National College of Ireland

Evaluating Kubernetes Built-In Self-Healing Across Local and Cloud Environments Under Common Microservice Failure Scenarios

MSc Research Project
Cloud Computing

Oluseyi Iyetomide Simon Coker
Student ID: x23370751

School of Computing
National College of Ireland

Supervisor: Dr. Abubakr Siddig

National College of Ireland
Project Submission Sheet
School of Computing

National College of Ireland

Student Name:	Oluseyi Iyetomide Simon Coker
Student ID:	x23370751
Programme:	Cloud Computing
Year:	2025
Module:	MSc Research Project
Supervisor:	Dr. Abubakr Siddig
Submission Due Date:	11/12/2025
Project Title:	Evaluating Kubernetes Built-In Self-Healing Across Local and Cloud Environments Under Common Microservice Failure Scenarios
Word Count:	6,262
Page Count:	23

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Oluseyi Iyetomide Simon Coker
Date:	11th December 2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	✓
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	✓
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	✓

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Evaluating Kubernetes Built-In Self-Healing Across Local and Cloud Environments Under Common Microservice Failure Scenarios

Oluseyi Iyetomide Simon Coker
x23370751

Abstract

Kubernetes has become a common platform for running microservices, mostly because it does a lot of the heavy lifting when things fail. One of the main reasons people trust it is its ability to fix basic problems on its own. When a pod crashes or becomes unhealthy, Kubernetes quietly replaces it and keeps the service running. Even though developers rely on this behaviour every day, there is surprisingly little work that looks closely at how well Kubernetes performs without any extra support from tools such as Istio, RAMSES or other resilience add-ons. This study tries to fill that gap by examining how Kubernetes copes with a few realistic microservice failures on its own. To do that, I used two setups. The first was a local Minikube cluster running through my own machine, and the second was a small K3s cluster on an EC2 instance. I deployed the same simple microservice on both and tested four scenarios: a clean baseline run, a pod crash, resource pressure using CPU and memory stress, and a network issue created by adding delay and packet loss. Each test was run under steady load so that I could measure things like recovery time, throughput and response latency. The results were fairly consistent across both environments. Kubernetes managed to recover from all four scenarios and kept the service available most of the time. The cloud setup recovered slightly faster, most likely because it had more stable resources to work with. When I compared these findings with results from studies that rely on external resilience tools, it became clear that Kubernetes does quite well on its own, although larger or more demanding systems would still benefit from additional layers of protection.

1 Introduction

Kubernetes is now one of the default choices for running microservices in production. It places containers on nodes, restarts them when they fail, and keeps the cluster in a healthy state without constant intervention. This behaviour is often described as self healing. When a pod crashes or becomes unresponsive, Kubernetes replaces it and eventually routes traffic back to the new instance. In practice, many teams rely on these features, but there is still limited clarity on how well Kubernetes copes with common failures when running on its own, without extra resilience tooling.

Much of the recent work on resilience in Kubernetes assumes that the cluster is extended with service meshes, anomaly detectors, or adaptive control frameworks. For example, Singh et al. (2024) and Capozucca et al. (2021) show that these add-ons can

improve recovery by adding retries, traffic redirection, or dynamic configuration adjustments. At the same time, other studies highlight cases where Kubernetes struggles, including slow or partial failures, scheduling delays, and long-running microservices that degrade over time (Gan et al., 2021; Carrión, 2022; Zhang et al., 2022). Because many of these studies combine the platform with external tooling, it becomes difficult to isolate how much resilience Kubernetes provides on its own.

This project focuses on that baseline. It evaluates how Kubernetes recovers from a small but realistic set of pod-level faults without any additional resilience frameworks. Four scenarios are examined: a clean baseline run, a deliberate pod crash, a CPU and memory stress test, and a network fault that injects delay and packet loss. Each scenario is executed on two environments that reflect typical development and cloud setups. The first is a local Minikube cluster running inside WSL2 on a Windows host. The second is a lightweight K3s cluster running on an Amazon EC2 instance. The same simple HTTP microservice is deployed in both environments so that differences in behaviour can be attributed to the orchestration layer rather than application logic.

The main research question asks how effective Kubernetes native self healing is at restoring availability and maintaining performance across these fault scenarios, and how this compares with results reported in studies that depend on external resilience layers. The aim is not to challenge the usefulness of meshes or adaptive controllers. Instead, the project provides a clear baseline that other researchers and practitioners can use when judging how much additional value these tools bring on top of what Kubernetes already provides.

To answer the research question, the experiments use controlled fault injection and load testing to measure recovery time, throughput degradation, and the time taken to return to a steady state in each environment. Section 2 reviews the existing work on resilience in Kubernetes and microservice systems. Section 3 explains the experimental design and fault injection procedures. Section 6 presents and analyses the experimental results, and Section 7 summarises the findings and outlines possible directions for future work.

2 Related Work

This section reviews the most relevant work on Kubernetes reliability, failure handling, and microservice resilience. The focus is on what the literature already understands about native Kubernetes behaviour, where its limitations appear most often, and why it is still important to study Kubernetes on its own rather than through frameworks that attempt to improve it.

2.1 Native Kubernetes Self-Healing

Kubernetes was designed to restart failed containers and move workloads back to a desired state. Studies such as Kambala (2025) show that these mechanisms work well for straightforward pod crashes, especially for stateless microservices. Similar observations appear in cloud-focused work such as Shabariram et al. (2024), who note that Kubernetes usually restores service availability without human intervention, although failures are not always detected immediately. This delay becomes important when evaluating how dependable Kubernetes truly is under different conditions.

2.2 Where Kubernetes Struggles

Several papers highlight cases where Kubernetes' default checks are not enough. Gan et al. (2021) show that slow or partially unresponsive pods may continue running long after they should have been restarted, while Zhang et al. (2022) demonstrate that long-running microservices can degrade over time without triggering any probe failures. These findings matter because many real-world failures do not follow the clean crash–restart pattern that Kubernetes handles best.

Scheduling behaviour can also influence how fast a cluster recovers. Carrión (2022) explain that resource pressure and scheduling delays can lengthen recovery time, which is particularly noticeable in small clusters such as Minikube or low-resource EC2 nodes.

2.3 Fault Injection and Chaos Experiments

A large body of work uses fault injection to understand failure behaviour in distributed systems. Al-Arif et al. (2021) introduce Defektor, a framework for running structured fault campaigns, and show that real failures often behave differently from synthetic stress tests. In containerised environments, Baumann et al. (2019) highlight the difficulty of detecting faults without deeper observability tools.

Although this project does not use a full chaos-engineering platform, the experimental approach is aligned with the method used in these studies: trigger a specific fault, observe how the system responds, and measure the time needed to return to a stable state.

2.4 Resilience Tools Built on Top of Kubernetes

Because Kubernetes has detection blind spots, many researchers propose additional layers to improve recovery. Service meshes such as Istio provide retries, circuit breaking and traffic shaping. Singh et al. (2024) show that these features significantly reduce disruption during failures. Adaptive systems like RAMSES (Capozucca et al., 2021) monitor behaviour and adjust workloads in response to observed drift. Other approaches include proactive recovery for stateful services (Tran et al., 2022) and autoscaling strategies that respond to changing load patterns (Rodriguez et al., 2018).

Across these studies, the same assumption appears repeatedly: Kubernetes alone is not enough for strong resilience. However, very few papers actually benchmark plain Kubernetes without enhancements, leaving a gap in understanding that this project directly addresses.

2.5 Monitoring, Detection and Observability

Since self-healing depends on accurate detection, several authors analyse how Kubernetes identifies faults. Matsuo and Ikegami (2021) show that anomaly-detection methods flag problems earlier than Kubernetes' probes, while the unsupervised approach in Myrtollari et al. (2025) detects degradation long before pod-level signals fail. Rouf et al. (2021) show that automated DevOps systems can react to runtime drift faster than Kubernetes' event-driven model.

The consistent message in this work is that Kubernetes recovers well once a failure is detected, but detection itself may be slow or incomplete.

2.6 Summary of Key Studies

Table 1 summarises the main academic studies discussed in this section, highlighting their core theme and how each one informs the design of this project.

Table 1: Summary of key studies on Kubernetes resilience, failure handling and observability.

Paper	**Theme**	**Relevance to this project**
Al-Arif et al. (2021)	Fault injection	Supports the chaos-style fault injection approach (pod crash, CPU load, network delay).
Baumann et al. (2019)	Resilience monitoring	Highlights the need for real measurements and observability when assessing resilience.
Capozucca et al. (2021)	External resilience framework	Shows benefits of adaptive frameworks, motivating comparison with plain Kubernetes.
Carrión (2022)	Scheduling	Shows how scheduling and resource pressure impact recovery time in small clusters.
Gan et al. (2021)	Failure modes ("Mutiny!")	Identifies Kubernetes detection gaps and motivates the chosen failure scenarios.
Matsuo and Ikegami (2021)	Anomaly detection	Demonstrates weaknesses of probe-based detection, informing the observability discussion.
Müller et al. (2022)	Stateful recovery	Illustrates limits of simple restart behaviour for stateful workloads.
Prakash (2023)	Benchmarking orchestration tools	Supports the experimental benchmarking design across different cluster setups.
Rodriguez et al. (2018)	Autoscaling	Provides background for the resource-stress tests and CPU-pressure interpretation.
Rouf et al. (2021)	DevOps automation	Highlights limitations of native healing and value of external operational automation.
Shabariram et al. (2024)	Cloud self-healing	Shows the value of baseline self-healing evaluation in cloud and edge deployments.
Singh et al. (2024)	Istio / enhanced resilience	Main external comparison point for resilience beyond plain Kubernetes.
Tran et al. (2022)	Proactive recovery	Shows recovery strategies that Kubernetes lacks natively, framing the research gap.
Zhang et al. (2022)	Aging microservices	Demonstrates undetected degradation, echoing the slow-failure scenarios in this project.

Paper	Theme	Relevance to this project
Kambala (2025)	Self-healing baseline	Directly relevant to the research question on native Kubernetes self-healing.
Pandey (2025)	Edge resilience	Links resilience findings to small-scale clusters like the K3s EC2 node.
Samarakoon et al. (2023)	Adaptive systems	Shows wider relevance of self-healing and adaptation in IoT-edge infrastructures.
Flora et al. (2022)	Failure handling	Supports pod-level fault analysis and long-running microservice degradation.
Myrtollari et al. (2025)	Unsupervised anomaly detection	Highlights detection gaps and the need for richer monitoring beyond probes.

2.7 Summary and Research Gap

The literature agrees on two points. First, Kubernetes provides a solid baseline for handling straightforward pod crashes. Second, it struggles with more subtle failures such as slow degradation, partial unresponsiveness and probe inaccuracies. Many resilience tools attempt to compensate for these issues, but their benefits cannot be evaluated without a clear understanding of what Kubernetes already provides by default.

Only a small number of studies isolate Kubernetes and measure native recovery without extra frameworks. This project addresses that gap by running controlled pod crashes, resource stress and network faults on Minikube and a lightweight K3s EC2 node, providing a clean baseline against which enhanced systems can be compared.

3 Methodology

The goal of this project was to observe how a container-orchestrated microservice behaves when common failures occur, and to compare this behaviour across two environments: a local Minikube cluster and a lightweight cloud cluster running K3s on EC2. I designed a small set of controlled experiments that let me trigger specific failures, apply steady load, and record how Kubernetes responded. This section explains how the environments were configured, how the experiments were run, and how the results were processed so that the study can be replicated.

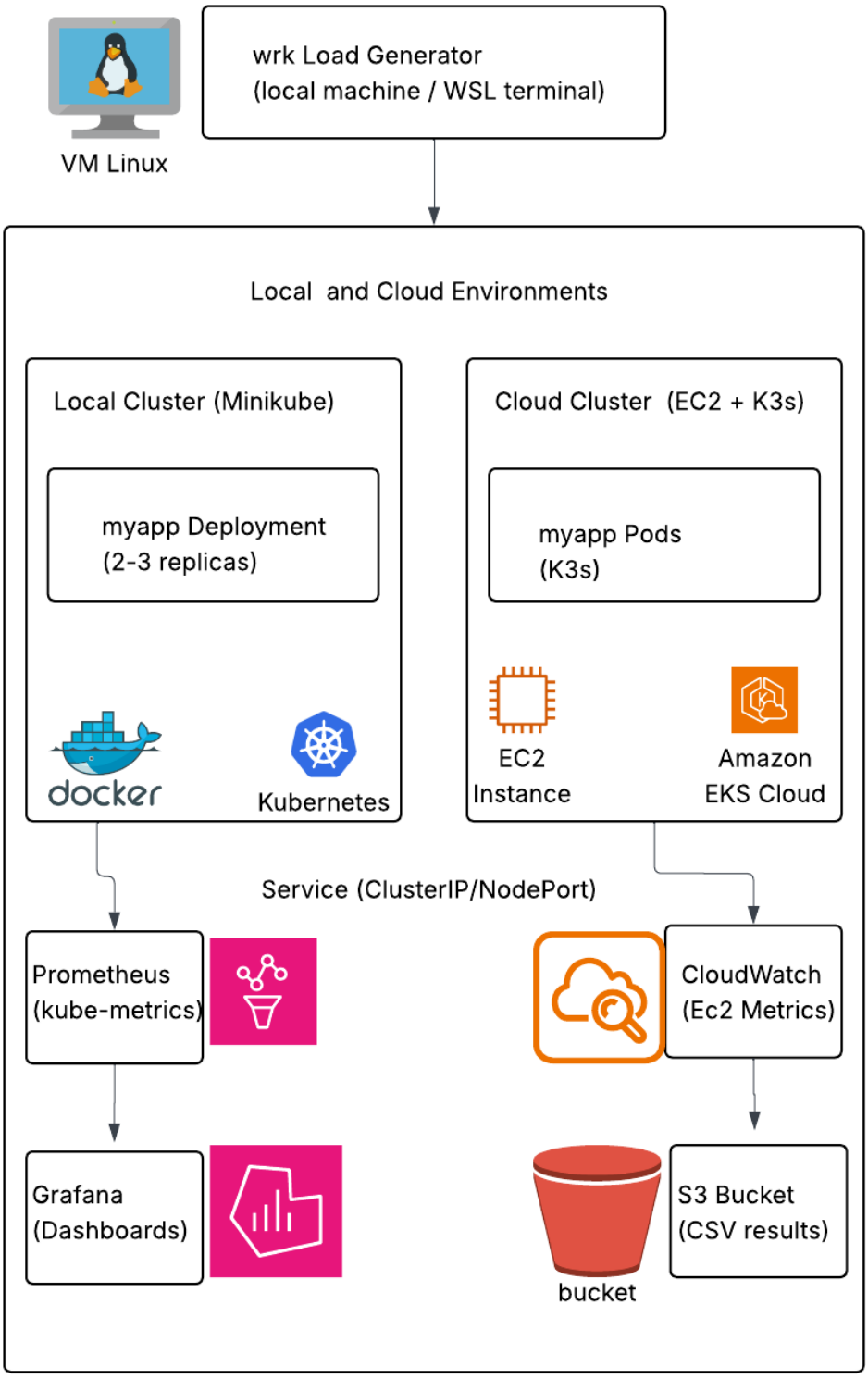

Figure 1: High-level architecture of the experimental setup across Minikube and K3s environments.

3.1 Overall Experimental Approach

Each experiment followed the same pattern. I deployed the same application image, `hashicorp/http-echo`, to both clusters (HashiCorp, 2025). Once the deployment was

stable, I used `wrk` (Gil Tene and Contributors, 2013) to send HTTP traffic for 60 seconds. While that traffic was still running, I injected a single fault into the system: a pod crash, a CPU stress job or a network delay and loss configuration. During and after the fault I observed how Kubernetes reacted using cluster events, timestamps, recovery time objective (RTO) measurements and metrics from Prometheus. I then repeated the same scenario on the cloud cluster with the same load and timing. Because the sequence stayed identical across all runs, differences in behaviour can reasonably be attributed to the environment or the failure type rather than to changes in the procedure.

3.2 Local Environment Setup (Minikube on WSL2)

The local testbed ran on a Windows 11 laptop using WSL2 with Ubuntu 22.04. Inside that environment I started Minikube with the Docker driver and allocated 4 GB of RAM and 2 vCPUs to the Minikube VM (Minikube Project, 2025a). Prometheus and Grafana were installed via the `kube-prometheus-stack` Helm chart in a `monitoring` namespace. Minikube behaves as a single-node Kubernetes cluster: it uses cgroups for resource limits, honours liveness and readiness probes, and restarts pods when they fail. This was enough to act as a realistic small cluster without the overhead of a multi-node setup. Prometheus scraped metrics continuously, and Grafana dashboards provided a live view of CPU usage, pod status and latency trends during each experiment.

3.3 Cloud Environment Setup (K3s on AWS EC2)

The cloud experiments used the same idea on AWS. I provisioned a `t3.small` EC2 instance with 2 vCPUs and 2 GB RAM running Amazon Linux 2023 (Amazon Web Services, 2025c). K3s was installed with the standard script:

```
curl -sfL https://get.k3s.io | sh -
```

K3s exposes the Kubernetes API in a lighter form (K3s Project, 2025a), which makes it practical on a small instance. Because the `t3.small` type uses CPU credits, the node could be throttled when heavily loaded, which was useful in the resource-stress experiments. CloudWatch tracked instance-level metrics such as `CPUUtilization` and basic network statistics (Amazon Web Services, 2025a). After each set of runs, `wrk` logs and summary CSV files were copied to S3 for later analysis (Amazon Web Services, 2025e).

3.4 Application Under Test (AUT)

To keep the focus on Kubernetes rather than application bugs, I used `hashicorp/http-echo` as the application under test (HashiCorp, 2025). The container is stateless and lightweight, and simply returns a short text string for each HTTP request. Because its behaviour is predictable, changes in performance or availability during faults are more likely to come from orchestration effects such as scheduling, throttling or networking rather than application logic. The deployment configuration used two replicas behind a Kubernetes Service in both environments, so requests always passed through the same abstraction.

3.5 Load Testing Setup (wrk)

Load generation relied on the `wrk` HTTP benchmarking tool (Gil Tene and Contributors, 2013). The command was kept identical for every run:

```
wrk -t2 -c50 -d60s http://<cluster-endpoint>/
```

This configuration uses two worker threads and fifty open connections for a continuous 60-second run, producing a steady stream of GET requests. It applied enough pressure to reveal performance changes without overwhelming either cluster. From each run I recorded requests per second, average latency, maximum latency and the standard deviation of response times. Because the `wrk` parameters were identical on Minikube and K3s, differences in performance can be traced back to how the environments and their schedulers behaved under the same workload.

3.6 Fault Injection Design

The fault injection design focused on three failure types that map to common operational issues: a sudden pod crash, heavy CPU pressure on the node and degraded network conditions. Together, they exercise different aspects of detection and self-healing.

3.6.1 Pod Crash (Unexpected Termination)

In the pod crash scenario, I simulated an abrupt failure by deleting the active `myapp` pod while `wrk` was running:

```
kubectl -n app delete pod <name>
```

This mirrors events such as an out-of-memory kill or a container crash: the pod disappears and Kubernetes has to notice and replace it. During each run I captured Kubernetes events and pod status transitions so that I could reconstruct the timeline from deletion, through scheduling and container creation, to the point where the replacement pod reached the Ready state. This timeline is used to calculate the RTO for this failure mode.

3.6.2 Resource Stress (CPU Saturation)

The resource-stress scenario examined what happens when the node is heavily CPU-bound rather than when a pod fails outright. I deployed an extra pod on the same node and ran:

```
stress --cpu 2 --timeout 60s
```

The `stress` process aggressively used CPU for a minute and competed with the `myapp` pods for processing time (King, 2025). This setup mirrors noisy neighbour workloads, runaway batch jobs or unexpectedly expensive microservices. It does not kill the service but starves it of CPU, which is equally damaging for latency. Running `wrk` during the stress window showed how throughput and response times changed when the node was effectively overloaded.

3.6.3 Network Impairment (Delay and Loss)

The third fault targeted the network. Instead of crashing anything, I added a sidecar container that used `tc qdisc netem` to introduce a fixed 200 ms delay and 1% packet loss on the traffic path (Linux man-pages project, 2025). In Minikube the netem container intercepted traffic within the node; in the cloud the same design was applied so that `myapp` always sat behind the artificial impairment. This reflects cases such as congested links, long cross-region paths or misconfigured VPNs where the service is healthy but responses are slow or occasionally dropped.

3.7 Metrics Collected

To understand what was happening in each experiment, I collected metrics at several levels. On the client side, the `wrk` output gave a user-centric view: requests per second, average latency, maximum response time and latency standard deviation. These describe what a user would actually experience when calling the service and form the core of the performance analysis.

Within the local cluster, Prometheus provided infrastructure-level data (Prometheus, 2025). Key metrics included `container_cpu_usage_seconds_total` for each pod, memory consumption, restart counts and probe transitions. Plotting these over time made it possible to line up throughput drops with CPU spikes or pod restarts. I also tracked Kubernetes state directly. Commands such as `kubectl get pods` before and after each fault gave snapshots of what was running, while the event log recorded delete, kill, recreate and readiness events. Event timestamps, combined with pod start times, were used to calculate RTO for the pod-crash experiments. All raw artefacts were kept, including event tails, pod listings, CSV summary files (`summary.csv`, `rto.csv`) and the individual `wrk` outputs, so that calculations can be checked without rerunning the experiments.

3.8 Data Processing and Visualisation

After the experiments, I processed the data with a small Python workflow. For each run the script parsed the `wrk` output and extracted throughput and latency metrics, then combined the Minikube and EC2/K3s results so that each fault type had a local and cloud counterpart. This combined dataset underpins the comparisons in the evaluation chapter.

Using Matplotlib, I produced a compact set of plots. Some focus on the local environment, others on the cloud environment, and several place the two side by side so that differences are immediately visible. In particular, I generated figures for requests per second, average latency and measured RTO for the pod-crash tests. These visualisations form the backbone of the analysis and help discuss patterns without listing individual numbers.

3.9 Ensuring Fair Comparisons

Because the project compares environments and failure modes, I enforced a few simple rules to keep the experiments fair. The `wrk` configuration (threads, connections and duration) was identical for every run, so the load profile did not change. Faults were always injected at the same point in the timeline, after the system had reached a steady

state. The Deployment and Service manifests were the same on Minikube and K3s, and I deliberately avoided autoscaling or extra controllers that might alter recovery behaviour. Before each new experiment I recreated the deployment so there was no leftover state or cache from earlier runs. These controls mean that differences in the results should come from how the environments behaved under stress, rather than from configuration drift or inconsistent test execution.

4 Design Specification

The design of this project was driven by a simple question: how far can plain Kubernetes get on its own when things go wrong. Before writing any scripts, I chose an architecture, workload and failure model that would give a fair answer without turning the project into a full production system. This section explains those design choices and the reasoning behind them; the concrete commands appear in the methodology.

4.1 Design Goals and Requirements

The main functional goal was to build a small but realistic setup where I could trigger repeatable failures and measure how Kubernetes behaved in response. The system needed to support pod crashes, resource pressure and network degradation, all while serving HTTP traffic from a simple microservice on a standard Kubernetes platform (Kubernetes Project, 2025a). At the same time, it had to be light enough to run on my laptop and on a single small cloud instance.

Non-functional requirements shaped the design just as strongly. Experiments had to be reproducible, so both environments needed to be easy to rebuild from standard tools such as Minikube and K3s (Minikube Project, 2025b; K3s Project, 2025b). Comparability was essential: if too many variables changed between local and cloud runs, differences in results would be meaningless. Observability was a first-class requirement rather than an afterthought, so the design included Prometheus, Grafana and CloudWatch from the start (Prometheus, 2025; Grafana Labs, 2025; Amazon Web Services, 2025b).

Requirement	**Design decision**
Repeatable failures	Script pod crash, CPU stress and network impairment on the same Deployment.
Comparable environments	Run single-node Minikube on WSL2 and single-node K3s on a `t3.small` EC2 instance with matching manifests.
Built-in observability	Use Prometheus and Grafana locally; rely on CloudWatch plus logs and artefacts in the cloud.
Lightweight footprint	Use a tiny stateless HTTP echo service, modest `wrk` load and small VM sizes.

Table 2: Core requirements and corresponding design decisions.

4.2 High Level Architecture

At a high level the system is a simple pipeline. A load generator sends HTTP requests into a Kubernetes Service that fronts a stateless application. Kubernetes schedules and

restarts pods, while a monitoring stack records metrics. Fault-injection components sit alongside the workload and create crashes, resource exhaustion or degraded network conditions. The same pattern is used in both environments so that results can be compared directly.

The full architecture diagram for this system is already presented in the Methodology section (Figure 1). That figure illustrates the shared layout used in both Minikube and K3s, including the application pods, Service, monitoring stack and fault-injection points.

Both clusters use a single logical node that runs the control plane, application pods, fault-injection tools and observability stack. This keeps the architecture easy to reason about while still exercising the controller loops, probes and scheduler that underpin Kubernetes self-healing (Kubernetes Project, 2025a,b).

4.3 Environment Design

The choice of Minikube on WSL2 for the local side and K3s on an EC2 instance for the cloud side was deliberate. Minikube behaves like a standard single-node Kubernetes cluster but runs entirely on my development machine (Minikube Project, 2025b). It was therefore ideal for iterating quickly and running many experiments without cloud costs, while still exposing realistic cgroup throttling, probe handling and pod restart behaviour.

In the cloud, K3s provided a similar experience in a hosted setting (K3s Project, 2025b). Running on a `t3.small` instance kept the setup modest but realistic and gave access to normal AWS infrastructure features (Amazon Web Services, 2025d). The single-node layout mirrors Minikube so that the two clusters are structurally comparable, yet the instance type introduces genuine cloud side effects such as CPU credit limits and virtualised hardware.

Monitoring follows the same logic. Locally, Prometheus and Grafana run inside the Minikube cluster and scrape metrics directly from pods and nodes (Prometheus, 2025; Grafana Labs, 2025). In the cloud, AWS CloudWatch provides node-level statistics while application logs and artefacts are collected from the cluster (Amazon Web Services, 2025b). In both cases the design assumes that serious evaluation of self-healing requires first-class observability.

4.4 Workload and Failure Model

The application under test is the `hashicorp/http-echo` service, which returns a short text message for every HTTP request (HashiCorp, 2025). Using such a simple, stateless workload avoids business logic and database complexity and keeps the focus on orchestration-level behaviour. If the service slows down or fails to recover, it is more likely due to scheduling, resource limits or networking than to application bugs.

The `wrk` tool provides a steady stream of HTTP traffic with a fixed configuration across all experiments (Gil Tene and Contributors, 2013). The load is high enough to reveal performance differences but low enough that both clusters can handle it without collapsing. Using the same `wrk` profile in Minikube and K3s is central to the design, because it turns the experiments into a controlled comparison rather than two separate benchmarks.

The failure model revolves around three scenarios that map to common incidents. A pod crash represents sudden failure at the container level. Resource stress models noisy neighbours or runaway jobs that starve the application of CPU. Network impairment

introduces delay and loss between client and service, which is typical when traffic crosses unstable links. Together these scenarios cover abrupt failure, gradual performance degradation on the node and network-level problems, matching patterns reported in earlier studies of Kubernetes failure modes (Gan et al., 2021; Zhang et al., 2022; Carrión, 2022).

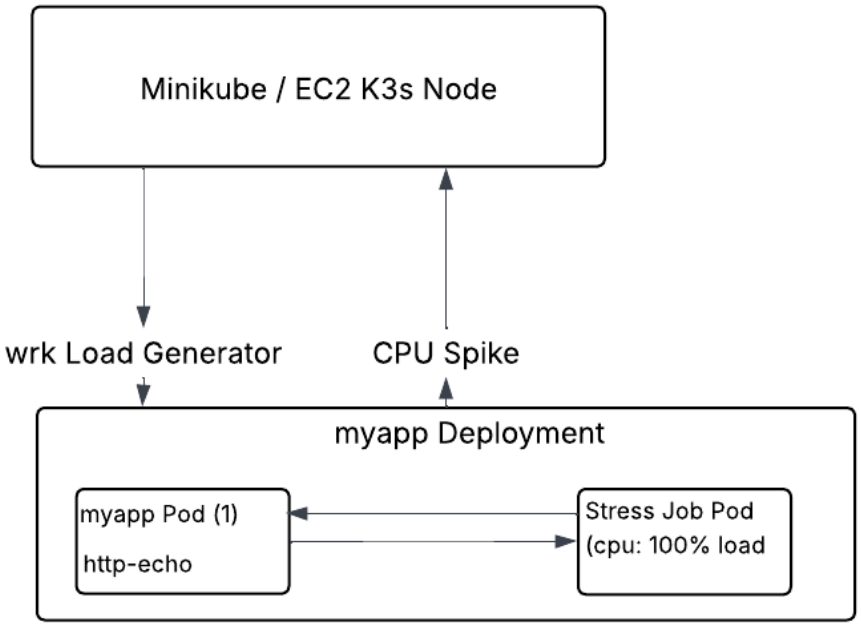

Figure 2: Design of the resource-stress scenario, where a CPU-burning pod competes with the application pod on the same node.

4.5 Observability and Data Flow

Data collection is designed to link user experience, cluster state and low-level metrics. The client-side `wrk` process produces statistics on throughput and latency for every run and acts as the primary measure of user-facing performance. Cluster metrics from Prometheus or CloudWatch capture CPU usage, pod restarts and other signals that help explain why performance changed (Prometheus, 2025; Amazon Web Services, 2025b). Kubernetes event logs and pod descriptions show when pods were deleted, recreated and marked as ready (Kubernetes Project, 2025a). The evaluation is based on aligning these sources in time: a dip in requests per second should line up with a CPU spike or pod restart, and the event log should confirm when the crash or restart occurred.

All data is written as plain files, such as CSV summaries and raw `wrk` logs, and stored either on the local filesystem or in an S3 bucket (Amazon Web Services, 2025f). This keeps the design simple and reproducible, without relying on an external database or analytics platform.

To give a visual sense of the resilience dimensions the design targets, Figure 3 sketches the main axes: recovery time, throughput impact, latency impact and error rate.

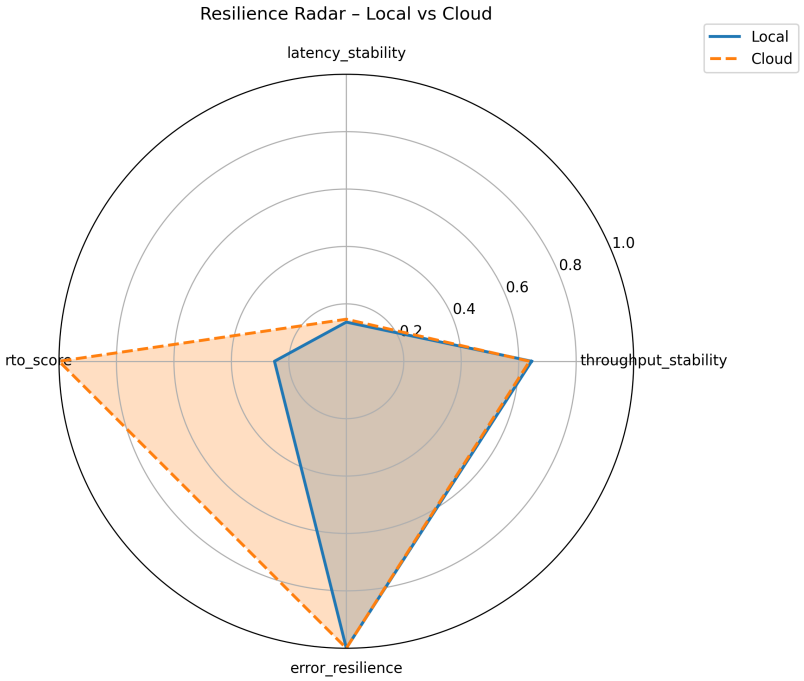

Figure 3: Conceptual resilience radar illustrating the main resilience dimensions targeted by the design: recovery time, throughput, latency and error rate.

4.6 Assumptions and Constraints

The design is shaped by a few explicit assumptions and constraints. Both clusters are single-node setups. This simplifies the system and keeps the comparison tight, but it means the experiments do not cover node failure or cross-node scheduling. The design also assumes there is no autoscaling or extra controllers such as service meshes or custom operators (Singh et al., 2024; Capozucca et al., 2021). Once those are added, it becomes hard to separate native Kubernetes behaviour from higher-level automation, which would undermine the core research question.

Experiment length and load level are also constrained. The system is intended for short runs with moderate traffic, not for multi-hour or production-scale stress tests. The aim is to capture clear patterns around crash recovery and performance degradation rather than to reach absolute throughput limits. Within these boundaries, the design provides a clean and consistent framework for the methodology and evaluation that follow, giving Kubernetes enough of a real-world environment to reveal how its self-healing behaves in practice while remaining simple enough to reason about and reproduce.

5 Implementation

This section explains how the experimental setup was built and executed in practice. The goal was not only to run fault-injection scenarios, but to implement them in a way that reflects how resilience testing is usually approached in real Kubernetes environments.

Rather than repeating the earlier environment descriptions, the focus here is on the concrete manifests, artefacts, workflows and scripts that made the experiments run end to end.

5.1 Application and Kubernetes Manifests

The entire system revolved around a single lightweight microservice, so the first implementation step was to create the Kubernetes manifests for hosting the `hashicorp/http-echo` container. The Deployment defined two replicas, selector labels and a minimal probe configuration (Kubernetes Project, 2025a). A simple Service exposed the pods through either a NodePort or a ClusterIP, depending on whether the request originated locally or from a port-forward. The manifests were intentionally small so that any interesting behaviour would come from Kubernetes itself rather than from application logic.

Additional manifests supported the fault-injection scenarios. The `myapp-toxiproxy` manifest introduced a proxy sidecar used for network impairment, ensuring that all traffic passed through a container configured to apply delay and packet loss. For the CPU-stress scenario, a short-lived `stress-job.yaml` defined a Job that consumed compute for a fixed duration. These files provided deterministic and repeatable ways to trigger failures without modifying the base workload.

5.2 Execution and Coordination of Experiments

With the manifests prepared, each experiment followed a consistent flow. The Deployment was applied and both replicas were confirmed to be Ready. The `wrk` load generator was then directed at the cluster endpoint, either by NodePort in K3s or by `kubectl port-forward` in Minikube. Traffic was maintained for the duration of each trial so that steady-state behaviour was established before any faults occurred.

During this load phase, the fault was introduced at a predictable point in the timeline. Pod-crash experiments removed the active `myapp` pod using `kubectl delete pod`, which forced Kubernetes to reschedule and recreate it. CPU-stress experiments triggered the Job that consumed all available compute on the node. Network-fault experiments enabled the proxy configuration that applied 200 ms delay and 1% loss. Throughout each run, the cluster state was monitored using `kubectl get pods` and `kubectl get events`, and the resulting event logs and pod transitions were stored in the `experiments/` directory.

Cloud experiments used the same process but relied more heavily on CloudWatch for observing CPU throttling and network behaviour (Amazon Web Services, 2025b).

5.3 Data Extraction and Processing Pipeline

All raw artefacts were processed through a small Python analytics workflow. The `plot_local_results.p` script parsed Minikube `wrk` logs, extracted throughput and latency values and generated the local-only figures. The cloud results were processed in the same way by `plot_cloud_results.py`, which consumed pre-exported CSVs from S3.

Once both datasets were available, `plot_combined_results.py` merged local and cloud data to generate comparative visualisations. Additional scripts provided more specific outputs. A degradation script identified how latency and throughput deviated from baseline. Another extracted error rates by parsing individual `wrk` logs for timeouts and connection failures. The RTO timeline script processed `rto.csv` to reconstruct the

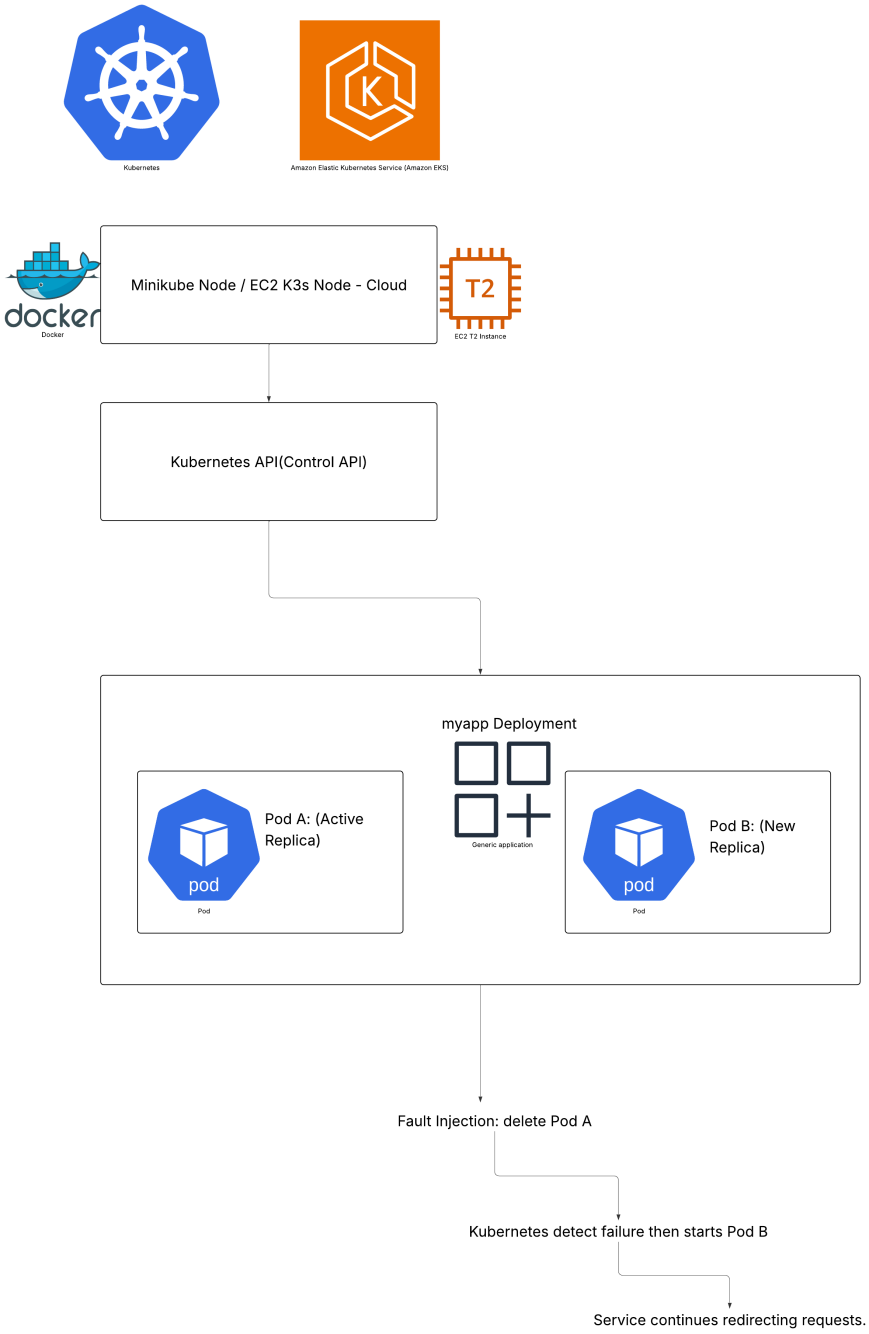

Figure 4: Execution timeline for pod-crash experiments, showing deletion, rescheduling and recovery.

exact sequence of pod deletion, scheduling, container start and readiness. Finally, the radar-plot script produced a multi-metric resilience profile summarising recovery time, throughput loss, latency impact and error rate.

These scripts formed a complete pipeline that transformed raw logs into structured visual outputs, ensuring that the evaluation relied on measurable evidence rather than on impressions gathered during live testing.

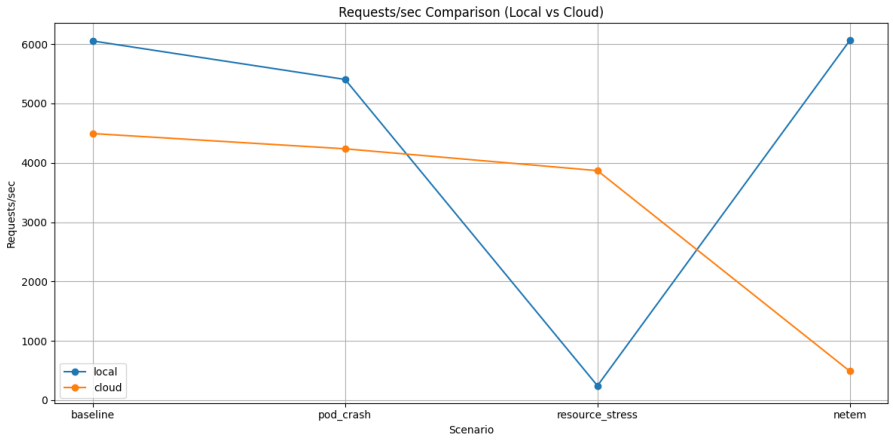

Figure 5: Combined throughput plot (requests/sec) for local vs. cloud environments.

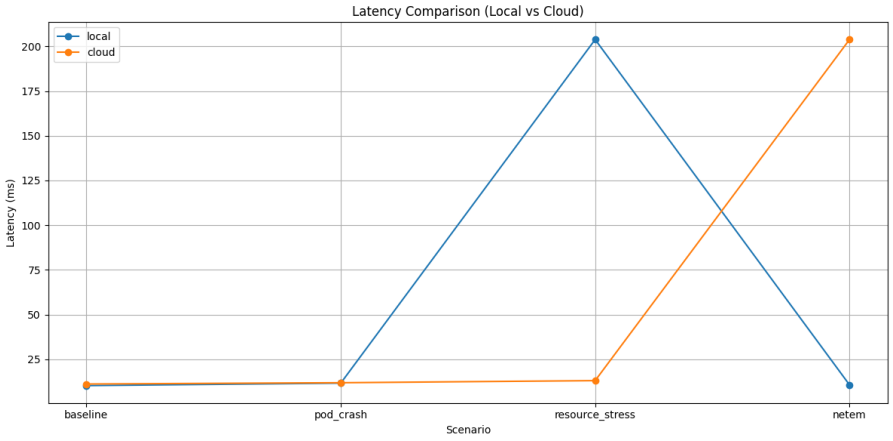

Figure 6: Combined latency comparison across failure scenarios.

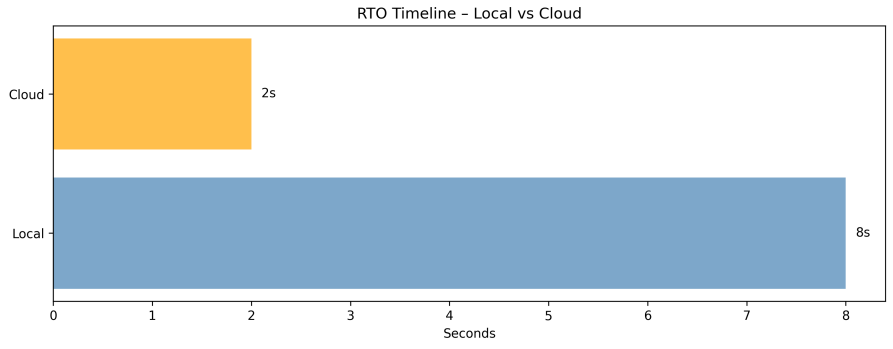

Figure 7: Recovery time objective (RTO) comparison calculated from event logs and `rto.csv`.

5.4 Ensuring Reproducibility and Traceability

Reproducibility was a major design goal. All artefacts generated during experiments including `wrk` logs, event output, pod listings, summary CSVs and final figures were stored systematically in the `experiments/` directory. File names followed a consistent pattern so the analysis scripts could locate their inputs without modification.

The Kubernetes manifests were stored in the `k8s/` directory and remained unchanged throughout the project to avoid configuration drift. The Python virtual environment was also preserved so that figures could be regenerated without reinstalling dependencies. With access to Minikube and K3s, any engineer could re-run the experiments, reprocess the data and reproduce every plot in the evaluation.

Artefact	Location	Used By
wrk logs	`experiments/*/wrk_*.txt`	Throughput/latency extraction
Event logs	`experiments/*/events_tail.txt`	RTO calculations
Summary CSVs	`summary.csv`, `rto.csv`	Combined comparison scripts
K8s manifests	`k8s/*.yaml`	Deployment and fault injection
Generated plots	`figures/*.png`	Evaluation section

Table 3: Implementation artefacts and their role in the analysis workflow.

The final outcome is a fully reproducible workflow that covers deployment, failure injection, data collection and visualisation. This implementation foundation supports the evaluation in the next section and allows the results to be regenerated exactly as they appear in the report.

6 Evaluation and Analysis

This section brings together the results from all fault scenarios and compares how Kubernetes behaved in the two environments used in the study: a local Minikube cluster and a small K3s node on EC2. The focus is on what happened during each failure, how the system recovered, and what this says about Kubernetes self-healing, rather than on the mechanics of running the tests.

6.1 Baseline Behaviour

Before introducing any failures, both clusters were tested under a clean baseline run. Minikube handled roughly 6,050 requests per second with an average latency of about 10 ms. The cloud cluster, which had fewer resources, stabilised at around 4,490 requests per second and just over 11 ms average latency. This gap was expected, because the EC2 node is a small virtual machine that runs both the control plane and the workload, while Minikube benefits from stronger local hardware.

The key point is that both environments produced clean, low-latency baselines without errors. All later results are interpreted relative to these values.

Environment	Requests/sec	Avg. latency (ms)
Minikube (local)	$\approx$ 6,050	$\approx$ 10
K3s on EC2 (cloud)	$\approx$ 4,490	$\approx$ 11

Table 4: Baseline summary metrics for local and cloud clusters.

6.2 Pod Crash: How Fast Does Kubernetes Heal?

The pod-crash scenario is where Kubernetes behaved closest to its design goals. When the active `myapp` pod was deleted during load, Minikube saw throughput dip by roughly

ten percent and latency rise by a similar amount. It took around eight seconds before a replacement pod reached the Ready state and the system settled back to normal.

The cloud environment showed the same pattern but recovered faster. Throughput fell by roughly six percent and average latency rose only slightly. The replacement pod appeared very quickly and the measured recovery time was around two seconds. The `wrk` logs showed a short spike of read errors and timeouts, but the error rate remained under one percent in both clusters.

Environment	RTO (s)	Max. error rate (%)
Minikube (local)	≈ 8	< 1
K3s on EC2 (cloud)	≈ 2	< 1

Table 5: Pod-crash recovery time and observed error rates.

This aligns with what prior work reports for straightforward failures (Gan et al., 2021; Kambala, 2025). Pod crashes are exactly the kind of event Kubernetes was designed to handle.

6.3 Resource Stress: A Noisy Neighbour on the Node

Resource-stress tests told a different story. When the stress Job consumed CPU aggressively for a full minute, Minikube almost collapsed under the load. Throughput fell from around 6,000 requests per second to roughly 245, and average latency jumped to more than 200 ms. The application pod did not crash and Kubernetes considered it healthy throughout.

The EC2 node experienced the same kind of pressure but degraded more gracefully. Throughput dipped by about fourteen percent and latency increased slightly.

Environment	Baseline RPS	Stress RPS	Avg. latency (ms)
Minikube (local)	$\approx 6{,}000$	≈ 245	> 200
K3s on EC2 (cloud)	$\approx 4{,}500$	$\approx 3{,}850$	Slight increase

Table 6: Throughput and latency changes during CPU-stress experiments.

6.4 Network Degradation: Slow but Not Broken

The network fault highlighted another blind spot. On Minikube, throughput stayed close to the baseline and only modest increases in latency appeared.

The cloud environment reacted more dramatically: throughput fell below 500 requests per second and latency rose significantly.

Environment	Baseline RPS	Degraded RPS	Avg. latency (ms)
Minikube (local)	$\approx 6{,}050$	Similar to baseline	Slightly higher
K3s on EC2 (cloud)	$\approx 4{,}490$	< 500	Much higher

Table 7: Throughput and latency behaviour under network delay and loss.

6.5 Local vs. Cloud: Patterns Across All Faults

Looking across all scenarios, several patterns stand out. Minikube delivered a stronger baseline but suffered heavily under CPU stress. The cloud cluster showed lower baseline throughput but more stability under load. Pod crashes were handled well in both environments, with the cloud recovering faster.

Scenario	**Minikube behaviour**	**Cloud behaviour**	**K8s reaction**
Baseline	High throughput, low latency	Lower but stable throughput	Not applicable
Pod crash	Small dip, quick recovery	Smaller dip, faster recovery	Pod restart succeeds
CPU stress	Severe degradation	Moderate degradation	No self-healing action
Network delay	Mild latency increase	Large throughput drop	No self-healing action

Table 8: Summary of observed patterns across scenarios in local and cloud environments.

6.6 Limitations and What Was Learned

This study was run on single-node clusters in both environments, so the experiments did not cover node failures, multi-node scheduling or replica movement across nodes. Each experiment lasted about a minute, which means the results do not capture long-term issues such as slow memory leaks or microservice aging effects (Flora et al., 2022). The application under test was also intentionally simple, so the system did not face real-world problems such as database failures or multi-tier outages.

The cloud environment introduced additional practical constraints. The AWS Academy Learner Lab blocks most IAM actions, which meant I could not assign an instance role or run the CloudWatch Agent. As a result, only the default EC2 metrics were available. Attempts to install the agent succeeded, but the logs showed missing credentials and metadata errors, so the agent never produced custom metrics. Although this limited observability, the default CPU and network metrics were still enough to understand the main behaviour of the system.

A few operational challenges also appeared during testing, such as keeping port-forwards active, choosing useful Prometheus metrics, and dealing with temporary AWS CLI credentials when uploading results to S3. These are normal issues that teams face when running live experiments, and they shaped how the tests were executed in practice.

6.7 Comparing Results with External Resilience Tools

Once the results are in front of you, it becomes clearer why tools such as Istio, RAMSES and proactive recovery frameworks are common in the literature. Kubernetes handled pod crashes very well, but it struggled with more subtle problems such as CPU contention and network delay. That pattern lines up with what Gan et al. (2021), Capozucca et al. (2021) and Singh et al. (2024) describe: native self-healing is strong for clean failures but weak for "alive yet degraded" services.

The pod-crash behaviour here looks similar to the Istio chaos tests in Singh et al. (2024). In their case, Istio smooths out the brief disruption by retrying failed requests and shaping traffic, which Kubernetes does not provide by default. The resource-stress scenario reflects the situations discussed by Gan et al. (2021) and motivates designs like RAMSES (Capozucca et al., 2021), which explicitly targets pods that are still running but no longer delivering acceptable performance. The network-delay results echo anomaly-detection and adaptive-controller work, where Kubernetes misses slow-burn issues because its probes only check simple liveness or readiness signals (Matsuo and Ikegami, 2021; Myrtollari et al., 2025; Tran et al., 2022).

Overall, these comparisons help position this work as a clean baseline. On its own, Kubernetes is stable and predictable for straightforward crashes, but blind to many issues that users feel first. That is exactly the gap external resilience tools are intended to fill: they add smarter detection, retries and adaptive control on top of the behaviour measured in these experiments.

7 Conclusion and Future Work

This project set out to answer a simple research question: *How effectively does native Kubernetes self-healing respond to common microservice failures in local and cloud environments, without relying on service meshes or external resilience frameworks?* The experiments were designed to isolate Kubernetes and observe how well its built-in mechanisms cope with pod crashes, resource pressure and network degradation.

The results show a clear pattern. Kubernetes performs very well when the failure matches the behaviour it is designed to detect. Pod crashes were handled quickly in both environments, the service stayed online and disruption was brief. This confirms what prior research reports: Kubernetes is strongest when dealing with clean, binary failures such as container exits or restarts.

The other scenarios exposed the limits of native self-healing. Under heavy CPU pressure, performance dropped sharply even though Kubernetes considered the application healthy. Network delay produced similar problems, especially in the cloud environment, where throughput collapsed despite the pod never failing its probes. These results support the findings in the literature on hidden failures and slow degradation, where Kubernetes does not react because the workload never becomes formally "unhealthy".

The comparison between Minikube and K3s also mattered. The cloud node recovered more quickly from pod crashes but reacted more noticeably to network delay, while Minikube struggled more under CPU contention. This shows that resilience is shaped not only by Kubernetes itself but by the underlying hardware and how resources are isolated.

Running the experiments also highlighted real operational challenges. Setting up observability, keeping port-forwards alive, collecting usable metrics and working within the restrictions of the AWS Learner Lab all shaped the process. These practical constraints reinforced the importance of choosing metrics that actually reflect the failure modes being tested.

There are several ways this work could be extended. A multi-node cluster would allow rescheduling behaviour to be observed directly. Longer-running tests could uncover slow memory leaks or aging effects. A more complex application with real dependencies would make it possible to study cascading failures. It would also be valuable to repeat these

experiments with tools such as Istio or RAMSES to compare plain Kubernetes against enhanced resilience frameworks.

In summary, the project meets its objective by providing a clear, evidence-based baseline for native Kubernetes resilience. The findings show that Kubernetes is reliable for clean crashes but far less responsive to resource saturation or gradual degradation. This helps clarify when native mechanisms are enough, and when additional tooling, richer detection, or adaptive control would be needed in real systems.

References

Al-Arif, B., van der Meulen, R. and Pouwelse, J. (2021). Defektor: An extensible tool for fault injection campaign management in microservice systems, *IEEE/ACM International Conference on Utility and Cloud Computing Companion*, pp. 223–230.

Amazon Web Services (2025a). Amazon cloudwatch user guide. Accessed: 15 October 2025.
URL: *https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.htm*

Amazon Web Services (2025b). Amazon cloudwatch user guide. Accessed: 15 October 2025.
URL: *https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.htm*

Amazon Web Services (2025c). Amazon elastic compute cloud (amazon ec2) documentation. Accessed: 15 October 2025.
URL: *https://docs.aws.amazon.com/ec2/index.html*

Amazon Web Services (2025d). Amazon elastic compute cloud (amazon ec2) documentation. Accessed: 15 October 2025.
URL: *https://docs.aws.amazon.com/ec2/index.html*

Amazon Web Services (2025e). Amazon simple storage service (amazon s3) user guide. Accessed: 15 October 2025.
URL: *https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html*

Amazon Web Services (2025f). Amazon simple storage service (amazon s3) user guide. Accessed: 15 October 2025.
URL: *https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html*

Baumann, L., Benz, S., Militano, L. and Bohnert, T. (2019). Monitoring resilience in a rook-managed containerized cloud storage system, *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 89–96.

Capozucca, A., Presti, M. and Tamburri, V. (2021). Ramses: An artifact exemplar for engineering self-adaptive microservice applications, *IEEE Transactions on Services Computing* **14**(6): 1590–1602.

Carrión, M. (2022). Kubernetes scheduling: Taxonomy, ongoing issues and challenges, *ACM Computing Surveys* **55**(1): 1–27.

Flora, J., Gonçalves, P., Teixeira, M. and Antunes, N. (2022). A study on the aging and fault tolerance of microservices in kubernetes, *IEEE Access* **10**: 132786–132798.

Gan, Y., Deligiannis, P. and Venkataraman, S. (2021). Mutiny! how does kubernetes fail and what can we do about it?, *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pp. 44–59.

Gil Tene and Contributors (2013). wrk: A modern http benchmarking tool. Accessed: 15 October 2025.
URL: *https://github.com/wg/wrk*

Grafana Labs (2025). Grafana documentation. Accessed: 15 October 2025.
URL: *https://grafana.com/docs/*

HashiCorp (2025). http-echo: Simple http server for testing. Accessed: 15 October 2025.
URL: *https://github.com/hashicorp/http-echo*

K3s Project (2025a). K3s: Lightweight kubernetes documentation. Accessed: 15 October 2025.
URL: *https://docs.k3s.io/*

K3s Project (2025b). K3s: Lightweight kubernetes documentation. Accessed: 15 October 2025.
URL: *https://docs.k3s.io/*

Kambala, V. (2025). Leveraging kubernetes for self-healing microservices: A comparative analysis and future directions, *International Conference on Emerging Information Technologies (ICoEIT)*.

King, C. I. (2025). stress-ng: POSIX system stress tool. Accessed: 15 October 2025.
URL: *https://github.com/ColinIanKing/stress-ng*

Kubernetes Project (2025a). Kubernetes documentation. Accessed: 15 October 2025.
URL: *https://kubernetes.io/docs/*

Kubernetes Project (2025b). Kubernetes scheduling and eviction: kube-scheduler overview. Accessed: 15 October 2025.
URL: *https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/*

Linux man-pages project (2025). tc-netem(8) — network emulator for tc. Accessed: 15 October 2025.
URL: *https://man7.org/linux/man-pages/man8/tc-netem.8.html*

Matsuo, Y. and Ikegami, D. (2021). Performance analysis of anomaly detection methods for application systems on kubernetes with auto-scaling and self-healing, *17th International Conference on Network and Service Management (CNSM)*, pp. 464–472.

Minikube Project (2025a). Minikube documentation. Accessed: 15 October 2025.
URL: *https://minikube.sigs.k8s.io/docs/*

Minikube Project (2025b). Minikube documentation. Accessed: 15 October 2025.
URL: *https://minikube.sigs.k8s.io/docs/*

Müller, R., Meinhardt, C. and Mendizabal, O. (2022). An architecture proposal for checkpoint/restore on stateful containers, *37th ACM/SIGAPP Symposium on Applied Computing (SAC '22)*, pp. 267–270.

Myrtollari, K., Andreou, C., Panagidi, K. and Hadjiefthymiades, S. (2025). An unsupervised anomaly-based detection system for microservices applications on kubernetes, *International Conference on Software Architecture and Systems*, pp. 244–256.

Pandey, P. (2025). Scalable and resilient microservices deployment for edge computing with kubernetes, *International Journal of Edge and Cloud Computing* **4**(1): 22–34.

Prakash, R. (2023). *Benchmarking container orchestration tools on application performance in the cloud*, Master's thesis, National College of Ireland, Dublin, Ireland.

Prometheus (2025). Prometheus documentation. Accessed: 15 October 2025.
URL: *https://prometheus.io/docs/*

Rodriguez, M., Calheiros, R. and Buyya, R. (2018). Reactive auto-scaling of containers in cloud computing environments, *Future Generation Computer Systems* **79**: 38–53.

Rouf, Y., Mukherjee, J., Litoiu, M., Wigglesworth, J. and Mateescu, R. (2021). A framework for developing devops operation automation in clouds using components-off-the-shelf, *ACM/SPEC International Conference on Performance Engineering (ICPE '21)*, pp. 265–276.

Samarakoon, S., Jayasanka, N., Bandara, S. and Hettiarachchi, C. (2023). Self-healing and self-adaptive management for iot-edge computing infrastructure, *Journal of Distributed Systems and Applications* **11**(3): 140–152.

Shabariram, C., Srivatsan, S., Vrinda, S. and Manimeghalai, R. (2024). Case study based investigation on self-healing cloud deployments for edge-based software development, *2024 International Conference on Smart Systems for Electrical, Electronics, Communication and Computer Engineering (ICSSEECC)*, pp. 85–90.

Singh, M., Arora, S. and Sharma, P. (2024). Boosting microservice resilience: An evaluation of istio's impact on kubernetes clusters under chaos, *IEEE Access* **12**: 65798–65812.

Tran, M.-N., Vu, X. and Kim, Y. (2022). Proactive stateful fault-tolerant system for kubernetes containerized services, *IEEE Access* **10**: 102181–102194.

Zhang, H., Li, Y. and Li, Y. (2022). My services got old! can kubernetes handle the aging of microservices?, *IEEE International Conference on Cloud Engineering (IC2E)*, pp. 132–144.