Software Configuration Engineering in Practice
Interviews, Survey, and Systematic Literature Review

Mohammed Sayagh, Noureddine Kerzazi, Bram Adams, Fabio Petrillo

Abstract—Modern software applications are adapted to different situations (e.g., memory limits, enabling/disabling features, database credentials) by changing the values of configuration options, without any source code modifications. According to several studies, this flexibility is expensive as configuration failures represent one of the most common types of software failures. They are also hard to debug and resolve as they require a lot of effort to detect which options are misconfigured among a large number of configuration options and values, while comprehension of the code also is hampered by sprinkling conditional checks of the values of configuration options. Although researchers have proposed various approaches to help debug or prevent configuration failures, especially from the end users' perspective, this paper takes a step back to understand the process required by practitioners to engineer the run-time configuration options in their source code, the challenges they experience as well as best practices that they have or could adopt. By interviewing 14 software engineering experts, followed by a large survey on 229 Java software engineers, we identified 9 major activities related to configuration engineering, 22 challenges faced by developers, and 24 expert recommendations to improve software configuration quality. We complemented this study by a systematic literature review to enrich the experts' recommendations, and to identify possible solutions discussed and evaluated by the research community for the developers' problems and challenges. We find that developers face a variety of challenges for all nine configuration engineering activities, starting from the creation of options, which generally is not planned beforehand and increases the complexity of a software system, to the non-trivial comprehension and debugging of configurations, and ending with the risky maintenance of configuration options, since developers avoid touching and changing configuration options in a mature system. We also find that researchers thus far focus primarily on testing and debugging configuration failures, leaving a large range of opportunities for future work.

1 Introduction

In September 2010, Facebook users experienced the following severe outage [50]:

> "*Early today Facebook was down or* ***unreachable*** *for many of you for approximately* ***2.5 hours****. This is the worst outage we've had in over four years. The key flaw that caused this outage to be so severe was an unfortunate handling of an error condition.* ***An automated system for verifying configuration values ended up causing much more damage than it fixed****.*
> ...
> *We're exploring* ***new designs for this configuration system*** *following* ***design patterns*** *of other systems at Facebook that deal more gracefully with feedback loops and transient spikes.*"

The above issue was not (directly) caused by a bug in the source code, but rather by the complexities of dealing with configuration in a software system. Instead of hardcoding the values of deployment-dependent constants such as URLs, or the choice of algorithm or feature that needs to be used, software systems extract such values into configuration options, which can then be modified by operators without having to recompile the source code. As such, software systems can easily be ported to a different environment or user base simply by reconfiguring the options. In the above example, Facebook used a database for storing these values.

While configuration options provide substantial flexibility to developers and users, they create a huge space of variability that can be hard to maintain [96], [115], [119], [49]. Configurable software systems can have hundreds of configuration options [112], which gives an effectively inexhaustible number of variants (2^{100} variants for a software system with 100 boolean options). Since it is impossible to test all variants before release, and options are not necessarily limited to only 2 values (they could be arbitrary strings!), certain configurations could lead to invalid behaviour of a software system or even an inability to boot up. Such situations are referred to as "configuration failures".

Configuration failures not only have a huge impact on availability, but they are expensive and common in both open source and commercial software systems [115]. Yin *et al.* [115] showed that configuration failures can account for 27% of all customer-support cases in industrial contexts, while a well-known Google engineer [104] highlighted them as one of the most important future research directions. In addition, other researchers [49], [16], [96] focused on preventing configuration failures, where a second string of efforts, including [106], [123], [66], [32], [31], [19], [20], [118], [117], [119], [18], [71], [98], have been conducted to help users troubleshoot such kinds of problems.

Despite all this research on dealing with configuration failures, especially from the end users' perspective, there is hardly any work on understanding the effort involved in "configuration engineering" required from practitioners, i.e., the

- *Mohammed Sayagh, Bram Adams, and Fabio Petrillo are with the Department du Genie Informatique et Genie Logiciel, Ecole Polytechnique de Montreal.*
E-mail: mohammed.sayagh@polymtl.ca, bram.adams@polymtl.ca, fabio.petrillo@polymtl.ca
- *Noureddine Kerzazi is with the Ecole Nationale Superieure d'Informatique et d'Analyse des Systemes (ENSIAS) Rabat Morocco.*
E-mail: n.kerzazi@um5s.net.ma

Manuscript received April 19, 2005; revised August 26, 2015.

development activities that make up the process to establish and maintain configuration options throughout the lifetime of a software project. In general, there is a lack of understanding of the challenges practitioners experience with configuration engineering as well as best practices that are recommended. Such an understanding would not only open up new research directions (other than targeting configuration failures), but also document a catalogue of configuration engineering challenges and workarounds. The latter catalogue is essential to developers[1], since it is hard for them to determine this knowledge only from their own system's context.

Hence, this paper investigates the state of the practice and challenges of configuration engineering in the trenches by conducting a set of interviews, a large Java practitioner survey and a systematic literature survey. These allow us to make the following contributions:

- Deriving the typical process of configuration engineering, comprising 9 essential activities carried out by developers to create and maintain configuration options in a software system. This process starts by the creation of new configuration options, followed by the management of storage medium and data format (type) of these options and the design of how to access these options in the source code. The process also includes the comprehension and maintenance of options, sharing of knowledge about them, and both prevention and resolution of configuration failures.
- Identifying the configuration engineering challenges faced by practitioners related to the 9 activities. For instance, we found that development teams typically do not plan the design of potential configuration options nor do they have an adequate process for reviewing code changes involving options, which might have a negative impact on the quality, understandability and maintainability of software products. Similarly, perhaps surprisingly, developers are afraid of removing options from their code base in order not to break existing functionality.
- Collecting a set of recommendations, both from practitioners and researchers, to address the challenges. The recommendations include basic guidelines on how to organize options, when developers should decide adding a new option, what should be in configuration documentation, and where developers should define it. They also include a set of guidelines on how to access configuration in the source code, and how to debug and resolve configuration failures. A common thread within the guidelines is for developers to consider configuration options as code. Similar to source code, they should review new options, define and respect a clear naming convention, encapsulate access to configuration options within dedicated helper classes, and log used configuration values to help debug configuration failures. In turn, the resolution of these failures should be documented.
- A public dataset of our analysis to the interviews [11], and survey responses [12].

Given the large scope of software configurability, we focus explicitly on software configuration options that do not require re-compilation nor re-deployment to take into account new values, i.e., options whose value can still be changed at run-time. In other words, we focus on post-deployment configurability, and leave other forms of configuration engineering as future work.

The remainder of the paper is organized as follows: Section 2 presents the background about software configuration. Section 3 presents our study methodology. Section 4 presents the 9 identified configuration engineering activities, while Section 5 presents the challenges that we identified from qualitative and quantitative analysis of the 14 interviews and the 229 survey responses. Section 6 provides the recommendations identified during our interviews, survey and systematic literature review. Section 8 discusses threats to validity, while Section 9 concludes with insights and future work.

2 Background on Software Configuration

This section defines the notion of software configuration used by this paper and discusses different types of configuration.

2.1 Software Configuration Options

Configuring a software system consists of customizing and adapting an application as well as its execution environment [44] to different users' requirements and contexts [87]. Conceptually, a configuration option of a software application corresponds to a key-value pair, for example to quickly enable or disable features and algorithms. A key represents a configuration option's name, whereas its value represents the choice by a practitioner or user regarding that option for a specific instance of the system. For example, to disable file uploads in a Wordpress installation, one can just switch off the PHP configuration option "file_uploads", without the need to change the actual PHP source code. "file_uploads" and "Off" are respectively the key and value of the corresponding configuration option.

One of the most common storage media for configuration options are ordinary files, which we refer to by "configuration files". Developers typically can choose to store their software system's options in different textual file formats, such as "txt", "ini", "xml", or "json", or even to use or design a configuration domain-specific language (DSL) [22] for their software system. Apart from configuration files, developers could also store their configuration options in a relational database, which can be accessed via a user interface, or even directly modified by end users. More recently, distributed key-value stores like Apache ZooKeeper [2] have become widely used for managing configuration options in distributed environments. Alternative means of storing configuration options would be as conditional compilation constants or even as command line arguments passed to an application.

2.2 Roles Involved in Software Configuration

Many roles are involved in the creation and usage of software configuration, which we can classify in two main categories: (1) the creators and (2) the consumers of a software configuration. The creator category consists of a variety of engineering roles, including architects, designers, developers, and testers. All of these are technically minded, and have access or influence on the resulting source code of a software system.

1. This was mentioned a number of times in our developer survey.

On the other hand, the consumer category comprises both technical and non-technical configuration users. Technical users include the same engineering roles as the creator category, but also roles responsible for software operations, such as deployers, integrators, and operators. These users usually manipulate technical configuration options related to both the execution environment and application, such as resource paths, accounts or options to tweak the security and performance of an application.

In contrast, non-technical users are the final consumers of a software application and do not necessarily have a technical background. They mostly change configuration options to help them achieve their objectives, such as information about their account, preferred algorithms or features, or GUI configuration.

2.3 Configuration vs. Binding Time

Given that configuration options could target different roles, who have different types of access to the source code and/or deployed version of a software system, software configuration typically is performed during three different phases. These phases determine the "binding time" of configuration options [58], i.e., the moment on which the options are assigned their values: before deployment, during deployment or after deployment.

Pre-deployment options typically are implemented using compile-time configuration options, which decide whether particular code snippets will be compiled in by the build system. For instance, in C/C++, a code snippet occurring between "#ifdef Config_Option" and "#endif" delimiters will be observed and hence compiled by a compiler. This happens only if the "Config_Option" option has been defined in a source code file or as a command-line parameter for the compiler (typically specified within a build script), otherwise the corresponding code simply is unavailable to end users in the resulting executable. Changes to compile-time options require recompilation of the source code.

An alternative means to implement pre-deployment binding of configuration options is to use build scripts such as Makefiles, which can select the set of source code files that should be compiled, or even the set of 3rd party libraries (and their specific versions) that should be linked to the compiled application code. This is common in large systems such as the Linux kernel, where a kernel should be as streamlined as possible and hence only contain the drivers expected on the target platform.

Next, during deployment (sometimes called "installation"), modern web apps use domain-specific languages (DSLs) such as Puppet, Chef, or Ansible to install and configure the environment of an application, i.e., the virtual machines, containers, or servers on which the application will run, with the required operating system, database, web server, selection of 3rd party libraries, network connections, etc. Such DSLs are not used for desktop or mobile apps, where instead an installer performs any environment-related configuration. Configuration scripts written in one of these DSLs or bundled within an installer basically auto-detect values for environment-related options, use those values to fill out (bind) template configuration files (e.g., of a web server or even the application), then move the instantiated templates to the right location in a virtual machine, container or computer.

In contrast to pre-deployment binding, however, options bound during deployment can still be tweaked after deployment. This is because each component in the environment of an application A is itself a software application, with its own collection of pre-deployment, deployment-time and post-deployment options. The main difference, from A's perspective, is that most of its environment's options will be bound during deployment, while most of A's options will be bound afterwards, because the latter are expected to be handled by the end user.

Finally, after deployment, two different binding times are possible, i.e., load-time and execution-time. Load-time configuration options are read by the source code when a software system boots up. Upon boot, the software system reads all the options that are defined in its configuration storage medium. Changes to these options just require restarting the application. Execution-time configuration options are the most flexible form of configuration options, since they do not depend on recompilation nor rebooting of the whole application. Instead, user modifications can be taken into consideration immediately after changing, provided the method responsible for configuration access is re-executed.

2.4 Run-time Configuration Options

This study focuses on the process followed and challenges experienced by developers when building in software configuration aimed at end users, sysadmins and operators. These roles have access to a system during and/or after deployment, hence at first sight our focus is on options bound during those two phases, i.e., options that are propagated via and can impact data and control flows in the source code. Pre-deployment options fall outside our scope, since such options typically only filter out unnecessary parts of the source code, then "disappear" during compilation.

However, the distinction between the binding times presented in the previous section is not always as strict, hence defining the scope of our study in terms of them introduces ambiguity. For example, during compilation a configuration tool like autoconf might have bound a system-specific value (e.g., the maximum integer size) to a C variable representing a configuration option, while this variable might be re-bound to a different value by end users at run-time. Similarly, most of the configuration options bound to an auto-detected value during deployment can still be overridden at run-time. For example, during installation an installer could detect 4 GB RAM on a laptop, yet after upgrading the memory to 8 GB RAM, a laptop owner should not need to reinstall (redeploy) the application, but just manually change 4 to 8 in the application's configuration file. In these (and other) cases, the exact type of binding is unclear.

To deal with this ambiguity, we instead define the scope of our study in terms **"run-time configuration options", i.e., configuration options whose value can still be changed by the end user, without having to re-deploy**. This is an approximation that roughly coincides with deployment-time and post-deployment options, while still including edge cases like the examples of maximum integer size and RAM amount. On the other hand, most of the pre-deployment options (in particular conditional compilation-based options) are excluded. Due to their different target audience, we believe that they likely face different challenges than run-time

options, and hence different workarounds and best practices. Future work should replicate our study on pre-deployment configuration options.

2.5 Configuration Failures and Faults

Assigning an incorrect value to a configuration option can lead the software to behave incorrectly, which is referred to by the term "configuration failure". Such failures can have different manifestations, including error messages, performance degradations, hidden and silent execution failures, or even crashes or refusal to boot up. If they occur in production, they can incur serious financial losses [4], [6].

Typical causes for configuration failures, i.e., configuration faults, include typos (e.g., "memory_limit = 64" instead of "memory_limit = 64M") as well as unrealistic or plain wrong values. A second type of configuration faults correspond to options whose value is syntactically correct, but does not match with the execution requirements, (e.g., "max_execution = 60" when a time-consuming script that requires more than 60 seconds is executed). For other types of configuration faults, including inconsistencies between configuration option values, we refer elsewhere [115].

2.6 Software Configuration Engineering

We define software configuration engineering as the discipline that encompasses activities involved in the creation, integration, and maintenance of run-time configuration options in a software application. For example, adding a new option to an existing code base, and managing configuration failures are two common configuration engineering activities. Other engineering activities also apply to software configuration including documentation and code review. While there exist substantial work on several activities related to software configuration engineering, many others are not covered at all by literature. Furthermore, there is no study that focuses on the full lifecycle of software configuration options. In fact, until now no official name existed for the overarching discipline of software configuration engineering. Hence, the next section discusses the methodology of our exploratory study on the different configuration engineering activities (as well as their challenges and recommendations).

3 Study Methodology

In order to understand the activities making up the discipline of software configuration engineering, the challenges involved with those and best practices that exist, we used a mixed-methods approach involving semi-structured interviews, a survey with open source stakeholders and a systematic literature survey. The 14 interviews with experts allowed to define 9 essential configuration engineering activities as well as to identify an initial list of challenges of these activities. The activities and challenges were used as inspiration for the open-ended and closed questions of our survey. The 229 survey responses allowed to validate the activities and challenges, as well as to identify best practices used to deal with these challenges. Finally, to put our findings into perspective, we performed a systematic literature survey focused on the 9 activities. The rest of this section provides more details about our study methodology.

TABLE 1
Interviewed subjects' experience and role.

Subject	Experience (#years)	Role
P1	28	Developer and Manager
P2	27	Researcher
P3	21	Developer
P4	15	Manager
P5	14	Developer and Industrial Researcher
P6	14	Infrastructure Architect
P7	13	Manager
P8	12	Developer and Architect
P9	10	Developer and Architect
P10	10	Developer
P11	8	Integrator
P12	7	Developer
P13	6	Developer
P14	5	Developer, Architect, and Manager

3.1 Semi-structured Interviews

Since we aim to understand activities, challenges and best practices used in practice, and no explicit configuration engineering process could be used as reference, we opted to start with semi-structured interviews of experts. Then, to put the experts' input into context, we performed a larger-scale survey building on the expertsâĂŹ answers. To conduct the semi-structured interviews, we first had to define the base set of open questions to discuss with experts, which only requires high-level guiding questions to explore concrete experiences and insights of the interviews. These open questions were obtained through brainstorming of the paper's authors around the central theme "*How do practitioners design, develop and maintain the configuration system of their application?*" To avoid confusion during the interviews (and later survey), we clearly defined the focus of our study in terms of run-time configuration options.

Throughout the brainstorming, we leveraged the extensive industrial experience with configuration of two of the authors, as well as the results of our earlier work on software configurations [81], [82], [80]. To mitigate the risk of missing major questions or themes, we chose a semi-structured format for the interviews. Such a format helped us complete the base themes and questions by allowing interviewees to change the direction of the interview if interesting new topics are being covered. Such topics could then be integrated in our base interview guide for future interviews. The final interview guide is available online [1].

The 14 interviewees were recruited from different companies and have different software engineering roles, ranging from developers to software architects and managers, including one infrastructure (i.e., environment) architect. Each of these interviewees belongs to a different company, except for two interviewed experts who did work in the same company, but under different roles (manager and integrator). The 13 companies also cover a wide range of domains, from governments to banking systems and online retailers. As presented in Table 1, the 14 interviewees had between 5 and 28 years of software engineering experience, with only four people having less than 10 years of experience.

Each interview was performed by two of the authors, one of which was the interviewer and one of which the scribe taking notes. The interview would start from an open question, then

the interviewee's answers would help select the next question to move to. Although we made sure that a typical interview could be finished within one hour, the semi-structured interviews eventually had a length of one to three hours.

3.2 Card Sort Analysis of Interview Data

In the second step, two human raters (authors) qualitatively analyzed the interview results by following a card sort approach [78]. Card sorting is a categorization technique that is widely used in information processing to derive taxonomies from input data [78]. In our case, the two raters printed the interview transcripts and notes, then cut those up in snippets containing relevant information. These snippets were then classified collaboratively into clusters that represent a common topic, for example "testing configuration options". Once finished, the raters archived the classified snippets for two days before doing a second iteration of classification in which topics related to the same high-level configuration-related activity (e.g., "quality assurance") were clustered together.

The resulting clusters enabled us to identify 9 configuration-related activities, together with challenges and anecdotal evidence related to the activities. The other two authors then validated these results. The final results of the card sort are available online [11].

3.3 Survey

In order to extend our understanding of the configuration-related activities and challenges identified via the interviews, and to identify how representative they are, we conducted a large 20 minute survey containing 25 closed and 5 open-ended questions. To arrive at this final list of questions, we first identified 56 questions from the individual topics (card sort clusters) within the 9 obtained configuration activities. Two of the authors discussed and reduced this list to 46 questions, after which the two other authors took the survey to evaluate its clarity, duration and relevance to our study. After addressing the identified issues, we sent the survey to personal contacts (including a number of open source developers). The 9 answers that we received were then used to validate and improve our questionnaire. We re-organized the order of questions into 4 pages (instead of 12), revised two unclear questions and could further reduce the number of questions to 30. Our survey questionnaire is available online [13].

We then sent out 2,000 personalized emails inviting open source project developers to participate in our survey. These developers were selected from the 1,000 Java projects on GitHub with the most commits, as obtained from the GHTorrent [38] database. This selection strategy allowed us to focus on mature and active projects, ignoring Github repositories that are not software projects (e.g., repositories for courses or student exercises). We focused on Java projects to control for the impact of programming language on configuration options, and since it is the most popular programming language on the TIOBE index at the time of performing this study. For each selected repository, we selected a sample of contributors, whose email address we obtained via the Github API [5]. We manually checked our targeted developers before sending emails.

Eventually, we received 229 responses, yielding a response ratio of 11.5%, which is a reasonable percentage compared to surveys in other domains [7]. Note that our invitation email explicitly asked the surveyees to answer the survey from the point of view of the specific project for which we had contacted them. Furthermore, the 229 responses do not include the 9 responses used for validation of our questionnaire.

The work experience of the 229 survey participants varied from a minimum of 1 year to a maximum of 45 years, with a median of 11 years and average of 13.35 years. The majority of these participants are developers (184), while 82 are architects, 27 managers and 19 academics (students, professors or lecturers). Note that some participants had more than one role at their organization. 3% of the participants have contributed to projects with a huge number of configuration options (between 1,000 and 10,000), 20% to projects with a large number of configuration options (between 100 and 1,000), and 51% to projects with a median number of options (between 10 and 100). Only 26% of the participants had contributed to projects with less than 10 options, indicating that the majority of survey participants are indeed practitioners who have been exposed to software configuration and hence can offer a large variety of perspectives.

3.4 Card Sort Analysis of Survey Data

Closed survey questions were analyzed quantitatively, while open-ended questions were again analyzed using card sort analysis, similar to the interview data. Given the scale of the obtained survey data, we performed its card sort electronically, eventually obtaining clusters representing configuration-related challenges and clusters representing best practices to address challenges, each with a variety of sub-challenges or -practices, and anecdotal evidence [8], [9].

3.5 Systematic Literature Review

Finally, to identify which best practices in configuration engineering are not yet covered by earlier research as well as to guide practitioners to published approaches and solutions, we performed a "systematic literature review". The **objective of our literature review** is to explore the research literature that focuses on software configuration, then classify it based on the 9 configuration activities, and the challenges and practices identified during the interviews and survey. We follow the systematic literature survey process prescribed by Kitchenham et al. [54].

The approach used in our literature survey consists of obtaining papers studying run-time software configuration from the online Compendex, Inspec, ScienceDirect, and Web of Science databases. We obtained a set of 632 papers, which we obtained by using the following search criteria and queries:

- **Criterion 1**: In order to select papers that discuss configuration, we focused on those in which the title and abstract contain the keyword "config*", "misconfig*", or "mis-config*".
- **Criterion 2**: To eliminate papers that are not related to software engineering, we only retained papers whose venue title contains the keyword "Software" (like International Conference of Software Engineering).
- **Criterion 3**: To reduce the amount of papers to classify in the first iteration, we initially focused on papers published in the last 10 years (From 2007 to

2017). Later on, we used snowballing (see below) to cover papers earlier than 2007.
- **Criterion 4**: Papers should be written in English.

Inclusion and exclusion criteria, we manually studied these 632 papers by classifying them into two categories: (1) papers focusing on run-time software configuration, and (2) papers that are out of scope for our research (e.g., on compile-time software configuration or software product families). For this initial classification, we read each paper's title, abstract, introduction, and conclusion.

At the end of this iteration, we obtained 69 papers that are related to run-time software configuration. Afterwards, 6 more papers were eliminated as out of scope based on peer-review and discussion between authors, resulting in 63 papers.

We then added 7 extra papers that we already knew (bringing our tally to 70), but were missed by the criteria above. 5 out of these 7 were published in venues that do not contain the keyword "Software", while two papers were not in the databases that we selected (Compendex, Inspec, ScienceDirect, and Web of Science).

To cover a larger search space, we then performed **"snowballing"**, i.e., the identification of additional papers from the references of the 70 papers selected in the previous steps, repeated recursively from each of the newly selected papers' references. At the end of this iteration, 36 new papers were added to our dataset, yielding 106 papers related to software configurations.

3.6 Card Sort Analysis of the Systematic Literature Review Data

To complement the practitioners' recommendations with the state of the art discussed in academic literature, we classified the final set of 106 papers according to the 9 configuration activities that we identified via the interviews and survey. Each paper could cover more than 1 such activity. The goal of this classification across activities is to lead practitioners to approaches for their specific context, to confirm practitioners' recommendations with existing researchers' validations, and to identify configuration research areas that have been covered less in detail and hence should likely be the focus of future work.

In order to conduct the card sort analysis, we followed these steps:

1) Each author focused on a subset of the 106 papers that we obtained at the end of the snowballing activity.
2) Each author read each of his papers to deeply understand its focus and contributions.
3) Each of the papers was then classified according to one or more of the 9 activities identified during the interviews and survey. At the end of this step, each paper was either classified into zero, one or more categories based on its contributions.
4) After the classification, each of the authors reviewed the papers of another author to verify the classification. If two people disagreed about a classification, they together discussed it to arrive at a final decision.
5) Within each activity, we then matched papers with the corresponding challenges that they are addressing.

TABLE 2
Mapping of each survey question (in the Appendix) to the activity it addresses.

Activity	Question
A1	9, 10, 11, 12
A2	5
A3	12
A4	6, 7, 8
A5	19, 20, 23, 24, 25
A6	21, 22
A7	15, 16, 17, 18
A8	13
A9	14, 26, 27, 28

The next three sections present the card sort results regarding configuration engineering activities, challenges and best practices, each supported by quantitative and qualitative evidence.

4 Configuration Engineering Process

The first goal of this study is to identify the configuration engineering process used in a typical software project. This process basically consists of a set of activities performed routinely by different software engineering actors, including developers, architects, and managers. These activities were identified from the card sort analysis of the interviews, and were later confirmed by the survey (and implicitly by the literature survey). The resulting list of 9 activities (visualized in Figure 1) should be taken into account by any new developer or software project when estimating development or maintenance effort. We also hope that these activities will become the focus of future research by our community to help address the challenges presented in the next section.

Table 2 shows the mapping between the 9 activities to the survey questions that covered them. Note that the last two survey questions were open-ended, allowing the surveyed developers to express challenges and recommendations on any configuration activity.

A1.Design and Creation of Configuration Options The most obvious activity, which 92% of survey participants performed at least once, is to add a new configuration option to a software system. This involves choosing a name, a type (e.g., boolean or string), a default value, the physical location (storage medium) to store the option's value, and providing comments or other documentation. Most of the participants create a new option to either configure technical aspects (like the memory usage or database credentials) of an application or to configure the actual functionality of the application.

These options can be specified in the user requirements or architecture document, or can be "invented" by an architect or developer, for example to "externalize" a source code constant into a configurable option. Such externalization refers to the act of allowing the value of an option to be changed from outside the source code, hence avoiding recompilation.

A2.Managing Storage Medium The storage medium for configuration options, i.e., the physical location where options and their values are stored (and can be changed), is mostly determined by personal preference as well as by the intended "binding time" of an option. The binding time is the moment at which a configuration option gets its value

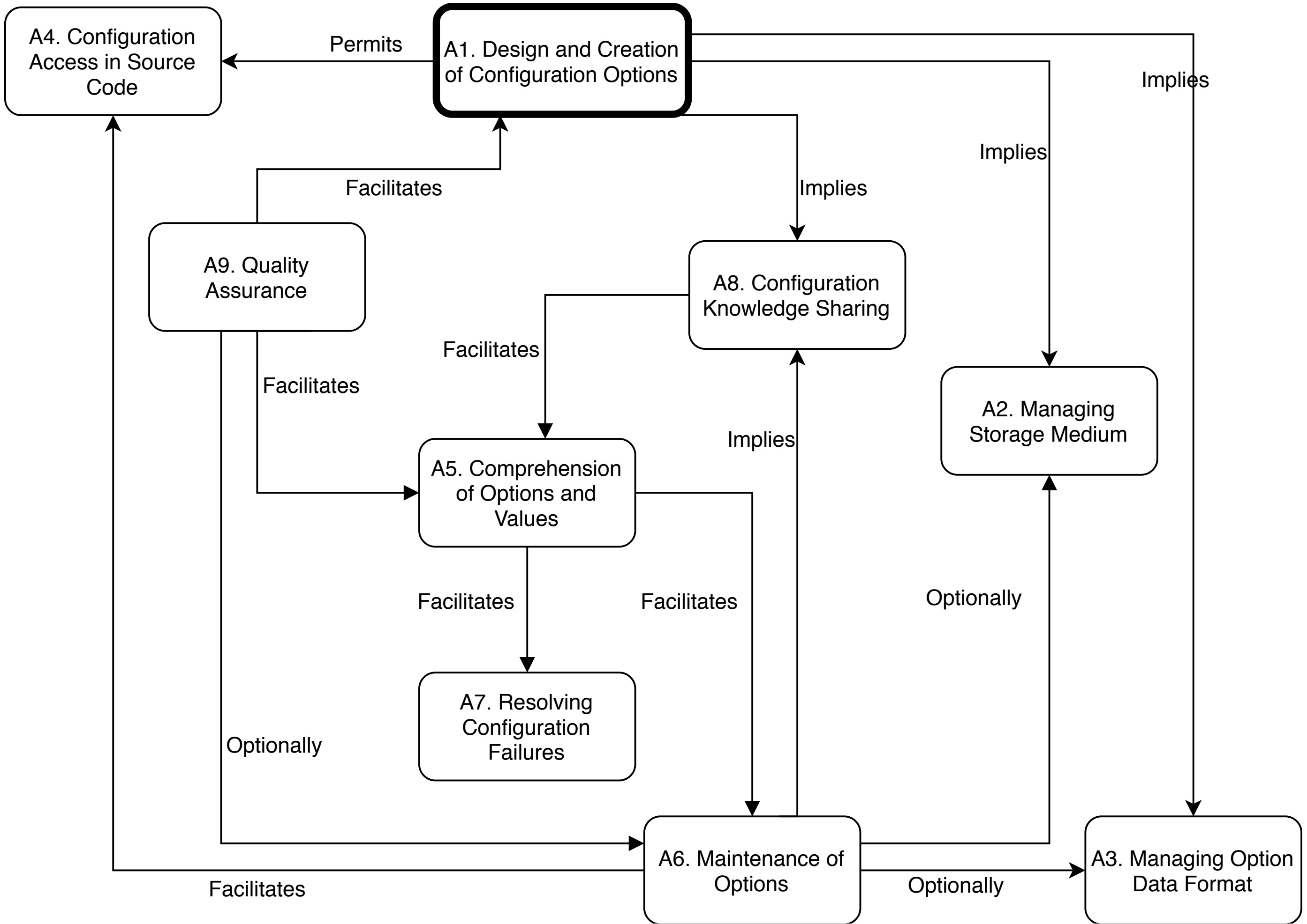


Fig. 1. The process of software configuration engineering.

assigned (bound) to it, as discussed in Section 2.3. For example, options stored as conditional compilation macros or as hard-coded source code variables are bound at compile-time (outside the scope of this study), while most configuration files and program arguments are bound at load-time. To obtain execution-time binding, options are typically stored in databases or (less often) in files. Sometimes more esoteric mechanisms are used, such as checking for the presence of a file to set a boolean option to true. Such a mechanism enables file system permissions to restrict access to an option.

Execution-time binding is used for options whose value could change dynamically (typically functional options), while load-time options are used for options that do not need to change after starting up an application, but do need to be changed by the end-user, like the amount of memory allocated to an application or the credentials for accessing a database. We found that 98% of the surveyed participants use load-time configuration options and 35% use execution-time options. Given the scope of our study (and our focus on Java projects for the survey), no compile-time option usage was recorded.

As shown in Figure 2 (based on question 5 of the survey), the top three most popular configuration storage media in our survey are *"properties"* files, *"environment variables"*, and *"XML"* files, respectively enticing 62% (143/229), 47%, and 45% of the surveyed participants.

The reason for listing the management of configuration storage medium of a software application as a separate activity is that it is not just a choice made during creation of an option (A1), but also involves maintaining or even changing the medium over time, for example by splitting up or merging configuration files, renaming them or even moving from files to databases.

A3.Managing Option Data Format A configuration system's data format (or schema) corresponds to the types of values allowed for each option, together with any constraints enforced on these values. Option types can range from pure strings (even to represent numbers or more complex structured data), to simple types distinguishing between string and numeric types, or complex types allowing more advanced, custom data formats.

A popular form of complex configuration format are hierarchical options, which use a nested data structure to store configuration values. Instead of having 5 separate options related to the same feature spread cross the configuration files, nested data structures allow to aggregate these options in a single, composite configuration option. In the survey, we noticed a more recent evolution from purely hierarchical option values towards graph-based structures.

Although the choice of data format is somewhat influenced by the choice of storage medium (a properties file cannot easily

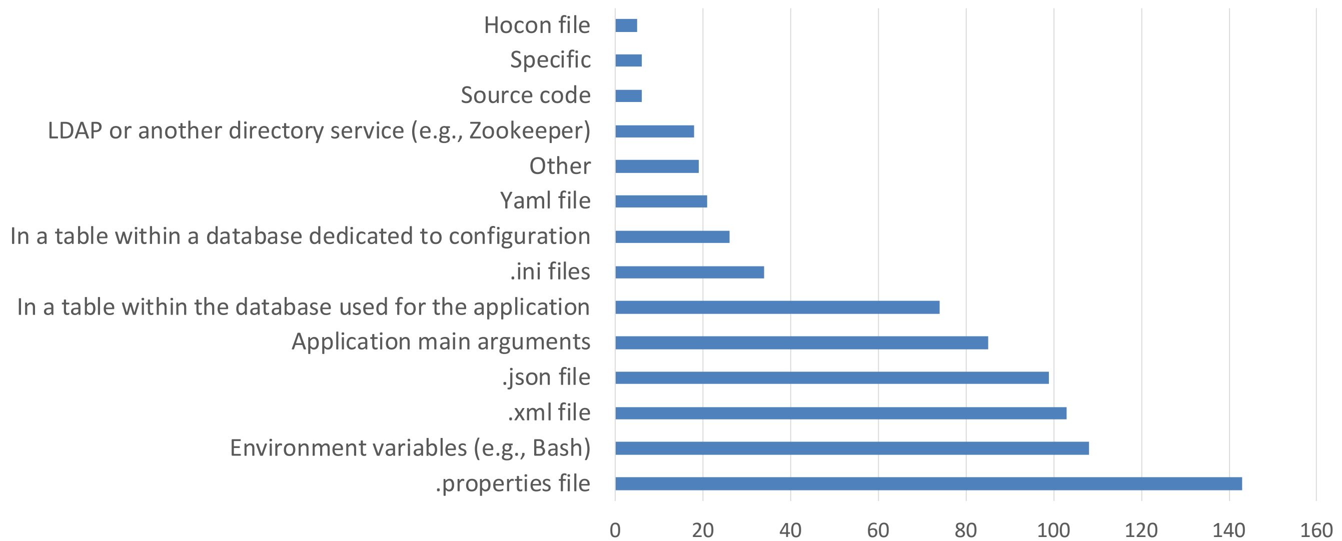


Fig. 2. Popularity of configuration storage media (based on survey question 5).

model hierarchical values), most storage media can handle any data format, to some degree. One of the main differences is the ability of a storage medium to automatically enforce constraints on the values. For example, XML schemas are able to check that an XML configuration file is valid, while properties files do not have such built-in support.

Similar to managing the configuration storage medium, managing the configuration data format covers more than just the initial choice of format, such as refactoring to another format as an application's features evolve, as well as refinement of configuration data constraints.

A4.Configuration Access in Source Code Once an option has been added, developers should access its value from within the source code in order to enable/disable features, to create connections or to perform specific calculations. To enable such access to the configuration options stored in the storage media, we identified three sets of approaches in use: using a third party configuration API (frameworks such as Java Preferences or Apache Commons), developing a custom API, or a hybrid approach with a custom API leveraging a third party API.

A5.Comprehension of Options and Values To use, manipulate, or maintain an option in the source code, end users and developers first need to understand the goal of that option and its possible values. The surveyed participants use a variety of artefacts to understand an option, ranging from comments in a configuration file, documentation of the source code, external documents, to readme files and manuals. Developers also rely on informal approaches to understand configuration by asking and discussing with their colleagues.

A6.Maintenance of Options Once an option is created, accessed and used in the source code, developers need to monitor its usage and usefulness over time, basically maintaining the option and the code that is using it. Maintenance activities can vary from renaming options and refactoring the code regions accessing them to removing redundant options. Although our interviewed and surveyed participants agree that maintenance of configuration options is important, they unanimously confirmed that it is not an easy task, as we discuss later. In fact, they consider configuration option maintenance to be the major area where tool support is lacking.

A7.Resolving Configuration Failures A major activity impacting developers and users of an application is to resolve failures that are caused by misconfigured options, i.e., options that were assigned an incorrect (default) value. This involves reproducing the failure to confirm that it is caused by a misconfiguration (and not a code-related fault), debugging the reproduced failure to identify the faulty configuration option and identifying the right value to fix the misconfiguration.

A8.Configuration Knowledge Sharing Sharing development knowledge between team members is a common activity in software engineering. As part of this knowledge, developers also need to share amongst each other important information regarding configuration options. This information ranges from data on newly added options to illustrations showing the impact of an option on source code regions, or metadata about common configuration failures and faults for tricky options. While activity A5 considers the understanding of an option from a configuration option user's perspective, activity A8 considers such understanding from the developers' perspective (e.g., for evolution and maintenance purposes).

A9.Quality Assurance Finally, similar to regular code development, quality assurance is a major activity for assuring the correctness and integrity of a project's configuration, with the goal of improving software configuration comprehension, improving the resilience of an application to configuration failures, reducing the software configuration's complexity, and improving its maintainability. For example, some projects developed static analysis tools to check the semantic constraints on the values of a configuration option (on top of the basic type checks of the configuration data format) or to test a system in the context of a given configuration. Traditional quality assurance activities like code review or tests equally apply to configuration options, their values and their access within the code.

Relations between Configuration Engineering Activities As shown in Figure 1, the configuration engineering process obviously starts by the creation of new configuration

options (A1). Creating a new option also implies deciding on a storage medium (A2) and on a data format (A3). These two last activities are separated from the creation of configuration (A1), as they are also impacted by the maintain of configuration options (A6). For exanple, developers can decide to change an option type from a string to a boolean, or to change the storage medium from a simple file to a more powerful database.

Once an option has been created, developers can start accessing it within the source code (A4). It is also important for the people who added the option to document and communicate the intent and constraints of the option to other developers (A8). This is for example important to facilitate the comprehension of options and future changes (A5).

On the other hand, effective quality assurance practices (A9), such as configuration-aware code review or the specification of naming conventions, can improve and facilitate the creation, maintenance and comprehension of an application's configuration system. The comprehension of options and their values (A5) is of fundamental importance for the maintenance of software configuration options (A6) and for debugging of their failures (A7).

The above relationships were derived during card sorting, in parallel with the identified activities. Most of these relations are rather trivial, while some of them require further scientific validations. For example, it is trivial that creating an option implies managing the option's storage medium. However, it is less clear that quality assurance (A9) facilitates comprehension of options (A5). The survey did not require respondents to evaluate the relationships.

5 Configuration Challenges

This section presents the challenges that we identified from card sort analysis of the 14 interviews and the 229 survey responses for each of the 9 activities. Figure 3 summarizes these challenges, ordered by the number of survey participants who mentioned them.

Although the interviews and survey address two different population categories (which are respectively industrial and open source Java engineers), we did not observe any inconsistent results between both categories (except C9.1). Therefore, we merged the results and discussions of both categories in the remainder of the paper. However, it might be worth to explore more deeply the differences between industrial and open source developers by replicating the survey in an industrial context.

5.1 Creation of Configuration Options

C1.1 Options are not Planned During the interviews, we learnt that in most of the cases there are no plans or guidelines on which options to add in the source code and where to add them, leading developers to mostly use their own judgement. Our survey (question 11) confirmed that 60% of the surveyees add options without any planning, while 40% specify options in the architecture specifications and 24% in the requirements (people could choose multiple answers). In 24% of the cases, new configuration options were meant to externalize a string constant from the source code, since developers typically *"hard code first and externalize later"*. Although our survey is sent to open source contributors, we observed a large number of participants mentioning architecture and specification documents. We think that this is due to the large projects we focused on, in which relying on such documents is a common practice. Furthermore, in practice open source developers, especially in larger projects, often do so in the context of a company who is paying them to represent their company's interests in a project's development.

Although six survey respondents in question 29 mentioned that careless addition of options can "*create [a] maintenance burden in the long term*" and "*They don't presume that others might need them*", the lack of option planning is not necessarily due to wrong developer intentions. For example, 7 respondents explained how it is difficult to decide what functionality should be made optional due to the lack of direct access to users or due to the users not knowing it either: "*The most important problem is knowing what options our users want and how to tailor those options to that audience. Often times we can't communicate with all of our users and that creates a very big gap of understanding between developers and our userbase.*" In the worst case (two survey responses) this can lead to the wrong data format being chosen for options: "*We should have anticipated nested configurations, but the config system sort of grew organically at first. Later we had to step back and redesign it.*"

C1.2 Adding Options Increases Complexity The interviewed experts and 14 survey respondents (in question 29) unanimously confirmed that a large number of options makes software configuration a more complicated task: "*There is no fun to create new options. Options are increasing the complexity*". Apart from complicating the configuration of a software system, it makes the meaning of options harder to understand and hence to maintain: "*I have bad experience with "configuration bloat" which means that sometimes there is a plethora of config option and you don't really need 99% for almost all usecases. In effect this means someone getting acquainted with the software will miss important config options because most of them can be safely ignored.*"

The risk of increasing complexity is even larger in practice as not only the number of configuration options in a file is growing, but "*[the number of] configuration files grows organically*" as well (5 respondents to question 29). In other words, the options are being spread out across more files, largely because developers (especially novices) "*do not immediately see the complexity they are adding by creating new options*" or configuration files, and because adding new options is largely an *ad hoc* decision that is not planned ahead of time (see above).

C1.3 Choosing the "Right" Default Value Even though options are added to enable end users to customize an application to their needs, 12 respondents stressed (in their responses to question 29) that it is essential to pick the right default values for each option. For 6 respondents the term "right" corresponds to enabling out-of-the-box functionality of the application for the majority of the end users: "*Many users never configure the program, and so choosing the best possible defaults for configuration options is important.*" For 4 others, the default values are even essential to avoid breaking the system: "*Graylog may destroy all your logs if the log retention module's configuration can not be loaded properly, because the default in the code is to have a very short retention period*". Unfortunately, all 99 people responding to question

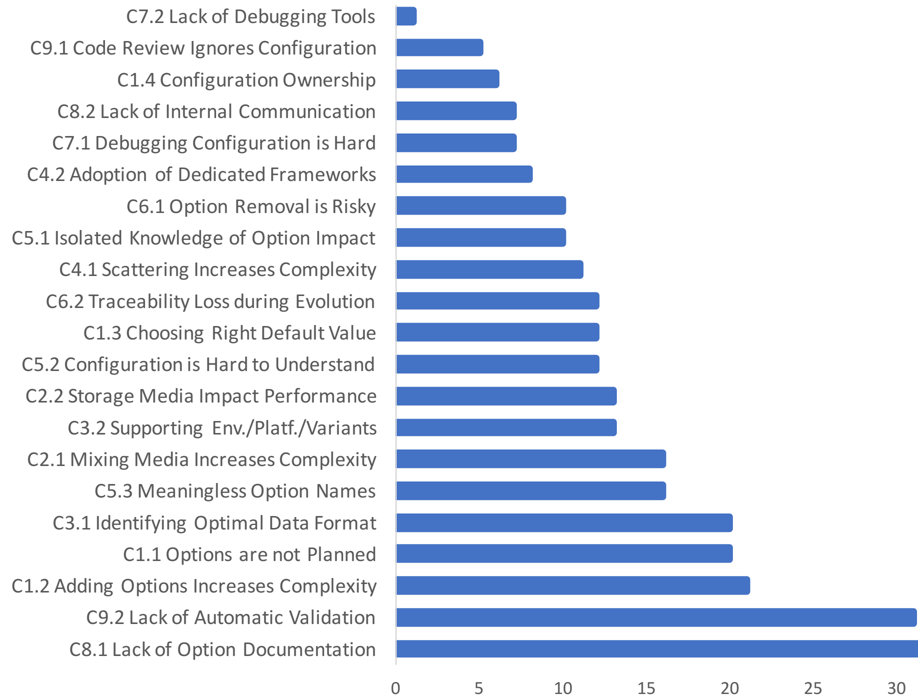


Fig. 3. The identified configuration-related challenges, ordered by the number of survey responses mentioning them (open-ended survey question 29).

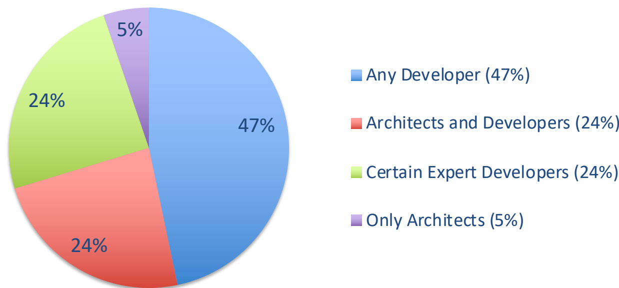


Fig. 4. Who is Responsible for Option Creation (survey question 9)?

12, and 12 respondents to question 29 concluded that finding the right default value is a largely *ad hoc* process.

C1.4 Configuration Ownership Figure 4 (question 9) shows how 47% of the survey respondents allow any developer to add new configuration options, while in 53% of the cases this is restricted to a group of experts, i.e., a group of developers (24%), architects (5%) or a mixture of developers and architects (24%). Especially in sensitive contexts like banking applications, architects are the main responsible party for configuration creation.

Allowing anyone to add options has several side effects. First, a configuration file can be filled with redundant configuration options configuring the same thing. Second, conflicts between a new option and existing ones generally do not pop up in a developer's individual workspace, but only during integration of the developer's changes with other developers' work: "*[they] had to merge all [modified] configuration files, and hence they got for some cases, more than one option that configure the same thing. This was complicated to fix and consider*". This is mostly because developers in general don't *"have a broad vision on different components and a long-lived experience on the system"* and because a configuration file is an external artefact that could be *"shared between different components and/or different applications"*.

5.2 Managing Storage Medium

C2.1 Mixing Storage Media Increases Complexity Having many different storage media in one software system increases the complexity of configuration options, and hence makes them harder to find, use, understand and maintain. One interviewed expert even considers software configuration as a *"neural network"*, because a potentially large number of options and configuration files is distributed across a variety of different storage formats, yet for the outside (i.e., end users and developers), they should look as one coherent set of options. Apart from confusion, other negative consequences include difficulties integrating the options from different storage media as well as updating the options in the right location.

Unfortunately, mixing storage media is quite commonplace if different options should have different binding time. Six respondents (question 29) also explained how a physical distinction needs to be made between confidential configuration data, such as passwords and secrets, and public configuration data. Hence, at least two storage media are required to achieve such a distinction. Similarly, in larger distributed systems different nodes might require their own configuration storage.

C2.2 Storage Media Impact Performance Surprisingly, there are also performance implications related to the choice of storage medium (and binding time). Seven respondents (question 29) mentioned how the choice of a storage medium with load-time binding time requires reloading the whole application just to refresh one option's value. Five

respondents experienced heavy load when using a configuration database with execution-time binding, as their system accessed the database too frequently to check for updated option values, to the point that any down-time of that database would cripple the system's behaviour. Finally, one respondent experienced race conditions, where an application would accidentally read an option's value right before changes would be made, hence missing the updated value.

5.3 Managing Option Data Format

C3.1 Identifying Optimal Data Format Identifying the right data format is a major configuration-related challenge, with 10 respondents (question 29) experiencing slowdown due to too weak option type systems, wrong types chosen for an option (e.g., string option not supporting special characters), inability to group related options or lack of support for adding comments to options and their possible values. Three respondents were unable to easily express hierarchical configuration data (i.e., configuration options that are composed of other sub-options), while four were unable to deal with graph or struct option values.

Finally, two people pointed out (question 29) the lack of reuse mechanism in existing data formats. Even though several configuration files could share common lines ("*Consider a configuration file that supplies values suitable for X region and Y environment, and another configuration file that supplies values suitable for Y environment and Z data center*"), one needs to duplicate those lines across all files. An *ad hoc* "include" mechanism seems a possibility, but is not straightforward to enable and maintain.

C3.2 Supporting Multiple Environment/Platforms/Variants As suggested by one of the respondents in the previous challenge, an application often needs to be deployed in multiple environments (development, test and production; 6 respondents), on multiple operating systems or on mobile platforms (3 respondents). Furthermore, different variants of a given application might need to be built, such as a cheap, light version vs. a more expensive, full-featured one (2 respondents). Having to provide a whole folder of configuration files, one per environment, platform and variant is to be avoided due to combinatorial explosion and because "*If you have different config files for each, you can hit problems when you release due to the production config being untested*". Finding a sufficiently expressive data format (and storage medium) avoiding such explosion, or at least making it more manageable, is a common challenge experienced by the surveyed participants.

5.4 Configuration Access in Source Code

C4.1 Scattered Access Increases Complexity 7 survey respondents claim (in question 29) that "*from the application perspective the location of the configuration values should not be visible or make a difference*", urging to build code abstractions to hide configuration storage media. If not, understanding configuration use, its maintenance and debugging become more difficult, while any changes in data format or storage mechanism risk propagating across the code base.

Despite 60% of the surveyed developers accessing configuration options from within a single class or module, 40% still read configuration options from multiple storage media in different places of the software system's code (question 8).

C4.2 Adoption of Dedicated Frameworks Similar to Sayagh et al. [81], the interviews and survey showed how developers typically do not tend to use existing configuration frameworks, or only opt for the most basic ones (e.g., the preferences API in the standard Java SDK). Yet, a large number of sophisticated frameworks exists to make accessing options cleaner, scale to large configuration spaces, improve type-safety or reduce memory footprint. It was suggested by one respondent that this is because *"developers are lazy to learn and try new configuration frameworks, they like to use the easiest method"*.

In any case, 66% of the surveyed developers created their own classes and functions to read and manipulate configuration options from configuration files, while only half (49%) of the surveyed developers had used an existing configuration framework in some project. The latter percentage is basically the same (48%) as that of developers using just a basic I/O library (instead of a dedicated configuration library) for reading configuration files (question 6).

5.5 Comprehension of Options

C5.1 Isolated Knowledge of Impact of Options Despite some of the interviewed projects using a dedicated configuration expert, no single expert was said to know the goal and impact of all the configuration options of her software system. *"Only the developer knows the meaning of options he created. Configuration file is like a black box for deployers (which are the configuration users)"*. The survey confirmed this (question 19): only 31% of developers know the impact of all configuration options, whereas 46% know the impact of the majority of configuration options, and 23% have only a limited knowledge about the impact of configuration options. The latter people know the impact of only a few options (12%), only the options used in the source code they worked on (8%), or do not know at all the impact of any configuration option (3%). As mentioned by a surveyed participant in question 29, isolated knowledge is a risk: "*The one [who] created that option has quit the team and that option [is] invoked in too many places of the code and hard to guess what it does*".

We also studied the relation between knowledge of options and the number of configuration options. Of the surveyed participants who had indicated that they understand all configuration options in their system, only 11% worked on a system with in between 100 and 1,000 options. The majority (41%) worked on a system with less than 10 options. Hence, the complexity of having more options (C1.2) plays a major role in terms of configuration knowledge. Future work needs to investigate on this direction to find the impact of different factors such as projects size and engineering roles on this challenge.

C5.2 Configuration is Hard to Understand We found that surveyed participants use different techniques to understand configuration options (Figure 5). Based on survey question 20, we found that 74% and 52% of developers use documentation or configuration file comments, respectively, to understand the role of a configuration option. Since documentation and comments are not always up-to-date or clear for everybody, 59% and 2% of developers have to use the source

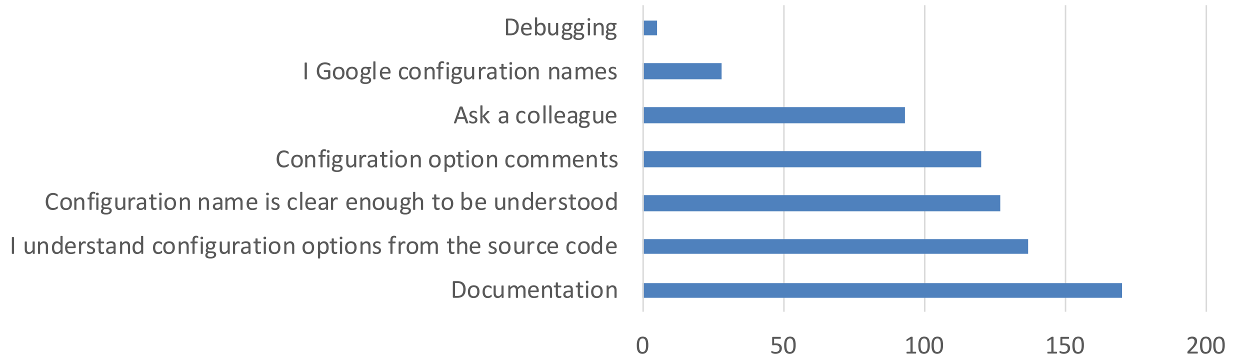


Fig. 5. Approaches to Understand Configuration Options (survey question 20).

code or debugging tools, respectively, to understand configuration options. Code search often is complicated because the option names do not appear as string literals or the option names are dynamically constructed via string concatenation. Instead, we found that 40% of developers ask a colleague for clarification about the goal of a configuration option and 12% of developers search online. Only 55% of developers rely on configuration names to understand the objective of a configuration option. Note that this survey question allowed respondents to choose multiple answers.

Despite all these sources, comprehension challenges remain an issue when trying to understand the options of third party libraries (3 respondents for question 29), the changes to options beween different versions of a library, or the interplay between different configuration options. For example, "*Having a stack of configurations, each one overriding parts of the previous ones. One needs to be very clear on the order of evaluation up front and remember to log the resolving.*" Options can depend on each other, override each other or interfere in other ways, all of which render configuration more complicated, especially as the number of options keeps on growing (C1.2).

C5.3 Meaningless Option Names 16 survey respondents highlighted (in their responses to question 29) the difficulty of finding meaningful names for options, where meaningful implies being short, descriptive and easy-to-search in the code base. In many projects, "*Many config options were added early on without any consistency*", but even to "*come up with an overall workable naming convention for all configuration options*" is not straightforward. Based on question 14, we found that only 54% of developers follow a naming convention for configuration options. Of the other developers, 23% have a naming convention but do not respect it, while 22% do not have a configuration naming convention at all.

5.6 Maintenance of Options

C6.1 Option Removal is Risky As shown in Figure 6 (and based on question 21), only 1% of the surveyed developers frequently change configuration options during maintainance, while 29% of the developers sporadically do so. On the other hand, 62% almost never make such changes and 8% never. As one interviewee put it (with 7 survey respondents making similar claims): "*Cleaning configuration options! Oh no way. No one can take such a risk, we don't clean the configuration files, because we don't know when and where the system can crash. Dead options are kept in the configuration files forever.*"

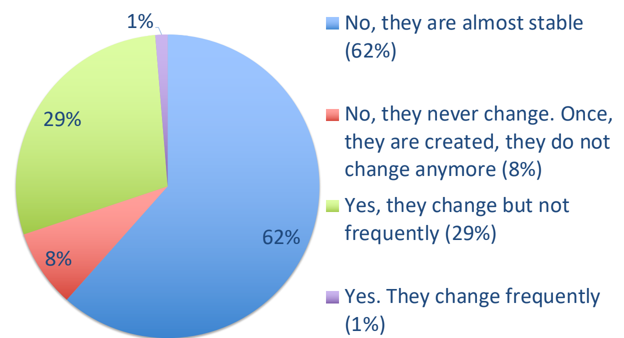


Fig. 6. How often configuration options are maintained by engineers (survey question 21).

Furthermore, for two interviewees, developers were not allowed to touch a configuration option due to the fact that these options were included in the official requirements specification, which corresponds to a contract with the client. As such, dead configuration options had to be left in the configuration file and/or in the user configuration UI. For most of the other interviewees, removal of dead options would only happen for options visible to the end user, while developer-only options typically are not a priority for removal.

C6.2 Traceability Loss during Option Evolution Based on question 22, we found that 58% of the survey respondents do not store removed configuration options anywhere, they just remove them from the configuration files. Only 4% use a version control system to track removed configuration options, while 3% store removed options in a database and 12% just comment out unused options in the configuration file.

A related challenge is the need for keeping new options backwards compatible with older ones (4 respondents to question 29): "*how to deal with introducing or removing configuration options in new software releases, considering that the release may have to be rolled back in case of a faulty release. If users have already started using new configuration options, the old release will not recognize them and error out*".

5.7 Resolving Configuration Failures

C7.1 Debugging Configuration Failures is Hard Via the interviews, we found that debugging configuration failures can

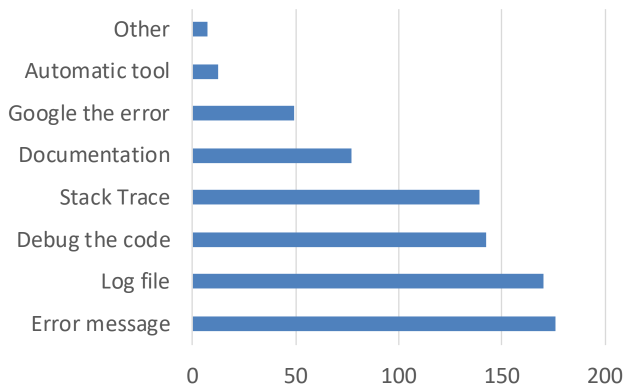


Fig. 7. Artefacts used to debug configuration failures (survey question 17).

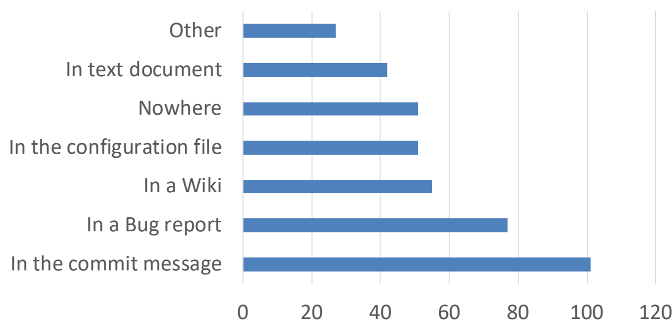


Fig. 8. How developers document configuration failures and their resolutions (survey question 18).

be difficult, especially in multi-component systems, or software systems that depend on other software components or services. For example, one of the interviewees lost considerable time trying to change an option hosted in another system, managed by another company. Unclear information about the values of configuration options further complicates debugging: "*error messages are horrible, they don't reflect anything about configurations*". 7 respondents (question 29) complained about execution logs not providing sufficiently detailed information to even determine if a failure is configuration-related or not: "Configuration options for things like timeouts retries or other connection options are quite useless without good monitoring".

C7.2 Lack of Debugging Tools The interviewed experts are debugging configuration failures with the same tools as they are using for debugging ordinary bugs. As shown in Figure 7 (and based on question 17), the most popular artefacts used to debug a misconfiguration are the failure message (76%), log files (74%), stack trace and the source code (62%). Only 5% of respondents use an automated tool to resolve misconfigured options. Generally, interviewees and survey respondents were not familiar with the many research tools for debugging configuration failures proposed in the literature [106], [66], [84], [32], [31], [71], [19], [20], [17], [118], [117], [119], [98], [92], [60], [52], [116], [101], [99], [101], [99], [82]. One respondent summarized the current state-of-the-practice as "*Dig deep and hack till it works*".

C7.3 No Strategies to Avoid Regressions As highlighted by Figure 8 (and based on question 18), the most popular artefacts in which developers record information about configuration failures and the workarounds used to resolve them are the commit message (44%), bug reports (33%), and wiki (24%). In addition, we found that 22% do not document configuration failures nor their solutions at all.

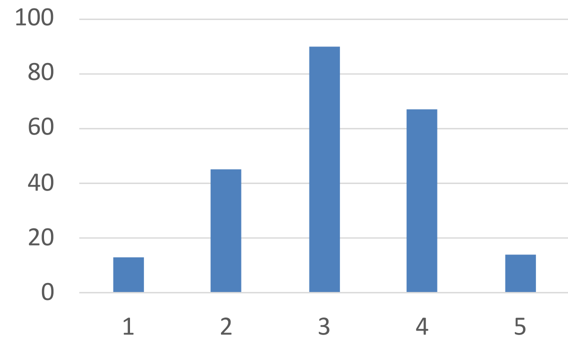


Fig. 9. Quality of documentation, rated from 1 (low) to 5 (high), based on survey question 23.

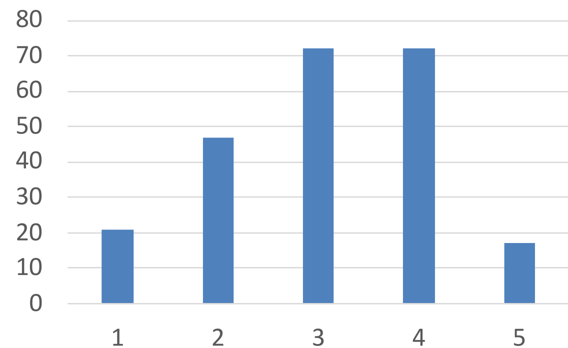


Fig. 10. Quality of configuration file comments, rated from 1 (low) to 5 (high), based on survey question 24.

On the other hand, in the case of projects who do document configuration errors and their resolution, surprisingly few people use that information. Most interviewees reported about developers losing precious time resolving the same configuration failures over and over again, without realizing that documentation was available about them. In other words, no clear process was in place supporting developers when resolving configuration failures.

5.8 Configuration Knowledge Sharing

C8.1 Lack of Option Documentation The interviewed experts mentioned that "*documentation and comments [(in the configuration file)] are rarely made for configuration options*", largely because of limited (time) budget and because developers do not like to spend additional effort on this. To confirm this, we asked survey respondents (questions 23, 24 and 25) to rate the quality of their option documentation and of the comments in their configuration files. As shown in Figure 9 and 10, 39% of respondents gave a rating of 3/5 for the documentation of configuration options, compared to 31% giving a rating of 3/5 for the quality of comments in their configuration file.

Hence, although developers know the impact of good documentation and comments on configuration, this documentation often either is missing or is incomplete. For example, the

presence of good documentation often depends on "*the writer context. The writing style of a documenter can influence the quality of the documentation*". 8 of the respondents provided examples of missing information, such as the goal and impact of an option, its links to user requirements, an explanation of its potential values, pointers to training/FAQ documentation or even an explicit domain model of the options.

C8.2 Lack of Internal Communication Finally, we also found that internal configuration-related communication amongst a development team is rather limited. As shown in Figure 11 (and based on question 13), 55% of respondents communicate newly created options via textual documentation, whereas 36% of respondents use the pull request as a way to communicate configuration options and 32% communicate new options via a wiki or a web platform. On the other hand, 25% of the respondents communicate new options only orally.

Such limited communication has as side effect that developers are not aware of existing options that could apply to their situation (5 respondents to question 29), leading them to add duplicate options: "*sometimes new people don't know that a feature already exists and can be deployed using just a configuration*". Furthermore (2 respondents to the same question), "*In addition to lack of discipline, it is often lack of knowledge in the first place. Chapters on Configuration management from books like Continuous Delivery would help and should be mandatory reading, I believe.*" Such books indeed provide techniques and practices to deal with secure vs. non-secure configuration data (either environment- or application-related), or to communicate between developers and operations (devops).

5.9 Quality Assurance

C9.1 Code Review Ignores Configuration The interviewed experts review configuration-related changes only in cases in which a configuration option will be manipulated by end-users, and ignore configuration that are changed by technical engineers like deployers or administrators. According to most of the interviewees, reviewers did not consider configuration options at all, as they either consider options as an external artefact or do not have enough knowledge about them: "*And we have all the routes in a configuration file, the problem was the configuration file was not being reviewed, so some developers were testing and changing the routes and pushing the changes without knowing.*"

Based on question 26, 36% of the survey respondents do not review all the configuration options: 15% review only their application's source code and completely ignore configuration options in their review, 8% do not follow any review process, they do not review neither their software source code nor the configuration options, 5% review only functional options, and 8% review only technical options. We found that 64% review both technical options and functional configuration options. This relatively large percentage seems largely due to the pull request mechanism provided by GitHub (since all survey respondents were identified via GitHub).

C9.2 Lack of Automated Validation Apart from code review, we asked the survey respondents (via question 27) which configuration quality assurance techniques in general they are using. As shown in Figure 12, 82% of developers mainly use tests, 19% have tools to validate the correctness of configuration values, while 17% use containers (like Docker) to configure the environment and application once then share the resulting container with other roles, avoiding costly (and error-prone) reconfiguration by everyone. In addition, 14% of respondents use a database to persist the entire history of configuration values and to have potential backups, while 5% do not follow any practice to assure the quality of their configuration options.

Offline, load-time and run-time validation tools generally are missing. For example, 10 respondents (question 29) mentioned the need for offline validation tools able to spot simple syntax or punctuation issues, and that could be integrated in the pre-commit hook of their Git repository. 2 respondents (same question) needed load-time validation of the semantics of the option values, while 6 mentioned similar run-time validation tools. The latter should be paired with adequate run-time handling of configuration errors, such as "*extensive bounds checking for configuration options, and storing a default and known-working configuration file inside the application, so that if values are missing, they can be replaced with the defaults. Some error output, letting the user know that they messed something up in the configuration file is helpful as well. Sometimes I've gone so far as to point out exactly where (down to the very character) a configuration error was found.*"

Finally, 12 respondents (question 29) mentioned the lack of testing tools of options, in particular tools allowing to test a code change across all relevant configurations instead of just for one option (value) at a time: "*The existence of options implies an additional testing burden, and even when it is possible in principle to test a given option (it does not require any special operating system, for example), often it is not tested, or not all combinations of interacting options are tested.*"

6 Expert Recommendations

While the previous section discusses configuration challenges faced by practitioners, this section provides recommendations by both open source and academic experts to address these challenges. They are obtained firstly from the interviewed and surveyed practitioners. In the two open questions 29 and 30, surveyed developers were invited to talk about a major configuration-related problem they faced in their experience and how they fixed it. They were also invited to recommend three good practices to follow and three bad practices to avoid when working with run-time software configuration in general. We enriched these practitioners' recommendations by academic approaches and findings identified during our systematic literature review (see Section 3).

A first goal of this section is to inform practitioners not only about how their colleagues have dealt with certain configuration challenges, but also about the state-of-the-art in academic literature. The second goal of this section (and the mapping in Table 3) is to emphasize areas that are not covered enough by literature, opening up avenues for validation of practitioners' recommendations along with new topics of research on promising technologies and methodologies to resolve important configuration challenges.

6.1 Creation of Configuration Options

R1.1 Treating Options as Scarce Resource In order to avoid having "*configuration files filled with useless informa-*

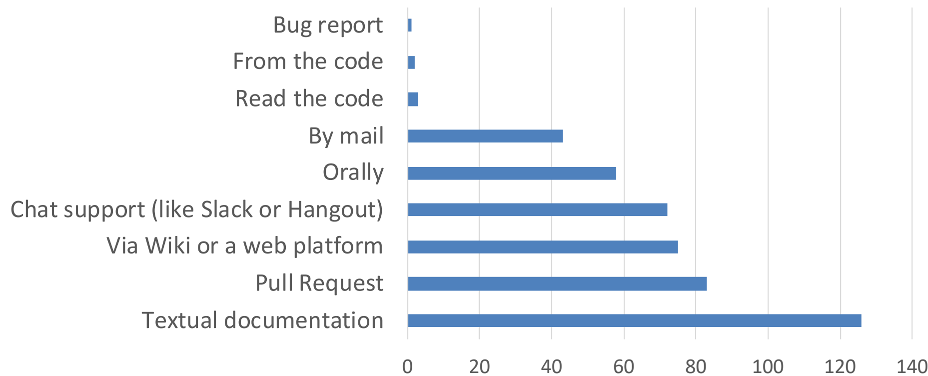


Fig. 11. Mechanisms used to communicate new options (based on survey question 13).

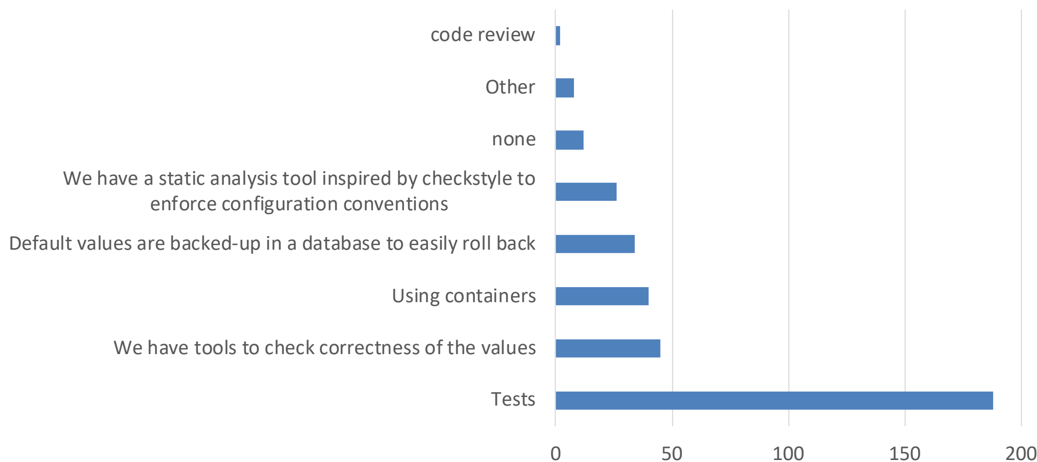


Fig. 12. Quality assurance techniques used by respondents (survey question 27).

tion", 66 survey respondents recommend to limit "*the number of options to an absolute minimum*". This is also discussed in the literature by Xu et al. [112], who found that end users change just a small portion of a software system's available configuration options, and hence developers should improve the design of their software system's configuration by presenting end users a minimal set of options to configure.

Developers should try to avoid adding new options, especially if they can "*determine the proper value by calculation*". Options should only be added if "*a major use case is unrealizable without one*". A second surveyed developer confirmed that one should use a planned and defined strategy to create new options: "*every configuration option has to be supported by a use case. Even [then,] if you do not have [an explicit] planning, if you can not think of a valid use case that the configuration option will [fix] that can not be [fixed better] by another solution, do not add it*". One should "*plan in advance the structure of the configuration, discuss it with colleagues, listing pro and [cons] of each approach and compare; ponder on what is gained with the new configuration; agree on what configurations will be needed in architectural/design phase*".

Reducing the number of configuration options is also important from the end users' point of view: "*Think about who is going to use the software, how they will want to use it, and how many ways it will be deployed. Does it need to just be for advanced users that know what they are doing? If so, it should be flexible with lots of configuration with sensible defaults with excellent documentation of the options. Will it be a consumer product that should work in most cases? In that case, think much more carefully about what you make configurable because you'll have to support misconfigured deployments and you don't want to confuse customers*".

R1.2 Assigning Option Ownership Apart from reducing the complexity of software configuration by limiting the number of configuration options and files, two of the interviewees stressed that they also limited the number of developers allowed to touch configuration options and files, even to only one developer in some cases. Any other developer willing to add, modify, or remove a configuration option should get the responsible developer's approbation. This could, for example, improve the names chosen for options, positively impacting challenge C5.3, as well as reduce the number of redundant and unnecessary options (challenge C6.1). Limiting the number of people allowed to manage configuration options is also suggested by 5 surveyed developers (question 30).

At the flipside, if one of the few responsibles leaves or would not formally document his or her options (cf. challenge C5.1), the recommended practice of clear ownership backfires. This is why Table 3 shows a negative impact of R1.2 on the C5.1 challenge. Therefore, it is important to trust the right developers for this task. One mitigation strategy presented by an interviewed expert is to follow a *"pair programming"* practice, where configuration changes are always made by two developers. Apart from increasing quality assurance, this practice

TABLE 3
Mapping the challenges (C1.1 to C9.2) to the recommendations (R1.1 to R9.5). The symbols + and - respectively indicate positive and negative impact of a recommendation on a given challenge (i.e, recommendation R1.1 positively addresses the challenges C1.1, C1.2, C5.2, and C6.1).

	R1.1	R1.2	R1.3	R1.4	R2.1	R2.2	R3.1	R3.2	R4.1	R4.2	R5.1	R5.2	R6.1	R6.2	R7.1	R7.2	R7.3	R8.1	R8.2	R9.1	R9.2	R9.3	R9.4	R9.5
C1.1	+										+							+						
C1.2	+	+	+										+									+		+
C1.3				+																+	+	+		
C1.4		+																+						
C2.1					+	+		-+		+												+		
C2.2					+																	+		
C3.1							+	+		+												+		
C3.2					+					+	+									+	+		+	+
C4.1							+		+			+	+									+		
C4.2										+								+						
C5.1		-							+		+	+	+	+				+	+					+
C5.2	+		+		+	+	+			-+	+	+	+	+	+			+	+			+		+
C5.3		+	+	+	+						+			+				+				+		
C6.1	+	+	+		+				+	+	+	+	+	+						+	+	+		
C6.2					+				+		+			+		+		+						
C7.1			+		+		+		+		+	+	+		+	+	+			+	+			
C7.2															+		+			+	+			
C7.3														+										+
C8.1				+							+		+			+		+	+					
C8.2		+												+		+		+						
C9.1		+							+		+									+		+		
C9.2									+	+	+				+					+			+	

automatically ensures that whenever one of the developers leaves, at least the second developer is still around to take over (and start pair programming with a third developer).

R1.3 Making Configuration Cohesive 8 survey respondents suggest that developers should "*make each configuration option responsible for one single thing, do not reuse the same option for different cases"*. In addition, they should never "*create multiple options that have the same role. [This problem is] not likely to happen in a small/new project but it could be the case in big projects"*. Therefore, a project team needs to *"keep all the configurations in check"*.

As potential side effects, it will be easier to define an explicit name for an option that has only one precise and explicit purpose. This is why we mapped R1.3 as a recommendation for C5.3 in Table 3. Furthermore, when an option is made for one explicit goal, its impact on the source code is minimized, which simplifies the removal of an option (challenge C6.1).

R1.4 Selecting Out-of-the-box Default Values 18 surveyed developers agree that one should choose "*proper default values*", which makes a software system work correctly from the first run: developers should never "*expect users to modify configuration for a first run*". This is mainly because "*a lot of [...] users will not change options and all of them will expect your software to work well with the defaults*". Furthermore, it might require quite some effort for them to find the optimal configuration for their deployment.

While these practitioner suggestions are not as concrete, researchers have explored this topic in far more detail. A large body of work tries to model the configuration of an application, then use this model to suggest an optimal configuration, for example using non-linear regression [40], Markov decision processes [25] or Plackett & Burman's statistical approach [27]. Other work [55], [86], [107] formulates the selection of configuration values as an optimization problem for which iterative search [55], multi-objective optimization [86] or smart hill climbing [107] can be used. While the above models all are offline models, Dia et al. [29] proposed an agent that automatically adjusts configuration option values of a generic application at run-time to guarantee CPU and memory usage objectives.

To improve the scalability of the above models and optimization algorithms, a lot of work has explored how to reduce the search space of configuration values. While a boolean configuration option by itself only has 2 possible values, a combination of 5 such options leads to 32 different combinations, which only gets worse when options have an integer or even string type. Thonangi et al. [95] proposed an adaptive search algorithm to minimize the space of configurations to explore in order to find an optimal configuration, while Siegmund et al. [85] use a threshold-based approach, which consists of detecting interacting options that significantly improve an application's performance.

A more common technique for reducing the search space are sampling algorithms, which select a reasonable subset of configurations, either to build a model or as a more targeted starting point for optimization. Osogami et al. have proposed different sampling heuristics [64], [65], while Sarkar et al. [79] used progressive and projective sampling. Progressive samping starts with a small sample set of configurations and each of their associated measurements for building a prediction model, then progressively increases its size until reaching an optimal sample size and prediction performance. Projective sampling starts by generating a set of initial points in the prediction learning curve from which it projects the optimal sample size to build an accurate optimal configuration prediction model. Two heuristics were proposed by Sarkar et al. [79] to improve the projective sampling, i.e., t-way feature coverage (changing t option values at a time for each test) and feature frequencies (each feature should be selected at least once in the initial sample). Duan et al. [33] used an Adaptive Sampling algorithm that selects experiments to run in order to find an optimal configuration. Zheng et al. [122] proposed MassConf, which collects a software user's configuration and environment information, then uses it to propose an optimal configuration for that user.

Only one research paper provides explicit guidance on selecting configuration values. However, its guidelines are specific to the configuration of garbage collectors in programming languages. Gousios et al. [39] found that such garbage collector configurations have a substantial impact on the performance of server applications. One of their recommendations is to calculate an application's memory allocation rate

and its object sizes to adequately configure the Java garbage collector.

6.2 Managing Storage Medium

R2.1 Minimizing the Number of Storage Media In addition to reducing the number of configuration options, 31 surveyed developers believe that it is also important to minimize the number of configuration files and mechanisms. Developers should indeed "*consolidate configuration options in as few files as possible*" and should avoid having "*too many floating configuration files*". Several surveyed participants recommend to create "*a central file with all default options*".

Minimization is especially important when considering that projects typically have different variants of their configuration file, one per environment (e.g., production, test or staging). The less configuration files, the less variants of these files there will be, hence minimization favourably impacts challenge C3.2 in Table 3. Similarly, a minimal number of storage media makes traceability of those media and their values easier (challenge C6.2).

On the other hand, one interviewed expert mentioned that there is no best choice for configuration storage medium. He suggested that a team should carefully discuss which medium should be adopted in advance. This is not only by considering the developers' preferences, but also by considering the existing technologies for reading and manipulating options.

R2.2 Organizing Configuration Options 20 surveyed developers suggested to clearly organize configuration options according to a uniform abstraction. One should "*avoid mixing 3rd party configuration values with application specific configuration*" in the same configuration file. However, organizations should not overdo it. They should avoid using "*a domain specific language for configuration [...], don't make your user learn another language just to configure your application*", just "*use a standard [file] format*". This is indeed a popular topic discussed by practitioners on Stack Overflow [3].

To abstract away from the specific storage medium (and data format) used, several software systems provide configurators, which are tools that help users easily configure their software system. However, such configurators typically are built *ad hoc* and suffer from a low quality. Therefore, Perrouin et al. [67], Abbasi et al. [14], and Boucher et al. [23] proposed approaches to reverse-engineer these configurators into a model of the configuration's data format (i.e., the configuration options with their types and constraints), then generate more consistent configurators from that model. Behrang et al. [21] presented an approach to find inconsistencies between user interface configurators and source code via static analysis. Evaluation of their approach on Mozilla Core and Firefox was able to find 40 real inconsistencies.

6.3 Managing Option Data Format

R3.1 Using Simple Option Types 9 developers recommend to simplify the types of configuration options as much as possible in order to improve the understandability of these options (challenge C4.1). Boolean configuration options are indeed much easier to configure compared to string configuration values. Based on empirical analysis, Xu et al. [112] recommend developers to simplify their configuration option types by explicitly identifying the configuration values used the most by their users. Similarly, developers should improve the error messages generated upon "*typing errors*", since it is "*not always clear if values are integers/floats/strings and can result in problems*". In addition to defining simple data types, developers should "*be prepared for weird configuration values from users*".

Configuration option types are not limited to generic types like boolean and integer, but typically include more specific types like IP addresses or file system paths. Identifying such types is important to easily configure a software system and prevent errors, but this is not straightforward. One approach to identify such types is proposed by Xu et al. [114] based on a classification tree and a keyword-based method. The latter method exploits the naming convention used by configuration mechanisms to deduce the semantics of an option. Xu et al.'s keyword dictionary has been built by manually analyzing 1,000 options of real open source projects.

R3.2 Identifying Sensitive Configuration Data While one should select good default values to enable key-in-hand software deployment, one should not ignore the security aspects of a configuration. As suggested by 4 surveyed developers, sensitive configuration data should be secured: one should "*not store passwords and other sensitive parts of configuration in VCS*" (Version Control System). In addition, developers should find a way to "*mask sensitive data and securing it in production*". While using separate storage media enables more secure, encrypted storage of sensitive configuration options, it does increase the number of media to look at in order to understand a system's configuration. This is why we mapped R3.2 to C2.1 in Table 3 as "-+".

To enforce the security aspect of sensitive configuration data, Sun et al. [93] proposed to automatically generate access control configuration based on access control requirements. In other words, this approach goes from a specification that indicates which role could perform which operation and under which conditions, and then generates access control configuration of J2EE applications. In addition, Wang et al. [100] proposed an approach to identify and reverse engineer access control configuration options, which allow or deny users from accessing a resource, e.g., a file, folder, or URL. Their approach consists of analyzing configuration files to extract configuration options, and uses taint analysis on the source code to understand how it interprets configuration options, and in which order it verifies the value of each these options. The goal of these analyses then is to generate a set of rules that define the relation between users, permissions, and resources. These rules can then be used to detect security policies defined in a configuration file.

6.4 Configuration Access in Source Code

Once developers decide to create a new configuration option and add it to a storage medium (e.g., a configuration file), they need to use this new option in the source code. Below, we provide a set of recommendations about how to use options in the source code.

R4.1 Encapsulation of Configuration Access in Single API 27 surveyed developers discussed how to use and access configuration options in the source code. The major recommendation from these discussions is to "*prevent reading of options from too many places in the code*" and hence to avoid

accessing directly the configuration options from all over the code base. In addition, surveyed developers suggest to "*use a single strategy [a]cross the application*" to read options, and hence not to mix-and-match different APIs for doing so.

In particular, surveyed developers suggest to "*try to centralize the configuration in an API*", which should not be "*too verbose (e.g. Config.getInstance(getValue(Config.KeyType.X, ConfigKey.ValueType.STRING, default_value) because it's a pain to read*". Developers should "*make sure the configuration API is well understood*". Apart from making it easier for developers to access configuration values, a specific API also limits the number of methods in which load- and run-time validation of configuration options can be put (challenge C9.2 in Table 3).

R4.2 Adopt Existing Configuration Frameworks 13 developers suggest to use an existing configuration framework or library instead of reinventing the wheel by making a new framework or API. A good framework can help developers abstract away from low-level configuration option access, in the sense that the framework can automatically identify where an option is defined and stored (challenge C2.1). Furthermore, a framework that is easy to use can help developers in navigating the source code and hence understand the impact of an option in the source code (challenge C5.2). For this reason, some developers specifically suggest to use a framework leveraging annotations and configuration value injection, which reads configuration option values then injects them in the right source code variables and attributes. Various developers explicitly stressed that developers should take the time to "*familiarize themselves with the mechanisms made available by the framework so that they are able to select the most appropriate mechanism*".

While configuration frameworks are recommended by practitioners, the choice of a specific framework is not straightforward, since there are dozens of (open source) frameworks on the market. Denisov et al. [28] compared three major Java configuration frameworks, while Sayagh et al. [81] analyzed the popularity of 11 Java configuration frameworks and the important factors to consider in choosing a suitable configuration framework. These studies identify a (potentially large) set of configuration framework characteristics to consider in choosing a suitable configuration framework, including how actively a configuration framework is maintained by its developers and how well it is documented.

6.5 Comprehension of Options

R5.1 Explicit Option Naming Convention 67 developers think that having a good configuration name is essential to improve the quality of a software configuration. An option name should be sufficiently "*clear and descriptive*" to be understood by end users. In addition, configuration options should be "*structured [...] from the start, probably split them by responsibility/modules/plugins/etc - whatever fit[s] your project best*".

Explicit naming convention could also include an indication about the environment used for that option (challenge C3.2). For example, one can distinguish between the environment of two options from their names like "*development.database.username*" and "*production.database.username*", where the first option is for the development environment and the second one concerns the production environment. Similarly, the configuration name can provide an indication of the plugin or component it covers, making it easier for developers to identify where an option can be used and for what feature (challenge C5.1), but also to remove an option entirely from the code base once it is no longer deemed useful (challenge C6.1).

R5.2 Comprehension from Code For challenge C5.2 in Section 5.5, we found that 52% of developers try to understand the meaning of configuration options by perusing the source code. Without automation, such code analysis is considered to be a tedious job. For this reason, several techniques are proposed in the literature to automatically map configuration options to the source code that they impact. Lillack et al. [56] used taint analysis to track under which Android configuration option(s) a code fragment could be executed. The authors found that their approach works especially well for options related to network and storage, since those options rarely interact with each other. Identifying which options control a source code area can help developers identify options that are rarely accessed in the source code, in order to perform cleaning and refactoring (C6.1). In addition, one could use the approach of Siegmund et al. [84] or Zhang et al. [121] to identify which options have what impact on software performance, which can help on debugging performance configuration errors (C7.1).

6.6 Maintenance of Options

R6.1 Pro-active Dead Option Detection 26 surveyed developers believe that one should "*clean out obsolete configuration immediately otherwise it hangs around after everyone has forgotten what it was for in the first place*". In effect, dead configuration options are considered as a form of technical debt in a code base, similar to the observations made by Rahman et al. [74] about feature toggle maintenance in Google Chrome. Those are options that are used during development to enable new features on demand for testing and release, and that should be removed from the code base once the features are stable (making them permanently on).

Note that, apart from dead options, unused options are undesirable as well and should be actively maintained. The latter options are options that are being used (as opposed to dead options), but whose value is not changed often by users. Hence, such options should be "internalized" again, i.e., be turned into a constant value.

R6.2 Limit and Trace Configuration Changes While cleaning and refactoring options is highly recommended, developers should do so with care, as configuration options are sensitive elements of a code base that are hard to test. Developers indeed should not "*change the name of a configuration key unless what it does is changing*", and especially "*between two major versions*". As suggested by one developer, it is important to "*not change configuration without understanding what it does and ever ask an experienced programmer what to do*".

If there is no other choice, and an invasive change needs to be made that risks to break compatibility, developers should explicitly manage the traceability of the configuration options involved. As suggested by 9 surveyed participants, developers should "*Keep track of what [they] did*" and "*keep working versions for rollback*".

6.7 Resolving Configuration Failures

R7.1 Right Granularity of Execution Logs 11 surveyed developers suggest to log configuration information and generate meaningful error log messages in case of configuration failures. Developers should "*log when/from where a configuration file is used*". In case of configuration failures, a software system should fail appropriately "*if a configuration is done wrong, or a configuration value is [missing]*". In particular, a software system should not "*silently fail or give an ambiguous error message about the bad configuration. The more details the user can be given, the better*". A suggestion for an error message could be: "*this failed because setting X was bad. Change the setting X in file Y to value Z to make it work again*".

Zhang et al. [120] proposed ConfDiagDetector, which injects configuration faults in a software system to analyze (using natural language processing) the quality of how the resulting failures are being reported or logged. This can be used to help practitioners evaluate the granularity and quality of their logs. Better error logs also help build more powerful configuration debugging tools (challenge C7.2).

R7.2 Document Configuration Failures and Resolutions To avoid regression of configuration failures (i.e., the same configuration failure re-occurring), 4 interviewed experts mentioned that documenting a configuration error and the way it was resolved previously is mandatory. Such documentation does not necessarily need to be formal, since several developers recommended to document configuration failures and their fixes in a project's wiki.

Documenting configuration failures is also important to build up a sufficiently rich data set for building models that can predict whether a bug report is related to configuration. Such models can reduce debugging effort by focusing on configuration options only instead of on the entire source code. Examples of such models are proposed and evaluated by Xia et al. [108] and Bowen et al. [24], which focus on predicting if a bug report is related to configuration. In addition to predicting if a bug report is related to configuration, Wen et al. [105] also predict which option is misconfigured.

R7.3 Automated Configuration Failure Debugging A wealth of research results are available on techniques (highlighted in bold) for various aspects of automated configuration failure debugging. In other words, this is by far the most researched configuration challenge/activity. Despite this, most of these approaches are not known or used in practice according to the interviewed and surveyed engineers. Here, we discuss the papers that are the most closely related to the configuration activities and challenges. We refer elsewhere [96] for a more detailed and technical survey on such debugging approaches.

Whitaker et al. [106] proposed one of the first approaches to debug configuration failures, which aims to **identify the moment on which a software system changes from a working to a non-working state**. Once this transition is identified, the approach checks all modifications made to the initial (working) state to find the culprit option. Similarly, Otsuka et al. [66] compare each configuration value with its past (in)correct values by differencing configuration values before and after a failure. Siegmund et al. [84] build models that describe the impact of a configuration option on software performance. Such models can be used to narrow down the scope of options to debug in case of performance bugs.

Other research efforts use **static or dynamic source code analysis** to identify the root cause of a misconfiguration. Dong et al. [32], [31] use a backward slicing technique that starts from a failing line in the source code and a forward slicing that starts from the lines reading configuration options. When both slicing techniques meet, the overlapping code paths are analyzed for configuration options. Rabkin et al. [71] use static data flow analysis to map each source code line to the configuration options that may impact it.

Attariyan et al. [19], [20] use **dynamic control and data flow analysis** to identify the root cause of configuration failures. In other work [17], they assign a performance cost to each source code block, then use (dynamic) taint tracking to detect configuration causes of performance failures. Zhang et al. [118], [117] instead profile the misconfigured system, then compare the execution trace with a set of profiles of correct execution runs. Zhang et al. [119] use dynamic profiling to instrument two software versions and execute both, then statically compare the two execution traces to find the culprit option. Attariyan et al. [18] proposed an approach to detect configuration failures by using a healthy execution environment (no configuration failure) as an oracle of correct execution.

Instead of analyzing only the source code, Wang et al. [98] proposed an approach that relies on **user feedback**. The approach takes a failure as input, proposes a fix to that failure from an initial set of sorted options obtained from an existing approach (i.e, those discussed previously), gets feedback from the user to understand whether that fix actually resolved the failure, then adjusts that fix's priority based on the user's feedback. Su et al.'s AutoBash [92] also relies on existing user experience to fix configuration failures. It observes the actions followed by a user to fix a configuration failure, including the actions made to fix a failure and the tests executed to verify that fix's effectiveness. AutoBash then saves a collection of solutions (actions and tests) from different users, and tries them one by one until successful.

While reliable configuration failure data (e.g., logs) are essential to debug configuration failures, **having too much data actually complicates the debugging process**. To easily debug configuration failures in the presence of huge amounts of trace data, i.e., execution traces of system files, registry, and process operations collected from different users of a system, Mickens et al. [60] proposed an approach that relies on decision trees. From a trace of configuration actions (reading or writing files and registry entries on a Windows system), the authors train a decision tree in which each node represents a file or registry key and the edges (or the decisions) decide to read or not a particular option. The leafs of the decision tree represent software exit codes, allowing the user to understand the circumstances leading to a specific failure message.

Debugging of registry configuration failures of Windows applications is another popular research topic. Kiciman et al. [52] proposed an approach to use existing snapshots of correct registry configurations to automatically recover (correct) constraints between registry configuration keys. Violations of the recovered constraints then indicate the presence of configuration failures. Yuan et al. [116] instead generate a database of rules from the successive events in

traces from registry access, which can be used to detect inconsistent events and violations that can identify the cause of a configuration failure. Wang et al. [101], [99] instrument a Windows application to find which registry key it is using, then compares those against a database of existing configuration snapshots. Finally, Strider and PeerPressure follow a trial-and-error approach to identify (and fix) a misconfigured option. While Strider [101] requires manual identification of incorrectly configured ("sick") machines, PeerPressure [99] relies on Bayesian estimation to identify sick machines and snapshots.

While most of the above work considers configuration failures in a specific component or layer of a software deployment, or considers the whole system as a black box, a more recent area of configuration failure debugging research started focusing on **failures that cross the boundaries of software components and stack layers** [49], [103], [80], [82]. Jin et al. [49] found that modern software systems are developed with multiple programming languages and hence need configuration tools that help debug configuration across those languages. Chen et al. [103] found that to resolve cross-component configuration failures, researchers should understand how different components communicate with each other and which configuration options of different components configure the same aspect and hence can cause interferences. For example, both the PHP configuration option *mysql.max_persistent* and the MySQL option *max_connections* configure the amount of allowed connections to a MySQL database, yet it is hard to debug the configuration of both systems to find the culprit of a global configuration failure.

Sayagh et al. [80] analyzed the relation between different layers of the WordPress application stack, and found that an option in one layer could have a substantial impact on other layers. For example, up to 85.16% of WordPress configuration options are used by at least two different WordPress plugins (layer on top of WordPress layer), which can cause serious inconsistencies between different plugins when such "shared" options are changed in incompatible ways. The authors also found that 45% of cross-stack configuration failures are responsible for crashes in production and that debugging such failures requires at least as much effort as single-layer configuration failures. They proposed a modular source code analysis to help debug such failures [82], which basically fuses slicing graphs of individual layers by mapping function bodies to their corresponding calls from higher layers.

To complement the above automated debugging analyses, a number of researchers performed qualitative analysis on configuration failures to **extract recommendations for developers and users to avoid such failures** in the first place. Yin et al. [115] conducted a comprehensive study on the characteristics of configuration failures including their causes, types, impacts, and how software systems react facing a configuration failure. For example, they found that up to 53.7% of configuration failures are caused by options that violate a predefined format, motivating the development of configuration checkers that verify how well configuration options respect such formats. Similarly, Arshad et al. [16] studied configuration failures in the GlassFish and JBoss Java EE servers, their types, when they occur, and their manifestation (silent vs. non-silent failures). They found that 89% of configuration failures manifest silently and 91% of these failures require a source code modification. They also proposed "*ConfInject*", an approach and tool to inject configuration faults in a software system to evaluate how resilient it is. They used "*ConfInject*" to compare the resilience of GlassFish and JBoss web servers, and found that JBoss performs better than GlassFish. Han et al. [42] found that 59% of 193 analyzed performance bugs are caused by configurations faults. They recommend to identify configuration options that are more likely to cause a performance degradation and prioritize them during performance tests, to test software in a system closely similar to the production environment, and to rely on profiling to debug performance configuration failures.

Finally, once an option has been found to have an incorrect value, a number of approaches have been proposed to **suggest a correct option value**. Swanson et al. [94] try to propose a correct option value relatively close to the current (buggy) configuration using Firefox' configuration constraints (feature model) and a sampling algorithm (e.g., the n-hop algorithm [37], random sampling or covering array sampling). Xiong et al. [110], [111] proposed an algorithm that automatically generates range fixes for a violated constraint. Such fixes correspond to a range of correct values for one or multiple misconfigured options, from which a user can choose correct option values that respect his or her software system's configuration constraints. The approach starts from a misconfigured configuration and the software system's configuration constraints, then generates a range of possible values that each configuration option should have. Evaluation on Linux and eCos showed that the approach could find the range of fixes within a second.

6.8 Configuration Knowledge Sharing

R8.1 Communication between Developers and Users 8 developers highlight the importance of communication to define better configuration options. First, developers have to discuss configuration options before starting to code, defining their defaults, ranges and restrictions. Further, one developer proposes to "*let everyone discuss about the configurability needs*", then "*work with the client to understand what they will need to change*".

5 surveyed participants mentioned the importance of sharing configuration modifications. Developers should *"make sure all team members are updated about configuration changes"*, not only developers that should be notified by the new changes, but also the software users. This comprises newly created options, modification of option names and their default values, and also "*how it affects them*". Such openness of knowledge makes it much easier to find the best "owner" of a configuration option (challenge C1.4 in Table 3).

R8.2 Pragmatic Usage Documentation The most popular theme in our survey, discussed by 87 developers, is related to documentation. In short, it is "*extremely important [...] that users are not going out of their way to find out what an option does*".

One should "*have a proper documentation above each configuration option in the config file to know what that option does and how does it work in details*". One should document why an option is defined and why it should be added, because "*some users may not be using the software as you are so that the use*

cases/workflows that make sense to you, may not fit them". Default values should be detailed in the documentation, with concrete examples. Furthermore, interactions between configuration options should also be specified in the documentation.

Some developers recommend to indicate explicitly the locations where configuration documentation should be added. In addition to the comments added in the configuration file, it is recommended to document options in the project's README file, in the official project documentation, and also in the source code, in order to help developers easily identify where an option is used. One surveyed developer also suggested to automatically generate documentation based on the configuration information in the code base. Only in two interviewed cases, an architect mentioned that the developers have an explicit guideline that forces them to document configuration options. One of these two interviewees' company has an entire team focusing on documentation, including configuration-related documentation.

While we did not find any paper mentioning how configuration options should be documented, Murakami et al. [61] propose an approach that helps manual creators to create documentation. By exploiting the commit logs, which provide the line(s) changed in a configuration file, and a pre-prepared explanation of that modification, the approach generates an HTML manual that can then be edited manually.

A second approach that could be used to help developers understand options and hence create documentation, is the approach of Zhou et al. [123]. It tries to recover the mapping between a configuration option and its associated variable in the source code, which then allows the mining of the option's potential values and constraints. To maintain the mapping between configuration options in the source code and their associated documentation, Dong et al. [30], Rabkin et al. [72] and Zhou et al. [123] propose static analysis that helps developers automatically identify where each configuration option is used in the source code.

6.9 Quality Assurance

Finally, surveyed developers and our systematic literature review propose a set of recommendations to improve the quality of configuration options.

R9.1 Validation of Specific Configuration 25 surveyed developers discussed how to assure the quality of a software configuration by validating the correctness of configuration values. The proposed recommendations differ in terms of the time at which validation is performed.

Many surveyed developers recommended the use of offline configuration validators: "*having an open key/value configuration system is flexible but some of the most difficult configuration errors are of the typo variety (numReceivers vs numRecievers - typo difficult to catch). Would ideally have validators that would warn on unexpected configurations*". This recommendation is aligned with the work of Jha et al. [46] on "ManifestInspector". It identifies errors in an Android manifest configuration file by statically parsing a manifest and validating it against a set of rules stored in a database. Such rules are basically a set of possible XML tags with a set of possible attribute values. They were able to identify 59,547 errors in 11,110 of the 13,483 analyzed Android apps. Eshete et al. [34] build a model that checks the correctness of configuration options in web applications against a "Gold Standard" of configuration values that they built from online discussions, documentation, and expert opinions. Similarly, Zhang et al. [47] detect configuration errors based on a sample of configurations that are enriched with environment configurations.

Several developers recommended validation of configuration values at application boot-time. For example, developers should "*avoid making applications un-bootable in the absence of non-default configuration*". In addition, a surveyed developer suggest to "*do not check logic at run-time to avoid performance penalty, but do check simple value at loading/parsing/importing setting to configuration, accept only valid value*". Configuration value validators provide quick feedback of bad configurations (challenge C1.3) and also support reviewers in identifying incorrect option values (challenge C9.1).

No approaches were suggested or identified for run-time configuration validation. Instead, the most important feature of run-time configuration options that was discussed is the challenge of changing an option value without restarting the software system. A bug could lead to using the old values of these changed options and hence to miss the users' configuration updates. To fix such problems, Toman et al. [97] proposed an approach based on dynamic taint analysis. It consists of tracking both the usage of a configuration via data propagations and the modification of configuration options in the source code, then checking which version of an option is used in each statement that uses that option. Comparison of the version used by that statement with the last updated version then allows to find inconsistently updated options.

R9.2 Configuration-aware Testing In addition to validation of configuration values, developers also suggest to implement automated testing for configurations. Developers should "*not rely on manual testing of selected few configurations*". Apart from being able to support development of new configuration options, such tests also help to detect configuration regression bugs (challenges C7.1/2).

10 surveyed developers further suggest to adopt explicit testing strategies on configuration options. They basically propose to consider configuration options in integration tests or unit test suites. In addition, one developer proposes "*to have a clear configuration modules for integration testing with the appropriate mock framework, and integrating by using [CI] tools like Jenkins*". In one interviewed expert's context, developers use mutation testing [63] for configuration options.

Configuration-aware testing is the second most popular configuration activity in research (right after configuration failure debugging). Gao et al. [36] proposed a guideline to help developers test their software system in the presence of a configuration subsystem. They represent the configurability of the system as a semantic tree, which can then be used by developers to manually determine the conditions under which to test their software.

Most of the studies, however, focus on **reducing the number of tests to run in highly configurable software systems**, since ideally all possible combinations of all options should be tested. Most of the sampling algorithms discussed for R1.4 could be applied here as well. Bestoun et al [15] use the Cuckoo search algorithm to optimize the number of configurations to test by sampling the most relevant inputs

and configurations. Qu et al. [68], [109], [70] propose an approach based on "combinatorial interaction testing techniques". Such techniques sample a set of configurations to test from all possible combinations of configuration values, yet are not feasible for configuration-aware testing due to the large number of possible configurations. Hence, Qu et al. select a specific subset of configuration options to test between two versions. Huning et al. [45] propose a fuzzing technique to test a software system against vulnerabilities that occur only under certain configurations. Marijan et al. [57] built TITAN, which optimizes a test suite for highly configurable software systems using combinatorial interaction, a constraint-based approach to minimize tests, and test prioritization for regression tests. Fouche et al. [35] used the coverage arrays approach to select a subset of configuration options to test, which can be dynamically increased in size, depending on the available testing resources.

Qu et al. [69] instead use **static slicing** to determine if, given a configuration, a test needs to be executed under additional configurations. Nguyen et al. [62] dynamically identify which configurations (combinations of configuration values) cover a source code location in order to identify the minimum configurations necessary to cover the highest number of locations using the existing unit tests. Kim et al.'s SPLat [53] requires running a test, then testing all possible combinations of values of the options that were used in that single test, without focusing on other options. Souto et al. [91] extended this approach by considering sampling heuristics like one-disabled and most-enabled-disabled approaches. One-disabled approach consists of testing an application by disabling one option at a time, where most-enabled-disabled consists of testing an application with all its options enabled in a first run and all of them disabled in a second test. Souto et al.'s EvoSPLat [90] focuses on regression testing. EvoSPLat starts by a lightweight analysis that reports under which configuration options a source code modification could be executed, fixes that configuration option's values to guarantee that the modified source code will be covered, then tests the other configuration options' values based on the previously discussed SPLat approach [53].

While most of the work above assumes **that the search space of possible configurations is huge**, and hence needs to be filtered, a separate line of work focuses on **validating this assumption**. Meinicke et al. [59] (using dynamic analysis) and Reisner et al. [76] (using symbolic execution) both found that the interaction between options, i.e., "a partial setting of configuration options such that specific line, block, edge, or condition coverage is guaranteed to occur under that setting, but is not guaranteed by any of its subsets" [76], is surprisingly low. Such low interaction substantially reduces the possible configuration space, and hence makes testing of all relevant configurations easier. To find those relevant configurations, Song et al. [88] proposed an algorithm that uses a low strength covering array, run-time instrumentation, and machine learning to build an interaction tree that represents interactions between options. They extended this approach [89] by considering heuristics related to the interaction between configuration options, which they observed in two open source projects (vsftpd and ngIRCd). Again, they found that interactions (i.e., in terms of code controlled by a combination of options having specific values and without which such code is not reached) are rare.

In contrast to the above work, Keller et al. [51] **evaluate how resilient a software system is to configuration errors** by injecting faults into its configuration files. Xu et al. [113] extract configuration option constraints from source code in order to report error-prone and inconsistent constraints.

Another direction of testing highly configurable software systems is proposed by Robinson et al. [77], who aims to reduce the number of tests to generate and run by generating tests only for the options changed by a customer compared to the base configuration. Their approach consists of identifying the options modified by the user, finding the impacted area of the source code via control flow and data flow analysis, then selecting or generating tests that cover those source code locations.

Cohen et al. [26] developed a family of greedy Combinatorial Interaction Testing (CIT) sample generation algorithms that aim to reduce the number of tests to run by generating samples that satisfy the software configuration constraints. Comparative evaluation of the cost-effectiveness of these algorithms on four real-world highly-configurable software systems and on a population of synthetic examples shows that their techniques reduce the cost of CIT in the presence of constraints to 30% of the cost of widely-used unconstrained CIT methods, without sacrificing the quality of the solutions.

R9.3 Configuration-aware Reviewing 14 surveyed developers recommend reviewing changes to configuration options. Developers should "*carefully review any pull request that impacts configurations, reviewers must be not only developers but documentation writers and UI experts to make sure that [not only will the changes] just work, but it is clear how to use it and well documented*". This is less obvious than it sounds, since patches impacted by configuration options typically do not show the conditional checks (which are higher up in the code) nor the default option values (which are in separate files or other media). Only if the patch itself manipulates an option, reviewers receive an explicit hint that the patch is impacted by some option(s). Still, they need to manually check if the newly added/removed configuration-related code impacts other code that is not part of the patch. Because of this "hidden" nature of configuration-related code and data, one surveyed expert went as far as proposing to send "*configuration options as a separate patch*". The presence of detailed documentation or comments about configuration options (C5.2) might have a positive impact on configuration-aware reviewing.

R9.4 Fail-safe Cross-Environment Configuration 3 developers believe that configuration options should be easy to adapt to different environments (e.g., testing, staging and production), ideally in an automated fashion. Furthermore, the deployment of the software configuration storage media with the chosen values of configuration options should be defined as a workflow. Following one of the developers, "*nothing is more frustrating than having to figure out how to put an environment specific configuration on release day. A CI job which you can just trigger is ideal for deployment*".

R9.5 Approaches to Help Users Configure their Software In contrast to R1.4, the techniques discussed here aim to help users customize the default configuration options to their context (instead of finding generic default values

that work for as many end users as possible). One of the first approaches to help J2EE developers choose an optimal configuration value is proposed by Raghavachari et al. [73]. It allows to select a range of possible configuration values for each option, select a random value from that range, deploy the web application with that configuration and measure its performance, then choose a second value in order to compare its performance until finding a better configuration. Note that the approaches discussed for value generation in R1.4 also apply to R9.5 [40], [25], [55], [86], [107], [29], [95], [85], [64], [65], [79], [33], [122], [39], but applied by end users instead of developers.

Configuration complexity can be reduced by using Hamidi et al.'s [41] approach to reduce the number of decisions a user has to make in order to configure her software. Their approach models software configuration as a set of options that have relations between them, for example to express that whenever a user decides to assign a value v1 to an option X, she assigns a value v2 to another option Y. Such patterns can be used to automatically change configuration values based on few user decisions.

Huang et al. [43], [83] support developers configuring Java frameworks by recommending XML configuration snippets collected from open source repositories. Chen et al. [102] automatically detect correlated configuration options in multi-component/layer architectures by using a database of configuration choices obtained from online fora and websites. Ramachandran et al. [75] instead analyze the names and values of options, complemented by configuration data made available for analysis by software vendors or available online (e.g., fora). Finally, Jin et al. [48] proposed an approach to help users find which options they need to change the value for in a given situation, instead of asking around in an online forum.

While the above approaches for R9.5 seem promising, we did not record any usage of such automated techniques in our interviews or survey.

7 Implications

This section discusses potential implications of our findings for both practitioners and researchers.

7.1 Implications for Practitioners

While many configuration engineering activities are not covered by academic literature, many challenges related to creation of configuration option, configuration knowledge sharing and quality assurance have been studied. In particular, a substantial amount of work focuses on default value generation (R1.4), configuration failure debugging (R7.3), documentation generation for end users (R8.2), configuration validation (R9.1), configuration-aware testing (R9.2) and optimization of option values for end users (R9.5). While this work is of course not complete, practitioners should look into these techniques, provided they have access to the corresponding academic pulications.

Even if the other configuration engineering activities and challenges saw substantially less academic interest, practitioners can still build on their experience with programming best practices, since many of those still apply in the context of configuration options. In fact, one of the interviewees literally quoted that "*Configuration is code too*". Known programming tenets like the KISS principle (R1.1 and R2.1), option cohesion (R1.3), option granularity (R3.1), encapsulation of sensitive option data (R3.2), good naming conventions (R5.1), choosing the right abstraction for storage media (R2.2) and API for configuration access (R4.1), reuse of third party configuration libraries (R4.2), generation of clear logs (R7.1) and consistently documenting failures and resolutions (R7.2), all still apply to the context of configuration engineering. While standard methodologies and tool support might not exist, organizations could create their own based on these general principles.

In addition to these technical principles, we also identified some organizational recommendations for which academic support is missing. A team should not allow any developer to add or manipulate configurations, but should rely on expert developers with a broad vision of the software system, its architecture and its configuration options (R1.2). Similarly, the organization should establish guidelines to force developers to maintain configuration compatibility (or at least ensure traceability) between different versions of the application (R6.2). Finally, option developers need to know how to effectively communicate with other stakeholders the rationale and constraints of new or changed options (R8.1). For the moment, less guidance exists for these organization-level challenges.

Finally, the main challenges for practitioners in terms of configuration engineering support relate to comprehension of options and values, maintenance of options and quality assurance. In particular, tool and methodological support is needed for configuration understanding (R5.2), option maintenance (R6.1) and configuration-aware reviewing (R9.3). While identified as major challenges that impact the quality and maintainability of an application, surprisingly little research has been performed on them. At the same time, no general principles exist that could guide practitioners in the short-term.

7.2 Implications for Researchers

The first implication that we would mention is for researchers to advertise or publish their work in venues where developers can find them, for example industry conferences or even just simple blog posts. This follows from repeated comments of interviewed and surveyed practitioners about either not being aware of academic results or not having access to them. In particular, the wide variety of approaches related to debugging configuration failures saw only little adoption. While the adoption problem of course applies to any research domain, it especially matters in the domain of configuration engineering, which is at the same time specialized and general-purpose.

As discussed in the previous section, apart from the existing work on recommendations R1.4, R7.3, R8.2, R9.1, R9.2 and R9.5, most of the other recommendations, challenges and activities have been covered substantially less or sometimes not at all. This opens up a wide range of new research opportunities. Here, we point out some obvious opportunities, but many more exist. Basically, any cell in Table 3 could be potential avenue for future work.

First of all, it would be interesting to empirically study the impact of the programming best practices discussed earlier in the context of configuration engineering. For example, do

naming conventions, reuse of third party configuration libraries and pair-programming improve the quality of software configuration in general? Related to this, many configuration formats exist, including key-value pairs, hierarchical configuration options, or even complete DSLs for configuration. Further guidelines are required to help practitioners decide what format to use for which configuration goal and for which kind of users (technical engineers vs. regular end users).

We found that developers do not plan the evolution of their configuration options, while they fear thorough maintenance of options (i.e., option refactoring and removal). Therefore, developers require approaches to help them detect and remove dead options, or perform option refactoring in general. In practice, developers also need automated approaches to keep the configuration storage media in sync with the options used in the source code, in order to propagate any code changes (e.g., renaming) made to the options. This is a challenging problem, since we found that option names often are not literally mentioned in the source code. This same problem renders the understanding of software configuration in general complex. There are some existing approaches that locate configuration accesses in the source code, but even they still need to be improved to deal with string concatenation in the name of configuration options.

In general, the existing work on configuration activities, challenges and recommendations typically focuses on individual applications, without considering interactions with the execution environment or other layers of a software stack [80], [82], [103]. Apart from exploring the previously unexplored configuration activities, future work should also revisit existing analyses and methodologies from a system perspective instead of the application perspective. This will allow to develop frameworks and libraries for system-level configuration engineering, including system-level validation, debugging, etc.

8 Threats to Validity

Despite the effort spent on our qualitative study design, and gathering and clustering data from three different sources (expert interviews, a large survey, and a systematic literature review), we identified a number of threats to validity.

Internal Validity As first internal threat to validity, our results can miss challenges that are not faced by our interviewees. To mitigate this risk, we invited and interviewed experts that play different roles in different domains. Our 14 interviewed industry experts range from managers to developers and freelancers. They belong to different companies (located in four different countries) and work on different technologies and platforms.

The second threat to validity is related to the data collection during our interviews, as it is possible to misunderstand a challenge or recommendation discussed with interviewed experts. To mitigate this risk, we conducted all our interviews in face-to-face meetings (except one via a skype call), and all interviews were attended by two authors of the paper to avoid any confusing interpretation or misunderstanding. One author led the interview, while the other took notes. At the end of each interview, a debriefing was held to validate and complete the notes.

A third internal threat to validity is related to the data collection in our survey. Surveyed participants might have answered questions incorrectly due to different factors, such as misunderstanding of a question, lack of experience or lack of interest. To mitigate this risk, two authors carefully prepared the questionnaire, while the two other authors reviewed it. During this prepration, we avoided using any research-specific terms. In addition, we performed a pilot study with 9 respondents that allowed us to validate and improve the survey before sending it out to GitHub developers. In particular, the pilot study allowed to reformulate unclear questions or descriptions, restructure the questionnaire to make it easier to follow, shortening the list of questions by merging related questions, and incorporating other comments.

Furthermore, based on our selection criteria, the invited open source developers all are serious contributors to top Java projects on GitHub, and they were promised that responses would be treated anonymously. Only two unprofessional responses had to be filtered out. Our decision to focus on the top 1,000 most active Java projects allowed to filter out projects that are not software engineering-related or that are too small to actually need configuration options, since their developers would not be knowledgeable about configuration options. Our demographic results for the number of options in the survey respondents' projects confirm that the number of commits was a good proxy for configuration expertise. If one would know the concrete configuration access API used by each GitHub project, one could perform a more accurate sampling based on actual configuration API usage.

The fifth internal threat to validity is related to the difference between interviewed and surveyed participants' contexts (industrial vs. open source projects), and due to which we might miss some challenges or recommendations specific to either context. As is clear from the discussion of our results in sections 5 and 6, both types of participants agreed on the majority of challenges and recommendations, except for C9.1. For the latter challenge, open source projects rely on GitHub's pull request mechanism to review configuration as well as code changes, while the pull request mechanism could be less adopted in industrial contexts. Furthermore, no additional configuration activities were identified from the survey responses' open questions.

The sixth internal threat to validity concerns our three card sort analyses, which respectively concern the analysis of the interviews, open survey questions, and the classification of the papers obtained from our systematic literature review. Such card sort analyses could lead to subjective results, which could influence our results and findings. To mitigate this risk, all our card sort analyses initially were conducted by two authors for the interviews, and by one author for the survey and systematic literature review. Then, each of these card sort analysis results were reviewed by the other authors. In case of conflicts, the card sort classifier and the reviewing authors together discussed the conflicts to reach a final decision. Finally, the consistent results across all analyses provide further confidence in our findings.

Our seventh internal threat to validity concerns the four SLR selection criteria, since they shaped to a large extent the final selection of papers. In particular, we used criterion 2 to reduce noise in our first iteration caused by systems research on execution environment configuration or tweaking of system performance. The latter domain uses configuration options as a means rather than a goal, and focuses especially

on the performance of middleware and environments. While the results of our first iteration based on the selection criteria might make us miss relevant papers, the recursive snowballing iteration allowed us to recover missing papers and their relevant references, helping us complete a stable set of papers.

External Validity As with all qualitative studies, there is an inherent risk to generalize our findings for all organizations. While the 14 interviewees and 229 respondents cover a wide range of projects and application domains, and provided consistent results across the discussed challenges and recommendations, future work should consider additional organizations and developers. Furthermore, we only considered Java projects in the interviews and survey because (1) it is one of the most popular programming languages and (2) the Java ecosystem has a wealth of configuration frameworks [28], [81]. Other languages like C++, Python or Javascript might experience different configuration challenges and recommendations. Hence, future work should consider such languages. The results of our systematic literature survey are not language-specific.

Despite these limitations, we noticed that the major challenges and recommendations that came out of our study were saturated, in the sense that the progressive card sort analyses on the three data sources confirmed earlier activities and challenges, without suggesting new ones. R7.3 and R9.5 only popped up in the systematic literature review. Further studies on different domains and participants are required to verify our results and make our findings more general.

9 Conclusion

This paper aims to improve the general understanding of the software configuration engineering process, its challenges and existing practices (both from practice and academia). Through a series of 14 semi-structured interviews, a survey with 229 GitHub developers, and a systematic literature survey, we found how configuration engineering comprises 9 activities, and is impacted by 22 open challenges. Finally, we discussed 24 recommendations to overcome or avoid the challenges, derived from practitioners' experience and academic literature. The data of our studies is available online [10].

Our interviews identified an initial set of activities, challenges, and recommendations, which were then enriched and confirmed via the survey. The survey also provided insights about the popularity of the challenges from a larger set of participants. On the other hand, our systematic literature review revealed various recommendations that are not fully covered, or not covered at all, by the literature (R1.2, R1.3, R2.1, R4.1, R5.1, R6.1, R6.2, R8.1, R9.3 and R9.4), and two additional recommendations for which we did not record any usage in the practice (R7.3 and R9.5).

These contributions have a number of major implications for practitioners and researchers. First of all, they consolidate the state of the art and state of the practice in configuration engineering, providing both practitioners and researchers a clear overview of this domain, its challenges and best practices. Second, our findings reveal activities and recommendations without substantial research effort thus far, in particular the choice of configuration framework (R4.2), comprehension of configuration options (name, meaning, etc.) (R5.1), dead option detection (R6.1) and configuration-aware reviewing (R9.3). While the other challenges and practices of course have value, these in particular could need more attention.

Finally, our study could be extended outside the scope of run-time configuration options, in particular to pre-deployment configuration engineering (e.g., using conditional compilation), as well as to other roles and software domains.

References

[1] "Original questions ideas of the interview," http://mcis.polymtl.ca/publications/2018//TSE/Interviews/.
[2] "Apache zookeeper," http://zookeeper.apache.org, 2008.
[3] "Configuration file," https://stackoverflow.com/questions/648246/at-what-point-does-a-config-file-become-a-programming-language, 2009.
[4] "Amazon misconfiguration," 2011, https://www.pcworld.com/article/226033/thanks_amazon_for_making_possible_much_of_the_internet.html.
[5] "Github api," https://developer.github.com/v3/, 2012.
[6] "Configuration error," 2014, https://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale/.
[7] "Study about surveys," 2015, https://www.checkmarket.com/blog/survey-invitations-best-time-send/.
[8] "Card sort of surveyed experts challenges," 2018, http://mcis.polymtl.ca/publications/2018/TSE/survey/ChallengesCardSort.html.
[9] "Card sort of surveyed experts recommendations," 2018, http://mcis.polymtl.ca/publications/2018//TSE/survey/BestPracticesCardSort.html.
[10] "Data," 2018, http://mcis.polymtl.ca/publications/2018//TSE/.
[11] "Interview card sort," 2018, http://mcis.polymtl.ca/publications/2018//TSE/CardSort/.
[12] "Survey answers," 2018, http://mcis.polymtl.ca/publications/2018//TSE/survey/.
[13] "Survey questionnaire," 2018, https://goo.gl/forms/cR7CkPgxcXxaddC12.
[14] E. K. Abbasi, M. Acher, P. Heymans, and A. Cleve, "Reverse engineering web configurators," in *Proceedings of the Conference on Software Maintenance, Reengineering, and Reverse Engineering*, ser. CSMR-WCRE'14, 2014, pp. 264–273.
[15] B. S. Ahmed, T. S. Abdulsamad, and M. Y. Potrus, "Achievement of minimized combinatorial test suite for configuration-aware software functional testing using the cuckoo search algorithm," *Information and Software Technology*, vol. 66, pp. 13–29, 2015.
[16] F. A. Arshad, R. J. Krause, and S. Bagchi, "Characterizing configuration problems in java ee application servers: An empirical study with glassfish and jboss," in *Proceedings of the 24th International Symposium on Software Reliability Engineering*, 2013, pp. 198–207.
[17] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating root-cause diagnosis of performance anomalies in production software," in *Proceedings of the 10th Usenix Symposium on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 307–320.
[18] M. Attariyan and J. Flinn, "Using causality to diagnose configuration bugs," in *USENIX Annual Technical Conference*, Conference Proceedings, pp. 281–286.
[19] ——, "Automating configuration troubleshooting with dynamic information flow analysis," in *Proceedings of the 9th Usenix Symposium on Operating Systems Design and Implementation (OSDI)*, 2010, pp. 1–14.
[20] ——, "Automating configuration troubleshooting with confaid," *USENIX; login*, vol. 36, no. 1, pp. 27–36, 2011.
[21] F. Behrang, M. B. Cohen, and A. Orso, "Users beware: Preference inconsistencies ahead," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 295–306.
[22] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki, "Variability modeling in the real: A perspective from the operating systems domain," in *Proceedings of the International Conference on Automated Software Engineering*, 2010, pp. 73–82.

[23] Q. Boucher, E. K. Abbasi, A. Hubaux, G. Perrouin, M. Acher, and P. Heymans, “Towards more reliable configurators: A re-engineering perspective,” in *Proceedings of the 3rd International Workshop on Product Line Approaches in Software Engineering*, 2012, pp. 29–32.

[24] X. Bowen, D. Lo, X. Xin, A. Sureka, and L. Shanping, “Efspredictor: Predicting configuration bugs with ensemble feature selection,” in *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pp. 206–213.

[25] H. Chen, G. Jiang, H. Zhang, and K. Yoshihira, “Boosting the performance of computing systems through adaptive configuration tuning,” in *Proceedings of the ACM symposium on Applied Computing*, 2009, pp. 1045–1049.

[26] M. B. Cohen, M. B. Dwyer, and J. Shi, “Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach,” *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 633–650, 2008.

[27] B. K. Debnath, D. J. Lilja, and M. F. Mokbel, “Sard: A statistical approach for ranking database tuning parameters,” in *procedings of the 24th International Conference on Data Engineering Workshop*, 2008, pp. 11–18.

[28] V. Denisov, “Overview of java application configuration frameworks,” *International Journal of Open Information Technologies*, vol. 1, no. 6, pp. 5–9, 2013.

[29] Y. Diao, J. L. Hellerstein, S. Parekh, and J. P. Bigus, “Managing web server performance with autotune agents,” *IBM Systems Journal*, vol. 42, no. 1, pp. 136–149, 2003.

[30] Z. Dong, A. Andrzejak, D. Lo, and D. Costa, “Orplocator: Identifying read points of configuration options via static analysis,” in *Proceedings of the 27th International Symposium on Software Reliability Engineering*, 2016, pp. 185–195.

[31] Z. Dong, A. Andrzejak, and K. Shao, “Practical and accurate pinpointing of configuration errors using static analysis,” in *Proceedings of the 31st International Conference on Software Maintenance and Evolution*, ser. ICSME’15, 2015, pp. 171–180.

[32] Z. Dong, M. Ghanavati, and A. Andrzejak, “Automated diagnosis of software misconfigurations based on static analysis,” in *Proceedings of the International Symposium on Software Reliability Engineering Workshops*, 2013, pp. 162–168.

[33] S. Duan, V. Thummala, and S. Babu, “Tuning database configuration parameters with ituned,” *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 1246–1257, 2009.

[34] B. Eshete, A. Villafiorita, K. Weldemariam, and M. Zulkernine, “Confeagle: Automated analysis of configuration vulnerabilities in web applications,” in *Proceedings of the 7th International Conference on Software Security and Reliability (SERE’13)*, 2013, pp. 188–197.

[35] S. Fouche, M. B. Cohen, and A. Porter, “Incremental covering array failure characterization in large configuration spaces,” in *Proceedings of the 18th International Symposium on Software Testing and Analysis, ISSTA 2009*, pp. 177–187.

[36] J. Gao, J. Guan, A. Ma, C. Tao, X. Bai, and D. C. Kung, “Testing configurable component-based software - configuration test modeling and complexity analysis,” in *Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering*, 2011, pp. 495–502.

[37] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, “Using feature locality: can we leverage history to avoid failures during reconfiguration?” in *Proceedings of the 8th workshop on Assurances for self-adaptive systems*, 2011, pp. 24–33.

[38] G. Gousios, “The ghtorrent dataset and tool suite,” in *Proceedings of the 10th Working Conference on Mining Software Repositories*, 2013, pp. 233–236.

[39] G. Gousios, V. Karakoidas, and D. Spinellis, “Tuning java’s memory manager for high performance server applications,” in *Proceedings of the 5th International System Administration and Network Engineering Conference SANE*, 2006, pp. 69–83.

[40] J. Guo, K. Czarnecki, S. Apely, N. Siegmundy, and A. Wasowski, “Variability-aware performance prediction: A statistical learning approach,” in *Proceedings of the 28th International Conference on Automated Software Engineering*, pp. 301–311.

[41] S. Hamidi, P. Andritsos, and S. Liaskos, “Constructing adaptive configuration dialogs using crowd data,” in *Proceedings of the 29th International Conference on Automated Software Engineering*, 2014, pp. 485–490.

[42] X. Han and T. Yu, “An empirical study on performance bugs for highly configurable software systems,” in *Proceedings of the 10th International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM’16, 2016.

[43] S. Huang, Y. Q. Lu, Y. Xiao, and W. Wang, “Mining application repository to recommend xml configuration snippets,” in *Proceedings of the 34th International Conference on Software Engineering, ICSE’12*, pp. 1451–1452.

[44] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, 1st ed. Addison-Wesley Professional, 2010.

[45] D. Huning, C. Murphy, and G. Kaiser, “Confu: Configuration fuzzing testing framework for software vulnerability detection,” *International Journal of Secure Software Engineering*, vol. 1, no. 3, pp. 41–55, 2010.

[46] A. K. Jha, L. Sunghee, and L. Woo Jin, “Developer mistakes in writing android manifests: An empirical study of configuration errors,” in *Procedings of the 14th International Conference on Mining Software Repositories (MSR’17)*, 2017, pp. 25–36.

[47] Z. Jiaqi, L. Renganarayana, Z. Xiaolan, G. Niyu, V. Bala, X. Tianyin, and Z. Yuanyuan, “Encore: exploiting system environment and correlation information for misconfiguration detection,” *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 687–700, 2014.

[48] D. Jin, M. B. Cohen, X. Qu, and B. Robinson, “Preffinder: Getting the right preference in configurable software systems,” in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE 2014*, pp. 151–162.

[49] D. Jin, X. Qu, M. B. Cohen, and B. Robinson, “Configurations everywhere: Implications for testing and debugging in practice,” in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE’14, pp. 215–224.

[50] R. Johnson, “More details on today’s outage,” MathWorld–A Wolfram Web Resource, September 2010, https://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919/.

[51] L. Keller, P. Upadhyaya, and G. Candea, “Conferr: a tool for assessing resilience to human configuration errors.” IEEE, Conference Proceedings, pp. 157–66.

[52] E. Kiciman and Y.-M. Wang, “Discovering correctness constraints for self-management of system configuration,” in *Autonomic Computing, 2004. Proceedings. International Conference on.* IEEE, 2004, pp. 28–35.

[53] C. H. P. Kim, D. Marinov, S. K. D. Batory, S. Souto, P. Barros, and M. D’Amorim, “Splat: Lightweight dynamic analysis for reducing combinatorics in testing configurable systems,” in *Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE’13*, 2013, pp. 257–267.

[54] B. Kitchenham and S. Charters, “Guidelines for performing systematic literature reviews in software engineering,” in *Technical report, Ver. 2.3 EBSE Technical Report. EBSE.* sn, 2007.

[55] P. Lengauer and H. MÃűssenbÃűck, “The taming of the shrew: increasing performance by automatic parameter tuning for java garbage collectors,” in *Proceedings of the 5th international conference on Performance engineering*, pp. 111–122.

[56] M. Lillack, C. Kastner, and E. Bodden, “Tracking load-time configuration options,” in *Proceedings of the 29th International Conference on Automated Software Engineering (ASE’14)*, 2014, pp. 445–456.

[57] D. Marijan, M. Liaaen, A. Gotlieb, S. Sen, and C. Ieva, “Titan: Test suite optimization for highly configurable software,” in *Proceedings of the 10th International Conference on Software Testing, Verification and Validation*, 2017, pp. 524–531.

[58] S. McConnell, *Code Complete*, ser. DV-Professional. Microsoft Press, 2009. [Online]. Available: https://books.google.ca/books?id=3JfE7TGUwvgC

[59] J. Meinicke, W. Chu-Pan, C. Kastner, T. Thum, and G. Saake, “On essential configuration complexity: Measuring interactions in highly-configurable systems,” in *Proceedings of the 31st International Conference on Automated Software Engineering (ASE’16)*, 2016, pp. 483–494.

[60] J. Mickens, M. Szummer, and D. Narayanan, “Snitch: Interactive decision trees for troubleshooting misconfigurations,” in *Proceedings of the 2Nd USENIX Workshop on Tackling Computer Systems Problems with Machine Learning Techniques*, ser. SYSML’07. USENIX Association, 2007, pp. 8:1–8:6.

[61] Y. Murakami, E. Kagawa, and N. Funabiki, "Automatic generation of configuration manuals for open-source software," in *Proceedings of the 5th International Conference on Complex, Intelligent and Software Intensive Systems*, 2011, pp. 653–658.

[62] T. V. Nguyen, U. Koc, J. Cheng, J. S. Foster, and A. A. Porter, "Igen: Dynamic interaction inference for configurable software," in *Proceedings of the 24th International Symposium on Foundations of Software Engineering (FSE'16)*, 2016, pp. 655–665.

[63] A. J. Offutt and R. H. Untch, *Mutation 2000: Uniting the Orthogonal*. Boston, MA: Springer US, 2001, pp. 34–44.

[64] T. Osogami and T. Itoko, "Finding probably better system configurations quickly," *Performance Evaluation Review*, vol. 34, no. 1, pp. 264–75.

[65] T. Osogami and S. Kato, "Optimizing system configurations quickly by guessing at the performance," in *Proceedings of the International Conference of Performance Evaluation Review*, 2007, pp. 145–156.

[66] H. Otsuka, Y. Watanabe, and Y. Matsumoto, "Learning from before and after recovery to detect latent misconfiguration," in *Proceedings of the 39th International Conference of Computer Software and Applications*, 2015, pp. 141–148.

[67] G. Perrouin, M. Acher, J. M. Davril, A. Legay, and P. Heymans, "A complexity tale: Web configurators," in *Proceedings of the 1st International Workshop on Variability and Complexity in Software Design*, 2016, pp. 28–31.

[68] X. Qu, "Configuration aware prioritization techniques in regression testing," in *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, 2009, pp. 375–378.

[69] X. Qu, M. Acharya, and B. Robinson, "Impact analysis of configuration changes for test case selection," in *Proceedings of the 22nd International Symposium on Software Reliability Engineering*, 2011, pp. 140–149.

[70] X. Qu, M. B. Cohen, and G. Rothermel, "Configuration-aware regression testing: An empirical study of sampling and prioritization," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, 2008, pp. 75–85.

[71] A. Rabkin and R. Katz, "Precomputing possible configuration error diagnoses," in *Proceedings of the 26th International Conference on Automated Software Engineering*, 2011, pp. 193–202.

[72] ——, "Static extraction of program configuration options," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, 2011, pp. 131–140.

[73] M. Raghavachari, D. Reimer, and R. D. Johnson, "The deployer's problem: configuring application servers for performance and reliability," in *Proceedings of the 25th international conference on Software engineering*, 2003, pp. 484–489.

[74] M. T. Rahman, L.-P. Querel, P. C. Rigby, and B. Adams, "Feature toggles: A case study and survey," in *Proceedings of the 13th IEEE Working Conference on Mining Software Repositories (MSR)*, Austin, TX, May 2016, pp. 201–211.

[75] V. Ramachandran, M. Gupta, M. Sethi, and S. R. Chowdhury, "Determining configuration parameter dependencies via analysis of configuration data from multi-tiered enterprise applications," in *Proceedings of the 6th International Conference on Autonomic Computing*, 2009, pp. 169–178.

[76] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter, "Using symbolic evaluation to understand behavior in configurable software systems," in *Proceedings of the 32nd International Conference on Software Engineering (ICSE'10)*, vol. 1, 2010, pp. 445–454.

[77] B. Robinson and L. White, "Testing of user-configurable software systems using firewalls," in *Proceedings of the 19th International Symposium on Software Reliability Engineering*, 2008, pp. 177–186.

[78] G. Rugg and P. McGeorge, "The sorting techniques: a tutorial paper on card sorts, picture sorts and item sorts," *Expert Systems*, vol. 22, no. 3, pp. 94–107, 2008.

[79] A. Sarkar, G. Jianmei, N. Siegmund, S. Apel, and K. Czarnecki, "Cost-efficient sampling for performance prediction of configurable systems (t)," in *Proceedings of the 30th International Conference on Automated Software Engineering*, 2015, pp. 342–352.

[80] M. Sayagh and B. Adams, "Multi-layer software configuration: Empirical study on wordpress," in *Proceedings of the 15th International Working Conference on Source Code Analysis and Manipulation (SCAM'15)*, 2015, pp. 31–40.

[81] M. Sayagh, Z. Dong, A. Andrzejak, and B. Adams, "Does the choice of configuration framework matter for developers? empirical study on 11 java configuration frameworks," in *Proceedings of the 17th International Working Conference on Source Code Analysis and Manipulation (SCAM'17)*, 2017, pp. 41–50.

[82] M. Sayagh, N. Kerzazi, and B. Adams, "On cross-stack configuration errors," in *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*, 2017, pp. 255–265.

[83] H. Sheng, X. Yanghua, L. Yiqi, W. Wei, and W. Yu, "Xml-snippet: A coding assistant for xml configuration snippet recommendation," in *Proceedings of the 36th Annual Computer Software and Applications Conference*, 2012, pp. 312–321.

[84] N. Siegmund, A. Grebhahn, S. Apel, and C. Kastner, "Performance-influence models for highly configurable systems," in *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)*, 2015, pp. 284–294.

[85] N. Siegmund, S. S. Kolesnikov, C. KÃďstner, S. Apel, D. Batory, M. RosenmÃijller, and G. Saake, "Predicting performance via automated feature-interaction detection," in *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, 2012, pp. 167–177.

[86] R. Singh, C.-P. Bezemer, W. Shang, and A. E. Hassan, "Optimizing the performance-related configurations of object-relational mapping frameworks using a multi-objective genetic algorithm," in *Proceedings of the 7th International Conference on Performance Engineering*, 2016, pp. 309–320.

[87] I. Sommerville, *Software Engineering*. USA: Addison-Wesley Publishing Company, 2010.

[88] C. Song, A. Porter, and J. S. Foster, "itree: Efficiently discovering high-coverage configurations using interaction trees," in *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, 2012, pp. 903–913.

[89] ——, "Itree: Efficiently discovering high-coverage configurations using interaction trees," *IEEE Transactions on Software Engineering*, vol. 40, no. 3, pp. 251–265, 2014.

[90] S. Souto and M. d'Amorim, "Time-space efficient regression testing for configurable systems," *Journal of Systems and Software*, vol. 137, pp. 733–746, 2018.

[91] S. Souto, M. D'Amorim, and R. Gheyi, "Balancing soundness and efficiency for practical testing of configurable systems," in *Proceedings of the 39th International Conference on Software Engineering, ICSE'17, 2017*, 2017, pp. 632–642.

[92] Y.-Y. Su, M. Attariyan, and J. Flinn, "Autobash: improving configuration management with operating system causality analysis," in *ACM SIGOPS Operating Systems Review*, vol. 41. ACM, 2007, pp. 237–250.

[93] L. Sun, G. Huang, Y. Sun, H. Song, and H. Mei, "An approach for generation of j2ee access control configurations from requirements specification," in *Proceedings of the 8th International Conference on Quality Software*, ser. - International Conference on Quality Software. Inst. of Elec. and Elec. Eng. Computer Society, pp. 87–96.

[94] J. Swanson, M. B. Cohen, M. B. Dwyer, B. J. Garvin, and J. Firestone, "Beyond the rainbow: Self-adaptive failure avoidance in configurable systems," in *Proceedings of the 22nd International Symposium on the Foundations of Software Engineering (FSE'14)*, 2014, pp. 377–388.

[95] R. Thonangi, V. Thummala, and S. Babu, "Finding good configurations in high-dimensional spaces: Doing more with less," in *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems*, 2008, pp. 1–10.

[96] X. Tianyin and Z. Yuanyuan, "Systems approaches to tackling configuration errors: a survey," *ACM Computing Surveys*, vol. 47, no. 4, pp. 1–41, 2015.

[97] J. Toman and D. Grossman, "Staccato: A bug finder for dynamic configuration updates," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 56. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[98] B. Wang, L. Passos, Y. Xiong, K. Czarnecki, H. Zhao, and W. Zhang, "Smartfixer: Fixing software configurations based on dynamic priorities," in *Proceedings of the 17th International Software Product Line Conference*. ACM, Conference Proceedings, pp. 82–90.

[99] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang, "Automatic misconfiguration troubleshooting with peerpressure," in *OSDI*, vol. 4, 2004, pp. 245–257.

[100] R. Wang, X. Wang, K. Zhang, and Z. Li, "Towards automatic reverse engineering of software security configurations," in *Proceedings of the 15th Conference on Computer and Communications Security*, 2008, pp. 245–255.

[101] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang, "Strider: A black-box, state-based approach to change and configuration management and support," *Science of Computer Programming*, vol. 53, no. 2 SPEC. ISS., pp. 143–164, 2004.

[102] C. Wei, W. Heng, W. Jun, Z. Hua, and H. Tao, "Determine configuration entry correlations for web application systems," in *Proceedings of the 40th Annual Computer Software and Applications Conference*, 2016, pp. 42–52.

[103] C. Wei, Q. Xiaoqiang, W. Jun, Z. Hua, and H. Xiang, "Detecting inter-component configuration errors in proactive: a relation-aware method," in *Proceedings of the 14th International Conference on Quality Software*, 2014, pp. 184–199.

[104] M. Welsh, "What i wish systems researchers would work on," May 2013, http://matt-welsh.blogspot.ca/2013/05/what-i-wish-systems-researchers-would.html.

[105] W. Wen, T. Yu, and J. H. Hayes, "Colua: Automatically predicting configuration bug reports and extracting configuration options," in *Proceedings of the 27th International Symposium on Software Reliability Engineering*, 2016, pp. 150–161.

[106] A. Whitaker, R. S. Cox, and S. D. Gribble, "Configuration debugging as search: finding the needle in the haystack," in *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation*, 2004, pp. 77–90.

[107] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang, "A smart hill-climbing algorithm for application server configuration," in *Proceedings of the Thirteenth International World Wide Web Conference*, 2004, pp. 287–296.

[108] X. Xia, D. Lo, W. Qiu, X. Wang, and B. Zhou, "Automated configuration bug report prediction using text mining," in *Proceedings of the 38th Annual IEEE Computer Software and Applications Conference*, 2014, pp. 107–116.

[109] Q. Xiao, M. Acharya, and B. Robinson, "Configuration selection using code change impact analysis for regression testing," in *Proceedings of the International Conference on Software Maintenance*, 2012, pp. 129–138.

[110] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki, "Generating range fixes for software configuration," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 58–68.

[111] Y. Xiong, H. Zhang, A. Hubaux, S. She, J. Wang, and K. Czarnecki, "Range fixes: Interactive error resolution for software configuration," *IEEE Transactions on Software Engineering*, vol. 41, no. 6, pp. 603–619, 2015.

[112] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, you have given me too many knobs! : Understanding and dealing with over-designed configuration in system software," in *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering(ESEC/FSE'15)*, 2015, pp. 307–319.

[113] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do not blame users for misconfigurations," in *Proceedings of the 24th Symposium on Operating Systems Principles*, 2013, pp. 244–259.

[114] X. Xu, S. Li, Y. Guo, W. Dong, W. Li, and X. Liao, "Automatic type inference for proactive misconfiguration prevention," in *Proceedings of the 29th International Conference on Software Engineering and Knowledge Engineering*, 2017, pp. 295–300.

[115] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, 2011, pp. 159–172.

[116] D. Yuan, Y. Xie, R. Panigrahy, J. Yang, C. Verbowski, and A. Kumar, "Context-based online configuration-error detection," in *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*. USENIX Association, 2011, pp. 28–28.

[117] S. Zhang, "Confdiagnoser: An automated configuration error diagnosis tool for java software," in *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*, 2013, pp. 1438–1440.

[118] S. Zhang and M. D. Ernst, "Automated diagnosis of software configuration errors," in *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*, 2013, pp. 312–321.

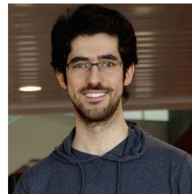
[119] ——, "Which configuration option should i change?" in *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*, 2014, pp. 152–163.

[120] ——, "Proactive detection of inadequate diagnostic messages for software configuration errors," in *Proceedings of the 24th International Symposium on Software Testing and Analysis*, 2015, pp. 12–23.

[121] Y. Zhang, J. M. Guo, E. Blais, and K. Czarnecki, *Performance Prediction of Configurable Software Systems by Fourier Learning*, 2015, pp. 365–373.

[122] W. Zheng, R. Bianchini, and T. D. Nguyen, "Massconf: automatic configuration tuning by leveraging user community information," in *Proceedings of the International Conference of Software Engineering Notes*, 2011, pp. 283–288.

[123] S. Zhou, X. Liu, S. Li, W. Dong, X. Liao, and Y. Xiong, "Confmapper: Automated variable finding for configuration items in source code," in *Proceedings of the 2nd International Conference on Software Quality, Reliability and Security-Companion*, 2016, pp. 228–235.



Mohammed Sayagh is a PhD candiate at the Lab on Maintenance, Construction, and Intelligence of Software (MCIS) in Ecole Polytechnique Montreal (Canada). His research interests include software configuration debugging and maintenance, as well as multi-layer and components architectures source code analysis. His work has been published at major software engineering venues, such as International Conference on Software Engineering (ICSE). He obtained his engineering degree in Software Engineering from Faculty of Science and Techniques in Mohammedia (FSTM) - Morocco. More details about his work is available on "http://mcis.polymtl.ca/~msayagh".

Noureddine Kerzazi Noureddine Kerzazi is an assistant professor at (ENSIAS) Mohammed V University, Morocco. His research interests include improvement of software processes and practices, Continuous Integration/Delivery, Branching strategies, Post-release failures Analysis, and software team members' Coordination. Noureddine obtained his PhD degree in computer science engineering from Polytechnique Montreal in 2010. He has been Integrator & Release Manager at an online payment processor for more than three years. He leads development of DSL4SPM, a conceptual tool for software process modeling.

Bram Adams is an associate professor at Polytechnique Montreal, where he heads the Lab on Maintenance, Construction, and Intelligence of Software. His research interests include release engineering in general, as well as software integration, software build systems, and infrastructure as code. Adams obtained his PhD in computer science engineering from Ghent University. He is a steering committee member of the International Workshop on Release Engineering (RELENG) and program co-chair of SCAM 2013, SANER 2015, ICSME 2016 and MSR 2019.

Fabio Petrillo is a Research Associate at Concordia University and Lecturer at Polytechnique Montreal. He received his Ph.D. degree in Computer Science in 2016 from Federal University of Rio Grande do Sul (Brazil) under the supervision of Dr. Marcelo Pimenta and Dr. Carla Freitas. Dr. Petrillo has worked on Software Quality, and Architecture, Debugging, Service-Oriented Architecture, RESTful analysis on Cloud, and Agile methods. He has been recognized as a pioneer and an international reference on Digital Games and Software Engineering. Further, he was the creator of Swarm Debugging, a new collaborative approach to support debugging activities.