Exploring Differences and Commonalities between Feature Flags and Configuration Options

Jens Meinicke, Chu-Pan Wong, Bogdan Vasilescu, Christian Kästner
Carnegie Mellon University

ABSTRACT

Feature flags for continuous deployment and configuration options for customizing software share many similarities, both conceptually and technically. However, neither academic nor practitioner publications seem to clearly compare these two concepts. We argue that a distinction is valuable, as applications, goals, and challenges differ fundamentally between feature flags and configuration options. In this work, we explore the differences and commonalities of both concepts to help understand practices and challenges, and to help transfer existing solutions (e.g., for testing). To better understand feature flags and how they relate to configuration options, we performed nine semi-structured interviews with feature-flag experts. We discovered several distinguishing characteristics but also opportunities for knowledge and technology transfer across both communities. Overall, we think that both communities can learn from each other.

ACM Reference format:
Jens Meinicke, Chu-Pan Wong, Bogdan Vasilescu, Christian Kästner. 2020. Exploring Differences and Commonalities between Feature Flags and Configuration Options. In *Proceedings of Software Engineering in Practice, Seoul, Republic of Korea, May 23–29, 2020 (ICSE-SEIP '20),* 10 pages.
https://doi.org/10.1145/3377813.3381366

1 INTRODUCTION

Feature flags and configuration options are broadly used in practice and share technical similarities and challenges, but they also have distinguishing characteristics that are rarely made explicit but that are important for practices around management, removal, documentation, and testing. We explore these differences and what lessons can be learned across both communities.

Feature flags have received a lot of attention recently: Practitioners discuss them and best practices around them widely in blog posts and at conferences, and multiple startups compete at providing tool support. At a technical level, feature flags are a design pattern to conditionally enable a code path (e.g., an if-statement controlled by a Boolean flag), where the decision is typically controlled by an external configuration mechanism. In practice, feature flags are used to enable collaborative development in the same branch (continuous integration) [20, 34], as well as experimentation in production (continuous experimentation) [5, 50] and canary releases (continuous deployment) [44, 49]. For example, as illustrated in Listing 1, feature flags may be used to run an A/B test to determine whether changing a button in a web application changes outcomes such as click rate.

Listing 1: Example of variability using feature flags [13].

```
renderCheckoutButton() {
   if (features.for({user:currentUser}).
       isEnabled("showReallyBigCheckoutButton")) {
      return renderReallyBigCheckoutButton();
   } else {
      return renderDefaultCheckoutButton();
   }
}
```

At first glance, feature flags have a lot of similarities with *configuration options*, for example, command-line parameters, configuration files, or even `#ifdefs` in the source code, which also control the behavior of a program and often make decisions between different code paths. With configuration options, users typically decide which functionality to include (e.g., whether to perform logging or whether to enable security features) or make tradeoffs between competing qualities by setting parameters like buffer size or compression levels. The implementations of feature flags and configuration options often look similar and some discussed problems are similar (e.g., testing, complexity).

However, unlike feature flags which have become popular recently but have seen only limited attention from researchers, there are several communities that have studied configuration options for decades, including the community on software product lines [3, 11, 35] and the systems community [58]; some even argue that configuration engineering should be its own discipline [43]. As a result, there is a vast amount of accumulated knowledge about configuration options, on topics such as design and management [11, 35], implementation strategies [1, 3], feature interactions [10, 21], and quality assurance strategies [33, 52]. How do the two worlds relate? And what could each learn from the other? For example, regarding configuration options, we have ample evidence from the literature that the number of options seems to only ever grow [25, 56], often creating significant challenges for users trying to configure the systems [19, 57] and surprising interactions at runtime [10, 21]; but we also know how to diagnose, fix, or even prevent configuration mistakes [16, 48, 59]. For feature flags, we now see similar discussions again [14, 17, 26, 38]: too many flags, hard to remove, hard to test, surprising interaction faults, and many more. Is there an opportunity to transfer some of the lessons learned from the configuration-options community?

To better understand practices and challenges around feature flags and how they relate to configuration options, we performed nine semi-structured interviews with experts with direct experience using feature flags or developing infrastructure for feature flags, and

ICSE-SEIP '20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7123-0/20/05.
https://doi.org/10.1145/3377813.3381366

we contrast our findings to the literature on configurable systems and software product lines. Through our interviews, we identify many similarities and many opportunities for technology transfer across the communities, but also several distinct challenges. For example, testing challenges across large configuration spaces are often similar and there are plenty of opportunities to adopt tools like combinatorial interaction testing also for feature flags, whereas the common temporary nature of feature flags raises new process and tooling challenges but also novel opportunities for their removal, as we will discuss. *Our overall goal is to educate researchers and engineers from both communities, feature flags and configurable systems, about the differences between either concepts, and to point out opportunities for technology transfer and novel research and tooling.*

In summary, the contributions of this paper are:

- We report on the results of nine semi-structured interviews with experts from the feature flags community.
- We identify differences and commonalities between feature flags and configuration options.
- We discuss existing solutions and best practices from configurable systems that may help with feature flags.

2 RELATED WORK ON FEATURE FLAGS

While feature flags have rarely been studied in academic research, they are heavily discussed by practitioners and used in industrial settings for purposes like trunk-based development, A/B testing, and canary releases. Most of the literature around feature flags comes directly from industry, in the form of talks and blog posts (e.g., [14, 17]), typically in the context of discussions on continuous delivery, continuous deployment, and continuous experimentation. Rahman et al. [37, 38] and Mahdavi-Hezaveh et al. [26] performed recent "grey literature" surveys of such blog posts and practitioner talks on feature flags, providing an overview of common topics and challenges, such as problems with cleaning up old flags.

On the academic side, Rahman et al. [37, 38] analyzed feature flag usage in the open-source code base behind Google Chrome, and found that feature flags are heavily used to control which features to deploy but that they are often long lived, resulting in additional maintenance and technical debt. Mahdavi-Hezaveh et al. [26] surveyed developers from 38 companies on their practices regarding feature flags, finding that most companies do not clean up feature flags systematically. Tang et al. [50] and Bakshy et al. [5] discuss how to implement A/B testing infrastructure at scale, though they focus mostly on infrastructure design to support experiments.

We are not aware of any discussion contrasting feature flags and configuration options in either academic or practitioner literature. As far as we are aware, feature flags are not mentioned in publications of configurable systems or software product lines, and configuration options are not raised as a contrast in discussions around feature flags and continuous deployment. Rahman et al. [37] and Fowler [14] mention that there are different categories of feature flags, namely *business and release toggles* and *release, experiment, ops, and permission toggles*, where business toggles and op toggles roughly relate to configuration options, but neither relates their discussion to the literature on configurable systems.

In summary, there is ample discussion online about practices around using feature flags and a rich academic research literature on configuration options. However, it appears that both communities are unaware of each other, while there seems to be potential to learn from each other.

3 METHODOLOGY

Our goal is to identify fundamental similarities and differences between configuration options and feature flags. As discussed above, these concepts are rarely distinguished explicitly in existing literature. We rely on the existing literature on configuration options and our own background with years of research in this area, but choose to interview practitioners having extensive experience with feature flags to understand feature flags and how developers think about them.

Participants. To recruit experienced participants for our interviews, we (a) used our professional network to ask for references to developers working with feature flags or on feature-flag infrastructure, (b) contacted authors of blog posts on feature flags and engineers of open-source projects that use feature flags, and (c) contacted developers suggested by previous interviewees. We conducted 9 interviews, all with professionals deeply familiar with the topic, with multiple years of experience working with feature flags. Six interviewees (I1-6) work in three large software companies, one (I7) is a consultant on DevOps topics, one (I8) works for a feature-flag tooling service, and one (I9) works with a nonprofit software organization. Four interviewees work in different parts of the same organization, but on different teams with very different perspectives on feature flags. I3, I6, and I8 are developers of feature-flag infrastructure; I1–I5 and I9 actively use feature flags in their work.

Semi-Structured Interviews. We developed an interview guide based on our analysis of the existing literature on feature flags (academic literature, blogs, talks, see Sec. 2) and hypothesized differences. The interview guide covers commonly mentioned problems (e.g., testing and removal of feature flags) and challenges that typically associated with configuration options (e.g., documenting dependencies and feature interactions). This resulted in a list of topics that we covered in every interview:

- Goals of using feature flags
- Development process:
  - Creating new feature flags
  - Changing the state of a feature flag
  - Removing feature flags (cleanup) and consequences regarding (a) technical debt and (b) code complexity
- Documentation:
  - Feature traceability (flag-to-code relation)
  - Dependencies among feature flags
- Analysis:
  - Testing
  - Interactions among feature flags
- Open challenges when working with feature flags

We conducted semi-structured phone interviews which lasted between 30 and 60 minutes. We structured the interviews to cover all the topics of our interview guide, but the order in which we asked questions was guided by the flow of each individual interview and the experiences of the interviewee. During the interviews, we brought up relevant practices and problems with configuration options (e.g., challenges with feature interactions) and asked whether the interviewees saw similar challenges or had developed solutions. That is, we covered both feature flags and configuration options from the perspective of interviewees who are experts in feature flags. We recorded all interviews with the interviewees' consent and subsequently transcribed and analyzed them.

Saturation and Data Analysis. The authors discussed results after each interview but saw little need to adjust the interview guide for subsequent interviews. At the end of each interview we asked whether we missed any topics regarding their experience with feature flags, but we did not discover any missing topics. In fact, we reached *saturation* after about 5 or 6 interviews; while we heard more stories in later interviews, they only confirmed the differences discussed in earlier ones and added no new insights. That is, we believe that our interviews are sufficient to give us a generalizable overview on the challenges and differences that we are looking for.

After completing all interviews, we analyzed the transcripts using standard coding techniques [41]. We started with a preliminary coding frame derived from the interview guide and added extra (sub)codes as we identified themes in the interviews. The first author coded all transcripts and the second author refined the coding to mediate false and incomplete characterizations. All authors discussed the codes. Based on the coding, we derived an overview of practices that use feature flags. This overview enables us to identify new research directions and challenges, as well as existing solutions and practices, including practical guidelines for feature flags, resulting in the structure of Section 4.

Threats to Validity and Credibility. Our study has typical threats for qualitative analysis. Our results are affected by selection bias as developers who did not want to be interviewed may have different experiences. That is, they may not experience feature flags as a challenge (e.g., due to the smaller size of their system or due to better practices) or are not interested in this topic. Our interviewees tend to work on very large and commercial systems, and some participants (I7, I8) have experience with feature flags across many different companies, but interviewee I9 was the only one working on feature flags in an open-source context (we conjecture that feature flags are less commonly used in open source). Given the extensive experience of the interviewees and the quick saturation we are confident that we have identified the relevant concerns.

4 DIFFERENCES BETWEEN CONFIGURATION OPTIONS AND FEATURE FLAGS

In the following, we discuss commonalities and differences between configuration options and feature flags based on our interviews. In general, we found that the concepts of *configuration option* and *feature flag* are distinct but similar and it is often challenging to distinguish them. They are used for different purposes and there is value in distinguishing them explicitly as they have very different goals and come with different practices (e.g., test, document, or remove them). At the same time, they often face similar problems, and are sometimes amenable to similar solutions.

We introduce a clear distinction between the concepts in Subsection 4.1 and discuss differences in subsequent subsections. In Table 1, we summarize the key differences.

4.1 Goals

We distinguished three main goals for which our interviewees introduce feature flags in their code, even though they use the same technical mechanism to implement all of them:

- **Hiding incomplete implementations:** Hiding incomplete implementations is important when collaborating on trunk-based development, as one wants to avoid that an implementation affects other developers before it is finished; in this case, the implementation is hidden behind an *if* statement guarded by a Boolean flag that deactivates this code (by default); when the code is completed, the *if* statement and corresponding flag can be removed. This is a common practice when developing "at HEAD" as popularized by Google [36] and practiced by many other companies; and it is a strategy to avoid late merge conflicts.
- **Experimentation and release:** Developers are often interested in experimenting with different code paths, e.g., to compare outcomes in an A/B test or to test changes in a canary release with the ability to incrementally roll them out and to quickly roll them back if needed. These decisions are usually temporary and are not needed anymore once a decision has been made or a feature has been deployed.
- **Configuration:** Developers often want to offer choices to other stakeholders (e.g., end users, operations team, sales team) about which functionality to include. For example, configuration options are used to select which drivers to include in a specific Linux kernel or to decide on a page size for a database that balances query performance with storage efficiency (density). In software product lines, configuration options are planned strategically to serve many (actual or potential) customers in a domain [3, 35].

Our interviewees reported that they usually do not distinguish between different kinds of flags and do not explicitly keep track of the goals behind each flag. Moreover, the goals behind a flag can shift over time. For example, a feature flag that initially guards a new feature may become a configuration option that should be only available for 'premium' users. Alternatively, a configuration option such as a database's page size may be tweaked during experiments in production to find a good value.

The discussion above reveals an important distinction between feature flags and configuration options: The traditional notion of configuration options covers only the third goal, whereas the first two are specific to feature flags. That is, depending on one's perspective, configuration options can be seen as a special subset of feature flags or one can see feature flags as using configuration options for a new purpose. Therefore, *in the remainder of the paper, we refer to all configuration decisions that relate to the first two goals as feature flags and to all configuration decisions relating to the third goal as configuration options.* If the distinction is not important, we refer to them together as *configuration decisions*.

We argue that it is important to distinguish the different goals behind feature flags and configuration options. This sentiment is

Theme	Configuration Options	Feature Toggles
Goals	Customization	Continuous deployment / rollout / experimentation
Who makes configuration decisions?	End user / system vendor	Developer / operator
Complexity	High	Depends on number of flags
Removing configuration decisions	No	Yes (ideally)
Feature traceability	Depends on implementation	Often difficult due to indirections
Documentation	Description, dependencies, . . .	Owner, expiration date, . . .
Constraints	Complex	None or very few
Dependencies	Many, often hierarchical groups	Few, at most nesting or grouping
Feature interactions	Yes	Mostly not important
Testing	Sampling, systematic	Few specific configurations, single flips for unit tests

Table 1: Overview on differences between configuration options and feature flags.

also shared by interviewee I6: *"I definitely wish that there was a clear separation between feature flags and configuration flags."* As we will discuss, this distinction relates to dependent issues such as management, removal, documentation, and testing that differ based on the intended goal of a configuration decision.

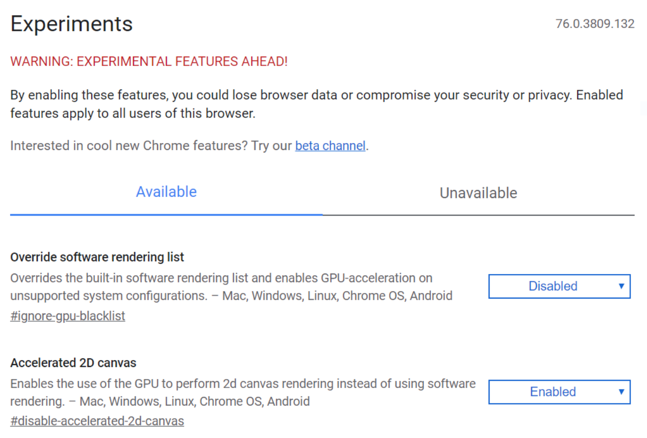
Key differences: Configuration decisions may be introduced either for concurrent development, for experimentation and release, and for configuration. These goals are distinct and have different associated challenges.

Recommendation: Clearly label the goals of each configuration decision, for example, using naming conventions to avoid confusion and technical debt.

4.2 Who is Making Configuration Decisions?

In the world of product lines and configurable systems, one key distinction that drives many other considerations (especially testing) is who is in charge of configuring the system:

- **Developers/operators configure:** In many traditional software product lines, companies release a small to medium number of distinct products (e.g., a basic, professional, and enterprise version of a product, or distinct variants for 15 different customers). Some features of a product or service may be activated for premium customers only. In these cases, developers or operators are in charge of configuring the system and users receive the configured (and tested) configuration. For example, HP's printer firmware has over 2000 Boolean flags, but the company releases only around 100 distinct printers, each of which goes through continuous integration before release [39].
- **Users configure:** Most end-user and system software has dozens or even thousands of configuration options with which end users can customize the software. For example, end users can use graphical interfaces to change various options in Firefox or Chrome, specify hundreds of parameters in Apache's `httpd.conf` configuration file, or select from over 14 000 compile-time options when they compile their own Linux kernel. When configuration is in the user's hands, developers do not know which configurations will be used eventually. In most cases (except for web-based systems and systems with good telemetry), they may not even know which configurations are actually used.


Experiments 76.0.3809.132

WARNING: EXPERIMENTAL FEATURES AHEAD!

By enabling these features, you could lose browser data or compromise your security or privacy. Enabled features apply to all users of this browser.

Interested in cool new Chrome features? Try our beta channel.

Override software rendering list
Overrides the built-in software rendering list and enables GPU-acceleration on unsupported system configurations. – Mac, Windows, Linux, Chrome OS, Android
#ignore-gpu-blacklist
Disabled

Accelerated 2D canvas
Enables the use of the GPU to perform 2d canvas rendering instead of using software rendering. – Mac, Windows, Linux, Chrome OS, Android
#disable-accelerated-2d-canvas
Enabled

Figure 1: Screenshot of Chrome's configuration dialog to enable experimental features.

This distinction has consequences for testing, because one can focus quality assurance on a few known configurations before they are delivered to the customer in the first case, but one may want to systematically make assurances for all potential configurations a user may select in the second case. Being in control of the configuration has further advantages: One can observe which configurations are used, such as monitoring which configuration decisions have not been changed in weeks; one does not have to care about all the combinations of configuration decisions that are not actually used; and one can safely remove configuration decisions without having to fear breaking user configurations.

None of our interviewed experts brought this distinction up by themselves, but they realized it when prompted. Most feature flags are clearly in the *operators configure* world. In most of our interviews the individual developers are in charge of configuring the feature flags. When used for A/B testing, for canary releases, or to hide unfinished features, they are controlled by the operation teams and the used configurations are (or should be) known to the operators. However, feature flags are sometimes also used for experimental releases that end users can configure (e.g., Chrome's 'chrome://flags/' [37], see Figure 1), blurring the distinction and raising testing challenges. Similarly, some interviewees reported that they expose experimental feature flags of their infrastructure software to other developers within the same organization, essentially turning them into configuration options and thus also giving up control over the used configurations.

Interviewees recognize a conflict here. On the one hand, early feedback from *beta users* who know that they are using new (unfinished) features can be beneficial. On the other hand, giving up control over the options may make it hard to remove them. It needs to be made clear that features are experimental (as in Chrome) to manage the expectations of end users who will use them. Interviewee I9 reported experience of exposing experimental feature flags as follows: *"We don't have control, especially if [company x] builds it into something and we're just like, well you know also that's not sound and it has problems and it will break and you'll lose some data [...] We do want people to experiment with it. But it's usually not everybody."*

Key differences: Operator-controlled configuration decisions have many advantages and most feature flags fall into this category. Exposing feature flags to end users has consequences, especially for cleanup and testing, and should be a carefully considered deliberate decision.

Recommendation: Explicitly plan and document configuration decisions that are exposed to end users; prefer operator-controlled decisions.

4.3 Complexity and Combinatorial Explosion

Each additional configuration decision in a system increases that system's complexity (independent of its goal), where the number of configurations that developers and tools may need to reason about grows exponentially with the number of configuration decisions. That is, developers spend additional time understanding the branching structure and thus have a harder time to deliver and implement new features. It becomes harder to test the system and reason about different configurations. For example, a recent study has shown that as few as three configuration options can severely challenge developers to correctly understand the behavior of 20-line programs [30].

Although one might argue that the huge number of potential combinations does not matter as long as one cares only about a small set of fixed configurations (e.g., because they are operator-controlled, see Sec. 4.2), developers may still need to reason about all the different code paths when maintaining or extending the implementation. Large configuration spaces are also challenging for those making the configuration decisions (operators or end users), who may need to decide between many alternatives, often without well understanding the choices, their consequences, or even their interactions [57]

This problem is well recognized in the literature on configurable systems and also our interviewed feature-flag experts are well aware, e.g., interviewee I4 remarked that feature flags challenge code comprehension *"especially if they are pre-existing because you wouldn't know why they are there or how they interact or how they are actually set"* and interviewee I6 further argues that feature flags *"lead to a lot of accidental complexity as opposed to apparent complexity."* There is a general agreement among our interviewees to keep the number of feature flags in check. Interviewee I7 suggests that having more than 20 feature flags per team is *"worrisome."* This aligns with the struggle to remove feature flags, as we will discuss.

Key differences: Feature flags and configuration options both add complexity – they challenge understanding, maintenance, and quality assurance. Feature flag experts generally try to keep the number of flags low.

Recommendation: Raise awareness of the complexity cost of configuration decisions and remove unnecessary decisions.

4.4 Temporary versus Permanent Configuration Decisions

The central difference that follows from the distinct goals of feature flags and configuration options (see Sec. 4.1) is that configuration options are usually intended to be *permanent* whereas feature flags are intended to be *temporary*.

Empirical evidence shows that configuration options are often added but almost never removed [25, 56]. It is cheap to introduce an option, but it can be expensive to maintain. If end users can configure the system, it can be very difficult to ever remove options that some users might use, especially since developers often do not know which options are actually used.

In contrast, our interviewees generally agree that feature flags for concurrent development, experimentation, and releases should be removed once the feature is completed, the experiment is done, and the feature is released. Since the value of feature flags and the rationale for introducing them are often known, there are great opportunities to track feature flags and introduce automation or enforcement mechanisms for removal. In practice though, feature flag removal and technical debt from not removing feature flags seem to be a key (process) challenge [37], which almost all interviewees strongly confirm (e.g., interviewee I1 states *"If you're not prompt on cleaning up, tech debt can actually wind up staying around a long time [...]. One of the big drivers of tech debt that we have is still code paths sticking around under a feature flag that's not being used anymore."*), and which we discuss next.

Key differences: Feature flags are generally temporary and should be scheduled for removal, which is not usually a concern for configuration options.

Recommendation: Be explicit about the expected lifetime of a feature flag and the condition when it will become obsolete.

4.5 Removing Configuration Decisions

As discussed in Sec. 4.3, each configuration decision increases complexity and may make reasoning, testing, and debugging harder. Removing configuration decisions can fight this growing complexity.

Classic configuration options are usually intended as permanent and are rarely removed, despite complaints that many options are rarely used and that too large configuration spaces are challenging for developers and users [56]. In contrast, the temporary nature of feature flags suggests that feature flag removal should be common, though both observations from Rahman et al. [37] in Chrome and comments from most of our interviewees indicate that developers intend to remove feature flags, but rarely do so consistently. In fact, removal of obsolete flags from the code and configurations was consistently identified as *the* key challenge for feature flags. For example, interviewee I3 explains that *"you develop the feature or maybe just a bug fix, you wrap it into a flag, you deploy it, and then you start working on something else. [...] [3 month later] when it could be removed, it's very hard to think about, go back, and remove this one feature flag that wrapped five lines of code."*

Deciding which configuration decisions can be removed can be challenging. In feature flags that decision is often associated with completing a specific task (e.g., finishing a feature implementation, an experiment, or a rollout), whereas for configuration options it may be less obvious. While users and developers tend to complain about too many options [56], configuration options are often kept in case they might be useful for special or future use cases. Furthermore, who makes configuration decisions (see Sec. 4.2) influences how challenging it is to identify what can be removed: When users configure the system, it is often not clear which configuration options are actually used, whereas in operator-configured systems one can usually identify unused options.

Once one decides to remove a configuration decision, the technical removal requires changes to the implementation and potentially multiple places of the configuration infrastructure that manage the list of options and their values. In theory, removal could be as simple as removing an *if* statement and a Boolean variable, but if the scope of the implementation is not well understood it is easy to make mistakes. Even for *#ifdef* directives, removal is more complicated than one might expect due to possible data-flow among options with *#define* and *#undef* statements [8]. Conceptually, removing an option is merely a form of partial evaluation [22], where a program is specialized for known values of certain options, but in practice few practical tools exist for removing options.

Our interviews revealed that removing feature flags is still a mostly manual process, if done at all. Interviewee I8 points out that indirections (e.g., use of design patterns to hide feature flags) make it hard for engineers to remove feature flags safely. In addition, two interviewees explain that they face technical dependencies in the configuration infrastructure that require multiple steps: first removing the *if* statement, then all configuration settings, and finally the configuration declaration – each step having to go through code review and continuous integration separately, turning a seemingly simple removal process into a week-long tedium. One interviewee reports observing that late removal is particularly challenging if the task is given to new developers after the original developer left the team (a common occurrence and pain point in their organization).

How strictly removal is pursued differs from project to project and we heard very different approaches. In general, the stricter the process of removing flags the fewer problems and pain points with feature flags our interviewees report. Some organizations track a feature flag's lifetime and, once it expires, automatically create an issue that assigns to the original creator the task of removing the flag. Other organizations define a maximum number of flags per team. One organization even fails the build if stale feature flags are detected, thus pressuring developers into immediate removal. Interviewees from organizations that do not enforce cleanup as part of the development process reported that feature flags accumulate.

A common theme among interviewees who report often lacking removal and growing technical debt is that developers have no incentive for removal. They are not required by policies or tools and the short-term cost of removal is not offset by a clear long-term benefit of a simpler code base. In fact, it was a common theme that the cost-benefit calculation is difficult to make and that many developers have no appreciation of the potential long-term cost. The cost of leaving flags in is difficult to estimate as it manifests very differently and is only observable indirectly. For example, interviewee I6 reported that *"we identified [...] tens of thousands of flags that, as far as we know from our available data sources, were no longer necessary. [...] And the refactoring involved to actually safely remove all of the flags was fairly labor intensive. So I think in practice we removed like 1% of the things that we identified and couldn't really identify, in provable terms, how much does that matter?"* Both managers and developers tend to prefer developing new features over cleanup. Interviewee I2 performs specific training and gives talks to educate developers and managers about feature-flag related technical debt, to foster a mindset favorable of cleaning up feature flags.

Our interviewees generally agree that it is good to make the cleanup of a flag part of finishing a feature. That is, flags should be removed timely following a (possibly automatically enforced) policy, for example making removal mandatory after two stable builds where the feature did not cause any problems. It might be useful to invest in tools that simplify such removal, as some interviewees have done in their organizations.

Key differences: Feature flags are usually temporary and developers often intend to remove them, but rarely do so unless forced by policy or technical steps. Removal can be a significant pain point.

Recommendation: Implement a process and explore automation for removing feature flags.

4.6 Documenting Feature Flags

In the product line community, features are a central mechanism for communication, planning, and decision making. Much effort is taken to explicitly *document features* (and options) and *their dependencies*. The description of features, their possible values, and their constraints is explicitly separated from the actual values chosen for any specific configuration. Simple notations such as *feature diagrams* [3, 12] are widely adopted to group and document features and especially to describe constraints on possible configurations (most prominently documenting multiple features to be optional, mutually exclusive, depending on another, or in a hierarchical relationship where child features depend on parent features). Clear documentation of constraints enables automated reasoning and checking of configurations.

Beyond tools focusing on product lines like FeatureIDE[1] [28] or pure::variants,[2] a great example is the Linux kernel's variability

[1]https://featureide.github.io/

[2]https://www.pure-systems.com/

model [46], for which the kernel developers have built their own domain-specific language to describe and document options and an interactive configurator to select configurations that adhere to all constraints. These *kconfig* files, illustrated in Listing 2, are part of the Linux kernel source tree and versioned with git. Configurations are simple option-value mappings in a separate file that can be generated and checked by tools that process the kconfig language. This is used for over 14 000 configuration options in the kernel alone.

Listing 2: Excerpt of the Linux configuration model

```
menu "Power management and ACPI options"
depends on !X86_VOYAGER
config PM
    bool "Power Management support"
    depends on !IA64_HP_SIM
    ---help---
    "Power Management" means that ...
config PM_DEBUG
    bool "Power Management Debug Support"
    depends on PM
config PM_SLEEP
    bool
    depends on SUSPEND || XEN_SAVE_RESTORE
    default y
```

Academic research has invested a considerable amount of effort into tools that can work with such documented feature models, for example, detecting inconsistencies among constraints [6, 9], analyzing the evolution of model changes [25, 53], resolving conflicts in configurations [55], or guiding humans through the configuration process [18, 45]. For configurable systems more broadly, Sayagh et al. [42, 43] has found quite some consensus on how to specify, load, and document options.

In contrast, in our interviews we found that feature-flag practitioners pay significantly less attention to documentation and dependency management. They often use ad-hoc mechanisms and spread configuration knowledge across many source files and configuration files. While several interviewees wished for better documentation practices, especially for learning about the purposes and goals behind various feature flags, many also indicate that more sophisticated mechanisms for describing dependencies are rarely needed and would be considered as a sign of over-engineering a simple concept. Given the short-lived nature of feature flags, they argue that documentation is less important and simple comments and grouping mechanisms are sufficient. Almost all interviewees suggested that dependencies among feature flags are rare, at most there is nesting of feature flags within a feature controlled by one other feature flag; there is usually no concept of an invalid combination of flags. Hence, they see limited need for a more complicated modeling language, e.g., interviewee I7 notes *"I think that the potential benefits of having that capability [of specifying dependencies] are far outweighed by the extra complexity, in terms of implementing it and in terms of reasoning about it."* Having to document constraints among flags is even seen by some as a sign of technical debt, as there are too many flags in the system.

At the same time, our interviewees identify useful metadata, that is currently rarely tracked formally, such as (a) the flag owner, to know who is responsible for it and who should eventually be blamed for failures and for cleanup, (b) an expiration date or event such as successful integration, (c) a description, (d) the location to find the flag in the code, and (e) valid states the flag can take. We suspect that feature flag practitioners can learn from best practices in the product line community, to use a single (and simple) unified configuration language and maintain a version-controlled central configuration model that ensures that all options are documented in a single place.

Finally, we note that documentation and implementation of features may drift apart. For example, researchers have found flags in the Linux kernel implementation that can never be enabled, as well as documented flags that are never used in the implementation [51]. A static consistency checker that assures that only documented options can be used in the implementation and that all documented options are used in the implementation can be a good idea and is easy to implement. In addition, more advanced static analyses have been developed in academia to identify dependencies among features from the implementation and check whether those align with the documented constraints [e.g., 31].

Key differences: Documenting configuration options and their dependencies has been explored in depth in product lines, enabling many forms of automation. Documentation requirements for feature flags are different given their short-term nature, but current feature flag practice seems to largely rely on ad-hoc mechanisms.

Recommendation: Set and enforce clear documentation standards for feature flags and configuration options.

4.7 Tracing Configuration Decisions

For many tasks, from debugging to removal, developers may need to find and understand the implementation associated with a feature flag or configuration option. In some cases, tracing configuration decisions to corresponding implementations is fairly straightforward, e.g., search for the *#ifdef* or *if* statement that evaluates a well-named flag and guard a few statements. However, tracking a configuration decision's flow from where it is defined to where it is used in an *if* statement, and identifying what code is guarded directly or indirectly by that *if* statement can be nontrivial, when configuration values are stored in intermediate data structures (e.g., hash maps) or propagated across function calls and various variables (in control flow and data flow). Furthermore, even when only few statements in a method are directly guarded by a flag, these few statements can of course invoke lots of other code elsewhere, that might not obviously be associated with this decision [24].

Two strategies may help to simplify traceability:

- **Disciplined implementation:** Tracing and analysis can be much simplified if *discipline* is used for implementing configuration decisions. A key hygiene strategy is to *separate configuration options as much as possible from other computations in the program*; that is, avoid using options as input parameters to more sophisticated implementations, but rather mostly propagate them to the *if* statement where they make a decision. In our interviews, most feature-flag practitioners report using APIs rather than local variables to query decisions (see Listing 1) and using those API calls

```
int v = Options.SDKVersion;
boolean wifiOn = Options.WIFI_ON
boolean proxySupported;
if(v > 8)
  proxySupported = true;              SDK>8
else
  proxySupported = false;             SDK≤8
if(proxySupported && wifiOn)
  [...]                          SDK>8 ∧ WIFI
```

Figure 2: Understanding how code fragments rely on Android's options [24].

almost exclusively in *if* statements.

- **Modularity:** There are many attempts at modularizing features in product lines into modules or plug-ins, and driving configuration and composition at an architectural level [35]. Many researchers have even explored dedicated language mechanisms (feature modules, aspects, delta modules), although they have not seen much adoption in practice yet [3, 7]. Research has also shown that, in configurable systems, configuration options usually have *fairly local effects* and *do not interact with most other options* (but also that effects of options are often not strictly local and often cause indirect effects through control-flow and data flows) [23, 24, 29, 40]. Several feature-flag practitioners report that they try to hide features behind abstractions to simplify reasoning about their impact and ease cleanup; they tend to try to localize feature code to a single *if* decision per feature flag.

Most developers seem to already follow a fairly strong discipline for configuration options and feature flags, because they still want to be able to reason about their effects. Interviewees confirm concerns that feature flags that interact and hide in complicated data flows are difficult to reason about for humans and analysis algorithms alike due to the combinatorial explosion of possible combinations and corresponding code paths; hence such implementations are usually avoided. With regard to modularity, we heard different opinions: While most interviewees value abstractions, especially for larger features, some explained that the extra effort and added implementation overhead for the abstraction step is not worth it for short-lived and small features.

More broadly, our interviews revealed different solutions for traceability issues. In almost all cases, finding code that belongs to a feature flag involves manual code search. However, interviewees I1 and I5 put additional comments and metadata in the implementation to establish a clearer mapping. Interviewee I2 described how they document features they work on in commit messages such that *git-blame* can reveal which feature a line of code belongs to. Finally, interviewee I3 reported that they were working on dedicated IDE support to show context information in the editor (without giving much technical detail on how the information is retrieved).

Regarding configuration options, several researchers have developed analysis tools to track configuration options across various data and control flows, for example, to approximate possible interactions, detect the scope of an option's implementation, or detect dependencies among options as shown in Figure 2 [2, 24, 29, 31, 47, 51, 57–59]. These analyses are more precise for more disciplined implementations that better separate configuration logic from program logic. None of our interviewees were aware of any such tools for feature flags.

Key differences: Identifying how flags relate to implementations can be surprisingly complicated, but discipline in the implementation and modularity can make both manual and automated analyses easier.

Recommendation: Keep implementations simple and separate configuration logic from program logic.

4.8 Analysis and Testing

As discussed, configuration decisions drastically increase the complexity of a program (see Sec. 4.3). This makes any quality assurance activities that go beyond a single configuration challenging. Every Boolean configuration decision doubles the size of the configuration space and it quickly becomes infeasible to test all possible configurations – 320 independent Boolean options result in more possible configurations than there are atoms in the universe, which is still small compared to the Linux kernel's over 14 000 compile-time options.

Testing features behind a configuration decision in isolation is often not enough. Even when features work in isolation, they may interact with other features in unanticipated ways. Feature interactions have long been studied, e.g., in telecommunication systems [10, 21]. Feature interactions are a failure of *compositionality*: Developers think of two features as independent and develop and test them independently, but when composed, surprising things may happen. Often interactions are intended (true orthogonality is rare) and extra coordination mechanisms are needed, such as prioritizing one feature over another. Coordination is usually easy once interactions are known, however finding and understanding interactions is nontrivial, especially when features are developed separately and there are no clear specifications.

Feature interactions are a key problem in software product lines and configurable systems and much research has been devoted to detection and resolution. Our interviewed feature-flag experts were often less concerned about interactions. Even though all of them acknowledge the possibility of interactions among feature flags, and most had at least one story to share, they consider interactions as rare and consider most of their features guarded by flags as independent and orthogonal. For example, interviewee I5 said *"I don't believe we really have many flags that are dependent on each other."* If they cared about interactions, they often did not know where to start – for example interviewee I3 reported that *"it's very hard to figure out which kinds of combinations of flags need to be tested."*

For quality assurance of configurable systems, there are generally two strategies: test exactly the configurations that are going to be deployed or attempt to assure quality across the entire configuration space. The strategy of testing individual configurations is commonly used in operator-controlled settings with few configurations. With feature flags it is common to run the specific configurations that are going to be deployed (e.g., in an A/B test or deployment) through continuous integration first, while not performing any tests on any other configurations. With this strategy bugs and interactions among specific configuration decisions may remain undetected until an affected configuration is actually needed, but it also means that developers do not proactively need to fix interactions that never occur in any used configurations.

In contrast, if configurations are user-controlled or if operators plan to change between configurations rapidly and frequently (e.g., in self-adaptive systems), it may be worth to proactively attempt to identify defects across the entire configuration space. While separately testing all configurations is clearly infeasible for all but the smallest configuration spaces, there are many different approaches to cover large configuration spaces (e.g., sampling configurations).

None of the feature-flag practitioners we talked to adopt a proactive strategy for quality assurance of the entire configuration space. Typically, they are vaguely interested but regard the strategy as too expensive. Most organizations ensure that each configuration undergoes continuous integration tests before it is deployed. In day to day practice though, it seems that many teams make changes to configurations without testing the specific configurations – which is dangerous. Interviewees reported that they sometimes rapidly change to untested configurations in emergency situations, e.g., to quickly roll back a feature that crashes the entire system. Some discussed that one could build infrastructure to restrict such emergency changes to tested configurations, but they had yet to implement such infrastructure.

To proactively explore large configuration spaces, there are different strategies. While some academic approaches can even provide guarantees for an entire configuration space [52, 54], the more pragmatic and ready to use approaches systematically sample configurations from the configuration space. The most promising approach is *combinatorial testing* [33]: Combinatorial testing selects a small set of configurations, such that every combination of every pair of options is included in at least one configuration, as exemplified in Table 2. Since a single configuration can cover specific combinations of many pairs (e.g., a single configuration A, B, !C covers A and B together as well as A without C and B without C), combinatorial testing can cover interactions among many options with very few configurations (e.g., 18 test configurations for pairwise coverage of 1000 Boolean options). It is easy to get started with NIST's existing tables[3] or with one of the many academic and commercial tools to generate configurations. Many other sampling strategies for large configuration spaces have been explored and compared [27].

None of our interviewees were aware of combinatorial testing or other systematic sampling strategies. They were surprised to learn how small covering arrays could be even for large configuration spaces. We suspect that this is primarily a technology transfer problem and expect that it would be easy to integrate systematic sampling strategies in a feature flag infrastructure and test regime.

Finally, automated tests are only as good as the executed tests. It can be a good idea to write feature-specific tests that are only executed if the feature is enabled, and test the behavior of that feature independent of other features [4, 15, 32]. Only interviewee I8 described any specific test strategies for feature flags.

[3]https://math.nist.gov/coveringarrays/ipof/tables/table.2.2.html

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0	0	1	0	0	0	1	1	1	0	0	1	1	1	1
0	1	0	1	1	0	0	0	1	1	0	0	0	1	1
1	0	0	1	0	1	0	1	0	0	0	0	0	1	0
1	1	1	0	1	1	0	1	1	0	1	1	0	0	0
1	1	0	0	0	0	1	0	0	1	1	1	1	0	1
0	0	1	1	1	1	1	0	0	1	1	0	1	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	1	0	1	1	1	1	1	1

Table 2: With 8 configurations all pair-wise combinations of 15 options (A–O) can be covered. For every pair of options there is at least one configuration that enables both options, one of the options, or none of the options.

Key differences: While configurable systems research has produced many strategies to proactively assure the quality of the entire configuration space, feature flag practitioners tend to test only individual configurations. It is controversial whether testing larger configuration spaces would be useful.

Recommendation: Explore the feasibility of random sampling and combinatorial testing.

5 CONCLUSIONS

Feature flag and *configuration option* are similar concepts, but we argue that they have distinguishing characteristics and requirements. This paper explores commonalities and differences between them. We performed nine semi-structured interviews with feature-flag experts and contrasted the findings with existing literature and research from configurable systems. We suggest to explicitly distinguish configuration option and feature flag as separate concepts, but also point out the many opportunities to transfer knowledge and tools, for example for more systematic testing of feature flags.

ACKNOWLEDGEMENTS

We are grateful to our interviewees for their time and input. We also appreciate the many informal discussions on the topic with other researchers. Kästner, Meinicke, and Wong have been supported in part by the NSF (awards 1552944, 1717022, and 1813598) and AFRL and DARPA (FA8750-16-2-0042). Vasilescu has been supported in part by the NSF (awards 1717415 and 1901311).

REFERENCES

[1] Michalis Anastasopoules and Critina Gacek. 2001. Implementing Product Line Variabilities. In *Proc. Symposium on Software Reusability (SSR)*. ACM, 109–117.
[2] Florian Angerer, Andreas Grimmer, Herbert Prähofer, and Paul Grünbacher. 2015. Configuration-aware change impact analysis (t). In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. IEEE, 385–395.
[3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer-Verlag.
[4] Sven Apel, Alexander von Rhein, Thomas Thüm, and Christian Kästner. 2013. Feature-Interaction Detection based on Feature-Based Specifications. *Computer Networks* 57, 12 (2013), 2399–2409.
[5] Eytan Bakshy, Dean Eckles, and Michael S Bernstein. 2014. Designing and deploying online field experiments. In *Proc. Int'l Conf. World Wide Web (WWW)*. ACM, 283–292.

[6] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas.. In *Proc. Int'l Software Product Line Conference (SPLC) (Lecture Notes in Computer Science)*, Vol. 3714. Springer-Verlag, 7–20.

[7] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. 2004. Scaling Step-Wise Refinement. *IEEE Trans. Softw. Eng. (TSE)* 30, 6 (2004), 355–371.

[8] Ira Baxter and Michael Mehlich. 2001. Preprocessor Conditional Removal by Simple Partial Evaluation. In *Proc. Working Conf. Reverse Engineering (WCRE)*. IEEE, 281–290.

[9] David Benavides, Sergio Seguraa, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (2010), 615–636.

[10] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. 2003. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks* 41, 1 (2003), 115–141.

[11] Paul Clements and Linda Northrop. 2001. *Software Product Lines: Practices and Patterns*. Addison-Wesley.

[12] Krzysztof Czarnecki and Ulrich Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. ACM/Addison-Wesley.

[13] Patricio Echagüe and Pete Hodgson. 2019. *Feature Flag Best Practices*. O'Reilly Media, Inc.

[14] Martin Fowler. 2010. FeatureToggle. https://martinfowler.com/bliki/FeatureToggle.html

[15] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. 2012. Test Confessions: A Study of Testing Practices for Plug-In Systems. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, 244–254.

[16] Rebecca M. Henderson and Kim B. Clark. 1990. Architectural Innovation: The Reconfiguration of Existing Product Technologies and the Failure of Established Firms. *Administrative Science Quarterly* 35, 1 (March 1990), 9–30. Special Issue: Technology, Organizations, and Innovation.

[17] Pete Hodgson. 2017. Feature Toggles (aka Feature Flags). https://martinfowler.com/articles/feature-toggles.html

[18] Arnaud Hubaux, Patrick Heymans, Pierre-Yves Schobbens, Dirk Deridder, and Ebrahim Khalil Abbasi. 2013. Supporting multiple perspectives in feature-based configuration. *Software & Systems Modeling* 12, 3 (2013), 641–663.

[19] Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. 2012. A User Survey of Configuration Challenges in Linux and eCos. In *Proc. Int'l Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*. ACM, 149–155.

[20] Jez Humble and David Farley. 2010. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education.

[21] Michael Jackson and Pamela Zave. 1998. Distributed Feature Composition: A Virtual Architecture for Telecommunications Services. *IEEE Trans. Softw. Eng. (TSE)* 24, 10 (1998), 831–847.

[22] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall.

[23] Chang Hwan Peter Kim, Darko Marinov, Sarfraz Khurshid, Don Batory, Sabrina Souto, Paulo Barros, and Marcelo d'Amorim. 2013. SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 257–267.

[24] Max Lillack, Christian Kästner, and Eric Bodden. 2018. Tracking Load-time Configuration Options. *IEEE Trans. Softw. Eng. (TSE)* 44, 12 (2018), 1269–1291.

[25] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2010. Evolution of the Linux Kernel Variability Model. In *Proc. Int'l Software Product Line Conference (SPLC)*. Springer-Verlag, 136–150.

[26] Rezvan Mahdavi-Hezaveh, Jacob Dremann, and Laurie Williams. 2019. Feature Toggle Driven Development: Practices usedby Practitioners. *arXiv preprint arXiv:1907.06157* (2019).

[27] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. In *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 643–654.

[28] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer.

[29] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. 2016. On Essential Configuration Complexity: Measuring Interactions In Highly-Configurable Systems. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. ACM, 483–494.

[30] Jean Melo, Claus Brabrand, and Andrzej Wąsowski. 2016. How does the degree of variability affect bug finding?. In *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 679–690.

[31] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2015. Where do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *IEEE Trans. Softw. Eng. (TSE)* 41, 8 (2015), 820–841.

[32] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. 2014. Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 907–918.

[33] Changhai Nie and Hareton Leung. 2011. A Survey of Combinatorial Testing. *ACM Computing Surveys (CSUR)* 43, 2 (2011), 11:1–11:29.

[34] Chris Parnin, Eric Helms, Chris Atlee, Harley Boughton, Mark Ghattas, Andy Glover, James Holman, John Micco, Brendan Murphy, Tony Savor, et al. 2017. The Top 10 Adages in Continuous Deployment. *IEEE Software* 34, 3 (2017), 86–95.

[35] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, Berlin/Heidelberg.

[36] Rachel Potvin and Josh Levenberg. 2016. Why Google stores billions of lines of code in a single repository. *Commun. ACM* 59, 7 (2016), 78–87.

[37] Md Tajmilur Rahman, Louis-Philippe Querel, Peter C Rigby, and Bram Adams. 2016. Feature toggles: practitioner practices and a case study. In *Proc. Conf. Mining Software Repositories (MSR)*. ACM, 201–211.

[38] Md Tajmilur Rahman, Peter C Rigby, and Emad Shihab. 2019. The modular and feature toggle architectures of Google Chrome. *Empirical Software Engineering* 24, 2 (2019), 826–853.

[39] Jacob G. Refstrup. 2009. Adapting to Change: Architecture, Processes and Tools: A Closer Look at HP's Experience in Evolving the Owen Software Product Line. In *Proc. Int'l Software Product Line Conference (SPLC)*. Keynote presentation.

[40] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. 2010. Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems. In *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 445–454.

[41] Johnny Saldaña. 2015. *The Coding Manual for Qualitative Researchers*. Sage.

[42] Mohammed Sayagh, Zhen Dong, Artur Andrzejak, and Bram Adams. 2017. Does the choice of configuration framework matter for developers? Empirical study on 11 Java configuration frameworks. In *Proc. Int'l Workshop Source Code Analysis and Manipulation (SCAM)*. IEEE, 41–50.

[43] Mohammed Sayagh, Noureddine Kerzazi, Bram Adams, and Fabio Petrillo. 2018. Software Configuration Engineering in Practice: Interviews, Survey, and Systematic Literature Review. *IEEE Trans. Softw. Eng. (TSE)* (2018).

[44] Gerald Schermann, Jürgen Cito, Philipp Leitner, Uwe Zdun, and Harald Gall. 2016. *An empirical study on principles and practices of continuous delivery and deployment*. Technical Report 4:e1889v1. PeerJ Preprints.

[45] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. 2011. A Comparison of Decision Modeling Approaches in Product Lines. In *Proc. Int'l Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*. ACM, 119–126.

[46] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. 2010. The Variability Model of The Linux Kernel. In *Proc. Int'l Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*. University of Duisburg-Essen, Essen, 45–51.

[47] Larissa Rocha Soares, Jens Meinicke, Sarah Nadi, Christian Kästner, and Eduardo Santana de Almeida. 2018. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. ACM, 41–52.

[48] Ya-Yunn Su, Mona Attariyan, and Jason Flinn. 2007. AutoBash: Improving Configuration Management with Operating System Causality Analysis. In *Proc. Symp. Operating Systems Principles (SOSP)*. ACM, 237–250.

[49] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. 2015. Holistic configuration management at Facebook. In *Proc. Symp. Operating Systems Principles (SOSP)*. ACM, 328–343.

[50] Diane Tang, Ashish Agarwal, Deirdre O'Brien, and Mike Meyer. 2010. Overlapping experiment infrastructure: More, better, faster experimentation. In *Proc. Int'l Conf. Knowledge Discovery and Data Mining (KDD)*. ACM, 17–26.

[51] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *Proc. Europ. Conf. Computer Systems (EuroSys)*. ACM, 47–60.

[52] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys (CSUR)* 47, 1 (6 2014), Article 6.

[53] Thomas Thüm, Don Batory, and Christian Kästner. 2009. Reasoning about Edits to Feature Models. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, 254–264.

[54] Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. 2018. Variability-Aware Static Analysis at Scale: An Empirical Study. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* 27, 4 (2018), Article No. 18.

[55] Yingfei Xiong, Hansheng Zhang, Arnaud Hubaux, Steven She, Jie Wang, and Krzysztof Czarnecki. 2015. Range fixes: Interactive error resolution for software configuration. *IEEE Trans. Softw. Eng. (TSE)* 41, 6 (2015), 603–619.

[56] 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. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 307–319.

[57] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do Not Blame Users for Misconfigurations. In *Proc. Symp. Operating Systems Principles (SOSP)*. ACM, 244–259.

[58] Tianyin Xu and Yuanyuan Zhou. 2015. Systems approaches to tackling configuration errors: A survey. *ACM Computing Surveys (CSUR)* 47, 4 (2015), 70.

[59] Sai Zhang and Michael D. Ernst. 2014. Which Configuration Option Should I Change?. In *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 152–163.