Beep Technical Architecture Document (TAD)

Table of Contents

1. Introduction, Beep User Stories and Functional Neighboorhood

1.1 Introduction

This Technical Architecture Document (TAD) provides a comprehensive overview of **Beep**, a web application designed to provide real-time communication, similar to Discord. It outlines the **infrastructure**, **architecture**, and **deployment** of the application, as well as **security architecture**, **data architecture**, and the **search engine** implementation.

The document is structured into several sections, each dedicated to a specific component of the system, allowing a clear understanding of how the application works. The primary goal of this document is to serve as the foundation for transitioning Beep from a monolithic architecture to a microservices-based system.

This document is intended for **developers**, **architects**, and **stakeholders** involved in the Beep project. As the project evolves and new features are introduced, this document will be updated to reflect changes in the system architecture, ensuring a detailed and current view of the Beep

platform's evolution.

Through this document, the intention is to ensure that the necessary design and architectural decisions are in place for Beep's **microservice** transformation, offering insights into its security, scalability, and overall system structure.

1.2 Beep User Stories

Authentication and Authorization

- As a guest user, I want to sign up so that I can create an account and access the platform.
- As a guest user, I want to sign in so that I can access my account.
- As a guest user, I want to reset my password so that I can recover my account.

User Profile Management

- As a registered user, I want to update my profile so that I can keep my information up to date.
- As a registered user, I want to change my password so that I can recover my account.
- As a registered user, I want to enable two-factor authentication so that I can add an extra layer of security to my account.

Server Management

- As a server admin or with the right permissions, I want to create a server so that I can manage a community.
- As a server admin or with the right permissions, I want to delete a server so that I can remove it permanently.
- As a server admin or with the right permissions, I want to update server settings so that I can configure the server according to my needs.
- As a server admin or with the right permissions, I want to manage roles and permissions so that I can control access to different features.
- As a server admin or with the right permissions, I want to be able to invite users to my server so that I can grow my community.
- As a server admin or with the right permissions, I want to create webhooks so that I can automate actions in my server.

Channel Management

- As a server admin or with the right permissions, I want to create a channel so that I can organize discussions.
- As a server admin or with the right permissions, I want to delete a channel so that I can remove unnecessary channels.
- As a server admin or with the right permissions, I want to update channel settings so that I can configure the channel according to my needs.

- As a server admin, I want to move channels so that I can reorder them.

- As a registered user with the right permissions, I want to join a video or vocal call in a channel so that I can communicate in real-time with other users.

- As a registered user, I want to share my screen during a call so that I can present content to others.

- As a registered user, I want to mute/unmute myself during a call so that I can control my audio.

- As a registered user, I want to select my audio and video devices so that I can choose the appropriate hardware for calls.

Messaging

- As a registered user with the right permissions, I want to send messages in a channel so that I can communicate with other members.

- As a registered user, I want to edit my messages so that I can correct mistakes.

- As a registered user, I want to delete my messages so that I can remove them.

- As a registered user, I want to pin messages so that important messages are highlighted.

- As a server admin or with the right permissions, I want to delete inappropriate messages so that I can maintain a healthy community environment.

- As a registered user, I want to mention other users in a message so that they are notified.

- As a registered user, I want to be able to see a preview of links shared in messages so that I can decide if I want to open them.

- As a registered user, I want to send private messages so that I can communicate directly with friends.

Frineds

- As a registered user, I want to send friend requests so that I can connect with other users.

- As a registered user, I want to accept friend requests so that I can add new friends.

- As a registered user, I want to delete a friendship so that I can remove them from my friend list.

Notifications

- As a registered user, I want to receive notifications for friend requests so that I am aware when someone wants to connect.

- As a registered user, I want to receive notifications for private messages so that I know when someone contacts me.

- As a registered user, I want to receive notifications when I am mentioned in a channel so that I can respond quickly.

1.3 Functional Neighborhood

Authentication and Authorization

- Sign Up
- Sign In
- Reset Password

User Profile Management

- Update Profile
- Change Password
- Enable Two-Factor Authentication

Server Management

- Create Server
- Delete Server
- Update Server Settings
- Manage Roles and Permissions

Channel Management

- Create Channel
- Delete Channel
- Update Channel Settings
- Move Channels
- Join Video/Vocal Calls
- Share Screen

Messaging

- Send Messages
- Edit Messages
- Delete Messages
- Pin Messages
- Send private messages
- Mention Users

Friends

- Send Friend Requests
- Accept Friend Requests

- Delete Friendship

Notifications

- Receive Friend Request Notifications

- Receive Private Message Notifications

- Receive Mention Notifications

2. Infrastructure, Architecture and Deployment Overview

2.1 Infrastructure

Physical and Virtual Infrastructure

The Beep application is deployed on a physical blade server located in the Polytech datacenter.

Physical Server

A dedicated bare-metal server hosts the infrastructure. It runs Proxmox as the hypervisor.

Virtual Machine

A single virtual machine, `beep-vm`, runs Debian 12 and hosts the entire K3s Kubernetes cluster. Access to the VM is restricted to internal network users via VPN. The VM is provisioned with 8 vCPUs and approximately 16 GB of RAM.

Kubernetes Cluster

General

The infrastructure uses K3s, a lightweight Kubernetes distribution. It is configured as a single-node cluster, where `beep-vm` acts as both the control-plane and worker node. Networking is managed by the Flannel plugin.

Application and Namespaces

The Beep application is deployed within Kubernetes and organized into three distinct namespaces, each representing an environment: - `beep-development` - `beep-staging` - `beep-production`

Each namespace contains the application stack for its environment, including frontend, backend API, and WebRTC services.

Network and Access

DNS management is handled via Cloudflare. Domains are configured to point to public IPs served

by Cloudflare, which forwards traffic to the ingress controller inside the cluster. Access to the Kubernetes control plane is restricted and must go through the VPN.

Ingress

Ingress is managed by Traefik, deployed in the `kube-system` namespace. It acts as the main entrypoint for HTTP(S) traffic and routes requests to the appropriate services within the cluster.

Storage and Persistence

Persistent volumes are provisioned for stateful services, including PostgreSQL and Redis. These volumes ensure that application data is retained across restarts.

Container Registry

Harbor is used as a private container registry. Although deployed within the Kubernetes cluster, it serves as a shared registry and is not exclusive to the Beep application.

Supporting Infrastructure Services

In addition to core services, the following components are deployed as part of the cluster infrastructure:

- **Sealed Secrets** is used to manage encrypted Kubernetes secrets declaratively.
- **Cert-manager** automates TLS certificate issuance and renewal, ensuring secure ingress.
- **GitLab Runner** is deployed to execute CI pipelines inside the cluster.
- **Grafana, Loki, and Tempo** provide monitoring, log aggregation, and tracing capabilities.
- **Terraform** is used to provision and manage all infrastructure resources as code.

Infrastructure Diagram

image::../diagrams/png/Infrastructure_diagram.drawio.png[width=1000, align=center] // TODO add the diagram

2.2 High-Level Architecture

Overview

The Beep application is composed of several independent microservices that expose REST APIs, orchestrated behind an API gateway. Each service has its own dedicated database. The application also includes a standalone WebRTC service used directly by the frontend for real-time communication.

Components

Frontend

- Implemented in React

- Communicates with backend services via the API gateway (Kong)

- Interacts directly with the WebRTC service for media connections

API Gateway

- **Kong** is used to route HTTP(S) requests to the various microservices

- Performs authentication and token validation

Microservices

Each microservice is responsible for a specific domain and owns its data:

- **Notification API**
  - Handles user notifications
  - Backed by its own `Notif DB`

- **Permission API**
  - Manages access controls and user roles
  - Backed by `Permission DB`

- **Server API**
  - Manages user-created servers or groups
  - Backed by `Server DB`

- **Channel API**
  - Handles channels inside servers
  - Backed by `Channel DB`

- **Friend API**
  - Manages friend requests and social graph
  - Backed by `Friend DB`

- **Message API**
  - Manages messaging functionality between users
  - Backed by `Message DB`

WebRTC Service

- A separate service used for establishing peer-to-peer media connections

- Not exposed through Kong, it is accessed directly by the frontend using WebRTC protocols

Architecture Diagram

image::../diagrams/png/beep-app-architecture.png[width=1000, align=center] // TODO replace with the actual path

2.3 Deployment Overview

Application Topology

The Beep application is structured into three environments: development, staging, and production. Each environment contains the same stack:

- A frontend web application built with React

- A backend API implemented with AdonisJS

- A WebRTC service

These services are deployed in separate Kubernetes namespaces: - `beep-development` - `beep-staging` - `beep-production`

They communicate internally via HTTP, and external access is routed through an API gateway.

CI/CD Pipeline

The deployment of Beep is automated using GitLab CI pipelines. Each push to the main branches triggers the build, test, and deployment processes.

Deployment into the Kubernetes cluster is managed using ArgoCD, which synchronizes application manifests stored in the Git repository. Each environment is deployed from its corresponding manifest file.

ArgoCD manages only application components (Beep, WebRTC) and related configurations (ConfigMaps and secrets). Core infrastructure components such as PostgreSQL, Redis, and observability tools are deployed separately using Helm charts.

Authentication and Authorization

Keycloak is used as the central identity provider. It manages user accounts and authentication flows using the OpenID Connect (OIDC) protocol. Applications redirect users to Keycloak for login and receive identity tokens (ID and access tokens) upon successful authentication.

Kong, acting as the API gateway, does not interact directly with Keycloak but is configured to validate OIDC tokens issued by Keycloak. It uses the public keys published by Keycloak to verify the authenticity and validity of tokens included in client requests.

API Gateway

Kong serves as the main API gateway for routing and securing HTTP traffic. It performs:

- Request routing

- Rate limiting

- OIDC token validation using Keycloak's public keys

- Logging and metrics forwarding

Service Mesh

Linkerd is deployed in the cluster. It provides:

- Transparent service-to-service encryption via mTLS

- Traffic metrics and golden signals per service

- Failure handling with retries and timeouts

Configuration and Secrets

Configuration values are managed via Kubernetes ConfigMaps, which are set per environment. Sensitive data such as API tokens, SMTP credentials, and database passwords are encrypted and stored as Sealed Secrets. Each environment has its own sealed secret manifest. These are encrypted using the cluster's public key and decrypted at runtime by the Sealed Secrets controller.

Observability Overview

Grafana is deployed in the `monitoring` namespace and displays dashboards. Loki receives logs from services, and Tempo is configured to trace API calls. Uptime Kuma, also deployed in the cluster, monitors HTTP endpoints of the application externally.

Deployment Diagram

image::../diagrams/png/deployment_diagram.drawio.png[width=1000,align=center] // TODO replace diagram

3. Security

3.1 Authentication

Overview

Our current authentication is handled by Keycloak, and this part is focusing on how we control user onboarding and login flows through "Vanilla", LDAP and Google identity providers. The configuration enforces explicit user actions before allowing access via federated identity sources.

Goal

This setup ensures strong control over user identity, avoids unwanted or duplicate accounts, and enforces verification and linking logic that aligns with our security requirements.

Authentication Flows Summary

- Local sign-up requires email confirmation before login is allowed.

- LDAP users must explicitly sign up before login; automatic login on first bind is disabled.

- Google login requires prior manual account linking; automatic account creation on first login is disabled.

- All user flows are designed to be intentional, explicit, and secure.

Local User Sign-up

Users are allowed to register using a traditional username and password via Keycloak's built-in registration form. After completing the registration form, Keycloak sends a confirmation email to the user. The account remains inactive until the email is confirmed. Only after the confirmation is the user allowed to log in.

This process ensures that all local users have verified their email addresses before gaining access to the system so that we can be sure that the user is the owner of the email address. This is a critical step in preventing unauthorized access and ensuring that users are legitimate.

image::../diagrams/png/Signup.png[width=1000,align=center] // TODO add the diagram

Polytech LDAP Sign-up and Login

Polytech LDAP is configured as a user federation provider. However, users are **not allowed** to log in with their LDAP credentials unless they have explicitly performed a "Sign up with LDAP" beforehand.

When a user chooses "Sign up with LDAP", they are redirected to Keycloak's login page, where LDAP is the selected authentication mechanism. Upon successful authentication, Keycloak imports the user into its local storage. This initial import is required to enable subsequent logins.

When a user chooses "Login with LDAP", they are redirected to Keycloak's login page, where LDAP is the selected authentication mechanism. If a user attempts to log in with LDAP credentials without having signed up, Keycloak denies access since no local user record exists yet.

image::../diagrams/png/LDAP.png[width=1000,align=center] // TODO add the diagram

Google Login and Account Linking

Google is configured as an external Identity Provider through Keycloak's identity brokering feature : OpenID Connect provider. However, direct login via Google is **not allowed**.

Users must first log in using their local account and then explicitly associate their Google account through the "Associate my Google account" option. This triggers the standard OAuth2 login flow with Google and, upon success, Keycloak links the external identity to the existing local user.

Once the account is linked, users may subsequently use "Login with Google" to authenticate. If a user attempts to log in with Google without having completed the linking process, Keycloak denies access and does not create a new user automatically.

image::../diagrams/png/Google.png[width=1000,align=center] // TODO add the diagram

Technical Configuration Notes

- The `Registration Flow` includes an email verification step and must be completed to activate new local users.

- The `First Broker Login Flow` used for Google has been customized to remove automatic account creation and enforce explicit linking.

- LDAP user federation is set with:

- `Import Users = ON`

- `Edit Mode = UNSYNC` because we do not modify the LDAP directory, just read from it, importing users into Keycloak and modifying them there.

- Google IdP is set with:

- `Trust Email = ON`

- `Sync Mode = IMPORT`

- `Account Linking Only = ON` to ensure that users must link their Google account explicitly in their local account.

- `Hide on Login Page = ON` to prevent users from logging in directly with Google.

3.2 Authorization

4.1 Authorization service definition

an authorization service is a system that determines whether a user or service has permission to perform a certain action or access a certain resource. it usually works alongside an authentication service (which verifies identity) but focuses on what a user can or cannot do once authenticated.

it can be based on role-based access control (RBAC), attribute-based access control (ABAC), or policy-based access control (PBAC).

Articles to read: https://medium.com/@kgignatyev/unification-of-authorization-for-ui-and-backend-with-casbin-1523758658f7 https://medium.com/@rohanraman6/integrating-keycloak-for-authentication-and-authorization-a-step-by-step-guide-72bd382a2bd1

4.2 Technical and functional architecture and the technologies I recommend

I will use the permissions service to manage authorizations. This service is base on permissions and roles to manage the authorizations. Every service when they need to know the permisions of a user will ask the permissions service. The permissions service will check the permissions of the user and will return 401 if the user doesn't have the permission to do the action or 200 if the user has the permission to do the action.

The permissions service will have a database with the permissions and the roles of the users. The permissions service will have an API to get the permissions of a user.

Example of request to the permissions service to create a channel :

```
POST /permissions HTTP/1.1
Host: permissions-service:8080
Content-Type: application/json
Authorization: Bearer token

Body :
{
    "userId": "123",
    "resource": "channel",
    "action": "create",
    "serverId": "456"
}
```

Technologies:

For the permissions service, I will keep it simple and use a REST API with a database to store the permissions and the roles of the users. I will keep the technology already used in the project to keep the project simple and easy to maintain, which is AdonisJs for the API and PostgreSQL for the database.

4.3 Sequence diagrams for the application's main actions

Here is the sequence diagrame for a user trying to change roles for a server : image::../diagramsv2/png/Authorization_system1.drawio.png[width=1000,align=center]

Here is the sequence diagrame for a user trying to change roles for channel categories : image::../diagramsv2/png/Authorization_system2.drawio.png[width=1000,align=center]

Here is the sequence diagrame for a user trying to join a vocal channel : image::../diagramsv2/png/Authorization_system3.drawio.png[width=1000,align=center]

Here is the sequence diagrame for a user who is a global administrator trying to join a vocal channel : image::../diagramsv2/png/Authorization_system4.drawio.png[width=1000,align=center]

3.2 WAF, API Gateway, and Microservices Communication

3.1.1 Web Application Firewall (WAF)

Before any request reaches our API Gateway, it first traverses a Cloudflare **Web Application Firewall (WAF)**. The WAF inspects incoming traffic, blocking malicious or malformed requests and protecting the system against threats like **SQL injections**, **cross-site scripting (XSS)**, and **denial-of-service (DoS)** attacks. Only valid requests passing WAF checks are forwarded to the API Gateway.

3.1.2 API Gateway

After the WAF, requests arrive at the **API Gateway**, the central entry point for all client traffic. The API Gateway handles:

- **Authentication and authorization** – It verifies client credentials or tokens (obtained from Keycloak) before granting access.
- **Token forwarding** – Once Keycloak issues an access token, the client includes it in subsequent requests. The API Gateway forwards this token to the target microservice. Each microservice uses Keycloak's **public keys** to validate the token.
- **Routing** – It directs requests to the appropriate microservice based on request paths or rules.
- **Rate limiting** – It prevents abuse by limiting the number of requests a client can make within a specific timeframe.
- **Monitoring and logging** – All incoming and outgoing requests are logged for future analysis, debugging, and auditing.

By preventing direct access to microservices, the API Gateway adds a layer of security, ensuring microservices remain behind the gateway and are never directly exposed to the public internet.

3.1.3 Communication Protocols and Service Mesh

For inter-service calls, microservices communicate over **HTTP/REST**, making them straightforward to develop and integrate. Each microservice exposes a RESTful API, enabling flexible and decoupled service interactions.

Service discovery is managed via **Kubernetes DNS**, allowing services to be reached through resolvable names like `channel-service.beep.svc.cluster.local`.

To secure and observe internal service communication, **Linkerd** is deployed as the **Service Mesh**. Linkerd provides:

- **mTLS (Mutual TLS)** – Automatically encrypts and authenticates inter-service traffic. When a microservice calls another, Linkerd sidecar proxies transparently handle encryption and certificate validation, ensuring that only trusted services can communicate.
- **Mesh-enabled Ingress** – Both the ingress controller (Traefik) and the API Gateway (Kong or another) can be "meshed," meaning all traffic within the cluster is secured by mTLS.
- **Observability** – By default, Linkerd gathers metrics for success rates, latencies, and request volumes, enhancing visibility.
- **Resilience** – Features like retries, timeouts, and circuit breaking can be configured to handle communication failures gracefully.

Through Linkerd, each microservice benefits from end-to-end encryption and service-to-service authentication, creating a secure and reliable mesh for internal communication.

3.1.4 Service Mesh POC

The service mesh proof of concept (POC) is implemented using Linkerd, a lightweight service mesh

designed for Kubernetes. It provides features like mTLS, observability, and traffic management. The app is a simple page that displays a random fun compliment. To see more details about the POC you can read the POC's README.md. A live proof of concept (POC) of the system is accessible at the following addresses:

- Application frontend: http://compliments.162.38.112.221.nip.io/
- Linkerd dashboard: http://linkerd-dashboard.162.38.112.221.nip.io/
- Direct access to Traefik metrics/traffic: http://linkerd-dashboard.162.38.112.221.nip.io/namespaces/traefik/deployments/traefik

You can refresh the page to see some changes in the dashboard.

The domain resolution is handled via **nip.io**, a free wildcard DNS service that maps subdomains to IP addresses automatically. This enables quick setup and access to services using predictable URLs without manual DNS configuration.

3.2 Sequence Diagram

The diagram below illustrates a typical flow where a user makes a request that involves authentication via Keycloak, forwarding of the token by the API Gateway, and an internal microservice call secured by Linkerd's mTLS.

3.3 Logging & Monitoring

3.4 Data Security & High Availability

4. Search Engine