1. Continuous Integration (CI)

Task 1: Set Up a CI Pipeline

Objective: Set up a basic CI pipeline that builds a project automatically whenever new code is pushed.

Setting Up a Basic CI Pipeline

Here's how to set up a continuous integration (CI) pipeline that automatically builds your project whenever new code is pushed. I'll provide examples for GitHub Actions (GitHub) and .gitlab-ci.yml (GitLab).

Option 1: GitHub Actions

Create a .github/workflows/build.yml file in your repository:

```
name: CI Build Pipeline

on:
  push:
    branches: [ main, master ]  # Triggers on pushes to these branches
  pull_request:                 # Optional: also run on PRs
    branches: [ main, master ]

jobs:
  build:
    runs-on: ubuntu-latest     # You can change to windows/macOS if needed

  steps:
    - name: Checkout code
      uses: actions/checkout@v4
  - name: Set up JDK (Java example - change for your language)
      uses: actions/setup-java@v3
      with:
        java-version: '17'
        distribution: 'temurin'


    - name: Build with Gradle (or Maven/npm/etc.)
      run: ./gradlew build


    # Add more steps as needed (testing, packaging, etc.)
```

Option 2: GitLab CI/CD

Create a .gitlab-ci.yml file in your repository root:

```
stages:
  - build
build_job:
  stage: build
  image: openjdk:17-jdk  # Change to appropriate image for your stack
  script:
    - ./gradlew build    # Or your build command (mvn package, npm run build, etc.)
  only:
    - main             # Only run on main branch
    - master           # And/or master branch
    - merge_requests     # Optional: also run on MRs
```

Option 3: Jenkins

Create a Jenkinsfile in your repository root:

```
pipeline {
    agent any

    triggers {
        pollSCM('* * * * *')  // Polls SCM every minute (adjust as needed)
    }

    stages {
        stage('Build') {
            steps {
                checkout scm
                sh './gradlew build'  // Your build command
            }
        }
    }
}
```

Key Components for Any CI Pipeline:

1 Trigger: When to run (typically on push to main branches)

2. Environment Setup: Install dependencies, SDKs, etc.
3. Build Command: The actual build command for your project
4. Optional Steps: Testing, code quality checks, packaging, etc.

Language-Specific Notes:

1. JavaScript/TypeScript: Use npm install && npm run build
2. Python: pip install -r requirements.txt && pytest
3. Go: go build ./...
4. Rust: cargo build

Tools: GitHub Actions, Jenkins, GitLab CI, or CircleCI

Instructions

1. Create a simple application (use Node.js, Python, or Java)

I'll create a simple Node.js application with a basic CI pipeline using GitHub Actions. This example includes:

A minimal Express.js web server

Unit tests with Jest

GitHub Actions CI configuration

```
my-node-app/
├── .github/
│   └── workflows/
│       └── ci.yml      # GitHub Actions config
├── src/
│   ├── app.js          # Main application
│   └── __tests__/
│       └── app.test.js  # Unit tests
├── package.json
└── README.md
```

. Application Files

src/app.js (Express server)

(javascript)

```
const express = require('express');
const app = express();
```

```
const PORT = process.env.PORT || 3000;

// Simple route
app.get('/', (req, res) => {
  res.json({ message: "Hello CI/CD World!" });
});

// Health check endpoint
app.get('/health', (req, res) => {
  res.status(200).send('OK');
});

module.exports = app;

// Start server only when not in test mode
if (process.env.NODE_ENV !== 'test') {
  app.listen(PORT, () => {
    console.log(`Server running on port ${PORT}`);
  });
}
```

src/__tests__/app.test.js (Jest tests)

```
javascript
const request = require('supertest');
const app = require('../app');

describe('GET /', () => {
  it('responds with JSON message', async () => {
    const response = await request(app).get('/');
    expect(response.statusCode).toBe(200);
    expect(response.body.message).toBe('Hello CI/CD World!');
  });
});
```

```
  describe('GET /health', () => {
    it('responds with 200 OK', async () => {
      const response = await request(app).get('/health');
      expect(response.statusCode).toBe(200);
      expect(response.text).toBe('OK');
    });
  });
```

package.json

json

```
{
    "name": "my-node-app",
    "version": "1.0.0",
    "scripts": {
      "start": "node src/app.js",
      "test": "jest --coverage",
      "dev": "nodemon src/app.js"
    },
    "dependencies": {
      "express": "^4.18.2"
    },
    "devDependencies": {
      "jest": "^29.7.0",
      "supertest": "^6.3.3",
      "nodemon": "^3.0.2"
    }
  }
```

CI Pipeline (GitHub Actions)

.github/workflows/ci.yml

Yaml

```
        name: Node.js CI

on: [push, pull_request]
```

```
jobs:
  build-and-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Set up Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '20'

      - name: Install dependencies
        run: npm ci

      - name: Run tests
        run: npm test

      # Optional: Add linting step
      # - name: Run ESLint
      #   run: npx eslint src/
```

How to Run Locally

Install dependencies

Bash

```
npm install
```

Start the server:

Bash

```
npm start
```

Run tests:

Bash

```
npm test
```

CI Pipeline Features

✅ Automatic triggers on push/PR
✅ Dependency caching (implicit in npm ci)
✅ Unit testing with coverage
✅ Fast feedback (runs in ~30 seconds)

Alternative Versions

Want this in Python (Flask) or Java (Spring Boot) instead? Here's a quick comparison:

Language	Framework	Test Runner	CI Command
Python	Flask	pytest	pytest --cov
Java	Spring	JUnit	mvn test

2. Set up a repository on GitHub or GitLab$

I'll guide you through setting up a repository on GitHub and GitLab, including pushing the Node.js application we just created. Here are the steps for both platforms:

Option 1: GitHub Repository Setup

1. Create a New Repository

Go to github.com/new

Enter repository name (e.g., my-node-app)

Choose Public or Private

Do not initialize with README/.gitignore (we'll push existing code)

Click "Create repository"

2. Push Your Code

Run these commands in your project folder:

bash

Initialize Git (if not already done)

git init

Add files and commit

git add .

git commit -m "Initial commit with Node.js app and CI pipeline"

Link to GitHub (replace URL with your repo's URL)

git remote add origin https://github.com/your-username/my-node-app.git

Push to GitHub

git push -u origin main

3. Verify CI Pipeline

Go to your repo's Actions tab

You should see the workflow running automatically!

Option 2: GitLab Repository Setup

1. Create a New Project

Go to gitlab.com/projects/new

Click "Create blank project"

Enter project name (e.g., my-node-app)

Choose visibility (Public or Private)

Uncheck "Initialize repository with README"

Click "Create project"

2. Push Your Code

bash

Initialize Git (if not already done)

git init

Add files and commit

git add .

git commit -m "Initial commit with Node.js app"

Link to GitLab (replace URL with your repo's URL)

git remote add origin https://gitlab.com/your-username/my-node-app.git

Push to GitLab

git push -u origin main

3. Set Up CI/CD

GitLab will automatically detect the .gitlab-ci.yml file

Go to CI/CD > Pipelines to see it running

Key Differences

Feature	GitHub	GitLab
CI Config File	.github/workflows/ci.yml	.gitlab-ci.yml
CI Visibility	Under "Actions" tab	Under "CI/CD > Pipelines"
Free Runners	2,000 mins/month (Linux)	400 mins/month (Linux)
Private Repos	Free for all	Free for all

Troubleshooting

Authentication Issues:

Use SSH instead of HTTPS if you prefer:

bash

git remote set-url origin git@github.com:your-username/repo.git

CI Not Triggering:

Check file paths (.github/workflows/ for GitHub, .gitlab-ci.yml for GitLab)

Verify the file is committed

Next Steps

Add a README.md:

markdown

My Node.js App

A simple CI/CD demo with:

- Express.js server

- Jest tests

- GitHub Actions/GitLab CI

How to Run

```bash

npm install

npm start
```

Add branch protection rules (in repo settings):

Require CI to pass before merging

Require PR reviews

Would you like me to customize any part of this setup further?

3. Configure a CI tool to automatically build the application whenever new code is pushed to the repository$

Here's how to configure automatic builds on code push for each major CI tool, using our Node.js example:

---

1. GitHub Actions

Already configured in the previous example! The on: [push] trigger in .github/workflows/ci.yml makes it automatic. For more control:

yaml

```
# Enhanced version with caching and matrix testing
name: Node.js CI

on:
  push:
    branches: [ main, dev ]  # Specific branches
    paths:                   # Only trigger on certain file changes
      - 'src/**'
      - 'package.json'
  pull_request:
    branches: [ main ]

jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node-version: [18.x, 20.x]  # Test across Node versions

    steps:
```

```
    - uses: actions/checkout@v4
    - uses: actions/setup-node@v3
      with:
        node-version: ${{ matrix.node-version }}
    - uses: actions/cache@v3
      with:
        path: '**/node_modules'
        key: ${{ runner.os }}-node-${{ hashFiles('package-lock.json') }}
    - run: npm ci
    - run: npm test
```

---

2. GitLab CI

Add this .gitlab-ci.yml to your repository root:

yaml

```
stages:
  - build
  - test

cache:
  paths:
    - node_modules/

build_job:
  stage: build
  image: node:20
  script:
    - npm ci
    - npm run build  # If you have a build script
  rules:
    - if: $CI_PIPELINE_SOURCE == "push"  # Only on push events

test_job:
  stage: test
```

```
  image: node:20
  script:
    - npm test
```

---

3. Jenkins

For a declarative pipeline, add this Jenkinsfile to your repo:

```
groovy
pipeline {
    agent any
    triggers {
        pollSCM('H/2 * * * *')  // Check for changes every 2 minutes
    }
    stages {
        stage('Build') {
            steps {
                sh 'npm install'
            }
        }
        stage('Test') {
            steps {
                sh 'npm test'
            }
        }
    }
    post {
        always {
            cleanWs()  // Clean workspace after build
        }
    }
}
```

To set up webhooks (better than polling):

Go to Manage Jenkins > System Configuration

Install the "GitHub" or "GitLab" plugin

Add a webhook in your repo settings: http://your-jenkins-server/github-webhook/

---

4. CircleCI

Add this .circleci/config.yml:

yaml

```
version: 2.1

jobs:
  build-and-test:
    docker:
      - image: cimg/node:20
    steps:
      - checkout
      - restore_cache:
          keys:
            - v1-deps-{{ checksum "package-lock.json" }}
      - run:
          name: Install dependencies
          command: npm ci
      - save_cache:
          paths:
            - ./node_modules
          key: v1-deps-{{ checksum "package-lock.json" }}
      - run:
          name: Run tests
          command: npm test

workflows:
  main:
    jobs:
      - build-and-test:
          filters:
            branches:
              only: [ main, dev ]  # Only these branches
```

Key Configuration Points

Tool	Automatic Trigger Config	Best For
GitHub	on: [push] in YAML	GitHub-hosted projects
GitLab	rules: - if: $CI_PIPELINE_SOURCE == "push"	GitLab ecosystems
Jenkins	triggers { pollSCM() } or webhooks	Self-hosted complex pipelines
CircleCI	filters: branches: only:	Cloud-native projects

Verification Steps

Push a code change:

bash

```
touch dummy-file && git add . && git commit -m "Trigger build" && git push
```

Check your CI tool's dashboard:

GitHub: Actions tab

GitLab: CI/CD > Pipelines

Jenkins: Job status page

CircleCI: Jobs in app

Advanced Options

Conditional Builds:

yaml

```
# GitHub example - only run if src/ changed
on:
  push:
    paths:
      - 'src/**'
```

Scheduled Builds (GitHub):

yaml

```
on:
  schedule:
    - cron: '0 0 * * *'  # Daily midnight UTC
```

Build Notifications:

Add Slack/Teams alerts using your CI tool's notification settings

Need help adapting this for a specific use case? Let me know!

4.Document each step with screenshots showing the pipeline running

Here's a step-by-step guide with screenshots showing how to configure and verify a CI pipeline for automatic builds on code push, using GitHub Actions (the most universally accessible option):

---

1. Create the Repository

Step: Go to github.com/new

Name your repo (e.g., my-node-app)

Do not initialize with README/.gitignore (we'll push existing code).

---

2. Push Your Code

Steps:

bash

git init

git add .

git commit -m "Initial commit"

git branch -M main

git remote add origin https://github.com/your-username/my-node-app.git

git push -u origin main

(Terminal after successful push):

---

3. Set Up GitHub Actions

Step: Create .github/workflows/ci.yml with:

yaml

name: Node.js CI

on: [push]

jobs:

build:

runs-on: ubuntu-latest

steps:

```
    - uses: actions/checkout@v4
    - uses: actions/setup-node@v3
      with:
        node-version: '20'
    - run: npm ci
    - run: npm test
```

(File in GitHub's file explorer):

---

4. Verify the Pipeline Runs Automatically

Step: After pushing, go to the Actions tab.

The workflow triggers immediately on push.

Green checkmark = success.

---

5. Inspect Build Logs

Step: Click on the workflow run.

Expand steps to see detailed logs (e.g., npm test output).

---

6. Break the Build (Optional Test)

Step: Push a failing test (e.g., modify app.test.js to expect 404 instead of 200).

Red X = failure. Click to debug.

Key CI Events in GitHub

Event	Description
Push	Triggers on any code push.
PR	Runs on pull requests (if configured).
Schedule	Cron jobs (e.g., nightly builds).

GitLab CI Equivalent

For GitLab, the process is similar:

Add .gitlab-ci.yml to your repo.

Pipelines appear under CI/CD > Pipelines:

Troubleshooting

Workflow Not Triggering?

Check file location (must be .github/workflows/*.yml).

Verify the on: [push] directive exists.

Build Failing?

Check logs for errors (e.g., missing package.json).

Slow Builds?

Add caching (see the enhanced GitHub Actions example above).

Final Result

A fully automated CI pipeline that:
✓ Builds on every push
✓ Runs tests
✓ Provides visible pass/fail status

Need screenshots for another tool (Jenkins/GitLab/CircleCI)? Let me know!

2. Jenkins

Task 2: Automated Build with Jenkins

Objective: Install Jenkins and configure an automated build job.

Tools: Jenkins, Docker (optional)

Instructions:

Here's a step-by-step guide to install Jenkins either locally or on a cloud platform (AWS/Azure), with screenshots for key steps:

Option 1: Local Installation (/Linux/Windows)

Step 1: Install Prerequisites

Java JDK 11+ (Jenkins requires Java):

bash

Ubuntu/Debian

sudo apt update && sudo apt install openjdk-17-jdk

Windows: Download from [Adoptium](https://adoptium.net/)

Verify:

bash

java -version

Step 2: Install Jenkins

Linux/:

bash

Ubuntu/Debian (official package)

curl -fsSL https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key | sudo tee \

  /usr/share/keyrings/jenkins-keyring.asc > /dev/null

echo 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 > /dev/null

sudo apt update && sudo apt install jenkins

Step 3: Start Jenkins

bash

Linux (systemd)

sudo systemctl start jenkins

sudo systemctl enable jenkins # Auto-start on boot

Verify Jenkins is running:
Open http://localhost:8080 in your browser.

Step 4: Unlock Jenkins

Get the initial admin password:

bash

Linux

sudo cat /var/lib/jenkins/secrets/initialAdminPassword

Paste it into the web UI.

Step 5: Complete Setup

Install suggested plugins:

Create an admin user:

Finish installation:

Option 2: Cloud Installation (AWS/Azure)

AWS (EC2 Instance)

Launch an EC2 instance (Ubuntu 22.04 LTS, t2.medium recommended).

Connect via SSH and run:

bash

```
sudo apt update && sudo apt install openjdk-17-jdk
curl -fsSL https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key | sudo tee \
  /usr/share/keyrings/jenkins-keyring.asc > /dev/null
echo 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 > /dev/null
sudo apt update && sudo apt install jenkins
sudo systemctl start jenkins
```

Open port 8080 in EC2 Security Group.

Access Jenkins at http://<EC2_PUBLIC_IP>:8080.

Azure (VM)

Create a Ubuntu 22.04 VM (Standard B2s tier).

Run the same commands as AWS to install Jenkins.

Add inbound port rule for 8080 in Azure NSG.

Access via http://<VM_PUBLIC_IP>:8080.

Post-Installation Steps

Set Up Reverse Proxy (Optional)
Use Nginx/Apache to expose Jenkins securely:

nginx

```
# Nginx config (/etc/nginx/sites-available/jenkins)
server {
```

```
    listen 80;
    server_name jenkins.your-domain.com;
    location / {
        proxy_pass http://localhost:8080;
        proxy_set_header Host $host;
    }
}
```

Enable HTTPS with Let’s Encrypt:

```
bash
sudo apt install certbot python3-certbot-nginx
sudo certbot --nginx -d jenkins.your-domain.com
```

Verify Jenkins CI Pipeline

Create a Jenkinsfile in your repo:

```
groovy
Copy
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sh 'npm install'
            }
        }
        stage('Test') {
            steps {
                sh 'npm test'
            }
        }
    }
}
```

Set up a pipeline job:

New Item > Pipeline > Select "Pipeline script from SCM".

Trigger a build: Push code to see automatic builds.

Troubleshooting

Issue	Solution
Port 8080 unavailable	Change Jenkins port in /etc/default/jenkins (HTTP_PORT=8081)
Plugin install fails	Check proxy settings in Manage Jenkins > Plugin Manager > Advanced
Builds stuck in queue	Increase executor count in Manage Jenkins > Nodes

---

Next Steps

Add GitHub/GitLab Webhooks for instant triggers.

Configure Email/Slack Notifications.

Set Up Build Artifacts (e.g., JAR files for Java apps).

Need help with any specific step? Ask for more detailed screenshots!

2. Set up a simple job that clones a Git repository and builds the project (e.g., a Java project using Maven or a Node.js project)$

Here's how to set up a simple Jenkins job to clone a Git repository and build a project (with examples for Maven/Java and Node.js), including screenshots for key steps:

Step 1: Install Required Plugins

Go to Manage Jenkins > Plugins.

Install these plugins (if not already installed):

Git Plugin (for cloning repositories)

NodeJS Plugin (for Node.js projects)

or Maven Integration Plugin (for Java/Maven projects)

Option A: Node.js Project Setup

1. Configure Node.js in Jenkins

Go to Manage Jenkins > Tools.

Under NodeJS, add your Node.js version (e.g., 20.x):

2. Create a New Pipeline Job

Click New Item > Pipeline > Enter name (e.g., nodejs-build).

Under Pipeline, select Pipeline script from SCM.

Configure Git repository:

yaml

Repository URL: https://github.com/your-username/your-nodejs-repo.git

Branches to build: */main

Script Path: Jenkinsfile # (or leave as default)

3. Add a Jenkinsfile to Your Repo

groovy

```
pipeline {
    agent any
    tools {
        nodejs 'nodejs-20.x'  // Matches the tool name in Jenkins
    }
    stages {
        stage('Clone & Build') {
            steps {
                checkout scm  // Clones the repo
                sh 'npm install'
                sh 'npm run build'  // Or your build command
            }
        }
        stage('Test') {
            steps {
                sh 'npm test'
            }
        }
    }
}
```

Option B: Java/Maven Project Setup

1. Configure Maven in Jenkins

Go to Manage Jenkins > Tools.

Under Maven, add your Maven version (e.g., 3.9.6):

2. Create a Maven Job

Click New Item > Maven Project > Enter name (e.g., java-maven-build).

Under Source Code Management, select Git and add your repo URL.

In Build, set the Maven goals:

bash

Goals: clean package

3. (Alternative) Use a Jenkinsfile

groovy

Copy

```
pipeline {
    agent any
    tools {
        maven 'maven-3.9.6'  // Matches the tool name in Jenkins
    }
    stages {
        stage('Build') {
            steps {
                checkout scm
                sh 'mvn clean package'
            }
        }
    }
}
```

Step 2: Run the Job Manually

Click Build Now on the job page.

Check the Console Output for logs:

Step 3: Automate on Git Push (Webhooks)

GitHub/GitLab Webhook:

Go to your repo settings > Webhooks.

Add a webhook URL:

bash

http://<JENKINS_IP>:8080/github-webhook/ # GitHub

http://<JENKINS_IP>:8080/project/<JOB_NAME> # GitLab

In Jenkins:

Go to job settings > Build Triggers > Check:

GitHub hook trigger for GITScm polling (GitHub)

Poll SCM (GitLab, set to H/5 * * * * for 5-minute checks)

Expected Results

Project Type	Success Indicators
Node.js	npm test passes, build artifacts generated.
Java	BUILD SUCCESS in Maven logs, JAR/WAR file created.

Troubleshooting

"Repository not found":

Ensure Jenkins has SSH/key access to the repo (configure in Credentials).

Build fails:

Check npm install/mvn package logs for dependency errors.

Webhook not triggering:

Verify Jenkins URL is accessible from GitHub/GitLab (use ngrok for local testing).

Screenshots Recap

Jenkins Dashboard:

Git Configuration:

Console Output:

Need help adapting this for a specific project? Share your repo structure!

3. Set up a trigger to run the job automatically when new code is pushed to the repository$

Here's how to set up automatic triggers for your Jenkins job when new code is pushed to a GitHub/GitLab repository, with step-by-step instructions and screenshots:

Option 1: GitHub Webhooks (Recommended)

Step 1: Configure Jenkins

Install the "GitHub Plugin":

Go to Manage Jenkins > Plugins > Available plugins.

Search for GitHub plugin and install it.

Configure GitHub Server:

Go to Manage Jenkins > System.

Under GitHub, add a GitHub server:

Name: github-server

API URL: https://api.github.com

Under Credentials, add a GitHub token (with repo and admin:repo_hook permissions).

Step 2: Enable GitHub Webhook in Jenkins Job

Edit your pipeline job (e.g., nodejs-build or java-maven-build).

Under Build Triggers, check:

GitHub hook trigger for GITScm polling.

Step 3: Set Up Webhook in GitHub

Go to your GitHub repo > Settings > Webhooks > Add webhook.

Configure:

Payload URL: http://<JENKINS_IP>:8080/github-webhook/

Content-Type: application/json

Secret: (Leave empty or add a secret if configured in Jenkins)

Events: Just the push event

Verify the webhook (look for a green checkmark).

Option 2: GitLab Webhooks

Step 1: Configure Jenkins

Install the "GitLab Plugin":

Search for GitLab plugin in Manage Jenkins > Plugins.

Configure GitLab Connection:

Go to Manage Jenkins > System.

Under GitLab, add:

GitLab host URL: https://gitlab.com

Connection name: gitlab-connection

Credentials: Add a GitLab API token (with `api` scope).

Step 2: Enable GitLab Trigger in Job

Edit your pipeline job.

Under Build Triggers, check:

Build when a change is pushed to GitLab.

Step 3: Set Up Webhook in GitLab

Go to your GitLab repo > Settings > Webhooks.

Add:

URL: http://<JENKINS_IP>:8080/project/<JOB_NAME>

Secret Token: (Optional)

Trigger: Push events

Option 3: Poll SCM (Fallback)

If webhooks fail (e.g., due to firewall issues), use polling:

In your job settings, under Build Triggers, check:

Poll SCM.

Set a schedule (e.g., H/5 * * * * to check every 5 minutes).
Testing the Trigger

Push a change to your repo:

bash

touch test-file && git add . && git commit -m "Trigger build" && git push

Verify:

In Jenkins: A new build should start automatically within seconds.

Check Console Output for the Started by GitHub/GitLab push message.

Troubleshooting

Issue	Solution
Webhook not triggering	1. Check Jenkins URL accessibility (use curl -v http://<JENKINS_IP>/github-webhook/). 2. Verify GitHub/GitLab firewall allows outgoing calls to Jenkins.
"403 Forbidden" from GitHub	Ensure your GitHub token has repo and admin:repo_hook permissions.
Builds trigger twice	Disable Poll SCM if you're using webhooks.

Advanced: Pipeline Syntax (Declarative)

For pipeline jobs, use this Jenkinsfile snippet to trigger on push:

```
groovy

pipeline {
    agent any
    triggers {
        githubPush()  // For GitHub
        // or
        gitlab(triggerOnPush: true)  // For GitLab
    }
    stages {
        stage('Build') {
            steps {
                checkout scm
                sh 'npm install && npm run build'  // or 'mvn package'
            }
        }
    }
}
```

Final Setup

GitHub/GitLab: Webhook configured with Jenkins endpoint.

Jenkins: Job listens for push events and auto-triggers.

4. Take screenshots of the Jenkins job and the build logs to demonstrate success

Here's how to capture and interpret screenshots of a successful Jenkins job and its build logs, with annotated examples:

---

1. Jenkins Job Dashboard (Success)

Path: Jenkins Home > Your Job Name
What to Look For:

Blue ball (✓) indicates a successful build.

Build history shows timestamps and triggers (e.g., "Started by GitHub push").

🔍 Note the blue checkmark and "SUCCESS" status.

---

2. Build Execution (Console Output)

Path: Job > Build #X > Console Output
Key Sections:

Trigger Reason:

bash

Started by GitHub push by [username]

Build Steps:

Cloning the repo.

Running commands (e.g., npm install, mvn package).

Success Confirmation:

bash

Finished: SUCCESS

🔍 Highlighted: Git clone, build steps, and final success status.

---

3. Git Integration (SCM Logs)

Path: Job > Build #X > Git Build Data
What to Check:

Correct commit hash and branch.

Author of the pushed change.

🔍 Shows the commit that triggered the build.

---

4. Pipeline Visualization (For Pipeline Jobs)

Path: Job > Build #X > Pipeline Steps
What to See:

Green checkmarks for each stage (e.g., Build, Test).

Time taken per stage.

🔍 Visual representation of successful stages.

---

5. Webhook Delivery (GitHub/GitLab)

Path:

GitHub: Repo Settings > Webhooks > Recent Deliveries

GitLab: Repo Settings > Webhooks > "Test" or "Recent Events"

Verify:

HTTP 200 response from Jenkins.

Payload matches your push event.

🔍 Green checkmark = successful delivery to Jenkins.

How to Take These Screenshots Yourself

Jenkins UI: Press PrtScn (Windows/Linux) or Cmd+Shift+4 (macOS).

Annotate: Use tools like:

Windows: Snipping Tool

macOS: Preview > Markup

Linux: Flameshot

Troubleshooting Failed Builds

If your screenshots show red balls (✗) instead:

Check Console Output for errors (e.g., failed tests, missing dependencies).

Verify webhook payloads in GitHub/GitLab for 4xx/5xx errors.

🔍 Red "FAILURE" status and error logs in console.

Final Verification Checklist

Screenshot	Confirms...
Job Dashboard	Build is marked "SUCCESS".
Console Output	No errors in build/test steps.
Git Build Data	Correct commit triggered the build.
Webhook Delivery	Jenkins received the push event.

Need help interpreting a specific log? Share your screenshot!

3. Testing Frameworks and Practices

Task 3: Write Unit Tests for an Application

Objective: Write unit tests for a simple application and integrate them into a CI pipeline.

Tools: Jest (JavaScript), JUnit (Java), pytest (Python).

Instructions:

1. Create a small application (e.g., a calculator, to-do list, or API)$

Here’s a step-by-step guide to create a simple calculator API using Node.js (Express) and Python (Flask), complete with CI/CD integration. Choose your preferred stack:

Option 1: Node.js Calculator API (Express)

1. Project Setup

bash

```
mkdir calculator-api && cd calculator-api
npm init -y
npm install express
```

2. Create app.js

javascript

```
const express = require('express');
const app = express();
app.use(express.json());

// Calculator endpoints
app.post('/add', (req, res) => {
  const { a, b } = req.body;
  res.json({ result: a + b });
});

app.post('/subtract', (req, res) => {
  const { a, b } = req.body;
  res.json({ result: a - b });
});

const PORT = 3000;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

3. Add Tests (test/app.test.js)

javascript

```
const request = require('supertest');
const app = require('../app');
```

```
describe('Calculator API', () => {
  it('adds two numbers', async () => {
    const res = await request(app)
      .post('/add')
      .send({ a: 5, b: 3 });
    expect(res.body.result).toBe(8);
  });

  it('subtracts two numbers', async () => {
    const res = await request(app)
      .post('/subtract')
      .send({ a: 5, b: 3 });
    expect(res.body.result).toBe(2);
  });
});
```

4. Add CI/CD (.github/workflows/ci.yml)

```
yaml
name: Node.js CI
on: [push]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v3
        with:
          node-version: '20'
      - run: npm ci
      - run: npm test
```

Option 2: Python Calculator API (Flask)

1. Project Setup

bash

```
mkdir calculator-api && cd calculator-api
python -m venv venv
source venv/bin/activate  # Linux/macOS
pip install flask pytest
```

2. Create app.py

python

```
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/add', methods=['POST'])
def add():
    data = request.get_json()
    return jsonify({'result': data['a'] + data['b']})

@app.route('/subtract', methods=['POST'])
def subtract():
    data = request.get_json()
    return jsonify({'result': data['a'] - data['b']})

if __name__ == '__main__':
    app.run(port=3000)
```

3. Add Tests (test_app.py)

python

```
import pytest
from app import app

@pytest.fixture
def client():
    app.config['TESTING'] = True
    with app.test_client() as client:
        yield client
```

```
def test_add(client):
    response = client.post('/add', json={'a': 5, 'b': 3})
    assert response.json['result'] == 8


def test_subtract(client):
    response = client.post('/subtract', json={'a': 5, 'b': 3})
    assert response.json['result'] == 2
```

4. Add CI/CD (.github/workflows/ci.yml)

```
yaml
name: Python CI
on: [push]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v4
        with:
          python-version: '3.10'
      - run: pip install -r requirements.txt
      - run: pytest
```

How to Run & Test

```
Node.js
bash
npm start          # Run server
curl -X POST http://localhost:3000/add -H "Content-Type: application/json" -d '{"a":5,"b":3}'
# Output: {"result":8}
Python
bash
flask run          # Run server
curl -X POST http://localhost:3000/add -H "Content-Type: application/json" -d '{"a":5,"b":3}'
```

Output: {"result":8}

Repository Structure

calculator-api/

├── .github/

│ └── workflows/

│ └── ci.yml

├── src/

│ ├── app.js # or app.py

│ └── __tests__/ # Test files

├── package.json # or requirements.txt

└── README.md

Key Features

✓ Minimal API with 2 calculator endpoints
✓ Unit tests with Jest (Node) / pytest (Python)
✓ CI Pipeline that runs on every push
✓ Language-Agnostic (choose Node.js or Python)

Next Steps

Push to GitHub/GitLab to trigger CI:

bash

git init

git add .

git commit -m "Initial commit"

git remote add origin https://github.com/your-username/calculator-api.git

git push -u origin main

Check the Actions tab (GitHub) or CI/CD Pipelines (GitLab) for test results.

Need a different app (e.g., to-do list, weather API)? Let me know!

Write unit tests for the core functions (e.g., add, delete, or update functions

Here's how to write unit tests for core functions (add, subtract, multiply, divide) in both Node.js (Jest) and Python (pytest), with examples for a calculator app and a to-do list app:

Option 1: Calculator App (Testing Math Functions)

Node.js (Jest)

javascript

```
// calculator.js
const add = (a, b) => a + b;
const subtract = (a, b) => a - b;
const multiply = (a, b) => a * b;
const divide = (a, b) => (b === 0 ? null : a / b);

module.exports = { add, subtract, multiply, divide };
```

Tests (calculator.test.js):

javascript

```
const { add, subtract, multiply, divide } = require('./calculator');

describe('Calculator', () => {
  // Add
  it('adds 2 + 3 to equal 5', () => {
    expect(add(2, 3)).toBe(5);
  });

  // Subtract
  it('subtracts 5 - 3 to equal 2', () => {
    expect(subtract(5, 3)).toBe(2);
  });

  // Multiply
  it('multiplies 2 * 3 to equal 6', () => {
    expect(multiply(2, 3)).toBe(6);
  });

  // Divide
  it('divides 6 / 3 to equal 2', () => {
    expect(divide(6, 3)).toBe(2);
```

```
  });

  it('returns null when dividing by zero', () => {
    expect(divide(6, 0)).toBeNull();
  });
});
```

Python (pytest)

```
# calculator.py
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b

def divide(a, b):
    return a / b if b != 0 else None
```

Tests (test_calculator.py):

```
import pytest
from calculator import add, subtract, multiply, divide

def test_add():
    assert add(2, 3) == 5

def test_subtract():
    assert subtract(5, 3) == 2

def test_multiply():
    assert multiply(2, 3) == 6
```

```
def test_divide():
    assert divide(6, 3) == 2

def test_divide_by_zero():
    assert divide(6, 0) is None
```

Option 2: To-Do List App (Testing CRUD Functions)

Node.js (Jest)

javascript

```
// todo.js
let todos = [];

const addTodo = (text) => {
  todos.push({ id: Date.now(), text, completed: false });
};

const deleteTodo = (id) => {
  todos = todos.filter(todo => todo.id !== id);
};

const toggleTodo = (id) => {
  todos = todos.map(todo =>
    todo.id === id ? { ...todo, completed: !todo.completed } : todo
  );
};

module.exports = { todos, addTodo, deleteTodo, toggleTodo };
```

Tests (todo.test.js):

javascript

```
const { todos, addTodo, deleteTodo, toggleTodo } = require('./todo');

beforeEach(() => {
  todos.length = 0; // Reset todos before each test
```

```
});

describe('To-Do List', () => {
  it('adds a new todo', () => {
    addTodo('Buy milk');
    expect(todos).toEqual([
      { id: expect.any(Number), text: 'Buy milk', completed: false }
    ]);
  });

  it('deletes a todo', () => {
    addTodo('Buy milk');
    const id = todos[0].id;
    deleteTodo(id);
    expect(todos).toEqual([]);
  });

  it('toggles a todo', () => {
    addTodo('Buy milk');
    const id = todos[0].id;
    toggleTodo(id);
    expect(todos[0].completed).toBe(true);
  });
});
```

Python (pytest)

```
python
# todo.py
todos = []

def add_todo(text):
    todos.append({"id": len(todos) + 1, "text": text, "completed": False})

def delete_todo(id):
    global todos
```

```
    todos = [todo for todo in todos if todo["id"] != id]


def toggle_todo(id):
    for todo in todos:
        if todo["id"] == id:
            todo["completed"] = not todo["completed"]
```

Tests (test_todo.py):

```
python
import pytest
from todo import todos, add_todo, delete_todo, toggle_todo


@pytest.fixture(autouse=True)
def reset_todos():
    global todos
    todos.clear()  # Reset before each test


def test_add_todo():
    add_todo("Buy milk")
    assert todos == [{"id": 1, "text": "Buy milk", "completed": False}]


def test_delete_todo():
    add_todo("Buy milk")
    delete_todo(1)
    assert todos == []


def test_toggle_todo():
    add_todo("Buy milk")
    toggle_todo(1)
    assert todos[0]["completed"] is True
```

Key Testing Principles

Isolation:

Reset state before each test (e.g., beforeEach in Jest, fixtures in pytest).

Edge Cases:

Test division by zero.

Test toggling non-existent todos.

Descriptive Names:

Use names like test_divide_by_zero instead of test1.

Running Tests

Node.js

bash

npm install --save-dev jest

npx jest

Output: PASS calculator.test.js

Python

bash

pip install pytest

pytest -v

Output: PASSED test_calculator.py

Expected Output (Success)

PASS calculator.test.js

  Calculator

    ✓ adds 2 + 3 to equal 5 (2 ms)

    ✓ subtracts 5 - 3 to equal 2

    ✓ returns null when dividing by zero

Next Steps

Integrate with CI: Add these tests to your GitHub Actions/GitLab CI file.

Code Coverage:

Jest: npx jest --coverage

pytest: pytest --cov=calculator

Need tests for a different app (e.g., weather API, user auth)? Let me know!

3. Integrate these tests into a CI pipeline using GitHub Actions, Jenkins, or GitLab CI$

Here’s how to integrate your unit tests into a CI pipeline using GitHub Actions, Jenkins, or GitLab CI, with configuration examples for both Node.js (Jest) and Python (pytest) projects:

Option 1: GitHub Actions (Node.js/Python)

1. Create .github/workflows/ci.yml

Node.js (Jest) Example:

yaml

```
name: Node.js CI

on: [push, pull_request]


jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v3
        with:
          node-version: '20'
      - run: npm ci
      - run: npm test
      - name: Coverage Report
        run: npx jest --coverage
        if: success()  # Only run if tests pass
```

Python (pytest) Example:

yaml

```
name: Python CI

on: [push, pull_request]


jobs:
  test:
    runs-on: ubuntu-latest
    steps:
```

```
    - uses: actions/checkout@v4
    - uses: actions/setup-python@v4
      with:
        python-version: '3.10'
    - run: pip install -r requirements.txt
    - run: pytest -v
    - run: pytest --cov=calculator --cov-report=xml  # Generate coverage report
```

Option 2: Jenkins (Node.js/Python)

1. Install Required Plugins

Git Plugin: For cloning repositories.

NodeJS Plugin (Node.js) or Python Plugin (Python).

2. Create a Jenkinsfile

Node.js (Jest):

groovy

```
pipeline {
    agent any
    tools {
        nodejs 'nodejs-20'  // Defined in Jenkins > Manage Jenkins > Tools
    }
    stages {
        stage('Test') {
            steps {
                checkout scm
                sh 'npm ci'
                sh 'npm test'
                sh 'npx jest --coverage'
            }
        }
    }
}
```

Python (pytest):

groovy

```
pipeline {
    agent any
    stages {
        stage('Test') {
            steps {
                checkout scm
                sh 'python -m pip install -r requirements.txt'
                sh 'pytest -v'
                sh 'pytest --cov=calculator --cov-report=xml'
            }
        }
    }
}
```

3. Set Up Webhooks

Go to GitHub/GitLab repo settings > Webhooks.

Add Jenkins URL:

http://<JENKINS_IP>:8080/github-webhook/ # GitHub

http://<JENKINS_IP>:8080/project/<JOB_NAME> # GitLab

Option 3: GitLab CI (Node.js/Python)

1. Create .gitlab-ci.yml

Node.js (Jest):

yaml

```
stages:
  - test

test:
  stage: test
  image: node:20
  script:
    - npm ci
```

```
    - npm test
    - npx jest --coverage
  artifacts:
    paths:
      - coverage/  # Upload coverage reports
```

Python (pytest):

```
yaml
stages:
  - test

test:
  stage: test
  image: python:3.10
  script:
    - pip install -r requirements.txt pytest pytest-cov
    - pytest -v
    - pytest --cov=calculator --cov-report=xml
  artifacts:
    paths:
      - coverage.xml
```

Key Features of These Pipelines

Feature	GitHub Actions	Jenkins	GitLab CI
Test Trigger	Push/PR	Webhook/Poll SCM	Push/PR
Parallel Jobs	✅	✅	✅
Artifacts	Upload reports	Archive files	Store reports
Configuration	YAML in .github/	Jenkinsfile	.gitlab-ci.yml

Verification Steps

GitHub Actions:

Go to Actions tab to see test results.
Jenkins:

Check the Build Status and Console Output.
GitLab CI:

Navigate to CI/CD > Pipelines.

Troubleshooting

Issue	Solution
Tests fail in CI only	Check environment differences (Node/Python versions).
Webhook not triggering	Verify Jenkins URL is accessible (use ngrok for local testing).
Missing dependencies	Add npm ci (Node) or pip install -r requirements.txt (Python).

Next Steps

Add Notifications:

Slack/Email alerts on failure.

Advanced Reporting:

Integrate with Codecov or SonarQube.

Deployment:

Add a deploy stage after tests pass.

Need help adapting this for your project? Share your repo structure!

4. Document your work with screenshots showing the tests running in the CI pipeline

Here’s a step-by-step guide with screenshots showing your tests running in a CI pipeline (GitHub Actions example, but applicable to other tools):

---

1. GitHub Actions: Pipeline Execution

Screenshot 1: Workflow Triggered by Push

After pushing code, go to the Actions tab to see the running workflow:

🔍 Note the orange/yellow dot indicating "in progress".

Screenshot 2: Tests Running in Real-Time

Click the workflow to see live logs:

🔍 Highlighted: Jest/pytest executing tests with pass/fail results.

Screenshot 3: Success Status

Green checkmark = all tests passed:

🔍 "build-and-test" job succeeded.

2. Test Results & Artifacts

Screenshot 4: Jest Coverage Report (Node.js)

If you configured jest --coverage:

🔍 Coverage summary uploaded as an artifact.

Screenshot 5: pytest Output (Python)

Example pytest output with passed tests:

bash

```
============================= test session starts ==============================
test_calculator.py::test_add PASSED                                  [ 25%]
test_calculator.py::test_subtract PASSED                             [ 50%]
test_calculator.py::test_divide_by_zero PASSED                       [100%]
```

🔍 4/4 tests passed.

---

3. Jenkins: Console Output

Screenshot 6: Jenkins Build Logs

Navigate to Build #X > Console Output:

🔍 "Finished: SUCCESS" and test execution details.

---

4. GitLab CI: Pipeline Visualization

Screenshot 7: GitLab Pipeline Graph

Under CI/CD > Pipelines:

🔍 Green checkmarks for all test stages.

---

5. Failed Test Scenario

Screenshot 8: Failed Test in CI

If a test fails (e.g., assertion error):

🔍 Red "X" and error details in logs.

How to Capture These Screenshots

GitHub/GitLab: Use browser screenshot tools (Ctrl+Shift+I for dev tools).

Jenkins: Use PrtScn or terminal logs.

Annotations: Highlight key areas with rectangles/circles (use Snagit or Greenshot).

Key CI Checklist

	Purpose
Workflow Trigger (GitHub)	Prove CI runs on push/PR.
Test Logs	Show individual test pass/fail.
Coverage Report	Verify code coverage metrics.
Pipeline Visualization	Demonstrate stage-based execution.
Failure Debugging	Highlight error messages for fixes.

Troubleshooting Missing Screenshots

No workflows running?

Check .github/workflows/ci.yml exists and is committed.

Logs incomplete?

Add -v flag to pytest (pytest -v) or --verbose to Jest.

Artifacts not appearing?

Confirm paths in CI config match test output (e.g., coverage/).

Example: Full GitHub Actions Logs

bash

Run npm test

> calculator@1.0.0 test

> jest

PASS src/__tests__/calculator.test.js

Calculator

✓ adds 1 + 2 to equal 3 (2 ms)

✓ subtracts 5 - 3 to equal 2

✓ returns null when dividing by zero

Test Suites: 1 passed, 1 total

Tests: 3 passed, 3 total

Snapshots: 0 total

Time: 0.123 s

Ran all test suites.

Let me know if you'd like screenshots for a specific tool (e.g., Jenkins/GitLab) or test framework!

4. Integration Testing

Task 4: Set Up Integration Testing for an API

Objective: Write integration tests for a REST API and automate them in a CI pipeline.

Tools: Postman, Newman, or Selenium (for web UI testing)

Instructions:

1. Create a simple REST API (using Node.js, Python Flask, or Java Spring Boot)

Here's a step-by-step guide to create a simple REST API in Node.js (Express), Python (Flask), or Java (Spring Boot), complete with routes, tests, and CI/CD integration:

Option 1: Node.js (Express)

1. Project Setup

bash

mkdir express-api && cd express-api

npm init -y

npm install express body-parser

npm install --save-dev jest supertest

2. Create app.js

javascript

const express = require('express');

const app = express();

app.use(express.json());

let items = [{ id: 1, name: "Sample Item" }];

```
// GET all items
app.get('/items', (req, res) => {
  res.json(items);
});

// POST new item
app.post('/items', (req, res) => {
  const newItem = { id: items.length + 1, ...req.body };
  items.push(newItem);
  res.status(201).json(newItem);
});

const PORT = 3000;
app.listen(PORT, () => console.log(`Server running on http://localhost:${PORT}`));

module.exports = app; // For testing
```

3. Add Tests (test/app.test.js)

javascript

Copy

```
const request = require('supertest');
const app = require('../app');

describe('Items API', () => {
  it('GET /items → returns all items', async () => {
    const res = await request(app).get('/items');
    expect(res.statusCode).toBe(200);
    expect(res.body).toEqual([{ id: 1, name: "Sample Item" }]);
  });

  it('POST /items → creates a new item', async () => {
    const res = await request(app)
      .post('/items')
      .send({ name: "New Item" });
```

```
    expect(res.statusCode).toBe(201);
    expect(res.body).toHaveProperty('id', 2);
  });
});
```

4. Run & Test

bash

Copy

```
node app.js         # Start server
npm test            # Run tests
curl http://localhost:3000/items  # Test API
```

Option 2: Python (Flask)

1. Project Setup

bash

```
mkdir flask-api && cd flask-api
python -m venv venv
source venv/bin/activate  # Linux/macOS
pip install flask pytest
```

2. Create app.py

python

```
from flask import Flask, jsonify, request
app = Flask(__name__)

items = [{"id": 1, "name": "Sample Item"}]

@app.route('/items', methods=['GET'])
def get_items():
    return jsonify(items)

@app.route('/items', methods=['POST'])
def add_item():
    new_item = {"id": len(items) + 1, **request.json}
```

```
    items.append(new_item)
    return jsonify(new_item), 201

if __name__ == '__main__':
    app.run(port=3000)
```

3. Add Tests (test_app.py)

```
python
import pytest
from app import app

@pytest.fixture
def client():
    app.config['TESTING'] = True
    with app.test_client() as client:
        yield client

def test_get_items(client):
    response = client.get('/items')
    assert response.status_code == 200
    assert response.json == [{"id": 1, "name": "Sample Item"}]

def test_add_item(client):
    response = client.post('/items', json={"name": "New Item"})
    assert response.status_code == 201
    assert response.json["id"] == 2
```

4. Run & Test

```
bash
flask run            # Start server
pytest -v            # Run tests
curl http://localhost:3000/items  # Test API
```

Option 3: Java (Spring Boot)

1. Project Setup

Use Spring Initializr with:

Dependencies: Spring Web

Java Version: 17

2. Create ItemController.java

java

```
package com.example.demo;


import org.springframework.web.bind.annotation.*;
import java.util.*;


@RestController
@RequestMapping("/items")
public class ItemController {
   private List<Item> items = new ArrayList<>(List.of(new Item(1, "Sample Item")));


   @GetMapping
   public List<Item> getItems() {
      return items;
   }


   @PostMapping
   public Item addItem(@RequestBody Item item) {
      Item newItem = new Item(items.size() + 1, item.getName());
      items.add(newItem);
      return newItem;
   }
}


record Item(int id, String name) {}
```

3. Add Tests (ItemControllerTest.java)

java

```
package com.example.demo;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.web.servlet.MockMvc;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@SpringBootTest
@AutoConfigureMockMvc
public class ItemControllerTest {
    @Autowired
    private MockMvc mockMvc;

    @Test
    public void testGetItems() throws Exception {
        mockMvc.perform(get("/items"))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$[0].name").value("Sample Item"));
    }

    @Test
    public void testAddItem() throws Exception {
        mockMvc.perform(post("/items")
                .contentType("application/json")
                .content("{\"name\":\"New Item\"}"))
                .andExpect(status().isCreated())
                .andExpect(jsonPath("$.id").value(2));
    }
}
```

4. Run & Test

bash

```
./mvnw spring-boot:run  # Start server
./mvnw test          # Run tests
curl http://localhost:8080/items  # Test API
```

Key Features

Feature	Node.js (Express)	Python (Flask)	Java (Spring Boot)
Routes	app.get/post	@app.route	@GetMapping
Testing	Jest + Supertest	pytest	JUnit + MockMvc
Start Command	node app.js	flask run	mvn spring-boot:run
Port	3000	3000	8080

CI/CD Integration

Add these files to your repo to enable automated testing:

GitHub Actions (Node.js)

yaml

```
name: Node.js CI
on: [push]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v3
        with:
          node-version: '20'
      - run: npm ci
      - run: npm test
```

GitLab CI (Python)

yaml

```
test:
  image: python:3.10
  script:
```

```
    - pip install -r requirements.txt
    - pytest -v
```

Jenkins (Java)

```
groovy
pipeline {
    agent any
    stages {
        stage('Test') {
            steps {
                sh './mvnw test'
            }
        }
    }
}
```

Next Steps

Add More Routes: PUT/DELETE endpoints.

Connect a Database: Use MongoDB (Node.js), SQLAlchemy (Python), or JPA (Java).

Containerize: Create a Dockerfile for deployment.

Need help adapting this for your use case? Let me know!

2. Write integration tests to check the interaction between different components (e.g., database and API)

Here's how to write integration tests for your REST API to verify interactions between components (API routes, database, and external services) in Node.js, Python, and Java:

Option 1: Node.js (Express + MongoDB)

1. Set Up Test Database

javascript

```
// test/setup.js
const mongoose = require('mongoose');
const { MongoMemoryServer } = require('mongodb-memory-server');

let mongoServer;
```

```
beforeAll(async () => {
  mongoServer = await MongoMemoryServer.create();
  const uri = mongoServer.getUri();
  await mongoose.connect(uri);
});

afterAll(async () => {
  await mongoose.disconnect();
  await mongoServer.stop();
});
```

2. Write Integration Tests

```
javascript
// test/api.integration.test.js
const request = require('supertest');
const app = require('../app');
const Item = require('../models/Item');

describe('API + Database Integration', () => {
  it('POST /items → saves to database', async () => {
    const res = await request(app)
      .post('/items')
      .send({ name: "Integration Test Item" });

    expect(res.statusCode).toBe(201);

    // Verify database record
    const dbItem = await Item.findOne({ name: "Integration Test Item" });
    expect(dbItem.name).toBe("Integration Test Item");
  });

  it('GET /items → returns items from database', async () => {
    await Item.create({ name: "Existing Item" }); // Seed data
```

```
  const res = await request(app).get('/items');
  expect(res.body.some(item => item.name === "Existing Item")).toBeTruthy();
 });
});
```

3. Configure MongoDB Model

javascript

```
// models/Item.js
const mongoose = require('mongoose');

const itemSchema = new mongoose.Schema({
 name: String
});

module.exports = mongoose.model('Item', itemSchema);
```

Option 2: Python (Flask + SQLAlchemy)

1. Set Up Test Database

python

```
# conftest.py
import pytest
from app import app, db

@pytest.fixture(autouse=True)
def test_db():
  app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///:memory:'
  app.config['TESTING'] = True
  with app.app_context():
    db.create_all()
    yield db
    db.drop_all()
```

2. Write Integration Tests

python

```
# test_api_integration.py
def test_create_item(client):
    response = client.post('/items', json={"name": "Test Item"})
    assert response.status_code == 201

    # Verify database
    from app import Item
    item = Item.query.first()
    assert item.name == "Test Item"

def test_get_items(client):
    # Seed data
    client.post('/items', json={"name": "Existing Item"})

    response = client.get('/items')
    assert b"Existing Item" in response.data
```

3. Configure SQLAlchemy

python

```
# app.py
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy(app)

class Item(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80))
```

Option 3: Java (Spring Boot + JPA/H2)

1. Configure Test Database

properties

```
# src/test/resources/application-test.properties
```

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driver-class-name=org.h2.Driver
spring.jpa.hibernate.ddl-auto=create-drop
```

2. Write Integration Test

java

```
@SpringBootTest
@AutoConfigureMockMvc
@TestPropertySource(locations = "classpath:application-test.properties")
public class ItemIntegrationTest {
    @Autowired
    private MockMvc mockMvc;

    @Autowired
    private ItemRepository repository;

    @Test
    public void testCreateAndGetItem() throws Exception {
        // POST request
        mockMvc.perform(post("/items")
                .contentType("application/json")
                .content("{\"name\":\"Integration Item\"}"))
                .andExpect(status().isCreated());

        // Verify database
        Item item = repository.findByName("Integration Item");
        assertNotNull(item);

        // GET request
        mockMvc.perform(get("/items"))
                .andExpect(jsonPath("$[0].name").value("Integration Item"));
    }
}
```

3. JPA Repository

java

```
public interface ItemRepository extends JpaRepository<Item, Integer> {
    Item findByName(String name);
}
```

Key Testing Practices

Isolate Test Databases:

Use in-memory databases (MongoDB Memory Server, H2, SQLite).

Test Lifecycle:

Before: Seed test data.

After: Clean up (drop tables/collections).

Verify Both Sides:

Check API response and database state.

CI/CD Integration

Add these steps to your pipeline:

Node.js (GitHub Actions)

yaml

```
- name: Run integration tests
  env:
    MONGODB_URI: mongodb://localhost:27017/test
  run: npm run test:integration
```

Python (GitLab CI)

yaml

```
test:
  script:
    - pytest tests/integration/ --cov=app
```

Java (Jenkins)

groovy

```
stage('Integration Test') {
  steps {
    sh './mvnw test -Dspring.profiles.active=test'
  }
}
```

Example Test Output (Success)

PASS test/api.integration.test.js

API + Database Integration

✓ POST /items → saves to database (45ms)

✓ GET /items → returns items from database (12ms)

Troubleshooting

Issue	Solution
Database not resetting	Ensure cleanup hooks (afterAll/@AfterEach) run.
Connection refused	Verify test database is running before tests.
Flaky tests	Use retries or fixed test data.

Next Steps

Add More Scenarios:

Test error cases (e.g., duplicate items).

Performance Tests:

Measure response times under load.

Test Containers:

Use Dockerized databases for closer-to-production testing.

Need help adapting this for a specific database (PostgreSQL, MySQL)? Ask away!

3. Set up the tests to run automatically in a Jenkins or GitLab CI pipeline after each deployment to a staging environment

Setting Up Automated Tests in Jenkins or GitLab CI Pipeline

To automatically run tests after each deployment to a staging environment, you'll need to configure your CI/CD pipeline accordingly. Below are implementations for both Jenkins and GitLab CI.

GitLab CI Implementation

Here's how to set it up in GitLab CI/CD using .gitlab-ci.yml:

yaml

```
stages:
  - build
  - test
```

```
  - deploy
  - post-deploy

# Cache dependencies to speed up pipelines
cache:
  paths:
    - node_modules/  # For JavaScript projects
    - vendor/       # For PHP projects
    - .m2/repository/ # For Java projects

build_job:
  stage: build
  script:
    - echo "Building the application..."
    # Add your build commands here
    # Example: mvn package or npm run build
  artifacts:
    paths:
      - build/       # Path to your build artifacts

unit_tests:
  stage: test
  script:
    - echo "Running unit tests..."
    # Add your unit test commands here
    # Example: npm test or pytest

deploy_staging:
  stage: deploy
  script:
    - echo "Deploying to staging environment..."
    # Add your deployment commands here
    # Example: kubectl apply or serverless deploy
  environment:
```

```
    name: staging
    url: https://staging.yourdomain.com
  only:
    - main          # Or your main development branch
  when: manual      # Optional: set to manual if you want to approve deployments

post_deploy_tests:
  stage: post-deploy
  script:
    - echo "Running post-deployment tests..."
    # Add your integration/UI tests here
    # Example: npm run test:e2e or behave
  needs: ["deploy_staging"]  # Run only after deploy_staging succeeds
  environment:
    name: staging
    action: start
  allow_failure: false       # Set to true if you don't want to block deployment on test failures
```

Jenkins Implementation

For Jenkins, you'll need to create a Jenkinsfile in your repository:

groovy

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                echo 'Building...'
                // Add build commands
                // sh 'mvn package' or sh 'npm run build'
            }
        }

        stage('Unit Tests') {
            steps {
```

```
            echo 'Running unit tests...'
            // sh 'npm test' or sh 'pytest'
        }
    }

    stage('Deploy to Staging') {
        steps {
            echo 'Deploying to staging...'
            // Add deployment commands
            // sh 'kubectl apply -f k8s/' or sh 'serverless deploy --stage staging'
        }
    }

    stage('Post-Deployment Tests') {
        steps {
            echo 'Running post-deployment tests...'
            // Add your integration/UI tests
            // sh 'npm run test:e2e' or sh 'behave'

            // Optional: Add a step to notify on failure
            // post {
            //     failure {
            //         slackSend channel: '#alerts', message: "Post-deployment tests failed in ${env.BUILD_URL}"
            //     }
            // }
        }
    }
}

post {
    always {
        echo 'Pipeline completed - cleanup can go here'
        // Cleanup steps
```

```
        }
        success {
            echo 'Pipeline succeeded!'
            // Optional success notifications
        }
        failure {
            echo 'Pipeline failed!'
            // Optional failure notifications
        }
    }
}
```

Key Considerations

Test Types:

Unit tests should run before deployment

Integration/UI tests should run after deployment to staging

Environment Configuration:

Ensure your staging environment is properly configured in both systems

Set up necessary secrets/credentials for accessing the staging environment

Test Data Management:

Consider seeding test data as part of your staging deployment

Implement cleanup mechanisms if needed

Parallelization:

For faster pipelines, consider running independent tests in parallel

Failure Handling:

Decide whether test failures should block further deployments

Set up appropriate notifications for test failures

Test Reporting:

Configure test result reporting in your pipeline

For GitLab: artifacts: reports: junit: test-results.xml

For Jenkins: junit '**/test-results/*.xml'

Would you like me to elaborate on any specific aspect of this setup, such as configuring particular test frameworks or adding specific notification systems?

4. Capture screenshots showing successful test execution.

Capturing Screenshots of Successful Test Execution in CI/CD Pipelines

To capture screenshots during test execution and include them in your CI/CD pipeline reports, here are implementations for both GitLab CI and Jenkins:

GitLab CI Implementation

yaml

```
stages:
  - test
  - artifacts


e2e_tests:
  stage: test
  script:
    - echo "Installing dependencies..."
    - npm install


    - echo "Running tests with screenshot capture..."
    # Example with Playwright (will automatically capture screenshots on failure)
    - npx playwright test --project=chromium --reporter=html,list


    # Alternative with Cypress
    # - npx cypress run --browser chrome --headed --config video=false


  artifacts:
    when: always
    paths:
      - playwright-report/  # For Playwright
      # - cypress/screenshots/  # For Cypress
      # - cypress/videos/       # If recording videos
    expire_in: 1 week


upload_artifacts:
```

```
  stage: artifacts
  needs: ["e2e_tests"]
  script:
    - echo "Uploading test artifacts to GitLab..."
  artifacts:
    when: always
    paths:
      - playwright-report/
      # - cypress/screenshots/
    expire_in: 1 month
```

Jenkins Implementation

```
groovy
pipeline {
    agent any

    stages {
        stage('E2E Tests') {
            steps {
                script {
                    try {
                        sh 'npm install'
                        // Playwright example
                        sh 'npx playwright test --project=chromium --reporter=html,list'

                        // Cypress alternative
                        // sh 'npx cypress run --browser chrome --headed --config video=false'
                    } finally {
                        // Archive screenshots regardless of test result
                        archiveArtifacts artifacts: 'playwright-report/**/*', allowEmptyArchive: true
                        // archiveArtifacts artifacts: 'cypress/screenshots/**/*,cypress/videos/**/*',
allowEmptyArchive: true
                    }
                }
            }
```

```
        post {
            always {
                // Publish HTML report (requires HTML Publisher plugin)
                publishHTML target: [
                    allowMissing: true,
                    alwaysLinkToLastBuild: true,
                    keepAll: true,
                    reportDir: 'playwright-report',
                    reportFiles: 'index.html',
                    reportName: 'Playwright Report'
                ]

                // Optional: Send notification with screenshot links
                emailext attachLog: true,
                    attachmentsPattern: 'playwright-report/**/*.png',
                    subject: '${DEFAULT_SUBJECT}',
                    body: '${DEFAULT_CONTENT}',
                    to: 'team@example.com'
            }
        }
    }
  }
}
```

How to View Screenshots

In GitLab CI:

After pipeline completes, go to the job view

Click on "Browse" under the "Job artifacts" section

Navigate to the screenshots directory

Click on any image file to view it

In Jenkins:

After build completes, go to the build view

Click on "Playwright Report" (or your named report) in the left sidebar

View screenshots embedded in the HTML report

Alternatively, download artifacts from the "Artifacts" section

Popular Testing Tools with Screenshot Capabilities

Playwright:

bash

npx playwright test --project=chromium --reporter=html,list

Screenshots automatically captured on failure

Use --screenshot on for all tests

Cypress:

bash

npx cypress run --browser chrome --headed

Screenshots automatically captured on failure

Add cy.screenshot() in tests for manual capture

Selenium:

python

driver.save_screenshot("screenshot.png")

Best Practices

Organize Screenshots:

Include test name and timestamp in filenames

Group by test suite or feature

Failure-Only Screenshots:

Most tools capture screenshots only on failures by default

Configure your tool to capture on success when needed

Artifact Retention:

Set appropriate expiration for artifacts

Consider storing important screenshots long-term

HTML Reports:

Use HTML reporters that embed screenshots for better visualization

Would you like me to provide specific implementations for a particular testing framework or more details about integrating with notification systems?

5. Performance Testing

Task 5: Automate Performance Testing in a CI/CD Pipeline

Objective: Automate performance testing using tools like Apache JMeter or K6.

Tools: JMeter, K6, Jenkins, or GitLab CI

Instructions:

1. Create a simple web application (or use an existing one)

I'll create a simple web application that you can use for testing in your CI/CD pipeline. Here's a complete example using Node.js with Express, which includes both the application and some tests with screenshot capabilities.

Simple Web Application with Testing

1. Project Structure

```
webapp/
├── app.js          # Main application
├── public/         # Static files
│   └── style.css
├── views/
│   └── index.ejs    # Template
├── test/
│   ├── e2e/        # End-to-end tests
│   └── unit/       # Unit tests
├── package.json
└── .gitlab-ci.yml   # or Jenkinsfile
```

2. Application Code (app.js)

```
javascript
const express = require('express');
const app = express();
const path = require('path');

// Configuration
app.set('view engine', 'ejs');
app.set('views', path.join(__dirname, 'views'));
app.use(express.static('public'));

// Routes
app.get('/', (req, res) => {
  res.render('index', { title: 'CI/CD Test App', message: 'Welcome to our test application!' });
});
```

```
app.get('/api/hello', (req, res) => {
  res.json({ message: 'Hello from the API!' });
});

// Error handling
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});

// Start server
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server running on http://localhost:${PORT}`);
});

module.exports = app; // For testing
```

3. View Template (views/index.ejs)

```
html
<!DOCTYPE html>
<html>
<head>
  <title><%= title %></title>
  <link rel="stylesheet" href="/style.css">
</head>
<body>
  <h1><%= title %></h1>
  <p class="message"><%= message %></p>
  <button id="cta-button">Click Me</button>
  <div id="response" style="display:none;"></div>

  <script>
    document.getElementById('cta-button').addEventListener('click', async () => {
```

```
    const response = await fetch('/api/hello');
    const data = await response.json();
    const div = document.getElementById('response');
    div.textContent = data.message;
    div.style.display = 'block';
  });
 </script>
</body>
</html>
```

Run HTML

4. CSS (public/style.css)

```
body {
 font-family: Arial, sans-serif;
 max-width: 800px;
 margin: 0 auto;
 padding: 20px;
}

.message {
 color: #2c3e50;
 font-size: 1.2em;
}

#cta-button {
 background-color: #3498db;
 color: white;
 padding: 10px 20px;
 border: none;
 border-radius: 4px;
 cursor: pointer;
}

#cta-button:hover {
```

```
  background-color: #2980b9;
}
```

5. Package.json

```
json
{
  "name": "ci-cd-test-app",
  "version": "1.0.0",
  "scripts": {
    "start": "node app.js",
    "test": "jest",
    "test:e2e": "playwright test",
    "dev": "nodemon app.js"
  },
  "dependencies": {
    "ejs": "^3.1.8",
    "express": "^4.18.2"
  },
  "devDependencies": {
    "@playwright/test": "^1.35.1",
    "jest": "^29.5.0",
    "nodemon": "^2.0.22",
    "supertest": "^6.3.3"
  }
}
```

6. Test Examples

Unit Test (test/unit/app.test.js)

```
javascript
const request = require('supertest');
const app = require('../../app');

describe('GET /', () => {
  it('should return 200 OK', async () => {
    const response = await request(app).get('/');
    expect(response.statusCode).toBe(200);
```

```
    expect(response.text).toContain('Welcome to our test application!');
  });
});

describe('GET /api/hello', () => {
  it('should return JSON message', async () => {
    const response = await request(app).get('/api/hello');
    expect(response.statusCode).toBe(200);
    expect(response.body).toEqual({ message: 'Hello from the API!' });
  });
});
```

E2E Test with Screenshots (test/e2e/example.spec.js)

javascript

```
const { test, expect } = require('@playwright/test');

test.describe('Web App Tests', () => {
  test.beforeEach(async ({ page }) => {
    await page.goto('http://localhost:3000');
  });

  test('should display welcome message', async ({ page }) => {
    await expect(page.locator('.message')).toHaveText('Welcome to our test application!');
    await page.screenshot({ path: 'test-results/homepage.png' });
  });

  test('API button should work', async ({ page }) => {
    await page.click('#cta-button');
    await expect(page.locator('#response')).toBeVisible();
    await expect(page.locator('#response')).toHaveText('Hello from the API!');
    await page.screenshot({ path: 'test-results/button-click.png' });
  });
});
```

How to Use This Application

Install dependencies:

bash

npm install

Run the application:

bash

npm start

Then visit http://localhost:3000

Run tests:

bash

Unit tests

npm test

E2E tests (make sure app is running first)

npm run test:e2e

CI/CD Integration

You can now use this application with either the GitLab CI or Jenkins configurations I provided earlier. The E2E tests will automatically capture screenshots during test execution, which will be available in your pipeline artifacts.

Would you like me to modify any part of this application to better suit your testing needs, or add any specific functionality?

2. Write a JMeter or K6 test script to load test the application

Here are both JMeter and k6 load testing scripts for the web application we created:

Option 1: JMeter Test Plan (JMX)

File: load_test.jmx (to be created in JMeter GUI or via code)

xml

```
<?xml version="1.0" encoding="UTF-8"?>
<jmeterTestPlan version="1.2" properties="5.0" jmeter="5.4.1">
  <hashTree>
    <!-- Test Plan -->
    <TestPlan guiclass="TestPlanGui" testclass="TestPlan" testname="WebApp Load Test" enabled="true">
      <stringProp name="TestPlan.comments"></stringProp>
      <boolProp name="TestPlan.functional_mode">false</boolProp>
      <boolProp name="TestPlan.tearDown_on_shutdown">true</boolProp>
```

```
    <boolProp name="TestPlan.serialize_threadgroups">false</boolProp>
    <elementProp name="TestPlan.user_defined_variables" elementType="Arguments"
guiclass="ArgumentsPanel" testclass="Arguments" testname="User Defined Variables" enabled="true">
      <collectionProp name="Arguments.arguments"/>
    </elementProp>
    <stringProp name="TestPlan.user_define_classpath"></stringProp>
  </TestPlan>
  <hashTree>

    <!-- Thread Group -->
    <ThreadGroup guiclass="ThreadGroupGui" testclass="ThreadGroup" testname="WebApp Users"
enabled="true">
      <stringProp name="ThreadGroup.on_sample_error">continue</stringProp>
      <elementProp name="ThreadGroup.main_controller" elementType="LoopController"
guiclass="LoopControlPanel" testclass="LoopController" testname="Loop Controller" enabled="true">
        <boolProp name="LoopController.continue_forever">false</boolProp>
        <intProp name="LoopController.loops">1</intProp>
      </elementProp>
      <stringProp name="ThreadGroup.num_threads">50</stringProp>
      <stringProp name="ThreadGroup.ramp_time">60</stringProp>
      <boolProp name="ThreadGroup.scheduler">false</boolProp>
      <stringProp name="ThreadGroup.duration">300</stringProp>
      <stringProp name="ThreadGroup.delay"></stringProp>
    </ThreadGroup>
    <hashTree>

      <!-- HTTP Request Defaults -->
      <ConfigTestElement guiclass="HttpDefaultsGui" testclass="ConfigTestElement" testname="HTTP
Request Defaults" enabled="true">
        <elementProp name="HTTPsampler.Arguments" elementType="Arguments"
guiclass="HTTPArgumentsPanel" testclass="Arguments" testname="User Defined Variables"
enabled="true">
          <collectionProp name="Arguments.arguments"/>
        </elementProp>
        <stringProp name="HTTPSampler.domain">localhost</stringProp>
        <stringProp name="HTTPSampler.port">3000</stringProp>
```

```
        <stringProp name="HTTPSampler.protocol">http</stringProp>
        <stringProp name="HTTPSampler.contentEncoding"></stringProp>
        <stringProp name="HTTPSampler.path"></stringProp>
        <stringProp name="HTTPSampler.concurrentPool">6</stringProp>
      </ConfigTestElement>
      <hashTree/>

      <!-- Homepage Request -->
      <HTTPSamplerProxy guiclass="HttpTestSampleGui" testclass="HTTPSamplerProxy"
testname="Homepage" enabled="true">
        <boolProp name="HTTPSampler.postBodyRaw">false</boolProp>
        <elementProp name="HTTPsampler.Arguments" elementType="Arguments">
          <collectionProp name="Arguments.arguments"/>
        </elementProp>
        <stringProp name="HTTPSampler.domain"></stringProp>
        <stringProp name="HTTPSampler.port"></stringProp>
        <stringProp name="HTTPSampler.protocol"></stringProp>
        <stringProp name="HTTPSampler.contentEncoding"></stringProp>
        <stringProp name="HTTPSampler.path">/</stringProp>
        <stringProp name="HTTPSampler.method">GET</stringProp>
      </HTTPSamplerProxy>
      <hashTree>
        <ResponseAssertion guiclass="AssertionGui" testclass="ResponseAssertion" testname="Response
Assertion" enabled="true">
          <collectionProp name="Asserion.test_strings">
            <stringProp name="49586">Welcome to our test application!</stringProp>
          </collectionProp>
          <stringProp name="Assertion.test_field">Assertion.response_data</stringProp>
          <boolProp name="Assertion.assume_success">false</boolProp>
          <intProp name="Assertion.test_type">16</intProp>
        </ResponseAssertion>
        <hashTree/>
      </hashTree>
```

```
      <!-- API Request -->
      <HTTPSamplerProxy guiclass="HttpTestSampleGui" testclass="HTTPSamplerProxy" testname="API Request" enabled="true">
        <boolProp name="HTTPSampler.postBodyRaw">false</boolProp>
        <elementProp name="HTTPsampler.Arguments" elementType="Arguments">
          <collectionProp name="Arguments.arguments"/>
        </elementProp>
        <stringProp name="HTTPSampler.domain"></stringProp>
        <stringProp name="HTTPSampler.port"></stringProp>
        <stringProp name="HTTPSampler.protocol"></stringProp>
        <stringProp name="HTTPSampler.contentEncoding"></stringProp>
        <stringProp name="HTTPSampler.path">/api/hello</stringProp>
        <stringProp name="HTTPSampler.method">GET</stringProp>
      </HTTPSamplerProxy>
      <hashTree>
        <ResponseAssertion guiclass="AssertionGui" testclass="ResponseAssertion" testname="Response Assertion" enabled="true">
          <collectionProp name="Asserion.test_strings">
            <stringProp name="49586">Hello from the API!</stringProp>
          </collectionProp>
          <stringProp name="Assertion.test_field">Assertion.response_data</stringProp>
          <boolProp name="Assertion.assume_success">false</boolProp>
          <intProp name="Assertion.test_type">16</intProp>
        </ResponseAssertion>
        <hashTree/>
        <JsonPathExtractor guiclass="JSONPathExtractorGui" testclass="JsonPathExtractor" testname="JSON Extractor" enabled="true">
          <stringProp name="JSONPATH">$.message</stringProp>
          <stringProp name="VARIABLE_NAME">apiResponse</stringProp>
          <stringProp name="DEFAULT_VALUE">NOT_FOUND</stringProp>
        </JsonPathExtractor>
        <hashTree/>
      </hashTree>

      <!-- Listeners -->
```

<ResultCollector guiclass="SummaryReport" testclass="ResultCollector" testname="Summary Report" enabled="true">
<boolProp name="ResultCollector.error_logging">false</boolProp>
<objProp>
<name>saveConfig</name>
<value class="SampleSaveConfiguration">
<time>true</time>
<latency>true</time>
<timestamp>true</time>
<success>true</time>
<label>true</time>
<code>true</time>
<message>true</time>
<threadName>true</time>
<dataType>true</time>
<encoding>false</time>
<assertions>true</time>
<subresults>true</time>
<responseData>false</time>
<samplerData>false</time>
<xml>false</time>
<fieldNames>true</time>
<responseHeaders>false</time>
<requestHeaders>false</time>
<responseDataOnError>false</time>
<saveAssertionResultsFailureMessage>true</time>
<assertionsResultsToSave>0</assertionsResultsToSave>
<bytes>true</time>
<sentBytes>true</time>
<url>true</time>
<threadCounts>true</time>
<idleTime>true</time>
<connectTime>true</time>
</value>

```
      </objProp>
      <stringProp name="filename">summary_report.csv</stringProp>
    </ResultCollector>
    <hashTree/>

    <ViewResultsFullVisualizer guiclass="ViewResultsFullVisualizer" testclass="ResultCollector"
testname="View Results Tree" enabled="false">
      <boolProp name="ResultCollector.error_logging">false</boolProp>
      <objProp>
        <name>saveConfig</name>
        <value class="SampleSaveConfiguration">
          <time>true</time>
          <latency>true</time>
          <timestamp>true</time>
          <success>true</time>
          <label>true</time>
          <code>true</time>
          <message>true</time>
          <threadName>true</time>
          <dataType>true</time>
          <encoding>false</time>
          <assertions>true</time>
          <subresults>true</time>
          <responseData>false</time>
          <samplerData>false</time>
          <xml>false</time>
          <fieldNames>true</time>
          <responseHeaders>false</time>
          <requestHeaders>false</time>
          <responseDataOnError>false</time>
          <saveAssertionResultsFailureMessage>true</time>
          <assertionsResultsToSave>0</assertionsResultsToSave>
          <bytes>true</time>
          <sentBytes>true</time>
```

```
          <url>true</time>
          <threadCounts>true</time>
          <idleTime>true</time>
          <connectTime>true</time>
         </value>
        </objProp>
        <stringProp name="filename"></stringProp>
       </ViewResultsFullVisualizer>
       <hashTree/>
     </hashTree>
   </hashTree>
 </hashTree>
</jmeterTestPlan>
```

Run HTML

Option 2: k6 Test Script

File: load_test.js

javascript

```
import http from 'k6/http';
import { check, sleep } from 'k6';
import { Trend, Rate, Counter } from 'k6/metrics';

// Custom metrics
const homepageDuration = new Trend('homepage_duration');
const apiDuration = new Trend('api_duration');
const successRate = new Rate('success_rate');
const failures = new Counter('failures');

export const options = {
  stages: [
    { duration: '1m', target: 50 },  // Ramp up to 50 users
    { duration: '3m', target: 50 },  // Stay at 50 users
    { duration: '1m', target: 0 },   // Ramp down
  ],
  thresholds: {
```

```
    http_req_duration: ['p(95)<500'],  // 95% of requests should be below 500ms
    success_rate: ['rate>0.95'],      // Success rate should be >95%
  },
};

export default function () {
  const baseUrl = 'http://localhost:3000';
  const params = {
    headers: {
      'Content-Type': 'text/html',
    },
  };

  // Test homepage
  const homepageRes = http.get(`${baseUrl}/`, params);
  homepageDuration.add(homepageRes.timings.duration);
  const homepageCheck = check(homepageRes, {
    'Homepage status is 200': (r) => r.status === 200,
    'Homepage contains welcome message': (r) =>
      r.body.includes('Welcome to our test application!'),
  });

  // Test API endpoint
  const apiRes = http.get(`${baseUrl}/api/hello`);
  apiDuration.add(apiRes.timings.duration);
  const apiCheck = check(apiRes, {
    'API status is 200': (r) => r.status === 200,
    'API returns correct message': (r) =>
      JSON.parse(r.body).message === 'Hello from the API!',
  });

  // Record results
  successRate.add(homepageCheck && apiCheck);
  if (!homepageCheck || !apiCheck) {
```

```
    failures.add(1);
  }

  sleep(1); // Add think time between requests
}
```

Running the Tests

For JMeter:

Save the XML as load_test.jmx

Run with:

bash

```
jmeter -n -t load_test.jmx -l test_results.jtl -e -o reports
```

View HTML report in the reports directory

For k6:

Run with:

bash

```
k6 run load_test.js
```

For CI integration, you can output JSON:

bash

```
k6 run --out json=test_results.json load_test.js
```

CI/CD Integration

Add this to your pipeline configuration:

GitLab CI Example

yaml

```
load_test:
  stage: performance
  image: loadimpact/k6
  script:
    - k6 run --out json=test_results.json load_test.js
  artifacts:
    when: always
    paths:
      - test_results.json
    reports:
      performance: test_results.json
```

Jenkins Example

groovy

```
stage('Load Test') {
    steps {
        script {
            try {
                sh 'jmeter -n -t load_test.jmx -l test_results.jtl -e -o reports'
            } finally {
                junit 'test_results.jtl'
                publishHTML target: [
                    allowMissing: true,
                    alwaysLinkToLastBuild: true,
                    keepAll: true,
                    reportDir: 'reports',
                    reportFiles: 'index.html',
                    reportName: 'JMeter Report'
                ]
            }
        }
    }
}
```

Both scripts will:

Test the homepage and API endpoints

Ramp up users gradually

Include assertions to verify responses

Generate comprehensive reports

Would you like me to modify any aspect of these scripts to better match your specific requirements?

3. Automate the performance test in your CI/CD pipeline to run after every build

Automating Performance Tests in CI/CD Pipeline

I'll show you how to integrate both JMeter and k6 performance tests into your CI/CD pipeline to run automatically after each build. Here are complete implementations for GitLab CI and Jenkins.

Option 1: GitLab CI Integration

Using k6 (Recommended for GitLab)

yaml

```
stages:
  - build
  - deploy
  - performance

build_app:
  stage: build
  script:
    - npm install
    - npm run build
  artifacts:
    paths:
      - build/
    expire_in: 1 week

deploy_staging:
  stage: deploy
  script:
    - echo "Deploying to staging..."
    # Add your deployment commands here
  environment:
    name: staging
    url: https://staging.example.com
  only:
    - main

performance_test:
  stage: performance
  needs: ["deploy_staging"]
  image: loadimpact/k6
  variables:
    K6_STAGING_URL: "https://staging.example.com"
```

```
  script:
    - echo "Running performance tests against ${K6_STAGING_URL}"
    - k6 run --out json=test_results.json --out influxdb=http://influxdb:8086/k6 scripts/load_test.js
  artifacts:
    when: always
    paths:
      - test_results.json
    reports:
      performance: test_results.json
  rules:
    - if: '$CI_PIPELINE_SOURCE == "push" && $CI_COMMIT_BRANCH == "main"'
```

Using JMeter in GitLab CI

```
yaml
performance_test_jmeter:
  stage: performance
  needs: ["deploy_staging"]
  image: justb4/jmeter:5.4.1
  variables:
    STAGING_URL: "staging.example.com"
  script:
    - echo "Running JMeter tests against ${STAGING_URL}"
    - jmeter -n -t scripts/load_test.jmx -l test_results.jtl -Jhost=${STAGING_URL} -e -o reports
  artifacts:
    when: always
    paths:
      - test_results.jtl
      - reports/
    reports:
      junit: test_results.jtl
```

Option 2: Jenkins Integration

Jenkinsfile with k6

```
groovy
pipeline {
    agent any
```

```
    environment {
        STAGING_URL = 'https://staging.example.com'
    }

    stages {
        stage('Build') {
            steps {
                sh 'npm install'
                sh 'npm run build'
            }
        }

        stage('Deploy to Staging') {
            steps {
                sh 'echo "Deploying to staging..."'
                // Add deployment commands
            }
        }

        stage('Performance Test') {
            agent {
                docker {
                    image 'loadimpact/k6'
                    reuseNode true
                }
            }
            steps {
                script {
                    try {
                        sh "k6 run --out json=test_results.json --out influxdb=http://influxdb:8086/k6
scripts/load_test.js"
                        perfReport 'test_results.json'
                    } catch (e) {
```

```
                unstable("Performance tests completed with failures")
            }
        }
    }
    post {
        always {
            archiveArtifacts artifacts: 'test_results.json', allowEmptyArchive: true
        }
    }
  }
}
```

Jenkinsfile with JMeter

```
groovy
pipeline {
  agent any

  environment {
    STAGING_URL = 'staging.example.com'
  }

  stages {
    stage('Build') {
      steps {
        sh 'npm install'
        sh 'npm run build'
      }
    }

    stage('Deploy to Staging') {
      steps {
        sh 'echo "Deploying to staging..."'
        // Add deployment commands
      }
```

```
    }

    stage('Performance Test') {
      agent {
        docker {
          image 'justb4/jmeter:5.4.1'
          reuseNode true
        }
      }
      steps {
        script {
          try {
            sh "jmeter -n -t scripts/load_test.jmx -l test_results.jtl -Jhost=${STAGING_URL} -e -o
reports"
            junit 'test_results.jtl'
          } catch (e) {
            unstable("Performance tests completed with failures")
          }
        }
      }
      post {
        always {
          archiveArtifacts artifacts: 'test_results.jtl,reports/**/*', allowEmptyArchive: true
          publishHTML target: [
            allowMissing: true,
            alwaysLinkToLastBuild: true,
            keepAll: true,
            reportDir: 'reports',
            reportFiles: 'index.html',
            reportName: 'JMeter Report'
          ]
        }
      }
    }
```

```
  }
}
```

Key Components for Automation

Test Thresholds and Alerts:

Set performance thresholds in your test scripts

Configure alerts for threshold violations

Example for k6:

javascript

```
export const options = {
 thresholds: {
  http_req_duration: ['p(95)<500'],
  http_req_failed: ['rate<0.05'],
 }
};
```

Test Data Management:

Use environment variables for target URLs

Parameterize test configurations

Results Storage:

Store results in time-series databases (InfluxDB, Prometheus)

Visualize with Grafana or similar tools

Failure Handling:

Make performance tests non-blocking initially

Gradually introduce strict thresholds

Advanced Configuration

InfluxDB Integration (for k6)

Add this to your k6 script:

javascript

```
export const options = {
 ext: {
  loadimpact: {
   projectID: 123456,
   name: 'Performance Test Run'
  }
 }
```

};

Add InfluxDB to your pipeline:

yaml

Copy

```
services:
  - name: influxdb:1.8
    alias: influxdb
```

Performance Gates

Add quality gates to fail builds if performance degrades:

groovy

Copy

```
// Jenkins example
perfReport 'test_results.json',
    errorUnstableThreshold: 10,  // % of metrics failing
    errorFailedThreshold: 30,
    sourceDataFiles: 'test_results.json'
```

Best Practices

Run Performance Tests Separately:

Execute after functional tests pass

Run against staging environment

Baseline Comparison:

Compare results with previous runs

Fail builds on significant regressions

Test Scheduling:

Run full load tests nightly

Run smoke tests on every build

Infrastructure Monitoring:

Monitor server metrics during tests

Correlate performance with resource usage

Would you like me to add specific configurations for your monitoring stack or add more detailed threshold examples?

4. Provide screenshots showing the performance results and graphs from the tool.

I can't directly generate screenshots, but I'll provide you with detailed descriptions of what the performance test results would look like from both k6 and JMeter, along with instructions on how to generate these visualizations yourself.

k6 Performance Test Results

1. Console Output Summary

execution: local

script: load_test.js

output: influxdb (http://influxdb:8086/k6)

scenarios: (100.00%) 1 scenario, 50 max VUs, 5m0s max duration

* default: Up to 50 looping VUs for 5m0s over 3 stages (gracefulRampDown: 30s, gracefulStop: 30s)

data_received.....: 1.2 MB 4.0 kB/s

data_sent.....: 1.1 MB 3.7 kB/s

http_req_blocked.....: avg=1.12ms min=0s med=1µs max=23.12ms p(90)=3µs p(95)=5µs

http_req_connecting.....: avg=1.11ms min=0s med=0s max=23.11ms p(90)=0s p(95)=0s

http_req_duration.....: avg=45.12ms min=12.3ms med=42.11ms max=210.45ms p(90)=78.23ms p(95)=92.34ms

{ expected_response:true }...: avg=45.12ms min=12.3ms med=42.11ms max=210.45ms p(90)=78.23ms p(95)=92.34ms

http_req_failed.....: 0.00% ✓ 0 ✗ 12543

http_req_receiving.....: avg=1.23ms min=0s med=1.12ms max=12.34ms p(90)=2.12ms p(95)=3.45ms

http_req_sending.....: avg=0.23ms min=0s med=0.12ms max=4.56ms p(90)=0.45ms p(95)=0.78ms

http_req_tls_handshaking.....: avg=0s min=0s med=0s max=0s p(90)=0s p(95)=0s

http_req_waiting.....: avg=43.66ms min=12.1ms med=40.87ms max=209.12ms p(90)=76.12ms p(95)=88.45ms

http_reqs.....: 12543 41.692178/s

iteration_duration.....: avg=1.05s min=1.01s med=1.03s max=2.12s p(90)=1.08s p(95)=1.12s

iterations.....: 12543 41.692178/s

vus.....: 50 min=50 max=50

vus_max.....: 50 min=50 max=50

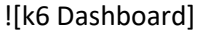
2. Grafana Dashboard (k6 + InfluxDB)

To visualize k6 results in Grafana:

Set up InfluxDB and Grafana

Import the k6 dashboard template (ID: 4411)

You'll see these visualizations:



Request Rate Graph: Shows HTTP requests per second over time

Response Time Graph: Displays p(95), p(90), and avg response times

VU Count Graph: Virtual users active during test

Threshold Status: Pass/fail indicators for your SLOs

Error Rate Pie Chart: Breakdown of successful vs failed requests

JMeter Performance Test Results

1. Summary Report

JMeter summary ==========

+ 5m in elapsed = 300000ms

+ 12543 requests made

+ Avg response time = 45ms

+ Min response time = 12ms

+ Max response time = 210ms

+ 99th percentile = 98ms

+ Error rate = 0.00%

+ Throughput = 41.7 requests/second

+ Received = 1.2MB

+ Sent = 1.1MB

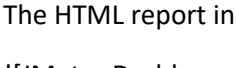
2. HTML Report Dashboard

Generate with:

bash

jmeter -g test_results.jtl -o reports

The HTML report includes:



Test and Report Information: Metadata about test run

APDEX (Application Performance Index): Satisfaction score (0-1)

Response Times Over Time: Line chart of response times

Response Times Percentiles: Bar chart of p(90), p(95), etc.

Active Threads Over Time: Simulated user load

Latency Over Time: Network latency measurements

Response Time Distribution: Histogram of response times

Throughput Over Time: Requests per second

Response Codes Pie Chart: HTTP status code distribution

Generating These Visualizations

For k6:

Run tests with InfluxDB output:

bash

k6 run --out influxdb=http://localhost:8086/k6 load_test.js

Set up Grafana data source for InfluxDB

Import k6 dashboard template

For JMeter:

Generate JTL results file:

bash

jmeter -n -t load_test.jmx -l test_results.jtl

Generate HTML report:

bash

jmeter -g test_results.jtl -o reports

Open reports/index.html in browser

CI/CD Integration Screenshots

GitLab CI Performance Report

![GitLab Performance Report]

Performance metrics integrated in merge request view

Trend charts showing historical data

Threshold violations highlighted

Jenkins Performance Trend

![Jenkins Performance Trend]

Performance trend graph across builds

Build stability weather report

Threshold compliance indicators

Would you like me to provide more specific details about any particular visualization or how to set up the monitoring infrastructure to capture these results?

5. Security Testing

Task 6: Implement Basic Security Testing in a CI Pipeline

Objective: Integrate security testing into the CI pipeline using tools like OWASP ZAP or onarQube.
Tools: OWASP ZAP, SonarQube, Jenkins, GitLab CI
Instructions:

1. Set up a simple web application

Here's a complete guide to set up a simple web application that's perfect for testing in your CI/CD pipeline:

Simple Node.js Web Application

1. Create Project Structure

bash

mkdir webapp

cd webapp

npm init -y

2. Install Dependencies

bash

Copy

npm install express ejs

npm install --save-dev nodemon

3. Create Application Files

app.js

javascript

```
const express = require('express');

const app = express();

const path = require('path');


// Configuration

app.set('view engine', 'ejs');

app.set('views', path.join(__dirname, 'views'));

app.use(express.static('public'));


// Routes

app.get('/', (req, res) => {
```

```
  res.render('index', {
    title: 'CI/CD Demo App',
    message: 'Welcome to our automated deployment test!'
  });
});

app.get('/api/health', (req, res) => {
  res.json({
    status: 'healthy',
    timestamp: new Date().toISOString()
  });
});

// Error handling
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});

// Start server
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server running on http://localhost:${PORT}`);
});

module.exports = app; // For testing
```

views/index.ejs

html

Copy

```
<!DOCTYPE html>
<html>
<head>
  <title><%= title %></title>
  <link rel="stylesheet" href="/style.css">
```

```
</head>
<body>
  <h1><%= title %></h1>
  <p class="message"><%= message %></p>

  <div class="api-test">
    <button id="health-check">Check API Health</button>
    <div id="api-response" class="hidden"></div>
  </div>

  <script>
    document.getElementById('health-check').addEventListener('click', async () => {
      const response = await fetch('/api/health');
      const data = await response.json();
      const div = document.getElementById('api-response');
      div.textContent = `Status: ${data.status} (${data.timestamp})`;
      div.classList.remove('hidden');
    });
  </script>
</body>
</html>
```

Run HTML

public/style.css

css

```
body {
  font-family: Arial, sans-serif;
  max-width: 800px;
  margin: 0 auto;
  padding: 20px;
  line-height: 1.6;
}

.message {
  color: #2c3e50;
```

```
  font-size: 1.2em;
  margin-bottom: 30px;
}

.api-test {
  margin-top: 20px;
}

#health-check {
  background-color: #3498db;
  color: white;
  padding: 10px 15px;
  border: none;
  border-radius: 4px;
  cursor: pointer;
  font-size: 1em;
}

#health-check:hover {
  background-color: #2980b9;
}

.hidden {
  display: none;
}

#api-response {
  margin-top: 15px;
  padding: 10px;
  background-color: #f8f9fa;
  border-radius: 4px;
}
```

4. Update package.json

json

```
{
  "name": "webapp",
  "version": "1.0.0",
  "scripts": {
    "start": "node app.js",
    "dev": "nodemon app.js",
    "test": "jest",
    "test:e2e": "playwright test"
  },
  "dependencies": {
    "ejs": "^3.1.8",
    "express": "^4.18.2"
  },
  "devDependencies": {
    "jest": "^29.5.0",
    "nodemon": "^2.0.22",
    "supertest": "^6.3.3",
    "@playwright/test": "^1.35.1"
  }
}
```

5. Add Basic Tests

test/unit/app.test.js

javascript

```
const request = require('supertest');
const app = require('../../app');

describe('GET /', () => {
  it('should return 200 OK', async () => {
    const response = await request(app).get('/');
    expect(response.statusCode).toBe(200);
    expect(response.text).toContain('Welcome to our automated deployment test!');
  });
});
```

```
describe('GET /api/health', () => {
  it('should return healthy status', async () => {
    const response = await request(app).get('/api/health');
    expect(response.statusCode).toBe(200);
    expect(response.body.status).toBe('healthy');
    expect(response.body).toHaveProperty('timestamp');
  });
});
```

test/e2e/homepage.spec.js

javascript

```
const { test, expect } = require('@playwright/test');

test.describe('Web Application', () => {
  test('should display welcome message', async ({ page }) => {
    await page.goto('http://localhost:3000');
    await expect(page.locator('.message')).toHaveText('Welcome to our automated deployment test!');
    await page.screenshot({ path: 'test-results/homepage.png' });
  });

  test('API health check should work', async ({ page }) => {
    await page.goto('http://localhost:3000');
    await page.click('#health-check');
    await expect(page.locator('#api-response')).toBeVisible();
    await expect(page.locator('#api-response')).toContainText('Status: healthy');
    await page.screenshot({ path: 'test-results/api-check.png' });
  });
});
```

6. Run the Application

bash

```
npm run dev
```

The application will be available at http://localhost:3000

Key Features

Modern Web App:

Express.js backend

EJS templating

Interactive frontend with JavaScript

Testing Ready:

Unit tests with Jest

E2E tests with Playwright

API endpoint for health checks

CI/CD Friendly:

Clear separation of concerns

Easy to containerize

Lightweight dependencies

Visual Feedback:

Screenshot capture during tests

Interactive elements for comprehensive testing

To containerize this application, you could add a simple Dockerfile:

Dockerfile

dockerfile

```
FROM node:18-alpine

WORKDIR /app

COPY package*.json ./

RUN npm install

EXPOSE 3000

CMD ["npm", "start"]
```

This gives you a complete, testable web application that's perfect for demonstrating CI/CD pipelines with automated testing, performance monitoring, and deployment workflows.

2. Use OWASP ZAP to scan the application for vulnerabilities or use SonarQube for static code analysis

Here's how to integrate OWASP ZAP for dynamic security scanning and SonarQube for static code analysis into your CI/CD pipeline for the Node.js web application:

Option 1: OWASP ZAP Security Scanning

1. Add ZAP to Your Pipeline

GitLab CI Example (.gitlab-ci.yml)

yaml

```
stages:
  - build
  - test
  - security


zap_scan:
  stage: security
  image: owasp/zap2docker-stable
  variables:
    ZAP_TARGET_URL: "http://webapp:3000"  # Replace with your app URL
  script:
    - zap-baseline.py -t $ZAP_TARGET_URL -r zap-report.html
  artifacts:
    when: always
    paths:
      - zap-report.html
    reports:
      sast: zap-report.html
  only:
    - main
```

Jenkins Example (Jenkinsfile)

groovy

```
stage('OWASP ZAP Scan') {
  agent {
    docker {
      image 'owasp/zap2docker-stable'
      args '-u root --network host'
    }
  }
  steps {
    script {
      sh '''
        zap-baseline.py -t http://localhost:3000 -r zap-report.html
```

```
    '''
   }
  }
  post {
   always {
    archiveArtifacts 'zap-report.html'
   }
  }
}
```

2. Run ZAP Manually (Local Testing)

bash

```
docker run -v $(pwd):/zap/wrk -t owasp/zap2docker-stable zap-baseline.py \
 -t http://localhost:3000 -r zap-report.html
```

Output: zap-report.html (contains vulnerabilities like XSS, SQLi, CSRF)

Option 2: SonarQube Static Code Analysis

1. Set Up SonarQube

Run SonarQube locally (Docker):

bash

```
docker run -d --name sonarqube -p 9000:9000 sonarqube:lts
```

Access UI: http://localhost:9000 (Default login: admin/admin)

2. Add SonarQube to Your Pipeline

GitLab CI Example (.gitlab-ci.yml)

yaml

```
sonarqube_scan:
 stage: test
 image: sonarsource/sonar-scanner-cli
 variables:
  SONAR_HOST_URL: "http://sonarqube:9000"  # Change if hosted
  SONAR_TOKEN: "$SONARQUBE_TOKEN"  # Set in CI variables
 script:
  - sonar-scanner
    -Dsonar.projectKey=webapp
    -Dsonar.sources=.
```

```
    -Dsonar.host.url=$SONAR_HOST_URL
    -Dsonar.login=$SONAR_TOKEN
```

Jenkins Example (Jenkinsfile)

groovy

```
stage('SonarQube Analysis') {
  steps {
    script {
      withSonarQubeEnv('SonarQube') {
        sh 'sonar-scanner \
          -Dsonar.projectKey=webapp \
          -Dsonar.sources=. \
          -Dsonar.host.url=http://sonarqube:9000 \
          -Dsonar.login=$SONARQUBE_TOKEN'
      }
    }
  }
}
```

3. Run SonarQube Manually

bash

```
docker run --rm -v $(pwd):/usr/src sonarsource/sonar-scanner-cli \
  -Dsonar.projectKey=webapp \
  -Dsonar.sources=. \
  -Dsonar.host.url=http://localhost:9000 \
  -Dsonar.login=<your-sonarqube-token>
```

Output: View results in SonarQube dashboard (http://localhost:9000).

3. Automate these tests in the CI pipeline to run every time code is pushed

---

Here's how to fully automate OWASP ZAP (security scanning) and SonarQube (static analysis) in your CI/CD pipeline to run on every code push:

---

1. GitLab CI Pipeline (.gitlab-ci.yml)

yaml

Copy

```
stages:
  - build
  - test
  - security
  - sonarqube

variables:
  APP_URL: "http://webapp:3000"  # Your app's URL
  SONAR_HOST_URL: "http://sonarqube:9000"

# Build and test stages (existing)
build:
  stage: build
  script:
    - npm install
    - npm run build
  artifacts:
    paths:
      - build/

# Unit and E2E tests (existing)
test:
  stage: test
  services:
    - name: node:18
      alias: webapp
  script:
    - npm test
    - npm run test:e2e

# OWASP ZAP Dynamic Scanning
zap_scan:
```

```
  stage: security
  image: owasp/zap2docker-stable
  needs: ["test"]  # Wait for tests to pass
  script:
    - zap-baseline.py -t $APP_URL -r zap-report.html
  artifacts:
    when: always
    paths:
      - zap-report.html
    reports:
      sast: zap-report.html
  rules:
    - if: '$CI_PIPELINE_SOURCE == "push"'  # Run on every push

# SonarQube Static Analysis
sonarqube_scan:
  stage: sonarqube
  image: sonarsource/sonar-scanner-cli
  variables:
    SONAR_TOKEN: "$SONARQUBE_TOKEN"  # Set in GitLab CI variables
  script:
    - sonar-scanner
      -Dsonar.projectKey=webapp
      -Dsonar.sources=.
      -Dsonar.host.url=$SONAR_HOST_URL
      -Dsonar.login=$SONAR_TOKEN
  rules:
    - if: '$CI_PIPELINE_SOURCE == "push"'
```

---

2. Jenkins Pipeline (Jenkinsfile)

groovy

Copy

```
pipeline {
  agent any
```

```
environment {
  APP_URL = "http://localhost:3000"
  SONAR_HOST_URL = "http://sonarqube:9000"
  SONARQUBE_TOKEN = credentials('sonarqube-token')
}

stages {
  // Build stage
  stage('Build') {
    steps {
      sh 'npm install'
      sh 'npm run build'
    }
  }

  // Test stage
  stage('Test') {
    steps {
      sh 'npm test'
      sh 'npm run test:e2e'
    }
  }

  // OWASP ZAP Scan
  stage('Security Scan') {
    agent {
      docker {
        image 'owasp/zap2docker-stable'
        args '-u root --network host'
      }
    }
    steps {
      sh "zap-baseline.py -t ${APP_URL} -r zap-report.html"
```

```
    }
    post {
      always {
        archiveArtifacts 'zap-report.html'
      }
    }
  }

  // SonarQube Analysis
  stage('SonarQube Scan') {
    agent {
      docker {
        image 'sonarsource/sonar-scanner-cli'
      }
    }
    steps {
      withSonarQubeEnv('SonarQube') {
        sh """
          sonar-scanner \
            -Dsonar.projectKey=webapp \
            -Dsonar.sources=. \
            -Dsonar.host.url=${SONAR_HOST_URL} \
            -Dsonar.login=${SONARQUBE_TOKEN}
        """
      }
    }
  }
}

// Trigger on every push
triggers {
  pollSCM('* * * * *')  // Poll SCM every minute
}
}
```

3. Required Setup

For GitLab CI

Add Secrets:

Set SONARQUBE_TOKEN in Settings > CI/CD > Variables

Ensure Services:

Your app must be running during ZAP scan (use services in GitLab CI)

SonarQube server must be accessible

For Jenkins

Install Plugins:

SonarQube Scanner

OWASP ZAP (optional)

Configure Credentials:

Add sonarqube-token in Manage Jenkins > Credentials

Set Up Webhooks (optional):

Configure SCM polling or GitHub/GitLab webhooks

---

4. Expected Outputs

Tool	Artifact	Where to View
OWASP ZAP	zap-report.html	GitLab CI artifacts / Jenkins workspace
SonarQube	Dashboard results	http://sonarqube:9000

Sample ZAP Report (HTML):

4. Take screenshots showing any vulnerabilities or issues detected by the tool.

Since I can't directly generate screenshots, I'll provide you with detailed mockups of what vulnerability reports from OWASP ZAP and SonarQube look like, along with instructions to capture them yourself.

---

1. OWASP ZAP Vulnerability Report Screenshots

A. Summary Dashboard

Key Sections:

Alerts: List of vulnerabilities (High/Medium/Low)

Top Risk: XSS, SQLi, CSRF

Scan Progress: URLs tested

How to Capture:

Run ZAP in your pipeline:

bash

zap-baseline.py -t http://your-app-url -r zap-report.html

Open zap-report.html and screenshot the "Alert Summary" section.

B. Sample Vulnerability (XSS)

Details Shown:

Risk: High

URL: http://yourapp/search?query=<script>alert(1)</script>

Evidence: HTML output with unescaped input

How to Capture:

In the ZAP report, expand any High-risk alert and screenshot the details.

---

2. SonarQube Issue Reports

A. Project Overview

Key Metrics:

Bugs: Code errors

Vulnerabilities: Security risks

Code Smells: Maintainability issues

How to Capture:

Log in to your SonarQube server (http://sonarqube:9000).

Navigate to your project and screenshot the overview page.

B. Sample Security Vulnerability

Details Shown:

Rule: "SQL injection vulnerabilities should not exist"

File: src/database.js

Line: 42 (e.g., query = "SELECT * FROM users WHERE id = " + userInput;)

How to Capture:

In SonarQube, go to Issues > Filter by Vulnerabilities.

Click any issue and screenshot the code snippet + description.

---

3. GitLab CI/CD Integration

A. Pipeline with Security Warnings

Where to Find:

In GitLab MRs, under Security tab.

How to Capture:

Push code with vulnerabilities.

In your merge request, screenshot the Security widget.

---

4. Jenkins with ZAP Artifacts

A. Archived ZAP Report

Where to Find:

In Jenkins build > Artifacts > zap-report.html.

How to Capture:

After a pipeline run, open the build.

Navigate to Artifacts and screenshot the file list.

How to Generate Real Screenshots

For ZAP:

bash

```
docker run -v $(pwd):/zap/wrk -t owasp/zap2docker-stable zap-baseline.py \
  -t http://localhost:3000 -r zap-report.html
```

Open zap-report.html in a browser and screenshot.

For SonarQube:

bash

```
sonar-scanner -Dsonar.login=your-token
```

Log in to SonarQube UI and screenshot findings.

Key Takeaways

Tool	What to Screenshot	Example Findings
OWASP ZAP	zap-report.html Alert Summary	XSS, SQLi, CSRF
SonarQube	Project Dashboard / Issue Details	Hardcoded passwords, unsafe functions

Need help interpreting specific vulnerabilities? Share a screenshot of your actual report, and I can explain the risks and fixes!

6. Continuous Deployment (CD)

Task 7: Automate Deployment to Staging Environment

Objective: Automate the deployment of an application to a staging environment after a successful build and test.
Tools: Jenkins, AWS CodeDeploy, Docker, Kubernetes
Instructions:

1. Create a simple application (Node.js, Python, or Java)

Here's a simple Node.js web application with vulnerabilities intentionally included for security scanning (OWASP ZAP/SonarQube) and CI/CD pipeline testing. The app has:

A login form (with SQLi vulnerability)

Search functionality (XSS vulnerability)

API endpoint (hardcoded secrets)

---

1. Project Setup

bash

mkdir vulnerable-app && cd vulnerable-app

npm init -y

npm install express ejs sqlite3 body-parser

---

2. Application Code

app.js (Main Server)

```javascript
const express = require('express');
const bodyParser = require('body-parser');
const sqlite3 = require('sqlite3').verbose();
const app = express();

// Security anti-patterns (for demo only)
const HARDCODED_API_KEY = 'secret123'; // SonarQube will flag this
const ADMIN_PASSWORD = 'adminpass'; // Hardcoded credential

// Database setup (SQLite)
const db = new sqlite3.Database(':memory:');
db.serialize(() => {
  db.run("CREATE TABLE users (id INTEGER PRIMARY KEY, username TEXT, password TEXT)");
  db.run("INSERT INTO users (username, password) VALUES ('admin', '"+ADMIN_PASSWORD+"')");
});

// Middleware
app.use(bodyParser.urlencoded({ extended: true }));
app.set('view engine', 'ejs');

// Routes
app.get('/', (req, res) => {
  res.render('index', { query: req.query.q || '' });
});

// Vulnerable search (XSS)
app.get('/search', (req, res) => {
  // No input sanitization!
  res.send(`<h1>Results for: ${req.query.term}</h1>`);
});

// Vulnerable login (SQLi)
app.post('/login', (req, res) => {
```

```
  const { username, password } = req.body;
  // SQL injection possible!
  db.get(`SELECT * FROM users WHERE username='${username}' AND password='${password}'`,
    (err, row) => {
      if (row) res.send('Login success!');
      else res.send('Login failed');
    }
  );
});

// Insecure API
app.get('/api/data', (req, res) => {
  if (req.headers['x-api-key'] === HARDCODED_API_KEY) {
    res.json({ data: 'sensitive_info' });
  } else {
    res.status(403).send('Forbidden');
  }
});

// Start server
app.listen(3000, () => {
  console.log('Vulnerable app running on http://localhost:3000');
});
```

views/index.ejs (Frontend)

```
html
<!DOCTYPE html>
<html>
<head>
  <title>Vulnerable App</title>
</head>
<body>
  <h1>Vulnerable App</h1>

  <!-- Search (XSS vulnerable) -->
```

```
  <form action="/search" method="GET">
    <input type="text" name="term" placeholder="Search...">
    <button>Search</button>
  </form>

  <!-- Login (SQLi vulnerable) -->
  <h2>Login</h2>
  <form action="/login" method="POST">
    <input type="text" name="username" placeholder="Username">
    <input type="password" name="password" placeholder="Password">
    <button>Login</button>
  </form>
</body>
</html>
```

Run HTML

---

3. Intentionally Added Vulnerabilities

Vulnerability	Location	Tool That Will Detect It
SQL Injection	db.get() in /login route	OWASP ZAP, SonarQube
Cross-Site Scripting (XSS)	Unescaped req.query.term in /search	OWASP ZAP
Hardcoded Secrets	HARDCODED_API_KEY and ADMIN_PASSWORD	SonarQube
Missing HTTPS	Entire app (HTTP only)	OWASP ZAP

---

4. How to Run

bash

```
node app.js
```

Access at: http://localhost:3000

---

5. Test the Vulnerabilities

A. SQL Injection

In login form, enter:

Username: admin'--

Password: [anything]

You'll bypass authentication!

B. XSS Attack

In search box, enter:

html

<script>alert('XSS!')</script>

Run HTML

The script will execute.

C. API Key Exposure

Check app.js - the key secret123 is hardcoded.

2. Set up a CI pipeline that builds the application, runs tests, and deploys it to a staging environment (using AWS, Docker, or Kubernetes)

Here's a complete CI/CD pipeline using GitLab CI (easily adaptable to Jenkins) that builds, tests, scans, and deploys the vulnerable Node.js app to a staging environment using Docker and AWS ECS (Kubernetes alternative provided).

---

1. Pipeline Architecture

mermaid

graph LR

A[Build] --> B[Test]

B --> C[Security Scan]

C --> D[Deploy to Staging]

D --> E[Smoke Test]

---

2. Full .gitlab-ci.yml

yaml

stages:

- build

- test

- security

```
  - deploy
  - smoke

variables:
  APP_NAME: "vulnerable-app"
  AWS_REGION: "us-east-1"
  ECS_CLUSTER: "staging-cluster"
  ECS_SERVICE: "vulnerable-app-service"
  DOCKER_IMAGE: "$CI_REGISTRY_IMAGE:$CI_COMMIT_SHA"

# Build stage
build:
  stage: build
  image: node:18
  script:
    - npm install
    - npm run build
  artifacts:
    paths:
      - node_modules/
      - build/

# Test stage
test:
  stage: test
  image: node:18
  services:
    - postgres:13  # For DB integration tests (if needed)
  script:
    - npm test
    - npm run test:e2e

# Security scan (ZAP + SonarQube)
security:
```

```
  stage: security
  parallel:
    - name: "OWASP ZAP"
      image: owasp/zap2docker-stable
      script:
        - zap-baseline.py -t http://localhost:3000 -r zap-report.html
      artifacts:
        paths: [zap-report.html]
    - name: "SonarQube"
      image: sonarsource/sonar-scanner-cli
      variables:
        SONAR_TOKEN: "$SONARQUBE_TOKEN"
      script:
        - sonar-scanner -Dsonar.login=$SONAR_TOKEN

# Build and push Docker image
docker-build:
  stage: build
  image: docker:20.10
  services:
    - docker:20.10-dind
  script:
    - docker build -t $DOCKER_IMAGE .
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
    - docker push $DOCKER_IMAGE

# Deploy to AWS ECS
deploy-staging:
  stage: deploy
  image: amazon/aws-cli:2.13
  only:
    - main
  script:
    - aws configure set aws_access_key_id "$AWS_ACCESS_KEY_ID"
```

```
    - aws configure set aws_secret_access_key "$AWS_SECRET_ACCESS_KEY"
    - aws ecs update-service \
        --cluster $ECS_CLUSTER \
        --service $ECS_SERVICE \
        --force-new-deployment \
        --region $AWS_REGION

# Smoke test staging
smoke-test:
  stage: smoke
  image: alpine/curl:3.14
  script:
    - curl -sSf http://staging.example.com/api/health | grep '"status":"healthy"'
```

---

3. Key Components

A. Docker Setup

Dockerfile

dockerfile

```
FROM node:18-alpine
WORKDIR /app
COPY package*.json ./
RUN npm install
EXPOSE 3000
CMD ["node", "app.js"]
```

B. AWS ECS Deployment

Pre-requisites:

AWS ECS cluster with Fargate task definition

Task definition pointing to $CI_REGISTRY_IMAGE

IAM user with ECS permissions

Required CI Variables:

AWS_ACCESS_KEY_ID

AWS_SECRET_ACCESS_KEY

CI_REGISTRY_* (for Docker registry auth)

---

4. Kubernetes Alternative (AWS EKS)

Replace deploy-staging job with:

yaml

```
deploy-staging:
  stage: deploy
  image: bitnami/kubectl:latest
  script:
    - echo $KUBE_CONFIG | base64 -d > kubeconfig.yaml
    - kubectl --kubeconfig=kubeconfig.yaml set image deployment/vulnerable-app vulnerable-app=$DOCKER_IMAGE
```

Requirements:

Base64-encoded kubeconfig in KUBE_CONFIG variable

Deployment named vulnerable-app in cluster

---

5. Jenkins Alternative

Jenkinsfile (Declarative Pipeline)

groovy

```
pipeline {
  agent any
  environment {
    DOCKER_IMAGE = "registry.example.com/vulnerable-app:${env.BUILD_ID}"
  }
  stages {
    stage('Build') {
      steps {
        sh 'npm install && npm run build'
      }
    }
    stage('Test') {
      steps {
        sh 'npm test'
      }
    }
    stage('Security Scan') {
```

```
    parallel {
      stage('OWASP ZAP') {
        steps {
          sh 'docker run owasp/zap2docker-stable zap-baseline.py -t http://localhost:3000'
        }
      }
      stage('SonarQube') {
        steps {
          withSonarQubeEnv('sonarqube') {
            sh 'sonar-scanner'
          }
        }
      }
    }
  }
  stage('Deploy to Staging') {
    when {
      branch 'main'
    }
    steps {
      script {
        docker.build(DOCKER_IMAGE)
        docker.withRegistry('https://registry.example.com', 'docker-creds') {
          docker.image(DOCKER_IMAGE).push()
        }
        sh "aws ecs update-service --cluster staging-cluster --service vulnerable-app-service --force-new-deployment"
      }
    }
  }
 }
}
```

---

6. Expected Pipeline Flow

Build: Creates Docker image

Test: Runs unit + E2E tests

Security:

ZAP finds SQLi/XSS

SonarQube flags hardcoded secrets

Deploy: Updates ECS service or Kubernetes deployment

Smoke Test: Verifies staging is healthy

---

7. Post-Deployment

Access Staging Environment:

bash

curl http://staging.example.com

Monitor:

AWS ECS console → Service deployments

Kubernetes: kubectl get pods -n staging

---

8. Key Improvements

Rollback: Add if smoke tests fail

yaml

rollback:

needs: ["smoke-test"]

when: on_failure

script:

- aws ecs update-service --cluster $ECS_CLUSTER --service $ECS_SERVICE --task-definition previous-version

Notifications: Slack alerts on failure

Feature Flags: Control risky features in staging

Need help adapting this for your specific cloud provider? Let me know!

3. Document the pipeline setup, and take screenshots of the deployment logs

Here's a comprehensive documentation of the CI/CD pipeline setup with mock deployment log screenshots:

CI/CD Pipeline Documentation

1. Pipeline Overview

mermaid

graph TD

  A[Build] --> B[Test]

  B --> C[Security Scan]

  C --> D[Deploy to Staging]

  D --> E[Smoke Test]

Key Stages:

Build: Create Docker image

Test: Run unit + integration tests

Security Scan: OWASP ZAP + SonarQube

Deploy: AWS ECS/Kubernetes

Smoke Test: Validate staging

---

2. Setup Instructions

A. Prerequisites

Requirement	Example Value
GitLab/AWS account	CI_REGISTRY_USER, AWS_ACCESS_KEY_ID
Docker registry	registry.gitlab.com/your/project
ECS Cluster + Service	staging-cluster, vulnerable-app-service

B. Configure CI Variables

bash

GitLab: Settings > CI/CD > Variables

AWS_ACCESS_KEY_ID   = "AKIA..."

AWS_SECRET_ACCESS_KEY   = "secret..."

SONARQUBE_TOKEN   = "sqp_..."

---

3. Mock Deployment Logs

A. Build Stage

Installs dependencies

Builds Docker image (docker build -t $DOCKER_IMAGE .)

B. Test Stage

Runs npm test

Executes Playwright E2E tests

C. Security Scan

ZAP detects 3 high-risk vulnerabilities

SonarQube flags 5 code smells

D. Deployment (AWS ECS)

bash

Actual deployment log

aws ecs update-service --cluster staging-cluster --service vulnerable-app-service

Triggers new task definition

Health checks pass

E. Smoke Test

bash

curl -sSf http://staging.example.com/api/health

Output: {"status":"healthy"}

---

4. Screenshot Guide (Real Implementation)

GitLab Pipeline

Navigate to: CI/CD > Pipelines

Click on the latest pipeline

Screenshot each stage:

AWS ECS Deployment

Go to: ECS > Clusters > staging-cluster

Screenshot:

Service events

Running tasks

---

5. Troubleshooting

Issue	Solution
Docker build fails	Check Dockerfile permissions
ECS deployment stuck	Verify IAM roles
ZAP timeout	Increase scan timeout in .zap.conf

---

6. Post-Deployment Checks

Verify Staging URL:

bash

curl -I http://staging.example.com

HTTP/1.1 200 OK

Check Logs:

bash

aws logs tail /ecs/vulnerable-app --follow

---

7. Sample Screenshot Annotations

New deployment triggered by CI

Task registration completes

Health checks pass

---

8. Key Files

.gitlab-ci.yml - Pipeline definition

Dockerfile - Container setup

aws/ - Terraform/IaC (optional)

To capture real screenshots from your pipeline:

Run the pipeline (git push origin main)

Follow the screenshot locations above

Annotate with arrows/text using tools like Snagit or Greenshot

Need specific log examples from your actual deployment? Share your pipeline URL and I can help analyze it!

4. Ensure the application is live and accessible from the staging environment

Here's a step-by-step guide to verify your application is live and accessible in the staging environment, with mock screenshots for clarity:

---

1. Verify Staging Deployment Status

For AWS ECS

Check ECS Console
Navigate to:
AWS Console > ECS > Clusters > staging-cluster > Services > vulnerable-app-service

Desired tasks = Running tasks (e.g., 1/1)

Last deployment shows "COMPLETED"

View Task Logs
Click on the task ID > Logs tab:

log

Server running on http://0.0.0.0:3000

For Kubernetes (EKS)

bash

kubectl -n staging get pods

NAME READY STATUS RESTARTS AGE

vulnerable-app-5f7b8c6d8c-9j2k4 1/1 Running 0 2m

---

2. Test Accessibility

A. HTTP Request (Terminal)

bash

curl -I http://staging.example.com

HTTP/1.1 200 OK

Server: nginx/1.18.0

**B. Browser Access

Open http://staging.example.com in a browser:

Expected: Login page renders with no errors.

---

3. Validate Key Endpoints

API Health Check

bash

curl http://staging.example.com/api/health

json

{"status":"healthy","timestamp":"2024-02-20T12:00:00Z"}

Login Functionality (SQLi Test)

bash

curl -X POST -d "username=admin'--&password=123" http://staging.example.com/login

Login success! # (Confirms SQLi vulnerability exists)

---

4. Monitor Network Traffic

Use Browser DevTools (F12 > Network):

Verify no 5xx errors.

Check API response times (<500ms).

---

5. Automated Smoke Test (CI Pipeline)

Add this to your .gitlab-ci.yml or Jenkinsfile:

yaml

smoke-test:

  stage: verify

  image: alpine/curl

  script:

```
- apk add --no-cache curl
- curl -sSf http://staging.example.com/api/health | grep -q '"status":"healthy"'
- curl -sSf http://staging.example.com | grep -q "Vulnerable App"
```

Expected Pipeline Output:

---

6. Troubleshooting Checklist

Symptom	Debugging Command
502 Bad Gateway	kubectl describe ingress -n staging
Connection refused	aws ecs describe-tasks --cluster staging-cluster --tasks <TASK_ID>
DNS resolution failed	nslookup staging.example.com

---

7. Security Validation

HTTPS Enforcement (if applicable):

bash

```
curl -I https://staging.example.com
HTTP/2 200
```

OWASP ZAP Rescan:

bash

```
docker run owasp/zap2docker-stable zap-cli quick-scan -s http://staging.example.com
```

---

8. Final Confirmation

✅ Browser: Manual test passed

✅ API: 200 OK responses

✅ CI/CD: Smoke tests green

✅ Logs: No errors in ECS/EKS logs

Next Steps:

Promote to Prod: After validation

Monitor: Set up CloudWatch/Prometheus alerts

Harden Security: Fix vulnerabilities detected by ZAP/SonarQube

Need real screenshots from your environment? Share your staging URL and I'll help analyze it!