Master thesis : Scalable Automated Deployment of Student Laboratory Environments in an OpenShift Cluster

Auteur : Raze, Félicien
Promoteur(s) : Donnet, Benoît
Faculté : Faculté des Sciences appliquées
Diplôme : Master en sciences informatiques, à finalité spécialisée en "computer systems security"
Année académique : 2025-2026
URI/URL : https://gitlab.uliege.be/tfes/lab-revamp; http://hdl.handle.net/2268.2/26106

Université de Liège
Faculté des sciences appliquées

Scalable Automated Deployment of Student Laboratory Environments in an OpenShift Cluster

Raze Félicien

Thesis presented to obtain the degree of :
Master of Science in Computer Science

Thesis supervisor :
Professor Benoît Donnet

Academic year: **2025 - 2026**

Acknowledgements

I would first like to thank my thesis supervisor, Professor Donnet, for his valuable advice and feedback even when the timing was not ideal for him.

I would like to thank Vincent Jacquot, the teaching assistant at the initiative of this thesis' subject, for his availability throughout the year and his regular feedback.

I would also like to thank Professor Hauzeur, the main administrator of ULiège's OpenShift cluster, for his insight regarding system design and his help with regard to the deployment of the system.

Finally, I would like to warmly thank my family and friends for their continuous support and kindness.

Abstract

Computer-based laboratories play a major role for many courses, as they are often factored in student evaluation. However, configuration mismatch risks often make virtual machine backed laboratories the most convenient solution, but they are still cumbersome and require student setup. The goal of this thesis is to create a scalable lab-deployment system on the university's OpenShift cluster, requiring no prior setup from the students, but offering them pre-configured laboratory environments. The system presented relies on a two-operator architecture and uses declarative configuration for laboratories, relying directly on OpenShift for user authentication. It indirectly leverages ULiège's SSO and allows the use of role-based access control for managing laboratory environment access, for groups or individual students, with supervision if needed. The system achieves very good performance on the cluster at ULiège, creating 50 student environments in less than 10 seconds, tearing them down in less than a second, thanks to utilising native OpenShift and Kubernetes resources and design philosophy. The performance of the system allows for significant real-life workloads with several labs occurring at the same time, by leveraging an existing OpenShift cluster. This eliminates the problem of configuration mismatch and is accessible to all students without requiring them to set up a VM, simply through their browser or terminal emulator.

Contents

List of Figures

List of Tables

Introduction

0.1 Context

For multiple courses at the university, students must participate in laboratories on their own laptops. However, this sometimes causes problems: students have a wide variety of computers with different operating systems, pre-installed software, and configurations. When a student tries to install software required for a lab prepared by a teaching assistant on their own computer, it is sometimes difficult to resolve dependency issues, leading teaching assistants to provide virtual machines to the students. These virtual machines, while providing an acceptable solution to the dependency issue, have several disadvantages:

- They are not performant: often these virtual machines rely on a type two virtual machine hypervisor like VirtualBox, meaning that the virtual machine does not run on "bare metal". Although this is not a real problem for most laboratories as they mostly do not require a large use of resources, it can be an obstacle for some students during some laboratories that require more power.
- They require a setup performed ahead of time by students. Sometimes, a setup guide needs to be written in advance by the teaching assistants. The virtual machine files are large in size (usually several gigabytes).
- An additional setup is needed if students want to have convenience features like bidirectional clipboard sharing or folder sharing.

The university has had an OpenShift cluster for several years now, mostly used for the "Software project engineering and management" course (PROJ0010-1). This means that most of the time the cluster's resources are unused.

0.2 Problem answered by this thesis

The thesis provides a solution to teaching assistants so that they can provide lab environments to students without requiring students to do any prior setup by exploiting the university's OpenShift cluster, since it has a lot of unused availability. This would address the problems of having students run virtual machines for their laboratories as described in the previous section.

0.3 Proposed solution and objectives

The project associated with this thesis consists of exploring the possibility of creating a system that would allow teaching assistants and professors to set up lab templates that could be easily deployed on the university's OpenShift cluster and accessible by the students without requiring them to do anything prior. This solution should allow *no configuration mismatch* between the teaching assistant's lab environment and the students' lab environments.

The solution should be able to be applied for a wide variety of different laboratory configurations, including but not limited to:

- Laboratories in the form of "capture the flag" (CTF) exercises, with a centralised submission server made available to all students.
- Individual laboratories with one or several environments per student, with specific background tasks and available programs for students.
- Group laboratories for which one or several are accessible by group logic, meaning that a lab environment is shared by all students of the same group.
- Configurations using virtual machines instead of the default OpenShift pods. This feature was ultimately removed from the requirements due to technical constraints; see section 3.1 for further details regarding this feature and the reason for its removal.

Chapter 1

Background: OpenShift/Kubernetes concepts

This chapter provides a basic understanding of concepts that are useful for this thesis. This section is sufficient to understand all of OpenShift specific concepts used for this thesis. Since OpenShift is developed on top of Kubernetes, this section will take a look at Kubernetes concepts before more specific OpenShift concepts. This section relies heavily on Kubernetes' documentation [1] and OpenShift's documentation [2].

1.1 Kubernetes

Kubernetes is an open source container orchestration system for automatic container deployment in a cluster of computers (nodes) [3]. It was announced and made open source in 2014 by Google, the design decisions for its initial release are explained in this paper [4] and are widely based on Google's experience with its cluster management tool, Borg.

Kubernetes allows for a high level of abstraction to enable users to handle creating containers in a hardware agnostic manner.

1.1.1 Control plane

Kubernetes' control plane is a set of systems whose main goal is to ensure that the cluster state matches with the *desired cluster state*. The control plane exposes the Kubernetes API with which the users can interact and orchestrates the different components to *reconcile* the current state and the desired state. The control plane itself can be distributed on a set of control plane nodes and manage a set of worker nodes. As Kubernetes is built towards high availability architectures, the control plane components are designed towards multi-node environments:

- The API server can run simultaneously on all control plane nodes.
- The cluster's database takes the form of a distributed key-value store, `etcd`, and uses consensus algorithms to coordinate nodes. In `etcd`'s case, the consensus algorithm arbitrating the database is Raft [5].
- Other components that cannot run simultaneously, such as the cluster scheduler or the controller manager, use a leader system where the tasks are performed on the elected leader control plane node only. If this elected node becomes unavailable, another node will take the role.

Together, Kubernetes' control plane components offer a high level of abstraction to the users, and while some APIs allow for relative control over the precise use of cluster resources, the task is mainly

handled by the control plane.

Through the API, users can create and interact with resources (also known as objects), either directly described by the standard Kubernetes API or by an extension of the Kubernetes API like OpenShift. Users can also extend the Kubernetes API themselves through the creation of a custom resource definition (CRD), allowing cluster users with the right permissions to create and manage these custom resources (CR) like any other resource made available through the Kubernetes API. These resources dictate the desired cluster state according to a set of rules. These resources are hierarchical objects that are often described using YAML.

Users can interact with the Kubernetes API directly through `kubectl`, a command line utility designed for interaction with Kubernetes clusters, but this can also be done with purpose-built programs (e.g. the OpenShift Console described in section 1.2).

The subsections of section 1.1 from subsection 1.1.4 to subsection 1.1.10 describe specific resources defined by the Kubernetes API that users can create and manage assuming they have the right permissions. These permissions are created and managed through the concept of role-based access control (RBAC), see subsection 1.1.8 for more details on how RBAC roles are handled. The names of Kubernetes resources are always capitalised.

1.1.2 Namespaces

Namespaces are the main concept of logical separation used in Kubernetes. Most of the resources in a Kubernetes cluster must be associated with a specific Namespace. Resources under a same Namespace are associated together and most interactions happening through the Kubernetes API between resources in a cluster are done in the same Namespace. By default, access to resources is given to users on a Namespace basis.

Resources that are defined outside of any Namespace are called cluster-scoped resources, versus namespace-scoped for resources that are attached to a specific Namespace.

1.1.3 Events

Events are resources used to inform users about specific events taking place related to a specific resource. They are ephemeral and should not be used to trigger automated responses, as their main goal is to notify users.

When emitted, an Event must be associated with a specific resource that the emitter can designate. When a user uses the `describe` Kubernetes API on a specific resource, Events associated with that resource will be fetched with it. OpenShift's web user interface integrates Events in its UI, making all Events in a specific namespace or Events tied to a specific resource easily accessible.

1.1.4 Pods

Pods are the smallest deployable computing units in a Kubernetes cluster. They can contain one or several containers that run using the same network and can share storage through explicitly defined volumes. If several containers are running inside a single Pod, it is not possible to give Kubernetes users access to a specific container inside the Pod only, and they will have access to all containers running in the Pod. Of course, users can *choose* which container they wish to access, and if access is done exclusively through the network, it is possible to do authentication on a container basis. For API-level shell access, Pods are

the smallest available granularity.

A container can be viewed as a box for a piece of software that contains everything specific to the software it needs to run. It can be pulled from a container registry, either locally, from a public container registry, or from a private container registry.

The Pod resource allows defining the containers running on it as well as various parameters such as resource limits, network configuration, etc...

Unlike other Kubernetes resources, the word pod is used not only to designate the Pod resource, but has a more general concept. For this reason, when not capitalised, pod designates the general concept of a wrapper around one or several containers.

1.1.5 ReplicaSets and Deployments

ReplicaSets describe a set of pods with identical configuration. The ReplicaSet is the resource describing the desired state of the pods it defines, ensuring that if a pod reaches an unexpected state, e.g., a crash, it is brought back to its desired state, e.g. running.

Although their configuration is the same, pods of a common ReplicaSet are distinct and have different IP addresses and states. Regular users usually do not interact directly with ReplicaSets but use Deployments.

Deployments encapsulate ReplicaSets and serve as a higher level of abstraction above ReplicaSets and mainly handle the strategy around a single ReplicaSet regarding the updating of the pods, their creation and removal. They can serve to define whether the pods should be recreated at the same time or using a rolling strategy to improve availability, as well as various parameters regarding reporting (e.g., how long to wait before a pod that is starting is considered stalled/failing).

1.1.6 Services

Services are a layer of abstraction on top of pods that aim to provide a stable network identity to pods. Without Services, a restarting pod has no guarantee of keeping the same IP address in the cluster as before, but Services can provide a static IP by which pods with changing IPs can be accessed. Services also provide a static DNS name from within the cluster.

Services have more capabilities with respect to networking, notably access to a Service from outside of the cluster. In standard Kubernetes, access to a Service from the outside of the cluster can be handled by different resources, either with the Ingress resource or with the more modern Gateway and HTTPRoute resources. Since the university cluster uses OpenShift, the project accompanying this thesis exclusively uses the OpenShift specific Route resource as described in subsection 1.2.2.

1.1.7 Persistent Volume Claims

Since pods and their storage are ephemeral, *persistent volume claims* aim at providing a static form of storage to the pods that might necessitate it. They are an abstraction above the actual persistent storage configuration of the cluster and, in their most basic form, allow for the reservation of a requested amount of space in persistent storage. The storage is provisioned by the control plane depending on the cluster configuration. If different classes of storage are available on the cluster, users can request a specific type of storage. A persistent volume claim can be configured in different ways, allowing one or several pods to have read and/or write access to the claim.

1.1.8 Roles and RoleBindings

Roles and *RoleBindings* are the way role-based access control (RBAC) is performed in a Kubernetes cluster. In Kubernetes, role-based access control relies on the definition of Roles that allow users and resources that have that Role to do some actions (verbs) over a given resource. For example, the appendix document A.1 shows the definition of the Role `example-role` which gives the same verbs (permissions) for the `ServiceAccounts`, `ConfigMaps`, `Deployments`, and `ReplicaSets` resources, but only gives the `get` and `list` verbs for the `Roles` and `RoleBindings` resources. It also gives the `impersonate` verb but *specifically* for the `example-service-account` ServiceAccount. A wildcard can be used for the verbs to give unrestricted access to resources, but its use is discouraged, and it is better practice to explicitly define the verbs given with this Role. Roles can also attribute verbs for custom resources.

RoleBindings are mainly used as a link between a subject and a Role resource, effectively giving the access defined in the referenced Role to the subject. "Subject" can reference a regular human user, or a ServiceAccount as described in subsection 1.1.10.

Roles and RoleBindings are both namespace-scoped resources, but equivalent resources exist at the cluster level: ClusterRoles and ClusterRoleBindings. They are similar to namespaced Roles and RoleBindings, but handle role-based access control at the cluster level.

1.1.9 ConfigMaps

ConfigMaps are a resource used to store key-value data in a non-confidential manner. They can be handed to pods to allow for specific configuration of containers without having to alter the images.

For confidential data, Kubernetes defines a "Secret" resource dedicated to small amounts of confidential data, such as passwords, keys, or tokens. These Secrets, unlike ConfigMaps, are kept in volatile memory and are not written in permanent storage on a worker node. In addition, more security measures can be taken to improve security regarding Secrets.

1.1.10 ServiceAccounts

A ServiceAccount serves as a namespaced identity identifier for entities within the cluster, also known as a "subject". Any pod within the Namespace is identified as any unique ServiceAccount from the Namespace. Regular users who have the `impersonate` verb over a specific ServiceAccount can use it as their identity for RBAC. During Namespace creation, a default ServiceAccount is created for the Namespace and, unless specified otherwise, pods in the Namespace will use that ServiceAccount to identify.

1.1.11 Operators

Operators extend Kubernetes' capabilities by defining logic that continuously watches the current state in order to keep the cluster in its desired state, as defined by the creator of the operator. Often, operators rely on custom resource definitions and can take actions and interact with the cluster based on the state of the custom resources. The logic reconciliation is handled inside the operator's controller. Operators are very flexible in how they handle the reconciliation process.

Operators can interact directly with the Kubernetes API and are effectively pods running on the cluster, in a Namespace. Operators can be namespaced or have cluster-wide capabilities, so operator pods with cluster-wide capabilities must have a suitable ServiceAccount with the right ClusterRole attached via a ClusterRoleBinding.

1.2 OpenShift

OpenShift [6] is an open source extension of Kubernetes developed by Red Hat, offering a superset of the capabilities available in standard Kubernetes. Although the original version of OpenShift was released in 2011 before Kubernetes' release, Kubernetes was adopted as their container management engine in OpenShift v3 [7].

1.2.1 Projects

Projects are a wrapper around usual Kubernetes Namespaces that allow for simplifying administrative tasks around that Namespace. For example, it enables users to create Projects themselves, without admin intervention if the cluster is configured that way. To be precise, the users can usually not directly create Projects, but can create ProjectRequests. When the cluster sees a ProjectRequest, it creates the Project and the associated Namespace. It can automatically attribute admin permissions over the Namespace without admin intervention and allow automatic security and resource limits to be added for the created Namespace. It also adds human readable names and descriptions.

The OpenShift cluster at ULiège is configured so that regular users can create ProjectRequests and cannot directly create Projects or Namespaces. It also automatically assigns the admin Role to the user that created the ProjectRequest for the specific Namespace.

For the purpose of this thesis, creating a Namespace will always involve creating a ProjectRequest.

1.2.2 Routes

Routes are OpenShift's specific way to associate an identity in the form of an address with a destination Service (or internal address) in the cluster. It also allows users to handle the security policy (for TLS termination) and offers basic load balancing capabilities.

They are OpenShift's standard way to expose a public URL using the cluster base URL. In the case of the ULiège OpenShift cluster, the base URL is `apps.speam.montefiore.uliege.be`. It is of course possible to point other domain names to that Service, for example, by pointing a `CNAME` DNS record for the desired domain name to a cluster-specific URL exposed by a Route.

1.2.3 SecurityContextConstraints

SecurityContextConstraints (SCCs) are OpenShift specific resources allowing specific security permissions for pods.

SCCs are relevant here because the default SCC for the cluster is the `restricted-v2` SCC, which imposes very restrictive security constraints on the pods created with this SCC. This could restrict some desired functionalities, like the ability to use `setuid` inside of a pod. Although generally good practice, in the context of labs that require students to exploit security issues, it can be too restrictive and not allow the students to use the exploits as expected.

1.2.4 OpenShift Console

OpenShift's console is a graphical web interface that allows users to interact directly with the cluster. It allows for complex interaction with the cluster and management of the different cluster resources users

have access to.

Users connect on the web interface via the method provided by the cluster. In the case of the ULiège cluster, authentication is exclusively done via the single sign on (SSO) service of ULiège.

OpenShift has its own command-line utility, `oc`, that offers a superset of the capabilities offered by `kubectl` for interacting with an OpenShift cluster. OpenShift's web interface allows for easily getting identification tokens for authentication on the command line interface, after users authenticate via the web interface.

Chapter 2

Related Work and tools

2.1 CrownLabs

CrownLabs [8] is a multi-component system that allows the automatic provisioning and creation of student lab environments in a Kubernetes cluster, developed since 2020 by a research group in the Italian university "Politecnico di Torino" (PoliTo). Its main goal at the start of its development was to allow students to have remote-access labs during the coronavirus pandemic. In its current state, CrownLabs allows students to connect to a specific configured laboratory environment, either a pod or a full graphical virtual machine, using KubeVirt [9], an extension of the Kubernetes API that allows provisioning full virtual machines in a way similar to how pods are provisioned.

Despite its similarity with the goal of this thesis, deploying CrownLabs on ULiège's OpenShift would come at a high deployment, operation maintenance cost, and requires some capabilities that the current configuration of the ULiège cluster does not offer, such as the ability to provision virtual machines, and some other significant extensions of the Kubernetes API.

The goal of this thesis is to develop a highly maintainable and modular system with easy onboarding that could run on any OpenShift cluster, exploiting the specific capabilities of OpenShift.

2.2 ContainerSSH

ContainerSSH [10] is an SSH server that can automatically launch new containers on Kubernetes or docker from a specific configuration. When a client reaches it, it authenticates the client, then creates a container (which would be a single container in a single pod for Kubernetes) with a configuration given by the configuration server.

For authentication, ContainerSSH provides two solutions:

- Identification via an ssh key: the server has a list of public ssh keys associated to the users. The user ssh software authenticates through signing a message with the user's private key. The authentication server can verify that the signature corresponds to the user using their public key. This would require students to provide accurate ssh keys and keep them securely. There could be mistakes making students lose valuable time during the labs.
- Identification via password: students simply use a password that the authentication server can verify. This means that either the students have to provide a password to the teaching assistant in advance of the laboratory, or the teaching assistant has to distribute passwords.

For the purposes of this thesis, the ideal authentication form would be direct identification with the students' ULiège accounts. Beyond the advantage of necessitating no work from the students before

the laboratory takes place, it also provides a stronger assurance that the students will not share their account, since giving access to their lab environment to another student would mean giving them access to their whole ULiège account. Password and SSH key authentication does not come with this guarantee, assuming that students do not reuse their SSH keys and passwords.

Since all students are users in the university's OpenShift cluster and log in through ULiège's single sign on (SSO) service, it means that simple RBAC rules can be used to ensure appropriate access permissions are given to each student during the lab.

ContainerSSH has not had a stable 1.0 release yet, making it not ideal for use in an academic project.

Chapter 3

System architecture and design

This chapter's goal is to explain the general design considerations that came when creating the system. More specific implementation considerations are explored in chapter 4.

3.1 Design constraints and requirements

3.1.1 Functional requirements

As briefly described in the introduction, the main requirements for the project explained in this thesis are the following:

Automatic pod creation according to template configurations

The main goal of this project is the automatic creation of laboratory environments for students according to a configuration template. These laboratory environments would have all software, scripts, and files needed for the students to take their laboratory during the time it is available. They should be automatically provisioned and removed, requiring minimal interaction from the teaching assistants to handle their creation and deletion.

The most important part of this approach is to ensure that there is no configuration mismatch between the student laboratory environment and the environment the teaching assistant used to design the laboratory.

Strong and flexible access control

The laboratory environments created should only be accessible to students who have explicit access to that particular laboratory environment. This can concern individual students, but also enable the creation of groups of one or more students which would all have access to the same laboratory environment.

Teaching assistants should always be able to access all student laboratory environments for help or monitoring.

Creation of Services

It should be possible for teaching assistants to create Services available for all students, e.g. a web server with a SQL injection vulnerability through API access.

These Services can only be available from within the cluster, but it should also be possible to create Services accessible with a web interface through an OpenShift Route (e.g. for CTF servers such as CTFd [11]).

Virtual machines deployment capabilities - withdrawn

Initially, a part of the functional requirements was to permit the creation of virtual machines instead of pods to directly and easily allow for cluster-secure laboratories on exploits that require kernel level access to be studied in laboratories.

There were discussions with the cluster administrator to explore together the possibility of adding OpenShift's own solution for virtual machines to be created and managed similarly to pods inside of the cluster, OpenShift Virtualization [12], to be added to the cluster. It would allow the cluster to have the same capabilities as a Kubernetes cluster with the KubeVirt extension [9] regarding the management of virtual machines. However, it would require the addition of a new *bare-metal* worker node, which means a physical machine added to the cluster. This would go against the philosophy of using underexploited resources of the university, since such a machine would be paid for and maintained by the university, and would be problematic regarding performances in case many students take a lab with a virtual machine at the same time, since all virtual machines would run on that bare-metal worker node. In contrast, pods can be created on any worker node in the cluster, allowing better performance through horizontal scaling.

Ultimately, this requirement was withdrawn, but since the way OpenShift Virtualisation handles virtual machines is very similar to pods, adding the possibility of creating virtual machines for laboratories could come at a minimal modification cost, should the cluster gain the ability to create virtual machines in the future.

3.1.2 Non-functional requirements

Security

The system should be as secure as possible, making sure that users are completely unable to access laboratory environments they should not have access to.

The system should always use the least-privilege principle and use available OpenShift security and isolation capabilities to enforce these rules, such as Namespace separation, and most importantly proper RBAC configurations.

Scalability

The system should be efficient and scalable, as there is a possibility that many students will take their laboratory at the same time. Sensible cluster resource limits should be used to ensure a smooth experience.

Reproducibility and Idempotency

The system should ensure that a setup created by a teaching assistant runs exactly the same whether it is run as a test by the teaching assistant or as the official laboratory.

The system should also follow the idempotency principle pushed forward by Kubernetes and OpenShift, ensuring that the interface does not interact with the cluster in an imperative way but allows for idempotent interaction, meaning that if the same instruction is given twice (e.g. deploy the laboratory named "example-lab"), the result is the same as if it had only been run once (e.g. the laboratory named "example-lab" is deployed).

Modularity

The system should be designed in a way that allows easy interaction with it by other programs and should be easily expanded.

3.1.3 Technical constraints

In addition to the cluster's inability to handle virtual machines described in subsection 3.1.1, the cluster comes with additional constraints:

- Permissions given to people who deploy the system in the cluster should be minimal and negotiated with a cluster administrator.
- The cluster authenticates users exclusively through ULiège's single sign on service.

3.2 High-level architecture: two operators

The core functionality of the system is split between two operators: the *core* operator and the *lab* operator. This allows for a significantly strong design and robust separation of concerns.

The responsibility is split between two operators: the core operator, which prioritises robustness, and the lab operator, which is more dynamic and handles a single lab. The core operator manages the lifetime of the lab operators. If a lab operator crashes, it can immediately be recreated by the core operator. Lab operators exclusively have access to one Namespace with RBAC constraints set by the core operator, so there is strong logical separation between the different lab operators should several of them run at the same time. Since they handle a complex task with a higher failure probability compared to the core operator, this design ensures the core operator can easily recreate any failing lab operator.

Each laboratory will take place in a single Namespace. If several laboratories occur simultaneously, it is possible and even advised to use two different Namespaces for the two different laboratories. All pods in the same lab will be created in the Namespace attributed to the particular lab. The lab operator is in charge of creating the right RBAC rules in order to allow students to access their pod and only their pod.

3.2.1 The core operator

The core operator relies on two custom resources to work:

- The *LabGroup* custom resource, whose role is to ensure that the Namespaces in which labs are deployed have the right configuration, meaning that the core operator has the right administrative rights over them, so that it can delegate them to the lab operator through RBAC.
- The *Lab* custom resource is the trigger for the creation of a laboratory. If a Lab resource exists, the desired state of the cluster includes its corresponding lab in a deployed state. It designates a Namespace in which the laboratory should be deployed, and the actual configuration of the lab as a ConfigMap.

Once the core operator detects a Lab custom resource, it spawns a lab operator associated with that resource and gives it all the RBAC permissions needed in the target Namespace, as well as a reference to the configuration.

Its logic is designed around a reconciliation loop that observes the current state of the cluster, diffs it with the desired state of the cluster, i.e. should one or several lab operators be present, the core operator

does not concern itself with the actual lab deployment, and takes all the actions necessary in order to reach the desired state.

3.2.2 The lab operators

Once created, a lab operator's goal is to:

1. Parse the given configuration, checking that it follows the correct structure.
2. Create Deployments for the desired pods, with the correct configuration options set.
3. Create the correct rules to configure the Namespace according to the configuration.
4. Watch and revive any component that might crash or stop.
5. When a notification that the Lab resource has been deleted is received by the lab operator, it disposes of all the created resources and shuts down gracefully.

The lab operator is the component that creates the RBAC rules that ensure correct isolation between students. It also relies on a reconciliation loop, more complex than the core operator reconciliation loop, since it has to manage and reconcile many pods and configurations.

Although the lab operator is not an operator in the strict sense of the term, since it does not technically define its own custom resource and is a "controller" in the strict sense, it will still be referenced as the lab (child) operator going forward. It still accomplishes most of what an actual operator would do, but it relies on ConfigMaps instead of custom resources.

3.3 Overview of the system design and users

Figure 3.1 shows a very simplified overview of how the different users interact with the system: admins, teaching assistants and students. Details of these interactions and what they trigger in the system are given in the following subsections. A more in-depth and comprehensive diagram can be seen at the end of the chapter with figure 3.2.

3.3.1 Simplified overview of the user interactions

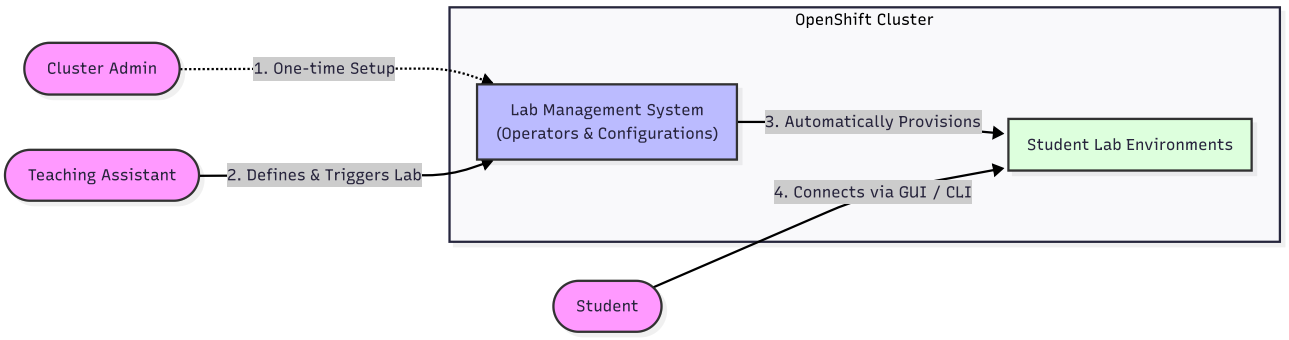

Figure 3.1: Diagram of basic user interaction.

3.3.2 Admin tasks

The only tasks the admin is required to take in this system were the initial creation of the custom resource definitions for the Lab and LabGroup custom resources, but this only has to be done once, as these definitions live at the cluster level and the attribution of Roles and RoleBindings to allow teaching assistants

to manage the needed custom resources. These Roles can be created either in a specific Namespace or cluster-wide with a ClusterRole and ClusterRoleBinding. To respect the principle of least-privilege, creating a Namespaced Role in the Namespace of the teaching assistant's choice is the best way to do it since these permissions are only needed to be given once even if the teaching assistant wants to deploy laboratories in several different Namespaces at the same time, since the custom resources are only created in a single Namespace, where the operators and configmaps live as well.

If someone new wants to be able to use labs, the admin or someone who has a ClusterRole bound allowing them to manage the custom resources in all of the cluster has to create a new Role and RoleBinding (or a new ClusterRoleBinding).

3.3.3 Teaching assistant tasks

Teaching assistants are the ones who actually deploy the core operator and handle the creation and management of the actual Namespaces where the labs will be deployed. Although it was envisaged to let the operators handle this task, it necessitates very high permissions over Namespaces and Projects cluster-wide since it needs to create *and delete* Namespaces. If a bug was present in the operator, it could accidentally delete a whole Namespace, and by extension all the resources present in it. Although this risk can be severely mitigated using a validation admission policy [13], this is still very delicate, and it was considered preferable to let teaching assistants do namespace-related tasks themselves, as it allows removing a very sensitive privilege from the operators, in favour of the least-privilege principle.

As a note, the finished system still enforces that the Namespaces used for laboratory deployment have the "lab-" prefix to ensure as much as possible that Namespace use for laboratories can not be accidental.

Teaching assistants can create one or more LabGroup custom resources, so that the Namespaces pointed to by the LabGroup custom resource are considered as available for labs by the core operator. If a Namespace is not correctly set up and does not have the correct RBAC rules set in place to allow the operators to deploy a lab in that Namespace, a clear warning message is issued in the core operator logs. When marking the Namespace as available for laboratories in a LabGroup custom resource, teaching assistants should make sure to give the correct permissions to the core operator's ServiceAccount. The correct permissions are shown in the appendix C.

At any point, teaching assistants can create ConfigMaps that describe the configuration of a specific lab in the operator Namespace, without triggering anything.

The actual trigger of laboratory deployment is the creation of a Lab custom resource that will engage several steps to occur:

1. The core operator's lab reconciliation logic triggers. After checking that the laboratory is in an accessible Namespace, the core operator spawns a child lab operator and points it to the target Namespace and ConfigMap.
2. The spawned lab operator takes the steps described in section 3.2.2.
3. All lab environments are running and accessible according to the configuration.

As a note, by default, all pods running inside of a Namespace are visible to people that have the admin Role inside of that Namespace. In this case, the teaching assistant that owns the Namespace in which the laboratory environments are created *always* has full access to all the pods that are spawned in it, including the deployed laboratory environments. The configuration allows adding other users to a monitor list, to allow other users, e.g. professors, other teaching assistants, or student monitors, access to the student

laboratory environments without granting them the Namespace admin Role.

In the event of the triggering of the deletion of the associated lab custom resource by the teaching assistant, the entire laboratory deployed is deleted, and the core operator deletes the associated lab operator. Details on how this is performed can be read in section 4.4.

It is also possible for teaching assistants to define startup and end scripts that will respectively trigger on container startup and container deletion. Their results can be saved inside of ConfigMaps, accessible only to the teaching assistants, even after lab completion.

3.3.4 Student tasks

One of the goals of this system is to allow students to have the easiest and most straightforward experience possible. In that regard, once all the setup is complete and a lab is deployed, they can access their attributed laboratory environment directly through OpenShift's console on their browser.

By default, the students do not have access to the whole Project itself and have no direct path through the UI, but they do have access to the page allowing them to interact with the terminal of any container inside their attributed pod. The url of this page can be predicted using the following form:

```
https://console-openshift-console.apps.speam.montefiore.uliege.be/k8s/ns/<namesp
↪ ace-name>/pods/<general-pod-name>-<team-id/student-id>/terminal
```

Only the allowed students have access to it if they are identified in the cluster.

However, a simple parameter allows students to have a clear UI path from the root of the OpenShift console UI, but it requires that students be able to list all pods running in the laboratory Namespace. Even though it does not give them access to the terminal interface of the other students pods, the choice to enable this is left to the teaching assistant.

It is also possible for students to use their own terminal emulator installed on their machine. It requires them to have either the `kubectl` or the `oc` command line tool installed. Although `oc` is OpenShift native and provides a superset of the capabilities offered by `kubectl`, the Kubernetes tool, here `kubectl` is sufficient. It also requires the student to be logged in the cluster. This can be done with a token given to students through the OpenShift console web user interface. Once logged-in, and if the related Lab resource is effectively deployed, students can access their laboratory environment terminal via this command:

```
$ oc exec -it <general-pod-name>-<team-id/student-id> -n <namespace-name> --
↪ /bin/bash
```

The above command assumes that bash is available in the provided image. If not, the students can use `/bin/sh` instead, since this is the default shell used by the OpenShift console.

Once the Lab custom resource has been deleted, the pods are immediately made unavailable. Although this might not trigger an immediate refresh of the console web interface, this will prevent any student interaction with the lab environment going forward.

3.4 Detailed diagram view of the system

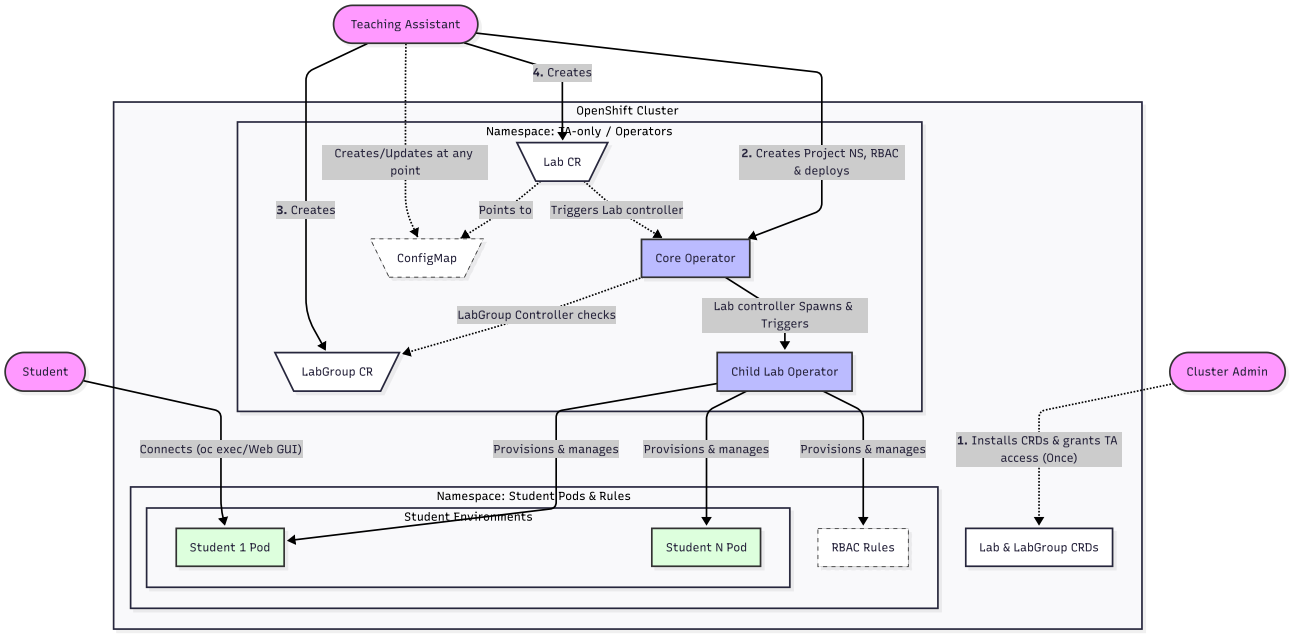

Figure 3.2: Diagram view of the overall system organization.

Chapter 4

Implementation

4.1 Development stack and tooling

4.1.1 OpenShift Operator SDK

Both the core operator and the lab operator were built using OpenShift's operator software development kit (SDK) [14]. OpenShift's operator SDK was chosen because it is the "OpenShift way" to create custom operators. It comes with tooling and documentation and provides functionalities such as:

- Easy tools for scaffolding using the command line.
- Higher level APIs and abstraction logic to allow for easier logic creation.

The scaffolding of both of the operators was created using the following command in their respective folder:

```
$ operator-sdk init --domain labs.montefiore.uliege.be --repo
↪  gitlab.uliege.be/tfes/lab-revamp
```

The OpenShift operator SDK allows for three main ways of implementing the operator logic:

- Through Go code: this is the most "traditional" way and allows for complex logic and state resolution within the Operator.
- With Ansible playbooks and roles: Ansible playbooks can be viewed as a set of instructions to execute should some conditions be met. It allows for relatively complex reconciliation logic and permits imperative logic.
- With Helm charts: this is a *strictly declarative* approach and allows for very simple operator creation. Although it is very simple to use, it has limited functionality because of the strictly declarative nature of helm charts.

The choice was made to use Go for both operators. Although Helm is closest to the declarative philosophy around OpenShift and is very easy to understand, it does not allow for complex enough logic, at least for the core operator. The following table shows the advantages and drawbacks of using Ansible playbooks vs using Go:

Table 4.1: Comparison of advantages and drawbacks of using Ansible VS Go.

	Ansible	Go
Advantages	- Standard, structured presentation of logic - Quicker development cycle	- More flexibility and more complex logic - Better runtime efficiency
Drawbacks	- Debugging is harder than in Go - Becomes hard to read for very complex logic - Performance overhead	- More complex to develop

In the end, because of the reasons listed above along with a better familiarity with Go, and for consistency across both operators, Go was chosen as the main tool for operator development.

4.1.2 Deployment of the operators

There are several ways to use the operators created with the OpenShift operator SDK. When performing the initial scaffolding for the operators, a makefile is created with a `deploy` target. This target allows for relatively straightforward deployment of the operator through the generation of manifests applied to the cluster.

OpenShift provides another solution: the Operator Lifecycle Manager [15]. It provides some advantages, such as safe updates and rollback in the case of errors and dependency resolution. OpenShift's operator lifecycle manager is mainly aimed towards publicly available operators.

Ultimately, since the developed operators do not necessitate any specific dependency besides what is in a base OpenShift cluster, and because it comes with significant development overhead and necessitates more cluster permissions to deploy, the choice was made to use the more simple and straightforward `deploy` target of the makefile, relying on OpenShift's `oc` command line tool. With it, deploying the operator is as simple as using the following command:

```
$ make deploy
```

This uses the image registry defined in the operator internal files in `config/manager/kustomization.yaml` to be deployed on the cluster, but this image is not updated automatically when deploying. Actually, building and updating the image can be done with the following make targets:

```
$ make docker-build
$ make docker-push
```

It is advised to tag the image with a version number or a timestamp to avoid caching issues when updating the operator. The following command builds the operator image, pushes it to the image registry tagged with a timestamp and deploys that image to the cluster correctly.

```
$ make docker-build docker-push deploy IMG=<img_registry_url>/<img_path>:$(date +%s)
```

The `deploy` target will automatically create all the resources needed for the operator to work correctly (Roles and RoleBindings in particular). If the user doesn't have the correct permissions, the command will throw an error listing the missing permissions. This can happen if the user hasn't received permissions over the custom resources handled by the operator for the target Namespace.

4.1.3 Miscellaneous tooling

OpenShift CRC

Although most of the later tests were done on the university's OpenShift cluster after talking with the cluster admin to deploy the required custom resource definitions and RBAC rules, the earlier tests were done locally using OpenShift CRC (CodeReady Containers) [16], also known as *RedHat OpenShift local*.

CRC allows creating and managing a minimal or full-fledged OpenShift cluster running exclusively on a local machine. It allowed for early testing in a non-critical environment, which makes it possible to deploy and modify custom resource definitions and ClusterRoles/ClusterRoleBindings since it gives full admin access to an OpenShift cluster. This allowed for better familiarity with the tools and permissions needed to deploy an operator on the cluster.

However, it has several disadvantages and is very slow and resource intensive on old computers and can easily trigger out-of-memory errors when run on a computer with 16 Gigabytes of RAM or less. Internally, CRC spins up a full-fledged virtual machine, which can necessitate a lot of disc space and computing power.

As soon as what needed to be deployed on the cluster was clear, further testing was done exclusively on the university cluster, although some adaptation was needed to fit the more restrictive configuration of the university cluster.

GitLab as git repository and image registry

Since the university also hosts its own GitLab [17] instance, this was the tool used to host the remote git repository associated with the thesis.

Use of generative AI

Generative artificial intelligence tools were used for several purposes:

- For research and documentation
- Code documentation
- Debugging and autocompletion
- Bibliography entry formatting
- Proofreading

The tools used for the purposes described above are Gemini [18], GitHub Copilot [19] and Perplexity [20].

Generative AI was not used for direct writing of this thesis, although it helped find minor mistakes during proofreading.

4.2 Proof of concept

Before all architectural choices were made, the possibility of a system with the requirements presented in section 3.1 was studied and a proof of concept was developed. This proof of concept is still available in the git repository under the `proof-of-concept` branch.

The proof of concept consists of a single operator with a single custom resource definition, `LabSession`, which directly includes the basic configuration of a single pod to run. All was done in a single Namespace, and running several pods available for different students necessitated the creation of several different custom resources, one for each distinct laboratory environment.

It allowed us to get familiar with basic OpenShift operator concepts and to show which RBAC rules are needed for a configuration where all students have access to only the pods they have explicit access to, and to show that these configurations can be handled automatically.

4.3 Implementing the core operator

This operator defines two different custom resources and has a controller to handle both types of custom resources.

4.3.1 LabGroup custom resource

Custom resource definition

The complete custom resource definition is given in appendix D.1, but here is the most important part:

```
// LabGroupSpec defines the desired state of LabGroup.
type LabGroupSpec struct {
    // GroupName is the name of the lab group, for example a professor's name.
    GroupName string `json:"groupName,omitempty"`

    // Namespaces is the list of namespaces that belong to this lab group.
    // +kubebuilder:validation:Required
    Namespaces []string `json:"namespaces"`
}
```

The other parts of the file are mostly automatically scaffolded fields. It shows that, in essence, this custom resource is very simple and consists of a named list of Namespaces.

This custom resource definition in go contains a `+kubebuilder` tag, also known as a marker comment. The written go code is not what is directly uploaded to the cluster. Instead, when using some `make` targets, go files are automatically scanned by `controller-gen` [21], a purpose-built CLI tool that can generate the YAML files that will actually be deployed in the cluster. The `kubebuilder` marker comments allow us to specify constraints like validation constraints, indications for the structure of the created YAML and tools that can be used, as well as RBAC permissions that must be given to the operator's ServiceAccount.

In the go code shown above, a marker comment was used to make the Namespace list required for the creation of a LabGroup custom resource.

Controller logic overview

The controller for this custom resource is also very straightforward. For any existing LabGroup custom resource in its Namespace, the controller goes through the list of Namespaces and verifies that each Namespace is correctly configured, meaning that it has the correct RBAC permissions set for the core operator ServiceAccount.

Since this controller runs inside of the core operator itself, it can use self subject access review to securely verify if the required permissions are given to the operator ServiceAccount. Self subject access

review simply means that the operator asks the cluster "Do I have that *verb* (permission) over this *resource* ?". If any missing permission is detected during the self subject access review for any Namespace, a warning with the concerned Namespace and permission is issued to the user.

For convenience, a simple YAML file provided in the system's repository can be applied to the cluster to create a Role with the correct permissions all at the same time to the targeted Namespace. This file is available in the appendix C. It can be created with:

```
$ oc apply -f core-operator-delegation-role.yaml -n <target-namespace-name>
```

Once the Role is created, it must be bound to the core operator ServiceAccount:

```
$ oc create rolebinding lab-core-operator-delegation-binding
 ↪  --role=lab-core-operator-delegation-role
 ↪  --serviceaccount=<operator-namespace-name>:core-lab-operator-controller-manager
 ↪  -n <target-namespace-name>
```

Once this setup is done for all Namespaces in the LabGroup custom resource, the setup is complete.

4.3.2 Lab custom resource

Custom resource definition

The complete resource definition is given in the Appendix D.2. In itself, the Lab custom resource definition is very simple too, as seen in its main fields:

```
// LabSpec defines the desired state of Lab
type LabSpec struct {
    // LabGroupRef references a LabGroup that defines which namespaces
    // this lab can be deployed to
    // +kubebuilder:validation:Required
    LabGroupRef corev1.LocalObjectReference `json:"labGroupRef"`

    // NamespaceName is the name of the namespace to use for the lab.
    // Must have "lab-" prefix.
    // +kubebuilder:validation:Required
    // +kubebuilder:validation:Pattern=`^lab-[a-z0-9]([-a-z0-9]*[a-z0-9])?$`
    NamespaceName string `json:"namespaceName"`

    // ConfigMapRef references the ConfigMap that will be accessible
    // by the child lab operator in the target namespace.
    // +kubebuilder:validation:Required
    ConfigMapRef ConfigMapReference `json:"configMapRef"`
}
```

There are 3 important fields in this Lab custom resource:

- A LabGroup reference.
- A name for the target Namespace in which the laboratory should be deployed.
- A ConfigMap reference that points to the actual configuration of the laboratory.

Marker comments are used to ensure that Lab custom resources always have those 3 fields set and that the Namespace is effectively prefixed with "lab-".

Controller logic overview

As the core operator has to be very robust, this controller also has a very straightforward task.

1. Checks that the LabGroup controller has marked the Namespace as valid, meaning it has its permissions correctly set up.
2. Checks that the ConfigMap containing the lab configuration exists, but does not parse it.
3. Creates the RBAC Role and RoleBinding so that the lab operator ServiceAccount (specific for that lab) has all the needed accesses to the target Namespace.
4. Actually creates a Deployment for this instance of the lab operator.

When creating the Deployment for the lab operator, the core operator injects all the needed information like the target Namespace, ConfigMap reference and laboratory name in the environment of the created lab operator.

In case of deletion of a Lab custom resource, the core operator intercepts the deletion using a Finalizer [22], a standard Kubernetes resource that can allow operators to intercept deletion events, and updates the desired cluster state to a state in which the laboratory is not deployed. The core operator signals the lab operator to shut down and once the lab operator has completed its shut down process, the core operator deletes its deployment to remove it completely.

It is entirely possible to have several core operators in different Namespaces in the same cluster, acting completely independently.

4.4 Implementing the lab operator

The lab operator has the harder task of deploying the actual lab and has an inherently higher likelihood to fail due to resource limitations, or faulty configurations.

Controller logic overview

Although not strictly an operator, since it does not rely on custom resource definitions, the lab operator functions exactly like a regular operator, trying to reconcile the current cluster state and the desired cluster state. Therefore, it will be referenced as the "lab operator" throughout this thesis.

In its reconcile loop, the lab operator:

1. Parses the laboratory configuration from the ConfigMap reference given by the core operator. Appendix E shows the supported configuration fields.
2. Reconciles all the necessary resources from the laboratory config. The reconciliation order is as follows:
   (a) Services
   (b) RBAC Roles and RoleBindings for students and monitors
   (c) Pods for laboratory environments
   (d) Network policies

3. Checks for orphaned resources and cleans them up. This can occur if the configuration was modified when the laboratory is deployed. The lab operator is able to diff the configuration and the current state and only create/delete the resources that need it.
4. Collects the result of any pod with a startup or end script and inserts it into a ConfigMap, only accessible to the teaching assistant.

To ensure that no rogue student pod brings the cluster to its knees by hogging resources, a default CPU and RAM limit is set on all pods. It can be modified by the laboratory configuration.

During graceful shutdown, after the operator creates the teardown signal as a config map, the lab operator cleans up the target Namespace, including all pods and configuration, and shuts down. Once the lab operator has finished and is effectively terminated, the core operator deletes the lab operator deployment and cleans up the permissions it gave to the lab operator. Once done, the lab custom resource is effectively deleted. In case the child lab operator stalls during shutdown, the core operator implements a grace period. If this grace period is exceeded, the core operator forces the termination of the lab operator.

4.4.1 Start scripts and end scripts

Kubernetes offers a native way to handle setting scripts that should run at container start and just before container end via lifecycle hooks [1], `PostStart` and `PreStop`. However, this approach has a major drawback in our case: the lifecycle hook scripts that are run on the pod are visible on the pod's description. If we were to use this solution to provide the ability for teaching assistants to setup the pod depending on available environment variables, it would make the scripts completely transparent to the students. While it could be fine in some cases, we assume that the setup of the laboratory and the eventual pre-grading logic in the scripts need to stay hidden from the students.

In our case, it is done through the lab operator, which uses the Kubernetes API to gain exec access into the pods. It allows using the set environment variables. That way, the scripts can stay exclusively in the laboratory ConfigMap, in the operator namespace, completely inaccessible to students.

However, it should be noted that for scripts that run for more than a few seconds, a student having access to the pod could identify it and watch it, since all processes on a pod with the default `restricted-v2` SecurityContextConstraint run under the same user id. To prevent it for start scripts, the `globalStartTime` parameter can be set in the lab config to allow pods running their start script before students gain access to it. For end scripts, `globalEndTime` allows having the same assurance, kicking students out before the end scripts are run when the Lab custom resource is deleted.

While long running scripts are not advised as exec sessions are not as robust as native execution on the pod, since it uses the network, it is possible to extend the default 30s time limit for the scripts by setting the `scriptTimeout` parameter.

Script results are saved to ConfigMaps in the laboratory Namespace, completely inaccessible to students, and are kept even after deletion of the Lab custom resource.

4.5 Typical lifetime of resources used by the operators

- The *LabGroup* custom resource is a resource that can stay permanently in the operator Namespace. If several teaching assistants share the same core operator, best practice would be for each of them

[1] https://kubernetes.io/docs/concepts/containers/container-lifecycle-hooks/

to have their own LabGroup instance, allowing them to each have their own list of Namespaces available for labs. If not, a single LabGroup instance is sufficient.

- The *Lab* custom resource defines the desired cluster state as a state in which the associated lab is deployed. Although it doesn't necessarily mean that the students have directly access to their laboratory environments if `globalStartTime` is set, it should not exist for more than a few hours, during which the lab is taken by the students.
- The *ConfigMap* containing the actual laboratory configuration needs to exist as long as the associated Lab custom resource exists in the cluster. Besides this, teaching assistants are free to do as they wish, delete them after the lab or keep them in the cluster for the next year, updating the student list.

4.6 Events

Both operators publish Events tied to the resource they are managing.

- The core operator can publish events tied to:
  - LabGroup custom resources when checking that the LabGroup resource references correctly set up Namespaces
  - a Lab resource when the core operator reconciles a Lab custom resource and sets everything *before* starting a lab operator
  - a specific pod, the pod where a lab operator will be running, when deploying it
- The lab operators publish events tied to the ConfigMap they are using to spawn the lab

These events can be for the normal course of events when deploying a laboratory, but can also inform the teaching assistants of errors, e.g. in case a configuration file can not be parsed correctly, in an accessible and centralized place. Although Events are there and are usually enough to determine the source of the problem, users can still access the logs of the different operators which include more complete messages.

4.7 Expansion of the capabilities through custom SecurityContextConstraints

The university's OpenShift cluster default SecurityContextConstraint is very restrictive. This is a standard SecurityContextConstraint, `restricted-v2`, and makes it so that pods created on the cluster have very little room when it comes to anything that might be exploited to gain root access.

Other standard SecurityContextConstraints exist, but they come with increased security risks, and giving some of the capabilities given by those less restrictive SecurityContextConstraints to many student pods should not be done.

However, it is possible to create custom SecurityContextConstraints. Appendix F shows such a case: this SecurityContextConstraint definition is based on `nonroot-v2`[2] but allows for privilege escalation through the use of `setuid`.

[2] For more information on standard SCCs, follow `https://docs.redhat.com/en/documentation/openshift_container_platform/4.16/html/authentication_and_authorization/managing-pod-security-policies`

While the ideal solution for secure close-to-kernel security manipulations for laboratories would be adding support for virtual machines inside of the cluster, adding custom SecurityContextConstraints available to the core operator ServiceAccount is a relatively safe method, even though it necessitates adding an SCC to the server, which is a task that requires admin approval.

In the current configuration, any SCC available to the core operator ServiceAccount will be available to use in lab configurations by the teaching assistant using the `useSCC` setting.

4.8 CLI tool for interaction with the cluster and configuration

A CLI tool was developed for easy scaffolding of configurations and containers, as well as simplified handling of student lists and groups. It can be used to interact with a deployed core operator.

It was first developed in python and used the `oc` command installed on the computer it runs on. However, the choice was made to change it to go for two main reasons:

- By importing Kubernetes native modules in go, there would be no reliance on the `oc` command
- It would also allow using the lab operator's existing config parser to offer validation before deploying a lab.

The re-written script exploits the `~/.kube/config` file, also used by `oc` or `kubectl` to authenticate as the user in the cluster. Its configuration is stored in `~/.config/labctl/config.json` for Linux (and equivalent default user config folders for Windows and Mac) and remembers information about which Namespace is used for operators and the deployment of configuration.

This command line utility offers the following functionalities:

- `setup`: allows changing the default Namespace and service account for the core operator.
- `scaffold-dockerfile`: creates a simple dockerfile with a `/lab` folder writable by students, complying with the `restricted-v2` SCC limitations. The default image the container is derived from is Ubuntu 26.04.
- `init`: scaffold a lab configuration and a Lab custom resource as YAML files. This command allows listing available LabGroups custom resources on the operator Namespace and choose the laboratory Namespace from the available Namespaces, or create a new LabGroup resource.
- `import-students`: import a list of students or groups from a CSV (or TSV) file to a config, either a local config file, or a config that exists in the operator Namespace. Students can either be appended to pre-existing students, or replace them.
- `add-scripts`: import start/end scripts into the configuration, either on a local file or directly in a config present on the cluster.
- `validate`: validates the configuration using the parser that the lab operator uses to check for configuration errors.
- `deploy`: applies the provided configuration file or files describing a Lab custom resource, a LabGroup custom resource or a laboratory configuration ConfigMap and deploys them in the cluster. It automatically validates any ConfigMap before applying it to the cluster. If no file name is provided, it will search in the current folder for `lab-config.yaml`, `lab.yaml` and `labg-roup.yaml`.
- `status`: get the status of the deployed Lab custom resources.

- `teardown`: delete a Lab custom resource from the cluster to end a lab. This can be a local file, or if none is specified, the user chooses among the Labs deployed in the operator Namespace.

Chapter 5

Use cases and evaluation

5.1 Practical example scenario

Here is a realistic example of how labs could take place from start to finish assuming that the cluster is already set up and the teaching assistant deploying the lab already has set up a Namespace where the core operator is running. This is a one-time set-up and should not be repeated in normal operation.

The lab setup here consists of an individual lab in which a blockchain is running in the background. This blockchain is primed on container start with 5000 random transactions in the history for a "realistic" blockchain. Since it may take 2 or 3 minutes to completely seed, it is better for the teaching assistant to launch the pods in advance and set the `globalStartTime` parameter so that students only gain access to the pods once the specified time is reached.

5.1.1 Choosing the Namespace for the laboratory

The teaching assistant may already have one or several dedicated Namespaces for laboratories. A likely scenario is to have one Namespace for each course, reusing it for every lab in the course. If this is already set up, the teaching assistant can go straight to the steps described in section 5.1.2.

Otherwise, a new Namespace can be created through the creation of an OpenShift Project, using OpenShift's console user interface, or with the `oc` command:

```
$ oc new-project <name>
```

The only constraint is that the name must begin with the prefix "lab". A display name and description may also be chosen without any constraint.

Once the Project is created along with its associated Namespace, the teaching assistant has to give the correct permissions to the core operator ServiceAccount. As described in section 4.3.1, this can be done with two simple `oc` commands.

Once the permissions have been correctly set up with the Role and RoleBinding to permit access to the Namespace by the core operator, the teaching assistant can grant its use for laboratories managed by the core operator through the creation or modification of an existing LabGroup custom resource instance, pointing to the created Namespace.

Once this is done, the core operator should show "All namespaces have proper permissions" in its logs.

5.1.2 Creating the lab configuration

The teaching assistant may use container images from public repositories and customise them through environment variables set in the lab configuration, but using a custom container image allows for the most configuration. In this example, the teaching assistant uses a light alpine base and installs Ganache [23] on it, for the creation of the blockchain. A startup script is inserted, first running a seeding command for the blockchain, then keeping the pod alive.

As this lab will use the default SecurityContextConstraint, everything on the pod, including the student shell, will run with a unique random user id attributed by OpenShift, but always part of the root group 0. It is a good idea to create a work directory with the correct permissions so that the students can have access in the container file:

```
RUN mkdir /lab
WORKDIR /lab

RUN chgrp -R 0 /lab && \
    chmod -R g=u /lab

ENV HOME=/lab
```

A full container file is shown in appendix G.1.1 along with the associated scripts.

The teaching assistant can upload their private image to the image registry of their choice, and can allow fetching the image from the cluster by creating a pull-secret for that registry inside of the *target* Namespace.

Once that image exists, it can be used in the actual configuration of the lab. The configuration can be built by creating a ConfigMap following the documentation. For this laboratory, a few important settings are:

- `globalStartTime`: This setting allows fixing the time (by default, the used time zone is Europe/Brussels) at which the pods become available to the students, otherwise the pods become available on their creation. This allows the pods to seed the blockchain (around 2 minutes).
- `allowNamespaceView`: This allows students to have a path through the UI from the OpenShift console root. This has the side effect that the students can list all pods that exist in the Namespace, but they still can only interact with their own pod.
- `imagePullSecret`: since this lab uses a custom image, it is important to specify a correct pull-secret. Without it, the pods will not be able to pull the container image.
- `containerPort`: a port is exposed for the ganache API. This port is only available from within the pod by default, but a setting can enable communication between the pods of different students if desired.
- `ifList`: a list of OpenShift subjects (usually user ids) that need to have a pod configured.

The configuration map can be created at any time before the laboratory is effectively deployed, and can stay permanently on the cluster. A full configuration map for this lab is shown in appendix G.2.1.

What actually triggers the creation of the pod is the creation of the Lab custom resource as shown in appendix G.2.2. To ensure proper seeding of the pods, with `globalStartTime` set to the correct time, the Lab custom resource can be created in advance. While it can be done a long time before the actual

start of the lab, it is best to refrain from deploying it too long in advance since it provisions the pods and uses cluster resources.

As soon as the start time is reached, the lab operator gives RBAC access to the students for their respective pods. If the Lab custom resource is created after the start time, the pods become available as soon as they are created.

Once created, the student will be able to navigate to their pod terminal with the OpenShift console browser interface:

Projects ⇒ lab-namespace ⇒ Pods ⇒ attributed-pod-name ⇒ Terminal

Their attributed pod name always contains the user id or group name. Figure 5.1 shows the view that a student has when accessing the pod. By default, the students can not view the logs or events linked to their pod, but log access can be given by the teaching assistant. It is however impossible to hide the other fields, particularly the environment, to the students, since the student needs the `get` verb on the pod in order to open a terminal session through the web interface or via `oc`. This inevitably gives access to the whole pod yaml.

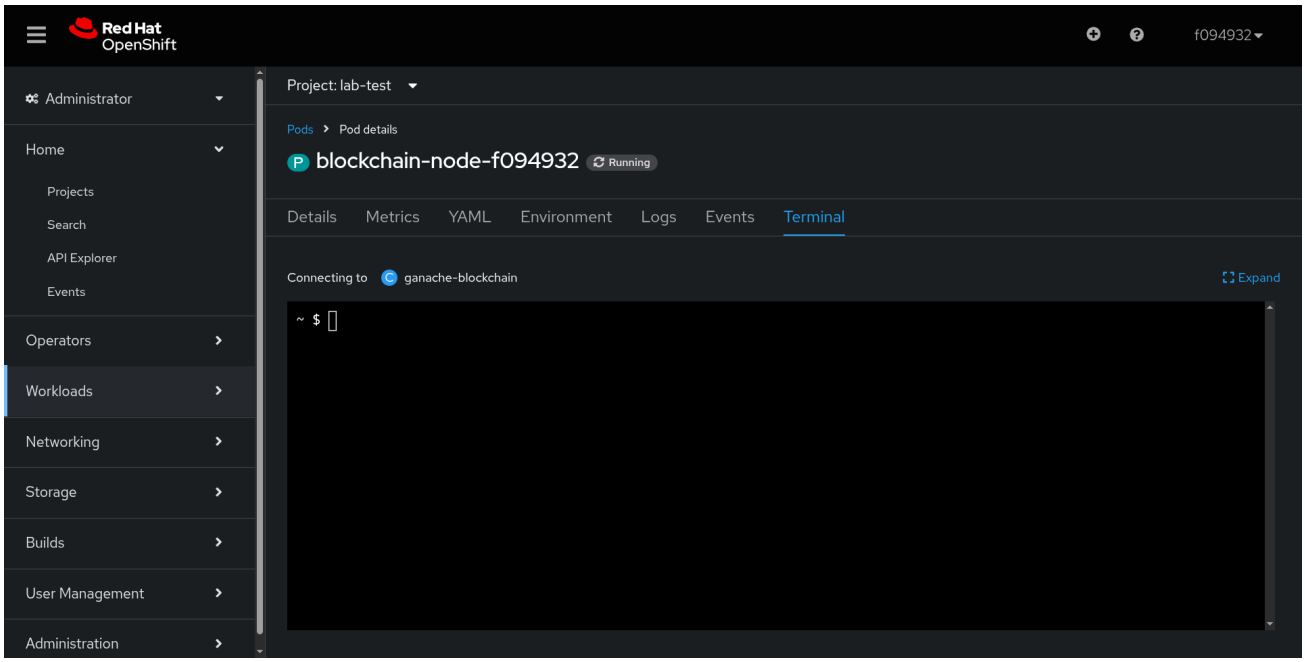

Figure 5.1: Terminal view of a student.

If any modification needs to be made, for example adding a student that was not in the initial configuration, the teaching assistant can simply modify the configuration and reapply it to the cluster. The change will immediately be picked up by the lab operator which will diff the current state and the configuration, and only make the necessary changes. Other student pods will keep running as usual.

Once the laboratory is complete and students were able to submit their answers, the laboratory can be torn down by the deletion of the lab custom resource associated to the lab. The ConfigMap containing the actual lab configuration can be kept on the cluster.

5.1.3 Evaluation of realistic performances

Using a dedicated pod in the same Namespace as all pods running, intensive use can be emulated on all pods at the same time, giving a solid estimation of the actual performances of lab environments running on the cluster. Each data point in this section was collected this way over a period of 10 seconds.

The dedicated testing pod runs a script making requests to the laboratory pods via their exposed port. This is only possible because the lab network policy allows it explicitly, usually this setting would be turned off.

Request type 1: 10 Ethereum wallets

The first test is done with a very light workload, asking for the accounts managed by Ganache. This is a very simple operation, and is mainly done to show how much raw traffic can occur.

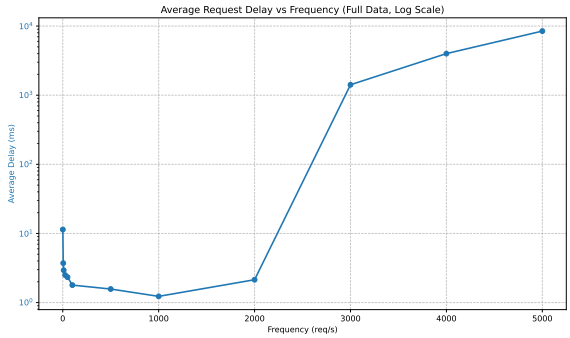

Figure 5.2: Benchmark with light workload, log scale for y.

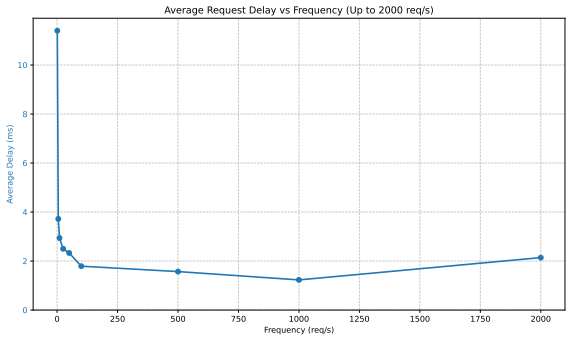

Figure 5.3: Benchmark with light workload, truncated.

This shows that the cluster can handle a lot of traffic. The fact that the latency goes down at first up to 1000 requests per second can be justified by caches warming up. At 2000 requests per seconds and more, there is a very significant increase in latency, with more than a second of delay at 3000 requests per second.

Although this test is most probably bound by the networking between the pods, it is still relevant for labs where pods would connect to a service that they have access to via the cluster network.

Request type 2: entire blockchain history

Since the blockchain is seeded with 5000 transactions, searching its history is a heavy workload. When a pod receives a request, Ganache will search through its history for events. Since standard transactions do not trigger events, Ganache will return a nearly empty message, but will have done the work of going through its whole history, making this benchmark bound by CPU or disk performances.

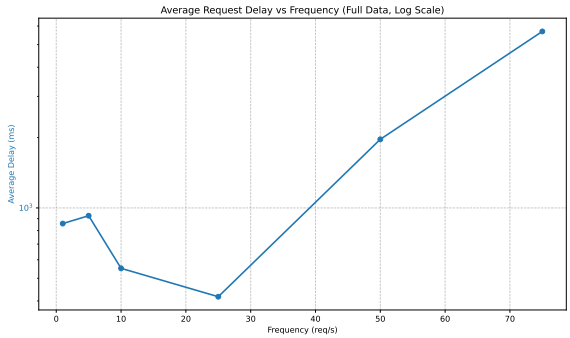

Figure 5.4: Benchmark with heavy workload, log scale for y.

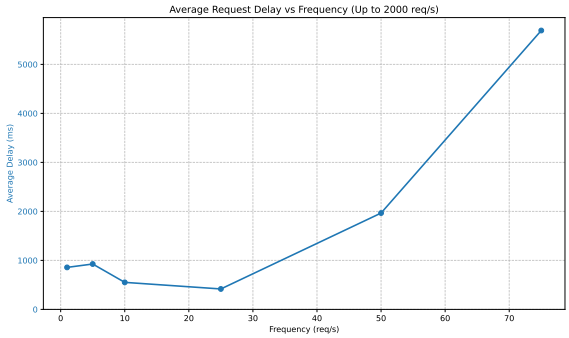

Figure 5.5: Benchmark with heavy workload, truncated.

This test shows that the cluster is sufficient for typical lab use, even with reasonably intensive computation workloads as it is able to answer 25 heavy requests per second in less than 0.5s on average, and even at 50 requests per seconds, the response time stays reasonable at 2s.

5.2 Evaluation of system capabilities

5.2.1 Benchmarking pod creation by the system

To benchmark the capabilities of the system itself, the actual workload running on the pods themselves needs to have as little impact on performances as the data of interest is the time it takes to deploy available pods. Tests were performed to evaluate the horizontal and vertical scaling of the system using very simple student environments: one simple and lightweight alpine container per pod. Pod performance does not depend on the system developed here but only on the cluster capabilities. This section will only talk about performances regarding the system, but real-world usage performance were analysed in section 5.1.3.

The testing is done with a python script running locally and interacting with the university's cluster through the `oc` command. Other clusters might react differently since they might not have the same bottlenecks as the ULiège cluster, but this can help analyse the tendencies of the system and its time complexity given the size of the input.

The time was measured between the confirmation of the creation of the resources creating the laboratory (the creation of a Lab custom resource) and the moment all required pods were marked as "running", meaning that their terminal interfaces are accessible to students.

To improve accuracy, each test was conducted in three separate iterations, with the final data representing the average.

Vertical scaling

The vertical scaling test consists of a single lab with 1 to 350 student lab environments deployed in the same Namespace. Figure 5.6 shows the results of this test.

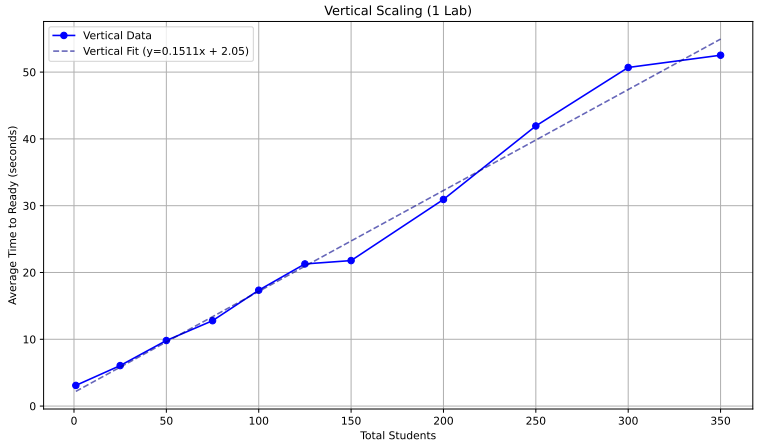

Figure 5.6: Vertical scaling benchmark of the system.

From the results, it is visible that the time it takes to start a typical lab for 50 student environments at the same time is less than 10 seconds. Moreover, the time disparity between the first pod becoming available and the last pod becoming available (spread) is very small, less than 3 second with 50 students environments, which would be a realistic laboratory setup. This suggests the approach is suitable for practical use, since it means that disparity between students regarding the availability of the laboratory environments is minimal.

Figure 5.7 shows the spread for the data presented in figure 5.6. Although spread scales linearly with the number of students in the lab, it stays relatively low for realistic numbers of students, at around 5 seconds for a 100 student laboratory.

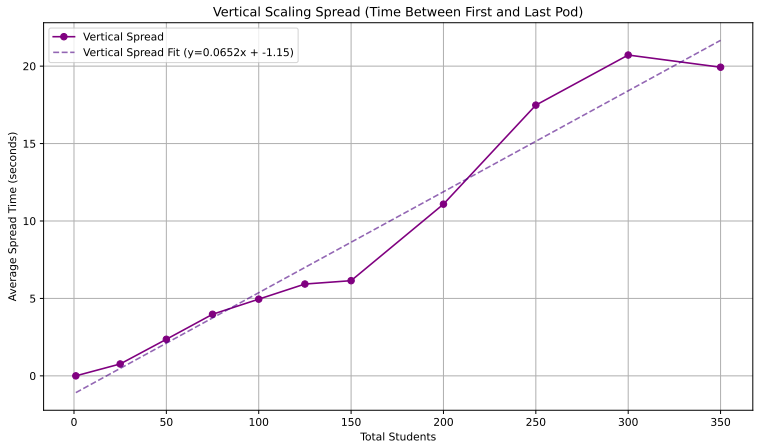

Figure 5.7: Vertical scaling spread benchmark.

Since pods are designed to be very lightweight, setting them up and launching them is not too taxing on the cluster. The images of the containers inside the pods are cached, making sure that pulling the images does not affect spread.

If, in the future, the possibility of launching virtual machines was added to the cluster, it might be necessary to pre-provision them and only give RBAC access to them once the lab is effectively beginning to prevent spread from becoming an issue. Since virtual machines are significantly more resource-heavy than pods, it is possible that the time spread between the first available and last available student environments

would become too big otherwise. Pre-provisioning and adding RBAC rules at the right time would allow overcoming this potential issue easily.

Horizontal scaling

The horizontal scaling test consists of 1 to 7 laboratories of 50 students deployed at the same time. This is a more realistic approach since most courses typically do not have 100 or more students at the same time.

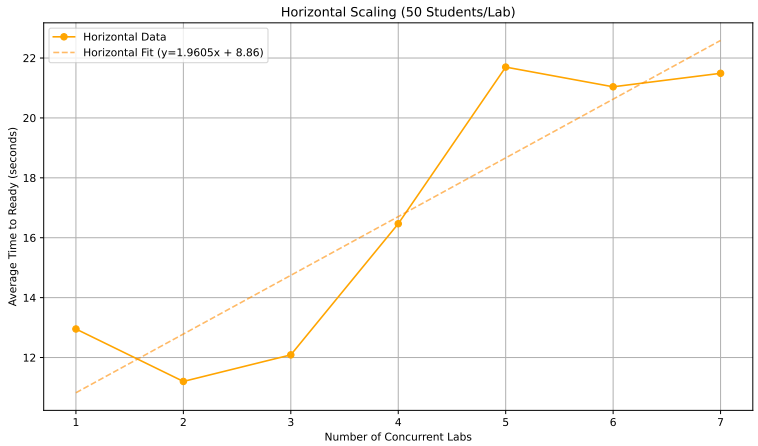

Figure 5.8: Horizontal scaling benchmark of the system.

This graph shows a strong linear relation between the time taken to launch the labs and the number of labs launched, and by extension, the number of student environments deployed. A comparison of system performances between horizontal and vertical scaling is done in section 5.2.1.

Here, three types of spread can be studied:

- The total spread: time between any first pod becoming available and the last pod becoming available, no matter the Namespace. This is relevant if several Namespaces are used for a single laboratory.
- The average in-namespace spread: this is the average spread in all Namespaces.
- The maximum in-namespace spread: This biggest Namespace spread.

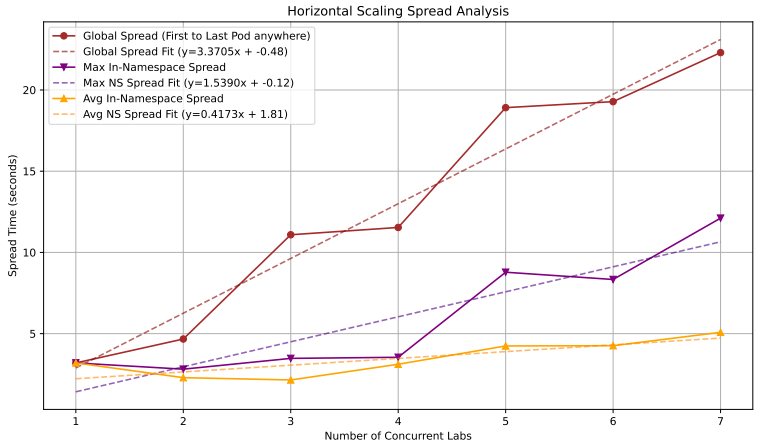

Figure 5.9: Horizontal scaling spread benchmark.

Here it is clear that spread remains at acceptable levels even regarding total spread over the system. However, assuming that a single lab occurs in a single Namespace, and several labs take place at the same time, this is even better: in-namespace spread is significantly lower than total spread.

Horizontal vs vertical scaling

Figure 5.10 shows a comparison of the two scaling types, horizontal scaling and vertical scaling.

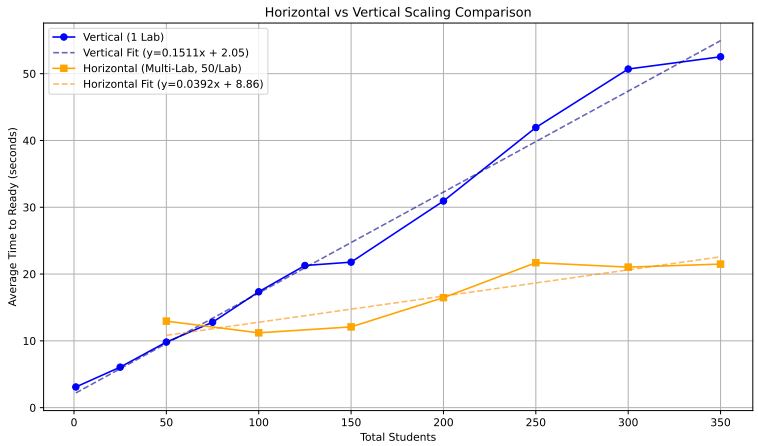

Figure 5.10: Horizontal vs vertical scaling for the system.

The graph shows clearly that horizontal scaling is significantly faster than vertical scaling. Splitting students across multiple Namespaces has significant performance implication. In regular use, this is the expected setup.

This advantage of horizontal scaling over vertical scaling can also be shown with the pod throughput of the system at each data point using the simple formula:

$$T_{\text{pod}} = \frac{N_{\text{pods}}}{t_{\text{deploy}}}$$

Where

- T_{pod} is the throughput of the system on the cluster, in pods per second
- N_{pods} is the number of pods deployed
- t_{deploy} is the time it took between the Lab resource creation and all pods being marked as “running”

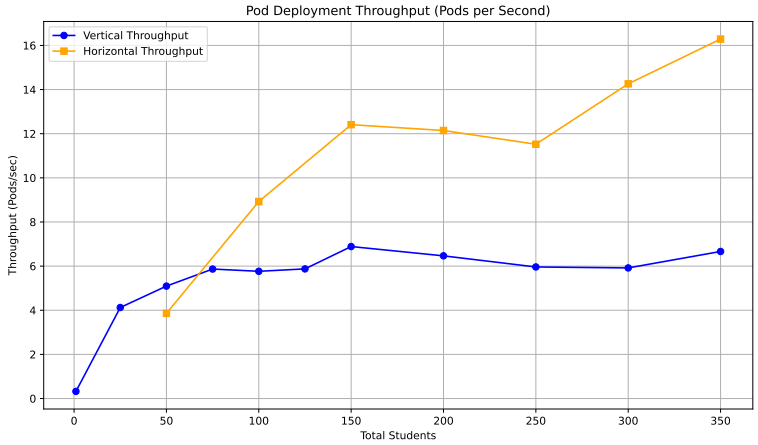

Figure 5.11: Horizontal vs vertical scaling for pod throughput of the system.

Horizontal scaling has a visibly better throughput. This can mainly be explained by the fact that there is one lab operator instance per Namespace. Since horizontal scaling uses several Namespaces, several instances of the lab operator can work in parallel, but in vertical scaling, there is only one lab operator instance handling all the reconciliation work.

The advantage of horizontal scaling is also shown by comparing the spread between the two scaling types as shown in figure 5.12. Although total spread is similar, considering only in-namespace spread shows a very significant reduction of spread using horizontal scaling.

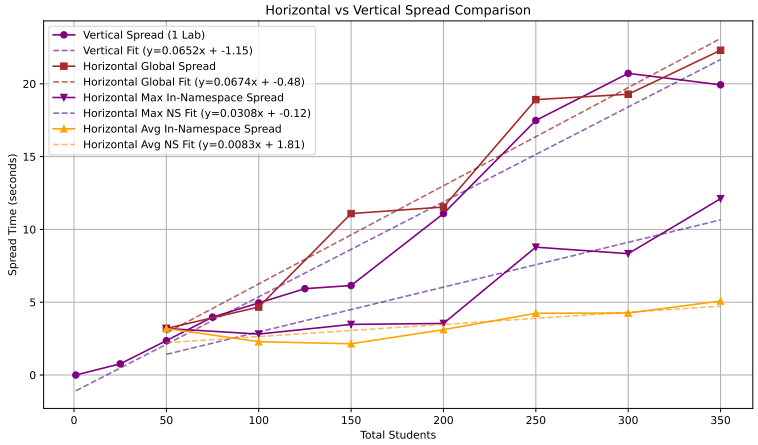

Figure 5.12: Horizontal vs vertical scaling for spread.

5.2.2 Healing capabilities

When a container crashes, it is automatically detected by the cluster, and the pod is immediately restarted. However, if the pod crashes several times in less than 10 minutes, Kubernetes imposes a pause time before trying to restart the pod. This is to prevent repeatedly crashing pods from taking up too many resources. This is expected behaviour and does not penalise pod restart too much on the first few restarts, but it grows exponentially up to 5 minutes [1]. This should not be an issue for regular use of the system, but can

[1] As stated in Kubernetes documentation: `https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#restart-policy`

explain why there is a cool-down for pods crashing repeatedly in a short amount of time.

Table 5.1: BackOff time depending on the number of crashes

Crash number	1	2	3	4	5	6	7+
BackOff time (s)	**0**	**10**	**20**	**40**	**80**	**160**	**300**

Table 5.2: Real BackOff time for repeated crashes

crash number	Avg Time (s)	Max Time (s)	Adj. Avg Time (s)	Adj. Max Time (s)
1	1.95	2.96	1.95	2.96
2	15.25	17.95	5.25	7.95
3	27.66	31.80	7.66	11.80
4	49.14	56.26	9.14	16.26
5	87.91	96.94	7.91	16.94
6	168.12	176.13	8.12	16.13
7	307.50	313.11	7.50	13.11
8	307.90	315.06	7.90	15.06

This data is backed by real-world testing performed by killing the long-running process keeping the container alive and measuring the time it takes before anyone can connect to the terminal of the pod again. Data shown in table 5.2 were collected with this method for the 50 pods containing an alpine container running a sleep command. The adjusted (adj.) time removes the BackOff time from the time it took to restart the pod to get the raw pod restart time, but the actual time a student would have to wait is not adjusted.

5.2.3 Deletion latency

The lab operator, which handles the actual deletion of the pods, cleans the different resources in this order:

1. Roles
2. RoleBindings
3. Deployments (which cascade to ReplicaSets and pods)
4. Services
5. Routes
6. Network policies

This setup ensures that Role and RoleBinding deletion occurs first. However, this only prevents new interactions from occurring and does not stop existing terminal sessions with the pods.

However, the Role and RoleBinding deletion is very fast, and as soon as Deployments are deleted, it cascades to the pods and existing terminal sessions close in around 30 seconds. Table 5.3 shows the time it takes for default pods to become inaccessible after the deletion of the Lab custom resource. The RBAC column shows the time it takes for RBAC rules to be deleted, preventing new interaction with the pod. The exec columns show the time it takes for existing terminal sessions to become inactive.

Table 5.3: Pod inaccessibility time after deletion is triggered.

pod count	RBAC	Avg exec	Max exec
1	0.81	30.62	30.62
5	0.52	30.36	30.37
10	0.59	30.46	30.48
25	0.80	30.79	30.93
50	0.97	30.83	31.12

This table shows that the number of lab environments in the lab has little to no influence over the shutdown time. However, it also shows a 30 seconds delay. This is explained because the keepAlive configuration option which was used to prevent the pod from terminating if it had no running process runs a `sleep` command, which doesn't handle `SIGTERM` commands issued by the cluster when the pod is marked for termination. The cluster has a default 30s grace period during which it waits for the pod to terminate, after which it sends `SIGKILL` and forces the termination of the pod.

This behavior can be changed by forcefully bumping the grace termination period to a lower number, or by using a background command that can handle the `SIGTERM` signal properly. Table 5.4 shows a lab configuration in which the long running command keeping the pods alive handles `SIGTERM`.

Table 5.4: Pod inaccessibility time after deletion is triggered with `SIGTERM` handling.

pod count	RBAC	Avg exec	Max exec
1	0.63	0.46	0.46
5	0.57	0.37	0.38
10	0.57	0.42	0.44
25	0.63	0.62	0.75
50	0.96	0.99	1.32

The table shows that through handling `SIGTERM`, it is possible to have nearly no latency between the deletion of the Lab custom resource and the actual deletion of all associated pods.

Chapter 6

Limitations and possible improvements

6.1 Limitations of the current system

6.1.1 No automatic provisioning of permanent storage

Although it is technically possible to automatically create PersistentVolumeClaims for each student environment, this should be treated with caution. Even with very small claims, the cluster will look for available PersistentVolumes to attribute to the claim. This includes existing very large volumes of several GiB. It can also create new PersistentVolumes if space is available, but depending on configuration, these volumes often have a minimal size of 1 or 2 GiB. Depending on the cluster configuration, PersistentVolumes can use different classes of storage (StorageClass resource). In the ULiège cluster case, there are 3 StorageClasses:

- `thin`: the default storage class, providing virtual drives through the kubernetes.io/vsphere-volume provisioner.
- `segi`: this is similar to `thin` but is reserved for use by IT service of ULiège.
- `sc-local`: this is the rawest form of persistent volumes, directly tying the PersistentVolumeClaim to a physical disc.

After testing, it is clear that while the total space requested by a laboratory of 50 students with a 10MiB permanent volume each would be small in theory, the permanent volumes attributed are very large in size (usually 2GiB, sometimes up to 20GiB). This makes it so that there are not enough resources for all 50 pods. This was done with the default `thin` StorageClass, since `segi` is reserved and `sc-local` ties to the physical drives in the servers and can not offer 50 drives.

6.1.2 Students have the `get` verb over their pod

To allow access to the shell of containers running inside a pod, it is mandatory to give the `get` verb over the pod the user needs access to. This means that the YAML that characterises the entire pod is available to the students that need to connect and exposes the precise configuration of the pod. This includes

- A list of all containers on the pod, with their image
- The base command ran on the pod (e.g., `sleep infinity`)
- The pod environment variables
- Security configuration

- The name of the pod pull secret and of other secrets mounted on the pod.

But it does not expose actual Secrets, only the Secret names are shown, and students can not access secrets on lab namespaces.

It would be possible to overcome this limitation by using another strategy for accessing the pods, for example, using `ssh`, since it would not require students to have RBAC access to their pod. This would come at the cost of not being able to use OpenShift's user interface and authentication, meaning that students have to give passwords or ssh keys, and would require adding a ssh access strategy, either through an ssh bastion or sidecar containers in each pod.

6.1.3 The default SecurityContextConstraint is `restricted-v2`

As discussed above, this restricts the use of some Linux functionalities, especially those close to the kernel. If a lab necessitates some access that the available SCCs can not provide, teaching assistants need to create a new SCC, and negotiate with the admin to allow it on the cluster and give access to the core operator.

This is cumbersome at best, and while many labs do not need any advanced Linux capability, this could be completely overcome by using virtual machines. While it is still possible to run purely software emulated virtual machines inside of the pods themselves, it was not investigated in the scope of this work, mainly since it would be possible to add native virtualisation capabilities to the cluster.

6.1.4 Laboratory participants need an account on the cluster

Since users access their pods through the cluster's own authentication, it means that all laboratory participants must have an account on the cluster. In the case of the ULiège cluster, since new users are added to the cluster if they authenticate with the ULiège SSO and are not a pre-existing user, it is the same as saying all laboratory participants need an ULiège account.

For the purposes of student laboratories, this is perfectly fine, but this means that the system can not be directly used to provide example laboratories to people that participate in the university's open house day, for example.

6.2 Possible improvements

6.2.1 Strategy for permanent storage

Although in the current cluster configuration creating PersistentVolumeClaims for a laboratory with more than 20 students is not recommended, as explained in section 6.1.1, other solutions can be imagined to attach permanent storage for the pods.

For example, a single storage service pod could be used for laboratories that require permanent storage. This pod would require a single PersistentVolumeClaim and would offer dedicated storage spaces that can be mounted by laboratory pods via NFS.

Although implementing that solution or a similar one could add some functionality to the system, it did not enter the scope of the project for two main reasons.

- Involuntary pod crashes rarely occur

- Students can copy and paste data from their terminal session even if the pod has crashed, through their terminal emulator or via the console OpenShift web user interface (before they click the "reconnect" button).

6.2.2 CLI or UI to simplify configuration

Although the command line interface tool mentioned in section 4.8 exists, it can be improved with functionalities that could be used by teaching assistants.

In addition, a real graphical interface could also be created and even hosted on the cluster. While the command line tool uses the `oc` command to interact with the server, a web interface could use OpenShift's OAuth to directly interact with the cluster's API. This would allow for a better centralisation of laboratory related tools and statistics. Although all of these functionalities are available on the cluster, having a centralised place for them is an advantage.

6.2.3 Full virtual machine support

If the OpenShift Virtualization extension were to be added to the cluster, along with the infrastructure needed to run it, this would allow one to spin up virtual machines where students could directly and securely interact with a full kernel. OpenShift Virtualization supports graphical virtual machines, and the OpenShift console browser interface allows for direct access to virtual machines without having to configure the computer accessing the virtual machine.

Since in OpenShift, virtual machines are handled in a very similar way to pods, it would require minimal work on the operators, and the CustomResourceDefinitions powering the system would not need to be modified.

6.2.4 Creating a CI/CD pipeline

Building the operators is a relatively intensive operation on the machine building it. The laptop that was used for the entirety of the development of the system sometimes struggled when building the operator. Since the GitLab instance of the University allows creating CI/CD (continuous integration, continuous deployment) pipelines, it would allow offloading the building costs of the operators on the GitLab runners. For example, any commit pushed on the repository would be validated by building the operators, but only the commits on the main branch would cause the built operators to be pushed on the image registry and deployed on the target namespace.

Compiling on a local machine still has the advantage of giving more direct feedback to the users.

Chapter 7

Conclusion

This thesis has shown the development steps and considerations taken to design a system aimed at improving the laboratory experience for students. It allows avoiding risks of configuration mismatches between the teaching assistants' computers and the students' computers without requiring students to run virtual machines locally. It takes advantage of the capabilities of the ULiège OpenShift cluster and allows defining laboratory configurations following the declarative and idempotency principles of Kubernetes.

The developed solution is based on a robust, two-operator architecture, making sure that the core operator is stable and stays alive, while the lab operator, with a harder task, can be dynamically restarted if necessary. It relies on two custom resources, LabGroup for namespace validation and Lab, which is used to determine the life of laboratory deployments. The actual configuration of the laboratories is contained in ConfigMaps. When a Lab custom resource is created, the core operator spawns a child lab operator that parses the configuration and takes all necessary actions to deploy the laboratory according to the configuration.

The produced laboratory management system can deploy a laboratory with 50 environments in less than 10 seconds, and permits pre-deploying a laboratory, giving access to students at a pre-defined time, at the exact requested second. There can be several instances of the core operator in the cluster living in different Namespaces, but even a shared instance can handle multiple laboratories occurring at the exact same time. The ULiège cluster has good performance even for labs with a heavy workload. Teaching assistants can set scripts at pod start, for setting each student environment individually, and at pod stop, for eventual pre-grading. These scripts and their results are completely hidden from students, and the results can be easily obtained by teaching assistants after the end of a laboratory.

Although some laboratories requiring close-to-kernel functionalities may not be easy to deploy on the cluster, requiring the teaching assistant managing the lab to write a custom SCC and a cluster administrator to review it and add it to the cluster, this setup would not need to be repeated for the same type of laboratories, and the eventual addition of support for virtual machines in the future would completely circumvent it.

The laboratory management system achieved uses modern tools and takes advantage of existing resources, offers practical tools for teaching assistants, and requires no work from students before taking their laboratory. This system has the potential to even differences between the students' hardware, giving them all access to an equal environment. I really hope this system has real-world use, as I am convinced that it has the potential to help students and teaching assistants alike.

Appendix A

Sample Resources

A.1 Role resource example

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: example-role
rules:
  # Core Resources
  - apiGroups: [""]
    resources: ["serviceaccounts", "configmaps"]
    verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]

  # Apps Resources
  - apiGroups: ["apps"]
    resources: ["deployments", "replicasets"]
    verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]

  # RBAC Resources
  - apiGroups: ["rbac.authorization.k8s.io"]
    resources: ["roles", "rolebindings"]
    verbs: ["get", "list"]

  # Specific service account
  - apiGroups: [""]
    resources: ["serviceaccounts"]
    verbs: ["impersonate"]
    resourceNames: ["example-service-account"]
```

Appendix B

Reconciliation charts

B.1 Reconciliation of the lab operator

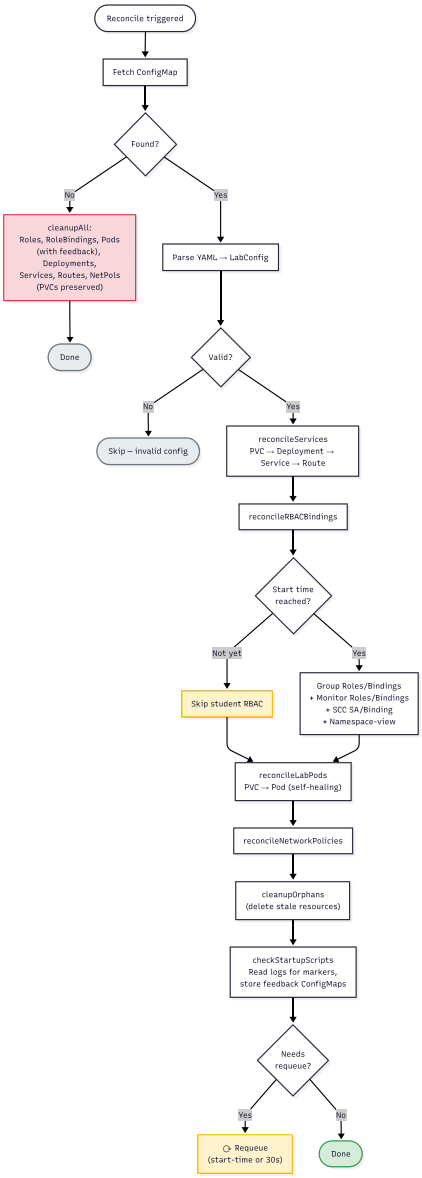

Figure B.1: Lab operator reconciliation flow.

B.2 Reconciliation of the core operator for LabGroup and Lab custom resources

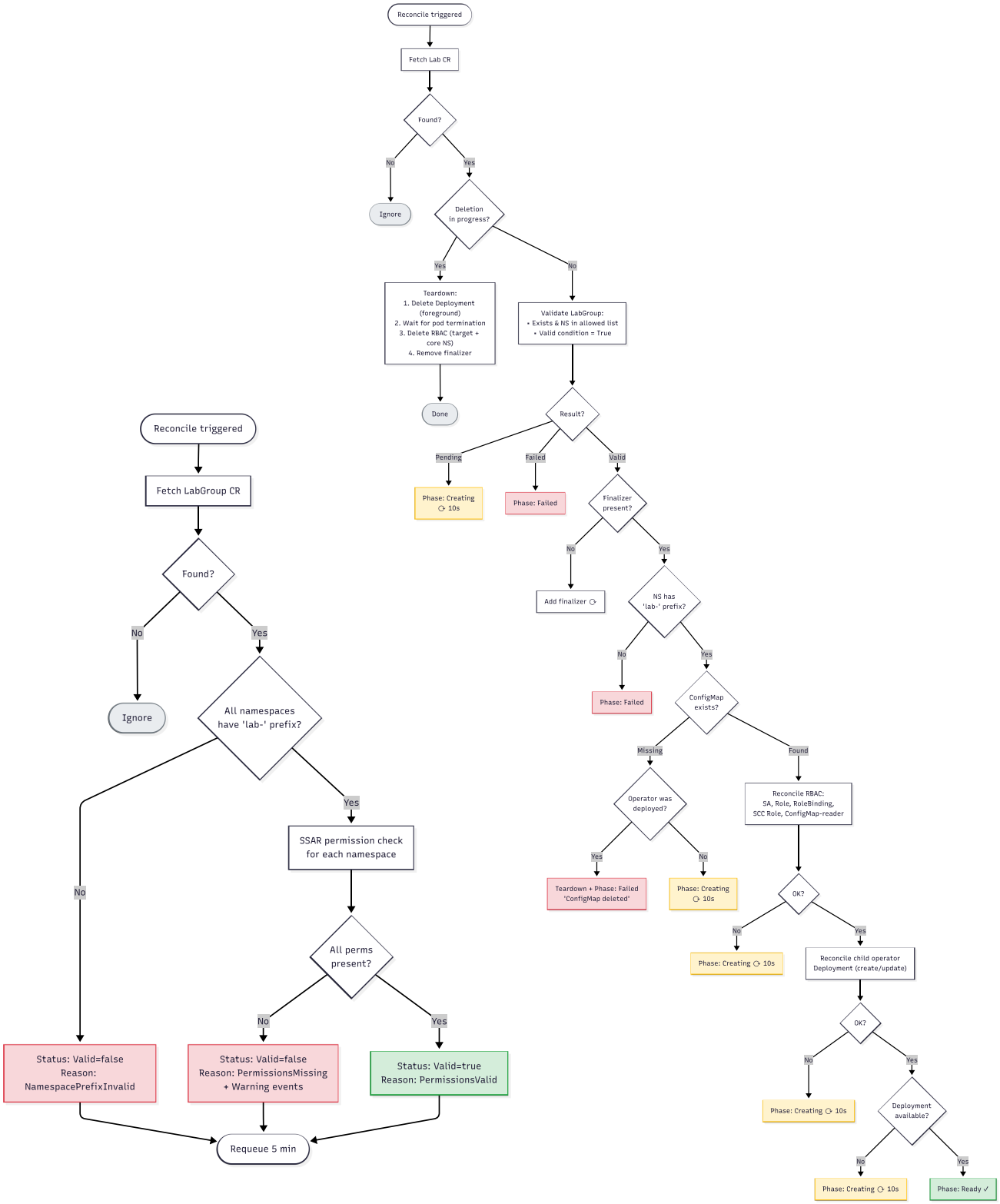

Figure B.2: Core operator reconciliation flow for LabGroup (left) and Lab (right) custom resources.

Appendix C

Namespace delegation Role for the core operator ServiceAccount

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: lab-core-operator-delegation-role
rules:
  # 1. Core Resources (Pods, Secrets, PVCs, Services, etc.)
  - apiGroups: [""]
    resources:
      - pods
      - services
      - secrets
      - persistentvolumeclaims
      - configmaps
      - serviceaccounts
    verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]

  # 2. Apps Resources (Deployments, StatefulSets, ReplicaSets)
  - apiGroups: ["apps"]
    resources:
      - deployments
      - statefulsets
      - replicasets
    verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]

  # 3. Networking (Ingress, NetPol)
  - apiGroups: ["networking.k8s.io"]
    resources:
      - ingresses
      - networkpolicies
    verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]

  # 4. OpenShift Routes
  - apiGroups: ["route.openshift.io"]
    resources:
      - routes
    verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
```

```
# 5. RBAC (Allowed to manage Roles/Bindings)
- apiGroups: ["rbac.authorization.k8s.io"]
  resources:
    - roles
    - rolebindings
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]

# 6. pods log and exec
- apiGroups: [""]
  resources:
    - pods/exec
  verbs: ["create", "get"]

- apiGroups: [""]
  resources:
    - pods/log
  verbs: ["get", "watch"]

# 7. Project access (for OpenShift)
- apiGroups: ["project.openshift.io"]
  resources:
    - projects
  verbs: ["get"]

- apiGroups: [""]
  resources:
    - namespaces
  verbs: ["get"]
```

Appendix D

Custom resource definitions in Go

D.1 LabGroup custom resource definition

```
/*
Copyright 2025.

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
*/

package v1alpha1

import (
        metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
)

// NOTE: json tags are required.  Any new fields you add must have json tags for the
 ↪  fields to be serialized.

// LabGroupSpec defines the desired state of LabGroup.
type LabGroupSpec struct {
        // INSERT ADDITIONAL SPEC FIELDS - desired state of cluster
        // Important: Run "make" to regenerate code after modifying this file

        // GroupName is the name of the lab group, for example a professor's name.
        GroupName string `json:"groupName,omitempty"`

        // Namespaces is the list of namespaces that belong to this lab group. All
         ↪  namespaces should be accessible by the operator.
        // +kubebuilder:validation:Required
```

```
        Namespaces []string `json:"namespaces"`
}

// LabGroupStatus defines the observed state of LabGroup.
type LabGroupStatus struct {
        // Valid indicates whether all namespaces in the LabGroup have proper
        ↪  permissions
        // +optional
        Valid bool `json:"valid,omitempty"`

        // InvalidNamespaces is the list of namespaces where permission validation
        ↪  failed
        // +optional
        InvalidNamespaces []string `json:"invalidNamespaces,omitempty"`

        // LastValidated is the timestamp of the last validation attempt
        // +optional
        LastValidated *metav1.Time `json:"lastValidated,omitempty"`

        // ValidationMessage provides details about validation failures
        // +optional
        ValidationMessage string `json:"validationMessage,omitempty"`

        // Conditions store the status conditions
        // +optional
        Conditions []metav1.Condition `json:"conditions,omitempty"`
}

// +kubebuilder:object:root=true
// +kubebuilder:subresource:status
// +kubebuilder:resource:scope=Namespaced

// LabGroup is the Schema for the labgroups API.
type LabGroup struct {
        metav1.TypeMeta   `json:",inline"`
        metav1.ObjectMeta `json:"metadata,omitempty"`

        Spec   LabGroupSpec   `json:"spec,omitempty"`
        Status LabGroupStatus `json:"status,omitempty"`
}

// +kubebuilder:object:root=true

// LabGroupList contains a list of LabGroup.
type LabGroupList struct {
        metav1.TypeMeta `json:",inline"`
        metav1.ListMeta `json:"metadata,omitempty"`
        Items           []LabGroup `json:"items"`
}

func init() {
```

```
        SchemeBuilder.Register(&LabGroup{}, &LabGroupList{})
}
```

D.2 Lab custom resource definition

```
/*
Copyright 2025.

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
*/

package v1alpha1

import (
        corev1 "k8s.io/api/core/v1"
        metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
)

// LabSpec defines the desired state of Lab
type LabSpec struct {
        // LabGroupRef references a LabGroup that defines which namespaces
        // this lab can be deployed to and validates permissions
        // +kubebuilder:validation:Required
        LabGroupRef corev1.LocalObjectReference `json:"labGroupRef"`

        // NamespaceName is the name of the namespace to use for the lab.
        // Must have "lab-" prefix (enforced by ValidatingAdmissionPolicy).
        // +kubebuilder:validation:Required
        // +kubebuilder:validation:Pattern=`^lab-[a-z0-9]([-a-z0-9]*[a-z0-9])?$`
        NamespaceName string `json:"namespaceName"`

        // ConfigMapRef references the ConfigMap that will be accessible
        // by the child lab operator in the target namespace.
        // +kubebuilder:validation:Required
        ConfigMapRef ConfigMapReference `json:"configMapRef"`
}

// ConfigMapReference defines a reference to a ConfigMap
type ConfigMapReference struct {
        // Name of the ConfigMap
        // +kubebuilder:validation:Required
```

```
        Name string `json:"name"`
}

// LabStatus defines the observed state of Lab
type LabStatus struct {
        // Phase represents the current phase of the Lab
        // +optional
        Phase LabPhase `json:"phase,omitempty"`

        // NamespaceCreated indicates whether the namespace has been created
        // +optional
        NamespaceCreated bool `json:"namespaceCreated,omitempty"`

        // OperatorDeployed indicates whether the child operator is deployed
        // +optional
        OperatorDeployed bool `json:"operatorDeployed,omitempty"`

        // OperatorReady indicates whether the child operator is ready
        // +optional
        OperatorReady bool `json:"operatorReady,omitempty"`

        // Conditions store the status conditions
        // +optional
        Conditions []metav1.Condition `json:"conditions,omitempty"`

        // Message provides additional information about the current state
        // +optional
        Message string `json:"message,omitempty"`
}

// LabPhase represents the phase of the Lab lifecycle
// +kubebuilder:validation:Enum=Pending;Creating;Ready;Terminating;Failed
type LabPhase string

const (
        LabPhasePending     LabPhase = "Pending"
        LabPhaseCreating    LabPhase = "Creating"
        LabPhaseReady       LabPhase = "Ready"
        LabPhaseTerminating LabPhase = "Terminating"
        LabPhaseFailed      LabPhase = "Failed"
)

// +kubebuilder:object:root=true
// +kubebuilder:subresource:status
// +kubebuilder:resource:scope=Namespaced
//
 ↪  +kubebuilder:printcolumn:name="Namespace",type=string,JSONPath=`.spec.namespaceName`
// +kubebuilder:printcolumn:name="Phase",type=string,JSONPath=`.status.phase`
// +kubebuilder:printcolumn:name="Operator
 ↪  Ready",type=boolean,JSONPath=`.status.operatorReady`
```

```
//
↪  +kubebuilder:printcolumn:name="Age",type=date,JSONPath=`.metadata.creationTimestamp`

// Lab is the Schema for the labs API
type Lab struct {
        metav1.TypeMeta   `json:",inline"`
        metav1.ObjectMeta `json:"metadata,omitempty"`

        Spec   LabSpec   `json:"spec,omitempty"`
        Status LabStatus `json:"status,omitempty"`
}

// +kubebuilder:object:root=true

// LabList contains a list of Lab
type LabList struct {
        metav1.TypeMeta `json:",inline"`
        metav1.ListMeta `json:"metadata,omitempty"`
        Items           []Lab `json:"items"`
}

func init() {
        SchemeBuilder.Register(&Lab{}, &LabList{})
}
```

Appendix E

Lab configuration options

E.1 Top-level structure

Field	Type	Meaning
`targetNamespace`	string	**(Unused)** Ignored; namespace is taken from the Lab CR, still supported for legacy.
`metadata`	object	Identifies the course/lab and global time window.
`defaults`	object	Default values applied when templates/services do not override them.
`labTemplates`	list	Definitions for student environments (pods).
`services`	list	Shared services accessible by one or more templates.

E.2 `metadata`

Field	Type	Meaning
`courseName`	string	**(Unused)** Human-readable course name.
`labName`	string	**(Unused)** Human-readable lab/exercise name.
`timezone`	string	Timezone for start/end times (default: `Europe/Brussels`).
`globalStartTime`	string (ISO 8601)	If specified, student RBAC permissions are delayed until this time.
`globalEndTime`	string (ISO 8601)	Time after which student access is revoked and active exec sessions are terminated.

E.3 `defaults`

Defaults are applied when a template/service does not provide its own value.

Field	Type	Meaning
kind	string	Must be pod (current supported kind).
resources.cpu	string	CPU request/limit value (e.g. 500m, 1). Defaults to 10m/500m.
resources.memory	string	Memory request/limit value (e.g. 512Mi, 1Gi). Defaults to 64Mi/512Mi.
persistence.enabled	boolean	Whether persistence is enabled by default.
persistence.size	string	Default PVC size (e.g. 10Gi).
lifecycle.startDelay	string	Delay before starting (duration-like string).
lifecycle.endDelay	string	Delay before terminating (duration-like string).
allowNamespaceView	boolean	If true, students can see the namespace in the OpenShift UI (but can list pods in that namespace).
allowLogAccess	boolean	If true, students are allowed to read the logs of their pods.
scriptTimeout	int	Timeout in seconds for start/end scripts executed in pods (default: 30).

E.4 labTemplates[]

Each entry describes one "type" of student environment. Groups/users listed in the template get a pod created from its podSpec.

You normally provide **either** groups **or** idList to define who gets a pod.

E.4.1 General fields

Field	Type	Meaning
name	string (required)	Unique template name.
groups	map<string, list<string>>	**(Required*)** Map of group name to list of student IDs. One pod per group name.
idList	list<string>	**(Required*)** Shortcut for "one pod per ID" (equivalent to one-student groups).
monitorList	list<string>	User IDs (e.g., instructors) who can access all pods of this template.
kind	string	Overrides defaults.kind (currently only pod is supported).
allowLogAccess	boolean	Overrides defaults.allowLogAccess for this template.

E.4.2 networkPolicy

Controls how pods can communicate.

Field	Type	Meaning
internetAccess	boolean	Allow outbound internet access.
allowInterLabComms	boolean	Allow pods within this same template to communicate with each other.
allowCrossTemplateComms	list<string>	Names of other templates these pods may communicate with.
allowServiceComms	boolean or list	True for all services, false for none, or a list of service names.
ingressRules / egressRules	list	Custom firewall rules (advanced).
protocol	string	tcp, udp, or any (used in custom rules).
port	number or any	Port (used in custom rules).
source / destination	string	CIDR or selector (used in custom rules).

E.4.3 podSpec

Defines what actually runs in the pod.

Field	Type	Meaning
useSCC	string (optional)	Name of a custom SCC to use for the pods (e.g., lab-privilege-escalation-scc).
runAsUser	int	Force a specific UID (only possible when using a custom SCC that allows it).
fsGroup	int	Force a specific GID for volume permissions (only possible when using a custom SCC that allows it).
imagePullSecrets	list<object>	Secrets to pull images from private registries (requires name).
containers	list	Container definitions (see below).
persistenceVolumes	list	Named volumes that can be mounted by containers (see below).

podSpec.containers[]

Field	Type	Meaning
name	string (required)	Container name.
image	string (required)	Container image reference.
command	list<string>	Container entrypoint command array.
args	list<string>	Container entrypoint arguments array.
keepAlive	boolean	If true, forces container to run sleep infinity to stay alive.
resources	object	Resource settings like cpu/memory.
defaultShell	string	Default shell path (e.g. /bin/bash).

Field	Type	Meaning
ports[]	list<object>	Container ports (requires containerPort, protocol is optional).
env[]	list<object>	Environment variables (requires name, value).
volumeMounts[]	list<object>	Mount named persistence volumes (requires name, mountPath).
startScript	string or list<string>	Commands to run at startup (remotely executed via exec. Heavier scripts should be included in the image instead).
endScript	string or list<string>	Commands to run at shutdown (remotely executed via exec).
lifecycle	object	Lifecycle hooks (startDelay, endDelay).

podSpec.persistenceVolumes[]

Field	Type	Meaning
name	string (required)	Volume identifier used by volumeMounts[].name.
size	string	PVC size (e.g. 5Gi).
keepAfterDeletion	boolean	If true, PVC is kept after the lab is deleted (default false).

E.5 services[]

Shared infrastructure accessible by multiple students/templates (e.g., shared DB).

Field	Type	Meaning
name	string (required)	Service identifier.
image	string (required)	Container image.
replicas	int	Number of instances (default 1).
internalDns	string	Optional internal DNS alias (hostname sets ExternalName, IP sets ClusterIP).
expose	object	Network exposure configuration (see below).
env[]	list<object>	Environment variables for the service container (requires name, value).
persistence	object	Persistence configuration (see below).
resources	object	Resource settings like cpu/memory (shown in the example config).
startScript	string or list<string>	Commands to run at startup (remotely executed via exec).
endScript	string or list<string>	Commands to run at shutdown (remotely executed via exec).
lifecycle	object	Lifecycle hooks (startDelay, endDelay).

E.5.1 services[].expose

Field	Type	Meaning
type	string (required)	internal, external, route, or monitor (reserved).
port	int (required)	Service port.
path	string	Optional URL path prefix (only for type: route).

Expose type meaning (simple):

- internal: only reachable inside the cluster (ClusterIP)
- external: public LoadBalancer (requires cloud support)
- route: OpenShift Route (public HTTPS endpoint)

E.5.2 services[].persistence

Field	Type	Meaning
enabled	boolean	Enable persistence for the service.
mountPath	string	Required if enabled; where the storage is mounted.
size	string	PVC size.
keepAfterDeletion	boolean	If true, PVC is kept after the lab is deleted (default false).

E.6 Environment variables available to startScript / endScript

- LAB_GROUP_NAME
- LAB_GROUP_MEMBERS
- LAB_STUDENT_ID (same as LAB_GROUP_NAME)
- LAB_STUDENT_SEED (deterministic per group+config)

Appendix F

Custom SecurityContextConstraint

```
allowHostPorts: false
priority: null
requiredDropCapabilities:
  - ALL
allowPrivilegedContainer: false
runAsUser:
  type: MustRunAsNonRoot
users: []
allowHostDirVolumePlugin: false
seccompProfiles:
  - runtime/default
allowHostIPC: false
seLinuxContext:
  type: MustRunAs
readOnlyRootFilesystem: false
metadata:
  annotations:
    kubernetes.io/description: Custom SCC for lab environments that need setuid
    ↪  support. Based on nonroot-v2 but allows privilege escalation. Use only for
    ↪  labs requiring setuid.
  creationTimestamp: '2026-03-27T11:01:09Z'
  generation: 1
  managedFields:
    - apiVersion: security.openshift.io/v1
      fieldsType: FieldsV1
      fieldsV1:
        'f:seLinuxContext':
          .: {}
          'f:type': {}
        'f:readOnlyRootFilesystem': {}
        'f:metadata':
          'f:annotations':
            .: {}
            'f:kubernetes.io/description': {}
        'f:volumes': {}
        'f:groups': {}
        'f:defaultAddCapabilities': {}
        'f:allowedCapabilities': {}
```

```
        'f:supplementalGroups':
          .: {}
          'f:type': {}
        'f:allowHostPID': {}
        'f:allowHostNetwork': {}
        'f:allowPrivilegeEscalation': {}
        'f:users': {}
        'f:runAsUser':
          .: {}
          'f:type': {}
        'f:allowHostPorts': {}
        'f:seccompProfiles': {}
        'f:priority': {}
        'f:requiredDropCapabilities': {}
        'f:allowPrivilegedContainer': {}
        'f:allowHostDirVolumePlugin': {}
        'f:fsGroup':
          .: {}
          'f:type': {}
        'f:allowHostIPC': {}
      manager: Mozilla
      operation: Update
      time: '2026-03-27T11:01:09Z'
  name: lab-privilege-escalation-scc
  resourceVersion: '937671064'
  uid: 1778907f-a40b-4c72-82fb-1c2d6dad3eae
fsGroup:
  type: RunAsAny
groups: []
kind: SecurityContextConstraints
defaultAddCapabilities: null
supplementalGroups:
  type: RunAsAny
volumes:
  - configMap
  - csi
  - downwardAPI
  - emptyDir
  - ephemeral
  - persistentVolumeClaim
  - projected
  - secret
allowHostPID: false
allowHostNetwork: false
allowPrivilegeEscalation: true
apiVersion: security.openshift.io/v1
allowedCapabilities:
  - NET_BIND_SERVICE
```

Appendix G

Practical example configurations

G.1 Container setup files

G.1.1 Container file

```
FROM node:18-alpine

# Install Ganache
RUN npm install -g ganache

# --- OPENSHIFT RESTRICTED-V2 COMPLIANCE ---
RUN mkdir /lab
WORKDIR /lab

COPY populate.js start.sh ./

RUN chgrp -R 0 /lab && \
    chmod -R g=u /lab && \
    chmod +x start.sh

ENV HOME=/lab

# Specify a non-root user as a baseline for restricted-v2 SCC
USER 1001

EXPOSE 8545

# Start the blockchain population script
CMD ["./start.sh"]
```

G.1.2 `start.sh` script

```
#!/bin/sh

echo "Starting Ganache..."
# Start ganache in the background so we can run our population script alongside it
ganache --host 0.0.0.0 &
GANACHE_PID=$!
```

```
echo "Running population script..."
node /lab/populate.js

echo "Ready for connections!"
# Wait for the background ganache process so the container doesn't exit
wait $GANACHE_PID
```

G.1.3 `populate.js` script

```
async function populate() {
    console.log("Waiting for Ganache to start...");
    let ready = false;
    while (!ready) {
        try {
            const res = await fetch("http://127.0.0.1:8545", {
                method: "POST",
                headers: { "Content-Type": "application/json" },
                body: JSON.stringify({ jsonrpc: "2.0", method: "eth_accounts",
                ↪  params: [], id: 1 })
            });
            if (res.ok) ready = true;
        } catch (e) {
            await new Promise(r => setTimeout(r, 500));
        }
    }

    console.log("Ganache is ready. Fetching accounts...");
    const res = await fetch("http://127.0.0.1:8545", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ jsonrpc: "2.0", method: "eth_accounts", params: [],
        ↪  id: 1 })
    });
    const { result: accounts } = await res.json();
    const sender = accounts[0];
    const receiver = accounts[1];

    console.log(`Populating chain with 5000 transactions to generate history...`);

    // We send transactions in batches of 100 to speed up population
    const batch = Array.from({ length: 100 }, (_, i) => ({
        jsonrpc: "2.0",
        method: "eth_sendTransaction",
        params: [{ from: sender, to: receiver, value: "0x1" }],
        id: i
    }));

    for (let i = 0; i < 50; i++) {
        await fetch("http://127.0.0.1:8545", {
            method: "POST",
            headers: { "Content-Type": "application/json" },
```

```
            body: JSON.stringify(batch)
        });
        console.log(`Mined ${ (i + 1) * 100 } blocks...`);
    }
    console.log("Blockchain populated with 5000 blocks successfully!");
}

populate();
```

G.2 Laboratory Configuration

G.2.1 ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: blockchain-lab-config
  namespace: labs
data:
  config.yaml: |
    metadata:
      courseName: "INFO8013-1 - Advanced Computer Security"
      labName: "Smart Contract Lab"
      # Default TZ: Europe/Brussels
      globalStartTime: "2026-06-06T10:08:00"

    defaults:
      kind: "pod"
      # Give a UI path from console root to lab
      allowNamespaceView: true

    labTemplates:
      - name: "blockchain-node"
        # Network policy to allow testing. Enables a pod (like the bench pod)
        # to access all other pods in this lab template.
        networkPolicy:
          allowInterLabComms: true
        # 50 students total
        idList: ["f094932", "f094933", "f094934", "s000004", "s000005", "s000006",
        ↪  "s000007", "s000008", "s000009", "s000010", "s000011", "s000012",
        ↪  "s000013", "s000014", "s000015", "s000016", "s000017", "s000018",
        ↪  "s000019", "s000020", "s000021", "s000022", "s000023", "s000024",
        ↪  "s000025", "s000026", "s000027", "s000028", "s000029", "s000030",
        ↪  "s000031", "s000032", "s000033", "s000034", "s000035", "s000036",
        ↪  "s000037", "s000038", "s000039", "s000040", "s000041", "s000042",
        ↪  "s000043", "s000044", "s000045", "s000046", "s000047", "s000048",
        ↪  "s000049", "s000050"]

        podSpec:

          imagePullSecrets:
```

```
      - name: gitlab-registry-secret

    containers:
      - name: "ganache-blockchain"
        image:
         ↪ "gitlab.uliege.be:5050/felicien.raze/operator/test-blockchain:0.4"
        ports:
          - containerPort: 8545
```

G.2.2 Lab custom resource

```
apiVersion: lab.labs.montefiore.uliege.be/v1alpha1
kind: Lab
metadata:
  name: lab-blockchain-security
  namespace: labs
spec:
  namespaceName: lab-test
  labGroupRef:
    name: lab-group-test
  configMapRef:
    name: blockchain-lab-config
```

References

[1] The Kubernetes Authors. Kubernetes documentation. https://kubernetes.io/docs/, 2026. Accessed: 2026-06-02.

[2] Red Hat. Openshift container platform documentation. https://docs.openshift.com/, 2026. Accessed: 2026-06-02.

[3] The Kubernetes Authors. Kubernetes: Production-grade container orchestration. https://kubernetes.io/, 2014. Accessed: 2026-06-02.

[4] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–18. ACM, 2015.

[5] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[6] Red Hat. Red hat openshift: The kubernetes platform for big ideas. https://www.redhat.com/en/technologies/cloud-computing/openshift, 2011. Accessed: 2026-06-02.

[7] Red Hat. Openshift enterprise 3: Evolving PaaS for the future. https://www.redhat.com/en/blog/openshift-enterprise-3-evolving-paas-future, 2015. Accessed: 2026-06-02.

[8] Marco Iorio, Alex Palesandro, and Fulvio Risso. CrownLabs—A Collaborative Environment to Deliver Remote Computing Laboratories. *IEEE Access*, 8:136458–136471, 2020.

[9] The KubeVirt Authors. Kubevirt: Virtual machine management on kubernetes. https://kubevirt.io/, 2026. Accessed: 2026-06-03.

[10] ContainerSSH Contributors. ContainerSSH: SSH that launches containers. https://containerssh.io/, 2026. Accessed: 2026-06-01.

[11] CTFd LLC. CTFd: The Capture The Flag Framework. https://ctfd.io/, 2015. Accessed: 2026-06-03.

[12] Red Hat. Openshift virtualization documentation. https://docs.openshift.com/container-platform/latest/virt/about-virt.html, 2026. Accessed: 2026-06-03.

[13] Kubernetes Authors. Validating admission policy. https://kubernetes.io/docs/reference/access-authn-authz/validating-admission-policy/, 2026. Accessed: 2026-06-03.

[14] Red Hat. Understanding the operator sdk | openshift container platform. https://docs.openshift.com/container-platform/latest/operators/operator_sdk/osdk-about.html, 2026. Accessed: 2026-06-03.

[15] Red Hat. Operator lifecycle manager concepts and resources | openshift container platform. https://docs.openshift.com/container-platform/latest/operators/understanding/olm/olm-understanding-olm.html, 2026. Accessed: 2026-06-03.

[16] CRC Project. Crc documentation. https://crc.dev/docs/, 2026. Accessed: 2026-06-03.

[17] GitLab. Gitlab documentation. https://docs.gitlab.com/, 2026. Accessed: 2026-06-04.

[18] Google. Google gemini. https://gemini.google.com/, 2026. Accessed: 2026-06-04.

[19] GitHub. Github copilot documentation. https://docs.github.com/en/copilot, 2026. Accessed: 2026-06-04.

[20] Perplexity AI. Perplexity: Ai search engine. https://www.perplexity.ai/, 2026. Accessed: 2026-06-04.

[21] Kubernetes SIGs. controller-gen reference - the kubebuilder book. https://book.kubebuilder.io/reference/controller-gen.html, 2026. Accessed: 2026-06-04.

[22] Kubernetes Authors. Finalizers | kubernetes. https://kubernetes.io/docs/concepts/overview/working-with-objects/finalizers/, 2026. Accessed: 2026-06-04.

[23] Truffle Suite. Ganache: A tool for creating a local blockchain for fast ethereum development. https://github.com/trufflesuite/ganache, 2024. GitHub repository.