TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

Seminar Report

Continuous Integration/Continuous Deployment

CS423 - Software Testing

Group 05

22125041 – Mai Đăng Khoa

22125042 – Lê Mai Khôi

22125075 – Nguyễn Duy Phúc

22125084 – Nguyễn Trọng Quý

HCM, 03/12/2025

Contents

1 Introduction

In modern software development, teams often work on the same codebase at the same time. Without good practices, this can easily lead to problems such as integration conflicts, bugs that appear late in the process, or code that works on one machine but fails on another. As projects grow larger, it becomes difficult to test every change manually and to keep the main branch stable for release.
To solve these issues, many teams use CI/CD [12], which stands for Continuous Integration and Continuous Delivery or Continuous Deployment. These practices help developers check, test, and deliver code in a fast and reliable way. In this report, we explain the ideas behind CI and CD, why they are important, and how they improve the quality of software projects.

2 Continuous Integration

Continuous Integration (CI) [11] is a practice where developers frequently merge their code into a shared repository. Each time code is pushed, the system automatically builds the project and runs tests. The goal of CI is to give fast feedback, catch errors early, and keep the main branch stable. Our demo can be found at: https://youtu.be/mZrboD2Ztlk

2.1 Why Continuous Integration is Needed

Before CI, developers often waited many days before merging their work. When they finally combined their changes, large merge conflicts appeared and were difficult to solve. Bugs were also found late because testing was slow and mostly manual. In addition, code often behaved differently on different machines due to missing dependencies or different local settings. CI solves these problems by checking every change as soon as it is pushed.

2.2 The Three Pillars of CI

CI is built on three main pillars:

- **Source Code Management (SCM):** Developers use Git and follow a clean workflow such as GitHub Flow [7]. Each new feature is created in a separate branch, and changes are reviewed through pull requests. This makes the history clear and prevents direct changes to the main branch.
- **Automated Build:** The CI system automatically checks out the code, installs dependencies, and builds the project [9]. This makes sure the code compiles correctly and that missing files or dependencies are detected early.
- **Automated Testing:** CI runs unit tests, linting, static analysis, and sometimes security scans. If any test fails, the pipeline fails and the code cannot be merged. This keeps the main branch clean and prevents regressions.

2.3 How a CI Pipeline Works

A typical CI pipeline starts when a developer writes code and pushes it to a feature branch. The CI system receives this push and performs fast checks: it builds the project and runs unit tests. When the developer opens a pull request, CI runs more complete tests such as lint checks, security scans, and full test suites. If everything passes, the pull request can be merged into the main branch.
After the merge, another pipeline may run to build production-ready artifacts such as JAR files, Docker images, or deployment packages. At this point, the code is integrated, tested, scanned, and ready for delivery.

2.4 What CI Automates

CI automates many important tasks, including:

- building and compiling the project,
- running unit tests and integration tests,
- checking code style and formatting,
- scanning code for bugs or vulnerabilities,
- measuring test coverage,
- generating and storing build artifacts.

By automating these tasks, CI saves time and removes human error. It makes sure every change follows the same quality standards.

2.5 CI Tools

Many tools support CI, such as GitHub Actions [5], GitLab CI [4], CircleCI [1], and Travis CI [10]. GitHub Actions is popular because it is built directly into GitHub and uses YAML files to define workflows. Enterprise teams also use tools like Jenkins [8] because they offer more customization. No matter which tool is used, the purpose of CI remains the same: to test and validate every change automatically.

2.6 Example: GitHub Actions

GitHub Actions allows developers to create workflows that run on events such as pushes or pull requests. A workflow is written in a YAML file and contains jobs and steps. Each job runs on a clean virtual machine, and each step can run commands or use predefined actions.
Below is a simple example of a CI workflow for a Node.js project:

```
1 name: CI Pipeline
2
3 on:
4   push:
5   pull_request:
6
```

```yaml
jobs:
  build-and-test:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout Code
        uses: actions/checkout@v3

      - name: Install Dependencies
        run: npm install

      - name: Run Tests
        run: npm test
```

Listing 1: GitHub Actions YAML file

The workflow starts on every push and pull request. It checks out the code, installs dependencies, and runs tests. If any test fails, the workflow becomes red and the code cannot be merged.

3 Continuous Deployment

3.1 From Integration to Release

In the DevOps lifecycle, if CI (Continuous Integration) is about "doing it right" (build & test), then CD (Continuous Delivery/Deployment) is about "getting it right to the user." At this stage, two levels must be clearly distinguished:

- **Continuous Delivery:** After passing CI, code is packaged and waits in a Staging environment. Pushing to Production requires a **manual approval** (a human decision). This model fits systems requiring high legal or risk control.
- **Continuous Deployment:** This is the ultimate goal of the project. Human intervention is completely removed. Any code change that passes the CI pipeline is **automatically** pushed straight to the Production environment. This reduces Time-to-Market to near zero.

Our demo can be found at: https://youtu.be/CqcILxtAmGg

3.2 Environment Strategy

To ensure safety for a fully automated process, a code flow was established through three distinct environments:

1. **Local Environment:** The development environment on the developer's machine, using mock data.
2. **Staging Environment** (`staging.marker.io`): A mirror of Production. This is the "laboratory" for running final integration tests with a snapshot of real (anonymized) data.
3. **Production Environment** (`app.marker.io`): The live environment serving actual end-users.

3.3 Technical Implementation Analysis

Two distinct deployment strategies are applied for the Backend and Frontend, optimized for each architecture type.

3.3.1 Backend Deployment: Containerization (Docker)

Strategy: Instead of manually copying `.jar` files (which is prone to environment errors), the application is packaged into a **Docker Image**. This ensures absolute consistency: *"If it runs on the Dev machine, it will definitely run on the Server."*

```yaml
jobs:
  build-and-push:
    runs-on: ubuntu-latest
    steps:
      # 1. Secure Login to Registry
      - name: Log in to GitHub Container Registry
        uses: docker/login-action@v3
        with:
          registry: ghcr.io
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}

      # 2. Automatic Versioning (Metadata Extraction)
      - name: Extract metadata (tags, labels)
        id: meta
        uses: docker/metadata-action@v5
        with:
          images: ghcr.io/${{ github.repository }}
          tags: |
            type=sha   # Tag with Commit Hash (e.g., sha-8f3a2)
            type=raw,value=latest # Always update the 'latest' tag

      # 3. Build & Push
      - name: Build and push
        uses: docker/build-push-action@v6
        with:
          context: .
          push: true
          tags: ${{ steps.meta.outputs.tags }}
          labels: ${{ steps.meta.outputs.labels }}
```

Listing 2: Backend Deployment Workflow (deploy-backend.yml)

Deep Dive Analysis:

- **Security:** The `${{ secrets.GITHUB_TOKEN }}` (a temporary token) is used to log in to the GitHub Container Registry (GHCR) instead of hard-coding passwords.
- **Traceability:** Tagging images with `type=sha` allows mapping the current deployment exactly back to the specific line of code in the Git history.
- **Automation:** As soon as the image is pushed to the Registry with the `latest` tag, the cloud infrastructure (Railway) automatically detects it and pulls the new version to update the server.

3.3.2 Frontend Deployment: Static Site Hosting (Cloudflare Pages)

Strategy: The React Frontend is compiled into static files (HTML/CSS/JS) and distributed via Cloudflare Pages [2] network to achieve the fastest page load speeds.

```yaml
jobs:
  deploy-frontend:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v3

      # 1. Install & Build (Clean Install)
      - name: Install & Build
        run: |
          npm ci
          npm run build
        env:
          # Inject API environment variable at Build time
          VITE_API_BASE_URL: ${{ secrets.PROD_API_BASE_URL }}

      # 2. Deploy to Cloudflare
      - name: Deploy to Cloudflare Pages
        uses: cloudflare/wrangler-action@v3
        with:
          apiToken: ${{ secrets.CLOUDFLARE_API_TOKEN }}
          accountId: ${{ secrets.CLOUDFLARE_ACCOUNT_ID }}
          command: pages deploy ./dist --project-name=simple-todo-app-
   frontend
```

Listing 3: Frontend Deployment Workflow (deploy-frontend.yml)

Deep Dive Analysis:

- **Stability (`npm ci`):** `npm ci` is used instead of `npm install` to ensure the exact installation of library versions locked in `package-lock.json`, eliminating the risk of "version drift" between builds.
- **Build-time Configuration:** Since React is client-side, the Backend API address is required at build time. The `VITE_API_BASE_URL` variable is "injected" from GitHub Secrets into the code right at this step.
- **Zero Downtime:** Cloudflare Pages supports Atomic Deployment – the entire new file set is uploaded before switching users over, ensuring the website never encounters a broken state during updates.

3.4 Continuous Monitoring

The system doesn't stop at Deployment. Once in Production, the application's "vital signs" are maintained using a 3-pillar monitoring system:

1. **Metrics:** Tracks hardware health (CPU, RAM) and performance (Latency).
2. **Logs:** Records detailed runtime errors for debugging.

3. **Alerting:** The system automatically fires Slack/Email messages to the Development team the moment indicators are abnormal, helping detect errors before customers complain.

3.5 Key Benefits

Implementing an automated CD system delivers 3 quantitative values:

- **Risk Reduction:** The process moves from "Big Bang Releases" (once every 3 months, high risk) to daily releases. Small bugs are easy to fix, and rollbacks are instant.
- **Hyper-Fast Feedback Loop:** Features coded in the morning can be used by customers in the afternoon. Product hypotheses can be validated immediately.
- **Human Liberation:** Deployment becomes a foundational, boring, and automated process, allowing developers to focus entirely on creating new features (Coding) instead of operations (Ops).

4 AI-First Approach in CI/CD

A central question many teams ask is whether AI can actually automate meaningful portions of CI/CD, and to what extent such automation is practical today.

The answer is nuanced. On the one hand, AI models, especially large language models (LLMs), are extremely capable at interpreting source code, identifying logical inconsistencies, and suggesting improvements. On the other hand, CI/CD pipelines demand determinism, repeatability, and strict reliability, qualities that AI models - with their stochastic, context-sensitive behavior - cannot always guarantee.

Therefore, an AI-first approach does not replace the pipeline's deterministic components; instead, it augments them, introducing a new layer of intelligent analysis that sits between human review and automated execution.

Our demo can be found at https://youtu.be/M4T7DVkbdjM.

4.1 LLM Approach: Copilot Review

A clear example of AI augmentation is GitHub Copilot's ability [6] to participate directly in pull request (PR) reviews. Instead of passively waiting for a human reviewer to leave comments, Copilot performs an immediate inspection of the incoming code changes. This inspection is not limited to superficial formatting issues.

Copilot tries to infer what the code is supposed to achieve, how it interacts with existing modules, whether error cases are handled, and whether the implementation patterns align with the conventions of the repository. In practice, this often means that Copilot identifies cases where the developer forgot to handle a corner case, or where a block of logic could be simplified to avoid redundancy.

What makes this significant is the speed: Copilot can leave a meaningful review within seconds of opening a PR, ensuring that the developer receives feedback early, even before the first human reviewer arrives.

The screenshot in Figure 1 illustrates this idea. In many real-world repositories, Copilot's comments resemble those of a thoughtful teammate: "This block may throw a null pointer in edge cases," or "Consider whether this query should be parameterized to avoid injection risks."

These insights would traditionally rely on a human's knowledge and intuition, but AI now performs them continuously and tirelessly. This does not eliminate the need for human review; however, it significantly reduces reviewer fatigue and shortens iteration cycles.

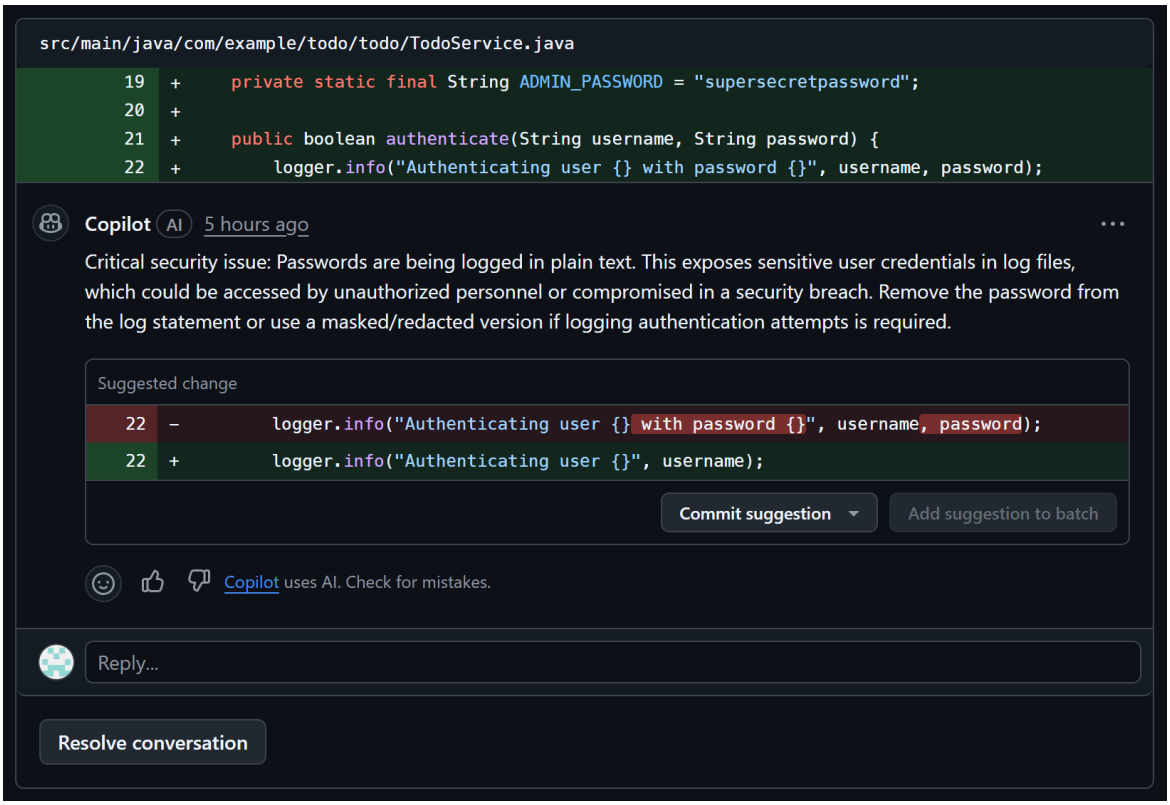

Figure 1: GitHub Copilot's comment in a PR conversation, suggesting the developer remove the password from the log.

The natural question that follows is: if Copilot can already review code, can it eventually take more autonomous actions in a CI/CD pipeline? For example, could an AI agent automatically fix failing tests, resolve merge conflicts, or update dependency versions?

Theoretically, yes—AI models can already generate patches, rewrite functions, and propose alternative implementations. However, the current practical limit is reliability. AI is extremely helpful as long as a human is supervising the final outcome. But in a pipeline, autonomy without supervision can become dangerous: an AI-generated patch that silently changes business logic or introduces subtle data inconsistencies could easily pass through a pipeline if not carefully monitored.

Thus, for the foreseeable future, AI-generated suggestions will remain part of the feedback and review loop rather than fully automated merge actions.

4.2 Deterministic Code Smell: SonarQube

While LLMs provide intelligent reasoning, deterministic tools like SonarQube [3] continue to play an essential role. SonarQube is not a generative system, but its rule engine has become increasingly sophisticated, detecting a wide range of vulnerabilities, code smells, and maintainability issues. What distinguishes SonarQube in an AI-first pipeline is not generative intelligence but the discipline it enforces.

SonarQube does not attempt to "understand intention"; instead, it ensures that any code entering the repository complies with explicit standards and well-defined quality gates. These gates act as a final checkpoint that even AI-powered reviews must pass.

In practice, SonarQube complements LLM-based reviewers rather than competing with them. Copilot is strong at identifying logical issues that static rules may not capture, such as missing error handling, ambiguous control flow, or business-specific inconsistencies. SonarQube, in contrast, excels at detecting concrete, rule-based problems such as potential injection vulnerabilities, unused variables, duplicated blocks, or insufficient test coverage. Figure 2 shows SonarQube's interface with some code smells in a Java file called TodoService.

One interesting consequence of this hybrid model is the shift in developer psychology. Historically, CI/CD pipelines have sometimes been viewed as adversarial: a pipeline "rejects" your code, and the developer must retry until it passes. But when the first layer of review is conversational and explanatory (e.g., Copilot's inline comments), the pipeline feels more like a *mentor*. Developers receive suggestions rather than failures. Meanwhile, SonarQube ensures that the code still respects the technical standards of the organization. The result is a workflow that encourages learning and improvement rather than frustration.

4.3 Discussion

This leads to another macro-level question: are we moving toward pipelines where AI continuously monitors the repository, not just at PR time but throughout the development cycle? To reach this level of autonomy, two conditions must be met. First, AI agents must gain deeper domain understanding. They must learn the semantics of a given codebase, the business rules, and the architectural principles that shape correct behavior.

Current models are very good at recognizing patterns, but true domain understanding remains a challenge. Second, teams must build feedback loops where AI-generated suggestions are evaluated, corrected, and eventually reinforced. Without this loop, models cannot specialize effectively to a project's needs. We might not be sure if this can be pulled off, but it could be a scenario someday in the future.

In summary, an AI-first CI/CD pipeline is not a vision of full replacement but a vision of augmentation. AI tools like Copilot enhance the review process with rapid, context-aware insights,

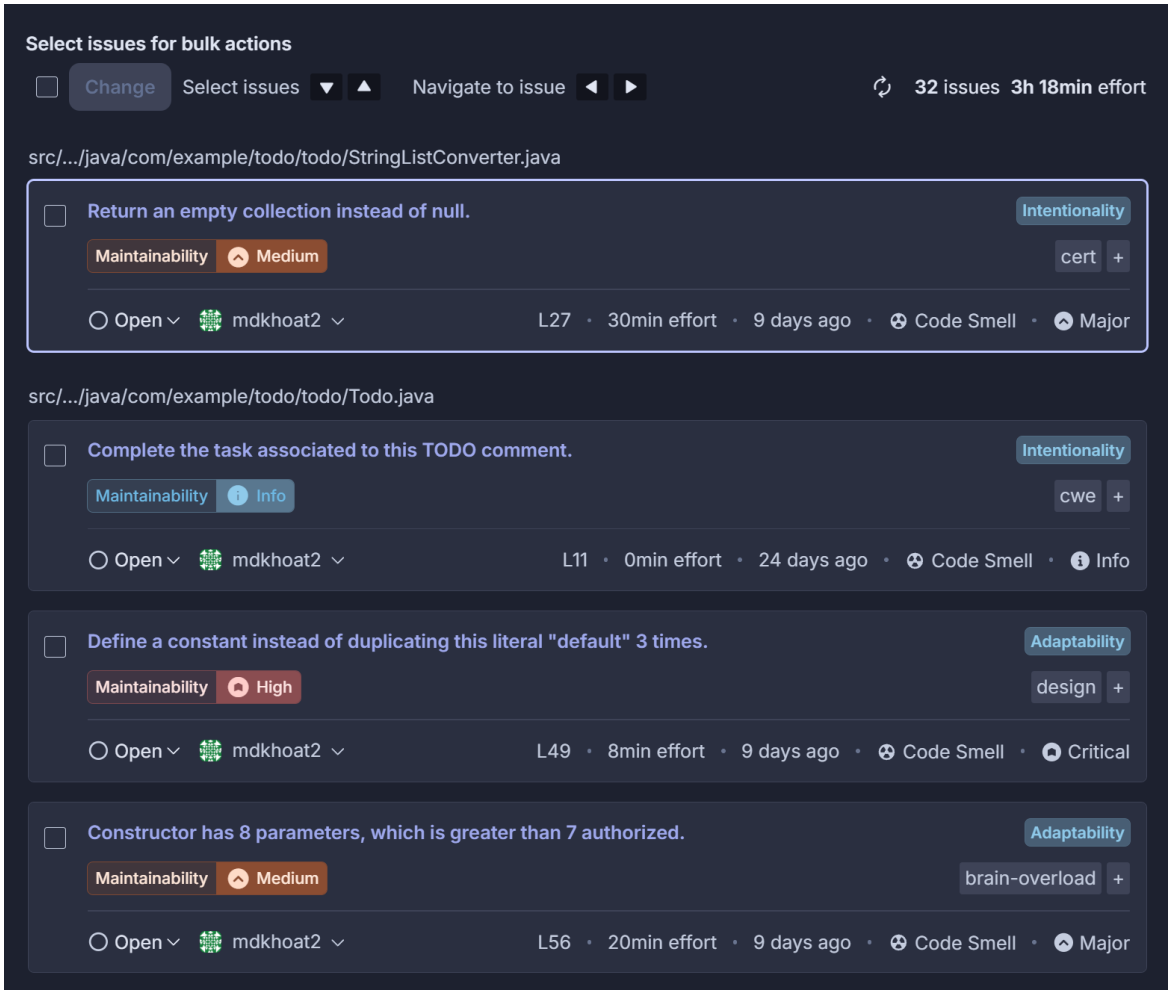

Figure 2: SonarQube's code smell. Each issue contains a brief description of what it is, where to find the issue, the impact of it, and how to fix it.

while systems like SonarQube uphold the foundations of deterministic quality enforcement. Together, they create an ecosystem where software quality is maintained continuously, intelligently, and collaboratively.

5 Conclusion

Continuous Integration helps teams find problems early, keep the main branch stable, and work more smoothly together. By automating builds and tests, CI reduces the time spent on debugging and makes it easier to maintain high code quality. For large teams or fast-moving projects, CI is an essential practice that supports safe and reliable development.
CI also prepares the foundation for Continuous Deployment, where tested code can be delivered or deployed automatically. Together, CI and CD help teams release software faster, with fewer errors, and with higher confidence in every change.

References

[1] Autonomous validation for the AI era — circleci.com. https://circleci.com/.

[2] Cloudflare Pages — pages.cloudflare.com. https://pages.cloudflare.com/.

[3] Code Quality, Security & Static Analysis Tool with SonarQube — sonarsource.com. https://www.sonarsource.com/products/sonarqube/.

[4] Get started with GitLab CI/CD | GitLab Docs — docs.gitlab.com. https://docs.gitlab.com/ci/.

[5] GitHub Actions — github.com. https://github.com/features/actions.

[6] GitHub Copilot · Your AI pair programmer — github.com. https://github.com/features/copilot.

[7] GitHub flow - GitHub Docs. https://docs.github.com/en/get-started/using-github/github-flow.

[8] Jenkins — jenkins.io. https://www.jenkins.io/.

[9] Setting Up a Basic CI/CD Pipeline with Automated Build and Test Stages. https://dev.to/bankolejohn/setting-up-a-basic-cicd-pipeline-with-automated-build-and-test-stages-10ik.

[10] Simple, Flexible, Trustworthy CI/CD Tools - Travis CI — travis-ci.com. https://www.travis-ci.com/.

[11] What is CI? https://aws.amazon.com/devops/continuous-integration/.

[12] What is CI/CD? https://github.com/resources/articles/ci-cd.