MANAGED CLOUD HOSTING PLATFORM

Your Ideas, Our Platform: Infinite Possibilities...

Supervised by Dr. Nesma Rezk

Mentoring Company Microsoft

Submitted By

Ahmed Atwa

Somaya Ayman

Nassar Khaled

Abdelrahman El Sayed

Ahmed Ayman

Mennatallah Amr

Sameh Ossama

Fady Adel

Sara Ashraf

Ahmed Sherif

Zeyad Waleed

Mentored by

Eng. Kerollos Magdy

Eng. Yassin Chaddad

Faculty of Engineering

Department of Computer Engineering

Ain Shams University 10/2025

Table of Contents

Abstract

Our project aims to develop a cloud-based Platform as a Service (PaaS) solution tailored for developers in Egypt and beyond. The platform will allow users to deploy applications without the need to manage underlying infrastructure, offering both static and dynamic hosting capabilities. We are building this PaaS on top of existing Infrastructure as a Service (IaaS) offerings, with a vision to eventually move towards a fully Egyptian infrastructure.

This project aspires to provide a comprehensive developer experience comparable to leading PaaS providers like Netlify and Heroku, offering features such as GitHub integration, multi-branch deployment, CI/CD, automatic scaling, and more. Our platform will focus on being cloud-agnostic, ensuring that it can operate independently of specific cloud providers and can easily migrate between infrastructures.

The final deliverable will be a fully operational PaaS platform with key features for both static and dynamic hosting, supported by a user-friendly dashboard and comprehensive documentation to guide developers in using the service.

Key elements of the project:

- **Core Hosting Features:** Static and dynamic hosting, designed to meet modern web application needs.
- **Platform Flexibility:** Cloud-agnostic architecture that can be migrated across infrastructures.
- **Developer Tools:** CI/CD, GitHub integration, version control, scaling, and performance monitoring.
- **User Focus:** A simple user dashboard for application management and an admin dashboard for system monitoring.
- **Future Aspirations:** Lay the groundwork for a fully Egyptian PaaS infrastructure, building upon this initial phase.

Background

Cloud Computing

Cloud computing is the delivery of computing services—such as servers, storage, databases, networking, software, and analytics—over the internet ("the cloud"). It enables users to access and manage these resources on-demand without needing to own or maintain physical infrastructure.

Key Characteristics

- **On-Demand Self-Service**: Users can provision resources as needed.
- **Broad Network Access**: Services are accessible over the internet from any location.
- **Resource Pooling**: Computing resources are shared among multiple users.
- **Scalability**: Resources can scale up or down automatically.
- **Pay-as-You-Go Pricing**: Users only pay for what they use.

Current Landscape of Cloud Computing and Hosting Services

The cloud computing industry has seen rapid evolution and widespread adoption in recent years. Today, businesses of all sizes are leveraging cloud services to enhance operational efficiency, scalability, and cost-effectiveness. The primary drivers behind this shift include the increasing demand for remote work capabilities, the need for scalable resources, and the desire to reduce capital expenditure on IT infrastructure.

Cloud computing is broadly categorized into several service models:

- **Infrastructure as a Service (IaaS)**: Provides virtualized computing resources over the internet, such as virtual machines, storage, and networking.
- **Platform as a Service (PaaS)**: Delivers a platform allowing customers to develop, run, and manage applications without dealing with the underlying infrastructure.
- **Software as a Service (SaaS)**: Offers software applications over the internet, on a subscription basis, eliminating the need for installation and maintenance on local devices.

Managed Cloud Hosting

Managed Cloud Hosting is a cloud computing model that provides a comprehensive platform for developers to build, deploy, and manage applications. It abstracts and manages the underlying infrastructure—such as servers, storage, and networking—allowing developers to focus solely on the application itself.

Key features of Managed Hosting include:

1- **Development Tools**: Integrated development environments, databases, and development frameworks that streamline the development process.
2- **Scalability**: Automatic scaling to handle varying workloads without manual intervention.
3- **Middleware**: Software that connects different applications or services, facilitating communication and data exchange.
4- **Managed Services**: Services such as database management, application hosting, and monitoring are managed by the PaaS provider, reducing the operational burden on developers

Types of Managed Hosting Services

There are several types of managed hosting services, each designed for specific use cases:

1- **Static Hosting:** Provides high-performance hosting for static websites, ensuring quick load times and reliability.

2- **Dynamic Hosting:** Enables easy deployment of applications by managing the runtime environment and associated services.

3- **Serverless Functions:** Allows code execution in response to events without managing servers, simplifying application development.

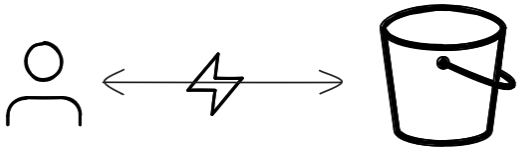
4- **Container as a Service (CaaS):** Manages and orchestrates containerized applications, offering a scalable and flexible environment.

5- **Database Hosting:** Delivers managed database services with features like automated backups, scaling, and high availability.

Static Hosting

Static Hosting refers to the process of hosting static assets—such as HTML files, CSS, JavaScript, and images on a server for fast and reliable delivery to users. Static content is pre-rendered, meaning it does not require server-side processing, which makes it ideal for high-performance websites.

Static Hosting services deliver content through high-performance servers, utilizing Content Delivery Networks (CDNs) to cache and serve files from locations closest to users.

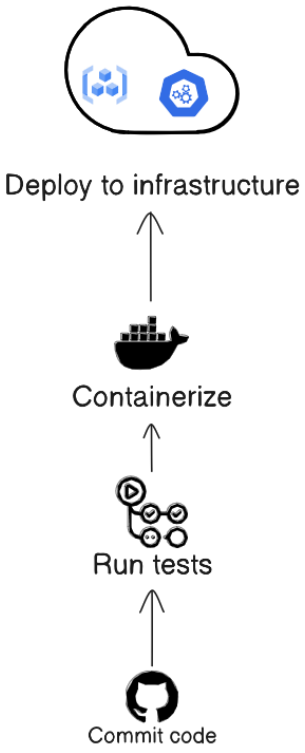


Static Hosting is ideal for:

- Personal blogs and portfolios: These sites are typically static and do not require dynamic content generation.
- Company landing pages and marketing sites: These pages benefit from fast load times and high availability.
- Documentation sites: Static hosting is well-suited for sites where content doesn't change frequently.

Dynamic Hosting

Dynamic Hosting, also known as Platform as a Service (PaaS), provides an environment for deploying and running applications without the need to manage underlying infrastructure. It automates the creation of runtime environments, handling everything from scaling to environment management, and also includes DevOps services like Continuous Integration and Continuous Deployment (CI/CD).



Our Dynamic Hosting service automates the deployment process. Developers upload their code, and our platform automatically creates the necessary environment, including runtime, libraries, and dependencies. The platform handles the infrastructure, scaling, and updates, allowing developers to focus solely on writing code.

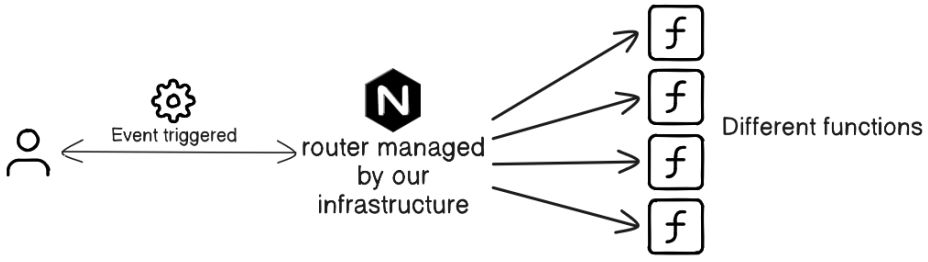
Dynamic Hosting is ideal for:

- Web applications: Deploy and scale web apps without worrying about server management.
- APIs and microservices: Easily manage and scale backend services.
- Prototyping: Quickly launch and test new applications without investing in infrastructure.
- Continuous Integration/Continuous Deployment (CI/CD): Automate the build, test, and deployment process to accelerate development and delivery.

Serverless Functions

Serverless Functions allow developers to run code without provisioning or managing servers. This architecture abstracts server management, allowing developers to focus solely on writing code that responds to specific events or triggers.

With Serverless Functions, developers upload individual functions or small pieces of code. These functions are executed automatically in response to events such as HTTP requests, database changes, or message queues. The platform manages the infrastructure, automatically scaling the function execution to meet demand.



Serverless Functions are ideal for:

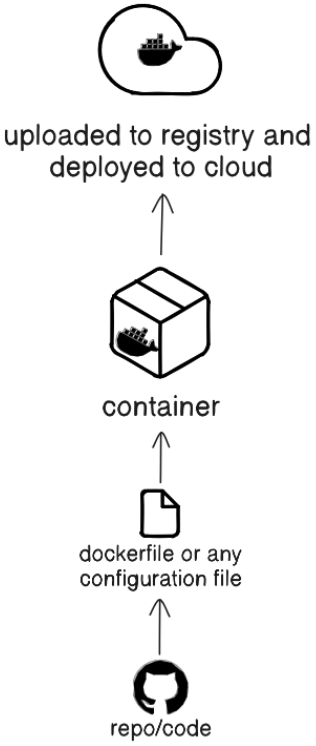
- Microservices: Break down applications into small, independent functions.
- Event-Driven Applications: Trigger functions in response to events like user actions, data changes, or scheduled tasks.
- APIs and Backend Logic: Easily deploy functions to handle backend processes without managing servers.

- Rapid Prototyping: Quickly deploy and test new functionality without infrastructure setup.

Containers as a Service (CaaS)

Container as a Service (CaaS) is a cloud service that allows developers to deploy, manage, and scale containerized applications using container orchestration tools. CaaS simplifies container management by providing a platform that handles the underlying infrastructure, networking, and scalability.

With CaaS, developers package applications and their dependencies into containers. These containers are then deployed to the CaaS platform, where they are automatically managed and orchestrated. The platform ensures efficient resource utilization, scales containers as needed and handles networking between containers.



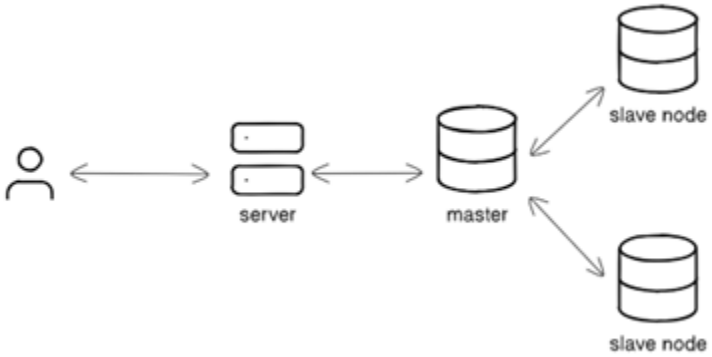
CaaS is ideal for:

- Microservices Architecture: Deploy and manage microservices in isolated containers.
- Development and Testing: Create consistent environments for development, testing, and production.
- Hybrid Cloud Deployments: Run containerized applications across multiple cloud environments seamlessly.
- Continuous Integration/Continuous Deployment (CI/CD): Integrate with CI/CD pipelines to automate the deployment of containerized applications.

Database Hosting

Database Hosting provides managed database services, allowing you to store, manage, and retrieve data efficiently without worrying about the underlying infrastructure. Our platform supports multiple database types, including SQL, NoSQL, and in-memory databases, tailored to fit various application needs.

Our Database Hosting service offers automated provisioning, scaling, and maintenance of databases. Once a database is created, the platform handles backups, updates, and scaling based on demand. Developers can focus on application logic while the platform ensures data is available, secure, and performant.



Database Hosting is ideal for:

- Web and mobile applications: Store and manage user data, content, and application settings.
- Analytics and reporting: Perform data analysis and generate reports using scalable database solutions.
- E-commerce: Manage product inventories, customer information, and transaction records.
- Real-time applications: Use in-memory databases for fast, low-latency data access in real-time applications.

Project Objectives

Primary Objective:

Our main goal is to develop a comprehensive Platform as a Service (PaaS) solution that simplifies the deployment and management of applications. By enabling developers to deploy applications without having to handle the underlying infrastructure, we aim to provide a feature-rich platform that addresses the diverse needs of modern development workflows, ensuring ease of use, scalability, security, and affordability.

Specific Objectives:

Core Services Offering:

- Static Hosting:
  - Provide features like manual file upload, GitHub integration, support for frontend frameworks, CDN integration, and previewing pull requests.
  - Enable easy deployment of static websites with CI/CD pipelines, automatic scaling, and SSL certificates.
- Dynamic Hosting (PaaS):
  - Offer a flexible environment for deploying dynamic applications with multi-language support, manual uploads, and GitHub integration.
  - Include features like rollback to previous deployments, independent version deployment with adjustable scaling, and traffic splitting between versions.
  - Leverage CI/CD for seamless deployments and updates.

Supporting Features Across All Services:

- Integrate key features such as continuous integration and delivery (CI/CD), SSL certificates, multi-branch deployment, and automatic scaling to ensure smooth workflows and secure hosting across all core services.

Extra Services (Planned for Future):

- Serverless Functions, Container as a Service (CaaS), and Database Hosting:
  - Though we have not fully detailed the implementation plan for these components, we aim to explore their inclusion based on time constraints. We recognize their importance for developers needing flexibility, scalability, and efficient resource management.

Building on IaaS Infrastructure:

- Initially, the PaaS layer will be developed on top of established IaaS infrastructure from leading cloud providers such as AWS, Google Cloud Platform (GCP), or Microsoft Azure. The decision will depend on sponsorship availability.
- In the future, our vision is to transition towards building on a fully Egyptian infrastructure to localize and enhance cloud computing services for the region.

Project Relevance

During market research, few gaps in the Egyptian cloud services market found out including the following:

1- **Absence of local providers:** Many regions, including Egypt, don't have local cloud service providers. This can lead to fewer choices for businesses and higher costs due to the reliance on international providers.

2- **Payment in Local Currency**: Many cloud services are billed in foreign currencies, such as USD, which can be problematic for local businesses due to fluctuating exchange rates and additional transaction fees. By allowing payments in Egyptian Pounds (EGP), the Managed Hosting project simplifies financial transactions, making cloud services more accessible and predictable for Egyptian companies.

3- **Scalability for SMEs**: Small and medium-sized enterprises (SMEs) might face challenges accessing scalable cloud services that fit their budget. Offering scalable solutions tailored to the needs of SMEs could fill this gap.

The project aims to address the growing need for a comprehensive cloud hosting solution and contribute to the Egyptian economy by providing the following:

1- **Integrated services:** Providing a managed platform that combines static hosting, dynamic hosting, serverless functions, container management, and database hosting as solutions to keep up with the growing needs of developers and businesses.

2- **Reduced Costs:** Reducing costs associated with data transfer and international billing, potentially offering more competitive pricing for services.

3- **Encouraging Startups:** Affordable and accessible local PaaS solutions can lower the barrier to entry for startups, providing them with the tools needed to develop and scale their applications without heavy upfront investments.

4- **Job creation:** Establishing and growing the project can create job opportunities within the tech industry, contributing to economic growth in Egypt

5- **Flexibility and customization:** The managed hosting platform as local PaaS provider will be more flexible and willing to customize its offerings to meet specific needs of local businesses and developers, providing more tailored solutions than global providers.

Project Vision

Long-term Vision:

Our long-term goal is to create a **homegrown Platform-as-a-Service (PaaS)** solution that not only supports local innovation but also reduces dependency on foreign technologies. We aim to foster the growth of the Egyptian tech ecosystem by providing a platform entirely built in Egypt. As local infrastructure, such as data centers, becomes available, we plan to source the entire PaaS solution domestically. This will enable us to deliver a **truly independent, secure, and sustainable** cloud platform for businesses and developers throughout the region.

Future Aspirations:

Looking ahead, our goal is to expand into global markets, positioning our PaaS as a leading provider in the Middle East and beyond. We plan to integrate advanced AI-driven features to enhance automation, scalability, and performance, while pushing the boundaries of DevOps automation to streamline workflows and improve deployment efficiency for businesses and developers worldwide. While the initial phase focuses on building the PaaS layer on top of existing IaaS infrastructure like AWS, Azure, or GCP, our long-term vision is to transition to a fully Egyptian infrastructure. By ensuring our platform remains cloud-agnostic, we aim to facilitate seamless migration to local cloud providers in the future, further solidifying our role as a localized yet globally relevant solution.

Feasibility Study

Market Analysis:

Egypt's market for cloud services is growing rapidly due to several key factors. The country is undergoing significant digital transformation, driven by the **Egypt Vision 2030** initiative, which emphasizes digitizing services across sectors such as government, healthcare, banking, education, and retail. This transformation has created a demand for scalable and secure cloud-based solutions that can support this digital growth.

The startup ecosystem in Egypt is thriving, with many businesses in fintech, e-commerce, and software development and significant challenge they commonly face is the **foreign currency issue**. Many global cloud services require payment through **dollar-based Visa cards**, which can be a hurdle for businesses that do not have easy access to dollar-denominated bank accounts. We can bridge this gap by integrating with **Egyptian payment systems**, allowing businesses to pay using **local currency**, making it easier for companies to use cloud services without dealing with exchange rates or foreign currency restrictions.

Competitor Analysis:

- Heroku is a widely used PaaS known for its ease of use, seamless GitHub integration, automatic deployments, and scalability through its "Dynos" system. It also provides add-ons and resource monitoring, making it versatile but somewhat expensive for long-term use.
- Railway stands out with its aesthetically pleasing UI, infrastructure visualizer, and flexibility to deploy custom Docker containers. It allows multiple environments per project, offers team collaboration, and integrates with GitHub and Vercel for PR environments.
- Vercel is ideal for serverless web applications and provides advanced features like server-side rendering (SSR), static site generation (SSG), incremental static regeneration (ISR), edge functions, and image optimization. Vercel excels in deployment automation and performance through its global CDN, while also offering strong team collaboration features.

- Netlify focuses on static site hosting with advanced features like pre-rendering for SEO, seamless GitHub integration, serverless functions, automatic build and deployment for branches, edge functions, and PR deploy previews. It offers tools for both front-end developers and full-stack applications with a focus on serverless infrastructure.

Key Comparisons:

Deployment Flexibility:

- **Heroku:** Automatic deployment on GitHub push, buildpacks for multiple languages, Dynos for scalable workloads.
- **Railway:** Custom Docker containers, webhook-triggered builds, GitHub PR environments, flexibility to deploy non-standard workloads.
- **Vercel:** Static site generation (SSG), server-side rendering (SSR), incremental static regeneration (ISR), and GitHub integration for automatic deployment.
- **Netlify:** Automatic build and deployment on GitHub push, serverless functions, edge functions, PR deploy previews, and control via the netlify.toml file for custom redirects and rules.

Collaboration Features:

- **Heroku:** Teams and pipelines for collaboration, role-based access control.
- **Railway:** Team management, multiple environments per project, and seamless GitHub integration for collaborative PR environments.
- **Vercel:** Strong team collaboration tools, including Teams for shared access, role-based permissions, and deployment previews for every pull request (PR). Allows multiple team members to review and comment on deployments.
- **Netlify:** PR deploy previews with collaboration tools such as screenshots, video recording, drawing, and direct issue creation on GitHub. These tools enhance real-time collaboration during the deployment review process.

Customization & API Integrations:

- **Heroku:** Add-ons and APIs to extend app functionality, but limited customization for custom deployment workflows.
- **Railway:** Custom API and webhook features, integration with GitHub for PR deployments, and flexibility in deploying custom services.
- **Vercel:** Provides serverless API routes, edge functions, and integrations with services like Vercel KV, Vercel Blob, and Vercel Postgres. Custom build processes allow for complex setups.
- **Netlify:** Serverless functions, cron jobs, form handling, API integrations, and pre-rendering for improved SEO. Offers extensive customizations through the netlify.toml file for redirects and build processes.

Resource Monitoring:

- **Heroku:** Provides resource monitoring tools like real-time logs, metrics, and add-ons for enhanced infrastructure insights.
- **Railway:** Built-in resource monitoring with basic performance metrics, though not as advanced as dedicated monitoring platforms.
- **Vercel:** Offers analytics for monitoring performance, build times, and user behavior, with additional insights via Vercel Analytics.
- **Netlify:** Offers performance analytics for websites but lacks more granular resource monitoring tools found in Heroku or Railway.

Technical Feasibility

This project will be built on existing IaaS (Infrastructure as a Service) platforms like AWS, Azure, or Google Cloud (GCP). By leveraging these well-established infrastructures, we will focus on developing the unique features of our PaaS, such as localization, integration with local payment systems, and providing all the essential tools needed to make hosting your platform seamless and enjoyable. Using IaaS allows us to scale effortlessly, pay only for the resources we use, and deliver a cost-effective, flexible solution. While tailored to the specific needs of businesses in Egypt, this platform will also be versatile and accessible for users anywhere in the world.

Project Scope

Scope Definition:

The scope of this project focuses on building a cloud-agnostic PaaS layer on top of an existing IaaS infrastructure (AWS, GCP, or Azure). Our goal is to minimize reliance on specific cloud provider features, allowing the platform to be easily migrated to other infrastructures in the future.

Deliverables:

By the conclusion of the project, we plan to deliver:

- **User Dashboard:** An interface allowing users to manage and deploy their applications.
- **Admin Dashboard:** A monitoring system for administrators to track platform health and user activities.
- **Landing Page:** A marketing site to explain the PaaS platform's features to potential users.
- **Backend API:** The system's backend API, connecting the frontend, the database, and the infrastructure.
- **System Platform and Infrastructure**: The system's core infrastructure that hosts the applications and performs the required operations.
- **Documentation Website (Bonus):** A guide for users, detailing how to use the platform's services and deploy applications.
- **Final Technical Documentation:** Detailed technical documentation outlining the architecture, components, and system operation, delivered to project supervisors.
- **Demo:** A final demo of the system, including static hosting, dynamic hosting, and any bonus features that were completed.

System Requirements

As part of the risk management, we will divide the system features into 2 sets: Minimum required features, and bonus features. We can guarantee that the minimum required features will be delivered in the final demo, and we will add from the bonus features as time allows.

Minimum Features

Static Hosting Features

- Manual upload
- Integration with GitHub
- Building frontend frameworks

Dynamic Hosting Features

- Manual upload
- Integration with GitHub

- Support for 1 language
- Health monitoring and auto-healing

Shared Hosting Features (Static and Dynamic Hosting)

- Automatic CI/CD from GitHub repository
- Multi-branch deployment
- Automatic scaling

System Non-functional Features

- High availability
- Fault tolerance
- Security
- Performance

Bonus Features

Static Hosting Features

- Pre-rendering
- Pull request previews
- Custom configuration for redirects and rewrites
- Managed form handling

Dynamic Hosting Features

- Support for more languages
- SSL Certificates
- Rolling back
- Scaling down to 0
- Independent version deployment and scaling
- Support for stateful apps

Shared Hosting Features

- Allow custom domains
- Manage multiple environments
- Logging of CI/CD
- Adding environment variables and secrets
- Custom routing and unifying services under 1 URL

Container as a Service

Run containerized applications on the platform.

Serverless Functions

Support for deploying event-driven serverless functions.

Command-line Interface tool

Build a CLI for users to interact with the platform directly from their terminal.

Project Plan

Milestones

Preliminary Research Milestone

- **Competitor Analysis:** Conduct an analysis of existing PaaS solutions to identify strengths, weaknesses, and opportunities for differentiation.
- **Gathering Features List:** Gathering all features present in competitors to define the features we will implement.
- **Field Research:** Explore topics in web development, DevOps practices, and system design to gather insights and best practices relevant to the project.
- **Early Design Phase:** Create initial designs to visualize the overall system architecture.

Static Hosting Milestone

- **Research Phase:** Investigate available technologies and approaches for implementing static hosting features, including performance optimizations and best practices.
- **Proof-of-Concept Phase:** Develop small-scale prototypes to demonstrate key static hosting features such as manual uploads, GitHub integration, and front-end framework support.
- **Implementation Phase**: Develop and integrate the static hosting features into the platform.
- **Deliverables Update:**
    - User dashboard updates for static hosting management.
    - Integration of static hosting functionalities.

Dynamic Hosting Milestone

- **Research Phase:** Explore various options for dynamic hosting solutions, including language support, environment management, and deployment strategies.
- **Proof-of-Concept Phase:** Create prototypes to showcase dynamic hosting capabilities like multi-language support, health monitoring, and auto-healing features.
- **Implementation Phase:** Develop and integrate the dynamic hosting features into the platform.
- **Deliverables Update:**
    - User dashboard updates for dynamic hosting management.
    - Implementation of dynamic hosting functionalities.

Testing and Optimization Milestone

- **Comprehensive Testing:** Conduct thorough testing of all implemented features for both static and dynamic hosting, including unit tests, integration tests, and user acceptance testing.

- **Bug Fixing:** Address any identified issues or bugs to ensure the platform operates smoothly and meets user expectations.
- **Optimization**: Refine the system for better performance, scalability, and security.
- **Deliverables Update:**
    - Admin dashboard development.
    - Landing page development.

Documentation Milestone

- **Technical Documentation:** Create detailed technical documentation for the project, covering architecture, system design, and implementation details.
- **User Documentation:** Develop user manuals and guides to help users navigate and utilize the platform effectively.
- **Deliverables Update:**
    - Finalizing all deliverables.
    - (Bonus) Documentation website for users.

Final Review and Demo Milestone

- **Project Review:** Conduct a comprehensive review of the project to assess progress against objectives and deliverables.
- **Demo Presentation:** Prepare and deliver a demo of the platform, showcasing key features and functionalities to stakeholders and supervisors.

Timeline

Preliminary Research Milestone

Period: 1/8/2024 – 1/11/2024

Static Hosting Implementation Phase

Period: 16/11/2024 – 27/12/2024

- Focus on implementing static hosting features like manual upload, GitHub integration, and frontend framework support.
- Deliverables:
    - User dashboard updates for static hosting management.
    - Integration of static hosting functionalities.
    - Testing and initial optimization for static hosting.

Dynamic Hosting Research, Proof-of-Concept and Implementation Phase

Period: 1/2 – 14/3

- Conduct research for dynamic hosting (language support, environment management, etc.).

- Build proof-of-concept for features like multi-language support and health monitoring.
- Implement dynamic hosting features such as GitHub integration, rollback capabilities, and scaling.

Dynamic Hosting Implementation Phase Continue

Period: 29/3 – 16/5

- Finish dynamic hosting.
- Deliverables:
    - User dashboard updates for dynamic hosting management.
    - Complete dynamic hosting functionalities.
- Testing and optimization for dynamic hosting.

Testing and Optimization Phase

Period: 29/3 – 16/5 (continued)

- Comprehensive testing of static and dynamic hosting features.
- Bug fixing and overall platform optimization.
- Adding bonus features as time permits.

Documentation

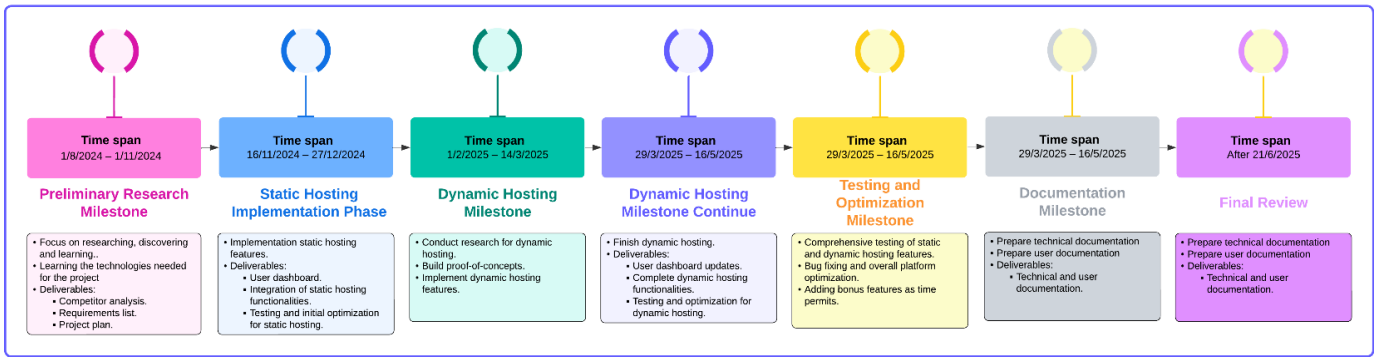
Period: 29/3 – 16/5 (continued)

- Prepare technical documentation covering system design, architecture, and implementation.
- Prepare user documentation for platform use, dashboard functionality, and features.
- Deliverables:
    - Technical and user documentation.
    - (Bonus) Documentation website for users.

Final Review

Period: After 21/6

- Finalize technical and user documentation.
- Conduct the final project review and demo preparation.



Project Management

Our project management approach was centered around **GitHub Issues and GitHub Projects**, ensuring structured backlog management, prioritization, and task tracking. We operated in **iterations**, with two-week cycles where we planned, executed, and reviewed progress. This structured approach allowed us to continuously evaluate our progress, iterate on feedback, and make necessary adjustments to ensure smooth project execution.

Team Structure & Leadership

Dynamic Team Organization

- The team consisted of **11 members**, dynamically assigned across three primary sub-teams:
    - **Frontend Team** – Responsible for the user interface, dashboards, and user experience, ensuring a seamless and intuitive interaction for end users.
    - **Backend Team** – Developed APIs, application logic, and database interactions, maintaining the core logic and scalability of the platform.
    - **DevOps Team** – Handled infrastructure, CI/CD pipelines, deployment, and system reliability, making sure the system was scalable and efficient.

Leadership & Decision-Making

- Each team had a **team leader**, responsible for guiding discussions, reviewing contributions, and ensuring the team remained aligned with project objectives.
- A **Project Leader (also the Project Manager)** served as the overall leader, making final decisions based on team input, ensuring all moving parts were synchronized and working efficiently.
- Leadership roles were **not fixed** and were reassigned as needed at the start of each iteration, allowing for flexibility in decision-making and fostering leadership development among team members.

Iteration-Based Development

Structured Workflow

- We followed a **bi-weekly iteration cycle (2 weeks per iteration)**:
    - **Sunday (Week 1) - Kickoff Meeting:** The entire team, along with our mentors, discussed goals for the iteration and decided what features to develop. During these meetings, major architectural and technical decisions were also addressed.
    - **Daily Development & Team Meetings:** Each team managed its own internal meetings independently as needed, ensuring team members were well-coordinated and aware of their tasks.
    - **Wednesday (Week 2) - Wrap-Up Meeting:** The team demonstrated completed work, shared lessons learned, and reflected on the development process, helping refine future iterations and improve efficiency.
- **Mentors attended the kickoff and wrap-up meetings** to provide guidance and share industry best practices, ensuring the project stayed on track and aligned with real-world needs.

Task Prioritization and Backlog Management

Managing Workflows in GitHub

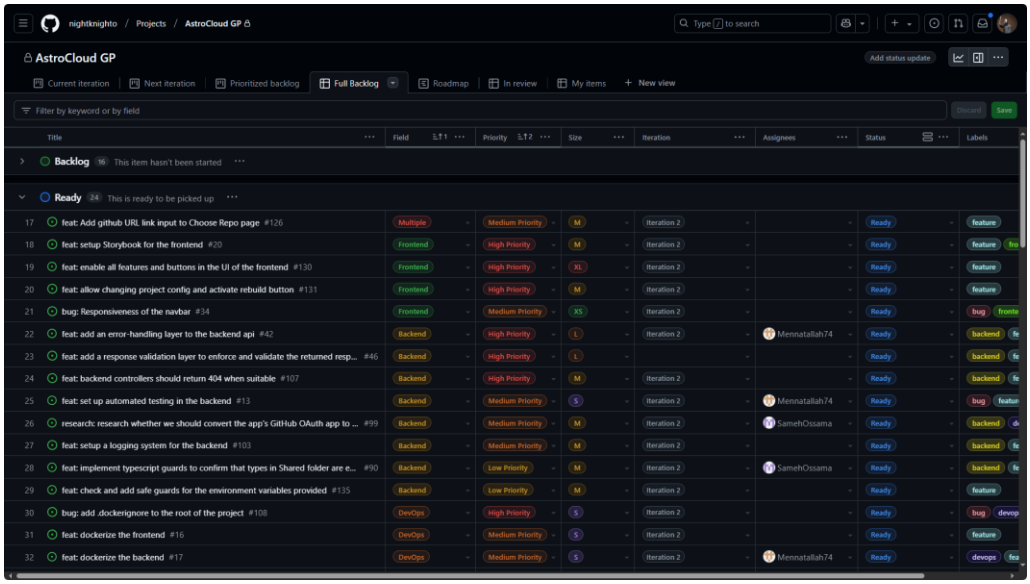
We used **GitHub Issues** to track every feature, bug, or idea. All issues were organized within **GitHub Projects**, structured into four main categories:

- **Backlog** – Newly created issues that were yet to be assigned, serving as a holding space for future tasks.
- **Ready** – Tasks selected for the current iteration, prioritized based on business needs and technical feasibility.
- **In Progress** – Tasks actively being worked on, ensuring visibility of what's currently being developed.
- **Done** – Completed and merged tasks, allowing the team to track what has been delivered.

Task Categorization & Prioritization

Each issue was assigned metadata for clarity and prioritization:

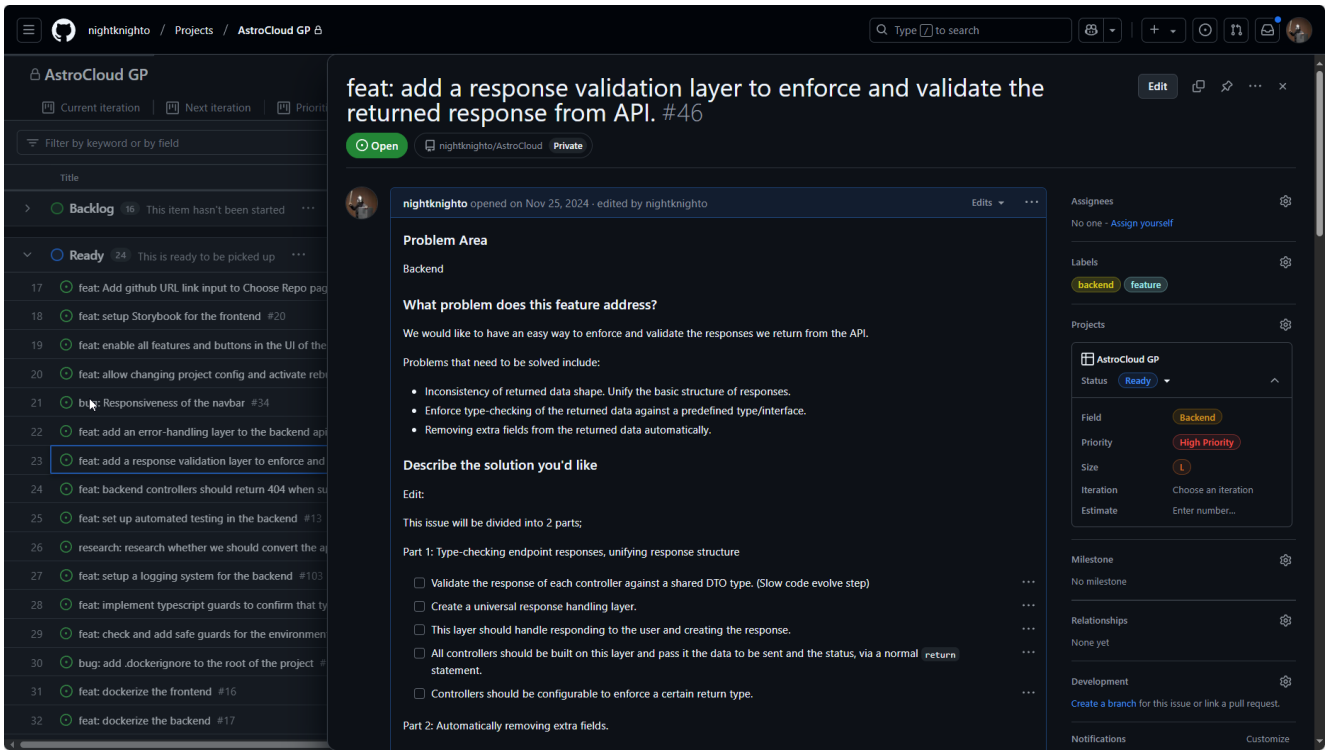
- **Field** – Categorized as Frontend, Backend, DevOps, Design, or Multiple, ensuring relevant team members could quickly identify tasks related to their area of expertise.
- **Priority** – Marked as Low, Medium, or High, allowing the team to focus on urgent and impactful tasks first.
- **Size** – Estimated effort: XS, S, M, L, or XL, providing a clear expectation of effort required for each task.
- **Iteration** – Indicating which iteration the task belonged to, ensuring better planning and accountability.
- **Assignees** – The responsible team members, providing clear ownership of tasks.



GitHub Issue Structure

Each issue was structured to ensure clarity, including the following sections:

- **Problem Area** – Specifies the field relevant to the issue (e.g., Backend, Frontend, DevOps).
- **Problem Statement** – Clearly defines the problem the feature or bug fix is addressing.
- **Proposed Solution** – Outlines how the issue should be resolved, sometimes divided into multiple implementation parts.
- **Additional Context** – Links to relevant resources, discussions, or references that help in solving the issue.



Task Execution & Tracking

GitHub-Driven Workflow

Each team was responsible for **self-managing** its workflow, ensuring that members were aware of their assigned tasks. When a team member picked a task:

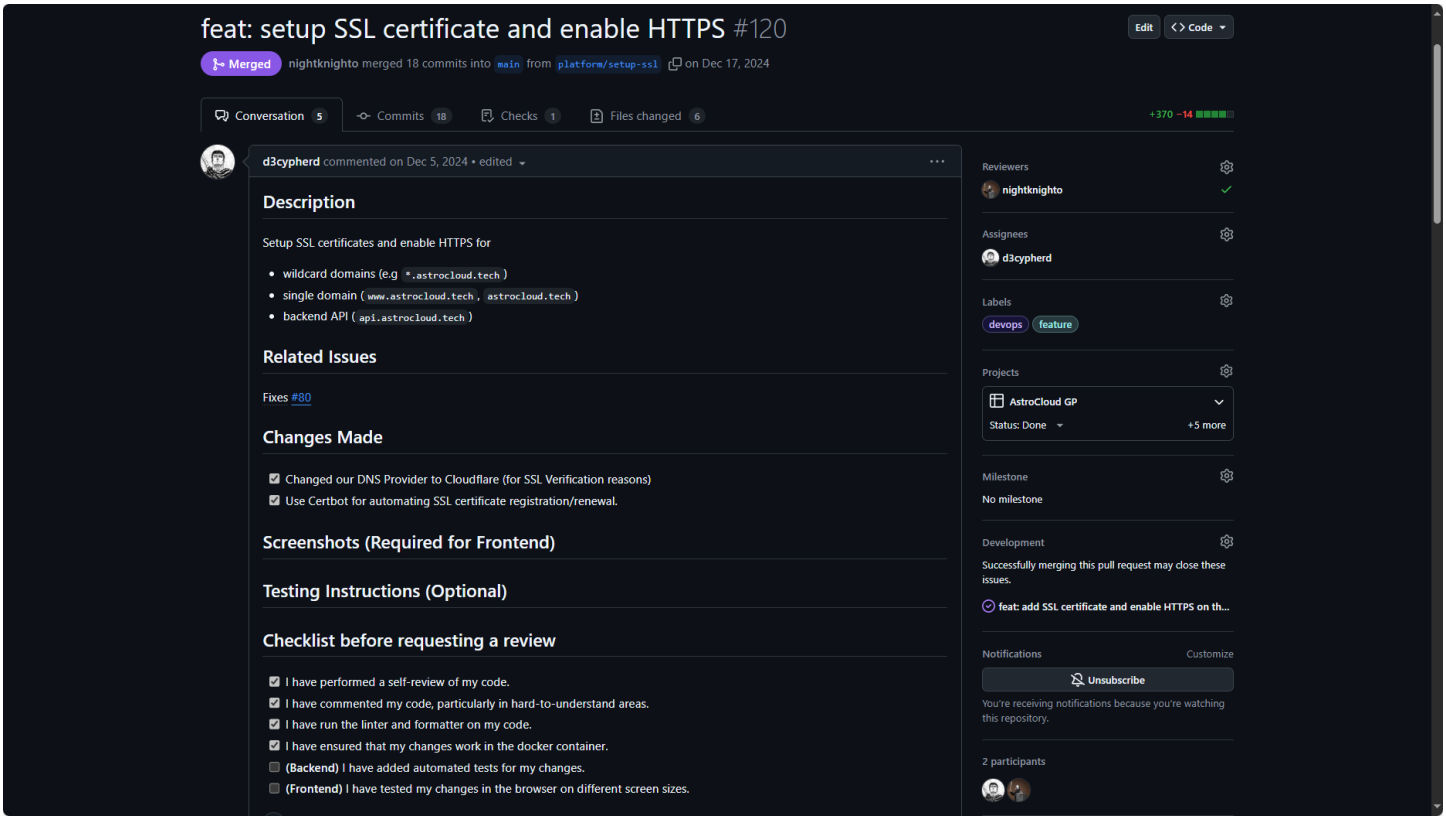
1. They **assigned themselves** to the GitHub Issue and moved it to **In Progress**, keeping the team updated on their progress.
2. Once development was completed, a **Pull Request (PR)** was created, linked to the issue, ensuring all work was reviewed before merging.
3. Each PR required **two approvals** before merging:
    - One from the **team leader** to ensure technical quality and adherence to standards.
    - One from the **project manager** to validate alignment with the project's overall goals.
4. Merging a PR **automatically closed** its associated issue and moved it to **Done**, streamlining the workflow.

Pull Request (PR) Structure

To maintain consistency and ensure clarity in code reviews, each PR followed a standardized structure:

- **Description** – A summary of the changes made, including new features or fixes.
- **Related Issues** – References to the GitHub Issues that the PR resolves.
- **Changes Made** – A detailed breakdown of modifications, listing affected files and components.
- **Testing Instructions** – Steps to verify the changes, including manual testing procedures or automated test cases.

- **Checklist Before Requesting Review** – A set of self-check items, ensuring that the code meets quality standards before review.



Communication & Meetings

Primary Communication Tools

- **Discord:**
  - **Dedicated team channels** (frontend-chat, backend-chat, devops-chat) for focused discussions, making it easy to track conversations.
  - **General channels** (e.g., general, random-memes) for broader team communication and informal interactions, helping maintain team spirit.
  - **Resource channels** (e.g., architecture-resources, similar-services, ui-ux-research) for technical references and research, providing a shared knowledge base.
  - **Meeting notes channel** to document discussions and decisions, ensuring continuity and transparency.

Meeting Structure

- **Full-Team Meetings** (Sunday & Wednesday) for tracking progress, blockers, and major decisions, keeping everyone aligned.
- **Team-Specific Meetings** are managed independently within each sub-team, allowing teams to focus on their specific goals and challenges.

Bug Tracking & Hotfixes

Efficient Issue Resolution

- For **regular bugs**, we followed the standard GitHub Issue workflow, ensuring all bugs were logged, assigned, and tracked properly.
- For **urgent bugs**, a PR could be opened directly without a corresponding issue to speed up resolution, ensuring critical fixes were deployed as soon as possible.

Handling Blockers & Dependencies

Minimizing Delays & Bottlenecks

- Large tasks were split into **independent units** that could be developed separately, minimizing interdependencies and reducing bottlenecks.
- Integration was either planned **from the start** (ensuring natural compatibility) or handled later through dedicated integration tasks, preventing delays and enabling parallel development.

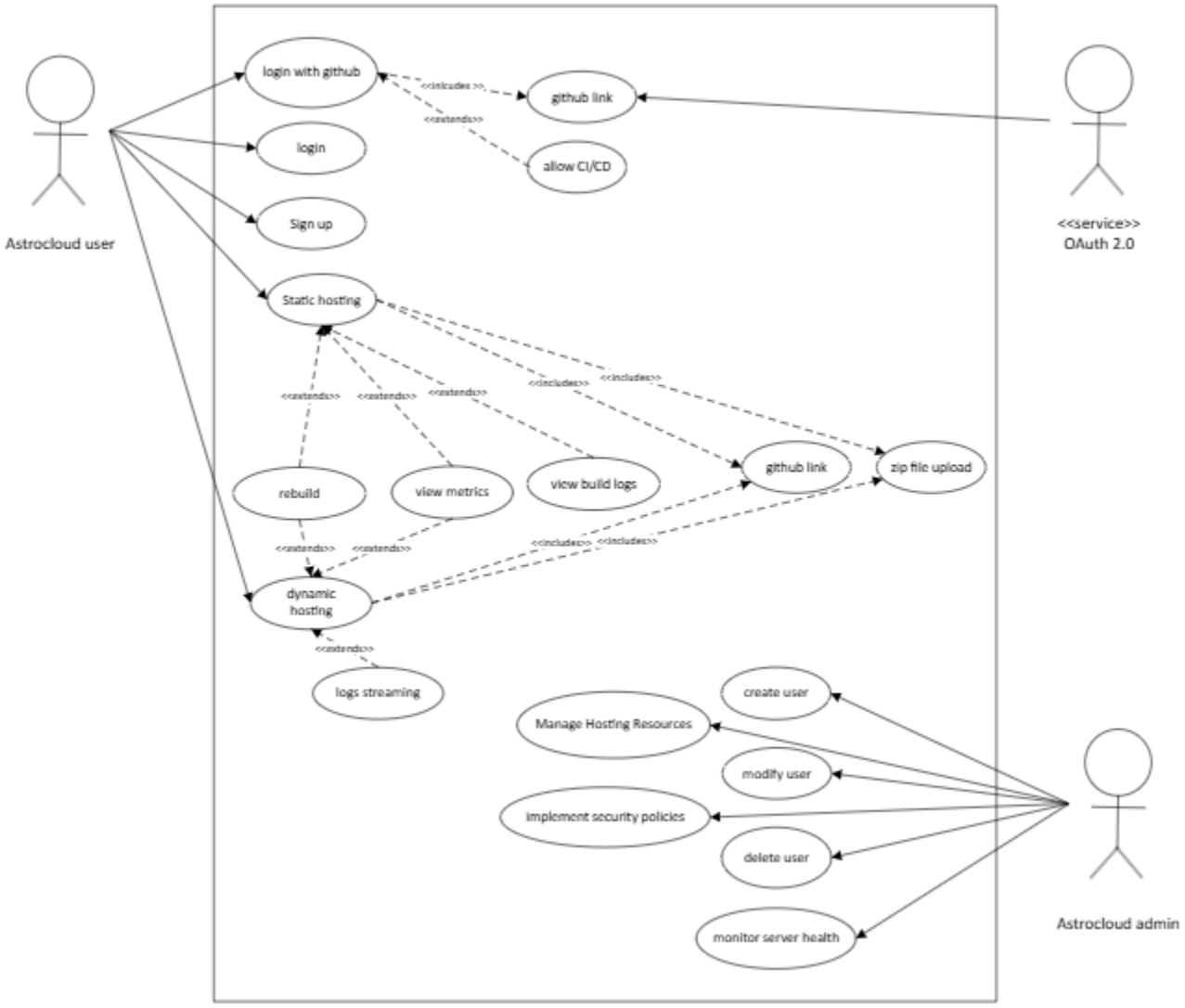
Ensuring Quality & Documentation

Maintaining High Standards

- While **formal documentation was not always up to date**, we ensured that:
    - PRs were **well-documented** before merging, reducing ambiguity and confusion.
    - Any **major process or change** was recorded in GitHub Issues or PR descriptions, ensuring decisions were well-documented.
    - Code was self-explanatory, with comments for complex logic, making it easier for new team members to onboard and understand the system.
    - Meeting notes were recorded in Discord to maintain transparency and provide reference points for discussions.

This structured approach ensured clear visibility, accountability, and smooth collaboration across all teams. By continuously refining our processes and leveraging efficient project management practices, we maintained a high level of productivity and adaptability throughout the project lifecycle.

Use Case Diagram



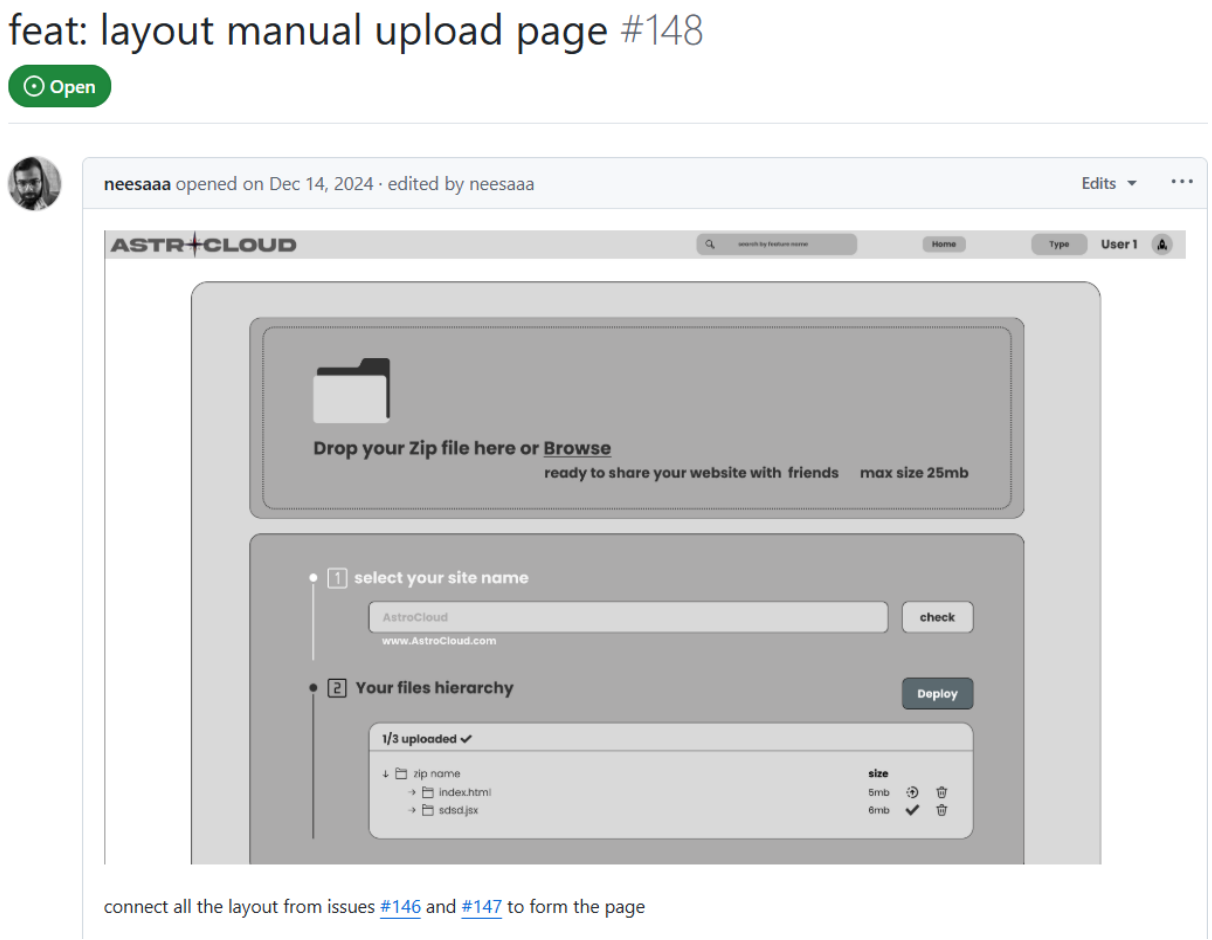
Frontend Workflow

Design

The UI/UX team designs the pages required for each functionality or feature, creating wireframes for different screen sizes using Figma. These designs are shared with the team for review and feedback, with changes made iteratively until approval is reached. Once finalized, a GitHub issue is created with the approved design as a reference for implementation. Throughout the implementation phase, the design team continues refining the designs, evolving the wireframes into final versions while incorporating feedback, resolving issues, and ensuring the designs meet project requirements.

Development

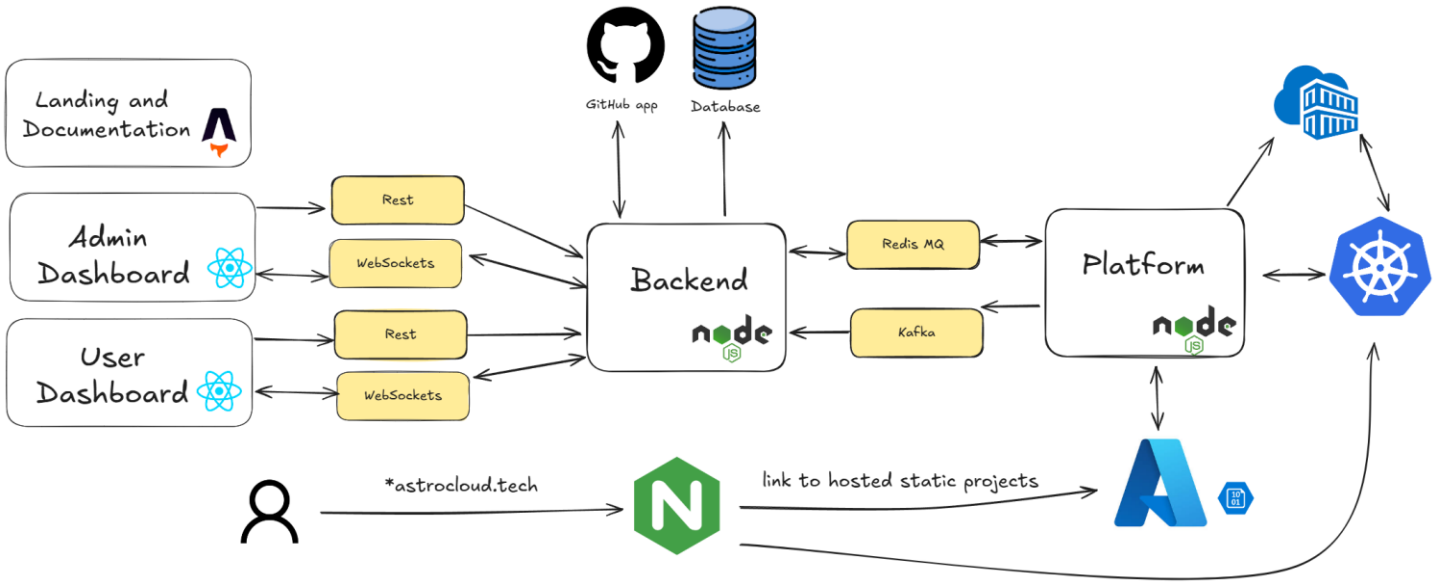
The front-end team focuses on one page at a time to ensure quick and efficient outputs. Each page is divided into two parts: UI and functionality, with the team prioritizing the UI before implementing functionality. The page is further broken down into smaller components, identifying reusable and page-specific elements. Components are assigned to team members for individual development and are created responsively to ensure compatibility with different screen sizes. Other team members integrate these components to form the complete layout of the page. The functionality is divided among team members, who collaborate with the backend and database teams to manage integration and ensure smooth operation. Each member creates a branch for their tasks, and upon completion, submits a pull request that is merged only after approval by at least two other team members.



Testing

Various inputs are tested to ensure the design handles data correctly and behaves as expected, while different user actions are simulated to evaluate the design's response under various interaction scenarios. Browser developer tools are utilized to simulate different screen sizes and resolutions, ensuring consistent behavior across devices. If a bug is identified during testing, a GitHub issue is created and assigned to a team member for resolution. The team addresses and resolves issues related to responsiveness, usability, or functionality throughout the testing process.

System Design & Architecture



System Components Overview

The system consists of three main entities: **Frontend**, **Backend**, and **Platform**.

Frontend

The frontend is divided into three distinct web applications:

- User Dashboard: The primary interface that users interact with. It allows users to manage their projects, monitor deployments, and configure settings.

- Admin Dashboard: Provides administrative functionality, including the ability to view all users, projects, and plans, along with access to advanced controls and system statistics.

- Landing Page: The initial page presented to first-time visitors. It introduces the platform, showcases its capabilities, highlights available features, and contains user documentation and examples.

Backend

The backend is responsible for processing user actions and requests. It acts as the intermediary between the frontend and the platform, handling business logic, API endpoints, and coordination between services.

Platform

The platform handles all hosting and deployment operations, including building, deploying, and managing both static and dynamic projects.

Interactions Between Components:

- **Frontend ↔ Backend:** Communicate via **RESTful APIs** and **WebSocket** connections.
- **Backend ↔ Platform:** Use **BullMQ (message queue)** and **Kafka (publish/subscribe)** for asynchronous, decoupled communication. This architecture enables independent development, testing, deployment, and scaling of each component.

Supporting Infrastructure

In addition to the core three entities, the system leverages several other services:

- PostgreSQL database: The main relational database used to store application data.

- NGINX web server: Hosts the main Kubernetes cluster used for deploying and managing dynamic applications.

- Azure Blob Storage: Stores files for all statically hosted websites.

- Azure Container Registry (ACR): Stores container images for dynamic projects.

- Azure Kubernetes Service (AKS): Hosts the main Kubernetes cluster used for deploying and managing dynamic applications.

- Fluent bit: A lightweight observability tool used to collect logs from dynamic projects.

- Prometheus: A monitoring and alerting toolkit used to monitor resource usage and performance metrics for dynamic projects.

In the following sections, we explore the architecture and code design of each of these core components in greater detail.

User Dashboard Architecture

Our project's frontend is organized to maintain separation of concerns, reusability, and scalability. Below is a breakdown of the folder structure and its contents.

Folder Structure Overview

The project's frontend structure is organized into several main directories. Each directory serves a specific purpose, making it easier to manage the different components, hooks, states, and page-specific logic.

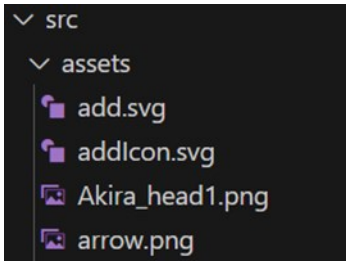
```
src/
├── assets/
│   └── media/
├── components/
│   ├── sharedComponents/
│   │   └── sharedComponent.tsx
│   └── pageComponents/
│       └── pageComponent.tsx
├── hooks/
│   └── customHooks.tsx
├── pages/
│   ├── page1.tsx
│   ├── page2.tsx
│   └── rest of pages
├── store/
│   ├── store1/
│   │   ├── context.tsx
│   │   └── provider.tsx
│   └── store2/
│       ├── context.tsx
│       └── provider.tsx
├── utilities/
│   ├── actions/
│   ├── mutations/
│   ├── Query/
│   ├── ApiError.ts
│   ├── ApiErrorFunctions.ts
│   ├── auth.ts
│   ├── formatters.ts
│   └── githubUrl.ts
├── App.tsx
└── main.tsx
```

1. Src directory

The root directory for all source code files.

2. Assets directory

This folder contains all media assets used across the project, such as images, icons, fonts, etc. These assets are imported and utilized in different components or pages.

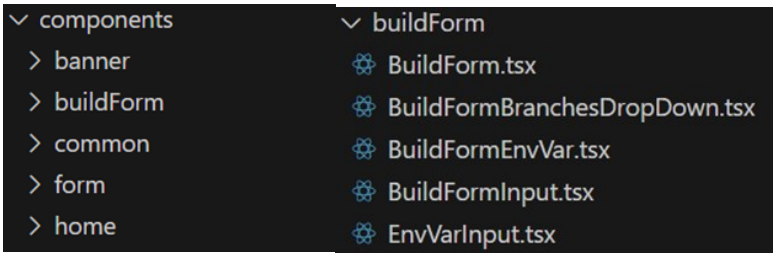


- **Purpose**: To store all static media resources used in the project.

3. Components directory

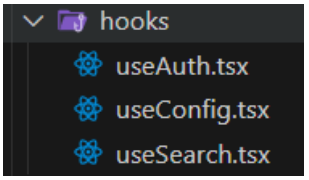
The components/ directory holds individual React components. It is divided into two parts:

- **Page-Specific Components**: Each folder inside represents a separate page, containing components specific to that page.
- **Shared Components**: This section contains reusable components that are shared across multiple pages (e.g., buttons, search Bars, etc).



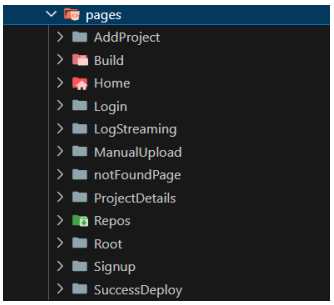
4. Custom-hooks directory

This directory contains custom React hooks, which are utilities that encapsulate and manage specific pieces of logic or state. Most of these hooks internally use useContext to manage different states and make them available across the app.



- **Purpose**: To encapsulate reusable logic for managing states across various components.

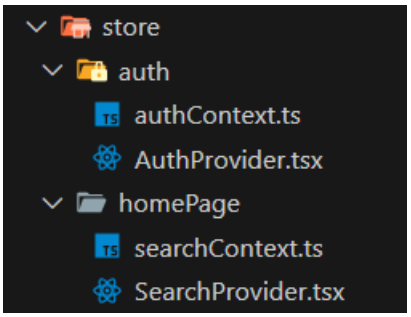
5. Pages directory



The pages/ directory contains folders for each route or page in the application. Each folder contains the tsx for that specific page.

- **Purpose**: To organize the app's pages, each containing their unique components.

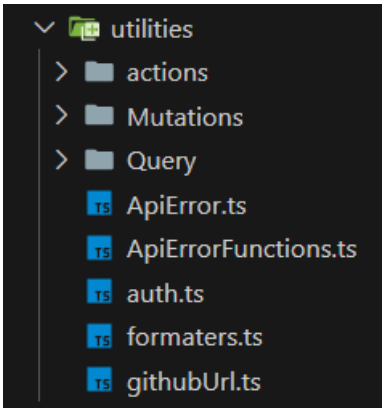
6. Store directory



This folder contains the context management logic for the application. Inside this folder, there are multiple directories that manage specific states, each containing:

- **Context File**: Manages the state and business logic.
- **Provider File**: Provides the context to other parts of the application.
- **Purpose**: Manages the app's global state, such as authentication status or user data, and provides them via context.

7. Utilities directory



The utilities/ folder houses various helper functions and files that provide utility services. It contains:

- **Actions Folder**: Holds logic related to form submissions, such as handling validation and processing data. Each form typically has its own action file.
- **Mutations Folder**: Holds logic related to build form submissions, such as handling validation and processing data and handle api errors of response .
- **Query Folder**: Orchestrates data retrieval from the backend for the application's pages using React Query's useQuery hook, enabling efficient fetching of server-side data such as user projects, user repos, or project details. These utilities implement query logic with support for caching, error handling, and automatic refetching to ensure optimal performance and seamless integration with page components.
- **ApiError.ts**: Defines a custom Api Error class , designed to handle API-related errors with a standardized structure. The class includes a statusCode property to capture HTTP status codes, enabling precise error handling and debugging across the application.
- **ApiErrorFunctions.ts**: Provides utility functions for processing API errors, enabling standardized error handling and user-friendly feedback. The module includes two key functions: HandleApiErrors for throwing ApiError instances and HandleObjectErrors for returning structured error objects.
- **auth.ts**: simplifies access to authentication data, enabling secure integration with backend APIs and a seamless user experience in authentication-related workflows.
- **formaters.ts**: ensures consistent and user-friendly presentation of project-related data, such as creation dates and runtime versions, across the application's pages.
- **githubUrl.ts**: enables robust integration with GitHub repositories by providing utilities for URL construction, parsing, and validation. It supports project pages that manage repository configurations, ensuring accurate URL handling
- **Purpose**: Contains all helper functions for form submission, data loading, error handling and other utility tasks.

Relationship Between Pages, Actions, Mutations and Queries

Concept

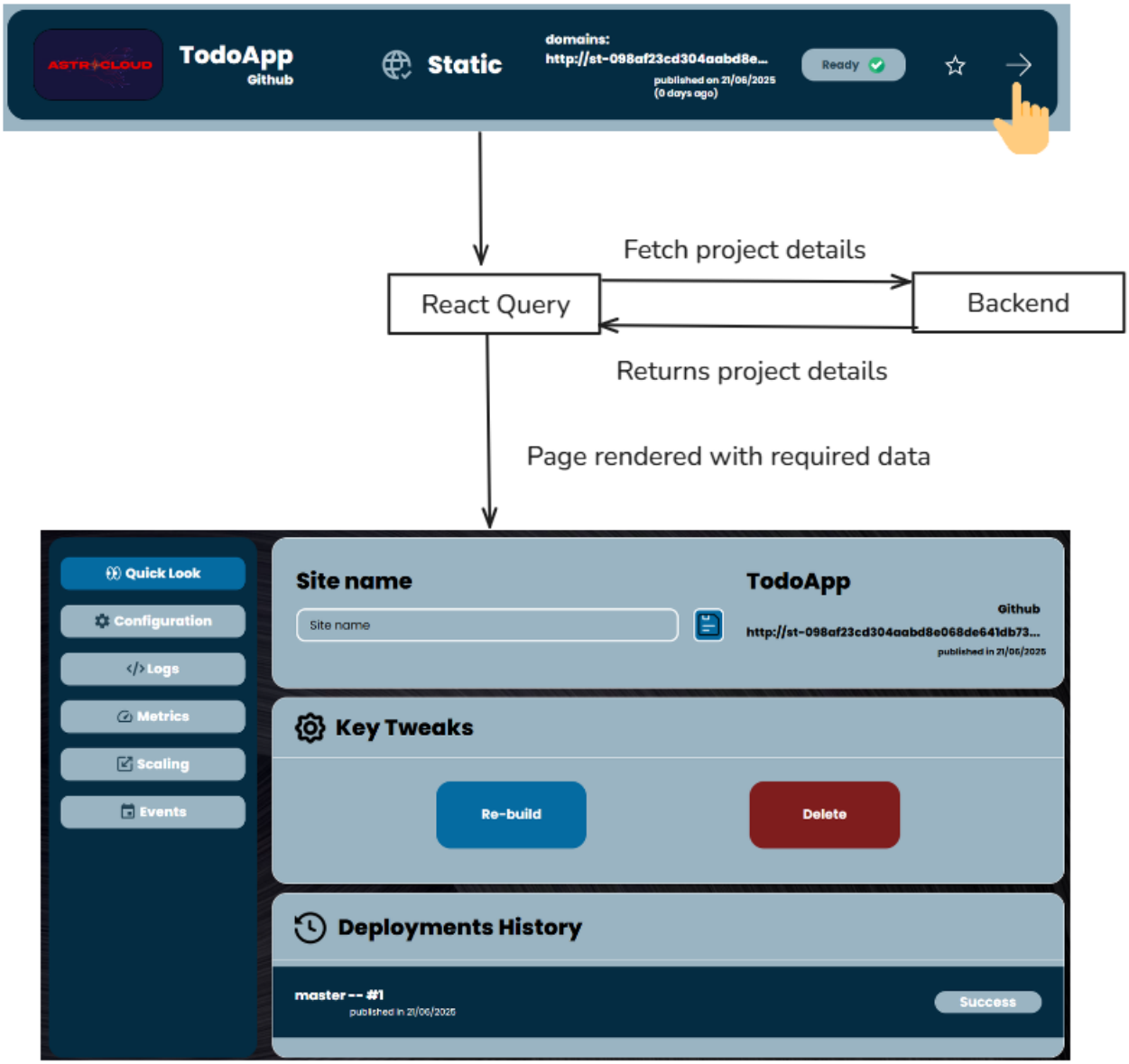
In a web application, each page typically represents a specific route. Pages often interact with **actions** and **mutations** and **queries** to handle user interactions and fetch data, respectively. Here's how these elements relate:

1. **Pages**: Represent the visual structure and user interface of a specific route.
2. **Actions** and **Mutations**: Handle user-triggered events (e.g., form submissions, button clicks) and perform corresponding operations like creating, updating, or deleting data.
3. **Queries**: Fetch the necessary data to populate a page before it is rendered or refreshed.

This relationship ensures the page has the required data and functionality to provide a seamless user experience.

Example

1. Opening project details page:



The **Project Details Page** illustrates how a query, a React component, and user-triggered actions work together to provide a seamless user experience. This page showcases detailed information about a specific project, enables users to perform actions like deleting or rebuilding the project, and ensures the data is up-to-date by fetching it dynamically.

Page Functionality

Purpose

The Project Details Page is designed to:

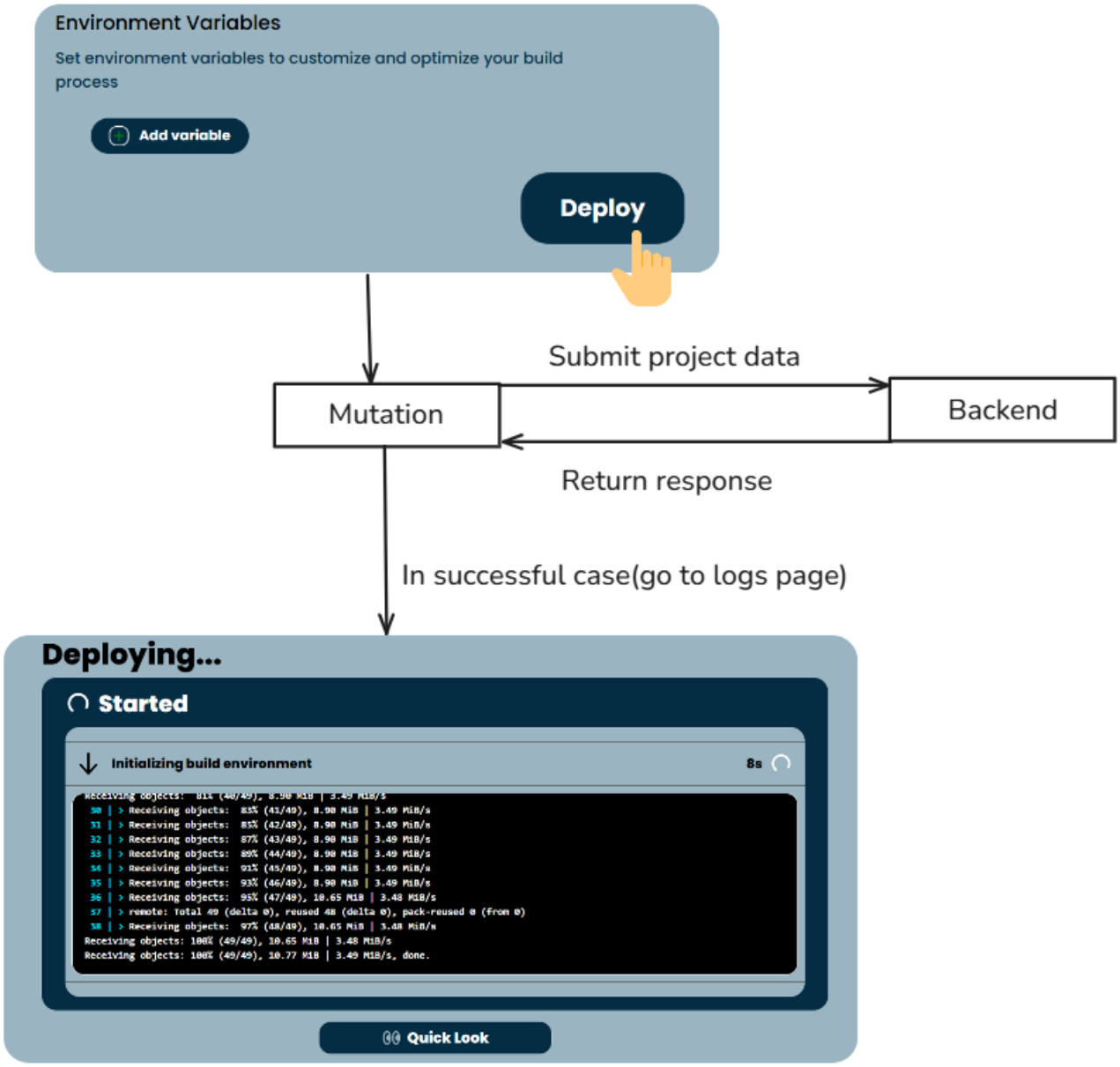
1. Display essential details about a specific project, such as its name, deployment URL, creation date, and build history.
2. Allow users to perform critical actions:
    a. **Delete Project**: Removes the project permanently.
    b. **Rebuild Project**: Triggers a rebuild of the project's static files.

Associated query: Fetching Project Data

- **Role**: The query efficiently fetches the required project details, ensuring the page renders with up-to-date data while providing better loading states and error handling.
- **Process**:
    - Authenticate the user by verifying the presence of a valid token.
    - Fetches project data from the backend using the project ID from the route parameters.
    - Handles errors gracefully, such as redirecting unauthorized users to the login page and returning an error message for failed requests.

2. Submitting Build form:

The interaction between the **Build Form Page** and the **Build Form mutation** is crucial for creating and deploying a static and dynamic projects. It handles user inputs, validates the data, processes the submission, and communicates with the backend API.



Workflow Steps:

1. **Form Rendering and User Input:**
    a. The **Build Form Page** presents a user interface where users input project details, such as site name,repo url, branch to deploy, build command,etc.
2. **Submission Handling:**
    a. Upon clicking the "Deploy" button, the form's **onSubmit** event is triggered:
        i. If validation passes, the form data is prepared for submission.
        ii. The button is temporarily disabled, and a loading indicator is shown to prevent duplicate submissions.
3. **Data Packaging:**
    a. The form data, including the environment variables (if provided), is sent to the **mutation Function** using the POST method.
4. **Mutation Function Processing:**
    a. The **Mutation Function** ensures user authentication by checking for a valid token. If the token is missing or invalid, the user is redirected to the login page.

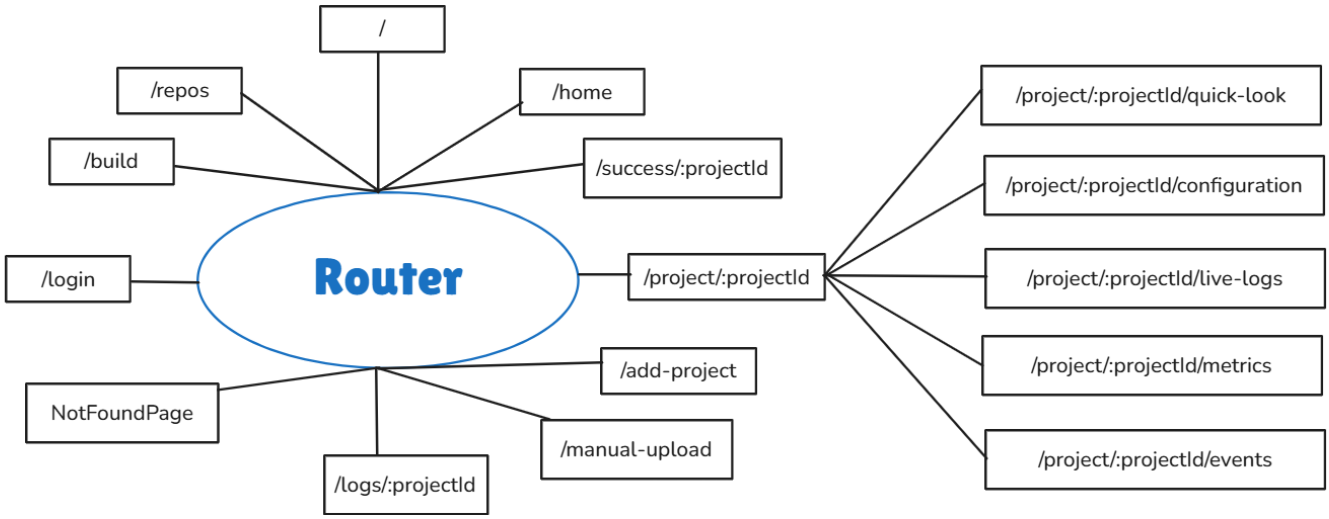
b. The submitted form data is parsed and converted into a structured payload containing Project name, Repository URL, Branch name, Build command, Root directory, Output directory, Environment variables.
c. Data validation and error handling occur to ensure the payload meets backend API requirements.

5. **Backend API Interaction:**
   a. The action sends the structured payload to the backend API endpoint for project creation and deployment.
   b. The backend processes the request, stores the project details, and begins the build process.
6. **Response Handling:**
   a. If the API response indicates success:
      i. Redirection to log streaming page occurs that show the logs that is being done to build the project and deploy it.
   b. If an error occurs:
      i. The error message is displayed on the form, we will navigate or stay in build form page based on status code that returns from backend.

Routing with React Router DOM

In our project, **React Router DOM** is used to manage navigation between the different pages and sections of the application. This library enables us to implement **single-page application (SPA)** behavior, where only the necessary content is loaded when navigating between different views, rather than reloading the entire page.

Architecture



Benefits of Using React Router DOM

- **Single-page Application**: React Router DOM helps us create a single-page application, making the navigation feel instantaneous as only relevant content is loaded.
- **Component-based Navigation**: We can encapsulate different page views as components, making the routing system clean and modular.
- **Seamless Navigation**: Links to different pages are handled without page refreshes, providing a smooth, app-like experience.
- **Dynamic URL Management**: React Router provides flexible ways to manage dynamic URLs and query parameters, which can be used to display customized content based on the route.

Design with Tailwind CSS

Tailwind CSS is used for styling our application, embracing its utility-first approach to create highly customizable and maintainable designs. With Tailwind, we apply individual utility classes directly in the HTML/JSX code to control the layout, typography, colors, and responsiveness of our pages.

Benefits of Using Tailwind CSS

- **Consistency**: By using utility classes, we ensure a consistent design system across the entire app, adhering to predefined spacing, color, and typography rules.
- **Flexibility and Customization**: Tailwind allows us to easily tweak the design to fit specific project requirements, whether through extending the configuration or applying custom classes.
- **Performance**: Tailwind CSS encourages the use of only the necessary styles, making the final CSS bundle smaller and more efficient. Unused styles are purged during production, optimizing the app's performance.
- **Responsive Design**: Tailwind's built-in responsive utilities allow us to build mobile-first applications with minimal effort, ensuring that the app looks great on all screen sizes.
- **Faster Development**: The utility-first approach allows developers to style elements quickly without having to write custom CSS for every component, speeding up the development process.

Admin Dashboard Architecture

Overview

The Admin Dashboard serves as the centralized control panel for managing the core operations of the platform—including users, projects, and server infrastructure. Designed with scalability, modularity, and user experience in mind, the admin interface streamlines administrative tasks through an intuitive layout and powerful analytics. This documentation outlines the structure and functionality of each major section, helping administrators monitor platform health, oversee deployments, manage user accounts, and maintain resource efficiency with ease.

Tech Stack

Category	Technology
Framework	Next.js
Language	Typescript
Styling	Tailwind CSS

Project Structure

```
admin/                          # Admin dashboard application root
├── app/                        # Contains folders for each page in the admin UI
│   ├── dashboard/
│   ├── login/
│   ├── plans/
│   ├── projects/
│   ├── server/
│   └── users/
├── assets/                     # Static assets like logos, icons, and images
├── components/                 # Shared and reusable UI components
├── hooks/                      # Custom React hooks for logic reuse
├── lib/                        # Utility functions, API clients, and services
├── styles/                     # CSS files for styling the admin interface
│   └── global.css              # Global stylesheet for consistent base styling
```

Admin Login:

The Admin Login page is the entry point for administrators to access the dashboard.

- **Authentication Methods:**
  - **Sign in with GitHub:** Securely authenticate using your GitHub account. This is a quick and convenient method for integrated development workflows.

Admin Dashboard:

The Admin Dashboard provides a comprehensive overview of the platform's key metrics, user activity, and resource utilization.

- **Navigation:** Accessible via the "Dashboard" link in the left-hand sidebar.
- **Key Metrics:**
    - **New Users Today:** Displays the number of new user registrations within the current day.
    - **Total Users:** Shows the cumulative count of all registered users on the platform.
    - **Total Deployments:** Indicates the total number of projects deployed across the platform.
    - *Daily/Weekly/Monthly Filters:* Toggle between daily, weekly, and monthly views for these metrics to observe trends over different timeframes.
- **System Usage:**
    - This section presents a graphical representation of your platform's resource utilization over time.
    - **CPU:** Monitors the central processing unit load.
    - **Memory:** Tracks memory consumption.
    - The graph allows you to visualize resource trends and identify potential bottlenecks or periods of high activity.
- **Recent Users:**
    - Provides a quick summary of the most active users in the system.
    - **User:** Displays the username and the date they joined.
    - **Status:** Shows their current activity status (e.g., Active, Last active: [Date]).
    - **Projects:** Indicates the number of projects associated with that user.

Server Management:

The Server Management section offers detailed insights into the server's performance and health, allowing us to monitor critical resources.

- **Navigation:** Accessible via the "Server" link in the left-hand sidebar.
- **Resource Monitoring:**
    - **CPU Load:** Shows the current CPU utilization percentage, compared to yesterday's performance.
    - **Memory Usage:** Displays the current memory consumption percentage, compared to yesterday's performance.
    - **Disk Usage:** Indicates the current disk space utilization percentage, compared to yesterday's performance.
    - **Network Usage:** Shows the current network data transfer rate (e.g., 500 MB/s), in comparison to yesterday's performance.
- **Server Status:**
    - A line graph visually represents the historical performance of your server's CPU and Memory usage over several months. This helps in identifying long-term trends and ensuring server stability.

Projects:

The Projects section allows administrators to manage all projects deployed in the system.

- **Navigation:** Accessible via the "Projects" link in the left-hand sidebar.
- **Project Overview:**
    - Displays a list of all projects, along with key details.
    - **Last published:** Shows the date and time of the last deployment for each project.
    - **Search by project name:** A search bar to quickly find specific projects.
    - **Sorting:** Projects can be sorted by various criteria (e.g., "Last published").
- **Project Details:**

- **Project Name:** The name of the deployed project.
- **Owner:** The user associated with the project.
- **View:** A link to view the live deployed project.
- **Status:** Indicates the current deployment status (e.g., Active).
- **Type:** Categorizes the project as "DYNAMIC" or "STATIC".
- **Actions:** Icons for quick actions such as deleting or managing project settings.

Users:

The Users section enables administrators to manage all registered users in the system.

- **Navigation:** Accessible via the "Users" link in the left-hand sidebar.
- **User Management:**
  - **Search users:** A search bar to quickly find specific users.
  - **User List:** Displays a table of all users with relevant information.
  - **User:** The username and their join date.
  - **Status:** The user's current activity status (e.g., Active, Last active: [Date]).
  - **Projects:** The number of projects the user has deployed.
  - **Actions:** Ellipsis icon (...) to reveal additional actions for user management (e.g., edit user details, suspend account).

Navigating the Admin Site:

- **Sidebar Navigation:** The left-hand sidebar provides quick access to the main sections of the admin site: Dashboard, Server, Projects, and Users.
- **Log Out:** The "Log Out" option at the bottom of the sidebar allows you to securely exit your admin session.

plans will be discussed further in

Landing and Documentation Website Architecture

Overview

The architecture of the landing website is built around modern web development best practices, focusing on **performance**, **SEO**, **content flexibility**, and **developer experience**. It uses the **Astro framework** to generate fully static pages with rich metadata support, providing fast load times and excellent indexing across search engines and social platforms.

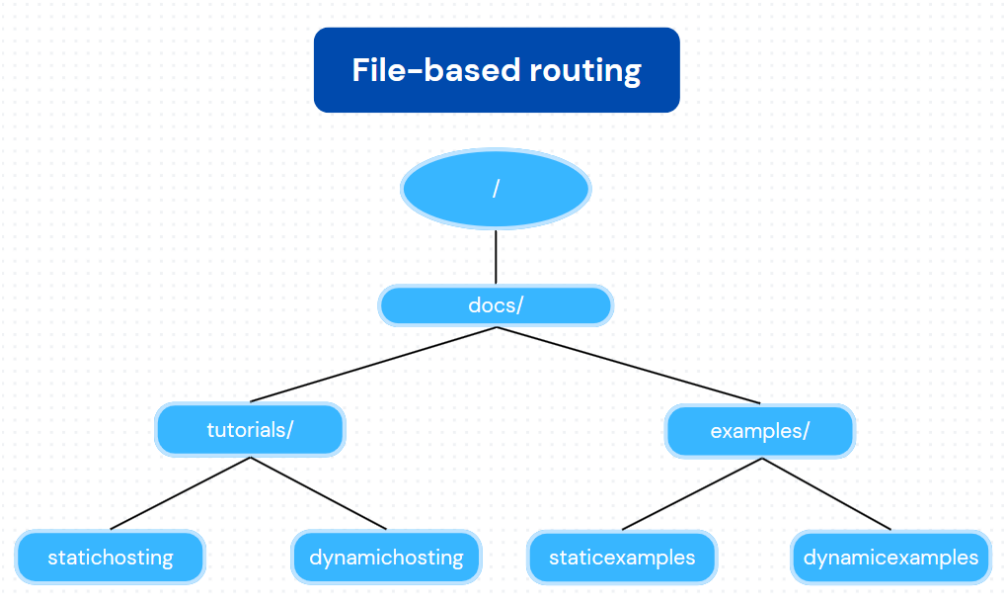
Key Architectural Principles

- **Separation of Concerns** between content (Markdown/MDX), layout (Astro components), and styling (SCSS).
- **Static Site Generation (SSG)** using Astro for fast performance and full SEO compatibility.
- **Dynamic Routing** for documentation content using Astro's [...id].astro catch-all route.
- **Content-Driven Structure** where Markdown files are transformed into fully interactive web pages.
- **Component Reusability** through modular .astro components.

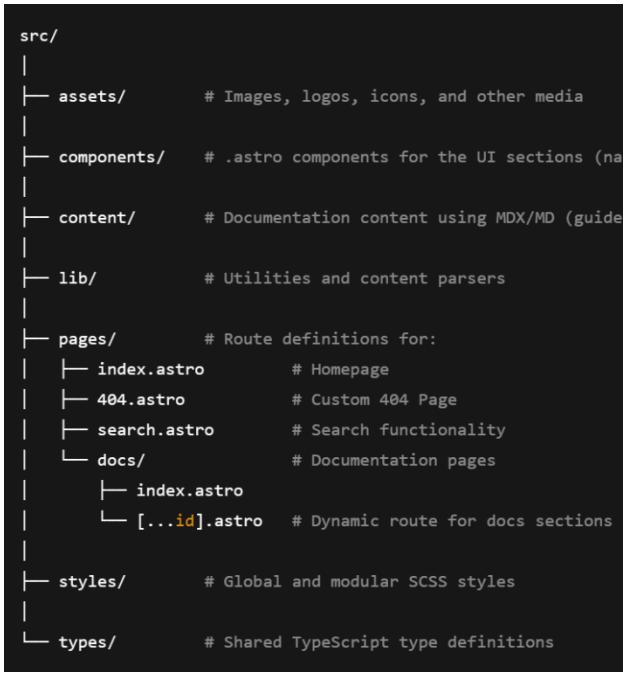
Tech Stack

Category	Technology
Framework	Astro
Styling	TailwindCSS, SCSS
Language	TypeScript
Content	Markdown & MDX
UI Theme	Astrogon (customized)
Animations	Integrated through SCSS
SEO Support	Metadata per page (Open Graph, Twitter, etc.)

Architecture



Project Structure



src:

The main source directory containing all the website's code, assets, and logic.

assets:

Holds static media files such as images, logos, icons, and screenshots used across the landing site.

components:

Contains reusable .astro UI components like the navbar, footer, feature cards, and layout blocks that are used throughout pages.

content:

Includes all written documentation and tutorial content in .md and .mdx format. This is where guides for static/dynamic hosting and real project examples are stored.

lib:

Stores utility functions and internal logic — for example, content parsers that process Markdown files or helper functions used in the components.

pages:

Defines the website's routes and page structure. Astro automatically turns these files into site pages. It includes:

- index.astro for the main landing page.
- 404.astro for custom error handling.
- search.astro for searching through content.
- docs/ folder for handling documentation pages using static or dynamic routing.

styles:

Contains global and modular SCSS styles that control the design and theming of the site, complementing Tailwind CSS for more detailed customization.

types:

Defines TypeScript types and interfaces shared across the project to maintain consistent typing and reduce errors, especially when working with content and metadata.

Customizations Made to the Astrogon Theme

We used the open-source Astrogon theme as a base and applied the following customizations:

- Modified color palette and animations to align with our branding.
- Replaced default content with custom tutorials and project examples.
- Added extra routes (/docs, /search).
- Extended component design using SCSS and Tailwind for responsiveness.
- Integrated Open Graph metadata for link previews.

Reference : https://astro.build/themes/details/astrogon/

Backend Architecture

Overview

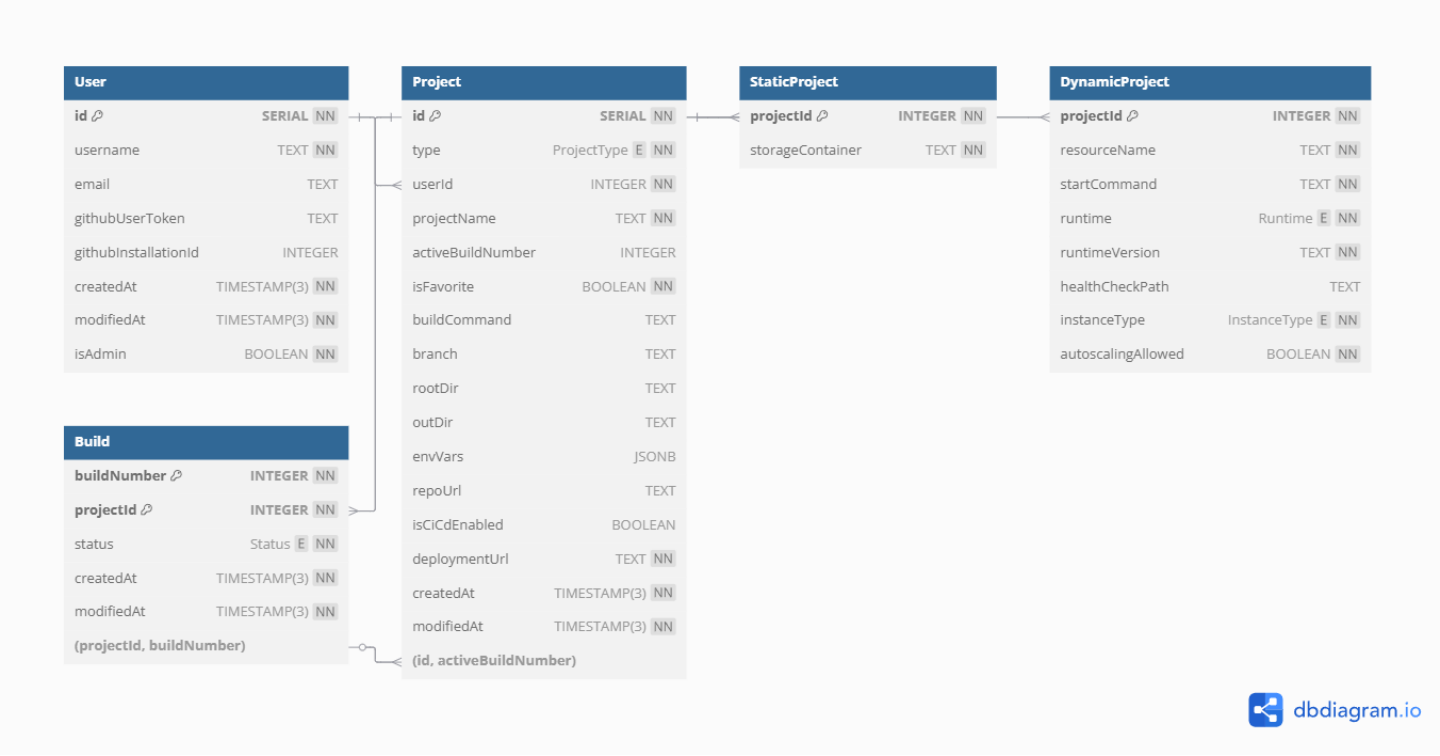
The backend is considered the heart of the project acting as the link between the platform and the frontend as shown in the figure below.

The frontend interacts with the backend mainly through REST APIs. Additionally, they are connected via WebSocket, which serves a specific function: sending logs to the frontend.

The platform communicates with the backend via a Redis queue, enabling two-way communication for tasks that require quick, real-time messaging (such as handling user requests or processing tasks).

Kafka, on the other hand, is used to establish one-way communication from the platform to the backend. Kafka acts as an event streaming platform for handling high-volume, fault-tolerant, and distributed messaging.

Database Schema



For this project we used PostgreSQL and a relational database design since it closely matches the relationships needed to structure our system. It is composed of 5 main entities: (User, Project,StaticProject,DynamicProject Build)

1- User: Contains the main user data required to identify him and access his GitHub account info.
2- Project: Represents a software project created by a user (can be static or dynamic).
3- StaticProject: Represents additional info for static projects (like simple frontend apps) and has o*ne-to-one relationship with* `Project`.
4- DynamicProject: Represents additional config for dynamic projects (e.g., backend apps) and has o*ne-to-one relationship with* `Project`.

5- Build: A Project constitutes multiple builds. A build is a version of the project that was deployed at a certain time. It contains data about the status of the process and the build logs. *Each project can have multiple builds (one-to-many)*.

RESTful API Design

The RESTful API serves as the heart of the project. It connects the frontend to the database and the platform while also providing many features and functionalities.

It contains the API endpoints, GitHub API integration using OctoKit for authentication and CI/CD, validation service and authorization service.

It also adheres to RESTful web API design standards and best practices for designing the API endpoints based on the available resources, using the correct request verbs and the correct response codes for different request types, it also has API versioning.

The current API endpoints can be reached at /api/v1/ and consists of 5 main routes:

/users

POST /: Create a new user after successful GitHub sign-up.

* GET /:id: Retrieve details of a specific user by his id.

/static/projects

* GET /: List all static projects for the logged-in user.

* POST /: Create a static project from the GitHub repository URL.

* GET /:id: Retrieve details of a specific static project owned by the user.

* PATCH /:id Update details of a specific project owned by the user.

* DELETE /:id: Delete a project owned by the user.

* GET /:id/builds: List all builds related to a specific project owned by the user.

* POST /:id/rebuild: Trigger a rebuild for a specific project owned by the user.

/auth

GET /github?code=CODE: Authenticate with GitHub using a code provided by the frontend.

/github

POST /webhooks: Handle GitHub webhook events from the GitHub App integration.

* GET /repos: List the authenticated user's repositories with GitHub App installed.

* GET /repos/:owner/:repo/branches: Retrieve branches for a specific repository.

/upload

POST /: Create a static project by uploading a ZIP file directly.

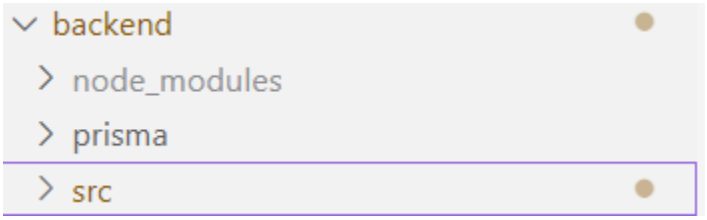
Note:

- Endpoints preceded by * can be accessed only from a request authorized by a bearer token.

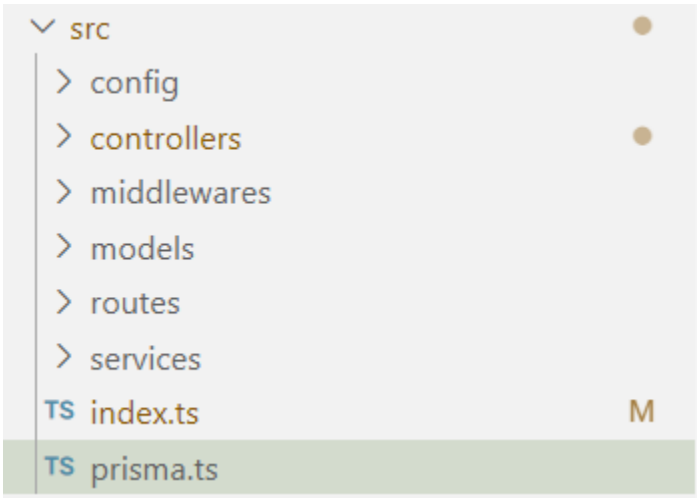
- Validation is provided for all required request query parameters and request body data.

Our model-based backend architecture

Since a model-based architecture is adopted for project development, the following figure outlines the hierarchy of the project files.



Getting inside the “src” directory, the code is divided as in the following figure



1. **index.ts file**

   sets up an **Express.js server** with **CORS** support, JSON request parsing, and error handling. It also imports and sets up a **result worker** to handle completed jobs.

   - *Express app setup*

```
const app = express();
app.use(cors());
app.use(express.json());

app.use(routes);
```

   - **`cors()`** middleware is used to handle CORS, allowing the server to accept requests from different origins.
   - **`express.json()`** middleware is used to parse incoming JSON requests into JavaScript objects.
   - **`routes`** is used to define the application's API endpoints by attaching the imported routes.

   - *Error handling middleware*

```
app.use((err: Error, _req: express.Request, res: express.Response, _next: express.NextFunction) => {
    console.error('Error:', err);
    res.status(500).json({ message: 'Internal Server Error' });
});
```

If any error is thrown during request processing, it logs the error and returns a 500 status with an "Internal Server Error" message.

- *Server Listening*

```
const PORT = CONFIG.port;
app.listen(PORT, () => {
    console.log(`Server is running on port ${PORT}`);
});
```

The server is set to listen on the port defined in `CONFIG.port`. A message is logged to the console once the server starts.

- *Worker Setup*

```
// resultWorker must be imported here to start the worker
resultWorker.on('completed', async (job) => {
    console.log(job.name, 'is done');
});
```

The **`resultWorker`** listens for the `completed` event, which indicates a job has finished. When a job is completed, it logs the job's name with a message (`is done`).

2. **Config directory**

It contains a configuration code for GitHub authentication environment variables.
The `config.ts` file defines and validates the application's configuration settings by ensuring all required environment variables are present. It contains the following:

- UNSAFE_CONFIG object
  It is a central place to define and manage configuration settings for the application. These settings are typically loaded from environment variables (`process.env`) to make the application flexible and adaptable to different environments (e.g., development, staging, production). Its mechanism based on the following:

  a. Retrieves the PORT environment variable (set by hosting platform). If PORT is not defined, it defaults to `3000`.
  b. Retrieves the `GITHUB_AUTH_CLIENT_ID` environment variable (Client ID for GitHub OAuth integration).

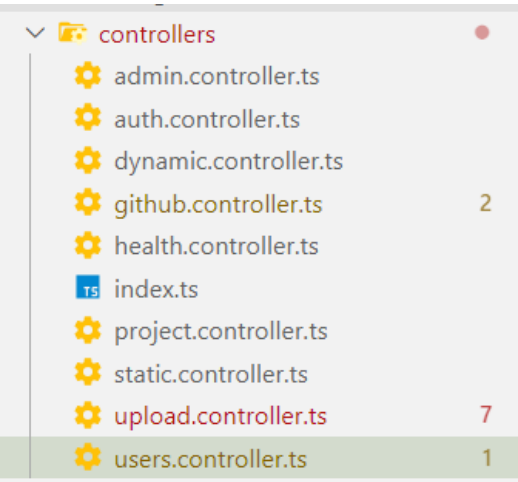
c. Retrieves the `GITHUB_AUTH_CLIENT_SECRET` environment variable (secret key for GitHub OAuth Client ID. It's used to securely authenticate requests to GitHub)
d. Retrieves the `JWT_SECRET` environment variable (it is used to sign and verify JSON Web Tokens (JWTs) in the application. It ensures that only the application can create and validate these tokens, enhancing security).
e. Retrieves the `DEPLOYMENT_URL_TEMPLATE` environment variable.

- `function` **`validateConfig`**`(config: object, path = ''): void`

  This function recursively checks the `UNSAFE_CONFIG` object to ensure no configuration value is missing or left blank. If a value is `undefined` or an empty string, an error is thrown with a descriptive message indicating which configuration key is invalid (using the `path` parameter for nested keys).
  The figure below shows the implementation of the function:

```
function validateConfig(config: object, path = ''): void {
    for (const [key, value] of Object.entries(config)) {
        const currentPath = path ? `${path}.${key}` : key;

        if (value === undefined || (typeof value === 'string' && value.trim() === '')) {
            throw new Error(`Configuration Error: Missing configuration value for: ${currentPath}.
            Please set the environment variable corresponding to this configuration key.`);
        }

        if (typeof value === 'object') {
            validateConfig(value, currentPath);
        }
    }
}
```

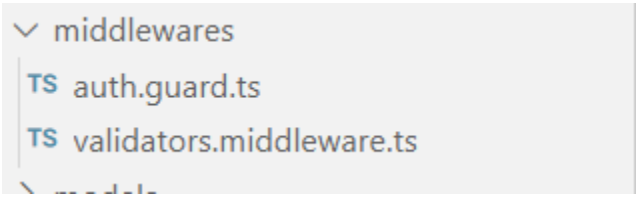
3. **Controllers directory**
   It contains files that define the logic for handling requests and responses for various routes or API endpoints.



- Admin.controller --> handles admin operations for users and projects, including dashboard stats, user/project listing, details, and updates.
- Auth.controller --> handles GitHub OAuth login by exchanging a code for an access token, retrieving user info, creating or updating the user in the database, and returning a JWT token with user details.
- Dynamic.controller --> manages dynamic project creation and updates by handling project data, initiating build jobs, interacting with Kafka for topic setup.
- Github.controller --> handles GitHub-related operations such as verifying webhooks, fetching repository visibility, listing accessible repositories, retrieving repository branches, and processing CI/CD jobs triggered by GitHub events.

- Health.controller --> checks the health of Kafka, Postgres, and Redis services in parallel and returns their statuses, responding with 200 if all are healthy or 503 if any are unhealthy.
- Project.controller --> manages project operations including retrieval, deletion, build listing, rebuilding, rollback, and marking projects as favorites, while handling both static and dynamic project types.
- Static.controller --> handles creation and updating of static projects, including initiating build jobs and Kafka topic setup.
- Upload.controller --> manages manual project uploads by generating secure upload URLs, validating project names and uploaded files, creating static or dynamic projects, and queuing corresponding processing jobs.
- Users.controller --> handles user creation and retrieval by registering new users and fetching user details based on their ID.

4. **Middlewares directory**



It contains middleware functions that are used to process requests and responses in the application's request-response cycle.

At auth.guard file, it contains the following middlewares:

- IsAuthenticated()
  - This middleware verifies the user's identity based on a JWT token.
  - After validation, it adds the `userId` to the `Request` object for subsequent use.

- IsAuthorizedForProject()
  - his middleware checks whether the authenticated user has permission to access a specific project.
  - Relies on the `userId` set by the `isAuthenticated` middleware.

- IsAuthorizedForContainerName()
  - Validates that the storage container name in the request body belongs to the authenticated user.

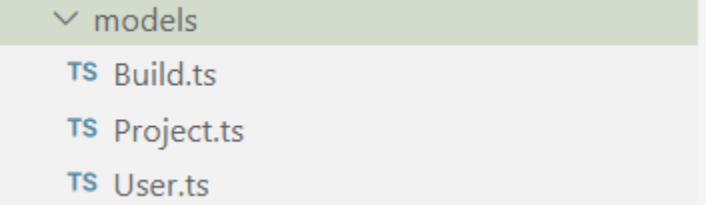
- IsAuthorizedAsAdmin()
  - Ensures the user has admin privileges.

At validators.middleware.ts, it contains the following middlewares:

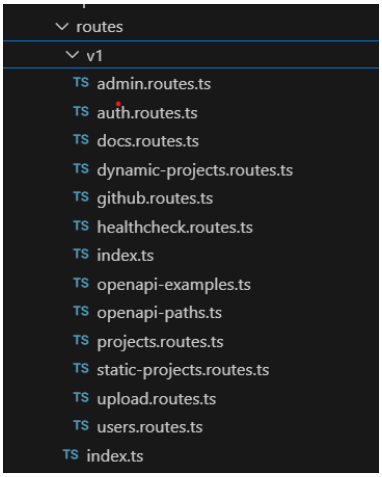
- validateBody(schema: z.ZodType)
  - validates the **request body** (`req.body`) against the provided Zod schema.
- validateParams(schema: z.ZodType)
  - validates **route parameters** (`req.params`) against the provided schema.
- `validateQuery(schema: z.ZodType)`
  - validates **query strings** (`req.query`) against the provided schema.
- validateNumericParams(params: string[])
  - Validates that specific route parameters are numeric.

- validateRequiredQuery(params: string[])
  - Ensures specific query parameters are present in the request.
- validateOptionalQuery(params: string[], numericParams?)
  - Validates optional query params:
    If present, must not be empty.
    If numeric, must be a valid number.
    All must be strings if provided.
- ValidateFile()
  - Ensures a file is uploaded in `req.file` and `projectName` exists in `req.body`.

5. **Models directory**
   It contains files that define the structure of the data and interact with the database. These models represent the application's core data entities, such as users, products, orders, or any other business entities. They include methods for creating, reading, updating, and deleting (CRUD operations) data in the database.
   The entities of the project are Build, Project, and User as shown in the figure below.

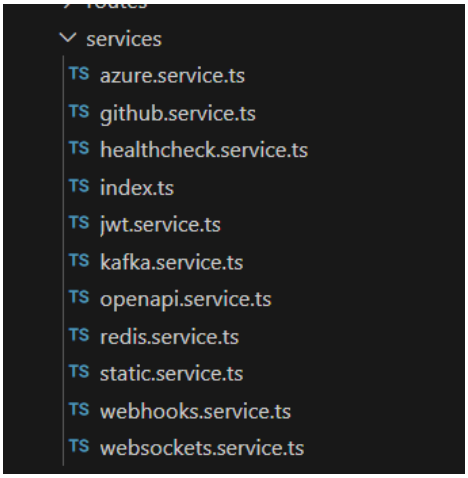


6. **Routes directory**



The “routes” directory contains files that define the application's API endpoints or routes. These routes map HTTP requests (such as GET, POST, PUT, PATCH, DELETE) to specific functions (controllers) that handle the business logic and interact with the database or other services.

7. **Services directory**
   It contains the business logic and background processing of the application. It encapsulates reusable operations like interacting with external APIs (e.g., GitHub), managing job queues, handling authentication (e.g., JWT), processing results from tasks, and serving static content. This layer ensures the application remains modular, maintainable, and scalable by separating core functionalities from controllers and routes.



- *Github.service.ts*

This module facilitates GitHub OAuth authentication and fetches user details. It is essential for integrating GitHub login functionality in the application.

- *Job-queue.ts*
  It manages build jobs using **BullMQ** with a Redis connection. It defines:

  - A job queue (`BUILD_JOB_QUEUE`) connected to Redis (`REDIS_CONNECTION`).
  - A function to immediately drain the queue and ensure all jobs are processed.
  - A function (`addJob`) to add new jobs to the queue, logging the job ID after it's added.

- *Jwt.service.ts*

  This module handles the creation and validation of JWTs for user authentication. It allows the app to generate tokens for authenticated users and verify them to extract user information during requests.

- *Static.service.ts*

  It defines a function `sanitizeEnvVars` within the `StaticProjectService` namespace, which sanitizes environment variables (`envVars`) passed as a parameter. The purpose of this function is to process different types of values in `envVars` and convert them into a format that can be safely stored or used, particularly when dealing with key-value pairs.

- *Azure.service.ts*

  It provides utilities to manage Azure Blob Storage, including setting CORS, generating presigned upload URLs, validating uploaded zip files, and deleting containers.

- *Kafka.service.ts*

It provides health check functions to verify the availability of Kafka, PostgreSQL, and Redis services by attempting connections and logging their status.

- *Openapi.service.ts*

  It defines the OpenAPI 3.1 specification for the AstroCloud API, including endpoints, JWT-based authentication, and schema components for project and build data, tailored for both production and local development environments.

- *Redis.service.ts*

  It manages user-to-container relationships in Redis, allowing linking, validating, and removing container names for users, and includes an initialization method to connect the Redis client.

- *Static.service.t*s

  It converts environment variables to a string format, handling booleans and objects appropriately, for consistent processing or storage.

- *Webhooks.service.ts*

  It handles GitHub webhook events to trigger project rebuilds on push or merged pull requests and updates user data on app installations or removals.

- *Websockets.service.ts*

  It sets up a WebSocket server for streaming real-time build logs, handling authentication, query validation, Kafka log streaming, and emitting log events to authenticated users in project-specific rooms.

Admin Functionalities

Admin Dashboard Endpoints

GET /admin/dashboard
Purpose: Returns statistical summaries for users and projects: Daily, weekly, monthly counts, and a list of the most recent registered users.

GET /admin/users/:id
Purpose: Returns a paginated and searchable list of users.

GET /admin/users
Purpose: Returns a paginated and searchable list of users.

PATCH /admin/users/:id
Purpose: Updates a user's information.

GET /admin/projects
Purpose: Returns a paginated and optionally filtered list of projects.

GET /admin/projects/:id
Purpose: Returns detailed information about a specific project.

POST /admin/projects/:id/rebuild
Purpose: Trigger a rebuild of a specific project

GET /admin/projects/:id/builds
Purpose: Retrieve all builds related to a specific project

Security and validations

- Each endpoint includes basic validation for required fields (e.g. numeric IDs, positive pagination values).
- Error messages are returned with appropriate HTTP status codes.

Validators used

- validateOptionalQuery(keys: string[], numericKeys?: string[]): Validates query strings and checks if numeric ones are valid numbers.
- `validateNumericParams(keys: string[])`: Ensures route parameters are valid numbers.
- `validateBody(schema)`: Validates the request body against a Zod DTO schema.

Authentication and authorization

- Authuard.isAuthenticated: Verifies that the user is logged in
- AuthGuard.isAuthorizedAsAdmin: verifies that user has admin privilege.

How Model-based architecture work

1. **Request Handling:**
   - A request enters the application through the route defined in `routes/routes.ts`.
   - A middleware in `middlewares/` processes the request (e.g., authentication or validation).

- The request is passed to the corresponding controller in `controllers/`, where the main logic resides.

2. **Business Logic:**
   - Controllers may delegate tasks to services in `services/` to handle complex logic (e.g., interacting with GitHub, managing JWTs, or processing queues).
3. **Data Management:**
   - Services or controllers interact with `models/` to query or modify data stored in a database.
4. **Configuration:**
   - `config/` provides settings required throughout the application, ensuring consistency.
5. **Versioning:**
   - ➢ The `routes/v1/` folder enables API versioning, ensuring that breaking changes don't affect existing clients.

Technologies used

Technology	**Purpose**
Node.js	Runs backend logic and API endpoints.
Prisma	ORM for database management.
PostgreSQL	Primary relational database.
Redis	In-memory store for caching and queuing.
Containerizes the backend service.	Containerizes the backend service.
JWT	Manages stateless user authentication.
TypeScript	Adds type safety to JavaScript.

Platform Architecture

Overview

The platform is responsible for handling hosting and uploading web apps. It's where static projects' websites files (HTML, CSS, JS) or dynamic projects' application images are prepared for deployment. Platform automates the process of building applications and deploying them to the cloud.

Services provided:

Platform mainly provides two major services

1. **Static Hosting:** It involves uploading static websites in two methods:
   - **GitHub repository upload**: This lets the user to upload his website through the GitHub repo of it, once platform receives it, it's cloned, then the app is built - if building is needed (React, Angular, Next...) - and the static files are uploaded to the cloud.
   - **Manual upload**: Manual upload lets the user upload his static files directly in a zipped folder, once the folder is received, it's unzipped and uploaded to the cloud.
2. **Dynamic Hosting:** It involves containerizing applications and deploying them to cluster in two methods:
   - **GitHub repository Integration**: This lets the user deploy his application through the GitHub repo of it, once platform receives it, it's cloned, then the app is built. A Dockerfile is constructed and a docker image is built. The image is then pushed to a registry for versioning purposes and rollbacks. Deployment to cluster starts by pulling the image from registry and pods are instantiated with relevant configuration and env variables.
   - **Manual upload**: Manual upload lets the user upload his application directly in a zipped folder, once the folder is received, it's unzipped then built and same process as GitHub unfolds here.

Platform responsibilities

Static Hosting

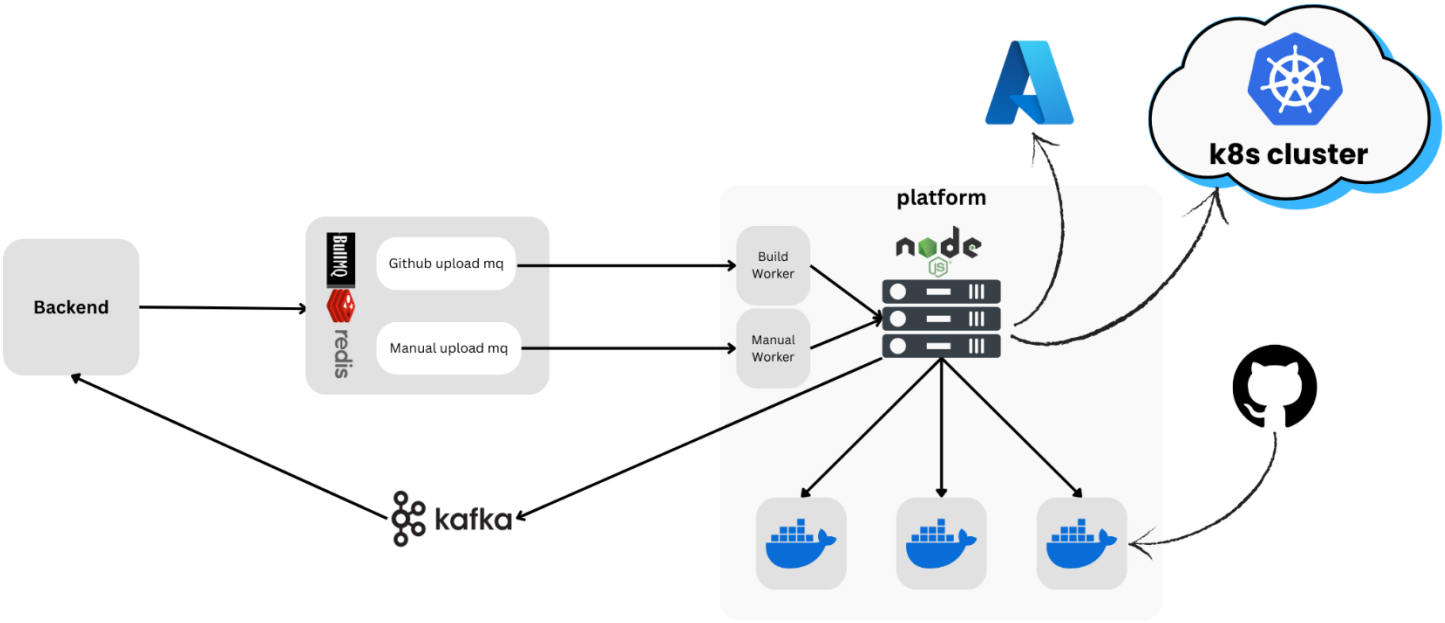
- **Automatic Building**: Detects if the uploaded app requires build (such as React, Angular, Vue...) based on the configurations provided by the user.
- **Build Execution**: Executes build commands (such as *npm run build*) as provided in the configurations for applications that need building.
- **Unzipping folders**: Unzip the files received in manual deployment mode to upload the files to the cloud.
- **Deployment**: Upload the static files to the cloud, whether those files have resulted from the build process, or the app was already prebuilt.
- **Progress and Logs**: Tracks build and deployment progress, logs errors, and provides the logs and status (Success, Failure) to the backend.

Dynamic Hosting

- **Automatic Building**: Detects the app runtime environment (such as Node.js, Python, etc...) and selects build workflow based on the configurations provided by the user.
- **Build Execution**: Executes build commands (such as *npm run build*) as provided in the configurations for applications that need building.
- **Unzipping folders**: Unzip the files received in manual deployment mode to then be build and generate container image.
- **Container Image Generation**: Automatically construct a Dockerfile and build a container image for the app.

- **Deployment**: Uploads the image to a remote image registry, then connects with the Kubernetes cluster to create all the required resources for the deployment, such as Kubernetes deployment, service, ingress, and more.
- **Monitoring and Logging**: Monitor deployed app resources and metrics, stream runtime logs and any errors to the backend.

Architecture



The above figure demonstrates the overall architecture of the platform.

The diagram is simplified and detailed explanations are provided for each component in the following section.

Components and Services

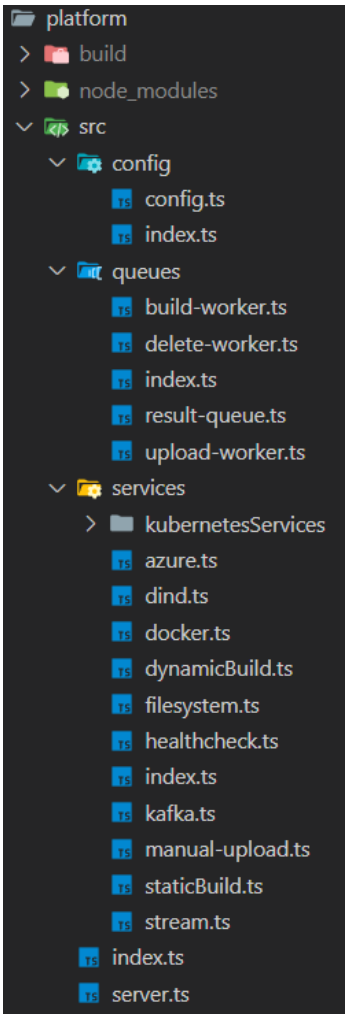
- **Workers:**
  - Workers receive messages from the backend for various tasks, including creating new projects, updating existing ones, deleting projects, or rolling back.
  - **Build worker**: it's responsible for the creation of new projects, both static and dynamic projects, It also handles rolling back dynamic projects' versions.
  - **Upload worker**: It's responsible for the creation of projects from ZIP file uploads for both static and dynamic.
  - **Delete worker**: Responsible for deleting projects and cleaning up their resources.
- **Message Queues:**
  - Message queues are used in the communication between the backend and the platform as the application is working in a microservice manner.
  - **Result message queue**: It's the message queue responsible for sending results from the platform to the backend after performing operations requested.
- **Docker Service:**
  - Docker is used for containerizing the cloning and building of websites intended to be uploaded via GitHub, also for building the docker images for dynamic projects and pushing them to the remote registry.
  - This has several reasons, mainly for security and isolation of the app from the main machine on which the platform is running to avoid security risks.
- **Kafka Service:**

- Kafka is used for streaming the logs resulted during the building of the projects, and for real-time streaming of runtime logs.

- **Manual Upload Service:**
  - It's the service or module responsible for unzipping the static files sent by user in case of manual upload.
- **Azure Service:**
  - Azure service is mainly responsible for creating storage containers for static projects to be hosted on, uploading the files to that storage container, and interacting with the remote image registry to list or delete dynamic projects' images.
- **Kubernetes Service**
  - It is divided into a group of closely related services for connecting with and deploying in Kubernetes. Each service handles one part of the complete dynamic project's deployment process.
  - Each service handles creating new projects, updating or rolling back existing projects, or deleting projects.

In addition to some important services and modules used by Node.JS such as file systems and readable streams.

Structure of Platform



The code structure is divided into 4 main parts (under **src**):

Config

- Contains the configurations and environment variables data needed for the operation of the platform.

Queues

- Houses the logic for the various queues and workers.
- Contains the 3 workers: **build, delete & upload**, and a single queue: **result queue**.

Services

- This contains the main logic used by the platform, divided into services such as docker services, Kubernetes services, and azure service.
- The two main files here are the staticBuild.ts and dynamicBuild.ts. These services are the main orchestrators of the whole deployment process. They call other services as needed.

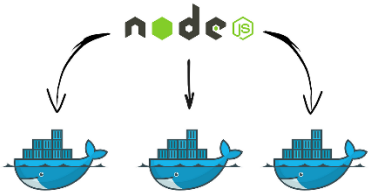
General

- Index.ts is the entrypoint of the platform, which initializes all the components.
- Server.ts starts a local server for health checking purposes.

The platform structure is based on modularity and separation of concerns; each module is responsible for a specific service and group of functionalities. This makes working with a team much easier, coder structure much greater, debugging and fixing errors lots easier and surely helps when adding new features.

Technologies used

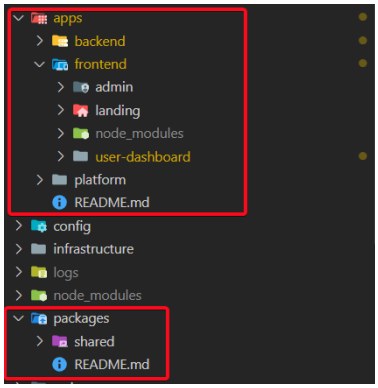
- **NodeJS**: NodeJS is the main language used to implement platform services.
- **Typescript**: used to enhance error debugging and overall code experience.
- **Redis bullmq**: Redis-based message queue bullmq was used as the message queue.
- **Docker**: Docker wasn't only used for containerization of the platform for deployment; Docker was used for the deployment service provided by our platform.
- **Dockerode**: Dockerode library was used to streamline using docker containers and executing commands in the containers programmatically in NodeJS.
- **Kafka**: Apache Kafka is an event streaming service that allows for real-time data streaming in distributed systems and microservice applications.
- **Azure SDK**: Microsoft azure was the main cloud service provider used to deploy static apps on.
- **Kubernetes SDK**: Used to connect to the Kubernetes cluster and send commands for the deployment and monitoring of the dynamic projects.



Shared Code

While designing our system and the different parts of it, we realized that we need to ensure consistency and allow for code reuse between the subsystems. These include unifying the types and interfaces used in data exchanges between frontend, backend and platform, creating shared modules and classes between them, common configurations, and so on.

Mono-repository Setup

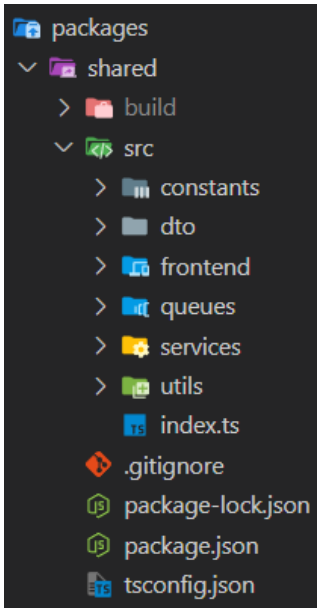


We set up a monorepo architecture to allow for a shared package to be shared between multiple projects easily. We used NPM Workspaces.

We split our code into 2 main folders: apps and packages. Apps are the full applications of our project like the frontends, the backend and platform. Packages are the helper code that is used by the applications.

Now, using this package and sharing code became very natural for the projects. Apps see and import from this shared package just like any other package, streamlining the process and improving the structure of our code, operations, and docker definitions.

Shared Package



We have a central package that contains all the shared code: the **shared** package.

It includes various utility functions, definitions, shared services, constants, and more, such as the DTO definitions, the job types and schema definitions for Redis, services like the logger service, shared constants like dynamic project tiers, kafka topic names, web sockets definitions, and such.

Shared Types

All files define types that are imported from the frontend, backend or platform. For example, **src/queues/jobs-type.ts** define the types and interfaces of a build job put into the message queue, ensuring consistency between what the backend sends and what the platform receives.

Shared types are also used to define the request body and the response body schemas of the REST APIs (in **src/dto**), enforcing the frontend to send the appropriate data as expected by the backend, and typing the response correctly. It also ensures that if the backend expectations are updated, the frontend will be forced to update to it immediately, thus leaving no hidden or unexpected bugs.

Schema Validation

We have implemented schema validators for objects at runtime using **Zod** library in the shared code. These shared validators are used by the Backend to validate that the incoming requests from the frontend conform to the expected schema. If they are not, the backend will refuse to serve the request.

These validators can be used by all the subsystems to unify validation across all communications.

We can construct the TypeScript types and derived types from these schema definitions, streamlining the whole process.

```
export const buildSchema = z.object({
    buildNumber: z.number(),
    projectId: z.number(),
    status: buildStatusSchema,
    logs: z.string().optional(),
    createdAt: z.string().datetime(),
    modifiedAt: z.string().datetime(),
});

export const buildCreateInputDTOSchema = buildSchema.omit({
    buildNumber: true,
    createdAt: true,
    modifiedAt: true,
});

export const buildUpdateInputDTOSchema = buildCreateInputDTOSchema.partial();
```

Managed Kafka Module

The file **src/services/kafka-services.ts** manages communication between different parts of an application using Kafka, a messaging service. It defines a custom ManagedKafka class that extends the Kafka class from the kafkajs library and includes utility functions for generating Kafka topic names and a regular expression for matching Kafka build topics.

ManagedKafka Class

The ManagedKafka class extends the Kafka class and manages a set of connected Kafka clients. It overrides the producer, admin, and consumer methods to add event listeners for connection and disconnection events, maintaining a set of connected clients. The class includes methods to disconnect all clients gracefully and set up handlers for process events and signals to ensure the application shuts down gracefully.

Purpose

The purpose of this file is to manage Kafka clients efficiently, ensuring they are properly connected and disconnected. It also ensures that the application can shut down gracefully, preventing potential issues with Kafka broker connections. This module is used to setup Kafka communications in both backend and platform.

Logger Service

The files in **src/services/logger** are for the logging system used by the Backend. It is a fully-customized logging solution that differentiates between different log levels, print logs in a beautiful and understandable way for developers, and is generally easier to grasp than the native console logs. This service is imported all over the backend, replacing all console.logs.

Constants

Important constants are stored and synced through **src/constants** files. This is the central source of truth all over the system. It contains important configurations and data such as: dynamic hosting runtimes and their versions, and dynamic hosting instance types with full information about their CPU, memory, price, etc.

DevOps Architecture

Deployment

Overview

Our infrastructure is running on a remote VM hosted on **Azure**. Using **Cloudflare** to manage our DNS records and server domain names. SSL certificates are provided by the non-profit organization **Let's Encrypt**. Automating SSL certificates renewal is done by **Certbot**. Incoming/Outgoing requests are served and routed by **NGINX** to either frontend, backend.

Communication Layer Between Services

Each service is deployed as a single container. There exist internal networks among these containers utilizing **docker networks** and they are defined as follows:

1- **Private Network**

   Containers: **backend** – **PostgreSQL** – **Redis** – **Kafka** – **platform**

   Description: This network connects all services needed for backend. It also connects backend together with platform for seamless communication and performance.

2- **Public Network**

   Containers: **backend** – **frontend (nginx)**

   Description: This network is facing the outside network and can be reached on **port 443 https**. It represents the gateway to our whole platform, where requests are routed to its defined destination, and connects the **frontend** with the **backend**.

Deployment Methodology

Docker Compose

- It is a tool for defining and running multi-container applications.
- Simplifies the control of your entire application stack, making it easy to manage services (containers), networks, and volumes in a single, comprehensible YAML configuration file.
- With a single command, you create and start all the services from the YAML configuration file.

In our project, we leverage Docker Compose by maintaining separate configuration files for different environments: one for development and another for production. This document focuses on the production-specific configuration file, docker-compose-prod.yml

Production Docker Compose

The docker-compose-prod.yml file is tailored for the production environment and includes:

- **Service Definitions**: Specifies all services required for the application, along with their respective production-specific environment variables and runtime arguments.
- **Network Configuration**: Establishes the required networks to ensure seamless communication between the services.
- **Secrets Management**: Incorporates secret files used by certain services to handle sensitive data securely.

Production Deployment Steps

1. **Copy Project Files**
   - Transfer all necessary project files to the designated production server. Ensure file integrity and consistency during the transfer process.
2. **Create** .env **Files**
   - Generate .env files for managing environment variables in both the backend and platform directories. Populate these files with the appropriate production-specific values.
3. **Generate SSL Certificates**
   - Execute the init-letsencrypt.sh script to generate SSL certificates, ensuring secure HTTPS communication for the application.
4. **Deploy Using Docker Compose**
   - Run the docker-compose-prod.yml configuration to initialize and launch all services defined for the production environment.
5. **Verify Application Status**
   - Confirm that the application is running successfully by performing functionality checks and ensuring all services are operational.

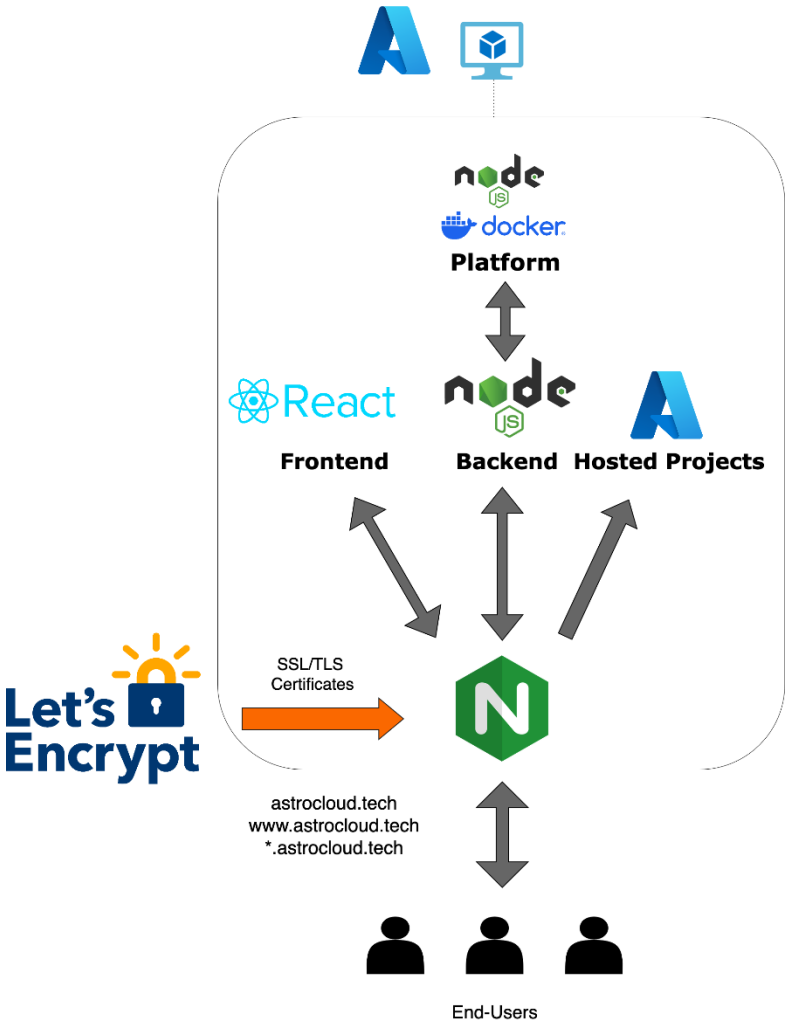


Figure: Deployment Architecture

Kubernetes Cluster

Overview

Kubernetes is selected as the orchestration platform for managing containerized applications hosted for users due to its robust support for scalability, service discovery, automated deployments, and monitoring capabilities. We are using Azure Kubernetes Service (AKS) for its managed DNS and ease of development and testing.

Cluster Setup and Configuration

The cluster consists of a single control-plane node and one worker node, which are scaled up based on resource demand. NGINX is used as an ingress controller managing traffic going in and out of the cluster, allowing/restricting access to resources running inside cluster.

Workloads and Services Deployed

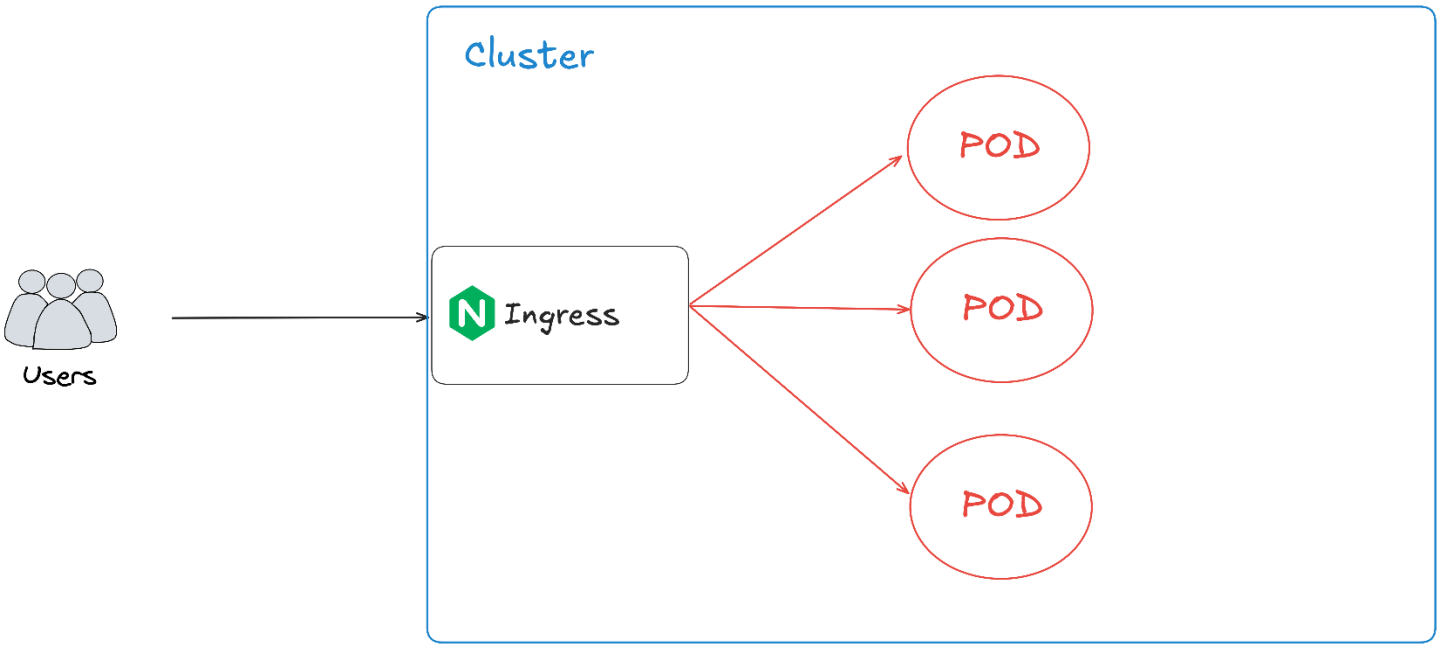
To support dynamic hosting, user applications are deployed on the cluster as containerized services, each accompanied by its own **Deployment**, **ConfigMap**, **Service**, and **Ingress** objects, which manage configuration, networking, and external accessibility.

Networking and Ingress

We configured an NGINX Ingress Controller to route external traffic to internal services. Domain-based routing allowed us to route each user to access his deployed application using unique subdomains.

Monitoring and Logging

Effective observability is crucial for maintaining the health and performance of the system. In our project, we implemented a centralized logging solution using **Fluent Bit** as the log collector on each Kubernetes node.



Dockerization

Introduction

Docker is a powerful open-source platform that revolutionizes how applications are built, shipped, and run. It leverages containerization technology to package applications and their dependencies into isolated and portable units called "containers."

Docker has become an essential tool for streamlining development, deployment, and management of modern applications. It allows us to package our static hosting platform, along with its dependencies, into lightweight and portable units called containers. These containers guarantee consistent environments across different development, testing, and production stages, leading to increased efficiency and reliability.

Why Dockerize Our AstroCloud project?

- **Consistent Environments:** Docker ensures identical environments across development, testing, and production by encapsulating the application code, libraries, and configurations within containers. This eliminates inconsistencies that can arise due to variations in development machine setups.
- **Simplified Deployment:** Docker containers are self-contained units, making deployment a breeze. We can easily deploy our platform to any Docker host without worrying about environment-specific configurations.
- **Scalability:** Scaling our platform becomes effortless with Docker. We can simply spin up additional containers to handle increased traffic or workload demands.
- **Isolation:** Docker containers provide process isolation, preventing conflicts between different services or applications running on the same host. This enhances the overall stability and security of our platform.
- **Version Control:** Docker allows us to version control our container images, enabling us to easily roll back to previous versions if any issues arise.

docker-compose-dev.yml

This is used during local development by developers when there is need to setup optimized for iterative code changes, debugging, and testing.

Services:

- **Frontend**: This service builds and runs the development version of the frontend application. It connects to port 5173 and connects to the public network.
- **Backend**: This service handles server-side logic and database interactions in the development environment. It connects ports to 3000 and 5555. It depends on the db, queue, and kafka services and access essential data in backend_secret.
- **Platform**: This service is of platform in development mode. It depends on the queue and kafka services. It uses secrets stored in platform_secret and connects to the private network.
- **Platform-Webserver**: This service uses a Nginx configuration to serve platform-related static content. It connects to port 80 and mounts configuration files locally for updates. Operates in the public network.
- **Queue**: This service runs Redis in the development environment using the redis:alpine image. It connects port 6379 for caching and queueing, with health checks ensuring availability. Operates in the private network.
- **Database (DB)**: This service provides PostgreSQL in development. It uses the postgres:latest image, connects to port 5432. Health checks ensure readiness before dependencies connect. Operates in the private network.
- **Kafka**: This service runs a Kafka broker for message streaming. It connects to port 9092 and exposes port 9093. Operates in the dev-private network.
- **Smee-Client**: This service is used for CI-CD in development. It uses a custom Dockerfile and operates in the dev-public network.

```yaml
services:
  frontend:
    ...
    ports:
      - "5173:5173"
    networks:
      - public

  backend:
    ...
    ports:
      - "3000:3000"
      - "5555:5555"
    networks:
      - private
      - public
    depends_on:
      db:
        condition: service_healthy
      queue:
        condition: service_healthy
      kafka:
        condition: service_healthy
    secrets:
      - backend_secret

  platform:
    ...
    networks:
      - private
    depends_on:
      queue:
        condition: service_healthy
      kafka:
        condition: service_healthy
    secrets:
      - platform_secret

  platform-webserver:
    extends:
      file: platform/webserver/compose-dev.yml
      service: nginx
    ports:
      - "80:80"
    networks:
      - public

  queue:
    ...
    ports:
      - "6379:6379"
    networks:
      - private

  db:
    ...
    ports:
      - "5432:5432"
    networks:
      - private

  kafka:
    ...
    ports: # Host connects on 9092
      - "9092:9092"
    expose: # Docker containers connect on 9093
      - "9093"
    networks:
      - private

  smee-client:
    ...
    networks:
      - public
    secrets:
      - backend_secret

networks:
  private:
  public:

secrets:
  backend_secret:
    file: ./backend/.env
  platform_secret:
    file: ./platform/.env

```

docker-compose-prod.yml

This is used for live production environments, where stability, security, and performance optimizations are critical and where the result is the application that is accessed by user.

Services:

- **Backend**: This service processes server-side logic and database interactions in production. It exposes ports 3000 and 5555 as for internal communication between services, and it depends on the db, queue, and kafka services and operates on public and private network and uses essential secret data from backend_secret.
- **Platform**: This service handles platform operations in production. It depends on the queue and kafka services. Operates in the private network and uses essential secret data from platform_secret.
- **Queue**: This service runs Redis in production using the redis:alpine image. It exposes port 6379 for caching and queuing. Operates in the private network with health checks configured.
- **Database (DB)**: This service uses the postgres image for the production database. It exposes port 5432. Operates in the private network with health checks enabled.
- **Kafka**: This service provides Kafka for message streaming. It exposes port 9093. Operates in the private network with health checks.
- **Smee-Client**: This service supports webhook forwarding in production. It runs from a custom Dockerfile and operates in the public network using backend secrets.
- **Gateway**: This service acts as a secure entry point for production. It is built from a production-ready Dockerfile. It connects to ports 80 and 443 and relies on backend service. It replaces frontend and webserver services and acts as reverse proxy for backend and platform.it depends on backend service so it will not start until backend service starts.it operates on public network.
- **Certbot**: This service handles SSL certificate generation and renewal. It uses the certbot/dns-cloudflare imag

```yaml
services:
  backend:
    ...
    expose:
      - "3000"
    networks:
      - private
      - public
    depends_on:
      db:
        condition: service_healthy
      queue:
        condition: service_healthy
      kafka:
        condition: service_healthy
    secrets:
      - backend_secret

  platform:
    ...
    networks:
      - private
    depends_on:
      queue:
        condition: service_healthy
      kafka:
        condition: service_healthy
    secrets:
      - platform_secret

  queue:
    ...
    expose:
      - "6379"
    networks:
      - private

  db:
    ...
    expose:
      - "5432"
    networks:
      - private

  kafka:
    ...
    expose:
      - "9093"
    networks:
      - private

  gateway:
    ...
    ports:
      - "80:80"
      - "443:443"
    depends_on:
      - backend
    networks:
      - public

  certbot:
    ...

networks:
  private:
  public:

secrets:
  backend_secret:
    file: ./backend/.env
  platform_secret:
    file: ./platform/.env
```

Third-Party Services

Redis (queue)

Redis server acts as the message queue in the system, decoupling the backend and the platform.

It is accessible from other docker containers via port 6379. It is on the private network as it is a critical component that must be secured.

It is configured with a health check that tests if it is up and running periodically.

```yaml
queue:
  image: redis:alpine
  restart: on-failure
  container_name: redis_mq
  expose:
    - "6379"
  networks:
    - private
  healthcheck:
    test: ["CMD", "redis-cli", "ping"]
    interval: 5s
    timeout: 5s
    retries: 10
```

Postgres (db)

PostgreSQL is the SQL DBMS server used in the system and accessed by the backend.

It is on the private network to secure access to it, and accessible from other containers on port 5432. Environment variables are defined for it to set up its authentication.

It is configured with a health check that tests if it is up and running periodically.

```yaml
db:
  image: postgres
  expose:
    - "5432"
  environment:
    - PGUSER=postgres
    - POSTGRES_USER=postgres
    - POSTGRES_PASSWORD=123
    - POSTGRES_DB=astrocloudd
  networks:
    - private
  healthcheck:
    test: ["CMD-SHELL", "pg_isready"]
    interval: 5s
    timeout: 5s
    retries: 10
```

Kafka

Kafka is the message broker server used by the system to stream events at a high velocity.

It resides in the private network to secure access, is accessed via ports 9092 and 9093.

It is configured with a health check that tests if it is up and running periodically.

A utility service for development only and debugging is kafka-ui container, which inspects the Kafka server and shows the data inside it in a browser tab.

```yaml
services:
  kafka:
    image: apache/kafka
    ports:
      - "9092:9092"
    environment:
    ...
    expose:
      - "9093"
    networks:
      - private
    healthcheck:
      test: ["CMD-SHELL", "/opt/kafka/bin/kafka-broker-api-versions.sh --bootstrap-server kafka:9093 || exit 1"]
      interval: 5s
      timeout: 5s
      retries: 10

  kafka-ui:
    image: ghcr.io/kafbat/kafka-ui:latest
    ports:
      - 8083:8080
    environment:
      DYNAMIC_CONFIG_ENABLED: 'true'
      KAFKA_CLUSTERS_0_NAME: local
      KAFKA_CLUSTERS_0_BOOTSTRAPSERVERS: kafka:9093
    depends_on:
      - kafka
```

Certbot

```
certbot:
    image: certbot/dns-cloudflare
    volumes:
      - ./nginx/ssl:/etc/letsencrypt
      - /etc/letsencrypt/cloudflare.ini:/etc/letsencrypt/cloudflare.ini
    entrypoint: /bin/sh -c "trap exit TERM; while :; do sleep 6h & wait $${!}; done;"
```

This service uses Cloudflare's version of certbot as that is the DNS provider we are using. It has a shared volume for storing the generated SSL certificates and another for reading Cloudflare's API key. It is starting on a continuous 6hr interval to check certificates validity and if needed renew them at once.

Nginx (Platform-webserver)

This service uses nginx as a reverse proxy for users hosted projects on azure blob store. It maps the URLs of the deployments given to users, to the Azure URLs in which the files are hosted.

It is accessed on port 80 (HTTP) and is connected to public network to allow for open access.

```
platform-webserver:
    extends:
      file: platform/webserver/compose-dev.yml
      service: nginx
    ports:
      - "80:80"
    networks:
      - public
```

Smee-Client

Smee is a Webhook payload delivery service for development. It receives Webhook payloads then sends them to your locally running application.

This service uses a Node 22 base image and installs the Smee client service.

It copies a start-smee.sh bash file which loads the correct Webhook proxy URL from the backend environment variables to start the Smee client Webhook listener service and redirect requests to the correct Webhook API handler endpoint.

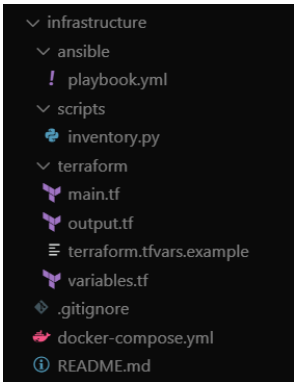
It is necessary for the functionality of the GitHub CI/CD feature to be used while the user is in a development environment to receive Webhook events from GitHub App.

```
smee-client:
  build:
    dockerfile: ./Dockerfile.smee
  networks:
    - public
  secrets:
    - backend_secret
```

```
# Use the official Node.js 22 image as the base image
FROM node:22.11.0-alpine

# Copy the script into the container
COPY ./start-smee.sh /usr/local/bin/start-smee.sh

# Install smee-client and make the script executable
RUN npm install -g smee-client && chmod +x /usr/local/bin/start-smee.sh
```

```
#!/bin/sh

# Read values from the backend_secret
export WEBHOOK_PROXY_URL=$(grep "WEBHOOK_PROXY_URL" /run/secrets/backend_secret | sed 's/^[[:space:]]*WEBHOOK_PROXY_URL[[:space:]]*=[[:space:]]*["]*//; s/["[:space:]]*$//')

# Print the values (for debugging purposes)
echo "Proxy Webhook URL: $WEBHOOK_PROXY_URL"

# Run the smee client with the provided URL
smee -u "$WEBHOOK_PROXY_URL" -t "http://backend-container:3000/v1/github/webhooks"
```

CI/CD



We automated the deployment pipeline from infrastructure provisioning to application launch. Using modern DevOps tools, we ensured a smooth, consistent, and repeatable process that takes the application from code to a live, publicly accessible environment with minimal manual intervention.

This CI/CD (Continuous Integration and Continuous Deployment) process is implemented in the infrastructure folder, which includes all necessary Terraform, Ansible, Docker Compose files and supporting scripts. The entire pipeline is orchestrated through GitHub Actions workflows.

Terraform

To automate infrastructure provisioning on Azure, we used **Terraform** as an Infrastructure as Code (IaC) tool. It allowed us to declaratively define the entire cloud environment required to host our application.

We started by configuring the Azure provider with authentication credentials and subscription information to allow Terraform to interact with Azure services.

Using Terraform; we automated the creation of:

- A **resource group** to hold all infrastructure resources.
- A **virtual network** and a **subnet** to provide a private networking layer for our application.
- A **network security group (NSG)** with inbound and outbound rules to allow SSH, HTTP, and HTTPS traffic to and from the virtual machine.
- A **public IP address** to make our VM publicly accessible.
- A **network interface** with its IP configuration and association with the NSG.
- A **TLS SSH key pair** to securely access the VM.
- A **Linux virtual machine** (Ubuntu 22.04 LTS) as the host for our application, with specified size, admin username, and SSH key-based login.

Ansible

To further automate the configuration and deployment of our application on the virtual machine provisioned using Terraform, we used **Ansible** as a configuration management and provisioning tool. Ansible helped us ensure that the environment was consistently and reliably set up, minimizing manual effort and human error.

Our Ansible playbook performed the following key tasks:

SYSTEM PREPARATION

- Updated the system package cache.
- Installed essential packages such as git, curl, and ca-certificates.

DOCKER INSTALLATION AND SETUP

- Added the official Docker GPG key and repository.
- Installed Docker CE and Docker Compose.
- Added the current user to the Docker group to allow running Docker without sudo.

DOMAIN CONFIGURATION AND DNS UPDATE

- Patched DNS records on **Cloudflare** dynamically using the uri module and an API Bearer token.
- Prepared the necessary DNS configuration for SSL certificate issuance via Let's Encrypt.

SSL CERTIFICATE SETUP

- Created necessary directories and securely placed the Cloudflare API token in a config file.

- Executed a custom script init-letsencrypt.sh asynchronously to generate SSL certificates using Certbot with DNS challenge.

APPLICATION DEPLOYMENT AUTOMATION

- Cloned the latest version of our project repository from GitHub.
- Injected environment variables and secrets securely (e.g., .env files, private keys, kubeconfig).
- Built the Docker Compose production images using docker-compose-prod.yml.
- Deployed the stack in detached mode (docker compose up -d) after ensuring the build and SSL generation completed successfully.

VALIDATION AND VERIFICATION

- Compared the expected list of services defined in the Docker Compose file with the actual running containers using Docker CLI.
- If any containers failed to run, the playbook would fail and report the missing ones.
- Finally, it displayed the list of running containers as confirmation of a successful deployment.

Docker Compose

To simplify and unify the infrastructure provisioning and configuration process, we created a **multi-service Docker Compose workflow** that orchestrates both Terraform and Ansible in isolated container environments. This allowed us to fully automate the deployment pipeline without requiring any software dependencies on the host machine.

Our docker compose defined two services:

1. TERRAFORM SERVICE

- **Image Used**: hashicorp/terraform:latest
- **Working Directory**: /workspace/terraform
- **Mounted Volumes**: Shared both terraform and ansible directories with the container.
- **Responsibilities**:
    - Initialize and apply our Terraform infrastructure definitions.
    - Automatically approve infrastructure changes.
    - Extract dynamic outputs from Terraform (e.g., VM private SSH key and public IP address).
    - Save these outputs into the Ansible directory for use in the next phase.

2. ANSIBLE SERVICE

- **Image Used**: alpine/ansible:latest
- **Working Directory**: /workspace/ansible
- **Depends On**: The terraform service (executes only after infrastructure is successfully provisioned).
- **Environment Variables**: Injects sensitive and deployment-specific data saved as GitHub secrets that will be taken from workflow that runs compose (e.g., Cloudflare API token, GitHub token, environment configs) into the container.
- **Entry Script Actions**:
    - **Waits for the virtual machine to become reachable via SSH**.
    - **Prepares the Ansible inventory dynamically** using the VM's public IP and SSH key extracted earlier.
    - **Executes the Ansible playbook** (playbook.yml) to configure the remote server, install Docker, fetch the codebase, apply environment files, generate SSL certificates, and run the application using Docker Compose.

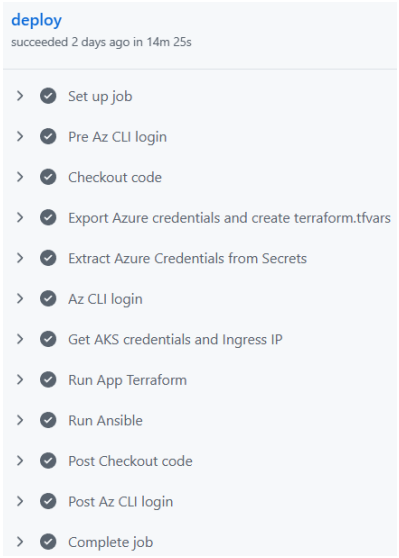
This approach provided multiple advantages:

- **Isolation**: Terraform and Ansible operate in separate containers, ensuring clean, consistent environments.
- **Portability**: The entire deployment pipeline could be executed with a single command on any system with Docker.
- **Security**: Secrets were passed through environment variables and never exposed in code.
- **Efficiency**: Automating the wait for SSH readiness and building the inventory dynamically ensured smooth orchestration without manual steps.

By using Docker Compose as the glue between infrastructure as code and configuration management, we successfully achieved **zero-touch automation** to deploy our application.

Application Workflow

The deployment workflow was designed to streamline and fully automate the application delivery pipeline. It can be triggered with a single click using GitHub Actions **(workflow_dispatch).**



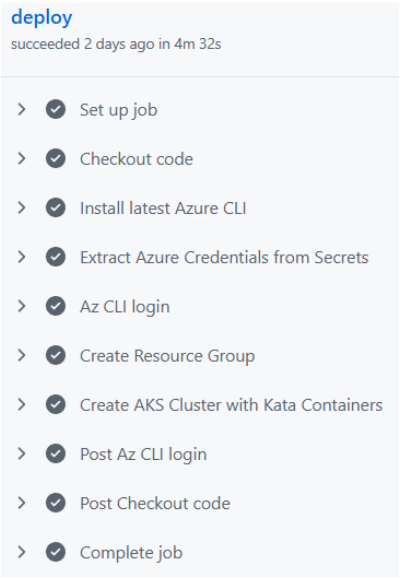
The workflow begins by checking out the code and securely retrieving all necessary secrets required to connect to Azure and to support the Ansible automation process.

Once the environment is prepared, the workflow runs the previously defined Docker Compose setup, which handles both infrastructure provisioning with Terraform and application configuration and deployment with Ansible.

And when are we done? We also have a separate workflow that can tear everything down — deleting the entire application setup with a single click.

Cluster Workflow

The cluster provisioning workflow automates the complete setup of a production-ready Kubernetes environment on **Azure AKS** with **Kata Containers support**. It is triggered manually through GitHub Actions using workflow_dispatch, allowing for controlled and repeatable infrastructure provisioning.



Depending on the selected VM size when running the workflow, the workflow dynamically provisions an AKS cluster with the desired compute capacity and advanced isolation using the Kata workload runtime.

After the cluster is created, the workflow retrieves credentials and configures access for further operations. It labels nodes for workload scheduling and deploys essential components such as the **NGINX Ingress Controller** to support traffic routing into the cluster. It also sets up secure access to the container registry by creating the necessary Kubernetes secrets.

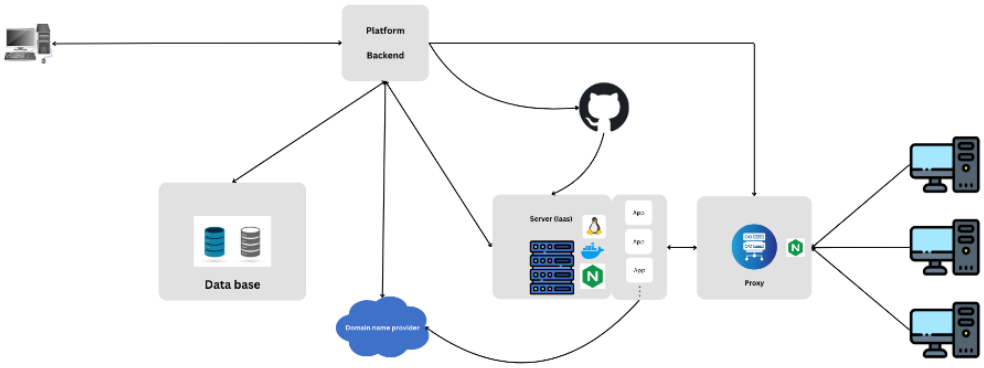
The entire cluster setup process is fully automated, ensuring consistency, reducing the risk of manual errors, and enabling fast provisioning of a secure, container-optimized environment ready for application deployment.

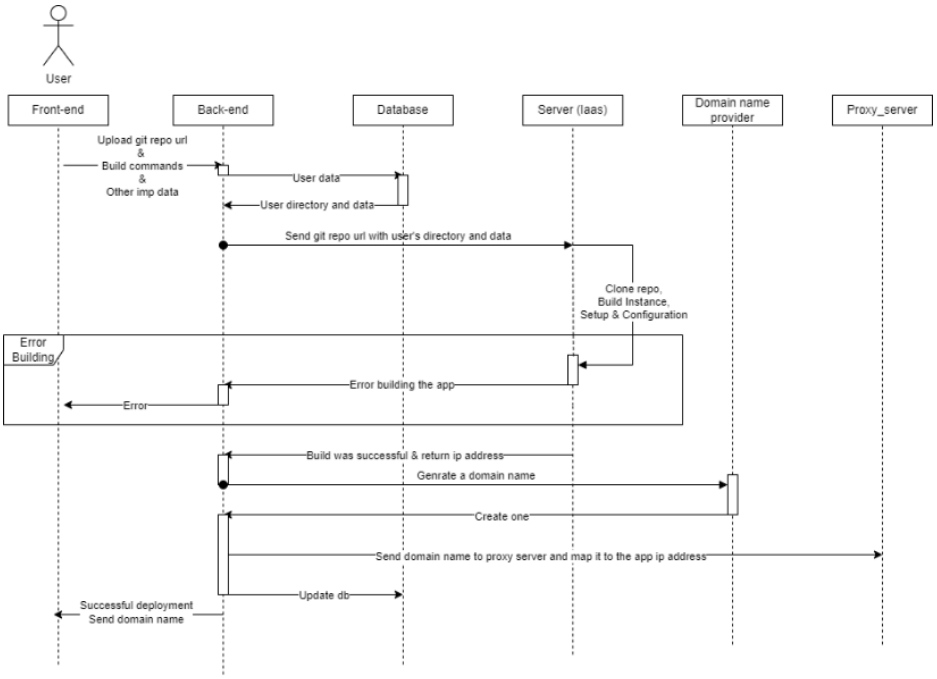
The entire cluster can be deleted with a click of a button using the delete-cluster workflow.

Preliminary Research Milestone

This milestone included:

- **Competitor Analysis**: We conducted an analysis on the top competitors in the field. The results were shown in Competitor Analysis.
- **Gathering Features List**: We gathered all features present in all of the top competitors, and then filtered them to define our project's scope. The results were shown in System Requirements.
- **Field Research**: We explored topics in Web Development, DevOps, and system design. We presented topics in meetings such as: Reverse proxy, event-driven architecture, docker and Kubernetes, microservices, etc.
- **Early Design Phase**: We created some initial designs for various major parts of the system to visualize some of the architecture we will start with.





Static Hosting Milestone

Overview of the main features

Log in with GitHub & GitHub Authentication

High-level description:

This feature allows users to log in to the website using their GitHub account and access their available repositories to choose from.

Importance:

This feature offers an easy way for the user to log in to the website with his GitHub account while also allowing us to list the user repositories for easier access when the user attempts to deploy a project.

Use case example:

A developer chooses to log in with his GitHub account. GitHub authenticates the user then returns his data. The user data is stored in the database. The user is authenticated with the website and his repositories are displayed.

Deploying from GitHub

High-level description:

This feature enables effortless deployment of projects directly from GitHub repositories. By connecting their GitHub account, users can easily select a repository and configure the deployment process with a few simple steps.

Key Configuration Options:

- **Branch:** Specify the branch to build from (e.g., "main," "master").
- **Build Command:** Define the command to execute the build process (e.g., "npm run build," "make build").
- **Build Directory:** Indicate the path to the folder containing the project files (e.g., "/src," "/app").
- **Deployment Directory:** Specify the path to the folder containing the built files ready for deployment (e.g., "/dist," "/build").
- **Site Name:** Provide a unique name for the deployed site (e.g., "my-project," "company-website").
- **Environment Variables:** Set any required environment variables for the deployment environment (e.g., API_KEY, DATABASE_URL).

Importance:

- **Streamlined Deployments:** Automates manual steps, reducing the risk of human error and saving valuable time.
- **Enhanced Developer Experience:** Simplifies the deployment process for both developers and non-technical users.
- **Seamless GitHub Integration:** Integrates seamlessly with existing GitHub workflows for efficient CI/CD pipelines.
- **Increased Reliability:** Reduces manual intervention, minimizing the chance of deployment failures.
- **Versatility:** Supports various project types and hosting needs, providing flexibility for diverse deployment scenarios.

Use case example:

A front-end developer working on a React application hosted on GitHub can easily deploy it using this feature:

1. Connect: Log in to the deployment platform using their GitHub account.

2. Select: Choose the React application repository from their GitHub account.
3. Configure:
    - Branch: Select the "main" branch.
    - Build Command: Specify "npm run build."
    - Build Directory: Set the build directory to "/src."
    - Deployment Directory: Set the deployment directory to "/build."
    - Site Name: Choose "my-react-app" as the site name.
    - Environment Variables: Add necessary environment variables like "REACT_APP_API_KEY."
4. Deploy: Trigger the deployment process. The platform will automatically build the application and deploy it to the specified location.

ZIP File Upload

High-level description:

The manual upload feature enables users to deploy their static websites by uploading a pre-built zip file of their application. This feature is designed for users who prefer to handle the build process for their static website locally or whose applications do not require a build step.

Importance:

This feature is crucial for providing flexibility in deployment workflows. By allowing users to directly upload their pre-built applications, it accommodates:

- **Accelerated Deployment:** For applications already built and ready for production, manual upload offers significantly faster deployment times.
- **Local Development Flexibility:** Ideal for developers who maintain their application code locally outside of platforms like GitHub.

Use case example:

A developer builds their React application locally using npm run build. After building they upload the resulting ZIP file containing the build folder. The platform automatically extracts the files and hosts the website.

GitHub CI/CD

High-level description:

This feature allows users to update their deployed projects automatically when they merge a new pull-request on the deployed branch or when a new commit is detected on said branch.

Importance:

Eliminate manual tracking and updating of deployed projects for every new commit or merged pull-request by allowing real-time updates to changes detected on the deployed GitHub repository.

Use case example:

A developer chooses to deploy a GitHub repository. After the project is deployed, the user attempts to merge a pull-request or make a new commit onto the deployed GitHub repository branch. The system detects these new changes and triggers an automatic rebuild for the deployed project. The user visits the same project website and sees his new changes in action.

Live Build Log Streaming

High-level description:

This feature allows users to see their application’s build progress in real-time as it happens. Instead of waiting for the build to be completed, users can track each step and quickly identify any errors.

Importance:

Real-time log streaming improves user experience by providing transparency and immediate feedback. It helps developers diagnose issues faster, reducing debugging time and deployment delays.

Use case example:

A developer connects their GitHub repository to deploy a React application. As the platform starts building the project, the user sees live logs showing installation progress, build steps, and potential errors. If an issue occurs, they can quickly adjust their code and retry the deployment without waiting for the process to finish blindly.

Subdomain Routing Server

High-level description:

The routing server (NGINX) enables users requests to reach its correct destination and response to be sent back to its correct sender, while implement HTTPS protocol across all communication channels.

Importance:

A web server is critical for hosting and delivering web content, enabling users to access websites and applications over the internet. It processes incoming client requests (e.g., HTTP/HTTPS) and serves static files (HTML, CSS, JavaScript) or dynamic content generated by backend applications. In essence, it acts as the bridge between users and the hosted content, ensuring secure and efficient communication.

Use case example:

A user visiting our website (astrocloud.tech) is first received by the routing server then it looks at the requested domain and according to its configuration it returns the frontend website to the user. Another example is when the user clicks on ‘Sign in with Github’, a request is sent to the routing server which then sends it to the backend and returns the backend’s response to user.

HTTPS

High-level description:

The HTTPS protocol enables users to establish encrypted and secure connections to our platform, ensuring the confidentiality and integrity of their data. This mechanism mitigates critical security risks, such as data interception (sniffing) and man-in-the-middle (MITM) attacks, by encrypting communication between the client and the server

Importance:

As the current standard for secure communication on the web, HTTPS is indispensable for protecting user privacy, authenticating our platform and maintaining trust. It is essential for safeguarding sensitive information, such as login credentials, personal data, and financial details, from potential threats.

Use case example:

A user logging into the platform can do so securely through an HTTPS connection, preventing unauthorized parties from intercepting their credentials or accessing sensitive information during transmission.

Research Phase

Static hosting refers to the practice of serving static files (HTML, CSS, JavaScript, images, and other resources) directly from a server or cloud storage to end-users. Unlike dynamic hosting, where server-side processing generates content dynamically, static hosting delivers pre-rendered content. This simplicity offers several advantages, including faster load times, improved scalability, enhanced security, and reduced costs.

Static hosting is widely used for personal portfolios, blogs, documentation websites, and any application where the content does not change frequently. Modern static hosting solutions often integrate with version control platforms like GitHub for seamless deployment and offer features like custom domains, SSL certificates, and CDN (Content Delivery Network) integration.

Our research has resulted in a reliable system for hosting static websites. Users can easily deploy their websites using two methods:

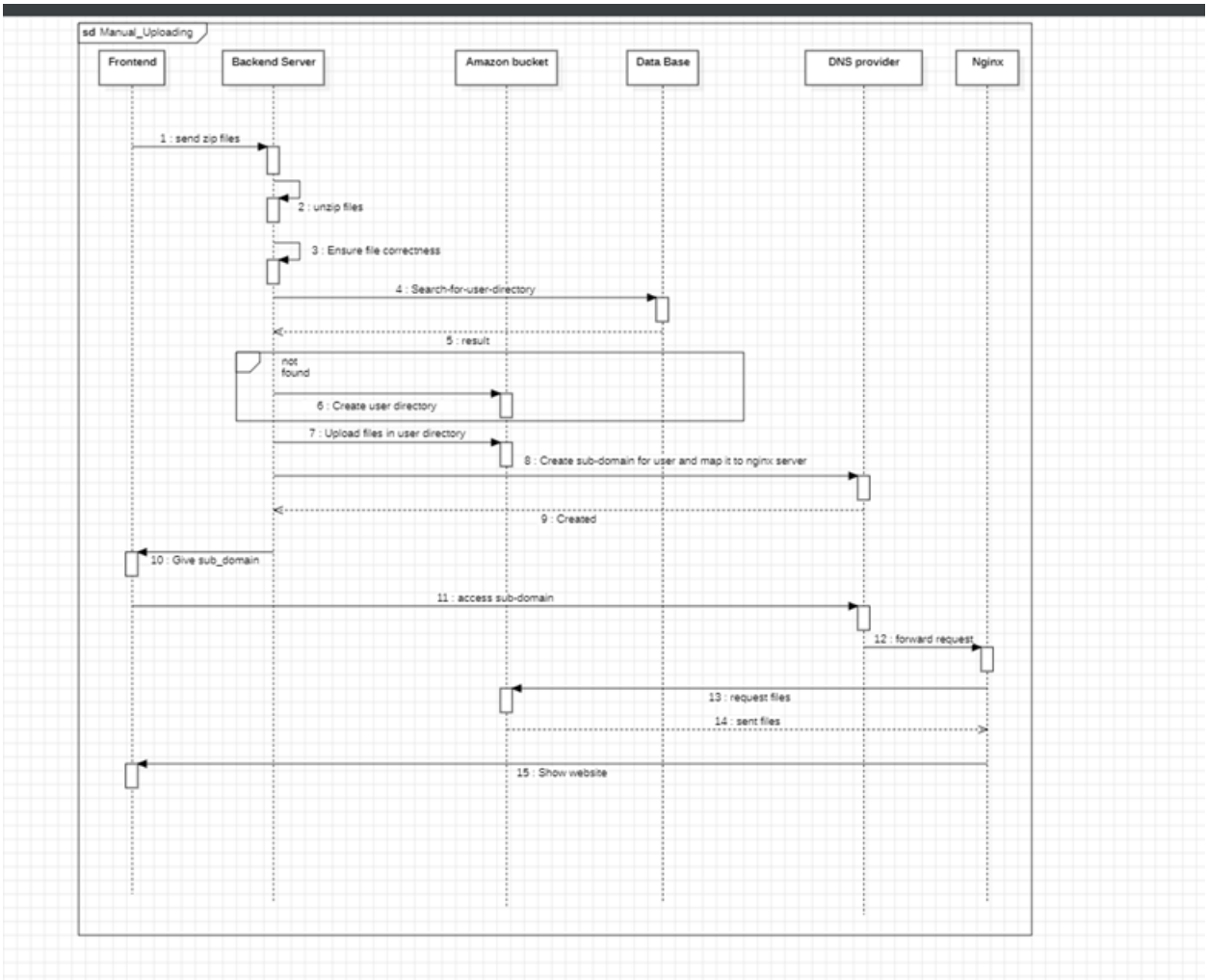
- **Manual Upload:** Upload their project files as a ZIP file for direct control.
- **GitHub Integration:** Connect their GitHub repository for hosting the project.

We will illustrate each feature starting from initial architecture till the final architecture that resulted from our static hosting research.

Manual Upload approaches:

Nginx approach:

- Suitable for smaller websites with limited geographic reach.
- When you need fine-grained control over server configuration.
- When you want to optimize performance on your own servers.



Workflow:

1- Frontend: Send ZIP Files

The user uploads their website as a ZIP file through the frontend interface (Step 1). Then frontend sends the file to the backend server for processing.

2- Backend Server: Unzip Files

The backend extracts the contents of the ZIP file (Step 2).

3- Backend Server: Validate Files

The backend checks the extracted files to ensure they are correct and compatible with the hosting service (Step 3).

4- Backend Server: Check for User Directory

The backend queries the database to see if a directory for the user already exists in the Amazon bucket (Step 4). Then result is returned (Step 5),if the directory does not exist, proceed to Step 6.If it exists, skip to Step 7.

5- Create User Directory

If the directory is not found, the backend creates a new user directory in the Amazon bucket (Step 6).

6- Upload Files

The backend uploads the unzipped files to the user's directory in the Amazon bucket (Step 7).

7-Subdomain Creation

The backend contacts the DNS provider to create a unique subdomain for the user (e.g., username.astroCloud.app) and maps it to the Nginx server (Step 8).

A confirmation is sent once the subdomain is created (Step 9).

8-Provide Subdomain

The backend sends the subdomain link to the user through the frontend (Step 10).

9-Access Subdomain

The user accesses their hosted website by entering the subdomain in their browser (Step 11).

10-Nginx: Forward Request

Nginx receives the request and forwards it to the appropriate directory in the Amazon bucket (Step 12).

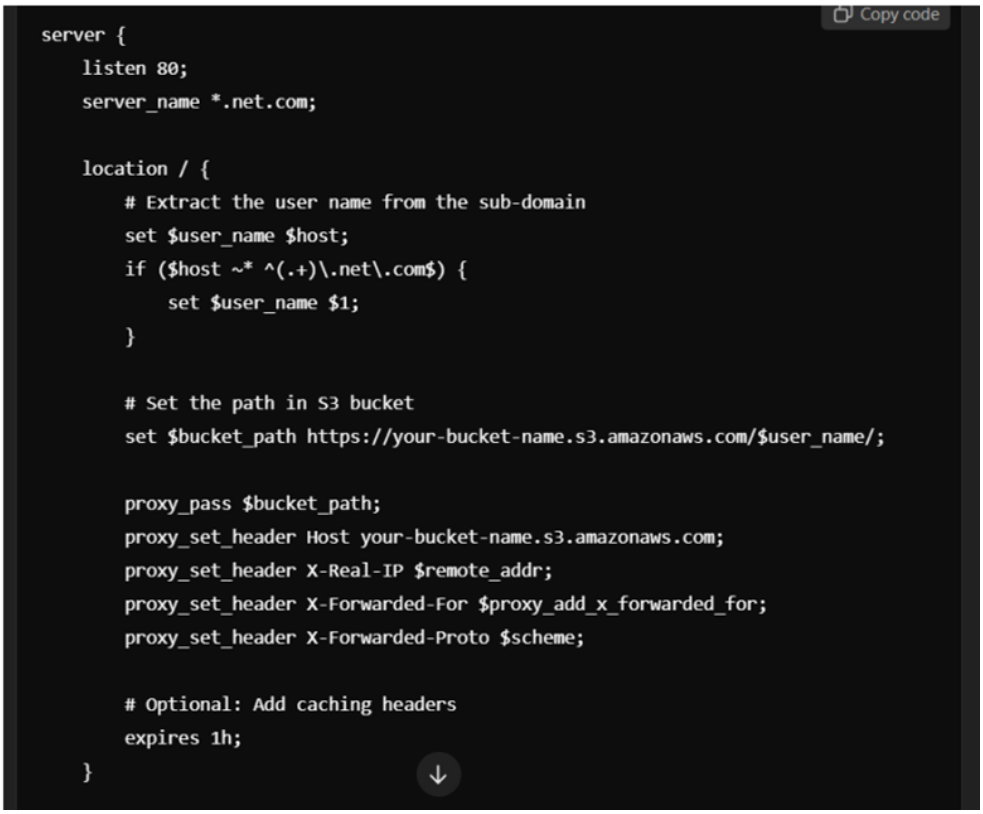
11-Amazon Bucket: Serve Files

Nginx requests the static files from the Amazon bucket (Step 13).Then bucket sends the files back to Nginx (Step 14).

12-Show Website

The requested files are served to the user's browser, displaying the website (Step 15).

Example of nginx configuration:



Explanation:

server_name: Uses wildcard (*.net.com) to handle all sub-domains. set $user_name: Extracts the username from the sub-domain. proxy_pass: Points to the corresponding path in the S3 bucket based on the sub-domain. Nginx Server: May be on Amazon EC2 or DigitalOcean or Linode.

CDN approach:

- Ideal for websites with a global audience.
- When you need to handle high traffic volumes and ensure fast delivery worldwide.
- When you want to improve SEO and reduce origin server load.

Workflow:

1. User Uploads Files

- Users interact with the backend by uploading a ZIP file of their website and specifying a project name.

2. Backend Processing

- Backend unzips the uploaded file and checks if a folder for the user already exists in the cloud storage. If not, it creates a new folder using the provided project name or a unique identifier (UUID). Then uploads the unzipped files to the appropriate folder in cloud storage. Then saves metadata (e.g., folder name, project name) in a database for tracking purposes.

3. Subdomain Generation

- Backend generates a subdomain (e.g., projectName.astroCloud.app) mapped to the specific folder in cloud storage and updates DNS to point the subdomain to the storage bucket via Nginx.

4. Nginx Configuration

- Nginx is dynamically configured to route incoming subdomain requests to the corresponding folder in cloud storage.

5. User Notification

- The backend sends the subdomain URL to the user, providing them with a link to access their hosted static site.

Serving the Static Content

When a client or user accesses the subdomain:

The request is mapped via DNS to the configured Nginx instance. Then nginx fetches the static files from the cloud storage directory mapped to the subdomain. Finally, files are served directly to the client's browser, enabling seamless website access.

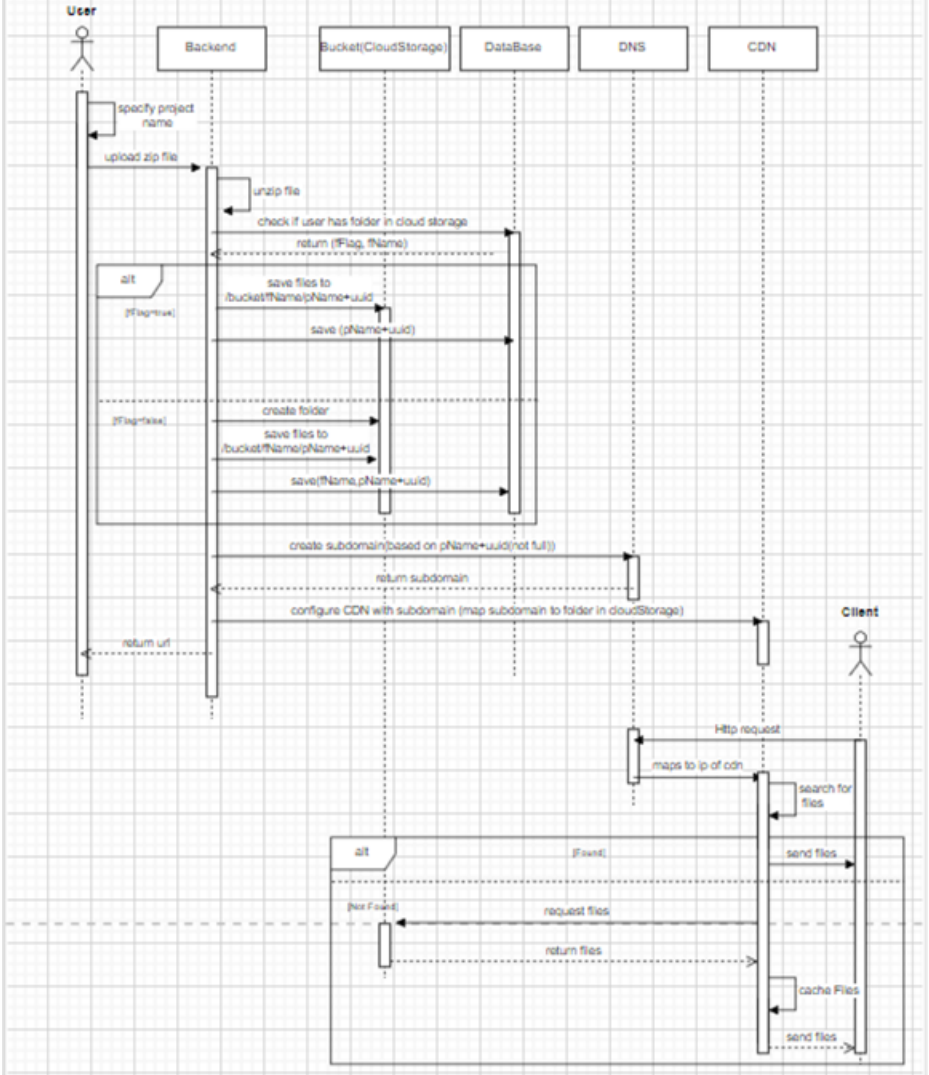
One-Time Effort

1- Create Bucket. (Cloud Storage)
2- Create Domain (astroCloud.app)
3- Issue Wildcard ssl certificate (*.astroCloud.app)
4- Create and configure CDN distribution , associate ssl certificate with it.

Names:

1- fName: folderName
2- fFlag : folderFlag
3- pName:projectName

What Happen



AWS (Amazon Web Services)

1. **Cloud Storage**: Amazon S3
2. **DNS**: Amazon Route 53
3. **CDN**: Amazon CloudFront
4. **SSL Provisioning**: AWS Certificate Manager (ACM)

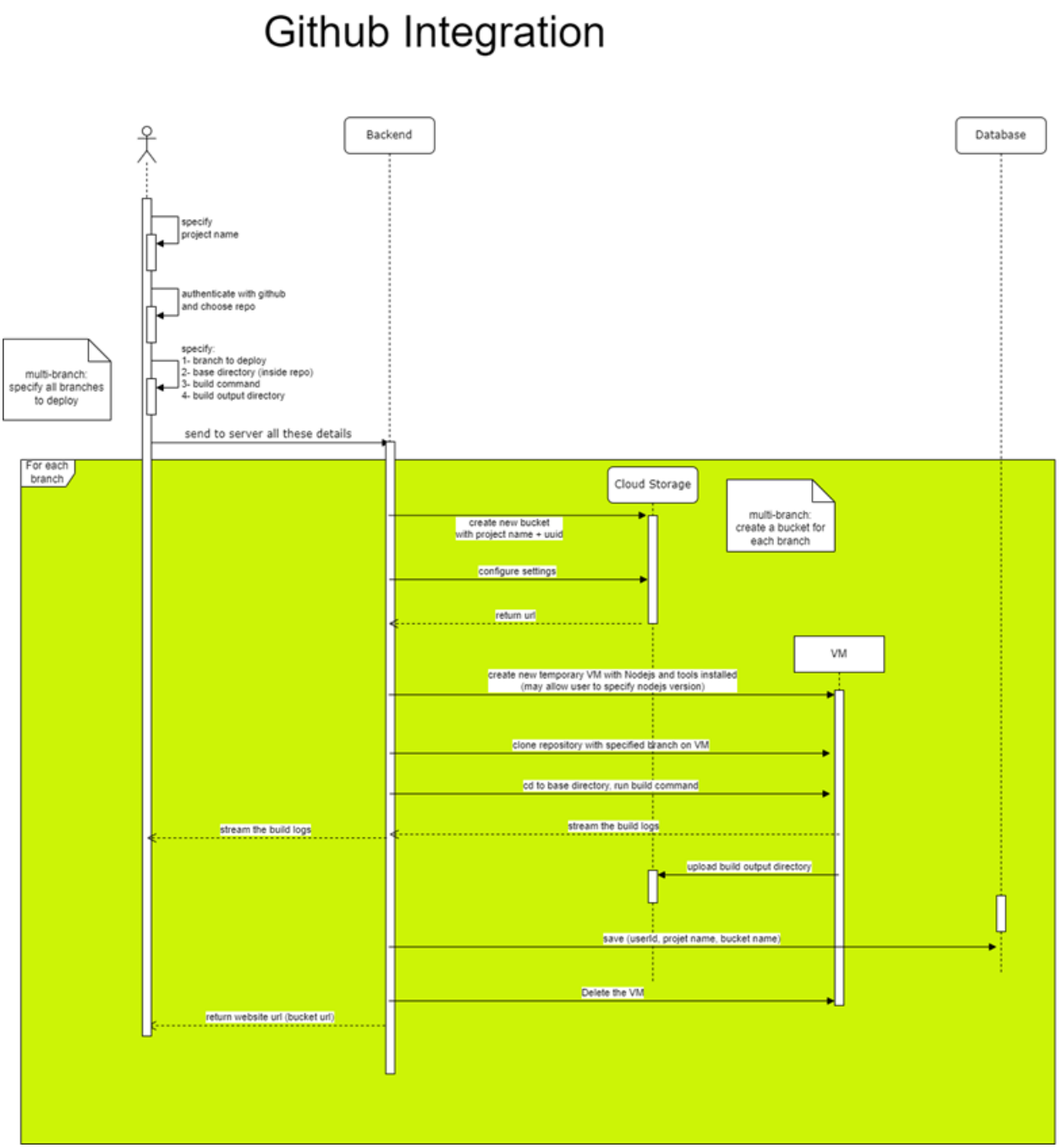
Microsoft Azure

1. **Cloud Storage**: Azure Blob Storage
2. **DNS**: Azure DNS
3. **CDN**: Azure CDN
4. **SSL Provisioning**: Azure Key Vault

Google Cloud

1. **Cloud Storage**: Google Cloud Storage
2. **DNS**: Google Cloud DNS
3. **CDN**: Google Cloud CDN
4. **SSL Provisioning**: Google Cloud Managed SSL Certificates

GitHub Integration approaches:



Workflow:

1-User Interaction:

- The user specifies a project name. Then the user authenticates with GitHub and selects the desired repository. Finally, user provides configuration details:
    - **Multi-branch or Single-branch deployment:** If multi-branch, specify all branches to deploy.
    - **Base directory:** The directory within the repository containing the website's source files.
    - **Build command:** The command to execute to build the website (e.g., npm run build).
    - **Build output directory:** The directory where the built website files are generated.

2-Backend Processing:

- The backend receives the configuration details and creates a new bucket in cloud storage using the project name and a unique identifier (UUID).If multi-branch deployment is selected, the backend creates a sub-bucket for each specified branch.

3-VM Creation and Build Process:

- The backend creates a temporary VM with Node.js and required tools installed.
- Then VM clones the specified repository from GitHub, checking out the selected branch and navigates to the base directory and executes the build command. Then The build logs are streamed back to the user.

4-File Upload and Cleanup:

- The built website files from the output directory are uploaded to the corresponding bucket in cloud storage. Then VM is deleted.

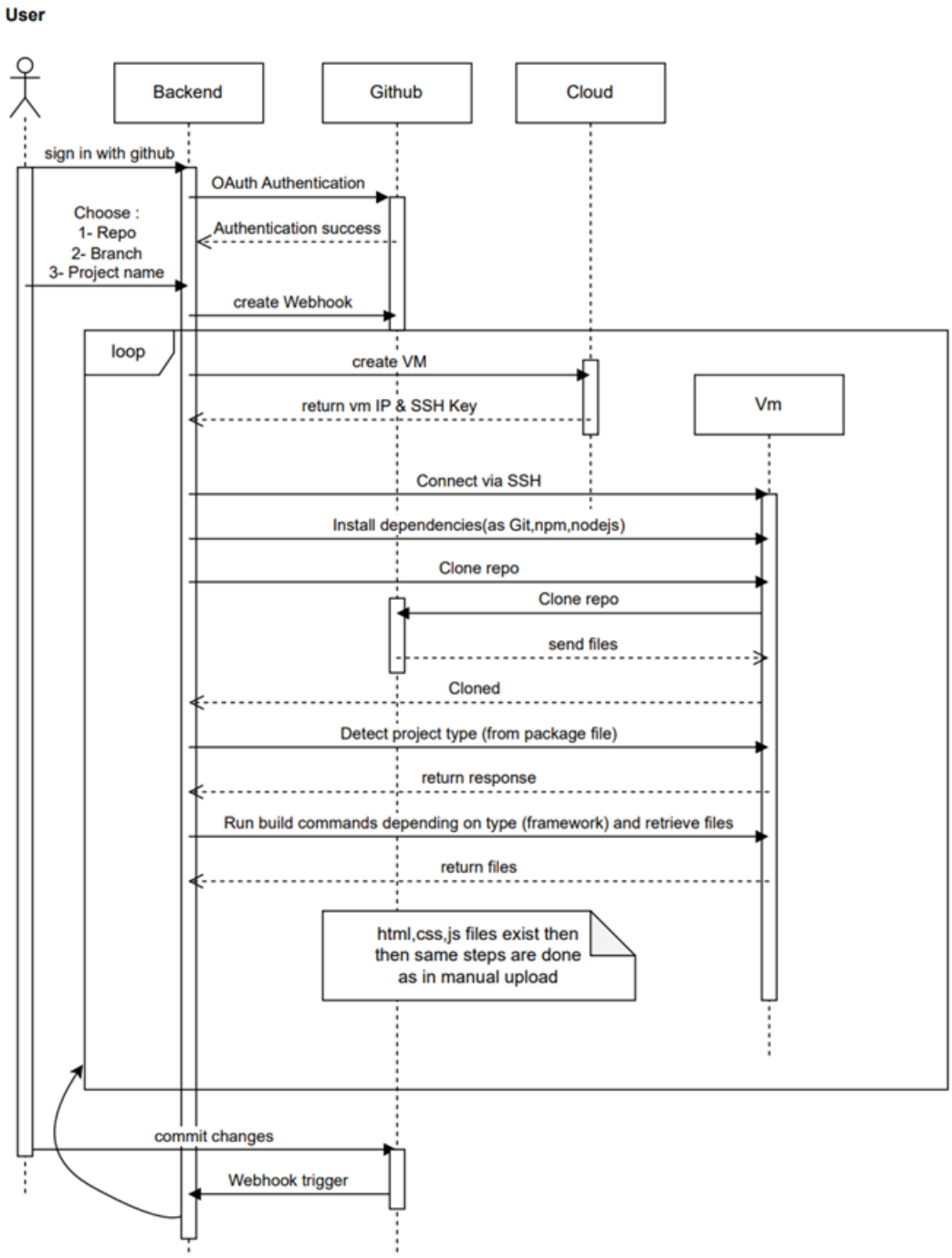
5-Database Interaction:

- The backend saves the user ID, project name, and bucket name (or bucket names for multi-branch) in the database.

6-Website URL Generation:

- The backend generates a URL for the hosted website, which is returned to the user.

GitHub Integration with CI-CD:



Workflow:

User Interaction:

- The user signs in with GitHub. Then user selects a repository, specifies a project name, and chooses the branch to deploy.

Backend Processing:

- The backend authenticates with GitHub using OAuth. Then webhook is created on the specified GitHub repository. Finally, A VM is created.

VM Initialization:

- The backend returns the VM's IP address and SSH key to the user. Then VM connects via SSH and installs necessary dependencies (e.g., Git, Node.js, npm). Finally The VM clones the specified repository from GitHub.

Project Detection and Build:

- The VM detects the project type (e.g., React, Vue, Next.js) based on the package.json file. Then the VM executes the appropriate build commands based on the detected project type. Finally, the built files (HTML, CSS, JavaScript) are returned to the backend.

Deployment:

- If the returned files are HTML, CSS, and JavaScript, the same steps as in the manual upload process are followed (creating a bucket, uploading files, generating a URL).
- The changes are committed to the repository.

Webhook Trigger:

- When changes are pushed to the GitHub repository, the webhook is triggered.Then entire CI/CD process is repeated, ensuring that the website is automatically updated with the latest changes.

Proof of Concepts Phase

After researching static hosting, we divided static hosting features gathered in the research phase to smaller features. The team was divided into sub-teams, with each sub-team assigned a small feature to implement. The proof-of-concept phase was an intermediate phase between researching and implementation, allowing us to explore different solutions and to validate the feasibility of our ideas and refine our approach before proceeding to full implementation.

In this section, we will briefly mention the objectives of each PoC and only dive into three PoCs.

Static Hosting on Azure with server analytics PoC

Objective

- **Automated File Extraction:** Seamlessly extract files from ZIP archives upon upload or delivery.
- **Azure Blob Storage Integration:** Upload extracted files to an Azure Blob Storage.
- **Dynamic NGINX Routing:** Implement dynamic NGINX routing capabilities to dynamically map incoming requests to their respective destinations within Azure Blob Storage without requiring manual configuration file changes.
- **Subdomain-to-Destination Mapping:** Develop a mechanism to automatically calculate and assign subdomains to destinations within Azure Blob Storage, enabling users to access files via intuitive and easily memorable subdomains.
- **GoAccess Integration:** Integrate GoAccess with NGINX to collect and analyze web server logs, providing valuable insights into website traffic patterns, user behavior, and performance metrics.

Static hosting deployment from ZIP on local machine PoC

Objective

- **Extract ZIP Files:** Implement backend functionality to extract uploaded ZIP archives for further processing.
- **Store Files Locally:** Save extracted files in a designated local directory on the backend server.
- **Append Dynamic URL to Hosts File:** Automatically update the system's hosts file with a dynamic URL for user access.
- **Enable Dynamic NGINX Routing:** Configure NGINX to route traffic dynamically, automatically calculating subdomains and destinations.

Static hosting deployment from GitHub link on local machine PoC

Objective

- **Create an Application Directory:** Generate a dedicated directory for each deployed application.
- **Download Git Repositories Locally:** Clone and store user repository from GitHub in the designated directory.
- **Implement Port Mapping via NGINX:** Use NGINX to map incoming requests to the correct local ports for hosted applications.

Monitoring Containers using Prometheus PoC

Objective

- **Monitor Container Performance:** Implement container monitoring using Prometheus to collect key metrics such as CPU utilization, memory usage, and resource consumption for each running container.
- **Visualize Container Metrics:** Utilize Grafana to visualize the collected metrics, providing insightful dashboards for monitoring container performance, identifying resource bottlenecks, and optimizing resource allocation.

- **Evaluate OpenLightSpeed as a Web Server:** Evaluate the suitability of OpenLightSpeed as a lightweight and efficient web server for local deployment of applications.

Logs streaming for static deployment PoC

Objective

- **Local Static Deployment:**
  - Establish a local development environment for static website hosting using the Apache web server.
  - Configure Apache to serve static files (HTML, CSS, JavaScript, images) efficiently.
- **Real-time Log Streaming:**
  - Implement a mechanism for real-time streaming of server logs to the front-end application.
  - Explore and evaluate WebSockets for log streaming.

Building framework PoC

Objective

- **Frontend File Upload:** Implement a user interface for uploading framework files (e.g., ZIP) through the frontend.
- **Backend File Processing:** Develop backend logic to receive and process uploaded files and extract files from uploaded archives.
- **Azure Container Provisioning:** Create a new Azure container dynamically for each project.
- **Framework Build and Deployment:**
  - Build the framework using the provided commands from user.
  - Upload the built files to the designated Azure Blob Storage container, frontend receives container URL for user to access deployed website.

Web Page Prerendering PoC

Objective

The proof of concept (POC) for prerendering static websites involves using Caddy as a reverse proxy to handle client and bot requests intelligently, and integrating a Prerender service to improve SEO(Search Engine Optimization) and ensure proper indexing by search engines.

Implementation

Architecture

- Caddy serves as a reverse proxy.
- The Prerender service handles requests from bots and crawlers by rendering JavaScript-heavy pages into static HTML.
- The setup includes a frontend application, a backend server, and Dockerized deployment.

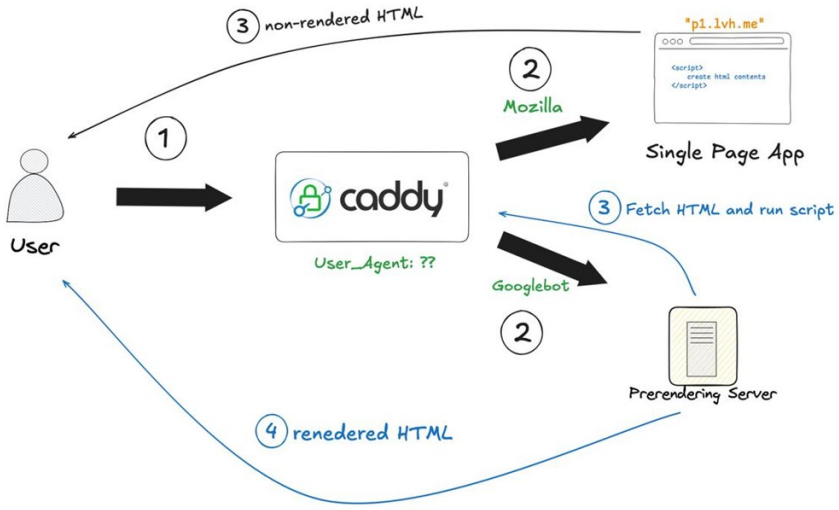
Workflow

- Caddy routes requests based on the User-Agent header:
  - **For bots/crawlers**: Requests are forwarded to the Prerender service, which fetches and renders HTML for better SEO.
  - **For users**: Content is served directly from /srv/{subdomain}.
- Static content is stored under /srv/{subdomain}, enabling domain-based organization.

Outcome

- **Enhanced SEO**: Bots and crawlers receive prerendered HTML, ensuring proper indexing of JavaScript-based pages.
- **Efficient Resource Allocation**: Caddy intelligently differentiates between user and bot requests, minimizing load on the Prerender service.
- **Dynamic Subdomain Support**: The setup supports serving static files for different subdomains under /srv/{subdomain}.

Diagram

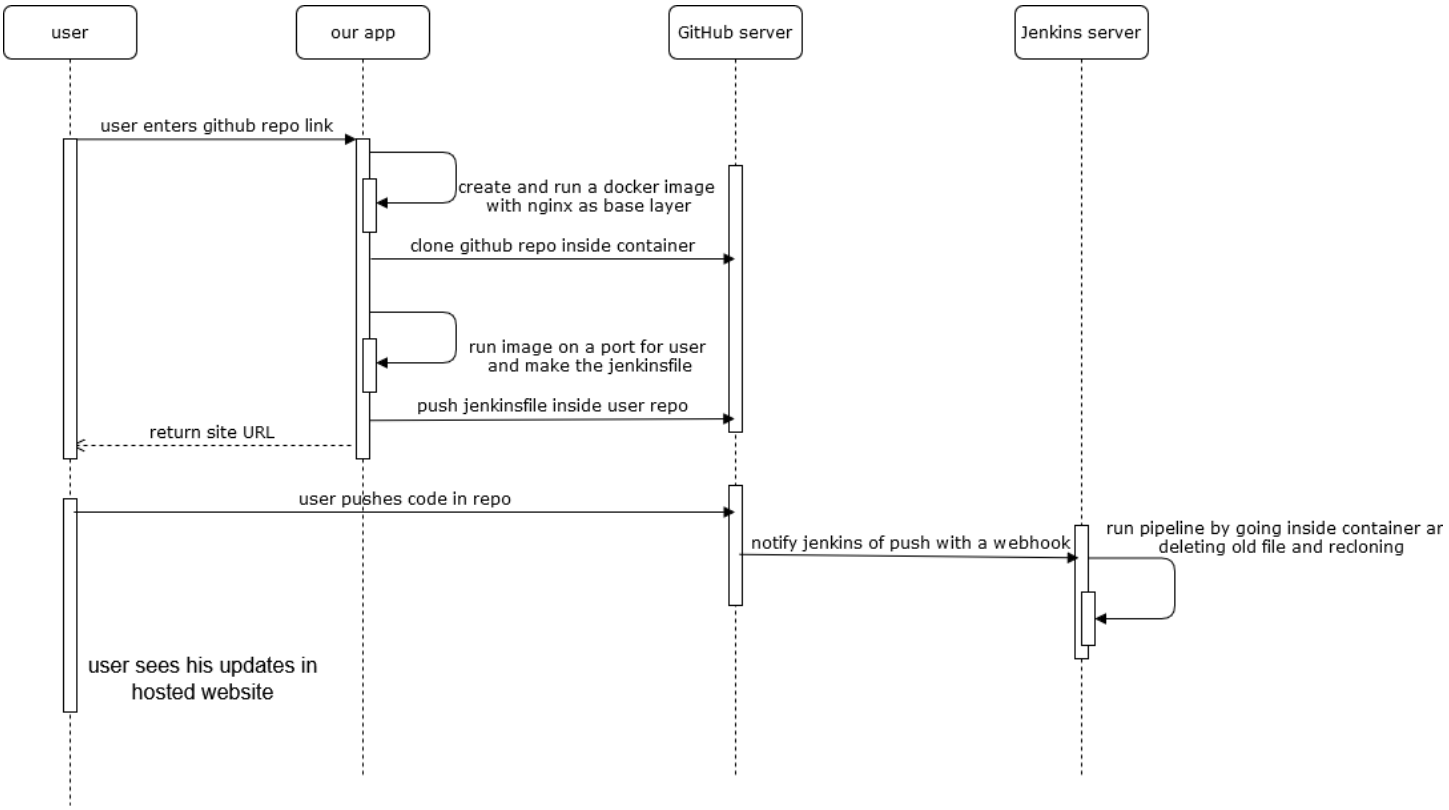


CI/CD using Jenkins PoC

Objective

This proof of concept (PoC) focuses on utilizing Jenkins for continuous integration and delivery (CI/CD) and integrating it with a dynamic hosting solution for users to host their static websites.

Sequence diagram



Implementation

- **Developed a user interface:** Created a web interface with a text input field for users to enter their GitHub repository URL, including a personal access token for secure access.
- **Implemented dynamic containerization:** Built Docker containers for each user based on a template Dockerfile, dynamically updated with user-specific data (repository URL, token, ID, port).
  Within the container:
  - The user's repository is cloned.
  - A Nginx server is configured to serve the static files on the designated port allocated to the user.
- **Integrated with Jenkins:** Configured a Jenkins pipeline manually to monitor user repositories and trigger builds upon updates.
- **webhooks:** Jenkins link was manually added to the GitHub webhook in user repository, when user pushes in main branch, a webhook event is received by Jenkins, which subsequently initiates the pipeline
- **Implemented dynamic Jenkinsfile generation:** Created Jenkinsfiles for each user based on a template, dynamically incorporating user-specific data. Each Jenkinsfile contains the pipeline script that executes when triggered. The script performs actions within the user's container, such as deleting outdated files and recloning the repository, ensuring that the hosted environment seamlessly reflects the latest content from the user's repository.

Outcome

Successfully demonstrated the viability of using Jenkins for CI/CD workflows and dynamic containerization to host static websites. Additionally, identified practical applications of Jenkins within the project's scope to enhance automation and streamline deployment processes.

Event-driven GitHub Repository Building PoC

Objective

- Discover Event-driven architecture and benefits of decoupling the API services from the platform services.
- Learn how Redis and message queues can be used to create a microservices architecture.
- Test out building frontend frameworks and deploying their output to the cloud.

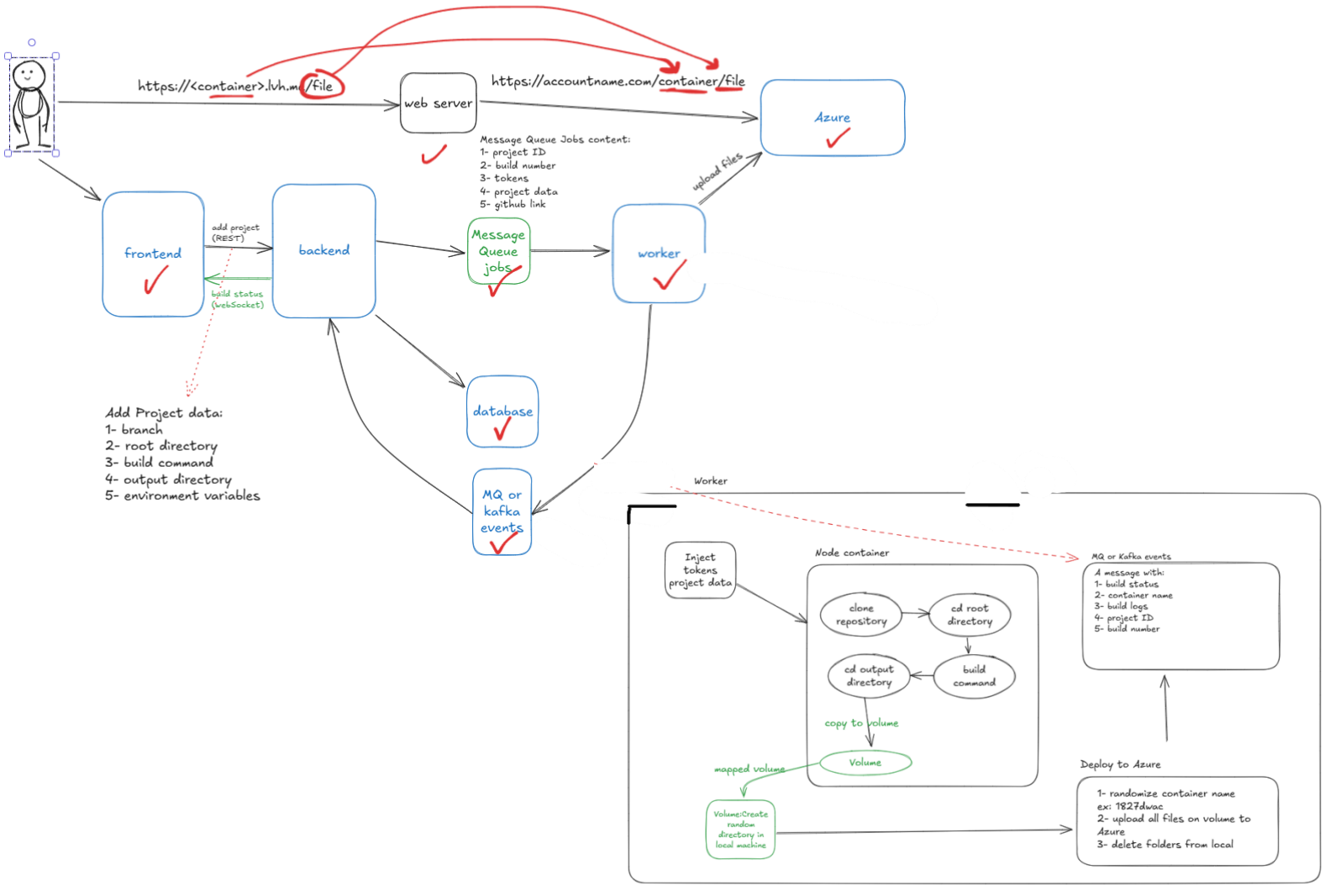
Implementation

- Created a separate Node.js service for the API backend and for the platform.
- Setup Redis server as message queue and integrate it into the backend and the platform.
- Built Platform service on top of a Docker-in-Docker image, and have the platform create a docker container at runtime for each build job.
- Designed the build job to clone the repository, go into root directory, run the provided build command, copy the build output to outside of the container, then upload to the cloud.

Outcome

- Architecture and design used in this PoC was the starting point of the project.
- Learned how to decouple services with message queues.
- Provided a reference for how to do GitHub repo deployment feature.

Diagram



Implementation Phase

The implementation of the static hosting was carried out in two focused iterations, each spanning 1–2 weeks. This phased approach ensured steady progress, adaptability, and the ability to incorporate feedback at every stage. Each iteration was meticulously planned to achieve specific milestones while maintaining alignment with the overall project objectives.

Iteration 1

Overview

In iteration one, we focused on implementing the core features of static hosting. Unlike the PoC phase, where work was more exploratory, the entire team collaborated in three dedicated teams: backend, frontend, and platform.

Project Initiation Process

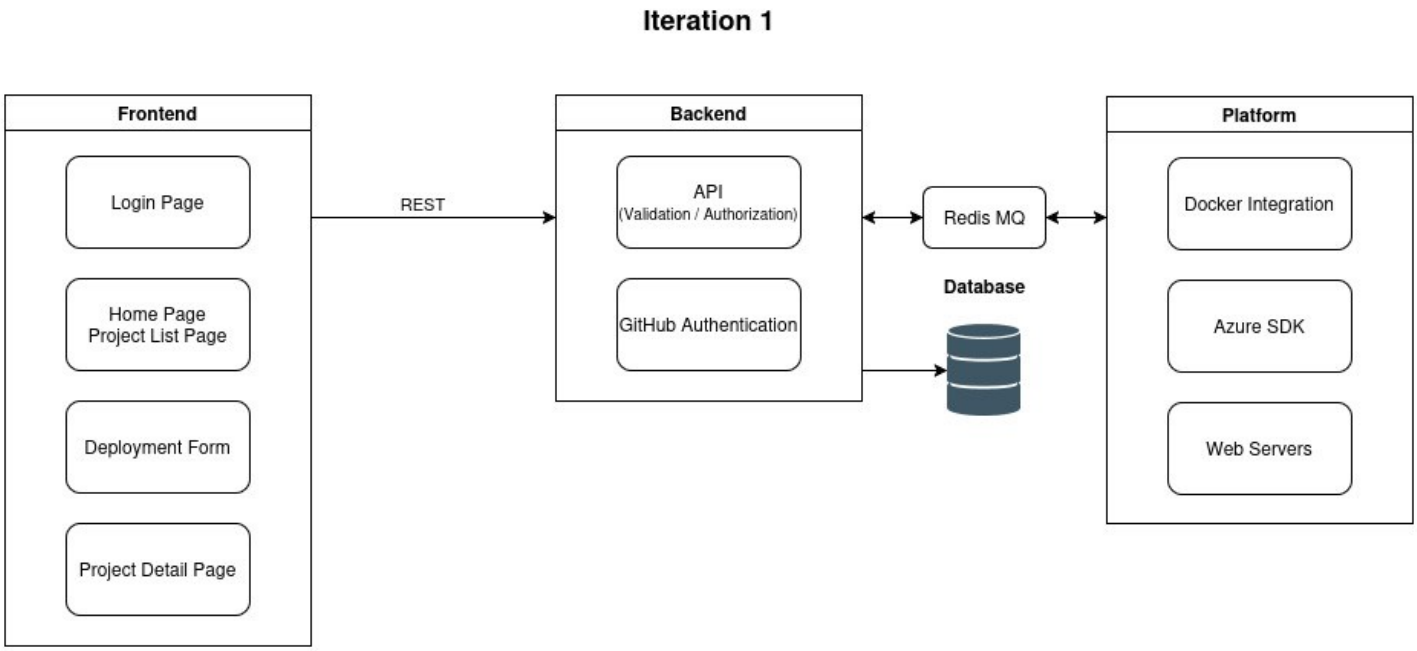
To establish a strong foundation, we initiated the project structure and hierarchy for each subproject:

1. **Repository Setup:** We created a separate folder for backend, frontend, platform and shared code, initialized the folder structures and initial code design, and ensured a modular and maintainable codebase.
2. **Development Standards:** A set of coding guidelines, branching strategies, and review processes were defined to maintain code quality and consistency.
3. **Environment Configuration:** We established local development environments using Docker to standardize dependency management and streamline collaboration.

The key features we prioritized in this phase were:

- **GitHub Login and Integration** – Enabling users to authenticate and link their repositories to our platform.
- **Building and Deploying GitHub Repositories** – Automating the build process for frontend projects based on various frameworks.
- **Subdomain Routing Server** – Implementing dynamic subdomain allocation for deployed projects, ensuring seamless accessibility.

By the end of iteration one, we had established a functional static hosting platform with essential integrations, setting the stage for the next phase of development.



Next we discuss the implementation details of the major features in this iteration: Login with GitHub, Deploy GitHub Repository, and Subdomain Routing Server.

Login with GitHub & GitHub Authentication

GitHub OAuth app

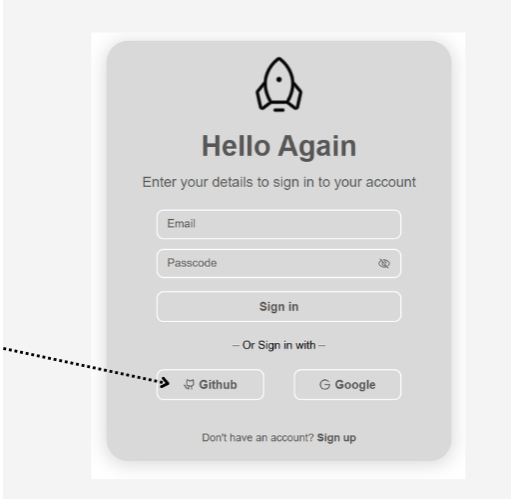
To enable GitHub Authentication, we first needed to create a GitHub OAuth app for our project's identity; AstroCloud. This tells the user who exactly is requesting access.

In the OAuth app, we also define the redirect URL that users will be redirected to after the consent screen.

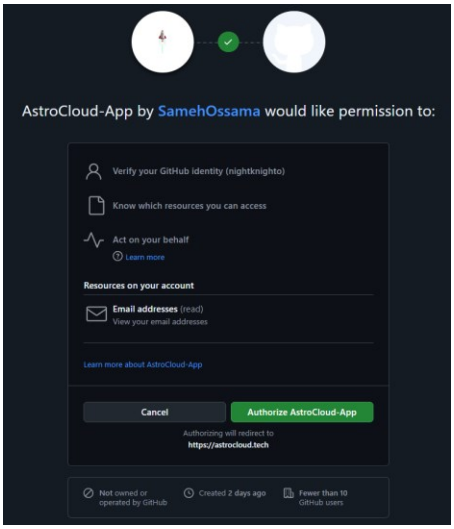
GitHub OAuth app was later updated to GitHub Apps, as discussed in "Migrating from GitHub OAuth to GitHub App" research. However, the process is still mostly the same.

Workflow

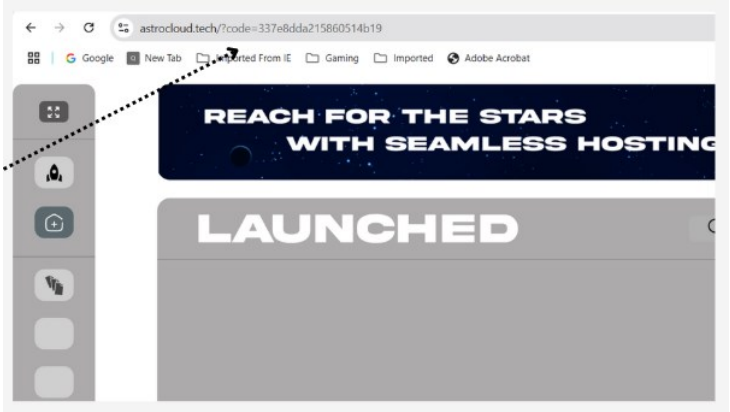
1. The user initiates the Authentication process by clicking the "GitHub" button in login form.



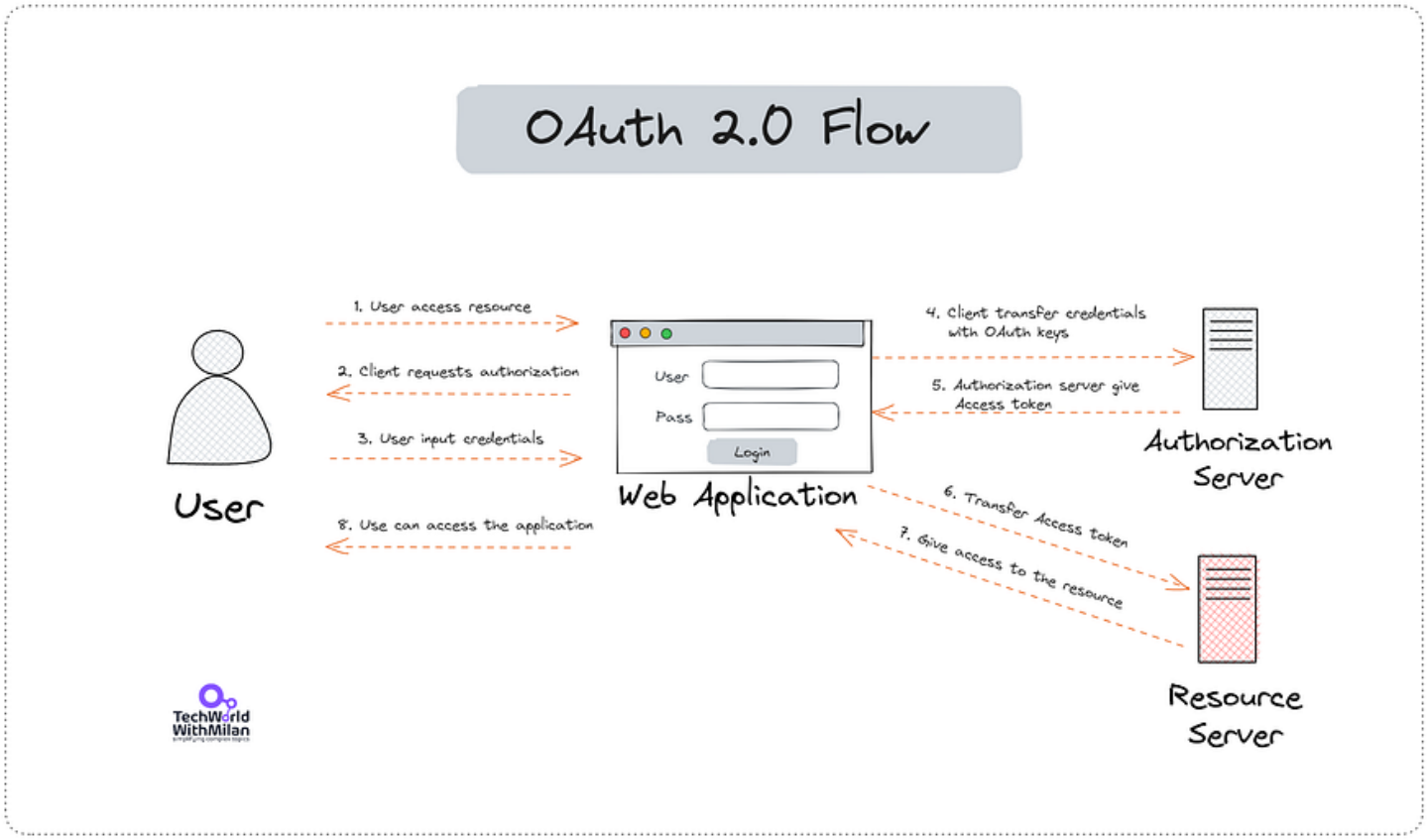
2. **Consent Screen**: The user is redirected to GitHub, where they are asked for permission to share their data.



3. **Authorization Code**: After granting consent, the user is redirected to home page authorization code appended to the redirect URL.



4. Frontend send the code from the redirect URL to backend.
5. The backend sends the code to GitHub's endpoint to exchange it for:
    - **Access Token**
    - The authenticated user's data (e.g., **GitHub username**, profile info).
6. If the retrieved GitHub username exists in the database, the associated user in the database is updated with the new access token. If the username does not exist, a new user is created with the data.
7. The backend then signs a JWT that includes the user ID and sends it back to the frontend.
8. On all further interactions with the backend, the JWT is sent along with the request to provide the user identity. The backend prohibits requests that are unauthenticated or have invalid or expired JWTs. The access token of the user is also passed to the platform when needed to authenticate cloning private repositories of the user.



Build & Deploy GitHub Repository

Deploying a website using GitHub lets the users and developers upload their websites to the cloud and host them using the GitHub repository of the web app.

It's an essential feature for a hosting service; it makes deployment of websites much easier and straight forward as most of the developer use GitHub as version control system for their apps and work, thus letting developers to upload directly using GitHub is a necessity for static hosting platform.

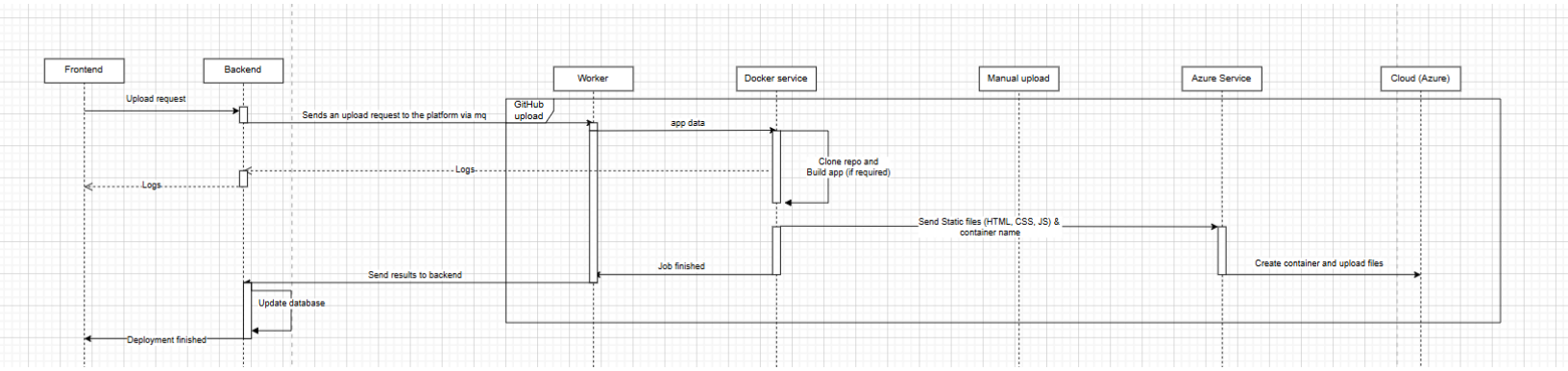
Workflow:

1- User must be logged-in using GitHub in order to be able to upload his web apps repos.
2- User selects the GitHub deployment option and starts inserting web app GitHub repository URL and other configurations including (git branch, build command, build directory…) as well as some environment variables (if exists).
3- Backend receives the request via restful API and firstly creates a new project entry for that user in the database, following it generates a container name (to host the website on it on azure cloud), then the request is directed to the platform via message queues where all the action starts.

The following occurs in the platform servers:

1. Platform receives the upload job via the message queue:
    a. Receive upload request from the backend using the message queue.
    b. The process of building and uploading starts from the GitHub deployment worker which is the responsible worker for the GitHub uploads.
    c. The request message usually contains the project id, the app git repository URL and app configurations (build command, GitHub branch …) as well as environment variables (if exists).
2. Cloning App (for GitHub upload):
    a. If the intended service used was GitHub upload, first thing to take place is to create a docker container on the local machine and start cloning the app repo inside of it.
    b. Whether a build is required or not is detected by the configurations sent by the user in the upload request.
3. Building:
    a. In case build was needed, the necessary dependencies are pulled (npm install) then the build command in the configurations (npm run build) is executed and building takes place.
    b. Building logs are monitored, stored and streamed back to the backend so it can be sent to the user in a live manner.
    c. Once the building is finished, the build files are copied from the container to the local machine to be deployed. Docker Binding was used to transfer files from the docker container to the local machine.
    d. Once the build is completed and files are stored, the docker container is shut down and killed.
4. Deployment:
    a. The static files (generated from build process or was uploaded by user directly) are uploaded to the cloud.
    b. A new storage container is created on azure for that application and static files are pushed to that container.
    c. Each app is uploaded to its specific container name provided by the backend.
5. Respond to backend:
    a. Once the build and upload are finished, the results are sent to the backend.
    b. Results usually include the status of deployment (success, Failure…), build number (if exists), building logs (if exists) and app id.

6. Update database:
    a. The backend server updates the database and adds the status of the website deployment for that user.



Subdomain Routing Server

Using NGINX as a web server fulfills two functions. Number one is reverse proxying for backend API requests, users API requests are first received by NGINX then sent by NGINX to backend. Also, works as a reverse proxy for users hosted websites. Number two, it works as a file server for frontend static files. It serves frontend files upon users visiting our website.

Frontend

URL: **astrocloud.tech – www.astrocloud.tech**

- Requests with above URL header are served frontend static files.

Backend

URL: **api.astrocloud.tech**

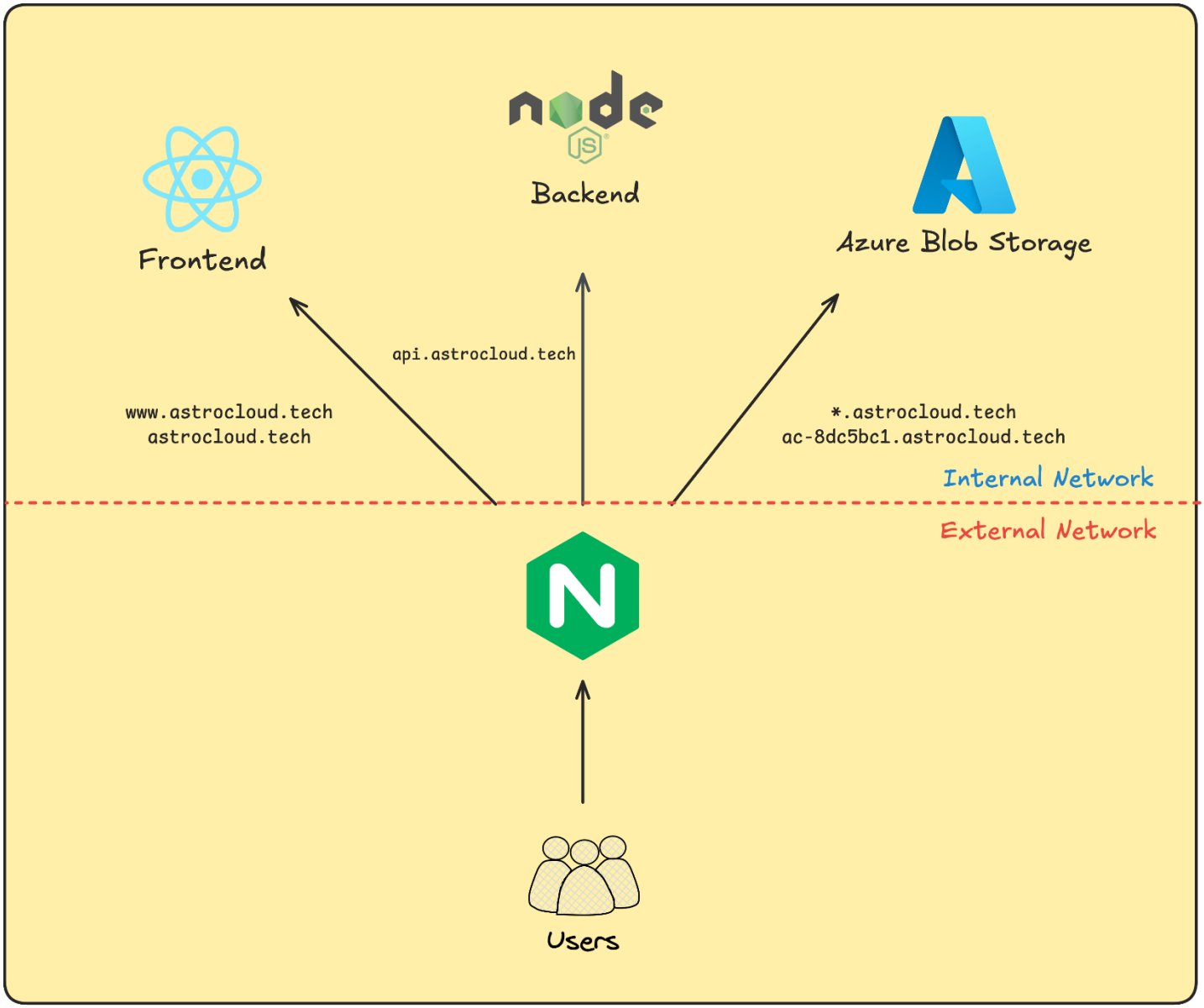
- Requests with above URL header are reverse proxied to backend server.

Azure Blob Storage (Hosted Projects)

URL: ***.astrocloud.tech**

- Requests with above URL header are reverse proxied to Azure Blob Storage container of hosted project.
- Example: **ac-8dc5bc1.astrocloud.tech**
- The subdomain part **ac-8dc5bc1** is used to point to the Azure container's id
- The final proxied URL is something like **<StorageAccountName>.blob.core.windows.net/ac-8dc5bc1/index.html**

This a descriptive figure for different routing scenarios


Backend
Frontend
Azure Blob Storage
api.astrocloud.tech
www.astrocloud.tech
astrocloud.tech
*.astrocloud.tech
ac-8dc5bc1.astrocloud.tech
Internal Network
External Network
Users

Iteration Outcomes

This iteration marks the start of the static hosting milestone.

With it, we have implemented:

- The starting API for the server.
- GitHub authentication as the main way to log in to the website.
- The ability to choose a GitHub repository and deploy it.
- Validation in the frontend and the backend for user input.
- User authorization.
- BullMQ to decouple the backend from the platform.

These features form a fully functional system by itself and with more features and improvements we can increase the efficiency of the system while giving the user better experience.

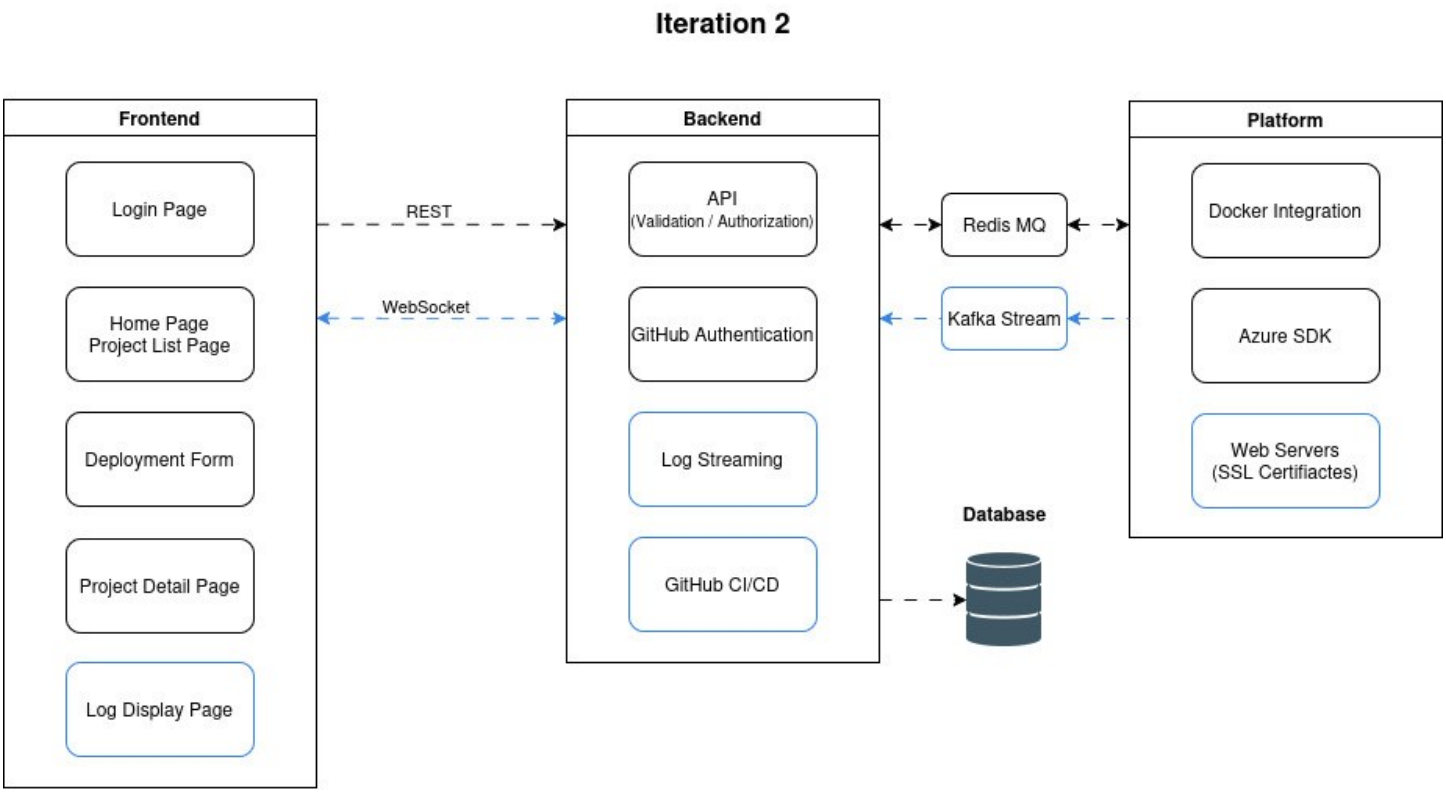
Iteration 2

Overview

In iteration two, we built upon the progress from iteration one by adding more features related to static hosting. These included:

- **Live Build Log Streaming** – Providing real-time visibility into build processes to enhance debugging and monitoring.
- **GitHub CI/CD Integration** – Automating deployments directly from GitHub repositories.
- **HTTPS Support** – Implementing secure HTTPS connections for both the platform and hosted websites.
- **Frontend Enhancements** – Improving the user dashboard with better UI/UX and new functionalities.
- **Code Quality Improvements** – Refining and optimizing the codebase to ensure maintainability and scalability.
- **Manual Upload**- Provide a method for users to deploy their websites by uploading the static files directly without GitHub.

These improvements strengthened our static hosting capabilities and set a robust foundation for the dynamic hosting phase.



Next we discuss the implementation details of the major features in this iteration: Log Streaming, GitHub CI/CD, ZIP File Upload, HTTPS/SSL Certificates.

Live Build Log Streaming

To implement log streaming in real-time, given our event-driven architecture, we needed a way to stream events continuously from the Platform to the Backend and then stream it from the Backend to the Frontend. We also need to retain the logs made and be able to retrieve them on demand, not only once.

To stream a high volume of events (logs) at a high velocity between the Backend and the Platform, we needed an event broker that can handle the rate and latency needed. Redis and message queues we had were not suitable for this use case, and message queues in general only retain messages for one-time consumption. We needed a Pub/Sub architecture that can stream at high volume, store the logs, and stream multiple times on demand. We decided to go with **Kafka**, as this is the use case it specializes in.

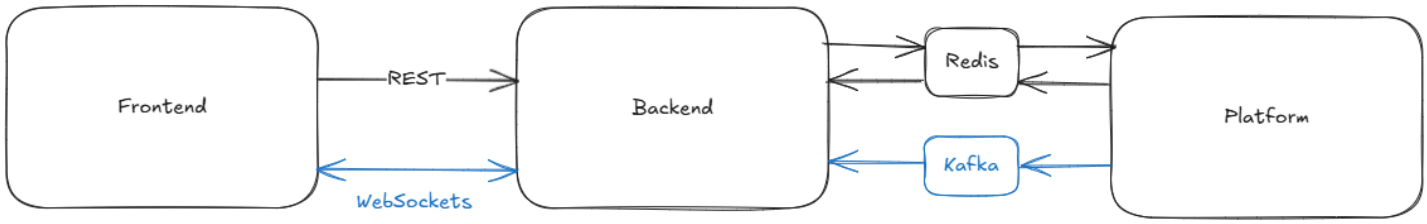
To stream logs from the Backend to the Frontend, we need a Server-to-Client communication channel, so that when the Backend receives a log, it can immediately emit it to the Frontend. REST APIs do not support two-way communication. **WebSockets** was the viable solution for a two-way communication, with which the Backend can freely emit to the Frontend.

Architecture

Kafka was added in between the Backend (the consumer) and the Platform (the producer). While building a user application, the platform would stream all logs emitted to Kafka in real-time, and the Backend would listen and consume all logs emitted to Kafka.

WebSockets communication was added between the Frontend and the Backend. A frontend page was made for each build that would create a WebSocket channel to listen to the logs for that build and display them.

When the Backend consumes logs from Kafka, it streams these logs to the Frontend to the correct WebSocket channel at real-time to be displayed to the user.



Implementation

On the initialization of the Backend, a Kafka consumer is created that listens on all active Kafka event topics.

On creating a new project or starting a new build, the respective Backend API is invoked. It creates a new project or build as usual, adds a build job to the message queue, but then it also creates a Kafka topic for this new build. This Kafka topic will receive and store all the logs emitted from this build.

The new topic's name is calculated using the project ID, build number, and the user ID. This calculation is done at all stages of the workflow to correctly redirect logs between services.

Upon creating the topic, the Kafka consumer is refreshed to register the new topic. This is due to a technical limitation in the used library that prevents the pre-running consumer from registering the newly created topic. With that, the Backend API Handler is done and returns success to the Frontend.

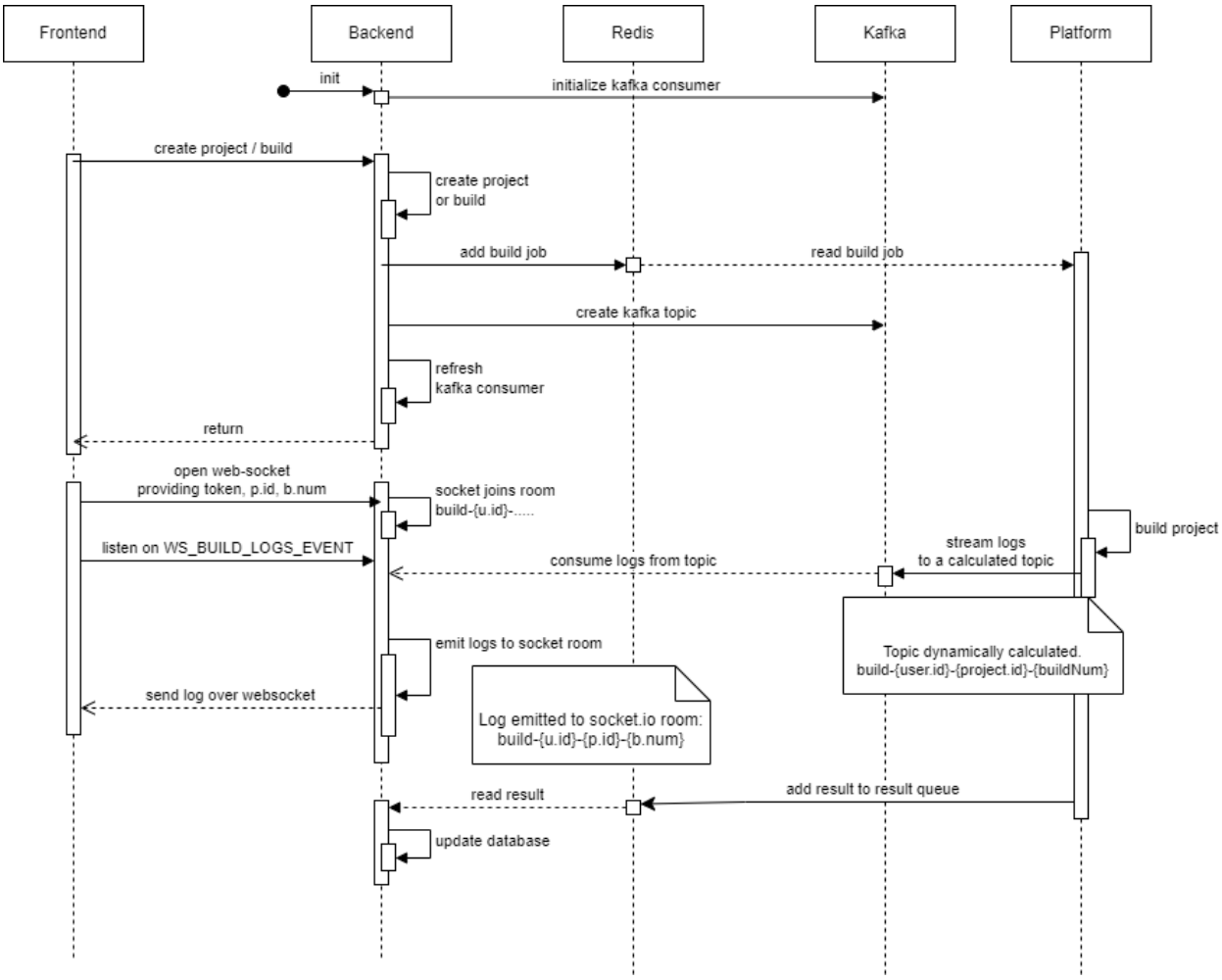
On success at the Frontend, the user is redirected to another page in which a WebSocket channel is opened with the Backend. The Frontend opens the WebSocket and provides the user JWT token, the project ID and the build number to the Backend. The user token is used by the Backend for authentication and extracting the user ID.

The backend then puts the newly connected user to a “room” named with the calculated topic name. Rooms in Socket.IO (the WebSockets library used) are used to group similar connections together and broadcast to the room.

Now on to the other side, the Platform receives the build job from the message queue and starts building. It then streams all build logs produced to the calculated Kafka topic at real-time.

The consumer in the Backend receives the emitted logs from Kafka and then broadcasts them to the WebSocket room with the same topic name. Frontend clients connected with the room receive the logs and display them.

Sequence diagram:



Design Choices

1. WebSocket Connection Authorization

We created a middleware for input validation, for token validation, but we did not create a middleware that checks if the client owns the build he is trying to see the logs for. Instead, we put the user ID into the Kafka topic’s name calculation. Since the user ID is extracted from the JWT token, this provides a secure mechanism for access control and authorization, while saving the cost of querying the database and doing the ownership check manually.

2. Single consumer in the Backend instead of a consumer for each open channel

We decided to go with a single generic consumer that will consume the events emitted for all ongoing builds and redirect each one appropriately to the WebSocket room, instead of creating a consumer for each newly-created build. This approach simplified the process and there were no drawbacks observed. A few problems that came up

with this decision: pre-running consumer not seeing the newly created topics (solved via refreshing), and more in the next points.

3. Resetting Consumer header to the start of the topic on new sockets

An issue with the single consumer was that if a WebSocket joined late, it will have missed all old logs emitted and only receive new logs. We need the new socket to receive all logs of a topic sequentially from the beginning, so on new sockets, we reset that specific topic pointer to the beginning, which will result in re-transmitting all the topic's logs to the sockets. This also allows for retrieving the build logs any time long after the build is finished.

4. Frontend keeps track of latest log offset received

With the topic pointer reset mechanism in place, a side-effect was that if an old WebSocket was connected and a new socket connects, the backend will re-transmit all logs again to all sockets of the room, resulting in the old socket receiving and displaying duplicate data. To solve this, each Frontend client keeps track of the offset of the last log received and only displays logs with higher offsets.

Screenshot

GitHub CI/CD

GitHub CI/CD functionality aims at providing real-time updates to the users deployed website to help the user in deploying his website after updating the project base repository.

Our solution integrates with the user's development pipeline by offering CI/CD functionality to the deployed project, i.e. building the repository and deploying it.

Implementing such feature necessitates a way to access the user repositories and listen to changes in real time, a form of communication between our service and GitHub internal API.

Possible solutions are:

- **Polling:** Which relies on repeatedly sending requests to the repository endpoint and comparing response results with the current deployed version to determine if there is updated information.
- **Webhooks:** Which is a lightweight, event-driven communication that automatically sends data between applications via HTTP. Triggered by specific events, webhooks automate communication between application programming interfaces (APIs) and can be used to activate workflows.

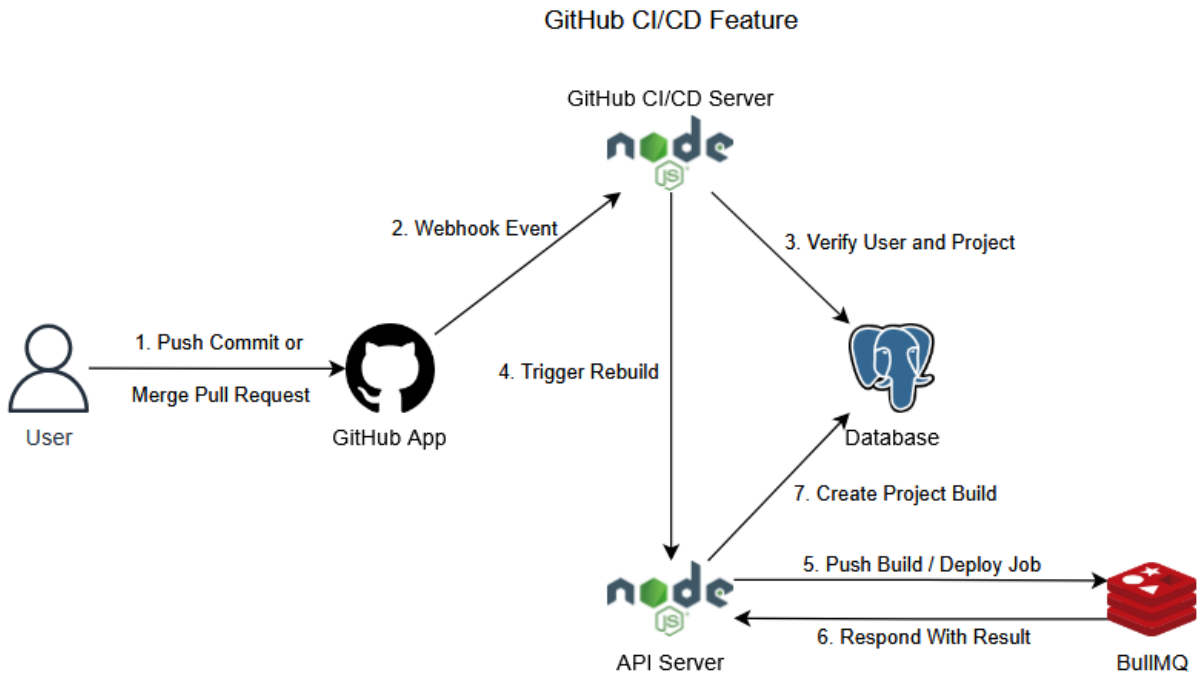
From these descriptions we can conclude the following:

- **Polling** is a bad option since it exhausts our rate limits to the GitHub API and our server resources while sending unnecessary requests and processing their responses.
- **Webhooks** are a great solution that aligns with our event driven architecture and don't use unnecessary resources in the process.

By choosing the **Webhooks** route we now have come across a new hurdle. GitHub webhook events are sent per repository, and with the increase in the number of users, handling new events for each and every repository becomes a demanding process. With this issue comes a design challenge, we need to create a new service that integrates with the GitHub API ecosystem while being fast enough to respond the Webhook events in real-time while also not bottlenecking the main server and API.

Also, we must determine the best way to receive the webhook events, whether by creating a new webhook URL for every new repository or finding a better way that involves a single handler for all the repositories available in our services.

Architecture



Implementation

Github CI/CD server

Solving the newly presented design challenge results in creating a new service, the GitHub CI/CD server.

GitHub service is a service that runs in parallel with the main server API with a sole purpose of responding to webhook events in the least amount of time possible by being decoupled from the main system via the BullMQ and asynchronous requests to the API.

It listens to the events received from GitHub Webhooks and responds with a specific action based on the event type.

Github App

GitHub Apps are tools that extend GitHub's functionality. GitHub Apps can do things on GitHub like open issues, comment on pull requests, and manage projects. They can also do things outside of GitHub based on events that happen on GitHub.

It is the successor of the GitHub OAuth app which was used in iteration 1 for GitHub authentication. OAuth apps can only act on behalf of a user while GitHub Apps can either act on behalf of a user or independently of a user. It also offers a great advantage over the GitHub OAuth App, which is, the centralized GitHub Webhook.

Find more details about GitHub OAuth and GitHub Apps in the "Migrating from GitHub OAuth to GitHub App" section.

Centralized GitHub Webhook

This is a webhook provided by GitHub to the GitHub app owner which serves as a centralized way to get events related to the repositories that users have installed the GitHub app on. This provides us an easy way to get all the webhook events from a single provider and process them one after the other. We can also choose which specific events we want to listen to for updates.

It also offers enhanced security via the Webhook secret, which hashes the content of the payload with a secret key that we provide, and we can check that the message is authentic at our Github CI/CD server.

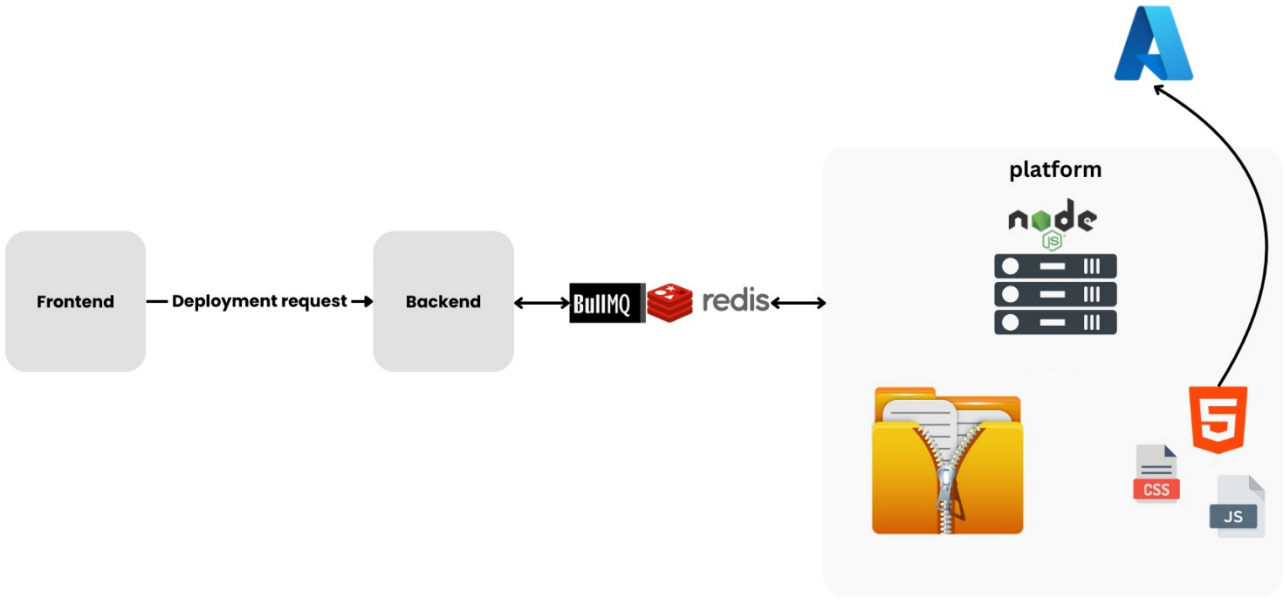
ZIP File Upload

Overview:

Zip file upload is a feature that enables the users to upload their websites for deployment by uploading the static files directly from their computers by uploading a zip file containing all the static files (HTML, CSS, JS).

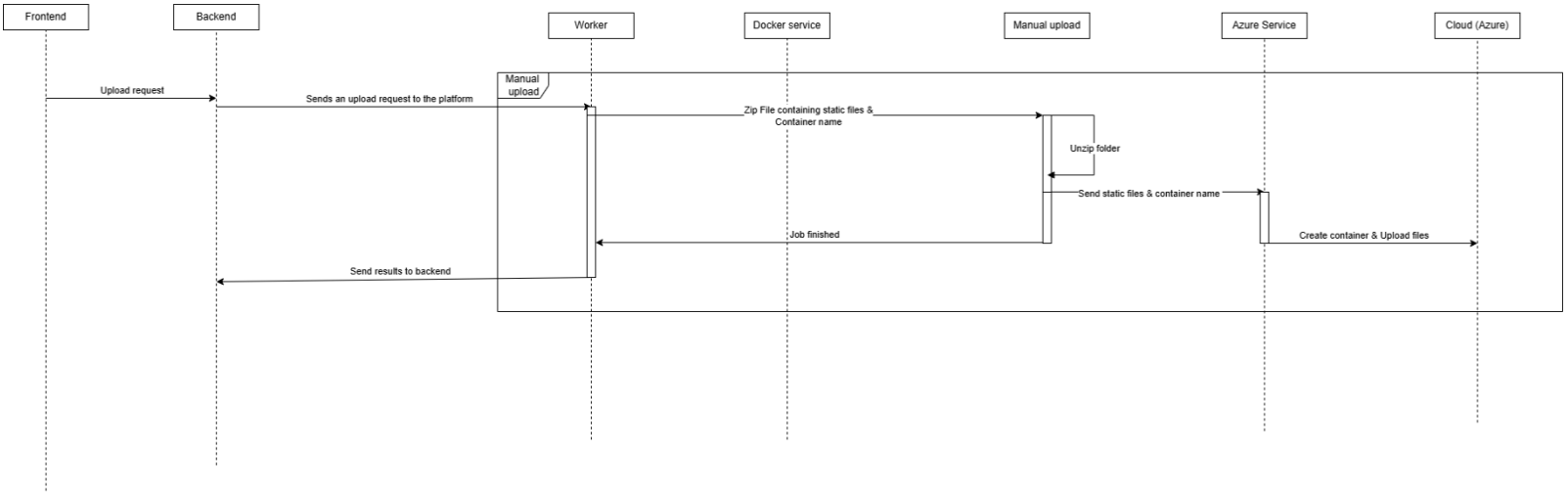
No building process will take place on the platform as the user is responsible for building the application before uploading it of building is required (React, Angular, Next...), and only static files resulting from the building of these apps are uploaded by the user to be deployed.

Architecture:



Workflow:

1. The user should be logged in by account to use the service.
2. The user should have the static files of his website prepared whether by building it (REACT, NEXT...) or having them originally in HTML, CSS, JS form, the user should create a zip file containing all these static files to upload it.
3. The user uploads that zip file from the front-end of our software.
4. Zip file is sent to the backend and then the backend forwards it to the platform for the deployment process.
5. Platform creates an azure container and uploads these files to it.
6. Results are sent back to the backend to be stored in the database and acknowledges the user of it.



The above figure shows the sequence diagram.

Implementation:

1. The zip file is sent from frontend to backend using normal RESTful API request.
2. Backend uses **multer** library to read the file form the request, updates database, assigns a container name to that specific project.
3. Then, it's forwarded to the platform using Redis message queue, The message contains the zip file to be uploaded, the project number and the container name (Azure container).
4. The zip file is then unzipped in the platform machine.
5. The unzipped files are then uploaded to the azure container created by azure service on the platform.
6. Results are then sent back to the backend, stored in the db and acknowledge the user of it.
7. Results usually contain the status of the deployment (SUCCES/ FAILURE) and the project number.

Challenges:

Problem:

The **multer** lib receives the file from the RESTful request as a buffer.

When sending this buffer directly to the platform, some sort of data loss or inability to properly transfer it happens on the message queue.

Solution: The solution was to convert this buffer to a string of base64 and send it over the message queue to the platform, once received on the platform it's then converted back to buffer format.

This buffer format is then converted to a readable stream to be unzipped properly by the unzipper library used in the platform.

In the upcoming iterations, it's being studied to change the method of sending the files from backend to the platform as sending files through the message queues is not a very effective way as these files are usually large in size and it's inappropriate to send very large size files through message queues.

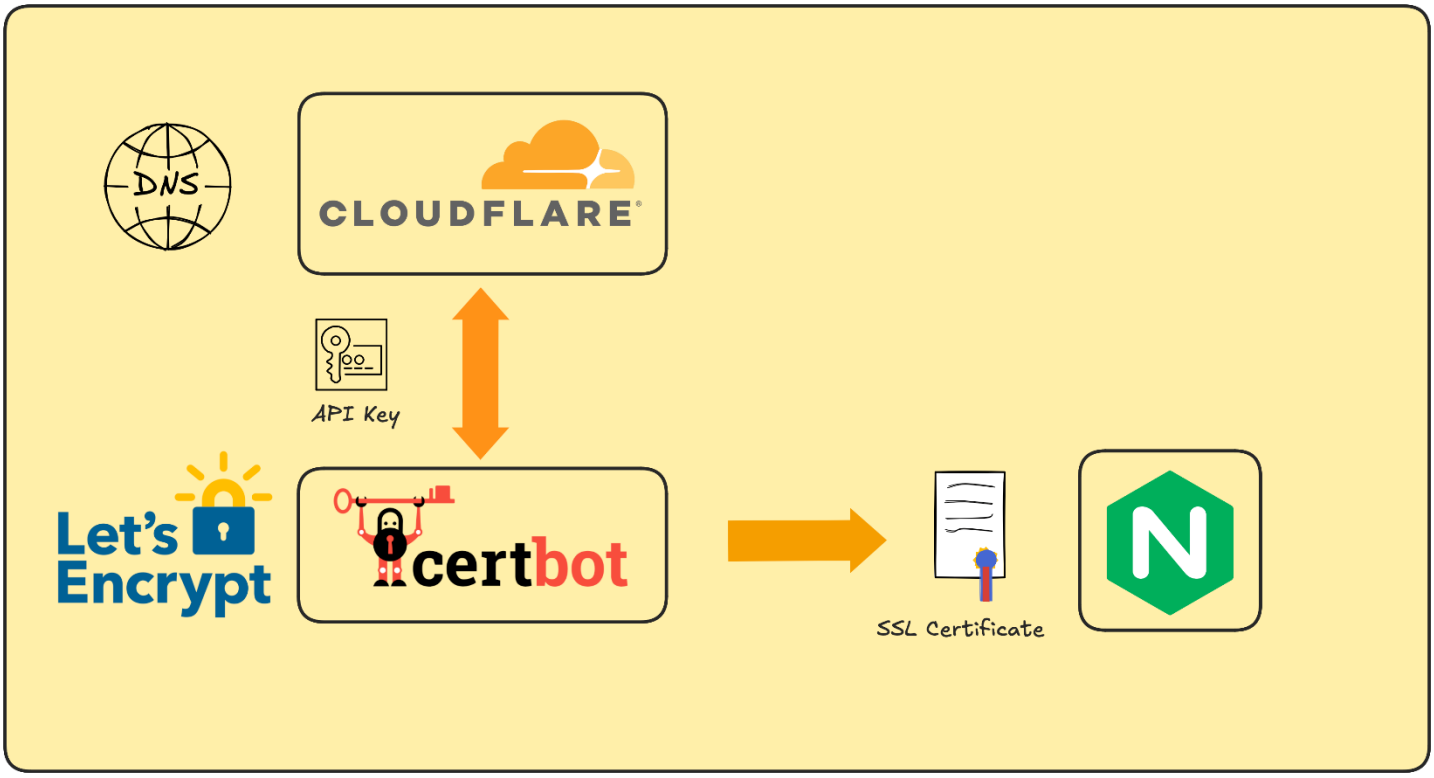
The new approach is to use presigned URL.

Presigned url in a very brief manner is a secure way to provide temporary access to a specific resource (e.g., files or objects stored in a cloud storage service like AWS S3 or Azure Blob Storage) without exposing sensitive credentials. It allows you to grant limited-time access to upload or download a file.

HTTPS

To implement HTTPS protocol, you need to provide an SSL certificate for users to determine the authenticity of your website and use the key provided to encrypt data transmitted. An SSL certificate is created and signed by a Certificate Authority (CA), which is a trusted organization by web browsers for signing SSL certificates and is valid for limited duration. Web browsers are generally distributed with a list of signing certificates of major certificate authorities so that they can verify certificates signed by them. After acquiring a signed SSL certificate, it is then integrated into the web server to activate HTTPS protocol on all communication. Another point to take in mind, which is extremely important, is certificate renewal. Having an automated process for renewing all SSL certificates for your website is a must.

Architecture



Let's Encrypt provides a convenient service called **Certbot**. **Certbot** is a software tool for automatically using Let's Encrypt certificates on manually-administrated websites to enable HTTPS. It provides also automatic certificate renewal. Given some parameters, Certbot generates SSL certificates to be used by the web server NGINX.

Implementation

When you get a certificate from Let's Encrypt, their servers validate that you control the domain names in that certificate using "*challenges*". The two most popular challenges are HTTP-01 challenge and DNS-01 challenge. Each has a specific use case which in our project we use both.

HTTP-01 challenge

How It Works:

1. Let's Encrypt provides a unique token to be placed in a specific HTTP endpoint (/.well-known/acme-challenge/) on your server.
2. Your server must host this token on the specified endpoint over HTTP.
3. Let's Encrypt then sends an HTTP request to the domain (e.g., http://example.com/.well-known/acme-challenge/<token>) to verify:

- The server responds with the correct token.
- The server is accessible over HTTP.

Use Case:

- Suitable when the domain resolves to a web server that you can control directly.
- Best for automating certificate issuance in environments where you can easily configure HTTP endpoints.

Challenges:

- Requires the web server to be running and publicly accessible.
- Not ideal for **wildcard certificates** (e.g., *.example.com), as it only validates the root domain or subdomains explicitly listed.

DNS-01 challenge

How It Works:

1. Let's Encrypt provides a unique token and requires it to be added as a TXT record in the domain's DNS configuration.
2. You create a DNS TXT record for the domain (e.g., _acme-challenge.example.com) containing the provided token.
3. Let's Encrypt queries the DNS servers for the domain to verify the presence of the TXT record and the correct token value.

Use Case:

- Ideal for domains where direct control of the HTTP server is not available.
- Required for issuing **wildcard certificates** (e.g., *.example.com) because it validates the entire domain zone.

Challenges:

- Requires access to the DNS management system (e.g., API or manual interface).
- Propagation delays in DNS can slow down the challenge verification.

In our project, we have a fixed domain name for our frontend and backend. Those are eligible for the HTTP-01 challenge. However, users hosted websites are given a random subdomain, so we can't determine beforehand the full domain and that makes it eligible for DNS-01 challenge.

Iteration Outcomes

This iteration marks the end of the static hosting milestone.

With it, we have implemented:

- CI/CD feature to update the user repositories on new updates using GitHub App.
- Live log streaming to display the live logs in real-time for the user using Kafka.
- Manual file upload as an alternative deployment option other than GitHub.
- HTTPS service to increase the security of our website traffic using Let's Encrypt.

These features aimed to combat the shortcomings of the first iteration and improve system performance by decoupling the different aspects of the system while adding new features.

Iteration 3 and Beyond

At the start of the second term, we identified several challenges and optimization opportunities. Our focus remained on enhancing static hosting features and refining the overall system.

Key Achievements

1. **Frontend Overhaul**
    - Our initial frontend lacked color and responsiveness. We addressed this by introducing a cohesive color scheme, ensuring full responsiveness, and improving the user experience across all pages.
2. **Backend Idempotency**
    - We implemented idempotency in the backend to enhance request/response reliability (more details to follow).
3. **Optimized File Uploads**
    - To save bandwidth and reduce latency, we introduced pre-signed URLs for manual uploads.
4. **Enhanced Logging System**
    - A custom logging solution was developed, significantly boosting developer productivity by streamlining debugging, monitoring, and reporting.
5. **Mono-Repository Transition**
    - We restructured the project into a mono-repo for better code management and collaboration.
6. **Automated Testing**
    - GitHub Action workflows were added to run automated tests on every pull request, ensuring consistent code quality.
7. **Resource Cleanup**
    - We enforced proper cleanup when deleting projects, eliminating resource leaks and ensuring complete removal of associated data.
8. **Stricter API Validation**
    - Robust response validation was implemented in the backend, and the frontend was refactored to rely on these validated responses—strengthening security and synchronization.

Pre-signed URLs

Overview:

Regarding manual uploading of projects, the first approach taken was to send the zip file of the project from user to backend and then backend sends it to the platform via message queue.

The problem is that sending large files via a message queue isn't a practical method.

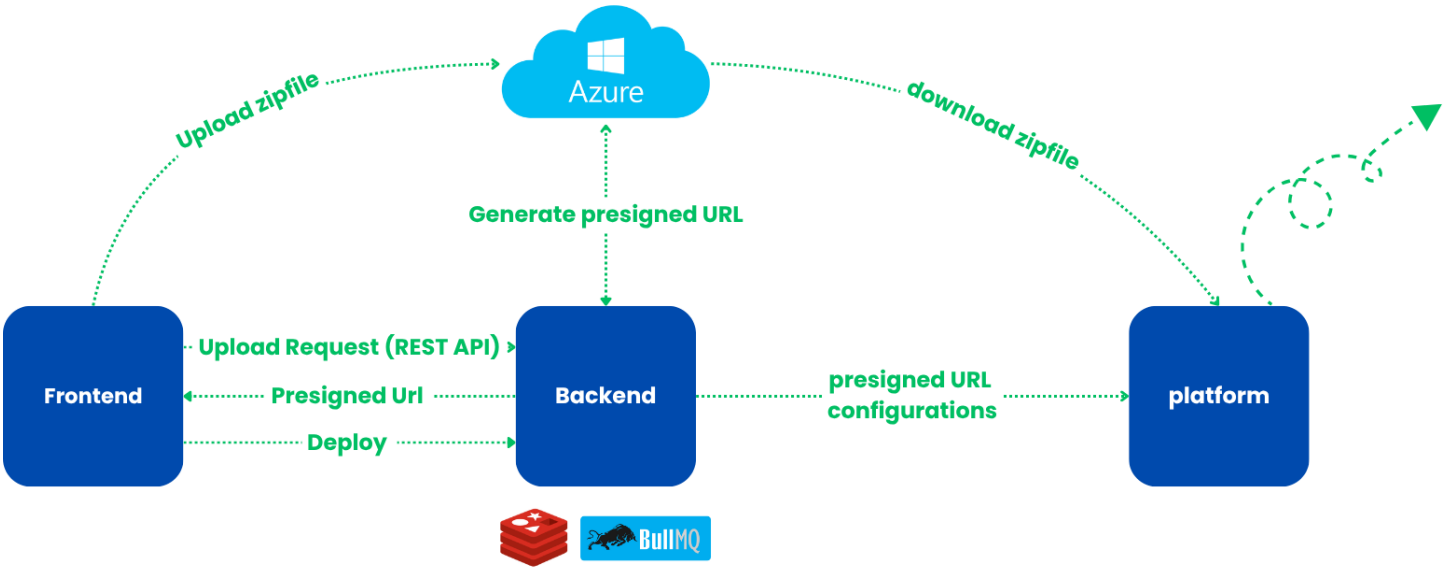
The solution is to use Microsoft azure's presigned URLs. A presigned URL is a secure, time-limited URL that gives temporary access to a private resource — typically in cloud storage.

Previous workflow:

When using the message queue to send zip file, the following workflow had taken place:

- User requests a manual upload and sends the zip file to the backend.
- The backend sends the zip file to the platform via the message queue.
- Platform unzips the zip file and uploads the static files to the Azure storage container, making it deployed to the cloud.
- Platform sends the result back to the backend via message queue.

Current workflow:



1. User requests a zip file upload. The request is sent from frontend to backend through REST API.
2. Backend requests a presigned URL from azure.
3. Azure creates a storage container and assigns it to the presigned URL, the container name/Presigned URL of azure sent to the backend as a response.
4. The backend sends the presigned URL to the frontend. But before doing so, The backend maps the userId to this presigned url using redis, so in when the frontend sends to the backend triggering that the zip file upload was finished on that specific URL, backend can check if that URL truly belongs to that user to avoid security risk of someone sending malicious files on a self-made URL.

5. Frontend starts uploading the zip file onto the presigned URL, once done, it sends it to the backend triggering a finished upload of zip file to the azure container via the URL.
6. The backend then validates that the URL sent from the frontend with the user, if valid and belongs to that user, it then deletes it from the Redis server and sends the presigned URL to the platform.
7. Platform then downloads the zip file from the presigned URL, unzip it, extarct the files and upload them to the cloud on the same azure storage container made for that specific presigned URL.
8. Once the process is finished, the Platform sends back to the backend the status of this process.
9. Backend updates the database and informs the user at the frontend that the upload is done sending the url to visit the website from.

NOTE:

- When the zip file is downloaded from the presigned URL, it’s deleted from the storage container.

Saving logs in Kafka instead of database

Initially, all build logs persisted in the database after each deployment. While this made logs easily accessible at any time, it introduced several drawbacks:

- Increased database size and I/O load
- Redundant storage of logs already present in Kafka
- Performance bottlenecks during frequent writes

We refactored the system to stop storing logs in the database. Instead, logs are now:

- **Fetched on-demand** from Kafka using the topic associated with each build
- Temporarily processed in memory
- Returned directly in the response from the endpoint: GET /admin/projects/:id/builds

The current optimization based on 2 functions:

- fetchLogsFromKafka(topicName: string)
  Purpose: Fetches all logs related to a deployment directly from Kafka topic.
  Mechanism:
  1. Creates a Kafka consumer with a **random group ID**.
  2. Checks if the topic exists: If not → returns `"Expired Topic".`
  3. Uses `getTopicMessageCount` to see how many messages are in the topic: If zero → `"Empty or Expired topic"`.
  4. Subscribes to the topic and **collects all messages**.
  5. Combines all logs into a single string and returns it.

- getTopicMessageCount(topicName: string)
  Purpose: Calculate the total number of messages in a Kafka topic.
  Mechanism:
  1. Connects to Kafka as an admin.
  2. Fetches partition offsets (`high`, `low`) for the topic.
  3. Calculates the number of messages in each partition: `high - low`.
  4. Sums all partitions to return the total message count.
  5. Closes the Kafka connection.

Backend Idempotency

Idempotency refers to the property of an operation whereby performing it multiple times has the same effect as performing it once. In HTTP, methods such as `GET`, `PUT`, and `DELETE` are defined as idempotent by design, whereas `POST` and `PATCH` are not.

In a cloud web hosting system, a lack of idempotency in certain operations can lead to unintended consequences. For example, if a user experiences lag and clicks the **Deploy** button multiple times, the backend might process these as separate requests, resulting in the creation of multiple identical projects, which is not the desired outcome.

To mitigate such issues, the backend has been designed with the following safeguards:

- **Database Constraints:**
  A unique constraint has been added to the database schema on the combination of project name and user ID. This ensures that even if multiple requests with the same project name are sent, only one will succeed. Any subsequent request will result in an error indicating that the project already exists.

- **Manual Upload Validations:**
  During manual uploads, users are prompted to specify a project name. The system performs validation checks to ensure the name is unique and valid before proceeding with the upload process.

- **Rebuild Request Control:**
  To prevent concurrent rebuilds of the same project, the system checks the status of the latest build before initiating a new one. This avoids deploying multiple builds simultaneously and ensures build consistency.

Dynamic Hosting Milestone

Overview of the main features

ZIP File Upload

High-level description:

The manual upload feature enables users to deploy their dynamic projects by uploading a zip file of their application. This feature is designed for users who are not using git, or are making special changes.

Importance:

This feature is crucial for providing flexibility in deployment workflows. By allowing users to directly upload their pre-built applications, it accommodates:

- **Accelerated Deployment:** For applications already built and ready for production, manual upload offers significantly faster deployment times.
- **Local Development Flexibility:** Ideal for developers who maintain their application code locally outside of platforms like GitHub.

Use case example:

A developer uploads a ZIP file containing a backend application. The platform will deploy it and return the link.

GitHub CI/CD & Rebuilding Project

High-level description:

This feature extends from the previously implemented GitHub CI/CD system to support dynamic projects. It enables users to automatically update their deployed dynamic applications when a new commit is pushed, or a pull request is merged into the deployed branch. The original implementation is described in the "GitHub CI/CD" section under the static hosting milestone.

Importance:

With the introduction of dynamic upload functionality and broader system capabilities, it became necessary to extend and decouple the CI/CD system from the rest of the backend services. This decoupling was crucial to prevent the service responsible for handling GitHub webhook events from making direct database calls, thereby reducing latency, avoiding tight coupling, and improving the system's scalability and reliability when responding to CI/CD triggers.

Use case example:

A developer deploys a dynamic project from a GitHub repository. After deployment, they merge a pull request or push a new commit to the deployed branch. The system automatically detects these changes and triggers a rebuild. When the developer visits the deployed project's URL, they observe that the site has been updated to reflect the latest changes.

Health Check, Auto-scaling & Project Monitoring

High-level description:

Our system provides real-time visibility into your project's performance while ensuring reliability through automated maintenance. It continuously tracks your application's health status, scales resources based on demand, and displays your project's CPU, memory, disk, and network metrics through an intuitive dashboard.

Importance:

This comprehensive solution gives you complete control over your deployment's stability and efficiency. The health checks prevent silent failures, auto-scaling maintains optimal performance during traffic fluctuations, and real-time monitoring empowers you with immediate insights into your project's resource consumption.

Use case example:

As your web application experiences a surge in visitors:

1. The dashboard shows real-time CPU/memory spikes
2. Auto-scaling automatically provisions additional instances
3. Health checks continuously verify each instance's responsiveness
4. You can immediately see which resources are being strained and adjust configurations accordingly

Build Rollback

High-level description:

This feature provides users with the ability to immediately rollback to the most recent successful build. It acts as a failure recovery mechanism in the event of a faulty deployment.

Importance:

When a deployment completes successfully at the build stage but results in runtime errors during execution, the user will have the ability to restore the last working version of his project, minimizing downtime while the issue is being investigated or resolved.

Use case example:

A developer pushes new code and deploys it. Although the build step completes without errors, the deployment encounters runtime issues. To restore functionality, the developer rolls back to the last successful build, allowing the application to remain available while they debug and fix the faulty changes.

Set active build version

High-level description:

This feature extends the functionality of the “build rollback” system by allowing users to select any previously successful build, older or newer, and set it as the current active build for the deployed project.

Importance:

Rather than being limited to rolling back only to the most recent build, users now have the flexibility to switch to any prior successful build. This enables use cases such as restoring an older stable version, testing various buildings and choosing one at the end, or skipping problematic builds without needing to redeploy code.

Use case example:

A team wants to compare the performance of two builds, one with a new feature enabled and one without. Using the "Choose Active Build Version" feature, they can toggle between the two builds in production to gather feedback and performance data before making a final decision.

Research Phase

Dynamic hosting refers to the practice of serving applications where content is generated on the server in real time, often based on user input, database queries, or other backend logic. Unlike static hosting, which delivers fixed, pre-rendered content, dynamic hosting supports interactive and personalized experiences, making it suitable for web apps, dashboards, content management systems, and more.

Dynamic hosting requires a backend runtime environment capable of executing server-side code (e.g., Node.js, Python, PHP), and supporting components like databases, sessions, authentication, and scaling. These demands introduce complexity around **resource isolation**, **security**, and **orchestration**, especially in **multi-tenant** environments where multiple users deploy and run applications on shared infrastructure

Our research focused on building a flexible and secure dynamic hosting platform that supports multi-tenant workloads. We compared various computer environments containers, virtual machines (VMs), and microVMs to evaluate the tradeoffs between performance, isolation, and operational complexity.

We excluded VMs and compared between containers and microVMs, through this exploration, we identified **microVMs** as the most balanced solution for securely hosting dynamic applications while maintaining efficiency and scalability. We compared between different microVMs solutions like kata, nabla and firecracker. We also researched how to use firecracker with containerd.

We settled on using kata containers in the end.

Proof of Concepts Phase

Kata containers

Kata Containers is a container runtime that combines the speed and flexibility of traditional containers with the strong isolation and security of virtual machines. Unlike standard containers that share the host's kernel, **Kata Containers run each container inside its own lightweight virtual machine using a dedicated kernel.**

To test if Kata is working properly, we run a container using Kata as the runtime and execute uname -a inside it. If the kernel version shown is different from the host's, it confirms that the container is running in its own VM with its own kernel — meaning Kata is functioning correctly and providing the intended isolation.

```
somaya@somaya:~$ uname -a
Linux somaya 6.8.0-58-generic #60~22.04.1-Ubuntu SMP PREEMPT_DYNAMIC Fri Mar 28 16:09:21 UTC 2 x86_64 x86_64 x86_64 GNU/Linux
somaya@somaya:~$ sudo ctr run --rm --runtime io.containerd.kata.v2 docker.io/library/alpine:latest test-kata uname -a
[sudo] password for somaya:
Linux localhost 6.12.13 #1 SMP Mon Feb 17 16:46:21 UTC 2025 x86_64 Linux
somaya@somaya:~$
```

To evaluate the feasibility of Kata Containers, we explored and tested it using multiple approaches.

Kata containers with rke2 and k3s cluster using kata deploy

Although there are well-documented steps for deploying Kata Containers and integrating them with RKE2 and K3s, the process consistently encountered failures. Even when Kata Containers appeared to install successfully, containerd often failed to recognize the runtime properly, and as a result, the Kubernetes nodes were not automatically labeled as Kata-ready.

Kata containers integeration with containerd using kata manager

To validate the integration, a **vanilla Kubernetes cluster** was provisioned using **Vagrant** and **VirtualBox**, and Kata Containers were installed using the kata-manager script.

1. Environment Preparation

- Set up VirtualBox-based virtual machines with nested virtualization enabled to support KVM.
- Ensured hardware virtualization support inside the VMs to enable lightweight virtual machines.

2. Installing and configuring containerd

- Deployed containerd as the container runtime.
- Configured system modules and network settings to ensure compatibility with Kubernetes.
- Generated and customized the default containerd configuration.

3. Deploying Kata Containers

- Installed Kata Containers using the official automation script (kata-manager).
- Integrated Kata as an additional runtime within the containerd configuration.
- Linked Kata binaries to system paths to ensure containerd could access them.

4. Verifying Kata Runtime

- Ran test containers using the Kata runtime to validate VM-level isolation.
- Verified the use of a separate kernel inside Kata-based containers as proof of virtualization.

5. Kubernetes Integration

- Installed and configured Kubernetes components (kubeadm, kubelet, kubectl) on both master and worker nodes.

- Disabled swap and tuned containerd settings (e.g., enabling SystemdCgroup) to meet Kubernetes requirements.
- Initialized the Kubernetes control plane and applied a network plugin (Calico) for pod networking.

6. Cluster Setup and Runtime Validation

- Successfully added worker nodes to the Kubernetes cluster.
- Validated cluster health and component functionality using kubectl.
- Confirmed the Kata runtime is functioning inside Kubernetes-managed workloads.

```
vagrant@master:~$ kubectl get nodes
NAME     STATUS   ROLES           AGE   VERSION
master   Ready    control-plane   52d   v1.29.15
node1    Ready    <none>          52d   v1.29.15
```

```
vagrant@master:~$ kubectl get pods
NAME          READY   STATUS      RESTARTS   AGE
nginx-kata    1/1     Running     0          52d
uname-pod     0/1     Completed   0          52d
```

```
apiVersion: v1
kind: Pod
metadata:
  name: uname-pod
spec:
  containers:
  - name: uname-container
    image: busybox
    command: ["sh", "-c", "uname -a; sleep 1"]
  restartPolicy: Never
```

[For detailed steps and observations, look here.]

Although Kata Containers eventually ran successfully on a local Kubernetes cluster, automating the setup proved difficult. As a result, we used AKS with built-in support for Kata Containers was used instead, providing a more stable and consistent environment.

Kubernetes Dynamic Hosting API

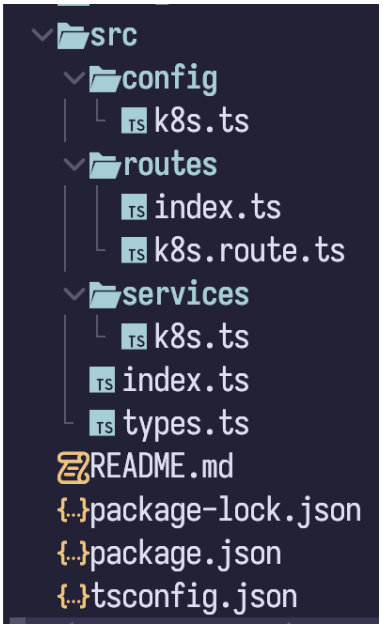
This section details the Proof of Concept (PoC) project, a Kubernetes API service designed for CRUD (Create, Read, Update, Delete) operations on Kubernetes resources.

Main Features

- **Pod Deployment**: This feature allows users to deploy new Kubernetes Pods from a specified Docker image. It includes an optional capability to immediately expose the newly created Pod via a Kubernetes Service.
- **Pod Exposure**: This feature enables the exposure of an existing Kubernetes Pod to external traffic by creating a NodePort Service.
- **Pod Deletion**: This feature provides the functionality to remove a specified Kubernetes Pod from the cluster.

Project Structure

The project follows a well-organized directory structure:



- ***package.json***: Manages project dependencies and scripts.
- ***tsconfig.json***: Configures the TypeScript compiler settings for the project.
- ***src/index.ts***: The main entry point of the application, setting up the Fastify server and registering routes.
- ***src/types.ts***: Defines custom TypeScript types used across the application, such as FastifyZodInstance.
- ***src/config/k8s.ts***: Handles the Kubernetes client configuration, loading the default kubeconfig and creating a Kubernetes API client.
- ***src/routes/index.ts***: Responsible for dynamically registering all routes defined in the routes directory.
- ***src/routes/k8s.route.ts***: Defines the API endpoints for Kubernetes operations, including deploying, exposing, and deleting pods.
- ***src/services/k8s.ts***: Contains the core logic for interacting with the Kubernetes API, encapsulating functions for creating, deleting, and exposing pods.

Technologies Used

- **Fastify**: A fast and low-overhead web framework for Node.js.
- **TypeScript**: A typed superset of JavaScript that compiles to plain JavaScript.
- **Zod**: A TypeScript-first schema declaration and validation library, used for request body validation with fastify-type-provider-zod.
- **@kubernetes/client-node**: The official Kubernetes client library for Node.js, enabling interaction with the Kubernetes API.

Core Functionality

The project provides the following key functionalities through its API:

- **Pod Deployment**:
  - **Endpoint**: POST /deploy
  - **Description**: Creates a new Kubernetes Pod from a specified Docker image. It also offers an optional feature to expose the deployed Pod via a Kubernetes Service.
  - **Parameters**:
    - image (string): The Docker image name to deploy.
    - expose (boolean, optional): If true, a service will be created to expose the pod.
    - servicePort (number, optional): The port for the Kubernetes Service.

    - containerPort (number, optional): The port that the container within the Pod is listening on.
    - nodePort (number, optional): The NodePort to expose the service on (if not provided, Kubernetes assigns one dynamically).
- **Logic**: The createPod function in KubernetesService constructs a V1Pod manifest with the given image and a unique name, then uses k8sApi.createNamespacedPod to deploy it. If expose is true, an internal POST /expose call is made to create a service.
- **Pod Exposure**:
    - **Endpoint**: POST /expose
    - **Description**: Exposes an existing Kubernetes Pod by creating a NodePort Service.
    - **Parameters**:
        - podName (string): The name of the Pod to expose.
        - servicePort (number, optional): The service port.
        - containerPort (number, optional): The container port.
        - nodePort (number, optional): The desired NodePort.
- **Logic**: The exposePodService function defines a V1Service manifest with NodePort type, linking it to the specified podName via labels. It then calls k8sApi.createNamespacedService to create the service and returns the assigned NodePort.
- **Pod Deletion**:
    - **Endpoint**: DELETE /delete-pod
    - **Description**: Deletes a specified Kubernetes Pod.
    - **Parameters**:
        - podName (string): The name of the Pod to delete.
- **Logic**: The deletePod function utilizes k8sApi.deleteNamespacedPod to remove the Pod from the Kubernetes cluster.

Error Handling

The application includes a global error handler for the Fastify instance. It specifically catches ZodError instances, returning a 400 Bad Request status with detailed validation issues. Other errors are sent directly as a 500 Internal Server Error.

Kubernetes Interaction

The project uses the @kubernetes/client-node library to interact with the Kubernetes API. It loads the default kubeconfig, which is typically used by kubectl, allowing it to connect to the configured Kubernetes cluster which runs using minikube. All Kubernetes operations are performed within the "default" namespace.

Kubernetes Cluster Setup using Kubespray

Kubespray is a tool for deploying a production ready Kubernetes cluster.

Features

It can be deployed on various cloud providers like AWS, GCE, Azure and it can also be used to deploy a local cluster. It supports highly available cluster and has a wide range of plugins that can be used while deploying the cluster, like network plugins.

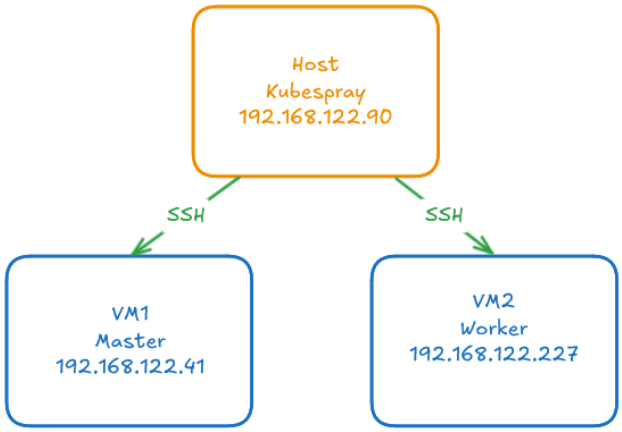
Prerequisites

Kubespray uses ansible playbook to deploy the Kubernetes cluster. Before deploying the cluster, the user should create the VMs that will host the Kubernetes cluster.

The host that runs the ansible playbook should have SSH access to all VMs that will be part of the Kubernetes cluster. Generate a key on the host machine and send it to the rest of the VMs

Deploying the cluster

We are going to deploy the following configuration:



1- Start by cloning the Kubespray code into our working directory:

```
> git clone https://github.com/kubernetes-sigs/kubespray.git
> cd kubespray
> git checkout release-2.17
```

2- Create a fresh virtual environment to install the dependencies for the Kubespray playbook:

```
> python3 -m venv venv
> source venv/bin/activate
```

3- Now we need to install the dependencies for Ansible to run the Kubespray playbook:

```
> pip install -r requirements.txt
```

4- Copy inventory/sample as inventory/mycluster:

```
> cp -rfp inventory/sample inventory/mycluster
```

5- Edit the ansible inventory file:

```
> vi inventory/mycluster/inventory.ini
```

6- Paste this example inventory.ini:

```
[all]
master ansible_host=192.168.122.41 # Control plane + etcd
node1 ansible_host=192.168.122.227 # Worker node
[kube_control_plane]
master
[etcd]
master
[kube_node]
```

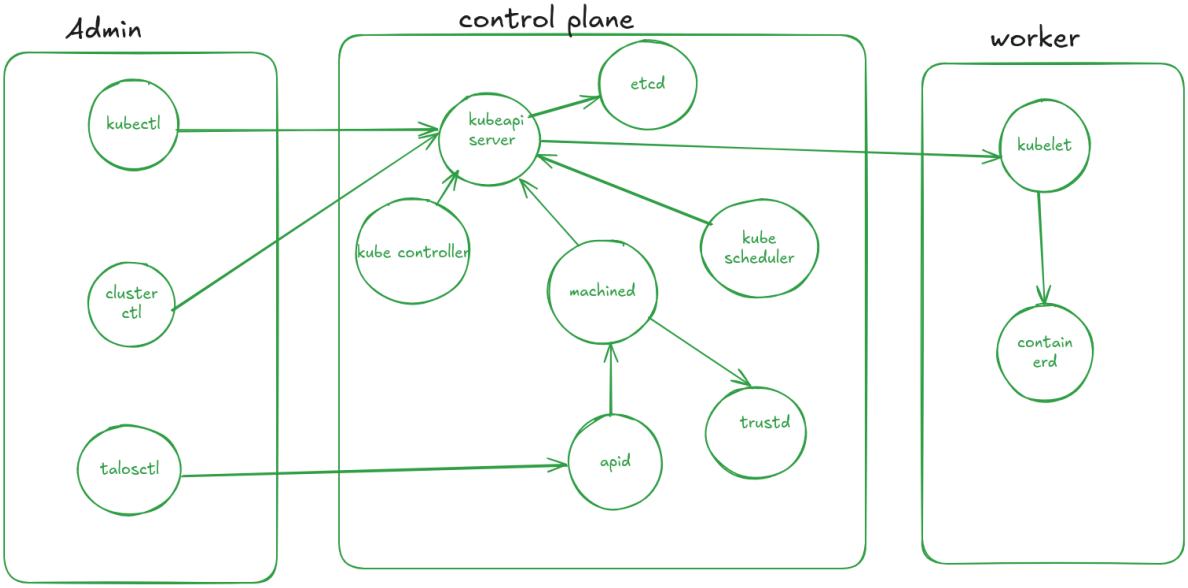
```
node1

[k8s_cluster:children]

kube_control_plane

kube_node
```

7- Run the following command to install our Kubernetes cluster. After about15 minutes we will have our Kubernetes cluster up and running.

```
> ansible-playbook -i /inventory/inventory.ini cluster.yml --user=ansible --ask-pass --
become --ask-become-pass
```

Talos OS

Talos OS is a secure, immutable, and minimal Linux distribution designed specifically for Kubernetes (K8s). Unlike traditional Linux OSes, Talos eliminates unnecessary components (e.g., SSH, shell, package managers) to reduce attack surfaces and simplify cluster management.



Architecture Components

1. Immutable & Read-Only Filesystem

- The OS runs from a squashFS (compressed, read-only filesystem).
- Configuration is applied via API calls (no manual file edits).
- Prevents drift and ensures consistency.

2. Secure Boot & Disk Encryption

- Supports UEFI Secure Boot to verify kernel integrity.
- Optional disk encryption (LUKS) for data security.

3. Minimalist Design

- No shell (/bin/sh), SSH, or interactive login.
- No package manager (all updates are atomic and OS-level).
- Only essential services for Kubernetes run as containers.

4. Declarative Configuration via API

- Managed entirely through a gRPC API (instead of SSH or manual configs).
- Uses YAML manifests for machine identity, network, and K8s settings.
- Supports machine configuration patches for updates.

5. Integration with Kubernetes

- Ships with kubeadm, kubelet, and containerd preconfigured.
- Auto-joins clusters using bootstrap tokens or discovery APIs.
- Self-upgrades via Kubernetes operators (e.g., Talos upgrade operator).

6. Component Breakdown

Component	Role
machined	Replaces init (PID 1); manages containers and services.
trustd	Handles PKI, certificate issuance, and cluster trust.
apid	Exposes the gRPC API for cluster management.
containerd	Container runtime for Kubernetes workloads.
kubelet	Kubernetes agent to run Pods.
networkd	Manages networking (CNI integration for Calico, Flannel, etc.).

Dokploy

An open source self-hosted Platform as a Service solution.

Launch Date:

- Created around April 2024 (relatively new).

Technology Stack:

- Built on **Docker** and **Docker Swarm** (similar to Coolify and CapRover).
- **License Controversy:** May not be fully open-source; discussions about its license exist (GitHub Thread).

Key Advantages Over Dokku

- **Multi-Node Support:** Scalable across multiple servers (unlike Dokku).
- **Web UI:** User-friendly graphical interface.
- **Simplified Setup:** Zero configurations needed to start.
- **Built-in Features:** No plugins required (unlike Dokku).

Deployment & Hosting

- **Lightweight:** Runs on a single VPS.
- **Pre-configured:** Ready for immediate use.
- **Excellent UI/UX:** Intuitive and easy to navigate.

Features

1. Flexible Application Deployment: Supports **Nixpacks**, **Heroku Buildpacks**, and custom **Dockerfiles**.

2. Native Docker Compose Support: Full integration for managing complex applications.

3. Traefik Configuration: Configure reverse proxy via **GUI or API**.

4. Multi-Server & Multi-Node Scaling: Deploy apps across multiple servers **without manual configuration**.

5. Advanced User Management: Granular **role-based access control (RBAC)**.

6. Database Management

- Supports **MySQL, PostgreSQL, MongoDB, MariaDB, Redis**.
- Automated **backups**.

7. API & CLI Access: Extend functionality programmatically.

8. Docker Swarm Clusters: Built-in support for **high-availability setups**.

9. Open-Source Templates: One-click deployment for popular tools.

10. Real-Time Monitoring & Alerts

- Track **CPU, memory, network usage**.
- Customizable **health alerts**.

Ideal Use Cases

- Developers needing a **simple, scalable PaaS**.
- Teams managing **multi-node Docker Swarm clusters**.
- Users who prefer **GUI over CLI** (e.g., Dokku alternatives).

Implementation Phase

The development of dynamic hosting was structured into three major iterations, each building on the progress of the previous one. This phased approach allowed us to refine functionality, optimize performance, and ensure a smooth transition to production.

Iteration 1 – Laying the Foundation

The first iteration was about establishing the core infrastructure for dynamic hosting. Key accomplishments included:

- **Initial Process Design:** Developed the fundamental workflow for dynamic hosting, including request handling, deployment, and routing.
- **Cross-Component Updates:** Modified the backend, frontend, and platform to support dynamic hosting, ensuring seamless integration.
- **First Functional Version:** Delivered a basic but fully operational dynamic hosting system, enabling users to deploy and access dynamic content.

This phase set the groundwork but was intentionally minimal to allow for iterative improvements.

Iteration 2 – Refinement and Production Readiness

The second iteration focused on stabilization, optimization, and preparing for production. Major improvements included:

- **Process Optimization:** Streamlined deployment workflows, reducing latency and improving reliability.
- **New Features:** Expanded functionality based on early feedback, such as enhanced configuration options and better error handling.
- **Admin Dashboard & Landing Page:** Began parallel development of the admin interface for managing deployments and a public-facing landing page.
- **Production Deployment:**
    - Set up a **Kubernetes cluster** for scalable, resilient hosting.
    - Implemented **production mode**, ensuring secure and efficient operation in a live environment.
    - Successfully launched the system in production, validating stability under real-world conditions.

This phase marked the transition from a proof-of-concept to a fully deployable solution.

Iteration 3 – Polishing and Final Touches

The final iteration prioritized completing the feature set, optimizing performance, and enhancing usability:

- **Feature Completion:** Added critical last-mile features, including advanced logging, user access controls, and deployment rollback.
- **Performance Tuning:** Optimized resource usage and response times, particularly for high-traffic scenarios.
- **Bug Fixes & Stability:** Resolved lingering issues to ensure smooth operation in production.

With these final enhancements, the dynamic hosting system reached maturity, offering a robust, scalable, and user-friendly solution.

Iteration 1 – Dynamic Hosting Process

Overview:

Dynamic hosting lets users upload backend applications over the internet to access it via API endpoints.

It is the hosting of backend apps that run all the time, handle requests. It requires that the application to be fully functioning 24/7 to provide the seamless experience to the user.

Here we are implementing the dynamic hosting of users' web apps. We are mainly using container approach to host dynamic apps by containerizing the backend application and deploy it on cloud using Kubernetes container orchestration.

We're using Azure Kubernetes Service (AKS) as the provider for the Kubernetes cluster used to deploy the apps.

We currently support the NodeJS runtime environment for dynamic application, more runtime environments such as (PHP, python, ...) are to be supported in the feature.

The hosted application is then exposed to the internet via our wildcard domain *.astrocloud.tech, ingress is used on the Kubernetes cluster to expose the services.

Dynamic hosting workflow incudes the user uploading the code of the app meant to be deployed, an Image is built for that app the app is containerized, then it's deployed on the Kubernetes cluster, then an ingress is created on the cluster to expose the application. (**further explanation will be provided for the workflow**)

In the first iteration of dynamic hosting, we have implemented the first stage of dynamic hosting which contains:

- Handling upload request.
- Saving user and projects data on our database.
- Building image for the code meant to be uploaded, and get the app containerized.
- Send build logs to the frontend so user can check it while the app is being built.
- Deploy the container on a local Kubernetes cluster (Cloud cluster in next iterations).
- Deployment of app on the Kubernetes cluster involves multiple steps:
    - Creating a deployment resource to the built image on a container.
    - Creating a service resource to expose that application deployed.
    - Create an ingress to expose the service for that application for outer access.
    - Implementing a webserver that routes the traffic from local machine to that ingress.

These are the basic feature implemented in the dynamic hosting in the first iteration; more features are implemented in later iterations such as auto scaling, health check, runtime logs streaming, monitoring, delettion of projects or updating them.

Structure:

Frontend: responsible for user interface, allows users to interact with our service, provides them the ability to upload projects, delete or update already uploaded projects, monitor their uploaded projects and so on.

Backend: is the intermediary between the frontend and the platform server. It exposes the API connecting to the frontend, allowing users to login and make upload requests. It communicates with the platform server to send the users upload request to the platform to handle it. It is connected to the database that stores the users and their projects' data.

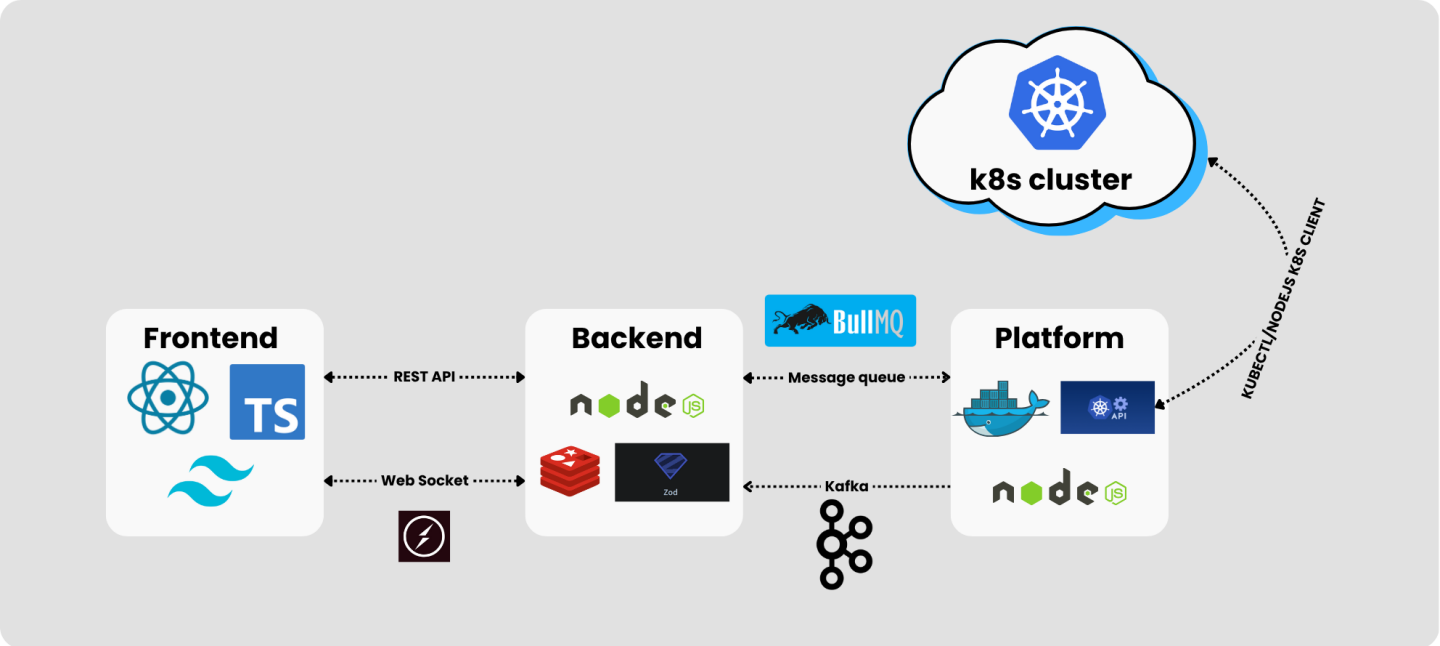
Platform: the platform server is the core of our application, it's where deployment takes place, in dynamic hosting the platform is responsible for cloning the repo of the projects to upload, build container images, deploy them to Kubernetes cluster. It receives upload requests from the backend and starts the deployment process.

Cluster: consists of multiple Kubernetes nodes including master node and worker nodes. It's the environment where apps are deployed and managed. In the first iteration, a local Kubernetes cluster was used. In later iterations, deployment was moved to a cloud-based environment using the azure Kubernetes service (AKS) for the projects to be publicly deployed and accessed.

Communication:

Frontend communicates with the backend via REST API, while backend communicates with the platform via message queues. Platform communicates with Kubernetes cluster via the Kubernetes NodeJS client.

Also, the platform sends logs back to the backend using Kafka, and from the backend it's sent to the frontend using web sockets.



Workflow:

1. User makes a project upload request. The request is sent to the backend via RESTAPI. The request usually contains the user auth data, the project needing upload repository URL, additional information regarding the project such as name, runtime environment (NodeJS), git branch, working directory, build command, health check URL, env vars.
2. Backend validates the user and creates a new project entry in the database.
1. Backend then sends a job to the platform via message queue.
2. Platform begins executing the process of building the application and deploying it to the cloud:
    a. Starting with cloning the git repo in the platform server local machine.
    b. Then create the Docker file containing build stages for that code.
    c. Then build the docker image from the code.
    d. Pushes the built image into azure registry to save it for when the k8s deployment fetches it.
    e. Starts deploying the image on Kubernetes cluster:

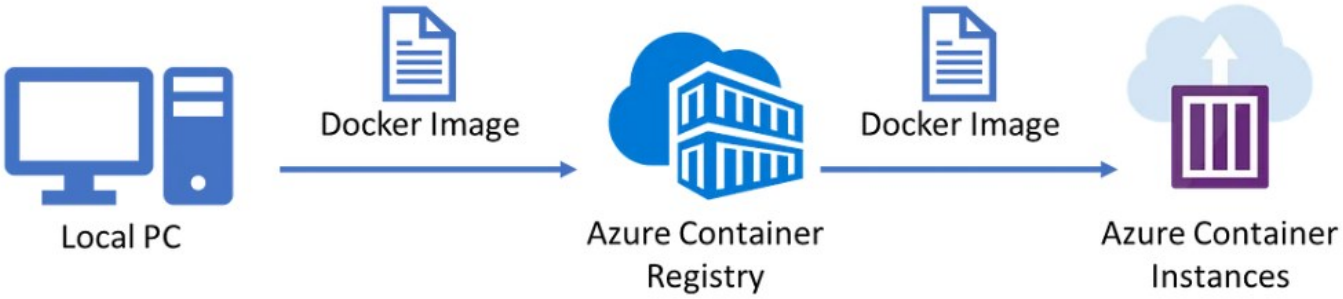
        i. Create a configMap to store the env variables if they exist to be sent to the deployment manifest.
        ii. Create a deployment resource on Kubernetes containing application metadata, container image, container port, configMap, resource limits (CPU/memory limits), livenessProbe, Production configurations such as runtime class name.
        iii. Create a service resource to expose that application.
        iv. Create an ingress to expose that service that has been created to outside access.
3. Platform should be sending build logs to the backend via Kafka all through the build process, and the backend sends those logs back to the frontend using webSocket.
4. Once the build and deploy of the app is done, results should be sent from platform to backend through the message queue with the status of the build whether it's a success or failure.
5. The status of the deployment is stored in the database, and the user should now have access to the deployed app (if deployment is successful).

This is the general workflow of the dynamic hosting process.

Technologies used:

- Frontend: **React, Tailwind** and **Typescript** for frontend website development.
- **Nodejs** as the runtime environment for the backend server and the platform server.
- **Bull** message queues built on top of **redis** servers to send messages from backend to platform and vice versa.
- **Kafka** (an object streaming service) is used to send logs live from the platform to the backend server.
- **Socket.io** to send logs back to the frontend.
- **Docker** is the main containerization tool used to build images for applications and containerize them.
- **Dockerode**: is a Node.js library that provides a programmatic interface to the Docker Engine API. It lets the platform interact with the docker engine to build images and monitor logs.
- **Nodejs Kubernetes-client**: A library that allows interactions between NodeJS applications and Kubernetes clusters.
- **AKS** (Azure Kubernetes service): It's the cloud provider of the kubernetes cluster.
- **ACR** (Azure container registry): is a private container image registry service provided by Microsoft Azure. It allows you to build, store, manage, and deploy Docker container images and OCI artifacts securely on Azure. It's used to store images built for the apps to be deployed so that the Kubernetes cluster can access them to deploy them.
- **Kubernetes** is the container orchestration tool used to manage the deployment of applications. It provides various management options for deployment, making it the best choice to manage deployments.

ACR:


Local PC
Docker Image
Azure Container Registry
Docker Image
Azure Container Instances

Iteration 2

ZIP File Upload

Overview:

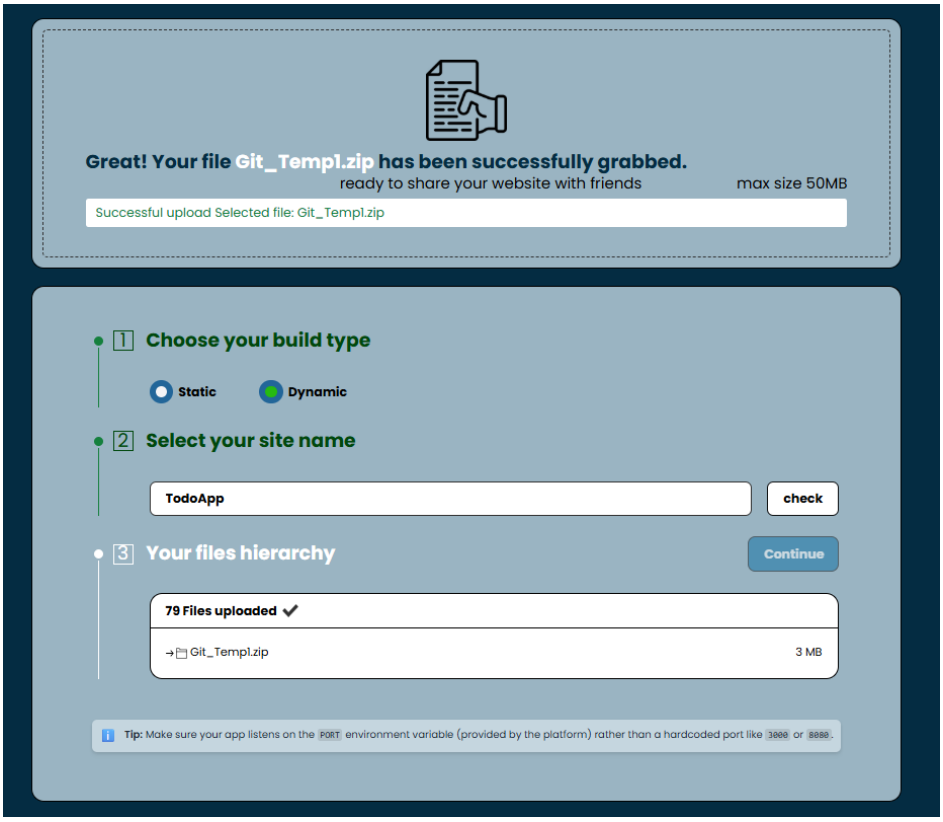
The manual upload feature enables users to deploy dynamic websites by uploading a ZIP file containing the project source code, selecting a dynamic build type, and configuring deployment parameters. The frontend integrates with a backend to handle file uploads and deployment triggers. The feature provides a seamless user experience with real-time feedback, error handling, and navigation.

workflow:

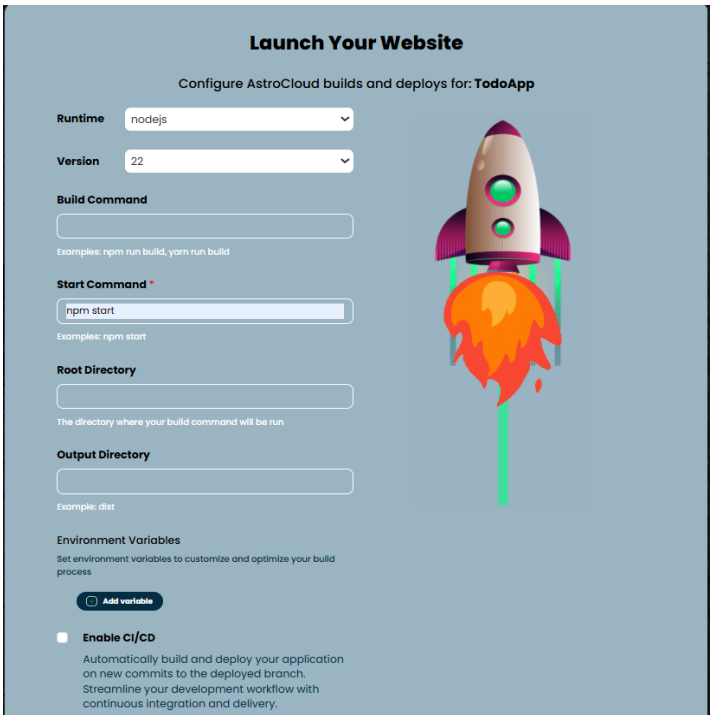
1- User is prompted to select a ZIP file, site name, and build type which is dynamic for this feature.
2- ZIP file is processed to check for size limits and emptiness, with a preview of its hierarchy.
3- Site name is validated via an API call to ensure correctness of site name.
4- User tigger deploy operation by clicking deploy button.
5- Frontend request upload URL from backend.
6- Backend create azure container and generate upload URL for that container and return container name and URL to frontend.
7- User will be redirect to the build form configuration page to fill other required fields like build command, start command, runtime, environment variables.
8- User tigger deploy operation by clicking deploy button.
9- Form data is sent to backend to trigger deployment and application link returned to user.

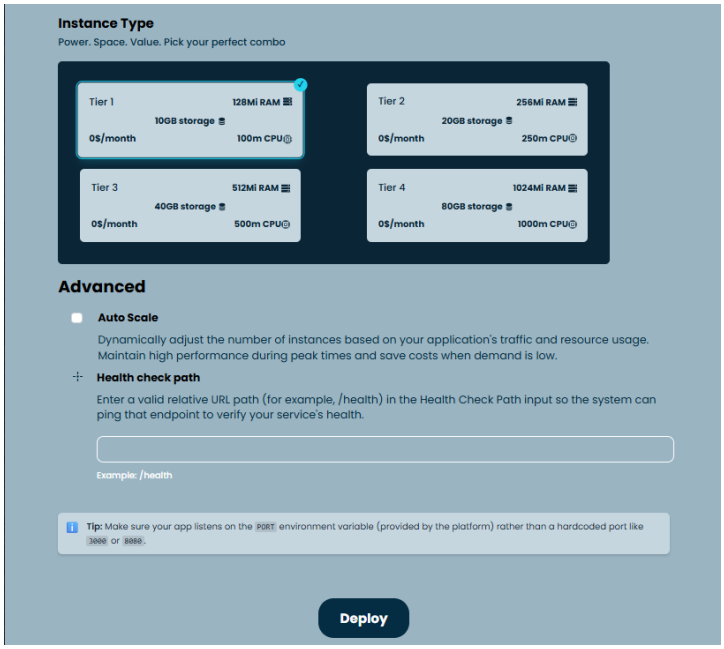
Example:

1-User select required dynamic project to deploy and select build type and provide site name and check it's validity and then click continue,



2. User provides build configuration fields like runtime version and build, start commands , environment variables, ...etc., then deploy.





3. Finally user visit provided link to access his application

CI/CD and Rebuild Project

Rebuild Project:

Rebuilding dynamic projects differs significantly from static project rebuilds.

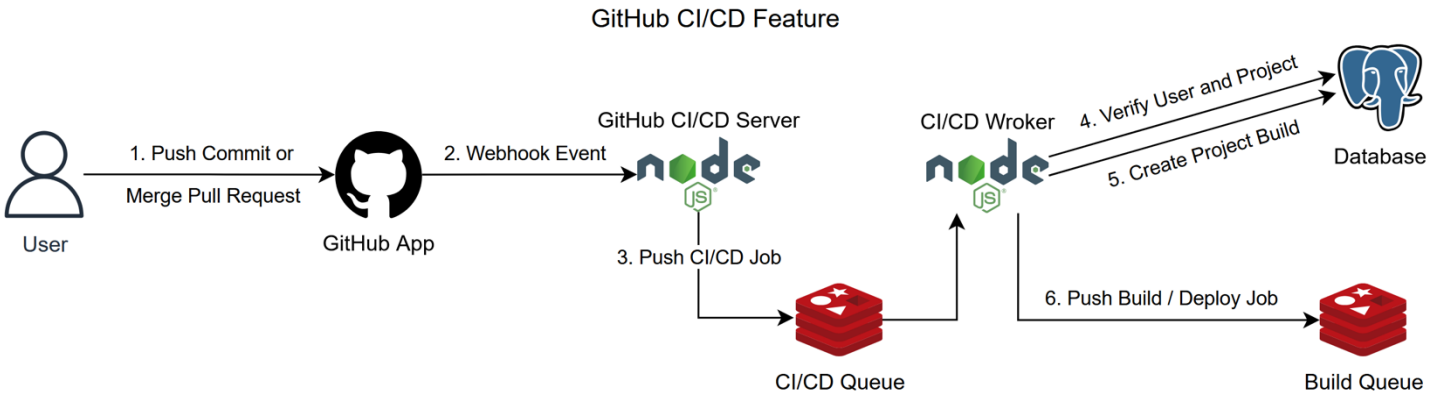
For static projects, the rebuild process involves retrieving the project files, building the project, and reuploading the resulting build files, a flow very similar to the original deployment process. In contrast, dynamic projects follow a more complex deployment pipeline: fetching the source files, building a Docker image, provisioning the necessary Kubernetes resources (such as deployments, services, and ingress), and finally uploading the image to the pod.

Upon inspection, we found that during dynamic project rebuilds, we can bypass the recreation of Kubernetes resources since they already exist. This optimization significantly reduces the rebuild time and improves performance.

CI/CD:

As part of expanding the system to support dynamic projects, it became necessary to decouple the GitHub webhook handler from the need to query the database during initial webhook event processing.

Architecture



Implementation

From comparing the old "architecture" found at the GitHub CI/CD at the static hosting milestone, we will find that the addition of the CI/CD Queue and Worker.

This queue aims at decoupling the GitHub CI/CD server from the main API server by making the worker access the database instead of the GitHub CI/CD Server.

In the old implementation, the server would assume that the user deployed the project once, but now we allow updating multiple projects if they exist.

The current flow now is as follows:

1- The user pushes a commit or merges a pull request to the deployed branch.
2- GitHub API send the webhook event to the GitHub CI/CD server
3- GitHub CI/CD server validates the body of the request only and pushes the payload to the CI/CD queue.
4- CI/CD Worker processes the job and processes the user and project data, attempts to check if the user and project exist, and that the user has enabled CI/CD for this deployed repository and branch.
5- CI/CD Worker finds all projects that meet the criteria and pushes them to the build job queue.

Benefits

- Fast GitHub Response:

GitHub expects a 2xx response within 10 seconds; this implementation returns 200 OK immediately to prevent retries and ensure reliability under load.

- Improved Reliability:

DB errors or delays don’t break the webhook flow since errors are handled in the background job.

- Better Scalability:

We can scale workers independently to handle high event volumes without stressing the API server.

- Separation of Concerns:

Keeps the controller lightweight (just validation + enqueue) and moves processing to dedicated workers.

Production Kubernetes Cluster Setup

For production deployment, we provisioned a managed Kubernetes environment using **Azure Kubernetes Service (AKS)**. This cluster hosts all critical platform components and user-deployed workloads in a scalable, secure, and cloud-native environment.

To enable seamless communication between the platform and the production cluster, we exported the AKS kubeconfig into the **platform container**, allowing it to authenticate with the Azure Kubernetes API server and issue deployment commands just as it would in development. This ensures that the platform maintains direct, programmatic access to production cluster resources.

Third-Party Observability Tools

To maintain observability and operational insight in the production cluster, we deployed two core tools using single, declarative manifests:

Fluent Bit for Log Collection

A **Fluent Bit** agent is deployed on each node as a **DaemonSet**, allowing us to collect logs from all containers and forward them to a centralized logging backend. This provides real-time log visibility and supports troubleshooting and analytics.

Fluent Bit was installed using a single manifest file, applied with:

```
> kubectl apply -f fluent-bit.yaml
```

This manifest includes the DaemonSet definition, config maps, and necessary service accounts.

Prometheus for Metrics Monitoring

Prometheus was installed to collect performance metrics from the cluster and all deployed services. These metrics power dashboards and alerting systems to ensure system health and performance visibility.

Prometheus was also installed using a standalone manifest applied via:

```
> kubectl apply -f prometheus.yaml
```

The manifest includes the Prometheus server, a service monitor, and necessary role bindings for cluster-wide metric scraping.

NGINX Ingress Controller

To expose internal services to external clients, we deployed the **NGINX Ingress Controller**. It acts as a reverse proxy and routes incoming HTTP/S traffic to the correct services inside the cluster based on Ingress rules.

The controller was installed using a single manifest:

```
> kubectl apply -f nginx-ingress.yaml
```

This setup includes:

- A Deployment of the Ingress Controller
- A Service of type LoadBalancer to expose it externally
- RBAC roles and bindings for permission management

Each application or service is defined with an **Ingress resource** that specifies path-based or host-based routing rules, enabling clean and organized service exposure through custom domains or subdomains.

Secure Image Pulling from Azure Container Registry

During dynamic hosting, user-deployed applications are packaged as private Docker images hosted on **Azure Container Registry (ACR)**. To enable AKS to pull these images securely, we created a **Kubernetes Secret** of type docker-registry containing Azure authentication credentials.

This secret is referenced in the relevant **PodSpecs** under the imagePullSecrets field, ensuring that private images can be accessed without exposing credentials in plaintext.

Build Rollback

Overview

To implement the rollback feature, we drew inspiration from the Kubernetes "`kubectl rollout`" command. While this command is commonly used in manual Kubernetes operations, it does not have a direct equivalent in the Kubernetes Client API module that we use to communicate with Kubernetes programmatically.

Upon inspection, we discovered that "`kubectl rollout`" is essentially a client-side wrapper around three lower-level operations:

1. **Listing all ReplicaSets** owned by the Deployment.
2. **Sorting them by revision** using the annotation: "`metadata.annotations['deployment.kubernetes.io/revision']`"
3. **Patching the Deployment**'s "`spec.template`" to match the pod template of a previous ReplicaSet.

By understanding and replicating the behaviour of these operations, we were able to manually implement a rollback mechanism within our platform.

ReplicaSets

ReplicaSets are a fundamental component in Kubernetes responsible for maintaining a stable set of running pod replicas for a given application. Each time a Deployment is updated, such as when a new container image is deployed or when the pod specification is modified, Kubernetes automatically creates a new ReplicaSet to represent that version of the Deployment. Previous ReplicaSets are preserved (unless explicitly deleted), effectively serving as versioned snapshots of the application's deployment history.

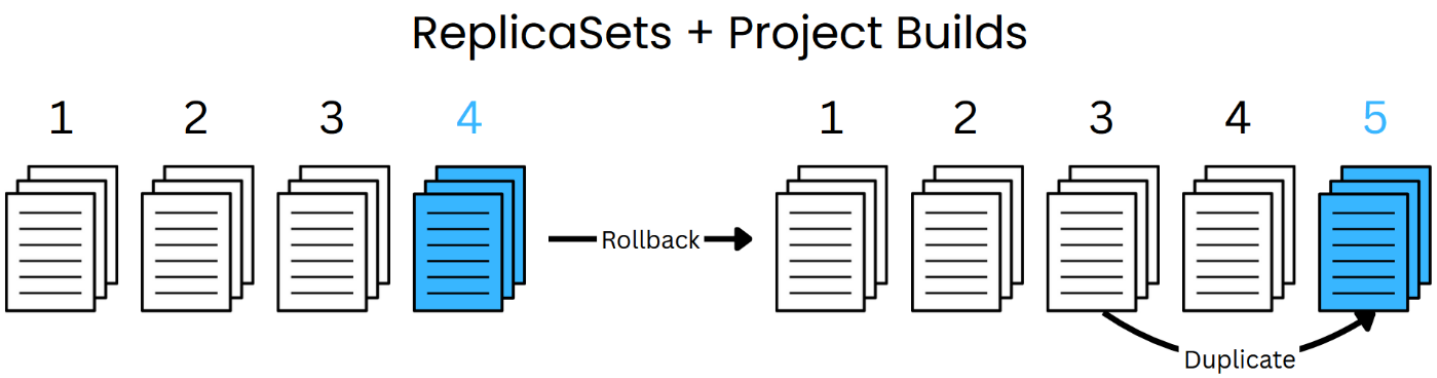
Implementation

We integrated rollback operations into the existing queue by introducing a new job type: "`dynamic:rollback`".

In our rollback implementation, **ReplicaSets** play a central role. When a user initiates a rollback, the backend API enqueues a "`dynamic:rollback`" job to the build queue. The build worker then processes this job through the platform, executing the rollback workflow as follows:

1. **List all ReplicaSets** associated with the target Deployment.
2. **Sort them** based on the "`metadata.annotations['deployment.kubernetes.io/revision']`" value to determine the revision order.
3. **Perform validation checks** to ensure that a valid ReplicaSet exists for the requested rollback revision.
4. **Patch the Deployment**'s "`spec.template`" with the pod specification from the selected ReplicaSet.

This process instructs Kubernetes to update the currently running pods using the configuration from the specified historical version. As a result, the application is restored to a previously stable state without requiring a new build or redeployment.



While effective for basic rollback needs, this implementation introduced several limitations that affect reliability and user control. These shortcomings are addressed in the following “Set Active Build Version" section.

Iteration 3

Set Active Build Version

Shortcomings of the Build Rollback Feature

While the "Build Rollback" feature, described in the previous iteration, offered a quick recovery path, it introduced several limitations:

1. **Redundant Build Creation:**
   Each rollback operation requires creating a new build entity in our system to mirror Kubernetes behavior, which generates a new ReplicaSet even when reverting to a previous version. Although no new code or configuration was introduced, this step was necessary to stay consistent with the platform deployment model. However, it introduced unwanted duplication, as users have a redundant build being created despite no actual changes.
2. **Unreliable ReplicaSet Retention:**
   Kubernetes may delete older ReplicaSets even if the configured ReplicaSet limit has not been reached. As a result, rollback operations could fail if the target ReplicaSet has been unexpectedly removed.
3. **Fixed One-Version Rollback:**
   The rollback mechanism was limited to reverting only one version at a time, offering no flexibility in selecting arbitrary historical builds.
4. **Rollback Looping Issue:**
   Due to the way Kubernetes generates ReplicaSets, repeated rollback actions can loop back to the same build, preventing the user from rolling back through the full history. This behavior contradicts the intended purpose of a rollback and creates confusion for users.

Overview

To overcome the limitations of using ReplicaSets for rollback, we introduced the **Set Active Build Version** feature. This enhancement allows users to select **any successful build** and activate it as the current deployment version.

Avoiding the Need for ReplicaSets

ReplicaSets store a snapshot of the deployment configuration, primarily the **container image** and **ConfigMap reference**. Upon close inspection, we found these two fields to be the only significant variables between different builds. By tracking the image tag and associated ConfigMap for each build, we eliminate the need to rely on ReplicaSets altogether.

ConfigMaps

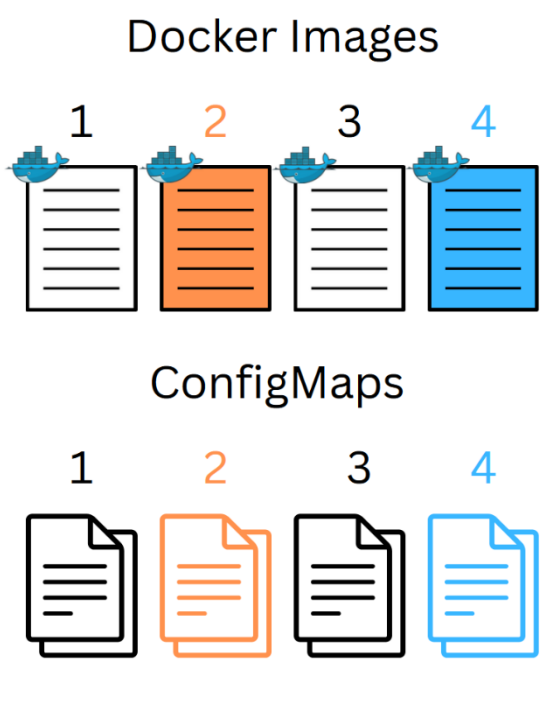
A **ConfigMap** in Kubernetes stores non-confidential key-value configuration data, which can be consumed by pods as environment variables. This allows application configurations to be decoupled from container images, improving portability and deployment flexibility.

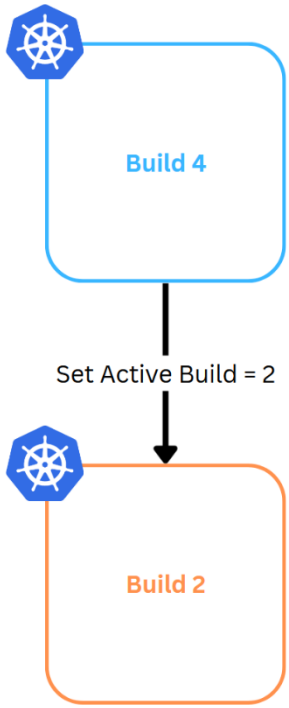
We leverage ConfigMaps to store the environment variables associated with each build.

Implementation

- **Build Metadata Storage:**
  For every new build, the platform creates:
  - A **ConfigMap** named in the format: "`<build-hash>-v<buildNumber>`", containing the build's environment variables.
  - A **Docker image** tagged as: "`<build-hash>:v<buildNumber>`".

- **Tracking Active Build:**
  The backend maintains the active build state using the "`activeBuildNumber`" field in the project's database record. This field is updated on every successful deployment or when switching to another build.
- **Build Selection Workflow:**
  When a user selects a specific build to activate, the backend enqueues a "`dynamic:rollback`" job to the build queue. The build worker processes this job by:
  - Listing available images for the deployment and validating that the selected image exists.
  - Verifying that the associated ConfigMap exists and retrieving its environment variables.
  - Patching the Kubernetes Deployment by updating:
    - "`/spec/template/spec/containers/0/image`" with the selected image tag.
    - "`/spec/template/spec/containers/0/envFrom/0/configMapRef/name`" with the selected ConfigMap name.
  - Updating the project's "`activeBuildNumber`" and stored environment variables to reflect the selected build.





Benefits

1. **Flexible Build Selection:**
   Users can now activate any successful build, not just the last one, while still retaining a "Rollback" button for quick one-step rollback.
2. **No Dependency on ReplicaSets:**
   The rollback mechanism no longer depends on Kubernetes ReplicaSet retention, increasing reliability and predictability.
3. **No Redundant Builds:**
   Activating an old build does not create a new build record or ReplicaSet, preserving clean build history.
4. **Consistent Rollback Behavior:**
   Users can navigate through the full build history without encountering the rollback loop issue from the previous implementation.

Health Check

Overview:

Health check lets Kubernetes check for the liveness of a certain deployed app overtime so in case the application is down for some reason, Kubernetes restarts that pod.

Kubernetes provides a liveness probe that checks the liveness of the application over time, It sends a request on the health check path provided (such as /healthz) and if a response of 200 is sent back, It means the pod is running well, if a response of 400 is received, it restarts that pod.

Implementation:

On creating a deployment resource on Kubernetes for a specific project, if a user provides a health check path, a Liveness Probe is created in that deployment manifest.

The liveness Probe should specify the time interval by which Kubernetes checks for that pod, The initial delay which is the time by which the Kubernetes should not check for the liveness of Probe since its initialization, The failure threshold which is the number of consecutive failures that Kubernetes triggers a pod restart after.

Auto-scaling:

Overview:

Auto-scaling is the increasing number of pods for an application if its usage (CPU/Memory) passes a certain threshold.

If the usage decreases back again to be less than threshold, The number of pods is decreased back again.

In our implementation, the threshold is 50% of (CPU or memory) usage of the requested amount of resources for that application.

The minimum number of pods for an app is one, while the maximum number is two pods.

Implementation:

When a project is being hosted and deployed on the Kubernetes cluster, an hpa (Horizontal pod autoscaler) resource is created with the minimum number of pods, maximum number of pods, the app to which it will affect and the threshold of the resources (CPU/Memory) usage that after exceeding it, the pods will be horizontally scaled (increased).

Code Example:

```
const hpa: V2HorizontalPodAutoscaler = {
    apiVersion: 'autoscaling/v2',
    kind: 'HorizontalPodAutoscaler',
    metadata: {
        name: `${resourceName}`,
        namespace: 'default',
    },
    spec: {
        scaleTargetRef: {
            apiVersion: 'apps/v1',
            kind: 'Deployment',
            name: `${resourceName}`,
        },
        minReplicas: 1,
        maxReplicas: 2,
        metrics: [
            {
                type: 'Resource',
                resource: {
                    name: 'cpu',
                    target: {
                        type: 'Utilization',
                        averageUtilization: 50, // 50% of requested amount
                    },
                },
            },
            {
                type: 'Resource',
                resource: {
                    name: 'memory',
                    target: {
                        type: 'Utilization',
                        averageUtilization: 50,
                    },
                },
            },
        ],
    },
};
```

Monitoring:

This feature enables users to get analysis of their deployed applications; monitoring provides metrics of the resource (CPU/ memory) usage of the apps for the users.

Prometheus tool was used to collect metrics from pods running on the Kubernetes cluster.

Prometheus is an open-source monitoring tool used to collect metrics for apps running on Kubernetes clusters.

Implementation:

On creating the Kubernetes cluster, Prometheus is installed on that cluster, it runs as a service as any other service on Kubernetes, so it's exposed to outside network by an ingress.

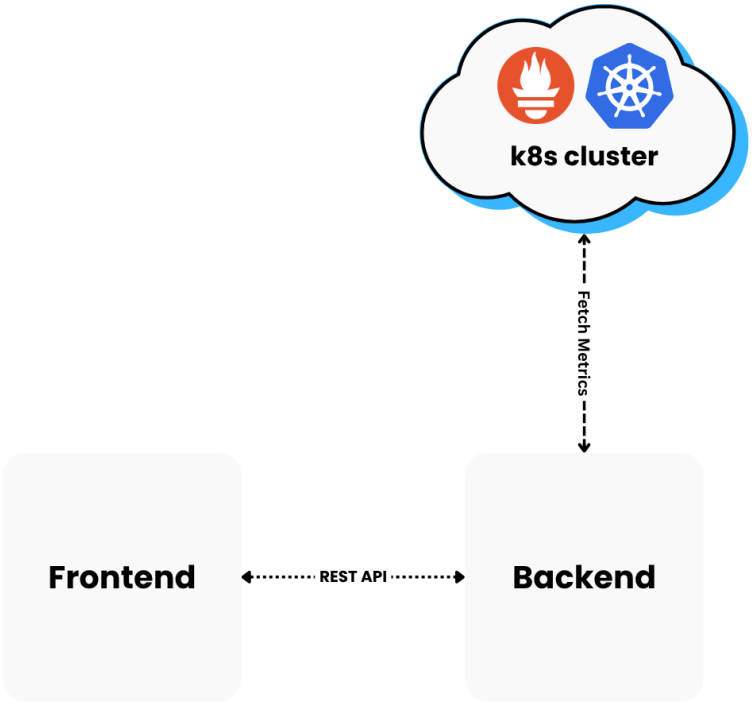
It should be exposed so that we can fetch metrics from it using an endpoint.

Once exposed via an API endpoint, it can be called via any REST request to fetch the metrics.

In our application, we've focused on providing resource (CPU/Memory) usage to the end users as a percentage.

Workflow:

1. The frontend requests the metrics from the backend.
2. The backend fetches the metrics from Prometheus service running on the Kubernetes cluster via REST API.
3. The backend then sends these metrics back to the frontend.



Testing and Optimization Milestone

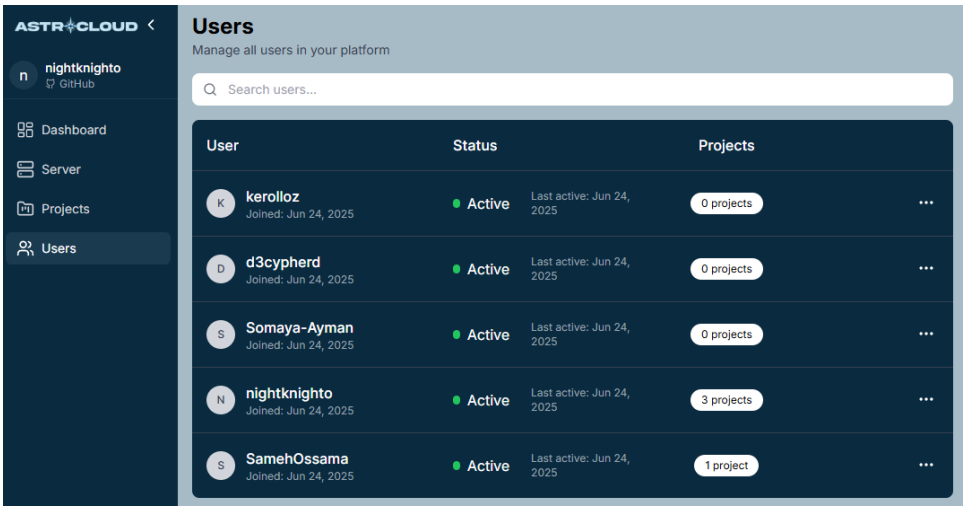
This milestone focused on improving efficiency across both the system and development lifecycle by optimizing workflows, automating processes, and reducing manual overhead. Key efforts included streamlining deployment pipelines, enhancing testing strategies, and refining authentication mechanisms—all while minimizing repetitive work and eliminating bottlenecks. The goal was to accelerate development cycles, improve maintainability, and ensure smoother operations through systematic optimization of both technical and operational processes.

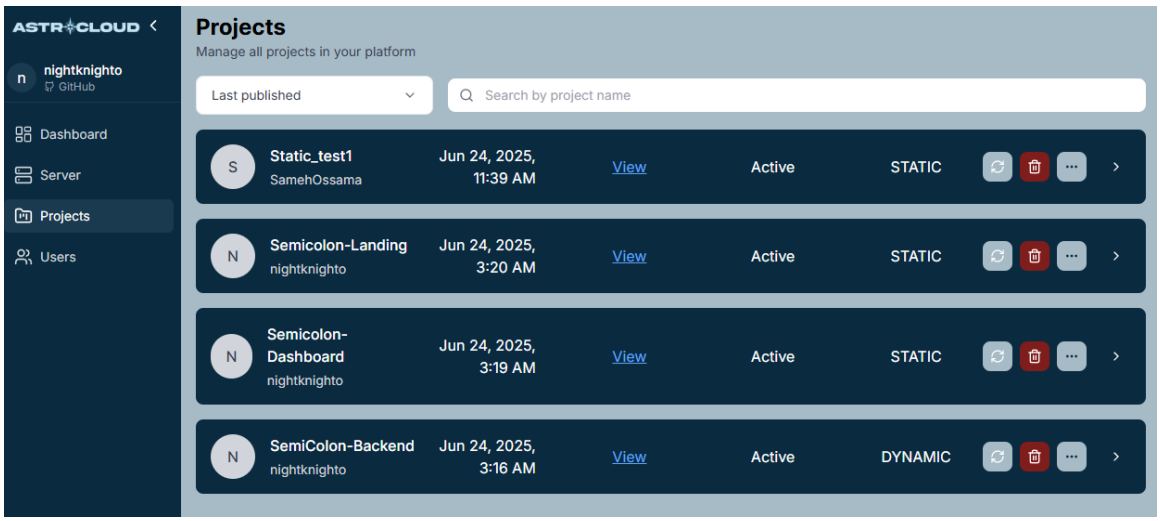
Admin Dashboard

The admin dashboard provides a RESTful APIs that allow administrators to manage and monitor the system's users and projects. It exposes endpoints to retrieve statistical data, perform CRUD operations on users and projects, and support filtering and pagination.

Key features

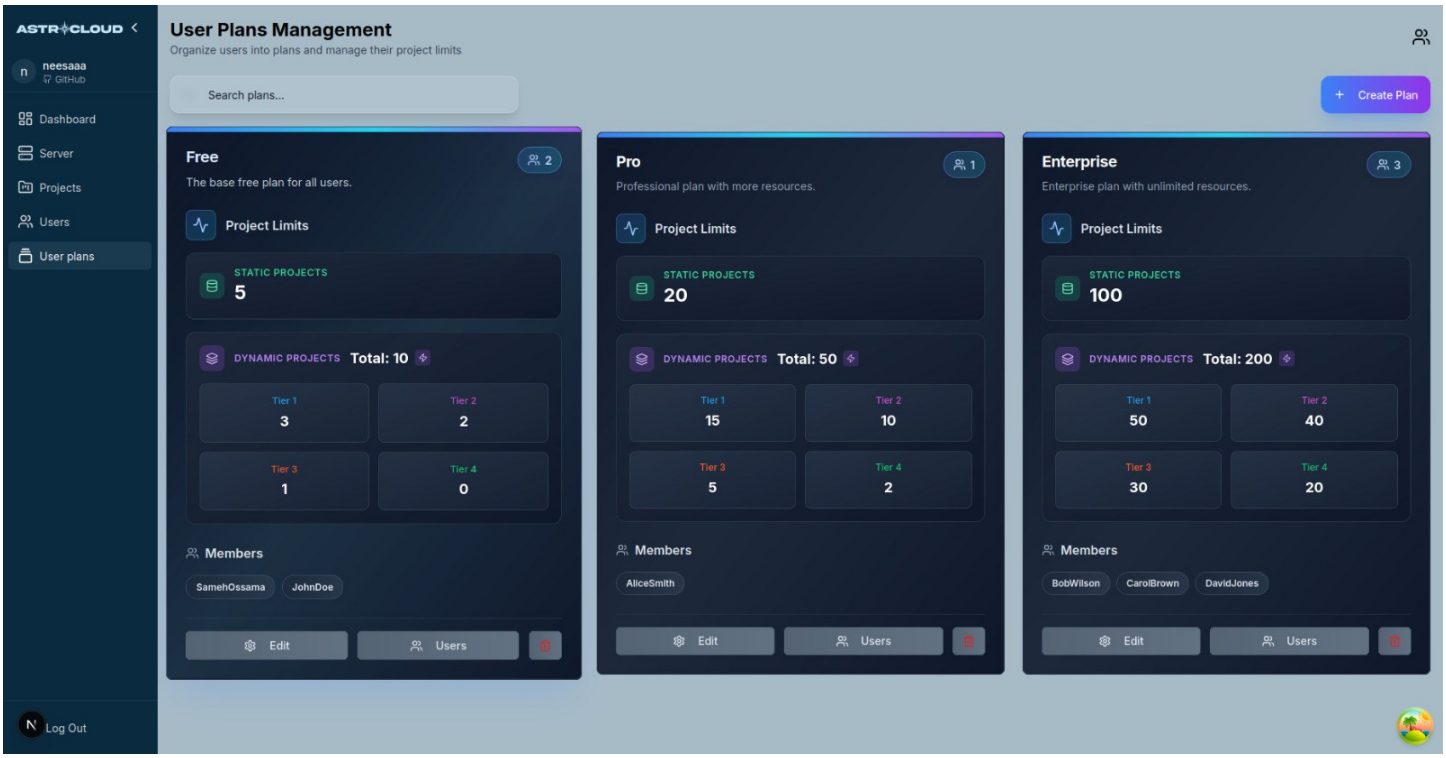
- **User Statistics:** View the number of new users on a daily, weekly, and monthly basis.
- **Project Deployment Stats:** Monitor how many projects are being deployed over time.
- **Recent Users:** Get a snapshot of the most recently registered users, along with their activity status and number of projects.
- **User & Project Management:** From the dashboard, admins can navigate to detailed views where they can edit user information, explore user projects, or manage deployments.





User Plans

The long-term viability and scalability of a cloud hosting platform like AstroCloud depend on its ability to effectively manage resource allocation and monetize its services. This chapter details the design and implementation of the User Plans Management system, a core administrative feature engineered to address this need. The system introduces a hierarchical structure of "Plans" to abstract resource entitlements, enabling administrators to enforce usage limits, segment users into distinct tiers, and streamline the process of upgrading or downgrading user accounts. This module is critical for implementing business models such as Freemium, tiered subscriptions, and enterprise-level packages.



System Architecture and Core Concepts

The User Plans Management system is built upon a set of core architectural entities that work in concert to provide granular control over the platform's resources.

The "**Plan**" Entity

A "Plan" is the fundamental data model in this system. It represents a contractual agreement of service level and resource allocation. It is not merely a group but an abstract template of entitlements that can be dynamically associated with any number of users.

Each Plan object contains the following key attributes:

- **Plan name**: A unique, human-readable identifier (e.g., "Free," "Professional," "Enterprise").
- **Plan description**: A string providing context for the plan's intended use case.
- **Resource quotas**: Various fields containing specific numerical limits for all managed resources.

The system is designed to govern two primary categories of user-deployable projects, each with its own consumption logic:

- **Static Projects**
  This is a simple integer quota representing the maximum number of static web projects a user can host. These projects (e.g., HTML/CSS/JS, frontend frameworks) have predictable, low-intensity resource footprints.
- **Dynamic Projects & Tiered Allocation**
  This represents a more complex and critical resource. Dynamic projects (e.g., Node.js, Python, Go backends) have variable demands for CPU, RAM, and bandwidth. To provide granular control and support a value-based pricing model, the Dynamic Projects quota is sub-divided into four distinct Tiers:
    - **Tier 1**: Intended for low-resource applications, such as development/staging environments or lightweight APIs.
    - **Tier 2**: A standard tier for production-grade applications with moderate traffic.
    - **Tier 3**: A performance-oriented tier offering higher CPU/RAM allocations for demanding applications.
    - **Tier 4**: An enterprise-grade or specialized tier, potentially offering dedicated infrastructure, advanced security features, or the highest level of performance.

  The Total Dynamic Projects limit for a plan is a hard cap; users cannot exceed this total even if tier-specific limits haven't been reached. Combined with tier-level quotas, this model allows the platform to control the distribution of high- and low-cost applications, preventing users on low-cost plans from consuming excessive high-performance resources.

User-Plan Association

The system employs many-to-one relationships between Users and Plans. A single Plan can be assigned to many Users, but a User can only subscribe to one Plan at any given time. This association dictates the resource limits applied to the user's account. Changing a user's plan (an operation we term "Plan Migration") is an atomic transaction that updates this relationship, instantly applying a new set of resource quotas.

User Interface (UI) and Functional Walkthrough

The administrative interface is designed for clarity and operational efficiency, presenting complex data in an intuitive, actionable format.

The Plans Dashboard

The primary administrative view is a card-based dashboard. This design was chosen for its high information density and ease of comparison between different service tiers.

Plan Card: Each card acts as a high-level summary of a single Plan. It prominently displays the Plan Name, the number of associated users, and a full breakdown of the Static and Dynamic project quotas, including the individual Tier limits. This provides an administrator with an immediate, at-a-glance understanding of the platform's user distribution and resource allocation.

Action Affordances: Each card contains three primary action buttons:

Edit: Navigates to a form where an administrator can modify the Plan's name, description, and all associated resource quotas.

Users: Invokes the "Manage Users" modal, providing a focused interface for managing the user population of that specific plan.

Delete: A destructive action to permanently remove the Plan. For system integrity, this action should be disabled if there are still users assigned to the Plan, forcing the administrator to migrate them first.

The "Manage Users" Modal

This modal interface provides a dedicated workspace for managing the membership of a single plan, preventing clutter and cognitive overload.

Scoped Context: The modal's title clearly indicates which plan is being managed (e.g., "Manage Users - Free").

User List & Search: It presents a paginated list of all users currently assigned to the plan. A search field enables rapid filtering by usernames, which is essential for platforms with a large user base.

Plan Migration Mechanism: The core function of this modal is user re-assignment. Next to each user is a "Move" dropdown menu.

Workflow: An administrator selects a user, clicks "Move," and chooses a new destination Plan from the dropdown list.

Asynchronous State Change: This action does not execute immediately. Instead, it stages the change. The UI provides feedback by potentially disabling the dropdown and updating a central counter.

Commit Action: The "Save Changes (N)" button at the bottom serves as a final confirmation. The (N) indicates the number of pending migrations. This batch-processing design is deliberate; it allows an administrator to stage multiple user migrations in one session and commit them as a single transaction, improving efficiency and reducing the likelihood of error.

Contextual Information: A summary of the current Plan's limits is displayed at the bottom of the modal. This serves as a constant reference for the administrator, reminding them of the plan context without requiring them to close the window.

Design Rationale and Strategic Importance:

The implementation of the User Plans Management system was driven by several key strategic objectives for the AstroCloud platform.

Business Model Enablement: This system is the technical foundation for a tiered pricing strategy. It allows AstroCloud to offer a "Free" plan to attract a wide user base and then monetize the service by encouraging users to upgrade to "Pro" or "Enterprise" plans for higher resource limits and advanced features (represented by the Tiers).

Scalability and Control: As the platform grows, manual resource monitoring becomes impossible. This system automates the enforcement of quotas, preventing resource abuse and ensuring fair usage. The tiered approach to dynamic projects, in particular, allows the platform to scale its expensive compute resources in a controlled and profitable manner.

Administrative Efficiency: The centralized dashboard and focused management modals provide administrators with a powerful yet simple toolset. It reduces the time and effort required to manage thousands of users, adjust service offerings, and respond to customer needs for plan upgrades or downgrades.

Flexibility: The system is designed to be extensible. New resource types (e.g., "Databases," "Concurrent Builds") could be added to the Plan entity in the future with minimal changes to the core architecture. Similarly, the meaning of the "Tiers" can evolve, representing different cloud providers, geographic regions, or hardware specifications without altering the user management workflow.

Mono-repository

A major code architectural update was switching from an unrelated multi-project repository into a proper and full-fledged mono-repository.

Before, we had 3 projects inside our repository: frontend, backend, and platform. There had to be some shared definitions and functionalities between them, so we created a “shared” folder and put code files into it directly. It wasn’t a proper package, just files in a folder, and each project would directly import them via long relative links. Basically it was a quick work-around.

Main problems of the old approach

- **Doesn’t share packages:** It wasn’t possible to share installed packages in each one. If a package was needed in multiple projects, it had to be installed in each one individually.
- **Duplication:** Duplicated packages was a result of this approach, and also since each project is separate, all configurations and toolings were duplicated.
- **External Imports from shared folder:** The manual imports had major problems, due to importing code files from outside the project directory.
    - The import statements were incredibly long and brittle. Any change to the file structure of the shared folder would break the current imports.
    - Development servers would not pick up changes done to the shared folder.
    - It was difficult to make automation scripts, and the Dockerfile definitions of all the projects became much more complex due to this shared folder.
- **Tooling inefficiencies:** Due it not being a proper mono-repo, the tooling available was lacking. You couldn’t run multiple commands in parallel, or specify order of operations.

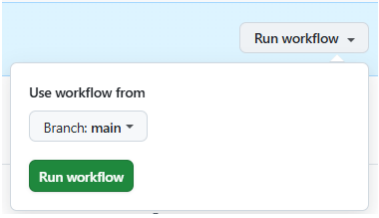
Mono-repository to the Rescue

Due to these problems, we switched to a mono-repository architecture using **NPM Workspaces**. We converted the shared folder into a standalone package, and linked it to all the projects, to be used like a typical external package.

Benefits of the New Approach

- **Third-party packages are shared**: NPM packages are shared among all projects in the repository, so saves space, time, and removes duplication.
- **Streamlines automation**: Simplifying from explicit imports of the shared folder to normal package imports made it much easier for automation scripts, simplified Dockerfile definitions, and removed brittleness of the imports.
- **Ability to isolate into sub-packages**: It is now possible to divide the shared code into multiple packages, and have projects import only the needed ones, instead of all the shared code.

Production Deployment Workflow

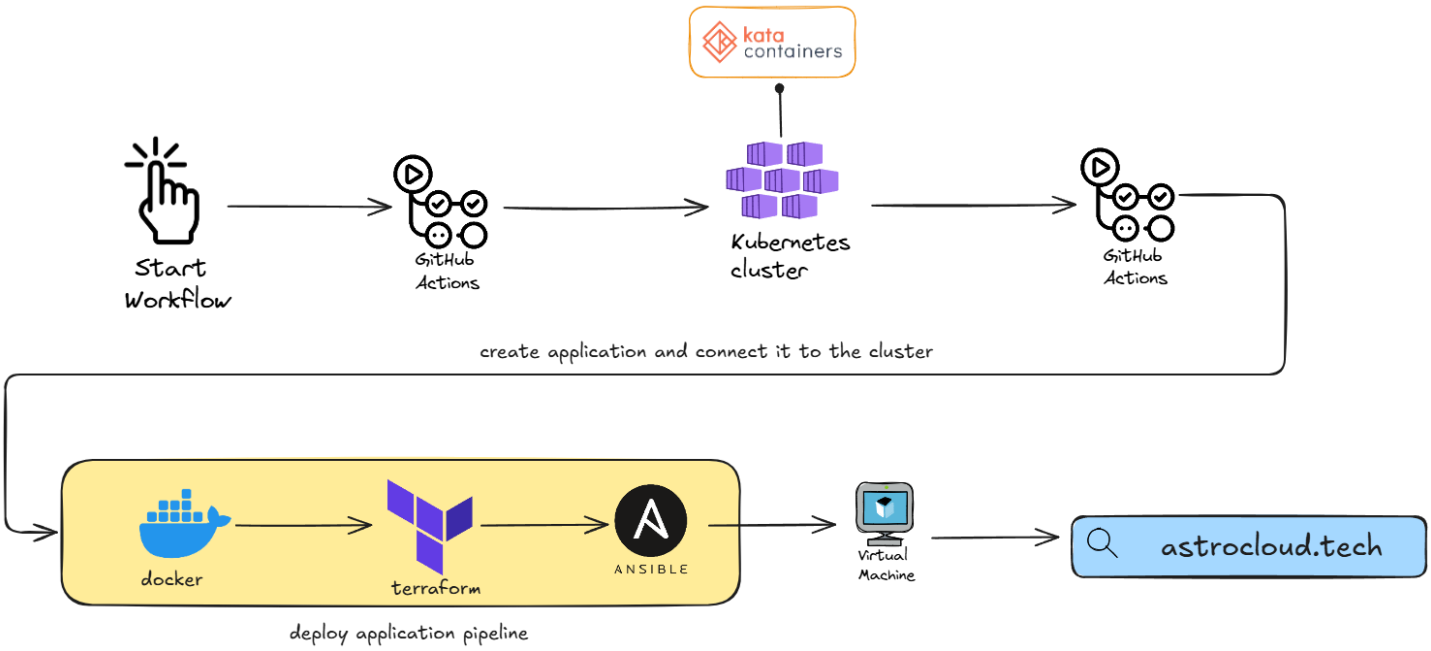


To minimize time-to-production and maximize reliability, we implemented a fully automated CI/CD pipeline that provisions infrastructure, configures environments, and deploys the application with a single trigger. Using Terraform, Ansible, Docker Compose, and GitHub Actions, this zero-touch workflow streamlines every phase from code commit to public accessibility with a single click.

Key optimizations include:

- **Automated Infrastructure Provisioning** on Azure using Terraform.
- **Agentless Configuration Management** with Ansible for consistent VM setup and Docker-based deployment.
- **Containerized CI/CD Execution** using Docker Compose to eliminate host dependencies.
- **SSL & DNS Automation** for instant secure access using Let's Encrypt and Cloudflare.
- **Production Kubernetes Cluster Setup** via GitHub Actions to spin up AKS clusters with Kata Containers.
- **One-Click Teardown Workflows** to cleanly destroy all resources after testing.

This dramatically reduces manual intervention, ensures repeatability, and enables anyone to try the application in a live environment with minimal effort.



Kubernetes Local Development Workflow

A critical component of dynamic hosting in our system is the **Kubernetes cluster**, which is responsible for running user-submitted applications. The **platform** is the control layer that interacts directly with the cluster, issuing deployment and management requests to the Kubernetes API server.

To simplify communication between the platform and the cluster during development, we embedded **Kind (Kubernetes IN Docker) inside the platform container itself**. This approach ensures that the platform has local access to the Kubernetes API server, as Kind automatically generates the default kubeconfig within the container environment. As a result, no additional network bridging or host configuration is needed — the platform can seamlessly interact with the cluster internally.

Challenge: Caching kindest/node Image Inside the Platform

One of the main development-time challenges we encountered was related to **caching the kindest/node image**, which is required by Kind to bootstrap Kubernetes nodes.

Problem

Attempting to pull the kindest/node image directly in the platform's Dockerfile does **not work** because the Docker daemon is not yet running at build time — and Docker-in-Docker (DinD) setups require the daemon to be started **after** the container launches.

Solution

To address this, we introduced a **Docker Registry Proxy** that:

- Acts as a caching layer for Docker images
- Stores pulled images in a **persistent volume** on the host machine
- Is used by the Docker daemon inside the platform container to fetch images

The platform's entrypoint script was updated to:

1. Apply the required Docker **proxy configuration** before starting the daemon
2. Start the Docker daemon with the proxy enabled
3. Ensure all image pull requests go through the registry proxy

This setup enables automatic caching of large images like kindest/node (~600MB). Once cached, they are served directly from the proxy on future requests — even across platform container rebuilds or deletions.

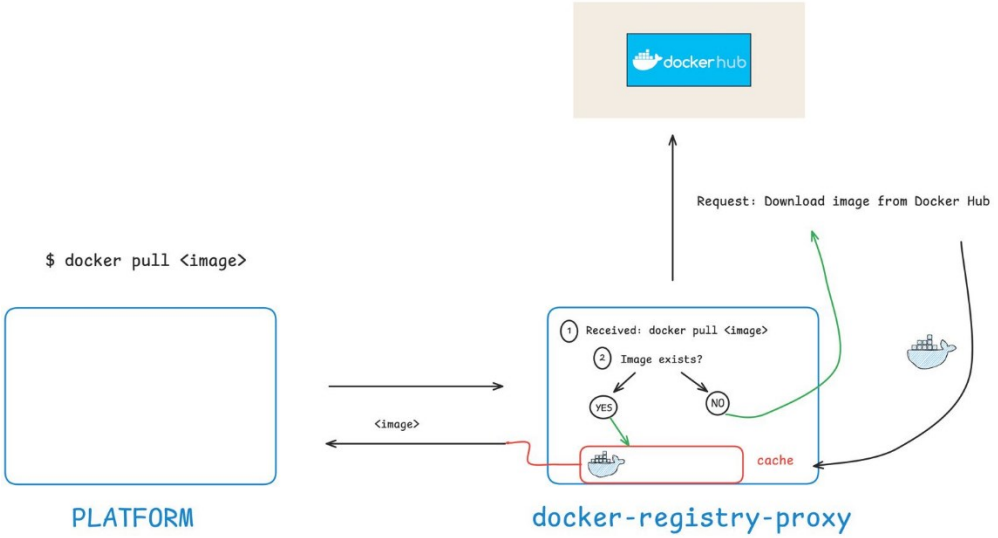


Figure: Caching docker images in docker-registry-proxy container

Result

This solution led to a **significant reduction in bandwidth usage (~600MB per fresh startup)** for the entire development team. Previously, every time the platform container was rebuilt or restarted, the image had to be re-downloaded. With the registry proxy in place, image pulls became nearly instantaneous after the first fetch, greatly improving local development efficiency and speed.

Refresh Token Authentication Strategy

Authentication Strategy – GitHub App Integration with Secure Token Management. AstroCloud uses a secure, token-based authentication system powered by **GitHub Apps**. This approach enables fine-grained access control, enhanced security, and a seamless developer experience across the platform. Authentication is managed through **JWT access and refresh tokens**, ensuring both session security and usability.

Authentication Flow with GitHub App

When a user initiates the login process, AstroCloud directs them to authenticate via the **GitHub App**. Upon user authorization, GitHub returns an authorization code to the frontend, which is then sent to the backend. The backend verifies the code with GitHub's API and, upon successful validation, issues two tokens:

- An **access token** (valid for **15 minutes**)
- A **refresh token** (valid for **7 days**)

The access token is used for authenticating user actions on the platform, while the refresh token is stored securely in an **HTTP-only cookie**, minimizing the risk of cross-site scripting (XSS) attacks.

Token Lifecycle and Renewal

The **access token** is short-lived by design. When it expires, the frontend detects the failure in API responses and silently attempts to obtain a new access token using the refresh token. If the refresh token is still valid, the backend issues a fresh access token without requiring the user to log in again.

This **silent token renewal** ensures a smooth and uninterrupted user session, even after temporary token expiry. Users can stay logged in for days without needing to reauthenticate—as long as the refresh token remains valid and securely stored.

Expiration and Re-authentication

Once the **refresh token** expires after 7 days, or if it is invalidated manually (e.g., logout or token theft mitigation), the backend rejects the token renewal request. In this case, the user is automatically redirected to the login page to reauthenticate through GitHub.

This expiry mechanism ensures **inactive sessions are safely closed**, reducing the risk of unauthorized access while maintaining a user-friendly experience.

Security and Best Practices

AstroCloud's strategy adheres to security best practices:

- **Access tokens** are short-lived and stored in memory
- **Refresh tokens** are stored in **HTTP-only cookies** to protect against XSS

- The authentication flow is protected from **CSRF** through secure cookie handling and server-side verification
- GitHub App scopes are narrowly defined, minimizing exposure

Health check endpoints for the system

A health-check endpoint is essential for verifying the connectivity and operational status of services across the backend and platform systems. These endpoints are integrated with a centralized monitoring system that evaluates service health and delivers real-time performance metrics to an administrative dashboard.

The system uses the following methods to assess the health of individual services:

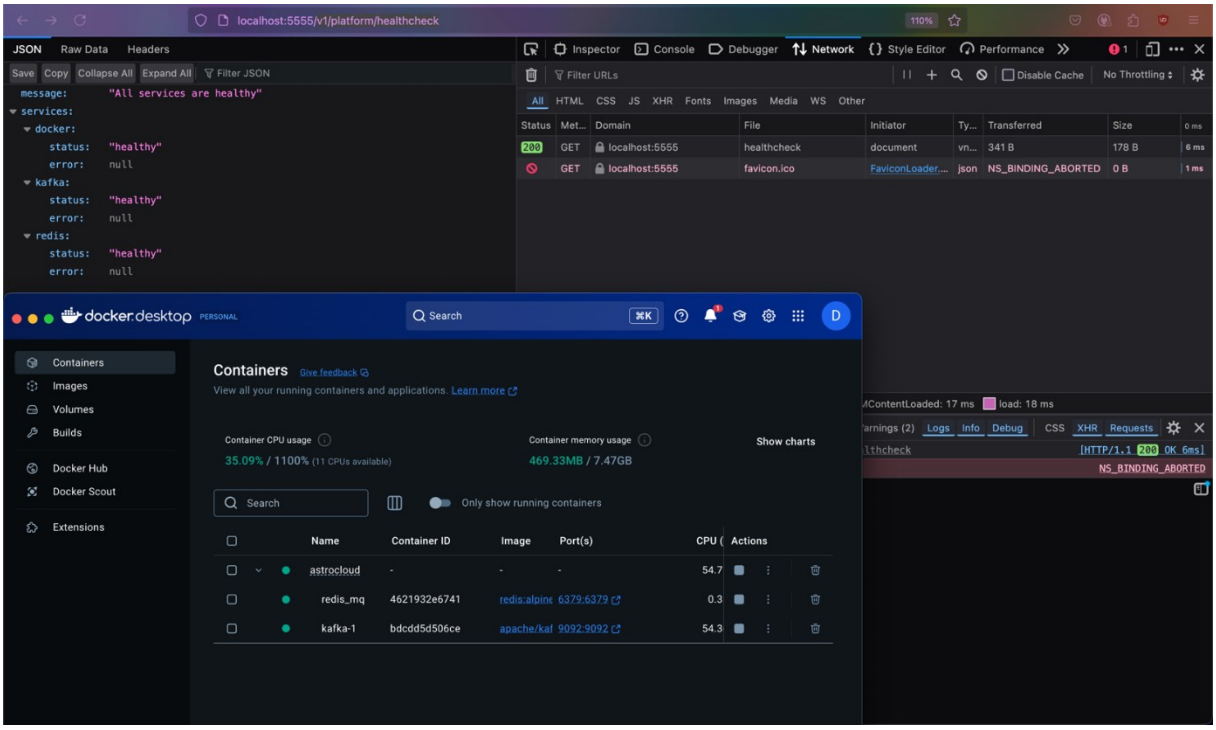
1. **Pinging**
   This method sends a basic request to the service's endpoint to verify that it is reachable and responding, without invoking any business logic.
2. **Connection**
   Establishes a connection (e.g., to a database or message broker) to ensure that the service is accessible at the network and protocol level.
3. **Dummy Test Operation**
   Performs a lightweight, non-intrusive operation (e.g., creating and deleting a temporary resource or executing a no-op command) to verify that the service is functioning correctly beyond just being reachable.

Each system exposes an internal /healthcheck endpoint that returns HTTP 200 for healthy, 503 for unhealthy, along with a JSON payload indicating health status of each service.

Sample Output

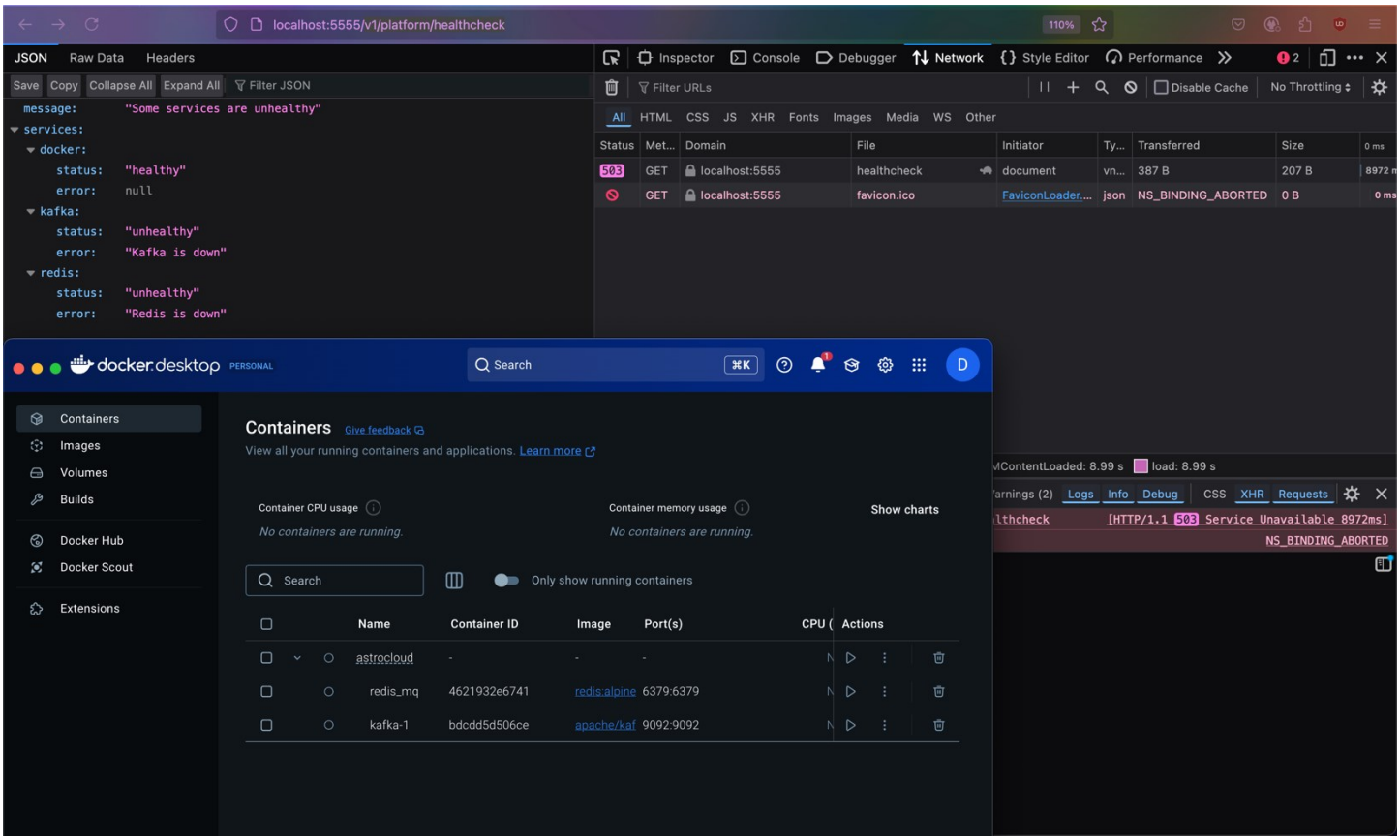
Healthy System

- All services are healthy, so returns **200 OK** with **healthy overall status**.



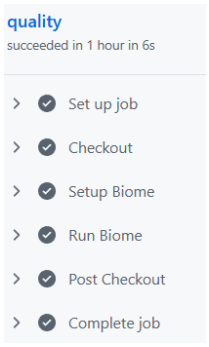
Failure in System

- Two services out of three are down, so endpoint returns **503 Service Unavailable** with **unhealthy overall status** and individual status of services.

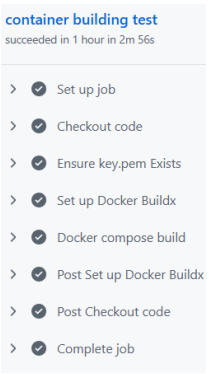


Quality Control CI Workflows

To optimize both code quality and deployment reliability in the development workflow, two GitHub Actions workflows were implemented as part of the CI/CD pipeline.



The first workflow focuses on code quality by integrating the **Biome** tool, a high-performance formatter and linter. This workflow is automatically triggered on every push to the main branch, pull request targeting main, or through manual execution via the GitHub Actions interface and ensures that all source files adhere to consistent coding standards, such as enforcing LF line endings and flagging only high-severity issues. By automating code linting, the workflow reduces the likelihood of introducing formatting inconsistencies or syntactic errors into the main branch, thereby improving code maintainability and readability.



The second workflow is designed to verify the integrity of the container build process. It attempts to build the production Docker containers using Docker Compose and is triggered under the same conditions as the first workflow. If the build succeeds, this confirms that the codebase is in a deployable state. However, if the build fails, it indicates that there are errors in the application code or configuration that must be resolved before deployment. This early detection mechanism helps identify and fix issues related to containerization or application dependencies at an early stage.

Together, these workflows automate essential validation steps, ensuring both high-quality code and reliable application builds throughout the development lifecycle.

Frontend error-handling

Astrocloud implements a robust error handling system to manage API responses effectively. The system uses two primary utility functions, HandleApiErrors and HandleObjectErrors, to process HTTP responses and throw or return errors based on the status codes. These errors are then caught in the frontend, where appropriate user feedback is provided via toast notifications, and users are navigated to relevant pages based on the error type.

Error Handling Utility Functions:

1. HandleApiErrors:

This function processes API responses and throws custom ApiError instances with tailored error messages based on the HTTP status code.

Usage:

This function is used usually in react queries and mutations to validate backend responses.

2-HandleObjectErrors:

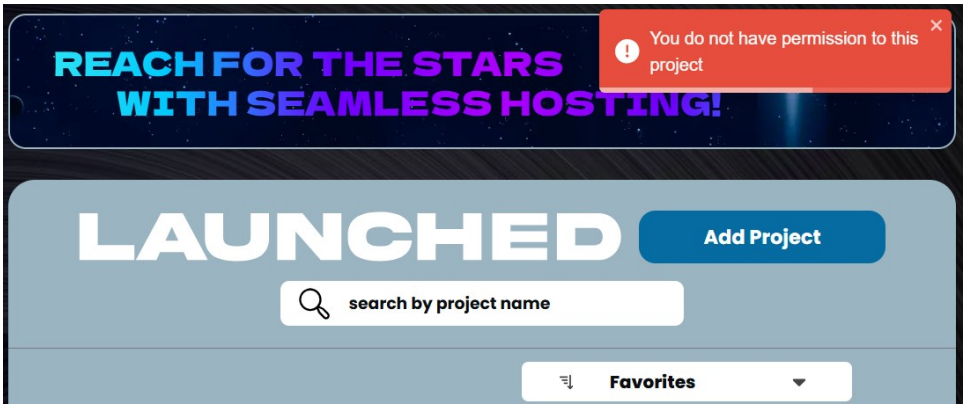
This function processes API responses and returns an error object instead of throwing an error, allowing for more flexible error handling in certain scenarios.

Usage:

This function is used usually in actions to validate response of form submition or user interaction to system.

Example:

User try to access forbidden project:



Frontend Optimizations

Initially, the frontend suffered from long load times due to a large JavaScript bundle and heavy static assets, impacting the user experience and performance. To address this, a series of key optimization measures were implemented:

1. **Converted Images to WebP Format:**
   All images were converted to the highly efficient WebP format, significantly reducing their file sizes and providing faster load times while preserving quality.
2. **Optimized WebP Images:**
   In addition to the format change, images were further optimized through compression techniques, minimizing their impact on page load and bandwidth usage.
3. **Preloaded Critical Assets:**
   Frequently used and large assets — including key JavaScript files and fonts — were preloaded, ensuring they were available as soon as needed, reducing delays in initial page rendering.
4. **Lazy Loading of Non-Critical Pages:**
   Pages and components not immediately required were configured for lazy loading, allowing the application to load only the necessary resources upfront and fetch others on demand.

Through these steps, the frontend load times were drastically improved, providing a smoother and more responsive user experience across the application.

Documentation Milestone

Landing Page & Documentation Website

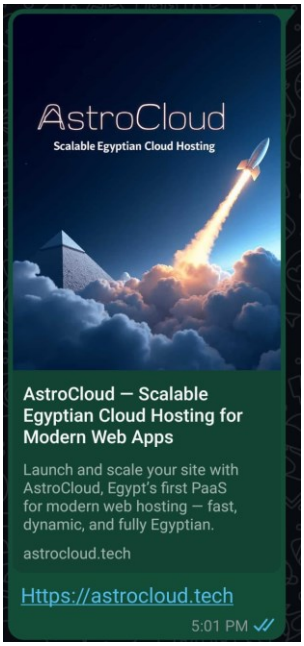
Overview

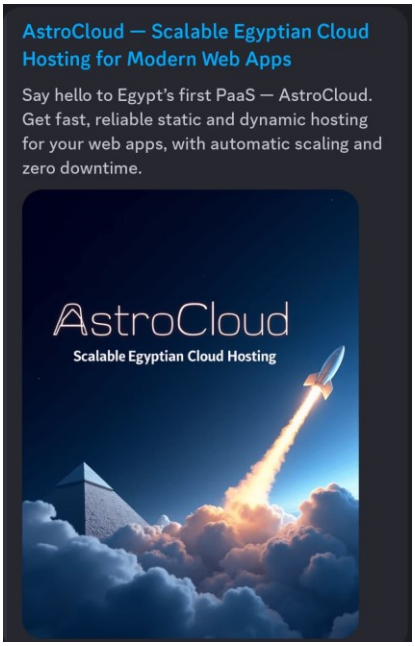
This landing website is the public-facing entry point of our hosting platform. It's designed using **Astro** to maximize SEO capabilities, especially since our main user dashboard is built with React (a Single Page Application), which can have limitations in SEO indexing.

The landing site provides a clear presentation of our platform's features, guides, and real project showcases. It includes visually appealing animations, clean navigation, and smooth scrolling across documentation sections.

Features

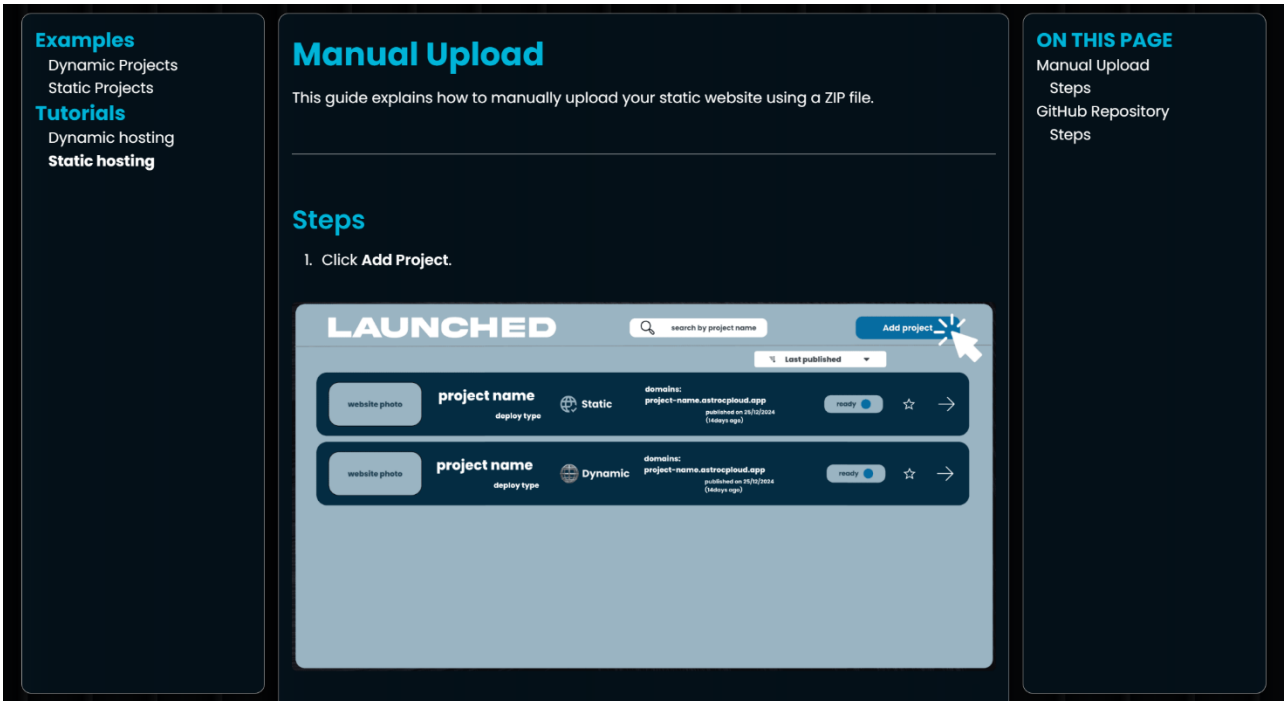
- **SEO-Optimized**:
  thanks to Astro's static rendering and metadata enhancements.





- **Static Hosting Guide**:
  This section provides clear and user-friendly guides for deploying static websites using two main methods:
  - **Manual Upload via ZIP**: Allows users to upload a compressed folder containing their static site files and deploy it with a custom subdomain.
  - **GitHub Integration**: Enables users to connect a GitHub repository, configure basic build settings, and deploy the project directly from source control.

  These tutorials are accessible through the documentation page and are designed to simplify the deployment process for users with varying levels of technical experience.



- **Dynamic Hosting Guide**:
  This section offers step-by-step guides for deploying dynamic applications using two supported methods:
  - **Manual Upload via ZIP**: Users can upload a ZIP file containing their backend application, configure runtime settings and environment variables through a build form, and deploy their project with ease.
  - **GitHub Integration**: Users can connect a GitHub repository containing their dynamic application, fill out the required runtime and environment configuration, and deploy directly from source control.

  The tutorials aim to simplify the deployment of dynamic services and provide a clear deployment flow from setup to real-time log tracking.
- **Documentation System** – Smooth MDX-powered navigation.
- **Real Project Examples**:
  This section showcases real-world examples of both static and dynamic projects deployed on the platform. It helps users understand how different types of applications can be hosted, configured, and deployed using the available tools and build options.
  - **Static Project Examples**:
    Includes frontend projects built with technologies such as **HTML/CSS**, **Vanilla JavaScript**, and **React.js**. Each example demonstrates how to configure the build process, define the output directory, and deploy the project using either ZIP upload or GitHub integration.
  - **Dynamic Project Examples**:
    Features backend applications developed using **Node.js**, **Express.js**, and similar technologies. These examples highlight runtime setup, environment variable configuration, and deployment of dynamic web applications and RESTful APIs on the platform.

Each example is accompanied by screenshots, build details, and GitHub repositories to provide practical reference for users deploying similar applications.

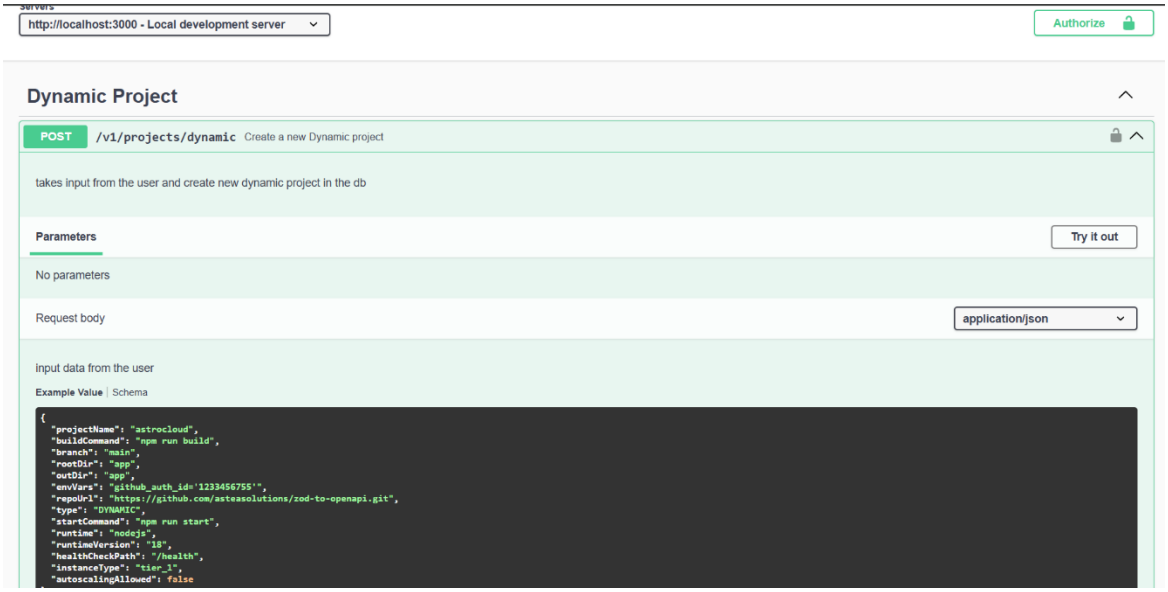
- **Responsive Animations & Styling**:
  The landing page features modern and responsive UI/UX enhanced with smooth scroll-based animations and dynamic background effects. These visual elements are implemented using SCSS and integrated with Tailwind to create a visually engaging and interactive experience. Animations include sliding, scaling, and floating effects triggered on scroll, along with a dynamic animated background that adds depth and motion to the layout — all while maintaining performance and accessibility across devices.

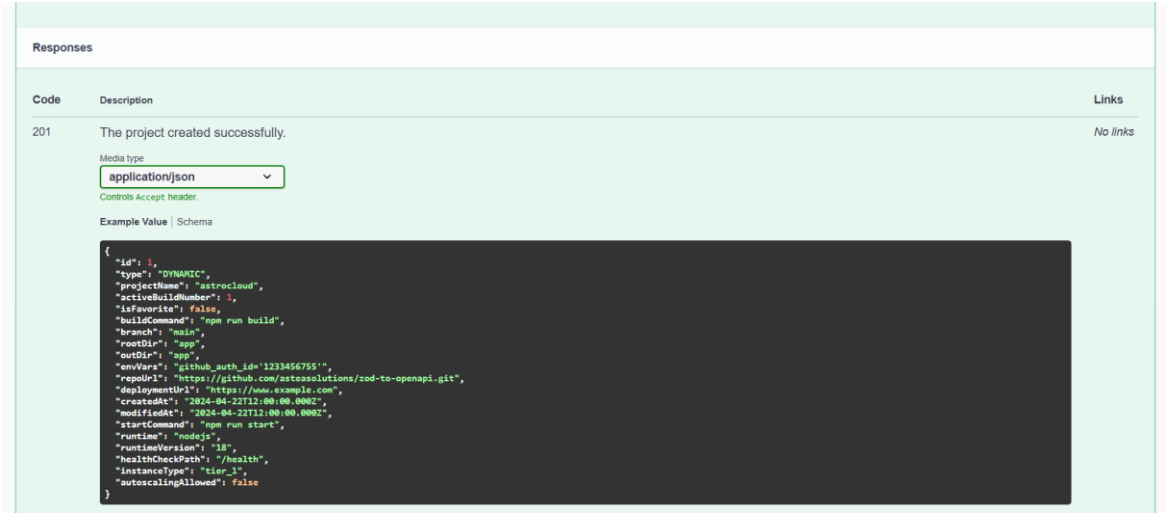
Backend OpenAPI Documentation

To ensure our API is well-documented, consistent, and automatically verifiable, we integrated **Zod** with **OpenAPI** using a toolchain like `zod-to-openapi`. This setup allows us to define validation schemas and API documentation in one unified place, reducing duplication and minimizing the risk of inconsistencies between the backend logic and the documentation.

Why Zod with OpenAPI?

- **Single Source of Truth:** By using Zod to define both the input validation and OpenAPI schema, we eliminate redundancy. The same Zod schema used to validate incoming requests is also used to generate OpenAPI-compliant documentation.
- **Automatic Swagger UI:** The OpenAPI spec generated from our Zod schemas powers the **Swagger UI**, which provides an interactive interface for developers to explore and test the API endpoints in real time.
- **Improved Developer Experience:** With ZodOpenApi, developers can:
    - See a live preview of all endpoints, request parameters, and response formats.
    - Understand required fields, expected types, and possible error codes.
    - Try endpoints directly from the browser.





Example in Context

As seen in the Swagger UI screenshots:

- The `POST /v1/projects/dynamic` endpoint allows users to create a new dynamic project.
- Both request body and response schema are fully defined using **Zod**.
- This schema is automatically translated to the OpenAPI spec, ensuring documentation always matches backend expectations.

Benefits for the Project

- **Fewer bugs** from mismatches between implementation and documentation.
- **Faster onboarding** for new developers due to accurate, self-updating docs.
- **Streamlined development**, as backend and docs evolve together with zero extra effort.

This approach ensures a clean, scalable, and developer-friendly API documentation strategy that evolves with the codebase.

How to Use Swagger UI

1. Open your browser and navigate to the Swagger UI URL: http://localhost:3000/v1/docs
2. At the top right, click the **"Authorize"** button.
3. A dialog will appear asking for a **Bearer Token**.
4. Paste your valid **JWT access token** (e.g., from login or a test user).
5. Click **"Authorize"** and then **"Close.**
6. Explore endpoints like:
   - `POST /v1/projects/dynamic` → for creating dynamic projects
   - `POST /v1/projects/static` → for creating static projects

User Dashboard Metadata

To enhance discoverability, social media integration, and accessibility of the web application, a comprehensive metadata strategy was implemented. This strategy consists of both global and dynamic metadata configurations.

Global Metadata (Static HTML Template)

Global metadata is defined in the main HTML template file (index.html), which is automatically included during the initial load of the React application. These settings ensure proper rendering across different devices and improve search engine indexing.

Key configurations include:

- **Font and Image Preloading**
- **Default Title and Description**

```
<title>AstroCloud – Scalable Egyptian Cloud Hosting for Modern Web Apps</title>
<meta
    name="description"
    content="Say hello to Egypt's first PaaS – AstroCloud. Get fast, reliable static and dynamic hosting for your web apps,
with automatic scaling and zero downtime."
/>
```

- **Author Tag**

```
<meta name="author" content="Astrocloud Team" />
```

- **Open Graph Tags (for Social Media Previews)**

```
<meta property="og:title" content="AstroCloud – Scalable Egyptian Cloud Hosting for Modern Web Apps"/>
<meta property="og:description" content="Say hello to Egypt's first PaaS – AstroCloud. Get fast, reliable static and dynamic
hosting for your web apps, with automatic scaling and zero downtime."/>
<meta property="og:image:width" content="768" />
<meta property="og:image:height" content="1024" />
<meta property="og:type" content="website" />
<meta prefix="og: http://ogp.me/ns#" property="og:title" content="AstroCloud – Scalable Egyptian Cloud Hosting for Modern Web
Apps"/>
<meta prefix="og: http://ogp.me/ns#" property="og:description" content="Launch and scale your site with AstroCloud, Egypt's
first PaaS for modern web hosting – fast, dynamic, and fully Egyptian."/>
<meta property="og:image" content="%VITE_FRONTEND_URL%/metaImages/metaImage.png"/>
<meta property="og:url" content="%VITE_FRONTEND_URL%" />
<meta prefix="og: http://ogp.me/ns#" property="og:image" content="%VITE_FRONTEND_URL%/metaImages/metaImage.png"/>
```

- **Twitter Metadata**

```
<meta name="twitter:card" content="summary_large_image" />
<meta name="twitter:title" content="AstroCloud – Scalable Egyptian Cloud Hosting for Modern Web Apps"/>
<meta name="twitter:description" content="Launch and scale your site with AstroCloud, Egypt's first PaaS for modern web
hosting – fast, dynamic, and fully Egyptian."/>
<meta name="twitter:image" content="%VITE_FRONTEND_URL%/metaImages/metaImage.png"/>
```

Dynamic Metadata (Per Page Configuration)

In addition to the global metadata, dynamic metadata is configured within individual pages of the React application. This ensures each page has contextually relevant metadata, which improves both SEO performance and user experience.

A reusable MetaData component is used to update metadata dynamically at runtime. This component accepts properties such as title and description and updates the document head accordingly.

```
const MetaData: React.FC<MetaDataProps> = ({ title = '', description = '' }) => {
    return (
        <Helmet>
            <title>{title}</title>
            <meta name="description" content={description} />
        </Helmet>
    );
};
```

Example Usage:

```
<MetaData
  title="Add New Project - Choose Import Method"
  description="Create a new project by selecting an import method: upload files, connect a GitHub repo, or use a template."
/>
```

This approach allows each route in the application to define:

- Custom page titles.
- Specific meta descriptions.

Note:

While the application uses metadata tags for enhancing social sharing and browser metadata (like titles and descriptions), it is important to note that as a Single Page Application (SPA), the content is rendered dynamically using JavaScript. As a result, standard search engine crawlers may not fully index page-specific content

Frontend and UI/UX Design

A polished UI (User Interface) and thoughtful UX (User Experience) can transform a site from “just live” to “high impact.” Even if your functionality is rock-solid under the hood, how it’s presented and flows for your users will determine whether they stick around, convert, and enthusiastically recommend you to others. Here are the keyways UI/UX on your deployed site makes a big difference:

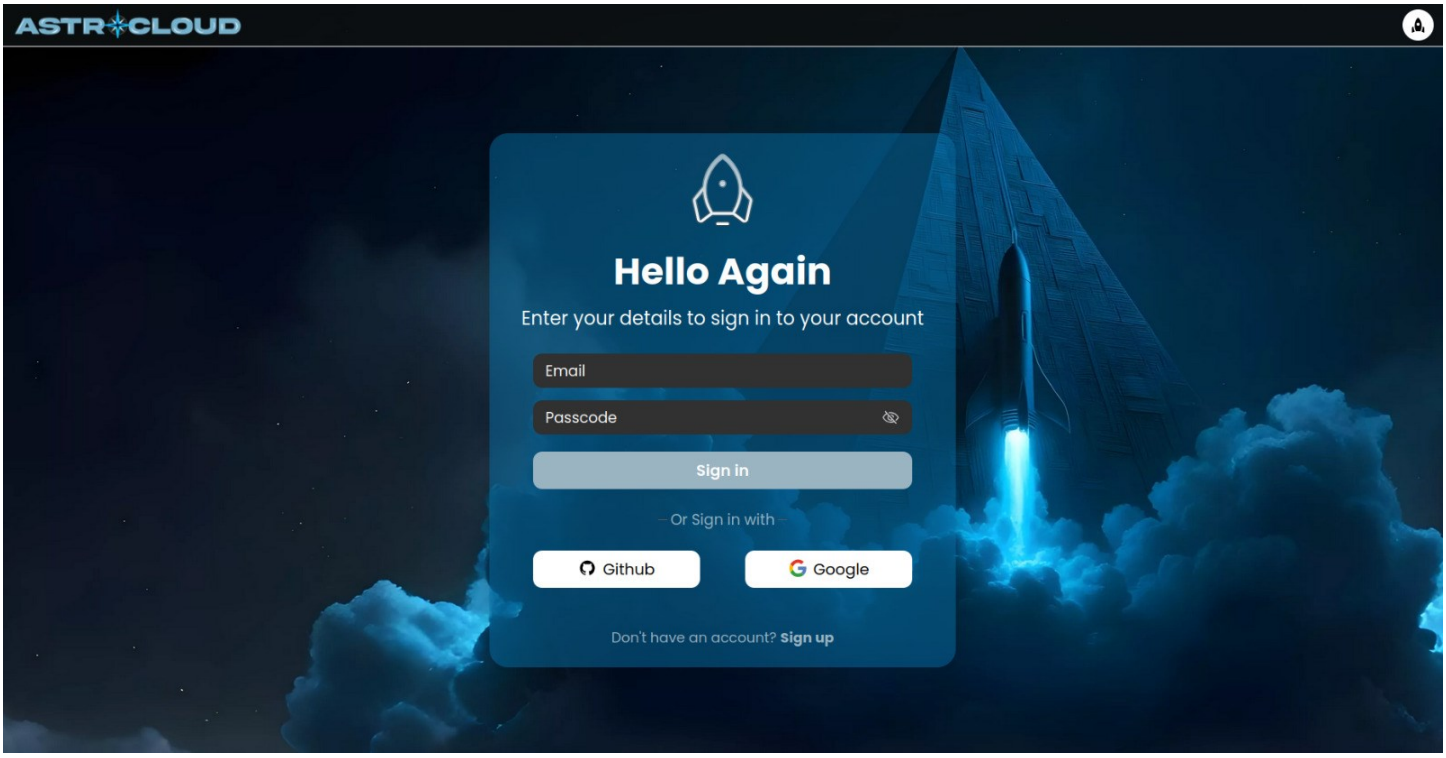
- First Impressions & Trust
    - Visual Consistency: A cohesive color palette, typography, and spacing immediately signal professionalism.
    - Clarity & Readability: Clean layouts and clear call-to-action buttons help users know what to do next—instilling confidence that your site is maintained and secure.

- Usability & Engagement
    - Intuitive Navigation: Well-structured menus, breadcrumbs, and “back” pathways keep users from getting lost. If people can’t find what they need in a few clicks, they’ll be bounced.
    - Feedback & Affordances: Hover states, loading indicators, and inline validation reassure users that actions are being processed correctly, reducing frustration.
- Conversion & Business Metrics
    - Focused UI Elements: Strategically placed buttons, forms, and prompts to guide users toward your goals—signups, purchases, downloads—without overwhelming them.
    - Streamlined Flows: Minimizing unnecessary steps (e.g. one-page checkouts, auto-fill forms) cuts friction and cart abandonment, directly boosting conversion rates.
- Accessibility & Inclusivity
    - Contrast & Legibility: Ensuring text meets WCAG contrast ratios and providing alternative text for images broadens your audience.
    - Keyboard & Screen-Reader Support: Proper HTML semantics and focus management let all users—regardless of ability—navigate and interact with your site.
    - Performance & Perceived Speed
    - Responsive UI: Adaptive layouts that load gracefully on mobile, tablet, and desktop foster engagement across devices.
    - Perceived Performance: Skeleton loaders, progressive image loading, and micro-animations give users a sense of responsiveness even when real loads take time.

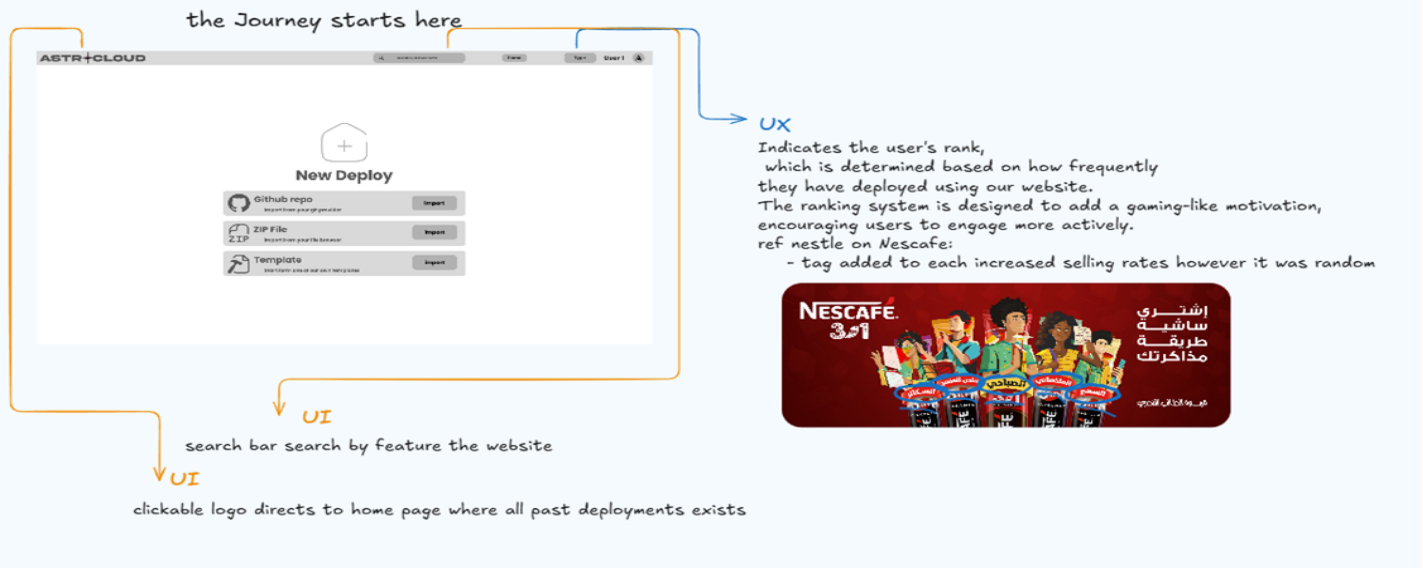
- Brand Differentiation & Loyalty
    - Personality & Tone: Thoughtful microcopy, iconography, and delight-driven animations help your brand voice shine through.
- Memorable Moments: Small touches—like custom 404 pages or playful button animations—create emotional connections that keep people coming back.

Login page:

Final design:

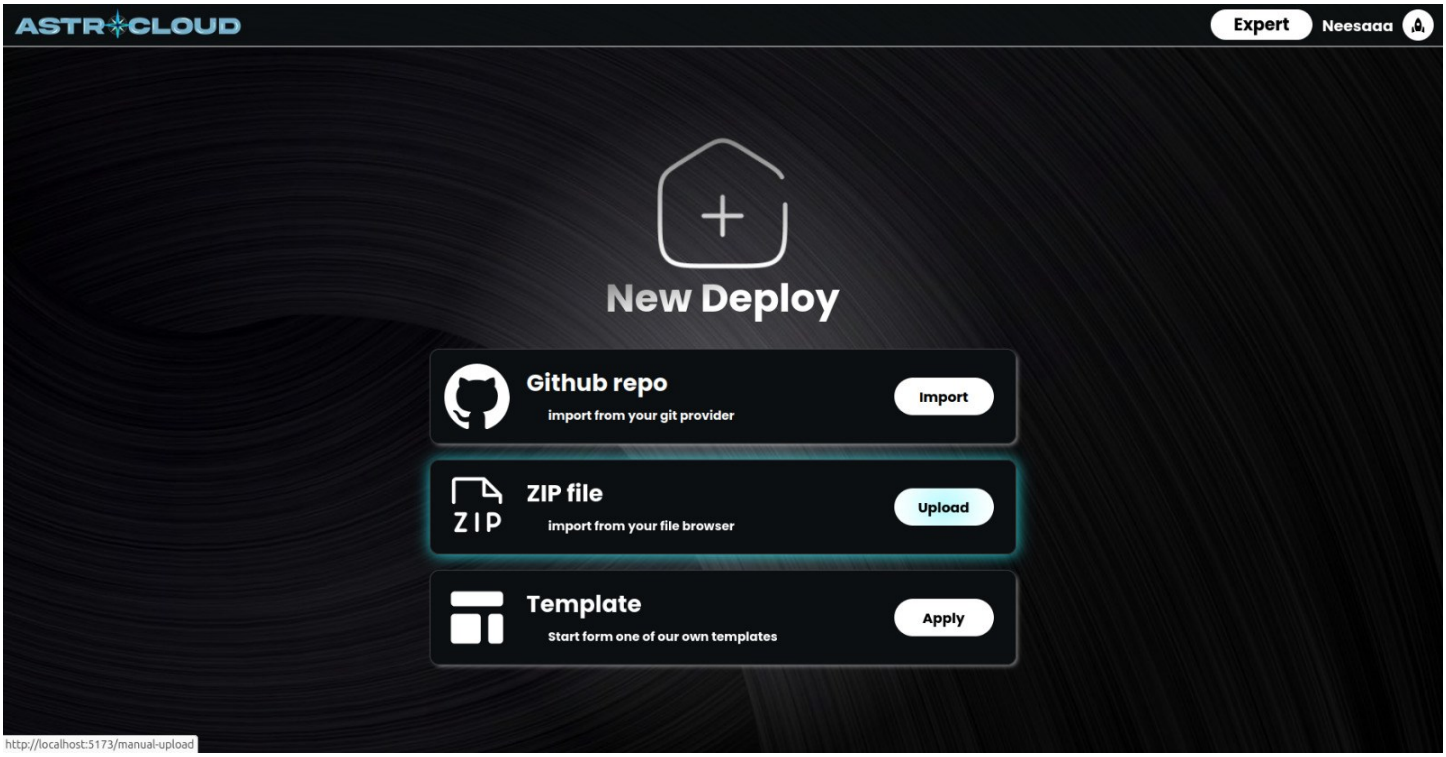


Add project page:

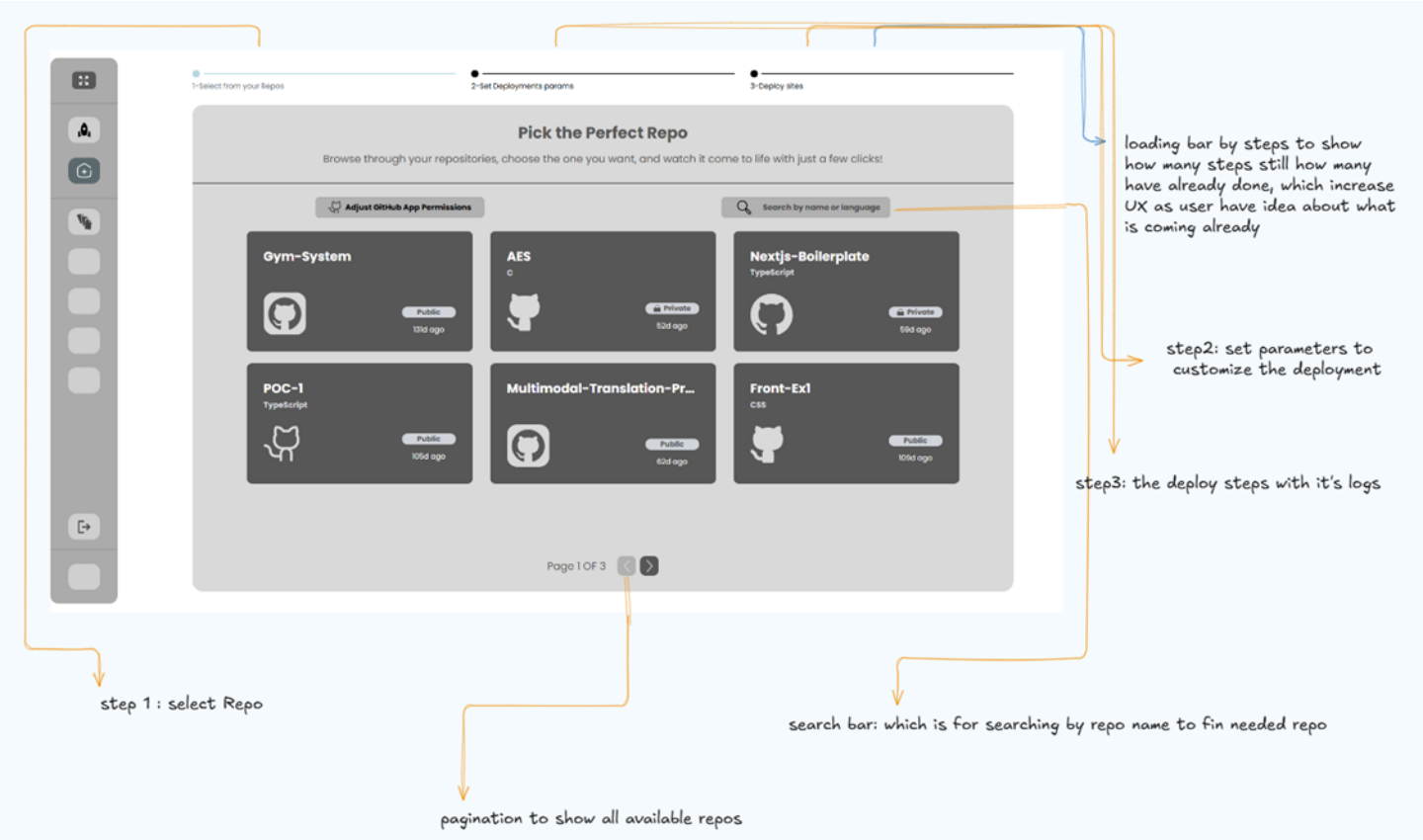

the Journey starts here
ASTR+CLOUD
New Deploy
Github repo
Import
ZIP File
Import
Template
Import
UX
Indicates the user's rank,
which is determined based on how frequently
they have deployed using our website.
The ranking system is designed to add a gaming-like motivation,
encouraging users to engage more actively.
ref nestle on Nescafe:
- tag added to each increased selling rates however it was random
NESCAFE
3in1
إشتري
ساشيه
طريقة
مذاكرتك
UI
search bar search by feature the website
UI
clickable logo directs to home page where all past deployments exists

---

Final:


ASTR+CLOUD
Expert
Neesaaa
New Deploy
Github repo
import from your git provider
Import
ZIP file
import from your file browser
Upload
Template
Start form one of our own templates
Apply
http://localhost:5173/manual-upload

Deploy by Github:

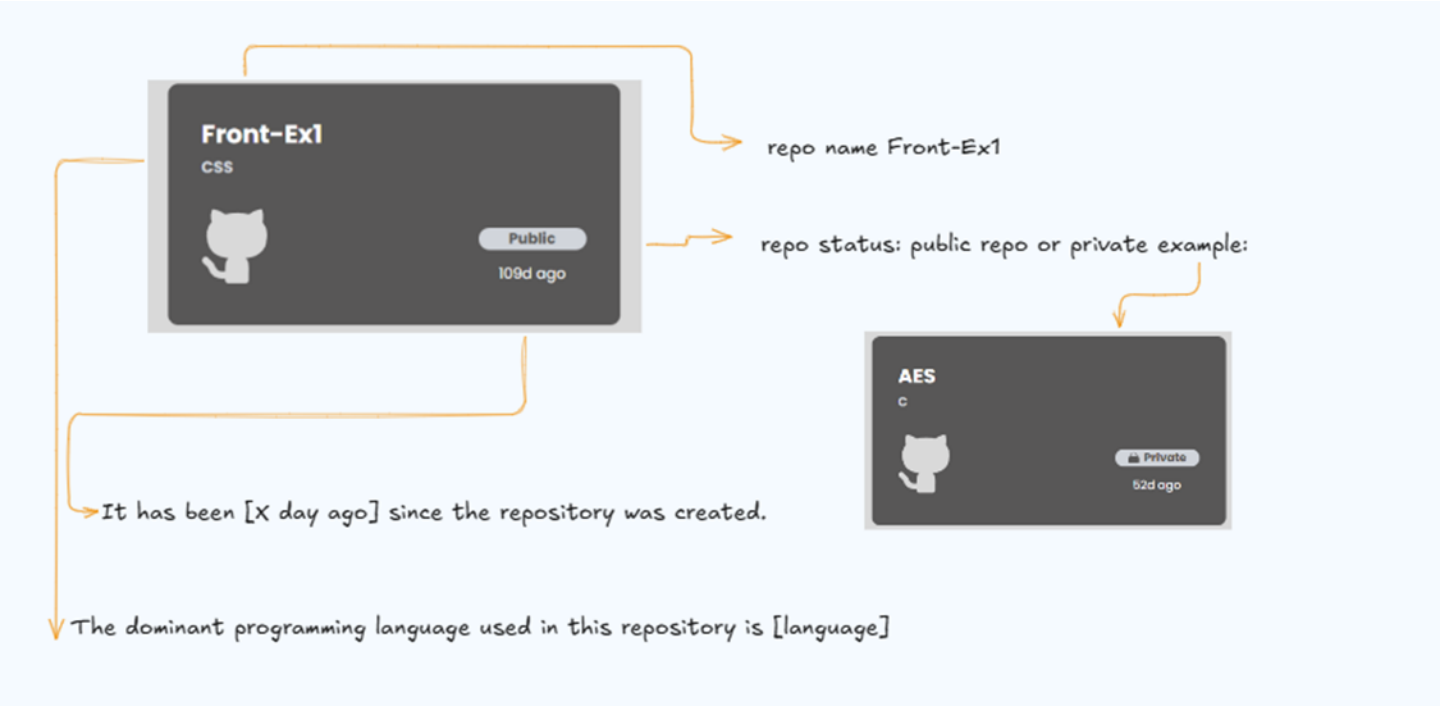


Selecting the repository from a **repos must-have page** rather than just a **dropdown menu** offers several advantages:

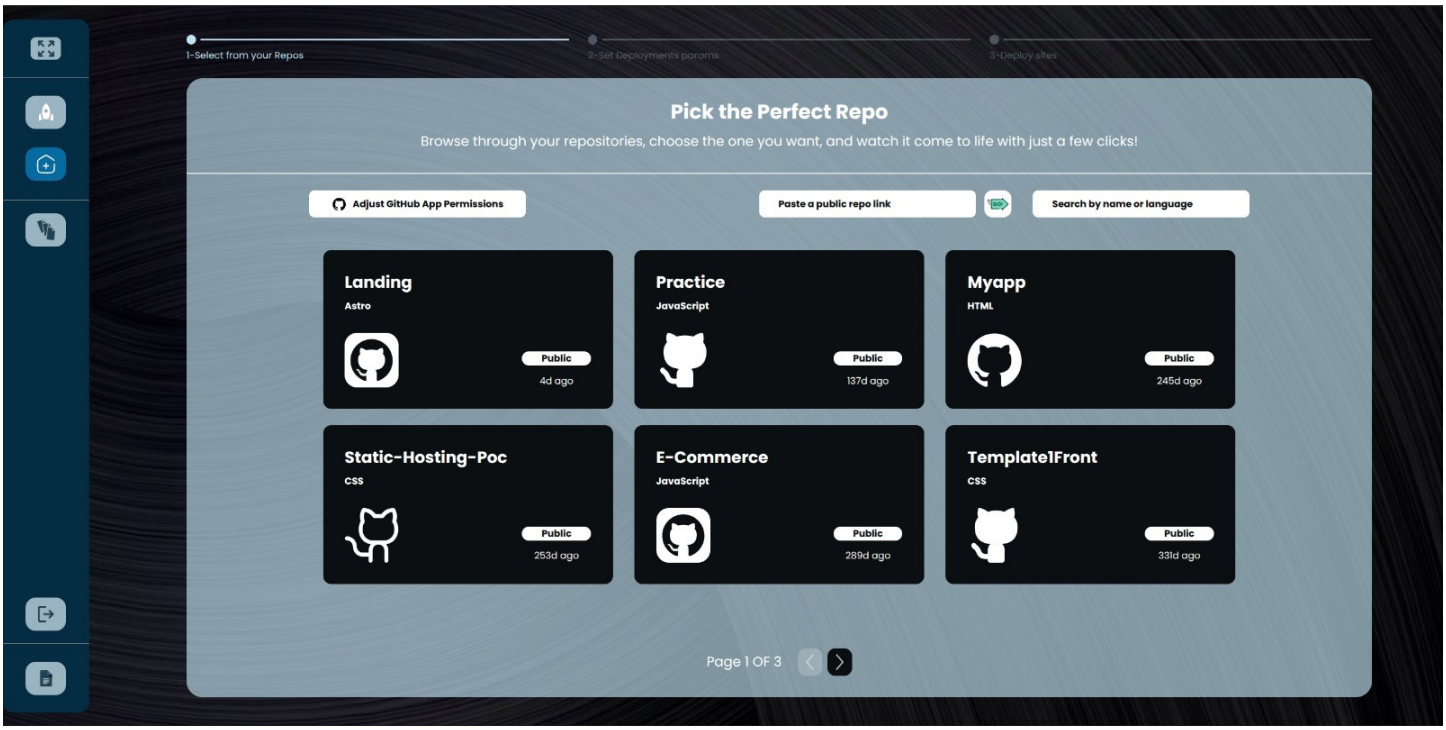
1. **Better Organization**: A dedicated page provides a clearer view of all available repositories, especially for users with multiple repos. It allows for easy browsing, filtering, and searching, helping users quickly locate the correct repository without scrolling through a long dropdown list.
2. **Improved Usability**: With a repository page, you can present more information, such as repository names, descriptions, last update times, and statuses. This helps users make a more informed choice without needing to guess which repository to select.
3. **Scalability**: As the number of repositories grows, a dropdown menu can become unwieldy and harder to navigate. A dedicated page scales better and ensures the selection process remains efficient as more repositories are added.
4. **Enhanced User Experience**: A page with repositories can include sorting, categorizing, and searching features, allowing users to interact with the content more easily. This approach gives users more control over their selection, leading to a smoother experience overall.

In short, using a **repos must-have page** enhances clarity, usability, and scalability, providing a more intuitive and organized way to choose a repository for deployment.

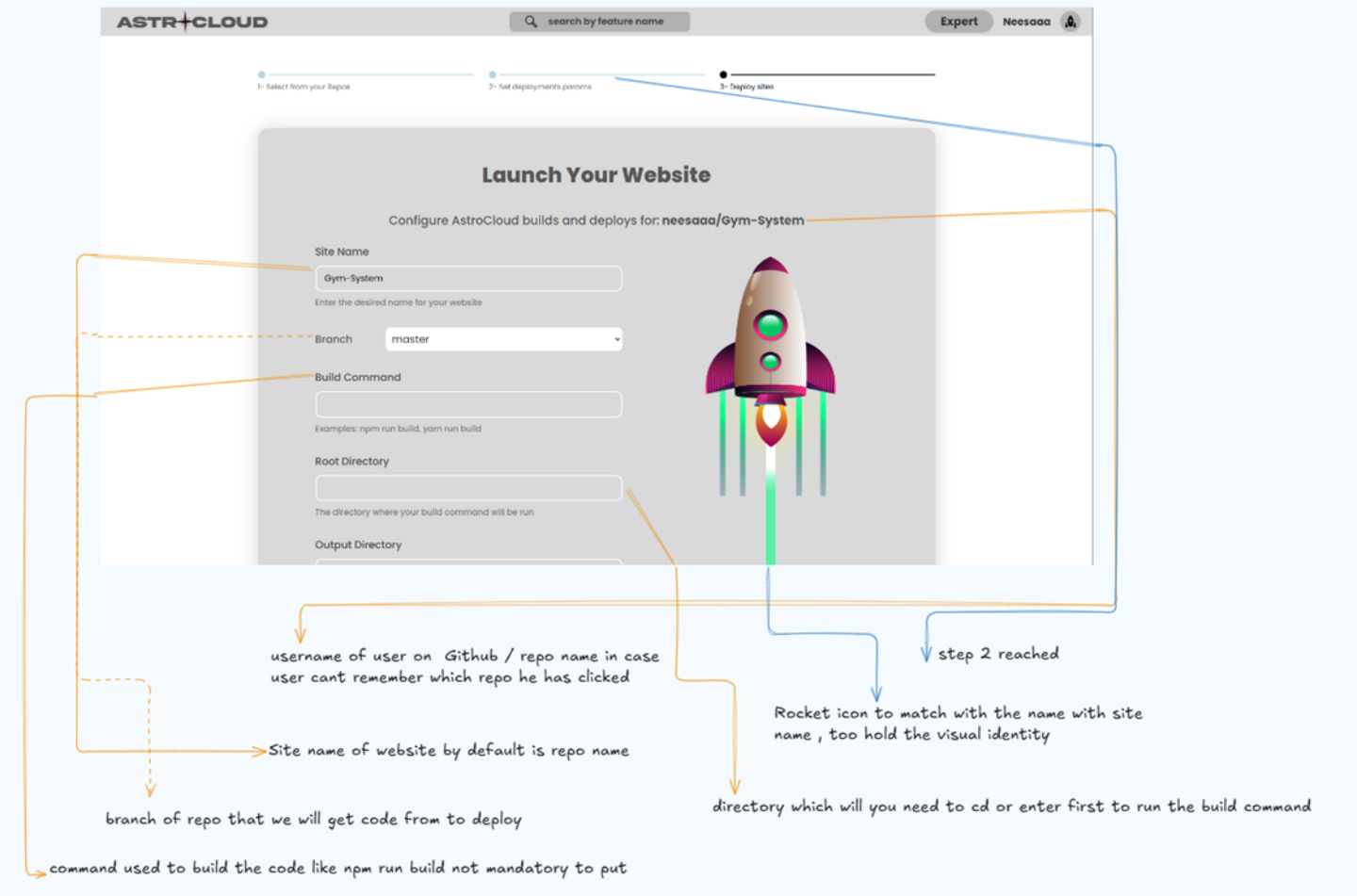
Each repo has card:



Final:



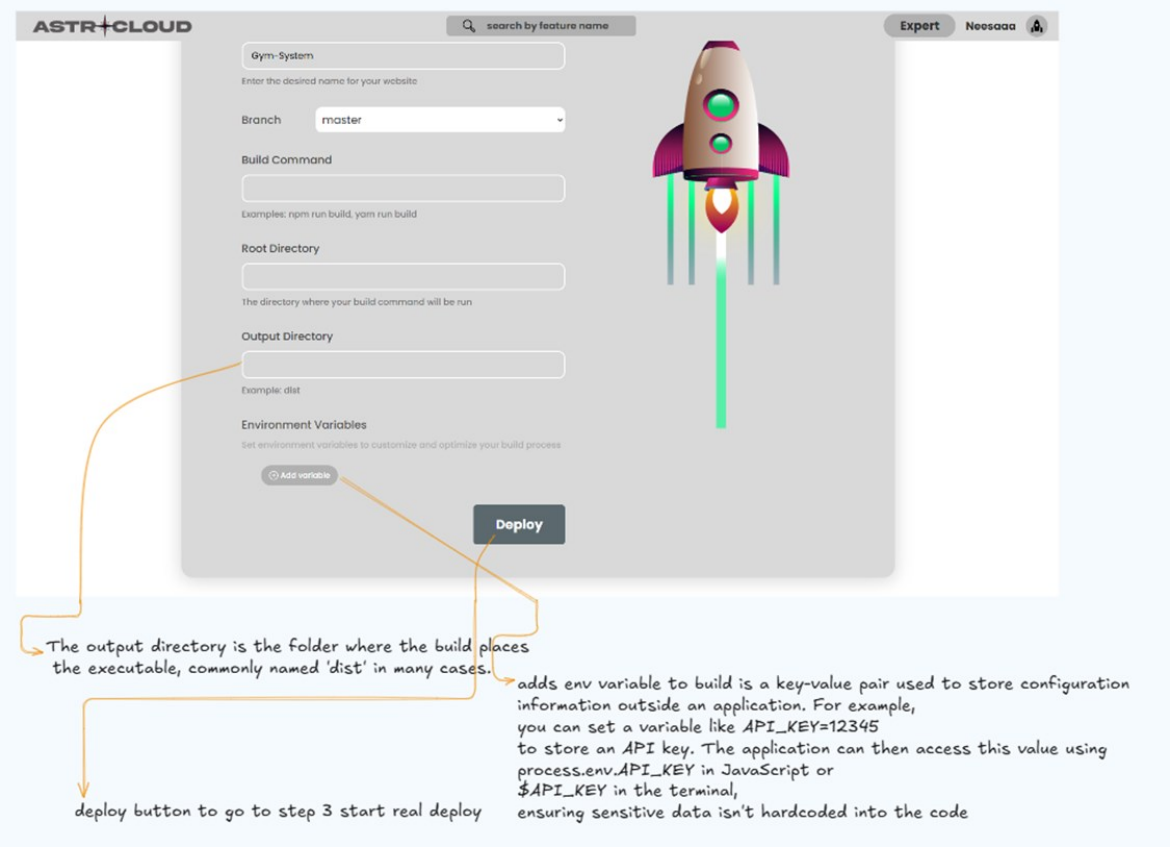
Now suppose we clicked specific repo:



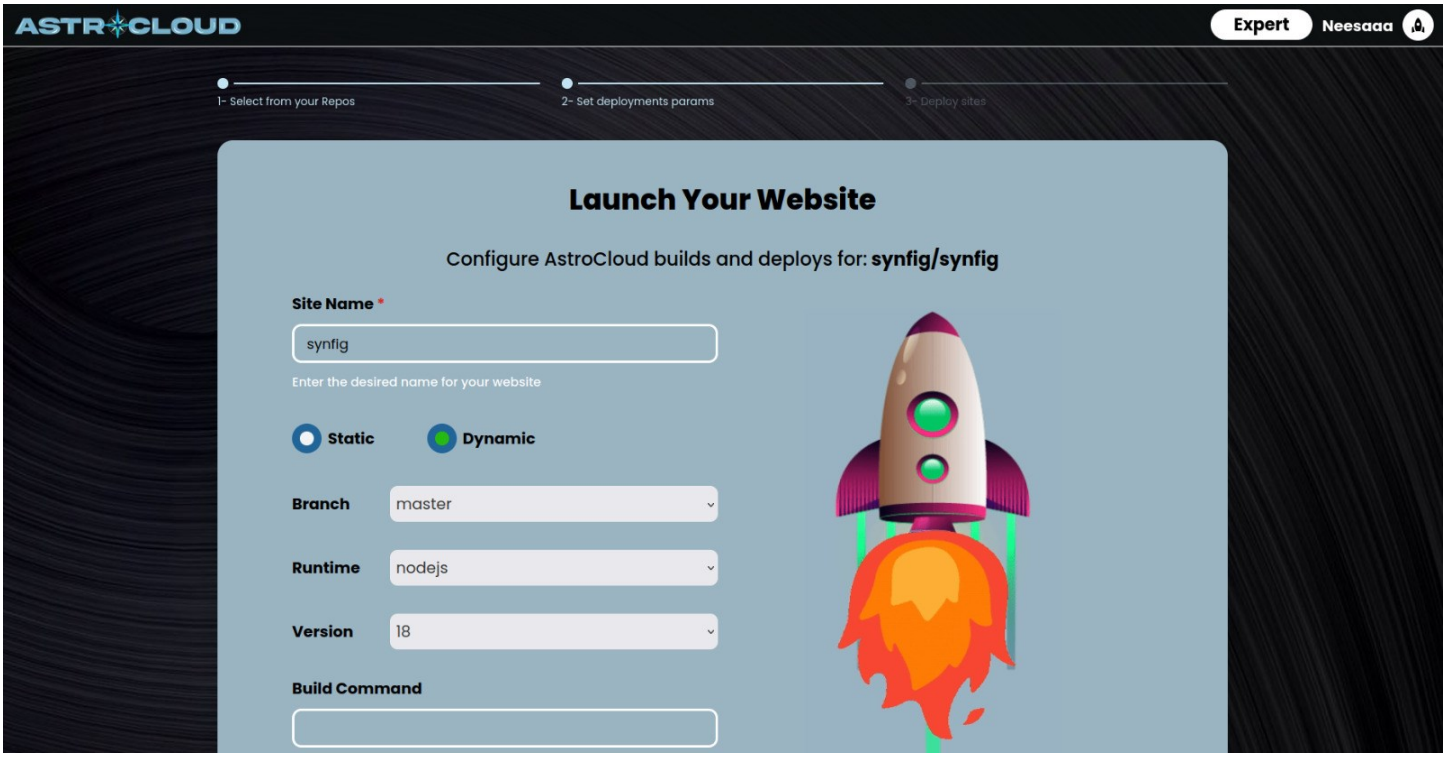
The form parameters like **command**, **output folder**, **site name**, and **environment variables** are crucial to fill out before deploying a site for several reasons:

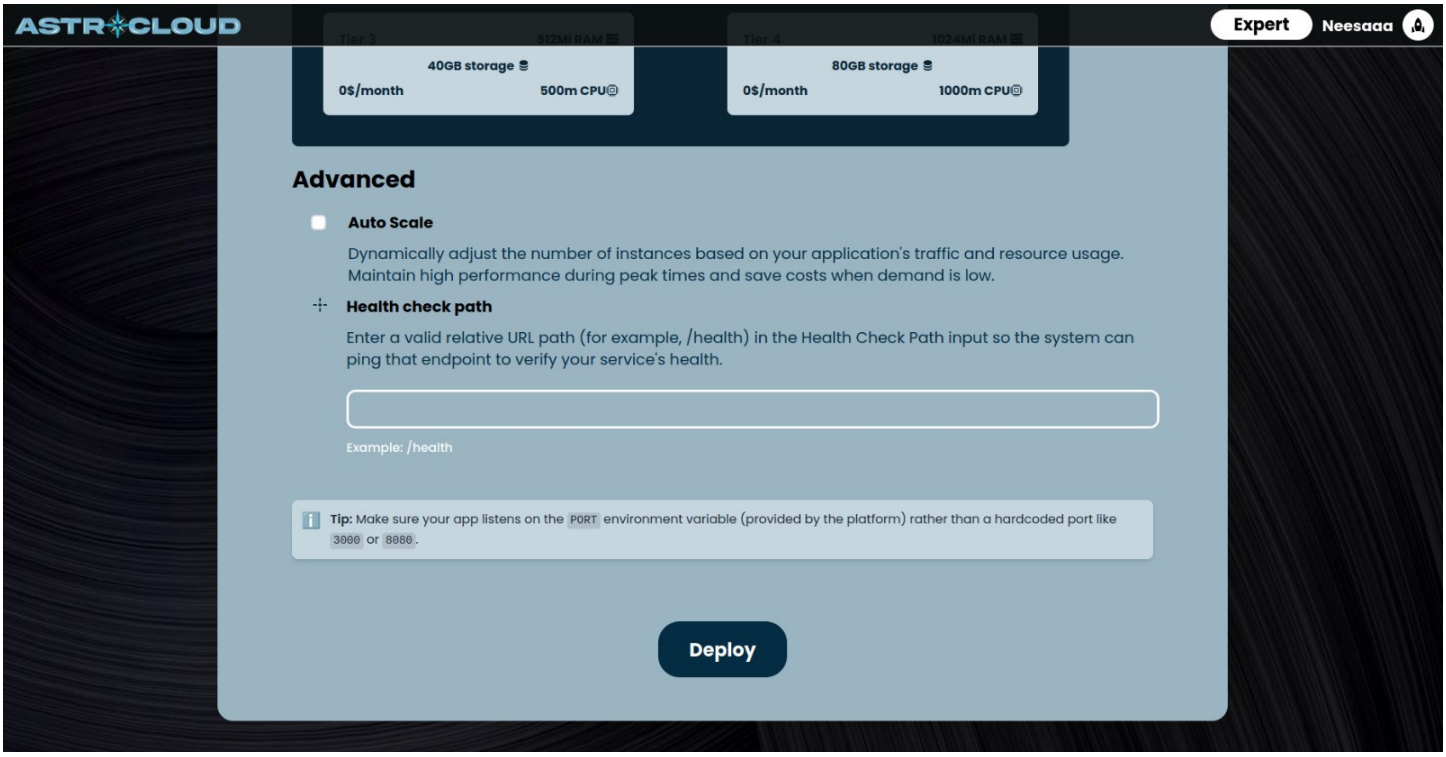
1. **Command**: The correct command ensures that the build or deployment process runs smoothly. It specifies the actions or scripts that need to be executed during deployment, like building the project or starting the server.
2. **Output Folder**: The output folder defines where the final build files, including the executable or site assets, will be stored. Without specifying this, the deployment system won't know where to find the necessary files to serve or distribute.
3. **Site Name**: The site name helps in identifying and referencing the deployed site. It's essential for configuring domain names, creating unique URLs, and managing different environments (e.g., development, staging, production).
4. **Environment Variables**: These store configuration values such as database URLs, API keys, and other sensitive information. Filling them out ensures that the deployed site has access to necessary resources and remains flexible across different environments without hardcoding sensitive data.

Filling these parameters ensures that the deployment is automated, correct, and secure, making the process more efficient and minimizing the chances of errors.


ASTRCLOUD
search by feature name
Expert
Neesaaa
Gym-System
Enter the desired name for your website
Branch
master
Build Command
Examples: npm run build, yarn run build
Root Directory
The directory where your build command will be run
Output Directory
Example: dist
Environment Variables
Set environment variables to customize and optimize your build process
Add variable
Deploy
The output directory is the folder where the build places the executable, commonly named 'dist' in many cases.
adds env variable to build is a key-value pair used to store configuration information outside an application. For example, you can set a variable like API_KEY=12345 to store an API key. The application can then access this value using process.env.API_KEY in JavaScript or $API_KEY in the terminal, ensuring sensitive data isn't hardcoded into the code
deploy button to go to step 3 start real deploy

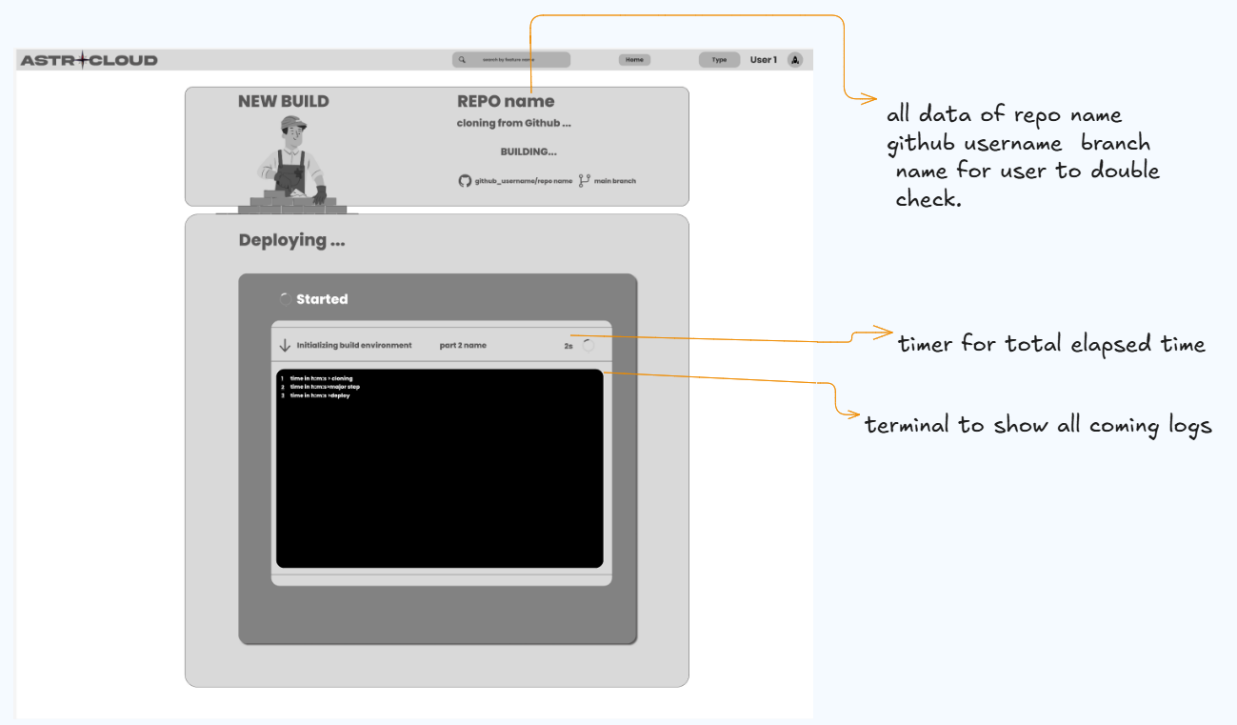
Final:


ASTRCLOUD
Expert
Neesaaa
1- Select from your Repos
2- Set deployments params
3- Deploy sites
Launch Your Website
Configure AstroCloud builds and deploys for: synfig/synfig
Site Name *
synfig
Enter the desired name for your website
Static
Dynamic
Branch
master
Runtime
nodejs
Version
18
Build Command



Now suppose we clicked Deploy button:

- We then go to log streaming page which shows logs live for the whole process

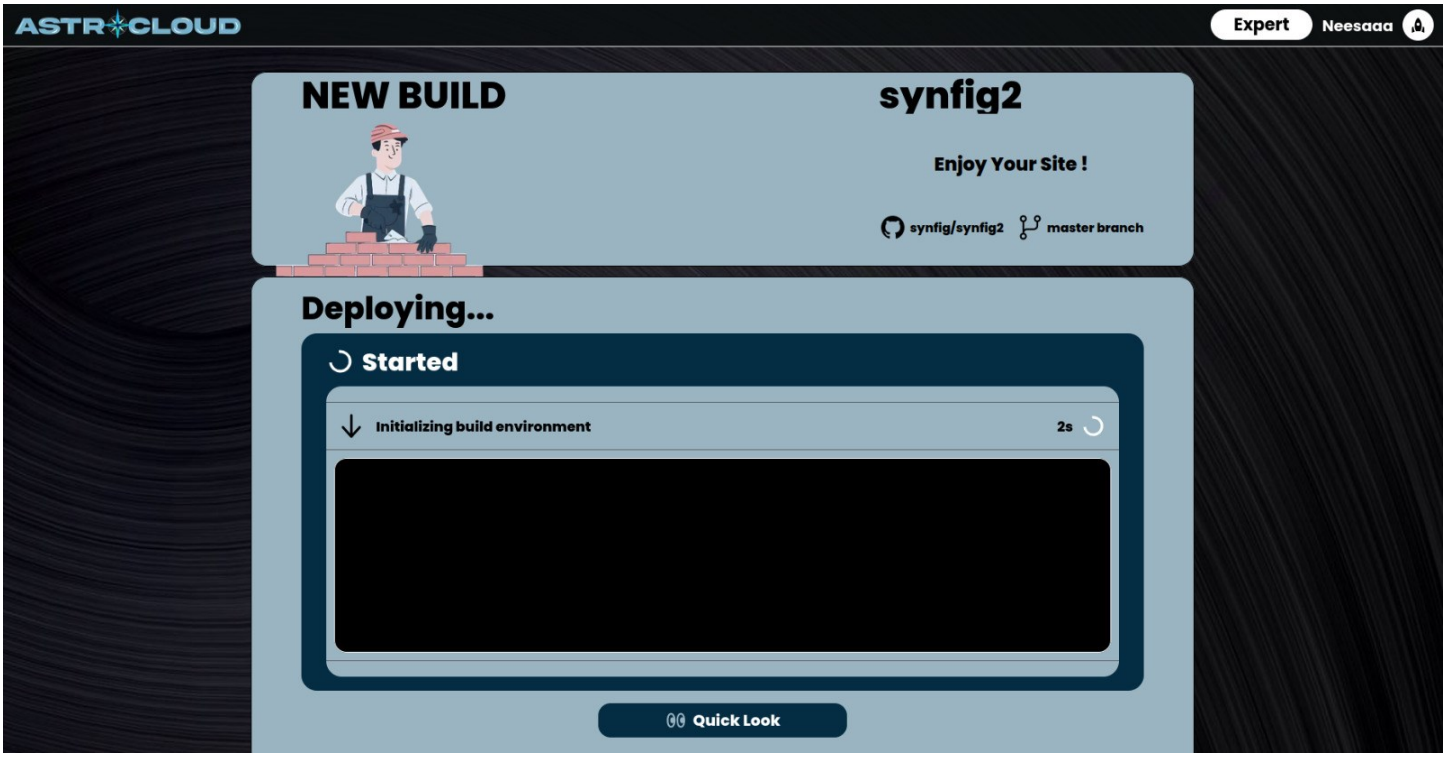


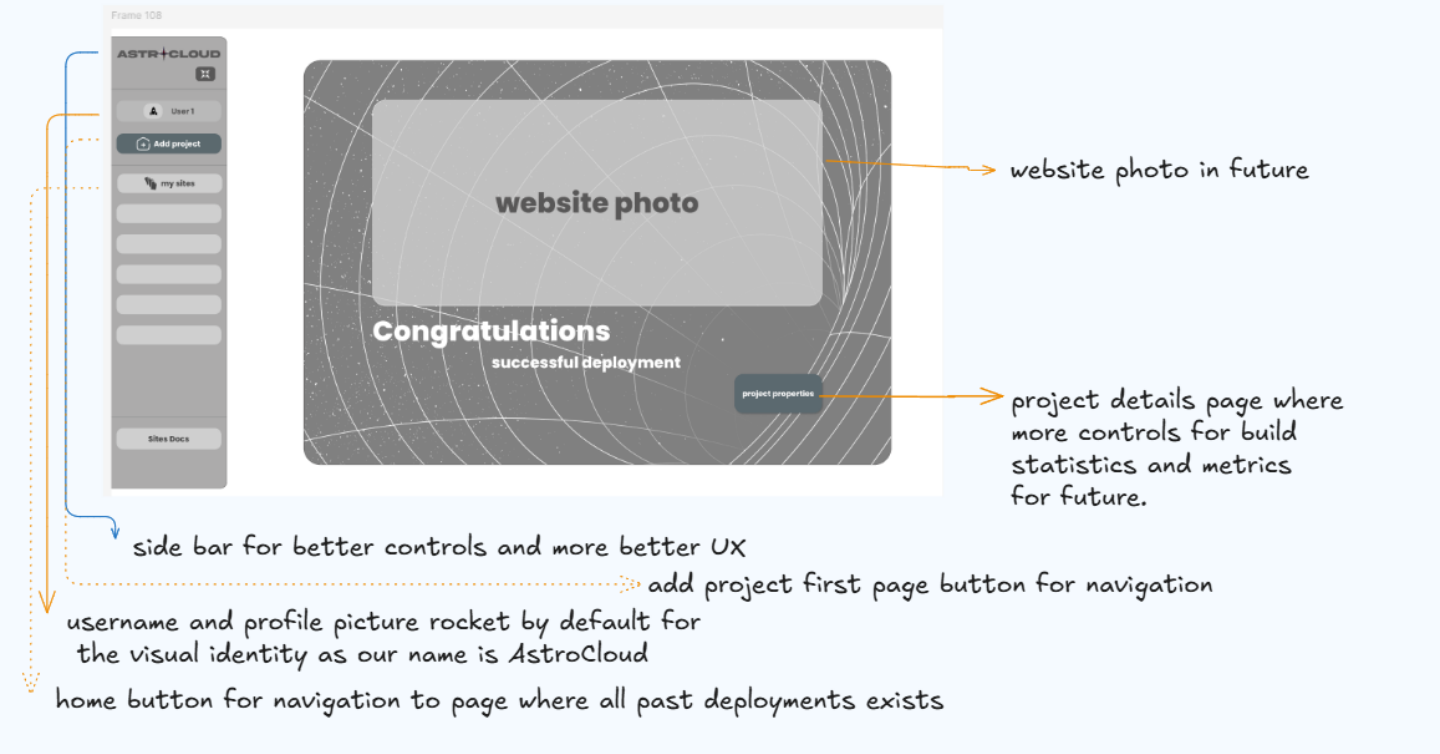
Having a **terminal showing logs** during site deployment is essential for UI and UX for several reasons:

1. **Transparency and Feedback**: The terminal provides real-time feedback to the user, showing exactly what is happening during deployment. This builds trust, as users can see the progress and status of each step, instead of waiting blindly.
2. **Error Diagnosis**: If something goes wrong during deployment, the logs help users identify the issue quickly. They can see error messages or warnings directly, which empowers them to troubleshoot without unnecessary delays.
3. **Control and Confidence**: Watching logs in real-time gives users a sense of control and reassurance. They can confirm that the deployment is progressing correctly or pause to address any unexpected behavior if needed.
4. **Professional Feel**: A terminal-like interface is often associated with technical expertise. For experienced users, it reinforces confidence in the platform, while for beginners, it creates an impression of professionalism and reliability.
5. **Enhanced UX**: Instead of navigating to a separate logs page or waiting for an email, users can monitor everything on the same interface. This seamless integration minimizes context switching and provides a cohesive experience.

In summary, a terminal showing logs enhances user trust, aids in error handling, improves the feeling of control, and delivers a smooth, professional deployment experience.

Final:





Then till building is complete a transition for success page is done:

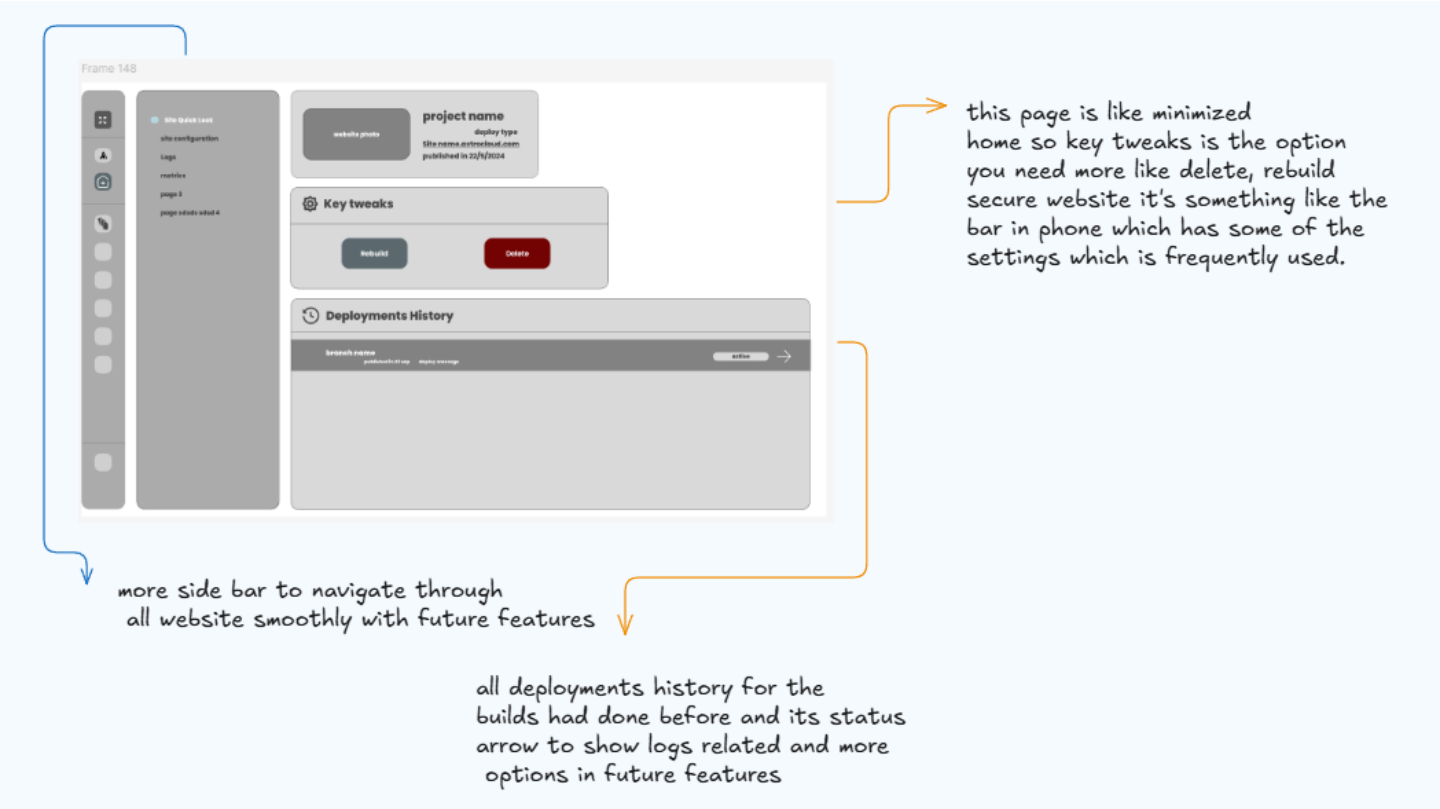
1. Successful Page

- **Confirmation and Satisfaction**: A dedicated success page reassures users that their deployment or action was completed successfully, reducing ambiguity.
- **Opportunity for Next Steps**: It can suggest follow-up actions, like viewing the deployed site, sharing a link, or accessing logs, improving user engagement.
- **Clear Communication**: Highlighting key details (e.g., deployment status, time taken, site URL) makes it easy for users to understand the result.
- **Celebratory Touch**: Adding visuals or animations (e.g., checkmarks or confetti) creates a positive emotional response, enhancing user satisfaction.

2. Sidebar with More Controls

- **Quick Navigation**: A sidebar provides easy access to frequently used features or sections, reducing the need to switch pages.
- **Contextual Actions**: Users can perform related tasks (e.g., manage repos, adjust settings, view logs) without leaving the current screen, streamlining workflows.
- **Scalability**: Sidebars can grow with additional features or tools without cluttering the main interface.
- **Customization**: A sidebar allows users to personalize their experience by pinning or rearranging controls for convenience.

Suppose we clicked project details button:



An **overview page** for controlling builds and displaying all previous builds is essential for the following reasons:

1. Centralized Management

- The overview page acts as a central hub where users can view, manage, and control all their builds in one place.
- It eliminates the need to navigate multiple sections to find relevant build information, improving efficiency.

2. Visibility of Build History

- Displaying all previous builds helps users track the progress and status (e.g., success, failure) of past deployments.
- Users can refer to past builds for troubleshooting, debugging, or understanding trends over time.

3. Control and Convenience

- The page provides direct controls to **restart**, **cancel**, or **rebuild** a specific deployment.
- It simplifies workflows by allowing users to manage builds without accessing separate tools or pages.

4. Error Handling and Debugging

- With a history of builds, users can pinpoint when and where issues occurred. Logs and details associated with each build make it easier to debug and resolve errors.

5. Accountability and Auditability

- For team-based environments, the overview page serves as a record of who initiated which build and when.
- It provides transparency and accountability, which are essential for collaboration and project management.

6. Scalability

- As the number of builds grows, an overview page can incorporate filtering, sorting, and searching options, allowing users to quickly locate specific builds or projects.

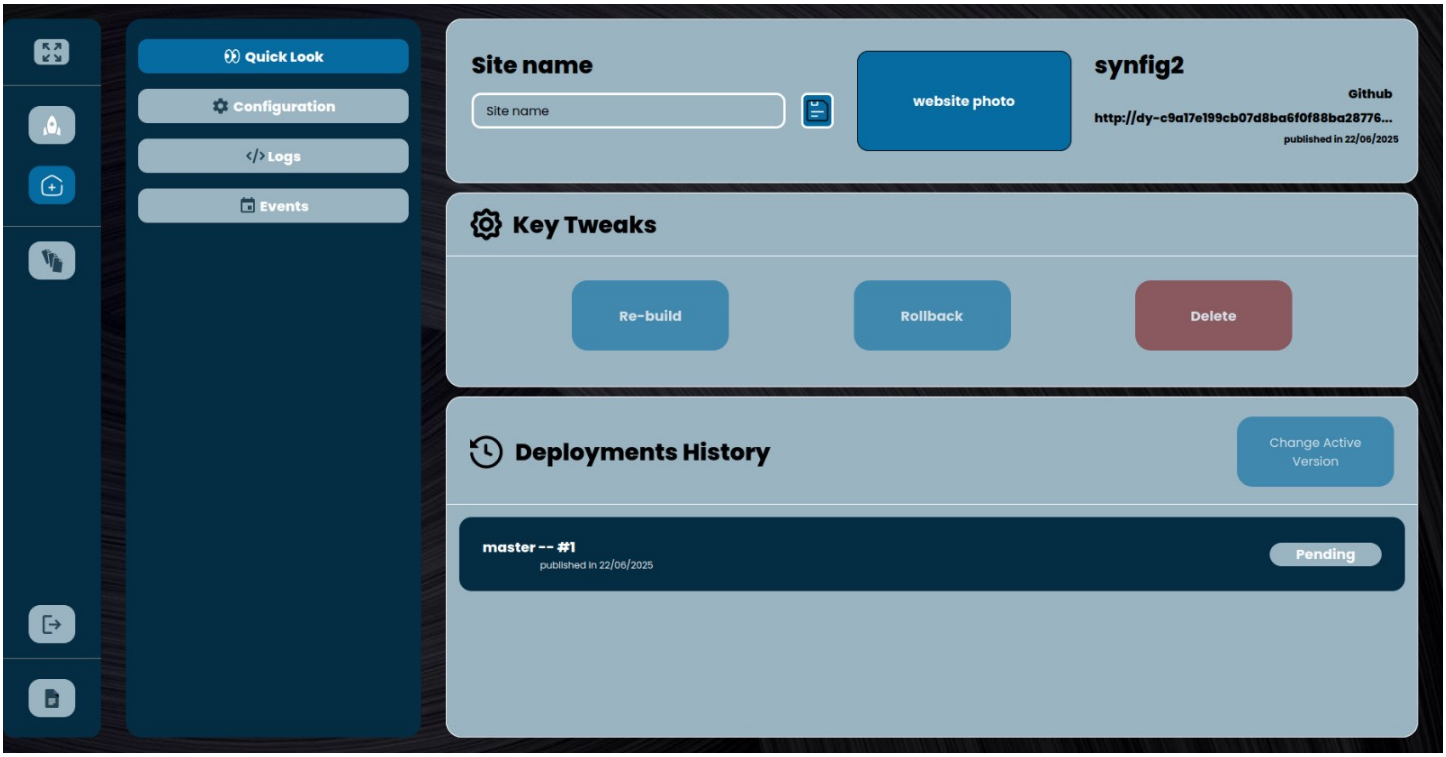
7. Improved UX

- A well-designed overview page offers a clear snapshot of the system's activity, helping users understand the current state of their projects at a glance.
- It ensures that important information like deployment times, errors, and durations are easily accessible.

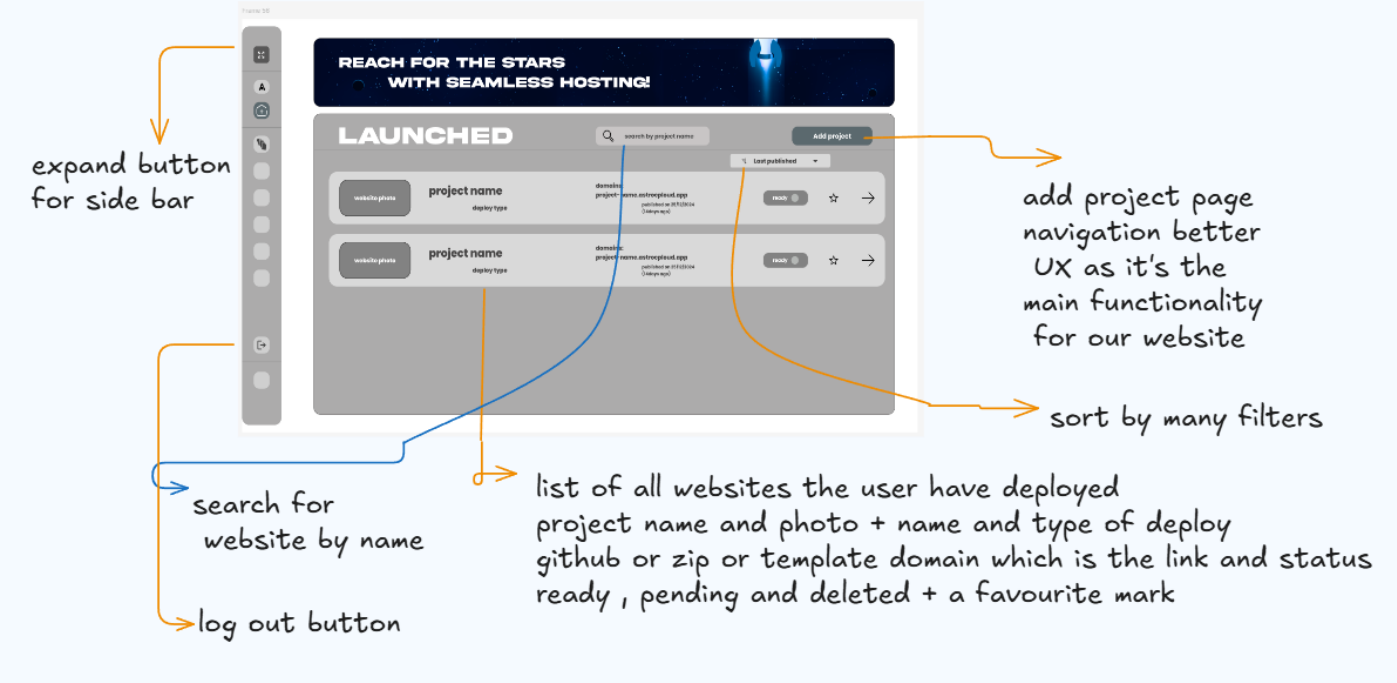
In summary, the overview page is crucial for managing builds effectively, maintaining transparency, and providing a seamless user experience. It simplifies build management and ensures users have access to all the tools and information they need in one place

---

Final:



Then our home page which has button my sites inside bar:



Home page just shows previous work and banner to make the user sure we are confident with our website

Separating each collection of related buttons in the **sidebar** improves **user understanding** and enhances the **UX** for the following reasons:

1. Logical Grouping for Clarity

- By grouping related buttons (e.g., "Build Controls," "Deployment Settings," "Logs and History"), users can easily locate the tools they need.
- Logical separation reduces cognitive load, making it easier for users to understand the interface at a glance.

2. Better Navigation Flow

- Organized sections create a clear navigation structure. Users can quickly access relevant controls without confusion or unnecessary searching.
- For instance, users know where to look for "Repo Settings" versus "Deployment Actions" instantly.

3. Enhanced Visual Hierarchy

- Separating buttons visually distinguishes their purpose, helping users intuitively understand how to interact with the sidebar.
- Headers or dividers between groups reinforce the hierarchy, guiding the user through the interface step by step.

4. Scalability

- When the application grows and more features are added, a grouped sidebar remains clean and manageable.
- New features can be added under the appropriate section without cluttering or confusing users.

5. Improved Focus and Usability

- Users are less likely to be overwhelmed when related buttons are grouped together.
- Focused grouping minimizes distractions and makes tasks faster by narrowing down choices in each section.

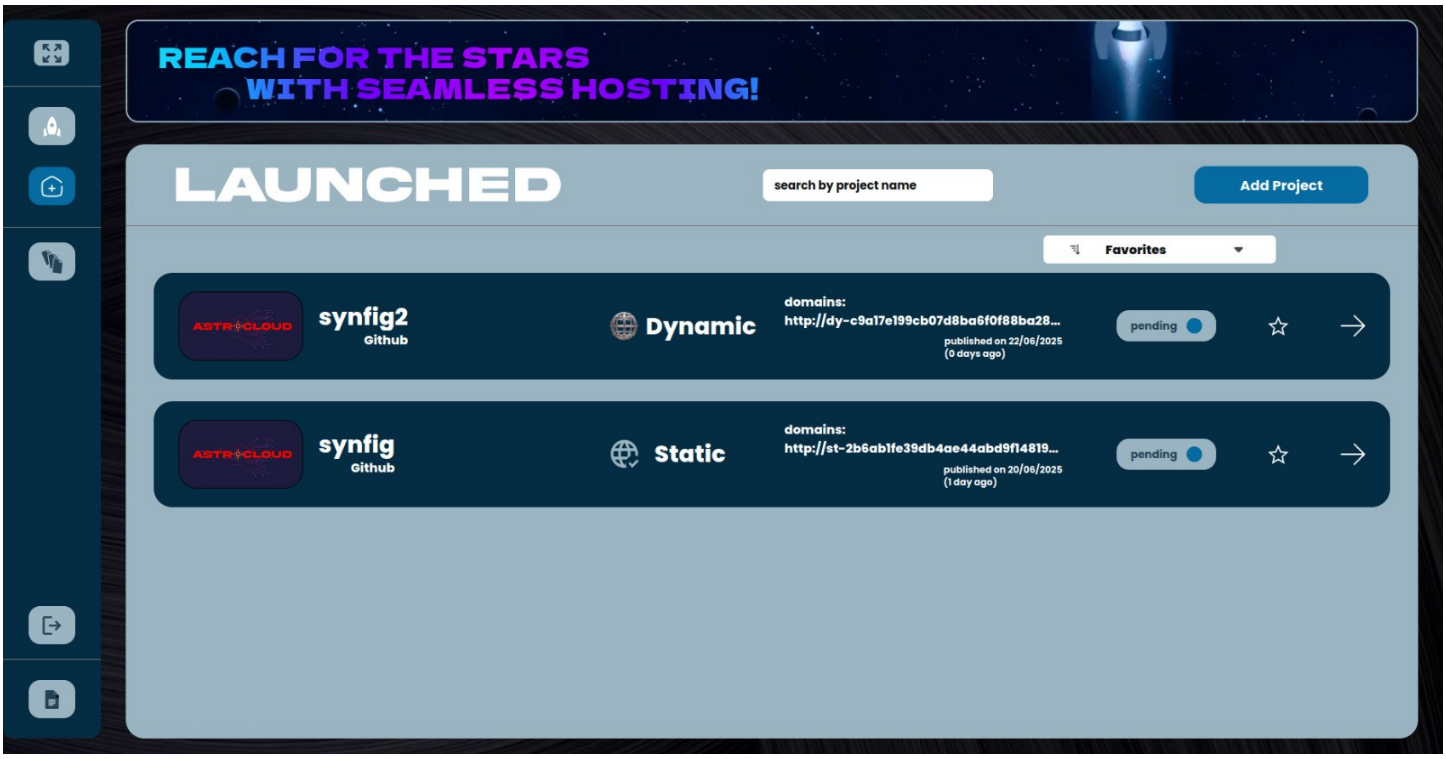
6. Customizability

- With clearly separated sections, future enhancements like collapsible groups allow users to hide or expand sections as needed.
- This personalization enhances usability for both advanced and beginner users.

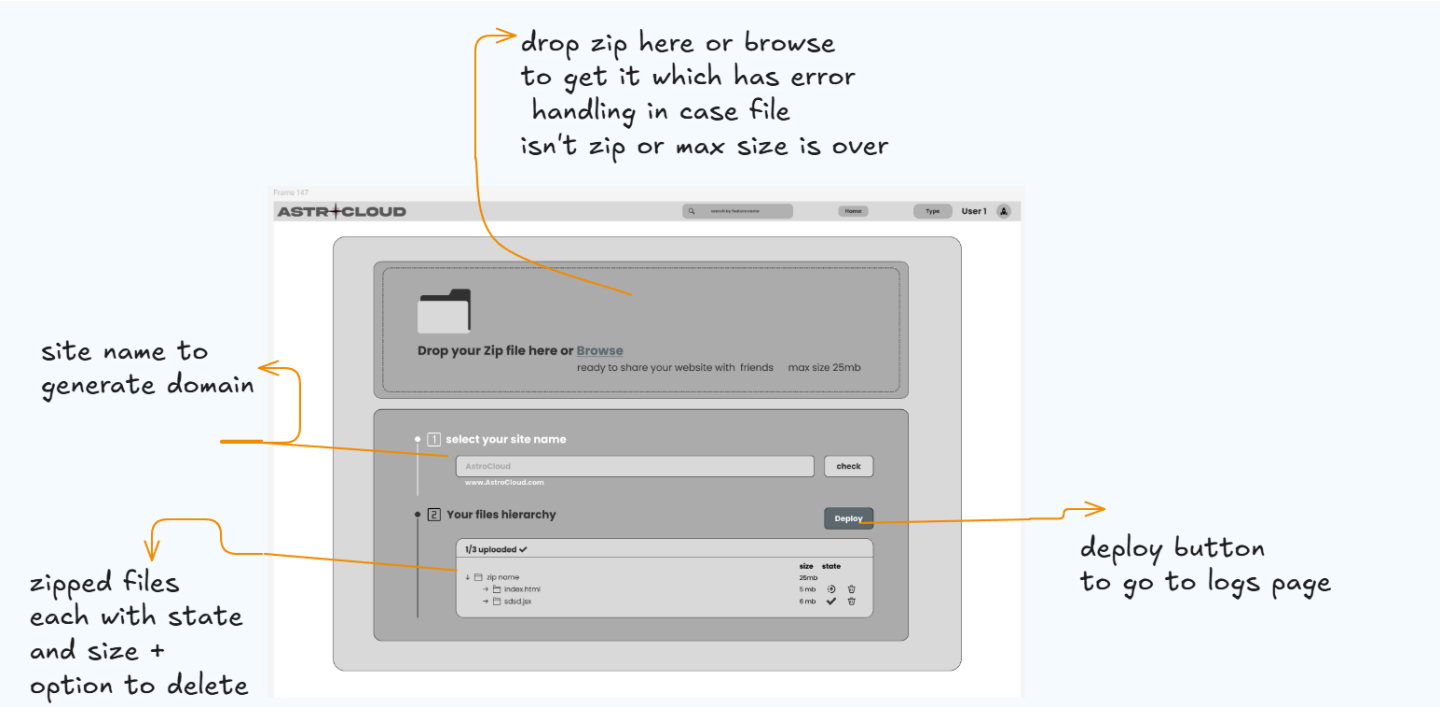
7. Consistency Across UI

- Consistent separation aligns with UI/UX best practices, creating a familiar and intuitive experience for users accustomed to well-designed dashboards.

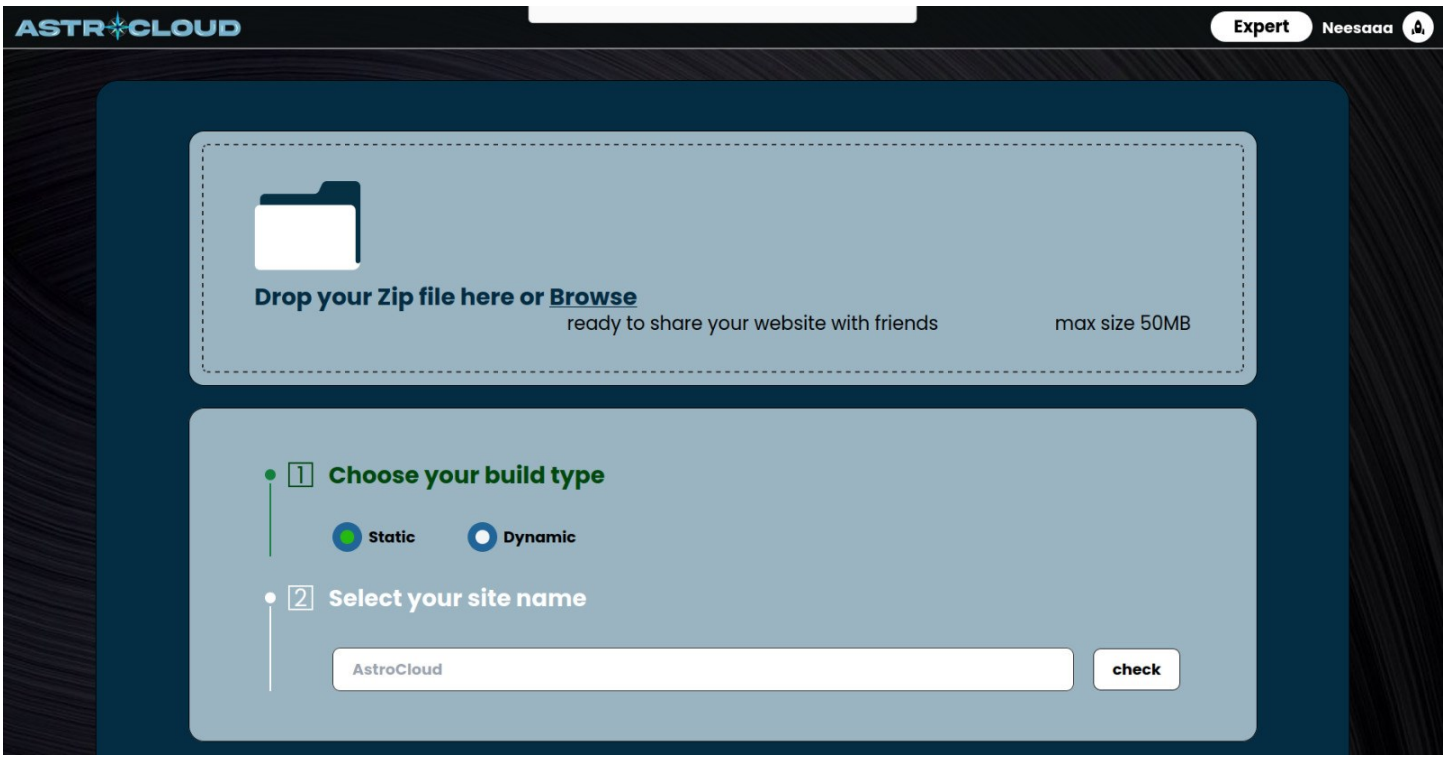
Final:



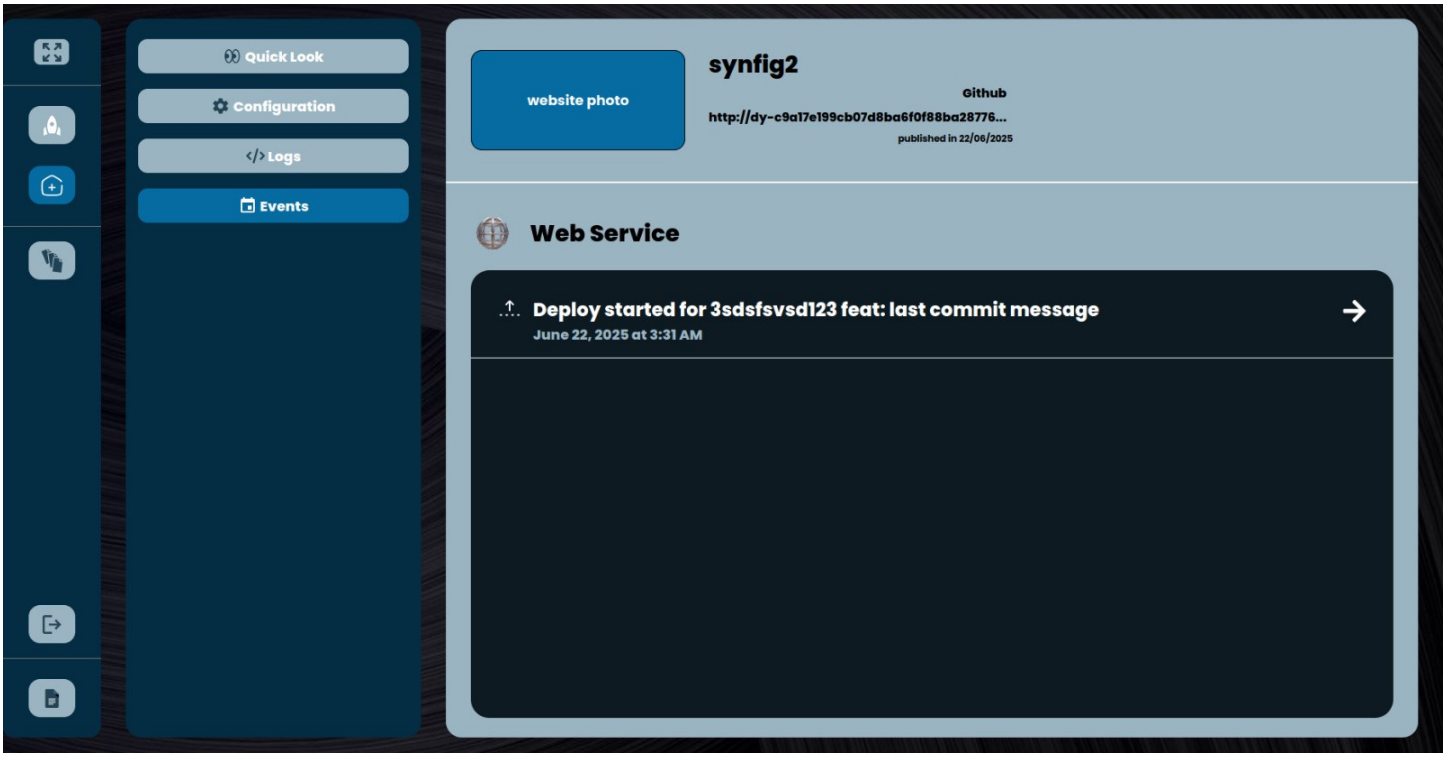
Now suppose we want to add a project from zip file from the start in add project page we clicked import on zip file



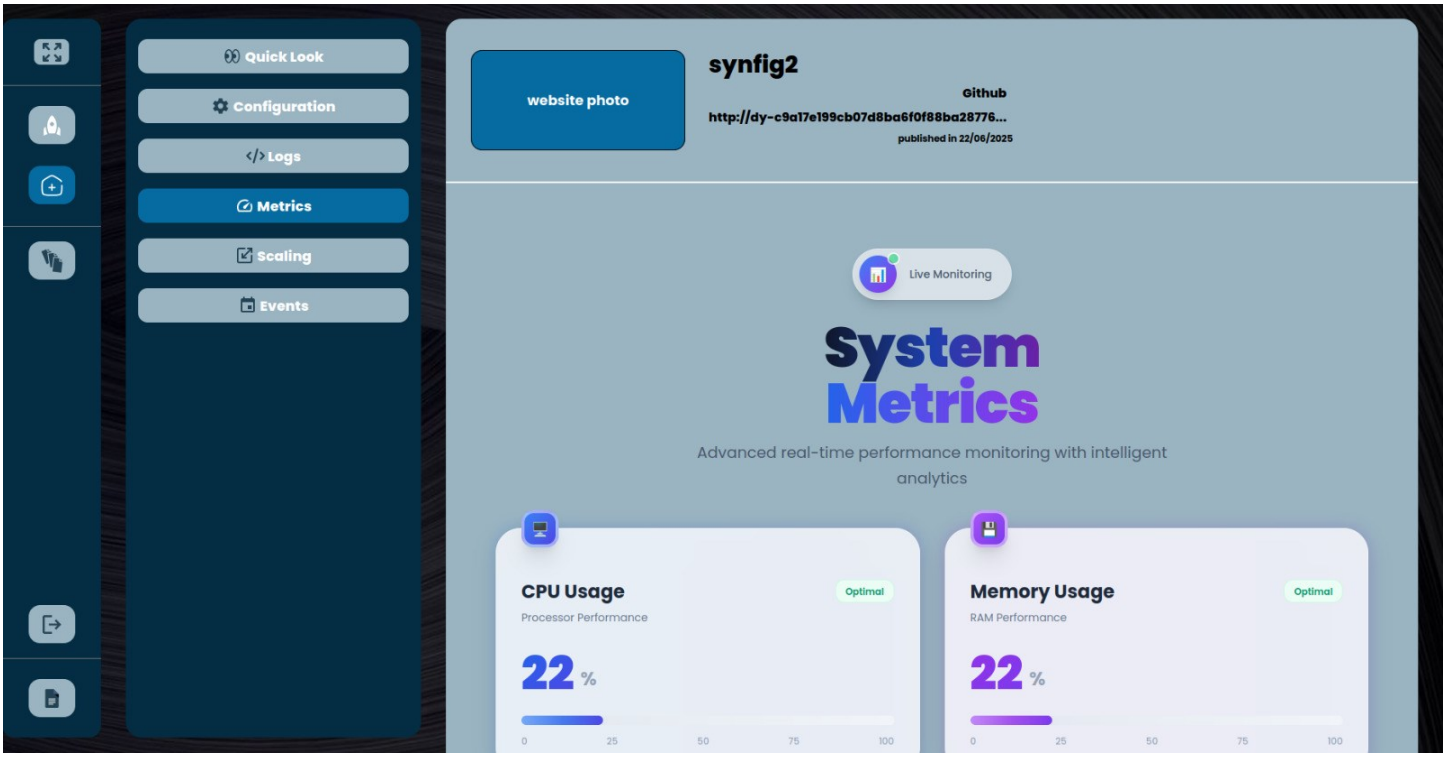
Final:



Events:

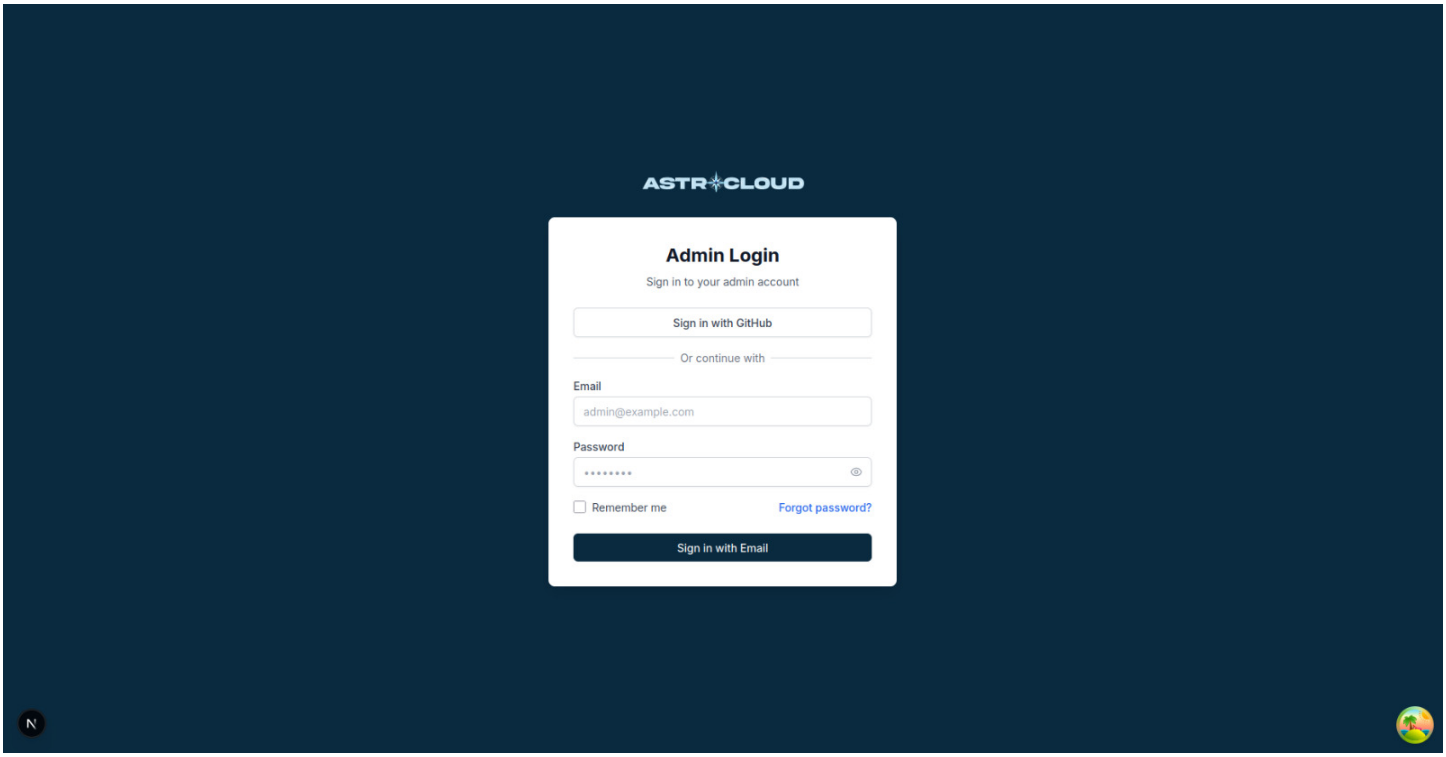


Metrics:

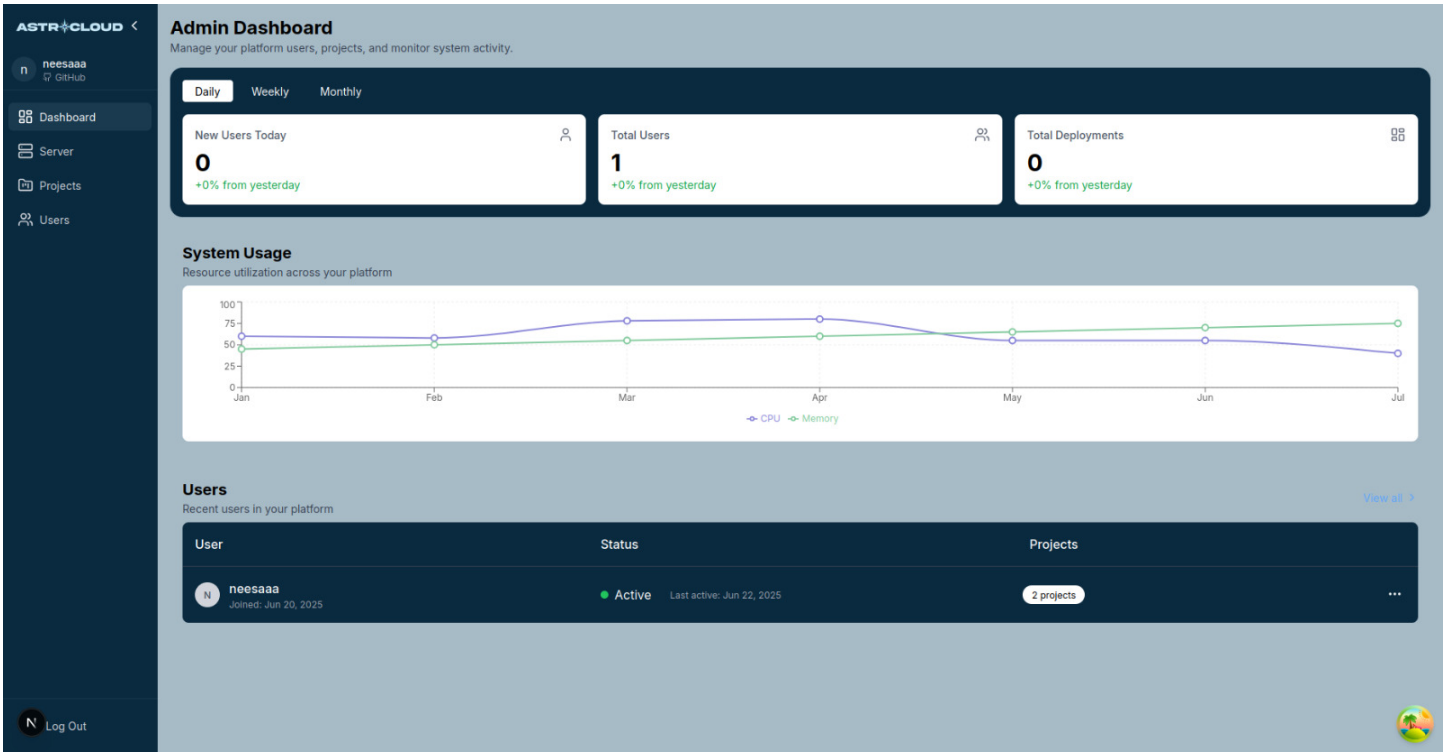


Now we will talk about **Admin page**:

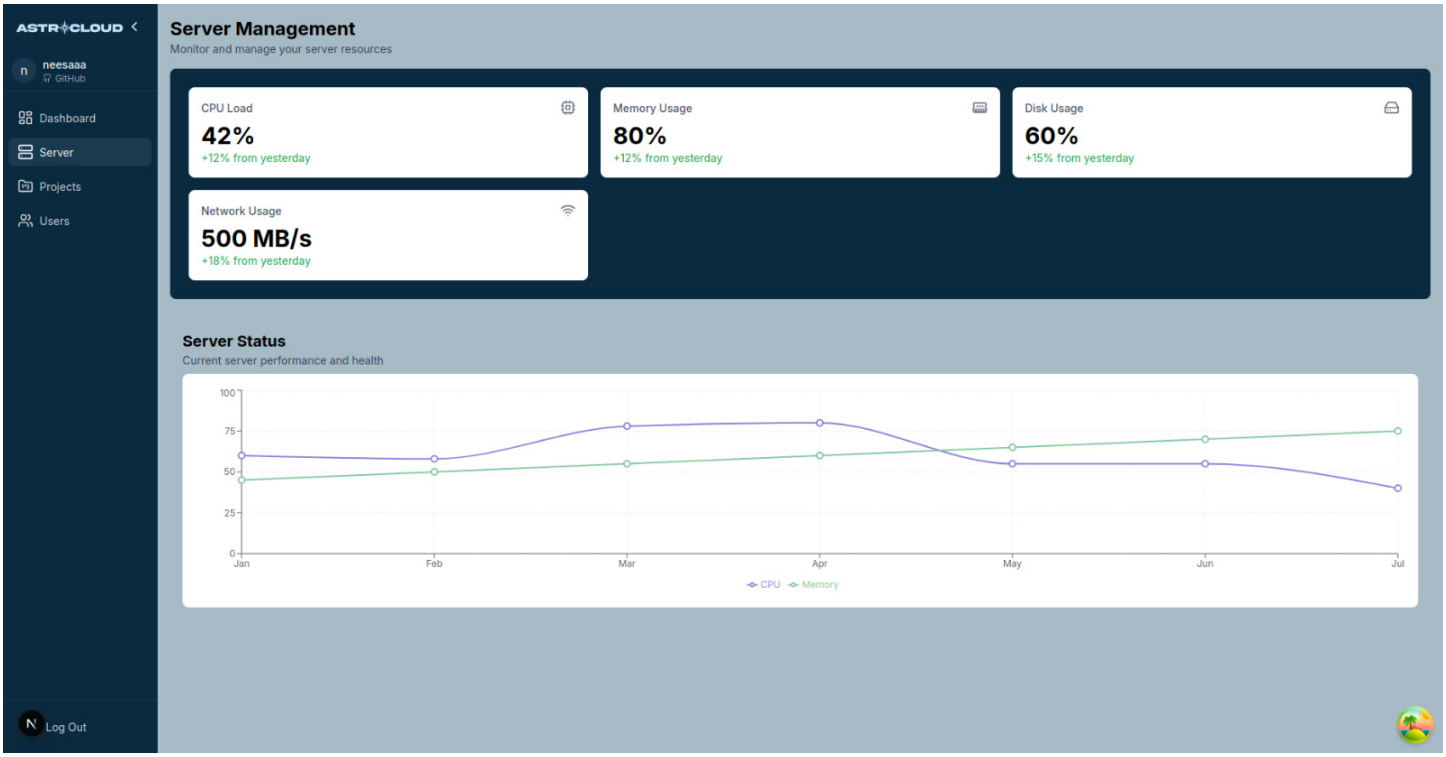
- First login with github



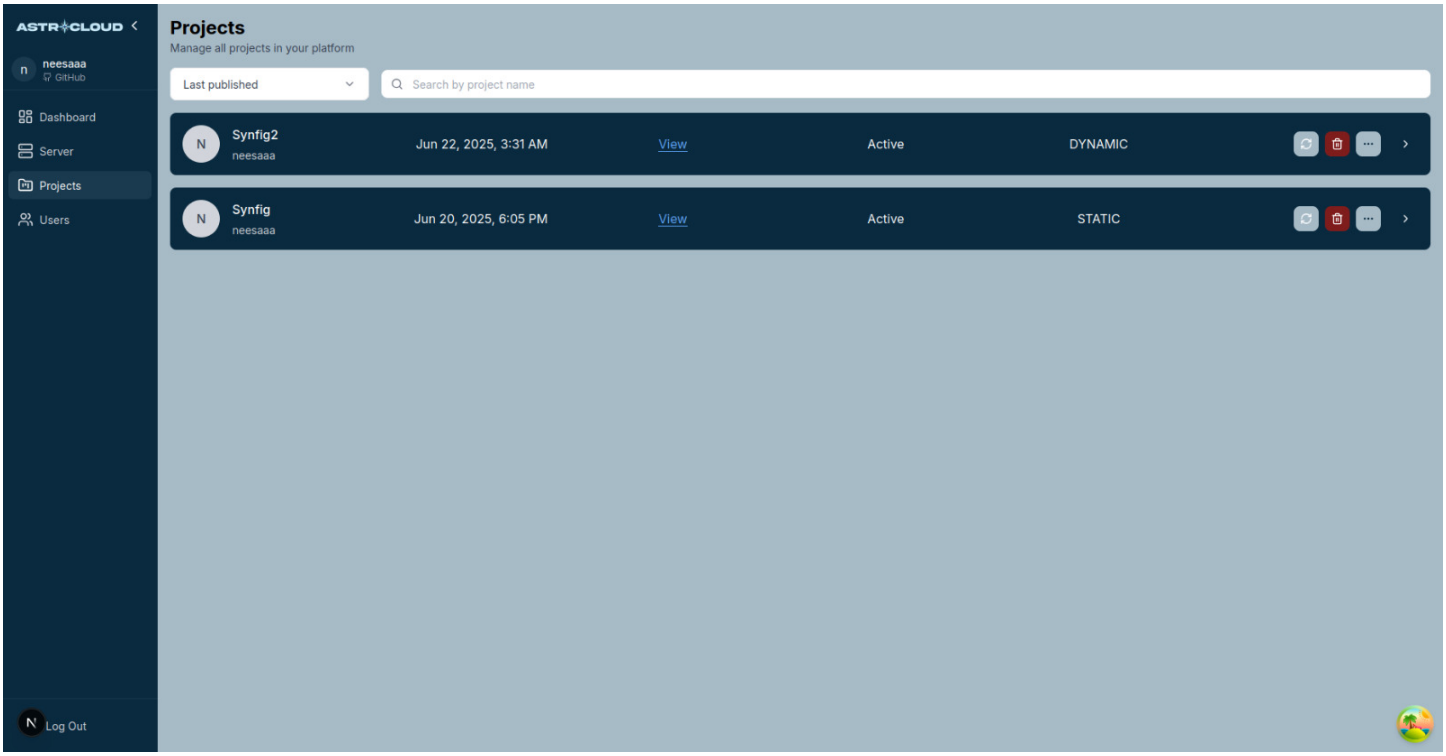
- Dashboard: To show new users, deployments brief of all site.



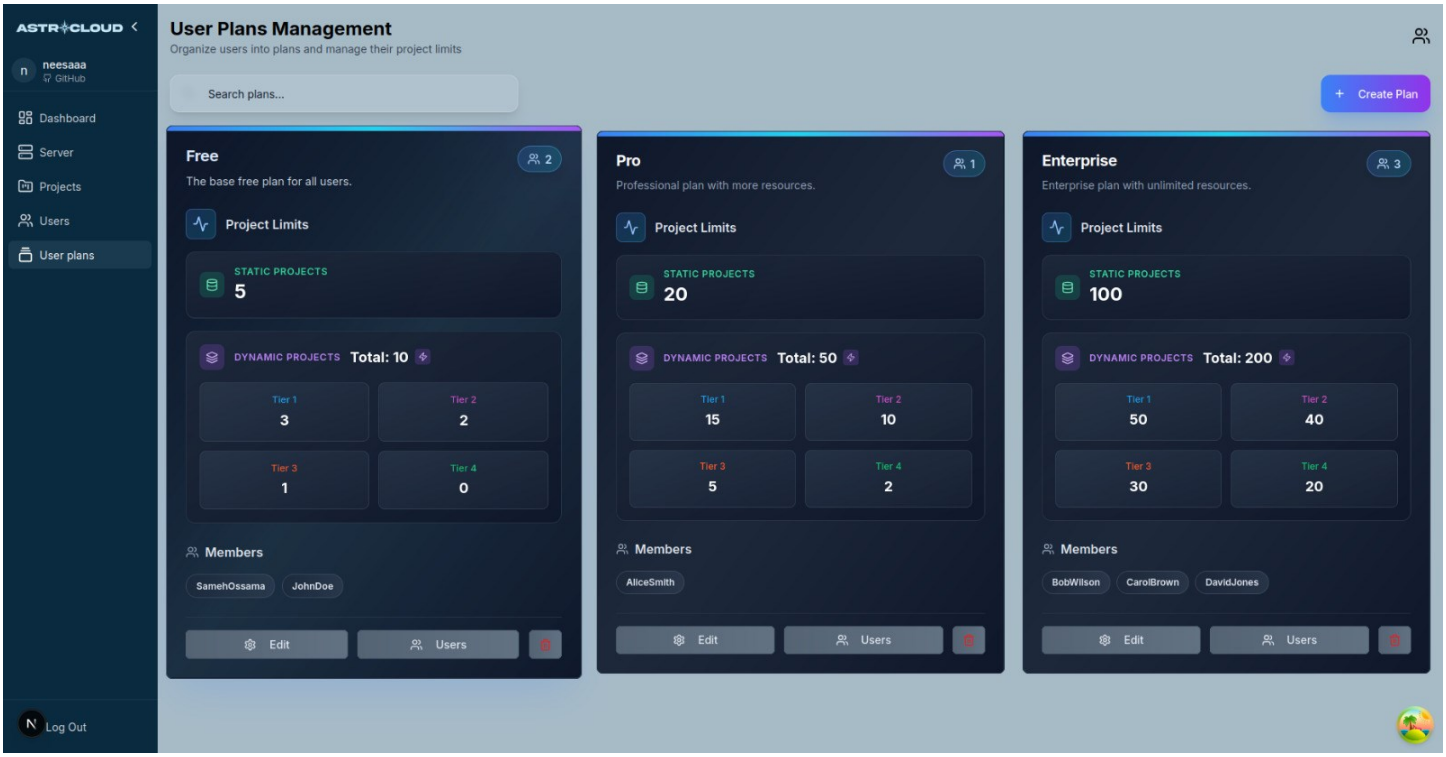
- Server: To show all data on the server and all its specs.



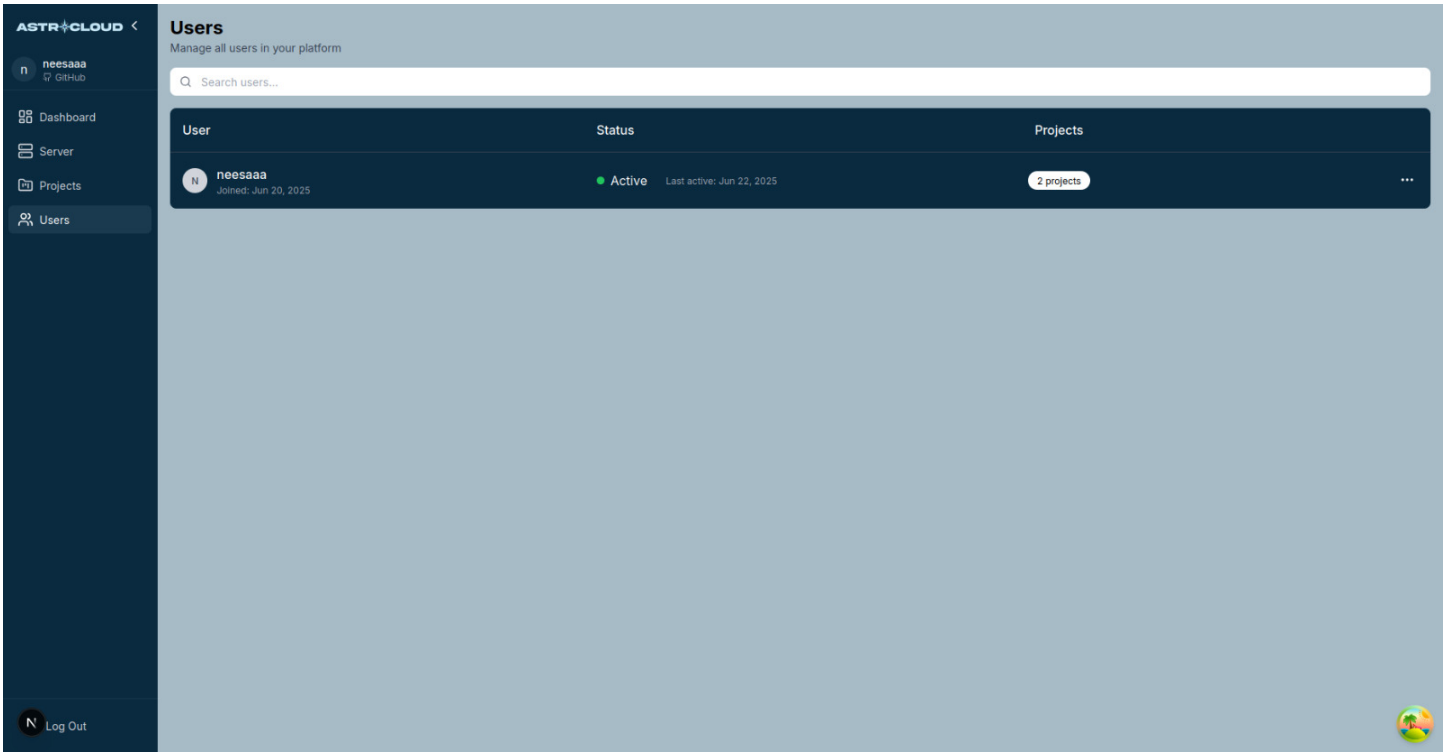
- Projects: To show all projects now.



- User plans: To control plans, add people to quota edit quota to limit use as we don't have infinite resources.



- Users: To show all users in site (screen from local)



Design Choices and Researches

Docker-in-Docker Architecture

There are three approaches to implement Docker inside of Docker.

Mounting the Host's Docker Socket

To execute Docker commands within a container, it is necessary to mount the host machine's Docker socket to the container. This allows the Docker CLI within the container to communicate with the Docker daemon running on the host machine, enabling the execution of Docker commands within the container's environment. The container itself must have the Docker CLI installed for this functionality to work.

Advantages:

- **Simplicity and Speed:** This method is relatively straightforward to implement and provides a quick way to run Docker within a container.

Disadvantages:

- **Security Risks:**
    - Granting a container access to the host machine's Docker socket presents significant security vulnerabilities.
    - It essentially provides the container with unrestricted access to all Docker operations on the host, potentially allowing malicious actions like container escape or data theft.
- **Lack of Isolation:**
    - There is no inherent isolation between containers created by the host and those created within the container. This can lead to unexpected interactions and potential conflicts between these containers.

Using the docker:dind Image

This approach utilizes a Docker image containing its own dedicated Docker daemon, separate from the host's daemon.

- **Advantages:**
    - Creates a separate namespace for the inner Docker daemon and its containers, preventing interference with the host's daemon or other containers.
- **Disadvantages:**
    - Requires privileged mode, granting the container extensive access to the host system.

Using the Nestybox Sysbox Runtime

This approach leverages the Nestybox Sysbox runtime, a container runtime that enables containers to function as lightweight virtual machines. This allows containers to run system-level software like Docker without requiring privileged mode or extensive configuration.

Advantages:

- Provides a secure and robust method for running Docker within a container without requiring privileged mode.

Disadvantages:

- Requires installation and configuration of the Sysbox runtime on the host system.

Migrating from GitHub OAuth to GitHub App

GitHub OAuth:

GitHub also supports OAuth apps. Unlike GitHub Apps, you do not install an OAuth app or control what repositories it can access.

Both OAuth apps and GitHub Apps use OAuth 2.0.

OAuth apps can only act on behalf of a user, while GitHub Apps can either act on behalf of a user or independently of a user.

OAuth apps can have *read* or *write* access to your GitHub data.

- **Read access** only allows an app to *look at* your data.
- **Write access** allows an app to *change* your data.

OAuth scopes:

Scopes are named groups of permissions that an OAuth app can request to access both public and non-public data.

When you want to use an OAuth app that integrates with GitHub, that app lets you know what type of access to your data will be required. If you grant access to the app, then the app will be able to perform actions on your behalf, such as reading or modifying data. For example, if you want to use an app that requests user:email scope, the app will have read-only access to your private email addresses.

OAuth scopes limitations:

A token has the same capabilities to access resources and perform actions on those resources that the owner of the token has, and is further limited by any scopes or permissions granted to the token. A token cannot grant additional access capabilities to a user. For example, an application can create an access token that is configured with an admin:org scope, but if the user of the application is not an organization owner, the application will not be granted administrative access to the organization.

GitHub App:

GitHub Apps are tools that extend GitHub's functionality. GitHub Apps can do things on GitHub like open issues, comment on pull requests, and manage projects. They can also do things outside of GitHub based on events that happen on GitHub.

Using GitHub Apps

In order to use a GitHub App, you must install the app on your user or organization account. When you install the app, you grant the app permission to read or modify your repository and organization data. The specific permissions depend on the app, and GitHub will tell you what permissions the app requested

before you install the app. When you install the app, you will also specify what repositories the app can access. If the app requires any additional configuration, the app will direct you to do so.

You may also need to authorize a GitHub App to verify your identity, know what resources you can access, or take action on your behalf. If you need to authorize the app, the app will prompt you to do so.

Use cases

- **GitHub Apps that act on behalf of a user**

  If you want your app to take action on behalf of a user, you should use a user access token for authentication. The app will be limited by the permissions that have been given to the app as well as the user's permission. With this pattern, the user must authorize the app before the app can take action.

  An example where the GitHub App acts on a user's behalf:

  - A GitHub App that uses GitHub as an identity provider.

- **GitHub Apps that act on their own behalf**

  If you want your app to take action on behalf of itself, rather than a user, you should use an installation access token for authentication. This means that the app will be limited by the permissions that have been given to the app.

  Some examples of automations you could create with a GitHub App, where the app acts on its own behalf, include:

  - A GitHub App that uses webhooks to react to an event given a certain set of criteria. For example, you could create an automation around the REST API endpoints.
- **GitHub Apps that respond to webhooks**

  If you want your app to respond to events on GitHub, your app should subscribe to webhooks. For example, you may want your app to leave a comment when a pull request is opened.

- **GitHub Apps that can take certain actions**

  When you set up your GitHub App, you can select specific permissions for the app. These permissions determine what the app can do via the GitHub API, what they can do on behalf of a signed in user, and what webhooks the app can receive.

Authorizing GitHub Apps

GitHub Apps may need to verify your GitHub identity or interact with GitHub on your behalf. These applications can request authorization for a GitHub App to perform these actions. If an application requests authorization, it will redirect you to a GitHub page prompting you to authorize the app. During authorization, you'll be prompted to grant the GitHub App permission to do all of the following:

- Verify your GitHub identity: When authorized, the GitHub App will be able to retrieve your public GitHub profile. The app may also be able to retrieve some private account information. During the authorization process, GitHub will tell you which account information the GitHub App will be able to access.
- Know which resources you can access: When authorized, the GitHub App will be able to determine which resources you can access that the app can also access. The app may use this, for example, so that it can show you an appropriate list of repositories.
- Act on your behalf: When authorized, the application may perform tasks on GitHub on your behalf. This might include creating an issue or commenting on a pull request.

Once you authorize a GitHub App, the app can act on your behalf. The situations in which a GitHub App acts on your behalf vary according to the purpose of the GitHub App and the context in which it is being used. For example, an integrated development environment (IDE) may use a GitHub App to interact on your behalf in order to push changes you have authored through the IDE back to repositories on GitHub.

Types of Authentications

- **Authentication as a GitHub App**

  To authenticate as itself, your app will use a JSON Web Token (JWT). Your app should authenticate as itself when it needs to generate an installation access token. An installation access token is required to authenticate as an app installation. Your app should also authenticate as itself when it needs to make API requests to manage resources related to the app. For example, when it needs to list the accounts where it is installed. For more information.

- **Authentication as an app installation**

  To authenticate as an installation, your app will use an installation access token. Your app should authenticate as an app installation when you want to attribute app activity to the app. Authenticating as an app installation lets your app access resources that are owned by the user or organization that installed the app. Authenticating as an app installation is ideal for automation workflows that don't involve user input.

- **Authentication on behalf of a user**

  To authenticate on behalf of a user, your app will use a user access token. Your app should authenticate on behalf of a user when you want to attribute app activity to a user. Similar to authenticating as an app installation, your app can access resources that are owned by the user or organization that installed the app. Authenticating on behalf of a user is ideal when you want to ensure that your app only takes actions that could be performed by a specific user.

Difference between authorization and installation

When you **install** a GitHub App on your account or organization, you grant the app permission to access the organization and repository resources that it requested. You also specify which repositories the app can access. During the installation process, the GitHub App will indicate which repository and organization permissions you are granting.

For example, you might grant the GitHub App permission to read repository metadata and write issues, and you might grant the GitHub App access to all of your repositories (Figure on the left).

When you **authorize** a GitHub App, you grant the app access to your GitHub account, based on the account permissions the app requested. During the authorization process, the app will indicate which resources the app can access on your account. When you authorize a GitHub App, you also grant the app permission to act on your behalf.

For example, you might grant the GitHub App permission to read your email addresses (Figure on the right).

Comparison:

Requesting permission levels for resources

Unlike OAuth apps, GitHub Apps have targeted permissions that allow them to request access only to what they need. For example, a Continuous Integration (CI) GitHub App can request read access to repository content and write access to the status API. Another GitHub App can have no read or write access to code but still have the ability to manage issues, labels, and milestones. OAuth apps can't use granular permissions.

Repository discovery

GitHub Apps	OAuth apps
GitHub Apps can look at /installation/repositories to see repositories the installation can access.	OAuth apps can look at /user/repos for a user view or /orgs/:org/repos for an organization view of accessible repositories.
GitHub Apps receive webhooks when repositories are added or removed from the installation.	OAuth apps create organization webhooks for notifications when a new repository is created within an organization.

Webhooks

GitHub Apps	**OAuth apps**
By default, GitHub Apps have a single webhook that receives the events they are configured to receive for every repository they have access to.	OAuth apps request the webhook scope to create a repository webhook for each repository they need to receive events from.
GitHub Apps receive certain organization-level events with the organization member's permission.	OAuth apps request the organization webhook scope to create an organization webhook for each organization they need to receive organization-level events from.
Webhooks are automatically disabled when the GitHub App is uninstalled.	Webhooks are not automatically disabled if an OAuth app's access token is deleted, and there is no way to clean them up automatically. You will have to ask users to do this manually.

Jenkins for CI/CD

Action Plan:

1. Jenkinsfile Template:
   - Create a reusable Jenkinsfile template.
   - Store user-specific data (e.g., repository URL, access token, user ID, allocated port) in a database.
   - Before creating a Jenkins job, dynamically assemble the template with the user-specific data to generate a customized Jenkinsfile.
   - Include the generated Jenkinsfile in the job during its creation.
2. Job Creation Functionality:

- Develop a function to automate the creation of Jenkins jobs.
- Explore and evaluate different methods for creating Jenkins jobs programmatically (e.g., Jenkins REST API, Jenkins CLI, or configuration as code).

3. Job Execution Workflow:
   - Configure the Jenkins job to queue tasks for workers when triggered.
   - The worker processes the tasks by:
     a. Deleting the existing hosted application.
     b. Re-hosting the user's updated application.
4. Process Optimization:
   - Ensure that the process is streamlined to handle multiple users efficiently.
   - Implement error handling to manage failed jobs and ensure smooth recovery.

Ways to create a Jenkins job:

1. Using Jenkins REST API

The Jenkins REST API allows users to interact with the Jenkins server and manage jobs programmatically.

Pre-requisites:

Ensure the Jenkins server is running and accessible.

Authentication:

Obtain an API token from Jenkins by navigating to Jenkins > Your Profile > Configure > API Token. Use this token for secure authentication when making API requests.

Required Libraries:

Install a library to handle HTTP requests (e.g., Axios for JavaScript).

Interacting with the API:

Send requests to Jenkins API endpoints to create, configure, or delete jobs. Provide the necessary data, such as job configuration in XML format, to create a job.

2. Using the Jenkins API Library for Node.js

The Jenkins API Library for Node.js simplifies interactions with Jenkins by providing a higher-level abstraction over the REST API.

Steps to Use the Library:

Install the library using the command npm install jenkins. Then set up the library with your Jenkins server URL and API token for authentication. Use the library's methods to create and configure jobs programmatically.

From these two methods, using Jenkins API was easier and fitter to our application.

Running Jenkins from a container:

Running Jenkins in a container allows us to automate the setup and configuration process, eliminating the need for manual intervention. By defining the Jenkins configuration in a YAML file, we ensure consistency, reproducibility, and streamlined deployment. This approach simplifies the management of authentication, authorization, environment variables, and agent configurations.

Key Features of the YAML File

Authentication Settings:

Configures how users log in to Jenkins. It uses Jenkins' built-in user database (local) to manage user authentication. Disables user sign-up (allowsSignup: false), meaning only administrators can create users. Predefines a list of users with credentials, such as an admin user (id: "admin" and password: "admin123").

Authorization Settings:

Manages what logged-in and anonymous users can do in Jenkins. Grants full access to logged-in users (loggedInUsersCanDoAnything). Restricts access for anonymous users (allowAnonymousRead: false).

Environment Variables:

Defines application-specific environment variables:

Agent (Node) Configurations:

Sets up a permanent Jenkins agent (permanent) for running jobs. Configures the agent's name (agent1), workspace directory where agent stores files (/home/jenkins/agent), and connection mechanism (inbound). Includes an authentication secret (secret: “”) for secure communication between the master Jenkins instance and the agent.

Docker Compose for Jenkins

Within the docker-compose file, we will define our services:

- **Jenkins Master:**

  This is the main part of the CI/CD setup and acts as the controller. It manages pipelines, jobs, configurations, and plugins.
  The image: “jenkins/Jenkins:lts” pulls the latest stable version of Jenkins.
  Ports:
  - 8080: Allows you to access Jenkins through a web browser.
  - 50000: Used for communication between the Jenkins master and its agents.

  Volumes:
  - jenkins_home is like the “hard drive” of Jenkins. It stores all Jenkins data, such as job configurations and plugins, so nothing is lost if the container is restarted.
  - jenkins.yml is the configuration file for Jenkins, defining its behavior (e.g., users, permissions, plugins).

  Environment Variables:

  - **JAVA_OPTS=-Djenkins.install.runSetupWizard=false:** Skips the Jenkins setup wizard, so the configuration is automated.
  - **CASC_JENKINS_CONFIG=/var/jenkins_home/casc_configs/jenkins.yml:** Tells Jenkins where to find the configuration file.

- **Jenkins Agent:**

  This is a "worker" that performs tasks (like running tests, building code, or deploying applications) as instructed by Jenkins master. Jenkins master doesn’t directly execute jobs, it delegates tasks to agents. This separation helps with scalability and performance.
  Image: The “jenkins/inbound-agent:latest” provides a ready-made container with everything needed to connect to the master.
  Environment Variables:

- **JENKINS_URL:** Specifies the URL of the Jenkins master, so the agent knows where to connect (e.g. "http://jenkins:8080").
- **JENKINS_AGENT_NAME:** The name of this agent (e.g., agent1) is defined for easy identification in Jenkins.
- **JENKINS_SECRET:** A unique token that ensures only authorized agents can connect to the master.

Dependencies: The agent waits for the Jenkins master to be up and running before starting (depends_on: jenkins).

Installing Jenkins plugins

Plugins are essential for extending Jenkins' core functionality. They allow Jenkins to integrate with version control systems, build tools, deployment platforms, and more. Some examples include:

- Git Plugin: Enables Jenkins to interact with Git repositories.
- Pipeline Plugin: Adds support for Jenkins pipelines.

Ways to Install Jenkins Plugins

- **Using the Jenkins UI**

  Navigate to Manage Jenkins > Manage Plugins then search for the desired plugin in the Available tab and click Install without restart or Download now and install after restart.

  Pros: User-friendly and requires no scripting knowledge, no need to restart Jenkins immediately unless explicitly chosen.

  Cons: Manual process, time-consuming if many plugins are required.
- **Using the Jenkins CLI**

  Jenkins provides a command-line interface (CLI) that can be used to install plugins.

  Pros: Useful for automation when combined with scripts, does not require UI interaction.

  Cons: Requires downloading the jenkins-cli.jar file, must handle authentication (e.g., API token).
- **Using a Bash Script with plugins.txt**

  You can automate plugin installation by listing required plugins in a plugins.txt file and using a script to read this file and install plugins.

  Example plugins.txt:
  ```
  git
  pipeline
  redis
  ```

  Pros: Fully automated, great for CI/CD pipelines, scales well for multiple plugins.

  Cons: It requires Jenkins to restart after installation, leading to potential downtime. Errors in the plugins.txt file can interrupt the process.
- **Pre-installing Plugins**

  Install plugins from Jenkins on format jpi in a folder. Then, mount this folder to the plugins directory inside the Jenkins container using a Docker Compose volume. This allows the Jenkins container to load the pre-installed plugins automatically.

  Pros

  - **Fast Setup:** Plugins are immediately available on container startup without any additional installation time.
  - **Offline Installation:** If you already have the plugins downloaded, you don’t need internet access to install them.
  - **Reusability:** You can share or version-control the plugins folder to standardize setups across multiple Jenkins instances.

  Cons

  - **Lack of Granularity:** All plugins in the folder are loaded, even if not required.

    - **Plugin Dependency Management:** Plugins with dependencies might fail if their required versions are not present.
    - **Version Locking:** This approach locks the Jenkins instance to the exact versions of the plugins in the folder.
- **Configuration as Code (CASC)**

  Define plugins in the Jenkins Configuration as Code YAML file (jenkins.yml) and let Jenkins install them on startup.

  Pros: Simplifies plugin management as part of a declarative setup, easily version controlled.

  Cons: Only works if you are using the Jenkins Configuration as Code plugin.

We explored all those methods, and the preferred method is Pre-installing Plugins.

Advantages of Docker Compose for Jenkins

- **Reproducibility:** The setup can be deployed consistently across different environments with minimal effort.
- **Scalability:** Adding more agents or services is straightforward with Docker Compose.
- **Isolation:** Each service runs in its own container, reducing conflicts and improving reliability.
- **Portability:** The configuration can be version-controlled and shared, enabling easy collaboration and deployment.

Advantages of using Jenkins:

- **Mature and Extensible:** Jenkins has been around for years and supports a vast array of plugins for every CI/CD need.
- **Self-Hosted:** You can host Jenkins on your own infrastructure, giving you complete control.
- **Flexible Pipelines:** Declarative and scripted pipelines allow for complex workflows.
- **Integration:** Jenkins integrates with every tool or platform (Git, Docker, Kubernetes, etc.).

Disadvantages of using Jenkins:

- **Setup Complexity**: Requires installation, configuration, and maintenance of the Jenkins server.
- **Scalability Issues:** Managing resources can be challenging for large-scale setups without a robust infrastructure.
- **UI and Usability**: Jenkins has a steeper learning curve compared to modern alternatives like GitHub Actions.

VMs vs Containers vs MicroVMs

We faced a dilemma while exploring dynamic hosting options, particularly regarding whether to host users' dynamic applications in containers, virtual machines (VMs), or microVMs. To make an informed decision, we compared these options based on various criteria.

	VMs	**microVMs**	**containers**
Security	More secure due to full OS isolation (larger attack surface due to more components).	More secure than containers; lighter than VMs but isolated like them.	Less secure due to shared host OS kernel (smaller attack surface).

Performance	Lower performance due to full OS overhead and slower boot times.	Near-container performance with better isolation. Fast boot times.	High performance, lightweight, fast startup, low resource use.
Observability	Harder to observe due to multiple layers (guest OS, hypervisor, etc.).	Moderate observability; simpler than VMs, but fewer tools than containers.	Highly observable; integrates well with tools like Prometheus.
Cost	Higher cost due to full OS per VM and higher hardware usage.	Lower cost than VMs; better efficiency with lightweight isolation.	Low cost due to efficient resource sharing and high density.
Complexity	High complexity; managing full OS, networking, and storage for each VM.	Moderate complexity; lighter than VMs, but still lower level than containers.	Low complexity: tools like Docker and Kubernetes simplify management.
Automation	Harder to automate due to full OS lifecycle and slow boot times.	Supports automation tools like Firecracker APIs; faster than VMs.	Highly automated via Kubernetes and container runtimes.
Configuration	Requires full manual setup for each VM (networking, security, etc.).	More configurable than containers, less effort than full VMs.	Minimal setup; handled by container runtime or orchestrators.
Flexibility & Control	Maximum flexibility and control over OS and kernel.	Balanced: offers sufficient control with reduced complexity.	Easier to manage but with limited low-level control.
Low-Level Control	Full control over OS, kernel, and hardware settings.	Partial control; better than containers, not as deep as VMs.	Limited to what the container runtime and host OS allow.
Ecosystem & Tooling	Fewer ready-made solutions compared to container platforms.	Growing ecosystem (e.g., Firecracker, Kata Containers).	Mature ecosystem (Docker, Kubernetes, etc.).

From this comparison we decided that microVMs will provide us with the best of both worlds.

MicroVMs comparisons

After deciding to pursue the microVM approach, we explored four notable approaches: **Nabla Containers**, **Kata Containers**, **Firecracker**, and **gVisor**. Each presents a unique tradeoff between security guarantees, compatibility with standard container tooling (like Docker and Kubernetes), and operational complexity.

	Nabla Containers	**Kata Containers**	**Firecracker**	**gVisor**
Isolation Model	Strong isolation via Unikernel + Solo5 (minimal syscalls, no shared host kernel)	Full VM per container with lightweight VMM (typically QEMU or Firecracker)	MicroVMs using KVM (like minimal VMs stripped of unnecessary features)	User-space kernel (Sentry) intercepting syscalls, running containers in sandboxed process
OCI Runtime-Spec Compliant	via runnc (drop-in for runc)	via kata-runtime	(Not a container runtime by itself, but can integrate via Firecracker-containerd or Kata)	via runsc

OCI Image-Spec Compliant	Requires custom Nabla images (not compatible with regular Docker images)	Runs standard OCI/Docker images without changes	Firecracker alone doesn't run OCI images; needs tools like containerd-shim	Runs most OCI/Docker images with only minor syscall/interface limitations
Security Features	Minimal syscall surface (only 7 host syscalls), unikernel isolation	Hardware virtualization, separate kernel per container, agent-isolated communication	Minimal device model and syscall surface, purpose-built for secure multi-tenancy	User-space syscall handling, limited access to kernel, file access proxy via Gofer
Performance	Good performance, but unikernel boot and build complexity can add overhead	Lower than native containers; boot times and I/O slower than containers, though optimized VM images help	High performance and fast startup (~100ms boot time); optimized for FaaS	Slightly slower than native containers; syscall translation overhead
Compatibility with Toolchains	Limited (custom image format), requires image rebuild	High compatibility with existing Docker/K8s setups	Low direct compatibility, but improving via Firecracker-containerd and Kata integration	High compatibility; works with Docker, containerd, and Kubernetes
Networking Support	Limited, custom setup	Full CNI support, works well in Kubernetes	Minimal virtual network support (custom VMM API or via containerd integration)	Supports networking via gVisor + CNI; may have limitations on advanced net features
Use Cases	Research, minimal-attack-surface containers, experimental workloads	Secure containerized workloads, K8s multi-tenancy, cloud VM/container hybrid	Function-as-a-Service (AWS Lambda), serverless, cloud workloads requiring isolation	GKE sandbox, secure multi-tenant apps, where syscall filtering is preferred over full VMs
Maturity	Experimental / Alpha	Production-ready and used in cloud environments	Actively developed by AWS, some experimental uses in container ecosystems	Production-grade in GCP (GKE Sandbox), widely tested

In the end, we chose Kata Containers. Its ability to run unmodified OCI-compliant container images and seamlessly integrate with Kubernetes via containerd and CRI made it the most balanced and production-ready solution.

MicroVMs (Firecracker) vs Containers (containerd)

Overview

Feature	Firecracker	Containerd
Definition	Firecracker is a lightweight virtual machine (microVM) manager that enables secure, multi-tenant workloads with minimal overhead. It provides a RESTful API and operates on Kernel-based Virtual Machines (KVMs).	A container runtime that manages the lifecycle of containers, handling image transfer, storage, execution, and supervision. It is a core component of Docker and adheres to the Open Container Initiative (OCI) standards.
Use Case	Optimized for running microVMs with fast startup times and strong security isolation. Used in serverless computing environments like AWS Lambda and AWS Fargate.	Designed for containerized applications, often used in Kubernetes clusters for deploying and managing container workloads.

Management Approach

Feature	Firecracker	Containerd
Kubernetes Integration	Uses firecracker-containerd, a plugin that translates Kubernetes container expectations into Firecracker microVMs.	Kubernetes communicates with Containerd using the Container Runtime Interface (CRI) through a native CRI plugin.
Orchestration	Requires custom implementation or additional layers for full Kubernetes compatibility.	Fully supported by Kubernetes, enabling seamless orchestration of containerized applications.

Security and Isolation

Feature	Firecracker	Containerd
Kernel Isolation	Each microVM runs its own kernel, providing better security and isolation.	Containers share the host machine's kernel, which can introduce security vulnerabilities.
Attack Surface	Smaller due to minimal dependencies and lightweight design.	Larger attack surface due to shared kernel and greater interdependencies.

Control Plane

Firecracker does not have a built-in control plane like Kubernetes but can be managed using custom orchestration tools. Containerd, on the other hand, integrates seamlessly with Kubernetes, making it easier to deploy and manage workloads at scale.

Networking

Feature	Firecracker	Containerd
Networking Standard	Uses the Container Network Interface (CNI) for networking, similar to containers.	Also uses CNI for networking in Kubernetes environments.
Implementation	Requires additional configuration to set up networking for microVMs.	Integrated with container networking solutions and Kubernetes services.

Comparison with Kubernetes

Firecracker has some limitations compared to Kubernetes-based container deployments:

- **Monitoring:** Firecracker lacks built-in monitoring tools; additional software is needed.
- **Feature Gaps:** Firecracker microVMs do not natively support key Kubernetes features, requiring manual implementation:
  - High availability
  - Auto-scaling
  - Service discovery
  - Capacity planning

Ideal Use Cases for Firecracker

Firecracker is best suited for scenarios where traditional containers may not provide sufficient security or isolation. Suitable applications include:

- **Security-sensitive environments:** Firecracker microVMs provide strong isolation, making them ideal for running workloads that require enhanced security.
- **Short-lived workloads:** Firecracker's fast startup time is beneficial for ephemeral computing tasks such as serverless computing.
- **Managed hosting environments:** Useful in managed services where multi-tenancy and workload isolation are critical.
- **Training and labs:** Firecracker's lightweight footprint makes it an efficient option for test environments and experimentation.

Firecracker and OCI Compatibility

Firecracker microVMs can be packaged as OCI-compliant images (similar to Docker images), making them deployable across cloud providers.

Scalability

Firecracker has been tested to run **4,000 microVMs on a single host**, demonstrating its ability to handle highly scalable workloads efficiently.

Conclusion

Firecracker and Containerd serve different purposes:

- **Firecracker** is ideal for security-critical, isolated, and fast-startup workloads like serverless computing.
- **Containerd** is optimized for container orchestration and Kubernetes-based deployments.

Organizations should choose between them based on their security, performance, and orchestration needs.

Firecracker

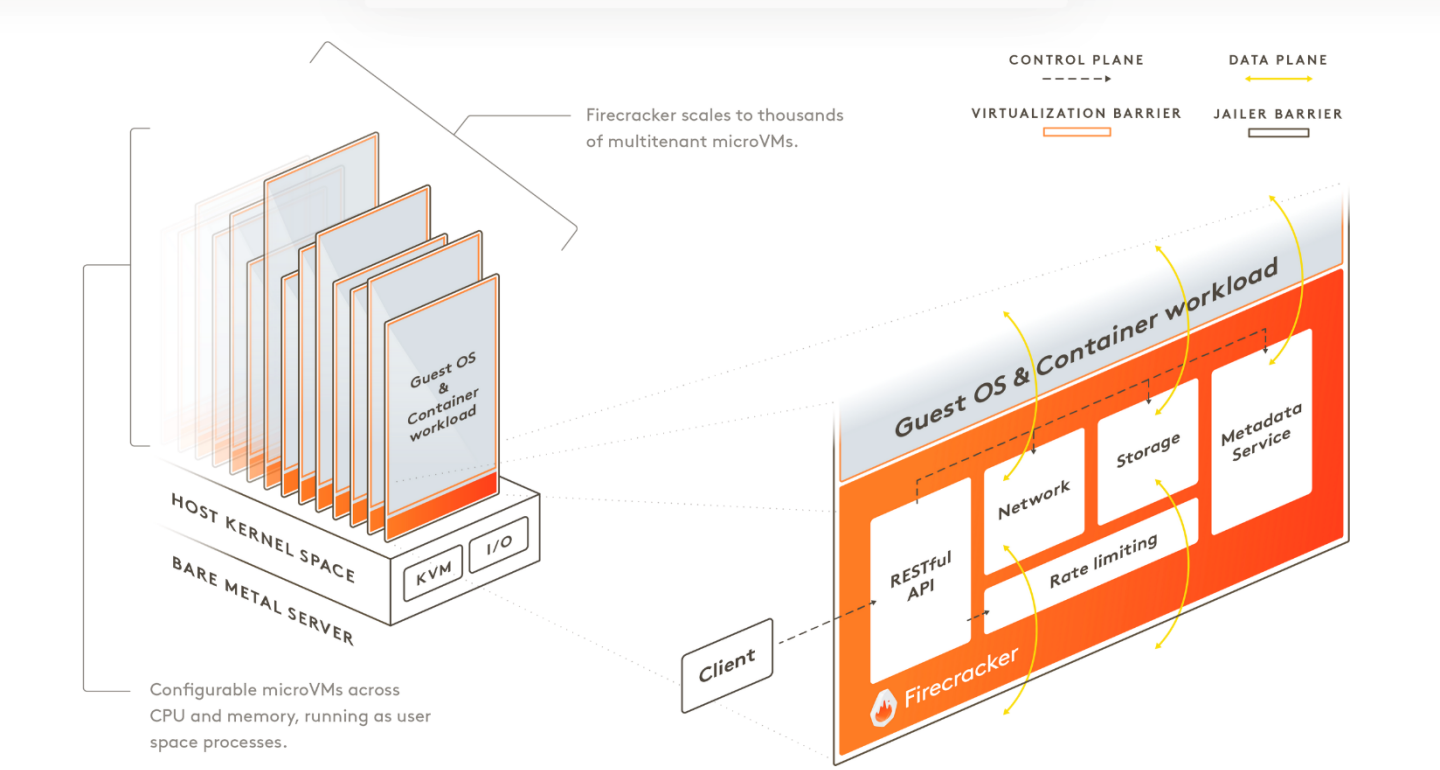
A virtualization technology for creating and managing secure, lightweight and multi-tenant (multiple clients share the same computing resources) containers aka. **microVMs**.

What makes it different than traditional containers

- Uses lightweight VMs each with its own kernel instead of shared-kernel containers.
- More isolated and secure environment (the reason above).

How it works

- Uses the Linux Kernel-Based Virtual Machine (KVM) as a platform to create microVMs.
- Firecracker processes are controlled via RESTful API for doing actions such as:
    - configuring number of vCPUs
    - starting the machine
    - provides built-in rate limiter, which controls network and storage resources used by the microVMs on the same machine
- A metadata service is used to securely share configuration information between host and guest machine.



Benefits

- **Secure from the ground up**
    - KVM-based virtualization is more secure than traditional VMs.
    - uses a minimal device model excluding non-essential functionality and decreasing attack surface.
- **Speed by design**
    - accelerate kernel loading (runs application code in as little as 125ms).

    - supports microVMs creation rate of up to 150 microVMs per second per host.
- **Scale and efficiency**
    - reduced memory overhead (less than 5 MiB) enables high density of microVMs on same server.
    - built-in rate limiter, enables optimized sharing of network and storage across microVMs.

Requirements

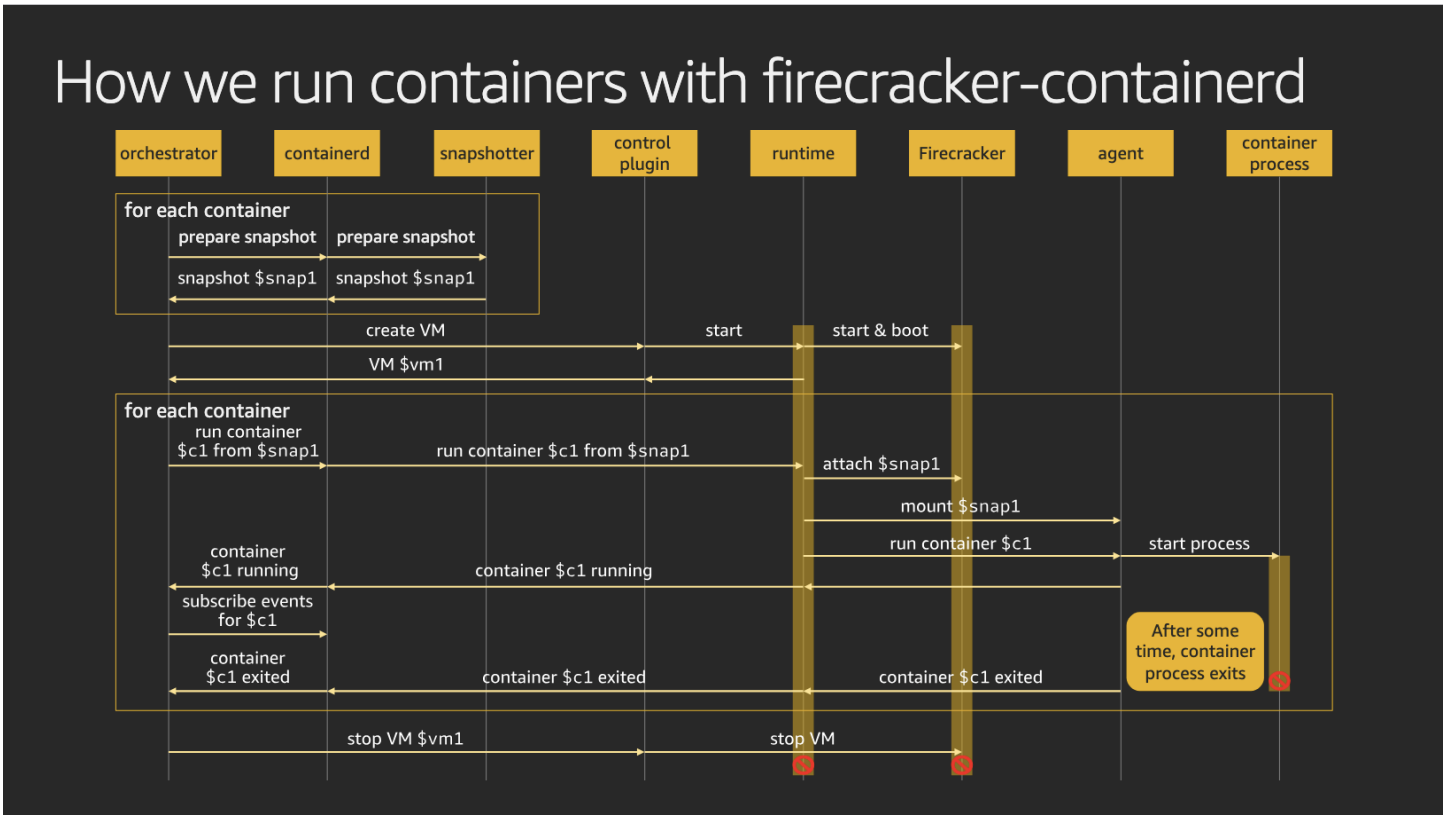
- Bare metal machine or a VM that supports nested virtualization using KVM
- Uncompressed kernel binary + init
- Root filesystem image

Firecracker-containerd

The firecracker-containerd project connects **containerd** with Firecracker, enabling the management of microVMs as container workloads. This integration allows Kubernetes to schedule and manage Firecracker microVMs seamlessly.

How it Works

- **Runtime Integration**: An out-of-process shim runtime links containerd to the Firecracker Virtual Machine Monitor (VMM). This setup allows containerd to manage Firecracker microVMs as if they were standard containers.
- **Agent Inside MicroVMs**: Within each microVM, an agent runs and invokes runC via containerd's containerd-shim-runc-v1 to create standard Linux containers inside the microVM.
- **Root Filesystem Image Builder**: A tool constructs a root filesystem for the microVM, containing runc and the firecracker-containerd agent, ensuring the microVM has the necessary components to run containers.



What makes it different from Kata-container

Kata-container: the value add here is more isolation, as the container is spawned inside of a minimal Firecracker VM.

firecracker-containerd: enables you to do the same as Kata; add isolation for a container; but this time in a bit more lightweight manner, as a containerd plugin.

firecracker-containerd still in Early Stage!

Currently integration with Kubernetes CRI is still in development, so we can't use it as a container runtime up till now.

References

https://kodekloud.com/blog/run-docker-in-docker-container/

https://docs.github.com/en/apps

https://docs.github.com/en/rest?apiVersion=2022-11-28

https://github.com/octokit/octokit.js

https://www.prisma.io/docs

https://react.dev/learn

https://v5.reactrouter.com/web/guides/quick-start

https://tailwindcss.com/docs/installation/using-vite

https://docs.docker.com/

https://github.com/apocas/dockerode

https://learn.microsoft.com/en-us/azure/developer/javascript/

https://nginx.org/en/docs

https://letsencrypt.org/docs

https://certbot.eff.org

https://azure.microsoft.com

https://www.inovex.de/de/blog/containers-docker-containerd-nabla-kata-firecracker/

https://benriemer.medium.com/docker-vs-containerd-vs-nabla-vs-kata-vs-firecracker-and-more-108f7f107d8d