Technical Report - Runtimes for Concurrency and Distribution

FRANCESCO BACCHIN, Università degli studi di Padova, Italy
ALESSANDRO CAVALIERE, Università degli studi di Padova, Italy

This report investigates the design, development and evaluation of a microservices-based system for image tagging. Emphasizing scalability, autonomy and fault isolation, the system is engineered to handle high workloads through horizontal scaling while conserving resources during low demand. Utilizing asynchronous communication and orchestration, the system aims to seamlessly integrate components for efficient image tagging and classification. Through an experimental evaluation and load testing, the report validates the system's ability to autonomously scale and adapt to varying workloads, ensuring its effectiveness in real-world scenarios.

Additional Key Words and Phrases: Microservices, Distribution, Orchestration, Scalability

1 PROBLEM STATEMENT

1.1 Scope

1.1.1 Microservices. Systems based on microservices architecture are widespread nowadays, each company which needs to scale is required to have such an architecture. This is an approach to developing a complex application, composed of many business components, as a suite of small services. Each one is running on its own, independently from all the others, using service-specific resources. These services typically implement a single business unit of the whole system, enabling teams and companies to organize and split the jobs to be done flawlessly. Adopting this approach gives companies several advantages such as:

- Scalability: services can be scaled independently, meaning that only the ones that need to welcome a higher request may scale, while all the others can stick on the same resource stack;
- Development: each microservice can be developed independently from all the others, which means that different components can use different technologies and they can be developed and deployed independently. They're also easy to maintain and debug, as they are implementing a single specific business unit;
- Isolation: each microservice is managing its data, meaning that microservices are loosely coupled. Furthermore, it's possible to achieve fault isolation, as errors and crashes are service-specific and they won't cause the whole system to crash.

There are also several challenges while dealing with this architectural paradigm. A microservices application has lots of services that need to work together to provide business features. Each service is simple and specific, but the entire system may be more complex as a result of that. Secondly, microservices embrace distribution and concurrency by definition, as they are components placed in different places that need to cooperate and interact. This can make it more difficult to ensure that the system is working as expected. Furthermore, you need to implement strategies and policies to enable servers placed in different locations to communicate. This also implies dealing with the internet network, which means accounting for latency, network errors, network instability and so on. To overcome these issues, asynchronicity is the only way to keep the system resilient and scalable. Lastly, development and deployment may also be a challenge. Data may not be consistent across the whole system, therefore developers need to deal with eventual consistency. Testing is also harder, as it's more difficult to test the system as a whole. The deployment phase requires specific knowledge about microservices orchestration.

Given all these premises, we aim to develop a system based on the microservices architecture to assess the benefits and drawbacks cited above. We want to face the problems related to microservices from the designing of the system to the concrete realization, both programmatically and theoretically.

1.1.2 Orchestration. Microservices are based on containers: a unit of software that packages up the code and all its dependencies so the application can run reliably and independently from the computing environment. Since microservices work with containers, there will be many pieces to configure and coordinate. Orchestration can help to automate many operations regarding container management. In addition, an orchestrator can ensure high availability across the entire infrastructure and can enable independent deployment, letting the system grow in different directions. We aim to delve into this unfamiliar topic, starting with individual, standalone containers. Our goal is to develop a system capable of self-management.

Orchestration can also be seen as the need to let different microservices talk with each other. Communication is key in a microservice architecture, as single components need to interact with other components to carry over their tasks. There are two main ways in which microservices can communicate: synchronous and asynchronous operations. Synchronous communication uses blocking semantics: one makes a request and waits for the other to respond. It is also known as the request-reply protocol. The synchronous pattern is common with RESTful APIs or gRPC. Asynchronous communication uses non-blocking semantics. Clients and servers send messages without waiting for a response. This pattern is common in messaging systems based on queues and topics. An architecture like this can be also called event-driven architecture because it is based on the publish-subscribe paradigm. For this concept, we want to build our system with scalability in mind. Synchronicity is, by definition, against the concept of scalability. That's why we want to assess that synchronicity is not working for scalability, and therefore use asynchronous communication patterns for letting the services talk with each other.

1.1.3 Scalability. This topic has been extensively explored in the previous sections, leaving little new to add. The key takeaway is that scalability ensures the presence of all other critical features. Therefore, our objective is to create a system inherently scalable, thereby unlocking all the advantages discussed so far. We aim for scalability to facilitate asynchronous communication between microservices and allow the orchestrator to efficiently manage workloads through horizontal scaling of specific components as needed. Moreover, we value scalability for the autonomy it provides to the system, enabling independent development and testing of components.

Scalability is the main goal of our investigation: we want to build a scalable system to perform an experimental evaluation of its features.

1.2 Boundary of the study

The application scenario that we will consider for the development of the system is the following.

The system is called "Image Labeling System". It is a distributed system for image tagging. The main purpose of it is to allow clients to load tons of images to be tagged with labels. Operators, people in charge of selecting the most accurate label for an image, will tag images and then they will be sent back to the client. The returned set of images can be used to train machine learning models. The system should be able to provide the following features:

- Images (task) submission: a client can load several images to be tagged with a set of given labels. A task is none other than the job of tagging all the images loaded at once with the set of labels provided;

- Images tagging request: operators can ask to be given a certain number of images to tag. To avoid trivial errors, an image will be sent to different operators to retrieve different classifications and pick the most selected one;
- Tagged image submission: once an operator has tagged an image, it will be sent back to the system for further evaluation.

Here is the typical workflow for a task:

(1) Images submission: the client needs images with labels to train a new machine learning model. He will send a bunch of images along with the possible labels that can be attached to the images. This will trigger the generation of a task inside the system. Once the task is created, the system will spawn as many subtasks as the number of images to classify. A subtask will involve the classification of a single image with the given labels;
(2) Images tagging request: operators will ask to be given a certain number of subtasks. The same subtasks will be assigned to different operators. This is done to avoid malicious tagging (and to give the system an additional complexity in managing this constraint);
(3) Images tagging: operators will tag the image by picking a label from the pool of labels available (based on what the client allowed). Then the result will be sent back to the system;
(4) Verification and aggregation: once the system has gathered enough tags for an image (subtask), it will aggregate the result and pick a "final" label. The definitive label is going to be the one selected the most among all the possibilities. The subtasks will be marked as completed when a final label is assigned to the corresponding image;
(5) Task Completion: The task is deemed complete once all subtasks are closed, at which point the labeled images will be returned to the client who requested the classification.

1.3 Purpose

1.3.1 Goals of investigation. The main goal of the investigation was to design and develop a system based on a microservices architecture, capable of scaling horizontally as needed to maintain performance levels even under high usage conditions. Similarly, the system is designed to scale down and conserve resources when the demand is low or decreasing.

To achieve this, a thorough exploration of the leading technologies currently available in the market was conducted, evaluating various options. After considering the pros and cons of each, the chosen technologies were selected and are detailed in this document.

Following the identification of the best technologies, another goal was to study and understand their functionality before actively using them in the actual implementation of the system.

The system was then implemented, and the final objective was to test how it responded to various workloads. Specifically, the focus was on the system's ability to autonomously scale and ensure resource usage remained below a specified threshold.

1.3.2 Experiments. For what concerns the experiments we want to assess that the small Proof of Concept that we created is right from a technical and practical point of view. From a technical perspective, we want to assess that the infrastructure we designed can be used in a real-world scenario such as the one described above. This implies checking that the system has all the features mentioned above, which makes microservices architectures strong and suitable for large-scale applications such as the one we are targeting. Therefore, we consider the PoC itself an experiment, which is meant to develop and test the single components both individually and globally, by using the system "as a whole". From a more practical point of view, we want to verify that the microservices can asynchronously interact with each other, guaranteeing scalability by design in this sense. We want to make sure that the orchestration of the system is working properly.

The main part of the experiment consists of a load test performed on the entire system. Since scalability is one of our major concerns, we think that if the system can handle an increased workload employing autoscaling, then the whole system is well-designed. All the previous assessments are meant to enable this kind of evaluation. Scalability can be seen as the culmination of several practices and features that can be present in a system only if the system is well-designed and implemented. To sum everything up, we want to evaluate the goodness of our solutions by testing the system with different workloads (variation across time), to verify that the individual components can handle variations in the workload (both increasing and decreasing). In other words, we want to assess that the system can scale both on individual microservices and globally.

2 EXPERIMENTAL IMPLEMENTATION

2.1 Technical choices made in the realization of the PoC

2.1.1 System overview. The system can be seen in fig. 1, which represents the individual components of the system and their interactions.

2.1.2 Main technologies. Before delving into the details of each component, we present a comprehensive list of the technologies utilized in developing the system. Our discussion will primarily focus on technologies that are common to multiple components or do not belong exclusively to any single component. Components can fall into two categories: the ones developed by us and the ones using external services, as they're state-of-the-art for the problem they're addressing.

To develop the two main microservices of the application, task-manager and subtask-manager, we chose Go as a programming language. It is well-suited for distributed systems and cloud-first development due to its strong support for concurrency, efficient resource management and built-in networking capabilities. Its simplicity, speed and reliability make it ideal for building scalable microservices and handling high levels of concurrent requests. Additionally, its native support for JSON and efficient handling of HTTP requests further contribute to its suitability for building cloud-based applications.

The RESTful paradigm has been chosen as the communication method to expose the features of the microservices to external users. It offers a uniform and scalable approach for components to communicate via the web, fostering loose coupling and scalability. This is achieved through the use of standard HTTP methods and data formats, which facilitate smooth interaction and data exchange between different components. We chose to use Gin, a Go web framework, to provide RESTful APIs. It features a robust routing mechanism, middleware support and efficient request handling, making it ideal for creating high-performance APIs and web services. It is one of the most popular web frameworks to build scalable and reliable applications thanks to its simplicity and speed.

Lastly, we mention Kubernetes. It is an open-source container orchestration platform which is designed to automate the deployment, scaling, and management of containerized applications. It provides a container-centric infrastructure, abstracting away the underlying infrastructure complexity and enabling developers to focus on building and deploying applications. Kubernetes automates tasks such as scheduling containers, scaling applications based on demand, managing storage and ensuring high availability. It offers features like service discovery, load balancing and rolling updates, making it well-suited for deploying and managing distributed applications across hybrid environments.

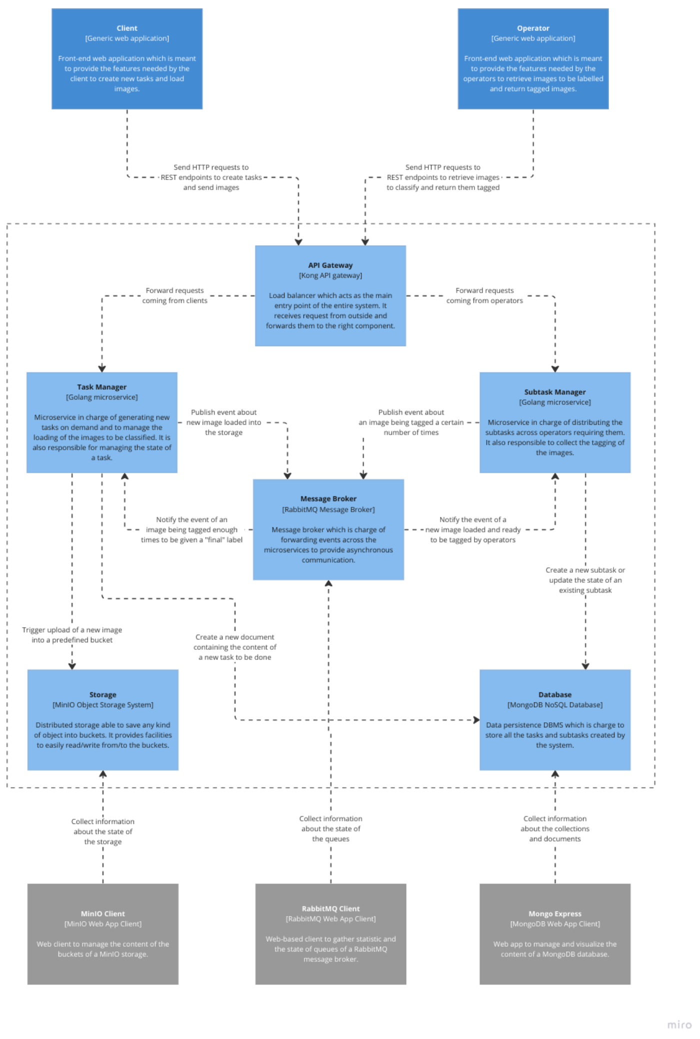

Client
[Generic web application]
Front-end web application which is meant to provide the features needed by the client to create new tasks and load images.
Operator
[Generic web application]
Front-end web application which is meant to provide the features needed by the operators to retrieve images to be labelled and return tagged images.
Send HTTP requests to REST endpoints to create tasks and send images
Send HTTP requests to REST endpoints to retrieve images to classify and return them tagged
API Gateway
[Kong API gateway]
Load balancer which acts as the main entry point of the entire system. It receives request from outside and forwards them to the right component.
Forward requests coming from clients
Forward requests coming from operators
Task Manager
[Golang microservice]
Microservice in charge of generating new tasks on demand and to manage the loading of the images to be classified. It is also responsible for managing the state of a task.
Publish event about new image loaded into the storage
Publish event about an image being tagged a certain number of times
Subtask Manager
[Golang microservice]
Microservice in charge of distributing the subtasks across operators requiring them. It also responsible to collect the tagging of the images.
Notify the event of an image being tagged enough times to be given a "final" label
Message Broker
[RabbitMQ Message Broker]
Message broker which is charge of forwarding events across the microservices to provide asynchronous communication.
Notify the event of a new image loaded and ready to be tagged by operators
Create a new subtask or update the state of an existing subtask
Trigger upload of a new image into a predefined bucket
Create a new document containing the content of a new task to be done
Storage
[MinIO Object Storage System]
Distributed storage able to save any kind of object into buckets. It provides facilities to easily read/write from/to the buckets.
Database
[MongoDB NoSQL Database]
Data persistence DBMS which is charge to store all the tasks and subtasks created by the system.
Collect information about the state of the storage
Collect information about the state of the queues
Collect information about the collections and documents
MinIO Client
[MinIO Web App Client]
Web client to manage the content of the buckets of a MinIO storage.
RabbitMQ Client
[RabbitMQ Web App Client]
Web-based client to gather statistic and the state of queues of a RabbitMQ message broker.
Mongo Express
[MongoDB Web App Client]
Web app to manage and visualize the content of a MongoDB database.
miro

Fig. 1. System overview by means of a C4model schema.

2.1.3 Limitations of the system. While developing the system we chose to focus mainly on the development of the essential components of the application. We decided to give more value to the infrastructure and its management rather than implementing very sophisticated and "precise" components. Because of this approach, we did not develop some features which should be present for sure in a real-world system. Additionally, we believe that developing these features would not have provided any benefit to the system or its experimental evaluation. From our perspective, it would have been akin to introducing noise into an already complex subject. Based on these considerations, below is a partial list of features that are not included in the system:

- Authentication: there are no authentication mechanisms or constraints. Including this feature would have implied the creation of another microservice in charge of validating the source and the request before letting it go through the system;
- Requests' constraints: there are no hard constraints on the request that can be made towards the system, which means that the system can welcome and process even "malformed" requests (e.g. containing unrealistic data). Since the system has been used mainly for testing purposes, there was no need to develop such hard constraints;
- Autoscaling: scalability has been one of our major concerns while implementing the system. As we will mention later, the custom-defined microservices have been configured to autoscale using Kubernetes. On the other hand, external systems such as MongoDB, RabbitMQ and MinIO have not been equipped with the autoscaling feature. This can be seen as (and actually is) a major limitation since we are dealing mainly with scalability. We didn't include autoscaling for these services as the process is not very trivial and it would have needed a lot of time to make everything work. Since we are leveraging external systems which are known to be scalable by design, we chose to not autoscale them. To avoid bottlenecks on this side, we configured such external systems to have enough hardware resources to handle quite well even huge workloads (i.e. the ones used during the experimental workload evaluation).

2.1.4 API gateway. Within our system, we decided to adopt the API Gateway pattern. The API Gateway pattern involves using a single entry point to handle all client requests and manage interactions between clients and backend services. This pattern simplifies client access to multiple services, facilitating the scaling and management of the microservices. We decided to leverage Kong as an API Gateway. Kong is an open-source API gateway and microservices management layer built on top of Nginx. It provides a scalable, high-performance platform for managing, securing and monitoring APIs and microservices. We picked the Kong Ingress Controller as the component implementing this pattern, as it was easily integrated into the Kubernetes cluster. On our system, as mentioned above, the API Gateway will act as the single entry point of the system, with the duty of routing the requests coming from the outside towards the right component, based on the functionality requested.

2.1.5 Image storage. MinIO is an open-source, self-hosted object storage server that allows users to store unstructured data like photos, videos and files of any kind in general. It is designed to be highly scalable and performant, capable of handling large amounts of data with ease. It's often used for building private cloud storage solutions, data lakes and as a backend for cloud-native applications. On our system, it's used as a storage which is in charge of saving the images (to be tagged) uploaded by the clients. To carry over this task, the microservices will use the APIs exposed by MinIO to load images into a predefined bucket.

2.1.6 Message broker. A message broker is a software intermediary that facilitates communication between different applications or components by routing, storing, and delivering messages. It acts as a central hub where producers can publish messages and consumers can subscribe to receive them. Message brokers enable asynchronous communication, decoupling producers and consumers and ensuring reliable message delivery. That is why it is a critical component of the system: without it, the microservices won't be able to interact. We picked RabbitMQ as a message broker. RabbitMQ is an open-source message broker software that implements the Advanced Message Queuing Protocol (AMQP). It provides features such as message queuing, routing, reliability, clustering, and high availability. RabbitMQ supports various messaging patterns, publish/subscribe in particular. We leveraged this particular pattern to implement asynchronous communication among the microservices. Both task-manager and subtask-manager will be publishers and subscribers at the same time. For this purpose, two queues have been defined:

- New subtask: this queue delivers events from task-managers towards subtask-managers. When a new image is uploaded, the task-manager will issue an event stating that a new image is ready to be tagged by operators. A subtask-manager, which is the consumer of events of this kind, will process the event by creating a new subtask and issuing it to operators;
- Completed subtask: this queue delivers events from subtask-managers towards task-managers. When an image (subtask) has received enough tags, that subtask is considered to be done. Therefore the subtask-manager will issue an event stating that the subtask has been completed. Such an event will be captured by a task-manager which will process it by assigning to the image a "final" label, based on some kind of business logic.

2.1.7 Database. MongoDB is a popular open-source NoSQL DBMS which is designed for storing and managing unstructured or semi-structured data. It uses a flexible document-oriented data model, where data is stored in JSON-like documents with dynamic schemas. MongoDB offers features such as high availability and horizontal scalability. From a theoretical point of view, each microservice would need its database. But, for the premises mentioned above, we didn't equip the microservices with individual and isolated databases. We believe that this practice would not have benefitted the system; instead, it would have introduced additional complexity to manage. To mimic this behaviour, which is indeed correct and should have been implemented, we established separate collections for the two systems. The collections are the following:

- Tasks: collection managed by task managers. Each document represents a specific task created by a task manager. A task consists of tags that can be assigned to images and a list of subtasks. Each subtask involves tagging a specific image that is part of the task. Initially, a subtask will have an ID linked to its parent task's ID and an empty label. The label will be completed once the subtask manager informs the task manager that the particular subtask has been finished;
- Subtasks: collection managed by subtasks-managers. Each document represents a subtask. A subtask is created as a response to the reception of an event indicating that a particular image has been uploaded. It is composed of the set of tags that can be assigned (to the image to which the subtask refers) and a list of assigned tags (i.e. the ones assigned by the operators). The id of a subtask is going to be the same as the one provided in the event payload (to guarantee a 1:1 correspondence between subtasks on the tasks collection and the ones on the subtasks collection). Every time an operator submits a tagged image, the corresponding subtask will be updated by increasing by one the value corresponding to the assigned label.

2.1.8 Task manager. The task manager is responsible for overseeing the submission of a group of images to be tagged. Additionally, it is tasked with assigning a label to a specific subtask upon its completion.

From a functional point of view, here is an overview of how the service behaves. Clients send a potentially huge block of images. Each block of images to be tagged corresponds to a task to be performed by the system. The block of images must be sent along with a set of labels: this one will be used by operators to tag individual images.

From the point of view of the implementation, upon reception of a request from a client, the service must process all the images following these steps:

(1) Initially, it will create a task, intended as the task of tagging those images with the provided labels;
(2) It must validate that a particular uploaded file is an image;

(3) It must upload the images into the storage (bucket in MinIO);
(4) On upload completion, it must populate the previously-created task with a new subtask, representing the duty of tagging the just uploaded image;
(5) Lastly, once the creation of the subtask (on the database) has been successful, the service must publish a new event into the message broker, stating that a new image is available and ready to be tagged by operators.

The working flow of this microservice is represented by fig. 2.

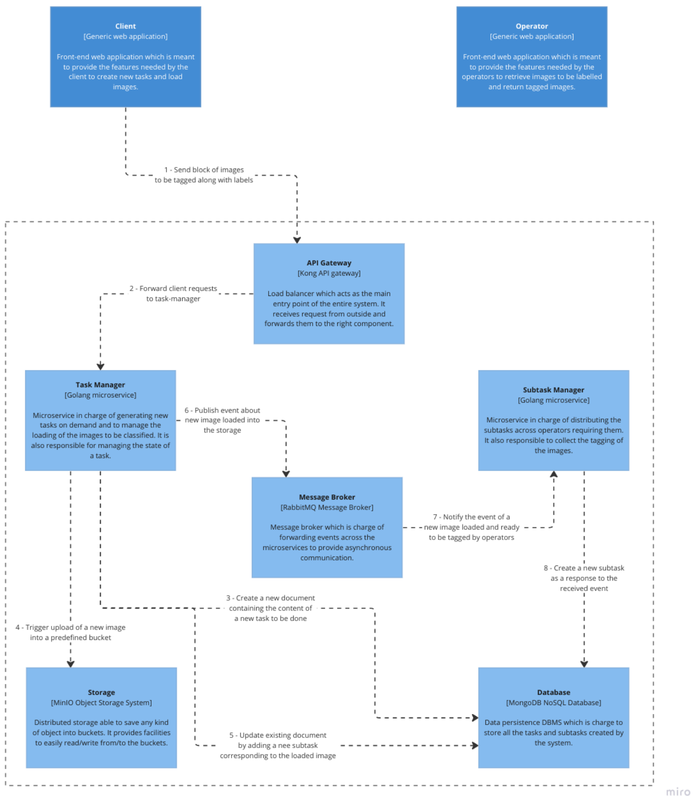

Client
[Generic web application]
Front-end web application which is meant to provide the features needed by the client to create new tasks and load images.
Operator
[Generic web application]
Front-end web application which is meant to provide the features needed by the operators to retrieve images to be labelled and return tagged images.
1 - Send block of images to be tagged along with labels
API Gateway
[Kong API gateway]
Load balancer which acts as the main entry point of the entire system. It receives request from outside and forwards them to the right component.
2 - Forward client requests to task-manager
Task Manager
[Golang microservice]
Microservice in charge of generating new tasks on demand and to manage the loading of the images to be classified. It is also responsible for managing the state of a task.
6 - Publish event about new image loaded into the storage
Subtask Manager
[Golang microservice]
Microservice in charge of distributing the subtasks across operators requiring them. It also responsible to collect the tagging of the images.
Message Broker
[RabbitMQ Message Broker]
Message broker which is charge of forwarding events across the microservices to provide asynchronous communication.
7 - Notify the event of a new image loaded and ready to be tagged by operators
8 - Create a new subtask as a response to the received event
3 - Create a new document containing the content of a new task to be done
4 - Trigger upload of a new image into a predefined bucket
Storage
[MinIO Object Storage System]
Distributed storage able to save any kind of object into buckets. It provides facilities to easily read/write from/to the buckets.
5 - Update existing document by adding a nee subtask corresponding to the loaded image
Database
[MongoDB NoSQL Database]
Data persistence DBMS which is charge to store all the tasks and subtasks created by the system.
miro

Fig. 2. Task manager working flow by means of a C4model schema.

2.1.9 Subtask manager. The subtask manager is responsible for providing the images (subtasks) to be tagged requested by the operators. Once they have tagged them, it has the task of storing, for each subtask, the various labels received. As soon as the necessary number of labels is reached, the subtask will be considered completed, therefore the task manager will be notified, along with the information related to the various labels assigned to that specific image.

An operator initiates the process by requesting a group of subtasks, each representing an image to be classified. These subtasks are dispatched to the requesting operator and include both the ID of the image and a selection of possible labels that can be assigned to it. The operator then assigns an appropriate label to each image, and these classification outcomes are communicated back to the subtask manager.

Here's a more detailed description of how the workflow is implemented:

(1) The subtask-manager creates subtasks in response to notifications from the task manager triggered by the upload of individual images. A subtask is defined by the image's ID, a set of assignable labels for that image and a list of labels that will be assigned by operators;
(2) When an operator requests a set of images, the service provides a set of images (subtasks) for completion;
(3) After the operator submits the classified images, the service updates each subtask by adding the new label provided by the operator;
(4) Once a subtask accumulates a sufficient number of labels, signaling that the classification process is complete, the service informs the task manager of this achievement, indicating that the subtask has met its required label count.

In fig. 3, a graphical overview of the flow when an image has been classified by an operator and the result is sent back to the subtask-manager service.

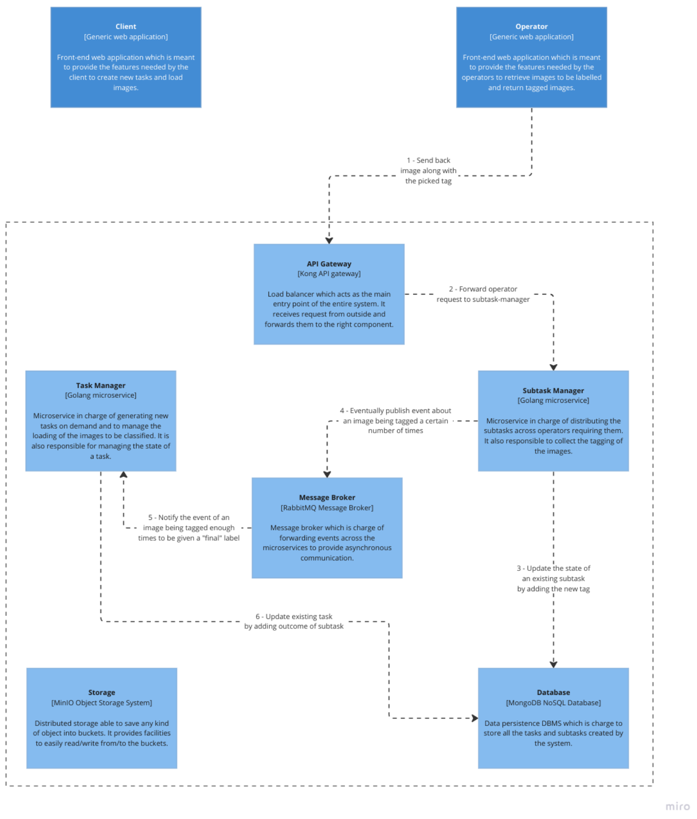

Client
[Generic web application]
Front-end web application which is meant to provide the features needed by the client to create new tasks and load images.
Operator
[Generic web application]
Front-end web application which is meant to provide the features needed by the operators to retrieve images to be labelled and return tagged images.
1 - Send back image along with the picked tag
API Gateway
[Kong API gateway]
Load balancer which acts as the main entry point of the entire system. It receives request from outside and forwards them to the right component.
2 - Forward operator request to subtask-manager
Task Manager
[Golang microservice]
Microservice in charge of generating new tasks on demand and to manage the loading of the images to be classified. It is also responsible for managing the state of a task.
4 - Eventually publish event about an image being tagged a certain number of times
Subtask Manager
[Golang microservice]
Microservice in charge of distributing the subtasks across operators requiring them. It also responsible to collect the tagging of the images.
Message Broker
[RabbitMQ Message Broker]
Message broker which is charge of forwarding events across the microservices to provide asynchronous communication.
5 - Notify the event of an image being tagged enough times to be given a "final" label
3 - Update the state of an existing subtask by adding the new tag
6 - Update existing task by adding outcome of subtask
Storage
[MinIO Object Storage System]
Distributed storage able to save any kind of object into buckets. It provides facilities to easily read/write from/to the buckets.
Database
[MongoDB NoSQL Database]
Data persistence DBMS which is charge to store all the tasks and subtasks created by the system.
miro

Fig. 3. Subask manager working flow by means of a C4model schema.

2.1.10 Others. The services not mentioned above such as Mongo Express, RabbitMQ UI and MinIO UI are just utility tools which are used to make it easier to develop and debug the system. They provide facilities to access the actual content of the service they're "targeting".

2.2 Design of evaluation experiments

As previously mentioned, our experimental evaluation aims to assess the system's capability to autonomously scale up and down in response to fluctuating workloads over time. In our opinion, if the system is resilient to such drastic workload changes, then it is proof that it has been developed correctly from a scalability perspective. If this is proven, then we can say that our system can scale, therefore it embraces the benefits mentioned in the first section about microservices architecture. We don't want to focus on the "external" components, as they're third-party products which already have their solutions to scale up. We will focus mainly on the microservices we created to fulfill the requirements of our application.

The anatomy of the test is pretty straightforward: the system will receive several requests from both clients and operators. To emulate the behavior of the two subjects, we defined two custom Go programs. The structure of both is pretty similar: initially, it will spawn several goroutines (representing different users of the same role interacting with the system). Each goroutine will loop for a certain amount of time (the duration of the test): on each iteration there will be a "business" request towards the system. This means that meaning that clients will upload new tasks while operators will ask images to tag and they will immediately send back the response. Between the iterations, a sleeping time has been placed: the purpose of this "pause" is to tune the number of requests, its value will be changed over time to simulate increasing and decreasing demand towards the application. Therefore, we can say that the number of requests from both sides will follow a "pyramidal" strategy: it will start slowly, then it will have a huge spike before going down again.

We deployed our Kubernetes cluster on a local machine using Minikube. Minikube is a tool that enables users to run a Kubernetes cluster locally on a single node. It has been designed to easily test Kubernetes deployments, allowing them to experiment with containerized applications without needing a full-scale cluster setup. Thanks to this tool, we were able to deploy our Kubernetes cluster locally on a Lenovo T440s running Ubuntu 22.04.3 LTS, with an Intel Core i5-4300U CPU equipped with 4 cores.

The test lasted 10 minutes. At the beginning of the test, both Go programs were launched simultaneously. 15 goroutines were launched to emulate clients, while 10 were launched to emulate operators. We decided to issue more clients to avoid subtasks shortage and because of the need to upload the images (which is a more expensive operation). To see the effects of the autoscaling, we gave each replica a limited amount of CPU and memory resources. We fixed the hard limit on CPU at 80%, meaning that if a replica exceeds such constraints, then autoscaling will come into play, deploying a new replica. The number of requests followed the pattern represented by table 1.

To collect the evidence about scalability happening inside the system, we collected some statistics:

- A shell script which will collect data about the state of the Kubernetes resources every 10 seconds. To retrieve such information we used two commands provided by kubectl, a command-line tool used to interact with a Kubernetes cluster. In particular, we collected information about the hardware resource usage of the replicas and the state of autoscaling (numbers of replicas deployed);
- A response time, defined as the elapsed time within an iteration of the for loop on each goroutine running.

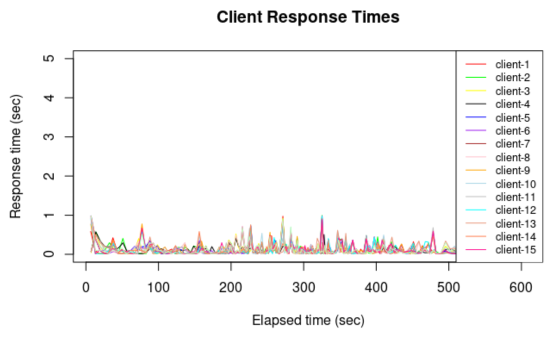
Table 1. Summary of the test setup

Elapsed time	Task manager delay	Subtask manager delay
0th min	6 sec	9 sec
1st min	5 sec	7 sec
2nd min	4 sec	5 sec
3rd min	3 sec	4 sec
4th min	2 sec	3 sec
5th min	2 sec	3 sec
6th min	3 sec	4 sec
7th min	4 sec	5 sec
8th min	6 sec	9 sec
9th min	6 sec	9 sec

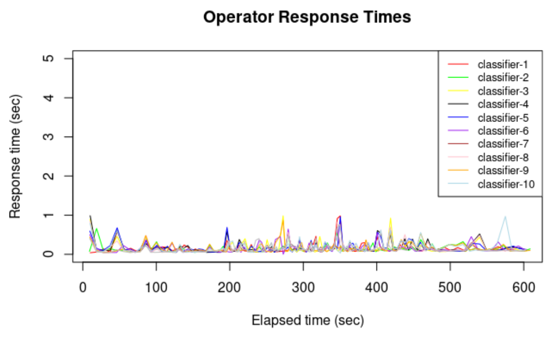
Using the first metric, our goal is to evaluate whether the system scales appropriately as the workload grows. By using the statistics, we want to verify that the response time doesn't increase even if the workload towards the system increases. Both metrics are related and they can prove the goodness of the result.

2.3 Results of evaluation experiments

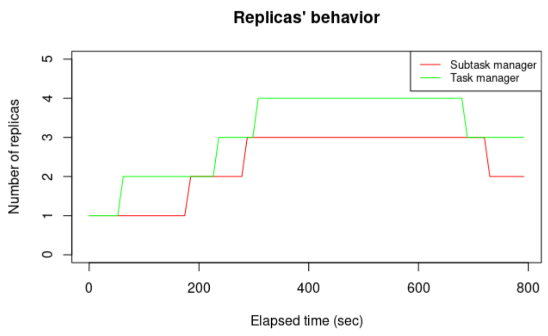
Figures 4a, 4b, 5a and 5b explain the outcome of the test.



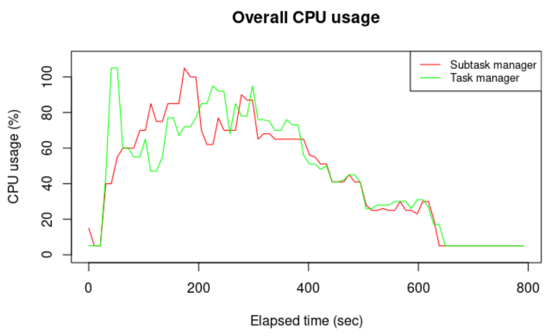
(a) Clients response time



(b) Operators response time



(a) Replicas behaviour per microservice



(b) CPU usage per microservice

First of all, let's have a look at fig. 5b about the overall CPU usage. This graph represents the mean CPU usage of all the replicas present at a specific moment. As we can see, we can recognize the "overall" shape of a pyramid, as a result of increasing workload.

On the other side, fig. 5a represents the number of replicas present at a specific moment. Even here it's possible to recognize the same shape. It's also interesting to notice that, if we overlap the two grapes, every time a new replica has been deployed there is a local minimum in fig. 5a, meaning that the system is dealing with an increased workload thanks to the addition of new resources.

We can already say that the system is scaling according to the workload. We can also see that after the test was over (600 seconds), the replicas were going to be deallocated, as a result of a huge drop in CPU usage.

For what concerns fig. 4a and fig. 4b, we can see the response times of the emulated users (goroutine). In this case, there's no evidence (apart from individual spikes) about increasing response time when the workload is increasing. It stays constant over time.

These metrics represent the fact that the system, using Kubernetes and load balancers, is handling the workload nicely, issuing new replicas when needed and removing them when not needed anymore. As a final consideration, we can say that this output demonstrates that our system can scale as a result of implementing it by using best practices and asynchronicity by design.

Other tests could have been performed to find "breaking points" within the system. In our opinion, our PoC would never break because of the pitfalls of the system itself, otherwise they would have arisen even when running the simple test mentioned above. We believe that, as long as the orchestrator can keep up the pace with the increasing workload, then the system has no problem in dealing with scalability.

3 SELF-ASSESSMENT

3.1 Own critique of own exam product

The two main criticisms we have regarding our final product are the following: first, we did not manage the operator's requests in a controlled manner. For instance, since we developed the product solely for testing purposes, we did not incorporate any protections or security measures against malicious activities by operators. An example of this would be that an operator could submit unlimited classifications for the same image, potentially leading to incorrect image classification. The second shortfall is the absence of an authentication system, allowing malicious users to interact with and potentially influence the data within the system without needing to authenticate.

These issues should be addressed if the system is to be deployed in the future. To safeguard the integrity of the obtained classifications, it is crucial to implement measures that address and prevent the issues mentioned earlier.

Another critical gap is that services not developed by us (e.g. rabbitMQ, MongoDB and MinIO) do not scale in response to increased workload. As mentioned in earlier sections, making these services scalable would not have significantly benefited the system since our goal was to test only the services implemented by us. However, if this system should be used in real-world scenarios, it should include the auto-scaling of external services.

Additionally, a complete front-end component should be developed to enable adequate interaction with the system.

For the sole purpose of the tests conducted, we do not believe our system has critical flaws. The design was constructed coherently and robustly, with a focus on scalability. The implementation follows the principle of separation, and the system has proven to be both solid and scalable.

3.2 Discussion of learning outcomes in this work

In this project, we had the opportunity to apply the theoretical concepts studied during the course by tackling the main challenges encountered in a real-world scenario of a system based on microservices. The primary theoretical concepts we encountered in the system development included:

- Microservices Architecture: we designed the architecture by breaking down the initial macro-problem into subproblems. For each identified subproblem, we associated a microservice able to solve a specific set of tasks independently. Our exploration into microservices revealed the balance between the autonomy of individual services and the complexity of managing them as a cohesive unit. We learned that while microservices offer significant advantages in terms of scalability, development, and isolation, they also introduce challenges in ensuring system-wide consistency and managing service interactions. This underscored the importance of careful design and planning in microservices architecture to mitigate complexities and leverage its benefits effectively;
- Orchestration: we discovered how orchestration tools, particularly Kubernetes, facilitate the automation of deployment, scaling, and management of services. This understanding highlighted the importance of orchestration in achieving high availability, independent deployment, and scalability in a microservices ecosystem;
- Scalability: we aimed to build a system that could dynamically adjust to workload variations. Through this process, we learned about the mechanisms and strategies that enable a system to scale horizontally. The practical experience of implementing and testing these strategies provided us with a deep appreciation for the role of scalability in maintaining system performance and reliability.

Additionally, we analyzed the best technologies for creating a scalable microservices system that are available in the market today. After selecting the ones that were optimal for us and our case study, we had the opportunity to learn them, finding a lot of material and excellent explanations online that made learning these technologies relatively quick to create the Proof of Concept in question.

We have thus developed an experience with these technologies, which have infinite applications in today's world. Overall, this work has been a comprehensive learning journey that has increased our understanding of microservices, orchestration, and scalability as well as equipped us with practical skills in system design, implementation and evaluation.