The Journal of Systems and Software 210 (2024) 111904

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

The slow and the furious? Performance antipattern detection in Cyber–Physical Systems☆

Imara van Dinten [a,*], Pouria Derakhshanfar [b], Annibale Panichella [a], Andy Zaidman [a]

[a] Delft University of Technology, Van Mourik Broekmanweg 6, 2628 XE, Delft, The Netherlands
[b] JetBrains Research, Huidekoperstraat 26-28, 1017 ZM, Amsterdam, The Netherlands

ARTICLE INFO

Keywords:
Software Performance Antipatterns
Cyber–Physical Systems
Antipattern detection
Software maintenance
Empirical Software Engineering
Static analysis

ABSTRACT

Cyber–Physical Systems (CPSs) have gained traction in recent years. A major non-functional quality of CPS is performance since it affects both usability and security. This critical quality attribute depends on the specialized hardware, simulation engines, and environmental factors that characterize the system under analysis. While a large body of research exists on performance issues in general, studies focusing on performance-related issues for CPSs are scarce. The goal of this paper is to build a taxonomy of performance issues in CPSs. To this aim, we present two empirical studies aimed at categorizing common performance issues (Study I) and helping developers detect them (Study II). In the first study, we examined commit messages and code changes in the history of 14 GitHub-hosted open-source CPS projects to identify commits that report and fix self-admitted performance issues. We manually analyzed 2699 commits, labeled them, and grouped the reported performance issues into antipatterns. We detected instances of three previously reported Software Performance Antipatterns (SPAs) for CPSs. Importantly, we also identified new SPAs for CPSs not described earlier in the literature. Furthermore, most performance issues identified in this study fall into two new antipattern categories: Hard Coded Fine Tuning (399 of 646) and Magical Waiting Number (150 of 646). In the second study, we introduce static analysis techniques for automatically detecting these two new antipatterns; we implemented them in a tool called `AP-Spotter`. We analyzed 9 open-source CPS projects not utilized to build the SPAs taxonomy to benchmark `AP-Spotter`. Our results show that `AP-Spotter` achieves 62.04% precision in detecting the antipatterns.

1. Introduction

The term CPSs was first coined at the 2006 National Workshop on Cyber–Physical Systems by Gill (2006). As described by Lee and Seshia (2017), a CPS is an integration of computation with physical processes whose behavior is defined by both cyber and physical parts of the system. Examples of CPSs are medical devices (Chen, 2017), automation of industrial manufacturing systems (Lee et al., 2015), air traffic control and aircraft avionic systems (Lee, 2015), smart cars (Birchler et al., 2022; Abdessalem et al., 2020; Gambi et al., 2019), and unmanned vehicles (Shi et al., 2011; Khatiri et al., 2023).

In recent years, CPSs became of interest across many industries (De-Franco and Serpanos, 2021; Okolie et al., 2018). The increased adoption of CPSs increases the urgency to tackle CPSs-specific challenges (Mittal and Tolk, 2019). A key challenge when working with CPSs is that it is difficult to consider the parts in isolation due to the tight interactions (Greer et al., 2019). As with other real-time systems, CPSs have a limited time to react to their environment (Jain et al., 2021). For example, a self-driving car needs to react fast if suddenly a deer wanders on the road. The performance of the system is a big factor in how well it responds.

One of the standard methods for achieving high software performance is to use a catalog of Software Performance Antipatterns SPAs (Smith and Williams, 2001, 2000). This catalog documents the common performance problems in the software architecture and design of systems. The description of these antipatterns helps detect bad design/coding choices that influence performance. A previous study empirically showed that SPAs are beneficial while providing reusable solutions applicable in various domains (Trubiani et al., 2014). Moreover, identifying SPAs helps design and inform refactoring actions, which ensure that the performance antipatterns can be removed from the project's architecture or designs (Aleti et al., 2018; Calinescu et al., 2017). Recent studies identified various performance antipatterns that are widespread in *classical* software (i.e., not related to CPSs) (Smith

☆ Editor: Leandro L. Minku.
* Corresponding author.
E-mail addresses: I.vanDinten@tudelft.nl (I. van Dinten), A.Panichella@tudelft.nl (A. Panichella).

https://doi.org/10.1016/j.jss.2023.111904
Received 10 May 2023; Received in revised form 4 October 2023; Accepted 12 November 2023
Available online 24 November 2023

and Williams, 2000, 2001, 2002b, 2003). In the context of CPSs, Smith (2020) started a CPS-specific SPA catalog in 2020. This catalog describes nine performance antipatterns, three of which are CPS specific, and six are also applicable to generic software. Although the antipatterns introduced in Smith's study facilitate the recognition and refactoring of CPS performance-related issues and show the relevance of performance antipatterns in an industrial context, their work has two main limitations. First, the proposed catalog was formulated based on the author's experience, rather than empirically collecting and analyzing data from existing and diverse CPSs. Second, their study did not assess how common and widespread these antipatterns are.

The aim of this paper is twofold: (1) building an extensive taxonomy (or catalog) of performance antipatterns in CPSs based on empirical data and evidence collected from heterogeneous open-source systems; (2) helping developers detect the most common and widespread antipatterns. Therefore, this paper presents two studies that cover both the classification (**Study I**) and the detection (**Study II**) of performance-related antipatterns in CPS.

Study I aims to identify, classify, and categorize performance issues into a *taxonomy*. Therefore, we investigate the following research questions

RQ1: *Which CPS-specific performance antipatterns can we identify in open-source CPSs?*

RQ2: *How prevalent are CPS-specific performance antipatterns in open-source CPS projects?*

To answer these two research questions, we analyzed the code history of 14 open-source CPS projects publicly available on GitHub and used in prior studies related to CPSs (Zampetti et al., 2022). We examined commits that reported and fixed self-admitted performance issues by analyzing (1) commit messages, (2) code and project documentation, and (3) code changes. Self-admitted issues are identified based on performance-specific keywords (e.g., *run-time*, *memory*) using a tool that we implemented and coined `PyRock`. This resulted in 1059 candidate commits to validate manually. Through manual analysis, we identified 530 (81.11%) commits discussing one or more performance-related issues. We further expanded the keywords by using textual analysis methods (De Lucia et al., 2014) and topic modeling (Panichella et al., 2013; Panichella, 2019). This resulted in 1640 additional commits, of which 163 commits (9.37%) contained one or more self-admitted performance-related issues. In total, we manually analyzed 2699 commits, labeled them, and grouped the reported performance issues in common categories (or antipatterns).

In this final set of 2699 commits, we found eight instances of Smith's (2020) CPS-specific antipatterns. Interestingly, we identified six potential new CPS-specific performance antipatterns with 638 instances in total. As this is exploratory research, we decided that for them to be confirmed as antipattern, they needed to occur in more than two projects. Following this criterion, we confirmed four antipatterns:

- *Magical Waiting Number.* (*9 projects, 150 instances*) Lack of proper waiting time, the potential of being often manually changed due to adding support for slower/faster hardware platforms.
- *Hard Coded Fine Tuning.* (*6 projects, 399 instances*) Variables that are closely related to hardware support, which keeps being changed throughout the project's history.
- *Fixed Communication Rate.* (*5 projects, 66 instances*) Frequent changes are made to the communication rate of the hardware modules, as the minimum latency is hard-coded (the same for all platforms in all situations) instead of dynamically.
- *Rounded Numbers.* (*5 projects, 10 instances*) Mathematical errors made due to type usage for situations requiring high accuracy.

Study II aims at helping developers identify and detect the two most frequently occurring new antipatterns (*Magical Waiting Number* and *Hard Coded Fine Tuning*). To this aim, we present and assess a novel tool, named `AP-Spotter`, that detects these antipatterns based on static analysis methods. The benefit of using a static analysis technique is that it is fast, compared to dynamic analysis, so it can give timely feedback to the developer. We implemented this technique with the `AP-Spotter` tool. Therefore, we formulated the following research question:

RQ3: *How precise can our approach detect performance antipatterns?*

To answer this research question, we ran `AP-Spotter` against a benchmark of 9 additional open-source CPS projects not utilized to build the new SPAs taxonomy (i.e., not considered in Study I). We manually validated the instances detected by `AP-Spotter` and reached a precision of 63.39% for Magical Waiting Number, and 60.98% for Hard Coded Fine Tuning.

Paper Structure The remainder of the paper is structured as follows: Section 2 provides background and related work. Section 3 is the repository mining study on Performance Antipatterns. Section 4 discusses the new potential antipatterns from our study. In Section 5, we explore automatically detecting CPS antipatterns and evaluate our approach empirically in our second study. Further, Section 6 discusses the threats to the validity of our studies. And finally, Section 7 presents our conclusion.

2. Background and related work

In this section, we give a brief overview of research relevant to our studies.

2.1. Software Performance Antipatterns

The concept of design patterns for software was introduced in 1994 by Gamma et al. (1994), as a schematic to follow for designing software components or subsystems. Software antipatterns stand opposite to design patterns, in that they describe patterns to be avoided because of potential issues in the software's security, performance, stability, or maintainability (Brown et al., 1998; Moesus et al., 2019).

The specific subcategory of antipatterns in which we conduct our study is called Software Performance Antipatterns (SPAs). SPAs focus on common patterns in software architecture and design, which lead to performance issues in the system (Smith and Williams, 2000).

These antipatterns have been introduced by Smith and Williams (2000) and were later extended for pipe-and-filter architectures (Smith and Williams, 2002b) and concurrent programs (Smith and Williams, 2003). Various studies have contributed towards defining SPAs for other application domains (e.g., databases Dugan et al., 2002), and provided solutions to tackle them (Smith and Williams, 2002a).

2.2. Known SPAs for CPSs

To the best of our knowledge, there is only one prior study that has identified SPAs specific to CPSs, this study was conducted by Smith (2020). In particular, Smith's (2020) recent study carried out a preliminary investigation into Performance Antipatterns for CPS. She identified three new SPAs specific to CPS and also recognized six other SPAs specific to CPS. Based on industrial experience in the field, Smith (2020) introduced three new SPAs specific to CPS and also recognized six other SPAs specific to CPS. These antipatterns are described in Table 1.

Despite the undisputable contribution by Smith (2020), the study did not provide any empirical data to support the findings. Therefore, our goal is to shed light on the prevalence of SPAs in CPSs by performing an empirical study on a large set of CPSs, eventually extending the existing SPAs taxonomy.

Table 1
Existing catalog of SPAs for CPSs by Smith (2020).

Name	Type	Description
Are We There Yet?	CPS-specific	Over-checking whether an event occurred. This problem usually stems from a polling procedure in CPS with small checking intervals, compared to the frequency of event occurrences. This Performance Antipatterns leads to overusing system resources.
Is Everything OK?	CPS-specific	Constantly checking the status of the system (e.g., storage space, battery usage). This performance issue happens when the status checker threads and processes are triggered too often.
Where Was I?	CPS-specific	Processes that lost information about the system's state, such as a system restart. It can also happen if too much time (i.e., more than 1 min) is given to processes that keep the user waiting. This type of antipattern leads to execution overheads to perform required calculations to drive the CPS back to the desired status.
Unnecessary Processing	Generic	Heavy and unnecessary processes are executed in critical scenarios (Smith and Williams, 2003). To tackle this antipattern, the execution of processes whose outputs are not required in critical scenarios should be postponed.
How Many Times Do I Have to Tell You?	Generic	Invoking a method many times in scenarios in which the CPS could call the method only once and store and reuse the returned outputs for the following processes. To address this antipattern, redundant calls should be removed.
More is Less	Generic	A CPS has access to too many resources that negatively impact the system's overall performance (Smith, 2020). Adding too many resources (such as threads and processes) may lead to extra overheads for tasks like scheduling and context switching
The Ramp	Generic	The performance and efficiency of the CPS are exponentially reduced as the processing time linearly increases (Smith and Williams, 2002b). This type of performance issue can occur in CPS for various reasons, such as changes in the environment or processing a large amount of historical information (Smith, 2020).
Museum Checkroom	Generic	A CPS uses a simple First Come First Serve (FCFS) queue to manage resource allocation to processes (Bondi, 2014). This can lead to performance issues in cases where this resource management system needs to handle too many processes. To resolve it, CPS developers should implement priority queuing.
Falling Dominoes	Generic	A failure of a module leads to more failures in other modules (Smith, 2020). Since CPSs includes many small interacting hardware pieces with various software modules, this common antipattern can also occur in a CPS. Developers must ensure that modules are as isolated as possible to prevent this antipattern from occurring.

2.3. From automatic detection of performance issues to CPS-PAs

Performance issues in a system can e.g., negatively impact its security (Wu et al., 2016; Ashibani and Mahmoud, 2017) and usability (Shackel, 2009). Therefore, detecting performance problems has been of interest among researchers (Velez et al., 2021), from researching Machine Learning methods for performance prediction (Kaltenecker et al., 2020) to the creation of several tools to aid in detecting performance issues earlier on. Some of the available tools are: (i) `PerformanceHat` (Cito et al., 2018) *(This tool aids to bring awareness to development choices and its impact on performance.)* and (ii) `Toddler` (Nistor et al., 2013) *(An automated oracle to detect redundant and inefficient use of loops, causing unnecessary performance degradation.)*. Applying performance models for analysis is a popular research area, examples of different approaches in this field are: (i) detecting performance regression using the system's history (Mühlbauer et al., 2019), and (ii) the interactions between configurations options towards performance (Velez et al., 2021). Research into the different modeling options also resulted in the creation of PUMA (Woodside et al., 2005), a tool architecture aiming to bridge the different available design- and performance tools. In another work, Pinciroli and Trubiani investigate how architectural patterns in cyber–physical systems can influence different performance metrics, e.g., the system response time (Pinciroli and Trubiani, 2021). Their approach relies on stochastic performance models and enables software architects to quantitatively evaluate architectural patterns in terms of performance.

As manually detecting antipatterns is tedious, time-consuming, and requires expert developers (Veliolu and Selçuk, 2017; Maiga et al., 2012), automatic detection of antipatterns has also become a popular research topic. Multiple approaches have been proposed in the literature to identify generic performance antipatterns, such as approaches based on first-order logic representation (Cortellessa et al., 2014), decision trees (Wert et al., 2013), multivariate analysis (Avritzer et al., 2021), and load testing (Trubiani et al., 2018). Examples of tools that have been created to automatically detect antipatterns are: (i) `DECOR` (Moha et al., 2010) *(Detects antipatterns such as The Spaghetti Code and The Blob antipatterns)* and (ii) `PMD` (PMD, 2023; Schügerl et al., 2009) *(A static code analyzer that can be used to detect Code Duplication and the Leak Collection antipattern)*.

To the best of our knowledge, a recent article by Pinciroli et al. (2021) is the only study that focused on detecting CPS-PAs. The authors modeled the performance antipatterns for CPSs that have been introduced by Smith (2020), by utilizing queuing networks. Then, their tool monitors the components' states dynamically to identify the performance antipatterns in the CPS's operation.

While their focus is on dynamically identifying performance antipatterns, we approach this detection with static analysis and rely on the project's historical development. Dynamic analysis can have the benefit of getting a rich insight into the antipatterns impact on the system's performance. For the static analysis method, the code does not need to be run, therefore it can provide faster feedback to the CPS developers (Ernst, 2003).

3. Study I — Commit-message based search for performance antipatterns

In our first study, we aim (1) to empirically investigate how widespread the antipatterns from the existing catalog created by Smith (2020) are, and (2) to potentially identify any previously undocumented antipatterns across open-source CPS projects.

Our investigation is steered by the following two research questions:

RQ1: *Which CPS-specific performance antipatterns can we identify in open-source CPSs?*

RQ2: *How prevalent are CPS-specific performance antipatterns in open-source CPS projects?*

This section is structured as follows: first, we discuss the projects selected for this study (Section 3.1), followed by a description of how we searched for performance issues (Section 3.2), the analysis (Section 3.3), and results (Section 3.4). We discuss the results from this study in Section 4.

3.1. Subjects

To derive a taxonomy of antipatterns in CPS projects, we selected a set of 14 open-source CPS projects hosted on GitHub. We selected these projects based on the following criteria:

Table 2
Projects selected for Study I.

Project name	Language	Nr. of commits	Stars	Forks	Domain
PX4-Autopilot	C++	35,537	4.8K	11.3K	Automotive
Andruino-esp32	C	1747	8.5k	5.4k	Arduino
Grbl	C	699	3.2k	1.4k	Arduino
DroneKit Android	Java	5810	211	217	Drones
Node AR Drone	JavaScript	281	1.7K	446	Drones
Android App Manager	Java	231	10	12	Robotics
Cylon	JavaScript	1323	3.8K	367	Robotics
Johnny Five	JavaScript	3355	12.4K	1.8K	Robotics
Robonomics-JS	JavaScript	68	13	8	Robotics
Robonomics-Contracts	JavaScript	502	78	31	Robotics
Vacuum Robot Mark II	Java/C++	54	28	3	Wheeled Robot
TurtleBot	C++	1142	236	280	Wheeled Robot
TurtleBot 3	Python	526	770	637	Wheeled Robot
Valetudo	JavaScript	1043	2.5K	258	Wheeled Robot

- *Relevance*: the project must be related to a CPS domain, such as robotics, drones, or automotive.
- *Activity*: the project must have a minimum of 50 commits as we have to identify self-admitted performance issues. This lower limit was chosen to be inclusive of less active projects while excluding projects that did not use GitHub to track and resolve issues.
- *Popularity*: a selection of popular and less popular projects must be made; We selected the two most popular projects in the CPS domain, i.e., the projects with the most stars and forks; nine relatively popular projects; and an additional four projects with less than 100 stars, as we are interested in antipatterns that arise independently of the maturity of a project. From the most and least popular projects, one project with a high star rating and one with a high fork rating was selected. For the less popular project, we selected two additional projects as this category of projects were shown to have a low number of commits. The other eight projects are selected to range between the least and most popular projects.
- *Programming language*: the project must be written in `C`, `C++`, `Java`, `JavaScript`, or `Python`. While `C`, `C++` are the most common programming languages for CPS (Zampetti et al., 2022), we also consider other programming languages to increase the generalizability of our results. Therefore, the resulting selection of projects must be a diverse selection of programming languages used.

This resulted in 14 projects, which are described in Table 2. Notice that the projects written in `C`, `C++`, and `Python` have also been used in a prior study (Zampetti et al., 2022) aimed at characterizing CPSs according to the type of functional bugs/issues they contain. Our selection comprises projects with various maturity levels: PX4-Autopilot and Vacuum Robot Mark II have the highest and lowest contributions, with 35,537 and 54 commits to the main branch, respectively. Furthermore, these projects reflect different applications of CPS, such as software for controlling drones, vacuum cleaners, or small robot kits. Table 2 also indicates the number of stars and forks for each of the CPS projects. The most popular projects in this dataset are Johnny-Five with 12.4K stars and PX4-Autopilot with 4.8K stars. This diversity in projects aims to ensure that we find antipatterns not specific to one CPS domain, a programming language, or the maturity projects.

3.2. Analyzing self-admitted performance issues

To build our taxonomy of antipatterns, we manually analyzed and classified self-admitted performance issues, i.e., issues related to performance aspects (e.g., memory usage) that are mentioned by the developers in the commit messages, or source code changes and comments. Our main methodology is, therefore, based on manual analysis of structured (source code) and unstructured (commit messages) data, which we manually classified into categories of antipatterns.

There are two ways to build a taxonomy (Rowley and Hartley, 2017): (1) *top down* (also called *enumerative*), where the categories are predefined, or (2) *bottom up* (or *faceted*), where the categories are created by analyzing the data. Since we have a pre-existing catalog by Smith and Williams (2002a), we used a hybrid approach: for every commit we analyzed, we checked if it matched any of the categories in Smith's taxonomy; in case of no match, we marked them and later clustered them into new categories.

As reported in Table 2, there are 52,318 commits in total to analyze across all projects. Since manually analyzing all commits in our dataset would be infeasible, we combined (1) keywords search, (2) information retrieval, and (3) topic modeling techniques to extract a subset of commits that are likely to contain performance-related issues. This "candidate" list of commits was then manually analyzed for validation (to check for their relevance) and classification (for the creation of the taxonomy).

The next subsections detail the steps of our methodology as well as the semi-automated tools we employed to identify the relevant commits.

3.2.1. Initial keyword selection

We relied on a keyword search to extract the list of candidate commits for the manual analysis. That is, we manually selected a set of keywords likely to be used in performance-related issues, and we searched for commits containing at least one of these keywords. We created an initial set of keywords based on the authors' experience. Further, we expanded the list of keywords by including keywords from the literature related to embedded systems and performance, i.e., Smith and Williams (2002a, 2000). We then manually analyzed a sample set of commits and added any keywords we also deemed relevant. Table 3 reports the resulting list of 22 keywords. These keywords were also validated with domain experts from the H2020 COSMOS project (COSMOS, 2021).

3.2.2. Keyword set expansion with information retrieval and topic modeling

To ensure we gathered all performance-related commits from each project, we applied both Information Retrieval (IR) and Topic Modeling (TM) techniques to expand our initial keyword set.

Data preprocessing. For each project, we downloaded all commit messages and code changes. Then, we pre-process these artifacts by tokenizing the commit message, removing stop words, and stemming. First, *tokenization* aims to extract words in the text and remove non-relevant characters, such as punctuation marks, special characters, and numbers (Panichella, 2019). As commit messages can contain code snippets, we split compound names (i.e., identifiers) into tokens using *camel case* and *snake case* splitting (Panichella et al., 2016). For example, the method name `get_data` will be split into the two tokens `get` and `data`.

Table 3
Keywords in our initial steps along with their description.

Keywords	Explanation
performance, runtime	As the focus of the research is performance, the keywords 'performance' and 'runtime' link directly to any commit that is related to this area.
slow, slower, slowing, fast, faster, increase, decrease	These adjectives are used to indicate a change in the commit in the described way. This could indicate a performance improvement or decrease.
memory, memory-heap, memory-leak, memory leak, bottleneck, overhead, deadlock, livelock, infinite, speed, impasse, hang, stuck	These keywords are chosen based on previous experience, books regarding performance, and found during the analysis phase.

We further applied *stop-word list* and *function* to remove words that do not contribute to the semantic content of the analyzed text (De Lucia et al., 2013; Panichella et al., 2015). The former is a list of generic words (i.e., prepositions, articles, auxiliary verbs, and adverbs) that are commonly found in any text, thus, not providing any useful information. Our stop-list includes the standard list for the English language (Panichella et al., 2015), plus a list of words that are specific to the programming languages (i.e., reserved keywords like `class` in `Java`). The stop-word function instead removes words that are too short (De Lucia et al., 2013), i.e., that contain less than three characters.

Finally, we applied *stemming* algorithms to reduce the words to their root form. To this aim, we used the Porter stemming algorithm (Porter, 1980).

Topic modeling. To better understand the context in which the keywords are also used, we also applied topic modeling (Blei et al., 2003; Panichella, 2019), specifically Latent Dirichlet Allocation (LDA), to each project separately. Fig. 1 depicts the overall process. Given a software project A ①, we apply LDA ② on all commit messages after the pre-processing steps described above.

LDA requires setting three hyperparameters, namely α, β, and the number of topics k (Blei et al., 2003). For this study, we performed unsupervised hyper-parameter tuning using genetic algorithms based on the silhouette coefficient, as suggested by literature (Panichella et al., 2013; Panichella, 2019). The resulting parameter values are as follows: $k = 20$, $\alpha = 0.5$, and $\beta = 0.2$. To address the probabilistic nature of LDA, we ran Gibb's iterative process 10 times with different random seeds (Panichella, 2019). Therefore, the topics obtained are the average across the repetitions. The topic modeling returned a list of 20 topics per project, each containing 20 words ③ that are statistically related to one another according to LDA.

Using this methodology, we could quickly analyze each project's topics and words that might be of interest for further analysis. We reviewed each word ④ and identified words that are related to performance issues. These words were then added to the list of keywords for the next step of the methodology.

While we started with an initial set of 22 keywords, this procedure led us to add 28 additional keywords after applying stemming. This means that keywords like "*time*" will be used to represent all words (in the commit messages) with the same root, such as "*timing*", "*timed*", "*timer*", and so on.

For the sake of brevity, we do not report the full list of keywords in this paper. However, interested readers can find the list of keywords in our online appendix (van Dinten et al., 2023).

3.2.3. *Extraction of candidate commits*

The keywords from the previous steps have then been used to extract commits that should be manually analyzed. To this aim, we created the tool `PyRock`. It mines the history of a Git project and returns all commits whose message contains one of the targeted keywords. Fig. 2 visualizes the tool's architectural design.

`PyRock` requires two input parameters: (1) the repositories list and (2) the local/remote flag. The former is a list of repositories on which we want to perform the automated code history analysis. The latter parameter indicates whether the repositories are available locally or `PyRock` needs to fetch them remotely. For the local mode, the user

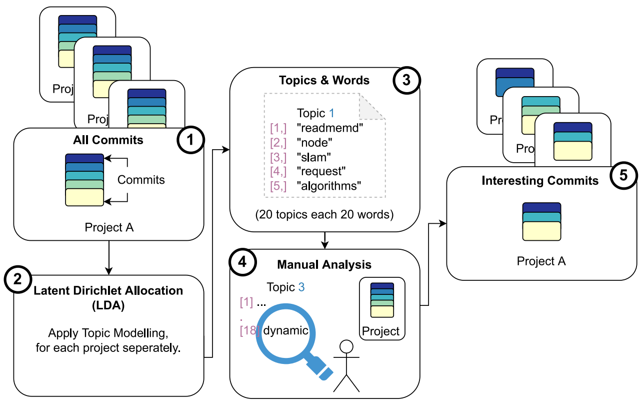


Fig. 1. Topic-Modeling process.

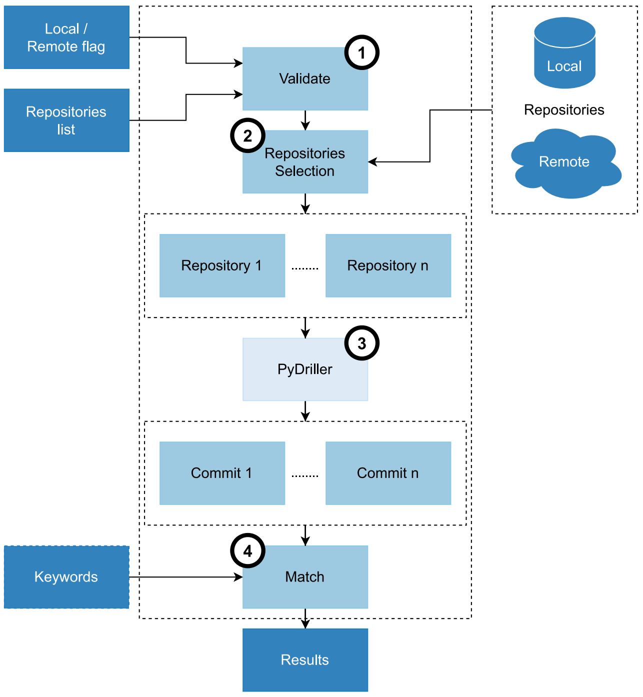


Fig. 2. PyRock Architecture.

also needs to provide the directory in which the local repositories are stored.

These user inputs are first validated by `PyRock`'s Validate module (see ① in Fig. 2). This module checks that the user has indicated which mode (local/remote) to run and which repositories to analyze. In local mode, `PyRock` will only check locally stored repositories; in remote mode, PyRock will only check remotely located repositories. Further, it is possible to run PyRock with one or a full list of repositories.

After verification, PyRock selects each repository with the Repositories Selection module, see ② in Fig. 2, for initiating the next step. In local mode, this module validates the input data and checks whether the given repositories' location contains the projects presented in the repository list. In remote mode, it checks whether the repositories' remote addresses are reachable.

To mine the history of the repositories (③ in Fig. 2), PyRock utilizes PyDriller (Spadini et al., 2018), a commonly used open-source Python framework for mining Git repositories. PyRock passes the information to PyDriller regarding each repository one at a time.

In the next step, the commit messages returned by PyDriller are passed through the Match module, see ④ in Fig. 2. This module utilizes a keyword file containing a list of keywords that could indicate a potential performance antipattern and formed using IR and TM as described in Section 3.2.2. See Table 3 for the keywords used in this analysis. This module considers any commit message containing at least one of the performance-related keywords as a candidate commit for further analysis. Finally, this module stores and returns the list of collected candidate commits, which are then considered for manual validation.

3.3. Manual analysis

As a result of the previous steps, we obtained a list of 2699 commits potentially related to self-admitted performance issues. Out of these, 1059 commits are related to the keywords in our initial set (see Section 3.2.1). In contrast, the remaining 1640 commits are related to the keywords found by applying information retrieval and topic modeling techniques (see Section 3.2.2).

Two authors of this paper manually analyzed these commits independently following an open coding procedure. The authors followed a rigorous procedure to ensure the quality of the analysis, handling potential conflicts and disagreements. In particular, each validator separately read and analyzed each commit by reading the commit message. If a commit was unclear, the validators further analyzed the code changes, the associated issues, pull requests, and documentation when needed. This allowed us to: (1) identify and remove false positives, (2) check eventual matches with the existing catalog by Smith and Williams (2000), and (3) identify self-admitted performance issues that do not belong to any of the existing CPS antipatterns.

Therefore, our manual analysis process required to follow seven steps as described below:

1. Check the **commit message** for the developer's explanation of what has been done.
2. **Code changes** in the commit. Can we find any of Smith's (2020) CPS antipatterns? Do we see other potential antipatterns?
3. Check if the commit is mentioned in any **issue or pull request**, to understand if the changes are linked to any other changes.
4. In case it is relevant, **read comments and notes** mentioned in the issues and pull requests. What were the design considerations discussed in the comments? More information received regarding the original issue?
5. Read the **documentation** of the changed classes to obtain more information regarding the developers' design considerations potentially.
6. Analyze the **final version** of the file in the main branch to check if the CPS developers modify/revert the changes in the commit under analysis. This can reveal if the changes were accepted or if other issues were found with the proposed solution.
7. In case it is relevant, read the documentation regarding the **software and hardware architecture** of the projects under analysis. To understand the reasoning for changes to the design due to added hardware support.

To further increase the reliability of our analysis, two external validators (students) who are not authors of this paper cross-reviewed 1025 (38%) randomly sampled commits. This sample size was determined using *power analysis* (Triola et al., 2006), leading to a confidence level of 99% and a margin of error below 3.2%. The overall agreement among the validators was 93.88%. In case of disagreements, the validators discussed the reasons for the disagreement and reached a consensus on whether the commit should be considered a performance issue or not and whether it fits in the existing taxonomy or not. In total, the manual analysis took 904 h, including the cross-review process and disagreement resolution.

The complete list of commits analyzed and the results of the manual analysis are available in our GitHub project.[1]

3.4. Results

Table 4 reports the results of our manual analysis for the 2699 commits that are potentially related to performance issues. It reports (1) the number of false positives (e.g., commits not actually related to performance issues), (2) commits related to performance issues that are further categorized based on the existing catalogs. In particular, we clustered the manually-validated (true positives) performance issues based on whether they are related to the antipatterns in the existing catalog by Smith and Williams (2002a) (columns marked with ①), generic performance antipatterns that occur in both traditional software and CPS as suggested by Smith (columns marked with ②), generic antipatterns reported in the literature for traditional software only (columns marked with ③), generic performance issues (columns marked with ④), and performance issues that do not match any of the existing antipatterns ⑤.

As we can observe, the number of false positives (i.e., commits not related to actual performance issues) is quite high, being 75.06% on average. This result was expected since some keywords (e.g., *increase*) can be used in the commit message for different purposes, such as increasing the number of features or increasing the waiting time for concurrency issues. However, we have kept ambiguous keywords in the search query to avoid missing relevant commits.

First and foremost, we can observe that performance issues are widely common in the analyzed projects. Only four out of 14 analyzed projects do not have any performance issues. The project with the highest number of performance issues is PX4-Autopilot, with 653 performance issues out of 1869 analyzed commits. Note that this project has the largest code history among all projects in our study. Nevertheless, instances of performance issues can be found in the other nine projects as well.

Finding 1. Self-admitted performance issues are widely-common in 71% of analyzed projects. Their frequency of occurrence in these projects ranges from 2 up to 653.

With regard to the antipatterns, we could find instances of antipatterns by Smith in the selected projects. However, they are not the most common, representing only 8 out of 770 performance issues identified in our manual analysis. Generic performance issues (e.g., not specific to CPS) cover in total 124 of the 770 performance issues. Finally, 638 performance issues (97.70%) are not covered by any of the existing catalogs within CPS and traditional software.

Finding 2. The existing catalog of antipatterns for CPS by Smith and Williams (2002a) characterizes only 8 out of 770 performance issues identified in our manual analysis.

CPS projects are often characterized by other performance issues that are not specific to CPS projects and that occur in more traditional software projects. The most common generic performance issues are

[1] https://github.com/ciselab/CPS_repo_mining.

Table 4
Number of manually analyzed commits and their classification in false positives (commits not related to performance issues) and true positives. Please note that there can be more than 1 performance issue per commit. The latter are classified based on the existing performance antipattern catalogs. The column *Performance Issues To Further Analyze* denotes the performance issues that do not match any of the existing antipatterns.

				①			②						③								④			⑤	
Project	Total # Commits	# Analyzed Commits	% False Positives	Are We There Yet	Is Everything OK	Where Was I	Unnecessary Processing	How Many Times Do I Have to Tell You?	More is Less	The Ramp	Museum Checkroom	Falling Dominoes	Excessively Flexible Storage	For-If	Extraneous Fetching	Unbuffered streams	Big-load	Hard Coding	Bottleneck	Code Duplication	Deadlock	Recreate Objects	Not Deallocating	Performance Issues To Further Analyze	Total number of Performance Issues
Android App Manager	231	7	85.71	–	–	1	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	1	2
Arduino-esp32	1747	140	81.43	–	–	1	2	–	–	–	–	–	–	–	–	–	–	2	–	–	1	–	8	15	29
Cylon	1323	21	100.00	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	0	0
DroneKit Android	5810	99	88.89	–	–	–	1	–	–	–	–	–	1	–	–	1	1	–	–	–	–	1	–	6	11
Grbl	699	217	91.71	–	–	–	1	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	26	27
Johnny Five	3355	147	89.12	–	–	–	–	–	–	–	–	–	2	–	–	–	–	–	–	–	–	–	–	12	14
Node AR Drone	281	48	97.92	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	2	2
PX4-Autopilot	35,537	1869	69.40	1	3	1	12	1	–	–	–	–	2	–	–	1	–	55	1	2	5	–	18	551	653
Robonomics-Contracts	502	7	100.00	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	0	0
Robonomics-JS	68	0	0.00	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	0	0
Turtlebot	1142	24	83.33	–	1	–	–	–	–	–	–	–	–	–	1	–	–	1	–	–	–	–	–	1	4
Turtlebot 3	526	27	37.04	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	21	21
Vacuum Robot Mark II	54	2	100.00	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	0	0
Valetudo	1043	91	92.31	–	–	–	–	1	–	–	–	1	–	1	–	–	–	–	–	–	–	1	–	3	7
Total	52,318	2699	75.06	1	4	3	16	2	0	0	0	1	5	1	1	2	1	58	1	2	6	2	26	638	770

related to unnecessary processing, memory, and network usage. However, they characterize only 16.10% of the total number of performance issues.

Finding 3. 97.70% of the identified self-admitted performance issues are not covered by any of the existing catalogs within CPS and traditional software.

4. Extending the taxonomy of CPS antipatterns

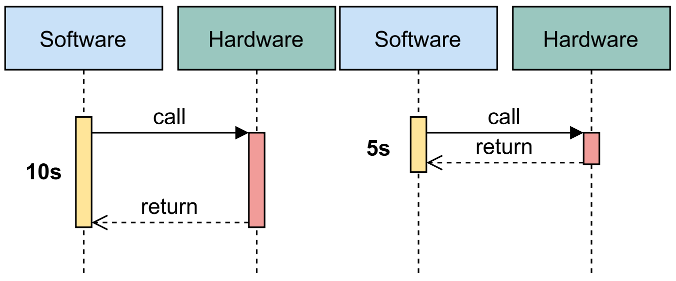
Given the large percentage of performance issues that do not match existing antipatterns, we manually analyzed these instances in order to identify common patterns and characteristics. To this aim, two authors (hereafter referred to as *annotators*) of this paper manually analyzed the 638 performance issues that remained `unclassified` after the first manual analysis (see Section 3.3). Each *annotator* manually analyzed each of the 638 performance issues by inspecting (1) the commit message, (2) the code changes, and (3) the documentation of the source software project (if available). The *annotators* were asked to identify the main reason/cause for the performance issue and to provide a short description of the antipatterns and the reason why it was identified. At the end of this procedure, the two *annotators* compared their results, discussed the differences, and agreed on a final classification.

The resulting classification is reported in Table 5. The table also includes the occurrence of each category across all projects in our study. In reporting our results, we distinguish between performance issues and performance antipatterns based on the number of projects in which their instances occur. We consider reoccurring performance issue patterns confirmed as an antipattern if they are found in more than two projects. This is critical to discriminate between issues specific to a single project (project-specific) from issues that can occur in CPS systems in general (common antipatterns). Instead, performance issues that have a negative effect on the system but occur in only one project are not considered a confirmed antipattern. Further studies are needed to confirm whether these project-specific issues might be common in other CPS systems or not.

Table 5
Number of instances of Potential new CPS-PA from the manually analyzed commits.

Project	Hard Coded Fine Tuning	Magical Waiting Number	Fixed Communication Rate	Rounding Errors	Bad Noise Handling	Delayed Sync With Physical Events	**Total**
Android App Manager	–	1	–	–	–	–	1
Arduino-esp32	8	5	1	1	–	–	15
Cylon	–	–	–	–	–	–	0
DroneKit Android	2	1	1	2	–	–	6
Grbl	14	9	1	2	–	–	26
Johnny Five	4	6	–	1	1	–	12
Node AR Drone	–	2	–	–	–	–	2
PX4-Autopilot	363	122	51	4	10	1	551
Robonomics-Contracts	–	–	–	–	–	–	0
Robonomics-JS	–	–	–	–	–	–	0
Turtlebot	–	–	–	–	–	1	1
Turtlebot 3	8	1	12	–	–	–	21
Vacuum Robot Mark II	–	–	–	–	–	–	0
Valetudo	–	3	–	–	–	–	3
Total	399	150	66	10	11	2	638

In the remaining parts of this section, we discuss six identified performance issues: *Magical Waiting Number*, *Hard Coded Fine Tuning*, *Fixed Communication Rate*, *Bad Noise Handling*, *Rounding Errors*, and *Delayed Sync with Physical Events*. We discuss each of these identified performance issues by providing (1) an explanation of the potential antipattern, (2) providing an example found in the Git history of one of the projects in our dataset, (3) discussing whether such an issue can



(a) Setup A (10s) (b) Setup B (5s)

Fig. 3. Magical Waiting Number, visualization of two hardware targets with different durations needed to complete calculations.

be considered an antipattern, and (4) proposing a solution to mitigate the issue.

4.1. Magical Waiting Number

This SPA refers to the lack of a proper waiting time in the CPS when interacting with hardware. When the CPS sends a request or invokes a module in the hardware, it needs to correctly estimate the time it takes for the hardware to finish the task and, if applicable, return the response. We detected many scenarios in our analysis in which the CPS developers either (i) mistakenly did not consider adding a waiting time when sending a request to hardware, or (ii) put a hard-coded incorrect global value for the time it expected the hardware devices to respond.

Example. Fig. 3 shows a visualization of this potential antipattern. Here, two different hardware targets are shown, each with a different amount of time needed to complete their calculations. If only one waiting time is manually set, no matter what type of hardware is used, a waiting time of 10 s will be used for both of them.

From this example in Fig. 3, when used in combination with hardware that could have handled a one-second waiting time, there is a nine seconds unnecessary delay. This could slow down the system in multiple ways, for example: the information from the hardware arrives slower than it could, resulting in other processes waiting for this information before continuing. As another example, a thread could be blocked, waiting for this information for an unnecessary duration. Depending on how often this information from the hardware is required, the system's performance will be more heavily influenced.

As an example, a reported issue in the Valetudo project[2] exposes a bug in which sending a request to the Viomi robot vacuum cleaner[3] to change the time zone, takes the entire connection between the robot and the controller down. The root cause of this performance bug is the little timeout considered by the CPS to complete the setting time zone task. According to the discussions about this bug in the Valetudo repository,[4] this task can take about 10 s. Hence, as presented in Listing 1, this bug is fixed by increasing the timeout to 12 000 ms.

```
this.sendCommand("get_prop", ["timezone"],
    {timeout: 12000}).then((res) => {
    if (res.length > 0) {
        const timezone = res[0];
        if (timezone !== 0) {
            // Set timezone to UTC
            this.sendCommand("set_timezone",
    [0],
            {timeout: 12000}).then(_ => {
                Logger.info(
                "Viomi timezone adjusted to
    UTC");
            });
        }
    }
});
```

Listing 1: Code snippit of a commit that added a timeout to fix an issue that was causing the connection to be broken.

As one commentator replied in the discussion *'If this is an actual timeout could we move this hardcoded 12 000 value somewhere into config.json?'*,[5] we recommend this as the first step to mitigate this antipattern. In such a configuration file, the timeout could be made to reflect the different hardware architecture if there is a need; see Listings 2 and 3.

```
const currentArchitecture = "
    architectureOlderVersion";
const configArchitectures = require('./config
    .json');
const durationTimeout = configArchitectures.
    currentArchitecture.durationTimeout;

this.sendCommand("get_prop", ["timezone"],
    {timeout: durationTimeout}).then((res) => {
    if (res.length > 0) {
        const timezone = res[0];
        if (timezone !== 0) {
            // Set timezone to UTC
            this.sendCommand("set_timezone",
    [0],
            {timeout: durationTimeout}).then(_
    => {
                Logger.info(
                "Viomi timezone adjusted to
    UTC");
            });
        }
    }
});
```

Listing 2: Proposed solution in original code snippet.

```
{
    "architectureOlderVersion": {
        "durationTimeout": 120000
    },
    "architectureNewestVersion": {
        "durationTimeout": 90000
    }
}
```

Listing 3: Snippet of JSON file as part of the proposed solution.

Another proposed solution would be to receive a message when the computation is ready. An example of this proposal can be seen in Listing 4.

```
const setTimeZone = async () => {
    const res = await sendCommand("get_prop",
     ["timezone"]);

    if (res.length > 0) {
```

[2] https://github.com/Hypfer/Valetudo/issues/799.
[3] https://www.viomi.com.
[4] https://github.com/Hypfer/Valetudo/pull/806.
[5] https://github.com/Hypfer/Valetudo/issues/799#issuecomment-946773526.

```
        const timezone = res[0];
        if (timezone !== 0) {
            // Set up an event listener
            const eventHandler = () => {
                Logger.info("Viomi timezone
    adjusted to UTC");
                removeEventListener("ready",
    eventHandler);
            };
            addEventListener("ready",
    eventHandler);

            // Set timezone to UTC
            await sendCommand("set_timezone",
     [0]);
        }
    }
};
```

Listing 4: Proposed solution containing an eventlistener to receive a message when ready. This example visualizes our second proposed solution, but would require further adjustments in the codebase before usage.

Is it a performance antipattern? As presented in Table 5, we have detected Magical Waiting Number instances in 150 commits that we have manually analyzed in this study. These commits are from nine different projects: PX4-Autopilot, Valetudo, Johnny Five, Node AR Drone, Grbl, Arduino-esp32, Android App Manager, DroneKit Android, and Turtlebot 3. The projects are developed in four different programming languages and used for various applications (e.g., controlling drones, vacuum cleaners, or robotic programming). Hence, given that this kind of bad coding practice is frequently found in various projects in our analysis, we consider Magical Waiting Number as a new CPS-Performance Antipattern (PA).

Proposed solution(s) The refactoring solution should aim to assign the waiting times dynamically according to the target hardware module.

Magical Waiting Number
In short, the key property of this antipattern is: an added waiting time to resolve an otherwise, potential, timing issue.

4.2. Hard Coded Fine Tuning

This potential antipattern occurs when a setting or value is manually tweaked to improve the CPSs performance. In these cases, the result of the software performance is verified by seeing the end result of the change, rather than a calculated or documented reason. Making a potential performance improvement with such a method can be a slow process, as an adjustment to the same value is done over multiple commits. We observed this antipattern based on the comments in the connected issues and assumed from the changes made to the same variable with the time/date of the changes.

Example. In PX4-Autopilot, we detected two linked commits[6],[7] where multiple stack sizes are reduced to free up some memory, see also Listings 5 and 6. However, one of the software modules in this CPS (sdlog) needed that amount of memory. There is no test to ensure the resources required by sdlog are upheld, and thereby, the build process did not fail after this memory reduction, the developers noticed the performance issue after implementation. These changes show that they are tweaking the settings manually to see the results to free up some memory.

[6] https://github.com/PX4/PX4-Autopilot/commit/ab63a77edf78a198117757a1d5e2dbe34cde1263.

[7] https://github.com/PX4/PX4-Autopilot/commit/edd2715f84532f6c4c748cc97f0fe8a2982aa885.

```
deamon_task = task_spawn("sdlog",
            SCHED_DEFAULT,
            SCHED_PRIORITY_DEFAULT - 30,
            2048,
            sdlog_thread_main,
            (argv) ? (const char **)&argv[2] :
      (const char **)NULL);
```

Listing 5: Example code: an adjustment was made to change the memory stack size to 2048.

```
deamon_task = task_spawn("sdlog",
            SCHED_DEFAULT,
            SCHED_PRIORITY_DEFAULT - 30,
            4096,
            sdlog_thread_main,
            (argv) ? (const char **)&argv[2] :
      (const char **)NULL);
```

Listing 6: Example code: the adjustment made to the memory stack size was reverted back to 4096.

Is it a performance antipattern? As the examples show, manual adjustments are not necessarily the most optimal setting for the system. The process of manual adjustments does indicate that an adjustment could positively impact the performance of the CPS. Outside of the area of CPS, a similar phenomenon could be adjustments of stack size. The reason we consider this antipattern as CPS specific, is the core challenge with CPSs: their real-time response in a real environment. Every time new hardware is added to the system, the hard-coded values need to be fine-tuned to include the new hardware. With an increasing range of hardware, the system needs to support, the constant adjustments could become more difficult to manage.

As reported in Table 5, we detected 399 commits in our manual analysis that strive to set the most optimum setting for the CPS. Given these findings, we consider the Hard Coded Fine Tuning as a new CPS-PA.

Proposed solution(s) The refactoring solution should aim to identify the values that have been tuned for each hardware argument, before the subsequent releases. And adjust these values such that they are easily adjustable based on the build target. Avoid having to use the slowest setting for the list of supported targets.

Hard Coded Fine Tuning
In short, the key property of this antipattern is: hard-coded variables that would, potentially, need adjustment(s) if another hardware support is added to the project.

4.3. Fixed Communication Rate

Many CPS projects contain multiple hardware modules working synchronously together. These hardware modules need to communicate with the minimum latency to make sure that the CPS performs as expected. In these projects, the CPS developers should make sure that this communication happens with the minimum latency to ensure the performance and efficiency of the CPS. However, setting an excessively high communication rate leads to a higher usage rate of resources (for instance, higher energy consumption), which is especially unfavorable for devices with limited energy resources (e.g., drones, robots, and smart vacuum cleaners).

In our analysis, we detected cases where CPS developers set a fixed communication rate between these devices and modules. In some other cases, they set a limit for these communication rates. As we have seen, for example, in a commit from PX4-Autopilot.[8] Later, these developers find scenarios where low communication rates negatively affect the system's performance.

[8] https://github.com/PX4/PX4-Autopilot/commit/81a4df0953e738041d9fdc2b2eb353a635f3003b.

Example. In the DroneKit Android project architecture, Android devices need to communicate with drones for controlling purposes. In this project, the CPS developers set a default communication rate between the Android device and the drone. However, they noticed that this default rate is not enough when the user enters the tuning screen. Hence, in one of the commits,[9] they implemented a dynamic procedure to increase the communication rate when a user opens the tuning screen and returns the rate back to default when they close it. See also Listings 7 and 8 for the code changes.

```java
public static void setupStreamRates(
    MAVLinkClient MAVClient,
      int extendedStatus, int extra1, int
    extra2, int extra3,
      int position, int rcChannels, int
    rawSensors, int rawControler) {
    requestMavlinkDataStream(MAVClient,
        MAV_DATA_STREAM.
    MAV_DATA_STREAM_EXTENDED_STATUS,
    extendedStatus);
```

Listing 7: Code snippet as part of the changes made (by the developers of the project) to adjust the communication rate when the user is in the tuning screen. Filename: MavLinkStreamRates.java

```java
public class TuningFragment extends Fragment
    implements OnTuningDataListner {

  private static final int NAV_MSG_RATE = 50;
  private static final int CHART_BUFFER_SIZE
    = 20*NAV_MSG_RATE; // About 20s of data
    on the buffer
    ...
    @Override
      public void onStart() {
        super.onStart();
        setupDataStreamingForTuning();
      }

      private void
    setupDataStreamingForTuning() {
        // Sets the nav messages at 50Hz and
    other messages at a low rate 1Hz
        MavLinkStreamRates.setupStreamRates(
    drone.MavClient, 1, 0, 1, 1, 1, 0, 0,
    NAV_MSG_RATE);
      }

      @Override
      public void onStop() {
        super.onStop();
        MavLinkStreamRates.
    setupStreamRatesFromPref((DroidPlannerApp
    ) getActivity().getApplication());
```

Listing 8: Code snippets as part of the changes made (by the developers of the project) to adjust the communication rate when the user is in the tuning screen. Filename: TuningFragment.java

Is it a performance antipattern? As is shown in Table 5, we detected 66 commits in our manual analysis that strives to tackle the fixed communication rate. We identified this performance issue in five projects: (i) DroneKit Android (implemented in Java), which provides a framework for developing applications for Android devices to control drones, (ii) PX4 Autopilot (implemented in C++) that enables the automated and manual control of moving devices such as multi-copters, small airplanes, airships, balloons, rovers, boats, and even small submarines, (iii) Arduino-esp32, Arduino ESP32 core (implemented in C), (iv) Grbl, Parallel-port-based motion control for CNC milling (also implemented in C), and (v) Turtlebot 3, a wheeled robot written in Python and C++. Given these observations, we consider the Fixed Communication Rate as a new CPS-PA.

Proposed solution(s) The refactoring action should aim to ensure that the communication rates between hardware components adapt during the operation of CPSs according to the need for communication between the components. However, when considering this proposed refactoring, the CPU and, thereby, power consumption can possibly be affected. It would be interesting for future work to investigate the impact of implementing the proposed solution.

Fixed Communication Rate
In short, the key property of this antipattern is: a fixed communication frequency rate could cause performance issues when set too low, or an unnecessary high energy consumption when set too high.

4.4. Rounding Errors

In some scenarios, CPSs contains software modules that perform calculations related to the physical events (e.g., the exact angle of a robotic arm or the location of a drone) in the project. These calculations should have the highest precision for more accuracy and reliability to prevent any threat to the safety of different processes in the CPS. For instance, one of the known mathematical calculation errors that can endanger the precision of the calculations is a rounding error in which one of the numbers is altered to a type with fewer decimals (Wilkinson, 1994; Frechtling and Leong, 2015).

Examples. In our analysis, we found eight commits in which CPS developers changed the number types in these calculations to increase the calculation precision and prevent rounding errors. As an example, a commit in DroneKit Android[10] changes the types of numbers related to the latitude, longitude, and altitude of the drone from float to double. The message of this commit also indicates that this change is applied to increase the precision of these numbers. At first sight, this bad practice leads to functional issues. For instance, in this example, the miscalculation of the drone's latitude, longitude, and altitude leads to problems in how the CPS functions. However, it can also – indirectly – negatively impact the performance of the CPS. For example, a miscalculation in detecting the proper coordination for the landing of drones can trigger other correcting processes (e.g., recalculating the right coordinate or recalculating other metrics for landing in the new location), which are energy and time-consuming.

Is it a performance antipattern? As presented in Table 5, we identified ten instances of Rounding Errors in our manual analysis. These instances are detected in five projects, two of which are for controlling various types of drones: DroneKit Android and PX4 Autopilot (implemented in C++ and Java). We also found Rounding Errors in Arduino-esp32, Grbl, and Johnny Five (the first two are implemented in C and the last one in JavaScript). As with these three projects, we think that this type of antipattern can be found in any CPS containing mathematical calculations for physical values (e.g., robotics and self-driving cars). For example, depending on the compiler and hardware used, there is a difference in precision and range for using a float (Kelley and Pohl, 2006). Some systems compensate for the functional errors with correction actions; these are pure overhead and cause performance issues or potentially crash the system. Given these findings, we consider Rounding Errors as a new CPS-PA.

[9] https://github.com/dronekit/dronekit-android/commit/2c9d9bc08147b0952eba4b6ef28701641a99bb21.

[10] https://github.com/dronekit/dronekit-android/commit/e29a5fde6f5c871ce956ffe6659e8b34f3d8a5b2.

Proposed solution(s) Assure that the types of variables do not introduce rounding errors for the values passed to the hardware-related methods. For example, review all number types (e.g., int, double, and floats) used in hardware-related code to verify that the appropriate precision is used.

Rounding Errors
In short, the key property of this antipattern is: using insufficient/incorrect data types for high-accuracy calculations.

4.5. Delayed Sync with Physical Events

This issue refers to scenarios in which the CPS does not notify running software processes and threads when an unexpected physical event occurs. We detect two cases in our analysis that expose this performance issue.

Examples. We detected this performance issue in TurtleBot and the PX4-Autopilot project. TurtleBot is a personal multi-functional robot kit with different input and output ports, including a USB port for connecting it to other controlling devices. In the detected issue, the driver node for communicating via this USB port is not notified and is stopped if the USB connection is disconnected. In this scenario, if the user plugs in another device, the driver node considers the new device as the previous one. This issue is fixed in one of the commits we manually analyzed in this study.[11] This commit assures that the driver node fast-fails when the USB device is disconnected. This change also ensures that the driver node does not mistakenly detect and re-associate with a newly plugged-in USB device as the previous USB device.

Is it a performance antipattern? Since we identified two instances of this issue in our analysis (see Table 5), we cannot confirm if this performance issue commonly occurs in CPSs. Hence, for now, we do not consider Delayed Sync with Physical Events as an antipattern.

4.6. Bad Noise Handling

Several hardware devices contain sensors that require software-based noise-handling techniques to collect accurate data, as the input collected from these devices can be noisy. However, in some situations, the noise-handling techniques are not efficient, which entails that the CPS needs to collect more data to increase the accuracy, but this also leads to an increase in I/O resource consumption. Similar to the previous section, we detected this performance issue in only two projects. There were a total of ten instances found across these two projects.

Examples. We detected this performance issue in PX4-Autopilot and the Johnny Five project. The Johnny Five project is a JavaScript robotics programming framework working with various hardware. This project handles the noises by selecting the median value collected from sensors. However, by looking at the changes in the code history of this project,[12] we noticed that the implemented median calculation was not efficient enough. One of the commits[13] in this project improves the noise handling procedure with a faster and more stable technique.

Is it a performance antipattern? Similarly to the previous performance issue, the Bad Noise Handling is detected in only two projects in our analysis. Therefore, we currently cannot confirm that this performance issue is common enough to be considered an antipattern.

[11] https://github.com/turtlebot/turtlebot/commit/f2d46b705722b61948313e3f2ec167dcaeeb3359.
[12] https://github.com/rwaldron/johnny-five/pull/138.
[13] https://github.com/rwaldron/johnny-five/commit/d3541a70d7767e52fb9aa67b32d9f32669abf45f.

Table 6
Order of CPS-PAs occurrences detected in our study.

Antipattern	Occurrence	Of total CPS-PA occurrences
Hard Coded Fine Tuning	399	63.03 %
Magical Waiting Number	150	23.70 %
Fixed Communication Rate	66	10.43 %
Rounding Errors	10	1.58 %
Is Everything OK	4	0.63 %
Where Was I	3	0.47 %
Are We There Yet	1	0.16 %
Total	633	

4.7. Discussion

There is a notable relationship between the Magical Waiting Number, Hard Coded Fine Tuning, and Fixed Communication Rate antipattern, as they relate to possible occurrences in which multiple tweaks are necessary in order to – or attempt to – find the right value for the system. From our observations, these tweaks seem to be the results of feedback after running it on real hardware, not run in a simulation or the results from unit- or integration-tests failures.

The main differences are the type of impact each has on the system. The Magical Waiting Number antipattern reduces the benefits of using faster components, potentially even crippling it by using out-of-date information. Regarding the Fixed Communication Rate antipattern, it impacts the balance between performance and energy consumption.

The Hard Coded Fine Tuning antipattern's main aspect is the hard-coded values that have the potential to undergo – or have already been undergoing – changes in order to attempt to optimize performance, which would have an impact on multiple aspects of the system. These are not related to the timing of hardware components (such as the Magical Waiting Number antipattern), or the frequency of communication (as is with the Fixed Communication Rate antipattern). But these smaller tweaks are due to, for example, stack memory usage.

These differences between the antipatterns and their impact, led us to the decision to have these as separate CPS-PAs, rather than subsections of a single antipattern.

4.8. Revisiting RQ1 and RQ2

Summarizing the discussion of each identified CPS-PA and results, we noted the occurrence of four previously undescribed CPS-PAs and three from Smith (2020) **(RQ1)**. Of these CPS-PAs, the Hard Coded Fine Tuning antipattern is the most prevalent with 63.03% of the total number of CPS-PA occurrences, followed by the Magical Waiting Number antipattern with 23.70%. A short overview of the detected CPS-PA is shown in Table 6 **(RQ2)**.

5. Study II – automated detection of the new CPS antipatterns

As we have seen in previous studies (Smith and Williams, 2000; Moesus et al., 2019), antipatterns can negatively impact a system's performance and stability. Informing the developer of existing antipatterns in their system gives them the ability to resolve the issues (Aleti et al., 2018; Calinescu et al., 2017), and therefore possibly improve the software's code quality (Gamma et al., 2009), performance (Trubiani et al., 2014), and stability (Smith and Williams, 2002a).

In this section, we propose a novel approach to statically detect the two most frequently occurring CPS-PA from our Study I (see Section 3), in particular, the *Magical Waiting Number* and *Hard-Coded Fine Tuning* antipatterns. We focus on these two antipatterns since they represent the large majority (71.30%) of the instances of performance issues we have identified in Study I (see Tables 4 and 5) and CPS-PA occurrences (86.73%).

To detect these two antipatterns, we rely on static analysis rather than dynamic analysis as done by Pinciroli et al. (2021). Our choice is due to the fact that static analysis is faster since it does not require re-running the code on the hardware or in a simulated environment, nor the test cases, which are usually very expensive to run (Birchler et al., 2022, 2023). Furthermore, it does not require to re-build old releases of the CPS systems, which can be very challenging due to obsolete hardware, dependencies, and libraries (Hassan et al., 2017; Zaidman et al., 2008, 2011; Khatami and Zaidman, 2023). However, static analysis has limitations as it requires manually validating the raised warnings and removing false alarms (Kharkar et al., 2022).

In this section, we first describe our approach for detecting Magical Waiting Number and Hard Coded Fine Tuning (Section 5.1), and then we evaluate against a benchmark (Section 5.2) of nine open-source CPS projects not considered in Study I.

5.1. Our approach: AP-spotter

We have implemented two detection strategies for the Magical Waiting Number and Hard Coded Fine Tuning patterns in our tool, `AP-Spotter`. Details of these strategies are in the following subsections (Sections 5.1.1 and 5.1.2). Our tool employs rule-based static analysis techniques to detect performance issues in CPSs. In contrast to dynamic detection approaches, static analysis techniques do not require high computational effort to monitor the application during execution. See Fig. 4 for the software architecture of our tool.

`AP-Spotter` runs over a project (refer to ① and ③ in Fig. 4) in search of one specific antipattern (②) at a time. It analyzes the project directory and selects the source code files (such as C++ classes) that are of interest for the specific antipattern under analysis. For instance, for the Magical Waiting Number antipattern, `AP-Spotter` selects the files that contain the `wait()` method or similar method/function calls for hardware–software communication and multiprocessing (step ④). Since we are interested in antipatterns at the source code level, `AP-Spotter` skips modules related to Git, Azure, Docker files, documentation, web pages, Gradle usage, tests, test data, examples, and templates. After this selection process, `AP-Spotter` creates an Abstract Syntax Tree (AST) using ANTLR4 (Parr, 2013) ⑤. Then, it analyzes the structure of the AST in search of the antipattern (⑥-⑦) for which it was run. Our tool is publicly available on GitHub.[14]

The parser used in this tool is specifically for C++ projects, but there are parsers available for other languages. This tool has been designed to process an Abstract Syntax Tree (AST) resulting from any parser; thereby making it possible to be extended to other languages for future research.

5.1.1. Detecting Magical Waiting Number

As described in Section 4.1, the Magical Waiting Number antipattern occurs when two events happen: (1) a software component sends a request to hardware, and (2) the software waits for a fixed amount of time or does not wait at all to read the response from the hardware.

To detect this antipattern, we first identify source code files that import drivers for the hardware; hereafter, we refer to these files as *candidate files*. We then check whether these two events described above occur using specific rules (regular expressions). An overview of the method for detecting this antipattern is presented in Fig. 5. This procedure contains three phases and is repeated for each candidate file. The first phase involves identifying the requests sent to hardware, in which the detector analyzes each file to determine if it has any direct request to hardware. In the second phase, our detector identifies waiting commands after sending the request, i.e., statically identifying waiting commands such as sleep or timeout methods. Finally, our detector analyzes the code history of the CPS to determine whether

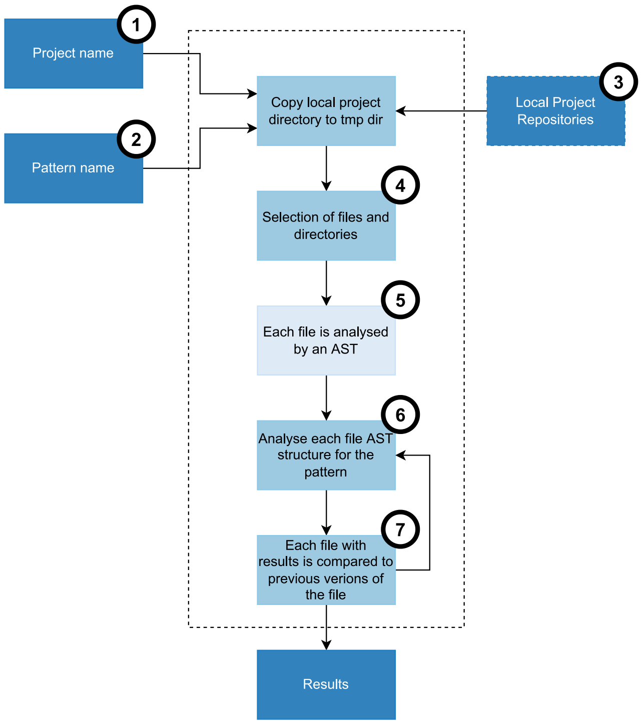

Project name
1
Pattern name
2
Copy local project directory to tmp dir
Local Project Repositories
3
Selection of files and directories
4
Each file is analysed by an AST
5
Analyse each file AST structure for the pattern
6
Each file with results is compared to previous verions of the file
7
Results

Fig. 4. `AP-Spotter`.

the waiting time was both hard-coded and manually changed in prior commits.

Identifying the requests sent to hardware (Phase 1 in Fig. 5) As the first step, the detector checks if it can find at least one request to the hardware in the given candidate file (condition ① in Fig. 5). The command used for sending the request to hardware varies depending on the language and the project. For instance, PX4-Autopilot, a well-known open-source CPS for controlling drones, is implemented in C++ and uses SerialLib[15] for serial communication between modules, or MAVLink[16] to communicate with MAVLink-based vehicles.[17] We designed regular expressions based on the documentation of these libraries to find the send requests in a given source code file; in case of a regex match, the file is considered as a *candidate file* to further check for the presence of the Magical Waiting Number antipattern.

Suppose the detector cannot find any hardware request. In that case, it assumes that it is not possible to find the Magical Waiting Number in this file, stops the process, and continues with the next source code file in the project repository. However, if it finds at least one request, it enters the second phase of the detection procedure.

Identifying the waiting commands after the request (Phase 2 in Fig. 5) The second phase starts by searching for the waiting command after the request to hardware (condition ② in Fig. 5). If it cannot find any waiting command, it checks if the system uses any method of detecting that the sent request has been finalized before requesting the results (condition ③ in Fig. 5). If it finds no method, the detector identifies the request command as a Magical Waiting Number; as the results might be read before it has completed the latest cycle of gathering results. In contrast, the detector assumes that it is not an antipattern if no usage can be found. If the detector manages to find any waiting command in condition ②, it checks the numeric value used as the amount of time that the system needs to wait for the request

[14] https://github.com/ciselab/CPS_SPA_Detection_Tool.

[15] https://github.com/imabot2/serialib.

[16] https://mavlink.io/en/.

[17] https://microsoft.github.io/AirSim/mavlinkcom/.

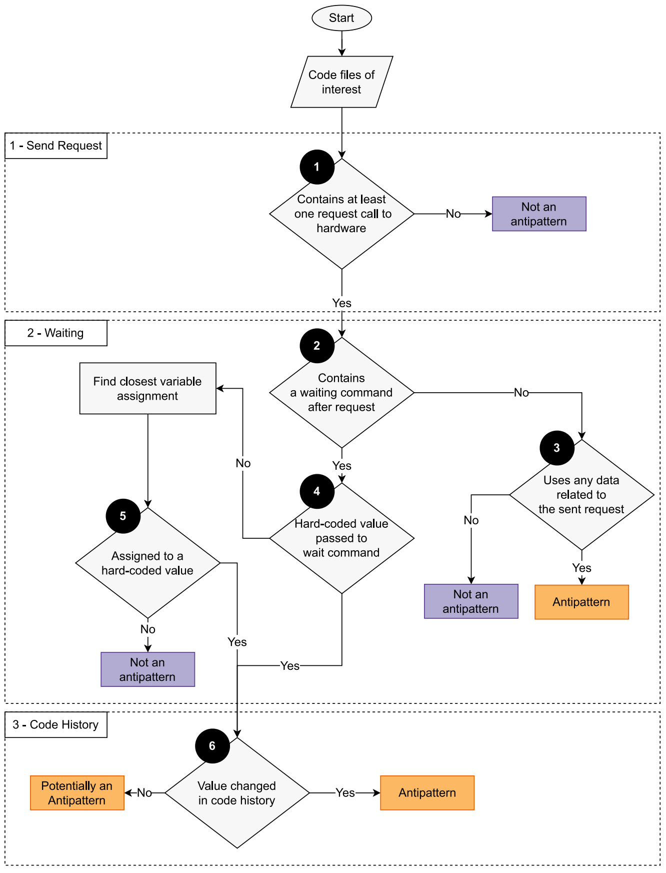

Start
Code files of interest
1 - Send Request
1
Contains at least one request call to hardware
No
Not an antipattern
Yes
2 - Waiting
2
Contains a waiting command after request
No
Find closest variable assignment
3
Uses any data related to the sent request
No
Not an antipattern
Yes
Antipattern
Yes
4
Hard-coded value passed to wait command
No
5
Assigned to a hard-coded value
No
Not an antipattern
Yes
Yes
3 - Code History
6
Value changed in code history
No
Potentially an Antipattern
Yes
Antipattern

Fig. 5. General overview for detecting the Magical Waiting Number antipattern in CPS.

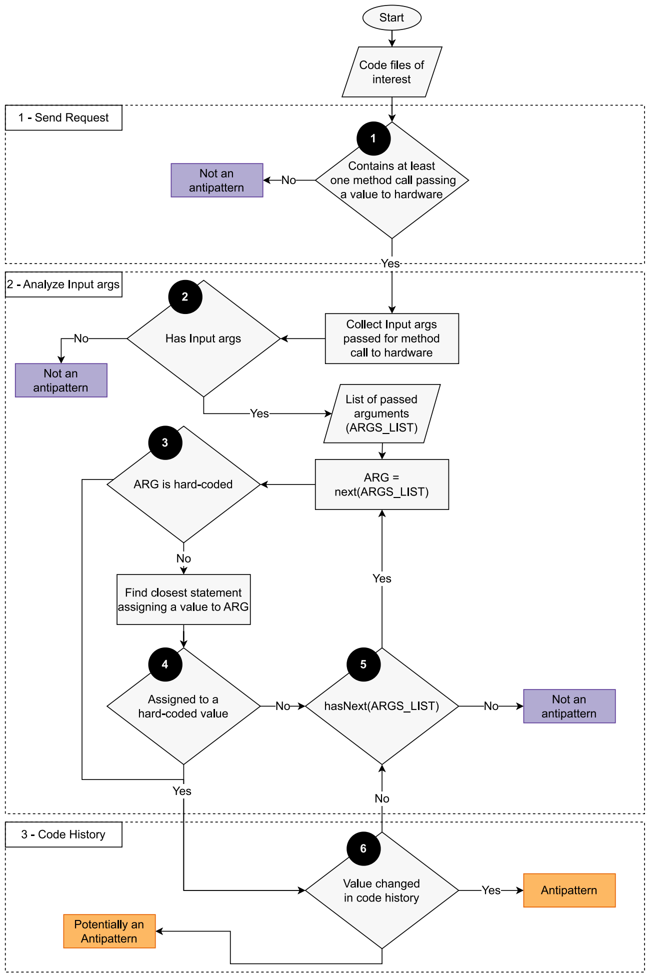

Start
Code files of interest
1 - Send Request
1
Contains at least one method call passing a value to hardware
No
Not an antipattern
Yes
2 - Analyze Input args
2
Has Input args
Collect Input args passed for method call to hardware
No
Not an antipattern
Yes
List of passed arguments (ARGS_LIST)
ARG = next(ARGS_LIST)
3
ARG is hard-coded
No
Find closest statement assigning a value to ARG
4
Assigned to a hard-coded value
No
5
hasNext(ARGS_LIST)
Yes
No
Not an antipattern
Yes
No
3 - Code History
6
Value changed in code history
Yes
Antipattern
Potentially an Antipattern

Fig. 6. General overview for detecting the Hard Coded Fine Tuning antipattern in CPS.

sent to the hardware (condition ④ in Fig. 5). If a fixed hard-coded value is used, the detector enters Phase 3. However, if a variable is used to set the waiting time, it continues Phase 2 by finding the closest statement that assigns a value to this variable. Then, it checks whether the assigned value is hard-coded (condition ⑤ in Fig. 5). If the variable is assigned dynamically, the detector decides that this pattern is not a Magical Waiting Number antipattern. However, if the value is assigned a hard-coded fixed value, the detection process enters Phase 3.

Analyze the code history (Phase 3 in Fig. 5) If the code under analysis is still a potential antipattern (AP) according to Phase 2, we start the last phase. The first condition happens when a hard-coded fixed value is directly passed as the waiting time to the wait method (true branch of condition ④ in Fig. 5). The second condition regards variables that are assigned to a hard-coded value and passed as the waiting time (true branch of condition ⑤ in Fig. 5). In both cases, the detector checks if the hard-coded value used as the waiting time is changed in the code history (condition ⑥ in Fig. 5). If this hard-coded value is changed, it shows that developers had to change the waiting time because they either detected a physical event or a specific hardware module that the previous waiting time was not suitable. Hence, the detector considers it as the Magical Waiting Number antipattern. However, if the hard-coded value is not changed in the code history, we cannot be sure if there is any scenario in which the current hard-coded value does not work, and thereby, the detector does consider it as a potential antipattern.

5.1.2. Detecting Hard Coded Fine Tuning

Detecting this performance antipattern in CPSs requires (1) identifying the method calls that pass any numeric values to hardware (e.g., requests, property setters, etc.); (2) checking whether the passed numeric arguments are hard-coded in the code; and (3) checking if the hard-coded value used for these arguments is changed in the project's code history. This antipattern can occur in any file; thus, we first need to identify the code files of interest (same as the files used in Section 4.2).

Fig. 6 illustrates the detection procedure for the Hard Coded Fine Tuning antipattern. Similar to the detection procedure for Magical Waiting Number, this detection method has three phases that can be applied to each interesting code file. The first phase identifies hardware-related methods, i.e., any request or method call passing numeric values to hardware modules. The second phase analyzes input arguments, where the detector analyzes each of the numeric input arguments to hardware-related method calls and detects the ones that are assigned from a hard-coded value. Finally, Phase 3 analyzes the code history to check whether the hard-coded values are modified in the CPS's code history.

Identify hardware-related methods calls (Phase 1 in Fig. 6) In the first step, the detector examines each of the lines of code in the interesting files to detect any hardware-related method calls (e.g., requests sent to hardware modules or method calls that set a value in the property of hardware, etc.). This step is represented by condition ① in Fig. 6. Similar to detecting requests to hardware modules for the Magical Waiting Number antipattern, if the detector knows the library used for communication between software and hardware, or one hardware module and another hardware component, it can easily detect hardware-related methods, as it just needs to scan for the particular

Table 7
Projects used with the detection tool AP-Spotter.

Project name	Language	Nr. of commits	Stars	Forks	Domain
AirSim	C++	3523	13.2k	3.8k	Automotive
Carla	C++	5439	7.8k	2.4k	Automotive
Arduino	C++	4236	14.1k	12.5k	Arduino
Arduino-IRremote	C++	684	3.6K	1.6k	Arduino
ArduinoJson	C++	1553	5.7k	1k	Arduino
RFID	C++	512	2.3k	1.3k	Arduino
Ardupilot	C++	53,400	7.4k	12.8k	Drone
CoppeliaSimLib	C++	375	57	28	Robotics
Ardumower	C++	1579	207	126	Wheeled Robot

methods. If the file under analysis does not contain any hardware-related method calls, the detector assumes that it cannot find any Hard Coded Fine Tuning antipattern in this file and continues the detection process with the following interesting files. However, if the file invokes at least one hardware-related method, the detection procedure for this file enters Phase 2.

Analyze input arguments (Phase 2 in Fig. 6) For each of the numeric input arguments passed to the hardware-related method call that is detected in Phase 1, the detector inspects if the passed argument is a hard-coded value (condition ③ in Fig. 6). If this value is hard-coded, the detector enters Phase 3. If the value passed as an argument of the hardware-related method is a variable, the detector finds the closest statement in the code that assigns a value to the variable. If this assigned value is hard-coded (true branch of condition ④ in Fig. 6), the detector enters Phase 3. If assigned dynamically (false branch of condition ④ in Fig. 6, in short: no hard-coded value has been found connected), the detector assumes that this case is not a Hard Coded Fine Tuning antipattern. This comparison is done until all arguments have been checked (⑤ in Fig. 6). If no further arguments are available and no antipattern has been detected (false branch of condition ⑤ in Fig. 6), the detector concludes that there are no Hard Coded Fine Tuning antipattern in this case.

Code history analysis (Phase 3 in Fig. 6) This section is similar to the procedure explained for the Magical Waiting Number antipattern. We examine the code history of the CPS to collect all the modifications to the hard-coded values, which are passed to the hardware-related method calls and requests (condition ⑥ in Fig. 6). These hard-coded values can be passed directly to the hardware-related method call or assigned to variables that are passed later to these methods. If the values are changed – the values within the method call –, it indicates that developers found scenarios in which the hard-coded value was not suitable (hence a Hard Coded Fine Tuning antipattern is detected). If it was not changed, we cannot be sure if this value is not suitable for all the possible scenarios in the operation of the CPS. Therefore, the detector would indicate it as a potential antipattern.

5.2. Empirical evaluation

In this section, we evaluate the precision of the AP-Spotter tool in detecting the Magical Waiting Number and Hard Coded Fine Tuning antipatterns. Therefore, our second study is guided by our third research question:

RQ3: *How precise can our approach detect performance antipatterns?*

5.2.1. Benchmark

To assess our tool, we have selected a set of 9 CPS projects that were not previously used in Study I. This is because we want to avoid any positive bias towards the projects we have manually analyzed to identify the new antipatterns. In particular, we selected 9 projects, whose statistics are summarized in Table 7. These projects differ in their sizes (number of commits, stars, and forks) and application domains. All projects are developed in C++ (a constraint of our tool), are open-source, and are publicly available on GitHub.

It is also worth noticing that six of these projects (i.e., *Arduino*, *Arduino-IR Remote*, *ArduinoJson*, *Carla*, *RFID*, and *Ardupilot*), have been used in a prior study aimed at classifying functional bugs in CPS (Zampetti et al., 2022). These projects have a high activity level and are well maintained, these projects are known by the research community as interesting projects to select when investigating code quality of CPS systems.

Three additional projects are added to the list for analysis: (i) *AirSim*, (ii) *CoppeliaSimLib*, and (iii) *Ardumower*. *AirSim* is a simulation platform by Microsoft and used for AI research and experimentation (Shah et al., 2018), including assessing reinforcement learning methods (Wei et al., 2019) and testing (Li et al., 2021). *CoppeliaSimLib* is a library part of Coppelia Robotics (Coppelia Robotics, 2010), its robotics simulation is i.a., of interest due to its physics engines support (Farley et al., 2022) and known in the robotics community. *Ardumower* is an open-source robotic project of interest in the DIY community. This project has a lower popularity and number of commits compared to some of the other projects selected. These projects have been selected for their industrial usage, DIY community, machine learning components, or popularity by researchers in their domain.

5.2.2. Study setup

To answer RQ3, we ran AP-Spotter on each project and collected the files for which our tool indicates the presence of an antipattern (hereafter called *warnings*). These warnings are potential antipattern or antipattern instances, according to our approach. To assess the detection precision, we manually validated the raised warnings considering (1) all commits (title, message, and code changes) related to the file and statements for which the warning is raised, (2) the project documentation, and (3) GitHub issues and pull-requests. This allowed us to gain more information about the nature and rationale for the applied changes.

For each project, the precision (1) is calculated as follows:

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

where 'TP' is the number of true positives and 'FP' is the number of false positives.

To assess the precision for each antipattern, we calculated the weighted average precision (Min et al., 2002)(2) as follows:

$$AP_{weighted} = \frac{\sum_{i=1}^{n}(TP\%_i W_i)}{\sum_{i=1}^{n} F_i} \tag{2}$$

For the weighted average precision ($AP_{weighted}$), the total number of files of each project is taken into consideration; as a project with a higher number of files, the "True positives" calculation would be more accurate. The formula (2) is as follows: the sum of true positives ($TP\%_i$) times the project's number of files (W_i) as the weight. This is then divided over the total number of files (F_i), these are the sum of files of the projects where we have the precision calculated. A weighted average precision is used to balance the project's size with the project's precision; for example, for the 'Arduino-IRremote' project in Table 8 (Hard Coded Fine Tuning with # of Instances) the percentage of true positives is 100%, but the total number of files is 122. Therefore, we take this into lower consideration than 'Ardupilot' with a true positive percentage of 50.70% with 4515 files.

5.2.3. Results

The total number of warnings that AP-Spotter raised for each project is reported in Table 8. In short, AP-Spotter detects the Magical Waiting Number antipattern in 3 projects, and the Hard Coded Fine Tuning antipattern in 4 projects.

For each project, the number of files containing warnings and the number of instances across these files are shown. In a number of files,

Table 8
Projects analyzed with AP-Spotter and the number of antipattern occurrences found. Finally, the resulting Weighted Average Precision (Min et al., 2002) for each antipattern.

Projects	Total files in project	Hard Coded Fine Tuning: # Files with warnings	TP Files	# Instances	TP Instances	Magical Waiting Number: # Files with warnings	TP Files	# Instances	TP Instances
AirSim	1450	0	–	–	–	7	3 (42.86%)	7	3 (42.86%)
Carla	1289	0	–	–	–	0	–	–	–
Arduino	1310	6	5 (83.33%)	9	8 (88.89%)	0	–	–	-
Arduino-IRremote	122	1	1 (100.00%)	1	1 (100.00%)	0	–	–	–
ArduinoJson	461	0	–	–	–	0	–	–	–
Ardumower	3350	4	3 (75.00%)	8	5 (62.50%)	0	–	–	–
Ardupilot	4515	59	26 (44.07%)	71	36 (50.70%)	5	3 (60.00%)	6	4 (66.67%)
CoppeliaSimLib	1289	0	–	–	–	4	3 (75.00%)	4	3 (75.00%)
RFID	68	0	–	–	–	0	–	–	–
Total	13 854	70	35 (50.00%)	89	50 (50.70%)	16	9 (56.25%)	17	10 (58.82%)
W. Avg. Precision	9297/7254		61.48%		60.98%		59.24%		63.39%

multiple instances of the antipattern are detected. For example, in the Arduino project, the antipattern Hard Coded Fine Tuning is found in 6 files, with a total of 9 antipattern instances. We manually validated each finding into true positives and false positives.

For the Hard Coded Fine Tuning antipattern, the sum of files is 9297. This result is the total number of files of the Arduino, Arduino-IRremote, Ardumower, and Ardupilot projects combined. For the Magical Waiting Number antipattern, this sum of files is 7254. Based on the total number of files from AirSim, Ardupilot, and CoppeliaSimLib. These sum of files are used to calculate the weighted average precision for each antipattern, see at the 'W. Avg. Precision' section in Table 8.

The total tool precision is also calculated using the weighted average precision calculation (Min et al., 2002); combining both the results from Hard Coded Fine Tuning en the Magical Waiting Number Instances results. This results in a total tool precision of 62.04%.

Examples of detected Hard Coded Fine Tuning antipattern instances are: (i) changes to the target's speed after test results[18] and (ii) re-tuning of the gyro sensitivity based on user-provided logs.[19] These stood out as their reasoning, as described in the commit message, clearly relates to tweaking done after received feedback. Either by the community or by test results. In the latter instance, it is unclear which stage of testing was meant by the developer.

Further, examples from the detected Magical Waiting Number antipattern instances are: (i) a hard-coded sleep duration, with no explanation regarding the chosen value, after started listening to Gazebo topics[20] and

(ii) relates to the support for multiple peripherals in the Software in the Loop (SITL).[21] A code snippet from this last example can be seen in Listing 9.

```
void SITL_State::wait_clock(uint64_t
    wait_time_usec) {
  while (AP_HAL::native_micros64() <
    wait_time_usec) {
    usleep(1000);
  }
}
```

Listing 9: ArduPilot SITL code-snippet.

This code snippet shows the possibility of using a waiting function, for which the duration can be given in microseconds. Even though this function seems to support the possibility of requesting any duration in microseconds, the waiting method always goes in steps of 1000 microseconds. This is not clearly described in the code or documentation.

5.2.4. *Discussion and revisiting RQ3*

In this section, we discuss the results of our empirical evaluation.

From our results, we can see that it is possible to automatically detect the Hard Coded Fine Tuning (60.98%) and Magical Waiting Number (63.39%) antipatterns, with a total tool precision of 62.04% **(RQ3)**. We also see a relatively high number of false positives. Reviewing the results to investigate these false positives, shows that two main challenges are causing these problems: (1) issues with building the AST, and (2) the definition of these antipatterns require them to be closely related to the hardware.

For the first point, regarding the AST, the AP-Spotter tool uses an existing library to generate the AST. An incorrectly generated AST will result in the AP-Spotter tool analyzing the information based on an incorrect AST, resulting in a possible false positive. This could be due to an issue in the library, or an edge case that the library does not take into consideration (yet). For future research, we plan to re-review these cases and provide assistance, in the form of pull requests and issue creation, to the further development of the library.

The second point is related to the definition of these CPS-PAs. They are closely related to hardware interaction with the rest of the system. In our automatic detection approach, we describe that detecting a hardware-related call is necessary; see Figs. 5 and 6. As this in itself is a difficult challenge that requires insight into each project and the libraries used, a more generic detection method will need to be investigated in the future. For the AP-Spotter tool, a generic approach was decided, which requires a pre-selection of interesting modules to be made before running the AP-Spotter tool. This could result in both possible false negatives and false positives. As we do not have a catalog of antipatterns existing in the selected projects, we cannot verify how many false positives are occurring.

5.2.5. *Additional analysis*

To further assess the performance of the AP-Spotter tool, we conducted an additional analysis by running AP-Spotter on the projects and commits we manually analyzed in Study I (Section 3). Since AP-Spotter is designed to detect antipatterns for projects written in C++ and the top two most frequent antipatterns, we focus our analysis on the projects in Table 4 written in C++ and that contain Hard Coded Fine Tuning and Magical Waiting Number antipattern. As such, we ran our tool on PX4-Autopilot as it is the only project of Study I that matched the requirements needed.

The dataset of the first study contains antipatterns on the commit level. Therefore, we follow the following methodology:

18 https://github.com/ArduPilot/ardupilot/commit/dd392f8c0a74fa0ff603ae5283792cd335fcdfcb.

19 https://github.com/ArduPilot/ardupilot/commit/53c4b163ce61a8d58651cb07e54bcfa0bbbdae44.

20 https://github.com/microsoft/AirSim/blob/main/GazeboDrone/src/main.cpp.

21 https://github.com/ArduPilot/ardupilot/blob/09a0d8d0c0ff060f1ea9d85d6923bf70c1b15f8f/libraries/AP_HAL_SITL/SITL_Periph_State.cpp.

1. We gather all the commits in which the Magical Waiting Number and/or Hard Coded Fine Tuning antipattern have been manually identified.
2. We excluded commits not including/changing any C++ files.
3. We run AP-Spotter on the files that are in the remaining commits.

To measure the performance of AP-Spotter, we used the recall as we are interested in understanding how many of these manually validated performance antipatterns it correctly detects. For completeness, we compute recall for each antipattern separately as follows:

$$Recall = \frac{TP}{TP + FN} \quad (3)$$

where TP indicates antipattern-related commits that AP-Spotter correctly detects, while FN denotes antipattern-related commits that our tool does not identify.

Total tool recall. Of the 1869 commits that are analyzed in Study I, 403 commits contained at least one Hard Coded Fine Tuning or Magical Waiting Number antipattern occurrence. From this selection of commits (403 in total), 304 commits are related to C++ files, and, thus, considered for evaluating our tool. These 304 commits contained a total of 321 antipattern instances (both types); notice that a single commit can contain files affected by both antipatterns.

Running the AP-Spotter tool resulted in finding the expected antipattern in 316 commits (thus, they are true positives), with 5 false negatives. Following the recall calculation in Eq. (3), this results in a recall value of 98%.

Recall for each Antipattern. The recall for each antipattern is as follows: (1) AP-Spotter reported 222 TP and 2 FN for Hard Coded Fine Tuning, resulting in a recall of 99%; (2) AP-Spotter returned 99 TP and 3 FN for Magical Waiting Number, with a recall of 97%.

6. Threats to validity

In this section, we review the threats to the validity of our studies separated by category.

To ensure the replicability of our studies, we provide the data collected during analysis and the tools created at https://github.com/ciselab/CPS_repo_mining for our first study, and https://github.com/ciselab/CPS_SPA_Detection_Tool for our second study. We also include README.md files in these replication packages as a guide on how to replicate our studies.

6.1. Internal validity

To select possible interesting commits for the first study, we selected commits based on performance-related keywords. This is to find developer-admitted performance issues and analyze these commits in search for possible antipatterns. This method relies heavily on the chosen keywords. To mitigate possible author bias in selected keywords, we extended our set of selected commits by applying Topic Modeling to the non-selected commits and adding relevant resulting commits to our data set. Further, these keywords were validated by domain experts from the H2020 COSMOS project (COSMOS, 2021). There is still a risk of interesting commits being missed due to how the developers write their commit messages. In future research, we want to analyze the projects for Performance Antipatterns and review the keywords used in their commit messages. Further, we want to research the possibility of combining keywords for higher accuracy of gathering commits with performance issues.

For the second study, we manually validated each result of the Automatic Detection Tool AP-Spotter. This method of validating the results would only confirm true and false positives. Potential false negatives that we are not aware of can still occur. One potential reason could be that the pre-selection criteria are too restrictive. This would be overcome by having a benchmark project where all antipattern in a system are known. As far as we are aware, such a benchmark does not yet exist. In future research, we will create such a benchmark for a small system.

6.2. Conclusion validity

For the first study, we selected 14 projects. Extending the pool of projects could potentially change the number of occurrences of each antipattern and possibly confirm other potential antipatterns. We mitigated this by having a broad and diverse selection of projects, but for future research, it would be of interest to keep an eye on possible occurrences of the potential antipatterns as described in this paper.

For the second study, we analyzed 9 projects, which differ from those used in the first study. Only 3 projects showed warnings for Magical Waiting Number and 4 projects for Hard Coded Fine Tuning. Selecting a larger set of projects would give us a more accurate estimate of the tool's precision.

6.3. External validity

In the first study, we did not consider the age and current activity level of each project. There is a possibility that the developers of older projects were not aware of the existence of the SPA. Thereby, these antipatterns could be more prevalent in older projects. The developers of the newer projects, on the other hand, could have been aware of the existence of these SPAs at the design stage of the project. Thereby, these SPAs would not be occurring as often.

For the second study, the detection tool AP-Spotter analyzes the current state of the project and the history of the file when detecting potential antipatterns. As such, it could be that at some point in time antipatterns were present in the system, but were resolved before the version that we analyzed. These antipatterns are not part of our analysis.

7. Conclusion

Since the coinage of the term CPS in 2006 by Gill (2006), CPSs have increasingly become more part of our daily lives (Ashibani and Mahmoud, 2017; Okolie et al., 2018; DeFranco and Serpanos, 2021), from smart cars (Birchler et al., 2022) to medical devices (Chen, 2017). This paper researches the occurrences of CPS Performance Antipatterns in Open-Source projects on GitHub, further, it presents an approach to detect the two most frequently occurring CPS-PA. The goal of this paper is to aid developers in performance-demanding CPS projects and increasing awareness of existing CPS-PAs.

We conducted two studies in this paper; first, we analyzed multiple open-source CPS projects to catalog the frequency of known CPS-specific antipatterns and search for unknown ones. In our second study, we proposed an automatic detection approach for the two most frequently occurring antipatterns, and we evaluated our approach against a set of different open-source CPS projects.

As we have seen from our results, we detected the following antipatterns in the open-source projects that we considered: (i) Hard Coded Fine Tuning, (ii) Magical Waiting Number, (iii) Fixed Communication Rate, (iv) Rounding Errors, (v) Is Everything OK, (vi) Where Was I, and (vii) Are We There Yet **(RQ1)**. The most frequently occurring antipatterns are: (i) Hard Coded Fine Tuning (399 out of 646 occurrences), and (ii) Magical Waiting Number (150 out of 646 occurrences) **(RQ2)**. Further, our automatic detection approach showed a precision of 63.39% for Magical Waiting Number and 60.98% for the Hard Coded Fine Tuning antipattern. The main challenges for automatic detection of these antipatterns are (i) building the AST, and (ii) detecting a hardware connection in the modules **(RQ3)**.

We hope that the AP-Spotter tool can be a starting point for practitioners and researchers to be used, extended, and adjusted for the continuing effort of improving the overall performance and code quality of CPSs.

This paper makes the following contributions:

- PyRock: an open-Source mining repository tool to find commits that are related to self-admitted performance issues based on the commit message.
- A data set of 2699 potentially performance-related commits from 14 Open-Source projects.
- A catalog/taxonomy of new CPS-PAs identified through manual classification and analysis of self-admitted performance issues.
- An approach for automatically detecting the two most frequent occurring antipatterns, namely Magical Waiting Number and Hard Coded Fine Tuning.
- AP-Spotter: An implementation of our proposed CPS-PAs detection approach.

Replication packages for both studies are available openly on GitHub.[22],[23] These replication packages contain the data collected during the analysis and source code of the tools introduced in this paper.

For future work, we want to conduct surveys and interviews with the CPS developers to better understand why we see such a high number of occurrences for each type of antipattern. Further, we would like to see if an automatic detection tool for these antipatterns would be helpful to the developers.

CRediT authorship contribution statement

Imara van Dinten: Conceptualization, Methodology, Software, Data curation, Writing – original draft. **Pouria Derakhshanfar:** Methodology, Data curation, Writing – original draft, Writing – review & editing. **Annibale Panichella:** Conceptualization, Methodology, Data curation, Writing – original draft, Writing – review & editing, Supervision. **Andy Zaidman:** Conceptualization, Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Imara van Dinten reports financial support was provided by EU Framework Programme for Research and Innovation ICT Leadership in Enabling and Industrial Technologies. Pouria Derakhshanfar reports financial support was provided by EU Framework Programme for Research and Innovation ICT Leadership in Enabling and Industrial Technologies. Annibale Panichella reports financial support was provided by EU Framework Programme for Research and Innovation ICT Leadership in Enabling and Industrial Technologies. Andy Zaidman reports financial support was provided by EU Framework Programme for Research and Innovation ICT Leadership in Enabling and Industrial Technologies.

Data availability

Data is available in GitHub, as specified in the paper.

Acknowledgments

We would like to thank Silviu Mărîi and Lucas Holten for their support with the manual analysis.

This work has been partially supported by the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 957254, project COSMOS (2021).

[22] https://github.com/ciselab/CPS_repo_mining.
[23] https://github.com/ciselab/CPS_SPA_Detection_Tool.

References

Abdessalem, R.B., Panichella, A., Nejati, S., Briand, L.C., Stifter, T., 2020. Automated repair of feature interaction failures in automated driving systems. In: 29th International Symposium on Software Testing and Analysis (ISSTA). ACM, pp. 88–100.

Aleti, A., Trubiani, C., van Hoorn, A., Jamshidi, P., 2018. An efficient method for uncertainty propagation in robust software performance estimation. J. Syst. Softw. (JSS) 138, 222–235.

Ashibani, Y., Mahmoud, Q.H., 2017. Cyber physical systems security: Analysis, challenges and solutions. Comput. Secur. 68, 81–97.

Avritzer, A., Britto, R., Trubiani, C., Russo, B., Janes, A., Camilli, M., van Hoorn, A., Heinrich, R., Rapp, M., Henß, J., 2021. A multivariate characterization and detection of software performance antipatterns. In: International Conference on Performance Engineering (ICPE). pp. 61–72.

Birchler, C., Ganz, N., Khatiri, S., Gambi, A., Panichella, S., 2022. Cost-effective simulation-based test selection in self-driving cars software with SDC-Scissor. In: International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp. 164–168.

Birchler, C., Khatiri, S., Derakhshanfar, P., Panichella, S., Panichella, A., 2023. Single and multi-objective test cases prioritization for self-driving cars in virtual environments. ACM Trans. Softw. Eng. Methodol. 32 (2), 28:1–28:30.

Blei, D.M., Ng, A.Y., Jordan, M.I., 2003. Latent Dirichlet allocation. J. Mach. Learn. Res. (JMLR) 3, 993–1022.

Bondi, A.B., 2014. Foundations of Software and System Performance Engineering: Process, Performance Modeling, Requirements, Testing, Scalability, and Practice. Addison-Wesley Professional.

Brown, W.J., Malveau, R.C., Mccormick, H.W., Mowbray, T.J., 1998. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, first ed. John Wiley and Sons Inc..

Calinescu, R., Češka, M., Gerasimou, S., Kwiatkowska, M., Paoletti, N., 2017. Designing robust software systems through parametric Markov chain synthesis. In: International Conference on Software Architecture (ICSA). IEEE, pp. 131–140.

Chen, H., 2017. Applications of cyber-physical system: A literature review. J. Ind. Integr. Manag. (JIIM) 02, 1750012.

Cito, J., Leitner, P., Bosshard, C., Knecht, M., Mazlami, G., Gall, H.C., 2018. PerformanceHat: augmenting source code with runtime performance traces in the IDE. In: Proceedings of the 40th International Conference on Software Engineering (ICSE). pp. 41–44.

Coppelia Robotics, 2010. CoppeliaSim. https://coppeliarobotics.com/.

Cortellessa, V., Di Marco, A., Trubiani, C., 2014. An approach for modeling and detecting software performance antipatterns based on first-order logics. Softw. Syst. Model. (SoSyM) 13, 391–432.

2021. COSMOS: DevOps for complex cyber-physical systems. https://www.cosmos-devops.org/.

De Lucia, A., Di Penta, M., Oliveto, R., Panichella, A., Panichella, S., 2013. Applying a smoothing filter to improve IR-based traceability recovery processes: An empirical investigation. Inf. Softw. Technol. (IST) 55 (4), 741–754.

De Lucia, A., Di Penta, M., Oliveto, R., Panichella, A., Panichella, S., 2014. Labeling source code with information retrieval methods: an empirical study. Empir. Softw. Eng. (EMSE) 19, 1383–1420.

DeFranco, J.F., Serpanos, D., 2021. The 12 flavors of cyberphysical systems. Comput. Soc. 54, 104–108.

Dugan, R.F., Glinert, E.P., Shokoufandeh, A., 2002. The sisyphus database retrieval software performance antipattern. In: 3rd International Workshop on Software and Performance (WOSP). ACM, pp. 10–16.

Ernst, M.D., 2003. Static and dynamic analysis: Synergy and duality. In: Workshop on Dynamic Analysis (WODA). pp. 24–27.

Farley, A., Wang, J., Marshall, J.A., 2022. How to pick a mobile robot simulator: A quantitative comparison of CoppeliaSim, Gazebo, MORSE and Webots with a focus on accuracy of motion. Simul. Model. Pract. Theory 120, 102629.

Frechtling, M., Leong, P.H.W., 2015. MCALIB: Measuring sensitivity to rounding error with Monte Carlo programming. Trans. Program. Lang. Syst. (TOPLAS) 37.

Gambi, A., Müller, M., Fraser, G., 2019. Asfault: Testing self-driving car software using search-based procedural content generation. In: 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). IEEE, pp. 27–30.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1994. Design Patterns: Elements of Reusable Object-Oriented Software, thirty-seventh ed. Addison-Wesley Professional.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 2009. Design Patterns: Elements of Reusable Object-Oriented Software, thirty-seventh ed. Addison-Wesley Professional.

Gill, H., 2006. National workshop on cyber-physical systems: NSF perspective and status on cyber-physical systems. http://varma.ece.cmu.edu/CPS/Presentations/gill.pdf, accessed: 23 Jun 2010 22:32:09 GMT.

Greer, C., Burns, M., Wollman, D., Griffor, E., 2019. Cyber-Physical Systems and Internet of Things. Tech. rep., National Institute of Standards and Technology.

Hassan, F., Mostafa, S., Lam, E.S., Wang, X., 2017. Automatic building of java projects in software repositories: A study on feasibility and challenges. In: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, pp. 38–47.

Jain, P., Aggarwal, P.K., Chaudhary, P., Makar, K., Mehta, J., Garg, R., 2021. Convergence of IoT and CPS in robotics. In: Emergence of Cyber Physical System and IoT in Smart Automation and Robotics: Computer Engineering in Automation. Springer International Publishing, pp. 15–30, Ch. 2.

Kaltenecker, C., Grebhahn, A., Siegmund, N., Apel, S., 2020. The interplay of sampling and machine learning for software performance prediction. Software 37 (4), 58–66.

Kelley, A., Pohl, I., 2006. A Book on C: Programming in C, fourth ed. Addison-Wesley.

Kharkar, A., Moghaddam, R.Z., Jin, M., Liu, X., Shi, X., Clement, C., Sundaresan, N., 2022. Learning to reduce false positives in analytic bug detectors. In: Proceedings of the 44th International Conference on Software Engineering. pp. 1307–1316.

Khatami, A., Zaidman, A., 2023. State-of-the-practice in quality assurance in java-based open source software development. abs/2306.09665. arXiv:2306.09665, https://doi.org/10.48550/arXiv.2306.09665.

Khatiri, S., Panichella, S., Tonella, P., 2023. Simulation-based test case generation for unmanned aerial vehicles in the neighborhood of real flights. In: 16th International Conference on Software Testing, Verification and Validation (ICST). IEEE, pp. 281–292.

Lee, E.A., 2015. The past, present and future of cyber-physical systems: A focus on models. Sensors 15, 4837—4869.

Lee, J., Bagheri, B., Kao, H.-A., 2015. A cyber-physical systems architecture for industry 4.0-based manufacturing systems. Manuf. Lett. 3, 18–23.

Lee, E.A., Seshia, S.A., 2017. Introduction to Embedded Systems: A Cyber-Physical Systems Approach, second ed. The MIT Press, p. 537.

Li, R., Liu, H., Lou, G., Zheng, X., Liu, X., Chen, T.Y., 2021. Metamorphic testing on multi-module uav systems. In: 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 1171–1173.

Maiga, A., Ali, N., Bhattacharya, N., Sabané, A., Guéhéneuc, Y.-G., Antoniol, G., Aïmeur, E., 2012. Support vector machines for anti-pattern detection. In: 27th International Conference on Automated Software Engineering (ASE). ACM, pp. 278–281.

Min, J., Park, J.-H., Li, H., 2002. Performance evaluation of object detection algorithms. In: 2002 International Conference on Pattern Recognition, Vol. 3. IEEE, pp. 965–969.

Mittal, S., Tolk, A., 2019. Complexity Challenges in Cyber Physical Systems. Wiley.

Moesus, N., Scholze, M., Schlesinger, S., Herber, P., 2019. A rating tool for the automated selection of software refactorings that remove antipatterns to improve performance and stability. In: Software Technologies (ICSOFT). Springer, pp. 28–54.

Moha, N., Guéhéneuc, Y.G., Duchien, L., Le Meur, A.F., 2010. DECOR: A method for the specification and detection of code and design smells. Trans. Softw. Eng. (TSE) 36 (1), 20–36.

Mühlbauer, S., Apel, S., Siegmund, N., 2019. Accurate modeling of performance histories for evolving software systems. In: 34th International Conference on Automated Software Engineering (ASE). IEEE/ACM, pp. 640–652.

Nistor, A., Song, L., Marinov, D., Lu, S., 2013. Toddler: Detecting performance problems via similar memory-access patterns. In: 35th International Conference on Software Engineering (ICSE). pp. 562–571.

Okolie, S., Kuyoro, S., Ohwo, O.B., 2018. Emerging cyber-physical systems : An overview. Int. J. Sci. Res. Comput. Sci. Eng. Inf. Technol. (IJSRCSEIT) 306–316.

Panichella, A., 2019. A systematic comparison of search algorithms for topic modelling—a study on duplicate bug report identification. In: 11th International Symposium on Search Based Software Engineering (SSBSE). Springer, pp. 11–26.

Panichella, S., Di Sorbo, A., Guzman, E., Visaggio, C.A., Canfora, G., Gall, H.C., 2015. How can I improve my app? Classifying user reviews for software maintenance and evolution. In: International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 281–290.

Panichella, A., Dit, B., Oliveto, R., Di Penta, M., Poshynanyk, D., De Lucia, A., 2013. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In: 35th International Conference on Software Engineering (ICSE). IEEE, pp. 522–531.

Panichella, A., Dit, B., Oliveto, R., Di Penta, M., Poshyvanyk, D., De Lucia, A., 2016. Parameterizing and assembling IR-based solutions for SE tasks using genetic algorithms. In: 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). IEEE, pp. 314–325.

Parr, T., 2013. The Definitive ANTLR 4 Reference, second ed. Pragmatic Bookshelf.

Pinciroli, R., Smith, C.U., Trubiani, C., 2021. QN-based modeling and analysis of software performance antipatterns for cyber-physical systems. In: International Conference on Performance Engineering (ICPE). ACM, pp. 93–104.

Pinciroli, R., Trubiani, C., 2021. Model-based performance analysis for architecting cyber-physical dynamic spaces. In: 2021 IEEE 18th International Conference on Software Architecture (ICSA). pp. 104–114.

2023. PMD: An extensible cross-language static code analyzer. https://pmd.github.io/, accessed: 20 Mar 2023.

Porter, M.F., 1980. An algorithm for suffix stripping. Program 14, 130–137.

Rowley, J., Hartley, R., 2017. Organizing Knowledge: An Introduction to Managing Access to Information. Routledge.

Schügerl, P., Walsh, D., Rilling, J., Charland, P., 2009. A contextual guidance approach to software security. In: 33rd Annual International Computer Software and Applications Conference (COMPSAC), Vol. 2. pp. 194–199.

Shackel, B., 2009. Usability – Context, framework, definition, design and evaluation. Interact. Comput. 21, 339–346.

Shah, S., Dey, D., Lovett, C., Kapoor, A., 2018. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In: Field and Service Robotics: Results of the 11th International Conference. Springer, pp. 621–635.

Shi, J., Wan, J., Yan, H., Suo, H., 2011. A survey of cyber-physical systems. In: International Conference on Wireless Communications and Signal Processing (WCSP). pp. 1—6.

Smith, C.U., 2020. Software performance antipatterns in cyber-physical systems. In: International Conference on Performance Engineering (ICPE). ACM, pp. 173–180.

Smith, C.U., Williams, L.G., 2000. Software performance antipatterns. In: 2nd International Workshop on Software and Performance (WOSP). pp. 127–136.

Smith, C.U., Williams, L.G., 2001. Software performance antipatterns; common performance problems and their solutions. In: 27th International Conference Computer Measurement Group Conference (CMG). pp. 797–806.

Smith, C.U., Williams, L.G., 2002a. Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software, Vol. 23. Addison-Wesley, Reading.

Smith, C.U., Williams, L.G., 2002b. New software performance antipatterns: More ways to shoot yourself in the foot. In: 28th International Conference Computer Measurement Group (CMG). pp. 667–674.

Smith, C.U., Williams, L.G., 2003. More new software performance antipatterns: Even more ways to shoot yourself in the foot. In: 29th International Conference Computer Measurement Group Conference (CMG). pp. 717–725.

Spadini, D., Aniche, M., Bacchelli, A., 2018. PyDriller: Python framework for mining software repositories. In: 26th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE). pp. 908–911.

Triola, M.F., Goodman, W.M., Law, R., Labute, G., 2006. Elementary Statistics. Pearson/Addison-Wesley, Reading, MA.

Trubiani, C., Bran, A., van Hoorn, A., Avritzer, A., Knoche, H., 2018. Exploiting load testing and profiling for performance antipattern detection. Inf. Softw. Technol. (IST) 95, 329–345.

Trubiani, C., Di Marco, A., Cortellessa, V., Mani, N., Petriu, D., 2014. Exploring synergies between bottleneck analysis and performance antipatterns. In: 5th International Conference on Performance Engineering (ICPE). ACM, pp. 75–86.

van Dinten, I., Derakhshanfar, P., Panichella, A., Zaidman, A., 2023. Appendix: The slow and the furious? Performance antipattern detection in cyber-physical systems. https://github.com/ciselab/CPS_repo_mining/tree/main/documents/appendix_tables, Provides additional information to this paper.

Velez, M., Jamshidi, P., Siegmund, N., Apel, S., Kästner, C., 2021. White-box analysis over machine learning: Modeling performance of configurable systems. In: Proceedings of the 43rd International Conference on Software Engineering (ICSE). IEEE Press, pp. 1072–1084.

Veliolu, S., Selçuk, Y.E., 2017. An automated code smell and anti-pattern detection approach. In: 15th International Conference on Software Engineering Research, Management and Applications (SERA). IEEE, pp. 271–275.

Wei, X.L., Huang, X.L., Lu, T., Song, G.G., 2019. An improved method based on deep reinforcement learning for target searching. In: 4th International Conference on Robotics and Automation Engineering (ICRAE). IEEE, pp. 130–134.

Wert, A., Happe, J., Happe, L., 2013. Supporting swift reaction: Automatically uncovering performance problems by systematic experiments. In: 35th International Conference on Software Engineering (ICSE). IEEE, pp. 552–561.

Wilkinson, J.H., 1994. Rounding Errors in Algebraic Processes. Courier Corporation.

Woodside, M., Petriu, D., Petriu, D., Shen, H., Israr, T., Merseguer, J., 2005. Performance by unified model analysis (PUMA). In: Proceedings of the Fifth International Workshop on Software and Performance (WOSP). pp. 1–12.

Wu, G., Sun, J., Chen, J., 2016. A survey on the security of cyber-physical systems. Control Theory Technol. 14.

Zaidman, A., Rompaey, B.V., Demeyer, S., van Deursen, A., 2008. Mining software repositories to study co-evolution of production & test code. In: First International Conference on Software Testing, Verification, and Validation (ICST). IEEE, pp. 220–229.

Zaidman, A., Van Rompaey, B., van Deursen, A., Demeyer, S., 2011. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. Empir. Softw. Eng. 16 (3), 325–364.

Zampetti, F., Kapur, R., Di Penta, M., Panichella, S., 2022. An empirical characterization of software bugs in open-source cyber–physical systems. J. Syst. Softw. (JSS) 192, 111425.

Imara van Dinten is a Ph.D. candidate in the Software Engineering Research Group at the Delft University of Technology. She received her MS in Robot Systems at the University of Southern Denmark in 2016, after which she worked as a Software Test Engineer. She currently researches topics on Software Quality for Complex Cyber–Physical Systems.

Pouria Derakhshanfar is a senior researcher in the Intelligent Collaboration Tools Lab (ICTL) of the JetBrains Research. He obtained his Ph.D. in Computer Science from the Delft University of Technology, with his dissertation entitled "Carving Information Sources to Drive Search-Based Crash Reproduction and Test Case Generation". He is currently working on automated test generation in IDEs and validation of automatically generated code.

Annibale Panichella is an assistant professor with the Software Engineering Research Group (SERG), Delft University of Technology (TU Delft), Netherlands. He is also a research fellow with the Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg. His research interests include security testing, evolutionary testing, search-based software engineering, textual analysis, and empirical software engineering. He serves and has served as program committee member of various international conference (e.g., ICSE, GECCO, ICST and ICPC) and as reviewer for various international journals (e.g., the IEEE Transactions on Software Engineering, the ACM Transactions on Software Engineering and Methodology, the IEEE Transactions on Evolutionary Computation, the Empirical Software Engineering, the Software Testing, Verification & Reliability) in the fields of software engineering and evolutionary computation.

Andy Zaidman is a full professor in software engineering at Delft University of Technology, The Netherlands. He received the M.Sc. and Ph.D. degrees in Computer Science from the University of Antwerp, Belgium, in 2002 and 2006, respectively. His main research interests include software evolution, program comprehension, mining software repositories, software quality, and software testing. He is an active member of the research community and involved in the organization of numerous conferences (WCRE'08, WCRE'09, VISSOFT'14 and MSR'18). In 2013 he was the laureate of a prestigious Vidi mid-career grant, while in 2019 he received the most prestigious Vici career grant from the Dutch science foundation NWO.