ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-TUBG006

HUNTING FOR HIDDEN BUGS: DEALING WITH TEST FLAKINESS IN SKA CONTROL SOFTWARE

G. Marotta*, C. Baffa, E. Giani, INAF-OAA, Firenze, Italy
S. Di Frischia, INAF-OAAb, Teramo, Italy
I. Novak, M. Colciago, Cosylab Switzerland, Brugg, Switzerland
G. Brajnik[1], E. Lena, Interactive Design Solutions Srl, Udine, Italy
[1]also at University of Udine, Udine, Italy

Abstract

Test flakiness—when a test intermittently passes or fails without changes to the code—poses a significant challenge in the validation of distributed control systems. This paper presents an investigation into test flakiness in CSP.LMC (Local Monitoring and Control for the Central Signal Processor), a key subsystem of the SKA (Square Kilometre Array) telescope. CSP.LMC is a Python application built on the TANGO framework, that is tested using a multi-level testing approach combining unit, component, and integration tests. To achieve scalable and reproducible deployment, the entire SKA control software runs within a Kubernetes environment. We systematically collect test outcomes and execution benchmarks to monitor system stability over time. A data mining approach is applied to uncover correlations and hidden patterns associated with test instability. Our analysis aims to uncover subtle software issues that are not easily detected through standard test evaluation. Furthermore, we aim to explore how the complexity of both the software architecture and its deployment may introduce sources of non-determinism that can lead to flaky tests. We discuss the impact of flakiness on the reliability of SKA control software and propose practical strategies to benchmark, detect, and mitigate flaky tests in complex distributed environments.

INTRODUCTION

The reliability of control systems is a fundamental requirement in large-scale scientific projects, where even minor software faults can have severe consequences for operations and data integrity. This is particularly true for the Square Kilometre Array Observatory (SKAO), an international effort to build the world's largest radio telescope [1]. The SKAO control system is responsible for orchestrating and monitoring a vast and heterogeneous set of distributed components. Ensuring the robustness of this system is essential to guaranteeing uninterrupted scientific operations and safeguarding the quality of astronomical data.

The complexity of the SKAO control system makes reliability an especially difficult goal. The telescope encompasses thousands of devices and services, each with intricate interactions and dependencies. This scale demands advanced strategies for testing, monitoring, and maintaining software reliability. At the same time, the system's architecture introduces challenges such as nondeterministic behaviour arising from the interaction of heterogeneous hardware and software components.

Within this context, test flakiness has emerged as a significant obstacle. A flaky test produces inconsistent results—passing or failing without corresponding changes to the system under test. Flakiness reduces trust in the testing process, obscures genuine defects, disrupts continuous integration and deployment (CI/CD) workflows, and ultimately leads to frustration among developers and testers. In projects such as SKAO, mitigating flaky tests is of paramount importance. Furthermore, the design and execution of the test suite itself may amplify system complexity, creating additional opportunities for flaky behaviour through its interactions with the System Under Test (SUT).

This paper presents an investigation of test flakiness in the Central Signal Processor Local Monitoring and Control (CSP.LMC) subsystem, a critical element of the SKAO control software. Our methodology combines the systematic collection of test outcomes from CI/CD pipelines, database-driven storage, and the analysis of trends through dashboards and exploratory sessions. With this approach, we aim to identify the causes and impacts of flaky tests in distributed environments and propose strategies for their detection and mitigation in large-scale scientific software systems.

SKAO CONTROL SYSTEM INFRASTRUCTURE

The SKAO control system (CS) is built around a modular, distributed software architecture designed to manage and coordinate a large number of heterogeneous devices. The CS software components are implemented in Python and based on the TANGO Controls framework [2], an open software middleware widely adopted in large-scale scientific facilities. Within this framework, each device is represented as a TANGO Device, encapsulating the logic for controlling hardware or providing higher-level coordination functionality. This model provides a uniform interface for monitoring and commanding system elements, ensuring interoperability across the diverse landscape of SKAO subsystems.

To support scalability, reproducibility, and resilience, all TANGO devices in the SKAO control system are containerized and deployed as pods within a Kubernetes cluster [3]. This design enables uniform deployment practices and facilitates automated orchestration, scaling, and recovery of software components. Kubernetes, originally developed for cloud-native applications, also offers capabilities that align

* gianluca.marotta@inaf.it

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-TUBG006

well with the needs of scientific software. On the one hand, it supports purely software-based services such as monitoring, logging, and orchestration. On the other hand, it provides mechanisms to accommodate hardware-specific requirements, for example by scheduling pods onto nodes with specialized hardware resources (e.g., FPGAs, GPUs, or high-throughput network interfaces) critical for data acquisition and processing in SKAO.

While this infrastructure provides significant capabilities, it could also introduce new challenges. The layered architecture of Python, TANGO, and Kubernetes can itself introduce nondeterministic behaviour. For instance, the scheduling of pods across nodes, asynchronous messaging between TANGO devices, and Python's runtime concurrency constraints (e.g., the Global Interpreter Lock) may lead to subtle variations in execution order between runs. Furthermore, the combination of heterogeneous hardware and distributed orchestration makes it difficult to guarantee fully reproducible runtime conditions. Although Kubernetes provides mechanisms such as resource management and scheduling policies to limit nondeterminism, residual variability can still manifest during testing as flaky behaviour. Understanding how these design choices influence test stability is therefore essential to ensuring the robustness of SKAO's software environment.

CASE STUDY: CSP.LMC

The Central Signal Processor (CSP) is one of the most critical elements of the SKA telescope, responsible for the real-time processing of the massive data streams generated by the telescope's antennas [4]. Without this processing stage, the raw data captured by the array cannot be transformed into scientifically usable products. The CSP is divided into three specialized instruments, each dedicated to a particular class of signal processing tasks. The *Correlator and BeamFormer* (CBF) combines signals from multiple antennas to electronically form beams—focused views of specific regions of the sky—and generates visibility data, the correlated measurements between antenna pairs that serve as the fundamental input for astronomical image reconstruction. The *Pulsar Search* (PSS) detects pulsars within the data stream, while the *Pulsar Timing* (PST) enables precise timing measurements of known pulsars.

Overseeing the orchestration of these instruments is the *CSP Local Monitoring and Control* (CSP.LMC) subsystem. CSP.LMC plays a dual role: internally, it provides real-time monitoring and adaptive control to ensure the smooth functioning of the CSP instruments; externally, it acts as the interface between the CSP and the Telescope Monitoring and Control (TMC) system, thereby serving as the integration point between CSP and the wider SKA control system. In practical terms, CSP.LMC acts as the "nerve centre" of the CSP, bridging the gap between system-level operations and low-level device management.

CSP.LMC is implemented as a collection of SKA software components realised as TANGO Devices. Its architecture follows an object-oriented approach, where specialized classes are responsible for different monitoring and control tasks [5]. Importantly, CSP.LMC is common to both the Low-Frequency and Mid-Frequency SKA telescopes, with only minor differences between the two instances. This makes it a general-purpose subsystem and an ideal candidate for systematic study.

This subsystem is particularly suitable as a case study on test flakiness for several reasons. First, it is a software-only component, which at the initial stage avoids the additional complications introduced by hardware heterogeneity. Second, it is hierarchically structured, with multiple TANGO Devices, each responsible for coordinating several underlying devices, making it a microcosm of the broader SKA control system. Third, prior exploratory work carried out in the previous year already highlighted the presence of flaky tests in CSP.LMC and demonstrated the potential of data mining approaches to detect patterns of instability [6].

TESTING STRATEGY FOR CSP.LMC

The reliability of a component such as CSP.LMC rests on a solid and comprehensive testing strategy [7]. In line with the official SKA testing guidelines [8], the validation process mirrors the layered architecture of SKA software, which combines multiple levels of abstraction and technologies as described in the previous sections. Accordingly, CSP.LMC is tested at three levels. At the *unit* level, tests verify the correctness of individual software modules and provide code coverage metrics (lines of code tested). *Component tests* extend this scope by validating the Python application either in isolation or within a full TANGO/Kubernetes environment, where external components are mocked in Python or emulated in Kubernetes to reproduce realistic conditions. Finally, *integration tests* verify CSP.LMC's interaction with other real SKA subsystems, ensuring that cross-component communication and system-level coordination function as intended. This multi-level strategy reflects the complexity of SKA's control software and is essential to establishing confidence in the robustness of CSP.LMC.

Part of the CSP.LMC test suite—specifically the tests executed in the Kubernetes environment, both with emulated and real subsystems—is written in Gherkin, [9] a domain-specific language designed for behavior-driven development (BDD) [10, 11]. Gherkin specifies tests in a human-readable, structured format using the *Given–When–Then* syntax to define scenarios. Each test thus follows the *Arrange–Act–Assert* pattern: *Given* steps establish context, a single *When* step triggers the action, and *Then* steps verify the outcomes. Scenarios should remain concise and readable, using the fewest steps necessary to ensure clarity, while serving as living documentation—based on the notion that well-written tests can act as alternative and complementary representations of a specification.

Each step in a Gherkin scenario is mapped to Python code that acts as a TANGO client, exercising the SUT. In CSP.LMC, steps are implemented in a parametric and

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-TUBG006

reusable manner, enabling flexible test composition while reducing redundancy across scenarios. This design also makes it possible to identify commonalities among flaky tests, potentially revealing recurring patterns and supporting the aggregation of statistics on failing steps.

CI/CD AND REGRESSION TESTING

All levels of testing are seamlessly integrated into the continuous integration and deployment (CI/CD) pipelines, which are implemented using GitLab [12]. CI/CD is a standardized and mandatory practice in SKA software development [13], supported by dedicated cluster resources. Each code change triggers automated pipelines that build the application and execute the complete testing stack—from unit to integration tests—ensuring that regressions are detected as early as possible in the development cycle. Beyond validating new contributions, these pipelines also perform regression testing, systematically re-running existing tests to confirm that previously verified functionality remains stable across updates.

However, despite the robustness of this CI/CD and regression testing framework, test flakiness occurs as a recurring challenge. "Flaky" failures are particularly disruptive in regression testing, where the expectation is that previously validated functionality should behave consistently across repeated executions. Instead, flaky tests introduce uncertainty into the results: a failing regression test may indicate either a genuine defect or a transient, non-deterministic issue. This ambiguity not only complicates fault diagnosis but also reduces developer confidence in the testing process, leading to wasted time, unnecessary re-runs, and a slower integration cycle.

DATA COLLECTION

Investigating test flakiness in CSP.LMC required the design of a dedicated data collection infrastructure. Since flakiness is primarily observed in tests executed within the Kubernetes environment, our investigation focuses on this context.

As a first step, we exclude development versions of the software. The rationale is that our goal is not to do regression testing – already in CI/CD – , but rather to detect failures in situations where they should not occur—namely, when both the code and the tests have already passed all verification stages and the official SKA release procedures.

To support this, we developed modified CI/CD pipelines, maintained on a dedicated GitLab branch (Fig. 1). These pipelines are simplified, consisting of only two stages, and are scheduled to execute every six hours. Because they are branch-specific, the version under test can be explicitly controlled and is decoupled from the usual SKA release process. In particular, no build stage is included; instead, the pipeline directly executes a test stage that deploys the SUT into a Kubernetes cluster. The pods in this deployment use container images retrieved directly from the official SKA artifact repository, ensuring that the tests run against validated software versions.

For this study, the test stage was configured to run not only on the standard SKA facility hosted by the UK Science and Technology Facilities Council (STFC) [14], but also on an additional cluster known as Federation hosted by the Italian National Institute for Astrophysics (INAF). Test jobs are executed in parallel across these two environments, allowing us to capture potential site-specific variations in flaky behaviour. Unlike standard CI/CD tests, these jobs are allowed to fail.

Following the execution of the tests, a second stage named *collect-test* is triggered. This stage runs two parallel jobs that process the artifacts produced during the previous test stage. The jobs extract execution data and generate HTML reports, which are both published on GitLab and stored for archival at the Federation facility. In addition, the results are ingested into a MariaDB database [15] hosted at Federation, providing a structured repository for subsequent analysis.

DATABASE STRUCTURE

In this section, we provide a description of the database used to collect test data. It is implemented as a multi-table relational database in MariaDB.

At the top level, the **pipeline_jobs** table records metadata about each CI/CD job execution, including the *facility* in which the job ran, the *execution timestamp*, and the *subsystem type*. The subsystem type distinguishes between tests executed with real subsystems and those that could, in principle, be run with emulated subsystems, although at present only real deployments are used. Each job is uniquely identified by a *job_id*, which corresponds to the identifier provided by GitLab. This identifier also serves as a shared primary key linking downstream tables.

Associated with every job are the product versions used in that execution, stored in the **product_versions** table. This table ensures that test outcomes can be correlated with the exact software versions deployed, a crucial feature for reproducibility and failure analysis. Versions of all CSP subsystems are included, namely CSP.LMC, CBF, PSS, and PST. It also includes *job_id*.

The **test_executions** table stores the results of each test executed within a job. For every test, information such as the name, start time, total duration, setup and teardown outcomes (and their respective durations), and final result (pass/fail) is recorded. Other than the *job_id*, each test is assigned a unique *test_id*, which acts as a primary key for linking to the subsequent tables.

Each test execution is further decomposed into test steps, captured in the **test_steps** table and linked to the corresponding test execution via the *test_id*. Steps follow the Gherkin-based syntax and include the specific *keyword* (Given–When–Then), *step description*, *outcome*, and *execution duration*. Analyzing results at this level makes it possible to detect whether flakiness is systematically associ-

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-TUBG006

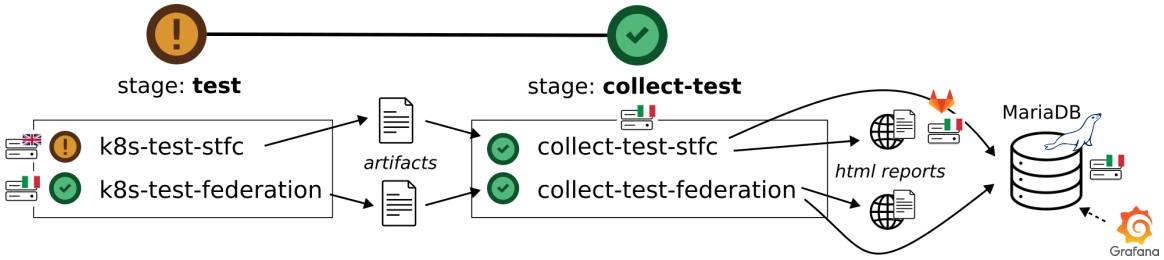

Figure 1: Schematic of the modified GitLab pipeline for the collection of test results. The symbols with the UK or Italian flags indicate execution on the STFC cluster and the Federation cluster, respectively.

ated with particular steps across multiple tests, thus helping to identify recurring instability patterns.

Finally, detailed diagnostic information is stored in the **test_logs** table that reports log entries from the test client. Each log entry is linked to a specific test execution through the *test_id* and includes *timestamped* records of the *log level*, *code source*, *thread*, and *message*.

Overall, this multi-table schema creates a hierarchical representation of testing activity: from pipelines, to jobs, tests, steps, and finally to logs. The relational links between tables (e.g., *job_id* linking jobs to tests and versions, *test_id* linking tests to steps and logs) enable failures to be traced end-to-end and support data mining at multiple levels of granularity.

ANALYSIS AND MITIGATION

Understanding the origins of flaky behaviour requires not only systematic data collection but also tools and methodologies for exploring and interpreting data.

The Global Fail Rate

To quantify test reliability, we introduce the *Global Fail Rate* (GFR), defined as the ratio of failed tests to the total number of executed tests for a specific release of CSP.LMC. Unlike raw failure counts, this normalized metric enables comparisons across different environments and between releases, providing developers and stakeholders with a measure of the effectiveness of mitigation strategies adopted for test flakiness.

GFR reflects, for each version of the SUT, the combined quality of both the control software and the testware. A high GFR may indicate low reliability of the software component itself, but it may also point to weaknesses in the test harness, where avoidable failures arise from the way the test suite exercises the production code.

Tools for Analysis (Grafana and HTML Reports)

The primary tool for monitoring and analysis is a set of Grafana [16] dashboards built on top of the MariaDB database. These dashboards allow both high-level and fine-grained views of the test results. At a global level, they provide an overview of test execution outcomes over time, highlighting long-term trends in stability and surfacing regressions or anomalies. At a more detailed level, they support drill-down analysis of specific jobs, tests, and even individual steps, making it possible to correlate failures with subsystem versions, facilities, or runtime conditions.

Figure 2 shows a Grafana dashboard used to analyze CSP.LMC Mid test results. It provides both global and detailed views of test reliability, including the GFR per software version, comparisons across test facilities (INAF-Federation and STFC), and fail rate trends over successive sessions. The dashboard also highlights the most frequently failing scenarios and steps, allowing developers to pinpoint sources of instability. Finally, a per-session breakdown links directly to HTML reports for further inspection.

In addition to Grafana, automatically generated HTML reports from each pipeline run provide a human-readable snapshot of execution outcomes, which can be used for quick validation or shared among developers for debugging. They provide also a link to the TANGO device logs. HTML reports and device logs are stored in the Federation cluster alongside the database and can be accessed directly through the Grafana dashboards. An example is reported in Figure 3.

Collaborative Database Exploration

Exploring such a complex range of behaviours is difficult to automate through software tools in a way that consistently produces results useful for developers seeking to define mitigation strategies for test flakiness. Moreover, this complexity cannot be delegated to a single tester or individual, as responsibility for interpreting results and proposing solutions must be shared across the team.

In our opinion, the best approach is the organization of periodic collaborative database exploration sessions to encourage collective analysis of flaky behaviour. These sessions bring together developers, testers, and system engineers to jointly interrogate the database using Grafana dashboards and HTML reports. This collective approach leverages domain expertise from multiple perspectives, enabling richer interpretations of flaky outcomes and faster identification of potential root causes. At the same time, these sessions provide an opportunity to refine the analysis pipeline itself, ensuring that the database schema, reporting, and dashboards continue to evolve in ways that best support the SKAO development community.

As a tangible outcome, each session should generate a series of action items, which can then be analyzed and prioritized within the CSP.LMC working plan.

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-TUBG006

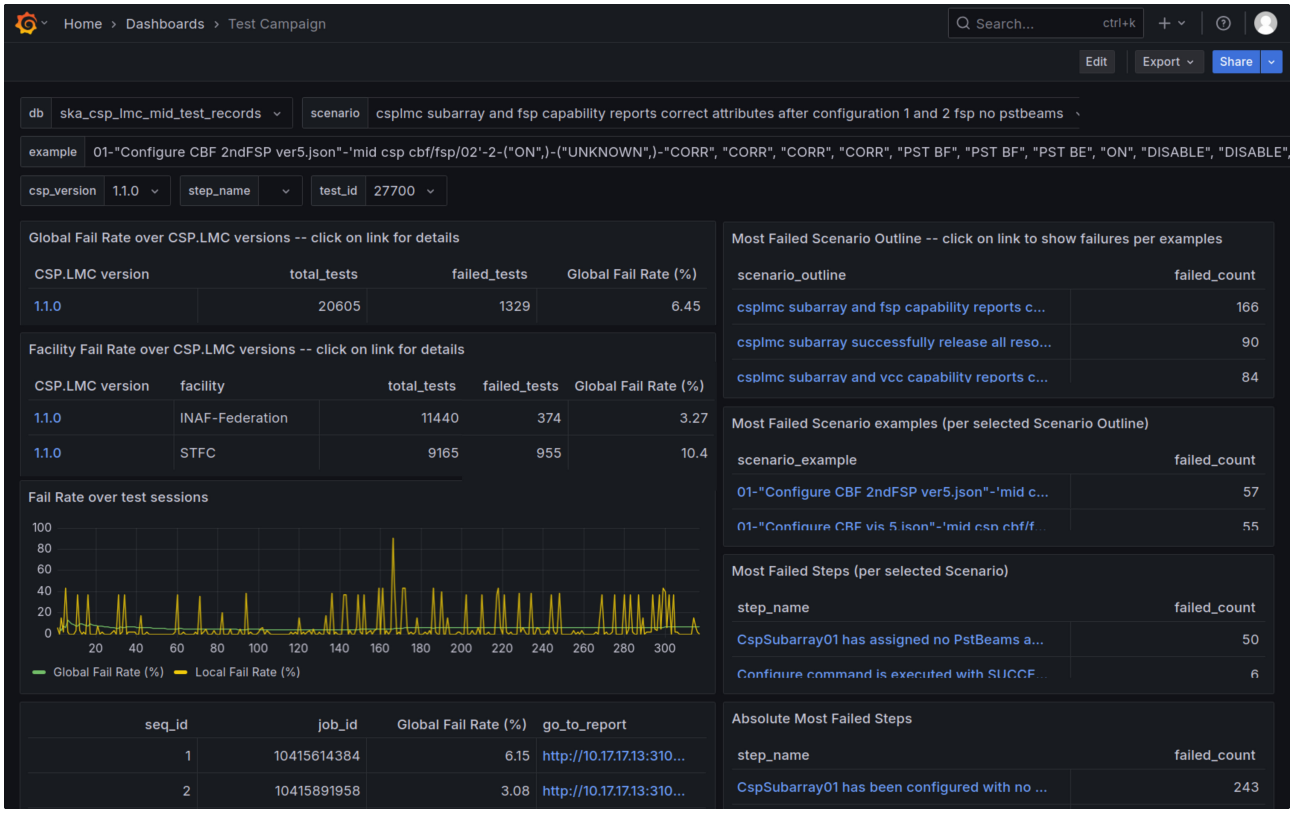

Figure 2: An example of a Grafana dashboard for exploration of Mid CSP.LMC results.

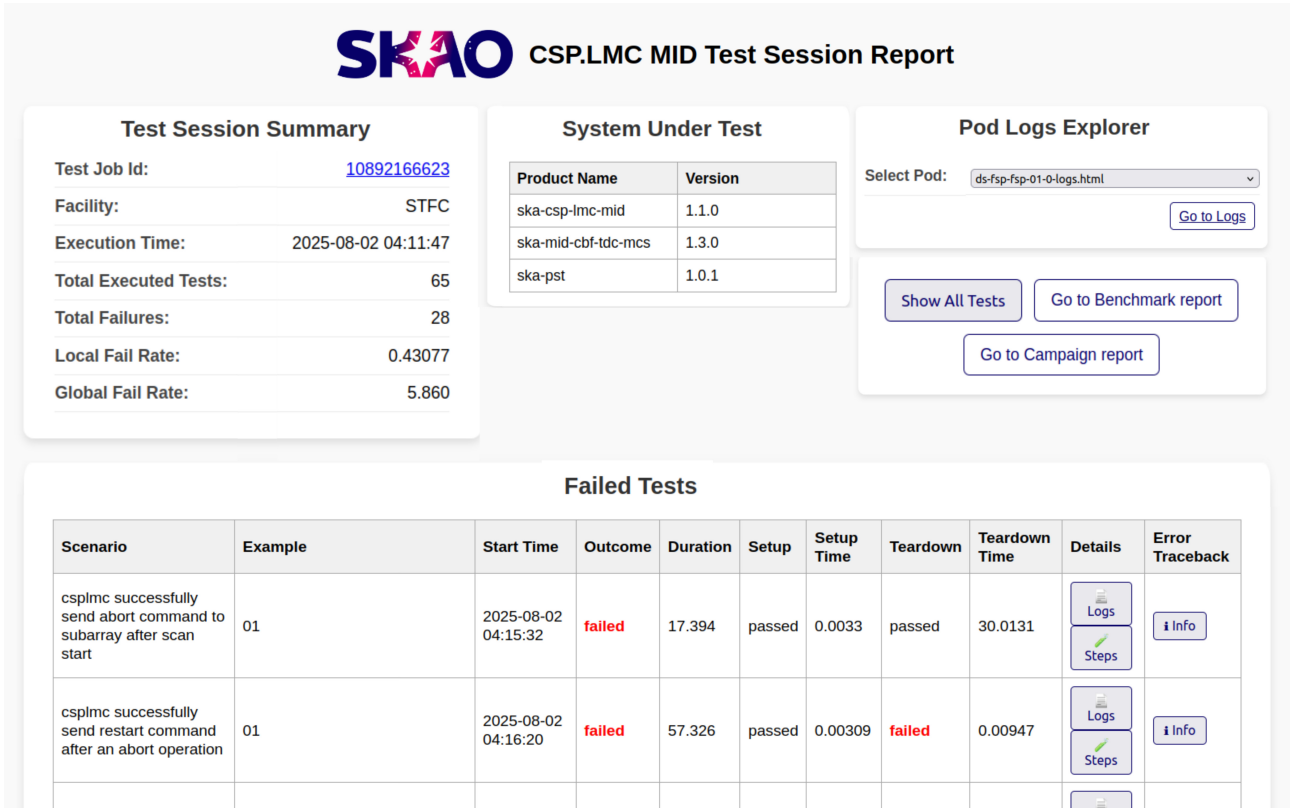

Figure 3: An example of an HTML report for a Test Session for CSP.LMC Mid.

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-TUBG006

FIRST RESULTS

We can compare the Global Fail Rate (GFR) with the results of the previous data collection campaign [6], although that study employed a different methodology for CSP.LMC Low. GFR decreased markedly, from 23.3% in version 0.13.0, to 6.7% in version 0.14.0, and finally to 0.82% in the current release (1.0.1). These changes reflect the continuous commitment of CSP.LMC developers to improve both code quality and testware quality. Qualitatively, we can infer that the first drop (from 0.13.0 to 0.14.0) was mainly due to dependency updates, while the second (from 0.14.0 to 1.0.1) resulted from a major refactoring of the CSP.LMC test suite. Unfortunately, structured data collection was not available for the earlier versions, so only the latest release can be analyzed in detail. However, the new data collection infrastructure described above will allow future changes in GFR to be tracked and analyzed more systematically.

For CSP.LMC Mid, GFR has been collected for the first time, with an observed value of 6.0%. An interesting outcome is the difference in GFR between the two facilities, which is unexpected given the facility independence of software-only tests. This discrepancy is more pronounced for CSP.LMC Mid (10.6% at STFC vs. 3.3% at Federation) than for CSP.LMC Low (0.9% at STFC vs. 0.71% at Federation). This observation deserves further investigation and may be related to the prevalence of sessions with high failure rates at STFC. These high failing sessions can be spotted in the "Fail rate over test session" graph of the dashboard presented in Figure 2.

In addition, the first database exploration session was conducted with the objective of analyzing the Most Failing Test (MFT) for CSP.LMC Low. During a one-hour session with two CSP.LMC developers, it became evident that the failures of this test were primarily due to initialization issues. Specifically, these problems resulted either in failed or delayed device deployments, or in communication errors within the TANGO framework. As an outcome, an action item was defined to reproduce the anomaly and improve the robustness of the test setup during initialization. This example illustrates how collaborative exploration sessions can effectively identify root causes and produce actionable strategies for mitigating test flakiness.

Furthermore, collecting detailed test information—particularly at the step level—has proven effective for establishing an initial benchmark of CSP.LMC operations. By aggregating identical When calls across multiple executions, it becomes possible to derive statistics on the execution times of successful CSP.LMC commands. These benchmarks provide valuable insight into the typical performance envelope of the system: for instance, they allow developers to distinguish between expected variability and anomalous slowdowns. Over time, such measurements can serve as reference values to detect regressions, verify the impact of code refactoring, or identify potential performance bottlenecks in the interaction between TANGO devices and Kubernetes orchestration.

CONCLUSIONS AND FUTURE WORK

With this work we aim to continue a well defined strategy to deal with the phenomenon of test flakiness in CSP.LMC, a critical subsystem of the SKA control software, in order to be an example on how to deal with this phenomenon in SKA development as well as in other big projects. By designing a dedicated data collection infrastructure and deploying simplified CI/CD pipelines, we systematically gathered and analyzed test outcomes across different environments and releases. The measure of the Global Fail Rate (GFR) provided a normalized metric to quantify reliability, capturing both software and testware quality. The integration of Grafana dashboards and HTML reports enabled multi-level exploration of flaky behaviour, while collaborative database exploration sessions demonstrated how collective analysis can translate observations into concrete mitigation actions.

Initial results show a marked reduction in GFR for CSP.LMC Low across recent releases and a first benchmark for CSP.LMC Mid, with facility-dependent differences on the Fail Rate that require further study. Database exploration also identified specific issues, such as initialization-related failures, demonstrating how collaborative analysis can produce actionable mitigation strategies. Step-level statistics have additionally provided useful benchmarks for CSP.LMC command execution times.

Looking ahead, several directions will guide future work. First, we plan to incorporate data analysis frameworks such as Pyhthon Pandas and Jupiter Notebooks, enabling more advanced exploration of flaky patterns and statistical correlations. Second, the data collection will be extended to include resource consumption metrics (CPU and memory usage of the pods hosting the devices) to complement functional outcomes with performance benchmarks. This integration will allow us to investigate the relationship between flaky behaviour and resource contention in Kubernetes environments. Finally, our long-term objective is to expand this methodology beyond CSP.LMC to other SKA subsystems, and to develop automated support for hunting hidden bugs by combining failure metrics, performance benchmarks, and collaborative exploration.

ACKNOWLEDGEMENTS

This work made use of the "INAF Federation" computing cluster, acquired with funding from the Italian "Piano Nazionale di Ripresa e Resilienza" (PNRR), supported by the European Union – Next Generation EU program, and operated by INAF.

REFERENCES

[1] SKA Observatory, http://www.skao.int

[2] Tango Controls, https://www.tango-controls.org/.

[3] Kubernetes Docs, https://kubernetes.io/docs/home/.

[4] C. Baffa *et al.*, "SKA CSP controls: technological challenges", in *Proc. SPIE 9913*, vol. 9913, p. 99132Z, Jun. 2016. doi:10.1117/12.2233325

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-TUBG006

[5] G. Marotta *et al.*, "Software design for CSP.LMC in SKA", in *Proc. SPIE*, vol. 12189, p. 121891A, Jun. 2022. doi:10.1117/12.2630140

[6] G.Marotta *et al.*, "Enhancing SKA software testing through data mining strategies", in *Proc. SPIE* vol. 13101, p. 131010E, Jun. 2024. doi:10.1117/12.3021029

[7] G. Marotta *et al.*, "Strategy and tools to test software in the SKA project: the CSP.LMC case", in *Proc. ICALEPCS'23*, Cape Town, South Africa, Oct. 2023, p. 34-39. doi:10.18429/JACoW-ICALEPCS2023-MO2BCO03

[8] SKAO Software Testing Policy and Strategy, https://developer.skao.int/en/latest/policies/ska-testing-policy-and-strategy.html

[9] Gherkin language, https://cucumber.io/docs/gherkin/reference

[10] Introducing BDD, https://dannorth.net/introducing-bdd/.

[11] BDD Description, https://cucumber.io/docs/bdd/.

[12] About Gitlab, https://about.gitlab.com/.

[13] M. Di Carlo *et al.*, "CI-CD practices at SKA", in *Proc. SPIE* vol. 12189 p. 1218903, Jun. 2022. doi:10.1117/12.3021029

[14] Science and Technology Facilities Council, https://www.ukri.org/councils/stfc/.

[15] MariaDB docs, https://mariadb.org/documentation/.

[16] Grafana docs, https://grafana.com/docs/grafana-cloud/