University of Southern Denmark

Department of Mathematics and Computer Science

DM885: Microservices and Dev(Sec)Ops

Cloud solution for solving optimization problems

Authors

Abhijeet Bhatta
abbha22@student.sdu.dk

Anders Black Borges Larsen
andla19@student.sdu.dk

Frederik Gram Kortegaard
frkor19@student.sdu.dk

Jakob Kofoed Skovlund
jasko19@student.sdu.dk

Lasse Usbeck Andersen
lasan19@student.sdu.dk

Niels Peter Roest
niroe18@student.sdu.dk

Troels Lind Andersen
trand19@student.sdu.dk

January 2, 2023

Contents

1 Introduction

The purpose of this project is to create a system, using microservices and the DevOps approach, that is able to solve optimization problems by utilizing a select number of solvers in the cloud. With this system a user should be able to upload a optimization problem, select solvers, and get an output from the fastest solver among the chosen.

Main challenge: ? Microservices and communication/splitting into small

2 Preliminaries

3 Infrastructure

3.1 Docker - Containerization

Docker is a technology that allows developers to package their applications as containers which can then be executed and act the exact same way on any host machine. Docker also provides some security, because each service is its own virtualized distro, it means that you cant easily escape and hijack other services, even if you exploit one. This is done through virtualization, each docker container is an entire linux distro (minus the kernel) which is usually packaged with a service or application of some sort. A developer can then write Dockerfile which specifies what the image should contain, said image can inherit a base image, copy the necessary files, install necessary libraries, and specify a command to be ran when the container is ran.

For our project we use docker to package our applications so they can be ran using kubernetes.

3.2 Kubernetes - Container orchestration

Manually managing docker containers is very cumbersome and error-prone. This is why we chose kubernetes for doing container orchestration. Kubernetes has many of the features you would expect of a container orchestration system, such as self-healing, scaling, internal network routing and much more. Generally, it consists of a control plane which manages one or more nodes, often with multiple Pods per node.

Kubernetes has high-level concepts such as Services, Deployments and Volumes which makes it easy to describe the system you want. Kubernetes is used by writing a specification in `yaml` of what you want in the system, and then applying that specification. An example of this can be seen in Listing 1, where the authentication service is configured in a way so it is exposed on the network as a service. This means that the authentication service can be accessed by other Pods on the kubernetes cluster, since it has the type ClusterIP. Practically this means that any requests to `http://auth-service.default.svc.cluster.local:5000` is directed to an authentication service Pod. Another concept we heavily use is Deployments. A Deployment is a set of Pods which the kubernetes control plane ensures is running, such that if a Pod of a certain kind fails, a new equivalent one will be deployed to replace it. They can also be configured to scale automatically.

Pods in kubernetes normally have non-persistent storage, which means that if a Pod goes down, anything it had stored locally would be lost. For some of our services we needed some form of persistent storage. This can be achieved in Kubernetes with Persistent Volumes (PV). PVs can be requested by Persistent Volume Claims (PVC), and represent persistent storage which can be mounted on a Pod.

Another feature of kubernetes we use is Config Maps. This is a simple way to add environment variables to Pods, which we used for services which needed access to a database. Both the database Pod and the Pod containing the service had a Config Map attached which set the needed database variables. How we

```yaml
apiVersion: v1
kind: Service
metadata:
  name: auth-service
spec:
  type: ClusterIP
  selector:
    app: auth-service
  ports:
    - protocol: "TCP"
      port: 5000
      targetPort: 5000
```

Listing 1: Part of the `yaml` file describing the authentication service.

actually write the security sensitive strings, like passwords, to these Config Maps without storing them in plaintext in our version control, will be described in Section 3.5.1.

We used the tool `kubectl` for managing the kubernetes cluster. With `kubectl` you do things like applying `yaml` files and restarting deployments. It was also used heavily when debugging, where it can be useful for getting information from the Pods like their status and logs as well as executing commands on the Pod.

The concepts of kubernetes Jobs and Cluster Roles will be described in Section 4.6 since they are tightly related to the job service.

3.3 Google Cloud - Cloud

The decision of which Cloud to use was a big one to make, and we tried to research thoroughly before choosing. One of the main restrictions for the project was the budget, and we this was one of our major concerns when choosing a cloud platform. Initially we tried out AWS, but the cost was simply too high for the free tier we had and we abandoned that idea. We saw that Google Cloud

3.4 Terraform - Infrastructure as code

Terraform is software that provides infrastructure as code for clouds. With terraform you can specify an entire cloud infrastructure as configuration files and apply them to any given cloud. Because terraform is declarative, configurations define what the final state should look like. It keeps track of what exists in the cloud vs. what the configurations specify, so it can detect discrepancies and delete, change or create new resources, without having to rebuild the entire infrastructure. The configuration files are simply text files, and as such can easily be version controlled. This makes it a lot easier to record changes in the infrastructure and to find where a possible error in the infrastructure was introduced.

Terraform is useful because it allows developers to more rapidly deploy applications and more rapidly change the infrastructure, achieving greater speed and efficiency when developing. It also made it very simple to move the project from one Google Cloud account to another, since the configuration simply could be applied to the new project and terraform automatically would provision the needed resources within 30 minutes.

Ideally you would want the terraform configurations to be applied to the cloud as part of the CI/CD pipeline, so that changes would take effect immediately when the changes were pushed. We chose not to

do this, but instead run it locally.

3.5 Github Actions - CI/CD

Github Actions is a service provided by Github which allows users to define a series of commands to be ran when a specific change occurs.

For our project we detect changes to any given service, and if a commit changes it. Said service is rebuilt and the image is uploaded to a google cloud container registry so they can be fetched by kubernetes. Kubernetes is then told to refresh the given service as to run the new version of the service.

3.5.1 Handling secrets

Naturally each service has some secrets, ranging from a database username and password, to secret keys. This is where github secrets come in, github secrets allows you to define a special variable which can be accessed from any github action like so:

```
env:
  SERVICE_IMAGE: fs_service
  FS_DB_USER: ${{ secrets.FS_DB_USER }}
  FS_DB_PASS: ${{ secrets.FS_DB_PASS }}
```

Listing 2: Part of the `Github Action` file describing the environment variables in the fs service deploy action.

From here said secret can then be passed onto the service, for most services these are written to kubernetes Config Maps. This is done through 'envsubst', which reads a file and replaces any instance of a suitable variable such as: `${A_VARIABLE}`, with the existing environment variable `A_VARIABLE`. This is then used to write each necessary secrets into each service's respective kubernetes Config Map, which is then applied using kubectl so each service now knows its respective secrets. Google Cloud also define containers from a projects registry by both the project id and service name, which necessitated also replacing the projectid for in each kubernetes definition.

3.6 Monitoring & Alerts

as with any production quality project ...NOT DONE... As monitoring is - at least in our opinion - a requirement for any project of this nature, we implemented multiple channels of monitoring so we could establish in depth insights for our product. The first of these channels is Actions, which as previously mentioned in section 3.5, includes not only deployment but also testing and source code analysis in the form of linting. As this topic has already been discussed, we will skip directly to our next monitoring channel; Google Cloud.

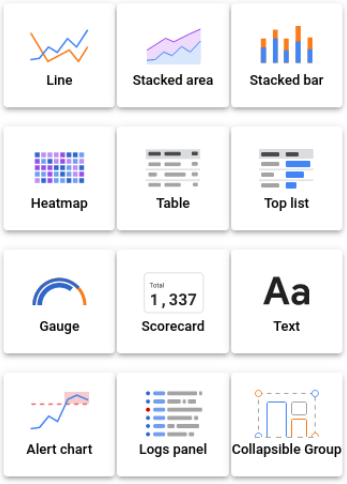

Figure 1: *Chart types in Google Cloud's monitoring dashboards*

Google Cloud features a comprehensive range of integrated monitoring solutions for most all of their products, in addition to marketplace add-ons and extensions. For this project, we utilize the *Google Kubernetes Engine* heavily, and in turn, also has great monitoring integration.

On figure 1 a sample of the available chart types already supported by Google Cloud can be seen. These charts can be easily dragged onto a grid layout in the Google Cloud Monitoring Dashboard, and setup in a matter of seconds. Once a chart type if selected, a user can select a metric from the menu seen on figure **??**, however, it is also possible to use more advanced data gathering by way of tools such as Prometheus or MQL. Referring back to our project, we created a handful of metrics which we felt created sufficiently valuable insight for a prototype.

In addition to logging and visualization, alerts and notifications are an essential part of a good monitoring platform, and luckily, Google Cloud also features integrated notification solutions which all work perfectly together with the same metrics available for logging. First we setup a notification channel - where we will receive our alerts - using email, however, in the future we would add Webhooks, SMS, and most likely Slack channel support, as all of these are ready to use out-of-the-box, and could provide greater availability for our alerts. Having defined notification channels, we created three alerts: Request Latencies, Rate Quota Usage, and CPU Limit Utilization. These metrics were chosen specifically as they represent important identities in monitoring: availability, stability, and scale-ability. Here it would be optimal to implement further alerts fulfilling common Site Reliability Engineering metrics such as Error Rate.

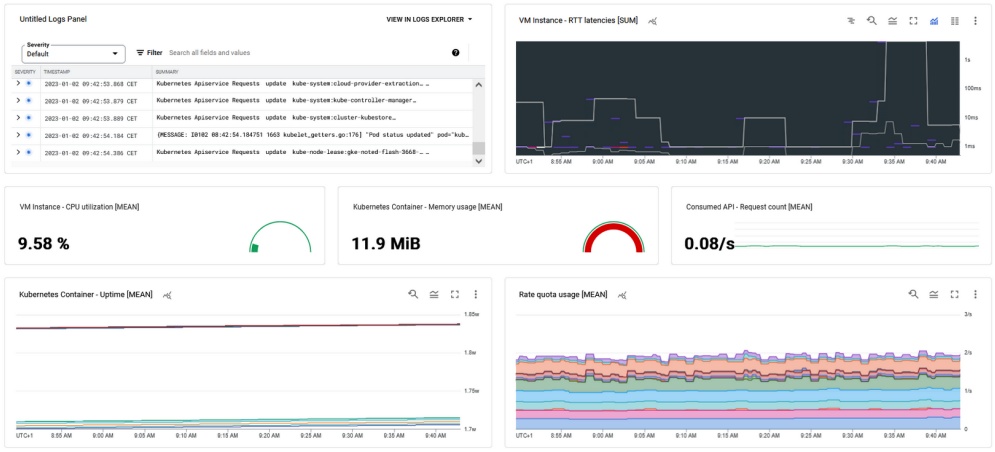

Figure 2: *Monitoring Dashboard for SolveIt*

Sections 3.3 and 3.4 described our ability to easily setup Google Cloud infrastructure using `.tf` files, and monitoring is no exception to this.

3.7 Testing

Testing is done through API request expectations.

4 Architecture

4.1 Technologies

All services with the exception of the frontend uses the FastAPI framework for their REST API.

All services with the exception of the frontend uses SQLAlchemy for managing their data and thei postgresSQL databases. From now on "all services" in this section refer to all except the frontend. SQLAlchemy is a python toolkit and object relational mapper, which allows us to declare our PostgresSQL schemas as python classes.

Insert how it roughly works?

All services contains a model.py which contains the models of the service database as a python class.

All services contains a schemas.py which contains the schemas of the service database as a python class.

Most services contains a database.py which initiates an engine that is used to connect to the given database URL, which is then used to create a session class that creates a new session everytime it is initiated.

Uvicorn?

4.2 Authentication service

4.2.1 JSON Web Tokens

4.2.2 Public-key Cryptography

4.2.3 Passwords & Hashing

4.3 File service

The file service API was developed using the python library FastAPI. This was due to FastAPI being a good API language with builtin model support, and that we collectively agreed to use it for all services. For the file service, I made 6 endpoints, the first of which is the `write` endpoint, which writes a file. Each endpoint follows at least some common rules, you may only upload as the user id your token has (unless you are admin), and you may not exceed your given storage quota (10MB by default). The `write` endpoint depends on a `userid`, a file object, and an authentication token. It then checks that you have enough space to perform this write, and if you do, then it creates a new file in the database, writes the file to storage, and returns the `fileid` given by the database. The database generates id's itself. It should be noted that it uses integers for this, counting from 1 and up, this potentially has security implications that will be addressed in a bit. Filenames are also sanitized to remove most non-alphanumeric characters. This is one of multiple measures to avoid script injection on the frontend, the frontend itself also passes names through a filter to prevent scripts from being executed. Another endpoint is the patch endpoint, which works alot like the write endpoint, with the exception that it creates no new files, it instead finds a file in the database and overwrites the given file with a new file, and updates the file in the database to match the new file. Additionally there is a `delete` endpoint, which deletes a file from the user. Furthermore

there is a `user delete` endpoint, which deletes all files from a user. On top of that, there's also a `file read` endpoint, which reads a file. Finally there's a `list` endpoint, which lists all files uploaded by a user.

4.3.1 Drawbacks

First there is the id generation, it should be using UUID just like the authentication service to prevent sequential access in the case another vulnerability was exploited. If we imagine an exploit where a malicious third party gained the ability to any read file, it would be trivial to find all files in the system, as they have sequential id's. Secondly, there is the way files are read. It reads the entire contents of the file into memory before writing it to a file, meaning a malicious user could spam file writes and overload whatever VM the service is running on. This is addressed in the load balancer, as it can be used to limit the "body" size of any given request (even if its currently set too high). Third, an admin can upload as any user, this is arguably a design flaw, as it doesn't make sense to upload as a non-existent user, but the effect is negligible since only admins can do it. An additional oversight is that there isn't a global storage limit, which may become problematic if you gain too many users.

4.4 Frontend

The frontend is developed without a framework and is instead pure HTML/JavaScript. This decision was taken due to a lack of experience with frameworks and since it was expected that time would be a precious resource. That being said, there can be several benefits to using a framework such as routing and best practices being built into the framework. Deciding not to use a framework has another great benefit: It is very easy for any of the other developers to edit the frontend. Since it is purely HTML and JavaScript, it requires only very baseline programming knowledge to edit and make work. There were multiple instances of other programmers changing the code of the frontend to fit the changes they made to their service.

For visual modules, we used Bootstrap, an easy to plug component library.

All API calls have been implemented using Fetch and half of the frontend consists of making, agreeing on and debugging these API calls. Looking back at the development of the system, this task could have been made vastly easier with some simple planning. Due to the way we divided the responsibility of the services, little to no communication between the developers making the API's. This resulted in numerous misunderstandings and several different implementations of very similar API calls. A small meeting between the developers asked to develop the API´s could have solved this issue.

Some measures to secure the frontend was taken. In order to secure against XSS (Cross Site Scripting) DOMParsing was using for the user inputs when receiving them from the various services. For authentication, JWT were used. With permissions also encoded into the token, the uses these value to validate the actions of the user. This is only secure due to every call also passing the token to the the service it has called and every service verifying the validity of the token with the Auth Service.

4.4.1 Gateway

4.5 Solver service

The motivation for making a solver service was that both the frontend and and job service requires access to the available solvers in a database, so taking out and making it into its own microservice made the most sense. The purpose of the solver service is to manage solvers, this includes, adding new solvers to the database of available solvers, removing a solver from said database, requesting all solvers or requesting a solver by id from said database and contains API endpoints for them all. The solver service were built using FastAPI for a REST API. The solver service mostly works as a gateway between other services and the database for storing solvers. Every solver is stored in a PostgresSQL database, hosted in google cloud.

Each solver in the database is described as an entry which contains a name for the solver as a string, the solvers docker image url as a string and an id as a UUID. The management of the database is done using SQLAlchemy.

On startup the solver service tries to add 3 predefined solvers to the solver database if they do not already exist, additionally it retrieves the public key from auth service to use later in authentication. When retrieving a solver or deleting a solver it check whether the required UUIDs are valid uuids, before attempting database interactions. When posting and image to the database we wanted the solver service to verify the image, checking if the given url contains a valid docker image. Currently it uses docker hubs API to check if the given namespace and repository combination exists on docker hub, this solution have the obvious flaw of not being able to verify images that is from any other site than docker hub. Ideally we wanted to boot up a new pod containing a the docker image where we would then be able to test whether it could pull from the given url and run it.

The solver service uses the auth service to authenticate the JWT passed to it, only allowed authenticated user to use its API, additionally you need admin permissions in order to be able to post or delete a solver.

4.6 Job Service

To facilitate running the different solvers on Minizinc problems, we needed a separate service to do this. This service is the job service. The job service has API endpoints for starting, stopping, managing and getting information from jobs. In this context a *job* is some Minizinc problem which should be solved by one or more solvers. More specifically a job consists of a `mzn` problem file, possibly a `dzn` file, as well as a list of solvers where each solver contains parameters for how many resources to use and a timeout. Each solver working to compute a problem will be denoted as a solver instances.

Since the endpoint for creating a job takes ids for the solvers and the problem file(s), it is also the responsibility of the job service to contact other services to retrieve the relevant information with these ids. It checks that the file id is valid and gets the contents of the `mzn` file from the file storage service, and the same for a `dzn` file if applicable. For the solver id, it needs to contact the solver service to get the link to the corresponding docker image. The decision to structure it like this, with an id and an extra call, and not just accept any image is deliberate. If we had just accepted a string of the link to an image given by the caller, the caller could abuse this to forge a request which contains a malicious image, which would then be run on the cluster.

When starting a job, resources are checked...

To handle starting new pods with arbitrary solver images, we used kubernetes jobs.

4.6.1 Kubernetes Jobs

The concept of Jobs in kubernetes is essentially a Pod with a set time to live and other features for managing itself as a temporary Pod. Its intended use is to compute something, produce a result and destroy itself. This behavior is perfect for solving Minizinc problems. It has really useful features for computing problems in parallel as well such as stopping other Pods when one has completed. Initially we had thought we could use this to parallelize the computation by using a single Job which would contain multiple Pods working on the problem. We then realized that this feature requires that all of the Pods had the same image [1], which was not possible for this project. We instead opted for starting a kubernetes Job for each solver for a problem. This allows to still limit the timeout and the resource use quite easily, but we have to manually monitor the Jobs and shut down the other solvers if one finds a solution.

4.6.2 Kubernetes ClusterRole

Since the job service has to start and stop kubernetes Jobs it needs to have the permission to actually do this. First, the deployment needs a kubernetes service account (this is a different concept than a Google Cloud service account). Then a ClusterRole with the necessary permissions were created and bound to the service account with a ClusterRoleBinding. Permissions are given to the ClusterRole by stating the resources and the verbs to interact with those resources. The resources and verbs can be seen in Listing 3, and reflect only the actions it needs to perform, as to preserve the principle of least privilege.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: executor
rules:
- apiGroups: ["", "batch"]
  resources: ["namespaces", "jobs", "pods", "pods/log"]
  verbs: ["get", "watch", "list", "create", "delete"]
```

Listing 3: The ClusterRole used for allowing the job service to manage other Pods.

4.6.3 Job Executor

The job executor is the part of the job service responsible for starting and controlling kubernetes Jobs dynamically. For this, the Python kubernetes client was used. This makes it possible to construct specs programmatically instead of using `yaml` files.

The job service also has the functionality of stopping a single solver or the entire job...

Describe how files are transferred...

5 Related Work and Discussion

6 Conclusion

7 ASSIGNMENTS

- Frederik
  Monitoring & Logging in Terraform [CURR]/Google
  Auth-service
  JWT
- Niels
  FS-service
  Secrets
  Docker
  Load-balancer/Gateway (NGINX)
  Testing
- Troels
  Terraform (svc-account, workload provider, cluster, gcr)

Google Cloud
Preliminaries
Kubernetes
Job service

- Abhijeet

- Lasse
  GitHub Actions
  Auth-service
  Passwords & Database Handling
  JWT
- Jakob
  Solver-service
  Architecture (technologies)
- Anders
  Frontend
- All
  Related Work and Discussion

References

[1] Jobs (kubernetes documentation). https://kubernetes.io/docs/concepts/workloads/controllers/job/. Accessed: 2022-12-27.