A Meta-Model to Support the Migration and Evolution of CI/CD Pipelines

Anonymous Author(s)*

ABSTRACT

In the realm of industrial software development, DevOps has emerged as the preferred approach for handling the highly iterative software production process. DevOps refers to the tight integration of development and operations activities, with Continuous Integration, Continuous Delivery, and Continuous Deployment (CI/CD) being pivotal methodologies for ensuring the delivery of high-quality, iteratively developed software.

To achieve CI/CD, pipelines of activities are deployed using commercial tools. Due to the dynamic nature of these tools, CI/CD pipelines must be constantly migrated to new versions or even new tools. This is a cumbersome and error-prone activity for software engineers, which requires systematic and automated support.

To address this issue, we propose a novel approach that leverages model-driven engineering (MDE) to support the migration of CI/CD pipelines. Our 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.

Our contribution comprises a high-level, abstract meta-model designed to represent the structure of existing CI/CD pipelines and a migration toolchain to transform them into a new target format. We validated our meta-model by modeling 200 existing pipelines and achieved a 96.5% success rate in applicability. Furthermore, we conducted a detailed case study demonstrating our model-driven approach's practical applicability in real-world migration scenarios. Finally, we demonstrate that our meta-model promotes equivalence between an original pipeline and a new one generated from it in a different technology by showing through test cases that the execution traces of both pipelines are identical.

CCS CONCEPTS

• Software and its engineering → Model-driven software engineering; **Software notations and tools**; **• Social and professional topics → Reengineering**.

KEYWORDS

CI/CD, DevOps, Reengineering, Model-Driven Engineering

ACM Reference Format:
Anonymous Author(s). 2018. A Meta-Model to Support the Migration and Evolution of CI/CD Pipelines. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

In 2022, the Information Technology (IT) market was valued at approximately $9.325 trillion globally [7], with maintenance and upkeep costs constituting half of total IT expenditures. Significantly, half of these costs are attributed to unplanned work, emergencies, and various changes [11]. The frequent misalignment of development teams and IT teams partially induces this cost. Traditionally, development teams focus on responding swiftly to market changes and customer demands, whereas IT teams focus on providing a stable, secure, and reliable service. This dynamic leads to technical debt, meaning decisions made over time lead to increasingly more complex problems and slower operations and thus impede the achievement of the organization's goals [11].

In recent years, DevOps, the seamless integration of development and IT teams, has emerged to enable organizations to respond to market demands with unparalleled agility in hopes of addressing the aforementioned problems. In particular, methodologies and technologies for Continuous Integration, Continuous Delivery, and Continuous Deployment (CI/CD) have emerged as a means for organizations to achieve rapid and frequent delivery of changes. CI/CD pipelines automate code changes' integration, testing, and deployment, facilitating frequent and reliable software releases [11].

Organizations have the option to employ various technologies such as GitHub Actions[1], GitLab CI/CD[2], Travis CI[3], CircleCI[4] or Jenkins[5]. Indeed, researchers have found that open-source projects often use several technologies simultaneously within the same project [8]. Moreover, they have also reported a growing rate of changes in the technologies used, peaking in 2021, when more than 15% of all commits to code repositories included changes in the CI/CD technologies used. These facts highlight a significant challenge: the necessity for migration between different CI/CD technologies. This need arises as each tool evolves, offering unique features and integrations, which can lead to shifts in technology preferences or requirements over time. Consequently, the ability to migrate efficiently between these diverse technologies is crucial for maintaining the agility and effectiveness of software development.

However, there is almost no support for migrating CI/CD pipelines. Apart from GitHub, which provides a tool that aims at migrating about 80% of scripts from other technologies [10], others offer only guidelines or focus only on analyzing existing CI/CD pipelines.

[1]https://github.com/features/actions
[2]https://docs.gitlab.com/ee/ci/
[3]https://www.travis-ci.com/
[4]https://circleci.com
[5]https://www.jenkins.io/

Our objective is to streamline the migration and evolution process by developing a meta-model capable of representing diverse CI/CD pipelines in a technology-agnostic manner. This meta-model serves as a common language capable of encapsulating the fundamental concepts of CI/CD.

Our approach sets itself apart from existing work in several ways. Unlike other meta-models, ours draws inspiration from several of the most widely used CI/CD technologies and is designed to abstract concepts from these technologies. Additionally, our objectives diverge from existing approaches, as our primary aim is to aid the migration and evolution of CI/CD pipelines, facilitating the reengineering process and preserving equivalence between the original and generated pipelines.

Our methodology empowers developers to transition seamlessly between CI/CD technologies, eliminating the need to rewrite pipelines or undergo extensive learning curves. By enabling the representation of existing pipelines through a model, new pipelines in different technologies can be effortlessly generated.

The following questions guide our research:

RQ1: Can we define a meta-model encompassing the various existing CI/CD technologies and languages, effectively abstracting the core elements and functionalities in a unified manner?
Our goal for RQ1 is to examine various CI/CD technologies and languages and develop a meta-model capable of representing the core concepts found in these languages. We intend to create an abstraction that transcends the specifics of individual languages. We study this RQ in Section 3.

RQ2: Can our meta-model accurately represent real-world CI/CD pipelines, thereby demonstrating its applicability in modeling existing configurations?
For RQ2, our goal is to evaluate the capability of our meta-model to represent real-world pipelines. Thus, we collected a set of GitHub Actions scripts and converted them into models that conform to our meta-model. Additionally, we conducted a real-world case study using our meta-model to migrate CI/CD technologies from CircleCI to GitHub Actions. We study this RQ in Section 4.

RQ3: To what degree does our meta-model support the creation of meaningful and syntactically correct and semantically equivalent pipeline models?
For RQ3, we aim to demonstrate that the models produced by our meta-model result in syntactically correct representations of a selected CI/CD technology. We begin the validation process with the smallest valid model, generating the corresponding GitHub Actions pipeline, and ensuring its compliance with GitHub Actions syntax. As we gradually expand the model to encompass all meta-model elements, we continuously verify that the generated pipelines maintain syntactic correctness throughout this progression. Furthermore, we aim to assess whether the pipelines generated using our meta-model in the reengineering process retain semantic equivalence with the original pipelines. We migrate various CircleCI pipelines to GitHub Actions using our approach. Subsequently, we execute the original and generated pipelines in their respective code repositories and compare the commands executed. We study this RQ in Section 5 and Section 6.

To demonstrate the applicability of our methodology, we present a case study of CI/CD migrations that happened between different technologies, modeling them using our meta-model (Section 7). This shows the real-world application of our methodology and meta-model presented. In Section 8, we discuss the potential threats to the validity of our work, and finally, in Section 9, we present our concluding remarks. The following section presents related work and how it compares with ours.

2 RELATED WORK

This section presents work that entails applying MDE to the DevOps process. As we will detail next, our work differentiates from others as they focus on a specific niche of DevOps or CI/CD, with the primary goal of generating code infrastructure or being compatible with specific technologies. On the other hand, we aim to produce a domain-agnostic meta-model to help in migration processes.

2.1 Methodology

We realized a literature review where we queried four major digital libraries: *ACM Digital Library*[6], the *IEEE Xplore*[7], *ScienceDirect*[8] and *SpringerLink*[9], for papers containing both topics of interests, namely MDE and DevOps. We use DevOps and not just CI/CD as CI/CD is part of DevOps, and thus, we can potentially find more related works. We did this work in February 2024. We started by defining an initial query for searching these libraries for relevant publications as follows:

DevOps AND ("model-driven" OR "model driven")

After running the query and some initial screening of the papers, we realized several terms related to DevOps should also be considered, namely 'MLOps', 'DevSecOps', 'CloudOps', 'AIOps', and 'DataOps', which represent various fields of application of DevOps. Thus, we included them in the search query. We also included expressions specific to CI/CD. Regarding the model part, we did not include terms such as "model-based" or similar. We focus on model-driven approaches for which models are paramount, in contrast with model-based approaches, where models are less significant. Thus, we extended the original query as follows:

((DevOps OR MLOps OR ChainOps OR DevSecOps OR CloudOps OR AIOps OR DataOps OR "CI/CD" OR "Continuous Integration" OR "Continuous Deployment" OR "Continuous Delivery")
AND
("model-driven" OR "model driven"))

After gathering the papers containing the search terms, we manually filtered, by reading the content, the papers whose work involved the application of MDE techniques in various areas of DevOps using the following inclusion and exclusion criteria:

Inclusion criteria

IC1 The work in the paper applies MDE to the field of DevOps by modeling DevOps processes. This includes any publication whose work creates or improves upon techniques, tools, modeling languages and frameworks for using MDE in DevOps processes.

[6]https://dl.acm.org
[7]http://ieeexplore.ieee.org
[8]https://www.sciencedirect.com
[9]https://link.springer.com

IC2 The work in the paper applies MDE to the field of DevOps by modeling the adoption or migration of DevOps practices or technologies. This includes any publication whose work models the adoption of DevOps practices and technologies for purposes of documentation or support.

IC3 The paper exposes the requirements for using MDE in DevOps. This includes any publication whose work gathers requirements from industrial or academic case studies for using MDE in any aspect of DevOps.

Exclusion criteria

EC1 The work does not address the topics of DevOps and MDE. We rejected any publications whose work did not contribute to any DevOps or MDE fields.

EC2 The paper applies DevOps techniques to MDE, but not vice versa. We dismissed the works that apply DevOps techniques to improve several aspects of MDE.

EC3 Secondary, tertiary, or survey works. These works were also dismissed from this literature review since our goal is to evaluate original primary works and report on the state of the art of MDE and DevOps.

EC4 (Extended) abstracts. We dismissed abstracts or extended abstracts since they did not provide enough information to evaluate their contributions to this field.

EC5 Proceedings. We also dismissed proceedings of conferences since they contain several publications, most of which are irrelevant to our query. The ones that are, should appear in the search on their own.

EC6 Domain-specific. We discarded works that are only interested in DevOps and CI/CD in a specific domain.

2.2 Results and Analysis

Running the defined query, we found the number of papers reported in the third column of Table 1. Note that we have run the query against the complete text of the publications (title, abstract, full text, etc.) to increase the chances of finding all the relevant papers. Indeed, the addition we made to the query produced more results.

Table 1: Number of papers found in each library.

Venue	First query	Second query
ACM Digital Library	184	214
IEEE Xplore	37	41
ScienceDirect	132	134
SpringerLink	690	803
Total	718	744

Following the aforementioned process, we identified eight papers relevant to our research. Here, we highlight some of their contributions and elaborate on how our work differs from them.

Colantoni et al. [6] introduce DevOpsML, an innovative approach for modeling DevOps Processes and Platforms, presenting a platform meta-model that captures the amalgamation of tools, interfaces, capabilities, and concerns. Additionally, the authors introduce a linking meta-model designed to connect various platforms and elucidate the DevOps process. DevOpsML is adept at modeling processes, platforms, and libraries 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. [5] present an ongoing project centered on the integrated modeling and scenario simulation of continuous delivery pipelines. Users can define DevOps 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 various supports, including an Xtext-based textual editor, a Sirius-based graphical viewer, and a GEMOC-based interpreter. An extended case study based on Keptn, an open-source tool for DevOps automation in cloud-native applications, is also presented. On the other hand, 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 DevOps in a technology-independent manner.

Rivera et al. [13] tackle the challenges associated with deployment in continuous delivery and DevOps. Their proposal introduces Urano, a mechanism designed to automate the deployment process by using UML to specify software architecture and its associated deployment. The authors use model-driven architecture principles to generate executable deployment specifications from user-defined UML deployment diagrams. They enhance these diagrams by defining and applying a UML profile that captures the semantics and requirements of installing, configuring, and updating software components. This enhancement allows for more expressive deployment specifications and their automatic realization. In contrast, our approach involves creating a meta-model compatible with most existing CI/CD technologies and flexible enough to represent 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. [3] 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 based on the Kaloom use case. The aim of this modeling framework is to act as a central component within the overarching DevOps engineering framework, serving as 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. [9] 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, a StalkCD DSL aimed at facilitating interoperability and transformation between different formats, as well as a set of transformations between tool-specific CI/CD definitions and analysis tools. 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 technologies. Furthermore, we tested our meta-model for its ability to represent CI/CD pipelines and for tasks extending beyond mere representation, such as reengineering of pipelines across technologies.

Pulgar et al. [12] introduce a meta-model heavily influenced by GitHub Actions. Their goal is to ensure that each modification to a pipeline is valuable. To validate their approach, the authors utilized three open-source projects. They analyzed the various versions of their CI/CD pipelines by parsing the configuration files into models and comparing the changes between the models of previous and subsequent files. 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. [2] develop a model for DevOps deployment choices, aiding enterprises in tailoring a suitable DevOps approach to meet their requirements while contemplating potential process reconfigurations. 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 primary objective is to leverage the meta-model for the reengineering and development of CI/CD pipelines.

Wurster et al. [14] propose the essential deployment meta-model (EDMM) to enable a common understanding of declarative deployment models by facilitating the comparison, selection, and migration of technologies. Wurster et al.'s approach is a meta-model of software deployment, but it is intended to help users to select the best deployment technology for their scenario. Our objective extends beyond tool selection, encompassing practical applications such as facilitating development and migration processes.

3 FROM TECHNOLOGIES TO A META-MODEL

In this section, we present our meta-model, starting by describing the methodology to achieve it.

3.1 Methodology

To define the fundamental concepts of CI/CD, we selected four technologies: GitHub Actions, Jenkins, GitLab CI/CD, CircleCI, and Travis CI. The selection of these tools was influenced by their prevalence. As identified in a survey analyzing CI/CD configuration files from open-source repositories on GitHub, these four technologies account for 96.3% of all found pipelines [8].

We undertook extensive research involving the examination of the documentation for these tools. Moreover, we analyzed numerous pipeline configuration files from the documentation, but also from existing projects. Arguably, the current (best) practices of using these technologies are in ongoing projects.

The goal was to devise a meta-model capable of representing structures encompassing most features supported by these languages. Indeed, this meta-model aims to capture the essence of any CI/CD pipeline. Thus, each of the concepts we present in the meta-model are in most of these technologies, although they may have different names. Indeed, our model includes features from different technologies, being more than an intersection of all technologies, as it contains features that are not necessarily in all technologies.

3.2 The Meta-Model

Our meta-model is illustrated in Fig. 1. It incorporates various functionalities, including the execution of parallel jobs, the flexibility to determine pipeline run conditions, the specification of environment variables and permissions, the inclusion of conditional statements (if and else), and the definition of software and operating system requirements. In the next paragraphs, we detail each of the concepts.

Pipeline This serves as the core class within our meta-model, symbolizing the CI/CD pipeline and encompassing various elements of pipelines and our meta-model. Each pipeline has a user-defined *Name* and the number of times the pipeline can be executed concurrently (*Concurrent*). For instance, if three commits to the repository simultaneously trigger the pipeline execution, we may decide that just two can be run concurrently by setting *Concurrent* to 2 (e.g., to save resources).
Multiple instances of the *Job* class can be associated with the *Pipeline*, reflecting the reality that a single CI/CD pipeline may comprise numerous tasks.

Job This class represents tasks within CI/CD pipelines, functioning as a crucial component in configuration files for organizing commands based on their distinct functionalities. For instance, a job may group all the tasks related to testing. The class encompasses three attributes: *Name*, signifying the job's identifier; *AllowFailure*, a boolean indicating whether the job should abort in the event of failure; and *Description*, which allows users to describe its purpose. Within its structure, the class incorporates instances of *Artifact* to depict both input and output artifacts. For instance, a job may consume a certain text file and produce an HTML report.
Moreover, the job may specify a list of *Tools* necessary to execute the commands to run. Depending on the CI/CD technology used, this may be interpreted as requirements necessary to be installed before the execution of the job (e.g., a Java compiler).

Artifact This class depicts the inputs and outputs of jobs within the pipeline. It incorporates an attribute, *Name*, which allows to define each of these inputs and outputs. Currently, these strings are not validated, for instance, to be valid available inputs. However, tool support could provide such aid.

Tool This class represents the necessary software for the pipeline and its various commands, where the attribute *Name* can be used to express each software. This can be used for a *Pipeline*, for instance, defining the pipeline needs to be executed under the *ubuntu* operating system, or by a *Job*, determining the job needs *Java* to be executed, or by a *Command*, which requires *make* to run. Once more, no validation is provided for these strings.

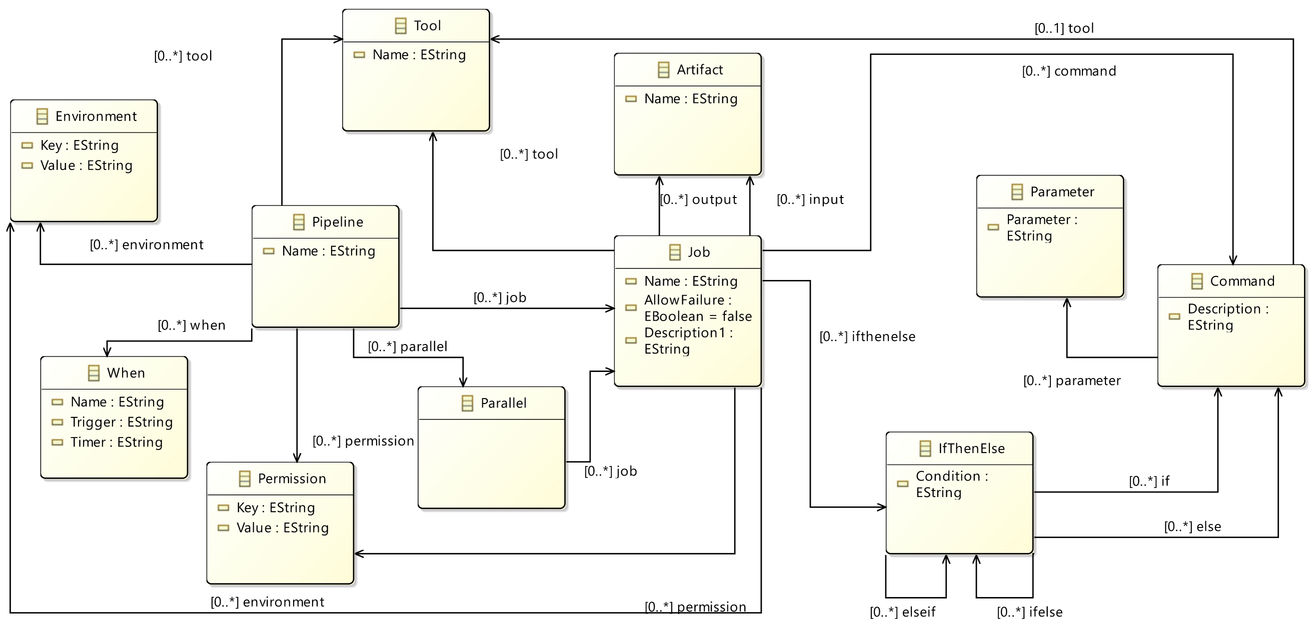


Figure 1: The CI/CD meta-model.

Command This class represents the smallest execution piece of a pipeline (e.g., execute a Python command – `python ...`, copy a file – *cp* ...). The *Description* attribute encodes the name of the command to be executed (e.g., cp).

Each *Command* may require a *Tool*, representing the software employed in the command. It may also contain multiple instances of the class *Parameter*, delineating the command parameters.

Parameter This class is defined with a string attribute, *Parameter*, and is employed to depict parameters associated with a given command. Once more, these parameter strings are not validated. For instance, a copy command may have two arguments, one for the source and another for the destination.

IfThenElse A job has a list of commands, but some commands should only be executed under certain circumstances. Thus, a job can have multiple *IfThenElse* statements which encode the common *if-then-else* constructor. The *Condition* itself is expressed using a string. We have considered making it more expressive (e.g., using classes to represent boolean operators), but this would make it (much) more complex. We thus decided to make it simpler. Nevertheless, this requires some extra validation to ensure the string indeed encodes a valid boolean expression.

Each branch may include a list of commands to be executed while also referencing other instances of the *IfThenElse* class, indicating 'ifelse' and 'elseif' alternatives.

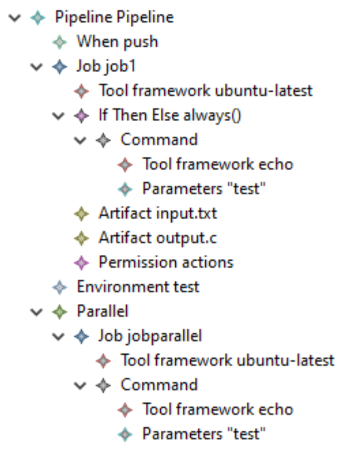
Parallel Regarding the order of the execution of the jobs, there are two different trends in current technologies: *1)* consider each job is executed after the previous one finished (i.e., sequentially) and have constructors to express parallel executions, or *2)* consider all jobs execute in parallel and have constructors to make jobs' subsets execute sequentially. Our meta-model is based on the first approach: all jobs execute sequentially. If the user wants to execute jobs in parallel, the class *Parallel* can be used to group such jobs. In the extreme case that all jobs are within a parallel class, the model represents the approach *2)*; note if the model has two or more parallel instances, they will be executed sequentially.

Our approach mirrors the GitHub Actions model, where jobs can be organized into groups that run in parallel to each other. Similarly, it aligns with the Jenkins and Travis CI model, where jobs execute in parallel sections.

When This class is employed to specify the conditions under which the pipeline should be executed. It is characterized by a string attribute, *Name*, and strings *Trigger* and *Timer*, which are utilized to denote actions and time periods (cron-like) determining when the pipeline should run. For instance, a pipeline may be executed every time there is a new commit to the corresponding repository (trigger 'on-commit') or every day at a particular hour (timer). A pipeline may execute because of different conditions.

Environment It is common to define dictionaries of information required by different pipeline parts or by particular jobs. For instance, one may define a value to be used throughout the pipeline but to be referred to by a name (key) instead of the value itself (similar to a variable). Thus, this class *Environment* encapsulates the environment variables utilized in a pipeline or jobs. It is defined by two string attributes: *Key* and *Value*, representing the key-value pairs of a dictionary.

Permission This class embodies permissions for both individual jobs and the entire pipeline. It is defined by two string attributes, *Key* and *Value*, which denote the key-value pairs specifying the permissions. For instance, one may define that the pipeline's 'actions' have only 'read' permissions.



```
Pipeline Pipeline
    When push
    Job job1
        Tool framework ubuntu-latest
        If Then Else always()
            Command
                Tool framework echo
                Parameters "test"
        Artifact input.txt
        Artifact output.c
        Permission actions
    Environment test
    Parallel
        Job jobparallel
            Tool framework ubuntu-latest
            Command
                Tool framework echo
                Parameters "test"
```

Figure 2: Pipeline example.

```
name: Pipeline
on: ["push"]
env:
 test: 1
jobs:
 job1:
  runs-on: "ubuntu-latest"
  permissions:
   actions: read
  outputs:
   output1: test.c
  inputs:
   input1: test.txt
  steps:
   if: ${{ always() }}:
    - run: echo "test"
 job1parallel:
  runs-on: "ubuntu-latest"
  steps:
   - run: echo "test"
```

Listing 1: GitHub Actions script for the model.

3.3 An Example

In Fig. 2, we present an example of utilizing our meta-model to construct a pipeline. We use the syntax provided by the Eclipse Modeling Framework (EMF)[10] as we implemented our meta-model using this tool. Note some attributes are not shown in this visualization (e.g., the value of the environment variable). The resulting GitHub Actions configuration file in Listing 1. This pipeline consists of one *Pipeline* entity, which activates upon a push to the repository, utilizing a *When* object for this purpose. Additionally, the pipeline incorporates two concurrently running *Job*s: "job1" and "jobparallel", nested within a *Parallel* object. It also includes an *Environment* representing an environment variable.

The first *Job*, "job1", has a *Tool* referencing the operative system "ubuntu-latest". It includes an *IfThenElse* construct with a condition of "always()", indicating that the condition always executes. Within this *IfThenElse* block, there exists a command with a *Tool* and a *Parameter*, representing a command that prints "test" to the terminal. Additionally, the *Job* incorporates an *Input* and *Output* entity, representing input and output files, respectively. Furthermore, two *Permission* objects are included, granting the job read permissions.

The second *Job*, "jobparallel", placed within a *Parallel* object, showcases parallel execution of jobs, contrasting with sequential execution by default.

3.4 Addressing RQ1

To address RQ1, we devised the meta-model presented in Fig. 1. This meta-model encompasses and represents the majority of features and functionalities present in the most popular CI/CD tools. Moreover, the meta-model was designed to be concise and abstract from specific language syntax. Thus, this meta-model helps to answer RQ1 positively. In the following sections, we will evaluate this meta-model and show it can indeed cope with existing pipelines and support the migration of scripts from one tool to another.

[10]https://eclipse.dev/modeling/emf/

4 USEFULNESS OF THE META-MODEL

We intend to show our meta-model has applicability by showing it can represent existing pipelines. Thus, we implemented our meta-model utilizing EMF and subsequently developed a parser for GitHub Actions to generate models compliant with our meta-model using Xtext[11], which integrates well with EMF.

The decision to create a parser specifically for GitHub Actions was based on a study indicating that GitHub Actions is overwhelmingly the most used CI/CD tool across open repositories on GitHub, representing almost 58% of all pipelines [8]. This choice is further substantiated by GitHub's status as one of the most popular software hosting services, boasting over 100 million users as of 2023[12].

Leveraging Xtext, we generated a grammar based on our previously constructed meta-model, subsequently adapting the terminal symbols to parse GitHub Actions configuration files.

Our goal with this phase is to answer RQ2, thus validating that our meta-model can represent real-world pipelines.

4.1 Collecting Pipeline Configuration Files

We randomly selected 200 repositories from a dataset of CI/CD repositories utilizing GitHub Actions pipelines [8], from which we extracted the GitHub Actions configuration files. Some of the scripts included special characters, such as Asian languages' letters (e.g., in file names). To avoid parsing issues, we eliminated such characters. Additionally, we corrected the indentation of some scripts to facilitate the parsing process (the YAML syntax of the files is more permissive than our parser).

4.2 Results

We used our parser to convert GitHub Actions configurations into a model conforming to our meta-model. Our parser successfully processed 193 files, representing 96.5% of the total files. We could not parse six files because of excessive nesting in the configuration file. Note this is a parsing issue and not a deficiency in the meta-model. Additionally, we encountered difficulties parsing one file that included the option for a default job. Indeed, this is not a possibility in our meta-model. Nevertheless, this was the single case we found in the 200 scripts randomly selected. Thus, from this sample, we can say this feature is not very used.

4.3 Addressing RQ2

To address RQ2, we observe that our meta-model effectively represents the majority of real-world GitHub Actions pipelines. Although we focus this evaluation on GitHub Actions pipelines, since our meta-model is inspired by several technologies, this study provides some empirical evidence that our meta-model is useful as it is capable of representing most existing pipelines.

5 SYNTACTICAL CORRECTNESS OF MODELS

In this validation phase, we aim to demonstrate that models conforming to our meta-model produce syntactically correct pipelines according to a selected CI/CD technology. To achieve this, we employ Acceleo[13] to generate GitHub Actions pipelines corresponding

[11]https://eclipse.dev/Xtext/
[12]https://github.blog/2023-01-25-100-million-developers-and-counting/
[13]https://eclipse.dev/acceleo/

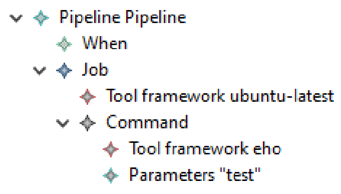


Figure 3: Smallest model.

```
name: Pipeline
on: ["push"]
jobs:
 job1:
  runs-on: "ubuntu-latest"
 steps:
  - run: echo "test"
```

Listing 2: Smallest valid GitHub Actions pipeline.

to various models conforming to our meta-model. We start with the smallest possible model and progressively enlarge it. We aim to establish that every model generated using our meta-model results in syntactically correct code conforming to GitHub Actions.

Subsequently, we utilize the widely adopted GitHub Actions script validator[14] to verify the generated scripts based on the language syntax. This tool analyzes GitHub Actions scripts by comparing them against published JSON schemas.

Our goal with this phase is to answer RQ3 by showing that all models generated using our meta-model are syntactically correct.

5.1 Smallest Valid Model

The smallest model resulting in a syntactically correct GitHub Actions pipeline is a *Pipeline* object with one *When*, one *Job*, one *Tool* and one *Command*; the generated script from this model is presented in Listing 2.

5.2 Adding Elements to Valid Models

After constructing and validating the smallest possible model, we validate the changes in the generated GitHub Actions scripts resulting from adding elements to our initial model and verifying for each step if the resulting GitHub Actions pipeline remains syntactically correct. The possible model evolutions are:

- Add an *Environment* to a *Pipeline*.
- Add an *Environment* to the first *Job* of the *Pipeline*.
- Add a second *When* to the *Pipeline*.
- Add a *Permission* to the *Pipeline*.
- Add a *Permission* to the first *Job* of the *Pipeline*.
- Add a *Parallel* to the *Pipeline*.
- Add a *Job* to the *Parallel* object.
- Add a second *Job* object to the *Pipeline*.
- Add a *Tool* to the *Pipeline*.
- Add a *Tool* to the first *Job*.
- Add an *Artifact* as an output to the first *Job*.
- Add an *Artifact* as an input to the first *Job*.
- Add an *IfThenElse* to the second *Job*.
- Add a *Command* as an *if* to the *IfThenElse*.
- Add a *Command* as an *ElseIf* to the *IfThenElse*
- Add a *Command* as an *Else* to the *IfThenElse*
- Add a second *Parameter* to a *Command*

After applying all these changes to the model described in Section 5.1, the resulting GitHub Actions pipeline is in Listing 3. For each of these model evolutions, we have generated the corresponding script and validated it, always getting a positive result.

[14]https://github.com/mpalmer/action-validator

```
name: Pipeline
on: ["push", "pull"]
runs-on: "ubuntu-latest"
env:
    test: 1
permissions:
    actions: read
jobs:
    job1:
        runs-on: "ubuntu-latest"
        permissions:
            actions: read
        outputs:
            output1: test.c
        inputs:
            input1: test.txt
        steps:
            - run: echo "test" "test2"
    job2:
        needs: job1
        runs-on: "ubuntu-latest"
        permissions:
            actions: read
        steps:
            - run: echo "test"
            if: ${{ condition }}:
                - run: echo "test"
            if: ${{ !condition && condition2 }}:
                - run: echo "test"
            if: ${{ !(condition || condition2) }}:
                - run: echo "test"
    job1parallel:
        runs-on: "ubuntu-latest"
        permissions:
            actions: read
        steps:
            - run: echo "test"
```

Listing 3: GitHub Actions pipeline for the finished model.

5.3 Addressing RQ3

By generating and validating the smallest possible pipeline from a model conforming to our meta-model, and ensuring that the model adheres to GitHub Actions specifications (i.e., the code generated from the model), along with demonstrating that augmenting existing models maintains the validity of the corresponding GitHub Actions pipeline, we provide empirical evidence that all models created using our meta-model generate syntactically correct GitHub Actions pipelines.

6 MODEL EQUIVALENCE

In Section 3, we have presented a meta-model to represent the essence of CI/CD pipelines. We have then shown in Section 4 that our meta-model can represent real-world scripts and that the scripts are syntactically correct (Section 5).

As the main goal of our work is to present a meta-model that can aid in migration and evolution processes, we want to evaluate to which extent our approach can also ensure the semantic correctness of this process, that is, given a pipeline in a certain technology, abstracting it into a model conforming to our meta-model, and generating a new script in a new technology, maintains semantic equivalence between both pipelines.

Note that we do not intend to prove the semantic equivalence of scripts, as this would require proving that two distinct technologies act similarly. Instead, we seek to gather empirical evidence that our approach can promote such an equivalence. Thus, we show that the results from executing both scripts are the same.

To achieve this, we gathered several CI/CD pipeline files from one language and parsed them into models conforming to our meta-model. Subsequently, we generated equivalent files for an alternate language and proceeded to compare the outcomes of executing pipelines in a given repository utilizing both technologies. This comparative analysis enables us to ascertain the consistency and reliability of our approach.

6.1 Collecting CI/CD Pipelines

To perform this evaluation, it is not sufficient to gather pipeline scripts as we need to execute them, migrate them to another technology, execute the newly generated pipeline, and show both the original and the generated produce the same result. Thus, we need scripts and the corresponding code to be able to execute them.

CircleCI provides a set of scripts to aid users in learning their technology.[15] The scripts are accompanied by code repositories that allow the scripts to actually execute, thus allowing us to analyze the results from the original script and the one we can generate. These examples encompass most features integral to a comprehensive CI/CD pipeline, such as jobs, operating system and tool requirements, commands, conditionals, and environment variables. The five CircleCI scripts utilized for this validation are as follows:

Setting up a Node.js Application This script orchestrates the installation of various JavaScript packages required for a Vue.js application. It then proceeds to execute multiple unit tests, both backend and frontend, utilizing Cypress.

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

Example of a Java Application This script incorporates the installation of Dockerize to facilitate testing of a Java Spring Boot Server coupled with a PostgreSQL database.

Example of a .NET Application This pipeline encompasses the processes of building, testing, and storing the test results of a .NET application.

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

6.2 Parsing and Generating Configuration Files

Subsequently, we developed an Xtext parser utilizing our meta-model to parse CircleCI configuration files and generate models conforming to our meta-model for these scripts.

However, the models derived from CircleCI configuration files contained commands and tools exclusive to CircleCI. These specific commands and tools needed to be replaced with their GitHub equivalent counterparts in the generated GitHub Actions scripts. The modifications made to the parsed CircleCI models were as follows:

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

- We substituted CircleCI commands (*Command*) such as "checkout", "python/install-package", "node/install-packages" with their GitHub Actions or generic equivalents, such as "actions/checkout@v4", "pip install", "yarn install", respectively. In these cases, the model was kept untouched, but some attribute values were updated. For instance, the attribute *Name* of the *Command* was changed from "checkout" (CircleCI) to "actions/checkout@v4" (GitHub).
- We replaced CircleCI tools (*Tool*) like "circleci/python@2.1.1", "cimg/openjdk:11.0", "cimg/postgres:14.1", "cimg/node:15.1" with GitHub Actions equivalents "python-version: "2.1"", "java-version: "11"", "postgres-version: "14.1"". "node-version: "15.1"" respectively. However, in CircleCI the underlying operating system (OS) is not explicit in the scripts, but in GitHub Actions needs to be. Thus, we evolved the model to include another *Tool* to express the necessary OS (we found the specific OS in the CircleCI's documentation).

All these changes can be automated, as they primarily involve mapping between CircleCI and GitHub Actions features. For a fully automatic process, it is necessary to have model-evolution rules that adapt the name of tools and other features used by one technology into the corresponding tools and features in the target technology.

Finally, we utilized Acceleo to generate the GitHub Actions configuration files from these specific model instances, resulting in files that were successfully executed.

6.3 Comparing Runtime Logs and Artifacts

We forked the repositories CircleCI makes available with their aforementioned script examples and added our generated GitHub Actions scripts to them. After running both the original and generated scripts for every example project, we gathered the execution logs made available by each of the platforms and compared them.

To make the comparison process easier, we cleaned both technologies' 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 also removed logs from the executed commands if these were overly verbose and irrelevant to comparing the technologies. Listing 4 and listing 5 are an abridged example of a log comparison.

After this cleaning, we verified that the same commands were executed and other script elements like environment variables and docker containers were correctly translated. As for packages used in the scripts, named *Orbs* in CircleCI and *Actions* in GitHub Actions, we cannot guarantee there will always be an *Action* or set of shell commands whose functionality maps directly to an *Orb*, but the logs also show that this could be done for these five examples.

Lastly, we also sought to compare all artifacts generated by the scripts. Only the .NET example script used this functionality. The artifacts generated by both the original CircleCI script and our generated GitHub Actions were the same except for timestamps and program execution times recorded in the artifact.

We provide the original logs for both CircleCI and GitHub Actions, their cleaned counterparts, the generated artifacts, and the scripts used to clean as part of our replication package [1].

```
(...)
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 4: CircleCI 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 ======
(...)
```

Listing 5: GitHub Actions Python example logs (abridged).

6.4 Addressing RQ3

By design choice, our meta-model includes names of tools and other features used in CI/CD pipelines. For instance, a *Command* has an attribute *Name* that contains the specific command to be executed by the pipeline (e.g., "pip install"). If these names differ in a target technology during a migration, they must be updated. This can easily be achieved by defining a set of model-to-model (M2M) transformations mapping different technologies' tool names.

Notably, the changes we made in the models generated from the CircleCI scripts did not change the outcomes of running the original and the generated scripts. Nevertheless, for a fully automated process, M2M transformations should be defined.

Thus, through these five test cases, we verified that the scripts generated from the meta-model were semantically equivalent to the original scripts for all supported features of the meta-model.

7 CASE STUDY

Although the migration to other CI/CD technologies is quite common [8], there is little support to perform it. Being a very technological field, one can find mostly blog posts and other gray literature describing such migrations. One of these cases is a blog post written by Ilya Cherepanov (a site reliability engineer), and Travis Turner (a tech editor).[16] In their blog entry, the authors highlight the migration of a Ruby on Rails web application's CI/CD workflow from CircleCI to GitHub Actions. The CircleCI workflow involves various stages, such as caching, data retrieval from prior executions, and the building and testing phases of the application.

To showcase the effectiveness of our meta-model in migrating and reengineering CI/CD pipelines, we took on the task of transitioning the original CircleCI pipeline to GitHub Actions. This process included representing the pipeline within our meta-model and making necessary modifications to align CircleCI-specific features with their GitHub Actions equivalents. Subsequently, we generated code from the model we created and compared the results with the authors' manual migration.

[16]https://evilmartians.com/chronicles/journey-from-circleci-to-github-actions

7.1 A Model to Represent the CircleCI Pipeline

We initiate the process by establishing a pipeline named "CircleCI migration". Subsequently, we incorporate a *Tool* tailored for the operating system on which the pipeline is intended to run. As the initial script did not specify, we opted for "ubuntu22.04" as the OS. Moving forward, we proceeded to generate pipeline jobs and begin populating them. The pipeline model can be seen in Fig. 4.

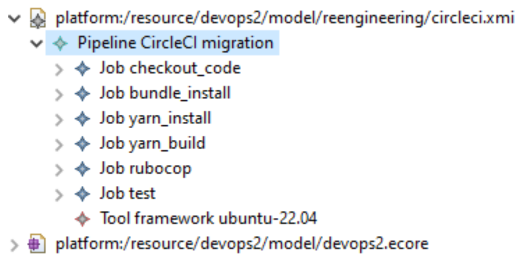

platform:/resource/devops2/model/reengineering/circleci.xmi
Pipeline CircleCI migration
Job checkout_code
Job bundle_install
Job yarn_install
Job yarn_build
Job rubocop
Job test
Tool framework ubuntu-22.04
platform:/resource/devops2/model/devops2.ecore

Figure 4: Pipeline represented in the case study.

The initial job, as seen in Fig. 5, labeled "checkout_code", involves checking out the code into the workspace. We substitute the original CircleCI commands with their GitHub Actions equivalents and seamlessly integrate these commands into the job.

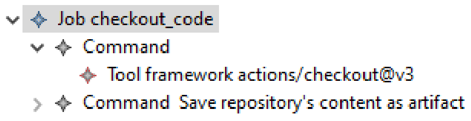

Job checkout_code
Command
Tool framework actions/checkout@v3
Command Save repository's content as artifact

Figure 5: Job "checkout_code" represented in the case study.

Moving to the second job[17], "bundle_install", its purpose is to install all dependencies of the Ruby project. We specify the requirement for "cimg/ruby:3.1.2-node" as a *Tool*. Subsequently, we introduce various commands for retrieving files from the cache, executing the bundle install command, and including an artifact for both "checkout_code" and environment variables.

The third job, "yarn_install", with the purpose of installing dependencies for the Node project, is similar to the previous one and so is the fourth job "yarn_build".

The fifth job is employed for static analysis of the project. With the *Tool* requirement specified, we include various commands for cache file retrieval, executing the rubocop command, and incorporating an artifact for "checkout_code" and environment variables. Lastly, the sixth job, "test", is tasked with running the project's tests and is similar to the previous one.

7.2 Results

We generated the pipeline configuration file using Acceleo and juxtaposed it with the results obtained from the authors' manual migration. Upon comparison, we observed minor divergences.

Notably, discrepancies were found in the pipeline trigger, as it was absent in the original CircleCI configuration file. Additionally, differences were noted in the database connection method; our migration process utilized the same commands as the original CircleCI pipeline, while the authors' manual migration employed the PostgreSQL GitHub Actions service. Those differences can be seen in Listings 6 and 7.

```
name: Release workflow for Kuvaq
on:
push:
branches:
- main
(...)
services:
postgres:
image: cimg/postgres:14.5
env:
POSTGRES_USER: root
POSTGRES_DB: circle_test
# Add a health check
options: --health-cmd pg_isready --health-interval 10s
--health-timeout 5s --health-retries 5
(...)
```

Listing 6: Pipeline from manual migration (abridged).

```
name: CircleCI migration
on: [push]
(...)
- name: Wait for database
run: dockerize -wait tcp://localhost:5432 -timeout 1m
- name: Database setup
run: bundle exec rails db:test:prepare
(...)
```

Listing 7: Pipeline resulting from our meta-model for the migration (abridged).

[17] Due to space limitations, we do not illustrate it, but it is similar to the previous case.

7.3 Discussion

The CI/CD pipeline example presented by Cherepanov and Turner is much more complex than the five example pipelines provided by CircleCI we used in Section 6. In spite of this, our meta-model supported the lengthy migration process to GitHub Actions described in the article. This real-world case study shows our work has practical applications and benefits. A future version, with all required M2M transformations properly defined, should be able to speed up migration significantly, a process that can take as much as five weeks when done manually [4]. This has the potential to increase productivity and reduce lock-in to CI/CD technologies.

8 THREATS TO VALIDITY

Our study faces several threats to validity, which we now address.

Our meta-model was constructed based on some of the most commonly used CI/CD technologies. It is conceivable that pipelines constructed using less commonly used technologies may not be fully represented using our meta-model. Nevertheless, the technologies utilized in our study encompass 96.3% of existing CI/CD pipelines [8].

While validating the usefulness of our meta-model to represent real-world pipelines, we primarily focused on parsing GitHub Actions configuration files, a language that influenced our meta-model. It could be the case that parsing other languages into valid models would not be possible. However, we have also parsed CircleCI scripts successfully. Thus, although we cannot guarantee this for different technologies, we can already cover a significant set of existing scripts (more than 61% [8]).

The syntactic validation faces similar limitations, that is, we cannot ensure scripts generated from our models with other CI/CD as targets will be syntactically correct. Nevertheless, there is also no evidence that this would happen.

When assessing whether semantic equivalence would be preserved during migration between CI/CD technologies using our approach, we relied on five examples provided in the CircleCI documentation. While not real-world test cases, these examples cover most of the constructs found in pipelines. Moreover, they include code repositories, which is fundamental to being able to execute the scripts and compare them, something we did we success.

9 CONCLUSIONS

We have developed a meta-model for CI/CD pipelines to support their creation and simplify migration across different technologies. Validations showed a 96.5% success rate in parsing and representing GitHub Actions files, confirmed syntactical correctness for each possible model confirming to the meta-model (apart from object repetition), and maintained model equivalence in real-world migrations, both for learning examples and a real-world case study.

Future efforts will focus on further integrating the meta-model into visual tools for easier creation and manipulation, as well as providing tool support for automating the migration process for increased efficiency.

REFERENCES

[1] Author Anonymous. 2024. *A Meta-Model to Support the Migration and Evolution of CI/CD Pipelines.* https://doi.org/10.5281/zenodo.10895301

[2] Zia Babar, Alexei Lapouchnian, and Eric Yu. 2015. Modeling DevOps Deployment Choices Using Process Architecture Design Dimensions. In *The Practice of Enterprise Modeling* (Cham) *(Lecture Notes in Business Information Processing)*, Jolita Ralyté, Sergio España, and Óscar Pastor (Eds.). Springer International Publishing, New York, New York, USA, 322–337. https://doi.org/10.1007/978-3-319-25897-3_21

[3] Francis Bordeleau, Jordi Cabot, Juergen Dingel, Bassem S. Rabil, and Patrick Renaud. 2020. 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*, Jean-Michel Bruel, Manuel Mazzara, and Bertrand Meyer (Eds.). Springer International Publishing, Cham, 139–151.

[4] CircleCI. 2024. *Introduction to CircleCI migration - CircleCI*. Circle Internet Services. https://circleci.com/docs/migration-intro/

[5] Alessandro Colantoni, Luca Berardinelli, Antonio Garmendia, and Johannes Bräuer. 2022. 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* (Montreal, Quebec, Canada) *(MODELS '22)*. Association for Computing Machinery, New York, NY, USA, 784–792. https://doi.org/10.1145/3550356.3561597

[6] Alessandro Colantoni, Luca Berardinelli, and Manuel Wimmer. 2020. 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* (Virtual Event, Canada) *(MODELS '20)*. Association for Computing Machinery, New York, NY, USA, Article 69, 10 pages. https://doi.org/10.1145/3417990.3420203

[7] The Business Research Company. 2022. Information Technology Global Market Report 2022, By Type, Organization Size, End User Industry. https://www.researchandmarkets.com/reports/5561700/information-technology-global-market-report-2022

[8] Hugo da Gião, André Flores, Rui Pereira, and Jácome Cunha. 2024. Chronicles of CI/CD: A Deep Dive into its Usage Over Time. arXiv:2402.17588 [cs.SE]

[9] Thomas F. Düllmann, Oliver Kabierschke, and André van Hoorn. 2021. 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)*. 214–223. https://doi.org/10.1109/SEAA53835.2021.00035

[10] Dawit Gebregziabher. 2023. GitHub Actions Importer is now generally available. https://github.blog/2023-03-01-github-actions-importer-is-now-generally-available/

[11] G. Kim, J. Humble, P. Debois, and J. Willis. 2016. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution Press, Portland, Oregon, United States. https://books.google.pt/books?id=ui8hDgAAQBAJ

[12] Corinne Pulgar. 2022. 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* (Montreal, Quebec, Canada) *(MODELS '22)*. Association for Computing Machinery, New York, NY, USA, 225–228. https://doi.org/10.1145/3550356.3552395

[13] Luis F. Rivera, Norha M. Villegas, Gabriel Tamura, Miguel Jiménez, and Hausi A. Müller. 2018. UML-driven automated software deployment. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering* (Markham, Ontario, Canada) *(CASCON '18)*. IBM Corp., USA, 257–268.

[14] Michael Wurster, Uwe Breitenbücher, Michael Falkenthal, Christoph Krieger, Frank Leymann, Karoline Saatkamp, and Jacopo Soldani. 2020. The essential deployment metamodel: a systematic review of deployment automation technologies. *SICS Software-Intensive Cyber-Physical Systems* 35, 1 (2020), 63–75. https://doi.org/10.1007/s00450-019-00412-x

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009