
COSMOS

DevOps for Complex Cyber-physical Systems

**Project Number 957254**

# D6.3 Prototypes for the Quality Assessment and Monitoring of CPS in the Field

**Version 1.0**
**29 June 2022**
**Final**

**Public Distribution**

**Intelligentia & Zurich University of Applied Sciences**

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

# Document Control

| Version | Status | Date |
|---|---|---|
| 0.1 | Outline | 1 June 2022 |
| 0.2 | Background completed | 10 June 2022 |
| 0.5 | First full draft | 20 June 2022 |
| 0.6 | Initial internal review version | 20 June 2022 |
| 0.7 | Further editing draft | 26 June 2022 |
| 0.8 | Initial Updates addressing reviewers' comments | 27 June 2022 |
| 0.9 | Updates addressing final reviewer comments and final draft release | 29 June 2022 |
| 1.0 | Final QA version for EC delivery | 29 June 2022 |

# Table of Contents

# Executive Summary

This report describes the first steps in the development of a framework for monitoring CPS quality attributes (QAs) that may change/degrade over the software evolution, with the purpose of monitoring CPS behavioral states, detecting/predicting CPS degradation forms, and recommending, when possible, system recovery and (micro-) fixes. The monitoring and fixing of QAs include also the assessment of the CPS *vulnerability proneness* (e.g., predicting the security attack surface of CPS) [154], a topic not yet investigated in the literature.

COSMOS targeted innovation is to extend traditional DevOps pipelines with monitoring CPS states and QAs from DevOps and CPS assets (e.g., simulation environments, sensors, etc.), to focus on CPS specific degradation forms, thus supporting their automated monitoring, prediction, and fixing.

From a high-level point of view, there is still limited research in the state-of-the-art concerning the automated monitoring, prediction, and fixing of CPS states and QAs from DevOps and CPS assets. Such investigations are critical to shed some light on the types of behavioral states and QAs that affect or characterize CPSs and how they can potentially lead to unsafe behaviors or CPS degradation forms in different CPS domains. Given such a background, this report begins by describing the current state-of-the-art and identified gaps in the literature concerning the following relevant research topics:

- Background information on current state of development and evolution of CPSs;
- Background information on current state of development and evolution of specific CPSs, close to COSMOS use case partners (AICAS and GMV) such as Unmanned Aerial Vehicles and Self-driving cars;
- Summary of main general research on monitoring CPS states;
- Discussion of background information on current state of monitoring of CPS states in the context of two specific CPSs, close to COSMOS use case partners (AICAS and GMV) such as Unmanned Aerial Vehicles and Self-driving cars;
- Focused related studies on flaky tests analysis, identification, and root-cause analysis;
- Focused related studies on monitoring and propagation of CPS Changes.
- Focused related studies on monitoring and propagation of vulnerabilities.

The deliverable discusses the general architecture of COSMOS Component for Quality Assessment and monitoring of CPS. Then, it discusses the ongoing implementation of developed prototypes. Finally, the deliverable outlines next steps and future work.

# 1 Introduction

This report describes the first steps in the development of a framework for monitoring CPS quality attributes (QAs) that may change/degrade over the software evolution, with the purpose of monitoring CPS behavioral states, detecting/predicting CPS degradation forms, and recommending, when possible, system recovery and (micro-) fixes. The monitoring and fixing of QAs include also the assessment of the CPS *vulnerability proneness* (e.g., predicting the security attack surface of CPS) [154], a topic not yet investigated in the literature.

COSMOS targeted innovation is to extend traditional DevOps pipelines with monitoring CPS states and QAs from DevOps and CPS assets (e.g., simulation environments, sensors, etc.), to focus on CPS specific degradation forms, thus supporting their automated monitoring, prediction, and fixing.

From a high-level point of view, there is still limited research in the state-of-the-art concerning the automated monitoring, prediction, and fixing of CPS states and QAs from DevOps and CPS assets. Such investigations are critical to shed some light on the types of behavioral states and QAs that affect or characterize CPSs and how they can potentially lead to unsafe behaviors or CPS degradation forms in different CPS domains. Given such a background, this report begins by describing the current state-of-the-art and identified gaps in the literature concerning the following relevant research topics:

- Background information on current state of development and evolution of CPSs (Section 2.1);
- Background information on current state of development and evolution of specific CPSs, close to COSMOS use case partners (AICAS and GMV) such as Unmanned Aerial Vehicles and Self-driving cars (Section 2.1.1 and Section 2.1.2);
- Summary of main general research on monitoring CPS states (Section 2.2);
- Discussion of background information on current state of monitoring of CPS states in the context of two specific CPSs, close to COSMOS use case partners (AICAS and GMV) such as Unmanned Aerial Vehicles and Self-driving cars (Section 2.2.1);
- Focused related studies on flaky tests analysis, identification, and root-cause analysis (Section 2.3);
- Focused related studies on monitoring and propagation of CPS Changes (Section 2.4).
- Focused related studies on monitoring and propagation of vulnerabilities (Section 2.5).

The deliverable discusses the general architecture of COSMOS Component for quality assessment and monitoring of CPS (Section 3.1). Then, it discusses the ongoing implementation of developed prototypes:

- The prototype for Monitoring UAV states, along with the architecture and/or preliminary results (Section 3.2). This task is an essential step toward the analysis of types of UAV States, relevant also for the GMV use case.
- The prototype for Monitoring SDC states, along with the architecture and/or preliminary results (Section 3.3). This task is an essential step toward the analysis of types of SDC States, relevant also for the AICAS use case.
- The prototype for flaky tests analysis and identification for UAVs, along with preliminary results (Section 4). This task is an essential step toward the analysis of CPS flaky tests, an important QA of UAV States, relevant also for the GMV use case.
- The prototype for flaky tests analysis and identification for SDCs, along with preliminary results (Section 5). This task is an essential step toward the analysis of CPS flaky tests, an important QA of SDC States, relevant also for the AICAS use case.

Finally, the deliverable outlines next steps and future work.

# 2 Background and Related work

This section overviews the current state-of-the-art and identified gaps in the literature, providing important background information for better contextualizing the innovations targeted by COSMOS.

## 2.1 Development of Cyber-physical Systems

Empirical studies have shown that CPSs are more difficult and expensive to test and integrate than traditional software systems [72, 165]. Common reasons for this are that the final version of the hardware is often not available, and the integration of hardware components requires a high, and error-prone manual effort. For these reasons, recent studies have investigated the challenges of CPS development and identified that an effective evolution of CPSs requires more flexible development and verification approaches, integrating Model-in-the-Loop (MiL), Software-in-the-Loop (SiL), and Hardware-in-the-Loop (HiL) paradigms [4].

Giraldo et al. [65] conducted a literature review on CPS research topics, finding that they can be categorized into security, privacy, defense, or domain-specific. Also, their literature review shows that there is a lack of research and tools for supporting CPS development and evolution. A follow-up study by Törngren and Sellgren [165] discusses how CPS engineering deals with the inner complexity of CPSs design and the challenges that arise from the environments in which the CPSs operate. According to Törngren and Sellgren, while semi-automated integration happens through software, there are distinguishing characteristics between software and physical systems that make it hard co-designing hardware and software. Those characteristics entail the usage of different approaches, techniques, abstractions, platforms, faults & failure modes, and development practices [165]. Törngren and Sellgren conclude that CPS development and testing need rapid prototyping, code/test generation, and various testing phases [150] encapsulating model-in-the-loop (MiL), software-in-the-loop (SiL), and hardware-in-the-loop (HiL) activities to effectively identify bugs in CPSs.

Understanding the nature of CPS safe and unsafe states as well as QAs of CPSs constitutes key areas for supporting the development (e.g., by conceiving monitoring and prevention techniques in case of harmful CPS behaviors), and evolution (e.g., predicting and fixing unsafe states) of CPSs.

### 2.1.1 Development of Unmanned Aerial Vehicles

With the boost of CPS in both academia and industry over the past decade, we have witnessed impressive advancements in the technology available in healthcare, avionics, automotive, railway, and the robotics sectors [173, 33]. Unmanned Aerial Vehicles (UAVs) [181], e.g., drones equipped with onboard cameras and sensors, have already demonstrated that autonomous flights are possible in real environments. This sparked great interest in a plethora of application scenarios, with crop monitoring [30], surveillance [13], infrastructure maintenance [136], medical and food delivery [40], military [70], search and rescue in disaster areas [37] representing only some of the relevant applications of UAVs (either autonomous or teleoperated). By 2027, the global UAV market size is projected to reach 25.13 billion USD, with a compound annual growth rate of 12.23% compared to 10.72 billion USD in 2019 [55]. As the autonomous flying robots and the consumer UAV market flourish, safe collocated human-UAV interactions are becoming increasingly important [176].

Remotely controlled air crafts have been in development since the first world war, where the early research and development was around the use of unmanned aircraft as weapons of war [92]. During the early stages of the UAV evolution, there were two main strategies for vehicles control: (1) Remote Piloted Vehicles (RPVs) and (2) development of UAVs capable of flying pre-programmed trajectories either as an offensive tool or for Intelligence, surveillance and reconnaissance [92]. Due to the tactical needs of the United States Air Force (USAF), there was a rapid research and development in the late 1970s that would drastically increase the capability of UAVs [157]. However, it was only in the 1980s that thanks to the increased availability of lightweight processors, sensor systems and GPS, UAVs were be able to perform complex tasks [51]. From

the mid-1980s, the research and development of aerial robots increased significantly, powered by the wide availability of small-scale helicopters available also to of academics and universities[51, 157].

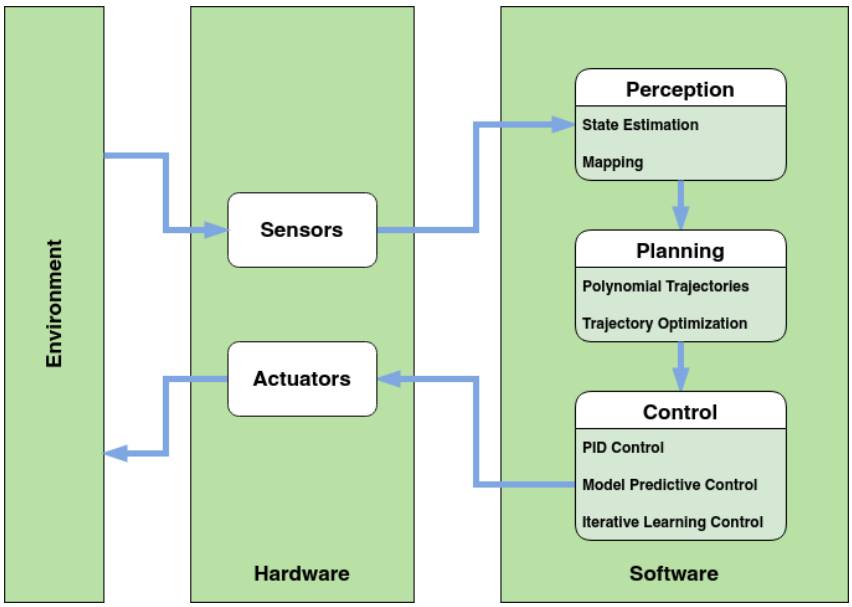


Figure 1: Typical robotic cycle with core competencies [131]

**UAV Architecture** The overall mechanism of a UAV is based on the so-called *Robotic Cycle* (Figure 1). The robotic cycle encompasses the underlying architecture of a robotic system and defines the relationship between a robot's main components (hardware and software) and their interaction with the surrounding environment [87]. The hardware comprises the robot sensors used to detect changes or events in the surrounding environment and the actuators, which are parts that act on the environment [87].

**Sensors**. There exists a vast amount of sensors which UAVs can leverage, but the minimum required sensor is the Inertial Measuring Unit (IMU). The IMU contains two sensors, a 3-axis gyroscope which measures the UAVs rotational motion and a 3-axis accelerometer [14], which measures the UAV's acceleration [51]. IMU is often accompanied by other sensors such as GPS, barometer and magnetometer in order to reduce the positional error of the UAV by combining all the sensors measurements in order to reduce the errors caused by sensor noise and inaccuracies. On more state of the art implementations, the UAV sensors include also optical flow sensors [115], LIDAR [85] and video cameras [109], all of which aim to improve the UAV's ability to track its movements and perform obstacle recognition.

**Actuators**.The actuators of the UAV are components which change the movement of air around them in order to create a force on the UAV, which leads to it moving in a certain way. In case of a quadcopter, there are 4 propellers which are fixed to a motor, and by varying the control signal to each motor, it is possible to control the quadcopter's rotation and movement in any direction. Typically, UAVs have 6 degrees of freedom (DOFs), as illustrated in Figure 3. Three transitional movements: longitudinal (along x-axis), lateral (along y-axis), and vertical (along z-axis) and three complementary rotational movements along each axis: *roll* (around x-axis), *pitch* (around y-axis) and *yaw* (along z-axis). The rotational movements express the UAV's *attitude*.

The software is traditionally decomposed into three sequentially placed primitives [120]; *Perception*, *Planning*, and *Control* [87]. The information gathered by the system's sensors is fed to the perception layer, which elaborates the information and makes the processed data available to the planning layer. The planning layer uses the provided information in order to compute a path strategy which the control layer has to actuate. The control layer is responsible for processing the action into signals for the actuators (e.g. rotors) with in turn act on the robot's actuators. This sequence of interaction is also referred as the *sense-plan-act paradigm* and is repeated at high frequency in order to keep the UAV stable [87, 51].

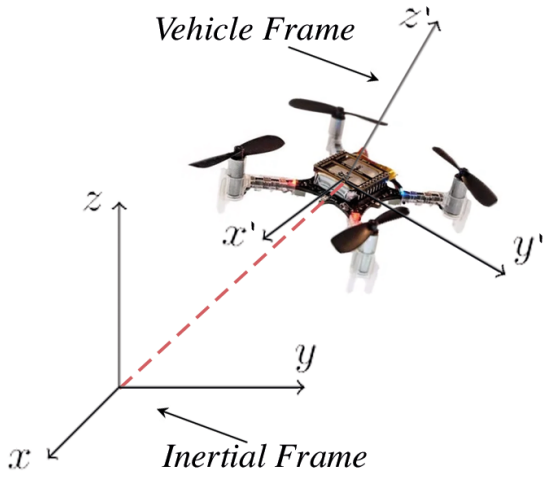


Figure 2: UAV State estimation

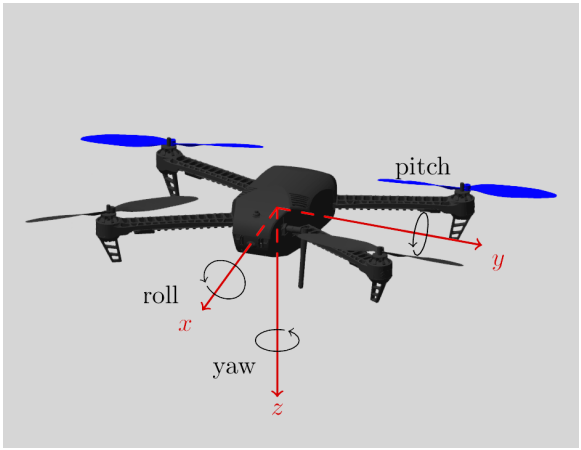


Figure 3: Generic quad-copter with coordinate overlay

The multi-rotor helicopter type, also referred to as multicopters [63, 125], is highly adaptable and only needs a limited space to take off and land thanks to its Vertical Takeoff and Landing (VTOL) ability. Therefore, this category of UAVs is one of the most widely researched.

*UAV Firmware.* Support for UAV developers has steadily increased and matured over the years with open access projects for the software (i.e., firmware) and hardware (e.g., flight controller and sensors). Well-known examples are Ardupilot [11] and PX4 [117] (autopilot software systems) and Pixhawk [134] (open standards for UAV hardware). The UAV's *firmware* is a software that is loaded into the drone micro-controller in order to provide the control on the UAV's hardware [60, 51].

*PX4* is a powerful open-source flight control software for UAV[118]. It is widely used both in academia and commercially as it provides a flexible toolset for UAV development and applications [116]. The first advantage of PX4, is that it provides a standardized API to interface with the UAV and Simulated environment. A second advantage is that it provides a low-latency, high-performance embedded solution for UAV control and interaction. Finally, PX4 can also be used as a Software In the Loop (SIL) tool together with the Robot Operating System (ROS) for small aerial vehicles, facilitating rapid development in a safe and simulated environment [118].

The firmware contains both the Flight Stack and the middleware. As depicted in Figure 4, the unprocessed signals from UAV's sensor are passed to the Drivers and sensor hub, which contain the protocols needed to read the data (e.g. $I^2C$, *UAVCAN*) and translate it into usable uORB messages. These messages are then passed to the flight control block, which contains the UAV state estimator. This estimation then propagates into three different components, which, based on the input from the ground control and the estimated state, compute the successive movements the UAV should perform. In particular, the Navigator is responsible for generating the position set-points, which the position controller transforms into attitude set-points[35]. Finally, the attitude and rate controller transform the set-points into control signals which the output driver can send to the actuators and servos the UAV possesses [35]. With the PX4 firmware, this cycle takes a couple of milliseconds from start to finish; the reaction time of the controlled UAV is, therefore, sufficiently quick for the UAV to respond to the input changes [141, 35].

Within PX4 there are three main components used in our prototype, the *state estimation*, the flight control and the motion control. The flight control is module responsible for the position and attitude control of the UAV, while the motion control is responsible for translating the output of the flight control into values which the UAV's actuators can use (i.e. Motors, Gimbal, Servos)[175, 35].

PX4 provides a well-documented API which enhances its ease of use in a SIL setting by developing it with simulations as a core capability of it [36]. In addition, this API enables the ability to offload the control inputs to be sent by another software instead of using other input devices, such as Joystick or gamepads [118, 36].

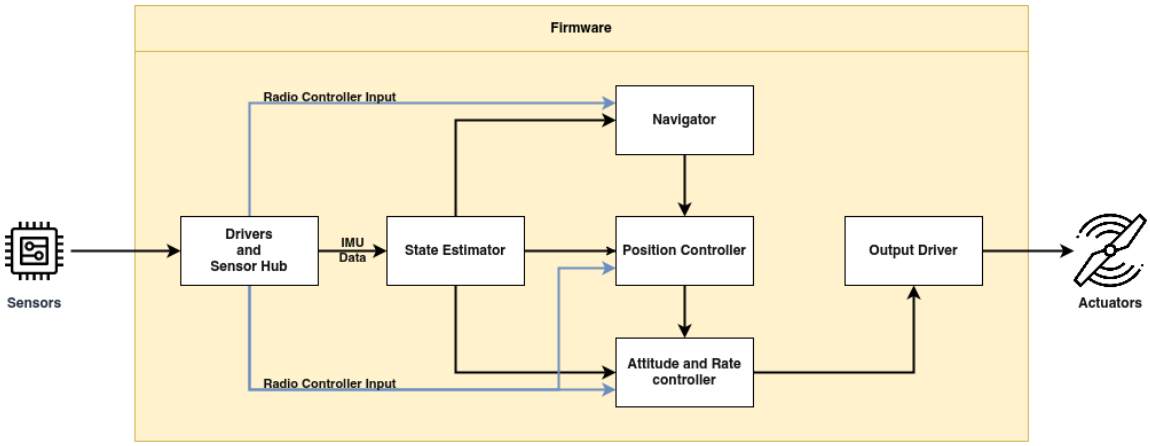


Figure 4: UAV Firmware components [35]

The API also opens up the possibility to design and implement the ground controller and test it on a proven test ground, which allows to test rapidly and prototype new firmware and control loops [118, 36].

*UAV Simulation. Gazebo* is an open-source software that provides a dynamic model of UAV, sensor model and 3D visualisation [93]. It was developed as a response to the increased use of outdoor robotics, and the need to simulate outdoor environments with realistic sensor feedback [93]. The architecture of Gazebo is comprised of three overarching components. The first component is the environment which models environmental factors such as gravity, or lighting [93]. The second components are the models, which are all physical objects which have at least one body and are connected with joints; such elements could be obstacles or other entities placed in the environment [93]. The third and last component are the robots models, which are physical objects with the added capability of sensors. In this thesis robot models are the UAVs.

Gazebo provides a well-designed user interface that offers easy user interaction with the simulation, as it does provide a rendered scene of what is being simulated [93]. While this approach is catered towards a human supervision or interaction approach, the simulator also provides an external interface that can be leveraged by software such as PX4 to send and receive data from the sensors or manipulate the environment. This lends the simulation to be run without the need of human interaction in combination with a third-party such as PX4 in a SITL setting in order to test and evaluate the UAV performance[35, 93, 57].

Over the past decade, many researchers have focused on improving multicopters' and UAVs' abilities by enhancing each of their components and implementing rigorous testing in simulation and real-world environments. Now, thanks to the increased availability of small and powerful computers and the increased capabilities of sensor technologies, UAVs can perceive small changes in the environment and react within a fraction of a second [54, 48]. Novel approaches to UAV flight controllers leverage these computational developments by removing the sequential hierarchy of the robotic cycle and instead use machine learning to integrate the cycle's phases into hybrid processes or end-to-end approaches[107, 51]. One of the grounding elements used to advance control systems rapidly is using computational models, and simulators in a Software In the Loop (SIL) configuration [26]. SIL are not only used to evaluate control algorithms but can be leveraged together with machine learning models to produce and rapidly optimise the control algorithms for the UAVs [64, 107]. This approach produces control systems capable of sensing and reacting within a hundredth of a second and out-compete human experts [107].

The reduced latency between the perception and the control phase, make the UAVs adapt faster to the environment and help the UAV moving in an unknown scenario safely and at high speeds [107]. The safety aspect of UAVs is critical and widely studied, as UAVs has fast spinning blades and can move at great speeds they can damage objects by crashing into them and hurt people [161]. The rapid development and deployment of new UAV systems able to perform tasks autonomously, has also lead to an increase to a rise in accidents in the wild.

In 2019 a UAV belonging to a pilot program of the Swiss Postal service had an uncontrolled crash near a group of children, launching an investigation into the accident and immediate pause on the pilot program [135]. The Swiss delivery program could only be restarted once all parties involved, including the Swiss Federal Office of Civil Aviation, would be satisfied with the mitigation that the UAV producer would apply to the system [135]. Another crash, which happened in 2017 in Japan, led to the injury of 6 people when a UAV crashed into a crowd, which it was hovering over as a demonstration of robotic technologies [79]. Such crashes lead to increased scrutiny by aviation authorities around the globe and the worsening of public opinion on UAV technologies.

### 2.1.2 Development of Self-driving cars

Among various and emerging CPS application domains, the usage of self-driving cars (SDCs) in transportation is expected to impact our society profoundly. Human errors cause more than 90% of driving accidents (e.g., driving while under the influence of alcohol, fatigue, and other distractions) [82]; hence, automated driving systems such as SDCs have the potential to reduce such errors and eliminate most accidents. However, the recent fatal crashes involving self-driving cars suggest that the advertised large-scale adoption of SDCs appears optimistic and premature [12, 68]. One of the main factors limiting the usage of autonomous driving solutions is the lack of adequate testing. Consequently, the risk of releasing SDCs equipped with defective software, which might become erratic and lead to fatal crashes, is still quite high [68].

Testing automation is crucial for ensuring the safety and reliability of SDCs [82, 90]. However, most developers rely on human-written test cases (at unit and system levels) to assess SDCs' behavior. This practice has several limitations and drawbacks: (i) limited possibility to repeat tests under the same conditions [90]; (ii) difficulty in testing SDCs in representative and safety-critical scenarios [68, 159, 77]; (iii) difficulty in assessing SDC's behavior in different environments and execution conditions [82].

As a consequence, SDCs practitioners in the field are facing a fundamental development challenge: *observability, testability, and predictability of the behavior of SDCs are highly limited* [68, 159, 77]. Thus, new testing practices and tools are needed to find SDC faults earlier during development and, eventually, support the widespread usage of autonomous driving.

The utilization of simulation environments can potentially address several of the challenges mentioned above [16, 28, 43, 3] since simulation-based testing is more efficient (i.e., have smaller testing costs) than and can be as effective as traditional field operational testing [7, 43]. Additionally, simulation-based testing can support and complement well-established hardware-in-the-loop (HiL), model-in-the-loop (MiL), and software-in-the-loop (SiL) development strategies. Consequently, an increasingly large number of commercial and open-source simulation environments have been delivered to the market to conduct testing in the autonomous driving domain [43, 16] as well as other CPS domains [149].

**Levels of Automation.** Yoganandhan et al. [180] propose five levels of automation:

1. *Driver Support*

   The driver of the vehicle gets support from some intelligent systems. E.g., the driver might get a notification that the car in front is too close. Another example is the parking of the vehicle. The driver can get visual and/or audio signals if the car is too close to a wall or another parked car.

2. *Limited Cruise Control*

   The vehicle is able to steer itself to a certain degree without a human-based interaction. E.g., the automatic lane-keeping system of a car that lets the vehicle cruise along a simple lane without manual steering of the driver.

3. *Conditional Automation*

   On this level the vehicle only provides an interface for human. The idea is that the driver is only interacting with the vehicle if it is really needed.

4. *High-level Automation*

   Compared to the third level, this level provides no human interface for steering the vehicle. This is more abstract than the following, which targets the goal of autonomous driving.

5. *Fully Autonomous Vehicle*

   The vehicle is fully autonomous and no human is required to interact with it. It also has no interface for it.

In Figure 5 the lidar and radar sensors provide data input for the autonomous driving system as described in Figure 6. These sensors are mainly meant to identify the distance to other objects, i.e., other cars. Furthermore, camera sensors give important data for the lane-keeping system since the lanes must be identified visually.

Data is processed in order to give the actuator of the system the right inputs. The output of the pipeline are the actions that the vehicle has to make. E.g., steering, throttle, light-signalling, etc.

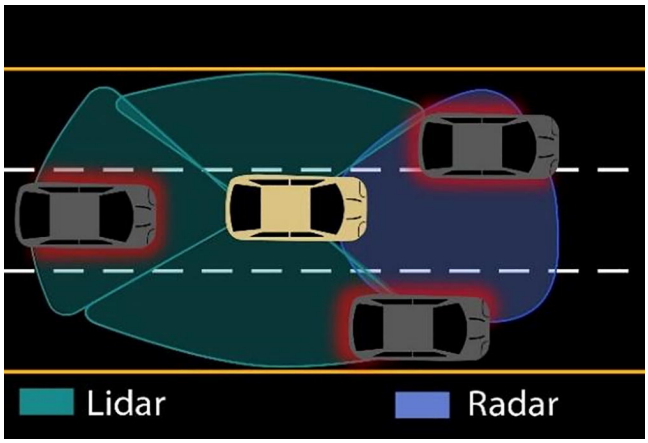


Figure 5: Lidar and radar as sensors for providing input the AI that steers the car [180].

## 2.2 Monitoring of Cyber-physical Systems States

Emerging CPS systems are characterized by an evolving *development that never ends* [165], and practitioners in the the field are facing a fundamental development challenge: *observability, testability, and predictability of the behavior of emerging CPS is highly limited and, unfortunately, their usage in the real world can lead to fatal crashes sometimes tragically involving also humans* [68, 159, 77]. ***DevOps practices and tools are potentially the right solution to this problem, but they are not developed to be applied in CPS domains*** [81]. For instance, in emerging safety-critical, dependable CPS—e.g., self-driving cars—requirement engineering for *DevOps has been largely unaddressed in the literature* [39].

In this context, CPS behavior tends to be unpredictable—its behavior is partially specified by humans, and partially learned, thus CPS may react differently to the same inputs over time—and it can lead to emergent situations [72]. Novel approaches need to be designed to enable a flexible, intelligent, and context-aware CPS behavior and states monitoring, to meet the required level of trustworthiness.

To maximize the overall quality of CPS software, an important requirement is to increase the ability of monitoring CPS behaviors and states at run-time as well as statically monitoring QAs of CPSs via DevOps technologies during the CPSs evolution.

Areas where researchers contributed to facilitate this tasks are the run-time monitoring [15] and simulation-based approaches/frameworks [145, 66, 103]. However, there is a lack of effective maintenance and evolution techniques able to trigger *static analysis*, and *monitoring* considering both HiL/simulation activities, in the field and operational data/inputs of CPS, to assess the system behavior.

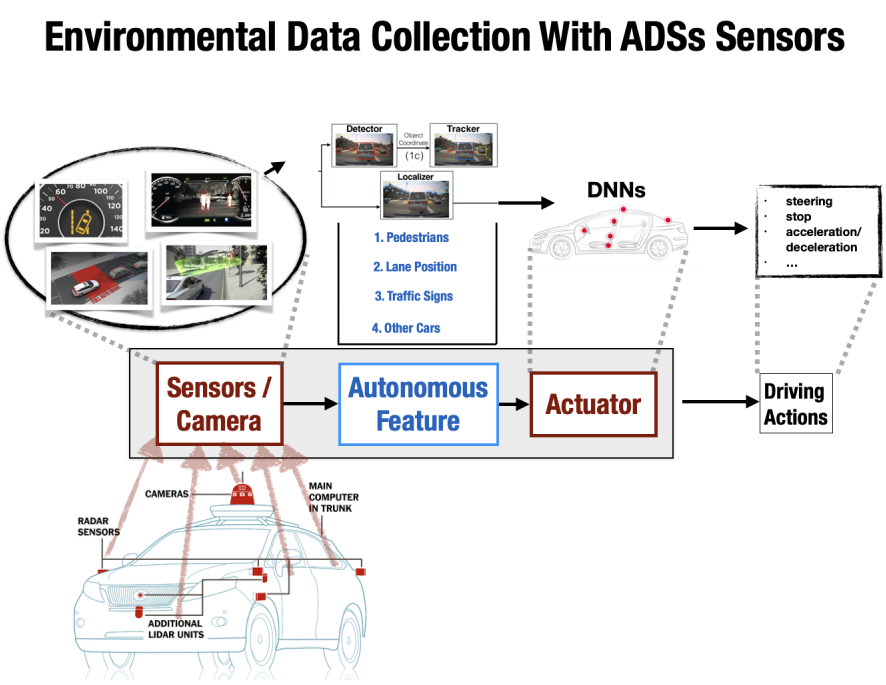


Figure 6: Autonomous driving system pipeline.

COSMOS targets to extend existing monitoring mechanisms to detect different forms of degradations by integrating new metrics and tools in DevOps pipelines, based on CPS-specific concepts of test effectiveness, anti-patterns, that we extend to handle HiL and SiL aspects. To automate the (micro-)fixing of detected/predicted issues, we will design tools based on meta-heuristics, historical analysis (e.g., of failures and previous fixes [182]), and data analytics.

### 2.2.1 Monitoring UAV and SDC States

Automated monitoring and testing of UAVs (and in general, robots and CPS) to ensure their proper behaviour represents still an open research challenge [7]. Current solutions are still very limited for autonomous systems such as UAVs, since testing such systems is substantially complicated by the need of these systems to continuously collect and analyse real-time data to make runtime decisions [164, 106], which makes it more difficult to determine whether these systems behave as *expected* [106] for the certification process [34, 75].

Indeed, supposed one or multiple physical variable(s) are out of their expected range for some time during a scenario. In that case, the autonomous system can enter in an *unexpected behaviour or state* [155] that can be perceived as erratic and can be potentially harmful to humans (e.g., physical instability, rotating or hovering in place for too long, terminating the mission immaturely, miss-calculation of positions, etc.). For dealing with such safety-related challenges, there is an increasing interest in adopting agile development paradigms within the CPS safety-critical domains [81, 164], in which the identification of hazards and the elicitation of safety requirements can be performed iteratively [38]. This has pushed researchers proposing the usage of Digital-Twins[1] technologies to simulate and test CPSs in a diversified set of scenarios [74, 27, 132, 22, 124] to support testing automation [8, 178], and regression testing [88, 25], and debugging [153, 143] activities. So, Simulation-based testing has been suggested as a promising direction to improve the UAV testing practices [6, 164, 169].

Liang et al. [105] conducted an empirical study with the goal of understanding safety-critical concerns of UAV software development. They investigated the use of bounding functions (runtime checks on the range of

[1] A digital twin is a virtual representation of a real-time digital counterpart of a physical object or process

variables) that were injected into the code by the developers of Paparazzi [133], an open-source UAV software system. Their study revealed that a large number of such functions are linked to safety-critical UAV variables and categorised them into five logical groups: *trajectory, sensor, speed and acceleration, engine, and pose management*. They argue that their bottom-up approach for extracting safety-critical physical variables, which starts with extracting the ones that developers put special constraints on, can ultimately lead to defining what *safety-critical* actually means for such systems, which is still an open challenge.

Wang et al. [169] studied UAV software bugs (reported as GitHub issues) for two popular open-source UAV Autopilot platforms (i.e., PX4 [117] and Ardupilot [11] ). They created a taxonomy of UAV-specific bugs and identified their root causes (detailed in deliverable D.6.1). They also identified some challenges in detecting and fixing UAV-specific bugs, including bug reproduction which is very hard in UAV domain considering the non-deterministic nature of the physical environment around the UAVs. They report that developers mainly use simulators to reproduce the bugs, but setting up realistic-enough simulation environments (that captures the same bugs as physical tests) is hard and expensive.

Mithra, an oracle learning approach proposed by Afzal [5] for testing drones, identifies patterns of normal and common behaviours of the system in many executions in simulation. First, it clusters drone telemetry logs to create a model for the normal drone behaviour in specific scenarios and form a test oracle. Next, in the testing phase, the same telemetry logs for the test flight are extracted and compared with the normal clusters in the oracle, and the test result (fail or pass) is decided based on its similarity to one of the clusters.

Lindval et al. [106] developed a framework for automated testing of autonomous drones in simulation with the aim to solve the test oracle definition problem. First, they define various metamorphic relations to be able to extract a success model for a test scenario. Next, they do model-based test case generation based on the extracted model to automatically create diversified test scenarios that should result in similar outcomes to the model according to the metamorphic relations.

PHYS-FUZZ [177] is a fuzzing approach explicitly tailored for testing mobile robots, taking into account the physical attributes and hazards of such robots. PHYS-FUZZ can help the robot developers find failing test scenarios (inputs and physical environment configurations) faster than traditional approaches.

Afzal et al. [7] studied the challenges of testing robotic systems and reported *logging and playback* and *simulation testing* as the most used testing practices by robot developers. They also recognise *engineering complexity* of the test environment, including the design of the realistic inputs to test the system, as one of the biggest challenges in this domain.

To understand the current capabilities and limitations of simulated testing for robotic systems, Afzal et al. [6] surveyed robotic practitioners about how they use simulators for testing and the challenges they face.

To study the effectiveness of simulation-based testing of drones, Timperly et al. [164] conducted an empirical study on fixed bugs in Ardupilot [11]. They investigate whether the same bugs could have been detected before field tests if proper simulation-based testing approaches were in place. They characterise the types of bugs that are capable of being discovered in simulation, and argue that the majority of the bugs can fall into such category by demonstrating the procedure to detect them in simulation for some specific case-study bugs.

## 2.3 Flaky Tests Analysis and Identification

Testing of cyber-physical systems (CPS) is getting more and more attention to developers and test engineers in recent years. There are several accidents with severe casualties in the past like the case with *Boeing 737 Max* with two plane crashes [158]. In the case of self-driving cars (SDC) as CPS there were also several deadly incidents such as with cars of Tesla [122, 171]. Also, the use of drones leads to incidents, such as the post delivery drone of the Swiss Post service in Zurich leading to two crashes [121, 146]. Testing in the context of CPS should take an important role since CPS can deadly harm human beings when they are not properly tested to ensure safety. Ideally, the tests for life-critical CPS should ensure trust in the system but they should also be well integrated into the development process without increasing the costs too much.

A flaky tests are tests that passes and fails periodically or in non-deterministic way, without any system changes (or for the same test inputs) [99, 17, 100]. Flaky tests introduce for developers several disadvantages for the whole DevOps pipeline. A continuous integration and delivery pipeline might fail to bring the changes to production due to failed builds caused by failing tests that are flaky [99, 17, 100]. In such a pipeline a build of the software with flaky tests will pass or fail without changing any code related to the test or system under test. Developers would therefore spend a lot of time trying to find a bug in the system even there is none. Thus, we see that flaky tests introduce more overhead in the development life-cycle of a software system. We need to address the flaky aspects in our test suites to ensure more reliability and quality of the software.

Several studies, related to flaky tests for traditional systems, were done but not yet for testing CPS in simulation [130, 47, 101, 183, 148, 126, 17, 142, 151, 44, 147, 162, 102, 111]. A couple of methodologies and tools were developed that try to predict the likelihood of a test case being flaky based on certain metrics (e.g., differential coverage [17]). These tools have shown, that they outperform the traditional *Rerun* approach, which simply reruns failing tests to verify if the outcomes change and therefore being flaky. In the case of testing CPS in virtual environments, we can not rely on metrics like code coverage. Therefore, we need to find other metrics based on the virtual environment itself to understand the flaky behavior of simulation-based testing because this kind of testing is getting more and more attention in the field of CPS.

In the case of system testing of CPS, simulation-based testing is a way to reduce costs while maintaining almost the same level of reliability of the system. In the use case of SDCs previous work was done for test selection, prioritization and the assessment of the cost-effectiveness of simulation-based testing of SDCs [89, 24, 23]. The goal of test selection is to select only relevant test cases that are likely to fail, whereas test prioritization defines the execution order of the selected tests so that defects are detected earlier in the testing process. Regression testing with test case selection and prioritization in CPS can lower the testing costs while keeping the system safe and reliable. However, previous work did not consider so far how flaky tests should be treated during regression testing with CPS.

CPS should rely on proper tests that are not flaky since flaky tests are nondeterministic and therefore not meaningful for verification and validation of CPS. Past studies have shown that flaky tests in traditional software systems have several different root causes [47, 111]. Such root causes are classified in different categories such as *Concurrency*, *Resource Leak*, *Async Wait*, *Test Order Dependency*, *Float Precision*, *Time*, *Randomness*, etc. in the test code itself or in the coder under test. In summary, the tests of CPS should be deterministic and avoid a nondeterministic behavior that might result from various root causes of the test itself or from the CPS.

## 2.4 Monitoring and Propagation of CPS Changes

Scaling static and change analysis of software and the inclusion of multiple programming languages are representing an ongoing effort [9].

**Multi-revision analysis**. Past research was dedicated to analyzing historic data in form of revisions of software projects [78, 19, 112]. Specifically, previous studies track revisions and bug reports for a software project in a common database [52] and provided flexible infrastructures for common logistical issues, to facilitate software evolution research Bevan et al. [20]. Complementary, Gall et al. [58, 53] proposed ChangeDistiller, a change analysis tool that compares abstract syntax tree (ASTs) of two different versions of a software system. Complementary, Zimmermann et al. [184] proposed approaches for predicting the location of further changes. To enable the large-scale analysis of repository data, the Boa infrastructure was proposed [45, 46], which provides access to the ASTs of the code of Java releases of projects. Le et al. [104] present a technique to merge control-flow graphs of multiple versions of a software system. Similarly, TypeChef [83] analyzes multiple *configurations* of C programs (i.e., `#ifdefs`).

**Multi-language analysis** Given the multi-language nature of software projects, research approaches for multi-language program analysis have been proposed [41, 10, 96]. Specifically, Tichelaar et al. [163] proposed a source code metamodel support refactoring of source code from different languages. Then, Strein et al.

have developed multi-language approach capturing the relationships in source code [156]. Differently, Rakić et al. [139] proposed a language-independent framework for software analysis [139]. Complementary, recent work proposed further different approaches focused on multi-language anaysis of programs for the detection of software vulnerabilities [69].

Finally, researchers investigated the characteristics of fair code reviews [61, 95, 29] as well as the defects developers typically fix during code reviews [113]. A very close work to work reported in this deliverable are the one of Beller *et al.* [18] and Panichella *et al.* [129] where the authors manually classified changes taking place in modern code review of OSS projects and desired a taxonomy of software changes. Our deliverable reports an extended change taxonomy of CPSs, which is more fine-grained compared to the one proposed in such previous work. It is important to mention that this deliverable describes a change analysis tools developed for analyzing and potentially classifying CPS change types based on the analysis of code and CPS operational data between revisions.

## 2.5 Monitoring and Propagation of Vulnerabilities and Security Flaws

Security testing activities concern both security functional testing, i.e., validating whether the specified security properties are implemented correctly, and security vulnerability testing, i.e., addressing the detection of system vulnerabilities [49]. In this deliverable we focus on security vulnerability testing.

Software vulnerabilities have been extensively investigated in prior studies. Most of these related work focused on (i) identifying types of security flaws that could expose users to risks, and (ii) proposed tools for detecting such flaws [154]. As result, a non-exhaustive list of vulnerabilities were studied and concern (i) inter-application communication vulnerabilities [154, 71, 50], (ii) hijacking of vulnerabilities [108], (iii) security risks related browsing [179], and other major types of privacy policy violations [152].

Jimenez et al. [80] studied vulnerabilities reported in the National Vulnerability Database (NVD) and characterized the corresponding fixes. Linares Vásquez et al. [168] conducted an empirical study characterizing the types of Android-related vulnerabilities, the survivability of vulnerabilities, as the number of days between the vulnerability introduction and its fixing. Similarly, Thomas et al. [160] explored API vulnerabilities and quantified the fixing rate on real devices. More recently, Watanabe et al. [172] used automated vulnerability scanners to conduct a large-scale study and discovered that more than 50% of vulnerabilities of apps stem from software libraries, particularly from third-party libraries.

Unfortunately, security vulnerability testing solutions for CPS are at their infancy, with most of the work focusing on power grids and attacks against system integrity [123, 4, 110]. Strategies to drive security analysis and testing of CPS based on security requirements and threat models had been proposed in the literature [140, 56, 86]; however, methods to automate the generation of security tests based on security requirements and machine learning models are missing. COSMOS aims at providing tools for detecting security vulnerabilities in different CPS application domains, focusing on the domains affecting/concerning the CPS behavior.

# 3 Implementation

## 3.1 Architecture of COSMOS Component for Quality Assessment and Monitoring of CPS

Figure 7 visualizes the main modules and automation goals of the COSMOS component for quality assessment and monitoring of CPS. Specifically, as can be observed from the figure, from an high-level of abstraction, COSMOS focuses on the following main high- and low-level challenges:

- To allow a DevOps pipeline to monitor a set of CPS relevant Quality Attributes (QAs):

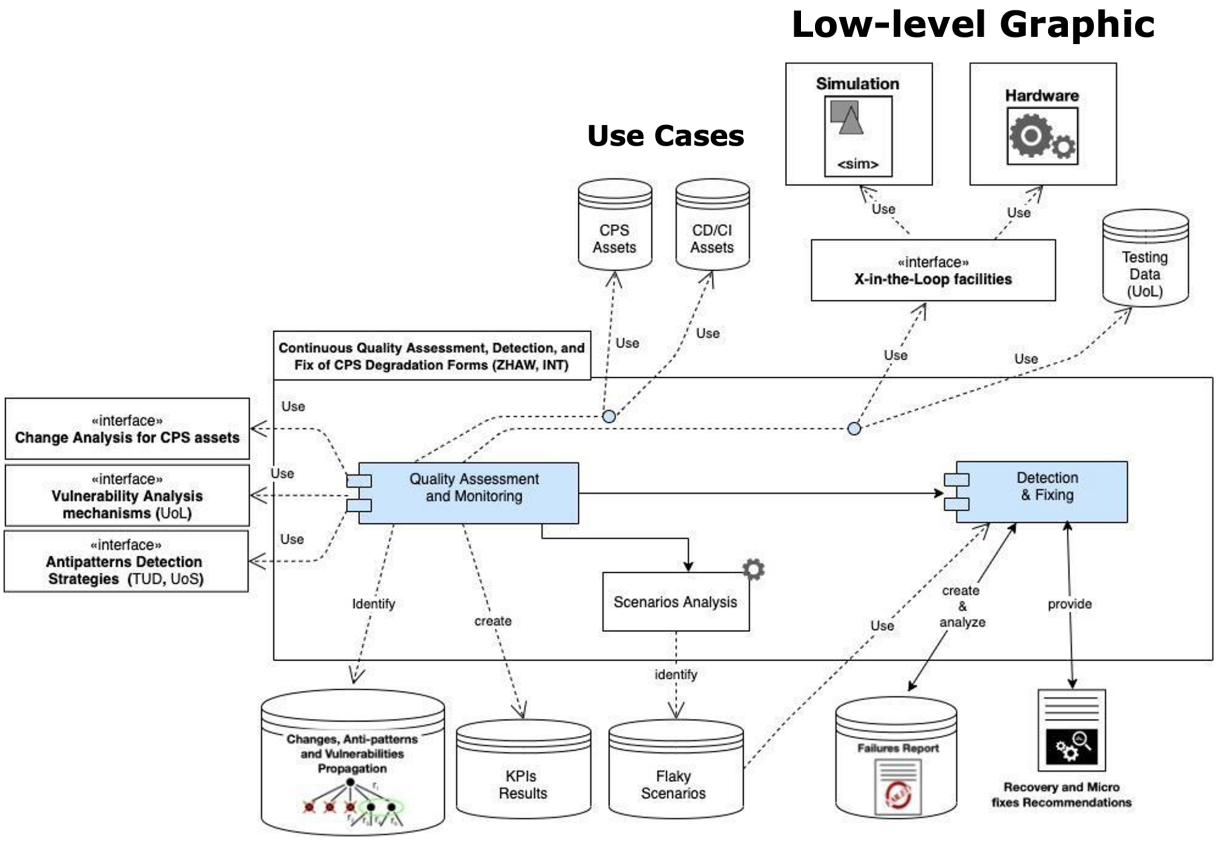


Figure 7: Overview of COSMOS Component for Quality Assessment and Monitoring of CPS

1. Monitoring CPS States
2. Monitoring Change Propagation
3. Monitoring of security vulnerabilities of CPS
4. Monitoring of Antipatterns

- To mitigate CPS degradation forms:
  1. Detection of flakiness issues in X-in-the-loop testing
  2. Prediction of CPS degradation patterns
  3. Design of recovery solutions based on (micro-)fixes.
- Monitoring of Key performance indicators (KPIs) related to key business and development goals.

From a technological perspective, in the context of monitoring CPS States and flaky tests, COSMOS focuses on addressing the realization of solutions enabling the monitoring or identification of (i) of CPS states with X-in-the-loop Facilities, (ii) of Flaky Scenarios / tests for CPS with X-in-the-loop Facilities. Moreover, COSMOS focuses on addressing the realization of mitigation strategies focusing on: (i) the prediction and monitoring of Flaky Scenarios / Tests of CPS, to mitigate the risks of encountering unexpected behaviors while improving the general quality of CPS tests; (ii) support fixing of Flaky Scenarios / tests of CPS. Finally,

In the context of monitoring and propagation of CPS changes and vulnerabilities, COSMOS focuses on addressing the realization of solutions enabling the (i) CPS Change Analysis and Propagation (in Open-source use cases and COSMOS use cases); (ii) the CPS vulnerability analysis and propagation (in Open-source use cases and COSMOS use cases), with specific focus on identifying static vulnerabilities to use for vulnerability proneness analysis (and potentially the vulnerability propagation) in CPSs.

The following sections elaborate on the status of the current development and the next steps of the COSMOS development activities.

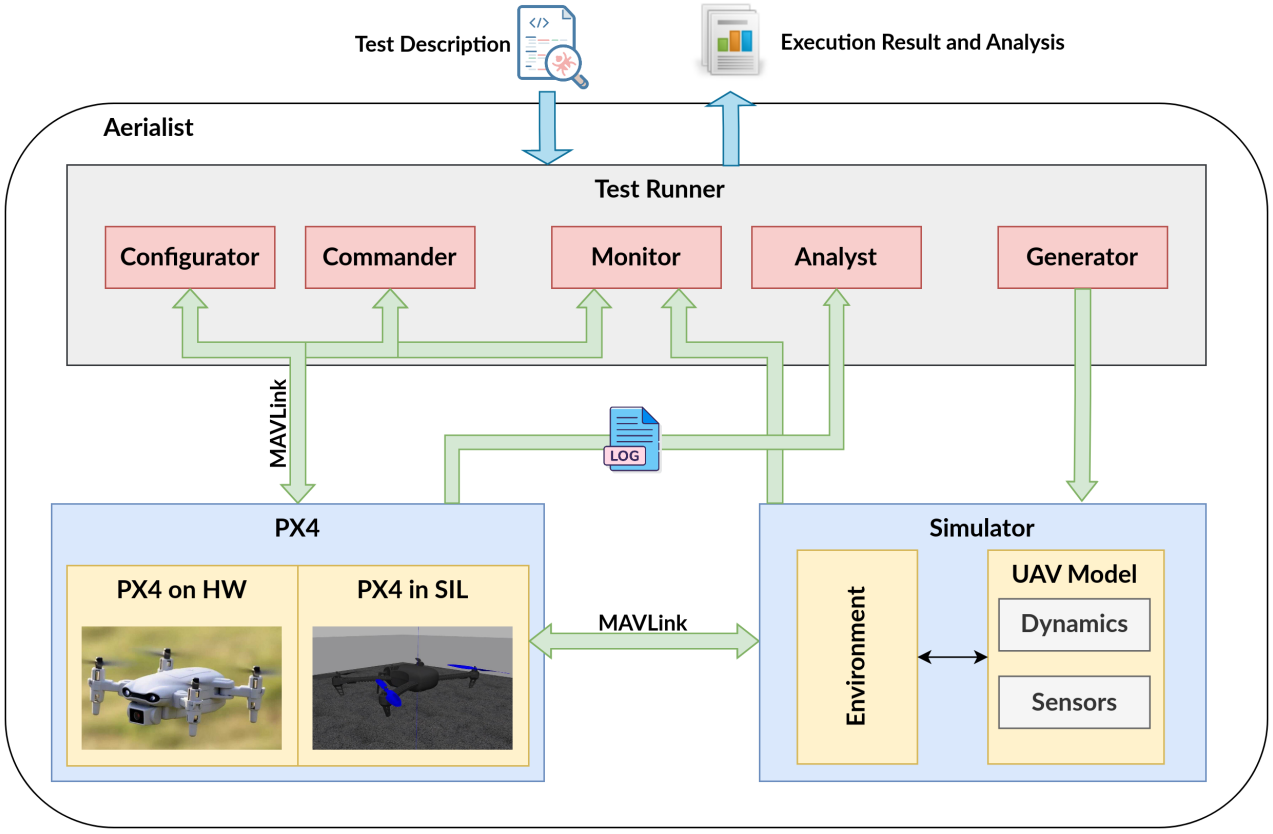


Figure 8: High level architecture of AERIALIST

## 3.2 Prototype for Monitoring UAV States

### 3.2.1 UAV Test Bench

To be able to monitor UAV states at runtime, we developed a modular and extensible test bench for UAV software called AERIALIST (Autonomous aERIAL vehIcle teST-bench). Figure 8 demonstrates the overall architecture of AERIALIST. The implementation [2] currently supports PX4 platform, but can be extended to support other UAV platforms as well.

The input is the test case description, which can be in a config file, set in specific environment variables, or provided directly to the Command Line Interface (CLI) as parameters. AERIALIST prepares the environment for running the test in the *Test Runner* subsystem, which abstracts any dependencies to the actual UAV, its software platform, specific flight modes and simulation environment. After setting up the simulation environment as described in the test description (if testing a simulated UAV), *Test Runner* connects to the drone (simulated or physical) and configures it as instructed at startup, and stats sending runtime commands. It also monitors UAV state during the flight, and extracts the flight logs at the end of the test for future analysis. We will describe each module in more details in the following sections.

**PX4 Platform** PX4 [117] is an open source flight control platform used to implement a UAV system. PX4 supports Software In-the-Loop (SIL) simulation to safely execute UAV flights in simulation environments, with the purpose of checking novel control algorithms before actually flying the UAV, limiting the risk of damaging the vehicle. It also supports Hardware In-the-Loop (HIL) simulation, by providing simulation inputs to a firmware deployed on a real flight controller board.

[2] `https://cosmos-devops.cloudlab.zhaw.ch/cosmos-devops/cosmos-tools/UAV-TestBench`

*PX4 Simulation Environments.* Simulators allow PX4 to control a modeled vehicle in a *simulated world*. Hence, PX4 communicates with a simulator (e.g., Gazebo [94]) to receive sensor data from the simulated world and send actuator control commands back. In this setting, PX4, similarly to a real vehicle, can interact with the simulated vehicle using a ground control station (GCS), an offboard API (e.g. ROS), or a radio controller/gamepad, both to send telemetry from the simulated environment and to receive commands. PX4 supports several HIL and SIL simulators [36]. In the context of our work, we considered Gazebo [94] as PX4's reference 3D simulation environment since it is particularly suitable for testing UAV's obstacle avoidance and computer vision functionalities.

*PX4 Flight Logs.* PX4 logs any message communicated between remote control (RC) and UAVs, or between its internal modules [2]. This includes the sensor outputs, location, other estimations based on sensor readings, the commands sent to the UAV, and the errors/warnings from the internal modules. Logs are stored, after each flight, on the UAV file system, and can be analysed later to investigate issues (and their root causes) happened during a flight [2].

The uLog file, abbreviated from Unified Log, is the standard file format for flight registers generated by PX4 [118]. It contains all Micro Object Request Broker (uORB) messages used for inter-thread and inter-process communication. These messages are used in the asynchronous communication in the robotic cycle following the publish/subscribe pattern [174]. This pattern relies on a message broker called uORB, which relays all messages from the publisher (e.g IMU) to the subscriber (e.g. EKF). Each message is published by a UAV's sensors or one of the various state estimators; each message contains a relative timestamp since the UAVs boot-up accompanied by the publisher's measurements. In order to facilitate the analysis and usage of this file format, there exist libraries that allow parsing the uLog in order to extract the data, for example, *pyulog* for Python or *ulogreader* for Javascript [118].

*PX4 Flight Modes.* Flight modes define how the autopilot responds to RC input, and how it manages the vehicle movements during fully autonomous flights. Flight modes provide different levels of autopilot assistance, ranging from automation of common tasks, execution of a pre-computed trajectory, takeoff and landing, to mechanisms that make it easier to regain (or hold), when needed, a certain altitude level or position. Flight modes can be divided into *manual* and *autonomous* modes. Manual modes allow the user to control the vehicle movement via the RC sticks, while autonomous modes are fully controlled by the autopilot, with no pilot/RC input. Table 1 summarises PX4 flight modes.

*PX4 Parameters.* Within PX4 the behaviour of a UAV can be configured and tuned using *parameters* [167]. These parameters allow for a fine-tuning of the UAV calibration values, and flight behaviour such as maximum speed, jerk, or acceleration, and safety configurations such as maximum time without ground control connection and maximum distance from ground control. Each parameter has a defined type and range of values it accepts with a maximum step size, if there is any. In addition, PX4 allows for non-critical parameters to be changed during flight, which is reflected in the uLog file. All parameters of the UAV are available in the uLog file together with the value of it and the timestamp at which the parameter was set.

*PX4 Integration.* There are a range of methods for developers to develop on top of PX4 platform and extend its functionalities or interact with the drones.

*MAVLink* [97] is a very lightweight messaging protocol for communicating with drones (and between onboard drone components). MAVLink follows a modern hybrid publish-subscribe and point-to-point design pattern: Data streams are sent / published as topics while configuration sub-protocols such as the mission protocol or parameter protocol are point-to-point with retransmission. Messages are defined within XML files. Each XML file defines the message set supported by a particular MAVLink system, also referred to as a "dialect". The reference message set that is implemented by most ground control stations and autopilots is defined in common.xml (most dialects build on top of this definition).

*MAVSDK* is a collection of libraries for various programming languages to interface with MAVLink systems such as drones, cameras or ground systems. The libraries provides a simple API for managing one or more vehicles, providing programmatic access to vehicle information and telemetry, and control over missions, movement and other operations. The libraries can be used onboard a drone on a companion computer or on the

Table 1: Flight modes in PX4 [118]

| Flight mode | Description |
|---|---|
| Position | The multi-copter responds to the control input<br>holds the position and altitude when no new input is given<br>does counteract existing inertia or wind |
| Altitude | The multi-copter responds to the control input<br>holds the altitude when no new input is given<br>does not counteract existing inertia or wind |
| Manual | The multi-copter responds to the control input<br>does not hold the position when no new input is given |
| Mission | Vehicle executes a predefined autonomous mission (flight plan)<br>that has been uploaded to the flight controller |
| Return | Vehicle flies a clear path to a safe location<br>often used in an event of a triggered failsafe event (e.g. RC connection lost) |
| Takeoff | Vehicle ascends to takeoff altitude and holds the position |
| Land | Vehicle descends and lands at the position where it was engaged |
| Hold/Hover | Vehicle hovers at the current GPS and altitude position |
| Return | Vehicle flies a clear path to to the position dictated by the parameter settings |

ground for a ground station or mobile device. MAVSDK is cross-platform: Linux, macOS, Windows, Android and iOS.

*ROS* (Robot Operating System) is a general purpose robotics library that can be used with PX4 for drone application development. ROS benefits from an active ecosystem of developers solving common robotics problems, and access to other software libraries written for Linux. It has been used, for example, as part of the PX4 computer vision solutions [137], including obstacle avoidance and collision prevention.

**Test Runner** To evaluate a test definition, we generate and execute the corresponding simulated test case automatically. The test case automates all necessary steps: setting up the test environment, building/running the firmware code, running/configuring the simulator with the simulated world properties, connecting the simulated UAV to the firmware, and applying the UAV configurations from the test case properties at startup. Then, the test case commands are scheduled and sent to the UAV, the flight is monitored for any issues, and after test completion, the flight log file is extracted.

*Test Description.* The de-facto testing standard of UAVs relies on *manually-written system-level tests* to test UAVs *in the field*. These tests are defined as specific software *configurations* (using parameters, config files, etc.), in a specific *environment* setup (e.g., obstacles placement, lightning conditions), and a set of runtime *commands*. Such runtime commands received during the UAV flight (from RC, GCS, onboard computers, etc.), make the UAV fly with a specific human observable *behavior* (e.g., flight trajectory, speed, distance to obstacles). We model a UAV test case as a set of *test properties* (e.g., ⟨*configuration, environment, commands*⟩) that control the flight and a set of pre-defined UAV *expected states* (e.g., ⟨*trajectory positions*⟩) during the flight. More specifically we define a test case by the following properties, which are fed to AERIALIST as an input file, or command arguments, and refer to them as *test description*:

- *Configs:* Drone configuration at startup (all parameter values and configuration files required to start the simulation).
- *Commands*: Timestamped external commands from the ground station or the remote controller to the drone during the flight (e.g., change flight mode, go in a specific direction, enter mission mode).
- *Environment* (optional): Simulated world's configurations (e.g., used simulator, obstacles' position and shape, wind speed and direction).
- *Expectation* (optional): time series of certain sensor reading that the test flights are expected to follow closely.

*Generator.* The Generator module deals with setting up the simulated world before testing UAVs in SIL mode. It sets up and prepares the simulation environment as described in the test description, in a specific simulator (e.g., Gazebo, jMAVSim), along with the described static and dynamic objects and simulated UAV.

*Configurator.* module is responsible for setting up and initialising the UAV under test (either simulated or real) before flying the UAV, according to the instructions in the test description. This includes building the code, connecting to the drone via MAVLink, setting the parameters, uploading any needed resources, etc.

*Commander.* module is responsible for all the runtime communications to the UAV, including scheduling and sending the Remote Control (RC) commands (e.g., manual sticks, flight mode changes, arm/disarm), communications from Ground Controll Station (GCS) or the offboard commands coming from a companion computer.

*Monitor.* is the module responsible for runtime analysis of UAV state during the flight. Using MAVLink, we are able to subscribe to any messages communicated between PX4 modules, including sensor values. These messages allow monitoring any runtime checks described in the test description to evaluate monitoring functionalities before deploying on the UAV. The tested and finalised monitoring solutions can then be developed directly in the PX4 firmware as an on board module, or as a ROS module running on the companion computer.

*Analyst.* is responsible for any post-flight analysis, mostly based on the extracted flight log. It parses the ULog files, and extracts any important and relevant data to analyse test result based on the given expectations in the test description.

**Virtualization** Setting up all the requirements and dependencies of AERIALIST can be problematic. To simplify all these steps, we created Docker [1] images with all the necessary tools and dependencies installed and configured and ready to use. Docker is an open-source platform that facilitates the development, distribution and deployment of applications [138]. Docker provides a way to run software in a isolated environment from the host's operating system [76]. Each application is contained in a so-called **container**, which is a standard unit of software that also contains all the software dependencies, such as system tools and libraries and code [76].

All application instances are run isolated from each other and rely on the Docker Engine, which is run on the host OS. Each simulation contains its own instance of the core architectural elements such as Gazebo, PX4 and the test runner in order to launch and control the various components. Once the simulations conclude, the AERIALIST gathers all the flight logs from the containers and stores them on the host machine.

The UAV performance in simulation relies heavily on the processing power of the computer running AERIALIST. Also, due to the nature of the control mechanisms and the surrounding environment, the UAV behavior (both in simulation and in real world) can be non-deterministic. To eliminate such effects on the test outcomes, AERIALIST enables users to deploy the test runner containers in a Kubernetes [98] cluster, instead of running on their local machine. This enables a vast range of abilities, including setting specific resource limits and requirements, and running multiple simulations of the exact same tests in parallel, to eliminate outliers in the test results. For instance, one can run each test case $n$ times, extract the logs, and use the average of the recorded *states* for the analysis.

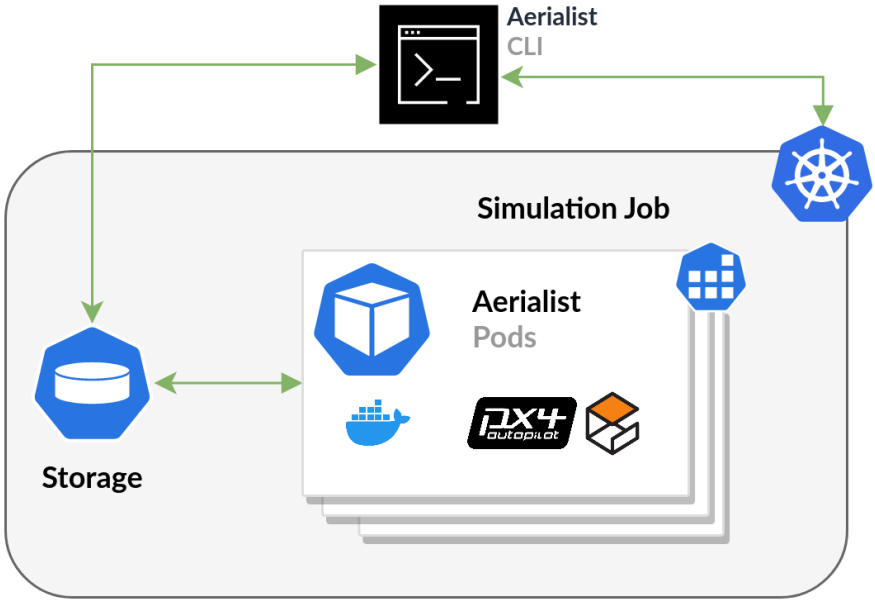


Figure 9: AERIALIST's Kubernetes deployment architecture

Figure 9 demonstrates AERIALIST's Docker and Kubernetes integration. Using the CLI, one can run a test case in a Kubernetes cluster, possibly for $n$ times in parallel. Tests (simulations) will be translated into a kubernetes Job, and executed inside isolated docker containers (with predefined and similar resource utilization, and PX4 and simulators already installed and wrapped in a Kubernetes Pod) and AERIALIST will wait until the mentioned number of parallel executions are finished. Possible errors in setting up the pods, and running the tests are handled automatically by Kubernetes engine, and the flight log is uploaded to a cloud storage after test executions. CLI will gather all the uploaded flight logs, and process them centrally afterwards.

**Using AERIALIST** AERIALIST can be used as a Python command-line utility. Since it needs proper setup and configurations of PX4 platform for execution, we prepared a Dockerfile to easily setup all the requirements as follows:

```
docker build . -t aerialist
docker run -it aerialist bash
```

This will open a bash terminal to the container with all the dependencies and requirements that directly supports the execution of following commands. As we detail below, AERIALIST's command-line supports the execution of various test scenarios described above by taking appropriate commands and inputs.

**Log Replay.** To replay a previously stored PX4 flight log ('.ulg') file, AERIALIST requires the following command:

```
./run.py --log /path/to/file.ulg experiment replay
```

**Manual Test.** To run an existing series of commands stored in a csv file, AERIALIST requires the following command:

```
./run.py --commands /path/to/file.csv experiment replay
```

**Mission Test.** To run an existing mission stored in a file, AERIALIST requires the following command:

```
./run.py --mission /path/to/file.plan experiment replay
```

**RC Test.** To enable manual RC inputs to the drone using keyboard inputs, AERIALIST requires the following command:

```
./run.py experiment manual
```

**Configurations.** AERIALIST supports various (optional) configuration parameters to combine with any of the above commands. The default values for each of them can be set by the corresponding environment variables, as documented in the repository.

```
--drone {sim,cf,ros,none}
                    type of the drone to connect to
--env {gazebo,jmavsim,avoidance}
                    the simulator environment to run
--params PARAMS     parameters file address to set at startup
--obst [OBST]       obstacle position and size to put in simulation world
--obst2 [OBST2]     obstacle position and size to put in simulation world


--headless          whether to run the simulator headless
--speed SPEED       the simulator speed relative to real time
--docker            whether to run the tests in docker containers
--k8s               whether to run the tests in Kubernetes cluster
-n N                no. of parallel runs (using Docker or K8S)

--cloud             whether to read and write files to the cloud
--output OUTPUT     cloud output path to copy logs
--trajectory TRAJECTORY
                    expected trajectory file address
```

### 3.2.2 UAV Runtime Monitoring

In this section, we discuss our methodology and approach for monitoring UAV behavior at runtime. We use the functionalities provided by AERIALIST for generating a big dataset of simulated flights, to be able to analyse the normal and common behavior of a UAV during flight. Then, we analyse the generated dataset for the common behavior, extracting the most relevant features (i.e., logged sensor values), and their expected range in a specific flight mode (i.e., landing). The extracted relations and ranges can then be developed and integrated into the monitoring module of AERIALIST, to enable automated monitoring of UAVs at runtime. We will discuss these steps in detail in the remaining of this section.

**Dataset Generation** To be able to study runtime behavior of UAVs, we first need a large enough dataset of logged flights to start the analysis. So, we developed a module called *Flight Generator*, that generates various randomized *test description* files, based on some predefined scenario templates. The generated files are then passed to AERIALIST for execution in the simulation environment, and the flight logs are extracted for future analysis. Table 2 describes the flight templates, and their randomized parameters we used for generating the dataset, while figure 10 demonstrates a sample flight trajectory for each of them. Flight generator picks random values for the parameters in a preset range, and we run the test descriptions using AERIALIST, each for 10 times in parallel to be able to also analyse the randomness involved in the platform and simulators. The resulting dataset is described in table 3

**Data Analysis** According to a survey on military UAV incidents [166], About 20% of UAV crashes occur during the landing phase, with an additional 8% occurring during the takeoff phase. Therefore, we decided to focus on monitoring UAVs in the landing phase as the first step towards a general monitoring solution for UAVs.

During our preliminary evaluation of our dataset, we noticed that over 75% of the simulated flights in our flight sets did not land successfully at first attempt, but instead bounced back up once or more times before

Table 2: Dataset scenarios description

| Scenario | Template | Parameters |
|---|---|---|
| Manual | Takeoff to height $h$<br>Hover in place for $t_h$ seconds<br>Fly in $(x_n, y_n)$ direction for $t_n$ seconds : $n$ times<br>Fly in $(z)$ direction for $t_z$ seconds<br>Land | $h, t_h, n, n \times (t, x, y), (t_z, z)$ |
| Autonomous | Setup a mission flight:<br>1) Takeoff to height $h$<br>2) Fly towards two way-points at $(x_n, y_n, z_n)$<br>3) Land<br>Place a $(s_x, s_y, s_z)$ sized obstacle at position $(o_x, o_y)$<br>Execute the mission | $h, 2 \times (x, y, z), (s_x, s_y, s_z), (o_x, o_y)$ |

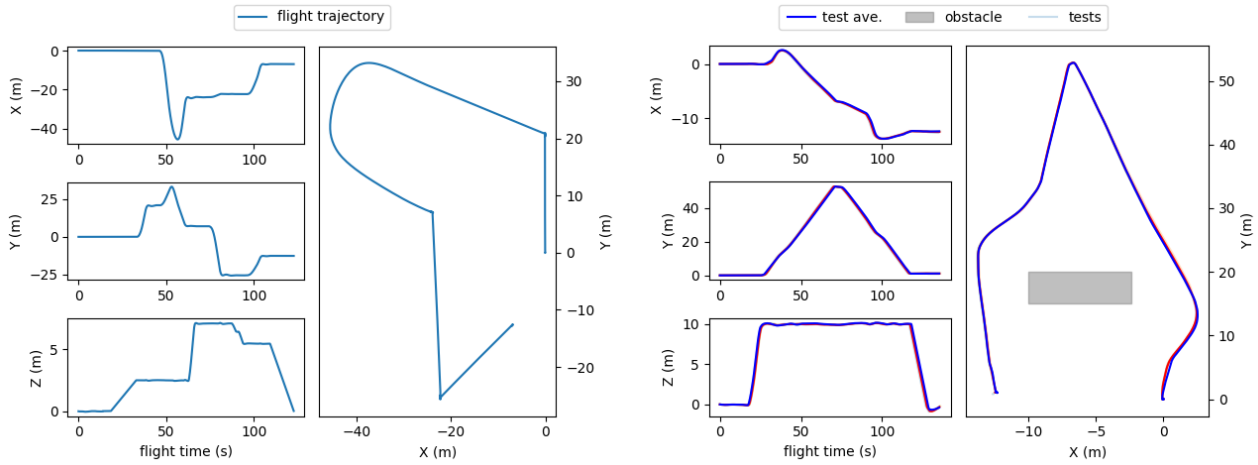


Figure 10: Sample flight trajectories for manual (left) and autonomous (right) flight templates

Table 3: Dataset statistics

| Dataset | tests # | exec. # | total # |
|---|---|---|---|
| Manual | 150 | 50 | 7500 |
| Autonomous | 150 | 10 | 1500 |

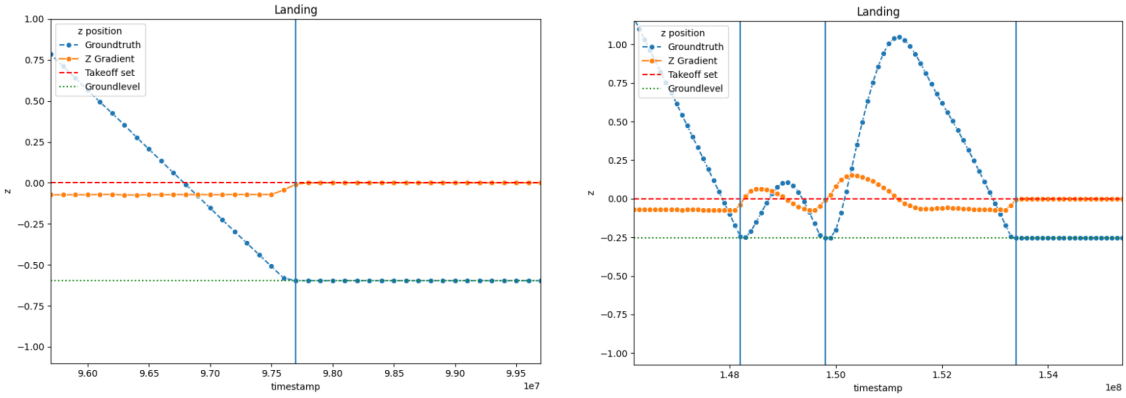


Figure 11: Landing phase of flights with an ideal (left) and bouncing (right) landing

their successful land (see figure 11). Furthermore, this landing hop sometimes reached more than 1 meter over the ground. This behaviour could pose risks to the UAV because if it has to collide with the ground too hard, it could damage the structural integrity of the UAV's frame and components. Furthermore, it also poses a risk to objects and persons around the landing area, as it performs quick manoeuvres to land again.

Hence, we focus our further analysis on defining the distinguishing characteristics of *ideal vs bouncing land*. We aim to come up with a monitoring solution that can observe UAV states during the landing phase, and detect if it is experiencing a bouncing one. To this end, we annotate our dataset with labels according to the flight landing behaviour, ideal or bouncing. We use the ground-truth values of the UAV's position stored in each of the flight logs to determine how many times the UAV contacted the ground. Figure 11 illustrates a comparison between misbehaving landing and non-misbehaving landing. The flight displayed in left side performs an ideal landing, while the flight in right side bounces back and tries to land two more times before succeeding. The plots also display a vertical line indicating the timestamp at which the UAV touched the ground.

*Feature Extraction.* The flight log files contain a vast amount of data, coming from as many as 65 message publishers in the examined UAV, but not all of the data is relevant in our context. Therefore, we decided to focus our analysis on four messages published by the UAV's state and position estimator. The selected messages are particularly relevant as they are artifacts of the Extended Kalman Filter (EKF) [119]. This makes them heavily reliant on the UAV's perception layer.

In particular, we selected the estimated sensor biases and estimated errors in combination with the estimated vibrations. Based on the functioning of the EKF filter, the growth in estimated sensor biases can lead to a growth in estimated position, velocity and angles errors. In addition to evaluating values related to the mathematical model governing the UAV, we added the vibration metrics of the UAV, as low vibrations are needed to establish a smooth flight [91]. As shown in Table 4, we selected the UAV's local and global position estimator, paired with the estimated sensor bias and estimated status. Each of the selected published messages contains many features, some of which were removed from our analysis as they were not used by the UAV and always reported a zero-value throughout the landing phase. The resulting feature set can be seen in Table 4.

All data stemming from the single flight logs are consolidated into a single dataset to apply feature selection methods. This is achieved by computing the average value for each physical feature during the landing phase for each flight log. This process generates a dataset containing the same amount of data entries as the number of flights in the dataset, with each data entry containing the average of each selected feature.

To better evaluate the impact of each selected feature, we compute the Mean Decrease Gini, also referred to as the Mean Decrease in Impurity [62, 114]. The Mean Decrease Gini is a measure of how each feature contributes to the homogeneity of the nodes and leaves in the resulting random forest [114, 170]. The Mean Decrease Gini is used to compute the feature importance and narrow the number of log entries that our model has to keep into account[114, 170]. The result of this investigation can be found in Figure 12, which shows the features in order of importance.

Table 4: Relevant perception features and their description

| Topic | Content | Description | Range | Detected Range |
|---|---|---|---|---|
| estimator_local_position | Timestamp | Publishing timestamp | $0, \infty$ | – |
| | (x, y, z) | Estimated x, y and z position | $-\infty, \infty$ | $0, 177.8m$ |
| | (vx, vy, vz) | Estimated speed in x,y and z direction | $-\infty, \infty$ | $-3.3, 8.9\frac{m}{s}$ |
| | (ax, ay, az) | Estimated acceleration in x,y and z direction | $-\infty, \infty$ | $-119.3, 79.0\frac{m}{s^2}$ |
| | eph, epv | Estimated positional error in horizontal direction and in vertical | $0, \infty$ | $0.1, 157.4m$ |
| | evh, evv | Estimated velocity error in horizontal direction and vertical | $0, \infty$ | $0.08, 157.4\frac{m}{s}$ |
| estimator_global_position | Timestamp | Publishing timestamp | $0, \infty$ | – |
| | Lat, Lon, Alt | Estimated latitude, longitude and altitude (GPS based) | $-\infty, \infty$ | – |
| | eph, epv | Estimated positional error in horizontal direction and vertical | $0, \infty$ | $1, 3.6m$ |
| estimator_sensor_bias | Timestamp | Publishing timestamp | $0, \infty$ | – |
| | Gyroscope bias | Estimated sensor bias present in the gyroscope | $-1, 1$ | $-0.05e{-}2, 0.27e{-}2$ |
| | Accellerometer bias | Estimated sensor bias present in the accelerometer | $-1, 1$ | $-0.42e{-}2, 20.02e{-}2$ |
| | Magnetometer bias | Estimated sensor bias present in the magnetometer | $-1, 1$ | $-0.03e{-}2, 0.35e{-}2$ |
| estimator_status | Timestamp | Publishing timestamp | $0, \infty$ | – |
| | Vibrations | Estimated vibrations percieved by the UAV | $0, 1$ | $0.79e{-}4, -0.68e{-}2$ |

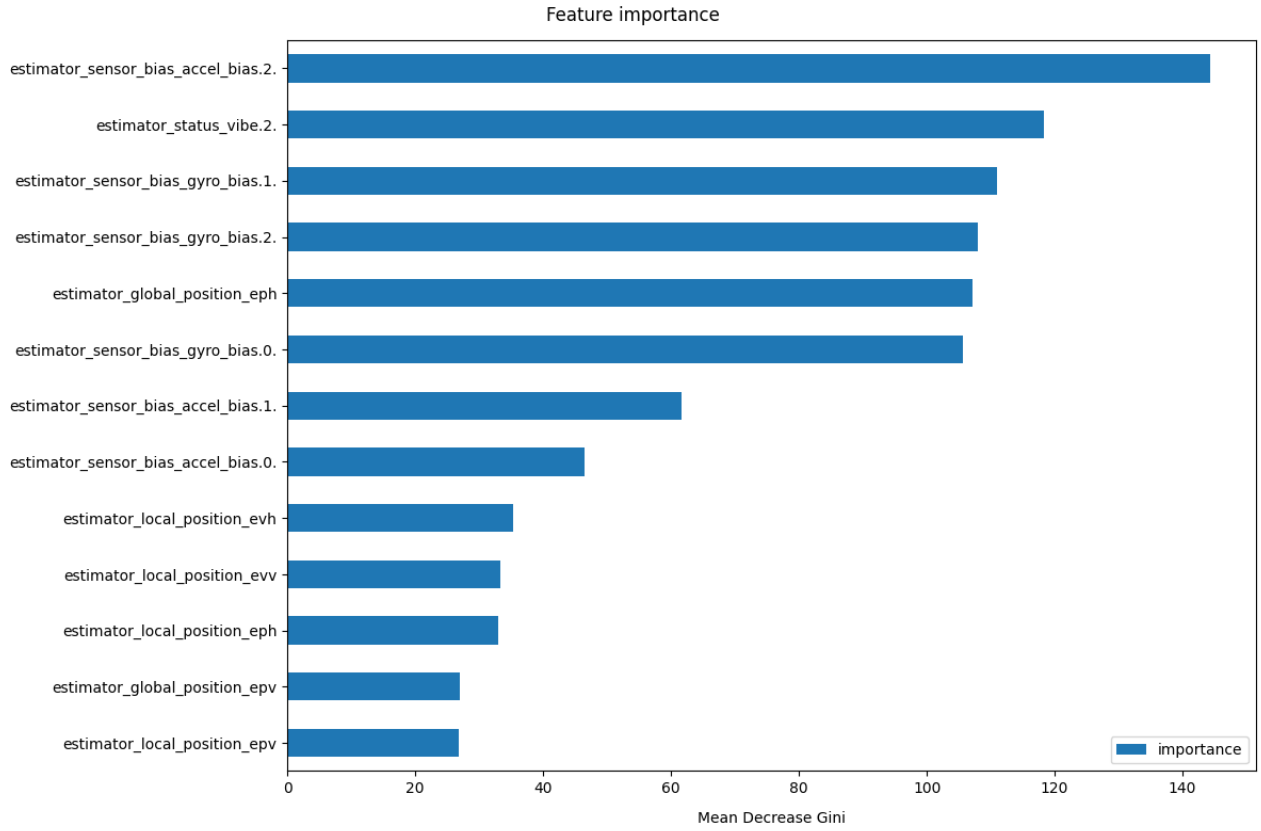


Figure 12: Top features ordered by importance from the top to the bottom

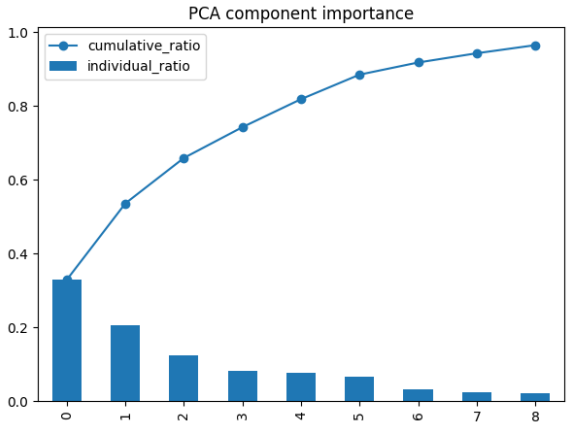


Figure 13: PCA component importance

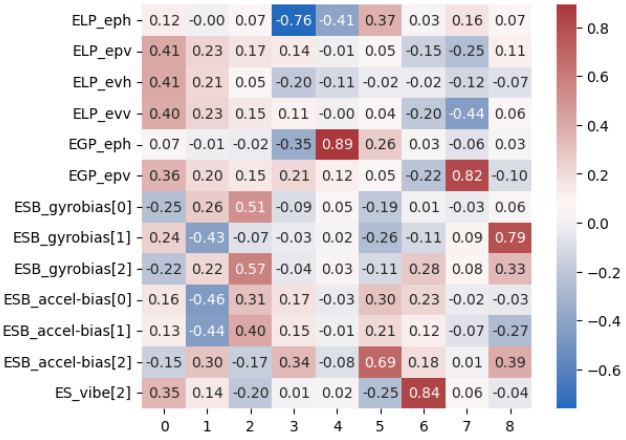


Figure 14: Heatmap of the contibution of features to the components from the PC analysis

Using the results from the Mean Decrease Gini investigation as a starting point, we reduced the features of interest to the following:

- Estimator Sensor Bias: Accelerometer Bias and Gyroscope Bias
- Estimator Local Position: Estimated positional and velocity error vertical and horizontal (epv, eph, evv, evh)
- Estimator Global Position: Estimated positional error vertical and horizontal (epv, eph)
- Estimator Status: vibrations in z-directions

We then performed a Principal Component Analysis on the dataset to identify which features have the highest impact on our ability to classify a UAV landing as ideal or bouncing. As shown in Figure 13, the PCA identified 9 principal components with non-negligible importance, thus with significance above 1%. The first component accounts for 33% of the variance in the data. The second component accounts for 21%, the first two components account thus for more than 50% of the variance in the data.

As visible in Figure 14, the first principal component is most strongly correlated with the estimated local velocity error and positional error. The second component increases with the decrease of the accelerometer bias and bias in the gyroscopes' pitch direction, and the third component relies on a combination of the gyroscope bias and the accelerometer bias. Figure 14 also displays how some of the features are correlated to each other, such as the positional error for global and positional estimation and the estimated gyroscopic bias along the UAV's x- and y-axis.

**Preliminary Results** Here, we aim to adress the following research questions:

*$RQ_1$:What are the UAV's physical features (sensor values and estimations) that can distinguish ideal and bouncing landings?*

The Mean Decrease Gini of the aggregated features, Figure 15, displays the significant impact of the sensor bias from the accelerometer and gyroscope on the decision tree generated by the model. This is aligned with the feature importance found without the aggregation and discussed in Figure 12, where we identified the sensor biases, combined with vibrations and positional errors to be the most important features. Figure 15 also shows that the Mean Decrease Gini on the aggregated feature set does not present one feature overly out-weighting the others. Combining these results with the results from the Principal Component Analysis, it appears that it is the combination of multiple aggregated features which leads to a better evaluation of a UAV's behaviour

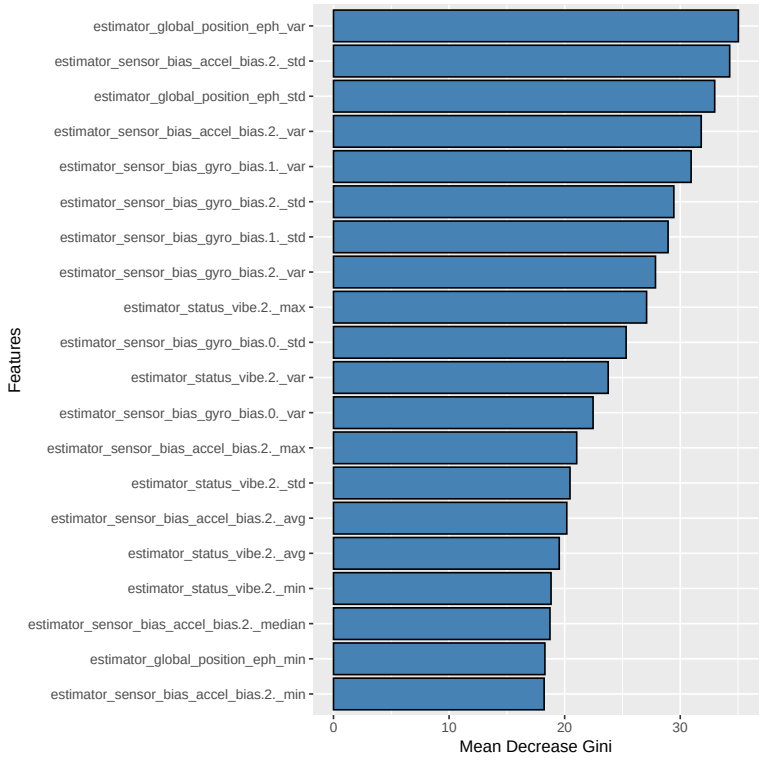


Figure 15: Mean Decrease Gini on aggregated features, sorted by top 20 most important features

at landing. Therefore the *physical features* as connected to UAV landing behaviour we identified in order of importance are: the global position error, the accelerometer bias, the gyroscope bias, the estimated vibrations on the vehicle z-axis, and finally the local position.

We identified 13 physical features connected to the landing behaviour. These comprise the accelerometer and gyroscope estimated sensor bias along each of the three-axis, the estimated positional error in the vertical and horizontal position for the global or local position, the estimated velocity error for the local position, and finally, the estimated vibrations along the UAV's z-axis.

*$RQ_2$: To what extent can we automatically classify the bouncing landings?*

Here, we train and evaluate various ML models using our dataset. The performance for each of the models is shown in Figure 16, the average prediction time for each entry lies at just 50 microseconds. The slowest model is the Random Forest Classifier with an average prediction time of 130$\mu$s, while the fastest is the Gaussian Naive Bayes Classifier with 2.6$\mu$s. Overall, the RF classifier and GNB performed nearly identical, with a slight difference of less than 0.01% on both the test and validation sets. The worst performing models are SVC and LRC, which only predicted some misbehaving flights correctly with a recall of 0.19 and 0.24, respectively. Overall the accuracy of RFC and GNB is never below 0.97, with a precision score near 1. This results in RFC and GNB being the best performing models across all scenarios.

With an increased offset, the models will perform differently. As shown in Figure 18, if the offset is increased beyond 1 second, the model accuracy, recall and F-Score decrease rapidly, to then stabilise at a value around 0.8. The model's performance seems to be worst at 2 seconds, which could be due to two factors. First, the results were only based on one random forest generation, and thus, due to the randomness in the creation process, the results could vary one run from the second one. Secondly, the model could be in a transition phase from one set of most important features to another that performs better after 2.5 seconds.

The Random Forest and the Gaussian Naive Bayes classification algorithms can automatically classify *bouncing* landings with precision, recall, accuracy and F-metric values above 96%. The models were trained on the aggregated data from the start of the landing until the end with a offset of 1 seconds. While all the models

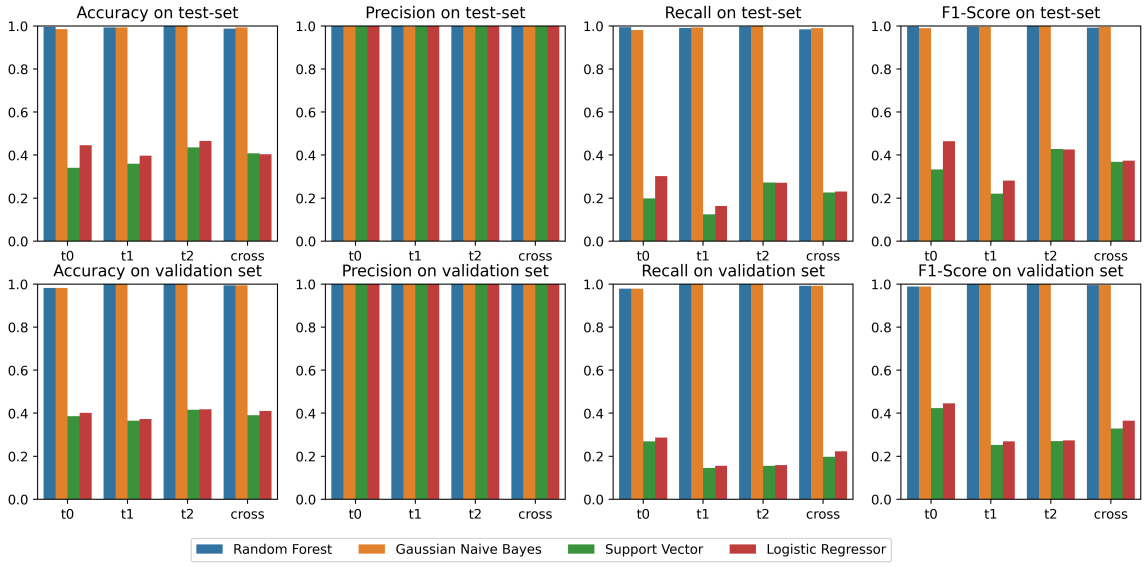


Figure 16: ML Model metric comparison on full landing phase classification

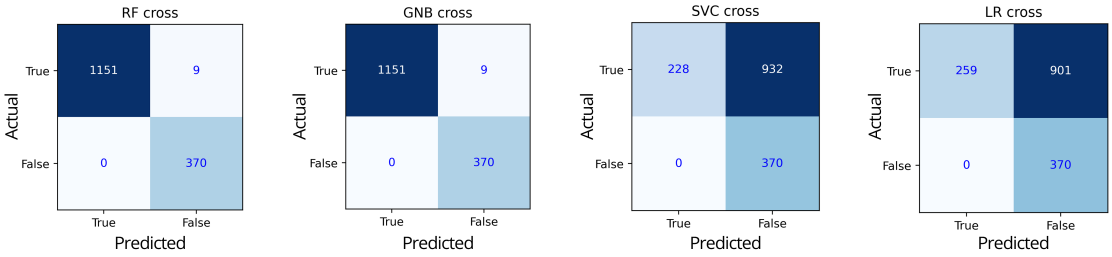


Figure 17: Confusion matrix for cross-scenario on validation set

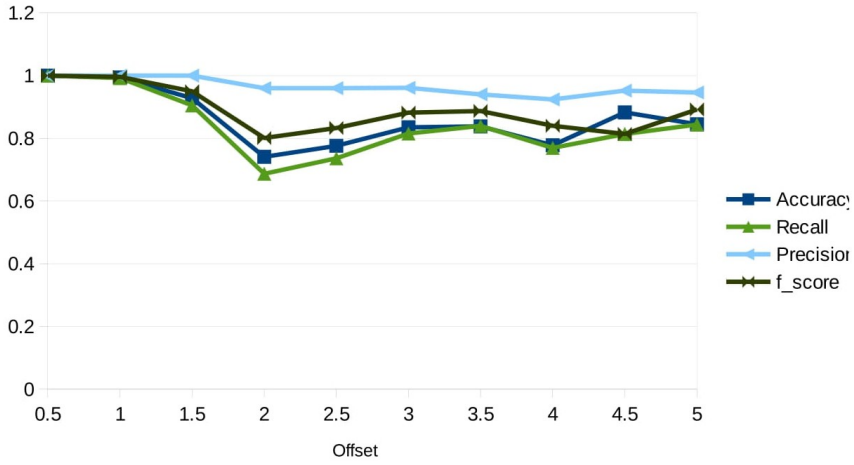


Figure 18: Evolution of performance metrics for Random Forest with different offsets

showed a precision equal to 1, the Random Forest Classifier and Gaussian Naive Bayes outperformed the Support Vector Classifier and Logistic Regressor. The RFC performed roughly equal to the GNB, in all scenarios it they were evaluated on, while instead SVC and LRC performed poorly in all scenarios. In contrast, there are no substantial differences ($\pm$ 1%) in the performance of the best models (RFC, GNB) on different scenarios. It is important to note that the GNB algorithm has a substantial advantage of predicting faster by a factor of 50 (2.6$\mu$s per prediction). We also show that with the increase of the offset, the performance metrics decreases, suggesting that there are some features which can more reliably predict the behaviour of the UAV during the last seconds.

## 3.3 Prototype for Monitoring SDC States

In this section, we overview of SDC-Scissor[3] software architecture and its main usage scenarios; we describe the simulation environment it uses (i.e., BeamNG.tech); and, finally, we discuss in detail the components, the approach and the technologies behind SDC-Scissor.

### 3.3.1 SDC-Scissor Architecture Overview & Main Scenarios

SDC-Scissor supports two main usage scenarios: *Benchmarking* and *Prediction*. In the *Benchmarking* scenario, developers leverage SDC-Scissor to determine the best ML model(s) to classify SDC simulation-based tests as safe or unsafe. In the *Prediction* scenario, instead, developers use those model(s) to classify and select newly generated test cases.

SDC-Scissor Software Architecture implements these scenarios by means of five main software components: (i) `SDC-Test Generator` generates random SDC simulation-based tests, and (ii) `SDC-Test Executor` executes them. The test results produced by `SDC-Test Executor` are recorded and used to label tests as safe or unsafe; (iii) `SDC-Features Extractor` extracts input features of the executed SDC tests, while (iv) `SDC-Benchmarker` uses these features and corresponding labels as input to train the ML models and determine which model best predicts the tests that are more likely to detect faults in SDCs; finally, (v) `SDC-Predictor` uses the ML models to classify newly generated test cases and enables test selection.

### 3.3.2 BeamNG.tech's Simulation Environment

SDC-Scissor uses BeamNG.tech to execute SDC tests as physically accurate and photo-realistic driving simulations. BeamNG.tech can procedurally generate tests [59] and was recently adopted in the ninth edition of the Search-Based Software Testing (SBST) tool competition [127].

BeamNG.tech is organized around a central *game engine* that communicates with the *physics simulation*, the *UI*, and the *BeamNGpy API*[4]. The UI can be used for game control and manual content creation (e.g., *assets*, *scenarios*). For example, developers can use the world editor to create or modify the virtual environments that are used in the simulations; testers, instead, can create test scripts implementing driving scenarios (i.e., the tests). The API, instead, allows the automated generation and execution of tests, the collection of simulation data (e.g., camera images, LIDAR point clouds) for training, testing, and validating SDCs. It also enables driving agents to drive simulated vehicles and get programmatic control over running simulations (e.g., pause/resume simulations, move objects around). The *game engine* manages the simulation setup, camera, graphics, sounds, gameplay, and overall resource management. The *physics core*, instead, handles resource-intensive tasks such as collision detection and basic physics simulation; it also orchestrates the concurrent execution of the vehicle simulators. The *vehicle simulators* —one for each of the simulated vehicles— simulate the high-level driving functions and the vehicle sub-systems (e.g., drivetrain, ABS).

[3] `https://github.com/ChristianBirchler/sdc-scissor`

[4] `beamngpy` is available on PyPI and Github (`https://github.com/BeamNG/BeamNGpy`)

Table 5: Full Road Attributes extracted by the *SDC-Features Extractor*

| Feature | Description | Range |
|---|---|---|
| Direct Distance | Euclidean dist. between start and end (m) | [0 – 490] |
| Length | Tot. length of the driving path (m) | [50.6–3,317] |
| Num L Turns | Nr. of left turns on the driving path | [0 – 18] |
| Num R Turns | Nr. of right turns on the driving path | [0 – 17] |
| Num Straight | Nr. of straight segments on the driving path | [0 – 11] |
| Total Angle | Cumulative turn angle on the driving path | [105 – 6,420] |

We employ the BeamNG.AI[5] lane-keeping system as the test subject for our evaluation: the driving agent is shipped with BeamNG.tech and drives the car by computing an ideal driving trajectory to stay in the center of the lane while driving within a configurable speed limit. As explained by BeamNG.tech developers, the *risk factor* (RF) is a parameter that controls the driving style of BeamNG.AI: low-risk values (e.g., 0.7) result in smooth driving, whereas high-risk values (e.g., 1.7 and above) result in an edgy driving that may lead the ego-car to cut corners [88].

### 3.3.3 The SDC-Scissor's Approach and Technology Overview

SDC-Scissor integrates the extensible testing pipeline defined by the SBST tool competition[6] in its `SDC-Test Executor`. We use the SBST tool competition infrastructure since it allows to (i) seamlessly execute the tests in BeamNG.tech and (ii) distinguish between *safe* and *unsafe* tests based on whether the self-driving car keeps its lane (non-faulty tests) or depart from it (faulty tests) [59]. Consequently, SDC-Scissor can accommodate various `SDC-Test Generators` for generating SDC simulation-based tests. In this deliverable, we demonstrate SDC-Scissor by using the Frenetic test generation [32], one of the most effective tool submitted to the SBST tool competition.

SDC-Scissor predicts whether the tests are likely to be safe or unsafe before their execution using input features extracted by `SDC-Features Extractor`. Specifically, this component extracts *Full Road Features* (FRFs), i.e., a set of SDC features that describe global characteristics of the tests. Those features include the main *road attributes* (see Table 5) and *road statistics* concerning the road composition (see Table 6). Road statistics are calculated in three steps: (i) extraction of the *reference driving path* that the ego-car has to follow during the test execution (e.g., the road segments that the car needs to traverse to reach the target position); (ii) extraction of metrics available for each road segment (e.g., length of road segments); and (iii) computation of standard aggregation functions on the collected road segments metrics (e.g., minimum and maximum).

SDC-Scissor relies on the `SDC-Benchmarker` to determine the ML model that best classifies the SDC tests that are likely to detect faults. It follows an empirical approach to do so: given a set of labeled tests and corresponding input features, `SDC-Benchmarker` trains and evaluates an ensemble of standard ML models using the well-established `sklearn`[7] library. Next, it assesses ML models' quality using either 10-fold cross-validation or a testing dataset; and, finally, selects the best performing ML models according to Precision, Recall, and F1-score metrics [88]. Noticeably, SDC-Scissor can use many different ML models; however, in this work, we consider only Naive Bayes [31], Logistic Regression[144], and Random Forests [73]. We do so because these ML models have been successfully used for defect prediction or other classification problems in Software Engineering [21, 84, 128, 42].

Finally, the `SDC-Predictor` uses the ML models to predict the likelihood that newly generated SDC tests are safe or not. Specifically, developers have the possibility to select the ML models recommended by the `SDC-Benchmarker` (considered most accurate), or they can select other models of their choice.

[5] `https://wiki.beamng.com/Enabling_AI_Controlled_Vehicles#AI_Modes`
[6] `https://github.com/se2p/tool-competition-av`
[7] https://scikit-learn.org/

Table 6: Full Road Statistics extracted by the *SDC-Features Extractor*

| Feature | Description | Range |
|---|---|---|
| Median Angle | Median turn angle on the driving path (DP) | [30 – 330] |
| Std Angle | Std. Dev of turn angles on the DP | [0 – 150] |
| Max Angle | Max. turn angle on the DP | [60 – 345] |
| Min Angle | Min. turn angle on the DP | [15 – 285] |
| Mean Angle | Average turn angle on the DP | [52.5–307.5] |
| Median Radius | Median turn radius on the DP | [7 – 47] |
| Std Radius | Std. Dev of turn radius on the DP | [0 – 22.5] |
| Max Radius | Max. turn radius on the DP | [7 – 47] |
| Min Radius | Min. turn radius on the DP | [2 – 47] |
| Mean Radius | Average turn radius on the DP | [5.3 – 47] |

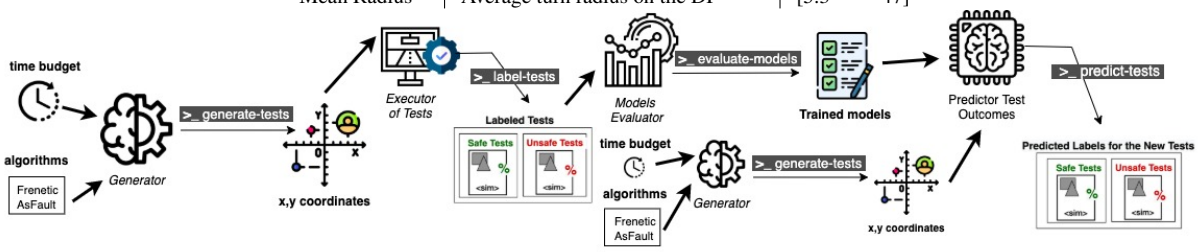


Figure 19: The SDC-Scissor's fine-grained view.

### 3.3.4 Using SDC-Scissor

SDC-Scissor tool is openly available and can be used as a Python command-line utility via `poetry`[8] as follows:

```
poetry install
poetry run python sdc-scissor.py [COMMAND] [OPTIONS]
```

To simplify SDC-Scissor's usage, we also enable to execute it as a Docker[9] container:

```
docker build --tag sdc-scissor .
docker run --volume "$(pwd)/results:/out" --rm
    sdc-scissor [COMMAND] [OPTIONS]
```

As we detail below, SDC-Scissor's command-line supports the execution of the main usage scenarios described in Section 3.3.2 by taking appropriate commands and inputs (see Fig. 19).

**Test generation.** To generate SDC tests by running the Frenetic generator within a given time budget (e.g., 100 seconds) SDC-Scissor requires the following command:

```
generate-tests --out-path /path/to/store/tests
    --time-budget 100
```

**Automated test labeling.** SDC-Scissor labels tests as safe and unsafe by executing them in BeamNG.tech. Since BeamNG.tech cannot be run as a Docker container, labelling tests can be only run locally (i.e., outside Docker). This labeling facility allows developers to create datasets that can be used for the training and validation of ML models (e.g., ML-based prediction of unsafe tests). Generating a labeled dataset, requires a set of already generated SDC tests and the execution of the following command:

```
label-tests --road-scenarios /path/to/tests
    --result-folder /path/to/store/labeled/tests
```

**ML models evaluation.** For identifying the models that SDC-Scissor could use for the prediction, SDC-Scissor implements a 10-fold cross-validation strategy on the input labeled dataset. The following command tells SDC-Scissor to benchmark all the configured ML models:

```
evaluate-models --tests /path/to/train/set --save
```

[8] https://python-poetry.org/
[9] https://www.docker.com

Note: the optional `save` flag forces SDC-Scissor to store the ML models' metadata for later inspection and usage.

**Train and test data generation.** Evaluating the prediction ability of SDC-Scissor requires separate training and testing datasets. The following command lets developers to split the available tests to achieve an 80/20 split:

```
split-train-test-data --tests /path/to/tests
    --train-dir /path/for/train/data
    --test-dir /path/for/test/data
    --train-ratio 0.8
```

**Test outcome prediction.** SDC-Scissor classifies unlabeled tests, i.e., it predicts their outcome, using a trained ML model with the following command:

```
predict-tests --tests /path/to/tests
    --predicted-tests /path/for/predicted/tests
    --classifier /path/to/model
```

**Random baseline evaluation.** SDC-Scissor allows to select tests using a random strategy that provides a baseline evaluation with the following command:

```
evaluate-cost-effectiveness
    --tests /path/to/tests
```

**Prediction performance.** SDC-Scissor allows to assess the performance of a classifier with the following command:

```
evaluate --tests /path/to/tests
    --classifier /path/to/model
```

### 3.3.5 Evaluation

We evaluated SDC-Scissor conducting a large study on two datasets, referred as *Dataset 1* and *Dataset 2*, that contain over $12,000$ SDC tests (see Table 7). We adopted the following experimental setup to obtain comprehensive and unbiased training datasets. For *Dataset 1*, we *randomly* generated $3,559$ valid tests using Frenetic [32], collected input features and executed them to collect labels. For the *Dataset 2*, instead, we generated $8,545$ tests using AsFault [59].

It is important to note that in executing all those tests, we experimented with different BeamNG.AI's risk factor as it influences the ego-car driving style. Specifically, we considered three configurations: cautious (RF 1.0), moderate (RF 1.5), and reckless (RF 2.0) driver. Using different values for the risk factor enabled us to study the effectiveness of SDC-Scissor on various SDCs' driving styles. We empirically validated our expectations by running the moderate driver using *Dataset 1* tests and running all the three configurations for *Dataset 2* tests. From Table 7 we can observe that the number of unsafe tests increased with increasing values of BeamNG.AI's

Table 7: Datasets Summary

| Dataset | Test Subject | Data Points Unsafe | Safe | Total |
|---|---|---|---|---|
| *Dataset 1* | BeamNG.AI moderate | 1'334 (37%) | 2'225 (63%) | 3'559 |
| *Dataset 2* | BeamNG.AI cautious | 312 (26%) | 866 (74%) | 1'178 |
| | BeamNG.AI moderate | 2'543 (45%) | 3'095 (55%) | 5'638 |
| | BeamNG.AI reckless | 1'655 (96%) | 74 (4%) | 1'729 |
| | **Total** | 5'844 (48%) | 6'260 (52%) | 12'104 |

risk factor. Hence, this result confirms that the risk factor indeed strongly influences the safety of BeamNG.AI and the outcome of tests.

To assess the performance of SDC-Scissor in optimizing simulation-based SDCs testing via test selection (i.e., in selecting unsafe tests before executing them), for both *Dataset 1* and *Dataset 2* we experimented with the ML models mentioned in Section 3.3.3 trained and validated using an 80/20 split.

As reported in Table 8, on *Dataset 1* SDC-Scissor accurately identified unsafe tests, with F1-score ranging between 35.1% and 56.1%. On *Dataset 2*, instead, it identified unsafe tests with F1-score ranging between 52.5% and 96.4%.

Table 8: Performance of the ML models with dataset split 80/20. The best results are shown in boldface.

| Dataset | RF | Model | Prec. | Recall | F1-score |
|---|---|---|---|---|---|
| *Dataset 1* | **RF 1.5** | Logistic | **45.8%** | 60.9% | 52.3% |
| | | Naïve Bayes | 40.2% | **92.5%** | **56.1%** |
| | | Random Forest | 41.3% | 30.5% | 35.1% |
| *Dataset 2* | **RF 1** | Logistic | **43.3%** | 87.3% | **57.9%** |
| | | Naïve Bayes | 36.7% | **92.1%** | 52.5% |
| | | Random Forest | 40.7% | 79.4% | 53.8% |
| *Dataset 2* | **RF 1.5** | Logistic | 78.1% | **65.3%** | **71.1%** |
| | | Naïve Bayes | **79.3%** | 53.2% | 63.6% |
| | | Random Forest | 75.8% | 62.7% | 68.6% |
| *Dataset 2* | **RF 2** | Logistic | **99.6%** | 82.8% | 90.4% |
| | | Naïve Bayes | 98.7% | **94.3%** | **96.4%** |
| | | Random Forest | **99.7%** | 92.7% | 96.1% |

Complementary to the previous experiments, we investigated, in the context of *Dataset 2*, SDC-Scissor's ability to be more cost-effective compared to a *random-based baseline* that randomly selects from the dataset the tests to be executed [88]. Specifically, SDC-Scissor was trained on 70% of tests from *Dataset 2* and tested on the remaining 30% of tests, while the random-based baseline randomly selected the same amount of tests directly from the remaining 30% of tests. Our evaluation on *Dataset 2* shows that for all RF (risk factor) values the best performing ML model of SDC-Scissor (i.e., Logistic) reduced the time spent in running safe/unnecessary tests than a random baseline strategy with a speed-up of circa 170%. On *Dataset 1*, instead, SDC-Scissor speed up testing up to 158% for the Naïve Bayes and 107% for Logistic.

### 3.3.6 Conclusions

This deliverable presented SDC-Scissor, a ML-based test selection approach that classifies SDC simulation-based tests as likely (or unlikely) to expose faults before executing them. SDC-Scissor trains ML models using input features extracted from driving scenarios, i.e., SDC tests, and uses them to classify SDC tests before their execution. Consequently, it selects only those tests that are predicted to likely expose faults. Our evaluation shows that SDC-Scissor successfully selected unsafe test cases across different driving styles and drastically reduced the execution time dedicated to executing safe tests compared to a random baseline approach.

As future work, we plan to replicate our study on further SDC datasets, AI engines and SDC features to study how the results generalize in the autonomous transportation domain. Additionally, given our close contacts with the BeamNG.tech team, we plan the integration of SDC-Scissor into BeamNG.tech environment to enable researchers and SDC developers to use SDC-Scissor as a cost-effective testing environment for SDCs. Finally, we plan to investigate the use of SDC-Scissor in other CPS domains, such as drones, to investigate how it performs when testing focuses on different types of safety-critical faults. Specifically, important for this is to investigate approaches that are more human-oriented or are able to integrate humans into-the-loop [159, 128, 42, 67].

# 4 Flaky Tests Analysis and Identification for UAVs

UAVs can behave nondeterministic both in simulation environments and in the real world because of various reasons including the noise in the sensor outputs, message delays in their inter module communications, battery malfunctioning, changes in weather condition, and also because of the randomness involved in their control algorithms, e.g., their AI components. We already discussed the nondeterminism in the landing phase of a UAV flight in section 3.2.2. Here we focus our analysis mostly on the obstacle avoidance in autonomous flights.

To be able to study test flakiness in UAV, we first generate *challenging* test cases for the UAV, and then analyse the behaviour of the drone, executing the exact same test case for multiple times, using AERIALIST.

## 4.1 Study Definition and Methodology

*Context.* Given a simulated test case configuration for autonomous flight (the mission waypoints and obstacle locations and sizes), we want to generate a more challenging simulated test case by introducing an additional obstacle, to force the UAV to get too close to the obstacle (i.e.,, having a distance below a predefined safety threshold) while still completing the mission. This will create a risky environment for the UAV to operate the mission in.

Specifically we want to answer the following research question:

*Can we modify the simulated test case properties during autonomous flights to make them more challenging for the UAV autonomous controller, and force it to behave non-deterministic?*

We first put a secondary obstacle with the exact size of the first one in a position far enough from the initial obstacle that does not change make the UAV to change trajectory, as illustrated in Figure 20(left). We then slightly move the second obstacle towards the first one, to close the path for the drone, and force it to take a more risky path. For each candidate test case, we run the test in simulation for 10 times in parallel using AERIALIST. We aggregate the logs, and compute the minimum distance of the flights to the obstacles, which we want to minimize.

## 4.2 Study Results

We repeat our search for the second obstacle's position for 10 times, and run each of the found test cases for 10 times in parallel. As can be seen from the best final solution across 10 runs in Figure 21, we were able to position and size the second obstacle in a way that the UAV (i) was forced to behave in a non-deterministic way across multiple parallel simulations, taking different routes through or around the obstacles; (ii) experienced an unsafe behavior, often very close to the first obstacle; (iii) even worse, occasionally the UAV crashed into the obstacle, in some simulations. The second obstacle was moved to almost $8m$ to the left and $1.1m$ upwards, making it increasingly harder for the UAV to follow the path. Interestingly, if we position the obstacles closer to each other (as illustrated in Figure 20(right), the UAV would always act in a deterministic way, always taking a route around the left of the first obstacle, without getting involved in risky situations.

As reported in Table 9, the algorithm was able to find crashing test cases consistently in all 10 runs, forcing the UAV to get as close as $0m$ to the obstacle, down from the $3.3m$ safety distance of the seed. Also, on average, the UAV crashed into the obstacle in 2.5 out of the 10 simulations for the best test cases found, and got unsafely close (less than 1.5m) in 6 more of them. All experiment results are available online [10].

[10] https://filer.cloudlab.zhaw.ch/index.php/apps/files/?dir=/ASE2022/Flaky%20Test%20Generation&fileid=2412623

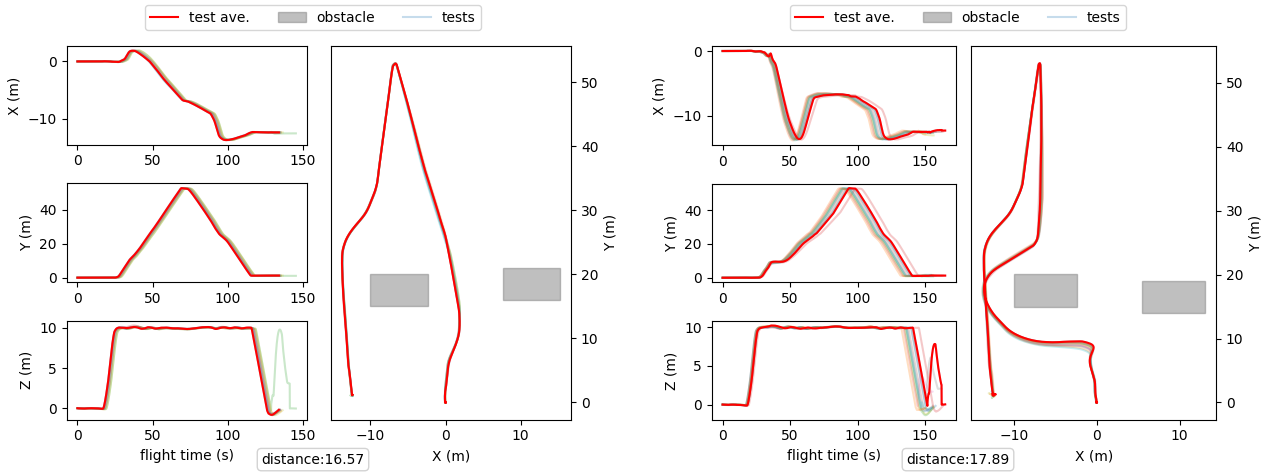


Figure 20: Intermediate test cases with deterministic flights

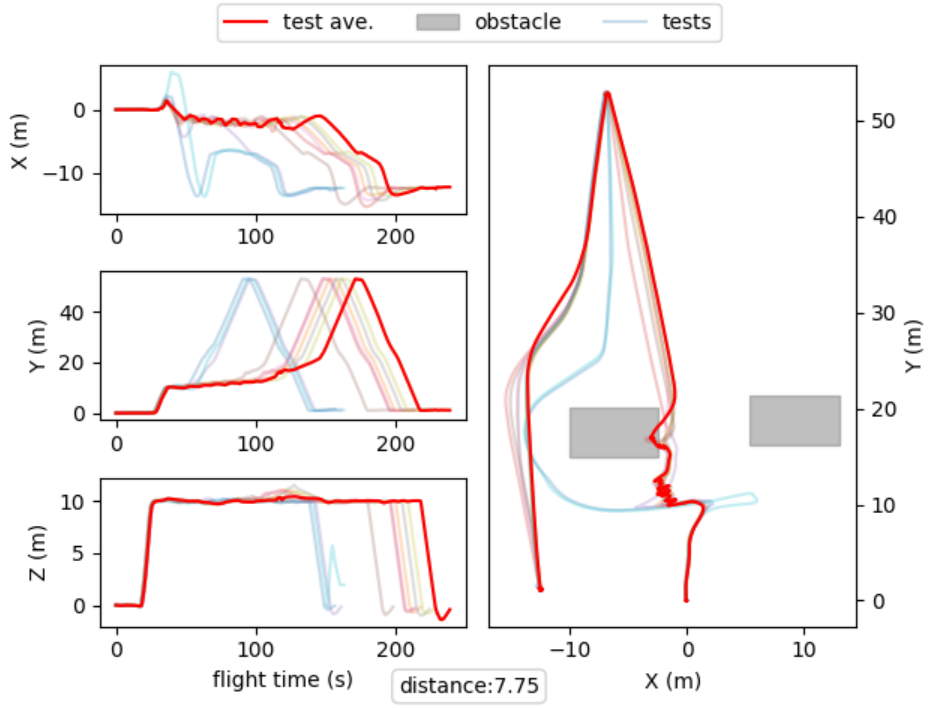


Figure 21: final test case with nondeterministic UAV trajectory

Table 9: Evaluation metrics

| Metric | Ave. |
|---|---|
| Seed Min_dist | 3.36 m |
| Final Min_dist | 0 m |
| Min_dist Red. | 100% |
| Crash Rate | 25 % |
| Unsafe Rate | 84 % |

Modifying a replicated field test in the simulation allows generating challenging test cases that can expose the UAV to nondeterministic behaviors or even crashes.

## 4.3 Future Works

Next step of the research of this deliverable is to study the characteristics of the UAV during the flaky tests. Specifically, we will investigate the root causes of the observed nondeterminism in the above test cases. This will lead us toward a solution to automatically distinguish test cases that can potentially be flaky, even if we have not observed contradicting test results with its limited executions.

# 5 Flaky Tests Analysis and Identification for SDCs

**Motivation**. The behavior or SDCs in simulation environments is non-deterministic as preliminary experiments showed. If a test case fails and passes in different runs then it is considered as flaky. Naturally, the identification of flaky tests depends on the definition of a failure. In the preliminary experiments the lane-keeping ability of the SDC was tested and checked if the SDC is going out of the lane to a certain percentage.

The following sections describe the methodology used to identify flaky tests and show the results of the preliminary experiments with the according tool[11].

## 5.1 Study Definition and Methodology

### 5.1.1 Methodology Overview

To assess the flakiness of SDCs in simulation-based tests several executions of the same tests are required to determine if the test outcomes of a single test changes among several executions. The definition of a failure impacts naturally the flakiness behavior of SDCs. For the preliminary assessment of flakiness the following definition of a failure applies. If the car drives off the lanes (from the boundaries of the road) for 50% then the test is seen as a unsafe scenario and a failure is observed.

To generate a dataset of test cases for SDCs the Frenetic test generator is used of the SBST2021 tool competion [32]. The generated dataset of test cases is passed into a pipeline that executes each single test case 10 times. The used pipeline is SDC-Scissor [23] which allows to develop specialized testing pipelines for SDCs with the BeamNG.tech [12] simulator. The BeamNG.tech simulator allows to simulate SDCs more realistic than other simulators since it uses a sophisticated soft-body physics engine. As soon the car goes off the lane by $50\%$, the test executions failed, otherwise it passes. The outcome (pass, fail, or error) of each test execution is tracked and stored as a CSV file for post-analysis.

For the post-analysis, for each execution of a test cases the outcome is analyzed. The analysis comes up with two types of flakiness:

1. If the outcome changes of a single test and there is no error in one of the executions then the test case is seen as strongly flaky.

2. If the out come changes because there is also an error in the test execution then the test case is seen as weakly flaky.

[11] `https://cosmos-devops.cloudlab.zhaw.ch/cosmos-devops/cosmos-devops-internal/-/tree/master/WP6/zhaw-cosmos-flaky`

[12] `https://beamng.tech/`

| ID | Risk Factor | Version | Chunk | Test ID | Executions | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 156 | 1.0 | 2 | 21 | 9 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 344 | 1.0 | 3 | 22 | 4 | 1 | 1 | 1 | 1 | 1 | 1 | -1 | -1 | 1 | 1 |
| 345 | 1.0 | 3 | 22 | 5 | 1 | 1 | 1 | 1 | 1 | 1 | -1 | 1 | 1 | 1 |
| 346 | 1.0 | 3 | 22 | 10 | 1 | 1 | 1 | 1 | 1 | 1 | -1 | 1 | 1 | -1 |
| 347 | 1.0 | 3 | 22 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | -1 | 1 | 1 | 1 |
| 348 | 1.0 | 3 | 22 | 8 | 1 | 1 | 1 | 1 | 1 | 1 | -1 | 1 | 1 | 1 |
| 349 | 1.0 | 3 | 22 | 7 | 0 | 0 | 0 | 0 | 0 | 0 | -1 | 0 | 0 | 0 |
| 351 | 1.0 | 3 | 22 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | -1 | 0 | 0 | 0 |
| 352 | 1.0 | 3 | 22 | 6 | 1 | 1 | 1 | 1 | 1 | 1 | -1 | 1 | 1 | 1 |
| 353 | 1.0 | 3 | 22 | 9 | 1 | 1 | 1 | 1 | 1 | 1 | -1 | 1 | 1 | 1 |
| 2401 | 2.0 | 3 | 24 | 5 | 1 | 1 | 1 | 1 | -1 | 1 | 1 | 1 | 1 | 1 |
| 2402 | 2.0 | 3 | 24 | 10 | 1 | 1 | 1 | 1 | -1 | 1 | 1 | 1 | 1 | 1 |
| 2403 | 2.0 | 3 | 24 | 2 | 0 | 0 | 0 | 0 | -1 | 0 | 0 | 0 | 0 | 0 |
| 2404 | 2.0 | 3 | 24 | 3 | 0 | 0 | 0 | 0 | -1 | 0 | 0 | 0 | 0 | 0 |
| 2405 | 2.0 | 3 | 24 | 8 | 0 | 0 | 0 | 0 | -1 | 0 | 0 | 0 | 0 | 0 |
| 2406 | 2.0 | 3 | 24 | 11 | 1 | 1 | 1 | 1 | -1 | 1 | 1 | 1 | 1 | 1 |
| 2407 | 2.0 | 3 | 24 | 1 | 1 | 1 | 1 | 1 | -1 | 1 | 1 | 1 | 1 | 1 |
| 2408 | 2.0 | 3 | 24 | 7 | -1 | 1 | 1 | 1 | -1 | 1 | 1 | 1 | 1 | 1 |
| 2409 | 2.0 | 3 | 24 | 6 | 1 | 1 | 1 | 1 | -1 | 1 | 1 | 1 | 1 | 1 |
| 2410 | 2.0 | 3 | 24 | 9 | 1 | 1 | 1 | 1 | -1 | 1 | 1 | 1 | 1 | 1 |
| 2411 | 2.0 | 3 | 24 | 4 | 1 | 1 | 1 | 1 | -1 | 1 | 1 | 1 | 1 | 1 |
| 2604 | 2.0 | 3 | 62 | 8 | -1 | -1 | 1 | 1 | 1 | -1 | 1 | 1 | 1 | 1 |
| 2605 | 2.0 | 3 | 62 | 7 | -1 | -1 | 1 | 1 | 1 | -1 | 1 | 1 | 1 | 1 |
| 2606 | 2.0 | 3 | 62 | 9 | -1 | -1 | 0 | 0 | 0 | -1 | 0 | 0 | 0 | 0 |
| 2607 | 2.0 | 3 | 62 | 1 | -1 | -1 | 1 | 1 | 1 | -1 | 1 | 1 | 1 | 1 |
| 2608 | 2.0 | 3 | 62 | 10 | -1 | -1 | 1 | 1 | 1 | -1 | 1 | 1 | 1 | 1 |
| 2609 | 2.0 | 3 | 62 | 5 | -1 | -1 | 1 | 1 | 1 | -1 | 1 | 1 | 1 | 1 |
| 2610 | 2.0 | 3 | 62 | 3 | -1 | -1 | 1 | 1 | 1 | -1 | 1 | 1 | 1 | 1 |
| 2611 | 2.0 | 3 | 62 | 2 | -1 | -1 | 1 | 1 | 1 | -1 | 1 | 1 | 1 | 1 |
| 2612 | 2.0 | 3 | 62 | 4 | -1 | -1 | 0 | 0 | 0 | -1 | 0 | 0 | 0 | 0 |
| 2613 | 2.0 | 3 | 62 | 11 | -1 | -1 | 0 | 0 | 0 | -1 | 0 | 0 | 0 | 0 |
| 2614 | 2.0 | 3 | 62 | 6 | -1 | -1 | 1 | 1 | 1 | -1 | 1 | 1 | 1 | 1 |
| 2697 | 2.0 | 3 | 10 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 3515 | 1.5 | 1 | 53 | 6 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3655 | 1.5 | 1 | 16 | 7 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 |
| 4346 | 1.5 | 3 | 20 | 8 | -1 | 1 | 1 | 1 | 1 | -1 | 1 | 1 | 1 | 1 |
| 4347 | 1.5 | 3 | 20 | 6 | -1 | 0 | 0 | 0 | 0 | -1 | 0 | 0 | 0 | 0 |
| 4348 | 1.5 | 3 | 20 | 2 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4349 | 1.5 | 3 | 20 | 7 | -1 | 1 | 1 | 1 | 1 | -1 | 1 | 1 | 1 | 1 |
| 4350 | 1.5 | 3 | 20 | 4 | -1 | 1 | 1 | 1 | 1 | -1 | 1 | 1 | 1 | 1 |
| 4352 | 1.5 | 3 | 20 | 9 | -1 | 1 | 1 | 1 | 1 | -1 | 1 | 1 | 1 | 1 |
| 4353 | 1.5 | 3 | 20 | 5 | -1 | 1 | 1 | 1 | 1 | -1 | 1 | 1 | 1 | 1 |
| 4354 | 1.5 | 3 | 20 | 3 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4580 | 1.5 | 3 | 59 | 11 | -1 | -1 | 0 | 0 | -1 | -1 | -1 | -1 | -1 | -1 |
| 4581 | 1.5 | 3 | 59 | 2 | -1 | -1 | 1 | 1 | -1 | -1 | -1 | -1 | -1 | -1 |
| 4582 | 1.5 | 3 | 59 | 9 | -1 | -1 | 1 | 1 | -1 | -1 | -1 | -1 | -1 | -1 |
| 4583 | 1.5 | 3 | 59 | 4 | -1 | -1 | 1 | 1 | -1 | -1 | -1 | -1 | -1 | -1 |
| 4584 | 1.5 | 3 | 59 | 5 | -1 | -1 | 0 | 0 | -1 | -1 | -1 | -1 | -1 | -1 |
| 4585 | 1.5 | 3 | 59 | 8 | -1 | -1 | 1 | 1 | -1 | -1 | -1 | -1 | -1 | -1 |
| 4586 | 1.5 | 3 | 59 | 7 | -1 | -1 | 1 | 1 | -1 | -1 | -1 | -1 | -1 | -1 |
| 4587 | 1.5 | 3 | 59 | 6 | -1 | -1 | 1 | 1 | -1 | -1 | -1 | -1 | -1 | -1 |
| 4588 | 1.5 | 3 | 59 | 3 | -1 | -1 | 0 | 0 | -1 | -1 | -1 | -1 | -1 | -1 |
| 4589 | 1.5 | 3 | 59 | 1 | -1 | -1 | 0 | 0 | -1 | -1 | -1 | -1 | -1 | -1 |
| 4590 | 1.5 | 3 | 59 | 10 | -1 | -1 | 1 | 1 | -1 | -1 | -1 | -1 | -1 | -1 |

Table 10: Strong (blue) and weak flaky test cases after the post-analysis. For each test case the test execution could result in a pass, fail, or error. These outcomes are marked as 1, 0, and -1 respectively.

All the results of the post-analysis are reported in a single CSV file. The CSV file will then give an overview of the flaky tests.

## 5.2 Study Results

The results as shown in Table 10 shows that two test cases were strongly flaky (marked with blue color) whereas the majority are weakly flaky. The results of the flaky tests are assessed by a dataset consisting of $4'756$ test cases with 10 test executions each. As shown in Table 10, two test cases are strongly flaky and $52$ are weakly flaky. Compared to the whole dataset their proportions are $0.04\%$ and $1.1\%$ respectively. However, out test cases had very simple scenarios, without other cars and human obstacles. We expect such number to increase.

# 6 Conclusion and Future Work

As elaborated in Section 3, to realize the COSMOS component for quality assessment and monitoring of CPS, COSMOS focuses on the following main high- and low-level challenges:

- To allow a DevOps pipeline to monitor a set of CPS relevant Quality Attributes (QAs):
  1. Monitoring CPS States
  2. Monitoring Change Propagation
  3. Monitoring of security vulnerabilities of CPS
  4. Monitoring of Antipatterns
- To mitigate CPS degradation forms:
  1. Detection of flakiness issues in X-in-the-loop testing
  2. Prediction of CPS degradation patterns
  3. Design of recovery solutions based on (micro-)fixes.
- Monitoring of KPIs related to key business and development goals

From a technological perspective, in the context of monitoring CPS States and flaky tests, COSMOS focuses on addressing the realization of solutions enabling the monitoring or identification of (i) of CPS states with X-in-the-loop Facilities, (ii) of Flaky Scenarios / tests for CPS with X-in-the-loop Facilities. Moreover, COSMOS focuses on addressing the realization of mitigation strategies focusing on: (i) the prediction and monitoring of Flaky Scenarios / Tests of CPS, to mitigate the risks of encountering unexpected behaviors while improving the general quality of CPS tests; (ii) support fixing of Flaky Scenarios / tests of CPS. Finally,

From a technological perspective, in the context of monitoring and propagation of CPS changes and vulnerabilities, COSMOS focuses on addressing the realization of solutions enabling the (i) CPS Change Analysis and Propagation (in Open-source use cases and COSMOS use cases); (ii) the CPS vulnerability analysis and propagation (in Open-source use cases and COSMOS use cases), with specific focus on identifying static vulnerabilities to use for vulnerability proneness analysis (and potentially the vulnerability propagation) in CPSs.

In previous sections we elaborated on the status of the current development. In the following sections we elaborate on the next steps of the COSMOS development activities concerning the realization of the COSMOS component for quality assessment and monitoring of CPS, with specific focus on activities that are expected to be addressed in next months of the COSMOS project.

## 6.1 Monitoring and Propagation of CPS Changes

In the previous, related deliverable D.6.1, we described the first steps in the development of a change analysis based framework that enables the automated evaluation of changes occurring in CPSs. Specifically, we reported the results of two studies conducted to derive (i) a broad CPS Bugs Taxonomy that can be applied to different domains; (ii) a focused characterization of safety issues of Unmanned Aerial Vehicles (UAVs), which is a specific CPS application domain; (iii) a fine-grained, qualitative categorization of hazard conditions and incidents in UAVs. Complementary, we introduced a preliminary taxonomy of changes occurring to open-source CPS projects. Finally, we discussed the ongoing implementation of two prototypes concerning the automated change analysis for detecting couplings and issue categorization/prioritization of CPSs, along with their architecture and/or preliminary data evaluation of selected open source CPS projects.

In the following sections, we elaborate on the next steps concerning the COSMOS activities focused on enabling the monitoring and propagation of CPS changes.

### 6.1.1 Extension of Change Analysis Framework

As part of future work, we will focus on type of changes that typically impact the behavior of CPS over certain functionalities or that impact other non-functional aspects. This activity is important to allow future COSMOS development concerning the identification of static co-evolutionary change patterns in CPS software and its assets (especially at the operational level) to determine the relations between (static) behavioral changes and CPS failures (i.e., the relation between a CPS failure and change patterns). Finally, we plan to extend the prototypes supporting the automated change analysis and issue categorization/prioritization of CPSs (discussed in the deliverable D.6.1), so that after a given change, both prototypes are used together for classifying CPS change types (possibly safety-related change types) using different types of information (e.g., starting from the GitHub issue fixed in the change, commit message, the source code, etc.).

More generically, the next period will be focused on slowly support and integrate change analysis strategies in the context of COSMOS use case partners. In the next section, we elaborate the current (high-level) plan of adaptation of the change analysis framework in the context of INT, a COSMOS partner in the satellite-specific CPS domain.

### 6.1.2 Monitoring and Propagation of Changes in the Context of Satellites on-board software

We elaborate on the next steps concerning the COSMOS activities focused on enabling the monitoring and propagation of changes in the context of INT, a COSMOS partner in the satellite-specific CPS domain. Hence, we discuss the context, needs, and envisioned automation in this context.

**Satellites Context**. Despite the technological advances in the satellite on board software development, verification and validation, the main overall design and implementation phases are still linked to manual coding and integration of software modules. In most cases these modules are libraries or previous implementations of specific on-board routines that have been proved against a real usage in previous missions.

Currently the Space market is living an interesting period because the Agencies (ESA at European level, and national space agencies) are pushing the industrial ecosystem to invest into the development of smaller and less expensive satellites platforms to operate services and functionalities that where previously operated only by major commercial and scientific missions. The interest to push the industrial market to invest in new constellations and new value-added services, also at benefit of the agencies and the EU countries, is moving into the EU space market newcomers, with less proven technology and less budget available in respect of the major missions looking for a valuable balance between reliability, speed in satellite delivery, and cost reduction of production. But a satellite (small or big) shall be always be considered as a complex system where several heterogeneous components have to efficiently inter-operate for delivering a certain mission goal. Moreover, on contrary of a single big satellite, constellations have to consider an even more complex scenario in planning and validating maneuvers where dozens of satellites have to coordinate each other to reach a certain mission objective that potentially can affect any other orbiting devices (space awareness). A wrong strategy can make the spacecraft not operational or create damage to any other space asset creating space garbage (named *space debris*) that may cause even more damages and debris by impacting other assets. Monitoring any issue in the controlling software before the CPS is on field for operations is fundamental in order to mitigate any possible major damage.

**INT Identified Challenges**. INT has analyzed internal mission software to provide valuable scenarios for validating the CPS change monitoring and propagation. Most of the software developed for the agency or other satellites OEM have been marked as restricted access and so no publicly available to the rest of the consortium partners. Despite these formal limitations, on some of the project ongoing has been identified some core component where it is very important to conduct traceability of changes against classified and reported bugs and malfunctions. For these assets it has been introduced in the working team a formal methodology to classify the various software modules under development against potential macro-features of the satellites (e.g. algorithm control, robustness scenarios, self-healing and Fault Detection Insulation and Recovery - FDIR -

scenarios, power management, payload data management, communication, standard misuse, external interfaces misuse).

**INT and ZHAW Objectives within COSMOS**. These data will flow into the main COSMOS goal of studying the effectiveness of the new CPS monitoring and propagation control of changes at single source of line code impacting potentially the overall safety and reliability of the overall spacecraft behavior. The COSMOS modules developed by INT and ZHAW are going to be integrated into a CI pipeline in order to detect and provide insight to the software development and VV team about the potential impact of a change that potentially introduces bugs in the system.

To make it possible to share data and information between the researchers partner and other industrial partners, it has been identified a not restricted software module that could be used for preparing a first analysis that later should be applied to the core components.

**INT Identified Scenarios**. One of the most recent projects INT is developing regards the development of onboard critical software for the Mass Memory Unit (MMU) of the Space Rider project commissioned by ESA. Space Rider is an automated robotic laboratory to be developed by Thales Alenia Space, a joint venture between Thales (67%) and Leonardo (33%), prime contractor for the ESA mission, and AVIO, which is planned to launch on Vega-C in 2023. Space Rider aims to provide Europe with an affordable, independent, reusable end-to-end integrated space transportation system for routine access and return from low orbit. It will transport payloads for an array of applications, orbit altitudes and inclinations. The Space Rider MMU module will be used for the storage of scientific data coming from the various instruments operating on the spacecraft. The laboratory will be reusable for at least 6 flights, and it is designed to stay 60 days in Earth's orbit, and then return on Earth with all the experiments data and results to be analysed by ESA scientists. The laboratory will be then prepared for a new flight.

In developing the MMU, INT proposed to ESA the reuse of a well-known open-source software library for covering some of the low-level aspects in interfacing the NAND flash memories that compose the MMU hardware. This software module is represented by YAFFS (https://yaffs.net/) an opensource filesystem implementation for NAND flash memories. YAFFS has been used recently also from NASA for one of the exploration missions in solar system.

Despite the initial optimism in reusing the software library, the agency asked INT to perform a complete qualification of the software that is not respecting the common ESA coding and engineering standards. For achieve this challenging goal, INT has to integrate the external library with the overall software ecosystem under development and implement any kind of software verification and validation (accordingly to ECSS ESA standards) that can prove the library grants safe usage of spacecraft computational resources to respect all functional and non-functional requirements regarding the new file system implementation.

The software change scenarios that INT is interested to detect are: incorrect algorithm implementation, incorrect numerical computation, mission condition checks, misuse of external interface, memory usage, invalid documentation (discrepancy of software documentation and software implementation), incorrect configurations. We will prioritize the monitoring of such aspects in the next months of the COSMOS project.

## 6.2 Monitoring and Propagation of CPS Vulnerabilities

From a technological perspective, in the context of monitoring and propagation of CPS vulnerabilities, COSMOS focuses on addressing the realization of solutions enabling the CPS vulnerability analysis and propagation. This solutions will be applied to both Open-source use cases and COSMOS specific use cases with a specific focus on identifying static vulnerabilities to use for vulnerability-proneness analysis (and potentially vulnerability propagation) in CPSs. This research will be conducted in collaboration between ZHAW and UoL.

The automation we envision aims at assessing the *vulnerability-proneness* levels of applications defined as "the number of different types of known security issues exhibited by the app". The questions we target to answer are:

| Id | Vulnerability Type | Description | # of vulnerable apps |
|---|---|---|---|
| v1 | <SSL_Security> SSL Connection Checking | The app connects to URLs that are NOT under SSL | 938 |
| v2 | <WebView> <Remote Code Execution> <#CVE-2013-4710#> WebView RCE Vulnerability Checking | The app makes use of the `addJavascriptInterface` method. This method can be used to allow JavaScript to control the host application. | 692 |
| v3 | <Implicit_Intent> Implicit Service Checking | The app uses an implicit intent to start a service. This is a security hazard because one cannot be certain what service will respond to the intent, and the user cannot see which service starts. | 420 |
| v4 | App Sandbox Permission Checking | Creating world-readable or word-writeable files is very dangerous, and likely to cause security holes in applications. It is strongly discouraged: applications should use more formal mechanism for interactions. | 279 |
| v5 | <SSL_Security> SSL Certificate Verification Checking | The app does not check the validation of SSL Certificate. This allows self-signed, expired or mismatch CN certificates for SSL connection allowing attackers to perform MITM attacks. | 201 |
| v6 | <KeyStore> <Hacker> KeyStore Protection Checking | The app uses KeyStore(s) not protected by password. | 200 |
| v7 | <Command> Runtime Command Checking | The app uses the critical function `Runtime.getRuntime().exec("...")`. This could allow an attacker to inject arbitrary system commands into the application | 191 |
| v8 | <#BID 64208 CVE-2013-6271#> Fragment Vulnerability Checking | `PreferenceActivity` class is extended without overriding the `isValidFragment` method. This could expose the app to fragment injection, when it is used on devices running a version of Android < 4.4 | 158 |
| v9 | AndroidManifest ContentProvider Exported Checking | The app uses `exported` ContentProvider, allowing any other app on the device to access it. | 156 |
| v10 | <SSL_Security> SSL Implementation Checking (Verifying Host Name in Custom Classes) | The app allows Self-defined `HOSTNAME VERIFIER` to accept all Common Names(CN). This allows attackers to do MITM attacks. | 148 |
| v11 | <SSL_Security> SSL Implementation Checking (Verifying Host Name in Fields) | The app does not check the validation of the CN(Common Name) of the SSL certificate, allowing attackers to perform MITM attacks | 114 |

Vulns that may cause MITM attacks

Vulns that may cause injection attacks

Vulns that may allow access to sensitive data

Figure 22: Examples of Types of Vulnerabilities Targeted by the Automated Monitoring of CPSs

- RQ1: Which are the different vulnerabilities exhibited by applications belonging to different application domains? Examples of vulnerabilities targeted in COSMOS are visualized in Figure 22.
- RQ2: Is it possible to predict the level of vulnerability-proneness of an app by using the app's contextual information? Here we plan to experiment with deep-learning and machine learning approach to enable the prediction of vulnerability-proneness levels (i.e., the attack surface of the systems).

For the moment, as use cases we are experimenting with a large-scale study on 5'931 firmware archives of mobile applications, with vulnerabilities identified with Androguard and QARK. Complementary, we are collecting CPS-specific datasets from COSMOS partners and open source CPS communities (e.g., the ROS ecosystem) to investigate the usage of our automation in such context. Specifically, regarding the COSMOS scenarios, the proposed approach and technology will be tested against a relevant industrial use-case emerging from Q-Media regarding the railway sector. For safety-relevant systems developed for the Railway segment all known vulnerabilities must be identified by the manufacturer before releasing the it to service. Their impacts on the system are assessed by risk analysis. If any risk is marked as greater than acceptable, mitigation measures must be applied to the identified vulnerabilities. As an example, if a penetration testing procedure identifies the presence of a non-compliant version of an OpenSSH server deployed in a certain field system, the severity level assigned to such an issue is typically Medium which is unacceptable for the area of use. Any possible mitigation or corrective action shall be considered and promptly implemented. But it also shall be possible to assess that such action will not impact any other modules and components. In the example a simple server upgrade to a higher version shall request a new assessment of any possible security impact on any other deployed software libraries or external tools integration. Overall, the *prediction of vulnerability-proneness levels* is of major benefit for Q-Media scenarios in handling product patch management.

# References


[1] Docker. https://www.docker.com (last access 05.04.2017).

[2] Log analysis using flight review | px4 user guide. https://docs.px4.io/master/en/log/flight_review.html. accessed: 07.02.2022.

[3] Nvidia drive constellation, Dec 2020.

[4] Sara Abbaspour Asadollah, Rafia Inam, and Hans Hansson. A survey on testing for cyber physical system. In Khaled El-Fakih, Gerassimos Barlas, and Nina Yevtushenko, editors, *Testing Software and Systems*, pages 194–207, Cham, 2015. Springer Intern. Publishing.

[5] Afsoon Afzal. *Automated Testing of Robotic and Cyberphysical Systems*. PhD thesis, Carnegie Mellon University, 2021.

[6] Afsoon Afzal, Deborah S Katz, Claire Le Goues, and Christopher S Timperley. Simulation for robotics test automation: Developer perspectives. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 263–274. IEEE, 2021.

[7] Afsoon Afzal, Claire Le Goues, Michael Hilton, and Christopher Steven Timperley. A study on challenges of testing robotic systems. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 96–107. IEEE, 2020.

[8] Miguel Alcon, Hamid Tabani, Jaume Abella, and Francisco J. Cazorla. Enabling unit testing of already-integrated AI software systems: The case of apollo for autonomous driving. In Francesco Leporati, Salvatore Vitabile, and Amund Skavhaug, editors, *24th Euromicro Conference on Digital System Design, DSD 2021, Palermo, Spain, September 1-3, 2021*, pages 426–433. IEEE, 2021.

[9] Carol V. Alexandru, Sebastiano Panichella, Sebastian Proksch, and Harald C. Gall. Redundancy-free analysis of multi-revision software artifacts. *Empir. Softw. Eng.*, 24(1):332–380, 2019.

[10] Tom Arbuckle. Measuring multi-language software evolution: A case study. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, pages 91–95, 2011.

[11] Ardupilot.org. Open source drone software. versatile, trusted, open. ardupilot, 2021.

[12] Radhakisan Baheti and Helen Gill. Cyber-physical systems. *The impact of control technology*, 12(1):161–166, 2011.

[13] Eulalia Balestrieri, Pasquale Daponte, Luca De Vito, Francesco Picariello, and Ioan Tudosa. Sensors and measurements for UAV safety: An overview. *Sensors*, 21(24):8253, 2021.

[14] Martin Barczyk and Alan F. Lynch. Integration of a triaxial magnetometer into a helicopter uav gps-aided ins. *IEEE Transactions on Aerospace and Electronic Systems*, 48(4):2947–2960, 2012.

[15] Ezio Bartocci, Jyotirmoy Deshmukh, Alexandre Donzé, Georgios Fainekos, Oded Maler, Dejan Ničković, and Sriram Sankaranarayanan. *Specification-Based Monitoring of Cyber-Physical Systems: A Survey on Theory, Tools and Applications*, pages 135–175. Springer, 2018.

[16] BeamNG GmbH. BeamNG.tech. https://www.beamng.gmbh/research. Accessed: 2018-10-11.

[17] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. Deflaker: Automatically detecting flaky tests. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 433–444. IEEE, 2018.

[18] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. Modern code reviews in open-source projects: which problems do they fix? In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 202–211, 2014.

[19] Hans Christian Benestad, Bente Anda, and Erik Arisholm. Understanding software maintenance and evolution by analyzing individual changes: a literature review. *Journal of Software Maintenance and Evolution: Research and Practice*, 21(6):349–378, 2009.

[20] Jennifer Bevan, E. James Whitehead, Jr., Sunghun Kim, and Michael Godfrey. Facilitating software evolution research with kenyon. pages 177–186, 2005.

[21] M. E. R. Bezerra, A. L. I. Oliveira, and S. R. L. Meira. A constructive rbf neural network for estimating the probability of defects in software modules. In *2007 International Joint Conference on Neural Networks*, pages 2869–2874, 2007.

[22] Christian Birchler, Nicolas Ganz, Sajad Khatiri, Alessio Gambi, and Sebastiano Panichella. Cost-effective simulation-based test selection in self-driving cars software with sdc-scissor. In *the 29th IEEE International Conference on Software Analysis, Evolution, and Reengineering - To Appear*, 2022.

[23] Christian Birchler, Nicolas Ganz, Sajad Khatiri, Alessio Gambi, and Sebastiano Panichella. Cost-effective simulation-based test selection in self-driving cars software with sdc-scissor. In *29th IEEE International Conference on Software Analysis, Evolution, and Reengineering, Honolulu, USA (online), 15-18 March 2022*. ZHAW Zürcher Hochschule für Angewandte Wissenschaften, 2022.

[24] Christian Birchler, Sajad Khatiri, Pouria Derakhshanfar, Sebastiano Panichella, and Annibale Panichella. Automated test cases prioritization for self-driving cars in virtual environments. *arXiv preprint arXiv:2107.09614*, 2021.

[25] Christian Birchler, Sajad Khatiri, Pouria Derakhshanfar, Sebastiano Panichella, and Annibale Panichella. Single and multi-objective test cases prioritization for self-driving cars in virtual environments. *CoRR*, abs/2107.09614, 2021.

[26] Adriano Bittar, Helosman V. Figuereido, Poliana Avelar Guimaraes, and Alessandro Correa Mendes. Guidance software-in-the-loop simulation using x-plane and simulink for uavs. In *2014 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 993–1002, 2014.

[27] Kinga Bojarczuk, Natalija Gucevska, Simon M. M. Lucas, Inna Dvortsova, Mark Harman, Erik Meijer, Silvia Sapora, Johann George, Maria Lomeli, and Rubmary Rojas. Measurement challenges for cyber cyber digital twins: Experiences from the deployment of facebook's WW simulation system. In Filippo Lanubile, Marcos Kalinowski, and Maria Teresa Baldassarre, editors, *ESEM '21: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Bari, Italy, October 11-15, 2021*, pages 2:1–2:10. ACM, 2021.

[28] Elizabeth Bondi, Debadeepta Dey, Ashish Kapoor, Jim Piavis, Shital Shah, Fei Fang, Bistra Dilkina, Robert Hannaford, Arvind Iyer, Lucas Joppa, and Milind Tambe. AirSim-w: A simulation environment for wildlife conservation with uavs. In Ellen W. Zegura, editor, *Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies, COMPASS*, pages 40:1–40:12. ACM, 2018.

[29] Amiangshu Bosu, Michaela Greiler, and Christian Bird. Characteristics of useful code reviews: An empirical study at microsoft. In *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*, pages 146–156, 2015.

[30] Carlos Carbone, Dario Albani, Federico Magistri, Dimitri Ognibene, Cyrill Stachniss, Gert Kootstra, Daniele Nardi, and Vito Trianni. Monitoring and mapping of crop fields with UAV swarms based on

information gain. In Fumitoshi Matsuno, Shun-Ichi Azuma, and Masahito Yamamoto, editors, *Distributed Autonomous Robotic Systems - 15th International Symposium, DARS 2021, June 1-4, 2021, Online Event*, volume 22 of *Springer Proceedings in Advanced Robotics*, pages 306–319. Springer, 2021.

[31] Rich Caruana and Alexandru Niculescu-mizil. An empirical comparison of supervised learning algorithms. In *In Proc. 23 rd Intl. Conf. Machine learning (ICML'06*, pages 161–168, 2006.

[32] Ezequiel Castellano, Ahmet Cetinkaya, Cédric Ho Thanh, Stefan Klikovits, Xiaoyi Zhang, and Paolo Arcaini. Frenetic at the SBST 2021 tool competition. In *International Workshop on Search-Based Software Testing*, pages 36–37. IEEE, 2021.

[33] Hong Chen. Applications of cyber-physical system: A literature review. *Journal of Industrial Integration and Management*, 02(03):1750012, 2017.

[34] Chih-Hong Cheng, Alois C. Knoll, and Hsuan-Cheng Liao. Safety metrics for semantic segmentation in autonomous driving. In *2021 IEEE International Conference on Artificial Intelligence Testing, AITest 2021, Oxford, United Kingdom, August 23-26, 2021*, pages 57–64. IEEE, 2021.

[35] Romain Chiappinelli and Hamish Willee. Px4 user guide: Architectural overview. `https://docs.px4.io/master/en/concept/architecture.html`, 2021. (Last access: 10/03/2022).

[36] Romain Chiappinelli, Hamish Willee, Michael Murphy, Morten Fyhn Amundsen, Jaeyoung Lim, and Peter van der Perk. Px4 user guide: Simulation. `https://docs.px4.io/master/en/simulation/`, 2021. (Last access: 10/03/2022).

[37] Sudipta Chowdhury, Omid Shahvari, Mohammad Marufuzzaman, Xiaopeng Li, and Linkan Bian. Drone routing and optimization for post-disaster inspection. *Comput. Ind. Eng.*, 159:107495, 2021.

[38] Jane Cleland-Huang and Michael Vierhauser. Discovering, analyzing, and managing safety stories in agile projects. In *26th IEEE International Requirements Engineering Conference, RE 2018, Banff, AB, Canada, August 20-24, 2018*, pages 262–273, 2018.

[39] K. Czarnecki. Requirements engineering in the age of societal-scale cyber-physical systems: The case of automated driving. In *2018 IEEE 26th International Requirements Engineering Conference (RE)*, pages 3–4, Aug 2018.

[40] Raffaello D'Andrea. Guest editorial can drones deliver? *IEEE Trans Autom. Sci. Eng.*, 11(3):647–648, 2014.

[41] L. Deruelle, N. Melab, M. Bouneffa, and H. Basson. Analysis and manipulation of distributed multi-language software code. In *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 43–54, 2001.

[42] Andrea Di Sorbo, Sebastiano Panichella, Carol V. Alexandru, Junji Shimagaki, Corrado Aaron Visaggio, Gerardo Canfora, and Harald C. Gall. What would users change in my app? Summarizing app reviews for recommending software changes. In *Proc. Int'l Symposium on Foundations of Software Engineering (FSE)*, pages 499–510, 2016.

[43] Alexey Dosovitskiy, Germán Ros, Felipe Codevilla, Antonio M. López, and Vladlen Koltun. CARLA: an open urban driving simulator. In *1st Annual Conference on Robot Learning, CoRL 2017*, volume 78 of *Proceedings of Machine Learning Research*, pages 1–16. PMLR, 2017.

[44] Saikat Dutta, August Shi, Rutvik Choudhary, Zhekun Zhang, Aryaman Jain, and Sasa Misailovic. Detecting flaky tests in probabilistic and machine learning applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 211–224, 2020.

[45] Robert Dyer. *Bringing Ultra-large-scale Software Repository Mining to the Masses with Boa*. PhD thesis, Ames, IA, USA, 2013. AAI3610634.

[46] Robert Dyer, Hridesh Rajan, and Tien N. Nguyen. Declarative visitors to ease fine-grained source code mining with full history on billions of ast nodes. pages 23–32, 2013.

[47] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. Understanding flaky tests: The developer's perspective. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 830–840, 2019.

[48] Davide Falanga, Philipp Foehn, Peng Lu, and Davide Scaramuzza. PAMPC: perception-aware model predictive control for quadrotors. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2018, Madrid, Spain, October 1-5, 2018*, pages 1–8. IEEE, 2018.

[49] Michael Felderer, Matthias Büchler, Martin Johns, Achim D. Brucker, Ruth Breu, and Alexander Pretschner. Chapter one - security testing: A survey. *Advances in Computers*, 101:1–51, 2016.

[50] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, 2011.

[51] Eric Feron and Eric N Johnson. *Aerial Robotics.* Springer, 2008.

[52] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 23–32, 2003.

[53] Beat Fluri, Michael Wuersch, Martin Pinzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33(11), 2007.

[54] Christian Forster, Matia Pizzoli, and Davide Scaramuzza. Svo: Fast semi-direct monocular visual odometry. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 15–22, 2014.

[55] Fortune Business Insights. Market research report: Unmanned aerial vehicle (uav) market size, share & covid-19 impact analysis, by class (small uavs, tactical uavs, and strategic uavs), by technology (remotely operated, semi-autonomous, and fully-autonomous), by system (uav airframe, uav payloads, uav avionics, uav propulsion, and uav software), by application (military, commercial and recreational), and regional forecast, 2020-2027. `https://www.fortunebusinessinsights.com/industry-reports/unmanned-aerial-vehicle-uav-market-101603`, 2020.

[56] Ivo Friedberg, Kieran McLaughlin, Paul Smith, David Laverty, and Sakir Sezer. Stpa-safesec: Safety and security analysis for cyber-physical systems. *Journal of Information Security and Applications*, 34:183 – 196, 2017.

[57] Fadri Furrer, Michael Burri, Markus Achtelik, and Roland Siegwart. Rotors—a modular gazebo mav simulator framework. In *Robot operating system (ROS)*, pages 595–625. Springer, 2016.

[58] H.C. Gall, B. Fluri, and M. Pinzger. Change analysis with evolizer and changedistiller. *Software, IEEE*, 26(1):26–33, 2009.

[59] Alessio Gambi, Marc Mueller, and Gordon Fraser. AsFault: Testing self-driving car software using search-based procedural content generation. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, May 2019.

[60] Jack Ganssle. *The Firmware handbook*. Embedded technology series. Newnes, Amsterdam ;, 1st edition edition, 2004.

[61] Daniel M. Germán, Gregorio Robles, Germán Poo-Caamaño, Xin Yang, Hajimu Iida, and Katsuro Inoue. "was my contribution fairly reviewed?": a framework to study the perception of fairness in modern code reviews. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 523–534, 2018.

[62] Carina Gerstenberger and Daniel Vogel. On the efficiency of gini's mean difference. *Statistical Methods & Applications*, 24(4):569–596, 2015.

[63] Wojciech Giernacki, Mateusz Skwierczynski, Wojciech Witwicki, Pawel Wronski, and Piotr Kozierski. Crazyflie 2.0 quadrotor as a platform for research and education in robotics and control engineering. In *22nd International Conference on Methods and Models in Automation and Robotics, MMAR 2017, Międzyzdroje, Poland, August 28-31, 2017*, pages 37–42. IEEE, 2017.

[64] Renato Giorgiani do Nascimento, Kajetan Fricke, and Felipe Viana. Quadcopter control optimization through machine learning. In *AIAA Scitech 2020 Forum*, page 1148, 2020.

[65] Jairo Giraldo, Esha Sarkar, Alvaro A. Cardenas, Michail Maniatakos, and Murat Kantarcioglu. Security and privacy in cyber-physical systems: A survey of surveys. *IEEE Design & Test*, 34(4):7–17, August 2017.

[66] Carlos A. González, Mojtaba Varmazyar, Shiva Nejati, Lionel C. Briand, and Yago Isasi. Enabling model testing of cyber-physical systems. In *Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MODELS)*, pages 176–186. ACM, 2018.

[67] Giovanni Grano, Adelina Ciurumelea, Sebastiano Panichella, Fabio Palomba, and Harald Gall. Exploring the integration of user feedback in automated testing of Android applications. In *Int'l Conf. on Software Analysis, Evolution and Reengineering*, 2018.

[68] The Guardian. Self-driving uber kills arizona woman in first fatal crash involving pedestrian, 2018.

[69] R. Hadjidj, X. Yang, S. Tlili, and M. Debbabi. Model-checking for software vulnerabilities detection with multi-language support. In *2008 Sixth Annual Conference on Privacy, Security and Trust*, pages 133–142, Oct 2008.

[70] Abigail R Hall and Christopher J Coyne. The political economy of drones. *Defence and Peace Economics*, 25(5):445–460, 2014.

[71] Roee Hay, Omer Tripp, and Marco Pistoia. Dynamic detection of inter-application communication vulnerabilities in android. In *International Symposium on Software Testing and Analysis*, pages 118–128, 2015.

[72] Philipp Helle, Wladimir Schamai, and Carsten Strobel. Testing of autonomous systems - challenges and current state-of-the-art. *INCOSE International Symposium*, pages 571–584, 2016.

[73] Tin Kam Ho. The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):832–844, 1998.

[74] Ziqi Huang, Yang Shen, Jiayi Li, Marcel Fey, and Christian Brecher. A survey on ai-driven digital twins in industry 4.0: Smart manufacturing and advanced robotics. *Sensors*, 21(19):6340, 2021.

[75] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. Taxonomy of real faults in deep learning systems. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 1110–1121. ACM, 2020.

[76] Docker Inc. Docker - what is a container?, 2022. (Last access: 28/02/2022).

[77] Félix Ingrand. Recent trends in formal validation and verification of autonomous robots software. In *3rd IEEE International Conference on Robotic Computing, IRC 2019, Naples, Italy, February 25-27, 2019*, pages 321–328, 2019.

[78] Mario Janke and Patrick Mader. Graph based mining of code change patterns from version control commits. *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

[79] Japan Times. Candy-carrying drone crashes into crowd, injuring six in gifu. `https://www.japantimes.co.jp/news/2017/11/05/national/candy-carrying-drone-crashes-crowd-injuring-six-gifu/`, Nov 2017. (Last access: 28/03/2022).

[80] Matthieu Jimenez, Mike Papadakis, Tegawendé F. Bissyandé, and Jacques Klein. Profiling android vulnerabilities. In *International Conference on Software Quality, Reliability and Security*, pages 222–229, 2016.

[81] Dr. Suzette Johnson, Harry Koehneman, Diane LaFortune, Dean Leffingwell, Stephen Magill, Steve Mayner, Avigail Ofer, Robert Stroud, Anders Wallgren, and Robin Yeman. *INDUSTRIAL DEVOPS - Applying DevOps and Continuous Delivery to Significant Cyber-Physical Systems*. National Academies Press, 2017.

[82] Nidhi Kalra and Susan Paddock. Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability? *Transportation Research Part A: Policy and Practice*, 94:182–193, 12 2016.

[83] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 805–824, New York, NY, USA, 2011. ACM.

[84] A. Kaur and R. Malhotra. Application of random forest in predicting fault-prone classes. In *2008 International Conference on Advanced Computer Theory and Engineering*, pages 37–43, 2008.

[85] James R Kellner, John Armston, Markus Birrer, KC Cushman, Laura Duncanson, Christoph Eck, Christoph Falleger, Benedikt Imbach, Kamil Krůl, Martin Krůček, et al. New opportunities for forest remote sensing through ultra-high-density drone lidar. *Surveys in Geophysics*, 40(4):959–977, 2019.

[86] R. Khan, K. McLaughlin, D. Laverty, and S. Sezer. Stride-based threat modeling for cyber-physical systems. In *2017 IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe)*, pages 1–6, Sep. 2017.

[87] Siciliano Khatib. *Springer Handbook of Robotics*. Springer Verlag, Berlin, 2nd edition edition, 2016.

[88] Sajad Khatiri, Christian Birchler, Bill Bosshard, Alessio Gambi, and Sebastiano Panichella. Machine learning-based test selection for simulation-based testing of self-driving cars software, 2021.

[89] Sajad Khatiri, Christian Birchler, Bill Bosshard, Alessio Gambi, and Sebastiano Panichella. Machine learning-based test selection for simulation-based testing of self-driving cars software. *arXiv preprint arXiv:2111.04666*, 2021.

[90] Jiseob Kim, Sunil Chon, and Jihwan Park. Suggestion of testing method for industrial level cyber-physical system in complex environment. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, April 2019.

[91] Moon Seong William Kim, Ki Hoon Jacob Han, and Jeong Heon. Investigating the aerodynamics of flight for multiple rotor drones. *Analysis of Applied Mathematics*, 1:93, 2017.

[92] Katharine Hall Kindervater. The emergence of lethal surveillance: Watching and killing in the history of drone technology. *Security Dialogue*, 47(3):223–238, 2016.

[93] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2149–2154 vol.3, 2004.

[94] Nathan P. Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems, Sendai, Japan, September 28 - October 2, 2004*, pages 2149–2154. IEEE, 2004.

[95] Oleksii Kononenko, Olga Baysal, and Michael W. Godfrey. Code review quality: how developers see it. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 1028–1038, 2016.

[96] Kostas Kontogiannis, Panagiotis K. Linos, and Kenny Wong. Comprehension and maintenance of large-scale multi-language software applications. In *22nd IEEE International Conference on Software Maintenance (ICSM 2006), 24-27 September 2006, Philadelphia, Pennsylvania, USA*, pages 497–500, 2006.

[97] Anis Koubâa, Azza Allouch, Maram Alajlan, Yasir Javed, Abdelfettah Belghith, and Mohamed Khalgui. Micro air vehicle link (mavlink) in a nutshell: A survey. *IEEE Access*, 7:87658–87680, 2019.

[98] Kubernetes. Kubernetes, 2022.

[99] Francis J Lacoste. Killing the gatekeeper: Introducing a continuous integration system. In *2009 agile conference*, pages 387–392. IEEE, 2009.

[100] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. Root causing flaky tests in a large-scale industrial setting. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 101–111, 2019.

[101] Wing Lam, Kıvanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. A study on the lifecycle of flaky tests. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1471–1482, 2020.

[102] Wing Lam, Stefan Winter, Anjiang Wei, Tao Xie, Darko Marinov, and Jonathan Bell. A large-scale longitudinal study of flaky tests. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

[103] P. G. Larsen, J. Fitzgerald, J. Woodcock, P. Fritzson, J. Brauer, C. Kleijn, T. Lecomte, M. Pfeil, O. Green, S. Basagiannis, and A. Sadovykh. Integrated tool chain for model-based design of cyber-physical systems: The into-cps project. In *2016 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data)*, pages 1–6, April 2016.

[104] Wei Le and Shannon D. Pattison. Patch verification via multiversion interprocedural control flow graphs. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1047–1058, New York, NY, USA, 2014. ACM.

[105] Xiaozhou Liang, John Henry Burns, Joseph Sanchez, Karthik Dantu, Lukasz Ziarek, and Yu David Liu. Understanding bounding functions in safety-critical uav software. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1311–1322. IEEE, 2021.

[106] Mikael Lindvall, Adam Porter, Gudjon Magnusson, and Christoph Schulze. Metamorphic model-based testing of autonomous systems. In *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*, pages 35–41. IEEE, 2017.

[107] Antonio Loquercio, Elia Kaufmann, René Ranftl, Matthias Müller, Vladlen Koltun, and Davide Scaramuzza. Learning high-speed flight in the wild. *Science Robotics*, 6(59):eabg5810, 2021.

[108] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: statically vetting android apps for component hijacking vulnerabilities. In *the ACM Conference on Computer and Communications Security*, pages 229–240, 2012.

[109] Yuncheng Lu, Zhucun Xue, Gui-Song Xia, and Liangpei Zhang. A survey on vision-based uav navigation. *Geo-spatial information science*, 21(1):21–32, 2018.

[110] Yuriy Zacchia Lun, Alessandro D?Innocenzo, Francesco Smarra, Ivano Malavolta, and Maria Domenica Di Benedetto. State of the art of cyber-physical systems security: An automatic control perspective. *Journal of Systems and Software*, 149:174 – 216, 2019.

[111] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 643–653, 2014.

[112] Ruchika Malhotra and Ankita Jain Bansal. Software change prediction: a literature review. *Int. J. Comput. Appl. Technol.*, 54(4):240–256, 2016.

[113] Mika Mäntylä and Casper Lassenius. What types of defects are really discovered in code reviews? *IEEE Trans. Software Eng.*, 35(3):430–448, 2009.

[114] Fernando Martinez-Taboada and Jose Ignacio Redondo. The siesta (seaav integrated evaluation sedation tool for anaesthesia) project: Initial development of a multifactorial sedation assessment tool for dogs. *PLOS ONE*, 15(4):1–10, 04 2020.

[115] Kimberly McGuire, Guido de Croon, Christophe De Wagter, Karl Tuyls, and Hilbert Kappen. Efficient optical flow and stereo vision for velocity estimation and obstacle avoidance on an autonomous pocket drone. *IEEE Robotics and Automation Letters*, 2(2):1070–1076, 2017.

[116] Lorenz Meier. Software overview. `https://px4.io/software/software-overview/`, Jul 2021.

[117] Lorenz Meier, Dominik Honegger, and Marc Pollefeys. Px4: A node-based multithreaded open source robotics framework for deeply embedded platforms. In *2015 IEEE international conference on robotics and automation (ICRA)*, pages 6235–6240. IEEE, 2015.

[118] Lorenz Meier, Dominik Honegger, and Marc Pollefeys. PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms. In *IEEE International Conference on Robotics and Automation, ICRA 2015, Seattle, WA, USA, 26-30 May, 2015*, pages 6235–6240. IEEE, 2015.

[119] N. J. Muga and A. N. Pinto. Extended kalman filter vs. geometrical approach for stokes space based polarization demultiplexing. *Journal of Lightwave Technology*, 33(23):4826–4833, December 2015.

[120] RR Murphy. *Introduction to AI Roboticsel. 1st*. Cambridge, MA, USA: MIT Press, 2000.

[121] Drohne krachte in zürich auf den boden - post bezeichnet vorfall als inakzeptabel. `https://www.nzz.ch/zuerich/absturz-von-post-drohne-bericht-stellt-gravierende-maengel-fest-ld.1492295?reduced=true`. Accessed: 2022-02-08.

[122] 2 killed in driverless tesla car crash, officials say. `https://www.nytimes.com/2021/04/18/business/tesla-fatal-crash-texas.html`. Accessed: 2022-01-28.

[123] Phu H. Nguyen, Shaukat Ali, and Tao Yue. Model-based security engineering for cyber-physical systems: A systematic mapping study. *Information and Software Technology*, 83:116 – 135, 2017.

[124] Vuong Nguyen, Stefan Huber, and Alessio Gambi. SALVO: automated generation of diversified tests for self-driving cars from existing maps. In *2021 IEEE International Conference on Artificial Intelligence Testing, AITest 2021, Oxford, United Kingdom, August 23-26, 2021*, pages 128–135. IEEE, 2021.

[125] Helen Oleynikova, Zachary Taylor, Marius Fehr, Roland Siegwart, and Juan I. Nieto. Voxblox: Incremental 3d euclidean signed distance fields for on-board MAV planning. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2017, Vancouver, BC, Canada, September 24-28, 2017*, pages 1366–1373. IEEE, 2017.

[126] Fabio Palomba and Andy Zaidman. Notice of retraction: Does refactoring of test smells induce fixing flaky tests? In *2017 IEEE international conference on software maintenance and evolution (ICSME)*, pages 1–12. IEEE, 2017.

[127] Sebastiano Panichella, Alessio Gambi, Fiorella Zampetti, and Vincenzo Riccio. Sbst tool competition 2021. In *International Conference on Software Engineering, Workshops, Madrid, Spain, 2021*. ACM, 2021.

[128] Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado Aaron Visaggio, Gerardo Canfora, and Harald C. Gall. How can i improve my app? classifying user reviews for software maintenance and evolution. In Rainer Koschke, Jens Krinke, and Martin P. Robillard, editors, *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, pages 281–290. IEEE Computer Society, 2015.

[129] Sebastiano Panichella and Nik Zaugg. An empirical investigation of relevant changes and automation needs in modern code review. *Empir. Softw. Eng.*, 25(6):4833–4872, 2020.

[130] Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. A survey of flaky tests. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(1):1–74, 2021.

[131] Scott Pendleton, Hans Andersen, Xinxin Du, Xiaotong Shen, Malika Meghjani, You Eng, Daniela Rus, and Marcelo Jr. Perception, planning, control, and coordination for autonomous vehicles. *Machines*, 5:6, 02 2017.

[132] Andrea Piazzoni, Jim Cherian, Mohamed Azhar, Jing Yew Yap, James Lee Wei Shung, and Roshan Vijay. Vista: a framework for virtual scenario-based testing of autonomous vehicles. In *2021 IEEE International Conference on Artificial Intelligence Testing, AITest 2021, Oxford, United Kingdom, August 23-26, 2021*, pages 143–150. IEEE, 2021.

[133] Piotr Esden-Tempski. Paparazziuav. `https://wiki.paparazziuav.org/wiki/Main_Page`, 2022. (Last access: 20/03/2022).

[134] Pixhawk.org. Pixhawk | the hardware standard for open-source autopilots., 2021.

[135] John Porter. Swiss drone crashes near children, forcing suspension of delivery program. `https://www.theverge.com/2019/8/2/20751383/swiss-drone-crash-delivery-program-suspended-matternet-post-hospital-samples/`, Aug 2019. (Last access: 28/02/2022).

[136] PwC. Insights for telecom, cable, satellite, and internet executives. *Communications Review / July 2017*, 2017.

[137] PX4.io. Obstacle avoidance | px4 user guide, 2022. accessed: 07.02.2022.

[138] Babak Bashari Rad, Harrison John Bhatti, and Mohammad Ahmadi. An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*, 17(3):228, 2017.

[139] Gordana Rakić, Zoran Budimac, and Miloš Savić. Language independent framework for static code analysis. In *Proceedings of the 6th Balkan Conference in Informatics*, BCI '13, pages 236–243, New York, NY, USA, 2013. ACM.

[140] S. U. Rehman, C. Allgaier, and V. Gruhn. Security requirements engineering: A framework for cyber-physical systems. In *2018 International Conference on Frontiers of Information Technology (FIT)*, pages 315–320, Dec 2018.

[141] Paul Riseborough. Px4 state estimation. In *PX4 Developer Summit Zurich 2019*, 2019. (Last access: 28/02/2022).

[142] Alan Romano, Zihe Song, Sampath Grandhi, Wei Yang, and Weihang Wang. An empirical analysis of ui-based flaky tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1585–1597. IEEE, 2021.

[143] Debayan Roy, Clara Hobbs, James H. Anderson, Marco Caccamo, and Samarjit Chakraborty. Timing debugging for cyber-physical systems. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2021, Grenoble, France, February 1-5, 2021*, pages 1893–1898. IEEE, 2021.

[144] Claude Sammut and Geoffrey I. Webb, editors. *Logistic Regression*, pages 631–631. Springer US, Boston, MA, 2010.

[145] F. Schloegl, S. Rohjans, S. Lehnhoff, J. Velasquez, C. Steinbrink, and P. Palensky. Towards a classification scheme for co-simulation approaches in energy systems. In *Int'l Symposium on Smart Electric Distribution Systems and Technologies (EDST)*, pages 516–521, Sep. 2015.

[146] Post-drohne über zürichsee abgestürzt. `https://www.srf.ch/news/regional/zuerich-schaffhausen/blutprobe-verloren-post-drohne-ueber-zuerichsee-abgestuerzt`. Accessed: 2022-02-08.

[147] August Shi, Jonathan Bell, and Darko Marinov. Mitigating the effects of flaky tests on mutation testing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 112–122, 2019.

[148] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. ifixflakies: A framework for automatically fixing order-dependent flaky tests. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 545–555, 2019.

[149] Seung Yeob Shin, Karim Chaouch, Shiva Nejati, Mehrdad Sabetzadeh, Lionel C. Briand, and Frank Zimmer. Hitecs: A uml profile and analysis framework for hardware-in-the-loop testing of cyber physical systems. In *Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MODELS)*, pages 357–367. ACM, 2018.

[150] H. Shokry and M. Hinchey. Model-based verification of embedded software. *Computer*, 42(4):53–59, April 2009.

[151] Denini Silva, Leopoldo Teixeira, and Marcelo d'Amorim. Shake it! detecting flaky tests caused by concurrency with shaker. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 301–311. IEEE, 2020.

[152] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D. Breaux, and Jianwei Niu. Toward a framework for detecting privacy policy violations in android application code. In Laura K. Dillon, Willem Visser, and Laurie Williams, editors, *International Conference on Software Engineering*, pages 25–36. ACM, 2016.

[153] Simón C. Smith and Subramanian Ramamoorthy. Attainment regions in feature-parameter space for high-level debugging in autonomous robots. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2021, Prague, Czech Republic, September 27 - Oct. 1, 2021*, pages 6546–6551. IEEE, 2021.

[154] Andrea Di Sorbo and Sebastiano Panichella. Exposed! A case study on the vulnerability-proneness of google play apps. *Empir. Softw. Eng.*, 26(4):78, 2021.

[155] Andrea Stocco, Michael Weiss, Marco Calzana, and Paolo Tonella. Misbehaviour prediction for autonomous driving systems. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 359–371. ACM, 2020.

[156] D. Strein, H. Kratz, and W. Lowe. Cross-language program analysis and refactoring. In *Source Code Analysis and Manipulation, 2006. SCAM '06. Sixth IEEE International Workshop on*, pages 207–216, Sept 2006.

[157] J.M. Sullivan. Evolution or revolution? the rise of uavs. *IEEE Technology and Society Magazine*, 25(3):43–49, 2006.

[158] After deadly 737 max crashes, damning whistleblower report reveals sidelined engineers, scarcity of expertise, more. `https://www.theregister.com/2021/12/15/boeing_737_max_senate_report/`. Accessed: 2022-01-28.

[159] The-Washington-Post. Uber's radar detected elaine herzberg nearly 6 seconds before she was fatally struck, but "the system design did not include a consideration for jaywalking pedestrians" so it didn't react as if she were a person., 2019.

[160] Daniel R. Thomas, Alastair R. Beresford, Thomas Coudray, Tom Sutcliffe, and Adrian Taylor. The lifetime of android API vulnerabilities: Case study on the javascript-to-java interface. In *Security Protocols XXIII - 23rd International Workshop*, pages 126–138, 2015.

[161] Adam Thorne. Drone crashes through darling harbour window, injuring occupant. `https://australianaviation.com.au/2021/01/drone-crashes-through-darling-harbour-window-injuring-occupant/`, Jan 2021. (Last access: 28/02/2022).

[162] Swapna Thorve, Chandani Sreshtha, and Na Meng. An empirical study of flaky tests in android apps. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 534–538. IEEE, 2018.

[163] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz. A meta-model for language-independent refactoring. In *Principles of Software Evolution, 2000. Proceedings. International Symposium on*, pages 154–164, 2000.

[164] Christopher Steven Timperley, Afsoon Afzal, Deborah S. Katz, Jam Marcos Hernandez, and Claire Le Goues. Crashing simulated planes is cheap: Can simulation detect robotics bugs early? In *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*, pages 331–342. IEEE Computer Society, 2018.

[165] Martin Törngren and Ulf Sellgren. *Complexity Challenges in Development of Cyber-Physical Systems*, pages 478–503. Springer International Publishing, Cham, 2018.

[166] Chris Cole Drone Wars UK. Accidents will happen - a review of military drone crash data as the uk considers allowing large military drone flights in its airspace, 2019.

[167] Hamish Willee Peter van der Perk. Px4 user guide: Architectural overview. https://docs.px4.io/master/en/advanced_config/parameters.html, 2021. (Last access: 10/03/2022).

[168] Mario Linares Vásquez, Gabriele Bavota, and Camilo Escobar-Velasquez. An empirical study on android-related vulnerabilities. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 2–13, 2017.

[169] Dinghua Wang, Shuqing Li, Guanping Xiao, Yepang Liu, and Yulei Sui. An exploratory study of autopilot software bugs in unmanned aerial vehicles. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 20–31, 2021.

[170] Huacheng Wang, Yanxia Jiang, and Hui Wang. Stock return prediction based on bagging-decision tree. In *2009 IEEE International Conference on Grey Systems and Intelligent Services (GSIS 2009)*, pages 1575–1580, 2009.

[171] Tesla driver faces felony charges in fatal crash involving autopilot. https://www.washingtonpost.com/technology/2022/01/20/tesla-autopilot-charges/. Accessed: 2022-01-28.

[172] Takuya Watanabe, Mitsuaki Akiyama, Fumihiro Kanei, Eitaro Shioji, Yuta Takata, Bo Sun, Yuta Ishii, Toshiki Shibahara, Takeshi Yagi, and Tatsuya Mori. Understanding the origins of mobile app vulnerabilities: a large-scale measurement study of free and paid apps. In *International Conference on Mining Software Repositories*, pages 14–24, 2017.

[173] Chathurika S. Wickramasinghe, Daniel L. Marino, Kasun Amarasinghe, and Milos Manic. Generalization of deep learning for cyber-physical system security: A survey. In *44th Annual Conference of the IEEE Industrial Electronics Society (IECON)*, pages 745–751, 2018.

[174] Hamish Willee. Px4: uorb messaging. https://docs.px4.io/master/en/middleware/uorb.html, 2015. (Last access: 28/02/2022).

[175] Hamish Willee. Px4 user guide: Controller diagrams. https://dev.px4.io/v1.11_noredirect/en/flight_stack/controller_diagrams.html, 2021. (Last access: 10/03/2022).

[176] Anna Wojciechowska, Jeremy Frey, Sarit Sass, Roy Shafir, and Jessica R Cauchard. Collocated human-drone interaction: Methodology and approach strategy. In *2019 14th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pages 172–181. IEEE, 2019.

[177] Trey Woodlief, Sebastian Elbaum, and Kevin Sullivan. Fuzzing mobile robot environments for fast automated crash detection. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5417–5423. IEEE, 2021.

[178] Franz Wotawa. On the use of available testing methods for verification & validation of ai-based software and systems. In Huáscar Espinoza, John McDermid, Xiaowei Huang, Mauricio Castillo-Effen, Xin Cynthia Chen, José Hernández-Orallo, Seán Ó hÉigeartaigh, and Richard Mallah, editors, *Proceedings of the Workshop on Artificial Intelligence Safety 2021 (SafeAI 2021) co-located with the Thirty-Fifth AAAI Conference on Artificial Intelligence (AAAI 2021), Virtual, February 8, 2021*, volume 2808 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2021.

[179] Daoyuan Wu and Rocky K. C. Chang. Analyzing android browser apps for file: // vulnerabilities. In *Information Security - International Conference*, pages 345–363, 2014.

[180] A. Yoganandhan, S.D. Subhash, J. Hebinson Jothi, and V. Mohanavel. Fundamentals and development of self-driving cars. *Materials Today: Proceedings*, 33:3303–3310, 2020. International Conference on Nanotechnology: Ideas, Innovation and Industries.

[181] Xuejun Zhang, Yang Liu, Yu Zhang, Xiangmin Guan, Daniel Delahaye, and Li Tang. Safety assessment and risk estimation for unmanned aerial vehicles operating in national airspace system. *Journal of Advanced Transportation*, 2018.

[182] Hao Zhong and Na Meng. Towards reusing hints from past fixes - an exploratory study on thousands of real samples. *Empirical Software Engineering*, 23(5):2521–2549, 2018.

[183] Celal Ziftci and Diego Cavalcanti. De-flake your tests: Automatically locating root causes of flaky tests in code at google. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 736–745. IEEE, 2020.

[184] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *Software Engineering, IEEE Transactions on*, 31(6):429–445, 2005.