Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

Resilience in Multi-Cloud Kubernetes

Claire Gregg

Supervisor: Stefan Weber

April 18th, 2025

A Thesis submitted in partial fulfilment
of the requirements for the degree of
Master of Computer Science

Resilience in Multi-Cloud Kubernetes

Claire Gregg, Master of Computer Science

University of Dublin, Trinity College, 2025

Supervisor: Stefan Weber

Cloud deployments are increasingly common for application developers. Application reliability is paramount, and massive amounts of money can be lost in just a few minutes of downtime. There are a number of cloud providers on the market - most popularly Azure, Amazon Web Services (AWS), and Google Cloud Platform (GCP). Deployment approaches for cloud applications generally stay within a single cloud provider. These deployments are most often zonal, regional, or multi-regional, where the application may be replicated across zones and regions to provide reliability against zonal and regional failures. These deployment approaches do not handle the scenario where a cloud provider has a global failure. Multi-cloud deployments are used to handle this failure scenario.

The goal of this project is to compare the resilience of applications deployed across multiple clouds - multi-cloud applications - against more standard deployment approaches. A multi-cluster Kubernetes application using Istio was developed for testing resilience. Deployment was attempted on a single bare metal server due to financial limitations, but was unsuccessful. Evaluation against the theoretical design was performed using binary and graph resilience metrics, and analysis of its intended behaviour in disaster scenarios. The multi-cloud application was found to be more resilient than the single cloud approach by all metrics.

Acknowledgments

First, I must thank my supervisor, Stefan Weber, for his consistent support, follow ups, and feedback throughout the months working on this project. His work to try to get me access to resources for this project, and support when I changed directions with my work were invaluable.

My thanks go to my parents, for keeping me going, my sisters, for always being on my side, and to my friend Abigail, for going through this with me.

My greatest thanks go to my partner Laura, for all the conversations, all the support (technical and otherwise), and for literally handling my infrastructure needs through the server in your closet.

I couldn't have done this without all of you, thank you.

Claire Gregg

University of Dublin, Trinity College
April 2025

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

We rely on cloud applications in so much of our lives - from banking, to taxes, to education, to television, to gaming, to keeping in contact with friends and family. Cloud computing has allowed rapid development, deployment, and scaling of applications, without having to make an initial investment in servers, or continued investments into maintenance. However, this has introduced a new area of risk that application developers have no control over - what happens if the cloud resources fail [25]?

Cloud providers logically separate their resources at two levels - regions and (availability) zones.[1,2,3] Regions are independent geographic areas. These are often used to locate resources closer to users. Regions are broken down into zones, which are intended to be physically and logically separated, and should fail separately. Cloud customers are recommended to deploy any resources they want to be highly available to multiple zones. An example of a single zone failure can be seen in AWS's outage in June 2018 [46]. In this incident, a "power event" caused connectivity issues for many AWS services in an availability zone in the `us-east-1` region. It is for incidents like these that customers are recommended to use multi-zone deployments.

However, deploying to multiple zones may not be sufficient to have a reliable application. In December 2021, AWS had a region-wide failure caused by a network device overload and automated scaling rules [6]. In 2020, Azure had a region-wide outage in its East US region due to a temperature spike [19]. In April 2023, a water leak triggered a fire in Google's `europe-west9-a` zone, impacting the full region [29]. With these and other failures, there has been a significant effort made towards having multi-region applications over multi-zone applications, where reliability is required.

[1] AWS: `https://aws.amazon.com/about-aws/global-infrastructure/regions_az/`

[2] Azure: `https://learn.microsoft.com/en-us/azure/reliability/availability-zones-overview?tabs=azure-cli`

[3] GCP: `https://cloud.google.com/docs/geography-and-regions`

However, even multi-region applications are not fully protected against outages. In April 2016, all compute services in GCP globally lost connectivity for 18 minutes [20]. In July 2024, Azure had a global outage caused by a DDoS attack [47].

The impact of these outages can be massive - take for example a regional outage in AWS in 2017, which is estimated to have lost companies at least $150 million [50]. The solution proposed for this problem is to deploy applications which require high levels of reliability and availability to multiple clouds. This protects against even a full cloud provider failing. An example of a multi-cloud deployment in industry is MongoDB's managed database solution, MongoDB Atlas, described as "the modern multi-cloud database"[4], which supports deploying over Azure, AWS, and GCP.

1.1 Multi-cloud Systems

There are multiple types of clouds [27]:

Public Cloud This is what people generally mean when discussing cloud computing. A centralised company buys and maintains all resources, and allows customers to pay to use them. AWS, Azure and GCP are all public clouds.

Private Cloud An organisation's internal data centre, with resources bought, managed, and maintained by the company. This is significantly more secure than the public cloud.

Hybrid Cloud Where a combination of public and private clouds are used, to gain the benefit of both cloud types. Often, sensitive data will be stored in the private cloud, while pushing large compute loads to the public cloud.

Community Cloud Where several organisations come together to build and maintain a cloud, which is shared between those groups.

This project will be focusing on public cloud as the most commonly discussed cloud type.

Berenberg et al. [9] describes six archetypes for public cloud deployment, also shown in Figure 1.1.

1. **Zonal** - Where the full application runs only in one cloud zone, optionally with failovers in another zone in the same region.
2. **Regional** - Where the application is deployed and serves across several zones in the same region, optionally with failovers in another region.
3. **Multi-regional** - Where the application is deployed and serves across multiple regions.
4. **Global** - Where the application is deployed and serves globally, typically by using global

[4] `https://www.mongodb.com/atlas`

versions of databases and other cloud resources. Required for most "always-on" services like retail and social media.

5. **Hybrid** - As discussed above, where the application is deployed and serves both in the public cloud and on-premise (private cloud). Berenberg adds a discussion of hybrid cloud's use in provide failovers between the public and private cloud.
6. **Multi-cloud** - Where the application is deployed and serves across two or more public cloud providers. For each cloud provider, one of zonal, regional, multi-regional or global deployment archetypes is used.

All of these archetypes increase application reliability more as the list goes on, with multi-cloud being the most reliable, especially if applications are deployed globally within each cloud provider.

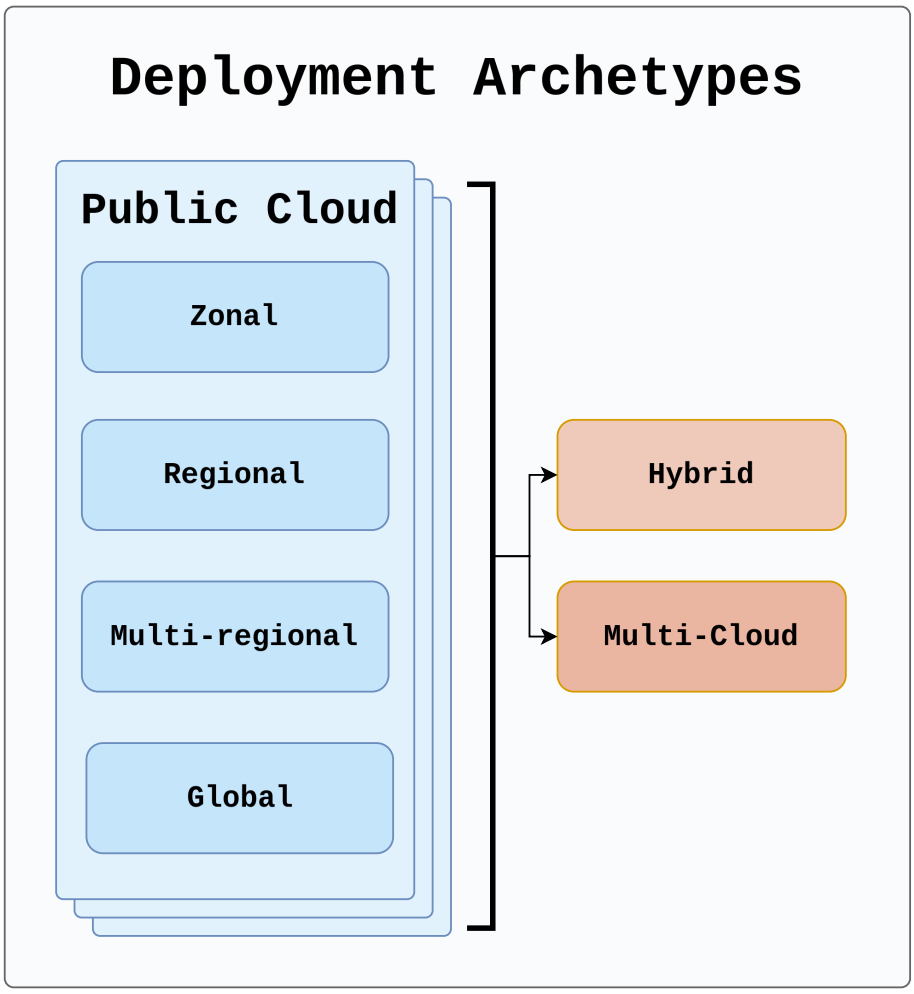


Figure 1.1: Cloud Deployment archetypes according to Berenberg et al. [9]. Within a public cloud, deployments can be done zonally, regionally, across multiple regions, or globally. These deployment approaches in the public cloud can be combined with a private cloud in **hybrid** deployment, or with one or more public clouds in **multi-cloud** deployment.

Sharma [45] discusses the reasoning behind multi-cloud deployments. Single cloud so-

lutions are easier to develop, deploy, and maintain. However, they lead to vendor lock-in, where a system has a dependence on tools only provided by the cloud provider they are using. This gives developers less freedom in their development. Instead, multi-cloud systems provide space for innovation and the ability to use the best tool available for each individual problem. Most importantly, multi-cloud systems mitigate risks, by removing the dependency on a single system. These multi-cloud systems come at the cost of complexity, security, and financial cost.

1.2 Kubernetes

Kubernetes is an open source container orchestration tool, first developed in 2014 by Google. It was based on their internal container management tools, Borg and Omega, taking learnings from the development and use of these tools [11]. Since then, it has become incredibly popular [35]. It is used for the deployment, management, discovery, and scaling of containerised applications - for example those running through Docker containers. It is most useful for microservice applications, where the services provided by the application are logically separated into smaller sub-services, with each deployed in a container

Kubernetes groups machines together into **clusters** [31]. A cluster consists of multiple machines called **nodes**, shown in Figure 1.2. Each cluster has at least one control plane node, which manages the overall state of the cluster. The cluster should also have worker nodes, on which the application microservices run. The microservices run on **pods** on the worker nodes. Kubernetes **Services** are defined to route messages to pods with a specific label - for example the service `nginx` might route to pods with the label `app=nginx`. With this, service discovery and load balancing happens automatically, and pods with frequently changing IP addresses have a stable endpoint for access. Within a Kubernetes cluster, **namespaces** allow for logical separation of components.

For more complex applications, especially those distributed over large geographical areas, multiple Kubernetes clusters can be combined.

As an orchestration system for distributed applications, Kubernetes was designed for fault tolerance and resilience. The easiest way to describe this is by discussing each control plane component in turn, and discussing how it contributes to the reliability of Kubernetes as a whole [33]. As such, the rest of this section will discuss the primary control plane components, as well as a common extension, specifically with a focus on resilience.

Before talking about each component, it is valuable to discuss **state** in Kubernetes. By default, Kubernetes pods are expected to be **stateless**. Practically speaking, this means that if the pod is replicated, there should be no difference between the original pod and the replica. An example of a **stateful** pod would be a frontend web server which caches some data about

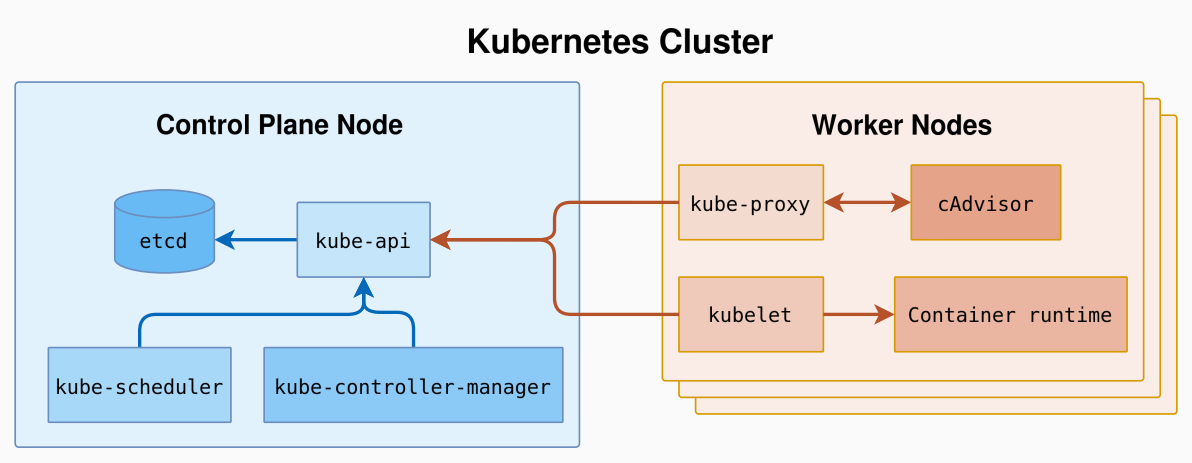


Figure 1.2: The structure of a Kubernetes cluster. Each node represents a machine in the cluster. The control plane node manages the cluster's state, while the worker nodes handle the actual microservices. On each worker node, the application components run on pods (which contain application containers). The pods run on the container runtime of the node - usually Docker. The `kubelet` ensures pods are running as expected, while the `kube-proxy` manages communication, service discovery and load balancing, and `cAdvisor` (container advisor) monitors resource usage, supplying this data to other components. On the control plane, `etcd` is the backing store for cluster state, `kube-scheduler` assigns new pods to nodes, `kube-controller-manager` works to get the cluster into desired states, and `kube-api` is used to interact with the control plane.

users *in-memory*. To make this pod stateless, this data would have to be entirely within an external database. There is support for stateful pods in Kubernetes - especially given it is required for databases. Instead of replicating a pod using a `ReplicaSet`, it is possible to do so with a `StatefulSet`. However, wherever possible, it is best to maximise the number of stateless pods, as they are far easier to replicate safely, given no syncing between the pods is required.

1.2.1 `kube-api`

The `kube-api` is the point of contact for the Kubernetes control plane. Any contact from tools attempting to contact or configure the cluster go to the `kube-api`, as does all communication from worker nodes. This component manages the state of the cluster, and is the only component which is allowed to contact the cluster's backing store - `etcd`.

As all data is stored in `etcd`, `kube-api` is stateless. Because of this, it is able to be scaled horizontally - by bringing up many new `kube-api` instances, including on different servers, as long as they all connect to synchronised `etcd` instances. This ensures the resilience of this important component, as several instances should always be available to contact.

1.2.2 `etcd`

Cluster state needs to be stored somewhere - this state includes information like pod manifests, cluster secrets, and more. This ensures that if the cluster is restarted (intentionally or after failure), the cluster state is not lost. The recommended backing store is `etcd`, as it is efficient and can remain consistent when distributed. This means it can be replicated across multiple servers, with the data remaining the same across the replicas. This ensures resilience of data in case one instance goes down.

`etcd` remains consistent between replicas [18] using the Raft consensus algorithm [38]. This is **strong consistency**, which in this case means the following qualities apply:

1. **Durable** - any operation successfully completed on the data store will not be lost.
2. **Strictly serialisable** - there is a total order to all operations completed on the data.
3. **Atomic** - operations happen in full or not at all.
4. **Linearisable** - from a client's point of view, all operations complete fully between their invocation and their response.

Raft provides this consistency through consensus, where the cluster of `etcd` instances all must agree on the database states. This is done through a strong leader, where all transactions go to the leader, and logs for each operation are replicated out to all the replicas from the leader. These leaders are elected whenever there is not currently a leader - which occurs on cluster startup and whenever a leader fails to send a heartbeat in time. Raft is particularly useful due to its effective handling of adding/removing members, which incurs no downtime.

Due to this use of raft in `etcd`, the state of the Kubernetes cluster remains consistent and resilient, once there are multiple `etcd` instances. Specifically, an odd number is recommended so that split votes are less likely.

A final useful feature in `etcd` is that it allows the `watch`ing of resources, so updates can be sent if the data object changes.

1.2.3 `kube-scheduler`

The `kube-scheduler` decides which node newly created pods should run on. The actual deployment of the pod to the selected node is handled by the `kube-api`. The `kube-scheduler` can use a variety of scheduling algorithms for this. The default algorithm for this works in two steps [33]. This can be customised, and entire alternate schedulers can be implemented.

1. **Find acceptable nodes.** In this step the list of nodes in the cluster are whittled down to only the nodes which actually run the pod, given its requirements.

This is based on a number of details - does the node have sufficient resources available, is a port needed by the pod already in use, has the pod requested a specific node, etc.

2. **Select the best node.** In this step, the decision is made of which node is the best to run the pod.

   This selection is made based on information like: the current pod distribution across nodes; resource utilisation; where other pods in the same ReplicaSet are already deployed; and more.

While a single scheduler type cannot be scaled horizontally, different schedulers can be used for different pod types. With this, there may be several schedulers running, on the same or different machines. So while each scheduler type can only have one instance, there may be some resilience offered by the multiple deployed schedulers

1.2.4 `kube-controller-manager`

The `kube-controller-manager` consists of a number of different controllers, which manage different parts of the state of the cluster. These controllers include the following, among many more.

- Deployment controller
- ReplicaSet controller
- Node controller
- Service controller
- Endpoint controller

All of these controllers act in the same way on different resources. Take for example the ReplicaSet controller.

1. `watch` all instances of the resource controlled by that controller.

   For ReplicaSet, that means `watch`ing each ReplicaSet resource. This will be updated if the one of the pods in the ReplicaSet fails (or an extra one is brought up), or if the replica count is changed.

2. Whenever there is an update, check the desired state against the actual state.

   For ReplicaSet, that means checking the desired pod count against the actual pod count.

3. If there is a discrepancy, send an update to the `kube-api` to fix it.

For ReplicaSet, this either happens when there are too many or too few pods in that ReplicaSet running. If there are too few, it posts new pod manifests to `kube-api`. If there are too many, it selects a pod to be deleted from the list of pods, and updates its manifest to terminate the pod.

4. Repeat! Keep `watch`ing the relevant resources.

The other controllers work in the same way. In particular, the Node controller handles keeping the list of nodes in the cluster state in sync with the list of actual machines running the nodes. This includes updating pods to terminate them if the node they are running on fails. With a combination of the Node controller and the `kube-scheduler`, when a node fails, any pods which were running on it will be brought up on another node. In this way, much of the resilience offered by Kubernetes is handled by the `kube-controller-manager`.

1.2.5 DNS

DNS server add-ons are deployed with most Kubernetes clusters, although they are not compulsory. They allow service discovery within the cluster using domain names. The most common options here are `kube-dns` (as discussed in [33]), or, more recently, CoreDNS[5]. These work in a similar way, by `watch`ing Service and Endpoint resources, and updating DNS records to route to these whenever they change.

CoreDNS is able to be replicated for greater resilience, as it makes use of Kubernetes' `ConfigMap` to store and synchronise DNS records, and is therefore stateless [8].

1.3 Research Question

This thesis aims to discover the resilience improvement brought by a multi-cloud Kubernetes deployment, compared to a single cloud deployment. This resilience improvement concerns protection from global cloud provider failures. In particular, the thesis aims to address the resilience discipline of challenge tolerance, using replication across diverse clouds. This is intended to support developers considering multi-cloud deployments, to allow the resilience gain from switching to a multi-cloud deployment to be compared against its costs. For this purpose, an example multi-cloud Kubernetes application will be created. This application should be deployed to allow for testing. The application's resilience will be evaluated using a variety of metrics and compared to that of a variety of single-cloud deployments of the application.

[5] `https://coredns.io/`

1.4 Dissertation Structure

The rest of this dissertation will consist of the following: Chapter 2, State of the Art, will discuss the state of resilience and multi-cloud in the literature and in industry; Chapter 3, Design, will discuss the design of an example resilient multi-cloud system; Chapter 4 will discuss the details of implementing and deploying the multi-cloud system, and problems faced during this time; Chapter 5 will evaluate the multi-cloud system; and finally Chapter 6 will give a summary of the work done, along with reflections about the project, and future work to be done on this project.

Chapter 2

State of the Art

This chapter will discuss the state of the art in the area of multi-cloud applications and resilience, to inform the work performed in this thesis.

2.1 Multi-cloud

Many varieties of multi-cloud solutions have been designed and discussed in the literature. In general, they are based on one of the following:

1. Kubernetes
2. jClouds

However, there are also examples of solutions not based on either of these technologies. In the academic area, Roboconf [40] builds a fully custom solution, with applications deployed on VMs, and architecture defined with a custom domain-specific language (DSL).

2.1.1 Multi-cloud using jClouds

Apache jClouds[1] is a Java toolkit to support developers in developing applications which are portable between multiple cloud providers. It does this by abstracting multiple cloud provider's services of the same type behind the same generic API. For example - AWS' *EC2*, Google's *Compute Engine*, and Microsoft's *Azure Compute* are all available as jclouds' **ComputeServices.**

jClouds is used for underlying interoperability in several tools which aim to provide more complete support for multi-cloud architectures. One of these is *Apache Brooklyn*[2], originally developed by Cloudsoft. Brooklyn allows developers to define their intended architecture in

[1] `https://jclouds.apache.org/`
[2] `https://brooklyn.apache.org/`

structured YAML files. Brooklyn then manages the deployment of this architecture. Brooklyn is compliant with OASIS's CAMP [37] and TOSCA [36] standards. Cloud Application Management for Platforms, or CAMP, is an OASIS standard focusing on the interoperability of cloud providers in situations relating to deployment, management and monitoring of cloud applications. Topology and Orchestration Specification for Cloud Applications, or TOSCA, similarly supports interoperability, by providing a standard to describe the topology of a cloud application in a provider agnostic manner.

However, there is differing levels of support for applications built at different relevant levels of the cloud - Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) [58]. Carrasco et al. [12] build an extension on Apache Brooklyn which allows developers to use Brooklyn to deploy onto multicloud PaaS or IaaS interchangably, and find no performance decrease with this.

Cloudiator [7] also builds on jClouds, independently of Apache Brooklyn. Cloudiator has functionality not available in Brooklyn:

1. Automatic discovery of cloud provider services and configuration options.
2. Prioritising keeping monitoring data close to its source to reduce costs.
3. Support for scaling applications based on complex user-defined metrics and thresholds.
4. An extra layer of abstraction on top of jClouds to handle inconsistencies in the jClouds model.

However, it is important to note Cloudiator's introduction of a *home domain* (as opposed to the cloud domain), which must be managed by the Cloudiator operator. This home domain includes all elements of the control plane - including the scaling engine and the recovery engine. This means that if this centralised control plane goes down, some of the resilience benefit from going multi-cloud is lost.

2.1.2 Multi-cloud Kubernetes

As discussed in the introduction, the standard deployment for Kubernetes is one cluster, with a single (possibly replicated) control plane. However, for larger applications, especially those that require more reliability, multi-cluster deployments can be used. This is what tends to be used in multi-cloud Kubernetes, as it allows logical separation of components as well as reduced intra-cluster latencies.

Multi-cloud Kubernetes is more widely used in industry than other multicloud options, given the existing popularity of Kubernetes as an orchestration tool.

One of these solutions is Google's Anthos[3]. Built as a multi-cluster Kubernetes man-

[3] `https://cloud.google.com/anthos`

agement tool, Anthos supports developers deploying applications across Kubernetes clusters in GCP, AWS, and Azure. This works by building a centralised Kubernetes control plane in GCP, as shown in Figure 2.1. However, this has a similar problem as Cloudiator, discussed in 2.1.1. The presence of a centralised control plane leads to decreased disaster resilience.

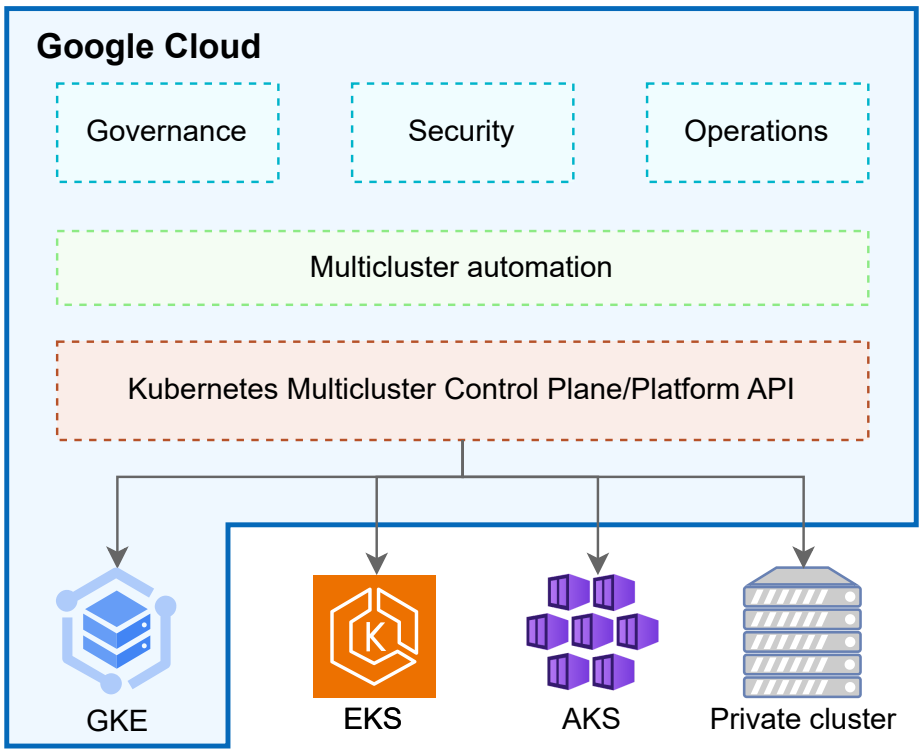


Figure 2.1: Architecture of Google's Anthos in a multicloud setup. All elements of the control plane run in GCP, while the Kubernetes clusters run in a number of locations. Clusters run in Google's Kubernetes Engine (GKE), Amazon's Elastic Kubernetes Service (EKS), and Azure Kubernetes Service (AKS), as well as any other possible locations, like alternate public or private clouds.

Rancher[4] and Platform9[5] have the same problem with a centralised control plane.

A solution to this single point of failure is federating the cloud application. One approach to Kubernetes federation is using a service mesh - an example of which is Istio[6]. Note that many multi-cloud Kubernetes solutions use Istio (for example: (1) Anthos, although this does not create a federated application; (2) Sharma et al.'s proposed architecture [45]). Service meshes allow developers to define how communications should happen between microservices. This could be communication between containers in a Kubernetes cluster, or between clusters. Istio specifically supports load balancing, security, complex routing, automatic metrics and

[4] https://www.rancher.com/
[5] https://platform9.com/
[6] https://istio.io/

logs, and custom policies and configuration [23]. The standard Istio service mesh is set up in *sidecar* mode, where a proxy is installed alongside every pod. These proxies intercept all network traffic to perform the configured actions, allowing any required multi-cloud business logic to be implemented. A multi-cloud example of this is discussed by Houshmand [22], using RedHat's OpenShift Service Mesh, which is based on Istio. Linkerd is the primary alternate to Istio, and sometimes provides better efficiency, while Istio has sometimes been found to be faster [16].

Another approach to federation within Kubernetes was KubeFed (v1 and v2)[7]. However, this project has been abandoned.

2.1.3 Security in Multi-cloud

Considerations for security and privacy in multi-cloud environments must first focus on the complexity inherent in a system build on top of several heterogenous clouds which have little interoperability. This is commented on by Ofori et al. [57] and Afolaranmi et al. [1]. This complexity takes many forms. First, when designing a system, there is no standard way to compare security features and offerings between two different cloud providers - a common problem in multi-cloud systems. This means that consideration of security at an early stage in a project is made more difficult. Given the existing difficulty in considering security at early stages in projects, this means that security is even more likely to take a back seat in design stages, although research is being done in this area, like by Afolaranmi et al. [1]. Much of the security disciplines in the triple-A of auditability, authorisability and authenticity are handled in cloud providers using policies and configurations. However, these policies and configurations tend not to be cloud provider agnostic, leading to likely inconsistent policies and configurations between cloud providers, giving more risk.

Multi-cloud systems generally provide a larger attack surface, which is inherent in their design. While a single cloud deployment has easy points to attack before entry into the cloud provider, multi-cloud deployments also offer the opportunity to attack on inter-cloud communication. This is on top of the risk inherent in using a public cloud (like AWS, GCP or Azure), where users do not have control of the physical hardware, or much of the infrastructure surrounding it. If, for example, AWS has a vulnerability, and a user are deployed only to Azure, they are not at risk. However, if they are in a multi-cloud setup, they might be at risk of vulnerabilities in AWS, Azure, and GCP. Another attack surface not commonly discussed is that of the orchestration tooling. Paladi et al. [39] discuss this - orchestration tooling has full control over most elements in a deployment, and the increased scale and complexity in a multi-cloud system provides greater opportunity for attackers to break into the system.

[7] `https://github.com/kubernetes-retired/kubefed`

Multi-cloud systems provide greater opportunities for bad actors to access systems. This is reflected in the literature, with Byzantine fault tolerance being a common point of discussion in designing secure and resilient systems [5][4]. Byzantine fault tolerance is described as "... the ability of a distributed computer network to function as desired and correctly reach a sufficient consensus despite malicious components (nodes)..." [41] - how the system handles some element lying. Similarly, work has been done on protecting against man in the middle attacks in multi-cloud systems, given the increased risk. [43].

While multi-cloud deployments do increase security risk to some degree, they also provide some degree of protection. This is through the protection against correlated faults through diversity - an example of a correlated fault could be a cloud provider having a vulnerability allowing bad actors to shut down an application. In a multi-cloud deployment, not all resources would have this risk given the increased diversity of deployment. This leads to partial failure rather than full failure, and is the other side to the earlier discussed increased risk when using multiple clouds. Multi-cloud solutions can also provide added support for data governance related to privacy focused legislation which specifies where certain data may be held, as it gives application developers more options.

2.2 Reliability and Resilience

Reliability and resilience have been defined in many ways in the literature. For this thesis, the approach follows that of Welsh et al. [54], using the definitions provided by Sterbenz et al. when building the Resilinets framework [49]. A summary of the relevant definitions, and some of their relationships is given here, and a summary of the relationships is in Figure 2.2.

Challenge tolerance These concepts relate to the design of systems which stay working when faced with challenges.

- **Survivability** The ability of a system to continue serving when facing both correlated failures (eg cyber attacks) and uncorrelated failures (eg single machine failure). Uncorrelated failures are generally addressed by *redundancy*, with this concept being referred to as **fault tolerance**. Correlated failures use *diversity* as a solution.
- **Disruption tolerance** The ability to handle disruption in communication between system components.
- **Traffic tolerance** The ability to handle unpredictable load on a system.

Trustworthiness These concepts describe "the assurance that a system will perform as expected" [49], and are directly measurable.

- **Dependability** The measurement of how much the system can be relied on. Useful

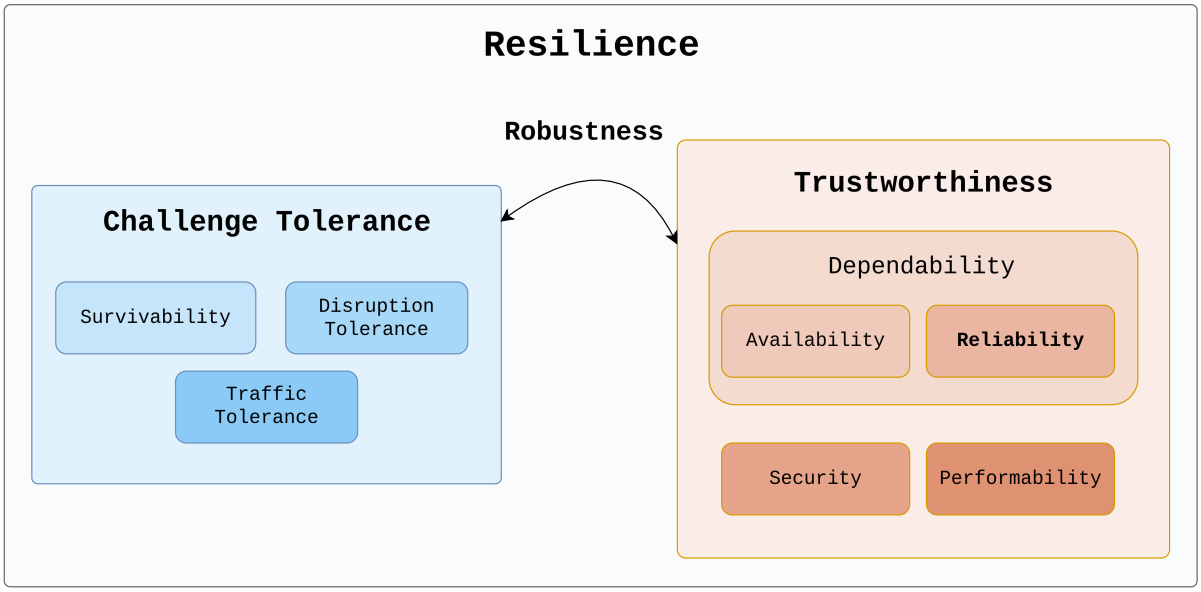


Figure 2.2: Classification of elements of resilience according to Sterbenz et al. [49].

metrics for dependability are mean time to failure (MTTF), mean time before repair (MTBR), and mean time between failures (MTBF).

Availability Readiness for usage, measured by the probability the system will be usable at any one time.

Reliability Continuity of service, measured by the probability the system *remains* operable for a period of time.

Security The protection of a system from unauthorised access or changes.

Performability The performance of a service with regards to its service specification - measured by metrics like response time or queries per second (QPS).

Robustness The relationship between trustworthiness and challenge tolerance - the quantifiable trustworthiness of a system in the face of challenges.

With these definitions, it is possible to classify work completed in the literature. Welsh et al. [54] perform a survey of resilience in the Cloud, which provides a good overview of methods used at different levels of cloud infrastrusture. Most relevant to this thesis are resilience at the instance and management level. At instance level, resilience mainly comes down to survivability via replication and diversity, while error detection and restarts allows for self healing. An example of diversity for resilience is given by Yu et al [59]. At the management level (which includes orchestration via tools like Kubernetes), more approaches are available. These include: multicloud and careful host/node selection to improve diversity; self-organisation and self-management to ensure no dependencies between components; observability and monitoring to improve time to notice errors; having brownouts instead of blackouts where possible - where service degrades but does not fail when some

part of the system fails. Note that the authors here comment on the need for management systems to not introduce a single point of failure (as discussed in 2.1.2).

Jin et al. [28] demonstrate a survivability approach specifically for stateful Kubernetes applications, focusing on disaster recovery specifically. This assumes disasters will take out Kubernetes clusters in a specific availability zone in the cloud, which need to be recreated in another zone. This makes use of a disaster recovery framework (Ramen[8]) to replicate application data to a recovery cluster in another availability zone, which is started up when the original cluster goes down. However, it is important to note that this paper assumes the persistent data for the application is stored in a managed database, which persists the same data between the original and recovery cluster.

2.2.1 Measuring & Testing Resilience

Resilience Evaluation

Welsh et al. [54] introduce some options for how cloud resilience can be evaluated.

1. **Binary** - does a resiliency feature exist in the cloud or not.
2. **State based** - a system is resilient if it has more operational states than failure states.
3. **Performance oriented** - the traditional resiliency metrics like MTBF.
4. **Graph based** - view the system as a graph, and evaluate based on numbers and layouts of nodes and links (and how the system is affected when nodes and links are removed).

Graph based metrics useful in modelling resilience in the cloud are often not designed for the cloud. Take for example the work done by Alenazi et al. [2] - this surveys a large set of graph robustness metrics, but this focuses on communication networks. However, a selection of these metrics are used by Welsh et al. [55], applying them to Fog and Edge systems, which have a strong relationship with the cloud. This gives a wider variety of options when evaluating resilience, as graph robustness does not need to be specific to the cloud.

Rosenkrantz et al. [44] provide an interesting set of graph resilience metrics. While they are not cloud specific, they are particularly useful for the cloud (and especially microservice architectures), as they are *service-oriented* metrics. The metrics allow for modelling of services and their dependencies on each other, providing a good resilience metric specifically for when elements of a cloud system are replicated, as is common for resilience.

[8] `https://github.com/RamenDR/ramen/`

Chaos Engineering

In general in software engineering, testing can be split into three different levels. A summary of tools available at each level is given in Figure 2.3.

1. **Unit testing** - the testing of isolated components. Tools for this vary by programming language and use case.

   Taking for example Java, simple unit tests can be performed using `JUnit`.[9] More complex components may required the use of frameworks like `Mockito`,[10] which allows "mocking" components which are depended on but are not being tested in a unit test.

   Unit testing tools can be used for Test Driven Development (TDD), where the developer defines the intended behaviour of a component through tests before developing the component itself. In this case, the component can be described as finished once it passes all the tests. More commonly, unit tests are created after code has been written, and are used to debug the component, ensuring it works as expected.

2. **Integration testing** - testing to make sure the components of the application work together as expected.

   This can test just two components together, or a subset of the components, or the whole system (sometimes called **system testing**). However, this differs from the following type of test in that dependencies outside of the system are mocked out rather than being tested.

   Given the large range of integration tests, a large range of tools can be used. For integration tests where just two components are being tested, frameworks like `JUnit` can continue to be used. At a higher level, when testing a full system (excluding dependencies), frameworks like `Cucumber`[11] can be used to test the behaviour of the full system. `Cucumber` is described as a Behaviour Driven Development (BDD) tool, an alternative to the TDD approach discussed in unit testing.

3. **End to End (E2E) testing** - testing the full system as the user will be using it.

   Tools for this include `bugbug`[12] and Selenium based tools like `Katalon Studio`.[13]

   Load/performance tests often fit into E2E testing as well, by simulating a massive number of users using an application and tracking its performance. Tools for this

[9] `https://junit.org/`
[10] `https://site.mockito.org/`
[11] `https://cucumber.io/`
[12] `https://bugbug.io/`
[13] `https://katalon.com/`

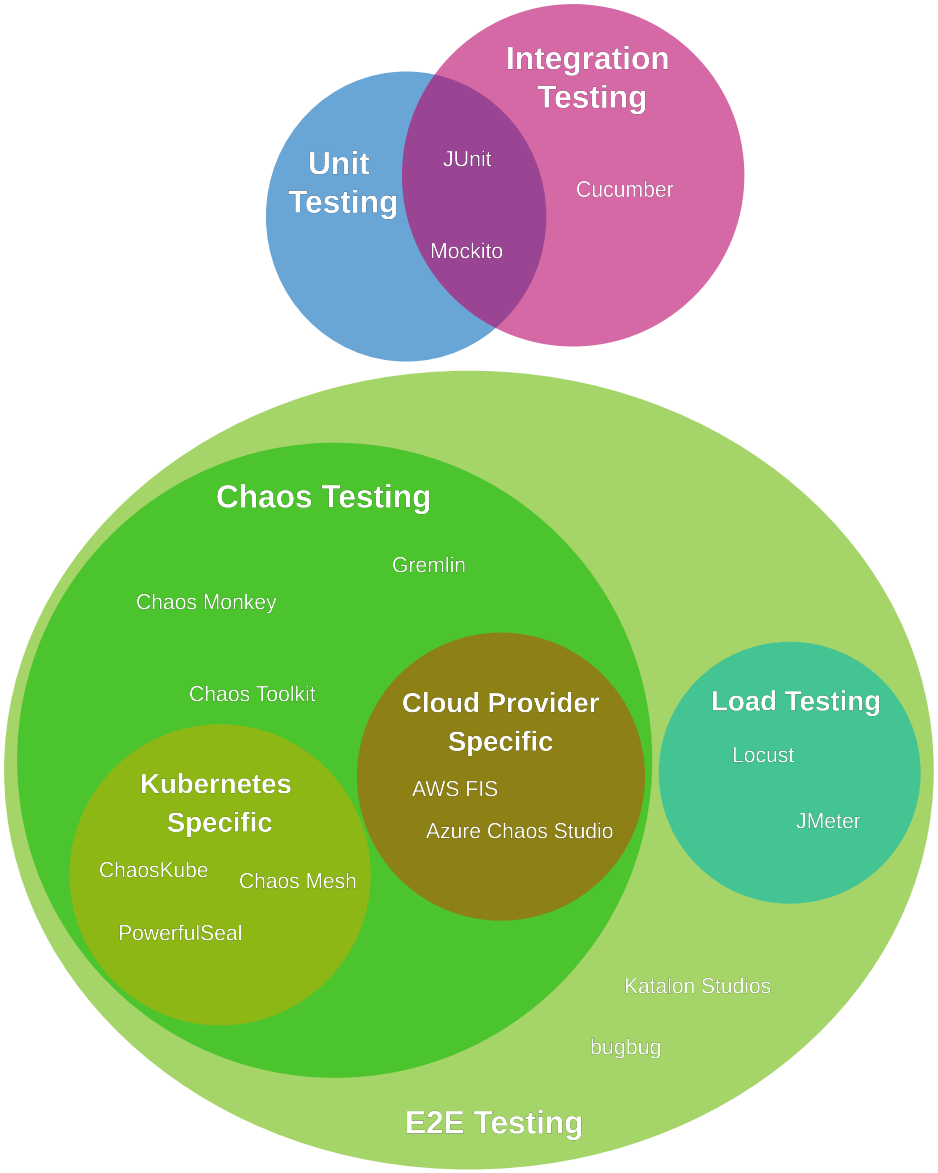


Figure 2.3: Software testing disciplines and example tools.

include `JMeter`[14] and `Locust`.[15]

Chaos tests also fall under E2E tests.

Chaos engineering can be described as breaking elements of a system to see how it responds, or more specifically as "... the discipline of experimenting on a system in order to build confidence in the system's capability to withstand turbulent conditions in production."[13] While the concepts of chaos in a system go back to chaos theory in physics, their first entry into the cloud was by Netflix, with the creation of Chaos Monkey (and the extended Simian

[14] `https://jmeter.apache.org/`
[15] `https://locust.io/`

Army) to test their system after migrating to AWS [10]. Alternate tools for chaos engineering include Chaos Toolkit, Gremlin, some cloud specific tools like Amazon's Fault Injection Service[16] and some Kubernetes specific tools in ChaosKube, PowerfulSeal[17] and Chaos Mesh.[18]These act as E2E tests by testing how the user would experience the system if elements of it were in failure.

Chaos engineering allows the scientific method to be brought into software testing, extending past unit and integration tests into proper tests of the system in practice. Bergstrom [10] describes the steps of chaos engineering as

1. **Validate the system** - the system at its baseline must be understood before undertaking any experiments. As well as this, the system must have sufficient logging, visibility, and documentation.
2. **Form hypothesis** - a hypothesis must be formed of the ways in which the system may fail.
3. **Plan experiment** - experiments to test the hypothesis must be planned out. This requires knowing the expected scope and impact of the experiment, labeling the criteria to trigger a halt to the experiment, as well as the duration and support requirements of the experiment.
4. **Run experiment** - the experiments must be run, while collecting system data.
5. **Monitor and repeat** - collection of data, analysis and conclusions should be drawn, and the experiment should be repeated.

Chaos engineering can test multiple different levels of a system, as can be seen in the lineup of tools available in Netflix's Simian Army [26]. While Chaos Monkey simulates a single server failure, Latency Monkey introduces artificial delays, and Chaos Gorilla simulates a full zone failure. This logic can be transferred to other models, with Raj et al. [42] discussing failure at container and VM levels.

Chaos engineering has been used to test Kubernetes extensively, as well as the combination of Istio and Kubernetes, as shown Raj et al. [42] and Singh et al. [48]. Both discuss the resilience improvement brought by Istio.

[16] `https://aws.amazon.com/fis/`
[17] `https://powerfulseal.github.io/powerfulseal/`
[18] `https://chaos-mesh.org/`

Chapter 3

Design

This section discusses the design of the resilient multi-cloud system used for evaluating resilience. A federated Kubernetes system was chosen as the multicloud solution due to the lack of a centralised control plane, and Kubernetes in-built self-healing functionality [34]. Istio was chosen as the service mesh due to its popularity. Istio's sample application `Bookinfo`[1] was used for this project due to its existing Istio integration. This is discussed in Section 3.1. Istio's architecture, and how it interacts with this sample application is discussed in Section 3.2. Some modifications to the sample application were required to test more resiliency, as is discussed in Section 3.3. This sample application was originally designed for a single cluster deployment, so modifications required to make it multi-cluster are discussed in Section 3.4, while the final resilient design is discussed in Section 3.5. Note that all code for this design can be found on Github, based on a fork from the Istio Bookinfo sample.

3.1 Sample Application

Istio's `Bookinfo` application was chosen as an appropriate example of a microservice cloud application. The architecture of the application as provided by Istio is shown in Figure 3.1. A variety of languages and frameworks are used in this application to show the functionality of a microservice architecture, where each component can be implemented in the way most suited to it.

The components of the application work as follows:

Product Page This microservice is implemented in Python. It is the point of entry to the `Bookinfo` service, and defines the frontend of the service. It also controls the API to allow interactions with the other services. It calls the `Reviews` and `Details` microservices to populate the data required.

[1] `https://istio.io/latest/docs/examples/bookinfo/`

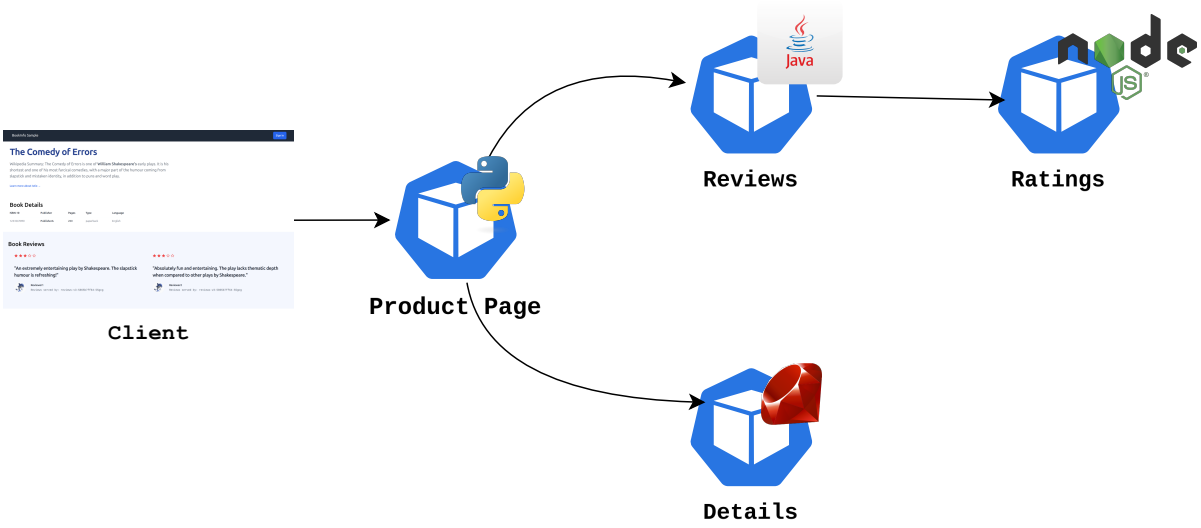


Figure 3.1: Initial architecture of the Istio `Bookinfo` application, showing 4 microservices. The `Product Page` service runs on Python, the `Details` microservice runs on Ruby, the `Reviews` microservice runs on Java, and the `Ratings` microservice runs on NodeJS.

Details This microservice is implemented in Ruby. This service provides book information. There is an option to contact an external book information provider, but this is not the default behaviour of the sample application, and was not tested during this project. Instead, the book information is provided statically from memory. Note that this means the `Bookinfo` application only supports providing information about one book.

Reviews This microservice is implemented in Java. It provides review details for the book - specifically two hard-coded reviews, with no way to add more reviews. It has a similar configuration to the `Details` microservice, where only one book is possible. The `Reviews` microservice also calls the `Ratings` microservice to populate ratings data, which it returns to the `Product Page` with the hardcoded reviews.

The `Reviews` microservice has several versions in the original Istio example to demonstrate Istio's capability for load balancing and version management - v1 does not call `Ratings`, v2 shows the ratings as black stars, and v3 shows the ratings as red stars. However, for the purposes of this project, all versions except v3 are ignored.

Ratings This microservice is implemented in NodeJS. It provides ratings for the book out of 5 stars. By default, these are hard-coded in memory. However, it is possible to configure this to read from a MongoDB or MySQL database, in order to test the system working with a real database.

With these details, it is obvious that this is not an accurate representation of a real cloud application. However, once the database option is used, it fulfills all the requirements for this particular project. There is no concern here about the usability of the test application beyond performance.

3.2 How does Istio Work?

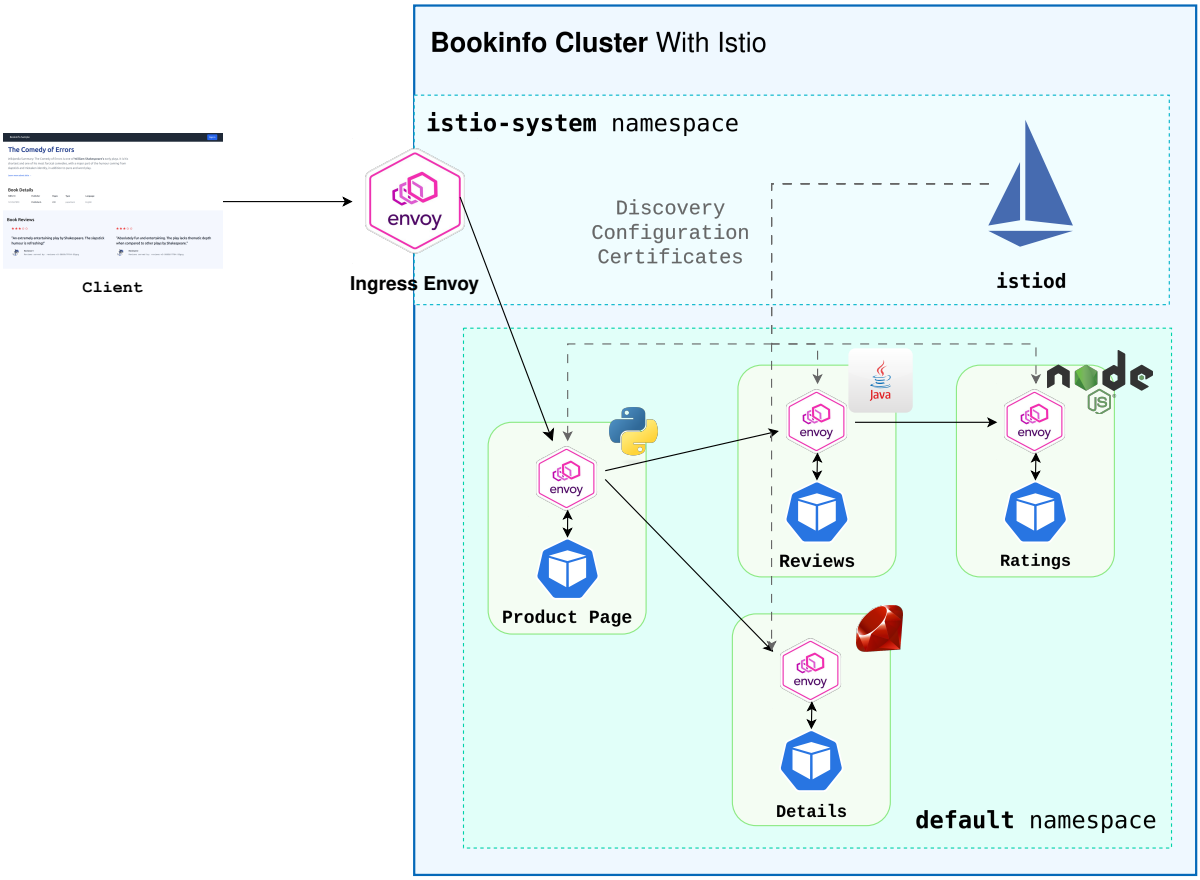


Figure 3.2: Istio `Bookinfo` application with Istio resources shown. Alongside every container is an Envoy proxy sidecar. Any communication to or from that container goes through the sidecar, whose behaviour is configured by `istiod` in the Istio control plane. Ingress is managed by another Envoy container, which sends requests through from the client to the `Product Page`. These resources are present in all versions of the design, but may not be shown for simplicity.

Istio is a service mesh, meaning it manages all communications between the services in the application - regardless of what cluster they are in. It is composed generally of a data plane and a control plane [24].

The data plane performs the actual handling of traffic between services. There are two possible ways for the dataplane to be deployed - in **sidecar** or **ambient** node. Sidecar mode is the default and original mode, which deploys an Envoy[2] proxy alongside every pod in the cluster. Ambient mode requires a single Layer 4 (TCP/UDP level) proxy per Kubernetes node - and an optional Envoy proxy per namespace if Layer 7 (HTTP/HTTPS/... level) features are required. Ambient mode was introduced to lower resource usage. However, it does not support multi-cluster deployments, and is less well tested. Due to this, sidecar mode is used for this project. The Envoy sidecars enforce load balancing and traffic control/routing rules.

[2] `https://www.envoyproxy.io/`

The control plane manages and configures the proxies. The control plane is deployed in a separate Kubernetes namespace, `istio-system`, which also holds any non-sidecar Istio resources. The primary element of the control plane is `istiod`, which provides service discovery, configures all proxies, and handles certificates.

The `Bookinfo` cluster can be seen with Istio resources in Figure 3.2.

3.3 Modifications to Sample Application

As discussed, `Bookinfo` needed some changes to make it appropriate for this project.

First was the addition of a MongoDB database, chosen as it was the default option when using a database with the `Ratings` microservice. MongoDB's existing replication[3] and existing instructions for deploying to multi-cluster Kubernetes deployment[4] were also useful. Eisenberg [15] provides instructions for how to configure the `Bookinfo` application to work with an external MongoDB database, by adding an Istio `ServiceEntry` with details of how to access the MongoDB instance. This MongoDB instance was set up in another Kubernetes cluster.

Once the MongoDB instance and cluster was successfully connected to the `Bookinfo` application, it was noticed that it was not possible to write to the database with the basic sample application, which only supported writes when the database was in memory. Modifications were made to the code of the `Ratings` microservice to allow writes to go through to the database. As the `Product Page` only has space for and accepts two ratings/reviews entries, the `Ratings` microservice is also modified to return the two most recent reviews written to the database, so that the writes have a visible effect to the client. Some final changes were also required to the `Product Page` to create an API endpoint to write ratings to the database. The upgraded architecture/deployment at this stage can be seen in Figure 3.3.

3.4 Multi-cluster

The application must be made multi-cluster so that it can be deployed in a multi-cloud manner. For this, the clusters must be configured with Istio to be able to access each other, specifically in Multi-Primary mode.[5] Multi-Primary mode has an Istio control plane on all clusters in the mesh. This includes the following steps.

[3] `https://www.mongodb.com/docs/manual/replication/`

[4] `https://www.mongodb.com/docs/kubernetes-operator/current/multi-cluster-overview/`

[5] `https://istio.io/latest/docs/setup/install/multicluster/multi-primary_multi-network/`

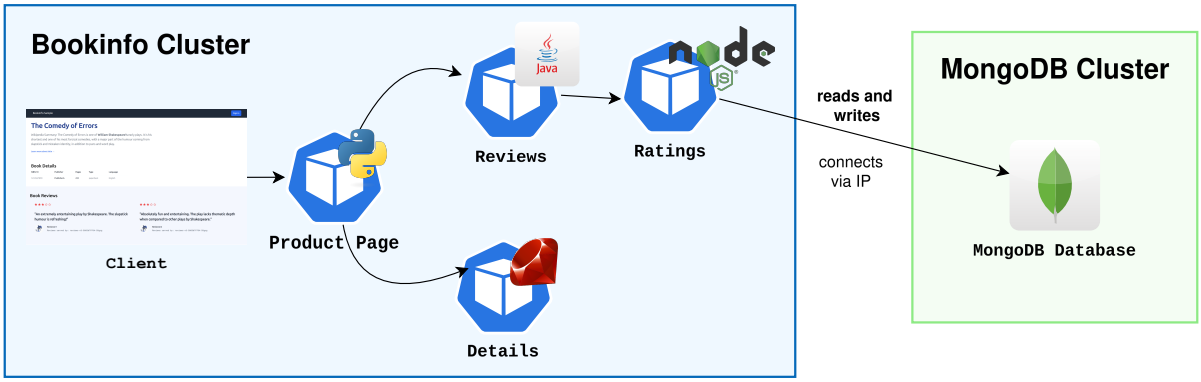


Figure 3.3: Architecture of `Bookinfo` application with database functionalities enabled. Note that the `Bookinfo` application is in a different cluster to the database, and for now connections between clusters are managed via IP addresses (or DNS lookups to IP addresses).

1. **Istio installation.** Istio needs to be installed in all of the clusters while giving values for: `meshID`, same for all clusters; `multiCluster.clusterName`, different for all clusters; and `network`, different for all clusters.
2. **Cluster certificates.** All clusters require the generation and installation of CA certificates.
3. **East-west gateway.** This gateway allows communication to pass between the clusters, still managed by Istio. This gateway is also configured with the setting `AUTO_PASSTHROUGH` for TLS, meaning that it allows all inter-cluster messages to pass through.
4. **Endpoint discovery.** All clusters have remote secrets created with the details of where to find the `kube-api` of the other servers.

With multi-cluster Istio configured, it is now possible to access any service in the mesh using its fully qualified domain name (FQDN), following Kubernetes standards [32]. This means that in any cluster, the `details` service could be accessed by the hostname shown in Listing 3.1.

```
details.default.svc.cluster.local
```

Listing 3.1: FQDN for the `details` service in the Istio service mesh. `details` is given by the service name, `default` is given by the Kubernetes namespace the service is deployed in, and the remainder is common between all services.

When there is replication of the same service across multiple clusters, the load is balanced equally between the replicas without any configuration - for example all the `reviews` instances across `Bookinfo` clusters 1 through N in Figure 3.4. However, this does not work if the service is not already declared in the cluster. Take for example the `reviews` service in `Bookinfo` cluster 1 trying to contact the service `mongodb-1` in Database cluster 1. Because

the service has not been declared in `Bookinfo` cluster 1, the DNS lookup fails. This fact is not documented, but it can be fixed by registering the `mongodb-1` service in `Bookinfo` cluster 1, routing to pods with the label `mongodb-1`. Once this is complete, Istio is able to manage the routing.

Splitting load between the `Bookinfo` clusters happens in two ways. First, Istio manages load balancing between all of the services inside the mesh. Second, DNS manages load balancing between the different `productpages` as ingress to the application. This is done by defining multiple A records for the hostname, referencing the IP addresses for each ingress. Then, DNS splits the load equally between these records, using round robin load balancing [14].

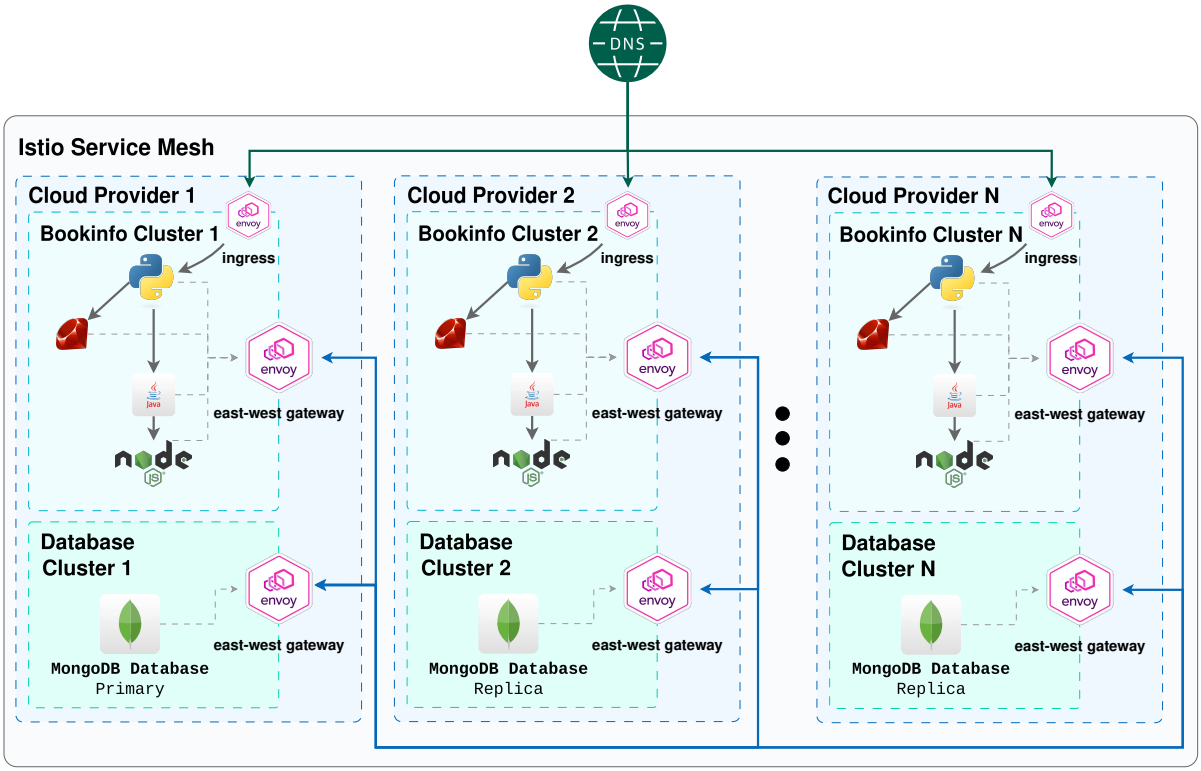


Figure 3.4: Architecture of `Bookinfo` application distributed across $2N$ clusters in N clouds. In each `Bookinfo` cluster there is a `productpage` microservice in Python, a `details` microservice in Ruby, a `reviews` microservice in Java, and a `Ratings` microservice in NodeJS. When a client makes a request, DNS balances load between the N ingress options using round robin load balancing. Load balancing between all `productpages`, `details`, etc is managed by Istio and performed through the east-west gateways. These are also used for communication between `Bookinfo` clusters and database clusters.

```
rs.initiate(
    {
        _id: "rs0",
        members: [
            { _id: 0, host: "mongodb1.mongodb.svc.cluster.local:27017"
               },
            { _id: 0, host: "mongodb2.mongodb.svc.cluster.local:27017"
               },
            // ...
            { _id: 0, host: "mongodbn.mongodb.svc.cluster.local:27017"
               }
        ]
    }
)
```

Listing 3.2: Command to run to configure a MongoDB replica set. `rs0` is the replica set name/ID. Each host name listed in the `members` is the FQDN of the pod for each database instance. Note that the FQDNs have `mongodb` instead of `default` as namespaces.

3.4.1 MongoDB Replica Set

In this multicluster setup, there is now database replication. This is managed by MongoDB's replica set.[6] The MongoDB replica set allows a group of MongoDB instances to share the same data. It replicates via a primary/replica model, with eventual consistency. The replica set manages leader elections if the primary fails. A replica set is created by running the command in Listing 3.2 in one of the database instances.

Details of how to connect to the database are defined in a `connection string`, which contains the hostnames for every member of the replica set. The client uses this string to connect to the replica set, and the MongoDB client handles routing to primary/secondaries as required. An example connection string is given in Listing 3.3.

```
mongodb://mongodb1.mongodb.svc.cluster.local:27017,
    mongodb2.mongodb.svc.cluster.local:27017,
    mongodb3.mongodb.svc.cluster.local:27017
    /test?authSource=test&replicaSet=rs0
```

Listing 3.3: Connection string for example MongoDB replica set `rs0`, which has 3 replicas. It connects to database `test`, authorising against user details in database `test`. The replica set consists of instances at the list of FQDNs listed, all of which are running on port 27017.

The primary instance receives all writes, while secondaries can receive reads, depending on configuration. Control over which replica reads are sent to is configured in the client code by setting the read preference.[7] This can be set to any of:

[6] https://www.mongodb.com/docs/manual/replication/

[7] https://www.mongodb.com/docs/manual/core/read-preference/

1. **Primary** - default, where all operations read from the primary.
2. **Primary preferred** - read from primary when available. If the primary is unavailable, read from a secondary member.
3. **Secondary** - all operations read from one of the secondaries.
4. **Secondary preferred** - read from one of the secondaries when available. If the replica set only has one primary member, and no other functioning members, read from the primary instead.
5. **Nearest** - read from a random replica set member which appears to be closest based on latencies.

This design uses the **nearest** read preference to reduce latencies.

Note that the FQDNs for MongoDB services are all unique: `mongodb1`, `mongodb2`, etc; as opposed to all `details` pods being behind the single `details` service. This is required so that the replica set members have unique identifiers so they can contact specific other replica set members.

3.5 Resilient Design

This section will discuss the overall resilient design, summarising the work performed in this chapter. A diagram of the full design is shown in Figure 3.4.

1. **Kubernetes self-healing.** The use of Kubernetes for the design means that when pods fail, they will be automatically restarted.
2. **Application partial failure.** The sample application is designed for partial failure. For example, if the ratings service is not available, the product page will still display to users, only missing ratings data.
3. **Istio communication.** The use of Istio for this application provides secure intercluster communication, without concerns about port/IP management, as well as message retries.
4. **Service replication.** The basic sample application discussed in Section 3.1 is replicated across multiple clusters in multiple clouds. If a service's pod in one cluster fails, load will instead go to its equivalent in other cluster. If a full cluster goes down, all services are available in another cluster.
5. **Database replication.** Application data has been moved into an appropriate database, which is replicated across multiple clusters. This means if one instance fails, others will be able to continue to serve requests. MongoDB also handles leader elections in case

the primary database instance fails.

6. **DNS Round Robin.** DNS round robin is used in this system to ensure there are multiple access points into the Istio mesh, meaning if one fails, the system is still entirely usable.

Chapter 4

Implementation

This chapter discusses the implementation and deployment of the multi-cloud system designed in Chapter 3. The design assumes at least 6 Kubernetes clusters distributed across at least 3 clouds, but practically deploying this is slightly more complicated. In Section 4.1, the available resources for this project are discussed, which dictates how the deployment must be performed. In Section 4.2, the details of how the deployment is performed are discussed. Unfortunately, this deployment did not go as planned. Problems with this deployment, and the attempted fixes are discussed in Section 4.3.

4.1 Available Resources

Ideally, this project would be deployed with two Kubernetes clusters in each of AWS, Azure, and GCP, with a cluster for the `Bookinfo` microservices and a database cluster in each cloud provider. However, this was not possible, due to a combination of lack of funding and time constraints. Surrounding the cost of cloud resources, a summary is given in Table 4.1. In the best case scenario, the deployment would likely end up costing a minimum of $34 per month. However, the best case scenario is unlikely, and it is likely that other costs would appear, as has happened with other students.

Dr. Weber, the supervisor of this project has put a significant amount of work into getting access to cloud resources for projects like this. In previous years, he has tried to work with AWS, and this year tried to discuss getting resources in GCP and Oracle's cloud.

[1] `https://azure.microsoft.com/en-us/free/students`

[2] `https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/`

[3] `https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/#bs-series`

[4] `https://aws.amazon.com/ec2/instance-types/t2/`

[5] `https://cloud.google.com/compute/vm-instance-pricing?hl=en#e2_sharedcore_machine_types`

	Azure	AWS	GCP
Credits?	$100 with student email.[1]	Significant applications required.	$300 with new account.
Free Tier Compute	`B2ats v2`, 1GB RAM	`t2.micro`, 1GiB RAM	`e2.micro`, 1GiB RAM
Note that Kubernetes requires an absolute minimum of 2GB RAM.[2]			
Free Tier Kubernetes	No Kubernetes management cost.	No free tier.	No Kubernetes management cost for 90 days for 1 cluster.
Cost for Compute	2 `B2pls v2`, 4GB RAM, min $50/month.[3]	2 `t2.small`, 4 GiB RAM, min $34/month.[4]	2 `e2-medium`, 4GiB, min $50/month.[5]
All of these compute prices exclude any hard disk usage.			
Cost for Kubernetes	2 clusters, free tier running on `B2pls v2`, minimum $60/month.	2 clusters, $146/month excluding underlying resources.	2 clusters, 1 cluster at $10 per month (`e2-medium`) using free tier, second cluster at $83 per month.

Table 4.1: Cost details for Azure, AWS, and GCP. Details of any credits available and what is provided at a free tier. Estimations of cost for running 2 clusters in each provider per month are also provided.

Unfortunately, no resources were able to be acquired.

While running on a laptop was considered, this was decided against. This would require large amounts of resources, not leaving any resources available for testing, evaluation, or other work on the laptop.

Instead, during the course of this project, a physical server was acquired for personal use, which could be used for this project. This device is a Dell PowerEdge, with 2 Intel® Xeon® CPU E5-2695 v4 18 core CPUs, and 64GB RAM, running Ubuntu Server 24.04.2 LTS. This was used to try to simulate multiple clouds for testing.

4.2 Deployment

Several infrastructure changes/decisions were required to handle deploying to a single bare-metal server instead of a group of cloud-managed Kubernetes clusters. First, allowing multiple clusters to be deployed on a single machine required specific tooling, which is discussed in Section 4.2.1. Kubernetes is designed to handle being deployed in public clouds which are intended to provide their own load balancers. When deploying on bare metal, MetalLB can be used as a load balancer, which is discussed in Section 4.2.2. The original plan of using

DNS round robin to distribute load between the replicated ingress points in the Istio mesh was not possible, as all ingress points were deployed on the same machine. Instead, Caddy was used to route connections to each ingress point based on URL details, which is discussed in Section 4.2.3. The final deployment is discussed in Section 4.2.4.

4.2.1 Kind

Kind (Kubernetes in Docker)[6] is a tool which allows running local Kubernetes clusters, intended for testing. It represents each Kubernetes node with a Docker container. Most importantly, it allows for multi-cluster development on a single machine, which is more complex when using minikube.[7] It is also far simpler to use than kubeadm,[8] the default tool for production Kubernetes clusters.

By default, kind clusters are created with only one node - the control plane node. It is possible to deploy applications to this node. This is the approach taken in this deployment, to reduce overhead. Any configuration to the cluster is passed in through the kind config, an example of which is shown in Listing 4.1. With four clusters deployed on the server with similar configurations to that shown in Listing 4.1, the deployment looks like that shown in Figure 4.1.

4.2.2 MetalLB

MetalLB[9] implements load balancers for Kubernetes cluster running on bare metal machines. It requires a set of available IPv4 addresses which it is allowed to assign to the cluster. When working with kind, these addresses can be found by running the command shown in Listing 4.2. Then, each cluster can have MetalLB installed, with some subset of these addresses available for its use. The details of this installation on the server, using these subsets of addresses, is shown in Figure 4.2.

The full process of creating kind clusters, installing MetalLB and certificates, and finally installing Istio is complex, and has many places it can and does go wrong. However, it was made possible using Yadav's Github repository [56], which was modified slightly for this project, with the modified version available on GitHub. This project follows the steps described here and in Section 3.4 to create a set of kind clusters connected in an Istio service mesh. The details of this deployment with MetalLB and Istio is shown in Figure 4.3.

[6] `https://kind.sigs.k8s.io/`
[7] `https://minikube.sigs.k8s.io/docs/`
[8] `https://kubernetes.io/docs/reference/setup-tools/kubeadm/`
[9] `https://metallb.io/`

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
networking:
  apiServerAddress: "0.0.0.0"
  apiServerPort: 6451
nodes:
  - role: control-plane
    kubeadmConfigPatches:
      - |
        apiVersion: kubeadm.k8s.io/v1beta3
        kind: ClusterConfiguration
        apiServer:
          certSANs:
            - "bookinfo.clairegregg.com"
            - "127.0.0.1"
            - "0.0.0.0"
            - "localhost"
```

Listing 4.1: Kind configuration for a cluster on startup. First, note that the kube-api is set to listen on port 6451. Setting the port is required to ensure all of the clusters set up on the same machine have different kube-api ports, allowing them to run and be contacted without failure. The second important element is the kubeadm patch. The goal of this is to set the Subject Alternative Names (SANs) that the certificate for the kube-api should accept. Specifically, this ensures that it can be accessed by URL from remote machines, as well as locally.

```
docker network inspect -f '{{$map := index .IPAM.Config 0}}{{
   index $map "Subnet"}}' kind
```

Listing 4.2: Docker command to give the IP address of the docker subnet where MetalLB can assign IP addresses. An example output is `172.19.0.0/16`.

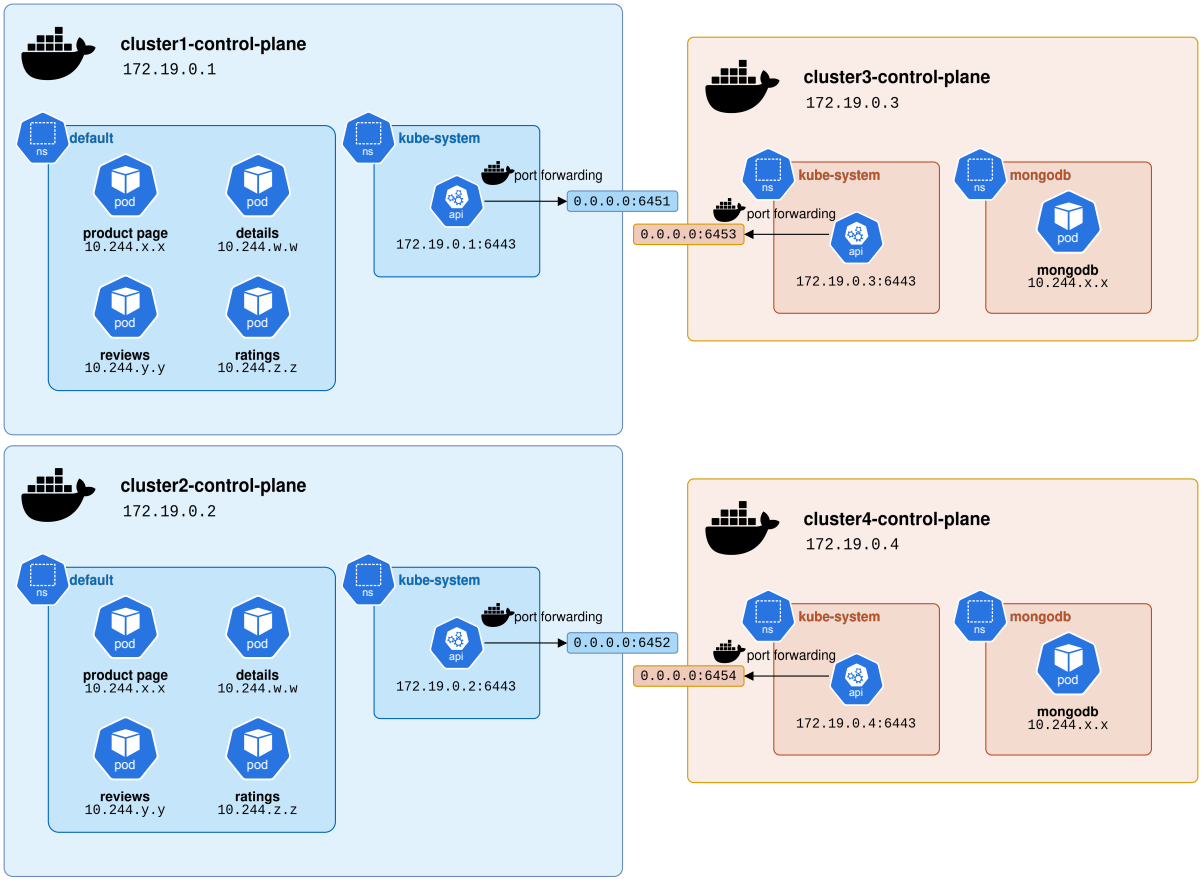


Figure 4.1: Deployment of 4 kind clusters om the server. Pods, namespaces, and example IP addresses are shown (based on real addresses). Each cluster has only one node, the control plane node, which is created automatically by kind, and deployed in a docker container. The IP address of the docker container is assigned by docker. Inside the cluster, pods have IP addresses starting with `10.244.`, with the remaining values randomly selected. Any components which may be accessed outside the cluster have the same IP address as the docker container. The `kube-api`'s ports are forwarded to `0.0.0.0` so they can be accessed on `localhost` and from machines outside the server. The external port (6451, 6452, etc.) is selected in the kind configuration.

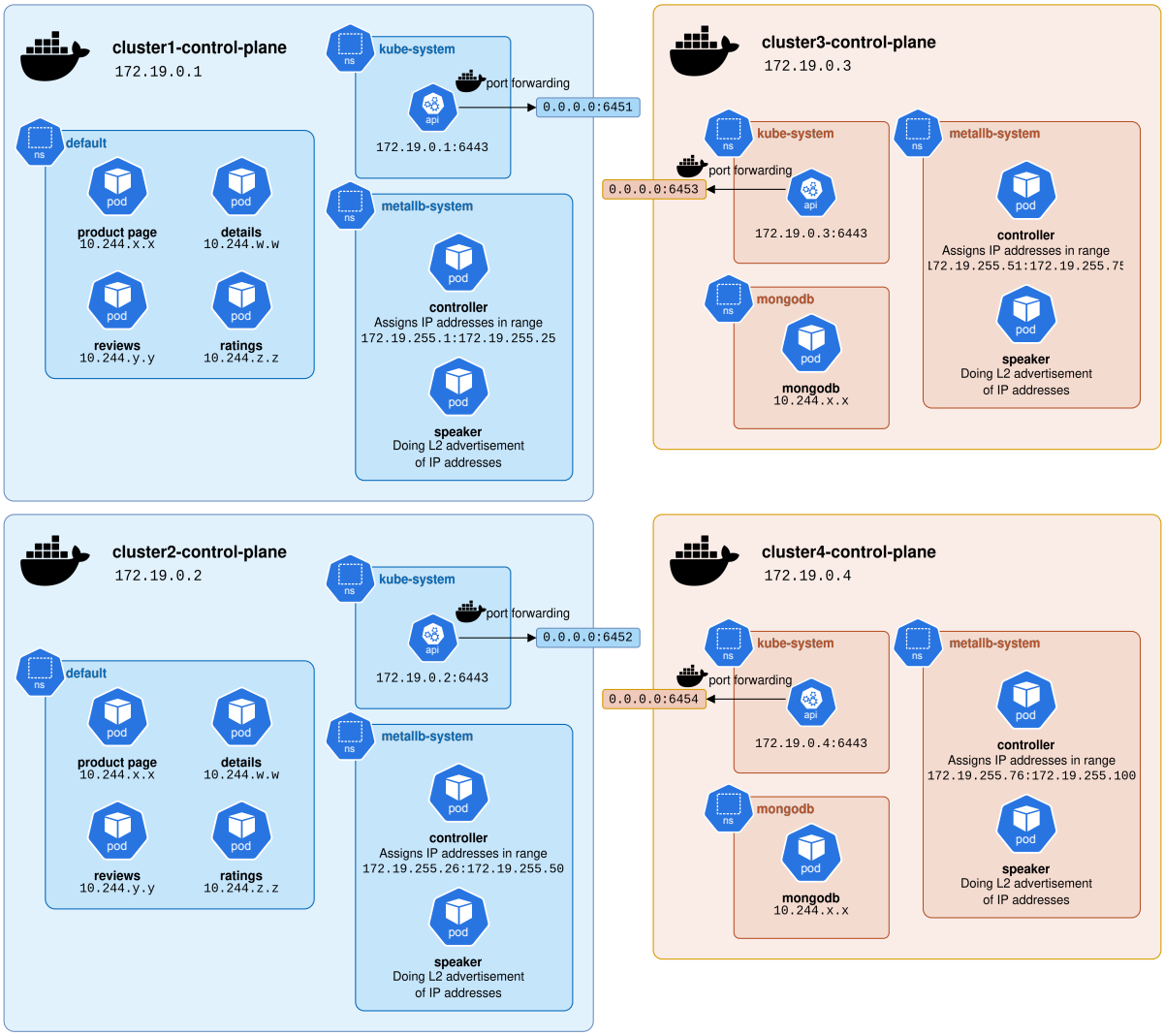


Figure 4.2: Continuation from Figure 4.1 with MetalLB installed. This adds two new components. The `controller` handles assigning IP addresses within the assigned subset of addresses. Whenever a component in the cluster wants an external IP, it will request one from the controller. The `speaker` advertises the utilised IP addresses.

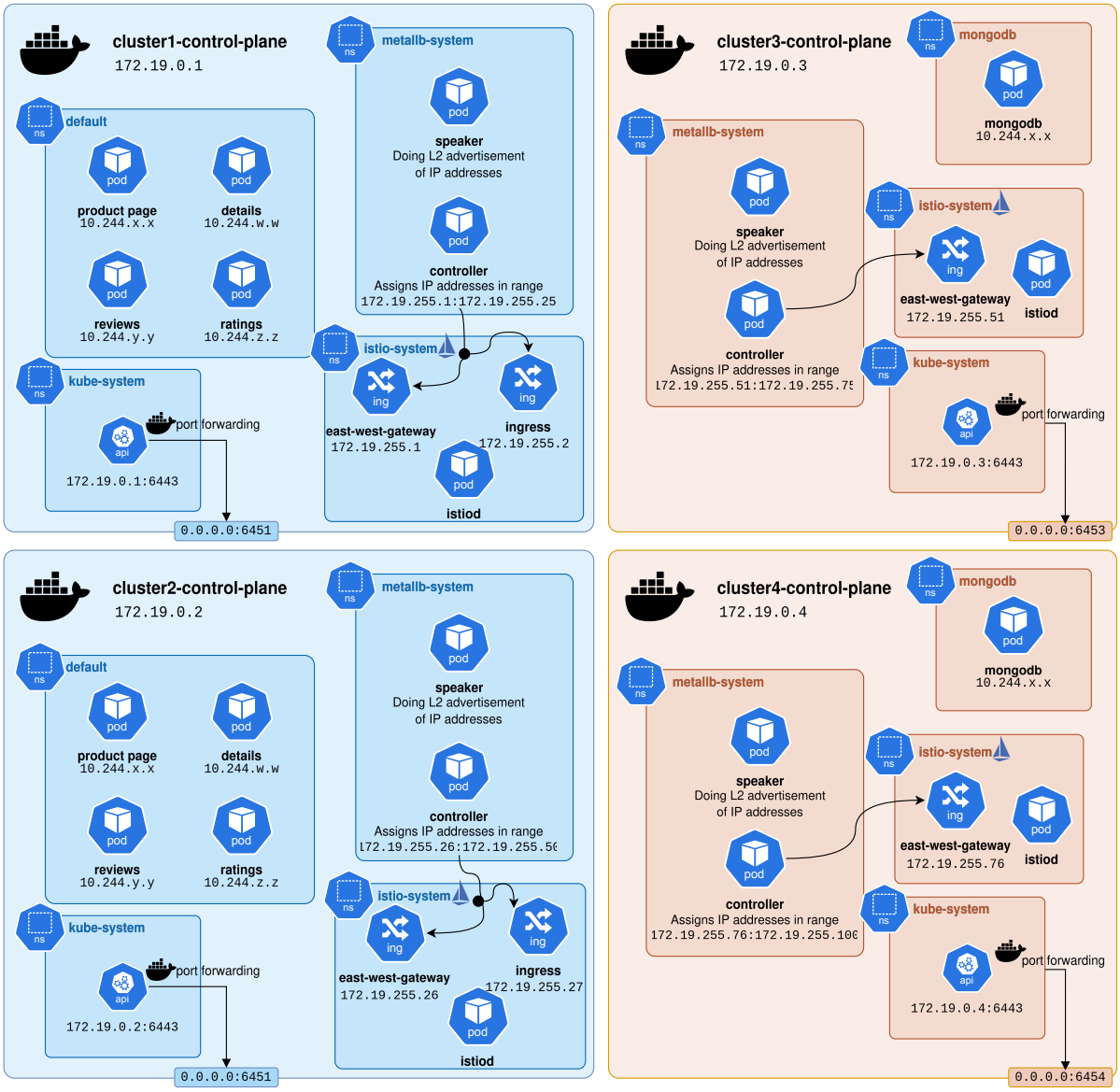


Figure 4.3: Continuation from Figure 4.2 with Istio also installed. This adds a number of new components to each cluster. `istiod` manages service discovery, configuration of proxies, and certificates. Two gateways are installed in the `Bookinfo` clusters - `ingress`, which accepts messages from the wider internet, and `east-west-gateway`, which allows inter-cluster communication. Only the `east-west-gateway` is installed in the database clusters. Note that to discover endpoints in the other clusters, Istio contacts their `kube-apis`.

4.2.3 Caddy

Caddy[10] is a simple to use alternative to nginx. Both caddy and nginx are designed as web servers, but are often used for routing communication through a system. This is caddy's purpose in this deployment - accepting requests on port 80 or 443 on the server, and reverse proxying them to the correct port based on the request's URL and the configuration of the caddy server. Caddy is able to perform round robin load balancing between multiple ports. With this, caddy performs the role of DNS in the original design (Figure 3.4), by load balancing requests to each of the ingress points.

Configuration for this behaviour in caddy is shown in Listing 4.3. Caddy runs in a docker container, listening on ports 80 (HTTP) and 443 (HTTPS), as shown in the final deployment in Figure 4.4.

```
http://bookinfo.clairegregg.com {
    reverse_proxy localhost:5501 localhost:5502 {
        lb_policy round_robin
    }
}
```

Listing 4.3: Configuration for caddy running on the server. While listening on port 80 and 443, if it receives any requests for http://bookinfo.clairegregg.com, it routes them to one of the two configured ingress points, at `localhost:5501` or `localhost:5502`, which both route to `product pages`.

4.2.4 Final Deployment

The final deployment, focusing on functionality rather than kind's structure, is shown in Figure 4.5. Note that this only has two replicas of each part of the system, intending to represent two clouds. The intention following this deployment was to increase to 3 replicas, especially due to MongoDB's expectation of an odd number of replicas. However, this was not possible, as discussed in the next section. This version was deployed successfully on a laptop, an Acer Swift SFX14-51G with a 12th Gen Intel® Core™ i7-1260P CPU and 16GB RAM. However, when that was the case, the device was unusable for anything else, including load testing.

[10] `https://caddyserver.com/`

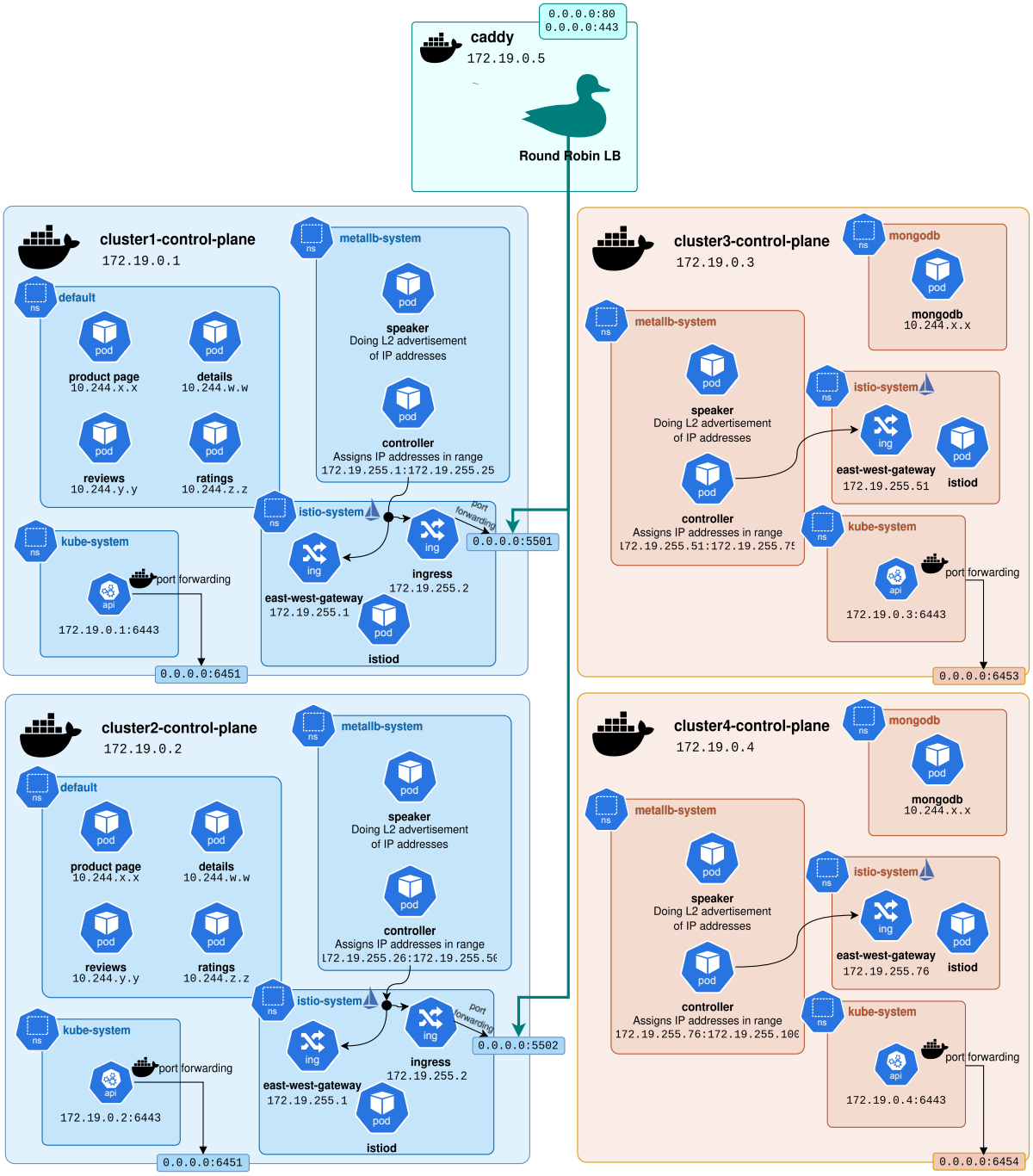


Figure 4.4: Continuation from Figure 4.3 with Caddy also configured on the server. Port forwarding is set up for the ingress from both `Bookinfo` clusters to `localhost` so caddy can access. Caddy listens on port 80 for HTTP connections and port 443 for HTTPS connections, and forwards them on to both `Bookinfo` clusters, load balancing using round robin.

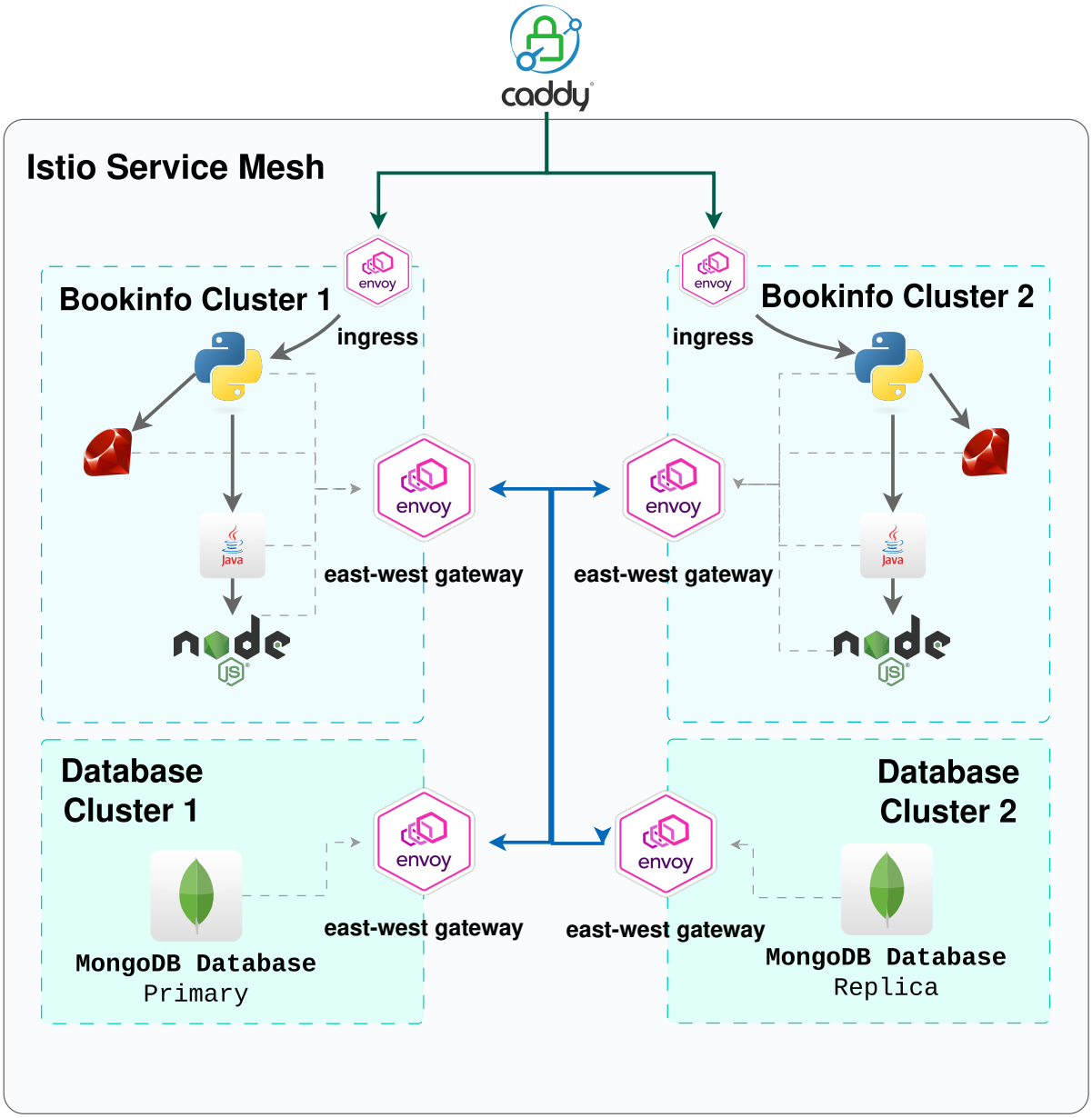


Figure 4.5: The final deployed version of the system. It simulates two clouds, each with 2 clusters deployed, one for `Bookinfo`, one for the database. These clusters are all deployed on kind, and have load balancing provided by MetalLB. Inter-cluster communication is managed by Istio. Ingress to the system is available in both `Bookinfo` clusters. Load is balanced between these two ingress points by caddy, which listens for requests and distributes them evenly between the two clusters.

4.3 Problems

When deployed to the server, massive latencies were observed, causing the deployment to fail due to timeouts. For example, latencies of up to 30 seconds were observed between the `details` microservice and the MongoDB database. Many possibilities were ruled out:

1. **Resource bottlenecking** - RAM usage never reached full. The highest average CPU usage reached was 4%, and the highest usage an individual CPU reached was 20%.
2. **Resource throttling** - While the server was originally configured to "performance per watt", changing this to "performance" did not affect the latency.
3. **General problem with inter-cluster communication** - The massive latency was not a generic inter-cluster communication problem, as, for example, latency between the `product page` microservice in `Bookinfo` cluster 1 and the `details` microservice in `Bookinfo` cluster 2 was under 10 microseconds.

While a solution to this problem may have been possible, none was able to be found in the time available for this project.

Chapter 5

Evaluation

This section will discuss an evaluation of the resilience of the system developed in this thesis. As previously discussed, it was not possible to deploy this solution fully, so an evaluation by exposing the system to a variety of failure scenarios was not possible. However, the original plan for experimental evaluation is discussed in Chapter 6, Conclusions, in the context of future work.

Instead, the evaluation takes a theoretical approach, using both binary metrics in section 5.1 and graph metrics (in section 5.2), based on a discussion by Welsh et al. [54]. Graph based evaluations make use of network robustness metrics to formally evaluate the system [54]. Additionally, evaluation is also performed in a more subjective manner in section 5.3, through a discussion of how the system should deal with specific failure types.

5.1 Binary Metrics Evaluation

Binary resilience metrics as described by Welsh et al. [54] come from the resilience disciplines described by Sterbenz et al. [49]. These metrics can be decided on by specifying for each resilience discipline listed, if the system has this feature. An overview of these features was given in Section 2.2. The binary evaluation is performed comparing to an unreliable single cluster deployment, and an example multi-cloud product with a centralised control plane, Google's Anthos. The results of this are given in Table 5.1. In general, the system performs well in challenge tolerance, which makes sense given the goal of the project. However, some binary metrics were not possible to gauge without an experimental approach. Also, despite Welsh et al.'s description of these as binary metrics, that is not true in this case, as a binary approach does not take things like partial fulfillment of goals, or partial failure of a system, into account.

Resilience Discipline	In solution?	Description	This Solution	Single Cluster	Anthos
Challenge Tolerance					
Fault tolerance	✓	Handles random uncorrelated failures - like a single server breaking. Technique is redundancy.	Redundancy of everything.	No redundancy	Redundancy of control plane & optional redundancy of components.
Survivability (excluding fault tolerance)	✓	Handles correlated failures, like natural disasters in a region or cyber attacks. Technique is diversity.	Has diversity geographically & of cloud providers.	No diversity	Only geographical diversity.
Disruption tolerance	~	Disruption tolerance handles breakdown in communication. Taking network partition as example.	Partition with majority would continue as normal, minority would have stale data & no writes.	N/A	Partition prevents centralised control plane from reaching some components.
Traffic tolerance	✗	Tolerating unpredictable load	Not designed for this, through Kubernetes autoscaling can be configured.	Could be configured through Kubernetes.	Enabled through GCP.
Trustworthiness					
Dependability	?	Reliance that can be placed on a system - combination of availability and reliability.	Not possible to calculate non-experimentally - based on mean time to failure (MTTF) and mean time to repair (MTTR).	Same as this solution	Autopilot (autoscaling) cluster has an SLA of >99.5%[1]
Security	✗	Property and measures taken to protect itself from unauthorised access.	Not a focus, while Istio supports mTLS, not configured here.	Same	Same - configuration specific.
Performability	?	Delivers performance required by service - eg response time.	Not possible to measure non-experimentally, especially given the response times in the deployment.	Same - application specific.	Same - application specific.

Table 5.1: Comparing binary resilience metrics for system designed in this thesis against a single cluster version, and an example multi-cloud Kubernetes product from the state of the art, Anthos. These disciplines of resilience come from [49]. ✓ represents that the system fulfils this resilience discipline, ~ represents partial fulfillment, ✗ represents failure to fulfil, and ? represents that not enough information is available.

[1] `https://cloud.google.com/kubernetes-engine/sla?hl=en`

5.2 Graph Metrics Evaluation

A variety of graph metrics are available for estimating the resilience of a system, but this evaluation will make use of a small selection, specifically to compare the resilience of a single cluster/cloud configuration to that of a multi cluster/cloud situation. This will start with a definition of the relevant graphs, go onto the calculation of some generic network resilience metrics, before finishing with some cloud specific resilience metrics (or more specifically, service based metrics). Code for these calculations is done using the python library `networkx` [21], and is available on GitHub.

5.2.1 Graph Definitions

A directed graph representation of the system in a single cluster deployment can be seen in Figure 5.1a, where directed lines represent dependencies between components. The equivalent for the multi-cloud deployment is in Figure 5.1b, where the number in each component shows which of clouds 1, 2, or 3 it is deployed in. Lines between the clouds show the support for load balancing between clouds, allowing instances of a service in any 2 regions to go down before the system fails as a whole. Note that this example of a deployment across 3 clouds is not the only way a multicloud solution can work, although 3 is the recommended minimum for a highly available system in MongoDB, and the number of replicas, and therefore clusters/cloud providers, must be odd.[2]

5.2.2 Generic Network Resilience Metrics

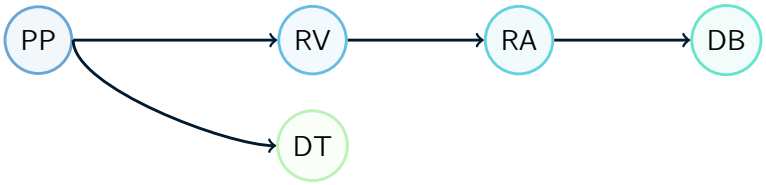
This selection of network resilience metrics is taken from Welsh et al. [55], given their single value outputs. These metrics are:

1. **Average node connectivity:** The more standard k-connectivity is the maximum number of nodes or edges possible to remove from a network before it becomes disconnected or partitioned. This is given in Equation 5.1, where u and v are two nodes/vertices in the graph, and $k_G(u, v)$ is the maximum number of nodes possible to remove before there is no path between the nodes.

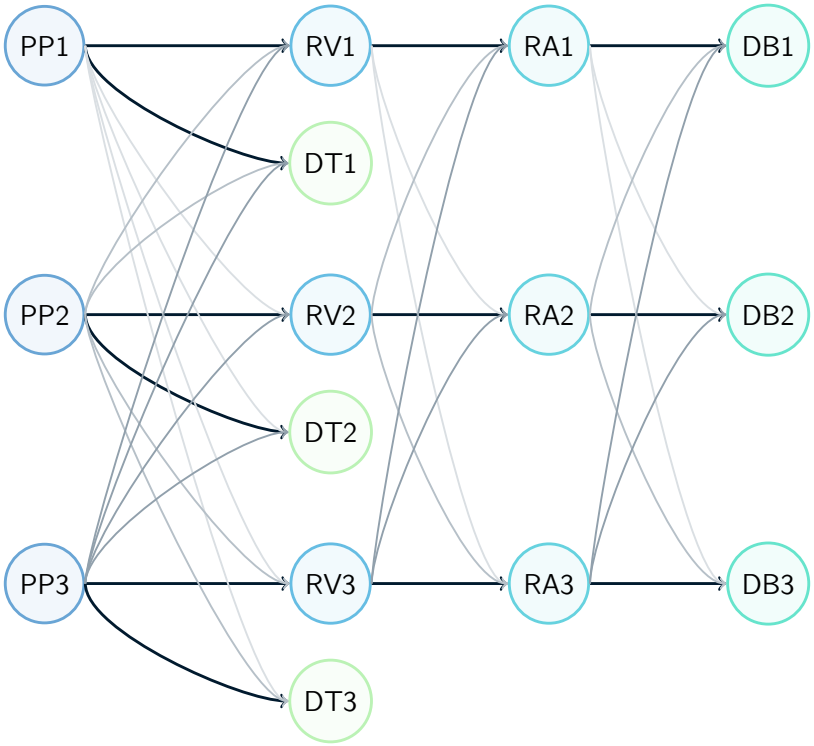
$$k(G) = min(k_G(u, v) : u,\ v \in V(G)) \tag{5.1}$$

   However, in directed acyclic graphs (which this system is), k-connectivity is always 0, because the graph does not start off with paths between all nodes. Instead, average node connectivity can be used, which gives a better view of how connected elements of the graph are to each other, and therefore how resilient they are to failures. This

[2] `https://www.mongodb.com/docs/manual/administration/production-checklist-operations/#std-label-production-checklist-replication`



(a) A graph representation of the single cluster solution.



(b) A graph representation of the multi-cloud solution.

Figure 5.1: PPs represent `product pages`, DTs represent `details` microservices, RVs represent `reviews` microservices, RAs represent `ratings` microservices, and DBs represent MongoDB replicated databases. Directed lines show the dependencies between microservices. This graph representation show an example deployment across cloud providers 1, 2 and 3. An Istio service mesh provides the links between services.

is given in 5.2, where p is the order (number of nodes) of the graph. Higher average node connectivity is better, as more elements can break before problems occur.

$$\bar{k}(G) = \frac{\sum_{u,v} k_g(u,v)}{\binom{p}{2}} \tag{5.2}$$

2. **Average shortest path:** This represents how easy it is to get from one node to another. Lower values of this are better as fewer elements are relied on to reach a goal.
3. **Network criticality:** This represents the robustness of a network against changes to the network structure, originally defined by Tizghadam et al. [51]. Lower values of this are better.

Metric	More Robust	Single Cluster	Multi-cloud
Average Node Connectivity	↑	0.35	0.56
Average Shortest Path	↓	0.55	0.47
Network Criticality (normalised)	↓	0.095	0.009
Effective Graph Conductance	↑	0.1	0.127

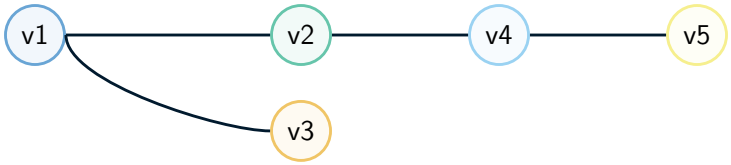
Table 5.2: Results for generic graph resilience metrics, comparing the single cluster and multi-cloud example graphs. The second column represents if higher or lower values represent more robustness.

4. **Effective graph resistance/Effective graph conductance:** Effective graph resistance is based on logic from physical networks - for example, electrical networks, where there is resistance in electrical lines making it more difficult for current to travel [53]. Effective graph resistance is then the sum of effective resistances between every two vertices [17], where the effective resistance between 2 vertices is increased the fewer paths between the two vertices are available [30]. Normalising this for a variable number of nodes in the graph as described by Alenazi et al. [2] gives the effective graph conductance. As the two graphs being evaluated have a different number of nodes, effective graph conductance is used. Effective graph conductance increases with robustness, so higher values are better.

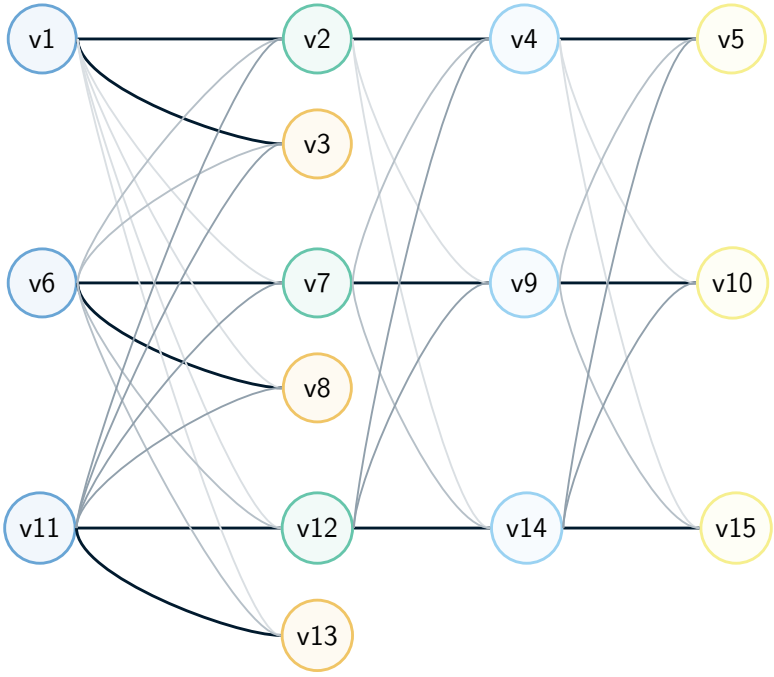
Values for each of these metrics for the single cluster and multi-cloud graphs are given in Table 5.2. It can be seen that all graph resilience metrics improve with the move from a single cluster to a multi-cloud deployment.

5.2.3 Service Based Metrics

These metrics are based on Rosenkrantz et al.'s work on resilience metrics for service-oriented networks [44]. These differ from the previously used metrics by introducing the idea that multiple nodes in a graph can be providing the same service, and each node can depend on a certain set of services This allows natural modelling of cloud based microservices. First, some redefinitions are required. The graphs themselves are redrawn as shown in Figure 5.2. Differences include the renaming of nodes to match how variables will be defined later, and making the graph undirected, as dependencies will be handled separately.



(a) A graph representation of the single cluster solution in a service oriented manner.



(b) A graph representation of the multicloud solution in a service oriented manner.

Figure 5.2: Blue represents product pages, orange represents details microservices, green represents reviews microservices, light blue represents ratings microservices, and yellow represent MongoDB replicated database. This graph representation show an example deployment across cloud providers 1, 2 and 3. An Istio service mesh provides the links between services.

With the graphs redrawn, some values need to be defined. First is S, the set of services. This is the same in the single cluster and multi-cloud setup:

- s_1 Product page
- s_2 Reviews service
- s_3 Details service
- s_4 Ratings service
- s_5 Database

Next are sets A_i, which specify the services available at each node v_i, and sets N_i, which specify the services depended on at each node v_i, as labelled in Figure 5.2. These sets for the single cluster setup are given in Equation 5.3, and for multi-cloud are given in Equation 5.4.

$$\begin{aligned} A_1 &= \{s_1\} \quad N_1 = \{s_2, s_3\} \\ A_2 &= \{s_2\} \quad N_2 = \{s_4\} \\ A_3 &= \{s_3\} \quad N_3 = \{\} \\ A_4 &= \{s_4\} \quad N_4 = \{s_5\} \\ A_5 &= \{s_5\} \quad N_5 = \{\} \end{aligned} \tag{5.3}$$

$$\begin{array}{llllll} A_1 = \{s_1\} & A_6 = \{s_1\} & A_{11} = \{s_1\} & N_1 = \{s_2, s_3\} & N_6 = \{s_2, s_3\} & N_{11} = \{s_2, s_3\} \\ A_2 = \{s_2\} & A_7 = \{s_2\} & A_{12} = \{s_2\} & N_2 = \{s_4\} & N_7 = \{s_4\} & N_{12} = \{s_4\} \\ A_3 = \{s_3\} & A_8 = \{s_3\} & A_{13} = \{s_3\} & N_3 = \{\} & N_8 = \{\} & N_{13} = \{\} \\ A_4 = \{s_4\} & A_9 = \{s_4\} & A_{14} = \{s_4\} & N_4 = \{s_5\} & N_9 = \{s_5\} & N_{14} = \{s_5\} \\ A_5 = \{s_5\} & A_{10} = \{s_5\} & A_{15} = \{s_5\} & N_5 = \{\} & N_{10} = \{\} & N_{15} = \{\} \end{array} \tag{5.4}$$

A node v_i is a *demand point* for a service s_j if $s_j \in N_j$.

A node v_i is a *service point* for a service s_j if $s_j \in A_j$.

A (sub)network is *self-sufficent* if each service required in that (sub)network is also provided by some node in the same (sub)network. Otherwise it is *deficient*.

With these, the edge resilience is calculated as the minimum number of edges to be deleted to cause a service deficiency in a network, and the equivalent for nodes. The results for this are given in Table 5.3. They show an obvious increase in resilience.

Setup	Edge Resilience	Node Resilience
Single cluster	1	1
Multi-cloud	3	3

Table 5.3: Edge and node resilience of single cluster and multi-cloud setups, per [44].

5.3 Situational Analysis

This section will discuss the reaction of the system to a variety of disaster situations. For this, the system shown in Figure 5.3 is used.

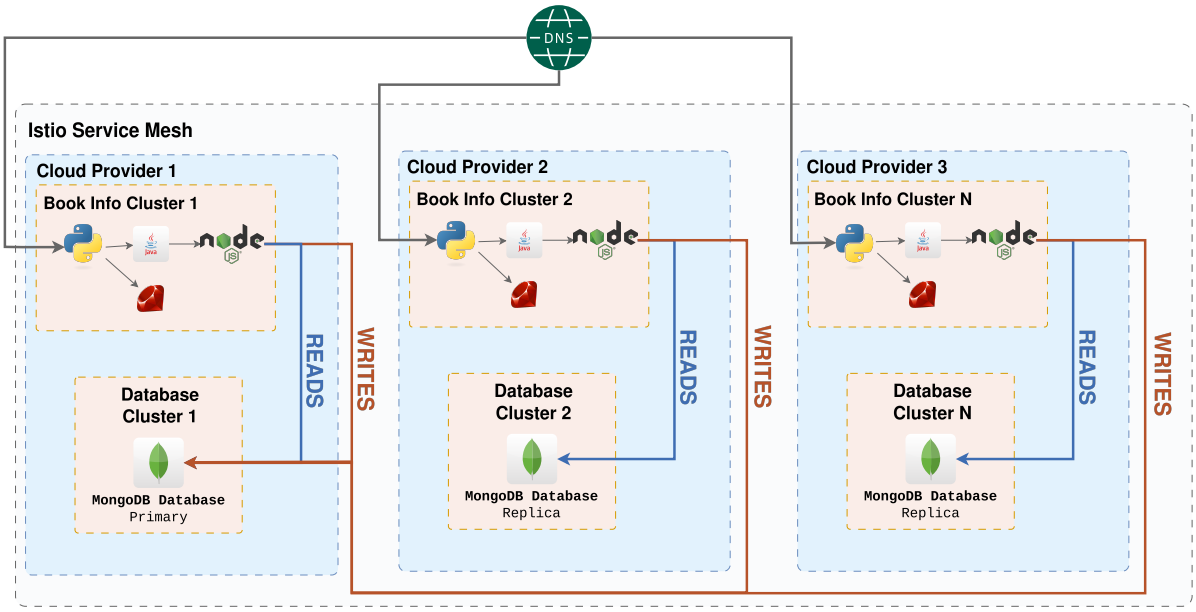


Figure 5.3: Deployed system to be evaluated in differing situations.

5.3.1 Pod Failure

There are two places pod failures can happen - in a `Bookinfo` cluster or in a database cluster.

Pod in a Bookinfo Cluster Fails

There are slightly different behaviors based on which pod in the `Bookinfo` cluster fails, both shown in Figure 5.4. In both cases, Kubernetes would see the failure and automatically restart the pod. The differing behaviour is during the time the pod is down. If the pod down is one of the `details`, `reviews`, or `ratings` microservice, this can be managed by Istio. Istio will send the messages to an alternate instance of the microservice in a different cloud if one instance has failed. As the `product page` is the entry point to the application, Istio cannot manage it. Instead, DNS round robin will swap between the multiple DNS entries for the instance until one connects. Eventually, Kubernetes restarts the pod, fixing the problem.

Pod in a Database Cluster Fails

The only pod available to fail in a database cluster is MongoDB. There are two different ways this can go. If the pod that goes down is a replica, the MongoDB client will manage rerouting to other replicas for reads, while writes continue to go to the primary. Kubernetes will restart the failed pod in the background, and it will reenter as a replica.

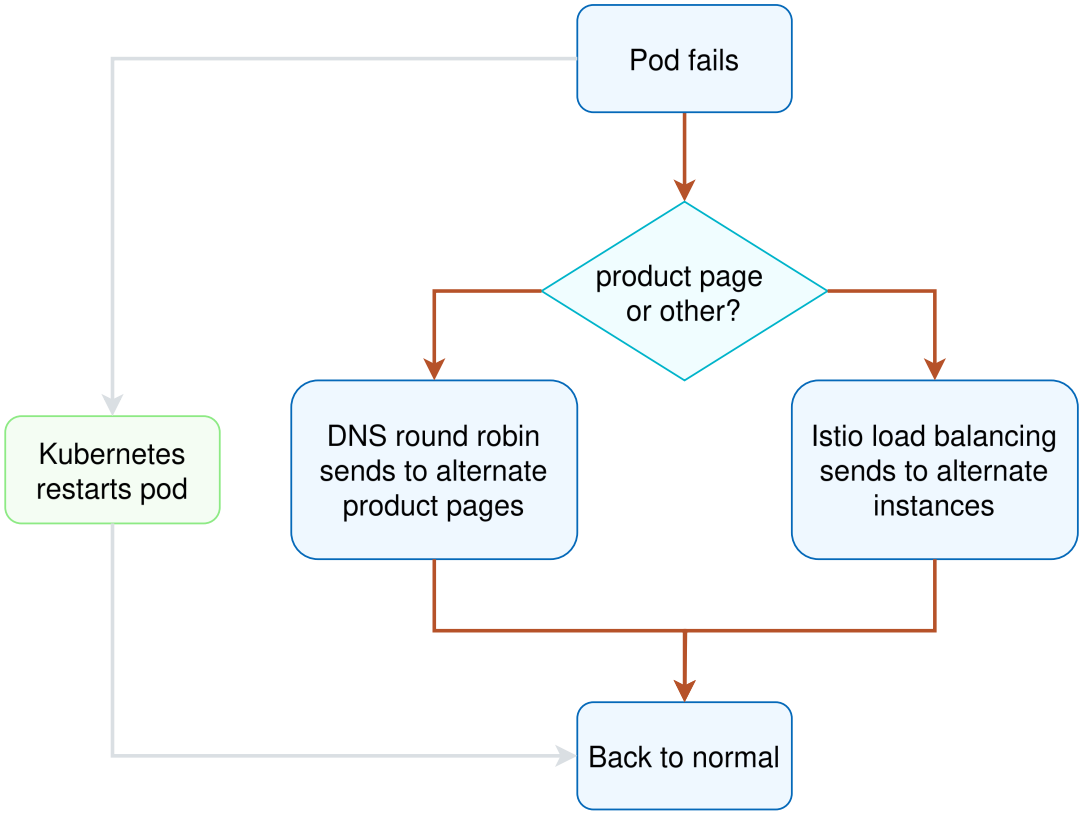


Figure 5.4: Behaviour of the system when a pod fails in one of the `Bookinfo` clusters.

In the alternate case, if the primary fails, there must be a leader election, as shown in Figure 5.5. While this is ongoing (for median 12 seconds[3], although this could be decreased by decreasing the default 10s heartbeat timeout), no writes would be accepted, but reads could still be routed to replicas. Once the leader election is complete, the new primary is able to accept writes. Once the original failed primary is restarted by Kubernetes, it rejoins the replica set as a secondary/replica.

5.3.2 Zone/Regional/Cloud Provider Failure

This section covers all of the possibilities that could cause all resources held in one cloud provider to fail. First, a discussion of how each type of failure could lead to all resources in a cloud provider failing.

Zone outage Zone failure would cause a cluster outage if the cluster was set up as a zonal cluster - meaning it only has a single control plane in a single cloud zone.[4] If both the database cluster and bookinfo cluster are deployed in the same zone, a zonal failure would cause both to fail. Zonal deployments are chosen for their reduced price.

[3] `https://www.mongodb.com/docs/manual/replication/#automatic-failover`

[4] For example, `https://cloud.google.com/kubernetes-engine/docs/concepts/configuration-overview#zonal`

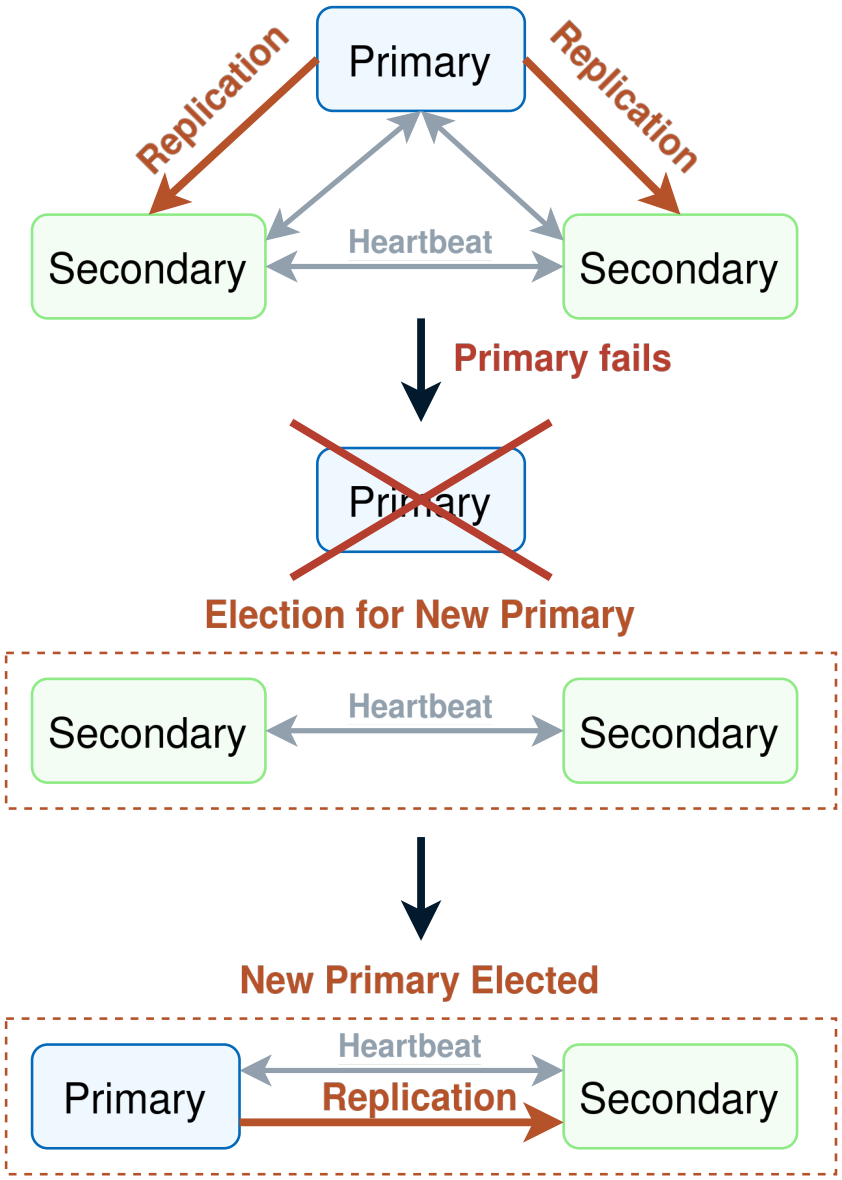


Figure 5.5: Sequence of actions when the primary MongoDB pod fails, performing leader election.

Regional outage If a cluster is set up to be a regional cluster, zone outages will not cause application failure (assuming appropriate replication across zones). However, a regional outage would still cause the cluster to fail.

Cloud provider outage No multi-region clusters are offered, as this would have problems with latency. However, if multiple clusters are deployed in the same cloud providers, all of them will fail in a cloud provider outage - this is why multi-cloud deployment is chosen.

In the case of a cluster failing, no specific self healing is available. Instead, the system behaves as if all pods in the zone/region/cloud provider have failed, following the steps discussed in Section 5.3.1, and waits for the instances to be brought back up by the cloud provider.

Taking as an example Google Cloud's SLA[5] (Service Level Agreement - the amount of uptime the cloud provider promises to the customer, needing to pay fines if it is not upheld), monthly uptime percentage for zonal cluster control planes is 99.5%, and monthly uptime percentage for regional cluster control planes is 99.95%. This means zonal clusters have an SLA for around 3 hours 40 minutes downtime per month, while regional accepts around 22 minutes downtime per month. This gives an idea of the protection granted by a multicloud deployment for a theoretical critical application.

5.3.3 Network Partition

The system can handle a network partition easily. Given the every microservice in the bookinfo cluster is stateless, no change would occur there. The only difference would be a lack of load balancing between the two partitions, which would only cause a problem if a pod fails without a backup in its partition. MongoDB would fare slightly differently, as it manages all of the state in the application. If the primary database replica is in the cluster with the majority of replicas, it will remain the primary and the majority partition will continue as normal. Otherwise, a leader election will happen in the majority partition, leading to the same outcome.

[5] `https://cloud.google.com/kubernetes-engine/sla`

Chapter 6

Conclusions

This thesis began by introducing the concept of multi-cloud, and why it may be necessary. This is due to the risk of cloud providers having global failures, which is not protected against by any other public cloud deployment approach. As well as mitigating this risk, multi-cloud systems also avoid vendor lock-in. Kubernetes was discussed as a popular orchestration tool that is particularly useful for the cloud. The architecture of a single Kubernetes cluster was discussed, along with a discussion of resilient components of its design.

In the state of the art, existing approaches to multi-cloud deployment were discussed, with a particular focus on the two most popular approaches - Apache jClouds and Kubernetes. Apache jClouds provides compliance with OASIS standards, and allows the generic use of many cloud resources through one API. There are multiple examples of Apache jClouds being used for multi-cloud deployment - such as Apache Brooklyn and Cloudiator. Apache Brooklyn has been extended on multiple occasions for further features, including adding self-healing based on user-defined metrics. However, there is not significant usage of jClouds outside of academia, so focus went to multi-cloud Kubernetes. Existing commercial solutions for multi-cloud Kubernetes exist - Google Anthos, Rancher, and Platform 9, among others. However, all these examples have a centralised control plane. Instead, a federated approach is desired. KubeFed v1 and v2 are unfortunately abandoned projects, so the best federation approach appears to be using a service mesh (Istio or Linkerd), as is done in RedHat's Open Shift Service Mesh. It is important to note that due to complexity, there are greater security risks in multi-cloud deployments.

Resilience is also discussed in the state of the art, with definitions of resilience disciplines from Sterbenz et al. [49] used to classify work on resilience performed in the relevant literature. This also allows the narrowing in of the focus of this project to challenge tolerance in particular. The measurement of resilience is also discussed, finding that there are generally four ways to measure resilience - binary feature evaluation, state based evaluation, performance metrics, and graph metrics. Graph metrics have a long history, including out-

side of application modelling. Many of these metrics were originally designed to measure the robustness of communication networks. However, they can certainly be used for cloud application.

Testing in software engineering is discussed, in order to provide context to what chaos engineering is used for. Testing happens at a number of levels, with chaos tests and load tests running as E2E tests. These test the systems as the user would make use of it. Chaos engineering is then discussed in more detail, including examples of tools, and the intended process of chaos engineering. Chaos engineering then allows collection of metrics which can be used to evaluate the reliability of systems, and can best be categorised as performance metrics - although perhaps "performance under stress" metrics would be more accurate.

The thesis continues by discussing the design of a multi-cloud Kubernetes application based on a sample application provided with the service mesh Istio. The architecture of Istio itself is also discusses. Modifications to this sample application were required to allow full testing of a database. Modifications were also required to make the application multi-cluster, so that it can be deployed to multiple clouds. These changes included replication of data between clusters using MongoDB's replica sets.

Unfortunately, the deployment of this design did not go ahead as planned. Initially, due to the lack of cloud resources, it was decided to deploy the application to a bare metal server, and the changes required for this deployment were discussed. These included using kind to allow multiple clusters to be deployed on a single machine, using MetalLB to replace the load balancers which would usually be provided by a cloud provider, and replacing DNS round robin with Caddy round robin. The architecture and networking of a multi-cluster setup using kind on a single machine was discussed in detail. However, something about this deployment did not work, and massive latencies were observed, to the degree that the application failed due to timeouts. Debugging and fixes were attempted, but no successful fix was possible.

This affected the evaluation of the project, which was originally intended to use chaos engineering and load testing to find estimates for performance-based metrics like mean time to repair (MTTR) and response time when the system was under load. Instead, a combination of binary and graph metrics were used for the evaluation, along with an analysis of how the system is expected to behave under specific disaster situations. Binary metrics were used to compare the theoretical solution in this thesis against a non-resilient single cluster setup, and a commercial multi-cloud Kubernetes option, Google's Anthos. Graph metrics generally focused on how well the system could continue to work if nodes (Kubernetes pods) or edges (connections between pods) were severed. All of these metrics showed an improvement in resilience by switching to a multi-cloud deployment.

This unsuccessful deployment leads this section to the question: what went wrong in this project? While there are certainly many factors, and part of the cause is due to time

limitations, the best answer available is the complexity of multi-cloud systems. This will be discussed in detail in Section 6.1, while Section 6.2 will discuss the future work to be done on this project.

6.1 Why is Multi-Cloud so Difficult?

This section is about both why multi-cloud is difficult, as well as why multi-cluster Kubernetes is difficult. Both come down to the same reason - complexity.

Tomarchio et al. [52] cite a number of challenges in the world of multi-cloud orchestration. First among these, and the element that is most often discussed, is the lack of interoperability between cloud providers. Indeed, there is very little incentive for cloud providers to make their tools interoperable, as they benefit from vendor lock-in. There is also a lack of existing architectural patterns to work from in multi-cloud, and basically a lack of history and experience in the area. Communications between clouds and an increased dependency on an increased number of networks also leads to difficulty.

However, the primary challenge, also discussed by Alonso et al. [3], is complexity. There is no specific way to match requirements between different cloud providers, and no standard way to compare services. There is also a lot of difficulty in performing cost comparisons between cloud providers. These problems generally are due to a lack of shared ways to describe cloud components.

As discussed in the State of the Art, increased security risk also causes difficulty in a multi-cloud setting. This is due to the larger attack surface, heterogenous configurations, and more opportunities for mismatching security policies. This makes it difficult to choose a multi-cloud solution when working with sensitive data.

Speaking from a business perspective, there is also great difficulty in choosing multi-cloud solutions due to the financial cost incurred. For a three cloud setup, businesses will often have to pay around three times the cost, compared to a single cloud deployment. Maintenance costs also increase dramatically in a multi-cloud deployment, due in large part to the complexity previously discussed. Overall, the resilience improvement must have very significant financial gain for multi-cloud to be seriously considered as a solution.

The complexity of multi-cluster Kubernetes is slightly different. Much of this complexity is due to the design of Kubernetes for cloud application developers rather than research. If the `Bookinfo` application were being set up in a cloud environment, much of the complexity would not have been present. Instead, the complexity in this project was mostly due to the need to work with tools like kind, which was never intended for actual deployment, and MetalLB, a project started by a single person to deal with the lack of load balancers available outside managed cloud environments. Documentation for these tools was limited, and this

project definitely pushed them outside of their intended usage (kind in particular). Aside from that, the increased reliance on networks as discussed for multi-cloud is also true in multi-cluster Kubernetes. This network complexity surely contributed to the failure to deploy discussed in the implementation chapter.

6.2 Future Work

The primary goal in future work done on this project is to get the system described in the design and implementation chapters working. Ideally, this would be done with 6 clusters (3 `Bookinfo`, 3 database) deployed across 3 public cloud - AWS, Azure, and GCP. With the resources available to deploy this to the cloud, this deployment should be fairly simple using all resources available on the project Github.

With this deployment up, the exciting work would begin, using a combination of chaos engineering and load testing to test the resilience of the application under a number of circumstances. There are three elements to be changed and tested in this experiment - load tests, chaos tests, version of deployment.

Load tests should be done with Locust[1], a Python load testing library, as performed by Singh et al [48]. The recommended tests are described in Table 6.1. These verify all elements of the service work, track the response times of each element, and test that the database is not losing data.

Name	Details	Metric
Front end	Test that the website loads	Response time
The website loading allows for partial failure, where specific microservices fail and their data is not included. To test this, API calls to retrieve data from specific microservices are required.		
`getProduct`	Load test the API endpoint for the details microservice	Response time
`getProductReviews`	Load test the API endpoint for the reviews microservice	Response time
`getProductRatings`	Load test the API endpoint for the ratings microservice	Response time
Database consistency	Perform a write against the database, and check it is still present at intervals of 10s for 5 minutes.	Time for consistency to fail, time for it to return.

Table 6.1: Load tests recommended for testing the deployed `Bookinfo` application.

[1] `https://locust.io/`

Type	Details	Time to repair
Pod failure	Simulate each pod type failing using chaos mesh - `product page`, `details`, `reviews`, `ratings`, and `mongodb`.	Time until the pod is restarted and starts handling load.
Cluster failure	Simulate a full cluster failure using chaos mesh.	No time to repair - only observing load testing changes.
Network partition	Delete an east-west gateway in one of the clusters	If primary database is in smaller partition, time until leader election is complete

Table 6.2: Chaos tests to run against `Bookinfo` deployments.

The load tests should then be combined with chaos tests. The recommended chaos tests are described in Table 6.2. While these chaos engineering tests are being run, metrics should be collected from load tests, along with the times to repair discussed in the table. Fault injection at the application layer using Istio[2] may also be considered.

These load and chaos tests should be run on a number of different deployments:

1. **Single cluster** - With only one replica of each microservice and the database. The network partition chaos test will have no effect, and the cluster failure will bring down the service entirely, so these chaos tests may be skipped in this deployment approach.
2. **Multi-cluster, single zone** - 3 sets of `Bookinfo` and database clusters, deployed in the same zone in one cloud provider.
3. **Multi-cluster, multi zone** - 3 sets of `Bookinfo` and database clusters, deployed in different zones in one cloud provider.
4. **Multi-cluster, multi region** - 3 sets of `Bookinfo` and database clusters, deployed in different regions in one cloud provider.
5. **Multi-cluster, multi cloud** - 3 sets of `Bookinfo` and database clusters, with each set deployed to a different public cloud.

While graph metrics were not originally part of the plan for evaluating this project, they are an interesting formal way to discuss the resilience of a system. It would be interesting to compare these metrics to the experimentally gathered mean time to repair (MTTR), to give an idea of their accuracy.

In summary, while there is plenty of discussion in the literature about aspects and descriptions of resilience, it would be interesting to gain more empirical data on the impact of certain deployment strategies on resilience, specifically in a cloud setting.

[2] `https://istio.io/latest/docs/tasks/traffic-management/fault-injection/`

Bibliography

[1] Afolaranmi, S. O., Ferrer, B. R., & Martinez Lastra, J. L. (2018). A Framework for Evaluating Security in Multi-Cloud Environments. In *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society* (pp. 3059–3066).

[2] Alenazi, M. & Sterbenz, J. (2015). Comprehensive comparison and accuracy of graph metrics in predicting network resilience. (pp. 157–164)., https://doi.org/10.1109/DRCN.2015.7149007.

[3] Alonso, J., Orue-Echevarria, L., Casola, V., Torre, A. I., Huarte, M., Osaba, E., & Lobo, J. L. (2023). Understanding the challenges and novel architectural models of multi-cloud native applications – a systematic literature review. *Journal of Cloud Computing*, *12*(1), https://doi.org/10.1186/s13677-022-00367-6.

[4] Alqahtani, H. S. & Sant, P. (2016). Multiple-Clouds Computing Security Approaches: A Comparative Study. In *Proceedings of the International Conference on Internet of Things and Cloud Computing*, ICC '16 New York, NY, USA: Association for Computing Machinery.

[5] AlZain, M. A., Pardede, E., Soh, B., & Thom, J. A. (2012). Cloud Computing Security: From Single to Multi-clouds. In *2012 45th Hawaii International Conference on System Sciences* (pp. 5490–5499).

[6] AWS (2021). Summary of the AWS Service Event in the Northern Virginia (US-EAST-1) Region https://aws.amazon.com/message/12721/.

[7] Baur, D. & Domaschka, J. (2016). Experiences from building a cross-cloud orchestration tool. In *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms*, CrossCloud '16 New York, NY, USA: Association for Computing Machinery.

[8] Belamaric, J. (2021). CoreDNS for Kubernetes Service Discovery, Take 2 https://coredns.io/2017/03/01/coredns-for-kubernetes-service-discovery-take-2/.

[9] Berenberg, A. & Calder, B. (2022). Deployment Archetypes for Cloud Applications. *ACM Comput. Surv.*, *55*(3), https://doi.org/10.1145/3498336.

[10] Bergstrom, J. (2022). Chaos Engineering. *The ITEA Journal of Test and Evaluation*, *43*(4), 208–218 https://itea.org/wp-content/uploads/2023/01/223438-ITEA-Journal-Dec22.pdf#page=22.

[11] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, Omega, and Kubernetes. *Commun. ACM*, *59*(5), 50–57, https://doi.org/10.1145/2890784.

[12] Carrasco, J., Durán, F., & Pimentel, E. (2018). Trans-cloud: CAMP/TOSCA-based bidimensional cross-cloud. *Computer Standards & Interfaces*, *58*, 167–179, https://doi.org/10.1016/j.csi.2018.01.005.

[13] ChaosCommunity (2019) http://principlesofchaos.org/.

[14] Cloudflare (2025) https://www.cloudflare.com/en-gb/learning/dns/glossary/round-robin-dns/.

[15] Eisenberg, V. (2018). Consuming external mongodb services https://istio.io/latest/blog/2018/egress-mongo/.

[16] Elkhatib, Y. & Poyato, J. P. (2023). An Evaluation of Service Mesh Frameworks for Edge Systems. In *Proceedings of the 6th International Workshop on Edge Systems, Analytics and Networking*, EdgeSys '23 (pp. 19–24). New York, NY, USA: Association for Computing Machinery.

[17] Ellens, W., Spieksma, F., Van Mieghem, P., Jamakovic, A., & Kooij, R. (2011). Effective graph resistance. *Linear Algebra and its Applications*, *435*(10), 2491–2506, https://doi.org/10.1016/j.laa.2011.02.024. Special Issue in Honor of Dragos Cvetkovic.

[18] etcd (2025). etcd API Guarantees https://etcd.io/docs/v3.5/learning/api_guarantees/.

[19] Foley, M. J. (2020). Microsoft's March 3 azure east US outage: What went wrong (or right)? https://www.zdnet.com/article/microsofts-march-3-azure-east-us-outage-what-went-wrong-or-right/.

[20] GCP (2016). Google Compute Engine Incident #16007 https://status.cloud.google.com/incident/compute/16007.

[21] Hagberg, A. A., Schult, D. A., & Swart, P. J. (2008). Exploring Network Structure, Dynamics, and Function using NetworkX. In G. Varoquaux, T. Vaught, & J. Millman (Eds.), *Proceedings of the 7th Python in Science Conference* (pp. 11–15). Pasadena, CA USA.

[22] Houshmand, M. (2023). A guide to creating a true hybrid/multi-cloud architecture with OSSM federation https://www.redhat.com/en/blog/a-guide-to-creating-a-true-hybrid/multi-cloud-architecture-with-ossm-federation.

[23] Istio (2024). What is Istio? https://istio.io/latest/docs/overview/what-is-istio/.

[24] Istio (2025). Architecture https://istio.io/latest/docs/ops/deployment/architecture/.

[25] Izrailevsky, Y. & Bell, C. (2018). Cloud Reliability. *IEEE Cloud Computing*, *5*(3), 39–44, https://doi.org/10.1109/MCC.2018.032591615.

[26] Izrailevsky, Y. & Tseitlin, A. (2011). The Netflix Simian Army https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116.

[27] Jadeja, Y. & Modi, K. (2012). Cloud computing - concepts, architecture and challenges. In *2012 International Conference on Computing, Electronics and Electrical Technologies (ICCEET)* (pp. 877–880).

[28] Jin, R., Muench, P., Janssen, T., Hatfield, B., & Deenadhayalan, V. (2024). Baking Disaster-Proof Kubernetes Applications with Efficient Recipes. In *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*, ICPE '24 Companion (pp. 174–180). New York, NY, USA: Association for Computing Machinery.

[29] Kavas, E. (2023). From Paris to the World: Lessons Learned from GCP's Europe-west9 region outage https://www.erol.ca/from-paris-to-the-world-lessons-learned-from-gcps-europe-west9-region-outage/.

[30] Klein, D. J. & Randić, M. (1993). Resistance distance. *Journal of Mathematical Chemistry*, *12*(1), 81–95, https://doi.org/10.1007/bf01164627.

[31] Kubernetes (2024) https://kubernetes.io/docs/concepts/overview/components/.

[32] Kubernetes (2025) https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/.

[33] Luksa, M. (2018a). *11: Understanding Kubernetes Internals*, (pp. 309–345). Manning.

[34] Luksa, M. (2018b). *4.1: Keeping Pod Healthy*, (pp. 85–90). Manning.

[35] Miyachi, C. (2021). The Rise of Kubernetes. In *2021 Cloud Continuum* (pp. 1–5).

[36] OASIS (2013). Topology and Orchestration Specification for Cloud Applications Version 1.0 https://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html.

[37] OASIS (2018). OASIS CAMP: Cloud Application Management for Platforms Version 1.2 https://docs.oasis-open.org/camp/camp-spec/v1.2/camp-spec-v1.2.html.

[38] Ongaro, D. & Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14 (pp. 305–320). USA: USENIX Association.

[39] Paladi, N., Michalas, A., & Dang, H.-V. (2018). Towards Secure Cloud Orchestration for Multi-Cloud Deployments. In *Proceedings of the 5th Workshop on CrossCloud Infrastructures & Platforms*, CrossCloud'18 New York, NY, USA: Association for Computing Machinery.

[40] Pham, L. M., Tchana, A., Donsez, D., de Palma, N., Zurczak, V., & Gibello, P.-Y. (2015). Roboconf: A Hybrid Cloud Orchestrator to Deploy Complex Applications. In *2015 IEEE 8th International Conference on Cloud Computing* (pp. 365–372).

[41] Raj, P. (2021). Chapter One - Demystifying the blockchain technology. In S. Aggarwal, N. Kumar, & P. Raj (Eds.), *The Blockchain Technology for Secure and Smart Applications across Industry Verticals*, volume 121 of *Advances in Computers* (pp. 1–42). Elsevier.

[42] Raj, P., Vanga, S., & Chaudhary, A. (2023). *The Observability, Chaos Engineering, and Remediation for Cloud-Native Reliability*, (pp. 71–93). Wiley.

[43] Reece, M., Rastogi, N., Lander, T., Dykstra, J., Mittal, S., & Sampson, A. (2024). Defending Multi-Cloud Applications Against Man-in-the-Middle Attacks. In *Proceedings of the 29th ACM Symposium on Access Control Models and Technologies*, SACMAT 2024 (pp. 47–52). New York, NY, USA: Association for Computing Machinery.

[44] Rosenkrantz, D. J., Goel, S., Ravi, S. S., & Gangolly, J. (2009). Resilience Metrics for Service-Oriented Networks: A Service Allocation Approach. *IEEE Transactions on Services Computing*, *2*(3), 183–196, https://doi.org/10.1109/TSC.2009.18.

[45] Sharma, V. (2022). Managing Multi-Cloud Deployments on Kubernetes with Istio, Prometheus and Grafana. In *2022 8th International Conference on Advanced Computing and Communication Systems (ICACCS)* (pp. 525–529).

[46] Sharwood, S. (2018). AWS outage killed some cloudy servers, recovery time is uncertain https://www.theregister.com/2018/06/01/aws_outage/.

[47] Sherrill, A. (2024). The July 2024 Microsoft Azure Service Outage https://tenhats.com/takeaways-from-the-july-2024-microsoft-azure-service-outage/.

[48] Singh, S., Muntean, C. H., & Gupta, S. (2024). Boosting Microservice Resilience: An Evaluation of Istio's Impact on Kubernetes Clusters Under Chaos. In *2024 9th International Conference on Fog and Mobile Edge Computing (FMEC)* (pp. 245–252).

[49] Sterbenz, J. P., Hutchison, D., Çetinkaya, E. K., Jabbar, A., Rohrer, J. P., Schöller, M., & Smith, P. (2010). Resilience and survivability in communication networks: Strategies, principles, and survey of disciplines. *Computer Networks*, *54*(8), 1245–1265, https://doi.org/https://doi.org/10.1016/j.comnet.2010.03.005.

[50] Sverdlik, Y. (2017). AWS Outage that Broke the Internet Caused by Mistyped Command https://www.datacenterknowledge.com/outages/aws-outage-that-broke-the-internet-caused-by-mistyped-command.

[51] Tizghadam, A. & Leon-Garcia, A. (2010). Autonomic traffic engineering for network robustness. *Selected Areas in Communications, IEEE Journal on*, *28*, 39–50, https://doi.org/10.1109/JSAC.2010.100105.

[52] Tomarchio, O., Calcaterra, D., & Modica, G. (2020). Cloud Resource orchestration in the multi-cloud landscape: a systematic review of existing frameworks. *Journal of Cloud Computing*, *9*(49), https://doi.org/10.1186/s13677-020-00194-7.

[53] Wang, X., Pournaras, E., Kooij, R. E., & Van Mieghem, P. (2014). Improving robustness of complex networks via the effective graph resistance. *The European Physical Journal B*, *87*(9), https://doi.org/10.1140/epjb/e2014-50276-0.

[54] Welsh, T. & Benkhelifa, E. (2020). On Resilience in Cloud Computing: A Survey of Techniques across the Cloud Domain. *ACM Comput. Surv.*, *53*(3), https://doi.org/10.1145/3388922.

[55] Welsh, T. & Benkhelifa, E. (2021). The Resilient Edge: Evaluating Graph-based Metrics for Decentralised Service Delivery. In *2021 Sixth International Conference on Fog and Mobile Edge Computing (FMEC)* (pp. 1–7).

[56] Yadav, S. (2025). multi-cluster-istio-kind https://github.com/sedflix/multi-cluster-istio-kind/tree/618d6877598f7e81379e211c181490f6f9cf336e.

[57] Yeboah-Ofori, A., Jafar, A., Abisogun, T., Hilton, I., Oseni, W., & Musa, A. (2024). Data Security and Governance in Multi-Cloud Computing Environment. In *2024 11th International Conference on Future Internet of Things and Cloud (FiCloud)* (pp. 215–222).

[58] Youseff, L., Butrico, M., & Da Silva, D. (2008). Toward a Unified Ontology of Cloud Computing. In *2008 Grid Computing Environments Workshop* (pp. 1–10).

[59] Yu, H., Wang, X., Xing, C., Xu, B., & Dalle, J. (2022). A Microservice Resilience Deployment Mechanism Based on Diversity. *Sec. and Commun. Netw.*, *2022*, https://doi.org/10.1155/2022/7146716.