St. Cloud State University

The Repository at St. Cloud State

Culminating Projects in Computer Science and Information Technology

Department of Computer Science and Information Technology

5-2023

Chaos Engineering for Microservices

Arunkumar Akuthota

Follow this and additional works at: https://repository.stcloudstate.edu/csit_etds

Part of the Computer Sciences Commons

Recommended Citation

Akuthota, Arunkumar, "Chaos Engineering for Microservices" (2023). *Culminating Projects in Computer Science and Information Technology*. 42.
https://repository.stcloudstate.edu/csit_etds/42

This Starred Paper is brought to you for free and open access by the Department of Computer Science and Information Technology at The Repository at St. Cloud State. It has been accepted for inclusion in Culminating Projects in Computer Science and Information Technology by an authorized administrator of The Repository at St. Cloud State. For more information, please contact tdsteman@stcloudstate.edu.

Chaos Engineering for Microservices

by

Arunkumar Akuthota

A Starred Paper

Submitted to the Graduate Faculty of

St. Cloud State University

in Partial Fulfillment of the Requirements

for the Degree

Master of Science in

Computer Science

May 2023

Starred Paper Committee:
Akalanka B. Mailewa, Chairperson
Aleksandar Tomovic
Mark B Schmidt
Sam E Espana Lopez

Abstract

Chaos engineering is a relatively new concept that is growing in popularity as it helps companies to be more resilient in the face of unexpected networking or software failure. The idea behind chaos engineering is that if you can create controlled failures, you can discover where your system is weak and then fix those weaknesses before something happens to your production environment. This research has been done on microservices, which are small pieces of code that perform specific tasks on behalf of a larger application. Microservices are often hosted on different servers and run by different teams, so they are much more fragile than monolithic applications. Microservices also tend to be written in different languages, which makes them more difficult to understand and test for bugs. The goal of this study was to determine whether microservices can be made more resilient through chaos engineering or not; specifically, if it is possible to find out what kinds of failures occur most often and how long they take to resolve.

Table of Contents

List of Figures

Chapter I: Chaos Engineering for Microservices

Chaos engineering is a discipline that aims to make software more resilient by testing systems in production. It is used to identify weaknesses and vulnerabilities in technology infrastructure, so that they can be fixed before they cause problems (Kutlu). The microservice system is a small independent system that is based on the organizational and architectural approach to the development of software and that will communicate well with defined APIs. The use of microservices helps developers to become technological and language-agnostic. There are different programming languages used by different team members for coding and debugging. Microservices monitor help in architecture check for performance and service for identifying the problems in future debugging. There are some problems faced while using microservices as these are too certain in degradations of the environment and failure of services. Some failure testing scenarios occurred while working with microservices such as errors in communication like time-outs. Microservices also face challenges while working with distributed applications, the only solution to provide the resilience of microservices is making distributed applications with the use of chaos engineering (Naqvi et al.). The research has been made to analyze the implementation of chaos engineering microservices and how they can be improved.

Chapter II: Background

Microservices are a popular development pattern, which has been adopted by many companies. The primary benefit of adopting microservices is that it allows companies to break down their software into smaller components, each of which can be developed and deployed independently. However, there are also several drawbacks to adopting microservices. One such drawback is that it becomes difficult for developers to maintain an overview over the entire software application due to its large size and complexity (De 289-294). This leads to errors in the software code and thus increases the risk of bugs and unexpected behavior. Another drawback is that microservices are often deployed on different servers, which makes it difficult for developers to test them together as part of a single system. In order to address these problems, chaos engineering was introduced as a way of monitoring how well applications can withstand unexpected failures or changes in their environment without breaking down entirely. Chaos engineering provides a framework for testing how resilient an application is by simulating failures like network outages or server crashes within the environment where they occur most often (e.g., production) (Björnberg).

Chapter III: Aim and objectives.

Aim

The aim of this research is to explore the effectiveness of chaos engineering in microservice architecture.

Objectives

- To understand the importance of chaos engineering for microservices.
- To describe the types and categories of applications which can be used to implement chaos engineering for microservices.
- To understand the chaos engineering with cloud traffic control.
- To understand the benefits and drawbacks of implementation of chaos engineering for microservices.
- To learn about the different chaos engineering applications executed by the organizations.

Chapter IV: Research Questions

1. What is the impact of implementing Chaos engineering on resiliency of Microservice Architecture systems?

2. What are the Benefits and Drawbacks of applying Chaos engineering on Microservices architecture-based systems?

3. What are the best tools for implementing chaos engineering in microservices?

Chapter V: Research methodology

In this research, qualitative research method and systematic literature review was applied for the research Chaos engineering for microservices. The aim of this study is to improve the understanding of chaos engineering for microservices in order to determine its usefulness and to provide recommendations for future research. The qualitative research method was used because it allows the researcher to gain a deeper understanding of the phenomenon under investigation, while systematic literature review was used because it provides a broad overview of related work in the field. The research results show that chaos engineering is an effective tool that can be used to improve an organization's readiness when it comes to managing incidents and mitigating risks. However, further studies need to be conducted before chaos engineering can be widely adopted by organizations.

Chapter VI: Literature Review

Chaos engineering is a method for testing software systems and their infrastructure. It involves injecting failure into the system to see how it responds, and then using that information to improve it. Chaos engineering is a new practice in the field of software engineering. It was created as an alternative to traditional testing methods and aims to help companies create more resilient systems by using controlled failure (Wang 63-66). Chaos engineers aim to break down complex systems into smaller parts, which makes them easier to understand and predict. They also try to find areas where they can predict failure before it happens, so they can work on preventative measures. The term "microservices" refers to a software architecture style that divides large programs into small, independent services. The most important benefit of this approach is that it allows individual services to be developed, deployed, and scaled independently. Other benefits include increased scalability, reliability and flexibility (Kesim).

Chaos engineering is important for microservices because it helps to ensure that the services that comprise a microservices architecture can withstand failure, and therefore, adapt to change. Microservices are made up of a number of components that each have their own functionality. They often have to communicate with one another in order to function properly, which means that if one of these components becomes unavailable or malfunctions, it can cause other components to fail as well (Yin et al. 147-171). This is why chaos engineering is so key: it allows us to identify problems with our systems before they become issues, so that we can fix them before they become disasters. Chaos engineering helps organizations to understand how their system will behave under stressors like increased load or faulty hardware; this lets us make sure that our systems can continue functioning even when faced with these kinds of issues.

Microservices are a popular solution to the problem of scalability and consistency in an enterprise setting. They allow organizations to scale each part of their application independently, which makes it easier to keep up with customer demand. However, microservices architecture can lead to problems when the pieces don't work together well (Jones et al.). Chaos engineering is a technique for testing the resiliency of their application against unpredictable failures. It involves intentionally introducing random failures into their system in order to see how it reacts. It helps organizations to identify weaknesses in such a system before they cause problems for customers. Microservices are a great way to keep the application from becoming too monolithic, but they can pose challenges in terms of debugging and deployment. When organizations have many small pieces, it is easier for each one to go wrong and harder to find the problem when it does. This is where chaos engineering comes in: it helps organizations to test microservices so that when something breaks, they can actually find out what went wrong. Chaos engineering involves injecting failures into their system, then seeing how that affects other parts of an organization's infrastructure. Organizations can do this by simulating a failure or failure mode in one component (e.g., a database), then observing how other components respond to that failure. For example, if the database fails because someone deleted some data, but their application is still running smoothly, then it means that the database failure was isolated enough not to impact anything else—which means that for now, it can be ignored until further notice (Jamshidi et al. 24-35). Chaos engineering can also help organizations to prevent future outages by showing what happens when certain parts of their infrastructure fail. So, if a database fails because someone deleted some data, and it causes an error in their application's API layer, then someone know that this is something to fix before it becomes a problem.

Chaos engineering is a new approach to system design and testing that involves creating controlled failures in a production environment, with the goal of discovering unexpected behaviors and improving resiliency. The idea of chaos engineering is to introduce failure into a system and observe how it behaves in order to gain insights about what could go wrong. This allows organizations to identify any potential issues before they happen and make changes to organization infrastructure or codebase so that they do not cause problems in their production environment. Chaos engineering has been used in companies like Netflix, Google, Facebook, and Amazon for years now, but it is only recently gained traction among smaller companies—and even then, only as an option for those who want it (Wang 63-66). A big reason for this is cost: implementing chaos engineering is not cheap. It requires hiring people who are knowledgeable about how to implement it properly (which is not always easy), as well as spending time developing test cases and running them against their infrastructure.

What is the impact of implementing chaos engineering on resiliency of microservice architecture systems?

Software systems may be tested using chaos engineering by deliberately triggering undesirable outcomes, including service interruptions or API limits. Here, we intentionally subject the system to a variety of failure circumstances in order to evaluate how well it copes with chaos. It also aids teams in simulating real-world situations, which are essential for discovering the hidden problems, monitoring blind spots, and performance bottlenecks that are otherwise hard to locate in distributed systems. The strategy is very useful for avoiding production delays or interruptions before they happen. Chaos engineering was pioneered by Netflix, and it involves simulating service outages by unexpectedly terminating instances of various systems. Live production systems were used for these so-called chaos experiments since only their actual traffic existed and real-world conditions offered accurate data on the architecture's robustness. The field of Chaos Engineering is growing in both popularity and sophistication. Although Chaos Engineering's originators, like Netflix and Amazon, are forward-thinking behemoths, the practice has now found favour with more traditional businesses and smaller teams.

It is necessary to know how the program is supposed to act before beginning chaos engineering.

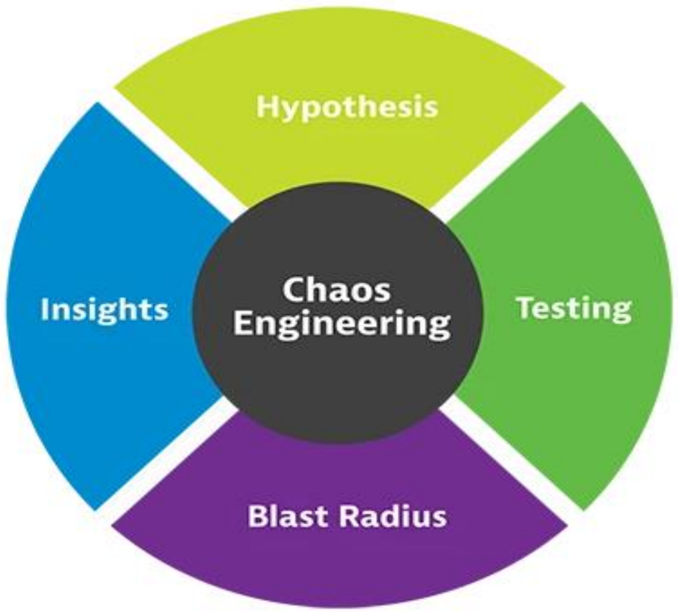


Fig. 1. Chaos Engineering.

Hypothesis: In engineering, one often asks oneself "what if?" while attempting to predict the results of a change. If they cut off service intermittently, they expect it to keep working. The hypothesis may be broken down into two parts: the query and the assumption.

Testing: Chaos engineers create situations of simulated uncertainty and load test the system, looking for disruptions in the application's supporting services, infrastructure, networks, and endpoint devices to verify their hypotheses. The hypothesis is invalidated by the occurrence of any failures in the stack.

Blast radius: As a result of isolating and analyzing errors, engineers may learn what goes wrong in unsteady cloud environments. The blast radius refers to the area around the test site that was affected by the explosion. Engineers specializing in chaos may regulate the range of explosions by conducting the experiments themselves.

Insights: New software and microservices will be more resilient to unforeseen circumstances because of the findings, which will be included into the software development and delivery process.

One sluggish service may easily drive-up latency for a whole microservices architecture. In reality, in today's world of microservice design and ecosystems, we have gone from a monolithic system's single point of failure to a distributed system's many points of failure. We need alternative approaches to testing in order to develop systems that are scalable, highly available, and dependable (Jamshidi et al. 24-35).

A. By using Chaos Engineering, a system becomes more robust.
B. By planning and carrying out Chaos Engineering experiments, we learn where our system has vulnerabilities that might lead to disruptions and, ultimately, a loss of consumers. This aids in the enhancement of incident response.
C. By revealing potential risks, it aids in better appreciating the system's overall precariousness.

The engineering time and resources needed to execute chaos engineering and reduce the damage produced by both deliberate and inadvertent actions are additional costs associated with adopting this approach. Because of this, many IT organizations, when initially presented with the concept, focus only on the dangers of chaos engineering rather than considering its possible advantages. Because of this, it is crucial to carefully consider the potential costs and benefits of chaotic engineering before deciding to use it. However, there are several upsides to using chaotic engineering. From the perspective of the consumers, the service's availability and dependability will increase, and the frequency of service disruptions will reduce. Engineers can cut down on avoidable disturbances to the system with the help of chaos testing, and the development team can focus more on creating backup and recovery logics. In general, this strategy has several beneficial benefits on ROI for businesses.

Resilience:

It's simpler than ever to create widespread apps using distributed components. There are many open source and cloud-hosted components and services to use as a foundation for your projects, and a wide variety of programming languages are supported. System dependability is not guaranteed, and neither are the underlying components and dependencies. In the event of an outage or interruption in service, or if infrastructure goes down, this might happen at any moment. It's uncommon for even seemingly little changes to one part of the system to have far-reaching consequences in another.

Applications and services must anticipate and account for system-wide failures, interruptions in known and unknown dependencies, abrupt, unexpected demand, and latencies. It is essential that applications and services be built to recover from and resist outages and interruptions. Resilient applications and services may smoothly recover from failures and continue functioning. While the dependability of its parts is important, the system as a whole must also be able to withstand shocks and recover quickly. Resilience testing must be performed in a fully integrated, production-like environment simulating the actual circumstances and load the system would experience (Heorhiadi et al. 57-66).

Since Netflix started migrating away from on-premises data centers and into the cloud in 2008, we have routinely performed some type of resilience testing in live environments. The term "Chaos Engineering" didn't even exist when we first started using it. The ball was set in motion by

Chaos Monkey, which became notorious for disrupting manufacturing services. Those advantages, previously only applicable on a microscale, were amplified by Chaos Kong and made available to those operating on a global scale. Failure Injection Testing (FIT) established a methodology for addressing the in-between. Our Chaotic Automation Platform takes the discipline one step further by allowing for continuous chaos experimentation throughout the whole microservice architecture, a goal originally envisioned by the Principles of Chaos. It was only when we gained knowledge and practice with these tools that we understood that Chaos Engineering isn't about intentionally breaking a service. Though breaking things is simple, it's not necessarily useful. When working on complicated systems, "Chaos Engineers" seek to bring to the surface the underlying chaos that exists inside them. Engineering in distributed systems may be improved via a deeper understanding of systemic impacts.

Microservice Architectures:

Microservice architectures have been gaining traction, with adopters like Amazon and Netflix, as a means to circumvent the difficulties associated with using monolithic programs in the context of cloud computing. Microservice architecture is an alternative to the traditional monolithic method to software development in which a single service is responsible for providing all of an application's functionality. Then, messages are sent between the different services. Independent teams of developers may utilize the technologies and programming languages that are most suited to the needs of a given use case while working on microservices. Furthermore, the microservice's deployment, scaling, and operation are all handled independently from other microservices, allowing for the optimal use of server types (high CPU, high memory, etc.) and scaling rules (Th¨ones 116-116). The design also facilitates the adoption of continuous delivery methodologies, since new deployments do not interfere with the functioning of other microservices, and their operation may continue without interruption. Microservice architectures are widely used by service delivery teams nowadays. Developers benefit from using microservices because it allows them to work with a smaller, more focused codebase and have more control over when and how their service is deployed. If you're considering switching from a monolith, there are some significant benefits to consider.

In conclusion, businesses may gain agility, decrease complexity, and scale their applications effectively in the cloud by adopting a microservice architectural style. Although the microservice architectural style eliminates the hazard of a single point of failure, it introduces others. Because each microservice is independent and uses the network to communicate, calls between them are vulnerable to disruption at any moment. As such, it is crucial to anticipate and gracefully deal with service failures by adhering to the design for failure concept.

Strategies for Building Resilient Microservices:

Many companies are switching from monolithic to microservices architecture to keep up with the rapid pace of technological change. This is due to the fact that an individual mistake will not result in the whole failure of the application. But does it mean your Microservices Architecture can withstand failures? It's not unusual for applications to have problems and glitches when they're being created. Failure is inevitable in a microservice ecosystem, thus it's better to accept it now than later. Microservices should be built with the possibility of failure in mind. This means your microservices architecture must be robust. Resilience is the capacity of an application to bounce back after malfunctions. It is important to consider the number of distributed services and how to make microservices robust while designing and building them.

Resilient Designs for Microservices:

There are three widely used methods of increasing fault tolerance and making applications more resilient to disruptions. Tolerating the failure of individual parts while maintaining overall system functionality is what we call fault tolerance (Zhou et al. 243–260).

Retry Pattern:

Microservices often rely on external resources like databases, modules, back-end services, and APIs. Service calls may fail if any of these components ever stop working. Retrying may fix these temporary failures. For both regular and unexpected breakdowns, the retry pattern implements a mechanism that tries to run the failed operation again and again until it succeeds. The retry count and timeouts are both at the control of IT administrators. Instead of immediately shutting down, failing services may now try to contact healthy ones many times until they get the expected answer.

Circuit Breaker Pattern:

The retry approach works well for short-term issues, but teams still need a reliable microservices resilience strategy for handling more severe, persistent problems. For example, if a retry mechanism repeatedly calls a badly damaged service until it gets the desired result, it might produce cascading service failures that are harder to detect and fix. The component created from the circuit breaker pattern resembles a standard electrical switch. This section is located between the services' terminals and the requests for such services.

If regular communication is maintained between these services, the circuit breaker will transmit information between them in a secure manner. After a certain number of unsuccessful retries, the breaker opens the message circuit, putting an end to service operations. If the circuit is open, the service is halted, and error prompts are sent to the client service for each unsuccessful transaction. After a certain length of time, the circuit breaker switches to a semi-open position (known as the circuit reset timeout). To test whether connections have been restored between the two services, the breaker calls will close the loop during this period. After sensing a single failure, the breaker will reset to the open position. As soon as the issue is fixed, the loop re-closes as usual.

Timeout Design Pattern

You've certainly heard of a timeout in football, but in Microservices it means waiting for a certain length of time before moving on to the next step. A complete transaction, from initial connection through receiving the last response byte, may be timed out. The SO TIMEOUT function won't work with this, unfortunately. When we use the OkHttp or JDK11 clients, we may avoid this problem.

What are the benefits and drawbacks of applying chaos engineering on microservices architecture-based systems?

The complexity of a microservice design exceeds that of a traditional system. Due to the increased number of components in a microservice context, management and upkeep become more labor intensive. Putting your apps and distributed systems through stress tests may provide a wealth of useful data for your development teams and business. Some of the advantages of using chaos testing tools in Chaos Engineering projects are listed below.

Benefits:

Increases Reliability and Resiliency: using a Chaotic Engineering tool to do controlled chaos experiments strengthens the system by putting its capabilities to the test. It's important to carefully choose measures and make educated guesses about the steady state before beginning these kinds of chaotic investigations. It is recommended that the first Chaos Engineering tests be conducted in staging or another pre-production phase when the explosion radius will be modest. This safeguards users from any potential adverse effects.

Increases End User and Stakeholder Satisfaction: You may execute tests close to production once you and your team build trust in Chaos Engineering, which increases the satisfaction of both end users and stakeholders. In a perfect system, all trials would be run using the same data that would be used in the production setting. The production environment simulates the live system, so any testing done there will give you a good understanding of how your final consumers will interact with the product. Reduced network outages and service interruptions mean a better overall system for end users (Naqvi et al.).

Advances Team Collaboration and Confidence: The insights gained from these chaotic trials strengthen the engineering team's expertise, which in turn boosts collaboration and confidence. This leads to quicker reactions, better teamwork, and higher levels of self-assurance. These learnings may then be utilized to train junior staff.

Improves Incident Response Time: Quicker troubleshooting, repairs, and reactions to incidents are possible as a result of the technical team's increased awareness and familiarity gained from prior chaotic experiments, which improves incident response time. Therefore, the knowledge gained through chaotic testing might lessen the occurrence of events in production. Response times may be sped up with the help of game days. The goal is to build time into your process for the team to go through potential emergency scenarios.

Enhances Performance Status Reporting for Applications: Chaos testing is widely regarded as one of the most all-encompassing methods to performance engineering and testing procedures. Conducting chaotic experiments on a regular basis helps build trust in distributed systems and ensures that programs continue to function properly in the event of catastrophic failure.

Business Benefits: Using chaos engineering, businesses may avoid costly disruptions that might otherwise result in significant revenue losses. This method also allows businesses to expand rapidly without compromising the quality of their offerings.

Technical Benefits: While the incident rate may be lowered thanks to the results of chaotic experiments, that's not the end of the technological advantages. Having a deeper understanding of system modes and dependencies helps the team create a more reliable system. The engineering staff may benefit greatly from the on-call preparation provided by a chaotic test.

Customer Benefits: Less downtime means less hassle for your customers. Customers gain most from Chaos Engineering's increased service reliability and availability.

A. Lessons learned through chaotic testing may help you prevent future problems in production.

B. Chaos Engineering allows the team to test how the system responds to failures, allowing them to adjust their strategy as needed.

C. As an aid to testing the team's reaction to the crisis, Chaos Engineering is a useful tool. This is useful for verifying that the appropriate group was alerted once an alarm has been raised.

D. At the most fundamental level, Chaos Engineering gives us an edge via increased system uptime. The system's ability to recover from errors is improved via chaos experiments.

E. Companies might lose a lot of money due to production outages depending on how they use the system, but chaos engineering can help them avoid that fate by bolstering employee trust in disaster recovery plans and increasing their investment in the success of such plans.

F. Accelerate innovation is the results of chaos testing are sent back to the development team so that they may make modifications to the software's architecture to make it more robust and increase the quality of their output.

One typical response to a paper on Chaos Engineering goes something like this: "Gee, that sounds very fascinating, but our software and our organization are both entirely different from Netflix, so this stuff simply wouldn't apply to us." While we use Netflix as a case study, the ideas discussed here are applicable to any company, and our approach to experiment design does not presuppose any specific technology stack or collection of tools. Whether you want to know if, why, when, and how you should implement Chaos Engineering methods, you'll find all that and more in Chaos Maturity Model, where we explore and go deeply into the Chaos Maturity Model.

Google, Amazon, Microsoft, Dropbox, Yahoo!, Uber, cars.com, Gremlin Inc., University of California, Santa Cruz, SendGrid, North Carolina State University, Sendence, Visa, New Relic, Jet.com, Pivotal, ScyllaDB, GitHub, DevJam, HERE, Cake Solutions, Sandia National Labs, Cognitect, Thoughtworks, and O'Reilly were just some of the companies represented at the most recent Chaos Community Day. This book is filled with real-world applications of Chaos Engineering from a wide variety of fields, including but not limited to the financial sector, the internet commerce sector, the aviation industry, and more.

Large financial organizations, manufacturers, and healthcare providers are just a few examples of non-digital native businesses that make considerable use of Chaos Engineering. Are financial dealings dependent on your elaborate system? Chaos Engineering is used by major financial institutions to ensure that their transactional systems are redundant. Is there a risk to human life? Clinical trials are the tried-and-true method for verifying the efficacy of new medical treatments in the United States, and Chaos Engineering takes many cues from this approach.

Financial, medical, and insurance organizations, as well as manufacturers of rockets, agricultural equipment, and tools, IT behemoths, and fledgling businesses all see the value in Chaos Engineering.

Characteristics of a Chaos Test:

While some may believe that releasing code into production and watching what occurs is chaos engineering, this is not the case. You need an advanced level of maturity in your infrastructure to get started. There are universal qualities shared by all chaos tests; your infrastructure needs to be sufficiently developed to support them to conduct chaos testing.

A. *Confidence:* You should never try out a feature in production if there's even a little chance it may fail. First, the chaotic test is tried out in a staging environment. If the problem arises there, you must address it. You should only launch it into production if you have complete faith that it will function as intended.

B. *Risk is contained:* It is common practice to split your traffic sample into a test group, which will have the problem introduced, and a control group, whose monitors will be compared to those of the test group. Perhaps 1% of your traffic is assigned to the test group, 1% to the control monitoring group, and the other 99% to the status quo. This dispels the common misconception that a chaos test just involves temporarily making a node unavailable. Instead, it is going dark for a small percentage (say, 0.5%) of its user base.

C. *Hypothesis:* Using KPIs, your tests should always contain a hypothesis. In the case of an online store, for instance, the theory may be that if the product recommendation service takes longer than 200 milliseconds to react, the system would simply stop suggesting things while leaving all other functionality untouched.

D. *Failure criteria:* Set an early cutoff point for the exam and stick to it. If the experimental group's average reaction time is more than 10% longer than the control groups, for instance, or if the experimental group records 1% more mistakes than the control group, the experiment will be terminated (Poltronieri et al.).

E. *Monitoring:* To promptly cancel the test, you need a well-developed monitoring infrastructure to keep an eye on the system's condition throughout the experiment. The importance of dispersed tracing has been hammered home by Netflix in particular. That's connected to the earlier idea about keeping things in. The experimental group should ideally include no more than 0.5 percent of all site visitors. Assigning half of your customers to the test group (and the other half to the control group) and then tracking each customer's calls throughout the system to ensure that all calls spawned by that customer remain in the appropriate group is necessary if you want to test with only 5% of your customers (Simonsson et al. 117-129).

F. *Automation:* Using continuous integration/continuous deployment scripts, the test may be run automatically. Automation also allows for instantaneous test termination in the event of an error. At the highest level of development, you never bother the personnel on call to run anything except well approved chaos tests in production. When these tests discover the failure conditions, they immediately terminate and send out a warning. Containers or a public cloud are often required to automate such infrastructure migrations. It might be difficult, if not impossible, to automate chaos testing on non-cloud virtual machines or bare metal.

It's challenging to get to that point of maturity, which is perhaps why many businesses haven't completely embraced it just yet. You can see from the above that it demands a level of public cloud or container, CI/CD, and end-to-end monitoring maturity that many firms wish to have for many reasons besides chaotic engineering but do not currently have.

However, like with any IT investment, even the most advanced degree of maturity may not be worth it in your specific case. Similar to how many organizations are deconstructing their monoliths without fully adopting microservices, many organizations see benefit in using a hybrid approach. For instance, one organization I spoke with conducted much more chaotic testing in a production-like acceptance environment than in actual production. They were satisfied with the results and decided there was no need to further develop their production chaos testing infrastructure.

Drawbacks:

A. The implementation of Chaos Monkey on a big scale, together with the associated testing, may result in additional expenses.

B. The application might be negatively affected by carelessness or incorrect actions in its creation and execution, which would have a negative effect on the consumer.

C. Unfortunately, there is no Interface provided to keep tabs on the project while it is being implemented. Scripts and settings files are processed.

D. Not all deployment types are supported.

Challenges of Microservices Architecture:

The complexity of a microservice design exceeds that of a traditional system. Due to the increased number of components in a microservice context, management and upkeep become more labor intensive. The following are some of the most significant obstacles that businesses encounter while adopting microservices:

A. Bounded Context

B. Dynamic Scale up and Scale Down

C. Monitoring

D. Fault Tolerance

E. Cyclic dependencies

F. DevOps Culture

Bounded context: The idea of a bounded context was developed in the framework of Domain-Driven Design (DDD). It advocates the Object model first strategy, which establishes a data model for which service is accountable and has a contractual obligation. A bounded context outlines the scope of the model's obligation and helps to ensure that it is met. It makes sure that outside influences won't disrupt the domain. Every model operates in a certain sub-domain, which entails an implicitly determined context. To rephrase, the service is the only custodian of the data and has the exclusive authority to make changes or delete it. The most crucial aspect of microservices is supported, making this a great tool (Escobar).

Dynamic scale up and scale down: Microservices may be running at a different instance of the type depending on the demand. Your microservice needs to automatically scale up as well as down. The price of the microservices is lowered as a result. The burden may be actively shared among us.

Monitoring: Because we now have numerous services making up the same functionality formerly supplied by a single application, the conventional approach to monitoring will not work well with microservices. Finding the reason for an application issue is often difficult.

Fault Tolerance: When one component of a system fails, it does not bring down the rest of the system. When the failure happens, the application may still function to some extent. A complete system failure may occur if the system does not have fault tolerance. With this circuit breaker, it is possible to build in fault tolerance. The circuit breaker design encapsulates calls to external services and flags problems when they occur. Microservices must be able to recover from failures both inside and outside of their own infrastructure.

Cyclic Dependency: The management of inter-service dependencies and their functioning is crucial. In the absence of timely attention, cyclic dependence might become an issue.

DevOps Culture: DevOps is a natural home for microservices. It allows for quicker service delivery, more data visibility, and lower data costs. It allows them to migrate from SOA to Microservice Architecture while retaining the benefits of containerization (MSA).

A. We need to ensure that all our microservices can grow in tandem as we add more of them. Granularity increases complexity because it introduces additional variables.
B. Because microservices are stateless, distributed, and self-sufficient, conventional logging approaches are useless. Events on different systems must be correlated in the logging.
C. Failure rates tend to rise in tandem with the number of interdependent services.

What are the best tools for implementing chaos engineering in micro services?

For several reasons, including the need to speed up deployments, businesses are favoring cloud-native deployments (i.e., those based on Kubernetes) over more conventional approaches. Since there are more potential points of failure in cloud-native systems than in conventional deployments, site reliability engineers (SREs) and development teams must now adapt to this new reality. Downtime that isn't anticipated may have serious consequences for a company's bottom line, reputation, and brand. The rising costs of unscheduled downtime and this rise in system-level complexity have pushed the testing of cloud-native systems into the spotlight. To assist teams, provide more dependable systems, chaos engineering offers the method through which system-level software testing naturally reveals weak areas. Modern software systems are very complex, thus it's important to test them thoroughly for bugs. As the name suggests, chaos engineering is the practice of evaluating a system's resilience to random, unpredictable interruptions of normal operation. Development teams may prevent bugs and improve software reliability by doing resilience testing. The infrastructure of a piece of software may be subjected to chaos testing to execute proactive experiments. Creating artificial failures may boost morale in a company if it shows that its systems can weather storms and recover quickly.

Chaotic Monkey, developed by Netflix, was the first widely used tool for chaos engineering. Although the resilience tool was somewhat basic, it did include the essentials for doing effective chaotic experiments. IT departments in some companies still use it. There are now a number of commercial and open-source alternatives, such as tools with more sophisticated controls, the ability to integrate with cutting-edge platforms, and more accurate components with which to conduct chaotic experiments (Heorhiadi et al. 57-66).

According to Ryan Petrich, CTO of Capsule, a Linux security provider that has tried several chaos engineering tools, errors are found with them before they become major concerns. Issues with Capsule8's transport layer security monitoring, such as certificate expiry, were uncovered during chaotic testing. Petrich and his colleagues found the problem and fixed it by upgrading the monitoring infrastructure to warn the team a week before certificates were set to expire.

Chaos testing also revealed that not all systems would trigger a failover to another area when they were brought down, which was an unpleasant surprise for Capsule8. Petrich warned that tools aren't everything, and that good chaotic engineering still demands discipline and generally clean operational methods. When "availability" and "resilience" are prioritized at work and precautions are taken to guarantee they are always up and running, "chaos engineering" is "an important discipline to question and verify assumptions," as he put it.

Why Use Tools for Chaos Engineering?

To build trust in complex systems, standard testing approaches have always been utilized. However, recently, chaos engineering tools have emerged as an alternative. The failure of a software platform is inevitable; thus, it is essential to identify vulnerabilities and address them before they disrupt company operations. Chaos engineering is used by major digital companies like Amazon, Netflix, and Microsoft to better understand their own systematic behavior and problems. This method bases itself on the concept of hypothesis testing and performance measurements to evaluate potential system designs. By using assumptions and well-executed chaotic experiments, chaos engineering tools may reveal hidden infrastructure flaws or unresponsive systems (Basiri et al. 1-1).

Chaos engineering entails the following procedures:

Creating a steady-state hypothesis: you should consider all the possible problems that may arise in the system. Create processes for chaotic testing using failure injection and anticipate a range of outcomes.

Simulate real-world scenarios: Real-world circumstances should be simulated. Design a battery of experiments to find out how the system responds to various inputs. Test out a few hypotheses using a small sample size and an experimental group.

Review system metrics: Investigate the results of the system in terms of its performance and the metrics by which it was measured. Learn how often your theories fail and what you can do about it.

Implement changes as needed: Adjust as necessary; after chaotic experiments are complete, you should know what to do. Fix any problems you see and keep going until you've eliminated almost all of them.

Automate chaos experiments: As soon as your system's resilience to the failure mode has been confirmed, you should conduct chaos experiments and automate them in your software delivery pipeline to provide continual validation regardless of any changes to the system's configuration that may occur in the environment. As soon as the experiment fails, you may be alerted and given the option to manually undo the modification that caused the failure or have it undone automatically. You are safe against a certain kind of failure that may bring down your system.

Your company may put its resilience to the test and its tolerance for error to the test with the use of a well-designed and comprehensive practice. Let's check over a few of the most widely used chaos engineering methods for making your systems better.

Examples of common chaos testing instruments include:

Chaos Monkey:

To test how well cloud systems can recover from errors, experts utilize a program called Chaos Monkey. Netflix developed this to check the stability and recovery capabilities of the AWS platform on which it relies. Because it causes chaos and damage like a crazed, armed monkey, Chaos Monkey was given its moniker. In addition, the concept of Chaos Engineering may trace its roots back to Chaos Monkey. It was designed with the idea that constant little failures are preferable than one catastrophic one. One of the first open-source Chaos Engineering tools, it is often credited with having sped up the spread of Chaos Engineering outside major corporations. Netflix used it as a foundation to develop their suite of failure injection tools, the Simian Army, albeit many of those tools have since been deprecated or merged into others, such as Swabbie.

There is just one kind of assault against this system, and that is to kill off running instances of virtual machines. You may schedule it to run for a certain amount of time, after which it will shut down one instance at random. Attempting to simulate unanticipated problems in production with unprepared personnel might backfire (Torkura et al. 1-1).

Key Features:

Among the first open-source technologies, Chaotic Monkey was a pioneering tool for chaos engineering. The Simian Army is a collection of fault injection tools created by Netflix after the company's creation. Some of Chaos Monkey's most notable traits are:

A. Identifies bottlenecks in the system to reduce downtime in production settings.

B. The capacity to conduct infra-level application availability and resilience testing.

C. Tests may be scheduled at certain times of the day.

D. Makes tracking simple.

Chaos Blade:

Alibaba developed it for use in testing various types of failure. It's compatible with a broad variety of environments, from Kubernetes to the cloud to bare metal, and offers a wide variety of attacks, such as packet loss, process termination, and excessive resource use.

For businesses looking to enhance their fault tolerance of distributed systems and guarantee business continuity as they migrate to the cloud or adopt cloud-native architectures, Alibaba's open-source experimental injection tool ChaosBlade may assist.

Chaosblade was developed at MonkeyKing as an internal open-source project. It incorporates the most successful aspects of the several companies that make up the Alibaba Group and is founded on over a decade of trial and error. In addition to being a breeze to use, ChaosBlade also allows for a wide variety of experimentation. Possible cases are:

A. Hardware components such as central processing units, random access memories, networks, disks, processes, and experimental settings;

B. Java applications: Databases, caches, messaging, the Java Virtual Machine (JVM), microservices, etc. are all examples of Java applications. Complex experimental situations may be injected using any class method you like;

C. Applications written in C++ may be exploited in several ways, including the manipulation of variables and return values, the wait before injecting new code, and the specification of arbitrary functions.

D. Container: including cases when the container, its CPU, memory, network, disk, and processes are terminated.

E. Cloud-native platforms: Pod network and Pod itself experimental situations, such as destroying Pods, and container experimental scenarios, such as the aforementioned Docker container experimental scenario, on cloud-native platforms; 5. CPU, memory, network, disk, and process experimental scenarios on Kubernetes platform nodes.

Chaos Mesh:

Seventeen different types of assaults are supported by Chaos Mesh. These include resource exhaustion, network delay, packet loss, bandwidth limitation, disk I/O latency, system time manipulation, and kernel panics. There aren't many open-source programs with a fully-featured online user interface, but Chaos Mesh is one of them (UI). You can witness the immediate effect of the executions in Chaos Mesh by viewing them alongside the metrics for the cluster, which is provided by the integration with Grafana. Failures may be introduced into any part of a Kubernetes infrastructure with the help of Chaos Mesh, a management solution for chaos engineering. All of the kernel, network, I/O, and pod systems fall under this category. With Chaos Mesh, you can simulate latency and have Kubernetes pods automatically terminate. Pod-to-pod communication may be disrupted, and fake read/write failures can be simulated. Rules for when and how an experiment is conducted may be defined. A YAML file is used to define these tests. A dashboard is included in Chaos Mesh for analyzing experimental data. It's built on top of Kubernetes and works with most cloud providers. It's free and open source, and it was just approved as a sandbox project at the CNCF. Incorporating Chaos Mesh into your DevOps process is a great way to include chaos engineering techniques into your application development (Basiri et al. 1-1).

Key Features:

A. When testing high-profile distribution systems like Apache APISIX and RabbitMQ, Chaos Mesh relies on a Kubernetes-based interface that comes with complete automation and graphical features.
B. The Chaos Mesh framework allows for event-driven fault simulations to be used for scenario testing.
C. Chaos Mesh enables experimentation on the platform with a wide range of controllable variables and built-in status monitoring.

Simmy:

Simmy is an open-source chaos-injecting program that works in tandem with the Polly.NET resilience project. Polly is the platform on which your scripts will be executed, and it lets you design chaos-injection rules. The system's rules include an exceptions policy, a behavior policy, and a general policy that allows for the introduction of any new behavior. The behavior is introduced at random by these regulations. Any time Polly is used to run code, Simmy may be used to add a chaos-injection policy (Monkey Policy) or policies. As a result, Simmy now proposes three forms of chaos management:

A. To simulate system failure, you may use the Fault feature, which injects exceptions or replacement results.
B. *Latency:* Introduces a delay into call executions.
C. *Behavior:* Inject any additional behavior prior to making a call.

All chaos policies (Monkey policies) aim to inject behavior at random, whether it errors, delay, or bespoke behavior; a Monkey policy lets you set an injection rate anywhere from 0% to 100%. the greater the pace of injection, the greater the likelihood of injecting them. The policy also lets you choose whether or not random injection is enabled, allowing you to release/hold (turn on/off) the monkeys independently of the injection rate you choose. For example, setting an injection rate of 100% will have no effect if you instruct the policy to ignore random injection.

Litmus:

Litmus, like Chaos Mesh, is a tool built specifically for Kubernetes and is hosted in the CNCF sandbox. It was first built for evaluating OpenEBS, a free and open-source data storage option for Kubernetes. During, after, and even before an experiment, you may verify the status of your application with the help of Litmus's built-in health monitoring function, Litmus Probes. Litmus has a considerably more complicated onboarding process compared to other platforms. Each experimental app and namespace in Litmus needs its own service account and annotations.

Azure:

In order to assess and enhance the robustness of your cloud-based applications and services, you may leverage the principles of chaos engineering with Azure Chaos Studio, a fully managed solution. It does tests that may mimic problems or occurrences, such as the failure of a region or an application that causes a virtual machine to occupy all of its CPU.

The capacity of a system to resist and recover from stressors is referred to as its resilience. Errors and failures caused by an interrupted application may have a significant impact on your operations and goals. It is critical to test and enhance your application's resilience whether you are designing, moving, or managing Azure apps (Torkura et al. 1-1).

You may set up your Azure Chaos Studio experiments in one of three ways:

A. With service-direct fault, you may execute code directly against an Azure resource without the requirement for instrumentation.

B. The agent-based fault method necessitates the installation of the chaos agent.

C. Injecting faults into an AKS cluster with the help of Chaos Mesh, a free, open-source chaos engineering tool for Kubernetes. Service-direct problems caused by Chaos Mesh need the addition of Chaos Mesh to the AKS cluster.

Chapter VII: Chaos in Practice

Microsoft provides three different tutorials on how to begin using Azure Chaos Studio on its documentation page: one for service-direct errors, another for agent-based faults, and a third for AKS Chaos Mesh issues. The following examples make use of agent-based and service-direct faults. Two virtual computers running Linux (Ubuntu 20.04) and the Apache web server are employed in this lab setting, with the whole thing buffered by a load balancer. They are set up on a virtual LAN with their own subnet and security group.

Testing complex systems by making them fail:

We may run into trouble testing what occurs if a service breaks under load, since more complicated systems need more testing. When a shopping cart's backend is forced to switch databases in the midst of a purchase, how exactly do transactions fail? Is there a plan in place for a restaurant delivery tracking app to handle a major messaging platform failure?

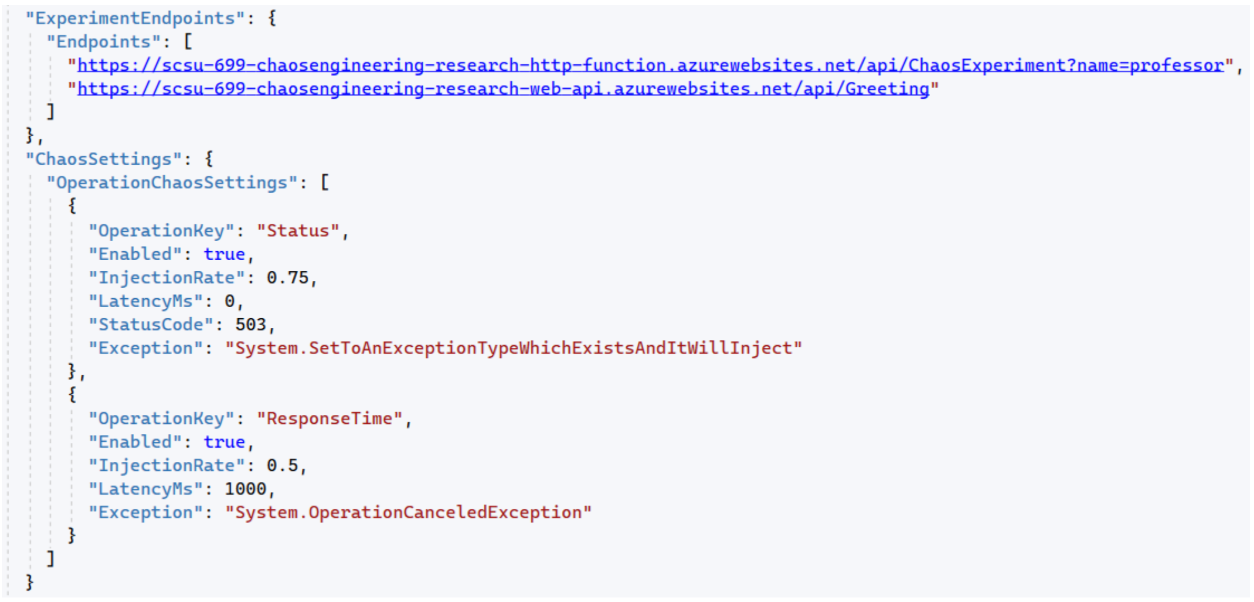
We need a model of testing that takes into account live systems and gradually fails off components in order to monitor system behavior. The goal is to see how well a system can handle a controlled dose of failure while under load. The chaos monkey tool developed at Netflix was the first of its kind, and it helped pave the way for a new approach to software testing called chaos engineering.

Although learning about system failures is a useful byproduct of chaotic engineering, the primary goal is to demonstrate the robustness of the system under stress. Regardless of what was going on in the world, Netflix had to provide its consumers with a consistently excellent viewing experience.

It's hardly unexpected that other platforms, notably hyperscale cloud providers like Microsoft Azure, have adopted such methods. You need to know that your Azure-hosted apps will keep functioning if a Microsoft server goes down. The Microsoft team known as chaos engineering routinely investigates the platform's vulnerability to failures in order to make sure that the services your apps rely on can absorb them without causing any harm.

Chaos Experiment using Simmy .NET:

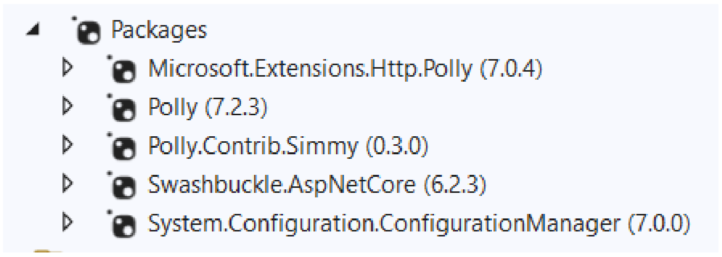
As part of the experiment, Fault service response and Latency will be injected. The Simmy package from the Polly is used for the experiment. It has the inject rate to determine how many requests should be affected with the faults and we will also mention the exception return types in case of failures. The configuration consists of Operation key, Enabled, Injection rate, Latency in milliseconds, Status Code and Exception. It also has experiment endpoints which endpoints are being invoked as part of the experiment. It should be as below figure.



```
"ExperimentEndpoints": {
  "Endpoints": [
    "https://scsu-699-chaosengineering-research-http-function.azurewebsites.net/api/ChaosExperiment?name=professor",
    "https://scsu-699-chaosengineering-research-web-api.azurewebsites.net/api/Greeting"
  ]
},
"ChaosSettings": {
  "OperationChaosSettings": [
    {
      "OperationKey": "Status",
      "Enabled": true,
      "InjectionRate": 0.75,
      "LatencyMs": 0,
      "StatusCode": 503,
      "Exception": "System.SetToAnExceptionTypeWhichExistsAndItWillInject"
    },
    {
      "OperationKey": "ResponseTime",
      "Enabled": true,
      "InjectionRate": 0.5,
      "LatencyMs": 1000,
      "Exception": "System.OperationCanceledException"
    }
  ]
}
```

Fig. 2. Chaos Experiment Settings Configuration.

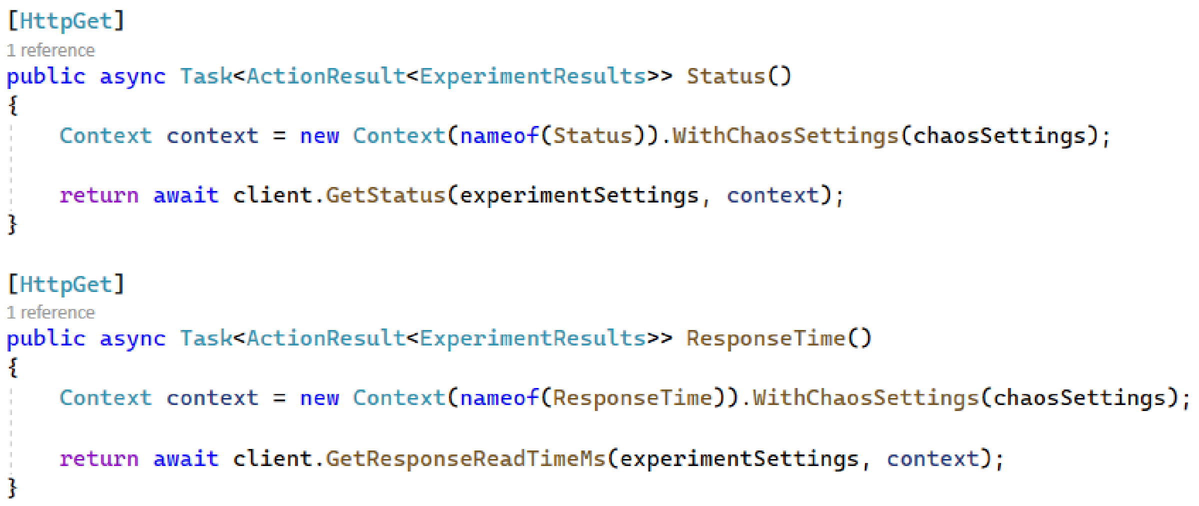
Below packages are used for the experiment and these packages will be available in NuGet library.



```
◢ Packages
    ▷ Microsoft.Extensions.Http.Polly (7.0.4)
    ▷ Polly (7.2.3)
    ▷ Polly.Contrib.Simmy (0.3.0)
    ▷ Swashbuckle.AspNetCore (6.2.3)
    ▷ System.Configuration.ConfigurationManager (7.0.0)
```

Fig. 3. Simmy and Polly Packages for Experiment.

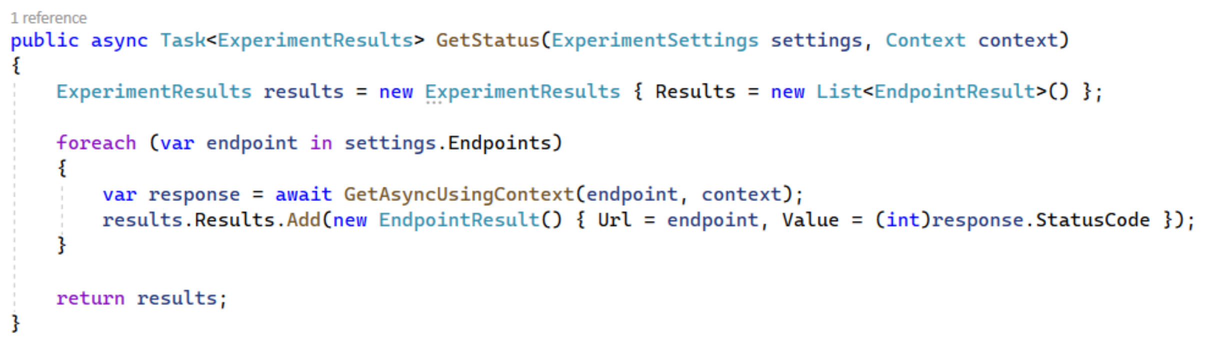
ResilientHttpClient is created to make rest API calls to the microservices, and the API calls will this client object where the faults will be injected into. To inject this, Polly Context will be used, and this context is initialized using the chaos settings given in the configuration as mentioned in Fig. 2. Below code will give insights on the Context creation. As part of this experiment, we implemented two endpoints. One is to fetch the status of the request, and another is to inject the latency as mentioned above in the chaos settings.



```
[HttpGet]
1 reference
public async Task<ActionResult<ExperimentResults>> Status()
{
    Context context = new Context(nameof(Status)).WithChaosSettings(chaosSettings);

    return await client.GetStatus(experimentSettings, context);
}

[HttpGet]
1 reference
public async Task<ActionResult<ExperimentResults>> ResponseTime()
{
    Context context = new Context(nameof(ResponseTime)).WithChaosSettings(chaosSettings);

    return await client.GetResponseReadTimeMs(experimentSettings, context);
}
```

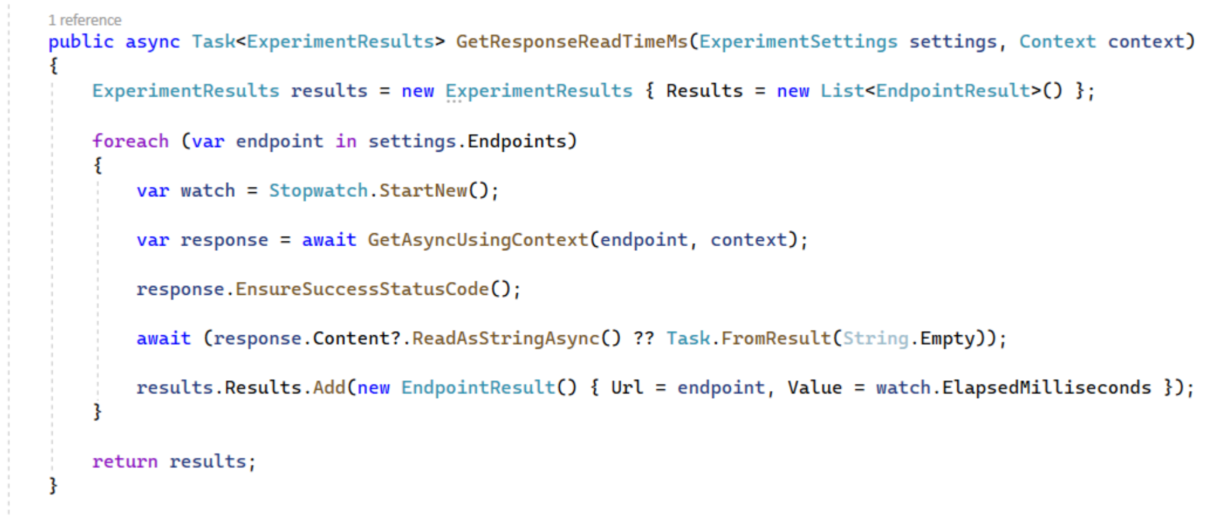
Fig. 4. Context Creation using the settings.

The context created will be used in the mentioned two methods Status and ResponseTime. For Status call, we will return the status code 503 as mentioned in the configuration and for response time, we will inject the latency and we will return the time elapsed for the call in milliseconds. Below code illustrates the same.



```
1 reference
public async Task<ExperimentResults> GetStatus(ExperimentSettings settings, Context context)
{
    ExperimentResults results = new ExperimentResults { Results = new List<EndpointResult>() };

    foreach (var endpoint in settings.Endpoints)
    {
        var response = await GetAsyncUsingContext(endpoint, context);
        results.Results.Add(new EndpointResult() { Url = endpoint, Value = (int)response.StatusCode });
    }

    return results;
}
```

Fig. 5. Status.



```
1 reference
public async Task<ExperimentResults> GetResponseReadTimeMs(ExperimentSettings settings, Context context)
{
    ExperimentResults results = new ExperimentResults { Results = new List<EndpointResult>() };

    foreach (var endpoint in settings.Endpoints)
    {
        var watch = Stopwatch.StartNew();

        var response = await GetAsyncUsingContext(endpoint, context);

        response.EnsureSuccessStatusCode();

        await (response.Content?.ReadAsStringAsync() ?? Task.FromResult(String.Empty));

        results.Results.Add(new EndpointResult() { Url = endpoint, Value = watch.ElapsedMilliseconds });
    }

    return results;
}
```

Fig. 6. Response Time.

The results of the experiment can be seen in any API testing tools like Postman, Browsers and Swagger. The experiment endpoints are microservices and they are hosted on Azure App service and Azure Function app. Below URL can used for the testing of the APIs.

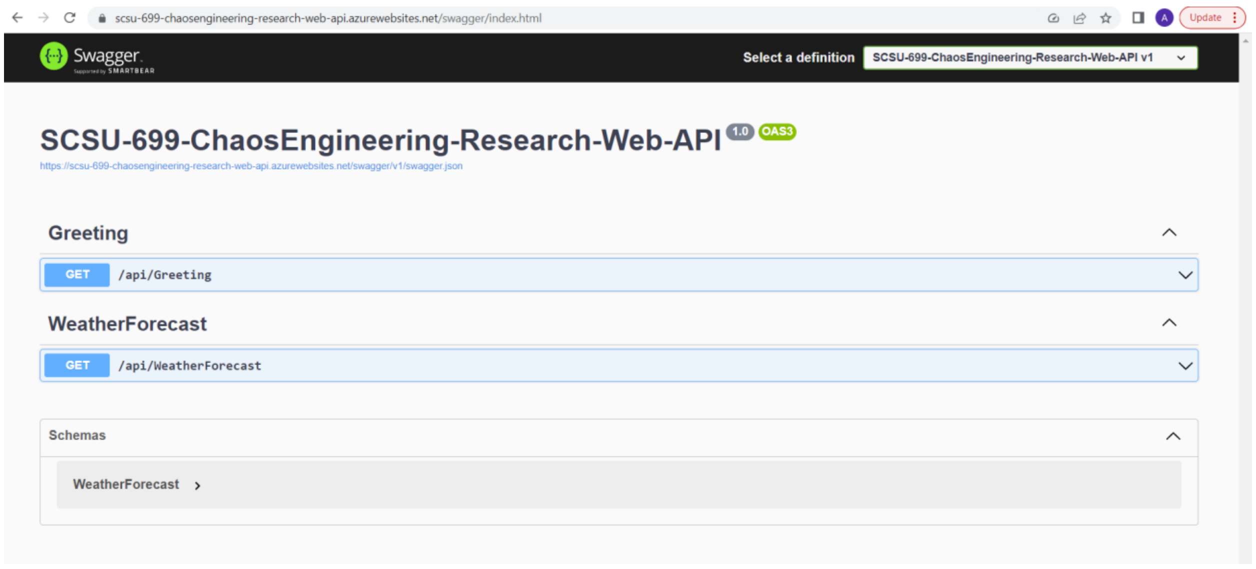
A. https://scsu-699-chaosengineering-research-web-api.azurewebsites.net/swagger/index.html

B. https://scsu-699-chaosengineering-research-http-function.azurewebsites.net/api/ChaosExperiment?name=professor

The experiment application is also an API hosted on Azure. It is deployed to Azure app service. Below URL can be used for testing.

A. https://scsu-699-chaosengineering-research-api-experiment.azurewebsites.net/index.html

Swagger can be used for testing APIs and Below figure illustrates on the API which has two endpoints Greeting and Weather Forecast. Greeting endpoint will send a good morning, Good Afternoon, Good Evening and Good night based on the server's current time. A name parameter can be passed to display the name as well in the response. Weather Forecast is to send the weather information based on the random values in the backend.


scsu-699-chaosengineering-research-web-api.azurewebsites.net/swagger/index.html

Select a definition: SCSU-699-ChaosEngineering-Research-Web-API v1

SCSU-699-ChaosEngineering-Research-Web-API 1.0 OAS3

https://scsu-699-chaosengineering-research-web-api.azurewebsites.net/swagger/v1/swagger.json

Greeting

GET /api/Greeting

WeatherForecast

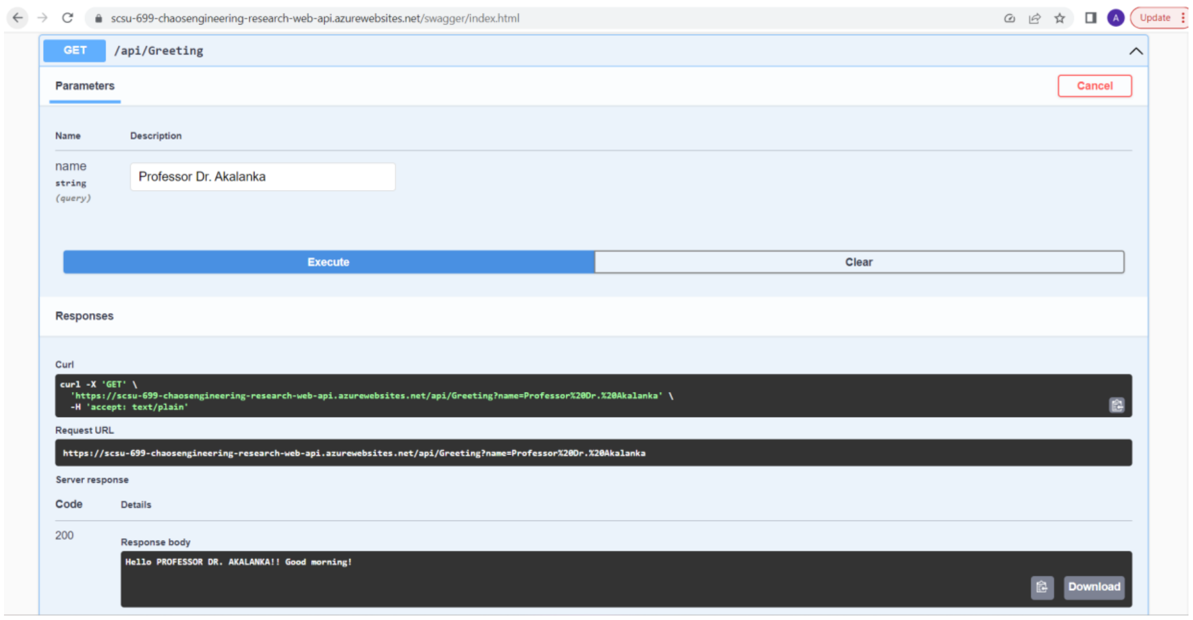
GET /api/WeatherForecast

Schemas

WeatherForecast

Fig. 7. Swagger for API.

Sample responses can be displayed as below:



scsu-699-chaosengineering-research-web-api.azurewebsites.net/swagger/index.html

GET /api/Greeting

Parameters

Name: name, string (query) — Description: Professor Dr. Akalanka

Responses

Curl

```
curl -X 'GET' \
  'https://scsu-699-chaosengineering-research-web-api.azurewebsites.net/api/Greeting?name=Professor%20Dr.%20Akalanka' \
  -H 'accept: text/plain'
```

Request URL

```
https://scsu-699-chaosengineering-research-web-api.azurewebsites.net/api/Greeting?name=Professor%20Dr.%20Akalanka
```

Server response

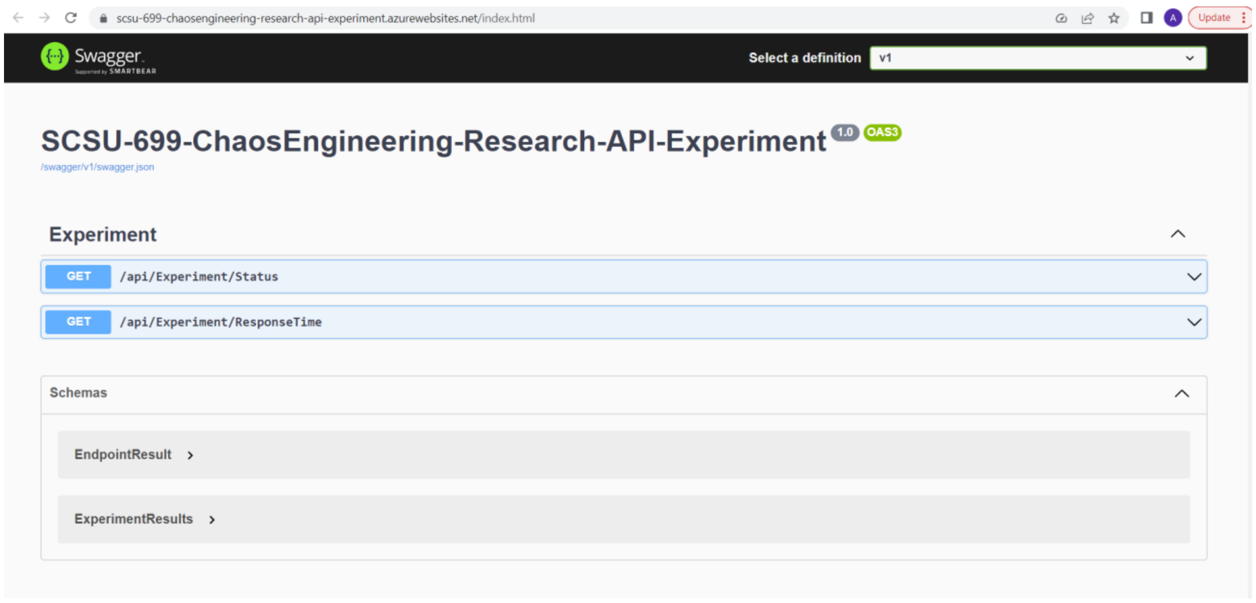
Code: 200

Response body

```
Hello PROFESSOR DR. AKALANKA!! Good morning!
```

Fig. 8. Sample API Response.

Experiment API has two endpoints as below and the swagger API would like as below.



scsu-699-chaosengineering-research-api-experiment.azurewebsites.net/index.html

Select a definition: v1

SCSU-699-ChaosEngineering-Research-API-Experiment 1.0 OAS3

/swagger/v1/swagger.json

Experiment

GET /api/Experiment/Status

GET /api/Experiment/ResponseTime

Schemas

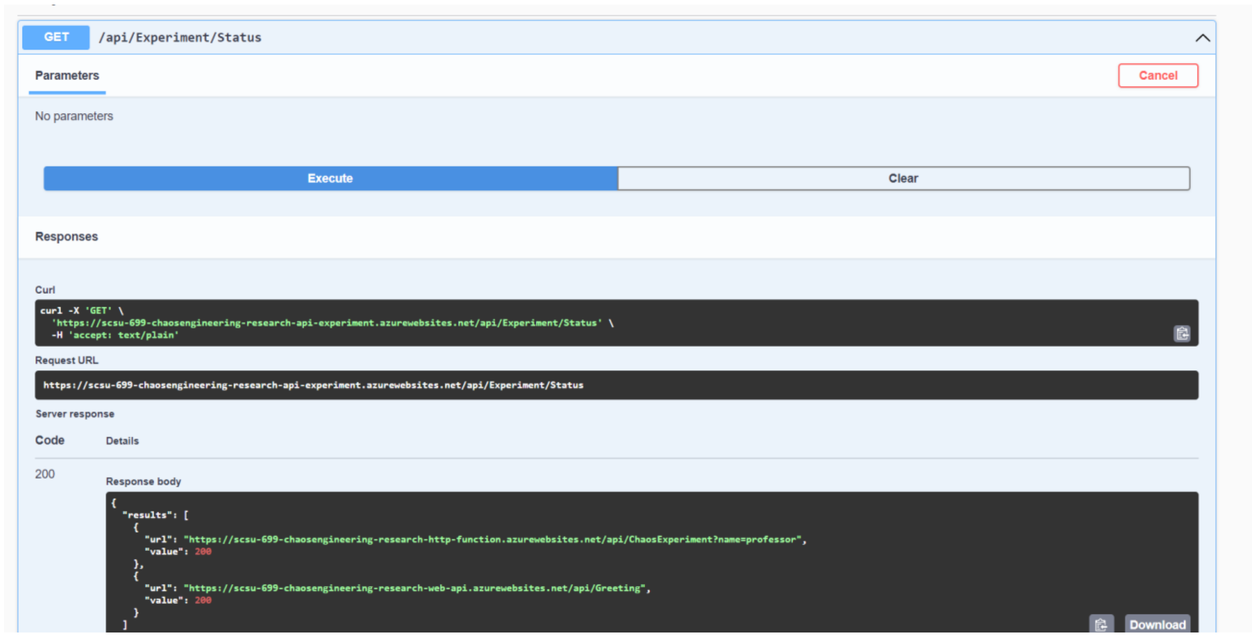
EndpointResult

ExperimentResults

Fig. 9. Experiment API.

Sample response for the experiment API.

Status Call:



Curl

```
curl -X 'GET' \
  'https://scsu-699-chaosengineering-research-api-experiment.azurewebsites.net/api/Experiment/Status' \
  -H 'accept: text/plain'
```

Request URL

```
https://scsu-699-chaosengineering-research-api-experiment.azurewebsites.net/api/Experiment/Status
```

Server response

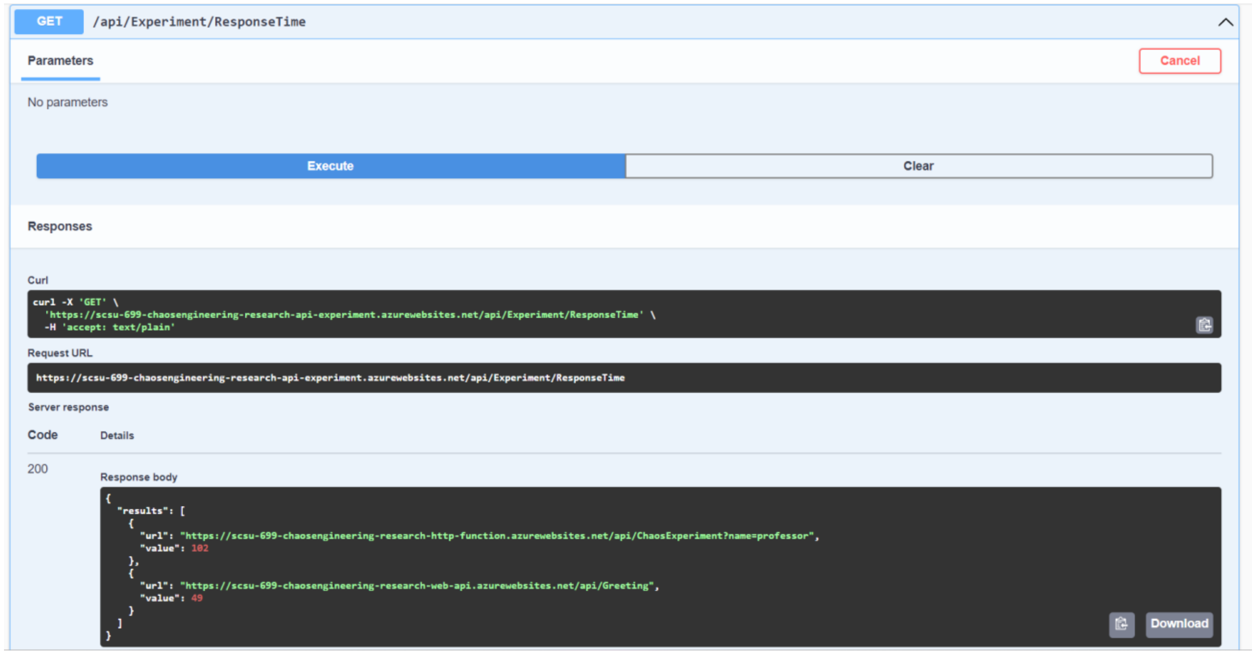
Code Details

200 Response body

```
{
  "results": [
    {
      "url": "https://scsu-699-chaosengineering-research-http-function.azurewebsites.net/api/ChaosExperiment?name=professor",
      "value": 200
    },
    {
      "url": "https://scsu-699-chaosengineering-research-web-api.azurewebsites.net/api/Greeting",
      "value": 200
    }
  ]
```

Fig. 10. Status call API Response.

Response Time Call:



Curl

```
curl -X 'GET' \
  'https://scsu-699-chaosengineering-research-api-experiment.azurewebsites.net/api/Experiment/ResponseTime' \
  -H 'accept: text/plain'
```

Request URL

```
https://scsu-699-chaosengineering-research-api-experiment.azurewebsites.net/api/Experiment/ResponseTime
```

Server response

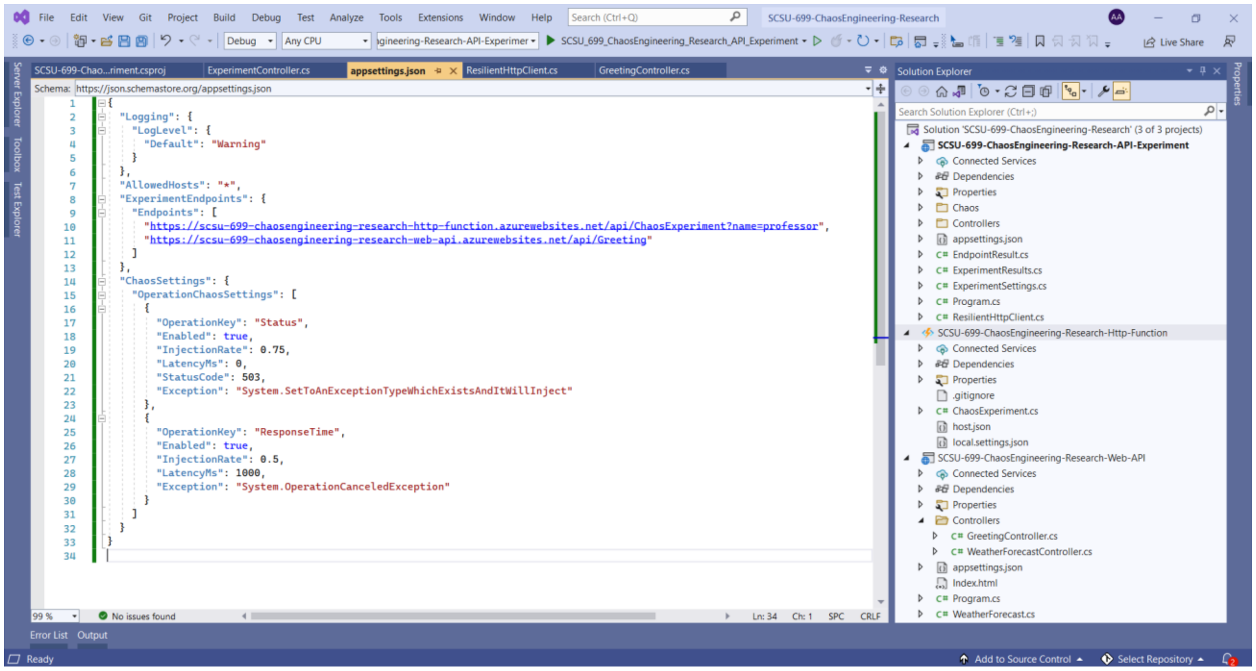
Code Details

200 Response body

```
{
  "results": [
    {
      "url": "https://scsu-699-chaosengineering-research-http-function.azurewebsites.net/api/ChaosExperiment?name=professor",
      "value": 102
    },
    {
      "url": "https://scsu-699-chaosengineering-research-web-api.azurewebsites.net/api/Greeting",
      "value": 49
    }
  ]
}
```

Fig. 11. Response Time API Response.

This entire application is developed using C#, Dotnet 6 and Azure. It is implemented using Visual studio and the code structure is defined as below.



```json
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ExperimentEndpoints": {
    "Endpoints": [
      "https://scsu-699-chaosengineering-research-http-function.azurewebsites.net/api/ChaosExperiment?name=professor",
      "https://scsu-699-chaosengineering-research-web-api.azurewebsites.net/api/Greeting"
    ]
  },
  "ChaosSettings": {
    "OperationChaosSettings": [
      {
        "OperationKey": "Status",
        "Enabled": true,
        "InjectionRate": 0.75,
        "LatencyMs": 0,
        "StatusCode": 503,
        "Exception": "System.SetToAnExceptionTypeWhichExistsAndItWillInject"
      },
      {
        "OperationKey": "ResponseTime",
        "Enabled": true,
        "InjectionRate": 0.5,
        "LatencyMs": 1000,
        "Exception": "System.OperationCanceledException"
      }
    ]
  }
}
```

```
Solution 'SCSU-699-ChaosEngineering-Research' (3 of 3 projects)
  SCSU-699-ChaosEngineering-Research-API-Experiment
    Connected Services
    Dependencies
    Properties
    Chaos
    Controllers
    appsettings.json
    EndpointResult.cs
    ExperimentResults.cs
    ExperimentSettings.cs
    Program.cs
    ResilientHttpClient.cs
  SCSU-699-ChaosEngineering-Research-Http-Function
    Connected Services
    Dependencies
    Properties
    .gitignore
    ChaosExperiment.cs
    host.json
    local.settings.json
  SCSU-699-ChaosEngineering-Research-Web-API
    Connected Services
    Dependencies
    Properties
    Controllers
      GreetingController.cs
      WeatherForecastController.cs
    appsettings.json
    Index.html
    Program.cs
    WeatherForecast.cs
```

Fig. 12. Solution Structure.

Chapter VIII: Conclusion

Chaos engineering is a powerful approach to improving the resiliency and reliability of microservices architecture-based systems. Through deliberate and controlled experimentation, chaos engineering can identify weaknesses and vulnerabilities in a system before they cause catastrophic failures in production. While there are certainly some drawbacks to implementing chaos engineering, such as the cost and complexity of setting up and running experiments, the benefits far outweigh these drawbacks. Some of the key benefits of chaos engineering include increased confidence in the system's ability to handle unexpected failures, improved understanding of the system's behavior under stress, and reduced downtime and associated costs. The research also showed that there are several tools available for implementing chaos engineering in microservices. Some of the most popular tools include Chaos Monkey, Chaos Monkey, Chaos Blade, Chaos Mesh, Simmy, Litmus and Azure Chaos Studio. These tools provide a variety of features and capabilities, including fault injection, latency injection, and network partitioning, which can help teams to better understand how their microservices will behave in various failure scenarios. Overall, the research suggests that chaos engineering is a valuable approach for improving the resiliency and reliability of microservices architecture-based systems. By identifying and addressing weaknesses in the system before they cause problems in production, chaos engineering can help teams to deliver more reliable and resilient services to their customers. While there are certainly challenges associated with implementing chaos engineering, the benefits of doing so make it a worthwhile investment for organizations looking to improve the quality of their microservices-based systems.

Works Cited

Basiri, Ali, et al. "Chaos Engineering." *IEEE,* vol. 33, 2016: 1-1.

Björnberg, Adam. "Cloud native chaos engineering for IoT systems." 2021.

De, Suman. "A Study on Chaos Engineering for improving Cloud Software Quality and Reliability. In 2021 International Conference on Disruptive Technologies for Multi-Disciplinary Research and Applications (CENTCON)." *IEEE* vol. 1 (2021, November), pp. 289-294.

Escobar, Daniel. "Towards the understanding and evolution of monolithic applications as microservices." 2016.

Heorhiadi, et al. "Gremlin: Systematic Resilience Testing of Microservices." 2016, pp. 57-66.

Jamshidi, Pooyan, et al. "Microservices: The journey so far and challenges ahead." *IEEE* vol. 35, no. 3, 2018, pp. 24-35.

Jones, Nora, et al. *Chaos Engineering*. 2018.

Kesim, Dominik. "Assessing resilience of software systems by the application of chaos engineering: a case study ." *Bachelor's thesis* (2019).

Kutlu, Kübra. "Machine Learning based Chaos Engineering for Cloud-Native Microservice Architectures." 2021.

Naqvi, Moeen Ali, et al. "On Evaluating Self-Adaptive and Self-Healing Systems using Chaos Engineering. arXiv preprint arXiv:2208.13227." 2022.

Poltronieri, Filippo, et al. "A Chaos Engineering Approach for Improving the Resiliency of IT Services Configurations." 2022.

Simonsson, Jesper, et al. "Observability and chaos engineering on system calls for containerized applications in Docker. Future Generation Computer Systems." vol. 122, 2021, pp. 117-129.

Th¨ones, Johannes. "Microservices architecture style." *IEEE Software* vol. 32, 2019, pp. 116-116.

Torkura, Kennedy, et al. "CloudStrike: Chaos Engineering for Security and Resiliency in Cloud Infrastructure." *IEEE*, 2020, pp. 1-1.

Wang, Yuwei. "Towards service discovery and autonomic version management in self-healing microservices architecture." *In Proceedings of the 13th European Conference on Software Architecture* 2, 2019, September, pp. 63-66.

Yin, Kanglin, et al. "Analyse resilience risks in microservice architecture systems with causality search and inference algorithms." *International Journal of Web and Grid Services,* vol. 16, no. 2, 2020, pp. 147-171.

Zhou, Xiang, et al. "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study." *IEEE* vol. 47, no. 2, 2021, pp. 243–260.