Beep architectural document:: Micro-services, here we come !

Table of Contents

1. Introduction and Goals

Beep is an open source project lead by Polytech DevOps students of 2023-2026 at the University of Montpellier. It is an alternative for Discord, a popular communication platform made for communities.

We started this project in late 2023 as a way to learn more about the software development process and to get familiar with the tools we will use in the future in a bigger project context.

In short, the main features of Beep are:

- An efficient platform for the creation of communities providing advanced configuration options for permissions, styles and more.
- A fast backend allowing to handle a large number of users and messages.
- Secure authentication using multi-factor authentication.
- Social features such as friendships and private messages.
- A search engine allowing to discover communities and within communities to find users and messages.

As of today, Beep was architected as a monolithic application, but we are currently having trouble with the scalability of the application and the codebase is becoming too complex. We are therefore planning to split the Beep application into multiple microservices, each one responsible for a specific function. This will allow us to scale the application more easily and improve the maintainability of the codebase.

This design document provides an overview of how we are going to break Beep's monolithic codebase into smaller, more manageable components while improving the **security** and **resiliancy** of the platform.

To not rush the migration, we chose to forbid usage of technologies like messages queues, CQRS or event sourcing. Thus we will try to keep Beep as synchronous as possible in order to keep the "ACIDity" of the infrastructure.

1.1. Requirements Overview

Before starting describing the technical details of Beep, let's first define in depth the features provided by Beep.

1.1.1. Functional Requirements

When defining the functional requirements, eight business domains were identified, in the followin section we will describe the features defining each of these domains.

1. Authentication

Definition: A user is a person that uses Beep. Thus, a visitor can not be considered as a user since he is not registered on the platform.

- As a visitor I can sign up to beep. I will provide my first name, last name, username, email, password and a picture of me.
- Then I will be able to sign in by providing my email and password.
- Eventually I will be able to activate the 2FA, so when I sign into beep I will be asked to enter 2FA code.
- I also want to be able to log out so I can switch account. This part we will be detailled further on throught the second question since most of these feature are answered thanks to an OIDC.
- As a Montpellier university student I want to be able to log in to Beep thanks to my university credentials through the LDAP access.
- AS a visitor I want to be able to join Beep through Google authentication.

2. Servers

Definition: A server is an entity that can host communities. It groups many users and messages in a single place and can be organized according to community needs.

- As a user I want to be able to **discover public servers** thanks to a **search feature**.
- Also I want to be able to join a **public server** so I can join a community.
- Also I want to be able to join a **private server** thanks to either an **invitation link** OR an in-app invitation (not yet implemented).
- Once in a server I want to be able to **see channels**, and **messages**, also I want to be able to **join voice calls**
- As a user I want to create a server either **public** or **private** depending on the level of visibility I want. I want to be able to name my server to make it unique or at least recognizable.
- As a **server admin**, I want to be able to add a name to the server, a picture, a banner and a description so my server becomes unique !
- As a **server admin** I want to be able to manage everything a user can do **within** my server thanks to a role system. It means that every resources → **message** and **channels** are impacted by these roles.
- As a **server admin** I want to be able to destroy my server.
- As a **server member** I want to be able to invite my friends to the server if its public. If it’s private, I need to be an admin.

3. Channels

Definition: A channel is an entity that can be used to send messages to a group of users. A user "subscribes" to a channel to receive messages from it and can also send messages to it so they are broadcasted to the member of the channel.

- As a **server admin** I want to be able to create channels within a server.
- Still as a **server admin** I want to be able to create **text channels** or **voice channels** to either send messages or discuss directly with my friends in vocal.
- Things can get messy, so as a **server admin** I a want a system of **folder channels** to regroup

text channels and voice channels.

- As a **channel admin** I want to be able to edit its name.
- As a **folder channel admin** I want to be able to edit its name.
- As a **channel admin** I want to set who can see the channel, thus introducing **private channels**.
- As a **channel admin** I want to be able to delete a channel and if it's a folder channel, all its subsequent channels.
- As a **channel member** I want to be able to see who are the other members and if they are connected in real time.

4. User

- As a user I want to be able to **choose the language** of beep. For now either **english** or **french**.
- I want to be able to select my **audio inputs** and **video inputs** for video calls.
- I want to be able to change my **name, last name, username, email and profile picture**.
- I want to be able to change my **password**.

5. Voice calls

- I want to be able to join voice calls to chat with other beep users.
- I want to be able to see who's in the voice call.
- I want to be able to see my friends cameras and share my camera only if I want it.
- I want to be able to share my screen to the people in the call.
- I want to be able to mute myself when in a voice call.
- I want to be able to leave a voice call.

6. Messages

- I want to send a text message to other users.
- I want to be able to **delete** my text messages. Or if I have enough right, delete other people messages
- I want to be able to edit my text messages.
- I want to be able to pin the current message to the channel where the conversation is to retrieve them later.
- I want to be able to answer to any messages to keep a conversation.
- I want to be able to see who sent a message and when.
- I want my text messages to support markdown and youtube, spotify, twitter integration to preview links.
- I want to be able to join files to my messages.
- I want to be able to ping other users that are on the channel/server in a message by typing @<username\>

- I want to be able to tag a channel with a \#<channel-name\>
- I want to be notified when I'm pinged on a channel.
- I want to be notified when receiving a message but with a level not as high as when I'm pinged.
- I want to be able to snooze these notifications
- I want to be able to reply to a message.
- I want to be able to search for messages within channels.

7. Friends

- I want to be able to discuss directly to other users thanks to a direct message system.
- To manage my inbox, I want to be able to ask other users to be my friend
- When someone asks me to be his friend, obviously I want to see the ask and see the asker identity
- I want to deny or accept a friend ask.
- I want to be able to remove friends so we are not friend anymore. This will automatically erase our conversation.
- I want to be notified when receiving a friend ask.
- I want to be able to snooze these notifications.

8. Automations

- Message can be sent automatically to a channel thanks to webhooks
- A user in a **channel** with the correct authorizations can create a webhook to automatically send messages to a channel
- A webhook owner can delete it
- A webhook owner can edit it

1.1.2. Technical Requirements

Now that we have defined the functional requirements, let's define the technical requirements by determining the amount of data we will need to manage.

We are targeting users who are European students and are above 18 years old. We can assume that they either use a smartphone or a laptop. In general, they send 100 messages per day.

In Europe in 2022, there were 18.2 million European students. If we reach 1% of that population, we could consider Beep a successful project. This means that we will deal with 182,000 messages per day. If a message is on average 100 characters, then we will have to store at least 18.2 million bytes of data, which is 0.0182 terabytes of data per day.

This estimate does not include files or voice calls, but we can assume that taking them into account will multiply by a factor of 2 the amount of data we will need to store, since voice calls are only in transit data.

Thus, per day, we will need to store 36.4 gigabytes of data. This is a lot of data considering that there is no budget for servers and that we will need to store it on servers that were lent to us by Polytech Montpellier.

In terms of in-transit data, we will need to handle 100 messages per second. If we imagine that there are constantly two people talking in a voice call, then we will need to manage 100 messages per second. This means that we will need to handle 100 gigabytes of data per second. If we refer to this WebRTC demo, we can estimate that the bandwidth taken by such a call is 1 Mbps. With a good compression algorithm, we can lower that to 200 Kbps. Thus, Beep will need an ingress bandwidth of 100.2 Gbps.

1.2. Constraints

Now that we have defined the functional requirements and the technical requirements, let's define our system constrants. As described before, we have at our disposal nine physical servers each with the same specifications :

- 2 CPUs
- 64Gb RAM
- Disks :
  - 1 X 256GB SSD
  - 2 X 1.2TB HDD

These servers are hosted by Polytech Montpellier, a French public university. Thus we are bounded by the network limitations and policies of Polytech Montpellier. There are only a subset of UDP ports that can be used by Beep. Thankfully, these ports are the same as Zoom's ports.

1.3. Quality Goals

We want Beep to be available to all users in Europe 95% of the time with a maximum response time of 500ms. However, based on the estimates that we made in the previous section, we won't be able to handle that ammount of money with the infrastructure that we have at our disposal. Thus we will set the max ammount of storage to 23 TB which corresponds to the total amount of space that can provide our infrastructure.

Thus user data retention will be limited to 15 TB. We have to keep some space for logs retention for example.

Roughly, our platform can store the data of **10 000 very active users** (meaning users that send 100 messages per day) which is approximately as much as the French Startup Alan.

Regarding security, we must first define our infrastructure before establishing a threat model. Initially, we can affirm that user personal data will not be disclosed. External actors will be unable to access our infrastructure or data storage. Additionally, we aim to prevent users from creating an excessive number of servers or other entities. To ensure a smooth and secure transition to microservices, it is crucial to avoid several common pitfalls. We must ensure that each microservice is properly isolated to prevent issues in one service from affecting others, avoiding tight coupling

that can lead to cascading failures. Caution must be exercised with data sharding to maintain data consistency and integrity across services, steering clear of inconsistent data handling practices that can result in data corruption or loss. Comprehensive logging, tracing, and monitoring should be implemented from the outset, as diagnosing issues in a distributed system without proper observability tools can be extremely challenging. Security measures such as encryption in transit, role-based access control, and secure authentication mechanisms must be in place from the beginning, avoiding the assumption that security can be addressed later. Load balancing should be managed carefully to distribute the load evenly across services, preventing any single service from becoming a bottleneck and ensuring optimal performance and user experience. Automated testing for individual services and their interactions should be implemented to avoid error-prone and time-consuming manual testing processes. By addressing these concerns proactively, we can enhance the security, scalability, and maintainability of our microservices architecture.

1.4. Stakeholders

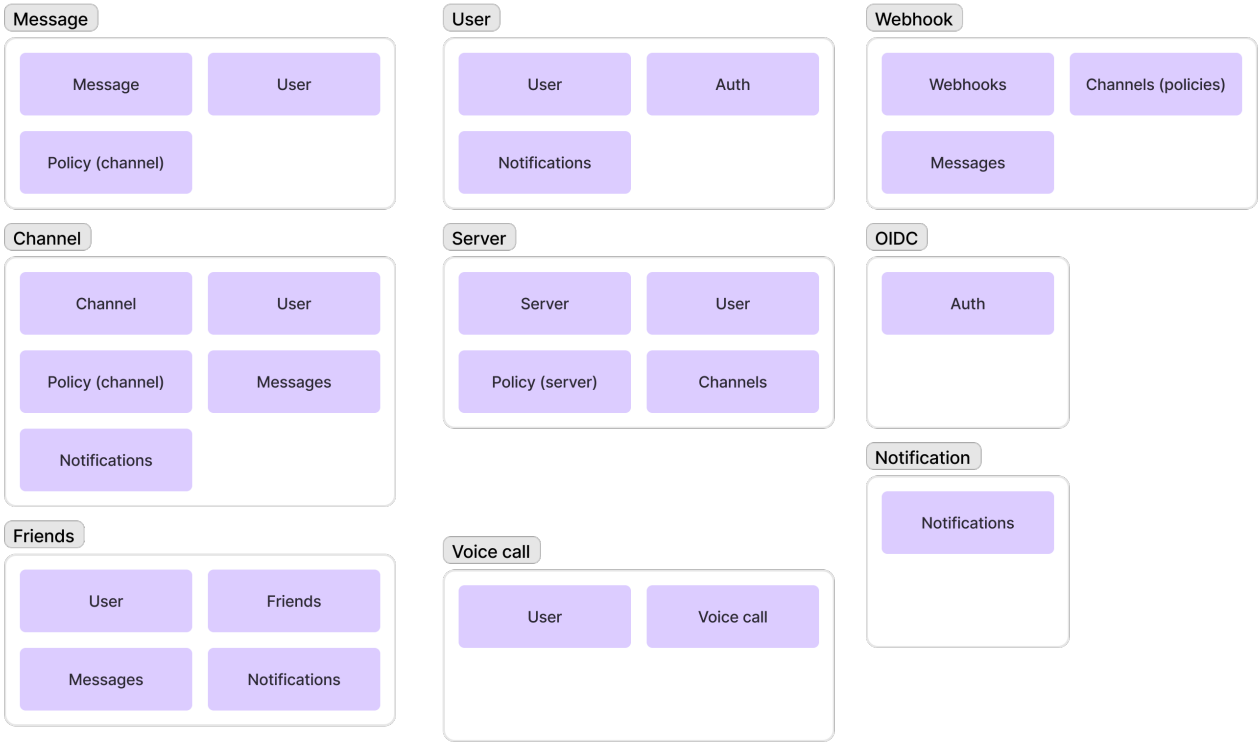
Role/Name	Contact	Expectations
Product Owner	*github.com/LeoFVO*	*In charge of the product and its development*
Documentation and Architecture consultant	*github.com/MonstyFred*	*Responsible for the documentation and architecture*

2. Architecture Constraints

In this section we will define the architecture constraints. In other words, we will define how we split the different domains of Beep into different microservices. What patterns were used to do that and finally which programming tools, languages and protocols must be used.

2.1. Breaking down the Beep monolith

As defined in the Introduction and Goals, we defined nine domains in Beep.



Each of these domains have their own data model and business rules. However some of them share the same technical constraints. We will go other every of these domains explicitly define these constraints so we can better pick the right micro-service definition.

2.1.1. 1. Authentication

The authentication domain needs to be extended to handle multiple authentications methods while keeping security features that we implemented previously, such as multi-factor authentication. We will be using OAuth2 for the authentication part and we will need to implement a JWT token to handle the authentication. JWT help us keeping the infrastructure stateless since by design it is a distributed proof of identity. We will be using Keycloak as an implementation of OAuth2. Also we will be handling several identity provides : * Montpellier University LDAP * Google

2.1.2. 2. Messages

The message domain is the one in charge of handling the messages. Messages are really simple since they are just a text with a timestamp a writer and a channel. They can be stored in any kind of database, however they need to be retrieved based on their channel ID. Actually, a message need to

belong to a user and a channel.

```
classDiagram
  User "1" --> "*" Channel : member
  Channel "1" --> "*" Message : has
  Message "1" --> "1" User : has
  class User {
    id: string
    username: string
    email: string
  }
  class Channel {
    id: string
    name: string
    description: string
  }
  class Message {
    id: string
    text: string
    timestamp: string
    owner: User
    channel: Channel
  }
```

Also messages need to be retrieved based on their content, their owner name, etc. Thus we will need to use a search engine to handle this kind of queries. Each queries are scoped to the channel. Messages must be sent in real-time thus the service that will handle this domain must include real time exchange protocols such as gRPC, WebSocket, SSE…

2.1.3. 3. Channels

Channels are an important part of Beep. A user subscribes to a channel by joining it. Then he can send messages to the channel. Thus a channel is a communication channel between users. They are the context in which messages are exchanged thus a message **need** to be in a channel. There are two types of channels :

- Text messages channels : these channels are used to send text messages to a group of users.
- Voice calls channels : these channels are used to perform real time voice calls between users.
- Folder channels : these channels are used to group text channels and voice channels. They don't contain messages per se but they are used to regroup channels.

A channel contains an ID, a description, a name, a list of users that are members of the channel and a list of messages that are sent in the channel. The members of a channel have a set of customizable roles/permissions.

2.1.4. 4. Servers

Servers are the entities that can host communities. They are the equivalent of a server in a chat

application. A server can host several channels. If we keep it simple, a server is an aggregation of channels. Each servers have members and just like channels, servers provide a system of authorization.

They have an ID, a name, a description, a list of users that are members of the server and a list of channels that are hosted by the server. The members of a server have a set of customizable roles/permissions.

2.1.5. 5. Users

Users are the entities that can send messages to channels and servers. They are the equivalent of a user in a chat application. Each user has a username, a first name, a last name, an email, a profile picture, a banner, a status, a connection status and a biography.

```
classDiagram
    UserIdentity "1" -- "1" OIDCIdentity
    OIDCIdentity --* Provider
    note for UserIdentity "Stored in the user service"
    note for OIDCIdentity "Stored in keycloak, about the userIdentityIdentifier, it
must be unique"
    class UserIdentity {
        + firstName : string
        + lastName : string
        + email : string
        + userHandle : string
        + identifier : string
        + profilePicture : string
        + banner : string
        + status : string
        + connectionStatus : boolean
        + biography : string
    }

    class OIDCIdentity {
        + userIdentityIdentifier: string
        + firstName: string
        + provider: Provider
        + lastName: string
        + email: string
        + id: string
    }
    class Provider{
        <<enum>>
        LDAP,VANILLA,GOOGLE
    }
```

2.1.6. 6. Voice calls

Voice calls are communications channels used to exchange audio and video data between users

leveraging the WebRTC protocol.

2.1.7. 7. Files

Files are a way to share documents with other users. They can be attached to messages but can also be profile pictures, banneers, etc.

2.1.8. 8. Automations

Automations are a way to send messages automatically to a channel. They are like fake users accessible through webhooks. A webhook is a method of augmenting or altering the behavior of a web page or web application with custom callbacks. These callbacks can be maintained, modified, and managed by third-party users and developers who may not necessarily be affiliated with the original website or application.

2.1.9. 9. Notifications

Notifications are a way to make a user aware of something that happened in the platform. They are used to notify users about new messages, new channels, new servers, etc. By design, notifications need to be in real time and asynchronous a little bit like messages but contrary to messages, they are highlighted in the application and they are not searchable through a search index.

2.2. Microservices

Based on the previous section, we can already see which domains share the same technical needs.

2.3. Channel & Servers microservice

First of all, we can already see that channels and servers share more or less the same data model and the same approach when it comes to permissions. Thus, they will be located in the same microservice. This will help making the search engine to discover servers because when looking for new servers we will be able to make aggregations based on channels.

```
architecture-beta
    group api(cloud)[ChannelServers]

    service db(database)[Database] in api
    service search(database)[SearchEngine] in api
    service auth(server)[Authorization] in api
    service server(server)[ChannelServersAPI] in api

    db:L -- B:server
    search:T -- R:server
    auth:R -- L:server
```

2.3.1. Connection pools, messages and notification microservices

Actually, messages and notifications share the same problem : they are both real time and asynchronous since users want to be notified while they are on the app without refreshing it, the same for messages. However notification can trigger mail send. So these two domains will share the same technologies but won't be in the same micro-services.

Messages

```
---
title: Message microservice
---
architecture-beta
    group api(cloud)[MessageService]

    service db(database)[Database] in api
    service search(database)[SearchEngine] in api
    service connection(server)[ConnectionPool] in api
    service message(server)[MessageAPI] in api

    db:B -- R:message
    search:T -- R:message
    connection:R -- L:message
```

Notifications

```
---
title: Notification microservice
---
architecture-beta
    group api(cloud)[NotificationService]

    service db(database)[Database] in api
    service mail(database)[MailServer] in api
    service connection(server)[ConnectionPool] in api
    service notification(server)[NotificationAPI] in api

    db:B -- R:notification
    mail:T -- R:notification
    connection:R -- L:notification
```

Connection pools

A connection pool is essentially a collection of bidirectional connections that services can utilize to establish connections. Unlike a pub/sub service, it is not designed to facilitate inter-service communications. While working on the implementation of a message server for my proof of concept (available on GitHub), I encountered challenges in creating a WebSocket server capable of broadcasting messages to all connected users. This highlighted the need for distributed

bidirectional servers to ensure scalability of the WebSocket server, a functionality adeptly provided by Phoenix Channels. Essentially, connection pools act as services that other services can connect to when they need to broadcast messages across various channels.

This concept is precisely utilized by the messages and notifications systems. The messages system broadcasts messages to specific channels, while the notifications system disseminates various types of notifications to groups of users. These user groups can be organized through channels defined by the notification service, enabling efficient and targeted communication.

2.3.2. Users and authentication microservices

There will be one microservice called users that will be in charge of both users and handling authentication strategies. This service will basically be a facade in front of keycloak. Will be in charge of storing identity user data.

Also this service will be in charge of the friendships between users. This service will be in charge of storing friendships data but also of handling the friendship requests and private messages. This is done to keep the friends ACID.

```
---
title: Users microservice
---
architecture-beta
    group api(cloud)[UsersService]

    service auth(server)[Keycloak] in api
    service database(database)[Database] in api
    service users(server)[UsersAPI] in api

    auth:R -- L:users
    database:L -- R:users
```

2.3.3. Automations and webhooks microservice

Since webhooks are a special case were you need to save the callback URL, associate it to a channel, etc, we will need to have a dedicated microservice for webhooks. Also in the future we can easily imagine beep featuring bots a little bit like Discord do.

```
---
title: Automations and webhooks microservice
---
architecture-beta
    group api(cloud)[AutomationsWebhooksService]

    service database(database)[Database] in api
    service webhooks(server)[WebhooksAPI] in api
```

```
    database:L -- R:webhooks
```

2.3.4. Voice calls

This microservice will be in charge of managing WebRTC connections and media streams. It will be written in Phoenix to handle the signalling and will use Stunner to handle STUN/TURN network protocols. For more informations please refer to Mathias's talk about WebRTC history in Beep.

2.3.5. Files

In the development of a scalable and efficient microservices architecture, the integration of MinIO for handling file storage presents a robust solution, particularly when interfaced through a dedicated file microservice. This file microservice, developed in Go, acts as a facade for MinIO, abstracting the complexities of direct interactions with the storage layer and providing a streamlined API for other services to consume. By employing this intermediary layer, both the user microservice and the message microservice can securely and efficiently manage file operations without being exposed to the underlying storage mechanics.

The file microservice ensures that all access to MinIO is authenticated and authorized, thereby enforcing security protocols and safeguarding user data. When a user needs to access files, the user microservice interacts with the file microservice, which verifies the user's authentication credentials before proceeding with the request. Similarly, the message microservice can leverage this file microservice to store and retrieve file attachments associated with user messages, ensuring that all file operations are conducted within a secure and controlled environment.

Using MinIO as the backbone for file storage offers several advantages, including high scalability, durability, and compatibility with the Amazon S3 API, which simplifies integration and migration efforts. The Go-based facade not only enhances performance through efficient handling of concurrent requests but also provides a clear and maintainable codebase. This architecture promotes a separation of concerns, where each microservice can focus on its core functionality while delegating file storage responsibilities to a specialized service, thereby fostering a modular and maintainable system design.

```
---
title: File microservice
---
architecture-beta
    group api(cloud)[FileService]

    service minio(server)[MinIO] in api
    service file(server)[FileAPI] in api

    minio:R -- L:file
```

2.3.6. Gateway

To ease the front-end integration, we will need to have a gateway. We will be using GraphQL to

handle this part and map each domain to the right microservice. GraphQL through its subscription system makes it possible to push real-time updates to clients, which is essential for real-time communications. This approach also allows for efficient caching and data retrieval, improving the overall performance of the application.

```
---
title: Gateway
---
architecture-beta
    group api(cloud)[Gateway]

    service users(server)[UsersService] in api
    service channels(server)[ChannelsService] in api
    service servers(server)[ServersService] in api
    service messages(server)[MessagesService] in api
    service notifications(server)[NotificationsService] in api
    service voicecalls(server)[VoiceCallsService] in api
    service files(server)[FilesService] in api
    service automations(server)[AutomationsService] in api
    service webhooks(server)[WebhooksService] in api
    service gateway(server)[GatewayAPI] in api

    users:R -- L:gateway
    channels:R -- L:gateway
    servers:R -- L:gateway
    messages:R -- L:gateway
    notifications:R -- L:gateway
    voicecalls:R -- L:gateway
    files:R -- L:gateway
    automations:R -- L:gateway
    webhooks:R -- L:gateway
```

2.4. Technological stack

2.4.1. Languages and frameworks

We will be using the following languages and frameworks:

- Go : to implement the codebase of the services with the help of the Optique framework. Optique is a modular framework that allows developers to build scalable and maintainable microservices with ease. It provides a set of tools and libraries that simplify the development process, including a powerful dependency injection system, a pluggable middleware system, and a flexible configuration system. The framework also includes a built-in web server, which makes it easy to develop and deploy microservices.
- Typescript : to implement the front-end of the application with the help of TypeScript. TypeScript is a superset of JavaScript that adds optional static typing to the language. It is designed to be a better JavaScript language that scales with modern web development. TypeScript helps developers catch errors early and write more reliable code.

- Elixir : to implement the WebRTC server with the help of Membrane and WebRTC Engine. Membrane is a framework for building scalable and fault-tolerant applications. It provides a set of tools and libraries that simplify the development process, including a powerful dependency injection system, a pluggable middleware system, and a flexible configuration system. The framework also includes a built-in web server, which makes it easy to develop and deploy microservices.

2.4.2. Inter-communication protocols

In a distributed context, managing communication between microservices involves several key considerations. One of the first decisions to make is the protocol that will facilitate this communication. There are two primary options: REST API and gRPC. Each has its own advantages and trade-offs, and the choice depends on the specific requirements and constraints of the system.

REST API

The REST API protocol is widely used for communication between microservices using HTTP requests. It leverages standard HTTP verbs such as GET, POST, PUT, and DELETE to define the type of request and the expected response. One of the significant advantages of REST is its simplicity and ubiquity. Most developers are familiar with HTTP and REST, making it easier to implement and maintain. Additionally, REST is stateless, which aligns well with the stateless nature of microservices.

Another benefit of REST is the ability to generate documentation and type-safe clients based on an API schema using OpenAPIv3. This can significantly improve developer productivity and ensure consistency across different services. Tools like Swagger can be used to generate interactive API documentation, which can be exposed via an endpoint such as `kubernetes.local/<my-service>/docs`. This documentation serves as a contract between services, clearly defining the expected inputs and outputs for each API endpoint.

However, REST was not originally designed with type safety in mind. While tools like OpenAPIv3 and Swagger can help mitigate this, they add an additional layer of complexity. The contract between services is defined in a file called `docs/openapi.yaml`, which is used to generate both the documentation and the client. This approach ensures that all services adhere to the same contract, reducing the risk of mismatches and errors.

gRPC

gRPC is another protocol used for communication between microservices, leveraging the concept of Remote Procedure Calls (RPC). It is a type-safe and battle-tested protocol that has been widely adopted in the industry. gRPC uses Protocol Buffers (protobufs) as its interface definition language (IDL), which allows for efficient serialization and deserialization of data. This makes gRPC particularly suitable for high-performance and low-latency applications.

One of the key advantages of gRPC is its type safety. The protocol ensures that the data types and structures are consistent across services, reducing the risk of errors and mismatches. Additionally, gRPC supports bi-directional streaming, which can be useful for real-time applications that require continuous data exchange.

gRPC also offers built-in support for code generation. The protocol definitions can be used to generate client and server code in multiple languages, ensuring consistency and reducing the amount of boilerplate code. This can significantly improve developer productivity and ensure that all services adhere to the same contract.

Choosing Between REST and gRPC

While both REST and gRPC have their advantages, the choice between the two depends on the specific requirements and constraints of the system. REST is generally simpler and more widely adopted, making it a good choice for smaller projects or systems with simpler requirements. However, gRPC offers more advanced features and is better suited for larger, more complex systems that require high performance and low latency.

So we will choose gRPC for the Beep project. Also the gRPC integration with Golang and Elixir is seamless and easy to use.

Finally, the frontend will communication with the gateway through GraphQL thanks to the Apollo client. This will allow us to have a single source of truth for the data and make it easier to maintain and update, while keeping the communication typesafe.

2.4.3. Databases

We will be using PostgreSQL as our database. PostgreSQL is widely regarded as one of the most powerful and reliable open-source relational database management systems available today. Its consistent top performance in various benchmarks underscores its efficiency and robustness, making it a preferred choice for developers and organizations alike. One of the key strengths of PostgreSQL is its extensive feature set, which includes support for complex queries, full-text search, JSONB for efficient storage and querying of JSON data, and advanced indexing techniques. These features enable PostgreSQL to handle a wide range of workloads, from simple web applications to complex data warehousing and analytics tasks.

Beyond its technical capabilities, PostgreSQL's widespread adoption and active community contribute significantly to its appeal. Being well-known and extensively documented, PostgreSQL offers a wealth of resources, tutorials, and third-party tools that make it easier to use and integrate into various projects. This extensive ecosystem not only simplifies the development process but also ensures that users can find support and solutions to common challenges more readily.

Moreover, PostgreSQL's adherence to SQL standards and its extensibility allow developers to customize and extend its functionality to meet specific needs. Whether it's through creating custom functions, data types, or extensions, PostgreSQL provides the flexibility to tailor the database environment to unique requirements. This combination of performance, flexibility, and community support makes PostgreSQL an outstanding choice for a wide array of database applications.

2.4.4. Search engines

There were a bunc of search engines available for the Beep project. I had to choose between ElasticSearch, Solr and Quickwit. But I chose Quickwit because it was the simplest and the most lightweight. Quickwit is a search engine that uses a simple and intuitive interface to allow users to search for content. It is designed to be easy to use and understand, making it a great choice for

developers and non-technical users alike. Quickwit also offers a range of customization options, allowing users to tailor the search experience to their specific needs. This makes it a versatile and flexible solution for a wide range of applications.

Also there is a boilerplate integration of Quickwit with the optique framework.

2.4.5. Authorization system

For the authorization system, we will be using Permify because it is a simple and easy to use solution. Permify is an open-source authorization system that provides a simple and intuitive way to manage user permissions and roles. It is designed to be easy to use and understand, making it a great choice for developers and non-technical users alike. Permify also offers a range of customization options, allowing users to tailor the authorization experience to their specific needs. This makes it a versatile and flexible solution for a wide range of applications.

2.5. Logs management

In a distributed context, ensuring observability through logs and traces is crucial for maintaining the health and performance of microservices. To achieve this, logs must be standardized across the system. Standardization ensures that logs can be processed consistently, regardless of the service that emitted them. Each log should contain several key pieces of information: a level of importance (such as DEBUG, INFO, WARNING, or ERROR), a timestamp indicating when the log was emitted, a body containing the log message, the name of the service that created the log, and the ID of the container that issued the log. This information is essential for understanding when and in which context an issue occurred, facilitating quicker incident resolution.

Logs will be emitted by various sources. For example, there will be different types of access logs, including those coming from the load balancer, the service mesh, and the applications themselves. Each of these sources may have its own logging structure, but standardizing the format for application logging (which encompasses all logs produced by services coded by the development team) is a critical first step.

2.5.1. Types of Logs

Application Logs

Application logs are generated by the services developed by the development team. These logs should adhere to a standardized format to ensure consistency and ease of processing. The format for application logs could be defined as follows:

```
{
  "level": "INFO",
  "timestamp": "2023-10-01T12:34:56Z",
  "service": "UserService",
  "container_id": "abc123",
  "message": "User authentication successful"
}
```

This format includes the log level, timestamp, service name, container ID, and the log message. By adhering to this structure, logs can be easily parsed and analyzed, regardless of the service that generated them.

Access Logs

Access logs are crucial for tracking requests and responses within the system. They are typically generated by the load balancer, service mesh, and the applications themselves. Access logs should include information such as the request method, URL, response status, and response time. For example:

```
{
  "timestamp": "2023-10-01T12:34:56Z",
  "service": "LoadBalancer",
  "container_id": "def456",
  "request_method": "GET",
  "request_url": "/api/users",
  "response_status": 200,
  "response_time": 123
}
```

Access logs help in identifying performance bottlenecks, tracking user activity, and diagnosing issues related to request handling.

Audit Logs

Audit logs are essential for tracking changes and actions within the system, especially those related to security and compliance. These logs should include information such as the action performed, the user who performed it, and the timestamp. For example:

```
{
  "timestamp": "2023-10-01T12:34:56Z",
  "service": "AuthService",
  "container_id": "ghi789",
  "action": "USER_CREATED",
  "user": "john.doe",
  "details": "User john.doe created successfully"
}
```

Audit logs are crucial for compliance and security audits, as they provide a detailed record of actions performed within the system.

Error Logs

Error logs are specifically designed to capture and record errors and exceptions that occur within the system. These logs should include information such as the error message, stack trace, and any relevant contextual information. For example:

```
{
  "level": "ERROR",
  "timestamp": "2023-10-01T12:34:56Z",
  "service": "PaymentService",
  "container_id": "jkl012",
  "message": "Payment processing failed",
  "stack_trace": "java.lang.NullPointerException...",
  "context": "User ID: 12345, Transaction ID: 67890"
}
```

Error logs are essential for diagnosing and resolving issues quickly, as they provide detailed information about the errors that occur.

2.5.2. Scenario: Incident Resolution with Audit and Access Logs

Consider a scenario where users report issues with accessing a particular feature in the application. To diagnose the issue, the development team can use audit and access logs to trace the problem.

1. **Access Logs**: The team reviews the access logs to identify any patterns or anomalies in the requests. They notice that requests to a specific endpoint are failing with a 500 status code.
2. **Audit Logs**: The team then reviews the audit logs to see if there were any recent changes or actions that could have caused the issue. They discover that a recent configuration change was made to the service handling the failing endpoint.
3. **Error Logs**: Finally, the team reviews the error logs to get more details about the failures. They find that the errors are related to a null pointer exception in the code, which was introduced by the recent configuration change.

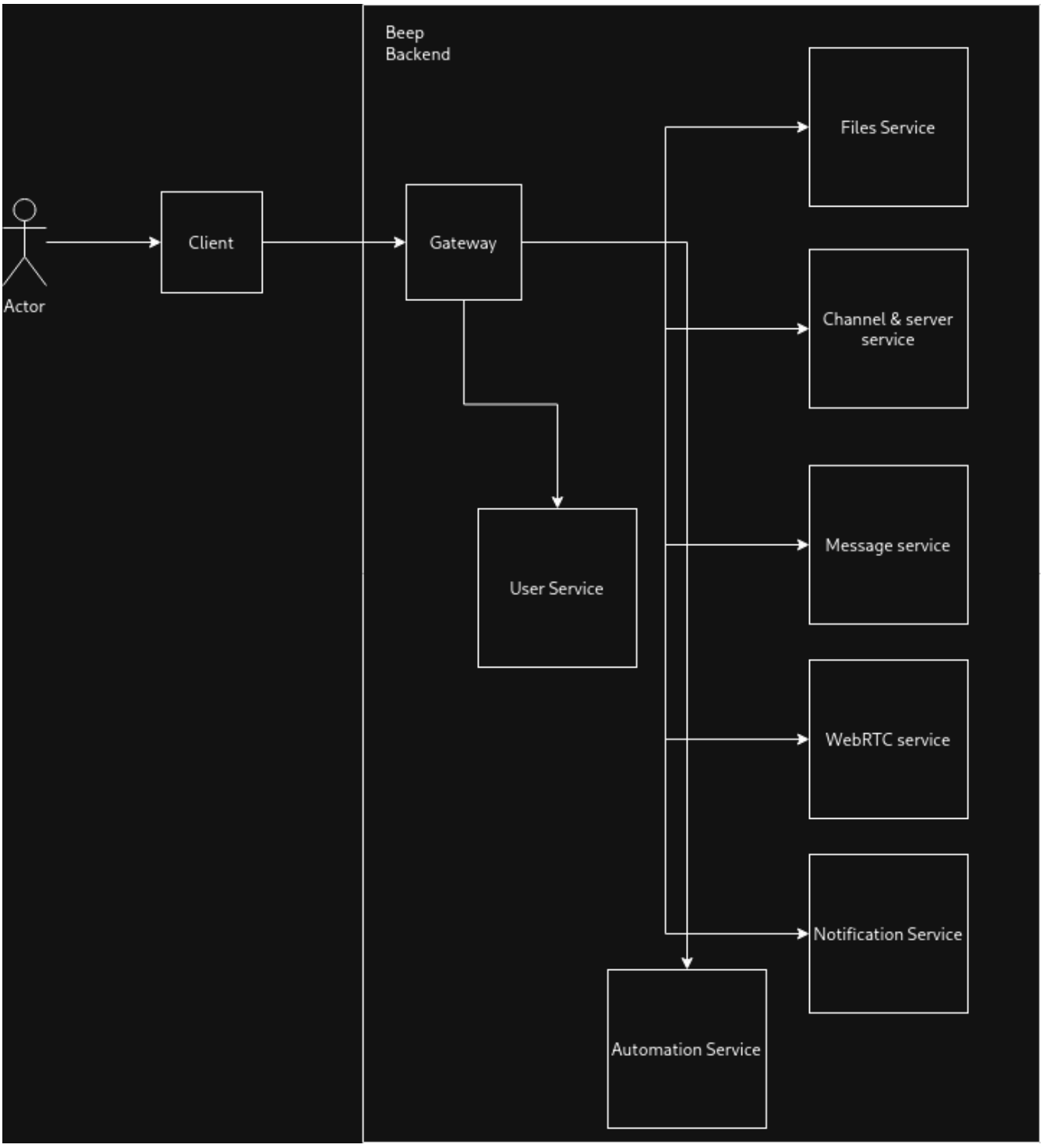
By correlating the information from access, audit, and error logs, the team can quickly identify the root cause of the issue and take corrective actions.

3. Building Block View

This section describes the interaction of each microservices with the rest of the system, and comprising the user.

3.1. Whitebox Beep system

<Overview Diagram>



4. Runtime View

In this section we will focus on the runtime view of the system. The runtime view describes the interactions of the building block instances with each other and their runtime environment. The figures below show the main interactions of our system at runtime.

4.1. Authentication runtime diagrams

4.1.1. User creates an account on beep - password flow

In this case, the user data is coming from a registration form in the frontend.

```
sequenceDiagram
    User->>Beep User service: POST /auth/register (user data)
    Beep User service->> Database: INSERT user
    Database-->>Beep User service: OK (user id)
    Beep User service->>Keycloak: POST /auth/realms/Beep/protocol/openid-connect/token
    Keycloak-->>Beep User service: 200 OK (access token)
    Beep User service->>Keycloak: POST create user {firstName, lastName, email,
userIdentifier}
    Keycloak->>Beep User service: 200 OK
    Beep User service->>User: 201 OK
```

4.1.2. User logs in to beep - password flow

```
sequenceDiagram
    User->>Beep User service: POST /auth/login (user data)
    Beep User service->> Database: SELECT user
    Database-->>Beep User service: OK (user)
    Beep User service->>Keycloak: POST /auth/realms/Beep/protocol/openid-connect/token
    Keycloak-->>Beep User service: 200 OK (access token)
    Beep User service->>User: 200 OK
```

4.1.3. User logs in to beep - LDAP flow

LDAP is a user federation protocol that allows to manage user identities from a centralized directory. The integration with keycloak consists in importing the user data from the LDAP directory into keycloak thus by default, every Polytech student will have an identity in Beep. Still, by default they won't be considered as fully registered since the *User service* won't register their identity. A user will only be created if they try to log into Beep with their polytech account.

Note that before hand, keycloak will contain the polytech user id and password since they will be imported from the LDAP thanks to LDAP integration.

```
sequenceDiagram
```

```
    User->>Keycloak: User tries to sign in
    Keycloak-->>User: 200 Ok (access token)
    User->>User service: GET any requests containing its access token
    User service->>Database: Checks if user identity exists
    Database-->>User service: No user doesn't exist
    alt IdentityProvider=="LDAP"
        User service->>Database: Create User identity
        Database-->>User service: OK (user id)
        User Service-->>User: 200 ok
    else
        User serivce->>User: 403
    end
```

As you can see on the diagram, if the IdentityProvider is LDAP, then it will create an identity in the user service.

4.1.4. User logs in to beep - social login flow

User creates beep account (google)

In this case, the user data is retrieved from google thanks to the openid connect protocol.

```
sequenceDiagram
    User->>Beep User service: POST /auth/register (user data)
    Beep User service->> Database: INSERT user
    Database-->>Beep User service: OK (user id)
    Beep User service->>Keycloak: POST /auth/realms/Beep/protocol/openid-connect/token
    Keycloack->>Google: POST /o/oauth2/v2/userinfo
    Google-->>Keycloak: 200 OK (user data)
    Keycloak-->>Beep User service: 200 OK (access token)
    Beep User service->>Keycloak: POST create user {firstName, lastName, email,
userIdentifier}
    Keycloak->>Beep User service: 200 OK
    Beep User service->>User: 201 OK
```

4.2. Server runtime diagrams

4.2.1. User creates a server

```
sequenceDiagram
    User->>Gateway: mutation{CreateServer(...){...}} (graphQL)
    Gateway->>ChannelServer: createServer(....) (gRPC)
    ChannelServer->>Database: INSERT server (TCP)
    Database-->>ChannelServer: RETURNING server
    ChannelServer->>Database: INSERT channel (TCP)
    Database->>ChannelServer: RETURNING channel (TCP)
    ChannelServer->>Quickwit: createServerIndex(...) (HTTP)
```

```
    ChannelServer->>Messages: createChannel(...) (gRPC)
    Messages->>QuickwitMessages: createChannelIndex(...) (HTTP)
```

Here we can see that the user creates a server by calling the `createServer` method of the `ChannelServer` service. This method creates a new server in the database and then calls the `createServerIndex` method of the `Quickwit` service to create an index for the server. The `createServerIndex` method sends a request to the Quickwit server to create an index for the server. The index is created by sending a POST request to the `/server` endpoint of the Quickwit server with the server data as the request body.

We need to create a new channel in the message service because the message service need to open a Phoenix channel to handle the messages of this channel.

4.2.2. User creates a new role

```
sequenceDiagram
    User->>Beep User service: POST /auth/login (user data)
    Beep User service->>Keycloak: POST /auth/realms/Beep/protocol/openid-connect/token
    Keycloak-->>Beep User service: 200 OK (access token)
    User->>Gateway: mutation{CreateRole(...){...}} (graphQL)
    Gateway->>Beep server service: createRole(...) (gRPC)
    Beep server service->>Permify: check if user is allowed
    Permify-->>Beep server service: OK
    Beep server service->>Permify: create role
    Permify-->>Beep server service: OK
    Beep server service->>Beep User service: 200 OK
    Beep server service->>User: 201 OK
```

The sequence diagram illustrates the authentication and role creation flow within the Beep application. Initially, a user sends a login request with their credentials to the Beep User service. The Beep User service then communicates with Keycloak, an identity and access management solution, to obtain an access token. Upon successful authentication, Keycloak returns an access token to the Beep User service.

Next, the user sends a GraphQL mutation request to the Gateway to create a new role. The Gateway forwards this request to the Beep Server service using gRPC. The Beep Server service then checks with Permify, an authorization service, to verify if the user has the necessary permissions to create a role. If the user is authorized, Permify responds affirmatively, and the Beep Server service proceeds to create the role in Permify.

After the role is successfully created, Permify confirms the action to the Beep Server service. The Beep Server service then notifies the Beep User service of the successful operation and returns a confirmation to the user, completing the process.

4.2.3. User sends a message in a channel in a server

```
sequenceDiagram
```

```
    User->>Gateway: mutation{SendMessage(...){...}} (graphQL)
    Gateway->>Keycloak: POST /auth/verify/user
    Keycloak-->>Gateway: 200 OK
    Gateway->>Beep server service: sendMessage(...) (gRPC)
    Beep server service->>Permify: check if user is allowed
    Permify-->>Beep server service: OK
    Beep server service->>Messages: sendMessage(...) (gRPC)
    Messages->>Database: INSERT message
    Messages->>QuickwitMessages: sendMessage(...) (HTTP)
    Messages->>Beep server service: 200 OK
    Messages->>Phoenix: broadcast message
    Beep server service->>Gateway: 200 OK
```

The sequence diagram outlines the process of sending a message within the Beep application. Initially, a user sends a GraphQL mutation request to the Gateway to send a message. The Gateway then communicates with Keycloak to verify the user's authentication by making a POST request to the /auth/verify/user endpoint. Upon successful verification, Keycloak responds with a 200 OK status, confirming the user's identity.

Following this, the Gateway forwards the send message request to the Beep Server service using gRPC. The Beep Server service then checks with Permify, the authorization service, to ensure the user has the necessary permissions to send the message. If the user is authorized, Permify responds affirmatively.

The Beep Server service then sends the message to the Messages service using gRPC. The Messages service inserts the message into the database and simultaneously sends the message to QuickwitMessages via an HTTP request for further processing or indexing.

5. Deployment View

Through this section we will see how the system looks like in deployment view. How the infrastructure is composed, what technologies are used to secure the system, and how we are keeping an eye on what's going on thanks to logging.

Just as a reminder, we have at our disposal the following hardware infrastructure, nine servers with the specification :

- 2 CPUs
- 64Gb RAM
- Disks :
  - 1 X 256GB SSD
  - 2 X 1.2TB HDD

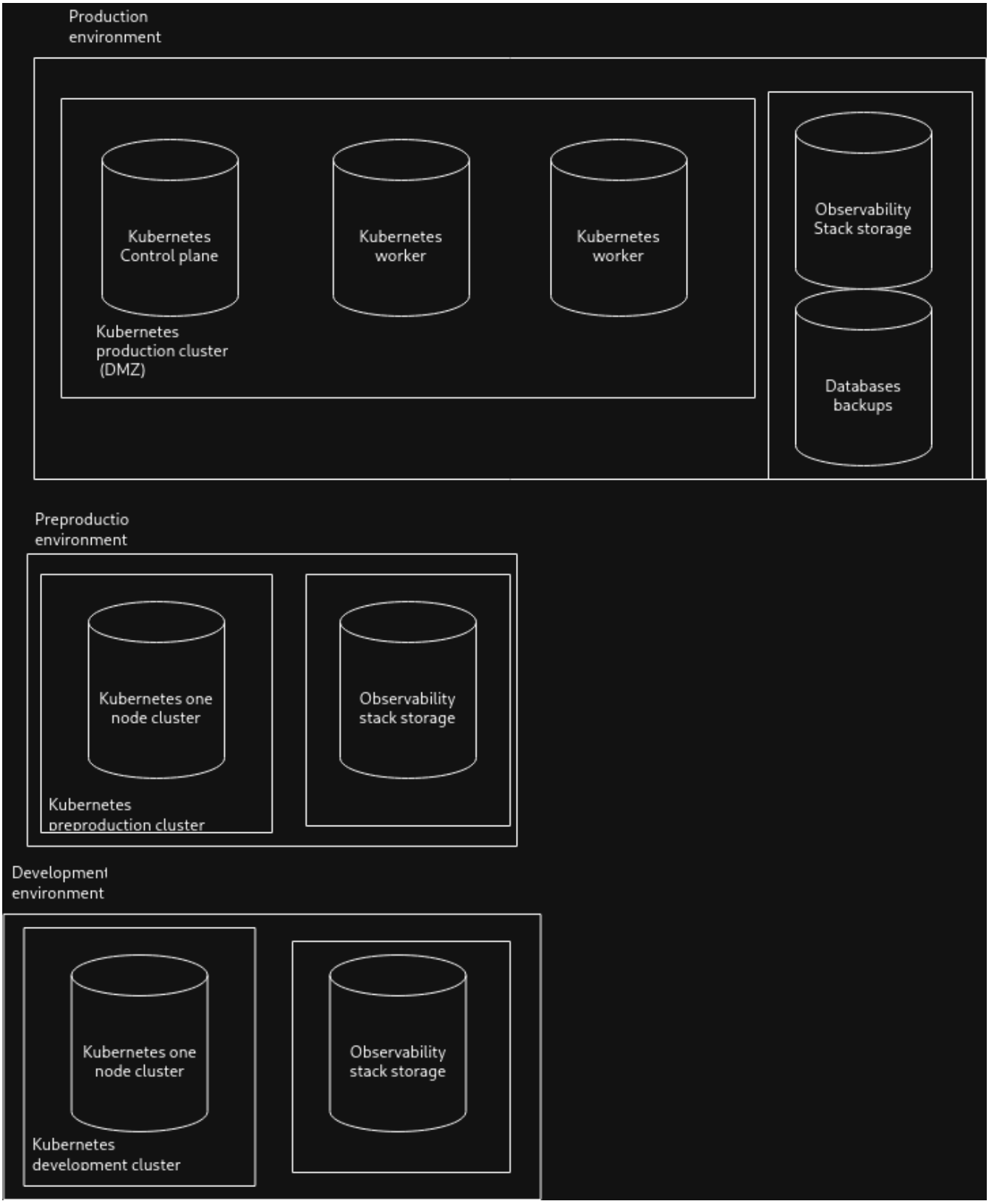
These servers are hosted by Polytech Montpellier, a French public university. For development purpose, we want to have at least three environments :

- Development
- Staging
- Production

Each environment need to be similar to the others, the only differences being the resource allocated to each environment. Each environment will be running on kubernetes.

5.1. Infrastructure Level 1

<Overview Diagram>



Motivation

> The idea here is that we have three kubernetes clusters, one for each environment. The production cluster will have most resources with three dedicated nodes. Each cluster is considered as a DMZ since it is hard to isolate the different services thanks to specific network configurations, for example VLANS. But we will see later how we can do that at the kubernetes level.

However, each kubernetes cluster will be monitored, thus the monitoring stack will generate a lot

of data. This data needs to be stored somewhere, so we will be using a dedicated server for the data retention.

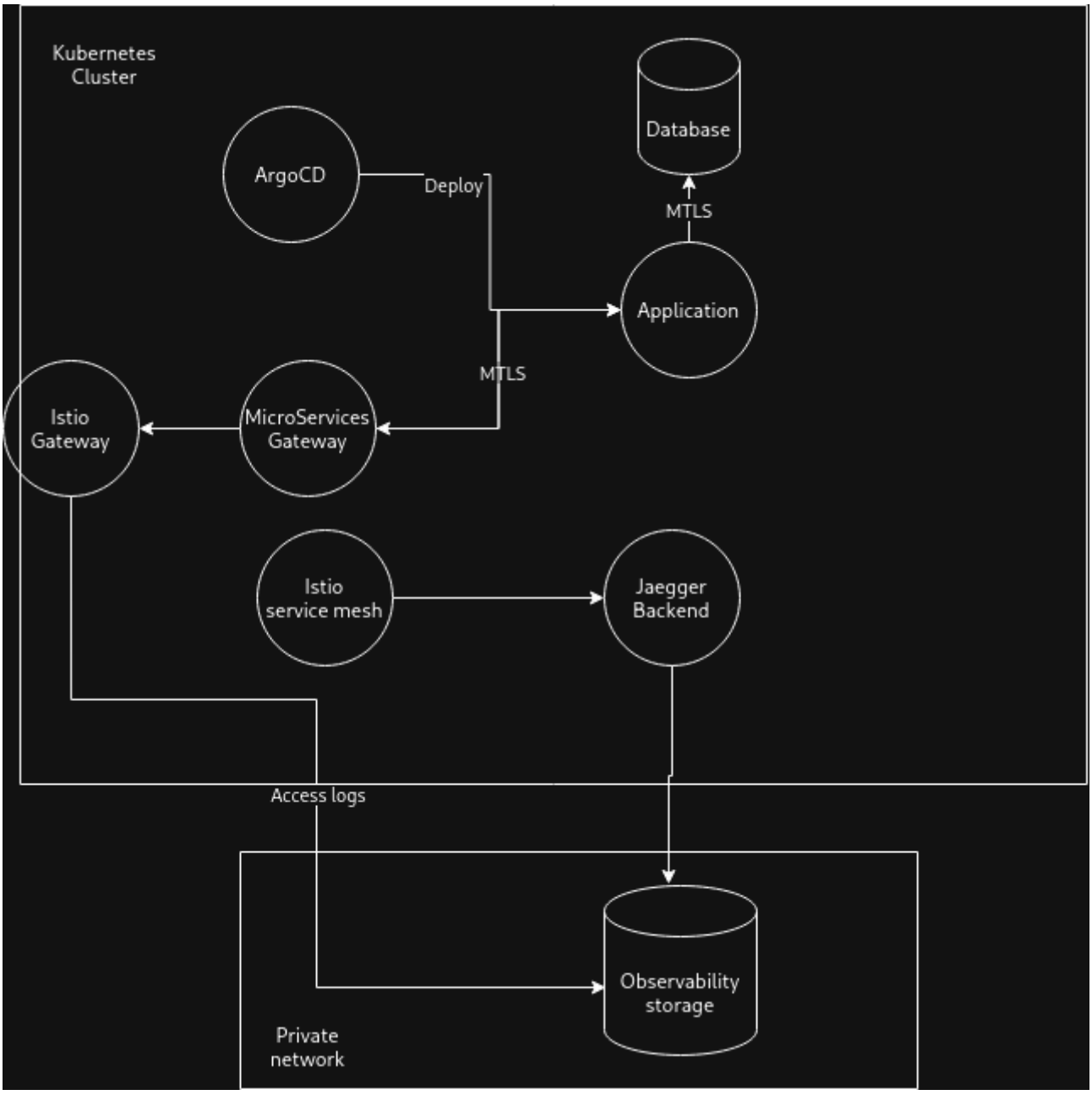
Log data retention is important even in a development environment, since we will be developing and testing Beep. We will need to keep logs for debugging purposes.

Quality and/or Performance Features

Putting the backups and log storage in dedicated servers that are only accessible by the kubernetes services will ensure that an attacker will have an harder time to access the data. That's the whole point of a DMZ.

5.2. Infrastructure Level 2

5.2.1. Focus on the Kubernetes Cluster



5.2.2. Load balancing

Istio is a service mesh that also provides a reverse proxy. According to this benchmark, Istio can serve around 4,000 requests per second on a 1 CPU server, and this metric can go up to 25,000 requests per second on an 8 CPU server. Given our hardware infrastructure, we can handle at least 100,000 requests per second, which is significantly more than the estimated volumetry of 333 messages per second.

To further optimize load balancing, we will use Kubernetes to automatically scale the number of pods based on the load. Rate limiting will be implemented to prevent abuse and ensure fair usage of resources. Caching mechanisms will be used to reduce the load on the backend servers. Circuit breakers will be implemented to prevent cascading failures. These strategies will help distribute the load evenly and maintain the performance of the Beep application.

Inter-service network security

The Kubernetes cluster operates within a Demilitarized Zone (DMZ), necessitating robust security measures for inter-service network communications. To address this, we will implement mutual Transport Layer Security (mTLS) for secure communication between services. Essentially, mTLS establishes a secure tunnel between two services, encrypting the network traffic with dedicated certificates. This encryption ensures that potential eavesdroppers are unable to intercept or decipher the communications, thereby enhancing the security and integrity of data exchanges.

This advanced security protocol will be facilitated by Istio, a powerful service mesh that provides comprehensive solutions for managing and securing microservices. Istio not only simplifies the implementation of mTLS but also offers additional features such as traffic management, observability, and policy enforcement, making it an ideal choice for securing our Kubernetes cluster. By leveraging Istio's capabilities, we can ensure that our inter-service communications are both secure and efficiently managed.

Logging and monitoring

Logging and monitoring are crucial aspects of any distributed system, as they provide valuable insights into the system's behavior and performance. In the context of our Kubernetes cluster, we will implement logging and monitoring solutions to capture and analyze system events, errors, and performance metrics. We will be using Istio as a Gateway for our services, which will enable us to collect and analyze traffic flows and performance metrics. This will help us identify bottlenecks, optimize resource allocation, and ensure the overall health and reliability of our system.

Aditionally, Istio provides a rich set of observability features, such as distributed tracing, log collection, and metrics aggregation. These features will enable us to gain deeper insights into the behavior of our services and identify areas for improvement.

We can pick any backend for our logging and monitoring solutions, so we will be using Jaegger for traces, Loki for logs and Prometheus for metrics. We will configure these services to use our external storage servers for data retention.

Deployment

For continuous deployment, we will be using ArgoCD, a powerful and flexible deployment tool that

simplifies the process of managing and deploying applications across multiple environments. ArgoCD allows us to define workflows for deploying our applications, managing their lifecycles, and ensuring consistent deployments across different environments.

Each resources in Kubernetes will be deployed using Helm and will be declared as a ArgoCDP application to keep a declarative approach to deployment. This will allow us to easily manage and update our applications, ensuring consistency and reducing the risk of human error.

Here is an example of a deployment workflow for an ArgoCD setup : https://github.com/Courtcircuits/cluster

Secret management

For secret management, we will be using sealed-secrets, a Kubernetes controller that allows us to store and manage secrets in a centralized location. This will ensure that sensitive information, such as database credentials, API keys, and other sensitive data, is securely stored and managed while keeping a Gitops approach to the project.

6. Cross-cutting Concepts

From our feature list we can extract a domain specific vocabulary. You may have observed that some words were highlighted previously.

- **user** : a user of Beep which identity has been registered.
- **entity**: an entity contains itself another component of beep. For example, a server is an entity because it contains channels. **Channels** are an entity because it contains **channels**, **messages** or **voice calls**.
- **server** : an entity that helps to regroup **channels**.
- **channels** : an entity that contains either **channels**, **messages** or a **voice call**
- **text channel** : a **channel** that only contains texts.
- **voice channel** : a **channel** that enable **voice calls**
- **folder channel** : a **channel** that contains **channels**.
- **message** : a text that is sent to another **user** and that might contain an attachment **file**
- **file** : it can be either a binary, image, video, text…. that can be stored in a computer filesystem.
- **member** : a member is part of an **entity**. It has associated **rights** and is associated to a **user**.
- **voice call** : a connection between **users** to support realtime audio calls + video calls.
- **notification** : an alert sent to a user making him aware of an event.

6.1. Defining the bounded contexts of the domains

- **user** : everything that is related to identity management of the user.
- **server** : managing the metadata of a server, the list of channels and its related policies.
- **channels** : managing the metadata of channels and its related policies.
- **message** : everything related to the management of messages through the application, the storage and its structure.

7. Risks and Technical Debts

7.1. ACID properties

Maintaining ACID (Atomicity, Consistency, Isolation, Durability) properties in a distributed system without the use of message queues presents significant challenges, particularly as the system scales and the complexity of transactions increases. Atomicity, which ensures that transactions are completed entirely or not at all, becomes difficult to guarantee when multiple services need to coordinate actions across different databases or data stores. Without message queues to manage and sequence these transactions, the risk of partial failures increases, potentially leaving the system in an inconsistent state.

Consistency, which requires that a transaction brings the database from one valid state to another, is similarly challenging to maintain. In a distributed environment, ensuring that all parts of the system see the same data at the same time is non-trivial. Message queues help by serializing access to shared resources and ensuring that updates are propagated in a controlled manner. Without them, race conditions and conflicts can arise, leading to inconsistencies that are difficult to detect and resolve.

Isolation, the property that ensures transactions are executed in isolation from one another, is also harder to achieve without message queues. In a highly concurrent environment, transactions can interfere with each other, leading to dirty reads, non-repeatable reads, or phantom reads. Message queues can help manage the flow of transactions and ensure that they are processed in a way that maintains isolation. Without this mechanism, developers must implement complex locking and concurrency control strategies, which can be error-prone and difficult to maintain.

Finally, Durability, which guarantees that once a transaction has been committed, it will remain so even in the event of a system failure, is challenging to ensure without message queues. Message queues provide a buffer that can hold transactions until they are safely processed and stored, even if parts of the system go offline. Without this buffer, transactions in flight can be lost during failures, leading to data loss and inconsistency.

In summary, message queues play a crucial role in managing the complexities of distributed transactions and ensuring that ACID properties are maintained. Without them, developers must rely on more complex and potentially less reliable mechanisms to achieve the same level of data integrity and consistency.

7.2. Designing asynchronous APIs without message queues

Designing asynchronous systems without message queues presents substantial challenges, particularly in terms of scalability and the effective implementation of asynchronous events. Scalability in distributed systems relies heavily on the ability to decouple components, allowing them to operate independently and at their own pace. Message queues facilitate this decoupling by acting as a buffer between producers and consumers, absorbing load spikes and enabling components to scale horizontally. Without message queues, systems often struggle to manage varying loads, leading to bottlenecks and degraded performance as the number of concurrent

operations increases.

Implementation of asynchronous events becomes significantly more complex without message queues. In an asynchronous system, events such as user actions, system notifications, or data updates need to be processed in a non-blocking manner to ensure responsiveness and efficiency. Message queues simplify this by providing a reliable mechanism to enqueue events and ensure they are processed in the correct order and at the appropriate time. Without this mechanism, developers must implement custom solutions to handle event sequencing, retry logic, and error handling, which can be error-prone and difficult to maintain.

Furthermore, message queues provide built-in support for features such as dead-letter queues, delayed messages, and priority queues, which are essential for robust asynchronous processing. These features help manage failures, retries, and prioritization, ensuring that the system remains resilient and responsive under varying conditions. Without message queues, developers must build these capabilities from scratch, adding layers of complexity and potential points of failure to the system.

8. Glossary

Term	Definition
User	A user of Beep whose identity has been registered.
Entity	An entity contains itself another component of Beep. For example, a server is an entity because it contains channels. Channels are an entity because they contain channels, messages, or voice calls.
Server	An entity that helps to regroup channels.
Channels	An entity that contains either channels, messages, or a voice call.
Text Channel	A channel that only contains texts.
Voice Channel	A channel that enables voice calls.
Folder Channel	A channel that contains channels.
Message	A text that is sent to another user and that might contain an attachment file.
File	It can be either a binary, image, video, text, etc., that can be stored in a computer filesystem.
Member	A member is part of an entity. It has associated rights and is associated with a user.
Voice Call	A connection between users to support real-time audio calls and video calls.
Notification	An alert sent to a user making them aware of an event.
Istio	An open platform to connect, manage, and secure microservices. It provides a way to control the flow of traffic and API calls between services, as well as a policy enforcement and telemetry collection.
gRPC	A modern open-source high-performance Remote Procedure Call (RPC) framework that can run in any environment. It is used for communication between microservices in Beep.
Kubernetes	An open-source system for automating the deployment, scaling, and management of containerized applications. It helps in managing the microservices architecture of Beep.
PostgreSQL	An open-source relational database management system known for its robustness and performance. It is used as the primary database for Beep.
REST	Representational State Transfer, an architectural style for designing networked applications. It is used for communication between some components of Beep.
Microservices	A software development technique—a variant of the service-oriented architecture (SOA) structural style—that arranges an application as a collection of loosely coupled services. Beep is transitioning to this architecture.

Term	Definition
OIDC	OpenID Connect, an identity layer built on top of the OAuth 2.0 protocol. It is used for authentication in Beep.
LDAP	Lightweight Directory Access Protocol, used for accessing and maintaining distributed directory information services over an Internet Protocol (IP) network. It is used for university credentials authentication in Beep.

9. Appendix

9.1. POCS

search engine + inter communication → chat.courtcircuit.xyz auth → auth.courtcircuit.xyz