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 distinguish 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 a number of distinguishing characteristics but also opportunities for knowledge and technology transfer across both communities. Overall, we think that both communities can learn from each other.

1 INTRODUCTION
Feature flags (aka. feature toggles or feature flippers) have received a lot of attention recently, with practitioners discussing them and best practices around them widely in blog posts and at conferences, and multiple startups competing at developing 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.

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 and are relatively understudied, 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 [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 by 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 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 a number of distinct challenges. For example, testing challenges across large configuration spaces often resemble the community on software product lines [3, 11, 35] and the systems community [58]; 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 by the configuration options community?

Our overall goal is to educate researchers and engineers from both communities, feature flags and configuration options about each other.
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

Feature flags have rarely been studied in academic research. Feature flags are heavily discussed by practitioners and used in industrial settings for purposes like trunk-based development, A/B testing, and canary releases, which seem less common in the open-source community. Indeed, most artifacts around feature flags come directly from industry in form of talks and blog posts (e.g., [14, 17]) in the context of discussions on continuous delivery, continuous deployment, and continuous experimentation. Rahman et al. [37, 38] and Mahdavi-Hezaveh et al. [26] recently performed gray literature surveys of such blog posts and videos on feature flags, that provide an overview of common topics and challenges.

On the academic side, Rahman et al. [37, 38] analyzed toggle 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 cleanup 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 and software product lines and configuration options are not raised as a contrast in literature on 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 exist a lot of information on practices around using feature flags and a lot of research 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 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 and our own background for context about configuration options but choose to interview practitioners having extensive experience with feature flags to understand the flags and how developers think about them better.

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 experience working with feature flags. Six interviewees (I1–I6) work in three large software companies, one interviewee (I7) is a consultant on DevOps topics, one interviewee (I8) is an engineer for a feature-flag tooling service, and one interviewee (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 development.

Semi-Structured Interviews. We developed an interview guide structured 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 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 on (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 guided the interviews to cover the topics of our interview guide, though the order in which we asked the questions was not fixed, but rather guided by the flow of each individual interview and the experiences of the interviewee. During the interviews, we brought up practices and problems from configuration options regarding the discussed topics (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 us-
ing 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 au-
thor coded all transcripts and the second author refined the coding
to mediate false and incomplete characterizations. Based on the cod-
ing, we derived an overview of practices that use feature flags. This
overview enables us to identify new research directions and chal-
enges, as well as existing solutions and practices, including practi-
cal guidelines for feature flags, resulting in the structure of Section 4.

Threats to Validity and Credibility. Our study has typical threats
for qualitative studies. 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
different 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 key 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 amendable to similar solutions.

We introduce a clear distinction between the concepts in Sub-
section 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 implementation, even though they
use the same technical mechanism to implement all of them:

- **Hiding incomplete implementations**: Hiding incomplete im-
  plementations 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 a mechanism to avoid late merge conflicts.

- **Experimentation and release**: Developers are often interested
  in experimenting with different code paths, e.g., to compare out-
comes in an A/B test or to test changes in a canary release with
the ability to quickly roll them back. 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 op-
tions 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.

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 perspec-
tive, 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
also shared by interviewee I6: “I definitely wish that there was a
clear separation between feature flags and configuration flags”. As
we will discuss, they directly relate to dependent issues such as
management, removal, documentation, and testing that differ based
on the intended goal of the configuration decision.

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

Recommendation: Clearly label the goals of each configura-
tion option, 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 dis-
tinction 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 small number of printers, or a basic, professional, and enterprise version of a product). 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 have dozens or even thousands of configuration options that end users can choose between, to 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 about 14000 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 do not even know which configurations are actually used.

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, which 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, canary release, or to hide unfinished features, they are controlled by the operation teams and the used configurations are (or should be) known to the operators.
Key differences: Operator-controlled 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 make it really hard for 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, cf. 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 14 notes: “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 16 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. 117 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 are both concerns for complexity, challenging 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 indeed almost never removed [25, 56]. It is cheap to introduce an option, but potentially expensive to maintain them. If end users can configure the system, it can be very difficult to ever get rid of 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 here to track feature flags and introduce 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., 11 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 makes 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 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, while developers intend to remove feature flags, they 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 13 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 the 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, the task turned out to be 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 and developers rarely go through the effort.

Our interviews revealed that removing feature flags is still a mostly manual process. I8 points out that indirections (e.g., use of design patterns to hide feature flags) make it hard for engineers to remove them 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 (i.e., removing the #define and #undef directives). For example, interviewee I6 points out that the removal of a feature flag requires changes to the implementation 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 talks to educate developers and managers about feature-flag related technical debt, to change this mindset.

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 make 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 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 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 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 some 14 000 configuration options in the kernel alone.

Listing 2: Excerpt of the Linux configuration model

```c
menu "Power management and ACPI options"
depends on X86_VOYAGER

config PM
bool "Power Management support"
depends on !IA64_HP_SIM
```

1https://featureide.github.io/
2https://www.pure-systems.com/
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 checks a well-named flag and a few statements guarded by it. However, tracing 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, as configuration values are stored in intermediate data structures (e.g., hashmaps) or propagated across function calls and various variables (in control 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 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), though 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 data flows) [23, 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 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, I1 and I5 put additional comments and metadata in the implementation to establish a clearer mapping. I2 describes 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, 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 [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 (Sec. 4.3), configuration decisions drastically increase the complexity of a program. 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; the Linux kernel has over 14 000 compile-time options alone.

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 not only on 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 well recognized as 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. Though, all of them acknowledge the possibility of interactions among feature flags, and most had at least one story to share, they consider them as rare and consider most of their features guarded by feature 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, they often did not know where to start, for example interviewee I3 reports that “it’s very hard to figure out which kinds of combinations of flags that 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 practice, it seems that many teams make changes to configurations without testing the specific configurations – which is dangerous.
feature independent of other features [4, 15, 32]. Only interviewee
executed if the feature is enabled, and test the behavior of that
It can be a good idea to write feature-specific tests that are only
test only individual configurations. It is controversial whether
testing of larger configuration spaces would be beneficial.
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 require-
ments. 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 distin-
guish 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.

REFERENCES
[5] Eytan Bakshy, Dean Eckles, and Michael S Bernstein. 2014. Designing and deploy-
//doi.org/10.1109/TSE.2004.23
Analysis of Feature Models 20 Years Later: A Literature Review. Information
Systems 35, 6 (2010), 615–636. https://doi.org/10.1016/j.is.2010.01.001
Patterns. Addison-Wesley, Boston, MA.
Media, Inc.
feature-toggles.html
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.
Conference’17, July 2017, Washington, DC, USA

Jens Meineick, Chu-Pan Wong, Bogdan Vasilescu, Christian Kaestner