## Introduction and Context

Modern development teams need reliable automation for building, testing, and deploying code while ensuring security throughout the software lifecycle. In this case study, you'll automate a Python library's CI/CD pipeline with GitHub Actions, integrate code scanning and dependency alerts, set up a custom auto-tagging release workflow, and enable the dependency graph to surface vulnerable packages.

## Case Study Overview

**Problem Statement (Real-World Case Study)**
CryptoLib is an open-source Python library for blockchain utilities. Contributors often forget to run tests before merging, leading to broken releases. Moreover, it lacks automated security checks, so vulnerabilities can slip into production. CryptoLib's maintainers want to automate testing and deployments, enforce code scanning, auto-tag releases based on version bumps, and receive dependency vulnerability alerts. Success is a green CI/CD workflow, zero high-severity vulnerabilities, and consistent semantic versioning without manual intervention.

**Learning Objectives**

- Create a CI/CD pipeline with GitHub Actions
- Implement code scanning and dependency vulnerability alerts
- Build a custom auto-tagging release workflow
- Enable and use the dependency graph for security insights

## Concepts Explained with Analogies

**GitHub Actions (CI/CD Pipelines)**
Analogy: A factory assembly line that runs quality checks and packages products automatically whenever new parts arrive.
Technical: GitHub Actions uses YAML workflows stored in `.github/workflows/` to trigger jobs (build, test, deploy) on repository events like `push` or `pull_request`

**Security & Insights (Code Scanning, Dependencies)**
Analogy: A security guard scanning every component for defects before letting it into the store.
Technical: Code scanning analyzes your codebase with CodeQL or third-party tools to detect vulnerabilities and errors, while Dependabot alerts flag insecure dependencies in your manifest files

**Custom Workflows (Auto-Tagging Releases)**
Analogy: A librarian automatically labels new editions of a book based on its revision history.
Technical: Custom workflows like `action-autotag` can parse commit messages for semantic version tags ( `#major` , `#minor` , `#patch` ) and create Git tags and releases accordingly

**Dependency Graph (Vulnerability Alerts)**
Analogy: A map of every supplier your factory relies on, highlighting those with safety recalls.
Technical: The dependency graph visualizes all direct and transitive dependencies and integrates with Dependabot to alert on known vulnerabilities in your packages

## Step-by-Step Guided Walkthrough

## Step 1: Create the CI/CD Workflow

1. In your repo, create `.github/workflows/ci.yml`.
2. Add the following YAML to run tests on Python versions 3.8–3.11 when code is pushed or a PR is opened:

```
name: Python CI
on: [push, pull_request]
jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: [3.8, 3.9, 3.10, 3.11]
    steps:
      - uses: actions/checkout@v4
      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: ${{ matrix.python-version }}
      - name: Install dependencies
        run: pip install -r requirements.txt
      - name: Run tests
        run: pytest
```

3. Commit and push. The workflow appears under the **Actions** tab and runs automatically

**Checkpoint:** How does the `matrix` strategy improve testing coverage?

## Step 2: Add Code Scanning and Dependency Alerts

1. **Code Scanning:**
    - In **Security** → **Code scanning**, click **Set up code scanning** and choose **Set up this workflow** under **CodeQL analysis**.
    - Commit `.github/workflows/codeql.yml` to scan on `push` and `pull_request`.
2. **Dependabot Alerts:**
    - Create `dependabot.yml` in `.github/` with:

    ```
    version: 2
    updates:
      - package-ecosystem: "pip"
        directory: "/"
        schedule:
          interval: "daily"
    ```

    - Enable alerts in **Settings** → **Security & analysis**

**Checkpoint:** Why run code scanning on both `push` and `pull_request` events?

## Step 3: Build a Custom Auto-Tagging Release Workflow

1. Install `Klemensas/action-autotag` by creating `.github/workflows/release.yml`:

```
name: Auto-Tag Release
on:
 push:
   branches: [ main ]
jobs:
 tag:
   runs-on: ubuntu-latest
   steps:
     - uses: actions/checkout@v4
     - uses: Klemensas/action-autotag@stable
       with:
         GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

2. Commit and push. On each push to `main`, the action reads `pyproject.toml` or `setup.py` version and tags a new release if it differs

**Checkpoint:** How does auto-tagging ensure consistency in semantic versioning?

## Step 4: Enable and Monitor the Dependency Graph

1. In **Settings** → **Security & analysis**, enable **Dependency graph**.

2. Navigate to **Insights** → **Dependency graph** to review all dependencies and known alerts

3. Address critical alerts by updating or patching libraries.

**Checkpoint:** What steps would you take when a high-severity vulnerability is reported?

## Best Practices and Tips

- Use **short-lived feature branches** and require status checks before merges.

- Keep workflows **DRY** by extracting common steps into composite actions.

- Enforce **required code scanning** and **Dependabot checks** in branch protection rules.
- Tag releases with **annotated tags** to include changelog information.
- Regularly **audit** the dependency graph for transitive vulnerabilities.

## Real-World Application and Extension

- Chain workflows to **deploy** the library to PyPI after successful testing and tagging.
- Integrate **Slack notifications** for build failures and security alerts.
- Customize auto-tagging to generate **release notes** based on commit history.