UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Thomas Palts

Kubernetes Custom Resources and Controllers for Managing Game Servers

Bachelor's Thesis (9 ECTS)

Supervisor:
Pelle Jakovits, PhD

Tartu 2025

Kubernetes Custom Resources and Controllers for Managing Game Servers

Abstract:

Modern multiplayer servers in Kubernetes face many challenges, including preventing deletion until the state is saved. This thesis presents the design and implementation of a custom Kubernetes operator for managing multiplayer game server lifecycles. The thesis introduces four custom resources—`Server`, `Fleet`, `GameType`, and `GameAutoscaler` to enable declarative control over server scaling and termination. A core focus is ensuring predictable and graceful shutdowns, with deep integration between Kubernetes resource management and in-game server logic, achieved through a sidecar component and RESTful interfaces. The system supports advanced orchestration features, including rolling updates, dynamic scaling via external signals, and finalizer-based deletion protection. Validation confirmed that the operator fulfills all design goals, showing reliability, responsiveness, and no significant performance overhead.

Keywords: Kubernetes, game server, custom resource, operator, controller

CERCS:

P170 Computer science, numerical analysis, systems, control

P175 Informatics, systems theory

Kubernetese kohandatud ressursid ja kontrollerid mänguserverite haldamiseks

Lühikokkuvõte:

Kubernetesel jooksvad mänguserverid seisavad silmitsi paljude väljakutsetega, millest üks on kustutamise takistamine kuni olek on salvestatud. See lõputöö tegeleb kohandatud Kubernetese operaatori disaini ja rakendamisega mänguserverite elutsüklite haldamiseks. Lõputöö tutvustab nelja kohandatud ressurssi: `Server`, `Fleet`, `GameType` ja `GameAutoscaler`, et võimaldada deklaratiivset kontrolli serveri skaleerimise ja sulgemise üle. Põhirõhk on sujuvate sulgemiste tagamisel, pakkudes integratsiooni Kubernetese ressursihalduse ja mängusisese serveri loogika vahel, mis saavutatakse Kubernetese kõrvalkäru ja RESTful liideste kaudu. Süsteem toetab täiustatud orkestreerimisfunktsioone, sealhulgas jooksvaid värskendusi, skaleerimist väliste signaalide kaudu ja finaliseerijal põhinevat kustutuskaitset. Valideerimine kinnitas, et operaator täidab kõiki disainieesmärke, näidates üles töökindlust, reageerimisvõimet ja olulist jõudluskulu puudumist.

Võtmesõnad: Kubernetes, mänguserver, kohandatud resurss, operaator, kontroller

CERCS:
P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)
P175 Informaatika, süsteemiteooria

Contents

1. Introduction

Modern multiplayer games come with many technical backend challenges, often invisible to players. These challenges include scalability, cost, state management, and latency mitigation. N. Kasenides and N. Paspallis [1] identify one of the most critical technical hurdles in massively multiplayer online games: synchronizing the state across a distributed system. Infrastructure changes—such as auto-scaling or restarts—make maintaining that state even more complex, especially on platforms that use containers and orchestration tools.

Kubernetes is a container orchestration platform widely adopted for scalability and automation. By design, it presumes all applications are stateless.

This thesis investigates the lifecycle management of multiplayer game servers in Kubernetes. Game servers inherently manage state, and players must be disconnected or safely transferred before the server shuts down.

Kubernetes's default behaviour can terminate servers during scaling or rescheduling events, risking data loss, corrupted states, and a degraded experience.

Existing tools, like Agones, provide feature-rich solutions for running game servers in Kubernetes. However, these tools can introduce considerable complexity and integration challenges. Additionally, many current solutions lack robust mechanisms to prevent premature server termination.

This thesis aims to design and implement a custom Kubernetes operator that enables declarative, lifecycle-aware management of multiplayer game servers. As part of the thesis, the project defines Custom Resource Definitions (CRDs) for game servers and implements an operator to enforce the lifecycle configurations. The operator also introduces mechanisms for webhook-based scaling and communication between game server instances and the operator using sidecars.

The **central problem this work addresses is** how to ensure stateful game server workloads are managed safely and predictably within Kubernetes, without introducing unnecessary complexity, while providing all the features for termination blocking. The key components of this solution include:

- A CRD definition for game servers, encapsulating key metadata and lifecycle rules.
- A controller responsible for reconciling the state of game servers based on defined policies.

- Mechanisms for handling autoscaling, ensuring that servers scale up or down based on metrics.
- Sidecar for communication between the game server pods and the controller.

This thesis does not cover broader aspects of multiplayer backend infrastructure, such as matchmaking, player authentication, or analytics. While autoscaling is supported, it is implemented via a REST API endpoint to allow custom behavior, acknowledging the diversity of requirements in multiplayer game development.

This thesis is organized into several main chapters, each covering important topics:

- Chapter 2 gives technical background on the tools and technologies used, related work, and reasons for their selection.
- Chapter 3 describes the design goals, system architecture, and development requirements.
- Chapter 4 covers the implementation, including controller logic, custom resources, and communication methods.
- Chapter 5 discusses validation steps and suggests possible future improvements.

The appendix has essential definitions for quick reference to complicated terms and definitions.

2. Background

This chapter presents a comprehensive overview of the core technologies, foundational concepts, and relevant prior work underpinning this thesis's research. It explains the technical landscape of the project and justifies the choice of specific tools and methodologies.

2.1 Containers

According to Dockers website[2]: A container packages code and all its dependencies into a single unit, allowing it to run quickly and reliably across different environments.

Containers are packages of software that include all necessary dependencies, making it easy to run applications consistently across different environments. At a high level, containers can seem similar to virtual machines. However, as Amro Abuabdo and Ziad A. Al-Sharif explain[3], they fall into different categories of virtualization. Containers share the host operating system's kernel and isolate applications at the process level, whereas virtual machines emulate entire operating systems alongside the host OS.

The primary goal of Docker containers is to eliminate environment-related issues (the classic "works on my machine" problem) and support the development and deployment of applications using the microservices architecture. The container image is a key component enabling this.

2.1.1 Images

According to Amazon Web Services (AWS)[4], Docker images are templates with instructions for creating a container. An image is a blueprint of the libraries and dependencies required inside the container for an application to run. Images are typically defined using Dockerfiles describing the base image, working directory, dependencies, and startup commands. A basic example can be seen in code 1.

```dockerfile
# Use a base image
FROM python:3.9

# Set the working directory
WORKDIR /app

# Copy application code
COPY . /app

# Install dependencies
RUN pip install -r requirements.txt

# Run the application on container startup
CMD ["python", "app.py"]
```

Code 1. Example Dockerfile for a Python application

Unlike virtual machines, containers leverage the host operating system and kernel features to isolate and run applications. The key technologies in the Linux kernel that enable containers are[5]: cgroups and namespaces.

2.1.2 Cgroups (control groups)

CGroups or control groups limit and isolate different resource usages (CPU, memory) for containers. According to J. Bottomley and P. Emelyanov[6] cgroups can be simplified as resource controllers and limiters on particular resource types. Commonly used cgroups are:

- **blkio** - controls block devices
- **cpu and cpuacct** - controls cpu resources
- **cpuset** - Controls CPU affinity for a process group
- **memory** - Controls memory allocations

Simply put, cgroups limit the specific resource usage of a container. For example, you can set it so the container can only use `10%` of the CPU.

2.1.3 Namespaces

Namespaces provide process-level isolation by restricting what a container can see and access on the host system. J. Bottomley's and P. Emelyanov's article[6] says that in the Linux kernel, there are six main namespace settings:

- Network: For tagging a network interface
- PID: Does a subtree from the fork, which remaps the visible PID (process ID) to 1 so that init can work correctly.
- IPC: Separates the system IPC (Inter-Process Communication) namespace on a per-container basis.
- Mount: Makes each container have a separate file-system root.
- User: Remaps UIDs between the host and the container.

Namespaces isolate resources and prevent access. Thanks to these namespaces, the container has a separate hostname, processes, and so on (when looking from inside the container). Namespaces provide a layer of security that prevents any malicious code in the container from causing too many issues.

2.2 Kubernetes

The rise of containers and the rapid growth of cloud computing led to the need for efficiently distributing workloads across many machines. While containers ensured consistent application environments, managing them at scale quickly became a complex challenge.

According to AWS[7], early solutions relied on purpose-built scripts, which soon proved inefficient and difficult to scale. These issues led to the development of container orchestration platforms, with Kubernetes[1] emerging as the industry standard. It offers flexibility, reliability, and a strong community ecosystem.

Kubernetes is fundamentally resource-based. It organizes its functionality around resources, which can be built-in or custom-defined. Developers can implement new behaviors through resource definitions and controllers, making Kubernetes highly modular and extensible.

[1] https://kubernetes.io/

2.2.1 Resources

According to the Kubebuilder documentation[8], a Kubernetes resource is an API with a well-defined schema structure and corresponding endpoints[2]. Because the structure is predictable, most tools work with the API, even if not part of the base Kubernetes.

Some of the most important built-in resources are:

- **Pods**[9] are the smallest deployable units in Kubernetes and represent a single running process in a cluster[3]. Typically, a Pod contains a single container, though it can include multiple tightly coupled containers and share resources. A typical example of this is the sidecar design pattern.
- **Deployments**[9] are higher-level resources that manage the updating of applications[4]. They allow applications to scale horizontally and perform rolling updates without downtime, automating many aspects of application management in Kubernetes. They scale by creating and updating replica sets.
- **ReplicaSet**[9] are resources that manage replicas of pods[5]. In their specifications, you define how many replicas should exist, and the replicasets manage making that amount of (healthy) pods for you.

To make it easier to understand, figure 1 describes the Kubernetes resources.

---

[2] https://book-v1.book.kubebuilder.io/basics/what_is_a_resource
[3] https://kubernetes.io/docs/concepts/workloads/pods/
[4] https://kubernetes.io/docs/concepts/workloads/controllers/deployment/
[5] https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/

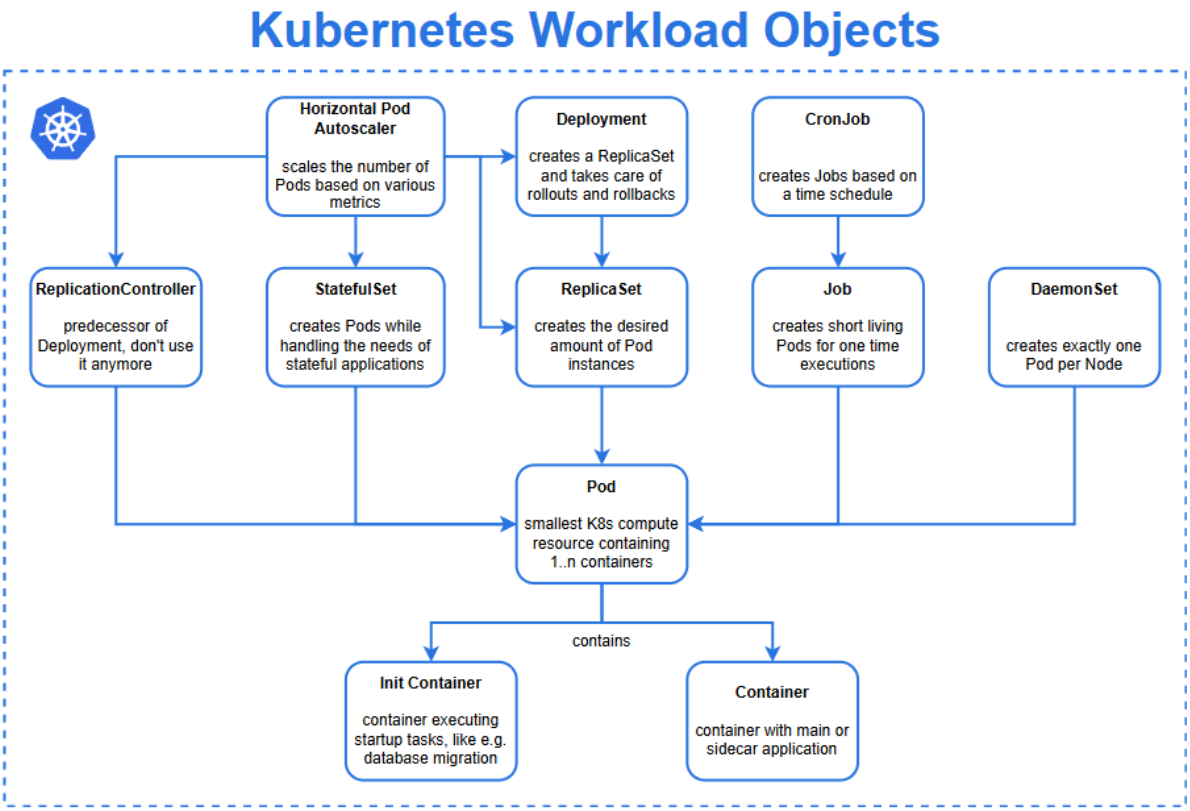

Figure 1. Kubernetes Workload Resources[10]

2.2.2 Webhooks

Kubernetes has the option to define custom admission webhooks[9]. Admission webhooks are HTTP endpoints that receive admission requests and do something with them[6]. In Kubernetes, you can create two types of webhooks: validating admission webhooks and mutating admission webhooks (sometimes called a defaulting webhook).

Kubernetes calls mutating admission (defaulting) webhooks first. They can modify the object created with custom defaults. These are useful for setting some complex logic for defaulting fields. For example, the operator checks if the timeout field is empty; if so, it defaults to 40 minutes there.

The admission webhook is called after the mutating one. The admission webhook is where you can define any logic to validate if the specs are valid, and return errors and warnings if needed.

[6] https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/

For example, the system checks if the *Server*'s pod spec has any containers defined; if not, it prevents the *Server* from being created.

2.2.3 Finalizers

Every resource in Kubernetes has metadata such as name, namespace, and labels. One of the more interesting metadata is finalizers. Finalizers are an array of strings that allow resource deletion to be triggered but prevent the resource from being deleted until the finalizers are removed.

Resource controllers often use these to handle any cleanup logic before deleting the resource. For example, many custom resources create underlying pods or deployments. These resources would have finalizers that delete the resources as needed.

As per the Kubernetes documentation[9], finalizers are an array of keys that tell Kubernetes to wait until conditions are met before fully marking a resource for deletion. They are used by controllers to clean up resources before deletion.

2.2.4 Control Loop

The control loop concept within Kubernetes is complex and nuanced. Fundamentally, it is a continuously executing loop that monitors the cluster's present state and evaluates its accordance with the desired state, trying to align it more closely with the desired state[9]. The core logical operations of every controller are executed within the control loop[7]. The **desired state** of the cluster represents the cluster configuration sought by the administrator, often articulated through manifest files in YAML format or through specifications. These documents specify various aspects of the desired state, including, but not limited to, names and replica counts. The **current state**, referred to at times as the observed state, is the actual state of the cluster that is stored within the "Status" fields of resources and is subject to updates by the controllers.

2.2.5 KubeCTL

According to the official documentation, kubectl is a command-line tool used for communicating with the Kubernetes cluster[9]. Every kubectl command is in this format[8]: kubectl [command] [type] [name] [flags].

---

[7] https://kubernetes.io/docs/concepts/architecture/controller/

[8] https://kubernetes.io/docs/reference/kubectl/

Command specifies the operation you wish to perform on a resource. Some commonly used examples are create, get, and delete. Type specifies the resource type, such as pod, `Server`, or deployment. Name specifies the name of the specific resource.

Finally, flags are optional but can be used to change behaviour. For example, specifying the Kubernetes namespace will look for resources only in that namespace. For example, a commonly used command is: `kubectl get pods -n prod`, which returns a list of pods in the *prod* namespace. A more resource-specific command would be: `kubectl describe pod example-pod -n prod`, which would return the specific information about a pod in the *prod* namespace.

Most kubectl command usages consist of creating resources, deleting them, and retrieving them to ensure they were made correctly. Sometimes, describe is also used when debugging resources. There are other commands, but in the context of this project, they are not very useful.

2.2.6 Helm

Originally designed as a simple utility, Helm has grown into Kubernetes's most commonly used package manager, with most clusters having it set up. According to Helm's website[11], it helps you define, install, and upgrade complex Kubernetes applications.

Helm uses their templating engine, built on top of the Go templates[9] library. In practice, this means that you have a values.yaml file, full of customizable values, which is then passed into the manifests. This system means that when installing a chart, you only have to edit values in that one file, instead of going through all of them.

2.3 Sidecar Pattern

As indicated by the Kubernetes documentation[9], sidecar containers are auxiliary containers that operate alongside the main application container within a Pod. They enhance the primary app container's functionality by offering extra services like logging, monitoring, security, or data synchronization, without modifying the main application code.

These sidecar containers can encompass a range of services from database systems to simple REST APIs. Within Kubernetes, sidecar containers may be implemented manually or, starting from Kubernetes version 1.29, incorporated as initialization containers, with the specific

[9] https://pkg.go.dev/text/template

configuration being dependent upon the intended use case. This architectural pattern in Kubernetes is significantly influenced by a similar pattern in microservices. As per the Azure documentation[12], the sidecar pattern is named as such because it resembles a sidecar attached to a motorcycle. The sidecar is attached to an application and provides supporting features. The sidecar shares its lifecycle and is retired alongside the application. The pattern is sometimes referred to as the sidekick pattern.

The underlying concept is maintaining a container that shares the lifecycle with the primary application, providing supplementary features to the principal application container. To help understand the relationship between pods, containers, and sidecars, the figure 2 is provided.

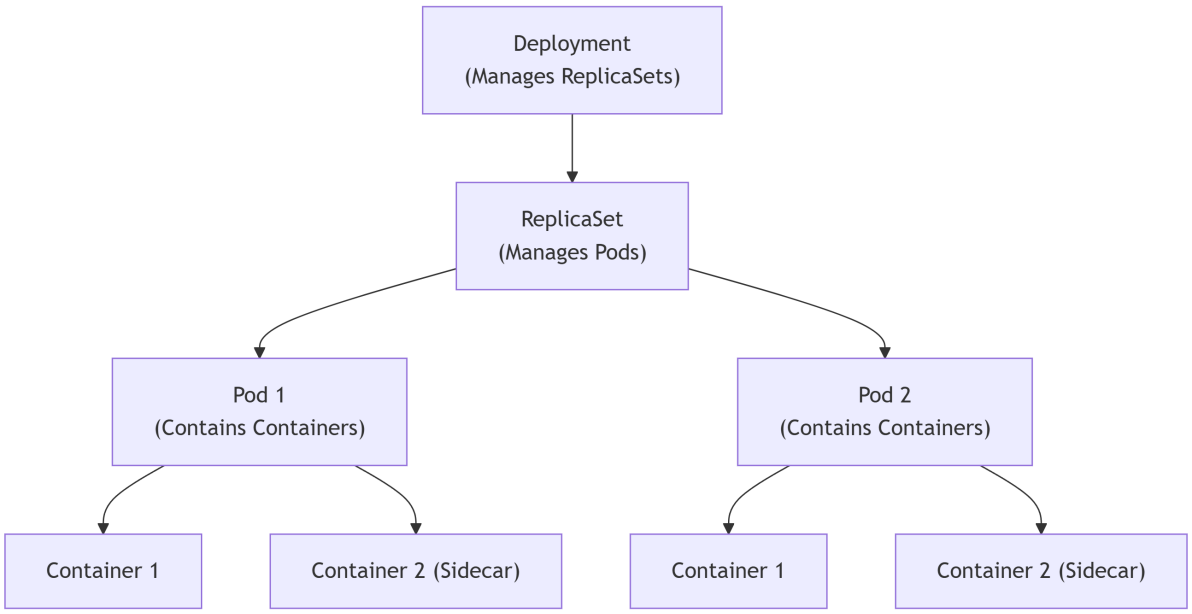

Figure 2. Kubernetes Deployments, Replicasets, Pods, Containers.

A good example of using the sidecar pattern, which is relatively easy to understand, would be logging. There are a few solutions to collect logging information from many pods, including injecting sidecars into the pods. This would usually mean that the application writes the data to some file, for example, app.log. The sidecar logging then detects any changes to that file and

sends them to a central logging server, often Elastic or Loki. An example of this is the Vector[10] sidecar pattern[11], where next to every pod that needs reporting, a secondary pod running software that collects logs and sends them to a central database is running.

2.4 Custom Resources

One of Kubernetes' most powerful features is its extensibility through custom resources. In addition to its predefined resource types, such as Pods, Deployments, and Services, Kubernetes allows developers to define new types of resources using Custom Resource Definitions (CRDs). Once a CRD is registered, Kubernetes users can create and manage custom resources in the same declarative way as built-in types.

There are many types of CRDs, but a paradigm that has become quite common is databases. A lot of database setups in Kubernetes tend to be quite complex. Thus, many operators and CRDs have been born to handle such issues. For example, the MongoDB Community Operator[12], where you can create a MongoDB resource with a single resource, instead of having to spend time configuring it. You can see an example MongoDB replica deployment in code 2.

```yaml
apiVersion: mongodbcommunity.mongodb.com/v1
kind: MongoDBCommunity
metadata:
  name: example-mongodb
spec:
  members: 3
  type: ReplicaSet
  version: "4.2.7"
```

Code 2. Simple MongoDB replica set

2.4.1 Controllers

Custom resources are passive; they are stored in the Kubernetes database by the Kubernetes API (etcd[13]) but do not trigger any behavior. A custom controller is required to make them functional.

[10]Tool for building observability pipelines

[11] https://vector.dev/docs/setup/deployment/roles/

[12] https://github.com/mongodb/mongodb-kubernetes-operator

[13] https://kubernetes.io/docs/concepts/overview/components/

Controllers[9] are pieces of software that watch for changes to custom (or built-in) resources and act accordingly to reconcile the desired and actual state of the system. They follow the Operator pattern[9], wherein a control loop continually observes the cluster state and applies changes as necessary. An example could be the previously discussed MongoDB operator, but another commonly used example is different monitoring solutions. Of these, the most common is Prometheus and its operator[14]. It has many resources, such as *PodMonitor*, for which you can specify how groups of pods are monitored and what port and path to scrape them. The controller then generates the required Prometheus scrape configurations automatically.

This means a controller function is triggered whenever a resource is created, updated, or deleted. The controller handles these events and updates the system to reflect the desired state described in the resource specification[9].

It is worth noting that the distinction between a **controller** and an **operator** is somewhat fluid and varies across documentation. Generally, an operator is seen as an application containing one or more controllers, webhooks and other business logic.

According to the Operator SDK documentation[15], operators are categorized into five capability levels:

- **Level 1 – Basic Install:** Automated installation and basic configuration
- **Level 2 – Seamless Upgrades:** Support for patching and minor upgrades
- **Level 3 – Full Lifecycle:** Management of application lifecycle, including storage and recovery
- **Level 4 – Deep Insights:** Collection of metrics, alerting, and analysis
- **Level 5 – Auto Pilot:** Autonomous scaling, tuning, anomaly detection, and scheduling optimization

[14] https://github.com/prometheus-operator/prometheus-operator
[15] https://sdk.operatorframework.io/docs/

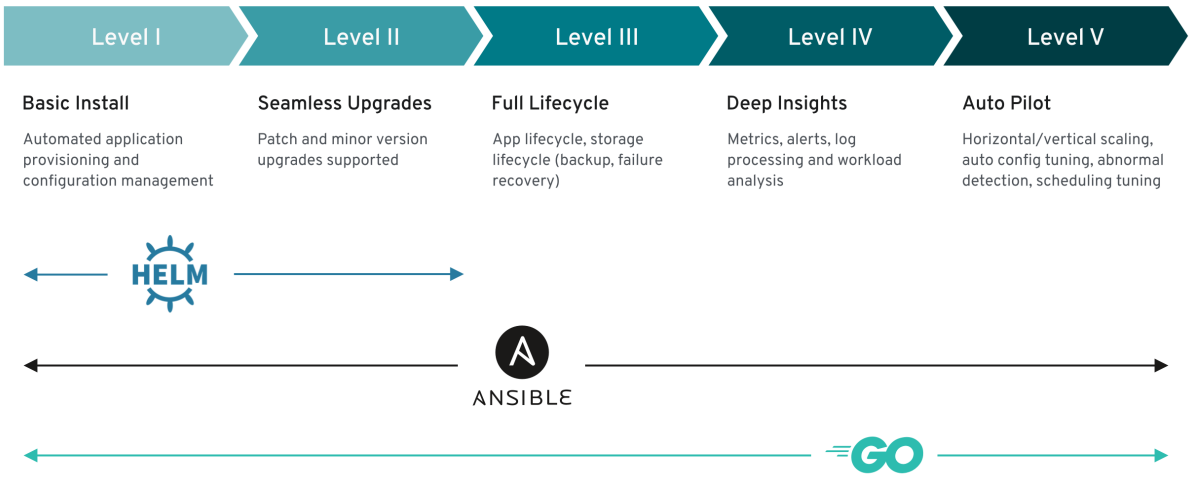

Figure 3. Operator capability levels according to the Operator SDK[13].

In reality, many controllers span multiple capability levels—for example, a controller may implement autoscaling (Level 5) but omit backup or recovery features (Level 3).

Operators can be built using different toolchains, the most common of which are:

- **Helm**
- **Ansible**
- **Go (or other general-purpose languages)**

The choice of toolchain depends on the complexity of the operator and the development team's familiarity with the tooling. Helm and Ansible are suitable for simple automation tasks, whereas Go offers much finer control and scalability for building robust, fully featured operators.

As noted by Ruxia Duan, Fan Zhang, and Samee U. Khan [14]:

> "An Operator that possesses the core functions at a level is said to have reached that particular maturity level. Operators developed using the Go programming language are capable of reaching all five levels."

Controllers exist for a wide range of use cases. For example, some manage database clusters within Kubernetes, while others—as in the case of the hybrid cloud autoscaler developed by Yuyang Wang, Fan Zhang, and Samee U. Khan [15]—handle complex multi-cloud scaling strategies based on service metrics.

Several frameworks exist to aid in the development of custom controllers. Among the most prominent are Kubebuilder and the Operator SDK.

According to the Kubebuilder Github[16], kubebuilder is a tool for building Kubernetes APIs using custom resource definitions (CRDs). Like web development frameworks, it increases speed and reduces complexity for rapidly building and publishing Kubernetes APIs. Kubebuilder streamlines the use of the controller-runtime library by generating scaffolding code, configuration files, and helper utilities, significantly improving the developer experience.

The Operator SDK, on the other hand, presents itself as a more generalized toolkit for building Kubernetes-native applications. According to its website[17], the SDK makes building Kubernetes native applications easier, as it usually requires deeper and application-specific knowledge. However, since version `v1.0`[18], the Operator SDK internally uses Kubebuilder for its Go-based operators. Its added value lies in its support for Helm- and Ansible-based operators—areas that are not relevant to the scope of this thesis.

2.5 GoLang

The Go programming language (often called GoLang) was conceived in late 2007, as an answer to some of the problems that the developers encountered at Google, who were developing software infrastructure[19]. Go prides itself on its simplicity and inherent support for concurrent workloads. Concurrency has been a consideration for Go since day one, which means the systems that support it are very thought out.

Go is especially relevant in cloud and Kubernetes, as it provides the speed of C-type languages, which is very useful in the cloud, while remaining simple. Its ability to address the challenges of building modern, high-performance applications makes it a good choice for many applications, especially cloud-based ones.

---

[16] https://github.com/kubernetes-sigs/kubebuilder
[17] https://sdk.operatorframework.io/
[18] https://www.redhat.com/en/blog/operator-sdk-reaches-v1.0
[19] https://go.dev/talks/2012/splash.article

One of the most well-known applications built in GoLang is Kubernetes[20]. This language choice has caused a sort of wave effect, where all the Kubernetes-adjacent software for monitoring[21], certificates[22], and more are also built in Go.

Building things such as custom controllers tends to be easier in GoLang than in other languages, as tools and libraries such as Kubebuilder already exist. Of course, that does not mean it is impossible in different languages. Kubernetes inherently exposes REST APIs to interact with the cluster, meaning things would still be possible, even if harder, in other languages.

2.5.1 Reasons to Use Go

According to Go's website[23]: Go is an open-source programming language supported by Google, which is easy to learn and thus great for teams with built-in concurrency and a robust standard library, and that has a large ecosystem of tools, partners, and communities.

The same website has a lot of case studies of different known and lesser-known companies using Go in unique ways and explaining why they made that decision. But, for most companies, the following quote from a Netflix blog post about caching sums it up quite well[16]:

> ”The decision to use Go was deliberate, because we needed something that had lower latency than Java (where garbage collection pauses are an issue) and is more productive for developers than C, while also handling tens of thousands of client connections. Go fits this space well.”

Another part of GoLang that developers tend to appreciate quite a bit is the dependency management, instead of the developer having to use a dependency-management framework like Maven, Gradle, NPM, or something else. Go makes it simple: all projects are shared via their source code repositories (often Github)[24]. This removes the ”invisible magic” that dependency management in other languages usually tends to become. While it can be more cumbersome on large-scale projects, it is generally easier to follow.

---

[20] https://github.com/kubernetes/kubernetes
[21] https://github.com/prometheus/prometheus
[22] https://github.com/cert-manager/cert-manager
[23] https://go.dev/
[24] https://go.dev/doc/modules/managing-dependencies

```
        type Student struct {
        FirstName     *string `json:"firstName"`
        LastName      *string `json:"lastName"`
        Email         *string `json:"email"`
        StudentNumber *string `json:"studentNumber"`
}
func SomeRandomHandler(a *app.App) func(http.ResponseWriter,
↪  *http.Request) {
        return func(w http.ResponseWriter, r *http.Request) {
                w.Header().Set("Content-Type", "application/json")
                var student Student
                err := json.NewDecoder(r.Body).Decode(&student)
                if err != nil {
                        a.Logger.Debug("Error decoding student for
                        ↪  creation", "error", err)
                        w.WriteHeader(http.StatusBadRequest)
                        return
                }
                a.Database.CreateStudent(student)
                json.NewEncoder(w).Encode(student)
        }
}
```

Code 3. Go: Basic struct, JSON tags and conversions

Unlike a large portion of modern languages, Go is not object-oriented. Instead, it uses the concept of struct(ures), similarly to C. These contain some data. A handy feature of these is that you can directly tell Go how to encode the structure into JSON with tags. Code 3 is a simple example, with some error handling removed.

In the same code example (code 3), you can see two features that many people like about GoLang: easy conversion to and from JSON and error handling.

For JSON conversion, the desired struct type, `Student`, is specified, and the provided body, an array of bytes, is decoded into the object. The object can then be processed and re-encoded into the writer to return a JSON response to the request.

Finally, another key feature is error handling. In Go, all errors are values. This means you handle them like any other value, do some if checks, etc. Errors being passed as values also means that when a function returns an error, it is self-documenting, as the fact that it can return an error is in the method signature.

Additionally, Go has a well-written and wide standard library, which can be explored on Go's package website[25]. It is extensive but also known for its simplicity and consistency across platforms. It covers everything from I/O and networking to cryptography and web servers, often making third-party libraries unnecessary for many applications.

Here are some of the most useful packages:

- *log and slog* - Both are in-built packages that enable logging with different levels, with slog also enabling structured logging.
- *os* - Provides a platform-independent interface to operating system functionality such as file and process management.
- *context* - Manages deadlines, cancellations, and request-scoped values across API boundaries and between processes.
- *crypto* - Encompasses a collection of cryptographic packages supporting encryption, decryption, hashing, and more.
- *testing* - Provides basic support for automated testing of Go packages, including unit tests and benchmarks.
- *net/http* - Provides all the necessary tools to build REST interfaces without extra libraries.
- *encoding/json* – Offers robust tools for working with JSON data, often used in web APIs and configuration handling.
- *sync and sync/atomic* - Allow low-level primitives for safe concurrent programming.

These standard packages highlight Go's philosophy of providing very powerful tools in a minimal and easily understandable way.

[25] https://pkg.go.dev/std

2.6 Game Server Workloads

Game server workloads present unique challenges when deployed on Kubernetes, primarily due to their real-time, stateful, and long-lived nature. Unlike stateless microservices, game servers often maintain persistent connections with clients, manage an in-memory state that cannot be easily replicated, and require low latency and high availability.

One major challenge is **lifecycle management**: game servers must be gracefully shut down or drained to avoid disconnecting players unexpectedly. Lifecycle management requires custom logic around Kubernetes' pod termination and readiness mechanisms.

Scalability is another concern. Autoscaling game servers is non-trivial, especially for session-based games, where servers must be available before players join but cannot be scaled down until all players have left. Traditional CPU/memory-based auto-scalers are insufficient; more advanced systems must consider player count, session state, and match availability.

Adapting game server workloads to Kubernetes requires custom controllers, lifecycle hooks, and deep integration with infrastructure and game logic.

Note that from now on, game server is a typical random multiplayer game server, while `Server` is the custom resource implemented in this thesis.

2.6.1 Stateful Connections

Game servers typically maintain stateful connections with clients to support real-time gameplay. These connections, often using TCP or UDP, are sensitive to disruptions such as pod restarts, rescheduling, or network interruptions. Unlike stateless services, game servers cannot easily fail over or resume a session on another pod without impacting user experience. This statefulness also complicates load balancing and service discovery, as traditional Kubernetes approaches like round-robin routing or service IP abstraction are ill-suited for directing players to a specific server instance hosting their session. As a result, game server orchestration must incorporate custom mechanisms for session tracking and controlled shutdowns to preserve game state and player connectivity.

So, to put it simply, modern game server workloads require stateful connections, meaning that you cannot just delete a game server. You need some way to block deletion, and an interface for the game server to allow the server to be deleted. This is discussed more in chapters 3.1 and 3.2.

2.6.2 Utility in the Game Industry

According to J. Lundgren[17] there is no good explanation for why Kubernetes is not used in the gaming industry the same way as it is in other computer science-related industries. The most reasonable explanation seems to be that it is a newer technology, only gaining traction in 2017. Game development takes many years, and thus, Kubernetes has not yet had time to start being used on a large scale.

Of course, studying who uses what backend technology can be complicated if the organizations themselves do not report it. Since that study came out in 2021, for example, an article[18] from the studio behind Fall Guys came out, which says they use Azure's managed Kubernetes.

A part of the reason that the larger game studios do not use Kubernetes, could also be the same one as discussed in this thesis, that it, by itself, is not compatible with some types of games.

2.7 Previous Solutions

Several existing solutions have attempted to address the problem of orchestrating game servers on Kubernetes.

2.7.1 Agones

Agones[19] is currently the most widely adopted and feature-rich platform in this space. According to their website:

> ”An open source, batteries-included, multiplayer dedicated game server scaling and orchestration platform that can run anywhere Kubernetes can run.”

In Agones, three resources are important in our context:

- Server - Has a 1-to-1 relationship with a pod, creates a pod with the specified spec, and handles allocations and port openings.
- Fleet - Creates a bunch of underlying Server objects to handle the creation of multiple servers with the same spec at the same time.
- FleetAutoscaler - Sends a request periodically based on a defined period to decide whether to scale the fleet.

Despite its popularity and robustness, Agones presents a few drawbacks for managing game servers:

- **Feature bloat** – While extensive functionality can be beneficial, Agones includes many features that are often better handled through game-specific logic. For example, built-in player tracking may not align with every developer's preferred architecture.
- **Matchmaking-centric design** – Agones emphasizes matchmaking workflows, highly relevant to session-based games like *CS:GO*. However, this is less suitable for open-world or MMORPG-style games such as *Baldur's Gate*, where matchmaking is either minimal or nonexistent.
- **Session-based logic** - Agones is built assuming all servers are temporary and session-based. However, there are non-temporary servers, such as MMORPG/open world style games.
- **Automatic port exposure** – By default, Agones opens a network port for each game server on the host machine. While useful in some scenarios, this behavior can conflict with architectures that prefer routing players through a proxy or load balancer before connecting to the appropriate server instance.
- **Hidden complexity in alpha features** – Many practical features are not enabled by default and must be explicitly activated during installation via Helm, which adds an extra layer of complexity.
- **"Hacky" solutions** - One of the core reasons for the thesis project was to block the deletion of servers as needed. Deletion blocking is technically doable with Agones, but would require allocating a server to a player, even if you want to stop it, even if no players are on.

In summary, while Agones is powerful and flexible, its broad scope and opinionated defaults make it less suitable for a lightweight, customizable approach. Stripping away unnecessary features often requires working around core features built into the system. This is especially true for non-session-based games, as it presumes all servers are temporary.

2.7.2 Thundernetes

Thundernetes[20] is a lesser-known alternative developed by Microsoft, offering a similar core idea: running dedicated game servers on Kubernetes. One of its notable features is the concept of *GameServerBuild*s[26], which allows grouping servers for better scaling control.

[26] https://playfab.github.io/thundernetes/gameserverbuild.html

However, Thundernetes has a significant limitation regarding autoscaling. It relies on a model where new servers are spawned based on whether existing game servers are full. While straightforward, this method lacks flexibility and does not account for more nuanced metrics like game-specific state or player demand forecasting.

In comparison, Agones offers a more flexible approach by allowing custom autoscaling logic through webhooks, a more robust and adaptable solution for managing and scaling game servers.

2.7.3 Similarities and Differences

There are some similarities and differences between the solution of this thesis and the previous solutions.

The main similarities with Agones are:

- Usage of sidecar pattern
- Fleet, Server, Autoscaler resource goals
- Webhook-based scaling

These parts of Agones seem the most useful and are relatively simplistic in their ideas.

However, the main differences are:

- No extra features - player tracking, server selection, etc
- Game resource to manage server versions
- Blocking server deletion using sidecar

This means the basic ideas behind the Agones architecture are largely present, but the precise features are kept simple, as is a better server blocking implementation. Additionally, server updating is a larger part of the considerations.

Now, when compared to Thundernetes, it is quite different. The main thing inspired by Thundernetes is server version management. However, Thundernetes uses a separate string for that, while the thesis operator checks it by checking if the pod specs are different.

While many of the ideas are taken from Agones and Thundernetes, it has unique advantages, especially when it comes to non-session-based games.

2.7.4 KEDA

KEDA (Kubernetes Event-Driven Autoscaling) is a tool that helps scale applications in Kubernetes based on real-world events[21]. With KEDA, you can adjust the scale of your applications automatically, based on the workload.

It's lightweight and works alongside existing Kubernetes components, like the Horizontal Pod Autoscaler (HPA).

In the context of this thesis, the main issue with KEDA is that it presumes all applications are stateless and can be instantly shut down. This is not ideal when dealing with game servers, as there is a need to drain the game servers slowly and then shut them down once the necessary logic is completed.

2.7.5 Example: Google Doodle

Google Doodle is the change in the Google Logo, often changing it into some form of game. In the past few years, these games have become multiplayer-focused or at least multiplayer-compatible. This, according to Google[22], uses Google Cloud and Agones underneath. Agones is used to scale the server replicas as needed, and together with OpenMatch, it is used to make the matches correctly. Most other popular games use similar cloud-based solutions.

2.7.6 Example: Fall Guys

Fall Guys, a battle royale game launched in 2020, runs on Azure[18]. It uses Azure Kubernetes Service (AKS) for primary hosting and Azure Cosmos DB as the underlying database. This works very well for them, as it enables them to scale very elastically. This is a very important requirement, as the game's player count fluctuates greatly depending on the time, updates, and current events in the world.

For example, during the Covid-19 crisis, at its peak, 25000 people logged into Fall Guys every minute, and 1000 new games were started. This all resulted in a very high need to scale servers fast, which Azure provided.

However, this type of solution does not work for all games, especially ones with MMORPG-style gameplay. For example, according to N. Kasenides and N. Paspallis[1], MMOG (Massively Multiplayer Online Games) are characterized by the requirement to synchronize and update their state rapidly. This requirement imposes many constraints and makes migration to the cloud very difficult, if not impossible.

In their article, N. Kasenides and N. Paspallis discuss a framework to make building MMOGs that leverage cloud technologies more accessible and less complicated. This framework is not directly related to this thesis but outlines that state management is an important aspect of these game servers and can be quite complex.

3. Planning and Design

This chapter outlines this project's planning and design and its thought processes.

3.1 Planning

The first step in the planning process was to evaluate existing projects in the Kubernetes game server space, particularly Agones, to determine which concepts could serve as inspiration. The main features that were identified as useful and worth replicating or adapting were:

- Webhook-based scaling for flexible, programmable scaling policies
- The use of a sidecar pattern to allow in-cluster HTTP communication from the game server to the controller
- The general structure and separation of concerns found in the Fleet, Server, and FleetAutoscaler resources

3.1.1 Requirements

Drawing from the reviewed alternatives and design motivations, a set of core requirements to guide implementation was formulated. This project is not intended for direct end-user interaction but focuses on backend automation and internal orchestration. Therefore, rather than formal user stories, the architecture is structured around four custom resources, each with well-defined responsibilities and operational constraints.

Table 1 summarizes controller responsibilities and the resources they watch. The figure helps understand the requirements.

Table 1. Responsibilities and watch targets of each controller.

Controller	Watches	Manages / Creates
Server Controller	Server, Pod	Adds finalizers, injects sidecars, manages Pods, handles timeout-based deletion
Fleet Controller	Fleet	Creates and maintains Server resources, enforces replica count, supports deletion strategies
GameType Controller	GameType, Fleet	Orchestrates active Fleet switching and handles rolling updates
Autoscaler Controller	GameTypeAutoscaler	Calls evaluation logic, adjusts GameType replica count, requeues periodically

The basic requirements are somewhat ambiguous because these were defined when the application was not fully built yet. They can be seen in table 2. There were also some less technical requirements, which can be called the non-functional requirements and can be seen in table 3.

Requirement Number	Requirement
FR1	Automatically manage game server lifecycles, including clean shutdowns and protection against premature deletion using the Server resource.
FR2	Allow game-specific logic to control when a Server is safe to delete.
FR3	Must inject a sidecar container with a standardized REST interface to a Server's pod.
FR4	Support zero-downtime rolling updates when changing game server versions. This updating mechanism means that when the version changes, new Server replicas are made, and old ones are deleted so that GameType.
FR5	Enable scaling based on external signals for GameType in GameAutoscaler.
FR6	Automatically apply Kubernetes finalizers to enforce lifecycle protection for Server and matching pods.
FR7	Automatically apply Kubernetes finalizers to enforce lifecycle protection for Fleets.
FR8	Automatically apply Kubernetes finalizers to enforce lifecycle protection for GameType.
FR9	Maintain a specified number of Server replicas as defined in the Fleet specification.
FR10	Support customizable deletion strategies (oldest-first and youngest-first) for Fleet.
FR11	Fleet should recover when manually deleting Servers.
FR12	Handles transitions between Fleets during version upgrades in GameType.
FR13	Periodically trigger a HTTP request to the webhook. The request is sent by the GameAutoscaler.
FR14	Patch GameTypes underlying fleets replica count based on returned data by the GameAutoscaler.

Table 2. Basic requirements for the application

Additionally, some basic non-functional requirements were defined to give some guidelines when building the application. However, as this thesis was already quite complex, the non-functional requirement did not include anything performance-related.

Requirement Number	**Requirement**
NFR1	The controller should have a simple configuration with minimal options to reduce complexity and potential errors.
NFR2	The controller code should be testable, enabling effective unit testing.
NFR3	The controller logic should be covered to a reasonable degree by automated unit tests.
NFR4	The system should support end-to-end tests to verify workflows in a simulated Kubernetes environment.
NFR5	The controller should validate and mutate manifests as needed to ensure correctness.
NFR6	The system should be compatible with Kubernetes versions 1.30.
NFR7	The controller should be written in GoLang using Kubebuilder.

Table 3. Non-functional requirements for the application

The figure 4 provides a simplified overview of the resource management and observation responsibilities within the system:

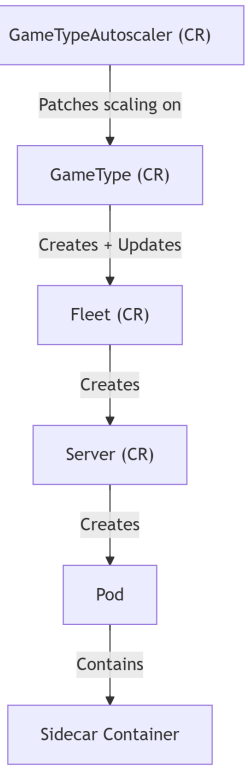

Figure 4. Management and watching of resources sequence

These requirements formed the foundation for designing the custom resources, controllers, and interactions.

3.2 Architecture

The goal proposed in chapter 1 was to design and implement a custom Kubernetes operator that enables declarative, lifecycle-aware management of multiplayer game servers. The central problem with that was how to do it predictably while providing termination blocking to game servers.

As part of this thesis, the custom operator and four matching custom resources were made to enable lifecycle management. Additionally, a service and sidecar REST interface were made. The game server can also use the sidecar interface to define if the `Server` can be deleted. This means that the lifecycle can be managed with a combination of the operator, custom resources, and REST interface calls from the game server itself. These should together cover both the goal and the problem, as they ensure safe termination using finalizers and the sidecar service, and enable lifecycle-aware management for the Servers.

The created application can be categorized into five large modules:

- `Server` and matching controller logic.
- `Fleet` and matching controller logic.
- `GameType` and matching controller logic.
- Service to interact with the custom resources more easily.
- Sidecar that runs in the `Server` pod for easy management of the shutdown and delete states.

The four resource types can be summarized like this:

- **Server** - Represents a pod with a sidecar injected. It has some additional settings related to handling the deletion process.
- **Fleet** – Represents a group of `Server` resources. It includes logic for prioritizing which Servers to delete when the desired replica count is reduced.
- **GameType** – Manages one or two `Fleets` , enabling non-disruptive updates (e.g., rolling out a new server version). It's primarily just a wrapper around the `Fleet` spec. Its example specification can be seen in the appendix in code 12.
- **GameAutoscaler** – Periodically makes an HTTP request and adjusts the replica count of the associated GameType resource based on the response. After updating, it requeues itself for reconciliation after a configured delay.

This section will explain what the created resources mean, what they do, and what they are suitable for. The precise logic on the controller side is covered in chapter 4.1.

3.2.1 Sidecar

This sidecar is a lightweight RESTful HTTP application that coordinates lifecycle events with the controller. It exposes the following four key endpoints:

- `GET /allow_delete` – Called by the controller to check whether the `Server` can currently be deleted.
- `POST /allow_delete` – Called by the game server itself to signal that deletion is safe (e.g., no players are connected).
- `GET /shutdown` – Used by the game server to check whether a shutdown has been requested.

- `POST /shutdown` – Called by the controller to initiate the shutdown process. Note that this does not mean the `Server` instantly shuts down; the `Server` has to allow the deletion process.

Each of these endpoints returns or accepts a simple boolean value encoded as JSON. This value is stored in memory within the sidecar. Although storing the state in memory introduces some fragility (e.g., loss during crashes), this trade-off is acceptable under the assumption that unexpected restarts are rare or undesirable for game servers.

3.2.2 Server

The architecture of the `Server` custom resource is the most intricate component of this controller. The core idea is that a `Server` object defines a Pod using the provided specifications, but with an additional sidecar container automatically injected.

When a new `Server` resource is created, a finalizer is automatically assigned to prevent premature deletion. A corresponding Pod is then created, with the sidecar injected, and the same finalizer is added to the pod. This setup ensures the Server can perform any necessary cleanup, such as disconnecting players or persisting state, before the final deletion occurs. Once the cleanup is complete, the game server sends a `POST /allow_delete` request to the sidecar, which allows the controller to proceed with the shutdown process.

Additionally, the controller automatically adds the following environment variables to all containers in a server to enable easier custom logic:

- `CONTAINER_IMAGE` - The container image that is read from.
- `SERVER_NAME` - The name of the server object.
- `FLEET_NAME` - If applicable, the name of the Fleet the Server belongs to.
- `GAME_NAME` - If applicable, the name of the GameType the Server belongs to.
- `POD_IP` - The IP of the pod that the container is in.
- `NODE_NAME` - The node name that the pod is running in.

In the appendix, a `Server` manifest is available for viewing in code 10.

A typical flow when a `Server` is created can be seen in figure 5. As you can see, when the controller detects a Server being made, it adds custom finalizers and creates a matching pod.

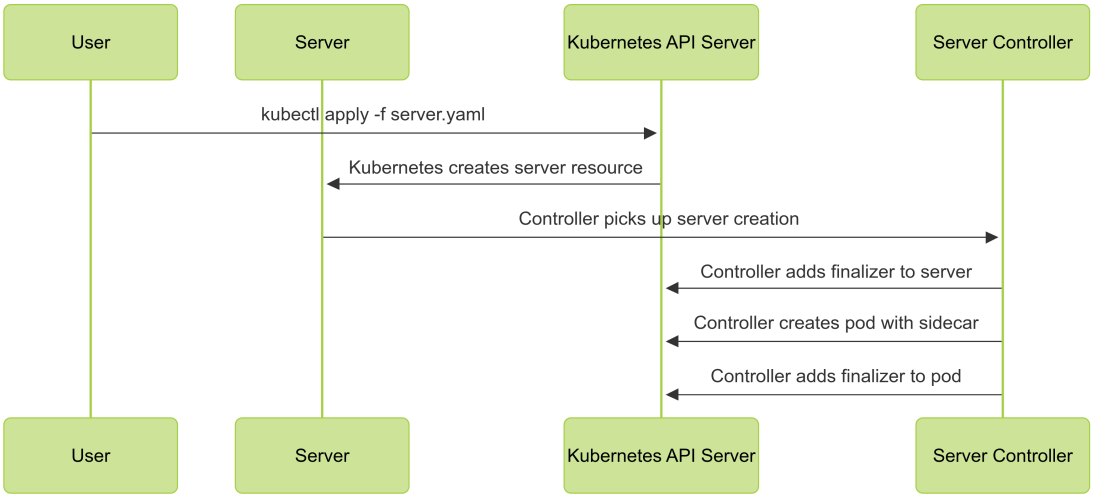

Figure 5. Server is created with pod

The main idea is for the game `Server` to implement logic to make those REST calls, allowing the server software to set deletion allowed when ready to be deleted.

3.2.3 Fleet

The `Fleet` resource is an easy way to create multiple replicas of the same `Server` specifications. A typical configuration can be seen in code 11. Simply put, it tries to maintain the replica count defined in the specification. But, to do so, it often has to delete servers. The server deletion process requires either the timeout to pass or `Server` deletion to be allowed.

A typical flow for `Fleet` would be as seen in figure 6. It is not nearly as complex as `Server`, as all the real complexity in `Fleet` is from choosing which `Server` to delete when scaling down.

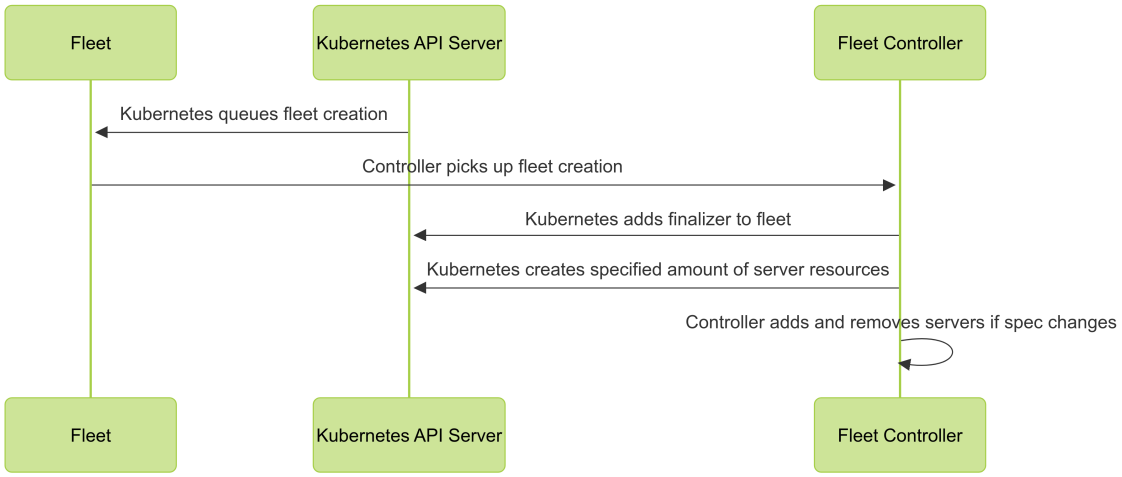

Figure 6. `Fleet` creation and scaling

3.2.4 GameType

A `GameType` is a resource that is a wrapper between `Fleets` for rolling updates. Its example specification can be seen in code 12. It registers changes to the underlying `Fleet` spec, creates a new `Fleet` with said spec, and tries to delete the old `Fleets` slowly. The multi-`Fleet` updating allows for a rolling update type effect.

The precise logic can be found in chapter 4.1.6. But the basic flow can also be seen in figure 7.

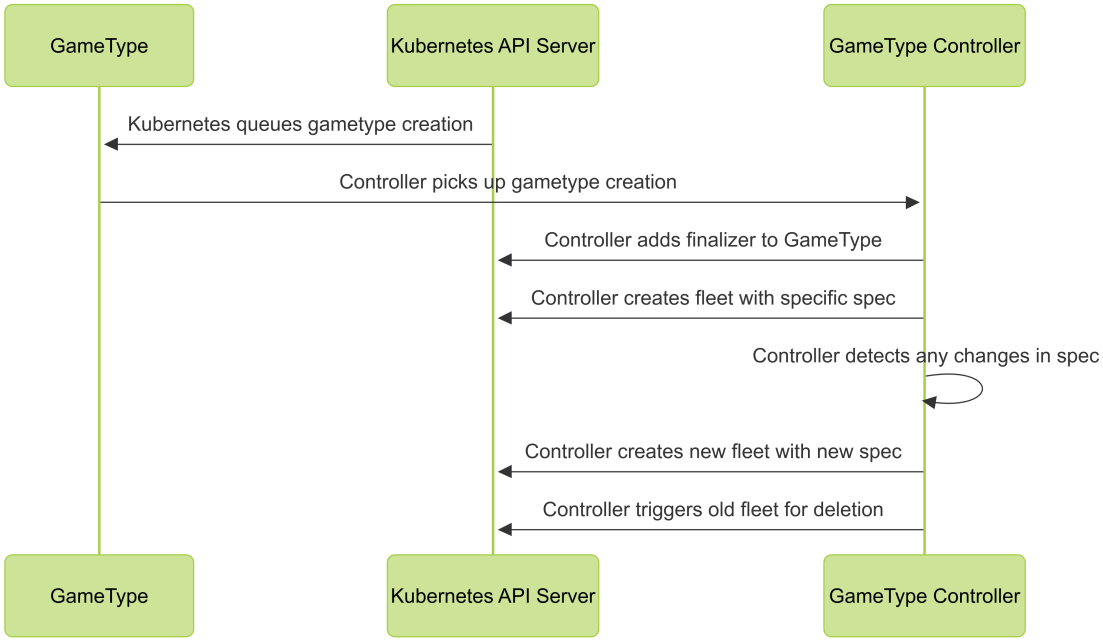

Figure 7. Gametype creation and updating

3.2.5 GameAutoscaler

The `GameAutoscaler` as a resource type is the simplest of all of them. It sends an HTTP POST request to the defined webhook periodically, based on the defined delay between the requests. Based on the response, it edits the gametype.

The `GameAutoscaler` resource requires a custom, developer-implemented service that handles the aggregation and processing of metrics. This service can be some complex logic, quite simple logic, or even an AI-predictive algorithm.

Figure 8 depicts one such implementation. But to help with understanding it:

1. The `GameAutoscaler` resource asks if the specific GameType should be scaled up. The asking is done as a POST request with a particular body. The ”Webhook” here is implemented by a developer who wants to use the controller and resources.

2. The Webhook responds with the currently desired replica count.
3. The `GameAutoscaler` resource finds the matching GameType and updates the replica count there.
4. The `GameType` in response updates the `Fleet` with the new replica count.
5. The `Fleet` in response creates or deletes some number of Servers.

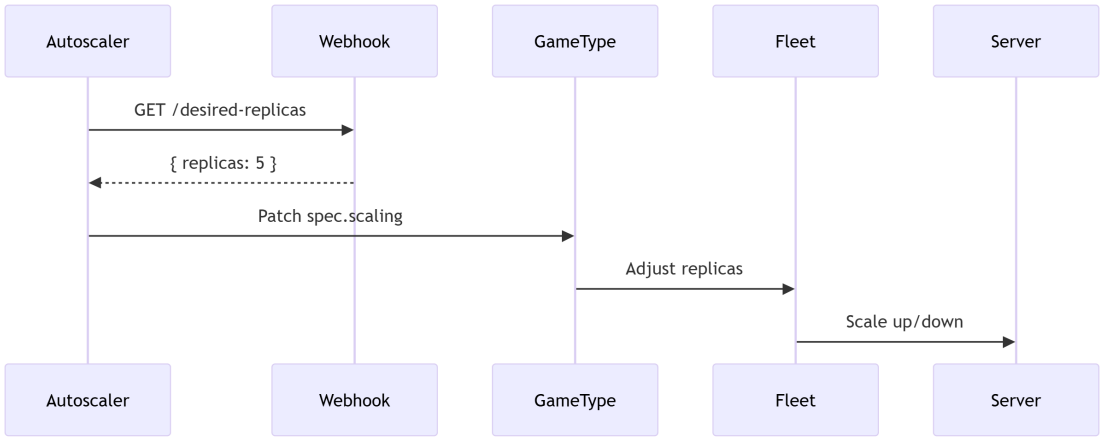

Figure 8. Autoscaling usage for GameTypes

Figure 9 is an overview of how the components interact.

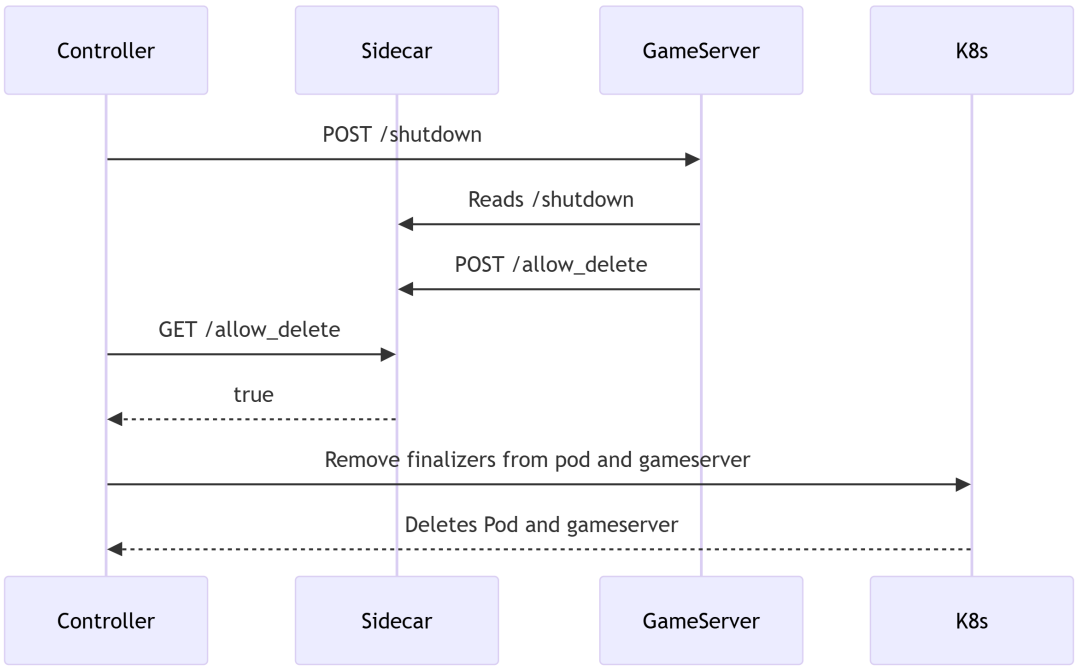

Figure 9. Lifecycle management between Server and Sidecar.

These all allow for easier management of the game servers, using REST and finalizers to block deletions.

4. Development

This chapter outlines the development details, process, testing, and DevOps information.

4.1 Implementation

Thanks to using Kubebuilder through the Operator SDK, the system implementation followed a relatively straightforward and structured process. To begin, a new Kubebuilder project was initialized. This involved downloading the Kubebuilder binary and setting it up as a CLI tool in the development environment. Once installed, the initialization process was handled through a series of CLI commands provided by Kubebuilder, such as ‘kubebuilder init‘ to initialize the project and ‘kubebuilder create api‘ to scaffold each custom resource and controller.

4.1.1 Server Implementation

The next step was implementing the custom resource types in order of importance and dependency, starting with the `Server` resource. Code 4 is an example of the ‘ServerSpec‘ definition used to represent the desired state of a `Server` resource.

```
// ServerSpec defines the desired state of Server
type ServerSpec struct {
    Pod v1.PodSpec `json:"pod,omitempty"`
    // +kubebuilder:validation:Optional
    TimeOut *metav1.Duration `json:"timeout"`
    // +kubebuilder:validation:Optional
    AllowForceDelete bool `json:"allowForceDelete,omitempty"`
}
```

Code 4. Spec struct in GoLang for Server

Each struct field is annotated with Kubebuilder markers, instructing Kubebuilder how to generate the CRD schema. For example, ‘+kubebuilder:validation:Optional‘ marks a field as optional, preventing Kubernetes from rejecting manifests that omit the field. Applying a manifest missing that field without these annotations would result in a validation error during CRD schema enforcement. A simple manifest for `Server` can be seen in the appendix, in code 10.

4.1.2 Sidecar

Another part of the `Server` implementation was implementing the initial sidecar service. The sidecar implementation was relatively simple, as GoLang has all the tools to build a simple HTTP REST server built in. It did come with defining some structs with matching JSON, though. Code 5 defines the struct for deletion, allowing requests, and code 6 defines the struct for shutdown requests. As you can see, functionally, they are very much the same JSON with a single boolean; the only difference is the key being called `allowed` or `shutdown`.

```
type DeleteRequest struct {
        Allowed bool `json:"allowed"`
}
```

Code 5. Sidecar delete allow requesting struct in GoLang for Server

```
type ShutdownRequest struct {
        Shutdown bool `json:"shutdown"`
}
```

Code 6. Sidecar shutdown requesting struct in GoLang for Server

4.1.3 Other Types

Once the `Server` type was in place, the types for `Fleet`, `GameType`, and `GameAutoscaler` were implemented similarly; these resources were intentionally designed to be composable and reflect a clear hierarchy: `Fleet` manages multiple `Server` instances, while `GameType` orchestrates one or more `Fleets` for rolling updates, and `GameAutoscaler` dynamically adjusts scaling parameters for a `GameType` based on policies. Simple manifests for viewing are available in the appendix: code 11, code 12, and code 14.

4.1.4 Server Controller Logic

After defining all the initial types, the implementation for the controller logic began, again following the order of resource dependency: `Server`, `Fleet`, `GameType`, and finally `GameAutoscaler`. The `Server` controller is the most complex among these. It includes logic to manage Kubernetes finalizers, handle pod lifecycle events, and make HTTP requests to a sidecar running in the same pod to check readiness before deletion.

The `Server` controller logic required careful consideration to ensure that all safety conditions were met and that deletions did not happen prematurely. The `Server` controller manages individual game server pods. It ensures the pod is created, applies finalizers for blocking deletion, and handles deletion logic that involves a check via HTTP to the sidecar.

The reconciliation loop follows this flow:

1. **Finalizer Check**: If the `Server` resource is not being deleted and has no finalizer, the controller adds the finalizer. This logic can be seen in code 18.
2. **Deletion Check**: If the `Server` resource is being deleted and has the finalizer, the controller performs a deletion check by contacting the sidecar.
3. **Pod removal**: If deletion is allowed, the pod finalizer is removed and the pod is deleted.
4. **`Server` finalizer removal**: If deletion is allowed and the pod deleted, the server finalizer is removed.
5. **Pod creation and finalizer adding**: If not being deleted, it ensures a pod exists and adds a finalizer to the pod as needed

Throughout, the controller emits Kubernetes events to create an easy-to-access log of events and their reasons. A finalizer is added to the `Server` resource and the managed pod to prevent premature deletion. An HTTP request is made to the sidecar on deletion to check deletion readiness.

One of the key challenges encountered early on was dealing with resource versioning and concurrency. In Kubernetes, whenever a resource is updated, it is essential to fetch the most recent version before performing another update. Failing to do so leads to update conflicts and may cause the controller to overwrite newer changes with stale data. This issue with performing updates was particularly relevant in the `Server` controller, where multiple steps in the reconciliation loop required stateful updates across several fields and statuses. To address this, a pattern of always re-fetching the latest version of a resource from the API server before applying any modifications was adopted.

The `Server` controller is unique because it needs to manage lifecycle, communicate with sidecars, and manage finalizers on two resources. This complicated logic made it the most complex component of the operator.

4.1.5 Fleet Controller Logic

The `Fleet` controller manages groups of `Server` resources. While it is simpler than the `Server` controller, it plays a role in maintaining the correct number of server replicas, handling deletions safely, and reacting to changes in replica count specifications.

The reconciliation loop performs the following operations in order:

1. **Finalizer Management**: If the `Fleet` resource is not being deleted and does not yet have a finalizer, the controller adds one. Ensuring cleanup logic can run before the `Fleet` is deleted. That logic is similar to the `Server` controller's logic. Code for that logic can be seen in code 17.

2. **Safe Deletion**: If the `Fleet` is being deleted, the controller lists all child `Server` resources labeled with the `Fleet`'s name and triggers their deletion. Only once all servers are confirmed deleted does the controller remove the finalizer from the `Fleet` resource. This guarantees that orphaned servers are never left behind after a `Fleet` is removed.

3. **Replica Reconciliation (Scaling)**: If the `Fleet` is active (not being deleted), the controller compares the number of current replicas to the desired count in the `Fleet` specification.

    - If there are too few, it creates new `Server` resources.

    - If there are too many, it attempts to select and delete excess `Server` resources.

After each operation, the controller updates the 'Fleet.Status.CurrentReplicas' field to reflect the current system state. The controller uses a label selector to query all `Server` resources associated with a given `Fleet`. It also uses optimistic concurrency: the resource is re-fetched before updates, and reconciliation is requeued if updates fail, ensuring eventual consistency.

4.1.6 GameType Controller Logic

The `GameType` resource manages one to two `Fleets`, depending on whether the `GameType` is in the middle of an update. The basic loop looks like this:

1. **finalizer adding**: If the `GameType` is not being deleted, then if necessary, add a finalizer to ensure full cleanup later

2. **Delete handling**: If the `GameType` is being deleted, delete the underlying fleets. Remove the finalizer once zero fleets remain for the `GameType`.
3. **Update handling**: Finally, the most important and complex type of the `GameType` controller. Handling any updates, this is done with the following logic:
    - If no underlying `Fleets`, create one.
    - If one underlying `Fleets`, check if its underlying pod spec is the same as the one currently in the `GameType` object's spec. If not, create a new fleet with the new spec.
    - If more than 1 `Fleet`, request the old ones deletion.

After each update operation, the controller updates 'gametype.Status.CurrentFleetName' to easily track which `Fleet` is the newest. The status field decides which `Fleets` replica counts to update in further reconciliation loops.

4.1.7 GameAutoscaler Controller Logic

The simplest control logic is the `GameAutoscaler`, which simply put, sends a request periodically, based on the configured period, and then changes a `GameType` fleet specs replica count. The reconcile loop looks something like this:

1. **Get Gametype**: Get `GameType` from the same namespace, based on the gamename defined in the spec.
2. **Send webhook**: Send request to the webhook
3. **Requeue**: If webhook responds with no scaling needed, then requeue the reconcile after the configured time
4. **Update replica count**: Otherwise, update the `GameTypes Fleet` spec replica count with the amount that the webhook responded with and requeue after the configured time.

The time configured here is set by the administrator in the specification for how often the webhook should be requested.

4.1.8 Service

The service module is a system component that offers quality-of-life features for interacting with the custom resources managed by the controller. It provides a lightweight REST API to create and delete these resources and manage their associated labels.

The primary motivation for this module stemmed from personal development needs—specifically, the need to create and modify custom resources without manually configuring Kubernetes RoleBindings or interacting directly with the Kubernetes API. The service reduces friction during development and testing by abstracting these operations behind a simplified HTTP interface.

While this service is not the central focus of the thesis, its design and implementation contribute to the overall usability of the operator. For instance, it supports the following key operations:

- Creating and deleting `Server`, `Fleet`, `GameType` and `GameTypeAutoscaler` resources
- Adding and removing labels from `Server` pods

A complete list of supported API routes, request formats, and expected behaviors is documented in the thesis project documentation[27].

4.2 Testing

Testing custom controllers in Kubernetes, especially when working with Kubebuilder, requires thoughtful planning due to the nature of distributed systems and the Kubernetes reconciliation loop. In general, Kubebuilder-based projects support two primary approaches[8]: unit testing using `envtest`[28], and end-to-end (E2E) testing using full Kubernetes environments such as `kind`[29].

Initially, much of the early testing was done manually in a local cluster, allowing controller behavior validation quickly and iteratively during development. Manual testing was especially useful while the design and API were still evolving. However, as the project matured, writing automated tests to verify correctness and guard against regressions became increasingly important.

After some experimentation and research, the decision was made to adopt a hybrid approach combining both unit and E2E tests. The nature of the system drove this decision: many of the custom resources interact with one another in ways that are difficult to isolate. For example,

[27] https://unfamousthomas.github.io/thesis-initial/service/
[28] https://book.kubebuilder.io/reference/envtest.html
[29] https://kind.sigs.k8s.io/

creating a `GameType` triggers the creation of a `Fleet`, which then spawns multiple `Server` resources, each of which results in an actual pod. Simulating and verifying this complete workflow in a pure unit testing environment proved complex and brittle.

That said, unit testing still offered several significant advantages. Tests ran significantly faster, were easier to run in CI/CD pipelines, and allowed for accurate code coverage analysis, which is much harder to achieve when running code inside containers. As a result, unit tests were used for more isolated functionality and basic validation logic, while E2E tests were utilized for integration-heavy scenarios and multi-resource interactions.

Kubebuilder uses Ginkgo[30] and Gomega[31] as its default test frameworks, and they were used for all testing due to their maturity and clean syntax. Their expressive assertions and built-in support for asynchronous testing were particularly useful when working with eventual consistency in Kubernetes.

4.2.1 Unit Testing

Unit tests in this project were built on top of `envtest`, which creates a lightweight, in-memory Kubernetes API server and etcd instance. This setup allows tests against real API machinery without requiring a full Kubernetes node. The CRDs used by the project are loaded into the environment, making it possible to simulate realistic behavior in a controlled and isolated context. The test environment is initialized using code similar to code 7.

---

[30] https://onsi.github.io/ginkgo/

[31] https://github.com/onsi/gomega

```
By("bootstrapping test environment")
testEnv = &envtest.Environment{
    CRDDirectoryPaths: []string{filepath.Join("..",
        "..", "config", "crd", "bases")},
    ErrorIfCRDPathMissing: true,
    BinaryAssetsDirectory: filepath.Join("..",
        "..", "bin", "k8s", fmt.Sprintf("1.30.0-%s-%s",
        runtime.GOOS, runtime.GOARCH)),
}
cfg, err := testEnv.Start()
```

Code 7. Unit test environment setup

The `By` statements serve as descriptive markers for each test step, which helps identify failures and clarify test logs.

A representative unit test example is shown in code 8. This test validates that when a `Server` resource is created, the controller correctly spawns a corresponding pod.

```
It("should create a Pod for the Server", func() {
    reconciler := &ServerReconciler{
        Client:          k8sClient,
        Scheme:          k8sClient.Scheme(),
        DeletionAllowed: checker,
    }
    By("Reconciling the resource")
    _, err = reconciler.Reconcile(ctx,
    reconcile.Request{NamespacedName: namespacedName})
    Expect(err).NotTo(HaveOccurred())

    _, err = reconciler.Reconcile(ctx,
    reconcile.Request{NamespacedName: namespacedName})
    Expect(err).NotTo(HaveOccurred())

    By("Validating a Pod is created")
    pod := &corev1.Pod{}
    Expect(k8sClient.Get(ctx, types.NamespacedName{
        Name:      ServerName + "-pod",
        Namespace: ServerNamespace,
    }, pod)).To(Succeed())
})
```

Code 8. Simple `Server` reconciliation unit test

Most unit tests were structured using `BeforeEach` and `AfterEach` blocks to manage the setup and cleanup of resources. These tests are fast to execute—typically under a minute—and can be run using standard Go tooling via `go test`.

4.2.2 End-To-End Testing

End-to-end tests are significantly more complex but provide crucial verification for interactions across multiple controllers and resources. These tests begin by creating a fresh `kind`[32] cluster (and tearing down any preexisting one, if necessary). They then install CRDs, configure

[32]Kind - Kubernetes in docker. Kubernetes cluster run locally for testing

dependencies like `cert-manager`, and build the required Docker images for the main operator and the sidecar containers.

While this setup takes time, often several minutes, ensuring a realistic environment that closely mimics a production-like Kubernetes cluster is necessary. The tests themselves are driven by executing real `kubectl` commands and applying manifests, mimicking how users or CI systems would interact with the controllers in practice. In code 9 is a simplified example of an E2E test that checks that a Pod matching the `Server` is created.

```
It("Creates a pod when server is created", func() {
    podName := serverName + "-pod"
    podCmd := exec.Command("kubectl", "get", "pod",
    podName, "-n", namespace)
    _, err := utils.Run(podCmd)
    ExpectWithOffset(1, err).NotTo(HaveOccurred())

    labelCmd := exec.Command("kubectl", "get", "pod",
        podName, "-n", namespace,
        "-o", "jsonpath={.metadata.labels.server}")
    output, err := utils.Run(labelCmd)
    ExpectWithOffset(1, err).NotTo(HaveOccurred())
    ExpectWithOffset(1, string(output)).
        Should(Equal(serverName))
})
```

Code 9. End-to-End tests check if pod exists with correct labels

Because these tests can take over 10 minutes to run, especially on slower machines or when rebuilding images, it's recommended to use the provided `Makefile` goal to simplify execution: `make test-e2e`. This target encapsulates all necessary setup steps, ensuring the environment is ready before the tests begin.

Despite their complexity, end-to-end tests are essential for validating the full system behavior and providing confidence that the controllers behave as expected in a real cluster scenario.

4.3 Documentation

At one point during the thesis writing, it was realized that this project would likely be helpful for other people, whether for inspiration or usage. Some utilities, like README files and documentation, were set up to make it easier.

This chapter focuses on the documentation, how it was written, and what was used for it.

4.3.1 MKDocs

The software used for the documentation is MKDocs[33]. It is a common software used for different types of software documentation. According to their website:

> ”MkDocs is a **fast**, **simple** and **downright gorgeous** static site generator that’s geared towards building project documentation.”

Additionally, to make the documentation a little bit nicer and provide more features for future-proofing, the material theme[34] was used. The theme enables a bunch of different features, like search. Many companies and open-source projects use Material for Mkdocs to create professional-looking documentation websites easily.

Simply put, MKDocs and Material Theme allow writing simple Markdown files as documentation, which is then rendered into statically served HTML, CSS, and JS.

4.3.2 Writing Documentation

For the documentation itself, the following subjects were covered:

- The four resource types (`Server`, `Fleet`, `GameType`, `GameAutoscaler`) - What each resource’s manifest looks like, how they work, what they do.
- Service - A quick overview of the service module and its usefulness.
- Sidecar logic - Sidecar routes and JSON, what they do.
- Getting Started - How to install the application on a cluster.
- Architecture - How the application is set up, how each thing communicates.

[33] https://www.mkdocs.org/
[34] https://squidfunk.github.io/mkdocs-material/

These give an overview of each part of the application. The idea was to keep the documentation as concise and useful as possible. For example, the `Server` documentation has:

- Quick description with introduction to sidecar.
- The manifest to create the resource.
- Environment variables that are automatically injected into the resource.
- More in-depth explanation of some of the fields of the manifest.
- Tips for creating the manifests.

This gives the application's user the necessary overview of what it does and should provide enough clarity to understand each part of the application.

4.4 DevOps

Several DevOps workflows to automate building, testing, and publishing the controller system were also set up as part of the implementation process. These workflows were set up in GitHub Actions, with tight integration with the development workflow.

The automation infrastructure was split into two primary categories of pipelines: testing and publishing.

4.4.1 Testing Pipelines

Currently, there are three direct testing pipelines and two fallback pipelines.

Direct Testing Pipelines

The direct testing pipelines did precisely that: test some module. The precise way the test was run depends on the module and the dependencies needed for the test. For example, end-to-end tests also require us to install Kind and set up the cluster. An example of a test workflow can be seen in code 15.

Some of these tests were also used as checks[35] that were required for a merge to be allowed. The checks allowed for blocking the merge if some of the tests failed.

[35] https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/collaborating-on-repositories-with-code-quality-features/about-status-checks

Fallback pipelines

Since some of the pipelines need to run for a merge to be allowed, but at the time of writing, GitHub still requires those checks to run, even if the files that were changes were in a path that would not run the workflow, a workaround was made, where if no changes were made to that part of the application, a fallback pipeline would run. A fallback pipeline is a simple pipeline with a single `echo` command. A simple fallback pipeline can be seen in code 16.

4.4.2 Publishing Pipelines

Currently, there are four publishing pipelines. Three of those just build and publish images, and one publishes the documentation. All of these pipelines run directly on the main branch.

Image Build and Publish

These pipelines build a Docker image based on some provided Dockerfile, and then publish it to GitHub's container registry[36]. This allows the images to be easily accessed as needed.

The precise logic depends on the module that is being built and published, but the workflows are available in the repository[37].

Documentation publish The documentation publishing workflow publishes the Mkdocs[38] based documentation into the documentation domain[39] every time there are changes there. It builds the pages from markdown format to standard HTML, CSS, and JS.

It then uses GitHub's pages tool[40], to publish the built artifacts to the *gh-pages* branch. The logic itself is pretty simple; there are just a lot of steps to set up the necessary environment. The pipeline workflow can be seen in the repository[41].

---

[36] https://docs.github.com/en/packages/working-with-a-github-packages-registry/working-with-the-container-registry

[37] Sidecar Publishing - https://github.com/UnfamousThomas/thesis-initial/blob/master/.github/workflows/build-and-publish-sidecar.yaml

[38] https://www.mkdocs.org/

[39] https://unfamousthomas.github.io/thesis-initial/

[40] https://pages.github.com/

[41] Documentation Publishing - https://github.com/UnfamousThomas/thesis-initial/blob/master/.github/workflows/deploy-docs.yaml

5. Results

This chapter outlines the results achieved from the design and development phases, validates requirements, and outlines any potential future developments.

5.1 Validation

Although some of the validations could be done with automatic tests, manual testing is better to ensure everything is correct.

5.1.1 Server

For the `Server` testing, a basic `Server` manifest was created. This can be found in the documentation, but for the sake of readability, it is also available at code 10. It is relatively basic, and is more so for checking if the `Server` resource itself works. Later on, it was also tested with *allowForceDelete* set to true.

First of all, checking whether the pod itself is created and is healthy is important. This can be seen in figure 10. As can be seen from the figure, the sidecar container is automatically added, meaning the requirement **FR3** is fulfilled.

```
pnap61729@game1:~/staging$ kubectl get server
NAME            AGE
server-sample   6s
pnap61729@game1:~/staging$ kubectl get pod -l server=server-sample
NAME                READY   STATUS    RESTARTS   AGE
server-sample-pod   2/2     Running   0          27s
```

Figure 10. `Server` is created with matching pod.

Furthermore, both the server and pod have the correct finalizers. This can be seen in figure 11, meaning they will not be directly deleted. This meets the requirement **FR6**.

```
pnap61729@game1:~/staging$ kubectl get server server-sample -o jsonpath='{.metadata.finalizers}'
["servers.unfamousthomas.me/finalizer"]pnap61729@game1:~/staging$
pnap61729@game1:~/staging$ kubectl get pod server-sample-pod -o jsonpath='{.metadata.finalizers}'
["servers.unfamousthomas.me/finalizer"]pnap61729@game1:~/staging$
```

Figure 11. `Server` and pod finalizers

Since communication with the sidecar is required, the pod's internal IP address must be determined. This can be seen in figure 12 with the pod's IP being *10.42.0.40*.

```
pnap61729@game1:~/staging$ kubectl get pod server-sample-pod -o jsonpath='{.status.podIP'}
10.42.0.40pnap61729@game1:~/staging$
```

Figure 12. Extract pod ip

Now, the next validation has a bunch of different actions:

- Using curl, check what is `Server` deletion allowed state.
- Using curl, check what is `Server` deletion requested state.
- Trigger deletion without waiting for it to finish.
- Run curl commands again to see if anything changed.

The above can all be seen in figure 13. As you can see, the `Server` persists after it is set to delete, but the shutdown is set to true, meaning that the game `Server` can check if its deletion has been requested easily. This fulfills requirement **FR1**.

```
pnap61729@game1:~/staging$ curl 10.42.0.40:8080/shutdown
{"shutdown":false}
pnap61729@game1:~/staging$ curl 10.42.0.40:8080/allow_delete
{"allowed":false}
pnap61729@game1:~/staging$ kubectl delete server server-sample --wait=false
server.network.unfamousthomas.me "server-sample" deleted
pnap61729@game1:~/staging$ curl 10.42.0.40:8080/shutdown
{"shutdown":true}
pnap61729@game1:~/staging$ curl 10.42.0.40:8080/allow_delete
{"allowed":false}
pnap61729@game1:~/staging$ kubectl get server
NAME            AGE
server-sample   7m42s
```

Figure 13. Sidecar state before and after delete

Lastly, the outcome of sending a POST request to permit deletion is observed. This is, once again, done via a POST request. This can be seen in figure 14. After that, it can sometimes take a while, as Kubernetes operates in eventual consistency rather than instant, but eventually the `Server` and pod both disappear. Currently, to avoid having to write `Server` software, it is done manually with curl, but in a real production environment, it would be the `Server` software next to the sidecar that would make these requests to localhost:8080. This enables the **FR2** requirement.

```
pnap61729@game1:~/staging$ curl -X POST http://10.42.0.40:8080/allow_delete \
  -H "Content-Type: application/json" \
  -d '{"allowed": true}'
{"allowed":true}
pnap61729@game1:~/staging$ kubectl get server
No resources found in default namespace.
```

Figure 14. Send request to allow delete

5.1.2 Fleet

As with the `Server` validations, a fleet manifest was created first. Again, this can be found in the documentation and code 11.

First, a fleet was created, and it was checked that the underlying servers were created correctly and the desired amount. This means requirement **FR9** is fulfilled. This can be seen in figure 15.

```
pnap61729@game1:~/staging$ kubectl get fleet
NAME           DESIRED REPLICAS   CURRENT REPLICAS
fleet-sample   3                  3
pnap61729@game1:~/staging$ kubectl get server -l fleet=fleet-sample
NAME                 AGE
fleet-sample-8ggnc   16s
fleet-sample-glvnb   16s
fleet-sample-r7lxs   16s
pnap61729@game1:~/staging$ kubectl get pod -l fleet=fleet-sample
NAME                     READY   STATUS    RESTARTS   AGE
fleet-sample-8ggnc-pod   2/2     Running   0          27s
fleet-sample-glvnb-pod   2/2     Running   0          27s
fleet-sample-r7lxs-pod   2/2     Running   0          27s
```

Figure 15. `Fleet` `Servers` after creation

Next, the presence of the finalizers is verified to ensure they were added correctly. This can be seen in figure 16. Thus, requirement **FR7** is met.

```
pnap61729@game1:~/staging$ kubectl get fleet fleet-sample -o jsonpath='{.metadata.finalizers}'
["fleets.unfamousthomas.me/finalizer"]pnap61729@game1:~/staging$
```

Figure 16. Fleet has the correct finalizers

Then, the effect of setting `spec.scaling.replicas` to 6 is evaluated. As you can see in figure 17, the number of underlying `Servers` automatically increases.

```
pnap61729@game1:~/staging$ nano fleet-example-manifest.yaml
pnap61729@game1:~/staging$ kubectl apply -f fleet-example-manifest.yaml
Warning: New timeout will not affect previously created servers
fleet.network.unfamousthomas.me/fleet-sample configured
pnap61729@game1:~/staging$ kubectl get pod -l fleet=fleet-sample
NAME                       READY   STATUS    RESTARTS   AGE
fleet-sample-5cpdg-pod     2/2     Running   0          53s
fleet-sample-8ggnc-pod     2/2     Running   0          4m37s
fleet-sample-glvnb-pod     2/2     Running   0          4m37s
fleet-sample-lqrqv-pod     2/2     Running   0          14s
fleet-sample-r7lxs-pod     2/2     Running   0          4m37s
fleet-sample-vx8q9-pod     2/2     Running   0          53s
pnap61729@game1:~/staging$ kubectl get fleet
NAME           DESIRED REPLICAS   CURRENT REPLICAS
fleet-sample   6                  6
```

Figure 17. Fleet after increasing replica amount

What is much more complex is checking what happens when the replica count is decreased. Based on our initial manifest, the `Server` should first try to find any `Servers` where deletion is allowed, if there are multiple, prioritize by oldest. If there are none, just find the oldest `Server` and request that to be deleted.

To begin testing this behavior, the oldest `Server` is identified. The goal is to observe whether scaling down the `Fleet` by one triggers its deletion, as indicated by shutdown-related updates. This can be seen in figure 18. This means requirement **FR10** is fulfilled.

```
pnap61729@game1:~/staging$ kubectl get servers -l fleet=fleet-sample --sort-by=.metadata.creationTimestamp
NAME                AGE
fleet-sample-4tdqn   67s
fleet-sample-5vcn4   67s
fleet-sample-gdcxb   67s
fleet-sample-sm7bq   67s
fleet-sample-vq2gf   67s
fleet-sample-whz8d   67s
pnap61729@game1:~/staging$ kubectl get pod fleet-sample-4tdqn-pod -o jsonpath='{.status.podIP}'
10.42.0.189pnap61729@game1:~/staging$
pnap61729@game1:~/staging$ curl http://10.42.0.189:8080/shutdown
{"shutdown":false}
pnap61729@game1:~/staging$ nano fleet-example-manifest.yaml
pnap61729@game1:~/staging$ kubectl apply -f fleet-example-manifest.yaml
Warning: New timeout will not affect previously created servers
fleet.network.unfamousthomas.me/fleet-sample configured
pnap61729@game1:~/staging$ curl http://10.42.0.189:8080/shutdown
{"shutdown":true}
```

Figure 18. Fleet scaling down with oldest server first

Next, it is important to check that when there are servers with delete already allowed, it removes those. This is seen in figure 19. This all means that requirement **FR10** is fulfilled.

```
pnap61729@game1:~/staging$ kubectl get server --sort-by=.metadata.creationTimestamp -o jsonpath="{.items[0].metadata.name}"
fleet-sample-r744dpnap61729@game1:~/staging$
pnap61729@game1:~/staging$ kubectl get pod fleet-sample-4zjpp-pod -o jsonpath='{.status.podIP}'
10.42.0.110pnap61729@game1:~/staging$
pnap61729@game1:~/staging$ kubectl get pod fleet-sample-r744d-pod -o jsonpath='{.status.podIP}'
10.42.0.23pnap61729@game1:~/staging$
pnap61729@game1:~/staging$ curl -X POST http://10.42.0.110:8080/allow_delete -H "Content-Type: application/json" -d '{"allowed": true}'
{"allowed":true}
pnap61729@game1:~/staging$ curl -X POST http://10.42.0.23:8080/allow_delete -H "Content-Type: application/json" -d '{"allowed": true}'
{"allowed":true}
pnap61729@game1:~/staging$ nano fleet-example-manifest.yaml
pnap61729@game1:~/staging$ kubectl apply -f fleet-example-manifest.yaml
Warning: New timeout will not affect previously created servers
fleet.network.unfamousthomas.me/fleet-sample configured
pnap61729@game1:~/staging$ kubectl get server
NAME                AGE
fleet-sample-4zjpp   3m19s
fleet-sample-6qrgd   3m19s
fleet-sample-kwqcs   39m
fleet-sample-zglrm   80m
```

Figure 19. Fleet scaling down with oldest allowed server first

Finally, fault tolerance is examined, specifically, whether the `Fleet` will generate a new `Server` when an existing one is deleted. To achieve this, a specific `Server` within the `Fleet` is requested for deletion, and the deletion is permitted. Figure 20 shows this process. This fulfills requirement **FR11**

```
pnap61729@game1:~/staging$ kubectl get pods -l fleet=fleet-sample
NAME                    READY   STATUS    RESTARTS   AGE
fleet-sample-4zjpp-pod   2/2     Running   0          4m45s
fleet-sample-6qrgd-pod   2/2     Running   0          4m45s
fleet-sample-kwqcs-pod   2/2     Running   0          41m
fleet-sample-zglrm-pod   2/2     Running   0          81m
pnap61729@game1:~/staging$ kubectl get pod fleet-sample-zglrm-pod -o jsonpath='{.status.podIP}'
10.42.0.143pnap61729@game1:~/staging$
pnap61729@game1:~/staging$ curl -X POST http://10.42.0.143:8080/allow_delete -H "Content-Type: application/json" -d '{"allowed": true}'
{"allowed":true}
pnap61729@game1:~/staging$ kubectl delete server fleet-sample-zglrm --wait=false
server.network.unfamousthomas.me "fleet-sample-zglrm" deleted
pnap61729@game1:~/staging$ kubectl get pods -l fleet=fleet-sample
NAME                    READY   STATUS    RESTARTS   AGE
fleet-sample-4zjpp-pod   2/2     Running   0          6m3s
fleet-sample-6qrgd-pod   2/2     Running   0          6m3s
fleet-sample-kwqcs-pod   2/2     Running   0          42m
fleet-sample-qwlsv-pod   2/2     Running   0          14s
```

Figure 20. Fleet creates new `Server` if one is manually removed

5.1.3 GameType

Like the previous resources, the `GameType` validation is similarly started by creating a new resource. The manifest is available in code 12.

First, the correct addition of finalizers is confirmed, as shown in figure 21. This handles the **FR8** requirement.

```
pnap61729@game1:~/staging$ kubectl get gametype gametype-sample -o jsonpath='{.metadata.finalizers'}
["gametype.unfamousthomas.me/finalizer"]pnap61729@game1:~/staging$
```

Figure 21. GameType has the correct finalizer

As a reminder, `GameType` acts as a wrapper for 1 to 2 fleets, so that rolling updates are possible. To test this functionality, the image version is changed from `latest` to an alternative value.

As shown in figure 22, after updating the image version, two underlying `Fleets` exist, whereas before only one existed. These are both fleets for the `GameType`. The original fleet cannot be deleted until its `Servers` have been allowed to be deleted via POST requests or the `Server` specified timeouts have passed. This whole functionality fulfills a bunch of requirements:

- **FR4** - Zero direct downtime for updating, as a new `Fleet` is created with the new spec.
- **FR12** - Transitions between `Fleets` during version upgrades are handled, in that the old fleet is deleted when the `Servers` allow for it or a timeout happens.

```
pnap61729@game1:~/staging$ nano gametype-example-manifest.yaml
pnap61729@game1:~/staging$ kubectl apply -f gametype-example-manifest.yaml
gametype.network.unfamousthomas.me/gametype-sample configured
pnap61729@game1:~/staging$ kubectl get fleet -l type=gametype-sample
NAME                     DESIRED REPLICAS   CURRENT REPLICAS
gametype-sample-csxk5    3                  3
gametype-sample-hcfws    3                  3
```

Figure 22. `GameType` has two `Fleets` while updating

The main functionality of the `GameType` resource is to enable updating fleets.

5.1.4 Fake Webhook

For the GameAutoscaler testing, a webhook-type service that responds to requests is needed. However, generally, a lot of the logic to calculate if a `Server` should be scaled can be quite complex and is not really important for the context of the thesis, and can be very dependent on the game server's logic. So, to avoid having to deal with that, a simple webhook rest api was made, which has an endpoint to register what the response should be.

This way, there is no need to implement complex metric collection and corresponding calculations. The code for this can be seen in the GitHub repository, under *example/webhook*[42].

Practically speaking, this is just a simple REST interface with two paths:

- **GET** /scale - Which is used by the Autoscaler to get current scaling information.
- **POST** /scale - Which is used by me to set current scaling information.

[42] https://github.com/UnfamousThomas/thesis-initial/tree/master/example/webhook

This enables me to easily check if the autoscaler is working, without implementing custom and complex logic to track a game servers metrics.

5.1.5 GameAutoscaler

A simple fake webhook is first deployed along with a matching `Service`, as illustrated in code 13.

Then, a matching `GameAutoscaler` manifest. This is in code 14. Note the fact that the game with spec.gameName should already exist.

The fake webhook defaults to scaling to 1 replica. After creating the autoscaler, the `Fleet`'s desired replicas go from 3 to 1. To confirm the behavior, the fake webhook is set to scale to 10 replicas, as shown in Figure 23.

```
pnap61729@game1:~/staging$ kubectl get fleet
NAME                   DESIRED REPLICAS   CURRENT REPLICAS
gametype-sample-p9qk9   15                 15
pnap61729@game1:~/staging$ kubectl get pod fake-webhook-57c8c787b5-dk7qn -o jsonpath='{.status.podIP}'
10.42.0.116pnap61729@game1:~/staging$
pnap61729@game1:~/staging$ curl -X POST http://10.42.0.116:8080/scale -H "Content-Type: application/json" -d '{"scale":true, "desired_replicas": 16}'
{"desired_replicas":16,"scale":true}
pnap61729@game1:~/staging$ kubectl get fleet
NAME                   DESIRED REPLICAS   CURRENT REPLICAS
gametype-sample-p9qk9   16                 16
```

Figure 23. GameAutoscaler and fake webhook scaling a GameType to 10

The autoscaling working like that means that requirements **FR14**, **FR13**, and **FR5** are met, as the autoscaling works based on external signals.

With this, all the initial requirements are fulfilled. However, the next sub-chapter will also discuss validating the Service that was made as a somewhat extra part of the application.

5.1.6 Service

Service validation, at its core, means sending HTTP requests and checking whether the correct objects are created, deleted, or edited.

To start with, code is a simple bash script that was written to make it easier to send create `Server` requests. It sends the `Server` manifest as a JSON to the service. As you can see in figure 24, a new `Server` is created once the request is processed.

Contrarily, if you send a delete request and the `Server` can be deleted (or is forced through the request), you can see the `Server` being deleted. This can be seen in figure 25.

```
pnap61729@game1:~/staging/service-tests$ kubectl get servers
No resources found in default namespace.
pnap61729@game1:~/staging/service-tests$ bash create_server.sh http://10.42.0.37:8080
{"apiVersion":"network.unfamousthomas.me/v1alpha1","kind":"gametypes","metadata":{"name":"server","namespace":"default","labels":{"label1":"value1"}},"spec":{"pod":{"conta
iners":[{"name":"game-server","image":"nginx:latest","resources":{}}]},"timeout":"5m0s"}}pnap61729@game1:~/staging/service-tests$
pnap61729@game1:~/staging/service-tests$ kubectl get servers
NAME     AGE
server   6s
```

Figure 24. Service creating the `Server` after the request

```
pnap61729@game1:~/staging/service-tests$ bash delete_server.sh http://10.42.0.37:8080
pnap61729@game1:~/staging/service-tests$ kubectl get servers
No resources found in default namespace.
```

Figure 25. Service deleting the `Server` after the request

The Fleet, Gametype and Autoscaler resource logic works the same way, with different endpoints, but to avoid overwhelming the reader with screenshots, will not have separate screenshots.

What should be discussed, though, is label management. Using the service, you can add and remove labels from a `Server`'s pod. Both adding and removing can be seen in figure 26.

```
pnap61729@game1:~/staging/service-tests$ kubectl get pod server-pod -o jsonpath='{.metadata.labels}'
{"server":"server"}pnap61729@game1:~/staging/service-tests$
pnap61729@game1:~/staging/service-tests$ curl -X POST http://10.42.0.37:8080/server/pod/labels -H "Content-Type: application/json" -d '{
  "metadata": {
    "name": "server",
    "namespace": "default",
    "labels": {
      "label1": "val1",
      "label2": "val2"
    }
  }
}'
pnap61729@game1:~/staging/service-tests$ kubectl get pod server-pod -o jsonpath='{.metadata.labels}'
{"label1":"val1","label2":"val2","server":"server"}pnap61729@game1:~/staging/service-tests$
pnap61729@game1:~/staging/service-tests$ curl -X DELETE http://10.42.0.37:8080/server/pod/labels -H "Content-Type: application/json" -d '{
  "metadata": {
    "name": "server",
    "namespace": "default"
  },
  "label": "label1"
}'
pnap61729@game1:~/staging/service-tests$
pnap61729@game1:~/staging/service-tests$ kubectl get pod server-pod -o jsonpath='{.metadata.labels}'
{"label2":"val2","server":"server"}pnap61729@game1:~/staging/service-tests$
```

Figure 26. Service adding and removing labels from a `Servers` underlying pod

As can be seen, when requests are sent to the service, resources are created, edited, and deleted. As such, the service works as intended.

5.1.7 Non-Functional Requirements

Non-functional requirements are often quite subjective, so these are described from the authors' point of view as the person who set out to do them.

First, configuration. Requirement **NFR1** was not to add too many configuration options. This seems to have been fulfilled, as the configuration was kept minimal and everything that was added was considered quite thoroughly. Next, testability. Thanks to HTTP request setups

through interfaces in GoLang, most of the important logic was tested directly with unit tests, so **NFR2** is fulfilled.

A reasonable number of unit tests were created. According to GoLang's coverage tool, more than 82% of the controller logic was unit tested. Of course, this number is not ideal as it only measures how much the code was hit, rather than if it was validated. But, looking at unit tests themselves, to me, most of the important scenarios were tested and validated. As such, requirement **NFR3** should be met.

The end-to-end tests use Kind to set up a simple Kubernetes environment, to make sure the basic scenarios of the controllers work correctly there. This fulfills the **NFR4** requirement.

Webhooks are used to validate and mutate minimally, but as needed, to ensure correctness. A lot of the logic there was doable directly with Kubebuilder, too. However, requirement **NFR5** is met.

The controller and resources were tested on Kubernetes version 1.30. As such, the requirement **NFR6** is met. It likely also works on any above versions, but has not been validated directly.

Finally, the controller logic was written in GoLang with the help of Kubebuilder, meaning that the **NFR7** requirement is met.

All non-functional requirements are correctly met and fulfilled.

5.1.8 Peformance Assessment

One reason the non-functional requirements did not address performance is that evaluating it for a Kubernetes operator is difficult. Since the entire application runs within Kubernetes, performance characteristics can vary significantly depending on the environment.

That said, some performance profiling was conducted using the controller unit tests. While this does not provide a complete picture, it offers a sufficiently detailed understanding.

In terms of CPU time profiling, the top two consumers are `syscall.SyscallN` and `runtime.cgocall`, each accounting for approximately 20,000,000 nanoseconds (0.02 seconds) locally. These are low-level OS kernel calls and cannot be optimized at the application level.

Another notable CPU consumer is the IP lookup operation. Each time a request is sent to a pod's sidecar, the pod's IP is resolved anew. This could be optimized in the future by caching the pod

IP in the `Server`'s status fields. The lookup currently takes roughly 10,000,000 nanoseconds (0.01 seconds), which is acceptable.

From a CPU time perspective, there are no hotspots; everything performs reasonably well.

Memory profiling also did not reveal any major inefficiencies. While initializing envtest involves a large number of allocated objects (around 259,000), the actual reconcile calls are lightweight in comparison. The most complex call, ServerReconcile, results in approximately 16,174 object allocations.

Based on this basic profiling, the application seems to be quite performant. In previous testing, the operator was left running on a dedicated `Server` for several weeks without any noticeable losses in performance, suggesting the absence of critical memory leaks.

However, in the context of Kubernetes, what matters more than raw performance metrics is how quickly the controller responds to changes in resource state. In most cases, response times are effectively instantaneous. For example, when running `kubectl apply` followed by `kubectl get`, the resources are typically already there when running the `kubectl get` command.

A bug previously delayed the `GameType` resource, but that has since been resolved. Kubernetes is eventually consistent by design, so occasional delays are expected. Even so, new resources are generally created within 30 seconds.

In summary, the controller performs well in terms of system resource usage and practical responsiveness.

5.2 Example Application Usage

Understanding an application's usage only from text and pictures can be quite complex. To facilitate easier understanding of what was created and what it could be used for, a simple example server and scale service were created.

To avoid the example server development taking too much time, the decision was made to make a server for an existing game. For this, a library for creating custom Minecraft[43] servers called Minestom[44] was used.

This was kept relatively simple, with these functionalities:

- Sending messages to players and the console when shutdown is detected.
- Constantly checking for shutdown state.
- Kicking players a few minutes after shutdown is detected.
- A simple command for checking what the `Server`'s state is.
- Reporting player count metrics to a central service via HTTP.

This is quite simplified from a real setup, but it does give an overview of what usage could look like in a real-life scenario.

The repository with the example setup is linked in the appendix and is publicly available. It contains a README for a better overview.

Now, it can be used to further validate the project. Once a player account has logged into the example server, and they do the `/serverstate` command, they can see the state of the server currently. This is seen in figure 27.

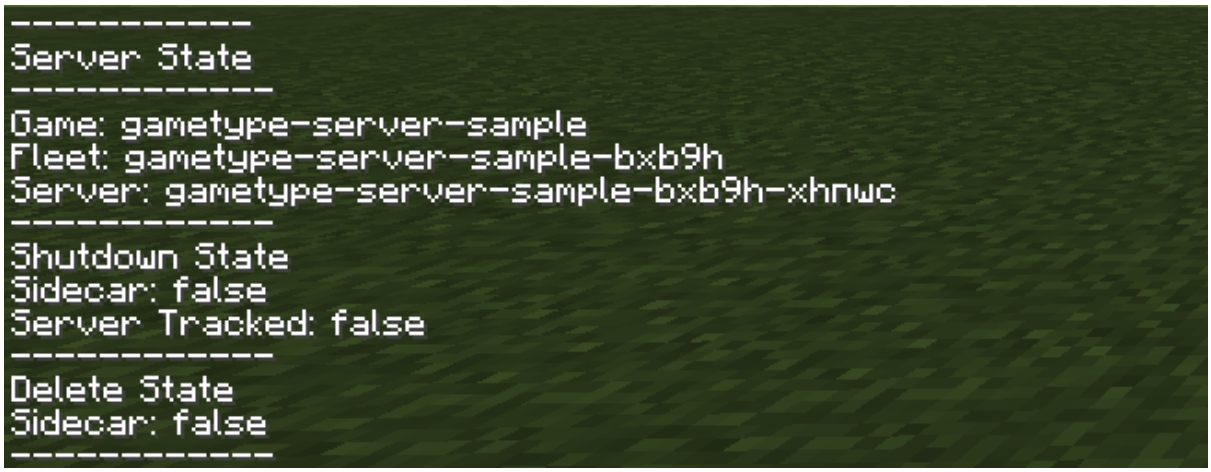

Figure 27. Example `Server` state before delete request

[43] https://www.minecraft.net/en-us
[44] https://minestom.net/

After that, through kubectl, it is possible to request the server's deletion. This can be seen from figure 28.

```
pnap61729@game1:~/loputoo-example/deployments/server$ kubectl delete server gametype-server-sample-bxb9h-xhnwc --wait=false
server.network.unfamousthomas.me "gametype-server-sample-bxb9h-xhnwc" deleted
```

Figure 28. Example `Servers` deletion being requested

Once that has happened, a message appears in the Minecraft chat interface in a few seconds to let the player know that the server will shut down soon. This can be seen in figure 29.

Figure 29. Player being notified of `Server` shutdown

The `/deleteoverride` command can be run, which requests the sidecar to allow deletion. After which, in half a minute or so, the players get kicked from the server, with figure 30 on our screen.

In real life, this logic would be a bit more complex. The flow could look something like this:

1. Shutdown is detected.
2. Begin cleanup logic for saving state, do not connect new players.
3. If no players are present, allow deletion of the `Server`.
4. Otherwise, send a message that the server is shutting down, and they will be kicked and/or redirected soon.
5. After a few minutes (delay to wait for any players that leave after seeing the message), try to redirect the players. If that is not possible, kick them.
6. No players are present now, thus server deletion can be allowed.

Of course, the specific implementation largely depends on the actual game server implementation. Some implementations could wait a few minutes and then allow deletion, ignoring connected players.

Figure 30. Player losing connection after `Server` deletion

In real life, you would want to avoid that message appearing, and rather redirect the player elsewhere or kick them pre-emptively to avoid giving strange error messages. If the command for overriding delete is not run, all players in the example server will get kicked after around a minute. The kick message can be seen in figure 31.

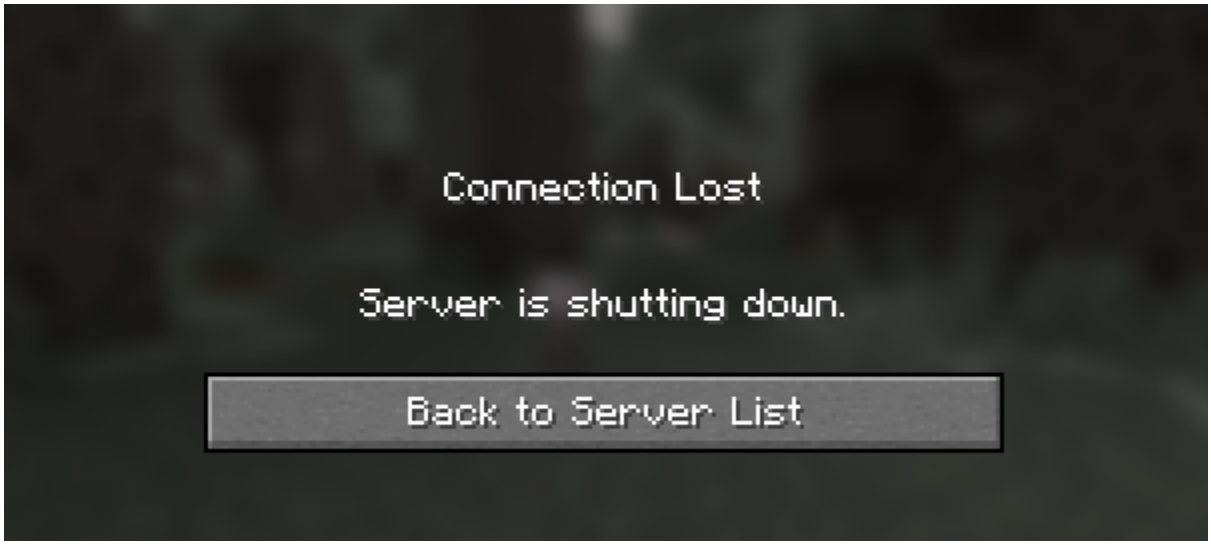

Figure 31. Player losing connection to `Server` after shutdown detected

5.3 Future Developments

Given the array of configuration options and other possible enhancements, the thesis has the potential for continuous development. This chapter highlights some of the more interesting and crucial enhancement options.

5.3.1 Configurations

On the note of configurations, quite a few could be useful, but were not implemented as part of this thesis due to time constraints. Here are just a few:

- A time must pass where deletion is allowed, before it deletes, to avoid accidental issues.
- Customizable port for the sidecar.
- Automatic external exposure of dynamically assigned ports for the `Server`, similar to how Agones handles it by creating Kubernetes `Service` objects on the fly. While a Kubernetes `Ingress` could be used for HTTP-based routing, our servers require dynamic allocation of TCP/UDP ports, which makes the standard Ingress approach unsuitable without significant customization. Implementing this functionality was out of the scope of this thesis.

5.3.2 GameType Updating Improvements

A lot of the logic related to rolling updates could be improved. For example, the controller could Wait for new `Fleets` `Servers` to be fully alive before deleting old `Fleet`. Or it could make sure not to update when two `Fleets` currently exist to limit the rolling updates happening at the same time. This logic could use some improvements, but it does work as a proof of concept.

5.3.3 Observability and Debugging

Improving visibility over the system is critical for using it in production. Future additions could include:

- Integration with Prometheus for metrics such as `Server` uptime, allocation rate, and scaling decisions.
- Better status reporting on custom resources.
- Option to dump logs or crash reports when a `Server` fails to start.
- Better and more thorough logging

5.3.4 Scaling Logic

Currently, the only way to implement autoscaling is to use a webhook with the `GameType` for horizontal scaling. Implementing some vertical scaling logic could be interesting as well. Additionally, more configurations for the `GameAutoscaler` resource, for example, other sync strategies, rather than just sending a request periodically.

There could also be some in-built logic for autoscaling automatically without being webhook-based. However, this would need to be considered quite a lot to determine what is useful without being too overwhelming configuration-wise.

5.3.5 Optimization and Testing

So far, this application has been optimized and tested for low-demand use cases. As usage grows, edge cases and performance bottlenecks will likely emerge. Additionally, the system's scalability with a larger number of `Servers` remains uncertain. To address this, future work should focus on stress testing the system under simulated high-load scenarios, implementing automated scalability benchmarks, and incorporating logging and monitoring tools to identify real-time edge-case failures.

5.3.6 Game Server Scheduling as a Research Problem

The dynamic and stateful nature of game server workloads poses unique challenges compared to traditional web or batch workloads. This project raises open questions, such as:

- How can schedulers be optimized for low latency and high utilization in multiplayer environments?
- What placement heuristics or machine learning based approaches are most effective in predicting server load or session lengths?

5.3.7 Autoscaling in State-Rich Systems

Unlike in stateless HTTP services, game servers often carry significant session and player state. This opens research questions, such as:

- How can autoscaling algorithms account for long-lived sessions, player behaviour, and migration costs?
- What models best predict when to scale in without disrupting active games?
- Could reinforcement learning be used to learn optimal scaling strategies over time?

All of these should give some ideas about how this project can be developed further, whether that be in the academic or software direction.

6. Conclusion

The thesis designed and implemented a custom Kubernetes operator for managing multiplayer game servers. A key objective was to provide a reliable, predictable, and flexible mechanism for controlling server shutdowns and preventing premature deletions, with lifecycle integration between custom Kubernetes resources and game server logic.

The resulting system successfully meets these goals. Through the creation of four custom resources (`Server`, `Fleet`, `GameType`, `GameAutoscaler`), alongside supporting components like the sidecar and REST interfaces, the system supports automated server management. The sidecar allows for graceful termination logic, while the `GameType` and `GameAutoscaler` enable advanced behaviors such as rolling updates and dynamic scaling based on external signals.

Validation efforts confirmed that all requirements have been satisfied. Key scenarios, such as finalizer-based deletion protection, rolling fleet updates, autoscaling via webhook, and RESTful resource management, were tested and shown to work as intended. The application was also tested with a real-life example game server, which showed how it could be used and that it worked as expected.

Additionally, although quite basic, performance assessments indicated no major CPU or memory bottlenecks, and the controller demonstrated responsiveness consistent with typical Kubernetes operations.

Nevertheless, this work represents an initial iteration. Multiple areas for future improvement exist, including expanded configuration options, better observability, and deeper optimization for high-load scenarios.

In conclusion, this thesis demonstrates that a custom Kubernetes operator can effectively manage multiplayer game server lifecycles with declarative patterns and lifecycle awareness. It lays a strong foundation for further exploration and development and provides a valuable tool for teams aiming to integrate game server orchestration into cloud-native systems.

References

[1] Kasenides N. and Paspallis N. Athlos: A Framework for Developing Scalable MMOG Backends on Commodity Clouds. en. *Software* 1.1 (Mar. 2022). Number: 1 Publisher: Multidisciplinary Digital Publishing Institute, pp. 107–145. DOI: 10.3390/software1010006. (05/11/2025).

[2] What is a Container? | Docker. https://www.docker.com/resources/what-container/ (12/07/2024).

[3] Abuabdo A. and Al-Sharif Z. A. Virtualization vs. Containerization: Towards a Multithreaded Performance Evaluation Approach. *2019 IEEE/ACS 16th International Conference on Computer Systems and Applications (AICCSA)*. 2019, pp. 1–6. DOI: 10.1109/AICCSA47632.2019.9035233.

[4] Docker Image vs Container - Difference Between Application Deployment Technologies - AWS. https://aws.amazon.com/compare/the-difference-between-docker-images-and-containers/ (05/15/2025).

[5] Chourasia A. How docker works behind the scenes. en. Apr. 2023. https://blog.kubesimplify.com/understanding-how-containers-work-behind-the-scenes (12/07/2024).

[6] Bottomley J. and Emelyanov P. Hypervisors and Containers. *;login:* 39.5 (Oct. 2014). https://www.usenix.org/system/files/login/articles/login_1410_02-bottomley.pdf (02/05/2025).

[7] What is Container Orchestration? - Container Orchestration Explained - AWS. https://aws.amazon.com/what-is/container-orchestration/ (05/15/2025).

[8] The KubeBuilder Contributors. Introduction - The Kubebuilder Book. en. https://book.kubebuilder.io/ (05/15/2025).

[9] The Kubernetes Contributors. Kubernetes Documentation. en. https://kubernetes.io/docs/home/ (05/15/2025).

[10] Brenner M. Kubernetes Overview Diagrams. en-us. Dec. 2020. https://shipit.dev/posts/kubernetes-overview-diagrams.html (05/15/2025).

[11] Helm. https://helm.sh/ (05/15/2025).

[12] Sidecar pattern - Azure Architecture Center. https://learn.microsoft.com/en-us/azure/architecture/patterns/sidecar (05/15/2025).

[13] Operator SDK Documentation. https://sdk.operatorframework.io/docs/ (05/15/2025).

[14] Duan R., Zhang F., and Khan S. U. A Case Study on Five Maturity Levels of A Kubernetes Operator. *2021 IEEE Cloud Summit (Cloud Summit)*. Oct. 2021, pp. 1–6. DOI: 10.1109/IEEECloudSummit52029.2021.00008. (04/20/2025).

[15] Wang Y., Zhang F., and Khan S. U. HCA Operator: A Hybrid Cloud Auto-scaling Tooling for Microservice Workloads. *2022 18th International Conference on Mobility, Sensing and Networking (MSN)*. Dec. 2022, pp. 885–890. DOI: 10.1109/MSN57253.2022.00143. (04/20/2025).

[16] Mansfield S., Enugula S., Madappa S., and Nguyen V. T. Application data caching using SSDs. en. Blog. Apr. 2017. https://netflixtechblog.com/application-data-caching-using-ssds-5bf25df851ef (05/15/2025).

[17] Lundgren J. Kubernetes for Game Development : Evaluation of the Container-Orchestration Software. 2021.

[18] Record-breaking Fall Guys scales faster with Azure. https://developer.microsoft.com/en-us/games/articles/2021/11/record-breaking-fall-guys-scales-faster-with-azure/ (05/15/2025).

[19] Agones. https://agones.dev/site/ (05/15/2025).

[20] Thundernetes. Thundernetes. https://playfab.github.io/thundernetes/ (05/15/2025).

[21] The KEDA Contributors. KEDA. https://keda.sh (05/15/2025).

[22] Howcraft J. and Mandel M. Making online, containerized games with managed services | Google Cloud Blog. https://cloud.google.com/blog/products/containers-kubernetes/making-online-containerized-games-with-managed-services (05/15/2025).

Appendices

Appendix I: Thesis Glossary

A short list of important terms discussed in the thesis, which is useful as a quick reference.

- Container - Unit of software that encapsulates code and all its dependencies.
- Image - Read-only templates that contain the instructions for creating a container.
- CGroups - Linux kernel system to limit resource usage in a container.
- (Linux) Namespaces - Process-level isolation providers which define what a container can see and access on the host system.
- Kubernetes - Standard platform for managing and orchestrating containerized workloads.
- Pods - Smallest resource in Kubernetes, typically contains one container.
- Deployments - Resource used for deploying applications in Kubernetes, can be used for rolling updates and scaling.
- ReplicaSet - Resources used for managing replicas of pods. Deployments create these automatically underneath.
- Finalizers - Strings in metadata for resources in Kubernetes, until they are in place, the resource cannot be deleted.
- API - Application Programming Interface, used for interacting with a program through an interface. Commonly, this is done via REST.
- Desired State - What the administrator wants the state of a resource to be in Kubernetes.
- Current State - What the current state actually is in Kubernetes.
- KubeCTL - Command line interface for interacting with the Kubernetes API.
- Helm - Commonly used Kubernetes package manager.
- Sidecar Pattern - Pattern where the main application pod has another pod next to it, providing some extra functionality.
- Custom Resources - Resources not defined by Kubernetes itself, but by third-party users.
- Resource Controllers - Applications that perform actions based on Custom Resources.

- Kubebuilder - Framework for building resource controllers.
- Go (or *GoLang*) - Programming language developed by Google and used by Kubernetes.
- (Kubernetes) Webhooks - Endpoints that Kubernetes asks about a resources creation. Usually used to make sure that all fields are correct and to set any defaults for values as needed.
- Operator - An application that manages some amount of custom resources in some domain. Usually contains a few resource controllers and webhooks.

Appendix II: Custom Resources

This section of the glossary has small descriptions of the custom resources defined as part of this thesis for easier access.

- Server - Custom resource that represents a game server, creates an underlying pod with a defined spec, and adds a sidecar.
- Fleet - Custom resource, used for making more or less of game servers.
- GameType - Custom resource used for changing the pod specs. Creates a new underlying fleet and slowly scales out the old one.
- GameAutoscaler - Custom resource that sends HTTP requests to an endpoint every some defined time, and based on the response, scales the GameType.

Appendix III: Main Github Repository

The GitHub repository with the application code is available at https://github.com/UnfamousThomas/thesis-initial. If, for whatever reason, the repository is no longer available there, please contact the author at *thom.palts@gmail.com*.

Appendix IV: Example Github Repository

The GitHub repository that can be used as an example is available at https://github.com/UnfamousThomas/thesis-example. If, for whatever reason, the repository is no longer available, please contact the author at *thom.palts@gmail.com*.

Appendix V: MKDocs Documentation

For the sake of easier understanding of the application and future potential open-sourcing of the application, documentation is available at https://unfamousthomas.github.io/thesis-initial/. If, for whatever reason, the documentation is no longer available there, please contact the author at *thom.palts@gmail.com*.

Appendix VI: Kubernetes Manifests

```
    apiVersion: network.unfamousthomas.me/v1alpha1
kind: Server
metadata:
  labels:
    someLabel: example-label
  name: server-sample
spec:
  timeout: 5m
  allowForceDelete: false
  pod:
    containers:
      - name: example-container
        image: nginx:latest
        ports:
          - containerPort: 80
            protocol: TCP
```

Code 10. Simple server manifest

```
    apiVersion: network.unfamousthomas.me/v1alpha1
kind: Fleet
metadata:
  name: fleet-sample
  labels:
    some-label: value1
spec:
  scaling:
    replicas: 3
    prioritizeAllowed: true
    agePriority: oldest_first
  spec:
    timeout: 5m
    allowForceDelete: false
    pod:
      containers:
        - name: example-container
          image: nginx:latest
          ports:
            - containerPort: 80
              protocol: TCP
```

Code 11. Simple fleet manifest

```yaml
apiVersion: network.unfamousthomas.me/v1alpha1
kind: GameType
metadata:
  name: gametype-sample
spec:
  fleetSpec:
    scaling:
      replicas: 3
      prioritizeAllowed: true
      agePriority: oldest_first
    spec:
      timeout: 5m
      allowForceDelete: false
      pod:
        containers:
          - name: example-container
            image: nginx:latest
            ports:
              - containerPort: 80
                protocol: TCP
```

Code 12. Simple GameType manifest

```yaml
apiVersion: network.unfamousthomas.me/v1alpha1
kind: GameAutoscaler
metadata:
  name: gameautoscaler-sample
spec:
  gameName: gametype-sample
  policy:
    type: webhook
    webhook:
      path: "/scale"
      service:
        name: fake-webhook
        namespace: default
        port: 80
  sync:
    type: fixedinterval
    interval: 30s
```

Code 13. Simple Fake webhook deployment manifest

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fake-webhook
  labels:
    app: fake-webhook
spec:
  replicas: 1
  selector:
    matchLabels:
      app: fake-webhook
  template:
    metadata:
      labels:
        app: fake-webhook
    spec:
      imagePullSecrets:
        - name: ghcr-secret
      containers:
        - name: kube-http-service
          image: ghcr.io/unfamousthomas/fake-webhook:latest
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: fake-webhook
spec:
  selector:
    app: fake-webhook
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

Code 14. Simple GameAutoscaler manifest

Appendix VII: Github Pipelines

This section has sections or full YAML files for GitHub Actions pipelines used in CI/CD.

```yaml
name: Operator E2E Tests

on:
  pull_request:
    paths:
      - 'operator/**'
      - '.github/workflows/run-operator-e2e-tests.yml' # re-run if
      ↪  workflow changes

jobs:
  e2e-test:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Set up Go
        uses: actions/setup-go@v5
        with:
          go-version: '1.24'

      - name: Install dependencies and run tests
        working-directory: ./operator
        run: |
          make kind-install
          make kind-create-cluster
          make test-e2e
```

Code 15. Github Actions: End-to-End test manifest

```yaml
name: Operator Unit Tests

on:
  pull_request:
    paths-ignore:
      - 'operator/**'
      - '.github/workflows/run-operator-unit-tests.yml' # re-run if
       ↪  workflow changes

jobs:
  unit-test:
    runs-on: ubuntu-latest

    steps:
      - run: echo "No relevant changes detected."
```

Code 16. Github Actions: Test fallback manifest

Appendix VIII: Controller Code

This section has snippets of code from the actual controller, sometimes with parts removed.

```
        if fleet.DeletionTimestamp == nil &&
        ↪  !controllerutil.ContainsFinalizer(fleet, FLEET_FINALIZER)
        ↪  {
                controllerutil.AddFinalizer(fleet, FLEET_FINALIZER)
                if err := r.Update(ctx, fleet); err != nil {
                        ...
                        return ctrl.Result{Requeue: true},
                        ↪  fmt.Errorf("failed to add finalizer to
                        ↪  fleet: %w", err)
                }
                ...
                return ctrl.Result{Requeue: true}, nil
        }
```

Code 17. Fleet Controller adding the finalizer to the fleet

```
if server.DeletionTimestamp == nil &&
↪  !controllerutil.ContainsFinalizer(server, SERVER_FINALIZER) {
                controllerutil.AddFinalizer(server, SERVER_FINALIZER)
                if err := r.Update(ctx, server); err != nil {
                        r.emitEventf(server, corev1.EventTypeWarning,
                        ↪  utils.ReasonServerUpdateFAiled, "failed to
                        ↪  update server: %s", err)
                        return ctrl.Result{}, fmt.Errorf("failed to
                        ↪  update server for finalizer: %s", err)
                }
                r.emitEvent(server, corev1.EventTypeNormal,
                ↪  utils.ReasonServerInitialized, "Finalizer added")
                return ctrl.Result{}, nil
        }
```

Code 18. Server controller checking and adding the finalizer

License

I, **Thomas Palts**

1. grant the *University of Tartu* a free permit (non-exclusive license) to reproduce, for the purpose of preservation, including for adding to the digital archives of the University of Tartu until the expiry of the term of copyright, my thesis ”Kubernetes Custom Resources and Controllers for Managing Game Servers” supervised by Pelle Jakovits;
2. grant the *University of Tartu* a permit to make the thesis specified in point 1 available to the public via the web environment of the *University of Tartu* including via the digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of a derivative works and any commercial use of work until the expiry of the term of copyright;
3. am aware the fact that the author retains rights specified in points 1 and 2;
4. confirm that granting the non-exclusive license does not infringe other persons’ intellectual property rights or rights arising from the personal data protection legislation.

Thomas Palts 14/05/2025