessed 01/23/2024 (cit. on p. 8).

[4] S. Arachchi and I. Perera. "Continuous Integration and Continuous Delivery Pipeline Automation for Agile Software Project Management". In: *2018 Moratuwa Engineering Research Conference (MERCon)*. 2018, pp. 156–161. DOI: 10.1109/MERCon.2018.8421965 (cit. on pp. 1, 6).

[5] N. Azad and S. Hyrynsalmi. "DevOps critical success factors — A systematic literature review". In: *Information and Software Technology* 157 (2023), p. 107150. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2023.107150. URL: https://www.sciencedirect.com/science/article/pii/S0950584923000046 (cit. on pp. 5, 6, 18).

[6] K. Bahadori and T. Vardanega. "DevOps Meets Dynamic Orchestration". In: *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*. Ed. by J.-M. Bruel, M. Mazzara, and B. Meyer. Cham: Springer International Publishing, 2019, pp. 142–154. ISBN: 978-3-030-06019-0. DOI: https://doi.org/10.1007/978-3-030-06019-0_11 (cit. on pp. 1, 5, 6).

[7] J. Bézivin. "Model Driven Engineering: An Emerging Technical Space". In: (2006). Ed. by R. Lämmel, J. Saraiva, and J. Visser, pp. 36–64. DOI: 10.1007/11877028_2. URL: https://doi.org/10.1007/11877028_2 (cit. on pp. 10, 11).

[8] G. Bou Ghantous and A. Gill. “DevOps: Concepts, practices, tools, benefits and challenges”. In: *PACIS2017* (2017). DOI: https://doi.org/10.1007/978-3-319-49094-6_44 (cit. on pp. 1, 5, 16, 17).

[9] H. Brabra et al. “Model-Driven Orchestration for Cloud Resources”. In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 2019, pp. 422–429. DOI: 10.1109/CLOUD.2019.00074 (cit. on p. 21).

[10] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. 1st. Morgan & Claypool Publishers, 2012. ISBN: 1608458822. DOI: 10.2200/S00441ED1V01Y201208SWE001 (cit. on pp. 8, 10, 11, 13, 16).

[11] J. Brooke. “SUS: A quick and dirty usability scale”. In: *Usability Eval. Ind.* 189 (1995-11). DOI: https://doi.org/10.1201/9781498710411 (cit. on p. 60).

[12] M. Broy, K. Havelund, and R. Kumar. “Towards a Unified View of Modeling and Programming”. In: *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*. Ed. by T. Margaria and B. Steffen. Cham: Springer International Publishing, 2016, pp. 238–257. ISBN: 978-3-319-47169-3. DOI: https://doi.org/10.1007/978-3-030-03418-4_1 (cit. on p. 14).

[13] A. Bucchiarone, A. Cicchetti, and A. Marconi. “Exploiting Multi-level Modelling for Designing and Deploying Gameful Systems”. In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2019, pp. 34–44. DOI: 10.1109/MODELS.2019.00-17 (cit. on pp. 23, 24).

[14] A. Bucchiarone et al. “Grand Challenges in Model-Driven Engineering: An Analysis of the State of the Research”. In: *Softw. Syst. Model.* 19.1 (2020-01), pp. 5–13. ISSN: 1619-1366. DOI: 10.1007/s10270-019-00773-6. URL: https://doi.org/10.1007/s10270-019-00773-6 (cit. on pp. 17, 23).

[15] J. Cabot. “Positioning of the low-code movement within the field of model-driven engineering”. In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS ’20. Virtual Event, Canada: Association for Computing Machinery, 2020. ISBN: 9781450381352. DOI: 10.1145/3417990.3420210. URL: https://doi.org/10.1145/3417990.3420210 (cit. on p. 23).

[16] J. Cabot et al. “Cognifying Model-Driven Software Engineering”. In: *Software Technologies: Applications and Foundations*. Ed. by M. Seidl and S. Zschaler. Cham: Springer International Publishing, 2018, pp. 154–160. ISBN: 978-3-319-74730-9. DOI: https://doi.org/10.1007/978-3-319-74730-9_13 (cit. on pp. 23, 24).

[17] L. Chen. “Continuous delivery: Huge benefits, but challenges too”. In: *IEEE software* 32.2 (2015), pp. 50–54 (cit. on pp. 1, 6).

[18] *CI/CD Pipelines Explained. Benefits and Best Practices*. https://www.opsera.io/blog/all-you-need-to-know-about-ci-cd-pipeline. Accessed 17/10/2023 (cit. on p. 6).

[19] *CI/CD: The what, why, and how*. https://github.com/resources/articles/devops/ci-cd. Accessed 17/10/2023 (cit. on p. 6).

[20] A. Colantoni, L. Berardinelli, and M. Wimmer. "DevOpsML: Towards Modeling DevOps Processes and Platforms". In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS '20. Virtual Event, Canada: Association for Computing Machinery, 2020. ISBN: 9781450381352. DOI: 10.1145/3417990.3420203. URL: https://doi.org/10.1145/3417990.3420203 (cit. on pp. 1, 21).

[21] A. Colantoni et al. "Towards Blended Modeling and Simulation of DevOps Processes: The Keptn Case Study". In: MODELS '22 (2022), pp. 784–792. DOI: 10.1145/3550356.3561597. URL: https://doi.org/10.1145/3550356.3561597 (cit. on p. 21).

[22] I. David et al. "Collaborative Model-Driven Software Engineering — A systematic survey of practices and needs in industry". In: *Journal of Systems and Software* 199 (2023), p. 111626. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2023.111626. URL: https://www.sciencedirect.com/science/article/pii/S0164121223000213 (cit. on p. 17).

[23] A. V. Deursen et al. "Model-Driven Software Evolution: A Research Agenda". In: (2007). URL: https://api.semanticscholar.org/CorpusID:15837208 (cit. on p. 17).

[24] DevBoost. *EMFText*. https://github.com/DevBoost/EMFText. Accessed 01/13/2024 (cit. on p. 15).

[25] P. Duvall, S. M. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2007. ISBN: 0321336380 (cit. on p. 6).

[26] Eclipse. *Acceleo*. https://eclipse.dev/acceleo/. Accessed 01/13/2024 (cit. on pp. 16, 56).

[27] Eclipse. *ATL*. https://eclipse.dev/atl/. Accessed 01/13/2024 (cit. on pp. 16, 54).

[28] Eclipse. *Sirius*. https://eclipse.dev/sirius/. Accessed 01/13/2024 (cit. on p. 15).

[29] M. Fowler. *Continuous Integration*. https://martinfowler.com/articles/continuousIntegration.html. Accessed 01/11/2024 (cit. on p. 6).

[30] *G2 Grid for DevOps Platforms in 2024*. https://www.g2.com/categories/devops-platforms. Accessed 02/07/2024 (cit. on p. 31).

[31] H. da Gião and J. Cunha. "Chronicles of CI/CD: A Deep Dive into its Usage Over Time". In: (2023). DOI: https://doi.org/10.48550/arXiv.2402.17588 (cit. on pp. 1, 31).

[32] H. da Gião, R. Pereira, and J. Cunha. "CI/CD Meets Block-Based Languages". In: *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2023, pp. 232–234. DOI: 10.1109/VL-HCC57772.2023.00039 (cit. on pp. 1, 22).

[33] M. Guerriero et al. "A model-driven DevOps framework for QoS-aware cloud applications". In: *2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE. 2015, pp. 345–351. DOI: 10.1109/SYNASC.2015.60 (cit. on p. 1).

[34] A. Hevner et al. "Design Science in Information Systems Research". In: *Management Information Systems Quarterly* 28 (2004-03), pp. 75–. DOI: https://doi.org/10.1007/978-1-4419-5653-8_2 (cit. on p. 3).

[35] R. P. Hugo da Gião and J. Cunha. "Model-Driven Approaches for DevOps: A Systematic Literature Review". In: (2023) (cit. on p. 2).

[36] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. 1st. Addison-Wesley Professional, 2010. ISBN: 0321601912 (cit. on p. 6).

[37] J. Hutchinson, J. Whittle, and M. Rouncefield. "Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure". In: *Science of Computer Programming* 89 (2014). Special issue on Success Stories in Model Driven Engineering, pp. 144–161. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2013.03.017. URL: https://www.sciencedirect.com/science/article/pii/S0167642313000786 (cit. on p. 17).

[38] J. Hutchinson et al. "Empirical assessment of MDE in industry". In: *2011 33rd International Conference on Software Engineering (ICSE)*. 2011, pp. 471–480. DOI: 10.1145/1985793.1985858 (cit. on p. 17).

[39] IDC. *IDC DevOps Platform Software Tools Market Shares 2022*. Tech. rep. Accessed 15/07/2024. 2022 (cit. on p. 31).

[40] D. Institute. *Upskilling IT 2021 Report*. Tech. rep. Accessed 15/07/2024. 2021 (cit. on p. 5).

[41] R. Jabbari et al. "What is DevOps? A Systematic Mapping Study on Definitions and Practices". In: XP '16 Workshops (2016). DOI: 10.1145/2962695.2962707. URL: https://doi.org/10.1145/2962695.2962707 (cit. on p. 5).

[42] S. P. Jácome-Guerrero, M. Ferreira, and A. Corral. "Software Development Tools in Model-Driven Engineering". In: *2017 5th International Conference in Software Engineering Research and Innovation (CONISOFT)*. 2017, pp. 140–148. DOI: 10.1109/CONISOFT.2017.00024 (cit. on pp. 10, 11, 15, 16).

[43] Jetbrains. *MPS*. https://www.jetbrains.com/mps/. Accessed 01/13/2024 (cit. on p. 15).

[44] R. Jolak et al. "Software engineering whispers: The effect of textual vs. graphical software design descriptions on software design communication". In: *Empirical Softw. Engg.* 25.6 (2020-11), pp. 4427–4471. ISSN: 1382-3256. DOI: 10.1007/s10664-020-09835-6. URL: https://doi.org/10.1007/s10664-020-09835-6 (cit. on p. 14).

[45] M. S. Khan et al. "Critical Challenges to Adopt DevOps Culture in Software Organizations: A Systematic Review". In: *IEEE Access* 10 (2022), pp. 14339–14349. DOI: 10.1109/ACCESS.2022.3145970 (cit. on pp. 1, 2, 17, 27).

[46] G. Kim et al. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution Press, 2014 (cit. on p. 5).

[47] *Knapsack Pro*. https://knapsackpro.com/ci_comparisons. Accessed 01/23/2024 (cit. on p. 8).

[48] V. Kulkarni and S. Reddy. "Model-Driven Development of Enterprise Applications". In: *UML Modeling Languages and Applications*. Ed. by N. Jardim Nunes et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 118–128. ISBN: 978-3-540-31797-5. DOI: https://doi.org/10.1007/978-3-540-31797-5_13https://doi.org/10.1007/978-3-540-31797-5_13 (cit. on p. 17).

[49] *LambdaTest*. https://www.lambdatest.com/blog/best-ci-cd-tools/. Accessed 01/23/2024 (cit. on p. 8).

[50] Q. Liao. "Modelling CI/CD Pipeline Through Agent-Based Simulation". In: *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2020, pp. 155–156. DOI: 10.1109/ISSREW51248.2020.00059 (cit. on p. 1).

[51] W. Liu et al. "Graphical Modeling VS. Textual Modeling: An Experimental Comparison Based on iStar Models". In: *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. 2021, pp. 844–853. DOI: 10.1109/COMPSAC51774.2021.00117 (cit. on p. 14).

[52] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. i).

[53] L. Lúcio et al. "Model transformation intents and their properties". In: *Software & systems modeling* 15 (2016), pp. 647–684. DOI: https://doi.org/10.1007/s10270-014-0429-x (cit. on p. 11).

[54] T. Mens and P. Van Gorp. "A Taxonomy of Model Transformation". In: *Electronic Notes in Theoretical Computer Science* 152 (2006). Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005), pp. 125–142. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2005.10.021. URL: https://www.sciencedirect.com/science/article/pii/S1571066106001435 (cit. on pp. 10, 11, 13).

[55] A. Moin et al. "Enabling Automated Machine Learning for Model-Driven AI Engineering". In: (2022-03). DOI: https://doi.org/10.48550/arXiv.2203.02927 (cit. on p. 23).

[56] *NASA TLX*. https://humansystems.arc.nasa.gov/groups/TLX/. Accessed 05/23/2024 (cit. on p. 61).

[57] R. F. Paige, N. Matragkas, and L. M. Rose. "Evolving models in Model-Driven Engineering: State-of-the-art and future challenges". In: *Journal of Systems and Software* 111 (2016), pp. 272–280. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2015.08.047. URL: https://www.sciencedirect.com/science/article/pii/S0164121215001909 (cit. on pp. 11, 17).

[58] P. by Perforce. *State of DevOps Report 2021*. Tech. rep. 2021 (cit. on p. 5).

[59] P. by Perforce. *State of DevOps Report 2023*. Tech. rep. 2023 (cit. on p. 31).

[60] E. Planas et al. "Towards a model-driven approach for multiexperience AI-based user interfaces". In: *Softw. Syst. Model.* 20.4 (2021-08), pp. 997–1009. ISSN: 1619-1366. DOI: 10.1007/s10270-021-00904-y. URL: https://doi.org/10.1007/s10270-021-00904-y (cit. on p. 23).

[61] *Proceedings of the 2007 Conference on Databases and Information Systems IV: Selected Papers from the Seventh International Baltic Conference DBIS'2006*. NLD: IOS Press, 2007. ISBN: 9781586037154 (cit. on pp. 8, 10).

[62] S. Raedler et al. "Model-Driven Engineering for Artificial Intelligence–A Systematic Literature Review". In: *arXiv preprint arXiv:2307.04599* (2023). DOI: 10.48550/arXiv.2307.04599 (cit. on p. 23).

[63] S. Rafi et al. "DevOps Practitioners' Perceptions of the Low-code Trend". In: *Proceedings of the 16th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '22. Helsinki, Finland: Association for Computing Machinery, 2022, pp. 301–306. ISBN: 9781450394277. DOI: 10.1145/3544902.3546635. URL: https://doi.org/10.1145/3544902.3546635 (cit. on p. 23).

[64] F. M. Ribeiro et al. "A Model-Driven Solution for Automatic Software Deployment in the Cloud". In: *Information Technology: New Generations*. Ed. by S. Latifi. Cham: Springer International Publishing, 2016, pp. 591–601. ISBN: 978-3-319-32467-8. DOI: https://doi.org/10.1007/978-3-319-32467-8_52 (cit. on p. 1).

[65] A. Rodrigues da Silva. "Model-driven engineering: A survey supported by the unified conceptual model". In: *Computer Languages, Systems & Structures* 43 (2015), pp. 139–155. ISSN: 1477-8424. DOI: https://doi.org/10.1016/j.cl.2015.06.001. URL: https://www.sciencedirect.com/science/article/pii/S1477842415000408 (cit. on pp. 10, 12, 14).

[66] D. E. Rzig, F. Hassan, and M. Kessentini. "An empirical study on ML DevOps adoption trends, efforts, and benefits analysis". In: *Information and Software Technology* 152 (2022), p. 107037. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2022.107037. URL: https://www.sciencedirect.com/science/article/pii/S0950584922001537 (cit. on p. 23).

[67] J. Sandobalin. "A Model-Driven Approach to Continuous Delivery of Cloud Resources". In: *Service-Oriented Computing – ICSOC 2017 Workshops: ASOCA, ISyCC, WESOACS, and Satellite Events, Málaga, Spain, November 13–16, 2017, Revised Selected Papers*. Malaga, Spain: Springer-Verlag, 2018, pp. 346–351. ISBN: 978-3-319-91763-4. DOI: 10.1007/978-3-319-91764-1_29. URL: https://doi.org/10.1007/978-3-319-91764-1_29 (cit. on p. 21).

[68] S. Sendall and W. Kozaczynski. "Model transformation: the heart and soul of model-driven software development". In: *IEEE Software* 20.5 (2003), pp. 42–45. DOI: 10.1109/MS.2003.1231150 (cit. on pp. 8, 11).

[69] M. Shahin, M. Ali Babar, and L. Zhu. "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices". In: *IEEE Access* PP (2017-03). DOI: 10.1109/ACCESS.2017.2685629 (cit. on pp. 1, 6, 16, 17).

[70] C. Singh et al. "Comparison of Different CI/CD Tools Integrated with Cloud Platform". In: *2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*. 2019, pp. 7–12. DOI: 10.1109/CONFLUENCE.2019.8776985 (cit. on p. 8).

[71] M. Skelton and C. O'Dell. *Continuous delivery with windows and .NET*. O'Reilly Media, 2016 (cit. on p. 6).

[72] E. Soares et al. "The effects of continuous integration on software development: a systematic literature review". In: *Empirical Software Engineering* 27.3 (2022-03), p. 78. ISSN: 1573-7616. DOI: 10.1007/s10664-021-10114-1. URL: https://doi.org/10.1007/s10664-021-10114-1 (cit. on p. 6).

[73] D. Ståhl and J. Bosch. "Modeling continuous integration practice differences in industry software development". In: *Journal of Systems and Software* 87 (2014), pp. 48–59. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2013.08.032. URL: https://www.sciencedirect.com/science/article/pii/S0164121213002276 (cit. on p. 6).

[74] J. G. Süß, S. Swift, and E. Escott. “Using DevOps toolchains in Agile model-driven engineering”. In: *Software and Systems Modeling* 21 (2022-05), pp. 1–16. DOI: 10.1007/s10270-022-01003-2 (cit. on pp. 17, 23, 24).

[75] *TechRepublic*. https://www.techrepublic.com/article/best-ci-cd-pipeline-tools/. Accessed 01/23/2024 (cit. on p. 8).

[76] T. Tegeler, F. Gossen, and B. Steffen. “A Model-driven Approach to Continuous Practices for Modern Cloud-based Web Applications”. In: *2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*. 2019, pp. 1–6. DOI: 10.1109/CONFLUENCE.2019.8776962 (cit. on pp. 1, 21).

[77] T. Tegeler et al. “An Introduction to Graphical Modeling of CI/CD Workflows with Rig”. In: *Leveraging Applications of Formal Methods, Verification and Validation*. Ed. by T. Margaria and B. Steffen. Cham: Springer International Publishing, 2021, pp. 3–17. ISBN: 978-3-030-89159-6. DOI: https://doi.org/10.1007/978-3-030-89159-6_1 (cit. on pp. 1, 22).

[78] *The REVISED CI / CD Pipeline - Making Improvements*. Accessed 22/10/2023. URL: https://www.youtube.com/watch?v=OcaUQrRo7-Q (cit. on p. 6).

[79] M. Völter. “MD*/DSL Best Practices Update March 2011”. In: *Journal of Object Technology* 8 (2009-01). Ed. by AITO (cit. on p. 14).

[80] J. Wettinger et al. “Streamlining DevOps automation for Cloud applications using TOSCA as standardized metamodel”. In: *Future Generation Computer Systems* 56 (2016), pp. 317–332. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2015.07.017. URL: https://www.sciencedirect.com/science/article/pii/S0167739X15002496 (cit. on p. 22).

[81] *What are CI/CD and the CI/CD pipeline?* https://www.ibm.com/think/topics/ci-cd-pipeline. Accessed 17/10/2023 (cit. on p. 6).

[82] *What is a CI/CD Pipeline?* https://codefresh.io/learn/ci-cd-pipelines/. Accessed 17/10/2023 (cit. on p. 6).

[83] *What is a CI/CD pipeline?* https://circleci.com/blog/what-is-a-ci-cd-pipeline/. Accessed 17/10/2023 (cit. on p. 6).

[84] *What is CI/CD?* https://about.gitlab.com/topics/ci-cd/. Accessed 17/10/2023 (cit. on p. 6).

[85] Z. Zhu et al. “Exploring MDE techniques for engineering simulation models”. In: *Wireless Networks* 27 (2020), pp. 3549–3560. DOI: https://doi.org/10.1007/s11276-019-02226-w. URL: https://api.semanticscholar.org/CorpusID:209517121 (cit. on pp. 11, 14).

I

Platform-Specific Metamodels

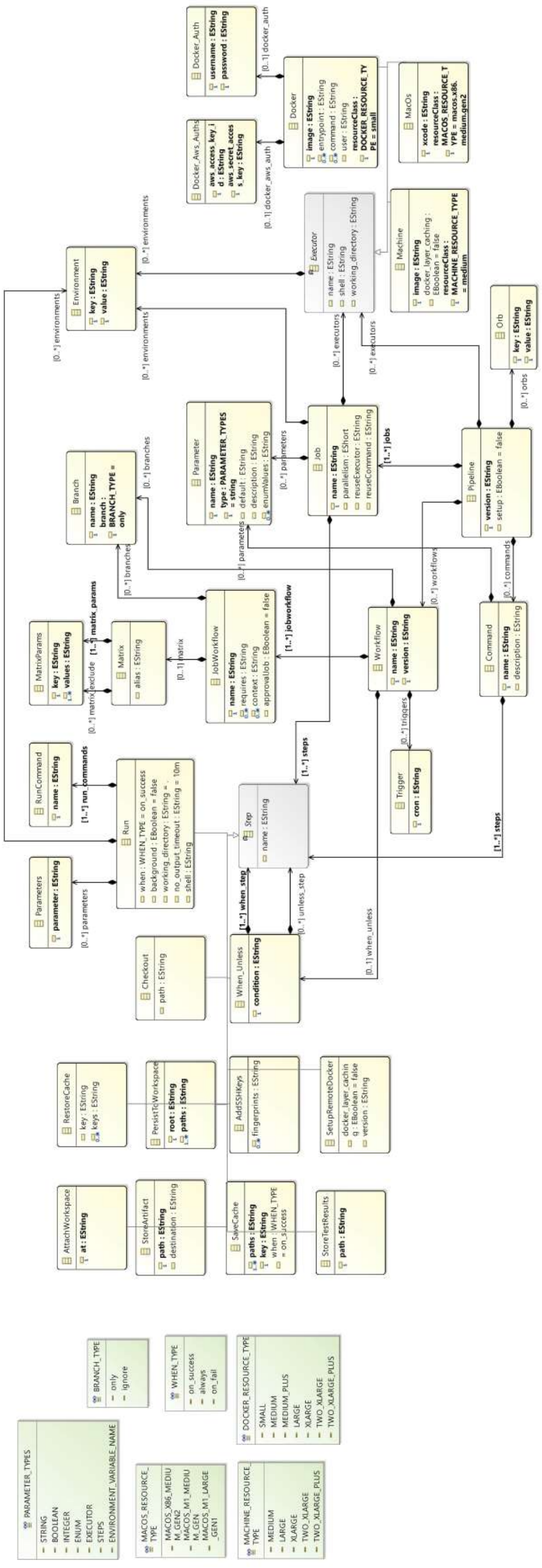


Figure I.1: CircleCI Metamodel

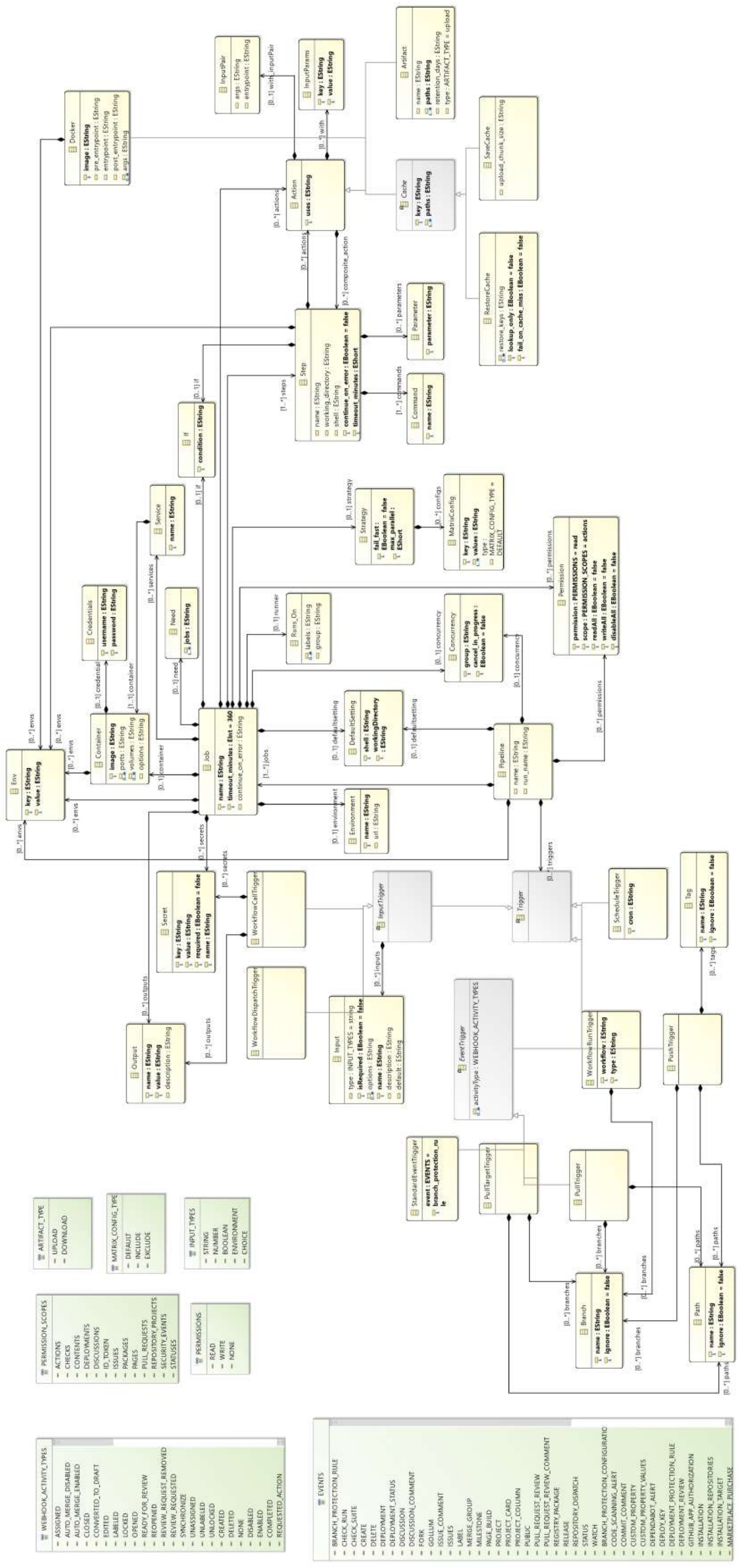


Figure I.2: GitHub Actions Metamodel

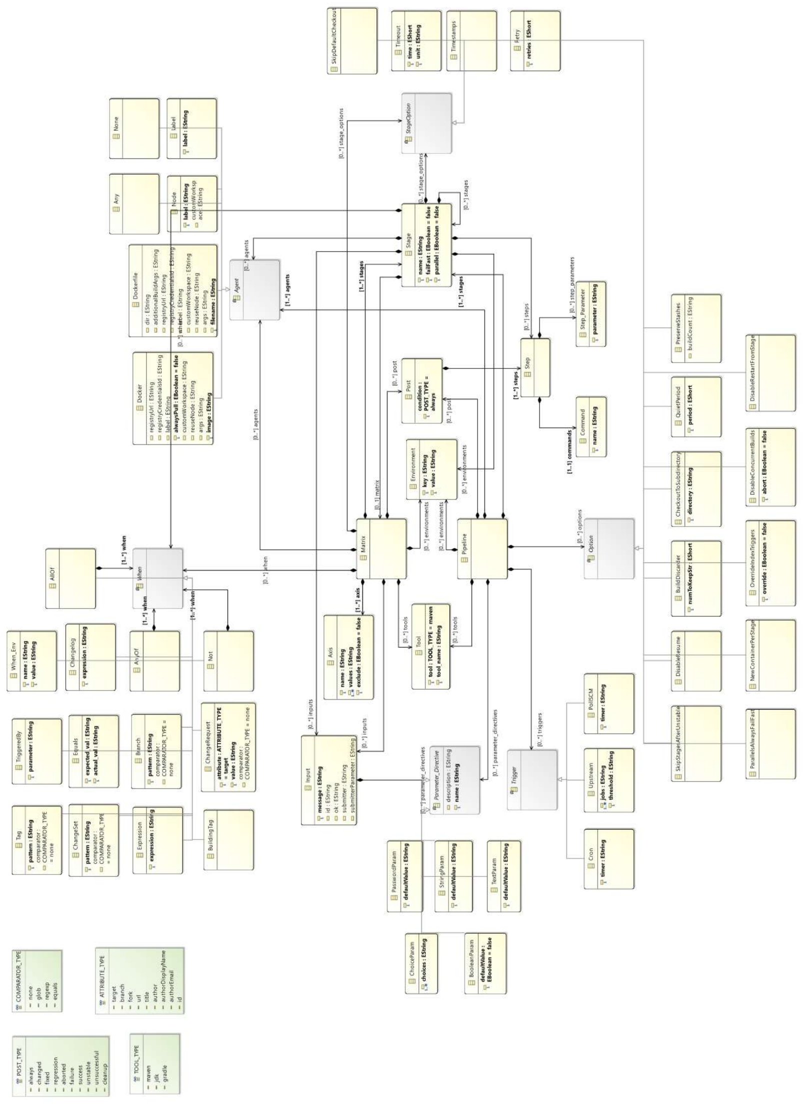


Figure I.3: Jenkins Metamodel

II

Platform-Specific OCL Invariants

Listing II.1: CircleCI OCL Invariants

```
package circleCI_metamodel : circleCI_metamodel = ... {
    abstract class Executor
    {
        attribute name : String[?];
        ...
        invariant nonDuplicateExecutorName('Duplicate Executor Name. Choose a
        different Executor name to ensure uniqueness within the pipeline.'):
            Pipeline.allInstances().jobs.executors->union(Pipeline
            .allInstances().executors)->forAll(p | p <> self implies p.name
            <> self.name);

        invariant mandatoryPipelineExecutorName('Pipeline Executor Name is
        empty. Define Executor name.'):
            Pipeline.allInstances().executors->forAll(p | p.name->notEmpty()
            and p.name <> null);
    }
}
```

Listing II.2: GHA OCL Invariants

```
package gHA_metamodel : gHA_metamodel = 'http://.../gHA_metamodel' {
    abstract class Trigger
    {
        invariant UniqueTriggerTypes('Only one instance of each
        trigger type (except ScheduleTrigger) is allowed'):
            not self.oclIsTypeOf(ScheduleTrigger) implies
                Trigger.allInstances()->select(t | t.oclType() =
                self.oclType())->size() <= 1;
    }
    class Need
    {
        attribute jobs : String[+|1] { ordered };
        invariant existingJobsNeeded('Referenced jobs do not exist.'):
            self.jobs->forAll(jobName | Job.allInstances()->exists(j |
            j.name = jobName));
    }
}
```

Listing II.3: Jenkins OCL Invariants

```
package jenkins_metamodel : jenkins_metamodel = ... {
    class Upstream extends Trigger
    {
        attribute jobs : String[+|1] { ordered };
        ...
        invariant existingJobs('Referenced jobs do not exist.'):
            self.jobs->forAll(jobName | Stage.allInstances()->exists(s |
            s.name = jobName));
    }
}
```

III

Platform-Specific Grammars

Listing III.1: Circle grammar definition

```
grammar org.xtext.example.circleci.Circleci with org.eclipse.xtext
.common.Terminals

import "http://www.example.org/circleCI_metamodel"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

Pipeline returns Pipeline:
    'Pipeline'
    ((BEGIN (setup?='setup')? 'version' version=EString END))
    ((orbs+=Orb)+ NEWLINE?)?
    ((commands+=Command)+ NEWLINE?)?
    ((executors+=Executor)+ NEWLINE?)?
    ((jobs+=Job)+ NEWLINE?)
    ((workflows+=Workflow)+ NEWLINE?)?
;

Command returns Command:
'Command'
(BEGIN
    'name' name=EString
    ('description' description=EString)?
    (parameters+=Parameter)*
    (steps+=Step)+
END);

Parameter returns Parameter:
'Parameter'
(BEGIN
    'name' name=EString
    'type' type=PARAMETER_TYPES
    ('default' default=EString )?
    ('description' description=EString )?
    ('enumValues' enumValues+=EString (',' enumValues+=EString)*)?
END);
```

Listing III.2: GHA grammar definition

```
grammar org.xtext.example.gha.GHA with org.eclipse.xtext
.common.Terminals

import "http://www.example.org/gHA_metamodel"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

Pipeline returns Pipeline:
'Pipeline'
((BEGIN'name' name=EString ('run-name' run_name=EString)? END))?
((envs+=Env)+ NEWLINE?)?
((permissions+=Permission)+ NEWLINE?)?
(defaultsetting=DefaultSetting NEWLINE?)?
(concurrency=Concurrency NEWLINE?)?
((triggers+=Trigger)+ NEWLINE?)?
((jobs+=Job)+ NEWLINE?);

Permission returns Permission:
'Permission'
(BEGIN
    (readAll?='readAll')?
    (writeAll?='writeAll')?
    (disableAll?='disableAll')?
    'permission' permission=PERMISSIONS
    'scope' scope=PERMISSION_SCOPES
END);

SaveCache returns SaveCache:
'SaveCache'
(BEGIN
    'uses' uses="\"actions/cache/save@v4\""
    'key' key=EString
    'paths' paths+=EString ( "," paths+=EString)*
    ('upload_chunk_size' upload_chunk_size=EString)?
    (composite_action+=Step)*
    (^with+=InputParams)*
    (with_inputPair=InputPair)?
END);
```

Listing III.3: Jenkins grammar definition

```
grammar org.xtext.example.jenkins.Jenkins with org.eclipse.xtext
.common.Terminals

import "http://www.example.org/jenkins_metamodel"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

Pipeline returns Pipeline:
    'Pipeline' NEWLINE
    ((agents+=Agent)+ NEWLINE?)
    ((options+=Option)+ NEWLINE?)?
    ((triggers+=Trigger)+ NEWLINE?)?
    ((parameter_directives+=Parameter_Directive)+ NEWLINE?)?
    ((environments+=Environment)+ NEWLINE?)?
    ((tools+=Tool)+ NEWLINE?)?
    ((stages+=Stage)+ NEWLINE?)
    ((post+=Post)+ NEWLINE?)?
;

StringParam returns StringParam:
'StringParam'
(BEGIN
    'name' name=EString
    ('description' description=EString )?
    'defaultValue' defaultValue=EString
END);

Matrix returns Matrix:
'Matrix'
(BEGIN
    (axis+=Axis)+
    (stages+=Stage)+
    (inputs+=Input)*
    (when+=When)*
    (environments+=Environment)*
    (agents+=Agent)*
    (tools+=Tool)*
    (post+=Post)*
    (stage_options+=StageOption)*
END);
```

IV

XMI2DSL Plugin

Listing IV.1: XMI2DSL plugin

```
public class XMIReader {

    public static void convertXMI2DSL(String filePath, Shell shell) {

        String extension = "";
        try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
            br.readLine();
            String secondLine = br.readLine();
            if (secondLine != null) {
                if (secondLine.contains("circleCI")) {
                    extension = ".circleci";
                } else if (secondLine.contains("gHA")) {
                    extension = ".gha";
                }
                else {
                    extension = ".jenkins";
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
            return;
        }

        ResourceSet resourceSet = new ResourceSetImpl();
        URI xmiUri = URI.createFileURI(filePath);
        Resource resource = resourceSet.getResource(xmiUri, true);

        EObject rootElement = resource.getContents().get(0);
        if (rootElement == null) {
            System.err.println("Root␣element␣not␣found␣in␣the␣XMI␣file.");
            return;
        }
        List<String> xtextLines = null;
        if(extension.contains(".circleci")) {
            CircleCiFormatter circleFormatter = new CircleCiFormatter();
            xtextLines = circleFormatter.generateXtext(rootElement);
        }
        else if(extension.contains(".gha")) {
            GHAFormatter ghaFormatter = new GHAFormatter();
            xtextLines = ghaFormatter.generateXtext(rootElement);
        }
        else if(extension.contains(".jenkins")) {
            JenkinsFormatter jenkinsFormatter = new JenkinsFormatter();
            xtextLines = jenkinsFormatter.generateXtext(rootElement);
        }
        writeXtextToFile(xtextLines, filePath, extension);
    }

    private static void writeXtextToFile(List<String> xtextLines, String filePath,
        String extension) {

        String directoryPath = new File(filePath).getParent();
        String baseFileName = new File(filePath).getName();
        String fileNameWithoutExt =
            baseFileName.substring(0, baseFileName.lastIndexOf('.'));
        String outputPath =
            Paths.get(directoryPath, fileNameWithoutExt + extension).toString();

        try (BufferedWriter writer = new BufferedWriter(new FileWriter(outputPath))) {
            for (String line : xtextLines) {
                writer.write(line);
                writer.newLine();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

V

Platform-Specific Formatters

Listing V.1: CircleCI Formatter

```java
public class CircleCiFormatter {

    public List<String> generateXtext(EObject object) {
        List<String> xtextLines = new ArrayList<>();
        generateXtextLines(object, xtextLines, 0);
        return xtextLines;
    }

    private void generateXtextLines(EObject object, List<String> xtextLines,
    int indentLevel) {
        EClass eClass = object.eClass();
        String className = eClass.getName();
        String indent = "\t".repeat(indentLevel);

        switch (className) {
            case "Pipeline":
                xtextLines.add(indent + "Pipeline");
                appendAttributesAndReferences(object, xtextLines, eClass,
                indentLevel);
                break;
            case "Command":
                xtextLines.add(indent + "Command");
                appendCommandAttributesAndRef(object, xtextLines, eClass,
                indentLevel + 1);
                xtextLines.add("");
                break;
            ...
            default:
                appendAttributesAndReferences(object, xtextLines, eClass,
                indentLevel + 1);
                break;
        }
    }
    private void appendAttributesAndReferences(EObject object,
    List<String> xtextLines, EClass eClass, int indentLevel) {
        String indent = "\t".repeat(indentLevel);

        for (EAttribute attribute : eClass.getEAllAttributes()) {
            Object value = object.eGet(attribute);

            if (value != null) {
                if (value instanceof PARAMETER_TYPES) {
                    xtextLines.add(indent + attribute.getName() + "␣" +
                    ((Enum<?>) value).name());
                } else if (value instanceof String &&
                        !((String) value).isEmpty()) {
                    xtextLines.add(indent + attribute.getName() + "␣\"" +
                               value + "\"");
                } else if (value instanceof Boolean && (Boolean) value) {
                    xtextLines.add(indent + attribute.getName());
                }
            }
        }
        processReferences(object, xtextLines, eClass, indentLevel);
    }

    private void processReferences(EObject object, List<String> xtextLines,
    EClass eClass,
        int indentLevel) {
        List<EObject> references = getReferences(object, eClass);
        for (EObject reference : references) {
            generateXtextLines(reference, xtextLines, indentLevel);
        }
    }
}
```

Listing V.2: GHA Formatter

```
public class GHAFormatter {

    public List<String> generateXtext(EObject object) {
        List<String> xtextLines = new ArrayList<>();
        generateXtextLines(object, xtextLines, 0);
        return xtextLines;
    }

    private void generateXtextLines(EObject object, List<String> xtextLines,
    int indentLevel) {
        EClass eClass = object.eClass();
        String className = eClass.getName();
        String indent = "\t".repeat(indentLevel);

        switch (className) {
            case "Pipeline":
                xtextLines.add(indent + "Pipeline");
                appendAttributesAndReferences(object, xtextLines, eClass,
                indentLevel);
                break;
            case "Job":
                xtextLines.add(indent + "Job");
                appendJobAttributesAndReferences(object, xtextLines, eClass,
                indentLevel + 1);
                xtextLines.add("");
                break;
            ...
            default:
                appendAttributesAndReferences(object, xtextLines, eClass,
                indentLevel + 1);
                break;
        }
    }

    private void appendAttributesAndReferences(EObject object,
    List<String> xtextLines, EClass eClass, int indentLevel) {
        String indent = "\t".repeat(indentLevel);

        for (EAttribute attribute : eClass.getEAllAttributes()) {
            Object value = object.eGet(attribute);

            if (value != null) {
                if (value instanceof String && !((String) value).isEmpty()) {
                    xtextLines.add(indent + attribute.getName() + "␣\"" +
                                   value + "\"");
                } else if (value instanceof Boolean && (Boolean) value) {
                    xtextLines.add(indent + attribute.getName());
                } else if (value instanceof List<?>) {
                    appendEnumValues(attribute.getName(), (List<?>) value,
                                     xtextLines, indentLevel);
                }
            }
        }
        processReferences(object, xtextLines, eClass, indentLevel);
    }

    private void processReferences(EObject object, List<String> xtextLines,
    EClass eClass, int indentLevel) {
        List<EObject> references = getReferences(object, eClass);
        for (EObject reference : references) {
            generateXtextLines(reference, xtextLines, indentLevel);
        }
    }
}
```

Listing V.3: Jenkins Formatter

```
public class JenkinsFormatter {

    public List<String> generateXtext(EObject object) {
        List<String> xtextLines = new ArrayList<>();
        generateXtextLines(object, xtextLines, 0);
        return xtextLines;
    }

    private void generateXtextLines(EObject object, List<String> xtextLines,
    int indentLevel) {
        EClass eClass = object.eClass();
        String className = eClass.getName();
        String indent = "\t".repeat(indentLevel);

        switch (className) {
            case "Pipeline":
                xtextLines.add(indent + "Pipeline");
                appendAttributesAndReferences(object, xtextLines, eClass,
                indentLevel);
                break;
            case "Job":
                xtextLines.add(indent + "Job");
                appendJobAttributesAndReferences(object, xtextLines, eClass,
                indentLevel + 1);
                xtextLines.add("");
                break;
            ...
            default:
                appendAttributesAndReferences(object, xtextLines, eClass,
                indentLevel + 1);
                break;
        }
    }

    private void appendAttributesAndReferences(EObject object,
    List<String> xtextLines, EClass eClass, int indentLevel) {

        String indent = "\t".repeat(indentLevel);

        for (EAttribute attribute : eClass.getEAllAttributes()) {
            Object value = object.eGet(attribute);
            if (value != null) {
                if (value instanceof String && !((String) value).isEmpty()) {
                    xtextLines.add(indent + attribute.getName() + "␣\"" +
                                   value + "\"");
                } else if (value instanceof Boolean && (Boolean) value) {
                    xtextLines.add(indent + attribute.getName());
                } else if (value instanceof List<?>) {
                    appendEnumValues(attribute.getName(), (List<?>) value,
                                     xtextLines, indentLevel);
                }
            }
        }
        processReferences(object, xtextLines, eClass, indentLevel);
        ...
    }

    private void processReferences(List<EObject> references,
    List<String> xtextLines,
        int indentLevel) {
        for (EObject reference : references) {
            generateXtextLines(reference, xtextLines, indentLevel);
        }
    }
}
```

VI

Platform-Specific ATL Rules

Listing VI.1: CICD2GHA ATL

```
-- @path GHA=/GHA_metamodel/model/gHA_metamodel.ecore
-- @path CICD=/CICD_metamodel/model/cICD_metamodel.ecore

module CICD2GHA;
create OUT : GHA from IN : CICD;

rule Pipeline2Pipeline {
    from
        s : CICD!Pipeline
    to
        t : GHA!Pipeline(
            name <- if not s.name.oclIsUndefined() then
                        s.name
                    else
                        OclUndefined
                    endif,
            run_name <- OclUndefined,
            envs <- s.pipeline_environment->collect(env |
                                            thisModule.transformEnvironment(env)),
            jobs <- s.jobs,
            triggers <- if s.triggers->notEmpty() and (s.inputs->notEmpty() or
                        s.output->notEmpty()) then
                            s.triggers->union(s.inputs)->union(s.output)->
                            collect(trigger |
                                if trigger.oclIsTypeOf(CICD!ScheduleTrigger) then
                                    thisModule.transformScheduleTrigger(trigger)
                                else
                                    Sequence{
                                        thisModule.CreateWorkflowDispatch(s.inputs,
                                        s.output)
                                    }
                                endif
                            )
                        else if s.triggers->isEmpty() and (s.inputs->notEmpty() or
                        s.output->notEmpty()) then
                            Sequence{thisModule.CreateWorkflowDispatchTrigger(s.inputs,
                            s.output)}
                        else
                            s.triggers->collect(trigger |
                                thisModule.transformScheduleTrigger(trigger))
                            endif
                            endif
        )
}

lazy rule transformScheduleTrigger {
    from
        s : CICD!ScheduleTrigger
    to
        t : GHA!ScheduleTrigger(
            cron <- s.timer
        )
}

unique lazy rule CreateWorkflowDispatch {
    from
        inputs : Sequence(CICD!Input),
        outputs : Sequence(CICD!Output)
    to
        workflowCallTrigger : GHA!WorkflowCallTrigger(
            inputs <- inputs,
            outputs <- outputs
        )
}
```

Listing VI.2: CICD2Jenkins ATL

```
-- @path Jenkins=/Jenkins_metamodel/model/jenkins_metamodel.ecore
-- @path CICD=/CICD_metamodel/model/cICD_metamodel.ecore

module CICD2Jenkins;
create OUT : Jenkins from IN : CICD;

rule Pipeline2Pipeline {
    from
        s : CICD!Pipeline
    to
        t : Jenkins!Pipeline(
            environments <- s.pipeline_environment ->collect(env |
                                    thisModule.transformEnvironment(env)),
            stages <- s.jobs->collect(job | thisModule.Job2Stage(job)),
            triggers <- s.triggers->collect(trigger |
                            thisModule.transformScheduleTrigger(trigger)),
            agents <- if s.agents->isEmpty() then
                            thisModule.CreateAnyAgent('')
                        else
                            s.agents->collect(agent |
                                if agent.container.oclIsUndefined() then
                                    thisModule.Agent2Node(agent)
                                else
                                    thisModule.DockerContainer2Docker(agent.container,
                                    agent.labels->first())
                                endif
                            )
```

```
                    endif,
            parameter_directives <- if s.inputs->notEmpty() then
                                        s.inputs->collect(input |
                                            if input.type = #STRING then
                                                thisModule.Input2StringParam(input)
                                            else if input.type = #TEXT then
                                                thisModule.Input2TextParam(input)
                                            ...
                                            else
                                                OclUndefined
                                            endif
                                            ...
                                        )
                                    else
                                        OclUndefined
                                    endif
        )
}
lazy rule CreateAnyAgent {
    from
        blank : String
    to
        agent : Jenkins!Any()
}
```

VII

Platform-Specific Acceleo Templates

Listing VII.1: GHA Template

```
[comment encoding = UTF-8 /]
[module generate('http://www.example.org/gHA_metamodel')]

[template public generateElement(aPipeline : Pipeline)]
[comment @main/]
[file ('GitHub␣Actions' + '.yml', false, 'UTF-8')]
[if (aPipeline.name->notEmpty())]
['name:␣' + aPipeline.name/]:
[/if]
[if (aPipeline.run_name->notEmpty())]
['run-name:␣' + aPipeline.run_name/]:
[/if]
[if (envs->notEmpty())]
env:
[for (e: Env | envs)]
    [generateEnv(e)/]
[/for]
[/if]
[if (permissions->notEmpty())]
[if (permissions->exists(p | p.disableAll = true))]
permissions: {}
[elseif (permissions->exists(p | p.readAll = true))]
permissions: read-all
[elseif (permissions->exists(p | p.writeAll = true))]
permissions: write-all
[/if]
[if (permissions->forAll(p | p.writeAll = false and p.disableAll = false and
    p.readAll = false))]
permissions:
[/if]
[for (p: Permission | permissions->select(p | p.writeAll = false and
    p.disableAll = false and p.readAll = false))]
    [generatePermissions(p)/]
[/for]
```

```
[/if]
[if (defaultsetting->notEmpty())]
defaults:
[for (default: DefaultSetting | defaultsetting)]
    [generateDefaults(default)/]
[/for]
[/if]
[if (aPipeline.concurrency->notEmpty())]
concurrency:
    [generateConcurrency(aPipeline.concurrency)/]
[/if]
[generateTriggers(aPipeline)/]
[generateJobs(aPipeline)/]
[/file]
[/template]
```

Listing VII.2: Jenkins Template

```
[comment encoding = UTF-8 /]
[module generate('http://www.example.org/jenkins_metamodel')]


[template public generateElement(aPipeline : Pipeline)]
[comment @main/]
[file ('Jenkins' + '.yml', false, 'UTF-8')]
pipeline {
[for (a: Agent | aPipeline.agents)]
    [generateAgent(a)/]
[/for]
    [generateOptions(aPipeline)/]
    [generateTriggers(aPipeline)/]
    [generateParameterDirective(aPipeline)/]
[if (aPipeline.environments->notEmpty()) ]
    environment {
    [for (e: Environment | aPipeline.environments)]
        [generateKeyValue(e)/]
    [/for]
    }
[/if]
[if (aPipeline.tools->notEmpty())]
    tools {
    [for (t: Tool | aPipeline.tools)]
        [t.tool.toString()/] [t.tool_name/]
    [/for]
    }
[/if]
    [generateStages(aPipeline)/]
[if (aPipeline._post->notEmpty())]
```

```
    post {
        [aPipeline._post.condition.toString()/] {
            [for (p: Post | aPipeline._post)] [generatePostSteps(p)/] [/for]
        }
    }
[/if]
}
[/file]
[/template]

[template public generateNodeAgent(aNode: Node)]
agent {
    node {
        label '[aNode.label/]'
        [if (aNode.customWorkspace->notEmpty())]
        customWorkspace '[aNode.customWorkspace/]'
        [/if]
    }
}
[/template]
```

VIII

Usability Average Score per Question

Figure VIII.1 compares the average SUS scores per question between Textual and VS Code, illustrating the contrast in scores across different questions to highlight the strengths and weaknesses of each interface.

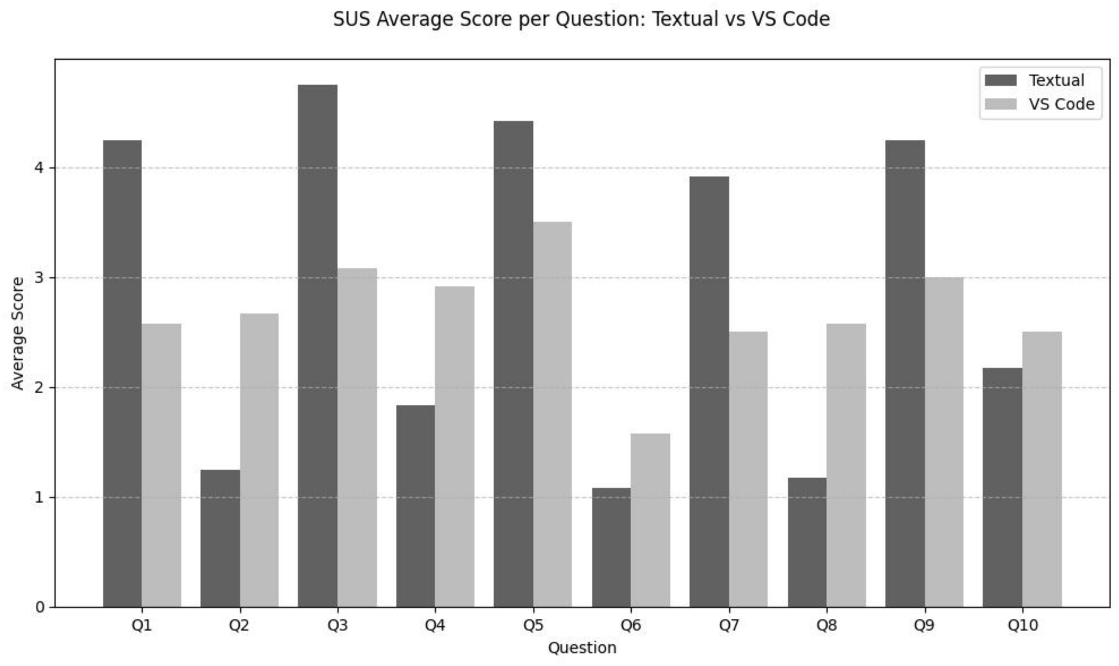


Figure VIII.1: SUS Average Score per Question

The Textual interface not only outperforms the VS Code interface across all SUS questions but also shows particularly strong performance in positively phrased questions (odd-numbered), with notable differences in Q1, Q3, and Q7. While even-numbered questions (negatively phrased) still favor Textual, the margin is narrower.

The statistical analysis reinforces the observation that, although there is a moderate positive correlation between the SUS scores of Textual and VS Code, the differences in usability scores are not statistically significant.

Table VIII.1: SUS per Question Textual vs VS Code

Statistic	Result	P-value
Mann-Whitney U Test	52.0	0.9096

Similarly, Figure VIII.2 compares the Visual and Buddy interfaces across SUS questions. The graph shows that Buddy has better results than Visual, although the differences are less significant compared to the Textual vs VS Code approaches.

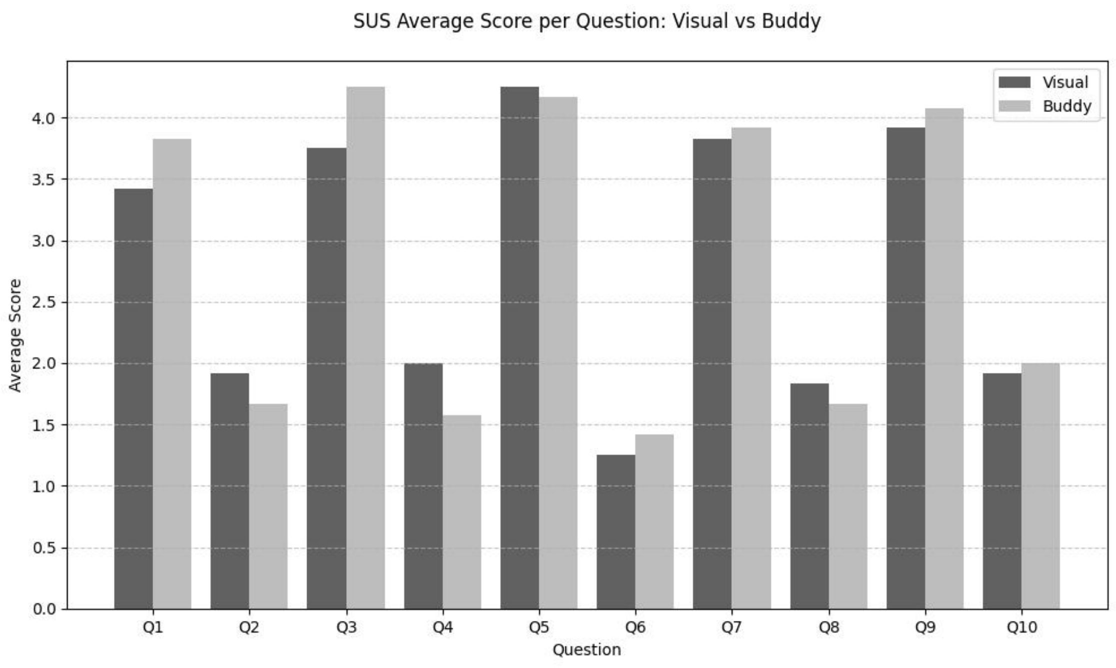


Figure VIII.2: SUS Average Score per Question

Concerning visual approaches, Buddy generally outperforms Visual across most questions. Notably, both tools achieve high scores on questions related to specific usability aspects (Q3 and Q5), indicating strong performance in these areas. Despite Buddy's overall higher performance, the discrepancies between the two tools are not substantial, affirming that Visual still provides a competitive user experience in certain aspects.

Table VIII.2: SUS per Question Visual vs Buddy

Statistic	Result	P-value
Mann-Whitney U Test	48.0	0.9095

These results indicate a very strong positive correlation between the usability scores of Visual and Buddy. The Mann-Whitney U test results show no significant difference in the scores, confirming that users perceive similar usability in both tools.