Build Systems and CI/CD

Kristóf Marussy, Ármin Zavada, Oszkár Semeráth

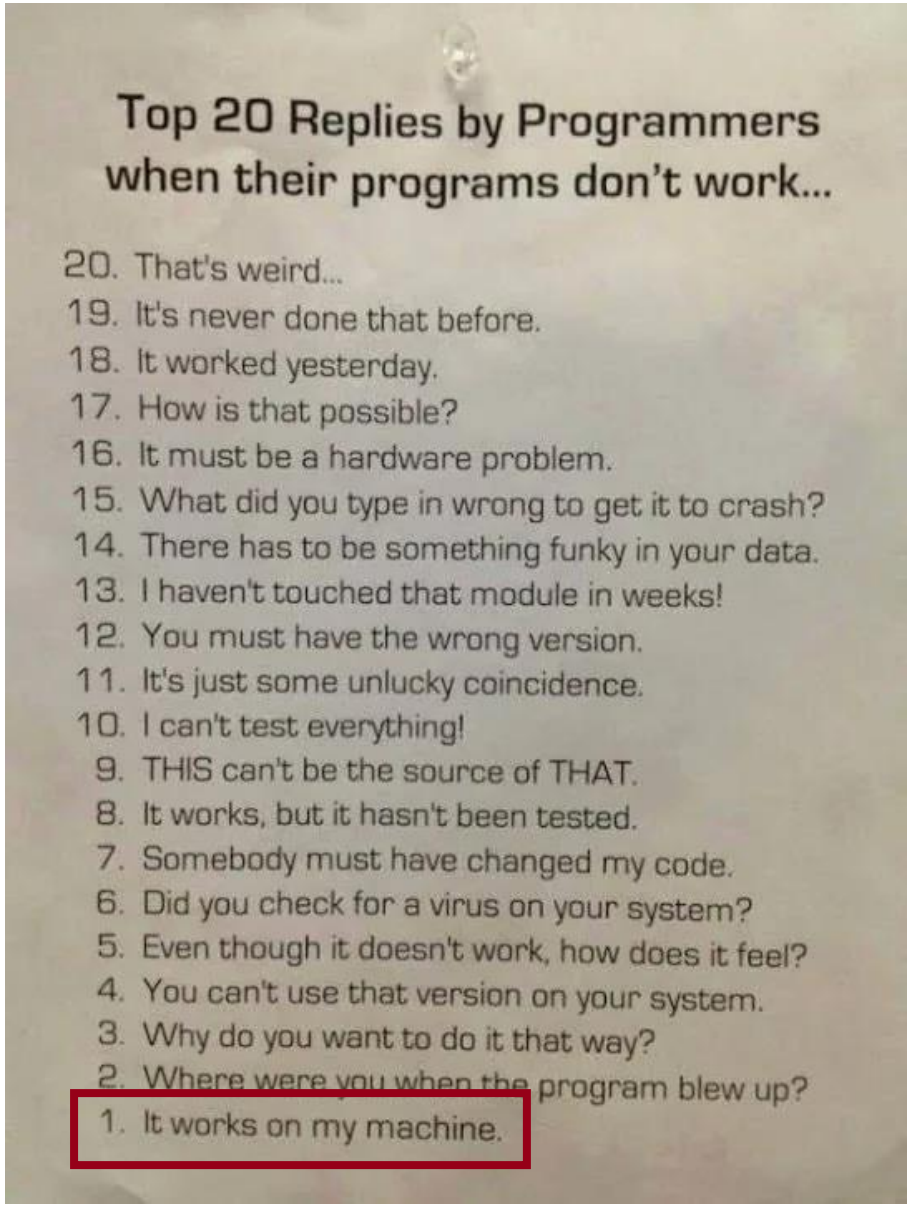
Critical Systems Research Group

Automated software engineering

Nearly all projects	Larger, long-term projects	Critical systems
• **Automated build** • **Dependency management** • IDE warnings • Unit tests • Debugging	• **Continuous Integration&Delivery** • Code generation • Static analysis • Performance evaluation • Data analysis	• Domain-specific languages • Custom static analysis rules • HW/SW simulation • Formal verification

Build automation

- **Ad-hoc approach:** each developer builds the software locally to produce an executable
  - **Dependence** of the local environment
  - Lack of **reproducibility**
  - Difficult to determine what needs to be **rebuilt**
    - Out-of-date versions of artifacts may get ‘stuck’ due to a missed rebuild
  - Difficult to **onboard** new developers



r/ProgrammerHumor

Build automation

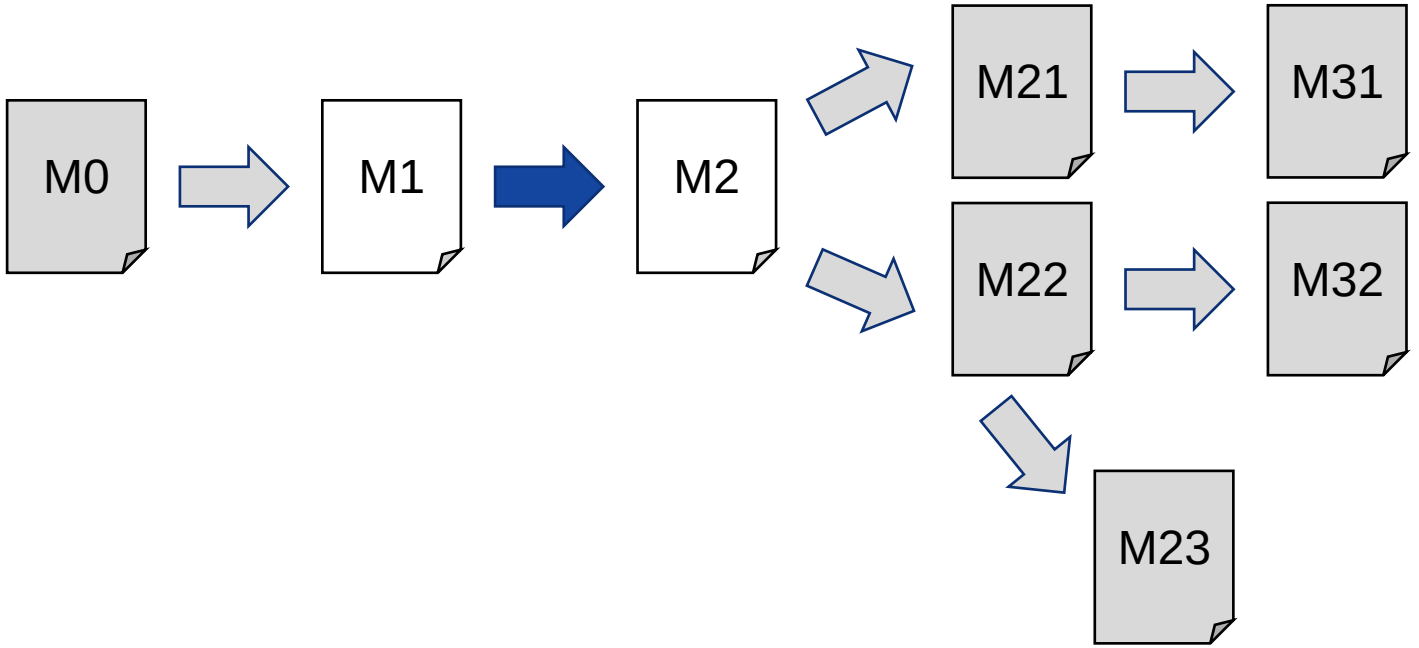
- **Solution:** describe the build process as an artifact ('build script') that can be executed reliably
  - Describe the **project structure**
  - Steps to **compile, test,** and **package**
- **Automated build:** can be run on the developer machine
- **Continuous Integration:** also run in an isolated (cloud) environment
  - Better reproducibility, less resource use on developer machines
- **Continuous Delivery:** execute tests in staging environment
  - Package ready to deploy or even deployed automatically

May be complex for large projects and require significant **engineering effort**

Build tools

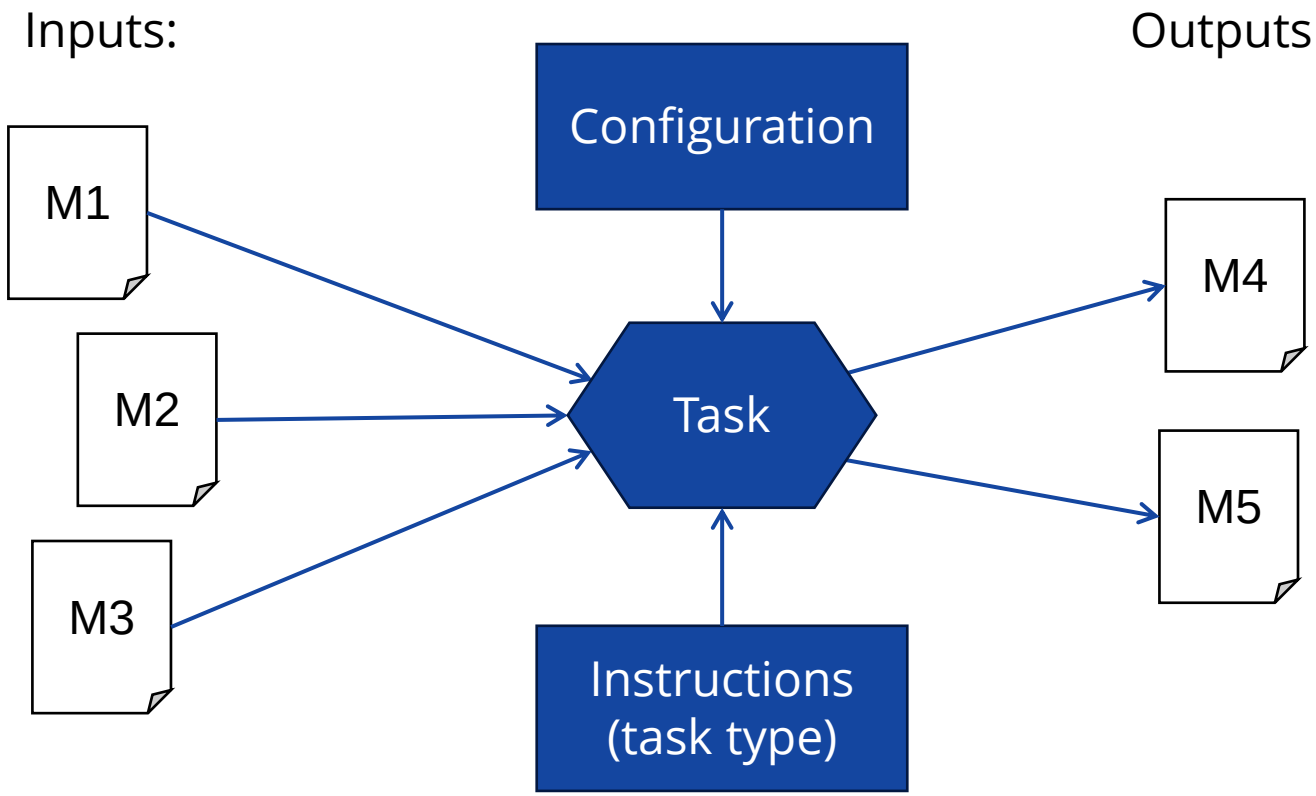
Reminder: development processes

- We create models and automate their processing
- Software is constructed by a chain of transformations



How does a single step look like?

- **Task:** single transformation to be executed

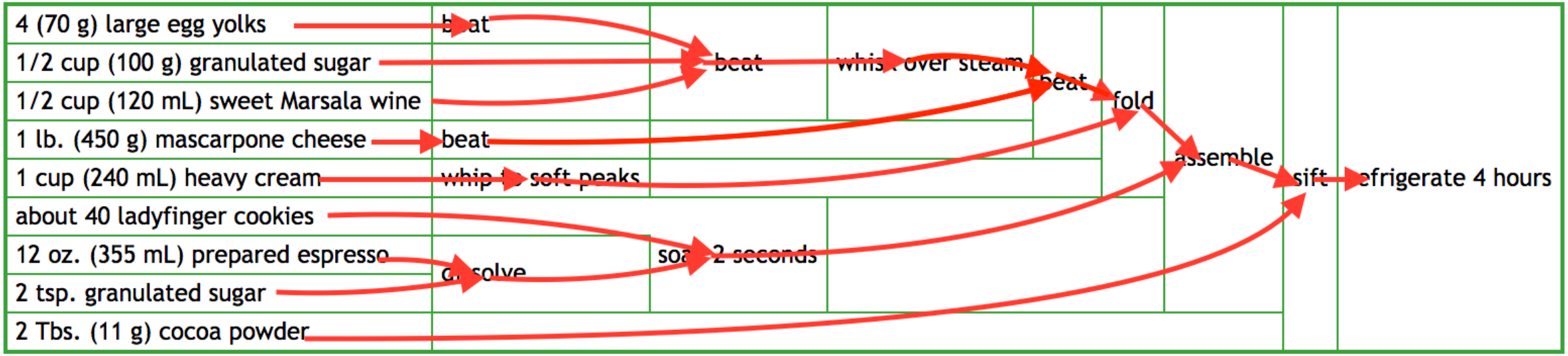


How does a single step look like?

- **Task:** single transformation to be executed
- Types of tasks
  - **Compilation:** turn source file(s) into a binary
  - **Linking:** turn multiple binaries into a single one (C, C++)
  - **Test:** run already compiled tests and produce a report
  - **Check:** run static analysis on source and produce a report
  - **Packaging:** create a deployable package from binaries
  - **Lifecycle tasks:** call other tasks during development lifecycle
    - **Build:** compile, link, test, check, and package
    - **Assemble:** compile, link, package, but skip verification (test, check)
    - **Clean:** remove intermediate and output files

Putting tasks together: data-flow graph

- **Pipeline view** of interdependent tasks



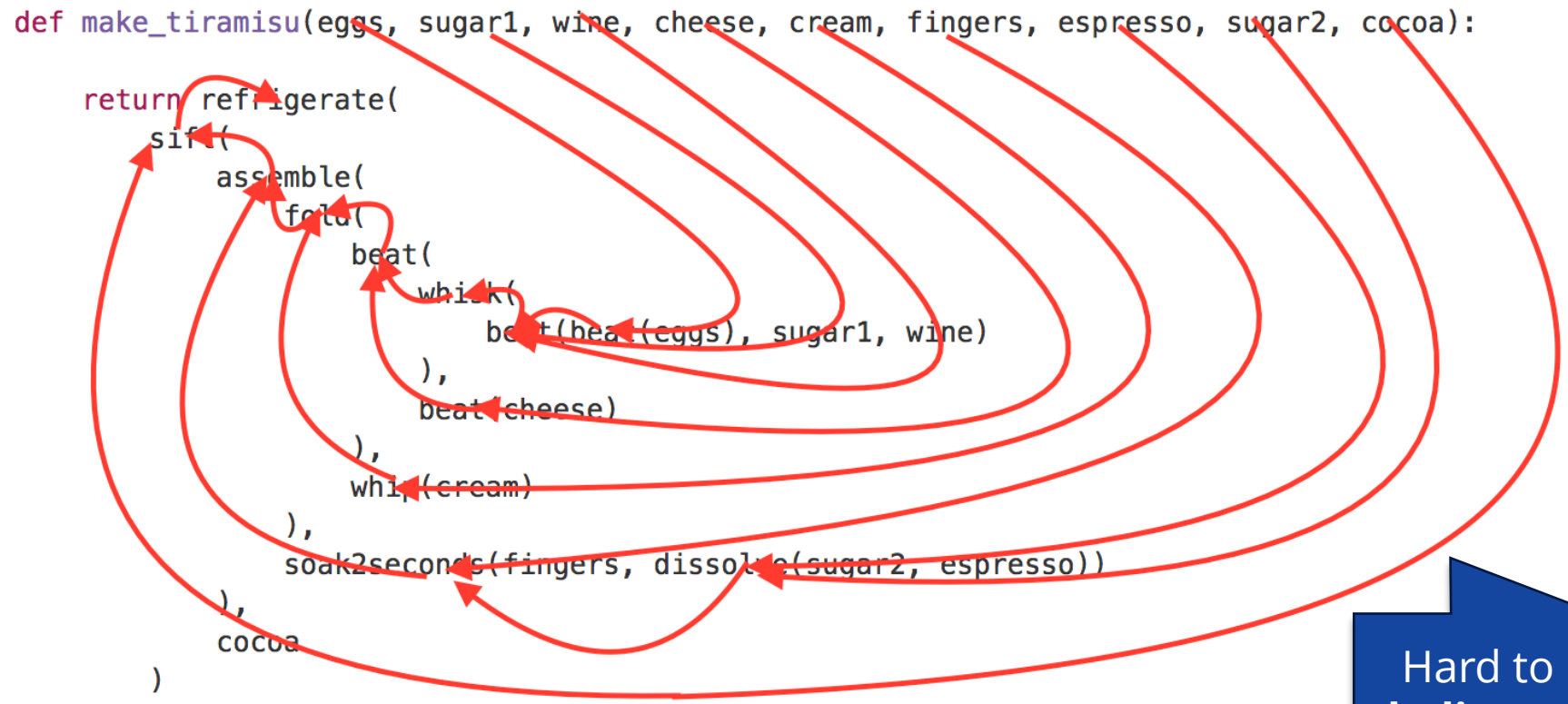
https://www.lihaoyi.com/post/SoWhatsSoSpecialAboutTheMillScalaBuildTool.html

Putting tasks together: data-flow graph

- **Task dependencies** expressed in code

```
def make_tiramisu(eggs, sugar1, wine, cheese, cream, fingers, espresso, sugar2, cocoa):

    return refrigerate(
        sift(
            assemble(
                fold(
                    beat(
                        whisk(
                            beat(beat(eggs), sugar1, wine)
                        ),
                        beat(cheese)
                    ),
                    whip(cream)
                ),
                soak2seconds(fingers, dissolve(sugar2, espresso))
            ),
            cocoa
        )
    )
```



Hard to read: need for a **dedicated** language or API

https://www.lihaoyi.com/post/SoWhatsSoSpecialAboutTheMillScalaBuildTool.html

Task definition and dependencies

- **Makefile** (GNU Make): tasks and task templates

```
CFLAGS=-O2 -fno-plt -Weverything
LDFLAGS=-Wl,-O1,--sort-common,--as-needed
.PHONY: all clean
all: output
clean:
	rm -f *.o output
%.o: %.c
	gcc -c $(CFLAGS) $< -o $@
output: main.o engine.o
	gcc $(LDFLAGS) $^ -o $@
```

Configuration

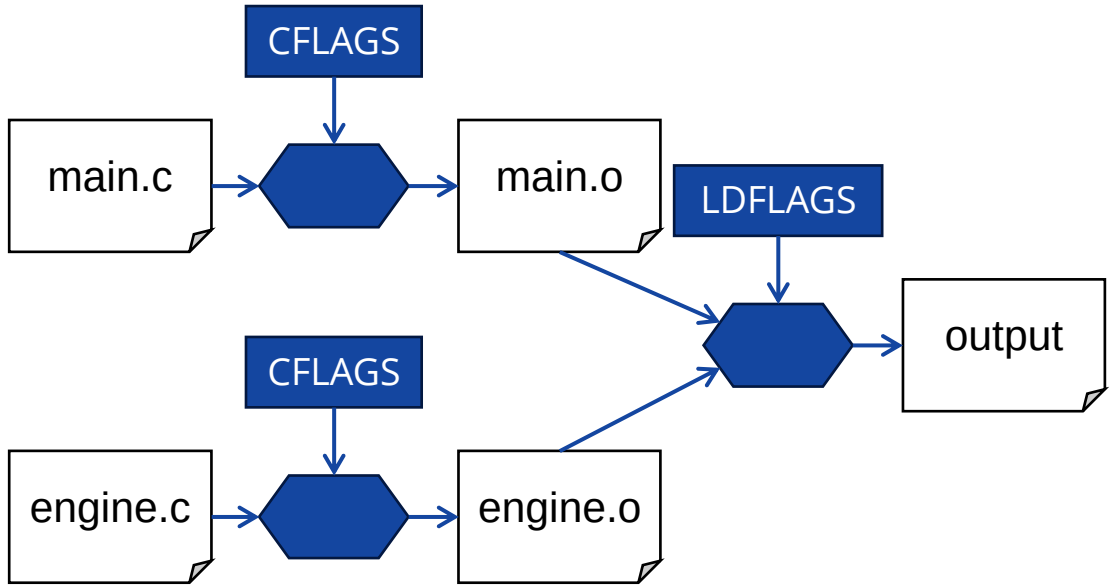
Lifecycle tasks

Task template

Input

Output

All inputs



Task definition and dependencies

- **Gradle:** task definitions in Kotlin code


```
tasks.test {
    useJUnitPlatform()
}

tasks.register<JavaExec>("runMyGenerator") {
    classpath = sourceSets.generator.runtimeClasspath
    mainClass = “org.example.generator.Main"
    inputs.dir("src/generator")
    outputs.dir("build/generated/myGenerator")
}
```

Configure built-in task

Register custom task

Configuration

Input

Output

More info: **practice session**

Approaches to writing build scripts

Convention

- ✓Large number of **built-in tasks**
- ✓**Opinionated defaults** that do not need any configuration
  - E.g., all Java source files are in `src/main/java`
- ✓Specific frameworks may provide **convention plugins** for build systems
  - E.g., Spring Boot plugin for Gradle
- ×**Limited flexibility**

Configuration

- ✓Most tasks are configured or written by hand
- ✓**Customizable** for large projects with many project-specific tasks
- ✓Adaptable to **legacy projects**
- ×Maintenance may require a dedicated **build engineer**
- ×**Learning curve** for newcomers to the project

Approaches to writing build scripts

- **Declarative:** strict separation of build logic (code executed by tasks) and configuration
  - **Task-oriented:** declarations describe tasks *(e.g., Ant, Maven, NPM)*
  - **Product-oriented:** declarations describe outputs *(e.g., Makefile, Ninja)*
  - Easy to read for small projects, but build files may grow very large
- **Hybrid:** declarations are generated by the build script
  - **Domain-specific languages** for builds *(e.g., Gradle, SBT, Scons)*
  - **Makefile generators** *(e.g., Autotools, CMake, Meson)*
  - Take care not to make scripts too complicated!

Approaches to writing build scripts

- **Declarative:** strict
  (code executed by
  - **Task-oriented:** de
  - **Product-oriented**
  - Easy to read for sm
- **Hybrid:** declaratio
  - **Domain-specific l**
  - **Makefile generators** *(e.g., Autotools, CMake, Mes*
  - Take care not to make scripts too complicated!

Gradle is such garbage. I can't count the number of times our ex-Gradler has said "you're not going to like the answer...“

https://news.ycombinator.com/item?id=35710085

SBT's bizarre abstraction and unreadable syntax is a huge, frustrating obstacle to adopting Scala

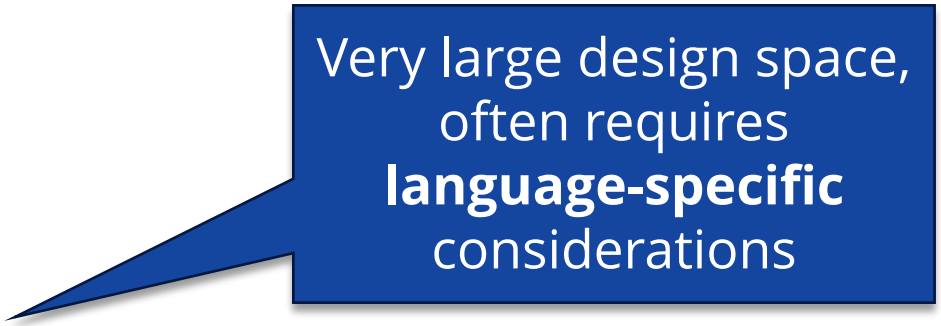
https://www.reddit.com/r/scala/comments/5a6muj/sbt_makes_me_want_to_give_up_scala/d9fvwj1/

Approaches to writing build scripts

- **Declarative:** strict separation of build logic (code executed by tasks) and configuration
  - **Task-oriented:** declarations describe tasks *(e.g., Ant, Maven, NPM)*
  - **Product-oriented:** declarations describe outputs *(e.g., Makefile, Ninja)*
  - Easy to read for small projects, but build files may grow very large
- **Hybrid:** declarations are generated by the build script
  - **Domain-specific languages** for builds *(e.g., Gradle, SBT, Scons)*
  - **Makefile generators** *(e.g., Autotools, CMake, Meson)*
  - Take care not to make scripts too complicated!
- **Imperative:** the build script directly executes actions
  - Custom shell script to build the project, hard to maintain
  - Only makes sense for special projects *(e.g., compiler bootstrapping)*

Why so many options?

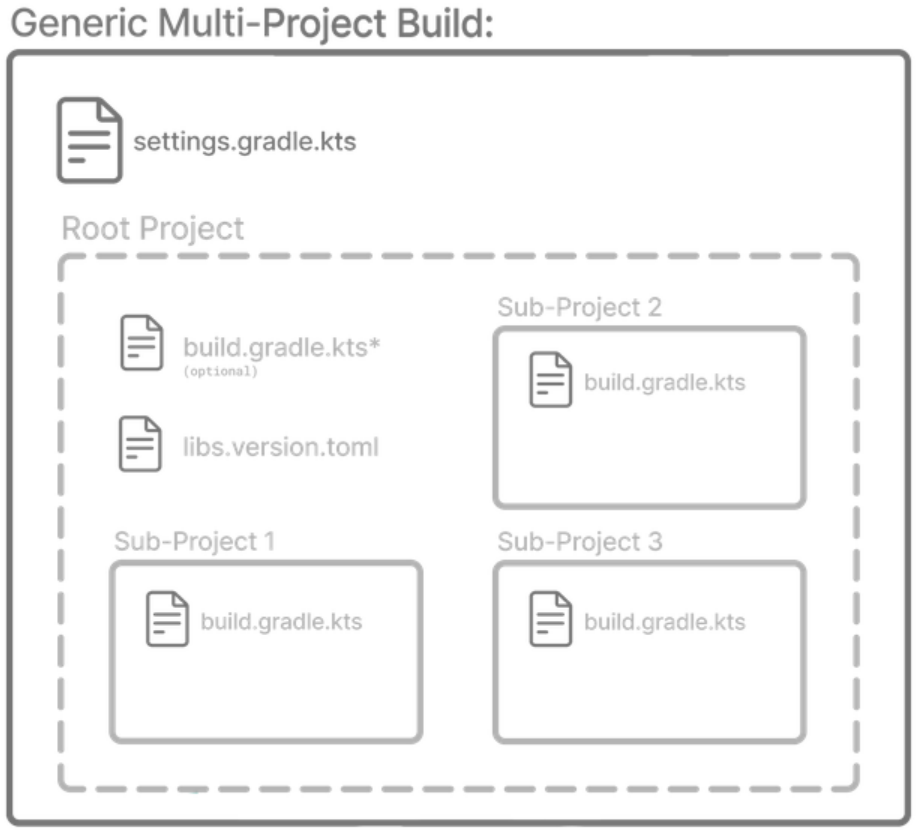
1. What tasks depends on what?
2. Where do input files come from?
3. What needs to run in what order?
4. What can be parallelized and what can't?
5. Where can tasks read/write to disk?
6. How are tasks cached?
7. How are tasks run from the CLI?
8. How are cross-builds handled? (e.g. Scala versions, JVM versions)
9. How do I define my own custom tasks?
10. How do tasks pass data to each other?
11. How to manage the repetition in a build? (e.g. every module has `compile`, `run`, `test`)
12. What is a "Module"? How do they relate to "Tasks"?
13. How do you customize an existing task or module?
14. What APIs do tasks use to actually do things?



https://www.lihaoyi.com/post/SoWhatsSoSpecialAboutTheMillScalaBuildTool.html

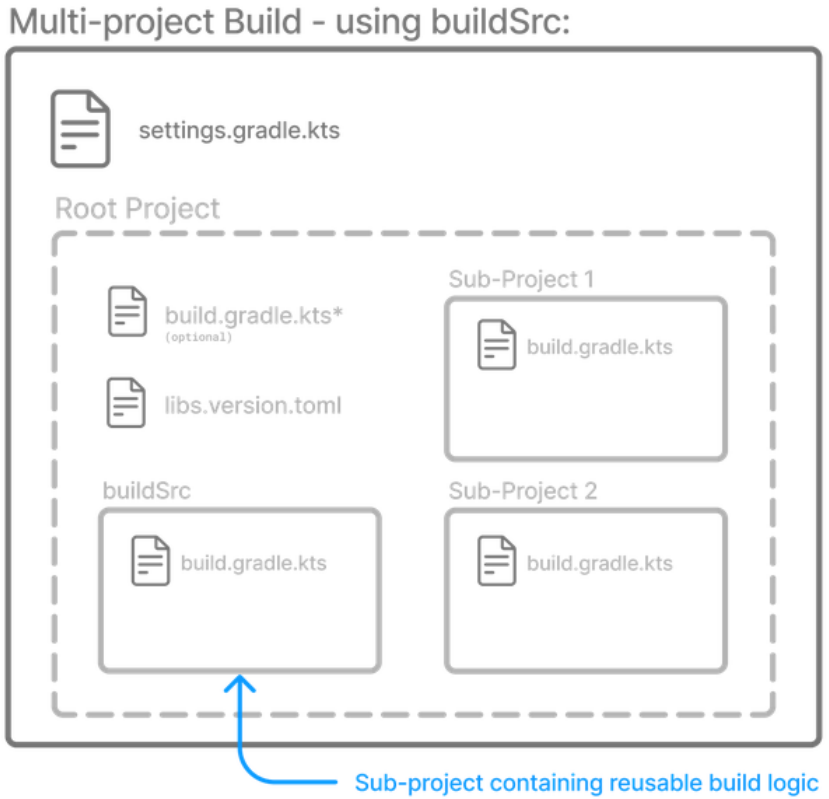
Structuring builds

- **Single-module project:** described by a single build file
- **Multi-module project:** build system tracks dependencies between components
  - Example (Gradle):



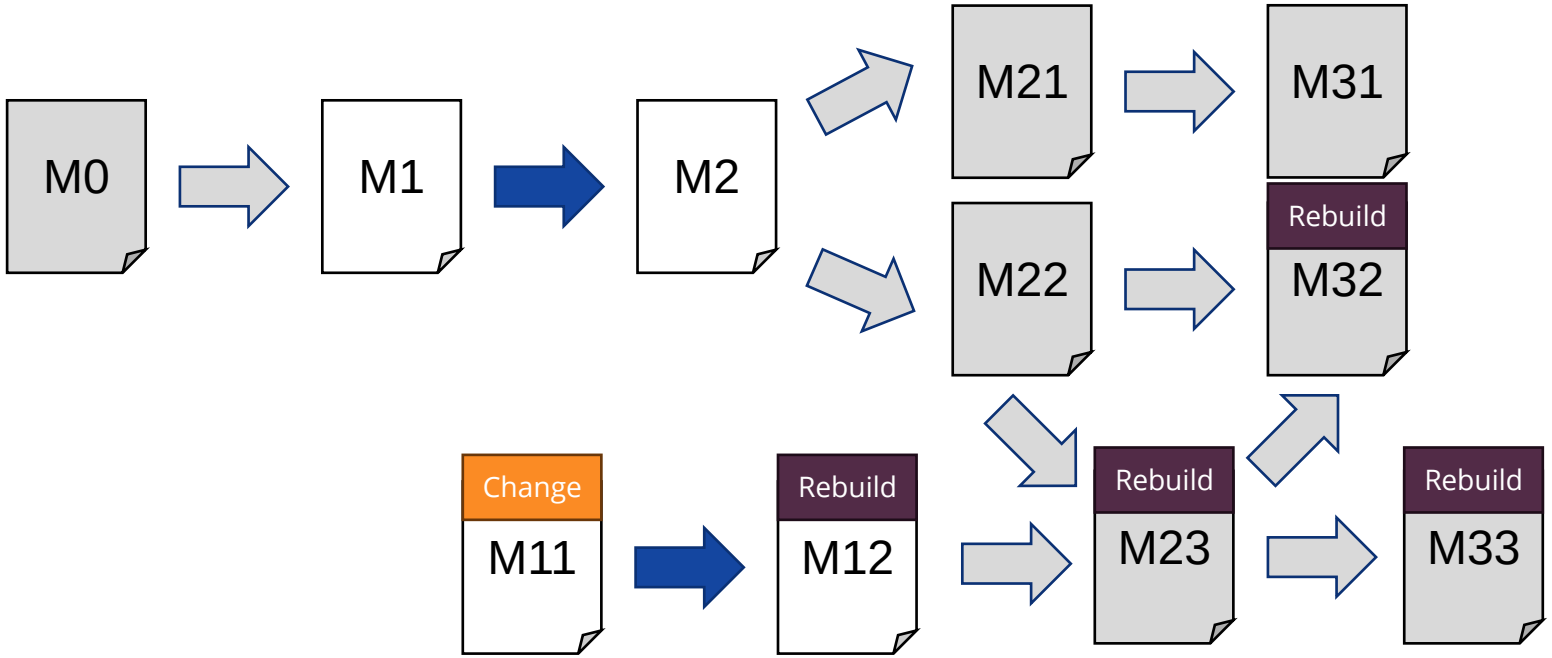
Structuring builds

- How to share build logic between subprojects?
  - Share configuration **within the project** *(Maven parent POM, Gradle buildSrc)*
  - Maintain build logic as a **dedicated project:** standardize build in large organizations with multiple separate projects
  - **Publish** build configuration as plugins: allows downstream users to take advantage of established conventions *(e.g., Spring Boot for Gradle)*



Incremental build

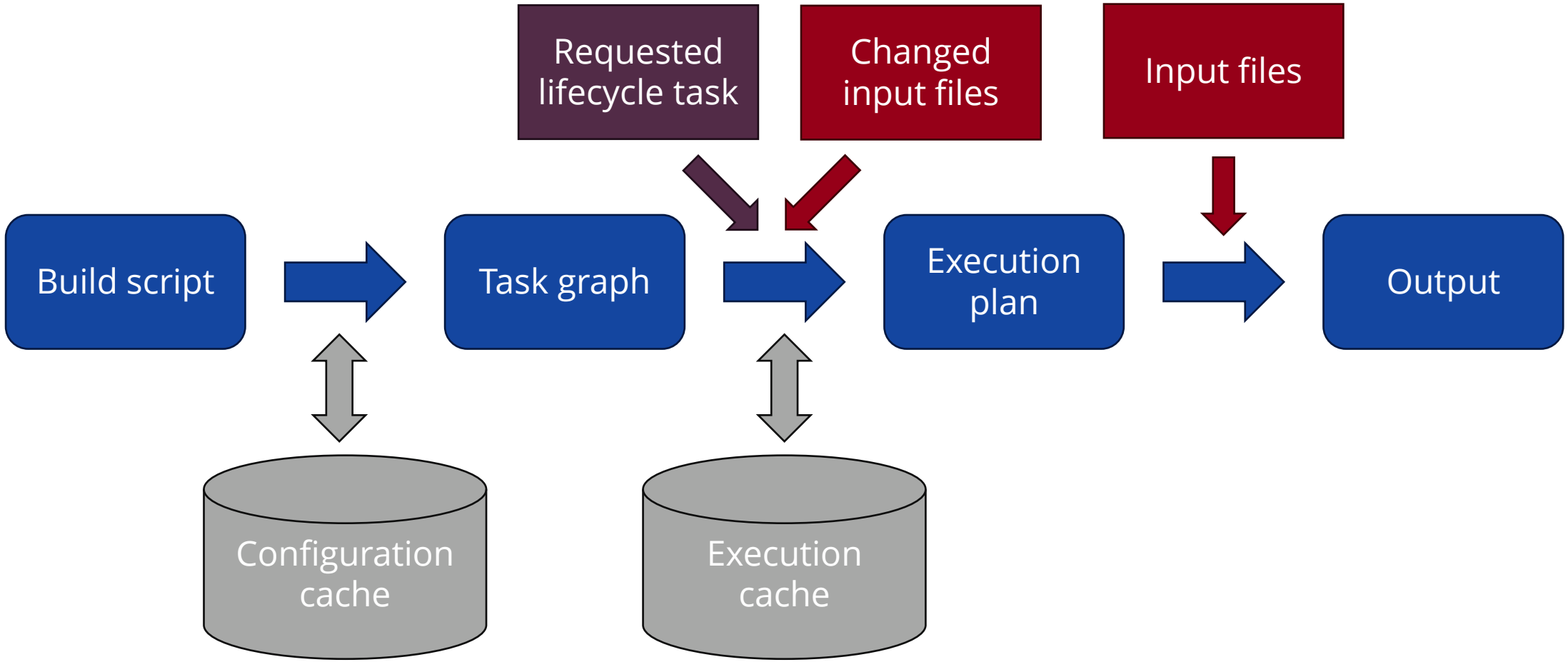
- What to rebuild when the project changes?



Incremental build

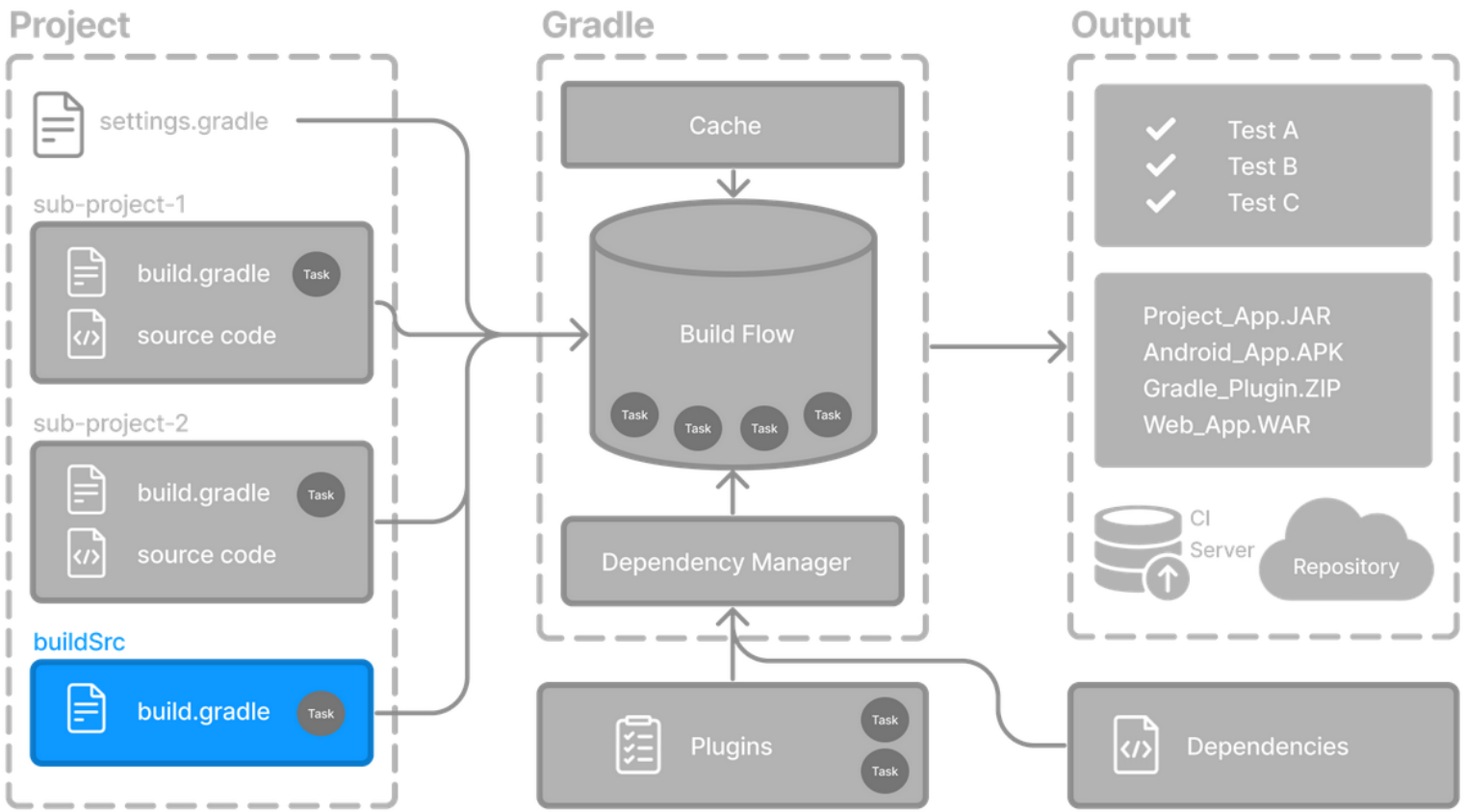
- What to rebuild when the project changes?
- **Non-incremental build:** always run all requested tasks *(e.g., Ant)*
- **Timestamp-based:** rerun if output does not exist or has an earlier timestamp than the inputs *(e.g., Makefile)*
- **Hash-based:** rerun if the hash of the input differs from the hash recorded in the execution cache *(e.g., Gradle)*
  - **Remote build cache:** also record the hash of the output and store the output file in a **content-addressable store** for retrieval without executing the task again
    - Requires **deterministic execution** of build steps

Build execution as a transformation chain

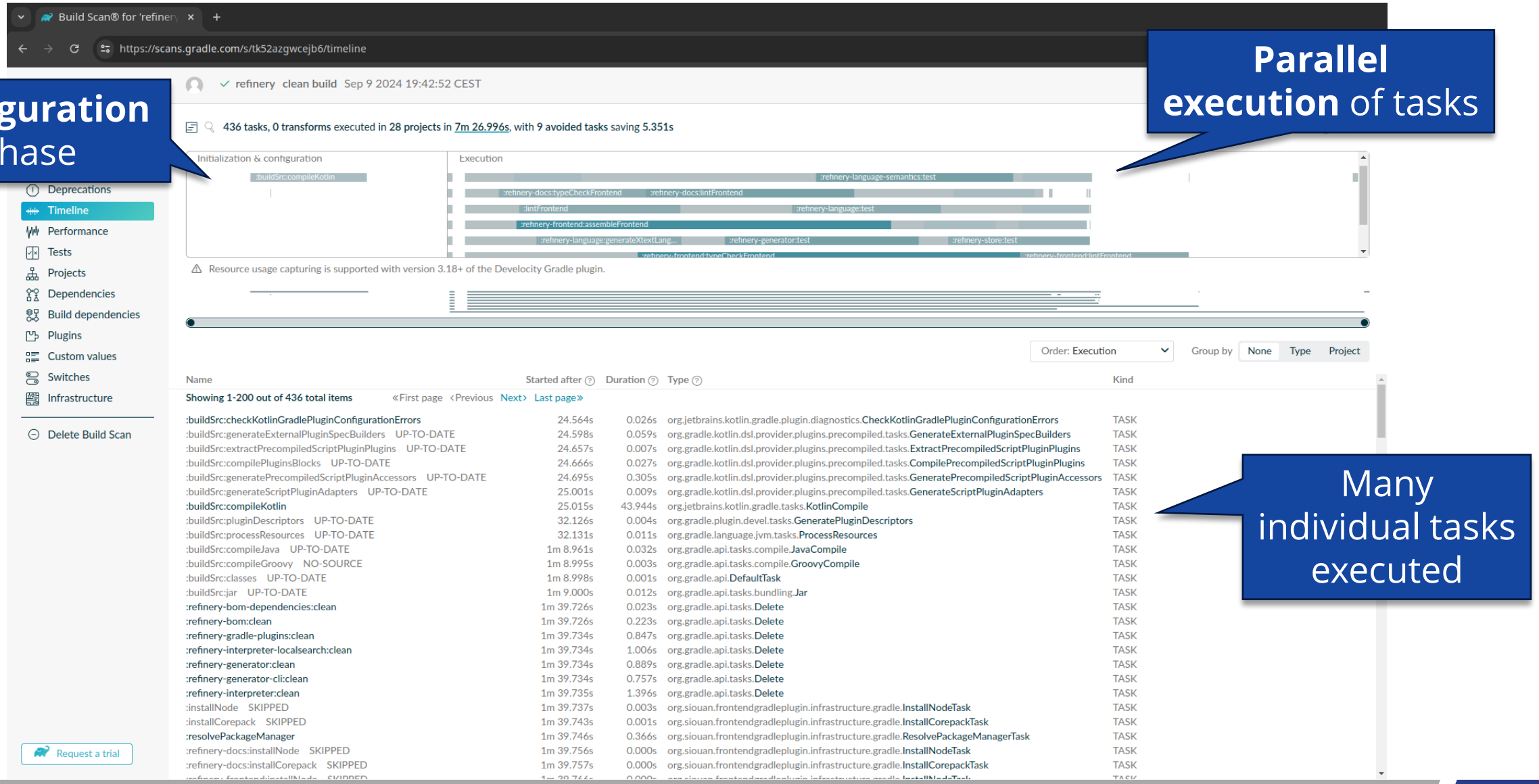


Build execution as a transformation chain

- Practical example from **Gradle**



Example build execution (Gradle build scan)



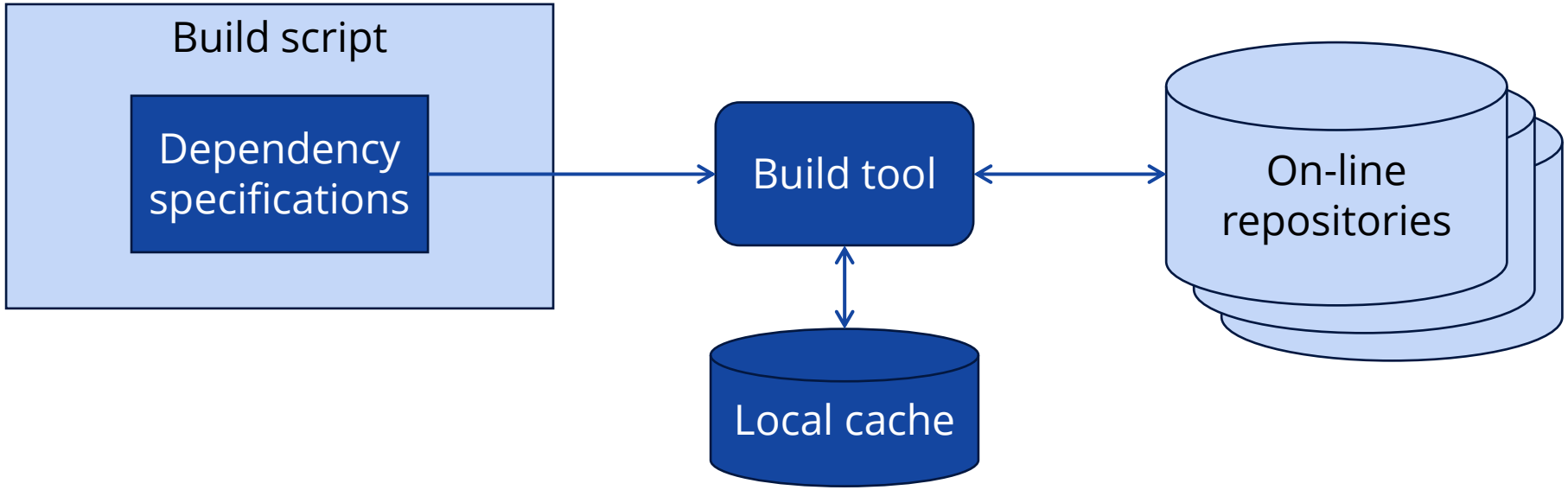
Dependency management

Program dependencies

- **Artifacts** used by the project
  - Libraries needed for building the code
  - Libraries needed for running the code
- Example: **external** .jar files for a Java project
- **Transitive** dependencies
  - Dependencies of dependencies
  - Dependencies of dependencies of dependencies
  - …
- A project may require a specific range of **versions**

Retrieving dependencies

- Not strictly part of build automation
  - E.g., Makefiles and Meson rely on the operating system to discover libraries
- But often supported by language-specific build tools
  - E.g., Maven repositories for Maven, NPM for NodeJS



Types of repositories

- **Public repositories** *(e.g., Maven Central, NPM)*
  - Publicly available infrastructure for a programming language
  - Anyone may contribute libraries, but there may be strict moderation (e.g., publishers must prove that they are the maintainer of the published library and use digital signatures)
  - **Open-source libraries** are usually available here
- **Repositories as a service** *(e.g., Github packages)*
  - Controlled access (authentication required)
  - Sharing non-public artifacts within an organization
- **Self-hosted repositories**
  - Can be hosted on-premise on existing infrastructure

Semantic versioning

https://semver.org

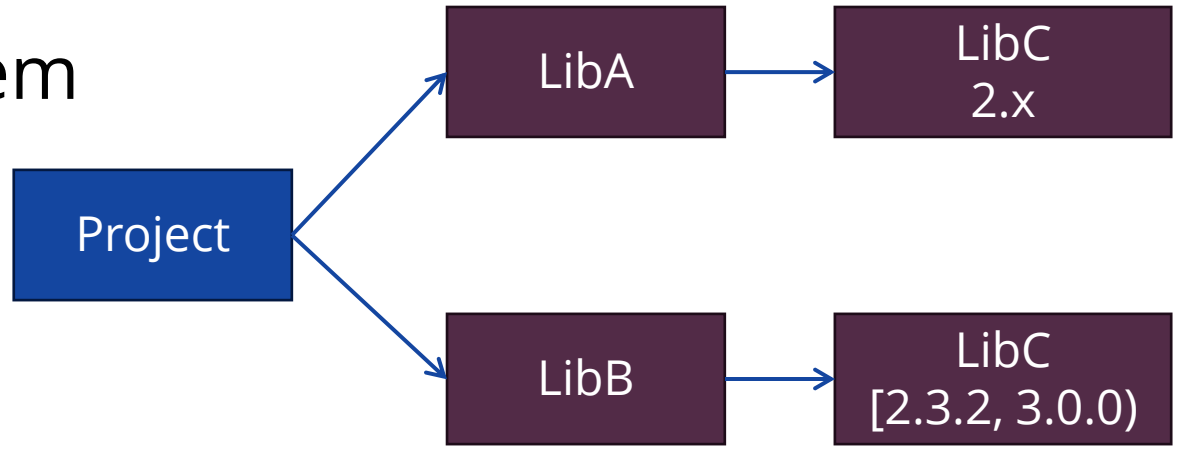
Given a version number, increment the:

- **Major version** when you make incompatible API changes
- **Minor version** when you add functionality in a backward compatible manner
- **Patch version** when you make backward compatible bug fixes

Allows a consumer to declare **compatible** library versions

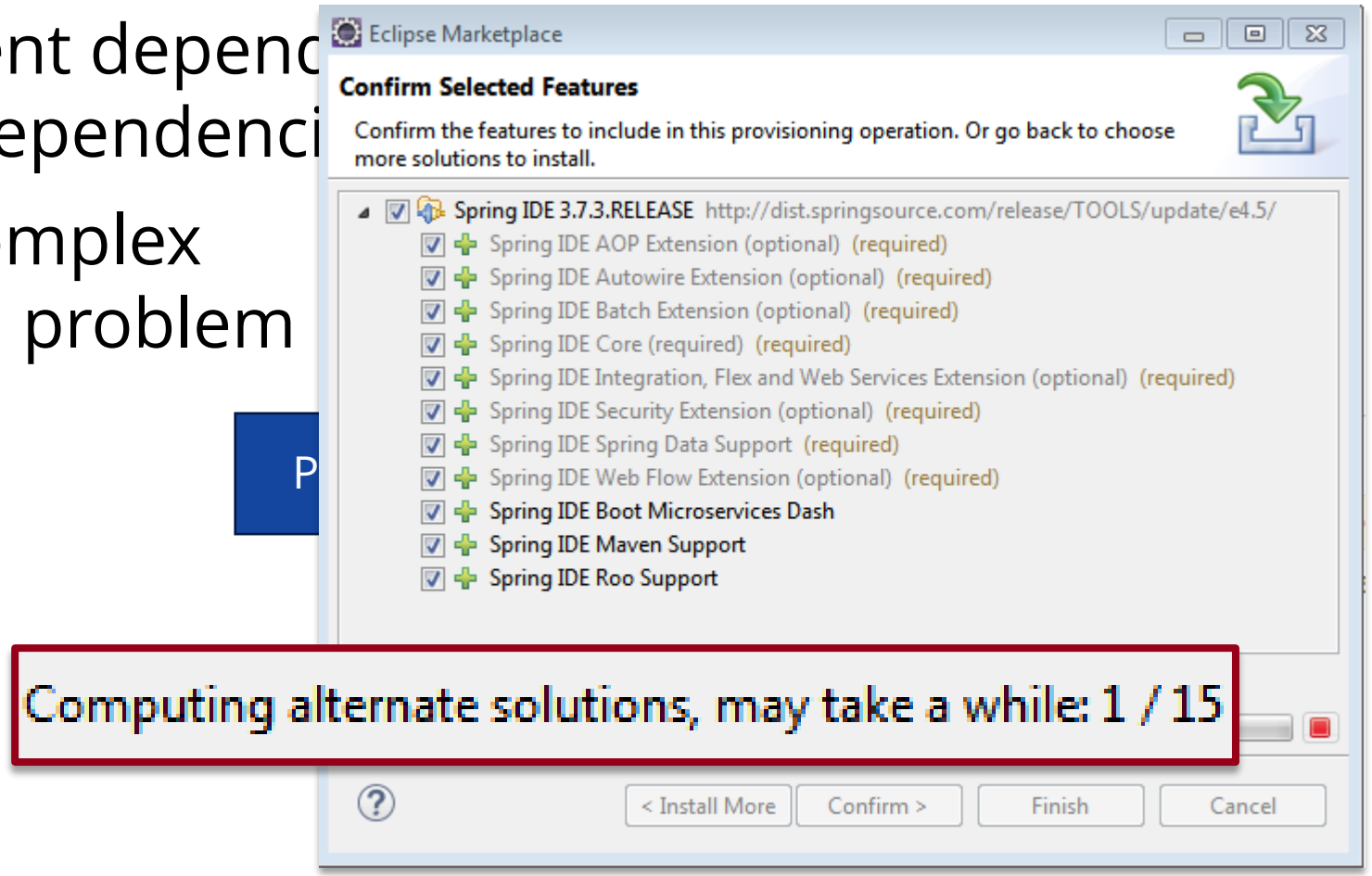
Transitive dependencies

- What to do when different dependencies declare different versions for their own dependencies?
- May require solving a complex **constraint satisfaction** problem



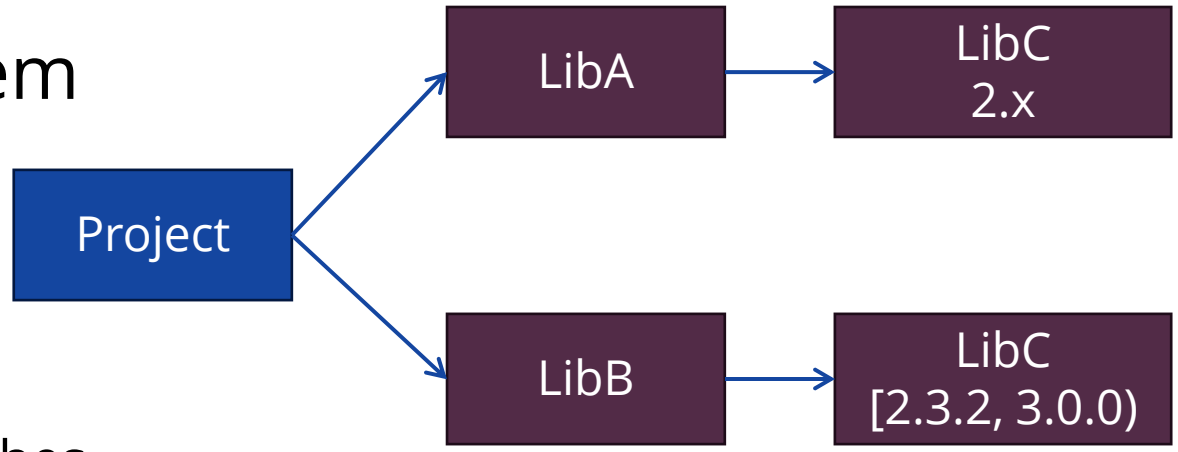
Transitive dependencies

- What to do when different depend
  versions for their own dependenci
- May require solving a complex
  constraint satisfaction problem



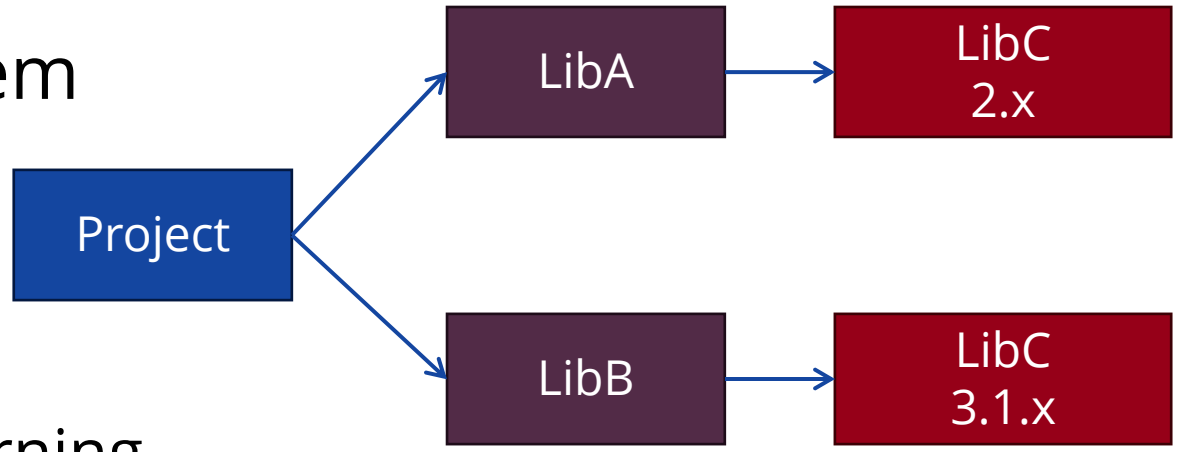
Transitive dependencies

- What to do when different dependencies declare different versions for their own dependencies?
- May require solving a complex **constraint satisfaction** problem
- **Compatible** version ranges
  - Resolve to the **earliest** allowed
  - Resolve to the **latest** allowed
    - Benefit from all recent security patches
    - But may introduce new incompatibilities and bugs
  - **Lockfiles** allow storing the resolution in the source repository
    - Use known tested versions, upgrade manually



Transitive dependencies

- What to do when different dependencies declare different versions for their own dependencies?
- May require solving a complex **constraint satisfaction** problem
- **Incompatible** version ranges
  - Try to use two versions simultaneously *(e.g., NodeJS)*
  - Use some version and emit a warning *(e.g., Gradle)*
  - Fail outright
  - Manual resolution overrides to select compatible (patched) versions



Do we trust our dependencies?

- Critical bug in the Log4J 1.x Java logging library allowing remote code execution
  - Only the 2.x branch is currently maintained
- Affected thousands of libraries as transitive dependencies
  - Required manual override to resolve patched versions

America's Cyber Defense Agency
NATIONAL COORDINATOR FOR CRITICAL INFRASTRUCTURE SECURITY AND RESILIENCE

Search

Topics | Spotlight | Resources & Tools | News & Events | Careers | About

Home / News & Events / News

BLOG

Apache Log4j Vulnerability Guidance

Released: April 08, 2022

RELATED TOPICS: CYBER THREATS AND ADVISORIES

Summary

Note: *CISA will continue to update this webpage as well as our community-sourced GitHub repository as we have further guidance to impart and additional vendor information to provide.*

CISA and its partners, through the Joint Cyber Defense Collaborative, are responding to active, widespread exploitation of a critical remote code execution (RCE) vulnerability (CVE-2021-44228) in Apache's Log4j software library, versions 2.0-beta9 to 2.14.1, known as "Log4Shell." Log4j is very broadly used in a variety of consumer and enterprise services, websites, and applications—as well as in operational technology products—to log security and performance information. An unauthenticated remote actor could exploit this vulnerability to take control of an affected system.

(Updated April 8, 2022) Organizations should continue identifying and remediating vulnerable Log4j instances within their environments and plan for long term vulnerability management. Consider the following in planning:

- **Scope of covered assets:** Due to the limited availability of initial information, identification and mitigation efforts may have been scoped to a limited number of an organization's assets. For long term mitigation, ensure the prevalence of log4j in all assets is considered and accounted for, including internally developed software and non-internet facing technology stacks.
- **Continuous enumeration and analysis:** Organizations need to perform comprehensive analysis to fully enumerate all Log4J vulnerabilities. This should include scanning (network and host) and comparing installed software with software listed in CISA's Log4j vulnerable software database.
  - **High fidelity scanning.** Consider using file system scanning scripts to identify vulnerable Log4j files or use vulnerability scanners that leverage file scanning.
  - **Newly vulnerable 3rd party software.** Organizations may lack insight into certain applications, such as Software as a Service (SaaS) solutions and other cloud resources. Organizations should continue to review the CISA log4j vulnerable software database and cross reference against used software.

Do we trust our dependencies?

- Targeted backdoor in the XZ Utils library
- Obfuscated payload committed to the XZ Utils repository as an integration test by a long-time contributor called *Jia Tan*
- Compromised binaries shipped in Linux distributions
- **Software supply chain attack**

America's Cyber Defense Agency
NATIONAL COORDINATOR FOR CRITICAL INFRASTRUCTURE SECURITY AND RESILIENCE

Search

Topics | Spotlight | Resources & Tools | News & Events | Careers | About

Home / News & Events / Cybersecurity Advisories / Alert

ALERT

Reported Supply Chain Compromise Affecting XZ Utils Data Compression Library, CVE-2024-3094

Release Date: March 29, 2024

CISA and the open source community are responding to reports of malicious code being embedded in XZ Utils versions 5.6.0 and 5.6.1. This activity was assigned CVE-2024-3094. XZ Utils is data compression software and may be present in Linux distributions. The malicious code may allow unauthorized access to affected systems.

CISA recommends developers and users to downgrade XZ Utils to an uncompromised version—such as XZ Utils 5.4.6 Stable—hunt for any malicious activity and report any positive findings to CISA.

See the following advisory for more information:

- Red Hat: Urgent security alert for Fedora 41 and Rawhide users

Software Bill of Materials (SBOM)

- Collect the **provenance information** of dependencies and code in a dedicated software artifact
  - Do we comply with the **licenses** of the dependencies?
  - Were the dependencies **signed** by their suppliers?
  - Is there a **support contract** in place with the suppliers?
  - Are there any outstanding security bulletins or **know vulnerabilities?**
  - Have the dependencies been **audited** according to industry regulations?
  - **Software transparency:** share SBOM with downstream users

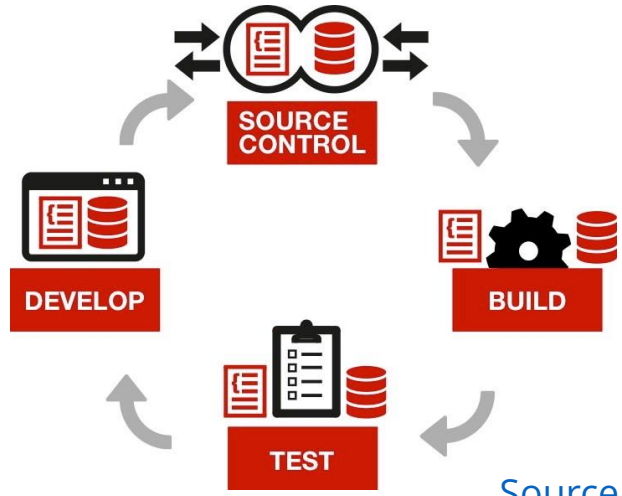
Continuous Integration and Deployment

Continuous Integration (CI)

Martin Fowler

https://martinfowler.com/articles/continuousIntegration.html

- *„a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily"*
- *„Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible."*



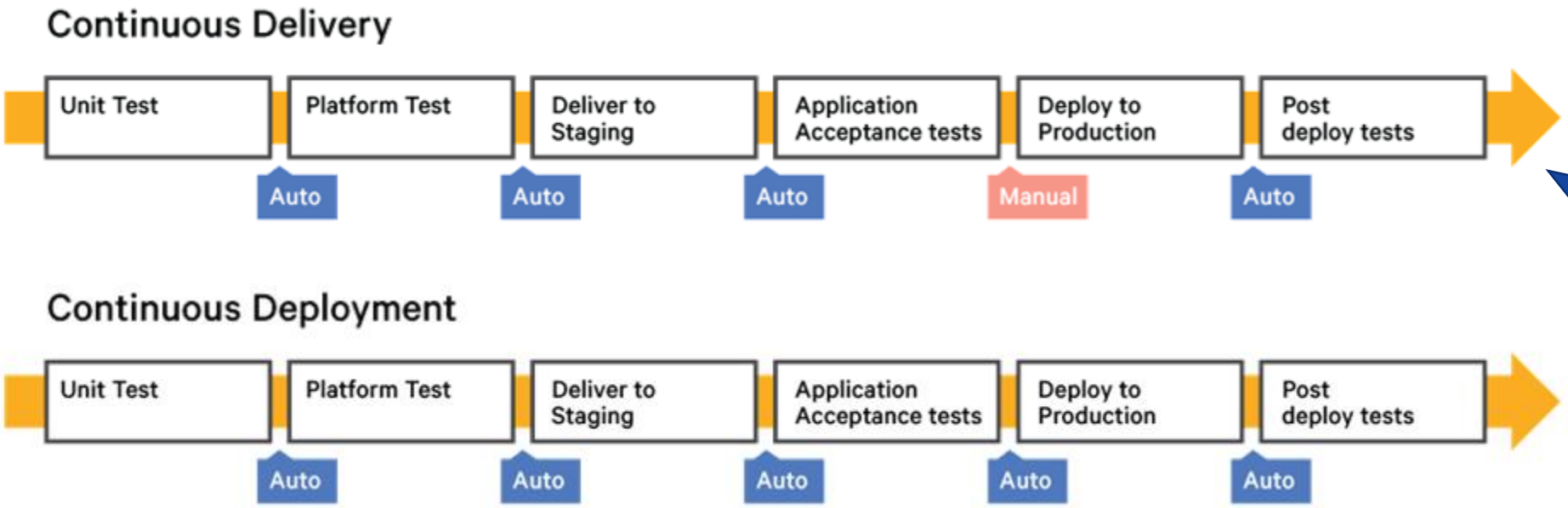
Source

Maven™ Gradle 5 JUnit 5

Continuous Delivery (CD)

„build software so that it is always in a state where it could be put into production”

Source: https://martinfowler.com/bliki/ContinuousDelivery.html



delivery pipeline

Source

GitHub Actions

docker

Characteristics of CI environments

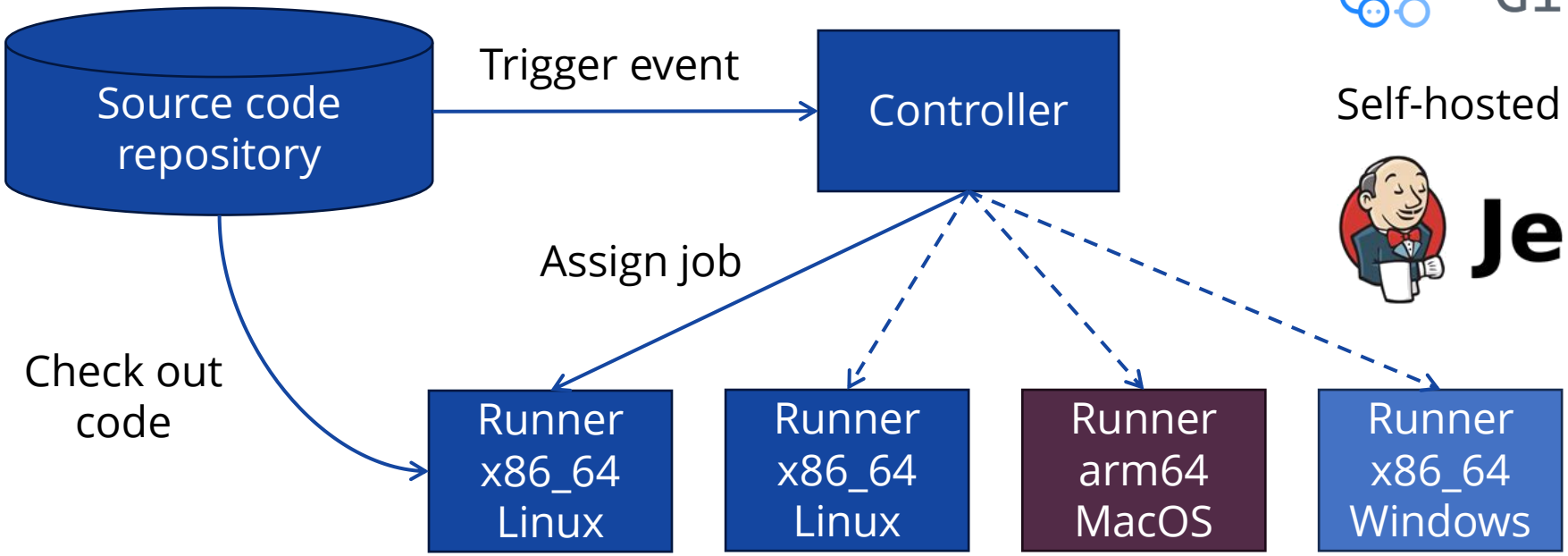
- Build and test the software in an **isolated environment**
  - Usually **cloud-based,** disposable environments (virtual machine, container)
  - **Reproducible** from the CI/CD pipeline configuration
- Support for **long-running** and parallel tasks

> "There are no other software projects like this," [Mark] Lucovsky said, "but the one thing that's remained constant [over the years] is how long it takes to build [Windows]. **No matter which generation of the product, it takes 12 hours to compile and link the system.**" Even with the increase in processing horsepower over the years, Windows has grown to match, and the development process has become far more sophisticated, so that Microsoft does more code analysis as part of the daily build. "The CPUs in the build lab are pegged constantly for 12 hours," he said. "We've adapted the process since Windows 2000. Now, we decompose the source [code] tree into independent source trees, and use a new build environment. It's a multi-machine environment that lets us turn the crank faster. But because of all the new code analysis, it still takes 12 hours."
>
> http://web.archive.org/web/20100712104930/http://www.winsupersite.com/reviews/winserver2k3_gold2.asp (2003)

Characteristics of CI environments

- Build and test the software in an **isolated environment**
  - Usually **cloud-based,** disposable environments (virtual machine, container)
  - **Reproducible** from the CI/CD pipeline configuration
- Support for **long-running** and parallel tasks
- Integration **triggers**
  - Commit/pull request/merge to the source repository
  - Periodic (e.g., nightly builds) especially for long-running tasks
  - Manual trigger (e.g., publish new software release)

General architecture



Cloud-based solution example:

GitHub Actions

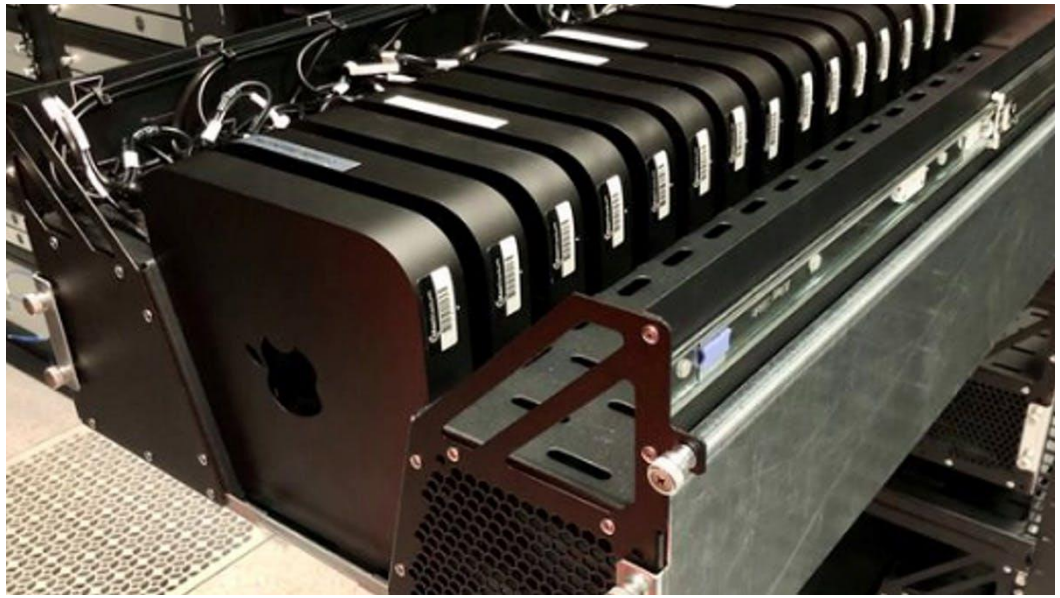
Self-hosted solution example:

Jenkins

```
Set up job
1  Current runner version: '2.319.1'
2  ▸ Operating System
6  ▾ Runner Image
7    Image: ubuntu-22.04
8    Version: 20240825.1.0
9    Included Software: https://github.com/actions/runner-images/blob/ubuntu22/20240825.1/images/ubuntu/Ubuntu2204-Readme.md
10   Image Release: https://github.com/actions/runner-images/releases/tag/ubuntu22%2F20240825.1
11 ▾ Runner Image Provisioner
12   2.0.384.1
```

Types of runners

- **Shared runners:** available on demand in the cloud
  (e.g., Github Actions runners)
  - Shared among many users of the CI/CD platform
  - Relies on virtual machines or containers for isolation
  - Usually billed according to use



Mac Minis racked in a datacenter,
https://www.macstadium.com/blog/m1-mac-minis-coming-to-macstadium

Types of runners

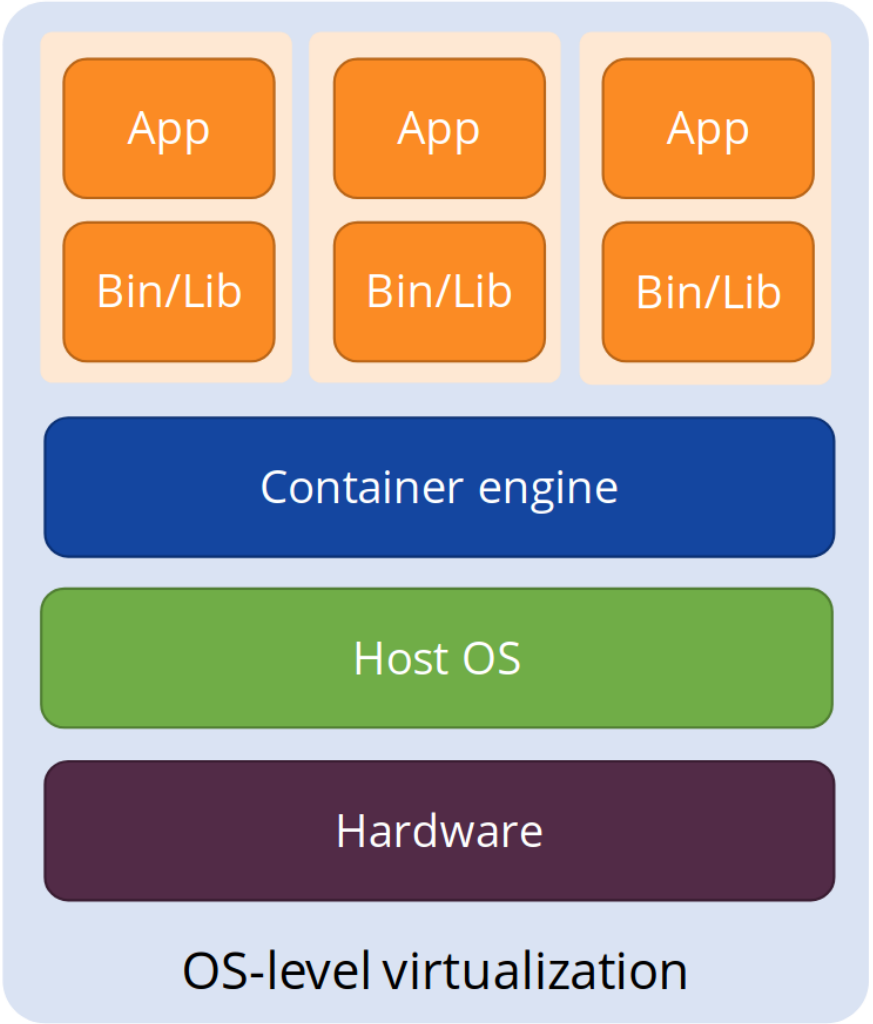
- **Shared runners:** available on demand in the cloud *(e.g., Github Actions runners)*
- **Provisioned runners:** leased for a specific organization in the cloud *(e.g., Github Actions large runners)*
  - Usually offer more configuration options than normal runners
  - Access to more resources and longer timeouts
  - Also virtual machine based isolation

Types of runners

- **Shared runners:** available on demand in the cloud *(e.g., Github Actions runners)*
- **Provisioned runners:** leased for a specific organization in the cloud *(e.g., Github Actions large runners)*
- **Self-hosted runners:** let the CI/CD controller execute jobs on runners hosted on premises
  - Allow access to specialized hardware
  - Allow access to specialized software (licensing considerations)
  - Allow access to the internal network
  - Allow keeping secret information entirely within the organization

Basics of containerization

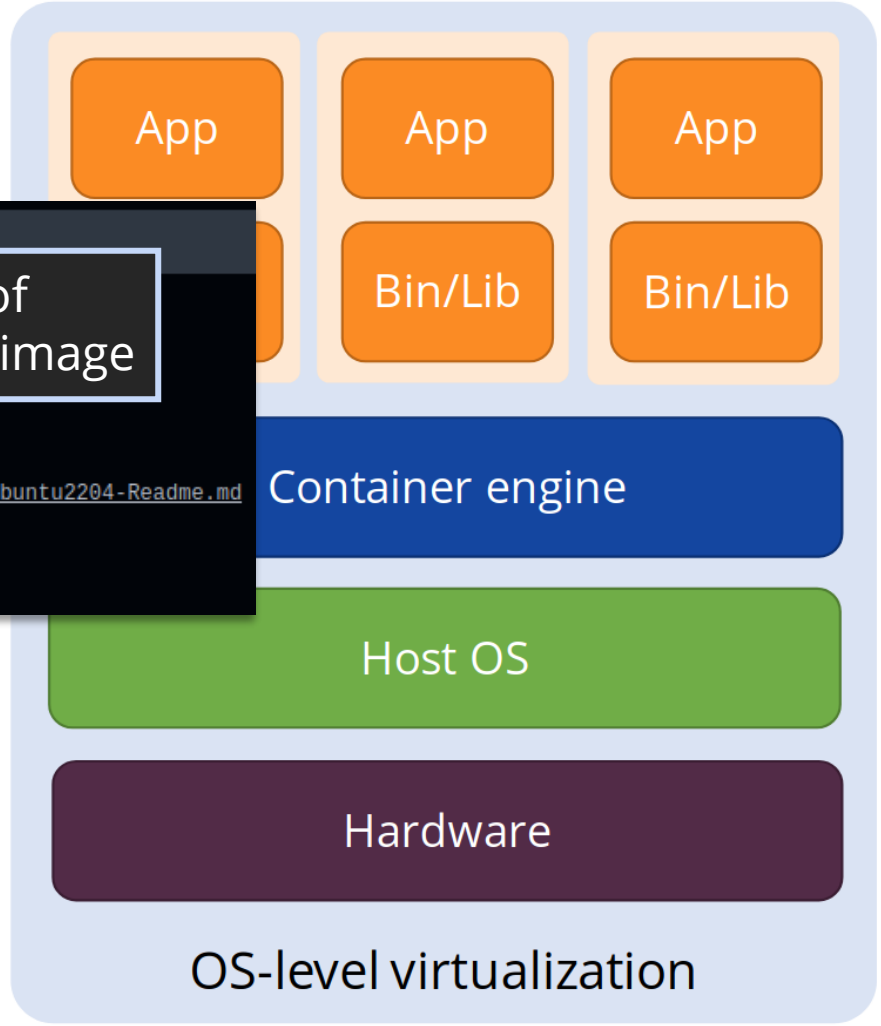
- Lightweight technique to **isolate** applications using OS features
- Containers…
  - share host **kernel,**
  - but have their own system **binaries** and **libraries** (usually a full Linux environment)
- Isolation from other **containers** and other **OS processes**
- A **container image** can be quickly started to create a new **container**
- Example runtime: docker



Basics of containerization

- Containerized runtimes let CI runners quickly spin up isolated environments

- A container image may also be the output of a CI/CD pipeline as an easily **deployable artifact**



Build scripts and CI/CD pipelines

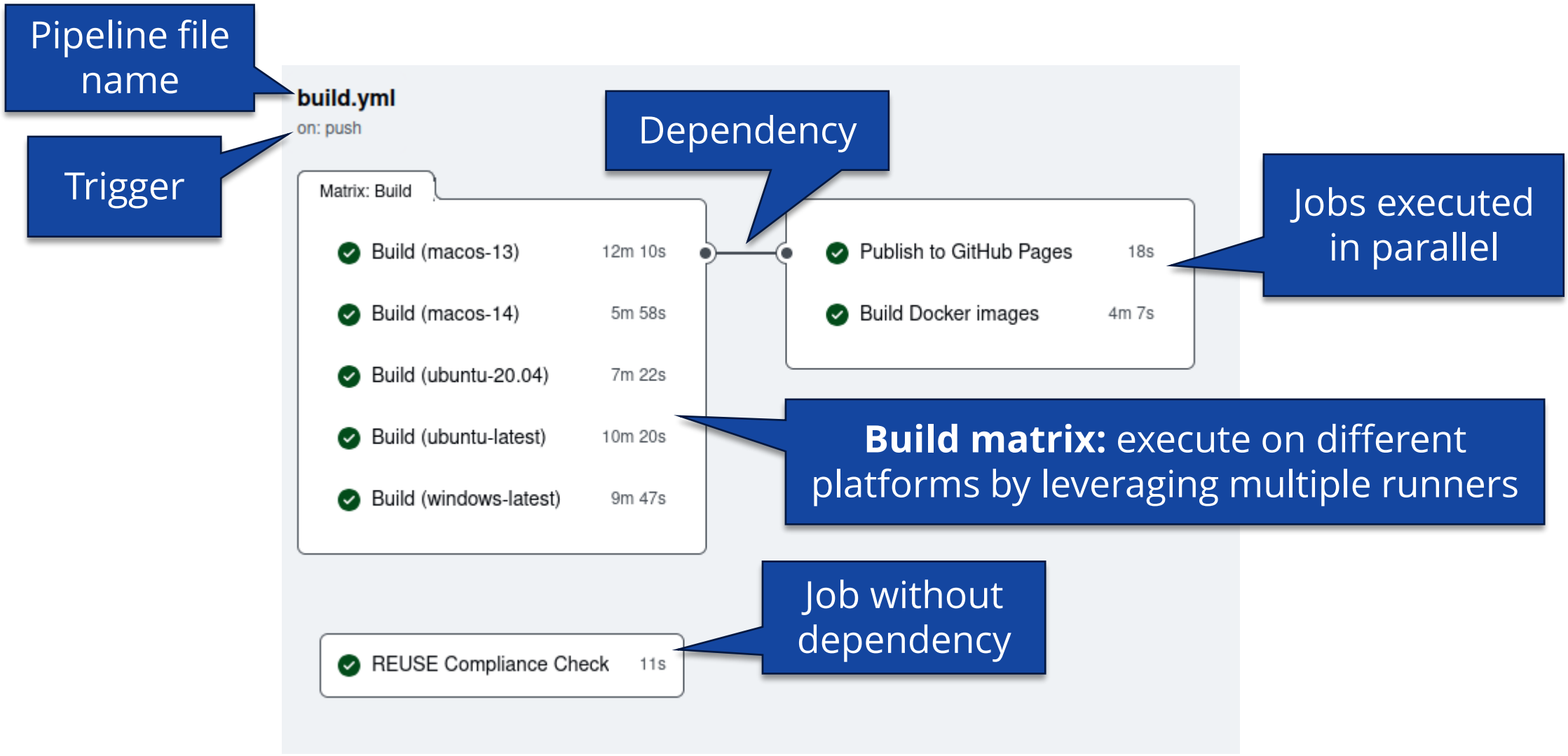
Build scripts

- Can be **declarative** or **imperative**
- Runs in a **single environment** at a time
- Usually triggered by the developer (**command line** or **IDE**)
- Steps are individual **tasks**

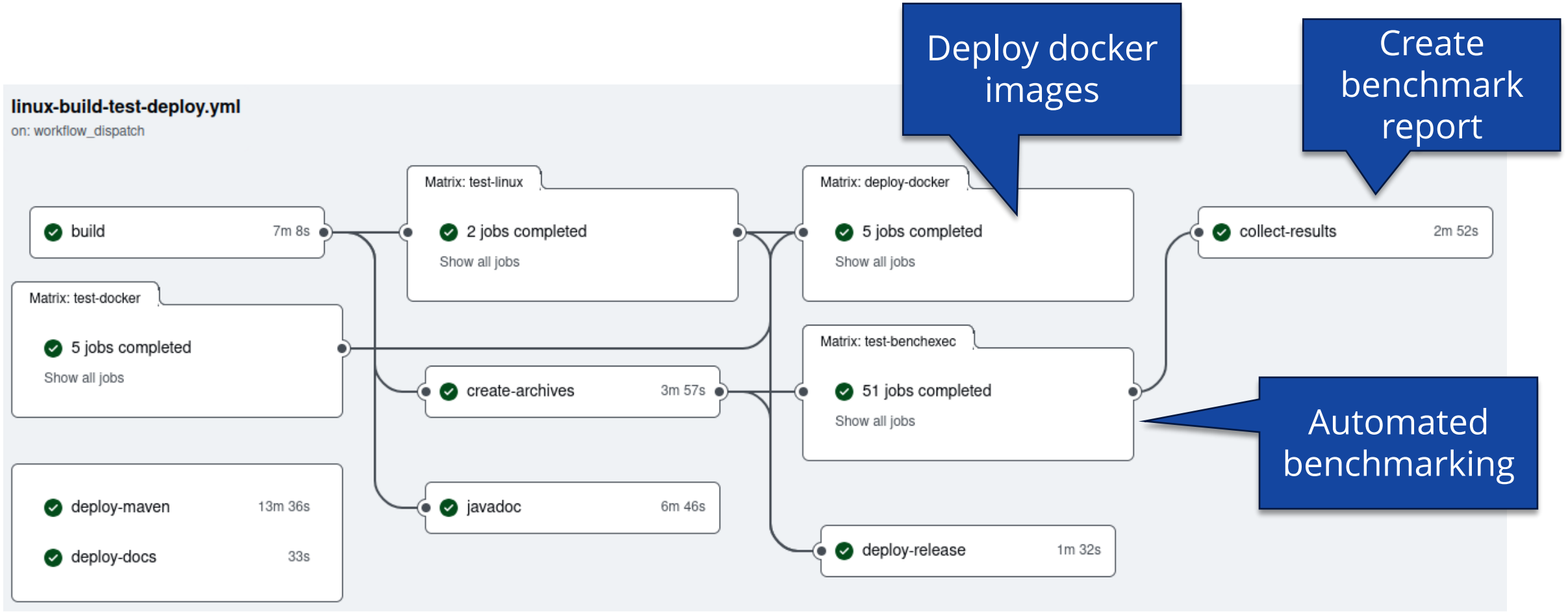
CI/CD pipelines

- Usually purely **declarative**
  - Avoid running custom code on the controller
- Define pipelines across **multiple environments**
- Various **triggers**
- May execute a build tool as an **intermediate step**

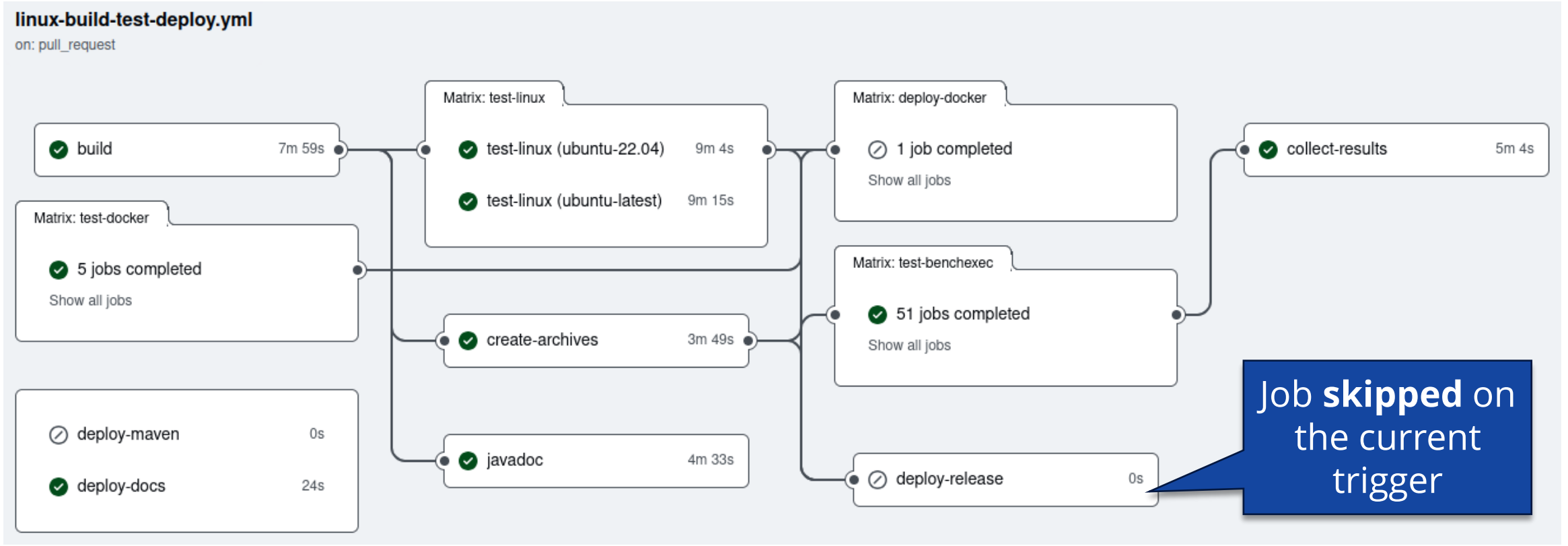
Common features



More complex example

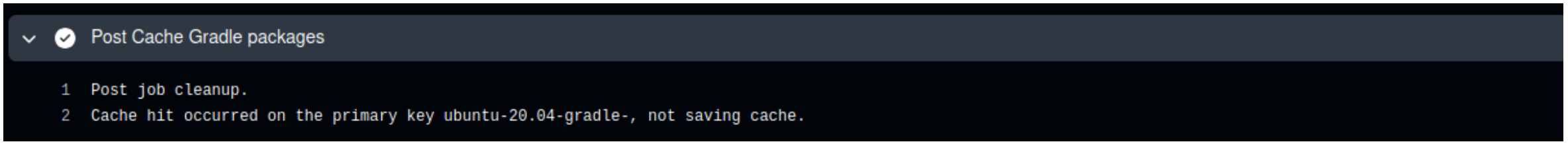


More complex example



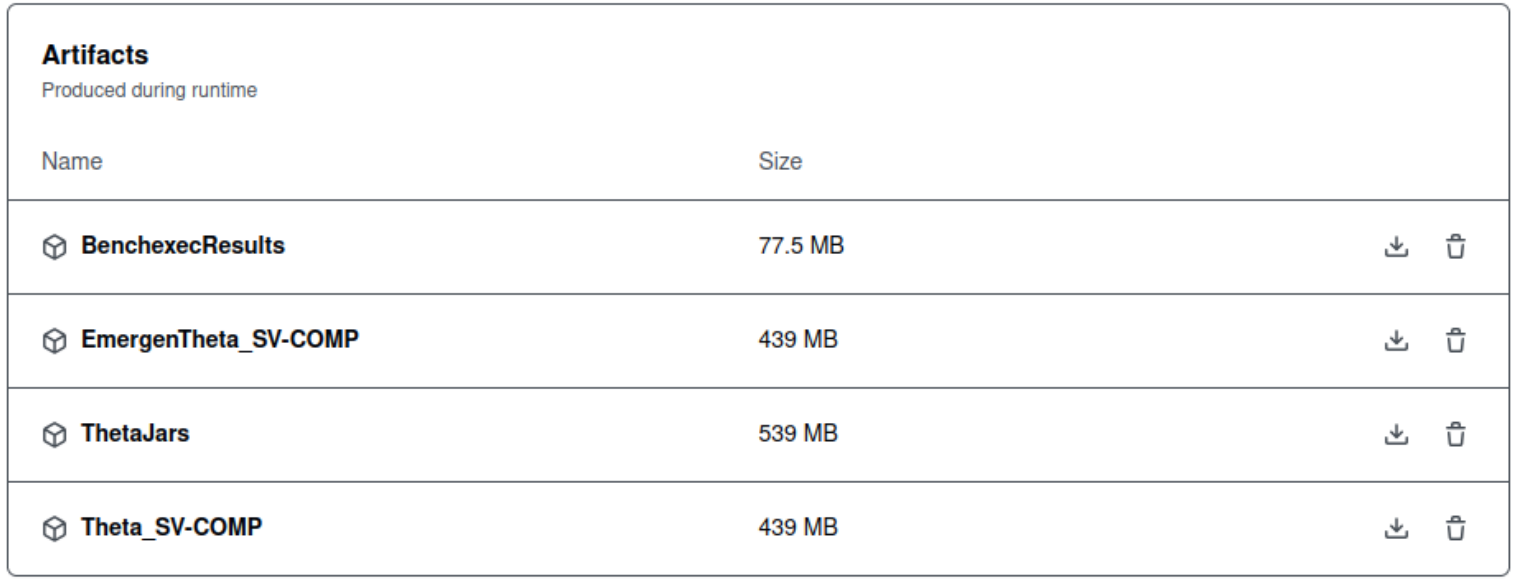
Common features

- **Build caching:** reduce the execution time of the pipeline by saving common input artifacts (e.g., downloaded dependencies)
  - **Warning:** inappropriate caches may make the build non-reproducible!



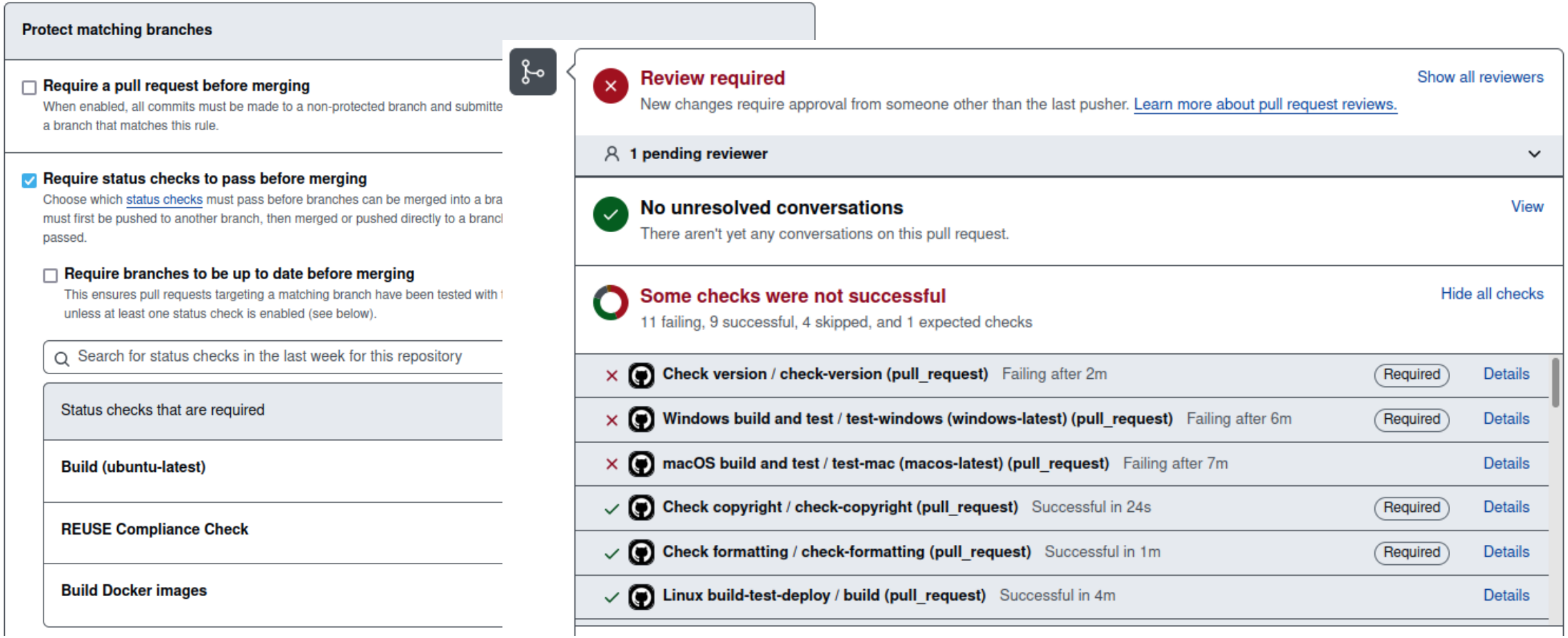
Common features

- **Artifacts:** save output to share between pipeline jobs or for later download
  - **Warning:** excessively large artifacts may slow down the pipeline and incur additional costs for storage and bandwidth!
  - Make sure to set an expiry date for the artifact



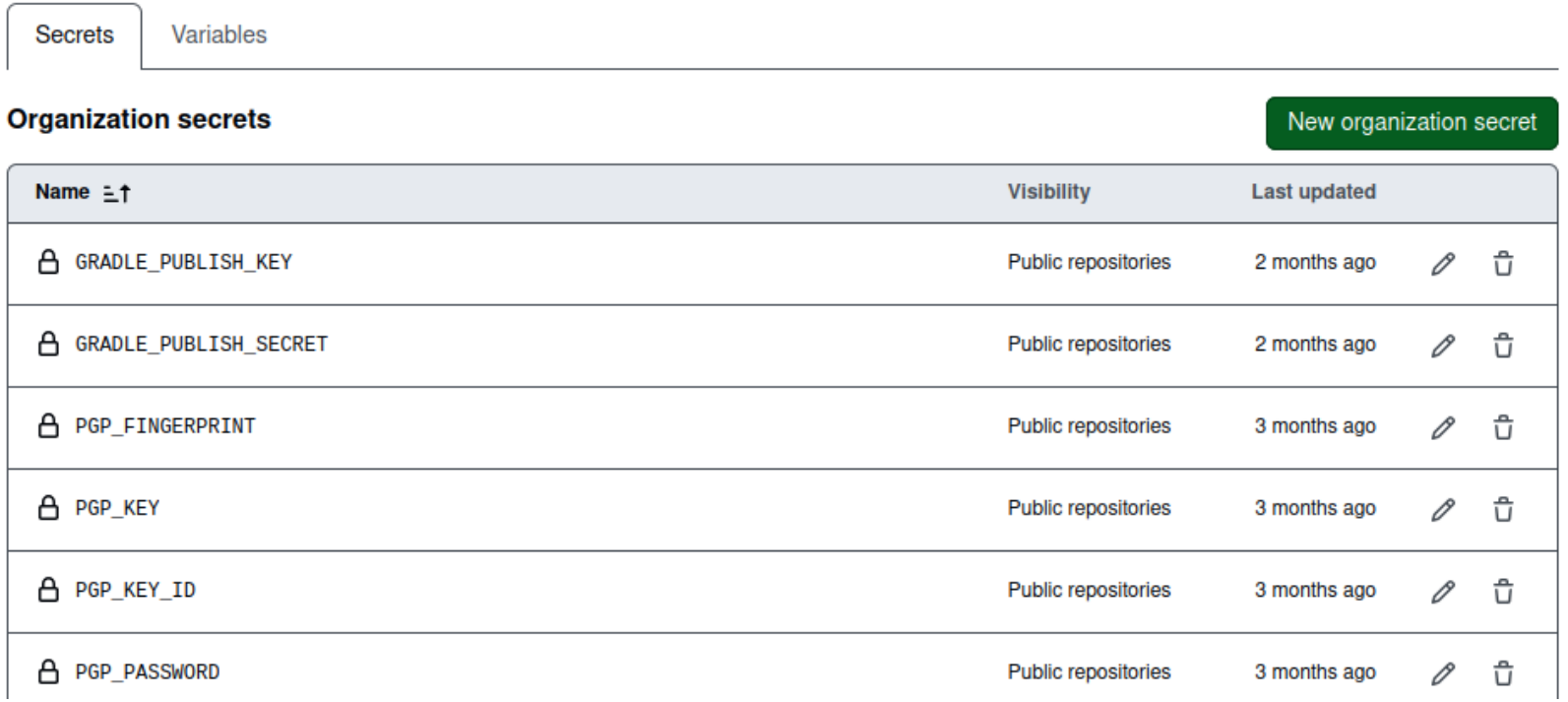
Common features

- **Branch protection:** only allow merges in the pipeline passes



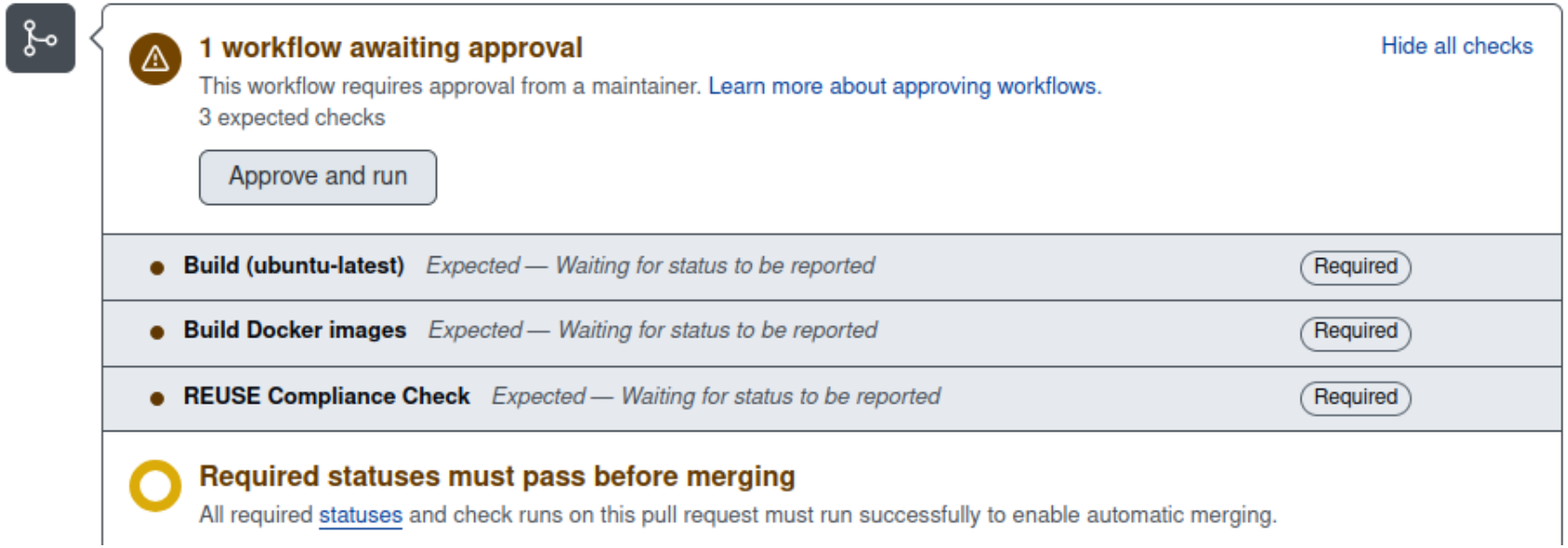
Common features

- **Secrets:** expose security keys and tokens to the pipeline without exposing them to the developer
  - Useful for **code signing** and **deployment**



Common features

- **Secrets:** expose security keys and tokens to the pipeline without exposing them to the developer
  - Useful for **code signing** and **deployment**
  - Still **requires trust:** if the developers can arbitrarily modify build scripts, secrets may be exfiltrated over the network



Environments in CI/CD

- **Development environment:** the development of the application and most automated tasks take place here
  - Developer machines, virtual and cloud based environments, CI/CD runners
- **Staging environment:** application is deployed here for end-to-end and user acceptance testing
  - Should be similar to the production environment, but usually smaller in scale
  - No access to private user data
  - The deployment is halted if defects are discovered
- **Production environment:** where the application runs

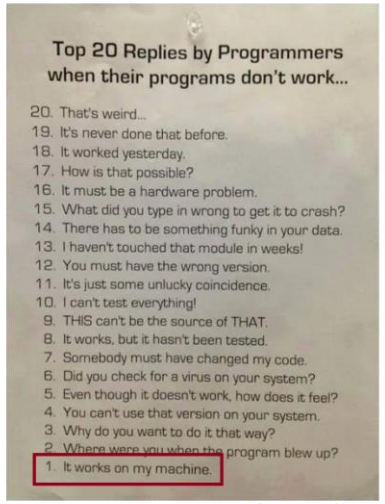
Communication between CI/CD and environments

- **Directly access** staging or production from the pipeline
  - Log in into the server (SSH) or upload files (FTP)
  - Requires the appropriate **secrets** exposed to the pipeline
  - Limited scalability (access single machine), not suitable for cloud environments
- **Webhooks:** notify the cloud by accessing a web link over HTTP
  - Upload the deployable artifacts to **shared storage,** then trigger the webhook to let the cloud **rollout** the new artifact
  - The cloud can also use a webhook to **trigger a pipeline** *(e.g., integration test after completing the staging rollout)*
  - Usually requires a **secret** in the URL to prevent unauthorized access
- **Deploy keys:** single-purpose keys to let the environment access the private source code repository

Summary

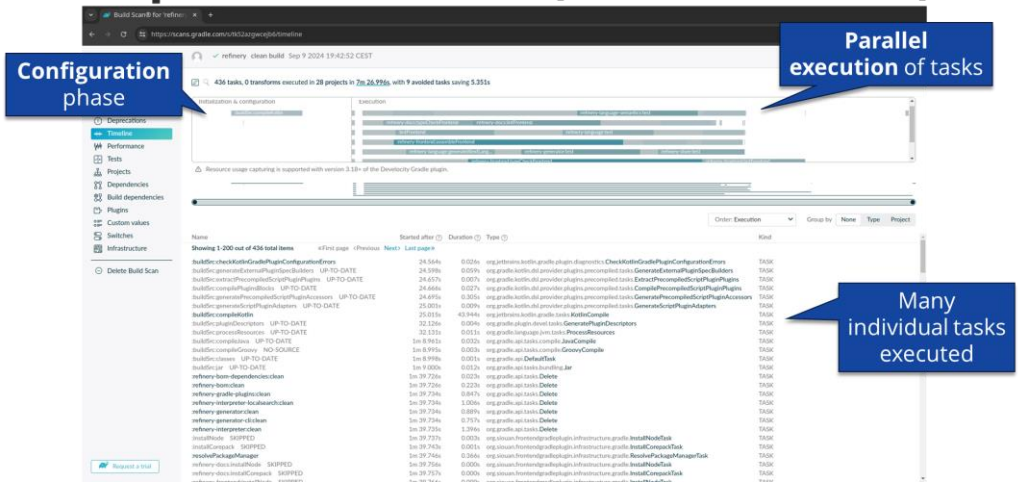
Build automation

- **Ad-hoc approach:** each developer builds the software locally to produce an executable
  - **Dependence** of the local environment
  - Lack of **reproducibility**
  - Difficult to determine what needs to be **rebuilt**
    - Out-of-date versions of artifacts may get 'stuck' due to a missed rebuild
  - Difficult to **onboard** new developers



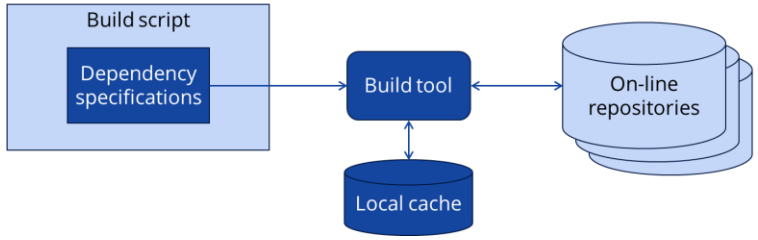
r/ProgrammerHumor

Example build execution (Gradle build scan)



Retrieving dependencies

- Not strictly part of build automation
  - E.g., Makefiles and Meson rely on the operating system to discover libraries
- But often supported by language-specific build tools
  - E.g., Maven repositories for Maven, NPM for NodeJS



Continuous Delivery (CD)

„build software so that it is always in a state where it could be put into production"

Source: https://martinfowler.com/bliki/ContinuousDelivery.html