Project Name: Cloudbuilder
Group #: 2

Team Introduction

Background

__**Individual Profiles:**__

Jeet Das: As Project Manager and Back End Developer, Jeet worked on the architectural decisions and integration of backend services. He managed the AWS services integration which includes ECR and ECS configurations, vital for the deployment processes.

Ashna Ali: A skilled Front End Developer, Ashna has designed and implemented the user interface using Next.js. Her contributions are not just limited to coding but also include user experience design, ensuring the interface is both functional and aesthetically pleasing.

Krutarth Lad: Focusing on backend development, Krutarth has handled the creation of RESTful APIs that connected the frontend of our application the backend service. His extensive experience in building web applications brought a wealth of knowledge to the team.

Hui Jin: As the Testing Lead, Hui has set up rigorous testing frameworks that ensure the reliability and stability of CloudBuilder. His role involves coordinating with all team members to ensure that every feature meets the highest standards of quality before it's rolled out.

__**Collaboration Tools Used:**__

Agile Tools: We worked using an adapted variant of agile where we did sprint planning and backlog management, facilitating an easy development environment and efficient management of tasks.

Communication: Regular use of Zoom for meetings and Discord for daily communication has ensured that team members are always aligned and can share updates and issues quickly.

Version Control and Code Review: GitHub has been instrumental for source code management and review, enabling collaborative coding and version tracking.

Motivation/Purpose

Detailed Rationale

Market Gap Identification:

Need for Streamlined Processes: Traditional cloud deployment solutions often involve cumbersome manual setups and configurations that can be error-prone and time-consuming. CloudBuilder was conceptualized to fill this gap by providing a more streamlined, automated approach to cloud deployment.

Reduction of Manual Intervention: Many existing tools require detailed knowledge of infrastructure and scripts, which can deter less experienced developers or slow down experienced developers with repetitive tasks. CloudBuilder automates these processes, significantly reducing the need for manual intervention and thus lowering the barrier to entry for deploying applications at scale.

Developer Pain Points:

Complexity of Deployment Scripts: Developers often face the challenge of writing and maintaining complex scripts for deployment, which not only consumes time but also increases the potential for errors. CloudBuilder simplifies this aspect by automating the script execution and handling complexities internally.

Opacity in Build and Deployment Logs: A common frustration is the lack of clear, actionable feedback during and after the deployment process. CloudBuilder addresses this by providing detailed, real-time streaming of logs directly to the user interface, making the deployment process more transparent and less daunting.

Management of Multi-Service Deployments: Coordinating between different services and managing their deployment in a synchronized manner can be challenging, especially as the complexity of applications increases. CloudBuilder facilitates this by orchestrating the build and deployment processes across multiple services seamlessly.

Strategic Importance:

Enhancement of Developer Productivity: By automating routine and complex tasks associated with deployment, CloudBuilder frees up developers to focus on the actual development of the application, fostering innovation and creativity.

Error Reduction: Automation reduces the chance of human error, which can lead to failed deployments or runtime issues. CloudBuilder’s automated checks and processes ensure that deployments are more reliable and consistent.

Cognitive Load Reduction: Managing the intricate details of deployment can be mentally taxing for developers. By simplifying this process, CloudBuilder allows developers to devote their cognitive resources to solving problems that are central to their application's functionality and user experience.

System Architecture

Comprehensive Breakdown

The CloudBuilder project utilizes a microservices architecture to ensure scalability, modularity, and ease of maintenance. This architecture allows individual components of the system to be updated, scaled, and debugged independently, enhancing the overall robustness and responsiveness of the application.

Architecture Diagram

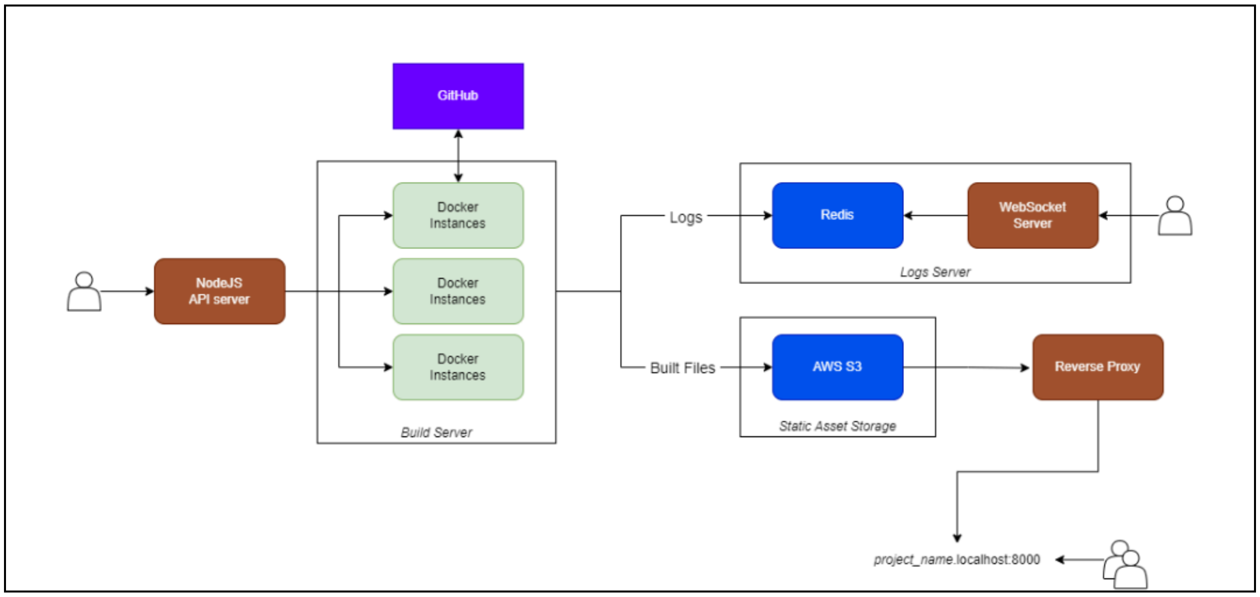


Figure 1: Overview of Cloudbuilders architecture

Component Descriptions

Each component of CloudBuilder plays a crucial role in the system's overall functionality:

Frontend:

Technology: The frontend is developed using Next.js, a React framework that enables server-side rendering and static website generation, which are beneficial for fast load times and SEO.

Functionality: It serves as the user interface where developers input the GitHub project URL they wish to build and deploy. It displays real-time progress updates during the build process and ultimately the URL of the deployed application.

Interaction: The frontend communicates directly with the API-server to submit build requests and retrieve updates on the build status. It also handles user authentication and session management to ensure secure access to deployment functionalities.

Build-Server:

Technology: This component runs in a Docker container, ensuring that each build process is isolated and consistent, regardless of the underlying infrastructure. The Docker images are stored on AWS ECR (Elastic Container Registry), and the containers are managed using AWS ECS (Elastic Container Service).

Functionality: Upon receiving a build request from the API-server, the build-server clones the specified GitHub repository, executes build scripts (e.g., npm install, npm run build), and pushes the resulting build artifacts to an AWS S3 bucket for storage and distribution.

Logging: Logs are streamed in real-time during the build process to provide feedback and facilitate troubleshooting. These logs are critical for developers to understand the build process and quickly address any issues that arise.

API-Server:

Technology: Built with Node.js and Express, this server acts as the middleware facilitating communication between the frontend and the build-server.

Functionality: It handles API requests from the frontend, such as initiating a new build or querying the status of an ongoing build. It manages task queues for builds, ensuring that build requests are processed in an orderly and efficient manner.

Reverse-Proxy:

Technology: Implemented using Express, known for its high performance and configuration flexibility.

Functionality: The reverse-proxy serves as the gateway through which all external HTTP requests are routed. It dynamically maps requests to the appropriate project subdomains hosted on S3, ensuring that each user's request reaches the correct application version.

Domain name: The reverse proxy allows for projects to be streamed into their own subdomain, following a common naming convention of generated project as the subdomain. These subdomains were served only locally since free subdomains did not give us the control we needed.

This detailed breakdown describes the role of each component within the CloudBuilder architecture but also highlights their interactions and dependencies, providing a clear understanding of how the system operates as a cohesive unit.

Features

Detailed Feature Descriptions

The CloudBuilder project offers a variety of features designed to streamline the web application deployment process. These features combine usability, automation, transparency, and security, making the deployment process as intuitive and secure as possible.

Interactive User Interface:

Usability: The CloudBuilder interface, developed with Next.js, provides a clean and straightforward user experience. Users are greeted with a minimalistic layout where they can simply enter the GitHub project URL and initiate the build process.

Feedback Mechanisms: During the build process, the UI dynamically updates to show the progress of the build, including stages like initialization, building, and deployment. Any errors encountered during these stages are prominently displayed, allowing users to make necessary corrections or adjustments.

Result Display: Once the build and deployment are successfully completed, the interface provides a direct link to the deployed application, allowing users to immediately access and test their live web application.

Automated Build Pipeline:

Triggering Builds: Users can initiate builds directly through the user interface by entering the GitHub URL of their project. This action sends a request to the API-server, which then forwards it to the build-server.

Build Process: The build-server, operating within a Docker container, clones the repository, installs dependencies, and executes predefined build scripts. This process is fully automated, requiring no user intervention beyond the initial request.

Handling Outputs: After a successful build, the artifacts are automatically uploaded to AWS S3, where they are stored and served. This seamless flow ensures that the user's latest build is always available and deployable.

Log Streaming and Handling:

Real-Time Monitoring: Build logs are an essential part of the deployment process, providing insights into the build progress and issues. CloudBuilder streams these logs in real-time from the build-server to the frontend, enabling users to monitor the build as it happens.

Log Access: The logs are accessible via the user interface, where they can be reviewed during and after the build process. This feature is crucial for troubleshooting and optimizing the build configurations.

Security Measures:

Data Handling: Security is a paramount concern, especially when handling sensitive information such as API keys and source code. CloudBuilder uses encrypted channels for all data transmissions, ensuring that sensitive data remains secure between transfers from the client to the server and vice versa.

Environment Isolation: By using Docker containers, each build process is isolated in its own environment. This not only improves security by limiting potential breaches to individual containers but also ensures that the build environment is consistent and controlled.

These features of CloudBuilder not only enhance the user experience by providing a smooth, efficient, and secure deployment process but also ensure that the developers have the tools they need to manage and troubleshoot their applications effectively. This comprehensive feature set is designed to address the common challenges faced by developers in web application deployment.

Technical Details

Reproducible Steps

Environment Setup:

We developed our set of services for the application primarily in Javascript. This requires the installation and setup of Node and NPM as our recommend tooling (https://nodejs.org/en/download).

Each service in our application needs to be setup and deployed independently. This includes the frontend Next.js server, the API server, the build server, and the reverse proxy. We offer more detailed information on the Github repository with instructions in a README file per service folder. We also have sample env files per service if they need one.

To get the Next.js started:
1. Install all dependencies (`npm install`)
2. Run the application (`npm run dev`)

We offer instructions for yarn, pnpm, and bun based on user-preference as well in the README.

To get the build server started (On AWS):
1. Setup an AWS Elastic Container Registry (ECR) repository and push the built Docker image to the ECR repository. We recommend the instructions to build and push the Docker image provided by AWS shown after creating an ECR repository.
2. Setup an AWS Elastic Container Service (ECS) cluster and connect it to the docker image store on ECR. This will kick off a pull, setup, and execution based flow that is highly scalable.

To get the API server running:
1. Install all dependencies (`npm install`)
2. Run the application (`node index.js`)

To get the reverse proxy running:
1. Install all dependencies (`npm install`)
2. Run the application (`node index.js`)

Configuration Files:

We provide sample env files stating the environment variables needing to be setup.

To get the Next.js application configured, no environment variables needed.

To get the build server configured, AWS access key ID and AWS secret access key need to be initialized with your credentials.

To get the API server configured, the same AWS access key ID and AWS secret access key need to be initialized with your credentials.

To get the reverse proxy configured, no environment variables needed.

Troubleshooting Common Issues:

1. Ensure S3 bucket is publicly accessible.
2. Ensure ECS subnets are set accordingly.
3. Ensure all environment variables are setup and if any code changes are needed are possibly for unintentionally hard-coded

Externally Accessible Website

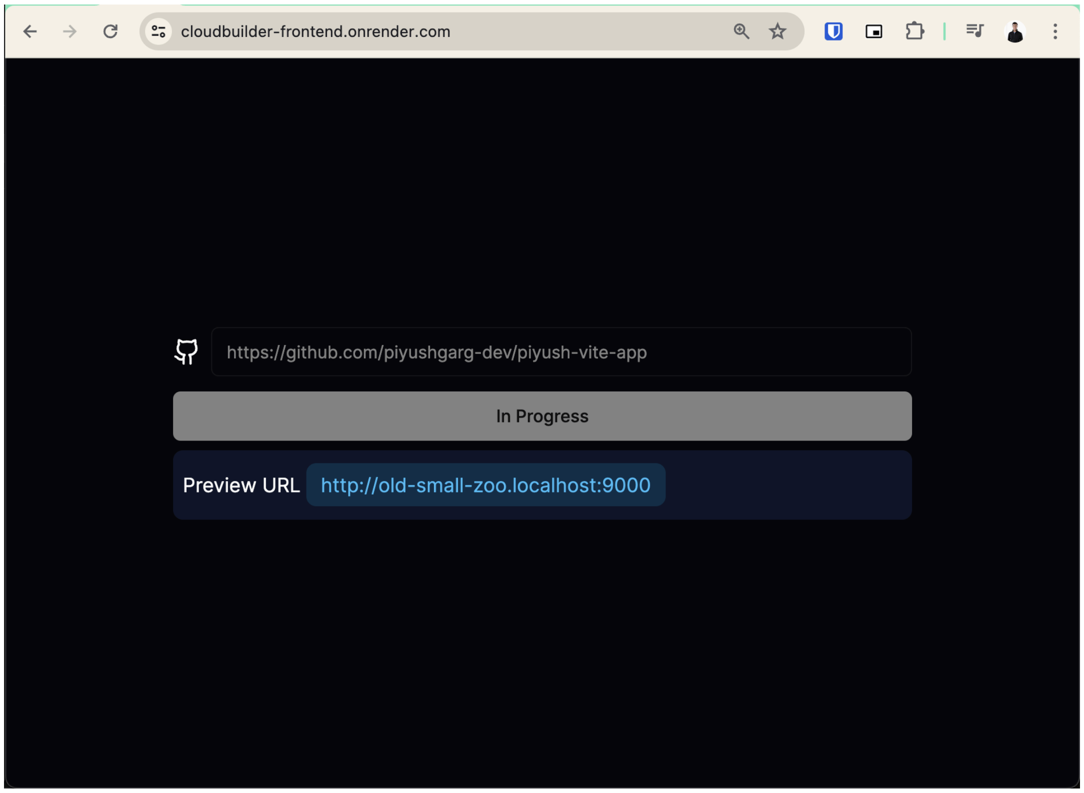
Access Details

To experiment with a variety of cloud deployment options, we tried different providers to understand how developer-based products are designed for efficiency while also giving us a deep understanding of the pros and cons of different cloud providers. We worked on hosting the entire collection of services, frontend, backend, and all supporting services onto Google Cloud, Amazon AWS, Vercel, DigitalOcean, and Render. We found that while some offered high flexibility, it often came at the expense of increased infrastructure ownership and management responsibilities. While more integrated solutions offered easy deployment options, they lacked in visibility, control, scalability, and cost.

Our current version of the frontend and API is hosted on Render, the backing services on AWS and the reverse-proxy is run locally. The URLs of the application are:

Frontend: https://cloudbuilder-frontend.onrender.com/
API server: https://cloudbuilder.onrender.com

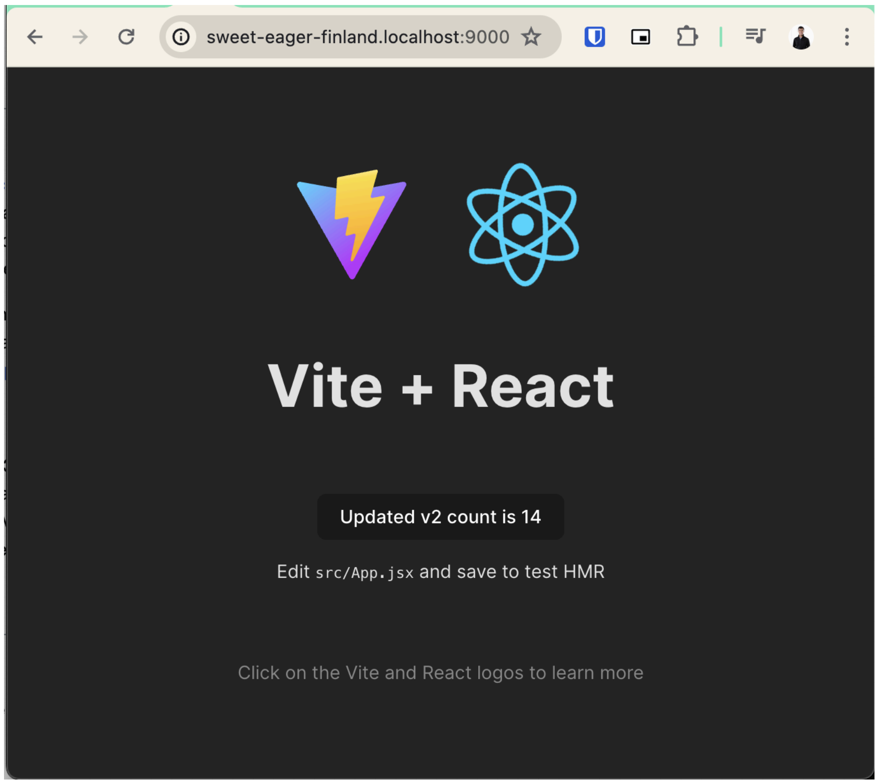
For quick testing and validation, we recommend setting up the reverse proxy server locally, which requires minimal setup. Upon setup, http://sweet-eager-finland.localhost:9000/ should be accessible with a sample React and Vite based web application running.


cloudbuilder-frontend.onrender.com

https://github.com/piyushgarg-dev/piyush-vite-app

Preview URL http://old-small-zoo.localhost:9000

Figure 2: Cloudbuilder landing page


sweet-eager-finland.localhost:9000

Vite + React

Edit `src/App.jsx` and save to test HMR

Click on the Vite and React logos to learn more

Figure 3: Sample web application deployed using Cloudbuilder

Links to Resources

GitHub Repository:

A link to the fully documented GitHub repository containing all source code, with instructions on cloning, installing, and running the project: https://github.com/jeetdas/cloudbuilder

Demo Video:

We provide an overview of the application developed and a short demo covering high-level features of the application here: https://youtu.be/St5S0britxY