Capstone Project Specification - VueCalc Cloud Project

Diogo Pereira [110996], João Gonçalo Santos [110947], João Matos [110846]

(Group 9)

Note: If any details are missing or unclear in this report, please refer to the README file in the project repository for further technical explanations and configurations.

1 Introduction

This project implements a multi-tiered calculator application that demonstrates the integration of modern microservices architecture, load balancing, persistent storage, and system monitoring. The system is designed to handle basic arithmetic operations—addition, subtraction, multiplication, division, and a custom-built modulo feature. These operations are handled by two distinct microservices, each written in a different programming language, showcasing the flexibility and interoperability of the microservices architecture. Additionally, the project integrates a persistent historical feature using another microservice, which enables users to retrieve past calculation operations from a *MongoDB* database.

The objective is to showcase a realistic example of how to deploy and provision a tiered, microservices-based, containerized web application on a public cloud provider. This solution includes the use of *Terraform* for infrastructure automation and *Ansible* for application deployment across distributed cloud resources. The system also incorporates HAProxy to load balance requests across multiple instances of all microservices, ensuring efficient distribution of workloads. Moreover, the project introduces Prometheus for monitoring the health and performance of the microservices by tracking key metrics, such as request latency, total request counts, error rates, etc. for each service, providing visibility into system performance.

This report outlines the architecture, tools, and methodologies used to deploy and manage the system, alongside an evaluation of its performance under varying loads.

The link to the project demonstration video can be found here: https://youtu.be/Pa9TOSa7dz0

2 System Design and Architecture

The system follows a multi-tier architecture, carefully separating responsibilities between the frontend, backend microservices, persistent storage components, and monitoring tools. This separation ensures flexibility, scalability, and maintainability throughout the system.

2.1 Basic System Components

The core of the system is a calculator application that operates based on a microservices architecture, where different operations are handled by distinct services. The frontend acts as the user interface, and the backend microservices (Expressed and Happy) handle the core calculation logic. These components are independent, ensuring modularity and scalability.

2.1.1 VueCalc Frontend

The frontend is built with Vue.js and serves as the user interface where users input numbers and select arithmetic operations. When a user submits a request (e.g., addition, multiplication), the frontend generates an HTTP request. Although part of the *Advanced Components* section, the load balancer is responsible for directing this request to the appropriate backend microservice, either Expressed or Happy, depending on the operation. The frontend then receives the response (the calculated result) from the microservice and displays it to the user.

2.1.2 Expressed and Happy Microservices

The backend comprises two core services, Expressed and Happy, both written in different frameworks to showcase architectural flexibility:

- Expressed Microservice: This microservice is built using *Node.js* with *Express* and is responsible for handling addition and subtraction operations. It exposes two API endpoints*: /add* and */subtract*, which take two numerical inputs and return the computed result.

- Happy Microservice: Developed using *Hapi.js*, the Happy service manages different arithmetic functions, such as multiplication, division, and the custom modulo feature, which was added specifically for this project. It provides three endpoints: */multiply*, */divide*, and */modulos*, each performing the corresponding operation.

Each of these microservices is replicated into two instances, allowing for load balancing and high availability. This replication ensures that incoming requests are evenly distributed, improving the system's ability to handle increased traffic.

This brings us to the *Advanced Components* section, where we detail the load balancer, persistent storage mechanisms, and monitoring tools that enhance the system's functionality.

2.2 Advanced Components

The system's robustness and scalability are enhanced by several advanced components, including the integration of a load balancer for efficient traffic distribution, a persistent storage solution for saving calculation results, and Prometheus for real-time monitoring and performance metrics.

2.2.1 Load Balancer (HAProxy)

The system employs *HAProxy* as a load balancer to distribute traffic across the replicated microservice instances. The load balancer is positioned between the frontend (VueCalc) and the backend services (Expressed, Happy, and Bootstorage). The primary responsibility of HAProxy is to distribute traffic evenly across the replicated microservice instances, ensuring high availability and fault tolerance.

HAProxy is configured to use the round-robin load balancing algorithm, which sequentially forwards each incoming request to a different microservice instance. This ensures that all instances share the computational load equally, preventing any single instance from being overwhelmed. For instance, when the last available server receives a request, the algorithm distributes it from the top of the server list all over again.

HAProxy is also configured to use ACL-based routing to direct traffic based on the type of operation requested, ensuring that all traffic passes through the load balancer:

Frontend Requests (from VueCalc):

- Requests for addition and subtraction are forwarded to the Expressed microservice.
- Requests for multiplication, division, and the custom modulo operation are forwarded to the Happy microservice.
- Requests regarding the history feature (which shows the results of historical calculations) are forwarded to the Bootstorage microservice (more information about Bootstorage service in 2.2.2).

Backend to Bootstorage:

- Requests from Expressed and Happy microservices to Bootstorage for storing operation results are also balanced across the existent Bootstorage instances**.**

In addition to distributing traffic, it conducts health checks on the backend instances. Thus, if an instance becomes unresponsive or unavailable, the load balancer automatically reroutes traffic to the remaining operational instances, minimizing downtime.

Note: The HAProxy dashboard, accessible via Load Balancer VM's IP and a dedicated port, allows real-time monitoring of the system's load distribution and the health of each backend instance.

2.2.2 Persistent Storage (Bootstorage and MongoDB)

To support persistent storage of operation history (to save and track the results of all operations performed), the system includes the Bootstorage microservice, which is built using *Spring Boot*. The Bootstorage service interacts directly with the *MongoDB* database, which is self-hosted on a separate virtual machine.

This microservice is responsible for receiving and storing each operation performed by Expressed and Happy microservices. It provides an API endpoint (*/create*) that the backend services use to log each operation along with the result. MongoDB, as the persistent data store, holds the history of all user operations, allowing users to retrieve past calculations.

2.2.3 Real-Time Monitoring (Prometheus)

To ensure comprehensive monitoring and observability of the system, Prometheus is integrated for real-time performance metrics and health monitoring. Prometheus is configured to scrape metrics from all microservices—Expressed, Happy, and Bootstorage—at regular intervals. Expressed and Happy expose metrics via a dedicated `/metrics` endpoint, while Bootstorage uses the `/actuator/prometheus` endpoint.

In addition to the default metrics from Prometheus (such as memory usage and uptime), service-specific metrics include:

- Average latency: Monitoring how long each request takes to complete.
- Total request count: Counting the total number of HTTP requests handled by each service.
- Error rates: Tracking the number of failed requests across services.

Prometheus scrapes these metrics every 15 seconds, ensuring timely insights into the system's performance. The Prometheus dashboard, accessible via Prometheus VM's IP and a dedicated port, allows real-time monitoring of all services.

2.3 System Architecture Diagram

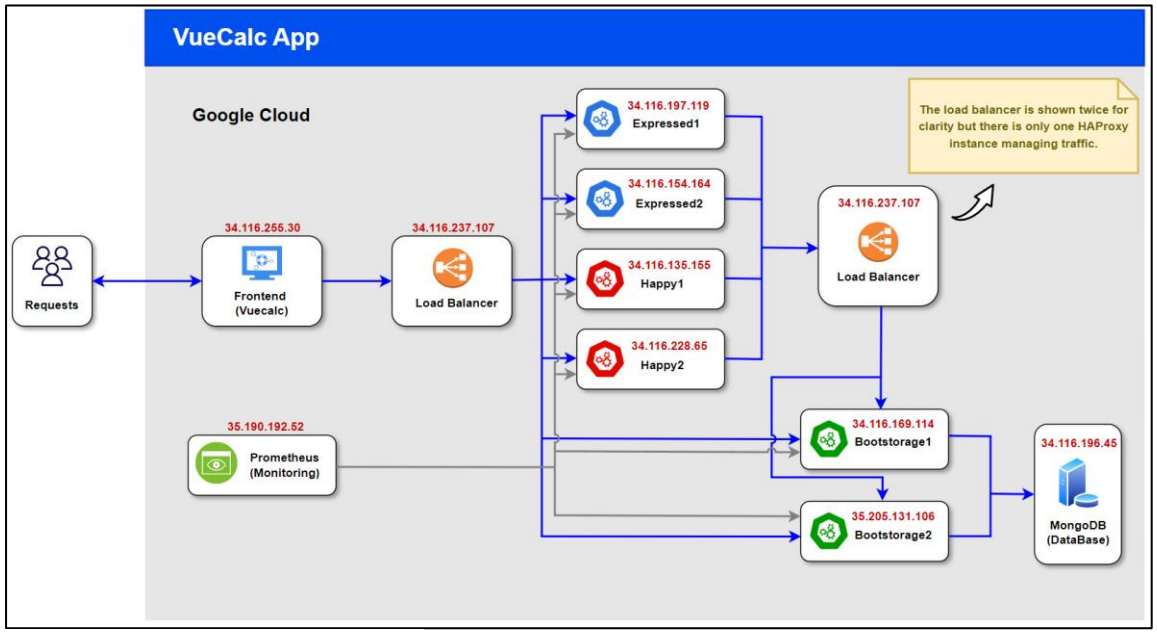

Fig. 1 *- Architecture of the solution*

3 Deployment

The deployment of the VueCalc Cloud Project follows a systematic approach using Vagrant, Terraform, and Ansible to automate and manage infrastructure and application setup on Google Cloud Platform (GCP).

3.1 Vagrant: Management VM

Vagrant is used to provision a local Management VM (*mgmt*), which serves as the control hub for the entire deployment process. This VM is responsible for executing Terraform and Ansible commands, ensuring a clean and consistent environment for infrastructure management.

A Vagrant file is used to manage a local VM, referred to as the Management VM (*mgmt*), which serves as the control node for the deployment. The Management VM is provisioned with the necessary tools (Terraform, Ansible, and SSH keys) to manage the deployment process across GCP. The following steps are performed within the Management VM:

- Vagrant creates the Management VM on the local machine, which is used to manage the entire deployment.
- SSH keys are generated within the Management VM, which are then distributed to the GCP VMs to allow secure access for configuration.
- Once created, the Management VM is used to run all subsequent deployment commands.

3.2 Terraform: Infrastructure Provisioning

The infrastructure for the project is deployed on Google Cloud Platform (GCP) using Terraform. Terraform automates the creation and management of cloud resources, such as virtual machines (VMs), networks, and firewall rules. It defines all components in code and manages their lifecycle, ensuring they are correctly created, configured, and accessible for further configuration steps. By running `terraform apply`, the infrastructure (as shown in **Fig. 1** - Architecture of the solution) is created and configured on GCP.

Note: External IP addresses for each of the virtual machines were reserved manually via the GCP Console to ensure they remain static, since we wanted to guarantee consistent access for the load balancer, frontend, and backend services, preventing IP changes when VMs reboot.

3.3 Ansible: Configuration and Deployment

Once the infrastructure is provisioned, Ansible is used to configure the VMs and deploy the microservices. By using playbooks, it ensures consistent and repeatable setups across all virtual machines. Ansible playbooks automate the following tasks:

- Ansible installs necessary packages, such as Docker on each VM to ensure consistent containerized deployment of the microservices.
- Each backend service (Expressed, Happy, and Bootstorage) is deployed as a Docker container. Ansible pulls the application code from the repository, builds Docker images, and starts the containers on the appropriate VMs.
- Ansible configures HAProxy on the load balancer VM, setting up round-robin load balancing across the replicated instances of Expressed, Happy, and Bootstorage microservices.
- On MongoDB VM, Ansible installs and configures MongoDB to store the results of the operations performed by the calculator. The Bootstorage service interacts with this database to persist user operation history.
- Ansible also configures Prometheus to monitor all microservices (Expressed, Happy, Bootstorage), enabling real-time performance monitoring by scraping metrics at regular intervals.

4 Evaluation

This section presents an evaluation of the system's performance through a study of its throughput (requests per second), average latency (and also maximum latency for us to be aware of the worst-case response time), and error rates as the number of concurrent clients increases. The study was conducted using the K6 performance testing tool. The load test file is stored in the *k6_grafana folder* (run command: `k6 run load_test.js`).

To accurately assess the system's behavior under different loads, we performed load tests targeting the microservices of the system: Expressed, Happy, and Bootstorage. Each test followed a controlled ramp-up period of 10 seconds to reach a specified number of users (clients), and then the load was maintained at that level for a specified period (1 minute) to collect stable data. The user levels tested were 200, 500, 1000, and 2500 clients. After each test, the underlying MongoDB database was cleaned to ensure that data from previous tests did not affect the results.

	Expressed			Happy			Bootstorage (for ~5000 entries in MongoDB) *		
Clients	Throughput (req/s)	Avg Latency (ms)	Max Latency (ms)	Throughput (req/s)	Avg Latency (ms)	Max Latency (ms)	Throughput (req/s)	Avg Latency (ms)	Max Latency (ms)
200	173,05	66,09	445,09	171,18	75,04	1380	168,75	82,45	676,3
500	431,79	66,27	379,51	427,3	77,31	1320	425,03	82,72	456,28
1000	852,63	78,62	2670	864,5	65,86	729,36	850,97	81,55	582,65
2500	1376,73	650,32	3250	2100,1	95,51	893,58	2048,8	122,18	1800

Error Rate = 0% for all microservices *

*However, as the data volume grew significantly (reaching 103,423 entries), the system encountered an error: "context deadline exceeded," indicating that the query or processing time exceeded the allowed threshold due to the size of the dataset.

Table 1 *- Results from load test*

Key Findings:

1. **Expressed Service (also uses Bootstorage Service to create an operation in MongoDB):**
    - Throughput: The Expressed microservice shows consistent scalability. At 200 clients, the throughput was measured at 173.05 req/s, which increased significantly to 1376.73 req/s at 2500 clients. This indicates that Expressed efficiently handles higher loads without degradation in throughput.
    - Latency: The latency remains fairly low at around 66ms under 500 clients, but it increases noticeably to 650.32ms when the client count reaches 2500, suggesting a bottleneck at higher loads. At 2500 clients, the maximum latency peaks at 3250ms, which is expected due to the additional computational and response time required to manage a larger client base.

2. **Happy Service (also uses Bootstorage Service to create an operation in MongoDB):**
    - Throughput: The Happy microservice also scales well underload. At 2500 clients, the throughput reaches 2100.1 req/s, showcasing its capacity to handle higher client demands.
    - Latency: Happy maintains a stable average latency even as the client load increases. From 75.04ms at 200 clients, it rises only to 95.51ms at 2500 clients, showing that the service is optimized to handle concurrent operations efficiently. Although Happy's maximum latency also increases with more clients, it is better managed compared to Expressed, with a peak of 893.58ms at 2500 clients.
3. **Bootstorage Service — History Feature (with ~5000 entries in MongoDB):**
    - Throughput: Bootstorage, responsible for tracking historical operation data, shows strong performance with 2048.8 req/s at 2500 clients, closely matching the throughput of the other microservices.
    - Latency: The average latency remains stable across different client loads, with values ranging from 82ms to 122ms at maximum load. Bootstorage experiences a significant jump in maximum latency when handling 2500 clients, reaching 1800ms. However, when the dataset exceeded 103,423 entries (major test), the system encountered a processing time error due to the dataset's size.