T08 Software Management, Build Systems, and DevOps

Dr. Marco Elver
Systems Research Group
https://dse.in.tum.de/

Tutorial outline

- **Part I: Lecture summary**
  - **Q&A for the lecture material**
- **Part II:** Programming basics
- **Part III:** Homework programming exercises (Artemis)

Lecture overview

- **Part I:** Software configuration management
  - Source code management
  - Version control systems
  - Branch management
- **Part II:** Build systems
  - Task-based build systems
  - Artifact-based build systems
  - Distributed builds
  - Dependency management
  - Hermeticity
- **Part III:** Release management
  - Release planning
  - Software versioning
  - Software upgrades
- **Part IV**: Configuration *
  - Continuous Integration
  - Continuous Delivery
  - Continuous Deployment
  - Continuous Fuzzing

Software Configuration Management

Policies, processes, and tools for managing software systems

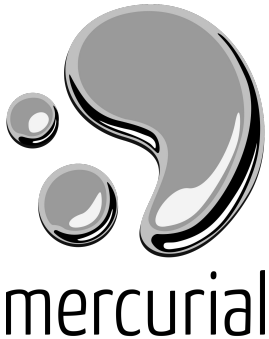
Pillars of Configuration Management:

1. **Version management:** Keeping track of multiple versions of system components
2. **System building:** Collecting components to create an executable system
3. **Change management:** Keeping track of requests for changes to delivered software from customers and developers
4. **Release management:** Preparing software for external release and keeping track of the system versions that have been released

📖 I. Sommerville, “Configuration management” (Chapter 25), In “Software Engineering,” 2015

Version Control Systems

- Version control system (VCS) keeps track of different versions of software
- Ensures that *concurrent* changes made by different developers do not interfere
- May track source code, binaries, assets, etc.
- All versions of components and metadata tracked in a **repository**



Centralized Version Control Systems

- Centralized **repository:** single repository stores all version of components being developed
- Developer only has local **workspace(s),** which are snapshots of a given repository state allowing modification of components
- **Examples**
  - Subversion: subversion.apache.org – open source, still widely used
  - Perforce: www.perforce.com – proprietary, mostly enterprise use
  - Concurrent Versions Systems (CVS): www.nongnu.org/cvs/ – open source, no longer recommended for new projects

Distributed Version Control Systems

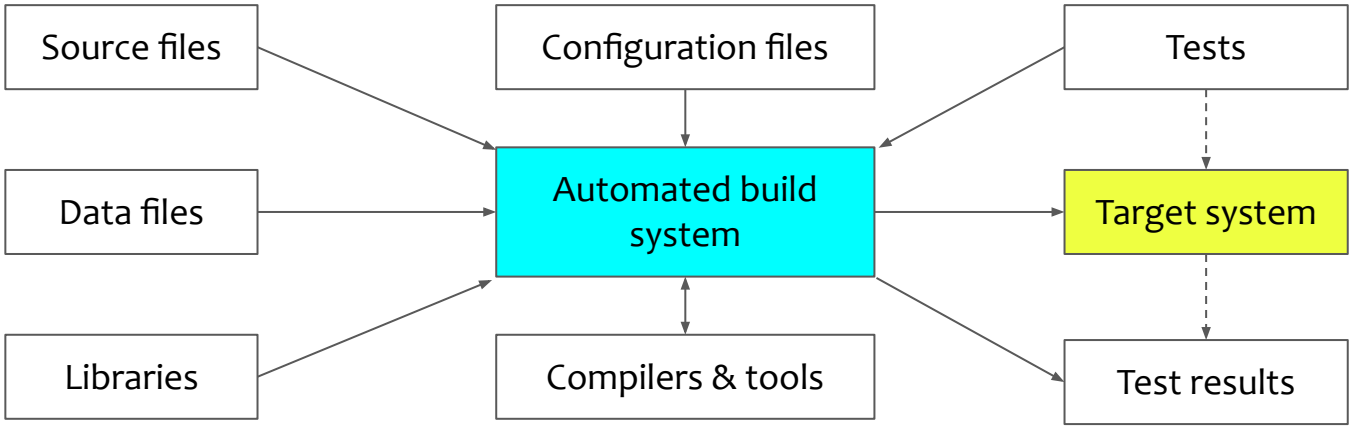
- Distributed **repository:** multiple replicas of repository exist concurrently, not necessarily synchronized (i.e. replicas can be in different states)
- Developer has a local copy of a **repository** snapshot, along with local **workspace(s)** to modify components
- **Examples**
  - Git: git-scm.com – open source, one of the most popular DVCS
  - Mercurial: www.mercurial-scm.org – open source
  - Darcs: darcs.net – open source
  - BitKeeper: www.bitkeeper.org – started proprietary, now open source, influenced creation of Git

💡 Check out Git's man pages for utilities:

```
git help
```

Build Systems

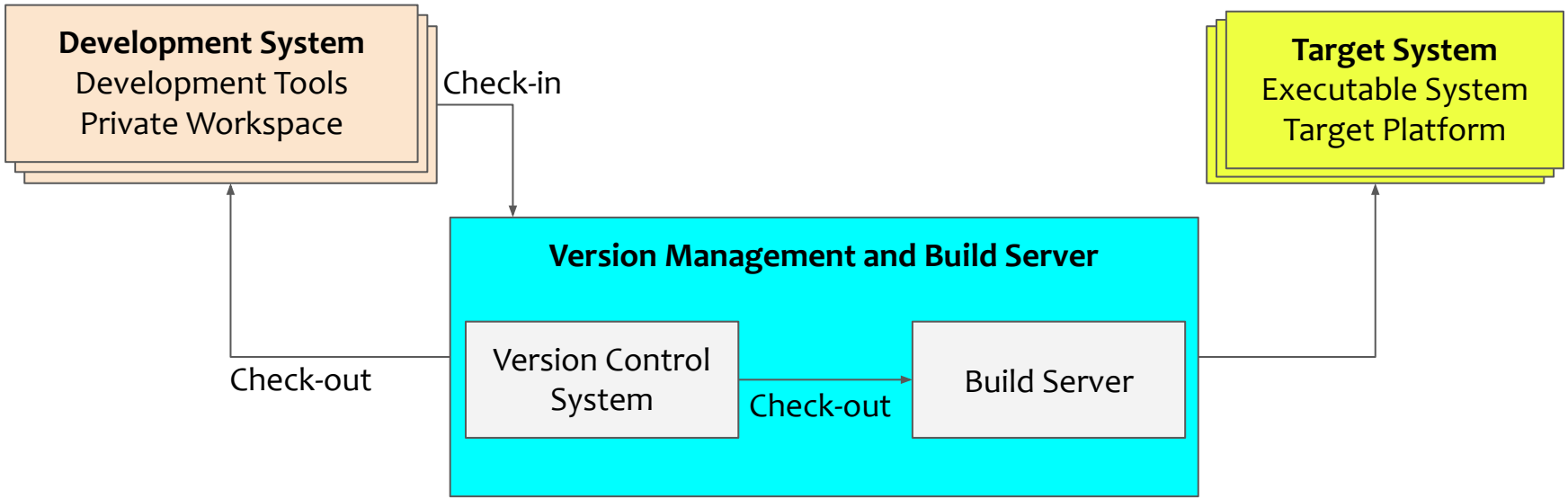
Assemble and create complete & executable system



📖 I. Sommerville, “System building” (Chapter 25.5), In “Software Engineering,” 2015

Build Systems

Build system must seamlessly account for different platforms



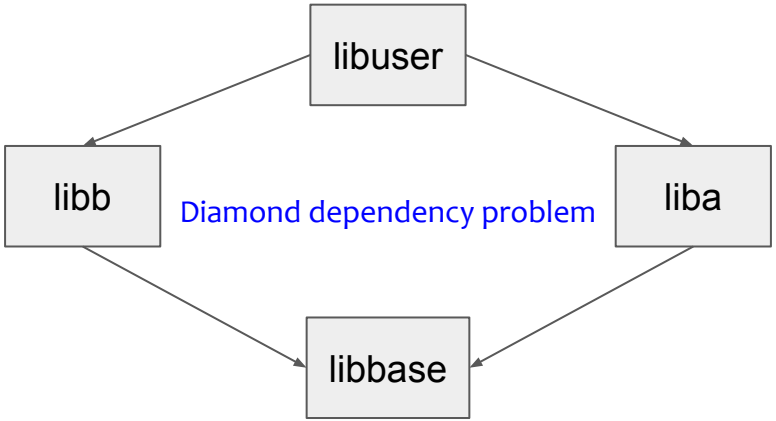
📖 I. Sommerville, “System building” (Chapter 25.5), In “Software Engineering,” 2015

Build systems

- Task-based Build Systems
  - Fundamental unit of work a “task”
  - Each “task” runs a set of commands
  - e.g., Ant, Maven, Gradle, Grunt, Rake, Makefiles
- Artifact-based Build Systems
  - Declarative build scripts describe “what” (not “how”) to build
  - Build system responsible for how to build artifacts
  - e.g., Bazel, Buck, Pants
- Hermetic Build Systems
  - Given the same inputs and configuration, it always returns the same output
  - Requires isolating build process from any inputs that may change arbitrarily
  - e.g., current datetime or Embedding unique identifiers in binary
- Distributed Build Systems
  - Distributed builds can scale builds across thousands of build servers
  - Remote caching
  - Remote execution

Dependency Management

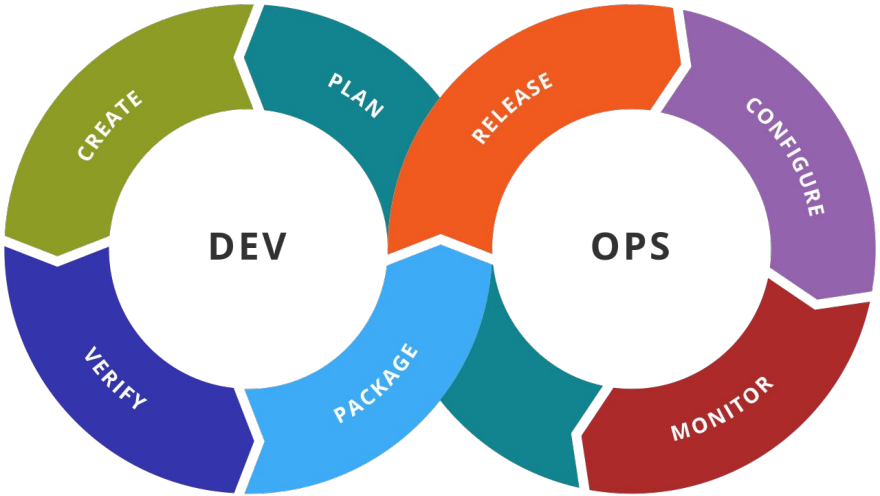
- Management of complex graph of dependencies.
  - How to deal with version incompatibilities? Example: libbase makes backwards incompatible change, and liba has to be updated as well.
  - How to deal with dependencies on different versions of same library? Example: liba depends on v1 of libbase, and libb depends on v2 of libbase.



📖 T. Winters, “Dependency management” (Chapter 27), In “Software Engineering at Google,” 2020

Release Management

Release management *covers the entire software development lifecycle: development, testing, releases, updates*



Release Management

Release Management Goals

- Increase release frequency to deliver new features and enhancements to users
- Reduce bottlenecks in the workflow by streamlining processes and collaboration
- Shorten feedback loops to gather feedback and incorporate into future releases
- Limit unplanned work by ensuring a well-defined release plan and scope
- Reduce defects and production incidents by testing & quality assurance
- Allow teams to add value with features and updates that align with user needs

Release Management & Agile Development

Agile principles emphasize the delivery of deployable code at any time.
Continuous Delivery (CD) is a critical aspect of agile development.

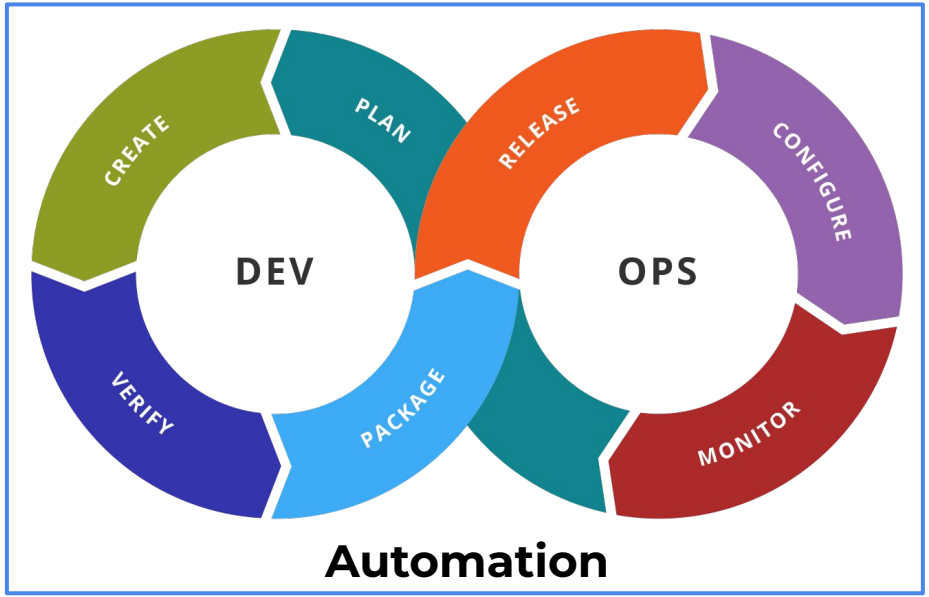
Effective Agile Release Management requires at least:

- **Deployment Automation:** Use of automated tools and processes to remove bottlenecks, reduce manual effort (**CI/CD pipeline**)
- **DevOps mindset:** Collaboration, communication, and coordination between development, operations, and other stakeholders involved in the release

Deployment Automation

Process of automating steps involved in deploying software

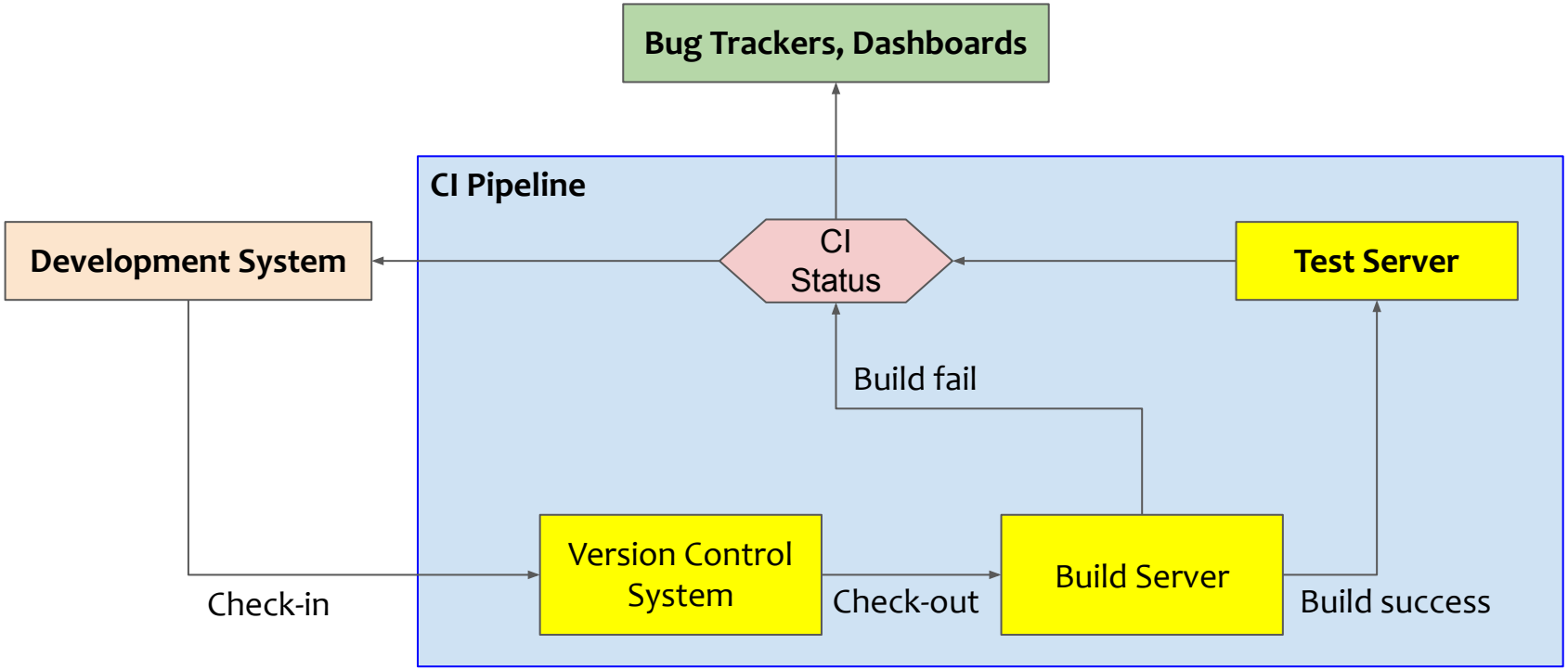
- Manual deployments are *time-consuming, error-prone, and inconsistent*.
- Deployment automation helps streamline the process (or parts of it).



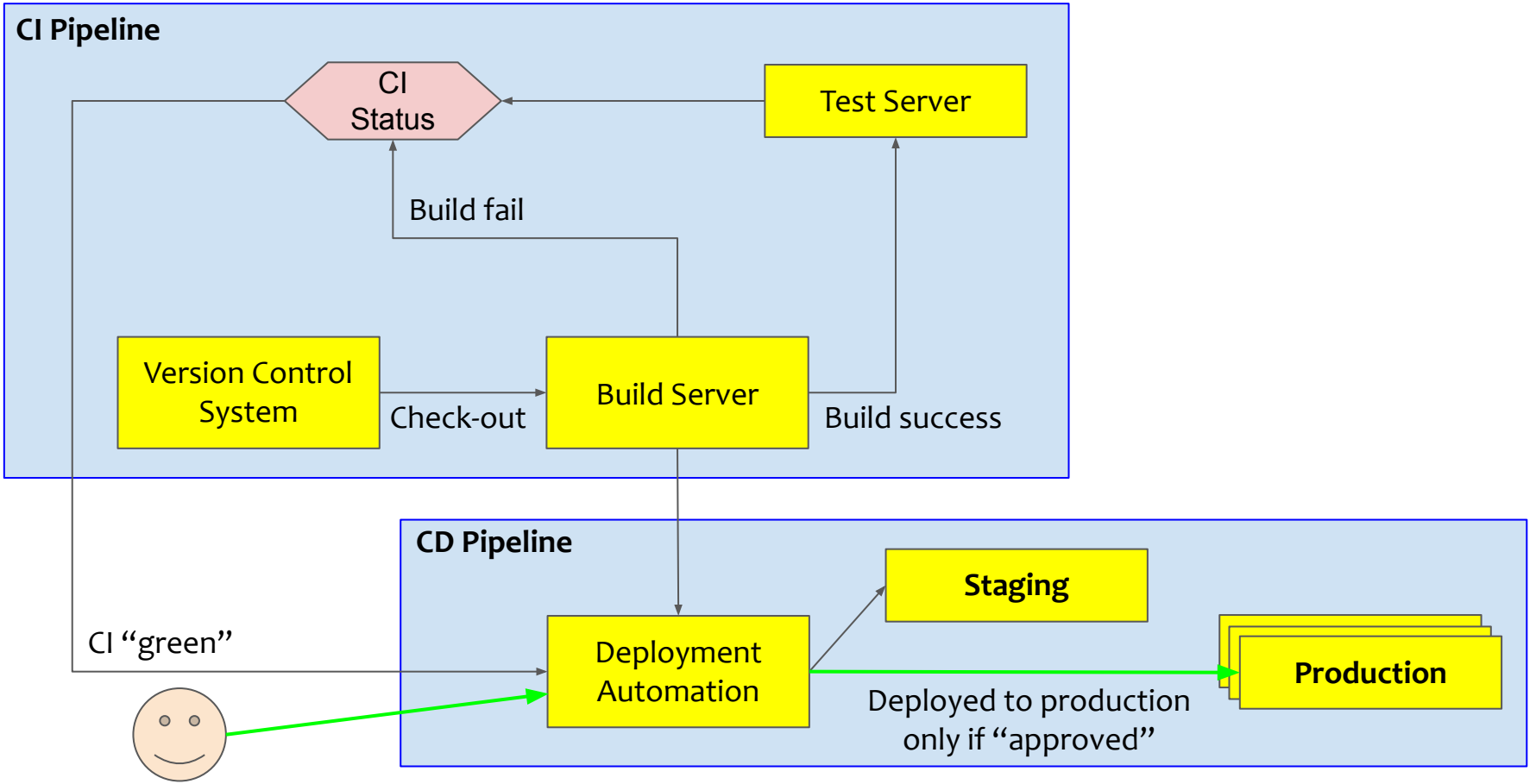
Deployment Automation Processes

- Continuous Integration
  - regularly merging code changes into a shared repository and performing automated builds and tests
- Continuous Delivery
  - ensures software is always in a releasable state, ready for deployment
- Continuous Deployment
  - every successful build and test automatically gets deployed to production
- Continuous Fuzzing
  - test programs with random, unexpected, inputs at scale, integrated with a CI pipeline

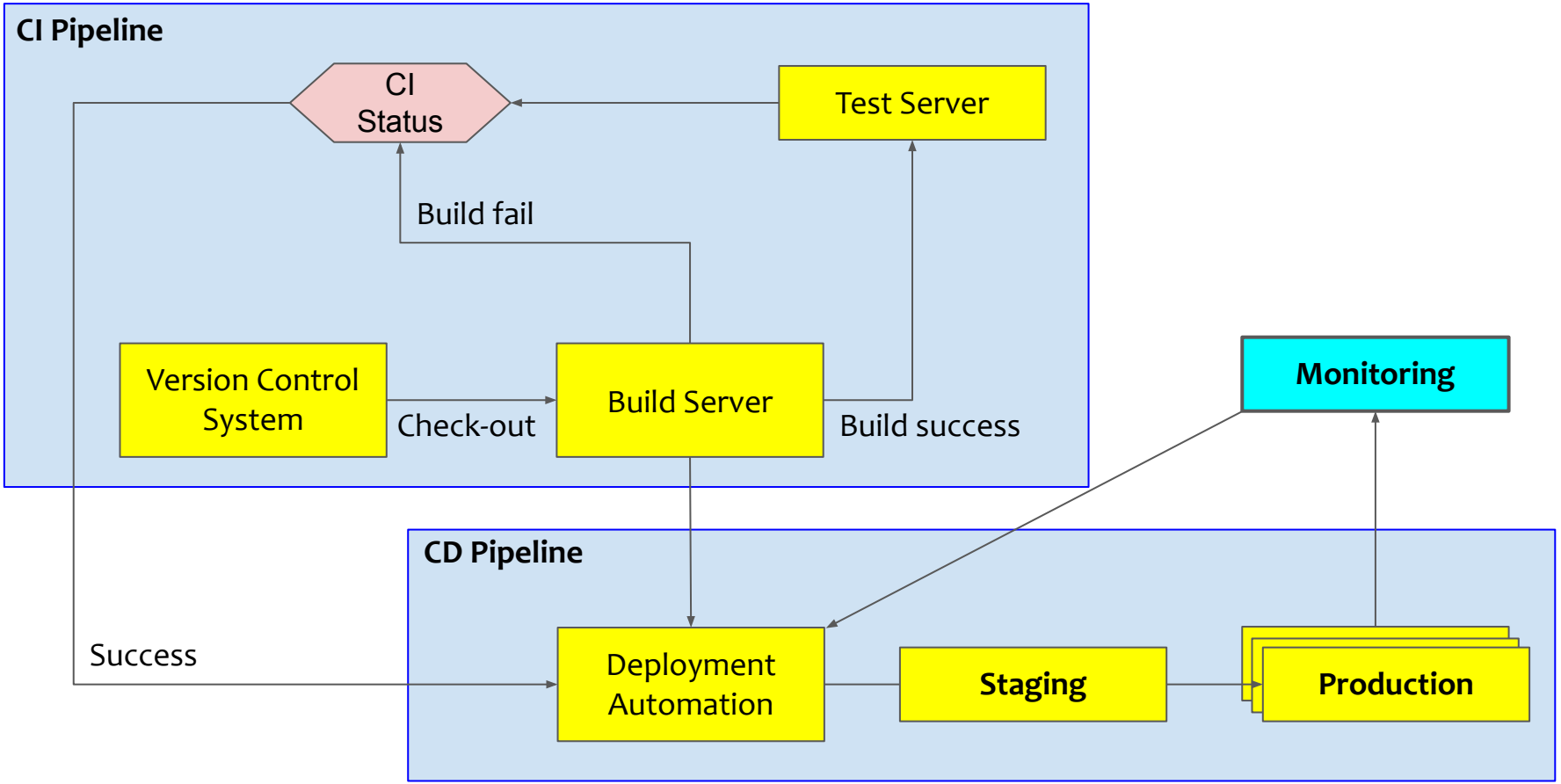
Continuous Integration



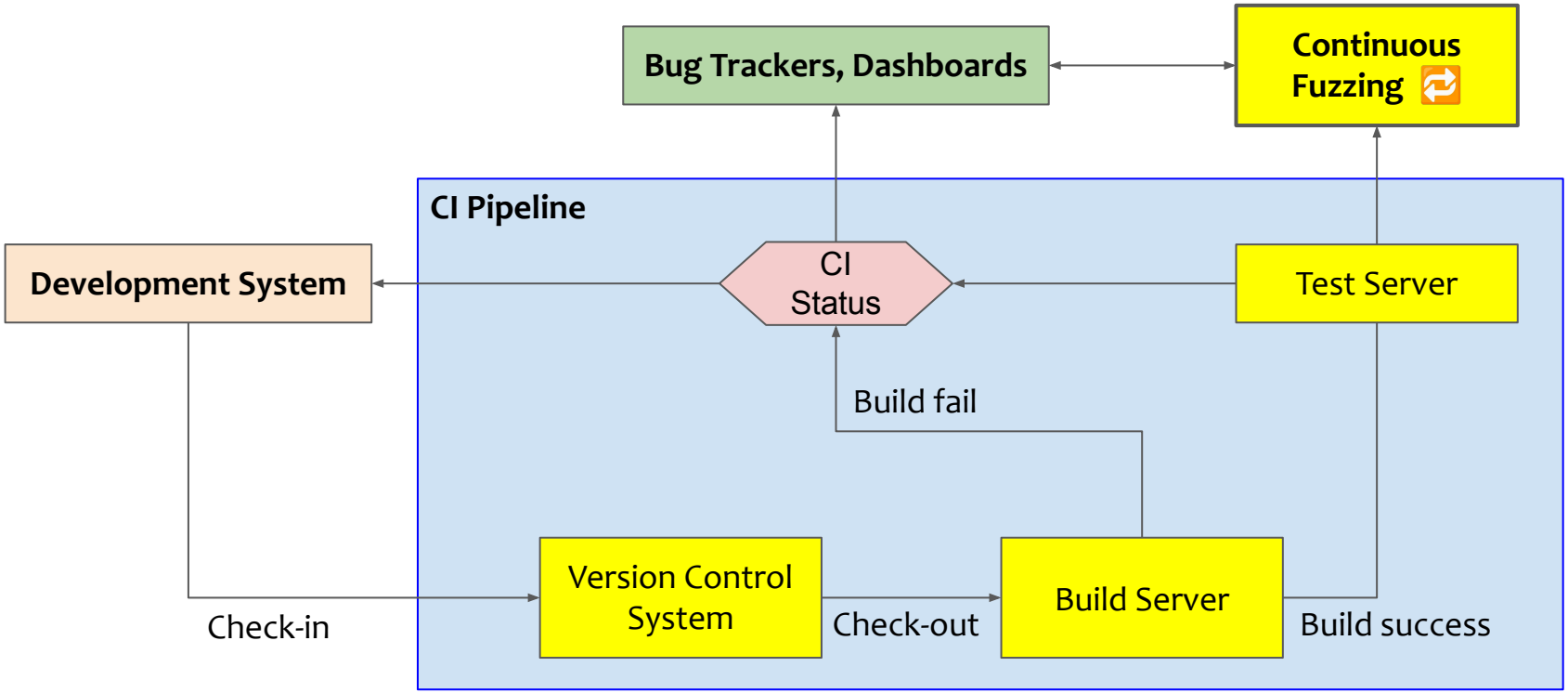
Continuous Delivery



Continuous Deployment



Continuous Fuzzing



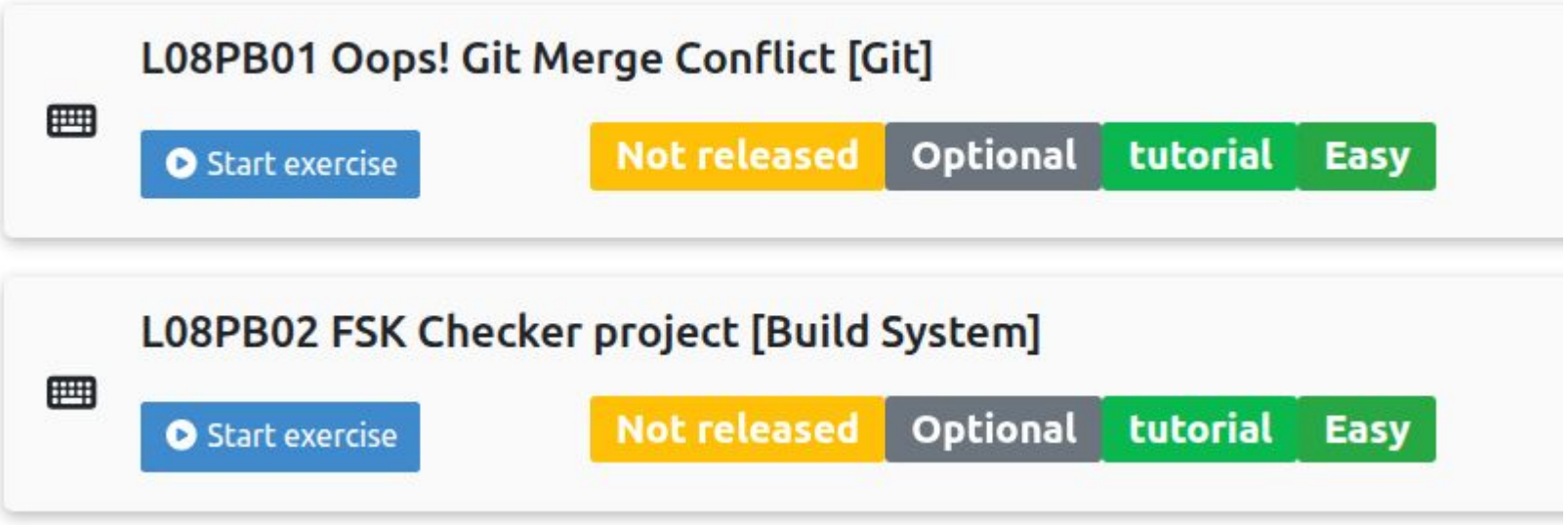
Tutorial outline

- ~~**Part I:** Lecture summary~~
  - ~~Q&A for the lecture material~~

- **Part II:** **Programming basics**

- **Part III:** Homework programming exercises (Artemis)

Programming Basics (PB) exercises



L08PB01 Oops! Git Merge Conflict [Git]

Goals

- How to **check out** to **branches**
- How to merge **branches**
- Resolution of **merge conflicts**
- Experience how **git merge** conflicts emerge and how to to resolve them

Objectives

- **Checkout** to each of the 5 feature **branches**
- Find the correct implementation and **merge it into the main branch**
- Resolve the **Git merge conflicts**

L08PB01 Oops! Git Merge Conflict [Git]

- Make sure, you configured your Git username and email
  - `git config --global user.email "YOUR EMAIL"`
  - `git config --global user.name "YOUR NAME"`

- Execute the setup script (this will create the branches)
  - `python3 create_branches.py`

L08PB01 Oops! Git Merge Conflict [Git]

Useful commands

- List all branches
  - `git branch`
- Checkout to a branch
  - `git checkout BRANCH_NAME`
  - Or create and switch to a branch: `git checkout -b BRANCH_NAME`
- Merge branch
  - `git merge BRANCH_NAME`
  - This merges `BRANCH_NAME` into the current **branch**

L08PB02 FSK Checker project [Build System]

Goals

- **Understand** language-agnostic **build systems**
- **Experience** building several libraries/binaries with a common build system
- **Combine** them into the final build result

Objectives

- Use the **Bazel build system** to build a Java project: **FSK Checker**
- **FSK Checker** binary depends on two libraries
- Bazel supports to incrementally build and link against those libraries

L08PB02 FSK Checker project [Build System]

Background

- FSK ratings indicate the appropriate age groups for viewers, helping parents and audiences make informed decisions about the content they or their children are about to watch

Tasks

- Build the **`:date_of_birth`** package from **`src/main/java/math`**
- Build the **`:movie`** library from **`Movie.java`**
- Build the **`:fsk_checker`** executable by linking **`Main.java`** with **`:data_of_birth`** and **`:movie`**

L08PB02 FSK Checker project [Build System]

Build System: Bazel

- **Bazel** is an open-source build system tool like **Maven, Gradle** etc.,
- **Bazel** supports projects in multiple languages and builds outputs for multiple platforms
- More information here - https://bazel.build/about/intro

Installation

- **Bazelisk** is the recommended way of installing **Bazel**
- It automatically downloads and installs the appropriate version of Bazel
- Make sure that you have Bazel **version 7 or later** installed
- **Install** Bazelisk from here - https://github.com/bazelbuild/bazelisk
- *Hint: If you run into issues, try using Java 21 as Bazel only has toolchains for up to this version*

Prerequisites

- Bazel >= 7.0 for Bzlmod (MODULE.bazel)
- Java <= 21
  - If your system uses JDK < 21 add `--java_runtime_version=remotejdk_21`

L08PB02 FSK Checker project [Build System]

Bazel example project

```
bazel-tutorial
├── BUILD
├── src
│   └── main
│       └── java
│           └── com
│               └── example
│                   ├── Calculator.java
│                   └── Main.java
└── MODULE.bazel (replaces WORKSPACE)
```

- The MODULE.bazel file helps recognise the directory and its contents as a Bazel workspace and lives at the root of the project's directory structure.
  - *Note: In Bazel 7 WORKSPACE was replaced by* MODULE.bazel
- One or more BUILD files, which tell Bazel how to build different parts of the project

L08PB02 FSK Checker project [Build System]

File bazel-tutorial/BUILD

```
// Defines a Java binary target that compiles and runs the Main class
java_binary(
    name = "Calculate", // Name of the binary target
    srcs = ["src/main/java/com/example/Main.java"], // Source file containing main()
    main_class = "com.example.Main", // Fully qualified name of the main class
    deps = [":calculator"], // Dependency on the "calculator" library target below
)

// Defines a Java library target that can be reused in other targets
java_library(
    name = "calculator", // Name of the library target
    srcs = ["src/main/java/com/example/Calculator.java"], // Source file(s)
)
```

L08PB02 FSK Checker project [Build System]

- Build the Project

```
bazel build --java_runtime_version=remotejdk_17 //:fsk_checker
```

- The `//` part before the target label indicates the location of the BUILD file, in this case the root of the project - `//`

- Bazel builds the project and places the build outputs in `bazel-bin/` directory in the root of the project

```
bazel-bin/fsk_checker
```

L08PB02 FSK Checker project [Build System]

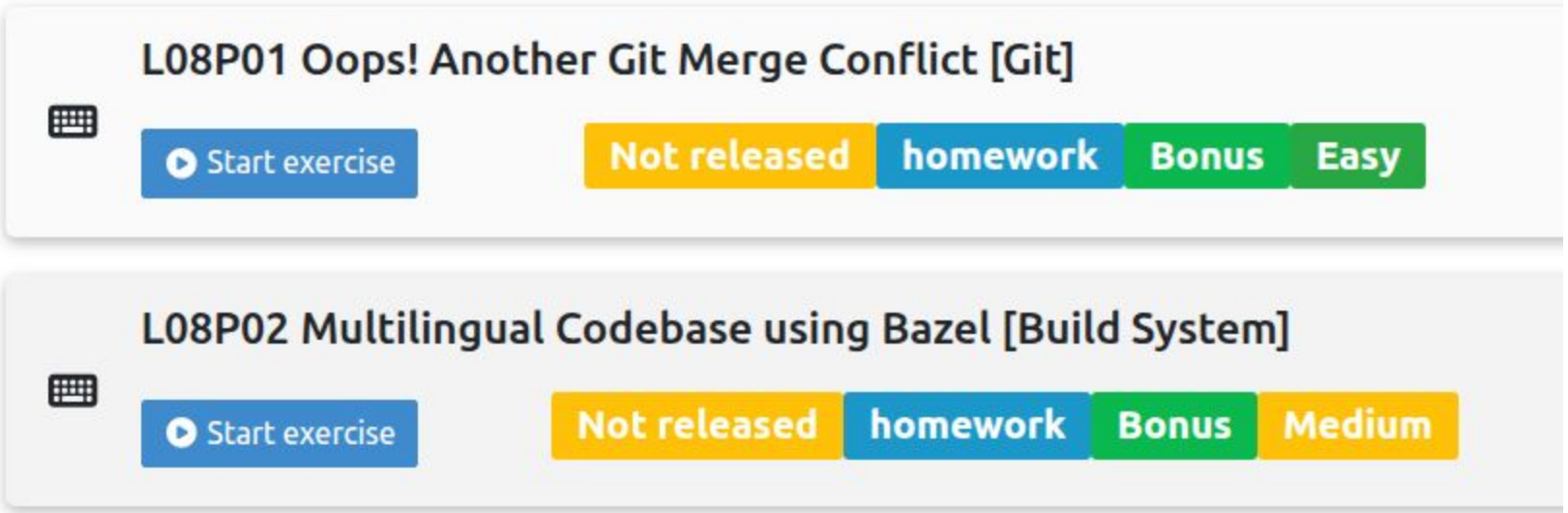
File BUILD.bazel

```
java_binary(
    name = "fsk_checker",
    srcs = ["src/main/java/Main.java"],
    main_class="src.main.java.Main",
    deps = ["//src/main/java/math:date_of_birth", ":movie"],

)

java_library(
    name = "movie",
    srcs = ["src/main/java/Movie.java"],
)
```

Tutorial outline

- ~~**Part I:** Lecture summary~~
  - ~~Q&A for the lecture material~~
- ~~**Part II:** Programming basics~~
- **Part III: Homework programming exercises (Artemis)**

Programming Homework Bonus (P) exercises



L08P01 Oops! Another Git Merge Conflict [Git]

Goals

- How merge conflicts emerge during collaboration
- How to resolve merge conflicts using Git
- The role of merge commits in conflict resolution

Objectives

- Create and commit changes locally
- Simulate a real-life merge conflict scenario
- Resolve a merge conflict by editing the code meaningfully
- Commit the resolved version and push it to the remote repository

L08P01 Oops! Another Git Merge Conflict [Git]

1. Create a class Hello with System.out.println("Hello World!")
2. Stage, commit (feat: add Hello class), and push
3. Update message to "Hello EIST!" and commit again
4. Try to push and observe the conflict error
5. Pull latest changes → resolve conflict by updating to "Hello EIST 2025!"
6. Commit (fix: resolve merge conflict) and push

NOTE : follow steps **in exact order!**

L08P01 Oops! Another Git Merge Conflict [Git]

Use Conventional Commits

- feat: → new feature (e.g., feat: add Hello class)
- fix: → bug fix or conflict resolution (e.g., fix: resolve merge conflict)
- chore: → minor changes, like formatting or comments

Structure

<type>: <short, meaningful description>

- Keep it clear and concise
- Use imperative mood (e.g., “add”, not “added”)
- Avoid vague messages like “changes” or “update”

L08P02 Multilingual Codebase using Bazel [Build System]

Goals

- Understand how to use **Bazel** to build multilingual (Java + Python) systems
- Learn how to manage dependencies using **MODULE.bazel** and pip_parse
- Apply the **Facade Pattern** to unify microservice interactions from the client side

Objectives

- Use Bazel to:
  - Build and run Python Flask microservices (py_binary)
  - Build Java components (java_library, java_binary)
  - Integrate third-party dependencies via pip_parse and Maven
- Link all components to execute an **end-to-end functional system**

L08P02 Multilingual Codebase using Bazel [Build System]

Client Side (Java)

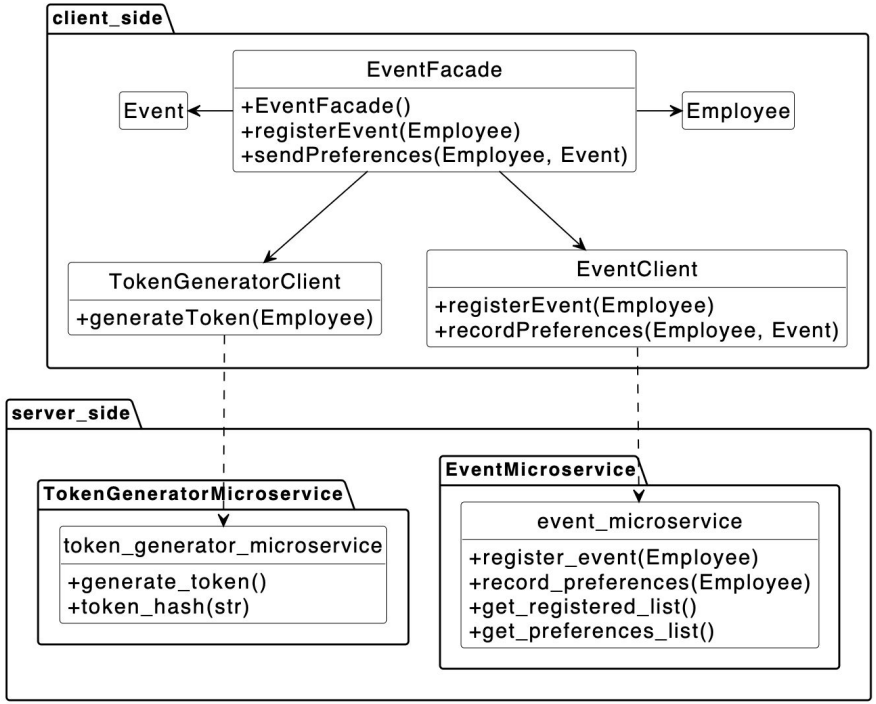
- EventFacade: entry point that orchestrates all operations
- EventClient: communicates with EventMicroservice
- TokenGeneratorClient: communicates with TokenGeneratorMicroservice
- Domain classes: Employee, Event

Server Side (Python / Flask)

- EventMicroservice
  - register_event(), record_preferences(), get_registered_list(), get_preferences_list()
- TokenGeneratorMicroservice
  - generate_token(), token_hash()

L08P02 Multilingual Codebase using Bazel [Build System]

UML Diagram of the client-side Facade Pattern and server-side Microservices:



L08P02 Multilingual Codebase using Bazel [Build System]

Install Bazelisk

- Use Bazel version 7 or newer
- Add to .bazelrc: `common --enable_bzlmod`
- Set up MODULE.bazel with Python dependencies
- Note: pip.parse hub_name and use_repo names must match
- Run for lock file:

```
touch server-side/requirements_lock.txt
bazel run //server-side:requirements.update
```

Microservices (py_binary)

Define BUILD in subdirectories

- `token_generator_microservice`
- `event_microservice`

L08P02 Multilingual Codebase using Bazel [Build System]

Java BUILD Targets

- `java_library(name = "event-client")`
- `java_library(name = "token-generator-client")`
- `java_library(name = "event-facade")`

Final Binary

- `java_binary(name = "client")`
- Set `main_class = <your.main.class.path>`

Hint: Why Facade Pattern?

→ To hide complexity and centralize microservice interactions

L08P02 Multilingual Codebase using Bazel [Build System]

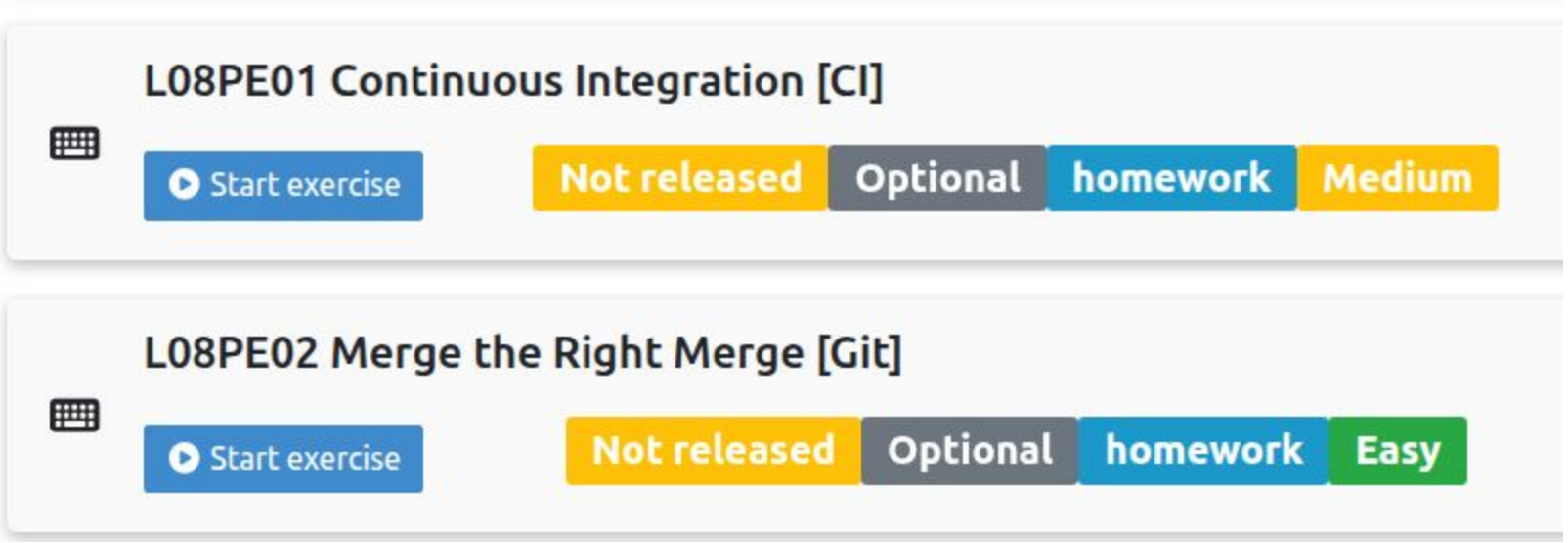
Makefile Commands

- `make run_event_microservice`
- `make run_token_generator_microservice`
- `make run_client`

⚠️ Notes

- Run each service in a **separate terminal window**
- If you get permission errors:
  `chmod +x bazel-bin/your_target`

Programming Extras (PE) exercises



L08PE01 Continuous Integration [CI]

Goals

- **Feel** a real CI loop: test → build → package
- **Practice** Gradle plugins & **dependency management**

Objectives

- **Fix** failing UniversityAppTest
- **Wire** Shadow **plugin**
- **Add** Log4j INFO on startup

L08PE01 Continuous Integration [CI]

- Run app, skim codebase
- **Execute** tests
  - **IDE** first, then `./gradlew test`
- **Edit build.gradle.kts**
  - Add shadow plugin
  - Set main class
  - Configure build task
- **Add** Log4j **deps**

L08PE01 Continuous Integration [CI]

Q: Why Gradle + Shadow?

A: Standard JVM builder; fat-JAR ships anywhere

Q: Why Log4j?

A: Lightweight visibility; CI pipelines can flag errors early

Q: Why run tests twice?

A: IDE for quick feedback, Gradle for the same headless steps CI uses

L08PE02 Merge the Right Merge [Git]

Goals

- Understand how to **use Git branches** effectively in a team setting
- Learn to **identify correct code implementations** and merge responsibly
- Practice using git merge correctly, **not by copy-pasting**

Objectives

- Switch between branches (feature/merge-i)
- Evaluate different implementations of the merge function
- Identify the correct one by **running & testing the code**
- Merge it into main using git merge, then commit & push

L08PE02 Merge the Right Merge [Git]

- You're a software engineer on a team with 5 junior developers
- Each teammate has created a feature branch implementing a merge function for MergeSort.java
- Only **one branch contains the correct implementation**

Provided Files

- src/git/MergeSort.java

Branches created by

- python3 create_branches.py
- git branch
- # main, feature/merge-1 ... feature/merge-5

L08PE02 Merge the Right Merge [Git]

- List & checkout to each feature/merge-i branch
- Test the merge() function (feel free to modify main() for testing)
- Identify the **correct** implementation
- Merge it into main
  - git checkout main
  - git merge feature/merge-X
- Commit and push
  - git add .
  - git commit -m "feat: merge correct merge implementation"
  - git push

L08PE02 Merge the Right Merge [Git]

- **Do not copy-paste** code from one branch to another
  → You must use git merge

- A correctly sorted output **does not always** mean the logic is correct
  → Review the implementation, not just the result

- You can modify main() to test different cases

Homework programming exercises

- Explain the homework programming exercises in Artemis

https://artemis.cit.tum.de/