CDEF Project Report

On

Sophius Deployment Platform

Name: Satyam Kumar Jha
Name: Ankit Sharma
Name: Fahed Ahmad
Name: Naveen Sharma
Name: Sundram Kumar
Under the Supervision of
Dr. Abhishek Singh

UNIVERSITY SCHOOL OF AUTOMATION AND ROBOTICS
GURU GOBIND SINGH INDRAPRASTHA UNIVERSITY
EAST DELHI CAMPUS, SURAJMAL VIHAR, DELHI-110032

Contents

Preface

The purpose of this report is to provide an in-depth analysis of the Deployment Platform, a modern cloud-based solution for deploying and managing web applications. This report details the architecture, technology stack, system components, and deployment workflows, along with considerations for scalability and security.

We aim to present a blueprint for understanding the Deployment Platform's capabilities while suggesting areas for further enhancements to adapt to emerging cloud trends.

Chapter 1

Introduction

The Vercel-like Deployment Platform is a modern solution designed to automate and simplify the deployment of web applications through a streamlined process that integrates GitHub repositories, AWS services, and real-time monitoring capabilities. This chapter introduces the platform's architecture, objectives, and core functionalities as demonstrated in the implementation.

1.1 Motivation

Traditional deployment processes often involve complex manual steps, leading to inefficiencies and potential errors. The Deployment Platform addresses these challenges by providing:

- Automated deployment pipeline triggered by GitHub repository submissions
- Real-time build logs through Socket.IO integration
- Distributed architecture utilizing Redis for pub/sub communication
- S3-based static file hosting with custom domain support
- Health monitoring system with automated checks

1.2 System Architecture

The platform consists of three main components:

1. **API Server**: Built with Express.js, handling:
    - HTTP and WebSocket connections on a unified port
    - GitHub webhook processing
    - Health monitoring endpoints
    - Real-time client communications
2. **Build Server**: Responsible for:
    - Cloning and building GitHub repositories
    - Publishing build logs through Redis
    - Uploading built assets to AWS S3
    - Managing build environment configurations
3. **Frontend Interface**: Implemented using Next.js, providing:
    - User-friendly deployment interface
    - Real-time build log visualization
    - Deployment status monitoring
    - Preview URL management

1.3 Key Features

- **Automated Deployment Flow**:
  - One-click deployment from GitHub repositories
  - Automatic subdomain generation using random-word-slugs
  - Real-time build progress monitoring
- **Real-time Communication**:
  - WebSocket-based live build logs
  - Redis pub/sub for distributed event handling
  - Immediate deployment status updates
- **Infrastructure Management**:
  - AWS S3 integration for static file hosting
  - Automated health checks every 14 minutes
  - CORS-enabled API endpoints
- **User Experience**:
  - Modern UI with GitHub integration
  - Live console output during builds
  - Instant preview URL generation

1.4 Technical Stack

The platform leverages modern technologies including:

- Node.js and Express for backend services
- Socket.IO for real-time communications
- Redis for distributed messaging
- AWS SDK for cloud infrastructure management
- Next.js for the frontend application
- GitHub API for repository integration

Chapter 2

Background

This chapter examines the architecture and components of our Vercel-like deployment platform, focusing on the technical foundations and design decisions that enable automated deployments.

2.1 System Components Overview

The platform consists of three main services:

- **API Server**: Built with Express.js and Socket.IO for real-time communication
- **Build Server**: Containerized build environment using Docker
- **S3 Reverse Proxy**: Custom proxy server for static file hosting

2.2 Technical Infrastructure

2.2.1 Core Technologies

The platform leverages several modern technologies:

- **Node.js Environment**
  - Express.js for HTTP routing
  - Socket.IO for real-time build logs
  - Redis for pub/sub messaging
- **Cloud Services**
  - AWS S3 for static file hosting
  - AWS ECS for container orchestration
  - Redis Cloud for distributed messaging
- **Build Infrastructure**
  - Docker containers for isolated builds
  - GitHub integration for source code access
  - Custom health monitoring system

2.3 Key Architectural Decisions

2.3.1 Real-time Communication

The platform implements a robust real-time communication system:

- Socket.IO for client-server WebSocket connections
- Redis pub/sub for inter-service communication
- Build log streaming for live deployment feedback

2.3.2 Build Process

The build process is designed for reliability and isolation:

- Containerized builds using Docker
- Automatic subdomain generation using random-word-slugs
- S3 integration for static file hosting
- Health monitoring with 14-minute interval checks

2.3.3 Frontend Integration

The platform provides a modern frontend experience:

- Next.js-based dashboard
- Real-time build status updates
- Live console output streaming
- Automatic preview URL generation

2.4 Deployment Workflow

The platform implements a streamlined deployment process:

1. GitHub repository URL submission
2. Automated container provisioning
3. Real-time build log streaming
4. Static file deployment to S3
5. Custom domain assignment
6. Health monitoring initialization

2.5 Monitoring and Reliability

The platform ensures reliability through:

- Automated health checks every 14 minutes
- Real-time error logging and notification
- Redis-based distributed system monitoring
- Containerized isolation for build processes

Chapter 3

System Architecture

The platform implements a distributed system architecture with three primary services working in concert to provide automated deployments and real-time monitoring.

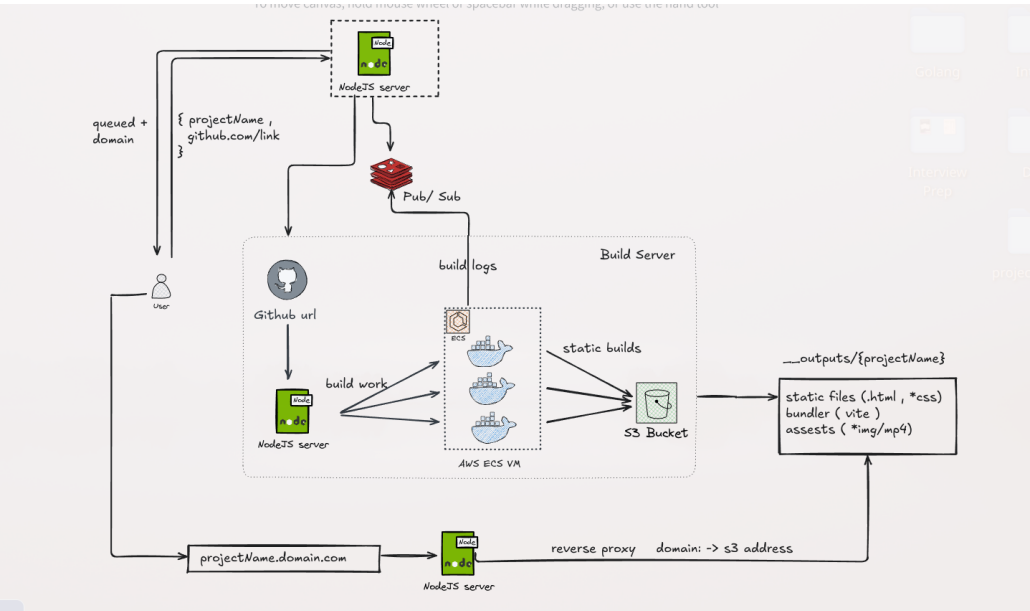


Figure 3.1: High-Level System Architecture

3.1 Core Services

3.1.1 API Server

Central coordination service built with Express.js that:

- Unifies HTTP and WebSocket traffic on a single port
- Manages GitHub repository submissions
- Coordinates with AWS ECS for container orchestration
- Implements health monitoring with 14-minute check intervals
- Facilitates real-time communication via Redis pub/sub

3.1.2 Build Server

Containerized build environment that:

- Executes in Docker containers on Ubuntu Focal

- Clones and builds GitHub repositories
- Uploads built assets to AWS S3
- Streams build logs through Redis channels
- Runs on Node.js 20.x with Git integration

3.1.3 S3 Reverse Proxy

Static file distribution service that:

- Routes requests based on subdomains
- Serves static assets from S3 storage
- Handles index.html fallbacks automatically
- Provides environment-based configuration

3.2 Communication Patterns

3.2.1 Deployment Flow

1. Frontend submits GitHub URL
2. API server validates and initiates ECS task
3. Build server clones, builds, and uploads to S3
4. Reverse proxy enables access via subdomain

3.2.2 Real-time Updates

- Socket.IO manages client-server communication
- Redis pub/sub handles inter-service messaging
- Build logs stream in real-time to frontend
- Health status updates every 14 minutes

3.3 Security Measures

- CORS-enabled API endpoints
- Containerized build isolation
- Environment-based configurations
- AWS IAM role-based access

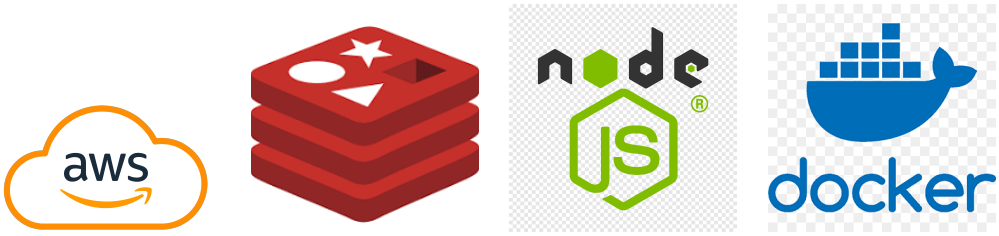
Chapter 4

Technology Stack

The Deployment Platform leverages a robust set of technologies to ensure reliability, scalability, and developer-friendliness. This chapter provides an in-depth analysis of the core technologies used.

4.1 Core Technologies

The platform is built using the following core technologies:



AWS Redis Node.js Docker

Figure 4.1: Core Technology Stack

4.1.1 Amazon Web Services (AWS)

AWS provides the foundational infrastructure for the platform. Key AWS services include:

- **Amazon ECS (Elastic Container Service):** Orchestrates containerized applications, ensuring scalability and fault tolerance.
- **Amazon S3 (Simple Storage Service):** Stores build artifacts securely, providing fast and scalable access to static assets.
- **Amazon VPC (Virtual Private Cloud):** Provides an isolated network environment, ensuring secure interactions between components.

4.1.2 Redis

Redis is used as an in-memory data store to manage:

- Real-time pub/sub messaging for build logs.
- Caching frequently accessed data for reduced latency.

4.1.3 Node.js

Node.js powers the backend API server, offering:

- High performance for handling concurrent API requests.
- Native support for WebSocket connections for real-time updates.

4.1.4 Docker

Docker simplifies the management of build environments by providing:

- Consistent build and runtime environments.
- Containerized execution of isolated build processes.

4.2 Additional Tools and Libraries

- **Nginx:** Serves as the reverse proxy for subdomain routing.
- **Framer Motion:** Adds interactive animations to the frontend interface.
- **WebSocket:** Enables bi-directional communication between the API server and the frontend.

Chapter 5

Deployment Workflow

This chapter demonstrates the deployment workflow through a practical example of deploying a Next.js application using our Vercel-like platform.

5.1 Example Deployment Scenario

Let's follow a complete deployment of a Next.js application:

5.1.1 Initial Submission

1. User visits the platform's frontend interface
2. Submits GitHub repository URL: `https://github.com/username/next-app`
3. Frontend validates URL format and initiates deployment process

5.1.2 Build Process Execution

```
> Initializing deployment...
> Generating project ID: elegant-purple-whale
> Cloning repository...
> Installing dependencies...
> Running build command: npm run build
> Uploading build artifacts...
> Configuring preview URL...
```

5.2 Detailed Workflow Diagram

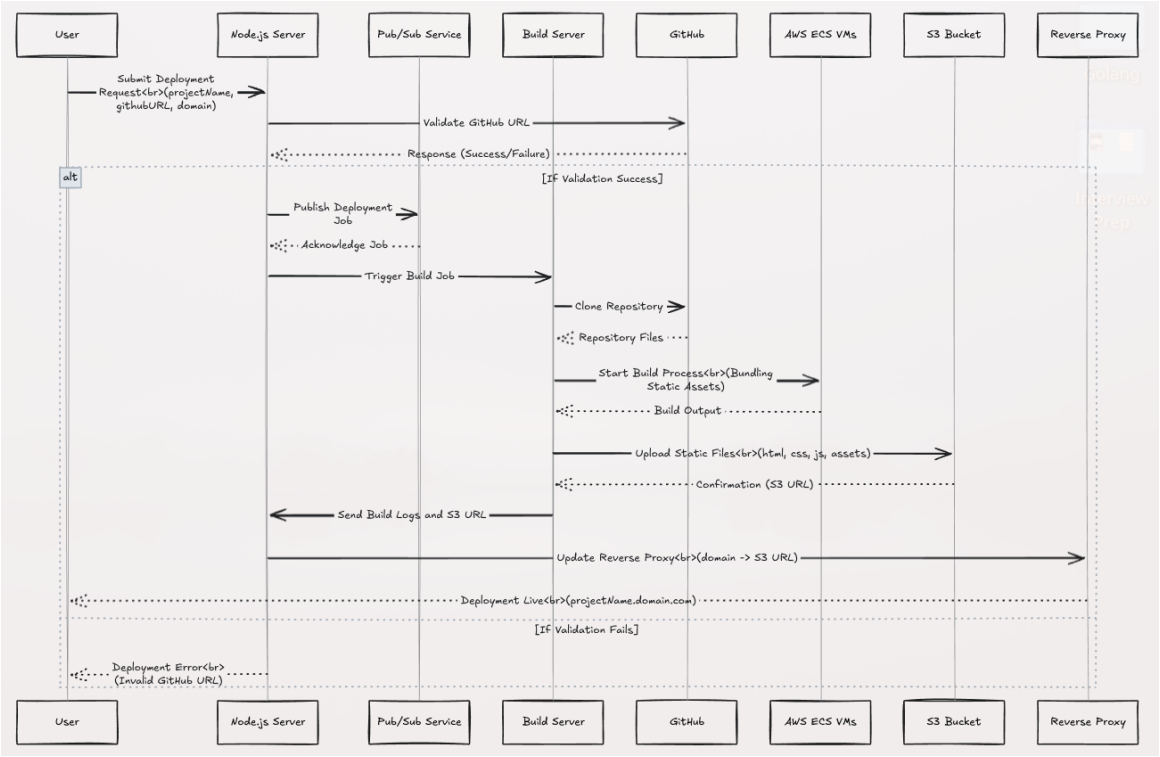


Figure 5.1: Deployment Workflow Diagram

5.3 Step-by-Step Breakdown

5.3.1 1. Repository Processing

When the user submits the GitHub URL:

- Socket.IO connection established for real-time updates
- API server generates unique slug (e.g., "elegant-purple-whale")
- ECS task initiated with repository details

5.3.2 2. Build Execution

The build server performs:

- Clones repository using `main.sh` script
- Installs project dependencies
- Executes build command
- Streams logs through Redis channels

5.3.3 3. Asset Deployment

Upon successful build:

- Static files uploaded to S3 bucket
- MIME types automatically detected
- Preview URL generated: `elegant-purple-whale.platform.com`

5.3.4 4. Monitoring Setup

Post-deployment:

- Health check endpoint configured
- 14-minute interval monitoring initiated
- Real-time console output available in frontend

5.4 User Interface Experience

During deployment, users see:

- Real-time build logs in a scrollable console
- Build progress indicators
- Preview URL upon completion
- Deployment status updates

5.5 Common Scenarios

5.5.1 Successful Deployment

```
> Deployment successful!
> Preview URL: https://elegant-purple-whale.platform.com
> Health check endpoint configured
> Monitoring system activated
```

5.5.2 Build Failure Handling

```
> Error: Build failed
> Cause: Missing dependencies in package.json
> Status: Deployment terminated
> Logs available for debugging
```

5.5.3 Health Check Monitoring

```
> Health check initiated
> Endpoint: /health
> Status: 200 OK
> Response: {"health": "ok"}
> Next check in 14 minutes
```

Chapter 6

Security Considerations

Security is a critical aspect of the Deployment Platform, ensuring user data, application artifacts, and the deployment process are safeguarded from unauthorized access and breaches.

6.1 Secure Communication

6.1.1 WebSocket Connections

The platform uses Secure WebSockets (wss://) to ensure encrypted bi-directional communication between the API server and frontend. This prevents interception of deployment logs or other sensitive data during transmission.

6.1.2 HTTPS Protocol

All web traffic is routed through HTTPS, providing encryption for data in transit. This is achieved using SSL/TLS certificates managed by AWS Certificate Manager.

6.2 Authentication and Authorization

6.2.1 AWS IAM Roles and Policies

Each component interacts with AWS services using specific IAM roles, enforcing the principle of least privilege:

- The build server has permissions only to read/write S3 buckets and initiate ECS tasks.
- The API server can access Redis and trigger build processes but cannot modify S3 artifacts directly.

6.2.2 Redis Authentication

Redis instances require authentication using a secure password. Additionally, Redis operates within the VPC to restrict access to internal components only.

6.3 Infrastructure Security

6.3.1 VPC Security Groups

Security groups define strict ingress and egress rules:

- The API server allows traffic only from the frontend and the build server.
- S3 buckets enforce access control lists (ACLs) to restrict public read/write access.

6.3.2 Network Isolation

The deployment platform operates within an isolated VPC, with subnets segregating public-facing components (e.g., frontend) from private ones (e.g., Redis, build server).

6.4 Data Security

6.4.1 Encryption

- **In Transit:** All data transmitted between components is encrypted using TLS.
- **At Rest:** Artifacts stored in S3 are encrypted using AES-256.

6.4.2 User Data Protection

The platform does not store sensitive information, such as GitHub credentials. OAuth tokens are securely passed through and never logged.

6.5 Monitoring and Alerts

- AWS CloudWatch monitors ECS tasks, API server logs, and S3 bucket activity.
- Alerts are triggered for suspicious activity, such as unauthorized API requests or failed authentication attempts.

Chapter 7

System Components

This chapter details the core components of our Vercel-like deployment platform.

7.1 Frontend Interface

Built with Next.js 15.0.3, the frontend provides deployment management capabilities.

7.1.1 Core Features

- **Repository Management:**
  - GitHub URL validation using regex pattern:

```
new RegExp(/^(?:https?:\/\/)?(?:www\.)?github\.com\/([^\/]+)\/([^\/]+)(?:\/)?
```

  - Real-time build status monitoring
  - Automatic subdomain generation
- **Real-time Updates:**
  - Socket.IO integration for live logs
  - Auto-scrolling log container
  - Build progress indicators

7.2 API Server

Express.js-based server orchestrating deployment processes.

7.2.1 Core Endpoints

Route	Method	Purpose
/project	POST	Initiates deployment
/health	GET	Service health check

Table 7.1: API Routes

7.2.2 Implementation Details

- **AWS Integration:**
  - ECS task management with Fargate

  - Configurable security groups
  - Subnet management across availability zones

- **Real-time Communication:**
  - Redis pub/sub for log streaming
  - Socket.IO for client updates
  - Project-specific channels

7.3 Build Server

Containerized build environment handling repository processing.

7.3.1 Core Services

- **Build Process:**
  - S3 integration for artifact storage
  - MIME type detection for uploads
  - Real-time log publishing via Redis

- **Environment Configuration:**

```
const s3Client = new S3Client({
    region: process.env.BUCKET_REGION,
    credentials: {
        accessKeyId: process.env.ACCESS_KEY,
        secretAccessKey: process.env.SECRET_KEY
    }
});
```

7.4 S3 Reverse Proxy

Custom proxy implementation for static file serving.

7.4.1 Implementation Details

- **Request Handling:**
  - Subdomain extraction and routing
  - Automatic index.html fallback
  - Path normalization

- **Code Example:**

```
app.use((req, res) => {
    const hostname = req.hostname;
    const subdomain = hostname.split('.')[0];
    const resolvesTo = `${BASE_PATH}/${subdomain}`;
    return proxy.web(req, res, {
        target: resolvesTo,
        changeOrigin: true
    });
});
```

7.5 Health Monitoring

Comprehensive system health checks across services.

7.5.1 Monitoring Points

- 14-minute interval health checks
- Service-specific endpoints
- Real-time status updates

7.6 Dependencies

- **Frontend Stack:**
  - Next.js 15.0.3
  - React 19.0.0-rc
  - Socket.IO Client 4.8.1
  - Tailwind CSS 3.4.1
- **Backend Stack:**
  - Express.js
  - AWS SDK
  - Redis 5.4.1
  - MIME Types 2.1.35

Chapter 8

Future Enhancements

The Deployment Platform is designed with scalability and adaptability in mind. This chapter outlines potential enhancements to improve its functionality, performance, and user experience.

8.1 Custom Domain Support

8.1.1 Overview

Users will have the ability to associate their projects with custom domains instead of subdomains. This will involve:

- DNS configuration assistance.
- Automated SSL certificate generation using Let's Encrypt.
- Domain validation through email or DNS records.

8.1.2 Implementation Plan

- Extend the API to accept and validate domain inputs.
- Integrate with AWS Route 53 for DNS management.
- Automate SSL provisioning via Certbot.

8.2 Build Cache Optimization

8.2.1 Current Challenges

Repeated builds for the same repository can lead to redundant computations, increasing build time and resource usage.

8.2.2 Proposed Solution

- Implement caching mechanisms for dependencies (e.g., npm, pip).
- Use Docker layer caching to avoid rebuilding unchanged parts of the image.
- Store and reuse previous build artifacts when appropriate.

8.3 Multiple Deployment Environments

8.3.1 Overview

Introduce support for staging, testing, and production environments to facilitate robust CI/CD workflows.

8.3.2 Key Features

- Environment-specific configuration files.
- Seamless promotion of builds from one environment to the next.
- Isolated resources for each environment.

8.4 CI/CD Pipeline Integration

8.4.1 Purpose

Enable users to connect the platform with existing CI/CD pipelines for continuous deployment.

8.4.2 Features

- Webhook support for build triggers.
- Integration with tools like Jenkins, GitHub Actions, and GitLab CI/CD.
- Real-time pipeline status updates on the dashboard.

8.5 Enhanced Monitoring and Analytics

8.5.1 Proposed Metrics

- Build success/failure rates.
- Average build time by project.
- Resource utilization trends.

8.5.2 Visualization Tools

- Dashboards displaying metrics in real time.
- Alerts for anomalies, such as increased build failures.

Chapter 9

Case Studies

This chapter presents real-world examples of static site deployments using our platform, demonstrating its core capabilities and limitations.

9.1 Case Study 1: Personal Portfolio

9.1.1 Project Overview

A frontend developer deploying a Next.js-based portfolio website.

9.1.2 Technical Implementation

- **Repository URL:** github.com/developer/portfolio
- **Build Process:**

```
// Automatic build steps
npm install
npm run build
// Output directory: /out
```

- **Deployment:** Automatic subdomain generation (e.g., elegant-purple-whale.platform.com)

9.1.3 Results

- Deployment time under 5 minutes
- Real-time build logs via Socket.IO
- Automatic static file serving through S3

9.2 Case Study 2: Technical Documentation

9.2.1 Project Overview

A static documentation site built with React and MDX.

9.2.2 Technical Details

```
// Socket.IO implementation for real-time logs
socket.emit("subscribe", `logs:${projectSlug}`);
socket.on("message", (message) => {
    const { log } = JSON.parse(message);
```

```
    console.log(`Build Status: ${log}`);
});
```

9.2.3 Key Benefits

- Automated builds on repository updates
- Health checks every 14 minutes
- S3-based content delivery

9.3 Case Study 3: Static Blog

9.3.1 Implementation Details

```
// S3 Configuration for Static Hosting
const s3Client = new S3Client({
    region: process.env.BUCKET_REGION,
    credentials: {
        accessKeyId: process.env.ACCESS_KEY,
        secretAccessKey: process.env.SECRET_KEY
    }
});
```

9.3.2 Deployment Process

- Automatic repository cloning
- Static asset compilation
- S3 upload with proper MIME types

9.4 Platform Limitations

9.4.1 Current Capabilities

- Static site deployments only
- GitHub repository integration
- Automatic subdomain assignment
- Real-time build logging

9.4.2 Not Supported

- Server-side rendering
- Database connections
- Custom domain configuration
- Environment variables management

9.5 Future Enhancements

- Build caching for faster deployments
- Custom domain support
- Environment variable configuration
- Advanced build configurations

Chapter 10

Monitoring and Analytics Dashboard

The platform provides a minimalist monitoring interface focused on deployment status and build logs.

10.1 User Interface

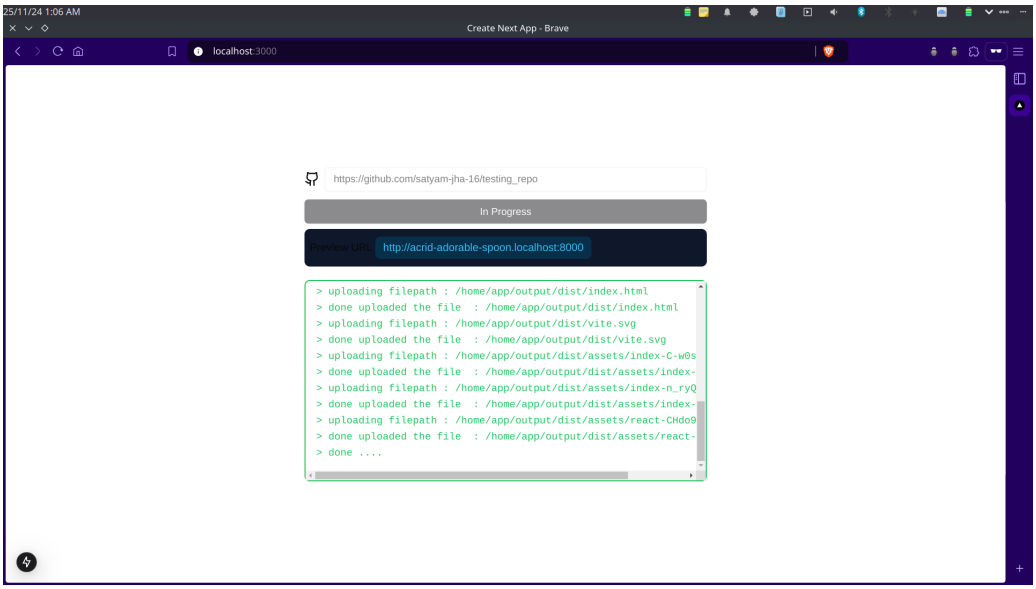


Figure 10.1: Project Deployment Monitor Interface

10.1.1 Core Components

- **Repository Input Form:**
  - GitHub URL validation with regex pattern:

    ```
    /^(?:https?:\/\/)?(?:www\.)?github\.com\/([^\/]+)\/([^\/]+)(?:\/)?$/
    ```

  - Project name input field
  - Deploy button with loading state
- **Build Log Display:**
  - Auto-scrolling log container
  - Real-time Socket.IO updates
  - Error message highlighting

10.2 Implementation Details

10.2.1 Frontend Components

```
const [repoURL, setURL] = useState<string>("");
const [logs, setLogs] = useState<string[]>([]);
const [loading, setLoading] = useState(false);
const [projectId, setProjectId] = useState<string | undefined>();
const [deployPreviewURL, setDeployPreviewURL] = useState<string | undefined>();
```

10.2.2 Real-time Updates

The system implements Socket.IO for live log streaming:

```
socket.emit("subscribe", `logs:${projectSlug}`);
const handleSocketIncommingMessage = (message) => {
    const { log } = JSON.parse(message);
    setLogs((prev) => [...prev, log]);
    logContainerRef.current?.scrollIntoView({ behavior: "smooth" });
};
```

10.3 Styling

The dashboard utilizes Tailwind CSS for responsive design:

- Form container with rounded borders
- Input fields with validation states
- Loading indicators
- Error message styling
- Auto-scrolling log container

10.4 Health Monitoring

- **Server Health:** 14-minute interval checks
- **Build Status:** Real-time progress updates
- **Error Handling:** Automatic error reporting and display
- **Redis Integration:** Log distribution via pub/sub

Chapter 11

Testing Strategy

11.1 Unit Testing

Ensures the reliability of individual components.

- **API Tests:** Validates endpoints for correct responses.
- **Utility Functions:** Tests reusable helper functions.

11.2 Integration Testing

Validates interactions between components.

- **API-Redis Communication:** Ensures real-time pub/sub messaging works as expected.
- **Frontend-Backend Flow:** Tests end-to-end deployment initiation and monitoring.

11.3 End-to-End Testing

Simulates real-world user scenarios to verify system behavior:

- **Build and Deployment Process:** Verifies that a GitHub repository can be deployed without errors.
- **Subdomain Routing:** Ensures deployed projects are accessible via assigned subdomains.

Chapter 12

Appendices

12.1 Example Deployment Request

Below is a sample request payload for initiating a deployment:

Listing 12.1: Deployment Request Format

```
{
    "gitURL": "https://github.com/username/repository",
    "slug": "my-project-name" // Optional, random slug generated if not provided
}
```

12.2 API Reference

Endpoint	Description
`POST /project`	Initiates a new deployment task. Returns project slug and preview URL.
`GET /health`	Health check endpoint to verify API server status.

Reference implementation:

12.3 Socket.IO Events

Event	Description
`subscribe`	Subscribe to project-specific log channel using format `logs:{projectSlug}`
`message`	Receive real-time build and deployment logs

Reference implementation:

12.4 Environment Variables

Variable	Purpose
BUCKET_REGION	AWS S3 bucket region
ACCESS_KEY	AWS access key ID
SECRET_KEY	AWS secret access key
REDIS_URI	Redis connection string
CLUSTER_ARM	ECS cluster ARN

TASK_ARM	ECS task definition ARN
SECURITY_GROUP	AWS security group ID

12.5 Build Process Steps

1. Clone GitHub repository
2. Install npm dependencies
3. Execute build command
4. Upload build artifacts to S3

Reference implementation:

12.6 Glossary

Term	**Definition**
ECS	AWS Elastic Container Service used for running build tasks
Fargate	Serverless compute engine for containers
Redis	In-memory database used for real-time log streaming
Socket.IO	Library enabling real-time bidirectional communication
S3	AWS Simple Storage Service for hosting static files

12.7 Dependencies

12.7.1 Frontend

- Next.js 15.0.3
- React 19.0.0-rc
- Socket.IO Client 4.8.1
- Tailwind CSS 3.4.1

12.7.2 API Server

- Express 4.21.1
- Socket.IO 4.8.1
- AWS SDK ECS Client
- Redis 5.4.1

12.7.3 Build Server

- AWS SDK S3 Client
- Redis 5.4.1
- MIME Types 2.1.35

Chapter 13

Conclusion

The Deployment Platform is a robust, scalable solution for modern web application deployment. Its features, such as real-time build logs, automatic subdomain assignment, and seamless GitHub integration, make it invaluable for developers seeking efficiency and reliability.

With the proposed enhancements and continuous improvements, the platform will evolve to meet the growing needs of development teams and organizations.

References

- Amazon Web Services Documentation. Available at: `https://aws.amazon.com/documentation/`
- Docker Documentation. Available at: `https://docs.docker.com/`
- Redis Official Website. Available at: `https://redis.io/`
- Node.js Official Website. Available at: `https://nodejs.org/`