Implementation: Using AWS cloud services to automate application deployment through a CI/CD pipeline.

Objective: To design and implement an automated CI/CD pipeline using Jenkins and AWS cloud services for efficient application deployment.

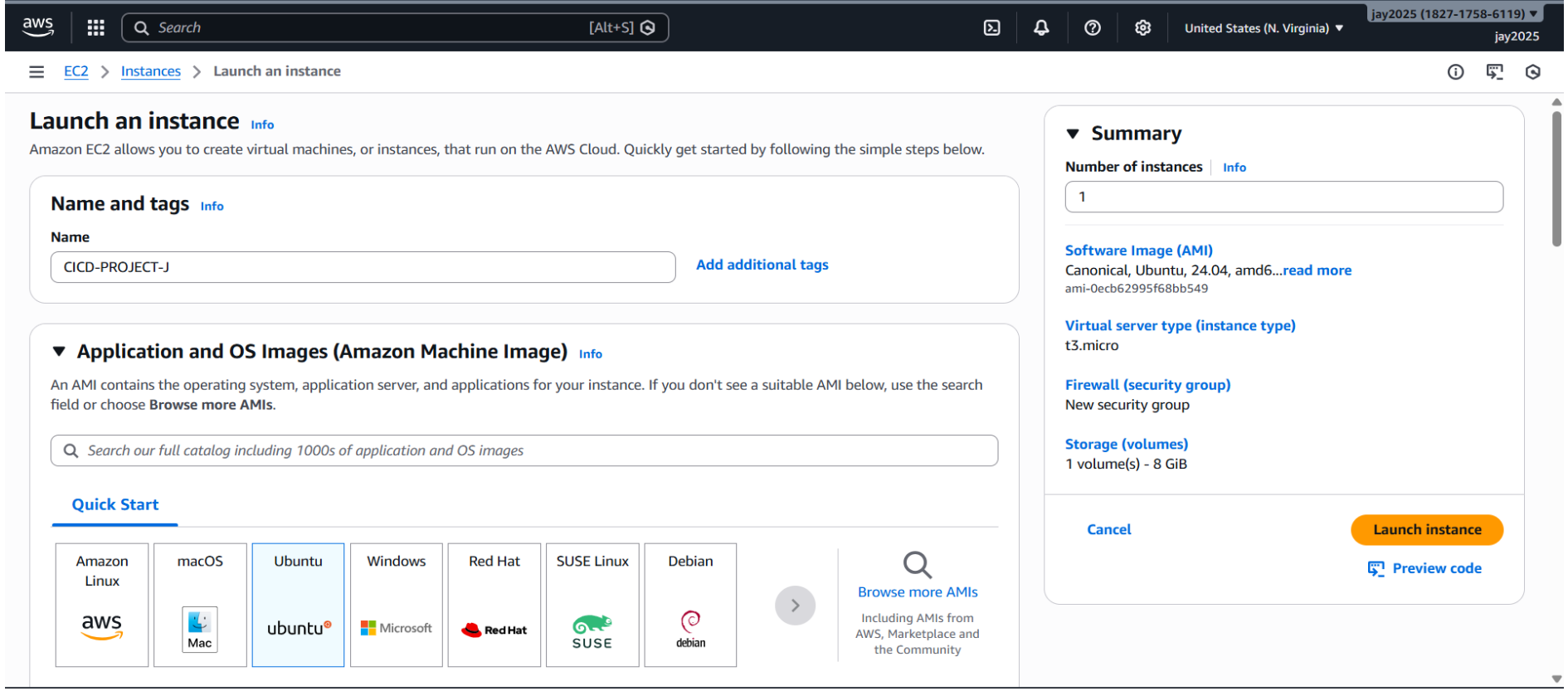
- **Tools & Technologies Used**

AWS CodeBuild
AWS CodeBuild is a fully managed build service that compiles source code, runs tests, and produces build artifacts automatically without managing servers

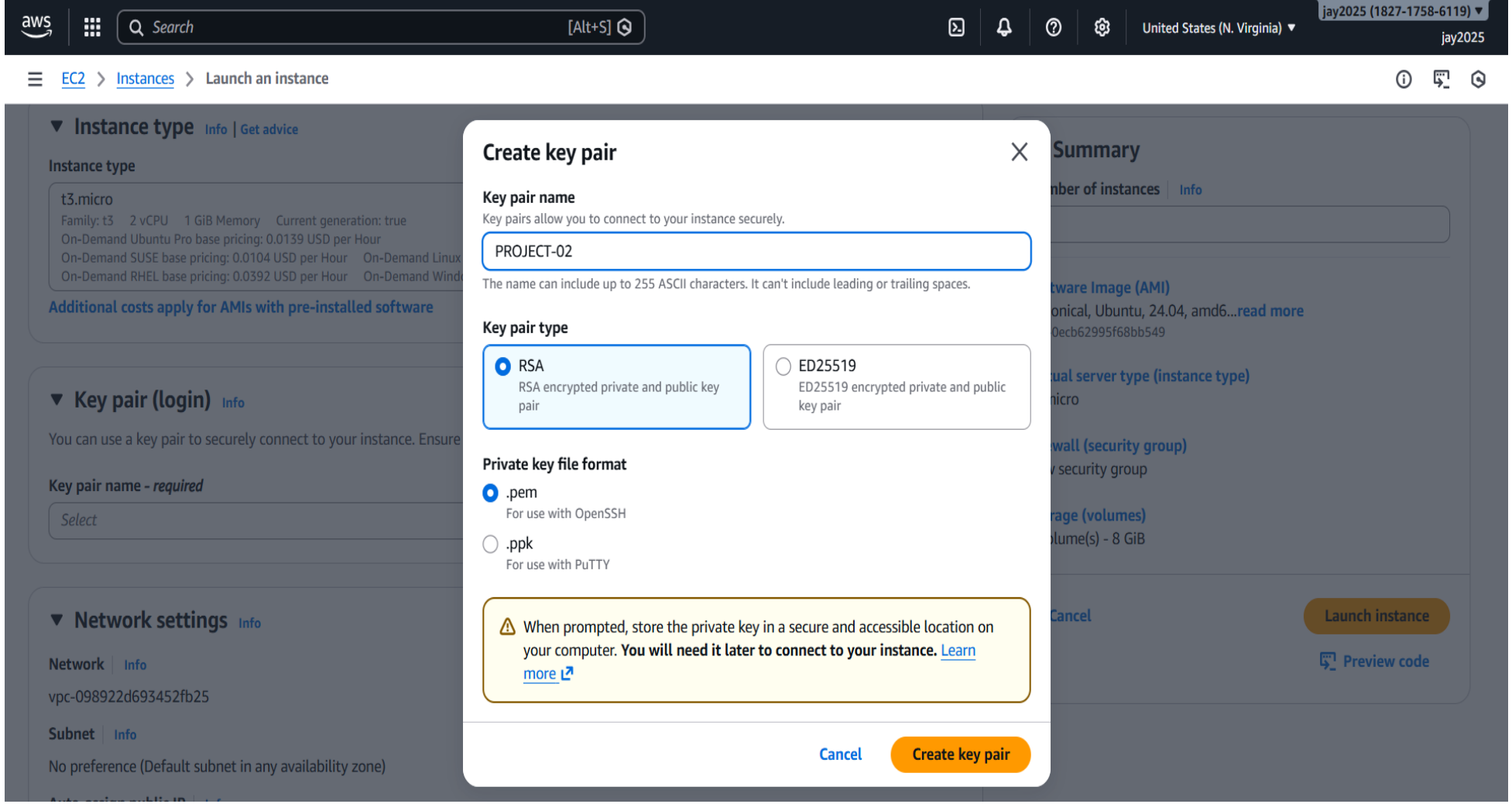
AWS CodeDeploy
AWS CodeDeploy is a deployment service that automates the process of deploying applications to EC2 instances, ensuring reliable and consistent releases.

CI/CD Pipeline
A CI/CD pipeline is an automated workflow that enables continuous integration and continuous deployment by building, testing, and deploying applications whenever code changes are made.

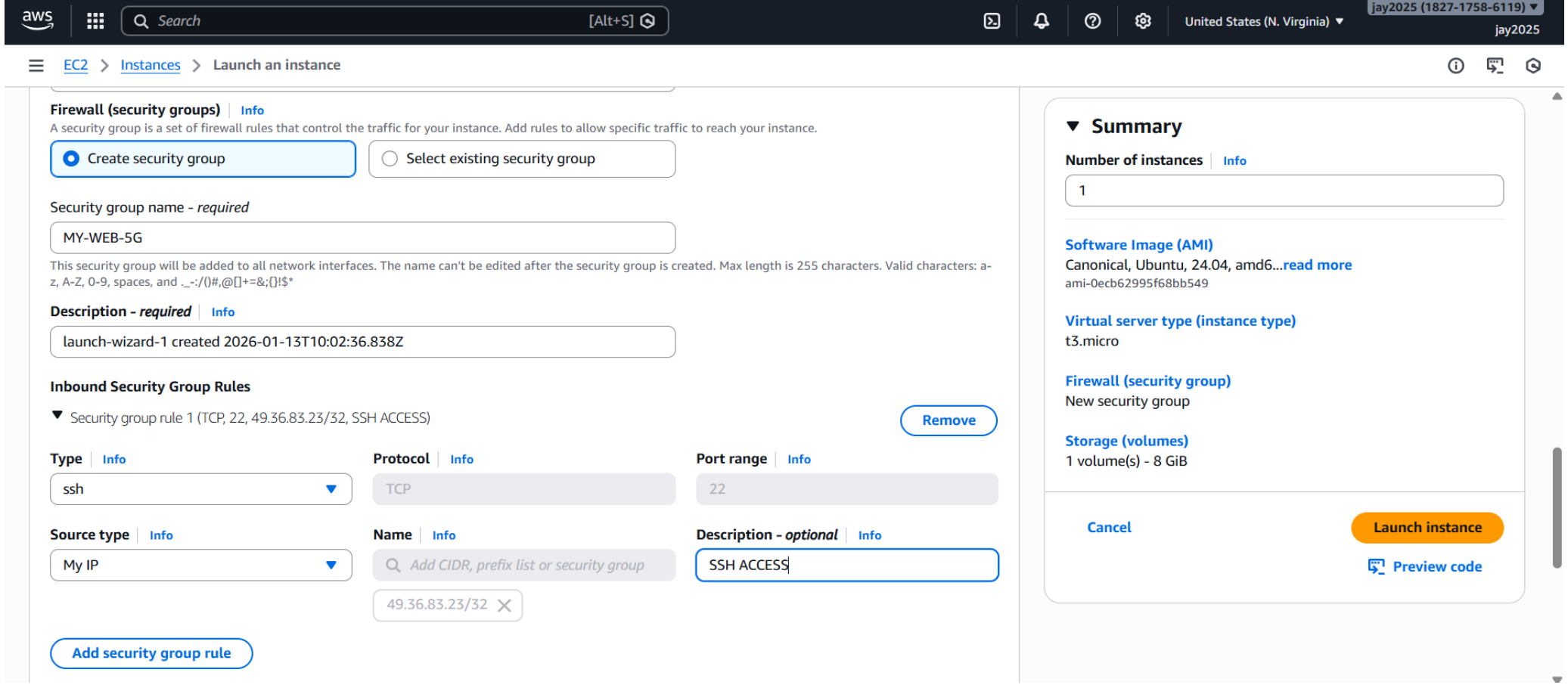
Jenkins
Jenkins is an open-source automation tool used to implement CI/CD pipelines by integrating source control, build, test, and deployment processes through plugins.



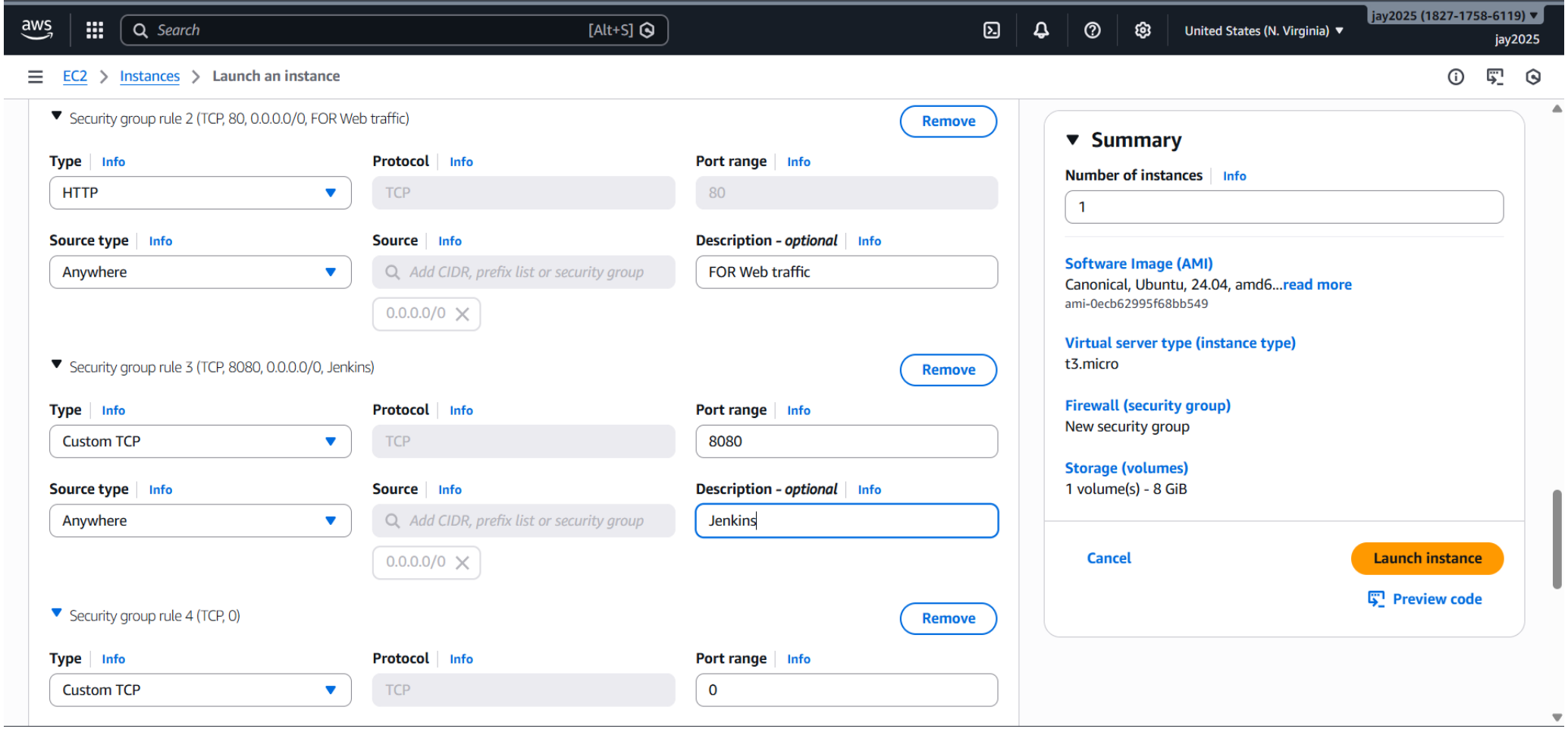
I'm starting the process of creating an AWS EC2 virtual server. I navigate to EC2 > Instances > Launch an instance and name my instance "CICD-PROJECT-J" for easy identification. I select Ubuntu 24.04 from the Quick Start options as my operating system because it's reliable for web development. I choose the t3.



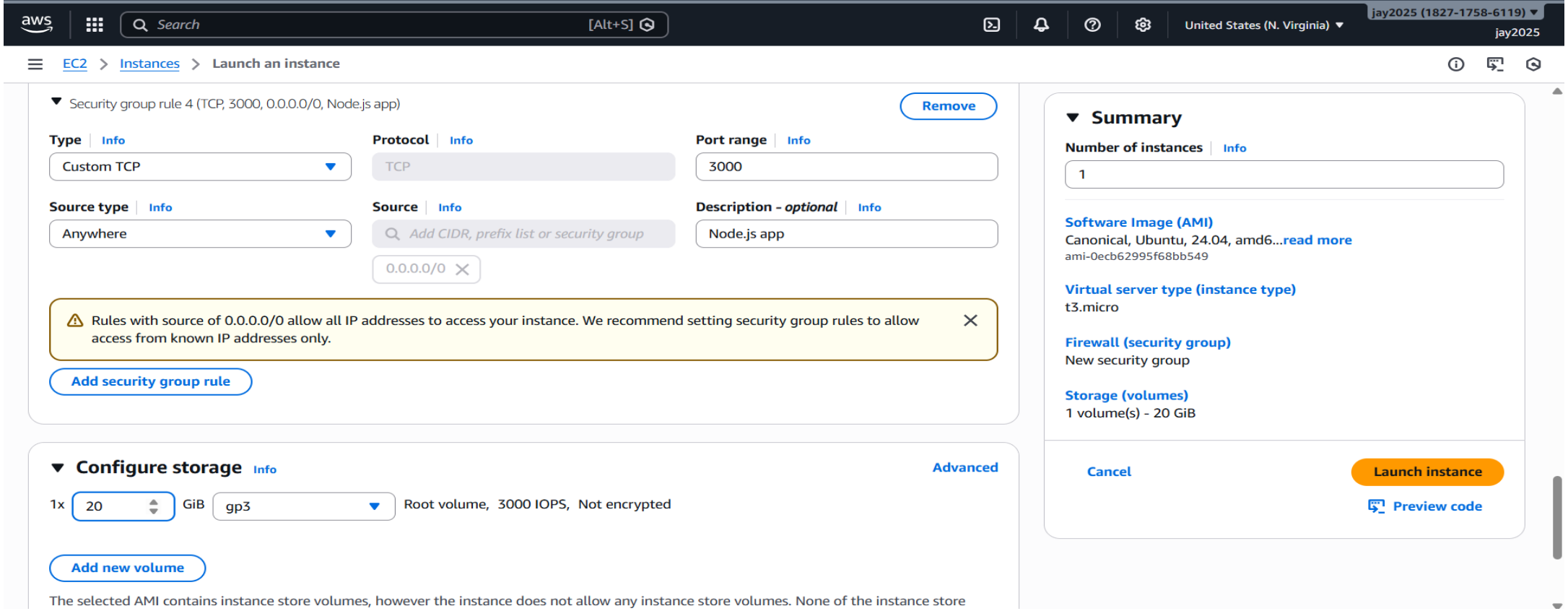
Here I'm creating a security key pair that will allow me to connect to my server remotely. I click "Create key pair" and name it "PROJECT-02" to match my project. I select RSA encryption type and .pem file format because it works with OpenSSH on my Windows/Linux system. Once I click "Create key pair," the private key file downloads to my computer - I need to keep this safe as it's my only way to access the server via SSH.



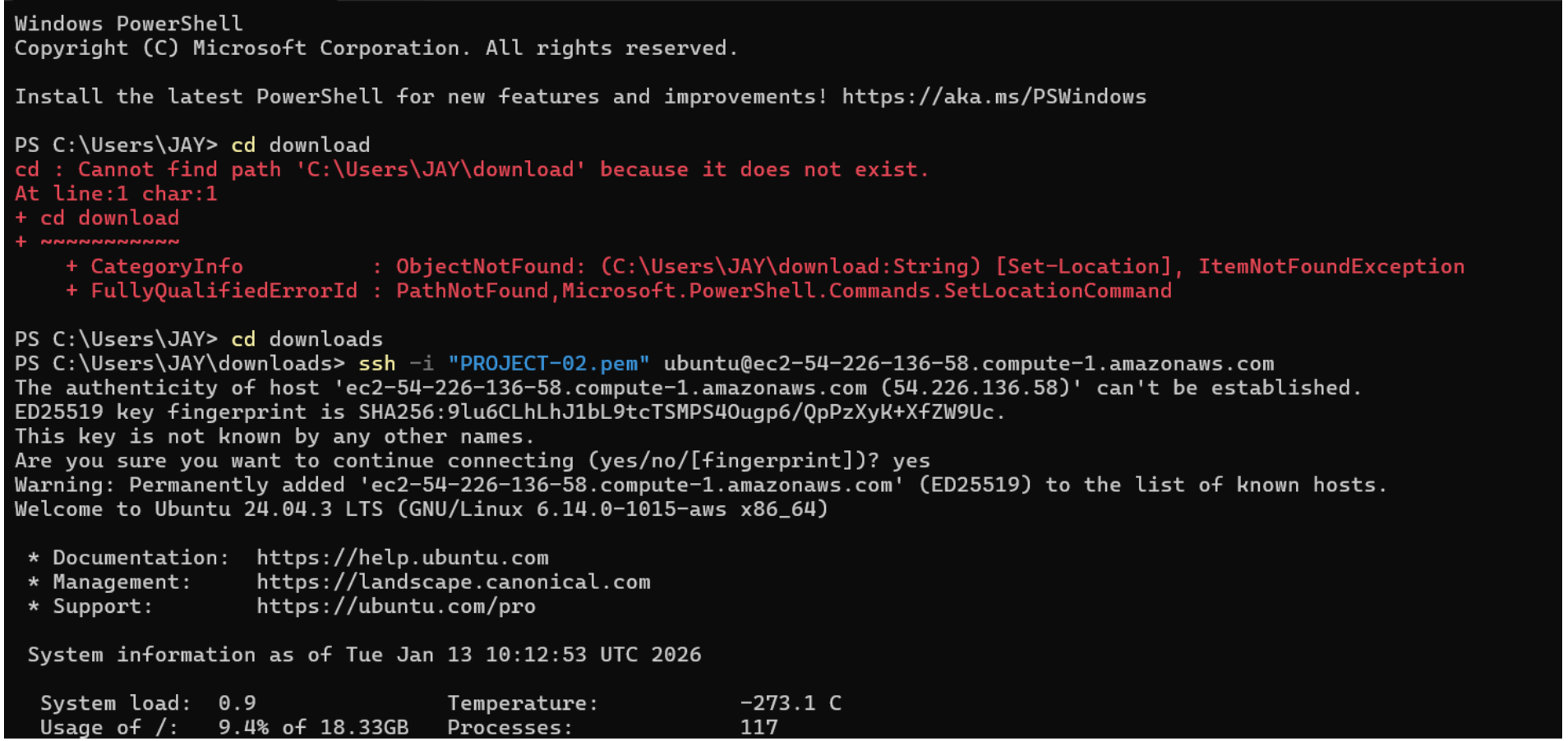
Now I'm setting up firewall rules by creating a new security group named "MY-WEB-SG". The first rule I add is for SSH access on port 22 using TCP protocol, which allows me to remotely manage the server through terminal. I set the source to "My IP" and enter my current IP address (49.36.83.23/32) so only I can SSH into the server, blocking unauthorized access attempts. I label this rule "SSH ACCESS" for documentation purposes.



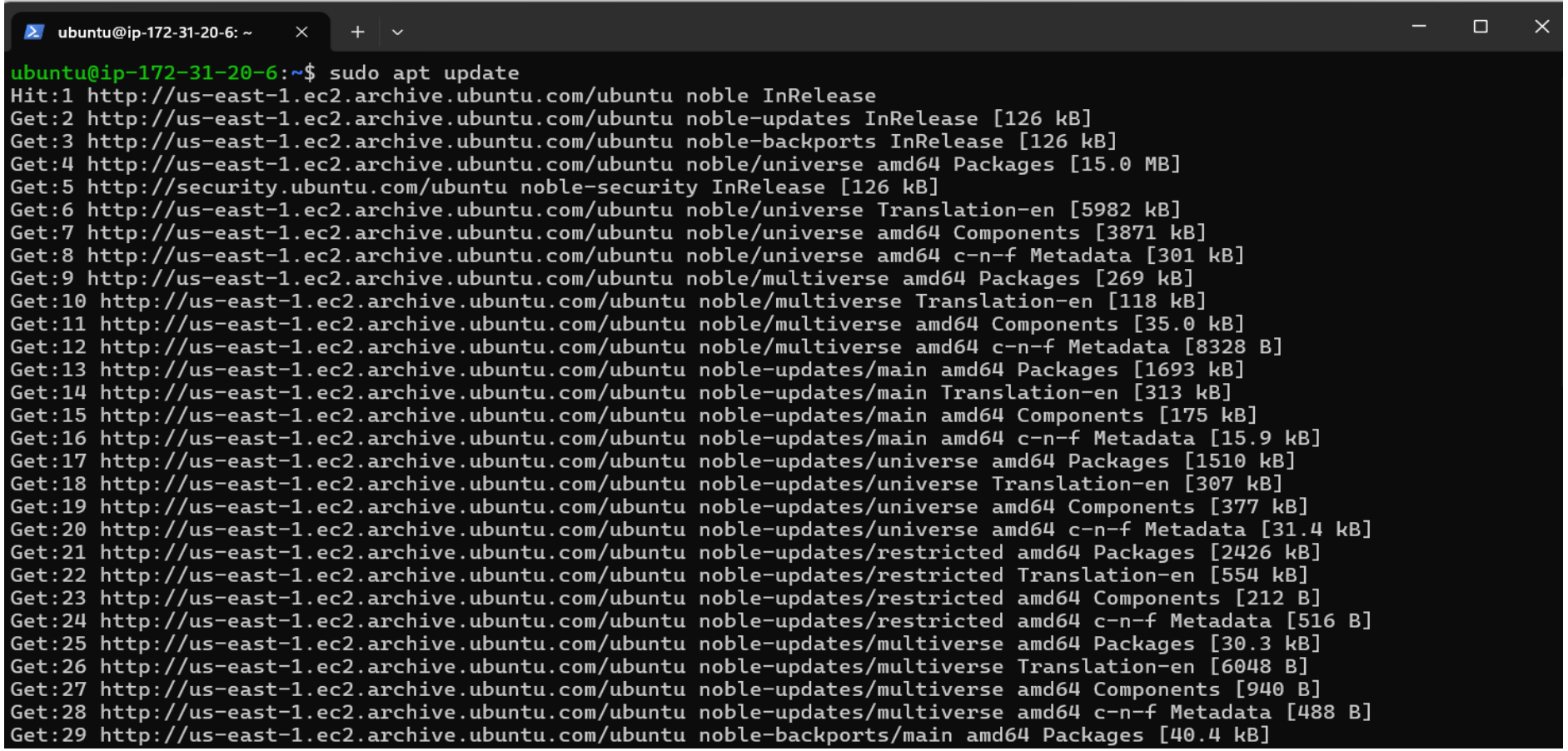
I'm adding more firewall rules to allow different types of traffic to my server. I create a rule for HTTP on port 80 with source "Anywhere" (0.0.0.0/0) and label it "FOR Web traffic" - this lets anyone access my website through a browser. Next, I add port 8080 using Custom TCP for Jenkins, also set to "Anywhere" so the Jenkins CI/CD dashboard can be accessed publicly. These rules enable my web application and automation tools to be reached from the internet.



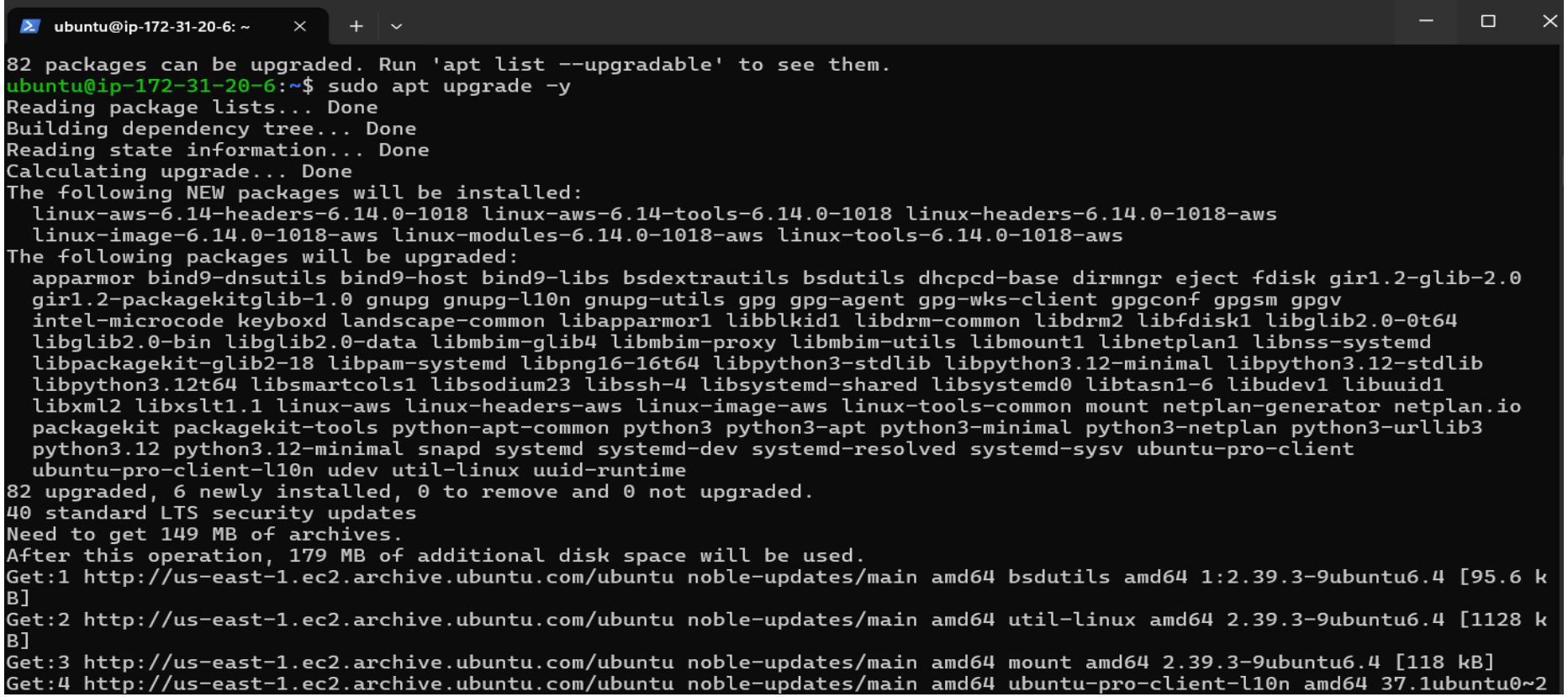
I'm configuring the fourth security rule for port 3000 with Custom TCP protocol and "Anywhere" source, which I describe as "Node.js app". This opens the port commonly used by Node.js applications like Express or React dev servers. AWS shows me a warning that 0.0.0.0/0 allows



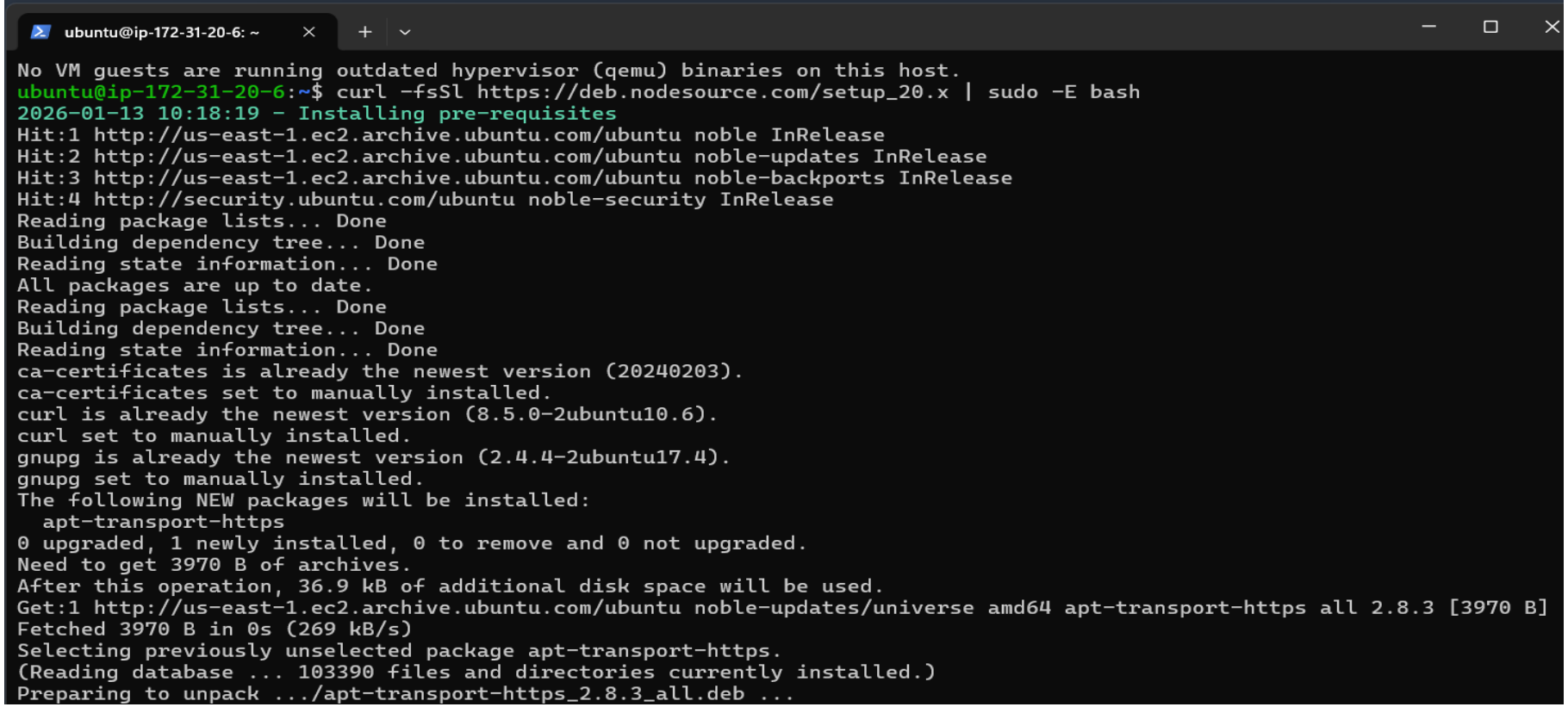
Now connected to powershell



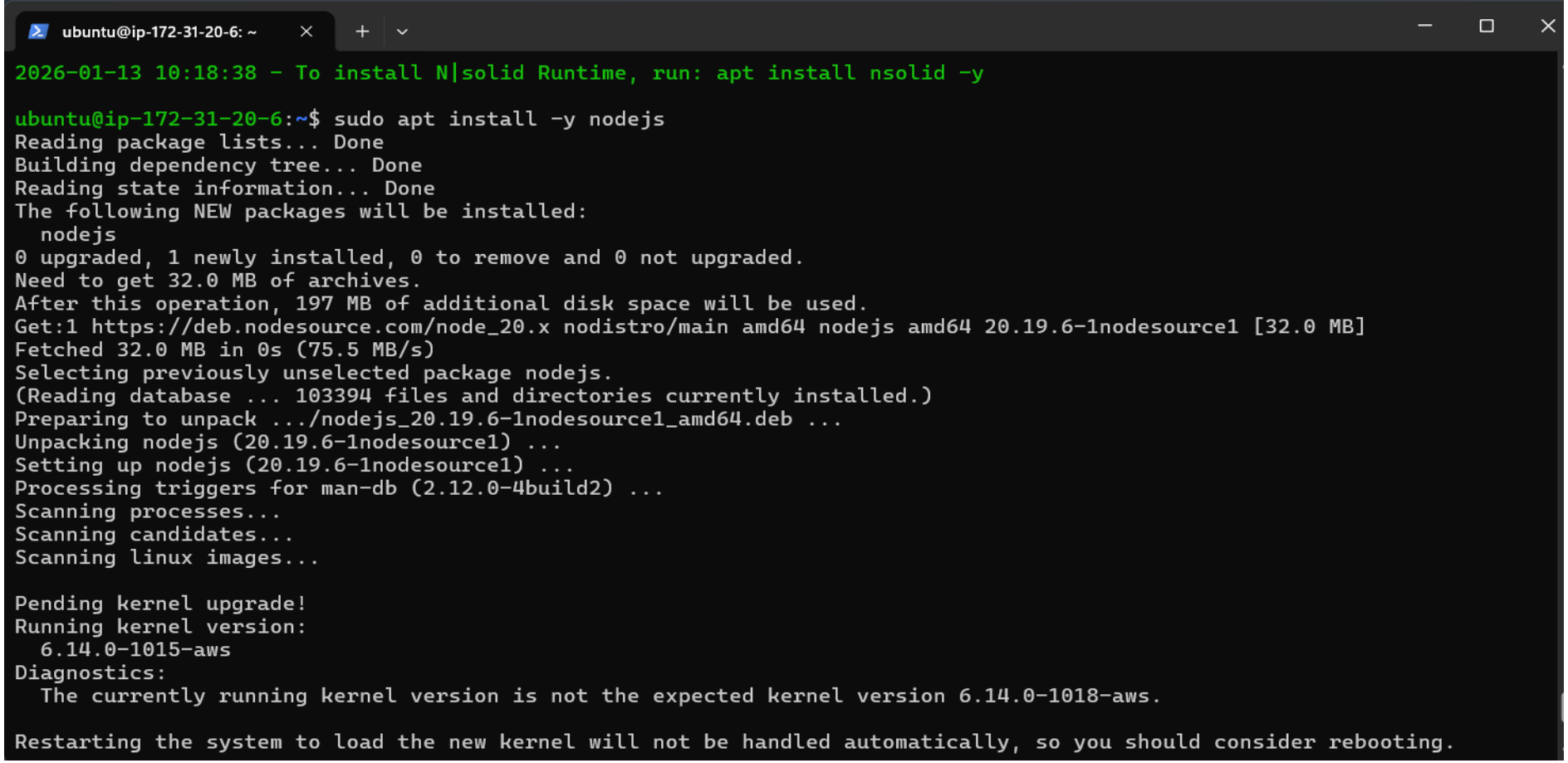
sudo apt update I'm updating the package list from Ubuntu repositories to get the latest information about available packages and their versions. This ensures I can install the most recent stable software versions.



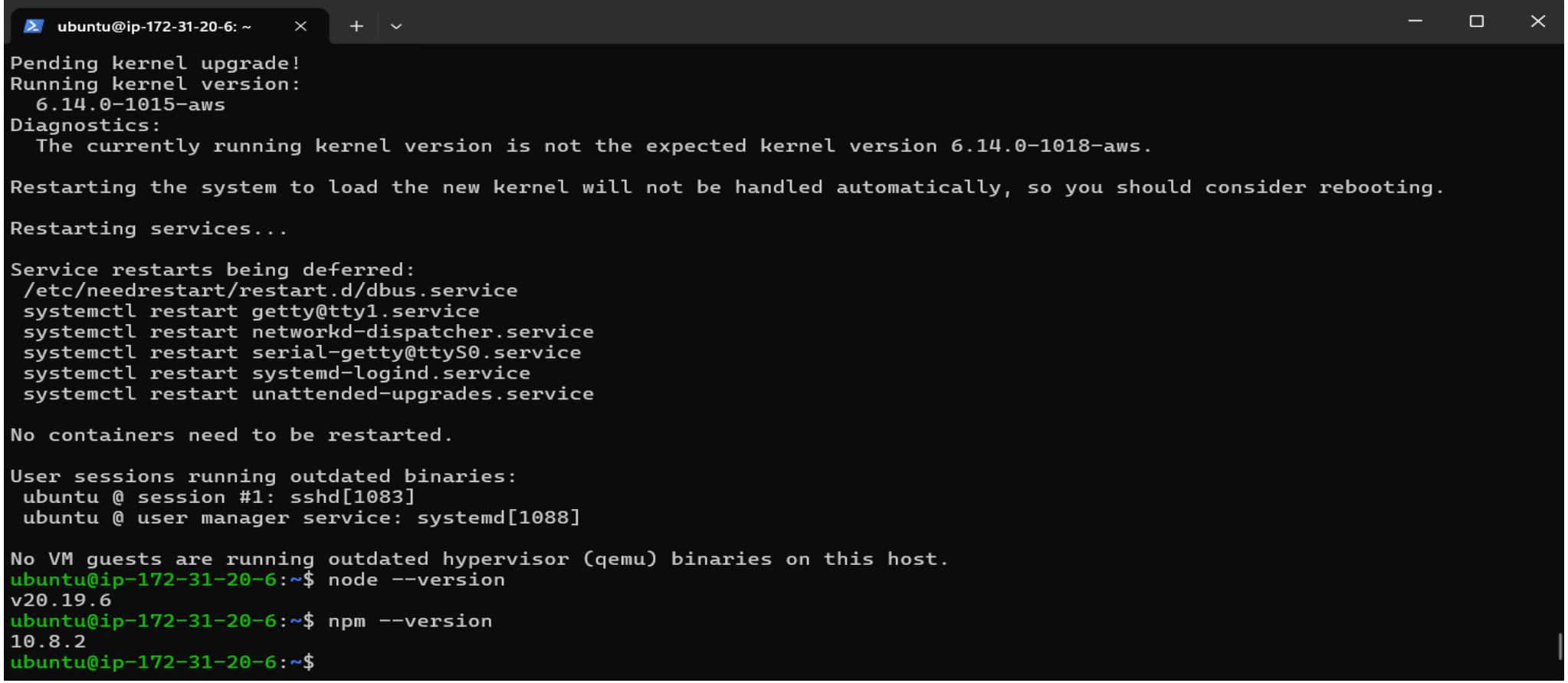
sudo apt upgrade -y I'm upgrading all currently installed packages to their latest versions using the -y flag for automatic yes to prompts. This keeps my system secure and up-to-date with bug fixes and improvements.



curl -fsSL https://deb.nodesource.com/setup_20.x | sudo -E bash - I'm downloading and executing the NodeSource setup script to add the Node.js 20.x repository to my system. This allows me to install the latest LTS version of Node.js instead of the older version in Ubuntu's default repositories.



sudo apt install -y nodejs I'm installing Node.js (which includes npm - Node Package Manager) from the NodeSource repository. Node.js is required to run my FastFood web application backend server.



node --version I'm verifying that Node.js installed correctly by checking its version number. This confirms the installation was successful and shows which version is now available on my system.

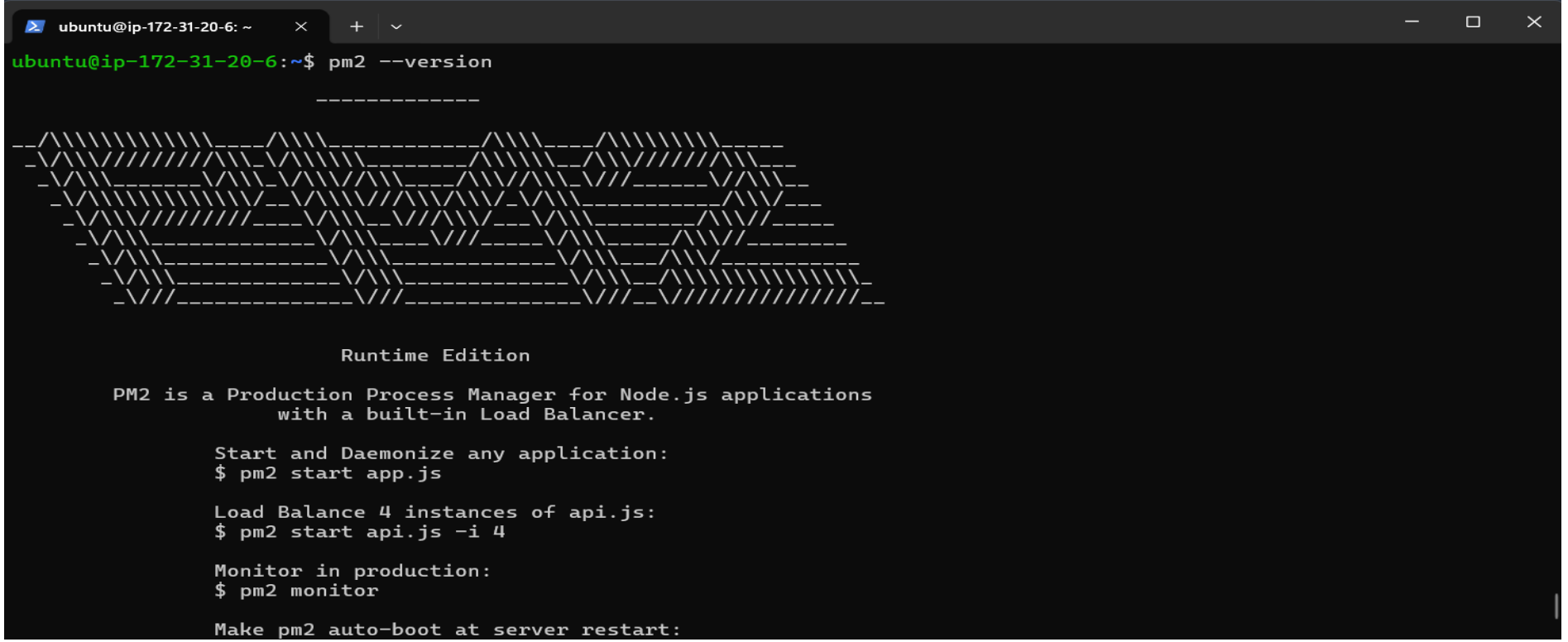
npm --version I'm checking the npm (Node Package Manager) version to confirm it was installed along with Node.js. npm is essential for managing JavaScript dependencies in my project.

```
ubuntu@ip-172-31-20-6:~$ sudo npm install -g pm2

added 133 packages in 8s

13 packages are looking for funding
  run `npm fund` for details
npm notice
npm notice New major version of npm available! 10.8.2 -> 11.7.0
npm notice Changelog: https://github.com/npm/cli/releases/tag/v11.7.0
npm notice To update run: npm install -g npm@11.7.0
npm notice
ubuntu@ip-172-31-20-6:~$
```

sudo npm install -g pm2 I'm installing PM2 globally using the -g flag, which is a production-grade process manager for Node.js applications. PM2 keeps my app running continuously, restarts it if it crashes, and manages logs efficiently.

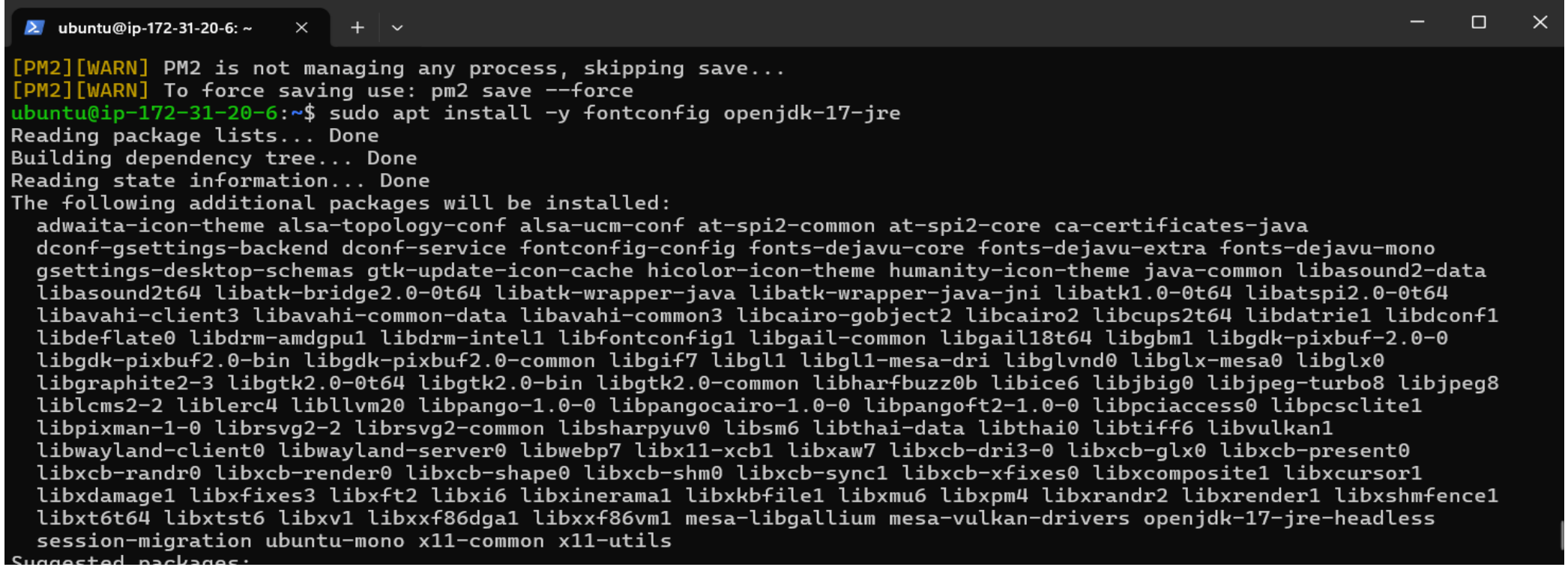


pm2 --version I'm verifying PM2 installation by checking its version number to ensure the global installation completed successfully and PM2 commands are available system-wide.

I'm executing the PM2-generated startup command to configure systemd to automatically launch PM2 (and all saved processes) when the system boots. This ensures my application starts automatically after server restarts without manual intervention.

```
ubuntu@ip-172-31-20-6:~$ pm2 save
[PM2] Saving current process list...
[PM2][WARN] PM2 is not managing any process, skipping save...
[PM2][WARN] To force saving use: pm2 save --force
ubuntu@ip-172-31-20-6:~$
```

pm2 save I'm saving the current list of PM2-managed processes to disk so they can be automatically resurrected on system reboot. Any apps currently running under PM2 will restart automatically after a reboot.



sudo apt install -y fontconfig openjdk-17-jre I'm installing Java 17 JRE (Java Runtime Environment) along with fontconfig, which are prerequisites for running Jenkins. Jenkins is a Java-based application that requires JRE to function.

```
ubuntu@ip-172-31-20-6:~$ java --version
openjdk 17.0.17 2025-10-21
OpenJDK Runtime Environment (build 17.0.17+10-Ubuntu-124.04)
OpenJDK 64-Bit Server VM (build 17.0.17+10-Ubuntu-124.04, mixed mode, sharing)
ubuntu@ip-172-31-20-6:~$
```

java --version I'm verifying the Java installation and checking which version is installed to confirm Java 17 is available and properly configured on my system.

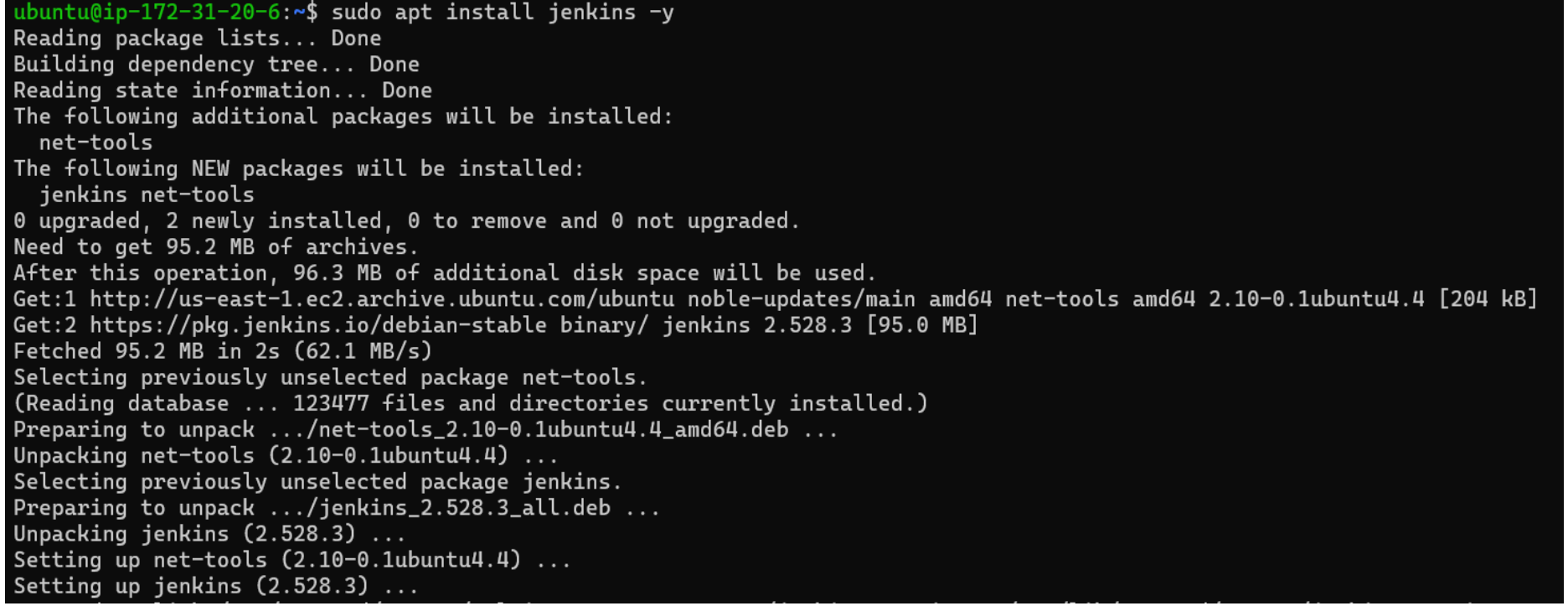
```
ubuntu@ip-172-31-20-6:~$ java --version
openjdk 17.0.17 2025-10-21
OpenJDK Runtime Environment (build 17.0.17+10-Ubuntu-124.04)
OpenJDK 64-Bit Server VM (build 17.0.17+10-Ubuntu-124.04, mixed mode, sharing)
ubuntu@ip-172-31-20-6:~$ curl -fsSL https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key | sudo tee /usr/share/keyri
ngs/jenkins-keyring.asc > /dev/null
ubuntu@ip-172-31-20-6:~$ echo "deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc] https://pkg.jenkins.io/debian-sta
ble binary/" | sudo tee /etc/apt/sources.list.d/jenkins.list
deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc] https://pkg.jenkins.io/debian-stable binary/
ubuntu@ip-172-31-20-6:~$
```

curl -fsSL https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key | sudo tee /usr/share/keyrings/jenkins-keyring.asc > /dev/null I'm downloading the official Jenkins GPG key and adding it to my system's keyring for package verification. This ensures that only authentic Jenkins packages from the official repository can be installed.

cho "deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc] https://pkg.jenkins.io/debian-stable binary/" | sudo tee /etc/apt/sources.list.d/jenkins.list I'm adding the Jenkins stable repository to my apt sources list with proper GPG key verification. This tells apt where to download Jenkins packages from and how to verify their authenticity.

```
ubuntu@ip-172-31-20-6:~$ sudo apt update
Hit:1 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble InRelease
Hit:2 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-updates InRelease
Hit:3 http://us-east-1.ec2.archive.ubuntu.com/ubuntu noble-backports InRelease
Hit:4 http://security.ubuntu.com/ubuntu noble-security InRelease
Ign:5 https://pkg.jenkins.io/debian-stable binary/ InRelease
Get:6 https://pkg.jenkins.io/debian-stable binary/ Release [2044 B]
Get:7 https://pkg.jenkins.io/debian-stable binary/ Release.gpg [833 B]
Hit:8 https://deb.nodesource.com/node_20.x nodistro InRelease
Get:9 https://pkg.jenkins.io/debian-stable binary/ Packages [30.3 kB]
Fetched 33.2 kB in 1s (65.3 kB/s)
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
All packages are up to date.
ubuntu@ip-172-31-20-6:~$
```

sudo apt update I'm updating the package list again to include packages from the newly added Jenkins repository. This allows apt to see and install Jenkins from the official Jenkins repository.

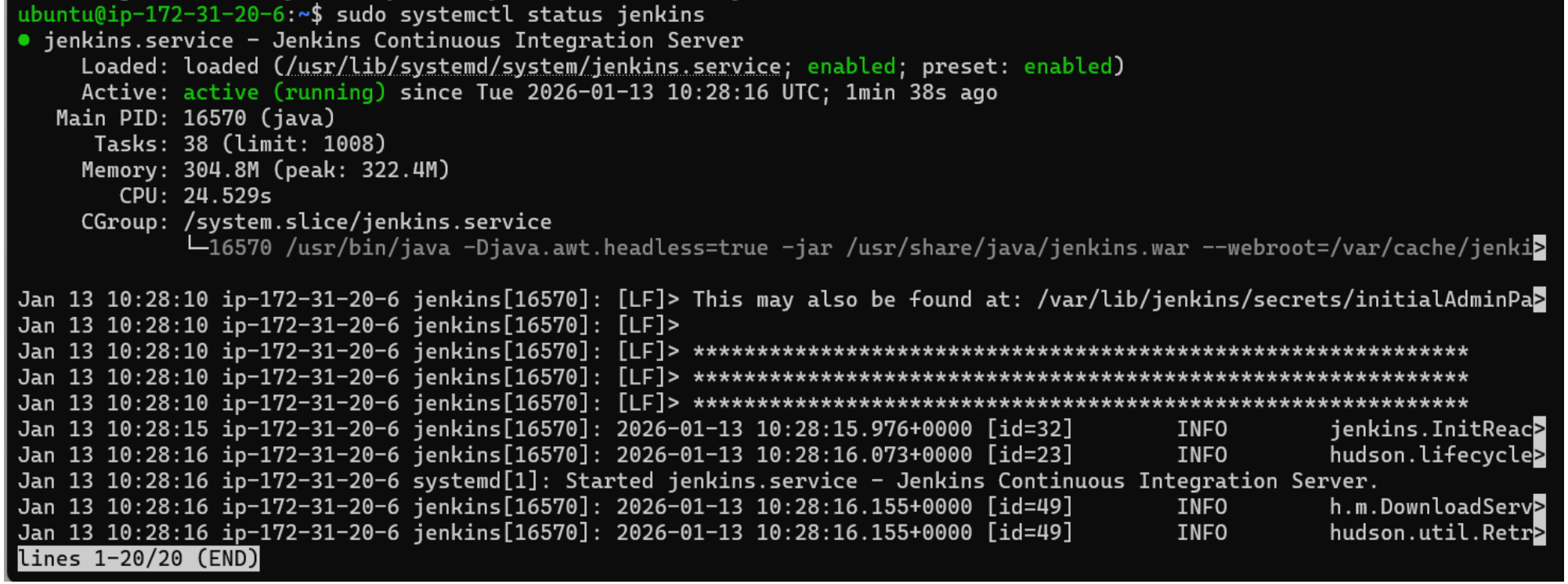


sudo apt install jenkins -y I'm installing Jenkins CI/CD automation server which will handle building, testing, and deploying my FastFood application automatically whenever I push code changes to GitHub.

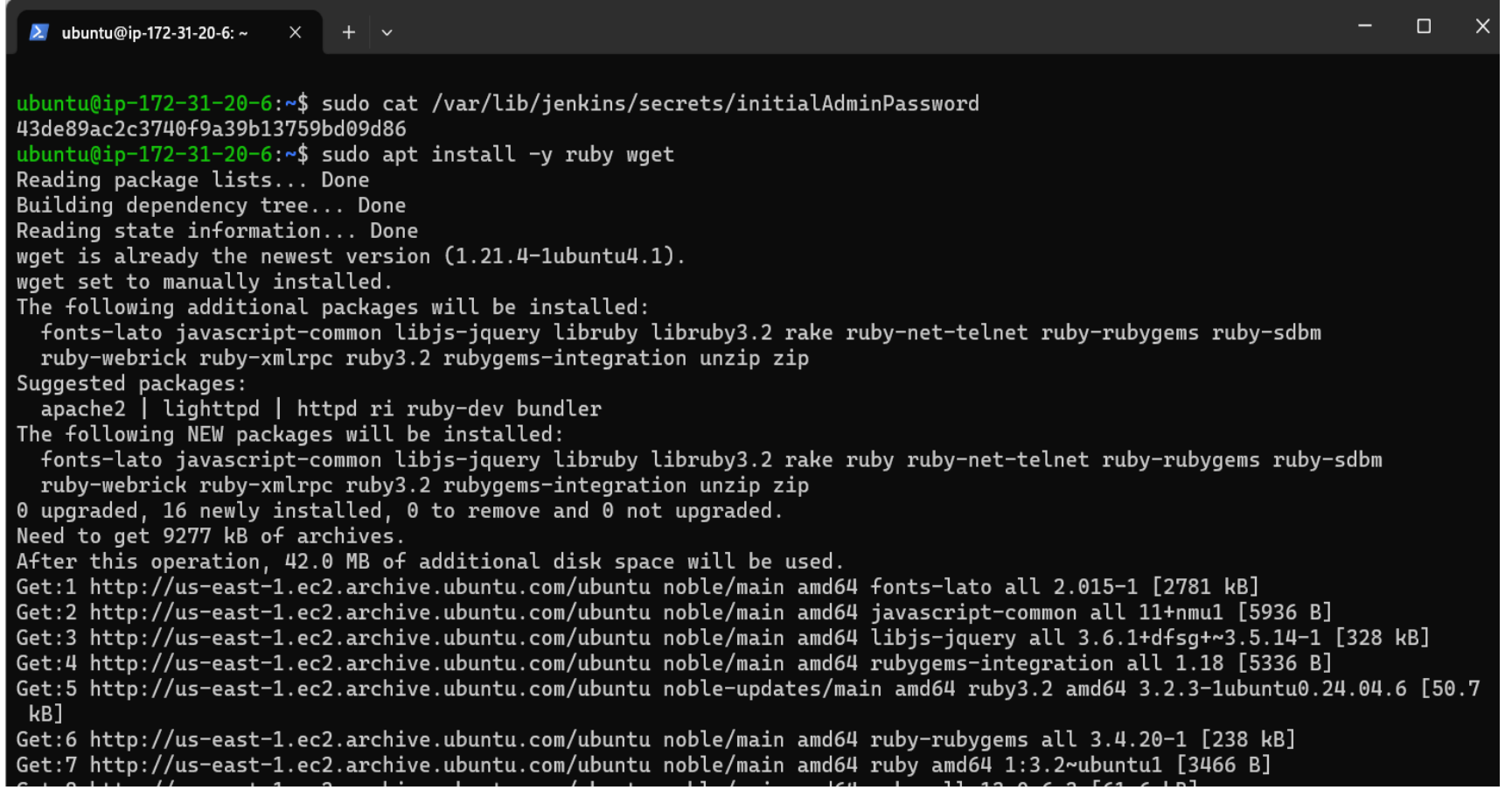
```
ubuntu@ip-172-31-20-6:~$ sudo systemctl start jenkins
ubuntu@ip-172-31-20-6:~$ sudo systemctl enable jenkins
Synchronizing state of jenkins.service with SysV service script with /usr/lib/systemd/systemd-sysv-install.
Executing: /usr/lib/systemd/systemd-sysv-install enable jenkins
ubuntu@ip-172-31-20-6:~$
```

sudo systemctl start jenkins I'm starting the Jenkins service for the first time. This launches the Jenkins web server which will be accessible on port 8080 for configuration and creating CI/CD pipelines.

sudo systemctl enable jenkins I'm configuring Jenkins to start automatically when the system boots. This ensures Jenkins is always available after server restarts without manual intervention.

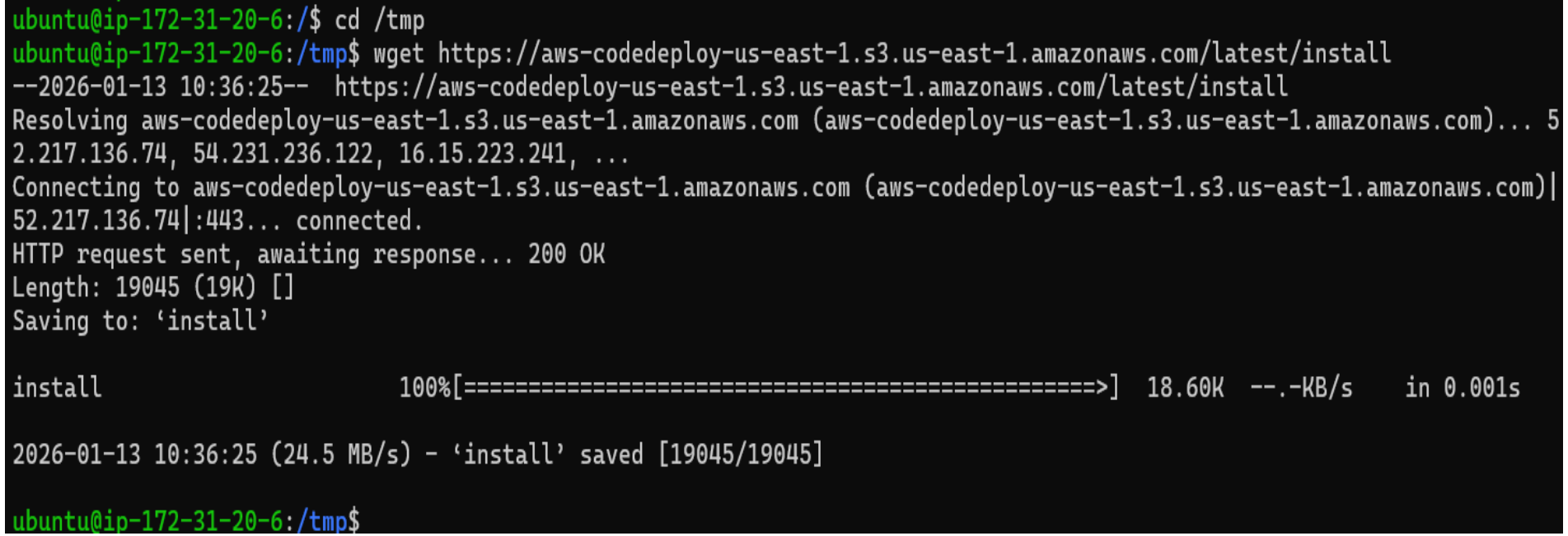


sudo systemctl status jenkins I'm checking the Jenkins service status to verify it's running properly and to see its current state, PID, and any recent log messages.

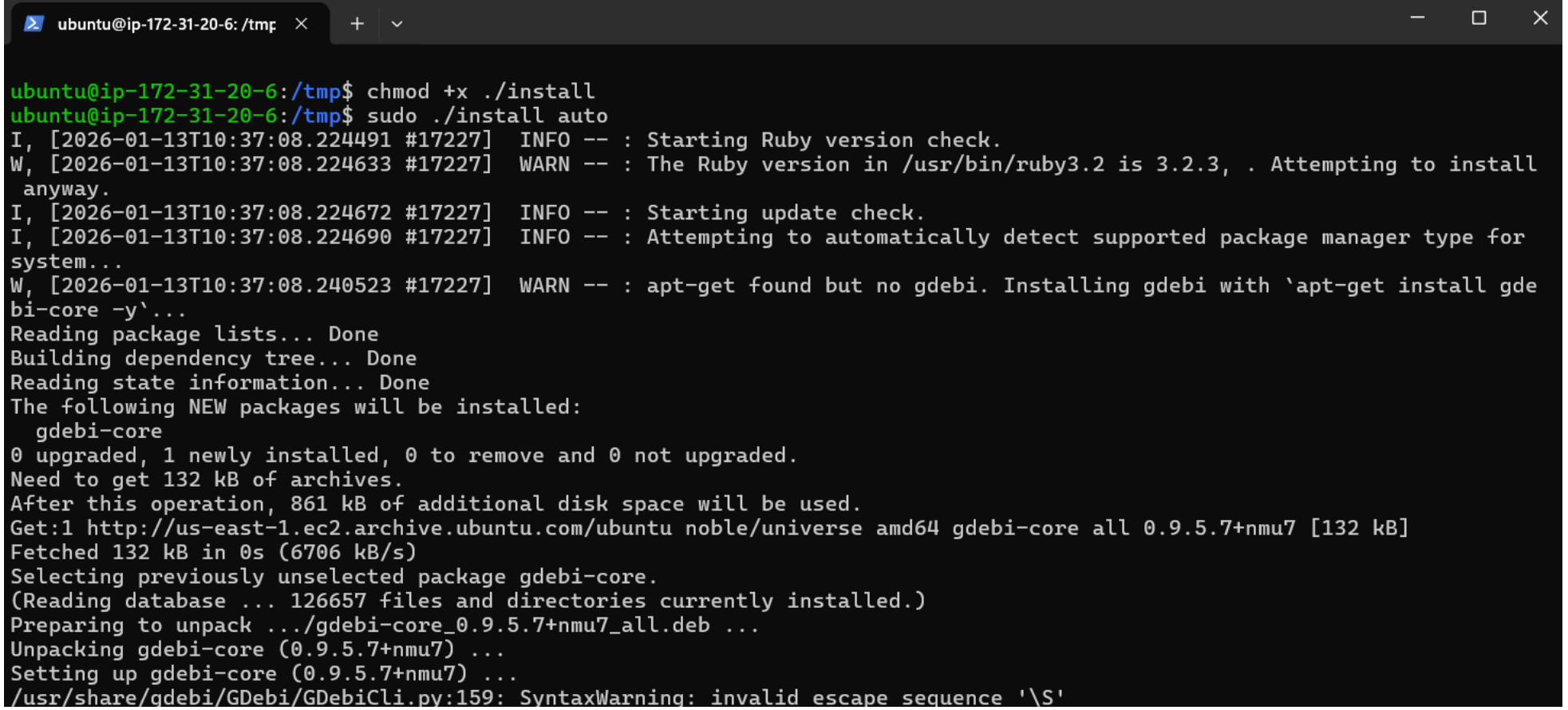


sudo cat /var/lib/jenkins/secrets/initialAdminPassword I'm displaying the initial admin password that Jenkins generated during installation. I need this password to complete the Jenkins setup wizard on first login through the web interface at port 8080.

sudo apt install -y ruby wget I'm installing Ruby and wget which are dependencies required by the AWS CodeDeploy agent installer script. Ruby is needed to run the agent, and wget is used to download files.

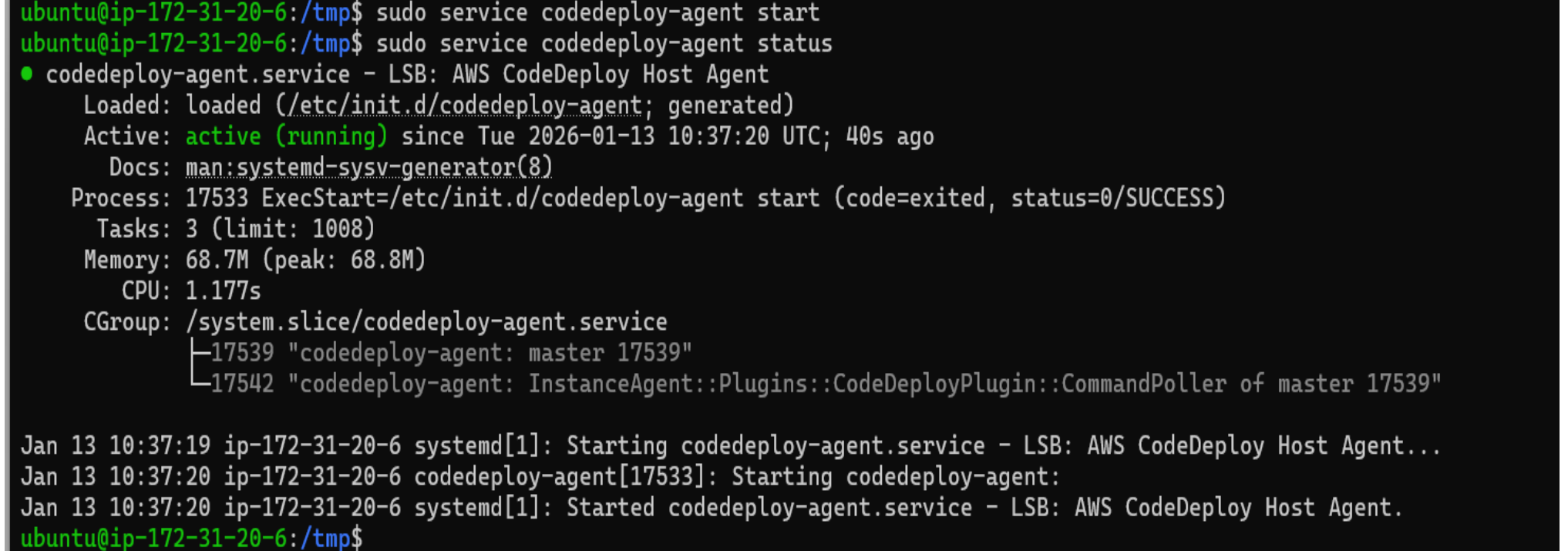


cd /tmp I'm changing to the temporary directory where I'll download the CodeDeploy agent installer. The /tmp directory is appropriate for temporary downloads that don't need to be preserved.



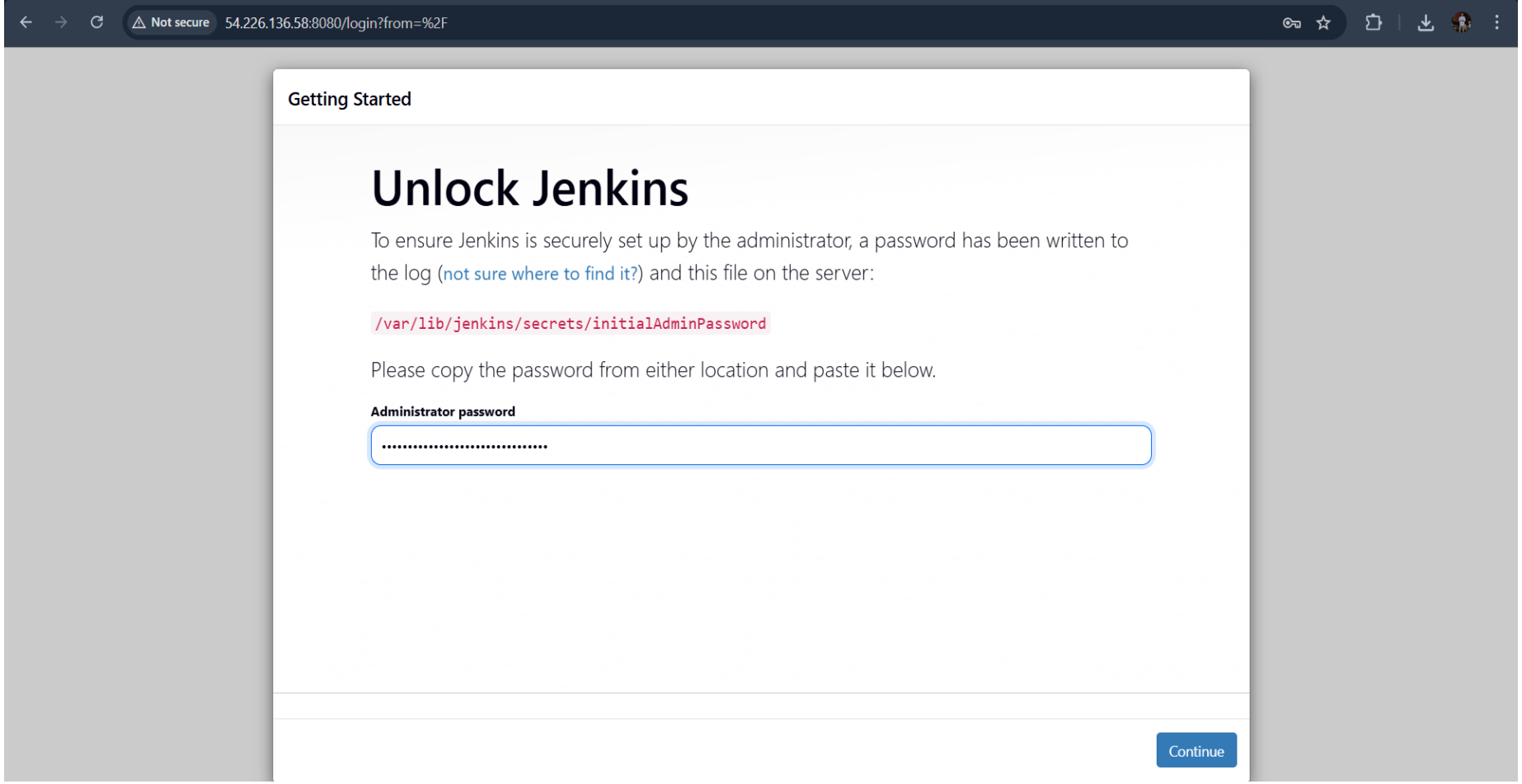
chmod +x ./install I'm making the downloaded install script executable by adding execute permissions. This allows me to run the script as a program to install the CodeDeploy agent.

sudo ./install auto I'm running the CodeDeploy agent installer with the auto parameter to automatically install and configure the agent. The agent will listen for deployment commands from AWS CodeDeploy service.



sudo service codedeploy-agent start I'm starting the CodeDeploy agent service using the legacy service command. This launches the agent so it can receive and execute deployment instructions from AWS CodeDeploy.

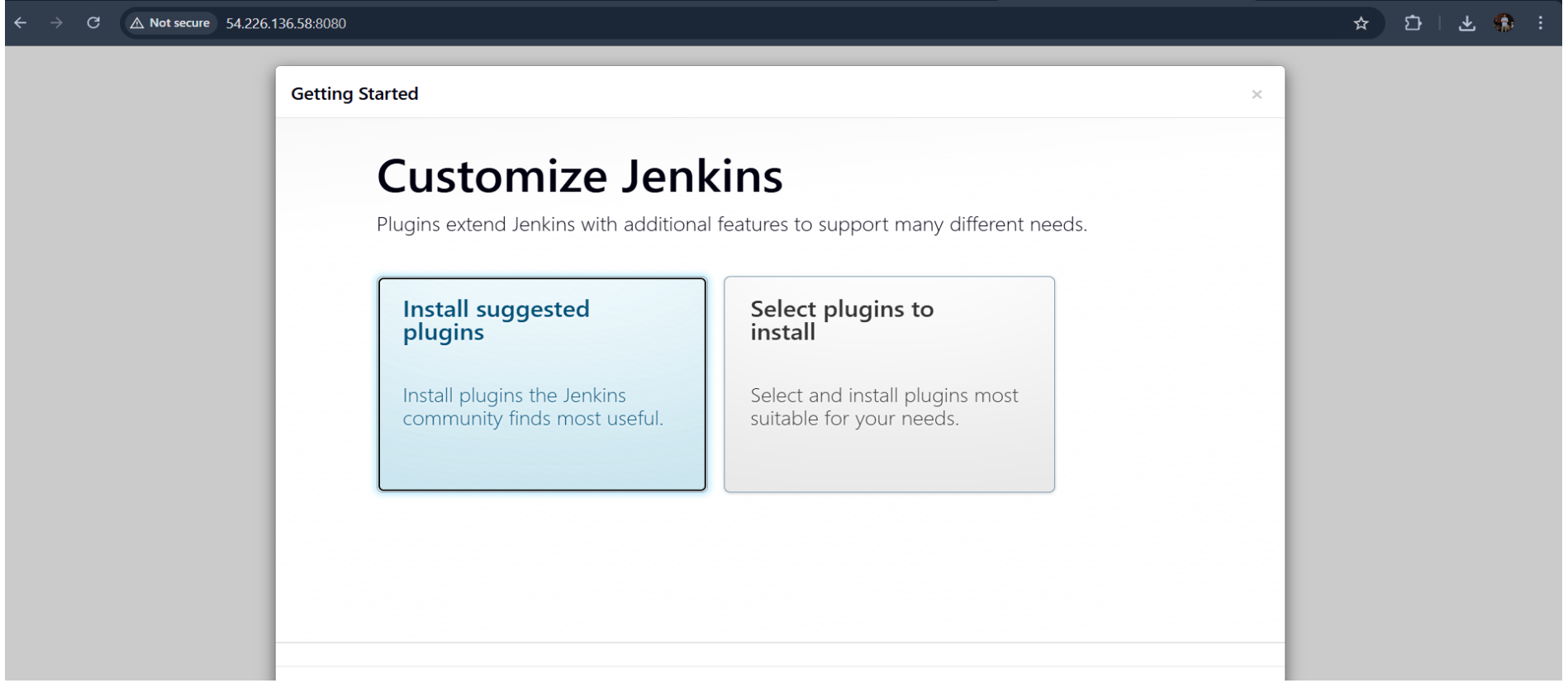
sudo service codedeploy-agent status I'm checking the CodeDeploy agent status using the legacy service command to verify it's running and ready to handle deployments.



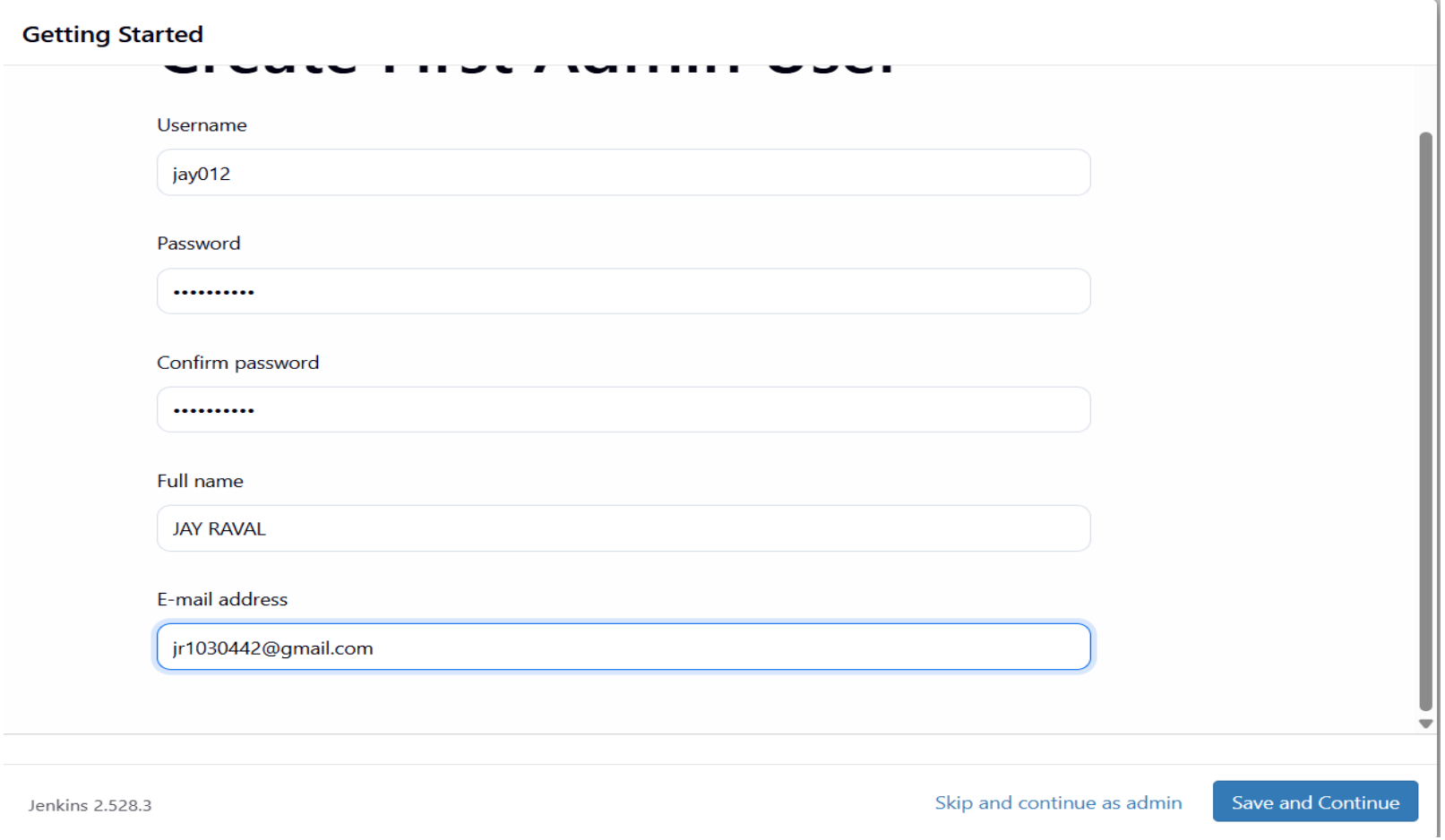
- This screen confirms Jenkins is successfully installed and running on your EC2 instance.
- It prompts for the initial admin password stored at .

/var/lib/Jenkins/secreats/initialAdminpassword

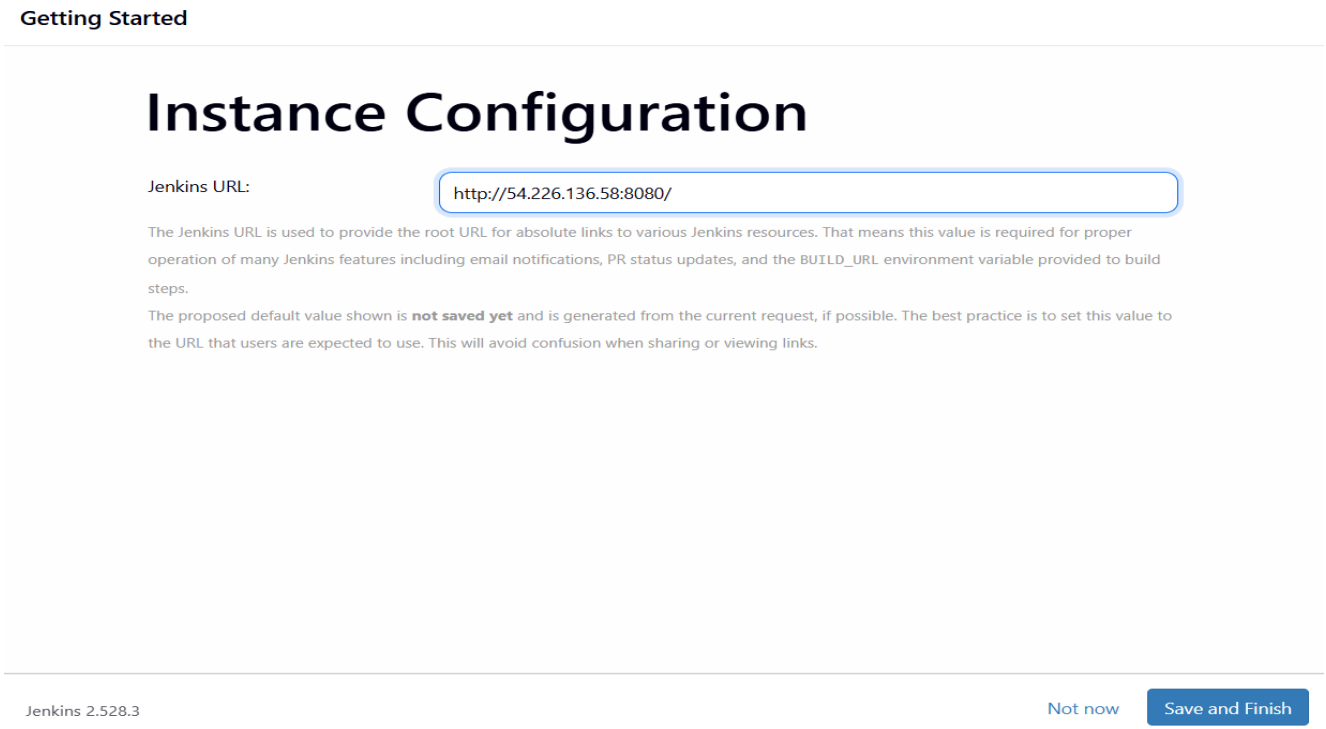
- This step ensures only the EC2 administrator can complete Jenkins setup securely.
- Once entered, you'll proceed to plugin installation and user creation.



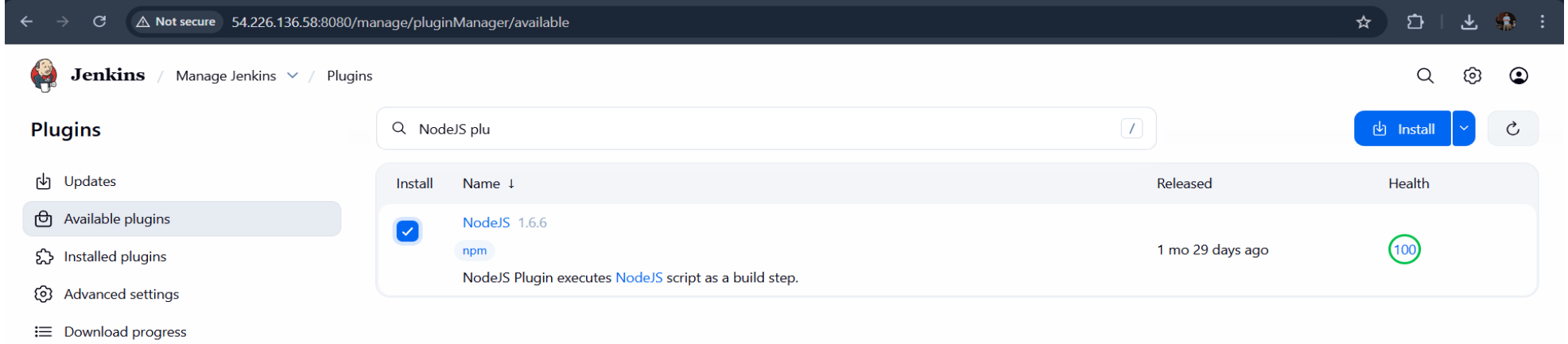
- The “Install suggested plugins” option is ideal for beginners—it includes commonly used tools like Git, Pipeline, and SSH.
- The “Select plugins to install” option lets you manually choose plugins based on my project needs.
- This step sets up Jenkins for CI/CD tasks like code integration, testing, and deployment.



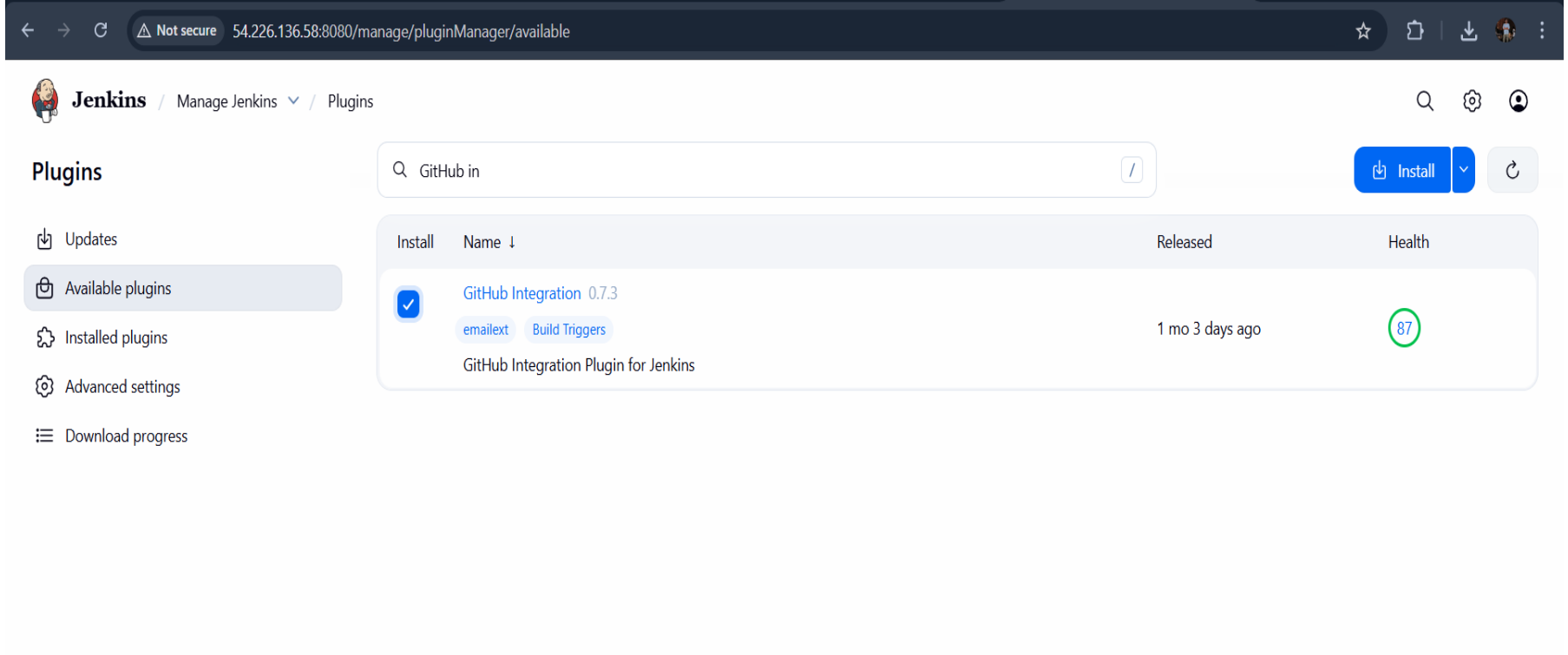
- This screen lets me create the first admin account to manage Jenkins securely.
- I've entered a custom username, full name, and email to personalize my Jenkins environment.
- The password I set will be used for future logins to the Jenkins dashboard.
- Clicking "Save and Continue" finalizes my admin setup and moves me to instance configuration.



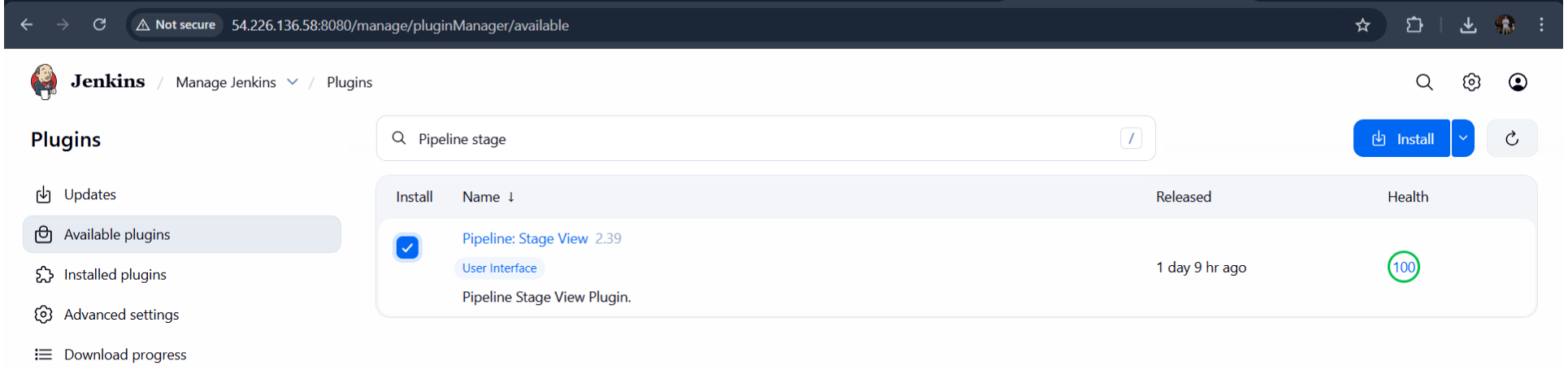
- This screen sets the base URL Jenkins uses for links, notifications, and environment variables like .
- The default value is my EC2 public IP with port 8080, which works fine for direct access.
- If I later use a domain or reverse proxy, I'll want to update this URL accordingly.
- Clicking "Save and Finish" finalizes the setup and activates my Jenkins dashboard.



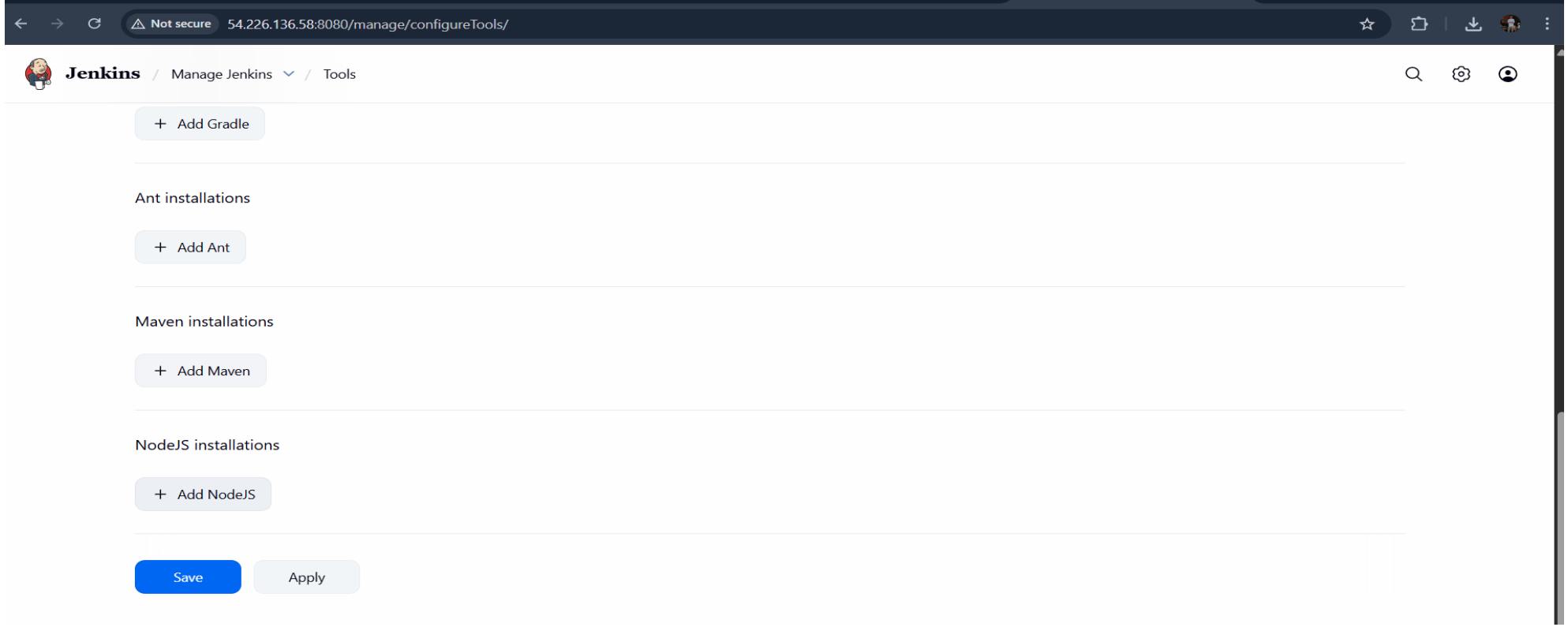
- I selected the NodeJS plugin, which enables Jenkins to run Node.js scripts during builds.
- This is essential for automating tasks like testing, packaging, or deploying JavaScript-based applications.
- The plugin is actively maintained and reliable for my CI/CD pipelines.
- Once installed, I can configure NodeJS versions and use them in my job build steps.



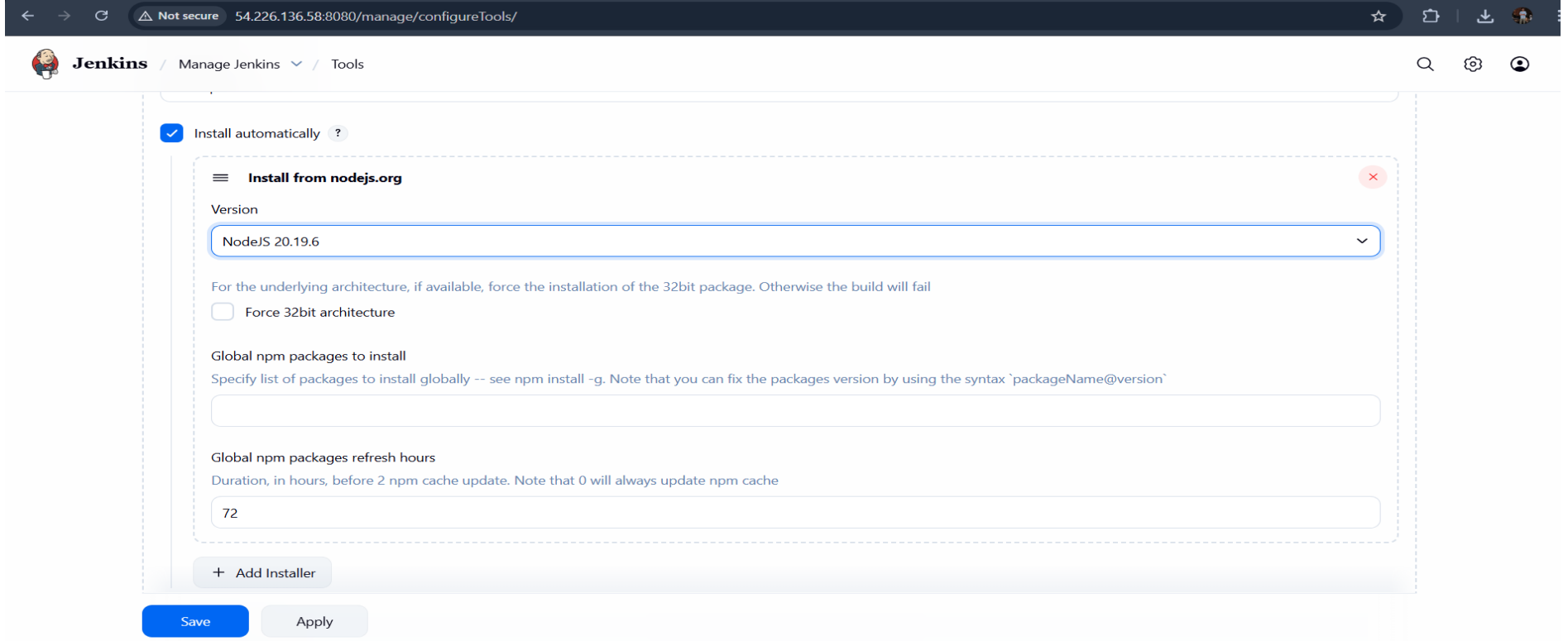
- I selected the GitHub Integration plugin, which enables Jenkins to interact with my GitHub repositories.
- It supports webhook triggers, pull request status updates, and Git-based build automation.
- The plugin integrates well with my CI/CD workflows.
- Once installed, I can link Jenkins jobs to GitHub projects for seamless code deployment.



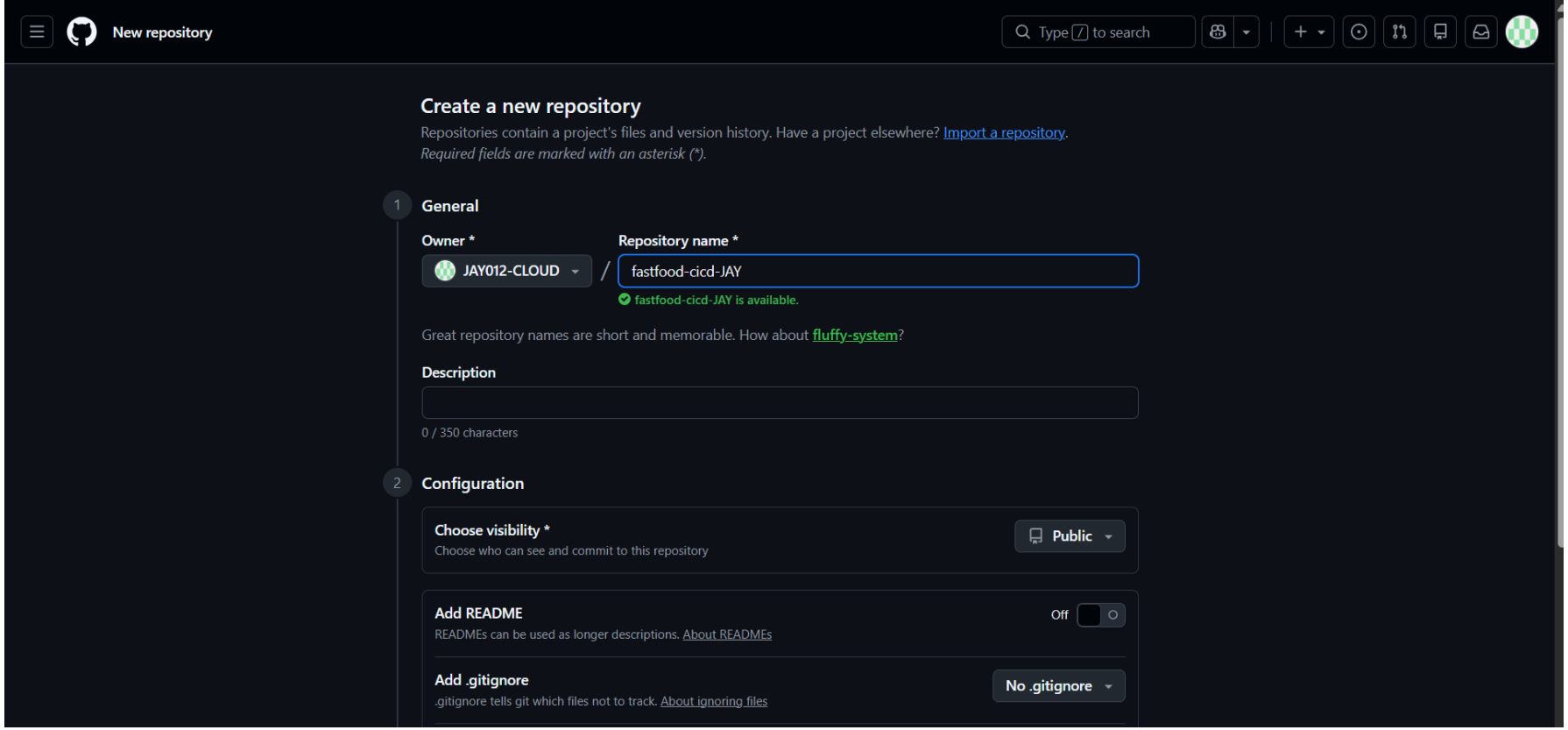
- I selected the Pipeline: Stage View plugin, which adds a visual breakdown of pipeline stages in my Jenkins jobs.
- It helps me monitor each stage's status, duration, and logs in a clean UI.
- This is especially useful for debugging and optimizing my multi-step build processes.
- With this plugin, Jenkins becomes more intuitive for tracking my complex workflows.



- Editing Node.js here lets me specify the name (used in jobs), the Node.js version, and optionally the installation path.
- This ensures Jenkins jobs can automatically use the correct Node.js runtime without needing manual setup on the EC2 instance.



- I enabled automatic installation so Jenkins fetches Node.js directly from nodejs.org during job execution.
- I selected version 20.19.6, ensuring consistent runtime across all builds.

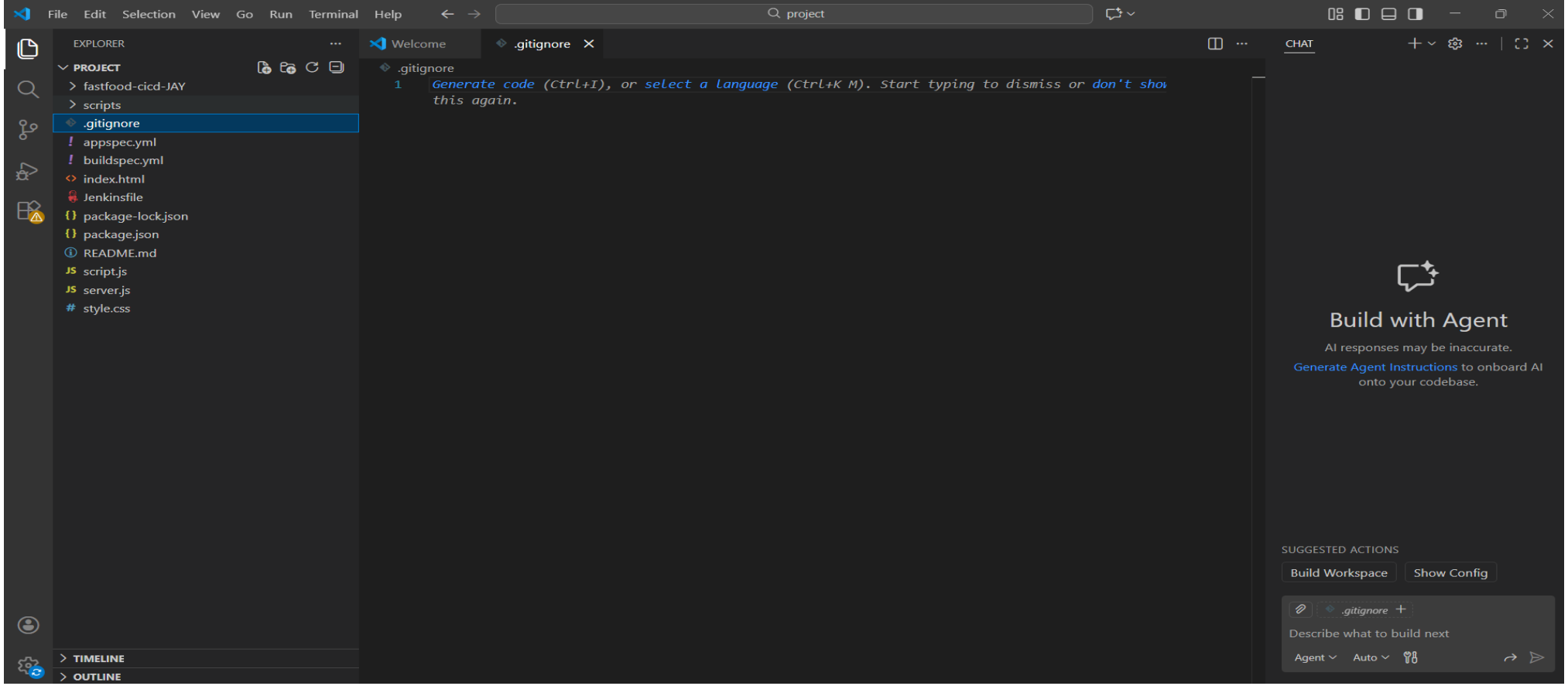


- I created a repository named fastfood-cicd-JAY to store my application code and CI/CD configuration files.
- This repo will hold important files like Jenkinsfile, buildspec.yml, and appspec.yml, which define my automation pipeline.

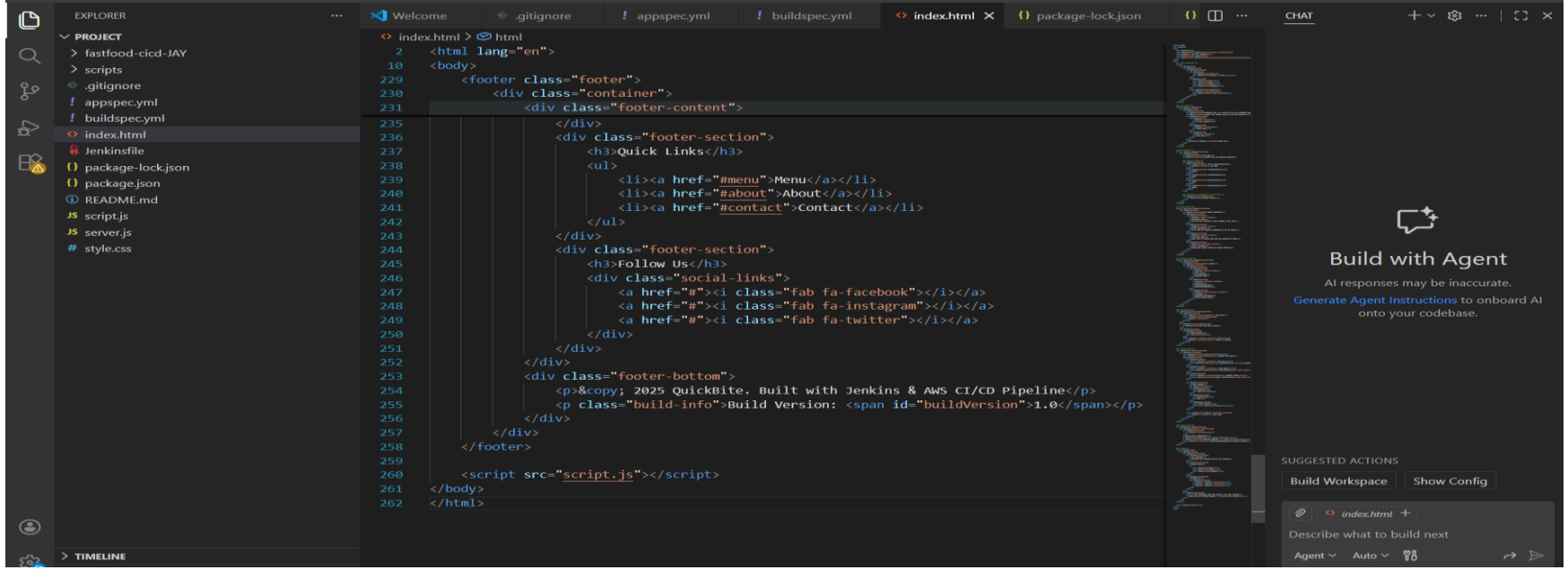
```
ubuntu@ip-172-31-28-21:~$ mkdir project
ubuntu@ip-172-31-28-21:~$ cd project
ubuntu@ip-172-31-28-21:~/project$ git clone https://github.com/JAY012-CLOUD/fastfood-cicd-JAY
Cloning into 'fastfood-cicd-JAY'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (3/3), done.
ubuntu@ip-172-31-28-21:~/project$ cd fastfood-cicd-JAY
ubuntu@ip-172-31-28-21:~/project/fastfood-cicd-JAY$
```

cd downloads I'm navigating to my downloads directory where I'll clone the FastFood CI/CD project repository from GitHub.

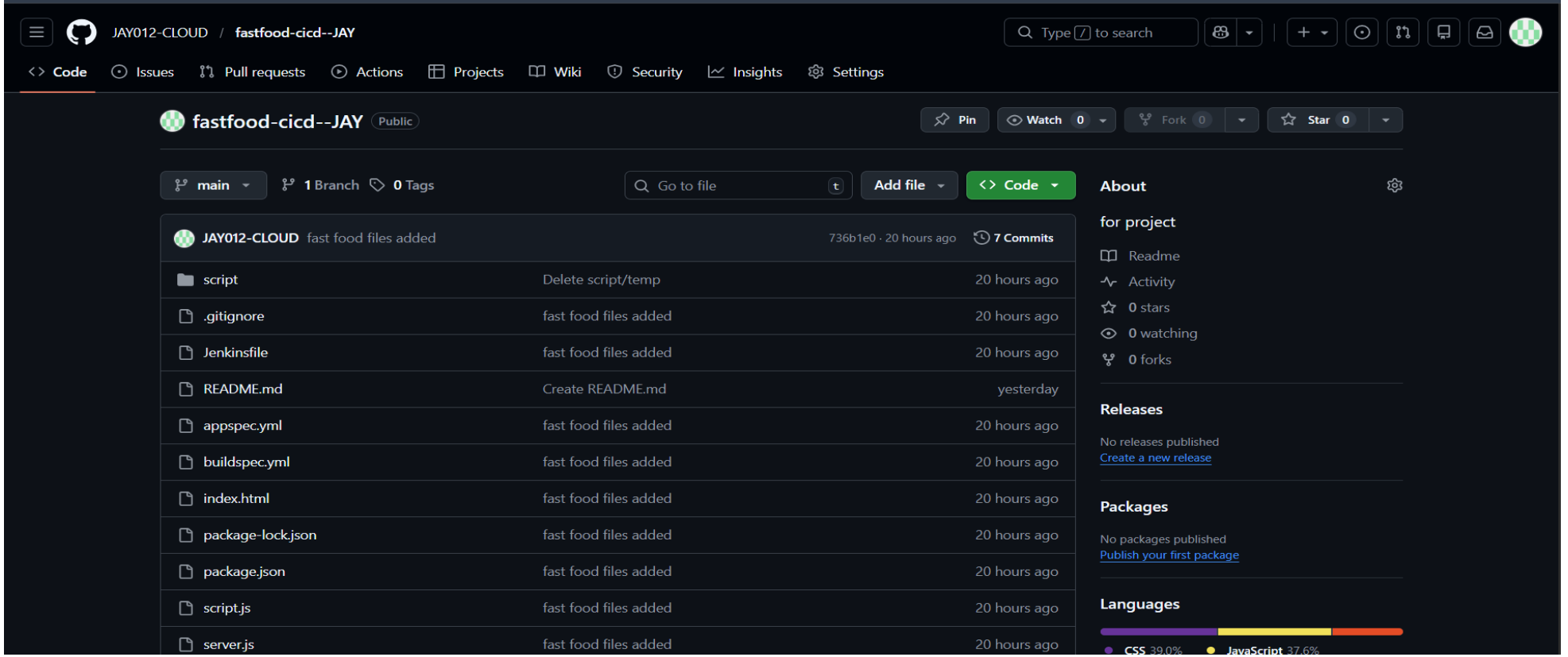
I'm cloning an existing FastFood CI/CD project repository from GitHub to my local machine. This downloads all project files, commit history, and branches to use as a starting template.



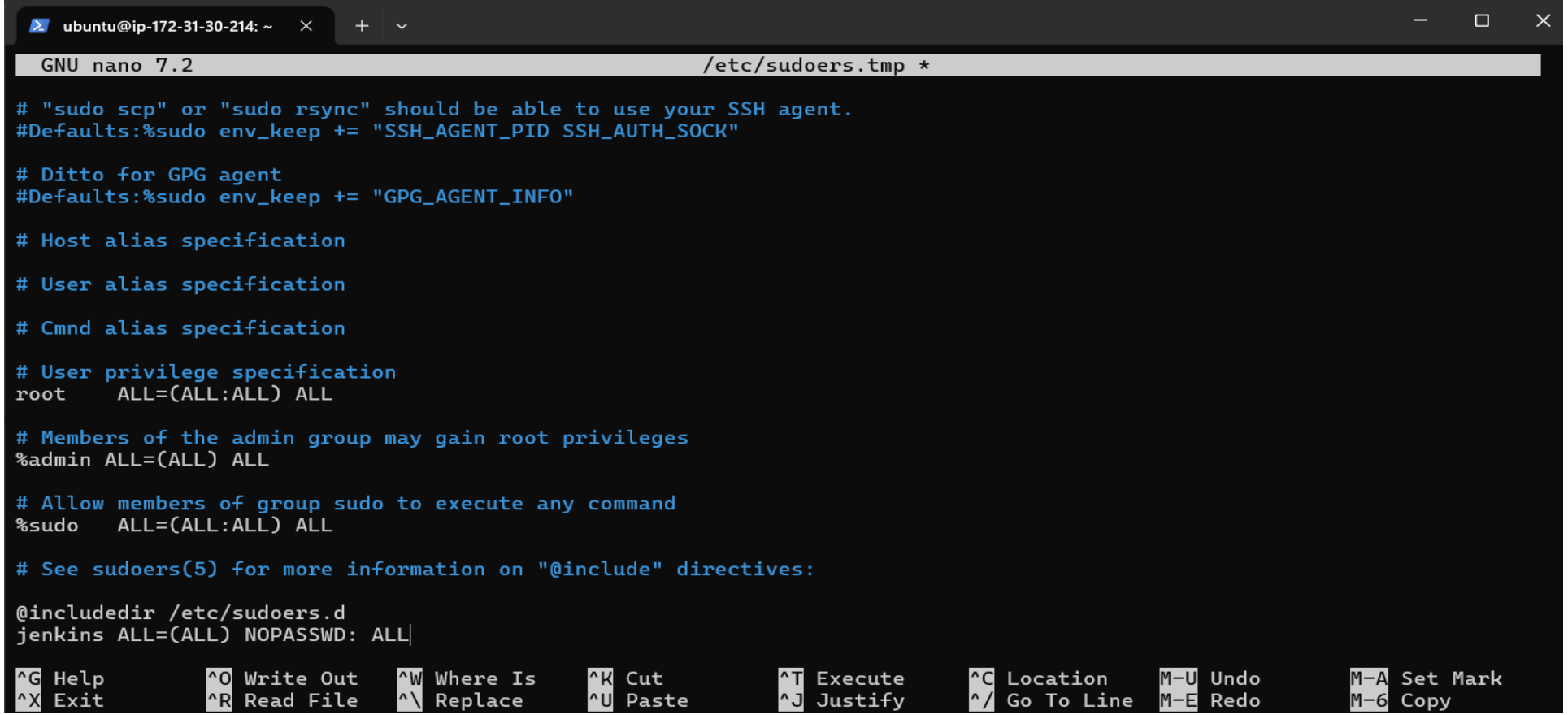
- I opened my project folder in VS Code, showing key files like Jenkinsfile, buildspec.yml, and appspec.yml for CI/CD automation.
- The presence of package.json and server.js confirms this is a Node.js-based web app.
- The Jenkinsfile will define my pipeline stages, while buildspec.yml and appspec.yml support AWS CodeBuild and CodeDeploy.
- This setup is ready to connect with GitHub and Jenkins for automated testing and deployment.



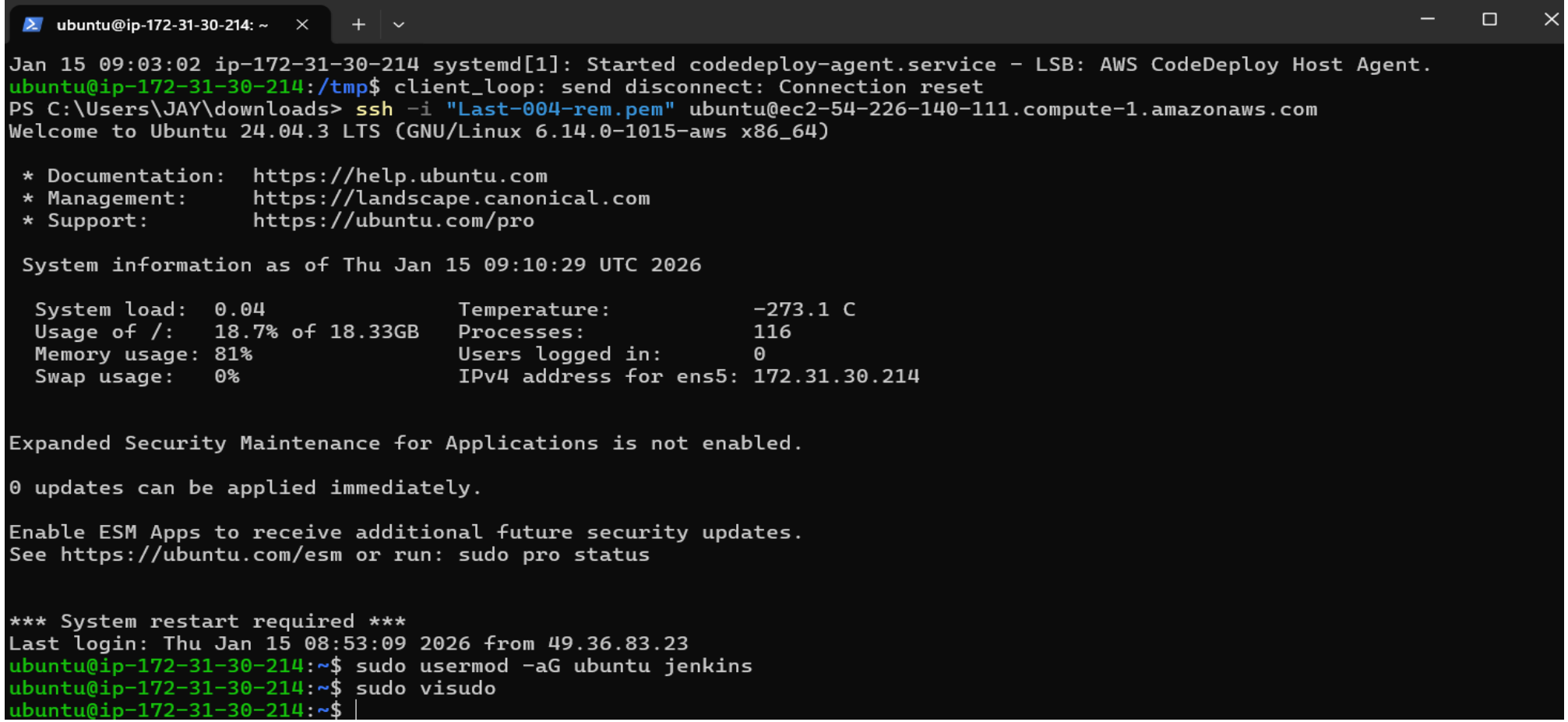
🌐 index.html
This is the main HTML file for your web app. It includes navigation, content sections, and a footer that displays build metadata like version and CI/CD tools used (Jenkins & AWS).
📜 Jenkinsfile
Defines your CI/CD pipeline stages. Jenkins reads this file to know how to build, test, and deploy your app. It's the heart of your automation workflow.
🧪 buildspec.yml
Used by AWS CodeBuild to define build commands, environment variables, and artifacts. It ensures your app is built consistently in the cloud.
🚀 appspec.yml
Used by AWS CodeDeploy to manage deployment steps, hooks, and file locations. It tells AWS how to deploy your app to EC2 or other targets.
📦 package.json
Lists your Node.js dependencies, scripts, and metadata. It's essential for installing packages and running build/test commands.
🔒 package-lock.json
Locks the exact versions of dependencies to ensure consistent builds across environments.
🖥️ server.js
Your Node.js backend file. It likely handles routing, server logic, and API endpoints for your web app.
📄 README.md
Provides documentation for your project — purpose, setup instructions, and usage notes. Useful for collaborators and future reference.
📁 script.js
Contains client-side JavaScript for interactivity, such as handling user actions or dynamic content updates.
🎨 style.css
Defines the visual styling of your web app — layout, colors, fonts, and responsiveness.
📁 scripts/
A folder likely containing deployment or utility scripts used in your CI/CD pipeline or local development.
🗂️ .gitignore
Specifies which files and folders Git should ignore (e.g., , logs, temp files). Keeps your repo clean and focused.



- I pushed my CI/CD project to GitHub under the repo name fastfood-cicd--JAY, hosted publicly on the main branch.
- The repo contains key files like Jenkinsfile, buildspec.yml, appspec.yml, and server.js — all essential for automated deployment.
- With 7 commits so far, I've begun versioning my code and tracking changes, including cleanup and initial setup.
- This repo is now ready to integrate with Jenkins for continuous build and deployment from GitHub to AWS.



- I edited the sudoers file to grant the jenkins user passwordless sudo access.
- The line added is: jenkins ALL=(ALL) NOPASSWD: ALL
- This allows Jenkins to run commands with elevated privileges during automation tasks without asking for a password.
- It is useful for CI/CD pipelines that need system-level operations like deployments, file transfers, or service restarts.
- I made sure to edit it safely using visudo to avoid syntax errors that could break sudo access.



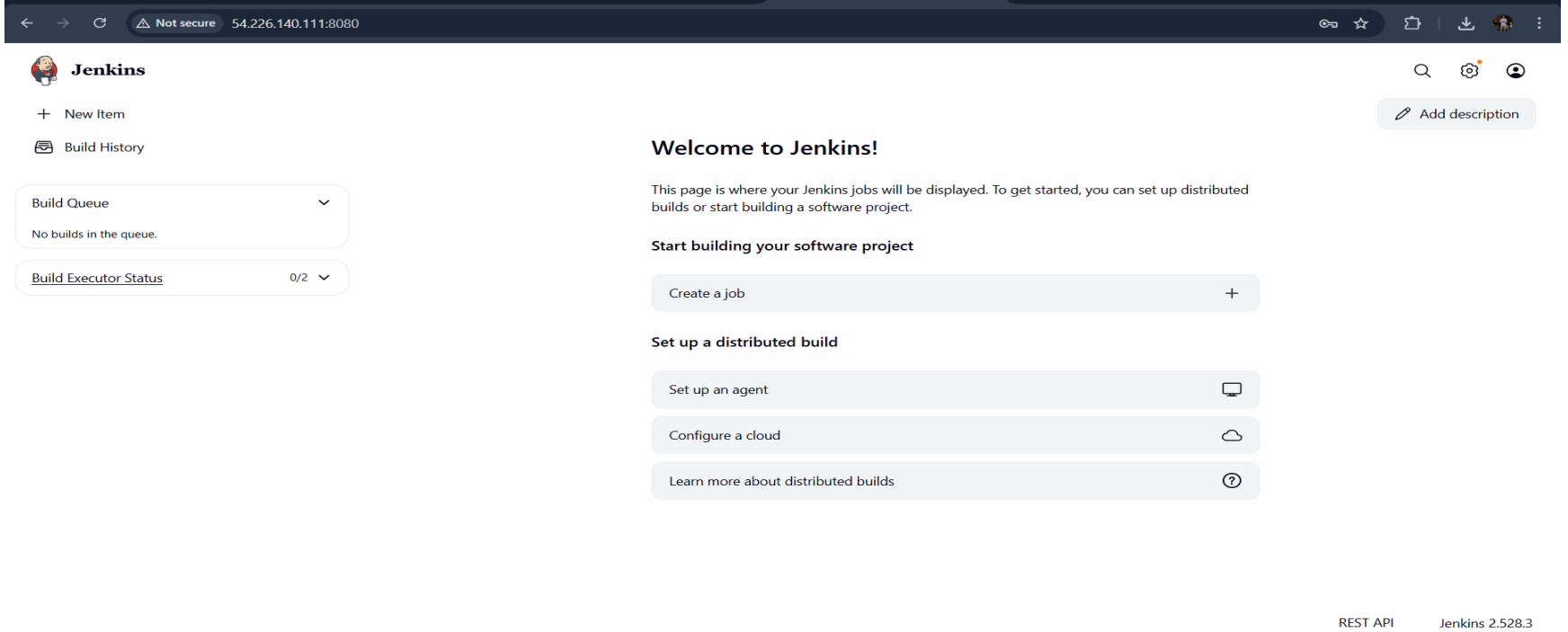
sudo usermod -aG ubuntu jenkins I'm adding the jenkins user to the ubuntu group, granting Jenkins permission to access files owned by the ubuntu user. This is necessary for Jenkins to deploy and manage application files.

sudo visudo I'm opening the sudoers configuration file using the safe visudo editor to grant Jenkins sudo permissions if needed. This allows Jenkins to execute privileged commands during deployment.

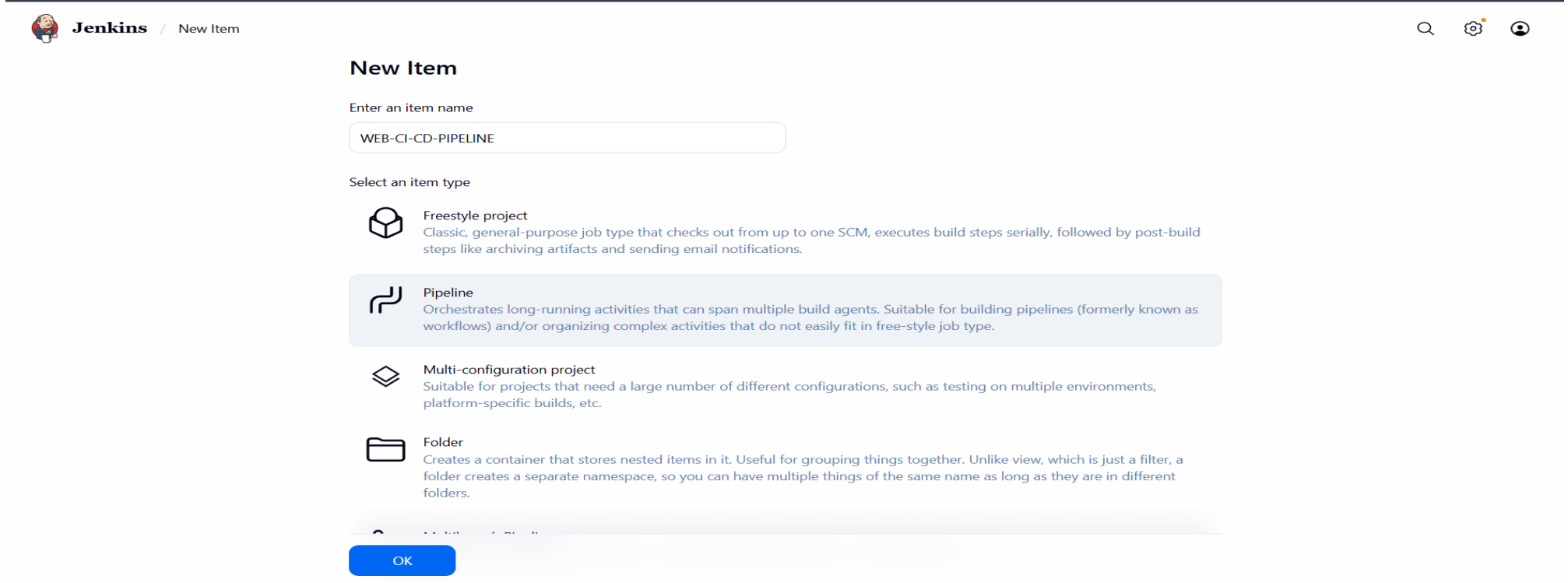
```
ubuntu@ip-172-31-30-214:~$ sudo systemctl restart jenkins

ubuntu@ip-172-31-30-214:~$
```

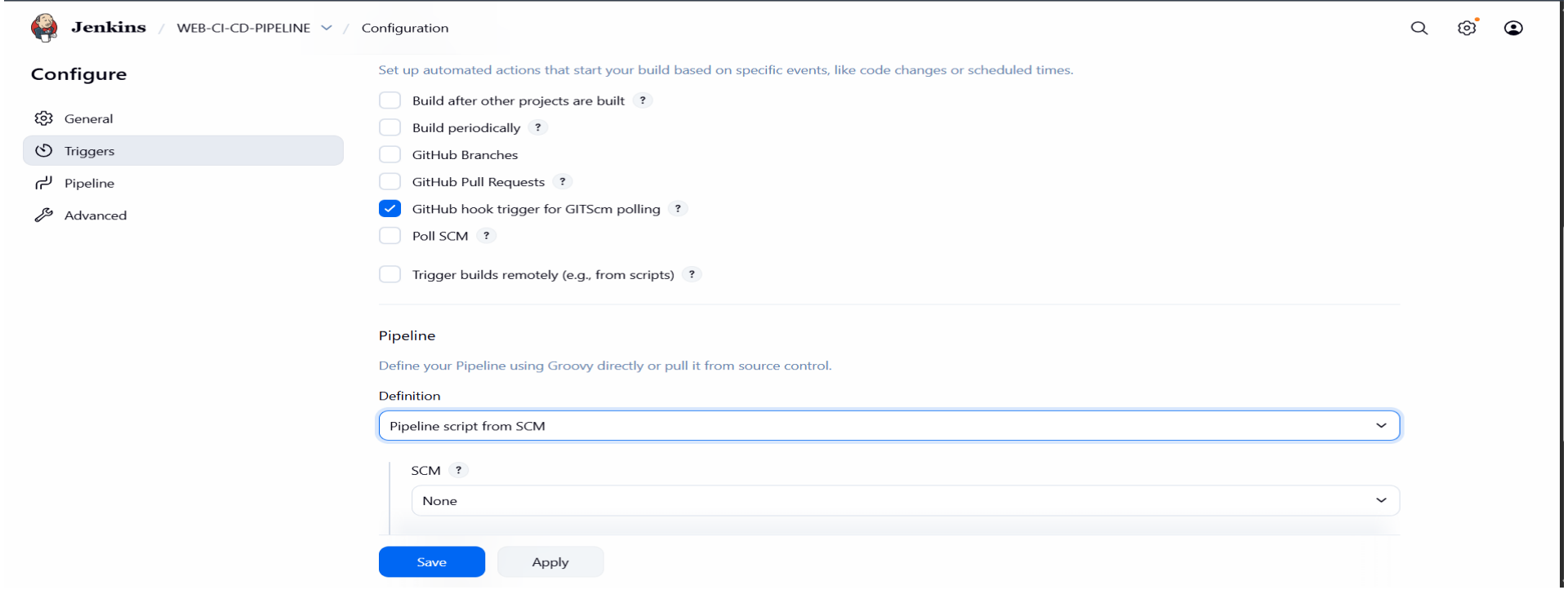
sudo systemctl restart jenkins I'm restarting the Jenkins service to apply the user group membership changes made in the previous step. Jenkins needs to restart to recognize its new group permissions.



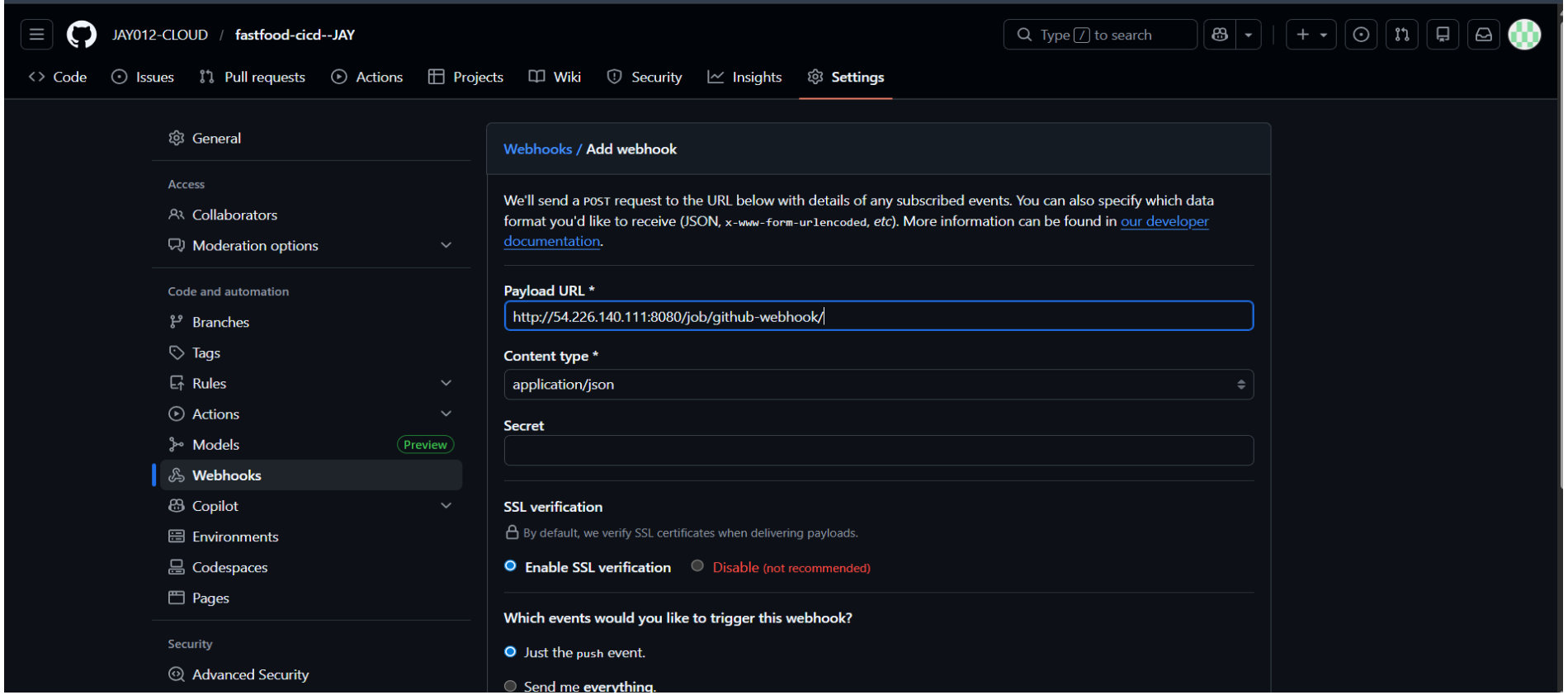
- I accessed Jenkins via browser at IP 54.226.140.111:8080 and reached the main dashboard.
- The interface confirms Jenkins is running and ready to manage CI/CD jobs.



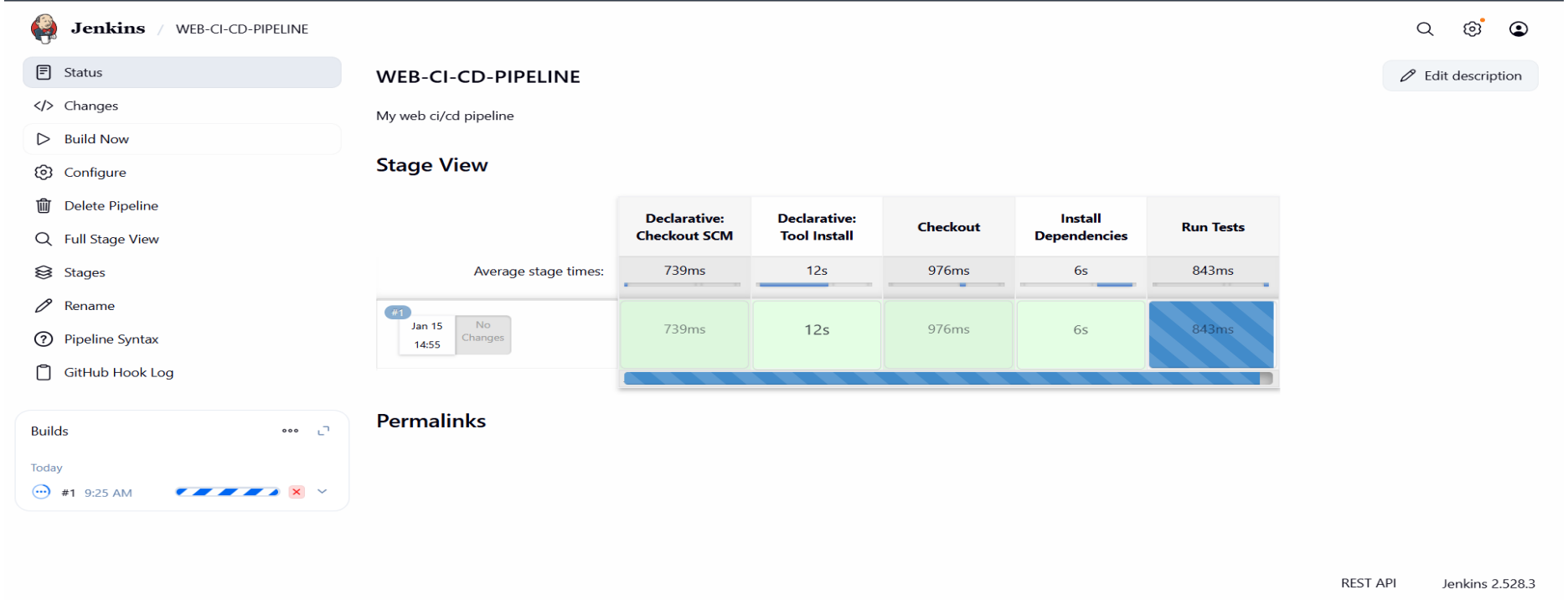
- I created a new Jenkins job named WEB-CI-CD-PIPELINE to manage the CI/CD flow for my web application.
- I selected the Pipeline type, which is ideal for defining multi-stage automation like build, test, and deploy.



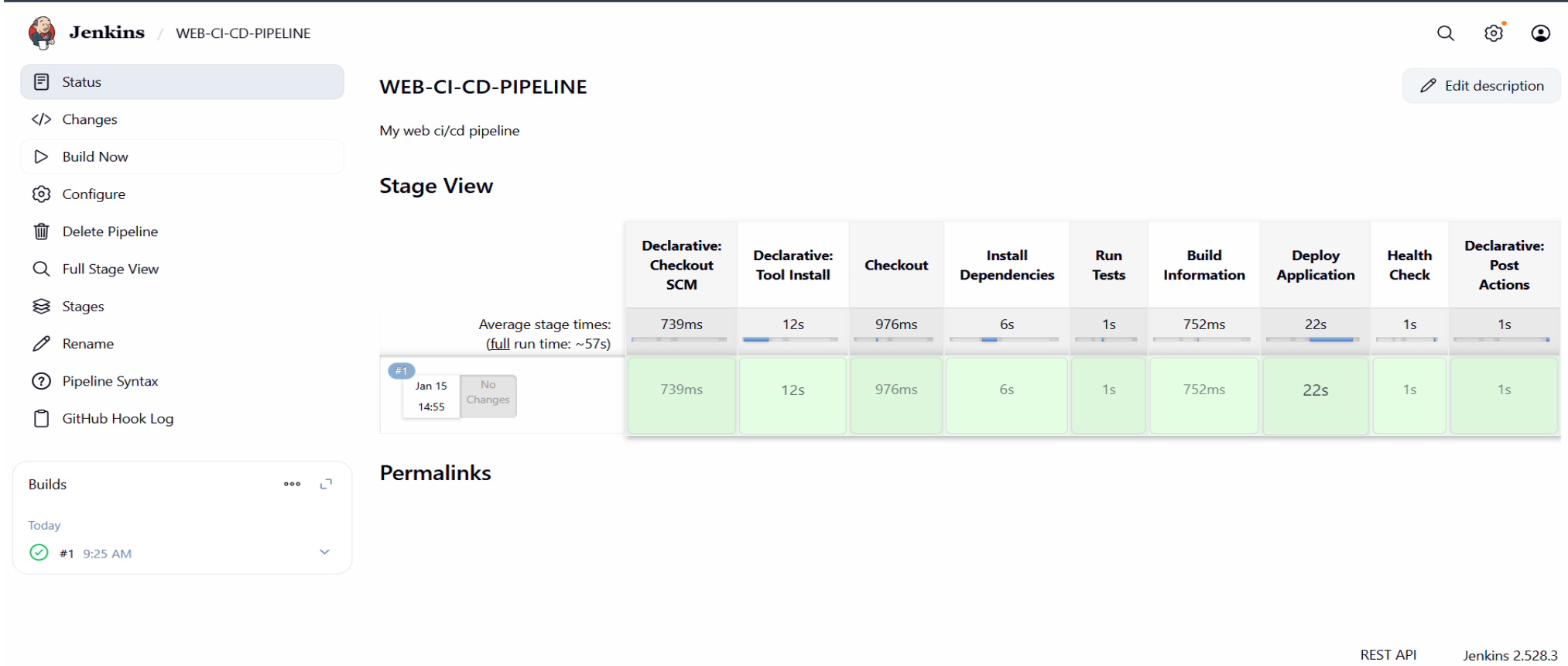
- I configured the WEB-CI-CD-PIPELINE job to trigger automatically using the GitHub hook trigger for GITScm polling.
- This means Jenkins will start a build whenever changes are pushed to my GitHub repository.
- I selected “Pipeline script from SCM” to pull the Jenkinsfile directly from source control.
- The SCM option is currently set to “None” — I will update this to Git and provide the repository URL and credentials.
- Once configured, Jenkins will fetch the pipeline definition and execute the CI/CD stages on every code update.



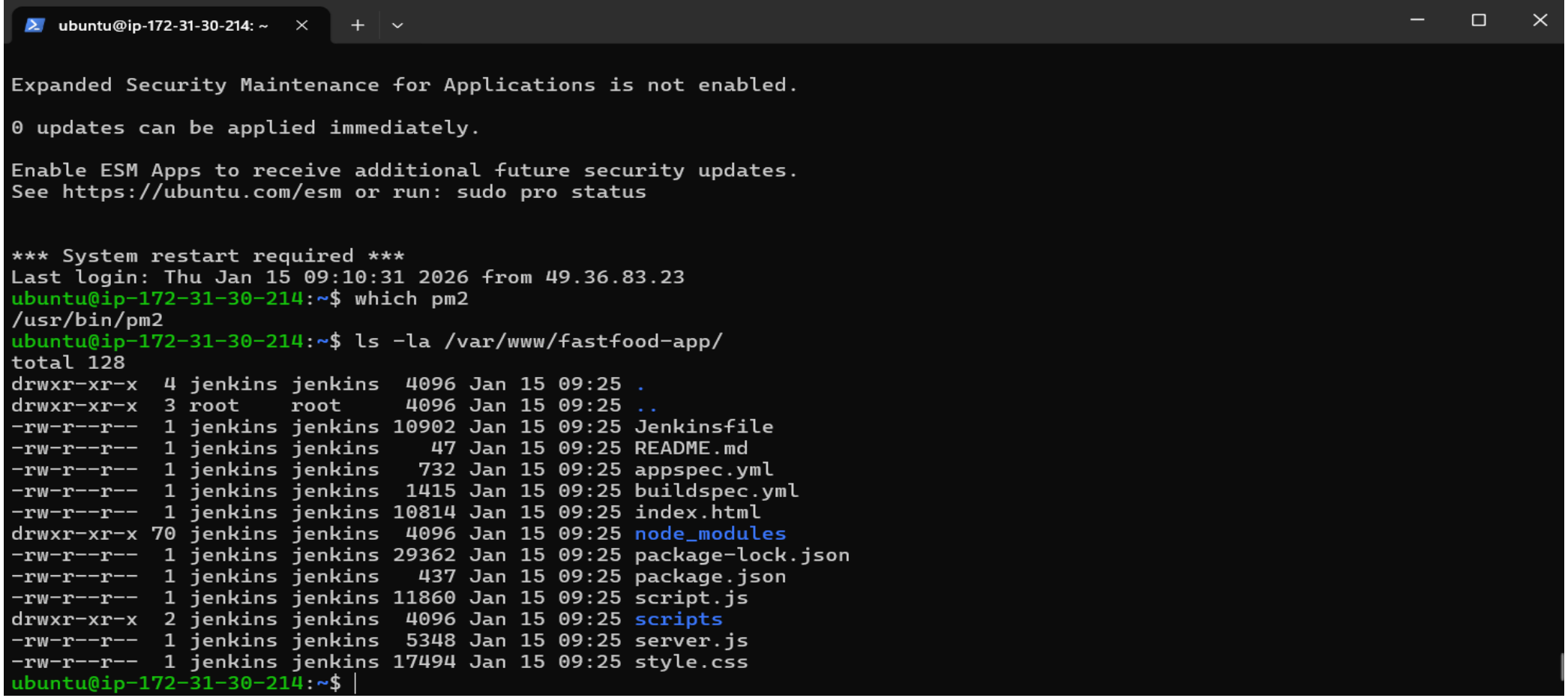
- I added a webhook in my GitHub repository fastfood-cicd--JAY to integrate with Jenkins.



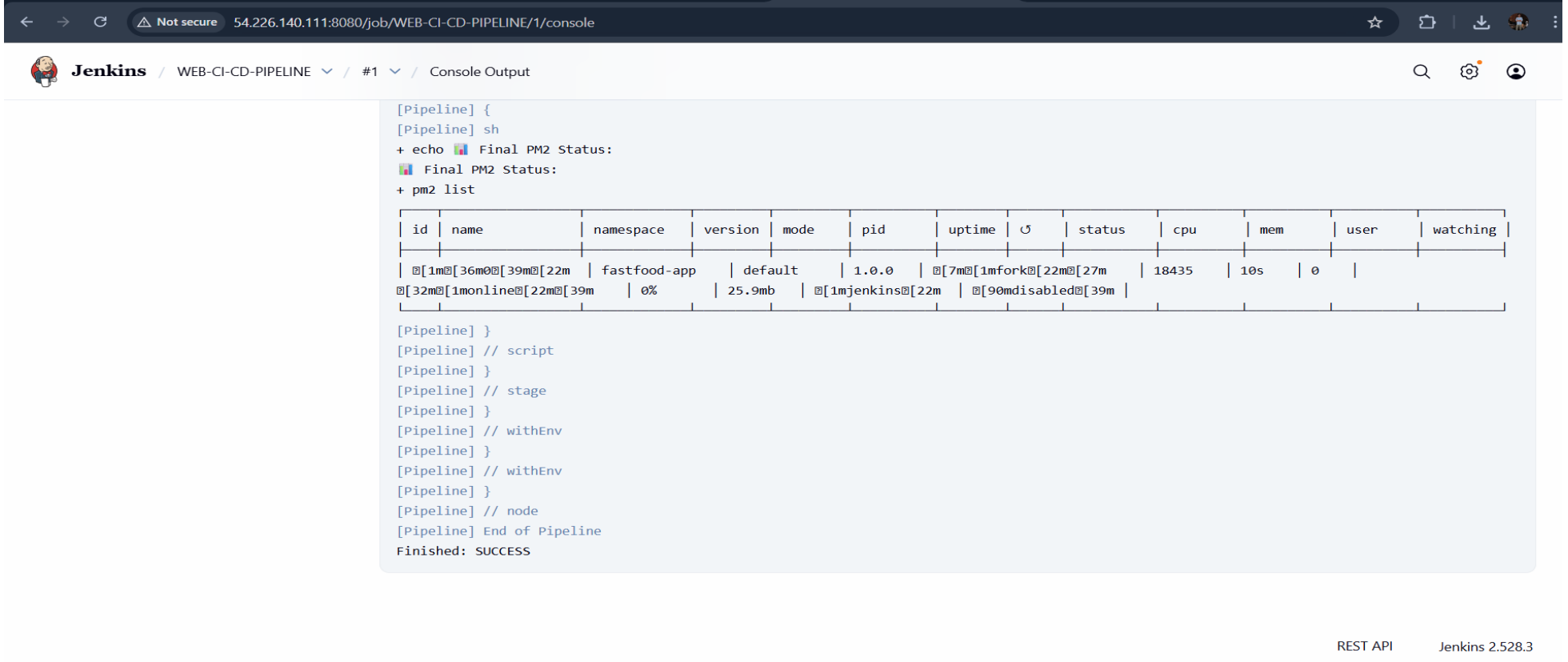
- After build
- I viewed the stage execution summary for the WEB-CI-CD-PIPELINE job in Jenkins.



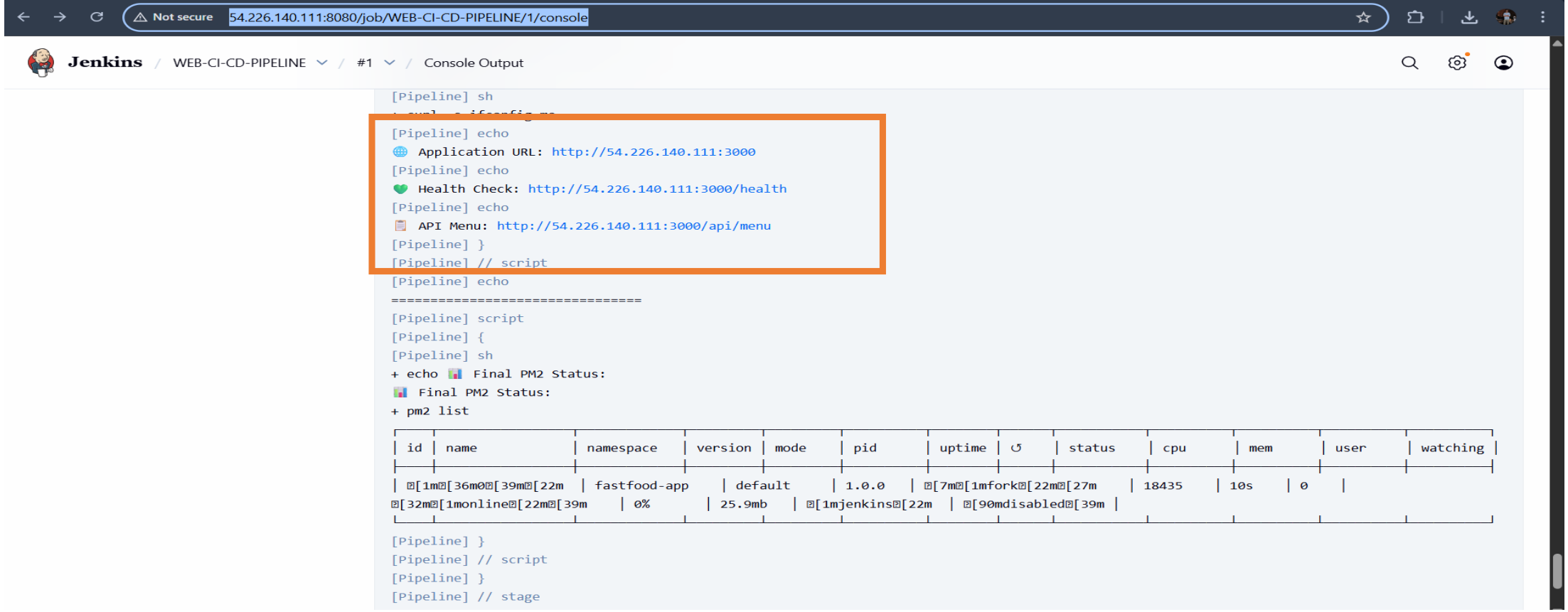
- Build #1 executed today at 9:25 AM completed successfully.
- This confirms that all CI/CD stages are functioning and the deployment pipeline is stable.



ls -la /var/www/ | grep fastfood-app I'm listing the /var/www/ directory with detailed information and filtering for fastfood-app to verify the directory exists and check its permissions and ownership.

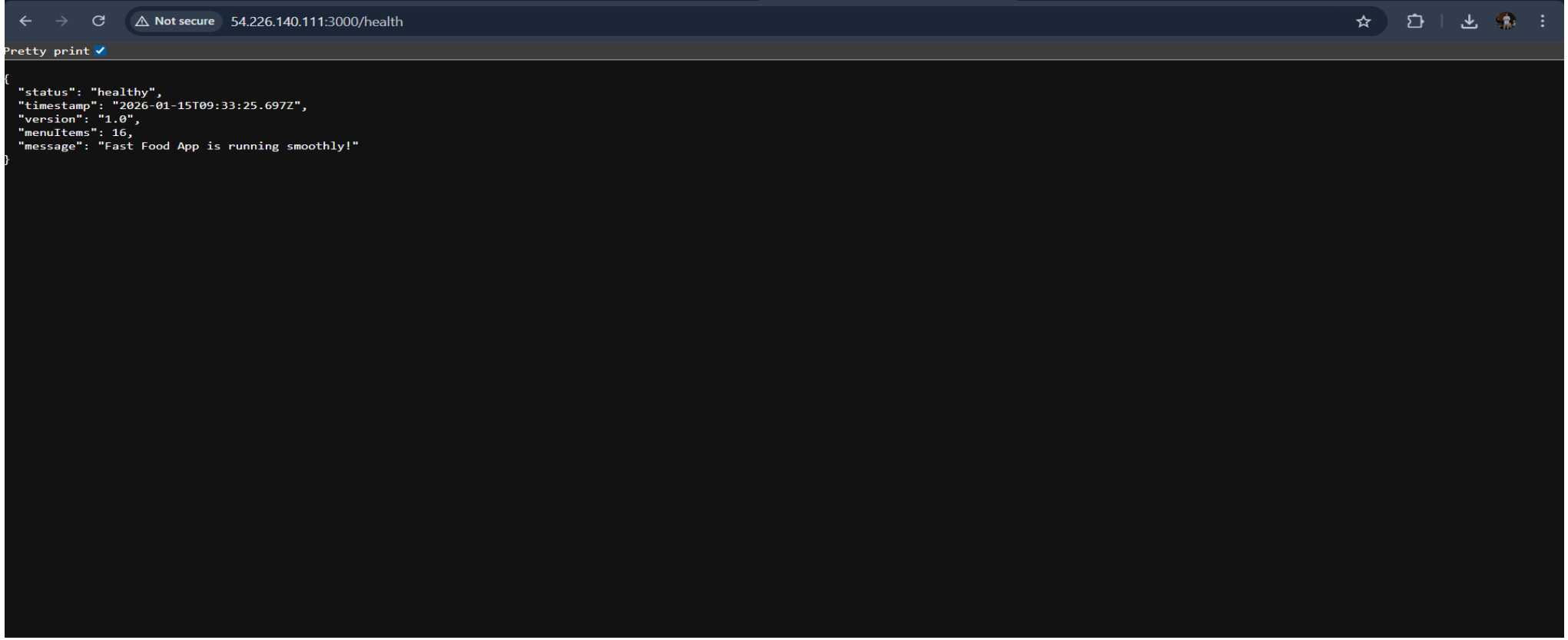


- The pipeline completed successfully, confirming that the application is running and managed correctly by PM2.

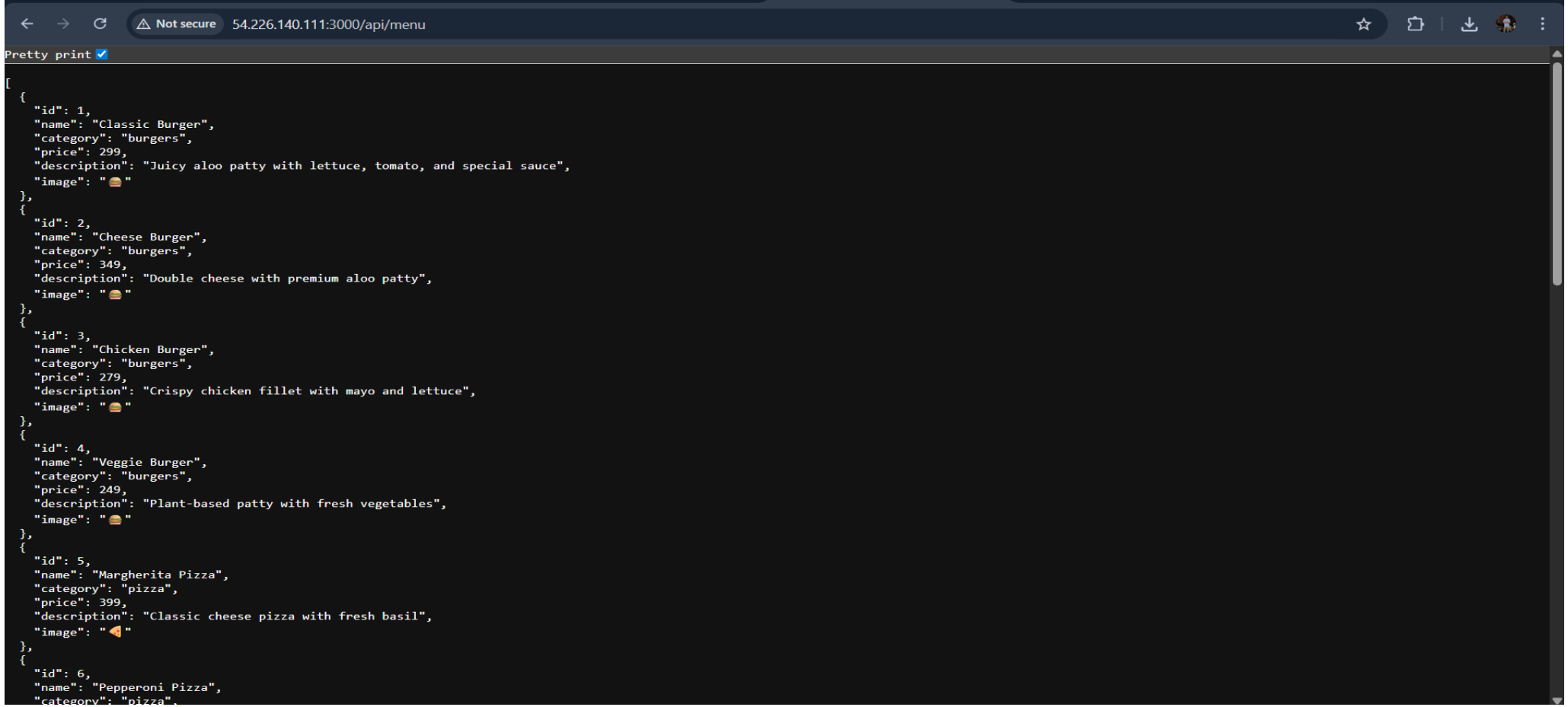


- I used to fetch the public IP address of the deployed EC2 instance.
- Jenkins printed the following deployment endpoints:
- Application URL: (54.226.140.111 in Bing)
- Health Check: (54.226.140.111 in Bing)
- API Menu: (54.226.140.111 in Bing)
- These URLs confirm that the Node.js application is live and accessible over the internet.

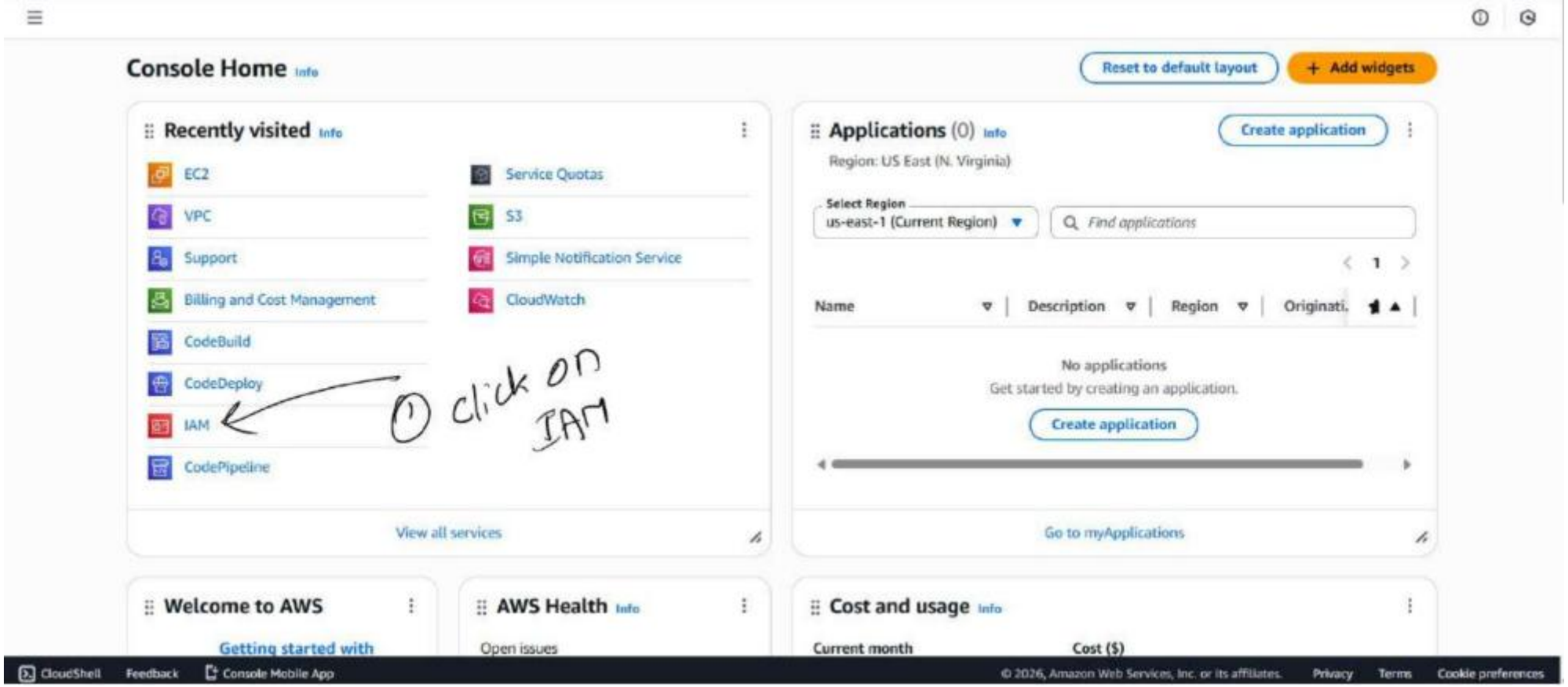
- I accessed the deployed web application at 54.226.140.111:3000 and confirmed the homepage is live.
- This confirms the CI/CD pipeline successfully deployed the Node.js application and it is publicly accessible.



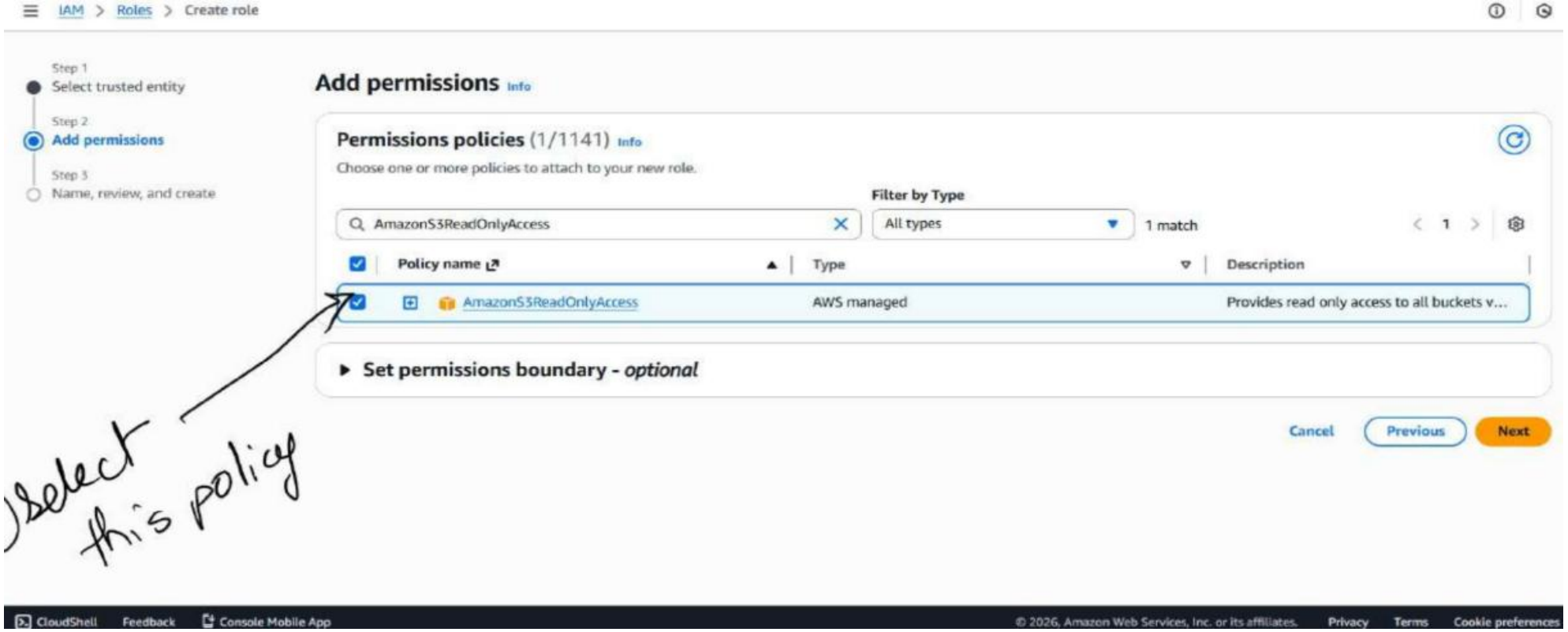
- This confirms the deployment is successful and the backend is responding correctly to health probes.



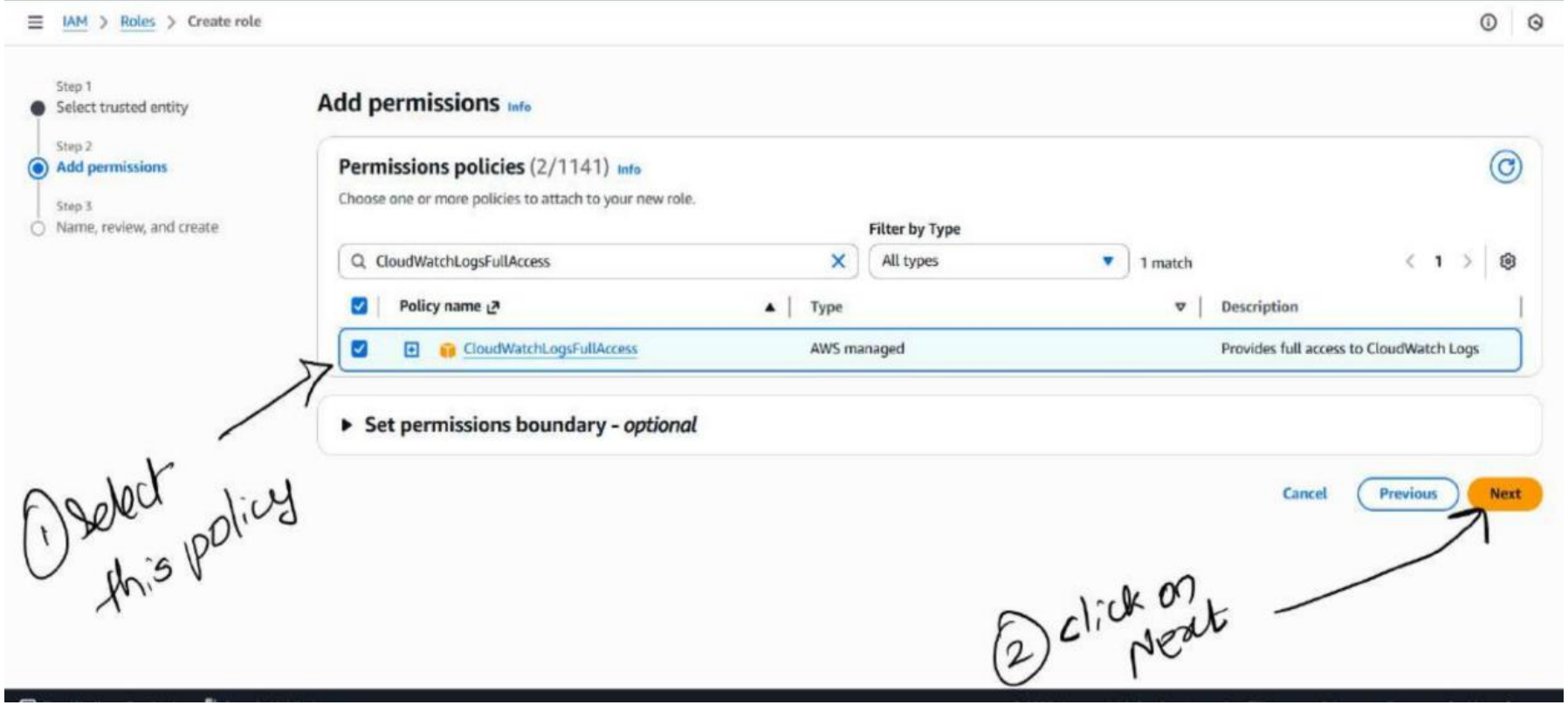
- This confirms the backend API is functioning and delivering structured menu data for frontend rendering and order processing.



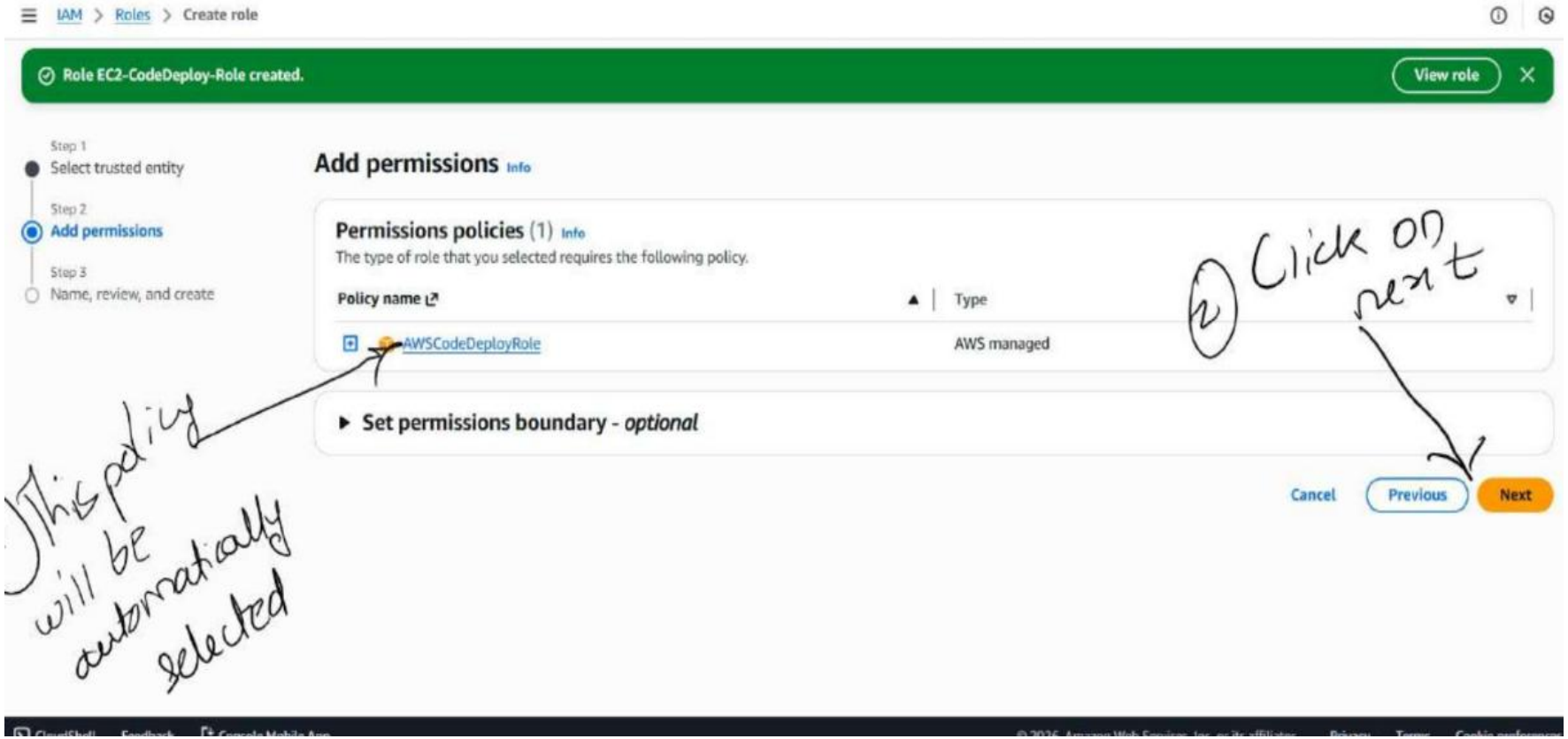
- IAM is highlighted as the first step, since roles and permissions must be configured before setting up CI/CD.



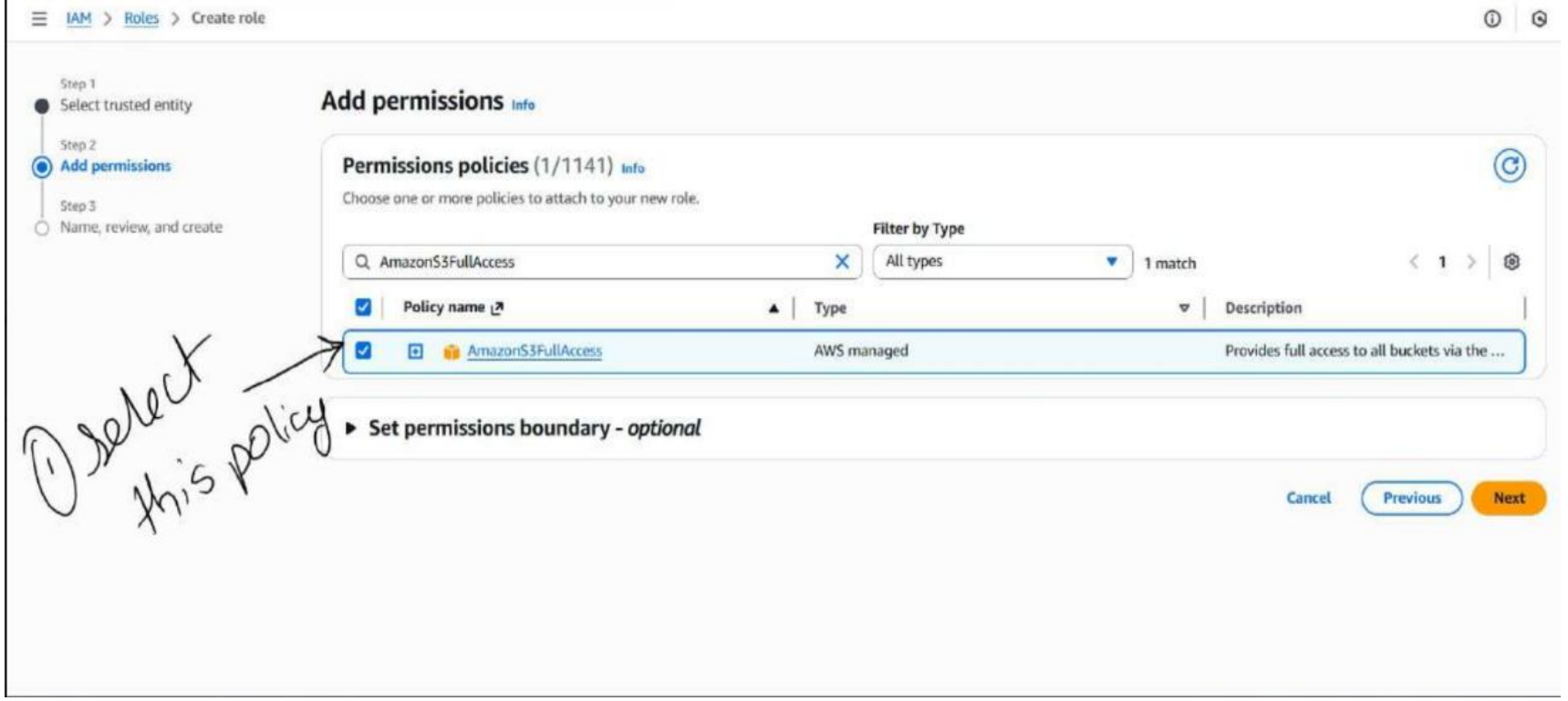
- I selected the AWS managed policy named AmazonS3ReadOnlyAccess.
- This policy provides read-only access to all S3 buckets in the account.
- It’s useful for services like CodeBuild that need to fetch build artifacts or configuration files from S3 without modifying them.



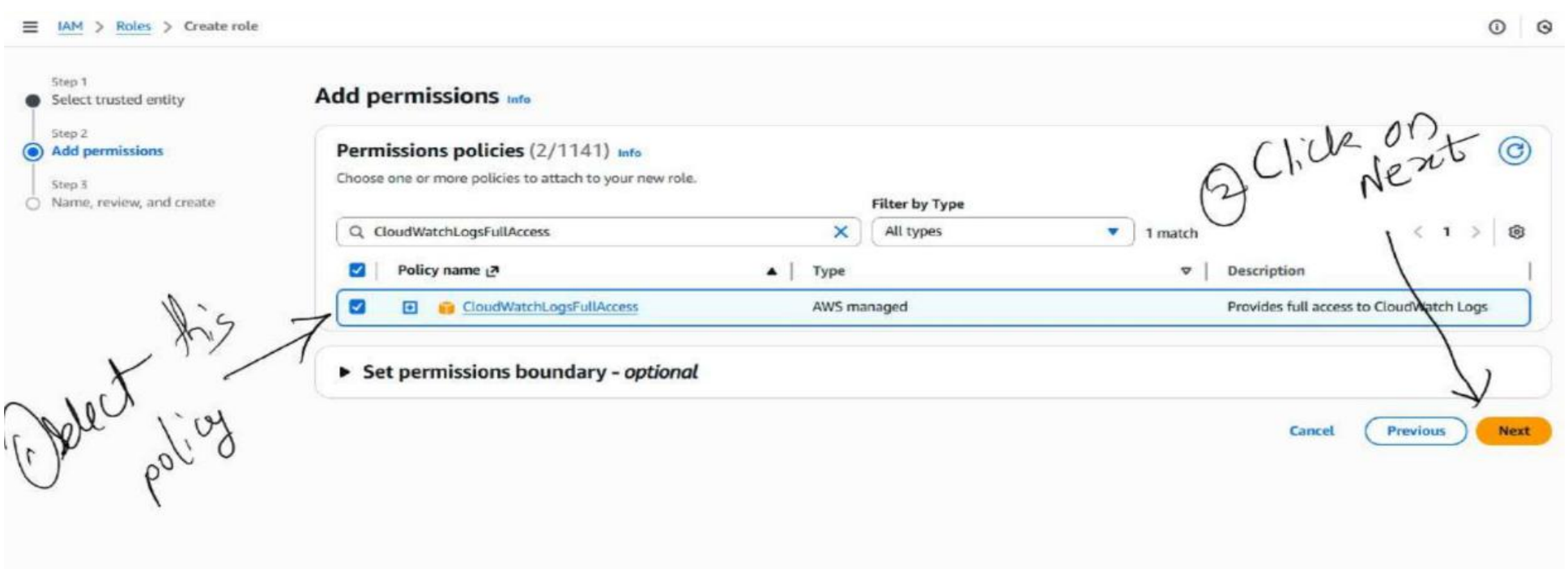
- I selected the AWS managed policy named CloudWatchLogsFullAccess.
- This policy provides full access to CloudWatch Logs, allowing services like CodeBuild to write detailed logs during build execution.
- It’s essential for monitoring build output and debugging failures.



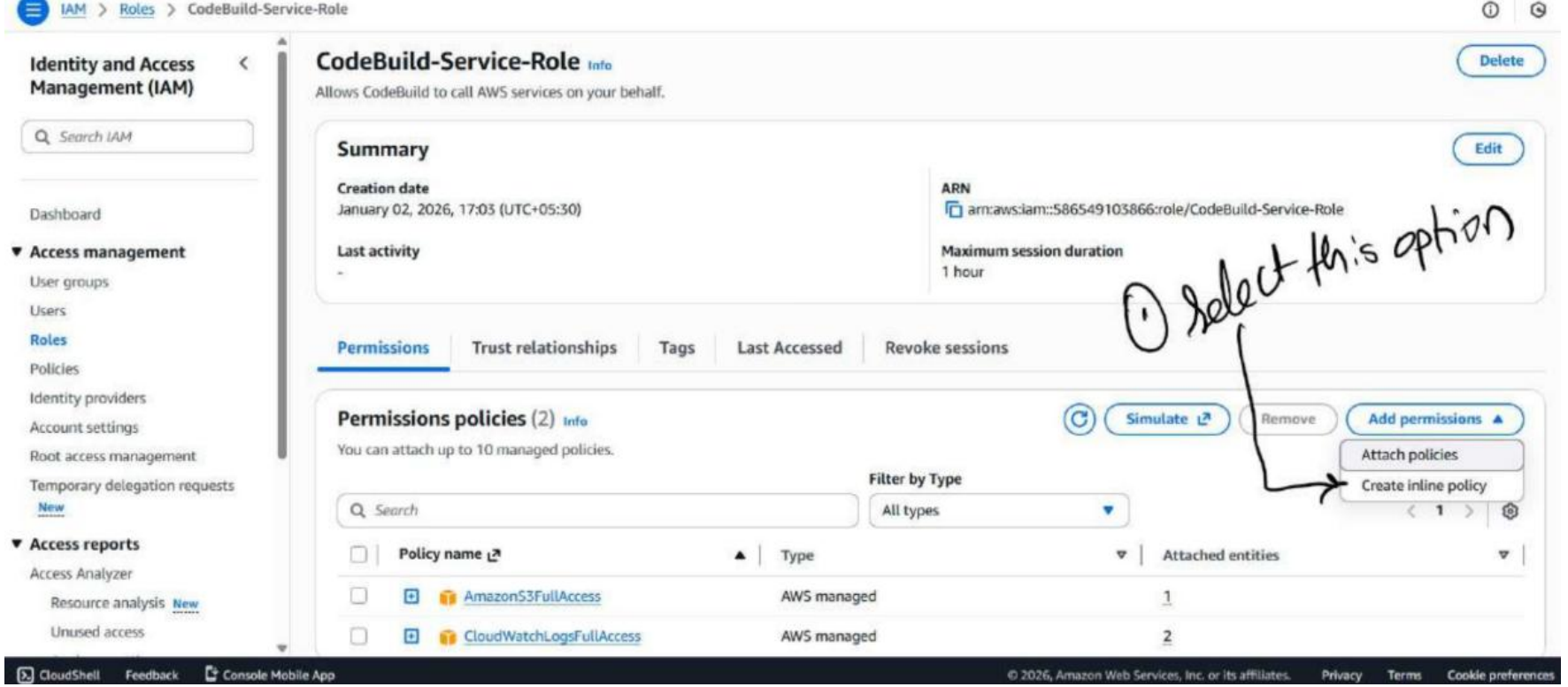
- The AWS managed policy AWSCodeDeployRole was automatically selected.
- This policy grants EC2 permission to work with CodeDeploy during deployments.



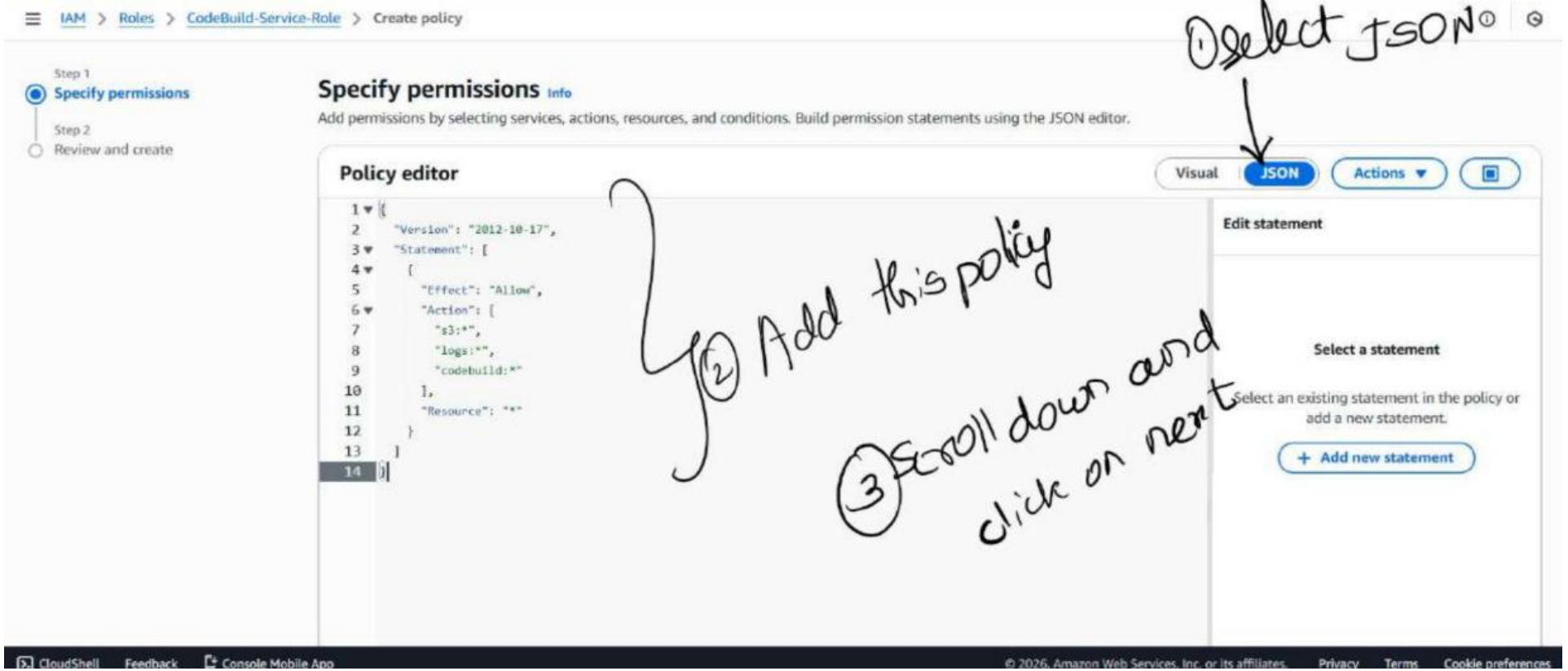
- I selected the AWS managed policy named AmazonS3FullAccess.
- This policy provides full access to all S3 buckets in the account.
- It’s essential for services like CodeBuild and CodeDeploy that need to read and write build artifacts or deployment packages.
- This permission allows uploading, downloading, and managing files in S3 during CI/CD execution.



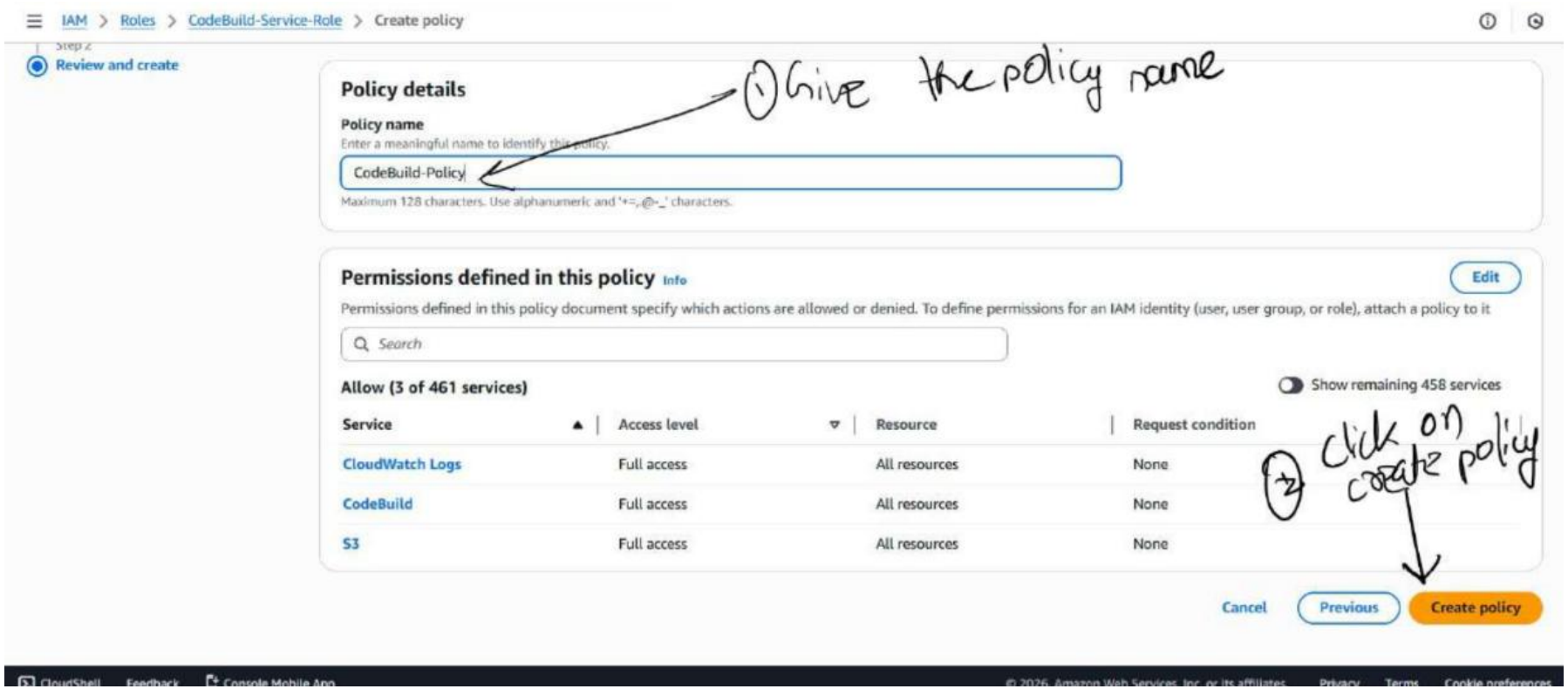
- I selected the AWS managed policy named CloudWatchLogsFullAccess.
- This policy provides full access to CloudWatch Logs, allowing services like CodeBuild to write logs during build execution.
- It’s critical for monitoring build output and troubleshooting errors in the CI/CD pipeline.



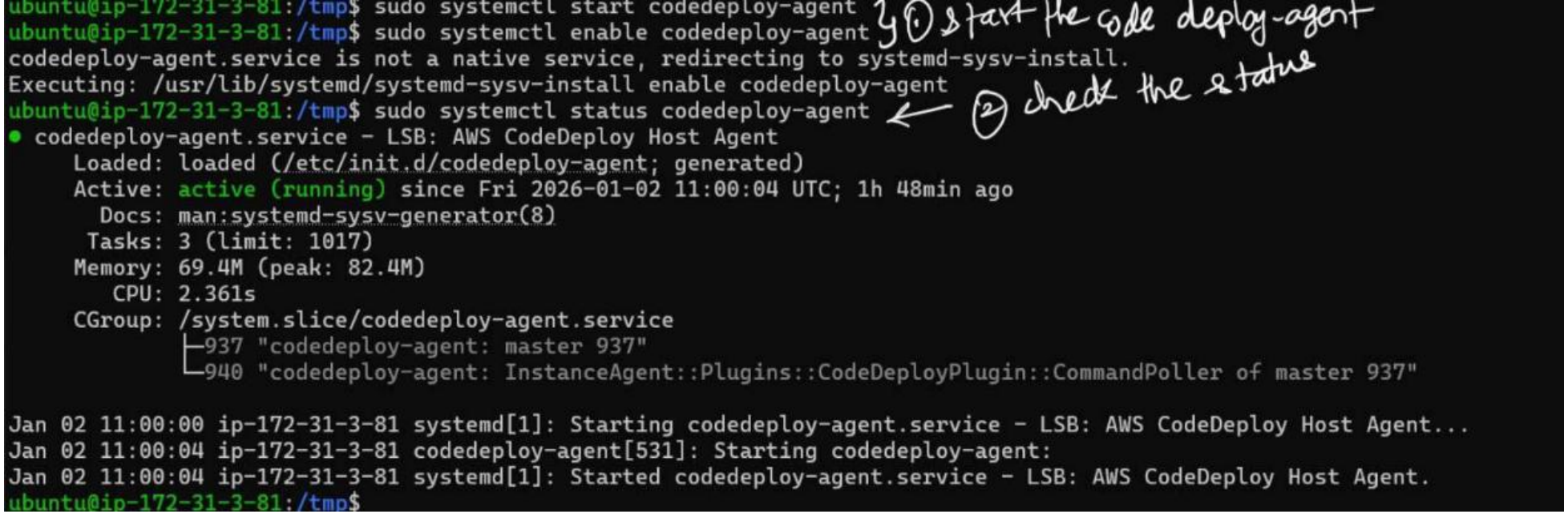
- I’m now ready to add a custom inline policy , using the “Create inline policy” option under Add permissions.



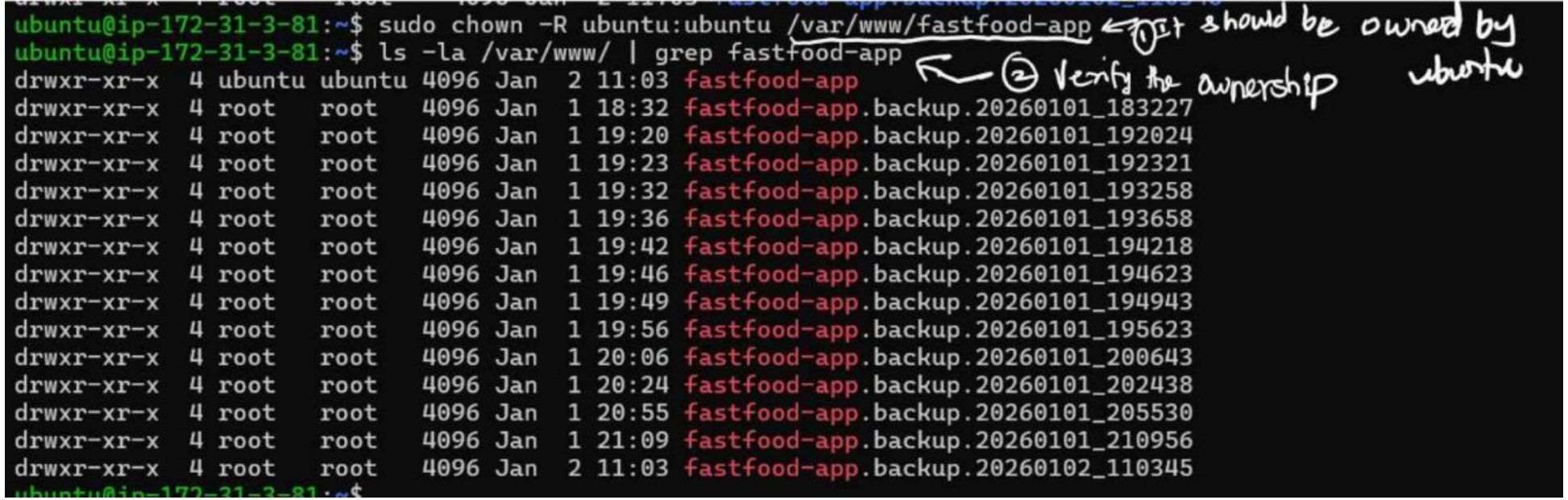
- I created a custom inline policy named CodeBuild-Policy.
- The policy grants full access to three services:
- S3 → for storing and retrieving build artifacts.
- CloudWatch Logs → for logging build output.
- CodeBuild → for managing build projects and execution.
- All permissions apply to all resources in the account.



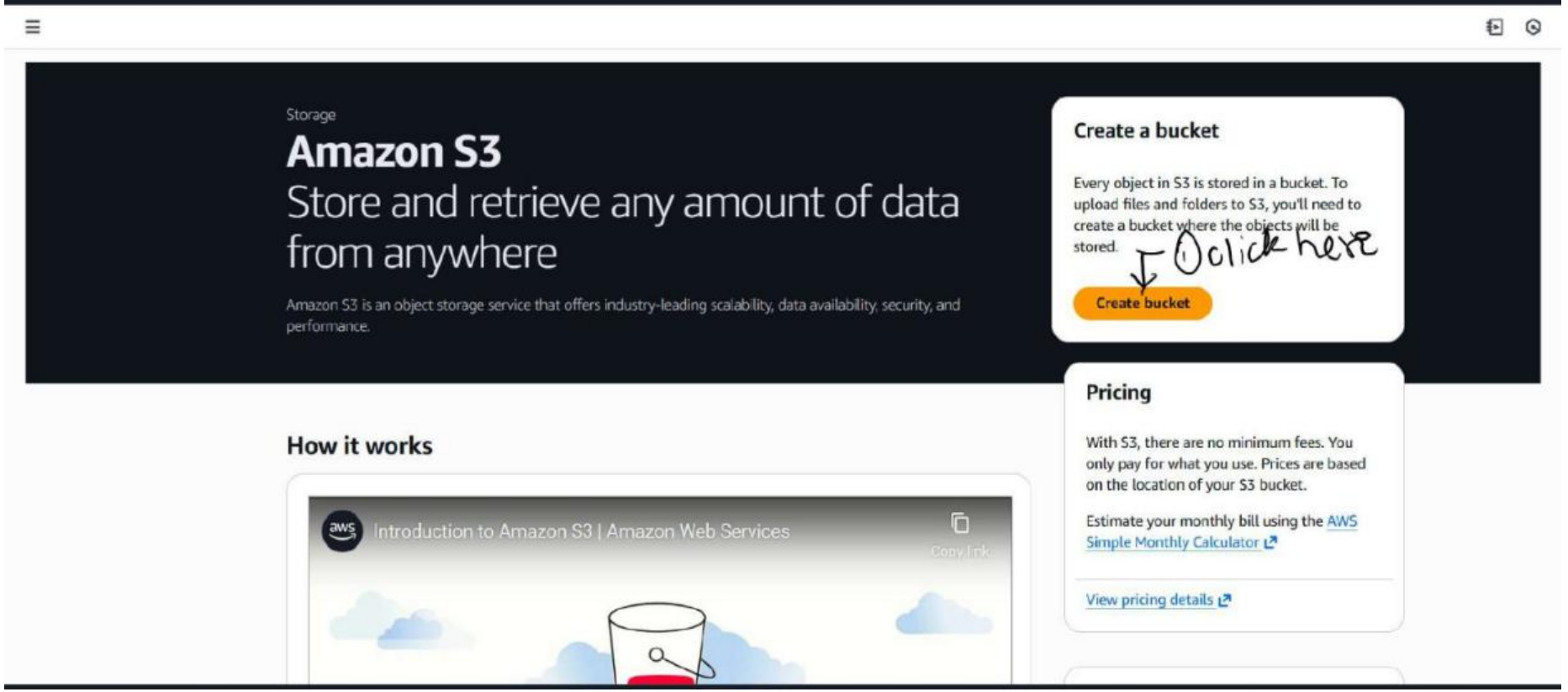
- All permissions apply to all resources in the account.
- No request conditions were defined, keeping the policy broad and flexible.
- I clicked “Create policy” to finalize and attach it to the CodeBuild-Service-Role.



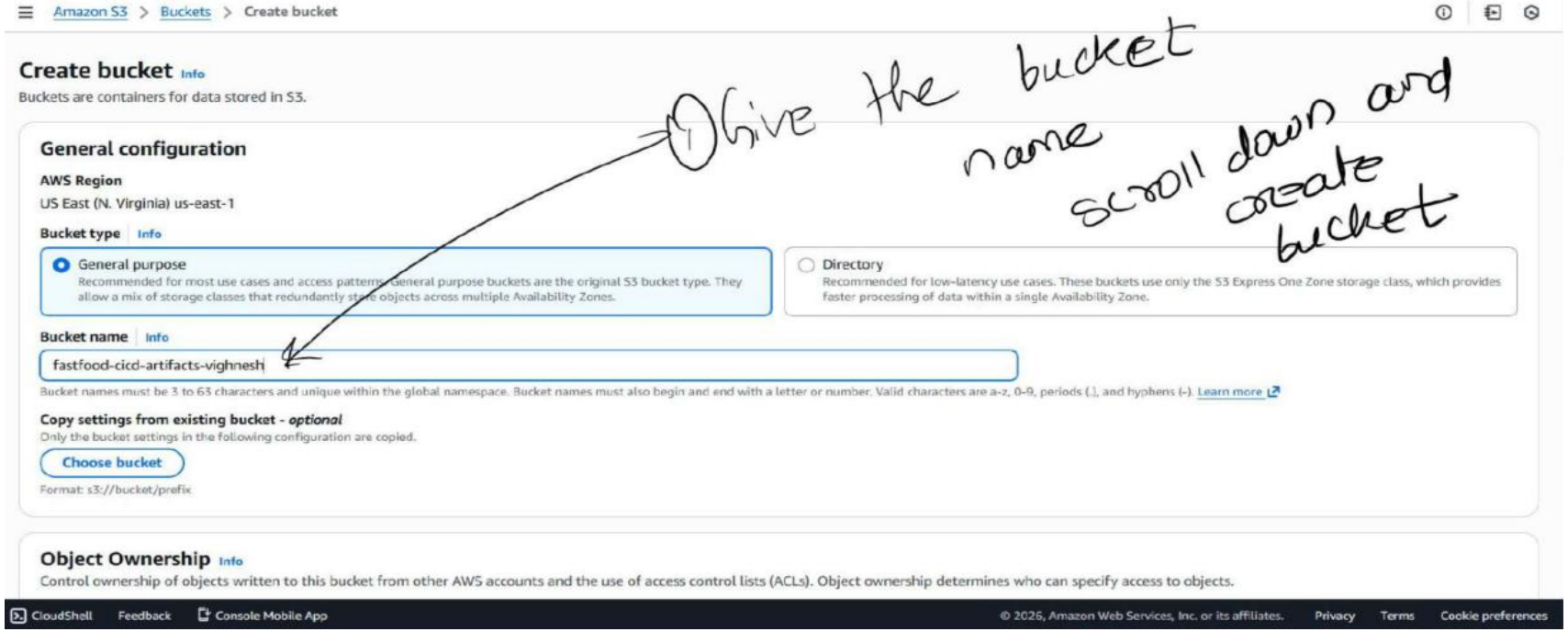
- I enabled the agent to run automatically on boot using systemctl enable.
- Checking the status confirmed the agent is active and running, with processes for the master and the command poller.



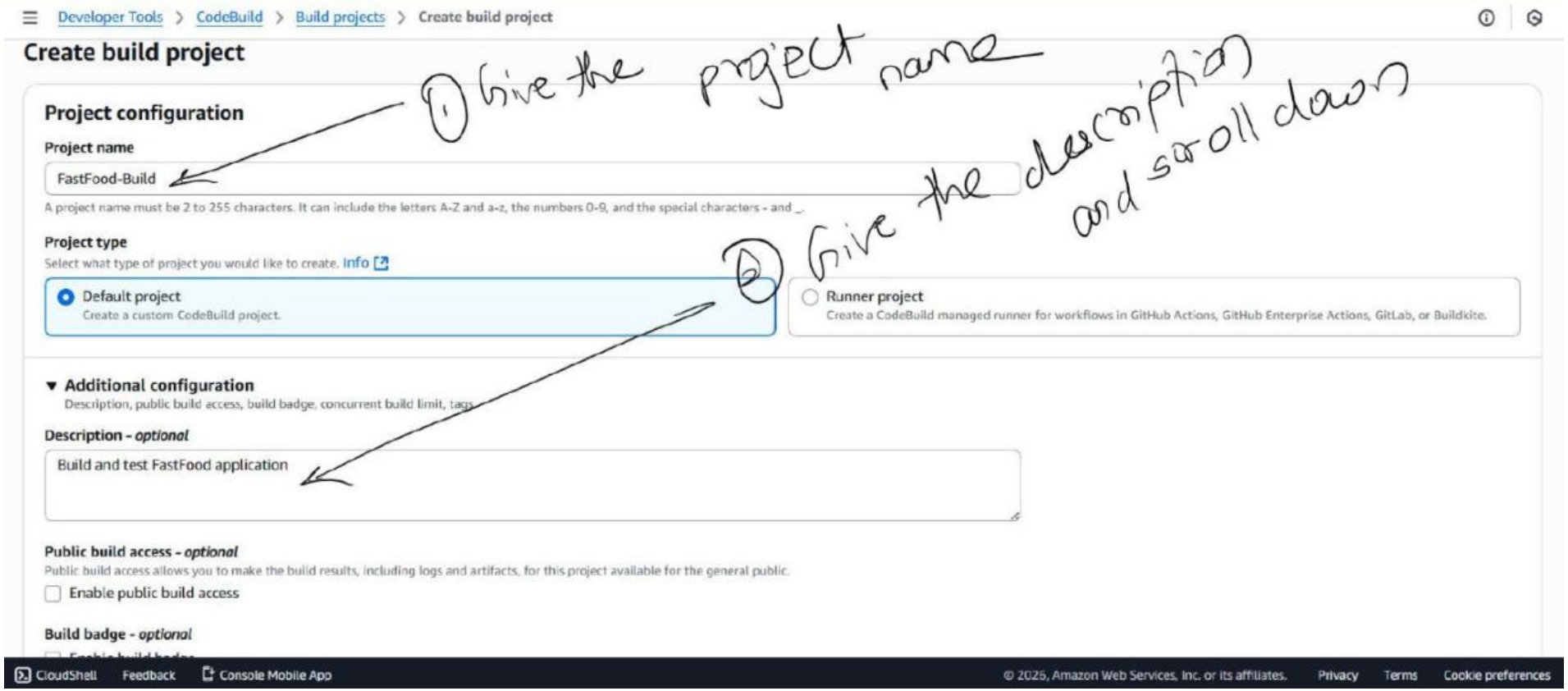
- I changed ownership of the /var/www/fastfood-app directory to ubuntu:ubuntu using chown.
- This ensures the ubuntu user has full control over the live application directory.
- I verified with ls -la and confirmed fastfood-app is owned by ubuntu.



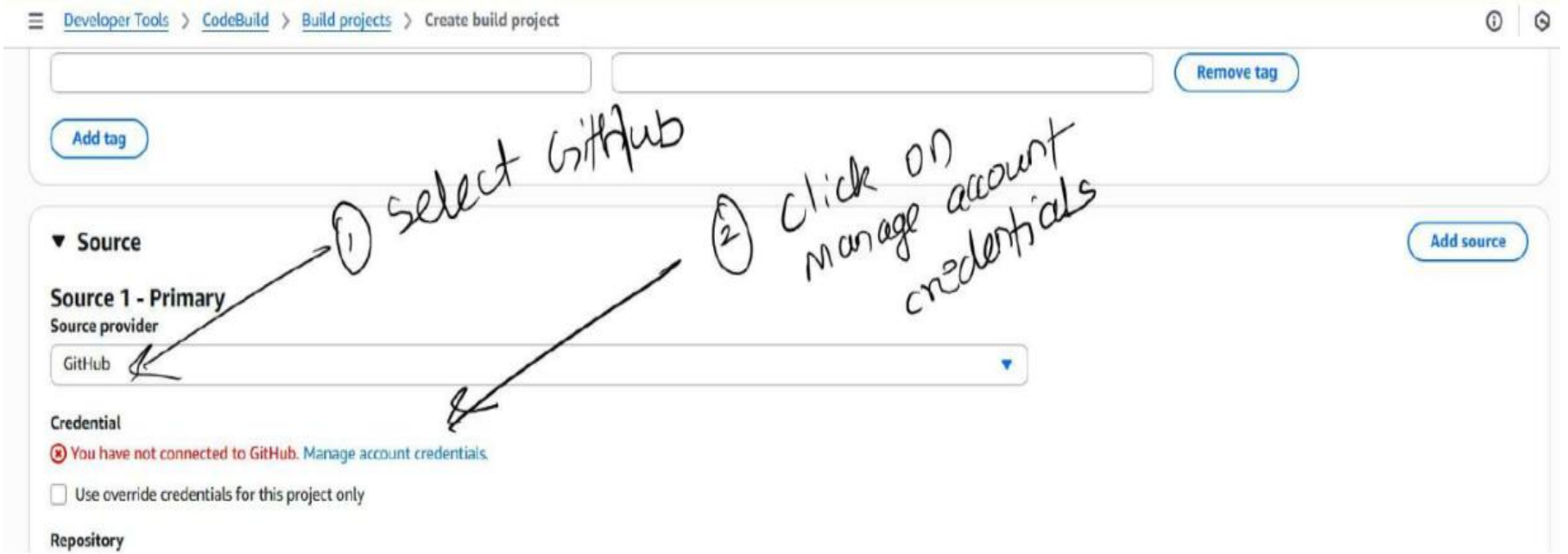
- S3 offers scalable object storage with high availability, security, and performance.
- To begin, I clicked on Create bucket



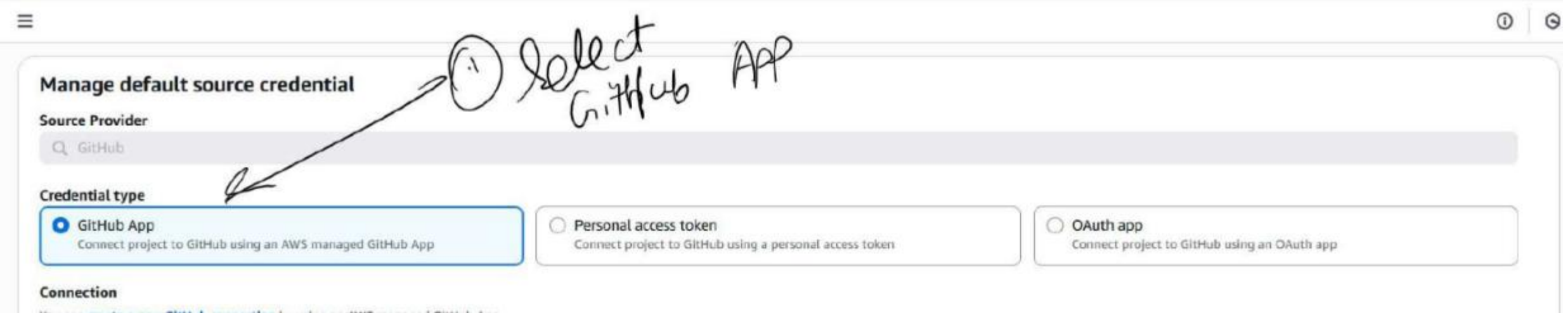
- Bucket type: General purpose, suitable for storing build and deployment artifacts.
- Bucket name entered.



- I opened the AWS CodeBuild console to create a new build project.
- I entered the project name as FastFood-Build.
- This name uniquely identifies the build process for the FastFood application.
- In the description field, I wrote “Build and test FastFood application” to clarify the project’s purpose.

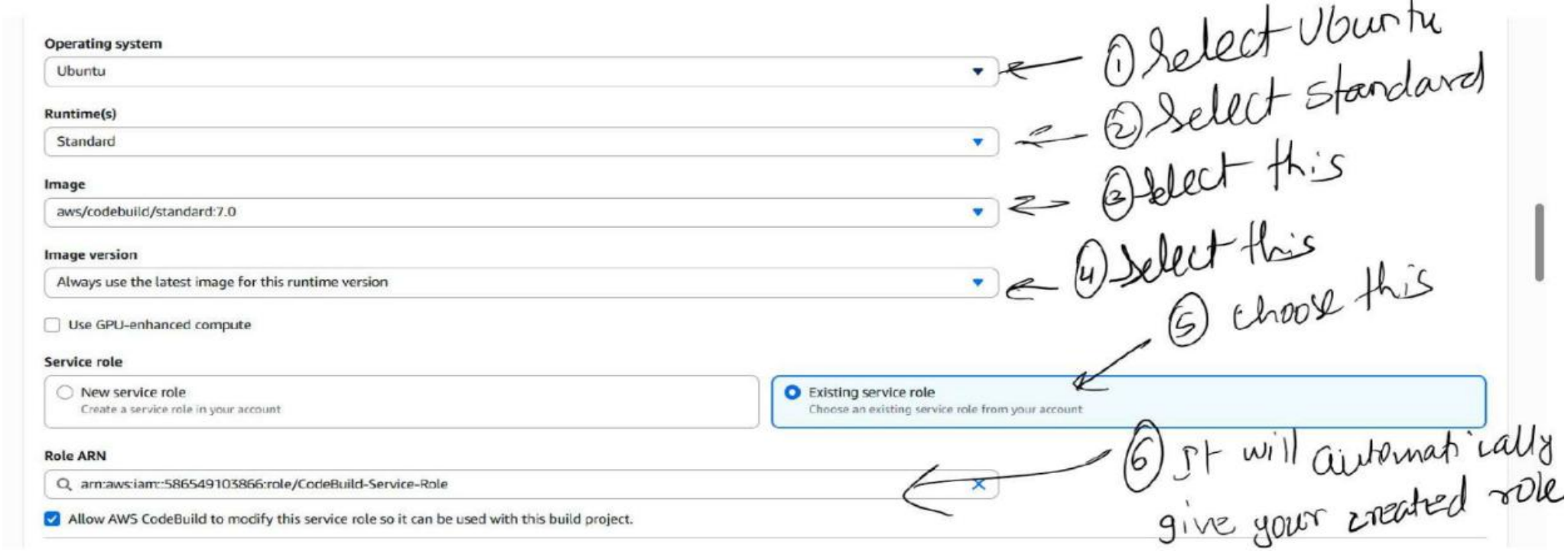


- I selected GitHub as the source provider for the build project.
- This allows CodeBuild to pull source code directly from a GitHub repository.
- Since the account was not yet connected, a warning appeared stating that GitHub credentials were missing.
- I clicked on “Manage account credentials” to begin the GitHub authorization process.

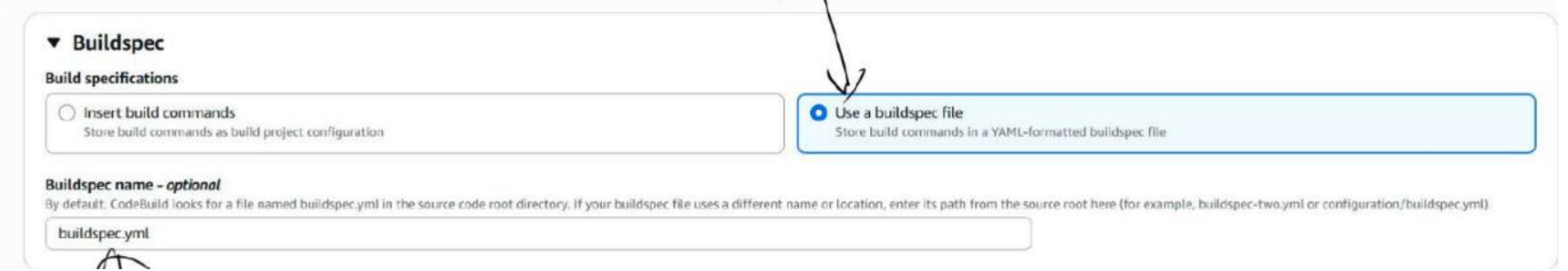


- I selected GitHub as the source provider.
- Under credential type, I chose “GitHub App” to connect using AWS’s managed GitHub integration.
- This method simplifies authentication and allows CodeBuild to securely access the repository.
- It avoids manual token management and supports automated builds triggered by GitHub events.
- This step completes the GitHub connection setup for the build project.

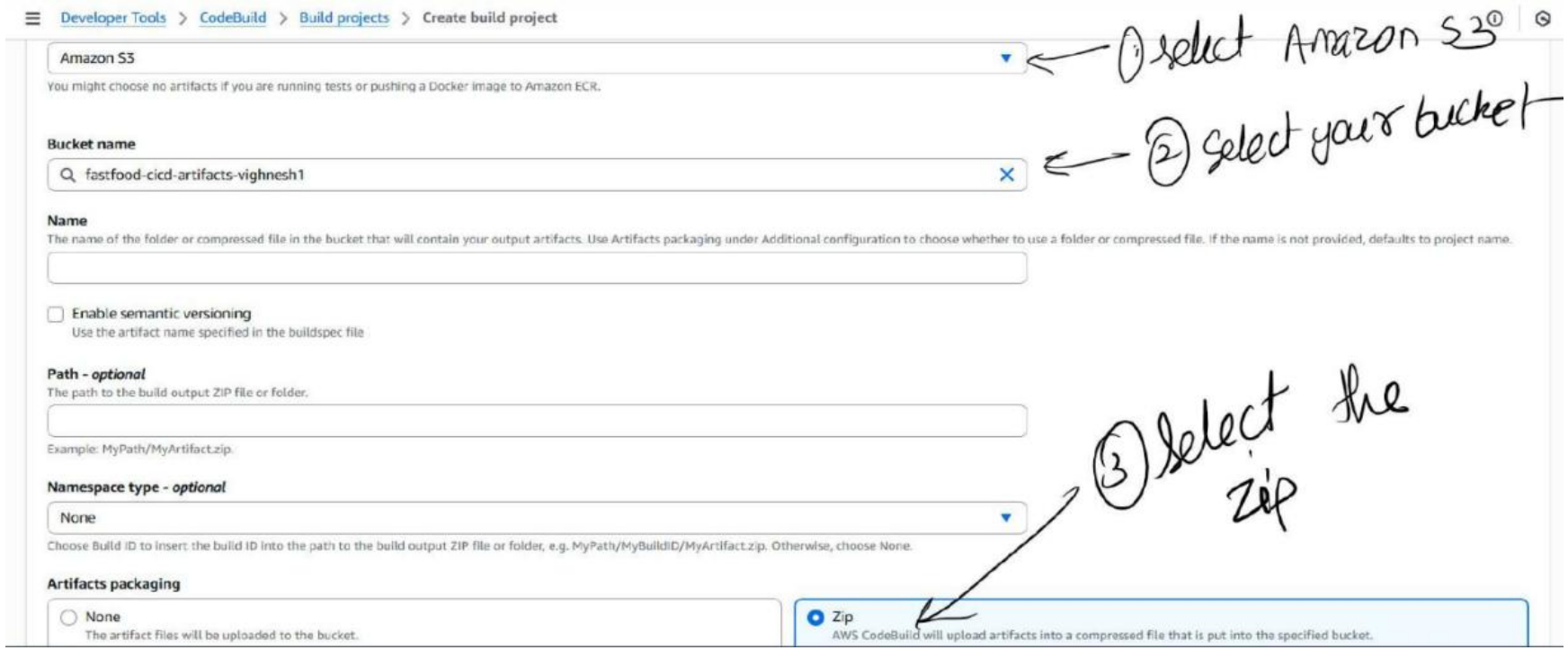
- I entered the GitHub repository URL (github.com in Bing).
- This connects the CodeBuild project to the source code for the FastFood application.
- The repository field confirms the link to the correct GitHub project.



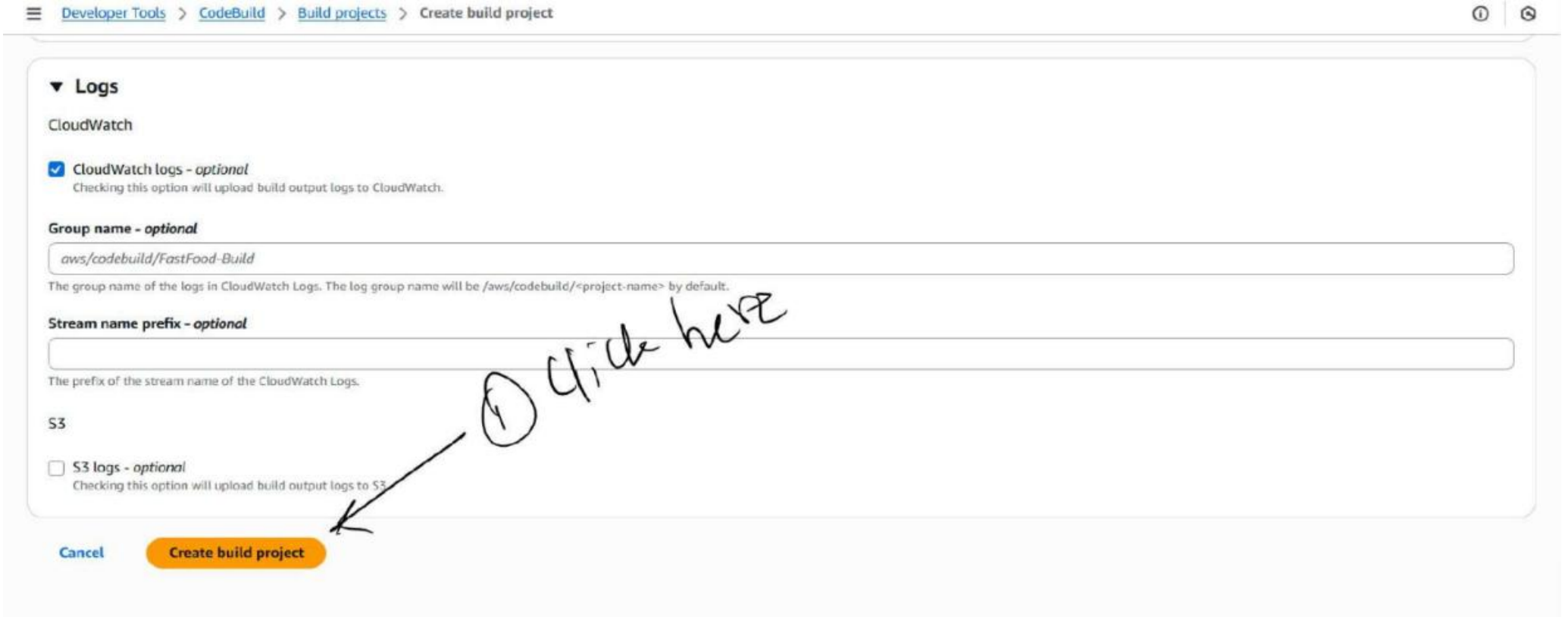
- I selected Ubuntu as the operating system for the build environment.
- I chose Standard as the runtime, which supports a wide range of build tools and languages.
- For the image, I selected , which includes pre-installed runtimes like Python, Node.js, and Java.
- I set the image version to “Always use the latest image for this runtime version” to ensure the build uses updated tools.
- I did not enable GPU-enhanced compute since it's not needed for this application.
- For the service role, I selected “Existing service role” and chose from the account.
- I allowed CodeBuild to modify the service role so it can be used with this build project.
- This completes the environment setup for the build process.



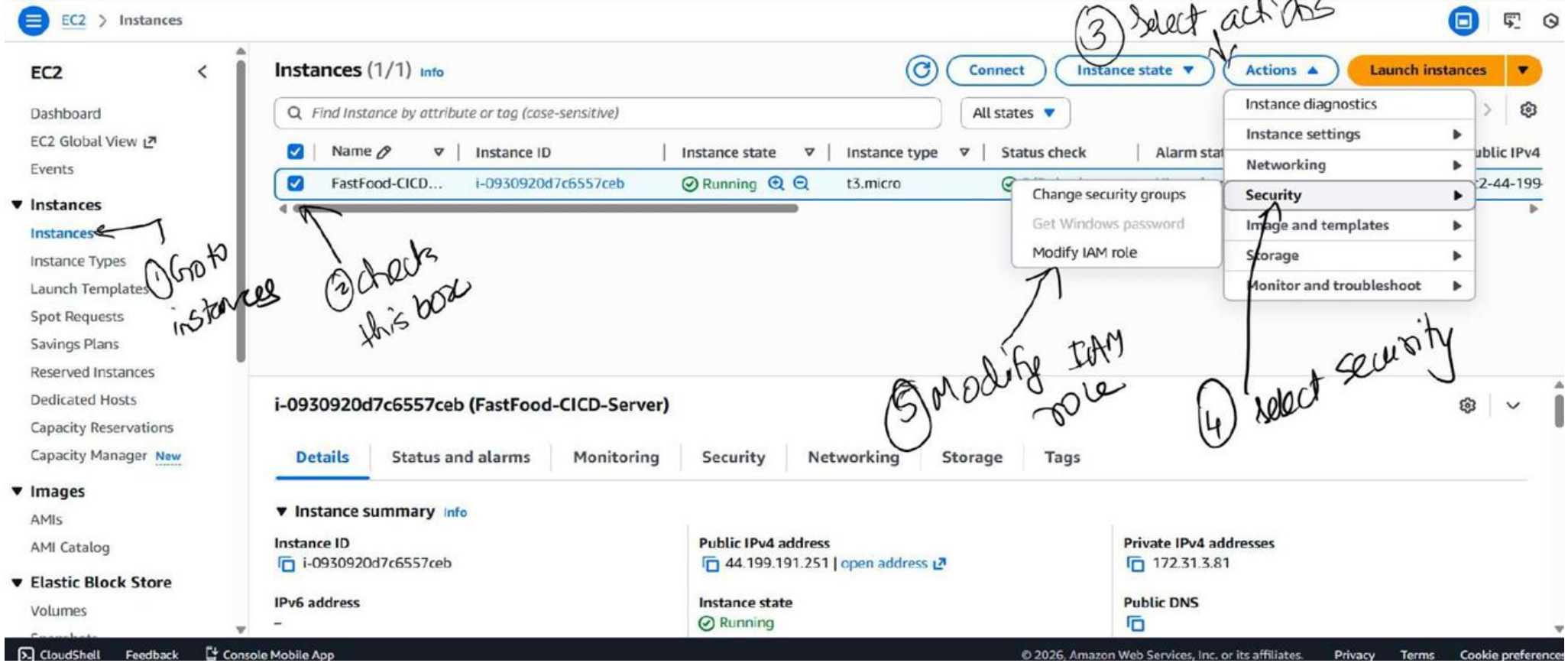
- I selected “Use a buildspec file” to define build commands in YAML format.
- This allows the build instructions to be stored inside the source code repository.
- I entered the filename which is the default name CodeBuild looks for in the root directory.



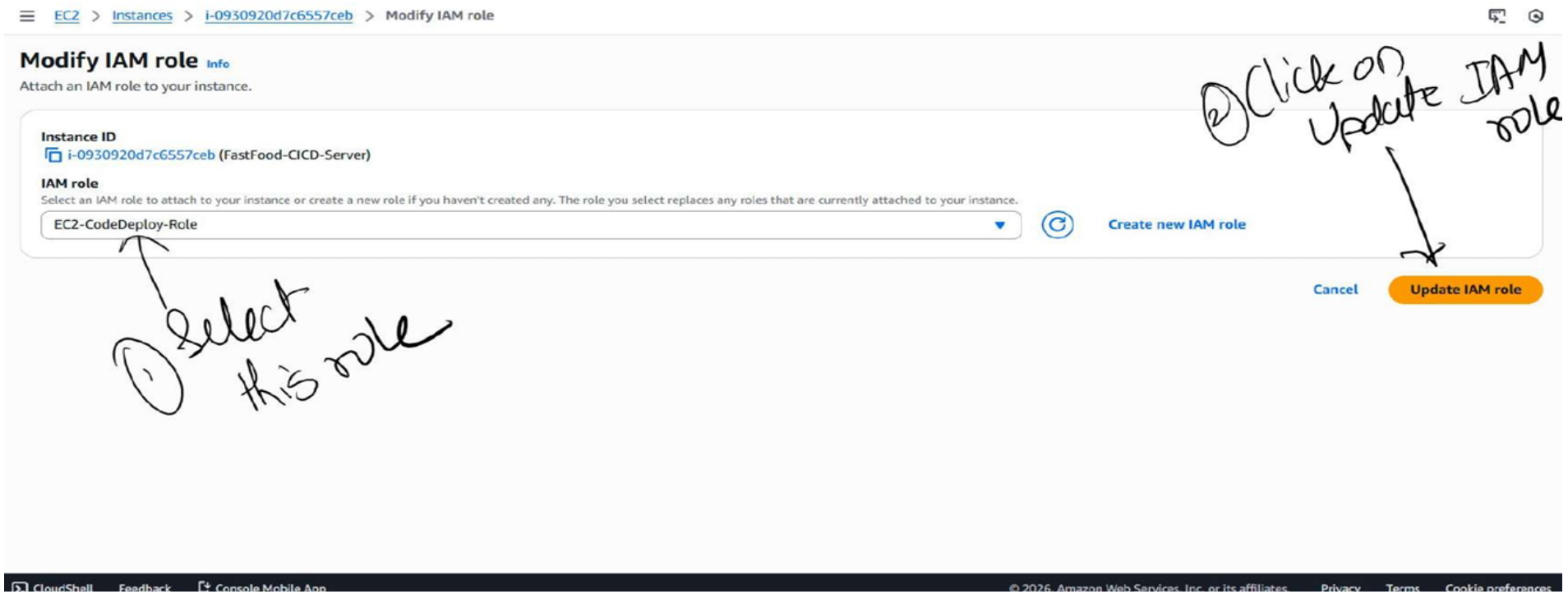
- I selected “Use a buildspec file” to define build commands in YAML format.
- This allows CodeBuild to read instructions directly from the source repository.
- I entered the file name as which is the default name CodeBuild looks for in the root directory.
- This file will contain phases like install, build, and post_build, along with artifact definitions.
- Using a buildspec file keeps the build logic version-controlled and easy to update.



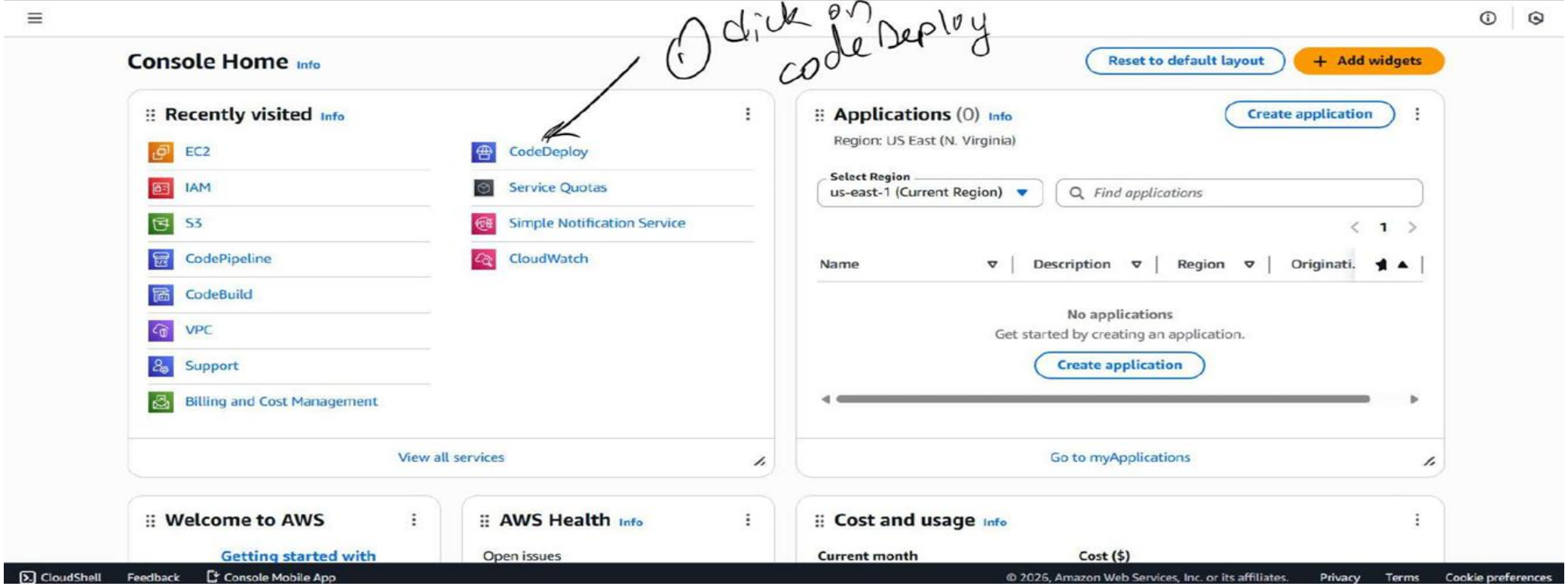
- I selected Amazon S3 as the destination for storing build artifacts.
- This allows the output of the build process to be saved in a centralized, versioned location.
- I chose the bucket named which was created earlier for this purpose.
- For packaging, I selected the “Zip” option so that all build output files are compressed into a single archive.
- This makes it easier to transfer and deploy the artifacts using CodeDeploy.



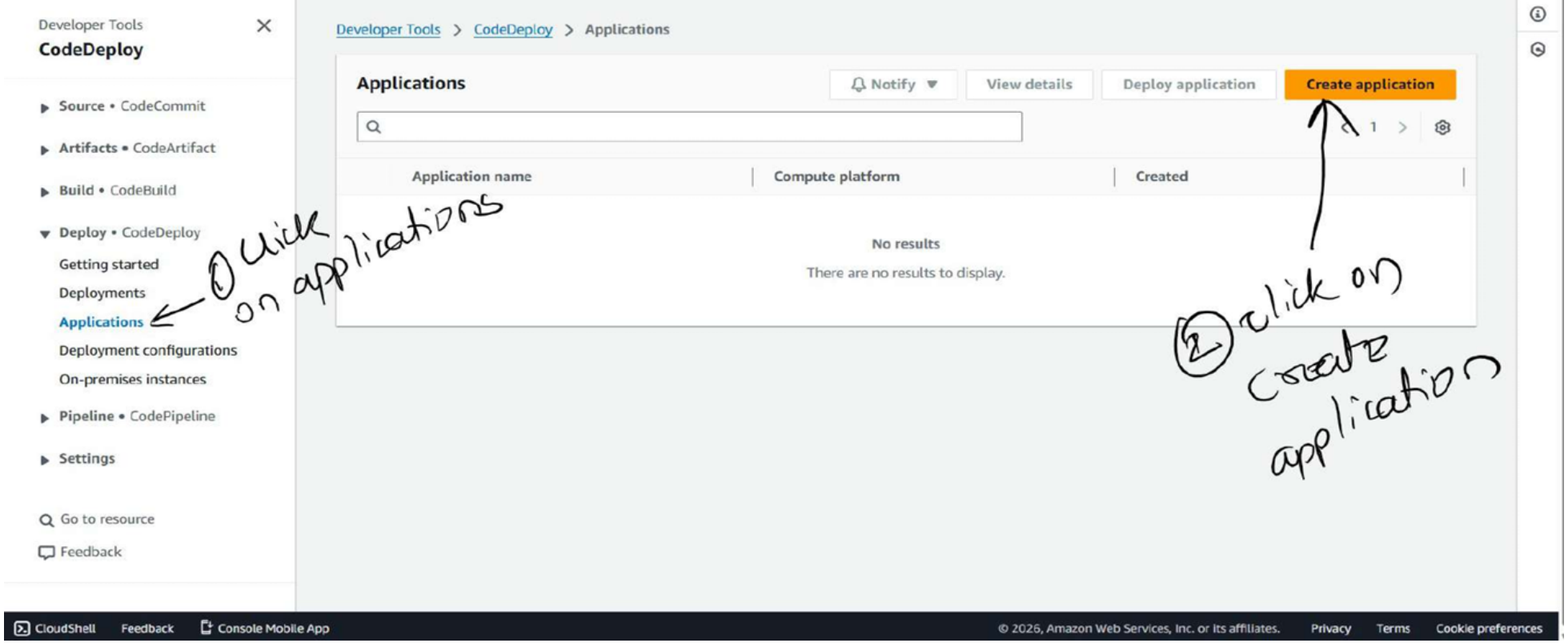
- I opened the EC2 console and navigated to the Instances section.
- From the Actions dropdown, I hovered over Security.
- I chose Modify IAM role to attach the correct permissions for CodeDeploy and CodeBuild access.



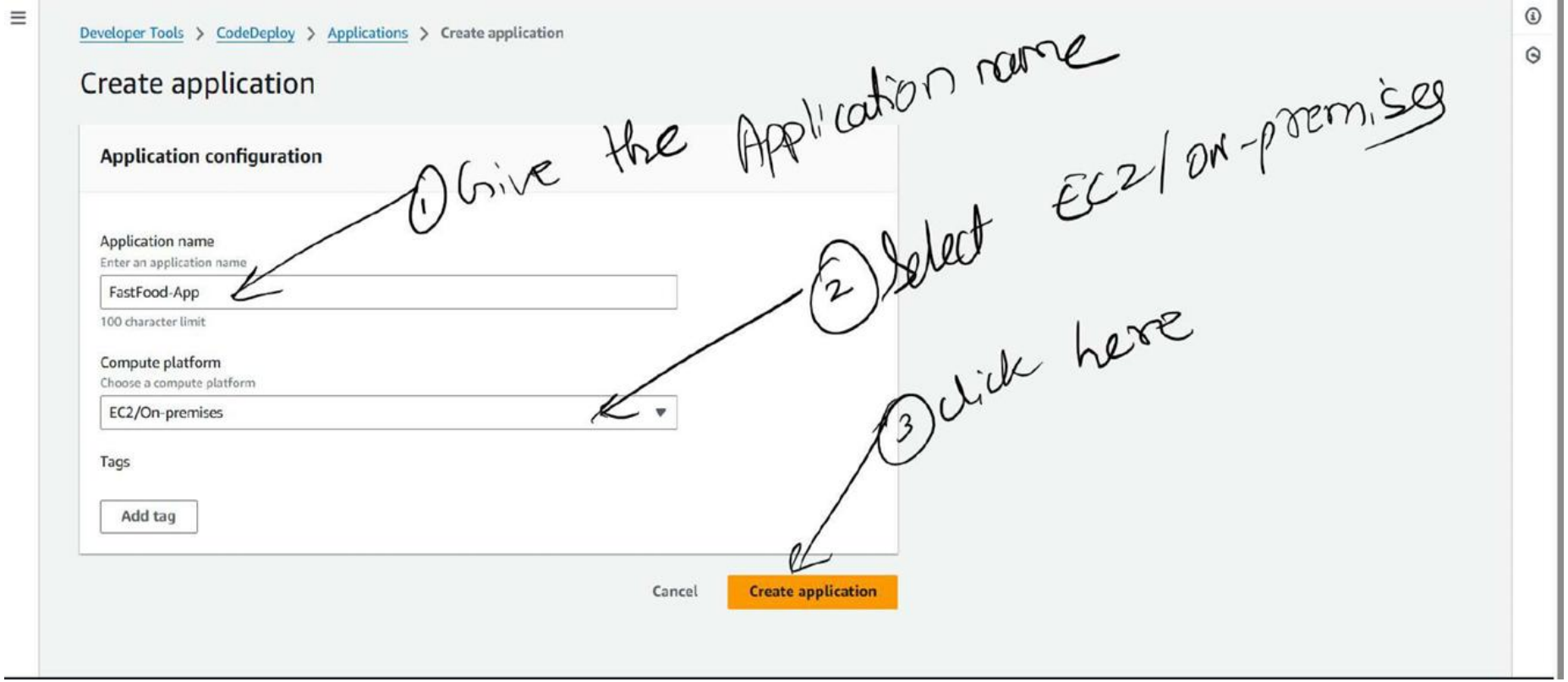
- I opened the Modify IAM role section for the EC2 instance named FastFood-CICD-Server.
- I selected the IAM role named EC2-CodeDeploy-Role from the dropdown.
- This role includes permissions for CodeDeploy to interact with the EC2 instance during deployments.
- I clicked on Update IAM role to attach it to the instance.
- This step ensures the EC2 instance can receive and execute deployment instructions securely.



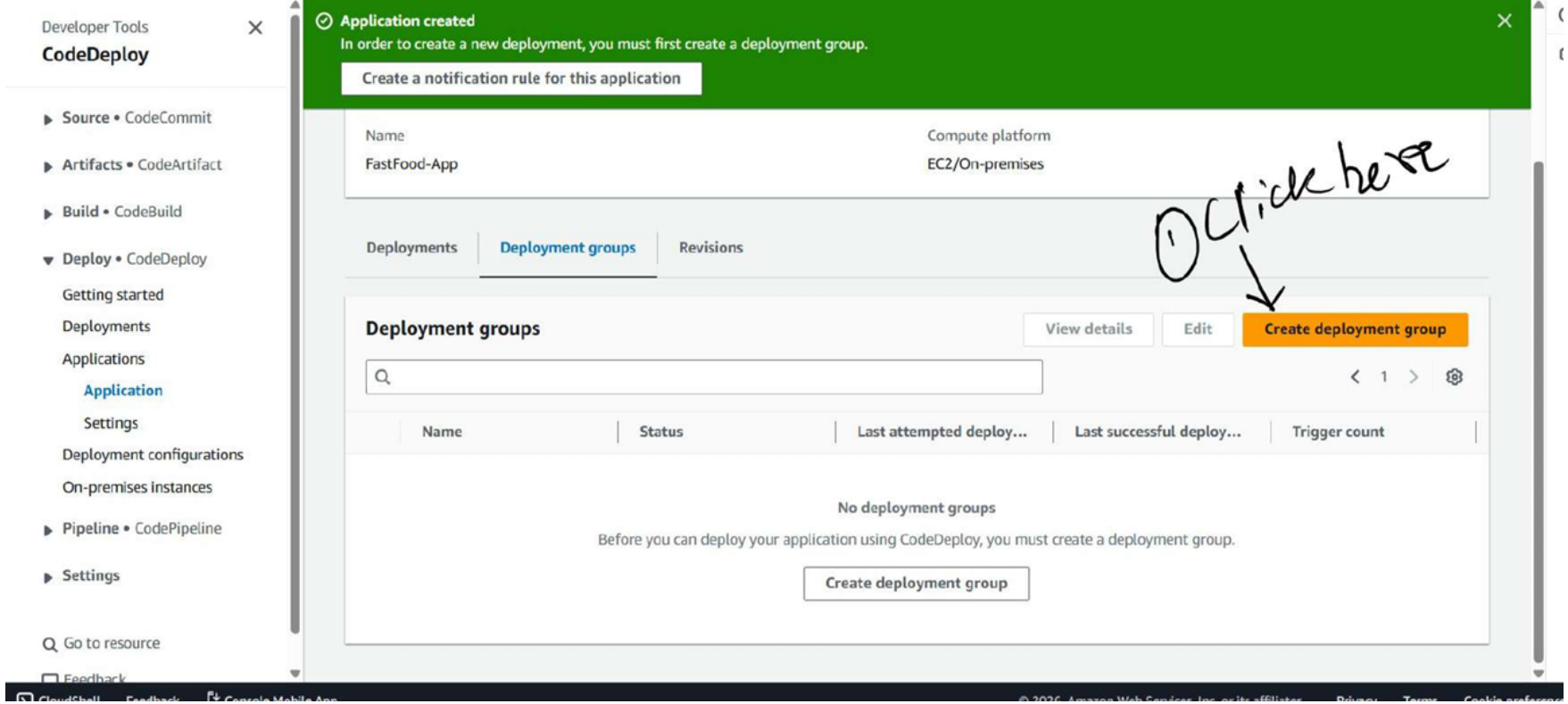
- I opened the AWS Console and navigated to the CodeDeploy service.
- In the Applications section, I clicked on “Create application” to begin the deployment setup.
- This application will manage deployments for the FastFood project.



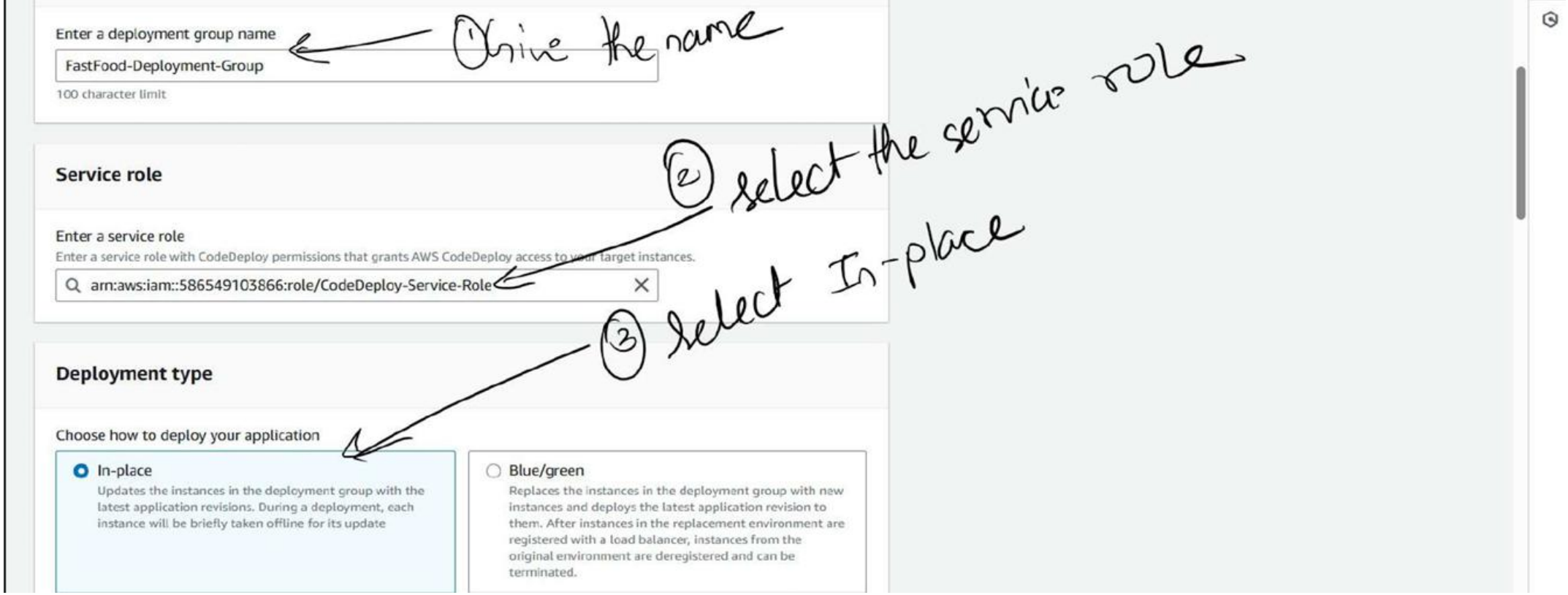
- I clicked on “Create application” to start a new deployment configuration.
- This application will manage how build artifacts are delivered to the EC2 instance.
- Creating the application is the first step before defining deployment groups and specifying deployment behavior.
- The application name will match the project it supports, keeping the CI/CD structure organized.



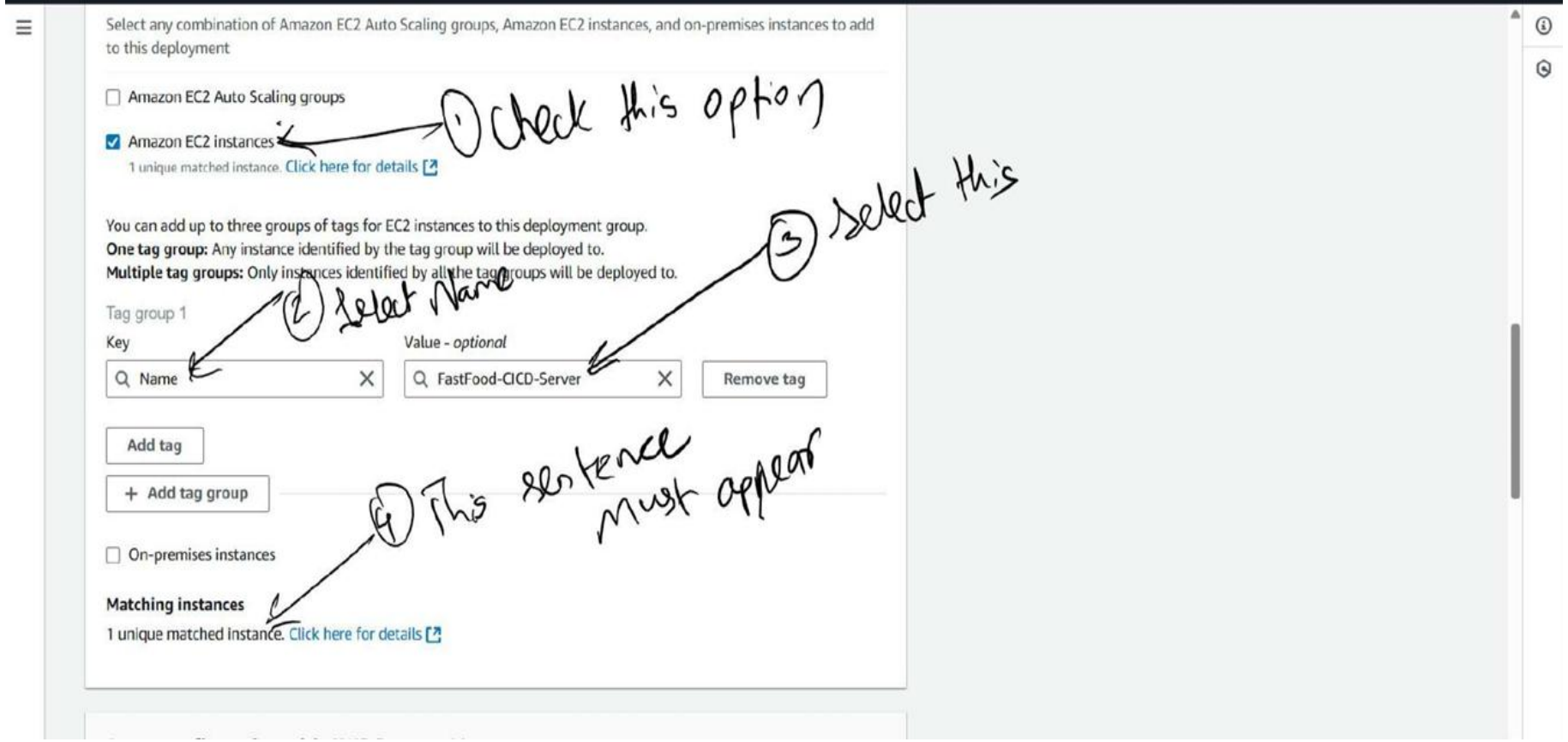
- I entered the application name which matches the project name for clarity.
- I selected the compute platform since the deployment target is an EC2 instance.
- This platform supports deployments to virtual machines managed by AWS.
- I left the tags section empty for now, as tagging is optional.
- Then I clicked “Create application” to finalize the setup.
- This completes the application creation step in CodeDeploy.



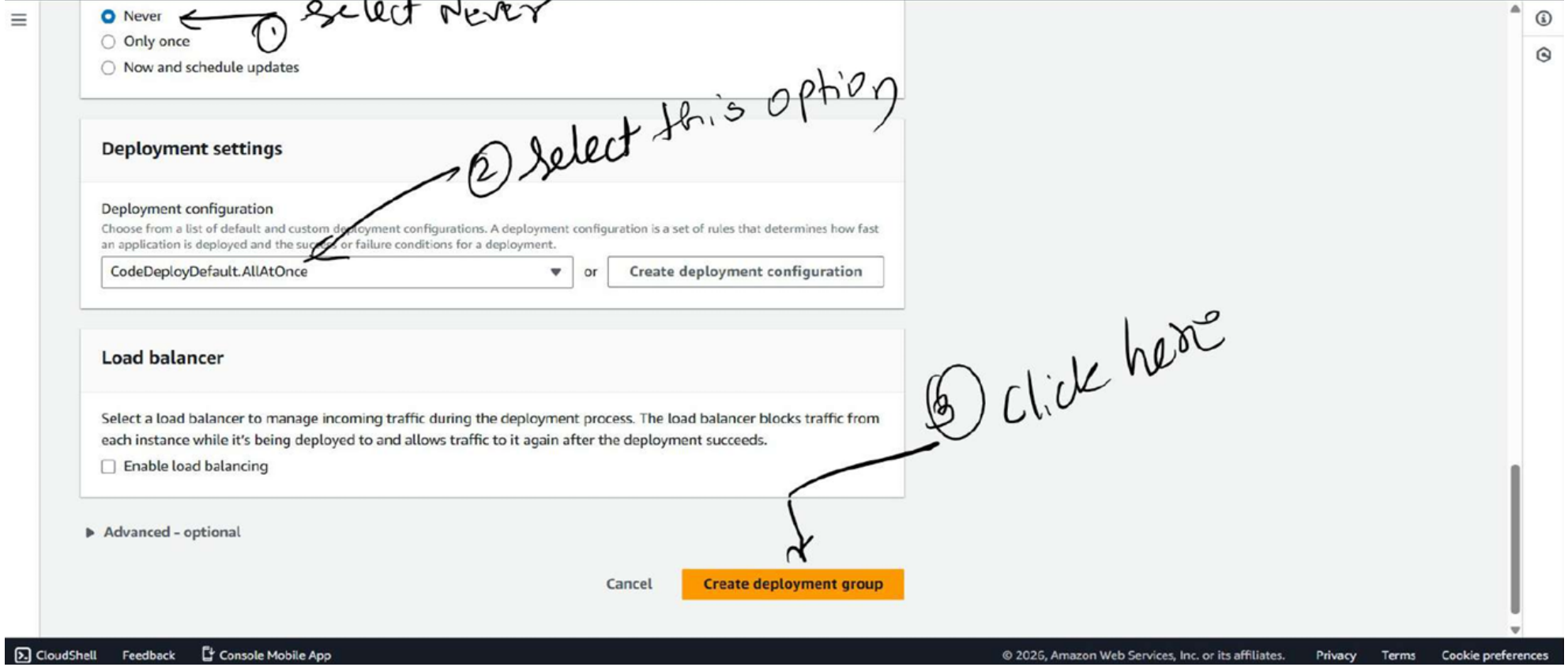
- I navigated to the Deployment groups tab, which showed no existing groups.
- I clicked on “Create deployment group” to begin configuring how deployments will be delivered to the EC2 instance.
- This step is required before any application revision can be deployed.
- The deployment group will define the target instance, deployment settings, and service role permissions.



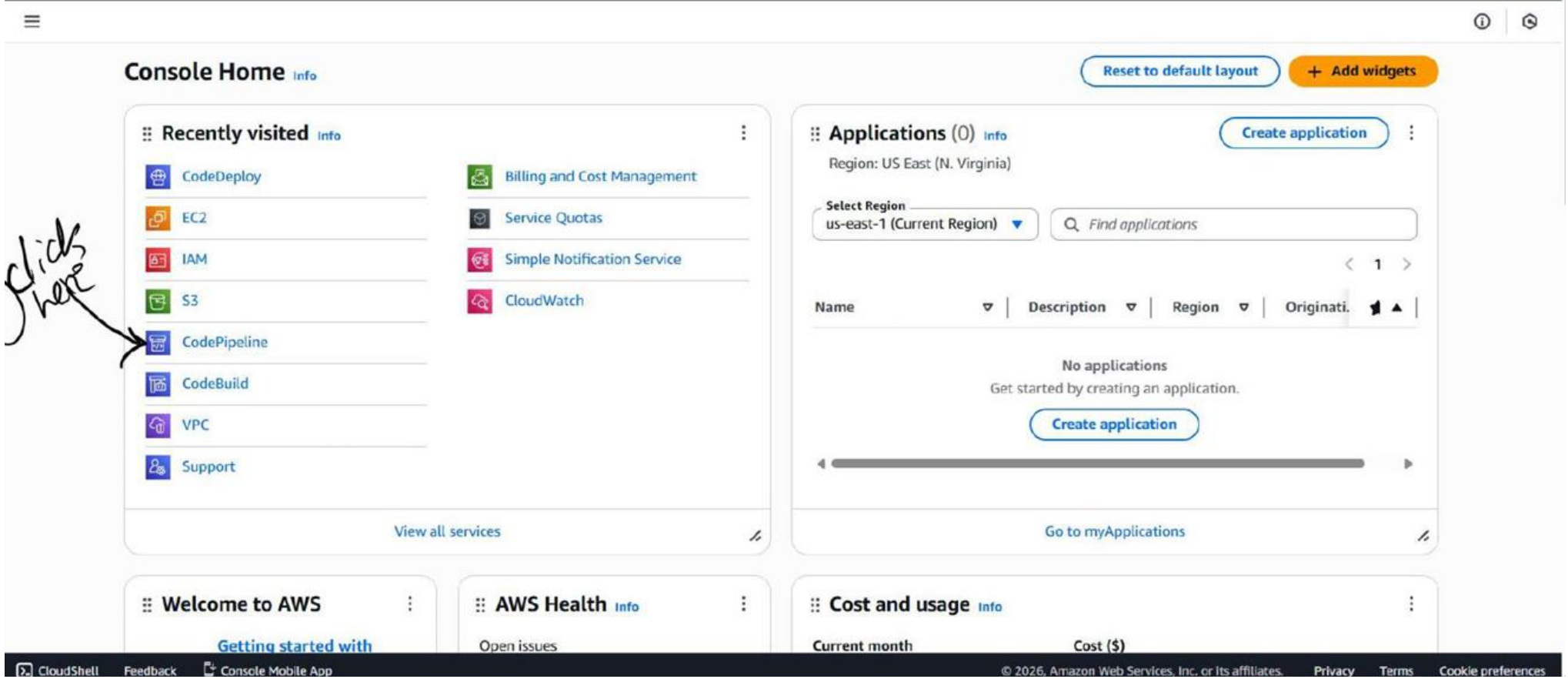
- I entered the deployment group name as FastFood-Deployment-Group, which clearly identifies the group linked to the FastFood-App.
- I selected the service role CodeDeploy-Service-Role, which includes permissions for CodeDeploy to manage deployments on EC2 instances.
- For deployment type, I chose In-place, which updates the existing EC2 instance with the latest application revision.



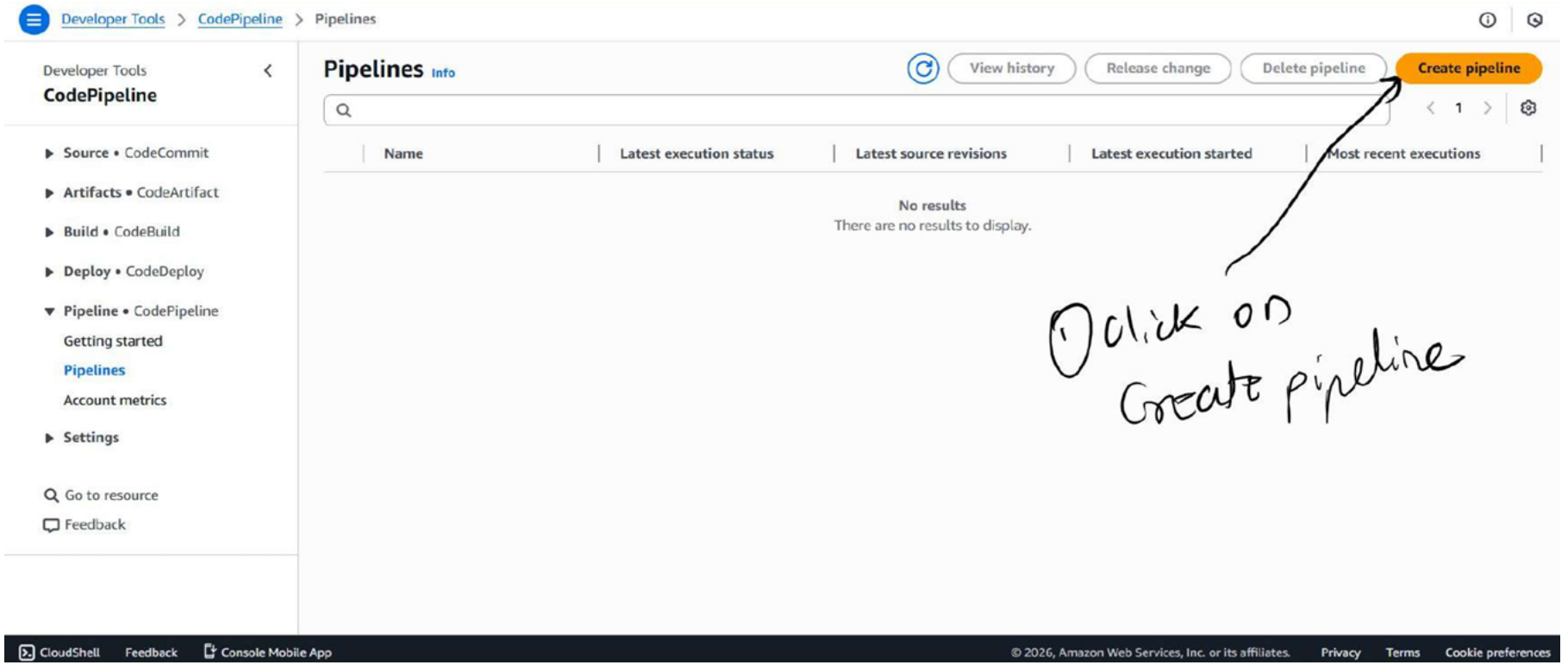
- I selected “Amazon EC2 instances” as the deployment target type.
- This allows CodeDeploy to deliver application revisions directly to the EC2 instance.
- I added a tag group with the key set to Name and the value set to FastFoodp-CICD-Server .
- This tag matches the EC2 instance created earlier for hosting the application.
- The interface confirmed “1 unique matched instance,” which verifies that the tag correctly identifies the target.



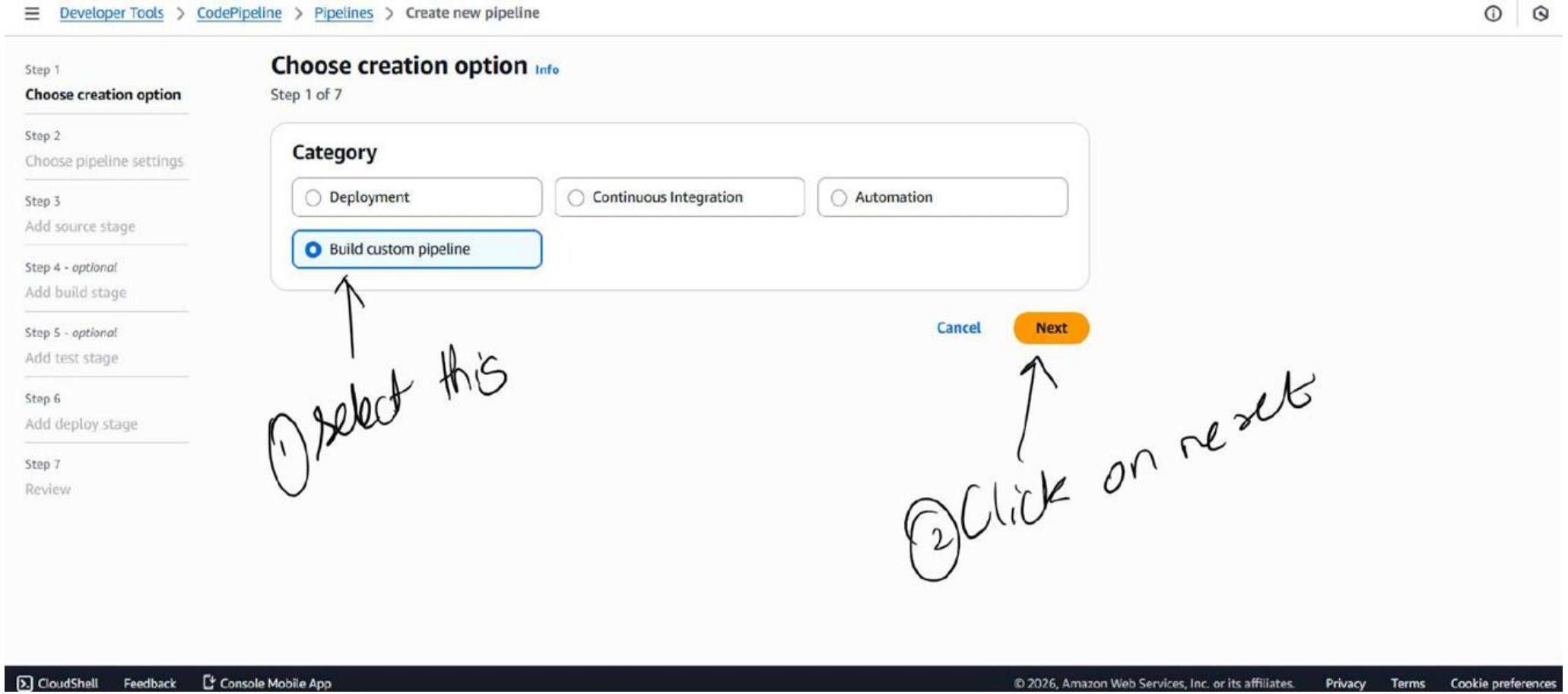
- I selected “Never” for scheduling updates, meaning deployments will only occur when manually triggered.
- For deployment configuration, I chose CodeDeployDefaultAllAtOnce , which updates all instances in the group simultaneously.
- This method is faster and suitable for single-instance deployments like FastFood-CICD-Server.
- I did not enable load balancing since the deployment targets only one EC2 instance and does not require traffic routing.
- Finally, I clicked “Create deployment group” to complete the setup.



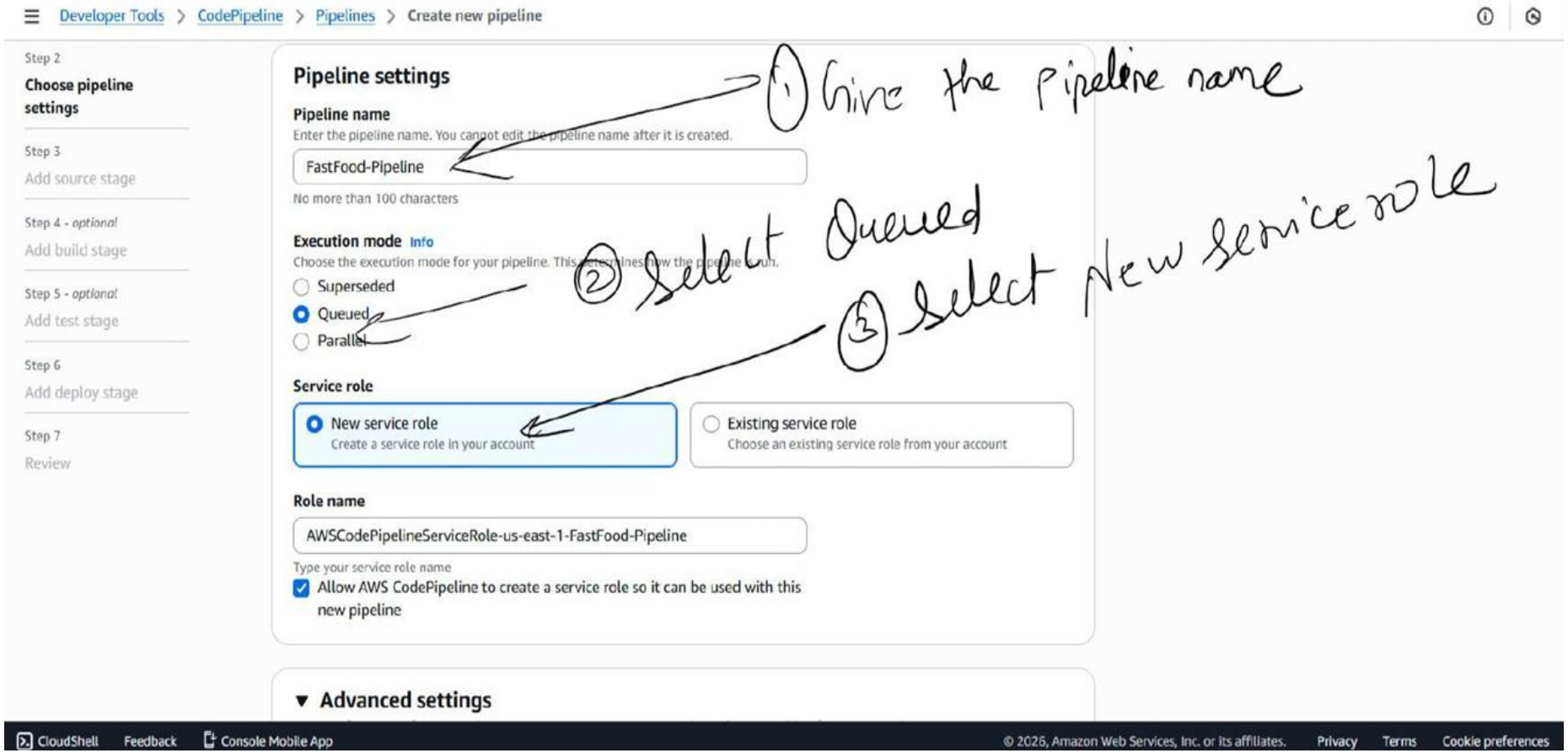
- Launching CodePipeline allows me to define the stages of the pipeline, including source retrieval, build execution, and deployment delivery.
- This step initiates the creation of a continuous integration and deployment process for the FastFood project.



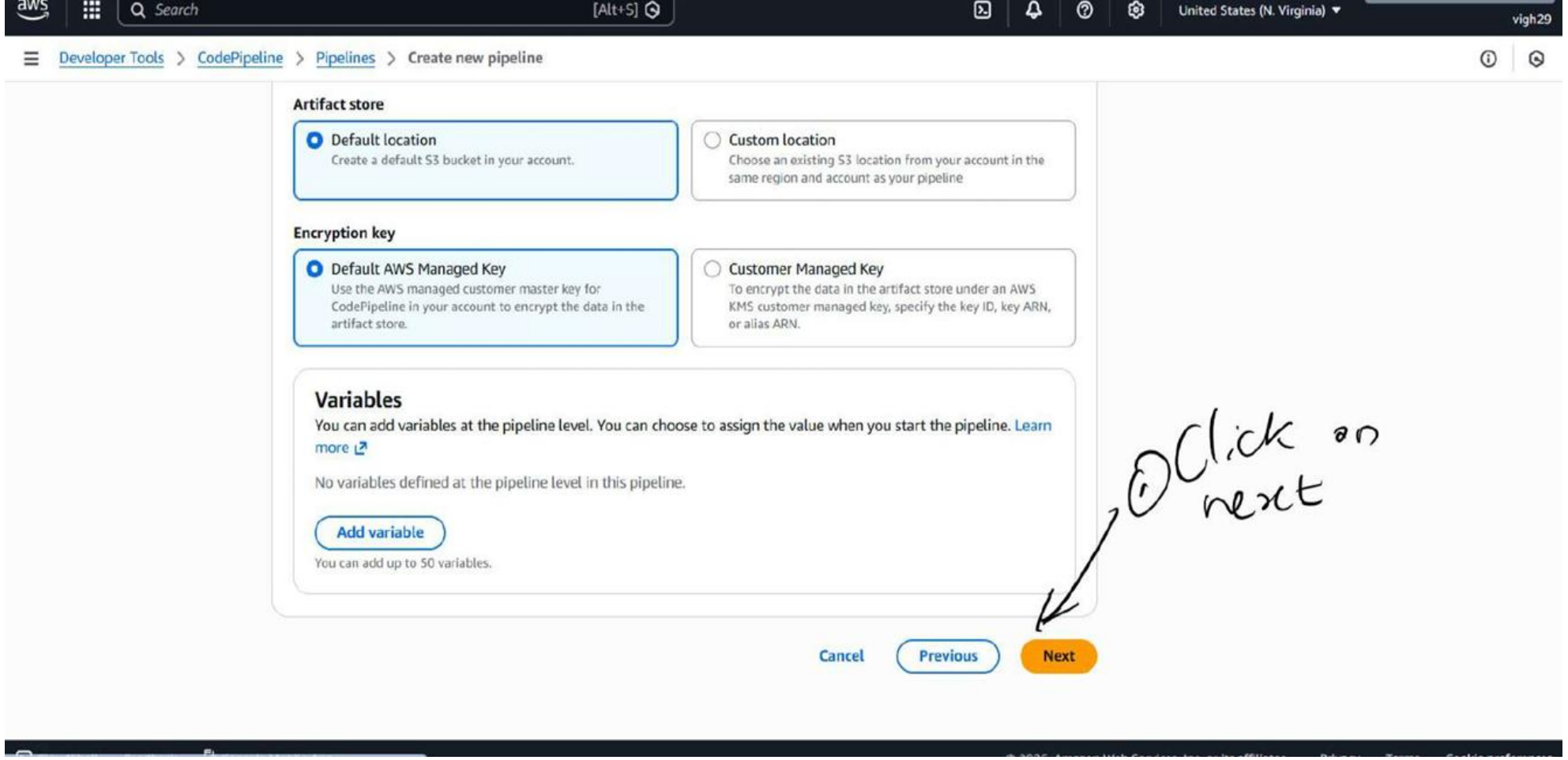
- I opened the AWS Console and navigated to the CodePipeline service.
- In the Pipelines tab, I clicked on Create pipeline to begin setting up the CI/CD workflow.
- This pipeline will automate the process by connecting CodeCommit, CodeBuild, and CodeDeploy.



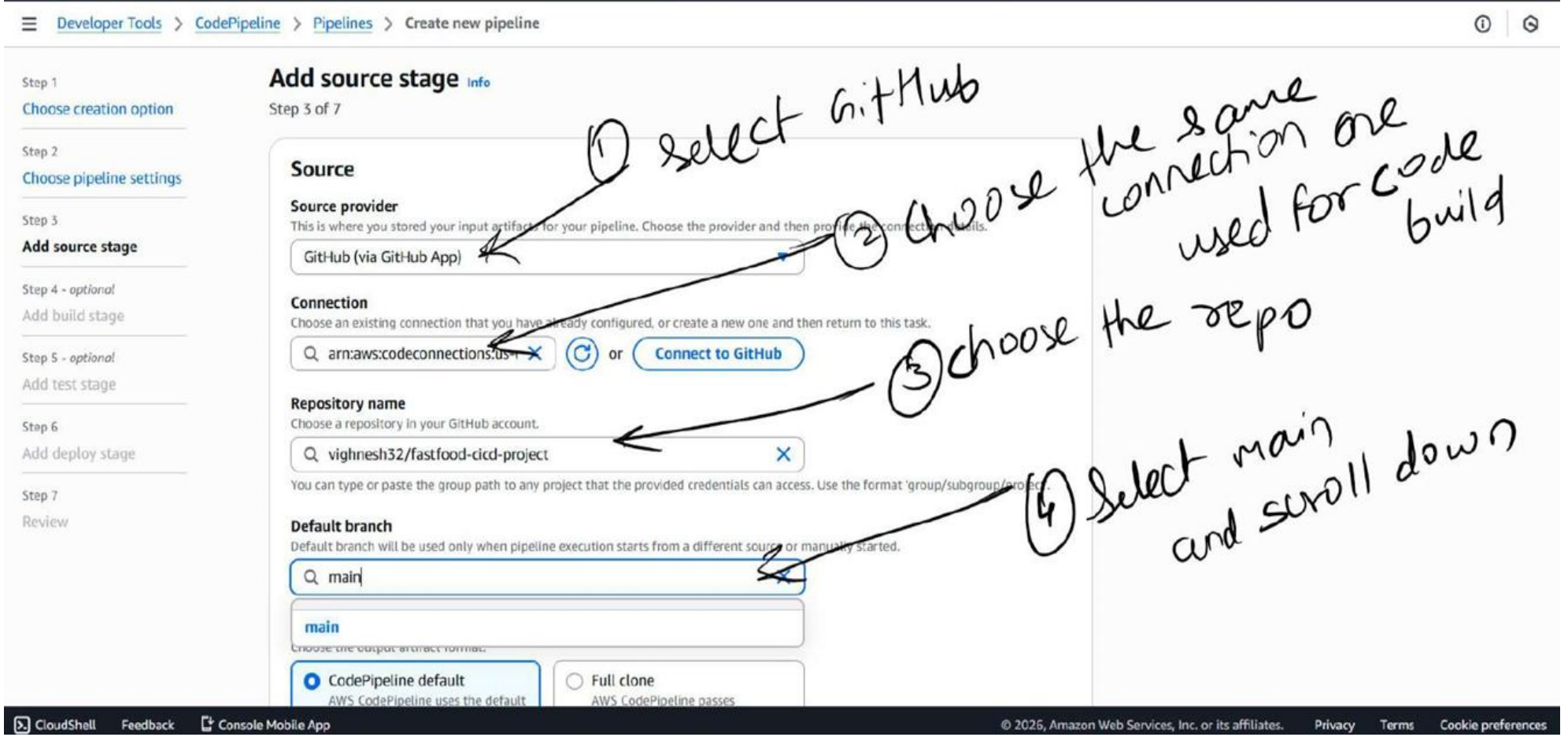
- I entered the deployment group name as FastFood-Deployment-Group.
- I selected the service role CodeDeploy-Service-Role, which grants AWS CodeDeploy access to the EC2 instance.
- For deployment type, I chose In-place, which updates the existing EC2 instance with the latest application revision.
- This method briefly takes the instance offline during deployment but avoids provisioning new infrastructure.



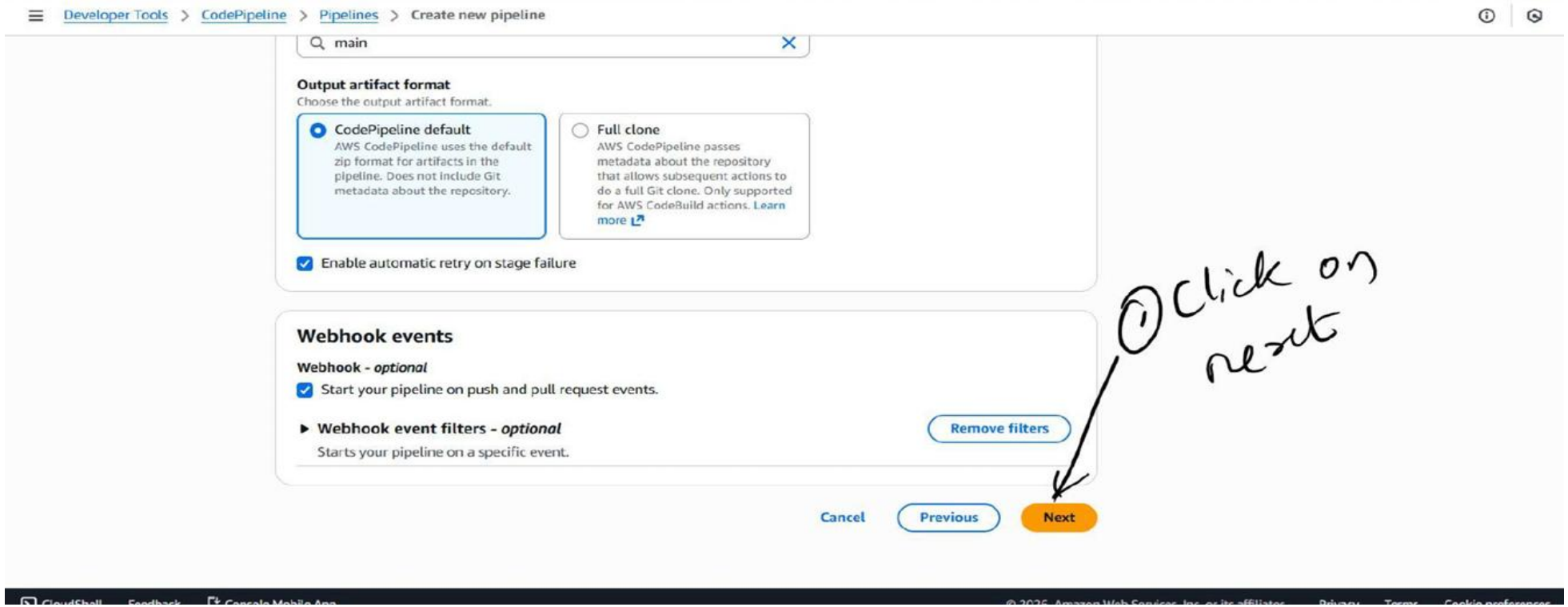
- I entered the pipeline name as FastFood-Pipeline.
- For execution mode, I selected Queued, which allows multiple executions to run in sequence.
- I chose New service role so that AWS CodePipeline can automatically create a role with the required permissions.
- The role name generated was AWSCodePipelineServiceRole-us-east-1-FastFood-Pipeline.



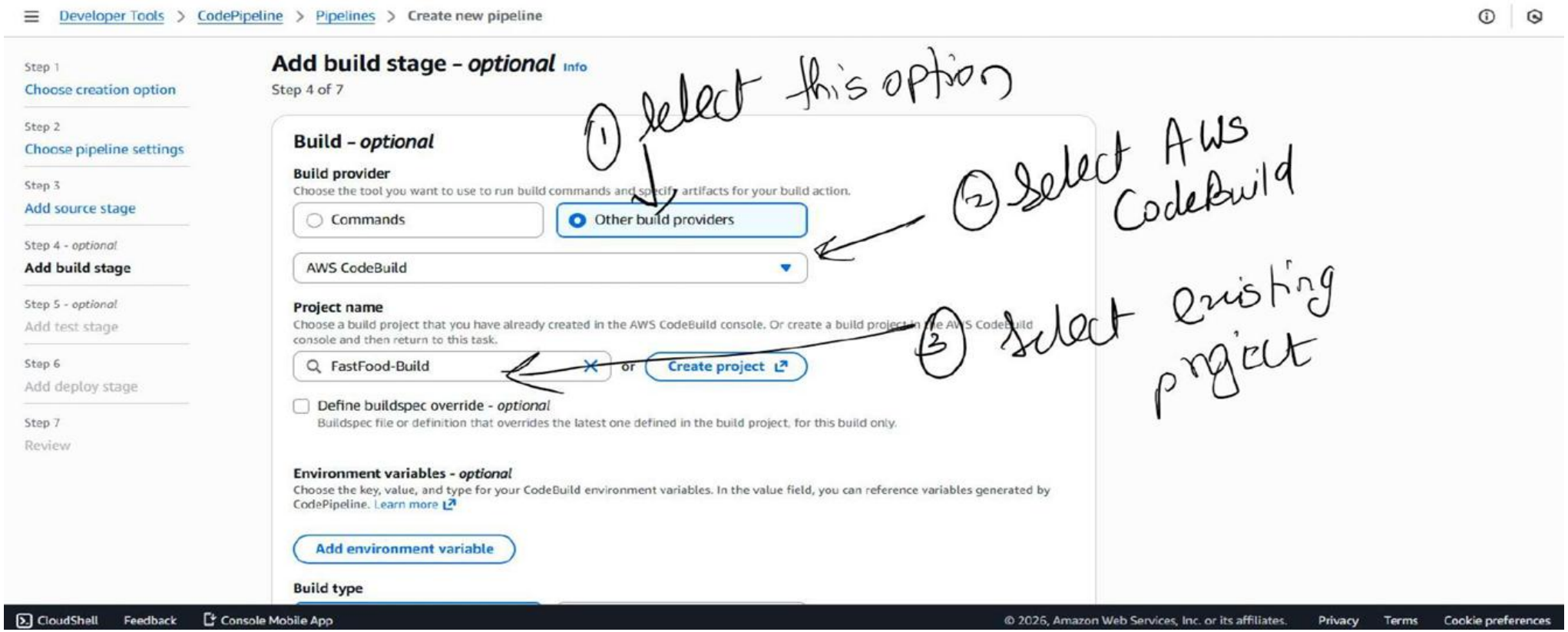
- I selected the default location for the artifact store, allowing AWS to create a default S3 bucket for storing pipeline artifacts.
- For encryption, I chose the default AWS Managed Key, which uses the built-in customer master key for CodePipeline.



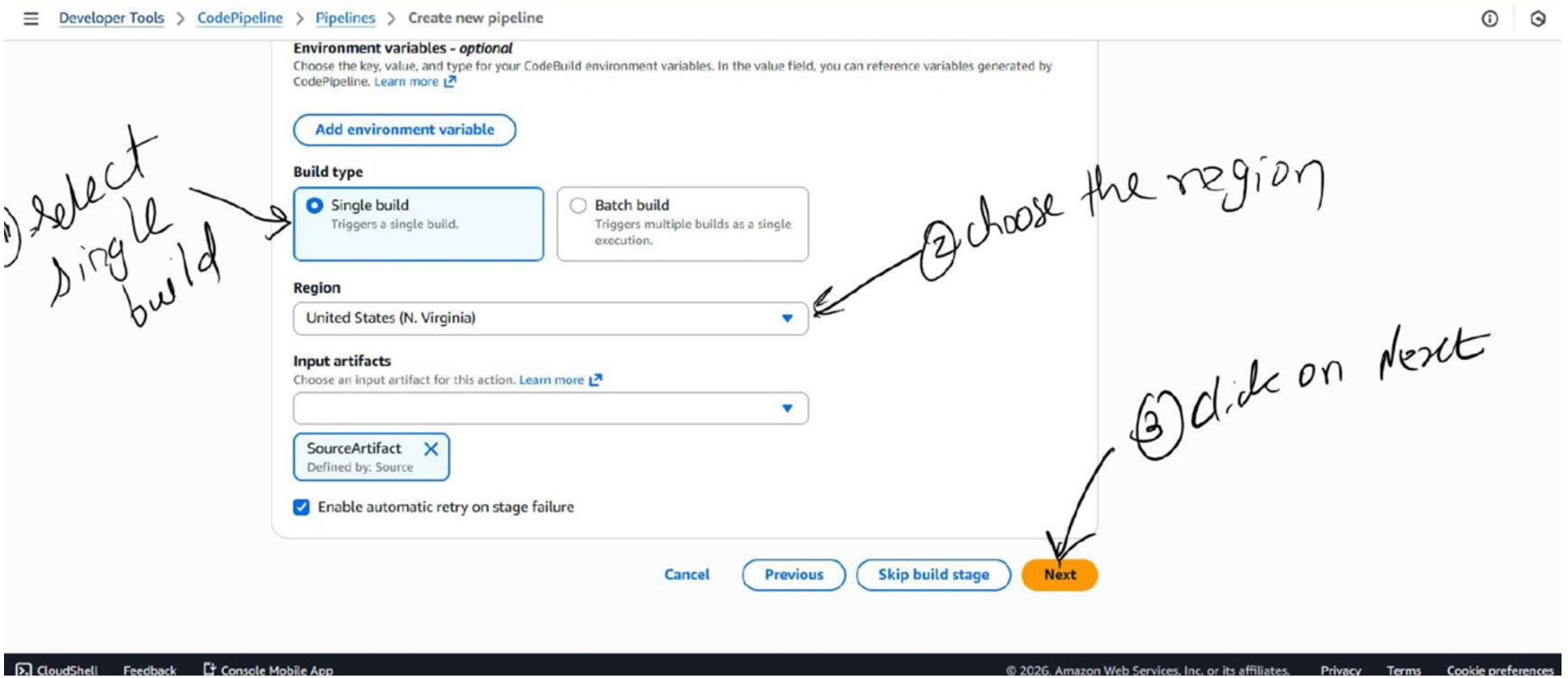
- I selected GitHub (via GitHub App) as the source provider.
- I chose the same connection used earlier for CodeBuild to maintain consistency.
- For repository name, I selected vighnesh32/fastfood-cicd-project.
- I set the default branch to main, which will be used for triggering pipeline executions.



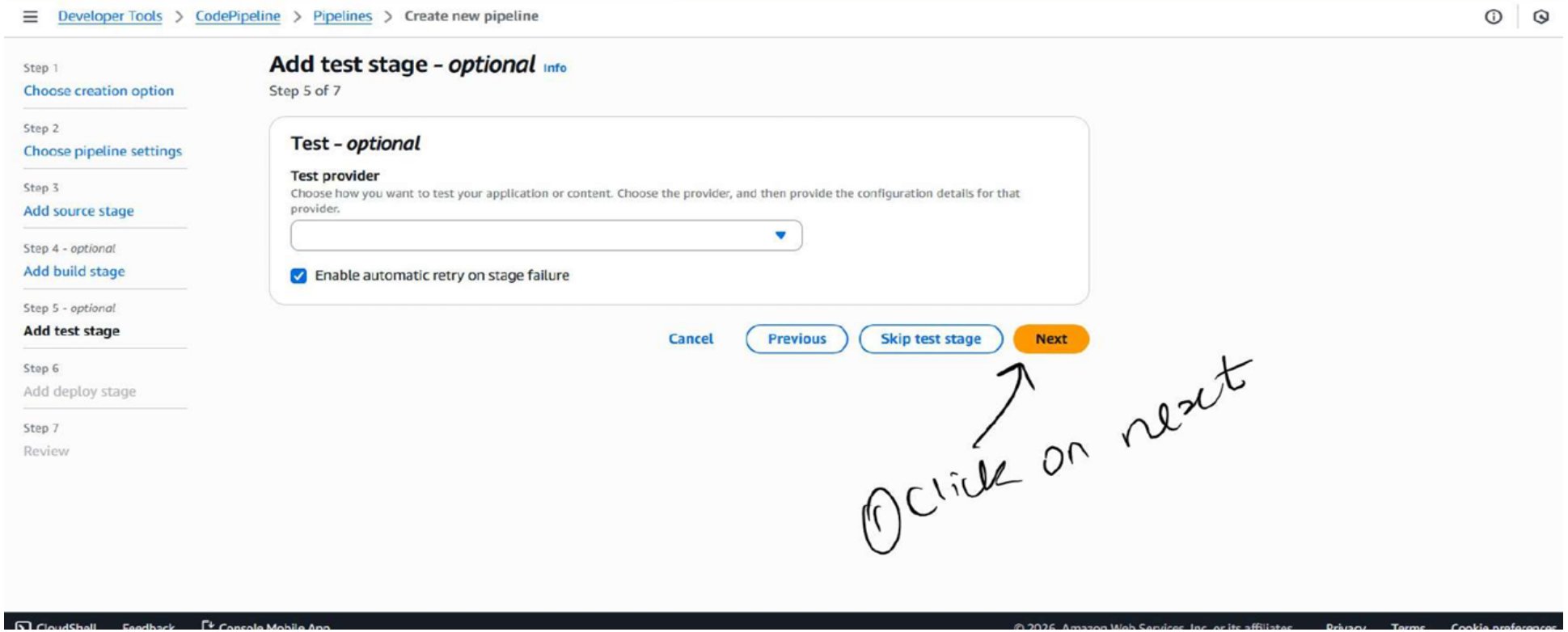
- I selected CodePipeline default as the output artifact format, which stores artifacts in zip format without Git metadata.
- I enabled automatic retry on stage failure to improve reliability during pipeline execution.
- I checked the webhook option to allow the pipeline to start automatically on push and pull request events from GitHub.



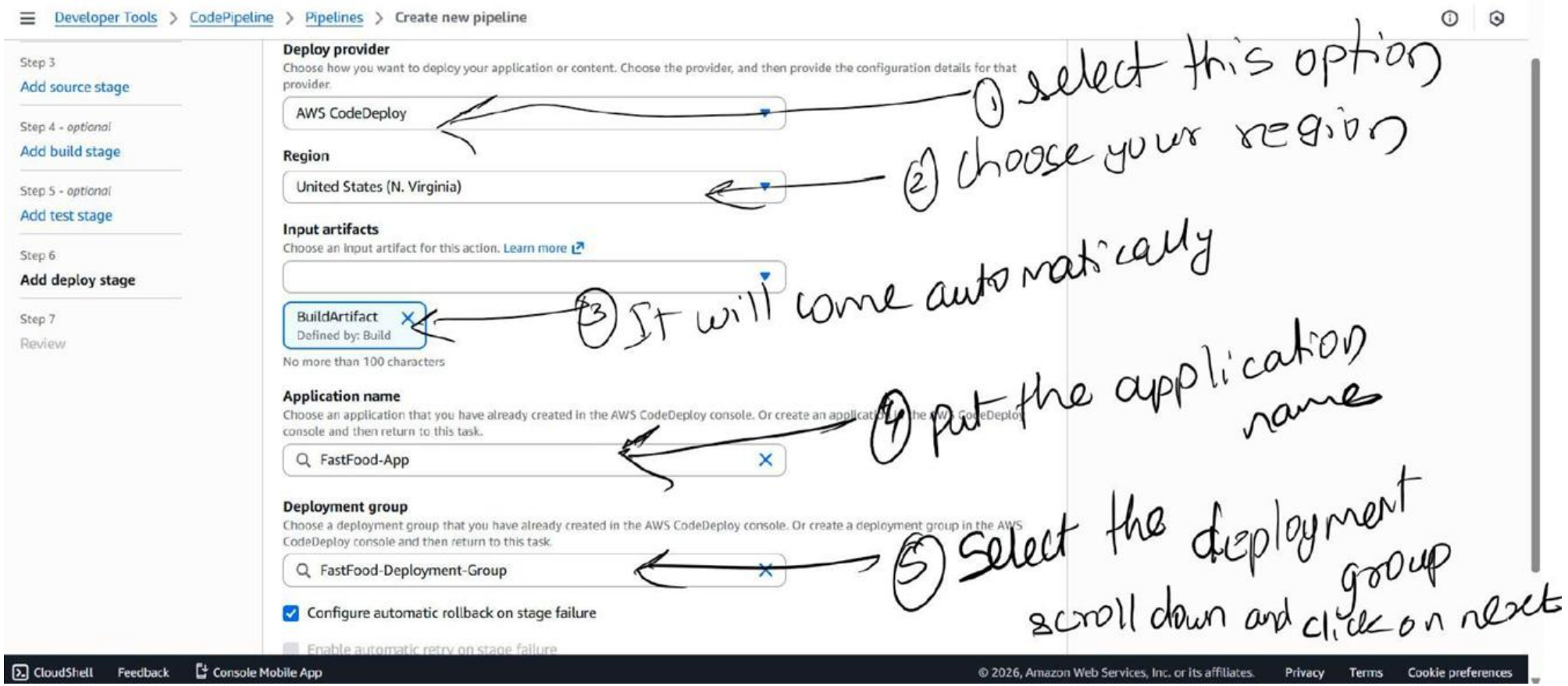
- I selected Other build providers and chose AWS CodeBuild as the build provider.
- For the project name, I selected FastFood-Build, which was already created in the CodeBuild console.
- I did not define a buildspec override or add environment variables at this stage.



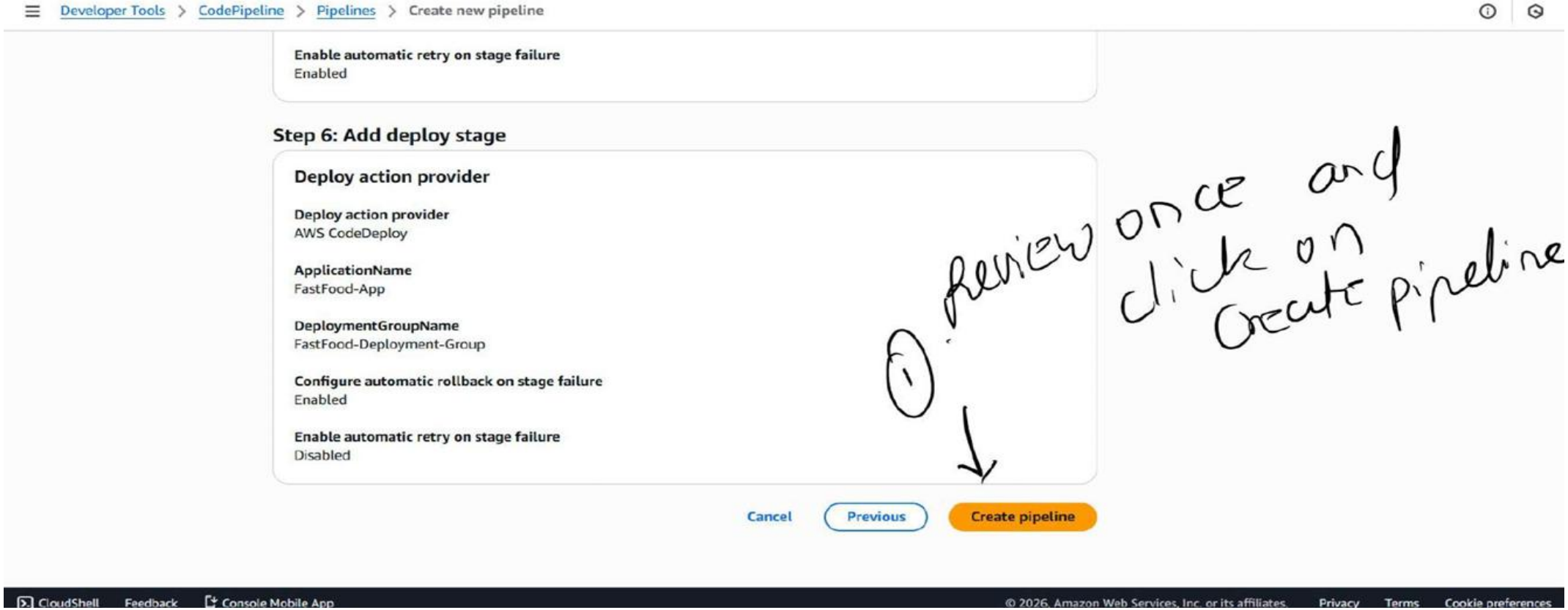
- I selected Single build as the build type, which triggers one build per pipeline execution.
- For region, I chose United States (N. Virginia) to match the location of the EC2 instance and other resources.



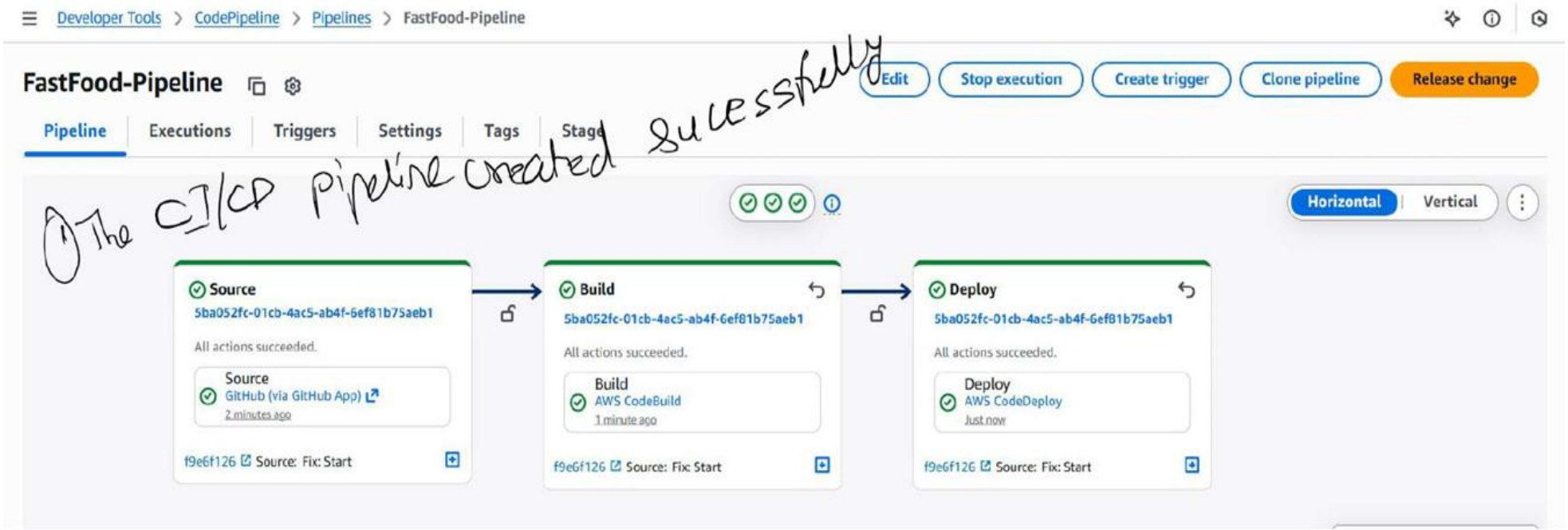
- I enabled automatic retry on stage failure to ensure resilience in case of transient issues.
- Then I clicked Next to proceed to the deploy stage configuration.



- I selected AWS CodeDeploy as the deploy provider.
- For region, I chose United States (N. Virginia) to match the location of the application and EC2 instance.
- The input artifact BuildArtifact was automatically selected from the build stage.
- I chose the application name FastFood-App, which was created earlier in the CodeDeploy console.
- For deployment group, I selected FastFood-Deployment-Group to target the correct EC2 instance.

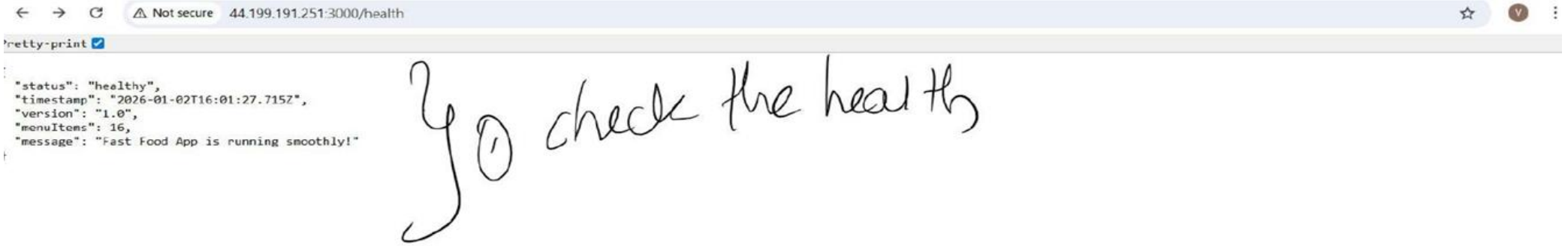


- I reviewed the deploy stage configuration with AWS CodeDeploy as the deploy action provider.

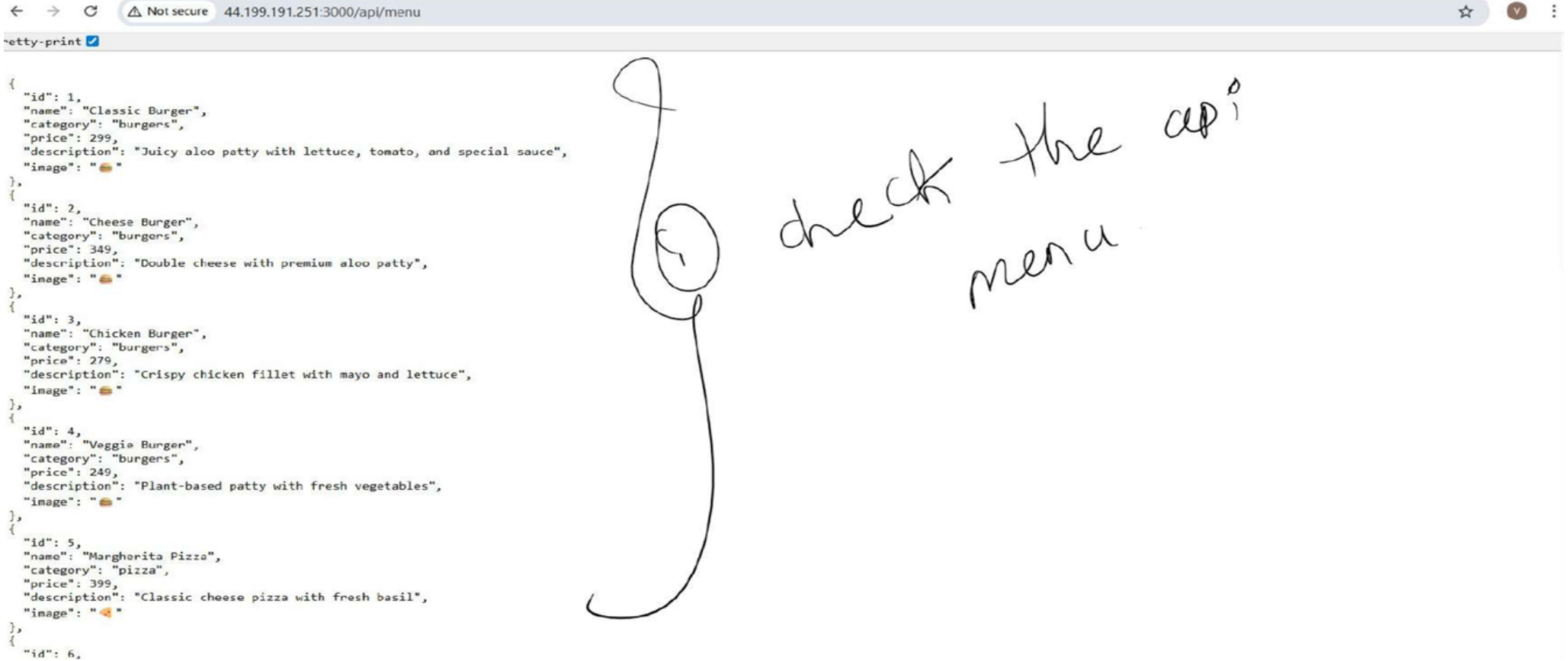


- The Build stage compiled the application using AWS CodeBuild.
- The Deploy stage delivered the application to the EC2 instance using AWS CodeDeploy.
- All actions succeeded, confirming that the CI/CD pipeline is functioning correctly.

- After successful deployment, I verified the frontend by accessing the QuickBite web interface.



- I verified the backend by accessing the health check endpoint
- The JSON response confirmed the application status as healthy with the message “Fast Food App is running smoothly!”



- I verified the API endpoint at (44.199.191.251 in Bing) to confirm menu data delivery.
- The response returned a valid JSON array containing multiple menu items with fields like id, name, category, price, description, and emoji.
- Sample items included Classic Burger, Cheese Burger, Chicken Burger, Veggie Burger, and Margherita Pizza.
- Each item was correctly categorized and priced, confirming that the backend is serving structured data as expected.