FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

A Two-Level Model-Driven Engineering Approach for Reengineering CI/CD Pipelines

André Flores

WORKING VERSION

Mestrado em Engenharia Informática e Computação

Supervisor: Jácome Cunha

Co-supervisor: Hugo Gião

July 1, 2024

A Two-Level Model-Driven Engineering Approach for Reengineering CI/CD Pipelines

André Flores

Mestrado em Engenharia Informática e Computação

July 1, 2024

Resumo

A integração, entrega, e implantação contínuas (CI/CD) facilitam o desenvolvimento colaborativo de software e melhora a qualidade do produto. A prática ganhou popularidade com a metodologia de *eXtreme Programming* e tem visto uma adoção crescente nos últimos anos.

Existem muitas plataformas de CI/CD disponíveis para automatizar *workflows* de desenvolvimento de *software*, e cada uma delas possui vantagens e desvantagens, dependendo dos projetos de software individuais. Muitos projetos de software utilizam várias plataformas CI/CD simultaneamente ou migram entre elas, à procura de funcionalidade, escalabilidade ou melhores preços. Essas plataformas frequentemente têm uma curva de aprendizagem elevada, principalmente devido à sua sintaxe, e os engenheiros de software relatam dificuldades ao migrar entre plataformas. A migração entre plataformas também pode ser um processo demorado.

Este trabalho tem como objetivo usar engenharia dirigida por modelos para facilitar a migração de *pipelines* de CI/CD entre plataformas. Isso será feito usando dois níveis de meta-modelos de pipelines CI/CD, um independente de plataforma e um específico a plataforma, para mapear funcionalidades comuns entre plataformas.

O resultado final é uma metodologia de migração que compila *pipelines* de CI/CD de uma plataforma para outra. A abordagem é inspirada pelo modelo de ferradura tradicional da reengenharia, que abstrai artefactos de *pipelines* existentes para um modelo detalhado que serve de representação intermédia. *Pipelines* semanticamente equivalentes podem ser geradas a partir deste modelo noutras plataformas de CI/CD. Para ser de ainda mais útil, a nossa abordagem também permite juntar vários *pipelines* de CI/CD de plataformas diferentes num único *pipeline*.

Para avaliar a abordagem, comparamos a execução do *pipeline* de CI/CD original com a do *pipeline* que gerámos noutra plataforma depois do processo de migração. Além disso, também testamos migrar o *pipeline* gerado de volta para a sua plataforma original, com intuito de verificar se houve alterações semânticas. Verificamos que a nossa abordagem é capaz de gerar *pipelines* com execução equivalente aos da plataforma original. Em muitos casos, os *pipelines* gerados podem ser migrados de volta para a plataforma original sem alteração semântica.

Abstract

Continuous integration, delivery, and deployment facilitates collaborative software development and improves product quality. The practice gained popularity with the eXtreme Programming methodology and has seen increased adoption in recent years.

There are many CI/CD platforms available to automate software development workflows, and each of them has advantages and disadvantages depending on the individual software projects. Many software projects use multiple CI/CD platforms simultaneously or migrate between platforms, chasing functionality, scalability, or better pricing. These platforms often have a high learning curve, primarily due to the different syntaxes, and developers report difficulties when migrating between platforms. Migration can also be a lengthy process.

This work aims to leverage model-driven engineering to facilitate the migration of CI/CD pipelines between platforms. This will be done using two levels of CI/CD pipeline meta-models, platform-independent and platform-specific, to map common functionality between platforms.

The result is a migration methodology that can compile CI/CD pipelines from one platform to another. The approach is inspired by the traditional reengineering horseshoe model, which abstracts existing pipeline artifacts into a comprehensive model as an intermediate representation. Semantic-equivalent pipelines can be generated from this model in any novel CI/CD tool. To be of even greater use, the migration approach also allows users to merge multiple CI/CD pipelines from various platforms into a single pipeline.

To evaluate the correctness of the approach, we compare the execution of the original CI/CD pipeline with the one of the generated pipeline in another platform after going through a migration process. Moreover, we also execute double round-trips through our migration software, where we attempt to transform the generated CI/CD pipeline back into the original platform to see if it is equivalent to the original CI/CD pipeline. We find that our approach is capable of generating pipelines with equivalent execution to the ones in the original platform. In many cases, the generated pipelines can be migrated back into their original platform without semantic alteration.

Acknowledgments

First and foremost, I would like to thank Professor Jácome Cunha and Hugo Gião for their guidance throughout the entire dissertation process.

I would also like to thank Professors Vasco Amaral, Gregor Engels, and Stefan Sauer for their expertise in model-driven engineering and Alexandre Oliveira for his help with modeling CI/CD platforms.

André Flores

"If I had more time, I would have written a shorter letter."

Cicero

Contents

List of Figures

List of Tables

List of Listings

Abbreviations and Symbols

API	Application Programming Interface
AST	Abstract Syntax Tree
ATL	ATL Transformation Language
CAMEL	Cloud Application Modelling Execution Language
CD	Continuous Delivery/Deployment
CI	Continuous Integration
CLI	Command Line Interface
DevOps	Development & Operations
DSL	Domain-Specific Language
DSML	Domain-Specific Modeling Language
ECL	Epsilon Comparison Language
EGL	Epsilon Generation Language
EMF	Eclipse Modeling Framework
EML	Epsilon Merging Language
EMOF	Essencial MOF
ETL	Epsilon Transformation Language
FuSaFoMo	Functional Safety Formal Model
GAI	GitHub Actions Importer
GHA	GitHub Actions
GMF	Eclipse Graphical Modeling Framework
GPML	General-Purpose Modeling Language
IDE	Integrated Development Environment
IoT	Internet-of-Things
JSON	JavaScript Object Notation
LCEP	Low-Code Engineering Platform
M2M	Model-to-Model
M2T	Model-to-Text
MDE	Model-Driven Engineering
MDSE	Model-Driven Software Engineering
MLS	Modeling Language Suite
MOF	Meta Object Facility
NPM	Node Package Manager
OCL	Object Constraint Language
OMG	Object Management Group
OSS	Open-Source Software
PIM	Platform-Independent Model
PIMM	Platform-Independent Meta-Model
PSM	Platform-Specific Model

PSMM	Platform-Specific Meta-Model
QA	Quality Assurance
QVT	Query/View/Transformations
SDLC	Software Development Life Cycle
SLR	Systematic Literature Review
RQ	Research Question
T2M	Text-to-Model
TCS	Textual Concrete Syntax
TOSCA	Topology and Orchestration Specification for Cloud Application
UML	Universal Modeling Language
VM	Virtual Machine
XP	eXtreme Programming

Chapter 1

Introduction

Continuous Integration, Delivery, and Deployment (CI/CD) means that changes to a program's code are consistently integrated into the current system and deployed to a production environment with little delay. These changes should only be integrated if, after adding the code, the system can be built from scratch and pass all required tests [11, 76].

Practicing CI/CD implies activities like frequent code commits and builds, automated building and testing, immediately fixing a broken build, etc. [52, 40, 131, 127, 6]. These activities are organized into CI/CD pipelines [121, 139, 72, 12, 150]. Listing 1.1 is an example of a high-level definition of such a pipeline.

```
on pull request creation:
    - run static analysis on code
    - build project
    - run unit tests
    - run acceptance tests
    - output static analysis, unit and acceptance test results, and unit test
     ↪  coverage report
```

Listing 1.1: High-level example of a CI/CD pipeline.

CI/CD brings several benefits. These include reduction of cost and risk of work integration in distributed teams, increased software reliability, reduced time to market, improved customer satisfaction, and enhanced productivity [52, 131, 40, 24, 6].

CI/CD pipelines are implemented using CI/CD platforms. These CI/CD platforms support the automated building, testing, and deployment of software. They also offer other features like integration with the code-hosting platform and package marketplaces where users can search for extensions that improve functionality or usability (differing platforms have different terminology for packages, e.g. Plugins in Jenkins, Actions in GitHub Actions (GHA), Orbs in CircleCI) [29, 48, 93].

```yaml
name: 'Link Checker: All English'

# The `on` key lets you define the events that trigger the workflow.
on:
  push:
    branches:
      - main

# The `jobs` key groups together all the jobs that run in the workflow file.
jobs:
  check-links:
    runs-on: ubuntu-latest
    # The `steps` key groups together all the steps that will run as part of the
    ↪ job.
    steps:
      # The `uses` key tells the job to retrieve the action named
      ↪ `actions/checkout`.
      - name: Checkout
        uses: actions/checkout@v4

      - name: Gather files changed
        uses: trilom/file-changes-action
        with:
          fileOutput: 'json'

      # The `run` key tells the job to execute a command on the runner.
      - name: Link check (warnings, changed files)
        run: ./script/rendered-content-link-checker.mjs --language en --max 100
        ↪ --check-anchors --check-images --verbose --list $HOME/files.json
```

Listing 1.2: Example of a CI/CD pipeline in GHA. Adapted from GitHub [138].

There are many CI/CD platforms available on the market. The Cloud Native Computing Foundation's curated list of CI/CD platforms numbers sixty-one, and it is not complete [133]. Recently, more CI/CD platforms have been emerging [121, 127].

Often, CI/CD platforms are configured as code in a configuration file written in their domain-specific language (DSL), as seen in listing 1.2. These configuration files are stored with the rest of the project's code.

1.1 Problem Definition

A project is not limited to using just one CI/CD platform at a time (co-usage) or always sticking to the same platform. Over its lifetime, a project can change the CI/CD platforms it uses. This process, referred to as migration, involves taking the CI/CD pipeline modeled in the current CI/CD platform/platforms and translating it into the new one(s).

There are diverse motives for migrating CI/CD platforms. In their study of the reasons for the changes in the CI/CD landscape [121], Mazrae et al. interview twenty-two respondents with experience setting up, managing, or maintaining the CI/CD process of projects. They query their interviewees on what motivated CI/CD platform migrations in their projects. They found that the interviewees were motivated by better integration with the code-hosting platform, better features, and reduced platform co-usage.

The study of GitHub repositories detailed in chapter 3 reveals a significant change rate in the CI/CD platform stacks of projects over time, meaning migrations are commonplace. Furthermore, it also finds a significant amount of co-usage of CI/CD platforms. Considering that one of the main motivators for migration is decreasing the amount of co-usage, this could mean many projects are looking to migrate.

Migrations are often hard to execute. CI/CD platforms have a high learning curve, there are fundamental differences between platforms, configuring CI/CD is trial-and-error by nature, and some features may be missing for continuous deployment. The syntax of the new tool is highlighted as a problem [121]. CI/CD implementations for complex projects can take around five weeks to migrate. If the implementation must support multiple projects, the timeline can shift to months [77].

While migration is taking place, projects can experience reduced productivity due to not only the effort being expended in the migration but also because CI/CD processes can break down during the process. This hardship in migrating CI/CD can also lead projects to stay with a given platform after it stops being optimal for their use case.

In summary, there is a constant need to change CI/CD platforms. How can we aid CI/CD developers to migrate their pipelines?

1.2 Objectives

This work's objective is to support developers in migrating CI/CD platforms. The objective is to make migration easier and faster, helping developers keep up productivity and reduce lock-in to a platform.

In practice, this is done through a CI/CD pipeline transpiler designed using model-driven engineering (MDE). With this transpiler, we want to automate the CI/CD migration process as much as possible. This is part of an ongoing effort to improve developers' experience with CI/CD by leveraging MDE to interact with CI/CD in a platform-independent manner.

This work should answer the following research questions (RQs).

RQ1. What are the main core concepts shared by and unique to the different CI/CD platforms?

Our goal for RQ1 is to examine various CI/CD platforms and develop a meta-model capable of representing their core concepts. We intend to create an abstraction that transcends the specifics of individual languages. We answer this RQ in section 5.2.

RQ2. Can a platform-independent meta-model be the basis for the accurate translation of CI/CD pipelines between platforms?

For RQ2, our goal is to evaluate the capability of our meta-model to represent real-world pipelines. Using model transformations, it should be possible to parse a CI/CD pipeline in a given platform to a platform-independent model. Afterward, we should be able to generate a CI/CD pipeline in a possibly different platform from the original one. We answer RQ2 in section 9.3.

RQ3. Can CI/CD pipeline migration be fully automated?

For RQ3, we intend to ascertain if a fully developed transpiler based on our approach could be used to completely automate CI/CD migrations. Section 9.4 provides an answer to this RQ.

1.3 Contributions

Our main contribution is a platform-independent meta-model (PIMM) for CI/CD pipelines. This meta-model can abstract CI/CD pipelines away from their implementation platform with enough detail to allow accurate migration of CI/CD platforms. Moreover, we also created platform-specific meta-models (PSMMs) for three popular CI/CD platforms: GHA, CircleCI, and declarative Jenkins.

Furthermore, we also contribute model transformations that implement a reengineering process that migrates CI/CD pipeline platforms. We can parse CI/CD scripts into platform-specific model (PSM) instances, transform that PSM into a platform-independent model (PIM), transform the PIM into a PSM of a different CI/CD platform, and generate a CI/CD pipeline script in that platform from the PSM.

We created a transformations DSL (TDSL) so users can interact with our models throughout the reengineering process. The TDSL simplifies model transformations relevant to migrating CI/CD platforms (e.g., changing platform-specific plugins) by allowing the user to specify the transformation using a syntax inspired by natural language.

The reengineering process and the TDSL are integrated into a command-line interface (CLI) that functions independently from the Eclipse Integrated Development Environment (IDE). This ensures our approach is valuable to users even if they are not familiar with MDE technologies.

Lastly, this work has been featured in part in three separate articles submitted for publication: “Chronicles of CI/CD: A Deep Dive into its Usage Over Time” by Gião et al. [57], “A Meta-Model to Support the Migration and Evolution of CI/CD Pipelines” by Gião et al. [56], and “A Two-Level Model-Driven Approach for Reengineering CI/CD Pipelines” by Flores et al. [51].

1.4 Document Structure

The rest of this document is structured as follows.

Chapter 2 gives further background information on CI/CD and presents current migration support given by CI/CD platform providers. Furthermore, it also details model-driven engineering concepts relevant to this work. Lastly, it includes related work.

Chapter 3 details a study of around 600,000 GitHub repositories that was done to discover trends in CI/CD usage. It focuses on the most popular platforms, co-usage of platforms, and migrations between platforms.

Chapter 4 presents an overview of the solution and an example of its execution.

Chapter 5 details the process of finding common characteristics to various CI/CD platforms and creating the PIMM. This chapter also addresses RQ1.

Chapter 6 details how we implement the reengineering process outlined in chapter 4

Chapter 7 details the implementation of the TDSL, used to complement automatic migration functionality.

Chapter 8 details the architecture of the proposed tool.

Chapter 9 details the evaluation of the solution and addresses RQ2 and RQ3.

Chapter 10 concludes with a discussion of the results and future work.

Chapter 2

State of the Art

This chapter details the current state of CI/CD and of model-driven engineering.

Section 2.1 goes into the emergence of CI/CD and its relevance in software engineering.

Section 2.2 lays out support given by CI/CD platform providers on migrating CI/CD platforms. It includes both automated and manual migration support.

Section 2.3 gives background information on MDE. It includes the motivation for using modeling in software engineering, details on models and model transformations, and how models and model transformations can be used in software reengineering.

Section 2.4 includes related work regarding MDE and CI/CD.

2.1 Background on CI/CD

In the late 90s, Beck proposed the eXtreme Programming (XP) methodology for software development to address shortcomings of the waterfall model [11].

The waterfall model, the more traditional approach to software development that was first described in 1970 [122, 113], is static and approaches software development linearly and sequentially, completing one activity before the other [3]. It can be said to involve four phases: requirement analysis, design, implementation, testing, and operation and maintenance [3].

Due to its rigid nature, the use of waterfall generates well-known problems. These include but are not limited to reduced ability to deal with change, increased rework, and unpredictable software quality due to late testing [113, 128].

According to Beck, waterfall arose from the measurement that the cost of changing software increased dramatically. However, the software community made strides to reduce this change cost by introducing relational databases, modular programming, and information hiding. With this in mind, there was no need for the software engineering community to be beholden to Waterfall [11].

As an Agile process [103], XP embraces change throughout the software development lifecycle (SDLC). To achieve this, XP uses shorter development cycles. XP also "turns the conventional software process side-ways", executing the four constituent phases of Waterfall a little at a time during the development cycle instead of sequentially [11].

XP aims to increase communication, improve software quality, improve customer feedback, and create smaller and more frequent software releases, shortening time to market [11].

When Beck presented XP, he outlined thirteen constituent practices: planning game, small releases, metaphor, simple design, tests, refactoring, pair programming, collective ownership, on-site customer, 40-hour work weeks, open workspace, just rules, and continuous integration (CI) [11].

Later, Humble and Farley extended the philosophy behind CI into software deployment. Continuous Delivery/Deployment (CD) expresses the steps to deploy software as a deployment pipeline. This pipeline automates the steps that take successfully integrated code and put a new software version into production, increasing release frequency [6, 76]. Together, CI and CD form CI/CD [6].

Circa 2007, DevOps was introduced [109]. A portmanteau of Development and Operations, DevOps is a development methodology that bridges the gap between these two areas by emphasizing communication and collaboration, CI/CD, and quality assurance (QA) [78]. CI/CD pipelines are at the core of DevOps [10].

Agile practices like XP have seen a significant adoption rate since they were introduced [91, 107] and DevOps's importance to organizations has been increasing [109]. In its annual reports on the state of DevOps, Puppet has found that high-performing organizations enabled by DevOps deploy code thirty times faster than their lower-performing peers [110]. They also have fewer failures on deployments and recover from failure much faster [111, 112]. Moreover, CI/CD has also become an essential part of cloud-computing [6].

With XP's and DevOps's increasing popularity, CI/CD has seen greater relevance in software development for companies or open-source software (OSS) communities since it ensures integrity and control over changes made to the software project [127, 121, 73].

2.2 Migration Support from CI/CD Platform Providers

This section details support for CI/CD migrations that is currently offered to practitioners, whether in the form of an automated migration tool (section 2.2.1) or manual migration guides (section 2.2.2).

Of the CI/CD platforms listed by the Cloud Foundation [133], only the ones that had ever achieved over 1% market share (figure 3.2) in the study detailed in chapter 3 were analyzed. These number eleven: AppVeyor [5], CircleCI [28], Codefresh [30], Concourse [34], Drone [38], GHA [61], Gitlab CI/CD [63], GoCD [64], Jenkins [80], Kubernetes [86], and Travis CI [136].

2.2.1 Automated Migrations

Only one of the aforementioned CI/CD platforms has an available tool for automated migration.

GitHub provides a tool called GitHub Actions Importer (GAI) to plan and automate migrations to GHA [8]. Its goal is to achieve an 80% conversion rate for every workflow, but this depends on the makeup of each pipeline. The tool extends to the GitHub CLI and runs on a Docker container.

GAI supports migration from seven CI/CD platforms: Azure DevOps, Bamboo, Bitbucket Pipelines, CircleCI, GitLab, Jenkins, and Travis CI. The user connects GAI to the existing CI/CD platform platform by supplying access credentials.

GAI can audit existing pipelines (to determine how complete and complex an automated migration can be), forecast the usage time of GHA by the transformed pipeline, dry-run a migration, and create a pull request with the migrated pipeline in the GitHub repository.

GAI's functionality can be extended with the use of custom transformers. The transformers can migrate items that GAI cannot migrate automatically. It can also change references to runners, virtual machines (VMs) where a pipeline is executed, and environment variables. Transformers are defined in a DSL built on top of Ruby.

There are limitations to GAI's functionality. These vary with the CI/CD being migrated from but are usually related to functionality that cannot be mapped one-to-one in GHA, unknown packages being used in the original pipeline, unsupported functionality in GHA, and secret environment variables.

2.2.2 Manual Migrations

Of the eleven providers analyzed, only four provide guidance on migrating to their CI/CD platform. These are: CircleCI, Codefresh, GHA, and Gitlab CI/CD.

The guides mostly center on comparing syntaxes between migrating platforms. These comparisons focus on basic common functionality or key differences between platforms and rarely provide help for more complex CI/CD pipelines [77, 62, 81, 94, 114]. Figure 2.1 is an example of such guidance.

Cases where the platform providers offer more detailed guides are laid out below. GHA does not provide any support more detailed than what was already specified.

CircleCI

CircleCI provides a detailed methodology on migrating CI/CD platforms [77]. The method includes various phases: assessment, planning, preparation, testing, and migration. They provide a rough time frame for migration: less than one week for a simple project and around five weeks for a complex one.

Codefresh

Codefresh offers a superset of Jenkins capabilities [81]. Codefresh has a detailed guide on migrating from Jenkins pipelines. This guide includes feature, architecture, and installation comparisons between the two platforms and general advice on creating Codefresh pipelines. There are detailed instructions on migrating Jenkins freestyle jobs, pipelines, credentials, pipelines that create Docker images, pipelines that deploy to Kubernetes, shared libraries, checking out source code, and step conditions. There is also a guide on co-usage with Jenkins while the migration is happening. This

GitHub

```yaml
name: My GitHub Actions Workflow

on: [push]

jobs:
  job_1:
    runs-on: ubuntu-latest
    steps:
      # job steps
  job_2:
    needs: job_1
    runs-on: ubuntu-latest
    steps:
      # job steps
```

CircleCI

```yaml
jobs:
  job_1:
    executor: my-ubuntu-exec
    steps:
      # job steps
  job_2:
    executor: my-ubuntu-exec
    steps:
      # job steps

workflows:
  my_workflow:
    jobs:
      - job_1
      - job_2:
          requires:
            - job_1
```

Figure 2.1: Example of syntax comparison guide from a CI/CD platform company [100].

guide includes instructions on how to run Codefresh pipelines from Jenkins jobs and how to run Jenkins jobs from Codefresh pipelines.

Codefresh can also run Actions available in the GitHub Actions Marketplace [62]. This could ease the transition process to GHA.

Gitlab CI/CD

Gitlab gives pointers to manage organizational change and technical questions to consider before a migration [114].

Advice on organizational changes includes setting and communicating clear migration goals, ensuring alignment from the relevant leadership teams, educating users on changes, finding ways to sequence or delay parts of the migration, not leaving the CI/CD pipeline in a partially-migrated state for too long, and not moving the CI/CD pipeline as-is, instead taking advantage of new functionality and updating the implementation.

The suggested technical questions center on the number of projects using the pipeline, the git branching strategy, the tools used to build and test code, security scanners, and deployment.

2.3 Model-Driven Engineering

This section lays the background for model-driven engineering. It details what modeling is (section 2.3.1), model-driven software engineering (section 2.3.2), modeling languages (section 2.3.3), meta-modeling (section 2.3.4), model transformations (section 2.3.5), Model-Driven Software Reengineering (section 2.3.6), and technologies used in MDE (section 2.3.7).

2.3.1 From Abstraction to Modeling

Abstraction is a natural behavior of the human mind. It can be defined as the capability of finding the commonality in many different observations and thus generating a mental representation [18].

To be able to abstract is to simultaneously [18]:

- generalize specific features of real objects (generalization)
- classify objects into coherent clusters (classification)
- aggregate objects into more complex ones (aggregation)

In science and technology, abstraction is often referred to as modeling. A model is "a simplified or partial representation of reality, defined in order to accomplish a task or to reach an agreement on a topic" [18]. This also means a model will never describe reality in its entirety. Notable models in science include the Bohr model of the atom [14, 18].

Models perform at least two roles regarding abstraction [18]:

- **Reduction** - models only reflect a portion of the original object's properties
- **Mapping** - models are based on an original object, which is taken as a prototype of a category of individuals and is abstracted and generalized to a model

Models can also be classified regarding their purpose [18]. They can be:

- **Descriptive** - for describing the reality of a system or context
- **Prescriptive** - for determining the scope and detail at which to study a problem
- **Defining** - for defining how a system shall be implemented

Models are meant to describe two main dimensions of a system: the static (or structural) and the dynamic (or behavioral). *Static models* focus on the static parts of the system, its managed data, and architecture. *Dynamic models* describe the behavior of the system by showing the execution sequence of actions and algorithms, collaborations among system components, and changes to the internal state of components and applications [18].

Since the human mind can process nothing without abstraction, it can be said that "everything is a model" [18].

2.3.2 Model-Driven Software Engineering

Model-Driven Software Engineering (MDSE), or simply Model-Driven Engineering (MDE), is a "methodology for applying the advantages of modeling to software engineering activities" [18]. According to Sendall and Kozaczynski, MDE's objective is "to increase productivity and reduce time-to-market by enabling development and using concepts closer to the problem domain at hand" [125].

MDE's core concepts are models and transformations (manipulation operations on models). According to Brambilla et al. [18], if one were to adapt Niklaus Wirth's equation [151], *Algorithms + Data Structures = Programs* to the MDE context, it would read: *Models + Transformations = Software*.

Model-driven engineering strictly adheres to the "everything is a model" philosophy. This goes beyond pure development activities and encompasses other model-based tasks of a complete software engineering process. To practice MDE, one often uses an IDE that supports defining models and transformations and compilers or interpreters that can make the final software artifacts [18].

2.3.3 Modeling Languages

Modeling languages are one of the principal components of MDE. A modeling language lets designers specify the models for their systems in graphical or textual representations. Modeling languages are formally defined, and designers must comply with their syntax when modelling [18]. Modeling languages can be classified as [18]:

- **Domain-Specific Modeling Languages (DSMLs)** - designed specifically for a certain domain, context, or company to ease the description of things in that domain
- **General-Purpose Modeling Languages (GPMLs)** - represent tools that can be applied to any sector or domain for modeling purposes

It is also possible to distinguish modeling languages according to their level of abstraction.

The distinction between static and dynamic models also highlights the importance of having different viewpoints on the same system. Multi-viewpoint modeling is a cornerstone of MDE, leading to the building of various models to describe the same system. These multiple models may use different modeling languages. Although it is possible to define a design composed of models in several independent languages, it is more convenient to explore a suite of languages that have a common foundation and are aware of each other. Thus, GPMLs typically include several coordinated notations that complement each other. These GPMLs are known as Modeling Language Suites (MLSs). The most known example of an MLS is the Universal Modeling Language (UML) [18].

In this case, we have defined a DSML to model CI/CD pipelines. These pipeline models will be descriptive and dynamic, as they represent the behavior of the CI/CD pipelines. This will allow us to migrate pipelines between CI/CD platforms by focusing on what they do instead of platform-specific elements.

2.3.4 Meta-Modeling

A natural extension of the definition of objects as instances of a model is to define the models themselves as instances of more abstract models. The more abstract models, called meta-models, highlight the properties of models themselves and, in a practical sense, constitute the definition of a modeling language, as they provide the capability of describing the whole class of models that can be represented by that language [18].

Following this chain of reasoning, one can create models that describe objects, meta-models that describe models, and meta-meta-models that describe meta-models. While it is theoretically possible to define infinite levels of meta-modeling, it has been shown that meta-meta-models can be defined based on themselves, providing little benefit to going beyond this level of abstraction [18].

While, when referring to an object in MDE, we say that an object is an instance of a model, when referring to a model, we say it conforms to a meta-model [18].

Meta-models can be used for [18]:

- defining new languages for modeling or programming
- defining new modeling languages for exchanging and storing information
- defining new properties or features to be associated with existing information (metadata)

This work will use a meta-model to represent CI/CD pipelines in a platform-independent manner, providing an abstraction from the CI/CD pipeline implementations of specific platforms. This platform-independent meta-model will serve as a grammar for our DSML.

2.3.5 Model Transformations

Besides models, transformations are the other crucial ingredient of MDE. They allow for the definition of mappings between different models [18]. According to Sendall and Kozaczynski, model transformations are "the heart and soul of model-driven software development" [125].

While transformations are applied at the model level, they are defined at the meta-model level.

Transformations can be further classified as model-to-model (M2M) transformations, model-to-text (M2T) transformations, and text-to-model (T2M) transformations.

Model-to-Model Transformations

Generally, M2M transformations take one or more models as input and return one or more models as output. In most cases, one-to-one transformations are sufficient, with one input and one output model. However, there are also situations where many-to-one, one-to-many, or many-to-many transformations are required, like merging models [18].

M2M transformations can also be classified as endogenous or exogenous. Endogenous transformations are transformations between models conforming to the same meta-model, while exogenous transformations are between models conforming to different meta-models [96, 18]. Endogenous transformations are also referred to as *rephrasing*, and exogenous transformations as *translating* [96, 140].

Examples of endogenous transformations include optimization, refactoring, simplification/normalization, and component adaptation. Exogenous transformations can be synthesis, reverse engineering, or migration [96].

Endogenous M2M transformations can be made *in-place*, where the input and output model are the same, or *out-place*, where the output model is created from scratch. By definition, exogenous transformations can only be *out-place* [96].

We use one-to-one exogenous and endogenous M2M transformations as part of a pipeline that transforms a platform-specific pipeline model into a platform-independent model and then into a different platform-specific model. We also use many-to-one transformations to merge platform-independent models as part of an effort to help developers consolidate pipeline technologies.

Text-to-Model and Model-to-Text Transformations

Text-to-model and model-to-text transformations automate the derivation of models from text and text from models respectively [18].

Text-to-model transformations are used in reverse engineering to obtain a higher-level system representation. By using them, it is possible to parse text, like code, into a model [18].

Text-to-model transformations depend greatly on the complexity of both the model being generated and the grammar of the text that is being parsed. For simple grammars and models, T2M transformations can be executed using embedded translation, i.e., generating the model directly from the text parser. More complex grammars and models may require multi-step transformations.

The primary purpose of M2T transformations in MDE is code generation. MDE's objective is to obtain a working system out of models. Since current execution platforms are mostly code-based, this implies transitioning the model to code level. M2T transformations can also generate other artifacts, like test cases or deployment scripts [18].

When implementing M2T transformations, there are several points to consider. M2T transformations vary in how much code is generated since it may not be possible to generate the code entirely from the model, what code is generated, generally code in high-level languages is preferable for better readability, and how the code is generated since there are multiple technologies to accomplish this [18].

Code generation can be described as a vertical transition from models with a higher level of abstraction to lower-level artifacts. These different levels of abstraction imply a gap that must be filled since not all specifics of the underlying platforms may be representable in the models [18].

This missing information has to be filled by the modeler using model augmentations, applying the convention-over-configuration principle for code generation, or leaving the specification open on the model level and entering the details at the code level [18].

These three approaches each have their pros and cons. Model augmentations allow detailed tweaking of the derived implementations by spending more effort preparing the models for code generation. The convention-over-configuration approach forgoes this effort, but the derived implementations can only be optimized at the code level. There is space for a hybrid of the model augmentation and convention-over-configuration approaches. The last approach implies only partial implementation generation, meaning the developer must complete the code themselves, which could lead to inconsistencies [18].

We use T2M and M2T transformations to integrate our transpiler directly with the CI/CD platforms by working with text files in their DSLs.

2.3.6 Model-Driven Software Reengineering

Chikofsky and Cross define software reengineering as "the examination and alteration of software systems to reconstitute it in new form and the subsequent implementation of the new form" [26, 92]. The process starts with the source code of the current system and ends with the source code of the new system. This process can involve just translating code from one language into another or also redesigning and determining the requirements in legacy systems, comparing them to new system requirements, and removing unneeded elements [92, 120].

Software reengineering involves a set of subpractices, namely, forward and reverse engineering, re-documentation, restructuring, and translation [146, 92]. Reverse engineering is "the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction" [120]. Forward engineering is the traditional approach to software development, starting with the conceptual design of a system and moving down through the abstraction levels until we have an implementation [92, 120].

In essence, software reengineering implies abstracting a system's implementation into a higher-level representation of that system (reverse engineering), applying transformations to that representation, and refining that representation into a new implementation (forward engineering). These steps map well to previously discussed MDE concepts: the higher-level system representation is a model, reverse engineering is a T2M transformation, changes to the model are M2M transformations, and forward engineering is an M2T transformation. MDE approaches to software reengineering have been gaining traction, as they can automate a significant part of the process [49, 13, 117, 108, 137].

This work will involve reengineering CI/CD pipelines using as a basis the CI/CD pipeline meta-model mentioned before. T2M transformations will generate a CI/CD pipeline model instance, those pipeline models will undergo M2M transformations, and M2T transformations will allow code generation in a different CI/CD platform.

2.3.7 Model-Driven Engineering Technologies

Model-driven engineering relies on technologies that support the development of models, their transformation, and their integration in the SDLC [18]. Jácome-Guerrero et al. list various MDE technologies and categorize them according to the elements of MDE they address [79]. Those are listed in the following sections.

Meta-Modeling Languages

Meta Object Facility (MOF) A language intended to model classes provided by the Object Management Group (OMG). Due to its complexity, a subset of MOF is understood to be enough for most use cases [98].

Essencial MOF (EMOF) A subset of MOF with less complexity, also provided by the OMG [98].

Ecore A part of the meta-modeling architecture of the Eclipse Modeling Framework (EMF), which provides the Eclipse IDE with meta-modeling capabilities. Ecore is a meta-model language that provides object-oriented concepts for creating meta-models and a subset of MOF [41].

Object Constraint Language (OCL) Used to define rules to determine if a model is well-formed. It can be used with any MOF meta-model [106, 141].

MetaDepth Modeling language that supports an arbitrary number of meta-levels. This makes it useful for defining multi-level languages [97].

Development Environments

OMG Meta-modeling Architecture Includes both MOF and UML since UML is defined with MOF [98].

Eclipse/Eclipse Modeling Framework The modeling project intended to support MDE in Eclipse. The EMF provides the basic mechanisms to handle meta-models [67].

Epsilon A family of scripting languages and tools for automating common model-based software engineering tasks [44].

Graphical Editors

Eclipse Graphical Modeling Framework (GMF) An Eclipse plugin that supports the development of graphical model editors from meta-models [15].

Graphiti Used to create highly sophisticated editors with support for EMF [147].

Sirius An Eclipse project that allows the development of graphical modeling editors, leveraging Eclipse technologies like EMF and GMF. It is highly adaptable and supports blended modeling. Relies on Acceleo and other projects to facilitate establishing relationships between model data and its graphical representation [126].

Textual Editors

Xtext An EMF-based framework to create textual modeling languages. It associates a textual representation with the meta-model. It provides mechanisms that allow the editing and manipulation of textual models. Includes a language for defining grammar and an application programming interface (API) for defining different aspects of a DSL. Automatically generates a parser, static analyzer, code formatter, code generator, etc., for the defined DSL [154].

EMFText An Eclipse plugin that allows defining a textual syntax for an Ecore meta-model. It can generate code without any EMFText dependencies. Allows automatic generation of default syntax and complete analysis of syntax to warn of potential problems [43].

Textual Concrete Syntax (TCS) A component of Eclipse Generative Modeling Technologies that allows the specification of a textual syntax for DSL, attaching syntactic information to meta-models. Offers an Eclipse editor that supports syntax highlighting, a schema, and hyperlinks per each DSL syntax represented in the DSL [134].

Model Transformations

Atlas Transformation Language (ATL) A hybrid model transformation language that implements imperative and declarative paradigms. An ATL transformation is composed of rules defining how the target model elements are created and initialized from the elements of the source model [105].

Query/View/Transformations (QVT) A family of languages the OMG provides that allow for the definition of transformations. There are two end-user languages, QVT Operational Mappings and QVT Relations, and a low-level language, QVT Core [1].

Epsilon Transformation Language (ETL)/Epsilon Merging Language (EML) A part of the Epsilon family of scripting languages to interact with models. ETL and EML are used to define M2M transformations in a rule-based and modular manner. EML can merge various models. To achieve this, the models must first be compared using the Epsilon Comparison Language (ECL). Elements determined to be matching are then merged according to the defined rules [102, 101].

Table 2.1: DevOps phases proposed by Zhu et al. [157].

Phase	Description
Development	Involves planning and developing software.
Integration	Core of the DevOps lifecycle. Committing new changes to the source code.
Testing	Automatic testing tools.
Monitoring	Performance monitoring and recording of the application.
Feedback	Gathering, analyzing, and using clients' software usage feedback.
Deployment	Deployment of the code to the production environment.
Operations	Automating all release operation processes.

Code Generation Languages

Acceleo An implementation of OMG's MOF-to-Text standard, part of the Eclipse M2T project. Allows easy code generation, high personalization capacity, interoperability, and traceability management. Uses a template mechanism [2].

Java Emitter Templates A tool to generate output files from an input model using templates [143].

Xpand Can generate code based on DSL models defined with Xtext. It is a statically typed template language [142].

Epsilon Generation Language (EGL) A template-based model-to-text language for textual artifacts from models [45].

2.4 Related Work

As part of the same ongoing effort to improve CI/CD developer experience using MDE as this work, Gião et al. conducted a systematic literature review (SLR) of model-driven approaches to DevOps [53]. The SLR finds relevant papers and categorizes them according to the phases of DevOps they cover. These phases, proposed by Zhu et al. [157], are Development, Integration, Testing, Monitoring, Feedback, Deployment, and Operations. More detail about the phases can be found in table 2.1.

From the relevant papers identified by Gião et al., only the ones about the Integration, Testing, Deployment, and Operation phases were analyzed. This was because attempts to model the other DevOps phases, Development, Monitoring, and Feedback, were irrelevant to this study's focus, CI/CD pipelines.

Colantoni et al. [32] propose DevOpsML, a modeling language for DevOps platforms and processes to support documentation implemented with EMF. In their work, they recognize the increasing interest in the integration of DevOps and MDE practices in low-code engineering platforms (LCEPs) and the tension between the often non-technical LCEP users and the current DevOps processes that are considered on a more technological level.

DevOpsML uses three meta-models: a platform meta-model to define platforms by their tools and interfaces; a process meta-model that describes DevOps processes like CI/CD pipelines; and

a linking meta-model, capable of linking process to process, platform to platform, and process to platform.

This work is related to ours, as Colantoni et al. create a meta-model that can describe the DevOps process. However, DevOpsML differs from the meta-model used in this work because it tries to model DevOps itself instead of CI/CD pipelines. This increased abstraction and scope would make it harder to directly execute transformations from PSMs to a PIM. This is not an issue for Colantoni et al., as the initial version of DevOpsML is intended to support documentation.

El Khalyly et al. [42] propose a DevOps meta-model and an Internet-of-Things (IoT) meta-model. The DevOps meta-model has a high degree of abstraction, while the IoT meta-model is more detailed. The authors claim a dependency from the IoT meta-model to the DevOps meta-model, as DevOps tools are "in service of Internet of Things ecosystem to guarantee the continuous integration, delivery and deployment of programs".

El Khalyly et al.'s work is related to ours as their DevOps meta-model addresses CI/CD pipelines. However, their meta-model has a much higher degree of abstraction. This is because El Khalyly et al.'s meta-model is part of a broader effort to standardize IoT systems, while ours is the basis for a CI/CD pipeline reengineering process.

Melchor et al. [95] present a model-driven framework for defining data science pipelines independent of a particular execution platform and tools implemented with EMF. This framework separates the pipeline definition into two different modeling layers: a conceptual layer, where a data scientist specifies all the data and operations to be carried out by the pipeline, and an organizational layer, where a data engineer can specify the execution environment where the operations will be implemented. This approach allows the usage of different tools, which improves replicability, the automation of process execution, improving reproducibility, and the definition of model verification rules, providing intentionality restrictions.

This paper is relevant as Melchor et al. model data science pipelines, a topic adjacent to CI/CD pipelines, in a platform-independent manner. Like them, we use two modeling layers to represent pipelines. However, their modeling layers, conceptual and organizational, describe what the pipeline does and the infrastructure it is executed on, respectively. As such, both meta-models have a similar level of abstraction, and there are no transformations between models conforming to them. Our modeling layers, platform-specific and platform-independent, seek to represent the pipeline wholly, and we execute vertical transformation from platform-specific to platform-independent and vice-versa.

Van den Heuvel et al. [71] introduce ChainOps, a model-driven DevOps approach for the blockchain. They focus on modeling smart contracts, computations in the form of executable code that promise to simplify trade ecosystems where parties may remain anonymous. They claim their model addresses concerns with trustworthiness, enables non-technical end-users, and reduces blockchain environment lock-in. They can apply M2T transformations to generate code for various blockchain technologies. ChainOps is based on the AstraKode Blockchain Modeler, a software-as-a-service modeling platform for blockchain technologies.

Van den Heuvel et al.'s work has a similar goal to ours. Both seek to use modeling to enable users with less platform-specific knowledge, reducing platform lock-in. Like us, Van den Heuvel et al. also do this by generating platform-specific code from a higher-abstraction-level model through M2T transformations. The approaches differ as Van den Heuvel et al. focus only on blockchain projects and CD. We seek to model CI/CD for any project.

Colantoni et al. [33] present a work in progress for modeling CD pipelines based on JavaScript Object Notation (JSON). This work is separate from their aforementioned DevOpsML [32].

They use a previously developed approach, JSONSchemaDSL, to semi-automatically generate JSON-based DSLs specified through a JSONSchema. JSONSchemaDSL generates an EMF meta-model, an Xtext grammar, and a Sirius graphical representation. With this, Colantoni et al. allow blended modeling, "the activity of interacting seamlessly with a single model through multiple notations, allowing a certain degree of temporary inconsistencies" [27], of JSON-based DSLs.

This approach is applied to Keptn, a CD platform, through its JSON-based DSL, Shipyard, leading to the modeling of CD pipelines.

Colantoni et al.'s work is the most similar to ours, as JSONSchemaDSL can generate meta-models for CI/CD pipelines of JSON-based CI/CD platforms. However, these meta-models are all platform-specific. Our approach includes meta-models like the ones Colantoni et al. generate, but it also includes platform-independent ones that serve as the basis for pipeline migration. This cannot be done using JSONSchemaDSL.

Düllmann et al. [39] propose a model-driven DSL-based CI/CD pipeline definition and analysis framework. Their work involves the creation of a meta-model for the Jenkins pipeline language. The DSL is aimed at facilitating interoperability and transformation between different formats. Through their approach, the authors analyzed 1,000 publicly available Jenkins files and successfully represented 70% of those files without any loss of information.

In contrast, our PIMM is not specific to a CI/CD language and was designed to abstract away from the intricacies of individual platforms. Our Jenkins meta-model is also more detailed. Furthermore, we tested our PIMM for its ability to represent CI/CD pipelines and for tasks extending beyond mere representation, such as reengineering pipelines across platforms.

Pulgar et al. [115] introduce a meta-model heavily influenced by GHA. Their goal is to ensure that each modification to a pipeline is valuable. To validate their approach, the authors utilized three open-source projects. Additionally, the authors created justification diagrams intended for sharing with the development team.

In contrast, our PIMM offers greater abstraction from specific CI/CD tools and encompasses more features than those of the authors. Moreover, we conduct different types of validations compared to Pulgar et al., as our primary focus lies in utilizing our meta-model to reengineer and develop pipelines.

Ferry et al. [47] present ENACT, a model-driven DevOps framework for trustworthy smart IoT systems. ENACT includes a continuous delivery toolkit with two enablers: an orchestration and continuous deployment enabler and a test emulation and simulation enabler. The first enabler has

a DSML that can support the automatic deployment of software components over IoT, edge, and cloud resources.

Babar et al. [9] model DevOps deployment choices to enable enterprises to devise a DevOps approach suitable to their requirements while considering possible process reconfigurations. Their approach enables the modeling of trade-offs of alternative deployment options [16]. Babar et al.'s work allows modeling CI/CD pipelines, but only in a very high-level manner.

Bordeleau et al. [16] identify requirements of a modeling framework for DevOps through a case study. This framework would be composed of processes, methods, and tools. They identify general, description, analysis, and simulation (to support continuous framework improvement) requirements. The modeling framework envisioned by Bordeleau et al. would serve to support organizations in putting DevOps processes into practice.

Wurster et al. [153] propose the Essential Deployment Meta-Model to enable a common understanding of declarative deployment models by facilitating the comparison, selection, and migration of platforms.

Kumar and Goyal [87] propose ADOC, a conceptual model for automated DevSecOps, DevOps embedded with security controls providing continuous security assurance, for OSS over the cloud. ADOC is based on a continuous security conceptual framework described in the article. There are three components to ADOC: the ADOC Engine, an end-to-end automated workflow with a set of practices and embedded security assurance controls; the OSS suite, the propellant for this ADOC Engine; and the cloud infrastructure and technologies to power ADOC.

These works are related to ours, as they attempt to use MDE to improve the user experience of DevOps processes like CI/CD. However, they go about this by supporting organizations in making DevOps-related decisions. Our work aims to improve the DevOps experience by simplifying the migration process between CI/CD platforms. As such, we attempt to model CI/CD pipelines themselves instead of more abstract DevOps processes.

TOSCA [104, 148], short for Topology and Orchestration Specification for Cloud Applications, is an emerging standard. Its main goal is to enhance the portability and management of cloud applications. It can be used to model and automate DevOps for cloud applications.

MODAClouds [37] is a European project undertaken to simplify cloud services using MDE. One of its goals is to support developers in building and deploying applications to multi-clouds across the full cloud stack. MODAClouds includes MODACloudML, a set of DSLs to support the design of multi-cloud applications with guaranteed quality of service.

MELODIC [74] allows modeling, deploying, and optimizing multi-cloud applications. The application is modeled using the Cloud Application Modelling Execution Language (CAMEL) and business process models. The CAMEL application model is transformed into a Constraint Programming Model for mathematical optimization of the deployment [33].

MORE [25] is a model-driven approach to automate a system's initial deployment and dynamic configuration. MORE includes a model to specify the high-level view of a system in the form of a desired deployment topology. This topology is then transformed into executable code for Puppet,

an infrastructure deployment and management platform [116], to get virtual machines, physical machines, and containers [123].

TOSCA, MODAClouds, MELODIC, and MORE are related to our work as they can model deployment configurations for cloud projects, an element of CI/CD. These approaches are also capable of generating deployment code from the models. Our approach differs from these as we also attempt to model continuous integration. We also want our approach to be viable for non-cloud applications. As such, we base our PIM on different CI/CD platforms.

Sandobalin [123] develops a DSL called ARGON that can model cloud infrastructure; the model then allows the generation of scripts of different configuration management tools for CD through model-to-text transformations. Sandobalin's infrastructure meta-model abstracts the capabilities of cloud computing.

Ketfi and Belkhatir [84] propose DYVA, a unified framework for dynamic deployment and reconfiguration of component-based software systems. The framework is based on their hierarchical meta-model, a PIM, that provides an abstract view of a component model. Then, this meta-model can be personalized into a specific component model. Changes in this personalized model, a PSM, trigger the deployment or the reconfiguration process.

Casale et al. [21] propose using MDE to support QA in data-intensive software systems. Their tool, DICE QA, would be capable of modeling big data applications. DICE QA covers simulation, verification, and architectural optimization. DICE could also generate code and performance, reliability, and safety models.

Kirchhof et al. [85] present MontiThings, a modeling infrastructure for systematic engineering of IoT applications. MontiThings is an extension of MontiArc, an architecture description language.

Song et al. [129] create a model-based tool, GeneSIS, to generate deployment plans for IoT devices without human interaction. The tool takes a PSM as an input and transforms it into a PIM. Using the PIM and constraint solving, the tool then assigns deployment plans to devices in the PIM. Lastly, the PIM is transformed into the PSMs necessary for device deployment.

Hugues et al. [75] propose TwinOps, a process that unifies model-based engineering, digital twins, and DevOps in one workflow that can be used to improve the engineering of cyber-physical systems. DevOps practices are combined with model-based code generation to facilitate deployments.

Meyers et al. [99] present an MDE framework that supports continuous testing and fast development iterations in safety-critical systems. Their framework is based on two DSLs. The first is a formal modeling language, the Functional Safety Formal Model (FuSaFoMo), that allows engineers to build a formal architectural description of a system. The second is a contract-based requirements language that can link to FuSaFoMo and specify system behavior. Test cases can be generated from these contracts to verify system behavior.

Rivera et al. [119] propose Urano, a tool for automating the deployment process that uses UML to specify software architecture and the deployment process. From graphical UML specifications, Urano generates textual specifications in Amelia, a DSL specifically conceived for specifying and

executing deployment workflows for distributed software systems. The Amelia specification can then be compiled into a Java-based application that automatically performs deployment operations [82].

Silva et al. [35] propose OpenTOSCA for IoT, a TOSCA-based system to deploy IoT applications fully automatically. This approach requires low-level technical details to function. Silva et al.'s work is an approach to CD in IoT systems.

Wettinger et al. [148] present a methodology for using TOSCA to describe DevOps artifacts by crawling knowledge repositories. From the TOSCA description, Wettinger et al. can deploy application topology using OpenTOSCA.

Ribeiro et al. [118] present a model-based solution to deploy software in the cloud automatically. Their solution uses UML models. After all required deployment information about the cloud provider and repository has been input into the model, the solution generates an automatic deployment. The deployment code is written in Ruby and for Chef, an infrastructure deployment platform [23].

Artač et al. [7] propose DICER, a framework to support the model-driven continuous design and deployment of data-intensive applications. DICER's architecture comprises two main components: the Modeling Environment and the Deployment Service. In the Modeling Environment, the user specifies infrastructure elements. This model is transformed into a TOSCA-compliant model and then a deployable TOSCA YAML blueprint. The Deployment Service receives the TOSCA YAML blueprint and handles it using Chef.

Alipour and Liu [4] propose a model-driven methodology to facilitate multi-cloud deployment of applications. They use abstraction levels: platform-independent and platform-specific. Model transformations transform the platform-independent model into platform-specific ones and then into deployment configurations. Alipour and Liu use this to create a consistent auto-scaling strategy across multiple cloud platforms.

Brabra et al. [17] propose a model-driven methodology to automatically transform TOSCA models of cloud resources and their orchestration into artifacts of specific DevOps platforms like Docker. Casale et al.'s RADON project [20] seeks to develop a model-driven DevOps framework for creating and managing microservice-based applications. Challita et al. [22] analyze the conceptual similarities between TOSCA and the Open Cloud Computing Interface, which focuses on standardizing an API for infrastructure-as-a-service providers and create a tool to fully deploy and manage cloud applications based on the two studied technologies. Ferry et al. [46] propose CloudMF for model-driven management of multi-cloud applications through the use of a DSL. Guerriero et al. [68] introduce SPACE4Cloud, a DevOps integrated environment for model-driven design-time quality of service assessment and optimization, and runtime capacity allocation of Cloud applications. Weerasiri et al. [144] present a model-driven framework for interoperable cloud resources management. This is done through high-level domain-specific models that describe elementary and federated cloud resources and a pluggable architecture to transform these into lower-level resource descriptions and management rules. In a separate work, [145], Weerasiri et al. also propose a visual notation for representing and managing cloud resources.

These works are related to ours as they use MDE to facilitate the deployment or testing of applications, elements of CI/CD. Many also involve reengineering, using methodologies similar to ours by having two modeling levels, platform-specific and platform-independent.

However, they differ from ours in that they attempt to model the architecture of the applications themselves and then generate deployment configurations for them. We are not directly concerned with application architecture as we model already existing CI/CD pipelines. This difference also reveals itself in the platforms we support, as we focus on GHA, CircleCI, and Jenkins. Overall, the models they use are static, while ours are dynamic.

Chapter 3

Understanding CI/CD Usage in Practice

In this chapter, we seek to better understand CI/CD usage in practice. This will help us understand how common platform migrations are and what are the most relevant platforms for the PIMM to support. To achieve this, we designed a study that involved mining public software repositories hosted on GitHub over a period of almost 12 years.

GitHub is the most widely used version control and software hosting service [152]. As of 2023, over 100 million developers used GitHub, and the platform hosted over 284 million public repositories [88]. GitHub also provides an API to collect information from its software repositories. With this API, it is possible to search repositories inside GitHub using parameters such as keywords in their name and README, size, number of stars, followers, and forks [124]. The insights collected using the API help understand various developer behaviors and have been used in several studies [69, 70, 65, 90, 19, 57].

In section 3.1, we present the methodology used for this work. This methodology includes how we found the CI platforms for this study, what information we have collected from the repositories and how it is related to our research questions, how we used the GitHub API to collect information from the repositories, and how we organized the data collected from the said repositories. Section 3.2 showcases and analyzes the mining results. Section 3.3 presents several works related to this one. Those works focus on mining information about several DevOps aspects in various software repositories. Section 3.4 discusses the threats to the validity of our work. Finally, section 3.5 presents the study's conclusions.

3.1 Methodology

This section details the process followed to collect the data for the analysis. Figure 3.1 presents an overall view detailed in the following paragraphs.

3.1.1 Collecting Repositories Using the GitHub REST API

The initial data collection phase involved assembling a representative sample of GitHub repositories reflective of real-world projects. To achieve this, school projects and smaller repositories

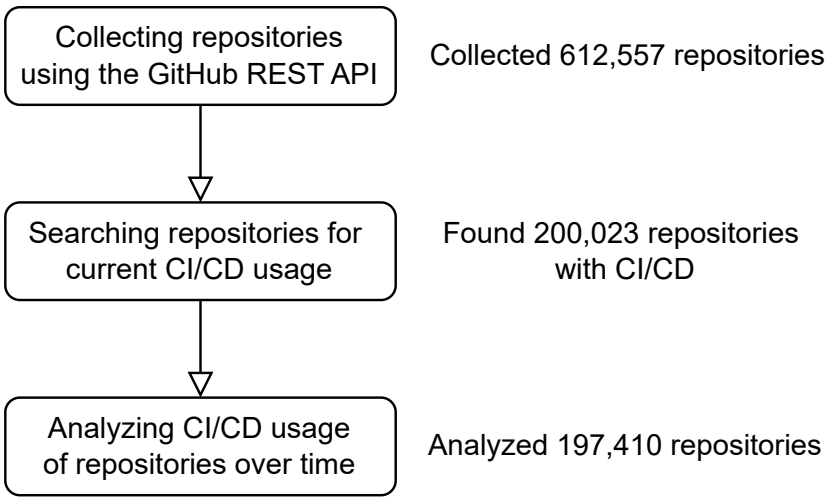


Figure 3.1: Data collection process.

were excluded, employing a methodology inspired by prior research projects [31]. That is, the focus was on repositories with a certain level of popularity. Ultimately, only repositories with 10 or more stars were collected.

The sampling process encompassed retrieving 1,000 repositories every week from January 1, 2012, to October 18, 2023. This timeframe was selected because the concept of DevOps gained prominence around that period. This assertion is supported by the emergence of the first comprehensive survey on the state of DevOps by Puppet Labs in the same year [109].

Afterward, we removed duplicated repositories from the dataset. This approach yielded 612,557 repositories spanning the specified date range. The entire dataset of repositories is provided for reference [59]. The source code used to create this dataset is also available [54].

3.1.2 Searching Repositories for Current CI/CD Usage

Following the collection phase, we sought to discern the platforms employed in each repository. We used a list of 61 CI/CD platforms curated by the Cloud Native Computing Foundation [133], established by the Linux Foundation, to determine which CI/CD platforms we would search for.

Identifying CI/CD Platforms

We identified pertinent artifacts and patterns for each platform in the list, enabling automatic recognition of repositories utilizing those platforms. Each platform was identified using one of two heuristics: *i)* some platforms use files with a particular extension; *ii)* for the others, specific types of files (e.g., YAML files) had to be inspected for content specific to the underlying platform.

From this process, we divided the 61 into four different categories: *i)* 39 platforms that could be identified (table 3.1); *ii)* 10 which could not be identified since there was not a clear artifact to use (table 3.2); *iii)* 4 platforms that required the use of specific code to be identified as they

are libraries embedded in the code (table 3.3); and *iv)* 1 deprecated platform with no current documentation that made it impossible to recognize (table 3.4).

Table 3.1: CI/CD platforms that can be identified and analyzed.

Agola	AppVeyor	ArgoCD	Bytebase
Cartographer	CircleCI	Cloud 66 Skycap	Cloudbees Codeship
Devtron	Flipt	GitLab	Google Cloud Build
Helmwave	Travis	Jenkins	JenkinsX
Keptn	Liquibase	Mergify	OctopusDeploy
OpenKruise	OpsMx	Ortelius	Screwdriver
Semaphore	TeamCity	werf	Woodpecker CI
GitHubActions	Codefresh	XL Deploy	Drone
Flagger	Harness.io	Flux	GoCD
Concourse	Kubernetes	AWS CodePipeline	

Table 3.2: CI/CD platforms that cannot be identified due to the lack of clearly identifiable artifacts.

Akuity	Bamboo
Buildkite	Bunnyshell
CAEPE	Keploy
Northflank	OpenGitOps
Ozone	Spacelift

Table 3.3: Libraries introducing unnecessary complexity.

Brigade	k6
OpenFeature	Unleash

Table 3.4: Deprecated platforms lacking documentation.

D2iQ Dispatch

Searching Repositories for CI/CD Platform Artifacts

We analyzed the latest available commit of all 612,557 repositories retrieved in section 3.1.1.

This analysis involved retrieving the complete file tree of each repository as well as the contents of any file that could include content that identified a CI/CD platform being used.

With this data, we searched for the 39 CI/CD platforms we could identify using the previously described methodology. We found a total of 200,023 repositories using one or more CI/CD platforms in their latest commit. The comprehensive dataset containing the repositories, the SHA of the commit we analyzed, and the corresponding platforms is also available for further examination [58]. The code used to create this dataset and the figures generated with the data are also available [54].

3.1.3 Analyzing CI/CD Usage of Repositories Over Time

Having identified 200,023 repositories with CI/CD, we investigated their CI/CD usage over time. This involved retrieving snapshots of the repositories at a given time interval.

These snapshots are retrieved using commits in the repository. For each commit, we retrieve the required data to identify CI/CD platforms (as described in section 3.1.2).

For this analysis, we discarded any repositories that were created after July 16, 2023 (so they were older than 90 days). This gave us a sample of 197,504 repositories.

Determining the Snapshot Interval

To determine the ideal sampling interval for the snapshots, we ran a test on a random sample of 10,000 repositories (from the 197,504 repositories with CI/CD and older than 90 days). We retrieved snapshots at 90-day, 180-day, and 365-day intervals for each repository of the 10,000 repositories. Each snapshot was then searched for CI/CD platforms.

The goal was to determine the number of changes in the CI/CD platform stack that would be lost by increasing the retrieval interval, a change being any difference in the CI/CD stack compared to the previous snapshot. For the sample, there were 14039 stack changes at a 90-day sampling interval, 11958 changes at a 180-day sampling interval, and 9962 changes at a 365-day sampling interval. From a 90-day to a 180-day rate, there was a 14.8% decrease in the detected changes, and from a 180-day to a 365-day rate, there was a 16.7% decrease in detected changes.

Based on these results, the analysis used a 90-day sampling rate. A lower sampling interval, or retrieving all commits for each repository, was not feasible due to GitHub API rate limits and the time for the study.

Retrieving Snapshots of Repositories

For each selected repository, the first commit was retrieved. A sequential iterative process was employed, where the latest commit (if one existed) was retrieved for each 90-day interval starting from January 1, 2012, or the repository's first commit push date, whichever was later. This process continued until the repository's last update at the time of retrieval. All commits were retrieved from the default branch of the repository.

We examined the snapshot commits using the same methodology described in section 3.1.2.

In this process, 19 repositories could not be analyzed because they had either been deleted or gone private in the time since we identified them. This left us with 197,485 repositories.

Cleaning the Data

After all repositories were processed, the retrieved data was cleaned. Any snapshots from before January 1, 2012, were discarded (we use the commit date for this instead of the date the commit was pushed to GitHub), and the last snapshot of each repository was set to the one used in the previous analysis. For some snapshots, the GitHub API could not return a file tree. Another

attempt was made to process these snapshots to eliminate any momentary API malfunction. Lastly, each snapshot's date and detected platforms were checked against the detected platforms' launch dates, and snapshots where a platform was detected before it was launched were removed. If a repository was left without snapshots at the end of these cleaning steps, it was discarded.

From an initial 197,485 repositories selected for temporal analysis, we finished with the CI/CD platform use history of 197,410. For the 75 repositories whose CI/CD platform history could not be cleaned, the reasons are as follows: another 39 repositories could not be cleaned because they had either been deleted or gone private, and 36 were discarded because they had no snapshots at the end of the cleaning steps.

The comprehensive dataset containing the repositories and the corresponding snapshots and platforms is also made available for further examination [60]. The code used to create this dataset and the figures generated with the data are also available [55].

3.2 Results

Having retrieved and cleaned repository data, we analyzed it to get a better perspective of CI/CD usage. In section 3.2.1, we detail CI/CD's increasing relevance. Section 3.2.2 details the usage of different CI/CD platforms and section 3.2.3 details how often repositories change CI/CD technologies.

3.2.1 The Increasing Relevance of CI/CD

Overall, we found an increase in CI/CD usage from 2012 to 2023. This can be seen in figure 3.2 where, even though a repository may use more than one CI/CD platform, we can detect a general increase in the number of active repositories using CI/CD over the years (an active repository is a repository with at least one commit in a given calendar year). Moreover, as seen in figure 3.3, CI/CD platforms are being integrated into the development workflow sooner as time goes on. The sharp decreases in 2022 and 2023 come from all analyzed repositories having at least one CI/CD platform in 2023.

This reveals an increase in the relevance of CI/CD over time, as not only are more projects using it, but they are also integrating it earlier into their lifetime. This decrease in time to CI/CD integration may also indicate that less complex and smaller projects are using CI/CD. This could mean these projects do not have a full-time DevOps developer, which would underline the need to make CI/CD pipeline changes more efficient, as the projects have fewer resources.

3.2.2 The Usage of Different CI/CD Platforms

Figure 3.2 shows two significant trends in CI/CD, Travis, and GHA. Travis usage steadily increased from 2012 until it peaked in 2019 with 73,284 repositories (36.6%). Since 2019, Travis's usage has been declining. This coincides with the rapid adoption of GHA; from 2019 to 2020, there was a 502.8% growth in the number of repositories using GHA, and from 2020 to 2021,

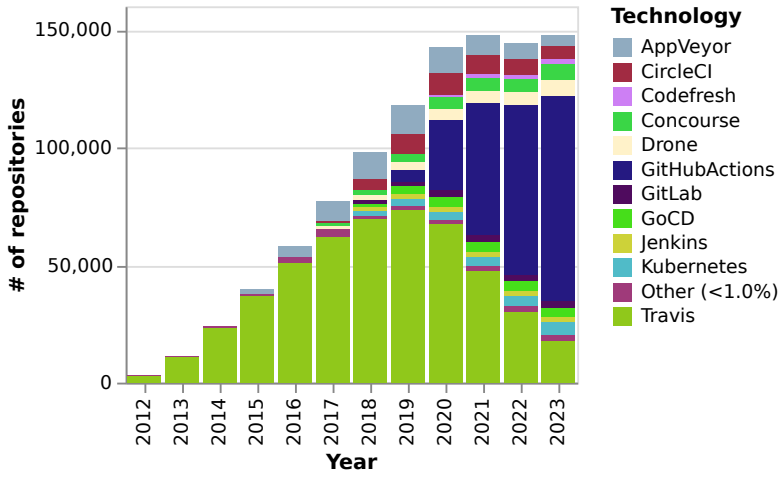


Figure 3.2: Number of active repositories where each platform was detected by year.

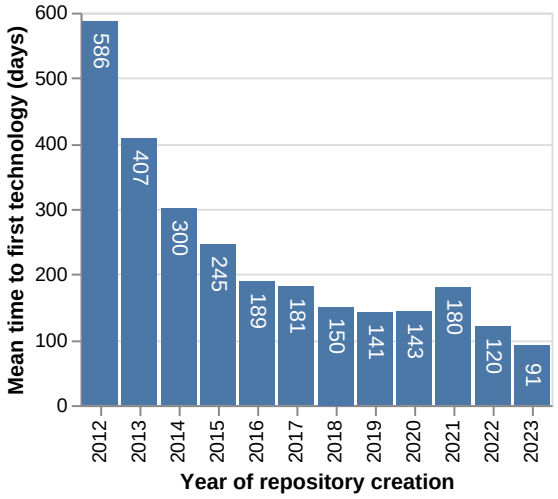


Figure 3.3: Mean time in days to first CI/CD platform detection by repository creation year.

there was an 86.6% growth. Of the 36,587 repositories that used Travis in 2019 and were still active in 2023, 59.7% were using GHA and not Travis in 2023, 21.1% used Travis and not GHA, and 16.9% used both. Of the 87,582 repositories using GHA in 2023, 45.4% had no snapshots from before 2020. The exodus from Travis and the influx from newer repositories have been the main drivers for GHA's growth.

Figure 3.4 shows the top 10 CI/CD stack transitions from repositories that solely used Travis in 2019. While many stopped being active, 22.9% moved from Travis to GHA. If we consider only the ones active, this means 53.7% of all active Travis projects moved to GHA.

A repository may use more than one platform at a time. As seen in figure 3.5, this is quite common. In the last 4 years, each year, more than 23,000 projects have included more than one platform, which accounts for about 20% of all projects (the platforms used in a repository in a given year are the union of the platforms of the snapshots retrieved in that year).

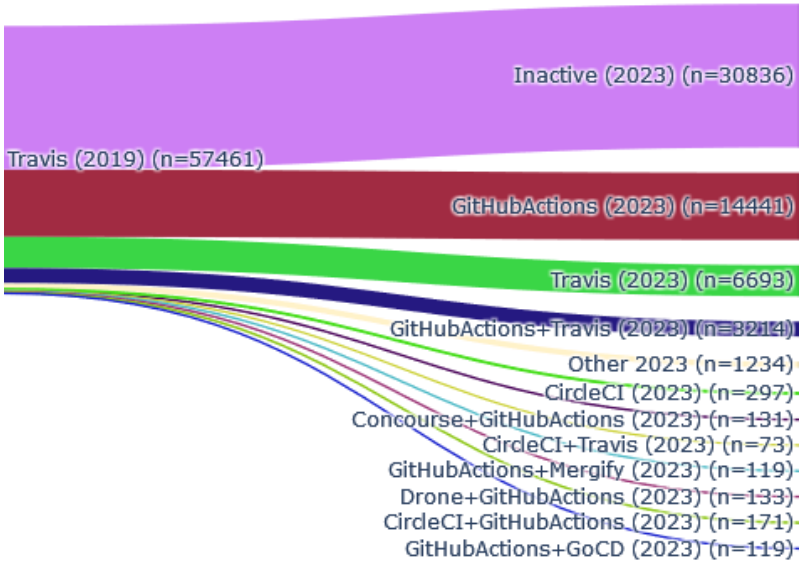


Figure 3.4: CI/CD platform stack transitions from repositories solely using Travis CI in 2019.

From this data, there is an observable migration in the CI/CD platforms used over the years. Taking into account Mazrae et al.'s reporting that one of the main reasons for migrating CI/CD platforms is to diminish co-usage [121], the significant number of projects co-using platforms could also represent a large number of users looking to migrate. This, too, substantiates the need to provide support to developers migrating CI/CD.

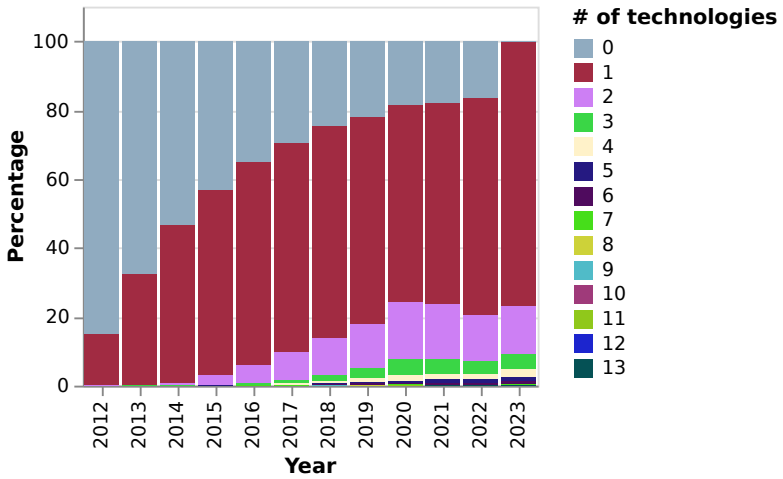
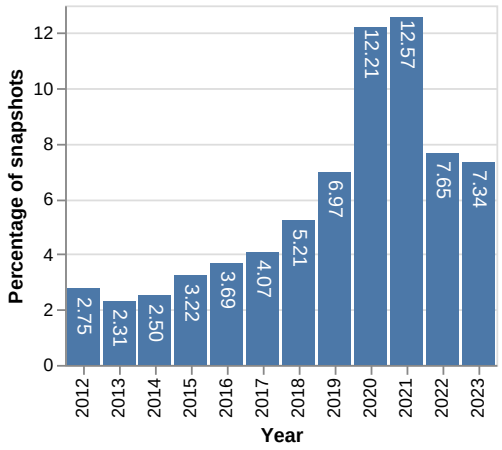


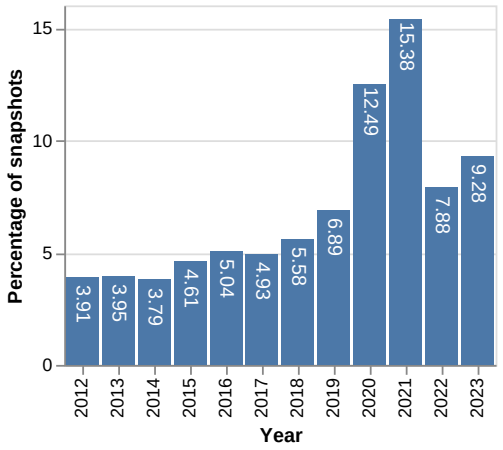
Figure 3.5: Percentage of active repositories using at most a given number of platforms in a given year by year.

3.2.3 The Change Rate of CI/CD Platforms

As figure 3.6a shows, the percentage of snapshots with CI/CD changes compared to the previous snapshot grows steadily from 2013 (2.3%) to 2019 (6.9%) and peaks in 2020 (12.2%) and 2021



(a) Percentage of snapshots with changes in the CI/CD platform stack from the previous snapshot by year (all repositories).



(b) Percentage of snapshots with changes in the CI/CD platform stack from the previous snapshot by year for the set of repositories active from 2012 to 2023 (n=8296).

Figure 3.6: Percentage of snapshots with changes in the CI/CD platform stack from the previous snapshot by year, considering all repositories (figure 3.6a) and the set of repositories active from 2012 to 2023 (figure 3.6b).

(12.6%), coinciding with GHA's explosive growth phase. Since 2021, this number has remained stable at almost 8%.

This is a very significant result since it shows that every year, between 2.3% (in 2013) and 12.6% (in 2021) of all snapshots include some change in the CI/CD platforms used. This represents a very significant amount of platform shift, with all the known issues with that. Moreover, it is essential to notice that this is constant over time, and there is no reason to think this may change in the near future. This means the research community can significantly contribute to aiding all these teams when they migrate and evolve their systems.

If we limit this analysis to a subset of active repositories from 2012 to 2023, that is, repositories for which we would retrieve CI/CD platform information for every year from 2021 to 2023 following the previously described methodology, we find interesting results. As figure 3.6b shows, similar trends can be observed but with higher change percentages over time. For this analysis, all snapshots for each repository before the first where we detected CI/CD platforms were discarded. This was done to eliminate a project's first choice of platforms from the analysis.

The percentage of CI/CD platform stack changes is significant and is consistently higher in long-running projects, meaning projects continuously look for platforms that better fit their workflow. Although one could expect some stability over the years for mature projects, this is not the case, as seen in the data. This reinforces the need to provide support for these changes.

3.3 Related Work

Xu et al. [155] introduce the idea of mining container image repositories for configuration and other deployment information of software systems. The authors also showcase the opportunities based on concrete software engineering tasks that can benefit from mining image repositories. They also summarize the challenges of analyzing image repositories and the approaches to address them.

Xu et al.'s work is related to this study as they, too, mine software repositories to discover information about the deployment of software systems. The works differ as they focus on the technologies used for deployment while we give a broader overview of the usage of CI/CD platforms.

Mazrae et al. [121] present a qualitative study of CI/CD platform usage, co-usage, and migration based on in-depth interviews. They identify reasons for using specific platforms, reasons for co-using CI/CD platforms in the same project, and migrations executed by the interviewees. Their study reveals a clear trend in migration from Travis to GHA.

Mazrae et al.'s work and ours are related as they seek to better understand the usage and migration of different CI/CD platforms. The studies differ in their methodology, but some of our study's conclusions align with Mazrae et al.

Goldazeh et al. [65] conduct a qualitative analysis of the usage of seven popular CI platforms in the GitHub repositories of 91,810 active Node Package Manager (NPM) packages having used at least one CI service over a period of nine years. Their findings include the fall of Travis, the rapid rise of GHA, and the co-usage of multiple CI platforms.

Decan et al. [36] study GHA use in nearly 70,000 GitHub repositories. They find that 43.9% of repositories use GHA workflows. They also characterize these repositories according to which jobs, steps, and reusable Actions are used and how.

Calefato et al. [19] study MLOps (DevOps but focused on machine learning projects) practices in GitHub repositoriesTheir preliminary results suggest that the adoption of MLOps workflows is somewhat limited.

Our research shares a similar objective to these studies and also mines GitHub repositories. However, we expand beyond the scope of the referenced papers. We conducted an analysis using a larger dataset comprising 612,557 repositories sourced from more diverse origins, encompassing a wider array of project types and programming languages. We also search for a wider array of CI/CD platforms. Moreover, our study was conducted more recently.

Zahedi et al. [156] present an empirical study exploring continuous software engineering from the practitioners' perspective by mining discussions from Q&A websites. The authors analyzed 12,989 questions and answers posted on Stack Overflow. The authors then used topic modeling to derive the dominant topics in this domain. They then identify and discuss critical challenges.

Zahedi et al.'s work relates to this study as they seek to better understand continuous software practices. However, the analyses differ in their approach, as we mine GitHub repositories

for empirical data and they analyze questions and answers on Stack Overflow, which are more subjective.

Liu et al. [90] mine 84,475 open-source Android applications from GitHub, Bitbucket, and GitLab to search for CI/CD adoption. They find only around 10% applications leverage CI/CD platforms, a small number of applications (291) adopt multiple CI/CD platforms, nearly half of the applications that adopt CI/CD platforms do not really use them, and CI/CD platforms are useful to improve project popularity.

Liu et al.'s objectives and approach are similar to ours. However, our analysis is done with a more significant sample of 612,557 repositories, and we do not limit ourselves to Android applications. We also delve into CI/CD platform migration.

Hilton et al. [72] studied CI by mining 34,544 OSS projects on GitHub and surveying 442 developers. They found many OSS teams that do not use CI. Of the ones that use CI, 90% used Travis. They find popular projects are more likely to use CI and that the median time for CI adoption is one year. Beller et al. [12], through an analysis of GitHub, found that Travis had seen a sharp increase in usage up to 2017, being used by one-third of popular projects. Vasilescu et al. [139] studied 1,884 GitHub projects in 2014. They found Travis usage in 918 repositories (48.7%).

These studies all share similar goals to ours as they investigate the landscape of CI/CD usage. Our findings, from the low CI/CD usage rate in the early to mid-2010s to the early dominance of Travis, align with theirs. However, their studies are all dated, the most recent being from 2017.

Widder et al. [150] conducted a qualitative study of 7,276 GitHub projects that had migrated away from Travis. They found that a project's dominant language is an important predictor for Travis abandonment. They also found more complex projects were less likely to migrate from Travis. In a follow-up [149], they investigate the pain points of Travis.

Widder et al.'s works are related to ours as they address one of the most significant shifts in CI/CD platform usage we discovered, the exodus from Travis CI. Otherwise, the studies differ in goals and methodology.

Lamba et al. [89] study the spread of CI/CD in NPM package repositories. Their analysis is done through repository badges, a recent innovation on code hosting platforms. They search for 12 CI/CD platforms in 168,510 NPM package repositories hosted on GitHub. Their study focuses on how CI/CD platforms gain market share.

While our works are related, as they both study CI/CD usage in repositories, they have key differences. Lamba et al. analyze repository badges while we analyze commits. We also have a broader scope as we study more repositories and more platforms.

There are several industry resources on the usage of CI/CD. JetBrains [135] provides results from yearly surveys of developers about the developer ecosystem from 2017 to 2023. Their results show Jenkins as the most popular CI/CD platform until 2022, when GHA takes over. They also reveal the increasing relevance of CI/CD. The Continuous Delivery Foundation, a project of the Linux Foundation, [132] provides a report on the state of CD. They find that 84% of developers participated in DevOps-related activities as of Q1 2023. They also find that, on average, developers

use 4.5 DevOps-related platforms used by developers concurrently (this number remained stable from 2019 to 2023). Stack Overflow's annual developer surveys [130] show an increase in CI/CD and DevOps usage year-over-year.

These studies' results generally align with ours, from the prominence of GHA, the increase in CI/CD usage, and the co-usage of several platforms. However, Jenkins had a much smaller representation in our data than in JetBrains's surveys. We address this is section 3.4.

3.4 Threats to Validity

There are multiple threats to the validity of the study, which are addressed in the following paragraphs.

The study focuses on open-source software and, in particular, on projects hosted on GitHub. Thus, the sampling does not include other kinds of software (e.g., proprietary). Conclusions cannot be generalized to these other kinds of software projects. Nevertheless, many companies also have their software on GitHub, and one may expect workers from these companies to use similar platforms in other projects. Moreover, others have reached similar conclusions by interviewing developers [65].

Since only GitHub was used, it cannot be said that these results apply to projects in other code repository services. However, there is no reason to consider projects hosted on GitHub to be significantly different from other projects in other repository services. Still regarding the use of GitHub as the source of the software projects analyzed, GHA is predominant. One of the main reasons for this may be related to the use of GitHub as the source of projects. However, GitLab CI/CD was also detected, the CI/CD platform used by another repository service (GitLab).

The sample repositories were collected by getting the 1,000 results sent by the GitHub API, doing it every week in the time frame. This resulted in more than 600,000 repositories, from which more than 200,000 have CI/CD. Although more repositories could have been collected, this would increase the time to retrieve them in a way that would make the work unfeasible. Moreover, the query did not impose any restriction on the results, except for the 10 stars used to have some "quality" metric for the projects. Thus, the repositories retrieved should not be biased in any other way.

Only platforms that could be identified through files in the repository were considered. Indeed, from the 61 platforms identified by the Cloud Native Computing Foundation, 14 platforms could not be identified (plus 1 deprecated). Nevertheless, 64% of all platforms could be identified.

Some platforms are detected through file contents, and there is no guarantee that a random file will not have a specific string inside that matches. However, the defined content would only make sense in the platform context. This probably did not happen. In any case, if it happened, it was for a minimal number of files that should not change the overall conclusions of the work.

There is an assumption that the presence of CI/CD artifacts (e.g., configuration files) means the underlying project is using such a platform. However, this may not be the case as some artifacts may be left forgotten from older usages.

3.5 Implications for Transpiler Design

Besides substantiating the need for a tool like the one we propose, this study helped us make several decisions regarding the CI/CD pipeline transpiler.

We determined to base our PIMM on some of the most popular CI/CD platforms. We chose GHA, CircleCI, and Jenkins. From the study, GHA is by far the most popular CI/CD platform, and CircleCI has had a significant user base for several years. Jenkins is also a very popular CI/CD platform [135] (even if we under-detect it, as addressed in section 3.4).

The existence of so many repositories co-using CI/CD platforms also led us to prototype an approach to merge CI/CD pipelines from several platforms using our PIMM.

Chapter 4

Overview of the Approach to CI/CD Pipeline Migration

In this chapter, we present our methodology for CI/CD pipeline migration. As described in section 4.1, this methodology is a reengineering process that makes use of two modeling levels and a DSL. In section 4.2, we present an example of pipeline migration using our methodology.

4.1 Automatic Migration Tool

The best way to support developers in migrating CI/CD would be to automate translating their existing pipeline into the new syntax as much as possible. This would speed up the migration, helping keep up productivity and reducing lock-in to any platform.

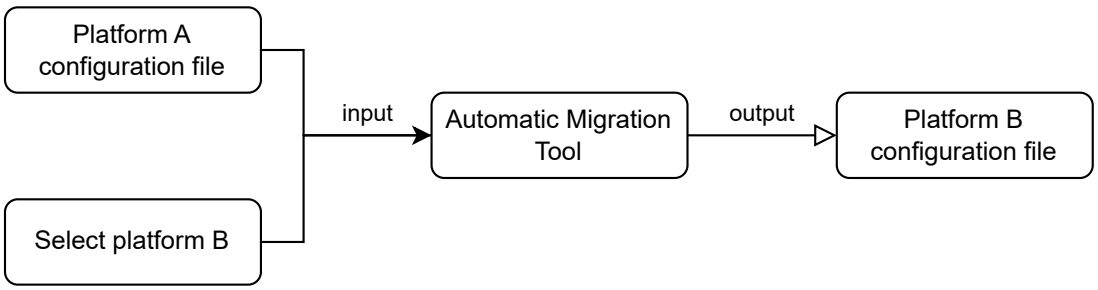


Figure 4.1: Automatic CI/CD migration tool.

Our goal is to create a transpiler for CI/CD pipelines that would function as described in figure 4.1. A user would input a CI/CD pipeline written in the DSL of a given platform A and choose the platform they want to migrate that pipeline into (platform B). The program would have two modes. Normal mode would migrate all the elements of the pipeline it could; this mode should be seen as a helper to the migration process. Strict mode would only migrate a pipeline if it was possible to generate a semantically equivalent pipeline in the new platform. Otherwise, the program exits with an error.

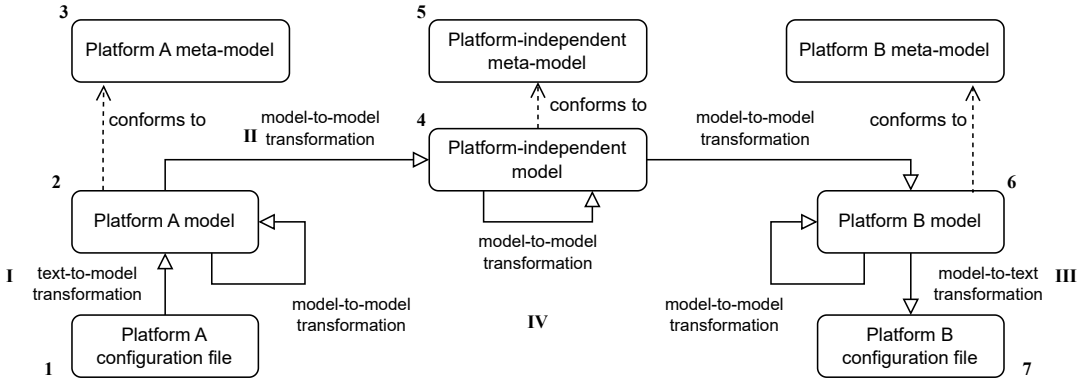


Figure 4.2: CI/CD pipeline reengineering process.

To this end, this work intends to leverage model-driven engineering by creating a PIMM that defines a modeling language for CI/CD pipelines. The PIMM would serve as the basis for a complete reengineering process in automatic CI/CD migration, shown in figure 4.2 as a horseshoe model [83]. Such a meta-model would need to allow modeling pipelines with low-level detail to make translation between platforms possible.

To improve migration functionality, users should be able to interact with the models during the migration process. We propose a Transformations DSL (TDSL) that lets users make changes that the tool cannot make on its own, much like the GAI DSL.

The tool's logic will be implemented using model transformations following the method base for migration methods specified by Grieger et al. [66]. Text-to-model transformations (**I**) will convert an input pipeline configuration file (**1**) to a PSM (**2**). That PSM will be transformed into a PIM (**3**) that conforms to the PIMM (**4**) through M2M transformations (**II**). We will then transform the PIM to a PSM for a different pipeline platform (**5**). The translated configuration file (**6**) will be generated from the new PSM through M2T transformations (**III**). The user can interact with this process with the TDSL to perform M2M transformations on the PIM and PSMs (**IV**). Strict mode will be implemented using a model validation on the input PSM (**2**).

Using two modeling levels, platform-specific and platform-independent, modularizes the reengineering process. Without this, T2M and M2T transformations would need to handle differences between platforms and the PIMM and change the pipeline representation from a text file to a model. This double responsibility would result in overly complex transformations that could not be iterated on easily.

The TDSL is also implemented through a reengineering process. TDSL scripts are transformed into a TDSL model. That TDSL model is transformed into ATL transformation rules, which are then compiled.

As one of the main motivators for migration is to consolidate CI/CD platforms [121], we also propose a way to merge multiple input pipelines into one output pipeline. This, together with the previously mentioned functionality, should allow developers to lessen the overhead of using multiple CI/CD platforms.

```yaml
version: 2.1

orbs:
  python: circleci/python@2.1.1

workflows:
  sample:
    jobs: [build-and-test]

jobs:
  build-and-test:
    docker:
      - image: cimg/python:3.10.5
    steps:
      - checkout
      - python/install-packages: {pkg-manager: "pip"}
      - run: {name: "Run tests", command: "pytest"}
```

Listing 4.1: CircleCI input script.

Chapter 5 details the creation of the PSMMs and the PIMM. Chapter 6 details the implementation of the reengineering process. Chapter 7 details the implementation of the TDSL. In chapter 8, we detail how we integrate all of these transformations into one, cohesive CLI.

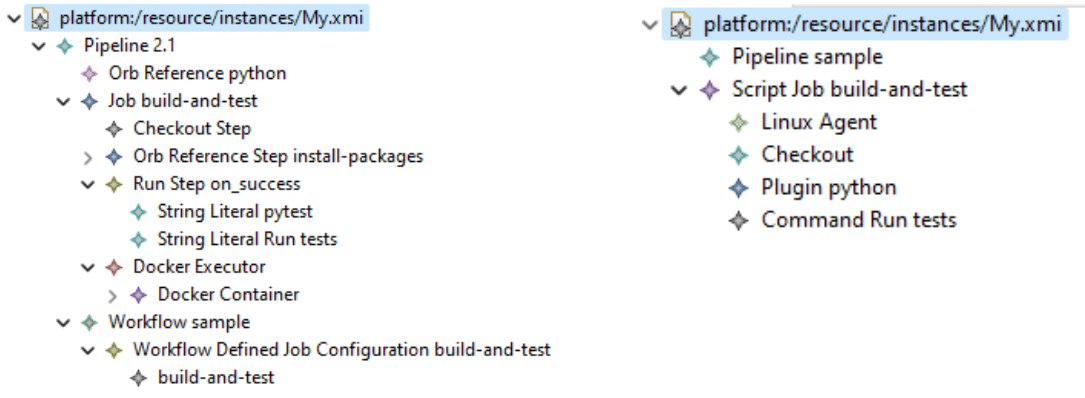
4.2 Execution Example

This section details the reengineering pipeline using the concrete example of migrating listing 4.1's CircleCI pipeline (**1**) to GHA.

A T2M transformation (**I**) generates the CircleCI model (**2**) shown in figure 4.3a. Model-to-model transformations (**II**) will then migrate that pipeline to a PIM (**4**), shown in figure 4.3b (this figure shows significantly less detail than figure 4.3a, this is only due to the Eclipse IDE showing the remaining PIM elements in different views).

This migration requires the use of the TDSL file from listing 4.2. This file applies three transformations (**IV**) on the PIM. Firstly, it adds a manual trigger to the pipeline as GHA requires at least one trigger for a valid pipeline. Afterward, it sets the Docker container's options so the pipeline uses the root user so GHA can use it. Lastly, it the replaces call to a CircleCI orb with a command to install Python packages. Figure 4.4a shows the PIM after the TDSL transformations have been applied. The PIM is then transformed to a GHA model (**6**), as shown in figure 4.4b.

The last step is to run an M2T transformation (**III**) to output the GHA pipeline script (**7**) from listing 4.3.

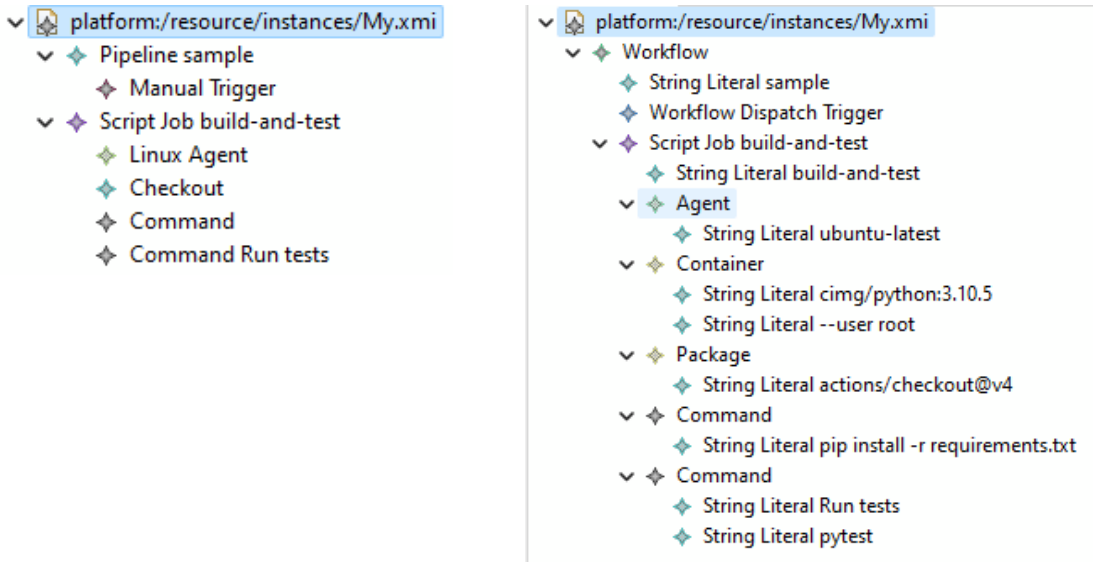


(a) CircleCI model of the input script. (b) PIM of input script.

Figure 4.3: CircleCI model and PIM representations of input pipeline script.

```
1 while {
2     add trigger when "input.triggers->isEmpty()" manual
3     set container options when "true" to '--user root'
4     replace step 2 on 'build-and-test' with command {
5         script 'pip install -r requirements.txt'
6     }
7 }
```

Listing 4.2: TDSL script for migration.



(a) PIM after TDSL transformations. (b) Output GHA model.

Figure 4.4: PIM and GHA model representations of output pipeline.

```yaml
name: "sample"

on:
  workflow_dispatch:

jobs:
  build-and-test:
    name: "build-and-test"
    runs-on:
    - "ubuntu-latest"
    container:
      image: "cimg/python:3.10.5"
      options: "--user root"
    steps:
    - uses: "actions/checkout@v4"
    - run: "pip install -r requirements.txt"
    -
      run: "pytest"
      name: "Run tests"
```

Listing 4.3: GHA output script.

Chapter 5

From CI/CD Concepts to Meta-Models

This chapter details the meta-models used by the transpiler and their creation process. Section 5.1 details the creation of the PSMMs. Section 5.2 addresses RQ1 and details how we design the PIMM to accurately represent CI/CD pipelines in a platform-independent manner.

5.1 Creating the Platform-Specific Meta-Models

We started development by researching three CI/CD platforms that represent the current CI/CD usage [57, 65, 135]. These are GHA [61], CircleCI [28], and Jenkins [80]. All of these platforms' providers make a configuration reference available for them. We used these references as the basis for the PSMMs.

The references allowed us to determine the features of each platform, as well as its valid configurations. With this, we can create a basic PSMM. After having a basic PSMM, we searched for commonalities between its classes to establish inheritance relationships. This lets us simplify the models by reducing redundancy.

Appendix A contains truncated versions of the finalized PSMMs used in the reengineering process. The full version can be found at [50].

5.2 Creating the Platform-Independent Meta-Model

The first research question prompts us to discover what concepts are core and common to different CI/CD platforms. We found that the studied platforms all share many traits and focused on designing a PIMM (figure 5.1) that could represent the diverse platforms' pipelines.

There are several points to consider when specifying concepts and properties for the PIMM [18]. For each PIMM class, we must determine intrinsic and extrinsic properties, i.e., properties that refer to basic objects and properties that refer to objects of other meta-model classes, respectively. This includes deciding which properties should truly belong to a class and which should be aggregated into another class that will become an extrinsic property. If this is done incorrectly, it could lead to a meta-model that is hard to work on and evolve.

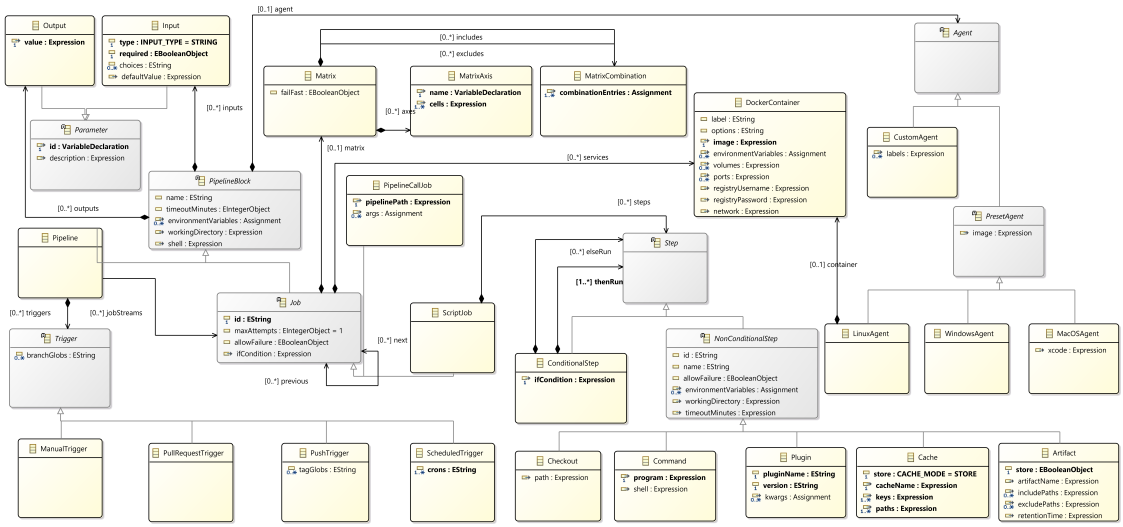


Figure 5.1: Truncated PIMM (missing **Expressions**, **VariableDeclaration**, and enumerated classes).

The arity of each property is also key. If it is wrong, the meta-model will not be an accurate abstraction of the CI/CD platform's pipeline and make it so PSMs are malformed.

Lastly, we must also decide whether an extrinsic property will be contained within its referring object or not. This is important as it will determine whether the referred object can outlive the referring object in case of deletion.

Sections 5.2.1 to 5.2.8 detail the core CI/CD platform concepts, how they are represented in the PIMM, and how the PIMM concepts map to the various PSMMs' concepts. Section 5.2.9 details core differences between the platforms.

5.2.1 Pipeline

All studied CI/CD platforms are based on the concept of a **Pipeline**. This **Pipeline** aggregates the various actions users want to execute when a certain event occurs. Platforms also let users specify certain configurations with a pipeline-wide scope.

Table 5.1: **Pipeline** PIMM classes and properties.

Concepts	**Intrinsic Properties**			**Extrinsic Properties**		
	name	**type**	**arity**	**name**	**type**	**arity**
PipelineBlock	name	EString	0..1	agent	Agent	0..1
	timeoutMinutes	EIntegerObject	0..1	inputs	Input	*
				outputs	Output	*
				environmentVariables	Assignment	*
				workingDirectory	Expression	0..1
				shell	Expression	0..1
Pipeline *extends* **PipelineBlock**				triggers	Trigger	*
				jobStreams	Job	*

The **Pipeline** is the main concept of the PIMM as it represents each CI/CD script and, as such, references every other model element either directly or indirectly.

As seen in table 5.1, **Pipelines** can have *triggers* that specify the events that start their execution (detailed in section 5.2.2), and *jobStreams*, that specify the **Jobs** that make up the **Pipeline** (detailed in section 5.2.3).

Moreover, **Pipelines** inherit multiple properties from the **PipelineBlock** abstract class, which groups common functionality of **Pipelines** and **Jobs**:

- a *name*, which serves only for display purposes
- *timeoutMinutes*, which define a maximum execution time for the **PipelineBlock**
- *inputs* and *outputs*, detailed in section 5.2.7, that are used so a **PipelineBlock** can be called by another **PipelineBlock**
- default *environmentVariables*, *workingDirectory*, and *shell*, that can be overridden in child elements
- an *agent*, to specify the default VM where the **PipelineBlock** is executed

Table 5.2 maps the **Pipeline** concept to the various PSMs' concepts.

Table 5.2: **Pipeline** class mappings.

Meta-model	Class
GHA	Workflow
Jenkins	Pipeline
CircleCI	Workflow

5.2.2 Triggers

Triggers define events that start the execution of a **Pipeline**.

GHA, Jenkins, and CircleCI all support this functionality, but their implementations differ. In GHA, all trigger configuration is done in the pipeline script. In Jenkins and CircleCI, the vast majority of configuration is part of the platform's settings and not included in the pipeline script. This is due to the tighter platform integration of GHA with GitHub. Still, all platforms end up supporting the same functionality.

In spite of not being able to generate pipeline scripts with configurations for other kinds of triggers in CircleCI and Jenkins, these are still included in the PIMM due to their relevance in the domain. In theory, integration with the CircleCI and Jenkins platforms could let us configure **Triggers** from their PSMs even if we cannot generate a script with them.

As seen in table 5.3, there are various kinds of **Triggers** supported by the PIMM.

- **PushTriggers** start pipeline execution when a commit is pushed. The **branchGlobs** property lets users restrict the git tags where this happens through the use of glob patterns.

Table 5.3: **Trigger** PIMM classes and properties.

Concepts	**Intrinsic Properties**			**Extrinsic Properties**		
	name	**type**	**arity**	**name**	**type**	**arity**
Trigger	branchGlobs	EString	*			
PushTrigger *extends* **Trigger**	tagGlobs	EString	*			
PullRequestTrigger *extends* **Trigger**						
ManualTrigger *extends* **Trigger**						
ScheduledTrigger *extends* **Trigger**	crons	EString	1..*			

- **PullRequestTriggers** start pipeline execution when a pull request is created.
- **ManualTriggers** let the user execute the pipeline without a repository event.
- **ScheduledTriggers** execute the pipeline at set intervals, defined by the *crons* property.

All **Triggers** have a *branchGlobs* property that lets users restrict the git branches where the **Trigger** applies.

There are far more events that can trigger pipelines. However, as the transpiler is not currently integrated with the CircleCI and Jenkins platforms, **Trigger** migration is limited. Because of this, these other events are not supported by the PIMM. Table 5.4 maps the PIMM's **Triggers** to the equivalent PSM classes.

Table 5.4: **Trigger** class mappings.

Meta-model	Classes
GHA	StandardEventTrigger, WorkflowRunTrigger, PullRequestTrigger PullRequestTargetTrigger, PushTrigger, ScheduleTrigger WorkflowCallTrigger, WorkflowDispatchTrigger
Jenkins	ScheduledTrigger, PollingTrigger, UpstreamTrigger
CircleCI	ScheduleTrigger

5.2.3 Jobs

A **Job** is a set of instructions that are run sequentially as a single execution block of the pipeline. Table 5.5 shows the PIMM's **Job** classes.

Table 5.5: **Job** PIMM classes and properties.

Concepts	Intrinsic Properties			Extrinsic Properties		
	name	**type**	**arity**	**name**	**type**	**arity**
Job *extends* **PipelineBlock**	id	EString	1	ifCondition	Expression	0..1
	maxAttempts	EIntegerObject	0..1	services	DockerContainer	*
	allowFailure	EBooleanObject	0..1	matrix	Matrix	0..1
				previous	Job	*
				next	Job	*
ScriptJob *extends* **Job**				steps	Step	*
PipelineCallJob *extends* **Job**				pipelinePath	Expression	0..1
				args	Assignment	*

All **Jobs** must have an *id* to uniquely identify them in the **Pipeline**. The *allowFailure* and *maxAttempts* specify whether a **Job** failing terminates **Pipeline** execution and the maximum number of times a **Job** should be run if not successful.

Users also have multiple options to configure the execution flow of jobs.

- The *ifCondition* property specifies a condition that must be met for the **Job** to execute.
- **Jobs** can be run either in parallel with one another or sequentially. By default, GHA and CircleCI run **Jobs** in parallel, while Jenkins runs them sequentially. However, users can configure this behavior. The PIMM runs **Jobs** in parallel by default, and dependencies are established using the *previous* and *next* properties.
- **Jobs** can also be configured with a *matrix*. **Matrices**, detailed in section 5.2.5, specify multiple arrays of values the user wants to run the **Job** with. The **Job** will then be run for all allowed combinations of values.

The PIMM also lets users specify ancillary Docker containers through the *services* property. These services run in the background throughout **Job** execution. They can be used to set up databases, among other things. Besides these properties, **Jobs** also inherit the **PipelineBlock** properties detailed in section 5.2.1.

There are two kinds of **Jobs**.

- **ScriptJobs** are composed of **Steps**, which are the atomic instructions of the **Pipeline** (detailed in section 5.2.6). Table 5.7 maps the **ScriptJob** to platform-specific concepts.
- **PipelineCallJobs** are a call to a separate **Pipeline**. This is done through the **Pipeline**'s file path (*pipelinePath*) and arguments (*args*). In GHA, calling a separate pipeline script is done through a specific kind of job, while in CircleCI and Jenkins, it is done through a specific plugin. Table 5.7 maps the **PipelineCallJob** to platform-specific concepts.

Table 5.6: **ScriptJob** class mappings.

Meta-model	Classes
GHA	ScriptJob
Jenkins	StepStage, MatrixStage, ParallelNestedStage
CircleCI	Job, WorkflowJobConfiguration

Table 5.7: **PipelineCallJob** class mappings.

Meta-model	Classes
GHA	WorkflowCallJob
Jenkins	StepStage, MatrixStage, ParallelNestedStage, Step
CircleCI	Job, WorkflowOrbJobConfiguration

5.2.4 Agents and Services

An **Agent** specifies where a **Pipeline** or **Job** will be run. There are various types of **Agent**.

Table 5.8: **Agent** and **DockerContainer** PIMM classes and properties.

Concepts	**Intrinsic Properties**			**Extrinsic Properties**		
	name	**type**	**arity**	**name**	**type**	**arity**
Agent						
CustomAgent *extends* **Agent**				labels	Expression	*
PresetAgent *extends* **Agent**				image	Expression	0..1
WindowsAgent *extends* **PresetAgent**						
LinuxAgent *extends* **PresetAgent**				container	DockerContainer	0..1
MacOSAgent *extends* **PresetAgent**				xcode	Expression	0..1
DockerContainer	label	EString	0..1	image	Expression	1
	options	EString	0..1	environmentVariables	Assignment	*
				volumes	Expression	*
				ports	Expression	*
				registryUsername	Expression	0..1
				registryPassword	Expression	0..1
				network	Expression	0..1

PresetAgents define preconfigured VMs made available by platform providers. These can be either **WindowsAgents**, **LinuxAgents**, or **MacOSAgents**. The *image* property lets users specify the particular VM they want to use.

A **LinuxAgent** can also have a **DockerContainer** so the user can better configure the execution environment.

CustomAgents defined **Agents** that need further user configuration. These are referred to with *labels*.

A **DockerContainer** has the information necessary to initialize a Docker container. The *image* property specifies the Docker image. The *environmentVariables*, *volumes*, *ports*, and *network* properties all map to basic Docker functionality. The *options* property lets users specify other Docker options. The *registryUsername* and *registryPassword* properties are used to access the Docker registry. The optional *label* property is used to refer to the **DockerContainer**.

Table 5.8 details PIMM's **Agent** classes. Table 5.9 maps the **Agent** PIMM class to PSMM concepts.

Table 5.9: **Agent** class mappings.

Meta-model	Classes
GHA	Agent
Jenkins	NoneAgent, AnyAgent, LabelAgent, NodeAgent, DockerAgent
CircleCI	DockerExecutor, MachineExecutor, MacOSExecutor, ExecutorReferenceExecutor, OrbReferenceExecutor

5.2.5 Matrices

A **Matrix** is used to define combinations of values to run the **Job** with. For example, in a **Matrix**, a user can define arrays values for an `OS` and a `program` they would like to run. The **Job** would then be run for every combination of `OS` and `program`.

As seen in table 5.10, a **Matrix** has an arbitrary number of *axes*. A **MatrixAxis** defines an array of values in the **Matrix**. It has a *name* property, so it can be referred to, and one or more *cells*, which specify array values.

The *includes* property is used to specify particular combinations of values that cannot be made using the **axes**. For example, a user can specify one single `version` value in addition to the aforementioned `OS` and `program` axes. The **Job** would then be run for every combination of `OS`, `program`, and the specified `version` value. The *excludes* property is used to disallow particular combinations of *axes* values.

Both the *includes* and *excludes* properties refer to a **MatrixCombination**. A **MatrixCombination** lets users multiple **Assignments** (detailed in section 5.2.8) for **Matrix** values.

The *failFast* property is used to specify whether the **Job** should fail immediately if one of the **Matrix** combinations fails or if it should keep running.

5.2.6 Steps

Steps are atomic instructions that run as part of a **Job**. As seen in table 5.11, there are various kinds of **Steps**.

Table 5.10: **Matrix** PIMM classes and properties.

Concepts	Intrinsic Properties			Extrinsic Properties		
	name	**type**	**arity**	**name**	**type**	**arity**
Matrix	failFast	EBooleanObject	0..1	axes	MatrixAxis	*
				includes	MatrixCombination	*
				excludes	MatrixCombination	*
MatrixAxis				name	VariableDeclaration	1
				cells	Expression	1..*
MatrixCombination				combinationEntries	Assignment	1..*

ConditionalSteps are used for flow control. They have one or more *thenSteps* that are executed when the *ifCondition* is true and can have *elseSteps* that are executed otherwise.

NonConditionalSteps can have an *id*, a display *name*, a *workingDirectory*, and *environmentVariables*. With *allowFailure*, they can be configured to continue **Pipeline** execution if they fail, and with *timeoutMinutes*, to fail after an allotted amount of time.

Commands run a specified *program* in the **Job**'s **Agent**. They can also specify a *shell* where the *program* is run.

Plugins run platform-specific packages that are made available in marketplaces. These are referred to by their *pluginName* and *version*. They may also receive arguments (*kwargs*).

Cache steps can either load or store data from the **Agent** to speed up subsequent **Pipeline** executions. The *cacheName* is used to refer to the cache. A **Cache** step also has to have one or more cache *keys* and *paths*.

Artifacts can either store output data specific to a certain **Pipeline** execution or download another artifact. They can have an *artifactName*, *include* and *excludePaths*, and a *retentionTime* before the artifact is deleted.

Checkout serves to load the git repository into the **Agent**. The *path* property specifies where the repository will be loaded.

Platforms differ in their implementations of steps. In Jenkins, all steps are a call to platform-specific plugins (even running a program in a shell). GHA only has two steps: executing programs in the agent or calling to platform-specific packages. The PIM is closest to CircleCI's model with regard to **Steps**, as it has native support for caching, uploading artifacts, and checking out repositories.

This is done as these kinds of steps are crucial to pipeline scripts, and all platforms support them, even if not natively. Having the PIMM abstract these steps allows us to automatically migrate this functionality from platform to platform. This abstraction does lead to some information loss, section 9.2 assesses the impact of this.

Tables 5.12 to 5.15 and 5.17 and section 5.2.6 have the mappings to the equivalent PSMM step classes.

Table 5.11: **Step** PIMM classes and properties.

Concepts	Intrinsic Properties			Extrinsic Properties		
	name	**type**	**arity**	**name**	**type**	**arity**
Step						
ConditionalStep *extends* **Step**				ifCondition	Expression	1
				thenRun	Step	1..*
				elseRun	Step	*
NonConditionalStep *extends* **Step**	id	EString	0..1	environmentVariables	Assignment	*
	name	EString	0..1	workingDirectory	Expression	0..1
	allowFailure	EBooleanObject	0..1	timeoutMinutes	Expression	0..1
Command *extends* **NonConditionalStep**				program	Expression	1
				shell	Expression	0..1
Plugin *extends* **NonConditionalStep**	pluginName	EString	1	kwargs	Assignment	*
	version	EString	1			
Cache *extends* **NonConditionalStep**	store	CACHE_MODE	1	cacheName	Expression	1
				keys	Expression	1..*
				paths	Expression	1..*
Artifact *extends* **NonConditionalStep**	store	EBooleanObject	1	artifactName	Expression	0..1
				includePaths	Expression	*
				excludePaths	Expression	*
				retentionTime	Expression	0..1
Checkout *extends* **NonConditionalStep**				path	Expression	0..1

5.2.7 Parameters

Pipelines and **Jobs** can have inputs and outputs. These are defined using **Parameters**, seen in table 5.18.

A **Parameter** has an *id*, so it can be referred to. It can also have a *description*.

Inputs have a *type*, a *required* property, and an optional *defaultValue*. They may also have *choices*, used to enumerate valid values in case the **Input** is of CHOICE *type*.

Outputs must have a *value*.

Tables 5.19 and 5.20 have the mappings to the equivalent PSMM classes.

5.2.8 Expressions and Variables

All platforms have some expression grammar. The PIMM addresses this with the **Expression** classes

Expressions include logical operators, literals, variable references, and formatted strings (that mix string literals and other expressions).

The only major difference between PIMM and PSMM happens with CircleCI. In CircleCI, logical operators can have an arbitrary number of operands. The PIMM uses more standard binary logical operators. This difference has no bearing on the functionality of either platform and can be handled by M2M transformations.

Table 5.12: **Command** class mappings.

Meta-model	Class
GHA	Command
Jenkins	Step
CircleCI	RunStep

Table 5.13: **ConditionalStep** class mappings.

Meta-model	Classes
GHA	IfStep
Jenkins	ConditionalStep
CircleCI	WhenStep, UnlessStep

Table 5.14: **Cache** class mappings.

Meta-model	Classes
GHA	Package
Jenkins	Step
CircleCI	SaveCacheStep, RestoreCacheStep

Table 5.15: **Artifact** class mappings.

Meta-model	Classes
GHA	Package
Jenkins	Step
CircleCI	StoreArtifactsStep, OrbReferenceStep

Table 5.16: **Checkout** class mappings.

Meta-model	Class
GHA	Package
Jenkins	Step
CircleCI	CheckoutStep

Table 5.17: **Plugin** class mappings.

Meta-model	Classes
GHA	Package
Jenkins	Step
CircleCI	SetupRemoteDockerStep, StoreTestResultsStep, PersitToWorkspaceStep, AttachWorkspaceStep, AddSSHKeysStep, OrbReferenceStep

Currently, PIMM **VariableDeclarations** happen in **Assignments**, **Parameters** and **MatrixAxes**. Platforms support other kinds of variables, such as accessing the SHA of the commit that triggered the pipeline or accessing secrets defined in the platform (used so sensitive information is not made public). These concepts are common to the various platforms but are not currently part of the PIMM. This means the PIMM cannot refer to these variables in a platform-independent manner and, consequently, cannot automatically migrate them.

5.2.9 Core Differences Between Platforms

There are clear differences between the platforms. Some of these differences can be handled by the PIMM, while others are not currently representable

CircleCI and Jenkins can be configured to be very modular through the use of functions and the reuse of **Jobs**. There are ways to imitate this in GHA, but they are more cumbersome and involve

Table 5.18: **Parameter** PIMM classes and properties.

Concepts	**Intrinsic Properties**			**Extrinsic Properties**		
	name	**type**	**arity**	**name**	**type**	**arity**
Parameter				id	VariableDeclaration	1
				description	Expression	0..1
Input *extends* **Parameter**	type	INPUT_TYPE	1	defaultValue	Expression	0..1
	required	EBooleanObject	1			
	choices	EString	*			
Output *extends* **Parameter**				value	Expression	1

Table 5.19: **Input** class mappings.

Meta-model	Class
GHA	Input
Jenkins	
CircleCI	Parameter

Table 5.20: **Output** class mappings.

Meta-model	Class
GHA	Output
Jenkins	
CircleCI	

the use of multiple files. We did not consider this functionality to be core to CI/CD pipelines. As such, the meta-model does not currently support functions or the reuse of **Jobs**.

GitHub Actions has some unique features due to its tight integration with GitHub, like setting permissions for the pipeline to interact with the repository. Moreover, all platforms have minor features that are not supported by the others. These are not supported by the PIMM.

Chapter 6

Implementing the Reengineering Process

This chapter details the implementation of the reengineering process outlined in figure 4.2.

The chapter's structure follows the reengineering process itself, starting with T2M transformations, detailed in section 6.1. Section 6.2 details the model validations used to implement strict mode. Section 6.3 details M2M transformations. Section 6.4 concludes the chapter with M2T transformations.

6.1 Text-to-Model Transformations

Text-to-model transformations are the first step of the reengineering pipeline. We use them to create PSMs of the CI/CD pipelines we parse. They are also the only main component of the tool's logic that is not implemented using entirely model-driven technologies.

The original approach we considered was to perform T2M transformations by generating a parser with Xtext to perform embedded translation. However, this could not be done as the DSLs being parsed have some complexity with regard to variable declarations and references. This requires the use of symbol tables to populate the PSM in such a manner that all the references are accurate.

For these reasons, we use a two-step output production strategy. This means that first, we parse the input text into an abstract syntax tree (AST). Afterward, we walk the AST and output the model as we visit its nodes. This allows the use of symbol tables, which makes variable declarations and references possible.

This can theoretically be done with MDE technologies. We can define a grammar/meta-model for a DSL and use Xtext to generate a parser for that DSL. From the DSL model outputted by the parser, we can use ATL to perform M2M transformations from that DSL model to a PSM (as outlined in section 6.3).

However, we elected to go a different way. Both platforms for which we implemented T2M transformations are based on YAML. As YAML has a very complex syntax, we chose to use a

```
import 'platform:/resource/d.fe.up.pt.cicd.gha.metamodel/model/GHA.ecore'

package GHA

context Workflow

inv MatrixSettings :
        if Matrix.allInstances()->select(
                matrix |
                matrix.axes->isEmpty()or
                matrix.failFast <> null or
                matrix.maxParallel <> null
        )->notEmpty() then null else true endif

endpackage
```

Listing 6.1: GHA-to-CircleCI Model Constraint Example.

pre-existing Java YAML parser package instead of trying to develop a grammar ourselves. For certain platforms, like GHA, this Java package is not enough to parse pipeline scripts. This is because GHA's DSL is an extension of YAML with expressions in certain, well-defined places. When the parser arrives at one of these expressions, it uses a GHA expression grammar that we developed to parse them. These parsers were based on the platforms' configuration references.

After building the AST, we visit the nodes using Java instead of ATL, as Java is better suited to dealing with the symbol tables.

6.2 Model Validations

We use model validations to implement the strict mode of the transpiler. When running in strict mode, we validate the input model to check if it can be guaranteed that the output pipeline will be semantically equivalent to the input one. These validations check whether certain platform features are being used. These validations must be defined for every permutation of input and output platforms, as certain pipeline elements may be migratable to one platform and not another.

Validations are implemented in Complete OCL, which separates constraint definition from the meta-model definition. This is useful as it lets the transpiler use different validation files for the same model in different contexts. Listing 6.1 shows an example of a constraint defined in CompleteOCL.

Currently, strict mode is only supported in GHA-to-CircleCI migrations. The implemented constraints are a product of differences between technologies, PIMM limitations, and limitations in the transformations. The current constraints for GHA-to-CircleCI strict mode migration forbid:

- environment variables defined at the **Pipeline** level

- default working directory or shell defined at the **Pipeline** level
- GHA staging environments
- definition of GHA permissions
- use of undeclared variables like accessing commit or platform information
- use of GHA-specific built-in functions
- use of GHA concurrency groups
- use of GHA secrets
- use of **Job** and **Step** timeouts
- use of certain **Matrix** and **DockerContainer** configurations
- use of identifiers to refer to **Steps**
- use of **Jobs** that call other **Pipelines**

6.3 Model-to-Model Transformations

Model-to-model transformations presented the main challenge of our approach, as they are the logic that allows pipelines to be translated from one platform to another (section 6.3.1. They also let us merge multiple pipelines into a single one (section 6.3.2),

6.3.1 Migrating Pipeline Platforms

We use M2M transformations to transform a pipeline from a PSM representation to our PIM and vice-versa. This way, the transpiler uses the PIM as an intermediate representation, and we avoid having to define transformations for every permutation of the input and output platforms.

The PIMM was designed to ease transformations to and from the PSMs. Where it was possible, we wanted one-to-one mappings between PIMM and PSMM, i.e., when a concept or property in one meta-model has a direct correspondence to one in the other meta-model. Still, this is not always possible due to significant platform differences. For example, Jenkins's way of executing **Jobs** in parallel by nesting them means the PIM to Jenkins transformation has to group the job dependency graph by levels and then nest **Jobs** from the same group.

While most of these differences between concepts can be handled in a single-step PSM-to-PIM/PIM-to-PSM transformation, some are more complex. To deal with this, we split the PSM-to-PIM/PIM-to-PSM transformation into multiple simpler transformations by defining helpers. All helpers are PIM-to-PIM transformations, allowing them to be reused for various platforms if appropriate.

Figure 6.1 shows an example of this. In PIM-to-GHA, the transpiler uses two helpers. GHA does not allow defining **Job** inputs while the PIMM does. This means **Job** inputs must be extracted

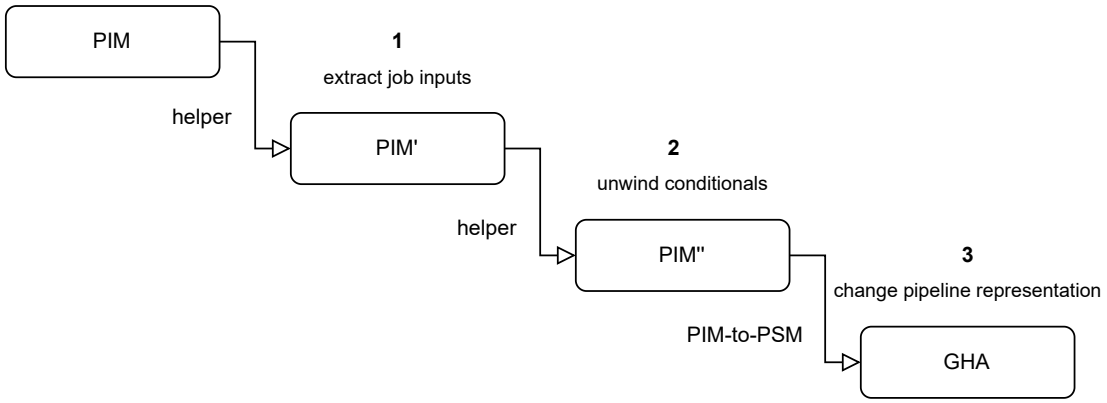


Figure 6.1: PIM-to-GHA transformations.

to the **Pipeline** so they are not lost (**1**). In the PIMM, **ConditionalSteps** can have an arbitrary number of child-steps to be executed when their condition is true; they also have an "else" block with multiple steps. GHA only allows one child-step per condition and has no "else" functionality. The helper unwinds **ConditionalSteps** on the PIM, creating multiple **ConditionalSteps**, each with one child-step (**2**) ("else" blocks are handled by negating the condition). Only after these transformations does the transpiler run a transformation to convert the PIM to a GHA model (**3**). Besides these, there are three more transformation helpers.

When executing PIM-to-Jenkins transformations, we must extract environment variables from **Steps** to their containing **Job** as Jenkins does not support defining environment variables at the **Step** level.

CircleCI is unique because it uses refinements in both CircleCI-to-PIM and PIM-to-CircleCI transformations. In CircleCI, a **Matrix** is defined outside of a **Job**, and the **MatrixAxes** correspond to **Inputs** on the **Job**. In the PIMM, the **Matrix** is an extrinsic property of the **Job**. This means there are extra **Inputs** defined on the **Job** after running the exogenous transformation. These must be removed and references to their variables altered to references to the **Matrix**. When transforming the PIM to CircleCI, the opposite process has to be executed.

The rules for these transformations are implemented in ATL. We chose ATL for its tight integration with EMF. This allows the ATL IDE to provide several development tools like a debugger, type checking, and contextual completions based on the meta-models used, easing the development process.

6.3.2 Merging Multiple Pipelines

Model mergers are used to combine various pipeline scripts into one, so we can help developers consolidate CI/CD platforms.

Pipeline merging only happens after the input PSMs have been transformed into PIMs. This lets us merge pipelines regardless of their original platform. A merger transforms two PIMs into one PIM. Merging models is a two-step process.

The first step is a comparison. We define rules to indicate what elements of the two input models match. We search for **Triggers** of the same type, **Pipeline Parameters** and **Jobs** with the same *id*, and **Pipeline** *environmentVariables* with the same name. This comparison creates a trace indicating these elements.

We merge the **Pipelines** using this trace. **Pipeline** *inputs*, *outputs*, *triggers*, *environmentVariables*, and *jobs* are combined. When **Jobs** match, the **Job** belonging to the leftmost PIM in the list of scripts to merge is given priority instead of combining the two **Jobs** (as combining jobs and guaranteeing the end result makes sense is too complex). Elements not determined to match by the comparison are copied into the new PIM.

Unlike the other transformations, model mergers are implemented using Epsilon languages as they have better support for comparing and merging models.

Although the transformation is only written to merge two PIMs into one, the tool we devised can merge an arbitrary number of pipelines through a reduction process.

6.4 Model-to-Text Transformations

Model-to-text transformations are responsible for code generation and only happen from PSMs. They are the simplest part of the reengineering process and are implemented in Acceleo.

Although these are vertical transformations, where we have to go from a higher abstraction level (the PSM) to a lower one (the pipeline scripts), since the PSMMs were created from pipeline configuration references of their respective platforms, the only missing information is the concrete textual syntax of the platforms. This is what simplifies M2T transformations.

Still, there are issues to consider, as the code generated has to be semantically equivalent to the PSM.

Pipeline DSLs often let the user specify the same pipeline element with different syntax. For example, in GHA, the three following trigger definitions are interchangeable: `on: push`, `on: [push]`, and `on: {push: null}`. This brings to light the differences between the GHA DSL and YAML, as in YAML, the three definitions are semantically different (they attribute a string, a list, and a map to the `on` key, respectively).

When generating code, we always choose the most flexible syntax available to avoid unnecessary complexity in the transformation. In the previous example, it would be defining the trigger as `on: {push: null}`. This syntax works in cases where there are multiple triggers (`on: {push: null, pull_request: null}`) and also supports further specifications to the trigger (`on: {push: {branches: "main"}}`). Using it, we can handle the largest amount of GHA trigger definitions without excessive use of conditionals. If the trigger does not have any `branches` property, we do not generate it, and if there are multiple triggers, we add an element to the `on` map.

Practically, using the transpiler to execute a GHA-to-GHA migration will most likely result in code that is not semantically equivalent in YAML, even if it is in GHA.

Chapter 7

The Transformations DSL

The TDSL is designed so the user can interact with the models throughout the reengineering process. With the TDSL, the user can migrate **Pipeline** elements the transpiler cannot migrate automatically or make other alterations to the **Pipeline**.

The TDSL's functionality is implemented through M2M transformations. All of these transformations are endogenous. Most transformations are PIM-to-PIM, as the TDSL's objective was to interact with the scripts in a platform-independent manner. However, the user can also define PSM-to-PSM transformations. A user can specify an arbitrary number of transformations in a single TDSL file.

```
1 before translating {
2     on circleci select workflow frontend
3 }
4 while translating {
5     replace step 2 on 'frontend-test' with command {
6         script 'npm install'
7     }
8 }
```

Listing 7.1: TDSL example.

Just like other one-to-one M2M transformations, TDSL transformations are implemented in ATL. Figure 7.1 shows the process of transforming a TDSL file to ATL transformations as a reengineering process. A TDSL file (**1**) is transformed to a TDSL model (**2**) using M2T transformations (**I**); this is detailed in section 7.2 along with TDSL syntax. The TDSL model conforms to the TDSL meta-model (**3**); this meta-model is detailed in section 7.1 along with TDSL functionality. The TDSL model is transformed (**II**) into as many ATL files (**4**) as there are TDSL transformations (each file performs one transformation). These ATL files are then compiled (**III**) to ATL assembly. Section 7.3 details the transformations from a TDSL model to ATL assembly files.

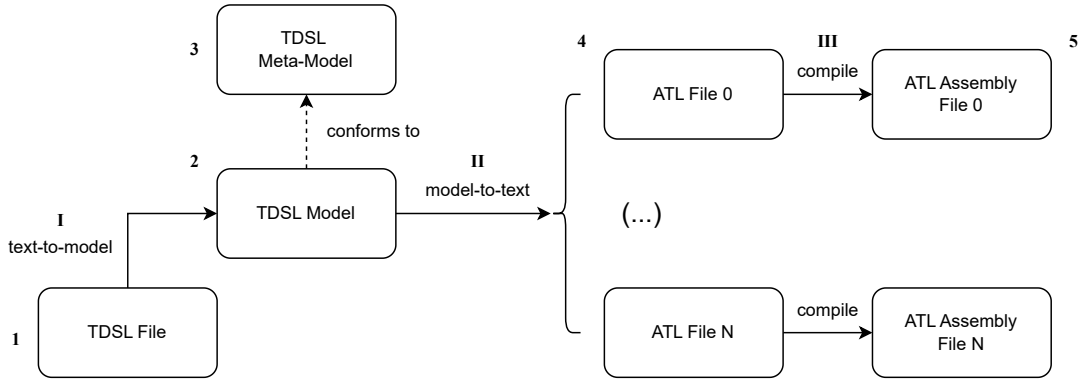


Figure 7.1: TDSL reengineering process.

7.1 The Transformations DSL Meta-Model

The TDSL meta-model (figure 7.2) was made to simplify common transformations that users must make when migrating CI/CD platforms that cannot be made automatically. These transformations can be made throughout the reengineering process, i.e., on the input PSM, on the PIM, and on the output PSM.

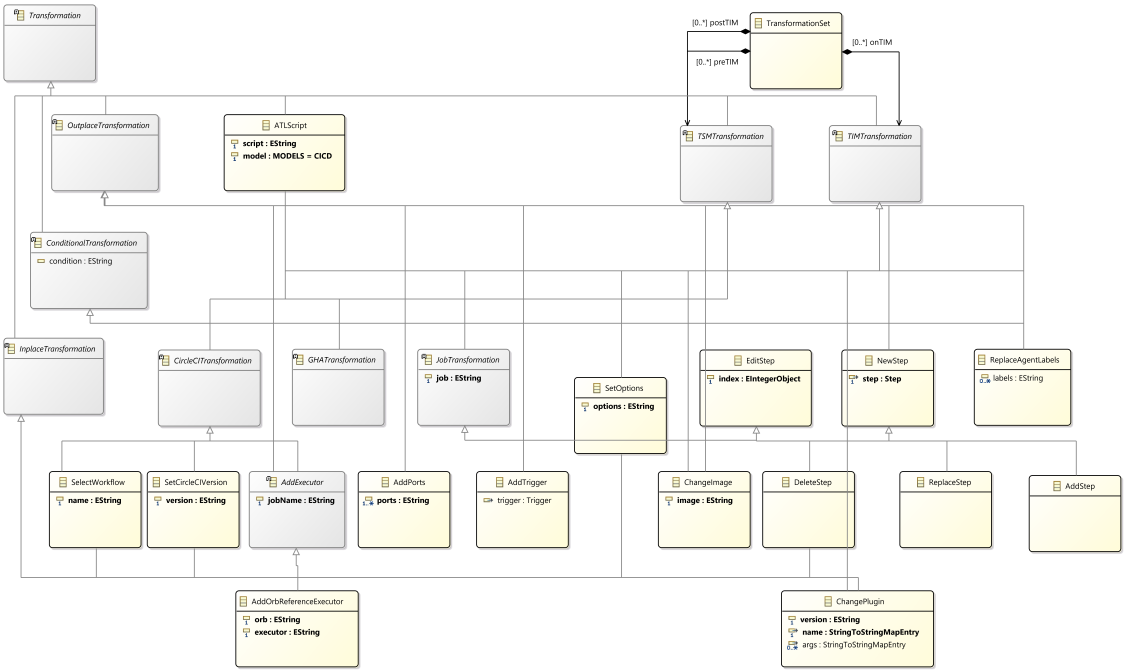


Figure 7.2: TDSL meta-model.

This meta-model stands apart because it references other meta-models, the PIMM and the PSMMs (in future iterations). Referencing these meta-models means we can avoid repetition in the TDSL meta-model when, for example, the TDSL is used to create a new PIMM **Step**, as we avoid copying the PIMM definition into the TDSL meta-model. This also increases the robustness of the TDSL meta-model as, when the other meta-models change, it changes too.

The main class of the TDSL meta-model is the **TransformationSet**. It lets users define an arbitrary number of **Transformations**. These are split between the *prePIM*, *onPIM*, and *postPIM* properties. The *onPIM* property contains **PIMTransformations** and the *prePIM* and *postPIM* properties contain **PSMTransformations**.

All **Transformations** are either **OutplaceTransformations** or **InplaceTransformations**. Certain **Transformations** are also **ConditionalTransformations**.

On the PIM, there is support for common migration pipeline changes such as changing **Plugins**' name, version, and arguments, replacing **CustomAgent** labels, adding **Triggers**, adding, replacing and deleting **Steps**, and interacting with Docker containers. Table 7.1 details **PIMTransformations**.

Table 7.1: TDSL **PIMTransformations**.

Concepts	Intrinsic Properties			Extrinsic Properties		
	name	**type**	**arity**	**name**	**type**	**arity**
ChangePlugin *extends* **InplaceTransformation**, **PIMTransformation**	version	EString	1	name	StringToStringEntry	1
				args	StringToStringEntry	*
ReplaceAgentLabels *extends* **OutplaceTransformation**, **PIMTransformation**, **ConditionalTransformation**	labels	EString	*			
AddTrigger *extends* **InplaceTransformation**, **PIMTransformation**, **ConditionalTransformation**				trigger	PIMM::Trigger	1
JobTransformation *extends* **PIMTransformation**	job	EString	1			
EditStep *extends* **JobTransformation**	index	EIntegerObject	1			
NewStep *extends* **EditStep**, **OutplaceTransformation**				step	PIMM::Step	1
AddStep *extends* **NewStep**						
ReplaceStep *extends* **NewStep**						
DeleteStep *extends* **EditStep**, **InplaceTransformation**						
AddPorts *extends* **OutplaceTransformation**, **PIMTransformation**, **ConditionalTransformation**	ports	EString	1..*			
SetOptions *extends* **InplaceTransformation**, **PIMTransformation**, **ConditionalTransformation**	options	EString	1			
ChangeImage *extends* **OutplaceTransformation**, **PIMTransformation**, **ConditionalTransformation**	image	EString	1			

Table 7.2: TDSL **PSMTransformations**.

Concepts	Intrinsic Properties			Extrinsic Properties		
	name	**type**	**arity**	**name**	**type**	**arity**
CircleCITransformation *extends* **PSMTransformation**						
AddExecutor *extends* **OutplaceTransformation**, **CircleCITransformation**	jobName	EString	1			
AddOrbReferenceExecutor *extends* **AddExecutor**	orb	EString	1			
	executor	EString	1			
SetCircleCIVersion *extends* **InplaceTransformation**, **CircleCITransformation**	version	EString	1			
SelectWorkflow *extends* **InplaceTransformation**, **CircleCITransformation**	name	EString	1			

Currently, the only PSM transformations supported explicitly by the TDSL are on the CircleCI meta-model. This is because the TDSL focuses on interacting with the reengineering process at the PIM level, and PSM interaction is only meant to handle PIMM limitations. It can add orb (CircleCI plugins) executors to a job, set the CircleCI version of the script, and select the workflow to be migrated. Table 7.2 details **PSMTransformations**.

The class **ATLScript** makes it possible to input ATL directly into the TDSL. This can be done to any pipeline model. Table 7.3 details **ATLScripts**.

Table 7.3: TDSL **ATLScript**.

Concepts	Intrinsic Properties			Extrinsic Properties		
	name	**type**	**arity**	**name**	**type**	**arity**
ATLScript *extends* **Transformation**, **PIMTransformation**, **PSMTransformation**	script	EString	1			
	model	MODELS	1			

7.2 The Transformations DSL Grammar and Parser

Unlike the PSMs, the TDSL's current iteration has no support for variable declarations or references. Thus, it is possible to create TDSL models using embedded translation, and T2M transformations are done entirely by the Xtext parser.

Another difference to the other T2M transformations is that we control the TDSL's syntax. When specifying the grammar, our goal was to allow the user to make alterations to the models with the feel of natural language. Listing 7.1 shows a TDSL script used to select a CircleCI workflow and replace a step on the pipeline.

A TDSL script can have three main sections, `before`, `while`, and `after` (listing 7.2). These correspond to the *prePIM*, *onPIM*, and *postPIM* properties of a **TransformationSet**.

```
1 TransformationSet returns TransformationSet:
2         {TransformationSet}
3         ('before' 'translating'? '{' (prePIM+=TSMTransformation)* '}')?
4         ('while' 'translating'? '{' (onPIM+=TIMTransformation)* '}')?
5         ('after' 'translating'? '{' (postPIM+=TSMTransformation)* '}')?
6 ;
```

Listing 7.2: TDSL entry parser rule.

In the `before` and `after` sections, the user can define **PSMTransformations**. These require the meta-model to be specified. In the `while` section, the user can define **PIMTransformations**. Both support defining raw ATL transformations.

7.3 From the Transformations DSL to ATL

Unlike the CI/CD pipeline reengineering process (figure 4.2), the TDSL's reengineering process only uses a single level of models. This is because the TDSL is simpler than the CI/CD platforms. Generating an ATL file directly from the TDSL model also helps deal with some ATL limitations.

ATL has a specific mode for implementing endogenous transformations, which is called refining mode. This mode serves to make in-place alterations to models. Using it, we only need to specify rules for what model elements must be altered. This mode is theoretically ideal for implementing TDSL transformations, as they are always endogenous and not very complex.

However, refining mode lacks several key ATL features needed for the TDSL. For this reason, some TDSL transformations are implemented using the regular ATL mode (out-place). This means we need to specify rules explicitly copying every model element that is unaltered.

This is easier to do directly in text form than transforming the TDSL model to an ATL model and then generating ATL text (or compiling directly from the ATL model). M2T transformations are responsible for most of the TDSL's reengineering logic.

After the ATL file has been generated, the final step of the M2T transformation is to compile it into ATL assembly. This is done by the ATL engine, as detailed in section 8.1.1.

Another difference to highlight is that we generate a separate ATL file for each **Transformation** specified in the **TransformationSet**. This helps make the TDSL more modular.

Chapter 8

ACICDTrip – A Tool for CI/CD Reengineering

The goal of this work is to simplify the CI/CD migration process. Achieving this goes beyond creating the meta-models and defining transformations. There is a need to integrate the various technologies and create software that users can install and run without hassle.

This presents a problem. Usually, MDE software comes in the form of an Eclipse Plugin that extends the IDE's functionality. Eclipse is only used by about 10% of developers according to Stack Overflow's developer surveys [130]. This is due to its usability problems.

Making our transpiler an Eclipse Plugin would severely limit the number of DevOps practitioners it could help. Consequently, we sought to integrate the various MDE technologies used for the meta-model, transformations, and validations in the form of a CLI that could run independently from the Eclipse IDE.

Section 8.1 details how we run MDE technologies outside of Eclipse. Section 8.2 presents the architecture of the CLI.

8.1 Running Eclipse Technologies in Standalone Mode

All of the MDE technologies used in the project can be run in standalone mode, i.e., outside of the Eclipse IDE, by exporting their generated Java packages. However, support for this varies from technology to technology.

The first step in running Eclipse MDE technologies standalone is to generate meta-model Java packages. These packages are the backbone of a program of this kind, as they are used by all other technologies. Eclipse can automatically generate these packages from the Ecore files where we define the meta-models.

We manually register the meta-model Java packages before running any of the MDE technologies. This is because if the meta-models are not registered, MDE technologies will not have the required information to operate on the models.

These packages also allow us to have a Java Object representation of the models. We use this in T2M transformations to create the models as we visit the AST.

Both Xtext and Acceleo are simple to run standalone as they generate Java packages with all required configurations already done. OCL differs from these as the projects only include CompleteOCL files. When running OCL standalone, the file with the constraints is parsed into a constraint map by the OCL package. To validate a model, we must iterate through the constraints in the constraint map and evaluate them individually using the OCL package.

ATL and Epsilon technologies are more complex to run standalone. These are detailed in sections 8.1.1 and 8.1.2 respectively.

8.1.1 ATL

ATL is unique among the MDE technologies in that we have to run two separate features: the model transformations and the compilation of ATL files.

Running ATL Transformations

ATL projects only include the ATL files themselves and their compiled counterparts (in ATL Assembly). They do not include any generated Java packages. This means we need to set up the ATL environment ourselves.

The transformations are run in the **EMFVM**. To configure it, we need to convert the meta-models into ATL's **IModel** format (ATL does not use the generated EMF model packages directly), load the ATL Assembly file with the compiled transformation, and the input model (also converted to the **IModel** format). If running in refining mode, this is all we need to do as transformation is done in-place. Otherwise, we also need to load the output model object. After running the transformation, we convert the output model back to Ecore.

The main issue of running ATL this way is the lack of descriptive error messages if there is a problem with the transformation. When running in Eclipse, ATL outputs a stack trace indicating where the error occurred.

Compiling ATL Transformations

Compiling ATL standalone is a simple matter when compared to running it. We only need to use the compiler package and input the ATL file path and the output ATL Assembly file path.

8.1.2 ECL, ETL and EML

Epsilon has a different philosophy regarding models.

When using Epsilon technologies, every model must be in a different **ResourceSet**. To deal with this, we need to create a separate **ResourceSet** and register the meta-models for each of the three input models.

The models must also have particular names in concordance with the ECL and EML files. The input models are named “Left” and “Right” to work with ECL and EML rules. They must both also be aliased to “Source”, so they can work with the ETL rules that copy non-matching model elements. The output model is called “Target”.

We make use of two modules. The **EclModule**, to run the model comparison and create the match trace, and the **EmlModule** to take the match trace and the input models and run the merger.

8.2 CLI Architecture

The CLI makes use of three key classes to implement the reengineering pipeline logic: **AbstractReverseEngineer** (section 8.2.1), **AbstractForwardEngineer** (section 8.2.2), and **AbstractTransformer** (section 8.2.3). Section 8.2.4 details other major classes used by the CLI.

8.2.1 AbstractReverseEngineer

The **AbstractReverseEngineer** orchestrates the transformations from an input CI/CD pipeline script into a PIM.

To do this it makes use of an **AbstractParser** to parse text to a PSM, PSM **EndogenousAbstractTransformers** to apply TDSL transformations to the PSM, a **ToPIMAbstractTransformer** to transform the PSM to a PIM, and **EndogenousCICDAbstractTransformers** to apply TDSL transformations to the PIM. These are received as constructor arguments.

The CLI can use various **AbstractReverseEngineers** when it receives multiple input files. In this case, the output PIMs are then merged using **AbstractMergers**.

Figure 8.1a shows the **AbstractReverseEngineer** class diagram, including PSM-specific subclasses.

8.2.2 AbstractForwardEngineer

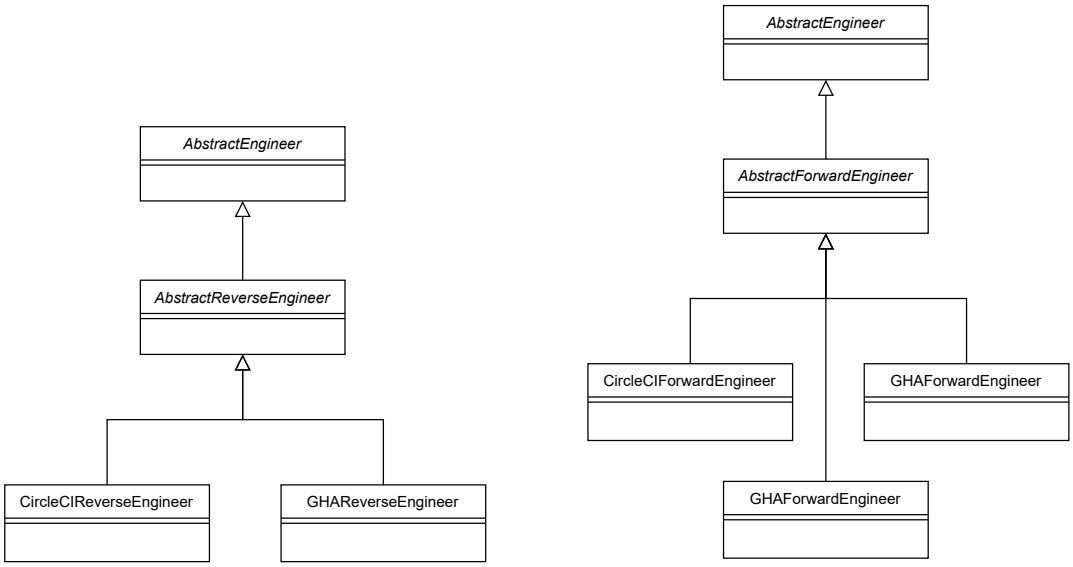
The **AbstractForwardEngineer** orchestrates the transformations from an input PIM to a CI/CD pipeline script.

To do this, it makes use of **EndogenousCICDAbstractTransformers** to apply TDSL transformations to the PIM, a **FromPIMAbstractTransformer** to transform the PIM to a PSM, **EndogenousAbstractTransformers** to apply TDSL transformations to the PSM, and an **AbstractGenerator** to transform the PSM to a CI/CD pipeline script. These are received as constructor arguments.

Figure 8.1b shows the **AbstractForwardEngineer** class diagram, including PSM-specific subclasses.

8.2.3 AbstractTransformer

AbstractTransformers run M2M transformations, implementing the logic from section 8.1.1.



(a) **AbstractReverseEngineer** class diagram. (b) **AbstractForwardEngineer** class diagram.

Figure 8.1: **AbstractEngineer** class diagrams.

ExogenousAbstractTransformers call on **EndogenousAbstractTransformers** to implement exogenous transformation helpers. In the case of **ToPIMAbstractTransformers**, these are called after the exogenous transformations. In **FromPIMAbstractTransformers**, these are called after. The helpers being used are determined by the specific instantiable subclass.

AbstractTransformers can also receive an **AbstractValidator** as a constructor argument. If this is the case, validation is run before transformation.

Figure 8.2 shows the **AbstractTransformer** class diagram.

8.2.4 Other Classes

The other major classes used by the CLI are as follows:

AbstractParsers Run T2M transformations, implementing the logic detailed in section 6.1.

AbstractGenerators Run M2T transformations, calling the Acceleo generators.

AbstractValidators Run OCL validations.

TransformationsDSLtoATLASMCompiler Compiles TDSL to ATL Assembly transformations. To do this, it makes use of an **AbstractParser** for the TDSL and an **AbstractGenerator** for ATL. These are constructor arguments.

AbstractMergers Run model mergers.

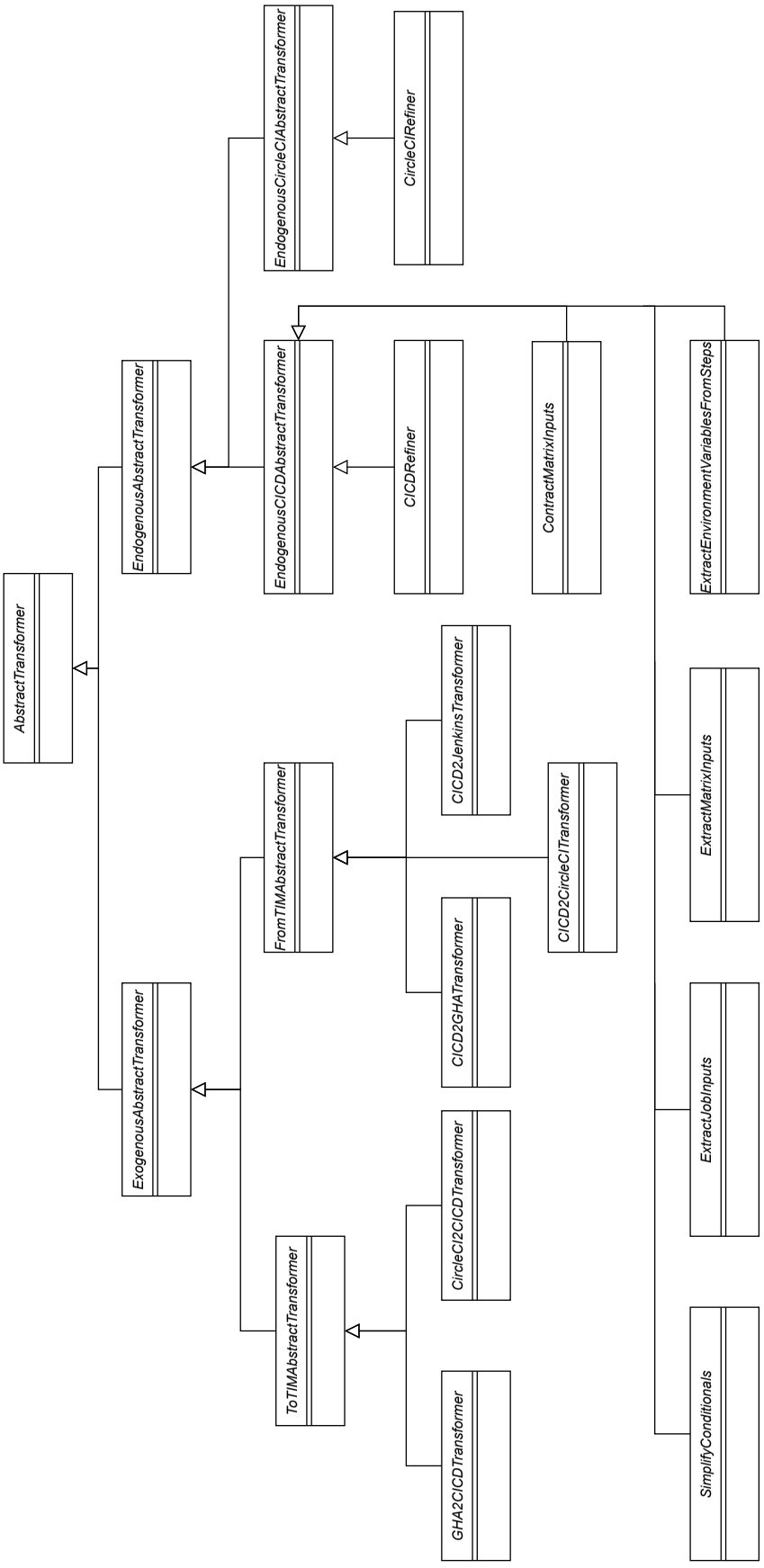


Figure 8.2: **AbstractTransformer** class diagram.

Chapter 9

Evaluation

To help us answer RQ2 and RQ3, we prepared two evaluations.

In section 9.1, we migrate pipelines using ACICDTrip and execute them in their respective repositories. We then compare the execution logs of the original and migrated pipelines to determine if they are equivalent. This helps us get a better perspective of our tool in practice.

In section 9.2, we execute a double-round-trip, where we migrate GHA pipelines to CircleCI and then back into GHA. We then compare the original and migrated GHA pipelines to determine if they are semantically equivalent. This lets us get a better perspective of our tool's functionality for a large number of scripts.

9.1 Evaluating ACICDTrip in Practice

This evaluation helps us ascertain if ACICDTrip can be used to migrate projects in practice. This will mean using the TDSL to change **Plugins** among other transformations.

9.1.1 The Process

To compare execution logs of scripts, we needed not only example scripts to migrate but also the underlying codebase to execute. This proved a challenge, as we did not want to create these projects ourselves, and finding projects that could be built out-of-the-box so we did not lose much time configuring them proved difficult.

We used a set of example projects provided by CircleCI so users can experiment with their platform[1]. These five projects use many PIMM concepts and should be well set up for CI/CD.

To execute this evaluation, we need to fork each CircleCI repository and set up a CircleCI organization so we can run the CircleCI-provided pipelines. We must also migrate these pipelines to another CI/CD platform and execute them. We chose GHA as the destination platform due to its simple setup. As the forked repositories were already hosted on GitHub, we only needed to add the migrated pipelines to the repository in the correct directory to run GHA.

[1] https://circleci.com/docs/examples-and-guides-overview/

After both pipelines have been run, we need to compare their execution logs. This comparison is not as simple as running `diff` between the two logs, as they are formatted differently; we needed to compare them manually. To help with this, we cleaned the logs by removing timestamps of command execution, indentation from lines, and printed lines while pulling docker images. In the case of GitHub Actions, we also removed the logs of permissions granted to the action. Lastly, we removed logs from the executed commands if these were overly verbose and irrelevant to comparing the platforms. Figure 9.1 shows an example of the cleaned logs.

This evaluation depends greatly on the functionality of the TDSL. It will let us manually alter the elements of pipelines that cannot be migrated automatically, allowing us to execute them. To this end, this evaluation also lets us see how easily we can transform pipelines using the TDSL. A transformation common to all the pipelines is adding a **Trigger** so the pipeline can be run manually, as these are not defined in the original CircleCI pipelines. Other transformations required for each project are detailed in the following section.

```
(...)
Operating System: Ubuntu 20.04.6 LTS
OSType: linux
(...)
3.10.5: Pulling from cimg/python
Status: Downloaded newer image for cimg/python:3.10.5
(...)
pip install -r requirements.txt
(...)
pytest
(...)
ERROR openapi_server/test/test_cart_controller.py
ERROR openapi_server/test/test_database.py
ERROR openapi_server/test/test_image_controller.py
ERROR openapi_server/test/test_menu_controller.py
!!!!!!!!!!!!!!!!!!!! Interrupted: 4 errors during collection !
======================== 7 warnings, 4 errors in 0.64s ======
```

(a) CircleCI Python project example logs (abridged).

```
(...)
##[group]Operating System
Ubuntu
22.04.4
LTS
##[endgroup]
(...)
3.10.5: Pulling from cimg/python
Status: Downloaded newer image for cimg/python:3.10.5
(...)
pip install -r requirements.txt
(...)
pytest
(...)
ERROR openapi_server/test/test_cart_controller.py
ERROR openapi_server/test/test_database.py
ERROR openapi_server/test/test_image_controller.py
ERROR openapi_server/test/test_menu_controller.py
!!!!!!!!!!!!!!!!!!!! Interrupted: 4 errors during collection !
======================== 7 warnings, 4 errors in 0.39s ======
(...)
```

(b) GHA Python project example logs (abridged).

Figure 9.1: CircleCI and GitHub cleaned pipeline logs comparison.

9.1.2 Results

Our criteria to determine if logs were equivalent were if the key steps of each pipeline were executed and if they had the same output. This depends on the context of each pipeline.

Java Project

This pipeline incorporates the installation of `Dockerize` to facilitate testing of a Java Spring Boot Server coupled with a PostgreSQL database.

The script can be migrated to GHA using the CLI + TDSL and should have equivalent execution as the original CircleCI script. We must add a **Step** to checkout repository contents and replace a CircleCI-specific **Plugin**. Moreover, to deal with certain differences between how CircleCI and GHA handle Docker container networking, we need to specify ports on the PostgreSQL container and replace the `localhost` address on one of the steps with `172.17.0.1` (the IP of the Docker bridge network). Listing 9.1 shows the TDSL script used.

The problem in this particular case is in the codebase itself, which has the `localhost` address hardcoded in a configuration file instead of as an environment variable. After changing the address in the configuration file, the migrated pipeline executes successfully and with equivalent logs to the original.

No pipeline elements were lost when abstracted to the PIM. Because of the need to alter the source code, this project is only considered a partial success.

```
1  while {
2      add trigger when "input.triggers->isEmpty()" manual
3      add container ports when "input.label = 'cimg/postgres:14.1'" ports
       ↪  {'5432:5432'}
4      insert step 1 on 'maven/test' with checkout  {}
5      replace step 3 on 'maven/test' with command {
6          script 'dockerize -wait tcp://172.17.0.1:5432 -timeout 1m'
7      }
8      replace step 4 on 'maven/test' with command {
9          script 'mvn verify'
10     }
11 }
```

Listing 9.1: Java Project TDSL script.

.NET Project

This pipeline builds, tests, and stores the test results of a .NET application.

To migrate this script, we need to alter the **Agent** the pipeline runs on from a CircleCI Windows orb to a Windows VM. We also need to remove two steps specific to CircleCI. The first one helps connect the CircleCI executor to the repository (this is not needed with GHA). The second one

is a **StoreTestResultsStep** step whose functionality is the same as the **StoreArtifactsStep** when converted to GHA. Listing 9.2 shows the TDSL script used.

The pipeline execution succeeds in both the migrated and original pipelines. The steps executed and their results are equivalent. No pipeline elements were lost when abstracted to the PIM. We also compared the artifacts generated by the pipelines; they were exactly the same, with the exclusion of execution timestamps.

```
1 while {
2     add trigger when "input.triggers->isEmpty()" manual
3     set labels {'windows-latest'}
4     delete step 1 on 'build-and-test'
5     delete step 5 on 'build-and-test'
6 }
```

Listing 9.2: .NET Project TDSL script.

Monorepo Project

This pipeline orchestrates the building and testing of both Python Flask and Vue.js applications within a monorepo environment.

This CircleCI script features two workflows. The PIMM equivalent of a workflow is a **Pipeline**. Unlike the CircleCI meta-model, the platform-independent and GHA meta-models only support one **Pipeline/Workflow** at a time. However, we can use the TDSL to select the workflow to translate and run the program twice, generating two GHA scripts.

For the backend script, we only need to replace a CircleCI **Plugin** with a **Command** to install Python packages. This can be seen in listing 9.3.

```
1 before translating {
2     on circleci select workflow 'backend-test'
3 }
4 while translating {
5     replace step 2 on 'backend-test' with command {
6         script 'pip install -r requirements.txt'
7     }
8 }
```

Listing 9.3: Monorepo Project Backend TDSL script.

The frontend script is similar. We must change a **Plugin** to a **Command** that installs node packages. We also need to add an option to the Docker image used by the original pipeline to change the user. This last change is a particularity of the Docker images CircleCI provides, which

give permissions to a user called `circleci`. Another possibility would have been to use another Docker image.

```
before translating {
    on circleci select workflow frontend
}
while translating {
    set container options when "true" to '--user root'
    replace step 2 on 'frontend-test' with command {
        script 'npm install'
    }
}
```

Listing 9.4: Monorepo Project Frontend TDSL script.

Both the original and migrated pipelines fail out of the box. The failures are related to the path of the `package.json` and `requirements.txt` files provided in the repository. As they fail in the same way, this is considered a success. No pipeline elements were lost when abstracted to the PIM.

NodeJS Project

This script orchestrates the installation of various JavaScript packages required for a Vue.js application. It then executes multiple backend and frontend unit tests utilizing Cypress.

Migrating this script requires changing the Docker image provided by CircleCI to a more standard one on the `test` **Job**. We must also replace a **Plugin** with a **Command** to install packages. On the `cypress/test` **Job**, we need to insert a **Checkout** step and replace a CircleCI **Plugin** with a GHA one. We must also remove the Docker container and run the **Job** directly on the VM. Listing 9.5 shows the TDSL script used.

The original and migrated pipelines both succeed. Having run the same steps and with the same output. No pipeline elements were lost when abstracted to the PIM.

```
while {
    add trigger when "input.triggers->isEmpty()" manual
    set container image when 'true' to 'node:16'
    replace step 2 on 'test' with command {
        script 'yarn install'
    }
    insert step 1 on 'cypress/run' with checkout  {}
    replace step 2 on 'cypress/run' with plugin {
        name 'cypress-io/github-action'
        version 'v6'
        args {
            'command' = 'yarn  run test:e2e --headless'
        }
    }
    run atl on cicd {
    "
-- @path CICD=/d.fe.up.pt.cicd.metamodel/model/CICD.ecore

module cicdRefinement;
create OUT : CICD refining IN : CICD;

rule RemoveContainer {
    from
        input : CICD!DockerContainer (
            input.refImmediateComposite().refImmediateComposite().id =
            ↪  'cypress/run'
        )
    to
        drop
}
    "
    }
}
```

Listing 9.5: NodeJS Project TDSL script.

Python Project

This script is designed to facilitate the building and testing of a Python application. It achieves this by installing several Python packages essential for the application and subsequent execution of multiple unit tests.

Migrating this script is a matter of altering the Docker container user of the image provided by CircleCI and replacing a **Plugin** with a **Command** to install Python packages. Listing 9.6 shows the TDSL script used.

Both the original and migrated pipelines succeed, and the logs are equivalent. No pipeline elements were lost when abstracted to the PIM.

```
while {
    add trigger when "input.triggers->isEmpty()" manual
    set container options when "true" to '--user root'
    replace step 2 on 'build-and-test' with command {
        script 'pip install -r requirements.txt'
    }
}
```

Listing 9.6: Python Project TDSL script.

9.2 Evaluating ACICDTrip for a Large Number of Pipelines

With this evaluation, we want to better understand how ACICDTrip functions for a greater number of real-world scripts. This was not feasible to accomplish using the methodology from section 9.1 due to the need to use the TDSL and run the pipelines.

9.2.1 The Process

There are several challenges in checking whether we can migrate a migrated pipeline back into the original platform without semantic changes (as seen in figure 9.2). This is because the platforms themselves have differences in features. Because of this, we will only attempt to evaluate this for strict-mode-compatible scripts, as in strict mode, the program is meant to exit with an error if it finds a feature that cannot be migrated.

For this evaluation, we used the GHA and CircleCI platforms, which are the ones for which we implemented both the reverse engineering (text to PSM to PIM) and forward engineering (PIM to PSM to text) processes. We started the evaluation with GHA pipeline scripts as we only implemented strict-mode validation for the GHA-to-CircleCI migration.

We randomly selected 10,000 GHA-using repositories from the dataset retrieved for the chapter 3 study. In these repositories, we found 25,487 GHA scripts. We could migrate 22,684 (89,0%) of these to CircleCI in normal mode, but only 4,091 (16.1%) were strict-mode-compatible. We migrated the 4,091 scripts back into GHA. Most of the pipelines that failed validation (82.3%) were due to references to variables not yet supported by the PIMM. The most common examples of these variables are user-defined secrets (e.g., API tokens) and commit information (e.g., SHA).

We used the same TDSL script for all pipelines. This script was used in the GHA-to-CircleCI migration, and the only transformation it makes is setting the CircleCI version to “2.1” so the CircleCI pipeline is valid (listing 9.7).

```
after translating {
    on circleci set version to "2.1"
}
```

Listing 9.7: Double Round-Trip TDSL script.

GHA uses a YAML-based syntax. To compare the original and migrated pipelines, we start by running `yamldiff`[2], a CLI to compare two YAML or JSON files.

However, while the GHA DSL is based on YAML, it also extends it with expressions. GHA also provides various shorthands to speed up pipeline definition. For example, the aforementioned way of defining the same trigger as `on: push`, `on: [push]`, or `on: {push: null}`. This means the migrated pipeline may be different in YAML but the same in GHA.

Consequently, we need to filter out differences from the output of `yamldiff` if they do not impact the GHA pipeline. We use regular expressions to evaluate certain differences and eliminate them if they match. What follows is a list of the YAML differences we ignored:

- **Trailing whitespace** - we ignore any differences in trailing whitespaces in strings between the original and generated files.
- **String to one-item list** - `key: string` is the same as `key: [string]`.
- **List to empty map** - `key: [listvalue]` is the same as `key: map[listvalue:<nil>]`.
- **String to empty map** - `key: string` is the same as `key: map[string:<nil>]`.
- **Empty map to null** - `key: <nil>` is the same as not having `key` at all.
- **String output to map** - `*.outputs.output-name: value` is the same as `*.outputs.output-name: map[value: value]`.
- **Full variable reference** - In GHA, it is possible to omit part of a variable reference, e.g., we can refer to `jobs.job-0.env.ENV_VAR` just as `env.ENV_VAR`. The transpiler always generates the full variable reference.
- **If without brackets** - GHA lets users omit the "${{ (...) }}" syntax that denotes an expression when defining a conditional for flow control. The transpiler always generates expressions with "${{ (...) }}" syntax.
- **Container image** - `*.container: value` is the same as `*.container: map[image: value]`.

[2] https://github.com/sahilm/yamldiff

```yaml
name: "Workflow"

on: push

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
```

(a) Input GHA pipeline script.

```yaml
version: 2.1

jobs:
  build:
    machine:
      image: "ubuntu-latest"
    steps:
      - checkout:

workflows:
  version: 2.1
  Workflow:
    jobs:
      -
        build:
```

(b) Intermediate CircleCI pipeline script.

```yaml
name: "Workflow"

jobs:
  build:
    name: "build"
    runs-on:
    - "ubuntu-latest"
    steps:
    - uses: "actions/checkout@v4"
```

(c) Output GHA pipeline script.

Figure 9.2: Example of a GHA pipeline (figure 9.2a) being migrated to CircleCI (figure 9.2b) and then back into GHA (figure 9.2c).

9.2.2 Results

There are some limitations to using CircleCI as an intermediary technology in this evaluation. CircleCI does not define most **Triggers** in the pipeline script (it does this in the platform settings), meaning we lose **Trigger** information when migrating the GHA pipeline. Display names of **Steps** are also altered in certain situations. We ignore differences that stem from these limitations as a fully-developed ACICDTrip would have tighter integration with the CircleCI platform and migrate the **Triggers**, and the **Step** display names have no bearing on pipeline execution.

The abstraction of GHA plugins like `actions/checkout` to PIMM **Steps** like **Checkout** means we lose version information of these **Plugins** in the migration (ACICDTrip generates pipelines with the latest version). These differences are moot and only a result of this particular kind of evaluation. Some platforms use native **Steps** for this functionality, while others use **Plugins**. If the platform we are migrating to uses native **Steps** (e.g., CircleCI), the version is irrelevant; if it uses **Plugins** (e.g., Jenkins), we do not want to use another platform's **Plugin**'s version. The abstraction lets us migrate these steps accurately and automatically.

Taking this into account, 3,316 of the scripts suffered no semantic change. This gives us an 81,1% successful migration rate.

Of 775 pipelines with semantic change (pipelines may have multiple changes):

- 404 had **Plugins** lose arguments when being migrated to **Checkouts**, **Artifacts**, or **Caches**. This is because they have extra functionality not supported by the PIMM.
- 31 had lost **Plugin** environment variables. CircleCI does not natively support environment variables in **Orb** steps. We send these as arguments instead. This avoids loss of information as, when changing the GHA **Plugin** to a CircleCI one, the CircleCI one may instead take these values as arguments.
- 100 had differences because strings were parsed as floating point numbers. This happens most in **Plugins** as we have no information on the type of the argument we are parsing. The string value "3.10" is parsed as a float 3.1. This causes changes mostly when the **Plugin** argument indicates a version of some kind, as 3.10 should be read as a string in that context.
- 16 had differences due to encoding. The transpiler only supports UTF-8.
- 54 had macOS version mismatches as CircleCI does not directly store the macOS version.
- 252 had differences that are not easily classifiable. These should be seen as the result of bugs in the current version of the transpiler.

9.3 Addressing RQ2

The second research question prompts us to evaluate whether the PIMM can be the basis for accurate translation of CI/CD pipelines between platforms. The evaluations allow us to answer this question positively.

In section 9.1, all of the pipelines could be migrated to GHA. The PIMM supported the pipelines completely, and the transformations we defined were capable of accurately migrating from CircleCI to the PIMM and then to GHA. We needed to use the TDSL for some transformations that could not be done automatically. Still, all of these transformations except selecting the pipeline to migrate were done on the PIM. There was no need to substantially interact with the CircleCI or GHA models in the migration process, even if there was a need to interact with the codebase in one project. We also made no alterations to the generated scripts.

In section 9.2, there are many pipelines that cannot yet be migrated in strict mode. Still, the vast majority (81.1%) of pipelines supported by strict mode can be migrated with no semantic alteration.

Any reengineering process is lossy. Modeling is an abstraction, and consequently, a model can never contain all the information of the original object. This is doubly true for us, as we are trying to model CI/CD pipelines in a manner that is independent of their underlying platform. Inevitably, there will be elements we cannot represent and details of the elements we do represent that we will lose.

The glaring examples of this are the **Checkout**, **Artifact**, and **Cache** steps. This abstraction lets us automatically migrate core CI/CD functionality between platforms, but it means we lose information about the **Plugin** version and arguments of some platforms.

The version alteration is irrelevant to migrating between platforms as long as we guarantee that the functionality of the step is the same. However, we cannot say the same for the loss of arguments. This is the most common semantic difference amongst the 775 altered files in section 9.2.2. Further PIMM development could address some of these, but there are limits.

For example, GHA's `actions/checkout` package has an optional argument to checkout the repository's submodules. CircleCI's `checkout` does not support this functionality, but it can be mimicked by running certain shell commands. If we added this functionality to the PIMM's **Checkout**, we could add those commands when running the PIM-to-CircleCI transformation. However, this would make the CircleCI-to-PIM transformation much more complex because of the need to group multiple CircleCI steps into one PIMM step. Moreover, if we try to optimize the PIMM to one particular platform migration, we could lessen its ability to represent other platforms' pipelines as it becomes more platform-specific.

Overall, while there is room for further development of the PIMM by adding new concepts and detailing some existing ones, the evaluation shows that the current abstraction level is appropriate to allow accurate migration between platforms.

9.4 Addressing RQ3

The third research question asks if CI/CD migration can be fully automated. We find the answer to this question to be negative.

Firstly, although different CI/CD platforms have much common functionality, there are still significant differences in feature sets. For example, GHA's tight integration with GitHub lets users

set up permissions for pipelines in the pipeline itself. In other platforms, these permissions have to be defined when generating a token for the platform to access the repository. CircleCI has more options for users to control execution flow, like calling functions, defining multiple pipelines in one script, and easily reusing jobs. These different feature sets are one of the main motivators for migration [121].

This means there will always be pipelines that cannot be wholly migrated from one platform to another. Section 9.2 shows this can happen even in the pipelines supported by strict mode, as 404 scripts had **Plugins** lose arguments because of extra functionality.

Moreover, migrating CI/CD sometimes necessitates changes to the codebase and changes that can only be done with context-specific knowledge. Section 9.1 has an example of this. Migrating meant changing the address and ports of a Docker container, which can only be done with knowledge of the ports used by the container. This address also needed to be changed in the codebase.

Finally, **Plugins** need to be changed between platforms. In theory, this could be done automatically, but there is no guarantee there will always be a corresponding **Plugin** in another platform. This would imply manual work to replace it with multiple **Plugins** or **Commands**.

9.5 Threats to Validity

The PIMM was devised based on several of the most popular CI/CD platforms from chapter 3 and, as such, should be able to represent most pipelines. Still, it is conceivable that CI/CD pipelines implemented in more niche platforms may not be representable.

The CircleCI projects used in section 9.1 are not representable of real-world pipelines as they are part of sample projects. Still, they use most core PIMM functionality other than **Matrices** and show how the PIMM can be used to migrate platforms.

Regarding section 9.2's evaluation. We show that our tool can be self-consistent in its transformations from platform to platform. However, even in the 81,1% of cases where there was no semantically relevant alteration, we cannot guarantee that the intermediate script of the double round trip is semantically equivalent in the intermediate platform. As seen in section 9.2, changing platforms requires some manual work, as there are some particularities to each platform that are too low-level to be considered in the PIMM. The successful migration rate and other statistics also only apply to the GHA-to-CircleCI migration and cannot be generalized for all migrations.

The 10,000 GHA-using repositories used for this evaluation were selected at random. While they are all real-world projects made public by users, we cannot guarantee that the scripts we retrieved from these repositories are in use and representative of their projects. The large sample size should minimize these concerns.

Chapter 10

Conclusions and Future Work

With our work, we found there are enough core concepts common to diverse CI/CD platforms to define a common language, the meta-model we propose. This allows the full migration of existing pipelines in many situations. Nevertheless, a fully automated reengineering process does not seem feasible in all cases.

Often, changing technologies requires some manual work, as there are some particularities to each platform that are too low-level to be considered in the PIMM. For example, GHA scripts require at least one **Trigger** definition to be well-formed; however, when translating from CircleCI, there is often missing information related to **Triggers**.

To aid with these manual changes, we designed an initial DSL. In fact, a fully-fledged TDSL could also be a *lingua franca* for CI/CD pipelines, letting developers write pipelines without being concerned about the syntax of the technology they will end up using. Future TDSL versions may also interact with the PSM-to-PIM and PIM-to-PSM transformations; this way, it would be possible for the user to configure the version of the platform-specific plugins that **Checkout**, **Artifact**, and **Cache** steps get migrated to.

There is also room for further development of the PIMM. The next development path should be adding support for user-defined secrets and other relevant pipeline variables, as this is revealed to be a major current limitation. These features are present in multiple CI/CD platforms, so the PIMM would stay platform-independent.

For better usability, future versions of ACICDTrip should provide more user feedback about the migration process, such as listing the elements of the pipeline the transpiler cannot migrate and will require user input. This can be done by defining more model validations and checking their constraints.

References

[1] *About the MOF Query/View/Transformation Specification Version 1.3*. URL: https://www.omg.org/spec/QVT/1.3/About-QVT (visited on 12/27/2023).

[2] *Acceleo | Home*. URL: https://eclipse.dev/acceleo/ (visited on 12/27/2023).

[3] Adetokunbo A A Adenowo and Basirat A Adenowo. "Software Engineering Methodologies: A Review of the Waterfall Model and Object-Oriented Approach". In: *International Journal of Scientific & Engineering Research* 4.7 (2013), pp. 427–434.

[4] Hanieh Alipour and Yan Liu. "Model Driven Deployment of Auto-Scaling Services on Multiple Clouds". In: *2018 IEEE International Conference on Software Architecture Companion (ICSA-C)*. 2018 IEEE International Conference on Software Architecture Companion (ICSA-C). Seattle, WA: IEEE, Apr. 2018, pp. 93–96. ISBN: 978-1-5386-6585-5. DOI: 10.1109/ICSA-C.2018.00033. URL: https://ieeexplore.ieee.org/document/8432188/ (visited on 06/25/2024).

[5] *AppVeyor*. URL: https://www.appveyor.com/ (visited on 06/21/2024).

[6] S.A.I.B.S. Arachchi and Indika Perera. "Continuous Integration and Continuous Delivery Pipeline Automation for Agile Software Project Management". In: *2018 Moratuwa Engineering Research Conference (MERCon)*. 2018 Moratuwa Engineering Research Conference (MERCon). May 2018, pp. 156–161. DOI: 10.1109/MERCon.2018.8421965. URL: https://ieeexplore.ieee.org/abstract/document/8421965?casa_token=vM2GUu7Bq7cAAAAA:wZUfCkE4AhbTQo8WZkG7ECIfw0KAsoPmDqj5x9DVaOYkfcc6oSSxOSFsSPUyzahjRvti4ELGc28 (visited on 11/29/2023).

[7] Matej Artač et al. "Model-driven continuous deployment for quality DevOps". In: *Proceedings of the 2nd International Workshop on Quality-Aware DevOps*. ISSTA '16: International Symposium on Software Testing and Analysis. Saarbrücken Germany: ACM, July 21, 2016, pp. 40–41. ISBN: 978-1-4503-4411-1. DOI: 10.1145/2945408.2945417. URL: https://dl.acm.org/doi/10.1145/2945408.2945417 (visited on 01/04/2024).

[8] *Automating migration with GitHub Actions Importer*. GitHub Docs. URL: https://docs.github.com/en/actions/migrating-to-github-actions/automated-migrations/automating-migration-with-github-actions-importer (visited on 12/06/2023).

[9] Zia Babar, Alexei Lapouchnian, and Eric Yu. "Modeling DevOps Deployment Choices Using Process Architecture Design Dimensions". In: *The Practice of Enterprise Modeling*. Ed. by Jolita Ralyté, Sergio España, and Óscar Pastor. Lecture Notes in Business Information Processing. Cham: Springer International Publishing, 2015, pp. 322–337. ISBN: 978-3-319-25897-3. DOI: 10.1007/978-3-319-25897-3_21.

[10] Kiyana Bahadori and Tullio Vardanega. "DevOps Meets Dynamic Orchestration". In: *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*. Ed. by Jean-Michel Bruel, Manuel Mazzara, and Bertrand Meyer. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 142–154. ISBN: 978-3-030-06019-0. DOI: 10.1007/978-3-030-06019-0_11.

[11] K. Beck. "Embracing change with extreme programming". In: *Computer* 32.10 (Oct. 1999), pp. 70–77. ISSN: 00189162. DOI: 10.1109/2.796139. URL: http://ieeexplore.ieee.org/document/796139/ (visited on 12/09/2023).

[12] Moritz Beller, Georgios Gousios, and Andy Zaidman. "Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub". In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR). Buenos Aires, Argentina: IEEE, May 2017, pp. 356–367. ISBN: 978-1-5386-1544-7. DOI: 10.1109/MSR.2017.62. URL: http://ieeexplore.ieee.org/document/7962385/ (visited on 12/17/2023).

[13] Francisco Javier Bermúdez Ruiz, Jesús García Molina, and Oscar Díaz García. "On the application of model-driven engineering in data reengineering". In: *Information Systems* 72 (Dec. 1, 2017), pp. 136–160. ISSN: 0306-4379. DOI: 10.1016/j.is.2017.10.004. URL: https://www.sciencedirect.com/science/article/pii/S0306437915300508 (visited on 12/26/2023).

[14] N. Bohr. "On the constitution of atoms and molecules". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 26.151 (July 1913), pp. 1–25. ISSN: 1941-5982, 1941-5990. DOI: 10.1080/14786441308634955. URL: https://www.tandfonline.com/doi/full/10.1080/14786441308634955 (visited on 12/19/2023).

[15] Richard C. Gronback Boldt Nick. *Graphical Modeling Framework | The Eclipse Foundation*. URL: https://eclipse.dev/modeling/gmp/ (visited on 12/27/2023).

[16] Francis Bordeleau et al. "Towards Modeling Framework for DevOps: Requirements Derived from Industry Use Case". In: *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*. Ed. by Jean-Michel Bruel, Manuel Mazzara, and Bertrand Meyer. Lecture Notes in Computer Science. Cham:

Springer International Publishing, 2020, pp. 139–151. ISBN: 978-3-030-39306-9. DOI: 10.1007/978-3-030-39306-9_10.

[17] Hayet Brabra et al. "Model-Driven Orchestration for Cloud Resources". In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 2019 IEEE 12th International Conference on Cloud Computing (CLOUD). Milan, Italy: IEEE, July 2019, pp. 422–429. ISBN: 978-1-72812-705-7. DOI: 10.1109/CLOUD.2019.00074. URL: https://ieeexplore.ieee.org/document/8814534/ (visited on 06/25/2024).

[18] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Cham: Springer International Publishing, 2017. ISBN: 978-3-031-01421-5. DOI: 10.1007/978-3-031-02549-5. URL: https://link.springer.com/10.1007/978-3-031-02549-5 (visited on 12/19/2023).

[19] Fabio Calefato, Filippo Lanubile, and Luigi Quaranta. "A Preliminary Investigation of MLOps Practices in GitHub". In: *Proceedings of the 16th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '22. Helsinki, Finland: Association for Computing Machinery, 2022, pp. 283–288. ISBN: 9781450394277. DOI: 10.1145/3544902.3546636. URL: https://doi.org/10.1145/3544902.3546636.

[20] G. Casale et al. "RADON: rational decomposition and orchestration for serverless computing". In: *SICS Software-Intensive Cyber-Physical Systems* 35.1 (Aug. 2020), pp. 77–87. ISSN: 2524-8510, 2524-8529. DOI: 10.1007/s00450-019-00413-w. URL: http://link.springer.com/10.1007/s00450-019-00413-w (visited on 06/25/2024).

[21] Giuliano Casale et al. "DICE: Quality-Driven Development of Data-Intensive Cloud Applications". In: *2015 IEEE/ACM 7th International Workshop on Modeling in Software Engineering*. 2015 IEEE/ACM 7th International Workshop on Modeling in Software Engineering (MiSE). Florence: IEEE, May 2015, pp. 78–83. ISBN: 978-1-4673-7055-4. DOI: 10.1109/MiSE.2015.21. URL: https://ieeexplore.ieee.org/document/7167407/ (visited on 12/07/2023).

[22] Stéphanie Challita et al. "Model-based cloud resource management with TOSCA and OCCI". In: *Software and Systems Modeling* 20.5 (Oct. 1, 2021), pp. 1609–1631. ISSN: 1619-1374. DOI: 10.1007/s10270-021-00869-y. URL: https://doi.org/10.1007/s10270-021-00869-y (visited on 06/25/2024).

[23] *Chef Software DevOps Automation Solutions | Chef*. Chef Software. URL: https://www.chef.io/ (visited on 01/04/2024).

[24] Lianping Chen. "Continuous Delivery: Huge Benefits, but Challenges Too". In: *IEEE Software* 32.2 (Mar. 2015). Conference Name: IEEE Software, pp. 50–54. ISSN: 1937-4194. DOI: 10.1109/MS.2015.27. URL: https://ieeexplore.ieee.org/abstract

/document/7006384?casa_token=zca1QQTWpkwAAAAA:3KlDQ3L7xFSm4tG1NGoTPSWU5IKRCyOmVFYurgy0421HIz4u77Jm9q1Hcmlwy8ZMTI9tbKxtB-8 (visited on 11/29/2023).

[25] Wei Chen et al. "MORE: A Model-Driven Operation Service for Cloud-Based IT Systems". In: *2016 IEEE International Conference on Services Computing (SCC)*. 2016 IEEE International Conference on Services Computing (SCC). San Francisco, CA, USA: IEEE, June 2016, pp. 633–640. ISBN: 978-1-5090-2628-9. DOI: 10.1109/SCC.2016.88. URL: http://ieeexplore.ieee.org/document/7557508/ (visited on 01/04/2024).

[26] E.J. Chikofsky and J.H. Cross. "Reverse engineering and design recovery: a taxonomy". In: *IEEE Software* 7.1 (Jan. 1990). Conference Name: IEEE Software, pp. 13–17. ISSN: 1937-4194. DOI: 10.1109/52.43044. URL: https://ieeexplore.ieee.org/abstract/document/43044 (visited on 12/26/2023).

[27] Federico Ciccozzi et al. "Blended Modelling - What, Why and How". In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). Munich, Germany: IEEE, Sept. 2019, pp. 425–430. ISBN: 978-1-72815-125-0. DOI: 10.1109/MODELS-C.2019.00068. URL: https://ieeexplore.ieee.org/document/8904858/ (visited on 12/27/2023).

[28] *CircleCI*. URL: https://circleci.com (visited on 06/21/2024).

[29] *CircleCI Orbs*. CircleCI. URL: https://circleci.com/orbs/ (visited on 12/17/2023).

[30] *Codefresh*. URL: https://codefresh.io/ (visited on 06/21/2024).

[31] Eldan Cohen and Mariano P. Consens. "Large-Scale Analysis of the Co-commit Patterns of the Active Developers in GitHub's Top Repositories". In: *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 2018, pp. 426–436.

[32] Alessandro Colantoni, Luca Berardinelli, and Manuel Wimmer. "DevOpsML: towards modeling DevOps processes and platforms". In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems. Virtual Event Canada: ACM, Oct. 16, 2020, pp. 1–10. ISBN: 978-1-4503-8135-2. DOI: 10.1145/3417990.3420203. URL: https://dl.acm.org/doi/10.1145/3417990.3420203 (visited on 12/07/2023).

[33] Alessandro Colantoni et al. "Towards blended modeling and simulation of DevOps processes: the Keptn case study". In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS '22: ACM/IEEE 25th International Conference on Model Driven Engineering Languages

and Systems. Montreal Quebec Canada: ACM, Oct. 23, 2022, pp. 784–792. ISBN: 978-1-4503-9467-3. DOI: 10.1145/3550356.3561597. URL: https://dl.acm.org/doi/10.1145/3550356.3561597 (visited on 12/07/2023).

[34] *Concourse*. URL: https://concourse-ci.org/ (visited on 06/21/2024).

[35] Ana C. Franco Da Silva et al. "OpenTOSCA for IoT: Automating the Deployment of IoT Applications based on the Mosquitto Message Broker". In: *Proceedings of the 6th International Conference on the Internet of Things*. IoT'16: The 6th International Conference on the Internet of Things. Stuttgart Germany: ACM, Nov. 7, 2016, pp. 181–182. ISBN: 978-1-4503-4814-0. DOI: 10.1145/2991561.2998464. URL: https://dl.acm.org/doi/10.1145/2991561.2998464 (visited on 01/04/2024).

[36] Alexandre Decan et al. "On the Use of GitHub Actions in Software Development Repositories". In: *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME). Limassol, Cyprus: IEEE, Oct. 2022, pp. 235–245. ISBN: 978-1-66547-956-1. DOI: 10.1109/ICSME55016.2022.00029. URL: https://ieeexplore.ieee.org/document/9978190/ (visited on 01/31/2024).

[37] Elisabetta Di Nitto et al., eds. *Model-Driven Development and Operation of Multi-Cloud Applications: The MODAClouds Approach*. SpringerBriefs in Applied Sciences and Technology. Cham: Springer International Publishing, 2017. ISBN: 978-3-319-46031-4. DOI: 10.1007/978-3-319-46031-4. URL: http://link.springer.com/10.1007/978-3-319-46031-4 (visited on 01/04/2024).

[38] *Drone*. URL: https://www.drone.io/ (visited on 06/21/2024).

[39] Thomas F. Düllmann, Oliver Kabierschke, and André van Hoorn. "StalkCD: A Model-Driven Framework for Interoperability and Analysis of CI/CD Pipelines". In: *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2021, pp. 214–223. DOI: 10.1109/SEAA53835.2021.00035.

[40] Paul M. Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Google-Books-ID: PV9qfEdv9L0C. Pearson Education, June 29, 2007. 313 pp. ISBN: 978-0-321-63014-8.

[41] *Ecore - Eclipsepedia*. URL: https://wiki.eclipse.org/Ecore (visited on 12/27/2023).

[42] Badr El Khalyly et al. "A new metamodel approach of CI/CD applied to Internet of Things Ecosystem". In: *2020 IEEE 2nd International Conference on Electronics, Control, Optimization and Computer Science (ICECOCS)*. 2020 IEEE 2nd International Conference on Electronics, Control, Optimization and Computer Science (ICECOCS). Kenitra, Morocco: IEEE, Dec. 2, 2020, pp. 1–6. ISBN: 978-1-72816-921-7. DOI: 10.1109/ICECOCS50124.2020.9314485. URL: https://ieeexplore.ieee.org/document/9314485/ (visited on 12/07/2023).

[43] *EMFText*. EMFText. URL: http://devboost.github.io/EMFText/ (visited on 12/27/2023).

[44] *Epsilon*. URL: https://eclipse.dev/epsilon/ (visited on 06/23/2024).

[45] *Epsilon Documentation*. URL: https://eclipse.dev/epsilon/doc/ (visited on 06/23/2024).

[46] Nicolas Ferry et al. "CloudMF: Model-Driven Management of Multi-Cloud Applications". In: *ACM Transactions on Internet Technology* 18.2 (May 31, 2018), pp. 1–24. ISSN: 1533-5399, 1557-6051. DOI: 10.1145/3125621. URL: https://dl.acm.org/doi/10.1145/3125621 (visited on 06/25/2024).

[47] Nicolas Ferry et al. "ENACT: Development, Operation, and Quality Assurance of Trustworthy Smart IoT Systems". In: *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*. Ed. by Jean-Michel Bruel, Manuel Mazzara, and Bertrand Meyer. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 112–127. ISBN: 978-3-030-06019-0. DOI: 10.1007/978-3-030-06019-0_9.

[48] *Finding and customizing actions*. GitHub Docs. URL: https://docs.github.com/en/actions/learn-github-actions/finding-and-customizing-actions (visited on 12/17/2023).

[49] Franck Fleurey et al. "Model-Driven Engineering for Software Migration in a Large Industrial Context". In: *Model Driven Engineering Languages and Systems*. Ed. by Gregor Engels et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2007, pp. 482–497. ISBN: 978-3-540-75209-7. DOI: 10.1007/978-3-540-75209-7_33.

[50] André Flores and Hugo Gião. *ACICDTrip Repository*. June 2024. DOI: 10.5281/zenodo.11922315. URL: https://github.com/DreFlo/auto-cicd-migration/releases/tag/v0.1.1.

[51] André Flores et al. *A Two-Level Model-Driven Approach for Reengineering CI/CD Pipelines*. Manuscript submitted for publication. 2024. URL: https://dreflo.github.io/auto-cicd-migration/doc/thesis/inforum.pdf.

[52] Martin Fowler. *Continuous Integration*. martinfowler.com. URL: https://martinfowler.com/articles/continuousIntegration.html (visited on 11/29/2023).

[53] Hugo Gião, Jácome Cunha, and Rui Pereira. *Model-Driven Approaches for DevOps: A Systematic Literature Review*. Manuscript submitted for publication. 2023. URL: https://h4g0.github.io/Survey_MDE_DevOps.pdf.

[54] Hugo Gião et al. 2023. URL: https://anonymous.4open.science/r/DevOps-Repositories-122E.

[55] Hugo Gião et al. 2023. URL: https://anonymous.4open.science/r/github-devops-mining-C0D7.

[56] Hugo Gião et al. *A Meta-Model to Support the Migration and Evolution of CI/CD Pipelines*. Manuscript submitted for publication. 2024. URL: https://dreflo.github.io/auto-cicd-migration/doc/thesis/models.pdf.

[57] Hugo Gião et al. *Chronicles of CI/CD: A Deep Dive into its Usage Over Time*. Manuscript submitted for publication. 2023. URL: https://dreflo.github.io/auto-cicd-migration/doc/thesis/msr.pdf.

[58] Hugo Gião et al. *CI/CD repos with tools*. Nov. 2023. DOI: 10.6084/m9.figshare.24578740.v2. URL: https://figshare.com/articles/dataset/Untitled_Item/24578740.

[59] Hugo Gião et al. *CI/CD repositories from GitHub*. Nov. 2023. DOI: 10.6084/m9.figshare.24578746.v1. URL: https://figshare.com/articles/dataset/CI_CD_repositories_from_GitHub/24578746.

[60] Hugo Gião et al. *CI/CD repositories with tool history*. Nov. 2023. DOI: 10.6084/m9.figshare.24578752.v1. URL: https://figshare.com/articles/dataset/CI_CD_repositories_with_tool_history/24578752.

[61] *GitHub Actions*. URL: https://github.com/features/actions (visited on 06/21/2024).

[62] *GitHub Actions pipeline integration*. URL: https://codefresh.io/docs/docs/integrations/github-actions/ (visited on 12/06/2023).

[63] *GitLab CI/CD*. URL: https://docs.gitlab.com/ee/ci (visited on 06/21/2024).

[64] *GoCD*. URL: https://www.gocd.org/index.html (visited on 06/21/2024).

[65] Mehdi Golzadeh, Alexandre Decan, and Tom Mens. "On the rise and fall of CI services in GitHub". In: *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). Honolulu, HI, USA: IEEE, Mar. 2022, pp. 662–672. ISBN: 978-1-66543-786-8. DOI: 10.1109/SANER53432.2022.00084. URL: https://ieeexplore.ieee.org/document/9825792/ (visited on 11/12/2023).

[66] Marvin Grieger et al. "Concept-Based Engineering of Situation-Specific Migration Methods". In: *Software Reuse: Bridging with Social-Awareness*. Ed. by Georgia M. Kapitsaki and Eduardo Santana de Almeida. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 199–214. ISBN: 978-3-319-35122-3. DOI: 10.1007/978-3-319-35122-3_14.

[67] Richard Gronback. *Eclipse Modeling Project | The Eclipse Foundation*. URL: https://eclipse.dev/modeling/emf/ (visited on 12/27/2023).

[68] Michele Guerriero et al. “A Model-Driven DevOps Framework for QoS-Aware Cloud Applications”. In: *2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. 2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC). Timisoara, Romania: IEEE, Sept. 2015, pp. 345–351. ISBN: 978-1-5090-0461-4. DOI: 10.1109/SYNASC.2015.60. URL: http://ieeexplore.ieee.org/document/7426104/ (visited on 06/25/2024).

[69] Mubin Ul Haque, Leonardo Horn Iwaya, and M. Ali Babar. “Challenges in Docker Development: A Large-Scale Study Using Stack Overflow”. In: *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ESEM ’20. Bari, Italy: Association for Computing Machinery, 2020. ISBN: 9781450375801. DOI: 10.1145/3382494.3410693. URL: https://doi.org/10.1145/3382494.3410693.

[70] Jordan Henkel et al. “Learning from, Understanding, and Supporting DevOps Artifacts for Docker”. In: *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. New York, NY, United States: Association for Computing Machinery, 2020, pp. 38–49. DOI: 10.1145/3377811.3380406.

[71] Willem-Jan van den Heuvel et al. “ChainOps for Smart Contract-Based Distributed Applications”. In: *Business Modeling and Software Design*. Ed. by Boris Shishkov. Lecture Notes in Business Information Processing. Cham: Springer International Publishing, 2021, pp. 374–383. ISBN: 978-3-030-79976-2. DOI: 10.1007/978-3-030-79976-2_25.

[72] Michael Hilton et al. “Usage, costs, and benefits of continuous integration in open-source projects”. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE’16: ACM/IEEE International Conference on Automated Software Engineering. Singapore Singapore: ACM, Aug. 25, 2016, pp. 426–437. ISBN: 978-1-4503-3845-5. DOI: 10.1145/2970276.2970358. URL: https://dl.acm.org/doi/10.1145/2970276.2970358 (visited on 12/17/2023).

[73] Helena Holmstrom et al. “Global Software Development Challenges: A Case Study on Temporal, Geographical and Socio-Cultural Distance”. In: *2006 IEEE International Conference on Global Software Engineering (ICGSE’06)*. 2006, pp. 3–11. DOI: 10.1109/ICGSE.2006.261210.

[74] Geir Horn and Pawel Skrzypek. “MELODIC: Utility Based Cross Cloud Deployment Optimisation”. In: *2018 32nd International Conference on Advanced Information Networking and Applications Workshops (WAINA)*. 2018 32nd International Conference on Advanced Information Networking and Applications Workshops (WAINA). Krakow: IEEE, May 2018, pp. 360–367. ISBN: 978-1-5386-5395-1. DOI: 10.1109/WAINA.2018.00112. URL: https://ieeexplore.ieee.org/document/8418097/ (visited on 01/04/2024).

[75] Jerome Hugues et al. “TwinOps - DevOps meets model-based engineering and digital twins for the engineering of CPS”. In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS ’20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems. Virtual Event Canada: ACM, Oct. 16, 2020, pp. 1–5. ISBN: 978-1-4503-8135-2. DOI: 10.1145/3417990.3421446. URL: https://dl.acm.org/doi/10.1145/3417990.3421446 (visited on 12/07/2023).

[76] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, July 27, 2010. 956 pp. ISBN: 978-0-321-67022-9.

[77] *Introduction to CircleCI migration - CircleCI*. URL: https://circleci.com/docs/migration-intro/ (visited on 12/06/2023).

[78] Ramtin Jabbari et al. “What is DevOps? A Systematic Mapping Study on Definitions and Practices”. In: *Proceedings of the Scientific Workshop Proceedings of XP2016*. XP ’16 Workshops. Edinburgh, Scotland, UK: Association for Computing Machinery, 2016. ISBN: 9781450341349. DOI: 10.1145/2962695.2962707. URL: https://doi.org/10.1145/2962695.2962707.

[79] Santiago P. Jácome-Guerrero, Marcelo Ferreira, and Alexandra Corral. “Software Development Tools in Model-Driven Engineering”. In: *2017 5th International Conference in Software Engineering Research and Innovation (CONISOFT)*. 2017 5th International Conference in Software Engineering Research and Innovation (CONISOFT). Mérida, Mexico: IEEE, Oct. 2017, pp. 140–148. ISBN: 978-1-5386-3956-6. DOI: 10.1109/CONISOFT.2017.00024. URL: http://ieeexplore.ieee.org/document/8337945/ (visited on 12/26/2023).

[80] *Jenkins*. URL: https://www.jenkins.io (visited on 06/21/2024).

[81] *Jenkins pipeline integration/migration*. URL: https://codefresh.io/docs/docs/integrations/jenkins-integration/ (visited on 12/06/2023).

[82] Miguel Jiménez et al. “DevOps’ shift-left in practice: an industrial case of application”. In: *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment: First International Workshop, DEVOPS 2018, Chateau de Villebrumier, France, March 5-6, 2018, Revised Selected Papers 1*. Springer. 2019, pp. 205–220.

[83] R. Kazman, S.G. Woods, and S.J. Carriere. “Requirements for integrating software architecture and reengineering models: CORUM II”. In: *Proceedings Fifth Working Conference on Reverse Engineering (Cat. No.98TB100261)*. Proceedings Fifth Working Conference on Reverse Engineering (Cat. No.98TB100261). Oct. 1998, pp. 154–163. DOI: 10.1109/WCRE.1998.723185. URL: https://ieeexplore.ieee.org/abstract/document/723185 (visited on 02/02/2024).

[84] Abdelmadjid Ketfi and Noureddine Belkhatir. “Model-driven framework for dynamic deployment and reconfiguration of component-based software systems”. In: *Proceedings of the 2005 symposia on Metainformatics - MIS ’05*. the 2005 symposia. Esbjerg, Denmark: ACM Press, 2005, 8–es. ISBN: 978-1-59593-719-3. DOI: 10.1145/1234324.1234332. URL: http://portal.acm.org/citation.cfm?doid=1234324.1234332 (visited on 01/04/2024).

[85] Jörg Christian Kirchhof et al. “MontiThings: Model-Driven Development and Deployment of Reliable IoT Applications”. In: *Journal of Systems and Software* 183 (Jan. 2022), p. 111087. ISSN: 01641212. DOI: 10.1016/j.jss.2021.111087. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121221001849 (visited on 12/07/2023).

[86] *Kubernetes*. URL: https://kubernetes.io/ (visited on 06/21/2024).

[87] Rakesh Kumar and Rinkaj Goyal. “Modeling continuous security: A conceptual model for automated DevSecOps using open-source software over cloud (ADOC)”. In: *Computers & Security* 97 (Oct. 1, 2020), p. 101967. ISSN: 0167-4048. DOI: 10.1016/j.cose.2020.101967. URL: https://www.sciencedirect.com/science/article/pii/S0167404820302406 (visited on 01/03/2024).

[88] GitHub Staff Kyle Daigle. *Octoverse: The State of Open Source and rise of AI in 2023*. Nov. 2023. URL: https://github.blog/2023-11-08-the-state-of-open-source-and-ai/.

[89] Hemank Lamba et al. “Heard it through the Gitvine: an empirical study of tool diffusion across the npm ecosystem”. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE ’20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Virtual Event USA: ACM, Nov. 8, 2020, pp. 505–517. ISBN: 978-1-4503-7043-1. DOI: 10.1145/3368089.3409705. URL: https://dl.acm.org/doi/10.1145/3368089.3409705 (visited on 02/01/2024).

[90] Pei Liu et al. “A First Look at CI/CD Adoptions in Open-Source Android Apps”. In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’22. Rochester, MI, USA: Association for Computing Machinery, 2023. ISBN: 9781450394758. DOI: 10.1145/3551349.3561341. URL: https://doi.org/10.1145/3551349.3561341.

[91] Leszek A. Maciaszek and Joaquim Filipe, eds. *Evaluation of Novel Approaches to Software Engineering: 9th International Conference, ENASE 2014, Lisbon, Portugal, April*

28-30, 2014. Revised Selected Papers. Vol. 551. Communications in Computer and Information Science. Cham: Springer International Publishing, 2015. ISBN: 978-3-319-27217-7. DOI: 10.1007/978-3-319-27218-4. URL: http://link.springer.com/10.1007/978-3-319-27218-4 (visited on 12/09/2023).

[92] Manar Majthoub, Mahmoud H. Qutqut, and Yousra Odeh. "Software Re-engineering: An Overview". In: *2018 8th International Conference on Computer Science and Information Technology (CSIT)*. 2018 8th International Conference on Computer Science and Information Technology (CSIT). Amman: IEEE, July 2018, pp. 266–270. ISBN: 978-1-5386-4152-1. DOI: 10.1109/CSIT.2018.8486173. URL: https://ieeexplore.ieee.org/document/8486173/ (visited on 12/26/2023).

[93] *Managing Plugins*. Managing Plugins. URL: https://www.jenkins.io/doc/book/managing/plugins/ (visited on 12/17/2023).

[94] *Manually migrating to GitHub Actions*. GitHub Docs. URL: https://docs.github.com/en/actions/migrating-to-github-actions/automated-migrations (visited on 12/06/2023).

[95] Fran Melchor et al. "A Model-Driven Approach for Systematic Reproducibility and Replicability of Data Science Projects". In: *Advanced Information Systems Engineering*. Ed. by Xavier Franch et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 147–163. ISBN: 978-3-031-07472-1. DOI: 10.1007/978-3-031-07472-1_9.

[96] Tom Mens and Pieter Van Gorp. "A Taxonomy of Model Transformation". In: *Electronic Notes in Theoretical Computer Science* 152 (Mar. 2006), pp. 125–142. ISSN: 15710661. DOI: 10.1016/j.entcs.2005.10.021. URL: https://linkinghub.elsevier.com/retrieve/pii/S1571066106001435 (visited on 12/20/2023).

[97] *metaDepth: A framework for deep meta-modelling*. URL: http://metadepth.org/ (visited on 12/27/2023).

[98] *MetaObject Facility | Object Management Group*. URL: http://www.omg.org/mof/ (visited on 12/27/2023).

[99] Bart Meyers et al. "A Model-Driven Engineering Framework to Support the Functional Safety Process". In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). Sept. 2019, pp. 619–623. DOI: 10.1109/MODELS-C.2019.00094. URL: https://ieeexplore.ieee.org/abstract/document/8904799 (visited on 01/03/2024).

[100] *Migrate from GitHub Actions - CircleCI*. URL: https://circleci.com/docs/migrating-from-github/ (visited on 12/27/2023).

[101] *Model Merging (EML) - Epsilon*. URL: https://eclipse.dev/epsilon/doc/eml/ (visited on 06/01/2024).

[102] *Model Transformation (ETL) - Epsilon*. URL: https://eclipse.dev/epsilon/doc/etl/ (visited on 06/01/2024).

[103] James Newkirk. "Introduction to agile processes and extreme programming". In: *Proceedings of the 24th international conference on Software engineering*. 2002, pp. 695–696.

[104] OASIS. *Topology and orchestration specification for cloud applications (TOSCA) Version 1.0, Committee Specification 01*. URL: http://docs.oasisopen.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html. (visited on 01/04/2024).

[105] Obeo. *ATL | The Eclipse Foundation*. URL: https://eclipse.dev/atl/ (visited on 12/27/2023).

[106] Object Management Group. *OMG Unified Modeling Language Specification*. Mar. 2000.

[107] Efi Papatheocharous and Andreas S. Andreou. "Evidence of Agile Adoption in Software Organizations: An Empirical Survey". In: *Systems, Software and Services Process Improvement*. Ed. by Fergal McCaffery, Rory V. O'Connor, and Richard Messnarz. Vol. 364. Series Title: Communications in Computer and Information Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 237–246. ISBN: 978-3-642-39178-1. DOI: 10.1007/978-3-642-39179-8_21. URL: http://link.springer.com/10.1007/978-3-642-39179-8_21 (visited on 12/09/2023).

[108] R. Pérez-Castillo et al. "Reengineering technologies". In: *IEEE Software* 28.6 (2011), pp. 13–17. ISSN: 0740-7459. DOI: 10.1109/MS.2011.145.

[109] Puppet by Perforce. accessed: 16/11/2023. 2023. URL: https://www.puppet.com/resources/history-of-devops-reports.

[110] Puppet by Perforce. *State of Devops Report 2013*. 2013.

[111] Puppet by Perforce. *State of Devops Report 2015*. 2015.

[112] Puppet by Perforce. *State of Devops Report 2017*. 2017.

[113] Kai Petersen, Claes Wohlin, and Dejan Baca. "The Waterfall Model in Large-Scale Development". In: *Product-Focused Software Process Improvement*. Ed. by Frank Bomarius et al. Vol. 32. Series Title: Lecture Notes in Business Information Processing. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 386–400. ISBN: 978-3-642-02151-0. DOI: 10.1007/978-3-642-02152-7_29. URL: http://link.springer.com/10.1007/978-3-642-02152-7_29 (visited on 12/09/2023).

[114] *Plan a migration from another tool to GitLab CI/CD | GitLab*. URL: https://docs.gitlab.com/ee/ci/migration/plan_a_migration.html (visited on 12/06/2023).

[115] 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*. MODELS '22. Montreal, Quebec, Canada: Association for Computing Machinery, 2022, pp. 225–228. ISBN: 9781450394673. DOI: 10.1145/3550356.3552395. URL: https://doi.org/10.1145/3550356.3552395.

[116] *Puppet Infrastructure & IT Automation at Scale | Puppet by Perforce*. URL: https://www.puppet.com/ (visited on 01/04/2024).

[117] Thijs Reus, Hans Geers, and Arie van Deursen. "Harvesting Software Systems for MDA-Based Reengineering". In: *Model Driven Architecture – Foundations and Applications*. Ed. by Arend Rensink and Jos Warmer. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 213–225. ISBN: 978-3-540-35910-4. DOI: 10.1007/11787044_17.

[118] Franklin Magalhães Ribeiro et al. "A Model-Driven Solution for Automatic Software Deployment in the Cloud". In: *Information Technology: New Generations*. Ed. by Shahram Latifi. Advances in Intelligent Systems and Computing. Cham: Springer International Publishing, 2016, pp. 591–601. ISBN: 978-3-319-32467-8. DOI: 10.1007/978-3-319-32467-8_52.

[119] Luis F. Rivera et al. "UML-driven automated software deployment". In: *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*. CASCON '18. USA: IBM Corp., Oct. 29, 2018, pp. 257–268. (Visited on 01/03/2024).

[120] D. L. H. Rosenberg. *Software Re-engineering*. Goddard Space Flight Center, NASA.

[121] Pooya Rostami Mazrae et al. "On the usage, co-usage and migration of CI/CD tools: A qualitative analysis". In: *Empirical Software Engineering* 28.2 (Mar. 7, 2023), p. 52. DOI: 10.1007/s10664-022-10285-5. URL: https://doi.org/10.1007/s10664-022-10285-5.

[122] Winston W Royce. "Managing the development of large software systems: concepts and techniques". In: *Proceedings of the 9th international conference on Software Engineering*. 1987, pp. 328–338.

[123] Julio Sandobalin. "A Model-Driven Approach to Continuous Delivery of Cloud Resources". In: *Service-Oriented Computing – ICSOC 2017 Workshops*. Ed. by Lars Braubach et al. Vol. 10797. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 346–351. ISBN: 978-3-319-91763-4. DOI: 10.1007/978-3-319-91764-1_29. URL: https://link.springer.com/10.1007/978-3-319-91764-1_29 (visited on 12/07/2023).

[124] *Search*. GitHub Docs. URL: https://docs.github.com/en/rest/search?apiVersion=2022-11-28 (visited on 01/05/2024).

[125] S. Sendall and W. Kozaczynski. “Model transformation: the heart and soul of model-driven software development”. In: *IEEE Software* 20.5 (Sept. 2003), pp. 42–45. ISSN: 0740-7459. DOI: 10.1109/MS.2003.1231150. URL: http://ieeexplore.ieee.org/document/1231150/ (visited on 12/20/2023).

[126] *Sirius - Eclipsepedia*. URL: https://wiki.eclipse.org/Sirius (visited on 12/27/2023).

[127] Eliezio Soares et al. “The effects of continuous integration on software development: a systematic literature review”. In: *Empirical Software Engineering* 27.3 (May 2022), p. 78. ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-021-10114-1. URL: https://link.springer.com/10.1007/s10664-021-10114-1 (visited on 11/01/2023).

[128] Ian Sommerville. *Software engineering*. 9th ed. OCLC: ocn462909026. Boston: Pearson, 2011. 773 pp. ISBN: 978-0-13-703515-1.

[129] Hui Song et al. “Model-based fleet deployment in the IoT–edge–cloud continuum”. In: *Software and Systems Modeling* 21.5 (Oct. 2022), pp. 1931–1956. ISSN: 1619-1366, 1619-1374. DOI: 10.1007/s10270-022-01006-z. URL: https://link.springer.com/10.1007/s10270-022-01006-z (visited on 12/07/2023).

[130] *Stack Overflow Insights - Developer Hiring, Marketing, and User Research*. URL: https://survey.stackoverflow.co/ (visited on 02/05/2024).

[131] Daniel Ståhl and Jan Bosch. “Modeling continuous integration practice differences in industry software development”. In: *Journal of Systems and Software* 87 (Jan. 2014), pp. 48–59. ISSN: 01641212. DOI: 10.1016/j.jss.2013.08.032. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121213002276 (visited on 11/29/2023).

[132] *State of Continuous Delivery Report 2023: The Evolution of Software Delivery Performance*. CD Foundation. URL: https://cd.foundation/state-of-cd-2023/ (visited on 02/05/2024).

[133] Case Study. *Cloud native computing foundation*. Nov. 2023. URL: https://www.cncf.io/.

[134] *TCS - Eclipsepedia*. URL: https://wiki.eclipse.org/TCS (visited on 12/27/2023).

[135] *The State of Developer Ecosystem in 2023*. JetBrains: Developer Tools for Professionals and Teams. URL: https://www.jetbrains.com/lp/devecosystem-2023 (visited on 02/01/2024).

[136] *Travis CI*. URL: https://www.travis-ci.com (visited on 06/21/2024).

[137] William M. Ulrich and Philip Newcomb. *Information Systems Transformation: Architecture-Driven Modernization Case Studies*. Google-Books-ID: hDzIedYPG7AC. Morgan Kaufmann, Feb. 4, 2010. 449 pp. ISBN: 978-0-08-095710-4.

[138] *Using scripts to test your code on a runner*. GitHub Docs. URL: https://docs.github.com/en/actions/examples/using-scripts-to-test-your-code-on-a-runner (visited on 12/06/2023).

[139] Bogdan Vasilescu et al. "Quality and productivity outcomes relating to continuous integration in GitHub". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE'15: Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. Bergamo Italy: ACM, Aug. 30, 2015, pp. 805–816. ISBN: 978-1-4503-3675-8. DOI: 10.1145/2786805.2786850. URL: https://dl.acm.org/doi/10.1145/2786805.2786850 (visited on 12/17/2023).

[140] Eelco Visser. "A Survey of Rewriting Strategies in Program Transformation Systems". In: *Electronic Notes in Theoretical Computer Science*. WRS 2001, 1st International Workshop on Reduction Strategies in Rewriting and Programming 57 (Dec. 1, 2001), pp. 109–143. ISSN: 1571-0661. DOI: 10.1016/S1571-0661(04)00270-1. URL: https://www.sciencedirect.com/science/article/pii/S1571066104002701 (visited on 12/20/2023).

[141] Eclipse Web. *Eclipse OCL (Object Constraint Language)*. projects.eclipse.org. Jan. 31, 2013. URL: https://projects.eclipse.org/projects/modeling.mdt.ocl (visited on 12/27/2023).

[142] Eclipse Web. *Eclipse Xpand*. projects.eclipse.org. Jan. 31, 2013. URL: https://projects.eclipse.org/projects/modeling.m2t.xpand (visited on 12/27/2023).

[143] Eclipse Web. *Java Emitter Templates (JET2)*. projects.eclipse.org. Jan. 31, 2013. URL: https://projects.eclipse.org/projects/modeling.m2t.jet (visited on 12/27/2023).

[144] Denis Weerasiri et al. "A Model-Driven Framework for Interoperable Cloud Resources Management". In: *Service-Oriented Computing*. Ed. by Quan Z. Sheng et al. Vol. 9936. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 186–201. ISBN: 978-3-319-46294-3. DOI: 10.1007/978-3-319-46295-0_12. URL: https://link.springer.com/10.1007/978-3-319-46295-0_12 (visited on 06/25/2024).

[145] Denis Weerasiri et al. "CloudMap: A Visual Notation for Representing and Managing Cloud Resources". In: *Advanced Information Systems Engineering*. Ed. by Selmin Nurcan et al. Vol. 9694. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 427–443. ISBN: 978-3-319-39695-8. DOI: 10.1007/978-3-319-39696-5_26. URL: https://link.springer.com/10.1007/978-3-319-39696-5_26 (visited on 06/25/2024).

[146] Frank Weil and LLC UniqueSoft. "Legacy Software Reengineering". In: *Unique Soft LLC* (2015).

[147] Michael Wenz. *Graphiti Home | The Eclipse Foundation*. URL: https://eclipse.dev/graphiti/ (visited on 12/27/2023).

[148] Johannes Wettinger et al. "Streamlining DevOps automation for Cloud applications using TOSCA as standardized metamodel". In: *Future Generation Computer Systems* 56 (Mar. 2016), pp. 317–332. ISSN: 0167739X. DOI: 10.1016/j.future.2015.07.017. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167739X15002496 (visited on 01/04/2024).

[149] David Gray Widder et al. "A conceptual replication of continuous integration pain points in the context of Travis CI". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE '19: 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Tallinn Estonia: ACM, Aug. 12, 2019, pp. 647–658. ISBN: 978-1-4503-5572-8. DOI: 10.1145/3338906.3338922. URL: https://dl.acm.org/doi/10.1145/3338906.3338922 (visited on 01/31/2024).

[150] David Gray Widder et al. "I'm leaving you, Travis: a continuous integration breakup story". In: *Proceedings of the 15th International Conference on Mining Software Repositories*. ICSE '18: 40th International Conference on Software Engineering. Gothenburg Sweden: ACM, May 28, 2018, pp. 165–169. ISBN: 978-1-4503-5716-6. DOI: 10.1145/3196398.3196422. URL: https://dl.acm.org/doi/10.1145/3196398.3196422 (visited on 12/17/2023).

[151] N. Wirth. *Algorithms + Data Structures = Programs*. Series In Automatic Computation. Prentice-Hall, 1976. URL: https://books.google.pt/books?id=XRhOxgEACAAJ.

[152] Emma Witman. *What is GitHub? How to start using the code hosting platform that allows you to easily manage and collaborate on programming projects*. Business Insider. URL: https://www.businessinsider.com/guides/tech/what-is-github (visited on 01/05/2024).

[153] Michael Wurster et al. "The essential deployment metamodel: a systematic review of deployment automation technologies". In: *SICS Software-Intensive Cyber-Physical Systems* 35.1 (Aug. 2020), pp. 63–75. ISSN: 2524-8510, 2524-8529. DOI: 10.1007/s00450-019-00412-x. URL: http://link.springer.com/10.1007/s00450-019-00412-x (visited on 01/04/2024).

[154] *Xtext - Language Engineering Made Easy!* URL: https://eclipse.dev/Xtext/ (visited on 12/27/2023).

[155] Tianyin Xu and Darko Marinov. "Mining Container Image Repositories for Software Configuration and Beyond". In: *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*. ICSE-NIER '18. Gothenburg, Sweden:

Association for Computing Machinery, 2018, pp. 49–52. ISBN: 9781450356626. DOI: 10.1145/3183399.3183403. URL: https://doi.org/10.1145/3183399.3183403.

[156] Mansooreh Zahedi, Roshan Namal Rajapakse, and Muhammad Ali Babar. "Mining Questions Asked about Continuous Software Engineering: A Case Study of Stack Overflow". In: *Proceedings of the Evaluation and Assessment in Software Engineering*. EASE '20. Trondheim, Norway: Association for Computing Machinery, 2020, pp. 41–50. ISBN: 9781450377317. DOI: 10.1145/3383219.3383224. URL: https://doi.org/10.1145/3383219.3383224.

[157] Liming Zhu, Len Bass, and George Champlin-Scharff. "DevOps and Its Practices". In: *IEEE Software* 33.3 (May 2016), pp. 32–34. ISSN: 0740-7459. DOI: 10.1109/MS.2016.81. URL: http://ieeexplore.ieee.org/document/7458765/ (visited on 12/07/2023).

Appendix A

Platform-Specific Meta-Model Figures

Figures A.1 to A.3 represent the PSMMs developed for this work. Expression and variable declaration classes have been truncated to improve legibility.

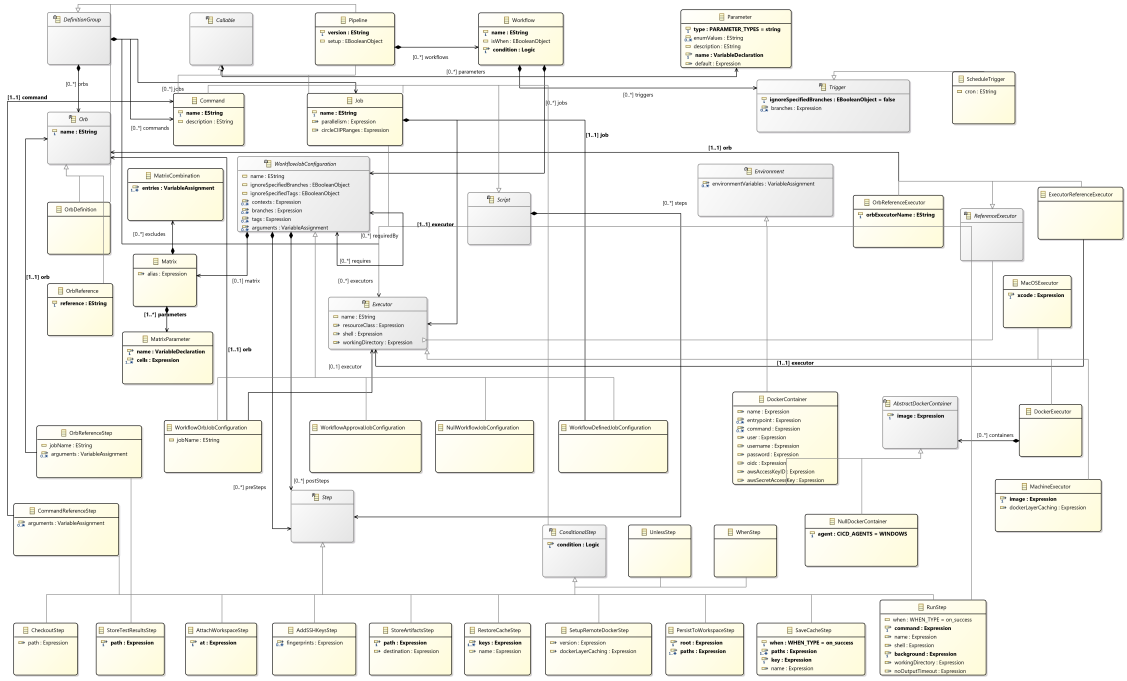


Figure A.1: CircleCI meta-model.

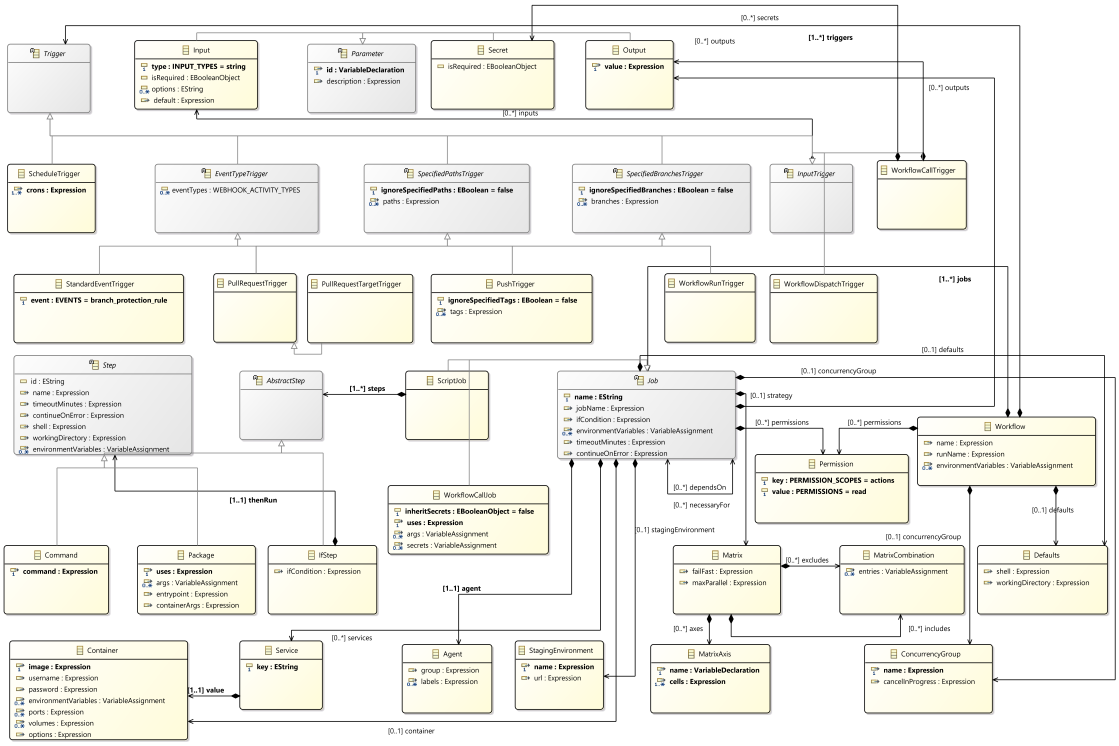


Figure A.2: GHA meta-model.

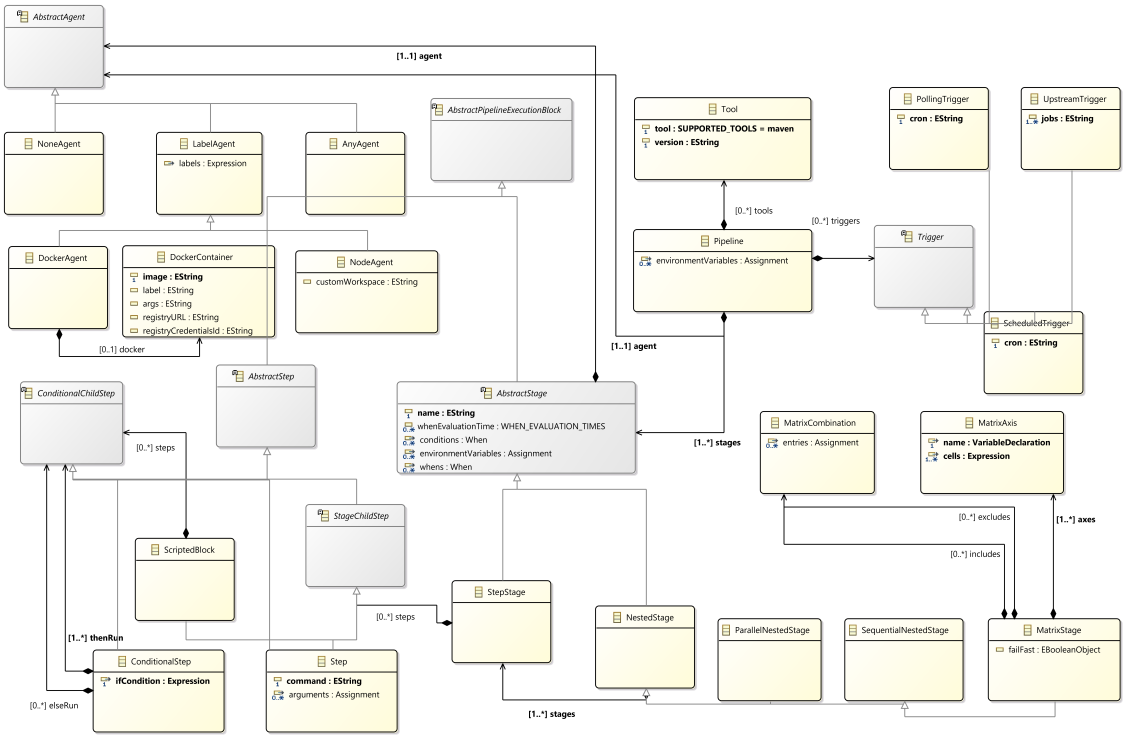


Figure A.3: Jenkins meta-model.