**CS691 – Jenkins Progress Report**

**Team: 1**

**Jenkins Admin: Abhishek Tamboli.**

**Sprint 1 - Planned and Completed Tasks**

|  |  |
| --- | --- |
| **Planned Tasks** | **Completed Tasks** |
| **1. Configuring Required Plugins in Jenkins** – Install essential plugins like AWS Credentials Plugin for securely managing AWS authentication, Slack Notification Plugin for real-time alerts, and GitHub Integration Plugin for source code automation. These plugins are necessary for pipeline automation. | We successfully installed and configured the AWS Credentials Plugin to securely manage our ECR access, GitHub Integration Plugin for SCM connection, and Slack Notification Plugin for real-time pipeline status updates. These plugins were tested and are now integrated into our Jenkins pipeline for automated builds and deployments. |
| **2. Parameterizing the Jenkins Pipeline for Dev & QA** - Modify the Jenkins pipeline to accept an environment parameter (Dev or QA). Based on the selected environment, Jenkins will update the Ansible inventory and deploy the HouseHunt application to either Dev (Dev-HouseHunt-Team1) or QA (QA-HouseHunt-Team1) dynamically. This step ensures that all further tasks execute based on the selected environment. | We implemented a parameterized Jenkins pipeline that allows the selection of deployment environments such as Dev or QA during pipeline execution. This provided flexibility for triggering builds and deployments based on the target environment. It also helped in managing environment-specific configurations and workflows through conditional stages in the pipeline. |
| **3. Automating Code Checkout & Version Control -** Set up GitHub Webhooks to trigger Jenkins whenever a new commit is pushed, ensuring automatic pipeline execution and version tracking. Integration Plugin for source code automation. These plugins are necessary for pipeline automation. | Jenkins was integrated with GitHub using Webhooks, ensuring automatic pipeline execution on every push. The Jenkinsfile was set up to pull the latest code changes from the correct branch and trigger a build automatically, streamlining version control and reducing manual effort. |
| **4. Building a Dockerized React Application -** Create a multi-stage Dockerfile where the first stage compiles the React app, and the second stage serves it using Nginx for efficient deployment. | We created a multi-stage Dockerfile for the React frontend. The first stage builds the app using Vite, and the second stage serves it using Nginx. This approach minimized image size and improved build performance. The Docker image was tested locally before being pushed to AWS ECR. |
| **5. Managing Docker Images in AWS ECR** - Configure Jenkins to authenticate with AWS ECR, tag Docker images properly, and push the latest builds to the ECR repository for versioned storage. | Jenkins authenticates with AWS ECR using IAM credentials, pulls repository names dynamically, and tags the images before pushing them to ECR. We created three repositories for client, api, and socket services, ensuring modular image storage and deployment flexibility. |
| **6. Automating Deployment with Ansible -** Develop an Ansible playbook that connects to the EC2 instance, pulls the latest Docker image from ECR, and deploys the application automatically. | Ansible playbooks were created and integrated into Jenkins. Upon build completion, Jenkins triggers these playbooks, which SSH into the EC2 instance, pull the latest Docker images from ECR, and deploy each container (client, api, socket) within a shared Docker network. |
| **7. Deploying the HouseHunt Application on EC2 -** Ensure the EC2 instance is properly configured, with the necessary security settings, ports open, and user access permissions set for deployment. | The EC2 server was configured with Docker, required system dependencies, and open ports (80, 8800, 4000). Nginx was used as a reverse proxy for frontend, API, and socket traffic. Containers were verified post-deployment using curl and browser tests. |
| **8. Updating the Ansible Inventory File -**  Automate the process of updating the inventory file in Jenkins to dynamically fetch and replace the EC2 server’s IP for Ansible deployment. | We dynamically update the Ansible inventory within Jenkins using shell scripts to fetch the public IP of the EC2 instance before deployment. This ensures the deployment always targets the correct host, even if the IP changes after instance recreation. |
| **9. Implementing Slack Notifications -**  Integrate Jenkins with Slack to send automatic notifications for build start, success, or failure, ensuring visibility in the Team1 Slack channel. | Jenkins was connected with our team’s Slack workspace using the Slack plugin. Build start, success, and failure notifications are automatically sent to our #team1channel, helping the team stay informed in real-time about pipeline activities. |

**Sprint 2 - Planned and Completed Tasks**

|  |  |
| --- | --- |
| **Planned Tasks** | **Completed Tasks** |
| **1. Create Separate Jenkins Pipelines for Dev and QA -** We plan to create independent Jenkins pipelines for Dev and QA environments instead of using a single parameterized job. This will simplify management and allow isolated testing and deployments. | We successfully separated the Jenkins pipelines for Dev and QA by creating individual Jenkinsfiles. This allowed each environment to operate independently, using distinct EC2 IPs and AWS ECR repositories. This improved deployment management and made environment-specific debugging easier. |
| **2. Plan to Integrate Jenkins Blue Ocean Plugin -** We intend to integrate the Blue Ocean plugin to improve the user experience with a visual pipeline interface. This will make it easier to track and debug pipeline stages. | The Blue Ocean plugin was installed and configured in Jenkins. It provided a modern UI with visual pipeline views, helping the team trace job execution stages, parallel steps, and failure points clearly and interactively. |
| **3. Plan to Add Conditional Stages in Jenkinsfile -** We are planning to include conditional execution in the Jenkinsfile to skip stages like Ansible deployment if the build fails. This ensures a cleaner and more efficient CI/CD process. | We integrated conditional when blocks in the Jenkinsfiles, allowing deployment stages to run only after a successful build. This improved pipeline reliability and prevented unnecessary executions in case of build failures. |
| **4. Plan to Configure Jenkins Backup Using ThinBackup Plugin -** To ensure job safety, we will configure the ThinBackup plugin to take periodic backups of Jenkins settings and configurations. This helps prevent data loss and misconfiguration issues. | The ThinBackup plugin was installed, and backup directories were configured. Scheduled backups now archive Jenkins job configurations, credentials, and system settings. This ensures disaster recovery capability and smooth restoration in case of corruption. |
| **5. Plan to Set Up Branch-Based Build Control -** We will configure Jenkins jobs to trigger only on specific branches like main and dev. This approach enforces controlled deployment and maintains environment stability. | GitHub branch protection rules were linked with Jenkins. Webhooks were configured so that only successful builds are allowed to be merged into protected branches. This step maintains repository integrity and enforces code quality. |
| **6. Plan to Add Slack Notifications with Commit Messages -** We aim to enhance Slack notifications by including the last Git commit message. This keeps the team informed about what code changes triggered the Jenkins job. | Slack messages now include the Jenkins job URL, Git branch, build number, and the latest commit message. This enhancement gives clear insight into what was deployed and helps the team stay updated without checking Jenkins manually. |
| **7. Plan to Monitor Jenkins Jobs with Build Metrics Plugin -** We plan to install and configure the Build Metrics plugin to track build performance and success rates. These insights will help optimize our Jenkins usage over time. | Although we initially planned to use the Build Monitor plugin, we switched to the **Dashboard View** plugin as it provided better visualization and compatibility. The plugin was configured to show build status, trends, and last success/failure for each job in a live dashboard. |
| **8. Plan to Practice Rollbacks Using Tagged Docker Images -** We will explore image rollback using Git tags and AWS ECR to recover from deployment issues. Jenkins will be configured to pull specific image tags for restoration. | This task was deferred for a future sprint. While it is technically possible to configure image tagging and rollback, implementing dynamic tag management in Ansible alongside CI/CD requires additional effort and planning. We’ll revisit this once the deployment flow is fully matured. |

**Sprint 3 - Planned and Completed Tasks**

|  |  |
| --- | --- |
| **1. Implement Role-Based Access Control (RBAC) -** We plan to configure the Role-Based Authorization Strategy plugin to define permission levels for admin, developer, and viewer roles, improving access control and security within Jenkins. | We successfully implemented RBAC using the Role-Based Authorization Strategy plugin. Custom roles were created for Admin, Developer, and Viewer, each with appropriate access permissions. This improved security by limiting job execution, credential access, and system configuration visibility. |
| **2. Integrate ESLint with Warnings Next Generation Plugin (updated from SonarQube) -** We will integrate ESLint for static code analysis in place of SonarQube. The Warnings Next Generation plugin will be used to display ESLint issues directly in Jenkins UI to ensure front-end code quality and maintainability. | We configured ESLint for static code analysis in the frontend project and integrated it with Jenkins using the Warnings Next Generation plugin. This integration displays ESLint issues directly in Jenkins build results, enabling the team to monitor code quality and maintain consistent frontend standards effectively. |
| **3. Set Up OWASP Dependency-Check -**  As part of securing our CI pipeline, we will configure the OWASP Dependency-Check plugin to scan third-party libraries for known vulnerabilities and generate alerts during build time. | OWASP Dependency-Check was integrated as a Jenkins post-build step for the API service. It scans dependencies for known CVEs and generates vulnerability reports. This ensures early detection of risky libraries and improves project security posture. |
| **4. Add Manual Approval for Production Deployment -** We plan to introduce a manual approval gate using the Promoted Builds plugin or Jenkins input step before deploying to QA. This allows us to verify readiness and prevents accidental releases. | We added a manual input step in the Jenkins pipeline before deploying to QA. This approval gate prevents accidental deployments and provides a checkpoint for team leads to review the build status and readiness. |
| **5. Integrate JMeter for Performance Testing -** JMeter scripts will be added to evaluate API performance. Test results will be visualized in Jenkins post-build to ensure application responsiveness and catch regressions early. | JMeter was integrated to execute performance testing on critical API endpoints. The Jenkins pipeline is configured to trigger these tests after each build and automatically archive the performance reports. This integration ensures continuous tracking of response times and facilitates early identification of performance issues. |
| **6. Retry Failed Stages Automatically -**  To make the pipeline more resilient, we plan to implement retry logic for unstable steps like Docker login or ECR push to reduce build failures caused by temporary issues. | We wrapped critical steps like Docker login and image push in retry blocks using the retry directive in Jenkinsfile. This reduces false build failures due to transient issues such as network glitches or AWS throttling. |
| **7. Publish Test and Coverage Reports -**  We will configure Jenkins to collect and display JUnit test results and code coverage reports, allowing the team to quickly verify the quality and reliability of each build. | JUnit and coverage reports were set up using standard Jenkins plugins. These reports are now available in the Jenkins build results, helping the team stay informed about test status and coverage. |
| **8. Integrate Jenkins with Grafana via Prometheus -** We plan to expose Jenkins job metrics using the Prometheus plugin and visualize real-time build stats, trends, and failure rates using Grafana dashboards. | We exposed Jenkins metrics using the Prometheus plugin and visualized them in Grafana using Dashboard ID 9964. Metrics like job success rate, build duration, and queue length are now visible in real-time, aiding continuous monitoring and health checks. |