COSMOS
DevOps for Complex Cyber-physical Systems

Project Number 957254

D5.1 Framework of metrics for production code anti-patterns for DevOps

Version 1.0
30 September 2021
Final

Public Distribution

Delft University of Technology

Project Partners: **Aicas, Delft University of Technology, GMV Skysoft, Intelligentia, Q-media, Siemens, Siemens Healthcare, The Open Group, University of Luxembourg, University of Sannio, Unparallel Innovation, Zurich University of Applied Sciences**

PROJECT PARTNER CONTACT INFORMATION

Aicas James Hunt Emmy-Noether-Strasse 9 76131 Karlsruhe Germany Tel: +49 721 663 968 0 E-mail: jjh@aicas.com	**Delft University of Technology** Annibale Panichella Van Mourik Broekmanweg 6 2628 XE Delft Netherlands Tel: +31 15 27 89306 E-mail: a.panichella@tudelft.nl
Intelligentia Davide De Pasquale Via Del Pomerio 7 82100 Benevento Italy Tel: +39 0824 1774728 E-mail: davide.depasquale@intelligentia.it	**GMV Skysoft** José Neves Alameda dos Oceanos Nº 115 1990-392 Lisbon Portugal Tel. +351 21 382 93 66 E-mail: jose.neves@gmv.com
Q-media Petr Novobilsky Pocernicka 272/96 108 00 Prague Czech Republic Tel: +420 296 411 480 E-mail: pno@qma.cz	**Siemens** Birthe Boehm Guenther-Scharowsky-Strasse 1 91058 Erlangen Germany Tel: +49 9131 70 E-mail: birthe.boehm@siemens.com
Siemens Healthineers David Malgiaritta Siemensstrasse 3 91301 Forchheim Germany Tel: +49 9191 180 E-mail: david.malgiaritta@siemens-healthineers.com	**The Open Group** Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of Sannio Massimiliano Di Penta Palazzo ex Poste, Via Traiano I-82100 Benevento Italy Tel: +39 0824 305536 E-mail: dipenta@unisannio.it	**University of Luxembourg** Domenico Bianculli 29 Avenue J. F. Kennedy L-1855 Luxembourg Luxembourg Tel: +352 46 66 44 5328 E-mail: domenico.bianculli@uni.lu
Unparallel Innovation Bruno Almeida Rua das Lendas Algarvias, Lote 123 8500-794 Portimão Portugal Tel: +351 282 485052 E-mail: bruno.almeida@unparallel.pt	**Zurich University of Applied Sciences** Sebastiano Panichella Gertrudstrasse 15 8401 Winterthur Switzerland Tel: +41 58 934 41 56 E-mail: panc@zhaw.ch

DOCUMENT CONTROL

Version	Status	Date
0.1	First draft	12 August 2021
0.3	Update Sections structure and add manual analysis final results	26 August 2021
0.5	Further revisions with the manual analysis review	15 September 2021
0.9	Internal review version	28 September 2021
1.0	Final version of EC delivery	30 September 2021

TABLE OF CONTENTS

TABLE OF FIGURES

EXECUTIVE SUMMARY

This deliverable presents an extensive analysis of bad coding practices in CPSs that can lead to performance issues in the system. We performed this analysis by examining the code history of a diverse set of 12 CPS projects openly available on GitHub. First, this report describes the methodology that we used for performing automated and manual analyses. Next, it presents the results, including (i) the performance issues detected in our analysis (with some representative examples), (ii) how frequent these performance issues occurred in our analysis, and (iii) a discussion regarding whether these performance issues can be considered as CPS-related performance antipatterns. Besides this deliverable, we provide a tool called PyRock, which eases the process of exploring the code history of CPSs for finding code changes that might lead to performance issues. Finally, we present a replication package of our study.

1. INTRODUCTION

1.1 WORK PACKAGE OVERVIEW

In recent years, Cyber-physical Systems (CPSs) have been of interest in many contexts. The applications of CPSs have been emerging to address various challenges in different scenarios (e.g., medical devices, transportation) [1]. On the one hand, the growth in applications of CPSs increases their impact on our everyday lives, making their performance more important. On the other hand, the expansion of CPS development and implementation tasks leads to higher demand for experts in new CPS-related technologies, which is not an easy demand to fulfil, and thereby increases the risk of performance failures [2]. Hence, it is crucial to perform different techniques to assess the performance of the CPSs and ensure that this system is less likely to encounter performance issues.

One of the standard solutions for achieving high software performance is to use a portfolio of Software Performance Antipatterns (SPA) [3, 4], which are documenting the common performance problems in the software architecture and design of the systems, to ease the detection of bad design/coding choices that influence performance. A previous study [5] confirmed that SPAs are beneficial, while providing reusable solutions applicable in various domains. Moreover, identifying the SPAs helps design and inform refactoring actions, which ensure that the performance antipatterns can be removed from the project's architecture or designs, and thereby, the project is less prone to performance issues [6, 7].

One of the domains that SPAs can be helpful in, is the domain of CPSs. This work package focuses on the SPAs in CPSs and, subsequently, potential refactoring operations that can be applied to remove these anti-patterns.

1.2 TASK OVERVIEW

As the first step towards designing a refactoring framework for CPSs, it is required to gather information about the performance-related issues and antipatterns in CPSs. In a recent study, Smith [2] carried out a preliminary investigation into the performance antipatterns for this type of system. This deliverable identified three new SPAs specific to CPSs. It also recognized six other SPAs previously defined in other types of systems. Although the antipatterns introduced in Smith's study facilitate the recognition and refactoring of CPS performance-related issues, this article does not provide any empirical evidence regarding how common these antipatterns are. Also, it is not evident in how many CPS projects these identified antipatterns were observed.

In this study, we conduct an extensive analysis on a set of open-source CPS projects to identify the performance issues found and fixed by the original developers within the code history of these projects. This analysis aims to examine:

- How often do the SPAs identified by Smith [2] occur in open-source CPS projects?
- Are there any other new performance-related antipatterns detectable from the open-source CPSs, and how often they identified and fixed?

- What are the general SPAs, which commonly exist in other types of systems, that are also commonly detected in CPSs?

To perform this analysis, we mine repositories of the detected CPS projects to identify candidate commits that might be related to performance issues. We identify these commits via keyword matching in the commit messages. Next, we manually analyze the source code changed by each of these candidate commits to find truly performance-related issues.

1.3 PURPOSE OF THIS DELIVERABLE

This deliverable is structured in three main sections:

- The automated procedure to find candidate commits that could expose performance-related issues from the CPSs code history (Section 3).
- The methodology that we used to perform the manual analysis on the detected commits to identify the performance-related issues (Section 0).
- An in-depth discussion about our findings from the analysis about the commons SPAs that occurred in the analyzed CPS projects (Section 5).

This deliverable provides the catalog of performance-related antipatterns, which are required to design and introduce refactoring actions in Deliverable D5.2.

The remainder of this deliverable is organized as follows: Section 2 reports the background and related works on SPAs for various systems. Section 3 describes our methodology for finding the open-source CPS projects and the automated procedure we designed to collect the possibly interesting commits from the code history. Section 4 explains the manual analysis that we performed for detecting performance-related issues. Section 5 presents the results of our analysis (i.e., the SPAs that we identified in our analysis and the ratios of their occurrences). Section 6 explains the replication package that we prepared for this deliverable. Our Future works are discussed in Section 7. Finally, Section 8 concludes this report.

2. BACKGROUND

2.1 SOFTWARE PERFORMANCE ANTIPATTERNS

Design patterns are good coding practices to follow when trying to implement a certain type of structure [8, 9]. Over the years, more design patterns emerged, showing good practices for implementation and the reasoning behind it. Antipatterns followed this trend, where patterns were found in implementation that should not be done; for reasons such as security, performance or maintainability [10].

Among these Antipatterns, Software performance antipatterns (SPAs) mainly focus on the common patterns in the software architecture and design, which lead to a performance issue in the system [4]. Various prior studies contributed to this field of research by introducing multiple SPAs and solutions to tackle them [11, 12, 13, 14]. However, these studies did not perform any empirical study to report the occurrence of such antipatterns.

In this deliverable, we perform an empirical study on a set of CPS projects.

2.2 SPAs IN CYBER-PHYSICAL SYSTEMS

CPS have been emerging over the years, where Industry 2.0 introduced mass production, it took until Industry 4.0 (first coined in 2013) for CPS to be put on the map [15] [16].

Since CPS has been increasingly part of our industry, the need for CPS specific research arose [1, 17, 18, 19]. This would also include research about CPS specific performance antipatterns (CPS-PA). Since in many CPS projects, the resources (e.g., battery and computational resources) are limited, it is crucial to minimize the CPSs performance issues. By looking at the prior studies, we found only one paper related to CPS-PAs [2]. In this paper, Smith identifies three new CPS-PAs, which were unidentified previously, and six common software performance antipatterns that can also be found in CPSs.

2.2.1 CPS-PAs identified by Smith [2]

Are We There Yet?: This antipattern refers to 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 events occurrences. This performance antipattern leads to overusing resources in the system.

Is Everything OK?: This performance antipattern is similar to the previous one: it refers to constantly checking the status of the system (e.g., storage space, battery usage). Same as Are WE There Yet? antipattern, this performance issue happens when the status checker threads and processes are triggered too often.

Where Was I?: This antipattern refers to processes in CPSs that lost the information about the system's state after a certain event, such as system restart. It also can happen if CPS gives too much time (i.e., more than 1 minute) to processes that can keep the users waiting. This type of antipatterns leads to execution overheads to perform required calculations to drive the CPS back to the desired status.

2.2.2 Common performance antipatterns also detected in CPSs

Unnecessary Processing: This antipattern reflects the scenarios in which heavy and unnecessary processes are executed in critical scenarios [11]. 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?: This antipattern refers to invoking a method many times in scenarios in which CPS could call the method only once and store and reuse the returned outputs for the following processes [12]. To tackle this antipattern, redundant calls should be detected and removed.

More is Less: This antipattern happens when CPS has access to too many resources that negatively impact the system's overall performance [2]. Adding too many resources (such as threads and processes) may lead to extra overheads for tasks like scheduling, context switching, etc.

The Ramp: In this antipattern, the performance and efficiency of the CPS are exponentially reduced as the processing time linearly increases [14]. This type of

performance issue can occur in CPSs for various reasons, such as changes in the environment or processing a large amount of historical information [2].

Museum Checkroom: This antipattern occurs in scenarios where CPS uses a simple FCFS queue to manage resource allocation to processes [20]. This can lead to performance issues in cases that this resource management system needs to handle too many processes. To tackle this performance antipattern, CPS developers need to implement priority queuing to prioritize the processes that will release the resources in a short time.

Falling Dominoes: This performance antipattern happens in cases that of failure of a module leads to more failures in other modules [2]. Since CPSs include many small interacting hardware pieces with various software modules, this common performance antipattern can also occur in CPSs. To tackle this antipattern, CPS developers need to ensure that modules are as isolated as possible.

The CPS-related performance antipatterns identified by Smith can help the subsequent studies introduce automated approaches for identifying performance issues in CPSs. However, this study did not provide any empirical evidence about the identified antipatterns. Hence, in this deliverable, we perform an independent empirical study to find the CPS-PAs identified by Smith and new CPS-PAs.

2.3 MINING SOFTWARE REPOSITORY

A common methodology to gather empirical evidence is called Mining [21] (MSR), in which researchers analyze the data available in software repositories (e.g., commits, commit messages, author, date of changes) [22, 21]. In this deliverable, we utilize this technique to collect the code changes. Then, we manually analyze the collected information for detecting the code changes that expose performance issues in CPSs.

2.3.1 PyDriller

PyDriller [23] is an open-source Python Framework to help developers mine Git repositories. PyDriller has been used in various MSR-related prior studies. For instance, V. Lenarduzzi et al. [24] use this tool for building a technical debt dataset. Also, Kazerouni et al. [25] and Thongtanunam [26] utilized it for Software Quality and Testing. In this deliverable, we implemented a tool called PyRock that utilizes PyDriller for collecting the commits from the given software repositories,

3. IDENTIFYING POTENTIAL PERFORMANCE ANTIPATTERNS IN CODE HISTORY

3.1 PROJECT SELECTION

To collect the subjects for this deliverable's analysis, we collected a list of 12 CPS repositories publicly available on GitHub. These projects were collected in a collaborative effort between three research partners in COSMOS: Delft University of Technology, Zurich University, and University of Sannio. As presented in Table 1, the projects used in this study are selected from four different programming languages: Java, Python, C++, and JavaScript. Moreover, this benchmark is composed of projects with various levels of maturity: 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 CPSs, such as software for controlling drones, vacuum cleaners, small robot kits, etc. Finally, Table 1 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.

We perform an automated (Section 3.2) and manual analysis (Section 0) on the code history of these selected projects to identify performance antipatterns in CPSs.

Table 1: List of open-sour CPS projects used in this deliverable

Project Name	Programming language	# of commits	# of stars	# of forks	Description
Android App Manager	Java	231	10	12	A library for using ROS (Robot OS) in Android.
Cylon	JavaScript	1,323	3.8K	367	A framework for robotics.
Dronekit Android	Java	5,810	211	217	A framework for creating Android apps, controlling drones.
Johnny Five	JavaScript	3,355	12.4K	1.8K	A JavaScript robotics programming framework.
Node AR Drone	JavaScript	281	1.7K	446	A client for controlling Parrot AR Drone 2.0 quad-copters.
PX4-Autopilot	C++	35,537	4.8K	11.3K	A tool for controlling vehicles.
Robonomics-JS	JavaScript	68	13	8	A library to work with data from Robonomics (An open-source platform for IoT applications) network.
Robonomics-Contracts	JavaScript	502	78	31	Robonomics network infrastructure based on Ethereum Blockchain.
Vacuum Robot Mark II	Java/C++	54	28	3	Code for interacting with Vacuum Robot.
TurtleBot	C++	1,142	236	280	A framework for programming for a robot called TurtleBot.

TurtleBot 3	Python	526	770	637	A framework for programming for a robot called TurtleBot3.
Valetudo	JavaScript	1,043	2.5K	258	A cloud-free system for controlling vacuum cleaner robots.

3.2 COMMITS SELECTION

To scope down the number of commits to manually analyze, we implemented a semi-automated tool called PyRock. The tool identifies and selects changes in the codebase that are potentially exposing (or fixing) a performance issue in CPS projects. The outputs of this tool (i.e., the identified commits and changes in the code) are later manually analyzed by the authors (Section 0).

3.2.1 PyRock

For commit selection, we implement a tool called PyRock to parse, analyze, and filter the commit messages of each project. Figure 1 visualizes the tool's architectural design.

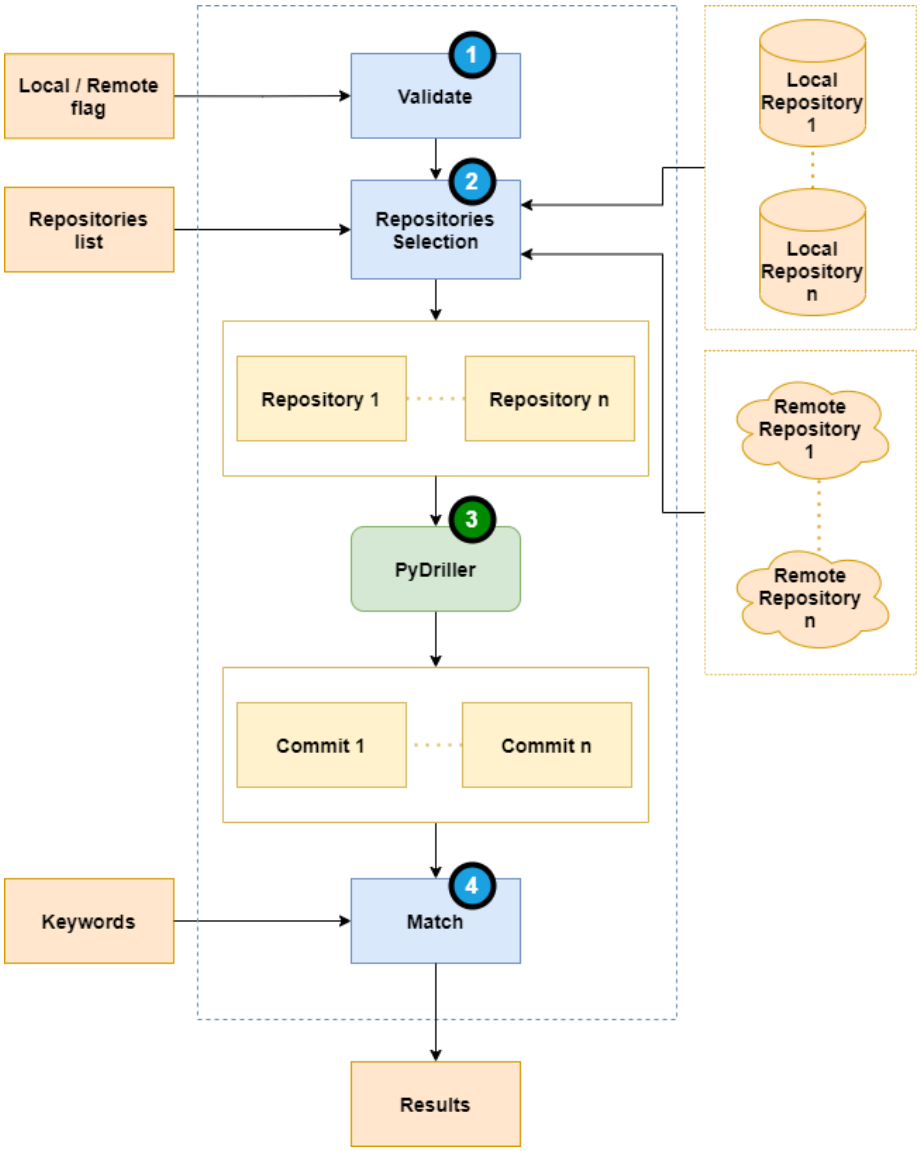

Figure 1: PyRock

1- PyRock requires two input parameters:

- *Repositories list:* list of repositories on which we want to perform the automated code history analysis.

- *Local/Remote flag:* this flag indicated whether the repositories are available locally (local mode) or PyRock needs to fetch them remotely (remote mode). In the former case, the user also needs to provide the directory in which the local repositories are located.

2- These user inputs are first validated by PyRock's *validate module* (see ① in Figure 1). This module checks that the user has indicated which in mode (local/remote) and with which repositories to run. 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.

3- After verification, PyRock selects each repository with the *Repositories Selection module*, see ② in Figure 1 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 repository list. In remote mode, it checks whether the repositories remote addresses are reachable.

4- For repository mining (③ in Figure 1), PyRock utilizes PyDriller [23] a commonly used open-source Python framework for mining Git repositories. PyRock passes the information regarding each repository one at a time to PyDriller. Then, PyDriller returns the list of all candidate commits in the code history of the project.

5- In the next step, the commit messages returned by PyDriller are passed through the *Match module*, see ④ in Figure 1. The matching method utilizes a keyword file, containing a list of performance-related keywords that could indicate a potential performance antipattern. 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 candidates commits as result. These results are then used to perform the manual analysis, see Chapter 4.1.

Performance-related keywords: The performance-related keywords used in PyRock can be classified into three categories:

1- *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.

2- *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.

3- *memory, memory-heap, memory-leak, memory leak, bottleneck, overhead, deadlock, livelock, infinite, impasse, hang, stuck, speed*: These keywords are

chosen based on previous experience, books regarding performance and found during the analysis phase.

! To keep the script fast to run, a commit is added to the candidate list as long as it matches one of the keywords, without checking whether it matches other keywords. On the authors' local machine (i7-1185G7, 32GB RAM, NVMe Micron 2300 1 TB), running the analysis for the selected projects (Table 1, totaling 49,872 commits) took about three minutes.

3.2.2 Results

Figure 2 shows the occurrence of each keyword in our study. The green bars indicate the number of times a keyword is detected before any other one (if it exists) in a commit message. The blue bars are the total number of times this keyword occurred across all the commit messages. For example, the keyword "overhead" occurred 2 times in a commit message where PyRock returned the commit due to this keyword. The keyword "overhead" occurred a total of 3 times in all the commit's messages.

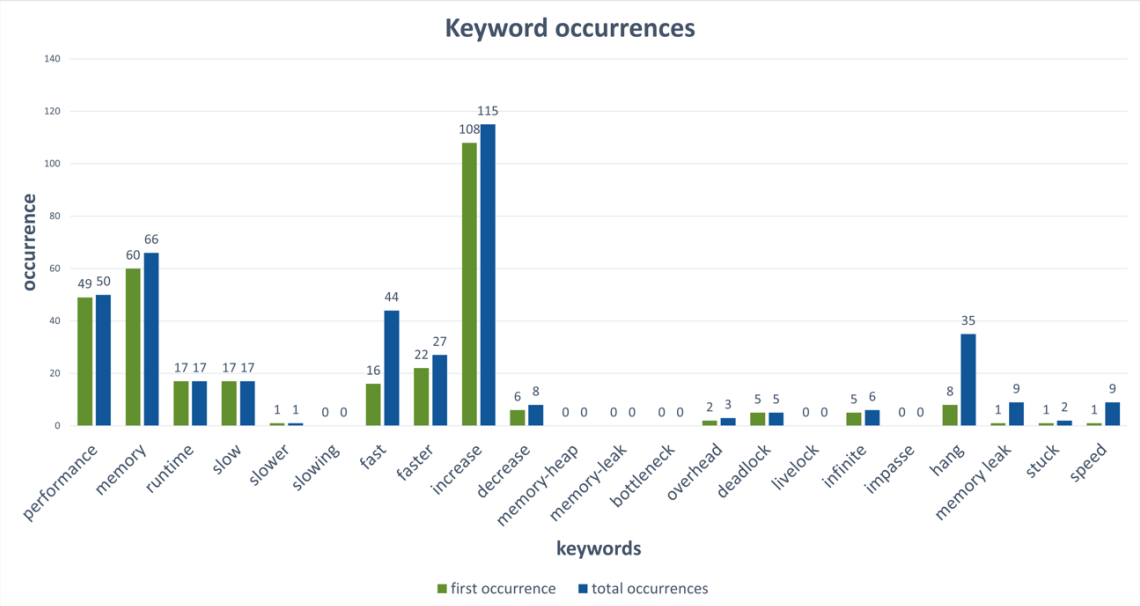

Figure 2: Keyword occurrences

Figure 2 also shows which keywords are often used in the selected projects. As expected, keywords such as "increase" occur relatively often. Though, due to the wide range of applications and contexts of "increase, this keyword has a low chance to actually indicate a performance antipattern in the commit.

Figure 3 shows how often a commit found with a certain keyword resulted in finding an antipattern in that commit. As shown in Figure 3, the keyword "increase" occurred often and around 65% of the times there was an antipattern occurrence. If we compare this to the keyword "memory", in around 26% of the cases an antipattern was found.

For this analysis, we prefer recall (i.e., also selecting the common keywords) over precision (i.e., using only highly specific keywords, such as "deadlock"). Using these broadly applied words mitigates the potential threats to the validity of missing potential antipatterns.

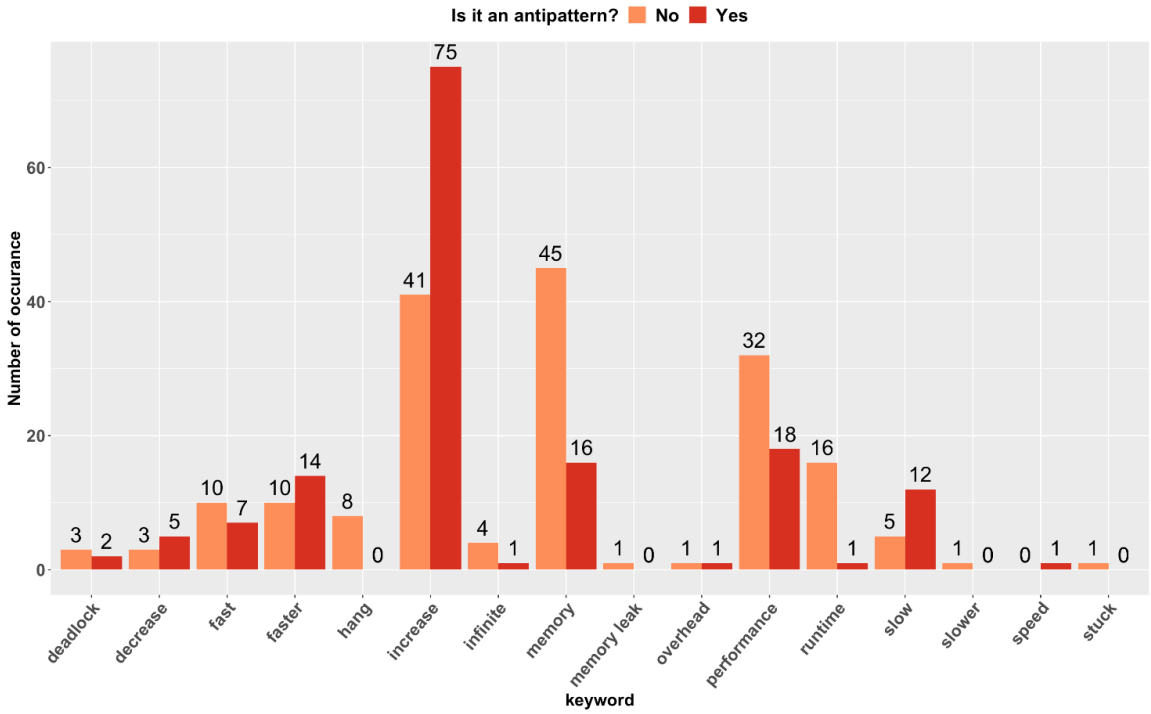

Figure 3: Keyword - Antipattern

There are also some keywords that did not occur at all. Keywords such as memory-leak and memory-heap did not happen because of the writing style as these keywords did occur, but the occurrences were catalogued as “memory”. In contrast, other keywords such as “impasse” did not appear in these projects.

Figure 4 shows the number of resulting commits for each analyzed project. The PX4-Autopilot project has 1101 commits that contained at least one of the keywords in their commit message. If we compare to other projects, we observe results ranging from 0 to 39, but we should also realize that the PX4-Autopilot project has a total of 35,537 commits, which is around 27 times more commits than the average of all other projects (see Table 1).

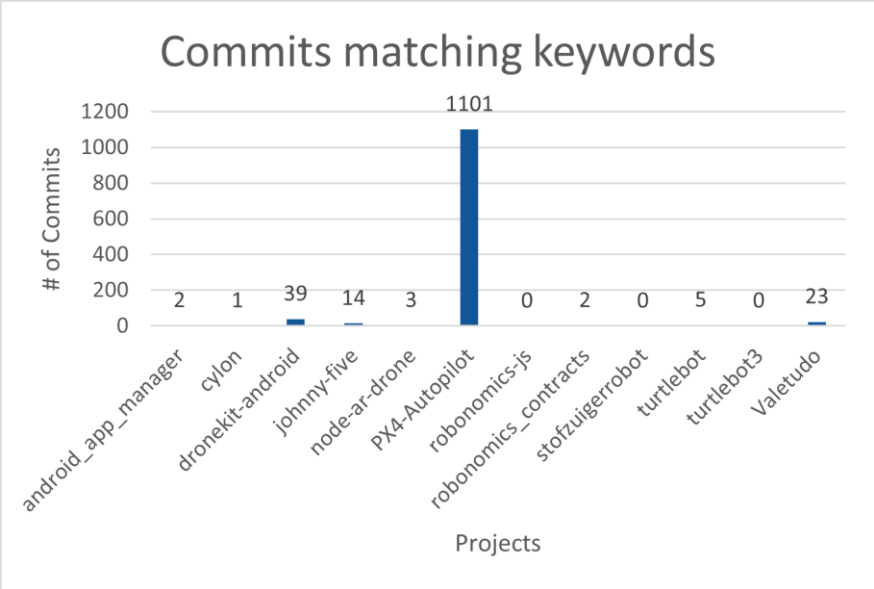

Figure 4: Commits matching keywords

Figure 5 has the resulting candidate antipattern commits against the total number of commits per project. Figure 5 shows PX4-Autopilot resulted in a relatively high amount of commits with 3.10%, where node-ar-drone has 1,07%. This observation was expected as the majority of commits analyzed in this deliverable were from PX4-Autopilot.

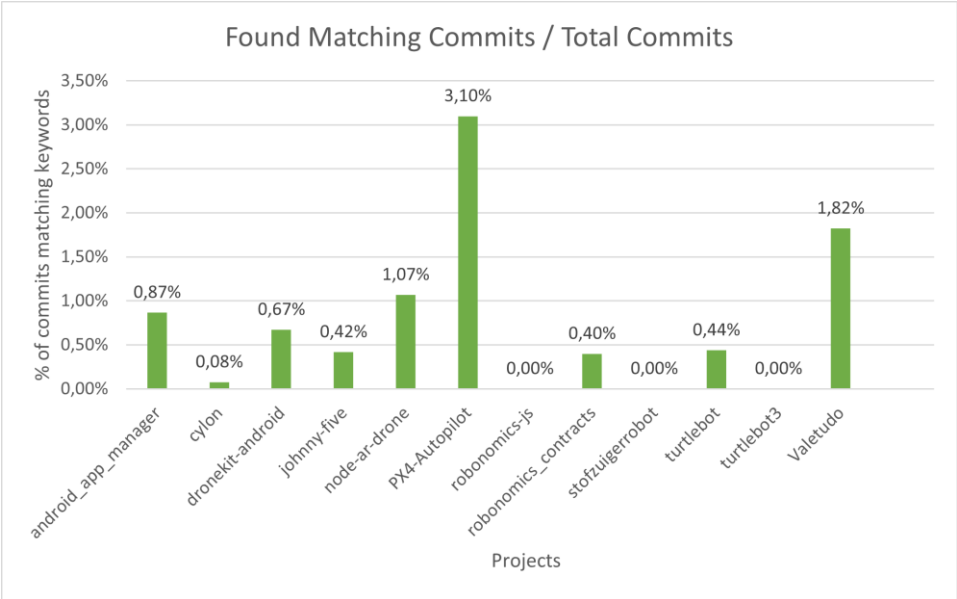

Figure 5: Matching Commits / Total Commits

4. CPS-RELATED PERFORMANCE ANTIPATTERN DETECTION

4.1 MANUAL ANALYSIS

After collecting the interesting commits, provided by PyRock, from the code history of the selected CPSs, two authors of this deliverable performed an extensive manual analysis on each of the commits. For analysis, they split the set of commits into two parts. Both authors followed the same methodology for the manual analysis:

1- Check the commit message and changes in the commit.

2- Check if the commit is mentioned in any issue or pull request

3- In case it is relevant, read comments and notes mentioned in the relevant issues and pull requests.

4- Analyze the methods and features implemented in the modified files.

5- Read the documentation of the changed classes.

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.

7- In case it is relevant, read the documentation regarding the software and hardware architecture of the projects under analysis.

For more accuracy, after completing the manual analysis task by each of the authors, the other author randomly reviewed about 50% of the cases. In case of disagreement about each analysis report, they discuss it in co-reviewing sessions to reach an agreement.

In general, we manually analyzed 319 commits from 12 CPS projects. In total, the entire manual analysis process took about five person-months.

4.2 MANUAL ANALYSIS RESULTS

According to manual analysis reports, first, we classified each of the commits into four general categories (also demonstrated in Figure 6):

- *Performance Issues:* We identified 104 commits that are either fixing or introducing performance issues. In our analysis, these commits are considered as *potential SPA exposers*. We provide more in-depth discussions about these cases in Section 5.

- *Non-performance Antipatterns:* This category refers to commits that expose general non-performance coding antipatterns such as Hard coding, Code duplication, etc. We identified 39 commits, which are revealing these types of antipatterns. However, these antipatterns are not the focus of our study as we concentrate only on performance issues.

- *CI/CD Performance Issues:* These cases are about antipatterns in continuous integration, a widely used software engineering practice. Duvall et al. [27]

present the antipatterns in this context. We identified 10 commits exposing these types of antipatterns. However, similarly to the previous category, detecting these antipatterns is not the goal of this study.

- *Not an Antipattern:* The manual analysis performed in this study did not find any antipatterns or issues in 181 commits. This high number of commits without revealing any antipattern stems from the generic keywords we used to identify the interesting commits. However, we selected these nonexclusive keywords to ensure that our analysis covers any code change in the history of CPS that has even the slightest chance to find any performance issue.

We do mention that some of the commits in our manual analysis are tagged with more than one antipattern as we found the were related to multiple types of performance (or non-performance) issues.

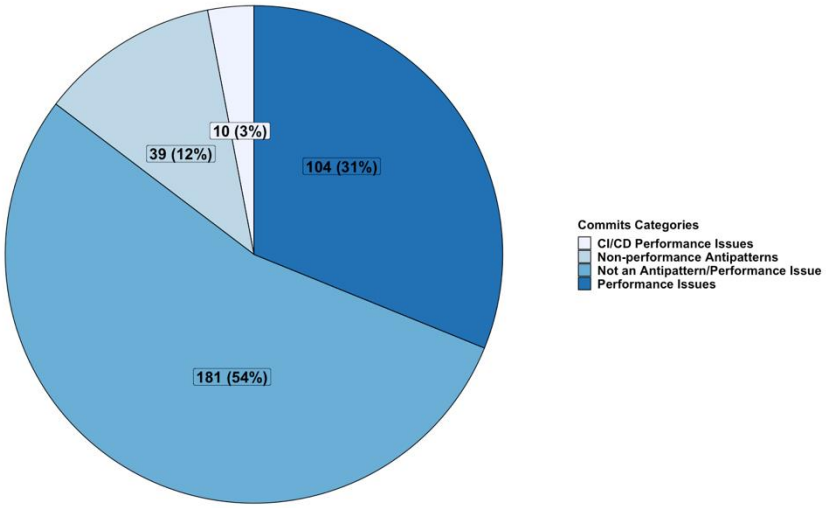

Figure 6: Results of manual analysis of 319 commits, which possibly expose a performance issue.

5. DETECTED PERFORMANCE ANTIPATTERNS

As mentioned in Section 4.2, in our manual analysis, we identified 104 commits that are exposing bad coding designs leading to performance issues (potential SPA exposers). We tagged each of these commits using three main categories:

1. *New CPS-PAs*, which are commits that are indicating a new type of performance-related bad coding practices that are not acknowledged in previous studies.

2. *General SPAs*, which are exposing common performance-related coding designs that can also occur in any other types of systems.

3. *Known CPS-PAs*, which are reflecting the CPS-related performance antipatterns that were previously introduced by Smith [2].

In the remaining parts of this sections, we discuss each category and their identified performance issues (or performance antipatterns).

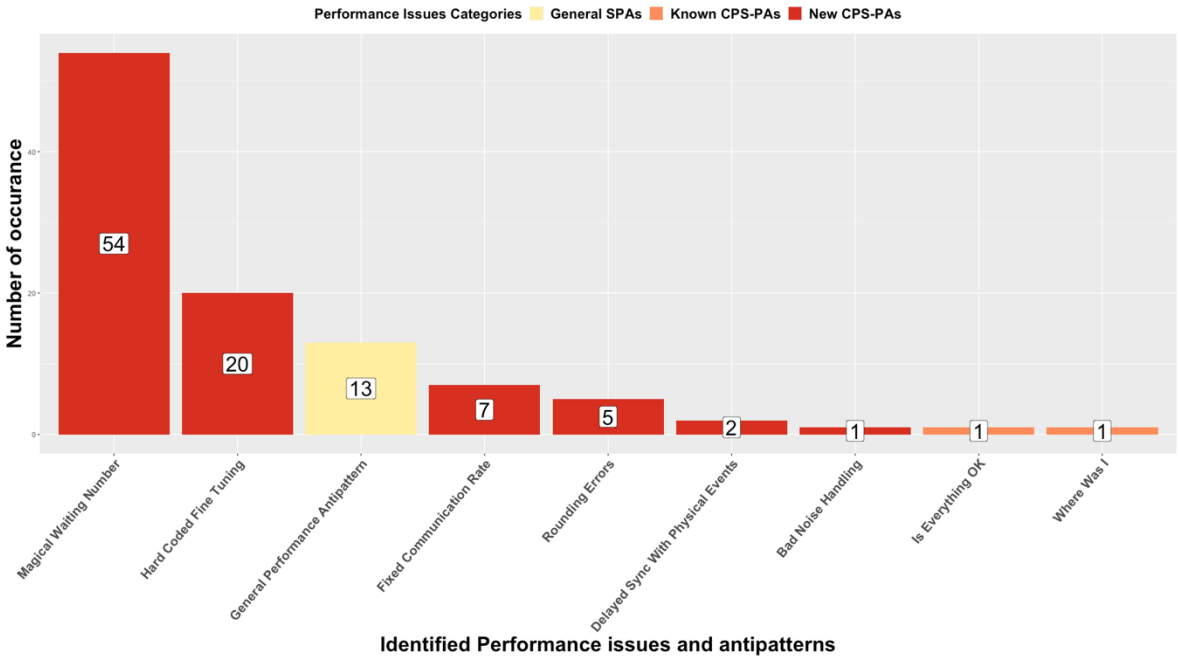

Figure 7: Identified performance issues and their frequency.

5.1 NEW CPS-PAS

In this category, we identify five performance-related bad coding practices in CPSs that are not identified in prior studies. For now, we call each of these bad practices as *potential SPAs*, while we are not sure if they are happening frequently enough to be considered as new CPS-related performance antipatterns. The five potential SPAs are illustrated by red bars in Figure 7. The most common one (*Magical Waiting Number*) is detected in 54 commits, and the least common one (*Unstable and Slow Noise Handling*) occurs only in one commit. In this section, we discuss each of these potential SPAs in detail.

5.1.1 Magical Waiting Number

This potential 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 expects the hardware devices response.

In the first scenario, the CPS assumes that it can continue its process without considering the execution status in the hardware that it is communicating with.

Example 1: In one of the reported bugs in Johnny Five (https://github.com/rwaldron/johnny-five/issues/1295), the CPS does not consider the instructions execution time in the LCDs. By changing the IO plugin used in this project to a faster one, this bug leads to unexpected outputs in the LCDs. For fixing this bug, developers have added 37 microseconds of sleep time which covers "the vast majority of instructions"[1]. For further details about this example, check the provided manual analysis report[2].

In the second scenario, the CPS developers set a hard-coded value as the maximum time required by a hardware piece to accomplish its task. In these cases, the CPS usually sleeps for the selected amount of time and then checks for the hardware's response. However, later, the value selected as the maximum response time is changed in the code history as the CPS developers discover a new situation for which this maximum timeout is not enough for some specific scenarios. This issue can happen either (i) when the CPS needs to be compatible with different types of hardware (e.g., with different speeds), or (ii) when the hardware's process time can change due to the external physical (known or unknown) events and circumstances.

In both cases, since each hardware in each physical condition might have different reactions to requests coming from the CPS, setting a global hard-coded fixed value to pause CPS before reading the response can lead to performance issues in the project. This timeout needs to be large enough to cover even the slowest hardware, but it should not introduce extra latencies in interacting with fast devices. The first challenge is verifying that the selected value supports both the slowest and fastest scenario. Our analysis detected many cases where this value was miscalculated and later changed for adapting more hardware and scenarios. The second issue might happen when this timeout is large or repeatedly executed. In this scenario, this large timeout can lead to a bottleneck for the cases with fast hardware.

Example 2: As another example, a reported issue in the Valetudo project (https://github.com/Hypfer/Valetudo/issues/799) exposes a bug in which sending a request to the Viomi robot vacuum cleaner (https://www.viomi.com) 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 CPS to complete the setting time zone task. According to the discussions about this bug in the Valetudo repository[3], this task can take about 10 seconds. Hence, as presented in Figure 8 this bug is fixed by increasing the timeout to 12000 milliseconds. The report of this manual analysis is available in our replication package[4].

[1] Look at this commit.
[2] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/johnny-five.md#commit-13
[3] https://github.com/Hypfer/Valetudo/pull/806
[4] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/Valetudo.md#commit-11

```
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");
                        });
                    }
                }
            });
```

Figure 8: Code change[5] to fix a performance issue in Valetudo. The highlighted texts are the added codes.

This bad code practice can lead to various minor (Example 1) or major (Example 2) issues.

For more cases regarding this potential CPS-PA, check the provided manual analysis. Also, the following list provides some more examples:

- Commit #187 in PX4-Autopilot[6].
- Commit #2 in PX4-Autopilot[7].
- Commit #13 in Valetudo[8].
- Commit #28 in Dronekit Android[9].

Is it a performance antipattern? As presented in Figure 7, we have detected Magical Waiting Time in 54/319 commits that we have manually analyzed in this deliverable. These commits are from four different projects: PX4-Autopilot, Valetudo, Johnny Five, Dronekit Android. These projects are developed in three 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 Time as a new CPS-PA.

5.1.2 Hard Coded Fine Tuning

This potential antipattern occurs when a setting or value is manually tweaked to improve the CPS's performance. In these cases, the result of the software performance

[5] To see the change in the project's repository, check this commit.
[6] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/PX4-Autopilot.md#commit-187
[7] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/PX4-Autopilot.md#commit-2
[8] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/Valetudo.md#commit-13
[9] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/dronekit-android.md#commit-28

is verified by seeing the end result of the change, rather than a calculated reason. Making a potential performance improvement with such a method, would be a slow process which could result in multiple adjustments to the same value.

Example 3: We detected in PX4-Autopilot two linked commits[10] where multiple stack sizes were reduced to free up some memory. However, one of the software modules in this CPS (sdlog) needed that amount of memory. Since there was no test making sure the sources required by sdlog were upheld, and thereby the build process did not fail after this memory reduction, CPS developers noticed the performance issue after implementation. These changes show that they are tweaking the settings manually to see the results in order to free up some memory.

Example 4: In another case in the code history of PX4 Autopilot[11], CPS developers adjusted the descend altitude without updating the documentation properly (i.e., adding the rationale behind adjusting this value). This change could have been the result of feedback received when using the system, in which the altitude adjustment is up for fine-tuning after deployment to experience the change, instead of calculated reasoning.

For more cases regarding this potential CPS-PA, check the provided manual analysis. Also, the following list provides some more examples:

- Commit #173 in PX4-Autopilot[12].
- Commit #142 in PX4-Autopilot[13].
- Commit #218 in PX4-Autopilot[14].

Is it a performance antipattern? As described in the examples, manual adjustments do not prove that these are the most optimal setting for the system. This could indicate that there might be a more optimal solution than the one provided, which could positively impact the software performance. It also would hold true for any area where values such as frequency and stack sizes are manually tweaked. Also, as demonstrated in Figure 7**,** we detected 20 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.

5.1.3 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 CPS performs as expected. As an example, Figure 9 presents the general

[10] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/PX4-Autopilot.md#commit-55
[11] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/PX4-Autopilot.md#commit-177
[12] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/PX4-Autopilot.md#commit-173
[13] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/PX4-Autopilot.md#commit-142
[14] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/PX4-Autopilot.md#commit-218

hardware and software architecture for PX4-Autopilot (one of the CPS projects in our analysis). This CPS provides a framework to control different vehicles automatically or manually. This system contains a hardware module for controlling the flights and, in general, movements (Flight Controller Figure 9), and another hardware for providing advanced features, such as collision prevention and object avoidance (Mission Computer in Figure 9). Also, these two main modules communicate with various other small hardware devices, such as sensors, cameras, and actuators.

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).

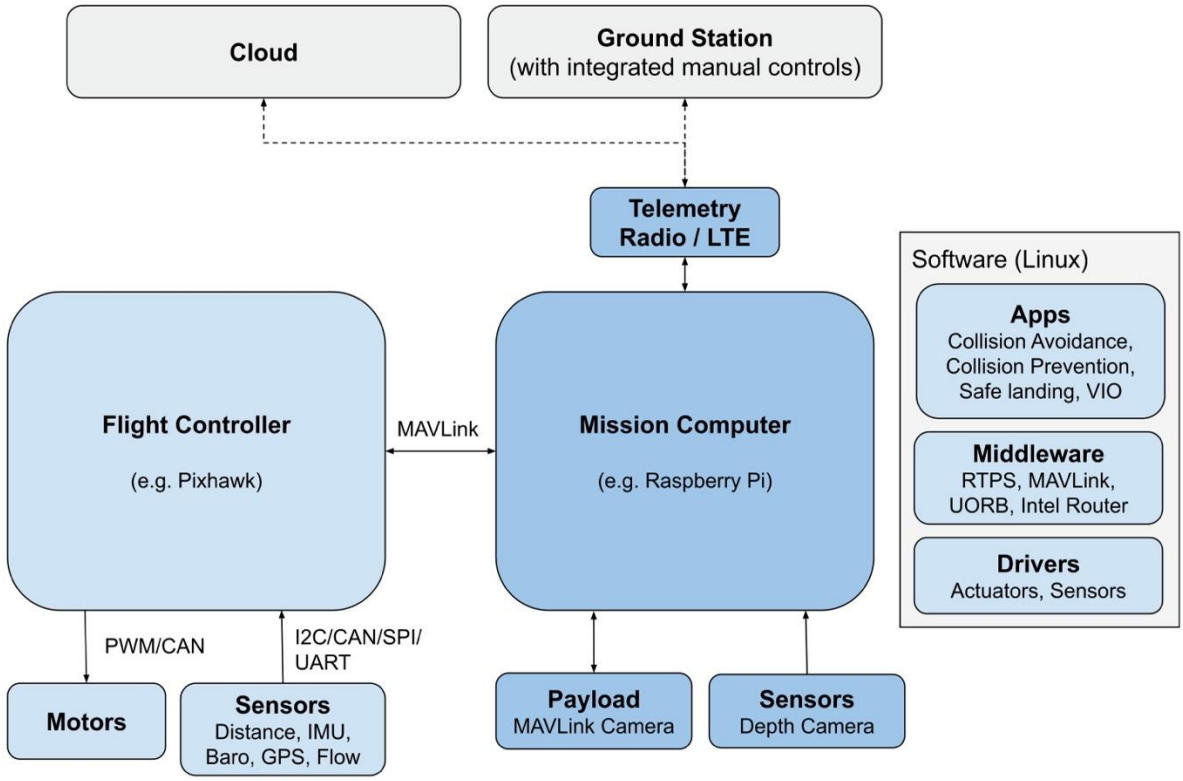

Figure 9: High-level overview of hardware and software modules for a typical PX4 Autopilot system. This figure is taken from the project's guide page[15].

In our analysis, we detected cases that CPS developers set a fixed communication rate between these devices and modules. In some other cases, they set a limit for these communication rates. Later, they find scenarios in which the low communication rate negatively affects the system's performance.

Example 5: As an example, in one of the commits[16] of the PX4 Autopilot system, CPS developers remove the 50 Hz sending rate limit in one of the software modules, called

[15] https://docs.px4.io/master/en/concept/px4_systems_architecture.html
[16] https://github.com/PX4/PX4-Autopilot/commit/8838b18da75d6f4354f73b38152c2ca98f9197aa

Attitude Controller. This software module is in the Flight Controller hardware (previously presented in Figure 9), which controls the vehicle's movement. Figure 10 depicts the high-level overview of the software architecture in Flight Controller. As presented in this figure, the Attitude Controller is the last step before sending the final output to the Output Drivers module for delivering the commands to motors and sensors. The change in this commit ensures that Attitude Controller provides the output as soon as possible to minimize latency. Also, this commit adds a comment to the code indicating that there is no need to add any limit in this module since the driver controls the communication rate.

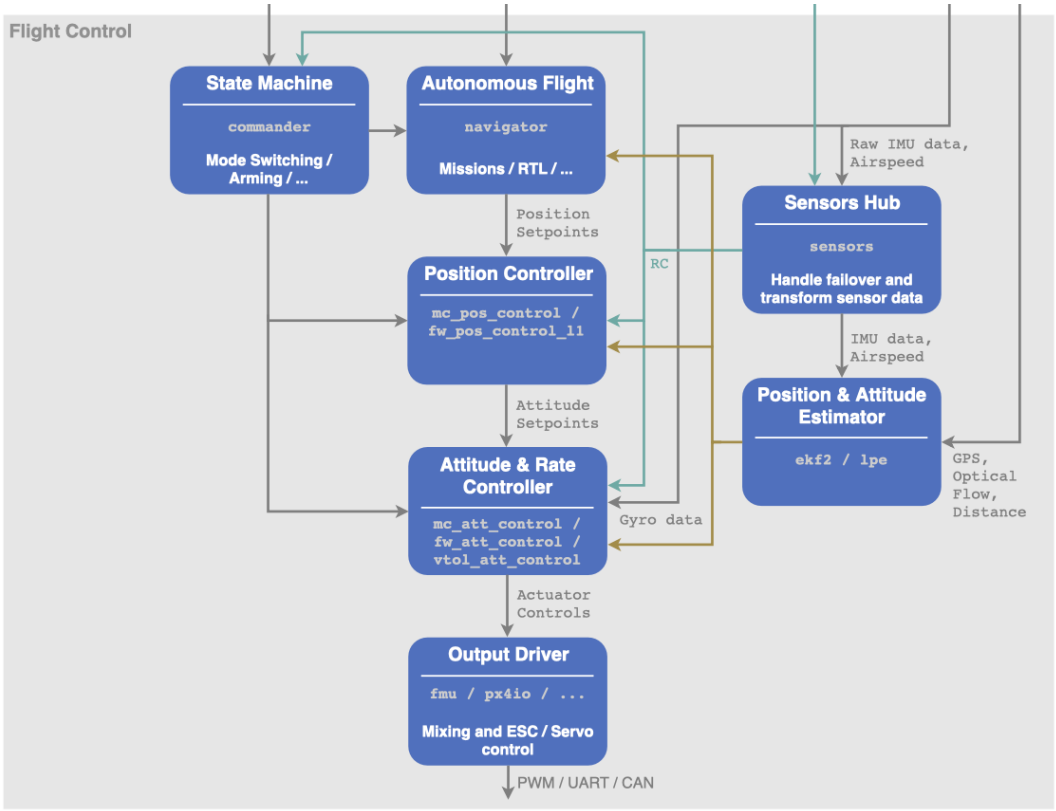

Figure 10: The general software architecture of Flight Controller in the PX4 Autopilot. This figure is taken from the project's guide page[17].

The solution for this antipattern is setting dynamic communication ratios between various software and hardware modules. This solution is implemented in the following example that we observed in our manual analysis. This example is also reported in our manual analysis[18].

Example 6: In the Dronekit Android project architecture, the 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 drone. However, they noticed that this default rate is not enough when the user enters the Tuning screen. Hence, in one of the commits[19], they implemented a dynamic procedure to increase the

[17] https://docs.px4.io/master/en/concept/architecture.html
[18] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/PX4-Autopilot.md#commit-209
[19] https://github.com/dronekit/dronekit-android/commit/2c9d9bc08147b0952eba4b6ef28701641a99bb21

communication rate when a user opens the Tuning screen and returns the rate back to default when they close it. The report of this manual analysis is available in our replication package[20].

For more cases regarding this potential CPS-PA, check the provided manual analysis. Also, the following list provides some more examples:

- Commit #90 in PX4-Autopilot[21].
- Commit #221 in PX4-Autopilot[22].
- Commit #186 in PX4-Autopilot[23].

Is it a performance antipattern? As is shown in Figure 7**,** we detected seven commits in our manual analysis that strive to tackle the fixed communication rate. We identified this performance issue in two projects: (i) Dronekit Android (implemented in Java), which provides a framework for developing applications for Android devices to control drones; and (ii) PX4 Autopilot (implemented in C++) that enables the automated and manual control of moving devices such as multicopters, small airplanes, airships, balloons, rovers, boats, and even small submarines. Also, we think that this performance issue can detected in any device containing multiple hardware devices. Given these findings, we consider the Fixed Communication Rate as a new CPS-PA.

5.1.4 Rounding Errors

In some scenarios, CPSs contain 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 rounding error in which one of the numbers is altered to a type with fewer decimals.

Example 7: In our analysis, we found five 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[24] changes the types of

[20] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/dronekit-android.md#commit-10
[21] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/PX4-Autopilot.md#commit-90
[22] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/PX4-Autopilot.md#commit-221
[23] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/PX4-Autopilot.md#commit-186
[24] https://github.com/dronekit/dronekit-android/commit/e29a5fde6f5c871ce956ffe6659e8b34f3d8a5b2

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 resolution of these numbers. The report of this manual analysis is available in our replication package[25].

At first look, this bad practice leads to functional issues. For instance, in Example 7, the miscalculation of the drone's latitude, longitude, and altitude leads to problems in how CPS functions. However, it can also negatively impact the performance of the CPS, indirectly. For example, in Example 7, 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.

For more cases about this potential CPS-PA, check the following examples:

- Commit #27 in Dronekit Android[26].
- Commit #58 in PX4-Autopilot[27].
- Commit #81 in PX4-Autopilot[28].

Is it a performance antipattern? As presented in Figure 7, we identified five instances of Rounding Errors in our manual analysis. These instances are detected in two projects for controlling various types of drones: Dronekit Android and PX4 Autopilot. These two projects are implemented in C++ and Java. We also 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). Given these findings, we consider Rounding Errors as a new CPS-PA.

5.1.5 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 exposes this performance issue. The following examples explain these two cases.

Example 8: We detected this performance issue in the TurtleBot 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 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

[25] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/dronekit-android.md#commit-25
[26] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/dronekit-android.md#commit-27
[27] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/PX4-Autopilot.md#commit-58
[28] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/PX4-Autopilot.md#commit-81

the commits we manually analyzed in this study[29]. 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 reassociate with a newly plugged-in USB device as the previous USB device. The report of this manual analysis is available in our replication package[30].

Example 9: We observed the second instance of this antipattern in PX4 Autopilot. In this case, the CDC/ACM driver is handling the requests coming from different devices. If these devices disconnect the communication, the driver does not understand the device is not available anymore and still tries to handle its requests. However, since the resource is unavailable, the driver enters an infinite loop, leading to a performance issue in the whole CPS. As mentioned by the comment added in the fixing commit[31], "The driver needs to reset the software (in order to flush the requests) and to disable the software connection when the device is unregistered". The report of this manual analysis is available in our replication package[32].

Is it a performance antipattern? Since we identified two instances of this issue in our analysis, we cannot confirm if this performance issue commonly occurs in the CPSs. Hence, we cannot consider Delayed Sync with Physical Events as an antipattern.

5.1.6 Bad Noise Handling

This performance issue happens in CPSs that include data collecting hardware devices such as sensors. The input collected from these devices can be noisy, and thereby, the CPS developers need to implement noise handling techniques to collect the accurate data. If this nose canceling process is not efficient, the CPS needs to collect more data, which leads to more I/O resource consumptions. Same as the previous section, as shown in Figure 7, we detected only one instance of performance issue. This scenario is presented in the following example.

Example 10: We detected this performance issue in the Johnny Five project. This CPS 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[33], we noticed that the implemented median calculation was not efficient enough. One of the commits[34] in this project improves noise handling procedure with a faster and more stable technique. The report of this manual analysis is available in our replication package[35].

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

[29] https://github.com/turtlebot/turtlebot/commit/f2d46b705722b61948313e3f2ec167dcaeeb3359
[30] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/turtlebot.md#commit-2
[31] https://github.com/PX4/PX4-Autopilot/commit/5b83507116be57e0c84daea74d30dea382f20f97
[32] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/PX4-Autopilot.md#commit-36
[33] https://github.com/rwaldron/johnny-five/pull/138
[34] https://github.com/rwaldron/johnny-five/commit/d3541a70d7767e52fb9aa67b32d9f32669abf45f
[35] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/johnny-five.md#commit-2

5.2 KNOWN CPS -PAS

As shown by Figure 7 (orange bars), we identified multiple known CPS-PAs, previously introduced by Smith [2]. These performance antipatterns are discussed in Section 2. This section presents the instances that we found in our analysis.

In total, we detected two commits, which are either fixing or introducing known CPS-PAs: one "*Where was I?"* and one "*Is everything OK?"*.

5.2.1 Where was I?

As described in Section 2, this performance antipattern can happen in various scenarios. One of these scenarios occur in cases where CPS attempts to connect to the previously known devices while they are not available to reconnect due to the changes in the environment. In these cases, Smith suggested that the timeout for reconnection should not be more than 1 minute to avoid the user's frustration [2]. However, in our manual analysis, we detect a commit[36] in Android App Manager where the WIFI connection timeout is increased to 90 seconds. The report of this manual analysis is available in our replication package[37].

5.2.2 Is everything OK?

During our manual analysis, we detected a commit[38] in TurtleBot, limiting the joint state publisher, which reports the states of the torque-controlled joints (i.e., their angles and locations). This commit assures at least a 0.1 milli seconds (10 Hz) gap between two joint state publishes. This change is related to the antipattern introduced by Smith, Is Everything OK, which is explained as: *"This antipattern refers to repeatedly checking the CPS platform status, such as the remaining battery life, storage space, etc."*. In this case, the commit makes sure that the CPS avoids this antipattern in checking the state of the joints. The report of this manual analysis is available in our replication package[39].

5.3 GENERAL SPAS

Besides the CPS-related performance antipatterns, we detected other common performance antipatterns that are also identified in other types of systems. As shown in Figure 7, we identified 13 instances of these common performance antipatterns in our CPS projects. Figure 11 illustrates these antipatterns and their number of occurrences. In general, we classify these antipatterns into two categories:

1- *Common performance antipatterns already reported by Smith* [2]*:* Smith's study reported 6 performance antipatterns that are common with other system types

[36] https://github.com/ros-android/android_app_manager/commit/980febbe5e1af05a21c9f08a8133e8b2804f2265
[37] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/android_app_manager.md#commit-1
[38] https://github.com/turtlebot/turtlebot/commit/b9ab8e2c7e6c8c067c74ed6b7b05f27c09a639f5
[39] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/turtlebot.md#commit-1

(i.e., also non-CPSs). We discussed these common antipatterns in Section 2 of this deliverable. However, this study identified three of these antipatterns:

a. One case of "Failing Dominoes".

b. One instance of "How many times do I have to tell you?".

c. One case of "Unnecessary processing".

2- *Common performance antipatterns only identified in our study:* We also detected six other common performance antipatterns that are not identified and reported by Smith. These antipatterns occurred more often than the antipatterns reported by Smith:

a. Three cases of "Using Massive Arrays".
b. Two cases of "Improper Instantiation" and "Unbuffered streams".

c. one case for "For-If", "Extraneous Fetching", and "Large payload sizes".

In the remainder of this section, we explain each of these performance antipatterns and provide one example for each of them.

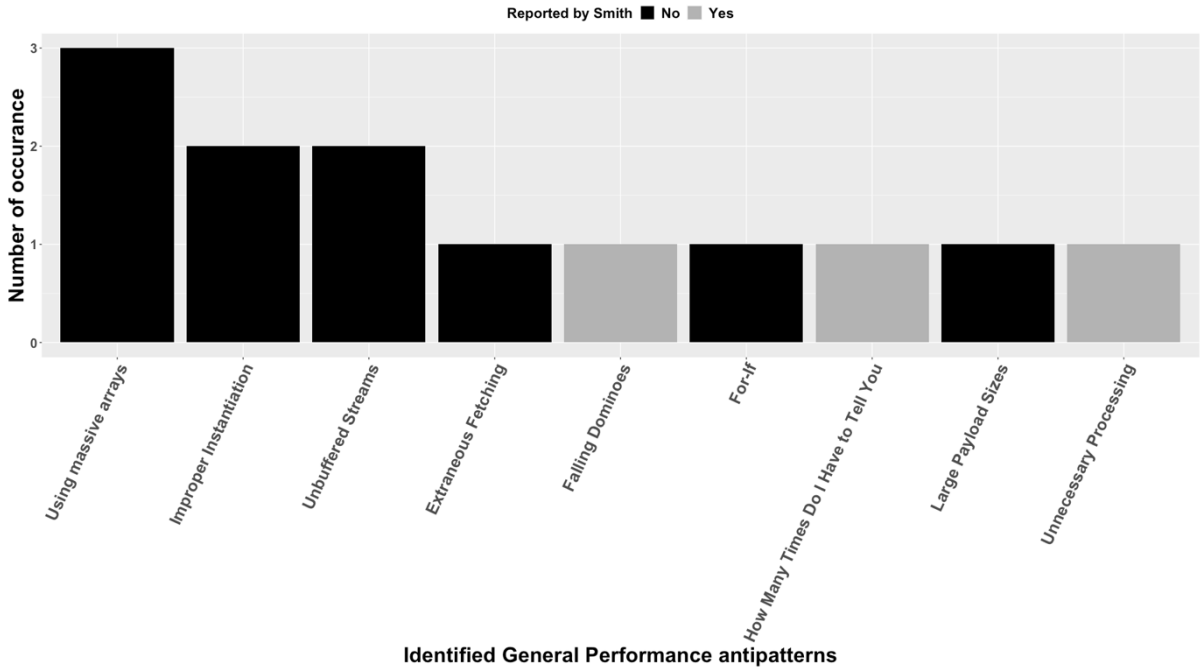

Figure 11: Overview of general performance antipatterns detected in this deliverable

5.3.1 Failing Dominoes

This antipattern refers to the spread of failure of a module to the other modules in the CPS [2]. In the code history of the Valetudo project, we identified a pull request[40] that provides a temporary solution for this type of antipattern. In this scenario, one of the components (Valetudo process) has a memory leak issue. This problem consumes most

[40] https://github.com/Hypfer/Valetudo/pull/198

of the memory resources up to the point that other components and processes are also experiencing performance issues. In this commit, CPS developers could not detect the reason behind the memory leak. So, they limit virtual memory allocation to ensure that each of the components has the required amount of memory. The CPS developers later fixed this issue in the subsequent changes[41].

5.3.2 How many times do I have to tell you?

This antipattern happens when a method is repeatedly called while it could be invoked only once [12]. We found one instance of this antipattern in the Valetudo project[42]. In this case, the code contains a nested loop (presented in Listing 1) only for calling a method (J*imp.rgbaToInt*). We identified a commit that fixes this antipattern by replacing this nested loop with only three invocations of the method, saving the returned values, and using them whenever needed.

```
for (var xOffset=0; xOffset<scale; xOffset++) {
   for (var yOffset=0; yOffset<scale; yOffset++) {
      image.setPixelColor(Jimp.rgbaToInt(rCol,gCol,bCol, alpha),
      xPos+xOffset, yPos+yOffset );
   }
}
```

Listing 1: An example of "How many times do I have to tell you?", detected in our analysis.

5.3.3 Unnecessary processing

This antipattern addresses the unnecessary execution of processes that are leading to performance issues for the CPS [11]. In this study, we detected one case in the Dronekit Android project, where a commit[43] identifies and fixes a performance issue by removing the heavy and unnecessary process of map reset (i.e., removing and reloading the whole map used in the UI of the drone controller) in the clear*()* method of PlanningActivity class.

5.3.4 Using Massive Arrays

According to Jezek et al. [28], one of the antipatterns is over-using arrays with large sizes, leading to memory leak issues. In our analysis, we detected three cases in which the CPS developers use the arrays with unlimited sizes (mostly for saving the history of the data coming from hardware devices). For instance, a commit[44] in Johnny-five exposes this antipattern in a module that stores the full history of the states of each Servo (a physical part in CPS) in an array. However, this project only needs the last five states of the servos, and keeping the entire history is unnecessary. Hence, this commit limits the array size to 5 to prevent memory leakage.

For more instances of this performance antipattern, check the following examples:

- Commit #19 in Dronekit Android[45].

[41] https://github.com/Hypfer/Valetudo/commit/a9fa64fcec23639b7367a0d3fa61b84cd07eaebe
[42] https://github.com/Hypfer/Valetudo/commit/ddc1c5f74ec06d63d2438932220e369678998342
[43] https://github.com/dronekit/dronekit-android/commit/faf8f09a5a9eeaeca02ce59f9ecb842e7c2f0794
[44] https://github.com/rwaldron/johnny-five/commit/79a715443e8a229a2ad7b71a5a352bac31f2246f
[45] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/dronekit-android.md#commit-19

- Commit #14 in Johnny-five[46].

5.3.5 Improper Instantiation

As explained by Microsoft documentation regarding the performance antipatterns[47], this antipattern links to frequently instantiating and destroying objects that can be shared and reused. Our analysis detected two cases of this performance antipattern. For instance, thanks to PyRock, we detected a commit[48] in Dronekit Android that replaces a class (*UiLanguage*) with a static method with the same functionality. As mentioned in the message of this commit, *"This avoids memory allocation for the creation of the* UiLanguage *object, and prevent possible leakage of Activity object, as only an application context is needed to make the config update."*

Commit #7 in Valetudo[49] is another instance of this performance antipattern that can be found in our manual analysis.

5.3.6 Unbuffered Streams

Unbuffered Stream is a known Java performance antipattern characterized by a system reading a file directly without utilizing buffered memory[50]. However, this issue can also happen in other programming languages. For instance, we detected a commit[51] in PX4 Autopilot, which is implemented in C++, that reveals and fixes the same antipattern. This commit ensures that the system reads the date from a memory buffer instead of reading it directly from a file. Since many CPSs also work with files, this antipattern can also exist in this type of systems.

Commit #17 in Dronekit Android[52] is another instance of this performance antipattern that can be found in our manual analysis.

5.3.7 Extraneous Fetching

This antipattern may happen if the CPS tries to minimize I/O requests by retrieving all the data that it might need[53]. This leads to the loading of unnecessary data and consequently using more memory resources. We observed one instance of this antipattern in the TurtleBot project. In this scenario, *urdf.xacro* files load unnecessary content, including every robot configuration that contains every base, stack, and sensor combination, and thus many unnecessary data is loaded into memory. This antipattern is fixed in a pull request[54], which refactors each robot configuration file to include only the required base, stacks, and sensor files.

[46] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/johnny-five.md#commit-14
[47] https://docs.microsoft.com/en-us/azure/architecture/antipatterns/improper-instantiation/
[48] https://github.com/dronekit/dronekit-android/commit/ffc8e75d7c692c5977516339bb2575c4a1266d5d
[49] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/Valetudo.md#commit-7
[50] https://www.odi.ch/prog/design/newbies.php
[51] https://github.com/PX4/PX4-Autopilot/commit/35c82ff2fc63ab823770f9776e6b6a0f81cd4452
[52] https://github.com/ciselab/CPS_repo_mining/blob/main/analysis/dronekit-android.md#commit-17
[53] https://docs.microsoft.com/en-us/azure/architecture/antipatterns/extraneous-fetching/
[54] https://github.com/turtlebot/turtlebot/pull/238

5.3.8 For-If

This antipattern addresses the issue when we have unnecessary "if conditions" (i.e., conditions that can be handled outside the loop) in the "for loop" [55]. One of the commits manually analyzed in our study addresses this antipattern. In this commit, a loop needs only 31 indexes of an empty array called *segment*. However, in the version with the antipattern, the code checks the availability of each segment index whenever needed in the for loop (look at Listing 2). Since only 31 indexes of this segment are used in the loop, the fixing commit sets these indexes before the loop and removes the extra if condition from inside the loop.

```
For (…){
   if (!parsedBlock.segments[segmentId]) {
      parsedBlock.segments[segmentId] = [];
   }
}
```

Listing 2: an example of for-loop antipattern

5.3.9 Large Payload Sizes

In this study, we detected a commit[56] that fixes spurious memory allocations in MAVLink communications. This commit reduces the maximum size of the payloads created and sent by MAVLink (a protocol for communicating with unmanned vehicles). This antipattern can occur in systems such as CPSs where there are lots of networking tasks are involved. It is worth mentioning that the MAVLink max size was 512 bytes. However, in the new version of the MAVLink, this threshold is reduced to 256[57]. The large payloads lead to more memory and I/O consumption.

6. REPLICATION OF RESULTS

All of the tools and results presented in this deliverable (including the implementation of PyRock and the commits collected by this tool) are openly available on GitHub[58]. Moreover, all the PyRock executions can be replicated using the README file provided in this artifact. Also, we will present the instruction to replicate this study in this deliverable. Besides, the reports regarding the extensive manual analysis performed in this study are also available in this artifact[59]. The artifact includes a Docker file to ease the replication in any machine.

6.1 REPRODUCING THE COMMIT SELECTION USING PYROCK

For portability and replicability of this tool, we use docker. For easier docker setup, we provide two scripts for building docker image and running the docker container.

[55] https://devblogs.microsoft.com/oldnewthing/20111227-00/?p=8793
[56] https://github.com/dronekit/dronekit-android/commit/27b16751ffad6dccda2fb45717e61377fce28f78
[57] https://mavlink.io/en/guide/serialization.html#payload_truncation
[58] https://github.com/ciselab/CPS_repo_mining
[59] https://github.com/ciselab/CPS_repo_mining/tree/main/analysis

6.1.1 Docker image setup

Execute the following script for building the docker image:

```
. docker_scripts/build-cps-repo-mining.sh
```

6.1.2 Docker container setup

The script docker_scripts/run-cps-repo-mining-container.sh is created for this task. For running the mining in remote mode, this script can be executed without any input parameter. However, to perform the mining process for local mode, we should pass the absolute path of the directory containing local repositories as the input argument:

```
. docker_scripts/run-cps-repo-mining-container.sh [local_repositories]
```

6.1.3 Repositories

The list of repositories used in this project is available at pd/dict_repo_list.py. To run PyRock in local mode, first, run the following script to clone the repositories in the given output directory:

```
docker exec -it cps-repo-mining-container bash -c “python3 pd/dict_repo_list.py [out-
put_directory]”
```

The output of this directory can be passed as an input for commit selection.

6.1.4 Commit selection

To run commit selection run the following command:

```
docker exec -it cps-repo-mining-container bash -c "python3 pd/repository_commits_mining.py
[la/ra] (local_repositories_dir)"
```

The input argument can be la (local mode) or ra (remote mode). In case of selecting local mode, the second input argument should provide the directory containing the local repositories of the projects given in pd/dict_repo_list.py.

The result of commit selection will be saved in results/ in the root directory.

6.2 REPRODUCE ANALYSIS

6.2.1 Manual analysis

The reports regarding the result of our analysis are also available on our GitHub repository[60]. In this directory, each file contains the manual analysis report for each of the CPS projects.

To parse the results of manual analysis in csv file, run the following command:

```
docker exec -it cps-repo-mining-container bash -c "python3
```

[60] https://github.com/ciselab/CPS_repo_mining/tree/main/analysis

```
pd/manual_analysis_report_parser.py"
```

This script parses all the reports and generates results.csv in the root directory. This csv file will be used to run the R script (presented in Section 6.2.2) to generate the final figures used in this deliverable.

6.2.2 Automated analysis

To run the analysis script, go to the following directory:

```
cd data-analysis/r-scripts/
```

Then, run initial_analysis.R:

```
Rscript initial_analysis.R
```

7. FUTURE WORK

This deliverable is the first step of fulfilling the goal in task T5.1 in the COSMOS project. The nest step of this task is to implement a refactoring framework for detecting and providing the performance-related antipatterns in the given CPS projects.

As the next step, we use the performance antipatterns identified in this deliverable to present solutions for refactoring the CPS. Next, given the identified CPS performance antipatterns and the introduced solutions, we utilize machine learning and static analysis techniques to (i) identify CPS performance antipatterns in the given CPS projects and (ii) suggest the refactoring guidelines according to the detected antipatterns.

By providing a framework suggesting performance-related solutions, we help the CPS developers deliver CPS products with good trade-offs between functional and non-functional properties, and thereby achieve a product with high performance and efficient resources usages.

As another step to extend this study, we plan to perform the same analysis on the CPS projects provided by the COSMOS industrial partners. This extension aims to analyze how frequently the performance antipatterns identified in this analysis occur in these CPSs.

8. CONCLUSION

One of the challenges in the development process of CPSs is assuring the performance of the system. One of the practices towards achieving this goal is to identify the performance antipatterns occurring in CPSs (i.e., bad coding practices commonly happen while developing CPSs). By understanding these antipatterns, we can later introduce refactoring operations that can suggest solutions to tackle these antipatterns.

Hence, in this deliverable, we performed an extensive analysis on a set of diverse open-source CPS projects available on GitHub. We investigated the code history of these projects to detect code changes that are addressing or introducing performance issues.

According to the frequency of these performance issues in our analysis, we identified if they are common enough to be considered performance antipatterns.

In total, we identified six types of new CPS-specific performance issues, among which four of them are wide-spread across multiple application domains and programming languages and, therefore, can be considered as performance antipatterns. Moreover, in our study we have also identified instances of CPS-specific performance antipatterns that were previously introduced in literature. Besides, we identified nine general performance antipatterns that can also happen in systems other than CPSs.

Knowing these antipatterns can be helpful from two aspects: (i) By knowing these performance antipatterns, we can utilize techniques, such as static analysis and machine learning, to introduce automated methods for detecting antipatterns in CPSs. (ii) We can design refactoring operations to address the detected performance antipatterns. For the following deliverable, we aim to achieve these two goals using the outcome of this deliverable. For practitioners, the implication is that they can follow an automated procedure to get suggestions about refactoring candidates that can improve the performance of their CPS.

9. BIBLIOGRAPHY

[1] H. Chen, "Applications of cyber-physical system: a literature review," *Journal of Industrial Integration and Management*, vol. 2, 2017.

[2] C. U. Smith, "Software performance antipatterns in cyber-physical systems," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, 2020.

[3] C. U. Smith and L. G. Williams, "Software Performance Antipatterns for Identifying and Correcting Performance Problems," in *Int. CMG Conference*, 2012.

[4] C. U. Smith and L. G. Williams, "Software performance antipattern," in *Proceedings of the 2nd international workshop on Software and performance*, 2000.

[5] C. Trubiani, A. Di Marco, V. Cortellessa, N. Mani and D. Petriu, "Exploring synergies between bottleneck analysis and performance antipatterns," in *Proceedings of the 5th ACM/SPEC International Conference on Performance engineering*, 2014.

[6] A. Aleti, C. Trubiani, A. van Hoorn and P. Jamshidi, "An efficient method for uncertainty propagation in robust software performance estimation," *Journal of Systems and Software*, vol. 138, 2018.

[7] R. Calinescu, M. Češka, S. Gerasimou, M. Kwiatkowska and N. Paoletti, "Designing robust software systems through parametric Markov chain synthesis," in *2017 IEEE International Conference on Software Architecture (ICSA)*, 2017.

[8] F. Bushmann, R. Meunier and H. Rohnert, *Pattern-oriented software architecture: A system of patterns. John Wiley&Sons 1 (1996), 476*, 1996.

[9] E. Gamma, R. Johnson, R. Helm, R. E. Johnson, J. Vlissides and others, Design patterns: elements of reusable object-oriented software, Pearson Deutschland GmbH, 1995.

[10] W. H. Brown, R. C. Malveau, H. McCormick and T. J. Mowbray, AntiPatterns: refactoring software, architectures, and projects in crisis, John Wiley & Sons, Inc., 1998.

[11] C. U. Smith and L. G. Williams, "More New Software Performance Antipatterns: Even More Ways to Shoot Yourself in the Foot," *Computer Measurement Group Conference*, pp. 717-725, 2003.

[12] C. S. Cates, "Where's Waldo: Uncovering Hard-to-Find Application Killers," in *Int. CMG Conference*, 2004.

[13] R. F. Dugan Jr, E. P. Glinert and A. Shokoufandeh, "The sisyphus database retrieval software performance antipattern," in *Proceedings of the 3rd international workshop on Software and performance*, 2002.

[14] C. U. Smith and L. G. Williams, "New software performance antipatterns: More ways to shoot yourself in the foot," in *Int. CMG Conference*, 2002.

[15] Y. Yina, K. E. Steckeb and D. Lic, "The evolution of production systems from Industry 2.0 through Industry 4.0,"

International Journal of Production Research, vol. 56, p. 848–861, 2018.

[16] S. O. Okolie, S. O. Kuyoro and O. B. Ohwo, "Emerging Cyber-Physical Systems : An Overview," *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, pp. 306-316, 2018.

[17] A. Humayed, J. Lin, F. Li and B. Luo, "Cyber-physical systems security—A survey," *IEEE Internet of Things Journal*, vol. 4, no. 6, pp. 1802-1831, 2017.

[18] C. K. Keerthi, M. A. Jabbar and B. Seetharamulu, "Cyber physical systems (CPS): Security issues, challenges and solutions," in *2017 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)*, 2017.

[19] J. Shi, J. Wan, H. Yan and H. Suo, "A survey of cyber-physical systems," in *2011 international conference on wireless communications and signal processing (WCSP)*, 2011.

[20] A. Bondi, Foundations of software and system performance engineering: process, performance modeling, requirements, testing, scalability, and practice, Pearson Education, 2015.

[21] A. E. Hassan, "The road ahead for mining software repositories," in *Frontiers of Software Maintenance*, 2008.

[22] K. K. Chaturvedi, V. B. Singh and P. Singh, "Tools in Mining Software Repositories," *13th International Conference on Computational Science and Its Applications*, 2013.

[23] D. a. A. M. a. B. A. Spadini, Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018, ACM Press, 2018.

[24] V. Lenarduzzi, N. Saarimäki and D. Taibi, "The Technical Debt Dataset," *In Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE'19)*, pp. 2-11, 2019.

[25] A. M. Kazerouni, C. A. Shaffer, S. H. Edwards and F. Servant, "Assessing Incremental Testing Practices and Their Impact on Project Outcomes," *In Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*, p. 407–413, 2019.

[26] P. Thongtanunam and A. E. Hassan, "Review Dynamics and Their Impact on Software Quality," *IEEE Transactions on Software Engineering*, 2019.

[27] P. Duvall and M. Olson, "Continuous delivery: Patterns and antipatterns in the software life cycle," [Online]. Available: https://dzone.com/refcardz/continuous-delivery-patterns#section-1.

[28] K. Jezek and R. Lipka, "Antipatterns causing memory bloat: A case study," in *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017.

[29] R. Pinciroli, C. U. Smith and C. Trubiani, "QN-based Modeling and Analysis of Software Performance Antipatterns for Cyber-Physical Systems," *ICPE*, 2021.

[30] B. Ferro Castro, "Pattern-oriented software architecture: A system of patterns," *Computaci{ó}n y Sistemas*, vol. 1, no. 002, 1969.

[31] C. Larman, "Enterprise JavaBeans 201: The Aggregate Entity Pattern-The appropriate design of entity beans-especially when they are backed by a relational database-is the subject of ongoing debate. So how do," *Software Development*, vol. 8, no. 4, pp. 46-55, 2000.