Adaptation Strategy: Implementation and Evaluation

Ahmet Karapinar
University of Amsterdam
ahmetkarapinarr00@gmail.com

***Abstract*–The work focuses on a self-adaptive microservices architecture to enhance service availability and response time using novel strategies for failure detection and recovery. It emphasizes a utility-based proactive adaptation mechanism implemented in RAMSES, which integrates the Monitor, Analyze, Plan, and Execute (MAPE) cycle over shared knowledge. The primary goal is to compare the novel strategy against a baseline reactive strategy, demonstrating its advantages in handling failures more effectively.**

I. Introduction

Modern software systems have grown increasingly complex and dynamic, requiring robust strategies to maintain stability, resilience, and continuous functionality [1]. Traditional approaches, such as manual interventions, scheduled maintenance, and predefined error-handling procedures, have proven effective in predictable and controlled environments [1]. However, with the evolution of software architectures to include distributed components, microservices, and cloud-native systems, these traditional methods face significant limitations [1]. Human-driven processes are inherently inefficient, prone to errors, and resource-intensive, often leading to prolonged downtime and increased operational costs. These challenges highlight the need for intelligent, autonomous solutions capable of adapting seamlessly to dynamic and unpredictable conditions [1]. Dashofy, van der Hoek, and Taylor (2002) underscore the significance of architectural models in facilitating dynamic system adaptation to enhance fault tolerance in distributed systems [19].

Self-healing systems have emerged as a transformative solution to these challenges, offering the ability to autonomously detect, diagnose, and recover from faults in real-time [2]. Ghosh et al. (2007) define self-healing systems as a paradigm that integrates proactive fault detection and recovery, leveraging intelligent mechanisms to maintain system continuity [22].Unlike traditional systems, which rely on reactive troubleshooting and static configurations, self-healing systems engage in continuous monitoring of their operational environment, dynamically adapting to changing conditions and executing proactive recovery strategies [2]. This proactive capability enables modern software systems to maintain dependability and efficiency even in the face of unforeseen challenges, such as hardware failures, resource bottlenecks, or sudden demand surges [2]. Georgiadis, Magee, and Kramer (2002) highlight the importance of self-organizing software architectures in distributed systems, facilitating decentralized coordination and dynamic adaptation to changes in the environment [18].

Among self-healing techniques, utility-driven approaches stand out for their ability to optimize system performance and resource efficiency [5]. These approaches leverage utility functions to evaluate the costs and benefits of potential adaptations, guiding decisions that maximize overall performance while minimizing disruption. Such proactive strategies advance beyond conventional reactive methods by incorporating mechanisms that anticipate failures and implement corrective actions before issues escalate [3]. For instance, utility-driven strategies can continuously assess service health, balancing resource utilization and availability to ensure consistent service quality. Georgiadis et al. describe how self-organizing systems can proactively adjust their structure to maintain functionality, aligning with these utility-driven principles [24].

A. Hypothesis

Our work investigates the efficacy of a novel adaptation strategy designed to proactively address failures within a microservices architecture. The primary hypothesis driving this work is: *If a proactive utility-based adaptation strategy coupled with dynamic load balancing is employed, then it will outperform a reactive baseline strategy in maintaining higher availability and lower response times during failure scenarios.* This hypothesis is grounded in the premise that early detection and intervention, combined with intelligent traffic redistribution, can prevent failures from causing significant disruptions and ensure efficient recovery. Garlan, Cheng, and Schmerl (2002) highlight that architecture-based self-repair mechanisms play a vital role in enhancing system dependability and ensuring uninterrupted operations [23].

The adaptation strategy, integrated into the RAMSES framework, leverages utility-based health scores calculated from key system metrics such as availability, response time, CPU usage, and remaining disk space. By preemptively identifying instances at risk of failure, the strategy enables the system to take actions such as scaling services and dynamically adjusting load balancer weights. The dynamic load balancing mechanism ensures that traffic is redirected from unhealthy instances to newly added, healthier instances, reducing the burden on degraded components and improving fault recovery.

To validate this hypothesis, we conducted a comparative evaluation between the novel proactive strategy and a reactive baseline strategy under identical failure conditions. The experimental setup simulates failure events, progressively degrading an instance's performance and ultimately forcing it to fail. Through these experiments, we aim to demonstrate that the novel strategy not only maintains better availability and response times during normal operation but also reduces the damage incurred during failure events and accelerates recovery afterward. The integration of dynamic load balancing further enhances the system's resilience by enabling adaptive traffic management during fault recovery phases.

B. Comparioson with Existing Solution

The primary motivation for our chosen strategy stems from the inherent limitations of the existing reactive adaptation mechanism within the RAMSES framework. The existing strategy relies solely on detecting failures after they occur, triggering reactive measures to address service disruptions. While this approach can resolve failures eventually, it often leads to significant degradation in availability and response time during the failure period. The delay between failure occurrence and adaptation can result in a substantial impact on service reliability and user experience.

Our novel strategy addresses this shortcoming by integrating a proactive adaptation mechanism that leverages a utility-based health score to detect potential failures before they fully manifest. By continuously monitoring critical metrics such as availability, response time, CPU usage, and disk space, the system identifies unhealthy instances early and takes preemptive actions. This proactive approach not only prevents instances from failing outright but also minimizes the disruption caused by performance degradation.

In summary, our strategy solves the critical problem of delayed and inefficient recovery in reactive adaptation mechanisms by proactively detecting potential failures and incorporating dynamic load balancing. This results in a more resilient system that maintains higher availability and lower response times during failure scenarios and recovery phases, addressing the key limitations of the existing solution.

C. Overview and Terminology

This paper evaluates and compares a novel proactive adaptation strategy with a traditional reactive baseline strategy within the RAMSES framework. The primary objective is to demonstrate the advantages of a proactive utility-driven approach for self-healing microservices. The novel strategy is built on real-time monitoring and analysis using a composite health utility score, incorporating metrics such as availability, response time, CPU usage, and disk space. This aligns with the deployment strategies outlined by Sztajnberg and Loques, where adaptation decisions are guided by application descriptions and system metrics [21]. The results show significant improvements in availability and response time during failure events compared to the baseline strategy. The proactive strategy not only detects failures beforehand but also dynamically adjusts load balancing for efficient fault recovery, minimizing service disruption.

Throughout the report, terminology such as "health utility score," "adaptive load balancing," and "proactive self-healing" are used to describe core concepts. The MAPE-K feedback loop framework underpins the strategy, ensuring a structured approach to monitoring, analyzing, planning, and executing adaptations. This framework is integrated with RAMSES, which provides modularity and real-time monitoring capabilities for self-adaptive systems. The methodology includes the implementation of a fault detection mechanism and dynamic load balancing during shadowing phases, ensuring resilience in dynamic microservice environments. By emphasizing the comparative effectiveness of proactive and reactive strategies, this report illustrates the potential of intelligent self-adaptive solutions for modern distributed systems.

II. Related Work

In the context of self-adaptive systems, proactive self-healing strategies, particularly those that make use of utility functions for decision-making, have been subjected to extensive research. Gorlick et al. explore the use of architectural models to guide system reconfigurations and ensure consistent performance, even under adverse conditions [20]. It is the intention of these tactics to improve the dependability and resilience of the system by proactively anticipating future problems and allowing for autonomous adaptation. The purpose of this section is to examine relevant works in utility-based self-healing, with a particular focus on approaches that demonstrate conceptual parallels with our strategy. When calculating the healthUtilityScore for each instance, our technique takes into account variables such as availability, CPU use, and the proportion of disk space that is still available. Through the process of aggregating average measurements for services and proactively adding instances once a predetermined health threshold is exceeded, it improves the overall availability of the application.

In their 2017 paper, Ghahremani, Giese, and Vogel developed a hybrid strategy that combines rule-based and utility-driven self-healing principles. Their method involves the systematic calculation of utility values for adaptation rules, which allows them to avoid the process of optimization, which can be quite expensive. However, runtime goal evaluation has also been highlighted as an effective technique for improving adaptability, as it allows self-healing systems to respond dynamically to changing conditions without requiring exhaustive optimization [17].For the purpose of locating errors and initiating adaptation actions in a dynamic way, it makes use of pattern-based utility functions. The system evaluates the effects of adaptation actions, such as the reallocation of resources and the restructuring of workflows, on utility effectiveness. In spite of the fact that this method improves decision-making, it does not offer a direct calculation of composite health scores, such as the healthUtilityScore that we employ in our strategy. According to Ghahremani, Giese, and Vogel (2017), their approach places more of an emphasis on patterns than on instance-level indicators like response time or resource utilization. This approach not only offers scalability, but it also lacks the capability to make comprehensive decisions regarding particular service instances [3].

In their 2019 paper, Magableh and Almiani [6] developed a utility-driven self-healing technique for microservices. This strategy involves the maintenance of cluster state and the avoidance of simultaneous conflicting modifications. For the purpose of informing decisions, such as scaling instances and reallocating resources, the model does real-time computations of multi-dimensional utility values. Despite this, they place more of an emphasis on maintaining a constant execution context as opposed to directly improving measures such as the average response time or the average availability. Our technique, on the other hand, makes it a point to actively monitor these averages for each service. We use healthUtilityScore as a composite indicator to determine when to deploy new instances, which allows us to guarantee proactive fault recovery and better availability.

For the purpose of structural adaptation in cyber-physical systems (CPS), Ratasich et al. (2018) suggested a property-guided, utility-based strategy. The approach that they take finds components that are not performing up to expectations and replaces them with alternatives that are superior in terms of utility attributes such as accuracy. This technique is less important for software-level adaptations that dynamically manage system instances in real-time. In contrast, our strategy estimates response time, availability, and healthUtilityScore for individual instances in order to trigger proactive adaptations [5]. This strategy is more relevant for software-level adaptations.

Moreno et al. (2015) presented a proactive adaptation framework that makes use of probabilistic model verification in order to improve adaptation decisions in environments that are uncertain. Through the process of studying prospective environmental changes before they occur, their approach is able to foresee failures and commence implementation of adaptations. Despite the fact that this method is in line with the proactive part of our plan, it requires stochastic modeling and forecasting, which are both computationally demanding. By applying specified thresholds for healthUtilityScore, our technique, on the other hand, accelerates the decision-making process. This ensures real-time adaptation while simultaneously decreasing the amount of processing demands [7].

By doing a study of probable utility bounds, Stevens and Bagheri (2020) suggested a utility-based technique that reduces the amount of runtime adaptation space. Thallium, the tool that they use, precomputes utility bounds for adaptation operations, which makes it easier to make decisions more quickly during runtime. This method is consistent with the focus that our strategy places on making decisions in an effective manner; nevertheless, it relies on precomputation rather than the real-time evaluation of instance-specific metrics such as reaction time or availability. According to Stevens and Bagheri's research from 2020, their approach is better suited for environments that are either static or semi-dynamic, in contrast to our strategy, which makes adjustments dynamically based on the performance of the service in real time [8].

In the field of cloud computing, Dai et al. (2011) presented a framework for consequence-oriented self-healing. This framework makes use of fuzzy logic and decision diagrams to anticipate failures and carry out recovery procedures. For the purpose of improving recovery efficiency, the framework determines a number of measures, including workload and resource use. Our approach improves decision-making by unifying instance-level indicators into a healthUtilityScore, which directly directs instance scaling and service recovery measures (Dai et al., 2011). Although their method corresponds with our emphasis on availability and resource measurements, our approach improves decision-making by combining these metrics [9].

For the purpose of evaluating and ranking different configurations, VanSyckel et al. (2013) suggested a proactive adaptation paradigm for pervasive systems. This approach makes use of cost and appropriateness metrics. They take a proactive approach to modifying configurations in order to minimize adaption delays by anticipating future changes in the context. Despite the fact that this technique is in line with our goal of increasing availability, it places more of an emphasis on high-level configuration management rather than instance-specific measures like reaction time and healthUtilityScore, which are essential to our strategy [10].

In conclusion, the comparison highlights the special aspects of our technique, which performs an evaluation of instance-level parameters (response time, availability, and healthUtilityScore) in order to proactively identify adaption requirements. In contrast to previous approaches that concentrate on utility optimization, scalability, or predictive modeling, our method offers a practical solution that is specifically built for systems that are characterized by dynamic microservices. It does this by implementing instance-level monitoring and adaptation, which results in an increase in both the average availability and response time.

III. Exemplar Used

RAMSES, which stands for Reconfigurable Adaptive Monitoring for Self-Exploration Systems, is a framework that is modular in nature and provides support for the creation and evaluation of self-adaptive microservice applications. Maintainability, reusability, and scalability are all significantly improved as a result of the architecture's separation of adaption logic from the core program operation [11]. In order to construct its control architecture through the utilization of microservices, RAMSES makes use of the MAPE-K feedback loop, which stands for Monitor-Analyze-Plan-Execute over commonly held knowledge. This approach is consistent with the extensive use of MAPE-K as a fundamental framework for self-adaptive systems [12]. In accordance with contemporary software development standards, this architecture supports real-time monitoring and decision-making, as well as the execution of adaption strategies.

The framework makes use of an API-led integration strategy that incorporates a Contract-First design. This design offers reusable APIs for probing and actuating, which enables the framework to promote smooth integration with a variety of managed systems. The use of APIs for real-time monitoring aligns with broader efforts to create robust interfaces for self-adaptive systems [13]. The capability of RAMSES to enable advanced adaptation techniques is an important component of the system. These strategies include the dynamic addition or removal of service instances, the reconfiguration of load balancers, and the switching of service implementations in order to increase system performance and ensure system stability [11]. In the course of actual evaluations, such as its deployment in an e-food microservice application, RAMSES demonstrated its capacity to improve runtime Quality of Service (QoS) attributes, such as availability and response time, while simultaneously preserving self-* qualities, such as self-healing and self-optimization.

The capabilities of RAMSES are strongly aligned with our proactive self-healing strategy, which is centered on utility-driven adaptation. It makes use of its monitoring and adaptive capabilities in order to address particular issues that are prevalent in environments that are dynamic and comprised of microservices. Through the process of aggregating indicators like availability, CPU use, and remaining disk percentage, the method is able to calculate a healthUtilityScore for each individual service instance. This aligns with Bradbury et al.'s findings, which emphasize runtime monitoring and dynamic decision-making as key elements of self-management in software systems [25]. Providing a real-time assessment of the overall health and performance of an instance, the composite score serves two purposes. In the event that the healthUtilityScore of an instance falls below a certain threshold, the strategy will promptly launch an adaptation action in order to include a new instance into the service. This ensures that the overall average availability and response time of the service are either maintained or improved [12].

The application programming interfaces (APIs) of RAMSES offer robust monitoring and probing capabilities. These features enable the continuous collection of instance-level metrics, which in turn enables real-time health evaluations and ensures the accuracy of the healthUtilityScore number. The Contract-First design of the framework makes it easier to incorporate additional monitoring and actuation capabilities. This helps to ensure that proactive adjustments are performed without disrupting operations that are already in progress [14]. By dynamically adding new instances or reconfiguring services, the method makes the most efficient use of available resources. As a result, it reduces the impact that deterioration in performance or exhaustion of resources have on the overall availability of services.

This technique solves a basic difficulty associated with adaptation, which is to guarantee consistent service availability in spite of unequal resource consumption or performance degradation in certain situations. It is possible for an instance to have a negative impact on the overall availability and response time of the service if it displays low availability, high CPU consumption, or restricted storage space. RAMSES allows for the effective identification and resolution of these difficulties through the reallocation of workloads and the dynamic scaling of resources. Because of this, the system is able to make adjustments in real time to changing workloads or constraints on resources, hence maintaining quality of service standards and improving service resilience [12].

IV. Approach

A. Adaptation Strategy in depth

Our adaptation strategy is structured around two core phases: the ”Fault Detection Phase” and the ”Shadowing Phase.” These phases embody well-established theoretical principles from fault-tolerant distributed systems and adaptive resource allocation strategies, ensuring that system availability is consistently maintained under dynamic conditions. By integrating proactive mechanisms, our strategy aims to pre-emptively mitigate failures and optimize system performance. The scientific basis of this approach lies in the principles of fault tolerance, real-time monitoring, and dynamic resource management, which have long been pillars of resilient system design.

In the Fault Detection Phase, the focus is on proactive identification of potential or imminent failures in service instances. This phase introduces a utility function-based mechanism to assess the health of individual instances, a departure from traditional reactive failure detection methods. Key metrics such as availability, average response time, CPU usage, and disk utilization are continuously monitored, normalized, and synthesized into a composite health utility score. This score is dynamically adjusted through penalties for degraded performance metrics, such as high response times, ensuring that the system captures even subtle signs of impending failure. The system identifies unhealthy instances prior to complete failure by establishing a predefined threshold for the health utility score, triggering alerts when the score falls below this threshold. This proactive mechanism is rooted in predictive analytics and utility-based evaluation, which prioritize metrics according to their significance in maintaining service availability. Instances flagged as unhealthy are addressed immediately, thereby reducing the likelihood of cascading failures and service disruptions. This phase reflects a forward-thinking approach to system adaptation, minimizing downtime and ensuring high reliability through early intervention.

The Shadowing Phase is initiated after a new instance has been added in response to the detection of an unhealthy instance during the Fault Detection Phase. The primary objective of this phase is to dynamically recalibrate system resources to mitigate the impact of the unhealthy instance while leveraging the newly added one. Specifically, this phase adapts the load balancer's weights to distribute traffic more effectively based on the health scores of active instances. By redistributing traffic away from unhealthy instances and toward healthier ones, particularly the newly added instance, the system ensures optimal resource utilization and balanced load distribution. For example, a newly introduced instance, assumed to have a perfect health utility score, is allocated a larger share of incoming traffic, while the unhealthy instance receives a reduced share proportional to its health score relative to the total. This mechanism reflects dynamic resource allocation theories, which advocate for the continuous reconfiguration of resources in response to changing conditions. By prioritizing healthier instances, the system alleviates the burden on degraded ones, thereby preventing additional performance decline and potential failures. This method guarantees improved availability and reduced response times. This strategic rebalancing not only enhances system stability but also prolongs the operational lifespan of all instances.

Together, the Fault Detection and Shadowing Phases provide a comprehensive framework for proactive adaptation, ensuring that system health is evaluated holistically. By incorporating metrics such as availability, response time, CPU usage, and disk utilization into a unified framework, the strategy can effectively detect early signs of instance degradation. The proactive nature of the Fault Detection Phase enables the system to preemptively address potential issues before they escalate into failures. Following this, the Shadowing Phase focuses on redistributing traffic by dynamically adjusting load balancer weights after adding a new instance. Healthier instances are prioritized, while unhealthy ones are relieved of excess load, mitigating further degradation. This two-phased approach not only ensures high availability and low response time but also maintains a balanced and efficient system, adapting dynamically to evolving conditions while minimizing disruptions.

B. Explanation of the Theories

Theoretical Foundations: Within the context of self-adaptive systems, this work provides an implementation of an adaptation approach that is founded on utility theory and health-based monitoring methods. In order to compute an all-encompassing health score for system instances, the technique makes use of a multi-dimensional utility function that incorporates a number of different Quality of Service (QoS) measures.

Utility-Based Decision Making: The core of our approach relies on utility theory, which provides a mathematical framework for quantifying utility preferences and making decisions under uncertainty. In our context, we define a health utility score that aggregates multiple system metrics. This aligns with Shah and Patel's (2022) [16] finding that effective self-healing systems must integrate "real-time monitoring, predictive analytics, and fault remediation" capabilities. Our health utility score is calculated as:

$$H = \left(\frac{w_1 A + w_2 C + w_3 D}{w_1 + w_2 + w_3} \right) \times 100 - R_t$$

Where A represents system availability (0-1), C represents CPU utility (1 - CPU usage), D represents the disk space availability ratio, w_1, w_2, w_3 are importance weights, and R_t is a response time penalty factor. This formulation follows the principles of utility theory, normalizing each metric to a [0,1] scale for the sake of comparability. The weights indicate the significance of each factor, and the overall instance score is adjusted according to the response time of incoming requests.

Health-Based Monitoring: The strategy implements a health-based monitoring approach that continuously collects metrics on service instances, and evaluates both individual instance health and aggregate service health. This aligns with Heart and Ibrahim's (2020) [15] assertion that health checks should be automated processes that continuously monitor service status.

Whenever an instance's utility score falls below the predefined threshold, the health monitoring system identifies it as "unhealthy," which then prompts the system to take appropriate measures for adaptation.

Load Balancing Theory: When unhealthy instances are detected, the strategy employs a two-fold adaptation approach identified by Heart and Ibrahim (2020) [15] as essential for maintaining service continuity:

1) **Instance Addition:** For each service with issues, a new instance is added to maintain service capacity.
2) **Load Balancing:** For services with exactly one unhealthy instance, the load balancer weights are adjusted according to:

Load Balancing Theory ensures a proportional distribution of traffic based on instance health, gradually reducing load on degraded instances while maintaining service availability.

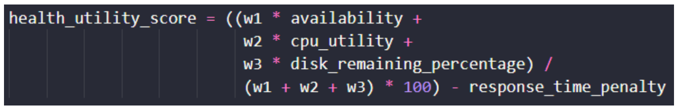
C. RAMSES Design with Novel Strategy

Our self-adaptive system is built upon the RAMSES framework, with our novel adaptation strategy serving as a key component to enhance fault detection and proactive adaptation capabilities. The system operates within the modular structure provided by RAMSES, where each phase of the MAPE-K loop is implemented to ensure robustness and adaptability.

The Knowledge component is central to our strategy, acting as the repository for all critical system data, particularly the structure of *self.knowledge.analysis_data*. This aspect of the Knowledge base is meticulously designed to organize and store key outputs of the Analyze phase. It comprises the following fields:

- **Failed Instances:** Instances explicitly detected as failed or unreachable are stored in this field. It serves as a direct indicator of critical issues requiring immediate adaptation, such as adding new instances to maintain availability.
- **Unhealthy Instances:** This field flags instances falling below the predefined health utility score threshold. These instances are proactively identified based on deteriorating performance metrics, enabling corrective actions before further degradation occurs.
- **QoS History:** This detailed instance-level record includes metrics like availability, average response time, CPU usage, and disk space utilization. It enables trend analysis and informed decision-making about the state of individual instances.
- **Service Average Metrics:** This aggregated view provides metrics such as average availability, response time, and health utility score across all instances of a service. It supports a holistic assessment of service performance.

The self.knowledge.plan_data structure is another critical aspect of the Knowledge component. This field is used to



```
health_utility_score = ((w1 * availability +
                         w2 * cpu_utility +
                         w3 * disk_remaining_percentage) /
                        (w1 + w2 + w3) * 100) - response_time_penalty
```

Fig. 1. Health Utility Score

schedule adaptation plans. It contains a sequence of operations that must be executed to address detected issues. For instance, in cases of unhealthy instances, an addInstances operation is first scheduled to add a new instance to the affected service. Following this, a changeLBWeights operation adjusts the load balancer weights to distribute traffic more effectively, prioritizing healthier instances.

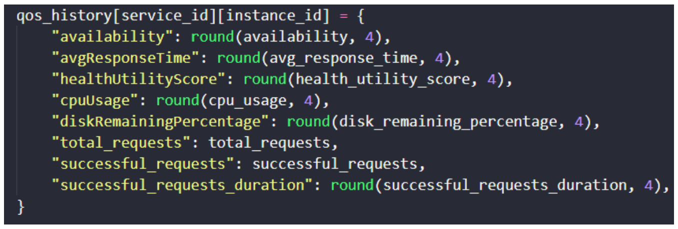
The execution of adaptation plans ensures that planned operations are performed in sequence, prioritizing instance addition before adjusting load balancer weights. This ensures the system maintains availability while redistributing traffic efficiently.

D. Integration with UPISAS and Experiment Runner

In the **Monitor** phase, the monitor function is responsible for collecting real-time data about the system's state. This function integrates with the UPISAS framework by making a GET request to the system's monitoring endpoint, fetching the latest snapshot of metrics for each service and its instances. The implementation utilizes the *self._perform_get_request* method to retrieve this data and appends it to self.knowledge.monitored_data. The data structure of self.knowledge.monitored_data is directly leveraged in the Analyze phase to calculate critical metrics.

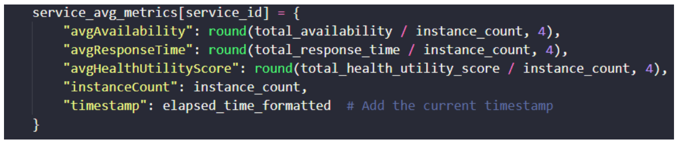
The **Analyze** method begins by accessing the *self.knowledge.monitored_data*, which contains the most recent monitored information about all services and their instances. It initializes key data structures, including lists for failed_instances and unhealthy_instances to track problematic instances, and dictionaries for qos_history and service_avg_metrics to store quality-of-service metrics and average service-level data, respectively. To ensure previous adaptations are not repeated unnecessarily, the method checks for the existence of *self.knowledge.adapted_instances* and initializes it as a set if it does not exist.

The method then defines constants such as weights for the utility function and thresholds for penalizing response times, which are used to compute the health utility score. For each service in the monitored data, the method processes all associated instances, extracting and aggregating key metrics such as availability, average response time, CPU usage, and disk utilization. These metrics are carefully calculated, with null-safe checks applied to ensure robustness in cases of missing or invalid data. The health utility score is derived using a weighted combination of availability, CPU utility, and remaining disk space, with deductions applied for high response times. This score provides a composite view of an instance's overall health. Implementation can be observed from Fig.1.



```
qos_history[service_id][instance_id] = {
    "availability": round(availability, 4),
    "avgResponseTime": round(avg_response_time, 4),
    "healthUtilityScore": round(health_utility_score, 4),
    "cpuUsage": round(cpu_usage, 4),
    "diskRemainingPercentage": round(disk_remaining_percentage, 4),
    "total_requests": total_requests,
    "successful_requests": successful_requests,
    "successful_requests_duration": round(successful_requests_duration, 4),
}
```

Fig. 2. QOS History Metrics



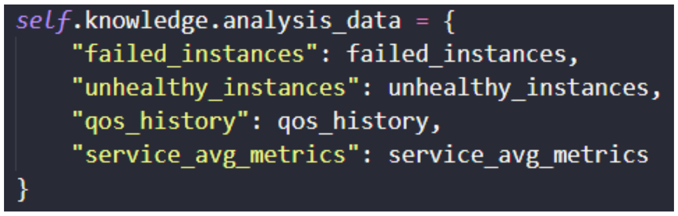
```
    service_avg_metrics[service_id] = {
        "avgAvailability": round(total_availability / instance_count, 4),
        "avgResponseTime": round(total_response_time / instance_count, 4),
        "avgHealthUtilityScore": round(total_health_utility_score / instance_count, 4),
        "instanceCount": instance_count,
        "timestamp": elapsed_time_formatted  # Add the current timestamp
    }
```

Fig. 3. Service Average Metrics

Instances with a health utility score below a predefined threshold are flagged as unhealthy, while those explicitly marked as failed or unreachable are added to the failed_instances list. To prevent duplicate adaptations, adapted instances are tracked in *self.knowledge.adapted_instances*. At the service level, the method calculates aggregated metrics such as average availability, average response time, and average health utility score, which are stored in service_avg_metrics, alongside a formatted timestamp representing the time elapsed since the start of the strategy as can seen in Fig.3. Finally, the method updates *self.knowledge.analysis_data* Fig.4 with the results, including the lists of failed and unhealthy instances, the QoS history Fig.2, and the service average metrics. To avoid unnecessary memory growth, it resets the monitored data at the end of the phase. If no issues are detected, the method concludes by indicating that no adaptation is required; otherwise, it returns True to signal that adaptation actions are needed.

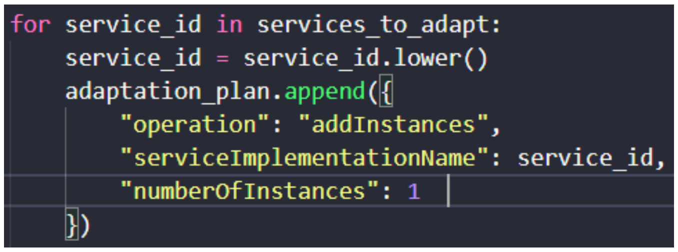
The **Plan** method plays a critical role in the adaptation strategy by crafting a series of actions to address system issues detected during the *analyze* phase. This phase evaluates the analysis data stored in *self.knowledge.analysis_data* to decide how to adapt the system effectively. Specifically, the method focuses on two primary objectives: adding new instances to services with unhealthy or failed instances and adjusting load balancer weights to optimize traffic distribution.

The method starts by extracting the list of *failed_instances* and *unhealthy_instances* from the analysis data, along with



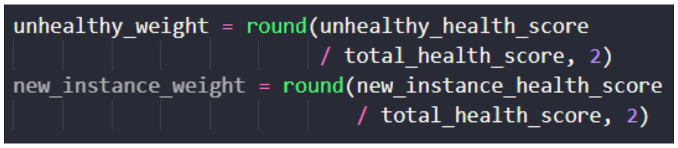
```
self.knowledge.analysis_data = {
    "failed_instances": failed_instances,
    "unhealthy_instances": unhealthy_instances,
    "qos_history": qos_history,
    "service_avg_metrics": service_avg_metrics
}
```

Fig. 4. knowledge.analysis_data



```python
for service_id in services_to_adapt:
    service_id = service_id.lower()
    adaptation_plan.append({
        "operation": "addInstances",
        "serviceImplementationName": service_id,
        "numberOfInstances": 1
    })
```

Fig. 5. Add instance to Service

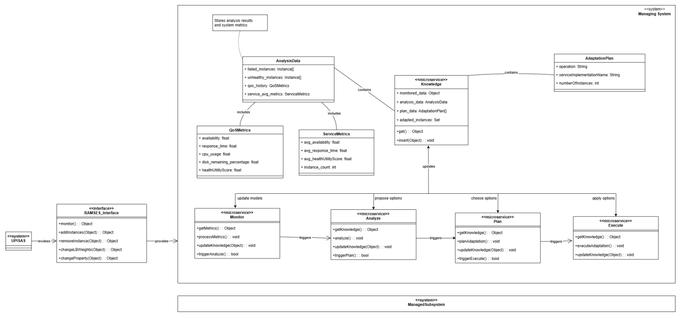


```python
unhealthy_weight = round(unhealthy_health_score
                         / total_health_score, 2)
new_instance_weight = round(new_instance_health_score
                            / total_health_score, 2)
```

Fig. 6. Calculation of Load Weights

qos_history for detailed quality-of-service metrics. An empty *adaptation_plan* list is initialized to store the planned adaptation actions. The first step in planning is identifying the services that require adaptation, which are deduced by collecting the unique *service_id* values from both *failed_instances* and *unhealthy_instances*. For each identified service, an ”addInstances” operation is added to the *adaptation_plan*. This operation specifies the addition of one new instance to the affected service, ensuring system resilience and improved availability. Next, the method addresses load balancing for services with exactly one unhealthy instance. For such services, the health score of the unhealthy instance is retrieved from the *qos_history*, and a new instance is assumed to have a perfect health score of 100. The method calculates proportional load balancer weights for the unhealthy instance and the newly added instance using their respective health scores. The unhealthy instance’s weight is computed as the ratio of its health score to the total health score of both instances, while the new instance’s weight is calculated similarly. These weights are then incorporated into a ”changeLBWeights” operation, added to the *adaptation_plan*. This operation ensures that traffic is directed primarily to the healthier instance, protecting the degraded instance from further stress. Finally, the *adaptation_plan* is stored in *self.knowledge.plan_data* for execution in the next phase, and the method concludes by printing the updated plan and returning a boolean value indicating whether any adaptation actions were planned. This approach ensures that adaptation actions are precise and well-targeted, maintaining system performance and reliability while minimizing disruption.

The **Execute** method is responsible for enacting the adaptation plan generated during the *plan* phase. It iterates through each action in *self.knowledge.plan_data*, ensuring that the adaptations are applied sequentially. For each action, the method validates its schema using the preloaded *execute_schema* to ensure compatibility with the system. It then sends the adaptation request as a POST request to the



<<system>>
Managing System
Stores analysis results and system metrics
AnalysisData
+ failed_instances: Instance[]
+ unhealthy_instances: Instance[]
+ qos_history: QoSMetrics
+ service_avg_metrics: ServiceMetrics
contains
<<microservice>>
Knowledge
+ monitored_data: Object
+ analysis_data: AnalysisData
+ plan_data: AdaptationPlan[]
+ adapted_instances: Set
+get() : : Object
+insert(Object) : : void
contains
AdaptationPlan
+ operation: String
+ serviceImplementationName: String
+ numberOfInstances: int
includes
includes
QoSMetrics
+ availability: float
+ response_time: float
+ cpu_usage: float
+ disk_remaining_percentage: float
+ healthUtilityScore: float
ServiceMetrics
+ avg_availability: float
+ avg_response_time: float
+ avg_healthUtilityScore: float
+ instance_count: int
updates
update models
propose options
choose options
apply options
<<system>>
UPISAS
invokes
<<interface>>
RAMSES_Interface
+monitor() : : Object
+addInstances(Object) : : Object
+removeInstance(Object) : : Object
+changeLBWeights(Object) : : Object
+changeProperty(Object) : : Object
provides
<<microservice>>
Monitor
+getMetrics() : : Object
+processMetrics() : : void
+updateKnowledge(Object) : : void
+triggerAnalyze() : : bool
triggers
<<microservice>>
Analyze
+getKnowledge() : : Object
+analyze() : : void
+updateKnowledge(Object) : : void
+triggerPlan() : : bool
triggers
<<microservice>>
Plan
+getKnowledge() : : Object
+planAdaptation() : : void
+updateKnowledge(Object) : : void
+triggerExecute() : : bool
triggers
<<microservice>>
Execute
+getKnowledge() : : Object
+executeAdaptation() : : void
+updateKnowledge(Object) : : void
<<system>>
ManagedSubsystem

Fig. 7. Overview of the self-adaptive system’s design

designated endpoint, *execute*, which is constructed using the system’s base URL.

The method handles errors gracefully, logging any issues such as unreachable endpoints, which would raise a custom *EndpointNotReachable* exception. This mechanism ensures robust execution of adaptation actions, including adding instances or adjusting load balancer weights, as planned in the previous phase. The method concludes by confirming successful execution or reporting failures for debugging and further analysis. The **Experiment Runner** integrates the RAMSES system and our novel strategy for testing adaptation mechanisms under controlled scenarios. It is implemented as part of the RunnerConfig class, which is configured to execute the experiment seamlessly using predefined event-based triggers.

The experiment begins with the initialization of the runner configuration, where essential components like the exemplar (RAMSES) and strategy (RamsesNovelStrategy) are instantiated during the ”before run” phase. The ”start run” phase ensures that Scenario 5, created to evaluate our strategy, is initiated properly. A critical step here involves disabling the existing adaptation mechanisms of RAMSES to ensure that only our novel strategy operates during the experiment. This step is crucial for isolating and testing the effectiveness of our proactive and reactive adaptations.

The core of the experimental process is handled in the ”interact” phase, where the strategy is executed iteratively. Every 5 seconds, the strategy’s monitor, analyze, plan, and execute phases are triggered. This cycle continues for a total of 7 minutes, with the goal of simulating realistic system conditions and testing how effectively the strategy adapts to changes like increased request loads or simulated failures. The iterative execution ensures the system continuously evaluates and responds to dynamic conditions, such as detecting unhealthy instances and adjusting load balancer weights based on health scores.

At the end of the experiment, the ”stop run” phase ensures proper shutdown of the scenario, preserving the integrity of the experiment data. The runner also includes provisions for measuring and recording results, although the details of data aggregation and analysis are abstracted within the provided code.

In summary, the experiment runner acts as the orchestrator for testing our novel strategy within the RAMSES framework.

It facilitates the execution of Scenario 5 under controlled conditions, leveraging RAMSES' extensibility to integrate and evaluate the effectiveness of our adaptation mechanisms. The structured phases, from setup to monitoring and final shutdown, ensure reproducibility and clarity in experimentation. For detailed implementation:GitHub Repository. https://github.com/ahmetkarapinar/Upisas_2_2

E. Baseline Strategy

The baseline strategy operates on a reactive adaptation mechanism, where failures are detected only when they have already occurred. Unlike the proactive nature of our novel strategy, the baseline approach does not attempt to predict or preempt potential issues. It relies solely on monitoring instance statuses to identify outright failures or unreachable states. Upon detection of such failures, the strategy triggers an immediate response by adding a new instance to replace the failed one. This ensures continuity of service but does not account for early indicators of degradation, such as decreasing availability or worsening performance metrics. The baseline strategy thus serves as a straightforward, minimal adaptation mechanism, offering a clear comparison point for evaluating the effectiveness of more advanced, proactive approaches.

V. Evaluation

A. Experiments

For the experimentation, a custom scenario was specifically designed to compare the novel strategy with the baseline strategy. This new scenario was created using the predefined functions in the rest-client folder of the RAMSES project. The scenario was built as a Docker image to ensure portability and ease of use across different experimental setups. In UPISAS experimentation, this image is directly pulled from Docker Hub and utilized as part of the testing environment. The image is publicly available under the name *"ahmetkarapinar/scenario5:latest."*

The scenario runs for a total duration of five minutes. At the 60-second mark, intentional faults are introduced by simulating fake exceptions for the sole instance of the order service. This is achieved by increasing its failure probability to 0.5, a well-established method for simulating availability issues. As the scenario progresses, further stress is applied to the same instance. At the 120-second mark, the request load directed to this instance is significantly increased, leading to a rise in CPU usage and a reduction in its remaining disk space. Simultaneously, artificial delays are introduced by increasing the sleep duration of the instance from 0 to 2000 milliseconds. This adjustment simulates a noticeable degradation in the instance's average response time, mimicking real-world conditions where resource exhaustion or processing bottlenecks affect performance. Finally, at the 240-second mark, a deliberate failure is injected, rendering the instance inoperative.

This scenario effectively replicates the behavior of a service instance that is on the verge of failure and requires immediate detection. We believe that this carefully designed scenario provides an ideal test bed for evaluating whether the novel strategy can proactively identify and adapt to such failures before they occur, thereby demonstrating its effectiveness in maintaining service availability and reliability.

B. Measurement Metrics

To gauge the successful adaptation of the exemplar, we focus on two critical metrics: availability and response time. These metrics are fundamental to evaluating the effectiveness of any self-healing strategy, as they directly reflect the system's ability to maintain service quality under adverse conditions. Availability, in particular, is widely regarded as the most robust metric for assessing self-healing capabilities. It measures the proportion of time the service remains operational and accessible to users, providing a clear indication of the system's resilience to failures. A high availability score demonstrates that the adaptation mechanism successfully mitigates failures and ensures uninterrupted service delivery.

Response time, on the other hand, complements availability by evaluating the system's performance during and after adaptation. It captures the efficiency of the system in handling requests, even as it undergoes changes to address failures or deteriorating conditions. A lower response time signifies not only effective failure management but also an optimized load distribution and resource utilization, which are essential for maintaining user satisfaction.

By analyzing availability and response time, we can holistically evaluate the success of the adaptation strategy. A strategy that effectively prevents significant drops in availability while maintaining or improving response time demonstrates its capability to ensure seamless and reliable operation. These metrics provide quantifiable evidence of the system's ability to self-heal and adapt dynamically to maintain service continuity and quality.

C. Results

The results clearly demonstrate that our novel strategy successfully detects impending failures before they occur. To better understand this achievement, let us closely examine the simulation. As described earlier, at the 60-second mark, the specific instance of the ordering service begins to experience a decline in availability as it can be seen in Fig8. Importantly, this decline does not trigger the predefined QoS maintenance strategies of RAMSES. This is because the decrease in availability is deliberately subtle, selected to bypass RAMSES' built-in mechanisms. By carefully choosing this value for the simulated exceptions, we ensured that the failure detection capabilities of our novel strategy would be meaningfully tested. As shown in the Fig8, the availability of the instance drops to approximately 0.6, a value insufficient to activate RAMSES' default mechanisms, yet indicative of potential issues.

At the 120-second mark, additional stress is applied to the instance by increasing its request count. This leads to heightened CPU utilization and a reduction in the remaining disk space, further deteriorating the instance's performance.

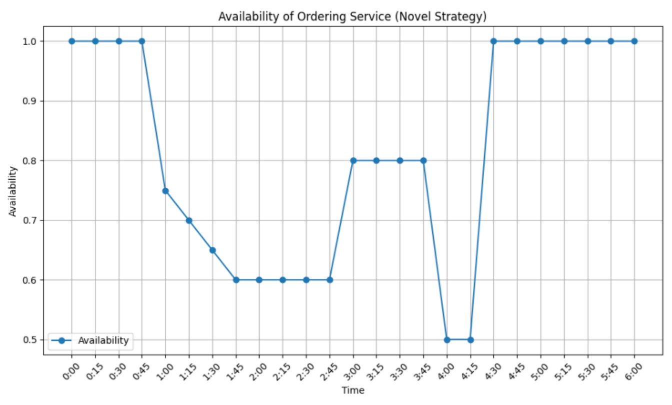


Fig. 8. Ordering Service Availability with Novel Strategy

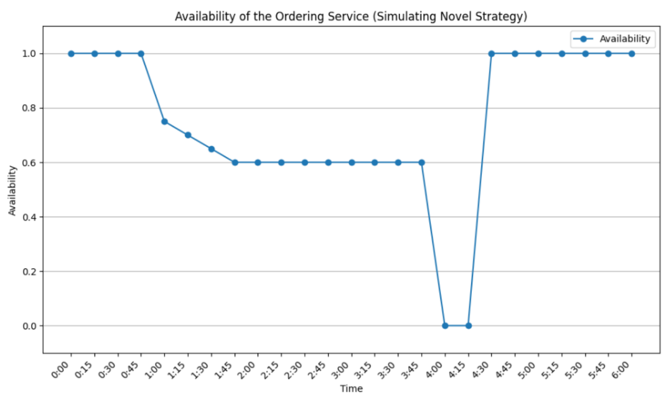


Fig. 9. Ordering Service Availability with Baseline Strategy

Simultaneously, artificial delays are introduced through extended sleep periods, simulating an increase in average response time as it can be seen in Fig11. Since our health utility score is calculated based on average response time, availability, CPU usage, and remaining disk space, these combined factors result in the health utility score dropping below the predefined threshold. This decline serves as a warning signal for our strategy, allowing it to detect an imminent failure. Our strategy promptly responds by adding a new instance to the ordering service, effectively addressing the issue before the failure occurs. The addition of the new instance to the Ordering service can be observed in both Fig8 and Fig11 at time 3:00.

In addition to adding a new instance, the second phase of our strategy, referred to as the ”Shadowing Phase,” comes into play. During this phase, load balancing weights are adjusted based on the health utility scores of the instances. The unhealthy instance is assigned a lower weight, ensuring it receives fewer requests compared to healthier instances. This proactive adjustment not only mitigates the immediate impact of the failure but also reduces the strain on the unhealthy instance, potentially delaying or preventing its complete failure.

As evident from Fig10 and Fig11, the addition of the new instance leads to significant improvements in the availability and average response time of the ordering service. At the 240-second mark, the unhealthy instance eventually fails, as was intended in the simulation. However, this failure had already been anticipated and addressed by our strategy. By adding a new instance and adjusting the load balancing weights beforehand, the strategy ensured that the system maintained stable availability and performance, effectively avoiding a drastic drop in service quality. These results validate the robustness and foresight of our novel strategy in managing and mitigating potential failures of the strategy. Fig10 and Fig11 clearly show that during a failure event, the novel strategy not only recovers more quickly but also sustains less damage compared to the baseline.

The baseline strategy provides a reactive approach to failure management, focusing on addressing issues only after they have occurred. This contrasts with the proactive measures implemented in our novel strategy, which seeks to detect and mitigate failures before they impact the system significantly. To better understand the differences, let us examine the actions of the baseline strategy in detail and compare its outcomes to those achieved by our novel approach.

In the simulation scenario, the baseline strategy detects failures by monitoring the state of each instance. When an instance’s status is flagged as ”FAILED” or ”UNREACHABLE,” the baseline strategy responds by adding a new instance to replace the failed one. This action ensures that the overall service remains functional, albeit after some degradation in system performance and availability. The baseline strategy operates solely on observable failure states, without considering the warning signs that might indicate an impending failure. For example, in our simulation, the ordering service instance begins to show declining availability and increasing response times at the 60-second and 120-second marks, respectively. These signs, while indicative of potential failure, do not trigger any response under the baseline strategy because the instance is still technically operational as can be seen in Fig.8.

The consequences of this reactive approach become evident at the 240-second mark when the unhealthy instance fails entirely. At this point, the baseline strategy identifies the failure and adds a new instance to compensate. However, this delayed reaction results in a noticeable decline in service availability and performance during the interim period between the failure and the addition of the new instance as shown in both Fig.8 and 11. The lack of proactive measures means that the system must endure the full impact of the failure before recovery actions can be implemented.

In contrast, the novel strategy addresses these limitations by employing a proactive failure detection mechanism. As discussed in the earlier paragraphs, the novel strategy utilizes a health utility score based on metrics such as availability, response time, CPU usage, and remaining disk space. This allows the strategy to identify instances that are at risk of failing and take corrective actions before the failure occurs. For example, when the health utility score of the ordering service instance drops below the predefined threshold at the 3:00 mark, the novel strategy preemptively adds a new instance and adjusts the load balancing weights. This action reduces

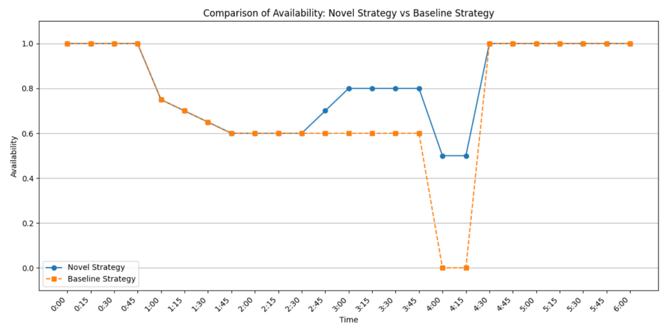


Fig. 10. Ordering Service Availability Comparison

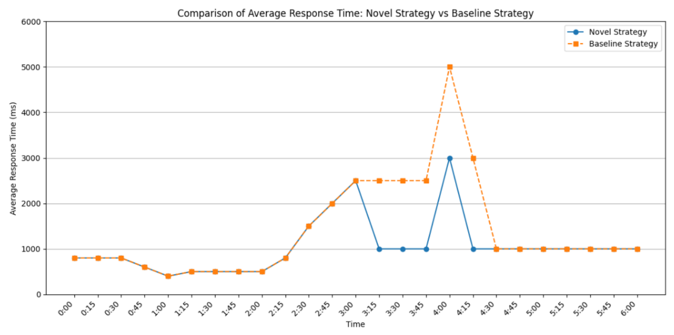


Fig. 11. Ordering Service Response Time Comparison

the load on the unhealthy instance, giving it a better chance to recover while maintaining overall service stability.

The comparison between the two strategies highlights the advantages of the novel approach. While the baseline strategy effectively manages failures after they occur, it lacks the foresight to prevent or mitigate their impact in advance. This often results in temporary declines in availability and performance, which could be detrimental in scenarios where high service reliability is critical. The novel strategy, by contrast, minimizes these disruptions by acting on early warning signs and redistributing system resources dynamically. The results demonstrate that the novel strategy successfully detects impending failures and takes proactive actions, leading to superior performance in terms of availability and response time compared to the baseline strategy before the failure occurred (between 1:00 and 4:00). Furthermore, at the critical failure moment at 4:00, the novel strategy incurred less damage than the baseline, owing to the precautions it had already implemented. During the recovery phase, the novel strategy once again outperformed the baseline by achieving a faster recovery between 4:00 and 4:30. These effects are quantitatively evident in the comparison of availability and response time, as illustrated in Fig.10 and Fig.11.

D. Hypothesis Assessment

In the introduction, we hypothesized that our novel strategy, utilizing a proactive failure detection mechanism combined with adaptive load balancing, would outperform the baseline reactive approach in maintaining system availability and performance under adverse conditions. The results of the experiments strongly validate this hypothesis.

Our novel strategy employs a utility-based function to proactively detect early signs of failure. This utility function incorporates multiple metrics, such as availability, response time, CPU usage, and remaining disk space, to calculate a comprehensive health score for each instance. Unlike traditional mechanisms that focus solely on maintaining predefined QoS metrics, this health utility function provides a holistic view of the instance's operational state, enabling the system to detect potential failures before they fully manifest. By acting on these early indicators, the strategy adds a new instance and adjusts load balancer weights to mitigate the impact on system performance.

The results clearly show the advantages of this approach. In the ordering service instance, the novel strategy detected early signs of failure, such as declining availability and increasing response time, well before the failure occurred. This proactive detection allowed the system to initiate adaptation actions, including adding a new instance and redistributing load, ensuring that availability remained stable and response times improved. Importantly, even when the unhealthy instance ultimately failed, the system's availability and performance were not drastically affected, demonstrating the strategy's effectiveness in maintaining system resilience.

In contrast, the baseline strategy, relying on reactive mechanisms, detected failures only after they had fully occurred. This delayed response led to a temporary degradation in availability and response time, as the system struggled to compensate for the already failing instance. The absence of proactive detection mechanisms meant that the baseline strategy lacked the ability to preemptively safeguard the system from service disruptions.

These results confirm that the utility-based proactive failure detection mechanism employed in our novel strategy is a significant improvement over traditional reactive methods. By addressing potential failures before they escalate and complementing this with targeted adaptation actions, the strategy not only ensures the continuity of service but also provides a robust alternative to mechanisms that solely aim to maintain predefined QoS metrics. This reinforces the importance of proactive approaches in modern self-healing systems.

VI. Discussion

The results of our evaluation emphasize the significant advantages of our novel strategy over reactive self-healing mechanisms. While both strategies aim to maintain system availability and performance, they are fundamentally different in approach. Reactive mechanisms, such as the baseline strategy, address issues only after failures occur. In contrast, our novel strategy proactively detects potential failures before they manifest, leveraging a utility-based function that incorporates metrics like availability, response time, CPU usage, and disk space.

One key implication of our results is the effectiveness of proactive detection in maintaining higher system availability and more stable response times under stress. By identifying and mitigating potential failures before they escalate, our strategy ensures that disruptions are addressed early, avoiding the cascading effects typically seen in reactive systems. This proactive approach results in smoother system operation and improved user experience, as evidenced by the graphs showing stable availability and response times even during failures.

The baseline strategy, which represents a traditional reactive self-healing mechanism, revealed its limitations during the evaluation. Reactive mechanisms depend on identifying and addressing failures after they occur, which inherently involves a period of degraded service before recovery actions take effect. This delay was apparent in the transient dips in availability and response time degradation observed in the baseline strategy's performance. These shortcomings highlight the reactive approach's inability to prevent failures proactively, making it less suitable for dynamic and high-stakes environments.

Another implication of our findings is the novel strategy's ability to adapt system resources dynamically. By adjusting load balancing weights based on health scores, the strategy optimizes resource utilization, ensuring that healthier instances handle more requests while protecting vulnerable ones. This adaptability enhances the system's overall resilience and mitigates the risk of overloading failing instances, a scenario that reactive strategies are not equipped to handle preemptively.

Finally, it is crucial to note that our utility-based failure detection mechanism operates independently of QoS maintenance mechanisms like those in RAMSES. While traditional QoS mechanisms focus on maintaining predefined performance thresholds, our approach seeks to detect and address potential failures proactively, even before QoS metrics are significantly affected. This distinction underscores the novel strategy's ability to provide an additional layer of resilience, complementing rather than replacing existing QoS maintenance systems.

A. *Future improvements and limitations*

Given the results of our experiments, several improvements and limitations of the current system become evident. One significant enhancement that we could pursue, if we had more time and resources, would be the integration of machine learning (ML) techniques to refine and dynamically adjust the utility function used for proactive failure detection. While our current approach relies on static weights and predefined thresholds, incorporating ML would allow the system to evolve and learn from operational data, making it more adaptable to varying and complex real-world conditions. The dynamic adjustment of utility functions through ML, as discussed in [6], highlights how this method could significantly enhance real-time adaptation and fault detection policies [6].

Incorporating such techniques would enable the system to analyze historical and real-time data to detect patterns and trends that indicate potential failures, such as increases in CPU usage, abnormal memory consumption, or spikes in response times [6]. Anomaly detection models like Numenta's Unsupervised Real-Time Anomaly Detection (NUPIC), as referenced in the paper, could add an entirely new dimension to failure prediction, reducing false positives and capturing subtle signs of impending issues that might be overlooked by static utility functions [6]. For instance, an ML-based model could weigh response time more heavily during periods of high user activity, prioritize CPU and memory metrics under computationally intensive workloads, or even adjust the sensitivity of failure detection thresholds dynamically during off-peak hours when fewer users are interacting with the system.

Another example could be the dynamic adjustment of health utility weights based on the criticality of specific services. If a service handles high-priority transactions, such as payments or user authentication, the ML model could assign greater weight to the availability metric for that service to ensure it remains operational under all circumstances. Conversely, for services like data logging or analytics, which may tolerate occasional delays, the system could reduce the emphasis on response time or CPU usage to focus resources on higher-priority components.

However, implementing such an advanced mechanism within the constraints of a master's course was challenging, given the time and technical limitations. Building a robust ML pipeline would have required not only significant time for design and implementation but also extensive computational resources for training and testing the models. These factors would have added considerable complexity to the project, making it feel overly ambitious for the scope of the course.

Nevertheless, exploring ML-based approaches remains an exciting direction for future work. By dynamically adapting utility functions and leveraging real-time data, the system could move beyond simple proactive detection and evolve into a self-optimizing framework capable of handling a wide variety of failure modes [6].

Another significant limitation encountered in our work was the complexity and constraints of scenario creation within RAMSES. The framework relies heavily on predefined functions for scenario execution, which limited our ability to design and perform diverse simulations. This restriction hindered the exploration of more varied and nuanced failure scenarios, reducing the scope for testing the adaptability and robustness of the proposed strategy in broader contexts.

VII. Conclusion

In conclusion, this work focuses on evaluating and comparing a novel proactive adaptation strategy with a traditional reactive baseline strategy within the RAMSES framework. The primary goal was to assess improvements in handling failures, particularly in terms of availability and response time. The novel strategy introduced a utility-based proactive detection mechanism, leveraging real-time metrics to anticipate potential failures and adapt the system accordingly. Through a systematic two-phase approach of failure detection and load balancing adjustment, the novel strategy demonstrated its effectiveness in maintaining higher availability and mitigating response time spikes during simulated failure events.

The results clearly highlight the novel strategy's ability to recover more effectively from failures compared to the baseline. While the baseline strategy only reacted after failures occurred, resulting in significant drops in availability and prolonged response time spikes, the novel strategy proactively detected potential issues and implemented adaptations to minimize disruption. The novel strategy proves to be highly effective in handling failures, demonstrating significant success in recovery for both availability and response time. During failure events, the novel strategy not only recovers more quickly but also sustains less damage compared to the baseline. It consistently maintains higher availability and mitigates response time spikes more effectively, ensuring smoother performance and reducing the impact of failures on the overall system. This highlights its superior capability in minimizing disruption and maintaining service reliability under challenging conditions.

In terms of contributions, the group members evenly split the responsibilities. Ahmet was primarily responsible for implementing the adaptation strategy, including developing the monitoring, analyzing, planning, and execution phases within the UPISAS framework. Elif focused on research and resource exploration, ensuring the integration of the latest concepts and identifying relevant methodologies to strengthen the theoretical foundation of the project. Marc supported the experimental setup and evaluation, managing the experiment runner and ensuring the accurate execution of scenarios to validate the strategy's performance.

Coordination was achieved through regular team meetings, where progress and challenges were discussed. Tasks were delegated based on each member's expertise, and everyone contributed equally to the overall effort. This collaborative approach ensured that all aspects of the project were thoroughly addressed without any significant issues or imbalances in workload.

References

[1] D. Ghosh, R. Sharman, H. Rao, and S. Upadhyaya, "Self-healing systems - survey and synthesis," Decision Support Systems, vol. 42, no. 4, pp. 2164-2185, 2007.

[2] J. Park, G. Yoo, C. Jeong, and E. Lee, "Self-management system based on self-healing mechanism," in Proceedings of the International Conference on Software Engineering Research, Management and Applications (SERA'06), Seattle, WA, USA, Aug. 2006, pp. 372-382.

[3] S. Ghahremani, H. Giese, and T. Vogel, "Efficient utility-driven self-healing employing adaptation rules for large dynamic architectures," in Proceedings of the IEEE International Conference on Autonomic Computing (ICAC), Columbus, OH, USA, Jul. 2017, pp. 59-68.

[4] G. A. Moreno, "Adaptation timing in self-adaptive systems," Carnegie Mellon University Technical Report, 2017.

[5] D. Ratasich, T. Preindl, K. Selyunin, and R. Grosu, "Self-healing by property-guided structural adaptation," in Proceedings of the IEEE Industrial Cyber-Physical Systems (ICPS), St. Petersburg, Russia, May 2018, pp. 199-205.

[6] B. Magableh and M. Almiani, "A self-healing microservices architecture: A case study in Docker Swarm Cluster," in Proceedings of the Advances in Intelligent Systems and Computing (AISC), 2019, pp. 846–858.

[7] G. A. Moreno, J. Cámara, D. Garlan, and B. Schmerl, "Proactive self-adaptation under uncertainty: A probabilistic model checking approach," in Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), Bergamo, Italy, 2015, pp. 1-12.

[8] C. Stevens and H. Bagheri, "Reducing runtime adaptation space via analysis of possible utility bounds," 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), Seoul, South Korea, 2020, pp. 1522-1534.

[9] X. Dai and J. Xiang, "Consequence-oriented self-healing and autonomous recovery for cloud-based systems," Proceedings of the International Symposium on Software Reliability Engineering (ISSRE), Hiroshima, Japan, 2011, pp. 1–10.

[10] S. VanSyckel, D. Schäfer, G. Schiele, and C. Becker, "Configuration management for proactive adaptation in pervasive environments," 2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems (SASO), Philadelphia, PA, USA, 2013, pp. 131–140.

[11] V. Riccio, G. Sorrentino, E. Zamponi, M. Camilli, R. Mirandola, and P. Scandurra, "RAMSES: an artifact exemplar for engineering self-adaptive microservice applications," 2024 IEEE/ACM 19th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2024, pp. 161–167.

[12] P. Arcaini, E. Riccobene, and P. Scandurra, "Formal Design and Verification of Self-Adaptive Systems with Decentralized Control," ACM Transactions on Autonomous and Adaptive Systems (TAAS), vol. 11, pp. 1–35, 2017.

[13] A. Qasim and S. A. R. Kazim, "MAPE-K Interfaces for Formal Modeling of Real-Time Self-Adaptive Multi-Agent Systems," IEEE Access, vol. 4, pp. 4946–4958, 2016.

[14] J. Cleland-Huang, A. Agrawal, M. Vierhauser, M. Murphy, and M. Prieto, "Extending MAPE-K to Support Human-Machine Teaming," 2022 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2022, pp. 120–131.

[15] Ibrahim, Adeoye & Heart, Slyvester. (2020). Self-Healing Techniques in API and Microservices Frameworks.

[16] Harshal Shah, Jay Patel. (2024). Self-Healing AI: Leveraging Cloud Computing for Autonomous Software Recovery. In revista.redc (Vol. 16, Number 4, pp. 180–200). revista.redc. https://doi.org/10.5281/zenodo.14219873

[17] Garlan, David & Schmerl, Bradley & Cheng, Shang-Wen. (2009). Software Architecture-Based Self-Adaptation. 10.1007/978-0-387-89828-5_2.

[18] Georgiadis, Ioannis & Magee, Jeff & Kramer, Jeff. (2002). Self-organising software architectures for distributed systems. 33. 10.1145/582129.582135.

[19] Dashofy, Eric & van der Hoek, Andre & Taylor, Richard. (2002). Towards architecture-based self-healing systems. 21. 10.1145/582129.582133.

[20] Gorlick, Michael & Taylor, Richard & Heimbigner, Dennis & Johnson, Gregory & Medvidovic, Nenad. An Architecture-Based Approach to Self-Adaptive.

[21] Sztajnberg, Alexandre & Loques, Orlando. Describing and deploying self-adaptive applications.

[22] Ghosh, Debanjan & Sharman, Raj & Rao, Raghav & Upadhyaya, Shambhu. (2007). Self-healing systems — survey and synthesis. Decision Support Systems. 42. 2164-2185. 10.1016/j.dss.2006.06.011.

[23] Garlan, David & Cheng, Shang-Wen & Schmerl, Bradley. (2002). Increasing System Dependability through Architecture-Based Self-Repair. Architecting Dependable Systems. 2677. 61-89. 10.1007/3-540-45177-3_3.

[24] Georgiadis, Ioannis & Magee, Jeff & Kramer, Jeff. (2002). Self-Organising Software Architectures for Distributed Systems. Proceedings of the first ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02). 33-38. 10.1145/582128.582135.

[25] Bradbury, Jeremy & Cordy, James & Dingel, Juergen & Wermelinger, Michel. (2004). A survey of self-management in dynamic software architecture specifications. Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering. 10.1145/1075405.1075411.