Architecture design

Portfolio semester 6

Table of contents

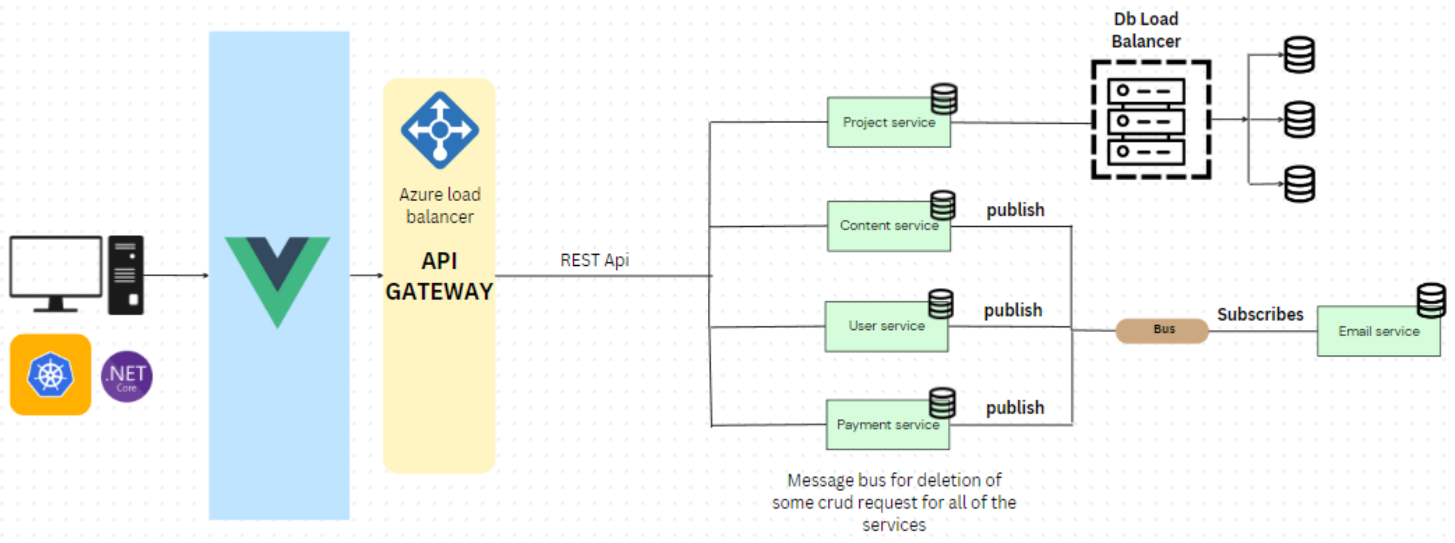
Introduction

In enterprise software development, functionality is not the only consideration in creating an effective architecture. Quality attributes that are relevant to high volume data and events in an enterprise context must also be taken into account. This is the focus of Learning Outcome 5 - Scalable Architectures. With future adaptation in mind, the architecture should be designed to consist of independently running parts, such as microservices, that can be deployed and monitored independently. Asynchronous communication through messaging is also important. To ensure scalability, the developer must investigate which performance indicators to measure and monitor, and validate that the application automatically scales under realistic loads. Proper technology selection is also crucial. Techniques such as Event Storming are used during analysis and design to help create scalable architectures. The use of industry standards such as the C4 model and UML helps to communicate and justify architectural choices to stakeholders and the development team.

Microservice Architecture

In its essence, microservices are decoupled, almost stand-alone applications that have their own persistent storage. Furthermore, they communicate through a predefined protocol, based on the type of latency needed.

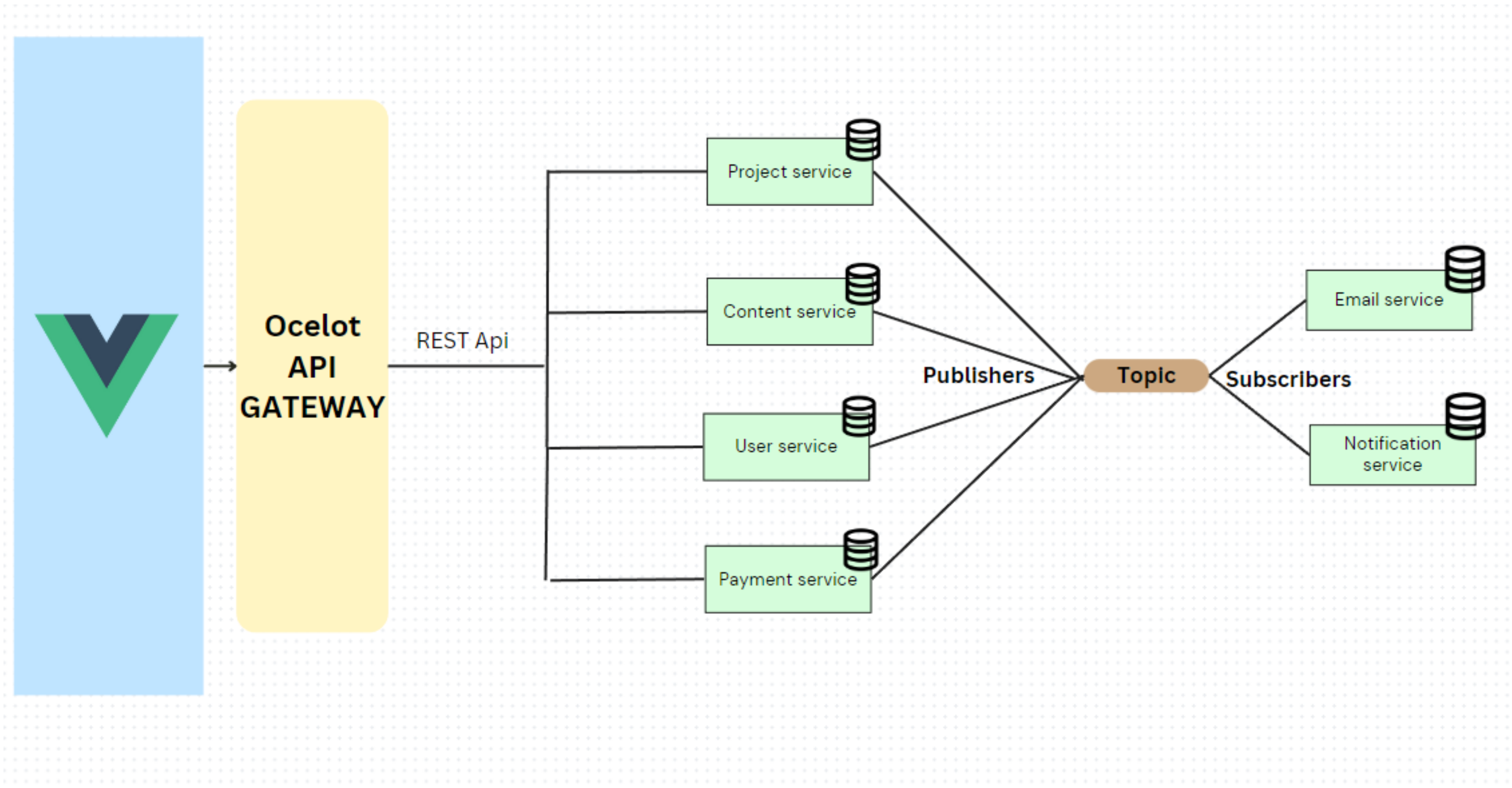
Microservice architecture overview - v1



In the above diagram, 4 services are defined, all of which can be accessed through an inbuilt API gateway that uses different techniques and algorithms for loadbalancing.

Furthermore, three of the services can be seen to publish messages into a topic for which the email service is responsible for consuming.
After further observation of what the requirements are for the project and what is necessary for the services, the following updated overview can be seen below:

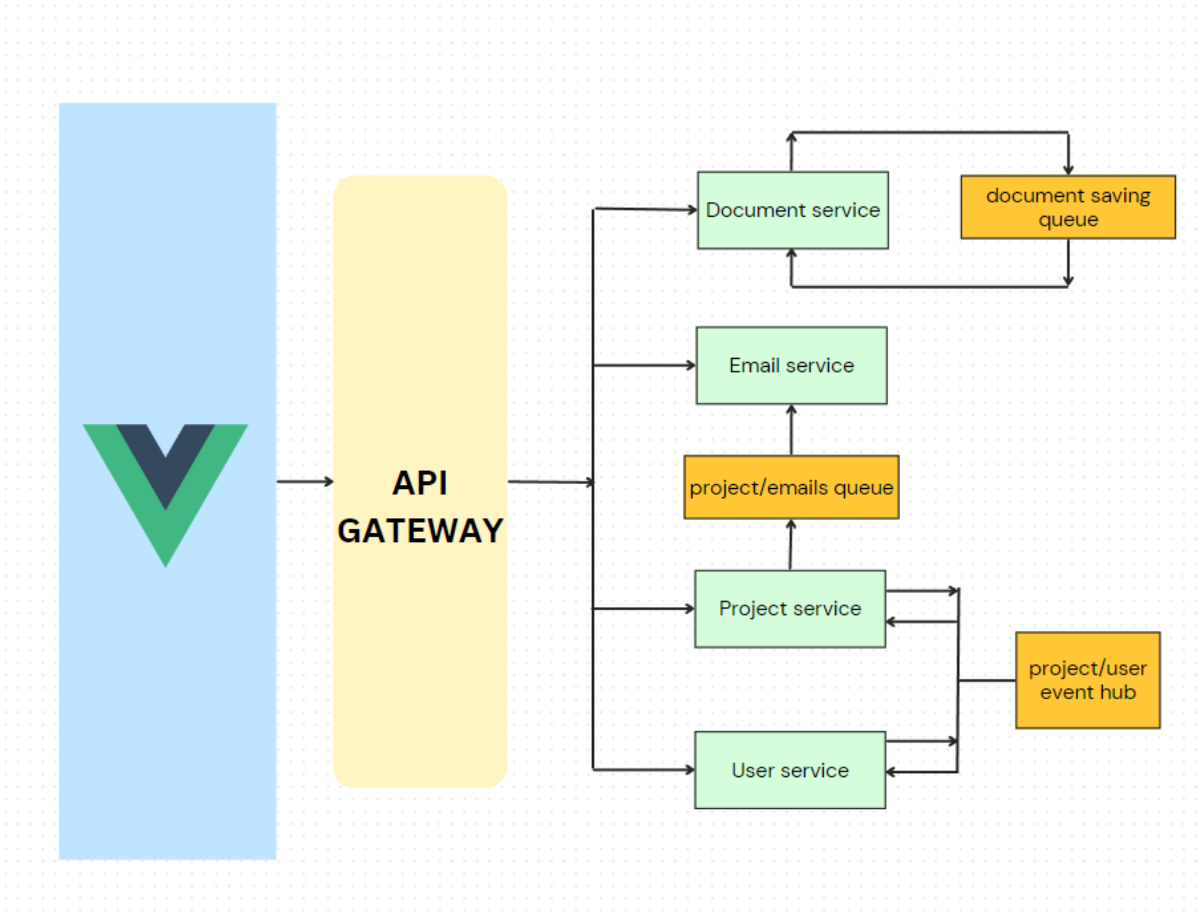
Microservice architecture overview - v2



Since the application is following the microservice architecture design patterns and best practices, each of the services contain a copy of the data they need from the other services. This means that they are fully functional, despite the fact that one of the other services shuts down or must go through updates.

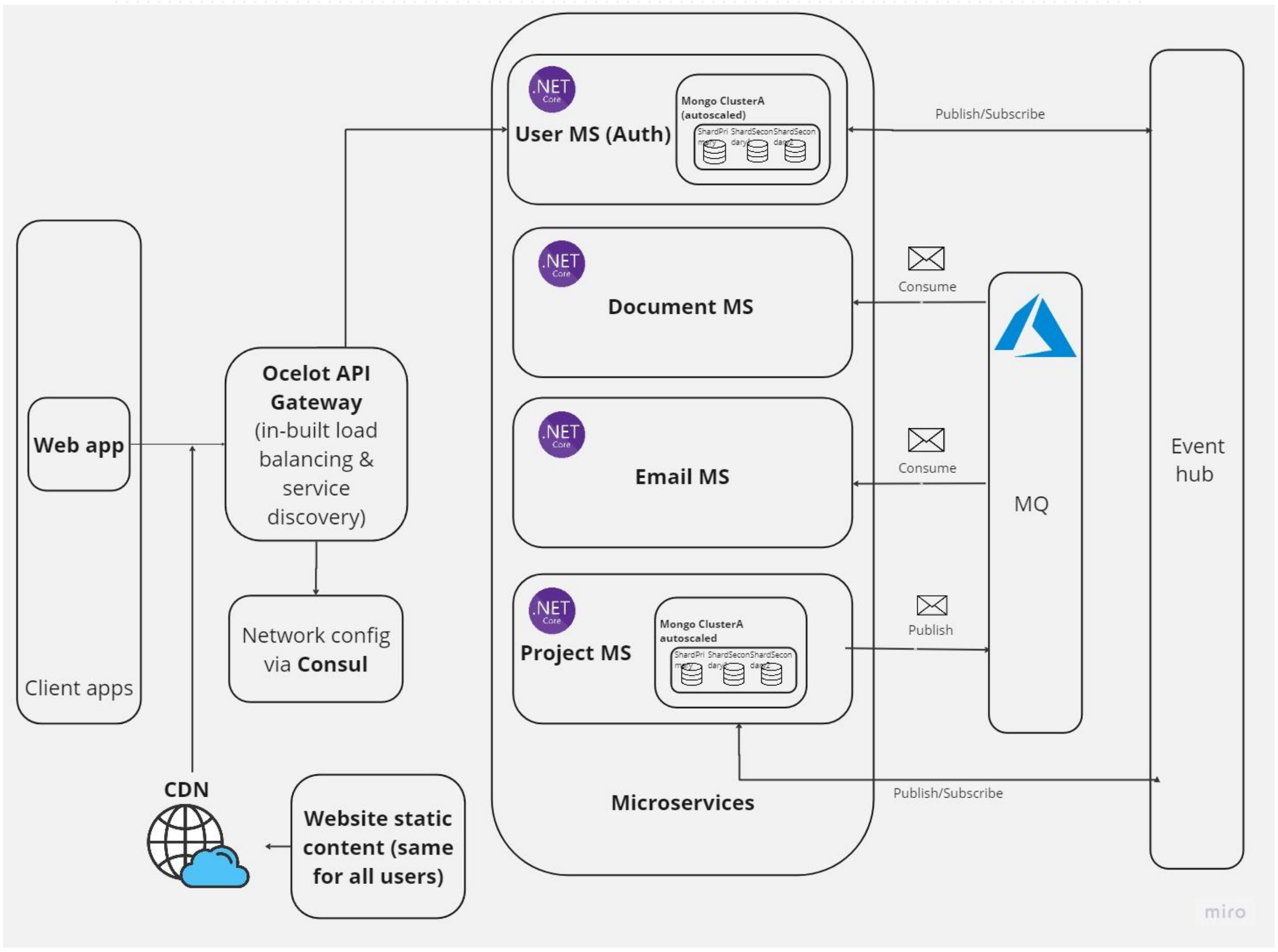
This means that when a record must be updated / deleted globally, all of the services must listen to a specific event channel/bus that will notify them on what record must be deleted.

Update 18/05/23 - Architecture diagram v4



The above diagram best describes the current state of Copycloud's system architecture. At this current moment, this involves a singular queue for document saving, a message queue for emails between the project service and email service. Finally, the project service and the user service are both publishers and subscribers to an event hub. The reason behind this is so that event based deletion of users/projects can take place and data can be distributed/deleted globally.

Update 24/05/23 - Architecture diagram v5



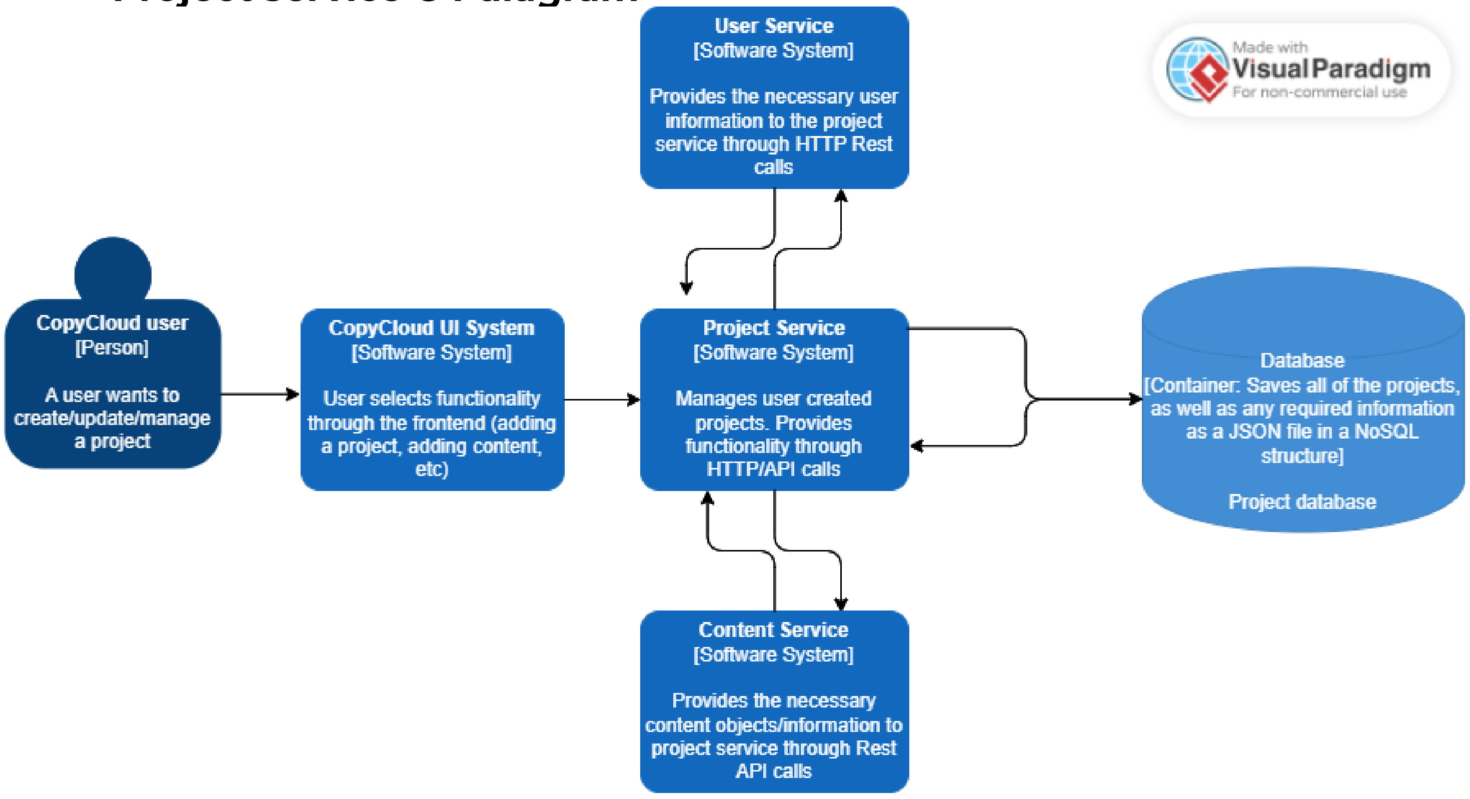
Web app
Client apps
Ocelot API Gateway (in-built load balancing & service discovery)
Network config via Consul
CDN
Website static content (same for all users)
User MS (Auth)
Mongo ClusterA (autoscaled)
Document MS
Email MS
Project MS
Mongo ClusterA autoscaled
Microservices
Publish/Subscribe
Consume
Consume
Publish
MQ
Event hub
Publish/Subscribe
miro

Project specific requirements

This section concerns the architectural designs behind the project service. It serves the purpose of creating, updating and reading projects, as well as all of the user generated content that goes into each one.

In the following section, different project specific use cases and their architectural choices will be explained with the use of diagrams.

Project service C4 diagram


User Service
[Software System]
Provides the necessary user information to the project service through HTTP Rest calls
Made with Visual Paradigm For non-commercial use
CopyCloud user
[Person]
A user wants to create/update/manage a project
CopyCloud UI System
[Software System]
User selects functionality through the frontend (adding a project, adding content, etc)
Project Service
[Software System]
Manages user created projects. Provides functionality through HTTP/API calls
Database
[Container: Saves all of the projects, as well as any required information as a JSON file in a NoSQL structure]
Project database
Content Service
[Software System]
Provides the necessary content objects/information to project service through Rest API calls

In order to properly break down the components of a system, we must inspect specific use cases and functionalities for the project itself. In the above shown C4 diagram, we can see the breakdown of the project CRUD functionality flow.
The communication between the user service, project service and content service is done through regular HTTP REST calls, as the project needs to validate each response simultaneously.

Notifications and emails

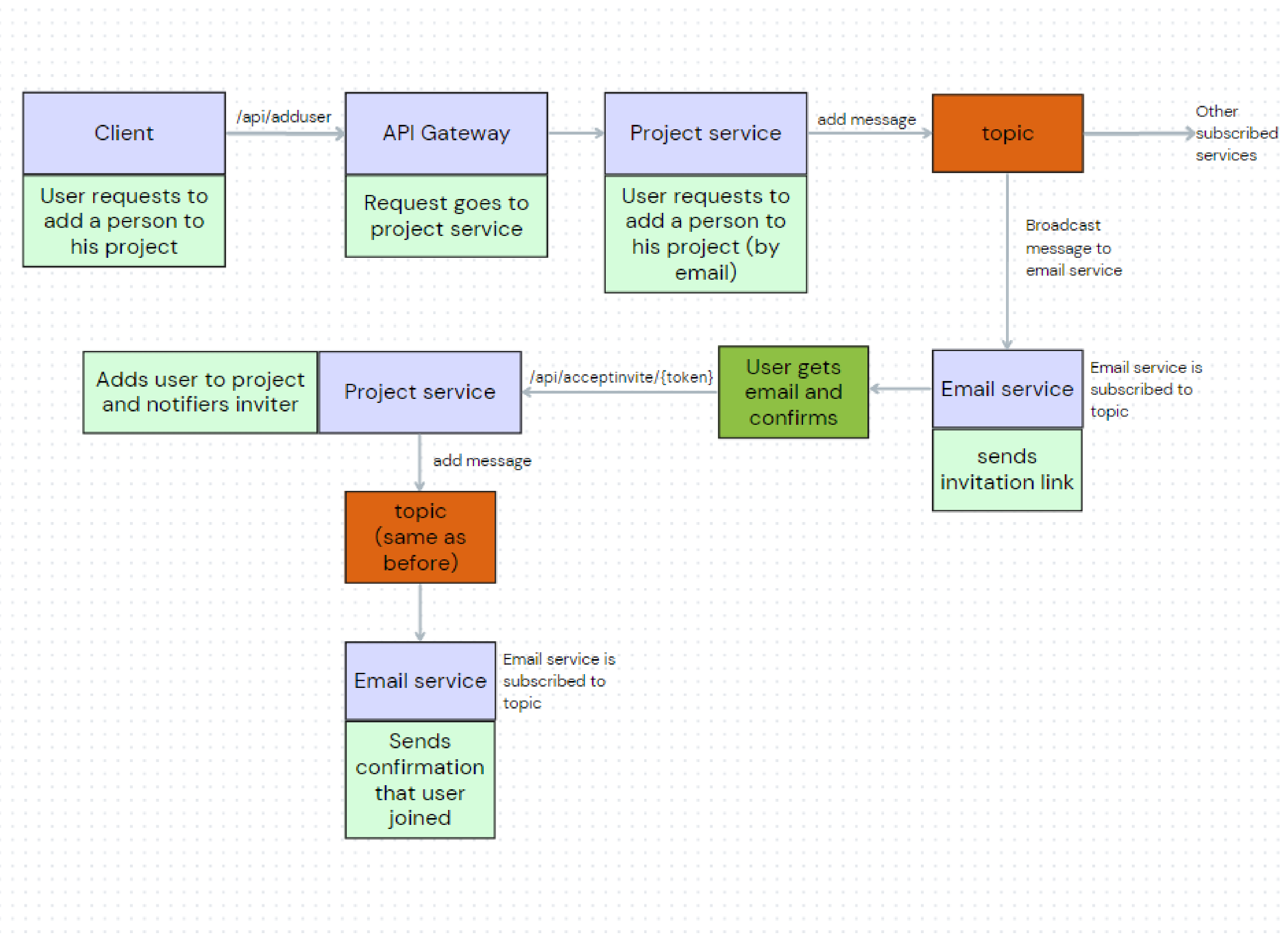
Notifications and emails are a task that can be done without the necessity of an immediate response from the server.

Messaging is a useful approach for implementing notifications and emails in enterprise software due to its asynchronous nature. In traditional email and notification systems, the sender must wait for a response before continuing with their workflow. Messaging, on the other hand, allows the sender to send the message and continue with their workflow without waiting for a response. This can greatly improve the responsiveness and scalability of enterprise software systems, especially in cases where high volume data and events are involved.

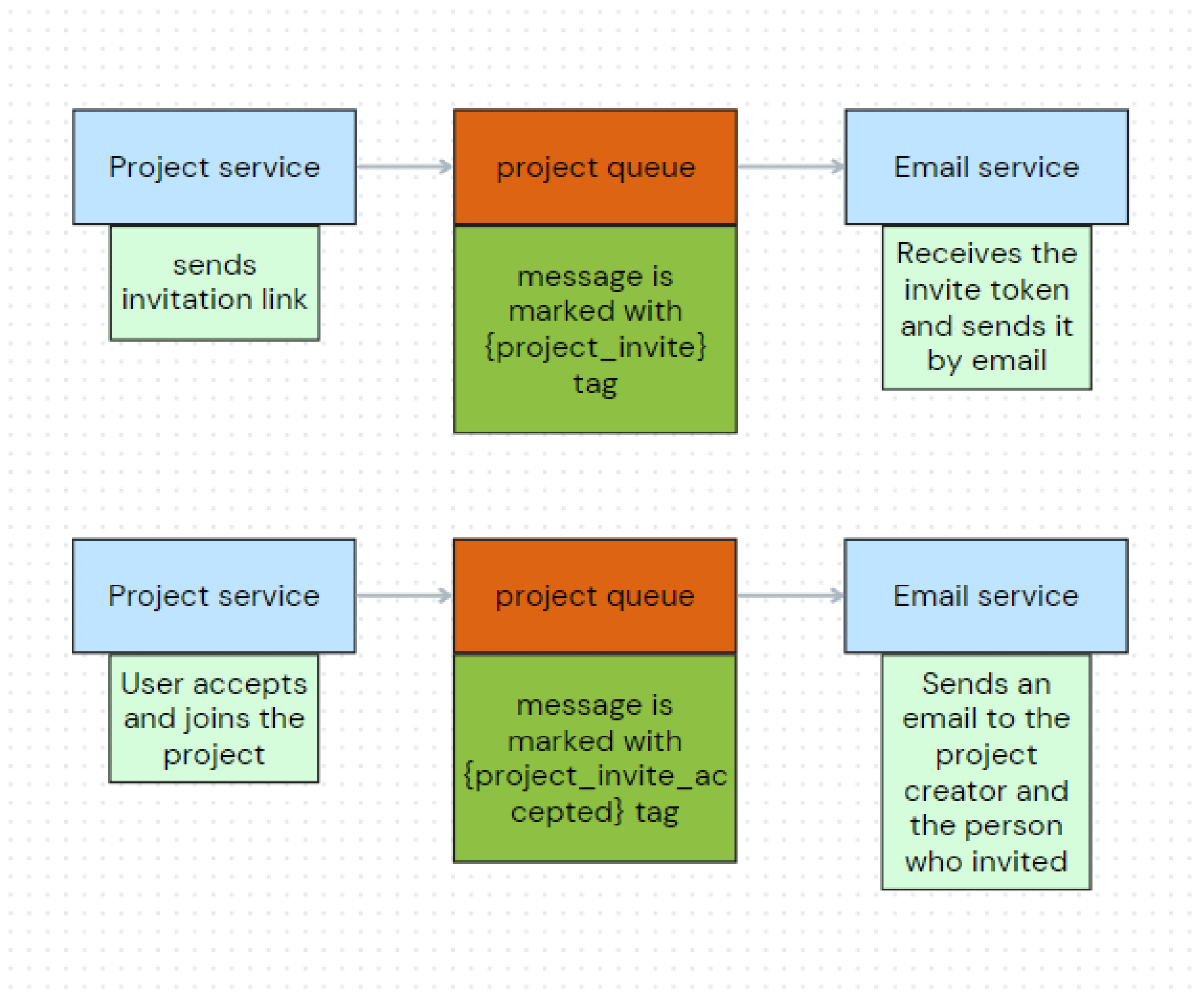
Messaging also provides a flexible and reliable communication channel between different parts of the enterprise software system. For example, when a user triggers an event that requires sending a notification or email, messaging can be used to reliably and securely deliver that message to the appropriate service or microservice responsible for handling the notification or email. This decoupling of services allows for greater flexibility and adaptability in the software architecture, making it easier to modify or add new services as needed.

In order to better understand the use case of messaging communication within the notification and email systems in the project, the following diagram shows the flow of how the project invitation process looks like.

Project invitation functional flow


Client
User requests to add a person to his project
/api/adduser
API Gateway
Request goes to project service
Project service
User requests to add a person to his project (by email)
add message
topic
Other subscribed services
Broadcast message to email service
Email service
Email service is subscribed to topic
sends invitation link
User gets email and confirms
/api/acceptinvite/{token}
Project service
Adds user to project and notifiers inviter
add message
topic (same as before)
Email service
Email service is subscribed to topic
Sends confirmation that user joined

For testing purposes, the current system utilises regular one directional message queues instead of message topics. As the messaging in the project is delegated to a cloud provider (Azure), the operational costs are unnecessary for the testing phase. The diagram below shows in detail how this is substituted with regular queues.

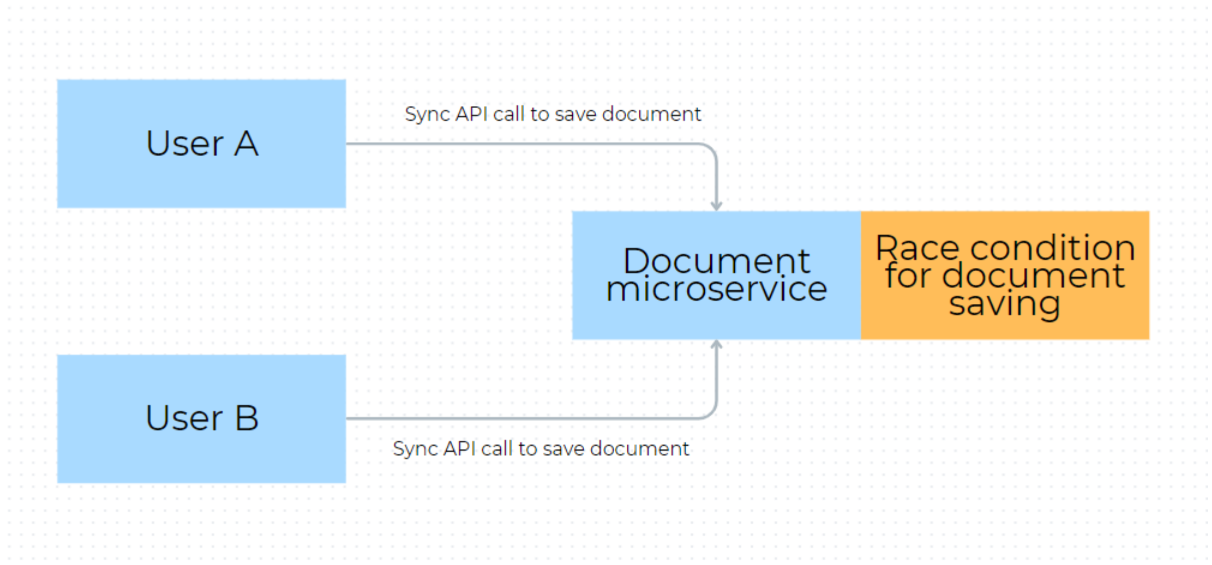

Project service
sends invitation link
project queue
message is marked with {project_invite} tag
Email service
Receives the invite token and sends it by email
Project service
User accepts and joins the project
project queue
message is marked with {project_invite_accepted} tag
Email service
Sends an email to the project creator and the person who invited

Document saving

As copycloud is an online document collaboration tool, saving of documents must be done in a secure and seamless way.

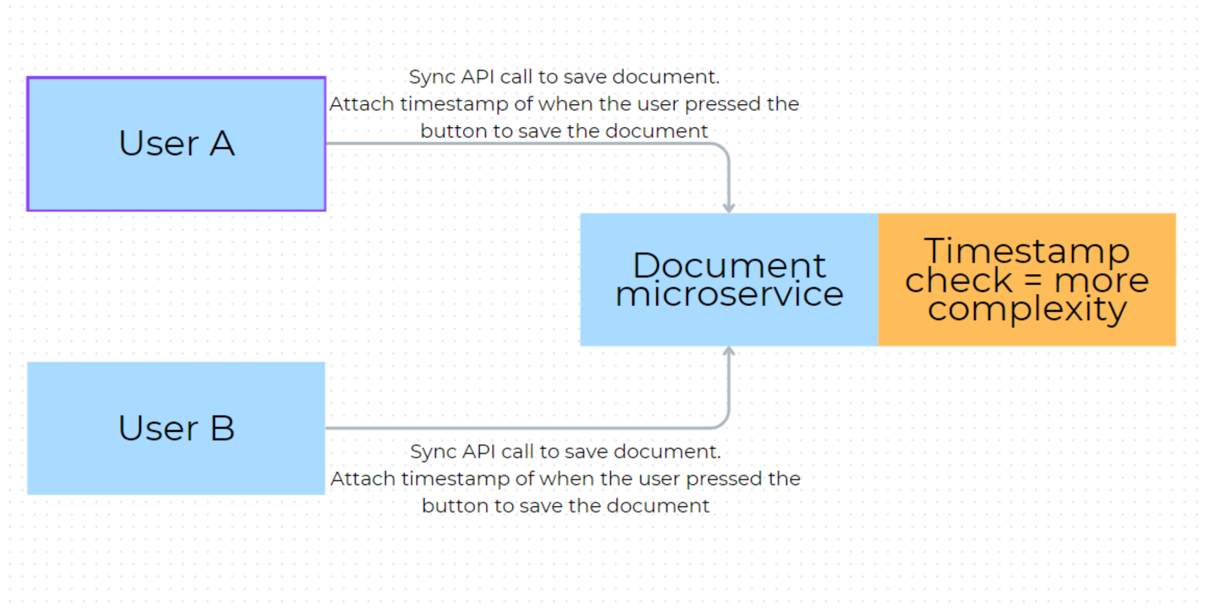
In order to define a proper strategy of how documents must be saved in the most secure and user friendly way, we first go over some of the pitfalls of the development of such a system.

Example 1 - Race condition


User A
Sync API call to save document
Document microservice
Race condition for document saving
User B
Sync API call to save document

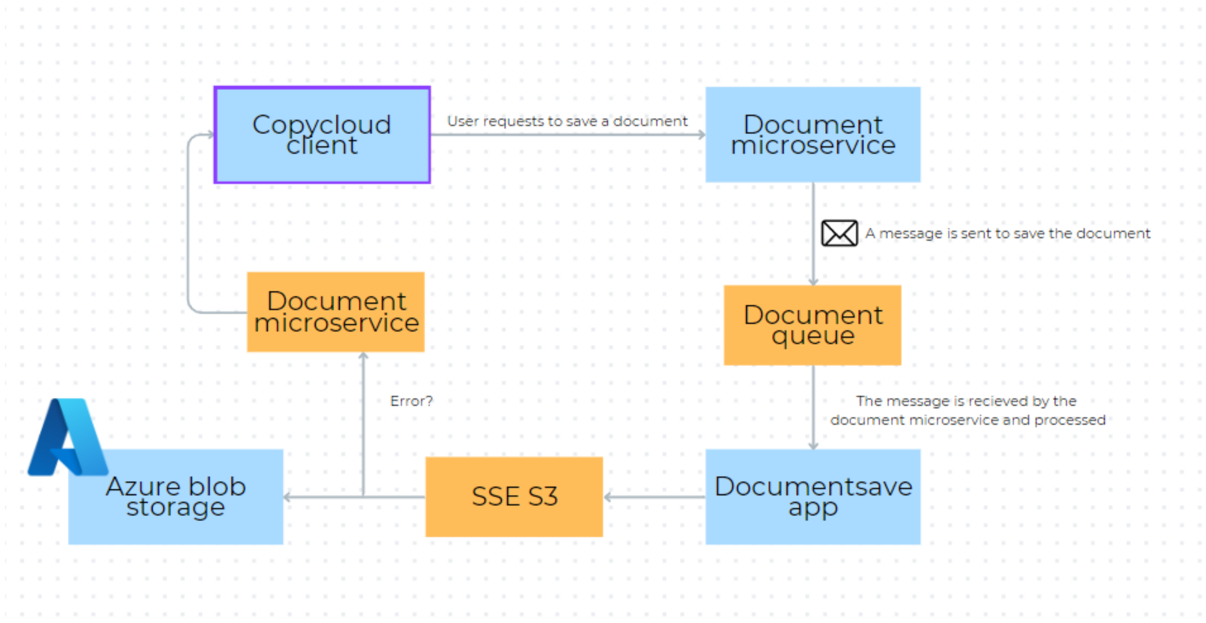
In the above diagram, we can see a prime example of why using simple synchronous call is a bad practice here. In this scenario, it is possible that two users request to save a document at the same time, however since there is no mechanism of checking who saved it first, the document is saved by the user with better bandwidth.

Example 2 - Timestamp check


User A
Sync API call to save document.
Attach timestamp of when the user pressed the button to save the document
Document microservice
Timestamp check = more complexity
User B
Sync API call to save document.
Attach timestamp of when the user pressed the button to save the document

The first viable solution of how this can be dealt with is by attaching a timestamp variable to each api call. This way, the system can check if there are any other calls made and filter out which should be executed first. This, however, increases the complexity of saving by a large factor and therefore is undesirable.

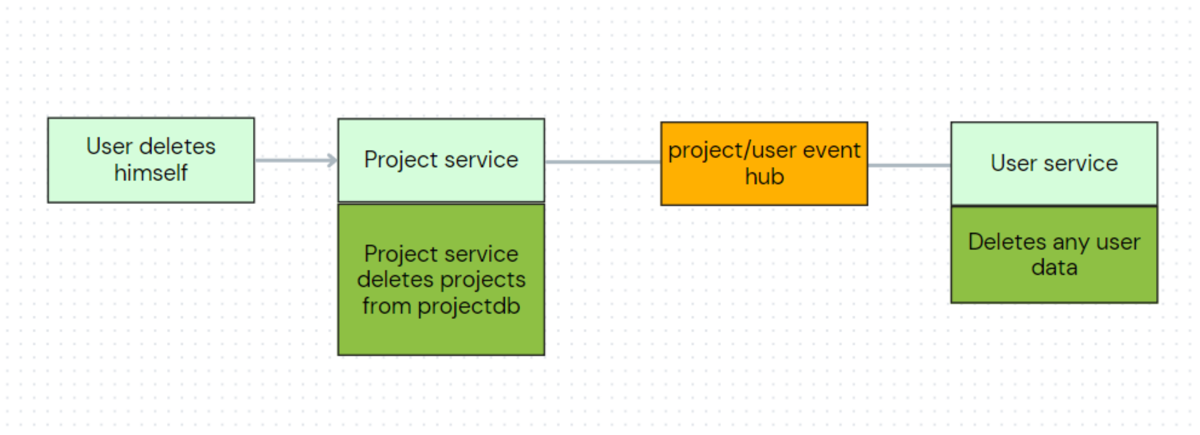
In order to correctly save documents, we need to introduce a system that is both fast, but also ensures that data is preserved and the document is saved in the correct order of execution. To achieve this, we can refer to the following diagram that provides a high overview of such a system. Copycloud uses an event based approach, where a message is published and consumed by the same service, making it impossible to save data incorrectly.


Copycloud client
User requests to save a document
Document microservice
A message is sent to save the document
Document queue
The message is recieved by the document microservice and processed
Documentsave app
SSE S3
Error?
Document microservice
Azure blob storage

We can see in the diagram above that Copycloud uses a queue that provides an entry point for messages which contain the changes in a given document. Furthermore, a background application running within the same microservice is constantly checking for such messages and consuming them in the order that they have been published in.

Since publishing a message is still done by a synchronous call from the user, there is a slight possibility that the above shown bad effect of race condition might be achieved. To deal with this, messages will have a timestamp attached to them of when the user actually requested to save a document, further ensuring that it will be saved in the correct order.

Use case of user removing his account



In the above shown diagram, we see the case of a user deciding to stop using Copycloud/violating policies and getting his account removed. As per GDPR rules, any data associated with a user that leaves/removes his account must be deleted alongside. Some exceptions of this involve keeping reference/identifiers of bank transfers and other 3rd party legalities.

The above flow shows an event based architecture where the user makes a request for his account to be deleted. This is sent to an event hub that is between the project and user service. The project service deletes any projects the user has created and removes him from any external projects he's joined. Finally, the user service removes any data associated with that user.