

Cloud Computing and Cloud-based Applications

Lab Work Description

September 5, 2023

Introduction

Throughout the course, you will work on transforming an old school ‘runs on my machine’ target application to one that is running fully scalably in the cloud — being in a redundant Kubernetes cluster on OpenStack, all fully automatically deployed. You can do so either using the provided ‘sample application’, or you can build your own application (BYOA) from the ground up if you prefer to do so. You can move from BYOA to the sample application at any point during the lab, but not the other way around.

Description

The course lab will run in 4 iterations covering 4 different topics, each worth between 20% and 30% of your final assignment grade. Each individual iteration will receive a grade on a 10-point scale, but by implementing advanced features you can obtain up to one extra bonus point. The final (average) grade across all iterations, however, will be capped to 10.

For each iteration, there is a list of criteria your application should meet, and a list of advanced criteria you are encouraged to pursue, but not required to. Meeting all ‘basic’ criteria will award at most 7 points for that iteration. By implementing some advanced criteria, you can further increase your iteration grade to at most 11 points (i.e. 1 point ‘bonus’). The points per advanced criterion are listed below. You are always welcome to work on your own ideas — extra work might award you additional points as well. Finally, some iterations list ‘work ahead’ items — these won’t give you bonus points and are suggestions for relevant items to work on.

Each iteration comes with a set of deliverables, explained in more detail below. All deliverables have to be submitted as a merge request from a separate branch named after the iteration number to the `main` branch, in the group repository that will be created for this purpose. In your README, include for each criterion in the iteration whether you met it or not. After submitting the MR, please do not continue to work on the same branch, but create a new branch for the next iteration instead. This way, the MR does not update after the deadline. The deadline for each deliverable is the end of the last lab session for each iteration, as defined also on BrightSpace. Failure to submit one or more deliverables will result in grade penalties.

Each iteration will be assessed during the last lab session for each topic, except if the group requests this to be done at an earlier session already. The TA will test your application and you will demonstrate its functionality for all criteria. The TA will also ask some questions to gauge your knowledge and understanding of the lab topics. An individual grade for each team member will be determined based on their answers and their contribution to the lab. Unjustified absence for a lab will yield a grade of 0.0 for that iteration and send you to the (already scheduled) resit.

In case of issues related to the collaboration between group members the students are advised to first identify the issue with their TA, who will act as a mediator towards resolving the issue. If no agreement can be reached between the group members through this process, or an existing agreement is found to be invalidated in practice, then the student can ask for a reassignment to another group. The latest instance where this request can take place is before the last lab session.

BYOA Requirements

You are more than welcome to work on your own, custom codebase, rather than the provided one. However, there are some criteria such an application needs to fulfil in terms of complexity, to provide equal opportunities for all students. A successful BYOA implementation, meeting all criteria below, will yield 2.0 bonus points on your final grade. This bonus is not capped and is in addition to any other bonuses you have accumulated in each iteration. BYOA always needs to be approved by the TA before you can start working — so please contact them as soon as possible!

For a BYOA to be acceptable, the following requirements need to be fulfilled:

- The codebase can be built from the ground up, or can be an existing one — as long as the existing codebase has little or none of the work for the project done already. A `Dockerfile` or two is fine, but a full Kubernetes implementation is too much.
- The application should have at least three ‘tiers’ or components, such as frontend, backend, and worker in the provided codebase.
- The application should have a minimal UI to access the functionality of the application. A simple Postman collection or OpenAPI specification demo page suffices but does not count as a tier. A proper deployed frontend app would also suffice as one of the tiers.
- The application should use at least one form of persistent storage, such as relational databases, non-relational databases, or caches and one or more message queues for communication between tiers.
- All communication within the app needs to be done asynchronously.
- You are completely free to pick your own technology stack; PHP or Python-based projects are however strongly discouraged.
- If frameworks or other tooling you use provide functionality that conflicts or overlaps with the criteria described below then you are not allowed to use such functionality. For example, you should not use the Docker or Kubernetes integration of Java Quarkus.
- You do not have to finish the entire application in the first iteration — you are allowed to continue building and developing your app throughout the course. However, ...
- ...BYOA is never an excuse for not meeting deadlines or not being able to fulfil criteria of the assignment. So ‘My framework does not support using `.env` files’ does not exempt you from the requirement of using those files, and ‘We have not yet managed to let the backend talk to the database’ does not exempt you from deploying all these components anyway.

Iteration 1: Dockerization (20%)

For the first week, you will work on fully Dockerizing the target application. As indicated before, you will either work on the provided codebase, or you will already be making progress on writing your own app from scratch. In the latter case, no actual functionality is expected to be delivered at this point in time, beyond what is necessary to fulfil the criteria defined below.

The goal of this iteration is to have the entire application run in Docker. Someone starting from a clean Linux distribution with only Git and Docker (Compose) installed should be able to run your entire app, including all its dependencies, *using only the single command* `docker compose up`.

Additionally, the app should be split up into appropriate services, with each their own Docker image: frontend, backend, and the worker (or equivalent concepts and components in your own application). All databases should also run in Docker, this is already provided for the standard codebase and should continue to work.

1.1 Criteria

1. In a fresh repository clone, on a machine with only Git and Docker (Compose) installed, after setting up the `.env` file (see below), running the command `docker compose up -d` in the root directory of your repository should result in your entire app running including any dependencies - this includes automatically building the Docker images. Subsequently running `docker compose down` should terminate all your containers, but should not remove any of the database data.
2. Each component of your application (frontend, backend, worker - or equivalent components for custom applications) should have its own `Dockerfile` that is used to build an image dedicated to that part of the application.
 - You should be using a `.dockerignore` file to exclude unnecessary files from being included in the image.
 - You should be using a multi-stage docker build where relevant to reduce image size.
 - You should optimise the order of operations in the `Dockerfile` to reduce build time for simple code changes.
 - Any compilation or building work should happen during the runtime of the `Dockerfile`. After the image has been built, starting the app should be nearly instant - without any compilation or other processing.
 - You should *NOT* use automatically-generated `Dockerfiles`, e.g. through `.NET` tooling.
3. All settings, including at least all server addresses, ports, and passwords, should be configurable through a `.env` file in the root of the repository. This file **must not** be included in the repository itself; instead, a template file should be provided. Note that these values are typically used as *runtime* settings, so they should not be used in `Dockerfiles`.
4. All parts of your app should communicate with each other through an internal Docker network.
5. Any data from databases should be persisted in a volume, these volumes should be mounted and re-used automatically with `docker compose up`.

Advanced

6. (1.5p) Set up a GitLab CI/CD pipeline to automatically build the Docker images and publish them to the GitLab registry.

7. (1p) In addition to the previous, also set up automated dependency and vulnerability scanning for your application and have the scan result be integrated in the GitLab MR view.
8. (1.5p) Make your setup flexible for both development and ‘production’ purposes. Development of the application should not require rebuilding the image for every single code change, as that is slow and tedious. At the same time, the production setup should be optimized: it should not include debugging code or other artifacts.
9. **(Work ahead)** Expose ‘health checks’ in your app that allow Docker (and later Kubernetes) to inspect the health of your app. This health check should include not only basic functioning of the application, but should also include whether the connections to various databases is established.

1.2 Considerations

The proper Dockerization of an application requires some careful thought and planning — especially when you have not done so before. To guide you, here are some points to think about before you start:

- The provided application has some shared libraries (‘Mongo‘, ‘Shared‘, etc) that are used by both the worker and the backend components. Will you just rebuild the code for these libraries for each component, or are you going to re-use the binaries?
- Ideally, the frontend will be compiled to static code such that an nginx server can just serve the static files for your frontend to work. However, how will the frontend then know how to connect to the backend? Are you using a default domain name or address, or are you going to make things more dynamic? How will you do so?
- Pay attention to port mappings in Docker: these contain two port numbers that are not the same conceptually! One is the port that is exposed at the host, while the other is the port inside the container that is being connected to. Only the first should be configurable by the user.

1.3 Deliverables

You should deliver all work for this iteration in a merge request on GitLab, as for every iteration. This merge request should contain all the work you did during this iteration, which includes at least:

1. A `Dockerfile` for every component of your application, in the appropriate directory,
2. A `docker-compose.yml` file for spawning all your containers,
3. A template `.env` file,
4. An updated `README` with information about the application and how to run it.

1.4 Resources

Here are some resources you can use to get started:

- Docker Documentation - Build a .NET image
- Vue.js Documentation - Dockerization
- Docker Documentation - Multi-stage builds
- Docker Compose Introduction
- Docker Image - Redis

- [Docker Image - RabbitMQ](#)
- [Docker Image - MongoDB](#)
- [GitLab Documentation - Building Docker Images](#)
- [GitLab Documentation - Container Registry](#)
- [GitLab Documentation - Container Scanning](#)
- [ASP.NET Documentation - Health Checks](#)

Iteration 2: Kubernetes (25%)

During the second iteration, you will work on making your application available in a Kubernetes cluster. This requires the entire app to be containerised first! The focus of this iteration will be on understanding the Kubernetes lifecycle and operating a cluster — so it is not yet required to have your application replicated or automatically recovering, but that will be required for the final iteration.

Before you can run your app in Kubernetes, you first need to set up the cluster itself. For local development, it is probably most convenient to run a small instance locally through Minikube or some alternative. Another option is to spin up a VM in an OpenStack deployment to install a Kubernetes cluster there, manually. Options include RKE, Kubeadm, K3S, or also Minikube. It might be good to gain some experience with this route, as you will be automating this in the next iteration.

In either case, it is recommended to install some kind of management dashboard in your cluster, to ease development and debugging, while also providing nice insight into the cluster state. Options include the default Kubernetes dashboard, Rancher Server, Komodor, or any other tool of your choice.

2.1 Criteria

1. Deploy and run the target app on a Kubernetes cluster.
2. Create appropriate Kubernetes resource definitions for the various parts of your application. These include `ReplicaSets`, `Deployments`, and `Secrets`.
3. Make sure to use `Services` and `Ingresses` to expose your application outside your Kubernetes cluster.
4. Organise your Kubernetes resources into Helm charts - one for every component of your application (e.g. frontend, backend, worker) and use pre-existing Helm charts for the shared infrastructure (e.g. MongoDB, RabbitMQ, and Redis).
5. Using a single command, be able to automatically deploy all Helm charts with appropriate settings to your cluster. For example, you can use an installation script.
6. Have all your application components and dependencies run in Kubernetes.
7. **Never store any plaintext secrets (passwords, API keys, tokens) in your repository!** You must either generate these on-the-fly (and store them somewhere securely), or encrypt these secrets in your repository. A tool like sops can help with this.

Advanced

8. (1p) Instead of using a deployment installation script, use Helm management tools such as Terraform, ‘Helmfile’, or Kustomize to configure and deploy Helm charts to your cluster.
9. (1p) Automate the deployment of your application in Kubernetes using GitLab CI/CD. Use the GitLab Agent to provide connectivity between GitLab and your cluster.
10. (2p) As an alternative to explicitly deploying resources to your cluster, adopt a GitOps model. Whereas the tools mentioned before are push-based, a GitOps workflow is pull-based: it listens for changes in your repository and automatically updates the cluster to conform to the specification in your repository. Relevant tools for such an approach include FluxCD and ArgoCD.
11. (1p) Make your application publicly available. On request, we will provide you with a group-specific subdomain name in Cloudflare. You can use cert-manager to obtain Let’sEncrypt certificates and external-dns to dynamically update the Cloudflare DNS records.

12. (1p) Expose ‘health checks’ in your app that allow Kubernetes to inspect the health of your app. This health check should include not only basic functioning of the application, but should also include whether the connections to various databases is established. Enable automatic restarts when the health check fails.
13. **(Work ahead)** Integrate your Kubernetes cluster with OpenStack, for example by using OpenStack volumes inside Kubernetes through Cinder.
14. **(Work ahead)** Replicate all components of your application (but not databases) at least 3 times and have them automatically fail-over and recover from issues (note: required for the final iteration, but optional for this one).

2.2 Considerations

Implementing a full Kubernetes setup is not trivial, and depending on your application, you might run into some challenges. Below, you can find a short list of points to consider carefully during implementation.

- The provided application supports multitenancy — how do you support this in a Kubernetes context? Do you explicitly list tenants in your ingress, creating more of a ‘whitelist’ functionality? How do you keep those up-to-date? Or do you use wildcards?
- As mentioned in the criteria, you should never include secrets in your repository — except for when these are securely encrypted. However, this might pose a challenge for implementing CI/CD: how will the CI/CD runner access the secrets?
- How do you handle the health checks of the ‘worker’ component? It does not host a webserver by default, so the ‘normal’ route of accessing some kind of `/ping` route won’t work.

2.3 Deliverables

Once again, you should deliver all work for this iteration in a merge request on GitLab. This merge request should contain all the work you did during this iteration, which includes at least:

1. For each component in your application, the relevant Helm chart resources,
2. An installation script to deploy all Helm charts, or relevant definition files for Terraform, helmfile, Kustomize, or any other tool used for deploying Helm charts,
3. An updated `≡ README` with information about your Kubernetes implementation, how to deploy the app, and how to manage secrets.

2.4 Resources

Once again, here are some resources you can use to get started. Note that not all resources will be relevant to you: you might use a different Kubernetes engine for which things are handled differently, or you might decide to not pursue a specific advanced feature. Some of these links are especially useful if you opt to use the sample target app.

- Kubernetes Engine - MiniKube
- Kubernetes Engine - Docker Desktop
- Kubernetes Engine - Kubeadm
- Kubernetes Engine - RKE2
- Kubernetes Engine - K3S
- ASP.NET Documentation - Deploying to Kubernetes

- Helm - Introduction
- Helm - Charts for a .NET app
- Helm Charts - Redis
- Helm Charts - RabbitMQ
- Helm Charts - MongoDB
- Helm Manager - Helmfile
- Helm Manager - Kustomize
- Helm Manager - Helmwave
- GitLab - Connecting to Kubernetes
- GitLab - Using GitOps
- GitOps - FluxCD
- GitOps - ArgoCD
- Cert Manager for nginx (Note: K3S does not use nginx but Traefik)
- Cert Manager - Installation
- external-dns
- OpenStack integration through Cinder
- ASP.NET - Health Checks
- Secret Management - sops
- Secret Management - sops with FluxCD
- Secret Management - Hashicorp Vault
- Integrating Vault with Kubernetes

Iteration 3: Terraform (30%)

In this last ‘regular’ iteration, you will work on deploying Kubernetes clusters, together with your entire app, to OpenStack using Terraform. Essentially, the manual labour in setting up Kubernetes for the previous iteration should now be automated and put into an *Infrastructure as Code* form.

The final order of operations will consist roughly of: setting up networking and images in OpenStack, spawning a number of VMs with the correct settings, then installing the Kubernetes cluster distribution of your choice on these VMs, and then deploying your application on top of this cluster.

At the end of this iteration, you should be able to do all of the above **using 1 command**, which can be just a script that calls various other automation tools.

3.1 Criteria

1. Use Terraform to deploy VMs with an OS of your choice to OpenStack.
 - It is recommended to use an OS that supports automatic configuration and provisioning through OpenStack, such as Flatcar Container Linux (through ignition) or Ubuntu (using cloud-init).
2. Automatically deploy a Kubernetes cluster on those VMs, for example using RKE, K3s, Kubeadm, or ClusterAPI.
3. The deployed Kubernetes cluster should have at least 3 nodes for redundancy and failover.
4. Integrate Kubernetes with OpenStack for volumes, by using Cinder as a storage provider.
5. Automatically deploy your application inside the newly-created cluster, using the tooling you created for the previous iteration.
6. Store your Terraform state in GitLab using their Terraform integration - do **NOT** include the state file in your repository!
7. Properly manage your deployment secrets. Once again, **NEVER store passwords or other secrets in plaintext in a repository!**

Advanced

8. (1p) Allow the deployment and management of multiple co-existing clusters through the use of Terraform workspaces. For example, use separate clusters for development and production environments.
9. (1p) Automate the Terraform deployment and management through GitLab CI/CD.
10. (2p) Automatically register the new Kubernetes clusters in GitLab to further automate deployment of your app.
11. (2p) Use Hashicorp Vault to automatically create and provide temporary, short-lived credentials for database access in your cluster.

3.2 Considerations

The full Terraform deployment might be the most tricky part of the entire project, since many moving parts should now come together and work as one. This makes it even more important to carefully plan ahead, before spending days on a solution that will never work. To help you with this, here are some points for you to consider:

- Terraform is not really built to ‘provision’ a new VM — there are many explanations online about this vision of Terraform. The bottom line is that this makes it difficult to get a VM to autonomously ‘do’ things after its creation, and you can quickly find yourself writing script

upon script upon script to ‘automate’ things (connecting to the VM through SSH, running commands — all very error-prone). Instead, you could consider taking one of the following routes:

- One option is to use Flatcar or another similar OS that is specifically designed for use in situations like this: they support a full (declarative) configuration that is applied upon the first boot, which allows you to automatically start a variety of processes without needing to access the VM from outside. This can then include the automatic deployment of a Kubernetes cluster.
- Another option is to use a separate tool like Ansible or Salt to provision your VMs — they are designed for this purpose. However, we would not recommend this approach for the small scope of this course, since these tools incur quite some overhead.
- Think about how to retrieve credentials for the newly-spawned Kubernetes clusters — you’ll need those to be able to deploy the application on top of it. It’s not allowed to manually SSH into a server — it all needs to be automated.
 - Using a GitOps model can simplify this part: the credentials would never need to leave Kubernetes, as the cluster will pull all the necessary resources on its own. That way, deployment can continue automatically.
 - When using GitLab integrations, installing the GitLab agent is another way to connect your cluster to outside tools to allow automated deployment of your application (through CI/CD) without having to manage credentials.
- Think about how to do secret management. The more you automate, the harder this will get. Try to ignore the urge to just ‘temporarily’ hardcode a credential ‘just to see whether it works’ — this will quickly become a mess to then try and fix later and you can quickly end up with credentials in plaintext in your Git history.

3.3 Deliverables

As always, you should deliver all work for this iteration in a merge request on GitLab. This merge request should contain all the work you did during this iteration, which includes at least:

1. Terraform files used for deploying VMs and clusters,
2. A script to perform the full deployment of everything,
3. An updated `README` with information about your approach, paying special attention to the considerations listed above.

3.4 Resources

- Terraform - OpenStack provider
- Flatcar Container Linux - Running on OpenStack
- Flatcar Container Linux - Terraform Instructions
- Flatcar Container Linux - Terraform Examples
- Cloud-init - Documentation
- Kubernetes Engine - RKE2
- Kubernetes Engine - K3S
- Kubernetes Engine - Kubeadm
- Kubernetes Engine - ClusterAPI

- [OpenStack - Cinder integration in Kubernetes](#)
- [GitLab - Terraform State Management](#)
- [Terraform - Workspaces](#)
- [GitLab - Terraform Helpers](#)
- [GitLab - Terraform How-To](#)
- [RKE2 - Helm Automation Integration](#)
- [K3S - Helm Automation Integration](#)
- [Hashicorp Vault - Redis](#)
- [Hashicorp Vault - RabbitMQ](#)
- [Hashicorp Vault - MongoDB](#)


Iteration 4: Final Product (25%)

In this final week of the course, you will finish your entire application. You will mainly have time to polish the work from previous iterations, and you should pick at least one of the following topics to implement as well. The listed points are the maximum number of points for that item.

- A. (4p) Replicate all application components, including the databases. Have at least 3 instances of each component running, and make sure to automatically fail-over and recover when one instance fails. This should *demonstrably* work by killing a random database instance. (Note: this is trickier than it might sound)
- B. (6p) Refactor the application logic to make use of a serverless framework.
- C. (6p) Fully automate the entire deployment of your application in GitLab CI/CD. This should feature different environments for different branches: production, staging, and development. In addition, it should feature ‘review apps’ on their own separate environments. Integrate with GitLab where possible.
- D. (4p) Rewrite the provided frontend of the sample target application to something that is more usable, provides more debugging information, and runs more quickly.

4.1 Deliverables

Also for this last iteration, you should deliver all work for this iteration in a merge request on GitLab. This merge request should contain all the work you did during this iteration, which should include:

1. The full, updated codebase of the application,
2. All deliverables of all previous iterations,
3. An updated  README with all information about your implementation, work done, special features, and deployment/usage instructions.