# Architecture and Design Document LiDARC

## Architectural Overview

The system is designed as a distributed, event-driven architecture. The responsibilities are split between frontend, backend, and specialized worker services. Long-running and compute-intensive tasks are handled asynchronously via a message broker (*RabbitMQ*) and the corresponding workers.
The User can upload files via the frontend that get stored in the ***MinIO*** bucket. A successful upload triggers the ***Metadata-Worker*** through ***RabbitMQ*** to extract important metadata from the file and store it in the ***PostgreSQL***. After successful uploading different events can be triggered through *RabbitMQ* as each worker is listening on a specific queue to receive a job to work on.
Those jobs are meant for different steps of a started comparison and include:

1. ***Preprocessing-Worker***
    a. Computational extraction of the points from each file and store it in a JSON Schema form in ***MinIO***.
2. ***Comparison-Worker***
    a. Takes all corresponding preprocessing files and starts comparing each cell of file A to the same cell of file B and stores the result in ***MinIO***.

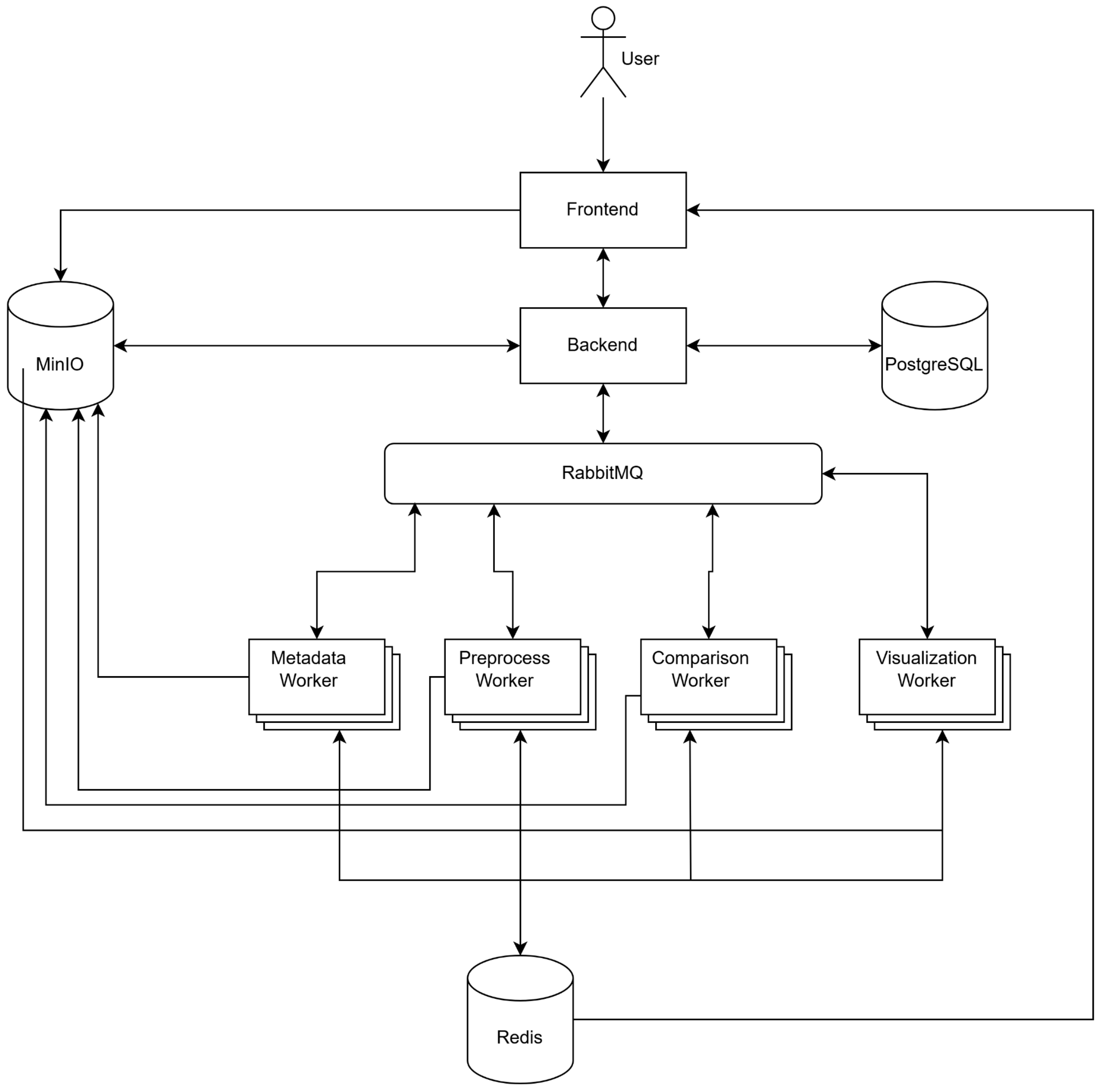
Finished comparisons can be viewed in the *Comparisons* Tab. An opened comparison triggers the

3. ***Visualization-Worker***
    a. Takes the finished comparison result and converts the data into ECharts friendly format. Stores finished visualization data in ***Redis*** Cache.

The three datastorages store:

- MinIO
    - Uploaded files
    - Preprocessing of compared files
    - Comparison Results of compared files
- PostgreSQL
    - Metadata of uploaded files
- Redis Cache
    - Visualization Result of viewed Comparison Result

## Architecture Model



## Component Breakdown

### Frontend

Main task of the frontend are User interaction, triggering of backend requests and visualization of results. All computation intensive task are outsourced to the workers and the frontend just displays the received results in different charts.

## Backend

The Backend is the Orchestration layer of our system as it receives job requests from the frontend and publishes the corresponding tasks into RabbitMQ. It also validates input and persists metadata.

## RabbitMQ

Functions as the communication Backbone to our system. The backend publishes task it receives through frontend actions or received results from workers. It decouples producers (backend) and consumers (workers) and enables asynchronous processing. Each Worker has its own designated exchange/queue for jobs and then the results which enables a messaged based workflow between the workers. Finished workers publish an ACK message into the result queue, which triggers backend actions (new Job, finished upload/comparison,..)

## Processing pipeline/Workers

Each worker has a specific task which adds to the overall workflow. They get triggered through designated queues and publish a result message into worker specific result queues. All workers cache small files for faster access.

### Metadata Worker

Extracts and processes metadata of an uploaded file. It receives the job after successful upload into the MinIO bucket and the result gets stored in PostgreSQL. It is the first stage of our pipeline as the uploaded files can be viewed without any started comparisons. The information extracted from the worker is viewed in the frontend.

### Preprocessing Worker

Extracts the points of raw pointcloud files from MinIO and does computational preprocessing based on the provided setup of the comparison. It produces a normalized format and stores it for each file in MinIO. A finished Preprocessing result triggers the comparison worker through the backend.

### Comparison Worker

Works with the preprocessing result of the Preprocessing Worker. It compares the cells of each preprocessed file A to the  preprocessed file B. The result includes the comprised cells with the comparison values for A and B, as well as statistics of the whole files.

### Visualization Worker

If a finished comparison is viewed by the client, the backend triggers the visworker to provide the frontend with visualization data of the specific comparison. The visualization worker fetches the corresponding comparison result and does transformation tasks to provide the frontend with echarts friendly format. Additionally it performs chunking, initially based on the overall cellCount of the comparison to make visualization quicker at startup. The user has the opportunity to set the chunking Size via a slider, which triggers the Visworker again with the selected value. Output is the chunked comparison result to visualize in all echarts components.

## Datastorage Architecture

### MinIO

Object storage for whole pointcloud files, preprocessing results and comparison results. It is used for raw and intermediate data and chosen for its S3 compatibility and scalability.

### PostgreSQL

Persists Metadata for each uploaded pointcloud file and used to display information in the frontend. It guarantees strong consistency.

### Redis

Enables fast access to cached visualization data as well as fast access to small files.

## Data & Control Flow

When a user initiates a file upload, the frontend requests a presigned upload URL from the backend. The backend validates the request and returns the URL, allowing the frontend to upload the file directly to object storage. After a successful upload, the backend triggers an asynchronous metadata extraction process via the message broker. Extracted metadata is persisted in the relational database and made available to the frontend.

When a comparison is started, the backend coordinates a multi-stage processing pipeline. Preprocessing jobs are distributed to workers and executed concurrently. Once all preprocessing tasks have completed, a comparison job is triggered. Visualization data is generated on demand and provided to the frontend asynchronously via Redis Cache.

## Scalability & Deployment Considerations

The system is designed with scalability and reliability as core architectural principles. Workers can be scaled and redeployed according to the job demands due to Kubernetes integration. Asynchronous communication via RabbitMQ decouples producers from consumers and enables load buffering, preventing system overload during peak usage. Fault tolerance is achieved through isolated worker responsibilities and persistent message queues. Dynamic Job Status Management ensures that no job failures get properly propagated to the frontend. Additionally, the use of dedicated storage systems for object data, relational data, and transient state improves overall reliability and allows each component to fail or recover independently.

The system can be deployed using a single command:
**docker compose up -d**