Project funded by the EU Horizon 2020 programme under the Marie Skldowska-Curie grant agreement No 871342

# uDevoOps

# Software Quality Assurance for Microservice Development Operations Engineering

# Deliverable D5.1. Testing process integrated in $\mu$DevOps fully defined


May 2025

**Abstract**

This document reports the results of Deliverable D5.1 of the $\mu$DevOps project, entitled "Testing process integrated in $\mu$DevOps fully defined". The type of the deliverable is marked as *Report*, and its dissemination level is *Public*. The document will be made available through the project's website, `https://udevops.eu/`.

The document describes the overall testing process that puts results of WP2, WP3 and WP4 together in a workflow to implement a context-aware (WP2), in vivo (WP3) and risk-based (WP4) testing process. The solution implements a conceptual workflow starting from the several sources of information in a uDevOps engineering process and encompasses two main areas: quality assessment and quality improvements through risk mitigation, with several techniques that we implemented in throughout the process targeting quality requrements that range from reliability to performance, energy consumption, and, to a less extent, security.

The report describes broadly about all the strategies that we have implemented for all the quality attributes we are targeting, while pointing to previous deliverables for more details or to the project's papers derived from new techniques for further details.


# CONTENTS


# 1 INTRODUCTION

The rise of microservice-based systems has transformed how modern software is developed, deployed, and maintained. Within this landscape, the integration of testing activities into the continuous development and operations workflow (DevOps) becomes not just beneficial but essential to maintain high levels of quality, reliability, and performance. This deliverable, D5.1, presents a comprehensive and fully defined testing process tailored for microservice architectures in the context of $\mu$DevOps.

This document consolidates and builds upon outcomes from previous work packages (WP2, WP3, and WP4), each contributing specific techniques and tools for context-aware, in vivo, and risk-based testing. The result is a unified framework that seamlessly integrates diverse sources of data – from code repositories and system logs to runtime performance metrics - into a coherent process for both quality assessment and quality improvement.

The proposed testing workflow addresses a wide array of quality attributes, including but not limited to reliability, performance, energy


consumption, and robustness. It employs advanced methodologies such as machine learning for fault prediction, causal inference for performance issue diagnosis, and large language models (LLMs) for log-based test case generation. These approaches enable intelligent automation across the testing lifecycle, from operational profile extraction to test case prioritization and failure root cause analysis.

Moreover, the $\mu$DevOps testing process is not merely about detecting faults; it aims to systematically reduce risk by guiding development and operational decisions through actionable insights. The services developed span both the development (Dev) and operational (Ops) phases and are structured around two orthogonal goals: quality assessment and quality improvement. This includes services for fault avoidance, fault prediction, fault removal, and fault tolerance, supporting a wide range of testing strategies – from traditional functional testing to cutting-edge ex-vivo and AI-driven techniques.

The rest of this deliverable details the high-level design of the testing process, the specific services and techniques developed, and their integration into the $\mu$DevOps pipeline. Particular emphasis is placed on innovations introduced in WP5, including new methodologies and tools that enhance the overall framework's effectiveness and applicability in real-world microservice systems.


# 2 HIGH-LEVEL PROCESS DESCRIPTION

Figure 2.1 illustrates the uDevOps comprehensive microservices quality assessment and improvement framework protptyped in the uDevOps project.

In the following, its structure is descried.

- **Data Sources and Metrics**. The uDevOps is meant to be *context-aware*. In order to support quality assessment and improvement decisions to minimize the risk of failure *depending on the context*, we need to harness data about the system under test In a DevOps process, data come from several sources, both at development (*Dev* data) and operational time (*Ops* data); we list the ones we harnessed in our project, starting for *Ops* data:
  - Data coming from logs capturing relevant events such as failures, errors, warnings, or internal exception,
  - Data from traces whenever we instrument the software to

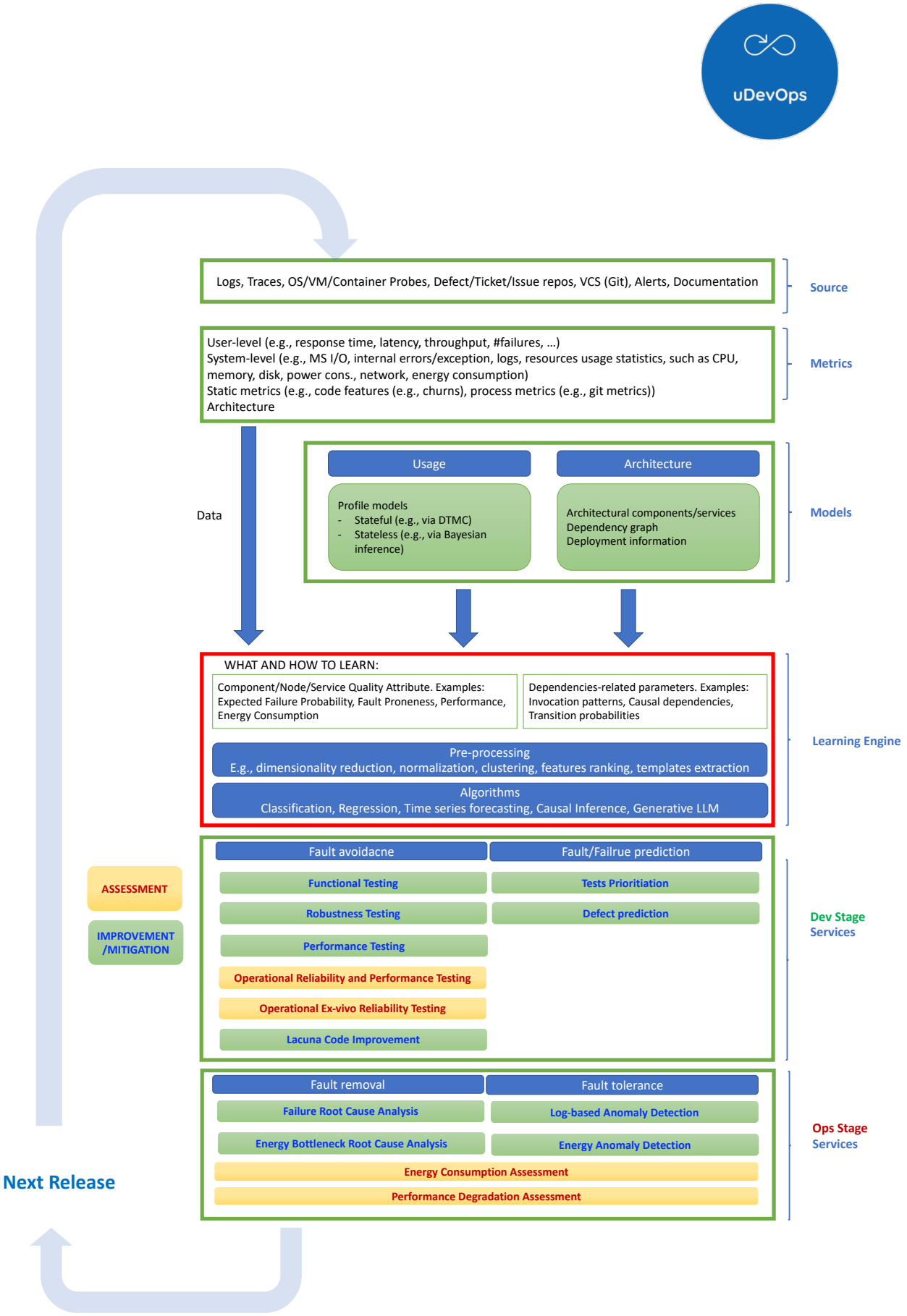



**Figure 2.1.** Usage workflow of the $\mu$DevOps learning engine.


characterize, for instance, the typical invocation sequences and build a reliable service depenedency graph;

- Data coming from resource usage statistics collected by the OS or by the VM/Container engine like Docker stats; these include both user and system metrics, such as: response time, latency, throughput, failure counts, at user-level; CPU consumption, memory, disk, network, power usage, at system level.
- Data coming from the development (*Dev* data) are especially useful to have the architecture of the system, deployment information (e.g., which service to which container and node) and static dependencies (e.g., call graph between services) – design and deployment documentation is the source of data here.
- Code churn metrics and version control systems data (e.g., from Git), such as number of commits, lines added, modified, removed, test case metrics, and generally all those data exploited in Just-in-Time defect prediction and for test case prioritization.
- Data entered by users about experienced issues or alerts in issue tracking systems.


- **Models**. While some data directly go through the Learning Engine block to be processed as input to the learning algorithms, some other data are used to first parameterize other types of models (non-ML). Examples include usage profile model capturing how often a user exercises a given service or a sequence of service – a piece of information useful for estimating quality figures aware of the actual contexts in which the system is used - or architectural and dependency model starting from documentation.

- **Learning Algorithms**. The learning algorithms process data collected from monitoring, whose selection depends on the decision to be supported (i.e., the SQA objective). We adopt a standard machine learning workflow to aid SQA decisions by utilizing open-source libraries that provide machine learning and causal inference algorithms.

  The algorithms make appropriate predictions aligned with tho SQA objective to support. The used families of algorithms include: Classification, Regression, Reinforcement Learning, Causal Inference and Large Language Models, for the techniques described in the rest of the deliverable or in the previous project's deliverables.

- **Services**. At the bottom, we have services supporting both *Dev* stage decisions and *Ops* stage ones. These are distinguished across two dimensions: **Dependability & Performance goal**; **Assessment or**


**Improvement goal**. As for the former, we adopt a little bit extended notion of *failure* and *fault* compared to the usual dependability taxonomy. We consider as a *failure* any deviation from the expected quality requirement, being it a dependability, security, performance or even energy requirement; and as a *fault* the adjudged cause for such a deviation. Based on this, we distinguish the services w.r.t. the goal supported:

- **Fault Avoidance**. This refers to all those good design and development practices that avoid a failure to occur in operation. Such activities include good design principles like modularity, information hiding and encapsulation (which we do not support), code analysis and improvement (we have a service for this), and testing, wherein we have most of our services. Our solutions support functional, reliability, robustness, and performance testing for both fault detection and quality assessment, including energy. Testing is followed by debugging for code improvement, with which we do not deal. Several of the implemented techniques here are *ex-vivo*, namely the exploit in different ways data gathered from the field, typically for a faithful assessment of a quality attribute of interest.

- **Fault Prediction**. This encompasses activities aimed to predict


defective modules, such as files or components, or defective commits, like done in today's just-in-time (JIT) defect

- **Fault Removal**. Fault removal can refer both to development-time failures that call for correction, such as debugging the code to remove faults after testing or other V&V activities exposed a failure, or can result from operational-time failures. We refer to this latter case, and focus on a fault removal activity that is common in microservice (more generally distributed) systems, that is *root cause analysis* (RCA). In microservice architectures, wherein the system is composed of tens or hundreds of small loosely coupled services, before going to debug the code engineres typically try to efficiently identify the service within which the failure originated. In often case, there can be long propagation chains within a microservice system before the problem reaches the user interface. RCA here refers to the activity that, starting from the failed service, attempts to automatically track back the failure until the original failing service is identified. It usually exploits data from monitoring and logging to infer such chains. We implemented several algorithms for this activity.
- **Fault Tolerance**. This aims to ensure the system works correctly even in presence of failures. It encompasses two


steps: error detection and recovery, in turn consisting of error handling (e.g., via rollback, roll-forward, compensation/masking) and fault handling (with four phases, diagnosis, which can exploit RCA, isolation, reconfiguration, re-initialization). In uDevOps, we focused more on the error detection stage and subsequent RCA, with anomaly detection from logs data or referred to energy hotspot.

The second classification dimension distinguishes between assessment and improvement.

- **Assessment.** Techniques for assessment aims at estimating the expected value of a quality attribute of interest. For instance, in the case of testing, the goal is to test the system in the same conditions expected in operation, thus with a testing profile similar to the operational profile. In such a case, the *ex-vivo* approach can be particularly useful, as data gathered from monitoring in previous iterations can well serve to estimate the expected profile.
- **Improvement.** In this case, the technique aims to expose as many problems as possible. Typical testing for fault detection, either functional, robustness or performance, falls in this category, as the generation of test cases aims to either cover the requirements (regardless of the actual impact of


the requirement on the end user) or to trigger failures, e.g., in unexpected conditions. Techniques for code improvement (e.g., we implemented dead code elimination), for defect prediction or tests prioritization also fall in this category.

The subsequent Sections detail the services designed and experimented within the project as part of this process, following the two above dimensions (*fault avoidance*, *prediction*, *removal*, *tolerance*; *assessment vs improvement*). In Deliverable D5.2, we will release a proof-of-concept containing prototypes of a subset of such services that we implemented to demonstrate the process.

Before starting, let us remark that, in the following, the concept of **failure** is broadly meant as any deviation of a delivered service from the intended service a system is designed for, including both functional and non-functional requirements such as correctness, reliability, robustness, performance, security and energy efficiency – the ones we put the focus on.

A further note is that **we will hereafter present briefly the services based on techniques already introduced in other deliverables, while we will devote some more space to new techniques developed in WP5.**


# 3 FAULT AVOIDANCE SERVICES

## 3.1 OVERVIEW

This Section reports the techniques we implemented that support fault avoidance, highlighted in the previous Section. We distinguish assessment and improvement techniques, the former useful for risk assessment the latter for risk mitigation as described in Deliverable D4.1.

## 3.2 SERVICES FOR QUALITY ASSESSMENT

### 3.2.1 Operational Reliability and Performance Testing

We developed two ex-vivo techniques that exploits data gathered from the field to derive tests whose results enable an unbiased estimate of reliability and/or of performance. The two techniques mainly differ in the granularity. The former generates tests considering the observed operational profile in terms of equivalence classes of inputs for each invoked end-point, thus considering the contribution to the failure probability of different

combinations of input classes. The latter has a rougher grain, as the profile is defined in terms of frequencies of invocation, not looking at the specific input values, using a Discrete-time Markov chian to describe the profile.

Here we briefly describe the two techniques, prototyped in WP3.

**Adapative Operational Reliability Testing**

The service offers a *stateless* testing technique, where microservices are tested individually by generating invocations to the endpoints API that harness historical data. Specifically, the technique proposed uses *adaptive sampling* for reliability-assessment testing.

It acts as a run-time testing strategy, triggered upon request by a stakeholder who needs an estimate of the microservices operational reliability. It achieves unbiasedness, accuracy and efficiency by three key activities:

1. **Monitoring**: Field data are gathered about the microservices' usage profile and about failure/success of demands. This provides updated estimates representing the real reliability at the time when the assessment is requested.

2. **Testing**: Using only passive observations (monitoring) is inadequate for estimates with high accuracy and confidence. The testing algorithm harnesses such data and adaptive statistical sampling, to


drive test generation and accelerate the exposure of failures. The input space is partitioned into subdomains using specification-based partitioning. It focuses on defining equivalence classes based on input arguments of a method's signature, for instance based on string length and content. Data gathered reports the usage and failure frequency of each partition (called *test frame*). This is the usage profile.

3. **Estimation**: The testing algorithm identifies the most relevant test cases in few steps, by forcing a disproportional selection of test cases with respect to the observed usage profile. In principle, such a type of sampling would yield biased estimates. Therefore, a proper weight-based estimator is adopted at the end of testing in order to counter-balance the selection strategy, ultimately providing an accurate and unbiased estimate with small variance.

The technique is described in detail in Deliverable D3.1.

**State-based Operational Reliability and Performance Testing**

This implements a stateful technique for performance and reliability testing. It integrates *performance and reliability assessment* in a DevOps context, with a focus on continuous testing and monitoring. It involves realistic testing in production or staging environments and comprises three main steps:


### 3.2. SERVICES FOR QUALITY ASSESSMENT

- Definition of Operating Conditions from field data, including: *i)* Workload specification (valid/invalid requests based on API); *ii)* Behavioural models based on Discrete-Time Markov Chains, DTMCs to capture the sequences and probabilities of invocations; *iii)* Workload intensity (number of concurrent users) and Behaviour mix (namely, distribution of user types) definition. Session logs provide raw data for automatic extraction of these models. The DTMC is used to simulate user sessions with transitions between request types, according to the specification.

- Ex-Vivo Testing. Tests are executed in controlled deployments replicating real workloads, thanks to DTMC-based synthetic user generation. Requests and inputs are sampled probabilistically from the DTMC, in order t oget tests that cover multiple configurations allowing to assess system response under varying conditions.

- Integrated Performance & Reliability Analysis. The gathered data are used to assess performance (measuring how close the average response time is to a threshold, defined based on usability or scalability standards) and reliability as Ratio of successful (2xx) HTTP responses. Results are automatically visualized using Apache Zeppelin notebooks.

The technique allows to fully automate testing from usage data to analysis and enables early detection of performance/reliability issues.

Details are reported in Deliverable 3.1.

### 3.2.2 Log-based Reliability Testing

This service is particularly useful when operational data about specific inputs are expensive to collect (e.g., require instrumentation) and we want to minimize the manual intervention that is foreseen to define the workloads for reliability and performance testing of the previous technique – which, however, remains useful for workload-driven performance testing not provided by the techniques hereafter explained. This technique, in fact, exploits logs data, and is useful for operational reliability assessment, as well as for fault detection and coverage driven by the observed behaviour in operation.

The technique is descried in detail, being it developed in WP5.

Overview: The technique developed is a black-box testing technique that enables *stateful operational* testing exploiting data from the field in the form of sequences of invocation from collected *logs*, unlike the previous two strategies. Black-box testing is a powerful way to assess and/or improve MSA correctness and performance. Most existing techniques generate HTTP requests from the given API microservice specification, with strategies including search-based testing Arcuri (2021), heuristics and graphical structures for inferring data dependency (Corradini et al.) or producer-consumer dependency Atlidakis et al. (2019), as well as injecting




faults in the request Heorhiadi et al. (2016). Their performance is judged in terms of coverage metrics (often response code coverage, schema coverage) Martin-Lopez et al. (2019a), and of fault detection, meant as mismatch between specified and returned HTTP code, or schemas violation Golmohammadi et al. (2023).

Although these techniques provide important evidence for reasonable trust in the system and for marking it as ready for release, they do not account for how the system is actually used in the field. The detected faults and achieved coverage refer to the adopted *testing* profile rather than to the *operational* profile, with potential mismatch between the believed and actual quality. It may easily happen that a service found to be heavily faulty at testing time will rarely cause the system failure in operation if that service is rarely used; conversely a quite robust but highly-exercised service might impact runtime reliability more. Similarly, coverage achieved during testing may not be representative of what will be actually exercised at runtime; ideally, one should cover more thoroughly those services likely to be exercised at runtime – a notion formalized in a different context as *operational coverage*.

The testing literature often neglects operational-time information, as it has always been difficult to gather, and a design-time estimation of the operational profile is unreliable. However, with microservices, this is no longer true. Thanks to the frequent releases and built-in monitoring facilities, engineers actually know how the system is being used, and what


service they should focus on.

We proposed Log-driven Microservice testing with Transformers (*LoMiT*), an *ex vivo* testing technique that harnesses operational data coming from monitoring to generate stateful test cases representative of the expected operational behaviour – namely, enhancing *operational coverage* - and able of detecting faults in services more likely to be exercised in the next release's operational stage.

The solution utilizes operational log data to understand system behaviors and enhance the test suites with representative test cases. The strategy learns the relationship between service invocation sequences and the consequent system behaviors as captured by logs (e.g., occurrence of 500 HTTP status codes, high response times) by a fine-tuned Large Language Model (LLM). During inference, the model is asked to generate new invocation sequences based on logs of interest given as input, such as logs with erroneous or high-latency behaviors. This allows to both mimic the operational behaviour and triggering more sequence-dependent failures.

Figure 3.1 shows the main steps characterizing the technique. The proposal leverages both *event logs* and *system trace data*.

**Event logs** consist of sequences of text lines generated automatically during the software execution, typically saved in log files, describing the system's runtime behavior Cinque et al. (2020), including notifications related to system failures.

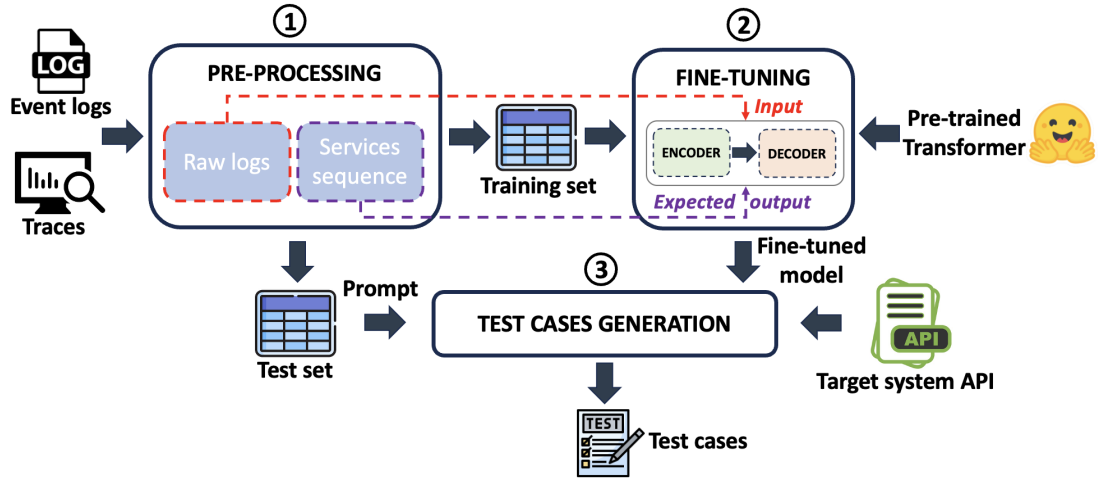



**Figure 3.1.** Transformer-based *ex vivo* testing technique for regression test cases generation using event logs and system traces.

**System trace data** provide detailed data on the interaction of the system's services, allowing to track source-destination of the services invocation, the obtained response codes, invocation latency, etc.

A transformer (originally pre-trained on English) is fine-tuned using both data sources with the aim to learn the association between service invocation chains $\mathcal{X}$, embedded via $n$ features $\{x_1, x_2, \ldots, x_n\} \in \mathcal{X}$, and the produced logs $\mathcal{Y}$, described by $m$ features $\{y_1, y_2, \ldots, y_m\} \in \mathcal{Y}$. This way, the LLM learns the joint probability distribution $P(\mathcal{X}, \mathcal{Y})$.

The fine-tuned LLM is then prompted to generate new sequences of service invocations $X'$ given a subset of $k$ features of interest of data sources $Y$: $Y' = \{y'_1, y'_2, \ldots, y'_k\} \in \mathcal{Y}$, representing a behavior that the tester aims to reproduce, e.g., logs with services failures or with high latency. In other words, we ask for $P(\mathcal{X}'|\mathcal{Y}')$.

```
"trace_id":"67d42fcc48af866c",
"involved_services":"ts-station-service--ts-food-service--ts-route-
service--ts-food-map-service--ts-travel-service",
"involved_endpoints":"http://ts-station-service:12345/api/v1/
stationservice/stations/id/Shang%20Hai--http://ts-food-service:18856/api/
v1/foodservice/foods/2021-04-30/Shang%20Hai/Su%20Zhou/D1345--
...{TRUNCATED}",
"latency_sec":0.022428,
"raw_logs":"2023-11-20 08:12:47.872 INFO 1 --- [io-12340-exec-3]
auth.service.impl.UserServiceImpl: Register User Info is:  a3b3c833-3fcc-
4aff-a0d3-3c638e02676a\n ...{TRUNCATED}",
"201":0,
"200":13,
"500":0,
"WARNING":0,
"INFO":28,
"ERROR":0
```


**Figure 3.2.** Example of entry in the dataset obtained for the TrainTicket subject.

The LLM fine-tuning requires a pre-processing step (Subsection 3.2.2) to associate each trace — representing a specific system invocation - with the corresponding entries in the event log. The LLM fine-tuning is done with a subset (Training set in Figure 3.1) of the dataset produced by the pre-processing stage.

All the three steps of the technique are detailed in the following.

### Pre-processing

Given a collection of logs and traces, the pre-processing step links each trace $\mathcal{T}_i$ (identified by a trace ID $i$) to the event log entries generated during the invocations included in the trace. Specifically, this stage: (*i*) identifies the time span of each trace (time interval between the first and the last recorded events) and extracts the *sequence of services* invoked within the

trace; (*ii*) aggregates the log entries that occur within the extracted time span (hereinafter *raw log*); (*iii*) labels the raw log entries with the extracted services sequence.

These steps generate a dataset including, for each trace ID, (*i*) the sequence of services involved in the trace – denoted as $\mathcal{X}$; (*ii*) the raw log entries associated with the trace, along with additional data extracted from the trace and logs, i.e., latency (in seconds) of the interaction described by the trace, the endpoints involved in the interaction, the number of log entries per severity level (e.g., WARNING, ERROR) and the distribution of `http` response codes (e.g., 200, 400, 500) – all these features being denoted as $\mathcal{Y}$. Figure 3.2 shows an entry from the dataset obtained for TT.[1] Finally, the dataset is evenly divided into Training and Test sets, the first one used for model fine-tuning, the second for test cases generation.

**Fine-tuning**

We use a T5-small model, a reduced version of the T5 transformer-based sequence-to-sequence model. T5-small is based on the encoder-decoder architecture, and encompasses six layers in each encoder and decoder, eight attention heads, and around 60 million parameters. T5 is pre-trained on the colossal clean crawled corpus (C4) dataset (a massive collection of clean English text scraped from the web), and on a mixture of unsupervised

[1]Some fields are truncated in the figure for the sake of clarity.


and supervised tasks, including question answering and natural language inference. T5 models each problem into a text-to-text format: a text input is provided to the model, where the task to be performed is indicated as prefix; its output is in text format.

T5-small can be fine-tuned for different type of tasks, such as summarization, classification, and translation. In this study, we fine-tuned the pre-trained model (retrieved from the Hugging Face transformer library) on the task of generating sequence of service invocations from raw logs. This task is modeled as a *translation* task, where the raw logs represent the input, i.e., the interesting log entries collected in operation, while the services sequence is the expected output, i.e., the sequence of invocations generating the log entries, which are subsequently used to improve regression test suites by including representative test cases suggested by the raw logs.

The fine-tuning has been carried out using the Training set generated during the pre-processing step. The training set is further divided in 90% for training and 10% for testing, with the latter used to assess the performance of the model after each training epoch. The model has been trained for 18 epochs, with a 0.00001 learning rate and 0.01 weight_decay. The *Levenshtein distance* is used to evaluate the performance of the model after each epoch. This metric is used to measure the distance between two services sequences, i.e., compare the actual services sequence (the label) and the generated

one.[2]

### Test cases generation

The fine-tuned transformer is used to generate the test cases suggested by the raw logs. The generation makes use of `uTest` Giamattei et al. (2022), the pairwise combinatorial strategy we introduced in our previous deliverables, which leverages OpenAPI specifications of microservices to generate test cases, as well as to execute them and collect the obtained results.

The test cases generation stage is shown in Figure 3.3. Each entry int the Test set is processed through three main steps: (*i*) the raw logs field of the entry is provided to the fine-tuned transformer, which generates the corresponding services sequence; (*ii*) the generated sequence is used to perform a look-up operation into the services API of the target system, with the aim to find the API of the services included within the sequence provided by the transformer; (*iii*) the identified services API are then passed to `uTest`, which generates the test cases encompassing valid input values.

Figure 3.4[3] shows an example of prompt provided to the fine-tuned transformer for services sequence generation in the context of the TT

[2]To directly apply the metric, we replaced the service names with single characters, e.g., the sequences `ts-station-service--ts-food-service--ts-route-service` and `ts-station-service--ts-travel-service--ts-route- service` in TT become `A-B-C` and `A-D-C`, with 1 as Levenshtein distance.

[3]Raw logs are truncated in the figure for the sake of clarity.

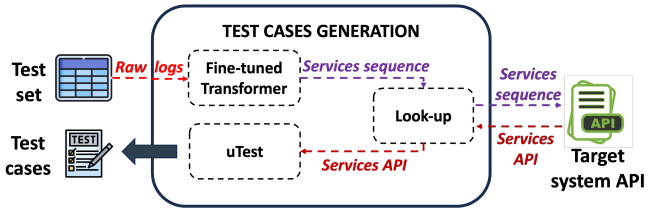



**Figure 3.3.** Test cases generation stage.

```
Prompt:

translate raw_logs to involved_services:
"2023-11-15 22:57:22.925  INFO 1 --- [o-16115-exec-15]
adminuser.service.AdminUserServiceImpl    : ADD USER INFO : UserDto(userName=e19e5c30-8b9b-
4d8f-a21d-b852ff36ca33, password=e19e5c30-8b9b-4d8f-a21d-b852ff36ca33, gender=0,
documentType=0, documentNum=23b6c7a5-3945-4ec7-9945-f36b537f27a6, email=string)
2023-11-15 22:57:22.932  INFO 1 --- [io-12342-exec-1]
user.service.impl.UserServiceImpl         : Save User Name id\u00ef\u00bc\u0161e19e5c30-8b9b-
4d8f-a21d-b852ff36ca33
…{TRUCATED}…"

Expected Output: ts-user-service--ts-admin-user-service
Obtained Output: ts-user-service--ts-admin-user-service
```

**Figure 3.4.** Example of prompt to the fine-tuned transformer for the TrainTicket subject.

subject. As it can be noted, the prompt asks the transformer to generate the *involved_services* (i.e., the services sequence) from the *raw logs* provided as input. In this case the generated services sequence exactly matches the expected output (as it can be inferred comparing *Expected output* and *Obtained output* in Figure 3.4).

Given a service from the generated sequence, the look-up operation selects the API that matches the service name; if multiple matches are found, APIs that support the POST HTTP method are preferred as it is commonly used to generate new entities in the target system.

After generating the test cases, `uTest` runs them on the target system and reports the obtained results, including number of executed tests,

**Table 3.1.** Coverage metrics.

| Metric | Description |
|---|---|
| *Status code class*<br><br>*(2 classes)* | A test suite reaches 100% status code class coverage when it is able to trigger both correct and erroneous status codes.<br>If it triggers only status codes belonging to the same class (either correct or erroneous), coverage equals 50%. 2XX class<br>represents a correct execution and 4XX and 5XX classes represent an erroneous execution. |
| *Status code class*<br><br>*(3 classes)* | As the previous one, but considers 2XX, 4XX and 5XX classes separately. A test suite reaches 100% status code class coverage<br>when it triggers all the three status code classes, 66% in case of two classes, and 33% if just one class is triggered. |

amount of succeeded/failed tests, HTTP response codes (e.g., 200, 400, 500), and coverage metrics, which can be used to evaluate the testing campaign.

We consider both the *failure rate*, i.e., the number of failures exposed by the executed tests, and the coverage metrics defined in Martin-Lopez et al. (2019b), and summarized in Table 3.1.

The technique is evaluated on two well-known benchmark systems, namely `TrainTicket` and `SockShop`, and results are under publication Della Corte et al. (2025).

### 3.2.3 Empirical energy-consumption Assessment

In the context of the uDevOps project, we have defined and run an empirical analysis of energy consumption and performance metrics of web apps, considering 10 different Internet content platforms across 5 categories, measuring energy consumption, network traffic volume, CPU load, memory load, and frame time of their native and Web versions. This was done during WP4, details are in D4.1 and D4.2, wherein a similar solution for IoT, another context where microservice architectures was shown to be useful.

It is worth noting that, although the empirical assessment can be seen as a testing session, it is not automatically generalizable as a "technique". There are in fact manual steps to be done to apply this assessment to other applications, encompassing the creation of the testbed and the analysis of results.

### 3.2.4 Performance Degradation Assessment

Similarly to the previous case, we have defined a workload-driven test methodology to assess performance degradation over long running executions – a phenomenon called runtime software aging. In the uDevOps project, this is applied in the context of ML-based systems running under low-resource constraints, such as in a cloud-edge computing where a tiny ML task (e.g., object detection) is run at the edge nodes, with limited computational capabilities. This is also an instance of what is called AIoT

systems, created according to a microservice- like style.

Similarily to the previous case, this constitutes an example of workload-driven testing for empirical assessment, which can indeed be replicated for another system but entailing manual intervention for tailoring the workload specification and testbed creation.

This was also done during WP4, details are in the corresponding deliverables and referenced papers.

### 3.2.5 ML Services Assessment

A trend we addressed during WP5 was the assessment of reliability of ML software, corresponding to *accuracy* in this case. As ML algorithms are increasingly *servitized*, microservice architectures encompassing ML services are more and more common. In this perspective, we devised sampling-based testing solutions (as those defined in WP3) for both ML image classification algorithms using DNNs and for Large Language Models (LLM) for a specific task – we opted for sentiment analysis.

A countless number of software systems today rely on DNN and LLM predictions. Before release, engineers need to test the DNN/LLM to estimate their accuracy (i.e., probability of not having mispredictions). This allows to establish a release criterion and to correct or fnie-tune the DNN/LLM until the criterion is met.

Within the reference scenario, an AI model is designed to function

within a specific context and is trained using a training dataset. The tester's aim is to choose a small, yet representative, subset of (unlabeled) inputs from an operational dataset. These inputs serve as test cases to evaluate the model accuracy. The manual labeling process is costly. The challenge lies in creating a compact test set that can provide an unbiased and high-confidence assessment of the model's accuracy. Simultaneously, the testers are keen on identifying mispredictions, which are crucial for debugging and retraining. Thus, the objectives are threefold: to develop a small dataset that can accurately estimate model accuracy and effectively highlight mispredictions.

To address this, the study introduces a probabilistic, sampling-based operational testing approach tailored for DNNs amd for LLMs, for, respectively, image classification and natural language processing. This methodology uses auxiliary variables – specifically, prediction confidence, test-training distance metrics, and entropy — to guide the sampling process and improve the efficiency of manual labeling efforts.

The core idea is to sample a small, representative subset from the operational dataset that provides a reliable estimate of the modelps accuracy and exposes likely failures. Various sampling algorithms are explored, including:

- Simple Unequal Probability Sampling (SUPS)
- Rao, Hartley, and Cochran Sampling (RHC-S)

- Stratified Simple Random Sampling (SSRS)
- Gradient-Based Sampling (GBS)
- Two-stage Unequal Probability Sampling (2-UPS)
- DeepEST (DNN Enhanced Sampler for Testing)

We have tested the technique, called DeepSample, on DNNs for image classification Guerriero et al. (2024) and on an LLM for sentiment analysis, DistilBERT Asgari et al. (2025), using multiple datasets (ImageNet, MNIST, Cifar, Udacity for images, SST2, IMDB for sentiment analysis) showing consistent benefits in both estimation error and failure exposure. This operational testing framework enables cost-effective, accurate, and explainable evaluation of DNNs and LLMs. It provides a practical MLOps-oriented solution for ongoing model validation in real-world environments. Details are in two papers published recently Guerriero et al. (2024) and Asgari et al. (2025).

## 3.3 SERVICES FOR QUALITY IMPROVEMENT

While the previous techniques give estimate of a quality of interest, thus can be used at *acceptance testing* stage, the following techniques are for exposing possible failures in the compliance to functional or non-functional requirements. Thus they are more oriented to give feedback at *development testing* stage.




### 3.3.1 Functional and Robustnes Testing Service

We briefly review this technique, introduced in the previous WPs. This solution has been packaged and containerized into a set of services that can be used to automatically generate grey-box tests, taking the API specification as input.

The grey-box testing strategy for Microservice Architectures (MSAs) enhances system observability to identify failures and assess internal test coverage, crucial for complex distributed systems. Unlike black-box testing, it allows visibility into both edge and internal service interactions, facilitating tracing and understanding of failures. Grey-box testing, implemented through a solution like MacroHive, generates test cases using microservice APIs and tracks internal requests using a service mesh pattern. This approach provides detailed visibility, detecting both masked failures (internal issues hidden by correct edge responses) and propagated failures (internal issues causing visible edge errors)..

The solution detects both edge and internal failures without manual inspection. It uses a service mesh (called uProxy and uSauron) to monitor request flows and a test generator (uTest) that creates combinatorial test cases from OpenAPI specs. Proxies intercept traffic and send detailed logs to uSauron, which maps them to specific tests to identify failure causes and service dependencies. The system supports efficient, automated, and fine-grained testing of complex microservice interactions. Details are in past




deliverables and in the referenced papers.

The service can easily be configured to focus more on functional or robustness testing, with variants of tests generation available for both.

### 3.3.2 Performance Configuration Testing Service

Performance testing of microservice applications is essential for understanding how workloads affect user experience and resources usage, for choosing among deployment alternatives, and to engineer proper mitigation means, like performance bug removal and capacity planning. In designing performance tests, engineers need to balance the goal of exposing performance issues with the stringent release deadlines typical of agile microservice development processes. As running tests for all possible workload configurations is unfeasible, the challenge is how to find *critical* configurations, expected to cause performance failures. Systems may exhibit performance failures, e.g., due to lack of robustness against heavy loads , or even at low load, due to other types of faults.

Existing microservices performance testing techniques and tools support the automation of tasks like tests execution de Camargo et al. (2016), the deployment of test configurations given a manual specification in a Domain Specific Language Ferme and Pautasso (2018), or the generation of tests with a workload inferred from the observed operational profile to mimic the expected usage Avritzer et al. (2018, 2020); Camilli et al.

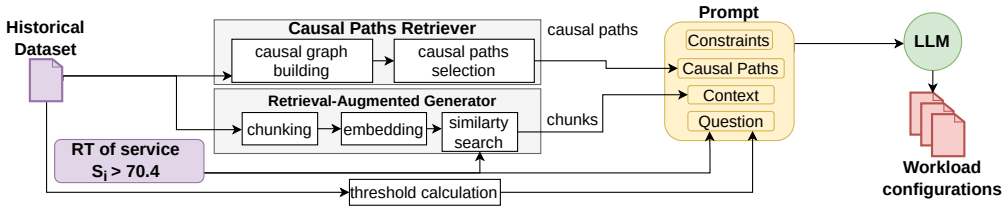



**Figure 3.5.** CALLMIT architecture

(2022a,b). During WP5, we devised a technique named CALLMIT (CAusality-enhanced Large Language model for MIcroservices performance Testing), a strategy to automate the generation of critical workload configurations and of the subsequent test cases, with the aim of saving testing effort by running only those ones more likely to lead to a degradation.

CALLMIT harnesses Causal Reasoning and Large Language Models (LLM). Specifically, we devise a new Retrieval Augmented Generation (RAG) strategy to improve the prompt to the `LLM` that uses the (automatically-inferred) causal relationships between the microservices' performance metrics to provide configurations more likely to produce a failure.

CALLMIT is experimentally evaluated in several variants, and compared to a conventional RAG technique on three popular subjects The results are that causal models can actually improve the ability of LLM to correctly identify performance-critical workload configurations.

Figure 3.5 shows the architecture of the proposed framework. CALLMIT exploits an LLM augmented with a causal graph for detecting workload configurations that lead to performance issues in a microservices application. A performance issue is defined as any instance where one or


more observed performance metrics exceed a specified threshold during a testing or monitoring session.[4] A workload configuration is defined as a triple $<$*User size*, *Load type*, *Spawn rate*$>$, where:

- *User size (US)*: number of concurrent users;
- *Load type (LT)*: operational profile, specified categorically (e.g., uniform, unbalanced towards some services);
- *Spawn rate (SR)*: number of users to spawn per second.

CALLMIT uses a dataset obtained by monitoring past executions (as usual in microservice applications) in order to guide the generation of workload configurations. As in past works Giamattei et al. (2024), we build the dataset from monitoring data (logs and performance metrics), inferring the workload configuration parameters (WP) (*US*, *LT*, *SR*) and the resource usage (i.e. CPU and memory usage), response time (RT) and request rate for each service. The data is used to augment the prompt given to the LLM with two items, namely a context and a set of causal paths. As for the former, historical data is used like in a conventional RAG system (Retrieval-Augmented Generator in Figure 3.5), providing the LLM with contextual data. As for the latter, past data is used to build a causal graph, which is given to the LLM in the form of causal paths to improve its outputs (Causal Paths Retriever in Figure 3.5).

[4]The thresholds can be either defined manually or inferred automatically as in Avritzer *et al.* Avritzer et al. (2020).




Retrieval-Augmented Generator.

CALLMIT leverages Retrieval-Augmented Generation for the purpose of supplying the LLM with contextual data. This requires to specify *how the historical dataset is partitioned* (chunking), *which embedding model is used*, and *how the chunks to be provided to the prompt are selected*. To preserve the semantics of data in the dataset, the latter is partitioned by rows, with each row serving as an individual chunk. The chunks and the question are then converted into a vector through a pretrained embedding model (all-MiniLM-L6-v2-f16). The best chunks to retrieve are identified by running a similarity search (cosine similarity) with the question.

Causal Paths Retriever.

The Causal Paths Retriever (CPR) enhances the LLM by augmenting the prompt with causal knowledge.

Let a weighted directed acyclic graph (DAG) $\mathcal{G} = (\boldsymbol{X}, \mathcal{E}, \mathcal{W})$ be the *causal graph* representing a linear causal model – specifically a Structural Equation Model (SEM) - where nodes $X_i \in \boldsymbol{X}$ are random variables, edges $e_i \in \mathcal{E}$ denote the causal relationships between them, and $\mathcal{W}$ is an $|\boldsymbol{X}| \times |\boldsymbol{X}|$ adjacency matrix $\mathcal{W} = \{w_{i,j}\}$, with $w_{i,j}$ representing the connection strength from $X_i$ to $X_j$. CALLMIT learns this model automatically from the historical dataset via a causal discovery algorithm, that is `dLiNGAM` Shimizu et al. (2011). For each service, the dataset has four variables for the considered performance metrics (request rate, RT, CPU usage, and memory

consumption), and three variables representing the workload parameters WP (user size, load type, spawn rate).

The causal model is meant to represent the relations between the WP and the performance metrics of interest (e.g., RT, memory usage) for all the services. To prompt the LLM, we consider a set of causal paths in the graph. A causal path $\mathcal{P} = X_0 \to X_1 \to \cdots \to X_n$ is a finite directed path, namely a sequence of (non-repeated) edges $(e_1, e_2, \ldots, e_{n-1}) \in \mathcal{E}$ directed in the same direction that joins a sequence of distinct vertices $(X_1, X_2, \ldots X_n) \in \boldsymbol{X}$. The task of the CPR is to get the most informative paths that link the WP to the performance metric of interest, so as to highlight what changes in the WP is more likely to affect that metric, net of possible confounders.

Since the number of nodes and edges can increase significantly as the number of services grows, and since the performance of LLM can decrease with long prompts Levy et al. (2024), CALLMIT selects a subset of causal paths. These need to be *workload-related causal paths*, i.e. paths originating from the WP nodes (i.e., US, LT and SR) and leading to the target node of interest (i.e., a node representing the service-metric pair for which we want to generate a critical configuration, e.g., response time of service $S_i$.

In order to reduce the number of selected paths and focus on the most impacting ones, we exploit the notion of *causal strength* - an estimate of how much the effect variable is expected to change for a change in the cause variable Janzing et al. (2013) - in the form of *edge strength* (or connection


strength) coefficients $\mathcal{W}w_{i,j}$ as computed by `dLiNGAM` Shimizu et al. (2011) based on the covariance matrix. The *strength of a causal path* is the sum of the causal strengths of its constituent edges.

We use the strength to select only the top-$k$ paths. In the experimentation, we test two variants of CALLMIT, top-$1$ and top-$5$. Moreover, in order to reduce the time to search for the strongest paths, we consider a *pruned* causal graph, i.e., a graph in which several edges are removed, based on a *causal strength threshold* – all the edges with a strength less than a causal strength threshold $Th_{CS}$ (set to $0.3$ during experimentation) are removed from the graph, resulting in a new graph $\mathcal{G}' = (\boldsymbol{X}, \mathcal{E}', \mathcal{W}')$ where $\mathcal{E}' = \{e \in \mathcal{E} \mid |\mathcal{W}'(e)| > Th_{CS}\}$. This gives four variants of CALLMIT (top-$1$ and top-$5$, with and without pruning).

The prompt is composed of the following sections:

- *Generation Constraints*: the constraints specify the accepted values for WP, i.e. the boundaries for the *User size*, the categories for the *Load type*, and the acceptable ranges for *Spawn rate*;
- *Causal Paths*: the causal paths selected;
- *Contextual Data*: the context is provided by the Retrieval-Augmented Generator;
- *Question*: the question invokes the LLM generation, specifying the target metric and the thresholds.




Thresholds for establishing whether a configuration is critical or not are defined according to Avritzer *et al.* Avritzer et al. (2020) (*scalability thresholds*) as $\tau_X = \mu_X + 3 \cdot \sigma_X$, where $X$ is the variable of interest, and $\mu_X$ and $\sigma_X$ are its mean and standard deviation over past executions.

Experimentation. We evaluate the benefit in detecting performance issues of augmenting the LLM prompt with causal paths. The experiments use three subjects and two LLM. These are queried to detect which workload configurations result in performance issues, by varying the user size *US* – a natural number, which we assume bounded in $[U_{min}, U_{max}]$ - the load type *LT* – which we assume to be categorical variables in the experimentation - and the spawn rate *SR*.

As evaluation metrics, we use *precision*, number of performance issues correctly detected over number of all potential performance issues predicted by the model; *recall*, number of issues correctly detected over number of actual issues; $F_1$, which is their armonic mean.

Subjects `μBench`[5] Detti et al. (2023) is an artificial microservice application composed of 10 services, with a random service dependency graph and a random stressing function per service (stress/idle function). `SockShop`[6] is a benchmarking application for cloud-native microservices testing, emulating an e-commerce site; it is composed of 8 services (6 edge services).

[5] `https://github.com/mSvcBench/muBench`.

[6] `https://github.com/ocp-power-demos/sock-shop-demo`.

**Table 3.2.** Results in detecting response time and CPU issues. Underlined: Best avg. values. Boldface: significantly different from RAG

| LLM | Performance metric | Technique | µBench | | | TeaStore | | | SockShop | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | *precision* | *recall* | $F_1$ | *precision* | *recall* | $F_1$ | *precision* | *recall* | $F_1$ |
| phi3.5 | RT | S | 0.490 | 1.000 | 0.657 | **0.782** | 0.902 | **0.837** | 0.462 | 1.000 | 0.632 |
| | | $S_5$ | 0.580 | 1.000 | 0.718 | 0.667 | 1.000 | **0.800** | 0.462 | 1.000 | 0.632 |
| | | SP | 0.300 | 1.000 | 0.458 | **0.940** | **0.576** | **0.700** | 0.462 | 1.000 | 0.632 |
| | | $SP_5$ | **0.200** | 1.000 | **0.333** | 0.667 | **0.571** | 0.615 | 0.462 | 1.000 | 0.632 |
| | | RAG | 0.430 | 1.000 | 0.599 | 0.333 | 1.000 | 0.500 | 0.462 | 1.000 | 0.632 |
| | CPU | S | **1.000** | **1.000** | **1.000** | **0.450** | **0.112** | 0.180 | **0.846** | 1.000 | **0.917** |
| | | $S_5$ | **1.000** | **0.900** | **0.945** | **0.500** | **0.125** | 0.200 | **0.846** | 1.000 | **0.917** |
| | | SP | **1.000** | 0.890 | **0.940** | **0.500** | **0.125** | 0.200 | **0.846** | 1.000 | **0.917** |
| | | $SP_5$ | **1.000** | **1.000** | **1.000** | **0.500** | **0.125** | 0.200 | **0.846** | 1.000 | **0.917** |
| | | RAG | 0.870 | 0.756 | 0.807 | 0.134 | 0.417 | 0.200 | 0.815 | 1.000 | 0.898 |
| gemini | RT | S | 0.720 | 1.000 | 0.837 | 1.000 | 1.000 | 1.000 | **0.500** | 1.000 | **0.666** |
| | | $S_5$ | **0.770** | 1.000 | **0.869** | 1.000 | 1.000 | 1.000 | **0.454** | 1.000 | **0.621** |
| | | SP | 0.690 | 1.000 | 0.816 | 1.000 | 1.000 | 1.000 | **0.500** | 1.000 | **0.666** |
| | | $SP_5$ | 0.700 | 1.000 | 0.824 | 1.000 | 1.000 | 1.000 | **0.385** | 1.000 | **0.556** |
| | | RAG | 0.680 | 1.000 | 0.807 | 1.000 | 1.000 | 1.000 | 0.308 | 1.000 | 0.471 |
| | CPU | S | 1.000 | 1.000 | 1.000 | 1.000 | **1.000** | **1.000** | **1.000** | 1.000 | **1.000** |
| | | $S_5$ | 1.000 | 1.000 | 1.000 | 1.000 | **1.000** | **1.000** | **1.000** | 1.000 | **1.000** |
| | | SP | 1.000 | 1.000 | 1.000 | 1.000 | **1.000** | **1.000** | **1.000** | 1.000 | **1.000** |
| | | $SP_5$ | 1.000 | 1.000 | 1.000 | 1.000 | **1.000** | **1.000** | **1.000** | 1.000 | **1.000** |
| | | RAG | 1.000 | 1.000 | 1.000 | 1.000 | 0.878 | 0.935 | 0.923 | 1.000 | 0.960 |

TeaStore[7] v. Kistowski et al. (2019) is a benchmark emulating a web store with automated customer orders, comprising a registry and five services.

Baseline and CALLMIT variants As baseline, we use the LLM simply integrated with a RAG system (called RAG) and compare it against four variants of CALLMIT, which enhances RAG with the causal paths retriever. We consider these variants for paths selection:

[7] https://github.com/DescartesResearch/TeaStore-




**S** top-$1$: It selects the **S**trongest *causal path* (one for each workload parameter);

**S**$_5$ top-$5$: It selects the five **S**trongest *causal path* (five for each workload parameter);

**SP** top-$1$ with pruning: It selects the **S**trongest *causal path* (one for each workload parameter) on the **P**runed graph;

**SP**$_5$ top-$5$ with pruning: It selects the five **S**trongest *causal path* (five for each parameter) on the **P**runed graph;

Experiments Setup. For the experiments, we have synthesized a historical dataset for each subject. Specifically, we considered: all values in ($U_{max} - U_{min}$) for the user size *US*; three values for the load type *LT* – one for balanced workload (every service receives the same amount of requests), and two with a workload unbalanced toward a subset of services simulating two use cases (CPU- and memory-intensive, respectively) - three values for spawn rate *SR* $\{1,5,10\}$. Each workload configuration lasts three minutes and is run five times, with a two-minute pause between runs to avoid carryover effects (consecutive runs influencing each other, e.g., in CPU and memory usage).

To construct the causal graph, we provide the `dLiNGAM` *causal structure discovery* algorithm with the historical dataset and prior knowledge. Prior knowledge is specified as required or forbidden edges. We defined the edges connecting *WP* to the request rate for each service as

mandatory.

Finally, considering the LLM scaling law outlined in Kaplan et al. (2020), we opted for two LLM with a remarkably difference in size: `phi3.5` et al. (2024) and `gemini-1.5-flash` (`gemini`) Team (2024).

Results

Table 3.2 reports the results (average over 10 repetitions) with the Dunn's test results comparing each variant *vs* RAG ($\alpha = .05$) Dunn (1964). Results about memory for `SockShop` are in Table 3.3, as `µBench` and `TeaStore` did not exhibit memory issues.

**Does the causal path retriever improve performance?** CPR shows superior performance compared to `RAG`, achieving consistent improvements across all metrics. On average, CPR outperforms the baseline by $6.5\%$ in $F_1$ score. This improvement is primarily driven by a $13.1\%$ increase in *precision*, despite a slight decrease in *recall* ($1.68\%$). The *recall* reduction is primarily

**Table 3.3.** Results in detecting memory issues on `SockShop`

| Technique | `phi3.5` | | | `gemini` | | |
|---|---|---|---|---|---|---|
| | *precision* | *recall* | $F_1$ | *precision* | *recall* | $F_1$ |
| S | 0.569 | 1.000 | 0.724 | 0.815 | 1.000 | 0.898 |
| $S_5$ | 0.638 | 1.000 | 0.779 | 0.815 | 1.000 | 0.898 |
| SP | **0.785** | 1.000 | **0.879** | 0.808 | 1.000 | 0.893 |
| $SP_5$ | 0.692 | 1.000 | 0.818 | **0.846** | 1.000 | **0.917** |
| RAG | 0.669 | 1.000 | 0.800 | 0.769 | 1.000 | 0.870 |


because RAG, which does not exploit any kind of knowledge, tends to always generate some configuration without specifically looking for critical ones (hence, its low precision) – this happens mainly with TeaStore, where the large reduction impacts the average, while *recall* is 1 in most of the other cases.

On µBench, CPR achieves a consistent improvement, with a gain of $4.45\%$, $5.1\%$, and $4.24\%$ in *precision*, *recall*, and $F_1$ score, respectively. On TeaStore, CPR yields a larger improvement, $31.8\%$, in *precision* but with a decrease in *recall* ($12.47\%$). Nonetheless, the $F_1$ score improves by $11.31\%$. On SockShop, where both CPR and the baseline maintain a *recall* of $1$, the performance improvement is measurable in terms of *precision*, which increases by $7.95\%$.

**Does the causal strength selection strategy improve performance?** We compare the selection strategies (S, $S_5$) against RAG. On average, the two strategies enhance performance by $13.73\%$ in *precision*, $8.44\%$ in $F_1$ score, with a slight decrease in *recall*. The largest impact is on *precision*, consistently with the observation that RAG exhibits high *recall*.

**Does pruning impact performance?** We compare the two pruning strategies (SP and $SP_5$) against RAG. On average, they yield a $4.56\%$ increase in $F_1$ score, which is about half the improvement of non-pruning techniques.

This reduced gain is due to a $3.12\%$ decrease in *recall*. We conclude that pruning trades the reduced paths retrieval time off with performance.

**How much does CALLMIT benefit different LLM?** For `gemini`, the baseline performance is already high, so augmenting the LLM prompt with CPR has a lower impact. Nonetheless, CALLMIT improves performance on average by $5.65\%$, $1.77\%$, and $5.33\%$ in *precision*, *recall*, and $F_1$ score, respectively. For `phi3.5`, the improvements are more pronounced, expecially for $F_1$, which increases by $8.09\%$. This gain is driven by the improvement in *precision* ($24.51\%$), paid in terms of *recall* ($5.53\%$ decrease). Applying CALLMIT to a larger LLM results in a less pronounced but consistent improvement across the metrics.

The technique showed promising performance and can easily be extended to deal with other qulaity requirements. Further details are in the paper we pusblished Mascia et al. (2025).

### 3.3.3 Code Improvement

A technique for dead code elimination was proposed in WP4, briefly recall here.

To reduce failure risks during coding, Lacuna was introduced, a tool that detects and removes unused JavaScript code in web apps using static and dynamic analysis. Dead code—often introduced by third-party




libraries—can degrade performance, especially in mobile apps. Applying Lacuna to 30 real-world mobile web apps showed that dead code elimination improves load times and reduces network usage, without requiring changes to coding style or structure. Details are in WP4 and the referenced papers.

### 3.3.4 Inconsistency Architecture Detection

Inconsistent software architecture documentation (SAD)—across natural language texts and formal models—can lead to development and maintenance issues.

In WP4, an approach was proposed to enhance traceability link recovery (TLR) for automatically detecting such inconsistencies. It identifies unmentioned model elements (present in the model but undocumented) and missing elements (described but not modeled). Evaluated on open-source projects, the method achieved strong results, with an F1-score of 0.81 for TLR and accuracies of 0.93 and 0.75 for unmentioned and missing elements, respectively—outperforming existing approaches.

This strategy is well suited to support inconsistency check detection in microservice architectures. More details are in WP4 and the related references.




### 3.3.5 Sustainability-aware Design Support

To reduce risks from poor design, Funke et al. (2023) explored sustainability modeling in software systems by introducing variability features—alternative implementations of software features with differing sustainability impacts. This work was in the context of WP4.

The project partner VU extended existing decision-making frameworks to include these features and applied the approach to a real-world system (`Zahori`) through expert interviews and case study analysis. This allowed the evaluation of alternative scenarios and supported data-driven, sustainable design decisions, helping architects and stakeholders make informed choices during the design phase of microservice applications.

More details are in WP4 and the referenced paper.


# 4 FAULT PREDICTION SERVICES

## 4.1 OVERVIEW

This Section reports the techniques we implemented that support fault prediction. There are two services with this focus, both classified as improvement techniques. Both the strategies were introduced in WP2, and here are briefly recalled.

### 4.1.1 Test Case Prioritization

The objective of this technique is to support test prioritization in an MSA, namely: given a list of tests to run, the goal is to run first the ones more likely to expose failures. Prioritization is done by applying machine learning (learning-to-rank) algorithms to features of request/response and/or of the invoked Microservice that correlate more with quality metrics, such as performance (e.g., response time), reliability (e.g., status code) or coverage. The feedback allows for prioritizing test cases. The tests can be already


given, or can be generated automatically by the testing tool developed in the context of this project and described in the previous chapter, starting from the API specification of the microservices under test.

### 4.1.2 Fault Prediction

The objective of this technique is to support the early identification of commits more likely to introduce defects, namely: given an application developed in a continuous integration/DevOps setting (hence with frequent commits), the goal is to alert on those commits more likely to introduce a defect in the deployed code and to identify the metrics more stable and relevant for the prediction. This is done by applying just-in- time (JIT) prediction enriched with the feature stability score computation.

Deliverable 2.2 provide examples of usage of both these functionalities on pre-defined datasets.


# 5 FAULT REMOVAL AND FAULT TOLERANCE

## 5.1 OVERVIEW

This Section reports the techniques we implemented that support fault removal and fault tolerance.

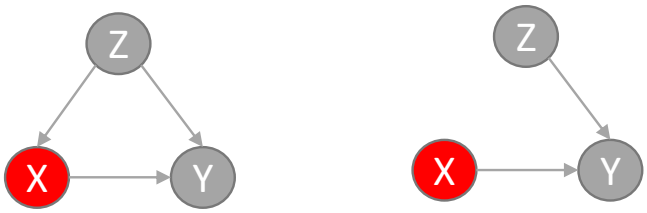
### 5.1.1 Failure Analysis

This strategy is an extension of the testing service we initially developed in WP4. Along with components of the test generation service described in WP4 (`MacroHive` service), Delivearble 4.1, we developed a further service based on *causal analysis*. `MacroHive` includes a test generator callled `uTest`, two services for monitoring, `uSauron` and `uProxy`, which monitor inter-services interaction, and a newly-developed service for root cause analysis. This latter service is called `uKnows`, which collects data by `uSauron` and `uProxy`, and exploits *causal reasoning* to determine which microservices are responsible for failures and, in general, for erroneous


behaviors.

Causality is the influence by which an event contributes to the production of other events Nogueira et al. (2022). There are two main activities in causal reasoning: Causal Structure Discovery (CSD), aimed at extracting a causal model (a mathematical representation of causal relationships between random variables) from observational data; and Causal Inference (CI), aimed at quantifying the effect of changing one or more random variables on others, starting from a causal model. The random variables describe quantity of interest (e.g., a categorical variable representing the response code of a microservice), and can be of any type (e.g., both numerical and categorical); the value they take depends on the probability distribution associated with them.

A widely adopted solution to model causality is to use *Graphical Causal Models* (GCMs). A GCM consists of a causal Direct Acyclic Graph (DAG) where nodes are random variables and edges define cause-effect (tail-arrow) relationships between couples of variables. The latter determines the impact of a change of a certain variable, called cause, over an outcome of interest, called effect. The most prevailing case is a Structural Causal Model (SCM), a GCM that uses a Functional Causal Model (FCM), where the value of each variable $X_i$ is assumed to be a deterministic function of its parents $Pa(X_i)$ and of the unmeasured disturbance $U_i$ ($X_i = f(Pa(X_i), U_i)$). An SCM is formally defined as follows.


**(a)** Original SCM

**(b)** SCM after an intervention on X

**Figure 5.1.** Example of Structural Causal Model and intervention

**Structural Causal Model (SCM)**. An SCM is a Directed Acyclic Graph $\mathcal{G} = (\boldsymbol{X}, \mathcal{E})$, where nodes $\in \boldsymbol{X}$ are random variables and edges $\in \mathcal{E}$ are the causal relationships between them. Causal relationships are described as a collection of structural assignments $X_i := f_i(Pa(X_i), U_i)$ that define the random variables $X_i$ as a function of their (endogenous) parents $Pa(X_i)$ and of (exogenous) independent random noise variables $U_i$.

CI aims at estimating the effect of setting one variable $X_k$ to a specific value "$x$" ((i.e., doing an intervention on $X_k$) on one or more variables $X_i$ of interest. Pearl Pearl and Mackenzie (2018) introduced the *do-operator* (written as $P(X_i|do(X_k = x))$), a mathematical representation of physical intervention, that changes the SCM graph by removing causal relations of $X_k$ with its predecessors and replacing the definition $X_k := f_k(Pa(X_k), U_k)$ in the SCM with $X_k := x$. An example is in Figure 5.1a, where X, Y, and Z represent the behavior (properly codified - e.g., the HTTP status codes of responses in our case) of 3 microservices ($M_x$, $M_y$, and $M_z$) calling each other. A variable $Z$ is a function of its parents and of a variable $U_z$ capturing




the random noise – the noise variables are usually not represented in the graph for simplicity, but there is a hidden node, $U_x$, $U_y$, $U_z$, associated with each variable with an arrow pointing to $X$, $Y$ and $Z$ respectively. When there is no parent, say for $Z$, the equation becomes $Z = U_z$. In the example, $U_z$ is the random variable capturing the behavior $Z$ and the value it takes depends solely on the $U_z$ distribution. The behavior $Z$ affects the other two (e.g., an erroneous/correct behavior causes others erroneous/correct behavior), and $X$ affects $Y$; thus: $X = f(Z, U_x)$ and $Y = f(X, Z, U_y)$. Suppose we want to estimate the impact of failure of $M_x$ on $M_y$ ($X \rightarrow Y$ in the graph). $Z$ is said to be a *confounder* since it causally affects both $X$ and $Y$, generating a spurious association. This means, for instance, that the failure of $M_y$ can be falsely attributed to $M_x$, even if $M_z$ was the ultimate cause. Performing an intervention leads to the SCM reported in Figure 5.1b, which allows estimating the real effect of the behavior $X$ on $Y$, because $X$ is no longer influenced by $Z$.

Causal Inference requires a causal model, which can be built in two main ways: by intervening on variables and observing the post-intervention distributions (through controlled experiments), or by using CSD algorithms that aim to seize causal structure from observational data. Causal discovery algorithms can be divided into constraint-based (e.g., PC, FCI, RFCI), score-based (e.g., GES, FGES, GFCI), and FCM-based (e.g., LinGAM) Glymour et al. (2019). An extensive discussion of CSD algorithms can be found in Nogueira et al. (2022).

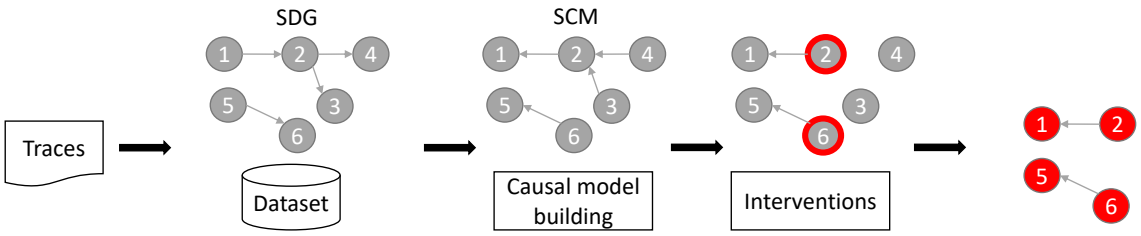



**Figure 5.2.** Workflow of `uKnows`

`uKnows` leverages the power of causal reasoning to automatically infer causal relations between microservices. Figure 5.2 shows its workflow. It extracts the Service Dependency Graph (SDG) from the output of `uSauron`, collected during the testing sessions, and derives a Direct Acyclic Graph (DAG). Multiple tests in a session have different inputs, thus dependencies based on different inputs will be captured by the graph.

Then, `uKnows` parses and transforms traces in a dataset containing an entry for every HTTP request, with the microservices involved in the interaction. The DAG and the dataset are the input for the CSD algorithm. In particular, the SDG is used as prior knowledge; it specifies required edges in the causal graph: a causal relation in the causal graph must exist between two microservices connected in the SDG. Then, the CSD algorithm infers the final SCM by fitting the causal graph with the dataset. This model characterizes cause-effect relationships between microservices.

For CSD, we use `py-causal`,[1] a python library that wraps the Java tool `Tetrad` Ramsey et al. (2018). We use the FCI algorithm Spirtes et al. (2001),

[1]py-causal v1.2.1, `https://zenodo.org/record/3592985`.


one of the simplest solutions, with the default settings.[2] The FCI algorithm starts with a complete undirected graph connecting all the nodes and applies conditional independence tests to remove edges (with only edges that indicate potential causal relationships). The method tests possible d-separations $X \perp\!\!\!\perp Y|Z$ in the skeleton. If there is at least a variable in Z that d-separates[3] the edge, then it is removed Nogueira et al. (2022). Finally, FCI applies several rules to direct the edges Spirtes et al. (2001). It gives asymptotically correct results even in the presence of *confounders* (unobserved direct common cause of two measured variables) Glymour et al. (2019). As shown in Figure 5.2, the SCM built from the SDG will have inverted arrows. For instance, a failure in microservice $M_1$ is likely to find causes in downstream nodes ($M_2$, $M_3$, and $M_4$) and not the other way around.

In the *Interventions* step, `uKnows` does interventions to predict what would be the effect of a failure (in robustness testing) or of an erroneous behavior (in functional testing) of a microservice on the others causally related to it. We consider erroneous behavior HTTP status codes of classes 4xx and 5xx Martin-Lopez et al. (2019a); we consider failures only response codes of the class 5xx.

[2]FCI settings: testId=fisher-z-test, depth=-1, maxPathLength=-1, completeRuleSetUsed=False.

[3]Let $X$,$Y$, and $Z$ be disjoint subsets of all vertexes in the DAG. $Z$ *d-separates* $X$ and $Y$ just in case every path from a variable in $X$ to a variable in $Y$ contains at least one vertex $X_i$ such that either $X_i$ is a *collider* (i.e. the arrows converge on $X_i$ in the path) and no descendant of $X_i$ (including $X_i$) is in $Z$, or $X_i$ is not a collider, and $X_i$ is in $Z$.




Assume we want to assess the effect of a failure of $M_2$ on $M_1$ and $M_6$ on $M_5$. We perform interventions on $M_2$ and $M_6$, setting them to fail (Figure 5.2) and thus removing all the edges coming from $M_2$ and $M_6$ parents. Then, using a new data sample derived by sampling from the post-intervention distributions, we estimate the outcome by looking at the sample statistics. If a failure is predicted, a causal relation is detected and reported in the output (Figure 5.2, last step). Note that the interventions are not physical, but queries to the model - a real intervention would be to inject a failure in $M_2$, or to inject a fault to cause its failure. This allows saving the cost of executing such tests, by exploring the effect of numerous hypothetical failures of microservices without actually injecting faults or failures. Clearly, this is traded off by the accuracy of the estimate, that when using the causal surrogate model is a *prediction* of the effect, ultimately depending on the model accuracy.

To show how interventions are performed, let us consider a chain of invocations as a sequence of microservices $M_1, ..., M_n$, where $M_1$ calls $M_2$ that in turn calls $M_3$, and so on till $M_n$. The chain in the SCM will have all the edges directed in the opposite direction, namely from $M_i$ to $M_{i-1}$. The interventions are done as follows: for each node $M_k$ in the chain with at least one out edge (we do not intervene on $M_1$ as it does not have effects on others) the engine intervenes on the model querying what would happen to the $M_k$'s successors (i.e., *do they fail or not?*) if $M_k$ exhibited an erroneous behavior or failed. In other words, we evaluate $P(M_i|do(M_k = fail))$


for each $i - th$ successor of $M_k$ in the chain. The query on the model returns a value for each microservice in the chain corresponding to its expected behavior (HTTP status code) if $M_k$ fails. From the output of the interventions, `uKnows` builds an output graph by drawing edges from the nodes which it intervened on to each node for which a failure is predicted. The result of these interventions is a graph highlighting the microservices causally involved in erroneous behaviors (functional testing) or failures (robustness testing). These are the microservices that developers should focus on, since they cause erroneous behavior in other microservices in the system. For example, in Figure 5.2 they are nodes $M_2$ and $M_6$. The interventions are done via `do-why` Sharma et al. (2019), a Microsoft's library to perform inference on causal models.

### 5.1.2 Log-based Anomaly Detection

Operational failures can be mitigated through early error detection, particularly using anomaly detection techniques.

In the context of WP4, Cinque et al. (2022) from the CINI partner introduced *Micro2vec*, a log mining approach for microservice-based systems that converts diverse log data into numeric representations without requiring prior format or application knowledge.

Validated on both a Clearwater IP system and an air traffic control system, *Micro2vec* enables effective anomaly detection by identifying


patterns across multiple log sources. It also supports the generation of synthetic log variants, enhancing detection accuracy beyond traditional one-class classifiers.

This functionality is extremely useful to identify problems at runtime and trigger the subsequent root cause analysis. Details for this are in Deliverable D4.1

### 5.1.3 Energy Anomaly Detection and Root Cause Analysis

Within the context of this activity, we have tailored algorithms for anomaly detection (AD) and root cause analysis (RCA) for detecting and tracing energy consumption bottlenecks.

In microservice-based systems anomalous CPU or memory usage observed in a particular service may reflect changes in request patterns or shifts in resource demands across interconnected services. Variations in system activities can lead to increased loads or create bottlenecks, thereby limiting subsequent processes. These intricate interactions have a substantial impact on the total energy consumption of the system, making the detection and resolution of inefficiencies quite challenging.

Consequently, evaluating the effectiveness of anomaly detection and root cause analysis techniques tailored to energy consumption issues in distributed computing contexts is crucial. DevOps teams utilize Anomaly Detection (AD) algorithms in production to oversee system operations and


pinpoint deviations from typical patterns, like resource spikes or latency concerns.

Root Cause Analysis (RCA) algorithms work in conjunction with AD by identifying the primary causes of these anomalies, facilitating resolution by the teams. Both AD and RCA are valuable tools for enhancing system reliability and reducing downtime in practical scenarios.

Within WP5, we examined the efficacy of AD and RCA algorithms for identifying and diagnosing energy consumption anomalies arising from performance issues within microservice-based systems. By intentionally introducing performance-driven energy anomalies, such as increased CPU and memory consumption within a service, the capability of these algorithms is evaluated. We evaluated AD and RCA in two popular benchmarks, SockShop, having 13 services, and Train Ticket, having with 41 services, are used as experimental platforms. Monitoring is conducted using Prometheus for collecting service-level metrics like CPU and memory usage, and Scaphandre for gathering service-level power consumption data, which is then transformed into energy usage statistics. The AD and RCA algorithms applied in this investigation are chosen from existing studies, prioritizing those suitable for multivariate data analysis, requiring minimal processing, and having openly accessible code to ensure integration with collected performance and energy metrics.

The algorithms we tailored for the energy consumption AD and RCA

are in teh following Tables.

**Table 5.1.** Anomaly Detection Algorithms used in this study

| Algorithm | Year | Type | Characteristics |
|---|---|---|---|
| Birch Zhang et al. (1996) | 1996 | Unsupervised | Hierarchical Clustering, Scalable |
| iForest Liu et al. (2008) | 2008 | Unsupervised | Isolation-based, Scalable |
| OC-SVM Shin et al. (2005) | 2005 | Semi-supervised | Binary Classification, Kernel-based, Sensitive |
| LOF Breunig et al. (2000) | 2000 | Unsupervised | Outlier Detection, Density-based, Sensitive |
| KNN Cover and Hart (1967) | 1967 | Unsupervised | Classification, Instance-based, Sensitive |

**Table 5.2.** Root Cause Analysis Algorithms used in this study

| Algorithm | Year | Type | Characteristics |
|---|---|---|---|
| MicroRCA Wu et al. (2020) | 2020 | Topology Graph-based Analysis | Random Walk, Models Anomaly |
| | | | Propagation, Lightweight |
| CausalRCA Xin et al. (2023) | 2023 | Causal Graph-based Analysis | Gradient-based, Fine-grained |
| RCD Ikram et al. (2022) | 2022 | Causal Graph-based Analysis | Hierarchical Learning, Lightweight, Scalable |
| CIRCA Li et al. (2022) | 2022 | Causal Graph-based Analysis | Cause Inference, Regression-based, |
| | | | Descendant Adjustment |

With this, we are able to provide AD and RCA services for energy consumption. Technical details and results on how well each algorithm performs for these two tasks are in a paper we published recently Floroiu et al. (2024).

### 5.1.4 Performance Degradation Assessment

We have finally integrated a strategy designed to evaluate performance degradation over extended operation periods, an issue referred to as runtime *software aging* as per Cotroneo et al. (2014). This was developed in WP4 and is mentioned in Deliverable D4.1, thus just briefly recalled here.

We ran the evaluation for ML-based systems functioning under resource constraints typical of cloud-edge computing, where a small-scale ML task, such as object detection, is executed on edge nodes with limited processing power, which can be encapsulated by microservices. This scenario is an example of AIoT systems, structured similarly to a microservice architecture.

Effective object detection is a central challenge in Computer Vision. Numerous algorithms aim to satisfy two seemingly contradictory objectives: achieving high accuracy and maintaining efficiency, all while operating in real-time with high reliability. These algorithms often run continuously for prolonged durations, as seen in scenarios like video surveillance or autonomous vehicles, which exposes them to the risk of software aging. Within the uDevOps framework Pietrantuono et al. (2022) investigate software aging in contemporary object detection algorithms. We specifically conducted long-term experiments to examine how software aging affects various algorithms, implementation libraries, and datasets. We gathered data on resource usage (e.g., free/buffer/cache memory, resident memory




size) and performance metrics (e.g., frames per second) to statistically examine the presence and magnitude of aging phenomena and differentiate between various configurations (i.e., algorithms, libraries, datasets). The findings revealed that all aging indicators employed in our study exhibited resource or performance degradation, irrespective of the specific algorithm, library, or dataset. Additionally, four out of six aging indicators, exceeding 50%, highlighted major trends. More comprehensive insights are provided in the work by Pietrantuono et al. (2022). Although this is not implemented as a standalone service, the testing strategy therein used can easily be integrated, as it consists in defining long-running test workloads, gathering data, and run our analysis on collected data to establish if aging is present or not.


# 6 CONCLUSION

This document presented the work done in WP5 for the definition of the overall testing process for quality assessment and improvement of microservice systems in the context of a DevOps engineering process. This work consisted in putting together the algorithms and techniques developed in WP2 (for learning-based algorithms) WP3 (for sampling-based algorithms) and WP4 (for assessment and improvement) under the same process, as well as in defining new techniques to support either the asssessment or improvement of one of the quality attributes of interest including reliability, performance, energy consumption.

We have first described the process-level view, with the services at its core whose aim is to provide an assessment or improvement functionality starting from gathered data over DevOps releases. Then, we reviews each individual service, describing more in details those services developed in WP5.

Each technique at the core of these services has been prototyped,


at various maturity levels, and experimented against popular microservice system and/or dataset benchmarks. This forms the basis for a future implementation of fully functional framework for continuous Microservice DevOps Engineering testing.


## REFERENCES

Arcuri, A. (2021). "Automated black- and white-box testing of restful apis with evomaster." *IEEE Software*, 38(3), 72–78.

Asgari, A., Guerriero, A., Pietrantuono, R., and Russo, S. (2025). "Adaptive probabilistic operational testing for large language models evaluation." *To appeaa in Proceedings of the The 6th ACM/IEEE International Conference on Automation of Software Test (AST 2025)*.

Atlidakis, V., Godefroid, P., and Polishchuk, M. (2019). "RESTler: Stateful REST API Fuzzing." *IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, 748–758.

Avritzer, A., Ferme, V., Janes, A., Russo, B., Schulz, H., and van Hoorn, A. (2018). "A quantitative approach for the assessment of microservice architecture deployment alternatives by automated performance testing." *Software Architecture*, C. E. Cuesta, D. Garlan, and J. Pérez, eds., Cham, Springer International Publishing, 159–174.

Avritzer, A., Ferme, V., Janes, A., Russo, B., van Hoorn, A., Schulz, H., Menasché, D., and Rufino, V. (2020). "Scalability assessment of microservice architecture deployment configurations: A domain-based approach leveraging operational profiles and load tests." *Journal of Systems and Software*, 165, 1–16.

Breunig, M. M., Kriegel, H.-P., Ng, R. T., and Sander, J. (2000). "Lof: identifying density-based local outliers." *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, 93–104.

Camilli, M., Guerriero, A., Janes, A., Russo, B., and Russo, S. (2022a). "Microservices integrated performance and reliability testing." *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, AST '22, New York, NY, USA, Association for Computing Machinery, 29–39, <https://doi.org/10.1145/3524481.3527233>.

Camilli, M., Janes, A., and Russo, B. (2022b). "Automated test-based learning and verification of performance models for microservices systems." *Journal of Systems and Software*, 187, 1–22.

Cinque, M., Della Corte, R., and Pecchia, A. (2020). "An empirical analysis of error propagation in critical software systems." *Empirical Software Engineering*, 25(4), 2450–2484.

Corradini, D., Zampieri, A., Pasqua, M., Viglianisi, E., Dallago, M., and Ceccato, M. "Automated black-box testing of nominal and error scenarios in restful apis." *Software Testing, Verification and Reliability*, 2022, e1808.

Cotroneo, D., Natella, R., Pietrantuono, R., and Russo, S. (2014). "A survey of software aging and rejuvenation studies." *J. Emerg. Technol. Comput. Syst.*, 10(1).

Cover, T. and Hart, P. (1967). "Nearest neighbor pattern classification." *IEEE transactions on information theory*, 13(1), 21–27.

de Camargo, A., Salvadori, I., Mello dos Santos, R., and Siqueira, F. (2016). "An architecture to automate performance tests on microservices." *Proceedings of the 18th International Conference on Information Integration and Web-Based Applications and Services (iiWAS)*, ACM, 422–429.


## *REFERENCES*

Della Corte, R., Pietrantuono, R., and Russo, S. (2025). "Log-driven testing of microservice systems with transformer." *To appear in IEEE Internation Conference on Web Services*.

Detti, A., Funari, L., and Petrucci, L. (2023). "$\mu$Bench: An Open-Source Factory of Benchmark Microservice Applications." *IEEE Transactions on Parallel and Distributed Systems*, 34(3), 968–980.

Dunn, O. J. (1964). "Multiple comparisons using rank sums." *Technometrics*, 6(3), 241–252.

et al., M. A. (2024). "Phi-3 technical report: A highly capable language model locally on your phone, <https://arxiv.org/abs/2404.14219>.

Ferme, V. and Pautasso, C. (2018). "A Declarative Approach for Performance Tests Execution in Continuous Software Development Environments." *ACM/SPEC International Conference on Performance Engineering*, ACM, 261–272.

Floroiu, M. S., Russo, S., Giamattei, L., Guerriero, A., Malavolta, I., and Pietrantuono, R. (2024). "Anomaly detection and root cause analysis of microservices energy consumption." *2024 IEEE International Conference on Web Services (ICWS)*, 590–600.

Giamattei, L., Guerriero, A., Malavolta, I., C. Mascia, R. P., and Russo, S. (2024). "Identifying performance issues in microservice architectures through causal reasoning." *2024 IEEE/ACM International Conference on Automation of Software Test (AST)*, 149–153.

Giamattei, L., Guerriero, A., Pietrantuono, R., and Russo, S. (2022). "Assessing Black-box Test Case Generation Techniques for Microservices." *Quality of Information and Communications Technology*, A. Vallecillo, J. Visser, and R. Pérez-Castillo, eds., Cham, Springer International Publishing, 46–60.

Glymour, C., Zhang, K., and Spirtes, P. (2019). "Review of causal discovery methods based on graphical models." *Frontiers in Genetics*, 10.

Golmohammadi, A., Zhang, M., and Arcuri, A. (2023). "Testing RESTful APIs: A Survey." *ACM Transactions on Software Engineering and Methodology*, 33(1), 27:1–27:41.

Guerriero, A., Pietrantuono, R., and Russo, S. (2024). "Deepsample: Dnn sampling-based testing for operational accuracy assessment." *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, Association for Computing Machinery, <https://doi.org/10.1145/3597503.3639584>.

Heorhiadi, V., Rajagopalan, S., Jamjoom, H., Reiter, M. K., and Sekar, V. (2016). "Gremlin: Systematic resilience testing of microservices." *IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 57–66.

Ikram, A., Chakraborty, S., Mitra, S., Saini, S., Bagchi, S., and Kocaoglu, M. (2022). "Root cause analysis of failures in microservices through causal discovery." *Advances in Neural Information Processing Systems*, 35, 31158–31170.

Janzing, D., Balduzzi, D., Grosse-Wentrup, M., and Schölkopf, B. (2013). "Quantifying causal influences." *The Annals of Statistics*, 41(5), 2324–2358.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. (2020). "Scaling laws for neural language models, <https://arxiv.org/abs/2001.08361>.


## *REFERENCES*

Levy, M., Jacoby, A., and Goldberg, Y. (2024). "Same Task, More Tokens: the Impact of Input Length on the Reasoning Performance of Large Language Models." *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, L.-. Ku, A. Martins, and V. Srikumar, eds., Bangkok, Thailand, Association for Computational Linguistics, 15339–15353, <https://aclanthology.org/2024.acl-long.818>.

Li, M., Li, Z., Yin, K., Nie, X., Zhang, W., Sui, K., and Pei, D. (2022). "Causal inference-based root cause analysis for online service systems with intervention recognition." *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 3230–3240.

Liu, F. T., Ting, K. M., and Zhou, Z.-H. (2008). "Isolation forest." *2008 eighth ieee international conference on data mining*, IEEE, 413–422.

Martin-Lopez, A., Segura, S., and Ruiz-Cortés, A. (2019a). "Test Coverage Criteria for RESTful Web APIs." *Proc. of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST)*, ACM, 15–21, <https://doi.org/10.1145/3340433.3342822>.

Martin-Lopez, A., Segura, S., and Ruiz-Cortés, A. (2019b). "Test coverage criteria for restful web apis." *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, A-TEST 2019, ACM, 15–21, <https://doi.org/10.1145/3340433.3342822>.

Mascia, C., Pietrantuono, R., Guerriero, A., Giamattei, L., and Russo, S. (2025). "Microservices Performance Testing with Causality-enhanced Large Language Models." *2nd ACM International Conference on AI Foundation Models and Software Engineering (FORGE)*, ACM, 149–153.

Nogueira, A. R., Pugnana, A., Ruggieri, S., Pedreschi, D., and Gama, J. (2022). "Methods and tools for causal discovery and causal inference." *WIREs Data Mining and Knowledge Discovery*, 12(2), e1449.

Pearl, J. and Mackenzie, D. (2018). *The Book of Why: The New Science of Cause and Effect*. Basic Books, Inc., USA, 1st edition.

Pietrantuono, R., Cotroneo, D., Andrade, E., and Machida, F. (2022). "An empirical study on software aging of long-running object detection algorithms." *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, 1091–1102.

Ramsey, J., Zhan, K., Glymour, M., Sanchez Romero, R., Huang, B., Ebert-Uphoff, I., Samarasinghe, S. M., Barnes, E. A., and Glymour, C. (2018). "TETRAD - A Toolbox for Causal Discovery." *8th International Workshop on Climate Informatics*.

Sharma, A., Kiciman, E., et al. (2019). "DoWhy: A Python package for causal inference.

Shimizu, S., Inazumi, T., Sogawa, Y., Hyvärinen, A., Kawahara, Y., Washio, T., Hoyer, P. O., and Bollen, K. (2011). "DirectLiNGAM: A Direct Method for Learning a Linear Non-Gaussian Structural Equation Model." *Journal of Machine Learning Research*, 12, 1225–1248.

Shin, H. J., Eom, D.-H., and Kim, S.-S. (2005). "One-class support vector machines—an application in machine fault detection and classification." *Computers & Industrial Engineering*, 48(2), 395–408.

Spirtes, P., Glymour, C., and Scheines, R. (2001). *Causation, Prediction, and Search*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA, USA, 2nd edition.


## *REFERENCES*

Team, G. (2024). "Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context, <https://arxiv.org/abs/2403.05530>.

v. Kistowski, J., Eismann, S., Grohmann, J., Schmitt, N., Bauer, A., and Kounev, S. (2019). "TeaStore - a micro-service reference application for performance engineers." *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE)*, ACM, 47–48.

Wu, L., Tordsson, J., Elmroth, E., and Kao, O. (2020). "Microrca: Root cause localization of performance issues in microservices." *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, IEEE, 1–9.

Xin, R., Chen, P., and Zhao, Z. (2023). "Causalrca: Causal inference based precise fine-grained root cause localization for microservice applications." *Journal of Systems and Software*, 203, 111724.

Zhang, T., Ramakrishnan, R., and Livny, M. (1996). "Birch: an efficient data clustering method for very large databases." *ACM sigmod record*, 25(2), 103–114.