

Decentralized File Exchange Hub: A Cloud-Native Approach

Rafael Pereira^a

^aComputer Science and Communications Research Centre, School of Technology and Management, Polytechnic of Leiria, 2411-901 Leiria, Portugal

ARTICLE INFO

Keywords:

Decentralized file exchange
Microservices architecture
Containerization
Cloud-native deployment
Real-time communication
Scalability and resilience

ABSTRACT

The exchange of files has been a fundamental aspect of communication and collaboration throughout the history of computing. From the early days of floppy disks and bulletin board systems to the rise of email attachments and file-sharing platforms, the need to transfer digital files has continued to evolve. In recent years, with the proliferation of cloud-based services and an increasing focus on decentralization, the landscape of file exchange has become more complex and diverse.

In this context, our decentralized file exchange hub aims to revolutionize the way individuals and organizations share and manage files. By leveraging the power of cloud-native technologies and a modular, microservices-based architecture, our system simplifies the file exchange process, providing an intuitive and efficient experience for users. By combining the benefits of a React-based Presentation Provider served by Nginx, an Express File Management Service, a real-time Asynchronous Message Communication Server using Socket.IO, a File Gateway Service, and MongoDB as the primary database, we are able to offer a robust and scalable solution that adapts to the ever-changing demands of the digital world.

With our system in place, users can easily exchange files without worrying about the underlying infrastructure, ensuring that they can focus on their core tasks and projects. Our application stores files in Google Cloud Storage, and our deployment pipeline is streamlined using Google Cloud Build and Cloud Run, as well as other GCP services such as Cloud Storage Buckets and Cloud DNS. As the need for secure, reliable, and efficient file-sharing solutions continues to grow, our decentralized file exchange hub stands as a testament to the potential of modern cloud computing technologies to reshape the landscape of digital collaboration.

1. Introduction

The rapid growth of cloud computing and microservices has revolutionized the way applications are developed and deployed. As a result, there is an increasing need for effective strategies to manage the complexity and scalability of these systems. This report explores the design and provisioning of a distributed cloud-based file exchange hub that utilizes containerization, microservices, and cloud services to achieve high availability, performance, and maintainability.

The file exchange hub aims to provide a user-friendly, secure, and efficient platform for users to share files with one another. By utilizing a decentralized approach, the hub ensures enhanced privacy and control over shared data, while also promoting better performance and availability. The system leverages the power and flexibility of modern cloud computing technologies, such as containerization, microservices, and cloud services, to deliver a streamlined and scalable solution. This innovative approach addresses the limitations often found in traditional, centralized file-sharing systems, offering a more resilient and adaptable alternative for seamless file exchange.

The proposed solution is divided into four main modules: Presentation Provider (Nginx), API (Express), Asynchronous Message Communication server (Socket.IO), and Database (MongoDB). Each module is designed to serve a specific purpose and can scale independently to adapt to changing


loads and requirements. The application also leverages Google Cloud Platform (GCP) services such as Cloud Run, Cloud Storage Bucket, and Cloud DNS for deployment, file management, and domain management.


The report is structured as follows: Section 2 presents the architecture of the application, detailing the purpose and design of each module. Section 3 discusses the two scenarios in which our application can be set up and run, namely the local and production scenarios, and outlines their respective architectures and communication patterns. Section 4 describes the provisioning process, emphasizing the use of GCP services, Terraform, and GitHub Actions to automate and streamline the application components' deployment. Section 5 provides a comprehensive discussion on the different scenarios, difficulties faced during provisioning, and offers recommendations for developers working on similar projects. Finally, the report concludes with a summary of the key findings and insights gained from the project.

2. Architecture

This project follows a microservices architecture, which allows for improved scalability, resilience, and maintainability. The system is divided into four main components, each responsible for a specific aspect of the application. Figure ?? presents a diagram of the architecture, illustrating the relationship between each component.

1. Presentation Provider (Nginx): This component acts as the frontend server, hosting the React web application and serving static files. It is responsible for delivering the user interface to the clients and handling incoming HTTP requests.

 rafael.m.pereira@ipleiria.pt (R. Pereira)

 ORCID(s): 0000-0001-8313-7253 (R. Pereira)



[https://www.linkedin.com/profile/view?id=](https://www.linkedin.com/profile/view?id=rafaelmdespereira)

'rafaelmdespereira' (R. Pereira)

2. **API (Express File Management Service):** This component is the backend server implemented using Express.js. It manages the file uploading and downloading processes, interacting with the Database and Cloud Storage Bucket to store and retrieve files.
3. **Asynchronous Message Communication Server (Socket.io):** This server enables real-time communication between the clients, ensuring that users in the same room receive instant updates when a new file is shared.
4. **Database (MongoDB):** The database stores information about the rooms, users, and file metadata, including the bucket URL and timestamp. MongoDB was chosen for its flexibility, scalability, and performance.
5. **Cloud Storage Bucket (Azure Blob Storage):** This component stores the uploaded files, providing a reliable and scalable storage solution.

In this architecture, each component communicates with the others through well-defined interfaces, allowing for efficient collaboration and easy maintenance. The use of containerization and cloud-native deployment further enhances the system's scalability and resilience.

2.1. Presentation Provider: Nginx-based Frontend

The presentation layer of our file exchange hub is built using a React web application. We utilize Nginx as a reverse proxy and web server to serve the static content generated by the React application. Nginx offers excellent performance, stability, and low resource usage, making it an ideal choice for our frontend. Furthermore, its ability to handle a large number of simultaneous connections ensures a responsive user experience, even during peak loads.

2.2. API: Express-based File Management Service

Our file management service is implemented using Node.js with the Express framework, providing a robust and scalable API for managing file metadata and coordinating file transfers. This service is responsible for handling user authentication, managing file metadata, and communicating with the File Gateway service for actual file uploads and downloads. Express is chosen for its ease of use, extensive documentation, and compatibility with Node.js, which allows for efficient development and seamless integration with other components of the File Exchange Hub.

2.3. Database: MongoDB

MongoDB, a NoSQL database, is utilized to store information about rooms, files, and users. Its flexible schema and horizontal scalability makes it a fitting choice for our cloud-native application, as it can easily adapt to changing requirements and handle large volumes of data. Additionally, MongoDB provides a rich query language and high-performance indexing capabilities, allowing for efficient data retrieval and manipulation.

2.4. Asynchronous Message Communication Server: Socket.IO

To enable real-time communication and notifications within our file exchange hub, we employ Socket.IO, a popular li-

brary for real-time web applications. This server facilitates instant messaging between users in the same room, ensuring a seamless and interactive experience for all participants. Socket.IO is selected for its flexibility, reliability, and compatibility with various platforms and transports, making it a powerful solution for real-time communication in our application.

2.5. File Gateway: Node.js-based File Streaming Service

The File Gateway service is a Node.js-based application designed to efficiently stream files between clients and the cloud storage service. It acts as an intermediary, managing incoming file requests and delivering the requested files to the clients in real-time. This service ensures optimal performance and reduced latency by employing features such as caching, parallel streaming, and buffering. The File Gateway is built using Node.js for its high-performance capabilities, non-blocking I/O model, and ease of integration with other components, making it an ideal choice for this crucial aspect of the File Exchange Hub.

2.6. Cloud Storage Bucket: Azure Blob Storage

To store and manage the actual files exchanged by users, we leverage Azure Blob Storage. This fully managed object storage service offers high availability, durability, and scalability, allowing our file exchange hub to efficiently store, retrieve, and share files. The choice of Azure Blob Storage is driven by its integration with other Azure services, its security features, and its cost-effective pricing model, making it a suitable solution for our application's file storage needs.

3. Scenarios

In this section, we will discuss the two scenarios in which our application can be set up and run: the local scenario and the production scenario. Each scenario has its own architecture and communication patterns between the different services. Understanding these scenarios helps in setting up, testing, and deploying the application effectively.

3.1. Local scenario

In the local scenario, all components are hosted and run locally on the developer's machine. The following diagram (Figure 1) depicts the relationships between the components:

In this setup, the Presentation Provider communicates directly with the Express File Management Service, Asynchronous Message Communication Server, and File Gateway Service, which are all running on the same local machine. The Express File Management Service is responsible for user authentication and managing file metadata. The Asynchronous Message Communication Server handles real-time communication, and the File Gateway Service manages the actual file uploads and downloads by interacting with the local storage. Local provisioning is performed using both Docker Compose and Terraform, streamlining the process of setting up the local environment.

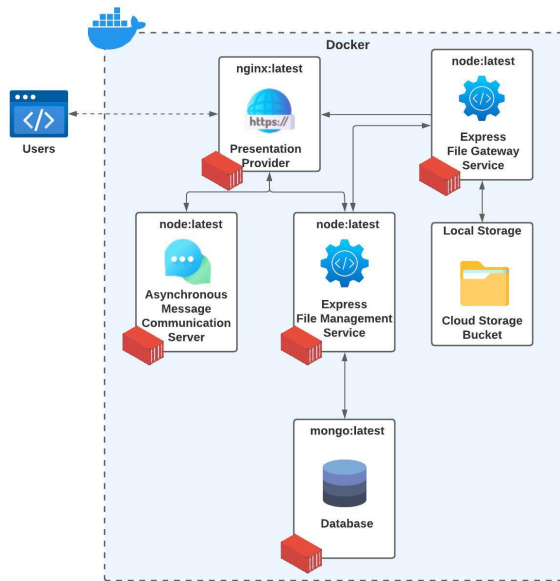


Figure 1: Local system architecture diagram.

3.2. Production scenario

In the production scenario, the services are deployed on Google Cloud Run, and the storage is handled by Google Cloud Storage. The following diagram (Figure 2) depicts the relationships between the components:

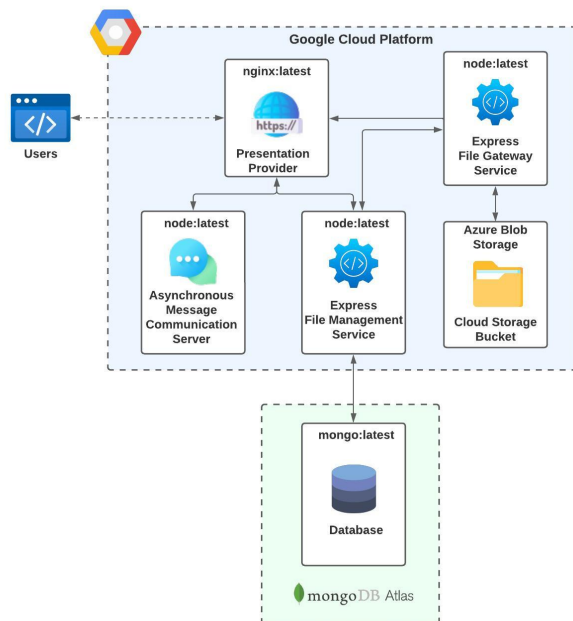


Figure 2: Production system architecture diagram.

In this setup, the Presentation Provider communicates with the Express File Management Service, Asynchronous Message Communication Server, and File Gateway Service

through their respective exposed endpoints. The Express File Management Service is responsible for user authentication and managing file metadata. The Asynchronous Message Communication Server handles real-time communication, and the File Gateway Service manages the actual file uploads and downloads by interacting with the Google Cloud Storage. This architecture provides better scalability and reliability, as each service can be independently managed and scaled in response to the system's demands. The file bucket is not public; only users with access to the file download URL can access it, which happens if the URL is shared on purpose or someone has access to the Room within the File Exchange Hub platform. For privacy reasons and reduced allocated space, files inserted in the file bucket have a lifetime of 1 day, after which they are automatically deleted by the settings defined when creating the bucket. Additionally, file records in the MongoDB have an expiration time associated with them, meaning that these expired files can no longer be consulted.

4. Provisioning

The provisioning process is designed to ensure a smooth transition from development to production, leveraging the power of various Google Cloud Platform (GCP) services. The provisioning strategy uses GCP's Cloud Run, Google Cloud Storage, Google Cloud DNS, and Cloud Build services, along with GitHub Actions, Docker Hub, and Terraform, for automating and streamlining the process. Figure 3 presents a diagram of the provisioning process, illustrating the steps and relationships between each involved service.

1. GitHub Actions: This service automates the build and provisioning of the application components. It is triggered by a push to the repository, builds the Docker images, and pushes them to Docker Hub.
2. Terraform: Terraform applies the infrastructure changes and triggers Cloud Run to pull Docker images from Docker Hub and deploy the containerized services, as well as creating the necessary Google Cloud Storage buckets and Google Cloud DNS configurations.
3. GCP Services: A combination of Google Cloud Run, Google Cloud Storage, and Google Cloud DNS services are used to deploy, store, and map the application components to the custom domain.

Figure 3 depicts the provisioning process and the relationships between the components, illustrating the following flow: (1) The developer commits and pushes changes to the repository, (2) which triggers GitHub Actions to (3) build and push Docker images to Docker Hub. (4) Terraform is then initiated, applying infrastructure changes, (5) retrieving and writing the Terraform state to the cloud storage bucket as the remote backend for persistence and collaboration, (6) subsequently triggering Cloud Run, Google Cloud Storage, and Google Cloud DNS configurations. (7) Cloud Run pulls images from Docker Hub, (8) deploys the containerized services, and (9) maps the Presentation Provider service to the domain <https://fileexchange.com>.

The provisioning process ensures that the application components are consistently built and deployed across environments, minimizing human error and facilitating continuous integration and delivery.

4.1. Automated Continuous Provisioning with GitHub Actions and Terraform

In order to streamline the provisioning process and ensure a consistent and efficient release cycle, we implemented Continuous Provisioning using GitHub Actions and Terraform. GitHub Actions automates the process of building and pushing Docker images to Docker Hub, while Terraform manages the infrastructure required to deploy and update the services on Google Cloud Run, create Google Cloud Storage buckets, and configure Google Cloud DNS.

When changes are pushed to the repository, GitHub Actions triggers an automated workflow that builds and pushes the updated Docker images for each microservice to Docker Hub. Following the successful completion of this step, the workflow then initiates the Terraform CLI to manage the provisioning of the microservices on Google Cloud Run, as well as creating the necessary Google Cloud Storage buckets and Google Cloud DNS configurations.

Using Terraform, we define the infrastructure for each microservice as code, including Google Cloud Run services, IAM policies, Google Cloud Storage buckets, and Google Cloud DNS configurations. This approach enables us to maintain a version-controlled and easily auditable infrastructure while also allowing for seamless updates and configuration changes. By integrating with GitHub Actions, we ensure that every push to the repository automatically triggers a new provisioning, resulting in a streamlined and consistent release process.

4.2. Google Cloud Platform: Scalable Cloud Services

Google Cloud Platform (GCP) provides a suite of cloud services that enable developers to build, deploy, and scale applications efficiently. For this project, we leveraged several GCP services, including Google Cloud Run, Google Cloud Storage, and Google Cloud DNS.

4.2.1. Cloud Run: Scalable Containerized Services

Google Cloud Run is a serverless platform that enables the deployment and management of containerized applications. By leveraging the power of containers, Cloud Run allows developers to focus on writing code without worrying about the underlying infrastructure. The platform automatically scales applications based on demand, ensuring optimal resource usage and minimizing costs.

4.2.2. Google Cloud Storage: Storing Files and Terraform State

Google Cloud Storage is a highly scalable and durable object storage service, which we used to store the uploaded files from users and the Terraform state. By storing the Terraform state in a cloud storage bucket, we enable team mem-

bers to collaborate on infrastructure changes and maintain a persistent and version-controlled state.

4.2.3. Google Cloud DNS: Custom Domain Configuration

Google Cloud DNS is a high-performance, resilient, and global DNS service that enables the mapping of the Presentation Provider service to the custom domain <https://fileexchange.com>. By utilizing Google Cloud DNS, we can manage DNS records programmatically using Terraform, ensuring that our domain configurations are version-controlled and easily updatable.

The combination of these GCP services allows us to build a scalable, resilient, and efficient file exchange system that demonstrates the power and flexibility of modern cloud computing technologies for software applications.

5. Discussion

Throughout the development of our decentralized file exchange hub, we encountered various challenges and gained valuable insights into deploying applications both locally and in the cloud. In this section, we will discuss the different scenarios, difficulties faced during provisioning, and offer recommendations for other developers embarking on similar projects.

Comparing the local and production scenarios, we noticed that while local deployment is more straightforward to set up and manage, it lacks the scalability and reliability offered by cloud deployment. In the local scenario, all components run on the developer's machine, which simplifies communication between services but limits the system's ability to handle high loads and provide high availability. On the other hand, deploying our application to the cloud using Google Cloud Platform services, such as Cloud Run, Cloud Storage, and Cloud DNS, enables us to achieve better scalability and reliability while maintaining a modular architecture. This approach allows each service to be independently managed, scaled, and updated, thus improving the system's overall resilience and flexibility.

During the provisioning process, we faced several challenges, including configuring custom domain mapping, setting up Google Cloud Storage buckets for storing uploaded files and Terraform state, and ensuring security across all components. To overcome these challenges, we relied heavily on documentation, best practices, and online resources, which proved invaluable in guiding our development efforts. Additionally, we learned the importance of thoroughly testing each service independently and as a part of the larger system to ensure seamless integration and functionality.

6. Conclusion

This report has presented a decentralized file exchange hub designed with a cloud-native approach, addressing the limitations of traditional centralized solutions. By leveraging containers, microservices, and cloud services, we have

created a scalable, resilient, and efficient system for exchanging files. Our use of modern technologies such as Nginx, Express, Socket.IO, MongoDB, and Google Cloud Platform services demonstrates the power and flexibility of cloud computing for modern software applications, enabling a more decentralized and secure file exchange experience.

Additionally, the integration of GitHub Actions, Terraform, and various Google Cloud Platform services has streamlined the provisioning process, providing a robust Continuous Integration and Continuous Deployment (CI/CD) pipeline for ensuring consistent and reliable application delivery. Our implementation of custom domain mapping, Google Cloud Storage for user file storage and Terraform state persistence, and Cloud DNS for domain management showcases the power of utilizing comprehensive cloud services to improve the user experience and simplify infrastructure management.

The decentralized file exchange hub showcases the potential for creating high-performance, scalable, and flexible systems using cutting-edge cloud technologies and best practices.

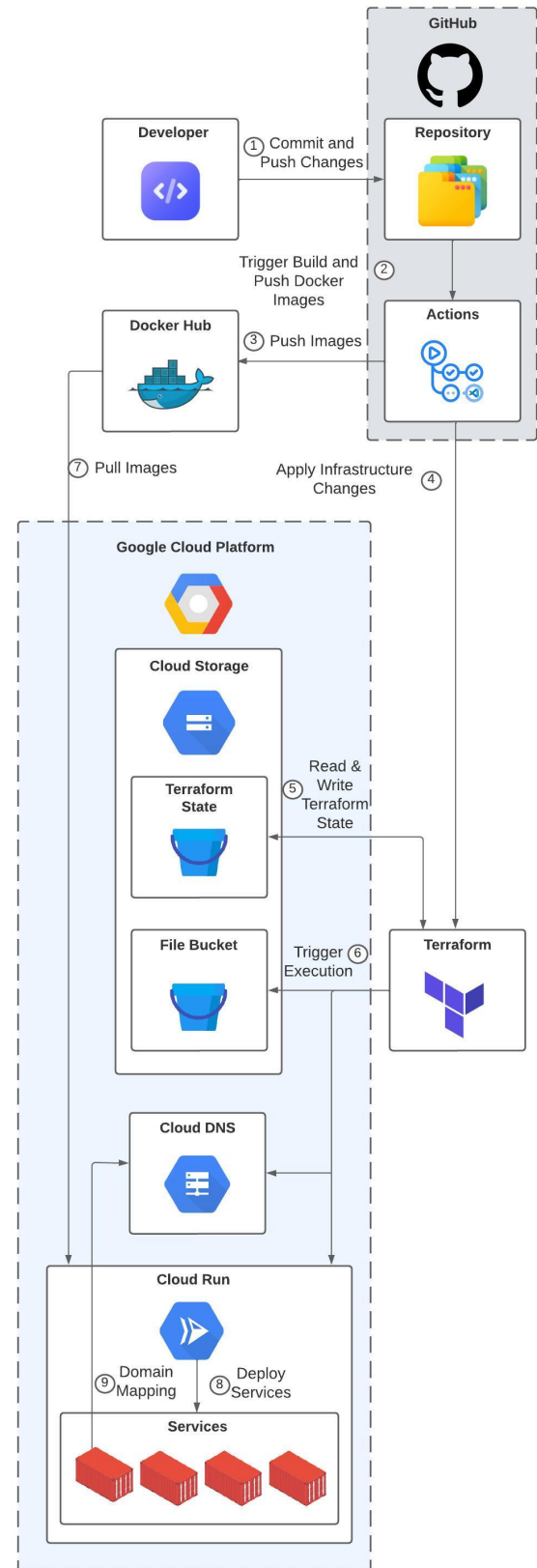


Figure 3: Production provisioning process diagram.