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

André Flores[1], Hugo Gião[1], Vasco Amaral[2], and Jácome Cunha[1]

[1] HASLab/ INESC TEC, Faculty of Engineering, University of Porto
up201907001@edu.fe.up.pt hugo.a.giao@inesctec.pt jacome@fe.up.pt
[2] NOVA School of Science and Technology & NOVA LINCS
vma@fct.unl.pt

Abstract. In the realm of industrial software development, Continuous Integration, Continuous Delivery, and Continuous Deployment (CI/CD) has emerged as the preferred approach for handling the highly iterative software production process. However, CI/CD pipelines must constantly be migrated to new versions or new platforms. This manual, cumbersome, and error-prone activity requires systematic and automated support.
To address this issue, we propose a novel approach that leverages model-driven engineering (MDE) to support the reengineering of CI/CD pipelines. The approach we propose is inspired by the traditional reengineering horseshoe model. However, ours relies on two levels of intermediate CI/CD pipeline meta-models. We support the abstraction of existing pipelines into models conforming to our meta-models, from which semantic-equivalent pipelines can be generated to (possibly) other CI/CD platforms, thus providing a full engineering path.
Our contribution comprises a platform-independent meta-model designed to represent the structure of existing CI/CD pipelines, three platform-specific meta-models, a migration CLI to transform them into a new target format, and a DSL for users to interact with the process.
We show that, after reengineering a pipeline, the execution of both is equivalent. We also evaluate to which extent it would be possible to fully automate CI/CD reengineering using our approach.

Keywords: Model-Driven Engineering · Reengineering · Reverse Engineering · Continuous Integration · Continuous Delivery · Continuous Deployment · DevOps · CI/CD

1 Introduction

Continuous Integration, Delivery, and Deployment, known as 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. In recent years, CI/CD has become crucial for organizations to meet market demands by enabling rapid and frequent changes to their projects. CI/CD pipelines automate the integration, testing, and deployment of code changes, ensuring frequent and reliable software releases [28].

In implementing CI/CD, organizations can make use of several platforms like GitHub Actions (GHA) [18], GitLab CI/CD [20], Travis CI [33], CircleCI [4] or Jenkins [25], just to mention a few of the dozens that exist. Often, organizations will use several platforms for the same project [22, 32].

Over time, the rate of changes to the CI/CD platforms used by projects has been increasing, peaking in 2021 when over 15% of all commits to code repositories included changes to CI/CD platforms [22]. This underscores the need for migration between different CI/CD platforms. Therefore, efficiently migrating between these diverse platforms is essential for maintaining the agility and effectiveness of software development.

However, there is almost no support for migrating CI/CD platforms. Some platform providers have migration guides [6–8, 19, 21], but these are mostly basic syntax comparisons. GitHub provides a tool that aims at migrating about 80% of scripts from (only seven) other platforms [17], but it only migrates to theirs.

Our objective is to streamline the migration and evolution process by developing a meta-model capable of representing diverse CI/CD pipelines in a platform-independent manner. Unlike other CI/CD meta-models, ours draws inspiration from several of the most widely used CI/CD platforms and is designed to abstract concepts from these platforms with low-level detail. This is because the meta-model serves as an intermediate representation (IR) for a CI/CD pipeline transpiler through a reengineering process. Using model merging, we can also merge several pipelines from different platforms into a single one.

With our work, we seek to 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 3.

RQ2: Can a platform-independent meta-model (PIMM) be the basis for 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 (PIM). Afterward, we should be able to generate a CI/CD pipeline in a possibly different platform from the original one. Using the model transformations defined in Sections 4 and 5 and the tool introduced in Section 6, we answer RQ2 in Section 7.

RQ3: To which extent 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 7 provides an answer to this RQ.

In the following section, we present an overview of our approach.

2 An Overview of Reengineering CI/CD Pipelines

Our goal is to create a transpiler for CI/CD pipelines. For a pipeline written in platform A's domain-specific language (DSL), we seek to devise a program that outputs a pipeline in the correct syntax for platform B's DSL and semantically equivalent to the original pipeline.

To this end, this work intends to leverage model-driven engineering by creating a platform-independent meta-model (PIMM) that defines a modeling language for CI/CD pipelines. This process is detailed in Section 3. The PIMM allows automatic CI/CD migration through a reengineering process, shown in Figure 1 as a horseshoe model [27].

The tool's logic will be implemented using model transformations following the migration methods specified by Grieger et al. [24]. Text-to-model (T2M) transformations (**I**) will convert an input pipeline file (**1**) to a platform-specific model (PSM) (**2**) that conforms to one of the platform-specific meta-models we designed (**3**), as explained in Section 4. That PSM will be transformed into a platform-independent model (PIM) (**4**) that conforms to the PIMM (**5**) through model-to-model (M2M) transformations (**II**). We will then transform the PIM to a PSM for a different pipeline platform (**6**), again through M2M transformations. The M2M transformations are introduced in Section 5. The translated configuration file (**7**) will be generated from the new PSM through model-to-text (M2T) transformations (**III**). Due to the lack of space and their simplicity, M2T transformations are not further described. The user can interact with this process with a transformations DSL (TDSL) to perform M2M transformations on the PIM and/or PSMs (**IV**). This DSL allows the user to replace platform-specific plugins, which we do not migrate automatically, and change where pipeline jobs are executed, among other particularities of CI/CD languages. Due to space limitations, we do not further detail TDSL. A complete description of the M2T transformations and TDSL can be found at [15].

Using two modeling levels, platform-specific and platform-independent, modularizes the reengineering process. If we had used just one meta-model, T2M, and

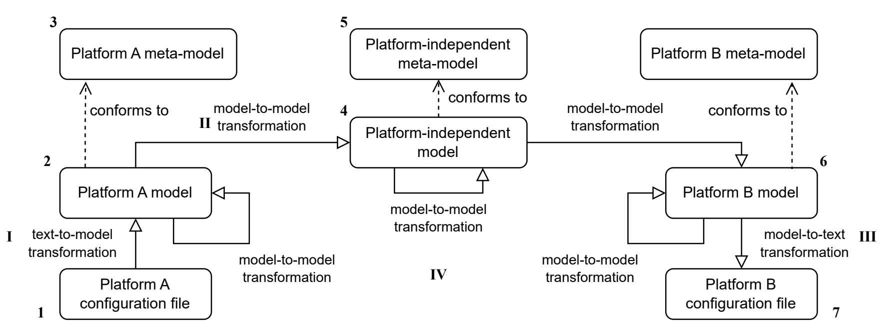

Fig. 1. CI/CD pipeline reengineering process.

M2T transformations would have two responsibilities: *i)* to convert a pipeline from textual representation to a model and *ii)* to handle any differences between the pipeline's platform and the PIMM. This would add complexity to the transformations and make the transpiler harder to develop.

3 The Meta-Models

We started development by researching three CI/CD platforms that represented the current CI/CD landscape [22, 23, 26]. These are GHA, CircleCI, and Jenkins. We used these platforms' configuration references as the basis for their PSMMs. The references allowed us to determine the features of each technology, as well as its valid configurations. With this, we could create a basic PSMM. Afterward, we searched for commonalities between PSMM classes to establish inheritance relationships, thus reducing redundancy. The PSMMs can be seen in [15].

We then devised the PIMM. Designing the PIMM was mostly a matter of finding common functionality between the PSMMs. For an example of the class mappings between PIMM and PSMMs, see Appendix A. Domain knowledge was also important to determine the core functionalities of CI/CD pipelines.

For each PIMM class, we must determine intrinsic and extrinsic properties and the arity of each property [3]. If this is done incorrectly, it could lead to a meta-model that is hard to work on and evolve or one that is not an accurate abstraction of the CI/CD platform's pipeline.

Figure 2 shows a truncated version of the PIMM (without **Expression** and **VariableDeclaration** classes and enumerations), which we now further detail.

The **Pipeline** is the main class of the model. It represents each CI/CD script and refers to all other properties.

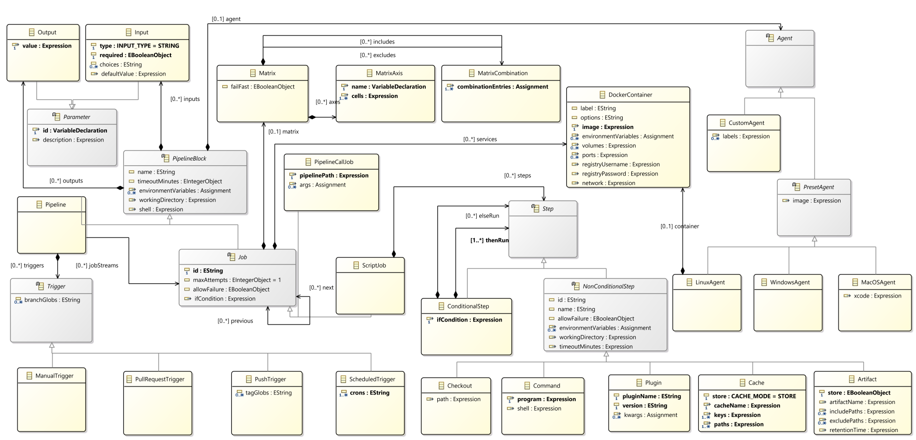

Fig. 2. The platform-independent meta-model.

A **Job** is a set of instructions that are executed as a single block of the pipeline. It can be composed of various steps, a **ScriptJob**, or be a call to a separate pipeline script, a **PipelineCallJob**.

Jobs are run in parallel by default. The *previous* and *next* properties are used to indicate dependencies between **Jobs**. CI/CD platforms differ in their parallelization implementations, which are handled by M2M transformations.

Matrices are an extrinsic property of **Jobs** used to define combinations of values the to run the **Job** with. For example, in a **Matrix**, a user can define various values for an operating system (OS) and a program they would like to run; the **Job** would then be run for every combination of OS and program.

An **Agent** specifies where a **Pipeline** or **Job** will be run. This is done through various subclasses. An **Agent** can also have a **DockerContainer**, which holds the information necessary to configure a Docker container.

A **Service** is a **DockerContainer** that runs in the background while a **Job** is executing and can be accessed by it.

Trigger defines events that start the execution of a **Pipeline**. All platforms have a **Trigger**, but they do not all handle all their configuration in the **Pipeline**. Instead, they leave this to platform settings. Despite this, several **Trigger** subclasses are still included in the TIM due to their relevance.

Parameters are used to specify inputs and outputs for **Pipelines** and **Jobs**.

Steps are atomic instructions that run as part of a **Job**. There are various kinds of **Steps**. **Commands** run a specified program in the **Job**'s **Agent**. **Plugins** run platform-specific packages made available in marketplaces. **Cache** steps can either load or store data from the **Agent** to speed up subsequent **Pipeline** executions. **Artifacts** can store output data specific to a certain **Pipeline** execution. **Checkout** serves to load the git repository into the **Agent**. **ConditionalSteps** are used for flow control. They have **Steps** that are executed when the condition is true and can have **Steps** that are executed otherwise.

Expressions include logical operators, assignments, literals, variable references, and formatted strings (that mix string literals and other expressions). These features are common to all studied CI/CD platforms. **Expressions** are used throughout the PIMM. **VariableDeclarations** are a property of **Assignments**, **Parameters** and **Matrices**.

Addressing RQ1 The PIMM represents the core concepts of CI/CD pipelines. This is because the platforms we based ourselves on make up the majority of CI/CD usage [22, 23, 26]. These platforms' PSMMs also have significant differences, yet the PIMM can merge all of their core functionality.

4 Text-to-Model Transformations

After defining the PSMMs, to fully support a migration process, we need to be able to parse CI/CD pipelines from their usual text form into models conforming to their platform's meta-model. Thus, we defined T2M transformations to parse CI/CD code into a model.

The original approach we considered was to perform T2M transformations by generating a parser with Xtext [14] to perform embedded translation, i.e., directly outputting the PSM from the parser. However, this could not be done as the DSLs being parsed have some complexity regarding 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. Firstly, we parse the YAML pipeline and create an abstract syntax tree (AST). Then, we visit the tree nodes and populate the model.

Instead of creating a model and parser for YAML ourselves, we elected to use an existing Java package to parse scripts. Certain platforms, like GHA, extend the YAML syntax with their proprietary expressions syntax. In this case, we created a parser for these expressions based on an Xtext expressions grammar, which we use in conjunction with the base YAML parser. Walking the AST is then done programmatically using Java.

5 Model-to-Model Transformations

Model-to-model transformations presented the main challenge of our approach, as they are the logic that allows translating pipelines between platforms.

Transformations between models conforming to different meta-models only happen from a PSM to a PIM or from a PIM to a PSM. This is because we want the PIM to be an intermediate representation (IR) in the reengineering process. The rules for these transformations are implemented in ATL [12].

The PIMM was designed to ease translations to and from the PSMMs. Where possible, we wanted one-to-one mappings between PIM and PSM, i.e., when a concept or property in one meta-model has a direct correspondence to another concept or property in the other meta-model.

However, this cannot be done for all cases due to significant differences between platforms. For instance, Jenkins's way of executing **Jobs** in parallel by nesting them means the PIM-to-Jenkins transformation has to group the job dependency graph into levels. Since CircleCI splits a **Job** into its definition and its calls, transformations with the PIM have to map one class to two. Most differences between models can be handled entirely by PSM-to-PIM/PIM-to-PSM transformations. However, generating an output model directly would often be overly complex.

To deal with this, we split the transformation from a PSM to a PIM and vice-versa into multiple steps by using helpers. These helpers are either PIM-to-PIM or PSM-to-PSM transformations we define to make simple alterations to the models. For example, 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 child-steps. GHA only allows one child-step per condition, and has no else functionality. To deal with this, we unwind **ConditionalSteps** on the PIM, creating multiple **ConditionalSteps**, each with one child-step. Else blocks are handled by negating the condition.

Besides these transformation helpers, we also use PIM-to-PIM and PSM-to-PSM transformations to implement the TDSL.

6 ACICDTrip – A Tool for CI/CD Reengineering

We designed our tool as a CLI implemented in Java. We focused on creating a CLI because our goal was to create a tool that could be used outside the Eclipse IDE (unlike most MDE software). This was important so that it would be accessible to all users, even if they are not familiar with MDE or MDE tools themselves.

Our transpiler has two modes it can operate on. Normal mode will attempt to translate any input pipeline to the selected CI/CD platform. Due to differences in platform features and current transpiler limitations, this mode does not attempt to guarantee semantic equivalence between the input and output pipelines. It should be seen as a helper to the migration process instead of an attempt to replace it wholly. The second mode, strict mode, runs an OCL [13] validation on the input model to check if it can be transformed to the output platform entirely automatically while keeping pipeline semantics intact (we do not consider alterations to platform-specific **Plugins** that must be made).

All of the transformations and OCL validations are run in standalone mode (outside of Eclipse). Besides T2M transformations, the only logic implemented in the CLI itself is integrating the various MDE technologies used.

The source code for ACICDTrip can be found at [16].

7 Evaluation

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

In Section 7.1, 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 provides us with a perspective on the use of our tool in practice.

In Section 7.2, 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 gives us a perspective of our tool's functionality for a large number of scripts.

7.1 Comparing the Execution of Original and Migrated Pipelines

The Process To compare the execution of pipelines, both original and migrated ones, we needed not only example pipelines to migrate but also the underlying codebase. CircleCI provides a set of five example repositories to introduce users to their platform [5]. The scripts used for these repositories encompass most features integral to a CI/CD pipeline.

ACICDTrip could migrate all of the example scripts to GHA with the help of the TDSL. The TDSL needs to be used to change platform-specific **Plugins** or delete certain steps that only make sense in the CircleCI context, like adding SSH keys so the CircleCI Windows VM can access the repository. We also needed to change some Docker images provided by CircleCI. One of the scripts uses a CircleCI feature where multiple pipelines can be configured in one script. This script can still be translated, but it needs to be done twice, selecting each pipeline (Appendix C shows a TDSL script used for this). It should also be noted that one of the examples had a hardcoded URL that needed to be changed to work in GHA, which was done manually.

The Results After running the CI/CD pipelines, we compared the logs they outputted. Our criteria to determine logs to be equivalent were if the key steps of each pipeline were executed and if their output was the same (Listings 1.1 and 1.2 in Appendix B are an example of a comparison). We determined the original and new platforms' logs to be equivalent in all five examples. Three failed out-of-the-box and three succeeded; when migrated to GHA, they all failed and succeeded in the same way as the originals.

7.2 Double Round-Trip

The Process Several challenges are involved in checking whether we can migrate a migrated pipeline back into the original platform without changes. 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 it, the program is meant to exit with an error if it finds a feature that cannot be migrated.

We randomly selected 10,000 repositories that used GHA from Gião et al.'s dataset of repositories using CI/CD [22]. In these repositories, we found 25,487 GHA scripts. We could migrate 22,684 (89%) 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. The majority of the pipelines that failed strict-mode 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).

After migrating the scripts, we compared the original and the generated GHA scripts using `yamldiff` [29]. We filtered out differences between the scripts that had no semantic impact on the execution of the script in GHA, e.g., GHA lets users forgo array syntax when there is only one element in an array (`on: push` is the same as `on: [push]`) but in this case the code is always generated with array syntax. Appendix D lists the differences we discarded.

The 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), which means 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 the 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 that is not supported in 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 of 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.

Clearly, the main limitation to semantic equivalence is the abstraction of certain **Plugins**. If we used an intermediate platform other than CircleCI for this evaluation, we could circumvent this. However, when migrating, users must always search for new platform-specific **Plugins**, and this is only an issue due to the double round-trip nature of the evaluation.

7.3 Discussion

Addressing RQ2 In Section 7.1, all pipelines could be migrated to GHA. The PIMM supported the pipelines completely, and the transformations we defined could accurately migrate 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 7.2, many pipelines cannot yet be migrated in strict mode. Still, the vast majority (81.1%) of pipelines supported by strict mode can be migrated without semantic alteration.

Thus, the proposed PIMM can support the migration between different CI/CD platforms.

Addressing RQ3 The different CI/CD platforms have many common functionalities. However, there are still significant differences. This means there will always be pipelines that cannot be wholly migrated from one platform to another. Section 7.2 shows a clear example of this. Even in the pipelines supported by strict mode, 404 had **Plugins** lose arguments because of extra functionality.

Moreover, migrating CI/CD sometimes requires changes to the codebase and changes that can only be done with context-specific knowledge. Section 7.1 has an example of this. Migrating meant changing the address and ports of a Docker container, which can only be done with the 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. Theoretically, this could be done automatically, but there is no guarantee that another platform will always have a corresponding **Plugin**.

Thus, based on these facts, it seems that a fully automatic reengineering process is not possible.

7.4 Threats to Validity

We now discuss some threats to the validity of our results.

The PIMM was based on several of the most popular CI/CD platforms [22] and, as such, should be able to represent most pipelines. Still, some platforms may not be representable.

We have used ACICDTrip to migrate several CircleCI scripts to GHA, achieving good results. We have chosen CircleCI because they provide example scripts and the corresponding code, and GHA as this is currently the most popular platform. Although nothing in these platforms would make the migration work better with our approach, we cannot make any claims about the generalization of these results.

We show that our tool can impose consistency in the transformations it makes from platform to platform. However, in several situations, some transformations are required (using the TDSL we provide). During our evaluation, we have implemented these so it could be possible to achieve the migrations. The transformations are as direct as possible and should not influence the results.

8 Related Work

Colantoni et al. [10] introduce an innovative approach for modeling DevOps Processes and Platforms, presenting a platform meta-model and a linking meta-model designed to connect various platforms and elucidate the DevOps process. It is adept at DevOps while ensuring compatibility and fulfillment of requirements among different libraries and platforms. In contrast, our objective extends beyond verifying pipeline correctness; we aim for a meta-model that can act as the foundation for a reengineering process. Furthermore, our proposal undergoes a comprehensive validation process incorporating real-world configuration files to substantiate its correctness.

Colantoni et al. [9] present an ongoing project centered on the integrated modeling and scenario simulation of continuous delivery pipelines. Users can define CI/CD processes using a JSON-based domain-specific language (DSL), enabling the semi-automated generation of fully functional executable DSLs and tool support through JSON schema documents. The tool provides graphical and textual ways of interacting with models. Our focus is not solely on generating configuration pipelines, as we intend to fully support the reengineering process. Colantoni et al.'s approach also does not allow for modeling CI/CD in a platform-independent manner.

Rivera et al. [31] tackle the challenges associated with deployment in continuous delivery and DevOps. They introduce a mechanism designed to automate the deployment process by using UML to specify software architecture and deployment. Executable deployment specifications are then generated from these deployment diagrams. In contrast, our approach uses a meta-model compatible with most existing CI/CD platforms and capable of representing a wide range of existing pipelines, a goal not explicitly addressed by the authors. Additionally, while the authors evaluated their approach regarding usability through case studies, our approach was assessed using real-world pipelines.

Bordeleu et al.'s [2] primary objective is to contribute to developing a comprehensive DevOps engineering framework comprising processes, methods, and tools. The authors delve into various aspects of the DevOps system at Kaloom, an industry partner. They outline a set of requirements for establishing a DevOps modeling framework. The aim is to be the foundation for analyzing, simulating, and automating the DevOps process. Our objective extends beyond collecting requirements for modeling CI/CD scripts; we aim to provide a concrete solution. Our solution prioritizes representing a diverse range of pipelines from existing tools, focusing on comprehensiveness rather than usability.

Düllmann et al. [11] 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 meta-model is not specific to a CI/CD language and was designed to abstract away from the intricacies of individual platforms. Furthermore, we tested our meta-model for its ability to represent CI/CD pipelines and for tasks extending beyond mere representation, such as reengineering pipelines across platforms.

Pulgar et al. [30] 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 meta-model 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.

Babar et al. [1] develop a model for DevOps deployment choices, aiding enterprises in tailoring a suitable DevOps approach to meet their requirements. As part of their study, the authors utilize business process analysis (BPA) to model a standard DevOps process. On the other hand, our work diverges from that of Babar et al. in several aspects. Firstly, we introduce a meta-model specifically crafted to represent CI/CD pipelines. Additionally, we provide a more extensive validation process for our meta-model. Furthermore, our objective is to leverage the meta-model for the reengineering and development of CI/CD pipelines.

Wurster et al. [34] propose a meta-model to enable a common understanding of declarative deployment models. Wurster et al.'s approach is a meta-model of software deployment and it helps users select the best deployment technology for their scenario. Our objective extends beyond tool selection, encompassing practical applications such as facilitating development and migration processes.

9 Conclusions

With our work, we found there are enough core concepts common to diverse CI/CD platforms to allow a definition of a common language, the meta-model we propose, allowing, in many situations, the full migration of existing pipelines. 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 **Trigger**s. 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.

There is 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 revealed itself to be a major current limitation.

References

1. Babar, Z., Lapouchnian, A., Yu, E.: Modeling DevOps deployment choices using process architecture design dimensions. In: Ralyté, J., España, S., Pastor, O. (eds.)

The Practice of Enterprise Modeling. pp. 322–337. Lecture Notes in Business Information Processing, Springer International Publishing, New York, New York, USA (2015). https://doi.org/10.1007/978-3-319-25897-3_21
2. Bordeleau, F., Cabot, J., Dingel, J., Rabil, B.S., Renaud, P.: Towards modeling framework for devops: Requirements derived from industry use case. In: Bruel, J.M., Mazzara, M., Meyer, B. (eds.) Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment. pp. 139–151. Springer International Publishing, Cham (2020)
3. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Synthesis Lectures on Software Engineering, Springer International Publishing (2017). https://doi.org/10.1007/978-3-031-02549-5, https://link.springer.com/10.1007/978-3-031-02549-5
4. CircleCI: Circleci, https://circleci.com, accesed on 2024-06-21
5. CircleCI: Circleci examples, https://circleci.com/docs/examples-and-guides-overview/, accesed on 2024-06-21
6. CircleCI: Introduction to CircleCI migration - CircleCI, https://circleci.com/docs/migration-intro/, accesed on 2024-03-28
7. Codefresh: GitHub Actions pipeline integration, https://codefresh.io/docs/docs/integrations/github-actions/, accesed on 2023-12-06
8. Codefresh: Jenkins pipeline integration/migration, https://codefresh.io/docs/docs/integrations/jenkins-integration/, accesed on 2023-12-06
9. Colantoni, A., Berardinelli, L., Garmendia, A., Bräuer, J.: 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. p. 784–792. MODELS '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3550356.3561597, https://doi.org/10.1145/3550356.3561597
10. Colantoni, A., Berardinelli, L., Wimmer, M.: 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, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3417990.3420203, https://doi.org/10.1145/3417990.3420203
11. Düllmann, T.F., Kabierschke, O., Hoorn, A.v.: 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). pp. 214–223 (2021). https://doi.org/10.1109/SEAA53835.2021.00035
12. Eclipse Foundation: ATL Transformation Language, https://eclipse.dev/atl, accesed on 2024-06-21
13. Eclipse Foundation: Object Constraint Language, https://projects.eclipse.org/projects/modeling.mdt.ocl, accesed on 2024-06-21
14. Eclipse Foundation: Xtext, https://eclipse.dev/Xtext, accesed on 2024-06-21
15. Flores, A.: A Two-Level Model-Driven Engineering Approach for Reengineering CI/CD Pipelines. Master's thesis, Faculty of Engineering, University of Porto (2024), https://dreflo.github.io/auto-cicd-migration/doc/thesis/thesis.pdf
16. Flores, A., Gião, H.: ACICDTrip Repository (6 2024). https://doi.org/10.5281/zenodo.11922315, https://github.com/DreFlo/auto-cicd-migration/releases/tag/v0.1.1

17. Gebregziabher, D.: Github actions importer is now generally available (Mar 2023), https://github.blog/2023-03-01-github-actions-importer-is-now-generally-available/
18. GitHub: GitHub Actions, https://github.com/features/actions, accesed on 2024-06-21
19. GitHub: Manually migrating to GitHub Actions, https://docs.github.com/en/actions/migrating-to-github-actions/automated-migrations, accesed on 2023-12-06
20. GitLab: GitLab CI/CD, https://docs.gitlab.com/ee/ci, accesed on 2024-06-21
21. GitLab: Plan a migration from another tool to GitLab CI/CD | GitLab, https://docs.gitlab.com/ee/ci/migration/plan_a_migration.html, accesed on 2023-12-06
22. da Gião, H., Flores, A., Pereira, R., Cunha, J.: Chronicles of ci/cd: A deep dive into its usage over time (2024)
23. Golzadeh, M., Decan, A., Mens, T.: On the rise and fall of CI services in GitHub. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 662–672. IEEE, New York, New York, United States (2022). https://doi.org/10.1109/SANER53432.2022.00084, https://ieeexplore.ieee.org/document/9825792/
24. Grieger, M., Fazal-Baqaie, M., Engels, G., Klenke, M.: Concept-based engineering of situation-specific migration methods. In: Kapitsaki, G.M., Santana de Almeida, E. (eds.) Software Reuse: Bridging with Social-Awareness. pp. 199–214. Lecture Notes in Computer Science, Springer International Publishing (2016). https://doi.org/10.1007/978-3-319-35122-3_14
25. Jenkins: Jenkins, https://www.jenkins.io, accesed on 2024-06-21
26. JetBrains: The state of developer ecosystem in 2023, https://www.jetbrains.com/lp/devecosystem-2023, accesed on 2024-02-01
27. Kazman, R., Woods, S., Carriere, S.: Requirements for integrating software architecture and reengineering models: CORUM II. In: Proceedings Fifth Working Conference on Reverse Engineering (Cat. No.98TB100261). pp. 154–163 (1998). https://doi.org/10.1109/WCRE.1998.723185, https://ieeexplore.ieee.org/abstract/document/723185
28. Kim, G., Humble, J., Debois, P., Willis, J.: The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. ITpro collection, IT Revolution Press, Portland, Oregon, United States (2016), https://books.google.pt/books?id=ui8hDgAAQBAJ
29. Muthoo, S., D. Reeve II, W., Boyd, T., Banken, H.: Yamldiff, https://github.com/sahilm/yamldiff, accesed on 2024-06-21
30. Pulgar, C.: 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. p. 225–228. MODELS '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3550356.3552395, https://doi.org/10.1145/3550356.3552395
31. Rivera, L.F., Villegas, N.M., Tamura, G., Jiménez, M., Müller, H.A.: Uml-driven automated software deployment. In: Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering. p. 257–268. CASCON '18, IBM Corp., USA (2018)
32. Rostami Mazrae, P., Mens, T., Golzadeh, M., Decan, A.: On the usage, co-usage and migration of ci/cd tools: A qualitative analysis. Empirical Soft-

ware Engineering **28**(2), 52 (2023). https://doi.org/10.1007/s10664-022-10285-5, https://doi.org/10.1007/s10664-022-10285-5

33. Travis: Travis ci, https://www.travis-ci.com, accesed on 2024-06-21

34. Wurster, M., Breitenbücher, U., Falkenthal, M., Krieger, C., Leymann, F., Saatkamp, K., Soldani, J.: The essential deployment metamodel: a systematic review of deployment automation technologies. SICS Software-Intensive Cyber-Physical Systems **35**(1), 63–75 (2020). https://doi.org/10.1007/s00450-019-00412-x, http://link.springer.com/10.1007/s00450-019-00412-x

A Platform-independent Meta-Model to Platform-specific Meta-Model Class Mappings

Tables 1 and 2 show the mappings of PSMM classes to the PIMM's **ScriptJob** and **PipelineCallJob** classes.

Table 1. ScriptJob class mappings

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

Table 2. PipelineCallJob class mappings

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

B CI/CD Logs Comparison

Listing 1.1 shows the abridged logs of an example CircleCI pipeline. Listing 1.2 shows the abridged logs of the same pipeline after being migrated to GHA.

Listing 1.1. CircleCI Python example logs (abridged).

```
(...)
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
(...)
=========================== short test summary info ==========
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 ======
```

Listing 1.2. GitHub Actions Python 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
(...)
=========================== short test summary info ==========
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 ======
(...)
```

C TDSL Example

Listing 1.3 shows an example TDSL script used for a CircleCI migration. Before the CircleCI PSM is transformed to a PIM (in the `before translating` section), this script selects which workflow should be migrated. After the PIM has been created (in the `while translating` section), this script replaces the second step on the "frontend-test" job with a command to run "npm install".

Listing 1.3. TDSL Example.

```
before translating {
    on circleci select workflow frontend
}
while translating {
    replace step 2 on 'frontend-test' with command {
        script 'npm install'
    }
}
```

D Ignored YAML differences

What follows is a list of the YAML differences we ignored:

- **Trailing whitespace** - we ignore any differences is 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, i.e. 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]`.