Information and Software Technology 174 (2024) 107519

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Towards antifragility of cloud systems: An adaptive chaos driven framework

Joseph S. Botros[a], Lamis F. Al-Qora'n[a], Amro Al-Said Ahmad[b,*]

[a] Department of Software Engineering, Faculty of Information Technology, Philadelphia University, Amman, Jordan
[b] School of Computer Science and Mathematics, Keele University, Staffordshire, United Kingdom

ARTICLE INFO

Keywords:
Antifragility
Resilience
Chaos engineering
Self-adaptive software
Resilience testing
Cloud computing

ABSTRACT

Context: Unlike resilience, antifragility describes systems that get stronger rather than weaker under stress and chaos. Antifragile systems have the capacity to overcome stressors and come out stronger, whereas resilient systems are focused on their capacity to return to their previous state following a failure. As technology environments become increasingly complex, there is a great need for developing software systems that can benefit from failures while continuously improving. Most applications nowadays operate in cloud environments. Thus, with this increasing adoption of Cloud-Native Systems they require antifragility due to their distributed nature.
Objective: The paper proposes UNFRAGILE framework, which facilitates the transformation of existing systems into antifragile systems. The framework employs chaos engineering to introduce failures incrementally and assess the system's response under such perturbation and improves the quality of system response by removing fragilities and introducing adaptive fault tolerance strategies.
Method: The UNFRAGILE framework's feasibility has been validated by applying it to a cloud-native using a real-world architecture to enhance its antifragility towards long outbound service latencies. The empirical investigation of fragility is undertaken, and the results show how chaos affects application performance metrics and causes disturbances in them. To deal with chaotic network latency, an adaptation phase is put into effect.
Results: The findings indicate that the steady stage's behaviour is like the antifragile stage's behaviour. This suggests that the system could self-stabilise during the chaos without the need to define a static configuration after determining from the context of the environment that the dependent system was experiencing difficulties.
Conclusion: Overall, this paper contributes to ongoing efforts to develop antifragile software capable of adapting to the rapidly changing complex environment. Overall, the research provides an operational framework for engineering software systems that learn and improve through exposure to failures rather than just surviving them.

1. Introduction

According to Taleb [1], an antifragile is defined as a system that gets stronger under stress, whereas antifragile systems, in contrast to robust systems, are able to adapt to changes in their environment and learn from earlier failures, reducing the effect of future failures and strengthening themselves over time. In other words, systems that experience degradation in their performance as a result of being subjected to uncertainty are classified as fragile, whereas systems that are classified as antifragile will thrive, flourish, grow, and profit from exposure to risk, uncertainty, randomness, and disorder [2]. An example of an antifragile system is the human immune system, which gets stronger with repeated exposure to germs [3]. Thus, Antifragility as property refers to a desirable characteristic that focuses on how a system or entity responds to stress or shocks. Unlike fragility, where an entity becomes more vulnerable to harm under stress, antifragility involves a quality of response that is not only resilient but actually benefits from such challenges. In essence, when faced with adversity, an antifragile system or entity reacts in a way that not only mitigates harm but also gains advantages or improvements in architecture and maturity. This quality can be visualised as a convex shape on a graph, while fragility is a concave function (more pain than gain[4]). Consequently, Antifragile design is becoming a growing field of study in software engineering [4, 6]. In a recent study by Grassi et al. [7,8], proposed conceptualising

* Corresponding author at: CR35, Colin Reeves Building, Keele University, Staffordshire, ST5 5BG, United Kingdom.
E-mail address: a.m.al-said.ahmad@keele.ac.uk (A. Al-Said Ahmad).

https://doi.org/10.1016/j.infsof.2024.107519
Received 18 November 2023; Received in revised form 11 June 2024; Accepted 20 June 2024
Available online 22 June 2024

antifragility as an extension of the "dependability" quality attributes in software systems.

When a fault or malfunction occurs in one component of a fragile system, it can propagate to other components, causing the system's failure, where the system is incapable of recovering. A resilient system, on the other hand, makes an effort to promptly and effectively recover from each failure event and stop faults from impacting other system components. However, antifragile systems are intended to grow stronger and benefit from such failure, whereas resilient systems are not expected to bounce back stronger after failure.

When we discuss a system's robustness or antifragility, we are talking about how it responds to an increase in stressors over a range of values until it reaches a particular stress threshold, after which the system may start to become more fragile. We should design and create the software to make antifragile software instead of investigating how the system responds to all kinds of incidents. The antifragile software will try to embrace uncertain environments by embedding adaptive and fault-tolerant methods in their architecture, and continuously simulating stressful situations that uncover fragilities. In Fig. 1, these types of systems are compared.

Cloud Computing is considered the dominant computing platform nowadays [10]. Most applications operate in cloud-based environments, whether public, private, or hybrid, because of the on-demand availability of computing resources and the ease of provisioning and delivering services to customers through different cloud service models. Systems that are provisioned within cloud environments acquire the adaptive behaviours of their environments by necessity.

Complex adaptive systems (CAS) are networks of several agents that collaborate, interact, and get feedback from the environment in which they operate. As a result, system behaviour becomes emergent rather than predetermined [11] which allows a system to react and modify according to the influence of its previous activity. Modern cloud systems can also be considered Complex Adaptive Systems (CAS) since they are composed of many independent agents competing over computing resources. Predicting the behaviour of a complex adaptive system in all circumstances is considered a complicated problem, it is more efficient to build an adaptive system that can gain benefit from unforeseen circumstances, or more specifically to build an antifragile system.

"Cloud-native" is a term that is used to describe any system designed to best utilize or apply cloud features [12]. Because of their inherent resilience, the ease with which microservices may be deployed and removed, the flexibility to execute chaos experiments to test system responses, and their built-in redundancy, which enhances dependability, cloud-native Systems are more likely to become antifragile [13]. Cloud-native applications consist of multiple small, interdependent services called containers, that are easy to deploy and remove from the cloud. Using Docker containers to bundle and deliver cloud-native applications is becoming more and more popular [14]. These containers can interact synchronously and asynchronously through messaging buses. Such distributed architecture also can have many emergent properties because of the inherent self-adaptation and self-organisation capabilities of cloud-native systems. Thus, it is faced with the same complexity in predicting the behaviour of the CAS systems that we've mentioned before. A crucial novel approach to developing cloud-native apps is serverless computing, which is built on technologies like AWS Lambda, Azure, and Google Functions [15].

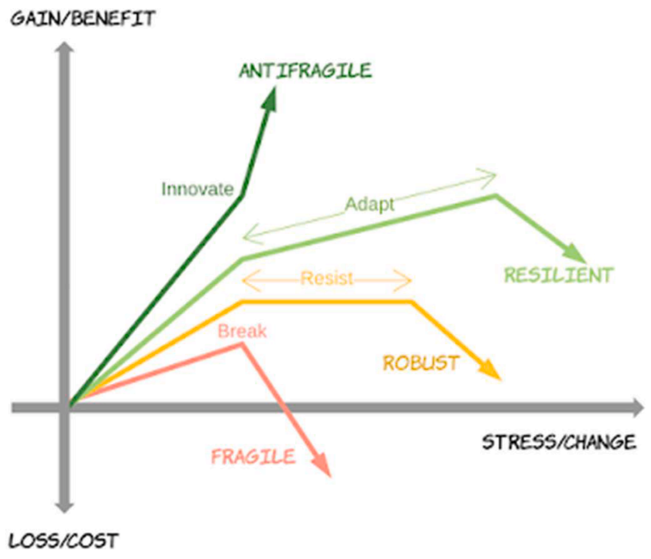


Fig. 1. Comparison of different quality of responses to stressors [9].

The advent of cloud computing and the reliance on its services to build heterogeneous distributed systems, which span multiple services that may originate from various cloud providers [16], suggests enhancing the resilience and robustness of such software systems to build resilient applications that can withstand services/faults disruption. Chaos engineering involves intentionally introducing controlled failures and disruptions into a system to proactively identify weaknesses and improve overall resilience [13]. This is important because cloud computing settings require resilience due to the dynamic and distributed nature of these systems. Software antifragility and chaos engineering are two concepts that can be connected to enhance cloud applications' resilience and benefit from shocks, volatility, and stressors, becoming more robust, responsive to unpredictable events, and more resilient as a result. In other words, antifragility adds chaos to a system so that it will react to become stronger rather than break down [13].

Resilience refers to a system's ability to recover from failures and restore itself to its original state [17]. Cloud computing offers several strategies and techniques like auto-scaling, disaster recovery, and auto-healing. Additionally, it provides fault tolerance techniques, such as load-balancing, that support the resiliency of cloud applications [18]. Despite these valuable services and strategies, we still use chaos engineering methods to test and enhance resiliency. Antifragility, on the other hand, is the ability of a system to learn from failures and become stronger in the face of future challenges. To surpass resilience, it is recommended to design systems that can adapt to changing environments and learn from incidents. This involves developing systems that can detect and respond to failures, as well as systems that can learn from those failures and adjust their behaviour accordingly. By doing so, systems can become more robust and better equipped to handle future challenges [3]. This paper is inspired by a scarcity of studies that propose an applicable architecture for building antifragile software [3,4,7]. Even though various antifragility principles may be found in literature, few practitioners have adopted them. This is regardless of the fact that they all share the same core idea of going beyond resilience and benefiting from stressors. We found that the lack of practical, systematic architectures that guide the process of upgrading current software into antifragile ones was the reason for this lack of adoption. We are using adaptive approaches to develop an operational framework that surpasses existing strategies of robustness and resilience and moves towards antifragility. Adaptive strategies can help to accomplish antifragility as they involve learning and improving from stressors and challenges [19]. In other words, cloud-native apps can accomplish antifragility by utilising a variety of adaptive mechanisms that help the system perform well in the face of change and uncertainty.

Our goal is to provide a practical demonstration of the integration of antifragility principles, such as adaptivity, into an overall architecture that includes all necessary supportive components and the process for helping existing systems achieve operational antifragility.

2. Research questions (RQs)

Our paper is guided by the following two key research questions:

RQ1: What are the key mechanisms and strategies that can enable automated operational antifragility in real-world cloud applications?

RQ2: How does the implementation of the UNFRAGILE framework enhance the antifragility of a cloud-native application?

To answer the research questions, the paper first examines the concepts and design principles of building antifragile software systems. Then, a thorough examination of the relevant research and the most recent best practices in the field is used to develop a UNFRAGILE framework that is intended to achieve automated operational antifragility in the face of challenges and failures, which answers the first research question. The UNFRAGILE framework invests adaptivity principles and employs chaos engineering as an effective methodology for testing and enhancing the stability of software systems by gradually introducing faults into systems in production environments to uncover fragilities. To answer the second research question, UNFRAGILE is then evaluated with a proof-of-concept application to showcase how the UNFRAGILE framework can be incorporated and utilised against a common real-world scenario, namely outbound latency in distributed cloud systems. Finally, the key ideas and contributions are summarised, and future work is presented. Overall, the paper contributes to the ongoing efforts to develop more resilient and antifragile software systems capable of withstanding the challenges and complexity of the software environment.

This paper is organised as follows: Section 3 describes the research background and key related work; Section 4 presents the UNFRAGILE architectural framework; Section 5 presents a case study on using a cloud-native system; Section 6 discusses the results the analysis; and finally, Section 7 concludes the paper and provides limitations and future work.

3. Research background and related work

3.1. Antifragility principles and system architecture

To build antifragile software systems, thinking must undergo a challenging change in perspective that embraces shocks as drivers of progress rather than just resilience. Although incorporating risk-seeking activities may assist in identifying system fragilities, genuine antifragility necessitates incorporating behaviours that let the system bounce back and adapt, gradually improving with each stressor. Many different principles and reference architectures in the antifragility literature and state-of-the-art can be utilised to harness antifragility.

Tolk [20] introduced the concept of constructing antifragile systems using agent-based systems engineering. He suggested employing the agent metaphor to create systems that continuously adapt to changing environments, providing new functionality as needed. By merging model-based systems engineering with the agent metaphor and utilising utility functions, the goal is to develop systems that not only survive but also improve under stress and in dynamic contexts. Their study presents a first step towards enabling systems engineering of antifragile systems. Although it acknowledges that more studies are required to achieve this goal, it provides evidence that such systems are capable of being designed.

Russo and Ciancarini [5] proposed a manifesto that is inspired by the Agile manifesto, consisting of 12 guiding principles to encourage practitioners to develop antifragile systems. These principles focus on client satisfaction and embrace changing scenarios. Russo and Ciancarini [4] followed up with a novel software architecture that attempted to accomplish their proposed principles. Such an architecture suggests a dynamic technique in which the system learns from errors and keeps getting better all the time. According to their paper, static fault tolerance alone is insufficient to be considered antifragile because experiencing more problems offers no benefit. Thus, they also suggested using fine-grained structures like microservices for flexibility and continuous improvement and approaching antifragility through embedding adaptive fault tolerance solutions. It also recommends fault injection approaches, test-driven development (TDD), and DevOps as methodologies to support antifragility.

To build antifragile software, Monperrus [6] outlined several principles, including a) Fault tolerance, which exposes the system to faults and embeds adaptive and recovery strategies; b) Automatic runtime repair, which modifies the system automatically at runtime in response to errors and bugs; and c) Failure injection in production. The paper also outlined another set of principles that can improve the software development process and harness antifragility. Monperrus [21] followed up with an envisioned software model that deals with unexpected failure in production systems by designing and building systems that can learn from its failure. by building fault-tolerant systems that are constantly subjected to perturbations and actively learning from its mistakes how to improve its behaviour and performance. His proposed architecture consists of a) a Monitoring module, b) a Perturbation module, and c) a Recovery module.

Hole [9] also made significant contributions by synthesising antifragility concepts with market pioneer systems such as Netflix for building highly available software. Several principles were proposed to ensure the system's antifragility. These principles for ensuring anti-fragility in complex information and communications technology (ICT) systems can be summarised as follows: modularity, which modularises the system to isolate local failures and reduce their impact on the entire system; weak links, which create weak links that break to prevent propagating failures; incorporate redundancy by deploying multiple copies of modules to increase system robustness; diversity which introduces diversity by including modules with different designs or implementations to increase system resilience. Together, these principles are meant to develop ICT systems that are resistant to disruptions and severe global behaviour that is a byproduct of being a CAS. Hole questioned whether the five principles are enough for ensuring antifragility to any particular class of impacts, and he stated that the answer to this question was not entirely recognised when he was writing his book in late 2015. The principles required for designing and operating an anti-fragile system will likely vary depending on the kind of system and impact class under consideration. Next, Hole argued that the five principles offer antifragility against malware propagation and downtime. To ascertain whether more principles are needed, more research needs to be conducted.

Subsequently, a tutorial article by Hole [3] explored the architecture and functionality of downtime-resistant software systems, leading to antifragile distributed systems. The tutorial examined four design principles and two operational principles, highlighting their significance and the relationship between them. The software system should consist of separate, isolatable processes with sufficient redundancy and diversity as per design principles. The principles of operation dictate that engineering teams should intentionally cause failures in production systems to learn from them and improve, thus minimizing downtime. A case study that demonstrates the application of both the anti-principles and the principles is also included in the paper. It also introduces three design choices and one operational choice gained by applying the principles to distributed software systems of separate processes. Finally, the author examines when and how to create socio-technical systems that are resistant to downtime and emphasises the significance of using the same principles for other types of antifragility and the importance of developing more concepts that could lead to achieving antifragility.

Jones [22] discussed the need to shift from designing fragile systems to antifragile systems, particularly within the context of NASA's engineering practices. The paper argues that the traditional methods of designing systems, which focus on meeting specific known requirements, inherently create fragile systems. These systems are prone to failure when subjected to conditions beyond their designed specifications. Also, it attributes software fragility to the traditional design approach known as Reductionism. This approach assumes that any system, no matter how complex, can be fully understood by breaking it down into its individual components. The paper argues that this precisely leads to building systems that fail to handle any future unexpected conditions effectively because they are designed to meet predefined requirements: The paper advocates for a change in design philosophy inspired by concepts from Complexity Science and Nassim Taleb's idea of antifragility, the paper also presents many principles that are specific to aviation technologies, including communal sensor networks for adaptive noise control, morphing wings for optimal flight performance, autonomous systems learning to fly, swarming of autonomous units for mission resilience, integrated vehicle health management for proactive maintenance, and self-healing materials that repair and strengthen under stress. These principles aim to create systems that adapt, learn, and thrive in unpredictable environments, which is aligned with the paper's vision for engineering antifragile systems

To conceptually define antifragility, Grassi et al. [8] proposed including antifragility as a new attribute under the dependability umbrella, which is a software quality attribute) taxonomy. In addition to the existing categories (fault prevention, fault tolerance, fault removal, and fault forecasting), the paper suggests adding a new category called "change triggering and exploitation." This category includes strategies aimed at encouraging and leveraging knowledge changes to evolve the system into an improved version or improved state by implementing a "virtuous chain" of transformations that enhance system quality. Thus, a system can be defined as “antifragile” if it can implement that virtuous chain of continuous change and improving, thereby continuously improving in response to changes and disturbances. This new attribute recognizes the system's ability to get better when exposed to stressors, going beyond resilience or robustness. The paper does not provide a concrete reference architecture or implementation for antifragile systems and considers it an open challenge and research questions towards defining a reference architectural model for antifragile systems. Specifically, it highlights the need to identify suitable architectural elements, methodologies for antifragility assessment, and suitable antifragility metrics. This presented a challenge that inspired us to create a detailed, well-organized architecture.

3.2. Chaos engineering and fault injection

The rationale of utilising chaos engineering to assess system dependability and maturity is not new in literature. It has been established that fault injection in production uncovers fragilities and transitions from the mindset of avoiding failure to the mindset of embracing faults for the sake of producing more robust and antifragile systems. it is summarised in the following quote [23]:

> “It's better to prepare for failures in production and cause them to happen while we are watching instead of relying on a strategy of hoping the system will behave correctly when we aren't watching.”

Many researchers utilised chaos engineering to assess the dependability of systems. For example, Malik et Al. [24] proposed a framework *CHESS* for the systematic evaluation of self-adaptive and self-healing capabilities of systems by using chaos engineering to evaluate system response during chaos and whether it reflects any self-adaptation by targeting the system with failure scenarios that affect system quality attributes (Availability, Reliability, Integrity, Performance). Moreover, Pierce et al. [25] applied chaos experimentation (specifically network degradation) and automatic fault injection to applications running in middleware systems. The experiments aimed to understand how the system responds to specific faults and network conditions, providing valuable insights into system operation and actionable strategies for enhancing resilience. The research emphasises the importance of applying Chaos Engineering to open systems architecture, particularly in the context of military mission systems.

Meiklejohn et Al. [26] presented SFIT and Filibuster as tools for identifying resilience issues in microservice applications. By combining static analysis and concolic-style execution, SFIT enhances test suites to cover failure scenarios. The authors emphasise the growing adoption of chaos engineering to identify issues related to partial failure. Filibuster demonstrates its effectiveness by detecting bugs in real-world microservice applications, offering an opportunity to address these issues early in the development process and improve application resilience.

Al-Said Ahmad and Andras [27] applied fault injection using the Application-Level Fault Injection (ALFI) technique to evaluate the scalability resilience of cloud-based software services. They conducted experiments on a real-world cloud-based software service running on the EC2 cloud and simulated delay latency faults. By comparing the results of the fault scenarios with baseline data, they assessed the impact of the injected faults on scalability resilience. Simulating network delay chaos was used to assess the scalability of the software.

Simonsson et Al. [28] built a novel fault-injection system called *ChaosOrca* that operates on the operating system level. The system aims at evaluating applications' self-protection through manipulating system calls and injecting fault in them, and the framework targets cloud-native systems that consist of containerised docker applications. The paper utilises chaos engineering as a main method for detecting fragility and verifying the quality of response for systems, whether it was fragile, robust, or antifragile. This is done by building chaos experiments that resemble real-life system stress. The chaos engineering process is followed as found in literature and state-of-art, and it is utilised as a part of our antifragility holistic framework.

In his master's thesis, KOSTENKO [29] investigated the applications of resilience and antifragility in microservices architecture. Chaos engineering was implemented to generate various chaotic assaults on applications. Following an examination of industry and state-of-the-art tools, the KOSTENKO framework was proposed with four components: A chaos toolkit, a load generator, a hypothesis validator, and a dashboard. A Spring Java web application was used to evaluate the framework's various resilience strategies (Timeout, Retry, Circuit breaker). The pattern employed for this work is categorised as a resilience pattern, and it significantly helps in the development of robust and resilient applications. However, when it comes to antifragility, it can be found that developing antifragile systems is still limited due to the lack of adaptivity in some patterns and their lack of a "learning from errors" dimension [30]. Thus, building a systematic framework around Antifragility is still required [4]. To address such limitations in the framework, we have introduced an evaluation matrix to evaluate whether an adaptation strategy can be considered antifragile according to its impact on the system's operational performance. We have also developed a case study by implementing an adaptation strategy that exceeds resilience and makes our case study system antifragile against one of the most common network failures.

3.3. Adaptive concurrency

Architectural complexity can lead to a number of unpredicted failures that may occur during runtime in distributed and microservices systems [31]. Examples of such failures include [31–33]: (a) Cascading failures, (b) Retry storm (backpressure), (c) Death spiral, and (d) metastable failure. Antifragility requires having a dynamic adaptive approach. Thus, when we are implementing antifragile software, we are aiming at embedding adaptive and dynamic approaches based on system context awareness (local strategies) and system-wide awareness (global strategies).

Shahid et al. [34] examined recently developed fault tolerance strategies for cloud computing and classified them into three groups: Reactive Methods, Proactive Methods, and resilient solutions. They stated that reactive methods let the system get into a defective condition, but they then try to back up the device. Proactive Methods assist in preventing the device from getting into a defective condition by incorporating actions that reduce defects before they impact the device. Newly developed resilient methods aim at reducing the time it takes for a device to detect a fault. Resilience strategies include circuit breaker, timeouts, load-shedding, and graceful degradation). The real problem, however, with such fault tolerance strategies is that they're inherently fragile [4] because they embed a fault model as an assumption.

Liu et al. [35] recommended a concurrency-aware system scaling framework that adapts to changing application workloads by guessing the optimal concurrency configuration and running actuators that perform autoscaling for cloud resources. Considering the relationship between concurrency and throughput, such concurrency-aware mechanisms are proven to be more reliable than static metrics rule-based approaches.

Brogi et al. [36] proposed a methodology for self-healing trans-cloud applications, which are complex systems deployed across multiple cloud providers and service layers. The methodology focuses on reducing the time that application components depend on faulty services, thereby minimising unstable states and the potential for cascading failures. It also considers the interdependencies between components during the recovery process. Their work also includes a prototype implementation, which demonstrates the effectiveness of adaptive methodologies protecting cloud systems from both application failures and cascading cloud service failures.

Zoghi et al. [37], discussed the design of adaptive applications that are deployed in cloud environments. To satisfy the adaptation goals, the authors suggested a search-based algorithm to optimise the deployment of applications. These goals include minimising resource costs and achieving a response time of less than 500 ms. The experiments measured the response time, and the number of iterations needed to attain the adaptation goals.

In conclusion, the antifragility of cloud services can be enhanced by an adaptive concurrency-aware concurrency configuration strategy that adjusts the maximum number of concurrent requests allowed to an outbound service in accordance with workload criticality and performance metrics. This can prevent overwhelming the system or its dependencies, producing failures such as cascading failures and death spirals.

4. UNFRAGILE architectural framework: a systematic improvement to system's response to stressors and failures

This paper proposes and evaluates the UNFRAGILE as an architectural framework. UNFRAGILE aims at developing antifragile systems through an iterative cycle of experiments, analysis, and improvements. UNFRAGILE is inspired by the system models proposed by [3,21,38], for engineering antifragile systems. We developed UNFRAGILE as shown in Fig. 2 with three modules (Chaos Module, Adaptation Module, and Monitoring Module). The purpose is to extend the existing cloud-native systems with supportive modules to enhance the software's antifragility towards chaotic events.

The components of the proposed UNFRAGILE framework and their interrelationships are visualised in Fig. 2 which shows a high-level software architecture diagram.

Fig. 3, shown below, defines our envisioned structures for implementing the UNFRAGILE framework for cloud-native systems, this architecture will be integrated with the system inside the docker container. Extending the systems with the UNFRAGILE framework components within a cloud-native ecosystem through utilising Docker containers is crucial for establishing a continuously evolving software that is approaching antifragility. This integration ensures a seamless and effective system that can handle any challenges that arise by constantly simulating chaotic situations, monitoring application behaviour, and adjusting it to enhance its adaptation. This is important because enhanced user experience and sustainable business operations depend on the capacity to adapt to the changing demands and complexities of modern IT infrastructures.

The following subsections explain the three modules, as shown in Fig. 3.

4.1. *Chaos module*

Fig. 4 demonstrates chaos module overall process. Chaos module is responsible for managing environment manipulation and fault injection within the production environment. This can be tailored to fit the requirements of any perturbation experiment, or it can be combined with an off-the-shelf tool that is currently on the market. The steps involved in planning and executing a chaos attack are depicted in Fig. 4. The first step in the procedure involves choosing a stressor or defect to be tested, such as unexpected instance termination, network perturbations, resource stress-testing, or simulating unexpected software behaviour, in order to find out whether the system is capable of withstanding in an antifragile manner.

In accordance with the chaotic experiment's design, we monitor system metrics, including the system's steady state. We also formulate a hypothesis, identify the blast radius, and, last, outline the parameters of the experiment's duration and scope. Subsequently, the experiment is carried out, and the prescribed step-by-step perturbation is started. Following analysis of the data, observations are made, where monitoring modules and custom dashboards are utilised to view the system before, during, and after chaos.

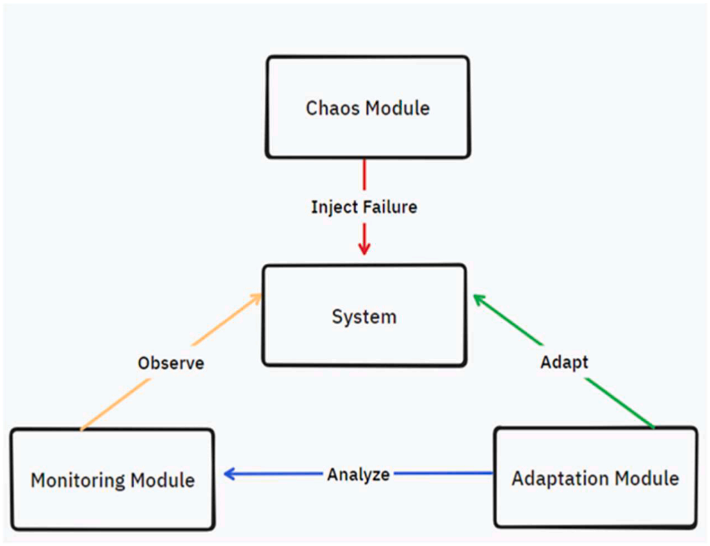


Fig. 2. UNFRAGILE framework overall architecture.

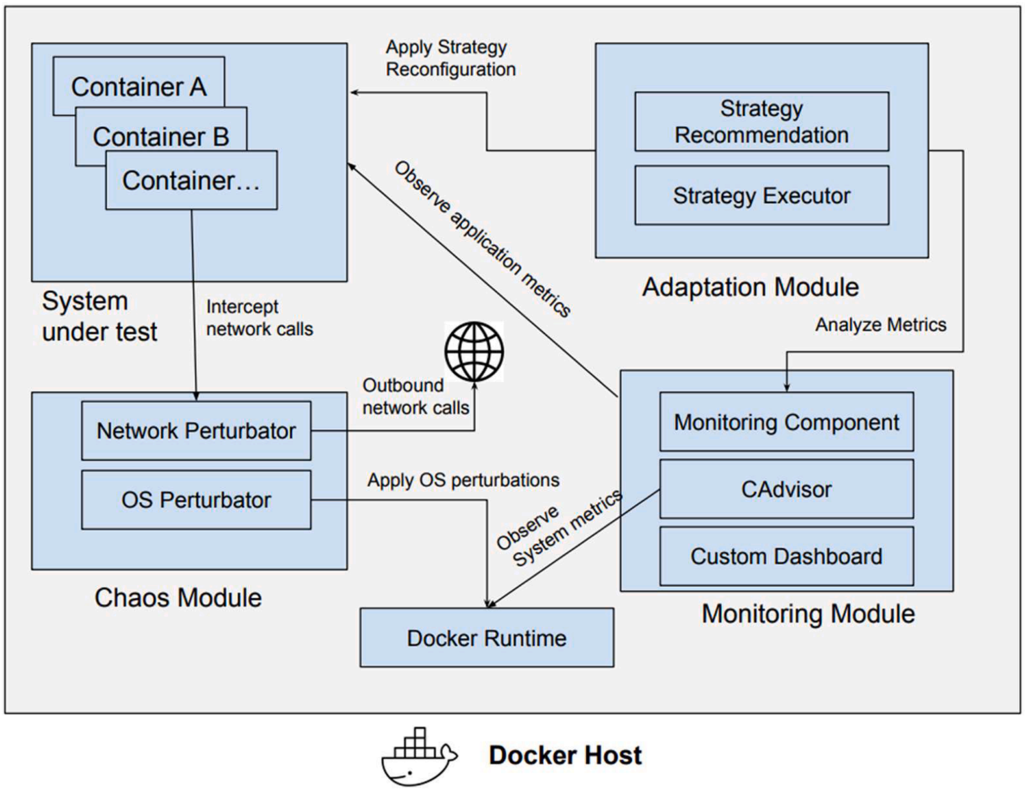


Fig. 3. UNFRAGILE framework detailed architecture.

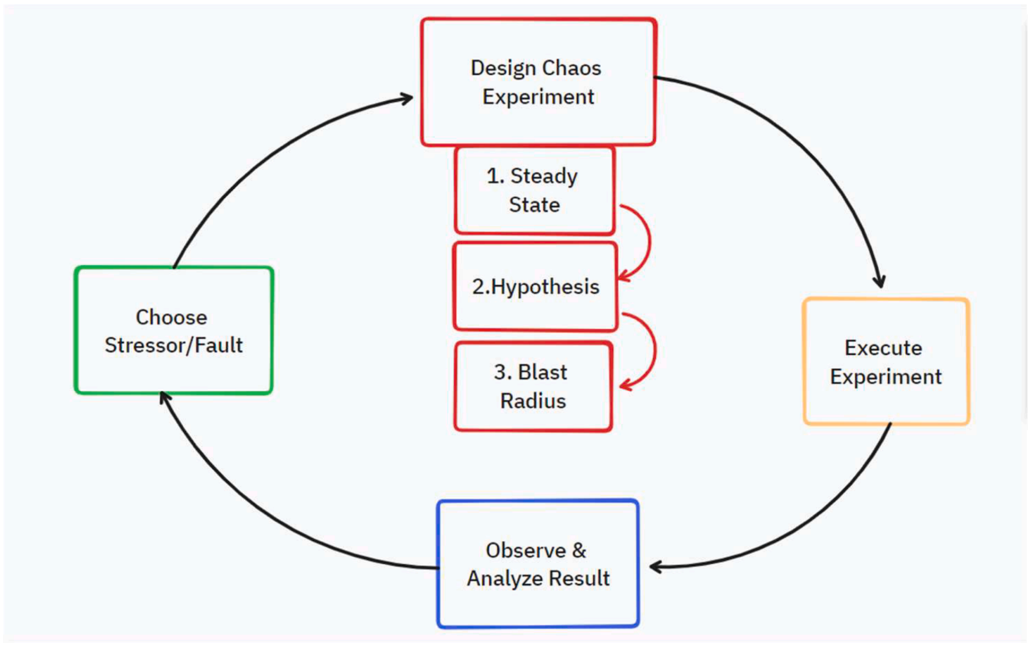


Fig. 4. UNFRAGILE framework - chaos module overall process.

4.2. *Monitoring module*

Application performance monitoring (APM) is the practice of tracking software performance metrics using monitoring tools and telemetry databases for better monitoring management [39]. APM is an essential aspect of systems observability & instrumentation [40]. It helps developers and operations teams to identify and resolve performance issues in real-time. APM tools such as Prometheus [41] provide visibility into the performance of applications, infrastructure, and networks. In our UNFRAGILE framework, system's observability is a necessary condition when evaluating the robustness, antifragility, and quality of system response under perturbations experiments. Furthermore, the observability of system components through the use of APM tools helps in identifying and eliminating the fragilities underlying causes.

Many standard application metrics such as response time, error rate, and throughput, as well as system metrics like CPU usage, memory usage, and network latency, need to be monitored for cloud systems. Some of these standard metrics are included by default in the majority of APM tools, and more customised metrics can be added based on system requirements. It may be possible to gain an understanding of the system's behaviour, health, and performance before, during, and after the experiments by observing these metrics over time.

To ensure the success of experiments, custom monitoring dashboards are highly helpful in chaos engineering. These dashboards enable real-time monitoring of critical experiment performance metrics during chaotic experiments, enabling engineers to analyse system behaviour and implement modifications as needed. They also contribute significantly to post-experiment analysis by collecting and presenting important metrics that enable a full understanding of the results. Furthermore, specialised dashboards allow non-technical stakeholders to make educated decisions based on empirical data by facilitating effective communication of chaos engineering outcomes.

Since the primary goal of UNFRAGILE framework is to make any system that already exists more antifragile, we know that introducing observability to already-existing systems presents significant challenges because of the need for code changes in several places to add monitoring and instrumentation boilerplate code. As a result, we recommend utilising contemporary APM features for cloud-native systems, such as auto-instrumentation [42,43], which can reduce the amount of code changes required in order to add monitoring to already-existing systems. In the context of cloud native systems, it provides additional functionality, such as logging and monitoring, without the need for disruptive code changes. Moreover, auto-discovery is the process of automatically discovering the application component to monitor without the need for manually adding it to the central configuration. This is also supported by modern cloud-native ecosystems by adding monitoring to a system with a small number of components by hand could be simple initially, but as systems get bigger and more complex, the task becomes more difficult and similar to attempting to find a needle in a haystack. The sheer number of moving parts and metrics makes it challenging to effectively correlate and pinpoint issues. A combination of these strategies can make the monitoring module which is required by the UNFRAGILE, be added in a plug-and-play manner.

4.3. *Adaptation module*

Fig. 5 below shows the overall process of the adaptive module.

4.3.1. *Adaptation process description*

The adaptation process is the process of detecting and eliminating fragility in the system while building self-adaptivity and antifragility. The adaptation module, as shown in Fig. 5, represents a crucial phase where the software undergoes either runtime self-adaptation or source code modifications based on the findings from the chaos experiment. The first step is to detect the fragility, which is done through monitoring and analytic tools that provide insights into the system's behaviour during the chaos experiment. Once the fragility is identified, a root cause analysis is conducted to determine the underlying fault responsible for the observed fragility. This analysis helps in understanding the weaknesses and vulnerabilities in the system. With a clear understanding of the root cause. The next step is to devise a recovery and mitigation strategy, this involves implementing incremental changes to the software to address the identified fault and make the system more resilient. The goal is to build adaptivity and antifragility into the system, enabling it to not only withstand chaos but also benefit from it by becoming more robust. The adaptation module is a continuous process that ensures the system evolves and adapts to the lessons learned from the chaos experiment, ultimately improving its overall stability and performance.

The overall process shown in Fig. 5 involves: (1) Observing the experiment results by collecting metrics from various observability tools and persisting it in time-series database for custom dashboard queries; (2) Analysing the system's response to chaos and categorising it as fragile, robust, or antifragile based on the experiment results; (3) Implementing strategies to address fragilities if the system is found to be fragile, including conducting a root cause analysis to identify bottlenecks and implementing mitigation strategies to achieve adaptive fault tolerance and antifragility; (4) Retrying the experiment after implementing the mitigation strategies to assess whether the fragility has been resolved. This iterative approach aims to enhance the system's resilience and adaptivity, ultimately moving towards an antifragile state.

4.3.2. *Static fault tolerance vs adaptive fault tolerance*

As we previously discussed, fault tolerance is unavoidable for with the increasing complexity of modern software, is emergent from

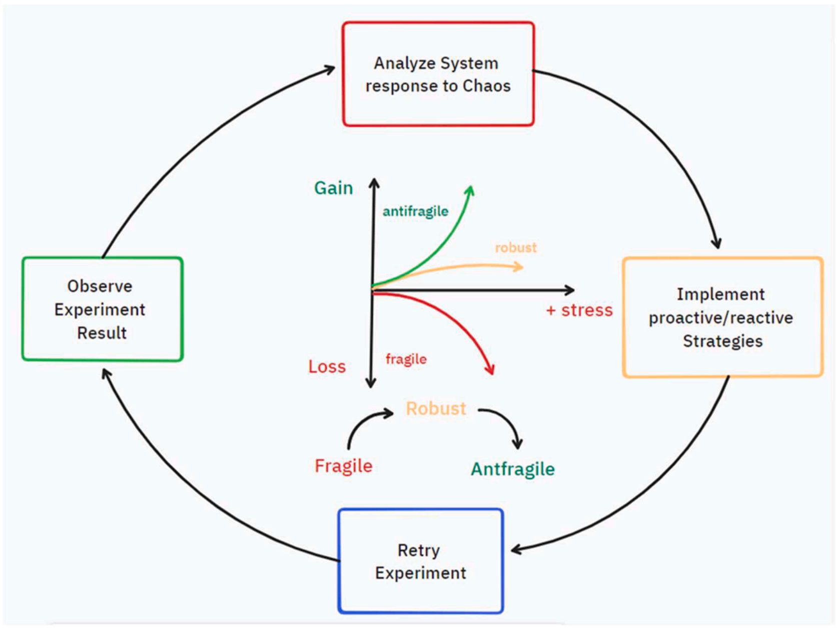

Analyze System response to Chaos
Implement proactive/reactive Strategies
Retry Experiment
Observe Experiment Result
Gain
antifragile
robust
+ stress
Loss
fragile
Robust
Fragile
Antfragile

Fig. 5. UNFRAGILE framework - adaptive module overall process.

increasing complexity in business domains, and emergent complexity from the nature of distributed systems, fault tolerance is an inevitable requirement for surviving unseen chaos and failure, but statically embedded fault tolerance is intrinsically fragile as it embeds a fault model as an assumption [4], according to Monperrus [6] a system with dynamic and adaptive fault tolerance capabilities is antifragile: that is, when exposed to faults, it continuously improves and becomes more mature than static fault tolerance, which can fail with changing environmental conditions [44]. Thus, we propose evaluating the maturity of the chosen fault tolerance strategy according to the matrix in the following Fig. 6:

4.3.3. Enhancing static fault tolerance strategies to become adaptive

Antifragile software is defined in the paper as a system that is capable of thriving under stress and adapting to chaotic environments. Although static fault-tolerant strategies are useful, they are still fragile [4], since they do not respond properly to the changing environment and stress by exploiting them. So, the paper recommends enhancing fault-tolerance strategies by making them adaptive through extending their behaviour to become context-aware by providing contextual knowledge for the hosting environment, and by giving them the capability of readjustment in response to contextual thresholds. This has the potential to ensure that the system tolerance can adapt to changing conditions, thus becoming more antifragile towards stress.

The first step is to augment the system with performance metrics. By making performance metrics accessible to the adaptation module, more accurate and context-aware strategies can be executed that better adapt the current environment state in a real-time manner. This allows converting these strategies to become more context-aware and self-adaptive to their environment.

Secondly, system reconfiguration by avoiding static and hardcoded configurations and making fault tolerance configuration customizable at the run-time level, the Adaptation module can adjust system configurations to find the best values to keep the system stable under chaos.

4.4. UNFRAGILE framework workflow

The UNFRAGILE framework sequence diagram in Fig. 7 illustrates the iterative workflow that aims at enhancing antifragility, which involves the Chaos Module, System Under Analysis, Adaptation Module, and Monitoring Module. The system user (engineer or developer) initiates a chaos experiment by selecting it from a predefined chaos experiments library; then the Chaos Module will select all target components and inject faults according to the chaos experiment into the System Under Analysis. The Monitoring Module collects performance metrics before, during, and after the experiment, and the monitoring data are then provided to the Adaptation Module. The Adaptation Module analyses the data to detect if fragility was found, and then it applies mitigation strategies that are related to the detected fragilities to the System Under Analysis. This loop continues, ensuring the system's quality is continuously evolving to an optimal state, from being fragile towards specific types of chaos and disruptions to becoming resilient and, ideally, antifragile, as long as the user validates improvements after each iteration.

5. Case study

5.1. Experiment description

We have built a chaos experiment that seeks to convert existing software to become antifragile towards a common cloud failure, namely outbound latency, through a series of exploratory phases that follow the process outlined in the UNFRAGILE framework, while evaluating the quality of software response to chaos and stress at each phase. This allows us to verify the feasibility and applicability of the UNFRAGILE framework.

Software systems are frequently constructed from distributed components that run on cloud platforms and rely on third-party service providers. Protocols like HTTP, TCP, and RPC are commonly used to communicate between these components. However, the failure modes of their interdependence pose difficulties for these distributed systems. When requests propagate slowly or stall indefinitely, inter-service communication can result in cascade failures. If a request is delayed for an extended time, the client devotes resources to it. When a significant amount of such requests depletes the server's finite resources, such as memory, threads, connections, or other constrained resources, resource depletion and system failure might occur.

5.2. Experiment design

To apply the UNFRAGILE framework to a real-world web application, a proof-of-concept .NET application is built. The application contains several APIs that resemble different system response maturity levels. The system components are hosted in Docker containers through Docker-compose configurations. It is possible to package and run an application in a loosely isolated container using Docker. Because of the isolation and security, multiple containers can be executed concurrently on a single host where applications are executed without depending on what is installed on the host because containers are lightweight and come with everything required to run them. The .NET Application APIs

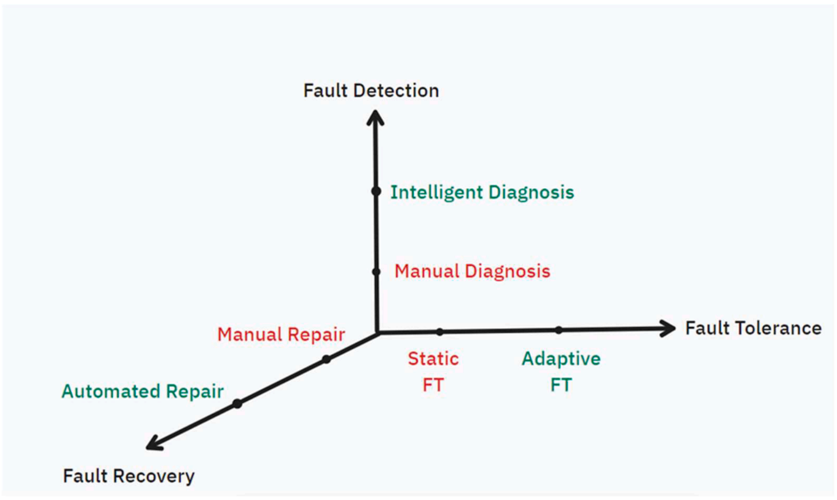


Fig. 6. Fault-tolerance antifragility matrix.

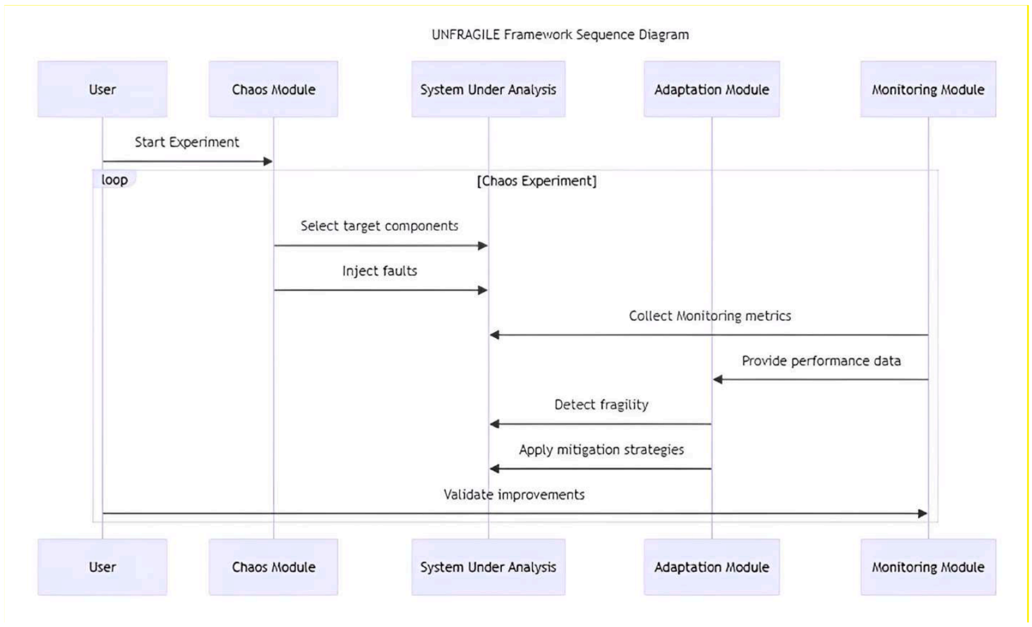


Fig. 7. Fault-tolerance antifragility matrix.

are dependent on another service that was built using NGINX to deliver the API response. To illustrate how the chaos and latency in outbound network calls can affect the system, ToxiProxy [45] is utilised to inject a gradual latency in the network calls to that dependency (NGINX web service) by routing all communications from .NET application to that service through ToxiProxy, while monitoring all relevant metrics and how they are affected from the injected latency. In the experiment, a tool called NBomber [46] is used which is written in .NET to generate traffic. The source code for the case study application, including all associated modules, is openly available on GitHub [47] for reproducibility and further exploration.

5.3. Experiment details

5.3.1. Experiment overall architecture

Fig. 8 demonstrates the overall architecture of our experiment's components and their relationships, which is also a concrete instantiation of UNFRAGILE framework architecture.

5.3.2. Hosting machine

All system components were hosted on an Amazon AWS EC2 VM instance from the T3 Family. The model used was T3. Large1 with 8 GiB of memory and 2 virtual CPUs. The instance had the capability to support up to 3 TB of EBS block storage for storage purposes. In terms of network performance, it could deliver speeds of up to 5 Gigabit. Overall, the T3 instance belonged to the T3 Family, which is known for its burstable performance that aligns with our experiment workload.

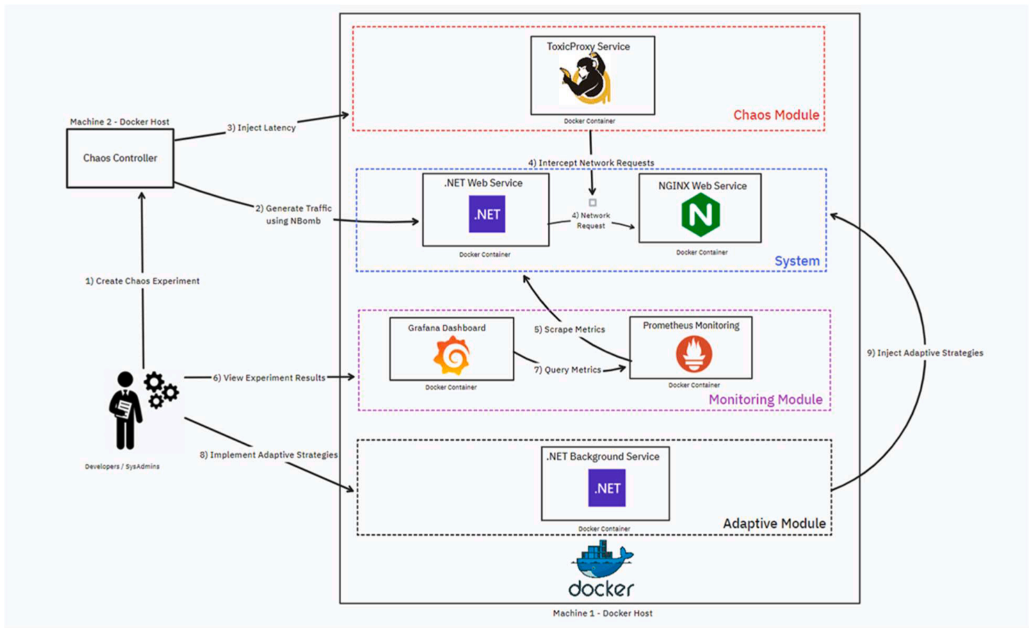


Fig. 8. UNFRAGILE case study application components architecture.

5.3.3. Chaos module

a) ToxiProxy [45]: an open-source proxy server developed by Shopify which is used to simulate latency, errors, and other network conditions. It is a popular tool for testing the resilience of applications to network failures. ToxiProxy can be used with a variety of programming languages, including Python, Java, and Go. This tool has been used because it exposes an API to configure latencies, which is required for the experiment since the latency is increased gradually to simulate a realistic workload. Also, the proxy is low-overhead by design.
b) NBomber [46]: a .NET library that can be used to send large numbers of HTTP requests to a server. It is a popular tool for testing the performance of servers and for performing denial-of-service attacks. The library can simulate real workload to cover complex cases by mixing Pull/Push scenarios, protocols (HTTP/WebSockets) and formats (XML/JSON/Protobuf) [46]. This tool has been selected for convenience because it supports customising simulation and can be integrated with ToxiProxy. It also provides experiment results in several formats that can be automatically parsed and persisted in a time-series database.

5.3.4. Monitoring module

a) Prometheus [41]: an open-source monitoring system and time-series database. It is used to collect metrics from a variety of sources, including servers, applications, and services. Prometheus stores the collected metrics in a time series database, which can be queried to generate graphs, tables, and alerts using Grafana. Prometheus is also used to scrape the system and hosting machine metrics that are exported by cAdvisor [48], One of the crucial features of Prometheus is Instrumentation, which means adding and exposing your own custom metrics. We also use Prometheus to instrument several metrics that are useful for the application. The metrics we have instrumented in our application using Prometheus Library are the following: a) Standard metrics include CPU usage percentage, memory usage percentage, TCP sockets metrics, and .NET API response time latencies. b) Additionally, custom metrics such as Injected latency, Adaptive concurrency limits, and 95 percentiles of latencies are also monitored to ensure comprehensive monitoring and analysis of system performance.
b) Grafana [49]: a dashboard and visualisation system for Prometheus collected metrics or other data sources. They are used to monitor and troubleshoot applications, systems, and infrastructure. Grafana is used to create the experiment's custom dashboards in order to monitor experiment results.

5.3.5. Adaptation module

A background .NET service that adjusts software concurrency configurations based on observed latency from an NGINX web service, using the Additive Increase and Multiplicative Decrease (AIMD) algorithm [50]. It utilises shared memory to update concurrency limits and shed load if requests exceed those limits. Load shedding [51] is essential to prevent system overload, prioritise efficient request handling, and maintain low latency for accepted requests. This approach creates a sustainable and efficient system by avoiding wasted work and ensuring high availability while mitigating the impact of excess traffic.

5.3.6. System under test

A proof-of-concept application is shown in Fig. 9, built using C# .NET 7.0 that provides blog content through APIs. The solution consists of two components:

a) A .NET 7.0 API service
b) NGINX Headless CMS.

The .NET Application receives API requests from clients for blog posts and consequently issues HTTP Requests to NGINX Web service to retrieve blog textual content. The application is a cross-platform cloud-native Docker application that exposes multiple APIs using the MVC pattern. It relies on the NGINX service to serve content, which is fetched through HTTP requests using the HTTPClient. On the other hand, the NGINX-based Headless CMS is a web server that is responsible for serving content over the network via API calls. When a client sends a request to the .NET service, it retrieves the required content from the NGINX server to generate the response. Using Docker and Docker Compose, the application and all of its dependencies are containerised, allowing for simple deployment and scaling across various cloud environments. Thus, by leveraging containerisation in application operation and deployment, it can be considered a cloud-native application.

5.3.7. Chaos engineering plan

The following steps are necessary in the chaos experimentation procedure, as we stated in the introduction: capturing a snapshot of system metrics at steady state, defining chaos variables, defining hypotheses, and defining the blast radius.

In this work, the experiments are designed to generate a workload of 100 concurrent requests, which are injected every 5 s, generating a total of 9600 requests over an 8-minute duration using the NBomber traffic generator [46]. This workload simulates a realistic scenario that would be suitable for a shared hosting docker container. Each experiment was conducted ten times, and the monitoring data, which included the average latencies, CPU, memory, and TCP allocations, were calculated. Minimal standard deviations were observed, ensuring an accurate representation and confirming the data is statistically significant. In total, the experiment involves 96,000 HTTP requests (9600 requests per experiment * 10 trials) and is repeated for each phase (steady state, fragile, robust, and antifragile), resulting in 384,000 requests. We will provide a snapshot of system metrics in the next section. Each phase was run for 1:20 h in real-time using the AWS cloud. A total of 5:20 h of

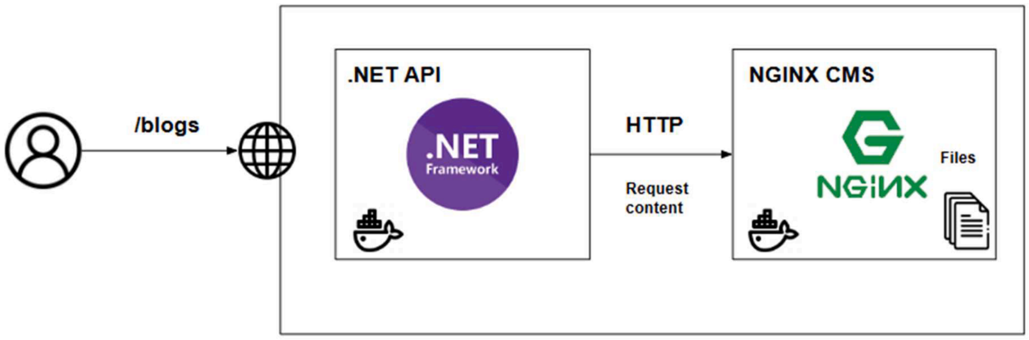


Fig. 9. Proof-of-concept application architecture.

real-time experiment time was conducted to collect the results for all phases (steady state, fragile, robust, and antifragile).

a) Chaos variables

In the experiment, "network latency" was injected gradually using ToxiProxy in outbound network communications, as shown in Fig. 10. The latency was increased after that by 0.4 s every 5 s over 8 min. The goal was to simulate a real-world cloud application network congestion workload. The latency reached a maximum of 15 s, which is typically experienced during congestion in cloud systems or when dependent systems are experiencing downtimes. The breakdown of the experiment is as follows:

1. For the first 2.8 min, the latency was increased gradually by 0.4 s every 5 s.
2. For the next 2.4 min (1/3 of the experiment time), the latency was kept at the peak level of 15 s.
3. Finally, for the last 2.8 min, the latency gradually decreased by 0.4 s every 5 s until it reached 0 s, simulating normal response times.

The purpose of this experiment is to mimic the behaviour of transient cloud system congestion, which typically occurs for a few minutes and then resolves itself with cloud auto-healing, manual intervention, etc.

b) Hypothesis: The .NET API metrics at a steady state will persist after the chaos perturbation experiment duration.
c) Blast radius: Chaos latency injection is only applied to NGINX web service, which is required for .NET Web API requests.

6. Results and discussion

We organised our data into four phases. First, we have our baseline, or steady-state, results, which are obtained prior to the introduction of any network latencies. This is followed by the three phases fragile, robust, and antifragile, and these results will serve as our reference point for assessing the hypothesis following the introduction of latency. Furthermore, our findings are categorised as antifragile, robust, and fragile phrases based on our analysis of the system response to chaos at each stage in comparison with the steady state. The following subsections explain these phases in detail.

6.1. Steady state results

The steady-state results before injecting any network latencies are documented, this will become our baseline for evaluating the hypothesis after injecting latency. Fig. 11 demonstrates the API latencies, TCP allocations, memory usage, and CPU usage for our baseline.

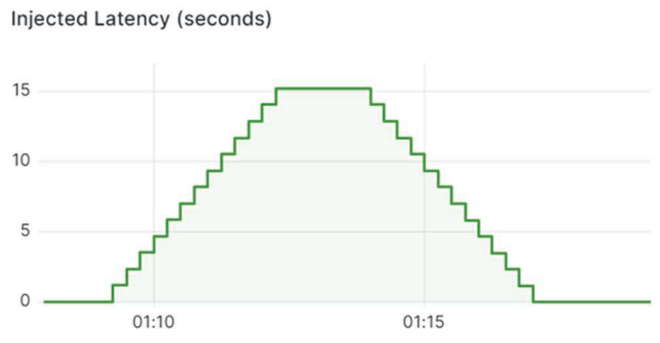


Fig. 10. Chaos experiment design - injected latencies.

6.2. First phase (Fragile phase) results

The results after injecting latency gradually up to (15 s) with ToxiProxy, generating realistic traffic over 8 min duration, repeating the same experiments 10 times and capturing the averages of each measurement (min, max, mean). Fig. 12 shows the results of the fragile phase compared to the results of the steady state for latency, TCP allocations, memory usage, and CPU usage.

Analysing the results for the relevant metrics (TCP, CPU, Memory) of the .NET application during the chaos experiment shows that the system was highly sensitive to the injected latency. The system's TCP connections experienced an unbounded increase (from the baseline of a maximum of 98 socket connections to a maximum of 5460 socket connections after injecting latency), and it will keep increasing if the experiment persists. The CPU unexpectedly was lower, we can explain this by the observation that the awaited network requests are locking the threads during the wait-time, and making it idle and unable to process all other HTTP requests till the response from the outbound service returns, which is apparent from the rise in latency to more than 70 s (compared to steady state) for some of the requests although our injected latency upper bound was 15 s, which means that the HTTP requests are experiencing long queueing time because there are no threads are available in thread pool to process them. All of which led to the accumulation of open sockets, and made most of the requests fail, which eventually caused the system to crash (before completing all the requests). Because the hosting machine is limited in the number of open socket connections at a given time. This behaviour is consistent with a concave response to stressors, indicating that the system is fragile under stress. The root cause analysis revealed that the API component, which was under stress, lacked asynchronous programming or any proactive protection measures against latency in outbound communication. This causes those threads to be locked, waiting for the response from the NGINX service, and unable to process other requests concurrently. To mitigate the fragility of the application, the following strategy is implemented.

6.2.1. 1st phase adaptation strategy

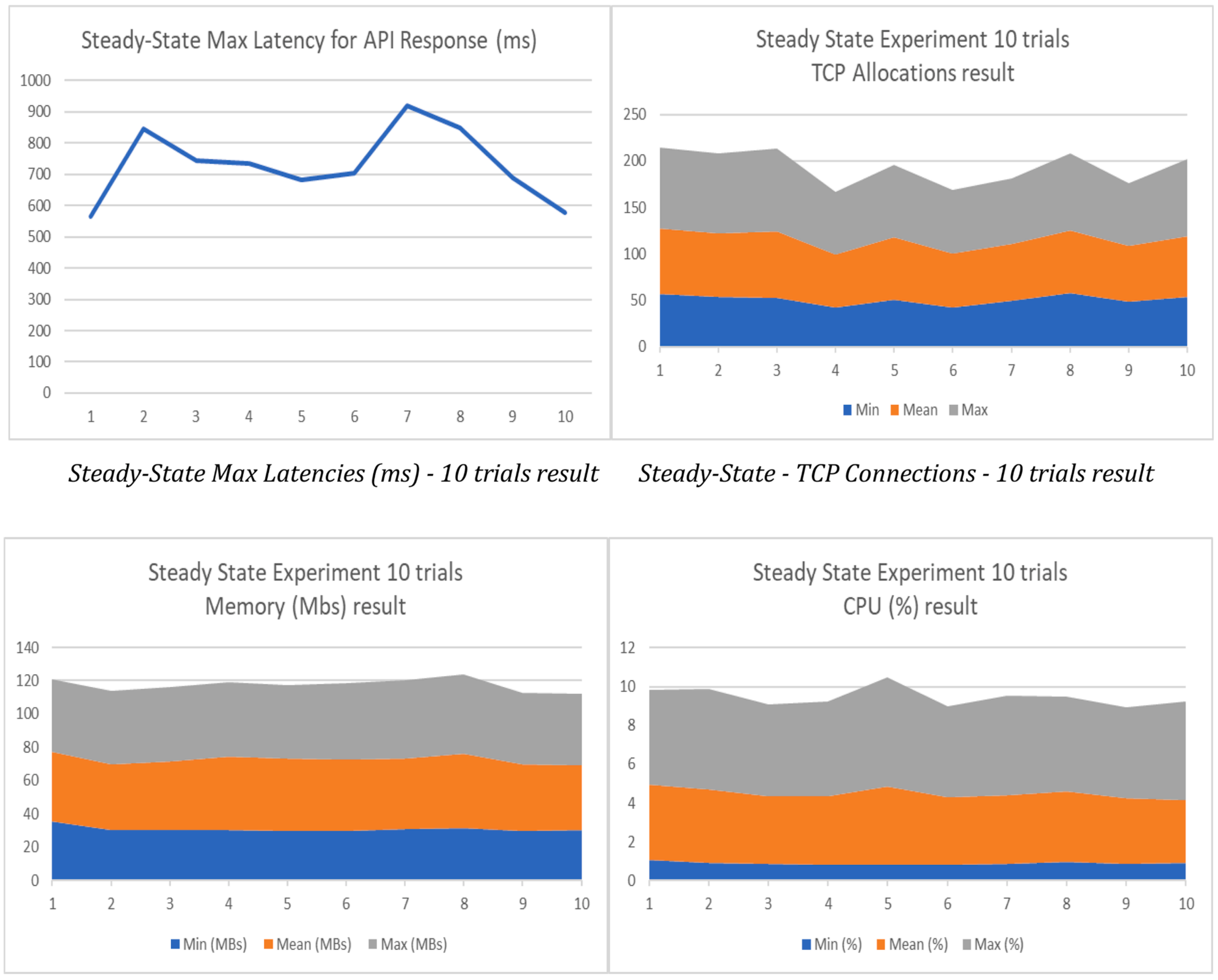
To fix the fragility of the .NET API, all network calls are converted to become asynchronous leveraging the TPL (Tasks Parallel Library) in . NET. Also, an upper bound timeout of 10 s is implemented, if the network call to NGINX service exceeds this timeout, the request will fail gracefully which will potentially reduce all the open socket connections, there are other strategies that can be implemented to tackle the same problem such as "Circuit breaker" and "Bulkhead isolation" and "Concurrency limits''. To build this proactive policy, Polly.NET [52] is used, which is a .NET library that provides resilience and transient-fault handling capabilities. It allows developers to handle transient faults such as network errors, timeouts, and other types of errors that can occur when interacting with external services or systems. Those capabilities can be implemented by applying Polly policies such as Retry, Circuit Breaker, Bulkhead Isolation, Timeout, and Fallback.

6.3. Second phase (Robust) results

The results after injecting latency gradually up to (15 s) with ToxiProxy while implementing asynchronous HTTP requests using Task Parallel Library (TPL) and proactive timeouts are shown in Fig. 13

6.3.1. 2nd phase analysis

After implementing proactive strategies, latency significantly improved significantly compared to the previous phase. The maximum recorded value decreased from more than 70 s to 10 s, demonstrating a noteworthy improvement. The system was able to withstand the disturbance period without crashing or excessively depleting system resources. Furthermore, the number of open socket TCP connections reduced dramatically, with a maximum of 2.8k connections compared to



Steady-State Max Latencies (ms) - 10 trials result *Steady-State - TCP Connections - 10 trials result*

Steady-State Experiment - Memory - 10 trials results *Steady-State Experiment - CPU - 10 trials results*

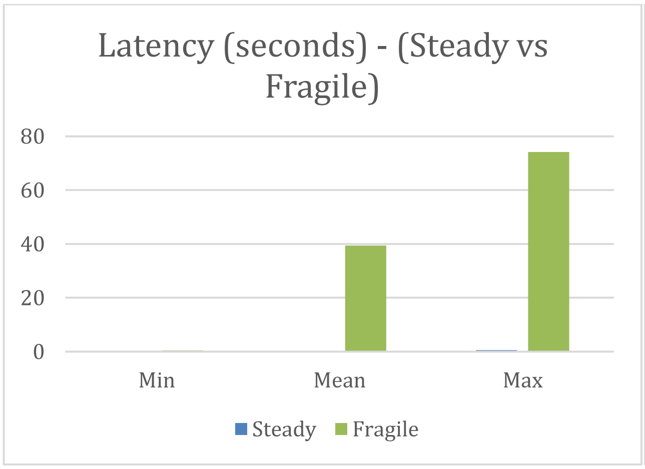
Fig. 11. Experiment results - baseline phase.

5.5k in the previous phase. The system remained stable, and if the request latency exceeded the defined timeout of 10 s, the system terminated the requests and returned an HTTP status 400 error response to the client. This behaviour can be seen in the Fig. 14.

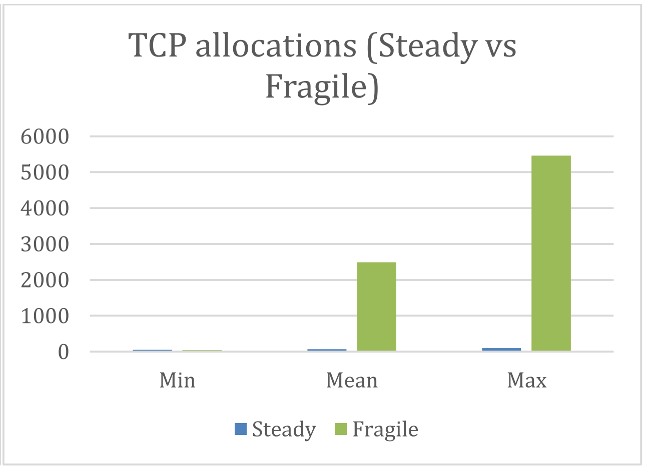
The system’s behaviour can now be considered ***Robust***, meaning that the system’s reaction to increasing stress is neutral and bounded. Although such a level of maturity might be acceptable for production-grade systems, setting a constant (or hardcoded) timeout value can lead to several problems. One of the significant issues is that the constant timeout may not be suitable for all network calls since different network calls can have varying response times, depending on factors such as network congestion, server load, and latency. If the timeout value is set too low, it can result in premature timeouts and incomplete requests, leading to poor user experience and lost data. On the other hand, if the timeout value is set too high, it can lead to prolonged wait times, which can also negatively affect user experience and system performance. Another problem with setting a constant timeout is that it may not be effective in dealing with variability in network conditions. For example, if the network experiences a sudden spike in latency, a fixed timeout value may not be sufficient to handle the increased response time, leading to more request timeouts and system failures. To mitigate these challenges, it is required to apply the principles of antifragility outlined in the literature and practice by implementing adaptive fault-tolerant strategies. One approach that has been demonstrated by Netflix [53] and other large systems providers to be particularly effective is to make the system self-adaptive to changing latencies. This involves empowering the system with context awareness and the ability to dynamically define limits, thus enabling it to respond to unexpected latencies in real time. Rather than relying solely on timeouts to manage latency issues. Thus, it is more effective to restrict the number of concurrent requests sent to the external outbound system. When the system receives more concurrent requests than it can handle, some requests will need to be queued, which, in turn, increases the overall timeout for these requests. Even in cloud environments with auto-scaling capabilities, the ability to handle concurrent requests is always constrained by the available processing power. Therefore, Netflix recommends controlling concurrent requests as a means of managing overall system performance [53] instead of focusing solely on fixed limits. By doing so, the system gains the ability to self-improve and self-adapt rather than relying on static policies that are inherently proactive, such as timeouts. which aligns with the antifragility matrix that we’ve proposed before.

6.4. *Adaptive strategy*

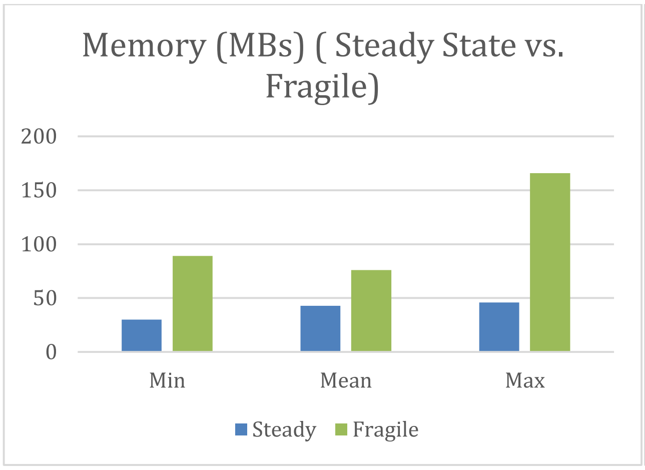
The adaptive concurrency limit is a technique developed by Netflix [53] and was adopted by many companies [54–56] to improve service availability and prevent cascading failures in their large distributed systems. The algorithm which is shown in Fig. 15 is based on AIMD



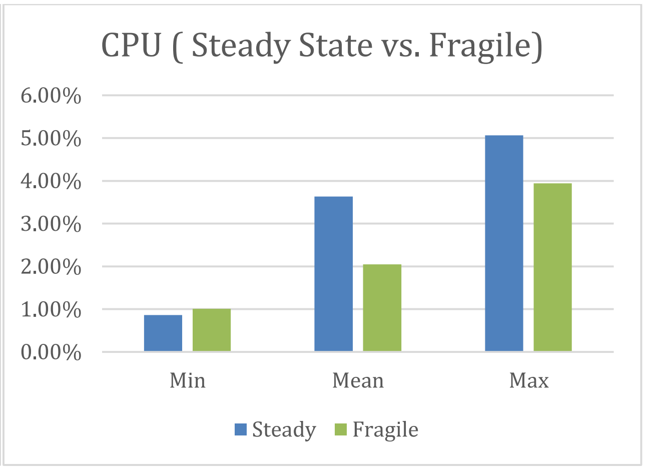
Chaos Experiment Result - Fragile – Latency



Chaos Experiment Result - Fragile – TCP



Chaos Experiment Result - Fragile – Memory



Chaos Experiment Result - Fragile – CPU

Fig. 12. Chaos experiment results - fragile phase.

(Additive Increase and Multiplicative decrease), which was originally used in TCP (Transmission Control Protocol) [53] . Concurrency refers to the number of requests a system can handle simultaneously, and it is usually limited by a fixed resource such as CPU [57]. When the number of requests exceeds the concurrency limit, the system must queue or reject them, which can lead to increased latency and, ultimately, system failure, and consecutively, the propagation of failure to the origin system because if increased latency is left unchecked, it will start disturbing the callers' subsystems leading to cascading failures and fragility through all of them. Concurrency refers to a system's ability to process several requests at the same time. It is usually determined by a restricted resource, like the CPU [58]. This latency can be explained by Little's law [59] which asserts that the concurrency of a system in the steady state is equal to the average service time multiplied by the average service rate.

$(L = \lambda W)$

Determining the optimal concurrency limit for an outbound system has been a manual and time-consuming process, and it quickly becomes stale as the system's topology changes due to outages, auto-scaling, or code push. Determining the optimal concurrency limits based on run-time metrics [35] can solve this problem. which is the exact solution implemented by Netflix in their distributed systems. The algorithm adjusts the concurrency limit based on latency measurements and adds an allowable queue size to account for bursts.

The algorithm initially sets a low concurrency limit and gradually increases it by a fixed amount (+1) if the latency of requests in the last period of time remains below the threshold, as shown in Fig. 16, this additive increase phase allows the system to utilise more resources and improve request throughput. If the latency exceeds a specific threshold or a request timeout, the algorithm switches to a multiplicative decrease phase. During this phase, the concurrency limit is reduced by a fixed percentage (60 % in this case) to prevent system overload and cascading failures. This adaptive mechanism helps protect the system by adjusting the workload in response to changing conditions. It aligns with the concept of overcompensation for potentially worse situations, as described in Taleb's book, "Antifragile". *"A system that overcompensates is necessarily in overshooting mode, building extra capacity and strength in anticipation of a worse outcome and in response to information about the possibility of a hazard."* [1]. For our implementation, the initial configurations [58] are as follows: (a) the latency threshold is set to 1 s (99 Percentile of the latencies at steady-state), (b) the decrease percentage is 60 %, the update interval is 2 s (where the Adaptation module calculates the 95 percentiles of the recorded latencies and updates the concurrency limit every 2 s), and the (c) round-trip time (RTT) uses the 95 percentiles for latencies.

The adaptation module of the UNFRAGILE framework is designed to detect system fragility, analyse root causes, and apply mitigation strategies to enhance system antifragility. In this case study, we leverage the

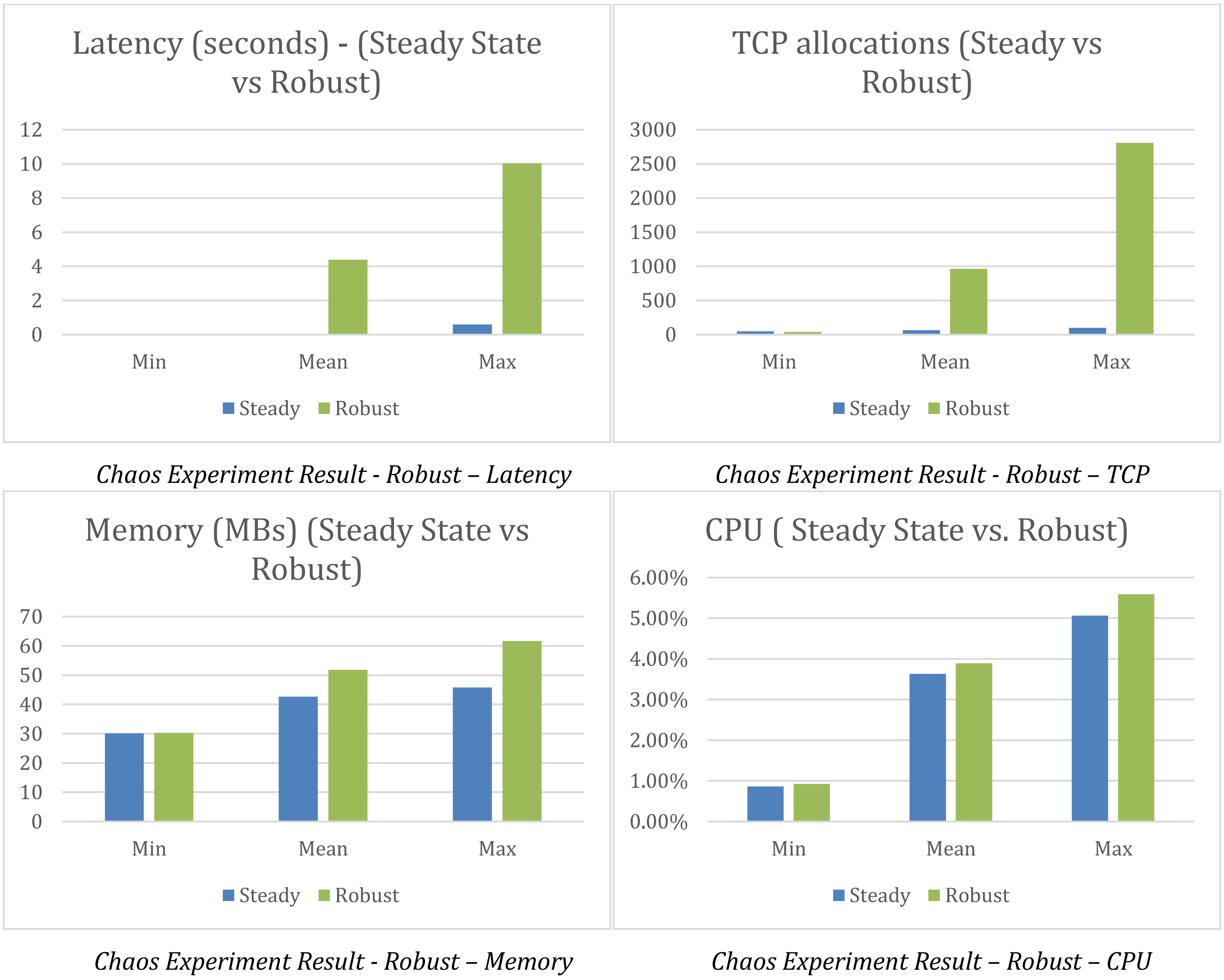


Fig. 13. Chaos experiment results - robust phase.

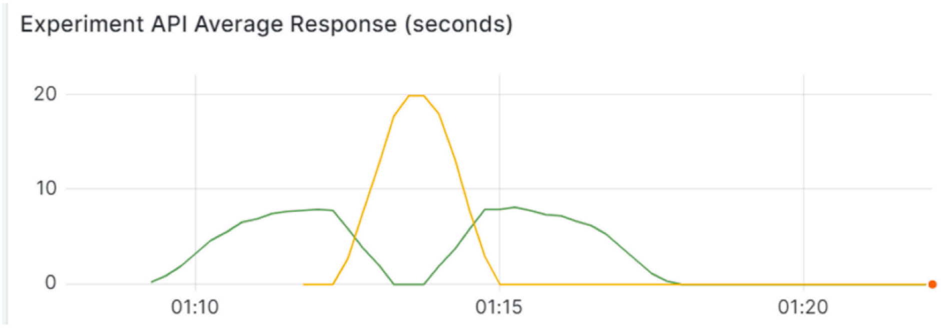
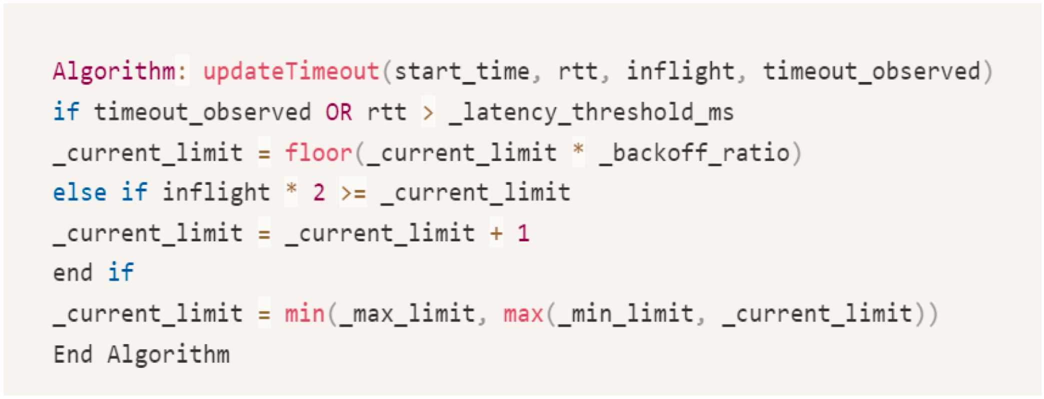


Fig. 14. Chaos experiment result - robust - average responses.

AIMD (Additive Increase Multiplicative Decrease) algorithm to adjust system parameters dynamically in response to real-time performance data. The AIMD process involves periodically increasing resource allocations (additive increase) until an anomaly is detected, at which point it decreases resource allocations by a multiplicative factor (multiplicative decrease). This reconfiguration process allows the application to adapt to disruptive situations, this has all been implemented as .NET services through the following components:

- **Adaptive concurrency policy**: This policy manages the number of concurrent outbound requests in the system, with the capability to dynamically adjust the concurrency levels. It utilises a semaphore lock in conjunction with the Polly.Net library to implement and apply the policy on HTTPClient, the responsible driver for all network requests in .NET
- **Detection engine**: Implemented as a recurring background job using the BackgroundService library in .NET [60]. The detection engine reads monitoring data from the monitoring module (Prometheus). It frequently executes to gather all necessary parameters, such as the number of in-flight requests, available slots, and the P95 latency. Based on these metrics, it evaluates whether the adaptive concurrency policy requires reconfiguration.



```
Algorithm: updateTimeout(start_time, rtt, inflight, timeout_observed)
if timeout_observed OR rtt > _latency_threshold_ms
_current_limit = floor(_current_limit * _backoff_ratio)
else if inflight * 2 >= _current_limit
_current_limit = _current_limit + 1
end if
_current_limit = min(_max_limit, max(_min_limit, _current_limit))
End Algorithm
```

Fig. 15. Adaptive concurrency limits algorithm [53].

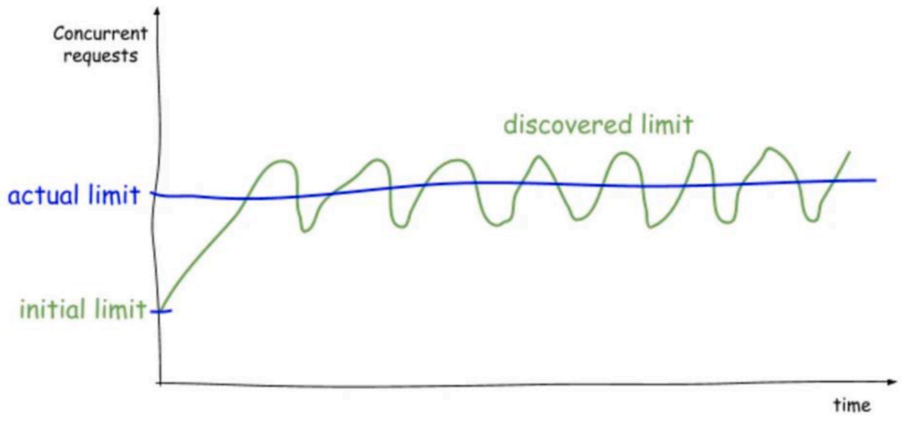


Fig. 16. AIMD algorithm in action [53].

- **Mitigation executor**: After determining if the current number of concurrent requests is adequate or needs adjustment, the mitigation executor updates the policy accordingly, increasing or decreasing the concurrency limits as necessary by scaling up or down resources and modifying configurations.

6.5. *Third phase (Antifragile) results*

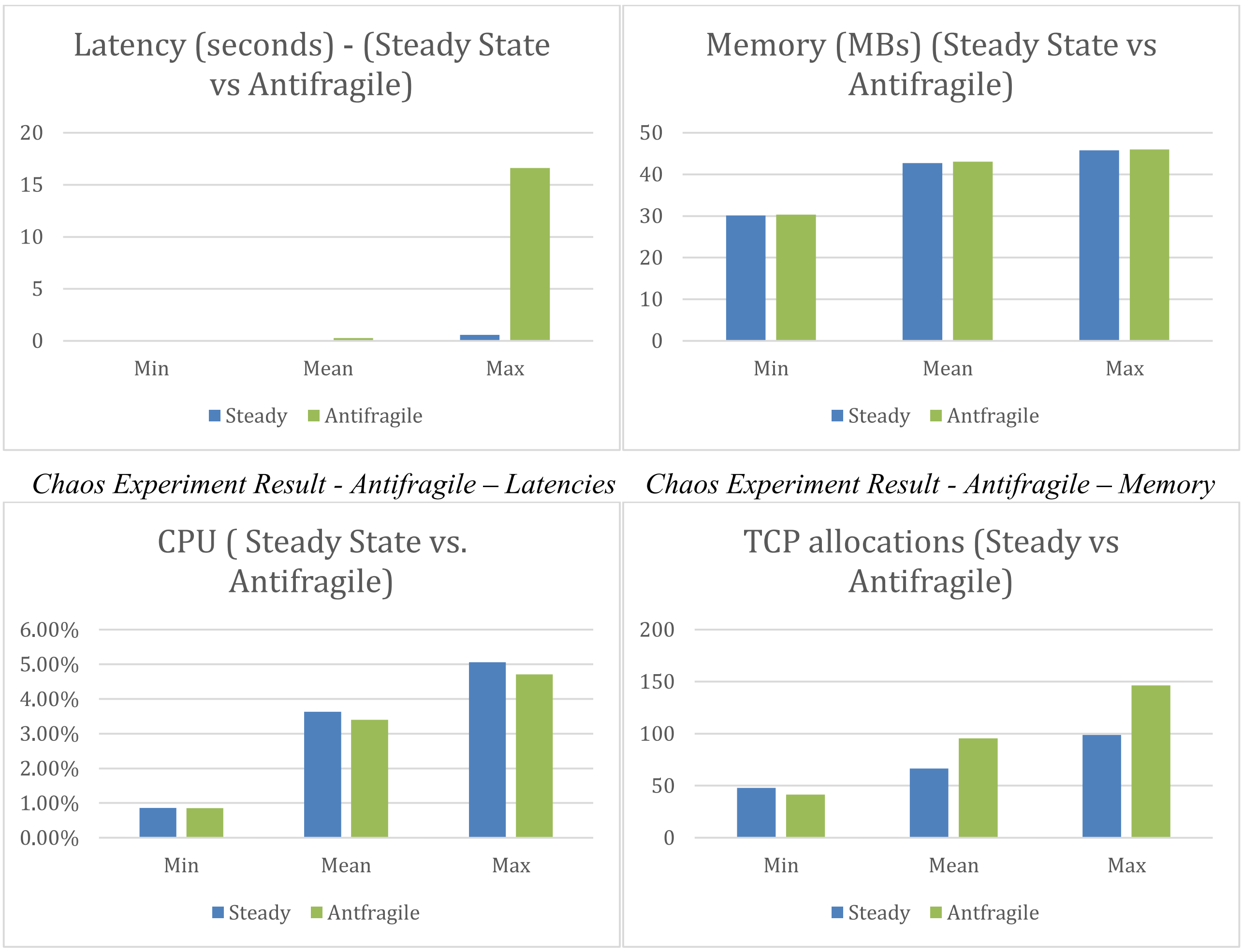
Fig. 17 shows the results after injecting latency gradually up to (15 s) with ToxiProxy while running an adaptive concurrency limit strategy using the AIMD algorithm in the adaptation module.

6.5.1. *Third phase analysis*

An obvious improvement can be seen in the figures above in all of the performance metrics during the chaos experiment. Furthermore, the TCP ports were intensely reduced because the concurrent requests were reduced after the system had observed a gradual increase in the latency in the outbound dependency, which is a significant improvement compared to the previous iteration because the system resources are now protected against volatile behaviour of the external systems, not only this, it was not required to configure any static limits, the system was able to adapt to the changing environment according to the AIMD algorithm [53] in which adaptively reaches the best concurrency limit to enforce it on the components communicating with that outbound dependency according to the observed behaviour from that outbound dependency. Fig. 18 shows Adjusted Concurrency Limits that were autoconfigured by the adaptation module and was instrumented in the monitoring module.

The adjusted concurrent limit was recorded on a regular basis, and as shown in the charts above, limitations started as low as 10 concurrent requests and increased at the beginning of the experiment when the 95 percentiles of calculated latencies did not exceed the 1 s threshold. However, as the chaotic module began to inject more latencies, the adaptive module multiplicatively decreased the concurrency limit, and it continued to decline until the outbound systems hit the configured minimum number of concurrent requests (5 requests). The outbound system progressively recovered and improved its latency, while the adaptive module gradually raised the allowable concurrent requests as the experiment neared its finish. The client responses for rejected requests due to overload is 400, as shown in Fig. 19. The number of requests that have been load shedded by the adaptation module due to overload increased during the period when latency had increased beyond the threshold, then it returned to its normal rate gradually.

As observed in the three phases of application incremental evolution, fragile behaviour was uncovered through chaos engineering in the 1st phase (fragile phase). It was clear that the system was indeed fragile to outbound latency, which is a prevalent issue in cloud and distributed systems. The conducted empirical analysis for fragility, which is represented by the disturbance in application performance metrics and the impact of chaos on these metrics, especially allocated socket connections and memory, has shown that the system is sensitive to that class of chaos and that it quickly stresses system resources and propagate failure to other system components, which resembles the concave response as explained by Taleb and Douady [61]. Consequently, an adaptation phase was required, through which several strategies should be introduced to withstand chaotic networks in latency in network calls for external downstream dependencies. Moreover, it can be seen in the comparison figures below that the behaviour of the steady stage is similar to the behaviour of the antifragile stage, which means that without defining a static configuration, the system was able to



Chaos Experiment Result - Antifragile – Latencies *Chaos Experiment Result - Antifragile – Memory*

Chaos Experiment Result - Antifragile – CPU *Chaos Experiment Result - Antifragile – TCP*

Fig. 17. Chaos experiment results- antifragile.

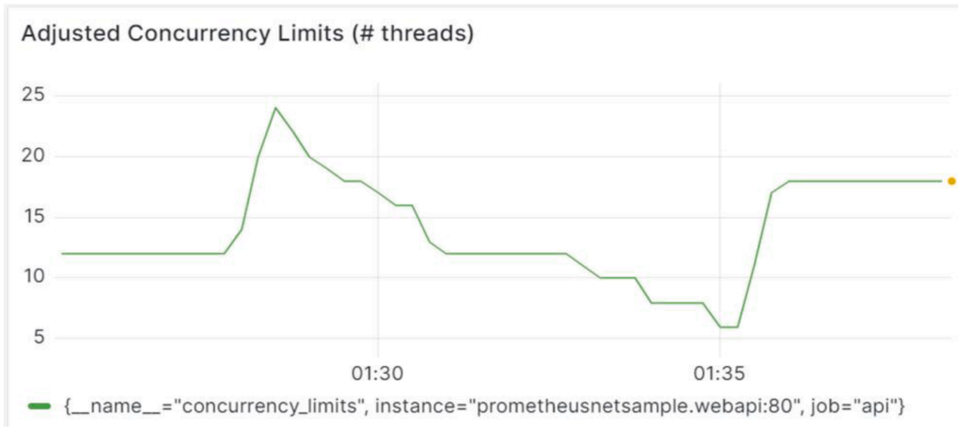


Fig. 18. Chaos experiment result - antifragile - adjusted concurrency limits.

self-stabilise during the chaos after learning from the environment context that the dependent system is suffering. This was achieved through the implementation of adaptive fault tolerance strategies, which were employed as a means to achieve antifragility, as suggested by [4]. This answers the first research question and demonstrates how automated operational antifragility of real-world cloud applications may be achieved.

Conventional resilience strategies are generally designed to facilitate the rapid recovery from system failures or to withstand such failures. In contrast, Adaptive fault tolerance approaches harness the lessons learned from failures to guide the application towards the optimal strategies for effectively managing the observed environment, this aligns with the antifragility definition. Our results analysis, which is shown in Fig. 20 for implementing an adaptive fault tolerance strategy, namely

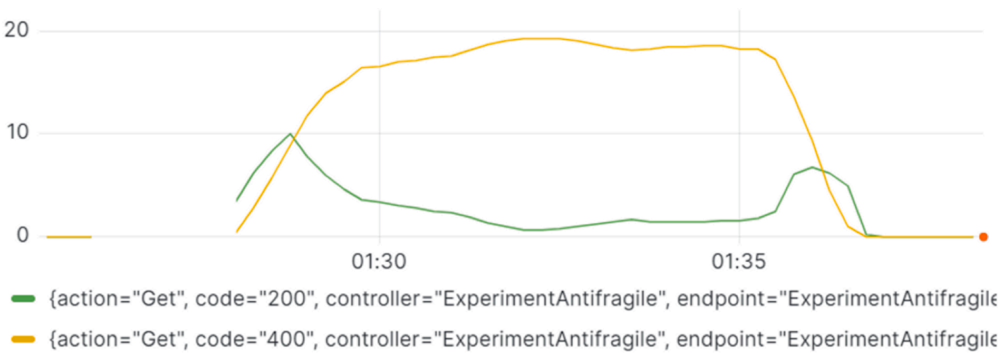


Fig. 19. Chaos experiment results - antifragile - API responses.

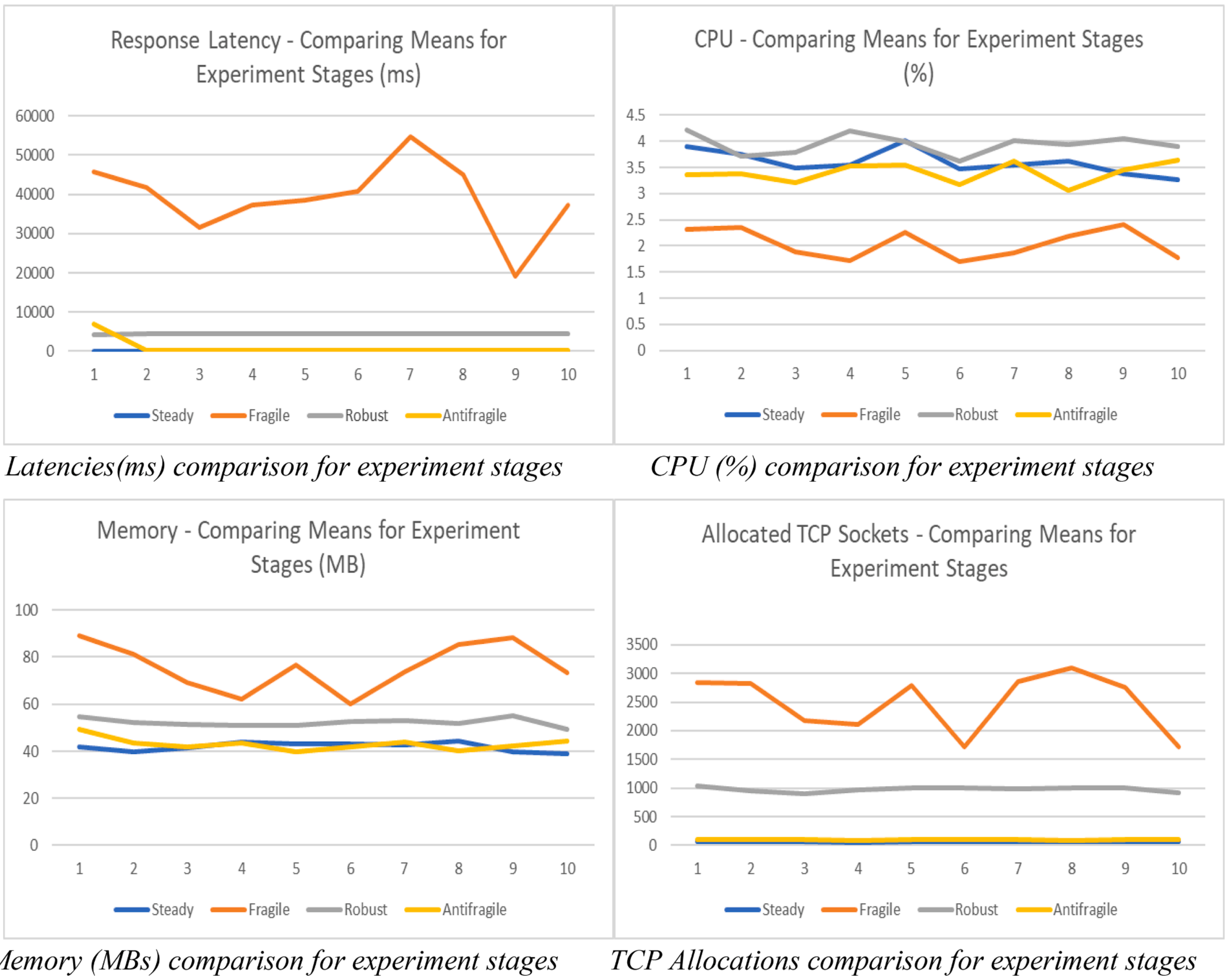


Fig. 20. Comparing means for experiment stages.

AIMD, demonstrates how the UNFRAGILE framework has been successfully used, answering the second research question.

The study demonstrates the UNFRAGILE framework's ability to be implemented incrementally through incrementally embedding adaptive fault tolerance strategies. This approach is used to differentiate between robust and antifragile responses and to demonstrate its integration within an incremental development lifecycle, ensuring an application that is truly antifragile in uncertain and chaotic environments.

6.6. *UNFRAGILE applicability to more complex architectural frameworks*

UNFRAGILE may be used for more complicated systems and in complex architectural frameworks, but it may also present complexity issues when integrating it into existing architectures. For example, if we are considering microservices architecture, this might require each microservice to have a substantial code refactoring process to be integrated with UNFRAGILE.

Performance overheads that could impair the general performance of such systems by increasing latency and use of resources is another issue that might be faced. Moreover, using UNFRAGILE for such systems may cause serious difficulties due to limited maintenance support. UNFRAGILE may as well have unidentified security gaps that create weak points when combined with current systems, potentially introducing new security vulnerabilities. As a result, UNFRAGILE demands that strong security be provided both before and after the transition.

The expense of implementing UNFRAGILE may also be a barrier because it entails overhead, the need for more experienced staff, and the possibility of failures and downtime during the transition. Furthermore, teams inside organizations may be resistant to change as a result of modifications to workflows and technologies, which makes resistance to change a significant difficulty to take into account. Organizations with established architectural frameworks must pay attention to their application and transition processes. These organisations must weigh the benefits and drawbacks and create a thorough transition plan that avoids causing problems with technology and business.

6.7. Implementation and scalability challenges

The question of scalability was addressed in the overall design of the UNFRAGILE framework. One of the design principles we have adopted is the decoupling of the main components (chaos, monitoring, adaptation) of the framework from the system under test. This approach helps avoid the pitfalls of common optimization frameworks that often make it harder to scale or integrate with existing systems. Decoupling means that the chaos, monitoring, and adaptation components can be scaled independently. The exact scaling strategy—whether horizontal or vertical—is left to the implementation details of the experiment that adopts the framework. In the paper, we suggested using cloud-native methods for implementation because they facilitate the introduction of scaling policies that fit specific requirements. By utilising cloud-native technologies, it becomes much easier to add more monitoring and chaos components, thereby enhancing the scalability and flexibility of the UNFRAGILE framework.

Another challenge for scaling the UNFRAGILE monitoring module is when the system has many components with complex relationships, but since chaos engineering is the methodology we use in designing the chaos experiments, the methodology originally defines the idea of the “blast radius”. This means that when designing the chaos experiment, the target components are selected, and every component in the system is monitored. Although the complexity of this task increases with the number of system components, we suggest mitigating this challenge by using auto-instrumentation when adding the system monitoring module. Auto-instrumentation and Auto-Discovery should make it easier to discover all operating system components and put them under the radar of monitoring before, during, and after the experiment. Cloud-native also makes it easier to introduce auto instrumentation through code injection at runtime and service discovery strategies (OpenTelemetery, for example, uses such strategies).

7. Conclusion, limitations, and future work

In an effort to create antifragile systems out of current software, the paper examined cloud software antifragility from the viewpoints of practitioners and software engineers. The UNFRAGILE framework, built on antifragility principles, formalises a process for fault injection, monitoring applications, and software adaptation. The results in Fig. 20 show that the framework has defined an overall architecture of supportive modules to help move existing systems from the fragile phase to the antifragile phase through continuous experimentation utilising chaos engineering and monitoring application response to stress. The results of applying and validating the framework show that it is applicable and that it has the potential to enhance existing software incrementally, which is suitable for modern-day agile software development. However, the framework’s limitations lie in its application to cloud-native environments and distributed architectures, suggesting more studies on concurrent actor models, as well as other computing models. However, a single case study may not be enough to generalise the applicability of the UNFRAGILE framework for every type of systems stressor, we can still explore the framework on other adjacent operational disruptions for some of the properties of subject systems. Moreover, our case study involves specific assumptions about the cloud environment, application architecture, and types of failures introduced. Specifically, we assumed a particular set of dependencies and failure modes typical of .NET applications running in Docker containers. However, the UNFRAGILE framework itself is designed to be generalizable beyond the chosen technologies. The cloud-native approach can be used to implement the framework across various technology stacks without requiring changes to the overall design of the components and their interactions. Our study demonstrated the efficacy of applying the UNFRAGILE framework to enhance the antifragility of a cloud application. the case study we have designed to prove that was .NET Cloud-native application, so it is important to acknowledge certain limitations:

- Assumptions are related to the cloud model we are using. However, these assumptions might not hold in different contexts, such as legacy systems or serverless architectures that have upper limit for request time. or other different computing models such as concurrent actor models.
- While we demonstrated the framework’s effectiveness in a controlled environment with an all-in cloud architecture, monitored and orchestrated using the Docker toolset, Real-world applications might encounter additional challenges related to scalability and performance and multi-cloud approaches that were not fully explored in our study. Future work should involve testing the framework in large-scale environments to evaluate its scalability and instantiating the same architectural components for UNFRAGILE in other technology stacks.
- UNFRAGILE experiments must be carried out in a production environment, a limitation of this approach is that it does not employ static analysis methods to detect fragility during the development phase.

Due to resource constraints such as time, cost, and computational resources for the public cloud, we gave priority to in-depth and high-quality analysis over the number of experiments, which has the potential to provide valuable insights. Additional experiments could provide further insights. However, they require more resources, cost, and time. Our primary goal is to develop the framework and to demonstrate its practicality and applicability to real-world settings. The chosen case and benchmarks reflect the effectiveness of the framework under realistic conditions. In future work, we will focus on evaluating the framework across several other dimensions to ensure its robustness and versatility. Dimensions that include:

- Different Technology Stacks: We will apply the UNFRAGILE framework to other technology stacks such as Node.js. This will help validate the framework’s applicability and effectiveness across diverse programming environments and platforms.
- Different Chaos Attacks: We will introduce a variety of chaos scenarios, including hardware failures, and resource starvation. By testing these different types of attacks, we aim to comprehensively assess the framework’s ability to detect, adapt, and improve system resilience under a wide range of stress conditions.
- Different performance measures: taking system recovery time and adaptability measures into account is important. In future studies, this will enable us to present a more comprehensive and multidimensional evaluation of the framework’s performance and antifragility.
- Different Cloud Deployment Models: We will implement the UNFRAGILE framework across various cloud deployment models, including public, private, and hybrid clouds. This will allow us to

evaluate the framework's performance and adaptability in different cloud environments, ensuring that it can effectively enhance anti-fragility regardless of the deployment model.

- Varying Systems Architecture: Our case study targeted cloud-native web applications, but other system architectures exist, such as serverless architecture, we will focus on exploring how to adapt UNFRAGILE framework into them.
- Other Adaptation techniques and a recommender module that employs machine learning to identify and implement the best adaptation strategies are potential future research projects.

Funding

This work was supported by the Researcher Development Framework award [No. CSRDF22-23-1AA], Keele University, for Dr Al-Said Ahmad.

Availability of code

https://github.com/josephwasily/Defragile.

CRediT authorship contribution statement

Joseph S. Botros: Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Data curation, Conceptualization. **Lamis F. Al-Qora'n:** Writing – review & editing, Writing – original draft, Supervision, Methodology. **Amro Al-Said Ahmad:** Writing – review & editing, Validation, Supervision, Methodology, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Code is publicly available.

References

[1] N.N. Taleb, Antifragile things that gain from disorder, Random House Trade Paperbacks 23 (3) (2013), https://doi.org/10.1108/10595421311319852.
[2] C. Keating, *Anti-fragile: how to live in a world we don't understand*, vol. 13, no. 11. 2013. doi: 10.1080/14697688.2013.830860.
[3] K.J. Hole, Tutorial on systems with antifragility to downtime, Computing 104 (1) (2022) 73–93, https://doi.org/10.1007/s00607-020-00895-6.
[4] D. Russo, P. Ciancarini, Towards antifragile software architectures, Procedia Comput. Sci. 109 (2017) 929–934, https://doi.org/10.1016/j.procs.2017.05.426.
[5] D. Russo, P. Ciancarini, A proposal for an antifragile software manifesto, Procedia Comput. Sci. 83 (2016) 982–987, https://doi.org/10.1016/j.procs.2016.04.196.
[6] M. Monperrus, Principles of antifragile software, in: ACM International Conference Proceeding Series, 2017, pp. 1–4, https://doi.org/10.1145/3079368.3079412.
[7] V. Grassi, R. Mirandola, D. Perez-Palacin, Towards a conceptual characterization of antifragile systems, in: *Proceedings - IEEE 20th International Conference on Software Architecture Companion, ICSA-C 2023*, IEEE, 2023, pp. 121–125, https://doi.org/10.1109/ICSA-C57050.2023.00036.
[8] V. Grassi, R. Mirandola, D. Perez-Palacin, A conceptual and architectural characterization of antifragile systems, J. Syst. Softw. 213 (2024) 112051, https://doi.org/10.1016/j.jss.2024.112051.
[9] K.J. Hole, Anti-fragile ICT Systems, Springer Nature, 2016, https://doi.org/10.1007/978-3-319-30070-2.
[10] J. Choi, D.L. Nazareth, T.L. Ngo-Ye, The effect of innovation characteristics on cloud computing diffusion, J. Comput. Inf. Syst. 58 (4) (2018) 325–333, https://doi.org/10.1080/08874417.2016.1261377.
[11] R. Dodder, R. Dare, Complex adaptive systems and complexity theory: inter-related knowledge domains, *ESD. 83: Res. Seminar Eng. Syst.*, MIT (2000) 14 [Online]. Available, http://web.mit.edu/esd.83/www/notebook/ComplexityKD.PDF.
[12] N. Kratzke, P.C. Quint, Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study, J. Syst. Softw. 126 (2017) 1–16, https://doi.org/10.1016/j.jss.2017.01.001.
[13] C. Rosenthal, N. Jones, Chaos engineering system resiliency in practice, O'Reilly Media (2020).
[14] B. Scholl, T. Swanson, P. Jausovec, Cloud native: using containers, functions, and data to build next-generation applications, O'Reilly Media, Inc (2019).
[15] D. Gannon, R. Barga, N. Sundaresan, Cloud-native applications, IEEE Cloud Comput 4 (5) (2017) 16–21, https://doi.org/10.1109/MCC.2017.4250939.
[16] L. Liu, Services computing: from cloud services, mobile services to internet of services, IEEE Trans. Serv. Comput. 9 (5) (2016) 661–663, https://doi.org/10.1109/TSC.2016.2602898.
[17] T. Welsh, E. Benkhelifa, On resilience in cloud computing, ACM Comput. Surv. 53 (3) (2020) 1–36, https://doi.org/10.1145/3388922.
[18] T.M. Tawfeeg, et al., Cloud dynamic load balancing and reactive fault tolerance techniques: a systematic literature review (SLR), IEEE Access 10 (2022) 71853–71873, https://doi.org/10.1109/ACCESS.2022.3188645.
[19] D. Hillson, Beyond resilience: towards antifragility? Contin. Resil. Rev. (2023) https://doi.org/10.1108/CRR-10-2022-0026.
[20] A. Tolk, Implementing antifragiles: systems that get better under change, in: International Annual Conference of the American Society for Engineering Management 2013, 2013, pp. 118–126. *ASEM 2013*.
[21] M. Monperrus, Software that learns from its own failures, ArXiv (2015) abs/1502.0 [Online]. Available, http://arxiv.org/abs/1502.00821.
[22] K.H. Jones, Engineering antifragile systems: a change in design philosophy, Procedia Comput. Sci. 32 (2014) 870–875, https://doi.org/10.1016/j.procs.2014.05.504.
[23] J. Allspaw, Fault injection in production, Commun. ACM 55 (10) (2012) 48–52, https://doi.org/10.1145/2347736.2347751.
[24] M.A. Naqvi, S. Malik, M. Astekin, L. Moonen, On evaluating self-adaptive and self-healing systems using chaos engineering, in: *Proceedings - 2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2022*, IEEE, 2022, pp. 1–10, https://doi.org/10.1109/ACSOS55765.2022.00018.
[25] A. Pierce, J. Schanck, A. Groeger, R. Salih, M.R. Clark, Chaos engineering experiments in middleware systems using targeted network degradation and automatic fault injection, in: *Open Architecture/Open Business Model Net-Centric Systems and Defense Transformation2021*, SPIE, 2021, p. 8, https://doi.org/10.1117/12.2584986.
[26] C.S. Meiklejohn, A. Estrada, Y. Song, H. Miller, R. Padhye, Service-level fault injection testing, in: SoCC 2021 - Proceedings of the 2021 ACM Symposium on Cloud Computing, 2021, pp. 388–402, https://doi.org/10.1145/3472883.3487005.
[27] A. Al-Said Ahmad, P. Andras, Scalability resilience framework using application-level fault injection for cloud-based software services, J. Cloud Comput. 11 (1) (2022) 1–13, https://doi.org/10.1186/s13677-021-00277-z.
[28] J. Simonsson, L. Zhang, B. Morin, B. Baudry, M. Monperrus, Observability and chaos engineering on system calls for containerized applications in Docker, Futur. Gener. Comput. Syst. 122 (2021) 117–129, https://doi.org/10.1016/j.future.2021.04.001.
[29] B.C.I. KOSTENKO, Antifragile Microservice Systems, Masaryk University, 2023 [Online]. Available, https://is.muni.cz/th/w3tej/.
[30] B. Rossi, "Antifragile microservice systems, supervisor's review," 2023. [Online]. Available: https://is.muni.cz/th/w3tej/posudek_vedouciho_Rossi.pdf.
[31] G. Liu, B. Huang, Z. Liang, M. Qin, H. Zhou, Z. Li, Microservices: architecture, container, and challenges, in: *2020 IEEE 20th international conference on software quality, reliability and security companion (QRS-C)*, IEEE, 2020, pp. 629–635, https://doi.org/10.1109/QRS-C51114.2020.00107.
[32] Y. Gan, M. Liang, S. Dev, D. Lo, C. Delimitrou, Practical and scalable ML-driven cloud performance debugging with sage, IEEE Micro (2022) 27–36, https://doi.org/10.1109/MM.2022.3169445.
[33] C. Ma and M. Ranney, "Failure mitigation for microservices: an intro to aperture." Accessed: Nov. 09, 2023. [Online]. Available: https://doordash.engineering/2023/03/14/failure-mitigation-for-microservices-an-intro-to-aperture/.
[34] M.A. Shahid, N. Islam, M.M. Alam, M.S. Mazliham, S. Musa, Towards Resilient Method: an exhaustive survey of fault tolerance methods in the cloud computing environment, Comput. Sci. Rev. 40 (2021) 100398, https://doi.org/10.1016/j.cosrev.2021.100398.
[35] J. Liu, S. Zhang, Q. Wang, J. Wei, Coordinating fast concurrency adapting with autoscaling for SLO-oriented web applications, IEEE Trans. Parallel Distrib. Syst. 33 (12) (2022) 3349–3362, https://doi.org/10.1109/TPDS.2022.3151512.
[36] A. Brogi, J. Carrasco, F. Durán, E. Pimentel, J. Soldani, Self-healing trans-cloud applications, Computing (2022) 1–25, https://doi.org/10.1007/s00607-021-00977-z.
[37] P. Zoghi, M. Shtern, M. Litoiu, H. Ghanbari, Designing adaptive applications deployed on cloud environments, ACM Trans. Auton. Adapt. Syst. 10 (4) (2016) 1–26, https://doi.org/10.1145/2822896.
[38] A. Abid, M.T. Khemakhem, S. Marzouk, M. Ben Jemaa, T. Monteil, K. Drira, Toward antifragile cloud computing infrastructures, Procedia Comput. Sci. 32 (2014) 850–855, https://doi.org/10.1016/j.procs.2014.05.501.
[39] D. Anderson, "What is APM? Application performance monitoring in a cloud-native world." Accessed: Oct. 10, 2023. [Online]. Available: https://www.dynatrace.com/news/blog/what-is-apm-2/.
[40] Z. Flower, "5 benefits of APM for businesses." Accessed: Jun. 20, 2023. [Online]. Available: ttps://www.techtarget.com/searchapparchitecture/feature/Learn-the-benefits-of-APM-software-in-the-enterprise.
[41] "Prometheus." Prometheus. [Online]. Available: https://prometheus.io/.
[42] N. Kratzke, Cloud-native observability: the many-faceted benefits of structured and unified logging—a multi-case study, Futur. Internet 14 (10) (2022) 274, https://doi.org/10.3390/fi14100274.

[43] R. Rai, "Automatic instrumentation of containerized .NET applications with OpenTelemetry." Accessed: Jul. 20, 2023. [Online]. Available: https://www.twilio.com/blog/automatic-instrumentation-of-containerized-dotnet-applications-with-opentelemetry.
[44] Z.T. Kalbarczyk, R.K. Iyer, S. Bagchi, K. Whisnant, Chameleon: a software infrastructure for adaptive fault tolerance, IEEE Trans. Parallel Distrib. Syst. 10 (6) (1999) 560–579, https://doi.org/10.1109/71.774907.
[45] "Toxiproxy." Shopify. [Online]. Available: https://github.com/Shopify/toxiproxy.
[46] "NBomber." NBomber. [Online]. Available: https://nbomber.com/docs/getting-started/overview/.
[47] J. Botros, "Defragile." GitHub, 2023. [Online]. Available: https://github.com/josephwasily/Defragile.
[48] "cAdvisor." Google. [Online]. Available: https://github.com/google/cadvisor.
[49] "Grafana." Grafana. [Online]. Available: https://grafana.com/docs/grafana/latest/dashboards/.
[50] V. Kumar, "Handling overload with concurrency control and load shedding — part 2." Accessed: Oct. 15, 2023. [Online]. Available: https://vikas-kumar.medium.com/handling-overload-with-concurrency-control-and-load-shedding-part-2-6b8b594d4405.
[51] D. Yanacek, "Using load shedding to avoid overload," Amazon Web Services. Accessed: Jul. 20, 2023. [Online]. Available: https://aws.amazon.com/builders-library/using-load-shedding-to-avoid-overload/.
[52] "The Polly Project." The Polly Project, 2019. [Online]. Available: https://thepollyproject.azurewebsites.net/.
[53] Netflix Technology Blog, "Performance under load." Accessed: Jul. 20, 2023. [Online]. Available: https://netflixtechblog.medium.com/performance-under-load-3e6fa9a60581#.
[54] D. Kleiman, "Adaptive concurrency control for mixed analytical workloads." Accessed: Jul. 20, 2023. [Online]. Available: https://klaviyo.tech/adaptive-concurrency-control-for-mixed-analytical-workloads-51350439aeec.
[55] Q.-M. Nguyen, "Gitaly adaptive concurrency limit." Accessed: Jun. 20, 2023. [Online]. Available: https://docs.gitlab.com/ee/architecture/blueprints/gitaly_adaptive_concurrency_limit/.
[56] "Backpressure." Camunda. Accessed: Jul. 20, 2022. [Online]. Available: https://docs.camunda.io/docs/self-managed/zeebe-deployment/operations/backpressure/.
[57] Netflix, "Netflix concurrency limits." Netflix /GitHub, 2023. [Online]. Available: https://github.com/Netflix/concurrency-limits.
[58] C. [Cloud N. C. Foundation], "Envoy, take the wheel: real-time adaptive circuit breaking - Tony Allen, Lyft." Accessed: Jul. 20, 2023. [Online]. Available: https://www.youtube.com/watch?v=CQvmSXlnyeQ.
[59] J.D.C. Little, A proof for the queuing formula: L= λ W, Oper. Res. 9 (3) (1961) 383–387.
[60] T. Dykstra, et al., Background tasks with hosted services in ASP.NET Core, Microsoft (2024). AccessedJun. 09[Online]. Available, https://learn.microsoft.com/en-us/aspnet/core/fundamentals/host/hosted-services?view=aspnetcore-8.0&tabs=visual-studio.
[61] N.N. Taleb, A map and simple heuristic to detect fragility, antifragility, and model error, SSRN Electron. J. (2012), https://doi.org/10.2139/ssrn.1864633.