Dimensions of Software Configuration

On the Configuration Context in Modern Software Development

Norbert Siegmund
Leipzig University

Nicolai Ruckel
Bauhaus-Universität Weimar

Janet Siegmund
Chemnitz University of Technology

ABSTRACT

With the rise of containerization, cloud development, and continuous integration and delivery, configuration has become an essential aspect not only to tailor software to user requirements, but also to configure a software system's environment and infrastructure. This heterogeneity of activities, domains, and processes blurs the term configuration, as it is not clear anymore what tasks, artifacts, or stakeholders are involved and intertwined. However, each research study and each paper involving configuration places their contributions and findings in a certain context without making the context explicit. This makes it difficult to compare findings, translate them to practice, and to generalize the results. Thus, we set out to evaluate whether these different views on configuration are really distinct or can be summarized under a common umbrella.

By interviewing practitioners from different domains and in different roles about the aspects of configuration and by analyzing two qualitative studies in similar areas, we derive a model of configuration that provides terminology and context for research studies, identifies new research opportunities, and allows practitioners to spot possible challenges in their current tasks. Although our interviewees have a clear view about configuration, it substantially differs due to their personal experience and role. This indicates that the term configuration might be overloaded. However, when taking a closer look, we see the interconnections and dependencies among all views, arriving at the conclusion that we need to start considering the entire spectrum of dimensions of configuration.

ACM Reference Format:
Norbert Siegmund, Nicolai Ruckel, and Janet Siegmund. 2020. Dimensions of Software Configuration: On the Configuration Context in Modern Software Development. In *Proceedings of The 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software configuration is a hot topic in research and industry [33]. Despite its importance, there are many different views about what aspects comprise configuration and how they interact. For example, in combinatorial testing [18, 24], configuration is usually seen as a set of input variables and parameters to a program that needs to be tested; in software product lines [1, 3], configuration corresponds to a selection of features or configuration options for generating a program variant with a desired functional behavior; in optimization [31, 37], the set of configuration options and parameters are regarded as configuration for optimizing non-functional properties; and in the deployment process [33], configuration is a means to define where, when, what, and how to deploy software artifacts.

There are many more areas related to configuration, such as virtualization, provisioning of software, and machine learning that all come with their own objectives, problems, and best practices. This diversity might be one reason why a holistic view on configuration does not exist in software engineering research. Another reason might be that there is no obvious connection between configuration activities in these different fields. However, the way software is developed has changed substantially in recent years due to innovations in tools, architectures, and processes, such as virtualization and containerization [8, 43], continuous integration and delivery [9], DevOps [12, 35], microservices and serverless systems [2, 23], cloud infrastructure [4], and deployment automation [11]. All these areas involve a non-trivial amount of configuration, and, more importantly, highly interact with each other, which one interviewee describes this way: These developments framed the term *infrastructure as code*, which can be interpreted as codifying the configuration of the infrastructure and residing it next to the application's code [**?**]. *"With the recent trends in architecture, such as microservices, it [configuration] is becoming more important. [...] the more distributed, and, thus, complex systems are, the more complex the problems get. So, I use tools to catch these problems. A simple example is when I start a Spring Boot application and this works standalone, then it is totally trivial. As soon as I start booting multiple of them, the traditional way would be to take Spring Cloud. [...] But then every Spring Boot application taken into Spring Cloud has a properties file with not 2 but 50 entries. All configuration. All of them are there to configure the tools that are supposed to make life easier. Hence, there is an exponential explosion of configuration options, which all can break. And this is why I would say it [configuration] is becoming more important."*$_{I1}$

This high degree of interaction requires researchers to overcome the isolated view of configuration, and instead integrate different views of configuration into research, making transfer to a realistic setting easier. For example, combinatorial testing aims at finding bugs in a program by covering a diverse set of combinations of input parameters and options. However, we found in our interviews that not all options and settings are available at the same time due to a step-wise configuration process. Also, the availability of data for testing depends on the deployment stage whereas knowledge about configuration settings is distributed over a diverse set of stakeholders. Hence, combinatorial testing needs to take this diversity into account to be more generally applicable in practice.

ESEC/FSE 2020, 8–13 November, 2020, Sacramento, California, United States

In addition to a limited applicability, research results could even be misleading when not considering the entire context of configuration. Consider, for example, the vast amount of research on performance optimization of configurable software systems, such as sampling relevant configurations [16], accurate performance prediction [21], or finding performance-optimal configurations [25]. These techniques usually consider only a static environment with a constant workload, which rarely occurs in practice due to virtualization and containerization. Having different stages with different data, and with different (virtualized) hardware makes a performance model hardly transferable, even when using transfer learning to reuse knowledge in different settings (e.g., [13, 14]): Often, performance-influencing factors are even unknown, such as optional JVM flags when running a Java application in a container.

In essence, we do not know which types of configuration exist, which binding times of configuration values are prevalent for which types of configurations, which stakeholders are involved in configuration, and what tasks during software development require configuration. Furthermore, all these factors interact, but we do not know how. Thus, we set out to shed light on which factors comprise configuration and how they interact. This work is the first attempt to frame the term configuration into a comprehensive model by interviewing a diverse set of practitioners about their experience with configuration. By considering developers, testers, consultants, DevOps, and cloud software architects, we obtain a broad view on configuration, ranging from software configuration via feature toggles over technical configuration via environment variables to infrastructure and development configuration via cloud settings and *continuous integration and delivery (CI/CD)* customizations. By further analyzing two qualitative studies in a meta study, we broaden the scope and saturate on the values and dimensions of configuration. We found that configuration is a complex task cutting across the entire software life cycle, involves a diverse set of stakeholders, impacts infrastructure and development activities, and is, most importantly, a highly context-sensitive process.

In building a comprehensive model about aspects of configuration, we provide a framework for researchers to place their work and studies into context and derive what factors need to be considered, as we show in Section 4. This helps to understand how research results apply to what interacting aspects of configuration, including their limitations and practical relevant circumstances. Furthermore, it can guide research of each isolated area to ask the right questions, and also how to combine these areas to increase practical relevance. For practitioners, the framework can be a means for orientation, such that stakeholders can place their current configuration activity in the model and derive possible interactions with and implications for other stakeholders.

Scope and Limitations. Clearly, this work is not exhaustive, as there are many domains, such as embedded systems, operating systems, intelligent systems, and cloud applications that all have their own perspective on configuration. We focus on enterprise applications, cloud applications, and rich clients, thereby addressing a large and relevant field. Our performed mixed method approach using a literature study of related work substantially widens the scope and validates our findings. Further, we selected a wide variety of stakeholders to view configuration through different lenses.

2 METHODOLOGY

Building a model on configuration requires a sound methodological approach, as a single study is insufficient to explore the diverse perspectives on configuration. Our overall methodological approach follows grounded theory [39], which is rooted in social science for theory building based on the analysis of mostly qualitative data, such as interviews, questionnaires, and literature analyzes.

Contrary to common research practice in software engineering, grounded theory starts with a single question and gathering qualitative data. Questions for interviews are not driven by related studies not to be biased in a certain direction. Instead, the collected data is reviewed and analyzed continuously, such that interview questions are adapted during the course of developing ideas, concepts, and a model in the end. The rationale of using grounded theory here is that developing a model about configuration requires starting with an unbiased view to cover as many aspects of configuration as possible rather than starting with related studies that already frame the picture of configuration.

We used a mixed method approach: First, we conducted several interviews to explore as many aspects as possible that involve configuration (see Section 2.1), based on which we developed an initial model of configuration. Here, we extracted 7 dimensions with 32 values in total. Second, we analyzed two closely related papers to (i) validate our findings and (ii) extend our model, yielding 1 additional dimension and 15 new values. Finally, we selected 16 additional research papers (18 in total) from different domains that involve the aspect of configuration to (iii) verify the applicability of our model and (iv) identify gaps in current research leading to new research opportunities. Next, we describe our methodology in detail, starting with the interviews.

2.1 Qualitative Study

Participants. To understand the view of practitioners on configuration, we conducted semi-structured interviews with 11 practitioners (cf. Table 1) from 9 different companies ranging from small (dozens) over medium (hundreds) to large (hundreds of thousands of employees): codecentric AG, Salesforce, Red Hat, Xceptance, 4Soft, Regiocom, REWE Digital, Accenture, and an e-commerce company. The companies are distributed both locally in Europe and globally. With this broad sample, our results are relevant for many software companies and practitioners.

Interview. We designed and continuously refined several interview questions to guide us in exploring aspects of configuration.[1]. They can be divided into five categories. The first category collects background information on developers (summarized in Table 1). The remaining categories cover several aspects of configuration. We start with general questions regarding configuration, so that we capture the intuitive understanding of practitioners' view on configuration. This way, we could shed light on how practitioners, depending on their role, view configuration, which helps us to understand the different dimensions that affect configuration. Then, we ask practitioners about their every-day work with configuration. This way, we can understand the role that configuration plays for

[1]The entire final set of questions and all material are available at: https://github.com/AI-4-SE/Dimensions-of-Software-Configuration

Table 1: Overview of our interviewees (ID) including a summary of their experience (Exp.) in years and expertise.

ID	Exp.	Role	Domains
I1	10 y	Consultant, Senior Developer	DevOps, Fullstack
I2	14 y	Developer	Backend, Microservice
I3	22 y	Developer, Software Architect	DevOps, Microservice
I4	15 y	Developer, Test Engineer	Backend
I5	11 y	Senior Developer	Backend, Microservice
I6	7 y	Software Architect	Backend, Microservice
I7	6 y	Team Lead	Fullstack
I8	19 y	Cloud Foundation Architect	Backend
I9	5 y	Developer	Backend
I10	7 y	Consultant, Tester, Developer	Fullstack
I11	20 y	Senior Software Engineer	Frontend

practitioners. Next, we asked practitioners to describe challenges when working with configuration. This allowed us to determine where practitioners struggle. Finally, we asked about best practices and areas for improvement by including hypothetical questions. This way, we can identify relevant questions, giving guidance on configuration research.

Execution and Analysis. Each interview was conducted with one practitioner and the first two authors, except on one occasion where we interviewed two practitioners at once. Each interview lasted about 90 minutes and was recorded and transcribed (available on the project's website). Based on the transcription, we applied a card-sorting approach to identify higher-order topics [10].

The card-sorting process has been conducted by all authors of the paper to include an unbiased view during concept generation. The process was as follows: We conducted the card sorting participant by participant and question by question. Every author read through a question and made notes, such as mentioned artifacts, problems during configuration, involved stakeholders, or guidelines and processes in the company. After these individual readings and marked higher-order topics, we compared and discussed our findings. After reaching consensus that all higher-order topics for one response have been identified, we moved on to the next question.

Open questions and missing information were discussed to possibly phrase a new question in the upcoming interviews, following grounded theory. For the annotations, we quickly saw categories emerging, such as artifacts, the intent of the configuration, or when configurations are made. We could phrase questions for most of these categories (which later end up in the dimensions), such as *what* is configured, *when* it is configured, *how* the configuration process is handled, etc. Since answers to these questions reappeared in every interview, we came to dimensions (questions/categories) with different values (different answers to the same question). Moreover, we could also spot interactions among dimensions, such as, '*Who* configures *what* at *which* state?' This process has been performed for all interviews. Afterwards, we went through all transcripts again to make sure that dimensions, artifacts, and interactions that we extracted from later interviews have not been missed in the earlier interviews. Since three authors performed this task first individually per question (to not bias a personal opinion) and afterwards discussing and comparing the results, the reported dimensions and artifacts are an agreement. The only (rare) cases with disagreement were when it was not clear whether to further split dimensions. For instance, the intent dimension is rather broad and could be divided into smaller, multiple dimensions, such as one for functional and one for non-functional configuration. However, this would have cluttered the model with too many dimensions, of which a large portion may never be relevant. Instead, we chose a modeling approach where each dimension is possibly always involved when it comes to software configuration.

2.2 Meta Study

To validate the findings of the interviews, extend the scope, and show how to incorporate previous work into a holistic model, we analyzed two papers that reported on related interviews from practitioners, from which we can infer additional aspects of configuration. The first two authors individually read each paper carefully and made annotations/cards, similar to the card-sorting phase. We then compared the annotations with the dimensions and values already found. Despite a large agreement, we found additional values (due to the increased scope of these papers) and also the new complexity dimension caused by the implementation-specific and technical interviews in [6]. Interestingly, for the remaining 16 papers, we found no new dimension or value, which substantially increases our confidence to have covered a substantial spectrum of configuration aspects. Next, we give a short summary about the papers and explain their relevance.

Study 1: What is a Feature? Berger and others conducted a study about the meaning, usage, and interpretation of the term "feature" [6]. They interviewed six stakeholders (developers, product managers, architects) of three companies (Keba, Opel, Danfoss) to understand their use of features in devising configurable software systems and product lines, how they model features, how features arise during development, and how they are used. As features are a central element of configuration, the study represents an important source of possible dimensions of configuration.

To relate the results of the study to our configuration model, two authors of this paper read carefully through the interview responses (summarized in Table 2 in the paper by Berger and others). Contrasting to our terminology, Berger and others summarized their responses based on "facets of features". For each facet, we identified whether it is comparable to an existing dimension from our own interviews. If we could not find a matching dimension, we discussed whether the facets represent a new dimension of configuration. Moreover, when finding a matching dimension, we verified whether the values of a dimension has already been covered by our interview questions. If not, we added a new value. In Section 3, we report which dimensions and values have been added this way.

Study 2: Software Configuration Engineering in Practice. Sayagh and others conducted a qualitative study about the engineering process of run-time configuration options in industry [33]. They identify challenges and best practices about creating, maintaining, and using configuration options across different software artifacts. As another result, they extracted nine activities related to configuration, which partially map to our dimensions of configuration.

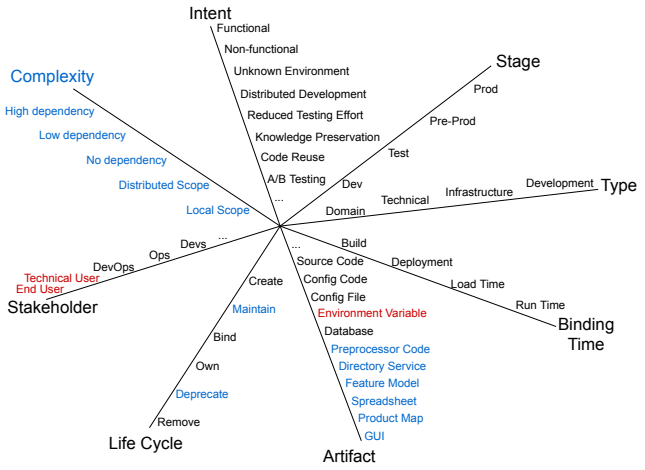


Figure 1: Dimensions of configurations and their values, encoded by origin of study: our study$_{S0}$ as black, [6]$_{S1}$ as blue, and [33]$_{S2}$ as red.

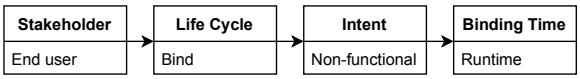
Moreover, they identified 24 recommendations for improving software quality with respect to configuration and 22 challenges that developers face.

The study has been conducted in three phases: First, the authors interviewed 14 software engineers from 13 companies (government to banking system vendors) to obtain common topics of configuration activities using card sorting [30]. Next, they conducted a survey to extend the scope and refine the answers, yielding 229 responses. Finally, they performed a literature survey on the identified activities to complement the obtained recommendations with academic approaches, for which they also used card sorting. We went through the responses and identified activities and match them with the previously revised model to possibly extend it.

3 RESULTS: A MODEL ON CONFIGURATION

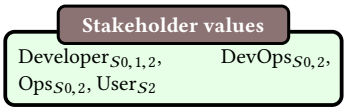
We call the extracted higher-order topics playing part in configuration *dimensions of configuration*. We discuss each dimension and explain their meaning based on the interviews. Based on our research methodology, we iteratively explore more dimensions and values by taking related interviews and survey answers of the two studies (Section 2.2) into account. Due to space limitations, we report only on the final model (not intermediate steps), but highlight at the beginning of each dimension the study from which their values originate with the following subscript: our study $_{S0}$, [6] $_{S1}$, and [33] $_{S2}$. To give an overview of our model, we summarize the origin of the dimensions of configurations and their values from the different sources of our analysis in Figure 1.

Formalization. To apply our model to giving context to research studies or exploring new research opportunities, we formalize the dimensions and values as a directed acyclic graph (DAG). Annotated vertices represent dimensions of our model, where an annotation represents a value, as shown in Figure 2 (Page 9). This way, a single dimension can occur multiple times in the graph with different values. Edges represent interactions between two dimensions and are, thus, directed. For example, if we analyze how end users (*stakeholder* dimension) configure (bind an option in the *life cycle* dimension) for the purpose of performance optimization (*intent* dimension) considering only run-time options (*binding time* dimension), we will obtain the following graph:



From this graph, we can immediately see possible generalizations and limitations by comparing it with our model in Figure 1. For instance, we cannot know how end users would configure load-time options or what the influence of different artifacts is on the way how end users optimize their software systems. Interestingly, also the type of configuration has not been specified, so it is unclear whether the users configure options more related to the application's domain or more related to infrastructure or technical issues. In our literature analysis from 18 papers, we saw such examples and show them on our supplementary website. Often, unspecified dimensions lead to ambiguity in interpreting results and general questions about transferring research to industry. Our model helps in identifying ambiguity and missing dimensions in research involving configuration.

3.1 Stakeholder



This dimension describes everyone who deals with configuration. Traditionally, there are three groups of stakeholders with clearly separated roles: *Users* configure a software system according to their needs, leading to the development of software product lines and configurable software systems [3]. *Operations people* administrate, maintain, and configure the underlying hardware system, the operating system, firewalls, and other environmental and infrastructure related systems. The configuration here has mainly the goal to optimize non-functional properties, such as security, reliability, and resource usage. Finally, *developers* may provide an initial configuration that should work for most scenarios and usually create configuration options in the first place. All of those stakeholder categories can be further divided to show role-specific aspects.

With the DevOps age, more stakeholders are affected by configuration, and the clear separation between these groups of stakeholders blurred. Most prominently, the role of DevOps has blurred the former strongly separated tasks of software development, software maintenance, and operation. Now, a developer may be responsible for configuring the environment of the software system (e.g., using Docker) and the infrastructure (e.g., database connection, microservice deployment, resource management, and network communication), as one interviewee stated: *"With the DevOps approach, requirements have changed [...] I know older developers, who need to change their mindset that they now own their run time environment. So, they not only need to understand the technology and what happens there, but also give recommendations for configurations or apply configurations on their own and take responsibility for them."*$_{I8}$

Consequences: The involvement of different stakeholders from different backgrounds in the configuration process strongly influences how configuration is presented, validated, and maintained. Thinking about configuration in terms of stakeholders will arrive at variability models whose presence of configuration options may change depending on the stakeholder: *"I give some thoughts about what is required, what is maybe even optional, and what should be hidden from the end customer. And in some way, I do the same for the internal stakeholders. He configures the things that are hidden but also for him, there is the question of what should he know, what should he query from others or get from somewhere else, what is optional, what should be avoided, or what is a deprecated feature that you can configure in theory, but we never want to activate it again. So, we want to have an interface description for a configuration to follow it in a way that I know how it works."*$_{I9}$

Thus, when working with configurations, be it in research (e.g., for performance improvements) or industry (e.g., planning which artifacts are used for an option and how to change the values), practitioners need distinct mechanisms how stakeholder-dependent configurations can be modeled, validated, and stored in a suitable way, which interacts with other dimensions of configuration and has not yet been addressed properly by the literature.

3.2 Type of Configuration

Type values

Development$_{S0,1,2}$, Technical$_{S0,1,2}$, Infrastructure$_{S0,1,2}$, Domain$_{S0,1,2}$

Types of configuration describes what should be configured. The interviewees identified two main types of configuration: *"The* domain-specific *configuration helps us [...] to faster provide and customize functions to different customers. And* technical configuration *can break many things if done wrong and would need more attention."*$_{I7}$.

A *domain-specific configuration* targets the software system from an end user perspective to tailor the system according to user's (mostly functional) requirements and often via feature toggles at run time. By contrast, *technical configuration* targets mainly the environment of a system or its deployment and hosting process (e.g., infrastructure, system libraries, and development tools). The increasing trend of virtualized hardware and the increasing complexity of systems has emphasized the need for such a technical configuration. Interestingly, technical configuration involves different stakeholders (e.g., developer, DevOps, Ops) compared to domain-specific configuration and usually is performed on different artifacts (e.g., automation scripts, Docker files, Kubernetes templates), and at different binding times (e.g., compile time, deployment time).

Technical configuration comprises *infrastructure configuration* and *development configuration*, as the kind of tools, configuration artifacts, and configuration effort differ substantially. Infrastructure configuration refers to adjusting a software system to the underlying hardware and software, such as connecting to the correct database system, using a specific port, or setting environmental variables in a Docker file. Development configuration as stated in the interviews involves setting up development tools, such as IDEs, and build tools (e.g., implementing a Maven or Gradle build script), the build and testing process (e.g., using Jenkins as the continuous integration servers), and automating the deployment process to different stages (e.g., testing or production) and to different environments (e.g., cloud providers or cloud infrastructures).

Consequences: The different types of configuration have their own properties: *"With a technical configuration, you define it once and then it is automatically deployed and will be bound at application start at latest, whereas the domain-specific configuration is virtually in constant alteration depending on how the user changes it."*$_{I6}$ These properties, which are mainly interactions with other dimensions of configuration, require their own tools, practices, and training. Thus, it is surprising that the literature usually does not differentiate or clearly articulate types of configuration [20, 27, 33]. For instance, configuration in the product lines community refers to deriving functional different variants of a software system[1] (i.e., domain-specific configuration). But, technical configuration influences domain-specific configuration and vice versa (cf. Section 4). So, in a realistic setting, it is difficult to ignore other types of configurations. Instead, when conducting experiments in realistic settings, these confounding factors need to be accounted for.

We also found that the terminology is not well standardized and may change depending on the stakeholder's perspective. That is, an infrastructure configuration can mean configuring the server infrastructure (as we have defined it), but also the development configuration: *"I usually work on infrastructure configuration at project level. That means I configure a Jenkins job, make sure that all the post and pre steps are called at the right time. [...] This is, however, way easier than a technical configuration, that is, the configuration of the project itself, because there are simply more open parameters."*$_{I4}$ Thus, a well-defined terminology that is also accepted by practitioners can avoid confusion and help to separate configuration tasks, which then can be handled by different expert groups. The distinction within technical configuration is useful also in other regards as one interviewee mentioned that it would be a best practice to unify the development configuration within a team (e.g., where to put curly brackets), which is hardly possible for infrastructure configuration. But, assuming that all such (coding) guidelines can be fully encoded in an IDE configuration, it becomes part of the software configuration process, similar to infrastructure as code, but on a different level of abstraction. Thus, clarifying the type of configuration is a suitable way to organize responsibilities and to introduce conventions avoiding more configuration.

3.3 Binding Time

Binding Time values

Build$_{S0,1}$, Deployment$_{S0}$, Load Time$_{S0,1}$, Run Time$_{S0,1,2}$

Binding time refers to the event of binding a configuration option to a certain value. The binding time of an option varies largely, depending on build time (e.g., in form of build scripts), deployment time (e.g., in form of configuration files), to load- and run time (e.g., in form of properties files, databases, command-line arguments, or user interfaces). Hence, binding time strongly interacts with the dimension's stakeholder, type of configuration, and configuration artifact: *"Often, we [configure] domain-specific things at compile time. That are mostly properties files, sometimes also derived classes. But we have also the possibility to change certain settings during run time. So, we have outsourced some configurations to configuration files and databases and there you can change things at run*

time. And then, we have a third [binding time]: during deployment. This is mainly in the direction of containers and Kubernetes. There, we try to extract the configuration files from the actual programming artifacts to put them next [to the code]. And during deployment, we use then a specific version of configuration."$_{I7}$

Consequences: Thinking about an appropriate binding time of configuration options is an undervalued task in academia and practice. We found that binding time is not fixed and should be changed when technical solutions make it possible: *"Usually, microservices are relatively fast to deploy, so that, most of the time, dynamic configuration is not needed. So, you reconfigure something in a file and redeploy the container."*$_{I2}$ Understanding which binding time is suitable for what type of configuration and which are the pros and cons in the context of the other dimensions of configuration deserves more research. Nevertheless, as identified best practice, binding time should generally be as early as possible to enable better validation and follow the principle of fail fast.

3.4 Configuration Artifact

Artifact values

Source Code$_{S0,1}$, Configuration Code$_{S0,1,2}$, Configuration File$_{S0,1,2}$, Database$_{S0,1,2}$, Commandline parameter$_{S2}$, Environment Variable$_{S2}$ Preprocessor Code$_{S1}$, Directory Service$_{S1,2}$, Feature Model$_{S1}$, Spreadsheet$_{S1}$, Product Map$_{S1}$, GUI$_{S1}$

This dimension describes how configuration manifests in development artifacts. Nowadays, configurations are distributed in many artifacts that exhibit their own structure, syntax, and semantic. We found that simple configuration files, such as properties files or ini files, are favored by our interviewees: *"Configuration should be easy. It should have an easy format that is also comprehensible and that is easily readable by humans and has possibly few indirections. Frankly, I don't want to create a programming language for configuration."*$_{I11}$. The main argument is that properties files are easy to understand and change. An improved version are XML files, as *"I have something like a schema behind, which tells me 'there are only numbers allowed' and you enter a string. Or, this tag is allowed only for these children, but this element is wrongly structured."*$_{I11}$, which helps to validate configurations.

The more complex artifacts are build scripts, container descriptions (e.g., Docker files), and automation scripts (e.g., Ansible playbooks or Jenkins files), which we call *configuration code*.[2] An often stated problem is that they have their own syntax and semantics, but no proper type system:*"In Kubernetes, I have difficulties to cope with all the YAML files. I can't see a structure behind them."*$_{I11}$ Thus, a stakeholder needs to learn by conventions how to write automation files and Docker files, but has no type checker or other validation tool to verify whether the defined configuration is feasible.

Consequences: There is a clash of interests that affects the choice of artifacts: understandability and ease of use versus expressiveness. Developers prefer configuration artifacts that are easy to read, manipulate, and comprehend. The ideal form seems to be key-value pairs with additional comments on the meaning, value ranges, and effect of the configuration option as well as on possible interactions with other options. Moreover, an improved variant with user-defined types of configuration options to limit and validate configuration values (e.g., as done with XML schema) reduces the danger of configuration errors and eases maintenance.

Unfortunately, since modern architectures are often distributed among several services, nodes, hardware systems, and require a complex environment configuration with deployment dependencies and infrastructure constraints, configuration is transforming into a programming task. That is, configuration must respect several conditions and has not just a single value binding, but requires, for example, executing scripts or triggering other deployment processes. Hence, to describe several configuration conditions, expressiveness is needed. It is unclear whether and how both worlds (understandability and expressiveness) can be unified. But, it is clear that, with each new tool or framework that brings its own semantic, the learning curve grows, the maintenance effort increases, and inconsistencies across multiple configuration artifacts rises [32].

3.5 Stage

Stage values

Dev$_{S0,1,2}$, Test$_{S0,1,2}$, Pre-Production$_{S0,2}$, Production$_{S0,1,2}$

This dimension describes that configuration happens in different stages of the development process, such as development or testing. Each stage usually describes a different infrastructure environment of the running system, variables, such as the JVM class path, and available resources, such as database systems. Additionally, it is often strongly connected to a stage in the CI/CD process, in which a software system is deployed and executed. Typical stages include testing, pre-production, and production.

Consequences: Environments differ for the same software systems, depending on the stages in which the software is currently running. This has severe consequences, as configuration must change depending on the stage and environment: *"So, you have your application, which you can configure. Then you have your application in the environment, in which it is deployed, and there is an additional configuration that diverges."*$_{I3}$. This implies two things: First, we need a structured way to apply configurations only for specific environments (similar to the findings of [33]). Second, we need a way to describe which options are valid and relevant in what environment. These are two novel aspects that lack explicit support in current variability modeling languages and tools [15, 26].

A further aspect is that the same configuration option can be bound via different configuration artifacts. For example, environment variables can be passed to the CI server via configuration of a Docker file, via configuration of the CI server itself, or via configuration of automation scripts (e.g., with Puppet or Ansible). So, the question arises which stakeholder is responsible for which artifact and what configuration should be bound by which mechanism and at which binding time? Furthermore, who owns the values when present multiple times and which artifact precedes others? *"I believe systems are complex to configure where the configuration is split among different layers and one layer overwrites another one and at the end, you can overwrite something at the command line. This arrives at a degree of complexity where you don't exactly know when I turn on this switch, what happens in the end [...]?"*$_{I11}$

[2]The artifact is often referred to infrastructure as code, which is, however, only a subset of all codified configuration and automation processes that we comprise under this term.

Stages influence, however, the configuration process in other ways. Later stages often involve secret keys and credentials to hide other confidential information. Usually, developers have no access to these configuration values, so an entirely new mechanism of deploying a configuration and obtaining configuration values (e.g., from a vault), has to be implemented and tested. This is one of the reasons why configuration errors might not be caught before a system is in production. Similarly, stages differ in the amount and quality of input and testing data. For instance, the pre-production stage is usually the first stage in which a software system can be tested with realistic data, but at the cost of longer testing runs and higher hardware provisioning cost. Also, at this stage, performance tests provide closer aligned results with the production deployment. Hence, the environment as well as the stage have a crucial impact not only on the configuration values, but also on the mechanisms and tools of creating, deploying, and maintaining configurations. This needs more attention from researchers.

3.6 Configuration Life Cycle

Life Cycle values

$\text{Create}_{S0,1,2}$, $\text{Maintain}_{S1,2}$, $\text{Bind}_{S0,1,2}$, Own_{S0}, Deprecate_{S2}, Remove_{S0}

This dimension describes the diverse aspects of configuration options from creation and maintenance over binding to deprecation, that is, all lifetime phases. Although some research has identified diverse problems when introducing more and more options [44], it is often not clear that a configuration option has its own life cycle, which starts in the requirements phase when, for example, a new optional feature is planned. Contrary to the literature, we found in our interviews that configuration options are usually introduced in an ad-hoc manner, which decreases maintainability and increases configuration complexity.

The life cycle continues with the value binding. That is, at a certain point in time, an option is bound to a certain value. At this dimension, the binding time is not so interesting, but more importantly is the question *who owns the value* and *for how long is the configuration valid*. Owning means that there is a designated stakeholder who is able and responsible for reconfiguring (rebinding) an option's value or maintaining the option's code or functional representation. So, the stakeholder who initially sets a configuration value might not be responsible in the future to change or test it: *"The developers say 'I can customize my application server with a certain VM size.'. Which VM size is later actually applied, we don't know. So, we say 'Okay, we have tested it with this minimum and that maximum. For what lays in between, we can give guidance where we can say for that many products in a shop, you would need the value a bit higher [...]. But, the final configuration is done by the Ops guys who say 'I know that this is this specific customer, we have measured [it], and these are the performance metrics, so we change the configuration.' And thus they own the values and are responsible for them. [...] This stakeholder takes responsibility for the toggling and the others must live with it."*$_{I8}$

The validity of a configuration with respect to time—and not constraints between other options or the environment—has only recently been in the focus of research [41]. However, with the trend of on-premise solutions, serverless, and functions and software as a service, such timely restricted configurations, be it of technical or domain-specific nature, will become more relevant and require novel solutions in configuration maintenance and evolution. The developers need to know about such aspects to be able to prepare the system for invalid states that can occur at run time without an explicit trigger of a configuration process.

Consequences: We need to model and provide support for the life cycle of individual configuration options—an aspect neglected so far in research. With an explicit modeling, we can better track why options are introduced (see next dimension), who binds the option's value and who owns the option's value. Knowing the stakeholders of these two separate phases (binding and owning) allows for better tool support and validation of the configuration values (e.g., developers provide reasonable default values and operations people adapt them to end user requirements) and enables short communication paths due to clear responsibilities. Furthermore, an explicit *option life cycle* allows for thinking about and planning the deprecation of configuration options, a still unsolved problem in practice [44]. Thus, we should focus on methods that support the removal of variability, and such addressing variability with life cycle information can be a means for it. Interesting, the described software configuration engineering process in [33] considers the whole development process whereas we found that we need to additionally consider the individual options' own life cycle.

3.7 Intent for Configuration

Intent values

A/B Testing_{S0}, $\text{Code Reuse}_{S0,1}$, $\text{Knowledge Preservation}_{S0}$, $\text{Reduced Testing Effort}_{S0}$, $\text{Distributed Environment}_{S0,1,2}$, $\text{Unknown Environment}_{S0,1,2}$, $\text{Non-Functional}_{S0,1,2}$, $\text{Functional}_{S0,1,2}$

This dimension describes why configuration options are introduced in the first place, which can have a strong effect on the other dimensions. We found that often, options are introduced to tailor functional behavior (*"When I write [code], I as a developer want to make it configurable. With a feature toggle, I want to be able to switch between code path left and right, such that I can pull the handbrake or unlock specific things for just one customer."*$_{I9}$), to tailor non-functional behavior, and to reuse code.

However, as the interviewees pointed out, there are also other intents that trigger the implementation of configuration options: knowledge preservation (*"I don't know what I've done two weeks ago, and this is why I make something configurable, something that I need to somehow adjust anyway."*$_{I1}$), *"the analysis of a program at run time"*$_{I11}$ (e.g., via logging levels), A/B testing, and distributed development of features (*"Due to the team's autonomy, it is desired that when you have implemented a feature, you can take it to production, but it is deactivated with a toggle. But it is already present in production code. If all teams have finished their [dependent] implementations, then each needs only to turn on the switch to activate the feature."*$_{I3}$). Interestingly, options are also introduced to enable comparability of testing results in an evolving software system. The rationale is that changes to a code base need to be controlled during testing: *"If you not automate [configuration] such that you have the same preconditions to execute a test, then you already have lost, because the results you will get are not comparable."*$_{I4}$

Another reason for introducing configuration is the need to run an application in an unknown environment: *"We just don't know how the data centers are structured [in which we deploy our software]. There could be a firewall between the App server and the database for one customer and for all others not. For one customer, we tried to configure the database so that the same load distributes on three hard disks. And one week later, we have a customer with three different data centers with high availability and no downtime. [...]"*$_{I8}$

Configuration options are also introduced to reduce testing effort. At first, this seems contradictory to the observations of combinatorial testing and the exponential growth of configurations. But when we know that options have only a limited scope, we may need to test only a few new configurations and not all combinations. The alternative to new options would be, for example, having different branches which all would need to be tested. This scales far worse, as one interviewee explained: *"In every JDK [version], they improve the garbage collector. With those improvements, the memory management changes, and we just don't know which consequences this has. That is, every change is a different behavior and this changes the performance. Now, when we don't know the right values yet, we make this configurable. So, we tell our test engineer 'Can you test all configurations?' instead of 'Here are ten branches [to test].'"*$_{I8}$

Consequences: The interviewees described many new intentions for configuration, showing that the reasons for integrating new configuration options are more diverse than tailoring functional or non-functional properties of an application. Knowing these intentions can help in developing tools to either control for these options, to avoid introducing them altogether, or to remove them again from the code base (given their appropriate documentation).

3.8 Configuration Complexity

Complexity values

Local scope$_{S1}$, Distributed scope$_{S1}$, High dependency$_{S1}$, Low dependency$_{S1}$, No dependency$_{S1}$

Although there are many aspects contributing to the configuration's complexity, there are certain values that need special attention. So far, we have not explicitly addressed the scope of a configuration with respect to software modules and artifacts. The interviewees of [6] reported on configuration that affects only individual modules (i.e., with a local scope) and configuration that is distributed among different modules, frameworks, services, and other artifacts (i.e., global scope). Moreover, an option or entire configuration might be dependent on other options and configurations at different degrees. For example, the presence and valid values of an option might depend on the configuration of its cloud infrastructure or which customer owns the product. In such cases, the complexity increases and interacts with other dimensions, such as the configuration artifacts, stage, and type. So, choosing the right artifact for a specific type of configuration depends on the complexity of the configuration.

Consequences: The literature and most configuration tools concentrate on local configuration. However, when having multiple users at different levels of the software (e.g., platform developer versus plug-in developer versus end user), a configuration might affect very different modules, developed, maintained, and deployed by different stakeholders, vendors, and customers. Research on multi software product lines has analyzed some isolated aspects of dependent and distributed configuration [28, 42], but have not taken into account most aspects of configuration, such as non-functional optimization. But, it is important to fully understand how far-reaching a single configuration is to estimate the impact of configuration choices on the whole software ecosystem.

3.9 Interactions

Interactions can be expressed through connections and paths in the model. Different paths connected with the same nodes (i.e., values of dimensions) represent different interactions. For example, contrary to many product line studies, we found that modeling only the domain variability is insufficient to derive a valid program variant. Our interviewees argued that often, technical configuration determines domain-specific configuration and vice versa: *"Infrastructure must partially provide configuration via environmental variables to the application. This is a clear interaction point. Then there is always the question of how do these environmental variables come to the infrastructure.[...] And then there is interaction among different infrastructure components and eventually the application."*$_{I6}$ Hence, the graph not just shows the presence of configuration-dependent dimensions and values, but how they interact, incorporated in the order of the path.

The importance of interactions for identifying possible misconceptions in current research can be seen be the following example: *"There are different teams that handle [configuration]. So, I can't say 'I'll introduce a property and now I want to know from all configurations where my software resides where was it configured from whom, how, and why."*$_{I8}$. So, we need to relax the current assumption in research of one variability model with a dedicated domain engineer who configures the system. Instead, when modeling a graph for a future study on variability models, it should contain several paths from stakeholders to, for instance, intent, configuration artifact, and compile time to make the different interactions explicit, and design a study based on this structure. In essence, our model can provide an important tool for study and research design in the area of software configuration to decide upfront on which aspects researchers concentrate on and which aspects might be neglected by the current design.

4 APPLICATION OF THE MODEL

To show our model in action, we provide a model of configuration for 18 research papers selected from three related research fields: configuration engineering, software product lines, performance optimization, testing, and configuration errors. We discuss two papers used in the meta study in detail, give a high-level discussion about two research areas, and provide the models for all papers on our supplementary web page. Moreover, as an important contribution to practice, we extract key best practices from our interviews.

4.1 Application to Research

To demonstrate how useful our model is when applied to specific studies, we provide the context for the papers of our meta study. Moreover, we discuss on a higher level some insights we gained when analyzing related literature in the area of software product lines and performance optimization.

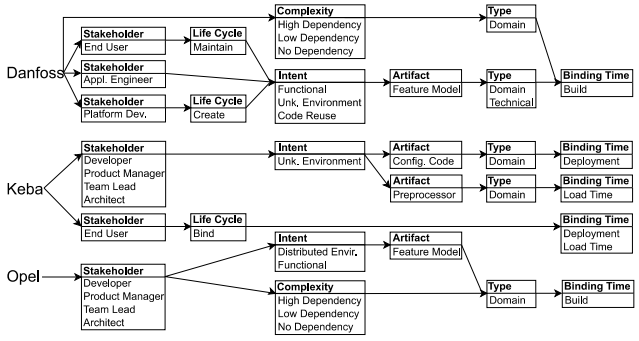


Figure 2: Configuration context of the qualitative study by Berger and others [6]. Boxes depict dimensions of configuration and their concrete values. Arrows depict analyzed and discovered interactions among the dimensions. As can be seen, the configuration context differs between companies.

Meta Study: Configuration Engineering. Figure 2 depicts the configuration context of the study by Berger and others [6]. In their study, the authors interviewed three different companies which resulted in different contexts. For example, Keba interviewees report about end users who bind configuration options at load and deployment time without going into detail about the type of configuration or the intent of the configuration. Others talk about different developers roles and how they implement configuration options to address unknown environments and how different artifacts lead to different binding times. However, they talk only about domain configuration and not technical configuration (e.g., about how to configure the infrastructure).

The important insight is, however, that the answers of different companies lead to entirely different contexts, which is not apparent in the paper and can lead to a more focused and balanced discussion about the findings. For instance, Opel interviewees are more interested in configuration options for functional tailoring or adaptation to a distributed environment and, unlike Keba, they use feature models as configuration artifacts and consider only build-time configuration. So, by just comparing these two contexts we can conclude that answers may not align due to different configuration intents, different implementation artifacts, and binding times. Moreover, we can further frame the study and the reported results in addressing mostly domain configuration without stages, with limited considered artifacts and binding times. So, the answers to the authors' research question might severely change when considering technical configuration, run time configuration, or options introduced for non-functional optimization or testing.

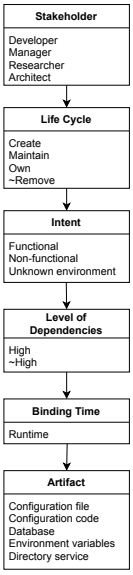
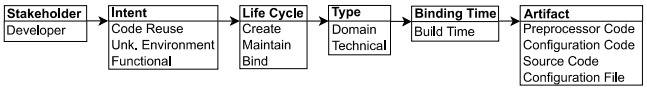


Figure 3: Configuration context of Sayagh and others [33].

The study by Sayagh and others on configuration in practice represents (shown in Figure 3) gives a more unifying context [33], possibly because the authors provided a summative report and did not differentiate among companies. Here, we see that no end users are in the context although also non-functional configuration has been studied, which is contrary to what we found in the performance-optimization papers, which mostly focus on end user configuration. Moreover, the binding time is limited to only run time, which makes a stark contrast to Berger and others.

Interestingly, they considered multiple environments, a large portion of an option's life cycle, and many configuration artifacts, which makes this study the most comprehensive ones regarding aspects of configuration. By making the context with our model clear, it is now easy to see where the study should be extended in the future and how this compares to previous work as Berger and others. For the first time, we can provide a quick overview about addressed dimensions of configuration and see (with missing configurations) where a study give more information or lacks context.

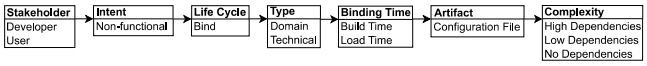
Software Product Lines (SPLs). We analyzed several papers in the area of software product line (SPL) research. The results are comprised in the figure below—a per paper context can be found at our complementary web page. The model shows that product line research mainly focuses on developers, which is natural as mostly implementation techniques are discussed. However, the new trend of DevOps seems to not have reached yet research in this area. Also not surprisingly is that most papers focus around code reuse and functional customization of software systems. What is missing in this context is (i) evolution of product lines, especially how to deprecate and remove configuration options (*life cycle*), (ii) different *binding times* and their effect on implementation techniques, (iii) modern staged deployments and cloud infrastructures for product line development (*stage* and *type*), and (iv) *complexity* analysis with respect to product line implementation techniques.



In relation to our interview, one aspect became apparent: Modern software development is hardly separable between domain-specific and infrastructure aspects: *"Certainly, when you activate a certain function, it often requires a specific component service and so that a system can use a service, it needs to be configured in that way. For example, email verification [as domain-specific functionality] and for this exists a service for email verification. And when you activate this function, you need to provide a configuration such that the service can be used. So, often in this direction: If you activate a functionality, you need a corresponding technical configuration to make it work."*[I7] Hence, future studies should make sure to control for other types of configuration, collect the intent of configuration, make the environment and adaptation of the corresponding configurations clear, and include more stakeholders with their role in an option's life cycle. Overall, we often see a small fraction of the dimensions and values covered by each study. This is not problematic per se if the paper makes this context clear and when we can build a bigger picture by combining multiple studies. Unfortunately, the coverage across papers is low so that the same context often evaluated multiple times. This is a clear outcome when applying our model. Covering the same dimensions multiple times can be a means to replicate former

findings, but we could not find only limited attempt to reevaluate or compare against prior work. Instead, we see that many dimensions and values, although practically relevant, are left untouched, which hinders transfer to a practical context and the derivation of generalizable insights.

Performance Optimization. Several papers focus on finding high-performance software configurations [21, 25]. They all take the same context: A user or the developer configures a system via configuration files at load time or build time. The environment is never made explicit. So it is entirely unclear at which *stage* performance optimizations are made and, more importantly, the infrastructure (*type*) is not taken into account although it has a profound influence on performance. Moreover, reconfiguration or software evolution does not play a role, so the life cycle of options addresses only binding options to their values.



A practical context would require extending the types of configuration to infrastructure configuration and also considering additional configuration artifacts. As performance is mainly driven by the underlying hardware, and not the user, but the DevOps and Ops persons are responsible for maintaining and owning performance-relevant configurations, research on performance optimization should broaden the scope toward these dimensions.

4.2 Best Practices

Based on our interviews and the developed model on configuration, we articulate challenges and point to best practices.

Configuration Errors. Configuration errors are an important issue in practice: *"They are harder to find than normal bugs. It also depends on the format of the configuration file."*$_{I11}$ So, there seems to be a relation from artifacts to errors. We found three main practices for avoiding configuration errors: explicit modeling of options, favor value binding at build time, and simplify configuration.

It seems that an explicit modeling of options, including documentation about the intent of an option can help in discussing whether an option is needed or not (anymore). This calls for a model, which, in fact, one company has introduced in form of a meta-configuration: *"Validation is part of the meta-configuration. [...] So, there is a kind of management model, that is, a configuration option is not only the name of the option, but also the data type, a validation, or where does this configuration value come from, so cross relations to other configuration values."*$_{I11}$

To better handle configuration errors, practitioners aim to move run-time configuration to build-time configuration. This enables multiple things: *"Real program code can have [errors], but there are more possibilities to verify things, also already at build time. Maybe that is a possibility to address this problem, that you may try to validate or check the configuration as part of building the software. [...] Everything that can be done as early as possible in the process, validation checking, helps to solve these problems."*$_{I11}$ Validation at build time has also the benefit that errors can be spotted during deployment and development and not at load time.

Finally, simplifying the configuration can be done in two ways. First, the configuration process can be made less error prone by guiding users through the process: *"I don't think that my users [of the configuration editor] know what makes sense to configure and what not. So, it is my task to find out how to present the configuration options to the users in a way that it is as easy as possible to configure things."*$_{I11}$ Another approach is the recent practice of *Convention over Configuration*, where a stakeholder (e.g., a developer) who is usually not responsible for binding options' values provides a feasible pre-configuration that is hidden from the user, avoiding the need to dig deep into configuration files. The stakeholder (e.g., ops engineer) who then owns the configuration can diverge from the standard values to adapt the system to their needs.

Distributed Configuration. Configuration is nowadays distributed over multiple artifacts, types, stakeholders, and environments: *"[Configuration] was somehow under control in a monolithic system, because one could see everything what is in there. But now, one has not even the chance for this, so one looks more in a spot-light manner on individual components. [...] So, when I don't know a thing about where the sources are, where the tests are, and there is no default structure, how should I find out something? And this structure is mega important for configuration."*$_{I1}$ A recurring pattern mentioned in our interviews are dedicated platform teams for specific types of configuration: *"A sister team of ours mainly does this [Kubernetes configuration]. They provide the basis for us and provide the low level things."*$_{I9}$ This delegation of configuration on expert knowledge and clear separation of responsibilities is a success factor when it comes to different types of configuration. We found further best practices similar to [33]: naming conventions of options, placing configuration files next to the code in a repository to enable versioning of configuration, reviewing configuration files and configuration-value changes in code reviews and pull requests.

Performance Engineering. All interviewees agreed that performance is important and that either they or another team handles performance issues: *"There is a team that does nothing else than performance testing. They have different setups at hand [...] and every new function becomes eventually part of a performance test."*$_{I11}$ One issue is that performance is difficult to measure or monitor and it is even unclear what to include into performance metrics: *"Performance is really difficult. What means performance? Response time of requests? From which requests? Resource consumption counts as well."*$_{I8}$ Unfortunately, we have not seen many solutions to performance optimization despite its relevance. There are dedicated performance engineers running stress tests and other related measurements, but with the main goal of finding good default values rather than optimized settings for individual customers. These observations back up the need for research on performance optimization, for example, by using machine learning or related techniques to discover underlying patterns. Current best practice is a proper monitoring infrastructure and logging of configuration changes to make spotting performance issues and their causes easier.

4.3 Threats to Validity

We transcribed the interviews, performed the card-sorting process on the German text and translated the extracted citations to English.

Both steps could introduce uncertainties and even change semantics. We controlled for this threat by having at least two authors look at each interview and making notes. We compared notes with the double-checked translation. Similarly, the entire author team was present during the card sorting sessions to minimize subjectivity.

Threats to external validity are certainly related to the number and selection of interviewees. To mitigate threats caused by our sample, we selected interviewees of different roles from different companies. We are aware that the dimensions of configuration and their interactions are certainly not exhaustive, as this is hardly possible with a single study. Nevertheless, we have covered a substantial area of configuration, and future work with different practitioners can certainly enrich the current version of our model.

5 RELATED WORK

In addition to the two papers of our meta studies, most research focuses on individual aspects of configuration, such as configuration for the DevOps role. For instance, configuration management for code and infrastructure has been found by multiple studies to be one essential technological enabler of DevOps [35, 38]. Other aspects of configuration seem to relate to the environment, as in CI/CD. Hilton and others studied the configuration effort of CI [9]. In particular, they found that some projects frequently change their configuration files over the project history. Our interviewees brought up similar aspects of configuration, but described them as more intertwined with different stakeholders and implementation artifacts.

Variability or feature models are central for defining the configuration space [5, 17]. However, most of the techniques concentrate on individual software systems and ignore other dimensions, such as infrastructure variability. Some approaches developed in the area of multi-software product lines [28] provide means to model different layers of configuration and different dimensions, such as Velvet [29], Clafer [15], and the common variability language (CVL). Still they focus on narrow aspects of configuration and our model has already identified ways for extending the scope such as explicitly modeling an options intent and life cycle.

Berger and others analyze features with respect to their types, numbers, and constraints in 128 variability models [7]. Lee and others report on modeling an elevator control software containing 490 features with 22 different operating environments [19]. Finally, a recent study by Nešić and others collected 34 principles for modeling variability in practice [22]. Again, although they do not aim for capturing dimensions of configuration, they agree on subsets, such as the life cycle of configuration, the existence of different stakeholders, and the different intents for configuration. Our work subsumes many aspects brought up by the literature by using expert interviews, so we link research to practice. Moreover, we lay out interactions among these aspects and highlight corresponding challenges and best practices.

6 CONCLUSION AND LESSONS LEARNED

"[Configuration] becomes ever more important, because we always thrive for writing less code and better reaching our goals via configuration."[I8] In this vein, we built a paper on configuration to provide context for research and practitioners about what aspects resemble the context of configuration. This way, research becomes better to translate to practice, better to compare with, a better to identify current gaps. We argue that our model derived from three different sources of qualitative interviews provides a strong foundation for further research on this matter. Again, the vast number of work related to configuration needs to be consolidated and we provide a means to it. In addition, there are further lessons:

Configuration is code, but often without a well-defined grammar and type system. This has serious implications on configuration validation, the learning rate of how to configure a system, and identifying and fixing configuration errors. Basically, we are coding blindly and hope for the best. Common feature models do not provide the required expressiveness. Also, they provide no support for the life cycle of options or mechanisms for staging and distributing configurations in different environments. Current solutions often propose distinct tools, such as FeatureIDE [40] for modeling and validation, but it seems that a lightweight solution is needed that smoothly integrates in the CI/CD process. A promising study that builds on similar insights of [33] has shown that codifying configuration with type information can improve the configuration process [34].

Configuration is a highly cross-cutting activity affecting nearly all stakeholders of a software system and manifesting in diverse artifacts across different environments. The tremendous diversity of configuration combined with the high degree of interaction makes the whole configuration aspect an increasing challenge. Specialized teams, strict conventions, a clear concept about an option's life cycle, and an explicit modeling of configuration throughout all environments seem to be key mechanisms to counter the complexity.

Configuration in research and practice diverge with respect to considered and limited contexts of studies. Balancing internal and external validity may play an important role in studies about configuration (including ours) [36]. Our interviews have shown that for some research areas, the scope should be substantially widened to create actionable results (e.g., performance optimization). However, this might come at the cost of limited explainability and verifiability. We believe that by making the context of research explicit (with our model), it allows for a series of experiments in varying contexts where internal validity does not need to be traded for ecological or external validity.

ACKNOWLEDGMENTS

Norbert Siegmund's work has been funded by the German Research Foundation (SI 2171/3-1, SI 2171/2) and the by the German Federal Ministry of Education and Research under grant BMBF 01IS19059B. Janet Siegmund's work has been funded by the German Research Foundation (SI 2045/2-2).

REFERENCES

[1] 2001. *Software Product Lines: Practices and Patterns.* Addison-Wesley Longman Publishing Co., Inc.
[2] Gojko Adzic and Robert Chatley. 2017. Serverless Computing: Economic and Architectural Impact. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE).* ACM, 884–889.
[3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer.
[4] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. 2016. Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software* 33, 3 (2016), 42–52.

[5] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the 9th International Conference on Software Product Lines (SPLC)*. Springer-Verlag, 7–20.

[6] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. 2015. What is a Feature?: A Qualitative Study of Features in Industrial Software Product Lines. In *Proceedings of the International Conference on Software Product Lines (SPLC)*. ACM, 16–25.

[7] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Trans. Softw. Eng.* 39, 12 (2013), 1611–1640.

[8] Rajdeep Dua, A Reddy Raja, and Dharmesh Kakadia. 2014. Virtualization vs Containerization to Support PaaS. In *Proceedings of the International Conference on Cloud Engineering (ICCE)*. 610–614.

[9] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-source Projects. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. ACM, 426–437.

[10] William Hudson. 2013. Card Sorting. In *Guide to Advanced Empirical Software Engineering*. The Interaction Design Foundation.

[11] Jez Humble and David Farley. 2010. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation.* Addison-Wesley Professional.

[12] Ramtin Jabbari, Nauman bin Ali, Kai Petersen, and Binish Tanveer. 2016. What is DevOps? A Systematic Mapping Study on Definitions and Practices. Association for Computing Machinery, 1–11.

[13] Pooyan Jamshidi, Norbert Siegmund, Miguel Velez, Christian Kästner, Akshay Patel, and Yuvraj Agarwal. 2017. Transfer learning for performance modeling of configurable systems: An exploratory analysis. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 497–508.

[14] Pooyan Jamshidi, Miguel Velez, Christian Kästner, and Norbert Siegmund. 2018. Learning to Sample: Exploiting Similarities Across Environments to Learn Performance Models for Configurable Systems. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 71–82.

[15] Paulius Juodisius, Atrisha Sarkar, Raghava Rao Mukkamala, Michał Antkiewicz, Krzysztof Czarnecki, and Andrzej Wąsowski. 2018. Clafer: Lightweight Modeling of Structure and Behaviour. *The Art, Science, and Engineering of Programming Journal* 3 (2018).

[16] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. 2019. Distance-based Sampling of Software Configuration Spaces. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Press, 1084–1094.

[17] Kyo Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report.

[18] D. Richard Kuhn, Raghu N. Kacker, and Yu Lei. 2013. *Introduction to Combinatorial Testing* (first ed.). Chapman & Hall/CRC.

[19] Kwanwoo Lee, Kyo C. Kang, Eunman Koh, Wonsuk Chae, Bokyoung Kim, and Byoung Wook Choi. 2000. *Domain-Oriented Engineering of Elevator Control Software.* Springer US, 3–22.

[20] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. 2016. On Essential Configuration Complexity: Measuring Interactions in Highly-configurable Systems. In *ProceedingsInternational Conference on Automated Software Engineering (ASE)*. ACM, 483–494.

[21] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. 2017. Faster discovery of faster system configurations with spectral learning. *Autom Softw Eng* (2017), 1–31.

[22] Damir Nešić, Jacob Krüger, Stefan Stănciulescu, and Thorsten Berger. 2019. Principles of Feature Modeling *(ESEC/FSE)*. Association for Computing Machinery, 62–73.

[23] Sam Newman. 2015. *Building Microservices.* O'Reilly Media.

[24] Changhai Nie and Hareton Leung. 2011. A Survey of Combinatorial Testing. *ACM Comput. Surv.* (2011), 1–29.

[25] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding Near-optimal Configurations in Product Lines by Random Sampling *(ESEC/FSE)*. ACM, 61–71.

[26] Daniela Rabiser, Herbert Prähofer, Paul Grünbacher, Michael Petruzelka, Klaus Eder, Florian Angerer, Mario Kromoser, and Andreas Grimmer. 2018. Multi-purpose, Multi-level Feature Modeling of Large-scale Industrial Software Systems. *Software & Systems Modeling* 17, 3 (2018), 913–938.

[27] Ariel Rabkin and Randy Katz. 2011. Static Extraction of Program Configuration Options *(ICSE)*. Association for Computing Machinery, 131–140.

[28] Marko Rosenmüller and Norbert Siegmund. 2010. Automating the Configuration of Multi Software Product Lines. University of Duisburg-Essen, 123–130.

[29] Marko Rosenmüller, Norbert Siegmund, Thomas Thüm, and Gunter Saake. 2011. Multi-Dimensional Variability Modeling. In *Proc.Int. Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*. ACM, 11–20.

[30] Gordon Rugg and Peter McGeorge. 2005. The Sorting Techniques: A Tutorial Paper on Card Sorts, Picture Sorts and Item Sorts. *Expert Systems* (2005), 94–107.

[31] Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. 2015. Cost-Efficient Sampling for Performance Prediction of Configurable Systems. IEEE Press, 342–352.

[32] Mohammed Sayagh, Noureddine Kerzazi, and Bram Adams. 2017. On Cross-stack Configuration Errors. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Press, 255–265.

[33] Mohammed Sayagh, Noureddine Kerzazi, Bram Adams, and Fabio Petrillo. 2018. Software Configuration Engineering in Practice: Interviews, Survey, and Systematic Literature Review. *IEEE Transactions on Software Engineering* (2018).

[34] Mohammed Sayagh, Noureddine Kerzazi, Fabio Petrillo, Khalil Bennani, and Bram Adams. 2020. What Should Your Run-time Configuration Framework Do To Help Developers? *Empirical Software Engineering (EMSE)* 25 (2020), 1259–1293.

[35] Mali Senapathi, Jim Buchan, and Hady Osman. 2018. DevOps Capabilities, Practices, and Challenges: Insights from a Case Study. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE)*. ACM, 57–67.

[36] Janet Siegmund, Norbert Siegmund, and Sven Apel. 2015. Views on Internal and External Validity in Empirical Software Engineering. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Press, 9–19.

[37] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. 2012. Predicting Performance via Automated Feature-Interaction Detection. IEEE Press, 167–177.

[38] Jens Smeds, Kristian Nybom, and Ivan Porres. 2015. DevOps: A Definition and Perceived Adoption Impediments. In *Agile Processes in Software Engineering and Extreme Programming*. Springer International Publishing, 166–177.

[39] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. 2016. Grounded Theory in Software Engineering Research: A Critical Review and Guidelines. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 120–131.

[40] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming* 79 (2014), 70–85.

[41] Thomas Thüm, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer. 2019. Towards Efficient Analysis of Variation in Time and Space. Association for Computing Machinery, 57–64.

[42] Guadalupe Trujillo Tzanahua, Ulises Juárez Martínez, Alberto Aguilar-Lasserre, and Karen Verdín. 2018. Multiple Software Product Lines: Applications and Challenges. In *Proceedings of the International Conference on Software Process Improvement (CIMPS)*. 117–126.

[43] James Turnbull. 2014. *The Docker Book: Containerization is the new virtualization.*

[44] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 2015. Hey, You Have Given Me Too Many Knobs!: Understanding and Dealing with Over-designed Configuration in System Software. ACM, 307–319.