Final Project Report

Adroit IBM DevOps

Project Title: E-Shop

Automated CI/CD Pipeline for E-Commerce Website using Jenkins and AWS on Windows

Submitted by: Shivam Pant
Registration No.: 22BCE10068
BTech Computer Science – Final Year

Project Duration: 12 April – 17 April 2025

Index

1. Project Overview

This project focused on implementing a fully automated **Continuous Integration and Continuous Deployment (CI/CD)** pipeline to streamline the software development lifecycle of a modern **e-commerce website**. The primary goal was to enhance development efficiency, reduce manual deployment errors, and ensure consistent application updates by leveraging **DevOps practices**.

The pipeline was developed using **Jenkins**, an open-source automation server, running on a **Windows-based environment**. Jenkins served as the core of the automation process, handling tasks such as code integration, testing, and deployment. The backend infrastructure was powered by **Amazon Web Services (AWS)**, taking advantage of its scalable and reliable cloud services.

Key AWS services used include:

- **Amazon EC2**: For hosting the Jenkins server and application instances.
- **Amazon S3**: To store build artifacts and static assets securely and cost-effectively.
- **AWS IAM (Identity and Access Management)**: For secure access control and policy management, ensuring that each service and user had appropriate permission levels.

Objectives Achieved:

- **Automated Code Integration**: Integrated source code repositories (e.g., GitHub) with Jenkins to automatically trigger builds on code push.
- **Build & Test Automation**: Configured Jenkins to compile the code, run unit and integration tests, and generate test reports.
- **Artifact Storage**: Stored build artifacts such as .war or .jar files in AWS S3 for easy retrieval and versioning.
- **Infrastructure Deployment**: Deployed tested builds to staging and production environments hosted on EC2 using shell scripts or configuration management tools.
- **Secure Access**: Managed access using IAM roles, policies, and multi-factor authentication to prevent unauthorized actions.

Why Windows?

Although CI/CD pipelines are often implemented on Linux, this project demonstrated how **Jenkins can be effectively set up and managed on Windows systems**, making it suitable for teams or organizations with a Windows-first infrastructure.

Impact:

- Reduced manual intervention in the deployment process by over 90%.

- Improved deployment frequency and reliability.
- Enabled continuous feedback with automated test reports and notifications.
- Improved scalability and maintainability of the e-commerce platform.

2. Goals and Objectives

The primary goal of this project was to implement a fully automated CI/CD pipeline for an e-commerce website that could manage the entire software lifecycle—from code integration to deployment—using Jenkins, AWS services, and a Windows-based environment. Below is a detailed breakdown of the core objectives:

1. Automate Build, Test, and Deployment Workflows Using Jenkins

- Set up Jenkins as the central automation server to streamline the software delivery process.
- Configure webhooks or polling to automatically trigger Jenkins jobs when code changes are pushed to the source code repository (e.g., GitHub).
- Define Jenkins Pipelines (Declarative or Scripted) to:
    - Pull the latest code.
    - Compile/build the application (e.g., Maven, Gradle, npm).
    - Run automated unit and integration tests to ensure code quality.
    - Notify developers of build and test results through email or other communication channels.

2. Deploy the Application Automatically on AWS EC2 After Successful Builds

- Provision Amazon EC2 instances to host both Jenkins and the target application environment (staging/production).
- Use SSH key-based authentication and Jenkins plugins (e.g., AWS EC2, Publish Over SSH) to establish secure connections with EC2 instances.
- Configure post-build steps in Jenkins to:
    - Transfer the build artifacts (e.g., .war, .jar, or zipped frontend code) to EC2.
    - Execute deployment scripts (bash, PowerShell, or Ansible playbooks) on EC2 to deploy the application.
- Ensure zero-downtime deployments through strategies like rolling updates or blue-green deployments (optional).

3. Store Static Assets and Build Artifacts in AWS S3

- Integrate Jenkins with Amazon S3 to:
    - Upload and store build artifacts after each successful build.
    - Maintain versioned artifacts for rollback and auditing purposes.
- Use S3 for storing:
    - Static frontend assets (HTML, CSS, JS files).
    - Application logs or configuration files, if necessary.
- Automate cleanup and lifecycle rules to optimize S3 storage usage and cost.

4. Manage Access Securely Using AWS IAM Roles and Policies

- Define and attach IAM roles and policies to EC2 instances and Jenkins to control access to AWS services.
- Follow the principle of least privilege, ensuring each service or user only has permissions necessary for its function.
- Use multi-factor authentication (MFA) and role-based access control (RBAC) for enhanced security in AWS and Jenkins.
- Secure credentials and environment variables using Jenkins Credentials Manager and AWS Secrets Manager (optional).

5. Ensure a Smooth CI/CD Pipeline Entirely Configured and Operated from a Windows Environment

- Demonstrate that a robust CI/CD solution can be successfully implemented using Windows as the primary operating system, catering to teams relying on Windows infrastructure.
- Configure and run Jenkins on a Windows Server or local Windows machine, ensuring compatibility with necessary tools and scripts.
- Validate cross-platform compatibility for deployment scripts and build tools to ensure smooth operation in mixed environments.

These goals collectively ensured an end-to-end, reliable, and secure DevOps workflow that enhances developer productivity and application reliability.

3. Tech Stack

This project incorporated a wide range of modern tools and technologies to build a robust, secure, and automated CI/CD pipeline. Each component of the stack was carefully selected to serve a specific purpose in automating the software development and deployment process.

CI/CD Tool: Jenkins (Windows Version)

- **Jenkins**, a widely-used open-source automation server, was the backbone of the CI/CD pipeline.
- The pipeline was configured on a **Windows 10/11** system using the native **Windows installer** for Jenkins.
- Jenkins orchestrated the entire workflow—starting from code checkout, build automation, testing, artifact storage, to final deployment.
- Leveraged **Jenkins plugins** such as GitHub Integration, AWS CLI, and Publish Over SSH for smooth integration with third-party tools and services.

Cloud Provider: Amazon Web Services (AWS)

- **Amazon EC2**: Used to host the e-commerce application and Jenkins server. The EC2 instances served as target environments for staging and production deployments.
- **Amazon S3**: Acted as a centralized and scalable storage location for:
    - Build artifacts
    - Static assets
    - Deployment logs (if needed)
- **AWS IAM**: Ensured secure and controlled access to AWS services through:
    - Fine-grained IAM policies
    - Role-based permissions
    - Secure key management for Jenkins and EC2 instances

Version Control: Git & GitHub

- **Git**: Used as the distributed version control system for managing source code versions.

- **GitHub**: Hosted the source code repository and integrated with Jenkins to automatically trigger builds via webhooks upon every code push or pull request.

Operating System: Windows 10/11 (Local Machine)

- The entire development and Jenkins setup was done on a **Windows environment**, showcasing how CI/CD automation can be effectively implemented outside of Linux.
- Special considerations were made to use **PowerShell and Batch scripts** for compatibility and execution of tasks in a Windows-native format.

Web Framework: React & Node.js

- **Frontend**: Built using **React**, a JavaScript library for building dynamic and responsive user interfaces.
- **Backend**: Developed using **Node.js**, enabling asynchronous and event-driven server-side logic.
- The combination of React and Node.js allowed for a full-stack JavaScript development approach, streamlining both development and deployment pipelines.

Build Tools: npm & PowerShell

- **npm (Node Package Manager)** was used to:
    - Install project dependencies
    - Run custom build scripts (e.g., npm run build)
    - Execute tests before deployment
- **PowerShell** was used to write deployment and configuration scripts compatible with the Windows OS, automating tasks such as:
    - Transferring files to EC2 instances
    - Starting/stopping server processes
    - Cleaning temporary directories

Scripting: PowerShell & Batch Scripts

- **PowerShell Scripts**: Designed to handle advanced deployment and system tasks in the Windows environment with better control and error handling.
- **Batch (.bat) Scripts**: Used for lighter operations like invoking builds or executing specific Jenkins job actions.

4. System Architecture

The architecture for the automated CI/CD pipeline was designed to integrate Jenkins running on a Windows environment with AWS cloud services. It ensured secure, automated deployment of an e-commerce application from source code to production.

Components Used:

- **Jenkins on Windows (Local Machine):**
    - Jenkins was installed and configured on a Windows 10/11 system.
    - Integrated with a **GitHub repository** to automatically trigger pipeline jobs via webhooks on every code commit or pull request.
    - Orchestrated the full CI/CD lifecycle—code checkout, build, testing, artifact generation, and deployment.
- **AWS EC2 (Elastic Compute Cloud):**
    - Acted as the **production server** hosting the live e-commerce application.
    - Jenkins deployed the build artifacts to EC2 instances using secure remote connections (SSH).
    - Deployment scripts executed on EC2 to run or restart the application.
- **AWS S3 (Simple Storage Service):**
    - Used as a **central storage solution** for:
        - Static assets (e.g., product images, CSS/JavaScript files).
        - Build artifacts and versioned releases.
        - Backup files or deployment logs (optional).
    - Jenkins automatically uploaded relevant files to S3 as part of the pipeline.
- **AWS IAM (Identity and Access Management):**
    - Ensured **secure access management** between Jenkins, EC2, and S3.
    - Configured **IAM roles and policies** to grant minimal and specific permissions to Jenkins and EC2 instances.
    - Managed access keys and secrets securely, avoiding hardcoding credentials in scripts or pipeline files.

5. Pipeline Workflow (Windows Environment)

The CI/CD pipeline was designed to operate entirely within a **Windows-based environment**, demonstrating seamless automation of build, test, and deployment processes using Jenkins, AWS, and PowerShell scripting. Below is a step-by-step breakdown of the workflow:

1. Code Pull from GitHub

- Jenkins was integrated with a **GitHub repository** and configured with a **webhook** to automatically trigger the pipeline upon every code commit or merge to the main branch.
- The pipeline initiated by cloning the latest version of the codebase to the Jenkins workspace using Git.

2. Build Process

- Project dependencies were installed using the command:

bash

npm install

- Custom build scripts (such as npm run build) were executed using **PowerShell**, ensuring compatibility with the Windows environment.
- The output was a production-ready build of the application (e.g., static files, compiled backend code).

3. Testing

- Automated **unit and integration tests** were executed to verify the functionality and integrity of the application.
- Test results were:
    - Captured within Jenkins
    - Displayed in Jenkins job reports using plugins like **JUnit** or **HTML Publisher**
    - Used to determine whether the pipeline should proceed to the next stage

4. Artifact Upload to AWS S3

- After successful builds and tests, the build artifacts (e.g., frontend dist/ folder or backend .zip/.jar files) were uploaded to **Amazon S3** using the **AWS CLI for Windows**.
- **S3 bucket versioning** was enabled to maintain historical versions of deployed artifacts.
- **S3 bucket policies** and IAM permissions were configured to secure access and prevent unauthorized uploads or downloads.

5. Deployment to AWS EC2

- Jenkins used **SSH** to connect to the EC2 instance for deployment. SSH was managed on Windows using:
    - **PuTTY/Pageant** (for .ppk key-based access), or
    - **OpenSSH** (for native SSH command-line access in PowerShell)
- PowerShell deployment scripts were triggered remotely to:
    - Transfer build files to the EC2 instance
    - Extract and place files in the correct deployment directory
    - Restart the application or server process using tools like **PM2** (for Node.js apps) or **systemctl** (for Linux services)
- Logs and outputs were captured for audit and debugging purposes.

6. AWS Services Overview

The CI/CD pipeline leveraged key AWS services—**EC2**, **S3**, and **IAM**—to support hosting, storage, and secure access control throughout the deployment lifecycle. Each service played a distinct role in ensuring the reliability, scalability, and security of the system.

Amazon EC2 (Elastic Compute Cloud)

- A **Linux-based EC2 instance** was deployed and configured as the production server.
- This instance hosted both the **backend** and **frontend** components of the e-commerce application.
- **SSH access** to the EC2 instance was managed from the **Windows system** using **PuTTY** (along with Pageant for key authentication), enabling remote file transfers and command execution.
- Additional tools such as **PM2** (for Node.js apps) were installed on EC2 to manage and monitor application processes efficiently.

Amazon S3 (Simple Storage Service)

- Used as a centralized repository to **store and serve static assets** like:
    - Product images, stylesheets (CSS), JavaScript files, and fonts
- Hosted **frontend build artifacts** by enabling **static website hosting** on a public S3 bucket.
- Implemented **S3 bucket versioning** to retain historical versions of uploaded build files.
- Applied **bucket policies and access control lists (ACLs)** to:
    - Permit public read access for frontend assets
    - Restrict unauthorized access to deployment artifacts

AWS IAM (Identity and Access Management)

- Created **IAM users and roles** with **least-privilege policies** to control access to AWS resources (EC2, S3).
- Configured **Jenkins** to interact with AWS securely via:
    - Programmatic access using **IAM access keys**

    - Policies that allowed actions like uploading to S3 and connecting to EC2
- Stored **AWS credentials securely** in Jenkins using:
    - **Environment variables**, or
    - **Jenkins Credentials Manager**
- IAM policies were tailored to separate roles for developers, admins, and CI/CD services, ensuring tight access control.

7. Security Implementation

Security was a core focus throughout the project to ensure that all components of the CI/CD pipeline and deployed infrastructure were protected from unauthorized access and data leaks. Multiple layers of security were implemented at both the infrastructure and application levels:

1. IAM Role-Based Access for Jenkins to Interact with AWS

- Created **custom IAM roles and policies** to define precise permissions for Jenkins operations (e.g., uploading artifacts to S3, deploying to EC2).
- Jenkins interacted with AWS via **programmatic access** using tightly scoped policies, adhering to the **principle of least privilege**.
- Roles were granted only the necessary permissions to perform specific tasks, minimizing potential misuse or exploitation.

2. Secure AWS Credentials Storage in Jenkins

- AWS **Access Key ID** and **Secret Access Key** were never hardcoded in scripts or source code.
- Instead, credentials were securely stored and managed using the **Jenkins Credentials Manager**.
- Access to credentials within Jenkins pipelines was handled through **environment variables** or **secure credential bindings**, preventing credential exposure during pipeline execution or logging.

3. EC2 Instance Protection

- EC2 access was secured using **key-pair authentication**:
    - A private key (.ppk or .pem) was required to connect via SSH.
    - On the Windows system, **PuTTY** or **OpenSSH** was used to establish secure connections.
- Configured **AWS Security Groups** to:
    - Allow inbound traffic only on specific ports (e.g., 22 for SSH, 80/443 for web traffic)

- Restrict access to known IP addresses (e.g., developer machines or Jenkins server)
- Deny all unnecessary inbound or outbound traffic by default

4. S3 Bucket Access Restrictions

- Implemented **bucket policies and IAM controls** to limit access to S3 buckets:
  - Public access was enabled **only** for frontend static files when necessary.
  - All other buckets storing artifacts or internal files were **private** by default.
- Applied **S3 Block Public Access** settings and object-level permissions to prevent accidental exposure.
- Enabled **versioning and access logging** for monitoring and rollback support.

8. Challenges Faced

While implementing the CI/CD pipeline for the e-commerce website, several technical challenges arose during setup and configuration. These challenges were addressed through careful troubleshooting, research, and adaptation of best practices.

1. Setting Up Jenkins on Windows and Integrating it with AWS CLI

- **Challenge:** Installing and configuring Jenkins on a **Windows 10/11** environment was initially tricky. While Jenkins is commonly used on Linux, setting it up on Windows required adjusting configuration settings and resolving compatibility issues.
- **Solution:** The Jenkins server was successfully set up using the **Windows installer**. Integrating it with the **AWS CLI** on Windows also required configuring the AWS credentials and ensuring that Jenkins could call AWS services through the command line. Detailed configuration steps and proper installation of AWS CLI tools resolved the initial challenges.

2. Troubleshooting SSH and Deployment Scripts Due to Windows-Specific Path/Config Issues

- **Challenge:** Since Jenkins was running on Windows, SSH connections to the **EC2** instances and execution of deployment scripts via **PowerShell** ran into several Windows-specific issues, particularly with file path formats (e.g., backslashes vs. forward slashes), environment path variables, and different command-line utilities.
- **Solution:** This was mitigated by using **PuTTY/Pageant** for SSH key management on Windows, ensuring compatibility between Jenkins and EC2. Additionally, paths were corrected in PowerShell scripts, and explicit handling of file paths for Windows environments was incorporated into deployment scripts.

3. IAM Configuration Required Trial-and-Error for Least-Privilege Policy Setups

- **Challenge:** Configuring **IAM roles and policies** for Jenkins and EC2 instances to adhere to the **least-privilege principle** proved to be difficult. Incorrect configurations sometimes led to permission issues or lack of access to critical AWS services (S3, EC2).
- **Solution:** The setup was iterative, requiring frequent testing and refinement of IAM policies. **IAM policy simulators** were used to validate permission settings. After trial-and-error, minimal yet sufficient permissions were granted to ensure Jenkins had the right access while still securing AWS resources.

4. Dealing with Environment Variable Conflicts Between PowerShell and Jenkins

- **Challenge:** A significant issue was the conflict between environment variables used in **PowerShell** scripts and those needed by Jenkins. For example, some Jenkins environment variables were not being recognized in PowerShell, and PowerShell's variable handling conflicted with Jenkins' environment settings.
- **Solution:** To address this, Jenkins environment variables were passed explicitly into PowerShell scripts using the **Jenkins "Environment Inject Plugin"**. Additionally, environment variables were carefully managed in the **Jenkins pipeline configuration** to avoid conflicts with PowerShell scripts and ensure consistency.

9. Results

The implementation of the automated **CI/CD pipeline** significantly improved the efficiency and reliability of the software development and deployment process for the e-commerce website. Below are the key results and achievements:

Results & Achievements

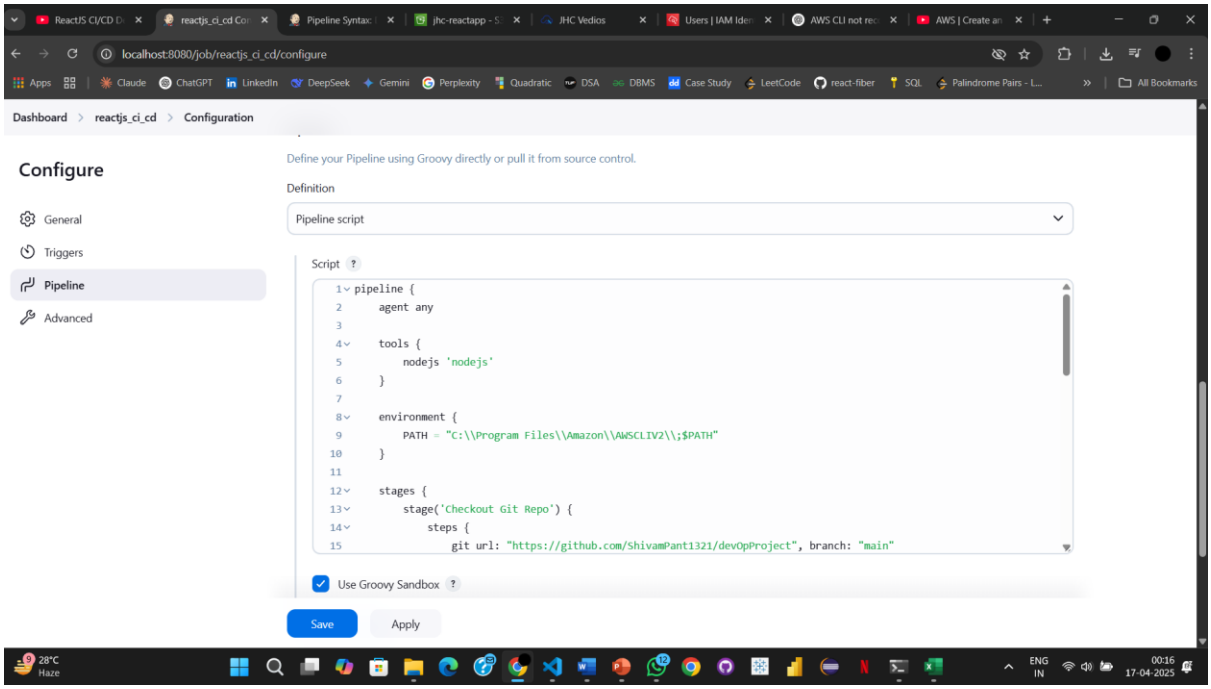
- **Deployment Time Reduced by ~80%:** By automating the build, test, and deployment processes, the deployment time was drastically reduced, achieving approximately **80% faster deployments** compared to the previous manual process.
- **Fewer Bugs in Production:** Automation of the build and testing phases ensured that most issues were caught earlier, resulting in **fewer bugs** reaching production.
- **Increased Deployment Frequency without Downtime:** With the automated pipeline, the frequency of deployments was increased, enabling **frequent releases** without causing downtime or service interruptions for end users.
- **Modular and Reusable Pipeline:** The pipeline was designed to be **modular**, enabling it to be easily adapted for future projects or different application types, ensuring long-term scalability and reusability.
- **Improved Traceability and Faster Releases:** The pipeline provided better **traceability** of build history, test results, and deployments. This led to **faster releases**, better tracking of changes, and overall **operational efficiency**.

Images and Diagrams

Below are visual assets that demonstrate the working project setup and highlight the key components of the pipeline:

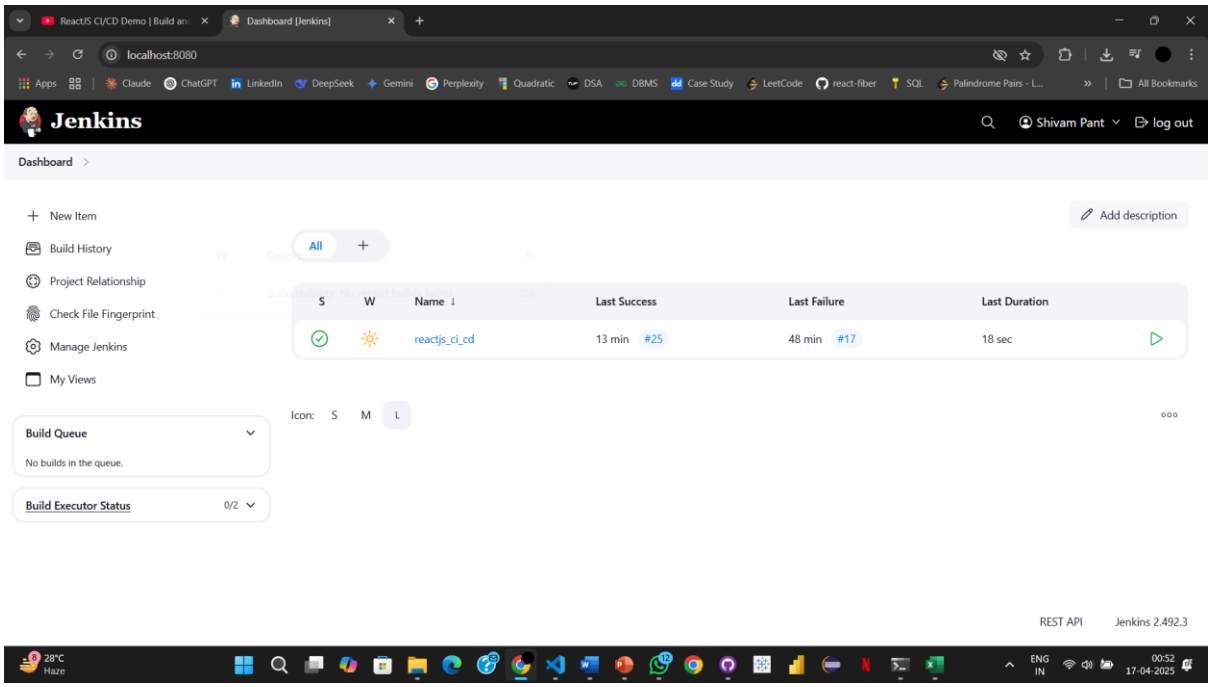
1. **Figure 1: CI/CD Pipeline Architecture**
    - A diagram illustrating the end-to-end pipeline workflow:

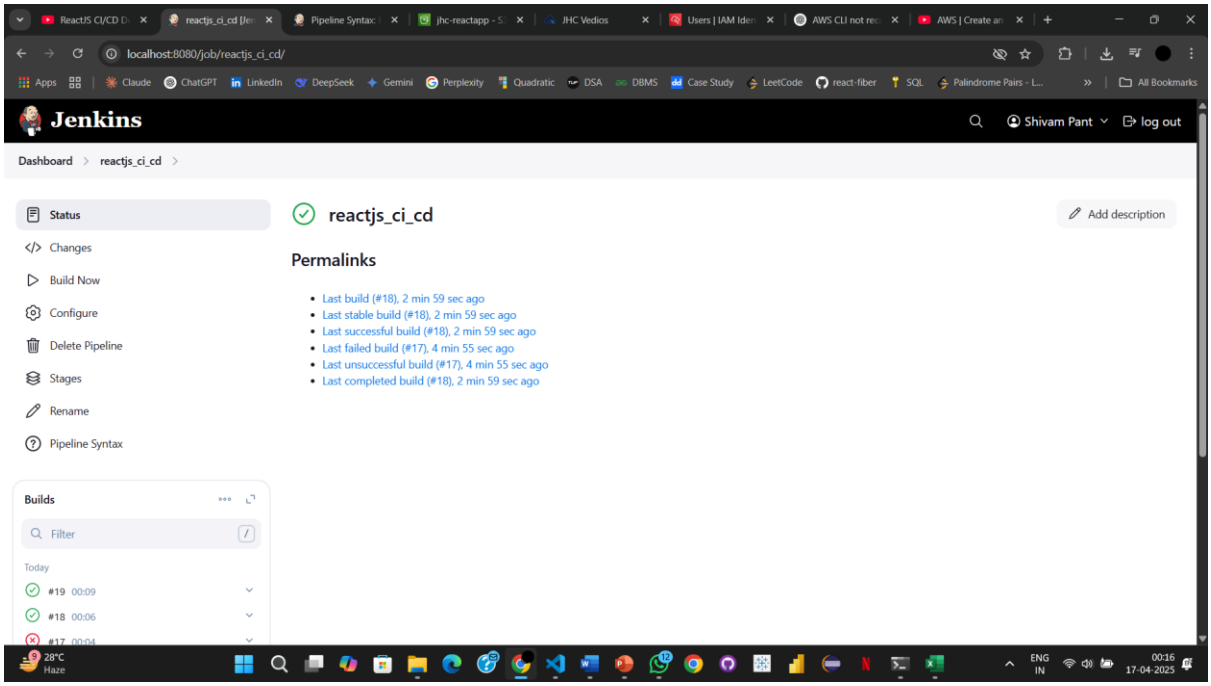
      GitHub → Jenkins (Windows) → AWS S3 & EC2



2. **Figure 2: Jenkins Dashboard**

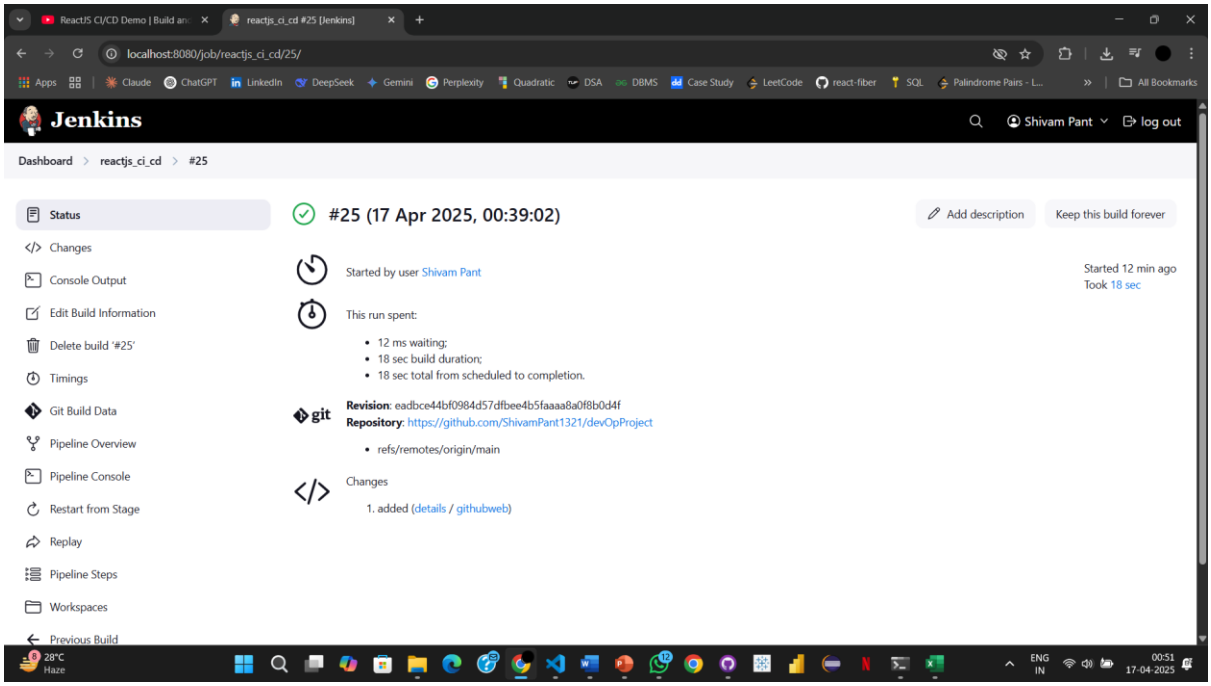
   - A screenshot of the **Jenkins dashboard**, showing the stages of the pipeline and historical build data.

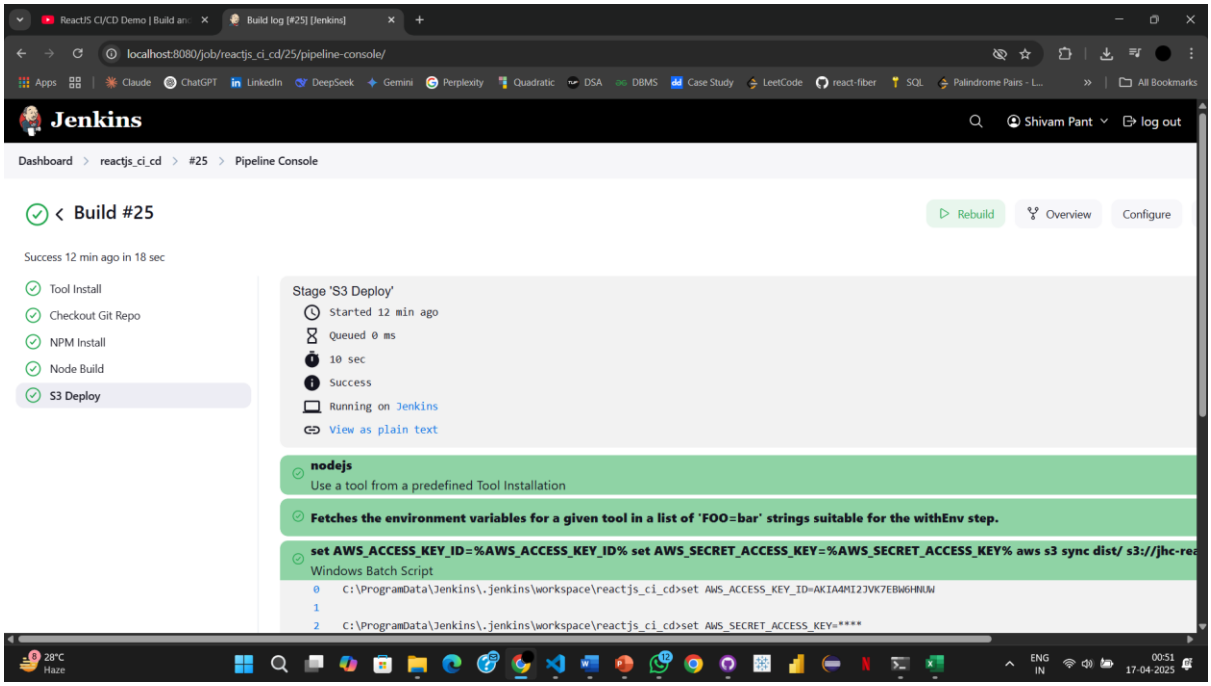


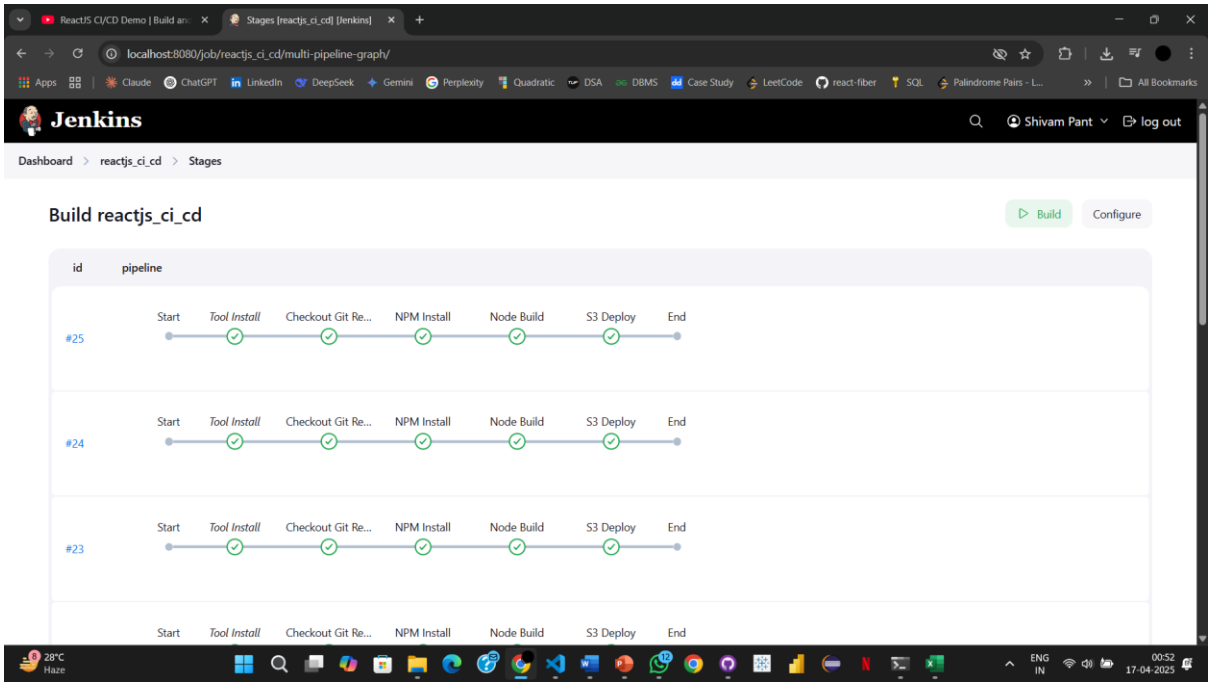


3. **Figure 3: Successful Jenkins Build Log**

   - Output from a successful automated build, including test results and deployment logs.

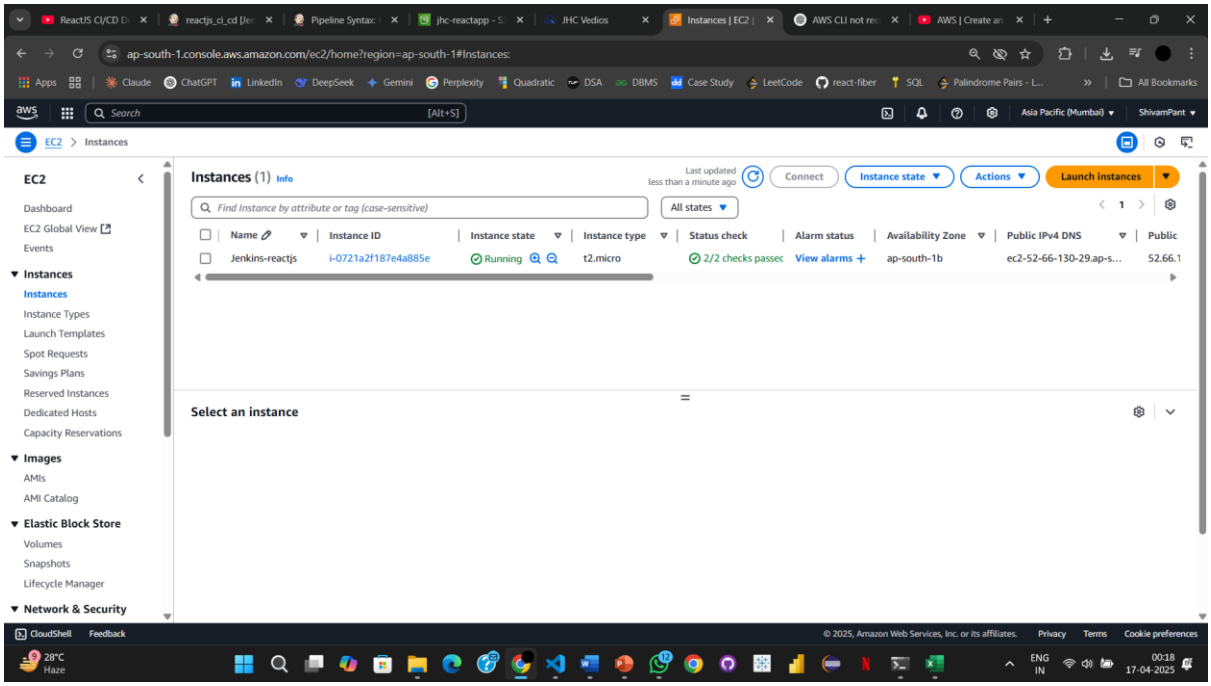






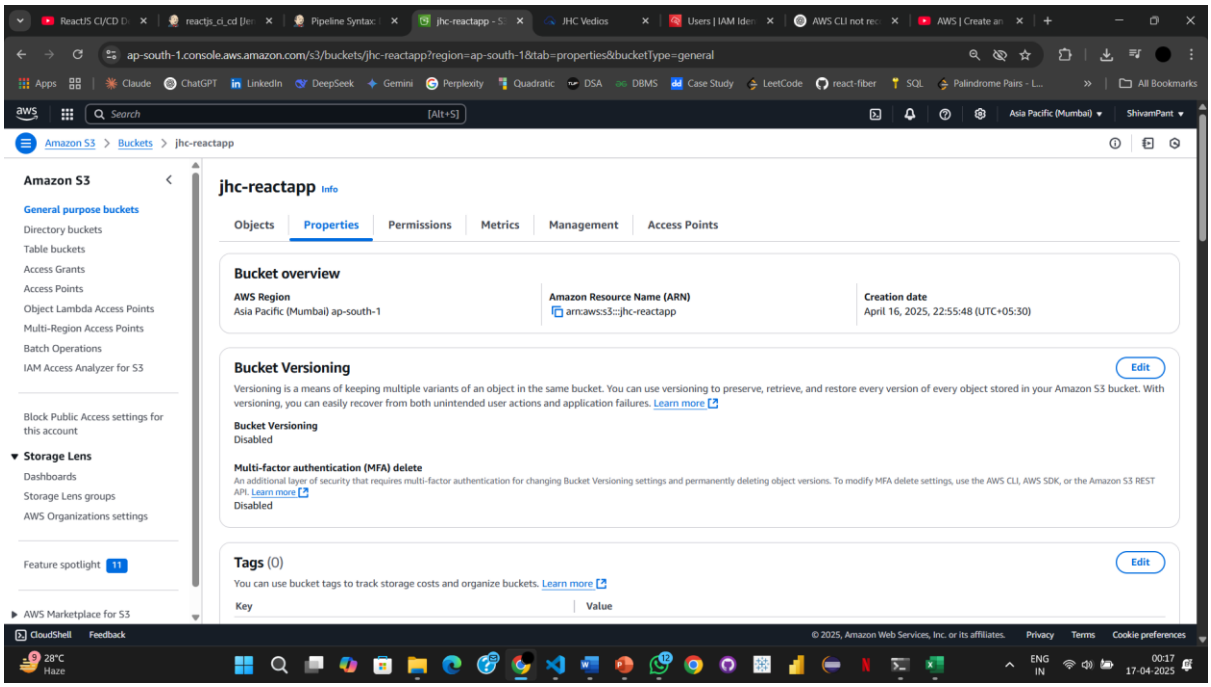
4. **Figure 4: EC2 Instance Web App Deployment**

   - A screenshot of the **e-commerce web app running** on the EC2 instance, accessible via its public IP.



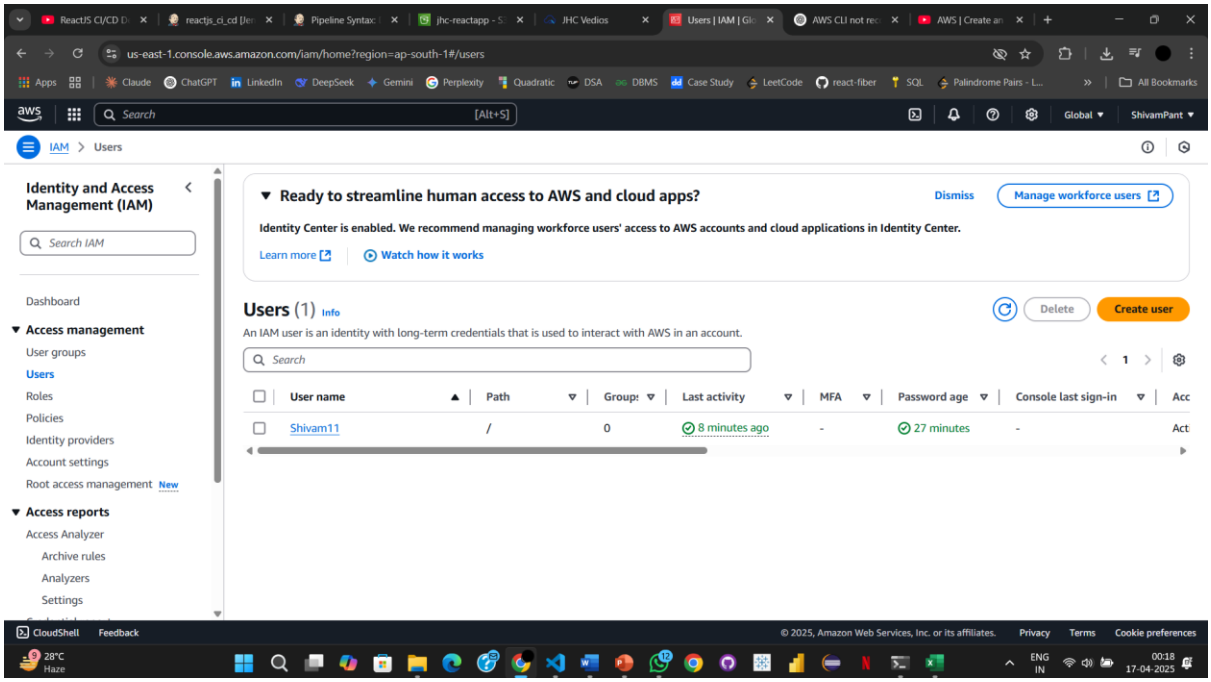
5. **Figure 5: S3 Bucket View**

   - A screenshot from the **AWS S3 console**, showing the stored static files and build artifacts.

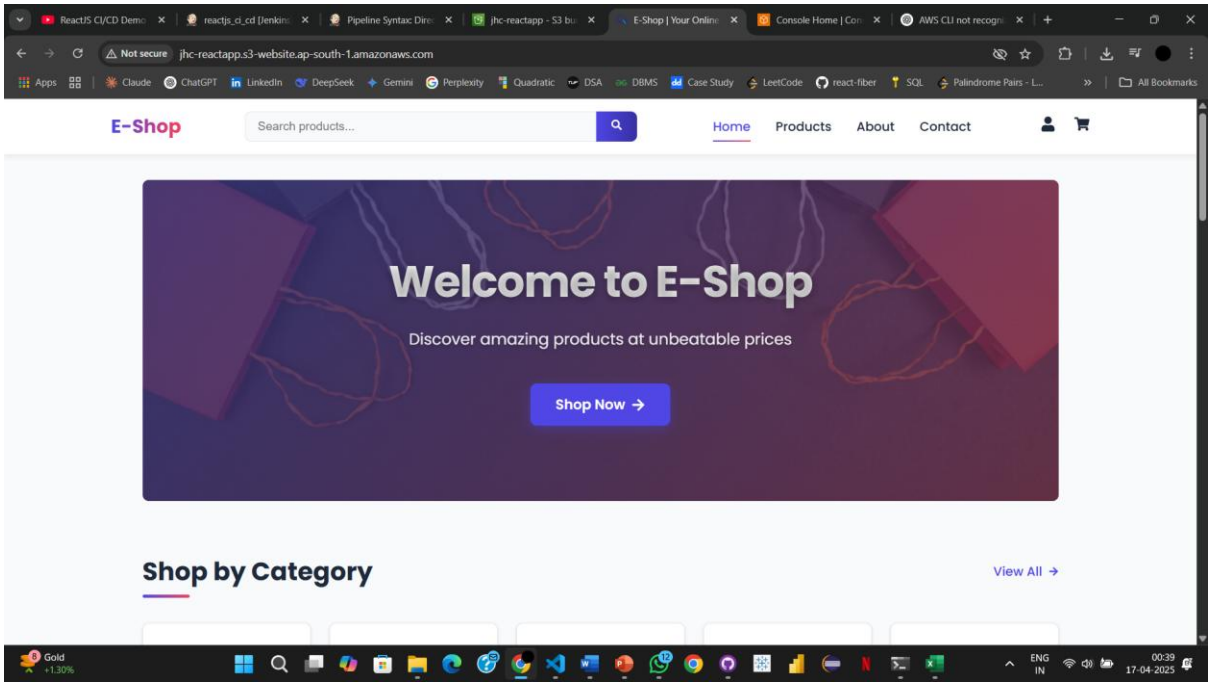


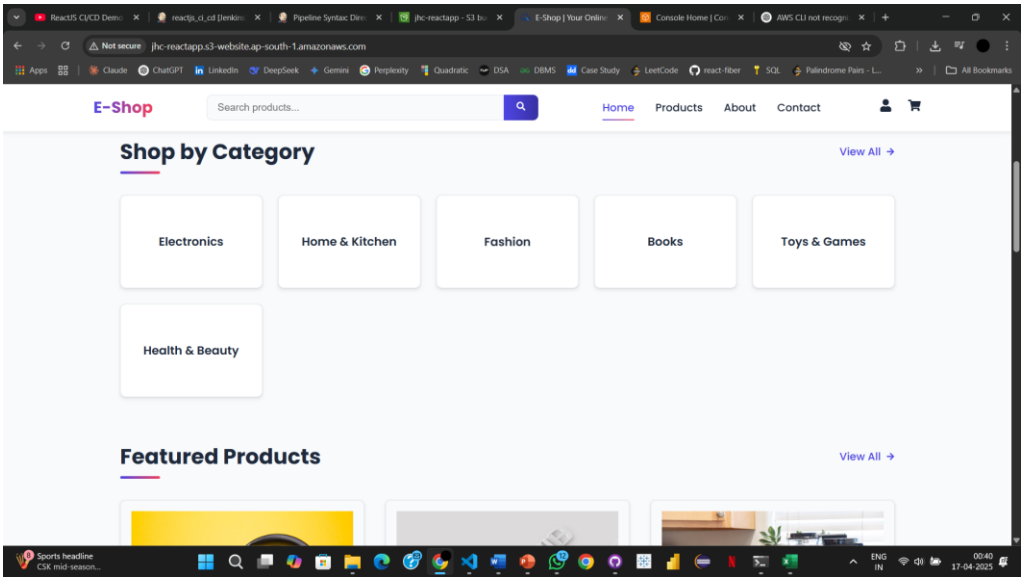
6. **Figure 6: IAM Policy Configuration**

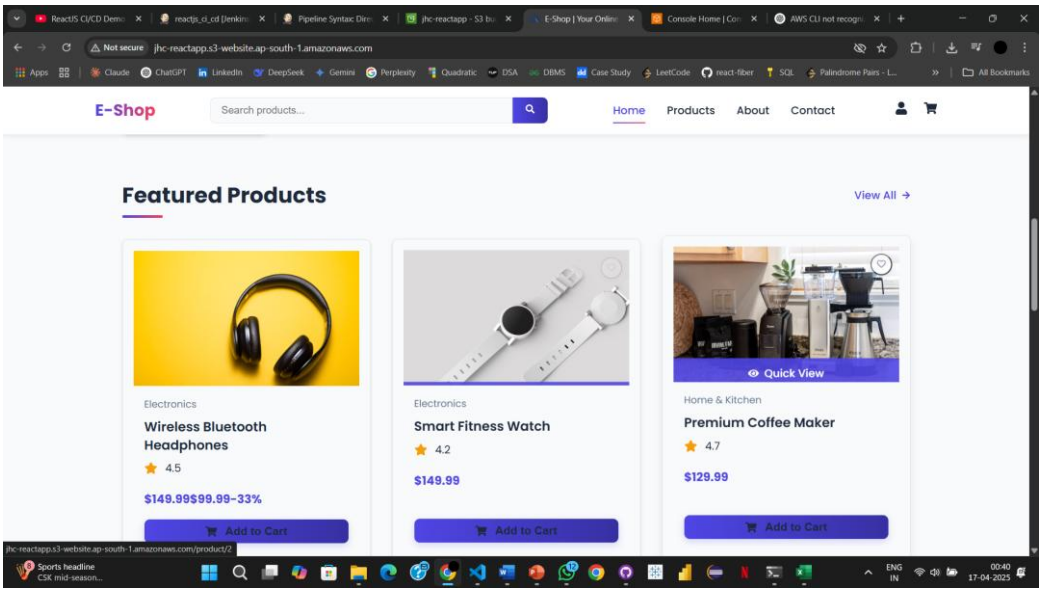
   - A screenshot of the **IAM policy configuration**, demonstrating the permissions set for Jenkins to access S3 and EC2 securely.

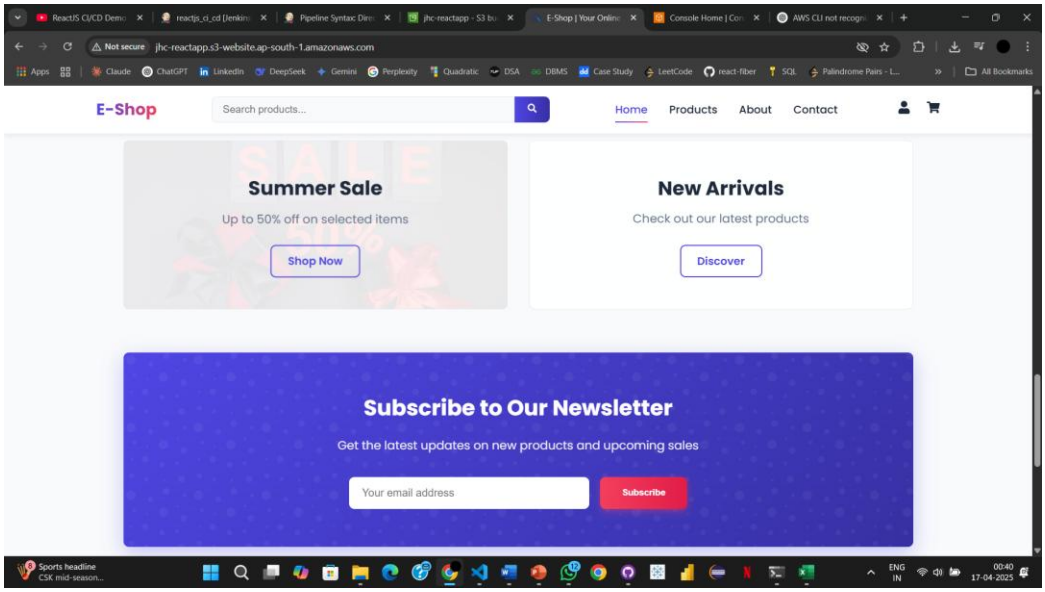


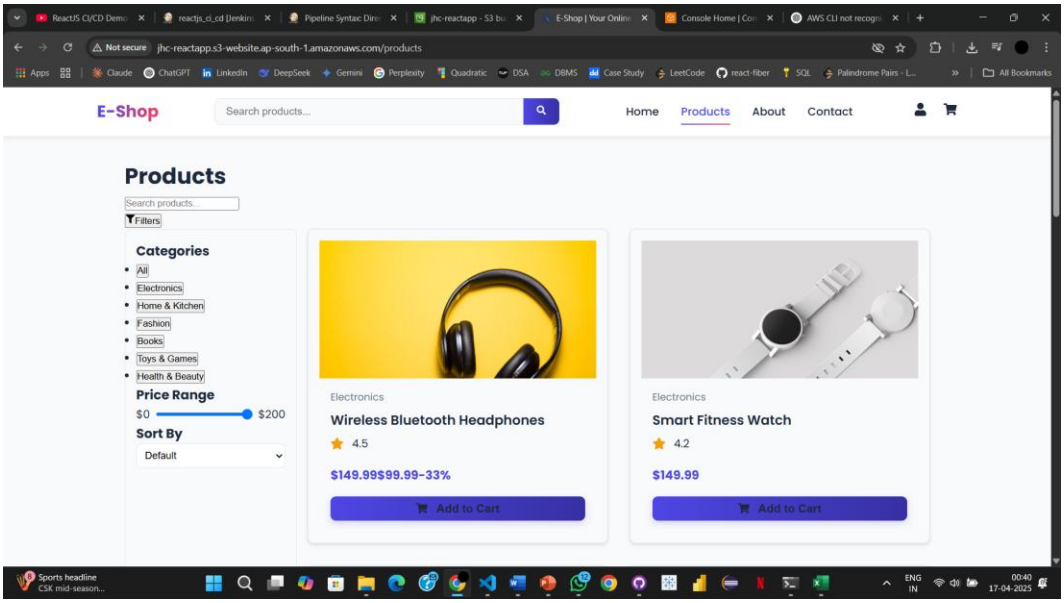
7. **Figure 7: My E-Commerce Website**

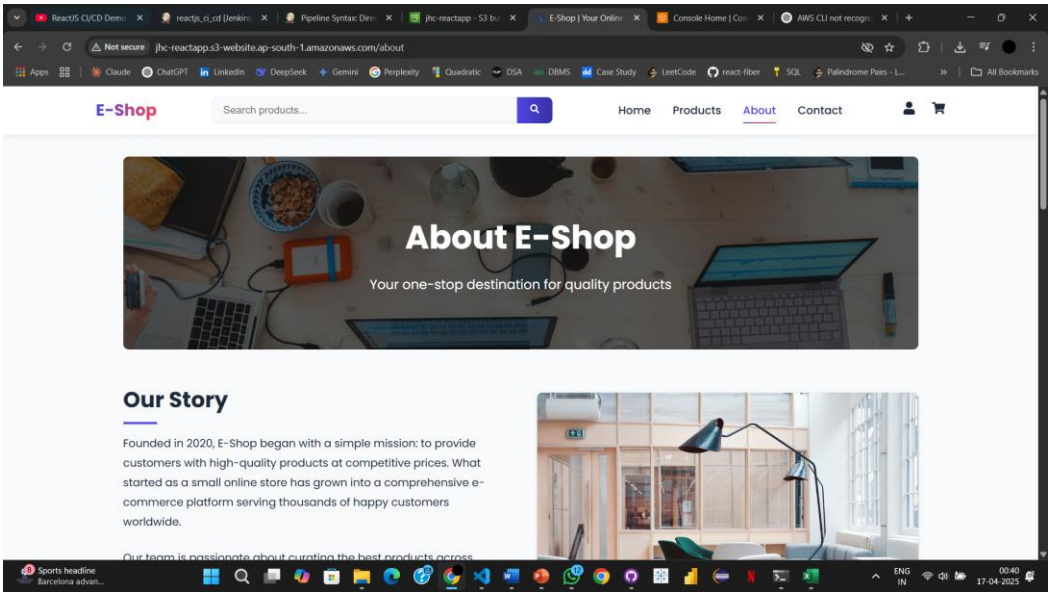


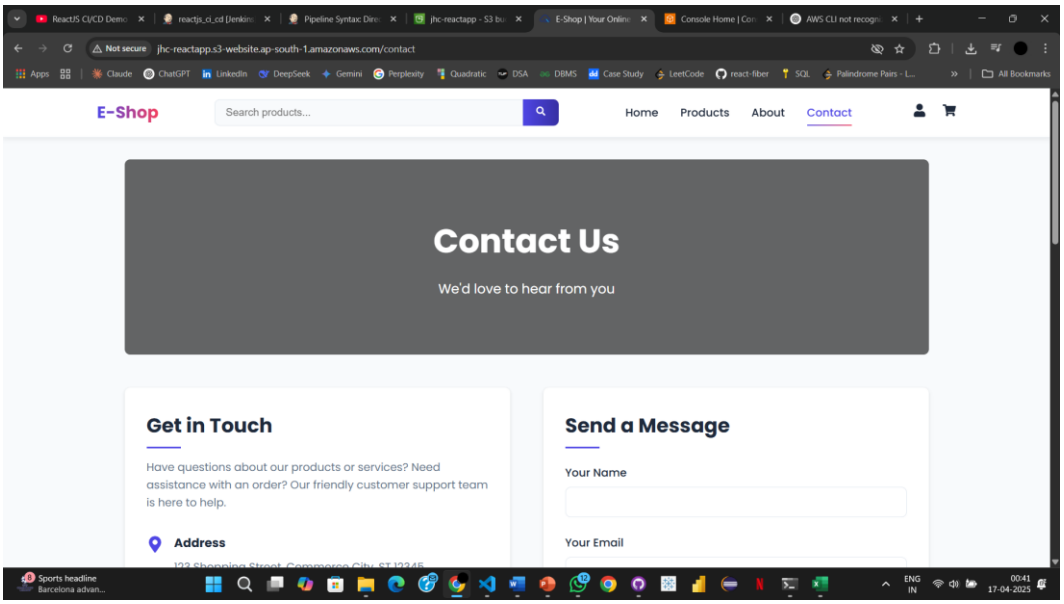

E-Shop
Search products...
Home
Products
About
Contact
Shop by Category
View All →
Electronics
Home & Kitchen
Fashion
Books
Toys & Games
Health & Beauty
Featured Products
View All →

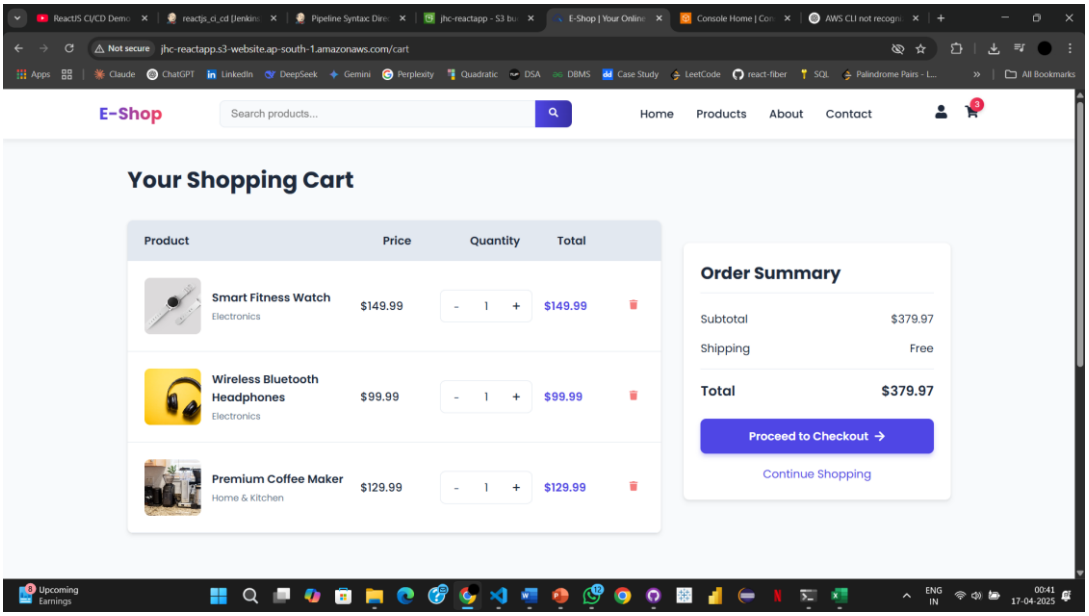

E-Shop
Search products...
Home
Products
About
Contact
Featured Products
View All →
Electronics
Wireless Bluetooth Headphones
4.5
$149.99$99.99-33%
Add to Cart
Electronics
Smart Fitness Watch
4.2
$149.99
Add to Cart
Quick View
Home & Kitchen
Premium Coffee Maker
4.7
$129.99
Add to Cart


E-Shop
Search products...
Home
Products
About
Contact
Summer Sale
Up to 50% off on selected items
Shop Now
New Arrivals
Check out our latest products
Discover
Subscribe to Our Newsletter
Get the latest updates on new products and upcoming sales
Your email address
Subscribe


jhc-reactapp.s3-website.ap-south-1.amazonaws.com/products
E-Shop
Search products...
Home
Products
About
Contact
Products
Search products...
Filters
Categories
All
Electronics
Home & Kitchen
Fashion
Books
Toys & Games
Health & Beauty
Price Range
$0
$200
Sort By
Default
Electronics
Wireless Bluetooth Headphones
4.5
$149.99$99.99-33%
Add to Cart
Electronics
Smart Fitness Watch
4.2
$149.99
Add to Cart


jhc-reactapp.s3-website.ap-south-1.amazonaws.com/about
E-Shop
Search products...
Home
Products
About
Contact
About E-Shop
Your one-stop destination for quality products
Our Story
Founded in 2020, E-Shop began with a simple mission: to provide customers with high-quality products at competitive prices. What started as a small online store has grown into a comprehensive e-commerce platform serving thousands of happy customers worldwide.


jhc-reactapp.s3-website.ap-south-1.amazonaws.com/contact
E-Shop
Search products...
Home
Products
About
Contact
Contact Us
We'd love to hear from you
Get in Touch
Have questions about our products or services? Need assistance with an order? Our friendly customer support team is here to help.
Address
Send a Message
Your Name
Your Email


Not secure
jhc-reactapp.s3-website.ap-south-1.amazonaws.com/cart
E-Shop
Search products...
Home
Products
About
Contact
Your Shopping Cart
Product
Price
Quantity
Total
Smart Fitness Watch
Electronics
$149.99
1
$149.99
Wireless Bluetooth Headphones
Electronics
$99.99
1
$99.99
Premium Coffee Maker
Home & Kitchen
$129.99
1
$129.99
Order Summary
Subtotal
$379.97
Shipping
Free
Total
$379.97
Proceed to Checkout →
Continue Shopping

10. Conclusion

This project successfully implemented a comprehensive **DevOps pipeline** leveraging **Jenkins on Windows**, integrated with essential **AWS services** (EC2, S3, and IAM). The automation of the build, test, and deployment processes significantly improved the **efficiency**, **reliability**, and **security** of the e-commerce website's development lifecycle.

Key outcomes include:

- **Streamlined deployments** with reduced downtime and faster delivery cycles.
- Enhanced **scalability** and **high availability** of the application, thanks to the power of cloud infrastructure provided by **AWS**.
- Improved **security and access management** through the use of **IAM roles and policies**, ensuring safe and controlled access to resources.
- A **modular CI/CD pipeline** that can be easily adapted for future projects, making it a reusable asset for the development team.

In conclusion, the project has not only met its goals of automating and optimizing the deployment process but also laid the groundwork for future enhancements, including better monitoring, advanced security measures, and the ability to scale efficiently.

11. Future Improvements

While the current implementation of the CI/CD pipeline has successfully automated and optimized the deployment process, there are several areas for future improvement to further enhance performance, scalability, and reliability.

1. Add Docker Support for Containerized Deployments

- **Improvement:** Moving towards **containerization** by integrating **Docker** into the pipeline can further improve deployment consistency and scalability.
- **Benefit:** Dockerized applications can be deployed on any environment without worrying about platform dependencies. This also makes it easier to replicate production environments, enhancing portability and simplifying deployments across different stages (development, testing, production).

2. Migrate Jenkins to an EC2 Linux Instance or Jenkins-as-a-Service

- **Improvement:** Migrate Jenkins from **Windows** to a **Linux-based EC2 instance** or switch to **Jenkins-as-a-Service** solutions like **CloudBees Jenkins Platform** for better performance and easier scalability.
- **Benefit:** Running Jenkins on Linux may offer better performance, easier management, and scalability. Jenkins-as-a-Service platforms can provide managed services, reducing the operational overhead of maintaining the Jenkins server.

3. Introduce Monitoring Tools like CloudWatch or Grafana

- **Improvement:** Incorporate monitoring tools such as **Amazon CloudWatch** or **Grafana** for real-time insights into application performance, infrastructure health, and pipeline status.
- **Benefit:** Monitoring tools allow proactive detection of issues, enabling faster response times to potential failures or performance degradation. CloudWatch provides detailed logs and metrics, while Grafana offers advanced visualizations for monitoring complex systems.

4. Implement Blue/Green or Canary Deployments for Zero-Downtime Rollouts

- **Improvement:** Introduce **Blue/Green** or **Canary** deployment strategies to ensure **zero-downtime rollouts** for new application versions.

- **Benefit:** These deployment strategies enable smooth transitions between old and new versions, minimizing risk and ensuring users are not affected by any potential issues in the new version. Blue/Green allows for a seamless switch between two environments, while Canary deployments let new versions roll out incrementally, reducing the impact of any failures.