



Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Artificial Intelligence and Systems Engineering

Infrastructure design and deployment of a cloud-based graph generator

BACHELOR'S THESIS

Author

Gergő Buzás

Advisor

Dr. Kristóf Marussy

December 5, 2024

Contents

Kivonat	i
Abstract	ii
1 Introduction	1
2 Background	3
2.1 Java	3
2.2 Gradle	3
2.3 Jetty	4
2.4 JSON	4
2.5 Hypertext Transfer Protocol	4
2.6 REST API	5
2.7 Remote Procedure Call	5
2.8 gRPC	6
2.8.1 Core concepts	6
2.9 WebSocket	7
2.10 Shell	8
2.11 Shell script	8
2.12 Cloud Computing	9
2.12.1 Cloud Native	9
2.12.2 Scaling	10
2.12.3 Load Balancing	10
2.13 Amazon-related cloud technologies	11
2.13.1 Amazon Web Services(AWS)	11
2.13.2 Amazon Elastic Compute Cloud (EC2)	11

2.13.3	Amazon Elastic Container Service (ECS)	12
2.13.4	Virtual Private Cloud (VPC)	12
2.13.5	Elastic Load Balancing (ELB)	12
2.13.5.1	Application Load Balancer	13
2.13.6	Application Auto Scaling	13
2.14	Containers	13
2.14.1	Docker	14
2.14.2	Docker Compose	14
3	Overview	15
3.1	Frontend	15
3.2	Backend	15
3.3	Requirements	16
4	Considerations	18
4.1	Architectural designs in question	18
4.1.1	REST API	18
4.1.2	Remote Procedure Call	20
4.1.3	WebSocket server	20
4.1.4	Decision	21
4.2	Infrastructure	22
4.2.1	Elastic Compute Cloud (EC2)	22
4.2.2	Serverless approaches (Fargate and Lambda)	23
4.2.3	Elastic Container Services (ECS)	24
4.2.4	Kubernetes	24
4.2.5	Decision	25
5	Implementation	26
5.1	Starting point	26
5.1.1	ModelGenerationService	26
5.1.2	ModelGenerationWorker	27
5.2	Generator Client	28
5.2.1	GeneratorWebSocketEndpoint	29

5.2.1.1	WebSocket Configuration	29
5.2.1.2	Communication towards the server	29
5.2.1.3	Receiving from the server	29
5.2.1.4	Parsing responses from the queue	30
5.2.1.5	Error Handling and Resource Management	30
5.2.1.6	Scalability and Extensibility	30
5.2.2	ModelRemoteGenerationWorker	31
5.2.2.1	Core functionality	31
5.2.2.2	IGenerationWorker	31
5.2.2.3	Workflow	32
5.2.2.4	Impact	32
5.3	Generator Server	32
5.3.1	GeneratorServerEndpoint	33
5.3.1.1	Core Functionality	33
5.3.1.2	Integration into the workflow	33
5.3.1.3	Scalability and Extensibility	33
5.3.2	ModelGeneratorDispatcher	34
5.3.2.1	Key features	34
5.3.3	ModelGeneratorExecutor	35
5.3.3.1	Key features and responsibilities	35
5.3.4	ServerLauncher	35
5.4	Containerization	36
5.4.1	Preparation scripts	36
5.4.2	Docker images	36
5.5	Deploying on AWS	37
5.5.1	Task definitions	37
5.5.2	Service definitions	37
5.5.3	Load Balancer definition	38
5.6	Challenges during the implementation	38
5.6.1	Resource usage of development	38
5.6.2	Deduplication for docker images	39
5.6.3	Health check for the generator server	39

5.6.4	Application Load Balancer and NAT	40
6	Evaluation	43
6.1	Requirement analysis	43
6.2	Benchmarks	44
6.2.1	Test setup	44
6.2.1.1	External threats to validity	44
6.2.2	Results	45
6.2.2.1	First test	45
6.2.2.2	Second test	47
6.2.2.3	Third test	48
6.2.2.4	Fourth test	49
7	Conclusion	51
7.1	Future work	51
	Bibliography	53
	Appendix	55
A.1	Starting Refinery backend	55
A.2	Generation Client	56
A.3	Generator Server	57

HALLGATÓI NYILATKOZAT

Alulírott *Buzás Gergő*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2024. december 5.

Buzás Gergő
hallgató

Kivonat

A gráf analízisos feladatokat a parciális modellezésre lehet visszavezetni. A parciális modellezés formális módszereivel lehetőség nyílik egy adott feladat megoldásainak ellenőrzésére, releváns tesztadatok automatikus generálásával, amelyeket a parciális modell definícióira alapozunk.

Számos eszköz elérhető a parciális modellek elemzésére, azonban a használatukhoz szükséges fejlesztőkörnyezet beállítása a legtöbb felhasználó számára nehézséget okozhat. Emellett ezek az elemzési módszerek gyakran olyan erőforrásigénnyel járnak, amely meghaladja a személyi munkaállomások kapacitását.

Egy felhőalapú környezetben futtatható parciális modellezési eszköz használata megoldást jelent a számítási kapacitás hiányára, és kiküszöböli a felhasználók számára a szükséges konfigurációs terheket.

A Refinery egy parciális modellező program, amely két fő komponensből áll: egy parciális nyelvi szerkesztőből és egy gráf megoldóból.

A szakdolgozatom végső célja, hogy megvizsgáljam a Refinery gráf-modell generátor webalkalmazás jelenlegi infrastrukturális megvalósítását, és feltárjam azokat a területeket, ahol javításokra van lehetőség.

A jelenlegi infrastruktúra bemutatását követően megvizsgálom, hogyan lehetne javítani a skálázhatóságát, részletezve az egyes megvalósítási lehetőségek előnyeit és hátrányait, valamint összehasonlítva, hogy ezek mennyire felelnek meg a dolgozat előtt meghatározott követelményeknek.

A végső megoldást az Amazon AWS infrastruktúráján, az Elastic Container Services-t használva valósítottam meg.

A megoldásom értékeléséhez Python szkriptekkel végeztem tesztek, amelyek azt vizsgálták, hogyan javultak a válaszidők több egyidejű generálási kérés esetén.

Abstract

Graph analysis problems can be approached through partial modeling. By leveraging formal methods of partial modeling, these problems can be verified using automatically generated test data derived from the partial model definitions.

While various tools exist for partial model analysis, setting up the necessary development environment can be challenging for many users. Moreover, the computational demands of these methods often exceed the capacity of personal workstations.

A cloud-based partial modeling tool can address these challenges by providing the required computational resources and eliminating the complexity of local configuration.

Refinery is an example of such a tool, comprising two primary components: a partial language editor and a graph solver.

The end goal of my thesis, is to examine the current infrastructural implementation of the Refinery graph-model generator web application, and find ways where it can be improved.

After describing the current infrastructure, I look into ways, how the scalability can be improved, and go into detail regarding the pros and the cons of each implementation. I compare how each of them satisfy the specifications that had been made before the thesis.

The final implementation is deployed using Amazon's AWS infrastructure and Elastic Container Services.

For evaluating my solution, testing was performed via Python scripts which checked how response times improved for multiple simultaneous generation requests.

Chapter 1

Introduction

Graph analysis problems can be approached through partial modeling. By leveraging formal methods of partial modeling, these problems can be verified using automatically generated test data derived from the partial model definitions.

Refinery [22] is a cloud-based "Graph Solver as a Service" web application. It supports problem definition for problems which can be mathematically retrieved to a graph-related problems. The application has an editor, which constantly checks for the correctness of our problem definition: both semantically and syntactically. The editor uses the custom-made partial modeling language of Refinery.

Users of the application can generate a model, representing their defined graph problems. The generation of said model can be quite resource heavy and may take several seconds to perform. This generation is currently performed in the same backend server, towards which the frontend clients of the application connect with.

Under increased usage from the users, new connections made to the backend server might not be possible because of the increased workloads on the server. Furthermore, response times might make the user experience worse for the users, due to the increased response times or timeouts.

This issue should only arises, under extreme usage-circumstances. The deployed infrastructure in production already mitigates this issue by the use of auto scaling and load balancers. However, the model generation is still done on the same server as the backend server of the application.

The scope of my thesis is the design and implementation of a generator service, which can be integrated into the workflow of the model generation. By implementing a generator service, it is hoped, that the response times under heavy usage can improve. All of this should be performed by putting extra care into the scalability of the application. The costs associated with the improvements should also be considered.

In Chapter 2, I introduce the technologies, that are already or could be used for implementation of the Refinery web application. This is done with extra attention put towards the model generation part of the application.

In Chapter 3, I go over the current implementation of the application. I describe the frontend and the current backend of the application. At the end, I set requirements for the modification of the application, which should be considered for all of the latter parts of the thesis.

In Chapter 4, I list the possible implementations for the generator service. The advantages and disadvantages of all implementation possibilities are considered both architecturally and infrastructurally. I compare the possibilities and draw a conclusion, of what should be implemented.

In Chapter 5, I implement the architectural and the infrastructural changes that were designed in Chapter 4.

In Chapter 6, I evaluate how the changes implemented in chapter 5 satisfy the requirements made in section 3.3. I benchmark the new implementation compared to the old implementation. I draw a conclusion, how succesful the project was.

Last, but not least, in Chapter 7, I summarize the results and describe the future improvements that can or need to be done in the future.

Chapter 2

Background

In this chapter I'll introduce the technologies and the phrases that need to be described for proper understanding of my thesis work. Some of these technologies were only considered during the designing part of the thesis, while others were also present during the implementation.

2.1 Java

Java [27] is a widely-used, object-oriented programming language known for its platform independence, achieved through the Java Virtual Machine (JVM).

The backend of Refinery was programmed using the Java programming language. In the scope of this thesis, Java the main programming being used.

Java supports core object-oriented programming principles like encapsulation, inheritance, and polymorphism, promoting code modularity and reusability. This comes in handy for the modification of the already existing codebase, while preserving the old implementation.

When working with Java, a Java Development Kit (JDK) has to be installed on the system. The Java Development Kit (JDK) is a distribution of Java technology by Oracle Corporation. The Refinery backend project uses JDK 21.

2.2 Gradle

Gradle [25] is the build automation tool used for the Refinery repository and it greatly helps the package management, building, testing and the local deployment of the project / subprojects.

Gradle supports Java as a programming language, which is the language in which the backend of Refinery was written.

Gradle was designed for multi-project builds, like the Refinery project, consisting of several subprojects. It operates based on a series of build tasks that can run serially or in parallel. This can come in handy for the backend and the generator service running simultaneously, locally as separate subprojects.

Incremental builds and caching of build components are supported, which provide faster build times during the development.

2.3 Jetty

Eclipse Jetty [13] provides a highly scalable and memory-efficient web server and servlet container, supporting many protocols such as HTTP/3,2,1 and WebSocket. The original backend of Refinery uses Jetty as its server and could be a viable option for the future generation microservice aswell.

2.4 JSON

JSON (JavaScript Object Notation) [1] is a lightweight data-interchange format.

JSON is built on two structures:

- A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

The Refinery web application uses JSON messages as a way to communicate model generation request states, results and metadata between the frontend and the backend.

2.5 Hypertext Transfer Protocol

HTTP (Hypertext Transfer Protocol) [19] is an application layer protocol, designed for enabling communication between clients and servers. HTTP works as a single request - single response protocol between the communicating endpoints.

HTTP requests are human readable. They consist of the following parts:

- **Method:** Indicates what type of request the client wants to send to the server. The most used are the following:
 - GET requests indicate resource fetching
 - POST requests indicate resource upload

- PUT / UPDATE requests indicate updating an existing resource
- DELETE requests indicate removal of an existing resource
- **Path:** The path of the resource for which we send the request
- **Version:** The version of the HTTP protocol
- **Headers:** Provide additional information for the server that is receiving the request.
- **Body:** For some requests (e.g.: POST), additional info is provided here.

2.6 REST API

A REST API [17] (also called a RESTful API or RESTful web API) is an application programming interface (API) that conforms to the design principles of the representational state transfer (REST) architectural style.

REST API's via the use of the HTTP protocol utilize a single request - single response communication methodology. REST APIs communicate through HTTP requests to perform standard database functions like creating (POST), reading (GET), updating (PUT) and deleting (DELETE) records (also known as CRUD operations) within a resource.

REST API's can be implemented really easily due to the efforts of many server framework providers. For Java, these include the very popular Spring and Jetty frameworks.

At the start of the thesis, Refinery doesn't use or provide any REST APIs, but the implementation of one for the model generation requests could be a viable option.

2.7 Remote Procedure Call

In distributed computing, a remote procedure call (RPC) [28] is when a computer program causes a procedure (subroutine) to execute in a different address space (commonly on another computer on a shared computer network), which is written as if it were a normal (local) procedure call, without the programmer explicitly writing the details for the remote interaction. That is, the programmer writes essentially the same code whether the subroutine is local to the executing program, or remote. This is a form of client-server interaction (caller is client, executor is server), typically implemented via a request-response message passing system.

At the start of the thesis, Refinery doesn't use any RPC libraries, however the use of one (like gRPC) could be used for the model generation service implementation.

2.8 gRPC

gRPC (gRPC Remote Procedure Calls) [26] is a cross-platform high-performance remote procedure call (RPC) framework. gRPC uses HTTP/2 for transport. It generates cross-platform client and server bindings for many languages (which includes Java).

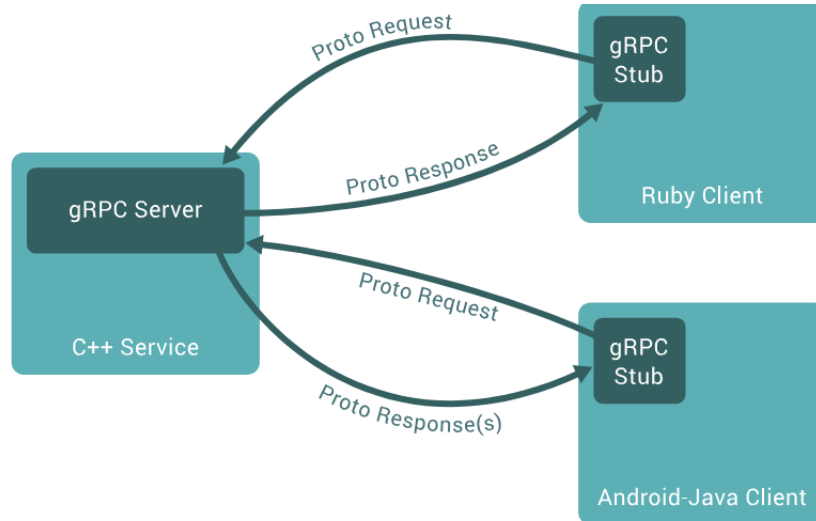


Figure 2.1: gRPC server and the clients, as represented in gRPC documentation [15]

In gRPC, a client application can directly call a method, which is defined on a server application on a different machine, as if it were a local object. gRPC clients and servers can run and talk to each other in a variety of environments. This includes both on-prem, and more importantly in our case, cloud-based environments. [15]

2.8.1 Core concepts

Some of the core concepts of gRPC include, but not limited to [16]:

- **Server streaming RPC:** The server responds with multiple responses to the client, which sends a single request. After sending all its messages, the server's status details (status code and optional status message) and optional trailing metadata are sent to the client. This completes processing on the server side. The client completes once it has all the server's messages.
- **Deadlines/Timeouts:** gRPC allows clients to specify how long they are willing to wait for an RPC to complete. On the server side, the server can query to see if a particular RPC has timed out, or how much time is left to complete the RPC.
- **Cancelling an RPC:** Either the client or the server can cancel an RPC at any time. A cancellation terminates the RPC immediately so that no further work is done. The changes made before the cancellation are not rolled back.

The current implementation of Refinery doesn't use the gRPC framework. However, it can be a possible solution for the generator microservice implementation. The remote method execution can lower the resource usage of the backend server.

2.9 WebSocket

The WebSocket Protocol [18] enables two-way communication between a client and a remote host. The goal of this technology is to provide a mechanism for applications that need two-way communication with servers that does not rely on opening multiple HTTP connections, but using one open connection, for the duration of the communication.

The following part is taken from RFC 6455 [18], which describes why the creation of WebSocket protocol was needed.

"Creating web applications using only HTTP, that need bidirectional communication between a client and a server has a major problem. It requires the abuse of HTTP to poll the server for updates while sending upstream notifications as distinct HTTP calls. This results in a variety of problems:"

1. *"The server is forced to use a number of different underlying TCP connections for each client: one for sending information to the client and a new one for each incoming message."*
2. *"The wire protocol has a high overhead, with each client-to-server message having an HTTP header."*
3. *"The client-side script is forced to maintain a mapping from the outgoing connections to the incoming connection to track replies."*

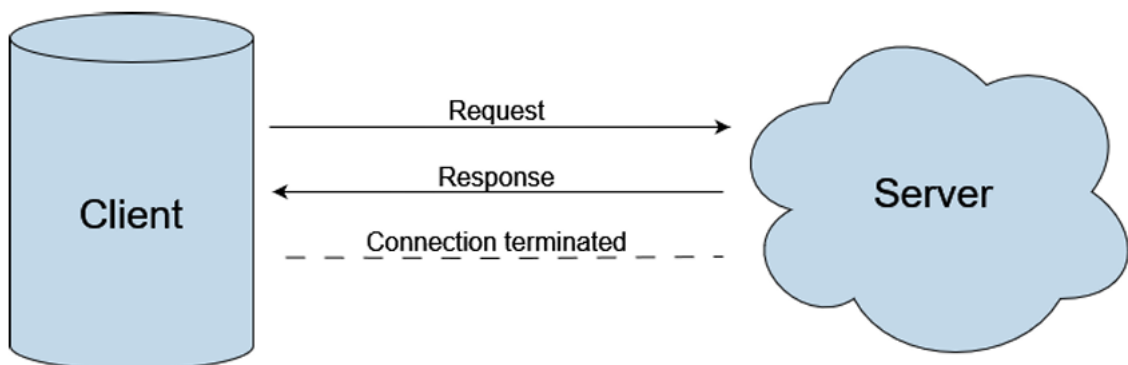


Figure 2.2: Regular HTTP connection

"A simpler solution would be to use a single TCP connection for traffic in both directions. This is what the WebSocket Protocol provides. A solution for these HTTP polling scenarios."

After the initial WebSocket handshake, the communication protocol is upgraded to WebSocket from HTTP. If no communication is happening, the connection is kept alive between the two communicating endpoints via the use of ping and pong messages. If a ping sender endpoint doesn't receive a pong response, the connection is lost, and the endpoints are disconnected. Other than that, the protocol allows for full bidirectional real-time messaging to the endpoints.

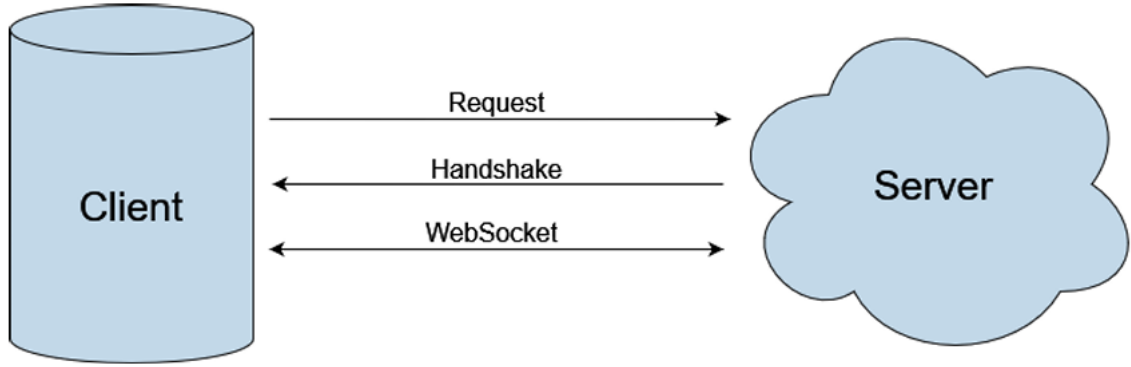


Figure 2.3: WebSocket connection over HTTP

The current implementation of Refinery already utilizes WebSocket. The frontend communicates editor changes and model generation states, results and metadata via the use of WebSocket. However, for the implementation of the generator microservice, it could also be a viable solution.

2.10 Shell

In computing, a shell [29] is a computer program that exposes an operating system's services to a human user or other programs. In general, operating system shells use either a command-line interface (CLI) or graphical user interface (GUI), depending on a computer's role and particular operation. It is named a shell because it is the outermost layer around the operating system.

In addition to shells running on local systems, there are different ways to make remote systems available to local users; such approaches are usually referred to as remote access (like the use of SSH) or remote administration.

Within the Refinery project, the use of a CLI based Unix shell (like bash) is needed for the execution of shell scripts. Furthermore, it can come in handy as a remote access for our VMs via the use of SSH, which allows debugging our infrastructure.

2.11 Shell script

A shell script [30] is a computer program designed to be run by a Unix shell, a command-line interpreter. Typical operations performed by shell scripts include file manipulation,

program execution, and printing text. The term is also used more generally to mean the automated mode of running an operating system shell.

The Refinery project has Unix shell scripts, which automate the build process and the containerization of the application.

2.12 Cloud Computing

The following description is taken from a popular public cloud provider’s, Google’s cloud computing introduction:

‘Cloud computing [14] is the on-demand availability of computing resources (such as storage and infrastructure), as services over the internet. It eliminates the need for individuals and businesses to self-manage physical resources themselves, and only pay for what they use.’

‘Cloud computing service models are based on the concept of sharing on-demand computing resources, software, and information over the internet. Companies or individuals pay to access a virtual pool of shared resources, including compute, storage, and networking services, which are located on remote servers that are owned and managed by service providers.’

The infrastructure of Refinery is described as a public cloud deployment. This means, that the infrastructure is run by third-party cloud service providers. The providers offer compute, storage and network resources over the internet.

The cloud services used for the deployment of Refinery are called ‘Infrastructure as a service (IaaS)’. IaaS service means [14] *‘on-demand access to IT infrastructure services, including compute, storage, networking, and virtualization. It provides the highest level of control over the IT resources and most closely resembles traditional on-premises IT resources.’*

2.12.1 Cloud Native

This following definition of Cloud Native is taken from AWS’s website [23]:

‘Cloud native is the software approach of building, deploying, and managing modern applications in cloud computing environments. Modern companies want to build highly scalable, flexible, and resilient applications that they can update quickly to meet customer demands. To do so, they use modern tools and techniques that inherently support application development on cloud infrastructure. These cloud-native technologies support fast and frequent changes to applications without impacting service delivery, providing adopters with an innovative, competitive advantage.’

2.12.2 Scaling

The following explanation of scaling is taken from VMWare's website [24]:

'Scaling in cloud computing refers to the ability to increase or decrease IT resources as needed to meet changing demand. Scalability is one of the hallmarks of the cloud and the primary driver of its exploding popularity with businesses. Scaling allows the businesses to pay only for the resources that they truly need.'

- **Vertical scaling:** (also referred to as "scaling up" or "scaling down") You add or subtract power to an existing cloud server upgrading memory (RAM), storage or processing power (CPU). Usually this means that the scaling has an upper limit based on the capacity of the server or machine being scaled; scaling beyond that often requires downtime.
- **Horizontal scaling:** (also referred to as "scaling in" or "scaling out"), You add more resources like servers to your system to spread out the workload across machines, which in turn increases performance and storage capacity. Horizontal scaling is especially important for businesses with high availability services requiring minimal downtime.'

2.12.3 Load Balancing

The following parts regarding load balancing are taken from Cloudflare at [10]:

'Load balancing [10] is the practice of distributing computational workloads between two or more computers. On the Internet, load balancing is often used for dividing network traffic among several servers. This reduces the strain on each server and makes the servers more efficient, speeding up performance and reducing latency. Load balancing is essential for most Internet applications to function properly. By dividing user requests among multiple servers, user wait time is vastly reduced. This results in a better user experience.'

'Load balancing is handled by a tool or application called the load balancer. A load balancer can be either hardware-based or software-based.'

- **Hardware load balancers** require the installation of a dedicated load balancing device.
- **Software-based load balancers** can run on a server, on a virtual machine, or in the cloud.

Content delivery networks (CDN) often include load balancing features. When a request arrives from a user, the load balancer assigns the request to a given server, and this process repeats for each request. Load balancers determine which server should handle each request based on a number of different algorithms. These algorithms fall into two main categories: static and dynamic.'

- **Static load balancing** algorithms distribute workloads without taking into account the current state of the system. A static load balancer will not be aware of which servers are performing slowly and which servers are not being used enough. Instead it assigns workloads based on a predetermined plan.

Static load balancing is quick to set up, but can result in inefficiencies. Round robin DNS and client-side random load balancing are two common forms of static load balancing.

- **Dynamic load balancing** algorithms take the current availability, workload, and health of each server into account. They can shift traffic from overburdened or poorly performing servers to underutilized servers, keeping the distribution even and efficient. However, dynamic load balancing is more difficult to configure. A number of different factors play into server availability: the health and overall capacity of each server, the size of the tasks being distributed, and so on.

There are several types of dynamic load balancing algorithms, including least connection, weighted least connection, resource-based, and geolocation-based load balancing.

2.13 Amazon-related cloud technologies

2.13.1 Amazon Web Services(AWS)

Amazon Web Services (AWS) [3] is the public cloud offering by the parent company Amazon. AWS has a vast offering of services, which help its end-users manage the infrastructures needed to host their own applications, microservices, APIs and what not. AWS services are delivered to customers via a network of AWS server farms located throughout the world.

The project uses Amazon Web Services for the infrastructure of Refinery.

In the following sections (2.14.2 - 2.14.6), I will go into detail what some of those services are and how they can be beneficial for the users of those services.

2.13.2 Amazon Elastic Compute Cloud (EC2)

Amazon Elastic Compute Cloud (Amazon EC2) [4] provides on-demand, scalable computing capacity in the Amazon Web Services (AWS) Cloud. Using Amazon EC2 reduces hardware costs so users can develop and deploy applications faster. Amazon EC2 can be used to launch as many or as few virtual servers as needed, configure security and networking, and manage storage. Capacity can be added (upscale) to handle compute-heavy tasks, such as monthly or yearly processes, or spikes in website traffic. When usage decreases, the capacity can be reduced (downscale) again.

2.13.3 Amazon Elastic Container Service (ECS)

Amazon Elastic Container Service (Amazon ECS) [5] is a fully managed container orchestration service that helps the deployment, management, and scaling of containerized applications. It is integrated with both AWS and third-party tools, such as Amazon Elastic Container Registry and Docker. This integration makes it easier for users to focus on building the applications, not the environment.

There are three main layers in Amazon ECS:

1. **Capacity:** The infrastructure where the users runs their containers. Options include Amazon EC2, AWS Fargate and On-prem virtual machines and servers (infrastructure outside of Amazon)
2. **Controller:** The deployment and management of the applications that are run on the containers. The Amazon ECS scheduler is the software that manages / controls the applications.
3. **Provisioning:** The tools used for interfacing with the ECS scheduler to deploy and manage the applications. and containers. Options include the AWS Management Console (Web GUI), AWS Command Line Interface (CLI), AWS Software Development Kits (SDKs).

Amazon ECS can be a valid option for the provisioning of Refinery's services.

2.13.4 Virtual Private Cloud (VPC)

With Amazon Virtual Private Cloud (Amazon VPC) [9], users can launch AWS resources in a logically isolated virtual network that is defined by the user. This virtual network closely resembles a traditional network that the user would operate, with the benefits of using the scalable infrastructure of AWS.

2.13.5 Elastic Load Balancing (ELB)

Elastic Load Balancing [6] automatically distributes incoming traffic across multiple targets, such as EC2 instances, containers, and IP addresses. It monitors the health of its registered targets, and routes traffic only to the healthy targets. Elastic Load Balancing scales load balancers as the incoming traffic changes over time.

Elastic Load Balancing supports the following load balancers:

- Application Load Balancers
- Network Load Balancers
- Gateway Load Balancers
- Classic Load Balancers

2.13.5.1 Application Load Balancer

An Application Load Balancer [6] functions at the application layer, the seventh layer of the Open Systems Interconnection (OSI) model. After the load balancer receives a request, it evaluates the listener rules in priority order to determine which rule to apply, and then selects a target from the target group for the rule action. Routing is performed independently for each target group, even when a target is registered with multiple target groups. Routing algorithm can be configured at the target group level. The default routing algorithm is round robin; alternatively, the least outstanding requests routing algorithm can be used as well.

Health checks can be configured, which are used to monitor the health of the registered targets so that the load balancer can send requests only to the healthy targets within a target group.

The application load balancer can be utilized for the Refinery applications. This allows our scaling services (like the backend and the generator) to receive requests based on the set load balancing algorithm. The services should be setup as separate target groups.

2.13.6 Application Auto Scaling

Application Auto Scaling [2] is a web service for developers and system administrators who need a solution for automatically scaling their scalable resources for individual AWS services beyond Amazon EC2.

Application Auto Scaling allows automatic scaling of scalable resources according to conditions that are defined.

The auto scaling is useful for the scaling of containerized Refinery services (backend, and generator service).

2.14 Containers

Containerization [20] is the packaging together of software code with all its necessary components like libraries, frameworks, and other dependencies so that they are isolated in their own container.

This is done so that the software or application within the container can be moved and run consistently in any environment and on any infrastructure, independent of that environment or infrastructure's operating system. It is a fully functional and portable computing environment.

The "lightweight" or portability characteristic of containers comes from their ability to share the host machine's operating system kernel, negating the need for a separate operating system for each container and allowing the application to run the same on any infrastructure even within virtual machines (VMs).

Containers are often used to package single functions that perform specific tasks—known as a microservice. Microservices are the breaking up of the parts of an application into smaller, more specialized services.

The services of the Refinery application should be developed with containerization in mind. This makes the distribution and deployment of the application much easier.

2.14.1 Docker

Docker [11] is an open platform for developing, shipping, and running applications. Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security lets you run many containers simultaneously on a given host.

Docker containers can be created by running docker images. Images are templates containing the read-only instructions needed to create the container running our application. To build an image, the user has to write a Dockerfile where they specify all of the steps needed to build their image (like linking libraries, exposing ports, setting environment variables, etc.)

The docker container is the running instance of the docker image. Those can be run by the use of the docker cli.

The implemented Refinery services can be containerized via the use of Docker images. The images can be uploaded to the Docker Registry, providing easier distribution.

2.14.2 Docker Compose

Docker Compose [12] is a tool for defining and running multi-container applications. It is the key to unlocking a streamlined and efficient development and deployment experience.

Compose simplifies the control of the entire application stack, making it easy to manage services, networks, and volumes in a single, comprehensible YAML configuration file. With a command, all of the services from the configuration file can be started.

Within the scope of the thesis, Docker Compose can be used for the local deployment of the containerized Refinery backend and generator services. This provides manual testing abilities before deploying the application on the cloud.

Chapter 3

Overview

The goal of the project is to examine how the scalability of the Refinery web-application could be improved. The scalability of an application is basically allowing our application to handle bigger loads and handle more simultaneous connections. This can mean reducing the operating costs and / or having better response times for each requests that the users of the application make.

As such, the current infrastructure has to be understood, and only then can ways be found, how improvements can be made.

Scalability can be greatly improved via the modification of the backend architecture of an application, so that is naturally what I am going to describe in greater detail.

After examining the current infrastructure and implementation of the graph generator, I will go into detail about the different approaches and ideas of how the backend could be restructured, so that we can achieve our goal in mind: improved scalability.

3.1 Frontend

The frontend of the Refinery web application is implemented using the React library. As a result, most of it is written in Typescript. It basically works as a text editor, where the users can define the rules for their graph models, and see whether their rules are semantically / syntactically correct. If so, they can generate a model based on their defined rules via the press of a button, which sends the states of the text editor to the backend. The generation results are shown on the frontend.

3.2 Backend

The backend is a Java Jetty application. The clients connect to this server via WebSocket. In the current implementation, this connection is used to send live information to the editor server, regarding the state of the user-defined model. This state is based on what the user

of the application wrote in the text editor in the rule-defining partial modelling language of Refinery. After each modification, only the differences from the last saved state are sent to the server.

The server uses XText for keeping track of the state of the model. It is a library provided by Eclipse, and it makes the creation of domain specific languages (such as the one, used for Refinery) much easier. The most important usecase for XText, is the saving and verification of the text, written in the editor.

When a client (= an instance of the Refinery Frontend) makes a connection to backend server, a WebSocket connection is made. Through this open WebSocket connection, the client continuously sends JSON messages to the server.

First, the initial update of the text is sent to the server. This initial update contains an example model, which also showcases for the user, how the language can be used for defining a graph problem.

After this initialization, only differences are sent to the backend, so that the edits made by the user can also be applied to the state on the backend session. After each user edit/modification, the validity of the model is verified by the server: both semantically and syntactically, and sent back to the client.

If no action is made by the user, the connection between the client and the server is kept alive via the use of the ping and pong messages of WebSocket.

If the user finished the creation of the graph model and the generate button is pressed on the client-side, a message containing the generation service request is sent to the server. The server then proceeds to generate the model based on the user input and sends the belonging states, results and metadata to the client.

This model generation may take a significant amount of time and many computing resources. If vast amount of users were to use the service at the same time, the increased generation times could lead to an unpleasing user experience and new user connections might also be very slow.

As a result it would be wise to restructure the Refinery infrastructure in a way, where the generation happens on a separate server, acting like a microservice. This way, the load would be taken off from the basic backend of Refinery, and the newly created microservice would be the one responsible for the creation of the model. Our task is the creation of this generator microservice.

3.3 Requirements

Now that we have an idea, of what we have to perform to enhance the scalability of Refinery, we have to set some requirements for the restructuring of the backend architecture and for the creation of the GeneratorServer service.

The requirements for the restructuring of Refinary should be as follows:

1. **The backend of the application should be scalable.** This should be done, so that under increased usage the user experience should not be compromised.
2. **The client should be able to cancel the model generation.** If the user reconsiders their need of the generated model, it should be a possibility that they can cancel it. Furthermore, canceling a not needed model generation improves resource usages (and by this, the scalability aswell).
3. **The client must be able to show the state of the model generation.** By this, the client-side of the application can showcase where the state of the model generation is at. Potential errors during the generation can also be showcased via the use of the state representation.
4. **There must be a timeout set for the model generation.** During the unlikely event of a bug in the generation implementation, a timeout can limit the duration of the model generation process. This also limits heavy resource usage durations. Furthermore, if the connection would be permanently lost with the frontend during the generation, the timeout could stop the no longer needed generation.
5. **(Optional) The implementation should allow the creation of other type of cliens other than web browser-based clients.** In the future the application could be extended to run as a separate service. The model generations could also be used by other developers, leveraging an API for the generations.
6. **(Optional) The editor and generator should be deployed together.** This should improve the overall development experience of the application. By deploying the two services together, the live infrastructure can be simulated on the local environment.

Chapter 4

Considerations

4.1 Architectural designs in question

As the requirements have been set, we can examine the different ways the generation of the model could be moved to a separate server. In this section, I'll go into detail about the different architectures that came up during the planning phase. Last, but not least, I'll explain, why I chose the WebSocket implementation after carefully researching my possibilities.

4.1.1 REST API

The first architecture that came to my mind, was a **REST API (2.6)**. Using the Spring boot framework would allow for fast development of our REST API. The state of the partial modelling editor would be sent to the REST API as a POST request, and the result would be returned as a Json, which the client would parse, and output the model accordingly. However, it was still in question how the generated model should be returned to the client.

- **Long return:** The first implementation that came up, was a regular REST API, where the response of the API would only be received, after the model generation had been finished. An implementation plan can be seen in figure 4.1. As we know REST responses shouldn't take a long time to respond to received requests. The Generator is probably the slowest service of Refinery, so responding with the model details may take a fairly long time. As such, the purpose of a single request - single response REST API would be invalid / way too slow for our usecase.
- **Long polling:** Here, the REST API would respond immediately with a "Starting generator service" message, had it received a request. The REST API would start and perform the generation, while the backend would constantly try to fetch the results from the API. An example implementation can be seen in 4.2

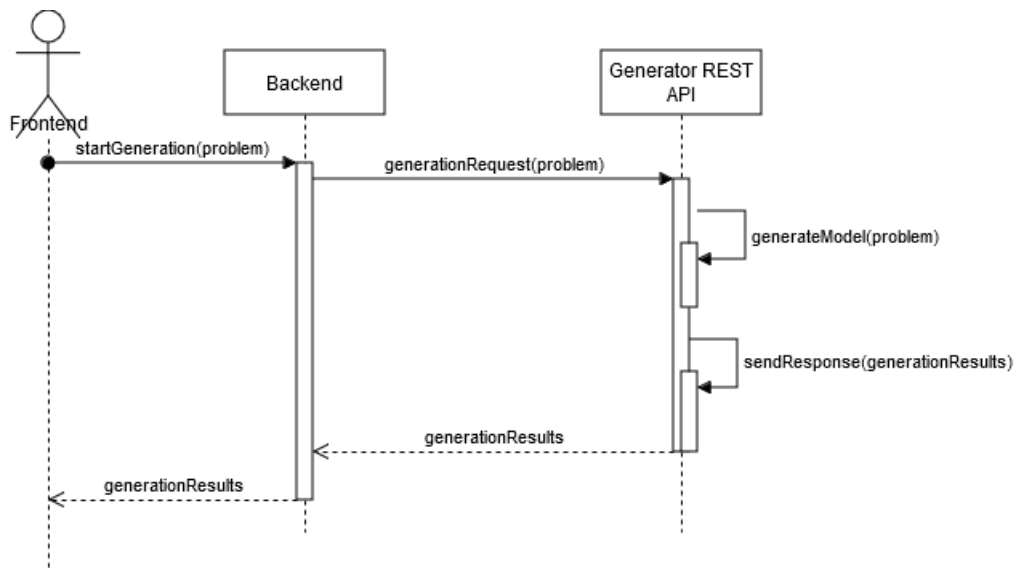


Figure 4.1: Possible long return implementation of generator REST API

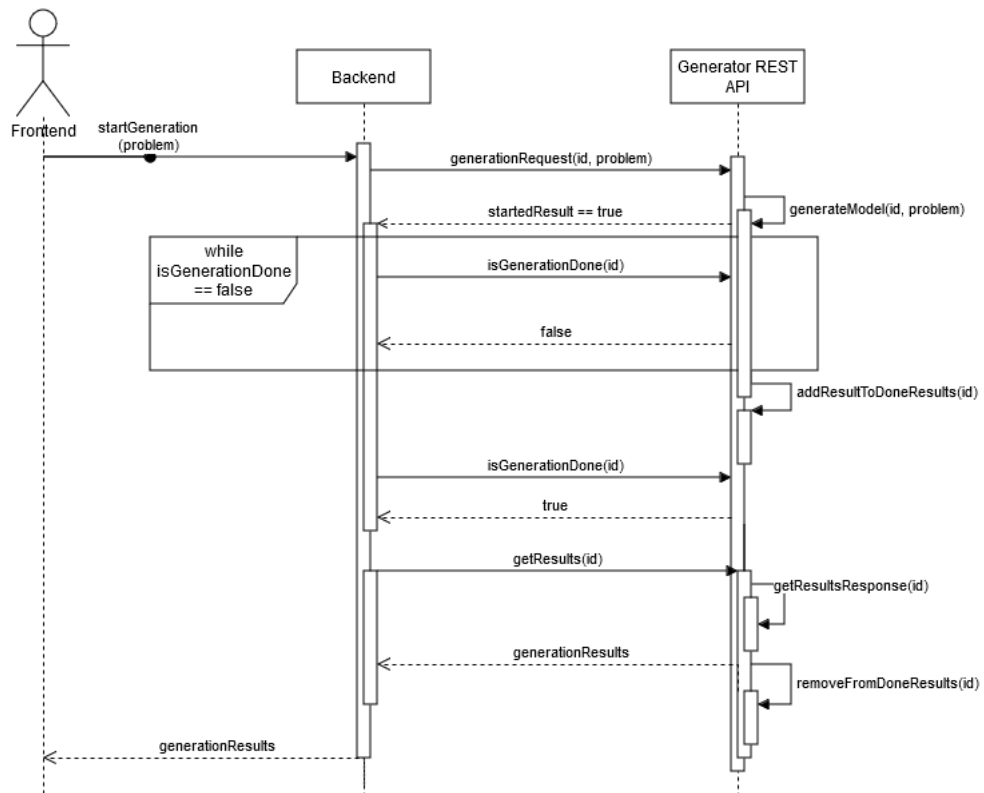


Figure 4.2: Possible long polling implementation of generator REST API

The limitation of both implementations is the fact that the request cannot be cancelled from the client side. As that is one of the main requirements of the architectural restructuring, this is a fact that cannot be looked over, so I concluded that another architecture should be looked at for the implementation.

4.1.2 Remote Procedure Call

The second architecture that came up during the planning phase was Remote Procedure Call (RPC) (2.7). The generation is pretty much a procedure, so calling it remotely sounded like a wise choice.

Using gRPC (2.8) would provide the timeout logic, and other clients could also be created in the future, as the generator server would function as a gRPC server. The newly created clients would only need to act as a gRPC client, and send requests according to the server's specification.

gRPC would allow us to use a one request - multiple responses implementation, for representing the current state of the generation. The cancellation of the generation can also be done via gRPC's provided cancelation code examples. The thread running the generation of the model could be stopped upon receiving a cancel request.

gRPC communicates over HTTP 2.0, so the newer HTTP protocol would have to be used, in order to utilize the usage of the gRPC library.

4.1.3 WebSocket server

Last but not least, comes WebSocket (2.9). This basically would work the same way as the REST API long polling implementation, however, the connection would not be closed after the initial request. This is due to the way WebSocket works.

Due to the nature of WebSocket, after the model generation request is sent out by the client, the connection is kept alive, and the server may send back multiple responses over the same connection. This comes in handy for the representation of the state. The server can continuously send to the client, where the model generation is currently at.

The current implementation for the communication between the web client and the editor server is already using a WebSocket connection. As such, the same cancelation and timeout logic can be applied for the generator server, meaning that those two requirements would also be satisfied. After all of these considerations WebSocket seemed like the best implementation moving forward, but the way it would be implemented had still to be decided:

- **Using a separate connections from the webclient side** This way, the webclient (or "frontend") would be the one making a connection with the generator server. This way there would be two connections created from the webclient: one towards the generator server, and a separate one towards the editor server.

- **Using the same connection as the editor** In this implementation, the editor server (backend) would function as a "proxy" towards the generator microservice. This is made possible by the fact that in the current implementation, the webclient already makes a WebSocket connection towards the editor server, and the editor server is the one performing the generation.

4.1.4 Decision

After investigating and evaluating the possible implementations, I decided to choose the WebSocket implementaiton utilizing the 'backend as a proxy towards the microservice' due to the reasons below.

The **REST API implementation** should not be used as the states where an ongoing generation stands would be hard to represent without the use of a workaround. Furthermore, the cancelation of ongoing generations would also need some workarounds for it to be implemented.

The **gRPC implementaiton** could satisfy all of the project requirements. The ongoing generations could be cancelled and timeout could be handled too. The gRPC implementation would introduce new dependencies for both the backend and the generator server projects. The final image of the backend and the generator would grow even bigger, but neither the overall user experience, performance or the scalability would be improved compared to the WebSocket implementation.

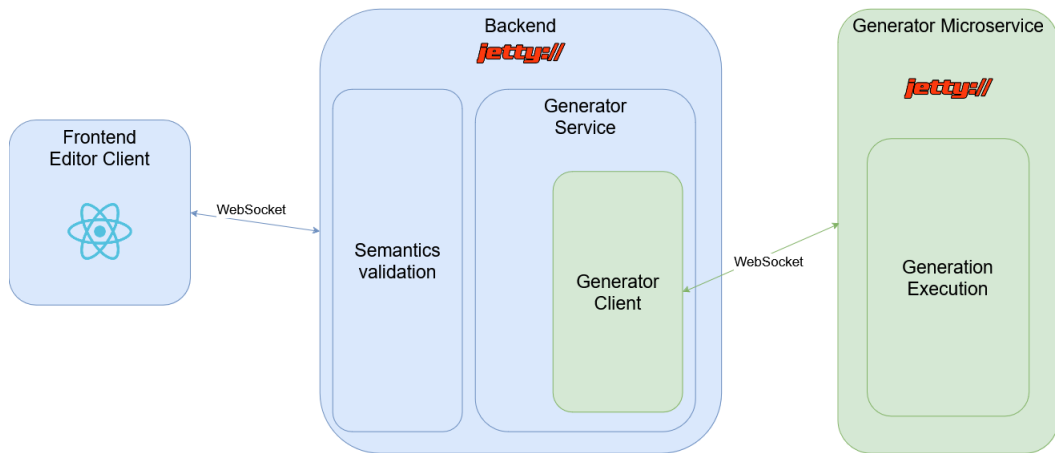


Figure 4.3: Final proposed architectural changes

The **WebSocket implementation** proposed in figure 4.3 could satisfy all of the project requirements. Furthermore, WebSockets are already used in the backend project so it should be straightforward to use in the microservice implementation aswell. This info also assures me, that there won't be any dependency issues related to this implementation. Another key factor, is that via the use of WebSockets, no additional dependencies have to be added. The editor server already communicates with the clients over WebSocket,

through which the model generation requests are sent over. Using WebSockets and the "backend as a proxy towards the microservice" implementation could reduce the overall work needed for the refactor, as the Frontend of Refinery doesn't have to be touched. The changes to the architecture of the application will not be noticable for the client, yet the performance may improve.

4.2 Infrastructure

Now that we have gone over the architectural changes that need to be implemented, we need to take a look at how the scalability of the application is really going to be improved, and how the application should be deployed. For this I'll utilize Amazon Web Services, as it is one of the leading industry standard cloud providers and the current Refinery is also deployed on AWS platforms.

AWS currently offers many options for running our application, both server and serverless methods. When thinking about server methods, the ways include EC2, ECS, EKS. When talking about serverless approaches Fargate and Lambda are the solutions that come to mind. In the following section I'll describe how each of these services could be used for our infrastructure, and come to a conclusion, of how our infrastructure will be deployed. When deciding about the infrastructure, the key considerations are going to include the specifications for our microservice, how easy it is to maintain our deployed application, and last, but not least the costs affiliated with said infrastructure.

4.2.1 Elastic Compute Cloud (EC2)

Amazon EC2 is basically a fully functional VM instance on Amazon's infrastructure. This is the most bare-bones approach as this would be the same, as running a linux server on any on-prem instances. By containerizing our application, no additional libraries would need to be installed: we just need to run our docker image, and the container would be ready to go.

Another advantage of this approach is the price. This is one of the cheapest services of Amazon, with instance hourly rates starting as little as \$0.0048, to going as high as \$0.1728 for a single instance (key factors in pricing include CPU cores, memory and network performance).

However, advantages of this approach stop right there. When it comes to maintaining our application, EC2 doesn't offer much. On the AWS dashboard, we could see, whether our instance is running, but nothing about the state of our application running in the docker container. Another disadvantage, is that scaling is pretty much non-existent from the get-go. Auto Scaling Groups and Load Balancers can be created by using EC2 instances, but the setup of them is tedious.

4.2.2 Serverless approaches (Fargate and Lambda)

As our generator application is gonna be containerized, serverless options such as AWS Fargate and AWS Lambda can come into play.

With AWS Fargate, one of the main advantages is that only real computing usage is billed, and downtimes aren't billed, when the service isn't used.

However, in practice, prices associated with Fargate seem to be much higher, compared to EC2 instance hourly rates: \$0.04656 dollars per CPU core and \$0.00511 per GB per hour. As a result, Fargate sounds great on paper, but with increased usage monthly costs can increase significantly.

With AWS lambda, there would not be a need for containerization, as AWS runs code snippets from compressed ZIP folders, if the code snippets are in supported languages. AWS lambda supports java, so that is a possibility. The pricing depends on the memory that we allocate for our lambda functions and the amount of time they take to complete. If we were to allocate 512MB of memory for our function, the price per 1ms in the Europe region would be \$0.0000000083. That price sounds better than the former two options. However, lambda is usually used for non-resource demanding codes. As such, the response times could become much worse than if we were to run the generation with a Fargate or EC2 approach.

Both solutions would allow for scaling. Fargate scales horizontally meaning that the number of instances running can be increased. Lambda scales vertically, with the resources allocated for the function.

Most crucially, the biggest downside of running our microservice with a serverless approach is that it would not satisfy many requirements that were made in the application specification. If a generation would be cancelled by the user, there would be no easy way of cancelling it, as essentially these services are running code snippets, where runtime modification is not possible without workarounds / AWS SDK usage. This would also make our application dependent on AWS, as the migration to a different cloud provider / deployment on a local instance would get more complicated, or even impossible, without changes to the codebase.

Last, but not least, the state of the generation would be tedious to send back to the backend and the frontend. We would either have to give up on the generation state representation, or create some kind of workaround (such as an API endpoint) where the state of the generation is stored. However, that would make the infrastructure much more complicated, and the whole premise of a serverless approach would be destroyed.

After all of these considerations, I have decided to give up on a serverless approach, as the drawbacks of such implementation outweigh the advantages of reduced costs, that could happen with low application usage.

4.2.3 Elastic Container Services (ECS)

Amazon ECS allows for setting up container tasks and running them inside services. Tasks can be run using EC2 or AWS Fargate instances. The amount of vCPUs and memory allocated for each container can also be configured. Setting up an ECS cluster has no additional costs affiliated with it, other than the cost of the running EC2 or Fargate instances.

Scaling can be made possible via the usage of auto scaling groups and load balancers. Within auto scaling groups, we can define how many running instances we want to allow our services to scale up to. The triggering effects can also be set, such as average CPU utilization and average memory usage. If our task reaches that threshold, a new instance of the task is launched. The load balancers immediately register the running instances within the auto scaling group, so that would also make the implementation pretty straightforward. The load balancer can forward traffic to generator microservice instances based on their routing algorithm, which can either be set as a round-robin or least connections.

ECS also helps the maintenance of the infrastructure. AWS can collect many logs associated with the container tasks. Monitoring of the resource usages can also be done via AWS management console dashboards.

Due to the drawbacks of a serverless implementation listed in Serverless approaches, it would be beneficial to use EC2 instances for the running of our containers. That implicates that the implementation has to be done in a way, where our docker container (generator microservice) acts as a server.

4.2.4 Kubernetes

Kubernetes could be used as a tool for container orchestration, without the need of any provisioning tools. However, the project is mainly related to the creation of a generator microservice and removing the generation from the main backend. As a result, I feel like using Kubernetes or Elastic Kubernetes Service (EKS) for this task would be way too overkill, as only two containers would be running at the same time. For a task of this size, the YAGNI principle should be applied.

EKS wouldn't even allow us to become more independent from the AWS infrastructure (as if that was an issue, with only docker containers being hosted as tasks). However, if using another public cloud provider was an aspect, that we would need to consider, this would be a viable option. Kubernetes also allows for greater customizability of auto scaling properties.

4.2.5 Decision

The final deployment of our infrastructure will be done via the use of ECS and EC2 instances running the defined container tasks. The tasks should be running on the same auto scaling group of EC2 instances.

A load balancer will be added, so that the network traffic can be forwarded to the tasks running our docker containers. The cost of an application load balancer is \$0.02394 per Application Load Balancer-hour in the Stockholm region, where I am planning to deploy the application in the future. This application load balancer will balance the traffic towards basic backend target groups, and also towards generator server instances.

Another key factor, is that Amazon has cheaper prices for EC2 instances with arm cpu architecture. It would be beneficial to create a multi-architectural docker images, so that when we deploy our application, cheaper running costs can be achieved.

With this plan, using t3a.micro arm EC2 instances, the costs associated with the deployment should be as follows:

$$\text{Minimum cost per hour} = 2 \times \$0.0108 + \$0.02394 = \$0.04554$$

So approximately 4.6 cents per hour. This includes the usage of two EC2 instances and an Application Load Balancer.

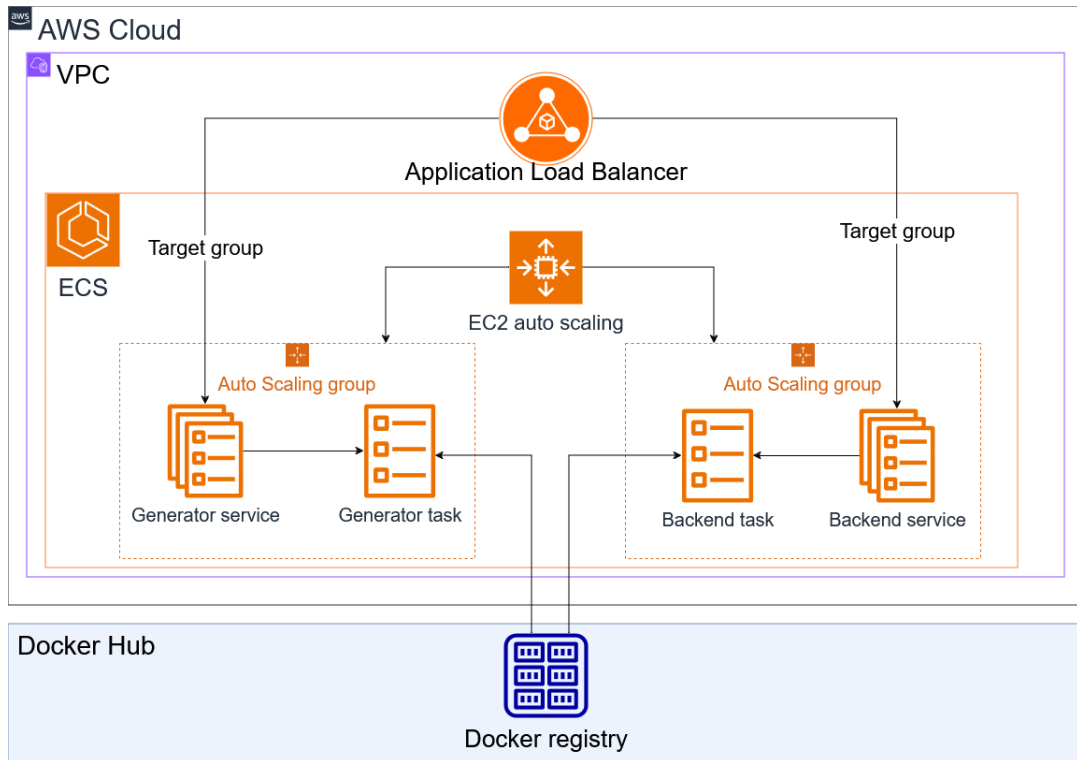


Figure 4.4: Final proposed infrastructure for deployment

Chapter 5

Implementation

This chapter delves into the implementation of the planned architecture defined in section 4.1.4. First I'll introduce the starting point of the project. Then I'll go into detail about how the backend creates a client, which communicates with the generator microservice. Finally I'll go into detail regarding the inner workings of the generator server.

The starting backend's class diagram can be found in the appendix at A.1.

5.1 Starting point

First, I will introduce the starting point of the project, so we get an overview, of how the backend initially worked regarding the model generations.

The `PushServiceDispatcher` is the class responsibly for calling the functions of the `ModelGenerationService` instance, and thus initiating the generation related methods.

5.1.1 `ModelGenerationService`

The *ModelGenerationService* is a core component of the application, designed to handle the initiation and management of model generation tasks on the backend server. This class encapsulates the logic for coordinating the execution of these tasks, while providing a seamless interface for both initiating and canceling model generation processes. It is a singleton class, so from a generator point of view, we might consider this class to be the real dispatcher.

The key features and responsibilities of the class include:

- **Service Initialization:** The *ModelGenerationService* is a singleton, ensuring a single, consistent instance throughout the application's lifecycle. It retrieves configuration parameters such as the model generation timeout from the environment, defaulting to 600 seconds if not explicitly set.

- **Model Generation Workflow:** The main responsibility of the service is to execute model generation requests by instantiating workers (*ModelGenerationWorker*), using multithreading. The workers perform the actual computational tasks:
 1. The `generateModel` method is provided the `PushWebDocumentAccess` instance, which holds the model described by the partial modeling language of Refinery.
 2. A `ModelGenerationWorker` is instantiated and configured with the document state (provided by the `PushWebDocumentAccess`), a random seed and a timeout value.
 3. The worker is started, as well as the resource heavy generation process. The worker is started on a background thread. The thread pool of for the generation workers can be configured via the `REFINERY_MODEL_GENERATION_THREAD_COUNT` [21] environment variable.
- **Cancellation Mechanism:** During this process listed above, the service interacts with the document's *ModelGenerationManager* to monitor the worker's activity. The timeout and cancellation is handled via the use of cancelation tokens.

The use of dependency injection for worker provisioning ensures that the service is decoupled from the specific implementation details of task execution, making it easier to extend or modify in the future.

5.1.2 ModelGenerationWorker

The *ModelGenerationWorker* class is the backbone of the backend's model generation process. It encapsulates the logic to handle the generation of computational models using a multithreaded approach, ensuring responsiveness and reliability during execution. Each instance of this class operates independently to process a specific model generation request.

The key features of the class include thread-safe execution, error management, scalable design via the use of the `ExecutorService` and the thread pool based multithreaded execution that it offers.

The main responsibilities of the class are:

- **Task Management:** Each worker is uniquely identified by a UUID, allowing precise tracking and management of tasks. The worker implements the `Runnable` interface, enabling it to be executed as a separate thread in a thread pool.
- **State Configuration:** The worker's state is set before execution via the `setState` method. This method configures the input document (`PushWebDocument`), random seed, and timeout duration required for the model generation process. The input text is extracted from the document and forms the basis of the model generation workflow.

- **Model Generation Execution:** The run method drives the core workflow. It first initializes a timeout mechanism and notifies the system that model generation has begun. The doRun method carries out the actual model generation, leveraging the following components:
 - **ProblemLoader:** Parses the input text into a formal problem representation.
 - **ModelGenerator:** Creates a generator instance to process the problem and produce a solution.
 - **MetadataCreator:** Extracts metadata (nodes and relations) from the generated model for further processing.
 - **PartialInterpretation2Json:** Converts intermediate results into a JSON format for downstream tasks.
- **Result Notification:** Once a model is successfully generated, or if an error occurs, the worker notifies the document’s precomputation listeners with the result. This ensures that other system components are informed of the status of the generation process.
- **Cancellation Handling:** The worker supports graceful cancellation through the cancel method. It uses a CancellationToken to periodically check whether the operation has been canceled and interrupts execution if necessary. A timeout mechanism automatically cancels the task if it exceeds the configured duration, ensuring that no worker consumes resources indefinitely.

The ModelGenerationWorker is the enabler of backend-driven computation. This implementation effectively delegates computationally intensive tasks, trying to minimize the burden on client communication.

5.2 Generator Client

Now that the deployed implementation has been described, we can restructure our application, so that the model generation is done via the generation microservice.

The design of the *ModelGenerationService* is scalable and modular: By delegating the actual computation to workers (ModelGenerationWorker), the service separates orchestration from execution, allowing for future modifications.

This comes in handy, when implementing our client for communication with the generator server. We have to implement a client, which communicates generation requests towards the generator server, and upon completion, receives such generation results.

The implemented client’s UML class diagram, alongside the modifications to the backend can be seen in appendix A.2.1.

5.2.1 GeneratorWebSocketEndpoint

The client-side component responsible for sending generation requests and handling responses is implemented as the `GeneratorWebSocketEndpoint` class, leveraging Jetty's WebSocket API. This endpoint facilitates asynchronous communication with the generator microservice, ensuring efficient request management over WebSocket connections.

In this section, I'll try to describe the key responsibilities and functionalities of the `GeneratorWebSocketEndpoint` class.

5.2.1.1 WebSocket Configuration

The server, which the client is communicating with, can be configured.

- The WebSocket URI is dynamically determined based on environment variables (`REFINERY_GENERATOR_WS_HOST` and `REFINERY_GENERATOR_WS_PORT`). If these are unset, default values (localhost and 1314) are used to ensure flexibility during deployment.
- The constructor initializes a `WebSocketClient` instance and sets default values, such as the worker's UUID and timeout duration.

5.2.1.2 Communication towards the server

- The client connects to the server via a `ClientUpgradeRequest`. The connection includes a custom header for the UUID of the worker.
- Sending the generation requests via `sendGenerationRequest` method. The method sends a structured JSON payload containing the request type, the UUID, the problem description, and a random seed. This starts the model generation process on the server.
- Sending generation cancel requests via the `sendCancelRequest` method. This client sends a JSON, with the UUID of the generation to be executed. Each generation opens a new session, with a new UUID for the server, so they can be uniquely identified.

5.2.1.3 Receiving from the server

Responses from the server are handled by the `onText` method, which processes various types of messages (result, error, `nodesMetadata`, `relationsMetadata`, `partialInterpretation`), parsing them into appropriate data structures and queuing them for consumption in their appropriate result queues.

5.2.1.4 Parsing responses from the queue

The client maintains several `LinkedBlockingQueue` instances to manage received data:

- `responseQueue`: Stores results or errors associated with the generation process.
- `nodesMetaDataQueue`: Holds metadata about nodes in the generated model.
- `relationsMetadataQueue`: Queues relation metadata details.
- `partialInterpretationQueue`: Handles intermediate model interpretation data.

The queues are accessed with timeout constraints to prevent indefinite blocking during retrieval. The timeout is set up by the class' timeout variable.

The `LinkedBlockingQueue` allowed the `onText` method, to put the received generation results into the queue, while the client is waiting for the results to arrive from the server. As soon as an element is put into the queue it can be popped from it. This way our code is safely waiting for the results to arrive from the server and no concurrency issues can arise. As I have already mentioned, each generation request instantiates a new client, so the size of these queues can only be between 0 and 1. No other generation client is accessing these queues of the instances.

5.2.1.5 Error Handling and Resource Management

- The `WebSocket` lifecycle events (`onOpen`, `onClose`, `onError`) are implemented to handle connection state changes gracefully.
- Proper resource cleanup is ensured through the `close` method, which terminates the `WebSocket` session and stops the client instance. In normal operation, the client is the one initiating the closing of the connection.
- Errors during critical operations, such as connection establishment or closure, are logged and propagated as exceptions.

5.2.1.6 Scalability and Extensibility

The `GeneratorWebSocketEndpoint` class is designed to support scalability.

- The `GeneratorWebSocketEndpoint` class is designed to support scalability.
- Its queuing mechanism ensures thread-safe handling of concurrent operations.
- The dynamic configuration allows seamless deployment in different environments without hardcoding server details.
- The modular structure facilitates extending the client to handle additional message types or features in the future.

This implementation forms the core communication layer between the client and the generator microservice, enabling efficient request handling, response parsing, and error management in a scalable and extensible manner.

5.2.2 ModelRemoteGenerationWorker

5.2.2.1 Core functionality

The ModelRemoteGenerationWorker manages remote model generation tasks by leveraging a WebSocket client (GeneratorWebSocketEndpoint). It sends model generation requests to a remote service, handles server responses, and retrieves metadata and interpretations required for completing the generation process. By offloading the generation task to a remote service, this class allows the backend to handle computationally expensive operations efficiently without overloading local resources. It preserves core mechanisms, such as timeout handling, task cancellation, and error reporting, ensuring a robust and reliable workflow.

5.2.2.2 IGenerationWorker

For implementation, the IGenerationWorker interface was created, which defines a unified API for all generation workers. This interface standardizes the core operations, including task initialization, execution, timeout management, and cancellation. This abstraction ensures that the backend can switch between local and remote workers without altering its core functionality.

The key methods from the interface include:

- `setState`: Configures the worker with the input document, random seed, and timeout settings.
- `start` and `startTimeout`: Enqueues the worker for execution and schedules a timeout to enforce task completion within a defined duration.
- `doRun`: Executes the remote model generation logic by communicating with the external service.
- `cancel`: Cancels the task, ensuring both local and remote operations are halted gracefully.

The interface is a reusable API, which both the old ModelGenerationWorker, and our newly created ModelRemoteGenerationWorker implement. The implementation of this interface grants us plug-and-play functionality. The usage of the new generator microservice is interchangeable with the old implementation. The backend can use either worker, based on the initial configuration, which can be set by an environment variable. The interface also allows for future workers to be implemented or a future hybrid functionality.

5.2.2.3 Workflow

The execution process of `ModelRemoteGenerationWorker` begins with setup and state configuration.

During task execution, the worker sends a generation request to the remote service by using a `WebSocket` client instance (`GeneratorWebSocketEndpoint`). The server's responses are processed incrementally.

The task completes successfully upon retrieving all necessary results, or it is terminated on errors, cancellation, or timeout.

Timeouts and cancellations are managed via a `ScheduledExecutorService`, ensuring that the system remains responsive and resources are freed promptly. In case of errors, the worker captures exceptions, logs the issues, and notifies the backend with appropriate error results.

The interaction between the `GeneratorWebSocketEndpoint` and the `ModelRemoteGenerationWorker` showcases how client-side operations and backend service orchestration are seamlessly integrated to ensure efficient, scalable, and reliable processing.

5.2.2.4 Impact

By integrating remote model generation, this implementation reduces the computational load on the backend server, making the system more scalable. The use of the `IGenerationWorker` interface allows a modular design, enabling seamless transition between local and remote workers or future extensions.

Furthermore, this approach demonstrates how backend systems can leverage external services without significant architectural changes, ensuring flexibility and scalability in modern distributed systems.

5.3 Generator Server

Now that the client side of the generation process has been introduced, the server side implementation is what should be described next. The main idea behind this implementation, was to take the experiences of the local model generation worker, and basically implement everything that is necessary for a generation to happen, in a multithreaded way, on our microservice. It is basically the implementation of the `ModelGenerationWorker` on a separate server.

The UML class diagram of the Generator Server can be found in the Appendix at A.3.1.

5.3.1 GeneratorServerEndpoint

. The GeneratorServerEndpoint is a WebSocket server endpoint that facilitates communication between remote clients and the backend for model generation requests. This class plays a crucial role in enabling real-time, bidirectional communication, supporting features like generation request handling, cancellation, and client disconnection.

5.3.1.1 Core Functionality

The two most important functionalities of the class are the handling of the generation requests and cancelation requests, which are received over the WebSocket sessions.

When a message of type "generationRequest" is received, the endpoint extracts generation details such as the unique identifier (uuid), random seed, and problem description. The data is sent by the client via JSON, so extracting the needed key-value pairs is pretty straightforward. These are then forwarded to the ModelGeneratorDispatcher, which manages the actual model generation task.

For "cancel" messages, the endpoint instructs the ModelGeneratorDispatcher to cancel an ongoing generation task associated with the provided uuid.

Any errors during the WebSocket communication are captured, logged.

When a WebSocket session is closed, the endpoint notifies the ModelGeneratorDispatcher to clean up resources associated with the disconnected client.

5.3.1.2 Integration into the workflow

The GeneratorServerEndpoint serves as the entry point for remote clients into the backend's model generation system. It relies on the ModelGeneratorDispatcher to manage generation and cancellation tasks. This provides a clean separation of concerns. The design ensures that the endpoint focuses solely on communication, while the dispatcher handles the generation tasks.

5.3.1.3 Scalability and Extensibility

This WebSocket-based implementation allows the backend to support multiple concurrent client connections efficiently. By using asynchronous communication, it ensures that tasks are queued and processed without blocking the ongoing generations. The modular design makes it easy to extend functionality, such as adding new message types or enhancing error handling mechanisms.

5.3.2 ModelGeneratorDispatcher

The ModelGeneratorDispatcher is a singleton class designed to manage and execute model generation requests on separate threads. Its primary role is to coordinate the model generation tasks, making sure that they run concurrently and their status updates and results are communicated back to clients over open WebSocket sessions. This dispatcher simplifies the handling of multiple requests and helps with maintaining system consistency.

5.3.2.1 Key features

The class implements the singleton design pattern to guarantee that only one instance of ModelGeneratorDispatcher exists throughout the application. This is achieved via the use of a private constructor which prevents instantiation of the class from the outside, ensuring that all interactions go through the singleton instance. This provides centralized control and avoids conflicting task management.

Each model generation request is executed in a separate thread to prevent blocking the main application flow. This is achieved through the ModelGeneratorExecutor class, which handles the task logic. The running model generations (ModelGenerationExecutor) are stored in a hashmap. The threads are identified by the UUID of the generation (key).

New generation requests are created, initialized and started on separate threads. They are then added to the previously mentioned hashmap.

Ongoing tasks can be cancelled via the UUID of the generation. As the hashmap stores these generation tasks identified by their UUID, it is fairly easy to do so.

Once a client has disconnected, the belonging task is cancelled, if not finished yet. The said task is removed from the hashmap. The UUID identification is useful here aswell.

The main methods of the class are:

- **getInstance:** Provides global access to the singleton instance of the dispatcher. Ensures thread-safe initialization using synchronized access.
- **addGenerationRequest:** First, dependencies are injected and a new ModelGeneratorExecutor is created for the request. Then the executor is initialized with task-specific details such as the random seed, problem string, and the WebSocket session with the client. Last, the executor thread is started and gets stored in the threadPool hashmap.
- **cancelGenerationRequest:** Retrieves the executor from the hashmap, based on the parameter UUID. Then the executor is signaled to cancel its operation.
- **disconnect:** Removes the executor from the threadPool and calls its disconnect method to release resources safely.

5.3.3 ModelGeneratorExecutor

The ModelGeneratorExecutor is a class responsible for performing the actual model generation tasks. It runs on a separate thread, making it suitable for concurrent processing. This class does pretty much what the original ModelGenerationWorker did. It even operates as part of a dispatcher system, executing generation tasks handed over by the ModelGeneratorDispatcher.

5.3.3.1 Key features and responsibilities

The main responsibilities of the class are the following:

- **Initialization:** Before starting, it initializes the problem by loading the problem description, random seed, and WebSocket session details. These were received by the GeneratorServerEndpoint over WebSocket, and are given to an executor object by the ModelGeneratorDispatcher.
- **Task Execution:** The class handles the execution of the model generation. This is done in the exact same way, as it was done previously at the local ModelGenerationWorker.
- **Client communication:** By getting an open WebSocket session, the class instance can send generation state results, metadata and errors to the clients.
- **Cancellation support:** The task can be interrupted or canceled at any point, ensuring that unnecessary processing is avoided. This is especially useful for timeout scenarios or user-initiated cancellations.

The key features of the class are:

- **Decoupled Execution:** Each ModelGeneratorExecutor instance works independently, processing a single request and reporting its status without interference from other tasks.
- **Feedback to the clients:** Task status updates are sent back to the dispatcher via the WebSocket session, ensuring real-time communication with clients.
- **Error Handling:** Any issues during validation or model generation are gracefully handled, with appropriate error messages sent to clients.

5.3.4 ServerLauncher

The ServerLauncher class serves as the entry point for starting a WebSocket server that processes model generation requests. It listens on a configurable port, defaulting to 1314 unless overridden by the *REFINERY_GENERATOR_WS_PORT* environment variable.

The class sets up a Jetty server, running the implemented `GeneratorServerEndpoint` (5.3.1). Once everything is configured, the server is started to handle incoming requests. This class essentially establishes the server infrastructure for the application.

5.4 Containerization

The new generator microservice was added to the "run" task of the backend gradle project, and as such, the workings of the application could be tested locally in a manual fashion.

Containerization (2.14) was the next step in updating our application.

Even before the restructuring, the build pipeline already existed. My main task was integrating the generator server docker image creation into these available scripts of the pipeline.

5.4.1 Preparation scripts

The process begins with a build script that compiles the project's distributed components, such as the backend and generator microservices, into portable archives. This is done via the use of Gradle (2.2).

One script organizes and optimizes the build outputs, ensuring shared dependencies are deduplicated and arranged for efficient reuse. It also prepares architecture-specific resources to support multi-architecture Docker builds, arranging all necessary files into a structured context for image creation. I had to modify this script, so that it also handles the same process for the newly created generator server's (5.3) archives.

Finally, a dedicated script builds and optionally pushes Docker images using a multi-stage build configuration. This configuration leverages tools like Docker Buildx and supports deployment across different architectures, ensuring compatibility.

5.4.2 Docker images

The project utilizes a structured and reusable approach to Docker (2.14.1) image creation, leveraging a common base image and architecture-specific optimizations to support multi-architecture compatibility.

A configuration file specifies the group of targets (like the base, backend and generator) and individual build configurations for each target. These targets are the created docker images.

These targets share a lightweight base image that includes essential runtime dependencies. Each target builds on this foundation, adding service-specific layers such as dependencies, binaries, and configuration files.

I implemented the `Dockerfile.generator` to containerize the generator service efficiently. It builds the service using a multi-stage process, layering runtime dependencies, application files, and configuration (like environment variables and exposed ports). It uses the base image as its starting point.

I also created a Docker Compose (2.14.2) file for the local deployment of the application. Via this, the manual testing of the created application docker images can be done. Before deploying to the production infrastructure, this can add an extra layer of ensuring proper workings.

The finished images were pushed to the Docker Hub registry. The public repository containing the images (backend, generator) can be accessible by anyone, and are needed for the AWS deployment.

5.5 Deploying on AWS

In this section I will describe the deployment of the modified Refinery backend and the Generator microservice. The deployment will be done according to the plan setup in 4.2.5.

I registered a new AWS account with my own private email address. For the creation of the infrastructure I used the AWS Management Console, which is the web GUI for managing our AWS infrastructure.

5.5.1 Task definitions

First I set my tasks up. The tasks are basically the container instances of an application within the AWS ECS environment. I created a task for the generator microservice called `refinery-generator` and the backend server called `refinery-language-web`. In the task configuration, I set the tasks to run on EC2 instances, provided the belonging docker registry link for the service, assigned the port mappings for the appropriate listening ports, and set the environment variables, such as the generator's host and port's. The `refinery-language-web` task definition needed the `"REFINERY_GENERATOR_WS_HOST"` to be equal to the load balancer of the generator microservice's load balancer (more on that later, in section 5.6.4). I set the networking mode for both tasks to be `awsvpc`, as they would be deployed on the same Virtual Private Cloud. Health checks also had to be implemented, so that our service knows, that the application is running and is in a stable state. For health checks, I used basic curl requests to the host and ports of the services. This caused some headaches in the generator microservice deployment (5.6.3).

5.5.2 Service definitions

I created a refinery cluster, with services running the defined tasks in 5.5.1. The services are within an EC2 auto scaling group. This allows us to scale our container instances, if

they are heavily used, and the resource usage is over a threshold. For this I set up two thresholds:

- CPU usage: If the average CPU usage is over 70% for more than 300 seconds, a new instance is deployed.
- Memory usage: If the average RAM usage is over 70% for more than 300 seconds, a new instance is deployed.

I used Amazon's t3.micro EC2 instances for the auto scaling group, which are eligible for the free tier accounts. They each have 2 vCPUs with 4 threads and 1GB of RAM.

5.5.3 Load Balancer definition

An application load balancer (ALB) was created for the backend and the generator server. The ALB would forward traffic to the running container instances within a target group, as long as their health checks pass. The ALB would be reachable for anyone on the internet on port 80. This port forwards traffic to the target group of the backend of the application. On port 81, the forwarding of the generator microservice requests is done towards the generator target group. However, this port is configured to accept requests only within the security group of the application. As such, only the backend can send requests to the microservice, hiding it away from the public internet. I set the load balancing algorithm to the default round-robin, but least-used can also be a viable option.

5.6 Challenges during the implementation

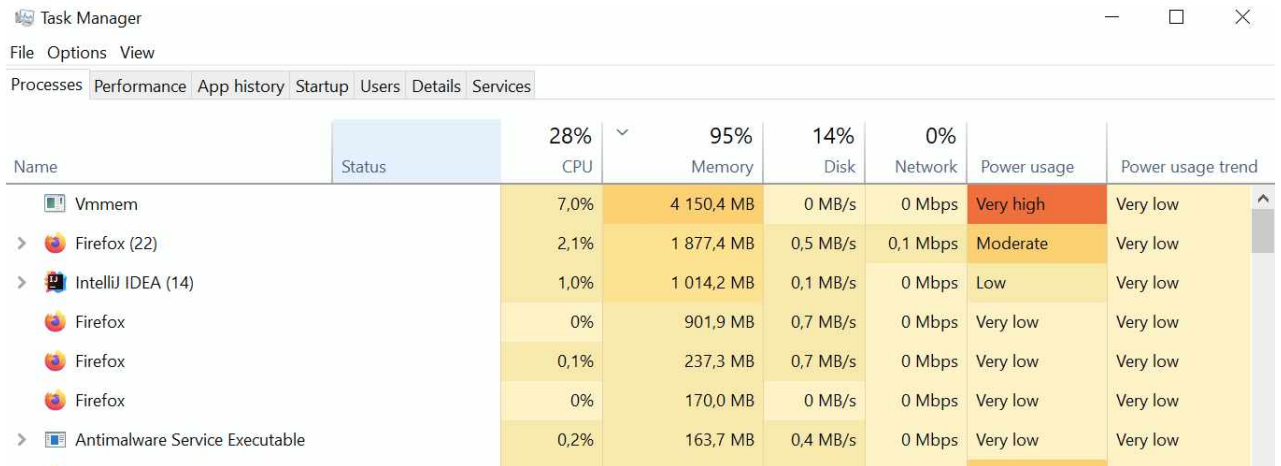
5.6.1 Resource usage of development

I wanted to develop on the Linux operating system, via the use of Windows Subsystem for Linux (WSL). WSL was needed because of the build scripts, which were implemented for UNIX shells.

When I initially opened the Refinery gradle project via a remote connection to my WSL the resource usage for the gradle project indexing was huge. The 8-core Ryzen 7 CPU was sitting at around 70% usage, alongside 16 gigabytes of my system memory being 92-99% occupied.

The initial indexing of the Gradle project took around 45-60 minutes. Even upon completion the resource usages didn't improve. The same amount of RAM was used by the WSL and it would not become lower.

As a result of this, I have given up on the development using WSL. I developed the application on Windows, while using WSL only for the dockerization (5.4) of the finished applications.



Name	Status	28% CPU	95% Memory	14% Disk	0% Network	Power usage	Power usage trend
Vmmem		7,0%	4 150,4 MB	0 MB/s	0 Mbps	Very high	Very low
> Firefox (22)		2,1%	1 877,4 MB	0,5 MB/s	0,1 Mbps	Moderate	Very low
> IntelliJ IDEA (14)		1,0%	1 014,2 MB	0,1 MB/s	0 Mbps	Low	Very low
Firefox		0%	901,9 MB	0,7 MB/s	0 Mbps	Very low	Very low
Firefox		0,1%	237,3 MB	0,7 MB/s	0 Mbps	Very low	Very low
Firefox		0%	170,0 MB	0 MB/s	0 Mbps	Very low	Very low
> Antimalware Service Executable		0,2%	163,7 MB	0,4 MB/s	0 Mbps	Very low	Very low

Figure 5.1: Resource usage during developing on WSL

5.6.2 Deduplication for docker images

The initial context preparation in section 5.4.1 was implemented for two different distributions (the basic backend named web, and a generator cli called cli). The common_libs creation for both of these distributed images was very straightforward as the two distributions only had limited amount of JARs that they both used.

With my project, a new distribution for the generator server was created. Originally I wanted to implement the deduplication in a way, where if two of the three distributions contained a JAR, the JAR would be put in the common_libs folder. However, this implementation didn't seem too useful. In this case all of the resources needed for the web distribution were added to the common folder. As such, the shell script would have been needed to be modified, so that it could handle empty dist folders during the pipeline. On the other hand the files of the common library would be put in the base docker image, even tho it might not be needed for the running of our application.

I decided to add only those resources to the common library, which are used in all of the created docker images. As a result, the base image is not bloated and the shell script didn't need many modifications.

5.6.3 Health check for the generator server

When deploying an application with the use of an Application Load Balancer, a target group of the tasks has to be specified. The load balancer needs to keep track of the running task docker container instances, so that it can decide, whether requests can be forwarded to the running microservice. A health check has to be put in use, so that the load balancer can register the running instances.

In my implementation of the generator server at section 5.3.1, I only created a WebSocket server. However, Jetty WebSocket servers won't respond to regular HTTP requests. With

this original implementation, the health check for the load balancer would not work: it always signaled that the task was not running, as it wouldn't respond to the curl http requests towards the default path (/).

As a workaround, I created a servlet called `HealthCheckServlet`, which responds to HTTP get requests at the path `/healthcheck`, by returning a "Server works!" plain text.

I added the servlet to the `ServerLauncher` of the generator microservice. By this, the load balancer can ensure the healthy state of the the server instances, by sending a curl request to the `/healthcheck` path of the generator server.

5.6.4 Application Load Balancer and NAT

Just when I thought that the load balancer issues were solved in 5.6.3, more issues arose. I modified my frontend, so that it would communicate with my own AWS deployed backend infrastructure. However, generation requests still would not work, with the requests resulting in a timeout, upon failing to connect with the generator microservice.

I initially thought this boils down to some kind of Access Control List (ACL) issue. I checked the ACL policies for the VPC, where all of my backend infrastructure was running. I allowed all inbound and outbound traffic in the VPC, just to insure that the VPC ACL policies aren't causing the issue. I also knew, that even though the VPC allows all traffic from the public internet, these policies can be overwritten with the usage of Security Groups, so that I can deny traffic from the public internet towards my generator microservice, and only allow inbound traffic coming from my backend server. To my surprise, this didn't resolve the issue either, the backend still couldn't connect with the generator server via the DNS of the ALB.

To insure, that it is not an ACL related issue, I set both the ALB's and the services' security groups to allow all incoming and outbound traffic. I was gifted with the same error messages on both the backend task logs and on my frontend: connection couldn't be made to the generator microservice.

Next I tried to check whether the EC2 instance, running the docker container (task) could reach the load balancer. I set up a key pair in the AWS Management Console, and assigned it to the auto scaling group of the EC2 instances. By forcing a redeploy of my backend service, I could finally SSH into the EC2 instances running the docker containers. By installing curl and ping on the instance I tried debugging network connectivity issues. The EC2 instance could send ICMP messages to the ALB, and it would respond. By sending curl messages to the health check path, that I created for the issue described in 5.6.3, the ALB would forward the traffic towards the generator server. The "Server works" plain text response was received by my EC2 instance running the backend.

Lastly, I wanted to check, whether the container running inside the EC2 instance can reach the generator server. The task was setup to run with the network mode `awsvpc`, so I assumed, that it should be able to reach the load balancer, as they are within the same

virtual private cloud. Since containers don't have anything installed, that is unnecessary for the running of the application, I copied a compiled static curl application, made for x86_64 compatible computers. Running this static curl, I sent a GET request to the ALB, requesting the "/healthcheck" path, however, no response was received.

Upon further research, I found out that ECS tasks (the containers) are not provided a public ip address by Amazon (only a private ipv4 address). The ALB is only granted a public address by AWS, and it cannot be given a private one. AWS doesn't perform Network Address Translation (NAT) for the endpoints within a Virtual Private Cloud, so I had to find a way, to perform the address translation, between the public and private addresses.

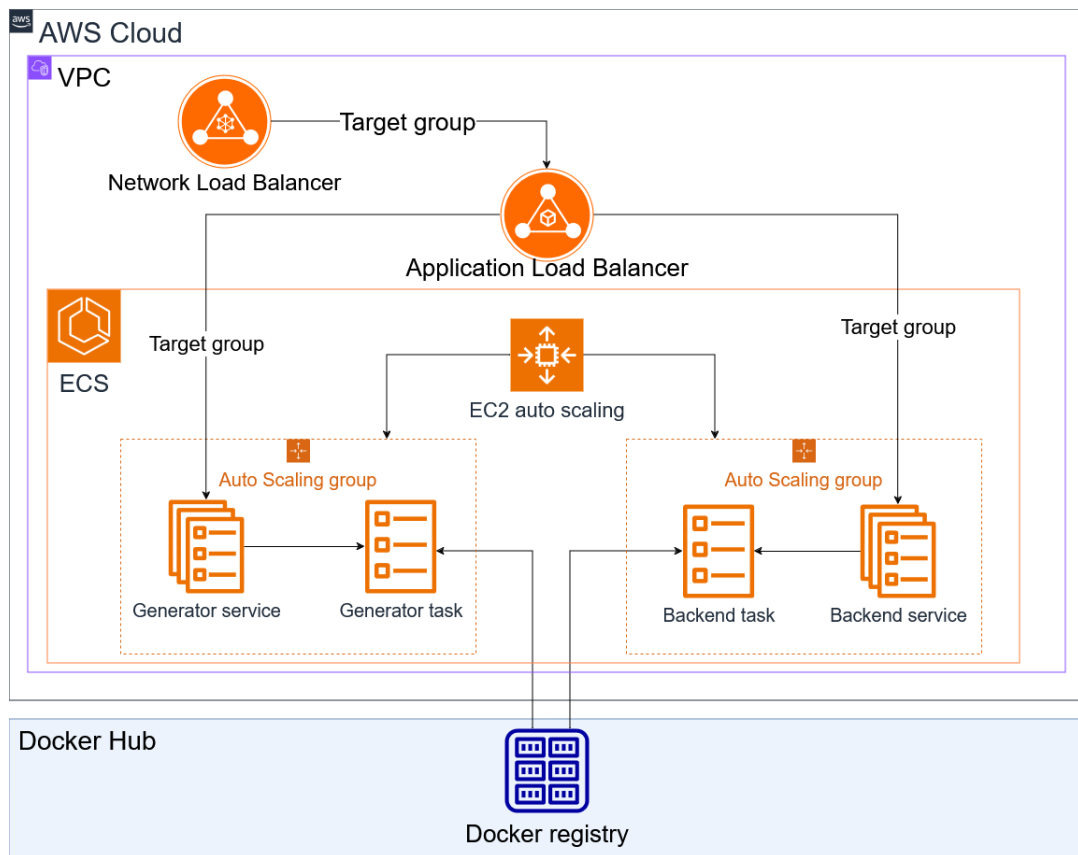


Figure 5.2: Final deployed infrastructure

My two choices were either the setup of a Network Load Balancer (NLB), or a Network Address Translation (NAT) gateway.

The NLB uses the same target group methodology the ALB used, however NLBs support the use of private ipv4 addresses. I can set the NLB's target group to be the Application Load Balancer with it's public address. This way the Network Load Balancer would act as a NAT gateway for the backend tasks.

The NAT gateway would allow us to connect the services in public and private subnets, and would provide future extensibility, if more features / tasks were to be added (even if some of them were not running in AWS).

I decided to go with the Network Load Balancer, as the hourly rate of a NAT gateway is more expensive, than the hourly rate of a Network Load Balancer [7] [8]. The hourly rate of a NAT gateway is \$0.046 dollars. The hourly rate of the NLB is calculated more complicatedly. However, based on the example calculations provided on the AWS website ([8]), with lower traffic the hourly rate would come down to nearly half of the price of the NAT gateway. I had to modify the task configuration of the backend: I modified the "REFINERY_GENERATOR_WS_HOST" environment variable to point to the DNS of the Network Load Balancer. The final implementation of the infrastructure can be seen in figure 5.2.

Chapter 6

Evaluation

6.1 Requirement analysis

In this section I'll go over the requirements that were made before the beginning of the project (3.3) and showcase how each of them are satisfied.

The backend of the application is scalable in several ways. The most obvious one, is how the application is deployed on AWS. If the resource usages hit a certain treshold, a new task is deployed. If the task cannot fit on an already running EC2 instance, the running EC2 instances scale up. This was desribed in section 5.5. The other way the application supports scalability is the way it handles multiple generation requests. Both the server (5.3) and the client (5.2) for the generator functionality support multithreaded operation. As a result, generation requests can run in parallel, and the resource utilization is improved.

The generation can be cancelled. The backend sends cancellation requests to the generator server (5.2.1.2). When the generator server receives a cancel request (5.3.1.1), it instructs the dispatcher (5.3.2.1) to cancel the generation on the executor (5.3.3.1).

The frontend is able to show the state of the model generation. The states are sent by the generator server's executor via the open session to the clients.

There is a timeout set for both the backend generator client and the generator server. This insures that even is some kind of unexpected operation happens, the generation won't be stalled and use resources in a locked state.

The editor and the generator can be deployed together. During development phase, the backend and the generator server can be deployed together, via the gradle "run" task. This starts running both the generator microservice and the backend locally. Other than that a docker compose file was written, so that the containerized applications can be run locally.

One optional requirement wasn't satisfied by this implementation. The creation of other type of clients other than the browser based one is possible, but tedious. My testing

python script basically acts as a different client, but it replicates the WebSocket messages of the frontend client, in order to create an editing session and send generation requests.

The price of the hosting has changed due to the issues described in 5.6.4. The price of the deployment for an hourly rate is expected to be:

$$\text{Minimum cost per hour} = 2 \times \$0.0108 + 2 \times \$0.02394 = \$0.06948$$

6.2 Benchmarks

6.2.1 Test setup

I wanted to test, whether the response times of the deployed application really improved with the modifications.

I wrote a Python script replicating the WebSocket messages of the Refinery frontend. First it creates the connection towards the server. Then sends the initial update of the example text, that can be seen in the editor at start. Then a generation request is sent to the backend and the start time of the generation is logged. When all of the generation results have arrived and none of them is an error, the end time of the generation is logged. The script is implemented in a way, where it can run in a multithreaded way. I can send out as many simultaneous generation requests from as many python clients as I'd want to. In the tests, I sent out 1, 25, 50, 75, 100, 125, 150, 175, 200, 225 generation requests at the exact same time, and analyze how the response times changed.

The response times were exported to a JSON file at the end of each test. I used python's matplotlib library to visualize the results of each test.

The running of the python test client could be done on an Amazon EC2 instance or on a personal computer. I have decided to run the test from my own personal computer, as this provides a more accurate representation of the real usage of the application.

6.2.1.1 External threats to validity

Since the infrastructure is deployed on Amazon's infrastructure, the availability of the service and the communication speeds between the two services depend on Amazon's network. As the results cannot be 100% accurate, repeated testing should be performed.

The network speeds are not constant and can be very volatile during the period of the test. The upload and download speeds of my computer, running the python test client can also make some test data dirty / invalid. Repeated testing should also solve this threat.

The load balancer for the backend allows accepts traffic from anything on the public internet. Although the chance of it is really low but denial of service attacks might invalidate the results of the test. I tried to lower the chance of this happening, by applying a security group to the load balancer for the duration of the test. With this, the load balancer

would only allow inbound traffic from my computer's public ip address. Obviously this address is not only assigned for my computer, but I think this significantly lowers the amount of connections, that might try to tinker with our server.

6.2.2 Results

6.2.2.1 First test

The first tests were performed with two t3.micro EC2 instances running at the start. These VMs have 2vCPUs and 1GB of memory. I allocated 2vCPUs and 0.9GBs of memory for each task. Since I was running two services, with the forementioned task configurations, those wouldn't be able to fit on a single EC2 instance. The auto scaling acted accordingly, and deployed the services on two separate EC2 instance.

The "REFINERY_MODEL_GENERATION_THREADS" environment variable wasn't set in the task configurations. As a result, the tasks ran with only 1 thread allocated for the model generation requests.

The results were as follows:

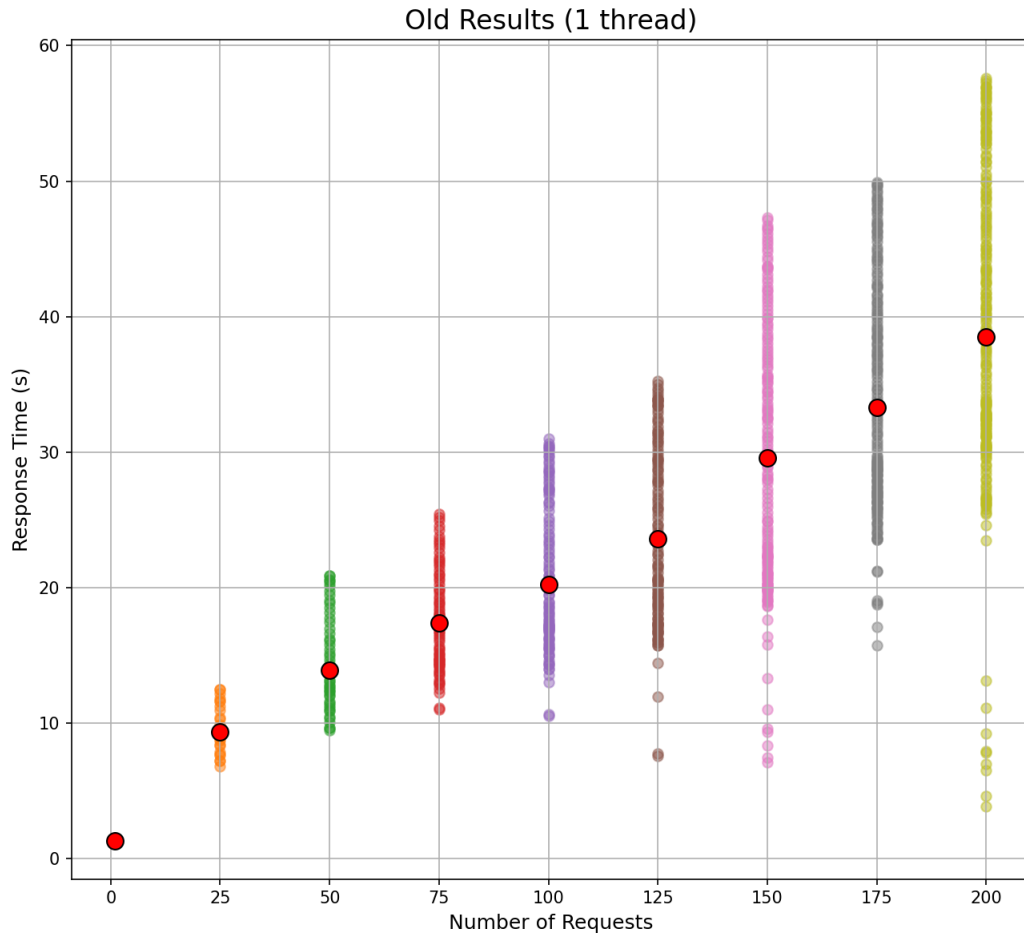


Figure 6.1: Generation requests with the old implementation

With only 1 thread being allocated the response times didn't improve. This comes down to the queueing mechanism of the dispatcher at the client side. Even though our generator server could handle multiple simultaneous generations, the client applies a queue for the sending out of the generation requests. That can also be seen in the results. The response times became bigger in a linear way.

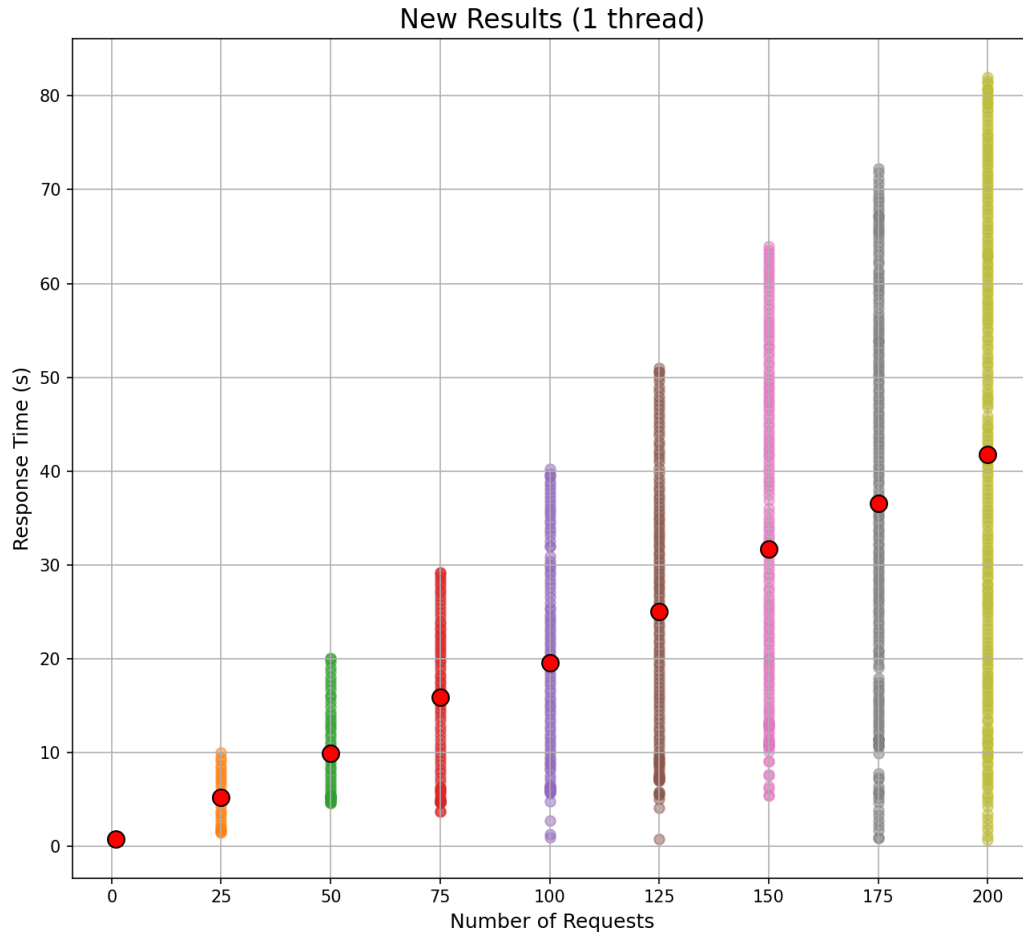


Figure 6.2: Generation requests with the generation running on the microservice

This is due to the nature in which the architecture is implemented. If the client in the backend only uses one thread for sending out the generation requests, the advantages of the multithreaded working cannot be utilized. If the multithreaded working is not used, the response times can only become worse, as there is an extra layer of networking communication added to the process.

6.2.2.2 Second test

These tests were performed with two t3.micro EC2 instances running at the start. These VMs have 2vCPUs and 1GB of memory. I allocated 2vCPUs and 0.9GBs of memory for each task.

The "REFINERY_MODEL_GENERATION_THREADS" environment variable, contrary to the test in 6.2.2.1, was set in the task configurations. I set the value of it to be 4. I chose 4, because the t3.micro EC2 instances each have 2 vCPUs with 4 computational threads.

The results were as follows:

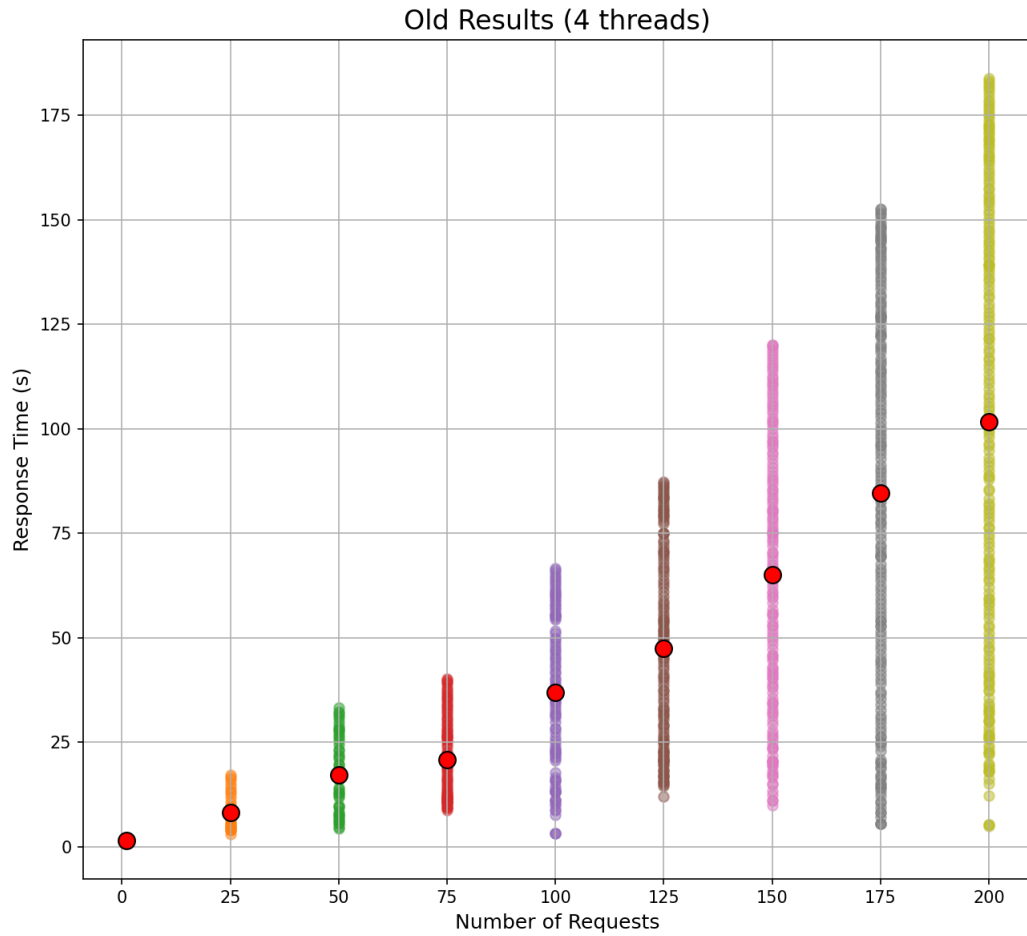


Figure 6.3: Generation requests with 4 threads, old implementation

The results in figure 6.3 indicate, that the model generation running in a true multi-threaded fashion slows down the old implementation. The deviation of the response times is much bigger, compared to the results of running on a single thread. Both the medians and the maximum of the response times grow in an exponential way.

In figure 6.4 a significant decrease can be seen in the response times. The times were better than the ones of the old implementations in figures 6.3 and the medians were also better

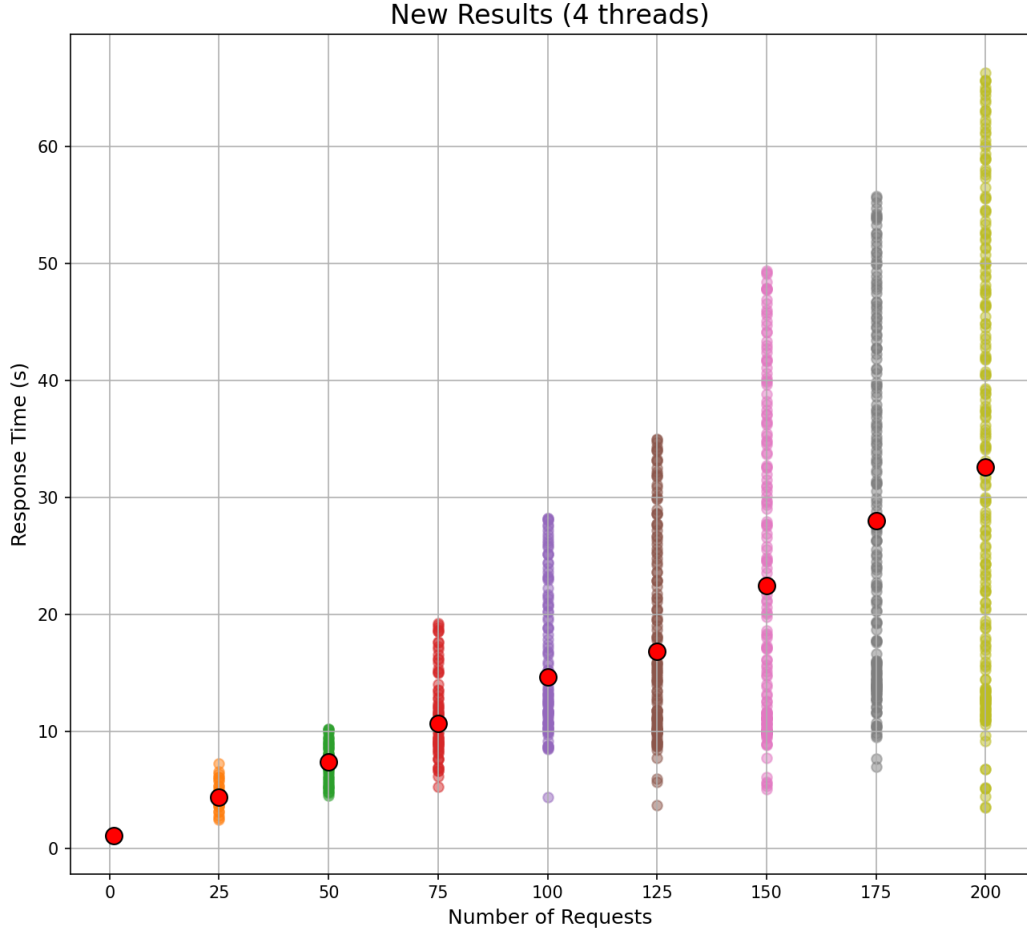


Figure 6.4: Generation requests with 4 threads and the microservice being used

compared to figure 6.1. This test provided the best response times for the application. This is due to the multithreaded generation actually being utilized.

6.2.2.3 Third test

These tests were performed with two t3.micro EC2 instances running at the start. These VMs have 2vCPUs and 1GB of memory. I allocated 1vCPUs and 0.4GBs of memory for each task. These task resource allocations were lowered compared to the allocations in 6.2.2.1 and in 6.2.2.2.

The response times of the test couldn't be acquired, because the application would not work as intended. I first adjusted the python test script, to handle timeouts aswell, because that seemed to be an issue upon initiated auto scaling. However, even with this modification, the application would not serve generation requests properly. This pointed out a bug in the implementation of the code.

In the backend's generation client, the PushServiceDispatcher is the only one using a real threadpool, with a limitation for the amount of threads that can be used. The imple-

mentation for the threadpool is not limiting maximum size at the generator server. This might cause the slowdown, which results in a timeout at the client side of the generation service. My python script also timed out in these cases. This happened for way too many requests, so it is something that should be investigated in the future.

However, the resource usages were very high with the described task resource allocations. The auto scaling of the service worked perfectly, with 5 backend and 8 generator instances running at some point during the test. After the end of the test, the services scaled down to only using 1 instance for each service.

6.2.2.4 Fourth test

These tests were performed with 6 t3.micro EC2 instances running at the start. These VMs have 2vCPUs and 1GB of memory. I allocated 1vCPUs and 0.4GBs of memory for the backend tasks. For the generator tasks, 2vCPUs and 0.9GBs of memory were allocated.

4 EC2 instances were running the backend tasks, while 2 EC2 instances had the generator microservice. These task resource allocations were lowered compared to the allocations in 6.2.2.1 and in 6.2.2.2.

The multithreading in the backend clients were setup to only use 2 generation threads.

The purpose of this test was to see whether the response times can be improved with more threads being allocated for the backend via the use of several running instances. As the backend uses a queue mechanism for the sendout of the generation requests, I hoped that this would significantly improve the response times.

The results were as follows in figure 6.5.

The only data being invalid is the first generation request. I forced my services to start with several instances, as I was trying to simulate a scaled up environment for my infrastructure. However, when the generator service receives its first request, it usually takes much longer to complete, due to the dispatcher not being initialized upon server start, only upon the first request.

The response times improved significantly compared to in figure 6.4. A linear growth can be noticed between the response times. The medians of this implementation are much lower. However, we have to consider, that with this implementation, the prices associated with hosting can be 3 times as high as the one implemented in 6.4. Thankfully, since we confirmed auto scaling to be working in 6.2.2.3, these response times can also be achieved, without the constant use of 6 EC2 instances.

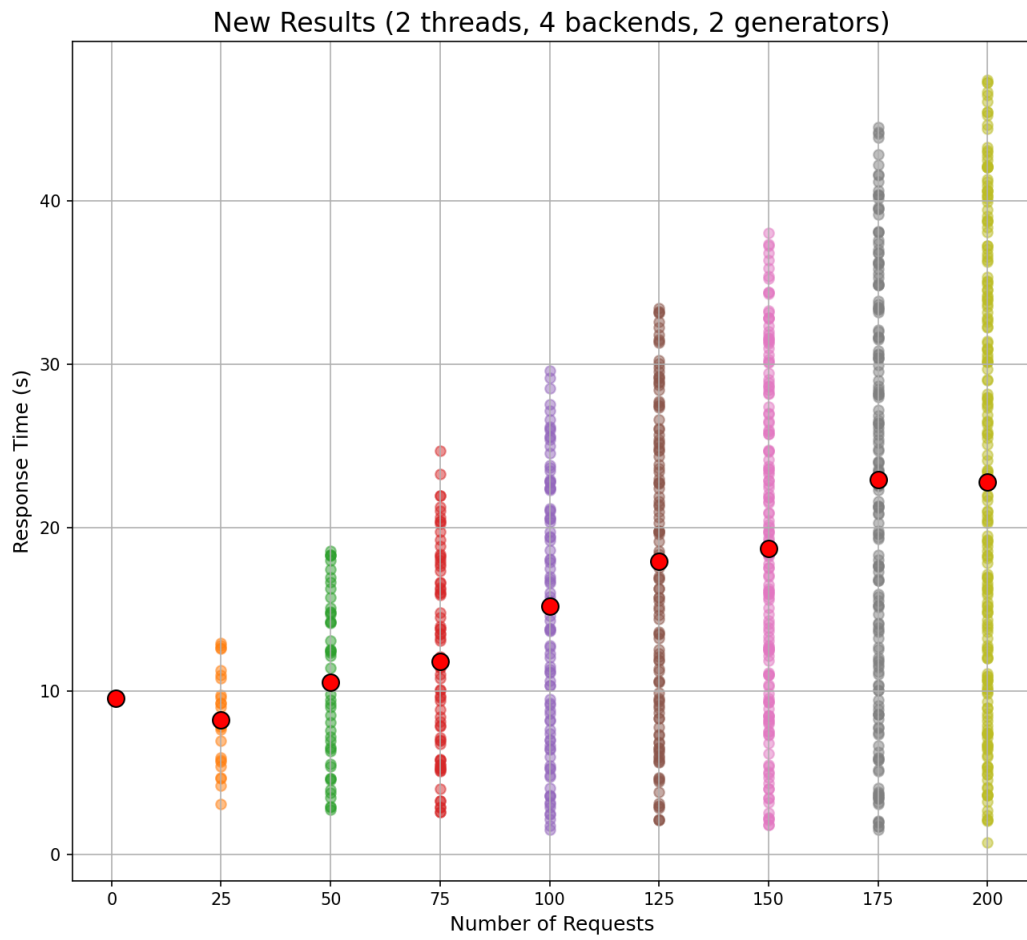


Figure 6.5: Generation requests with 2 threads, 4 backend instances and 2 generator instances

Chapter 7

Conclusion

The purpose of the thesis was to design, implement the architecture and deploy the Refinery graph solver's backend in the AWS cloud environment. The emphasis was on the model generation part of the already existing architecture.

In Chapter 3. I described the state of the application's architecture before my work. This included both the frontend and the backend, with the emphasis being on the backend. The requirements were set so the work could be started on the architecture.

In Chapter 4. I introduced the potential ways the architecture could be modified while describing the advantages and drawbacks of each implementation.

In Chapter 5. I described what changes were made to the codebase and the build scripts. Then I wrote about the dockerization of the application. Lastly, I deployed the application on AWS, with careful considerations to price and scalability.

In Chapter 6. I evaluated how the changes satisfy the requirements that were made in Chapter 3 section 3.3. I benchmarked my application: visualized the results and analyzed them. We could see, that under heavy usage, the response times could be improved, however it comes with bigger deployment costs / AWS fees, and under lower usage, the improvements are diminishing. As a result, I conclude that it is not worth it to constantly run the application with the generator microservice constantly running and being utilized, as the hourly rates are doubled.

7.1 Future work

The current implementation of the generator microservice doesn't handle threadpools correctly at the server side, as seen in 6.2.2.3. There is no limitation set for the maximum threads that can be started. This caused an issue where the microservice would slow down drastically, and generation responses would not be received at the client side. The fix for this should include the usage of Java's `ExecutorService` for the threadpools.

The response times benchmarked in 6.2 show an improvement compared to the old implementation. However, this improvement is only noticable during increased loads. However, when the usage is not as demanding, the local model generation on the basic backend is fast enough, with the new implementation being more expensive. A hybrid way of working should be implemented, where the model generation microservice is only deployed, when absolutely needed to.

The scalability methods are very limited because of AWS. The main scalability factors are the resource utilizations of the CPU and the RAM of each service. This could be improved in the future, where more factors are considered, and the scaling-up and scaling-down of the infrastructure is handled differently.

Bibliography

- [1] Introducing json. <https://www.json.org/json-en.html>, 2024.
- [2] AWS. What is application auto scaling? <https://docs.aws.amazon.com/autoscaling/application/userguide/what-is-application-auto-scaling.html>, 2024.
- [3] AWS. What is aws? <https://aws.amazon.com/what-is-aws/>, 2024.
- [4] AWS. What is amazon ec2? <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html>, 2024.
- [5] AWS. What is amazon elastic container service? <https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html>, 2024.
- [6] AWS. What is application load balancer? <https://docs.aws.amazon.com/elasticloadbalancing/latest/application/introduction.html>, 2024.
- [7] AWS. Amazon vpc pricing. <https://aws.amazon.com/vpc/pricing/>, 2024.
- [8] AWS. Elastic load balancing pricing. <https://aws.amazon.com/elasticloadbalancing/pricing/>, 2024.
- [9] AWS. What is amazon vpc? <https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html>, 2024.
- [10] Cloudflare. What is load balancing? <https://www.cloudflare.com/learning/performance/what-is-load-balancing/>, 2024.
- [11] Docker. What is docker? <https://docs.docker.com/get-started/docker-overview/>, 2024.
- [12] Docker. Docker compose. <https://docs.docker.com/compose/>, 2024.
- [13] Eclipse. The eclipse jetty project. <https://jetty.org/index.html>, 2024.
- [14] Google. What is cloud computing? <https://cloud.google.com/learn/what-is-cloud-computing>, 2024.
- [15] gRPC. Introduction to grpc. <https://grpc.io/docs/what-is-grpc/introduction/>, 2024.

- [16] gRPC. Core concepts, architecture and lifecycle. <https://grpc.io/docs/what-is-grpc/core-concepts/#bidirectional-streaming-rpc>, 2024.
- [17] IBM. What is a rest api? <https://www.ibm.com/topics/rest-apis>, 2024.
- [18] IETF. Rfc 6455: The websocket protocol. <https://datatracker.ietf.org/doc/html/rfc6455>, 2011.
- [19] mdn. An overview of http. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview>, 2024.
- [20] RedHat. What is containerization? <https://www.redhat.com/en/topics/cloud-native-apps/what-is-containerization>, 2024.
- [21] Refinery. Threading. <https://refinery.tools/learn/docker/#threading>, 2024.
- [22] Refinery. Introduction. <https://refinery.tools/learn/>, 2024.
- [23] Amazon Web Services. What is cloud native? <https://aws.amazon.com/what-is/cloud-native/>, 2024.
- [24] VMWare. What is cloud scalability? <https://www.vmware.com/topics/cloud-scalability>, 2024.
- [25] Wikipedia. Gradle. <https://en.wikipedia.org/wiki/Gradle>, 2024.
- [26] Wikipedia. grpc. <https://en.wikipedia.org/wiki/GRPC>, 2024.
- [27] Wikipedia. Java (programming language). [https://en.wikipedia.org/wiki/Java_\(programming_language\)](https://en.wikipedia.org/wiki/Java_(programming_language)), 2024.
- [28] Wikipedia. Remote procedure call. https://en.wikipedia.org/wiki/Remote_procedure_call, 2024.
- [29] Wikipedia. Shell (computing). [https://en.wikipedia.org/wiki/Shell_\(computing\)](https://en.wikipedia.org/wiki/Shell_(computing)), 2024.
- [30] Wikipedia. Shell script. https://en.wikipedia.org/wiki/Shell_script, 2024.

Appendix

A.1 Starting Refinery backend

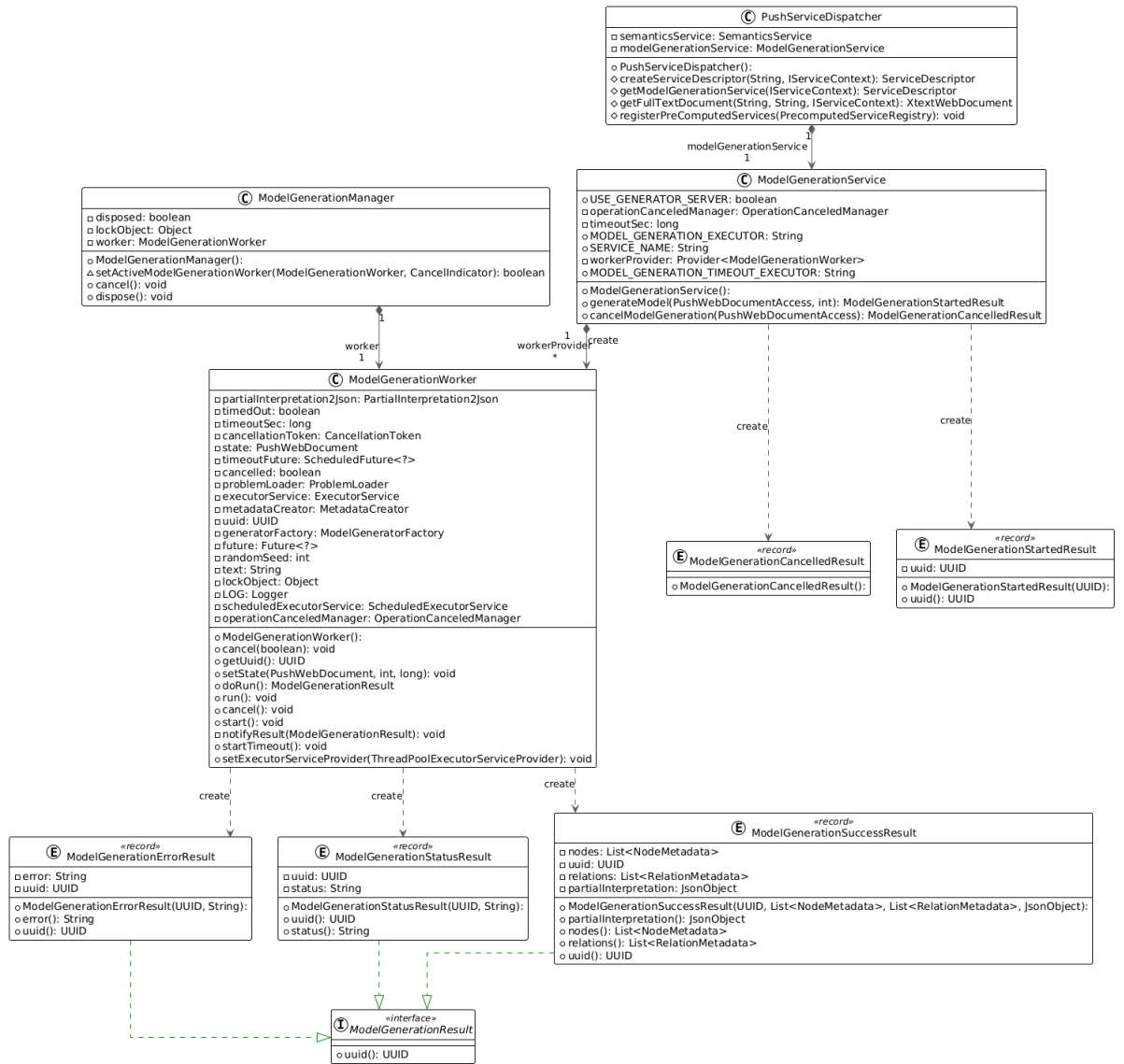


Figure A.1.1: UML class diagram of the backend architecture

A.2 Generation Client

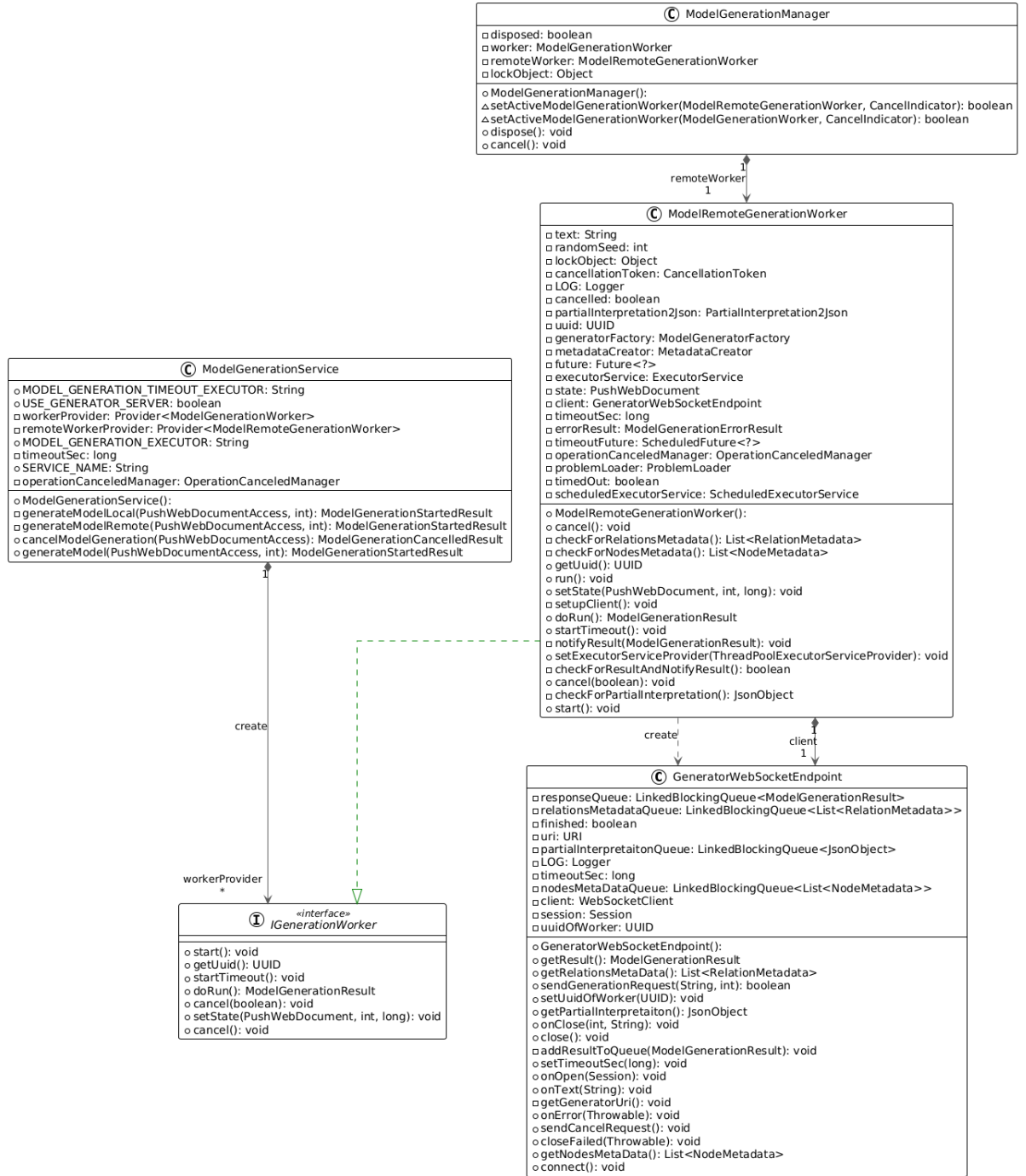


Figure A.2.1: UML class diagram of the generation client part of the backend

A.3 Generator Server

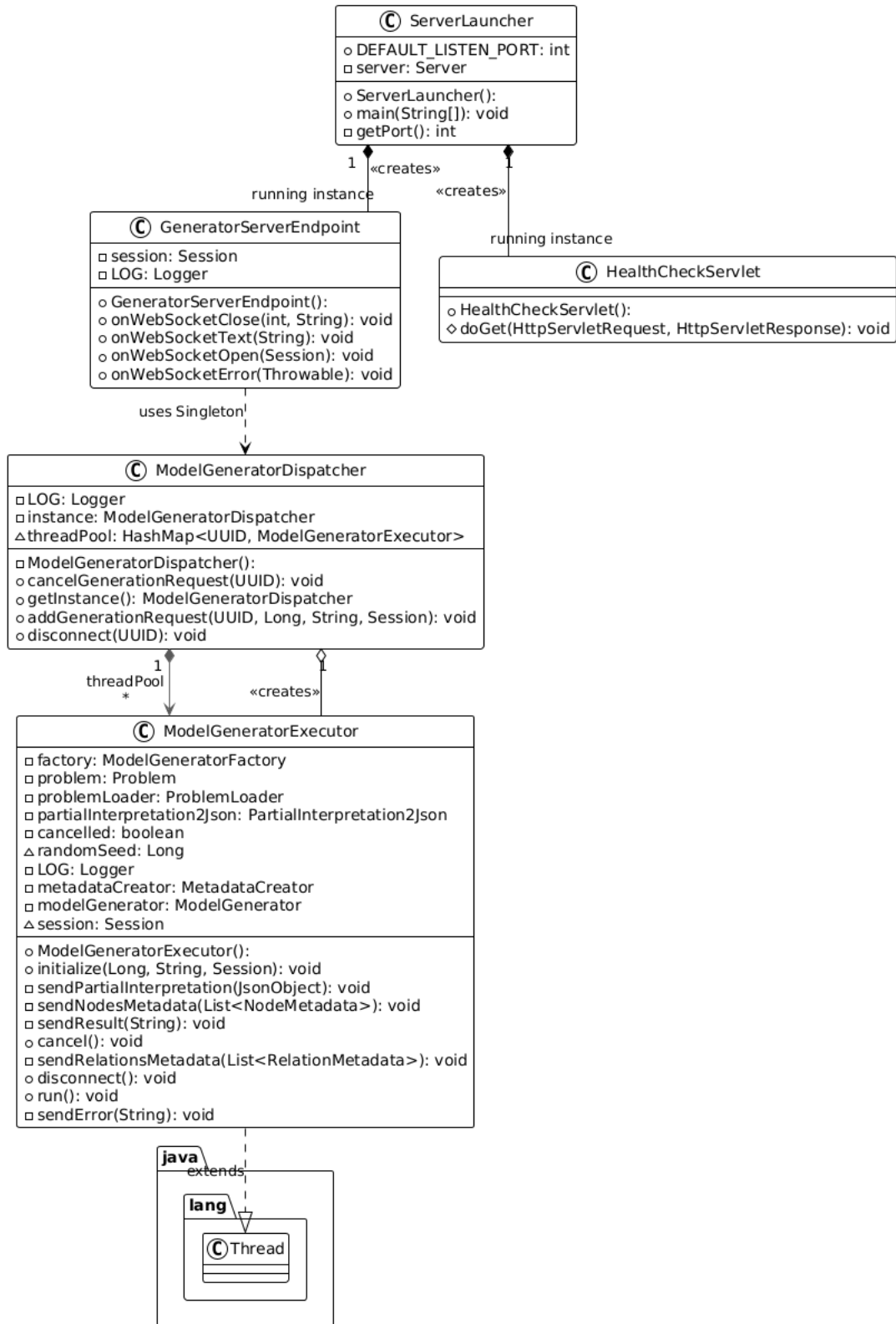


Figure A.3.1: UML class diagram of the generator server