Noname manuscript No.
(will be inserted by the editor)

Performance Evolution of Configurable Software Systems: An Empirical Study

Christian Kaltenecker · Stefan Mühlbauer · Alexander Grebhahn · Norbert Siegmund · Sven Apel

the date of receipt and acceptance should be inserted later

Abstract As a software system evolves, its performance can improve or degrade over time. Performance evolution is especially delicate in *configurable* software systems, where performance degradation may manifest only for specific configurations, making it especially hard to spot and fix.

Problem. Prior work concentrated mainly on performance-bug detection and root-cause analysis of a single version of a system. The big picture of how performance co-evolves with a system and what role configurability plays is largely unclear.

Approach. In an empirical study, we investigate the relation between configurability and performance evolution. Specifically, we analyze a total of 190 releases of 12 configurable real-world systems and examine the extent to which performance changes are specific to particular configurations and whether few or many configuration options cause performance changes. We triangulate our findings by analyzing change logs and commit messages of the respective projects to pin down causes of performance changes.

Results. We found that almost every release of every subject system exhibits performance changes in some of their configurations. Notably, the majority of performance changes affects only a subset of the configuration space, and most performance changes are triggered by multiple options (up to 6). In a deeper analysis, we found that a considerable number of releases mention performance changes in the change log and commits: performance changes are reported in 45% and 69% of the releases in the change log and the commit messages, respectively, but only a fraction report the involved configuration options.

Keywords configurable software systems · performance · evolution · performance changes

C. Kaltenecker · S. Apel
Saarland University,
Saarland Informatics Campus

A. Grebhahn
adesso SE

S. Mühlbauer · N. Siegmund
Leipzig University

1 Introduction

Software systems must evolve constantly to adapt to changes of hardware and user requirements (Xu et al., 2015). Software evolution is driven by the integration of new functionality or libraries, refactoring, and bug fixes. Beside functionality, the performance of the system may change considerably. A *performance change* refers to a situation in which the execution time (or another property such as throughput) of a software system degrades (performance regression) or improves (performance fix or performance optimization) compared to previous releases.

There is a substantial corpus of previous work on analyzing, detecting, and reverting performance changes (Chen and Shang, 2017; Burnim et al., 2009; Han et al., 2012; Mühlbauer et al., 2019), considering only a single or few default configurations across multiple releases of the software. However, performance changes may be *configuration-dependent*, that is, they appear only in a subset of configurations of the system in question (Han and Yu, 2016). As such, configuration-dependent changes could be easily missed by considering only the default configuration. Given that contemporary software systems are often configurable (Han and Yu, 2016), this calls for investigating performance changes not only across *multiple versions*, but simultaneously across *multiple configurations*.

So far, there is no clear picture of how severe and frequent performance changes are in configurable software systems and whether individual configurations or configuration options play a central role in the evolution of a system's performance behavior. A systematic analysis of performance changes of configurable software systems holds the promise of providing insights beyond just studying default configurations or average performance behavior. Developers and users are interested in which specific configurations exhibit diverging performance behavior and which configuration options (or interactions among options) are responsible for this. At a conceptual level, insights on the nature and prevalence of configuration-dependent performance changes can be used to improve configuration sampling and performance modeling techniques, where only a representative subset of all software configurations is used for performance prediction (Siegmund et al., 2015; Jamshidi et al., 2018; Kaltenecker et al., 2019; Pett et al., 2019).

To learn about performance changes in configurable software systems, we conduct an empirical study on performance evolution of 12 popular configurable open-source software systems from different domains across multiple releases and covering the entire configuration space. To pin down the performance changes to configuration options, we make use of the structure of performance-influence models (obtained by machine learning).

In particular, we address the following research questions:

- $RQ_{1.1}$*:* What is the fraction of the configuration space containing performance changes between consecutive releases?
- $RQ_{1.2}$*:* How stable is the relative performance of configurations in the presence of performance changes between consecutive releases?
- $RQ_{2.1}$*:* How frequent and how strong are changes of performance influences of individual configuration options and interactions between consecutive releases?
- $RQ_{2.2}$*:* How stable is the relative influence of configuration options and interactions in the presence of performance changes between consecutive releases?

To answer these research questions, we examine the prevalence and properties of performance changes at two levels of abstraction:

- *Configuration-level*: performance of individual configurations
- *Option-level*: performance influence of individual configuration options and interactions.

In a deeper analysis, we contrast this information to the change log and commit messages of the respective projects.
Overall, we make the following contributions:

- A novel approach to use performance-influence models to identify performance changes associated with specific configuration options.
- An empirical study of 12 popular configurable software systems involving their complete configuration spaces for a series of releases considering up to 15 years of evolution.
- Insights on what role configurability plays in performance evolution of configurable systems, which (kinds of) options and interactions cause performance changes, and which performance changes are documented.

In a nutshell, we found that almost all 190 releases that we analyzed exhibit, at least, one performance change in, at least, one configuration. Most performance changes (75%) affect less than half of the configurations of a system, and most of the performance changes (91%) affect multiple options (up to 6). Notably, despite the prevalence of performance changes, the performance ranking of configurations and influences of individual options are in many cases not affected. That is, developers and users can assume a certain stability of configuration-dependent performance behavior. About 43% of the performance changes are documented in change logs, 64% in commit messages. Specific configuration options were mentioned in 67% of the cases.
Our results have direct implications for configuration sampling, performance modeling, and transfer learning in the area of configurable software systems. That is, for instance, some performance changes affect only 1% of the configurations and demand for comprehensive performance measurements to spot performance changes. Additionally, we found that the relative influence of configuration options and interactions on the performance is stable in 80% of the releases. That is, performance engineers can assume a certain stability also on the options' influences while performing transfer learning across different releases (Jamshidi et al., 2017). A deeper analysis of change logs and commit messages shows that using a configuration-aware performance testing pipeline could help in identifying configuration-specific performance changes early. Our measurement and analysis framework provides a solid foundation for further experiments on different software systems and non-functional properties. All results along with analysis scripts and further information are available at a supplementary website[1] .

2 Preliminaries

2.1 Configurable Software Systems

A configurable software system offers a set $\mathcal{O}$ of configuration options, each of which can be selected or deselected.[2] $\mathcal{C}$ denotes the set of valid configurations, where $c \in \mathcal{C}$ represents a single configuration represented as a function $c : \mathcal{O} \rightarrow \{0, 1\}$, which assigns to each configuration option $o \in \mathcal{O}$ either 1 if it is selected in configuration c or 0 if not. For illustration, we show in Table 1 the configurations of a compression tool with four

[1] https://github.com/ChristianKaltenecker/PerformanceEvolution_Website

[2] Numeric configuration options can be discretized or explicitly represented (Siegmund et al., 2015); most options in our experiments are binary, the numeric options are discretized. This, however does not change the configuration space, only its representation and enables us to learn step functions as they typically appear in configurable software systems (Oh et al., 2017).

configuration options: Encryption (E), Compression (C), and two alternative compression algorithms, gzip (G) and ZPAQ (Z).
Note that not all combinations of configuration options $o \in \mathcal{O}$ are valid (i.e., $|\mathcal{C}| < 2^{|\mathcal{O}|}$), due to constraints among configuration options. In our example, exactly one compression algorithm, either gzip or ZPAQ has to be selected if Compression is selected; none of them can be selected if Compression is deselected.

Table 1: All valid configurations and their predicted performance values of a compression tool with three configuration options: Encryption (E),Compression (C), gzip (G), and ZPAQ (Z).

Configuration	$c(\mathsf{E})$	$c(\mathsf{C})$	$c(\mathsf{G})$	$c(\mathsf{Z})$	$\prod(c)$
c_1	0	0	0	0	10
c_2	0	1	1	0	5
c_3	0	1	0	1	25
c_4	1	0	0	0	30
c_5	1	1	1	0	20
c_6	1	1	0	1	45

2.2 Performance-Influence Models

Performance-influence models allow us to model and predict the performance of all individual configurations of a configurable software system (Siegmund et al., 2015). The resulting performance-influence model is a polynomial in which each additive term consists of a coefficient that describes either the base performance, the influence of a single configuration option (denoted as ϕ), or an interaction among multiple options (denoted as ψ) on the performance of the system. We denote a performance-influence model as a function $\prod : \mathcal{C} \rightarrow \mathbb{R}$, which takes a configuration $c \in \mathcal{C}$ and returns its predicted performance value. For illustration, consider the configurable system from Table 1. A corresponding performance-influence model could be as follows:

$$\prod(c) = 10 + \overbrace{20 \cdot c(\mathsf{E})}^{\phi_\mathsf{E}} + \overbrace{15 \cdot c(\mathsf{Z})}^{\phi_\mathsf{Z}} - \overbrace{5 \cdot c(\mathsf{G})}^{\phi_\mathsf{G}} - \overbrace{5 \cdot c(\mathsf{E}) \cdot c(\mathsf{G})}^{\psi_{\mathsf{E},\mathsf{G}}}$$

Notice that influences may be positive, negative, or negligible (close to 0). In our example, E increases the execution time by 20 (ϕ_E), whereas G decreases the execution time by 5 (ϕ_E). Only if both E and G are selected, the system is additionally speeded up by 5, which is effectively an interaction between two configuration options ($\psi_{\mathsf{E},\mathsf{G}}$). The configuration-independent base performance is denoted by the polynomial's intercept 10.
In general, performance models are of the following form:

$$\prod(c) = \overbrace{\beta_0}^{\text{Base}} + \overbrace{\sum_{o \in \mathcal{O}} \beta_o \cdot c(o)}^{\text{Option influences}} + \overbrace{\sum_{o_1..o_i \in \mathcal{O}} \beta_{o_1..o_i} \cdot c(o_1) \cdot \ldots \cdot c(o_i)}^{\text{Interaction influences}}$$

To obtain performance-influence models, we use multiple linear regression with feature forward selection (Andrews, 1974; Kuhn and Johnson, 2013). The underlying problem of multiple linear regression is to solve the following equation:

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}$$

where $\mathbf{X}$ denotes the input matrix in which each row corresponds to a configuration and each column represents a configuration option or interaction. $\boldsymbol{\beta}$ is a vector that encodes the influences of the configuration options and interactions; $\boldsymbol{\varepsilon}$ is a vector containing the prediction errors. Finally, $\mathbf{y}$ is a vector containing our dependent variable (i.e., our performance measurement results). The objective of multiple linear regression is to fit the vector $\boldsymbol{\beta}$ such that the error $\boldsymbol{\varepsilon}$ is minimal.
For further illustration, we use the example from Table 1 to fill the equation:

$$\begin{array}{c} \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \end{array} \begin{array}{c} \begin{array}{ccccc} \mathsf{Base} & \mathsf{E} & \mathsf{C} & \mathsf{G} & \mathsf{Z} \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \end{pmatrix} \end{array} \cdot \begin{pmatrix} \beta_0 \\ \beta_{\mathsf{E}} \\ \beta_{\mathsf{C}} \\ \beta_{\mathsf{G}} \\ \beta_{\mathsf{Z}} \end{pmatrix} + \begin{pmatrix} \varepsilon_{c_1} \\ \varepsilon_{c_2} \\ \varepsilon_{c_3} \\ \varepsilon_{c_4} \\ \varepsilon_{c_5} \\ \varepsilon_{c_6} \end{pmatrix} = \begin{pmatrix} 10 \\ 5 \\ 25 \\ 30 \\ 20 \\ 45 \end{pmatrix} \tag{1}$$

In this example, we have encoded only the base influence and all individual options, but not interactions. To also support interactions, the columns of the matrix C' and the vector β' in Equation 1 have to be expanded accordingly.

Algorithm 1: *Learning a performance-influence model*

```
1  Function learn_model (feature_model, performance_data) :
2      error ← ∞
3      error_reduction ← ∞
4      model ← ∅
5      while error > 1% and error_reduction > 0.1% do
6          candidates ← create_candidates(model, feature_model)
7          best_candidate_model ← ∅
8          best_candidate_error ← ∞
9          foreach candidate ∈ candidates do
10             candidate_model, candidate_error ← fit_and_predict(candidate, performance_data)
11             if candidate_error < best_candidate_error then
12                 best_candidate_error ← candidate_error
13                 best_candidate_model ← candidate_model
14             end
15         end
16         model ← best_candidate_model
17         error_reduction ← error − best_candidate_error
18         error ← best_candidate_error
19     end
20     model ← backward_selection(model)
21     return model, error
```

The overall idea of learning a performance-influence model is to refine a model iteratively until a user-defined threshold is reached (Siegmund et al., 2015; Kolesnikov et al., 2019b), as defined in Algorithm 1. Function `learn_model` receives the performance data and the feature model (i.e., information about the configuration options) as input. In Lines 2–3, we initialize two variables, prediction error and the error improvement, which are used to check against the threshold for aborting the learning process. Lines 5–19 contain the iterative procedure to perform multiple linear regression with feature forward selection (Andrews, 1974; Kuhn and Johnson, 2013). Therein, a list of different candidates (or features) is created in each step(Line 6). Each individual configuration option is a suitable candidate and so are interactions of configuration options with options that have already been added to the model. For instance, if a model contains the configuration option E, then also interactions with E

such as $E \cdot C$, $E \cdot G$, or $E \cdot Z$ become candidates. The rationale of this iterative extension of the model is to counter the combinatorial explosion of combining all configuration options. This iterative approach is hierarchical in that it can add interactions for only those options that have been found in reducing the model error in prior iterations. For instance, if $E \cdot G$ interact with each other, the approach would firstly include either E or G into the model and, in a later iteration, $E \cdot G$ if both together would reduce the prediction error for a hold-out set. After creating the candidates, each candidate is evaluated within a model that represents the state of the prior iteration (Lines 9–15). To this end, we first fit the model to the performance data of a hold-out set (see Equation 1) in Line 10 returning the model including the candidate and the overall error of the corresponding model. In Lines 11–14, the current candidate is selected as the best candidate if it reduces the error more than previous candidates. The best candidate of the current iteration is added to the model in Line 16. The reduction of the error resulting from the newly added candidate and the new error are then calculated. Note that choosing the best candidate represents a limitation of our approach since a worse performing candidate could lead to a better reduction of the error in future iterations. This iterative process is continued until one of the thresholds in Line 5 is no longer satisfied. Due to its hierarchical nature, the model can potentially include configuration options or interactions that may become irrelevant in later iterations. For instance, if only the interaction $E \cdot G$ is relevant for performance but the individual configuration options E and G are not, this approach would still include, at least, E or G as it reduced for some configuration the prediction error in previous iterations. To remove such unnecessary options and interactions, we apply a backward selection in Line 20. The backward selection removes all options and interactions that no longer improve the model error.

In many cases, it is desirable that a performance-influence model contains only the most relevant influences, which can be achieved by adjusting the learning procedure at the cost of predictive power (Kolesnikov et al., 2019b). In any case, predictions of performance-influence models are rarely totally accurate, even if we included all possible configurations for learning the performance-influence models. To some extent, the measurement setup introduces systematic error, resulting in noisy data.

Performance-influence models are not specific to execution time. They can be used to model any non-functional property that can be quantified on an interval scale. Performance-influence models have been applied to accurately predict execution time, throughput, memory consumption, binary footprint, energy consumption, verification effort, and more (Siegmund et al., 2013; Knüppel et al., 2018; Grebhahn et al., 2017). We selected performance-influence models for our empirical study since their additive structure makes them easy to interpret and compare. Typically, a performance-influence model is learned based on a sample set of configurations and used for performance prediction. However, instead of using them for predictions, in this paper, we use performance-influence models to explain which configurations or interactions thereof are affected by a performance change. For this purpose, we learn a performance-influence model based on the whole configuration space so that we obtain an accurate picture of the performance influences of configuration options and interactions thereof. In the past, Kolesnikov et al. (2019b) and Grebhahn et al. (2017) have successfully applied this approach for understanding and verifying the influence of configuration options and interactions on performance, respectively.

2.3 Software Evolution

Version control systems help developers to keep track of code changes that arise during software evolution. For this purpose, most version control systems provide the concept of revisions. A *revision* is effectively a view on the code base at a certain point in time. In what follows, $\mathcal{RV}$ denotes the set of revisions of a software system. To highlight revisions that (1) contain prominent changes, (2) are assumed as running stable, or (3) mark major milestones, a revision can be tagged as *release*, with $\mathcal{R} \subseteq \mathcal{RV}$ denoting the set of releases. In our study, we consider only releases (1) to focus on important revisions, (2) to keep measurement effort feasible, and (3) releases are usually the revisions that are used in production. Intermediate revisions are not guaranteed to compile/run without errors since those revisions typically are incremental modifications and "work in progress". Further, we measure not all, but only certain releases. The rationale behind this is that older software versions do not compile and run anymore on current operating systems, which limits the time span that we can observe. Furthermore, we do not measure each minor release in each software system since measuring each release would require to measure all configurations of the configuration space again. In this case, we opted to distribute the releases in similar time frames (e.g., one release per half year) to cover each time frame equally.

2.4 Multicollinearity

Multicollinearity is one of the biggest challenges in regression analysis and refers to a situation, in which a term of a linear model can be linearly predicted by other terms. That is, multiple terms represent the same effect such that it becomes unclear, which of these terms has the true influence on the independent variable and to what extent.
For a comprehensive and an unambiguous analysis of a software system's evolution, we have to assure that the terms of our models are not multicollinear. Otherwise, we can end up with different performance-influence models all predicting the same value, but with diverging influences of options and interactions, threatening internal validity of our analysis. As a countermeasure, one can apply a variance inflation factor (VIF) analysis (James et al., 2013; Dorn et al., 2023) and exclude terms that can be completely linearly predicted by other terms. For illustration, consider Table 1 and the following performance-influence models:

$$\prod\nolimits_1(c) = 10 + 15 \cdot c(\mathsf{C}) \cdot c(\mathsf{Z})$$
$$\prod\nolimits_2(c) = 10 + 15 \cdot c(\mathsf{Z})$$

Both performance-influence models predict the same performance values. The terms $c(\mathsf{Z})$ and $c(\mathsf{C}) \cdot c(\mathsf{Z})$ are perfectly multicollinear because when Z is selected in a configuration, C is also always selected. Hence, we cannot distinguish the influence of the interaction $c(\mathsf{C}) \cdot c(\mathsf{Z})$ from the influence of the option $c(\mathsf{Z})$. Having both terms in a performance-influence models would cause infinite possibilities of assigning coefficients to these terms, as demonstrated here:

$$\prod\nolimits_1'(c) = 10 - 10 \cdot c(\mathsf{C}) \cdot c(\mathsf{Z}) + 25 \cdot c(\mathsf{Z})$$
$$\prod\nolimits_2'(c) = 10 + 10 \cdot c(\mathsf{C}) \cdot c(\mathsf{Z}) + 5 \cdot c(\mathsf{Z})$$

Again, both performance-influence models make the same predictions but assign completely different coefficients to the terms. The VIF analysis detects such cases and declares the terms as multicollinear[3].

Algorithm 2: *Learning a performance-influence model*

```
1  Function learn_comparable_models (feature_model, releases, release_performance_data) :
2      terms ← ∅
3      foreach release ∈ releases do
4          model ← learn_model(feature_model, release_performance_data[release])
5          terms ← include_terms_from_model(terms, model)
6      end
7      terms ← variance_factor_analysis(terms)
8      models ← ∅
9      foreach release ∈ releases do
10         model ← fit(terms, release_performance_data[release])
11         models ← models ∪ {model}
12     end
13     return models
```

In our empirical study, we follow the approach of Algorithm 2 to bring the performance-influence models into a comparable form (i.e., all performance-influence models contain the same terms). In Lines 2–6, we learn a performance-influence model for each release. This is necessary to identify the performance-relevant configuration options and interactions. These configuration options and interactions are included as *terms* into the model in Line 5. This way, we obtain a set containing all relevant configuration options and interactions among them. However, this set cannot be immediately used as a performance-influence model since this step includes configuration options and interactions that might be multicollinear. Hence, we remove multicollinear terms by applying a VIF analysis (Dorn et al., 2023) in Line 7. Note that this does not affect our prediction error since we remove only perfectly multicollinear terms (i.e., terms that are completely interchangeable). After this step, we use the same terms and fit them for each release in Line 10. These performance-influence models contain the same configuration options and interactions and can now be compared.

3 Study Setup

In this section, we discuss our research questions and how we attempt to answer them by analyzing 12 subject systems.

3.1 Research Questions

Our overarching goal is to understand the performance evolution of configurable software systems. To this end, we study the characteristics of performance changes and their relation to configurability. For a detailed analysis, we consider two levels of abstraction: *configuration level* and *option level*.

[3] Note that we exclude only perfectly multicollinear terms, since perfect multicollinear terms are completely interchangeable. We do not use any threshold such as 5 as commonly used in literature because configuration options and their interactions can always be multicollinear to some extent due to overlap.

Configuration level As a first approximation, we address our goal at the level of individual configurations. In particular, we are interested in (1) whether performance changes affect typically many or only a few configurations and (2) whether performance changes alter typically the overall ranking of configurations with regard to their performance optimality.
For the first research question ($RQ_{1.1}$), we compare for each pair of releases each configuration with its successor in terms of the extent to which the performance has changed. This will allow us to make quantitative statements about how many performance changes exist in practice and what fractions and kinds of configurations are affected. These insights can inform sampling strategies and maintenance activities by prioritizing specific configurations that likely exhibit performance changes.

RESEARCH QUESTION 1.1
What is the fraction of the configuration space containing performance changes between consecutive releases?

For the second research question ($RQ_{1.2}$), we analyze to what extent performance changes affect the ranking of configurations with regard to their performance. That is, the slowest configuration has the lowest rank, the fastest configuration the highest rank, etc. Often developers and users are less interested in the actual performance values, but rather in their *relative importance*, including which configurations are performance-optimal and which fall below a certain threshold (Nair et al., 2017). It might be that performance changes exist but that most of them do not alter the performance ranking of configurations. That is, the *performance ranking* of configurations is *stable*. This would be useful for researchers (e.g., for transfer learning of performance models (Jamshidi et al., 2017, 2018)) and practitioners (so they can rely on a certain stability in the relative performance influences).

RESEARCH QUESTION 1.2
How stable is the relative performance of configurations in the presence of performance changes between consecutive releases?

Option level Beside knowing which configurations are affected by a performance change, we would like to know which configuration options or interactions among options are responsible for this change. As with configurations, we are interested in (1) whether typically many or only few options or interactions cause performance changes and (2) whether performance changes alter typically the overall ranking of performance influences of options and interaction. To obtain information on the influences of options and their interactions, we learn a performance-influence model per release and compare their terms and coefficients (see Section 2). Since we use linear regression to learn our performance models, multicollinearity might occur between multiple terms (see Section 2.4). As a countermeasure, we apply a VIF analysis and remove all terms causing perfect multicollinearity. By doing so, 9 out of 707 terms were removed leaving the predictions of our performance-influence models unaffected.
For the first research question ($RQ_{2.1}$), we compare for each pair of releases each influence of each model term with its successor regarding the extent to which its influence has changed. This will allow us to make quantitative statements about how many options and interactions are responsible for performance changes. Knowing whether many or only few options are responsible for performance changes helps to understand root causes of these changes and to guide corresponding actions. Identifying patterns here can inform performance engineers to guide and improve the detection and tracing of performance bottlenecks (Gahvari et al., 2011). Comparing each pair of releases further gives us the opportunity to assess the distribution

of relative influences of the configuration options on performance (i.e., all options have a similar influence on performance, or a few influence performance the most).

RESEARCH QUESTION 2.1
How frequent and how strong are changes of performance influences of individual configuration options and interactions between consecutive releases?

For the second research question ($RQ_{2.2}$), we analyze to what extent performance changes affect the global ranking of performance influences of configuration options and interactions. As with configurations, it is often sufficient to know which configuration options have a strong influence on performance without knowing exact performance values. For instance, when optimizing for performance, a user may concentrate on the configuration options having a strong influence on performance and ignore others (Xu et al., 2015). When optimizing for performance in a compression software, the performance-influence model might point out to consider low instead of high compression levels and to neglect debug options. For a developer, it might be interesting to confirm own expectations of how configuration options perform, as shown in a former study (Grebhahn et al., 2017).

RESEARCH QUESTION 2.2
How stable is the relative influence of configuration options and interactions in the presence of performance changes between consecutive releases?

Table 2: Overview of the subject systems, including application domain, lines of code (LOC) in the last measured release, number of valid configurations ($|\mathcal{C}|$) in each release, configuration options ($|\mathcal{O}|$), releases ($|\mathcal{R}|$), and performance metric.

Name	Domain	LOC	$\lvert\mathcal{C}\rvert$	$\lvert\mathcal{O}\rvert$	$\lvert\mathcal{R}\rvert$	Performance Metric
BROTLI	Compression	$30k$	181	30	12	Compression time
FAST DOWNWARD	Planning system	$90k$	374	39	9	Solving time
HSQLDB	Database	$194k$	864	29	19	Response time
LRZIP	Compression	$16k$	1 440	27	22	Compression time
MARIADB	Database	1 $969k$	972	21	22	Response time
MYSQL	Database	2 $792k$	972	21	20	Response time
OPENVPN	VPN software	$80k$	512	24	12	Response time
OPUS	Audio encoder	$54k$	6 480	31	12	Encoding time
POSTGRESQL	Database	1 $160k$	864	18	22	Response time
VP8	Video encoder	$324k$	2 736	27	15	Encoding time
VP9	Video encoder	$324k$	3 008	25	7	Encoding time
Z3	Constraint solver	$415k$	1 024	13	18	Solving time

3.2 Subject Systems

For our experiments, we selected 12 real-world configurable software systems based on the following criteria: (1) different sizes (number of configurations and configuration options) to evaluate scalability, (2) different application domains to increase external validity, (3)

different application architectures (e.g., client-server vs. desktop) to cover different performance aspects, and (4) actively maintained systems to detect historical changes in a realistic context; see Table 2, for an overview. As of 2023, all systems in our selection are actively maintained, and we consider lifetimes of 21 months (POSTGRESQL) to 137 months (OPENVPN). From the respective development histories, we extracted all releases, which we identified based on GIT tags and respective documentation. All considered configuration options represent run-time configuration options. We provide all variability models, selected releases, measurements, results from our deeper analysis, and a complete description of the configuration options on our supplementary website. It is important to note that we carried out the performance measurements on multiple machines in parallel to keep the measurement time manageable. While we use different machines across different subject systems, we use equally equipped machines for the measurements of each subject system. Parallelizing our performance measurements this way was possible, since we only compare revisions and configurations in subject systems and not across subject systems.

BROTLI is an open-source file compression tool by Google written in C. We considered 30 configuration options that give rise to 181 configurations, including configuration options setting the window size and compression level. We used UIQ2[4] to generate a general workload for compression (see Section 3.3 for more detail). As performance measure, we used compression time. The measurements took place on machines with Intel Core i7-4790 CPUs at 3.60 GHz with 16 GiB RAM (Debian 9). Overall, we considered 12 releases, from release 0.3.0 to 1.0.7, covering almost 3 years of history.

FAST DOWNWARD is an open-source domain-independent planning system for optimization. To identify performance-relevant configuration options and a proper workload, we contacted a domain expert. Based on the feedback, we considered 39 configuration options that give rise to 374 configurations. 7 out of 39 configuration options control different search heuristics; all other configuration options represent parameters for these heuristics. Here, we mainly consider different heuristics to solve the planning task. Each heuristic comes with its own parameters (i.e., configuration options). We measured the time to find an optimal solution for the planning task. All measurements were conducted on machines with Intel Xeon E5-2630 v4 at 2.20 GHz with 256 GiB RAM (Debian 11). Overall, we considered 9 revisions chosen in cooperation with the domain expert. In total, we cover 5 years of history.

HSQLDB is a lightweight database engine. We considered 29 configuration options that give rise to 864 configurations. Configuration options include support for different encryption algorithms, transaction control settings, and incremental backup. We measured throughput with the benchmarking tool POLEPOSITION[5]. We have used multiple thousands of read, insert, and update queries. We also considered nested queries. The tool emulates realistic user interaction by performing a number of insertions, deletions, updates, and queries. All measurements were conducted on machines with Intel Core i5-4590 CPUs at 3.30 GHz with 16 GiB RAM (Debian 9). Overall, we considered 19 releases, from release 2.1.0 to 2.4.1, covering over 7 years of history.

LRZIP is an open-source file compression tool. We considered 27 configuration options that give rise to 1,440 configurations. Relevant configuration options are, for instance, different compression algorithms, compression levels, and processor numbers. We used the same setup as for BROTLI. All measurements were conducted on machines with Intel Xeon E5-2650v2

[4] http://mattmahoney.net/dc/uiq/

[5] http://polepos.org

CPUs at 2.60 GHz with 128 GiB RAM (Debian 10). Overall, we considered 22 releases, from release 0.530 to 0.631, covering almost 6 years of history.

MARIADB and MYSQL are open-source relational database management systems. For both subject systems, we considered 21 configuration options that give rise to 972 configurations. Among others, we included different buffer pool sizes, table sizes, and flush methods. We measured throughput with the benchmarking tool POLEPOSITION. All measurements were conducted on machines with Intel Xeon E5-2650v2 CPUs at 2.60 GHz with 128 GiB RAM (Debian 10). For MARIADB, we considered 22 releases, from release 5.5.23 to 10.4.7, covering over 7 years of history. For MYSQL, we considered 20 releases, from release 5.6.10 to 8.0.17, covering over 6 years of history.

OPENVPN is an open-source software that provides secure communication between computers using virtual private networks. We considered 24 configuration options that give rise to 512 configurations. We included, for instance, support for compression, different encryption ciphers, authentication methods, and renegotiation settings. We set up an experiment with one client and one server exchanging files to measure the throughput of the application. All measurements of OPENVPN were conducted on machines with Intel Xeon E5-2650v2 CPUs at 2.60 GHz with 128 GiB RAM (Debian 10). Overall, we considered 22 releases, from release 2.1.0 to 2.4.6, covering over 11 years of history.

OPUS is a codec for lossy audio compression. We considered 31 configuration options, giving rise to 6 480 configurations. Configuration options include choices of bit rates, sample rates, and numbers of channels. We measured the performance of OPUS by repeatedly encoding a test vector, which has been used to validate the implementation against OPUS's file format specification. All measurements were conducted on machines with Intel Xeon E5-2620v4 CPUs at 2.10 GHz with 256 GiB RAM (Debian 10). Overall, we considered 12 releases, from release 1.0.0 to 1.3.1, covering almost 7 years of history.

POSTGRESQL is an open-source relational database management system. We considered 18 configuration options that give rise to 864 configurations. As configuration options, we include synchronous commits as well as different sizes of buffers and working memory. As with HSQLDB, we used the benchmarking tool POLEPOSITION for measurements. All measurements were conducted with machines with Intel Xeon E5-2650v2 CPUs at 2.60 GHz with 128 GiB RAM (Debian 9). Overall, we considered 22 releases, from release 9.6.3 to 11.2, covering almost 2 years of history.

VPXENC (VP8/VP9) is a video encoder that can be customized with different codecs, of which we study VP8 and VP9. We considered 27 and 25 configuration options that give rise to 2 736 and 3 008 configurations for VP8 and VP9, respectively. VPXENC provides a variety of configuration options, for instance, to adjust the quality or bitrate of the encoded video and multithreading operation. We used the raw trailer from the movie "Sintel" (480p, y4m format) as a benchmark and measured the encoding time of both codecs, respectively. VP8 was measured on machines with Intel Core i5-4590 CPUs at 3.30 GHz with 16 GiB RAM. VP9 was measured on machines with Intel Xeon E5-2650v2 CPUs at 2.60 GHz with 128 GiB RAM (Debian 10). For VP8, we considered 15 releases, from release 0.9.1 to 1.8.0, covering almost 9 years of history. For VP9, we considered 7 releases, from release 1.3.0 to 1.8.0, covering over 5 years of history.

Z3 is an open-source SMT solver from Microsoft Research. We considered 13 configuration options that give rise to 1 024 configurations. Configuration options include the generation of proofs, model validation, and model simplification. As a benchmark, we selected four sce-

narios from the International SMT Competition (LRA, QF_FP, QF_LRA, and QF_UFLRA). We measured and report the execution time for solving these tasks. Z3 was measured on machines with Intel Core i5-4590 CPUs at 3.30 GHz with 16 GiB RAM (Debian 11). Overall, we considered 18 releases, from release 4.3.2 to 4.8.13, covering more than 7 years of history.

3.3 Workloads

To obtain a representative workload and increase external validity (see Section 4.4 and Section 5) for each subject system, we selected one benchmark that originates from the respective system developers or community.

Audio Encoding (OPUS): For the audio encoding, we used test vectors provided by the developers of OPUS[6]. Test vectors are designed to test all aspects of the implementation of the audio encoder.

Compression (BROTLI/LRZIP): We used the tool UIQ2[7] to generate a large text compression workload. It creates a generic and general purpose compression workload of a specified size. The generated data was the same for both subject systems and has a size of about 100 MB.

Database (HSQLDB/MARIADB/MYSQL/POSTGRESQL): Each of the database systems supports SQL queries. We used the SQL benchmark POLEPOSITION[8], which was also used in multiple publications (Pukall et al., 2013; van Zyl et al., 2006). The benchmark enables us to generate different types of queries, such as SELECT, UPDATE, nested queries, and complex queries.

Planning System (FAST DOWNWARD): We applied the workload DATA-NETWORK-OPT18-STRIPS/P05[9] that was suggested by an experienced user of Fast Downward as a general workload. In addition, this workload does not contain specific characteristics that make the benchmark unsolvable for certain heuristics.

Solver (Z3): We selected multiple benchmarks from the Satisfiability Modulo Theories Library[10] having different types of logics LRA, QF_FP, QF_LRA, and QF_UFLRA. These benchmarks cover floating point, linear real arithmetic, free sort and function symbols, formulas with and without quantifier, and satisfiable and unsatisfiable formulas, thus, covering a large range of options provided by Z3.

Video Encoding (VP8/VP9): We used the Sintel trailer as a well-established workload when assessing the quality of different encoders. The Sintel trailer is listed in the Xiph repository[11] and has been used in different publications (Seidel et al., 2013; Pereira et al., 2020).

VPN (OPENVPN): Similar to compression, we created a generic general purpose file using UIQ2 with a size of 1 400 MB. We opted for UIQ2 since it generates compression workloads for the LZO compression, which is a functionality enabled by an option in OPENVPN. We adjusted the size of the file as suggested by a community guide for performance testing[12].

[6] https://opus-codec.org/docs/opus_testvectors-rfc8251.tar.gz
[7] http://mattmahoney.net/dc/uiq/
[8] http://polepos.org/
[9] https://github.com/aibasel/downward-benchmarks/blob/master/data-network-opt18-strips/p05.pddl
[10] https://smtlib.cs.uiowa.edu/benchmarks.shtml
[11] https://media.xiph.org/
[12] https://community.openvpn.net/openvpn/wiki/PerformanceTesting#Testcases

3.4 Operationalization

To answer our research questions, for each release, (1) we measured all configurations of a subject system and (2) learn a performance-influence model on the entire set of configurations, resulting in one model per system and release. $\mathcal{S}$ refers to the set of subject systems. For a system $s \in S$, $\mathcal{C}_s$ refers to its set of configurations (see Section 2.1) and $\mathcal{R}_s$ to its set of releases. $\mathcal{M}_s^r : \mathcal{C}_s \to \mathbb{R}$ maps the configurations $c \in \mathcal{C}_s$ of release $r \in \mathcal{R}_s$ to their *measured* performance values in $\mathbb{R}$. Π_s^r denotes the performance-influence model for revision $r \in \mathcal{R}_s$ of system s.

Configuration level Conducting performance measurements on the history of a configurable software system raises the question of whether the addition and removal of configuration options across releases should be considered. To simplify the analysis, we resort to a fixed set of options that is available across all releases of a subject system. While this way we might miss some interesting cases, our data set is still large and diverse enough to answer reliably our research questions.
The independent variables for $\text{RQ}_{1.1}$ and $\text{RQ}_{1.2}$ are (1) the subject system s, (2) the release r, and (3) the configuration c. The dependent variable is the performance value $\mathcal{M}_s^r(c)$. A confounding factor is measurement noise caused by particularities of the hardware and software platform (Mytkowicz et al., 2009). To control for this factor, we measured all configurations multiple times (3 to 5 times depending on the subject system) until the coefficient of variation (i.e., standard deviation divided by the mean) of the repetitions is lower than 10%.
To answer $\text{RQ}_{1.1}$, we determine the performance values $\mathcal{M}_s^r(c)$ for each configuration $c \in \mathcal{C}_s$ and each release $r \in \mathcal{R}_s$. We consider a performance change between a configuration of two consecutive releases relevant if:

$$\left| \mathcal{M}_s^{r_i}(c) - \mathcal{M}_s^{r_{i+1}}(c) \right| > 2 \cdot \max\left(\text{sd}_s^{r_i}(c), \text{sd}_s^{r_{i+1}}(c)\right) \tag{2}$$

where $\text{sd}_s^r(c)$ denotes the standard deviation of performance values of a configuration across repeated measurements. In other words, if a performance change does not exceed twice the larger standard deviation of the two releases, it is not further considered. The rationale for this conservative threshold is to filter out measurement noise and tiny performance changes.

Table 3: All valid configurations of our exemplary system from Section 2.1, their predicted performance values for two different releases, and the performance ranking of the configurations of the exemplary compression tool. The last column indicates whether the performance change is relevant according to Equation 2.

	Release 1		Release 2		
Configuration	$\prod(c)$	Rank ($\prod(c)$)	$\prod'(c)$	Rank ($\prod'(c)$)	Relevant
c_1	10	2	10	2	✗
c_2	5	1	6	1	✓
c_3	25	4	25	4	✗
c_4	30	5	30	5	✗
c_5	20	3	21	3	✓
c_6	45	6	45	6	✗

To answer $\text{RQ}_{1.2}$, we rank the configurations of each release r_i by their performance value. For illustration, we show the performance ranking of our exemplary compression tool for

two releases in Table 3. c_2 represents the fastest configuration in both releases and c_6 the slowest configuration. Further, instead of directly comparing the rankings of two consecutive releases, we first filter out irrelevant performance changes according to our definition in Equation (2). That is, the ranking order of the second release is affected only by relevant changes. In Table 3, we show in the last column which configurations are relevant according to Equation 2, assuming a relative standard deviation of 1%. After filtering, the ranking of only c_2 and c_5 would be compared, resulting in a perfect correlation, since both configurations maintain their ranking in both releases (i.e., $c_2 < c_5$ holds).
To quantify the similarity of two rankings (i.e., the performance rankings of the configurations of the current and the previous release), we use the Kendall's Tau correlation coefficient (Kendall, 1938). A correlation value of 1 indicates perfect correlation, a value close to 0 means no correlation, and -1 indicates that the rankings are fully opposed (i.e., the configuration with the highest rank in release r_i has the lowest rank in release r_{i+1}, the configuration with the second highest rank in release r_i has the second lowest rank in release r_{i+1}, etc.). In other words, a high correlation indicates that the performance ranking of configurations remains stable across releases, whereas a low correlation indicates that the ranking changes considerably. We omit computing Kendall's Tau for releases where the rank changes for less than two configurations. Calculating the correlation of the relevant configurations in Table 3, we would obtain a perfect correlation of $\tau = 1.0$.

Option level In $\text{RQ}_{2.1}$ and $\text{RQ}_{2.2}$, we aim at identifying the configuration options and interactions that are responsible for the performance change that we observed at the configuration level. To identify changes of the performance influence of an individual configuration option or interaction, we build on previous work by Siegmund et al. (Siegmund et al., 2015): We use *multiple linear regression* with *feature forward selection* to create for each revision $r \in \mathcal{R}_s$ a performance-influence model Π_s^r of the form described in Section 2. Note that we do not follow a sample-based learning approach (i.e., one that uses only a subset of configurations). Instead, we learn models on the whole configuration space. This would be impractical in practice but gives us the most accurate results. So, the independent variables for $\text{RQ}_{2.1}$ and $\text{RQ}_{2.2}$ are (1) the subject system s and (2) the release r; the dependent variable is the corresponding performance-influence model Π_s^r for $r \in \mathcal{R}_s$.
To answer $\text{RQ}_{2.1}$, we determine for each $r \in \mathcal{R}_s$ the performance influences $\beta_s^r(t)$ of all terms $t \in \Pi_s^r$. A term can either consist of the base term (i.e., β_0 in Section 2.2), a configuration option (i.e., $\beta_o \cdot c(o)$ for $o \in \mathcal{O}$), or an interaction among multiple options (i.e., $\beta_{o_1..o_i} \cdot c(o_1) \cdot \dots \cdot c(o_i)$ for $o_1, \dots, o_i \in \mathcal{O}$). Function $\beta_s^r(t)$ returns the coefficient of the term. Similar to $\text{RQ}_{1.1}$, we consider a performance change between two coefficients relevant if:

$$\left| \beta_s^{r_i}(t) - \beta_s^{r_{i+1}}(t) \right| > 2 \cdot \max(\overline{\text{sd}}_s^{r_i}, \overline{\text{sd}}_s^{r_{i+1}}).$$

where $\overline{\text{sd}}_s^{r_i}$ denotes the mean standard deviation of all configurations of release $r_i \in \mathcal{R}_s$. As with RQ1.1, if a change of performance influence does not exceed twice the larger average standard deviation of the two releases, it is not further considered. The rationale of using the *maximum* of the *mean* standard deviation is that we use the entire configuration space for learning performance models and thus accumulate the standard deviation over all configurations.
To answer $\text{RQ}_{2.2}$, we rank the terms $t \in \Pi_s^r$ based on their coefficients $\beta_s^r(t)$. Similar to $\text{RQ}_{1.2}$, the most influential term has the highest rank, the second most influential term has the second rank, and so on. As in $\text{RQ}_{1.2}$, we quantify to what extent the ranks between two releases r_i and r_{i+1} differ by using the Kendall's Tau correlation coefficient.

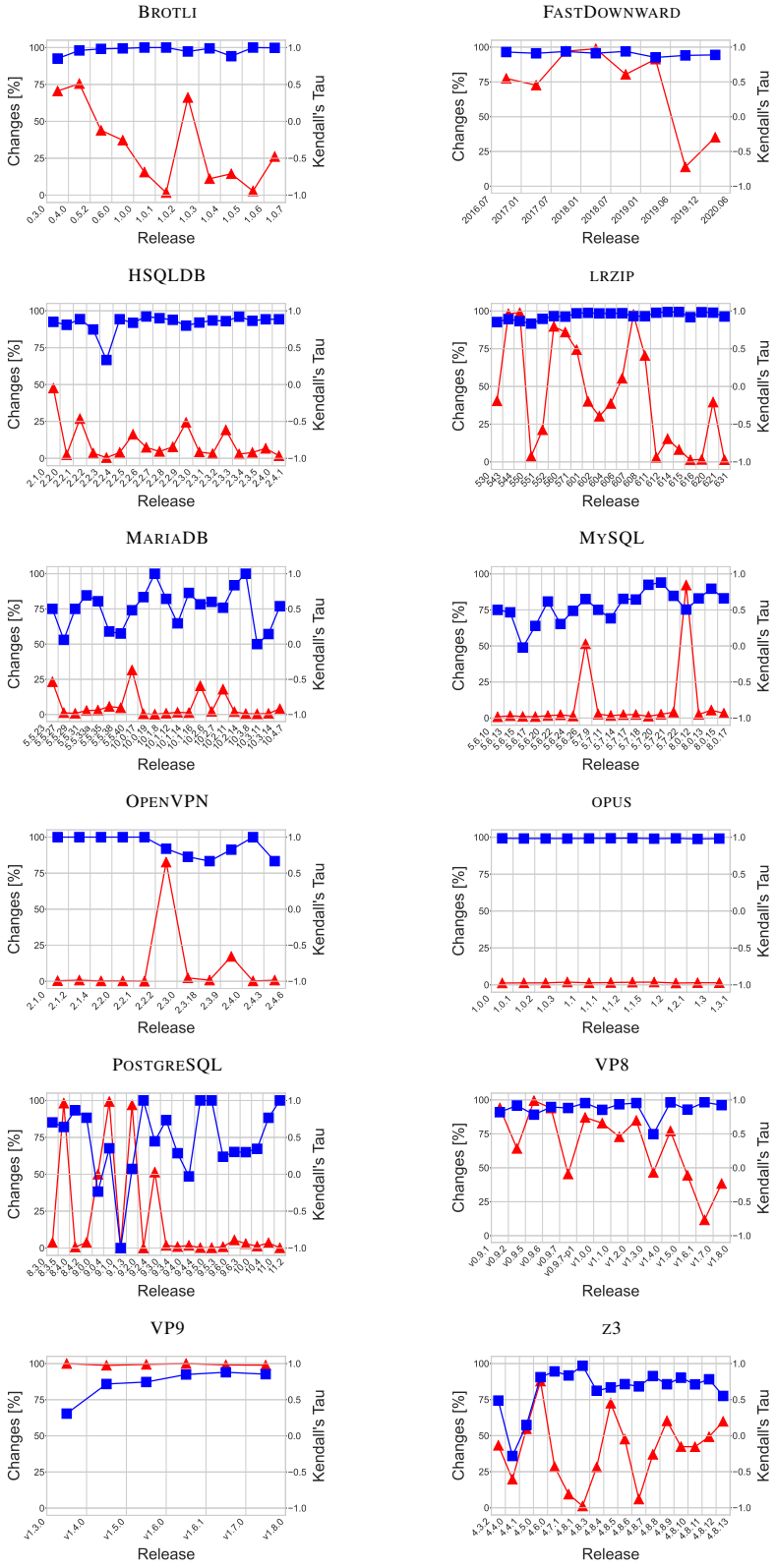

Fig. 1: Fraction of performance changes and stability of performance ranking at configuration level. The red line indicates the fraction of configurations of the whole configuration space containing performance changes (in %); the blue line indicates the stability of the ranked configuration performance as measured by Kendall's Tau.

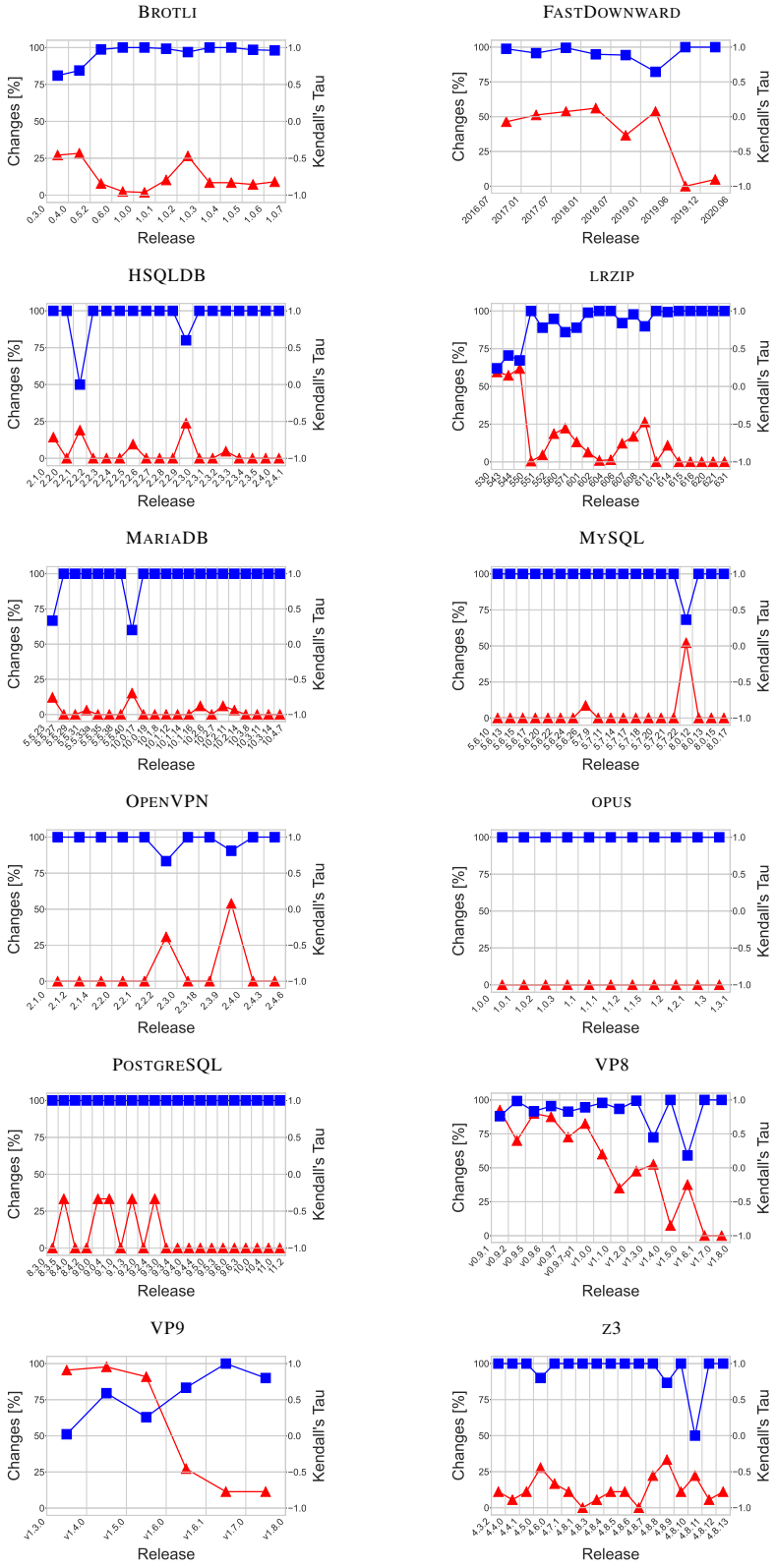

Fig. 2: Fraction of performance changes and stability of performance ranking at option level. The red line indicates the fraction of options containing performance changes (in %); the blue line indicates the stability of the ranked options performance as measured by Kendall's Tau.

4 Evaluation

In this section, we summarize our results (Section 4.1). We use these results in subsequent metadata analysis (Section 4.2) and discuss the results along with further observations (Section 4.3) and potential threats to validity (Section 4.4).

4.1 Results

In what follows, we refer to the plots given in Figure 1 and Figure 2. For each subject system, there is one plot per figure: the plots in Figure 1 show the number of changes (red line) and the stability of the performance ranking (blue line) at configuration level; and the plots in Figure 2 show the number of changes (red line) and performance ranking stability (blue line) at the option level.

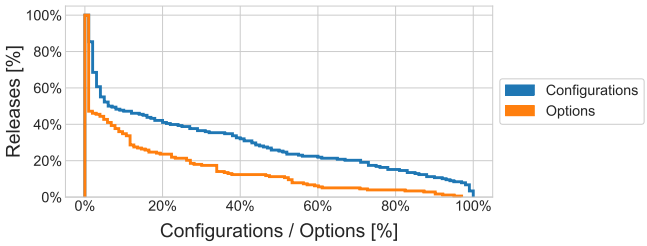

Fig. 3: Cumulative plot on the fraction of involved configurations (blue) or options (orange) in all performance changes of $RQ_{1.1}$ and $RQ_{2.1}$, respectively.

$RQ_{1.1}$: What is the fraction of the configuration space containing performance changes between consecutive releases?

In Figure 1, we show the fraction of configurations containing performance changes across consecutive releases (red lines)—the larger the value, the higher the fraction of configurations involved in a performance change. In Figure 3 (blue line), we provide a cumulative overview that shows how many of the 178 consecutive releases have a performance change in at least a certain fraction of configurations. For instance, we see that in more than 40% of the releases the performance changed in at least 20% of the configurations. Notably, 176 out of 178 (99%) releases have, at least, one configuration with a performance change.[13] Further, 2 (1%) performance changes are observed in the entire configuration space, 133 (75%) performance changes are observed in less than half of the configuration space, and 26 (15%) performance changes are observed only in 1% of the configuration space.
In Figure 4, we show the *intensity* of performance changes for VP9. Red color indicates performance degradation, blue color indicates performance improvement. For releases 1.4.0 and 1.6.0, we observe that the performance behavior of a considerable number of configurations (30%) of VP9 has changed substantially (i.e., the blue and the red colored configurations)—much more than our threshold of twice the standard deviation used in Figure 4.

[13] We have detected no configurations with performance changes between releases 9.2.0 and 9.2.4 of PostgreSQL and between releases 2.2.1 and 2.2.2 of OpenVPN.

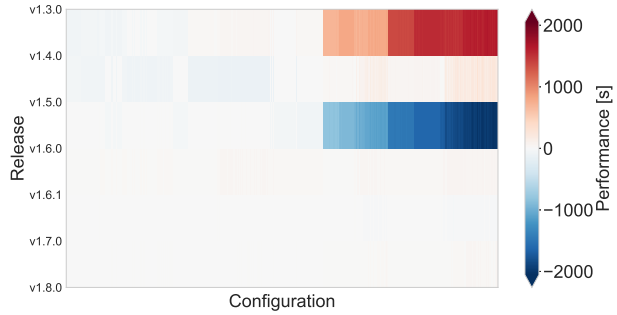

Fig. 4: Performance changes of VP9 across all configurations (x-axis) and releases (y-axis). We use a color palette to illustrate performance degradation (> 0, red) and performance improvement (< 0, blue). The configurations are sorted in ascending order according to their mean performance over all releases. There are 3 008 configurations on the x-axis; axis ticks have been omitted for readability.

SUMMARY $RQ_{1.1}$

Almost every release of every subject contains, at least, one performance change in some configuration. The majority of performance changes affects less than half of the configurations.

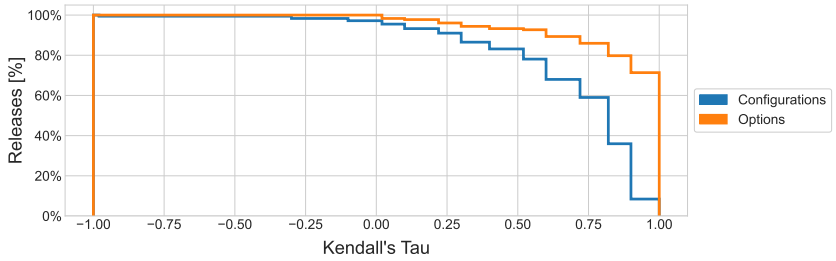

Fig. 5: Cumulative plot on the stability of configurations (blue) or options (orange) in all performance changes of $RQ_{1.2}$ and $RQ_{2.2}$, respectively.

$RQ_{1.2}$: How stable is the relative performance of configurations in the presence of performance changes between consecutive releases?

In Figure 2, we show the stability of the performance ranking of configurations, as quantified by Kendall's Tau (blue lines). A high value indicates high stability: the performance ranking of configurations changes only slightly (i.e., the fastest configurations stay the fastest, etc.). Across all systems and releases, the ranking is largely stable: $\overline{\tau} = 0.74$. In Figure 5, we provide an overview of the stability (blue line) between all 178 consecutive releases. 148 (83%) releases have a τ value higher than 0.5, 105 (59%) releases have a τ value higher than 0.80, and 64 (36%) releases have a τ value higher than 0.90. OPUS is most stable ($\overline{\tau} = 0.98$), POSTGRESQL is least stable ($\overline{\tau} = 0.36$).

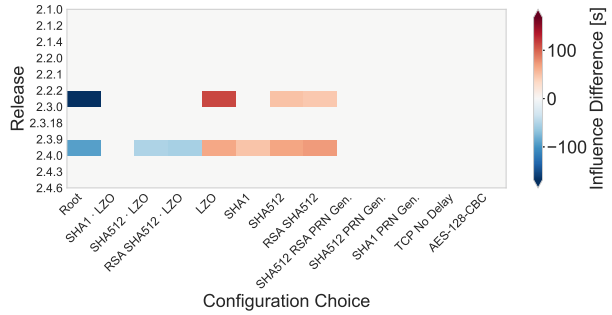

Fig. 6: Performance influence of options and interactions (x-axis) of OpenVPN across all releases (y-axis). A color palette illustrates performance degradation (> 0, red) and improvements (< 0, blue).

SUMMARY $RQ_{1.2}$

The performance ranking of configurations is largely stable across consecutive releases ($\overline{\tau} = 0.74$), with some notable exceptions.

$RQ_{2.1}$: How frequent and how strong are changes of performance influences of individual configuration options and interactions between consecutive releases?

In Figure 2, we show the fraction of how many options or interactions have changed from one release to another (red line). As explained in Section 3.4, the influences were determined by learning a performance-influence model per release. It is important to note that the prediction errors of the models were generally low (3.9%, on average), so we are confident that the influences are accurate.

Frequency: The fraction of configuration options and interactions involved in performance changes ranges from 0.45% (e.g., LRZIP) to 95% (e.g., VP8). In Figure 3 (orange line), we provide a cumulative overview that shows how many of the consecutive releases have a performance change in at least the certain fraction of configuration options. For instance, we see that about 12% of the consecutive releases indicate a change on more than 40% of the configuration options and interactions. On average, the influence of 28% of the configuration options and interactions change across all releases. While, in most of the changes (91%), multiple configurations options and interactions are involved, there are cases where just a single option is responsible for a performance change (PostgreSQL). Figure 6 shows the intensity of performance influences of individual configuration options and interaction for OpenVPN: In releases 2.3.0 and 2.3.9, we note substantial performance changes, each of which is caused by only a subset of options, some of which interact causing the effect (e.g., SHA512 and LZO).

Distribution: In Figure 7, we show the distribution of relative performance influences across all subject systems and releases. 83% of the model terms (options or interactions) have only a very small influence on performance (less than 7.5%), which is in line with theoretical considerations of influencing factors in sensitivity analysis (Saltelli, 2008); only 3% of the model terms have an influence of 80% and more on the system's performance. That is, the influence on the performance is mostly distributed over all configuration options and interactions. A notable exception is PostgreSQL, where only three terms are relevant,

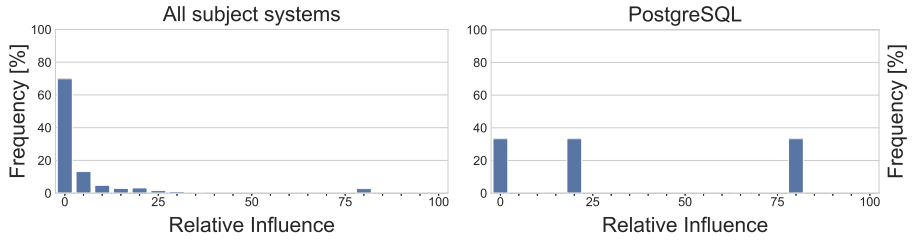

Fig. 7: Distribution of the relative influences of model terms across all subject systems (left) and for PostgreSQL (right).

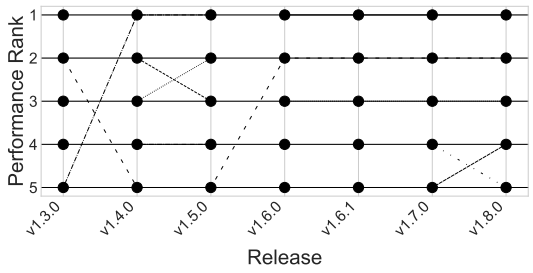

Fig. 8: Evolution of the performance ranking of the 5 most important model terms of VP9. Connected nodes illustrate the change of ranking from one release to another. An unconnected node means that the ranking in the next release is lower than 5.

namely the base term, fsync (which enables synchronized writes), and trackActivities (which enables the collection of information on the executed commands).

SUMMARY $RQ_{2.1}$

There is a substantial number of cases where influences of individual configuration options or interactions change across releases, but only few have a substantial influence on performance. Most performance changes (91%) are caused by multiple options and interactions, but there are cases where only a single option is responsible.

$RQ_{2.2}$: How stable is the relative influence of configuration options and interactions in the presence of performance changes between consecutive releases?

In Figure 2, we show the stability of the performance ranking of individual influences of options and interactions, as quantified by Kendall's Tau (blue lines). We included a cumulative overview in Figure 5 (orange line). In comparison to RQ1.2, stability is much higher: $\overline{\tau} = 0.91$. 151 (85%) have a τ larger than 0.8, and 142 releases (80%) have a τ larger than 0.9. For two subject systems (Opus and PostgreSQL), the performance ranking is stable across all releases. The performance model ranking (i.e., blue line of the right plot) of the consecutive releases 1.3.0 and 1.4.0 in VP9 contain slightly negative values, which indicate larger fluctuations and even a partial reversal of the ranking (see change of ranking of first and fourth options between 1.3.0 and 1.4.0 in Figure 8).
For illustration, we show in Figure 8 the evolution of the ranking of the 5 most influential configuration options or interactions of VP9. The ranking changes considerably over time, where

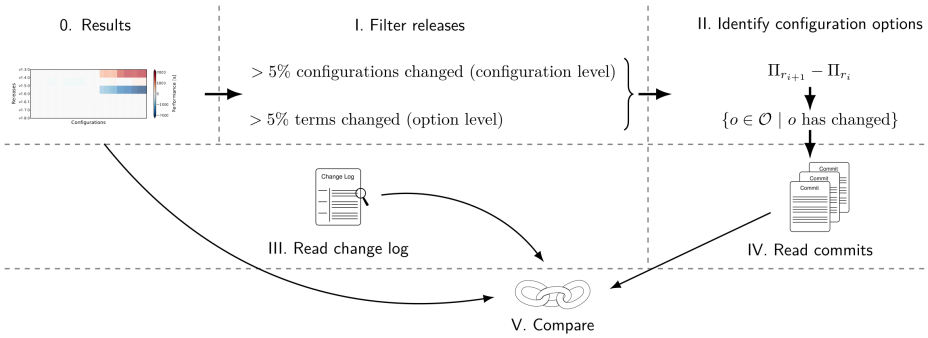

Fig. 9: Methodology of our deeper analysis. Step 0 includes our previously discussed results. In Step I, we select consecutive releases with certain degrees of performance change. Afterwards in Step II, we identify the configuration options with a changed performance influence from one release r_i to another r_{i+1}. In Step III, we read change logs for documented performance changes to find the cause and extract for each release whether performance changes were documented or not. In Step IV, we read commit messages of the relevant consecutive releases and include the changed configuration options from Step II to our analysis to aid finding the cause. In this step, we obtain for each release whether a performance change was documented in the commits and whether at least one affected configuration option was mentioned or not. Last, in Step V, we compare the results from Step 0, Step III, and Step IV. In particular, we show in which cases the change log and commit messages correspond or differ from our results and in which cases the configuration option is mentioned.

the most changes are in between 1.3.0 and 1.4.0. The reason is a performance regression in the options realtime and quality encoding, which was fixed in 1.6.0.

SUMMARY $RQ_{2.2}$

The performance ranking of influences of individual configuration options and interactions is largely stable across consecutive releases ($\overline{\tau} = 0.91$), with some exceptions.

4.2 Metadata Analysis

To triangulate the results of Section 4.1, we have conducted a deeper analysis that aligns the identified performance changes and influential model terms with reported cases in change logs and commit messages of the respective subject systems. In particular, we are interested in to what extent the learned performance models are able to pin down configuration options or interactions that are involved in a performance change.

Conduct In Figure 9, we show the steps of our deeper analysis. In Step I, we check the performance change of each consecutive release at the configuration level and the option level (see Figure 1 and Figure 2). We consider a release as relevant if the performance change at option or configuration level of one release exceeds 5% of the previous release. We exclude releases for which only the performance of the base program (i.e., the term base) has changed. There are two reasons for this: (1) a code change to the common base code affects all configurations; (2) a code change affects an option that is not included in our analysis.

For instance, changing the default value of an unconsidered configuration option (e.g., by enabling it by default) can be the reason for performance changes in base. This scenario occurred only in POSTGRESQL, in which in 4 out of 5 relevant releases, only the term base has a changed performance value.
Applying both filters, 79 out of 181 (43%) releases are relevant for our investigation. OPUS is the only subject system with no detectable performance changes. Thus, OPUS will not be considered in this analysis. By contrast, all releases of VP8 and VP9 are included in our analysis.
In Step II, we inspect performance-influence models of Section 4.1 in more depth to gather information on which configuration options and interactions thereof have actually changed. Based on this information, we search for documented performance changes in the entire change log between each pair of relevant consecutive releases including the change log for the current release for documented performance changes in Step III.
In Step IV, we analyze the commit messages between each pair of relevant consecutive releases. Fortunately, our selected subject systems are open source relying on publicly accessible version control systems (mostly git). Since reading all commit messages is infeasible for larger projects, we filter the commit messages using the following keywords similar to other studies (Jin et al., 2012; Chen et al., 2018): *slow*, *fast*, *time*, *perf*(ormance), *optim*(ize), and *regression*. Additionally, we added the name of the configuration options that we identified in Step II and check whether a configuration option is mentioned. If one of these keywords matches, we analyzed the commit message in detail.
Finally, in Step V, we contrast the obtained information by comparing them with each other. In particular, we report in how many cases the commit messages reported a performance change in comparison to the change log and in how many cases the configuration option was mentioned. For brevity, we provide only a summary of our analysis in Table 4; the full set of results is available on our supplementary website.
To reduce interpretation bias, the first and the second author performed the analysis of Step III and Step IV independently. After the analysis, they compared their results and discussed the differences to reach a consensus. Only in 3 pairs of releases of MARIADB, where the commit messages were larger than 10 MB, the third author checked and confirmed the results of the first author's manual analysis.

Results In Table 5, we list an excerpt of the results of our deeper analysis. We provide the complete list of results on our supplementary website[14]. Details on each result are also included on our supplementary website[15]. We show which of the consecutive releases have reported a speed-up or a slow-down in change logs or in commit messages, and whether the affected configuration option has been mentioned. The table also includes information which fractions of configurations improved or decreased performance.
7 out of 88 (8%) consecutive releases do not include a change log. In summary, in 35 out of 81 (43%) consecutive releases, the change log reported a performance change, whereas 2 reported a slow down and 33 a speed-up. In 56 pairs of releases (64%), the commit messages reported a performance change. Comparing change log and commit messages, we found that in 48 out of 81 (59%) consecutive releases, the change log and commit messages correspond to each other. In the remaining 33 consecutive releases (41%), 26 (32%) list other (and more)

[14] https://github.com/ChristianKaltenecker/PerformanceEvolution_Website/blob/master/MetadataAnalysis/AnalysisTable.md

[15] https://github.com/ChristianKaltenecker/PerformanceEvolution_Website/blob/master/MetadataAnalysis/MetadataAnalysis.md

Table 4: Overview of the number of relevant releases (RR) and releases reporting speed-ups (↑) or slow-downs (↓) in the change log and commit messages. The last column indicates the number of releases where at least one affected configuration option is mentioned in the commit messages.

System	|RR|	Change log		Commit		Option
		↑	↓	↑	↓	
BROTLI	9	2	0	4	0	5
FAST DOWNWARD	9	0	0	2	1	5
HSQLDB	8	4	0	1	0	1
LRZIP	15	7	0	7	1	9
MARIADB	5	2	0	5	0	5
MYSQL	3	2	0	3	0	3
OPENVPN	2	2	0	2	0	2
POSTGRESQL	1	0	0	0	0	1
VP8	14	7	0	10	0	12
VP9	6	6	0	6	1	5
Z3	16	1	2	15	2	11

Table 5: Excerpt of the 88 relevant consecutive releases, the fraction of sped up and slowed down configurations, whether speed-ups (↑) or slow-downs (↓) are mentioned in the change log or in the commit message, and whether changes in the identified options/interactions are reported.

System	Release	Speed-up (%)	Slow-down (%)	Change log	Commit	Option
BROTLI	0.3.0–0.4.0	54.4	16.1	↑	↑	✓
	0.4.0–0.5.2	66.1	9.4	✗	↑	✓
	0.5.2–0.6.0	36.1	7.7	↑	↑	✓
	0.6.0–1.0.0	10.0	27.2	✗	↑	✗
	1.0.2–1.0.3	51.6	14.4	✗	✗	✓
	1.0.6–1.0.7	26.1	0.0	✗	✗	✗
HSQLDB	2.1.0–2.2.0	0.3	47.2	↑	✗	✗
	2.2.1–2.2.2	26.6	0.2	↑	↑	✗
	2.2.5–2.2.6	0.9	15.2	✗	✗	✓
LRZIP	530–543	3.8	36.6	↑	↑	✗
	543–544	25.0	73.1	↑	↑	✓
	544–550	95.3	3.4	↑	↑	✓
	552–560	7.7	81.9	↑	↑	✗
	560–571	85.2	0.6	↑	↑↓	✓
MARIADB	5.5.23–5.5.27	22.5	0.7	↑	↑	✓
	5.5.35–5.5.38	1.6	4.0	✗	↑	✓
	5.5.40–10.0.17	5.9	25.6	↑	↑	✓
	10.1.16–10.2.6	5.9	25.6	✗	↑	✓
	10.2.7–10.2.11	17.8	0.1	✗	↑	✓
MYSQL	5.6.26–5.7.9	0.0	51.4	✗	↑	✓
	5.7.22–8.0.12	0.0	92.2	↑	↑	✓
	8.0.13–8.0.15	2.3	3.0	↑	↑	✓
POSTGRESQL	9.0.0–9.0.4	50.0	0.0	✗	✗	✓
VP8	1.3.0–1.4.0	40.3	6.2	✗	↑	✓
VP9	1.3.0–1.4.0	34.31	65.6	↑	↑↓	✓
	1.6.0–1.6.1	0.0	100.0	↑	↑	✓
Z3	4.8.7–4.8.8	8.1	28.9	↓	↑↓	✓
	4.8.8–4.8.9	16.2	44.0	↓	↑	✓

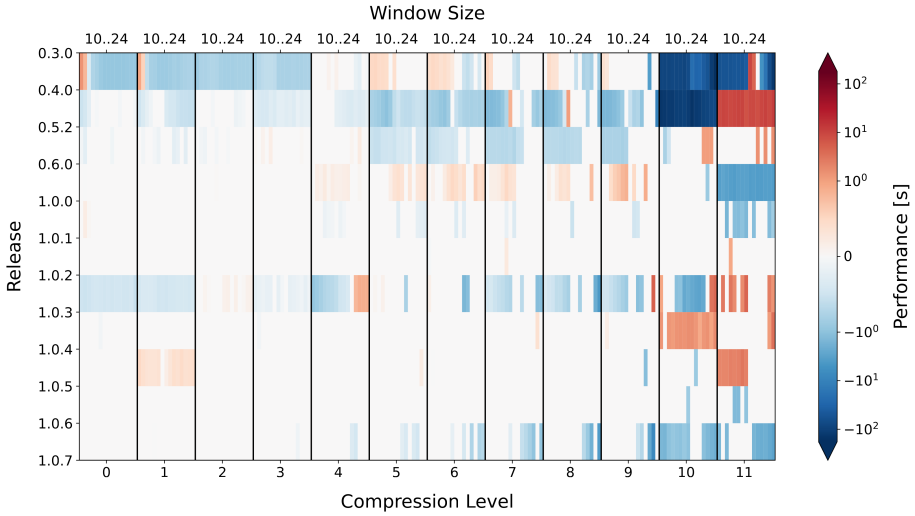

Fig. 10: Performance changes of BROTLI across all releases (y-axis). The color code highlights performance degradation (> 0, red) and performance improvement (< 0, blue).

performance-relevant information in the commit messages than in the change log. The change log delivers more performance-relevant information in only 5 consecutive releases (6%). In total, 60 out of 88 (68%) consecutive releases mention a performance change in the change log or commit message.
In 4 cases (5%), speed-ups and slow-downs were reported in commit messages. At least one affected configuration option was mentioned in 59 cases (67%), out of which 14 pairs of releases (16%) mention only changes in the configuration option's code base but no performance changes in the change log or commit messages. In 29 of the cases (33%), no affected configuration option is mentioned. Moreover, in 7 cases (8%), some configurations show a minor but relevant performance change while the performance-influence model does not (i.e., the performance-influence models are similar in these cases). In 12 cases (15%), the change log or commit messages report speed-ups without mentioning a configuration option.

Details To provide in-depth insight into our deeper analysis, we show in Figure 10 the configuration options WindowSize and CompressionLevel of BROTLI which control the compression rate of files. A blue color represents performance increase and a red color a decrease from one release to another. In the first pair of consecutive releases, 0.3.0 – 0.4.0, an increase in performance of compression levels $0 - 3$ can be observed, which is also mentioned in the change log and the commit message. However, the speed-up of compression levels 10 and 11 are not directly mentioned and may be a product of memory improvements, which was another focus of release 0.4.0. In release 0.5.2, the performance is improved for 66% of the configurations, which is not mentioned in the change log. One commit message, however, addresses speed and the affected configuration options: "*new hasher - improved speed, compression and reduced memory usage for q:5-9 w:10-16*"[16]
Note that q stands for compression level (or quality) and w for the window size. The slow-

[16] https://github.com/google/brotli/commit/2048189048

down in compression level 11, however, is not addressed until the next release 0.6.0 and mentioned there as fixed. We can see the fix for the compression level 11 only later in release 1.0.0. In release 0.6.0, the developer also report optimizations for mid-level compression levels (5–9). Another interesting pair are releases 1.0.2 and 1.0.3. Although more than half of the configurations experience a performance change in this range, there are no direct relations to these performance changes in the change log or the commit messages. Only a fix in compression level 10 is reported. The changes are a consequence of a new dictionary generator that was introduced in this release. In the latest release 1.0.7, where a quarter of the configurations was sped up but no configuration was slowed down, nothing relevant is reported in the change log and the commit messages. The changes focus on optimizations on the ARM architecture. Some of these changes may also affect the x86 architecture where our experiments were performed on.

Between releases 2.1.0 to 2.2.0 of HSQLDB in Table 5, we measured a slow-down in 47% of the configurations and a speed-up of only 0.3% of the configurations, whereas the change log reports only a speed-up. With the option-level analysis, we could relate the slow-down to the configuration option logSize, which controls the size of the log file before an automatic checkpoint occurs. A deeper analysis of commit messages did not confirm any evidence of a slow-down.

In Table 5, we show notable cases for LRZIP. In the pair 530–543, more than 36% of configurations show a slow-down and more than 3% show a speed-up. The change log and commit messages only mention the latter. In the option-level analysis, we find a slow-down in different compression algorithms, compression levels, and in multi-threading. The commit messages mention changes on multi-threading and compression algorithms, but in relation to decompression, which was not measured. In the next pair of releases, 543–544, we have a similar situation, with 73% of configurations showing a slow-down and 25% of the configurations showing a speed-up. According to our option-level analysis, similar configuration options as in the release pair 530–543 are affected. Commit messages report that the way how threads are spawned has been changed to improve the performance of compression[17]. However, this slow-down is addressed between 544–550, where the respective commit was completely reverted[18]. Another situation appears in the release pair 552–560. Change logs and commit messages report only speed-ups and no slow-downs. Again, multiple configuration options, such as compression algorithms, compression levels, and multi-threading are affected. Moreover, the commit messages do not mention any of the affected configuration options, only in relation to another operating system (Mac OSX). Later, in the release pair 560–571, more than 85% of the configurations are sped up and less than 1% have a slow-down. Both change log and commit messages report speed-ups in multi-threading, whereas only the commit messages also report a minor slow down.

MARIADB and MYSQL are also included in the excerpt in Table 5 since the first is a fork of the latter. Both projects use semantic versioning and introduce new functionality in new major releases that may break backward compatibility. In the major release of MARIADB between releases 5.5.40–10.0.17 and MYSQL between releases 5.7.22–8.0.12, the InnoDB engine was updated and, in the case of MYSQL, some refactoring was applied. Further refactoring of logging and binlogging was applied in MYSQL, between releases 5.6.26–5.7.9, which resulted in a slow-down. Releases 8.0.13–8.0.15 of MYSQL contain further bug fixes that result in speed-ups. Between releases 5.5.35–5.5.38, MARIADB applied several bug fixes and speed-up fixes. Later, between releases 10.1.16–10.2.6, the InnoDB engine was updated.

[17] https://github.com/ckolivas/lrzip/commit/688aa55c7930

[18] https://github.com/ckolivas/lrzip/commit/8dd9b00

Between releases 10.2.7–10.2.11, MARIADB reverted an InnoDB fix from MYSQL[19] and performed code optimization.
Interestingly, we observed that MARIADB and POSTGRESQL have the same fixes between releases 5.5.23–5.5.27 and 9.0.0–9.0.4, respectively. There, forcing $fdatasync$ for physical data synchronization on Linux causes an improvement in performance and assures that the files are synchronized on the physical storage, which is important for data recovery in case of system crashes. Interestingly, MARIADB reports speed-ups in the change log and commit messages, whereas POSTGRESQL does not.
Another interesting case in Table 5 includes VP8 and VP9. Both video encoders are developed in the same repository and VP9 represents the successor of VP8. The consequence is that the developers compare VP9 with its predecessor in terms of performance, which applies to the pair 1.3.0–1.4.0. There, the developers report a regression in the commit messages in comparison to VP8: "*Was 20% faster than speed -5 of vp8. Now 20% slower but adds motion search(...)*"[20]. This change demonstrates that VP9 comes with additional functionality at the cost of deviating from the performance of VP8. Interestingly, VP9 contains the single consecutive release 1.6.0–1.6.1 where all configurations indicate a slow down. To increase confidence in this particular findings, we have additionally executed all configurations of releases 1.6.0–1.6.1 on another current setup (i.e., another hardware and current operating system[21]) and were able to observe the slow-down too. The change log and the commit messages, however, report only speed-ups. Our performance-influence model related the changes to multiple configuration options and interactions, some of which are mentioned in the commit messages.
Z3 also contains pairs of consecutive releases (i.e., 4.8.7–4.8.8 and 4.8.8–4.8.9) where the developer reported a regression already in the change log and the commit message. The reason behind lies in nightly performance tests that are performed for Z3 on different platforms and, thus, the developers of Z3 are informed early about performance changes. However, the affected configuration options are not mentioned in these releases.

SUMMARY METADATA ANALYSIS

In most consecutive releases (68%), the developers mention performance changes in the change log or commit messages. In a similar amount of releases (67%), the developers mention the affected configuration option in the commit message, but there are cases (16%) where no performance change but changes in affected configuration options have been reported.

4.3 Implications

Insight: Need for prioritization of configurations for testing Our study shows that change in performance behavior is not the exception but the rule (i.e., 99% of the releases contain a performance change in $RQ_{1.1}$) as also confirmed by others (Jiang and Hassan, 2015; Mühlbauer et al., 2020). What is interesting is that most performance changes (78%) affect less than half of the configuration space and a non-negligible number (16%) only 1% of the configuration space. This is bad news for developers as, this way, performance problems are more difficult to spot with standard methods, such as testing default or random configurations

[19] https://github.com/MariaDB/server/commit/cb9648a6b5
[20] https://github.com/webmproject/libvpx/commit/ea8aaf15b55
[21] Intel Core i5-4590 CPU with 16 GiB RAM (Debian 11)

(we will get back to this shortly). Only in few (1%) cases, the whole configuration space is affected by a performance change, which is easy to discover by measuring the default configuration for instance. This result is notable and corroborates the need for performance modeling and testing methods that incorporate configurability. Random testing is unlikely sufficient to reveal cases where only few configurations are affected by a change. Furthermore, we found that, in 7% of the releases with a performance change, functional changes on the affected configuration options are reported but not observable with our models (i.e., a speed-up or slow-down). Combining configuration testing with performance modeling could help in such cases.

Insight: Mixed-strategy sampling Another notable result is that performance changes are often caused by *multiple* configuration options (i.e., in 91% of the changes in $RQ_{2.1}$). This includes (1) cases where the performance change is a *cumulation* of the individual influences of several options and (2) cases where multiple options *interact* and, this way, cause a performance change. Both cases are interesting as they demonstrate that configuration sampling methods based on simple structured coverage criteria (e.g., t-wise sampling) or simple random sampling are doomed to fail. The distribution of influences of options and interactions shows that only a *combination* of random and structured sampling methods is able to sufficiently cover the configuration space. That is, our results demonstrate that simple pair-wise sampling would miss many relevant interactions—in Z3, we found even a performance-relevant interaction among 6 configuration options! At the same time, pair-wise sampling would consider way too many pair-wise interactions that are irrelevant, rending the whole approach expensive or even intractable in practice (von Rhein et al., 2018). A random approach would likely miss important interactions, too. For example, in the case of POSTGRESQL, a single option is responsible for a substantial performance change between 9.0.0 and 9.0.4. Our results (in particular, distributions of influences) shall inform recent developments in combining structured and random sampling to improve sample quality and reduce cost. In the past, the application of such a combined sampling strategy, distance-based sampling, already outperformed other sampling strategies with regards to performance (Kaltenecker et al., 2019; Pereira et al., 2020).

Insight: Configuration sensitivity A further notable result is that, in about 80% of the releases (see $RQ_{2.2}$), the ranking of configuration options and interactions is stable ($\tau > 0.8$). This is good news, as developers and users can assume a certain stability of the relative performance of individual configurations. In other words, there is no immediate need for reconfiguring the system after a new release. However, there are exceptions such as POSTGRESQL, where the performance ranking changes considerably over time (see Figure 1). Knowing about this general behavior sheds light onto the *sensitivity* of the system's performance behavior on configuration. Our results suggest that this sensitivity varies across systems and developers need to know that for performance testing and tuning.
At the level of individual configuration options and influences, we observe a similar picture. The sensitivity of individual options regarding performance differs across systems and may change over time. An option that influences performance to a large extent in one release may have only a minor influence in the next release. This finding has implications for configuration sampling across revisions (Thüm et al., 2019) and transfer learning (Jamshidi et al., 2017): In both cases, a set of options is selected based on few revisions and then applied to other revisions (for further sampling or learning transfer). Our results indicate that this approach may work for most of the cases, but is too simplistic for the general case, as the set of relevant options and interactions may change considerably (e.g., VP9). For most cases nevertheless,

focusing on the configuration options or interactions with the highest influence could be a promising way when using sampling, since their relative influence remains largely the same.

Insight: Diverging performance behavior An interesting aspect of our selection of subjects is that VP8 and VP9 share some of their history and are still developed in the same repository. One might expect that this leads to similarities in performance behavior and evolution, since fixes and optimizations might be transferred easily. Our data do not confirm this expectation. On the contrary, we even found an opposing performance regression in 1.3.0–1.4.0: VP8 was sped up for 40% of the configurations and slowed down for only 6.2% of the configurations whereas VP9 shows a massive slow-down for 65.6% of the configurations. The same holds for MARIADB and MYSQL, where the first is a fork of the later. Both show different performance changes in their evolution. While this does not have to be a problem per se, our analysis framework provides proper means for developers to identify such divergences.

Insight: Main-effects sampling still necessary, but not sufficient Moreover, our results contribute to the *new feature-interaction challenge* (Apel et al., 2013). The idea is that there are different kinds of feature interactions, at different levels of abstraction, including functional and non-functional interactions that manifest in externally observable or internal behavior. The goal is to collect data from many different cases and triangulate results on interactions between options or features to learn about their nature and to predict one kind of interaction based on information about another kind (Kolesnikov et al., 2019a). Our results in $\text{RQ}_{2.1}$ and $\text{RQ}_{2.2}$ provide real-world data on likelihood and properties of performance feature interactions; our measurement and analysis framework offers a blueprint for conducting further experiments on other kinds of interactions (e.g., regarding memory utilization or energy consumption).

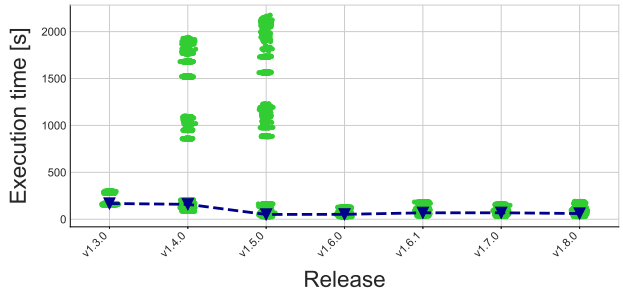

Fig. 11: The performance of all configurations (green dots) and the default configuration (blue dotted line) of VP9. The x-axis shows the releases and the y-axis the execution time in seconds.

Insight: Configuration awareness Another interesting issue of our empirical study is whether we are able to reveal new information in terms of performance changes in addition to what is already documented and thus well-known among developers and users. To investigate whether performance changes are explicitly documented by developers (i.e., the developers added the performance change intentionally), we manually analyzed the change logs (if available) of 6

out of 12 systems (i.e., FASTDOWNWARD, HSQLDB, LRZIP, VP8, VP9, Z3) in Section 4.2. Several performance changes have been documented by developers, but not all. We found that developers often report speed-ups in commit messages and change logs but only rarely slow-downs. The reason may be that developers become aware of these slow-downs only after deployment, as several cases indicate in which the slow-down was encountered and fixed one or two releases later. Such issues could be detected early by a configuration-aware continuous performance testing pipeline. Although some software systems, such as Z3 and VP9 use performance tests, these are not configuration-aware. This could explain why these subject systems report regressions, but only to a certain extent. Our results suggest that configuration-aware performance testing can indeed provide new information in an automated manner and simultaneously validates our findings. Interestingly, in some of the performance changes, we observed a slow-down, although the change logs reported a speed-up. In particular, version 1.4.0 of VP9 promises faster encoding in change log, although the change results in a slow-down of 265%, which can be considered as an unintentional slow-down. The reason behind this discrepancy is that the change log referred only to the default configurations; all other configurations, however, were affected by a massive slow-down, possibly untested and unaware by the developers. For illustration, we contrast in Figure 11 the performance of the default configuration and the mean performance of all configurations. Notably, in version 1.6.0 the performance regression has been fixed resulting in a speed-up of all configurations; the performance of the default configuration, however, remains largely unchanged. This performance optimization was achieved by avoiding and reordering some of the processor instructions for Intel chips and is mentioned in the change logs. This is an interesting aspect, since such cases demonstrate the importance of automated support and paves the way for further research in this area.

4.4 Threats to Validity

Construct validity To guarantee comparability across releases and to simplify benchmarking, we selected options that are available in all releases. While we may have missed interesting cases, this way, we increase internal validity by ruling out effects from option-specific benchmarks. Moreover, while performance changes could affect newly included configuration options that are enabled by default, this would affect either the whole configuration space or certain configuration options if the configuration option does depend on another configuration option. Either way, this would be visible in the performance-influence models. This affected also our deeper analysis and is the reason for why we have excluded consecutive releases where only the base code changes. In the end, only 4 pairs of consecutive releases of POSTGRESQL were excluded by this filter. In all other cases, the performance-influence model shows changes in certain configuration options or interactions or does not change at all. Another threat to validity arises from the selection of the keywords for filtering commit messages. Choosing another set of keywords may yield other results. However, all selected keywords were used in related publications (Jin et al., 2012; Chen et al., 2018) that focus on identifying performance regressions in commit messages or issue lists. One reason for the low number of reported configuration options is that developers may state configuration options under different names (e.g., q or $quality$ for the compression level in BROTLI). We have encountered few cases in which very specific parts of the code were addressed in a commit message, but a clear relation to a configuration option is hard to discover without domain knowledge and code inspection. Another reason could be data-flow dependencies between the configuration options. For instance, in HSQLDB, the configuration option blowfish was

not mentioned a single time in any commit message when a performance change occurred. When other configuration options affect the data that has to be encrypted by blowfish, then we relate the change to blowfish as the effect occur here, but the cause resides in code of another option.

Internal validity Measurement noise is not only caused by software but also by hardware (Mytkowicz et al., 2009). To limit measurement noise, we used identical hardware per subject system, running with a minimum DEBIAN installation. Furthermore, we preceded the measurements with a CPU warm-up phase. The measurements of the Java-based database (HSQLDB) are additionally preceded by a complete benchmark execution because of the JIT compilation as proposed by Georges et al. (Georges et al., 2007). Furthermore, we isolated the benchmark execution of client-server software (i.e., HSQLDB, MARIADB, MYSQL, OPENVPN, and POSTGRESQL) by running the server on a different node than the client(s) running the benchmark. To avoid wrong benchmark results, Costa et al. (Costa et al., 2021) observe and solve different bad practices in method-level performance tests. Since we measure the system as a whole and not individual methods by, for instance, issuing SQL queries to the database system, we are not affected by these bad practices. We varied the hardware across subject systems, since we do not need to compare measurements among systems. Furthermore, we used, if possible, the same version of the libraries over all releases, and we repeated our measurements three to five times until the relative standard deviation of the repetitions was lower than 10%. To control measurement noise, we used the standard deviation to pin down performance changes.
The choice of the learning algorithm may threaten internal validity. Other learning algorithms could have produced other results for $RQ_{2.1}$ and $RQ_{2.2}$. We used multiple linear regression with feature forward selection (Siegmund et al., 2015) because the additive structure of models enables us to track performance influences across releases by comparing the coefficients of model terms. Further, choosing always the best candidate in the feature forward selection (see Lines 11–14 in Algorithm 1) represents another limitation of our approach, since choosing a worse performing candidate in one iteration might lead to much better performing candidates in a later iteration. In other words, our learned models could not represent the optimum models. However, the prediction error of the models was 3.7% on average, which indicates that the models cover nearly all influences of options and interactions on performance accurately. To reduce spurious terms, which are only an artifact of the measurement and learning procedure, we checked the documentation (i.e., commit messages and change logs, if available) of our subject systems.
Additionally, we have used the variance inflation factor analysis to reduce variance in the performance-influence models as described in Section 2.4. This step removed a few terms by maintaining the error rate of the performance-influence models. Removing terms that are not perfectly multicollinear but exceed these thresholds removes important terms needed to predict specific parts of the configuration space and, thus, the error rate decreases. In a pre-study, we have applied the variance factor analysis by using the commonly threshold of 5 (Sheather, 2009) on the subject system LRZIP. From 230 terms, 160 were removed by the variance factor analysis but at the cost of increasing the error rate of the performance-influence model from 6 % to 60 %. In our setup, we removed only terms with perfect collinearity. In Table 6, we show the number of terms of the performance-influence models before and after the VIF analysis. Overall, we removed 14 out of 702 terms while the performance-influence models' error rate remained constant.
Finally, our metrics for identifying performance changes may threaten internal validity, since other metrics would identify other performance changes. For instance, the work of Costa et

Table 6: The number of terms of the performance-influence model per subject system before and after the variance inflation factor (VIF) analysis.

System	|Initial Terms|	|Terms after VIF|
BROTLI	166	166
FAST DOWNWARD	44	41
HSQLDB	21	21
LRZIP	220	220
MARIADB	35	33
MYSQL	25	23
OPENVPN	13	13
OPUS	66	66
POSTGRESQL	3	3
VP8	40	40
VP9	51	44
Z3	18	18
Total	702	688

al. (Costa et al., 2021) investigates the performance change of some bad practices at method level of one single configuration and uses the Wilcoxon non-parametric test and Clifft's Delta effect size to identify significant performance changes of their benchmark results. We refrained from using statistical tests to assess a significant performance change because the number of performance values per configuration (i.e., 3 or 5 performance values; 1 from each repetition) is far too low for a significance test and the suggested effect size metric, whereas the work of Costa et al. had at least 100 performance values due to a high number of repetitions. Increasing the number of repetitions on a similar level is infeasible despite the number of releases and configurations, we measured. Instead, we have used the standard deviation as an effect size to express the variance of measurement noise across multiple repetitions.

Due to the absence of a baseline, we need to resort to an automated approach, which we complemented, though, by studying commit messages and change logs manually (see above).

External validity To increase external validity, we chose configurable software systems from different domains, including throughput-intensive applications (compression tools, video encoders) and client-server applications (Web servers, databases). In total, our corpus contains software systems ranging from 181 to 6 480 configurations and 7 to 22 releases.

To keep experiment effort feasible, we limited the selection of configuration options to a tractable number. This limitation is due to our experiment setup, which aimed for high internal validity, and is not a principal limitation of our analysis framework. Considering more configuration options would require to sample the configuration space for learning performance influence models, instead of considering the whole space. While learning performance-influence models on small sample sets works well in practice (Kaltenecker et al., 2020), we aimed for high internal validity, ruling out possible inaccuracies.

The choice of the workload for performance measurement poses another threat to external validity. We have fixed the workload/benchmark across configurations and releases, this way gaining internal validity for external validity—see the discussion above. However, we used established community or developer workloads to catch typical scenarios, which already provided numerous interesting insights (see Section 3.3). For instance, the selection of the

developer workload might be a reason why we found no performance changes in OPUS. Varying the workload shall bring even more insights in further studies.

5 Related Work

In this section, we discuss related work with respect to (1) the role and evolution of software performance, (2) methods to analyze the performance changes, and (3) the evolution of software configurability.

Performance & Software Evolution Root causes of performance changes and their effect on maintainability have been studied before. Zaman et al. conducted an analysis of over 400 bugs from MOZILLA FIREFOX and GOOGLE CHROME (Zaman et al., 2012). They found that performance bugs often require more effort to fix and, therefore, are more costly than fixing functional bugs. A study on MOZILLA FIREFOX, APACHE, and MYSQL found a strong relation between configurability and performance: 113 out of 193 bugs were configuration related (Han and Yu, 2016).
Alcocer et al. studied the performance evolution of 19 software systems' releases. By analyzing the performance of multiple benchmarks, they found that one third of releases introduced performance bugs. The authors identified 9 patterns for performance changes (Alcocer and Bergel, 2015), which include performance improvements, due to removing redundant method calls or caching, as well as performance regressions arising from the composition of collection operations. Our work links both research directions—software configuration and software evolution—and explores performance of software systems across their configuration spaces and along their development histories.

Performance Change Detection The detection of performance changes has been approached from different angles, such as using different statistical methods, and taking one or more performance characteristics of the software system into account. For example, statistical process control charts were used to capture changes of an observed metric, such as the performance of the system, and provide thresholds, which, when exceeded by accumulated change, indicate a performance degradation (Nguyen et al., 2014; Malik et al., 2013; Lee et al., 2012). Other statistical approaches rely on testing and determining whether two observations are statistically different. For example, Heger et al. compare the performance distributions for different versions with ANOVA (Heger et al., 2013). Reichelt et al. apply different statistical tests to identify performance anomalies from performance histories (Reichelt and Kühne, 2018).
Aside from considering only a single performance measure, previous work considers multiple measures and their relations. Foo et al. mine repositories regarding performance regression tests and automatically detect performance changes by tracking the correlation of performance measures over time (Foo et al., 2010). Malik et al. analyze performance regression by automatically selecting a subset of performance measures that describe system performance (Malik et al., 2013). Using principal component analysis, they correlate the measures to obtain a performance fingerprint, which then can be compared across releases.
All this work illustrates that performance changes can manifest in many ways. However, it does not consider configurability and to what extent individual configuration options or interactions cause performance changes, which is the focus of this paper.

Evolution: Configurability & Performance Mühlbauer et al. devised a prediction technique for performance changes in software repositories, across versions and configurations (Mühlbauer et al., 2020). This work is the closest but complementary to ours: While we study the prevalence and properties of performance feature interactions in the wild, they propose a technique to discover them with little effort. In principle, we could have used their technique to collect the data for our study. But, as their approach only approximates performance changes with iterative sampling, we analyze the configuration space as a whole for accuracy.
Several studies have observed and categorized recurring patterns in the evolution of variability models, such as the introduction or removal of new configuration options (often called *features*) or splitting generic options into more precise ones. There are three relevant patterns: a new feature is added, a mandatory feature becomes optional, or a mandatory/optional feature is split into alternative features (Peng et al., 2011; Seidl et al., 2012; Passos et al., 2016, 2021). Our study considers only configuration options that exist in all releases of the software system, which is the majority, though. However, for the interpretation of our results (cf. $RQ_{2.2}$), these patterns provide some context that can help map changes in the performance influence across releases. Recent work by Jamshidi et al. explores the applicability of transfer learning to adapt performance-influence models to different environments (Jamshidi et al., 2017). Their key insight, after investigating 4 configurable software systems, is that only a subset of configuration options and interactions among them have a strong influence on performance and that the performance influence is generally preserved across environments and software releases.

Workload Dependence Clearly, the performance of a software system may change depending on the workload. There is a substantial corpus of work studying this phenomenon and providing models and solutions that incorporate workload-dependent performance (Feitelson, 2002; Wolf et al., 2014; Mühlbauer et al., 2023). The work of Costa et al. and Leitner et al. focuses on studying and improving performance tests in Java-based open source projects (Leitner and Bezemer, 2017; Costa et al., 2021). Our work is complementary in that we study system configurability, which is a further dimension that influences a system's performance. To increase internal validity, we fixed the workload per system in our experiments. Ultimately, our approach and previous work on workload-dependent performance behavior shall be combined.

6 Conclusion

Although performance evolution has been extensively studied in the literature, prior work concentrated on single or few default configurations. Since most software systems are configurable, performance changes can easily be missed this way. Specifically, we are interested in the role of *configurability* for performance evolution, for example, whether specific configurations exhibit diverging performance behavior and what configuration options (or interactions among options) are responsible for this.
In an empirical study, we analyzed performance changes of 12 real-world configurable software systems across 190 releases that span a total of 15 years of history. We found that almost every release of every subject system exhibits performance changes in some of their configurations. Notably, the majority of performance changes affects only a small subset of the configuration space, and most performance changes affect multiple options (up to 6), either by accumulation of influences or interactions among options.

A deeper analysis of these configurable software systems shows that performance changes are reported in the change log or the commit messages in most cases. Similarly often, changes regarding affected configuration options have been mentioned.
Our results confirm prior beliefs that configuration-dependent performance changes are the rule, not the exception. This has direct implications for configuration sampling, performance modeling, and transfer learning in the area of configurable software systems. For example, our results confirm assumptions that simple random configuration sampling is not sufficient to catch all relevant performance changes. Likewise, structured sampling strategies likely overestimate the prevalence of performance-relevant interactions among options. Our results clearly indicate that combined sampling strategies such as *distance-based sampling* hit a proper sweet spot.
A further notable insight is that, despite the prevalence of performance changes, the performance ranking of configurations and influences of individual options are in many cases not affected. That is, developers and users can assume a certain stability of configuration-dependent performance behavior. Still, we found cases where the performance ranking fluctuates considerably across releases. This phenomenon seems to be application- or domain-specific and is worth further exploring, as it has implications for transfer learning of performance behavior across releases since more stable applications or domains could focus on the most relevant configuration options; in other applications and domains such approaches are doomed to fail. Additionally, our deeper analysis demonstrates that using a configuration-aware performance testing pipeline could help in identifying configuration-specific performance changes early. Our measurement and analysis framework offers a solid basis for exploring these and related issues.

Acknowledgements We thank our reviewers for their constructive comments. Apel's work has been funded by the German Research Foundation (DFG) under the contracts AP 206/11-1, AP 206/11-2, and Grant 389792660 as part of TRR 248 – CPEC. Siegmund's work has been funded by the German Research Foundation (SI 2171/2-2), by the Federal Ministry of Education and Research of Germany, and by the Sächsische Staatsministerium für Wissenschaft Kultur und Tourismus in the program Center of Excellence for AI-research Center for Scalable Data Analytics and Artificial Intelligence Dresden/Leipzig, project identification number: ScaDS.AI, and by the BMBF project Agile-AI.

References

Alcocer J, Bergel A (2015) Tracking Down Performance Variation Against Source Code Evolution. In: Proceedings of the Symposium on Dynamic Languages (DLS), ACM, pp 129–139

Andrews D (1974) A Robust Method for Multiple Linear Regression. Technometrics 16(4):523–531

Apel S, Kolesnikov S, Siegmund N, Kästner C, Garvin B (2013) Exploring Feature Interactions in the Wild: The New Feature-Interaction Challenge. In: Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD), ACM, pp 1–8

Burnim J, Juvekar S, Sen K (2009) WISE: Automated Test Generation for Worst-Case Complexity. In: Proceedings of the International Conference on Software Engineering (ICSE), IEEE, pp 463–473

Chen J, Shang W (2017) An Exploratory Study of Performance Regression Introducing Code Changes. In: Proceedings of the International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 341–352

Chen Z, Chen B, Xiao L, Wang X, Chen L, Liu Y, Xu B (2018) Speedoo: Prioritizing Performance Optimization Opportunities. In: Proceedings of the International Conference on Software Engineering (ICSE), ACM, pp 811–821

Costa D, Bezemer C, Leitner P, Andrzejak A (2021) What's Wrong with My Benchmark Results? Studying Bad Practices in JMH Benchmarks. IEEE Transactions on Software Engineering 47(7):1452–1467

Dorn J, Apel S, Siegmund N (2023) Mastering Uncertainty in Performance Estimations of Configurable Software Systems. Empirical Software Engineering 28(2):33

Feitelson D (2002) Workload Modeling for Performance Evaluation. In: Performance Evaluation of Complex Systems: Techniques and Tools, Springer, pp 114–141

Foo K, Jiang Z, Adams B, Hassan A, Zou Y, Flora P (2010) Mining Performance Regression Testing Repositories for Automated Performance Analysis. In: Proceedings of the International Conference on Quality Software (QRS), IEEE, pp 32–41

Gahvari H, Baker A, Schulz M, Yang U, Jordan K, Gropp W (2011) Modeling the Performance of an Algebraic Multigrid Cycle on HPC Platforms. In: Proceedings of the International Conference on Supercomputing (ICSP), ACM, pp 172–181

Georges A, Buytaert D, Eeckhout L (2007) Statistically Rigorous Java Performance Evaluation. In: Proceedings of the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA), ACM, pp 57–76

Grebhahn A, Rodrigo C, Siegmund N, Gaspar FJ, Apel S (2017) Performance-Influence Models of Multigrid Methods: A Case Study on Triangular Grids. Concurrency and Computation: Practice and Experience 29(17)

Han S, Dang Y, Ge S, Zhang D, Xie T (2012) Performance Debugging in the Large via Mining Millions of Stack Traces. In: Proceedings of the International Conference on Software Engineering (ICSE), IEEE, pp 145–155

Han X, Yu T (2016) An Empirical Study on Performance Bugs for Highly Configurable Software Systems. In: Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM), ACM, pp 1–10

Heger C, Happe J, Farahbod R (2013) Automated Root Cause Isolation of Performance Regressions During Software Development. In: Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE), ACM, pp 27–38

James G, Witten D, Hastie T, Tibshirani R (2013) An Introduction to Statistical Learning, vol 112. Springer

Jamshidi P, Siegmund N, Velez M, Kästner C, Patel A, Agarwal Y (2017) Transfer Learning for Performance Modeling of Configurable Systems: An Exploratory Analysis. In: Proceedings of the International Conference on Automated Software Engineering (ASE), IEEE, pp 497–508

Jamshidi P, Velez M, Kästner C, Siegmund N (2018) Learning to Sample: Exploiting Similarities Across Environments to Learn Performance Models for Configurable Systems. In: Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), ACM, pp 71–82

Jiang Z, Hassan A (2015) A Survey on Load Testing of Large-Scale Software Systems. IEEE Transactions on Software Engineering 41(11):1091–1118

Jin G, Song L, Shi X, Scherpelz J, Lu S (2012) Understanding and Detecting Real-World Performance Bugs. In: Conference on Programming Language Design and Implementation (PLDI), ACM, pp 77–88

Kaltenecker C, Grebhahn A, Siegmund N, Guo J, Apel S (2019) Distance-Based Sampling of Software Configuration Spaces. In: Proceedings of the International Conference on

Software Engineering (ICSE), IEEE, pp 1084–1094
Kaltenecker C, Grebhahn A, Siegmund N, Apel S (2020) The Interplay of Sampling and Machine Learning for Software Performance Prediction. IEEE Software 37(4):58–66
Kendall M (1938) A New Measure of Rank Correlation. Biometrika 30(1/2):81–93
Knüppel A, Thüm T, Pardylla C, Schaefer I (2018) Understanding Parameters of Deductive Verification: An Empirical Investigation of KeY. In: Proceedings of the International Conference on Interactive Theorem Proving (ITP), Springer, pp 342–361
Kolesnikov S, Siegmund N, Kästner C, Apel S (2019a) On the Relation of Control-flow and Performance Feature Interactions: A Case Study. Empirical Software Engineering 24(4):2410–2437
Kolesnikov S, Siegmund N, Kästner C, Grebhahn A, Apel S (2019b) Tradeoffs in Modeling Performance of Highly-Configurable Software Systems. Software and System Modeling 18(3):2265–2283
Kuhn M, Johnson K (2013) Applied Predictive Modeling, vol 26. Springer
Lee D, Cha S, Lee A (2012) A Performance Anomaly Detection and Analysis Framework for DBMS Development. IEEE Transactions on Knowledge and Data Engineering 24(8):1345–1360
Leitner P, Bezemer C (2017) An Exploratory Study of the State of Practice of Performance Testing in Java-Based Open Source Projects. In: Proceedings of the International Conference on Performance Engineering (ICPE), ACM, pp 373–384
Malik H, Hemmati H, Hassan AE (2013) Automatic Detection of Performance Deviations in the Load Testing of Large Scale Systems. In: Proceedings of the International Conference on Software Engineering (ICSE), IEEE, pp 1012–1021
Mühlbauer S, Apel S, Siegmund N (2019) Accurate Modeling of Performance Histories for Evolving Software Systems. In: Proceedings of the International Conference on Automated Software Engineering (ASE), ACM, pp 640–652
Mühlbauer S, Sattler F, Kaltenecker C, Dorn J, Apel S, Siegmund N (2023) Analyzing the Impact of Workloads on Modeling the Performance of Configurable Software Systems. In: Proceedings of the International Conference on Software Engineering (ICSE), IEEE
Mytkowicz T, Diwan A, Hauswirth M, Sweeney P (2009) Producing Wrong Data Without Doing Anything Obviously Wrong! In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), ACM, pp 265–276
Mühlbauer S, Apel S, Siegmund N (2020) Identifying Software Performance Changes Across Variants and Versions. In: Proceedings of the International Conference on Automated Software Engineering (ASE), ACM
Nair V, Menzies T, Siegmund N, Apel S (2017) Using Bad Learners to Find Good Configurations. In: Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), ACM, pp 257–267
Nguyen T, Nagappan M, Hassan A, Nasser M, Flora P (2014) An Industrial Case Study of Automatically Identifying Performance Regression-Causes. In: Proceedings of the Working Conference on Mining Software Repositories (MSR), ACM, pp 232–241
Oh J, Batory D, Myers M, Siegmund N (2017) Finding Near-Optimal Configurations in Product Lines by Random Sampling. In: Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), ACM, pp 61–71
Passos L, Teixeira L, Dintzner N, Apel S, Wąsowski A, Czarnecki K, Borba P, Guo J (2016) Coevolution of Variability Models and Related Software Artifacts: A Fresh Look at

Evolution Patterns in the Linux Kernel. Empirical Software Engineering 21(4):1744–1793
Passos L, Queiroz R, Mukelabai M, Berger T, Apel S, Czarnecki K, Padilla J (2021) A Study of Feature Scattering in the Linux Kernel. IEEE Transactions on Software Engineering (TSE) 47(1):146–164
Peng X, Yu Y, Zhao W (2011) Analyzing Evolution of Variability in a Software Product Line: From Contexts and Requirements to Features. Information & Software Technology 53(7):707–721
Pereira J, Acher M, Martin H, Jézéquel JM (2020) Sampling Effect on Performance Prediction of Configurable Systems: A Case Study. In: Proceedings of the International Conference on Performance Engineering (ICPE), ACM
Pett T, Thüm T, Runge T, Krieter S, Lochau M, Schaefer I (2019) Product Sampling for Product Lines: The Scalability Challenge. In: Proceedings of the International Systems and Software Product Line Conference (SPLC), ACM, pp 14:1–14:6
Pukall M, Kästner C, Cazzola W, Götz S, Grebhahn A, Schröter R, Saake G (2013) JavAdaptor - Flexible Runtime Updates of Java Applications. Software: Practice and Experience 43(2):153–185
Reichelt D, Kühne S (2018) How to Detect Performance Changes in Software History: Performance Analysis of Software System Versions. In: Proceedings of the International Conference on Performance Engineering (ICPE), ACM, pp 183–188
Saltelli A (2008) Global Sensitivity Analysis: The Primer. John Wiley
Seidel I, de Moraes B, Wuerges E, Güntzel J (2013) Quality Assessment of Subsampling Patterns for PEL Decimation Targeting High Definition Video. In: Proceedings of the International Conference on Multimedia and Expo (ICME), IEEE, pp 1–6
Seidl C, Heidenreich F, Aßmann U (2012) Co-Evolution of Models and Feature Mapping in Software Product Lines. In: Proceedings of the International Software Product Line Conference on (SPLC), ACM, p 76
Sheather S (2009) A Modern Approach to Regression with R. Springer Science & Business Media
Siegmund N, Rosenmüller M, Kästner C, Giarrusso P, Apel S, Kolesnikov S (2013) Scalable Prediction of Non-functional Properties in Software Product Lines: Footprint and Memory Consumption. Information & Software Technology 55(3):491–507
Siegmund N, Grebhahn A, Apel S, Kästner C (2015) Performance-Influence Models for Highly Configurable Systems. In: Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), ACM, pp 284–294
Thüm T, Teixeira L, Schmid K, Walkingshaw E, Mukelabai M, Varshosaz M, Botterweck G, Schaefer I, Kehrer T (2019) Towards Efficient Analysis of Variation in Time and Space. In: Proceedings of the International Systems and Software Product Line Conference (SPLC), ACM, pp 69:1–69:8
von Rhein A, Liebig J, Janker A, Kästner C, Apel S (2018) Variability-Aware Static Analysis at Scale: An Empirical Study. ACM Transactions on Software Engineering and Methodology 27(4):18:1–18:33
Wolf F, Bischof C, Hoefler T, Mohr B, Wittum G, Calotoiu A, Iwainsky C, Strube A, Vogel A (2014) Catwalk: A Quick Development Path for Performance Models. In: Proceedings of the European Conference on Parallel Processing (Euro-Par), Springer, pp 589–600
Xu T, Jin L, Fan X, Zhou Y, Pasupathy S, Talwadker R (2015) Hey, You Have Given Me Too Many Knobs!: Understanding and Dealing with Over-Designed Configuration in System Software. In: Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of

Software Engineering (ESEC/FSE), ACM, pp 307–319
Zaman S, Adams B, Hassan A (2012) A Qualitative Study on Performance Bugs. In: Proceedings of the Working Conference on Mining Software Repositories (MSR), IEEE, pp 199–208
van Zyl P, Kourie D, Boake A (2006) Comparing the Performance of Object Databases and ORM Tools. In: Proceedings of the Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries (SAICSIT), South African Institute for Computer Scientists and Information Technologists, p 111