CSE 403 Software Engineering

Build systems &
Continuous Integration and Deployment

Outline

- **Build systems**
- **Continuous integration and deployment systems**
  - What are they?
  - How do they relate?
  - Best practices
  - Ideas to explore for your projects

What does a developer do?

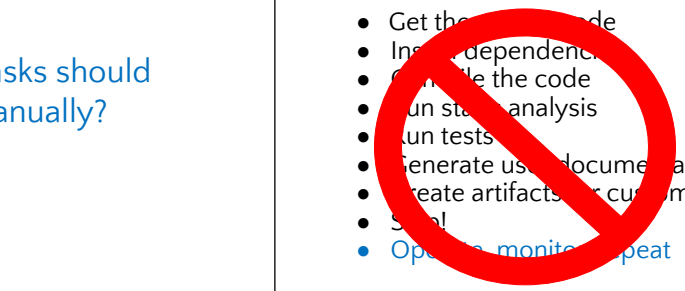
The code is written … now what?

- Get the source code
- Install dependencies
- Compile the code
- Run static analysis
- Run tests
- Generate user documentation
- Create artifacts for customers
- Ship!
- Operate, monitor, repeat

Which of these tasks should be handled manually?

What does a developer do?

The code is written … now what?



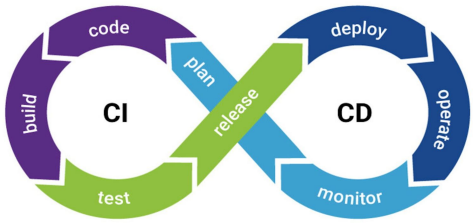
Which of these tasks should be handled manually?

NONE!

Instead, orchestrate with a tool

- **Build system**: a tool for automating compilation and other ~~tasks~~
- Is a component of a **continuous integration/deployment system**

✔Get the source code
✔Install dependencies
✔Compile the code
✔Run static analysis
✔Run tests
✔Generate user documentation
✔Create artifacts for customers
✔Ship!
✔Operate, monitor, repeat



These are all tasks!

Build systems: tasks

Tasks are code!

- Should be tested
- Should be code-reviewed
- Should be checked into version control

Build system best practices

- Automate, automate, automate everything!
- Always use a build tool (one-step build)
- Don't depend on anything that's not in the build file
- Use a CI tool to build and test your code on every commit
- Don't break the build!
  - Use pull requests to run CI run *before* committing to mainline

How can a build system help us?

1. **Dependency management**
   1. Identifies dependencies between files (including externals)
   2. Runs the steps in the right order
   3. Only runs the steps needed (due to dependency changes)

2. **Efficiency and reliability**
   1. Automates the build process, for any team member in any environment
   2. Formalizes the build process (no tribal knowledge)
   3. Eliminates the chance of errors
   4. Speeds up the process

The build configuration system

A build configuration (= buildfile):

- defines **tasks**
  - and external resources, such as libraries
- defines **dependencies** among tasks (= a graph)

A build system:

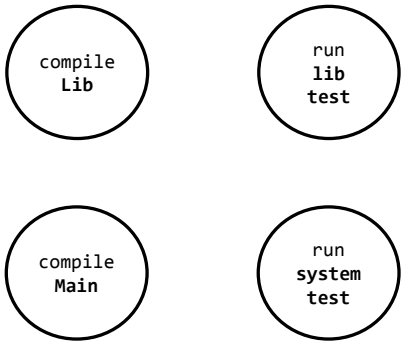
- **executes** the tasks (that are out of date)

Simple example code for dependency mgmt

```
% ls src/
   Lib.java
   LibTest.java
   Main.java
   SystemTest.java
```

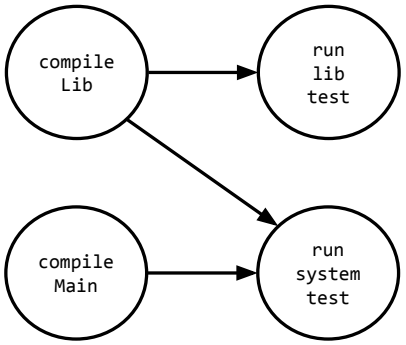

Build config: dependencies between tasks

```
% ls src/
   Lib.java
   LibTest.java
   Main.java
   SystemTest.java
```

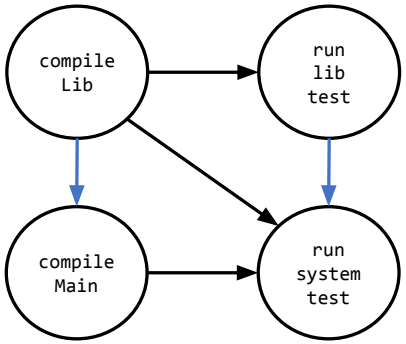


What are the dependencies between these tasks?
And why do I care?

Build config: dependencies between tasks

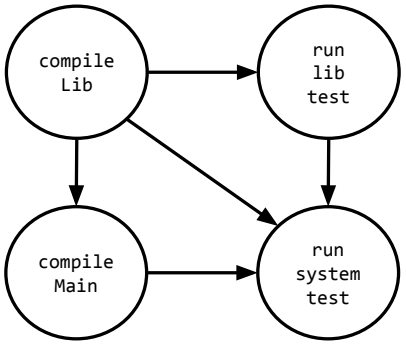


Build config: dependencies between tasks



Build config: dependencies between tasks

In what order should we run these tasks?



Build systems determine task order

Large projects have thousands of tasks

- Example: clone https://github.com/typetools/checker-framework, run `./gradlew tasks` or `./gradlew taskTree build`
- Dependencies between tasks form a directed acyclic graph (dag)
- Use topological sort to create an order for tasks
  - See Appendix (slides at end) for example

External code (libraries) also can be complex

- List all dependencies for reproducibility
  - A *hermetic build* is "insensitive to the libraries and other software installed on the build machine"[1]
- Build systems can manage external dependencies as well!

Dependency/package managers

Linux: apt, yum (snap is different)
Java: gradle, maven (artifacts at Maven Central)
JavaScript: npm
Python: pip
Ruby: RubyGems
Rust: cargo

The build configuration system

A build configuration (= buildfile):

- defines **tasks**
  - and external resources, such as libraries
- defines **dependencies** among tasks (= a graph)

A build system:

- **executes** the tasks (that are out of date)

Examples

https://github.com/plume-lib/plume-util/blob/master/build.gradle
CSE 403 staff Makefile

Example task: `gradle`

kind of rule

```
task reformat(type: Exec, dependsOn: getCodeFormatScripts, group: 'Format') {
   description 'Format the Java source code'
   // jdk8 and checker-qual have no source, so skip
   onlyIf { !project.name.is('jdk8') && !project.name.is('checker-qual') }
   executable 'python'
   doFirst {
       args += "${formatScriptsHome}/run-google-java-format.py"
       args += "--aosp" // 4 space indentation
       args += getJavaFilesToFormat(project.name)
   }
}
```


Example task: `gradle`

explicitly specified dependencies

```
task reformat(type: Exec, dependsOn: getCodeFormatScripts, group: 'Format') {
   description 'Format the Java source code'
   // jdk8 and checker-qual have no source, so skip
   onlyIf { !project.name.is('jdk8') && !project.name.is('checker-qual') }
   executable 'python'
   doFirst {
       args += "${formatScriptsHome}/run-google-java-format.py"
       args += "--aosp" // 4 space indentation
       args += getJavaFilesToFormat(project.name)
   }
}
```

Example task: `gradle`

```
task reformat(type: Exec, dependsOn: getCodeFormatScripts, group: 'Format') {
   description 'Format the Java source code'
   // jdk8 and checker-qual have no source, so skip
   onlyIf { !project.name.is('jdk8') && !project.name.is('checker-qual') }
   executable 'python'
   doFirst {
       args += "${formatScriptsHome}/run-google-java-format.py"
       args += "--aosp" // 4 space indentation
       args += getJavaFilesToFormat(project.name)
   }
}
```

code! (usually, following conventions is enough)

Example task: `bazel`

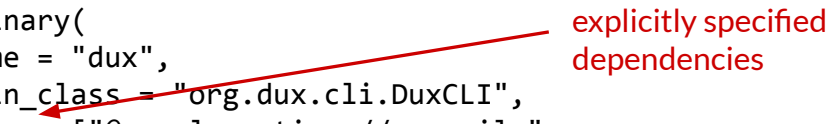
```
java_binary(
    name = "dux",
    main_class = "org.dux.cli.DuxCLI",
    deps = ["@google_options//:compile",
            "@checker_qual//:compile",
            "@google_cloud_storage//:compile",
            "@slf4j//:compile",
            "@logback_classic//:compile"],
    srcs = glob(["src/org/dux/cli/*.java",
                 "src/org/dux/backingstore/*.java"),
)
```


Example task: `bazel`

kind of rule

```
java_binary(
    name = "dux",
    main_class = "org.dux.cli.DuxCLI",
    deps = ["@google_options//:compile",
            "@checker_qual//:compile",
            "@google_cloud_storage//:compile",
            "@slf4j//:compile",
            "@logback_classic//:compile"],
    srcs = glob(["src/org/dux/cli/*.java",
                 "src/org/dux/backingstore/*.java"),
)
```


Example task: `bazel`




```
java_binary(
    name = "dux",
    main_class = "org.dux.cli.DuxCLI",
    deps = ["@google_options//:compile",
            "@checker_qual//:compile",
            "@google_cloud_storage//:compile",
            "@slf4j//:compile",
            "@logback_classic//:compile"],
    srcs = glob(["src/org/dux/cli/*.java",
                 "src/org/dux/backingstore/*.java"),
)
```

Example task: `bazel`

```
java_binary(
    name = "dux",
    main_class = "org.dux.cli.DuxCLI",
    deps = ["@google_options//:compile",
            "@checker_qual//:compile",
            "@google_cloud_storage//:compile",
            "@slf4j//:compile",
            "@logback_classic//:compile"],
    srcs = glob(["src/org/dux/cli/*.java",
                 "src/org/dux/backingstore/*.java"),
)
```

explicitly specified dependencies (also bazel tasks)

How to speed up the build

- Incrementalize - only rebuild what you have to
  - Compute hash codes for inputs to each task
    - Watch out: there are more inputs than you think
  - Before executing a task, check input hashes
  - If they have not changed since the last time the task was executed, skip it!
- Execute many tasks in parallel
- Cache artifacts (in the cloud)

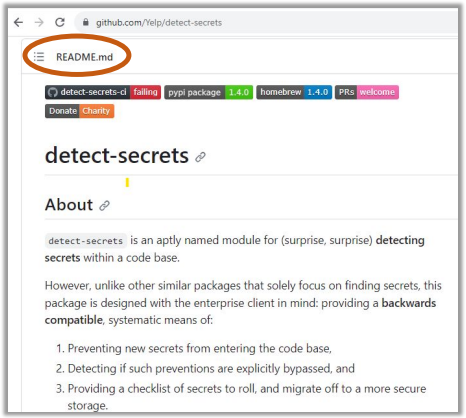
Static analysis

Can run before or after the compile step

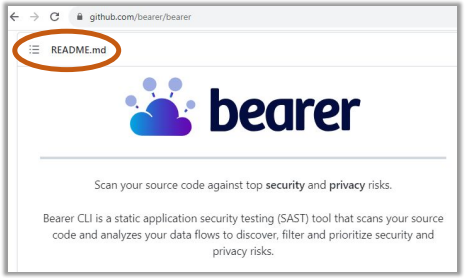
Examples:

- Credential scan
- Date scan
- Sensitive data scan
- Check formatting
- Linting
- Verification

Build systems: opportunity for static analysis



Could these types of static analysis tools be run earlier than CI?



Here's an example build system 'input'

Basic-Stats
build.gradle

Simple-C
Makefile
for the "make" build system

There are a *lot* of build systems

C, general: make, CMake

Java, etc.: gradle, sbt, maven, ant
Python: SCons
Ruby: rake

General: blaze, buck

A build configuration:
- defines **tasks**
- defines **dependencies** among tasks (a graph)

A build system:
- **executes** the tasks

Build system code may run at *graph construction time* or at *task execution time*

Assignment: evaluate and select a build system



Many other options!

Over to you to research



Java+		
	gradle	Open-source successor to **ant** and **maven**
	bazel	Open-source version of Google's internal build tool (blaze)
Python		
	hatch	Implements standards from the Python standard (uses TOML files, has PIP integration)
	poetry	Packaging and dependence manager
	tox	Automate and standardize testing
JavaScript		
	npm	Standard package/task manager for Node, "Largest software registry in the world."
	webpack	Module bundler for modern JavaScript applications
	gulp	Tries to improve dependency and packing

Outline

- Build systems
- **Continuous integration and deployment systems** ← We are here
  - What are they?
  - How do they relate?
  - Best practices
  - Ideas to explore for your projects

CI/CD: What's the difference?

Continuous Integration (CI)
- Automatically **test** upon each integration (≈ commit)
- Complements local developer workflows (may run different tests)
- **Goal:** find bugs quicker, improve quality



Continuous Deployment/Delivery (CD)
- Automatically pushes changes [to staging environment and then] to production
- **Goal:** users always use the latest version of the code



https://aws.amazon.com/devops/what-is-devops/

There are many CI tools

Jenkins

GitHub Actions

AWS CodePipeline

Azure Pipelines

Travis CI

GitLab

circleci

Bitbucket Pipelines

etc.

Assignment: Research, evaluate and choose a CI system

Continuous integration basics

- A CI **workflow** is **triggered** when an **event** occurs in your [shared] repo
  - Example events
    - Push a commit
    - Pull request
    - Issue creation
- A workflow contains **jobs** that run in a defined order
  - A job is like a shell script and can have multiple steps
  - Jobs run in their own vm/container called a **runner**
  - Example jobs
    - Run static analysis
    - Build, test
    - Deploy to production



Using GitHub Actions terminology; concepts span all CI systems

https://docs.github.com/en/actions

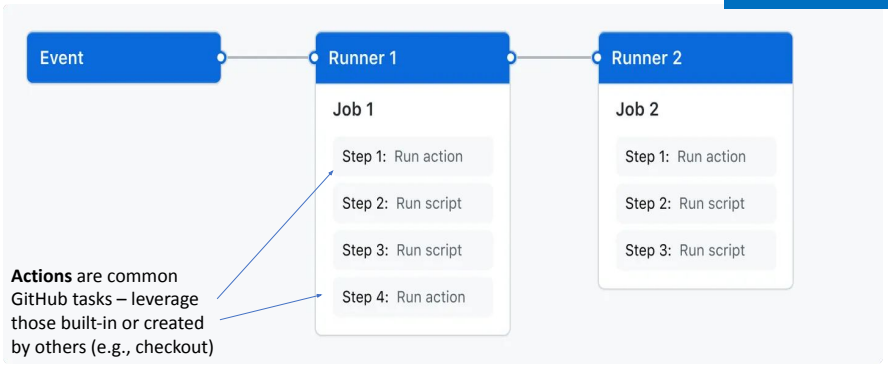
Demo

GitHub Actions:
https://github.com/plume-lib/plume-util/blob/master/.github/workflows/gradle.yml

Dependency examples:
https://app.circleci.com/pipelines/github/randoop/randoop/191/workflows/7d52ead9-c5c3-467d-87d4-5316d7de1692
https://app.circleci.com/pipelines/github/codespecs/daikon/97/workflows/5506fe7e-c466-43ee-b5c6-354d8189c97e

CI basics (w/ GitHub Actions)

What SW architecture is this using?



https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions

Let's try writing our own simple workflow

Follow along at:
https://github.com/alv880/UW-CSE403-Au23-Projects

Nice light starter tutorial – Automation Step by Step:
https://www.youtube.com/watch?app=desktop&v=ylEy4eLdhFs

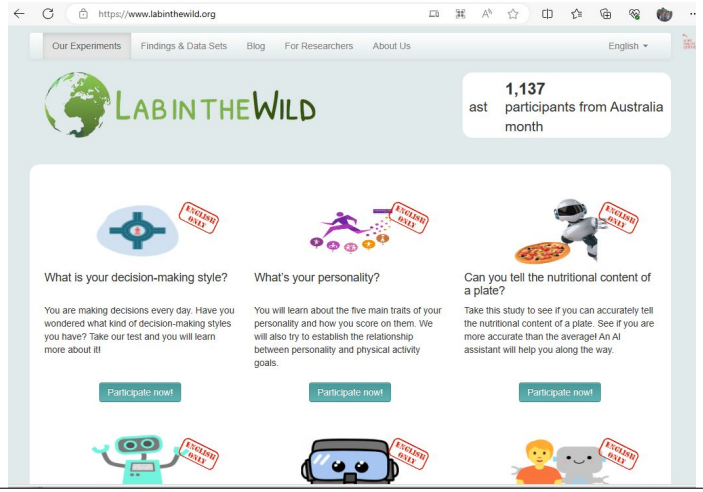
GitHub Actions Step by Step Tutorial

https://www.youtube.com/watch?v=ylEy4eLdhFs

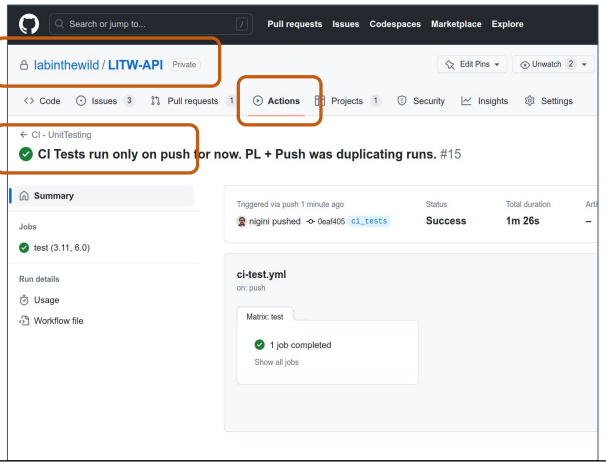
Example: CI at work at UW

Lab In The Wild is a research project drawing survey input from diverse community

Nigini Oliveira (researcher and 403 prof) provided this example



Example: CI with Github actions



Example: CI with Github actions

Unit tests are triggered on every push of new code

```yaml
name: CI - UnitTesting
on: [push]
jobs:
  test:
    runs-on: ubuntu-latest
    strategy: <2 keys>

    steps:
    - uses: actions/checkout@v3
    - name: Set up Python ${{ matrix.python-version }}
      uses: actions/setup-python@v3
      with: <1 key>
    - name: Set up MongoDB ${{ matrix.mongodb-version }}
      uses: supercharge/mongodb-github-action@1.8.0
      with: <1 key>
    - name: Install dependencies
      run: python3 -…tall hatch
    - name: Pre-fly setup
      run: cp $GITHU…GITHUB_ENV
    - name: Test with hatch
      run: |
        hatch run test:test
```

Workflow name

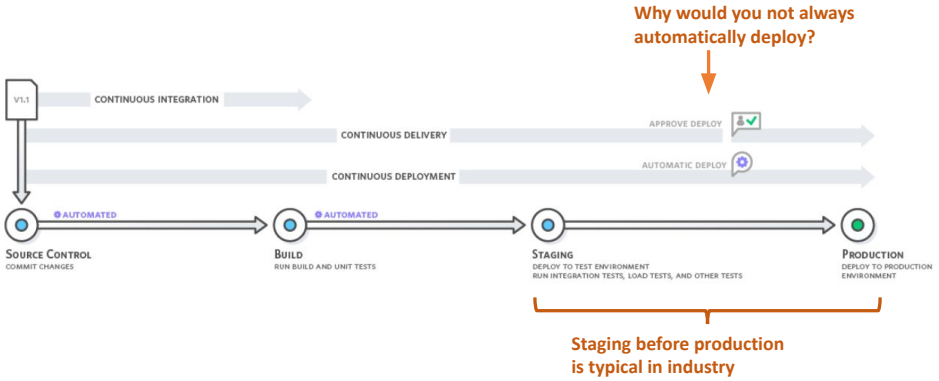
Trigger

Linux OS environment

Code reuse with established “actions”

One command to run test suite

Continuous delivery/deployment basics

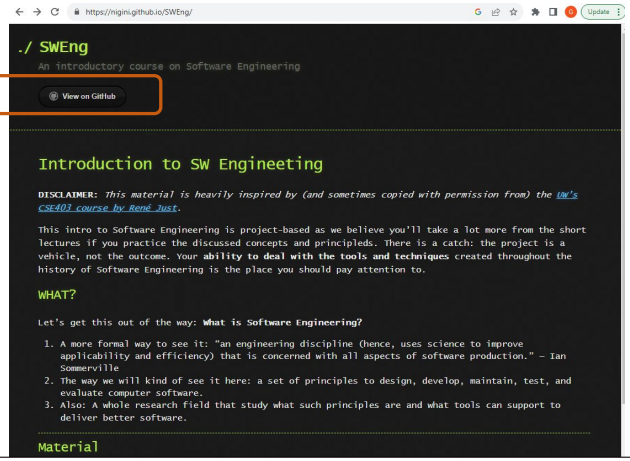


What is Continuous Delivery? – **Amazon Web Services**

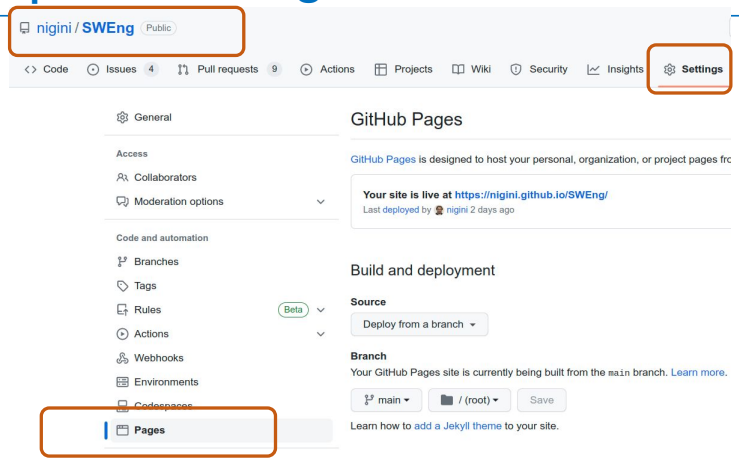
Example: CD with GitHub Pages

Spring ‘23 class hosted their 403 class website on GitHub pages

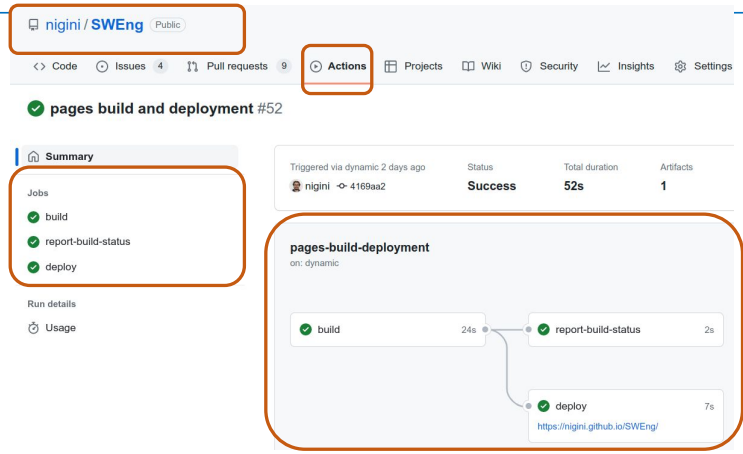
Used CD so that updates triggered publishing the website update



Example: CD configuration

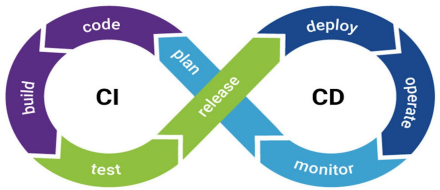


Example: CD configuration



Build & CI – Remember these best practices

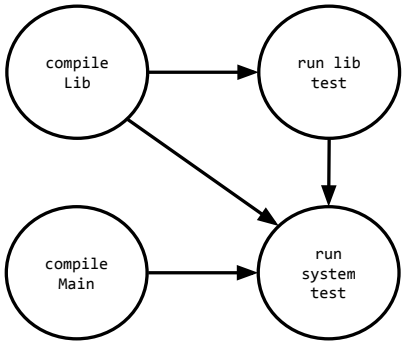
- Automate everything!
- Always use a build tool (one-step build)
- Use CI to test your code on every commit
- Don't depend on anything that's not in the build file (hermetic)
- Don't break the build!



Appendix – Topological sort example

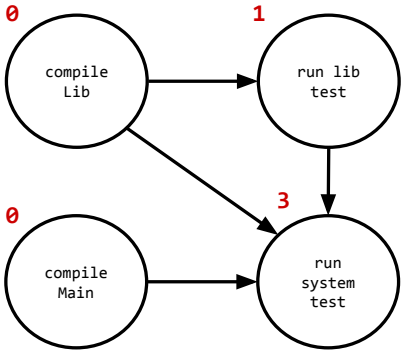
- Build tools use a topological sort to create an order to compiles
  - Order nodes such that all dependencies are satisfied
  - Implemented by computing indegree (number of incoming edges) for each node
  - No dependencies go first and open door to the others

Build systems: topological sort

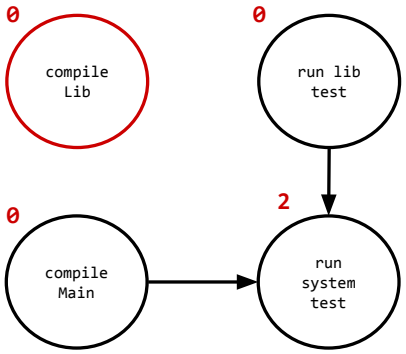


What's the indegree of each node?

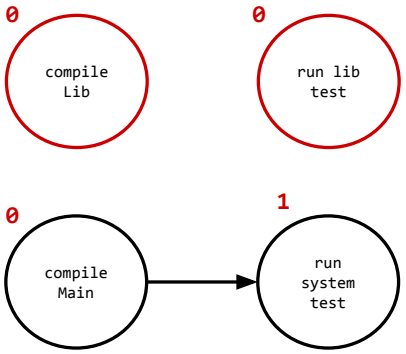
Build systems: topological sort



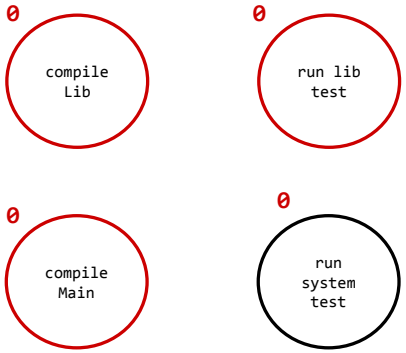
Build systems: topological sort



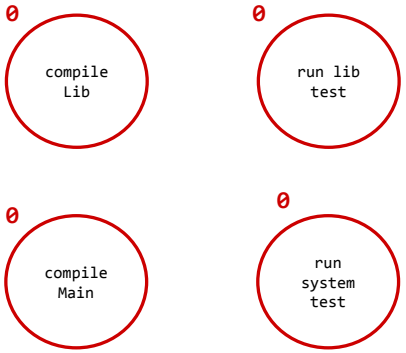
Build systems: topological sort



Build systems: topological sort



Build systems: topological sort



Build systems: topological sort

Valid sorts:

1. compile Lib, run lib test, compile Main, run system test

2. compile Main, compile Lib, run lib test, run system test

3. compile Lib, compile Main, run lib test, run system test

Which is preferable?

