A Lingualization Strategy for Knowledge Sharing in Large-Scale DevOps

Dissertation

zur Erlangung des Grades eines

D o k t o r s d e r I n g e n i e u r w i s s e n s c h a f t e n

der Technischen Universität Dortmund

an der Fakultät für Informatik

von

Tim Tegeler

Dortmund

2023

Tag der mündlichen Prüfung:

23. März 2023

Dekan:

Prof. Dr.-Ing. Gernot A. Fink

Gutachter:

Prof. Dr. Bernhard Steffen

Prof. Dr. Dr. h.c. Martin Wirsing

In memory of

Franz-Josef Tegeler

* 1953 – † 2001

Acknowledgements

First and foremost, I would like to thank *Bernhard Steffen* for supervising my dissertation. He was the one who inspired me to pursue a Ph.D. alongside my job at that time and later offered me the opportunity to work full-time at his chair to finish my dissertation. Our conceptual and technical discussions challenged me more than one time, but they were essential to sharpen my ideas and to avoid mistakes upfront.

Many thanks also go to *Martin Wirsing* for co-supervising my dissertation and motivating me to go the extra mile. The same applies to *Sven Jörges*, who advised me in the early stages of this dissertation. In addition to that, I would like to thank *Heinrich Müller* for chairing and *Ben Hermann* for completing my doctoral committee.

Furthermore, I want to thank *Falk Howar* for mentoring me during my studies and helpful advice whenever I needed it. This also applies to *Julia Rehder*, who plays a vital part at the chair and always has a friendly ear for me and my colleagues.

I consider *Steve Boßelmann* and *Stefan Naujokat* my senior advisors who welcomed me to the chair and helped me find my place in the group. Thank you for your advice and the countless hours that we worked together on research papers and software projects. In the same breath, I wish to acknowledge *Steven Smyth* and *Dominic Wirkner*, not only for their exceptional organization and construction of my amazing doctoral mortarboard, but also for their endless support and close teamwork on a daily basis.

Special appreciation go to my colleagues, *Alexander Bainczyk*, *Daniel Busch*, *Marcus Frohme*, *Frederik Gossen*, *Marc Jasper*, *Dawid Kopetzki*, *Michael Lybecait*, *Alnis Murtovi*, *Johannes Neubauer*, *Oliver Rüthing*, *Barbara Steffen*, and *Philip Zweihoff*. You proofed that not only software development is team work, but research as well. I enjoy the great atmosphere and environment in which we study, work and celebrate together.

In addition to that, I would like to mention *Jonas Schürmann* and *Sebastian Teumert* whom I had the pleasure to supervise during their Master's and Bachelor's thesis, respectively. Although I was your supervisor, you have taught me as much as I may have taught you. I'm very happy that both of you are my colleagues now. I'm curious about your future careers and scientific achievements.

I want to extend my gratitude to the *Lonnemann* family and the team of the *Creios GmbH* for supporting me at the early stages of my career and allowing me to pursue my Ph.D. in part-time. Furthermore, I want to thank my former colleagues *Simon Zuelsdorf* and *Marcel Willnow* for almost seven years of teamwork, which turned into friendships.

Finally, I want to thank my parents *Marianne* and *Meinhard* for their never-ending support throughout my educational career. You believed in me when others did not.

Abstract

DevOps has become a generally accepted practice for software projects in the last decade and approaches certain shortcomings of the agile software development and the steadily gaining popularity of cloud infrastructure. While it shifts more and more responsibilities towards software engineering teams, the prevailing opinion is to keep DevOps teams small to reduce the complexity of inter-team communication. In circumstances where products outgrow the performance capability of a single team, microservice architecture enables multiple DevOps teams to contribute to the same application and meet the increased requirements. Since DevOps teams operate typically self-sufficiently and more or less independently inside an organization, such large-scale DevOps environments are prone to knowledge-sharing barriers.

Textual Domain-Specific Languages (DSLs) are one of the cornerstones of DevOps and enable key features like automation and infrastructure provisioning. Nonetheless, most commonly accepted DSLs in the context of DevOps are cumbersome and have a steep learning curve. Thus, they fall short of their potential to truly enable cross-functional collaboration and knowledge sharing, not only between development and operation, but to the whole organization. DevOps teams require tools and DSLs, that treat knowledge sharing and reuse as a first-class citizen, in order to operate sufficiently on a large scale. However, developing DSLs is still presumed as an expensive task which can easily offset the resulting benefits.

This dissertation presents a *lingualization strategy* for addressing the challenge of knowledge sharing in large-scale DevOps. The basic idea is to provide custom-tailored Domain-Specific Modeling Languages (DSMLs) that target single phases of the DevOps lifecycle and ease the DevOps adoption for newly formed teams. The paradigm of Language-Driven Engineering (LDE) bridges the semantic gap between stakeholders by custom-tailored DSMLs and thus is a natural fit for knowledge sharing.

Key to a successful practice of LDE is as a new class of stakeholders. In the context of large-scale DevOps, language development can be realized by so-called Meta DevOps teams. Those teams, which themselves practice DevOps internally, manage a centralized repository of small DSMLs and offer them as a service. DevOps teams act as the customers of the Meta DevOps teams and can request new features or complete new DSMLs and provide feedback to already existing DSMLs. The presented Rig modeling environment serves as an exemplary DSML that targets the purpose of Continuous Integration and Deployment (CI/CD), one of the most important building blocks of DevOps. Rig comes with an associated code generator to fully-generate CI/CD workflows from graphical models. Those graphical models provide an executable documentation and assist knowledge-sharing between stakeholders.

The fundamental modeling concepts of the lingualization strategy are evaluated against previously published requirements by Bordeleau et al. on a DevOps modeling framework in an industrial context. In addition to that, Rig is evaluated based on results of a workshop during the *6th International School on Tool-Based Rigorous Engineering of Software Systems*. Both evaluations yield encouraging results and demonstrate the potential of the lingualization strategy to break down knowledge-sharing barriers in large-scale DevOps environments.

Attached Publications

This enumeration contains the publications, which are part of this dissertation, and introduces a special way of reference (e.g. [AP I: [152]]) in order to distinguish them from other literature. They have already been published in cooperation with other authors. Section 1.2 briefly outlines the content of each publication alongside comments of my contribution.

I Tim Tegeler, Frederik Gossen, and Bernhard 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)*. 2019. DOI: 10.1109/CONFLUENCE.2019.8776962: Below cited as [AP I: [152]]

II Tim Tegeler and Jonas Schürmann. **Evolve: Language-Driven Engineering in Industrial Practice**. In: *Electronic Communication of the European Association of Software Science and Technology,* (2018). DOI: 10.14279/tuj.eceasst.78.1089: Below cited as [AP II: [153]]

III Jonas Schürmann, Tim Tegeler, and Bernhard Steffen. **Guaranteeing Type Consistency in Collective Adaptive Systems**. In: *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles*. 2020. DOI: 10.1007/978-3-030-61470-6_19: Below cited as [AP III: [139]]

IV Tim Tegeler, Sebastian Teumert, Jonas Schürmann, Alexander Bainczyk, Daniel Busch, and Bernhard Steffen. **An Introduction to Graphical Modeling of CI/CD Workflows with Rig**. In: *Leveraging Applications of Formal Methods, Verification and Validation*. 2021. DOI: 10.1007/978-3-030-89159-6_1: Below cited as [AP IV: [154]]

V Philip Zweihoff, Tim Tegeler, Jonas Schürmann, Alexander Bainczyk, and Bernhard Steffen. **Aligned, Purpose-Driven Cooperation: The Future Way of System Development**. In: *Leveraging Applications of Formal Methods, Verification and Validation*. 2021. DOI: 10.1007/978-3-030-89159-6_27: Below cited as [AP V: [181]]

VI Tim Tegeler, Steve Boßelmann, Jonas Schürmann, Steven Smyth, Sebastian Teumert, and Bernhard Steffen. **Executable Documentation: From Documentation Languages to Purpose-Specific Languages**. In: *Leveraging Applications of Formal Methods, Verification and Validation*. 2022. DOI: 10.1007/978-3-031-19756-7_10: Below cited as [AP VI: [151]]

VII Sebastian Teumert, Tim Tegeler, Jonas Schürmann, Daniel Busch, and Dominic Wirkner. **Evaluation of Graphical Modeling of CI/CD Workflows with Rig**. In: *Leveraging Applications of Formal Methods, Verification and Validation*. 2022. DOI: 10.1007/978-3-031-19756-7_21: Below cited as [AP VII: [157]]

Contents

1 Introduction **1**
- 1.1 My Contribution . . . 2
- 1.2 Context of Attached Publications . . . 5

2 DevOps **9**
- 2.1 Continuous Practices . . . 12
- 2.2 Large-Scale DevOps . . . 13
- 2.3 Container Virtualization . . . 15

3 Knowledge Sharing in Large-Scale DevOps **17**
- 3.1 Domain-Specific Languages . . . 19
- 3.2 Lingualization Strategy . . . 20
- 3.3 Purpose-Specific Languages . . . 23
- 3.4 Code Generators . . . 26
- 3.5 mindset-supporting IDEs . . . 28
- 3.6 Organizational Infrastructure . . . 29
- 3.7 The Knowledge Sharing Process . . . 31

4 Graphical Modeling with Rig **33**
- 4.1 Rig Is a Cinco Product . . . 34
- 4.2 Integrability . . . 35
- 4.3 Meta-Level Bootstrapping . . . 38

5 Evaluation **41**
- 5.1 Modeling Concepts of the Lingualization Strategy . . . 41
- 5.2 Numbers on the Effect of Graphical Modeling with Rig . . . 44

6 Related Work **49**
- 6.1 Knowledge Sharing . . . 49
- 6.2 MD* in the Context of DevOps . . . 51

7 Conclusion **53**
- 7.1 Limitations . . . 53
- 7.2 Future Work . . . 55

References **59**

Online References **71**

Chapter 1

Introduction

> "Data expands to fill the space available for storage."
> — *Parkinson's Law of Data* [158]

In the past, programs were typically written in a single programming language on one computer by a single person [30]. Since then, software systems have gone beyond the scope of what once has been labeled "large computer programs" [14] and have blown up all three dimensions. They are developed in teams, run distributed on potentially unlimited amount of machines and are composed of different programming languages [30, 82].

This evolution forced software developers to continuously improve and adapt to the latest practices, ever since the *NATO Software Engineering Conferences* 50 years ago [35]. In the last decades, Agile became a popular term for a set of practices to approach the challenges of collaborative software development [35] and rapid requirements changes [65]. The rise of cloud infrastructure gave developers easy access to scalable computing resources [130]. Both trends are considered the main driver for another popular practice, called DevOps [37]. DevOps approaches the challenge of cross-functional teams and removes the barrier between development and operations [65]. At the same time, microservice architecture enabled decomposition of monolithic applications into smaller pieces and allowed independent DevOps teams to contribute to the same application [7], by using their own tool stack and the appropriate programming language for their use case.

However, globally operating online platforms have become so complex that the effort of continuous development and maintenance outgrew the productivity of small-scale projects [42]. Such environments with multiple self-organized teams are called large-scale Agile or respectively large-scale DevOps [42]. Since DevOps teams are typically independent units inside an organization that build and run their own service in a much broader ecosystem, large-scale DevOps is prone to 'knowledge silos' [42].

Despite the fact that knowledge is considered a competitive advantage [126, 133] and knowledge sharing is an important activity of knowledge management [26, 133, 110] in general, knowledge silos continue to exist in organizations. Such silos emerge when knowledge, like "skills, information, expertise, experience [or] intelligence" [110], is not actively shared among peers. This bad practice can originate from various issues inside an organization: Employees are not willing to share knowledge to protect their own position, team affiliation creates natural knowledge barriers, and organizations are not prioritizing knowledge sharing, accompanied by the lack of knowledge sharing infrastructure.

Knowledge sharing is also a prominent demand in the twelve principles behind the Agile manifesto [57], as the following two principles illustrate: "Business people and developers must work together daily throughout the project." [12] and "The most efficient and effective method of conveying information to and within a development team is face-to-face conversation." [12]

Accordingly, DevOps considers sharing "one of [...] four pillars" [65] and literature assumes that its practice successfully diffuses knowledge inside a team [37]. Nevertheless, knowledge sharing in large-scale DevOps between teams is an ongoing challenge [142, 65]. Since DevOps is more pragmatism than science, it did not receive "much attention from research" [167], but has already been addressed in popular novels like the *The Phoenix Project* (cf. [81, 80]).

This dissertation is motivated by the following area of conflict: Firstly, Dingsøyr et al. list "Knowledge sharing and improvement" [43] as one of eight research challenges in the context of large-scale Agile, while Agile itself lacks general knowledge sharing practices [133]. Secondly, knowledge sharing barriers can threaten the success of software projects or even whole organizations. Finally, if the team size of agile projects should be kept small [10, 7] to reduce complexity of communication [42], how can knowledge be shared efficiently in large-scale environments?

Literature on knowledge management in the context of Agile software development [133] lists two higher-level types of sharing strategies: On the one hand, the personalization strategy, which describes knowledge sharing in a network of people, via synchronous communication and personal interaction, and on the other hand, the codification strategy, which uses computer-based tools and repositories to store and access knowledge. In the context of large-scale DevOps, a distinction can also be made between *inter*-team and *intra*-team knowledge sharing. Intra-team means sharing within a team while inter-team is sharing among/between teams [69]. Due to the fact that multiple teams imply a higher number of participants, it is a trivial finding that sharing knowledge among multiple teams is more difficult than within a team. Applying the personalization strategy will eventually face scaling challenges.

In contrast to this, the asynchronous nature of the codification strategy (cf. [41]) allows the extraction of knowledge "from the person who developed it, made independent of that person, and reused for various purposes" [63]. This implies, as a synonym for programming, "a transformation of some original formulation into a different usually more cryptic form" [140]. However, this 'cryptic form' is necessary in order to be processed and persisted by a computer, but might not be the best representation for humans and requires a visual presentation.

This dissertation presents a holistic approach to the challenge of efficient knowledge sharing in large-scale DevOps. The foundation of this approach is a *lingualization strategy* which goes beyond the simple 'codification' of knowledge and makes it immediately applicable through a special form of Domain-Specific Modeling Languages (DSMLs) [91, 56]. It follows the principle of Language-Driven Engineering (LDE) [146] and shows how this strategy can iteratively be implemented, when practicing DevOps, to let knowledge sharing become a key factor of organizational success.

1.1 My Contribution

In the course of my Ph.D. studies, I have spent the first two years in part-time. While I have started my studies at the Chair of Programmingsystems at the TU Dortmund University, I was still employed in the software industry at the Creios GmbH, where I was mainly responsible for *Research and Development*. I was in charge of several web application projects, always with the focus on introducing new technology and approaches, while improving the overarching development workflow.

Since the business model of the Creios GmbH was focused on providing Software-as-a-Service during my time in the company, the software not only had to be developed but also operated in-house. Hence, I proposed DevOps as our default software development methodology and

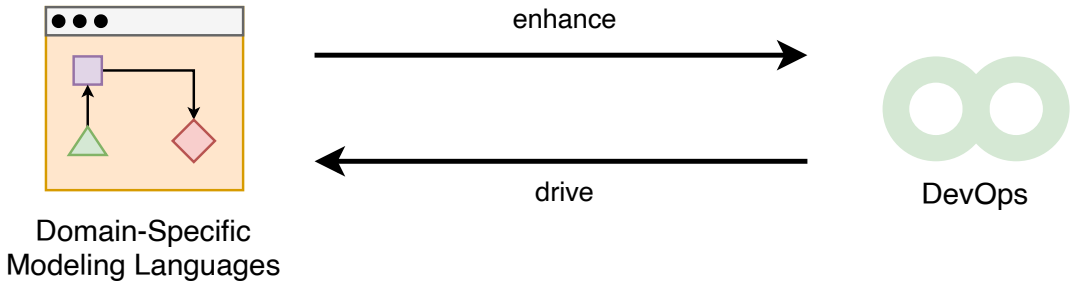


Figure 1.1: Reciprocal interaction of Domain-Specific Modeling Languages and DevOps

successfully adopted it for ongoing and new projects. At the same time, I came in touch with graphical DSMLs at the Chair of Programmingsystems. I was given the opportunity to contribute my years of practical experience to improve the management and development workflow of the aforementioned DSMLs projects.

This duality gave me valuable insight in both academic and industrial projects, and let me discover challenges that arose in both worlds. From my perspective, long-living academic projects often lack coordination of source code changes that come from various contributions, for instance final year projects and student project groups. Thus, they have a high potential of benefiting from project management, well-structured branching models, and (build) automation. In contrast to projects in academia which are not used in production, industrial projects run into danger to accumulate technical debt, especially if the software is in daily use and a corner stone of the companies' success. Bridging the gap between research and industry is the motivation of my scientific contribution.

In my talk *"Continuous Practices in the Context of Model-Driven System Development"* at the Doctoral Symposium of the ISoLA Conference in 2018 [89], I have illustrated the aforementioned observation and proposed the idea of approaching the challenges of academic projects in the Domain of DSML with the benefits of industrial software development methodologies like DevOps and vice versa. Figure 1.1 shows the envisaged reciprocal interaction. Since this starting point, I have made the following scientific contributions:

- Sketching the concept of a DSML including a first draft of a language design for Continuous Integration and Deployment (CI/CD) workflows, cf. [AP I: [152]].
- Developing a family of textual Domain-Specific Languages (DSLs), establishing them in my industrial projects at the Creios GmbH and evaluating the results, cf. [AP II: [153]].
- Supervising the implementation of Rig, a mindset-supporting IDE (mIDE) for modeling CI/CD workflows which builds upon the aforementioned language design, cf. [AP IV: [154]].
- Organizing a workshop on graphical modeling of CI/CD with Rig at the *6th International School on Tool-Based Rigorous Engineering of Software Systems* [97], cf. Section 5.1, and cooperatively evaluating Rig based on the gathered data of this workshop, cf. [AP VII: [157]].
- Evaluating the modeling concepts of the lingualization strategy against criteria of a modeling engineering framework for DevOps which were postulated and published by Bordeleau et al. in [18], c.f. Section 5.1.
- Extending the scope of LDE from a pure development perspective, to a holistic approach that incorporates the operational part of DevOps as a first-class citizen, cf. [AP VI: [151]].

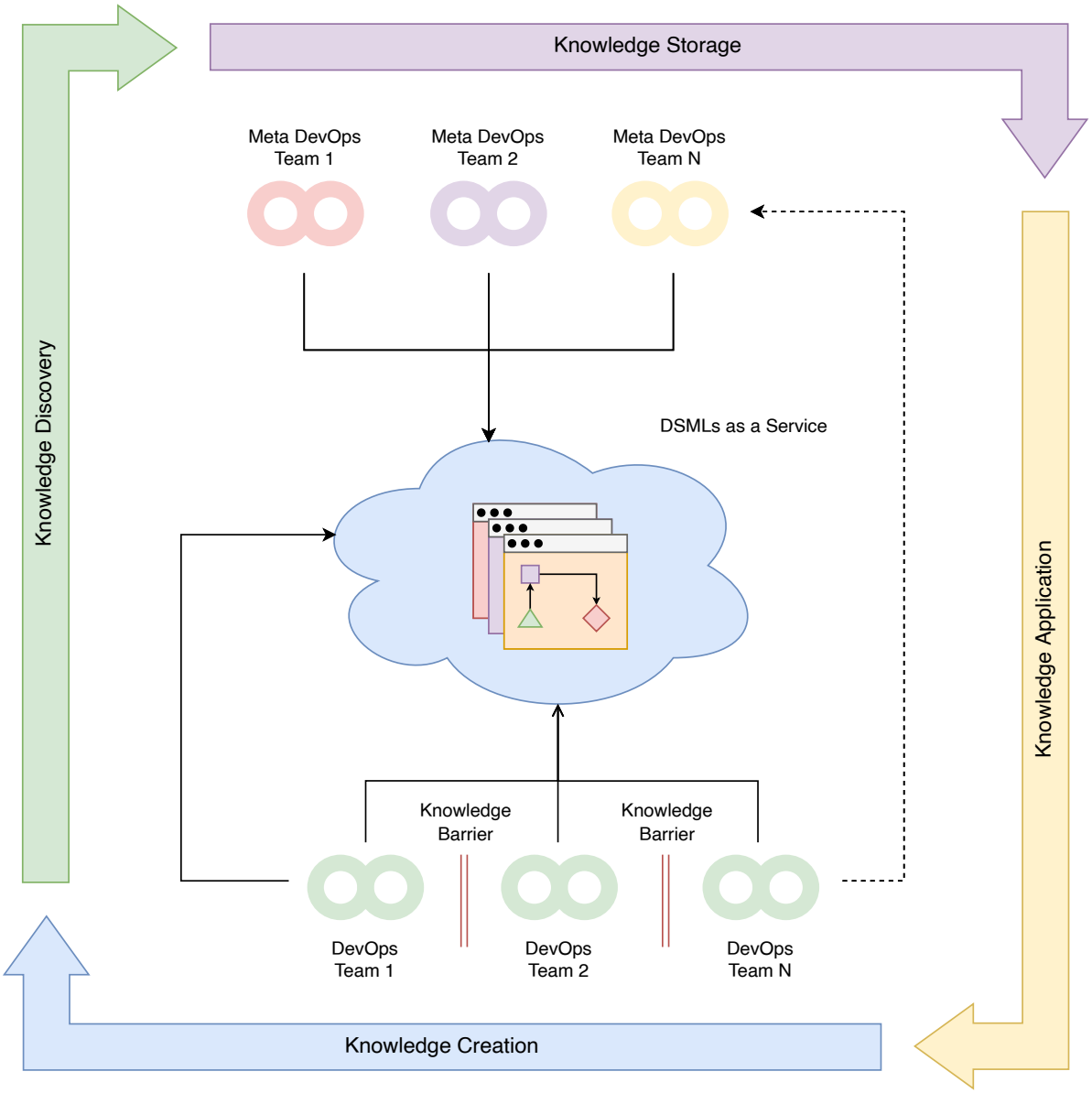


Figure 1.2: Knowledge sharing cycle of the lingualization strategy

The summarizing contribution of my dissertation is a lingualization strategy for knowledge sharing via DSMLs in large-scale DevOps environments that are prone to inter-team knowledge barriers. In the following chapters I will address the challenges that arise from practicing large-scale DevOps, and propose an organizational wide approach of how DSML can be developed to support the transfer of knowledge between independent DevOps teams. Beforehand, I will give a brief overview on the lingualization strategy in the remainder of this section alongside Figure 1.2.

DevOps teams in large-scale DevOps are more or less self-administering teams that are responsible for developing single components or services in a larger system. Accounted for by the uniqueness requirements of each subproject, it is hardly possible to foresee how to adopt DevOps as the software development methodology for a given project and which phases of the DevOps lifecycle (cf. Figure 2.3) take preference. The maturity of DevOps adoption is very likely to vary between teams in a large-scale environment and the independence of DevOps

teams increase the chance of inter-team knowledge barriers. As a result, new assembled teams face the challenge of adopting DevOps without directly benefiting from the experience and lessons learned of more mature teams.

The fundamental idea of the lingualization strategy is based on a continual knowledge sharing cycle by tapping into the experience of mature DevOps teams and help recent constituted teams to benefit from the existing knowledge. Textual DSMLs as part of the *as-code* movement [88], greatly influenced the practice of DevOps [37] and are widely accepted to support processes of the DevOps lifecycle, like build automation or infrastructure provisioning. Thus, it is my opinion that they are a promising medium to store, share and apply discovered knowledge, especially in the context of DevOps.

Key for the success of this knowledge sharing cycle are dedicated teams that are responsible for discovering innovative knowledge from existing teams and developing custom-tailored DSMLs that can help other teams to adopt certain phases of the DevOps lifecycle with less effort. I coined the term *Meta DevOps teams* for this "new class of stakeholders" [146]. As Figure 1.2 shows, Meta DevOps teams are located (on the meta level) above DevOps teams, since they are responsible for language development and don't participate in regular projects. This is a reminiscence of the term Meta model [72, 112, 92]. Furthermore, it is my opinion that DevOps as a software engineering methodology is a natural fit for language development (cf. Section 3.2). Hence the name *Meta DevOps*. In order to keep the entry barrier low for regular DevOps teams, such Meta DevOps teams provide the developed DSMLs as a web-based service.

The following chapters will elaborate on this sketch of the lingualization strategy and are structured as follows: Chapter 2 introduces the most important topics on DevOps and identifies the main issues in large-scale DevOps. Afterwards Chapter 3 builds the theoretically background of the lingualization strategy. Based on this background Chapter 4 presents a proof of concept of the lingualization strategy alongside Rig, an mIDE for modeling CI/CD workflows. In addition of the evaluation of Rig, supported modeling concepts of the lingualization strategy are evaluated against the criteria of a modeling engineering framework for DevOps in Chapter 5. Related work in the context of knowledge sharing strategies and other model-driven approaches in the domain of DevOps are discussed in Chapter 6. Lastly, Chapter 7 concludes my dissertation by discussing limitations of the lingualization strategy and future work.

1.2 Context of Attached Publications

This section presents the context of the attached publications, highlights content that is important for the remainder of this dissertation and outlines my contribution of each publication.

A Model-driven Approach to Continuous Practices for Modern Cloud-based Web Applications

[AP I: [152]] presents a graphical language for the modeling of Continuous Integration and Deployment workflows and the full code generation of textual configurations. It argues that writing correct CI/CD configurations manually remains an error-prone task at the time of writing, due to the lack of reuse, non-existing coding assistance and limited testing possibilities. The presented approach of graphical modeling and automated generation, 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. Although the paper is motivated by the example of modern cloud-based web applications, the approach of modeling

CI/CD is generally applicable for all types of applications. The graphical language and code generator are developed using the Cinco *Meta Tooling Suite* [113] and initiated the Rig project.

The presented concepts were discussed among all authors, but I came up with the initial idea of creating a graphical language for CI/CD workflows. Frederick Gossen and I created the first prototype of this language. I was main author of section 1, 2, 3, 4 and co-authored section 5, 6 and 7.

Evolve: Language-Driven Engineering in Industrial Practice

[AP II: [153]] introduces a toolkit and a family of textual DSL to reduce the labour of application developers by generating recurring modules and boilerplate code in an industrial project. The paper documents the impact of the DSLs on the development process and shows a reduction of 24% handwritten source code. A strong focus on CI/CD helped to quickly deliver new language versions to the application developers.

It was my idea of bringing LDE to industrial practice and to initiate the Evolve project. The toolkit, the DSL and the automated delivery process have been primarily implemented by myself. The presented concepts in this paper were discussed among both authors. I was main author of all sections.

Guaranteeing Type Consistency in Collective Adaptive Systems

[AP III: [139]] presents an approach how to guarantee type safety in collective adaptive systems by an external DSL called Type-Safe Functional GraphQL (TFG), that extends the query language of GraphQL. It evaluates queries against an existing GraphQL schema to spot type-errors, during generation time of the queries, during compile time of the target language and during runtime of the client application. The paper is structured alongside an exemplary request against the GraphQL API of GitHub and concludes how executable DSLs like TFG can support the *shift left* paradigm of DevOps where spotting errors as early as possible in the development process is vital.

TFG was implemented by Jonas Schürmann as part of his Master's thesis [137], which I supervised. The presented concepts in this paper were discussed among all authors. I was main author of section 1, 7, 8, and co-authored section 2.

An Introduction to Graphical Modeling of CI/CD Workflows with Rig

[AP IV: [154]] illustrates graphical modeling of CI/CD workflows with Rig. It was the cornerstone of a workshop at the *6th International School on Tool-Based Rigorous Engineering of Software Systems* [97] as part of a lecture focussing an international group of Ph.D. students. The paper is motivated by an example of a manually 'programmed' CI/CD configuration of the TodoMVC [120] and an equivalent graphical workflow modeled in Rig. It discusses how a graphical modeling language helps to abstract from the actual textual configuration and how a code generator can take away error-prone traits of the target language.

It was my idea to organize the aforementioned workshop on CI/CD workflows which initiated the writing of this introductory paper of Rig. The presented concepts of this paper were discussed among all authors. I was main author of sections 1, 2, 4, 7, 8. The Rig implementation has been primarily done by Sebastian Teumert.

Aligned, Purpose-Driven Cooperation: The Future Way of System Development

[AP V: [181]] discusses the challenge of aligning *space*, *time* and *mindset* in collaborative system development. The paper presents state-of-the-art cloud IDEs that already address the alignment of *space* and *time* and argues how LDE can integrate the *mindset* of different stakeholders, by dedicated Purpose-Specific Languages (PSLs) in the development process. This three-dimensional alignment problem is approached by the sketch of a web-based Cinco environment, that is realized by modern cloud native technology and driven by the DevOps lifecycle.

The presented concepts of this paper were discussed among all authors. I co-authored the sections 1, 2, 6 and was main author of section 2.1.

Executable Documentation: From Documentation Languages to Purpose-Specific Languages

[AP VI: [151]] presents the approach of executable documentation and shows how the domain-specific notation of graphical languages can be turned into fully-fledged modeling languages. The paper demonstrates this approach in the domain of DevOps, alongside GitLab's visualization for documenting executed CI/CD workflows. By applying the concept of LDE, this graphical notation is incorporated into a custom-tailored PSL and shipped as part of an Integrated Modeling Environment called Rig. This allows domain experts to model (i.e. document) CI/CD workflows in their well-known notation — instead of using cumbersome textual languages — and subsequently execute the model to generate CI/CD configurations.

I designated *executable documentation* as the term for the presented concept in this paper and contributed the DevOps perspective of executable documentation in practice. The presented concepts of this paper were discussed among all authors. I was main author of section 3.0 and 4 and co-authored section 1, 2, 3.3, 5 and 6.

Evaluation of Graphical Modeling of CI/CD Workflows with Rig

[AP VII: [157]] evaluates Rig based on pipeline data gathered during a workshop at the *6th International School on Tool-Based Rigorous Engineering of Software Systems* [97]. The paper formulates three research questions that are derived from the initial idea of graphical modeling of CI/CD workflows (cf. [AP I: [152]]) and discussed them alongside the interpreted pipeline data. The data leads to the conclusion that graphical modeling has the potential to reduce trial-and-error, eliminate syntax errors and lower the entry barrier, when compared with textual DSLs. In addition to this, the workshop setup and related lessons learned are illustrated to help the planning of future workshops, before the paper concludes with threats to validity.

Sebastian Teumert gathered and subsequently evaluated the accrued data from the aforementioned workshop, but it was my idea of embedding the results in broader evaluation of the overall workshop that constituted this paper. The presented evaluation, interpretation and threats of validity were discussed among all authors. I was main author of section 4 and co-authored section 1, 2, 3, 6, 7 and 9.

Chapter 2

DevOps

> "Some people get stuck on the word 'devops,' thinking that it's just about development and operations working together."
> — *Patrick Debois* [37]

In contrast to the Agile Manifesto [12], there is no concluding definition of DevOps [167]. The concrete practice of DevOps is unique to the organization, the use-case and the involved humans [35] in a project. Nevertheless, to the best of my knowledge, literature and research agree on the basic idea of DevOps as a technological, cultural and organizational shift [35]. The following section will give a 'snapshot of DevOps' [35] in the context of this dissertation and outline important aspects for the challenge of knowledge sharing in large-scale DevOps.

Business organizations are typically composed of different departments [93], in order to segregate duties, simplify controlling, and ensuring compliance [37]. Each department serves a specific role (e.g. accounting, sales, and human resources) inside the organization and employs its own staff with certain jobs. Software companies adopted this style of organizational structure and group employees by their expertise/qualification in separated departments [37]. As a consequence, software projects pass through different departments or i.e. phases [167] during their life span: In simplified terms, sales acquires a project, product management gathers requirements, development implements functionality and operations runs the system. Such waterfall-style software development methodology [14, 35] is prone to the loss of knowledge, especially during the transition between departments (cf. Figure 2.1). Following the practices of DevOps, organizations are structured alongside projects (cf. Figure 2.2), and the affiliation to a department takes a back seat. Hence, it reduces hierarchical structures [65] and organizational barriers [37].

The evolution of DevOps is driven by the higher release frequency of Agile Software Development and the emergence of cloud computing [37]. Although the general availability of cloud computing and its managed infrastructure virtualization let to sophisticated technologies like infrastructure-as-code, the ideas and the practice of DevOps are well-suited for traditional unmanaged on-premise solutions as well. DevOps can be considered an extension of Agile [65] that exceeds the scope of the development process [35]. The term DevOps itself is a portmanteau composed of *development* and *operations* [65]. Debois pointed out in [37], that is goes beyond both activities and must apply to whole organizations. It is not uncommon to include business stakeholders [167] as well, but this is out of scope of this dissertation.

The traditional four pillars of DevOps[1] are *culture*, *automation*, *measurement* and *sharing* [37, 65]. Afterwards, they have been expanded by the aspect of *lean* and summarized under the term CALMS [167], which highlights the Agile roots of DevOps.

[1] It is worth mentioning that [35] proposes the alternative pillars *collaboration*, *affinity*, *tools* and *scaling* with a focus on so-called 'effective DevOps'.

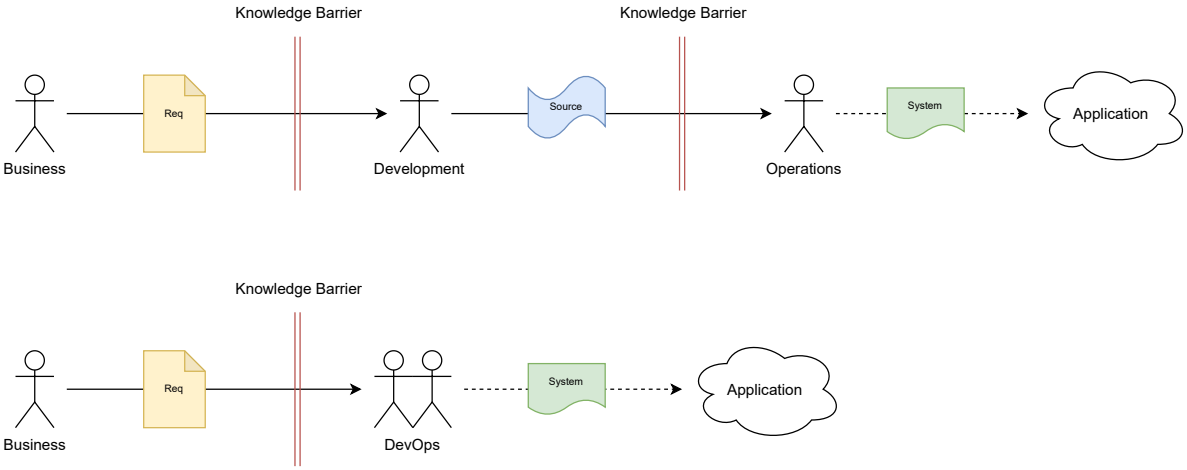


Figure 2.1: Traditional workflow (top) and DevOps workflow (bottom)

Culture represents the importance of cooperation across all areas through the whole lifecycle of a software project. Instead of aiming at self-seeking local solutions, members from different departments should acknowledge the concerns of their counterparts and come to accommodation that suits all. Debois et al. [37] demand two major adaptions to overcome the department barriers: On one hand, operations must be involved from the very beginning of the planning phase to recognize operational requirements early on. On the other hand, developers should practice (job) rotation with operations to assist in the event of a failure. Creating such a positive culture through cooperation is key for a true cross-functional collaboration.

Automation is the technological 'glue' between development and operation, and one dimension of the DevOps maturity model presented in [107]. In order to successfully automate tasks, the deep knowledge and the seamless integration of broad knowledge from both sides is necessary. Automation can be applied to small every day activities and even to complex and error-prone tasks. It culminates in the practice of Continuous Integration and Deployment (CI/CD) where pipelines automatically provision complete testing and production environments [37] (cf. Section 2.1). This includes not only contributions made to the application's source code, but modifications of the entire system.

Lean aims at simplifying and optimizing the manufacturing processes of a software system by elimination of waste [123, 173, 94]. Part of this practice is to minimize code contributions to discrete changes [167] and to maximize the *Archimedian Points*, "things that do *not* change" [148], while altering source code. Smaller steps improve the controllability and predictability of changes [148]. They reduce unnecessary 'noise' from feedback loops and allow a higher frequency of releases. As a result, the increased quality of the whole lifecycle and the improved delivery pace contributes to the business value of a project.

Measurement is the key element in order to improve a software project. Feedback (automated and manual) helps to reveal limitations and to schedule priorities of future development work. Debois et al. [37] picture an overarching measurement strategy of a system which includes both lower-level metrics and higher-level metrics. Lower-level metrics (e.g. hardware workload, error rate) let DevOps teams to observe health and stability of the system, while high-level metrics or respectively *key performance indicators* [55] (e.g. daily sales, user registrations) provide an insight into customer satisfaction and the business-wise success of a system. All metrics should be available to everyone in the team [37] and its result must be incorporated into the next version of the software.

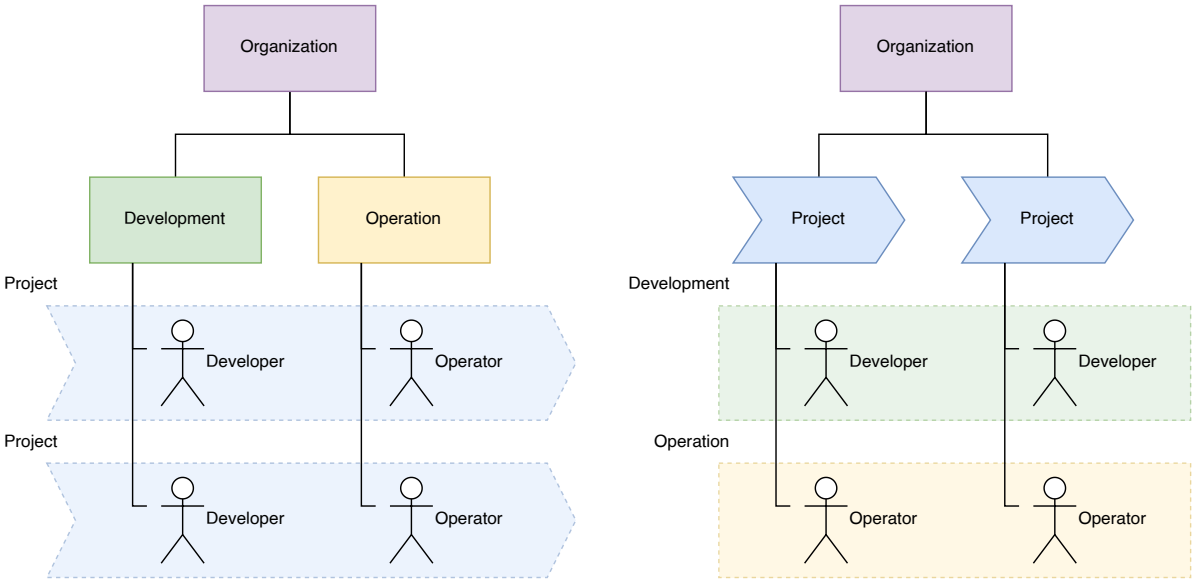


Figure 2.2: Traditional organizational structure (left) and DevOps structure (right)

Sharing enables the alignment and understanding of cross-functional teams. Knowledge silos (i.e. knowledge barriers, cf. Figure 2.2), which were a typical result of traditional project workflows, are threatening the performance and resiliency of a team, not only when it comes to incidents. Sharing involves project knowledge (e.g. functionality or requirements) and also system knowledge, like the configuration of production environments [37]. A well performing team shares best practices in order to ease the transition between development and operation.

Organizational structure is just a starting point when adopting DevOps. Transition from a traditional software development methodology (e.g. waterfall) [14] with occasional releases and limited project lifetime, to an ongoing and continuous development with rapid releases [167], requires a fundamental change of current practice.

Since DevOps is not a one-size-fits all approach and requires a custom adaption to local conditions of project or organization [35, 167], this dissertation won't address concrete tools or solutions for the most part, but focus on the practice on a broader level. DevOps operates alongside an infinite continuous collaboration (cf. Figure 2.3) lifecycle [176] with consecutive phases that capture the CALMS components.

Plan is the virtual starting point of the DevOps lifecycle. In the context of a novel project, it focuses on the planning of the first iteration with the goal of groundwork and less functionality. During a running and well established project, the results of the previous **Monitor** and **Continuous Feedback** phase are evaluated and scheduled for the nth iteration.

Create summarizes both development and operational work. Scheduled issues from the previous stage can be resolved by joined collaborations. This involves the full range from new features to major infrastructure adjustments. Development environments enable the local execution of (unit) tests and the examination of the whole application.

Continuous Delivery aims at fully automated releases that are stored in a repository and are ready for deployment [35]. This is realized by so-called pipelines that produce executable artifacts from source code. It relates to Continuous Integration (CI) that builds the application from scratch and performs test execution to ensure that new contributions do not break the pipeline.

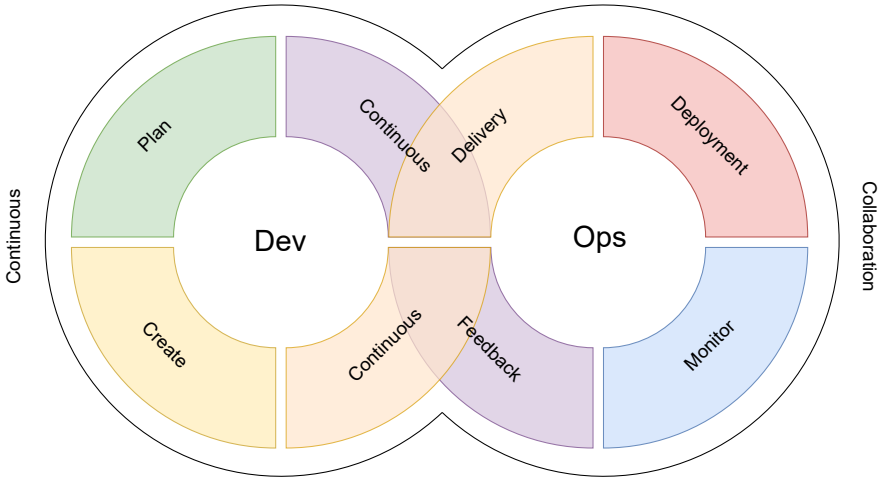


Figure 2.3: Continuous collaboration in the DevOps lifecycle (cf. [5, 79])

(Continuous) Deployment takes over where the previous stage ended and deploys the prepared releases to target environments. In the context of DevOps, teams use distinct environments for different purposes, like tests or demonstration. The deployment to the production environment takes the same path (i.e. pipeline) as the other environments.

Monitor visualizes the impact of applied changes to the production environment and uncovers potential regressions. As mentioned earlier in this section (cf. CALMS), this can include low-level and high-level metrics and should be open to all team members, independent of their expertise.

Continuous Feedback is the last stage in the DevOps lifecycle and marks the transition to the $(n+1)$th iteration. It gathers all the 'learned lessons' of the nth iteration. Part of this is the feedback from all stakeholders from the business to the end user.

Although Figure 2.3 may create the impression that lifecycle phases are assigned to a single department, based on their placement in the loop (left Dev, right Ops), both professions are accountable for a successful DevOps practice. To conclude, DevOps promotes transparency between all stakeholders and aims at uncovering issues as early as possible during the systems' lifecycle (i.e *shift left*) by reducing the time to production, emphasizing testing to spot errors automatically, improving the reliability of the deployment process, and optimizing the response to incidents [9].

2.1 Continuous Practices

This section aims to define related terms in the context of CI/CD that are often used synonymously in both literature and research. While the coinage of the term Continuous Integration (CI) is often attributed to an article of Martin Fowler from 2000 [177, 152, 51], later Fowler himself refers to Kent Beck's *Extreme Programming* methodology [11, 52] in a revised article from 2006 [50]. The slightly different definitions of Continuous Deployment (CD) may root in two independent developments of very similar concepts how to continuously deliver software by Jez Humble and David Farley, and Tim Fitz [82]. It is not in the scope of this section to finally resolve the origin of each term, but to present definitions that take effect for the upcoming chapters.

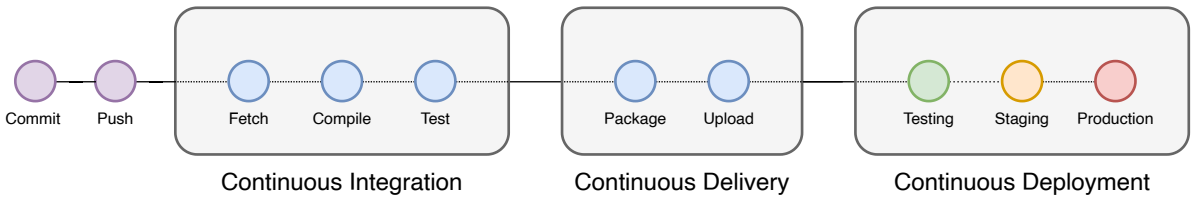


Figure 2.4: Schematic outline of jobs in a CI/CD pipeline

This dissertation considers *Continuous Practices* as an umbrella term for CI, Continuous Delivery (CDE) and CD [141]. At the higher level those practices aim at increasing the quality of software development, ensuring stability of software systems, and reducing costs of software projects [163]. Although the concrete practice of those three terms is unique to related projects and applied programming languages, they can be distinguished by their primary goal:

- CI aims at supporting the *integration* of a steady stream of changes in an existing code base by relying on automated dependency fetching, compiling and extensive (unit) testing.
- CDE aims at the *delivery* of compiled and packaged applications by uploading it to a central repository as a ready-to-deploy artifact.
- CD aims at the *deployment* of the former delivered artifact into deployment environments for testing purposes and 'all the way up' to the real production system for a continuous go-live.

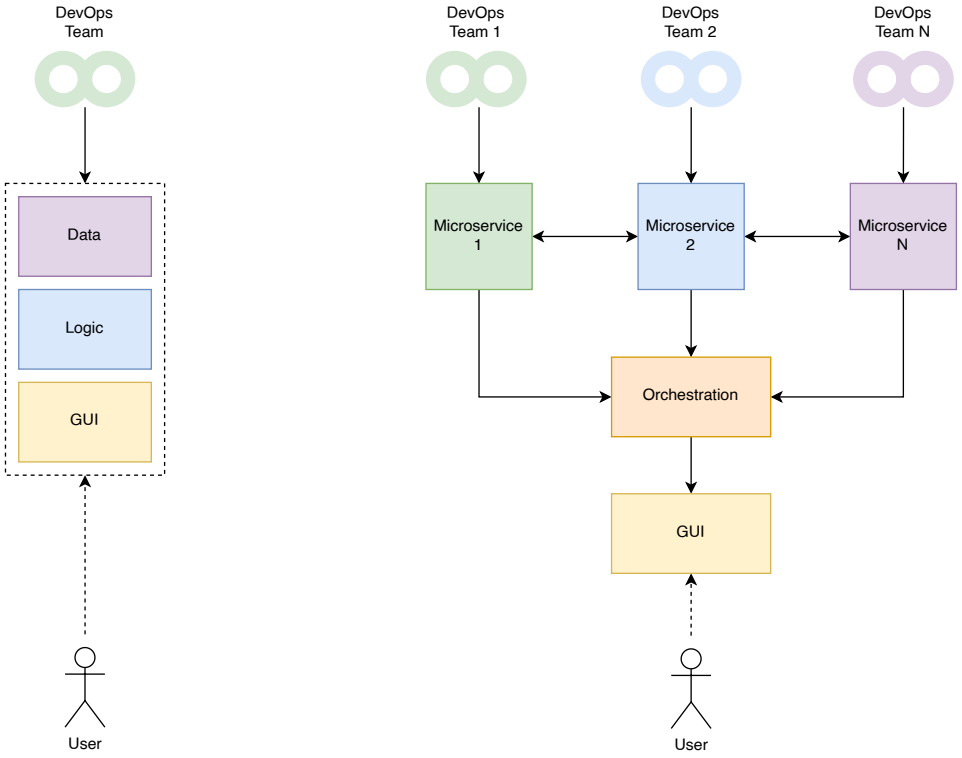
In addition to this triad of practices, the term *pipeline* is also overloaded and often used ambiguously in the context of CI/CD [AP IV: [154]]. Looking forward towards Chapter 4 of this dissertation, an explicit definition is necessary. [AP IV: [154]] proposes the introduction of the terms *workflow* and *configuration* as a distinction to *pipeline*: A workflow is the comprehensive model of all designated pipelines of a project. The execution of a concrete pipeline is instantiated from this workflow and depends on a variety of configurable factors (e.g. branch name, commit tags) that are evaluated by the CI/CD provider, especially when changes are published to the source code repository or when triggered by scheduled events (e.g. nightly builds).

Pipelines (cf. Figure 2.4) are composed of one or more jobs, where every job is (preferably) responsible for a single task, like fetching software libraries or compiling source code. They are the elemental entity in this context, since jobs are typically realized as shell scrips, which denies a semantic insight to the CI/CD engine. Jobs of a pipeline form a Directed Acyclic Graph (DAG) based on their dependencies on each other. For example, a job which is responsible for compiling the source code depends on the former fetched libraries.

Nowadays, a workflow is typically serialized in a textual configuration, often in form of a data-serialization language like YAML [13] and stored in the root folder of the project. This follows the general *as-code* movement where additional configuration, like infrastructure specification, is stored as text alongside the actual application's source code [88, 58, 170].

2.2 Large-Scale DevOps

The philosophy of Agile Software Development (ASD) and DevOps favors small teams, in order to reduce the complexity of communication [179, 171]. As a result, software projects that are practicing DevOps must scale by the number of teams. Dingsøyr et al. list in their taxonomy of scale large-scale projects with two to nine teams, and very large-scale projects with more



(a) Monolith

(b) Microservices Architecture

Figure 2.5

than nine teams [42]. For reasons of simplification, this dissertation won't distinguish between large-scale and very large-scale in the further course and use the term large-scale for projects with multiple teams.

When applications mature, features accumulate and teams are growing, the project maintenance and coordination of change requests [7] is steadily increasing. As a result organizations struggle to scale teams in monolithic projects accordingly. In response to the demand of organizational and technological scaling, organizations are migrating applications from monoliths to microservices architectures [36, 127].

Monolithic applications are composed of tightly coupled layers (e.g. Data, Logic and GUI), that are traditionally developed by a single team (cf. Figure 2.5a) and more importantly in this context, packaged and deployed as a single artifact. When practicing small-scale DevOps, the team is responsible for developing and operating the whole stack, from Database to GUI. Although the *modular monolith* approach allows the decoupling of layered monolithic applications into seperated modules with the benefit of "parallel development", the deployment as a single artifact left horizontal scaling remain a challenge [31].

In the context of web applications, microservices architecture became popular around 2015 [7] as an evolution of Service Oriented Architecture (SOA) [98, 100] to restrict services to being 'small' [178]. Microservices architectures enable multiple self-organized teams to work together on the same application (cf. Figure 2.5b), without getting in each other's way. In contrast to the modular monolith, which is often used as an intermediate step when transitioning towards microservices [60], it makes not only "parallel development" but "seperated deployment" [31] possible. A microservice takes over a single use case (e.g. user registration) and provides this service to the whole application via an Application Programming Interface (API), typically

based on the Hypertext Transfer Protocol (HTTP). As long as the developed microservice complies with the API specification, the team is independent in their choice of tools and software stack. As a consequence, applications are not assembled of tightly coupled layers, but are composed of independent and distributed building blocks. As a side effect, microservices architecture fulfills the philosophy of ASD and DevOps for small teams [10, 7]. If performed right, microservices architecture paves the way for large-scale DevOps.

2.3 Container Virtualization

A key technology for the realization of Continuous Practices (CP) in particular and the success of DevOps in general is container virtualization. Containers are being used for lightweight infrastructure provisioning during the whole DevOps lifecycle (cf. Figure 2.3), from local development environments in the creation phase, via the execution of CI/CD jobs, through to containerized deployments and monitoring. Although container virtualization was not a new concept, first the launch of Docker in 2013 gradually established containers as mainstream [48]. Docker is still one of the most influential solutions in this area (cf. [76, 27, 66]), thus the following introduction of container virtualization is greatly influenced by Docker's concept and terminology.

In contrast to 'heavy' virtual machines that rely on emulated hardware, 'light' containers run processes directly in the kernel of the host's operating system [48]. In addition to that, containers require considerably less resources since they are designed to execute just single processes instead of the numerous processes involved in a virtualized operating systems [48]. This resource efficiency allows hosts to start containers from so-called container images, without the overhead of hardware emulation.

In the case of Docker, container images are built from blueprints that are called Dockerfiles. Dockerfiles are textual configurations, written in Docker's own Domain-Specific Language (DSL) [66], and contain the structural design of an image in a sequence of basic statements (e.g. creating directories, copying artifacts or downloading dependencies). Most of those statements create new cacheable image layers on top of each other (cf. *build* step in Figure 2.6). Images, whose Dockerfiles begin with the same commands, can leverage the cache and reuse previously built layers up to the first statement that differs. Built images are persisted in local registries based on the above-mentioned image layers. (cf. *store* step in Figure 2.6).

The starting point of a Dockerfile is usually a statement that references a previously built image. This can be one of the provided base images (e.g. Alpine Linux, Ubuntu), official images of servers or programming languages which provide preconfigured functionality (e.g. nginx, Python), and all kinds of unofficial private or public third-party images. Besides local registries, images can be also pushed to and pulled from remote registries. Such distributed registries provide a convenient way of continuously delivering (cf. Section 2.1) applications including their run-time environment as container images.

Containers are instantiated and run from present container images (cf. *run* step in Figure 2.6). As already indicated, they provide a complete run-time environment alongside the actual application [76], which includes a compatible file system, external libraries or drivers. When instantiated, the container engine adds a seperated and mutable layer for the container on top of the existing and immutable layers provided by the image. This allows the executed process a read-write access to persist arbitrary data. Multiple independent containers can be instantiated from the same image (for example for realizing horizontal scaling) where every container is provided a distinct read-write layer (cf. *Containers* in Figure 2.6) and holds it's own state.

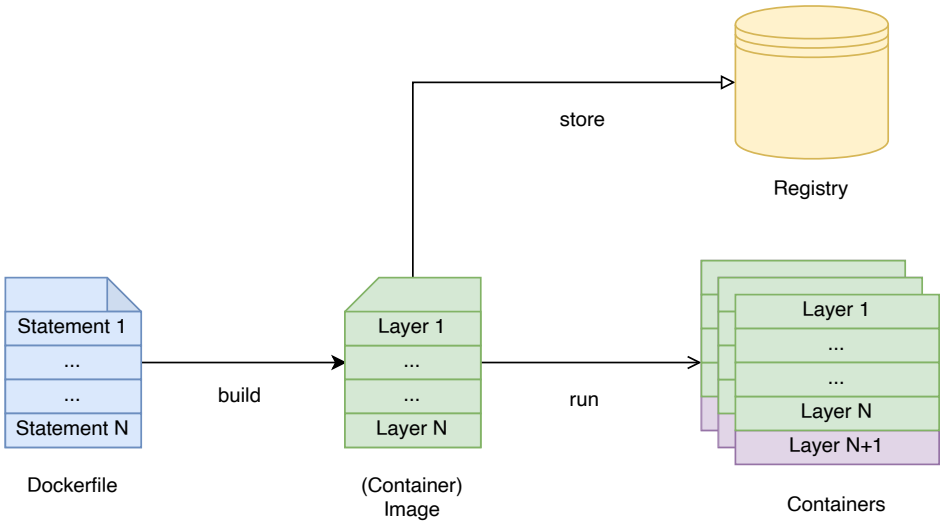


Figure 2.6: Terminology of container virtualization with Docker

In summary, container virtualization is a comfortable method to quickly deliver applications in form of containers. However, modern applications, especially when realized by microservices (cf. Section 2.2), are composed of more than one service. Complying with best-practices of isolating services by container virtualization, will lead to (at least) one container per service. As a consequence, such multi-container applications require orchestration to interoperate properly. This includes the provisioning of resources like processing power, memory, storage and networking. Attempting orchestration manually would be error-prone and not cost-effective in the long run.

Container orchestration systems support the textual definition of multi-container applications and automate their deployment, scaling and resource provisioning. Since Kubernetes has become the de-facto standard for production-grade use cases [27], the remainder of this section will briefly introduce the main terms needed for later chapters. Kubernetes allows engineers to describe applications deployments in a declarative configuration [27]. Intermediate steps (e.g. stopping containers, pulling images, and starting containers), between the current and the desired deployment state are automatically derived and executed. As many *as-code* approaches in the domain of DevOps, Kubernetes configurations rely on YAML.

Kubernetes is designed to run as a cluster composed of multiple physical or virtual nodes [134]. Nodes provide the actual hardware resources and handle the workload of deployed applications. Deployments are not directly realized by containers, but by so-called pods. "Pods [...] are the smallest deployment artifact in Kubernetes cluster" [27]. They allow the creation of tightly coupled groups of containers which share hardware resources and are deployed to the same node [27]. Grouping containers in a pod is needed when container communication cannot be realized exclusively via networking and must run on the same node in order to work together [134]. Containers that are isolated into different pods, exists in their own pod's virtual network and cannot directly communicate to each other. Kubernetes supports network communications to container APIs via services [134], which can be discovered by containers in the private network of the cluster.

Kubernetes and Docker are not only key for automating infrastructure provisioning, but for reducing edge-cases by bridging the gap between local and remote environments [28]. Their *as-code* approach makes operational knowledge explicit and allows, in cooperation with CI/CD, to observe how even complex applications are built, tested, delivered, deployed and orchestrated.

Chapter 3

Knowledge Sharing in Large-Scale DevOps

> "We're on the same team ... now we share the same information and tools."
> — *Eric Shamow* [37]

Since DevOps is not a one-size-fits-all approach to every project [6, 35], software engineers face major challenges while adopting it to their project. Development and operations of a system are more separated than ever before [37]. A successful practice is therefore highly dependent on craftsmanship and skill level of the involved software engineers and system administrators. Tacit or protected knowledge [41], for instance due to knowledge hoarding [124], contributes to the problem and teams run into danger to 'reinvent the wheel'. As a result, the full potential of large-scale DevOps is often not exhausted in reality.

The practice of DevOps postulates *sharing* (cf. CALMS in Chapter 2) and treats it as a first-class citizen in intra-team knowledge exchange, especially between developers and operators. Developers and operators own deep knowledge of their field of expertise, but in order to smoothly cooperate, they also need broad knowledge of their counterpart (cf. Figure 3.1). To successfully 'blur the lines' [136] experts must share their own deep knowledge and create an overlapping broad knowledge. In a sense, developers 'become' operators and vice versa (cf. [73]). The interface between the broad knowledge of both sides is crucial for a successful adoption of DevOps practices (e.g. *automation*) and the key aspect to prevent knowledge silos.

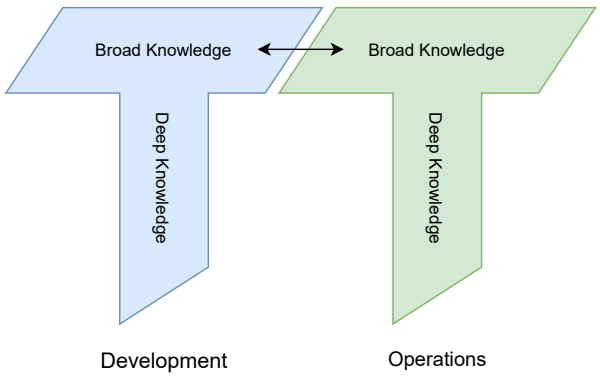


Figure 3.1: T-shaped knowledge in the context of DevOps

Although DevOps aims at breaking down knowledge silos by applying cross-functional teams, the implied independence of a DevOps team inside an organization makes it vulnerable to become a cross-functional knowledge silo itself. Hence, large-scale DevOps is prone to inter-team knowledge barriers (cf. Figure 3.2), which can threaten software projects and whole organizations in the long run. Thus, managing knowledge "is one of the most important things a distributed team can do. Not only does it remove communication barriers among team members, but it also increases the ease and efficiency with which they transfer information." [124]

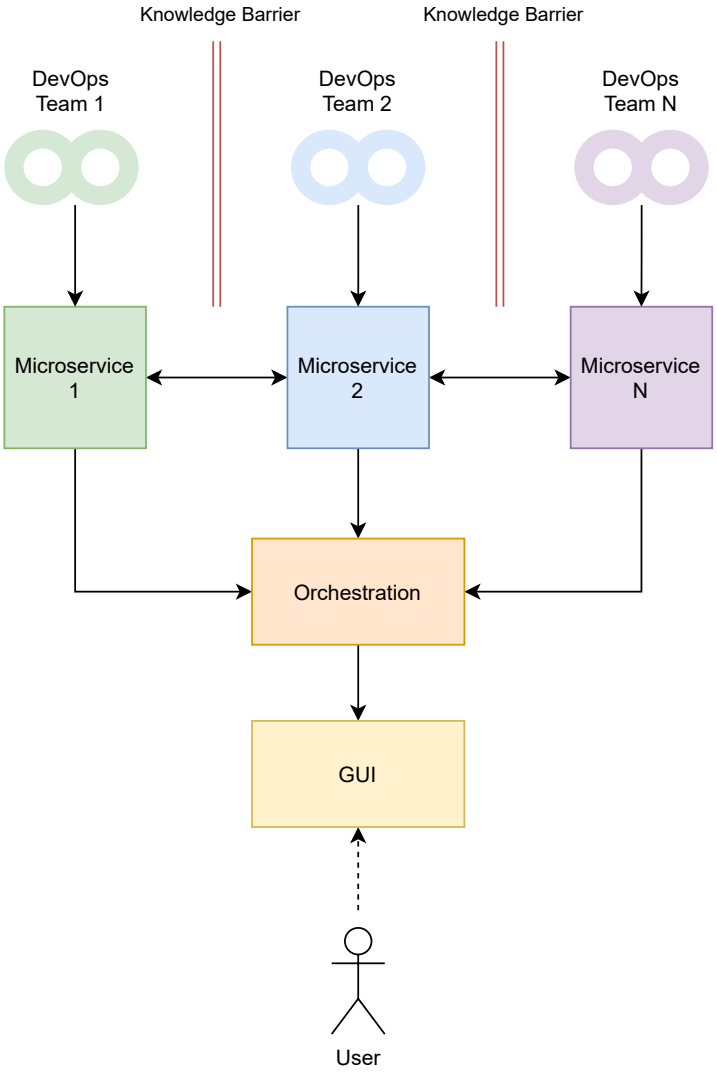


Figure 3.2: Knowledge barrier in large-scale DevOps

This dissertation holds the view that especially large-scale DevOps needs a knowledge culture "that enables and motivates people to create, share and utilize knowledge for the benefit and enduring success of the organization." [118]

Large-scale DevOps has a great potential for applying the codification strategy for knowledge sharing. DevOps teams have to face the same initial difficulties and challenges when adopting the DevOps lifecycle (cf. Figure 2.3). Although existing approaches for general cases can be applied, the challenge is the transfer of related "knowledge from their source to where it is needed" [90]. Especially sharing deep knowledge is not a trivial task, since it requires an in-depth understanding of the involved context [15].

The goal of the codification strategy is to make knowledge explicit [41] by transferring it into a digital format, persist it via databases and provide easy access to the organization [63]. Modern software documentation approaches (e.g. JavaDoc [71] or Doxygen [46]) have adopted this strategy and integrate documentation as part of the software's source code nowadays. Corresponding toolkits create documentation, as standalone software itself, from the annotated source code: Interactive Wikis, for example in the form of web applications, support efficient knowledge discovering and give insight into even the smallest components of a software product.

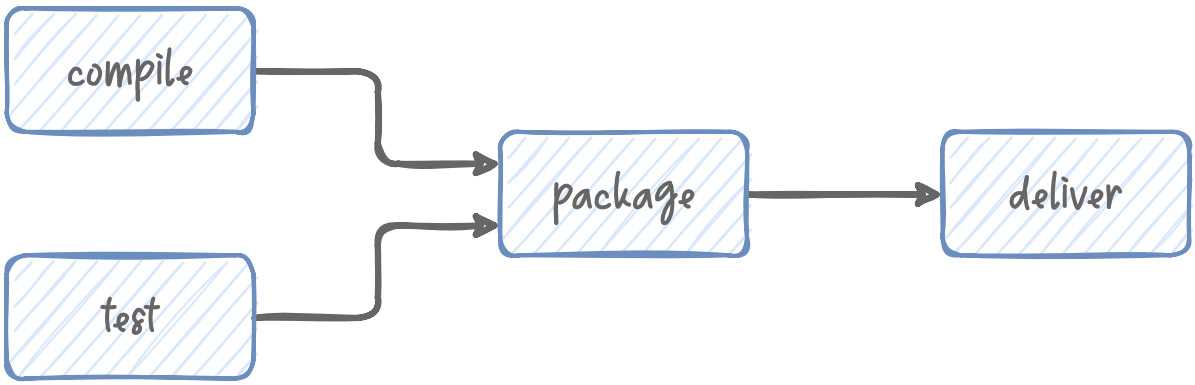


Figure 3.3: Sketch of a CI/CD workflow

Thereby, online accessible documentation has become the norm, while physical copies are more and more unusual [24, 162]. As a side note, codification enables also human and computer readable documentation (e.g. OpenAPI [119]).

Nevertheless, it is a well known problem that documentation is misleading, incomplete or outdated [22, 1]. This roots in individual, cultural, technological and organizational barriers [110]. The "lack of time" [110] is a major individual barrier, since documentation is not prioritized by agile teams. Following the Agile Manifesto [12], their focus lies more on "working software" than on "comprehensive documentation". Although documentation is often a major requirement by product owners, organizations might not understand knowledge as a competitive advantage and do not provide proper IT infrastructure for knowledge exchange.

3.1 Domain-Specific Languages

Domain-Specific Languages (DSLs) have the potential to enable cross-functional collaboration, not only between development and operations on the intra-team level, but to the whole organization on the inter-team level. In DevOps, they have their origin in the provisioning of agile infrastructure [37], but find their way into additional areas of the DevOps lifecycle, like Continuous Integration and Deployment (CI/CD). Most CI/CD providers create custom DSLs, on top of YAML [13], for serializing workflow configurations [152, 154].

In general DSLs are more expressive and easier to use [104, 39], compared to General-Purpose Languages. However, most DSLs in this context are still cumbersome, generally textual and have a steep learning curve [156]. Thus, they fall short of their potential of leveraging abstraction and acting as the interface between broad knowledge and deep knowledge.

When approaching a cross-cutting issue like deployment automation, usually the intensive collaboration between experts is necessary in order to be successful. This requires an overall, but broader alignment on the levels of *why*, *what* and *how*. Experts, especially from different areas, must be able to discuss and to understand each other on both the *why* and *what* level. In other words, they must speak the same language and use the same terminology, while the *how* level is of secondary importance initially. For example, when creating CI/CD workflows it is not yet important *how* they are implemented, but is crucial to understand *why* they are helpful and *what* they are.

First sketches of a CI/CD workflow (cf. Figure 3.3) could even be drawn on a piece of paper to align coworkers and convey the idea of an automated deployment. The concrete realization

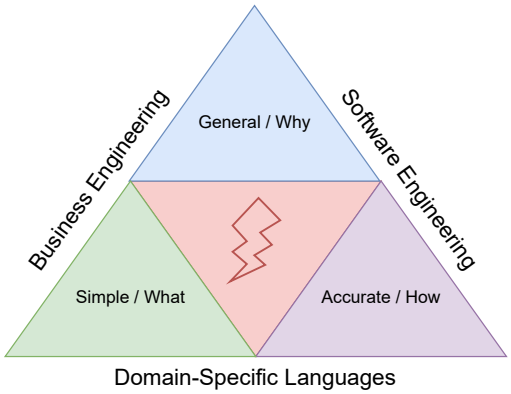


Figure 3.4: Based on Thorngate's tradeoff (cf. [160, 145])

must then be performed manually by specialists and needs the deep knowledge of the underlying technology (e.g. build tools, containerization, etc.). A resulting solution is hardly reusable, and the knowledge that originates from this process is difficult to share.

The barrier between planning (*what*) and implementation (*how*) is a traditional area of conflict (cf. Figure 3.4), resulting from the conceptual difference between Business Engineering and Software Engineering [145]. It is even further attributable to Thorngate's tradeoff between generality, accuracy and simplicity [160], where only two of the tree features can be satisfied at the same time.

Trading generality for accuracy and simplicity [104, 39], enables DSLs to bridge the gap between broad knowledge and deep knowledge in a specific domain. By dismissing the technology-centric view (i.e. *how* level) [96, 140] on a domain, a well-designed declarative DSL grants non-specialist access to the broad knowledge by providing a clear representation of the *what* level (cf. [146]). Code generators that are attached to a DSL enable fully-automatic compilation from the *what* level to the *how* level. The deep knowledge of the specialist is now materialized in the code generator [94] and allows the reuse or sharing of the knowledge. The interplay of language and generator solves the problem of out-dated documentation since instances of a DSL are both documentation and executable artifact at the same time.

The development of a DSL is still considered an expensive or difficult task [104, 78, 39, 113]. In order to be able to develop a DSL, not only domain knowledge but expertise in language development is necessary [104]. Furthermore, economic reasons have to be factored in before creating a new DSL [104]. DevOps at large scale justifies the upfront investment in language development for practices of the DevOps lifecycle (cf. Figure 2.3), especially when those practices, such as automation, can be generally applied to most projects.

3.2 Lingualization Strategy

This dissertation proposes a rigorous approach to knowledge sharing in the domain of the DevOps lifecycle, by applying the ideas of Language-Driven Engineering (LDE) towards language-driven knowledge sharing. It culminates in a *lingualization strategy* which goes far beyond the codification strategy for knowledge. Instead of 'just' storing knowledge (e.g. as documentation or in databases), the lingualization strategy aims at bringing knowledge to life through plain and simple Domain-Specific Modeling Languages (DSMLs).

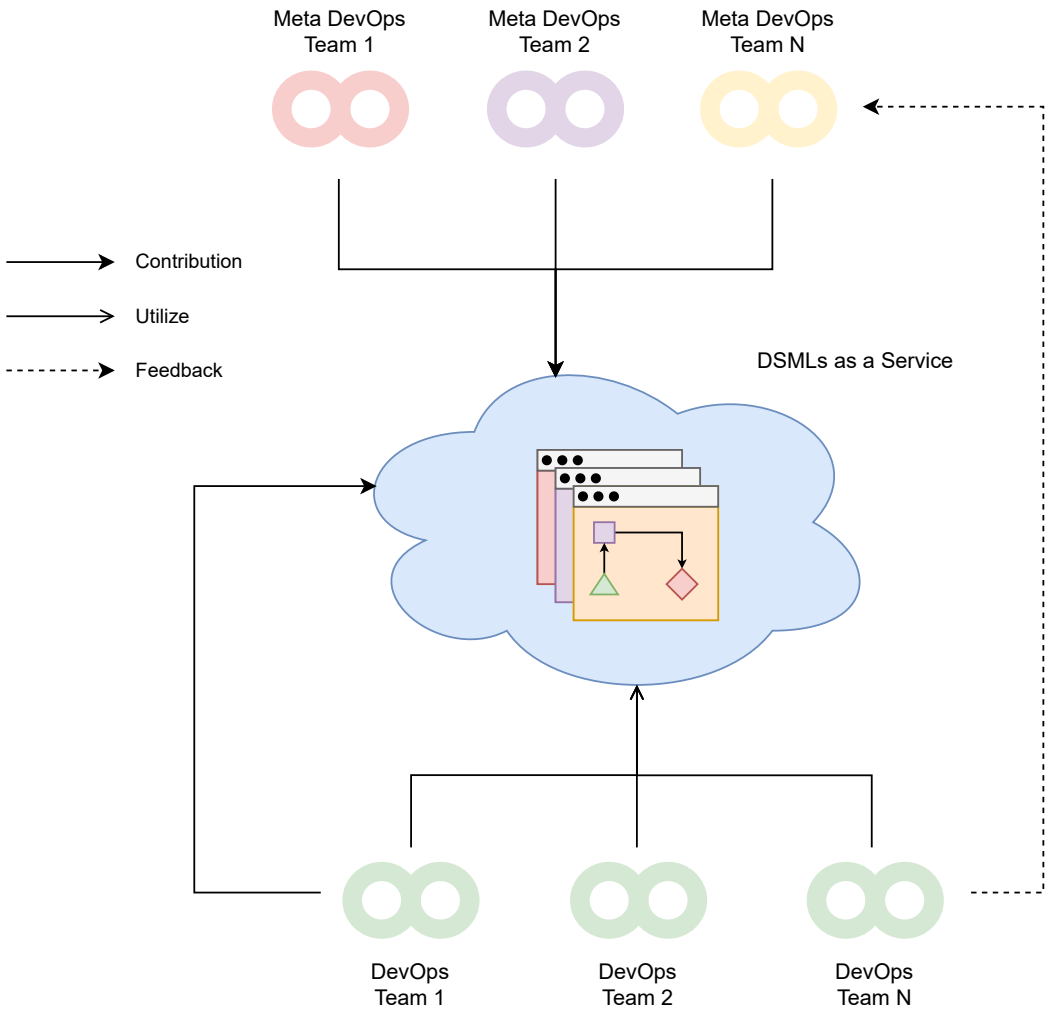


Figure 3.5: Providing *DSMLs as a Service* in large-scale DevOps

The term *lingualization* is generally defined as the "act of representing something as a text in some language" [59], but the strategy presented here considers not just textual, but in particular graphical languages as an excellent way of representing knowledge. Several positive characteristics, that are beneficial for knowledge sharing, are awarded to graphical models in contrast to textual notations (cf. [74]): They have a potential for greater expressiveness [84], flattening the learning curve [135] and allow humans to 'process' their representation in parallel [108].

Nevertheless, while the following will focus almost entirely on DSMLs, which in contrast to traditional textual DSLs make use of graphical models, the decision whether implementing a textual or graphical language should be solely based on the actual use-case. [AP II: [153]] shows how a family of textual DSLs were developed on-demand in an industrial setting to reduce verbose and reoccurring boilerplate code of a web-based application. An organization-wide adaption to other projects successfully replicated the former results and showcased the idea of knowledge sharing through DSLs between teams. Another example is demonstrated in [AP III: [139]] where a textual DSL extends GraphQL to guarantee type consistency.

Key to the lingualization strategy is a profitable language development which keeps knowledge sharing at low cost [133]. Like knowledge management, language development should not be an isolated task (cf. [63]), but a global objective of organizations practicing large-scale DevOps. Language development should be considered the prioritized way of sharing knowledge. As the

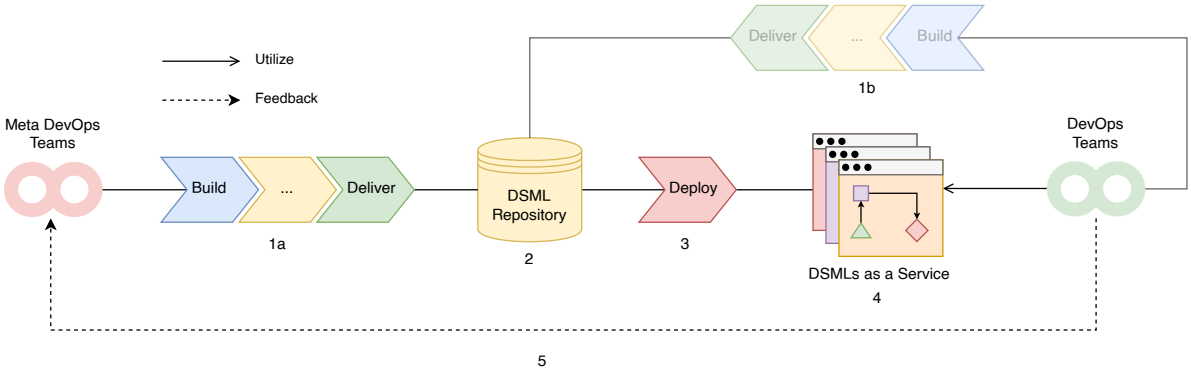


Figure 3.6: Detailed workflow of providing *DSMLs as a Service*

aforementioned characteristics promise, sharing knowledge through DSMLs is an efficient way to learn from each other (cf. [37]). They are comprising the *what* and *how* level. If treated right, DSMLs could even become first class citizens in the development environment and important building blocks of their DevOps lifecycle/ecosystem.

In reality DevOps teams already take on greater responsibilities than traditional software development teams (cf. Chapter 2). They face not only the development of software anymore, but are supposed to cover operational tasks as well. Thus, their focus lays primarily on the development of maintainable and operable systems. While they are successfully practicing intra-team knowledge sharing, inter-team knowledge sharing might be interpreted as an additional burden. Even with the reduced cost due to LDE, such small and specialized teams would be overloaded with the task of language development. The organizational structure has to be reconsidered and must represent the importance of the lingualization strategy of knowledge sharing as a competitive advantage (cf. Figure 3.5).

Dedicated Meta DevOps teams can unburden the existing DevOps teams and meet the demand of LDE to create "a new class of stakeholders" [146] for language development. Meta DevOps teams do not participate in the development of end-user applications, but it is their task to enable inter-team knowledge sharing in large-scale DevOps by applying the lingualization strategy. Knowledge that is worth to share and compatible with the lingualization strategy will be cast into a DSML and published organization wide. Thus, knowledge is not just accessible to all teams in the organization but directly reusable thanks to the interplay of DSML and code generator.

Since Mernik et al. defined *decisions*, *analysis*, *design*, *implementation*, and *deployment* as the DSL development phases [104], it is a natural fit that Meta DevOps teams practice DevOps as their software development methodology. For a more detailed workflow, cf. Figure 3.6: Meta DevOps teams plan new features of their maintained DSMLs based of the continuous feedback (cf. 5 in Figure 3.6) from language users, e.g. feature requests. Feedback is stored as user stories and analysed to drive the future development. Using automated CI/CD workflows, new versions of a DSML are continuously built and delivered (cf. 1a in Figure 3.6) to a centralized DSML repository (cf. 2 in Figure 3.6). This organization-wide repository allows early adopters (cf. [102]) and testers to try new versions before the actual release of a DSML. Stable versions are deployed (cf. 3 in Figure 3.6) as a service and made available for regular use (cf. 4 in Figure 3.6) to all DevOps teams.

Meta DevOps needs cross-domain experts that have experience in projects driven by DevOps, to be aware of needs and requirements, and language developers that know how to implement

and deploy DSMLs. Nevertheless, regular DevOps teams are not excluded from participating in the knowledge sharing process by providing DSMLs. Advanced DevOps teams can deliver their own DSML using the same workflow (cf. 1b in Figure 3.6) as Meta DevOps teams.

3.3 Purpose-Specific Languages

Up to this point, this dissertation uses the common terms of DSL and DSML to describe the class of 'programming' languages that have the potential for a lingualization strategy. DSMLs adopt terms of their target domains and make them an integral part of their language. This domain specificity can already have a positive impact on sharing knowledge through models [49], but if the 'bounded context' [49] is not well-defined, a DSML runs into danger to gradually cover too many aspects of its target domain and thus, looses an initial simplicity. Following the LDE approach the lingualization strategy focuses on the more narrow term *purposes* [151].

For instance, one could picture the DevOps lifecycle and its practice of continual improvement [75] as a domain of interests. In the wild, working DevOps lifecycles are often improvised [18] and usually a composition of numerous technologies to realize the consecutive phases from planning to monitoring a software system. This involves knowledge of scripting languages, protocols, virtualization and so on, which are forged together by manual labor. The deep knowledge in different areas and a close cooperation of specialists is needed to make it work. An attempt to cover this lifecycle to its full extent in a single DSML is almost an impossible task.

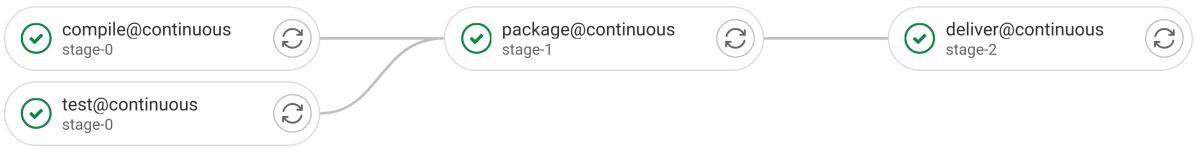
Approaching such a broad domain and trying to impose a comprehensive DSML on mature DevOps teams would be met with individual resistance and technical difficulties, since "people and processes don't change overnight" [37]. The massive impact on the already working DevOps lifecycle would come close to a *big bang adoption* [17], with all the included drawbacks. In order to perform a successful lingualization strategy, DSML adoption must be a non-intrusive task.

Following LDE and it's practice of *divide and conquer* [85], one can discover numerous of *purposes* inside a *domain*, like DevOps, and create multiple Purpose-Specific Languages (PSLs) instead of a single DSML. A DSML tries to cover a complete domain while a PSL serves just a single purpose. PSLs are simpler and their focus is much narrower than DSMLs, which benefits the knowledge sharing properties (i.e. 'less is more').

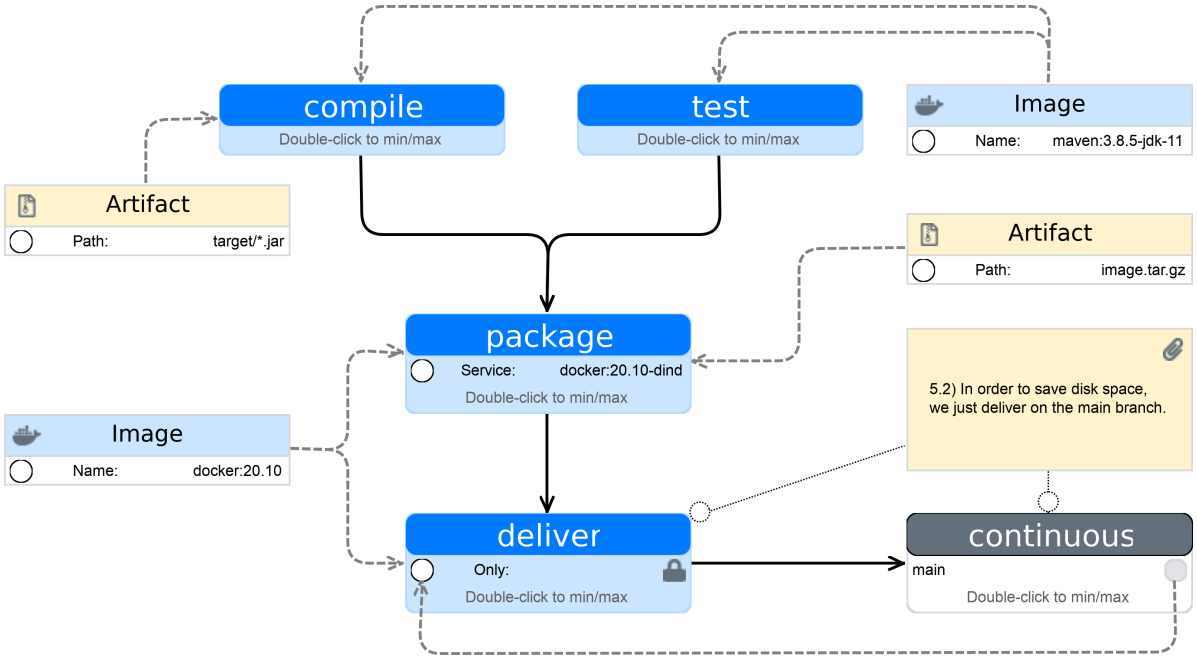
In this context, PSLs are in a sense a materialization of knowledge [94]. Instead of extracting information from codified knowledge and manually applying it to a current case, PSLs bridge the gap between *what* and *how* level by an explicit visualization and automated code-generation. The domain knowledge or respectively purpose knowledge is embedded [126] in a concrete PSL, and thus becomes independent of domain experts. During the development process of a PSL knowledge is transferred from source to destination, which is a major challenge of general knowledge management [90].

[AP VI: [151]] establishes the term *executable documentation* for the dual benefit that results from a well-designed PSL that is accompanied by a code generator. Key for such an executable documentation is the understandability of a PSL that allows experts from different backgrounds to reason from a concrete model and draw conclusions for their responsibility in the project. PSL-based executable documentation has the potential to improve the overlapping communication between different stakeholders, and to act as an interface with other organizational units [171].

A tangible example of applying executable documentation in the context of DevOps are CI/CD workflows. As already argued in Section 3.1 and illustrated in Figure 3.3, CI/CD workflows can



(a) GitLab's graphical notation of an exemplary CI/CD pipeline



(b) CI/CD workflow based on the graphical notation of Figure 3.7a

Figure 3.7

be easily drawn on a piece of paper to outline its concrete graph structure. CI/CD providers make use of this trait to document the outcome of executed CI/CD pipelines. Figure 3.7a displays GitLab's graph visualization of an exemplary pipeline, that represents the sketch of Figure 3.3, with four connected jobs in which all jobs have been successfully completed. The graph is read from left to right: While the jobs *compile@continuous* and *test@continuous* can be independently executed in parallel, *package@continuous* and *deliver@continuous* are premised on the result of previous jobs.

[AP VI: [151]] illustrates how this domain-specific notation of GitLab's visualization can be turned into a fully-fledged modeling language that not only supports the generation of CI/CD configurations, but also the graphical documentation of the modelled workflow. Figure 3.7b shows a model that reflects this visualization and especially addresses the execution order of the including jobs via directed edges between the nodes *compile*, *test*, *package* and *deliver*. The workflow is modelled in a graphical PSL that laid the foundation of the Rig project [AP I: [152]]. Chapter 4 will introduce Rig and further elaborate its PSL.

Such executable documentation benefits from the clear visualization and supports the knowledge sharing by allowing to include models like Figure 3.7b as graphics directly in textual documentation. In order to live up to those expectations, PSLs must avoid complexity by aiming at declarative languages that communicate on the *what* level and hide technical or im-

(I) "Make each [PSL] do one thing well. To do a new job, build afresh rather than complicate old [PSL] by adding new features."

(II) "Expect every [PSL] to become integrated by another, as yet unknown, [PSL]. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input."

(III) "Design and build [PSLs], even [mIDEs], to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them."

Figure 3.8: Unix Philosophy [102] Applied to Language-Driven Engineering

plementation details, in other words the deep knowledge, in form of the *how* level. This can be achieved by following LDE's key features *simplicity*, *service orientation* [95], and the *one-thing approach* [147, 95]. The Unix philosophy (cf. Figure 3.8), which was actually intended for implementing traditional programs [102], endorses the mindset behind LDE and succinctly expresses the essential rules for the lingualization strategy presented herein.

Simplicity is not just a characteristic that positively affects software maintainability from the developer perspective [106], but impacts the success of applications or programming languages, when compared to competitors, by an ease of use [105]. While especially enterprise software tent to become more complex over time [23], it is the aspiration of lingualization strategy to steadily provide easy to grasp PSLs, which are custom-tailored to the user's needs. In order to not lose simplicity over time, LDE offers two ways of achieving modularity for the language user. A horizontal composition allows multiple PSLs to be integrated into each other, while the vertical refinement transforms PSLs to more specific abstraction layers [146] and allows a step-wise code generation to executable code along those layers [180].

This means for Meta DevOps teams, to already keep in mind while planing a PSL, that it does not live in solitary. It is imagined as a service, to other components of a bigger system [98] or, as already mentioned, horizontally integrated respectively vertically refined in the spirit of LDE [146, 180]. Therefore, it is crucial to provide a well-designed and preferably stable interface so that other components can use a PSL as a service. It is important not to over specify a PSL to a particular issue, but maintaining a degree of generality, so it can be "used a lot and solve a lot of problems" [39]. Finding the sweat spot between (custom) tailoring and reusability will remain an ongoing challenge of Meta DevOps teams which emphasizes the need for a cheap and fast language development.

In addition to simplicity and service orientation, the one-thing approach reduces complexity by avoiding inconsistency. It advocates for a single source of truth during the system's lifecycle [95]. A central artifact, the 'one-thing', e.g. in the form of a model, serves as the object of observation to all stakeholders. Meeting the demand for simplicity, the artifact should be understandable only with broad knowledge of DevOps team members. All changes are made to the artifact and the implementation is generated from it. Furthermore, a PSL must allow the serialization of the 'one-thing' to support source control as first-class citizen, regardless of whether it is a textual or a graphical language. Repositories enable the efficient collaboration in a DevOps team [35], so even graphical languages should support a fitting serialization format (cf. GraText [92], MCaM [172]). Since the most conventional version control systems aim at text-based documents, binary representations are disapproved. This textual representation must be a compatible input to the code generator of the PSL, without any further preprocessing

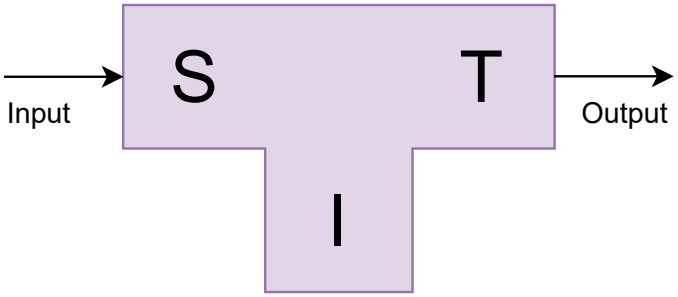


Figure 3.9: The interplay of the three languages (S,I,T) in a code generator (cf. [74, 68])

to support fundamental DevOps practices like CI/CD.

3.4 Code Generators

Similar to executable DSLs which can be interpreted or compiled [21], PSLs are shipped with associated code generators that translate instances of a PSL to executable artifacts. Code generation can be approached and realized in different ways (cf. [74] for a detailed list), but in general, it involves three (programming) languages [74]. Their interplay can be represented in a T-diagram (cf. Figure 3.9): Code generators take the source language S as their input, are implemented by a language I and produce an output in a target language T.

Code generators abstract from technical details [74], and take away issues of the target language which are only known to those who own deep knowledge in that field. They contain the essence of experience on the *how* level (i.e. failings and successes) of domain experts and let others benefit from it. [AP IV: [154]] addresses the "Norway Problem" in YAML [116], where the Alpha-2 Code [70] of Norway '`no`', if represented as an unescaped string, is interpreted as the boolean value `false`, and shows how the code generator of Rig solves such issues related to the target language.

This example illustrates, that code generators perform the 'heavy lifting' of knowledge reuse and that the division of labour between different stakeholder is the corner stone of the lingualization strategy. While PSL developers are very familiar with inconvenient traits of the target language, PSL users do not have to be aware how they are avoided by the code generator and can solely focus on realizing solutions by using the source language. In other words, Meta DevOps teams materialize deep knowledge of the domain in the PSL, but even more importantly for a consistent and convenient usability, in the code generator. When issues arise, Meta DevOps team can solve them once and for all on the meta level (cf. [148]) and easily provide the fix to the DevOps teams using CI/CD. So instead of manually migrating knowledge from codified documentation to their use-case by reinventing the wheel, DevOps teams gain advantage from the materialized knowledge of PSLs and code generators. Thus, knowledge sharing by the lingualization strategy accelerates not only the DevOps maturity of existing teams by adopting more phases of the DevOps lifecycle, but gives newly formed teams a headstart when introducing DevOps from scratch.

A distinctive feature of LDE is the more or less unrestricted numbers of code generators and their interoperability to achieve simplicity for the language user. While the realization of horizontal (composition) and vertical (refinement) interoperability [146, 180] is beyond the of scope of the dissertation, it impacts the lingualization strategy in the following way. The simplicity through modularization makes it convenient for DevOps teams, but potentially increases the

development effort for the Meta DevOps teams. In order to keep the development effort low, code generators for the lingualization strategy must comply to global standards. Sven Jörges lists in [74] five general and six specific requirements for code generators that serve as an ideal baseline. The following discusses the most important requirements here and highlights their value in the light of DevOps.

One major concern of practicing DevOps is the requirement of tested software. In line with this, code generators should support *validation* by means of testing as a first-class citizen. Testing must be straightforward, locally on development environments and remotely during the Continuous Integration (CI) phase. It may sound like an obvious requirement, but deterministic code generation is key for unit testing, so that it cannot be stressed enough. Especially multithreading should be handled with care to not end up in non-deterministic results. A convenient strategy to meet this demand is pure generators (cf. pure functions) were the generator has no side effects, and the generated code is only dependent on the input by the source language. Maintaining test cases for discovered issues in combination with automated test execution ensures that bugs are not reintroduced due to regressions.

Essential for exhausting the testing possibilities of code generators beyond unit testing, is the concept of *full-code generation*. A concrete model contains all necessary information to generate fully functional code [112], without any round-tripping or user interaction. While partial code generation, would also support unit testing based on the generated code as return value, only fully functional and executable code allows system testing of functional and non-functional requirements.

When Meta DevOps teams start to build upon existing code generators (by horizontal composition or vertical refinement), they require a straightforward *variant management* and distinct *product lines*. They must count on the stability of released versions and should be informed about breaking changes in their interface. A common approach to encode the compatibility of new releases (e.g. for software libraries), is semantic versioning [125]. Version strings are composed of three parts called *major*, *minor* and *patch* (e.g. 3.1.2), each one with a different meaning. While *minor* and *patch* represent backward compatible changes, an increment of *major* indicates incompatible modifications.

Generated code is usually a black box to a language user. Nevertheless, a code generator should aim at producing *receiver oriented code* that complies with common style guides of its target language. While this is not a trivial task, due to the narrow focus of PSLs, it's not beyond the realm of possibility. Providing receiver oriented code does not end in itself, but it eases testing and further processing by software development kits of the target language. Static code analysis, like linters, can produce warnings based on low-quality code and help to spot issues in the generator implementation. Most of those issues arise from template-based generation (cf. [74, 84]), where code is directly generated as a stream of characters by the interplay of implementation language and target language. An alternative approach makes use of transformation rules [74, 84, 87]. Instead of generating source code directly, first the generator transforms the input model of the source language to an output model of the target language (e.g. abstract syntax tree) and second makes use of established methods in the context of the target language to serialize the concrete code [74, 84]. In the context of receiver oriented code, this approach is advantageous, since a valid model of the target language already represents a syntactically correct program. Furthermore, if the source code representation is not necessary, the output model representation allows a direct compilation of the program without previously parsing.

In general, code generators must be expected to be executed in *tool-chains* like CI/CD pipelines

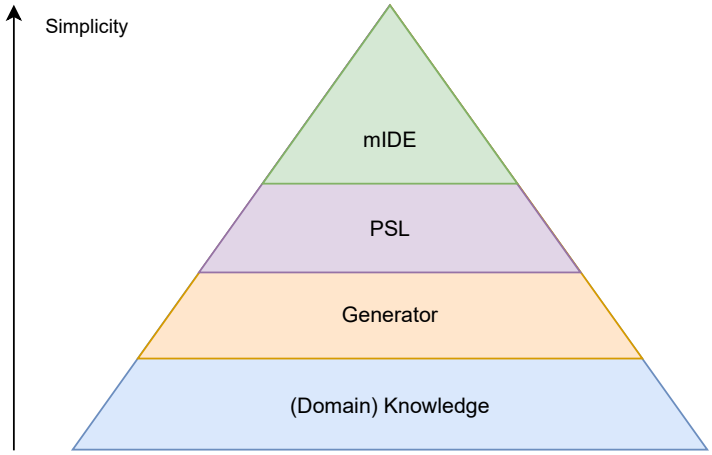


Figure 3.10: Stepwise simplification of domain knowledge by LDE

to enable automation in terms of DevOps. Container images are a suitable way to release code generators and support headless generation (i.e. without a graphical user interface), since CI/CD is often realized by container virtualization (cf. Sections 2.1 and 2.3). In combination with the aforementioned requirements, automated tool-chains contribute to a rapid, but reliable maintenance of code-generators, which are important for such a service-oriented approach.

3.5 mindset-supporting IDEs

Using a textual DSL or a general-purpose language, requires just a text editor to write source code, and a compiler to create an executable program from the written source code. However, developing applications this way is a tedious task which doesn't meet the requirements on cost, controllability and effectiveness (cf. [115]) of modern software development. In the worst case, the text editor is even completely unaware of the used languages' syntax, and the compiler is the only mechanism to provide feedback to the developer.

In order to increase the productivity, Integrated Development Environments (IDEs) assist developers during development time [67] before the code is compiled. The functionality reaches from basic features like source code highlighting and folding, to more sophisticated support, like autocompletion and coding assistance [67]. Apart from data modeling with e.g. entity–relationship model IDEs typically aim at general-purpose languages that are represented as text.

Steffen et al. coined the term of mindset-supporting IDEs (mIDEs) in [146] as a special form of IDEs, that support the development in a specific domain or for a specific purpose and put the mindset of domain experts in the foreground. The flat learning curve of using an mIDE which *speaks the language* of the domain experts, lets even novices participate during the software development process. It comes shipped with extensive domain-specific tool support, which goes beyond typical IDE support, to assist in using the PSL and executing the code generation.

An mIDE is paired with a corresponding PSL and code generator (cf. Figure 3.10). Actually, it can be directly derived and generated from the meta-level specification of the PSL [113], which will be addressed later in the dissertation (cf. Section 4.1). Meta DevOps teams are responsible for the development of the whole stack and publish the mIDE as turnkey applications, which doesn't require additional setup of e.g. the code generator.

In contrast to typical desktop applications, mIDEs in large-scale DevOps have specific non-functional requirements on modularity, setup and resources. The seamless integration into established tool chains is an absolute necessity for a successful adoption. To not hamper the integration, the stack must be *modular* and loosely coupled. Language server (cf. [25]), code generator and even the GUI should be decoupled in independent modules. Besides publishing them as a bundle in an mIDE, those modules could then be deployed distributed and scaled independently. Hence, providing them as e.g. container images with simple interfaces, would enable the reuse of single components.

In order to standardize the provisioning of mIDEs, their implementation and architecture should follow requirements related to the term cloud computing [103]. Delivering them as fat binaries or as container images allows both local execution and remote deployment for inter-organization usage. Web-technology lets full-blown applications run in a browser almost instantly with zero setup [40] and enables *mIDE as a service*. It allows users to try mIDEs without the burden of a complex installation [40].

For a dynamic use of PSLs in a software project, it is essential that mIDEs are *lightweight* and do not stand in the way of the daily task. When working on independent, but interacting PSLs in a service-oriented manner, engineers will use multiple mIDEs at the same time and could run into resource limits. This is why mIDEs should have a fast boot-up time and their resource footprint should be as small as possible.

3.6 Organizational Infrastructure

In [118] Oliver and Kandadi list IT infrastructure as one of ten key factors for the success of organizational knowledge sharing. In this context IT infrastructure comprises both standard communication technology (e.g. videoconferencing software) and more specific solutions to knowledge sharing, like knowledge portals. Knowledge portals are central platforms for easy browsing and retrieving codified knowledge [118]. Employees have access to a variety of tools that simplify the work with the data. In order to support such major challenge, proper IT infrastructure is needed on an organizational level [110]. Knowledge portals that are open to the whole organization might have also a positive impact to prevent knowledge hoarding [118].

A comprehensive lingualization strategy for knowledge sharing has an equivalent, maybe even greater, demand for IT infrastructure. Thus, the organizational overhead to provide this infrastructure is not to be neglected. While knowledge portals are often *just* operated by organizations so that employees can enter and access data, the lingualization strategy includes at least the complete development cycle of mIDEs, PSLs and code generators. Besides the organizational structure, with dedicated Meta DevOps teams (cf. Figure 3.5), IT infrastructure plays an important role to enable knowledge sharing. Implementing a DevOps-based lingualization strategy requires an interplay of multiple services (cf . Figure 3.11):

An important requirement of practicing DevOps are automated and reproducible builds of the system (cf. [38]). In order to enable this, the source code repository must contain all information that is needed for the build, which resembles the philosophy of the one-thing approach (cf. [147, 95]). This doesn't mean that every dependency is stored as source code, but that a pointer is present (e.g. pom.xml [3]) how to fetch dependencies during the build in the correct version. The source code repository is the main work object for the teams and every change to the source code of mIDE, PSL, code generator, etc. is committed there. This enables CI, which automatically builds and tests changes on centralized environments.

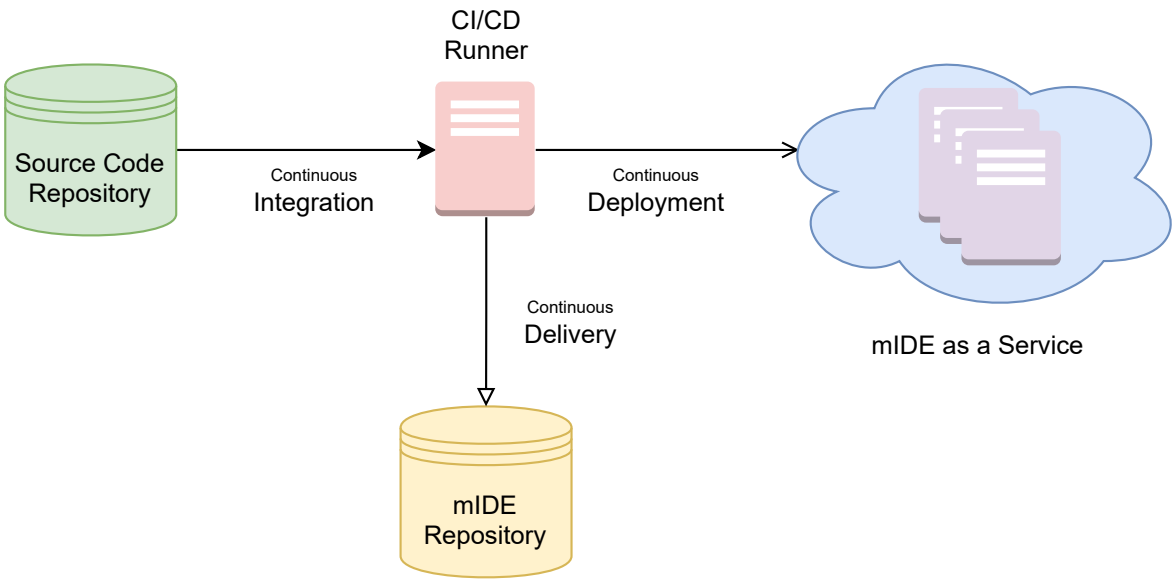


Figure 3.11: IT infrastructure to continuously provide *mIDEs as a Service*

Such centralized environments can be called *CI/CD Runner*. The realization requires dedicated machines whose only purpose is the execution of CI/CD tasks, like building, testing and deploying. Based on virtualization technology (e.g. by containers or virtual machines) reproducible environments with all necessary software development kits can be instantiated on demand. The definition of specialized environments which differ from standard solutions, should also be available to team members and versioned alongside to the applications' source code. In case of Docker, this can be achieved by storing related Dockerfiles in the source code repository and build custom container images through CI/CD in the same way as the application.

A central building block is the *mIDE repository* that contains all compiled artifacts that are ready to use. It stores the available mIDEs and makes them accessible organization-wide. The mIDE repository represents the organizational common knowledge (cf. [133]) that was discovered by the lingualization strategy. Such mIDEs are released by successful run of Continuous Delivery (CDE) pipelines. Following the demand of modularization (cf. Section 3.5), the repository doesn't store just mIDEs, but all the decoupled modules for independent usage. For instance, such repositories can be realized by Maven repository managers [3] in case of Java-based modules, or by Docker registries [44] for containerized modules.

In order to make the adoption of PSLs easier, the lingualization strategy provides *mIDEs as a service* to the whole organization. Continuous Deployment (CD) pipelines are deploying compiled mIDEs from the repository to a central private cloud infrastructure. The modularized architecture of mIDE, for instance as container images, require comprehensive provisioning and orchestration (cf. Section 2.3), which can be realized by applying Kubernetes. Employees can then use their corporate identity (i.e. single sign-on) to access the web-based service and start using the PSL without additional setup.

[AP V: [181]] sketches the concept of Cloud mIDEs which has been further elaborated in the dissertation of Philip Zweihoff [180]. Cloud mIDEs are web-based distributed environments and feature real-time collaboration between different stakeholders through custom-tailored PSLs. The centralized infrastructure based on cluster technology, allows the seamless transition between development and runtime environment. This concept lays the technical foundation for the here presented mIDE as a service approach.

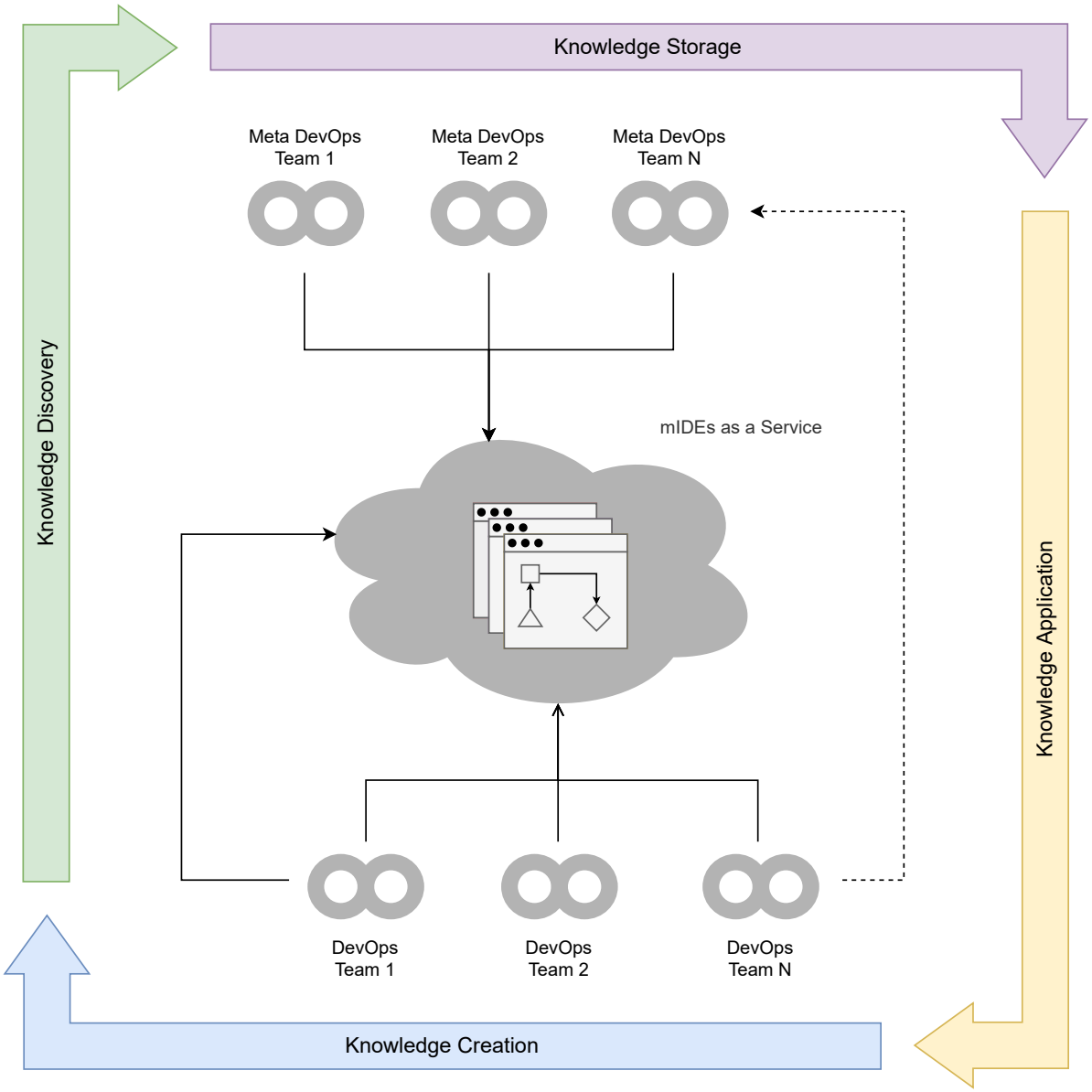


Figure 3.12: The four knowledge sharing activities of the lingualization strategy

3.7 The Knowledge Sharing Process

Ding et al. describe in [41] how the activity of knowledge representation in software documentation can be mapped to a general knowledge management process. Figure 3.12 shows, inspired by their mapping, a less complex classification of activities and illustrates how the four activities can be applied to the lingualization strategy as an interplay of producing and consuming knowledge. The knowledge sharing process of the lingualization strategy takes place on the organizational level and forms conceptually a continual improvement cycle (cf. [75]).

1. **Knowledge Creation:** Practicing DevOps is an ongoing process that has to be adapted to emerging functional and non-functional requirements. So even mature DevOps teams are compelled to continuously improve their practice, and gain more-and-more experience on a daily basis, which is beneficial for less mature teams.

When novel teams start with the DevOps methodology and adopt first lifecycle phases, it offers a precious opportunity to observe initial difficulties. Especially the impartiality of novices in this context allow a neutral point of view, which is value knowledge for advancing and innovating established practices.

The daily experience *creates* both explicit and tacit knowledge.

2. **Knowledge Discovery:** Meta DevOps teams have the responsibility to *discover* this knowledge from mature and novel DevOps teams. As its name implies, explicit knowledge is easier to discover than tacit knowledge. Solutions following the *as-code* approach, for example, contain explicit knowledge and can be discovered by accessing related source code repositories.

   The challenge lies in the identification of tacit knowledge. In contrast to explicit knowledge, discovering tacit knowledge requires personal interaction, since it "resides in people's head and is not easily visible and expressible" [41]. This can be realized by interviews or by temporary participation of Meta DevOps team members in the daily work of DevOps teams. Ideally, DevOps teams do not stay passive in this context, but act as active companions and call attention to knowledge that they consider worth sharing.

3. **Knowledge Storage:** Meta DevOps teams *store* the discovered knowledge by extracting concrete *purposes* that can be implemented by single PSLs. If discovered knowledge turns out to be rich of purposes, implementing a family of PSLs is preferred to a single comprehensive PSL.

   Following the lingualization strategy, the knowledge is stored or in other words materialized as a pair of PSL and code generator. As illustrated in Figure 3.10 and described in Section 3.5, custom-tailored mIDEs on top of PSLs simplify the modeling procedure. The knowledge storage process is illustrated by Figure 3.6 in more detail from a workflow perspective.

4. **Knowledge Application:** Providing the mIDEs as a service allows DevOps teams to first test PSLs and evaluate if these serve their use-case without additional setup, and thus finally *apply* materialized knowledge that otherwise might have stayed undiscovered.

   Meta DevOps teams have to make sure that DevOps teams are kept updated of discovered knowledge which is available in form of brand-new PSLs or improved versions with new functions. This requires changelogs, newsletters, chat rooms and public communication in the organization.

Naturally, knowledge sharing doesn't end with providing a PSL, but is an ongoing process of monitoring, developing, and deploying. In the context of PSL this resembles the *path-up/tree-down* concept, presented in the dissertation of Steve Boßelmann [19]. During the discovery activity, the knowledge moves the path from a DevOps team, which was responsible for the knowledge creation, up to a Meta DevOps team. After the storage activity is concluded, the knowledge travels the tree from the Meta DevOps team, which was responsible for the knowledge storage, down to all DevOps teams.

Chapter 4

Graphical Modeling with Rig

> "You build it, you run it."
> — *Werner Vogels* [117]

This chapter addresses the theoretical discussions from the previous chapters alongside an exemplary, but real-world application called Rig. Rig [154, 155] is an mindset-supporting IDE (mIDE) and graphical Purpose-Specific Language (PSL) for modeling Continuous Integration and Deployment (CI/CD) workflows. 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" [129] and the verb *to rig*, "to put in condition or position for use" [128]. Rig targets the phases *Continuous Delivery* (three) and *Continuous Deployment* (four) of the DevOps lifecycle as shown in Figure 2.3.

The initial idea of modeling CI/CD workflows in a graphical PSL has been published in [AP I: [152]] and fully implemented by Sebastian Teumert as part of his Bachelor's thesis [156]. Rig has been publicly introduced during the *6th International School on Tool-Based Rigorous Engineering of Software Systems* [97], where workshop participants had the opportunity to graphically model CI/CD workflows. The associated introduction to Rig, including a hands-on example of CI/CD for the TodoMVC application [120], can be found in [AP IV: [154]].

Before CI/CD providers introduced the possibility to configure workflows in a textual form, early build automations were typically managed through dedicated Graphical User Interfaces (GUIs) [58]. Such automations where isolated from the application's source code and thus hidden from ordinary developers.

Inherited from the general *as-code* movement, the introduction of textual workflow configurations (cf. Section 2.1) brought several advantages to the practice of CI/CD [170]. Instead of browsing a complex GUI to find the correct view for a certain setting, a textual representation allows the entire configuration to be stored in a central place. Furthermore, a text file can be stored, versioned and changed in the same way as other source code. This does not just ease collaboration or auditing in general [170], but enables knowledge sharing [58] between operation and development.

Although this type of configuration can already be labeled as domain-specific, it lacks basic qualities of a Domain-Specific Language (DSL), like simplicity and accuracy (cf. [104, 54, 39]). Workflows in this context are typically configured in cumbersome formats (cf. Section 2.1) which are created based on languages for data serialization, e.g. YAML [AP IV: [154]]. A more user-friendly textual PSL could already improve considerably the language-user's experience, but a graphical PSL can bring CI/CD, as the corner stone of practicing DevOps, to its full potential.

[AP I: [152]] assumes that graphical models have the ability to share CI/CD knowledge and bridge the gap between stakeholders, both on an intra-team and inter-team level. Although text

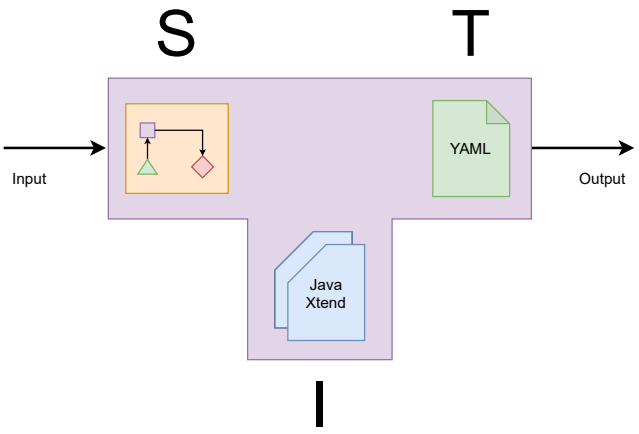


Figure 4.1: T-diagram of Rig's code generator (cf. Figure 3.9)

files are still ubiquitous in this context, CI/CD providers like GitLab recently start to visualize workflows up-front and present possible pipelines while editing the textual configuration [122]. The idea behind Rig goes beyond the simple visualization of workflows and allows the direct modeling in a graphical representation as Directed Acyclic Graphs (DAGs), without detouring through textual configurations. The plain structure of a DAG lays the foundation of a clear and accessible workflow description. Job relations in a workflow are well-defined by using a graph structure which allows a straightforward derivation of their (execution) order.

4.1 Rig Is a Cinco Product

Representing CI/CD in a graph-based structure makes it a prime example for being developed based on the Cinco *Meta Tooling Suite* [113, 32]. Although the realization of Rig as a Cinco product is out of scope of this dissertation, the following paragraphs give a short overview of the conceptional and technological background of Cinco. Refer to [AP I: [152]] for the high-level approach and to [156] for an in-depth discussion of design decisions and implementation details.

Cinco is a (meta) domain-specific Integrated Development Environment (IDE) to develop mIDEs that support graph-based modeling languages for a concrete purpose (i.e. PSL) [113]. Resulting mIDEs are specified in Cinco's own textual meta PSLs and called Cinco products. Graphical editors as part of the mIDE are fully-generated from those high-level specifications (cf. Figure 4.5), called metamodels [113]. In contrast to the *turnkey* generation of graphical editors, the compilation of PSL instances to executable artifacts must be manually implemented by the tool developer as a form of code generator (cf. Figure 4.1). Cinco supports Java and Xtend natively as implementation language I out of the box and provides a traversable graph structure to the generators' interface. In the case of Rig, the code generator processes the workflow as an instance of the graph source language S, derives the order of jobs and generates YAML files as instances of the target language T [156].

Cinco's fundamental mindset of empowering programmers and non-programmers to meet at eye level during the development process — by a strict dedication to graphical PSLs [113] — is a true enabler for knowledge sharing. This applies to various areas of information technology, as the numerous example of Cinco products confirm [16, 20, 29, 113, 114, 174, 166], but especially to (large-scale) DevOps, where *bridging the gap* between different stakeholders and

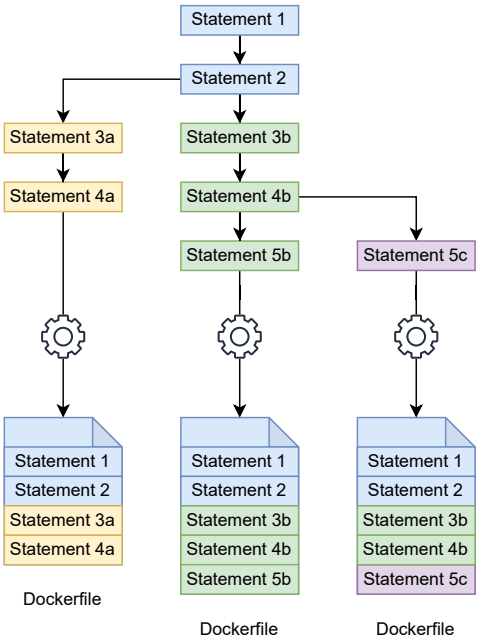


Figure 4.2: Conceptual sketch of the DockerPSL and the generation of Dockerfiles

teams is a necessity.

CINCO and its generated products (i.e. mIDEs) are built on top of the Eclipse Rich Client plattform [101, 113] and make use of the Eclipse Modeling Framework [149]. That implies that those mIDEs are classical desktop applications based on Java and do not yet fulfill the former addressed requirement to provide them as a service. [AP V: [181]] shows how the concept of Cloud IDEs can be extended and applied to the meta modeling approach of CINCO.

4.2 Integrability

As described in Section 3.3, one key feature of Language-Driven Engineering (LDE) is the integrability of PSLs. This section illustrates three different types of integration by an example of Rig and two additional PSLs in the area of container virtualization (cf. Section 2.3). Both PSLs[1] are built upon CINCO as well and originate from the Master's thesis of Timo Walter [165], which I co-supervised.

The first PSL targets the build process of container images and allows the graphical modeling of the desired build steps. In contrast to the sequential representation of statements as textual Dockerfiles, the *DockerPSL* allows the modeling in a tree structure (cf. Figure 4.2). Every node represents a single build step and includes a textual statement of Docker's DSL. Representing Dockerfiles as trees have several advantages, which are briefly discussed in the following. Please compare [165] for a comprehensive description of included features and design decisions.

A Dockerfile is typically the blueprint for a single image. Although Docker's concept of multi-stage builds involves the building of multiple images, these temporary images are just intermediate steps to reduce the layers (cf. Section 2.3) of the yielded image [45]. One single model of the DockerPSL can host and generate a family of Dockerfiles with the same roots. Dockerfiles with same statements can share the same path in the tree (cf. *Statement 1 & Statement 2*

[1] Timo Walter uses the traditional term DSL in [165], but this variance is neglected here in favor of readability.

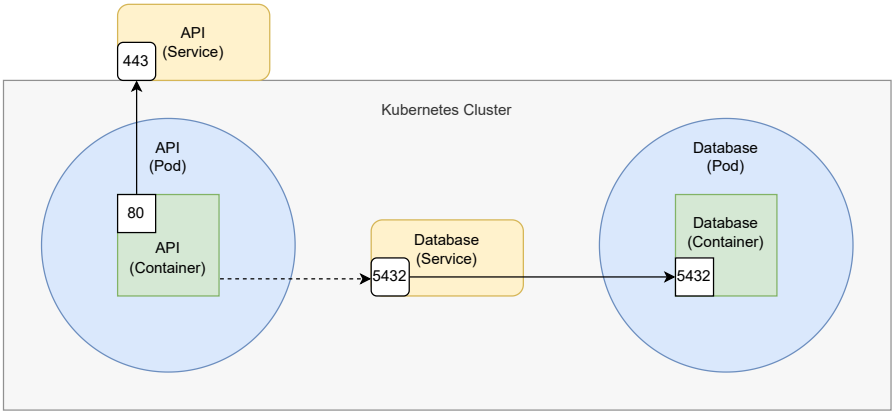


Figure 4.3: Exemplary sketch of the KubernetesPSL

in Figure 4.2). Sharing those statements traditionally would require a seperated Dockerfile from which a named image is build. Inheriting Dockerfiles would need to reference the image by its name and would build upon the cached layers. The DockerPSL allows sharing image layers without pre-building partial images. The generator creates complete and self-contained Dockerfiles, which can be transformed into images independently. Similar to pre-built images, layers that have been already built and cached can be re-used in the same way as before.

Furthermore, the DockerPSL allows not only the reuse of layers inside a model, but can integrate already existing models to inherit their statements as a root (cf. 4 in Figure 4.4). In addition to that, modeler can natively add metadata and information about required dependencies to other containers at run-time when the image is instantiated (e.g. an application container requires a database container for persistence). The generator prefixes this information in a structured manner as comments at the beginning of the Dockerfiles. This is not just a good practice in general when distributing Dockerfiles, but extends the implicit knowledge sharing aspect of creating container images from reproducible statements in a textual DSL.

The second PSL targets the generation of Kubernetes files and bridges the gap between the automated building of container images and the orchestration of running containers. Its design is based on the graphics work originating from the official Kubernetes documentation (cf. [159]). The flexible and powerful high-level specification languages of Cinco allowed not only the adoption of the color schema, but especially the shape of language elements for the *KubernetesPSL*: Clusters are represented as grey rectangles, pods as blue circles, services as yellow rectangles and containers as green squares. This allows "modelers to perceive themselves as working directly with domain concepts" as Kelly & Tolvanen propose in [78] and the expressiveness greatly reduces the entry barrier for beginners.

Figure 4.3 shows a sketch of a deployment for a simple microservice, providing a web-based Application Programming Interface (API). The microservice consists of a database container and an API container. Each container is embedded in a seperated pod, which allows the cluster to place the pods on available worker nodes without any restrictions. Thus, the deployment requires a database service in order to allow the API container to access the database's network port 5432 via the cluster's private network. The Kubernetes cluster rectangle draws the line between private and public communication. The API of the container should be accessible per Hypertext Transfer Protocol (HTTP) publicly from the internet. This is why the API service rectangle is placed not inside the cluster, but on it's edge. It represents the port 80 of the API container and opens the port 443 for secured connections to the outside.

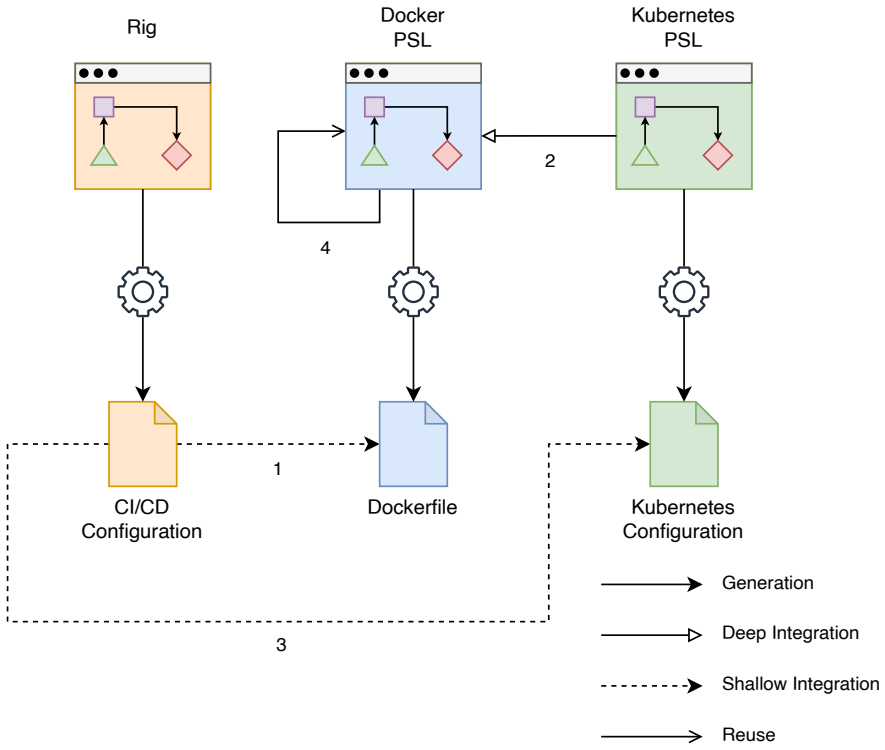


Figure 4.4: The integrability of PSLs by the example of Rig

Although Rig and both PSLs can be used as standalone solutions, they are conceptually designed to be compatible to each other (cf. Figure 4.4). Jobs of CI/CD workflows are often executed as containers where the job's runtime is described by a container image. We can use the DockerPSL to graphically model the Dockerfile from which the container image is built and use it in Rig as our container image for the configuration (cf. 1 in Figure 4.4). In the context of LDE, this is called a shallow integration, since Rig is not aware of the generation process of the images or respectively the underlying Dockerfile.

As mentioned before, Kubernetes deployments are configured by multiple logical entities, like pods and services, but rely on the lowest level on containers [27]. The KubernetesPSL allows the usage of previously built container images and to reference models of the DockerPSL (cf. 2 in Figure 4.4) in pod configurations by prime references, a feature of Cinco. LDE calls this deep integration, since the KubernetesPSL is natively aware of using another PSL.

In retrospective of the DevOps lifecycle and Section 2.1, CI/CD allows the automatic deployment of former delivered applications into deployment environments. When using Kubernetes as deployment environment, a deployment is triggered by applying Kubernetes configurations (serialized as YAML) against an existing cluster. This can be orchestrated by a *Deploy* job as a last step of a CI/CD workflow (cf. Figure 3.6). The job is executed as a container, which provides the Kubernetes client that applies the generated Kubernetes configurations (cf. 3 in Figure 4.4). Similar to (cf. 1 in Figure 4.4) this is a shallow integration where just the generated artifacts are used.

The purpose of graphically modeling container images and their subsequently orchestration can be located mainly in the CI/CD phases of the DevOps lifecycle, but could also be used during the creation phase on the local development machines. In line with the lingualization strategy both PSLs should be maintained by Meta DevOps teams.

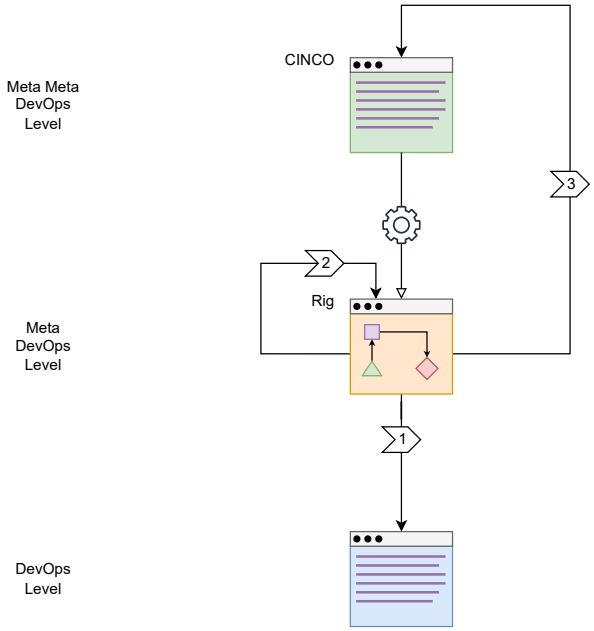


Figure 4.5: Rig supports the creation of CI/CD workflows on three layers

The integration of different PSLs in the same domain has the potential benefits of supporting the discoverability of knowledge and making even implicit relationships between domain elements transparent. This would allow stakeholders from different 'backgrounds' to easier understand how things are connected. Furthermore, the universality of the here presented PSLs could create an additional synergetic effect. They can even be used to define, build and manage the much-needed IT infrastructure for the lingualization strategy (cf. Section 3.6).

4.3 Meta-Level Bootstrapping

The development of Cinco and its products is intended as a multilayer approach that follows a strict top-down refining process (cf. [113]). Subjacent levels have by design no impact on superjacent levels. To put it simply, Cinco developers maintain the high-level specification languages and release Cinco. Cinco users develop one or more graphical PSLs based on this high-level specification languages and generate a Cinco product. Cinco product users model in the including graphical PSLs and generate purpose-specific source code.

Generated Cinco products are independent mIDEs without any backflow to Cinco. Rig is therefore unique in the landscape of Cinco products, since it can be shallowly integrated (cf. Section 4.2) to all three layers of this development process (cf. Figure 4.5): The initial intention of Rig was the generation of workflows for third-party projects (cf. *1* in Figure 4.5), like monolithic applications, microservices or libraries, independently of their complexity or technology stack. Nevertheless, this technology-agnostic approach allows Rig to be applied reflexively and generate workflows for its own development (cf. *2* in Figure 4.5). In addition to that, Rig can be even applied to build Cinco (cf. *3* in Figure 4.5) and as a consequence improve the development process of its own meta level, like it is envisaged by LDE. This type of shallow integration can be referred to as *meta-level bootstrapping*.

Figure 4.6 shows the CI/CD workflow of Cinco (cf. *3* in Figure 4.5) modeled in Rig. The goal of the workflow is two-fold. Firstly it builds Cinco from the source code as an Eclipse-based

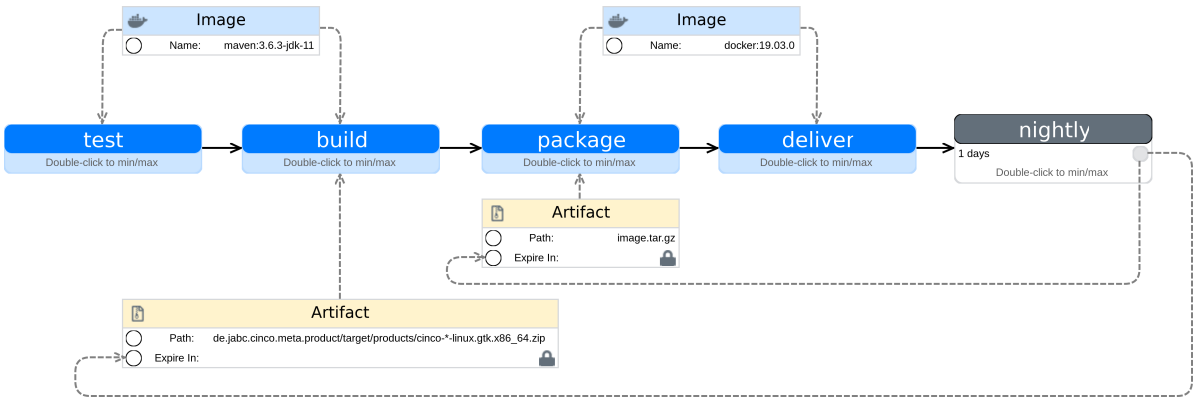


Figure 4.6: CI/CD workflow of Cinco modeled in Rig

(meta) mIDE and secondly delivers it as a container image to a container registry.

Following the approach of [AP I: [152]], workflows modeled in Rig are constructed against targets (cf. *nightly*). Targets can be used to specify the intention of workflows and allow the parametrization of entire pipelines. This feature comes in handy, if different deployment environments (e.g. staging and production) should be deployed via parameterized, but analogous pipelines. In this example, only a single target is present. The *nightly* target doesn't represent a deployment environment, but describes the workflow's objective: The nightly build of Cinco.

Since a nightly build is usually responsible to automatically build and deliver software products for short term testing of yesterday's changes, created artifacts are rather disposable. As already mentioned, targets allow the parametrization of pipelines based on values that are specific to the target. Here, the nightly target specifies an anonymous string parameter with the value of `1 days`. Connecting the anonymous string parameter via dashed edges (- - - -→) with the `Expire in` property of the job artifacts, will propagate the value when the textual CI/CD configuration is generated.

Left of the target, Figure 4.6 shows four sequentially arranged jobs. Each job is responsible for a single task of the nightly build pipeline. Jobs are connected via solid edges (——→) which form a DAG and determines their execution order. In this workflow the DAG is a simple sequence of jobs. Jobs without any incoming solid edge are the starting point of a CI/CD pipeline (in Figure 4.6 the *test* job), since they can be executed without any precondition.

After the *test* job executes Cinco's test cases, the *build* job compiles the source code into a complete mIDE and yields it in form of an archive as its artifact. This is important, since the *package* job requires the artifact of the *build* job, in order to wrap the mIDE in a container image that provides the runtime environment. The *package* job itself yields the container image as its artifact, which is again an archive. Finally, the *deliver* job loads the artifact of the former *package* job and pushes the container image to the container registry.

As discussed in Section 2.3, runtime environments of CI/CD jobs are often realized by container images, which is not to be confused with the aforementioned artifacts. Jobs require certain tools, in order to perform test, build or deliver tasks. For example, Cinco is a Maven project and thus requires Maven as an executable in the runtime environment of the *test* and *build* job. Rig allows the central definition of container images, which can be connected via dashed edges to related jobs and therefore reused. This concept of reuse on the model level, is just one example that demonstrates the simplicity and expressiveness of well-designed PSLs.

The meta-level approach of Cinco resolves a weak spot in the presented lingualization strategy

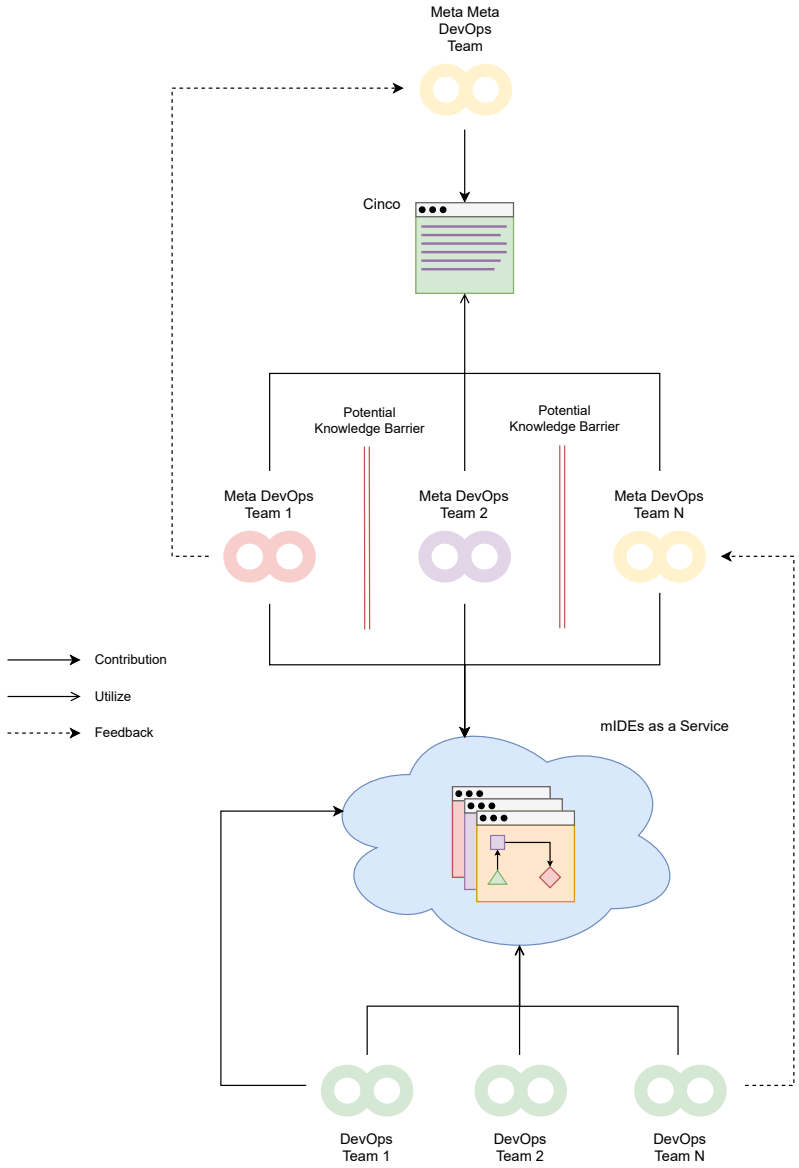


Figure 4.7: Potential knowledge barriers between meta DevOps teams

to overcome knowledge barriers between DevOps teams. When independently developing PSLs on the meta level, how do we ensure that knowledge barriers do not resurface between Meta DevOps teams? Providing a meta-level tooling suite, like Cinco, generalizes the language development of the Meta DevOps teams and ensures the over-arching alignment (cf. Figure 4.7). While this tool-stack alignment of the Meta DevOps teams contradicts DevOps' own pursuit of independent technology choices, the service orientation as conceptual background of Cinco [112] still allows language developers to integrate custom technology. As long as they comply with the defined standards and interfaces of Cinco in general, the specialized knowledge is kept to a minimum. The modular architecture of a cloud-based language development, described in [AP V: [181]], can help to liberate the technology choices, always with the drawback of potentially unshared knowledge.

As the term Meta Meta DevOps predicts, the team that develops Cinco should practice DevOps as well and use Rig's feature of meta-level bootstrapping to enhance their own DevOps lifecycle (cf. Figure 4.5).

Chapter 5

Evaluation

> "Beware of bugs in the above code; I have only proved it correct, not tried it."
> — *Donald E. Knuth* [86]

An evaluation of the presented lingualization strategy in all of it's complexity, especially the impact of the knowledge sharing aspect, goes beyond the boundaries of this dissertation. This is due to three major reasons. First, the DevOps lifecycle itself is a complex construct and varies a lot from one project to another, which aggravates comparability. Second, knowledge sharing is not trivial to quantify and measuring the overall success requires long-term study. Third, the large-scale of the approach includes DevOps (and Meta DevOps) teams with dozens of different stakeholders which makes is challenging to find suitable industrial cases.

Nevertheless, the lingualization strategy is strongly based on the modeling concepts of Language-Driven Engineering (LDE), which instead can be evaluated in the context of DevOps. Evaluating the modeling concepts first has by all means an added value, since it puts the foundation of the lingualization strategy to the test, before more complex studies are conducted. In addition to those theoretical reflections, evaluating an exemplary Purpose-Specific Language (PSL) in a practical environment allows one to draw conclusions about the general eligibility of graphical languages in the context of DevOps.

Thereby, the evaluation is structured as follows. For the first part, the evaluation of modeling concepts in the context of the lingualization strategy is based upon requirements on a DevOps modeling framework, derived from an industrial use-case which has been published in [18]. The second part of the evaluation focuses on Rig and reviews the results of a workshop in the context of the lingualization strategy (cf. [AP VII: [157]]). Rig serves as a prime example of PSLs since it features Continuous Integration and Deployment (CI/CD), one of the most important concepts in DevOps.

5.1 Modeling Concepts of the Lingualization Strategy

The authors of [18] describe the absence of sufficient modeling techniques in the context of DevOps and propose a modeling engineering framework. Although the authors argue in their paper for a "complete modeling framework for DevOps" [18], which contradicts the belief of the lingualization strategy, their requirements are a fitting set of criteria and lay the foundation to evaluate the presented approach. Besides general concerns (RG1–5), they distinguish between requirements on description (RD1–4) and analysis (RA1–6).

The requirements from [18] are listed in Table 5.1, where every row represents a single requirement alongside a score from one (● ○ ○ ○ ○) to five (● ● ● ● ●). The score illustrates how much the lingualization strategy complies with the related requirement. If a requirement is not yet supported, it is marked with ×. The following of this chapter will comment every score

	Requirement	Definition	Score
General	RG1	Modeling of different aspects	●●●●●
	RG2	Model integration	●●●●●
	RG3	Tool integration	●●●●●
	RG4	Customization	●●●●○
	RG5	Different Viewpoints and perspectives	●●●●○
Description	RD1	Appropriate modeling techniques	●●●●●
	RD2	Description of the flows	●●●●●
	RD3	Specification of real-time properties	●○○○○
	RD4	Specification of roles, methods and tools	●●○○○
Analysis	RA1	Identification of constraints	×
	RA2	Configurability of constraint analysis	×
	RA3	Addition of new analysis and simulation techniques	●●●●●
	RA4	Support for time-based analysis and simulation	×
	RA5	Investigation and evaluation of different alternatives	×
	RA6	Support for tool and technology migration analysis	●●●●●

Table 5.1: Scoring of the lingualization strategy; based on requirements published in [18]

and bring forward arguments how they come together. Every requirement analysis contains three parts. A brief summary which describes the requirement is given, an evaluation of the requirement in the context of this dissertation, and a conclusion of a final score.

General Requirements

RG1: This requirement describes the need for supporting the modeling of different aspects included in the DevOps lifecycle. As illustrated in Section 3.3, the lingualization strategy aims to break down the DevOps lifecycle by the different phases or respectively aspects into multiple PSLs, which is technologically backed by LDE's concept of horizontal composition and vertical refinement. Thus, a ●●●●● score is awarded.

RG2: If different aspects of the DevOps lifecycle are supported by different models, the support of model integration is a necessity. Model integration is supported as a first-class citizen in LDE. Section 4.2 presents how three exemplary PSL are integrated and distinguishes, as shown in the example, between shallow and deep integration. Thus, a ●●●●● score is awarded.

RG3: Every aspect of the DevOps lifecycle is related to concrete tools in this area. For example, *Continuous Delivery* and *Continuous Deployment* are often realized by container virtualization. In order to make use of such tools, an integration on model level is required. The authors of [18] especially demand a "lightweight integration [...], to increase agility [...] and enable fast turnarounds". The aforementioned shallow integration is a synonym for "lightweight integration" and, as described in Section 4.2, is used to integrate PSLs with Docker and Kubernetes. Thus, a ●●●●● score is awarded.

RG4: DevOps lifecycles are varying from one project to another in their concrete implementation and maturity. For that reason, modeling frameworks for DevOps require customization support. Following the Unix philosophy [102], of making each PSL to do one thing well, the lingualization strategy allows customization already on the language level, although it requires additional effort by variant management and rigorous product lining in order to not lose track (cf. Figure 3.9). Thus, a ●●●●○ score is awarded.

RG5: The fundamental idea behind DevOps is the alignment of stakeholders from different backgrounds and expertises. Naturally this must be addressed by a modeling approach in the context of DevOps. The lingualization strategy embraces this requirement by providing small declarative PSLs, that aim at simplicity to allow stakeholders from different backgrounds to reason from the graphical representation. Furthermore, the vertical refinement concept of LDE would allow Meta DevOps teams to implement transformations between compatible PSLs to meet the stakeholders individual needs on representation. However, a support for different viewpoints and perspectives is not supported out of the box. Thus, a ●●●●○ score is awarded.

Description Requirements

RD1: As demanded in RG1, different aspects must be supported by models. This requires the support of appropriate modeling techniques which have to suit the specific aspect. Among others, the authors of [18] name BPMN and UML as examples. Cinco, the technological foundation of LDE, provides Meta DevOps teams a meta tooling suite for convenient language development and allows the creation of graphical languages that support existing modeling techniques, such as CMMN (cf. [166]) or UML class diagrams (cf. [20]). In addition to that, Cinco allows custom tailoring of PSL for specific use cases, as illustrated in Chapter 4 by the example of Rig. Thus, a ●●●●● score is awarded.

RD2: The DevOps lifecycle is a continuous improvement process that comprises numerous of different steps in order to function properly. Solutions in this context require the support of modeling information flows. The authors of [18] demand modeling techniques that go beyond documentation and communication, and allow models to be executed. As illustrated in Chapter 4, Rig features the graphical modeling of CI/CD, which is of one of the most popular workflows in the domain of DevOps. Model execution in the form of code generation is a corner stone of the lingualization strategy and allows the transition from the *what* to the *how* level. Sections 3.3 and 3.4 outline the conceptual details how language and code generation interplay in this context. Thus, a ●●●●● score is awarded.

RD3: As Figure 2.3 shows, monitoring and the continuous feedback play a central in the DevOps lifecycle. This includes performance analysis, which requires, as described in [18], the specification of real-time properties on the model level. Although Rig allows the configuration of time-constraints for the validity of artifacts produced by CI/CD jobs, it is not yet supported as native model concept. Thus, a ●○○○○ score is awarded.

RD4: This requirement demands the ability to specify roles, methods and tools in form of models in order to support analysis and simulation of workflows. The lingualization strategy addresses the challenge of knowledge sharing as a bottom-up approach, where single PSL are developed by Meta DevOps team for concrete purposes. For that reason, a global model that includes 'all' roles, methods or tools was not yet intended. However, the development of a PSL that allows the modeling of a DevOps lifecycle with different phases is imaginable. Besides the specification of roles, method, tools and other PSL can shallowly or deeply integrated (cf. Section 4.2). Thus, a ●●○○○ score is awarded.

Analysis Requirements

RA1, 2, 4 & 5 are all rated as ×, since no mentionable work that match the analysis and simulation requirements of [18] has been published in the context of the lingualization strategy at the time of writing.

Essential for the analysis part of the requirement is the availability of runtime data produced by the runtime environment. In order to support this kind of analysis as a native modeling technique, runtime data must be made accessible by the runtime environment, aggregated by the modeling environment and appropriately annotated as part of the model.

Using the example of Rig, the execution of CI/CD workflows produces runtime data such as boolean outcomes if pipelines failed or succeeded and measurable quantities like the job duration. Based on the service-oriented nature of Cinco, implementing client libraries for public Application Programming Interfaces (APIs) of CI/CD providers (e.g. GitLab) would allow the access and aggregation of data for each pipeline. Enhancing Rig in this way would meet the analysis requirements of RA1 and RA4 for time-based analysis, since it aggregates data "related to the notion of time" [18], highlights "time spent in the different elements of the [work]flow" [18], and finally helps to "improve a [work]flow" [18]. Such backflow of runtime information would allow the analysis directly in the mindset-supporting IDE (mIDE) and assist the user in improving the workflow on the model level.

In order to support simulation of models, mIDEs require development environments, where models and generated source code can be tested immediately without compromising production environments. This development environment should closely resemble the production environment, so that simulation outcomes are comparable and reliable. In the context of Rig, executing CI/CD workflows in local development environments is still considered future work [156]. [AP V: [181]] describes how the lines between development and different execution environment blur, which would support similar, but detached execution environments for simulation and production usage conceptually out-of-the box.

RA3: Reducing the iteration time of the DevOps lifecycle in general and the execution time of including workflows in particular (e.g. CI/CD), gets more and more difficult, which requires the continuous adaptation of "new analysis techniques" [18]. LDE considers compatibility and accessibility as first-level concerns, which is underpinned by it's service orientation and vertical refinement concept [146]. Especially the vertical refinement concept allows the transformation of models in specific notations used by third-party analysis techniques, such as model checkers. Thus, a ●●●●● score is awarded.

RA6: The agility of the DevOps lifecycle should not just be reflected by the swift implementation of feature requests, but by the flexibility of tool integration or migration for the lifecycle itself. Model execution by code generation, which has been already addressed in RD2, is the enabler for tool migration in the context of the lingualization strategy. It allows the support of more than just one target tool at the same time, without compromising existing tool integrations or changing the meta model. Thus, a ●●●●● score is awarded.

5.2 Numbers on the Effect of Graphical Modeling with Rig

The here presented evaluation of Rig is based on results of a workshop at the *6th International School on Tool-Based Rigorous Engineering of Software Systems* (STRESS) [97] as part of the ISoLA Conference 2021 in Faliraki, Rhodes (Greece). Findings of this evaluation preluded the *2nd DIME Days* [99] with a retrospective on the aforementioned workshop, as part of the ISoLA Conference 2022, which took place at the same site as 2021. The majority of this section summarizes [AP VII: [157]] and presents the workshop setup, the research questions, the results including a brief interpretation, and the discussion of the threats to validity.

Workshop Setup

In order to establish a common understanding, the workshop was prefaced by presentations on the topic of DevOps and CI/CD, before participants had the opportunity to experience Rig in the hands-on part. For the hands-on part, the workshop organizers prepared three different exercises, each one thematically building on the previous one. The exercises were provided as a digital handout to the participants. Figures 5.1 to 5.3 show minial extracts of the exercises.

Exercise 1 – "Your first CI/CD Configuration": The first exercise **E1** served as an introductory one in the form of *Hello World*, so that participants got familiar with the creation of CI/CD workflows and the execution of pipelines:

```
1 hello-world:
2   script: echo "Hello World"
```

Listing 5.1: CI/CD configuration that will print 'Hello World'

Following the tradition of *Hello World* as the first exercise when learning to program, we will create and run a CI/CD pipeline containing a single job which will print 'Hello World'. A complete and executable example can be found in Listing 5.1.

Figure 5.1: Extract of **E1** from the workshop handout

Exercise 2 – "Build and Deliver a Web Application": The second exercise **E2** comprised the manual creation of CI/CD workflows for a simple web application called TodoMVC [120], with the goal to successfully transfer a provided build script to a working CI/CD configuration:

```
1 #!/bin/bash
2
3 npm ci # Fetch dependencies like React
4 npm run build # Build the actual application
5 npm test # Run unit tests
6 npm run bundle # Bundle application and dependencies
```

Listing 5.2: Bash script that locally builds TodoMVC

TodoMVC is written in TypeScript, uses the React library and can be compiled by Node.js and its package manager npm. To give you a starting point, please have a look at Listing 5.2. This bash script contains all relevant commands to build TodoMVC. Your task is to port this script and its commands (line 2–5) into a `.gitlab-ci.yml`.

Figure 5.2: Extract of **E2** from the workshop handout

Exercise 3 – "CI/CD the Model-driven Way": The third exercise **E3** finally introduced Rig as an alternative approach to create CI/CD workflows, alongside another exemplary web application called Knobster [138]. Instead of providing a build script, this exercise provided a working textual CI/CD configuration, which the participates have been asked to recreate using Rig:

```yaml
image: registry.gitlab.com/mazechazer/knobster

stages:
  - test
  - build

test:
  stage: test
  script:
    - make test

build:
  stage: build
  script:
    - make
  artifacts:
    paths:
      - build
```

Listing 5.3: CI/CD configuration of Knobster

Now that you know how to write CI/CD configurations manually, we will learn how we can use Rig to model CI/CD workflows and fully-generate `.gitlab-ci.yml` files. Please have a look at the manually configured `.gitlab-ci.yml` in Listing 5.3.

Figure 5.3: Extract of **E3** from the workshop handout

Research Questions

The initial idea of a model-driven approach for CI/CD [AP I: [152]], and the actual implementation of Rig [156], promised positive impact in terms of reducing trial-and-error, featuring correctness-by-construction and lowering the entry barrier for CI/CD as a whole. [AP VII: [157]] summarizes these promises under the following three research questions:

Q1: Does Rig reduce the total amount of pipelines that are run until a working configuration is reached?

Q2: Does Rig reduce the amount of invalid pipelines?

Q3: Does Rig reduce the amount of failing pipelines?

Results and Interpretation

Since the workshop was realized using GitLab and managed by the organizers in a centralized group with single repositories for each exercise, the subsequent data analysis was straightforward. GitLab presents comprehensive statistics for each pipeline execution. The aggregated

			E1: Intro	E2: TodoMVC	E3: Knobster
T	Total	$E + I$	20	86	34
E	Executed	$T - I$	20	67	34
I	Invalid	$T - E$	0	19	0
S	Successful	$T - F$	20	26	23
U	Unsuccessful	$E - S$	0	41	11
F	Failed	$I + U$	0	60	11
TSR	Total Success Rate	S/T	100%	30.23%	67.65%
EFR	Execution Failure Rate	U/E	0%	61.19%	32.35%
IR	Invalid Rate	I/F	0%	31.67%	0%
P	Participants		13	13	13

Table 5.2: Pipeline result statistics gathered after the workshop [AP VII: [157]]

data (cf. Table 5.2) showed three major findings when comparing the results of exercise **E2** and **E3**:

- The total number of executed pipelines (T) dropped from 86 to 34, which indicates a reduction of trial-and-error roundtrips.
- The rate of invalid pipelines (IR), e.g. due to syntax errors, dropped from 31.67% to 0%, which indicates that the correctness-by-construction principle of Rig removes a complete error class.
- The drop from 61.19% to 32.35% of failed pipelines (EFR), e.g. due to semantic errors, indicates that participants require fewer trials before successfully creating a working pipeline, which lowers the entry barrier.

Without going into too much detail, please cf. [AP VII: [157]] for a more comprehensive interpretation, those results indicate that the research questions **Q1**–**3** can be answered with *yes*.

Discussion

The absence of failed pipelines (F) in **E1** show that all participants succeeded on the introductory exercise without any abortive attempts. These results doesn't allow to directly draw conclusions, but they indirectly contribute to the validity of the numbers from **E2** and **E3**, since confounding effects due to the lack of experience with GitLab can almost be neglected. Nevertheless, their validity have to be critically examined. Firstly, the sample group of only 13 participants is quite small, without a separate control group. Secondly, **E2** and **E3** feature different web applications, which was intended to contribute to the workshop's diversity, but reduces the comparability of the findings. Finally, due to the lack of a prior survey, nothing is known about the experience of the workshop participants in the domain of DevOps.

As illustrated in [AP VII: [157]], a more comprehensive study that addresses those threats to validity have to be conducted in order to gather more reliable results. Furthermore, analysing not only the binary result if pipelines succeed or fail, but the complexity and structure of CI/CD workflows could be an interesting future field of research. In conclusion, the presented findings are a positive observation that Rig is a step in the right direction of applying graphical modeling to CI/CD and is encouraging to apply more such approaches to the DevOps domain.

Chapter 6

Related Work

> "Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
> — *Martin Fowler* [53]

This chapter discusses existing work that is related to the approach presented in this dissertation. Besides giving an overview of general knowledge sharing strategies, it compares other model-driven approaches in the domain of DevOps with exemplary Purpose-Specific Languages (PSLs) like Rig and with the concept of Language-Driven Engineering (LDE) as the foundation of the lingualization strategy.

6.1 Knowledge Sharing

As stated before, efficient knowledge sharing is an ongoing challenge when practicing DevOps. The literature on Agile and DevOps [35, 142] names numerous approaches of knowledge sharing in practice, which the lingualization strategy doesn't intend to replace, but to complement. This section categorized them into four different classes (cf. Figure 6.1) based on their designate scope of application (intra vs. inter) and chosen strategy (personalization vs. codification).

Intra / Personalization (QI) is the most basic type of knowledge sharing. It represents the foundation of teamwork and is essential for a successful collaborative development process. Team members have to continuously share knowledge with their peers in order to orchestrate up-coming work. While sharing information via direct communication often happens spontaneously and is inseparable from everyday tasks, several methods exist to give this process structure and regularity:

- Daily meetings as part of the Scrum development methodology [150], aim at sharing short-term information about personal sprint progress through brief statements.
- Pair programming combines the method of problem-solving by (typically) two collaborating engineers [169], with the welcoming side effect of knowledge exchange [168]. While it is not limited by team membership, it is more uncommon in inter-team scenarios, since different teams rarely work on the same piece of source code.
- In addition to this, job rotation enables team members to take over another role for a period of time and gain insight to the knowledge of a different task (cf. [132]). It aims at the mutual understanding by slipping into a role and fulfill the associated duties.

Inter / Personalization (QII) becomes a vital knowledge sharing strategy in projects with distributed teams. Especially when practicing large-scale DevOps, where multiple independent developed components have to work together.

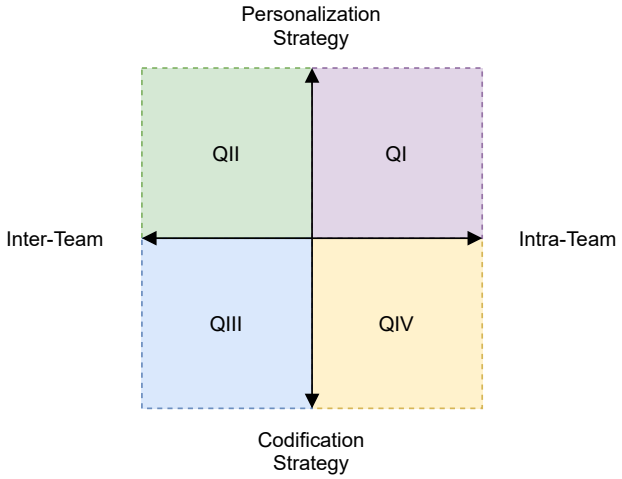


Figure 6.1: Four quadrants of knowledge sharing strategies

- *Scrum of Scrums* are meetings to coordinate the work between multiple Agile Software Development (ASD) teams, each one practicing *Scrum* itself [121]. It uses delegates as proxies to reduce communication lines and spread knowledge. In this context the Scaled Agile Framework serves an example for more structures approaches that "provide guidance for scaling agile development across the enterprise" [161].
- Open communities (inside an organization), e.g. book clubs, share knowledge through open communication between attendees. They are not aiming at producing solid results or finalizing decisions, but rather the exchange of opinions and stories [142].
- Support lines, fall as well into the category of open communities and offer support from senior engineers to junior engineers [142]. Although the support might be provided alongside technical issues and suggestions are incorporated into working code, communication is still the key aspect.
- Core teams are composed of representatives from each team in large-scale DevOps [7]. It is in their responsibility to keep track of the ongoing development and make inter-team decisions. Their focus lies on knowledge sharing through communication, but perform updates to central components (e.g. Application Programming Interface (API) gateways) in very special cases.
- In addition to the aforementioned job rotation, project rotation allows members to "acquire new technical and business knowledge" [132] by rotating between different projects.

Intra / Codification (QIV) can be considered the type of knowledge sharing that represents one of the corner stones of the DevOps practice. DevOps aims at both collaboration between experts in the same team, and the automation of manual tasks.

- The *as-code* movement, as the name already implies, can be considered a typical representative of codified knowledge. Before approaches like infrastructure-as-code became mainstream practice, the provisioning of infrastructure was a manual operation [88] that required deep knowledge in this domain. It was often hidden from developers and performed by few administrators, everyone potentially following their own procedure. Persisting server configurations as part of the applications source code repository shares and documents the infrastructural knowledge with all stakeholders [35]. The approach of executable code even let them directly apply this knowledge by triggering automated

provision processes.

- In the sense of applicable knowledge, automated software testing falls also into this category of knowledge sharing. This holds true especially for *Unit Testing* which aims at verifying the public interface of a specific source code section, called unit (e.g. function or class) [88]. While the testing of the unit's functionality is the main purpose, using source code to create test cases brings another advantage. Test cases show how a specific unit is set up (i.e. instantiating an object from its class), how its interface is invoked and which result is to be expected. Such examples are a good addition to the source code documentation and give developers the knowledge how to use foreign source code.

Inter / Codification (QIII) is the closest type of knowledge sharing to the presented lingualization strategy in this dissertation. It aims at aligning teams by materializing knowledge in scalable and reusable components. Hence, the knowledge is independent of its originator and can be applied in different contexts.

- Standardization committees discuss, establish and maintain global development standards. Those standards act as comprehensive guidelines for other teams when developing components that must comply with existing interfaces and integrated in an overall system [142].
- Cloud platform teams maintain a standardized tool portfolio and create templates for kick-starting new projects (e.g. microservices) in selected programming languages [7] that are designated for running in the cloud. Other 'self-organized' teams must choose from that portfolio when creating applications.
- Domain-driven design considers modeling as a "way of structuring domain knowledge" [49] in a distilled form which is used as the *ubiquitous language*. It advocates for using the same language throughout the whole development process of a software project including business and development, and provides *strategic design* practices for "maintaining model integrity" [49] on an inter-team level.

6.2 MD* in the Context of DevOps

Lately, the model-driven community discovered DevOps more and more as a domain of interest. As a result, a couple of MD*[1] approaches, which target DevOps and related technologies, have been published. This section gives an overview over such approaches and highlights similarities and distinctions. They are listed in order of their publication date.

The DICER tool [4] 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 DICER IDE is realized as a plugin for Eclipse and uses UML Deployment Diagrams as their modeling language. In contrast to the more general approach of the presented lingualization strategy, DICER focus especially on big data technologies [62]. Furthermore, their approach to DevOps features primarily the design and deployment phases, while this dissertation advocates to cover all phases of the DevOps lifecycle by an independent but integrable set of PSLs.

A family of modeling languages for developing and operating service-oriented applications is presented in [127]. Their approach targets different roles in a DevOps team with interconnected

[1] Markus Völter introduced MD* as a broader term in [164] to summarize similar model-driven approaches that use models as abstraction [127] and input for further processing, like validation or code generation.

languages for domain-, service-, and operation model. Similar to the lingualization strategy, the authors emphasize the documentation aspect of models and apply multiple interconnected but 'viewpoint-specific' languages to reduce their complexity. Although lowering the entry barrier when learning new languages, is in accordance with the concept of LDE the presented MD* approach is implemented by textual languages only. Furthermore, it remains unclear if model processing is considered future work or if the presented approach already supports model transformation into executable artifacts. In contrast to this, LDE considers model processing (e.g. by a code generator) a vital requirement for the right to exist. The presented PSLs in (cf. Chapter 4) support full-code generation (cf. [112]) which in case of Rig already had been evaluated in [AP VII: [157]]. The results of the evaluation are summarized in Section 5.2.

DevOpsML [33] is a framework built on Ecore and the Eclipse Modeling Framework [149] to apply MD* for DevOps. It aims at interconnecting model-based software engineering processes with platforms that provide fundamental functions of the DevOps practice. Their approach allows the specification of DevOps platforms including requirements and capabilities in dedicated models. Based upon a linking language, processes and platforms are woven together. At the time of writing, DevOpsML aims at documentation only and perceives model execution as future work. In contrast to that, the exemplary PSLs presented in Chapter 4 are fully implemented as functional mindset-supporting IDEs (mIDEs) with associated code generators. Rig, as one of the examples, has been successfully evaluated in [AP VII: [157]] and has been used in several projects to model Continuous Integration and Deployment (CI/CD) workflows, such as the workflow for Cinco as illustrated in Figure 4.6. Furthermore, as described in Section 1.1, the underlying concept of the lingualization strategy advocates for a reciprocal interaction between DevOps and MD* and goes beyond the unidirectional approach of DevOpsML.

In their paper [143] 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-sized organizations (SMOs), which they extract from a previous study: Maintaining a common architectural understanding, and handling the complexity of deploying and operating microservice application. The lingualization strategy can be distinguished from the presented approach based on three major characteristics. Firstly, their workflow uses textual modeling languages and implements visualization as a subsequent process, while LDE considers graphical models as first-class citizens. Secondly, their approach tries to cover the development and operating process in a single workflow, while the lingualization strategy emphasizes the need for loosely-coupled PSL to lower the general hurdle of adoption. Thirdly, since their workflow aims at SMOs, language developers remain external stakeholders and therefore are not actively involved in the continual improvement process of application and languages. In contrast to that, the lingualization strategy introduces the concept of Meta DevOps teams as language developers in large-scale environments.

StalkCD [47] is a model-driven framework for interoperable CI/CD pipeline definitions. It aims at parsing different CI/CD configuration formats (e.g. Jenkins CI, GitLab CI) 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. [2]). In contrast to Rig, StalkCD doesn't use graphical DSLs for the purpose of active modeling, but solely for visualizing CI/CD configurations as BPMN.

The presented approach in [34] aims at transforming textual DevOps artifacts, independent of the DevOps lifecycle phase, into models back and forth for consistency checking. In contrast to the underlying LDE concept of this dissertation, they do not treat models as the single source of truth (i.e. One Thing Approach) while they consider textual representation as the main engineering artifact.

Chapter 7

Conclusion

> "Eventually, programming should become a two-way conversation between the imprecise human language and the precise, if unimaginative, machine."
> — *Herbert D. Benington* [14]

This dissertation discussed the challenge of knowledge sharing in large-scale software projects. Especially projects that are based on a great number of agile DevOps teams, are running into danger to create inter-team knowledge barriers. As indicated before, this is a direct consequence of the intentional independence of DevOps teams, accompanied by the free choice of technology when creating microservices and the various state of their lifecycle maturity.

The main contribution of this dissertation is the concept of a lingualization strategy (cf. Figure 7.1) for sharing knowledge in the context of the DevOpy lifecycle by following the principle of Language-Driven Engineering (LDE) via custom tailored languages. Part of this concept is the introduction of so-called Meta DevOps teams, whose sole purpose is the discovery and storage of knowledge, by developing and distributing Purpose-Specific Languages (PSLs). Those teams meet the demand of LDE for dedicated stakeholders that practice language development.

Besides the knowledge sharing aspect of the lingualization strategy, maintaining a family of PSLs to support the complete DevOps lifecycle has the potential to make DevOps accessible to a wider audience by advancing from *as-code* towards a practice of *as-models* in the future.

7.1 Limitations

Introducing a radical approach to the challenge of knowledge sharing naturally comes with limitations and drawbacks. As already indicated in Section 3.3, the introduction of PSLs, especially to well-performing projects with established processes, is not a trivial task. Migrating from handwritten solutions to automated code-generation can be met with disapproval. In particular when approaching a domain like DevOps where technical, cultural and organizational challenges (cf. Chapter 2) blend into each other.

It is very likely that engineers initially prefer their own manually created solutions over third-party suppliers for different reasons, even if they are provided by teams from the same organization. Engineers might have been devoted a lot of work and time to create work routines [61]. They could fear uncertainty [61] accompanied by the loss of influence. A degradation from a developer to an 'ordinary' user might be perceived offensive [61]. The term "not invented here syndrome" summarizes the refusal of external ideas and solutions [77] on the receiver's side of the knowledge [61]. While sociological challenges are not the focus of this dissertation, eliminating initial prejudices by establishing and following guidelines of an incremental PSL migration might serve as a promising strategy. Key to a successful incremental migration is a stepwise adoption, in contrast to the direct changeover of a big bang approach [17].

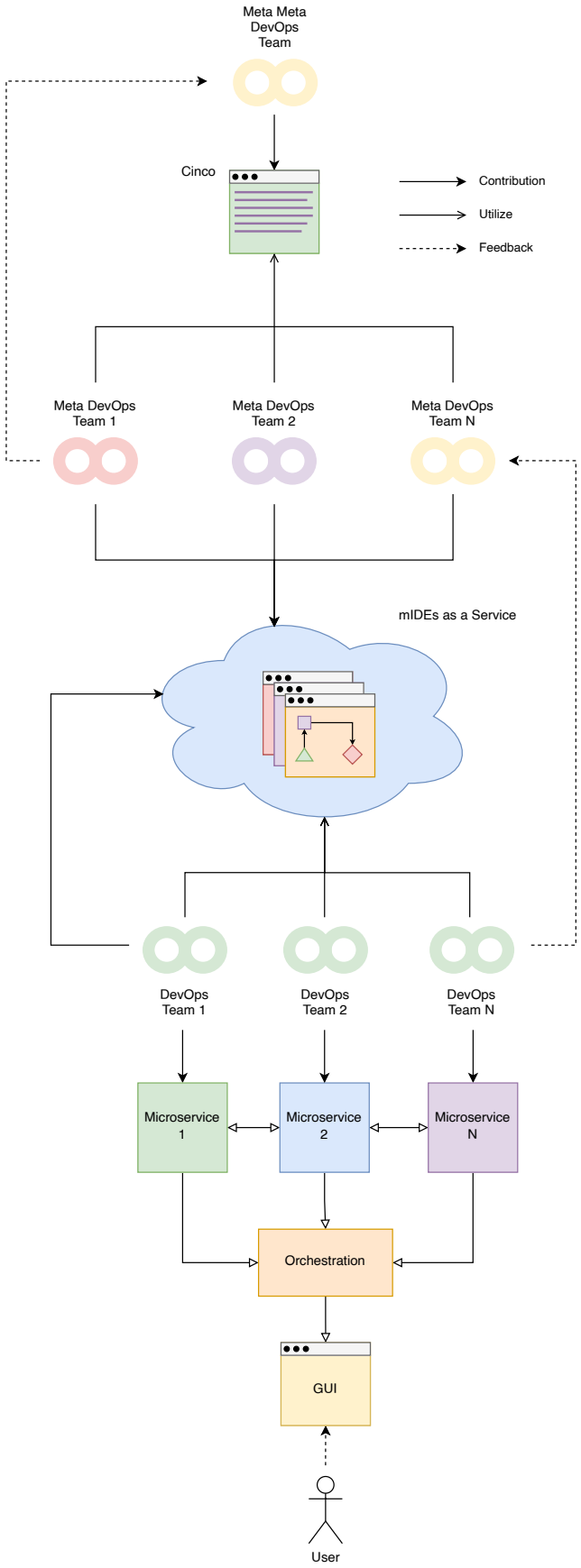


Figure 7.1: Collaboration in the lingualization strategy (composed of Figures 3.2, 3.5 and 4.7)

In addition to the skepticism of novel approaches, the fear of a technological lock-in might represent an additional obstacle when introducing PSLs. The divide and conquer principle of LDE enables DevOps teams to test and adopt PSLs without facing the threat of an overarching technological lock-in effect. The intended interoperability between languages and services, allows a free choice and combination of PSLs. Especially the shallow integration counteracts the lock-in effect by providing a clear interface between existing solutions and generated code. The presented DockerPSL is a good example, since it generates fully compatible Dockerfiles that do not differ a lot from handwritten files. Resulting container images can be built, distributed and instantiated like any other image.

Furthermore, early success when testing a PSL will convince and motivate to try other PSLs. Therefore it is important that Meta DevOps teams *eat their own dog food* (cf. [64]) and showcase successful practice of their PSLs. The successful tool chain integration into the DevOps process makes it easier to convince other teams to adopt a certain PSL as well. The real life scenario of bootstrapping by applying PSLs in order to create other PSLs is a compelling selling point. This requires projects to be accessible to other teams in the organization, which also helps to prevent the creation of tacit knowledge.

When promoting a custom tailored programming language for a specific purpose, one has to keep in mind that not only creating this language, but learning this language is also time-consuming. Even for experts that mastered different programming languages and for junior developers that belong to the generation of digital natives [40], learning a new language is not a trivial task. Based on the principles of LDE the lingualization strategy tries to create simple and expressive languages, to keep the learning curve flat. Besides the language itself, mindset-supporting IDEs (mIDEs) are playing an important role in creating a convenient modeling experience by accommodating the stakeholders' mindset. However, this doesn't completely take away the need for traditional knowledge sharing practices (cf. Section 6.1). Although the lingualization strategy aims at simplicity, the introduction of a new language requires at least basic documentation (codification strategy) as an entry point. Furthermore, workshops (personalization strategy) including presentations and hands-on courses can help to demonstrate the usefulness of a new language. Experience during the *6th International School on Tool-Based Rigorous Engineering of Software Systems* [97] showed how a structured workshop with incremental exercises eases the adoption process of a new tool like Rig (cf. [AP VII: [157]]).

A similar economical concern is raised by the decision of make or buy (cf. [83]). Before deciding to create a new language Meta DevOps teams should consider existing solutions in the domain. Practicing LDE already reduces the overhead of language development, but taking existing languages into account can be less expensive. Instead of creating a language from scratch, it might be economically reasonable to support established projects. But Meta DevOps teams should keep in mind, that such projects potentially exist in a bigger context and are supported by an active community, which may take a critical view on change and modifications. Eventually, this can prevent the language to become the perfect fit for the envisaged use-case.

7.2 Future Work

Applying PSL with the purpose of knowledge sharing raises interesting questions that are out of scope of this dissertation. Nevertheless, this section covers some of the future work in different areas.

The majority of this dissertation presents the lingualization strategy alongside the challenges of Continuous Practices (CP) and infrastructure provisioning. They are representing the techno-

logical crossover of development and operations, and thus play a special role during the DevOps lifecycle. Nevertheless, exploring how PSLs can be applied to more phases of the DevOps lifecycle is an interesting question. Projects like the GOLD Framework [144] target the domain of business modeling by introducing a family of multi-level PSLs for different stakeholders. This falls in line with Debois' demand of aligning more types of stakeholders than just development and operations during the development process [37]. Since GOLD is based on Cinco, a deep integration with the previously presented PSLs (cf. Section 4.2) is entirely possible. This integration would allow a more seamless interaction between business and DevOps (cf. Figure 2.2) and reduce knowledge barriers in the planning phase.

As this dissertation emphasizes, the lingualization strategy for knowledge sharing requires a great amount of IT infrastructure on the organizational level and demands strong coordination of involved services (cf. Section 3.6). In order to kick-start knowledge sharing, additional overhead should be avoided, thus infrastructure provisioning and maintenance are realized by modern but approved IT administration practices. However, as the lingualization strategy as a holistic project evolves and developed PSLs in the domain of automation are maturing, a more radical practice is conceivable. Future work should consider the potential effects of applying PSL not only to DevOps team, Meta DevOps teams or Meta Meta DevOps teams, but to the team(s) who are in fact responsible for layers of IT infrastructure layers below DevOps (e.g. physical or virtual machines, networking and storage).

As prominent examples of security vulnerabilities (e.g. Heartbleed bug [8] and Log4j2 exploit [111]) underline, creating secure applications is not a trivial task. Especially web-based systems constitute lucrative targets for attacks via the internet. Their publicly accessible interface leaves them more vulnerable to exploits than private systems. DevOps' aim for automated software releases allows software engineers to quickly respond and automatically push security patches into production. While this reactive response is crucial for exploits once they became public, the term DevSecOps [131] summarizes a more proactive approach and treat security as a first-class citizen during the development lifecycle. Since *knowledge sharing* was identified in [131] as one of 13 attributes that are essential for a successful DevSecOps culture, it raises the question if and how the presented lingualization strategy can support the transition from DevOps to DevSecOps.

Organizations "extend their knowledge base access to the business partners and customers. Though such access is restricted to certain areas, it is playing an important role in collaborative product development, service delivery and project accomplishments" [118]. Companies that are successfully practicing the lingualization strategy, should consider publishing mature and successful PSLs as open source. This would go beyond the inter-team knowledge sharing and allows organizations to contribute towards a cross-company knowledge transfer. This is not just interesting to improve knowledge transfer in business areas where cross-company collaboration is a necessity (i.e. logistics), but for companies to develop new business models by establishing global standards upon PSLs. Contributing companies would benefit as early adopters of their own standards and could provide "value-added services" [175] (e.g. consulting) by following examples of successful business models in the area of open source software [109].

An essential challenge of future work is the comprehensive evaluation of the lingualization strategy. While the idea of knowledge sharing presented here through PSLs is a proof of concept, practice in real-world environments has to demonstrate its success in a long-run. The aim at large-scale DevOps might make a formal study difficult to conduct. Nevertheless, experiences with Rig during a workshop at the *6th International School on Tool-Based Rigorous Engineering of Software Systems* [97] yielded positive results and let us cautiously optimistic that PSL lower the entry barrier for new technology and efficiently share knowledge.

List of Abbreviations

API	Application Programming Interface
ASD	Agile Software Development
CDE	Continuous Delivery
CD	Continuous Deployment
CI/CD	Continuous Integration and Deployment
CI	Continuous Integration
CP	Continuous Practices
DAG	Directed Acyclic Graph
DSL	Domain-Specific Language
DSML	Domain-Specific Modeling Language
GPL	General-Purpose Language
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
LDE	Language-Driven Engineering
OTA	One Thing Approach
PSL	Purpose-Specific Language
SMO	small and medium-sized organization
SOA	Service Oriented Architecture
TFG	Type-Safe Functional GraphQL
mIDE	mindset-supporting IDE

References

[1] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. "Software Documentation Issues Unveiled". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 1199–1210. DOI: 10.1109/ICSE.2019.00122.

[2] Thomas Allweyer. *BPMN 2.0 - Business Process Model and Notation*. Books on Demand, 2009. ISBN: 9783839121344.

[4] Matej Artač, Tadej Borovšak, Elisabetta Di Nitto, Michele Guerriero, and Damian A. Tamburri. "Model-Driven Continuous Deployment for Quality DevOps". In: *Proceedings of the 2nd International Workshop on Quality-Aware DevOps*. QUDOS 2016. Saarbrücken, Germany: Association for Computing Machinery, 2016, pp. 40–41. ISBN: 9781450344111. DOI: 10.1145/2945408.2945417.

[6] Zia Babar, Alexei Lapouchnian, and Eric Yu. "Modeling DevOps Deployment Choices Using Process Architecture Design Dimensions". In: *The Practice of Enterprise Modeling*. Ed. by Jolita Ralyté, Sergio España, and Óscar Pastor. Cham: Springer International Publishing, 2015, pp. 322–337. ISBN: 9783319258973.

[7] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture". In: *IEEE Software* 33.3 (2016), pp. 42–52.

[8] James Banks. "The Heartbleed bug: Insecurity repackaged, rebranded and resold". In: *Crime, Media, Culture* 11.3 (2015), pp. 259–279. DOI: 10.1177/1741659015592792.

[9] Len Bass. "The Software Architect and DevOps". In: *IEEE Software* 35.1 (2018), pp. 8–10. DOI: 10.1109/MS.2017.4541051.

[10] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015. ISBN: 9780134049847.

[11] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2000. ISBN: 9780321278654.

[14] Herbert D. Benington. "Production of Large Computer Programs". In: *Annals of the History of Computing* 5.4 (1983), pp. 350–361. DOI: 10.1109/MAHC.1983.10102.

[15] David Bennet and Alex Bennet. "The depth of knowledge: surface, shallow or deep?" In: *Vine* 38.4 (Jan. 2008), pp. 405–420. ISSN: 0305-5728. DOI: 10.1108/03055720810917679.

[16] Agata Berg, Cedric Perez Donfack, Julian Gaedecke, Eike Ogkler, Steffen Plate, Katharina Schamber, David Schmidt, Yasin Sönmez, Florian Treinat, Jan Weckwerth, Patrick Wolf, and Philip Zweihoff. *PG 582 - Industrial Programming by Example*. Tech. rep. Dortmund, Germany: TU Dortmund University, 2015.

[17] Jesus Bisbal, Deirdre Lawless, Bing Wu, Jane Grimson, Vincent Wade, Ray Richardson, and Declan 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*. 1997, pp. 529–530. DOI: 10.1109/APSEC.1997.640219.

[18] Francis Bordeleau, Jordi Cabot, Juergen Dingel, Bassem S. Rabil, and Patrick Renaud. "Towards Modeling Framework for DevOps: Requirements Derived from Industry Use Case". In: *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*. Ed. by Jean-Michel Bruel, Manuel Mazzara, and Bertrand Meyer. Cham: Springer International Publishing, Jan. 2020, pp. 139–151. ISBN: 9783030393069.

[19] Steve Boßelmann. "Evolution of ecosystems for Language-Driven Engineering". PhD thesis. Dortmund, Germany: TU Dortmund University, 2023. DOI: http://dx.doi.org/10.17877/DE290R-23218.

[20] Steve Boßelmann, Markus Frohme, Dawid Kopetzki, Michael Lybecait, Stefan Naujokat, Johannes Neubauer, Dominic Wirkner, Philip Zweihoff, and Bernhard Steffen. "DIME: A Programming-Less Modeling Environment for Web Applications". In: *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 9953. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 809–832. ISBN: 9783319471693. DOI: 10.1007/978-3-319-47169-3_60.

[21] Erwan Bousse and Manuel Wimmer. "Domain-Level Observation and Control for Compiled Executable DSLs". In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2019, pp. 150–160. DOI: 10.1109/MODELS.2019.000-6.

[22] Lionel C. Briand. "Software Documentation: How Much is Enough?" In: *Seventh European Conference on Software Maintenance and Reengineering, 2003*. 2003, pp. 13–15. ISBN: 0769519024. DOI: 10.1109/CSMR.2003.1192406.

[23] Robert O. Briggs and Jay F. Nunamaker, Jr. "Special Section: The Growing Complexity of Enterprise Software". In: *Journal of Management Information Systems* 37.2 (2020), pp. 313–315. DOI: 10.1080/07421222.2020.1759339.

[24] R. John Brockmann. *Writing better computer user documentation: From paper to hypertext (version 2.0)*. John Wiley & Sons, Inc., 1990. ISBN: 0471622591.

[25] Hendrik Bünder. "Decoupling Language and Editor – The Impact of the Language Server Protocol on Textual Domain-Specific Languages". In: *MODELSWARD 2019: Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development*. Prague, Czech Republic: SCITEPRESS - Science and Technology Publications, Lda, 2019, pp. 129–140. ISBN: 9789897583582.

[26] Vladimír Bureš. "Cultural Barriers in Knowledge Sharing". In: *E+M Ekonomics and Management* 6 (2003), pp. 57–62. ISSN: 1212-3609.

[27] Brendan Burns, Joe Beda, and Kelsey Hightower. *Kubernetes: Up and Running: Dive into the Future of Infrastructure*. O'Reilly Media, 2019. ISBN: 9781492046530.

[28] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. "Borg, Omega, and Kubernetes". In: *Communications of the ACM* 59.5 (Apr. 2016), pp. 50–57. ISSN: 0001-0782. DOI: 10.1145/2890784.

[29] Mounir Chadli, Jin H. Kim, Kim G. Larsen, Axel Legay, Stefan Naujokat, Bernhard Steffen, and Louis-Marie Traonouez. "High-level frameworks for the specification and verification of scheduling problems". In: *Software Tools for Technology Transfer* 20.4 (2017), pp. 397–422. DOI: 10.1007/s10009-017-0466-1.

[30] Robert Chatley, Alastair Donaldson, and Alan Mycroft. "The Next 7000 Programming Languages". In: *Computing and Software Science: State of the Art and Perspectives*. Ed. by Bernhard Steffen and Gerhard Woeginger. Cham: Springer International Publishing, 2019, pp. 250–282. ISBN: 9783319919089. DOI: 10.1007/978-3-319-91908-9_15.

[31] Binildas Christudas. *Practical Microservices Architectural Patterns: Event-Based Java Microservices with Spring Boot and Spring Cloud*. Berkeley, CA: Apress, 2019, pp. 21–34. ISBN: 9781484245019. DOI: 10.1007/978-1-4842-4501-9.

[33] Alessandro Colantoni, Luca Berardinelli, and Manuel Wimmer. "DevOpsML: Towards Modeling DevOps Processes and Platforms". In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS '20. Virtual Event, Canada: Association for Computing Machinery, Oct. 2020. ISBN: 9781450381352. DOI: 10.1145/3417990.3420203.

[34] Alessandro Colantoni, Benedek Horváth, Ákos Horváth, Luca Berardinelli, and Manuel Wimmer. "Towards Continuous Consistency Checking of DevOps Artefacts". In: *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. Oct. 2021, pp. 449–453. ISBN: 9781665424844. DOI: 10.1109/MODELS-C53483.2021.00069.

[35] Jennifer Davis and Ryn Daniels. *Effective DevOps: Building a Culture of Collaboration, Affinity, and Tooling*. O'Reilly Media, Inc., 2016. ISBN: 1491926309.

[36] Omar Al-Debagy and Peter Martinek. "A Comparative Review of Microservices and Monolithic Architectures". In: *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*. 2018, pp. 149–154. ISBN: 9781728111179. DOI: 10.1109/CINTI.2018.8928192.

[37] Patrick Debois, Jez Humble, Joanne Molesky, Eric Shamow, Lawrence Fitzpatrick, Michael Dillon, Bill Phifer, and Dominica DeGrandis. "Devops: A Software Revolution in the Making?" In: *Journal of Information Technology Management* 24.8 (2011), pp. 3–39.

[39] Tobias Dehling and Ali Sunyaev. "Domain-Specific Languages and Digital Preservation: Supporting Knowledge-Management". In: *Management* (2012), pp. 1273–1284.

[40] Davide Di Ruscio, Dimitris Kolovos, Juan de Lara, Alfonso Pierantonio, Massimo Tisi, and Manuel Wimmer. "Low-code development and model-driven engineering: Two sides of the same coin?" In: *Software and Systems Modeling* (Jan. 2022). ISSN: 1619-1374. DOI: 10.1007/s10270-021-00970-2.

[41] Wei Ding, Peng Liang, Antony Tang, and Hans van Vliet. "Knowledge-based approaches in software documentation: A systematic literature review". In: *Information and Software Technology* 56.6 (2014), pp. 545–567. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2014.01.008.

[42] Torgeir Dingsøyr, Tor Erlend Fægri, and Juha Itkonen. "What Is Large in Large-Scale? A Taxonomy of Scale for Agile Software Development". In: *Product-Focused Software Process Improvement*. Ed. by Andreas Jedlitschka, Pasi Kuvaja, Marco Kuhrmann, Tomi Männistö, Jürgen Münch, and Mikko Raatikainen. Cham: Springer International Publishing, 2014, pp. 273–276. ISBN: 9783319138350. DOI: 10.1007/978-3-319-13835-0_20.

[43] Torgeir Dingsøyr and Nils Brede Moe. "Research Challenges in Large-Scale Agile Software Development". In: *SIGSOFT Software Engineering Notes* 38.5 (Aug. 2013), pp. 38–39. ISSN: 0163-5948. DOI: 10.1145/2507288.2507322.

[47] Thomas F. Düllmann, Oliver Kabierschke, and André van Hoorn. "StalkCD: A Model-Driven Framework for Interoperability and Analysis of CI/CD Pipelines". In: *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. Sept. 2021, pp. 214–223. ISBN: 9781665427050. DOI: 10.1109/SEAA53835.2021.00035.

[48] Michael Eder. "Hypervisor- vs. Container-based Virtualization". In: *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)* (June 2016). DOI: 10.2313/NET-2016-07-1_01.

[49] Eric J. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004. ISBN: 9780321125217.

[53] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999. ISBN: 0134757599.

[54] Martin Fowler and Rebecca Parsons. *Domain-Specific Languages*. Addison-Wesley / ACM Press, 2011. ISBN: 0321712943.

[55] Fiorenzo Franceschini, Maurizio Galetto, and Domenico Maisano. *Management by Measurement: Designing Key Indicators and Performance Measurement Systems*. Springer Science & Business Media, Jan. 2007. ISBN: 9783540732112. DOI: 10.1007/978-3-540-73212-9.

[56] Ulrich Frank. "Domain-Specific Modeling Languages: Requirements Analysis and Design Guidelines". In: *Domain Engineering: Product Lines, Languages, and Conceptual Models*. Ed. by Iris Reinhartz-Berger, Arnon Sturm, Tony Clark, Sholom Cohen, and Jorn Bettin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 133–157. ISBN: 9783642366543. DOI: 10.1007/978-3-642-36654-3_6.

[57] Shahla Ghobadi and Lars Mathiassen. "Perceived barriers to effective knowledge sharing in agile software teams". In: *Information Systems Journal* 26.2 (2016), pp. 95–125. DOI: 10.1111/isj.12053.

[60] Nuno Gonçalves, Diogo Faustino, António Rito Silva, and Manuel Portela. "Monolith Modularization Towards Microservices: Refactoring and Performance Trade-offs". In: *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C)*. 2021, pp. 1–8. ISBN: 9781665439107. DOI: 10.1109/ICSA-C52384.2021.00015.

[61] David Grosse Kathoefer and Jens Leker. "Knowledge transfer in academia: an exploratory study on the Not-Invented-Here Syndrome". In: *The Journal of Technology Transfer* 37.5 (Oct. 2012), pp. 658–675. ISSN: 1573-7047. DOI: 10.1007/s10961-010-9204-5.

[63] Morten T Hansen, Nitin Nohria, and Thomas Tierney. "What's Your Strategy for Managing Knowledge". In: vol. 77. 2. Butterworth-Heinemann Oxford, England, 1999, pp. 106–116.

[64] Warren Harrison. "Eating Your Own Dog Food". In: *IEEE Software* 23.3 (2006), pp. 5–7. ISSN: 0740-7459. DOI: 10.1109/MS.2006.72.

[65] Aymeric Hemon, Brian Fitzgerald, Barbara Lyonnet, and Frantz Rowe. "Innovative Practices for Knowledge Sharing in Large-Scale DevOps". In: *IEEE Software* 37.3 (2020), pp. 30–37. ISSN: 0740-7459. DOI: 10.1109/MS.2019.2958900.

[66] Jordan Henkel, Christian Bird, Shuvendu K. Lahiri, and Thomas Reps. "Learning from, Understanding, and Supporting DevOps Artifacts for Docker". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE '20. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 38–49. ISBN: 9781450371216. DOI: 10.1145/3377811.3380406.

[67] Daqing Hou and Yuejiao Wang. "An Empirical Analysis of the Evolution of User-Visible Features in an Integrated Development Environment". In: *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*. CASCON '09. Ontario, Canada: IBM Corp., 2009, pp. 122–135. DOI: 10.1145/1723028.1723044.

[68] Robin Hunter. *The Design and Construction of Compilers*. Wiley New York, 1981. ISBN: 0471280542.

[72] Manfred A. Jeusfeld. "Metamodel". In: *Encyclopedia of Database Systems*. Ed. by Ling Liu and M. Tamer Özsu. Boston, MA: Springer US, 2009, pp. 1727–1730. ISBN: 9780387399409. DOI: 10.1007/978-0-387-39940-9_898.

[73] Denis L. Johnston. "Scientists Become Managers-The 'T'-Shaped Man". In: *IEEE Engineering Management Review* 6.3 (1978), pp. 67–68. DOI: 10.1109/EMR.1978.4306682.

[74] Sven Jörges. *Construction and Evolution of Code Generators - A Model-Driven and Service-Oriented Approach*. Vol. 7747. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Germany, 2013. ISBN: 9783642361272. DOI: 10.1007/978-3-642-36127-2.

[75] Abhinav Krishna Kaiser. "Continual Improvement". In: *Become ITIL® 4 Foundation Certified in 7 Days: Understand and Prepare for the ITIL Foundation Exam with Real-life Examples*. Berkeley, CA: Apress, 2021, pp. 249–270. ISBN: 9781484263617. DOI: 10.1007/978-1-4842-6361-7_10.

[76] Hui Kang, Michael Le, and Shu Tao. "Container and Microservice Driven Design for Cloud Infrastructure DevOps". In: *2016 IEEE International Conference on Cloud Engineering (IC2E)*. 2016, pp. 202–211. ISBN: 9781509019618. DOI: 10.1109/IC2E.2016.26.

[77] Ralph Katz and Thomas J. Allen. "Investigating the Not Invented Here (NIH) syndrome: A look at the performance, tenure, and communication patterns of 50 R & D Project Groups". In: *R&D Management* 12.1 (1982), pp. 7–20. DOI: 10.1111/j.1467-9310.1982.tb00478.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-9310.1982.tb00478.x.

[78] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Press, Hoboken, NJ, USA, 2008. ISBN: 0470036664. DOI: 10.1002/9780470249260.

[80] Gene Kim. *The Unicorn Project: A Novel about Developers, Digital Disruption, and Thriving in the Age of Data*. IT Revolution Press, 2019. ISBN: 9781942788775.

[81] Gene Kim, Kevin Behr, and George Spafford. *The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win*. IT Revolution Press, 2018. ISBN: 9781942788300.

[82] Gene Kim, Jez Humble, Patrick Debois, John Willis, and Nicole Forsgren. *The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations*. IT Revolution Press, 2021. ISBN: 9781950508433.

[83] Peter G. Klein. "The Make-or-Buy Decision: Lessons from Empirical Studies". In: *Handbook of New Institutional Economics*. Ed. by Claude Menard and Mary M. Shirley. Boston, MA: Springer US, 2005, pp. 435–464. ISBN: 9780387250922. DOI: 10.1007/0-387-25092-1_18.

[84] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. 1st ed. Addison-Wesley Professional, 2008. ISBN: 0321553454.

[85] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973. ISBN: 020103803X.

[86] Donald E. Knuth. "The correspondence between Donald E. Knuth and Peter van Emde Boas on priority deque". 1977.

[87] Dawid Kopetzki. "Generation of Domain-Specific Language-to-Language Transformation Languages". Dissertation. Dortmund, Germany: TU Dortmund University, 2019. DOI: 10.17877/DE290R-21179.

[88] Mohamed Labouardy. *Pipeline as Code: Continuous Delivery with Jenkins, Kubernetes, and Terraform*. Manning, 2021. ISBN: 9781638350378.

[89] Anna-Lena Lamprecht. "Track Introduction – Doctoral Symposium 2018". In: *Leveraging Applications of Formal Methods, Verification and Validation. Verification*. Ed. by Tiziana Margaria and Bernhard Steffen. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 451–456. ISBN: 9783030034214. DOI: 10.1007/978-3-030-03421-4_28.

[90] De Liu, Gautam Ray, and Andrew B. Whinston. "The Interaction Between Knowledge Codification and Knowledge-Sharing Networks". In: *Information Systems Research* 21.4 (2010), pp. 892–906. DOI: 10.1287/isre.1080.0217.

[91] Janne Luoma, Steven Kelly, and Juha-Pekka Tolvanen. "Defining Domain-Specific Modeling Languages: Collected Experiences". In: *4th Workshop on Domain-Specific Modeling*. Citeseer. 2004.

[92] Michael Lybecait. "Meta-Model Based Generation of Domain-Specific Modeling Tools". Dissertation. Dortmund, Germany: TU Dortmund University, 2019. DOI: 10.17877/DE290R-20418.

[93] Sylvia Maduenyi, Adunola Oluremi Oke, Olatunji Fadeyi, and Akintunde M Ajagbe. "Impact of Organisational Structure on Organisational Performance". In: *Nigeria: Thesis Submitted to Covenant University* (2015), pp. 354–356.

[94] Tiziana Margaria. "From Computational Thinking to Constructive Design with Simple Models". In: *Leveraging Applications of Formal Methods, Verification and Validation. Modeling*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 11244. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 261–278. ISBN: 9783030034184. DOI: 10.1007/978-3-030-03418-4_16.

[95] Tiziana Margaria and Bernhard Steffen. "Business Process Modelling in the jABC: The One-Thing-Approach". In: *Handbook of Research on Business Process Modeling*. Ed. by Jorge Cardoso and Wil van der Aalst. IGI Global, 2009. DOI: 10.4018/978-1-60566-288-6.ch001.

[96] Tiziana Margaria and Bernhard Steffen. "From the How to the What". In: *Verified Software: Theories, Tools, Experiments*. Ed. by Bertrand Meyer and Jim Woodcock. Vol. 4171. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, pp. 448–459. ISBN: 9783540691471. DOI: 10.1007/978-3-540-69149-5_48.

[97] Tiziana Margaria and Bernhard 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*. Vol. 13036. Lecture Notes in Computer Science. Cham: Springer, 2021. ISBN: 9783030891589. DOI: 10.1007/978-3-030-89159-6.

[98] Tiziana Margaria, Bernhard Steffen, and Manfred Reitenspieß. "Service-Oriented Design: The Roots". In: *Proc. of the 3rd Int. Conf. on Service-Oriented Computing (ICSOC 2005), Amsterdam, The Netherlands*. Vol. 3826. Lecture Notes in Computer Science. Springer, 2005, pp. 450–464. ISBN: 9783540308171. DOI: 10.1007/11596141_34.

[99] Tiziana Margaria, Dominic Wirkner, Daniel Busch, Alexander Bainczyk, Tim Tegeler, and Bernhard Steffen. "DIME Days (ISoLA 2022 Track Introduction)". In: *Leveraging Applications of Formal Methods, Verification and Validation*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 13702. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 367–373. ISBN: 9783031197550. DOI: 10.1007/978-3-031-19756-7_20.

[100] Tiziana Margaria Margaria and Bernhard Steffen. "Service Engineering: Linking Business and IT". In: *Computer* 39.10 (2006), pp. 45–55. ISSN: 0018-9162. DOI: 10.1109/MC.2006.355.

[101] Jeff McAffer, Jean-Michel Lemieux, and Chris Aniszczyk. *Eclipse Rich Client Platform*. 2nd. Addison-Wesley Professional, 2010. ISBN: 0321603788.

[102] M McIlroy, EN Pinson, and BA Tague. "UNIX Time-Sharing System". In: *The Bell system technical journal* 57.6 (1978), pp. 1899–1904.

[103] Peter Mell and Tim Grance. "The NIST Definition of Cloud Computing". In: *NIST Special Publication 800-145* (2011). DOI: 10.6028/NIST.SP.800-145.

[104] Marjan Mernik, Jan Heering, and Anthony M. Sloane. "When and How to Develop Domain-specific Languages". In: *ACM Computing Surveys* 37.4 (Dec. 2005), pp. 316–344. ISSN: 0360-0300. DOI: 10.1145/1118890.1118892.

[105] Maik Merten and Bernhard Steffen. "Simplicity Driven Application Development". In: *Journal of Integrated Design and Process Science (SDPS)* 17 (2013), pp. 9–23. DOI: 10.3233/jid-2013-0008.

[106] Richard J. Mitchell and Institution of Electrical Engineers. *Managing Complexity in Software Engineering*. Computing and Networks. Peregrinus, 1990. ISBN: 9780863411717.

[107] Samer I. Mohamed. "DevOps shifting software engineering strategy Value based perspective". In: *IOSR Journal of Computer Engineering* 17 (2015), pp. 51–57. ISSN: 2278-0661.

[108] Daniel Moody. "The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering". In: *IEEE Transactions on Software Engineering* 35.6 (2009), pp. 756–779. ISSN: 0098-5589. DOI: 10.1109/TSE.2009.67.

[109] Neeshal Munga, Thomas Fogwill, and Quentin Williams. "The Adoption of Open Source Software in Business Models: A Red Hat and IBM Case Study". In: *Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*. SAICSIT '09. Vanderbijlpark, Emfuleni, South Africa: Association for Computing Machinery, 2009, pp. 112–121. ISBN: 9781605586434. DOI: 10.1145/1632149.1632165.

[110] Saravanan Nadason, Ram Al-Jaffri Saad, and Aidi Ahmi. "Knowledge Sharing and Barriers in Organizations: A Conceptual Paper on Knowledge-Management Strategy". In: *Indian-Pacific Journal of Accounting and Finance* 1.4 (2017), pp. 32–41. DOI: 10.52962/ipjaf.2017.1.4.26.

[112] Stefan Naujokat. "Heavy Meta. Model-Driven Domain-Specific Generation of Generative Domain-Specific Modeling Tools". Dissertation. Dortmund, Germany: TU Dortmund University, Aug. 2017. DOI: 10.17877/DE290R-18076.

[113] Stefan Naujokat, Michael Lybecait, Dawid Kopetzki, and Bernhard Steffen. "CINCO: A Simplicity-Driven Approach to Full Generation of Domain-Specific Graphical Modeling Tools". In: *Software Tools for Technology Transfer* 20.3 (2017), pp. 327–354. DOI: 10.1007/s10009-017-0453-6.

[114] Stefan Naujokat, Louis-Marie Traonouez, Malte Isberner, Bernhard Steffen, and Axel Legay. "Domain-Specific Code Generator Modeling: A Case Study for Multi-faceted Concurrent Systems". In: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 8802. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 481–498. ISBN: 9783662452349. DOI: 10.1007/978-3-662-45234-9_33.

[115] P. S. Newman. "Towards an integrated development environment". In: *IBM Systems Journal* 21.1 (1982), pp. 81–107. ISSN: 0018-8670. DOI: 10.1147/sj.211.0081.

[117] Charlene O'Hanlon. "A Conversation with Werner Vogels: Learning from the Amazon Technology Platform: Many Think of Amazon as 'That Hugely Successful Online Bookstore.' You Would Expect Amazon CTO Werner Vogels to Embrace This Distinction, but in Fact It Causes Him Some Concern." In: *Queue* 4.4 (May 2006), pp. 14–22. ISSN: 1542-7730. DOI: 10.1145/1142055.1142065.

[118] Stan Oliver and Kondal Reddy Kandadi. "How to develop knowledge culture in organizations? A multiple case study of large distributed organizations". In: *Journal of knowledge management* (2006). ISSN: 1367-3270. DOI: 10.1108/13673270610679336.

[121] Maria Paasivaara, Casper Lassenius, and Ville T. Heikkilä. "Inter-Team Coordination in Large-Scale Globally Distributed Scrum: Do Scrum-of-Scrums Really Work?" In: *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '12. Lund, Sweden: Association for Computing Machinery, 2012, pp. 235–238. ISBN: 9781450310567. DOI: 10.1145/2372251.2372294.

[123] M. Poppendieck and T. Poppendieck. *Lean Software Development: An Agile Toolkit*. Agile Software Development Series. Pearson Education, 2003. ISBN: 9780133812961.

[126] Laurence Prusak. "The Knowledge Advantage". In: *Planning Review* 24.2 (Jan. 1996), pp. 6–8. ISSN: 0094-064X. DOI: 10.1108/eb054546.

[127] Florian Rademacher, Jonas Sorgalla, Sabine Sachweh, and Albert Zündorf. "Viewpoint-Specific Model-Driven Microservice Development with Interlinked Modeling Languages". In: *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. 2019, pp. 57–66. ISBN: 9781728114422. DOI: 10.1109/SOSE.2019.00018.

[130] Matthew N.O. Sadiku, Sarhan M. Musa, and Omonowo D. Momoh. "Cloud Computing: Opportunities and Challenges". In: *IEEE Potentials* 33.1 (2014), pp. 34–36. ISSN: 0278-6648. DOI: 10.1109/MPOT.2013.2279684.

[131] Mary Sánchez-Gordón and Ricardo Colomo-Palacios. "Security as Culture: A Systematic Literature Review of DevSecOps". In: *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 266–269. ISBN: 9781450379632. DOI: 10.1145/3387940.3392233.

[132] Ronnie E. S. Santos, Fabio Q. B. da Silva, Cleyton V. C. de Magalhães, and Cleviton V. F. Monteiro. "Building a Theory of Job Rotation in Software Engineering from an Instrumental Case Study". In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: Association for Computing Machinery, 2016, pp. 971–981. ISBN: 9781450339001. DOI: 10.1145/2884781.2884837.

[133] Viviane Santos, Alfredo Goldman, and Cleidson RB De Souza. "Fostering effective inter-team knowledge sharing in agile software development". In: *Empirical Software Engineering* 20.4 (2015), pp. 1006–1051. DOI: 10.1007/s10664-014-9307-y.

[134] Gigi Sayfan. *Mastering kubernetes*. Packt Publishing Ltd, 2017. ISBN: 9781788999786.

[135] Douglas C. Schmidt. "Guest Editor's Introduction: Model-Driven Engineering". In: *IEEE Computer* 39.2 (2006), pp. 25–31.

[136] Tobias Schneider and A Wolfsmantel. "Achieving Cloud Scalability with Microservices and DevOps in the Connected Car Domain." In: *Software Engineering (Workshops)*. 2016, pp. 138–141.

[137] Jonas Schürmann. "Functional Synthesis of Type-Safe GraphQL Communication Interfaces". MA thesis. Dortmund, Germany: TU Dortmund University, Mar. 2021.

[139] Jonas Schürmann, Tim Tegeler, and Bernhard Steffen. "Guaranteeing Type Consistency in Collective Adaptive Systems". In: *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 12477. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 311–328. ISBN: 9783030614706. DOI: 10.1007/978-3-030-61470-6_19.

[140] Bran Selić. "Design Languages: A Necessary New Generation of Computer Languages". In: *Leveraging Applications of Formal Methods, Verification and Validation. Modeling*. Ed. by Tiziana Margaria and Bernhard Steffen. Lecture Notes in Computer Science. Cham: Springer International Publishing, Nov. 2018, pp. 279–294. ISBN: 9783030034177. DOI: 10.1007/978-3-030-03418-4_17.

[141] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices". In: *IEEE Access* 5 (2017), pp. 3909–3943. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2017.2685629.

[142] Darja Smite, Nils Brede Moe, Georgiana Levinta, and Marcin Floryan. "Spotify Guilds: How to Succeed With Knowledge Sharing in Large-Scale Agile Organizations". In: *IEEE Software* 36.2 (2019), pp. 51–57. DOI: 10.1109/MS.2018.2886178.

[143] Jonas Sorgalla, Philip Wizenty, Florian Rademacher, Sabine Sachweh, and Albert Zündorf. "Applying Model-Driven Engineering to Stimulate the Adoption of DevOps Processes in Small and Medium-Sized Development Organizations". In: *SN Computer Science* 2.6 (Sept. 2021), pp. 1–25. DOI: 10.1007/s42979-021-00825-z.

[144] Barbara Steffen and Steve Boßelmann. "GOLD: Global Organization aLignment and Decision - Towards the Hierarchical Integration of Heterogeneous Business Models". In: *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*. Ed. by Tiziana Margaria and Bernhard Steffen. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 504–527. ISBN: 9783030034276. DOI: 10.1007/978-3-030-03427-6_37.

[145] Barbara Steffen, Falk Howar, Tim Tegeler, and Bernhard Steffen. "Agile Business Engineering: From Transformation Towards Continuous Innovation". In: *Leveraging Applications of Formal Methods, Verification and Validation*. Ed. by Tiziana Margaria and Bernhard Steffen. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 77–94. ISBN: 9783030891596. DOI: 10.1007/978-3-030-89159-6_6.

[146] Bernhard Steffen, Frederik Gossen, Stefan Naujokat, and Tiziana Margaria. "Language-Driven Engineering: From General-Purpose to Purpose-Specific Languages". In: *Computing and Software Science: State of the Art and Perspectives*. Ed. by Bernhard Steffen and Gerhard Woeginger. Vol. 10000. LNCS. Springer, 2019. ISBN: 9783319919089. DOI: 10.1007/978-3-319-91908-9_17.

[147] Bernhard Steffen and Prakash Narayan. "Full Life-Cycle Support for End-to-End Processes". In: *IEEE Computer* 40.11 (2007), pp. 64–73. ISSN: 0018-9162. DOI: 10.1109/MC.2007.386.

[148] Bernhard Steffen and Stefan Naujokat. "Archimedean Points: The Essence for Mastering Change". In: *LCNS Transactions on Foundations for Mastering Change (FoMAC)*. Vol. 9960. I. Cham: Springer International Publishing, 2016, pp. 22–46. ISBN: 9783319465081. DOI: 10.1007/978-3-319-46508-1_3.

[149] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley, Boston, MA, USA, 2008. ISBN: 0321331885.

[150] Jeff Sutherland and Ken Schwaber. "The scrum papers: Nuts, Bolts and Origins of an Agile Process". In: (2007).

[151] Tim Tegeler, Steve Boßelmann, Jonas Schürmann, Steven Smyth, Sebastian Teumert, and Bernhard Steffen. "Executable Documentation: From Documentation Languages to Purpose-Specific Languages". In: *Leveraging Applications of Formal Methods, Verification and Validation*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 13702. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 174–192. ISBN: 9783031197550. DOI: 10.1007/978-3-031-19756-7_10.

[152] Tim Tegeler, Frederik Gossen, and Bernhard 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)*. 2019, pp. 1–6. ISBN: 9781538659335. DOI: 10.1109/CONFLUENCE.2019.8776962.

[153] Tim Tegeler and Jonas Schürmann. "Evolve: Language-Driven Engineering in Industrial Practice". In: *Electronic Communication of the European Association of Software Science and Technology*, 78 (2018). DOI: 10.14279/tuj.eceasst.78.1089.

[154] Tim Tegeler, Sebastian Teumert, Jonas Schürmann, Alexander Bainczyk, Daniel Busch, and Bernhard Steffen. "An Introduction to Graphical Modeling of CI/CD Workflows with Rig". In: *Leveraging Applications of Formal Methods, Verification and Validation*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 13036. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 3–17. ISBN: 9783030891596. DOI: 10.1007/978-3-030-89159-6_1.

[156] Sebastian Teumert. "Visual Authoring of CI/CD Pipeline Configurations". Bachelor's Thesis. Dortmund, Germany: TU Dortmund University, Apr. 2021.

[157] Sebastian Teumert, Tim Tegeler, Jonas Schürmann, Daniel Busch, and Dominic Wirkner. "Evaluation of Graphical Modeling of CI/CD Workflows with Rig". In: *Leveraging Applications of Formal Methods, Verification and Validation*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 13702. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 374–388. ISBN: 9783031197550. DOI: 10.1007/978-3-031-19756-7_21.

[160] Warren Thorngate. ""In General" vs. "It Depends": Some Comments of the Gergen-Schlenker Debate". In: *Personality and Social Psychology Bulletin* 2.4 (1976), pp. 404–410. DOI: 10.1177/014616727600200413.

[161] Oktay Turetken, Igor Stojanov, and Jos J. M. Trienekens. "Assessing the adoption level of scaled agile development: a maturity model for Scaled Agile Framework". In: *Journal of Software: Evolution and Process* 29.6 (2017). ISSN: 2047-7473. DOI: 10.1002/smr.1796.

[162] Kathryn L. Turk and Michelle Corbin Nichols. "Online Help Systems: Technological Evolution or Revolution?" In: *Proceedings of the 14th annual international conference on Systems documentation: Marshaling new technological forces: building a corporate, academic, and user-oriented triangle*. SIGDOC '96. New York, USA: Association for Computing Machinery, 1996, pp. 239–242. ISBN: 0897917995. DOI: 10.1145/238215.238302.

[163] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. "Quality and Productivity Outcomes Relating to Continuous Integration in GitHub". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. New York, USA: Association for Computing Machinery, 2015, pp. 805–816. ISBN: 9781450336758. DOI: 10.1145/2786805.2786850.

[164] Marcus Voelter. "MD* Best Practices". In: *Journal of Object Technology* 8.6 (Sept. 2009). (column), pp. 79–102. ISSN: 1660-1769. DOI: 10.5381/jot.2009.8.6.c6.

[165] Timo Walter. "Entwicklung einer grafischen DSL für Docker und Kubernetes". MA thesis. Dortmund, Germany: TU Dortmund University, Jan. 2020.

[166] Jan Weckwerth. "Cinco Evaluation: CMMN-Modellierung und -Ausführung in der Praxis". MA thesis. Dortmund, Germany: TU Dortmund University, 2016.

[167] Anna Wiedemann, Nicole Forsgren, Manuel Wiesche, Heiko Gewald, and Helmut Krcmar. "Research for Practice: The DevOps Phenomenon". In: *Commun. ACM* 62.8 (July 2019), pp. 44–49. ISSN: 0001-0782. DOI: 10.1145/3331138.

[168] Laurie Williams. "Integrating Pair Programming into a Software Development Process". In: *Proceedings 14th Conference on Software Engineering Education and Training. 'In search of a software engineering profession' (Cat. No.PR01059)*. IEEE. 2001, pp. 27–36. ISBN: 0769510590. DOI: 10.1109/CSEE.2001.913816.

[169] Laurie Williams, Robert R. Kessler, Ward Cunningham, and Ron Jeffries. "Strengthening the Case for Pair Programming". In: *IEEE Software* 17.4 (2000), pp. 19–25. ISSN: 0740-7459. DOI: 10.1109/52.854064.

[171] Ewelina Wińska and Włodzimierz Dąbrowski. "Software Development Artifacts in Large Agile Organizations: A Comparison of Scaling Agile Methods". In: *Data-Centric Business and Applications: Towards Software Development*. Ed. by Aneta Poniszewska-Marańda, Natalia Kryvinska, Stanisław Jarząbek, and Lech Madeyski. Vol. 4. Cham: Springer International Publishing, 2020, pp. 101–116. ISBN: 9783030347062. DOI: 10.1007/978-3-030-34706-2_6.

[172] Dominic Wirkner. "Merge-Strategien für Graphmodelle am Beispiel von jABC und Git". Diploma thesis. Dortmund, Germany: TU Dortmund University, Feb. 2015.

[173] James P. Womack, Daniel T. Jones, and Daniel Roos. *The machine that changed the world: The story of lean production–Toyota's secret weapon in the global car wars that is now revolutionizing world industry*. Simon and Schuster, 2007. ISBN: 0743299795.

[174] Nils Wortmann, Malte Michel, and Stefan Naujokat. "A Fully Model-Based Approach to Software Development for Industrial Centrifuges". In: *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 9953. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 774–783. ISBN: 9783319471693. DOI: 10.1007/978-3-319-47169-3_58.

[175] Ming-Wei Wu and Ying-Dar Lin. "Open Source Software Development: An Overview". In: *Computer* 34.6 (2001), pp. 33–38. ISSN: 1558-0814. DOI: 10.1109/2.928619.

[176] Ravi Teja Yarlagadda. "DevOps and Its Practices". In: *International Journal of Creative Research Thoughts (IJCRT), ISSN* 9 (Mar. 2021). ISSN: 2320-2882.

[177] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. "The Impact of Continuous Integration on Other Software Development Practices: A Large-Scale Empirical Study". In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 60–71. ISBN: 978-1-5386-2684-9. DOI: 10.1109/ASE.2017.8115619.

[178] Liming Zhu, Len Bass, and George Champlin-Scharff. "DevOps and Its Practices". In: *IEEE Software* 33.3 (Apr. 2016), pp. 32–34. ISSN: 0740-7459. DOI: 10.1109/MS.2016.81.

[179] Ahmed Zia, Waleed Arshad, and Waqas Mahmood. "Preference in using agile development with larger team size". In: *International Journal of Advanced Computer Science and Applications* 9.7 (2018), pp. 116–123.

[180] Philip Zweihoff. "Aligned and Collaborative Language-Driven Engineering". Dissertation. Dortmund, Germany: TU Dortmund University, 2022. DOI: 10.17877/DE290R-22594.

[181] Philip Zweihoff, Tim Tegeler, Jonas Schürmann, Alexander Bainczyk, and Bernhard Steffen. "Aligned, Purpose-Driven Cooperation: The Future Way of System Development". In: *Leveraging Applications of Formal Methods, Verification and Validation*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 13036. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 426–449. ISBN: 9783030891596. DOI: 10.1007/978-3-030-89159-6_27.

Online References

[3] Apache Maven Project. *Best Practice - Using a Repository Manager*. Online. URL: https://maven.apache.org/repository-management.html (visited on 01/15/2023).

[5] Atlassin. *What is DevOps?* Online. URL: https://www.atlassian.com/devops (visited on 01/15/2023).

[12] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. *Manifesto for Agile Software Development*. Online. 2001. URL: http://agilemanifesto.org (visited on 01/15/2023).

[13] Oren Ben-Kiki, Clark Evans, and Ingy döt Net. *YAML Ain't Markup Language (YAML™) Version 1.2*. Online. 2009. URL: https://yaml.org/spec/1.2/spec.html (visited on 01/15/2023).

[32] *Cinco SCCE Meta Tooling Suite*. Online. URL: http://cinco.scce.info (visited on 01/15/2023).

[38] *Definition of Reproducible Builds*. Online. URL: https://reproducible-builds.org/docs/definition/ (visited on 01/15/2023).

[44] Docker Inc. *Docker Registry*. Online. URL: https://docs.docker.com/registry/ (visited on 01/15/2023).

[45] Docker Inc. *Use multi-stage builds*. Online. URL: https://docs.docker.com/develop/develop-images/multistage-build/ (visited on 01/15/2023).

[46] *Doxygen*. Online. URL: https://www.doxygen.nl/index.html (visited on 01/15/2023).

[50] Martin Fowler. *Continuous Integration*. Online. May 2006. URL: https://www.martinfowler.com/articles/continuousIntegration.html (visited on 01/15/2023).

[51] Martin Fowler. *Continuous Integration (original version)*. Online. Sept. 2000. URL: https://martinfowler.com/articles/originalContinuousIntegration.html (visited on 01/15/2023).

[52] Martin Fowler. *ExtremeProgramming*. Online. July 2013. URL: https://www.martinfowler.com/bliki/ExtremeProgramming.html (visited on 01/15/2023).

[58] GitLab Inc. *What is pipeline as code?* Online. URL: https://about.gitlab.com/topics/ci-cd/pipeline-as-code/ (visited on 01/15/2023).

[59] Glosbe. *Lingualization*. Online. URL: https://glosbe.com/en/en/lingualization (visited on 01/15/2023).

[62] Michele Guerriero. *DICER: A Tool for Model Driven Infrastructure as Code of Data Intensive Applications*. Online. URL: https://github.com/dice-project/DICER/wiki/DICER:-A-Tool-for-Model-Driven-Infrastructure-as-Code-of-Data-Intensive-Applications (visited on 01/15/2023).

[69] *Inter- vs. Intra-: What is the Difference?* Online. URL: https://www.merriam-webster.com/words-at-play/intra-and-inter-usage (visited on 01/15/2023).

[70] ISO 639-2 Registration Authority. *Alpha-2 Code (no) Search Result - Codes for the representation of names of languages (Library of Congress)*. Online. URL: https://www.loc.gov/standards/iso639-2/php/langcodes_name.php?iso_639_1=no (visited on 01/15/2023).

[71] *Javadoc Tool*. Online. URL: https://www.oracle.com/java/technologies/javase/javadoc.html (visited on 01/15/2023).

[79] Kharnagy. *File:Devops-toolchain.svg*. Online. Sept. 2016. URL: https://commons.wikimedia.org/wiki/File:Devops-toolchain.svg (visited on 01/15/2023).

[111] National Vulnerability Database. *NVD - CVE-2021-44228*. Online. URL: https://nvd.nist.gov/vuln/detail/CVE-2021-44228 (visited on 01/15/2023).

[116] Colm O'Connor. *The Norway Problem - why StrictYAML refuses to do implicit typing and so should you*. Online. URL: https://hitchdev.com/strictyaml/why/implicit-typing-removed/ (visited on 01/15/2023).

[119] OpenAPI Initiative. *OpenAPI Specification v3.1.0*. Online. Feb. 2021. URL: https://spec.openapis.org/oas/latest.html (visited on 01/15/2023).

[120] Addy Osmani, Sindre Sorhus, Pascal Hartig, Stephen Sawchuk, Colin Eberhardt, Sam Saccone, Arthur Verschaeve, Fady Samir Sadek, and Gianni Chiappetta. *TodoMVC - Helping you select an MV* framework*. Online. 2021. URL: https://todomvc.com/ (visited on 01/15/2023).

[122] *Pipeline Editor | GitLab*. Online. 2022. URL: https://docs.gitlab.com/ee/ci/pipeline_editor/ (visited on 01/15/2023).

[124] Andra Postolache. *Barriers to Knowledge Transfer and How to Overcome Them*. Mar. 2020. URL: https://www.quandora.com/barriers-knowledge-transfer-how-to-overcome-them/ (visited on 01/15/2023).

[125] Tom Preston-Werner. *Semantic Versioning 2.0.0*. Online. June 2013. URL: https://semver.org/spec/v2.0.0.html (visited on 01/15/2023).

[128] *Rig Definition & Meaning - Merriam-Webster*. Online. URL: https://www.merriam-webster.com/dictionary/rig (visited on 01/15/2023).

[129] *Rigging Definition & Meaning - Merriam-Webster*. Online. URL: https://www.merriam-webster.com/dictionary/rigging (visited on 01/15/2023).

[138] Jonas Schürmann. *Knobster*. Online. URL: https://knobster.jonas-schuermann.name/ (visited on 01/15/2023).

[155] Sebastian Teumert. *Rig | Low-Code CI/CD Modeling*. Online. URL: https://scce.gitlab.io/rig/ (visited on 01/15/2023).

[158] The Free On-line Dictionary of Computing. *Parkinson's Law of Data*. Online. 2003. URL: https://encyclopedia2.thefreedictionary.com/Parkinson%5C%27s+Law+of+Data (visited on 01/15/2023).

[159] The Kubernetes Authors. *Viewing Pods and Nodes*. Online. URL: https://v1-22.docs.kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/ (visited on 01/15/2023).

[170] Robert Williams. *Pipelines-as-Code: How to improve speed from idea to production*. Online. URL: https://about.gitlab.com/blog/2022/01/18/pipelines-as-code/ (visited on 01/15/2023).