A STUDY ON JUSTIFYING PLATFORM-INDEPENDENT CI/CD PIPELINES

BY

DEESHA PATEL, M.Eng.

a Report
submitted to the Computing and Software
and the School of Graduate Studies
of McMaster University
in partial fulfilment of the requirements
for the degree of
Masters of Engineering

Masters of Engineering (2023) McMaster University

(Computing and Software) Hamilton, Ontario, Canada

TITLE:	A study on Justifying platform-independent CI/CD Pipelines
AUTHOR:	Deesha Patel M.Eng. in Computing and Software, McMaster University, Hamilton, Canada
SUPERVISOR:	Dr. Sébastien Mosser
NUMBER OF PAGES:	xii, 75

Lay Abstract

Have you ever thought about handling future challenges by monitoring changes regularly? The tracking of changes is important as it convey the real meaning and intention of those changes. This research centers on developing Justification diagrams to visually represent operational changes. The study investigates different platforms to assess the efficacy of Justification diagram representation in comprehending and managing changes. It is crucial to connect these changes with the real world by aligning actual operations with these representations. This report details these findings and outlines directions for future work.

Abstract

Automated software development processes are facilitated by a pipeline within the software life cycle where output from one process becomes input for the next process. They help software industries to enhance processes for integrating and delivering new changes frequently to the market. While in-house pipelines are meticulously documented to mitigate the risk of leakages, the same is also required in software development pipelines to handle future challenges. Surprisingly, limited research has delved into this direction. The purpose of a pipeline is to enhance the quality of the software development. So, it is important to handle the quality of the pipeline itself. This report explores the feasibility of using Justification Diagrams for documenting changes in pipelines. Initially, we wrote a justification for a large open-source GitHub pipeline, shedding light on the significance of Justification Diagrams. Subsequently, we surveyed various platforms compatible with this justifying approach, aiming for precise reasoning, intentions, and results for their changes. Finally, we endeavored to pinpoint tangible operations within the pipeline, enhancing the readability of the Justification. The report offers insights into potential advancements in change tracking methodologies.

Acknowledgements

First and foremost, I am deeply thankful to **Dr. Sebastien Mosser**, whose expertise, mentorship, and unwavering support played a pivotal role in shaping and refining my master's studies and project. Your insightful feedback and dedication to excellence have been truly inspiring.

I extend my appreciation to **Jean-Michel Bruel** for his contribution to a new direction of research. His collective efforts significantly enriched the scope and depth of this study.

I am also grateful for the resources and guidance provided by **Corinne Pulgar**, which facilitated the deep understanding of the different concepts in this research.

A big thank you to my parents for providing me with financial support along the way. I also express my gratitude to my husband **Adarsh Patel**. His patience, understanding, and endless encouragement were the pillars that sustained me during the rigorous process of completing this paper.

Lastly, to my friends and family, your unwavering encouragement and understanding during the challenges of the research process have been a source of strength. Your support made this endeavor more fulfilling

Contents

List of Figures

List of Tables

Abbreviations

Abbreviations

ADR	Architecture Decision Record
CD	Continuous Delivery/Deployment
CI	Continuous Integration
CT	Continuous Testing
GA	GitHub Action
iBGP	Interior Border Gateway Protocol
IGP	Interior Gateway Protocol
IT	Information Technology
JD	Justification Diagram
VCS	Version Control system
YAML	YAML Ain't markup language

Chapter 1

Introduction

Consider an example of a car production process. Before the final release, a car goes through several preliminary development and testing stages. Firstly, the designer creates and verifies the blueprint for a car, considering various modalities. It is important to record this blueprint for the next step of production. An automotive engineer should have that blueprint. With the help of a blueprint, a trained engineer can construct the body of a car using the documents that guide it throughout different processes. Once the body is designed according to the plan, managers verify building plans with the blueprint. Then the vehicle moves towards the further stages of component installation like doors, roofs, wheels, and windows. For tracking and verifying requirements, engineers need to develop the whole document. After the production phase, functional car testing is performed with the manufacturing requirements before the release. The documentation played a vital role in understanding and verifying the flow of the processes in the whole car building process. Documentation is also essential in the software industry, especially in software pipelines. The software pipelines are the consecutive steps of software development processes such as build, test, and release which are executed in an order with an output of the first process considered as input for the next process.

1.1 Problem and Motivation

The pipelines provide automatic software lifecycle processes that can enhance the reliability and speed of the software lifecycle [1]. It is essential to write adequate and meaningful pipeline instructions that lead to the success of the whole software process. However, existing work for achieving a guarantee for the quality of the pipeline is limited. It is hard to maintain and track the justification of the pipeline itself. The following are challenges for engineers after designing pipelines:

- The engineer starts by targeting a design decision and then implementing constraints around it, but they usually fail to answer the exact reasoning for working in a particular way.
- It is difficult for engineers to track and pass those changes and intentions in the existing workflow to other team members.
- When there are "changes" in the file, the "why" is never addressed.
- Without a proper explanation of the exact "why" in the project, large organizations are not able to replicate those same effective modules in other projects.

One of the solutions is to use the Justification Diagram (JD) to address these challenges. JD is a diagrammatic representation based on the Toulmin schema. This type of solution documents the changes and justifies those with proper intentions. Upcoming chapters describe the JD in more detail with their limitations. By considering the JD, we try to answer the following questions:

1. How can semantics be extracted from the histories of pipelines to write the justification diagram?

2. How can we extend the usage of the JD for different pipeline platforms?

3. What suggested operations can support the approach to fill the gaps between workflow and JDs?

The primary objective of this report is to observe how the Justification Diagram can help to track the changes that occur with the evolution of a Continuous Integration/continuous delivery (CI/CD) pipeline. We will use docker-compose as an example case study. The report aims to broaden the application of the JD methodology to encompass GitHub, GitLab, and Azure environments. In addition to this analysis, the report proposes a new direction for Operational Justification Diagrams which targets to address the gap between JD and workflow itself.

The chapter 2 introduces background information of this thesis. Chapter 3 contains a case study that addresses the question 1. Question 2 is addressed in chapter 4. Next, chapter 5 will include the solution for operations that address question 3. Finally, chapter 6 concludes the report and addresses future work.

Chapter 2

Background

This chapter provides an introduction to several technical concepts involved in this project, primarily focusing on the introduction of DevOps, pipelines, and the Justification Diagram. Additionally, it highlights the various components and alternatives of the Justification Diagram.

2.1 Introduction to DevOps

The term "DevOps" was coined in the year 2009 [10]. DevOps represents a combined approach involving both Development and IT Operations [7]. This approach fosters a cultural shift in the software development industry, promoting collaborative work among development, testing, deployment, and monitoring teams, leveraging appropriate technologies.

Figure 2.1 illustrates the different phases of the DevOps workflow for both Development (Dev) and Operations (Ops). The Dev part consists of planning, coding, building, and testing, while Ops consists of releasing, deploying, operating, and monitoring. The

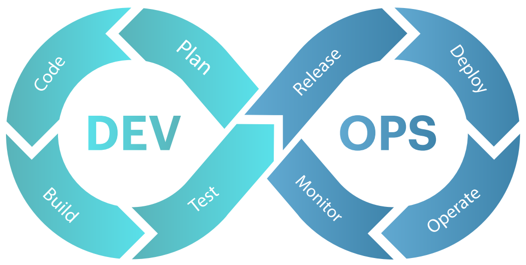


Figure 2.1: Common DevOps lifecycle [17]

software development process in DevOps begins with project planning and writing code for the product. In the building stage, the developed code is built for testing purposes. Continuous testing is a crucial part of this phase to ensure that the project functions as expected. In the release phase, the Ops team can verify the project and create a build for the production server. After thorough checks for vulnerabilities and bugs, the project moves to the deployment stage, making it accessible to end-users. Configuration and system management occur during the operating process handled by the operations team. The system also undergoes monitoring to track its behavior. These processes are iterative for the project's features. DevOps includes components such as Continuous Integration, Continuous Delivery, Continuous Deployment, and Continuous Testing in their processes.

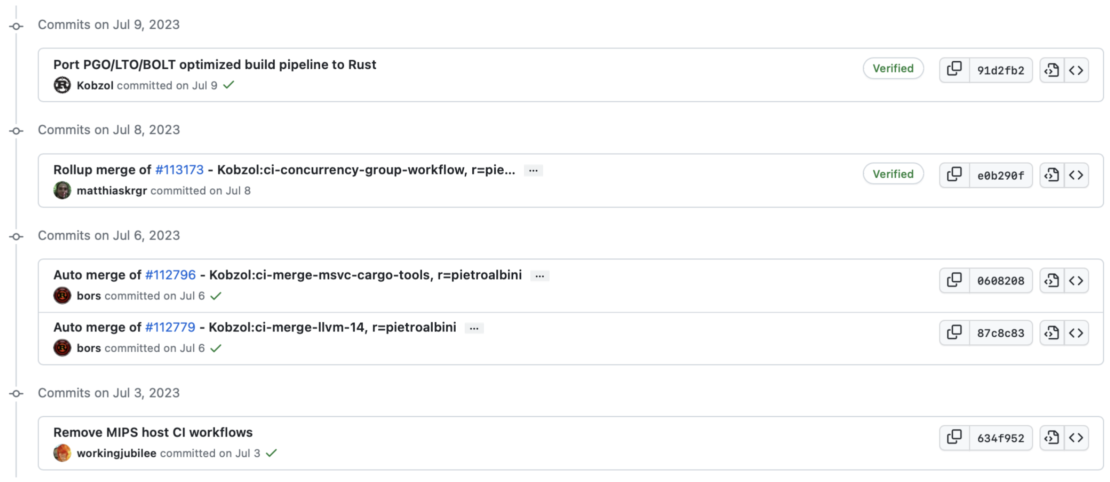
2.2 CI/CD Pipelines

Different components in the DevOps pipeline are used to achieve continuity in the software development life cycle. The first process is continuous integration. The repository plays an important role in achieving integration which is a platform to integrate new changes. A developer can push their code to repository several times throughout the day instead of keeping it local to their machine. The build system tries to find defects in new code and integrate it with existing code after it is pushed to a repository [27]. Once the build system verifies the new code, it automatically merges it into the repository. Another component of DevOps is to test those codes automatically using Continuous Testing (CT). CT not only covers software testing but also carries out testing of developed software on production-like systems to check system behavior. Continuous Deployment (CD) is used to deploy applications to the production server without human interference. Utilizing the feature of Continuous Deployment (CD) not only enhances software quality but also boosts customer satisfaction by enabling them to experience live implementations [11].

There are a variety of DevOps tools for different software development phases. Continuous planning is managed with tools like Jira, Trello, and GitHub. In the coding phase, Git and GitHub provide version control to handle multiple pushes to the central repository. Jenkins, Travis CI, and GitHub CI/CD workflows are valuable for building software. For Continuous Testing, Selenium, JUnit, and SonarQube ensure code quality. Docker, Kubernetes, and GitHub are commonly used for deployment and containerization, enabling Continuous Deployment. Prometheus, the ELK Stack, Splunk, New Relic, Nexus, JFrog Artifactory, and GitHub Actions are the most used Continuous Monitoring and feedback tools available in the market. By combining these tools, organizations can achieve enhanced software quality, collaboration, and a streamlined CI/CD pipeline. All these processes are continuous which is an key factor of DevOps pipelines.

2.3 Pipeline evolution

As mentioned earlier, the stages of the DevOps pipeline are iterative. An example to achieve can be described using GitHub. GitHub provides GitHub Workflow, which can be added to the project to implement CI/CD. Depending on the system's requirements, pipeline configuration gets regularly updated. Consider the Rust pipeline example depicted in Figure 2.2. This visualization captures the iterative evolution of the pipeline, showcasing multiple instances where add, delete, and update operations to the CI processes are performed in response to evolving project requirements.


Commits on Jul 9, 2023

Port PGO/LTO/BOLT optimized build pipeline to Rust
Kobzol committed on Jul 9 ✓ — Verified — 91d2fb2

Commits on Jul 8, 2023

Rollup merge of #113173 - Kobzol:ci-concurrency-group-workflow, r=pie… …
matthiaskrgr committed on Jul 8 — Verified — e0b290f

Commits on Jul 6, 2023

Auto merge of #112796 - Kobzol:ci-merge-msvc-cargo-tools, r=pietroalbini …
bors committed on Jul 6 ✓ — 0608208

Auto merge of #112779 - Kobzol:ci-merge-llvm-14, r=pietroalbini …
bors committed on Jul 6 ✓ — 87c8c83

Commits on Jul 3, 2023

Remove MIPS host CI workflows
workingjubilee committed on Jul 3 ✓ — 634f952

Figure 2.2: Rust pipeline evolution [22]

2.4 Requirement of Justification Diagram

Logging all changes with a proper explanation in current and future releases will make it easier to apply in other projects and deal with future challenges. Currently, GitHub provides a commit message with the new changes, but it may not be sufficient to analyze or understand every commit message. Sometimes, based on a programmer's expertise, it may contain incorrect information and cannot be easily verified. Therefore, we need a justification for every update to provide more clarity on those changes. Chapter 3 further describes the evolution of the pipeline and how the Justification Diagram explains the purpose of the change.

2.5 Introduction to Justification Diagram

Change inevitably carries a certain degree of risk, so it is necessary to document the purpose of those changes with sufficient evidence and data to reuse/handle/revert those changes. In December 2019, Microsoft encountered a 5-day security incident in their databases. The primary reason for that incident was a modification in their security rule [5]. Facebook also experienced outages several times. One of the recent outages occurred in all of the Facebook platforms, Messenger, Instagram, WhatsApp, and Oculus in October 2021 because of configuration changes in the backbone routers [4].

Instances like these highly emphasize gaining confidence in the developed software system before it goes to the production servers. Proper documentation of changes with confidence about reasons and supporting evidence is crucial for handling such issues. Therefore, we need all the details about the inputs, methods, assumptions, and restrictions used to develop the system. The required information about the evidence is used to achieve the conclusion.

In 2016, Polacsek introduced the concept of a justification diagram to establish trust in a product [19]. This diagram relies on the Toulmin schema [20], an argumentation pattern, to instill confidence in the outcome. The main aim of the Justification diagram is to provide a diagrammatic representation to reach a convincing conclusion constructed with supporting information such as specific methods, sub-conclusion, and facts. The Justification Diagram is a decision makers to gain confidence with diagrammatic representation.

Given that configuration changes are commonplace across industries, internal DevOps pipelines also undergo continuous incremental improvements. Leveraging DevOps pipelines is beneficial for attaining project excellence. Therefore, it is important to maintain the pipeline's quality, including all comprehensive evidence and verification. The Justification Diagram is proven to track changes for the CI/CD pipeline by Corinne Pulgar [21]. This report focuses on enhancing the Justification Diagram to make it independent of the CI/CD platform with operational supports.

2.5.1 Symbols of a Justification Diagram

A Justification Diagram uses different notations, described in table 2.1, for the diagrammatic representation. The Justification Diagram consists of a component called `'Strategy'`, which serves as a method for achieving specific goals. Typically, `'Strategy'` depends on `'Evidence'`. The `'Evidence'` provides the necessary information to execute the method. After method execution, the system produces results, which are considered as `conclusions` or `sub-conclusions` in the Justification Diagram. In some cases, `restrictions` are needed to limit the conclusions or sub-conclusions, especially in sensitive systems.

Consider an example of Facebook. Facebook has proposed a novel network design to modernize its conventional backbone network infrastructure. Initially, they proposed utilizing the Interior Gateway Protocol (IGP) with a full-mesh configuration of the Interior Border Gateway Protocol (iBGP) to establish a foundation for basic packet routing. The Justification Diagram is easy to handle documentation for those changes.

Table 2.1: Components in Justification Diagram

Symbol	**Definition**
🟦	**Evidence**: Evidence is used to provide support for the strategy. It represents real data or evidence required to validate the given strategy.
▱	**Strategy**: The strategy is a symbol to represent a method which is used for reaching a satisfactory conclusion or sub-conclusion with the help of real evidence.
□	**Sub-Conclusion**: This symbol is associated with the strategy to attain various sub-conclusions.
▩	**Conclusion**: A conclusion, as the name suggests, is a final judgment or outcome justified by various components tailored to a specific problem.
→	**Relation**: A relation is used to connect various justification components, specifying their direction. For instance, if the relation is from evidence to strategy, it indicates that evidence is employed to support a specific strategy.
🟨	**Restriction**: Restriction adds limitations to the conclusion and is an optional component in the Justification Diagram.

As shown in figure 2.3, the initial version of Facebook's solution comprises two topologies for internal and external routing, achieved through the IGP and iBGP in their network design. Both protocols are integral components in the Justification Diagram and serves to facilitate basic packet routing. Typically, internal routing is managed by configuring internal routers and the associated protocols, while external routing entails configuring routers, assigning autonomous systems, and advertising routes. Supporting both internal and external routing are filtering and access policies, as well as monitoring, verification, and route maintenance. Once the setup for basic packet routing is complete, we can confirm that the system is operational by assessing packet routing within the network. Finally, the justification is an internal part of the organization, making it suitable for inclusion in the conclusion. Thus, it is straightforward to validate various changes with tangible evidence in the Justification

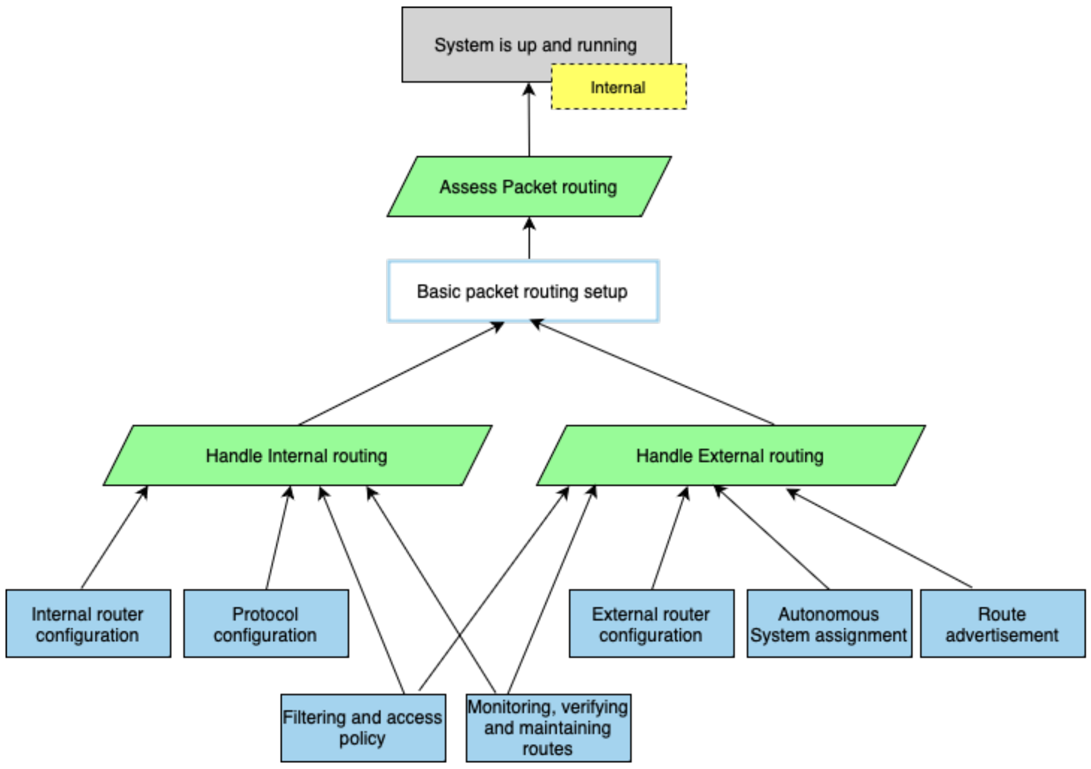

System is up and running
Internal
Assess Packet routing
Basic packet routing setup
Handle Internal routing
Handle External routing
Internal router configuration
Protocol configuration
External router configuration
Autonomous System assignment
Route advertisement
Filtering and access policy
Monitoring, verifying and maintaining routes

Figure 2.3: Facebook Network Justification diagram

Diagram.

2.6 Alternatives to Justification Diagram

There are several options available to convey meaningful expressions in the context of system updates. The first example involves using commits, which are plain text descriptions attached to updates in the GitHub workflow. However, relying on textual representation through commit text can make it more challenging to understand it directly. An alternative option is the Architectural Decision Records (ADRs), providing a short text description document that represents the high-level system decisions [2]. While ADRs offer a template, they primarily illustrate the system's decisions with implementation and decision information rather than more visual representations. For a more comprehensive and diagrammatic representation, we can turn to assurance cases, as suggested by the ISO standard on assurance cases [13]. Assurance cases offer a structured approach, incorporating claims, evidence, and explicit assumptions. To avoid assumptions for important decisions, we opt for the Justification Diagram, which provides a visual representation of the system along with the underlying justification for its accurate design decisions. Also, it is lighter than the assurance case as it often provides system-level abstraction instead of more detailed implementation. Ultimately, it makes it easier for stakeholders to understand decisions with their claims and supporting justifications without knowing implementation complexities.

Chapter 3

Case Study

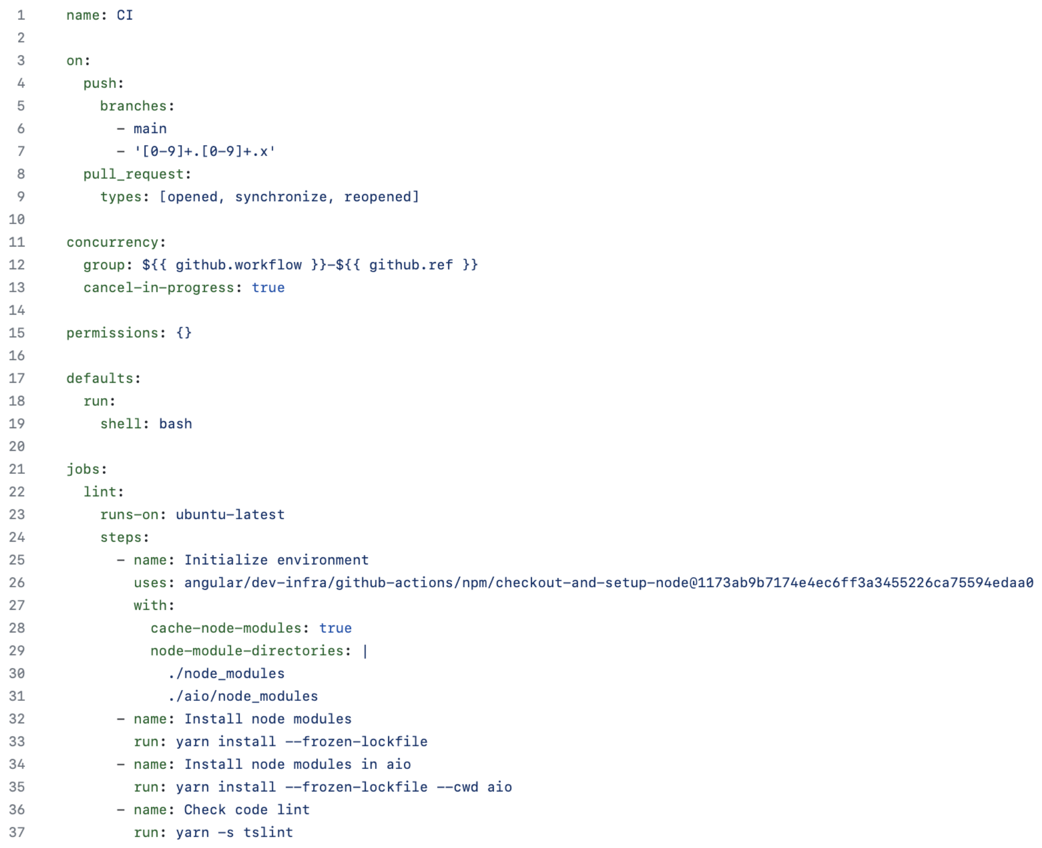
As discussed in Chapter 2, we use the Justification Diagram to track changes and tackle the challenges associated with updates. Specifically, we want to study and analyze how the Justification Diagram can contribute to the effectiveness of pipelines. Therefore, this chapter addresses **how semantics can be extracted for writing the Justification Diagram using the histories of pipelines.** Pipelines are an essential component in achieving successful software delivery, and they evolve. To document it through the Justification Diagram, I conducted this case study.

In this chapter, the initial section presents fundamental information about the GitHub workflow in Section 3.1 to understand the upcoming usage of GitHub workflows for justifications. The goal of this case study is discussed in Section 3.2, where the reasoning for this case study is defined and emphasized. Section 3.3 outlines the systematic approach of the study to illustrate the justifications from GitHub workflows. Section 3.4 and 3.5 discuss the observations and findings from the docker-compose pipeline. Section 3.6 offers details of the Justification Diagram written for the docker-compose pipeline and how those diagrams get altered when the GitHub pipeline changes (add/remove/update) their execution steps. Section 3.7 summarizes the conclusion of the case study.

3.1 Expressing pipeline with GitHub Workflow

The software industry has witnessed significant benefits of CI, CD, and CT strategies [25]. For that, various tools are available in the market which facilitate the implementation of these practices. Among all available tools, GitHub is one of the most popular code repository providers and stands out for its integration capabilities with a wide array of CI/CD tools. Also, GitHub offers a robust solution called GitHub Actions (GA), which allows developers to automate software development processes without third-party tools. Integration of GA is achievable by using the power of existing GitHub standard actions or creating custom actions required by specific project requirements [6].

The GitHub Workflow, usually written in YAML format used to automate various processes of software development. This YAML configuration file comprises several components. The YAML file starts by defining the workflow's name and is triggered on a branch by events such as code pushes to the repository or the creation of pull requests within the GitHub repository [26]. Within the workflow, individual jobs represent separate tasks, which can be executed either sequentially or in parallel based on requirements and resources. Each job consists of distinct steps, facilitating the combinations of job names, actions, and commands. GitHub offers a library of predefined actions that can be integrated to achieve specific tasks with required arguments. For example, the workflow incorporates advanced features like caching for efficient reuse of previously executed steps during subsequent runs and thus optimizing performance [12]. Moreover, we can also add permissions for files to limit access, environment variables for custom workflow values, and many other configurations to the GitHub workflow. An example of a sample GitHub workflow is depicted in figure 3.1 for Angular GitHub workflow.



```yaml
name: CI

on:
  push:
    branches:
      - main
      - '[0-9]+.[0-9]+.x'
  pull_request:
    types: [opened, synchronize, reopened]

concurrency:
  group: ${{ github.workflow }}-${{ github.ref }}
  cancel-in-progress: true

permissions: {}

defaults:
  run:
    shell: bash

jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - name: Initialize environment
        uses: angular/dev-infra/github-actions/npm/checkout-and-setup-node@1173ab9b7174e4ec6ff3a3455226ca75594edaa0
        with:
          cache-node-modules: true
          node-module-directories: |
            ./node_modules
            ./aio/node_modules
      - name: Install node modules
        run: yarn install --frozen-lockfile
      - name: Install node modules in aio
        run: yarn install --frozen-lockfile --cwd aio
      - name: Check code lint
        run: yarn -s tslint
```

Figure 3.1: Angular GitHub Workflow [3]

3.2 Goal of Case study

Integrating a GitHub workflow for the CI/CD pipeline with the repository is important for improving project quality in a DevOps context. As the code within the targeted repository undergoes regular updates, the YAML file for the workflow also evolves to meet new requirements for continuous integration and continuous delivery support in the project. Keeping track of these updates with each commit in GitHub is a crucial aspect of project management. The primary objective of this case study is to address these challenges through the Justification Diagram. Leveraging the components of the Justification Diagram detailed in Chapter 2, we can write a justification for each update in the CI pipeline. Furthermore, this case study aims to identify the problem statement addressed in this report.

3.3 Methodology of Case Study

This case study focuses on the CI pipeline for the GitHub workflow. There are different open-source GitHub projects that include workflow in their repository. We choose docker-compose for this case study because of these reasons:

- It has more than 80 commit histories on the workflow. : A high number of commits indicates active development, frequent updates, and ongoing maintenance. It is a positive metric for a responsive development team addressing issues, adding features, and maintaining code quality over time. Because of that, we can extract a lot of meaningful information from it.

- It goes through several changes. Frequent changes are a sign of a dynamic and responsive development process. Regular updates may indicate an agile development approach, adapting to evolving requirements, fixing bugs promptly, and incorporating user feedback effectively.

- It is a large community of users.

The process for this case study is structured into four distinct steps for the creation of the Justification Diagram, as outlined below:

- Step 1: Extract all relevant details regarding the changes committed to the docker-compose GitHub workflow.
- Step 2: Conduct a thorough comparison between each version and its predecessor (n-1 version) to get details into the updates and modifications made in each iteration.
- Step 3: Transform the identified changes into a meaningful explanation, encompassing details such as what was altered, its impact, and the reasons behind the modifications.
- Step 4: Represent the justification of the changes as a Justification Diagram.

For steps 1 and 2, Git commands (git diff) were used on our local machine to extract information from the `'ci.yaml'` and detailed analysis, respectively. In step 3, an in-depth analysis of the commit history was manually performed and documented in an Excel file (minimum version is in Appendix B). Finally, armed with these details, the Justification Diagram has been written in step 4.

3.4 Observation from evaluation

In Appendix B, our analysis focuses on a subset of 83 commits. We examined the first 50 commits for observation and analysis because they were enough for analyzing how the Justification Diagram can be written for tracking changes. To extract meaningful information from these 50 commits (that have the same intentions and conclusions), we needed to merge those 50 commits at some commits that resulted in 41 subsets of changesets. Instead of investing a lot of time in analyzing whole commit lists, we focused on the actual analysis of justification representations and their enhancements. We observed updates for `ci.yml` file from April 30, 2020, to November 4, 2021. During updates in this period, the GitHub workflow became consistent and mutual enough to execute the regular workflow.

3.5 Findings from observation

The analysis of Docker-compose gave a lot of insight into what happened. This section will describe the overall description of what changed and how the meaning got extracted. The aim of documenting these changes lead us to create JDs with those important details.

The first four commits in the project are used for enhancing the executability of the `‘ci.yml’` file. The process started with the execution of Makefile and manual installation of required dependencies. Subsequently, it evolved into the usage of a shell script for dependencies installation. This update aimed to establish a unified and efficient method for the installation procedures.

Subsequently, the 14 commits introduced various test cases on different platforms and updates for dependencies and GitHub actions. However, extracting the importance and meaningful context from these commits proved challenging due to insufficient information in the original commit messages. For instance, the gRPC end-to-end tests were added to the existing workflow, but the underlying intentions remained unclear. Additionally, the use of a make file to execute end-to-end tests lacked clarity and hindered a clear understanding of their purpose. Therefore, extracting the actual meaning of their changes was difficult. In addition, the absence of written comments in the YAML file contributes to the difficulty in understanding its content.

The subsequent 11 commits introduced updates to dependencies, build tags, and restrictions. However, some changes are significantly dissimilar from previous commits, making it challenging to extract information. Different build tags are used for configuring an environment that decides building and testing processes. The precise meanings of these build tags remained unidentified until a detailed analysis of folders was performed. Documenting this information could prove valuable for new members, providing clarity and context that would aid in understanding the purpose and utilization of these build tags.

In the last commits from analysis, the development efforts aimed to incorporate cross-platform builds, implement regular updates, and make changes to tests and build tags. Unfortunately, the clarity of their test and the build tag adjustments is unidentified due to a lack of additional information within the commit messages. The absence of explicit details makes it challenging to understand the purpose and outcomes of these modifications. One noticeable change is the removal of the example back-end from the build tag, suggesting a maturation of their system to accommodate diverse builds. However, no supporting evidence has been provided in the commit history. So, the extraction process for those commits was challenging.

Through our analysis and details, we realised that interpreting the meaning of changes can be challenging, especially for new team members or when dealing with an extensive and aging project history. In such cases, employing Justification Diagrams emerges as a beneficial approach. These diagrams serve as valuable tools by offering insights into the intentions behind detailed steps, thereby enhancing clarity and understanding for those engaging with the project.

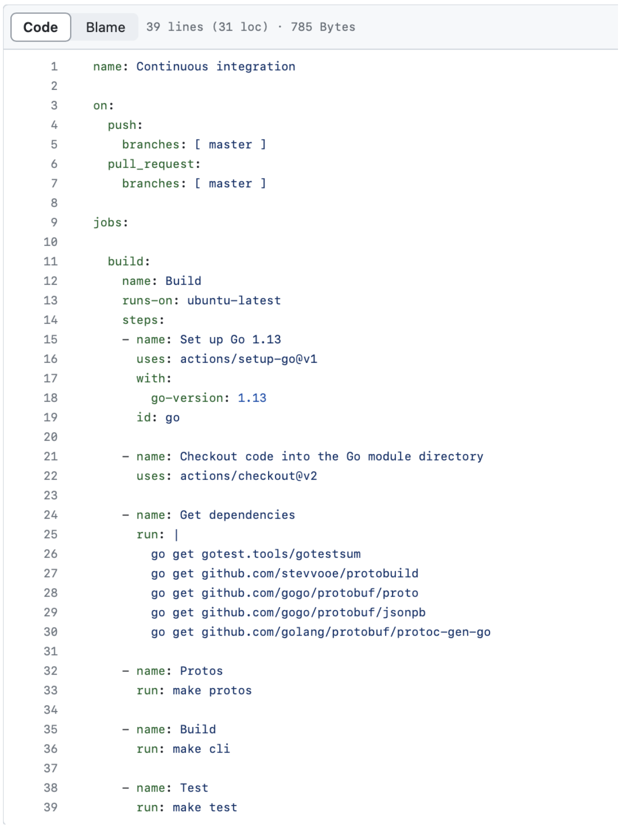
3.6 Justification for docker-compose workflow

A Justification Diagram is used to understand the implementation of evolutionary docker-compose workflow with details of a particular decision. Different branches in the Justification Diagram are updated by adding, removing, or modifying based on requirements in the GitHub workflow. The analysis shown in Appendix B includes information about the intentions behind changes and the semantic meaning of every change. This is essential for creating a Justification Diagram along with the strengths, weaknesses, and some improvements.

3.6.1 Initial Docker compose

Figure 3.2 shows the initial version of the Docker-compose pipeline. It includes build and test steps using a separate Makefile. Further, the workflow has steps for installing required dependencies for running other jobs and code checkout to enable the usage of code in other stages. The workflow triggered the push and pull request, especially on the master branch. Figure 3.3 illustrates the coding written for generating Justification Diagram (`docker_compose.jd`) and Figure 3.4 illustrates the Justification Diagram for the initial version of the workflow.

Justification modeled by writing a `‘jd’` file. `‘jPipe’` tool [14] is used to compile the justification file. The `‘docker_compose.jd’` file for the initial docker workflow is depicted in Figure 3.3, which consists of sub-conclusions, strategy, evidence, and conclusions. The first line of this file, represented as `‘JP’`, specifies the name of



```yaml
name: Continuous integration

on:
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]

jobs:

  build:
    name: Build
    runs-on: ubuntu-latest
    steps:
    - name: Set up Go 1.13
      uses: actions/setup-go@v1
      with:
        go-version: 1.13
      id: go

    - name: Checkout code into the Go module directory
      uses: actions/checkout@v2

    - name: Get dependencies
      run: |
        go get gotest.tools/gotestsum
        go get github.com/stevvooe/protobuild
        go get github.com/gogo/protobuf/proto
        go get github.com/gogo/protobuf/jsonpb
        go get github.com/golang/protobuf/protoc-gen-go

    - name: Protos
      run: make protos

    - name: Build
      run: make cli

    - name: Test
      run: make test
```

Figure 3.2: Initial version of docker-compose pipeline [9]

the justification. Also, each component is assigned a unique name to facilitate its own identity so it can link with other components. The ‘`supports`’ keyword is employed with two arguments, ‘`from`’ and ‘`to`’, stating the direction of the link and enabling the creation of connections between them. Lastly, the `"is"` keyword is used to assign the string text, usually written in double quotes, to the specific justification components information. The whole component list is written in between {}.

Two main strategies are employed to achieve the ultimate goal of `"Continuous Integration Validation"`: running all test cases and scheduling workflows. The scheduling workflow strategy is an essential part of any GitHub workflow because it enables the setup of workflows based on specific events, providing value for evidence

```
justification pattern JP {
    sub-conclusion ASu1 is "Project build"
    strategy St0 is "Verify building"
    evidence Su1 is "Checked out code in directory"
    Su1 supports St0
    St0 supports ASu1

    sub-conclusion ASu2 is "Reusable components"
    strategy St1 is "Reuse already developed packages"
    evidence Su2 is "Packages are public"
    Su2 supports St1
    St1 supports ASu2
    ASu2 supports St0

    sub-conclusion ASu3 is "Dependencies installed"
    strategy St2 is "Dependencies Management"
    evidence Su3 is "Commands"
    Su3 supports St2
    St2 supports ASu3
    ASu3 supports St0

    sub-conclusion ASu4 is "Software tested"
    strategy St3 is "Run test cases"
    evidence Su4 is "Test cases"
    Su4 supports St3
    St3 supports ASu4
    ASu3 supports St3

    strategy St4 is "Verify Functionality of workflow"
    ASu4 supports St4
    ASu1 supports St4

    strategy St5 is "Schedule workflow"
    evidence Su5 is "event"
    Su5 supports St5

    conclusion C is "Continuous Integration Validated"
    St4 supports C
    St5 supports C
}
```

Figure 3.3: Justification Diagram code for the initial version of Docker compose

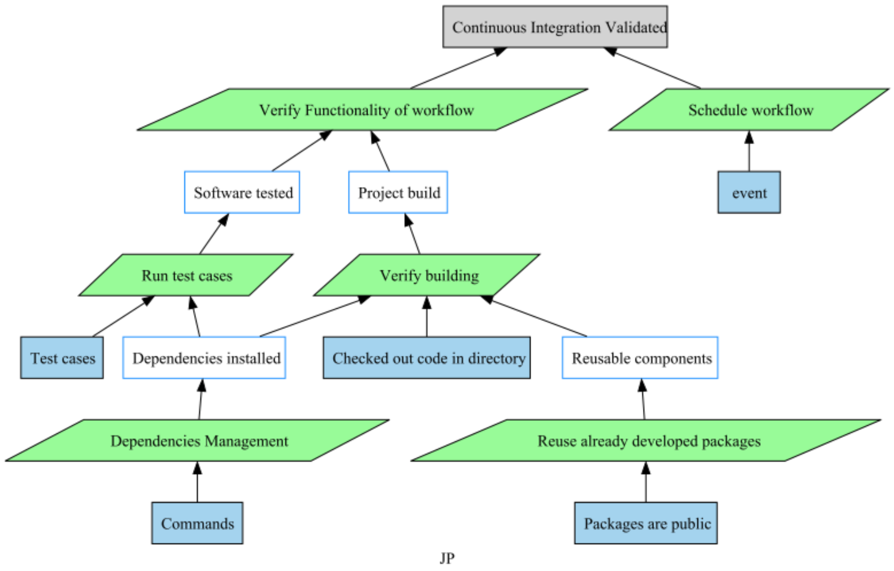
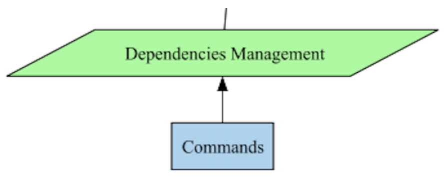


Figure 3.4: Justification Diagram for initial version of Docker compose

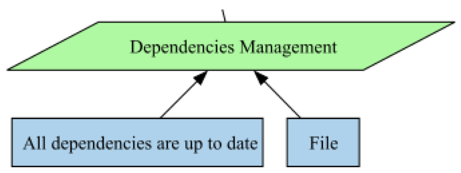
in the Justification Diagram. The verification of workflow functionality is sub-divided into two key components: project build and software testing. Both components must succeed to validate the workflow's functionality. The project build relies on source code, required dependencies, and reusable components, such as existing GitHub actions. It is imperative to ensure the project's successful build. Software testing includes the other half of the workflow that requires installed dependencies and test cases. Referring to Figure 3.4, the installation of dependencies is supported by commands. Reused components serve as a sub-conclusion with the publicly available GitHub packages. The dependencies are shared between the building and testing stages, highlighting their importance in the workflow. JP at the bottom represents the name of the Justification Diagram.

3.6.2 First change in Docker compose (Appendix B - 001)

In the next version, the workflow has been replaced by the execution of commands through an existing file. In other words, a shell script is now employed to manage the installation of dependencies. You can observe this updated workflow in Figure 3.5, which introduces a modified branch for the Dependencies Management Strategy. The primary requirement to acquire dependencies is that all the listed dependencies must remain up to date with the project. A shell script, serving as a file, is utilized to streamline the installation of dependencies for various stages within the workflow. Both of these changes have been incorporated as new branches in the Justification Diagram.



(a) Initial version



(b) Updated version

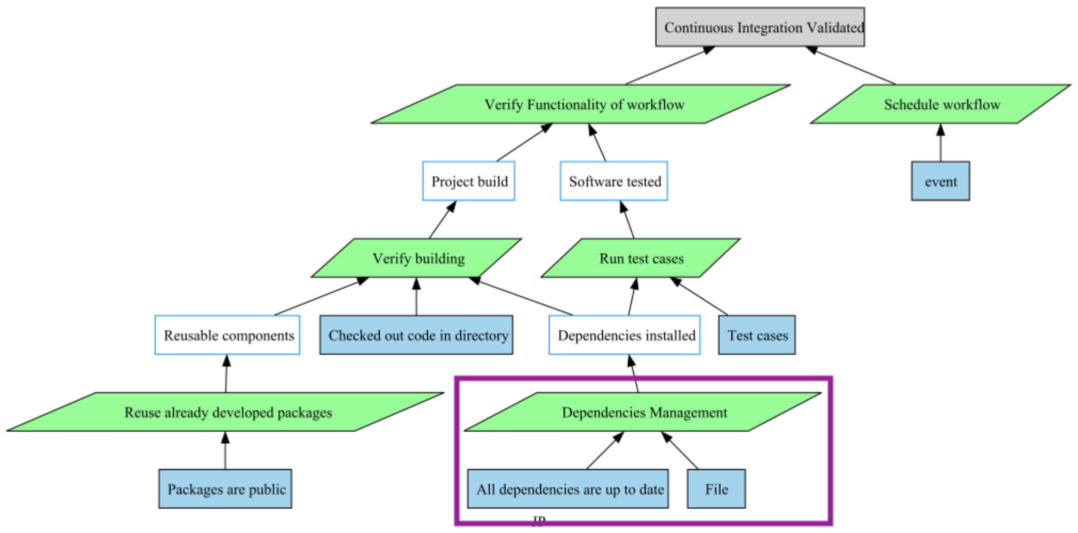
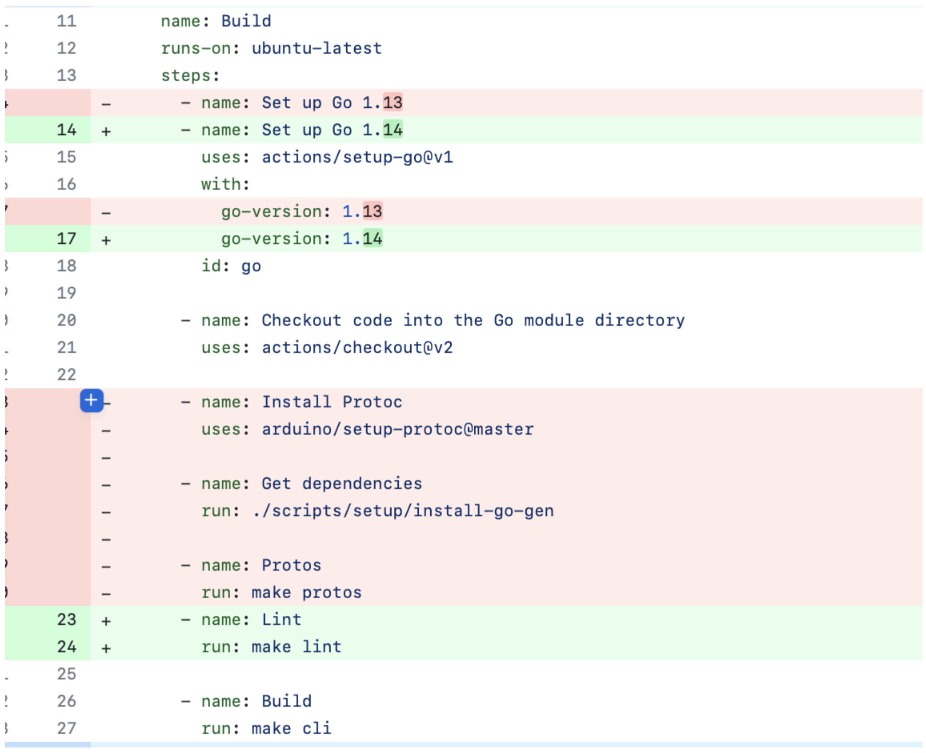


Figure 3.5: Justification Diagram for Docker compose version 2

3.6.3 Second change in Docker compose (Appendix B - 003)

In subset 003, docker-compose introduced a new feature by incorporating a linter in their workflow. As depicted in Figure 3.6, first, they removed the dependencies installation step and made it an internal process using a Dockerfile. But still, Go installation is there for dependencies. Also, they have added a linting operation through the Makefile. Our analysis in Appendix B - 003 provides more details about the changes and their intentions. It states that the linter is added to incorporate the validation in coding standards.



```
    name: Build
    runs-on: ubuntu-latest
    steps:
-     - name: Set up Go 1.13
+     - name: Set up Go 1.14
        uses: actions/setup-go@v1
        with:
-         go-version: 1.13
+         go-version: 1.14
        id: go

      - name: Checkout code into the Go module directory
        uses: actions/checkout@v2

-     - name: Install Protoc
-       uses: arduino/setup-protoc@master
-
-     - name: Get dependencies
-       run: ./scripts/setup/install-go-gen
-
-     - name: Protos
-       run: make protos
+     - name: Lint
+       run: make lint

      - name: Build
        run: make cli
```

Figure 3.6: Linter added to docker-compose

As depicted in Figure 3.7, code quality has been introduced as a new branch for supporting linting operations. They have integrated a linter to conduct code quality checks that can be considered a sub-conclusion for JD. To achieve this sub-conclusion,

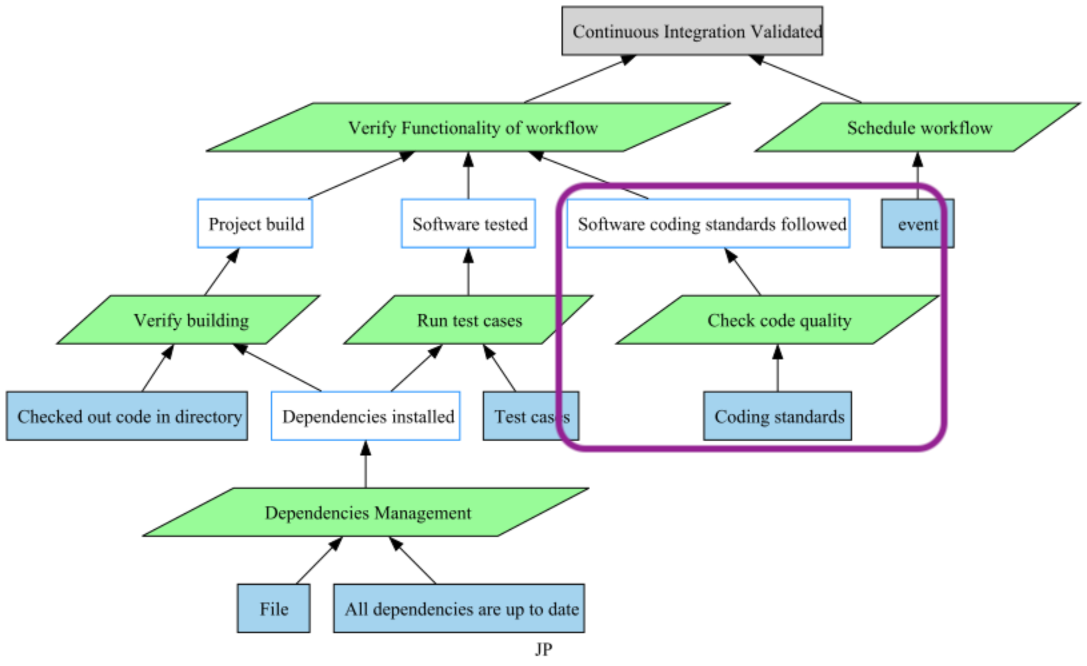


Figure 3.7: Justification Diagram for Linter added to docker-compose

we need to perform code quality checks. The strategy is `"Check code quality"` used with the sub-conclusion. The evidence for the coding standards mentioned in the linter is attached to the strategy. Also, for the dependencies removal, they are still using Go dependencies, which is valid in the Justification Diagram. Therefore, we can affirm that this method of justifying significant changes in the CI Workflow makes it easier to understand the intentions behind these changes.

3.7 Conclusion

Among those 50 commits (41 subsets), we selected the first commits for further processes in composing the Justification Diagram because they were initial commits to help identify and understand root causes, trends, and changes in the Docker-compose

GitHub workflow. It can help to provide upcoming changes from the initial versions.

Based on the above discussion, it is concluded that Justification is valuable for showing and explaining the intentions behind changes made for GitHub workflows. It clearly identifies whether the changes attach to a new branch (new subset added to JD) or an update in an existing branch (attached to the existing JD) of the Justification Diagram. By using these features of JD, we can say that JD is applicable in tracking pipeline changes, and we can study in detail the approach of JD.

Chapter 4

Platform Independent Justifications

In the previous chapter, we gained confidence in using the Justification Diagram to track the changes in the GitHub pipeline. We intend to extend the usage of JD to other platforms. This chapter addresses the question **How can we extend the usage of the Justification diagram for different pipeline platforms?** As shown in figure 4.1, the primary objective is to improve the usability of the Justification Diagram in various pipeline platforms. This chapter provides examples from GitLab and Azure pipelines, expanding the discussion to illustrate the applicability of the Justification Diagram across different platforms. While the initial section (Section 4.1) provides details on the utilization of Justification for GitLab pipelines, the subsequent section (Section 4.2) demonstrates the application of the Justification Diagram within Azure pipelines.

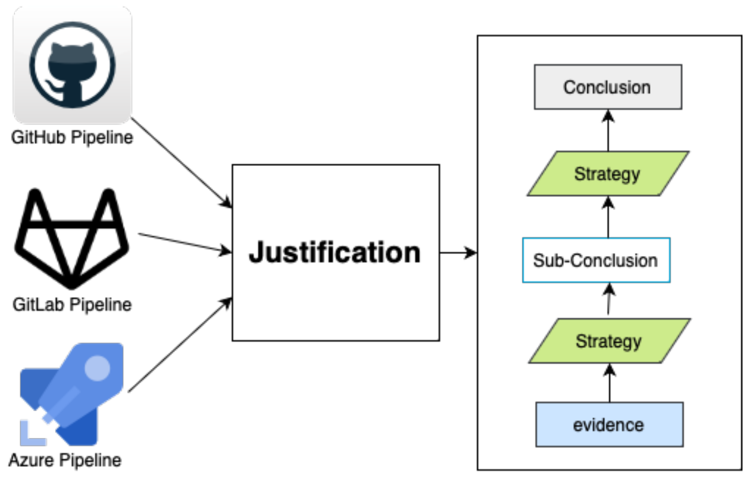


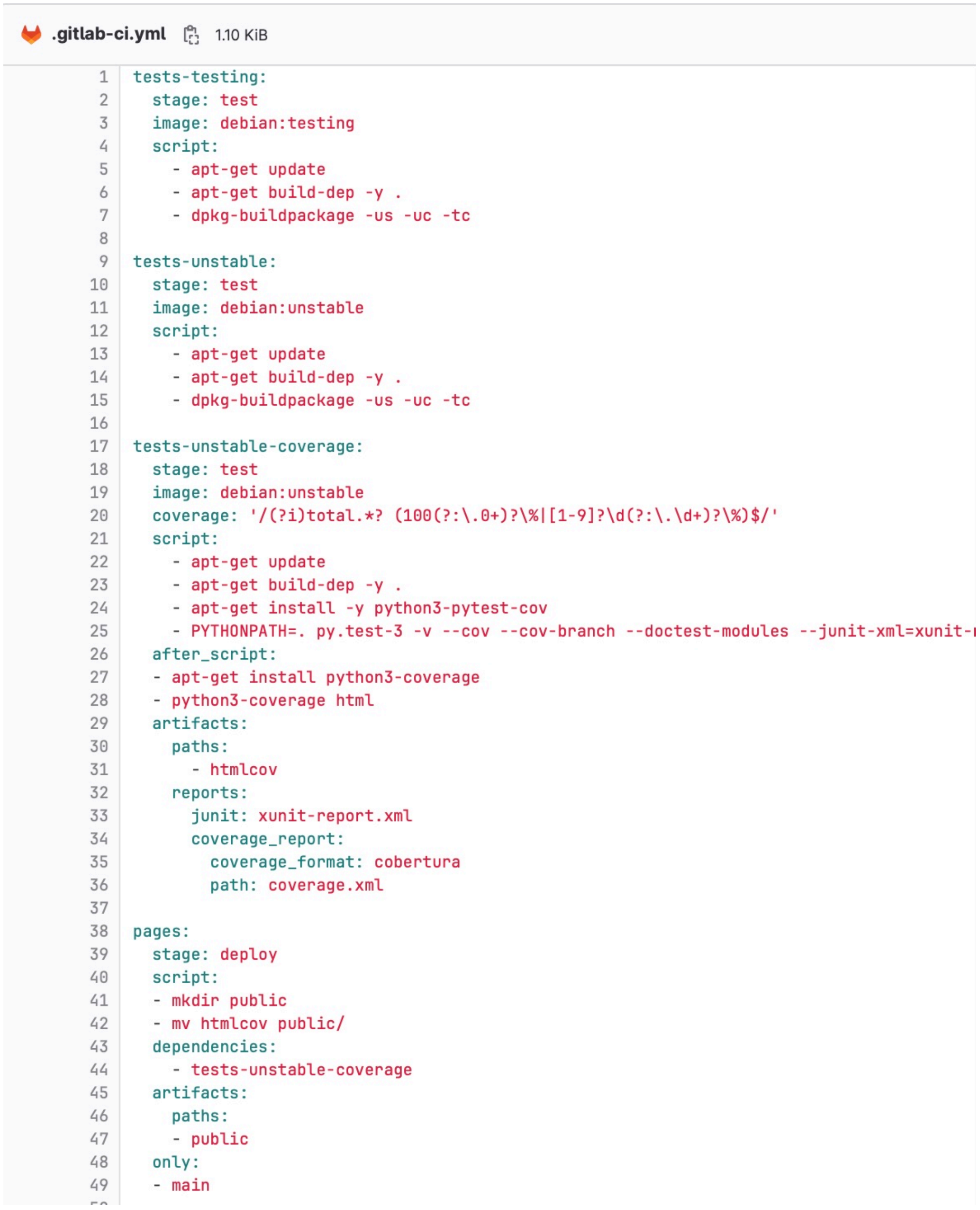
Figure 4.1: Platform Independent Justification

4.1 Justification Diagram with GitLab

GitLab is a Git repository management system that serves as a robust platform for software project management. It integrates code review, CI/CD, and the creation of wikis in their platform to make it more usable [23]. It gained popularity by its services to reach second online hosting services [23]. Different projects in the GitLab repository also include the workflow for achieving continuity in software processes.

Figure 4.2 shows a sample GitLab pipeline example, providing a practical example for defining a Justification Diagram for GitLab workflow. The workflow in the example is executing various testing and deployment stages. It's challenging to categorize which part performs what and how important they are. For future updates, it will be necessary to have those documented. Therefore, justification offers a straightforward and easy-to-understand visual representation for GitLab workflows.

The Justification Diagram is represented in Figure 4.3. The two main strategies, ‘`Perform Testing`’ and ‘`Deployment`’, are extracted from the GitLab example directly. The sequential successful execution of these strategies is required for obtaining



```yaml
tests-testing:
  stage: test
  image: debian:testing
  script:
    - apt-get update
    - apt-get build-dep -y .
    - dpkg-buildpackage -us -uc -tc

tests-unstable:
  stage: test
  image: debian:unstable
  script:
    - apt-get update
    - apt-get build-dep -y .
    - dpkg-buildpackage -us -uc -tc

tests-unstable-coverage:
  stage: test
  image: debian:unstable
  coverage: '/(?i)total.*? (100(?:\.0+)?\%|[1-9]?\d(?:\.\d+)?\%)$/'
  script:
    - apt-get update
    - apt-get build-dep -y .
    - apt-get install -y python3-pytest-cov
    - PYTHONPATH=. py.test-3 -v --cov --cov-branch --doctest-modules --junit-xml=xunit-
  after_script:
  - apt-get install python3-coverage
  - python3-coverage html
  artifacts:
    paths:
      - htmlcov
    reports:
      junit: xunit-report.xml
      coverage_report:
        coverage_format: cobertura
        path: coverage.xml

pages:
  stage: deploy
  script:
  - mkdir public
  - mv htmlcov public/
  dependencies:
    - tests-unstable-coverage
  artifacts:
    paths:
    - public
  only:
  - main
```

Figure 4.2: Debian GitLab example [8]

a conclusion called ‘`pipeline validation`’. For two tests (`tests-testing` and `tests-unstable`) shown in figure 4.2, we merged them to the single test strategy as the intentions behind them are the same. The ’`execution environments`" is evidence for testing, which can be any base images on which the test cases will run. For the test and deployment of the project, required dependencies are important evidence. For the deployment strategy, the evidence includes the fulfillment of requirements. Deployment only occurs when these requirements are satisfied for the configurations and builds, ensuring a robust and validated pipeline based on given conditions. In addition, test coverage has been translated as strategy, supported by evidence like a `coverage plan, execution environment considerations,` and `ensuring the utilization` of updated dependencies. The coverage plan covers the coverage statements with the artifact information, and the execution environment is limited for the base image and after-execution statements.

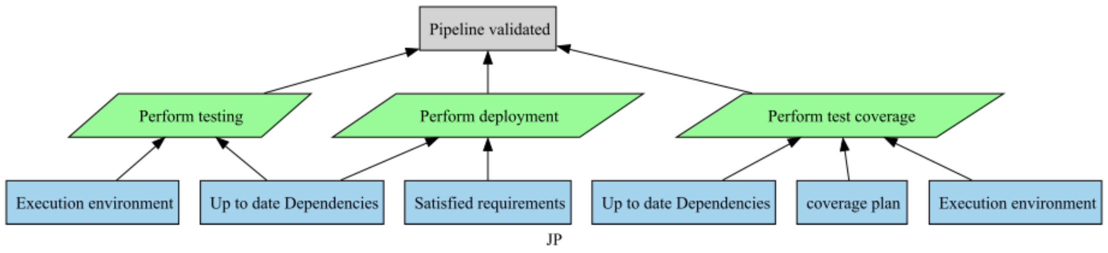
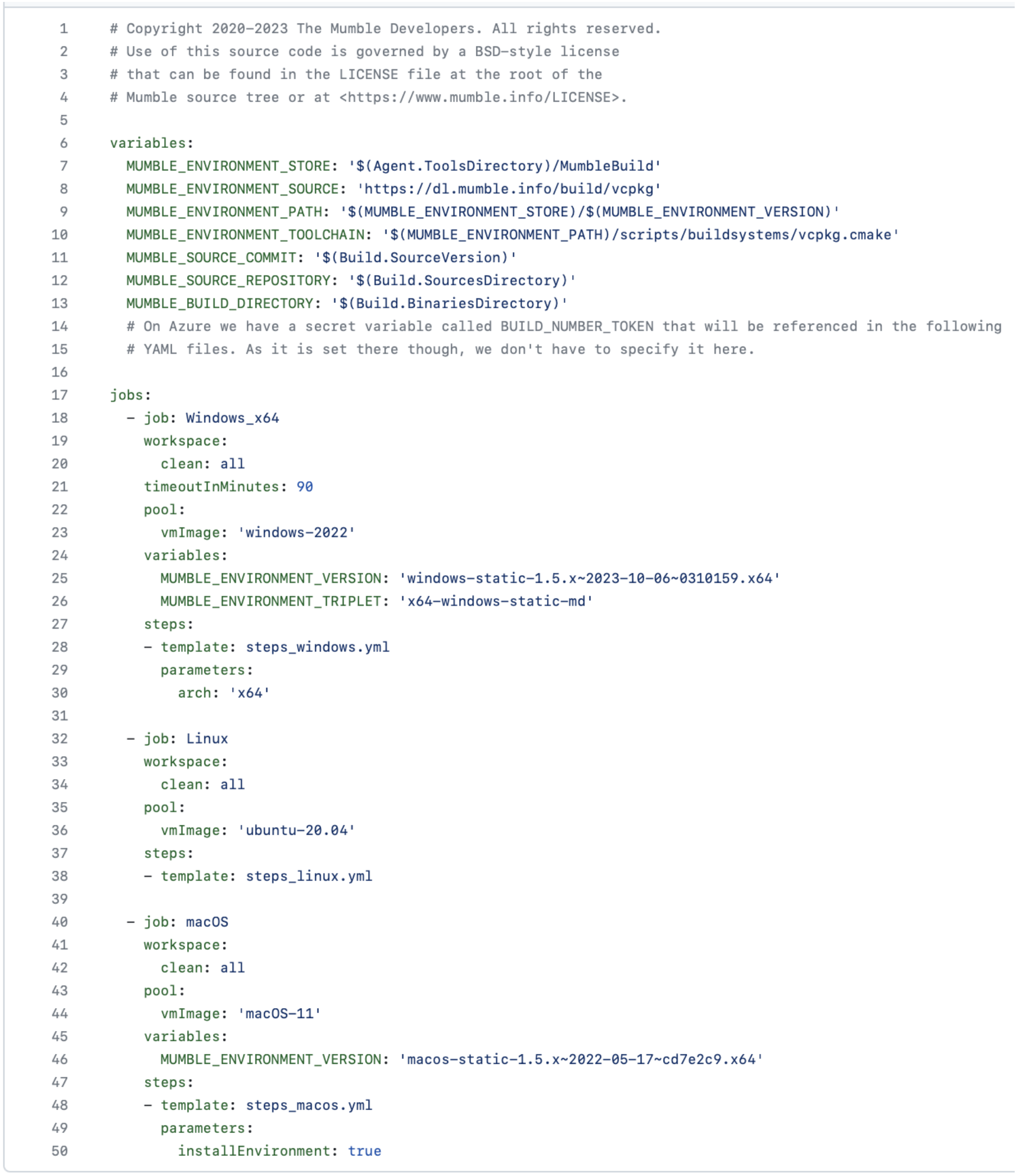


Figure 4.3: GitLab Justification example

After conducting analyses across GitLab projects, it becomes apparent that the creation of a Justification Diagram is not only feasible but also adaptable for any pipeline existing within the GitLab platform.

4.2 Justification with Azure

Microsoft introduced the Azure DevOps pipeline [15], a script-based robust, cross-platform, and powerful automation engine [28]. Azure Pipelines enables CI/CD and Continuous Testing (CT) within open-source code repositories. The Azure pipelines are also written in YAML format, which has some common naming conventions as GitHub workflow has.



```yaml
# Copyright 2020-2023 The Mumble Developers. All rights reserved.
# Use of this source code is governed by a BSD-style license
# that can be found in the LICENSE file at the root of the
# Mumble source tree or at <https://www.mumble.info/LICENSE>.

variables:
  MUMBLE_ENVIRONMENT_STORE: '$(Agent.ToolsDirectory)/MumbleBuild'
  MUMBLE_ENVIRONMENT_SOURCE: 'https://dl.mumble.info/build/vcpkg'
  MUMBLE_ENVIRONMENT_PATH: '$(MUMBLE_ENVIRONMENT_STORE)/$(MUMBLE_ENVIRONMENT_VERSION)'
  MUMBLE_ENVIRONMENT_TOOLCHAIN: '$(MUMBLE_ENVIRONMENT_PATH)/scripts/buildsystems/vcpkg.cmake'
  MUMBLE_SOURCE_COMMIT: '$(Build.SourceVersion)'
  MUMBLE_SOURCE_REPOSITORY: '$(Build.SourcesDirectory)'
  MUMBLE_BUILD_DIRECTORY: '$(Build.BinariesDirectory)'
  # On Azure we have a secret variable called BUILD_NUMBER_TOKEN that will be referenced in the following
  # YAML files. As it is set there though, we don't have to specify it here.

jobs:
  - job: Windows_x64
    workspace:
      clean: all
    timeoutInMinutes: 90
    pool:
      vmImage: 'windows-2022'
    variables:
      MUMBLE_ENVIRONMENT_VERSION: 'windows-static-1.5.x~2023-10-06~0310159.x64'
      MUMBLE_ENVIRONMENT_TRIPLET: 'x64-windows-static-md'
    steps:
    - template: steps_windows.yml
      parameters:
        arch: 'x64'

  - job: Linux
    workspace:
      clean: all
    pool:
      vmImage: 'ubuntu-20.04'
    steps:
    - template: steps_linux.yml

  - job: macOS
    workspace:
      clean: all
    pool:
      vmImage: 'macOS-11'
    variables:
      MUMBLE_ENVIRONMENT_VERSION: 'macos-static-1.5.x~2022-05-17~cd7e2c9.x64'
    steps:
    - template: steps_macos.yml
      parameters:
        installEnvironment: true
```

Figure 4.4: Mumble Azure example [16]

Figure 4.4 shows a sample code of the pipeline for the "Mumble" [16], an open-source voice-chat GitHub project. It requires some variables to be defined and initialized at the beginning of the pipeline and referenced in different jobs for their successful execution. There are three jobs: Windows, Linux, and macOS. The first job is to build a job in the Windows server, while the subsequent two jobs are for performing build on Linux and macOS operating systems. All three jobs have separate files for their executions. Those files include the operating system-based dependencies installation, script execution for build and test, and the artifact publications.

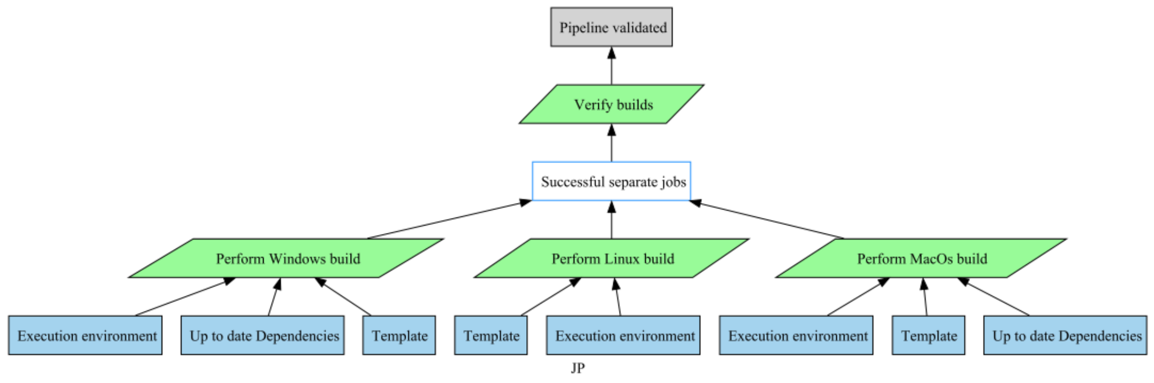


Figure 4.5: Azure pipeline example

The Justification Diagram depicted in Figure 4.5 outlines the rationale behind this process. The figure is divided into three distinct strategies for their builds: Windows, Linux, and macOS. The initial strategy involves executing a build on a Windows server, necessitating specific execution environments, dependencies, and templates. The template is referenced as an external file for the build process, while the execution environment specifies the 'vmImage' as a base environment to run the builds on it. Additionally, the same build procedure is performed on macOS. However, the Linux build does not mandate the installation of dependencies. These strategies collectively contribute to the sub-conclusion of `‘Successful separate jobs’`. The entire build process is validated, culminating in the conclusion of `‘pipeline validated’`. Here, the dependencies installation is separated because the dependencies depend on the platform on which the build execution is performed. Also, the execution environment are different. The templates for all strategies can be extended to add more detailed information. For the given `main.yaml` file, we keep it limited to the template.

4.3 Conclusion

With the analysis performed on two different platforms, we claim that the Justification Diagram is effectively used to represent the rationales behind the GitLab and Azure DevOps pipelines. With this consistency of the Justification Diagram, we are confident that we can now utilize the power to extend the Justification Diagram usage for further researching.

Chapter 5

Towards Operational Justifications

In the previous chapters, we discussed how to track changes with their intentions using the Justification Diagram and how it is used in GitHub, GitLab, and Azure pipelines. This chapter introduces the operational justifications along with the Justification Diagram. First, we need to understand the problem and its relevance to the case study in Section 5.1. Next, we proposed a solution in Section 5.2, which identifies and describes the methodology and extended analysis on different platforms. The validation of the proposed solution is observed in Section 5.3 with examples, and Section 5.4 includes the conclusion of operational justifications.

5.1 Problem statement

The Justification Diagram provides a clear trace of modifications, enhances communication throughout the development process, and simplifies risk management of pipelines. While this technique has brought positive changes, it has also introduced some real-world challenges to the developers and stakeholders.

Firstly, the diagram with the GitHub workflow may occasionally result in mismatched information. Developers forget to propagate the changes into the Justification Diagram after manually updating in the GitHub workflow. Also, multiple team members working on the same GitHub workflow can occasionally result in a mismatch Justification Diagram. Multiple branches in GitHub can also increase discrepancies between different GitHub workflow Justifications, so the information is mismatched because of a lack of synchronization information between Justification and GitHub.

Secondly, as it's a manual job for developers to manage the Justification Diagram with the GitHub repository, it also increases the workload of the developer, just doubling the same work. Due to its meticulous nature, the synchronous task between both can result in a waste of resources and the developer's efficiency.

Lastly, the Justification serves the purpose of documenting changes with the extra work of GitHub workflow rather than providing any actual implementation details that impact the real-world software development processes. Also, implementing real-world solutions is separated from the Justification Diagram. Based on these, documentation should provide references to the implementation that can also be useful for validating the Justification Diagram.

As discussed above, Justification written with a regular GitHub workflow can add extra burden to the developers and pipeline. The next section gives an in-depth analysis of these challenges, supported by practical examples.

5.2 Proposed solution

To address the above problems, we need to write Operational Justification Diagrams. It is a new terminology for justifications coined by Jean-Michel Bruel. It aims to add simple implementation information attached to the Justification Diagram that ultimately satisfies the large gap between the Justification Diagram and GitHub workflow by making them a combined solution for the pipeline. In the upcoming sections, we will understand how we create operational justifications and examples to showcase solutions that effectively mitigate the identified issues.

5.2.1 Methodology

This research follows a bottom-up approach to investigate the key attributes of the implementation details from the GitHub workflow file that can eventually make the Justification Diagram operational. The process of extracting information from multiple pipelines is as follows:

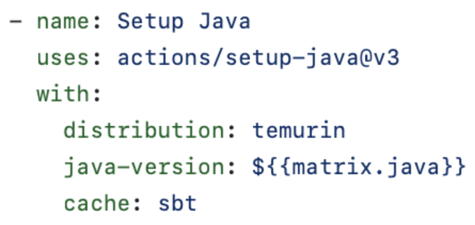
- Step 1: Extract the semantic information from the existing GitHub workflow to understand the behavior and mechanism of the job.
- Step 2: Compare the identified job structure with other open-source pipelines to find commonalities and variability of using the same patterns.
- Step 3: The translation process tries to encapsulate each operation, ensuring its relevance within the broader context of the workflow.

To extend our operational knowledge, we tried to cover the different pipelines from GitLab and Azure. We start with how their syntaxes are written and used to make a Justification Diagram unrestricted for their pipelines. This enhancement enhances the robustness of our operational justifications and broadens the scope of our understanding by incorporating diverse and industry-relevant perspectives. The following four sections give more details about all three steps.

5.2.2 Pipeline details extraction

The first step is to extract job information from the GitHub pipeline. For this study, a specific open-source GitHub pipeline job is selected to understand the detailed behavior and its semantic meaning in the broader context of the pipeline. This focused approach aims to analyze the complex structure of the particular job, examining its execution, variables, dependencies, and interactions with other stages in the pipeline.

The example job shown in Figure 5.1 represents a segment of the GitHub pipeline. Named `"Setup Java"`, serves as the identifier for a particular job and is displayed on the action when the pipeline is triggered. The job utilizes GitHub actions to execute the Java installation process, which targets version 3. Moreover, it uses additional parameters such as the distribution value, Java version, and caching parameters.



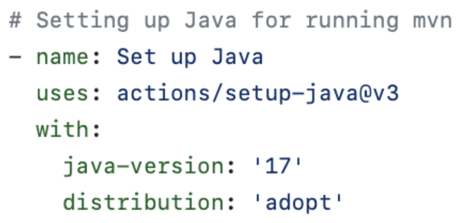
```
- name: Setup Java
  uses: actions/setup-java@v3
  with:
    distribution: temurin
    java-version: ${{matrix.java}}
    cache: sbt
```

Figure 5.1: Setup Java in Scala pipeline [24]

The key information extracted here is that for setting up Java within a workflow file. We can leverage GitHub actions with specific parameters based on the project's needs. It allows us a customized configuration, ensuring an efficient setup of Java within the workflow, so we can use this information to compare it with other GitHub pipelines that use the same dependencies installation in their project with different structures.

5.2.3 Comparison of jobs

Once we understand the meaning of the job, the next step is to compare different pipelines with the same semantic job. By analyzing various parameters and commonalities across these pipelines, such as configuration settings, dependencies, execution steps, and integration patterns, the research seeks to identify shared structural elements and distinctive characteristics.



```yaml
# Setting up Java for running mvn
- name: Set up Java
  uses: actions/setup-java@v3
  with:
    java-version: '17'
    distribution: 'adopt'
```

Figure 5.2: Setup Java in jPipe pipeline [14]

Figure 5.2 shows the Java setup within the jPipe pipeline, accepting parameters for ‘`java-version`’ and ‘`distribution`’. Having reviewed both Figure 5.1 and Figure 5.2, the subsequent step involves comparing and contrasting the information extracted from each. Both examples necessitate a ‘`name`’ and employ the GitHub action ‘`actions/setup-java@v3`’. However, some parameters are different for both. For instance, the previous figure utilizes ‘`distribution: temurin`, whereas the next one opts for ‘`adopt`’. Additionally, variations in the ‘`java-version`’ parameter with caching choices describe differences between the two. Conclusively, both configurations employ the same parameters with different values to understand their similarities and distinctions.

5.2.4 Translation of jobs

The last stage is to convert this analysis into translated jobs. By encapsulating the essence of each operation, the translation process facilitates operations that can be useful with the justification to have meaningful impacts that convey the same meaning as an actual pipeline job.

The next phase involves translating identified commonalities from previous steps into distinct operations. One such operation can be `‘setup_java()’`, designed to facilitate the Java setup process. This operation can be flexibly used, which provides optional parameters like `‘java-version’`, `‘distribution’`, and `‘cache’`. By consolidating these shared attributes into a singular operation, the workflow achieves modularity and flexibility, enabling streamlined and customizable setups for Java across diverse project requirements.

5.2.5 Extended Analysis

To enhance the usage of the proposed solution, an analysis was undertaken to identify and integrate it with other platforms such as GitLab and Azure. The comparison is structured into two distinct categories, each focusing on different aspects of the job behavior. The first category involves an analysis of entire job runs based on the semantics of the job. The second category revolves around analyzing syntaxes, such as `‘run’` in GitHub workflow for manual commands. Further elaboration on both categories is detailed in Table 5.1 and Table 5.2, respectively.

In Table 5.1, some job-oriented details are analyzed. For instance, the first example involves enabling the caching option with different parameters. Upon identifying options available across all three platforms, an operation named `‘add_cache()’` can

Table 5.1: Different Platform analysis based on Jobs

No.	GitHub	GitLab	Azure
1	actions/cache	cache	cache
2	actions/upload-artifact	artifacts	publishPipelineArtifact
3	run: make	make	make
4	actions/setup-java	apt-get install -y openjdk-11-jdk	useJavaVersion
5	actions/checkout	-	checkout: self
6	actions/setup-go	golang:latest	go

be formulated within the Justification framework. The second example used for uploading artifacts to specific locations is a functionality that can be encapsulated within the '`upload_artifact()`' operation, utilizing '`path`' as a parameter. The '`run_make()`' operation can be created to execute specific '`make`' commands, accepting commands as arguments. Additionally, the setup of Java discussed earlier can use the '`setup_java()`' command. Furthermore, the code checkout operation can be performed using '`checkout()`', while the Go installation can function with '`setup_go()`'.

Table 5.2: Different Platform analysis based on syntaxes

No.	GitHub	GitLab	Azure
1	run	script	script
2	if	rules: if	condition
3	needs	needs	dependsOn
4	env	variables	variables
5	strategy: matrix	matrix	strategy: matrix
6	timout-minutes	timeout	timeoutInMinutes
7	name	workflow: name	name
8	runs-on	image	vmImage
9	strategy: max-parallel	parallel	strategy: maxParallel

In Table 5.2, an examination of syntaxes necessary for custom user inputs in the pipeline file is given. The first example, utilized to execute various actions and commands, can be effectively addressed operationally through ‘`execute_runner()`’. The second example introduces workflow conditions written as ‘`set_condition()`’ with the specific condition as an argument. To meet job requirements, establish job-specific environments, and incorporate multi-platform requirements, ‘`depends_on()`’, ‘`set_environment()`’, and ‘`cross_platform()`’ operations are applied, respectively. For defining the maximum time allowance for a job, ‘`maximum_time()`’ can be used with time as a parameter. The naming of workflows is accomplished by the ‘`workflow_name()`’ operation, where ‘`name`’ can be passed as a parameter. Ensuring the appropriate base operating system for workflow execution is achieved with ‘`set_machine()`’. Finally, specifying the maximum parallel jobs for execution is facilitated by ‘`limit_parallel_execution()`’.

Through this analysis, we've identified essential commonalities in the behavior of pipelines across GitHub, GitLab, and Azure. As a result, these operations can be effectively utilized for the Operational Justification Diagram. Some of the operations are listed in table 5.3 with their meaning.

5.3 Validation

With the identified operations and their corresponding behavioral insights, we now possess all the fundamental components for constructing a comprehensive operational Justification Diagram. These operations, each playing a distinct role in the pipelines across different platforms, will be systematically integrated into the Justification Diagram. The upcoming sections will explain the file structure and examples of leveraging these components to write operational justifications.

Table 5.3: Different Operations

No.	Operation	Meaning
1	add_cache()	adding caching with parameters.
2	upload_artifact()	uploading artifacts to specific locations
3	run_make()	run make file.
4	setup_java()	setting up Java environment with specific arguments.
5	checkout_repository()	code checkout operation.
6	setup_go()	setting up go dependencies in project.
7	execute_runner()	execute various actions and commands.
8	depends_on()	add dependencies of jobs.
9	set_environment()	setting up environment variables used throughout the execution.
10	maximum_time()	setting up maximum time allowance to run job.
11	workflow_name()	setup name of the workflow.
12	set_machine()	assign base operating system for execution.
13	limit_parallel_execution()	limiting maximum parallel jobs for execution.
14	verify_dependencies()	verify the existence of dependencies.
15	verify_branch()	verify the existence of branch.
16	on()	assign branch and push or pull request for whole workflow.
17	verify_repository()	verify the existence of repository.
18	run_mvn()	run specific mvn commands.

5.3.1 Operational File Format

This section explains the justification file usually written in `'*.jd'` format, offering deeper insights into the utilization of operations. This file consists of five syntaxes.

1. **load**: This initial instruction is the gateway to import a specific Justification file onto which operations are added. It provides a separation of both justification and operational files.

2. **implementation**: The `'implementation'` keyword is used to describe the name of the implementation along with its specific justification.

3. **implements**: Typically used within the `‘implementation’` section, this keyword establishes a link between the Justification file and its corresponding operation by sharing the same name.

4. **probe**: The `‘probe’` keyword is deployed to verify specific conditions, offering a comprehensive evaluation beyond a single operation. Essentially, it aims to validate job-specific requirements.

5. **operation**: This keyword is used to utilize the above-mentioned operations, often associated with the Strategy or Evidence.

6. **expectation**: Employed to retrieve expected values either from the `‘probe’` or `‘operation’`, `‘expectation’` ensures that the written components adhere to the expected behavior, allowing for systematic checks.

5.3.2 Example

We can use Appendix A, which provides a detailed description of the example pipeline. From that detailed information, we can write the Justification Diagram that can be used for the next step. The operations used with these examples are already described in Table 5.3.

Firstly, the operation file is written for the `‘Trigger workflow’`. The practical execution is visualized in Figure 5.3. To start this process, the justification file is imported as a reference, where `‘aEV01’` denotes the version control hook evidence. Initial validation entails checking the repository’s existence using a probe, with the condition requiring verification before proceeding with subsequent operations. Moving forward, `‘aEV02’` corresponds to the GitHub branch, confirming its presence in

```
load "./justification.jd"

implementation imp of JP {

    implements aEV01 {
      probe is check_repository("https://github.com/ace-design
         /jpipe.git")
      expectation is true
    }

    implements aEV02 {
      probe is verify_branch("main")
      expectation is true
    }

    implements aST01 {
      operation is on("push", "main")
      operation is on("pull", "main")
      expectation is true
    }
```

Figure 5.3: Implementation for trigger workflow

the designated repository. The ultimate operation is governed by the '`contextual automation`' strategy (aST01), executing the '`on`' operation with two arguments: the trigger event and the specific branch. The expectation is set to true, ensuring the parameters are valid for the operation to proceed.

The provided implementation snippet in Figure 5.4 shows the operational details for caching. The initial segment contains the implementation of '`bST02_01_02`', a component of the '`optimise workflow`' strategy, featuring the '`add_cache()`' operation. This operation takes '`path`', '`key`', and '`restore-key`' as arguments, all of which need to be valid for ordered execution. Ensuring the validity of these arguments is needed for verification of the '`Target directory`' evidence, an argument passed to the '`verify_repository`' probe.

```
    implements bST02_01_02 {
      operation is add_cache("~/.m2/repository",
         "${{ runner.os }}-maven-${{ hashFiles('**/pom.xml')
            }}",
         "${{ runner.os }}-maven-")
      expectation is true
    }

    implements bEV01_02_01 {
      probe is verify_repository("~/.m2/repository")
      expectation is true
    }
```

Figure 5.4: Implementation for caching

```
    implements bEV01_05_02 {
      probe is verify_dependencies("mvn")
      expectation is true
    }

    implements bEV01_05_01 {
      probe is verify_repository("../test/")
      expectation is true
    }

    implements bST02_01_05 {
      operation is run_mvn("test")
      expectation is true
    }
```

Figure 5.5: Implementation for Testing

The implementation details for testing in the JD are illustrated in Figure 5.5. Verification of dependencies is facilitated through the '`verify_dependencies()`' operation, with the names of dependencies provided as arguments. This entire process is linked to '`bEV01_05_02`'. Ensuring the existence of the specified repository for test cases evidence is conducted using the '`verify_repository`' operation. The testing strategy is executed by '`run_mvn()`', where the '`mvn`' command serves as an argument. All expectations must be true for the successful execution of a specific operation and probe.

```
    implements bEV01_01_01 {
      probe is checkout()
      expectation is true
    }

    implements bEV01_04_01 {
      probe is verify_repository("../java/")
      operation is setup_java(17,"adopt")
      expectation is true
    }

    implements bST02_01_04 {
      operation is run_mvn("build")
      expectation is true
    }
```

Figure 5.6: Implementation for Building

The operational process for project building is described in Figure 5.6. The sequence starts by conducting an initial code checkout, ensuring its availability for operation, and setting up Java. Subsequently, a verification of the repository is performed to verify the existence of the requisite build packages. This two-step process is integral for a successful build. The final step involves the strategic execution of the project-building operation. This is accomplished by invoking the '`run_mvn()`' strategy, where the mvn command is an important parameter.

The artifact operation can be found in figure 5.7. The evidence says that the target directory must be present (`bEV01_07_01`) before the actual operation is performed in `bST02_01_07`. The strategy used an operation called `upload_artifact()` that accepts the name and path to upload the artifact.

```
implements bST02_01_07 {
  operation is upload_artifact("jpipe-artifact","target/
     jpipe.jar")
  expectation is true
}

implements bEV01_07_01 {
  probe is verify_repository("target/")
  expectation is true
}
```

Figure 5.7: Implementation for artifact

```
implements aST {
  probe is verify_artifact("jpipe-artifact")
  expectation is true
}
```

Figure 5.8: Implementation for verification

In the final result, the success of the project is validated by checking the presence of artifacts in the GitHub repository. As shown in Figure 5.8, the verification of these artifacts provides the decision, and based on the result, we can verify that the artifact is present if it's true.

5.4 Conclusion

Utilizing those Operations with the Justification Diagram can make it easier to understand which part we are addressing from the pipelines. The operation world with justification can be successful when a more detailed analysis is performed to create the actual methods attaching to Justifications. This can be a new way to generate GitHub workflow without depending on the actual workflow.

Chapter 6

Conclusion and future work

In this chapter, we provide a summary of the project and future work of this analysis.

6.1 Summary

In conclusion, this project has provided valuable insights into the strategic development of justifications for effectively managing changes. Beyond a traditional diagrammatic representation, our approach captures the intentions and conclusions from specific actions. The overarching objective is to ease the cognitive load for developers and team members, fostering a clear understanding of how each justification aligns with respective pipeline jobs.

Our analysis, initiated with a GitHub project, demonstrated the Justification Diagram's effectiveness in documenting changes within large, dynamic docker-compose GitHub pipelines. Building on this success, we expanded the application and support of the Justification Diagram to encompass pipelines in GitHub, GitLab, and Azure. This strategic extension significantly enhanced the accessibility and applicability of the Justification Diagram.

As exemplified in Chapter 4, our findings confirm the Justification Diagram's seamless integration with various pipeline structures. Also, we addressed operational aspects to enhance understanding between pipelines and justifications by incorporating additional operations. These operational enhancements lay the groundwork for future work, potentially enabling the automatic generation of workflows based on these operations.

In essence, our project contributes to the current understanding and implementation of justifications and also sets the stage for innovative advancements in the intersection of change management and workflow automation.

6.2 Future work

In this section, we discuss some possible future work that can make Justification approach better.

1. How can one implement these operations to automatically generate a GitHub workflow file, eliminating the need for manually specifying dependencies in the workflow?

   In essence, we are currently associating operational information with the Justification Diagram. However, future work may involve the generation of a workflow file directly from these operations. This innovative approach aims to streamline or replace the traditional method of developers manually writing pipelines, offering a more seamless process without sacrificing explanatory context.

2. How can we ensure the validity of the generated workflow?

   Upon completing the future work of crafting workflows, the subsequent challenge lies in verifying these pipelines. It becomes crucial to confirm that the proposed solution and generation process seamlessly align with the already identified operations.

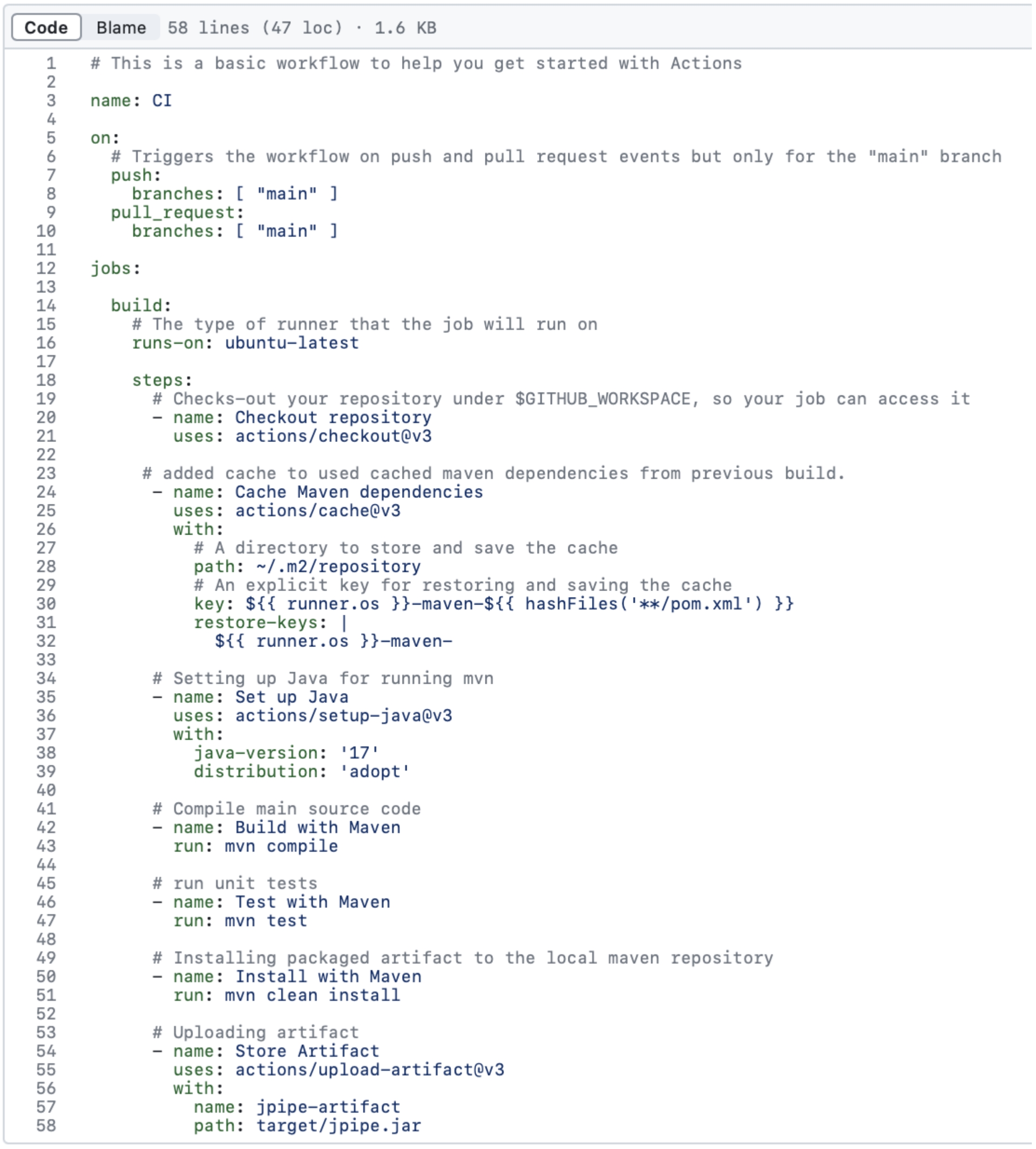
3. How can we introduce a platform that revolutionizes the Justification process, eliminating the need for developers to manually write justifications? Instead, envision a user-friendly environment where they can effortlessly create Justification Diagrams through a simple drag-and-drop interface. This innovation has the potential to bring significant and captivating changes to the world of Justification.

Appendix A

Sample Justification for JPipe

The initial example, featuring the operational justification diagram, is drawn from jPipe [14]. Within this context, jPipe integrates the `‘ci.yml’` file, visually represented in Figure A.1. The fundamental tasks embedded in this workflow are meticulously designed: starting with repository checkout to access code within the specified Ubuntu environment, caching to optimize job execution by reusing previously run tasks, Java installation to facilitate the execution of `‘mvn package’`, package building and testing for the actual build and testing processes in the designated directory, and artifact storage to upload the artifact to the specified path. It is noteworthy that all these tasks are configured to trigger automatically in response to both push and pull requests directed at the main branch.

Extracting these tasks into the justification diagram is a straightforward process. To achieve this, we create a `‘justification.jd’` file, exemplified in Figure A.2. The process commences with defining the justification's name, denoted as `‘JP’`. Subsequently, we meticulously map each job to the appropriate strategy and sub-conclusion. As an illustration, to initiate the workflow upon push and pull requests to



```yaml
# This is a basic workflow to help you get started with Actions

name: CI

on:
  # Triggers the workflow on push and pull request events but only for the "main" branch
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:

  build:
    # The type of runner that the job will run on
    runs-on: ubuntu-latest

    steps:
      # Checks-out your repository under $GITHUB_WORKSPACE, so your job can access it
      - name: Checkout repository
        uses: actions/checkout@v3

     # added cache to used cached maven dependencies from previous build.
      - name: Cache Maven dependencies
        uses: actions/cache@v3
        with:
          # A directory to store and save the cache
          path: ~/.m2/repository
          # An explicit key for restoring and saving the cache
          key: ${{ runner.os }}-maven-${{ hashFiles('**/pom.xml') }}
          restore-keys: |
            ${{ runner.os }}-maven-

      # Setting up Java for running mvn
      - name: Set up Java
        uses: actions/setup-java@v3
        with:
          java-version: '17'
          distribution: 'adopt'

      # Compile main source code
      - name: Build with Maven
        run: mvn compile

      # run unit tests
      - name: Test with Maven
        run: mvn test

      # Installing packaged artifact to the local maven repository
      - name: Install with Maven
        run: mvn clean install

      # Uploading artifact
      - name: Store Artifact
        uses: actions/upload-artifact@v3
        with:
          name: jpipe-artifact
          path: target/jpipe.jar
```

Figure A.1: jPipe CI workflow [14]

the main branch, we employ a strategy named ‘`contextual automation`’, leading to the sub-conclusion of ‘`trigger workflow`’. The key supporting evidence for this configuration includes version control hooks and the specified GitHub branch on which

```
justification pattern JP {

    strategy aST is "Assess Project success"

    sub-conclusion aSC01 is "Trigger workflow"
    strategy aST01 is "Contextual automation"
    evidence aEV01 is "Version control hooks"
    evidence aEV02 is "Github Branch"
    aSC01 supports aST
    aST01 supports aSC01
    aEV01 supports aST01
    aEV02 supports aST01

    sub-conclusion bSC01 is "Assess Project functionalities"
    strategy bST01 is "Verify workflow"
    bST01 supports bSC01
    bSC01 supports aST

    sub-conclusion bSC02_01_02 is "Caching dependencies"
    strategy bST02_01_02 is "Optimise workflow"
    evidence bEV01_02_01 is "Targeted directory"
    bSC02_01_02 supports aST
    bST02_01_02 supports bSC02_01_02
    bEV01_02_01 supports bST02_01_02

    sub-conclusion bSC02_01_07 is "Artifact Versioning"
    strategy bST02_01_07 is "Store artifacts"
    evidence bEV01_07_01 is "Artifact configuration"

    bSC02_01_07 supports bST01
    bST02_01_07 supports bSC02_01_07
    bEV01_07_01 supports bST02_01_07

    sub-conclusion bSC02_01_04 is "Project build"
    strategy bST02_01_04 is "Build with Maven"
    evidence bEV01_04_01 is "Project setup"
    bSC02_01_04 supports bST02_01_07
    bST02_01_04 supports bSC02_01_04
    bEV01_04_01 supports bST02_01_04

    sub-conclusion bSC02_01_05 is "Validated test cases"
    strategy bST02_01_05 is "Perform unit test"
    evidence bEV01_05_01 is "Test cases"
    evidence bEV01_05_02 is "Dependencies installed"
    bSC02_01_05 supports bST02_01_07
    bST02_01_05 supports bSC02_01_05
    bEV01_05_01 supports bST02_01_05
    bEV01_05_02 supports bST02_01_05
    bEV01_05_02 supports bST02_01_04

    evidence bEV01_01_01 is "Code base accessible"
    bEV01_01_01 supports bST02_01_05
    bEV01_01_01 supports bST02_01_04

    conclusion C is "CI"
    aST supports C

}
```

Figure A.2: jPipe justification file

the workflow is intended to be triggered. All of the components are identified with the unique name.

Following the provided instructions for the justification diagram, we can seamlessly generate the illustrative diagram showcased in Figure A.3. The systematic arrangement of all jobs enhances clarity. For instance, the `‘store artifacts’` task is intricately linked to the successful completion of the building and testing stages. Also, it needs verification of all jobs so it can assure the project functionalities by `‘verifying workflow’`. In this context, the justification serves as a valuable aid, offering a clearer understanding of these inter-dependencies. As caching plays a crucial role in the entire project, extending beyond the confines of any specific job and operating independently of others, it forms a direct link to the overarching strategy labeled `‘Assess project success’`. The final component involves triggering the workflow, a condition fulfilled by the `‘contextual automation’` strategy, substantiated by

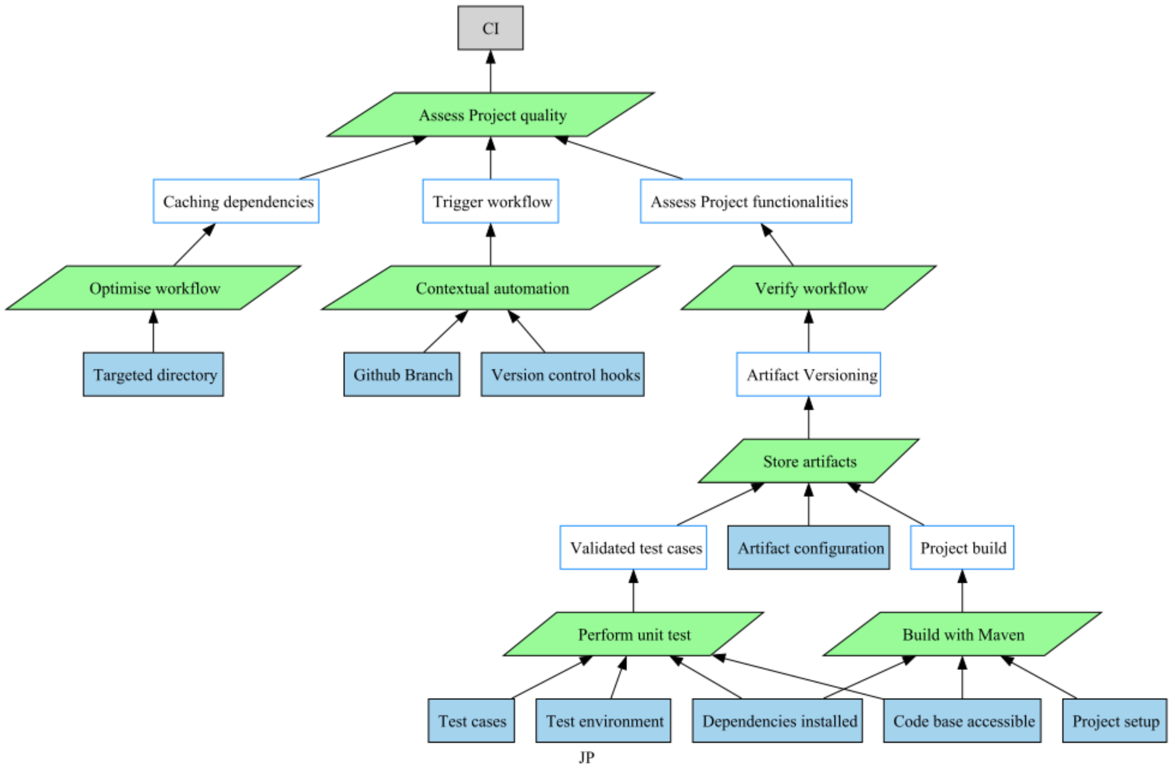


Figure A.3: jPipe justification diagram

evidence from the specified GitHub branch and version control hooks.

Appendix B

Analysis of Docker-compose

This appendix demonstrates the analysis of docker-compose. It contains changes, intentions, and semantic meaning of the changes.

No.	**Change**	**Intention**	**Semantic**
001	Replace manual dependencies installation to installation from file	Enhancement: To make a single point of change for dependencies instead of multiple change in different files.	Reduced complexity of multiple changes

Continued on the next page

Continued from previous page

No.	Change	Intention	Semantic
002	Added GitHub action to install protoc using arduino/setup-protoc@master.	Automation: Replace the manual code of installing protoc with Arduino provided protoc which gives reusability of code instead of writing it for all the different operating system.	Reusability of already developed modules to get speed and automation in workflow execution.
003	Linter added in workflow	Linter is added to check code styles and give feedback to best practices. That is helpful to maintain code quality during continuous integration.	Automation in validating Coding standard and guidelines
004	First local basic end-to-end test added. Also replace the pipeline triggering from push and pull to only push.	Testing: First step towards adding more end to end testing like adding test cases for docker start and stop commands.	Validating local test cases

Continued on the next page

Continued from previous page

No.	Change	Intention	Semantic
005	Job change to parallel and caching is added	Still build can be faster with separate job for Tests. For now the build job run setup of Go, Build, Test, and E2E Test.	Fostering build speed
006	Improve linting operation and dependencies installation	Error handling can be added to understand the exact issue in the workflow. For now, if any step fails, the workflow will continue to execute subsequent steps, which make it hard to troubleshoot the real issue.	Fostering build speed
007	installation of grpc for end to end test case and setup-node installation	Testing: Just test e2e test of the Javascript test using grpc client. In the direction of adding more e2e tests.	To perform end to end test.
008	The same workflow job for windows is added.	Validate cross building and testing on different Operating system.	Verify Build on cross plate-forms to prevent release time failure.

Continued on the next page

Continued from previous page

No.	Change	Intention	Semantic
009	The same workflow for Linux is added	Continue working on adding Linux CI in the workflow file to validate cross building.	Verify Build and test on cross plate-forms to prevent release time failure.
010	build tag for back-end targets are updated	Testing: Secondly they want to add more and more back-end services to be build and test during the new push request. So, they can achieve consistency during different scenario.	Back-end testing added
011	Updated GitHub actions/cache version from v1 to v2	Version Updates	Regular version updates

Continued on the next page

Continued from previous page

No.	**Change**	**Intention**	**Semantic**
012	Update in execution of workflow on different branch and different event.	Overall structure: Their intension was to trigger original workflow on push request and perform build/tests with Ubuntu server instead of both windows and Ubuntu.	Structural improvement
013	ACI end-to-end test case is added for azure cloud	Testing: More steps toward testing the compose with Azure cloud.	More Back-end testing added.
014	Moved ACI from ci.yaml file to master-ci.yaml.	Organize steps with most related steps	Structural improvement
015	Minor update in the version of linter.	Version updates to support up to date versions.	Version Updates
016	More test cases with ECS is added to test compose with Amazon Web Service.	Testing: More steps toward testing the compose with Amazon Web Service cloud.	Make system more robust with different tests

Continued on the next page

Continued from previous page

No.	Change	Intention	Semantic
017	Updated go and golang version	Version updates to support up to date versions.	Version Updates
018	Automatically adds licence and copyright information in the source files of the project using workflow file if this information is already not present.	Validation: Check/Add Licence and copy write information through workflow file.	Validation of source code.
019	The same Makefile target is used to validate with additional go checks. The target will check the go.* files are up-to-date.	Validation: Validation of go.* file.	Validation with source files

Continued on the next page

Continued from previous page

No.	**Change**	**Intention**	**Semantic**
020	Additional validation for checking imports those are not allowed in particular back-end contexts	Validation: One more validation for imports added.	Validation with source codes.
021	Added Dependabot in workflow.	ISSUE RESOLVE: Address the issue of dependabot doesn't remove old version of package from go.sum when updating	Automatic dependency updated to reduce the complexity of updating it manually.
022	Replaced the exact match with startWith function. (i.e. == with startWith())	The result of the "github.event.pusher.name" is the name of the dependabot which is not exact the passed value "dependabot-preview". It has something at the end. So, they replaced the == with startWith function.	Internal enhancement

Continued on the next page

Continued from previous page

No.	Change	Intention	Semantic
023	Finally, replaced the startWith by == with actual name of the dependabot.	Just play around the dependabot name to resolve the identified issue.	Automation in updating
024	Replaced the job with Make file command. The target use the go list functionality to replace the older dependency version with newer one.	Every time a push request was triggered the checking of dependencies updates. So, that was not necessary every time. That's why they removed it from the workflow file and added just a command in the makefile to run it whenever needed instead of every push request.	Internal improvement

Continued on the next page

Continued from previous page

No.	Change	Intention	Semantic
025	The push request is now just triggering for main branch. And pull request triggered and run the whole workflow without any condition.	As now the workflow is stable and can run all the basic builds and tests, it is time to run the workflow during pull requests from forks.	Trigger changes
026	Run e2e test metrics.	Reformation: They now started to implement real test cases with the help of test metrics. So, they start to remove backend tests. For now they just removed ECS test case from the default continuous integration workflow.	e2e tests added

Continued on the next page

Continued from previous page

No.	Change	Intention	Semantic
027	Implemented job in CI workflow file for cross-validation of build on different operating systems (Windows, Linux, MacOs)	Build verification testing: Perform validation of build failure on different operating systems before it goes through the release stage.	Cross validation
028	Start to remove testing backends: First removal of local backend	In order to remove all backends to start the unit, e2e, and metrics tests during regular CI workflow.	Mature enough to remove unnecessary tests
029	updated version of linter installtion file from v1.30.0 to v1.33.0	VERSION UPDATES	VERSION UPDATES

Continued on the next page

Continued from previous page

No.	**Change**	**Intention**	**Semantic**
030	workflow file download and setup docker CLI to run docker commands and docker actions.	They want to replace the make dependencies with some customize and fast docker actions. It can allow the use of caching with cache-type=gha which can help to make every job faster. Also, it can provide the workflow file to run the docker commands.	Added Docker supports
031	Also removed the ”example” source code from the repository as now the workflow is mature enough to work on real scenarios.	Stop running one more backend. The ”Example” backend was not performing any real action. It was just a static test to check the expected behavior of the compose commands.	Mature enough to remove unnecessary tests

Continued on the next page

Continued from previous page

No.	Change	Intention	Semantic
032	Just removed the unnecessary env command mistakenly not removed in the previous commit. But internally they moved the folder structure of the different tests in the single folder to associated backends.	Handle mistakes	Handle mistakes
033	New back-end added: Kube	Added new back-end functionality to check Kubernetes	New test for Kube.
034	Version updates of go and github action setup-go. and Version updates for setup docker and golangci-lint	VERSION UPDATES	VERSION UPDATES

Continued on the next page

Continued from previous page

No.	Change	Intention	Semantic
035	Added Github Action name.	Without name it was hard to understand what the particular job was performing and it was run with the previous job. So, adding seperated job makes it easy to understand.	Enhanced naming convention
036	Separate job for cross build can make it faster compare to previous versions	Every time even when Pull request performed, the whole build + cross build was performed which was making more resources consumption. By replacing it to just main branch, it reduces the execution time with less resources.	Cross building
037	version update for lint and resolve typo.	VERSION UPDATES	VERSION UPDATES

Continued on the next page

Continued from previous page

No.	**Change**	**Intention**	**Semantic**
038	Split cli into two builds compose-cli(docker) and composeV2 cli(local). Eventually, it perform builds for docker and local backend on regular and cross platforms.	MOVE builds from main branch to v2 to introduce newer docker compose version (v2)	Enhanced distribution of job
039	End to end tests are move to pkg and change the reference to run it in regular workflow.	MOVE tests from main branch to v2 to introduce newer docker compose version (v2)	Enhanced distribution of job

Continued on the next page

Continued from previous page

No.	Change	Intention	Semantic
040	replaced push and pull triggers from main branch to v2 branch, backend kube is removed, removed cross-compose-plugin and cross plateform moved to cmd for v2	Final move to newer version of docker compose - v2	Move to newer version
041	Updated Go version from 1.16 to 1.17	VERSION UPDATES	VERSION UPDATES

These information is extracted from the docker-compose GitHub repository [9]. The full analysis of some version can be accessed in excel file [18]. Some information is adjusted according to the requirements. For example two commits are merged to gain some important information from the change sets.

Bibliography

[1] 2022 accelerate state of devops report by google cloud [online]. Accessed on 2023-16-09.

[2] Adr process - aws prescriptive guidance [online]. Accessed on 2023-12-01.

[3] Angular github workflow [online]. Accessed on 2023-12-01.

[4] Facebook outage in october 2021 [online]. Accessed on 2023-16-09.

[5] Microsoft misconfiguration in december 2019 [online]. https://msrc.microsoft.com/blog/2020/01/access-misconfiguration-for-customer-support-database/. Accessed on 2023-15-09.

[6] Chaminda Chandrasekara and Pushpa Herath. *Hands-on github actions: Implement CI/CD with github action workflows for your applications.* Apress, 2021.

[7] Hugo da Gião. A model-driven approach for devops. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–3, 2022.

[8] debputy. A declarative manifest for building debian packages. `https://salsa.debian.org/debian/debputy/-/blob/main/.gitlab-ci.yml`.

[9] Docker. compose. `https://github.com/docker/compose.git`.

[10] S. Fleming. *The DevOps Engineer's Career Guide: A Handbook for Entry-Level Professionals to Get Into Continuous Delivery Roles for Agile Software Development*. Career Series. Stephen Fleming, 2019.

[11] Mayank Gokarna and Raju Singh. Devops: A historical review and future works. In *2021 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS)*, pages 366–371, 2021.

[12] Priscila Heller. *Automating Workflows with GitHub Actions: Automate software development workflows and seamlessly deploy your applications using GitHub Actions*. 2021.

[13] ISO. Assurance case. `https://www.iso.org/standard/52926.html`.

[14] Sébastien Mosser, Aaron Loh, Deesha Patel, and Nirmal Chaudhari. jpipe. `https://github.com/ace-design/jpipe.git`.

[15] Abrar Mohammad Mowad, Hamed Fawareh, and Mohammad A. Hassan. Effect of using continuous integration (ci) and continuous delivery (cd) deployment in devops to reduce the gap between developer and operation. In *2022 International Arab Conference on Information Technology (ACIT)*, pages 1–8, 2022.

[16] Mumble. A voice-chat program. `https://github.com/mumble-voip/mumble/blob/master/.ci/azure-pipelines/main.yml`.

[17] openxcell. Devops - the complete guide for 2023. `https://www.openxcell.com/devops/`.

[18] Deesha Patel. Docker-compose analysis. `https://mcmasteru365-my.sharepoint.com/:x:/r/personal/pated193_`

mcmaster_ca/Documents/Analysis%20of%20Docker-Compose.xlsx?d=w8d4355e40ee543d2a1b8094a109e8800&csf=1&web=1&e=jlAIJD. Accessed: 2023-11-03.

[19] Thomas Polacsek. Validation, accreditation or certification: a new kind of diagram to provide confidence. In *2016 IEEE Tenth International Conference on Research Challenges in Information Science (RCIS)*, pages 1–8. IEEE, 2016.

[20] Thomas Polacsek, Sanjiv Sharma, Claude Cuiller, and Vincent Tuloup. The need of diagrams based on toulmin schema application: an aeronautical case study. *EURO Journal on Decision Processes*, 6:1–26, 07 2018.

[21] Corinne Pulgar. Eat your own devops: a model driven approach to justify continuous integration pipelines. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 225–228, 2022.

[22] Rust. Rust source repository. https://github.com/rust-lang/rust/commits/master/.github/workflows/ci.yml.

[23] Hadi Safari, Nazanin Sabri, Faraz Shahsavan, and Behnam Bahrak. An analysis of gitlab's users and projects networks. In *2020 10th International Symposium onTelecommunications (IST)*, pages 194–200, 2020.

[24] Scala. Scala. https://github.com/scala/scala/blob/2.13.x/.github/workflows/ci.yml.

[25] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration,

delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017.

[26] Pablo Valenzuela-Toledo and Alexandre Bergel. Evolution of github action workflows. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 123–127, 2022.

[27] Manish Virmani. Understanding devops bridging the gap from continuous integration to continuous delivery. In *Fifth International Conference on the Innovative Computing Technology (INTECH 2015)*, pages 78–82, 2015.

[28] Chandrasekar Vuppalapati, Anitha Ilapakurti, Karthik Chillara, Sharat Kedari, and Vanaja Mamidi. Automating tiny ml intelligent sensors devops using microsoft azure. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 2375–2384, 2020.