Electronic Communications of the EASST
Volume 84 Year 2025

12th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation / 2nd AISoLA - Doctoral Symposium, 2024

Edited by: Sven Jörges, Salim Saay, Steven Smyth

Towards X-as-Models in the Context of CI/CD

Sebastian Teumert, Tim Tegeler

DOI: 10.14279/eceasst.v84.2680

Towards X-as-Models in the Context of CI/CD

Sebastian Teumert[1], **Tim Tegeler**[2]

teumert.sebastian@ul.ie
Department of Computer Science and Lero
University of Limerick (UL), Limerick, Ireland[1]

adesso SE, Dortmund, Germany[2]

Abstract: The X-as-Code approach has been highly successful over the past few decades, enabling reliably reproducible outcomes and the automation of a wide range of workflows. However, it still relies on textual configurations that demand in-depth knowledge of the system's conventions and extensive documentation to use effectively. In this paper, we present X-as-Models as an alternative paradigm that offers the same advantages of reproduce-ability and automation, but allows domain experts to configure the workflow in an intuitive, low-code/no-code way by simply manipulating self-documenting graphical models. We propose a shift in CI/CD practice towards an X-as-Models paradigm, motivated by a survey of existing solutions and their shortcomings. In particular, we highlight how the absence of a formal foundation has led to round-trip engineering becoming the de-facto approach for developing CI/CD workflows. We outline key aspects we see as essential for advancing the state of the art and present a plan to address them through a new generation of low-code/no-code tools grounded in the principles of Language-driven Engineering (LDE).

Keywords: X-as-Code, X-as-Models, Language-Driven Engineering, Model-driven Engineering, CI/CD, Continuous Integration, Continuous Deployment, Purpose-specific Languages, Domain-specific Languages, Graphical Modeling, Visualization, Formal Verification

1 Motivation

Practicing Continuous Integration and Deployment (CI/CD) is the centerpiece of a mature DevOps process [SBN] and reduces the time-to-market of software systems by automating manual tasks during development and deployment. According to *Stack Overflow's 2023 Developer Survey* [StO] most professional developers report having CI/CD available at work. Thus, it has finally become a widely-used technology to automate all steps (e.g. packing and deploying) from source code to a running production system. Figure 1 shows a schematic overview of CI/CD.

However, most CI/CD solutions follow the X-as-Code approach (e.g. Infrastructure-as-Code) and are still based on textual configuration files written in data serialization languages like YAML. Despite general benefits of X-as-Code, like consistency and scalability, basing the implementation on general-purpose data serialization languages brings unnecessary drawbacks. Such languages claim to be designed with human-readability in mind, but practice and stud-

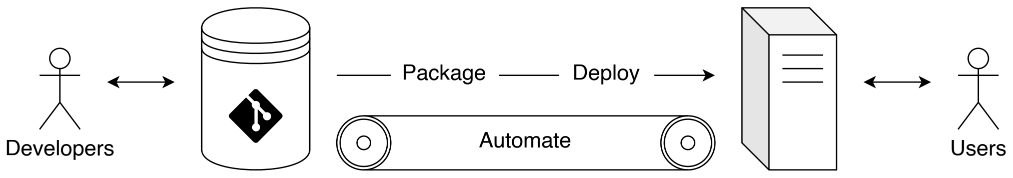

Figure 1: Schematic overview over a typical CI/CD process (as shown in [ST21])

ies [TTS+22] indicate that editing YAML is cumbersome and error-prone, due to indentation flaws and unintuitive parsing behavior (e.g. Norway Problem). As a consequence, well-trained experts in this field are required to create and maintain X-as-Code, which increases the probability of knowledge fragmentation.

At the time of writing, more than 70% of the developers have encountered knowledge silos at work at least once a week [StO]. The term knowledge silo describes the fragmentation or isolation of knowledge within an organization and is often associated with a negative impact on it. Bento et al. [BTL20] provide an overview over the different characterizations of knowledge silos. Relevant in the context of this paper is that knowledge silos are often characterized as groups of people with limited interaction with other groups, or having barriers to communication to effectively distribute their knowledge.

We have previously identified CI/CD workflows as being directed, acyclic graphs that lend themselves well to being represented graphically [TGS19] or as a model, rather than as a plain text file. X-as-Code as textual format necessarily requires programmers to write their workflows linearly, requiring a substantial mental load to identify and remember the structure of the workflow while reading or, more importantly, editing the configuration file.

In this text, we use the term *configuration* to mean the (textual or graphical) representation of the CI/CD workflow, while we use the term *pipeline* to mean one concrete instantiation and subsequent execution of the CI/CD workflow, and the term *workflow* to mean the holistic description of all possible pipeline instantiations as laid out in [TTS+22].

Furthermore, the lack of validation leads to so-called round-trip engineering (RTE), where changes to the CI/CD workflow are developed and then have to be put into the actual production repository for testing, and when these changes are found to be not working, a new iteration begins. This is a slow, error-prone way to work that pollutes the commit history and is not based on solid engineering principles but is a common approach to dealing with CI/CD configurations.

Our project *Rig* aims to resolve the aforementioned disadvantages of modern CI/CD solutions, by introducing a graphical Domain-Specific Language (DSL) for CI/CD workflows to automatically generate syntactically correct YAML code from which pipelines can be executed. Rig's DSL and code generator are bundled as an Integrated Modeling Environment (IME) (an IDE specifically enhanced with modeling capabilities). The graphical aspect lowers the barrier of entry and improves communication; thus, it may help to reduce knowledge silos. Rig can be a promising contribution to the transition of DevOps from X-as-Code towards X-as-Models.

2 State of the Art

In this section, we systematically review the state-of-the-art in modeling CI/CD workflows, especially with respect to the visualization and graphical authoring aspects, as well as validation. Other minor aspects include the collaboration between users, especially domain-experts and non-experts, as well as their compositionality, which is a key aspect for driving reuse, and thus knowledge transfer. We also review the current approaches to X-as-Models and how we aim to contribute to that field.

2.1 Continuous Integration and Deployment

While CI/CD is a novel topic for computer science research, it already has become a de-facto standard in modern software engineering. According to JetBrains' *The State of Developer Ecosystem 2022 Survey* [jet], the most used CI/CD systems—with at least 10% popularity among the surveyed developers—are GitHub, Jenkins, GitLab, Azure DevOps, CircleCI, Travis, BitBucket, and AWS CodeStar.

In order to illustrate the state-of-the-art of CI/CD, we analyzed these existing systems along the five categories Visualization, Format, Validation, Collaboration, and Composition (see Table 1). The selection of these five categories results from the refinement of initial plans for future work on Rig, as outlined in [TTS+22]. While possibly biased towards Rig, we see them as initial basis for future discussions in the area of model-driven CI/CD systems.

Visualization comprises how executed pipelines are displayed in the system's web view (**VD1–2**), and which graphical aids are provided by the native editor (**VE1–2**). We consider an appropriate visualization key for aligning the understanding of involved stakeholders and reducing knowledge fragmentation [TBS+22]. Another distinctive feature of CI/CD systems is the underlying paradigm. A chosen paradigm and it's level of abstraction can affect the reasoning about a given workflow. "Format" describes the paradigm used for the persisted workflow (**FMT**). **VAL1–3** concern which type of validation capabilities are provided. Proper validation can reduce trial-and-error roundtrips, known when creating workflows from scratch [TGS19]. Collaboration features are assessed in **COL**. Successfully adopting CI/CD requires cross-cutting knowledge [Teg23] and thus collaboratively maintaining involved workflows is important. Composition (**CMP1–3**) includes the reusablitity and parametrization of workflows. Adequate techniques for composition can help to avoid knowledge duplication [HT99].

In addition to the aforementioned popular systems, we include three more systems:
SemaphoreCI, which emphasize its visual capabilities or UI in the marketing, Dagger as a newly emerging solution for CI/CD workflows and our own work, Rig.

In our scoring, we give a maximum of one point (using •) for each property shown in Table 1, if the implementation covers all the major aspects and provides a satisfactory experience. For partial implementations, we gave half points (using ◦) and for missing or barely usable features, we gave no points (using $\times$). The evaluation was based on available documentation and first-hand assessment of the CI/CD systems. Similar to the chosen categories, the scoring is a preliminary ranking by the authors and we invite it to be challenged by future research.

Since we had eleven properties to evaluate, this means that the maximum number of achievable points is eleven. The highest scoring commercially available platforms were GitLab, CircleCI,

Item	Category	Subject	Description
VD1	Visualization (Display)	Workflow	Workflow is visualized completely
VD2		Pipeline	Concrete pipeline runs are visualized
VE1	Visualization (Editor)	Major blocks	Graphical editing and composition of the major blocks into a DAG
VE2		Auxiliary Elements	Auxiliary elements (Caches, Environments etc.) are editable graphically
FMT	Format	Paradigm	*D* Declarative *I* Imperative *M* Model
VAL1	Validation	Document Syntax	Syntax of the document itself (e.g. YAML syntax) is validated
VAL2		Auxiliary Syntax	Syntax of auxiliary grammars (embedded as strings into the document) is validated
VAL3		Consistency (Semantic)	Semantic validity and consistency of the modeled workflow is validated
COL	Collaboration	Mode of Work	● Live Collaboration ○ Merging via VCS
CMP1	Composition	Job (Step, Block)	Can be re -used and (re-)parameterized
CMP2		Pipeline	Can be re -used in another pipeline or triggered as subpipeline
CMP3		Auxiliary Elements	Auxiliary elements can be re-used and (re-)parameterized

Table 1: Properties used to evaluate the CI/CD platforms

and Dagger, with five points each. Our own implementation, Rig, scored slightly better with a score of 6. We believe that a unified CI/CD model that has a formal foundation should be able to achieve most, if not all, points in these categories while being general enough to support all platforms as generation target.

The data we gathered (see Table 2) shows that there are severe shortcomings found in the approaches of all major CI/CD platform when it comes to validation. Furthermore, all platforms treat collaboration and visualization as secondary concerns, if they at all address them. Composition of workflows from smaller elements is often an afterthought and implemented ad-hoc as necessary band-aid fixes to scalability problems, instead of being addressed as a first-level concern.

While none of the surveyed platforms use a model-based approach, some platforms have started visualizing the workflows or instantiated pipeline as graph. Of note are here GitLab and SemaphoreCI, who are the only major platforms that feature an editor that visualizes the complete workflow while editing it. Figure 2 shows the visualization of an executed CI/CD pipeline in GitLab. However, editing is still done textual (GitLab) or through forms in the UI (SemaphoreCI) and there is no interaction with the graph directly, and the visualization is limited

	Visualization						
Platform	**Display**	**Editor**	**Format**	**Validation**	**Collaboration**	**Composition**	**Total**
GitHub	× ○	× ×	D	● × ×	○	● ○ ×	3.5
Jenkins	× ×	× ×	I	× × ×	○	● ○ ○	2.5
GitLab CI	○ ○	○ ×	D	● × ○	○	● ○ ×	5.0
Azure DevOps	× ○	× ×	D	● × ×	○	● ○ ×	3.5
CircleCI	× ○	× ×	D	● ○ ●	○	● ○ ○	5.0
Travis	× ×	× ×	D	● ○ ×	○	○ × ×	2.5
BitBucket	× ○	× ×	D	● × ×	○	○ ○ ×	3.0
AWS CodeStar	× ×	× ×	UI	○ ○ ×	×	● × ×	2.0
SemaphoreCI	○ ○	○ ×	D	● × ×	○	○ ○ ×	4.0
Dagger	× ×	× ×	D	● ○ ×	○	● ● ●	5.0
Rig	● ×	● ●	M	● ○ ×	×	● × ○	6.0

Table 2: Overview over the current capabilities of major CI/CD solutions

to the major building blocks, not including auxiliary aspects like caches, artifact passing, deploy environments and other aspects. A majority of platforms nowadays visualize the running (or finished) pipelines, but again usually only the major building blocks.

When it comes to validation, the built-in editors of most platforms are able to check the document syntax itself, which is commonly YAML. However, auxiliary grammars (e.g. conditions) are often treated as bare strings and not further validated. Also, many platforms fail to properly validate logical errors, e.g. jobs missing their prerequisites in certain situations. Depending on the modeling chosen, some of these problems are prevented by design (e.g. BitBucket, CircleCI). Of note here is CircleCI, which offers a separate CLI validator that performs logical and structural checks to validate the workflow and all configured pipelines.

Collaboration is often not explicitly handled by the platform, and instead relies on textual merging of configuration files. This poses a problem for model-based approaches and UI-based approaches like AWS CodeStar.

In terms of composition, we found that almost all platform enable some form of re-use for the major building blocks of pipelines, with different levels of flexibility of parameterization and ease-of-use. Ranging from bare YAML anchors (BitBucket) over imports of pre-defined actions (GitHub) or templating (e.g. GitLab). However, re-using whole pipelines, just parts of the workflow or grouping of the major building blocks are often cumbersome. Some platforms allow calling of sub-pipelines, with varying degrees of parameterizability.

The re-use of auxiliary elements is often not part of the design. A notable outlier here is Dagger, which does not use declarative syntax but code, and thus can leverage the programming languages capabilities for modularization and re-use.

In summary, our review suggests that there is a need for better validation of workflows, a more robust model for modularization and re-use that leads to easier composition of workflows and a

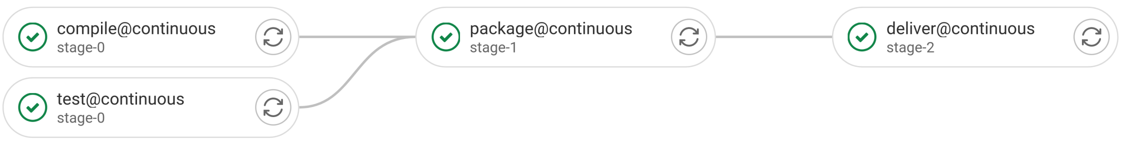

Figure 2: Visualization of an executed CI/CD pipeline with two concurrent jobs and two subsequent jobs that are dependent on each other, as shown by GitLab

better approach to visualization to better document the CI/CD process of the given project and to facilitate easier alignment between team members. We believe that a model-driven approach, especially a model-driven *graphical* approach can be leveraged successfully in order to advance the field significantly.

2.2 X-as-Models

In the context of this paper, we understand X-As-Models as an umbrella term for approaches to ease the adoption of DevOps-related practices through textual and graphical models. The following list of publications gives a summary of recent research in this domain:

- The presented approach in [CHH$^+$21] aims at transforming textual DevOps artifacts, independent of the DevOps lifecycle phase, into models back and forth for consistency checking.
- The DICER tool [ABD$^+$16] provides a model-driven framework to support the development and operations following DevOps practices. It combines a modeling environment with a deployment service in the DICER IDE to design and deploy data-intensive application.
- The ArgoCD tool [1] is a graphical editor for Kubernetes-based deployments. Their approach is focused more on the deployment and operation (GitOps) of the application and targets Kubernetes specifically. It highlights the importance of also supporting pull-based workflows, which have been credited with improving reliability and reducing manual intervention compared to push-based methods [Per24], but is not platform-agnostic.
- A family of modeling languages for developing and operating service-oriented applications is presented in [RSSZ19]. Their approach targets different roles in a DevOps team with interconnected languages for domain-, service-, and operation model.
- DevOpsML [CBW20] is a framework built on Ecore and the Eclipse Modeling Framework [SBPM08] to apply MD* for DevOps. It aims at interconnecting model-based software engineering processes with platforms that provide fundamental functions of the DevOps practice. At the time of writing, DevOpsML aims at documentation only and perceives model execution as future work.

[1] https://argoproj.github.io/cd/; accessed 2025-03-12 (archived).

- In their paper [SWR+21] the authors apply a model-driven workflow, based on their *Language Ecosystem for Modeling Microservice Architecture*. They target two main challenges when adopting DevOps for microservices architecture in small and medium organizations, which they extract from a previous study: Maintaining a common architectural understanding, and handling the complexity of deploying and operating microservice application.

All the aforementioned publications have in common that they focus on the broader topic DevOps. We think in order to successfully implement X-as-Models in practice and to mitigate a big bang adoption [BLW+97], we need to focus on specific problems, like CI/CD. To the best of our knowledge only two publications in this domain go in the same direction:

- StalkCD [DKH21] is a model-driven framework for interoperable CI/CD pipeline definitions. It aims at parsing different CI/CD configuration formats (e.g. Jenkins, GitLab) into in their StalkCD DSL. Based on this uniform intermediate representation the presented framework allows the analysis and transformation to other CI/CD configuration formats or even to the BPMN (cf. [All09]). StalkCD does not support modeling a graphical DSLs, but solely visualizing CI/CD configurations as BPMN.

- Da Gião et al. [GPC23] describe their vision for using a block-based language, similar to Scratch [MRR+10], for expressing CI/CD pipelines in visual block components. Although their language is an interesting approach to graphical modeling for pipelines, we believe using a graph-based language is the more naturally approach for a dataflow use-case like CI/CD.

We believe our approach fundamentally differentiates itself from current research in the domain of X-As-Code for DevOps by the following concepts.

Standardization: Since practicing CI/CD has become de-facto standard for professional software engineering, we believe it is time to consolidate current approaches and propose a unified standard.

Validation: Leveraging the domain knowledge of CI/CD and the archimedean points of the target plattform (e.g. GitLab) we can apply validation of concrete model instances and give the modeler feedback of the internal consistency of the designed workflow.

Reusability & Composition: Workflows in practice often follow resembling patterns (e.g. test $\rightarrow$ build $\rightarrow$ deploy) and use similar definitions for certain jobs. Our approach treats reusability and composition as first-class citizens in order to comply with the *don't repeat yourself* principle.

Visualizing: Adapting the domain-specific notations of CI/CD (see Figure 2) by turning them into a graphical model syntax allows us to enable workflow visualization that goes beyond the current visualization of CI/CD systems.

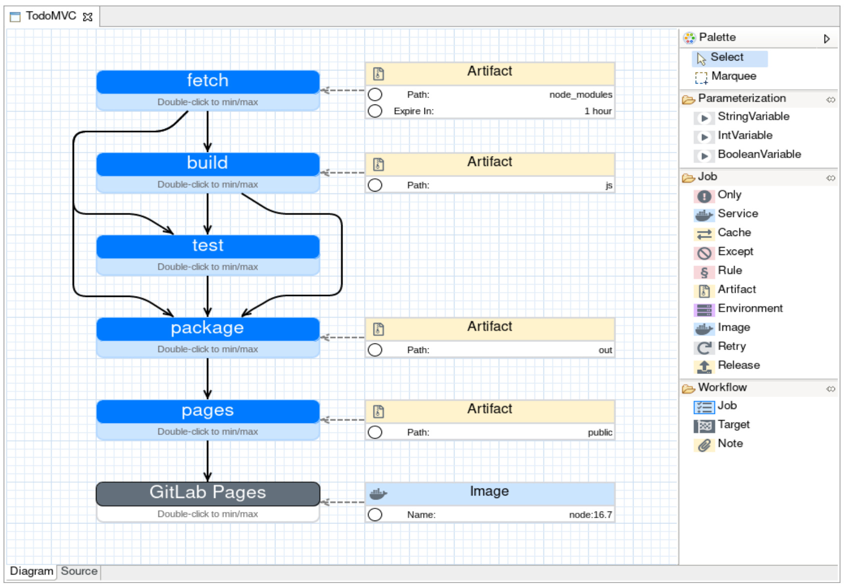

Figure 3: Exemplary model of Rig's graphical DSL for CI/CD (reprint [TTS+21]). Solid edges between jobs represent dependencies; artifacts are implicitly passed between dependent jobs. The dashed edges represent configurations applied to a job, in this case, the artifacts to be emitted and the container image to be used. Configurations applied to the target (bottom) are applied to all preceding jobs. The complete palette is shown to the right.

Collaboration: Modern tools should be designed with native support for collaboration from the very beginning. Our approach aims at going beyond the traditional application of leveraging version control systems to collaborative work on text-based files.

Code Generation: A model does not end in itself or is just used for visualizing something, but is inherent executable via code generation. Thus, models are instances of fully-fledged modeling/programming languages [TBS+22].

3 Preliminary Work

Rig is an IME and graphical language for modeling CI/CD workflows. [2] Currently, it is based on the Eclipse Rich Client Platform. See Figure 3 for an exemplary CI/CD workflow modeled in Rig.

Rig is different to the aforementioned platforms in a way that Rig uses a top-down approach where the models/visualization comes first and the source code comes second, while the platforms follow a bottom-up approach and treat the source-code as first-class citizens and the visualization is downstream.

[2] Its name originates from the noun *rigging*, i.e. "lines and chains used aboard a ship especially in working sail and supporting masts and spars" and the verb *to rig*, "to put in condition or position for use".

The preliminary work on Rig was published in the following five publications:

A Model-driven Approach to Continuous Practices for Modern Cloud-based Web Applications: [TGS19] presents our initial work on Rig and introduces the concept of a graphical language for Continuous Integration and Deployment workflows. The presented approach comprises the full code generation of textual configurations and allows reuse through parametrizing of predefined model elements, reducing the error-proneness by a strict model syntax, and enables advanced verification features like model-checking.

Visual Authoring of CI/CD Pipeline Configurations: [Teu21] expands on the approach published in [TGS19] and presents the first fully fledged implementation of Rig.

An Introduction to Graphical Modeling of CI/CD Workflows with Rig: [TTS+21] illustrates how CI/CD workflows can be modelled Rig using its graphical language. This introductory paper laid the foundation of a workshop focussing an international group of Ph.D. students at the *6th International School on Tool-Based Rigorous Engineering of Software Systems* [MS21].

Executable Documentation: From Documentation Languages to Purpose-Specific Languages: [TBS+22] shows how the domain-specific notation of graphical languages can be turned into fully-fledged modeling languages. The approach is demonstrated alongside GitLab's visualization for documenting executed CI/CD workflows and Rig's graphical language.

Evaluation of Graphical Modeling of CI/CD Workflows with Rig: [TTS+22] evaluates our findings based on pipeline data gathered during the aforementioned workshop at the *6th International School on Tool-Based Rigorous Engineering of Software Systems* [MS21]. The data leads to the conclusion that graphical modeling of CI/CD workflows has the potential to reduce trial-and-error, eliminate syntax errors and lower the entry barrier, when compared with textual DSLs.

4 Towards a Core Semantic for CI/CD

X-as-Code has become a popular way to automate a variety of fields or domains, while supporting versioning via source code management. From *Infrastructure-as-Code* descriptions like Helm, which are ubiquitous in the cloud computing world, over dependency management to the subject of this proposal, *Pipelines-as-Code*. However, as laid out before, textual representations might not be the optimal solution for all data structures, especially data structures that represent graphs. CI/CD workflows typically are directed-acyclic graphs and need to be written out in a linear fashion when represented in textual form. Ideally, code should be easy to read and understand, because ”code is more often read than written” (usually attributed to Guido von Rossum, Inventor of Python).

However, linear representations of DAG fail in this regard, and become hard to read, reason about and manipulate. In the initial version of Rig, we therefore introduced a rudimentary graphical way to represent CI/CD workflows and have shown that the problem is tractable with a graphical language (cf. Section 3). However, our previous work lacks a solid formal foundation and universality. It is specific to one CI/CD platform, in whose context it works well, but does not present a general or universal model for CI/CD workflows. The remainder of this section elaborates on the conceptual foundation for the next version of Rig.

We argue that a new approach is needed and envision a formal semantic and unified (graphical) model, as well as a proof of concept implementation as one of the first comprehensive tools that introduces a new way of thinking about and handling graphical data: *X-as-Model(s)*. The *X-as-Model(s)* paradigm is meant to introduce the same advantages as *X-as-Code*, but goes a step further and does not require users to serialize their ideas as textual representation but instead working directly with an intuitive, easy to read, understand and manipulate representation of the workflow data.

We are convinced that X-as-Models is a promising paradigm for this domain and want to apply it here to both validate the X-as-Models approach, as well as to advance the field of CI/CD by leveraging the created meta-model as foundation for formal validation of workflows and possible instantiations (as pipelines), as well as showing that the model itself can be *executable* and *editable* and serve as the single source of truth—as is required in the *One Thing Approach* (OTA, [MS09])—greatly easing the aspect of writing and maintaining documentation as well as collaboration between the different stakeholders of a project, who might not all work on the same aspect (especially for large mono-repositories), library or program of the project.

Language-driven Engineering (LDE) [SGNM19, SMB$^+$24] enriches domain languages—often graphical—with additional properties and features to enable full code generation in support of the One-Thing Approach (OTA). A central tenet of LDE is the composition of multiple Purpose-specific Languages (PSLs) to achieve broader objectives. In the context of CI/CD, this implies that a workflow model should be stable in its core structure (“closed for modification”) while remaining adaptable through the integration of additional PSLs (“open for extension”). For example, a PSL describing deployment environments could be combined with a CI/CD workflow model to ensure that generated artifacts match the requirements of a specific target platform. Such integrations could also enable formal verification and analysis of both the workflow and its outputs, such as checking compatibility with runtime constraints before deployment. Given these capabilities, LDE and the OTA provide a strong foundation for rethinking X-as-Code approaches to CI/CD.

4.1 Unification

In order to achieve the high level goals stated above, we propose a formal approach to CI/CD: Analyzing meta-models of current industry solutions (and, since for many industrial solutions, no explicit meta-model is given, inferring those meta-models in the first place) and formally defining a unified view of CI/CD. The unified model needs to be open and adaptive to allow for specialization, while at the same time general enough to allow most workflows to be described out-of-the-box. A Rig model should serve as the *ultimate source of truth* (in line with the OTA approach) and be the only model or artifact a programmer needs to deal with in order to describe their workflow.

Having a unified, formal model of a CI/CD workflow then serves as main driver for all of the other goals. A CI/CD workflow that follows a standardized model can be validated for internal consistency (logic) and external consistency (taking the external circumstances, e.g. concrete repository contents, into account) and can employ model-based approaches to composition, greatly increasing re-usability and cutting down copy & paste editing. Being a model, especially a model representing a directed, acyclic graph, it can be directly designed as a graphical model with the accompanying graphical editor that allows direct, graphical manipulation as well as serving as *executable documentation*, given the proper interpreter or code generator.

We have also noticed that there are several high-level paradigms that are used by different CI/CD platforms. Historically, a popular way to model CI/CD pipelines has been to model them as subsequent *stages*, with the possibility of executing jobs inside the same stage in parallel. This is a quite rigid model that has since largely fallen out of fashion for more modern, usually graph-based approaches. But while some providers allow only simple graphs that alternate between fan-in and fan-out structures, some allow full directed, acyclic graphs. Yet another difference is where constraints on the execution of jobs are placed. While some platforms like GitLab allow for arbitrary constraints placed at arbitrary places of the workflow, other platforms like BitBucket model the workflows as distinct collections of pipelines, all with a singular trigger at the start, reminiscent of Event-Condition-Action (ECA, [BM09]) systems.

However, we are optimistic that these approaches can be unified under a single, standardized description from which configurations that are compatible with each of these paradigms can be automatically derived. For example, conditions can always be moved to the front of a workflow, if one is willing to generate more paths. Similarly, the different branching strategies used by various platform can always be reduced to a DAG and converted into each other by re-ordering. We also envision that this standardization closes the gap between building locally on the machine of the developer and building inside a CI/CD pipeline. Having a unified model allows the local interpretation of the workflow and instantiation of a local pipeline, or alternatively the generation of single shell script suitable for local execution.

4.2 Validation

The (in-) validity of a configuration can be traced back to two fundamental aspects: The syntactic correctness of the configuration file itself, and the semantic validity of the workflow contained therein. Whereby the syntactic correctness can be achieved by employing an IME with correctness-by-construction guarantees, the semantic correctness can be approached from two different perspectives.

From the perspective of static analysis, the internal consistency of the workflow can be validated, for example by ensuring that the prerequisites for all workflow elements are met. In order for this to work, these prerequisites need to be expressed in machine-readable way. At the same time, if an element satisfies a prerequisite for another element, this also needs to be encoded in a machine-readable way. We envision to develop a standardized language to express these *requirements* and *capabilities* in order to devise automated checks to ensure they consistently hold. This can also be a leveraged as part of input aids for the user, highlighting as of yet unmet requirements and which elements are available to fulfil them.

The external consistency of a workflow can only be analyzed in the concrete context it will later be instantiated in. The external context consists of the repository contents, environment variables and operating system, among others. Validating the workflow in a concrete environment / external context is only possible if the external context is known with enough accuracy and if the requirements that the workflow places on the external context are known as well. Thus, these external requirements need also to be encoded in a machine-testable way.

As an example, imagine an application that consists of two applications: A frontend application (SPA) that is compiled with Node JS (Version 16 or higher) and a backend application copiled with Java 21 or higher and Maven 3.9 or higher. These are requirements that need to be fulfilled either internally or externally. Possibilities to internally satisfy these requirements include running these jobs on the appropriate Docker images (`node:16` and `maven:3.9-[..]`, respectively). Furthermore, the job building the backend might explicitly declare its dependency on a well-formed `pom.xml` placed in the directory it is supposed to be executed in. In this area, the major reserach contribution is finding an appropriate abstraction level and language to express those constraints in, as well as strategies for validating them. Another contribution lies in finding an appropriate scope of what can and should be validated.

Encoding these requirements and capabilities, as well as the collection of the external context and standardization of common elements are key to allow proper validation to occur. Local execution of the workflow, together with validation, can eliminate the need reound-trip-engineering, which takes a long time and pollutes the commit history of the affected repository, moving the field away from trial & error programming to more solid work regime based on engineering principles. Validation and local execution also open up the possibility of creating automated tests of the pipeline, both as unit-test as well as integration-tests of the build process itself.

4.3 Reusability & Composition

In our view, re-usability and composability are two features that go hand-in-hand. Models and model elements that are highly re-usable lend themselves well for composition.

As we have laid out before, re-use and composition in current industrial CI/CD solutions are often done in an ad-hoc way (cf. Section 2). In our previous work, jobs were defined and then directly used at the place where where they were defined. Re-use was facilitated by allowing multiple targets to parameterize a single job, yielding multiple executions of one job. However, re-use of jobs across multiple models and sharing of jobs across organizations has not been achieved.

For this work, we envision a re-use and composition model that has been designed from first principles, directly with these aspects in mind. We consider the separation of job declarations and their usage a cornerstone of this approach. Where possible, this should also extend to auxiliary aspects. A substantial research contribution in this area should be the identification of appropriate levels of granularity as well as identifying which elements can be re-used in which way. At a minimum, jobs as well as whole or partial workflows should be declarable as re-usable blocks and shareable in a library from which they can be imported and then used in another workflow. Multiple jobs (partial workflows) should be group-able into a single re-usable building block. We are confident that modeling CI/CD workflows in a data-flow like fashion is an appropriate model that captures the required parallelism and the semantics of CI/CD workflows well.

Another research contribution will be the code generation and re-composition of the workflow into platform-appropriate workflows. On some platforms it might be appropriate to combine multiple similar jobs into one job on the target platform, while on others it might be more appropriate to keep them separate. Research into the appropriate level of granularity and possibilities of user guidance in this process is important to yield appropriate executions on a variety of platforms.

This is also a performance aspect, since not all platforms allow for complete directed acyclic graphs, but only a fan-in, fan-out model. In such a model, grouping jobs into parallel executions of similar execution time might drastically improve overall execution times through better scheduling. In this area, we envision to also allow instrumentation of workflows to collect telemetry and to employ machine-learning techniques to improve on the chosen scheduling.

4.4 Visualization

Since we plan to work in a model-first approach and want to demonstrate the viability of X-as-Models, the proposed solution will be a (graphical) model definition that lends itself very well to being visualized as a graph. By emplyoing LDE concepts, we want this model to be fully interactable. Like our previous work, basing this on the CINCO Metamodeling Framework is a viable option, but we also want to explore alternate options to produce a fully-fledged, interactive visual editor that can then be empirically validated through user testing.

A focus should be to elevate every model aspect to a truly interactable element and avoiding the hiding of auxiliary information inside UI panels. Clarity and completeness of the visualization should be primary concerns.

At the same time, large workflows can become hard to read, so we plan to include hierachical structures that can be used to zoom out or zoom in to the model at the desired granularity. This will keep the complexity of the visualized graph low, while allowing to locally focus attention on interesting parts of the model. This should also serve as important driver for using the model as *executable documentation*, where different parts of the documentation can focus on different levels, e.g. giving a global overview, then stepping into more localized areas of the workflow and documenting solutions in more detail.

A formal definition of the hierachy, collapsing and exploding behavior should form the basis of this work, distinguishing it from other industrial ad-hoc solutions. Space should be given to developing heuristics for the automatted collapsing and exploding to create concise and redabale models at all levels.

4.5 Collaboration

Collaboration on textual configurations is solvable by textual merges using a VCS of ones choice (e.g. git). This is how collaboration currently works in the overwhelming majority of cases. However, merging *models* is hard, as their textual representation might change drastically with only slight changes to the actual model. There is ongoing work by Jonas Schürmann into a so-called technique of ”lazy merging”. [SS22] This would allow both asynchronous as well as live-collaboration. This is a promising concept we would like to explore for our tool as well.

Less powerful modes of live collaboration already exists within modern web-based IDEs like VSCode and Theia. We will also investigate if we can leverage these modes for our work.

4.6 Code Generation

We plan to carry on Rig’s initial approach to full code generation [TTS+21]. The full code generation approach [KT08] lets maintainers overwrite existing CI/CD configurations without worrying about syntactic or semantic changes. It guarantees that the generated CI/CD configuration file is compatible with the target CI/CD engine. However, in contrast to the initial version of Rig, we plan to extend the scope of our code generator. By leveraging the unifying character of the above-mentioned unified/standardized CI/CD model, we will be able to support more than a single CI/CD engine (i.e. GitLab) as the generation target.

The code generation process will be split in two consecutive phases. First, the concrete model will be transformed to a CI/CD engine-specific intermediate representation that is fitted for the actual code generation phase. Second, the instance of the intermediate representation will be used in order to generate the textual configuration file. Since most of those configuration files are based on data-serialization languages, we can utilize robust libraries (e.g. Jackson) for the serialization of syntactically correct JSON or YAML files.

5 Conclusion

We have examined CI/CD—specifically, the creation and maintenance of working workflows—as a promising candidate for raising the abstraction level from X-as-Code to X-as-Models. As discussed in Section 1 and Section 2, the field has evolved in an ad hoc manner without a formal foundation, making round-trip engineering the de facto approach for building workflows. Furthermore, we argue that graphical languages are a more intuitive representation of CI/CD workflows (which are DAGs) than a linear textual file that has to be read in sequence. Thus, a graphical workflow can be thought of as self-documenting or at least easier to be documented by auxiliary documentation and can be used as communication tool between all involved stakeholders, improving collaboration, reducing knowledge fragmentation, and opening up knowledge silos. We have also highlighted how a unified CI/CD model can be used both for static, contextual, and runtime verification of CI/CD workflows, eliminating the need for RTE and moving the field to more robust software engineering practices. Our preliminary results (cf. Section 3) show that graphical DSLs may indeed be more approachable and intuitive and may lead to a lower entry barrier for non-experts into the field. In Section 4 we have described the goals and advantages of a unified core semantic for CI/CD and a unified graphical model to model such workflows. Taken together, this paper shows the opportunities for research in this area and lays out how the X-as-Models approach can be leveraged for advancement in this domain.

Acknowledgements: This work was conducted with the financial support of Science Foundation Ireland (SFI) under grant number 18/CRT/6223 574 (SFI Centre of Research Training in AI).

Bibliography

[ABD+16] M. Artač, T. Borovšak, E. Di Nitto, M. Guerriero, D. A. Tamburri. Model-Driven Continuous Deployment for Quality DevOps. In *Proceedings of the 2nd International Workshop on Quality-Aware DevOps*. QUDOS 2016, pp. 40—41. Association for Computing Machinery, New York, NY, USA, 2016.
doi:10.1145/2945408.2945417

[All09] T. Allweyer. *BPMN 2.0 - Business Process Model and Notation*. Books on Demand, 2009.

[BLW+97] J. Bisbal, D. Lawless, B. Wu, J. Grimson, V. Wade, R. Richardson, D. O'Sullivan. An Overview of Legacy Information System Migration. In *Proceedings of Joint 4th International Computer Science Conference and 4th Asia Pacific Software Engineering Conference*. Pp. 529–530. 1997.
doi:10.1109/APSEC.1997.640219

[BM09] M. Berndtsson, J. Mellin. *ECA Rules*. Pp. 959–960. Springer US, Boston, MA, 2009.
doi:10.1007/978-0-387-39940-9$_5$04

[BTL20] F. Bento, M. Tagliabue, F. Lorenzo. Organizational Silos: A Scoping Review Informed by a Behavioral Perspective on Systems and Networks. *Societies* 10(3), 2020.
doi:10.3390/soc10030056

[CBW20] A. Colantoni, L. Berardinelli, M. Wimmer. DevOpsML: Towards Modeling DevOps Processes and Platforms. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS '20. Association for Computing Machinery, New York, NY, USA, 10 2020.
doi:10.1145/3417990.3420203

[CHH+21] A. Colantoni, B. Horváth, Horváth, L. Berardinelli, M. Wimmer. Towards Continuous Consistency Checking of DevOps Artefacts. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. Pp. 449–453. 10 2021.
doi:10.1109/MODELS-C53483.2021.00069

[DKH21] T. F. Düllmann, O. Kabierschke, A. v. Hoorn. StalkCD: A Model-Driven Framework for Interoperability and Analysis of CI/CD Pipelines. In *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. Pp. 214–223. 9 2021.
doi:10.1109/SEAA53835.2021.00035

[GPC23] H. da Gião, R. Pereira, J. Cunha. CI/CD Meets Block-Based Languages. In *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Pp. 232–234. 2023.
doi:10.1109/VL-HCC57772.2023.00039

[HT99] A. Hunt, D. Thomas. *The Pragmatic Programmer*. Addison Wesley, 9 1999.

[jet] Team Tools - The State of Developer Ecosystem in 2022 Infographic — JetBrains: Developer Tools for Professionals and Teams. Online.
https://www.jetbrains.com/lp/devecosystem-2022/team-tools/#ci-tools

[KT08] S. Kelly, J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Press, Hoboken, NJ, USA, 2008.
doi:10.1002/9780470249260

[MRR+10] J. Maloney, M. Resnick, N. Rusk, B. Silverman, E. Eastmond. The Scratch Programming Language and Environment. *ACM Trans. Comput. Educ.* 10(4), 11 2010.
doi:10.1145/1868358.1868363
https://doi.org/10.1145/1868358.1868363

[MS09] T. Margaria, B. Steffen. Business Process Modelling in the jABC: The One-Thing-Approach. In Cardoso and Aalst (eds.), *Handbook of Research on Business Process Modeling*. IGI Global, 2009.

[MS21] T. Margaria, B. Steffen (eds.). *Leveraging Applications of Formal Methods, Verification and Validation - 10th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2021, Rhodes, Greece, October 17-29, 2021, Proceedings*. Lecture Notes in Computer Science 13036. Springer, Cham, 2021.
doi:10.1007/978-3-030-89159-6

[Per24] A. P. Perumal. Enhancing Hybrid Cloud Infrastructure Resource Provisioning and Management Through GitOps Workflow Automation. *International journal of applied engineering and technology (London)* 6, 01 2024.

[RSSZ19] F. Rademacher, J. Sorgalla, S. Sachweh, A. Zündorf. Viewpoint-Specific Model-Driven Microservice Development with Interlinked Modeling Languages. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. Pp. 57–66. 2019.
doi:10.1109/SOSE.2019.00018

[SBN] M. Sánchez-Cifo, P. Bermejo, E. Navarro. DevOps: Is there a gap between education and industry? *Journal of Software: Evolution and Process* n/a(n/a):e2534.
doi:https://doi.org/10.1002/smr.2534

[SBPM08] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley, Boston, MA, USA, 2008.

[SGNM19] B. Steffen, F. Gossen, S. Naujokat, T. Margaria. Language-Driven Engineering: From General-Purpose to Purpose-Specific Languages. In Steffen and Woeginger (eds.), *Computing and Software Science: State of the Art and Perspectives*. Lecture Notes in Computer Science 10000. Springer, 2019.
doi:10.1007/978-3-319-91908-9$_1$7

[SMB+24] B. Steffen, T. Margaria, A. Bainczyk, S. Boßelmann, D. Busch, M. Driessen, M. Frohme, F. Howar, S. Jörges, M. Krause, M. Krumrey, A.-L. Lamprecht, M. Lybecait, A. Murtovi, S. Naujokat, J. Neubauer, A. Schieweck, J. Schürmann, S. Smyth, B. Steffen, F. Storek, T. Tegeler, S. Teumert, D. Wirkner, P. Zweihoff. Language-Driven Engineering An Interdisciplinary Software Development

Paradigm. 2024.
https://arxiv.org/abs/2402.10684

[SS22] J. Schürmann, B. Steffen. *Lazy Merging: From a Potential of Universes to a Universe of Potentials*. Electronic Communications of the EASST, 2022.
doi:10.14279/tuj.eceasst.82.1226
http://dx.doi.org/10.14279/tuj.eceasst.82.1226

[ST21] J. Schürmann, S. Teumert. CI/CD In Theory and Practice. (Conference session) International School on Tool-based Rigorous Engineering of Software Systems (STRESS), Rhodes, Greece, 10 2021.

[StO] Stack Overflow Developer Survey 2023.
https://survey.stackoverflow.co/2023

[SWR$^+$21] J. Sorgalla, P. Wizenty, F. Rademacher, S. Sachweh, A. Zündorf. Applying Model-Driven Engineering to Stimulate the Adoption of DevOps Processes in Small and Medium-Sized Development Organizations. *SN Computer Science* 2(6):1–25, 9 2021.
doi:10.1007/s42979-021-00825-z

[TBS$^+$22] T. Tegeler, S. Boßelmann, J. Schürmann, S. Smyth, S. Teumert, B. Steffen. Executable Documentation: From Documentation Languages to Purpose-Specific Languages. In Margaria and Steffen (eds.), *Leveraging Applications of Formal Methods, Verification and Validation*. Lecture Notes in Computer Science 13702, pp. 174–192. Springer International Publishing, Cham, 2022.
doi:10.1007/978-3-031-19756-7$_1$0

[Teg23] T. Tegeler. *A Lingualization Strategy for Knowledge Sharing in Large-Scale DevOps*. PhD thesis, TU Dortmund University, Dortmund, Germany, 2023.
doi:10.17877/DE290R-23666

[Teu21] S. Teumert. Visual Authoring of CI/CD Pipeline Configurations. Bachelor's thesis, TU Dortmund University, Dortmund, Germany, 4 2021.

[TGS19] T. Tegeler, F. Gossen, B. Steffen. A Model-driven Approach to Continuous Practices for Modern Cloud-based Web Applications. In *2019 9th International Conference on Cloud Computing, Data Science Engineering (Confluence)*. Pp. 1–6. 2019.
doi:10.1109/CONFLUENCE.2019.8776962

[TTS$^+$21] T. Tegeler, S. Teumert, J. Schürmann, A. Bainczyk, D. Busch, B. Steffen. An Introduction to Graphical Modeling of CI/CD Workflows with Rig. In Margaria and Steffen (eds.), *Leveraging Applications of Formal Methods, Verification and Validation*. Lecture Notes in Computer Science 13036, pp. 3–17. Springer International Publishing, Cham, 2021.
doi:10.1007/978-3-030-89159-6$_1$

[TTS+22] S. Teumert, T. Tegeler, J. Schürmann, D. Busch, D. Wirkner. Evaluation of Graphical Modeling of CI/CD Workflows with Rig. In Margaria and Steffen (eds.), *Leveraging Applications of Formal Methods, Verification and Validation*. Lecture Notes in Computer Science 13702, pp. 374–388. Springer International Publishing, Cham, 2022.
doi:10.1007/978-3-031-19756-7$_2$1