A Cloud-Native Architecture For Real-Time Courier, Express, And Parcel (CEP) Tracking

Vachan Shetty
Networking and Communications Department
SRM Institute of Science and Technology, Kattankulathur, India
Chennai, India

Abstract - In the mature landscape of logistics and Courier, Express, and Parcel (CEP) services, this paper provides a comprehensive implementation guide for building cloud-native package tracking applications. Rather than proposing novel features, we present a detailed architectural approach using distributed systems and modern development practices. The implementation showcases a multi-portal system with synchronized views for customers, drivers, and administrators, built using Next.js, Vite, and NestJS. The application demonstrates effective use of containerization with Docker, orchestration with Kubernetes (EKS), and various AWS services including S3, RDS, and SNS. We detail the practical aspects of implementing real-time tracking through WebSocket connections, managing distributed data with MySQL and MongoDB, and establishing robust CI/CD pipelines. This paper serves as a technical blueprint for developers, emphasizing cloud-native best practices, scalable architecture patterns, and DevOps methodologies in the context of logistics applications.

Index Terms – Courier, Express and Parcel (CEP); Create, Read, Update, Delete (CRUD);Auth (Authentication), Database(DB), Application Programming Interface (API), Amazon Web Services (AWS), Kubernetes (K8s)

I. INTRODUCTION

In an era where e-commerce and logistics have become integral to daily life, robust package tracking systems are no longer innovative but essential infrastructure. While numerous commercial solutions exist in the market, there remains a gap in comprehensive technical literature that guides developers through the practical implementation of such systems using modern cloud-native architectures. This paper presents an instructive approach to building a real-time package tracking application, focusing on the architectural decisions, technical stack choices, and implementation patterns that enable scalable, distributed systems.

The proposed implementation leverages a cloud-native architecture deployed on Amazon Web Services (AWS), demonstrating why traditional monolithic approaches fall short in meeting the demands of contemporary logistics operations. By adopting a distributed microservices architecture, the system maintains synchronized yet distinct views for three key stakeholders: customers tracking their packages, drivers managing deliveries, and administrators overseeing operations. This separation of concerns, coupled with real-time synchronization, showcases the advantages of modern architectural patterns in handling complex, multi-user systems.

At its core, the application employs cutting-edge technology stacks that are increasingly becoming industry standards. The frontend utilizes Next.js for the customer portal and Vite for the administrative dashboard, while NestJS powers the backend services. The paper explores the rationale behind these choices, discussing how they facilitate rapid development, optimal performance, and maintainable codebases. The system's data layer demonstrates the strategic use of MySQL and Prisma ORM, illustrating when and why different database paradigms are appropriate for various aspects of logistics applications.

Rather than proposing novel algorithms or untested methodologies, this research's significance lies in its practical value as a comprehensive guide for full-stack developers venturing into cloud-native logistics applications. Through detailed examination of architectural decisions, technology stack selection, and cloud integration patterns, the paper provides insights into building resilient, scalable systems that can handle the complexities of real-time package tracking. Special attention is given to cloud deployment strategies, API design patterns, and the implementation of real-time features using WebSocket connections and event-driven architectures.

This work contributes to the field by bridging the gap between theoretical knowledge and practical implementation, offering developers a blueprint for building modern logistics applications while understanding the underlying architectural principles and trade-offs involved in cloud-native development.

II. SIGNIFICANCE

1. *Enhancing Customer Experience*:
   - *Real-Time Visibility*: By providing users with real-time tracking information and updates, the application improves transparency and trust in the delivery process. Users can monitor their packages at every stage, from pickup to final delivery.
     The application provides comprehensive package tracking through a responsive Next.js frontend, enabling users to monitor their deliveries from pickup to final destination. WebSocket connections ensure real-time updates without page refreshes.
   - *Multi-Channel Communication*: Integration with AWS Simple Notification Service (SNS) enables automated SMS notifications at key delivery milestones, keeping users informed through their preferred communication channels.

2. *Streamlining Delivery Operations*:
   - Driver Portal Integration: A dedicated Vite-powered driver interface streamlines package management by:
     - Providing real-time access to assigned deliveries
     - Enabling status updates through a progressive web application (PWA)
     - Supporting proof-of-delivery capture and upload to AWS S3
     - Maintaining synchronization with the main system through event-driven architecture

3. *Advanced Cloud-Native Architecture:*
   - Containerized Microservices:
     - Separate Docker containers for customer, driver, and admin interfaces
     - NestJS backend services containerized by domain (auth, tracking, notifications)
     - Container orchestration through Kubernetes (K8s) enabling:
       - Automated container health monitoring
       - Self-healing through ReplicaSets
       - Horizontal pod autoscaling based on load metrics
       - Load balancing across multiple pods

   - High Availability and Scalability:

     - Multi-AZ deployment in AWS for fault tolerance
     - Kubernetes cluster configuration with node groups across availability zones
     - Automatic scaling based on CPU utilization and request metrics
     - Database replication with MySQL primary-replica setup

4. *DevOps Excellence*
   - *Comprehensive CI/CD Pipeline:*
     - Automated testing and deployment workflows using GitHub Actions
     - Multi-stage builds with development, staging, and production environments

- Infrastructure as Code (IaC) using AWS CDK for consistent environment provisioning
- Automated container image building and pushing to Amazon ECR
- Zero-downtime deployments through K8s rolling updates
- Custom metrics collection for system health and performance

5. *Promoting Digital Transformation in Logistics*:

- *Innovative Use of Modern Technologies*: The application demonstrates how emerging technologies, cloud computing, event-driven and microservices architectures, can be effectively integrated into traditional logistics processes. This paves the way for further digital transformation initiatives within the logistics sector.
- Real-time data synchronization across all three views using WebSocket connections
- Type-safe development with TypeScript across the entire stack
- Modular architecture enabling independent scaling of components

III. LOGISTICS AND SHIPPING PROCESS

This figure illustrates the three main components of the logistics and shipping process: the First Mile, Middle Mile, and Last Mile.

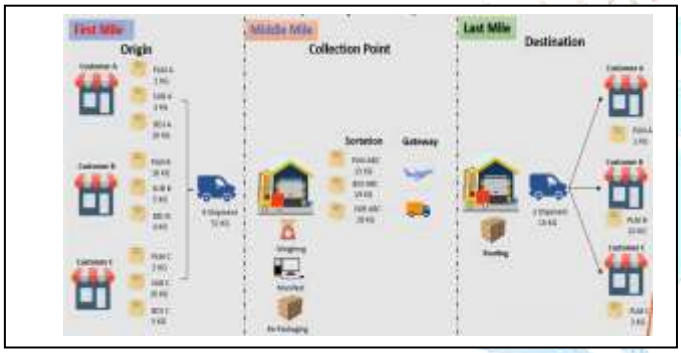

Fig. 1. Example of Shipping Process.

First Mile:
The First Mile represents the initial stage of the shipping process, starting at the origin point. In this image, we see:
- Shipments are collected from the customers, at the origin.
- A delivery vehicle picks up these shipments from the origin points.

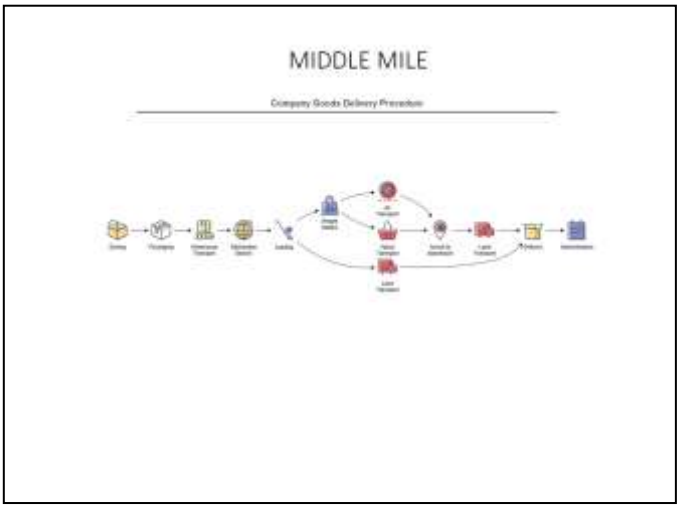

Fig. 2. Middle Mile. (*an illustration of the middle mile process*)

Middle Mile:

The Middle Mile shows the intermediary processes between pickup and final delivery:
- Shipments arrive at a Collection Point.
- Several operations occur here:
  1. Weighing of packages
  2. Creating a manifest (documentation)
  3. Re-packaging if necessary
  4. Sortation: Items are grouped into larger shipments
- The sorted shipments are prepared for various transportation methods (truck, airplane) through a gateway.

Last Mile:
The Last Mile represents the final stage of delivery to the end customers:
- Packages arrive at a distribution center.
- Routing occurs to determine the most efficient delivery paths.
- Delivery vehicles distribute the packages to their final destinations.
This project focuses primarily on the first mile and last mile phases.

IV. ARCHITECTURE

A. *Technology Stack Rationale:*

Before expounding on the application architecture, let us first give an overview of the technology stacks being utilized and why we chose to use them:

1. *Frontend Technologies*:

- Next.js for Customer Portal
    - Server-side rendering (SSR) capabilities improve initial page load times and SEO
    - Built-in API routes simplify backend integration
    - Automatic code splitting for optimal performance
    - Enhanced security through server-side operations
    - TypeScript support for type safety and better developer experience
    - Image optimization and built-in performance features
    - Simplified routing with file-system based routing
- Vite + Refine.dev for Admin Dashboard
    - Vite offers faster development server startup and hot module replacement
    - Refine.dev provides:
        - Built-in CRUD operations
        - Ready-to-use admin panel components
        - Data provider abstractions
        - Authentication providers
        - Advanced filtering and sorting capabilities

2. *UI Component Libraries:*
    - Shadcn UI for Customer Portal:
        - Highly customizable components
        - Accessible by default
        - Type-safe with TypeScript
        - Minimal bundle size
    - Material UI for Admin Dashboard:
        - Comprehensive component library
        - Enterprise-grade features
        - Consistent design language
        - Rich documentation and community support

3. *Backend Technologies*

- NestJS for API Services
  - TypeScript-first approach ensures type safety
  - Built-in support for dependency injection
  - Modular architecture promoting code organization
  - Decorators for simplified routing and middleware
  - Easy integration with various databases and ORMs
  - Built-in support for WebSocket connections
  - Microservices architecture support
- Prisma ORM
  - Type-safe database access with auto-generated types
  - Schema-first development approach
  - Automated database migrations
  - Superior query optimization compared to traditional ORMs
  - Built-in connection pooling
  - Support for multiple databases (MySQL, PostgreSQL, MongoDB)
  - Simplified database operations through Prisma Client
  - Rich filtering and relation handling
  - Real-time database events through Prisma middleware
- TypeScript
  - Static typing reduces runtime errors
  - Enhanced IDE support and developer tooling
  - Better code maintainability and refactoring
  - Improved team collaboration through explicit interfaces
  - Type checking during build process

4. *Authentication & Authorization*
   - Amazon Cognito
     - Scalable user management
     - Built-in security features
     - OAuth 2.0 and OpenID Connect support
     - Seamless integration with other AWS services
     - Multi-factor authentication capabilities
     - User pool management for different roles

5. *Data Visualization*
   - Recharts for Admin Analytics
     - React integration
     - Responsive design
     - Customizable components
     - Smooth animations
     - Support for real-time data updates
     - TypeScript compatibility

6. *Container Orchestration*
   - Docker
     - Consistent development environments
     - Isolated application containers
     - Simplified deployment process
     - Version control for containers
   - Kubernetes (EKS)
     - Automated container orchestration
     - High availability through pod replication
     - Self-healing capabilities
     - Horizontal scaling
     - Load balancing
     - Rolling updates with zero downtime

7. *Location Services*
    - Google Firebase & Maps Integration
        - Real-time location updates
        - Efficient geolocation tracking
        - WebSocket connections for live updates
        - Integration with Google Maps API for visualization
        - Cross-platform compatibility
        - Scalable real-time database

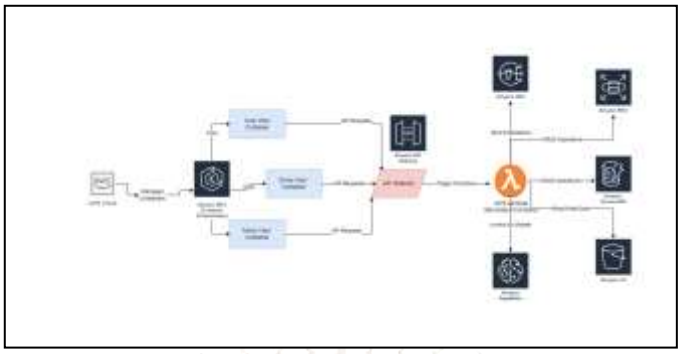

Fig. 3. Architecture Diagram (*a distributed, cloud-centric architecture*)

V. USER VIEW:

- The customer can track their parcels, get real-time notifications, and interact with the system to schedule deliveries or check status.

Key Technologies:

- Frontend: NextJs for each view, integrated with state management.
- Backend: NestJS with a microservices architecture.
- API Gateway: AWS API Gateway for routing requests.
- Database: MySQL, DynamoDB for data storage.
- Real-time Tracking: AWS SNS for notifications, WebSockets for real-time updates.

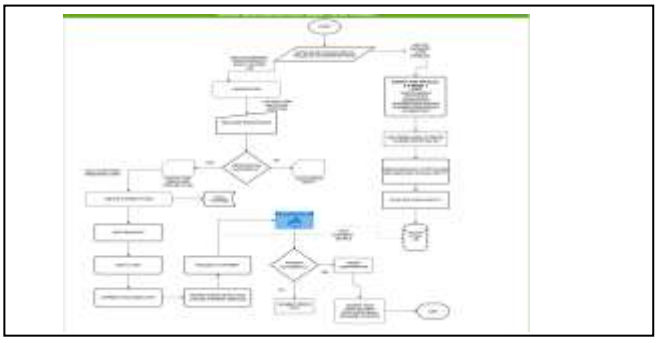

Fig. 4. Flowchart for User View (*user interaction workflow*)

The ER diagram below illustrates the relationships between the key entities in the system:

- User: Manages customers who use the application.
- Driver: Represents delivery personnel.
- Admin: Handles administrative operations.
- Package: Tracks parcel information.
- Delivery Status: Logs each update for the package.

- Route: Stores details about delivery routes.
- Notifications: Keeps records of customer or driver notifications.

Schema Design

- User Table: Contains user details, roles (user, driver, admin), and preferences.
- Package Table: Stores parcel details like weight, dimensions, destination, and current status.
- Delivery Status Table: Logs each update related to the package's journey.
- Route Table: Holds optimized routes based on current traffic, time, and package priority.
- Notification Table: Tracks notifications sent to users and drivers.
- Driver Table: Contains details about delivery drivers and their assigned routes.

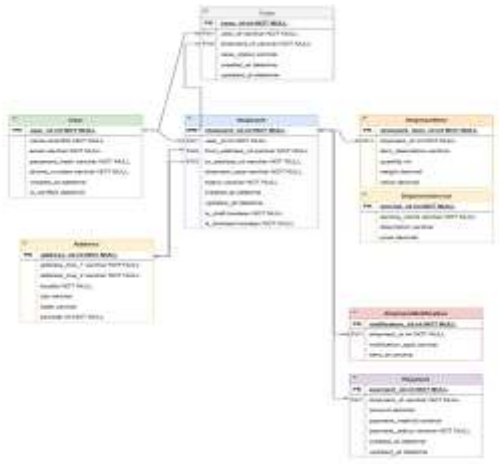

Fig. 5. ER Diagram for User View

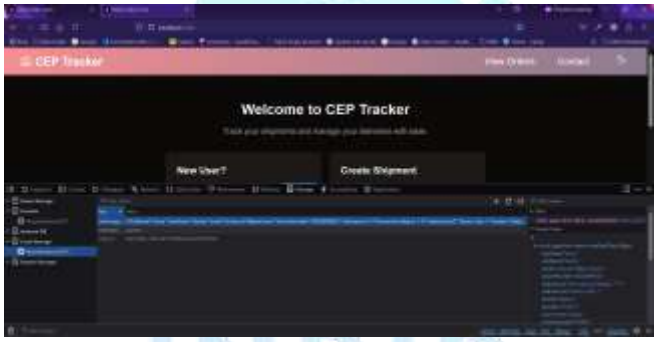

Fig. 6. Utilization of UUID generated in local storage to ID the user and allocate persistent storage for the multi-step registration form

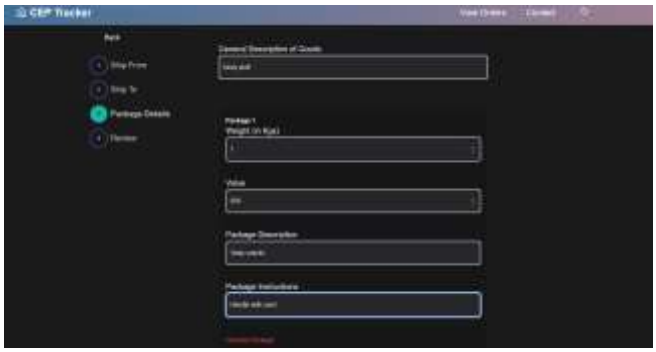

Fig. 7. Multi-Step Shipment Request Form For the User

- Once the shipment request form is completed, the user is directed to the Stripe payment gateway.
- After, the payment is complete, the request is sent to the admin view and the data is updated in the database.

VI. DRIVER VIEW

- The driver view essentially would enable the driver's real-time location to be tracked using Google Maps APIs, Google Firebase, in order to provide the user and admin a real-time view of the package status and ETA.
- It would be connected to the Nestjs backend as well.
- Since, we are running a distributed architecture, all three views would have to be run concurrently on different ports, say, ports 3001, 3002, 3003, when on localhost in development. The Nestjs backend could be on another port, say, port 3000.
- All three views must be synchronized, so that changes in one view are reflected in the other views.

VII. ADMIN VIEW

- The administrator has comprehensive control over the system, including monitoring deliveries, managing drivers, analyzing performance metrics, and handling customer service operations.

Key Technologies:

- Frontend: Vite and React with advanced data visualization libraries (e.g., Recharts) for analytics dashboards built using TypeScript

Fig. 8. Admin View Dashboard with Light/Dark Mode Toggle and real-time graphs for analytics

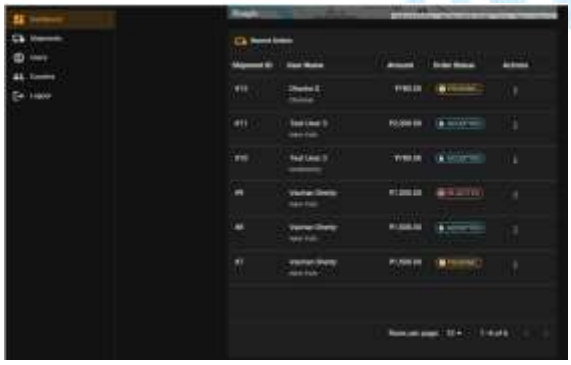

Fig. 9. Admin can accept or reject recent order from dashboard

- Backend: NestJS microservices architecture with specialized admin-only endpoints
- Access Control: AWS IAM for role-based access management with Amazon Cognito

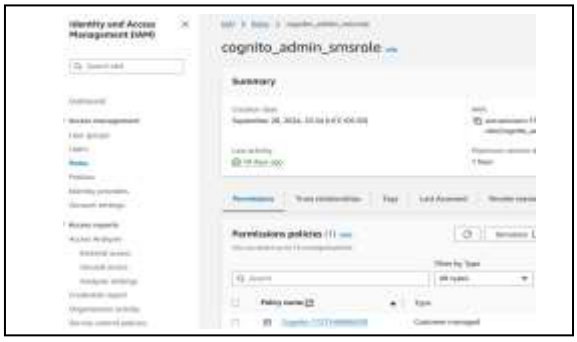

Fig. 10. Amazon Cognito Configuration for Admin Authentication

- Analytics:
  - Custom analytics pipeline for KPI monitoring
  - Uses Recharts for Visual Graphs
- Real-time Monitoring: WebSocket connections for live updates on system status
- Database Integration:
  - Primary: AWS RDS (MySQL) for transactional data

Architecture and Workflow:

- CRUD management for each entity: Courier, Shipments, Customers
- Real-time asynchronous updates
- Real-time Dashboard with visual insights
- Authentication using Amazon Cognito
- Distributed Architecture
- Admin View Connected with User View

Docker and Kubernetes:

- Added a Dockerfile in the Admin view backend to create a Docker image and containerize the application for future transfer to Amazon EKS on the cloud.
- We utilize docker compose to build and push the image to create the container in Docker Hub. We have one container for the Nestjs app, and one for the MySQL database.
- We are utilizing Kubernetes to create a pod cluster for the containers, and adding features such as horizontal scaling and ReplicaSets to improve availability and redundancy.
- Relevant Commands:
  a) docker build -t your_dockerhub_username/your_image_name:your_tag .
  b) docker login
  c) docker push your_dockerhub_username/your_image_name:your_tag
  d) docker-compose up –build -d

Fig. 11. Dockerfile to build a docker image

Fig. 12. docker-compose.yml file code snippet to build and run the docker container

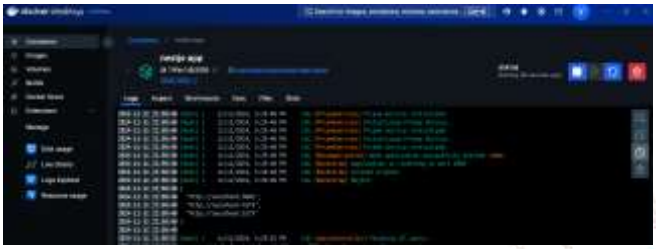

Fig. 13. Container logs as viewed in Docker Desktop

VIII. CONCLUSION

This research presents a practical implementation guide for building modern package tracking applications in the CEP industry, demonstrating how cloud-native architectures and distributed systems can effectively address the challenges of real-time delivery management. By leveraging a carefully selected technology stack—including Next.js for the customer portal, Vite with Refine.dev for the admin dashboard, and NestJS for backend services—the implementation achieves a balance between developer productivity and system performance.

The adoption of containerization through Docker and orchestration with Kubernetes (EKS) provides a robust foundation for scaling and managing the application in production environments. The integration of Firebase for real-time location tracking and WebSocket connections for live updates across different views demonstrates practical solutions for maintaining synchronized state in distributed systems.

The three-portal approach (customer, driver, and admin) showcases effective separation of concerns while maintaining data consistency through event-driven architecture. The implementation of Prisma ORM with a well-structured database schema illustrates best practices for managing complex data relationships in logistics applications.

Our DevOps approach, utilizing automated CI/CD pipelines and infrastructure as code, provides a blueprint for maintaining and scaling such applications in production. The detailed architectural decisions and technology choices documented in this paper serve as a valuable reference for developers building similar systems.

Future work could explore:

- Performance optimization techniques for handling increased scale

- Enhanced caching strategies for improved response times

- Implementation of advanced monitoring and alerting systems

- Integration with additional delivery service providers

- Expansion of the admin dashboard's analytical capabilities

This research contributes to the field by bridging the gap between theoretical knowledge and practical implementation, offering developers a comprehensive guide for building modern, cloud-native logistics applications. The documented approaches and architectural patterns provide a foundation for creating robust, scalable, and maintainable systems in the CEP domain.

IX. REFERENCES

1. "Next.js by Vercel - The React Framework for the Web," Vercel Inc., 2024. [Online]. Available: https://nextjs.org/docs

2. "Refine.dev Documentation - Build React-based CRUD Applications," Refine, 2024. [Online]. Available: https://refine.dev/docs

3. "TypeScript Documentation," Microsoft, 2024. [Online]. Available: https://www.typescriptlang.org/docs

4. "shadcn/ui Components," shadcn, 2024. [Online]. Available: https://ui.shadcn.com/docs

5. "Material UI - React components library," Material-UI, 2024. [Online]. Available: https://mui.com/material-ui/getting-started

6. "NestJS - A progressive Node.js framework," NestJS, 2024. [Online]. Available: https://docs.nestjs.com

7. "Vite - Next Generation Frontend Tooling," Vite, 2024. [Online]. Available: https://vitejs.dev/guide

8. "Prisma - Next-generation Node.js and TypeScript ORM," Prisma, 2024. [Online]. Available: https://www.prisma.io/docs

9. "Recharts - A composable charting library built on React components," Recharts, 2024. [Online]. Available: https://recharts.org/en-US/guide

Database & Storage

10. "MySQL Documentation," Oracle Corporation, 2024. [Online]. Available: https://dev.mysql.com/doc

11. "Firebase Documentation," Google, 2024. [Online]. Available: https://firebase.google.com/docs

Containerization & Orchestration

12. "Docker Documentation," Docker Inc., 2024. [Online]. Available: https://docs.docker.com

13. "Kubernetes Documentation," The Kubernetes Authors, 2024. [Online]. Available: https://kubernetes.io/docs/home

Cloud Services & DevOps

14. "Amazon S3 Documentation," Amazon Web Services, 2024. [Online]. Available: https://docs.aws.amazon.com/s3

15. "AWS Identity and Access Management Documentation," Amazon Web Services, 2024. [Online]. Available: https://docs.aws.amazon.com/iam

16. "Amazon Cognito Documentation," Amazon Web Services, 2024. [Online]. Available: https://docs.aws.amazon.com/cognito

17. "Amazon Simple Notification Service Documentation," Amazon Web Services, 2024. [Online]. Available: https://docs.aws.amazon.com/sns

18. "GitHub Actions Documentation," GitHub, 2024. [Online]. Available: https://docs.github.com/en/actions

Development Tools & Utilities

19. "JavaScript Reference," Mozilla Developer Network, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript

20. "Bash Reference Manual," GNU Project, 2024. [Online]. Available: https://www.gnu.org/software/bash/manual/bash.html

21. "draw.io Documentation," JGraph Ltd, 2024. [Online]. Available: https://www.drawio.com/doc

AI & Development Assistance

22. "ChatGPT Documentation," OpenAI, 2024. [Online]. Available: https://platform.openai.com/docs

23. "Claude Documentation," Anthropic, 2024. [Online]. Available: https://docs.anthropic.com

Additional References

24. M. Richards, "Software Architecture Patterns," O'Reilly Media, 2024.

25. S. Newman, "Building Microservices: Designing Fine-Grained Systems," O'Reilly Media, 2024.

26. R. Ford and C. K. Moe, "Building Real-time Web Applications with WebSocket," Manning Publications, 2023.

27. "Continuous Integration and Continuous Delivery," Martin Fowler's Blog, 2024. [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html

28. "The Twelve-Factor App," Adam Wiggins, 2024. [Online]. Available: https://12factor.net

29. "Clean Code: A Handbook of Agile Software Craftsmanship," Robert C. Martin, Prentice Hall, 2008.

30. "Cloud Native DevOps with Kubernetes," O'Reilly Media, 2024.