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 D4.1. Business risk assessment techniques

November 2023

Abstract

This document reports the results of Deliverable D4.1 of the μDevOps project, entitled “Business risk assessment techniques”. 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 strategies we developed to provide a measure of the risk associated with the failure of a system in meeting quality-of-service (QoS) attributes for microservices, namely reliability, performance (and additionally energy consumption), security. These include means to assess the risk and measures to mitigate the risk, both targeted in this report. The report describes broadly about all the strategies that we have implemented for all the quality attributes we are targeting, while pointing to the project’s papers derived from such techniques for further details.

CONTENTS

CONTENTS

1 INTRODUCTION

Quality assurance activities aim at exposing potential *failures* of the system before they impact the user experience in operation, and take action to prevent or mitigate them. The quality requirements we are targeting pertain to reliability, performance, security of the system, and, in addition, to energy consumption, which is becoming a major source of cost when consumption exceeds the expectation. In the rest of this document the term *failures*, therefore, refer to deviations from functional and non-functional requirements (i.e., from the expected behaviour) – namely, the system is said to fail when it does not satisfy these requirements, including latent requirements on the above-mentioned attributes (reliability, performance, security, energy consumption).

The goal of quality assurance is not only related to the possible failures of the system in satisfying its requirements and quality goals. The *impact* that these failures have can vary a lot from context to context. A failure of a functionality, even if technically severe (e.g., a crash), may not impact the

revenue or reputation of the company if it impacts on a small number of users, or if the damages it causes are not dangerous for the environment or for other people (like in critical systems).

Therefore, a more complete measure to take into account is the *risk* associated with the failure of a functionality or of an entire system. In the following report, we discuss about how the risk can be characterized in a microservice-DevOps context, how it can be assessed and how it can be mitigated. Most of the techniques we implemented focus on assessing or improving the probability of failure (with respect to the above requirements) of a functionality, which, once combined with qualitative metrics of impact, give a more complete measure concerning the risk associated with the failure.

In the rest of the document: Section 2 will present the main set of risk metrics relevant for microservice-based software systems (**risk measurements**). These all have the form of a product between the probability of failure and a metric of impact that this failure would have, depending on the context. This captures both the possibility that a failure occurs and the assessment of what happens if the failure occurs. It is important to recall that in the uDevOps project, the focus is on: reliability, performance, and security quality attributes. In addition, we are also caring about energy consumption and, more generally, about sustainability as further quality attribute. Therefore, as said, the term *failure* is generically meant as any event that cause any of the above quality objective to be

negatively affected.

Section 3 will present the techniques developed for **risk assessment**. Sampling-based strategies developed in WP3 will be the basis to assess both the probability of failures (in terms of *probability of failure on a demand* (PFD)) and the impact measure when the historical data is available.

Section 4 presents the strategies for the mitigation of the risk (**risk mitigation**). These range from testing and debuggging, aimed at exposing failures and then correct the source of the failure, to following good engineering practices that should avoid the introduction of faults or inefficiency causes in the system.

2 RISK MEASUREMENT

In the context of software development, the concept of risk is often associated with the Spiral model by Boehm , which is a risk-driven model. This model is a cyclic, iterative model for software development that involves a risk-driven approach to developing products. The model defines a framework for continuous risk management throughout the product life cycle. It foresees a series of cycles – hence the term spiral - each of which focuses on developing a set of requirements or user needs, enhancing the product incrementally; the focus of each spiral is on mitigating the risks. In each spiral there is a phase called *risk analysis* where all the risks associated with development and operation are identified, such as **technical, financial, market-related, operational, and/or environmental** risks. These risks need to be systematically assessed and mitigated. As a sort of meta-model, this model can be used in conjunction with any other development model, like the agile ones used for microservice-based systems. Relevant to our project is the concept of *risk*, which we encompass in our quality framework as enhancement of attributes like reliability, performance, security, energy

efficiency.

A specific focus on quality-related aspects in software development driven by the concept of risk is put by **risk-based testing** Felderer and Schieferdecker (2019). Risk-based testing (RBT) is a testing approach which considers risks of the software product as the guiding factor to support decisions in all phases of the test process According to the ISTQB Standard glossary (IST), a risk is a ”factor that could result in future negative consequences and is usually expressed by its likelihood and impact”.

Risk management comprises the core activities risk identification, risk analysis, risk treatment, and risk monitoring (AS).

The activities risk identification and risk analysis are often collectively referred to as risk assessment, while the activities risk treatment and risk monitoring are referred to as risk control.

Risk-based testing uses risk (re-)assessments to steer all phases of the test process: test planning, test design and implementation, test execution, test evaluation and monitoring of key metrics to assess the effect of risk treatment/mitigation strategies put in place.

In this context, risk is the product of these two main factors: **likelihood** and **impact**. The former tells about how likely it is for a risk to happen, and is often quantified by the probability of an operational failure to occur or, more simply, by rough categories of high, medium or low occurrence chance; the latter measures how serious the consequences could be if the risk actually

occurs, typically assessed in a scale of consequences. These concepts are also used in safety-critical systems, in the functional hazard assessment phase [1] where the ”impact” depends on the hazards that a failure and/or an external event could cause on the environment. Techniques like failure mode and effect analysis (FMEA) are used in this case to anticipate the potential failures and their expected effect.

The type of risk relevant for a software system clearly depends on the domain. Generally speaking, based on what said above, any risk can be seen as the probability that an undetected fault reaches the interface (i.e., becomes a failure) and have a negative impact on the user of a system.

Failures are considered here as any deviation of a delivered service from the intended service a system is designed for; this includes functional and non-functional requirements, being reliability, performance, security and energy efficiency the ones we put the focus on. As we deal with service-based systems, we can measure the probability of failure by a discrete metric; a typical metric is **Probability of Failure on a user Demand** (PFD) . Most of work we have done is about how to estimate and how to reduce this PFD by means of testing.

Given the PFD, what characterizes the risk is the metric used for the impact. The impact in turns depend on the *failure mode*, namely the type of failure, and on its effect, namely on the *damage* it causes in terms of, e.g.,

[1] https://www.eurocontrol.int/tool/safety-assessment-methodology

direct cost caused by the failure (loss of money due to unjustified energy consumed), or indirect impact on cost caused by time-related problems (unmet deadlines), and by poor quality (user satisfaction, damage to environment or even to human life in critical systems). The latter dimension of *effect* need to consider not only *what* is the damage, but also *who* is affected by the damage and to what extent (e.g., how many users).

Hence the resulting risk combines:

- **Failure occurrence**, i.e., the PFD
- **Failure mode**, i.e., what type of failure
- **Failure effect**, i.e., what is the damage, who is affected and to what extent.

Table 2.1 report the *risk matrix* that schematizes the dimensions to consider in order to define the risk. For instance, for a performance-related mode of failure affecting the subscribed users of the system and that undermines the user satisfaction, engineers should assign a relative weight. By giving weights to all the combinations of interests, a relative scale will be defined. This, normalized in [0,1] or in any convenient scale, will be multiplied by the PFD ro derive the risk measure.

Looking at the three aspects defined above:

As for the **failure occurrence**, we rely on the PFD concept and use testing and debugging to assess and mitigate it. This will be described in the next

Failure mode	Effect (Who)	Effect (What type of damage)
Reliability	System (classes of) users	User satisfaction
Performance	(Classes of) users of a function	Loss of money
Security	Physical environment (including people)	Physical damage
Energy	Developer company	Loss of company money
		Unmet deadlines

Table 2.1. Dimensions to consider to define risk

Sections.

As for the **failure mode**, one needs to distinguish failures according to a severity scale. For instance, a crash of the system could be weighted differently from a simply slowing down or of a value failure, depending on the context. While common failure mode classification schemes could be customized (e.g., crash, hangs, value failure and time failures Powell (1992)), we adhere to the quality requirements we are targeting: reliability-related failures (distinguished in *crash*, *hang*, *value failure*), performance-related failures (including time-related failures like slow response time, but also pertaining to the use of resources such as excessive memory consumption and/or progressive memory consumption), security-related failures (e.g., distinguished by the type of vulnerability exploited and the damage it may cause, e.g., loss of user's private data (hence loss of confidentiality), exhaustion of resources via DNS-like attacks, loss of integrity of data), energy-related failures (total energy consumption or unjustified, progressive, energy consumption with time).

As for the **effect**, this clearly depends on the application domains. Some possible classes of metrics are listed below combining *who* is affected (classes of system users, users of a functionality, the surrounding physical environment, the company) with the type of damage caused. Damages could be roughly classified as related to the user satisfaction, loss of user money, physical damages, loss of company money. Many other combinations are possible:

- **User-based**. Metrics focusing on the effect of a failure in terms of (poor) quality perceived by the user, hence the impact in terms of *user satisfaction*. For instance: *number of users affected by the failure*, which can be distinguished by classes with different weights (e.g., number of subscribers vs number of guests, with a greater weight assigned to the former) – this entails assessing the *operational profile* of the system, namely what parts of the systems the users (of a specific class) exercise more. Services of the system that may are of more value (hence, that used more and by higher-weighed users) should receive more attention during testing.

- Functionality-based: given the same users affected, some functions could lead to more damages for instance in terms of cost, e.g., because the deal with a critical task (e.g., money transactions). Therefore weights are assigned to functionalities to prioritize their test. This also requires a definition of the operational profile, and

the these metrics cab ne used in conjunction with the previous ones: weights could be assigned to any *user class/functionality* pair. It is important to stress that the operational profile is captured, in a uDevOps context, in a continuous manner through monitoring. In the project, we have focused on monitoring tools for uDevOps-based systems and we have exploited monitoring (and the operational profile) in the definition of testing techniques. Section 3 and 4 elaborate more on this

- Damage- and cost-based, external: the user should define metrics or give weights in a scale to assess the damage that every failure mode can cause, e.g., to the environment, to the users to other systems. This includes both physical damages and damages that can be quantified by a cost (e.g., loss of data). These are particularly important for safety- or mission/business-critical systems.

- Damage- and cost-based, internal: metrics classifying the damage/cost brought to the company developing the system; for instance, excessive consumption of energy, loss of internal data.

- Time-based: metrics reporting about the unmet deadlines (and the consequent cost). These are heavily affected by the internal practices put in place to improve the quality of the system (and reduce the quality-related risk): a very expensive testing technique could cause delays in the process, but a poor technique could expose to technical

quality-related risks. Therefore, cost-effectvie techniques are needed to mitigate this risk. This is discussed in the upcoming Sections 3 and 4.

The exact set of metrics will depend on the business model of the applications under development. It is important to remark that these metrics are not mutually exclusive, but can be considered together. In the rest of this Deliverable, our focus is primarily on the first two classes (user- and functionality-based), as more appropriate for the system we are using.

3 RISK ASSESSMENT

3.1 OVERVIEW

This Section reports the techniques we implemented that support the risk assessment phase. As discussed in the previous Section, the *effect* of a failure is related to the specific domain, therefore we hereafter focused on the estimation of failure occurrence probability (PFD) or related measures; we also borrow the concept of operational profiles to fine-tune the estimate toward the *operational* PFD. In this way, the technique can be easily applied by just changing the operational profile, and adapted to different classes of users of functionalities (see previous Section) hence giving an estimate considering also *who* is potentially affected by the failure. The resulting estimate needs to be combined with a scale of the type of effect (see Table 2.1), strictly depending on the specific application.

We hereafter distinguish between:

- Sateless testing-based estimate, wherein the demands are treated as

independent from each other.

- Stateful testing-based estimate, where the system architecture is modelled and the output depends on the sequence of demands.

The techniques we developed refer to different types of systems that are or can be developed according to a uDevOps paradigm. Besides the conventional systems, such as web apps or well-known benchmarks like TrainTicket Zhou et al. (2021), we also have worked on IoT systems (which are being frequently developed with microservices) and with AI systems using DNNs for specific tasks (these are gaining spread in uDevOps contexts like cloud-edge systems, with AI functionalities at the edge level implementing Tiny ML task). Therefore, all the works we describe hereafter refer to several type of systems.

Central to both methods is the concept of sampling-based testing, developed in the Deliverable 3.1 of uDevOps [1]. Here the main concepts are recalled: In sampling-based testing, the objective is to provide an estimate of the quality attribute of interest that is unbiased (hence, its expectation is the true value) and efficient (namely, with a minimal variance, that implies *high confidence*, or, conversely, with a small number of test cases given a minimum confidence in the estimate we want to have).

Estimation of the operational profile from monitoring data is already discussed in Deliverable 3.1, e.g., through the Dirichlet-based estimation

[1] www.udevops.eu

Pietrantuono et al. (2020). Therefore, hereafter it is assumed that an estiamte of the operational profile is available when required by the technique.

3.2 RELIABILITY-RELATED ASSESSMENT

3.2.1 Assessment in AI-based systems

The first technique we describe for PFD assessment is with reference to AI-based systems (e.g. Deep Neural Networks). In this context, the operational reliability corresponds to the operational *accuracy*, referred either to a classification (e.g., number of correctly classified examples) or a regression task (e.g., average difference between predicted and actual value). The development context is the transposition of DevOps to ML-based systems: MLOps.

Deep Neural Networks (DNN) are nowadays largely adopted in many application domains thanks to their human-like, or even superhuman, performance in specific tasks. However, due to unpredictable/unconsidered operating conditions, unexpected failures show up on field, making the performance of a DNN in operation very different from the one estimated prior to release. In large part, this is due to the *oracle problem*, which impacts the ability to automatically judge the output of the classification, thus hindering the accuracy of the assessment when unlabeled previously unseen inputs are submitted to the system.

3.2. RELIABILITY-RELATED ASSESSMENT

In the life cycle of DNN systems, the assessment of accuracy is typically addressed in two ways: offline, via sampling of operational inputs, or online, via pseudo-oracles. The former is considered more expensive due to the need for manual labeling of the sampled inputs. The latter is automatic but less accurate.

In the context of the uDevOps project, we contributed by defining a pseudo-oracle named *Image Classification Oracle Surrogate* (ICOS), a technique proposed by Guerriero et al. (2023a) to automatically evaluate the accuracy in the operation of Convolutional Neural Networks, namely DNNs for image classification. To establish whether the classification of an arbitrary image is correct or not, ICOS leverages three knowledge sources: operational input data, training data, and the ML algorithm. Knowledge is expressed through *likely invariants* - properties that should not be violated by correct classifications. ICOS infers and filters invariants to improve the correct detection of misclassifications, reducing the number of false positives. ICOS is evaluated experimentally on twelve CNNs – using the popular MNIST, CIFAR10, CIFAR100, and ImageNet datasets. Experimental results show that ICOS exhibits performance comparable to cross-referencing and self-checking in terms of accuracy, showing higher stability over a variety of CNNs and datasets with different complexity and size. ICOS likely invariants are shown to be effective in automatically detecting misclassifications by CNNs used in image classification tasks when the expected output is unknown; ICOS ultimately yields faithful assessments

of their accuracy in operation. Knowledge about input data can also be manually incorporated into ICOS, to increase robustness against unexpected phenomena in operation, like label shift.

Emerging iterative industrial-strength life cycle models for Machine Learning systems, like MLOps, offer the possibility to leverage inputs observed in operation not only to provide faithful estimates of a DNN accuracy but also to improve it through remodeling/retraining actions. In the context of uDevOps, we have proposed DAIC (DNN Assessment and Improvement Cycle) Guerriero et al. (2023b), an approach that combines "low-cost" online pseudo-oracles and "high-cost" offline sampling techniques to estimate and improve the operational accuracy of a DNN in the iterations of its life cycle. Preliminary results show the benefits of combining the two approaches and integrating them into the DNN life cycle.

Detailed results are in Guerriero et al. (2023a) and Guerriero et al. (2023b).

3.2.2 Stateful assessment

The techniques we describe hereafter use a model to represent sequences of requests and provide an estimate considering the user profile.

The idea of the first technique is to a novel methodology (called MIPaRT) and platform to automatically test microservice operations for performance and reliability in combination (Camilli et al. (2022)). The

proposed platform can be integrated into a DevOps cycle to support:

- continuous testing and monitoring by the automatic generation and execution of performance-reliability ex-vivo testing sessions;
- collection of monitoring data;
- computation of performance and reliability metrics;
- integrated visualization of the results.

Tests are generated based on a behavioural model of the users interacting with the system under test. The model is defined as a DTMC, where states and the transition between states can be extracted from historical data.

We apply this approach by operating the platform on an open-source benchmark. Results show that our integrated approach can provide additional insights into the performance and reliability behaviour of microservices as well as their mutual relationships.

This is described in more detail in Deliverable 3.1, and here is just recalled to have a complete picture.

3.3 PERFORMANCE- AND ENERGY-RELATED ASSESSMENT

3.3.1 Performance

The technique described in the previous Section presents MIPaRT working for both reliability and performance. The same strategy is therefore used to get an assessment of performance-related risk in a stateful manner and considering the operational profile.

Another technique that we have implemented is for the assessment of *performance degradation* over a long execution period – a phenomenon called *runtime software aging* Cotroneo et al. (2014). 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.

Efficient and effective object detection is a key problem in Computer Vision. Numerous object detection algorithms have been developed, whose aim is to achieve two conflicting goals, namely accuracy and efficiency, while being executed in real-time with high robustness. Many of these algorithms must run for an extended period of time, i.e., in video surveillance or in self-driving cars – a working condition that make them subject to the risk of software aging.

3.3. PERFORMANCE- AND ENERGY-RELATED ASSESSMENT

In the uDevOps context, Pietrantuono et al. (2022) analyze the phenomenon of software aging in state-of-the-art object detection algorithms. More specifically, we performed long-running experiments to analyze how software aging manifests under different algorithms, libraries/implementations, and datasets. We collected both resource consumption indicators (e.g., free/buffer/cache memory and resident memory size) and performance-related indicators (e.g., frames per second) and statistically analyze the presence or absence of aging phenomena, quantify their extent and assess the difference between various settings (i.e., algorithms, libraries, datasets).

Results highlighted that every aging indicator used in our experiments shows resource consumption or performance degradation, regardless of the algorithm, implementation, or dataset. It is also shown that four out of six aging indicators, more than 50% of the experiments, manifest aging effects with a peak of over 95%. Additionally, the results revealed that the algorithm and dataset factors seem to be less important than the library one (i.e., the specific implementation).

Additional details can be found in the paper by Pietrantuono et al. (2022).

3.3.2 Energy (and performance) assessment

This Section describes the studies we have conducted for energy consumption measurement along with performance assessment. Energy consumption monitoring is paramount as it allows for controlling the cost and detecting possible spikes that could be caused by unintentional faults or deliberate attacks.

The studies are on, again, computer vision tasks at the edge, mobile and web apps, and binaries in IoT systems, respectively.

On energy consumption, we also assessed the consumption entailed by using monitoring tools themselves, which are cornerstones of uDevOps development.

Computer vision. The rise of use cases of AI catered towards the Edge, where devices have limited computation power and storage capabilities, motivates the need for better understating of how AI performs and consumes energy.

With Hampau et al. (2022) we aim to empirically assess the impact of three different AI containerization strategies on the energy consumption, execution time, CPU, and memory usage for computer-vision tasks on the Edge.

We conduct an experiment with the used containerization strategy as the

main factor, with three treatments: ONNX Runtime, WebAssembly, and Docker. The subjects of the experiment are four widely-used computer-vision algorithms. We then orchestrate a series of runs where we deploy the four subjects on different generations of Raspberry Pi devices, with different hardware capabilities. A total of 120 runs (per device) are recorded to gather data on energy, execution time, CPU, and memory.

We found a statistically significant difference between the three containerization strategies on all dependent variables. Specifically, WebAssembly proves to be a valuable alternative for devices with reduced disk space and computation power.

For computer-vision tasks with limited disk space and RAM memory requirements, developers should prefer WebAssembly for deployment. The (non-dockerized) ONNX Runtime resulted to be the best choice in terms of energy consumption and execution time.

Web apps. Many Internet content platforms, such as Spotify and YouTube, provide their services via both native and Web apps. Even though those apps provide similar features to the end user, using their native version or Web counterpart might lead to different levels of energy consumption and performance.

In the context of the uDevOps project, with Horn et al. (2023), we aim to empirically assess the energy consumption and performance of native and

Web apps in the context of Internet content platforms on Android.

We select 10 Internet content platforms across 5 categories. Then, we measure them based on the energy consumption, network traffic volume, CPU load, memory load, and frame time of their native and Web versions; then, we statistically analyze the collected measures and report our results.

We confirm that native apps consume significantly less energy than their Web counterparts, with a large effect size. Web apps use more CPU and memory, with statistically significant differences and large effect sizes. Therefore, we conclude that native apps tend to require fewer hardware resources than their corresponding Web versions. The network traffic volume exhibits a statistically significant difference in favour of native apps, with a small effect size. Our results do not allow us to draw any conclusion in terms of frame time.

Based on our results, we advise users to access Internet content using native apps over Web apps, when possible. Also, the results of this study motivate further research on the optimization of the usage of runtime resources of mobile Web apps and Android browsers.

IoT. WebAssembly (WASM) is a low-level bytecode format that is gaining traction among Internet of Things (IoT) devices. Because of IoT devices' resources limitations, using WASM is becoming a popular technique for virtualization on IoT devices. However, it is unclear if the promises of WASM

regarding its efficient use of energy and performance gains hold true.

With this study, we aim to determine how different source programming languages and runtime environments affect the energy consumption and performance of WASM binaries.

Wagner et al. (2023) perform a controlled experiment where we compile three benchmarking algorithms from four different programming languages (i.e., C, Rust, Go, and JavaScript) to WASM and run them using two different WASM runtimes on a Raspberry Pi 3B.

The source programming language significantly influences the performance and energy consumption of WASM binaries. We did not find evidence of the impact of the runtime environment. However, certain combinations of source programming language and runtime environment leads to a significant improvement of its energy consumption and performance.

IoT developers should choose the source programming language wisely to benefit from better performance and a reduction in energy consumption. Specifically, Javy-compiled JavaScript should be avoided, while C and Rust are better options. We found no conclusive results for the choice of the WASM runtime.

Monitoring tools With Dinga et al. (2023) our aim was to identify, synthesize, and empirically evaluate the energy and performance overhead of monitoring tools employed in the microservices and DevOps context.

We selected four representative monitoring tools in the microservices and DevOps context. These were evaluated via a controlled experiment on an open-source Docker-based microservice benchmark system.

The results highlight: *i)* the specific frequency and workload conditions under which energy consumption and performance metrics are impacted by the tools; *ii)* the differences between the tools; *iii)* the relation between energy and performance overhead.

We obtained significant results in terms of energy and performance (CPU usage, CPU load, RAM usage, network traffic, and execution time), under specific frequency and workload conditions. Not all the tools impact energy efficiency and performance in the same way, but we observed a high energy consumption and a high CPU, RAM, and execution time for the same tools. The correlation analysis confirms the association for CPU and execution time, but not for memory, hence the latter is likely to have a smaller impact on energy. For a more granular analysis, to be able to detect energy hotspots in monitoring tools, we plan to deploy a software power meter in a future iteration, such as SmartWatts[2], that measures energy at container level.

[2]https://powerapi-ng.github.io/smartwatts.html

3.4 SECURITY-RELATED ASSESSMENT

3.4.1 UX evaluation

Security assessment is a key activity to expose vulnerabilities of a system. In the context of the uDevOps project, the partners are acquiring knowledge from the activities of one partner, Silensec, which develops a microservice-based cyber-range platform used for training on security topics. The platform allows the creation of attack-defense scenarios and the trainee are scored according to their actions. We have evaluated the user experience (UX) as quality attribute, in order to reduce the business risk arising from competition with similar services that could erode the market share. Usability is also a quality attribute that can impact the measure of ”risk”, as it leads to unsatisfactory UX, indirectly impacting the competitiveness of the solution. The evaluation we have conducted is reported in De-marcos et al. (2022); the results point to critical elements of three main functionalities: library of scenarios, scenario information and entering scenario. Since there are several currently available solutions that offer similar services, the user experience of the microservice web app may play a critical role in determining which application will get a dominant role in the market. The details of the evaluation are reported in De-marcos et al. (2022).

3.5 COMBINING THE ESTIMATES

This Section briefly describes how one could combine estimates obtained with different methods.

The techniques presented above adopt different techniques able to provide an estimate of probability of failure (PFD). In almost all the cases, this is derived by a frequentist strategy, where the number of observed failures is used by an estimator depending on how the sample was selected. In one case, we have also explored the use of a Bayesian strategy, with the advantage of having a simple way to update the estimates as more data becomes available.

The choice of the method to adopt depend on many aspects, including the information available (e.g., auxiliary variable for sampling), the uncertainty on the operational profile, how often the profile changes, the proneness of the system to fail (e.g., with few failures, a Bayesian approach may be better in avoiding overestimation), etc.

Regardless the estimators chosen, a simple yet effective policy to combine the estimates is by combining them. Every estimate is associated with a variance of the estimator, and this variance is directly related to the confidence we can put in the estimation (higher variance, less stable estimate, little confidence). Therefore, given a quality attribute (for instance Reliability $R = 1 - PFD$), a simple way to combine two estimates R_1 and R_2 obtained with two distinct methods is to take the weighted average

$\hat{R}_1 \cdot w_1 + \hat{R}_2 \cdot w_2$, where the weights w_1 and w_2 are the variances of the estimate $w_1 = V(\hat{R}_1), w_2 = V(\hat{R}_2)$.

The result can be combined with the failure impact (Table 2.1) to give the risk estimate.

4 RISK MITIGATION

4.1 OVERVIEW

Risk mitigation refers to all those strategies we have implemented to reduce the risk associated with reliability, performance/energy, security failures. The techniques we developed include debug testing (aimed at fault detection rather than or besides reliability assessment), anomaly detection and root cause analysis to support fault removal, as well as good design practices at different development stages that can reduce the risk of introducing failures (e.g., design for sustainability to improve, among others, the energy footprint of the application; inconsistency detection at architectural level; improving code quality via dead code detection and removal).

These are discussed with reference to one or more quality attribute (reliability, performance, energy, security) since, their application can easily be customized to work with all the attributes.

4.2 MITIGATION VIA TESTING

4.2.1 Optimal effort allocation for risk-aware test planning

We can exploit DTMC to model the reliability of single components and use an optimization model to allocate testing resources to each service, as quantitative support to *test planning*. In this way, components/services deemed more risky would receive proportionally more effort, depending on the prediction about their failing behaviour. The overall aim is to reduce the effort to attain a minimum desired PFD on the overall system. Such a strategy can be found in Pietrantuono et al. (2010), and can be easily extended to the case of service-based systems where each component is a microservice. Since the reliability (that is $1 - PFD$) accounts for the usage profile in this model, the formulation is more suited to assess the risk entailed by a service failure.

Architectural model

The architecture of a software system can be described by an absorbing DTMC, to represent terminating applications (as opposed to irreducible DTMCs, which are more suitable to represent continuously running applications). A DTMC is characterized by its states and transition probabilities among the states. The one-step transition probability matrix P = $[p_{i,j}]$ is a stochastic matrix so that all the elements in a row of P

add up to 1 and each of the $p_{i,j}$ values lies in the range [0, 1]. The one-step transition probability matrix with *n* states and *m* absorbing states can be partitioned as:

$$P = \begin{pmatrix} Q & C \\ 0 & I \end{pmatrix} \tag{4.1}$$

where Q is an *(n-m)* by *(n-m)* sub-stochastic matrix (with at least one row sum < 1), I is an *m* by *m* identity matrix, 0 is an *m* by *(n -m)* matrix of zeros and C an *(n-m)* by *m* matrix. If we denote with P^k the k-step transition probability matrix (where the entry *(i,j)* of the submatrix Q^k is the probability of arriving in the state j from the state i after k steps), it can be shown Huang and Lyu (2005); Trivedi (2001) that the so-called fundamental matrix M is obtained as

$$M = (I - Q)^{-1} = I + Q + Q^2 + \cdots + Q^k = \sum_{k=0}^{\infty} Q^k \tag{4.2}$$

Denoting with $X_{i,j}$, the number of visits from the state *i* to the state *j* before absorption, the expected number of visits from *i* to *j*, i.e., $v_{i,j} = E[X_{i,j}]$, is the $m_{i,j}$ entry of the fundamental matrix. Thus, the expected number of visits starting from the initial state to the state *j* is:

$$v_{1,j} = m_{1,j} \tag{4.3}$$

These values are called expected visit counts; denoted with $V_j = v_{1,j}$,

used to describe the usage of each component in the application control flow. To compute the variance of visit counts, denote with $\sigma^2_{i,j}$ the variance of the number of visits to *j* starting from *i*. Let M_D be the diagonal matrix with:

$$M_D = \begin{cases} m_{i,j} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

and define M_2 = [$m^2_{i,j}$], we have

$$\sigma^2 = M(2M_D - I) - M_2 \tag{4.5}$$

Hence

$$Var[X_{i,j}] = \sigma^2_{i,j} \tag{4.6}$$

To represent a service-based application as a DTMC, consider the dependency graph. Assuming that an application has *n* services, with the entry edge service indexed by 1 and the final service in an invocation chain denoted as *n*, DTMC states represent the services and the transition from state *i* to state *j* represents the invocation from service *i* to service *j*. Following the procedure explained above, we can compute the expected number of visits to each service and its variance.

The DTMC representation, along with the concept of visit counts, can be used to express the system reliability as a function of the services reliability. In particular, denoting with R_i the reliability of service *i*, the system reliability is the product of individual reliability values raised to the power

of the number of visits to each service, denoted by $X_{1,i}$ (i.e., each service reliability is multiplied by itself as many times as the number of times it is visited starting from the first one); i.e., $R \approx \prod_i^n R_i^{X_{1,i}}$. Since the number of visits to a service is a random variable (except for the last service), the so-computed system reliability is also a random variable. Thus, denoting with E[R] the total expected reliability of the system, we have:

$$E[R] \approx \prod_i^n E[R_i^{X_{1,i}}] \approx (\prod_i^{n-1} R_i^{E[X_{1,i}]}) R_n \tag{4.7}$$

where E[$X_{1,i}$] is the expected number of visits to service *i* (and $X_{1,i}$ is always 1 for the final service *n*). The adopted model is known to belong to the class of hierarchical approaches; this kind of models, though approximate, lead to quicker and more tractable solutions than composite models. To also take into account the second-order architectural effects and obtain a more accurate result, we can expand the above equation according to the Taylor series:

$$E[R] = [\prod_i^{n-1} (R_i^{m_{1,i}} + \frac{1}{2}(R_i^{m_{1,i}})(\log R_i)^2 \sigma_{1,i}^2)] R_n \tag{4.8}$$

Where $m_{1,i}$ = E[$X_{1,i}$] and $\sigma_{1,i}^2$ = Var[$X_{1,i}$] (since $X_{1,n}$ is always 1, $m_{1,n}$ = 1 and $\sigma_{1,n}^2$ = 0).

The second-order architectural effects are captured by the variance of the number of visits. The only source of approximation is the Taylor series cut-off. Note that the described model, as most of the architecture-based models, assumes independent failures among services.

Optimization Model

The above DTMC model can be exploited in an optimization model to decide how much effort to devote to test each service, given a reliability goal to attain. Since the final reliability depends on the visit counts (hence on the usage profile), a more reliable but more used service needs more testing than a less reliable but rarely used service. This model accounts for this, hence it is more prone for risk assessment.

The goal of the optimization model is to find the best combination of testing efforts to be devoted to each service so that they achieve a reliability level that can assure an overall reliability E[R] $\geq R_{MIN}$. For instance, if we assume that the reliability of each service grows with the testing time devoted to it, we can describe this relation by a software reliability growth model (SRGM). This relation can be represented as $T = f(\lambda)$, where T is the Testing Time and λ is the failure intensity.
The general optimization model will then look like:
determine the optimal values of $T_1, T_2, \ldots, T_n$ so as to

$$\text{Minimize} \qquad T = \sum_{i=1}^{n} T_i = \sum_{i=1}^{n} f_i(\lambda_i) \tag{12.a}$$

subject to:

$$E[R] = [\prod_{i}^{n-1} R_i] R_n \geq R_{MIN} \tag{12.b}$$

with i = 1 . . . *n-1*, *n* components and T indicating the total testing time

for the application to get a total reliability E[R] $\geq R_{MIN}$. Here λ_i variables are the decision variables (which determine the T_i variables and that are of course present in the constraint factors, via $R_i = \exp[-\int_0^{t_i} \lambda_i(\theta)d\theta]$).
If we assume for simplicity a constant failure intensity after release, the reliability of the service *i* at the end of the testing will be:

$$R_i = \exp[-\int_0^{t_i} \lambda_i(\theta)d\theta] = \exp[-\lambda_i t_i] \tag{4.10}$$

with t_i is the expected execution time per visit to service i; **this equation relates the failure intensity of service i to its reliability**. Each service can be characterized by a different SRGM (among the plethora of proposed ones). The solution of the optimization model will give the testing effort to devote to each service to minimize the risk of failures given the observed usage profile (e.g., in past releases).

4.2.2 Test generation and post-testing Failure propagation analysis

Besides test planning, we have worked on test generation for microservices. MSA automated testing is possible thanks to well-defined service interfaces specified in open formats like OpenAPI/Swagger. To support automated MSA functional and non-functional testing, with Giamattei et al. (2024) we have defined a framework that: (i) generates test cases with valid and invalid inputs, and executes and monitors tests; (ii) provides coverage and failure information not only on edge, but also on internal microservices; (iii) has the novel feature of identifying causal relations in observed chains

of microservices failures (*propagated* and *masked* failures). We abstracted the testing process of MSA, present the MacroHive framework and its causal inference engine, compare it experimentally to state-of-the-art tools, and discuss its benefits in the MSA testing process. MacroHive exhibits performance comparable to advanced existing tools in terms of edge-level coverage. However, MacroHive has a better failure rate and provides the unique advantages of giving insights about internal coverage and failures, and of inferring causality in failure chains, evidencing microservices to be improved to increase the whole MSA reliability.

Additional details are available in Giamattei et al. (2024).

4.2.3 Tests generation for fault detection

Besides the teset case generation techniques we have described in Deliverable 3.1, we are currently exploring how AI is supporting testing (other than testing of DNN), as also anticipated in D3.1.

With software systems becoming increasingly pervasive and autonomous, our ability to test for their quality is severely challenged. Many systems are called to operate in uncertain and highly-changing environment, not rarely required to make intelligent decisions by themselves. This easily results in an intractable state space to explore at testing time. The state-of-the-art techniques try to keep the pace, e.g., by augmenting the tester's intuition with some form of (explicit or implicit) learning from

observations to search this space efficiently. For instance, they exploit historical data to drive the search (e.g., ML-driven testing) or the tests execution data itself (e.g., adaptive or search- based testing). Despite the indubitable advances, the need for smartening the search in such a huge space keeps to be pressing. In the context of uDevOps, Giamattei et al. (2023) introduce Reasoning-Based Software Testing (RBST), a new way of thinking at the testing problem as a causal reasoning task. Compared to mere intuition-based or state-of-the-art learning-based strategies, we claim that causal reasoning more naturally emulates the process that a human would do to "smartly" search the space. RBST aims to mimic and amplify, with the power of computation, this ability. The conceptual leap can pave the ground to a new trend of techniques, which can be variously instantiated from the proposed framework, by exploiting the numerous tools for causal discovery and inference. We provide a preliminary evaluation of a basic instance of RBST, for testing an Autonomous Driving System against adaptive testing and an ML-driven search-based technique. Results show the benefit of exploiting cause-effect relations to derive safety-violating tests.

About performance testing of MSA, we are working on CAR-PT (CAusal-Reasoning-driven Performance Testing), a model-based technique for workload generation designed for the performance testing of MSA. CAR-PT leverages causal reasoning to effectively explore the space of operational conditions, with the goal of identifying those that lead to performance issues.

4.3 OPERATIONAL TIME TECHNIQUES

4.3.1 Anomaly detection

Risk mitigation is also achieved during operation, where errors can be detected and treated before they can cause manage. An important part of this task is *anomaly detection*. In the context of the project, with Cinque et al. (2022) we describe a study on log mining in the domain of microservices technologies. We focus on the detection of anomalies from logs, i.e., events requiring deeper inspection by analysts. Log mining is challenging in microservices systems due to the high number of heterogeneous logs. We present Micro2vec, a novel approach to mine numeric representations of computer logs without making assumptions on the format of underlying data and requiring no application knowledge; representations computed by Micro2vec are suited for anomaly detection. To cope with the lack of publicly-available datasets of labeled logs from production systems, we validate our approach by means of a mixture of direct measurements from logs, one-class classification experiments and generation of log variants. The study has been conducted in the context of a Clearwater IP Multimedia Subsystem setup consisting of microservices deployed in Docker containers, and on a real-world critical information system from the Air Traffic Control domain, which implements a communication model typically used with microservices. Results indicate that analyzing metrics inferred by different logs facilitates the detection of anomalies, which are characterized by

signature involving multiple logs; it also allows inferring explicable detection rules that are hard to be caught by human experts. In addition, log variants obtained from normal logs can support detecting real anomalies, and they improve over one-class classifiers. Further details are reported in Cinque et al. (2022).

4.3.2 Design-time techniques

4.3.3 Sustainability-aware design

Risk incurred by bad design can be mitigated via good design practices. This Section briefly describes the work done in the project on sustainability modelling, hence with the focus on the need to satisfy sustainability requirements.

Over the years, various thinking frameworks have been developed to address sustainability as a quality property of software-intensive systems. Notwithstanding, which quality concerns should be selected in practice that have a significant impact on sustainability remains a challenge. In this experience report, with Funke et al. (2023) we proposed the notion of variability features, i.e., specific software features which are implemented in a number of possible alternative variants, each with a potentially different impact on sustainability. We extended sustainability decision maps to incorporate these variability features into an already existing thinking framework. Our findings were derived from a qualitative case study

and evaluated in an industrial context. Data was collected by analysing a real-world application (Zahori provided by Panel, a project's partner) and conducting working sessions together with expert interviews. The variability features allowed us to identify and evaluate alternative usage scenarios of one real-world software-intensive system, enabling data-driven sustainability choices and suggestions for professional practices. By providing concrete measurements, we can support software architects at design time, and decision makers towards achieving sustainability goals.

This work is an example of quality-related activities carried out in the design phase on a microservice application for decision support to designers, aimed at preventing future issues affecting sustainability.

4.3.4 inconsistencies detection in software architecture

Documenting software architecture is important for a system's success, and allows reducing the probability of introducing errors in the system. Software architecture documentation (SAD) makes information about the system available and eases comprehensibility. There are different forms of SADs like natural language texts and formal models with different benefits and different purposes. However, there can be inconsistent information in different SADs for the same system. Inconsistent documentation then can cause flaws in development and maintenance. To tackle this, with Keim et al. (2023) we present an approach for inconsistency detection in natural

language SAD and formal architecture models. We make use of traceability link recovery (TLR) and extend an existing approach. We utilize the results from TLR to detect unmentioned (i.e., model elements without natural language documentation) and missing model elements (i.e., described but not modeled elements). In our evaluation, we measure how the adaptations on TLR affected its performance. Moreover, we evaluate the inconsistency detection. We use a benchmark with multiple open source projects and compare the results with existing and baseline approaches. For TLR, we achieve an excellent F1-score of 0.81, significantly outperforming the other approaches by at least 0.24. Our approach also achieves excellent results (accuracy: 0.93) for detecting unmentioned model elements and good results for detecting missing model elements (accuracy: 0.75). These results also significantly outperform competing baselines. Although we see room for improvements, the results show that detecting inconsistencies using TLR is promising.

4.4 CODE IMPROVEMENT

4.4.1 Code quality improvement via dead code elimination

A further phase where risk of failure can be reduced is coding. This Section reports about the work conducted to improve the code quality, via dead code elimination.

Web apps are built by using a combination of HTML, CSS, and

JavaScript. While building modern web apps, it is common practice to make use of third-party libraries and frameworks, as to improve developers' productivity and code quality. Alongside these benefits, the adoption of such libraries results in the introduction of JavaScript dead code, i.e., code implementing unused functionalities. The costs for downloading and parsing dead code can negatively contribute to the loading time and resource usage of web apps.

In the context of the project, Malavolta et al. (2023) presented a study with the following objectives:

- First, they present Lacuna, an approach for automatically detecting and eliminating JavaScript dead code from web apps. The proposed approach supports both static and dynamic analyses, it is extensible and can be applied to any JavaScript code base, without imposing constraints on the coding style or on the use of specific JavaScript constructs.
- Second, by leveraging Lacuna they conduct an experiment to empirically evaluate the run-time overhead of JavaScript dead code in terms of energy consumption, performance, network usage, and resource usage in the context of mobile web apps.

Lacuna is applied four times on 30 mobile web apps independently developed by third-party developers, each time eliminating dead code

according to a different optimization level provided by Lacuna. Afterward, each different version of the web app is executed on an Android device, while collecting measures to assess the potential run-time overhead caused by dead code. Experimental results, among others, highlight that the removal of JavaScript dead code has a positive impact on the loading time of mobile web apps, while significantly reducing the number of bytes transferred over the network.

Additional details are available in Malavolta et al. (2023).

5 CONCLUSION

This document presented the work done for the definition of risk assessment and mitigation strategies on atop of reliability, performance/energy, and security assessment and improvement.

Specifically, we have first described the main challenges to measure the risk associated with the failure of a system in meeting quality-of-service (QoS) attributes for microservices, namely reliability, performance (and additionally energy consumption), security.

Then, we presented the work done in the project to implement techniques for the assessment of risks, realted to reliability, performance and energy consumption, and security.

Finally, we have presented the work done to implement techniques for risk mitigation. Several means have been exploited to reduce the risk of failures, ranging from test planning, test generation, anomaly detection, post-testing failure analysis, design-time quality improvement practices, code improvement.

The techniques developed in WP4 and reported in this document will be the input to the final WP of the project, WP5.

REFERENCES

"Istqb: Standard glossary of terms used in software testing. version 2.2. tech. rep., istqb (2012).

"Standards australia/new zealand: Risk management as/nzs 4360:2004 (2004).

Camilli, M., Guerriero, A., Janes, A., Russo, B., and Russo, S. (2022). "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>.

Cinque, M., Della Corte, R., and Pecchia, A. (2022). "Micro2vec: Anomaly detection in microservices systems by mining numeric representations of computer logs." *Journal of Network and Computer Applications*, 208, 103515.

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).

De-marcos, L., Gutiérrez-Martínez, J.-M., Domínguez-Díaz, A., Caro-Álvaro, S., and Rodríguez, D. (2022). "Evaluating the ux of a microservice web app." *Central European Conference on Information and Intelligent Systems*.

Dinga, M., Malavolta, I., Giamattei, L., Guerriero, A., and Pietrantuono, R. (2023). "An empirical evaluation of the energy and performance overhead of monitoring tools on docker-based systems." *Service-Oriented Computing*, F. Monti, S. Rinderle-Ma, A. Ruiz Cortés, Z. Zheng, and M. Mecella, eds., Cham, Springer Nature Switzerland, 181–196.

Felderer, M. and Schieferdecker, I. (2019). "A taxonomy of risk-based testing.

Funke, M., Lago, P., and Verdecchia, R. (2023). "Variability features: Extending sustainability decision maps via an industrial case study." *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*, 1–7.

Giamattei, L., Guerriero, A., Pietrantuono, R., and Russo, S. (2024). "Automated functional and robustness testing of microservice architectures." *Journal of Systems and Software*, 207, 111857.

Giamattei, L., Pietrantuono, R., and Russo, S. (2023). "Reasoning-based software testing." *2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, Los Alamitos, CA, USA, IEEE Computer Society, 66–71, <https://doi.ieeecomputersociety.org/10.1109/ICSE-NIER58687.2023.00018> (may).

Guerriero, A., Lyu, M. R., Pietrantuono, R., and Russo, S. (2023a). "Assessing operational accuracy of cnn-based image classifiers using an oracle surrogate." *Intelligent Systems with Applications*, 17, 200172.

Guerriero, A., Pietrantuono, R., and Russo, S. (2023b). "Iterative assessment and improvement of dnn operational accuracy." *2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, Los Alamitos, CA, USA, IEEE Computer Society, 43–48, <https://doi.ieeecomputersociety.org/10.1109/ICSE-NIER58687.2023.00014> (may).

REFERENCES

Hampau, R., Kaptein, M., Emden, R., Rost, T., and Malavolta, I. (2022). "An empirical study on the performance and energy consumption of ai containerization strategies for computer-vision tasks on the edge." 50–59 (06).

Horn, R., Lahnaoui, A., Reinoso, E., Peng, S., Isakov, V., Islam, T., and Malavolta, I. (2023). "Native vs web apps: Comparing the energy consumption and performance of android apps and their web counterparts." *2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 44–54.

Huang, C.-Y. and Lyu, M. (2005). "Optimal testing resource allocation, and sensitivity analysis in software development." *IEEE Trans. on Reliability*, 54(4), 592–603.

Keim, J., Corallo, S., Fuchß, D., and Koziolek, A. (2023). "Detecting inconsistencies in software architecture documentation using traceability link recovery." *2023 IEEE 20th International Conference on Software Architecture (ICSA)*, 141–152.

Malavolta, I., Nirghin, K., Scoccia, G. L., Romano, S., Lombardi, S., Scanniello, G., and Lago, P. (2023). "Javascript dead code identification, elimination, and empirical assessment." *IEEE Transactions on Software Engineering*, 49(7), 3692–3714.

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.

Pietrantuono, R., Popov, P., and Russo, S. (2020). "Reliability assessment of service-based software under operational profile uncertainty." *Reliability Engineering & System Safety*, 204, 107193.

Pietrantuono, R., Russo, S., and Trivedi, K. (2010). "Software Reliability and Testing Time Allocation: An Architecture-Based Approach." *IEEE Trans. on Software Engineering*, 36(3), 323–337.

Powell, D. (1992). "Failure mode assumptions and assumption coverage." *[1992] Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing*, 386–395.

Trivedi, K. (2001). *Probability and statistics with reliability, queuing and computer science applications (2nd ed.)*. John Wiley and Sons Ltd., Chichester, UK.

Wagner, L., Mayer, M., Marino, A., Soldani Nezhad, A., Zwaan, H., and Malavolta, I. (2023). "On the energy consumption and performance of webassembly binaries across programming languages and runtimes in iot." *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, EASE '23, New York, NY, USA, Association for Computing Machinery, 72–82, <https://doi.org/10.1145/3593434.3593454>.

Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Li, W., and Ding, D. (2021). "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study." *IEEE Transactions on Software Engineering*, 47(2), 243–260.