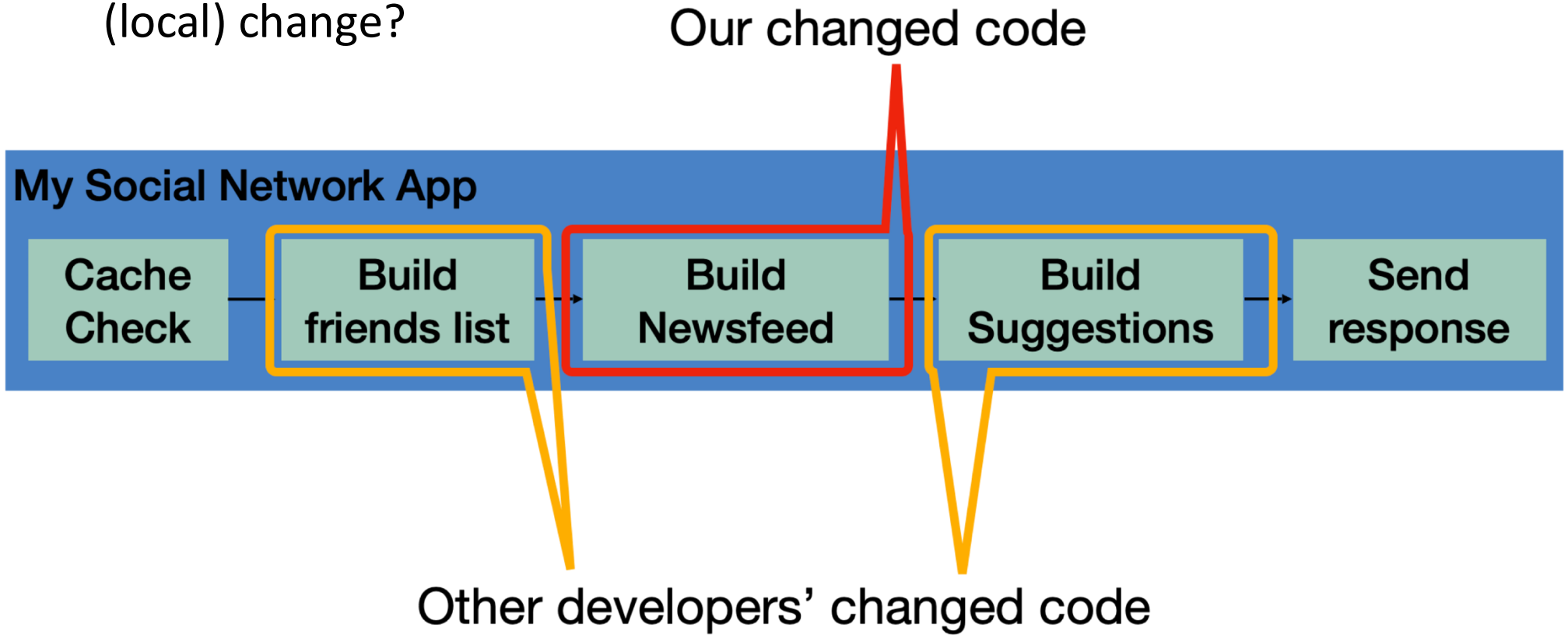
CS 4530: Fundamentals of Software Engineering
Lesson 5.3
Continuous Integration

Rob Simmons

Khoury College of Computer Sciences

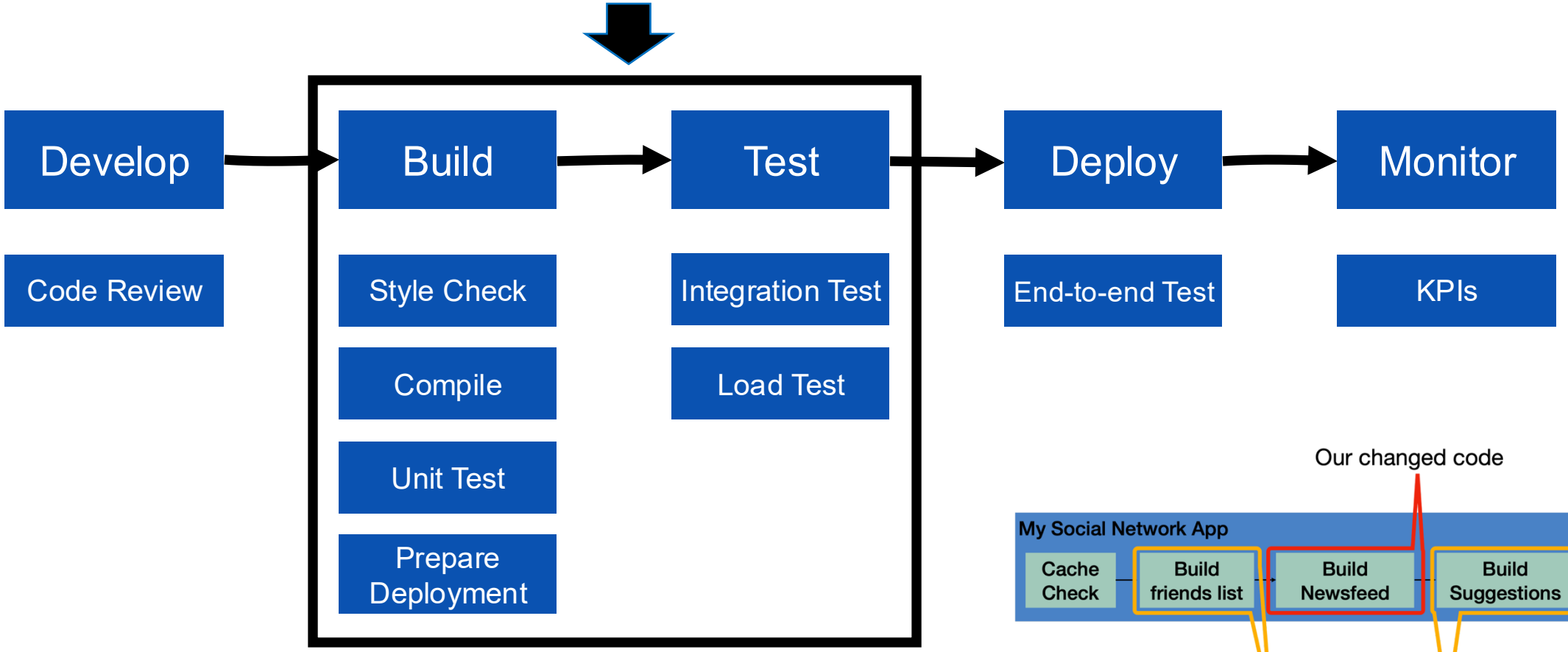
Continuous Integration (CI) provides global feedback on local changes

- Given: Our systems involve many components, some of which might even be in different version control repositories
- Consider: How does a developer get feedback on their (local) change?



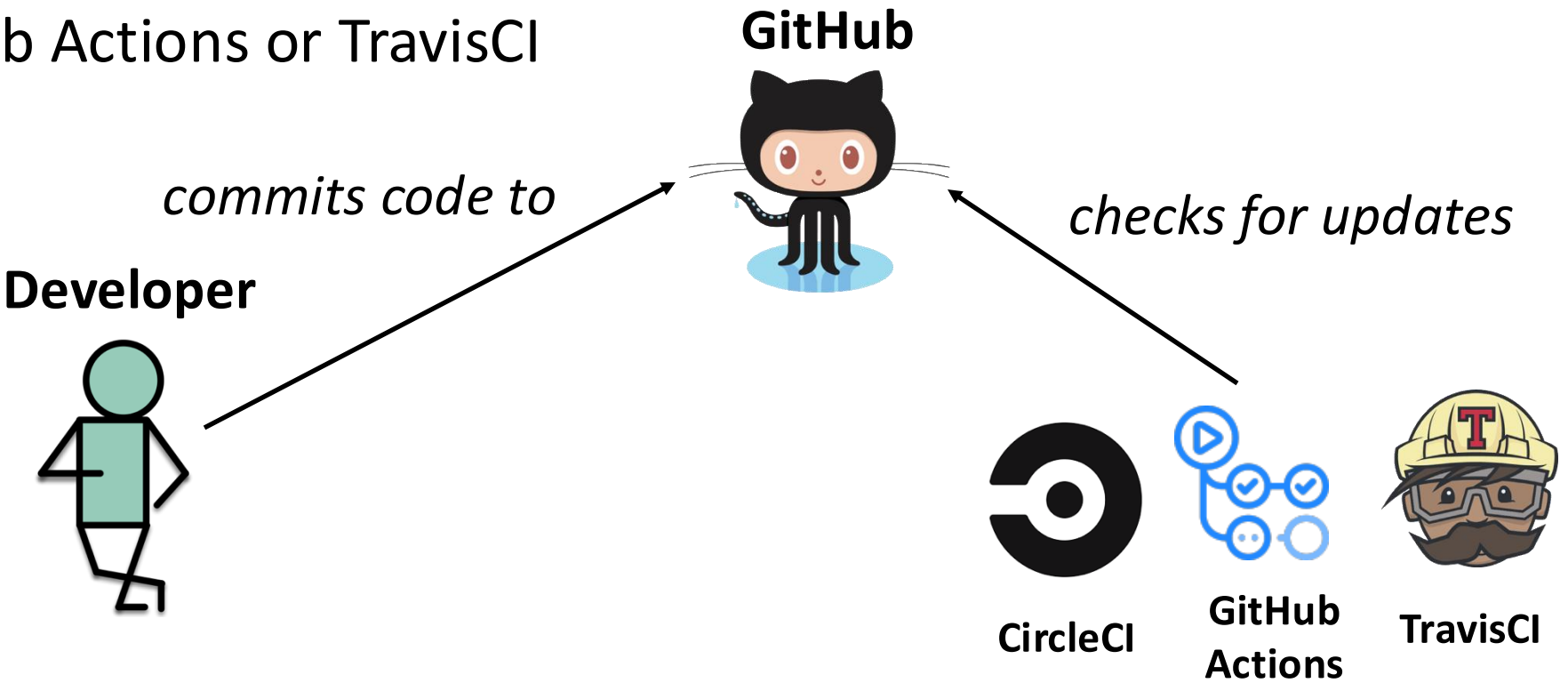
A CI process is a software pipeline

Automate this centrally, provide a central record of results



CI may be triggered by commits, pull requests, or other actions

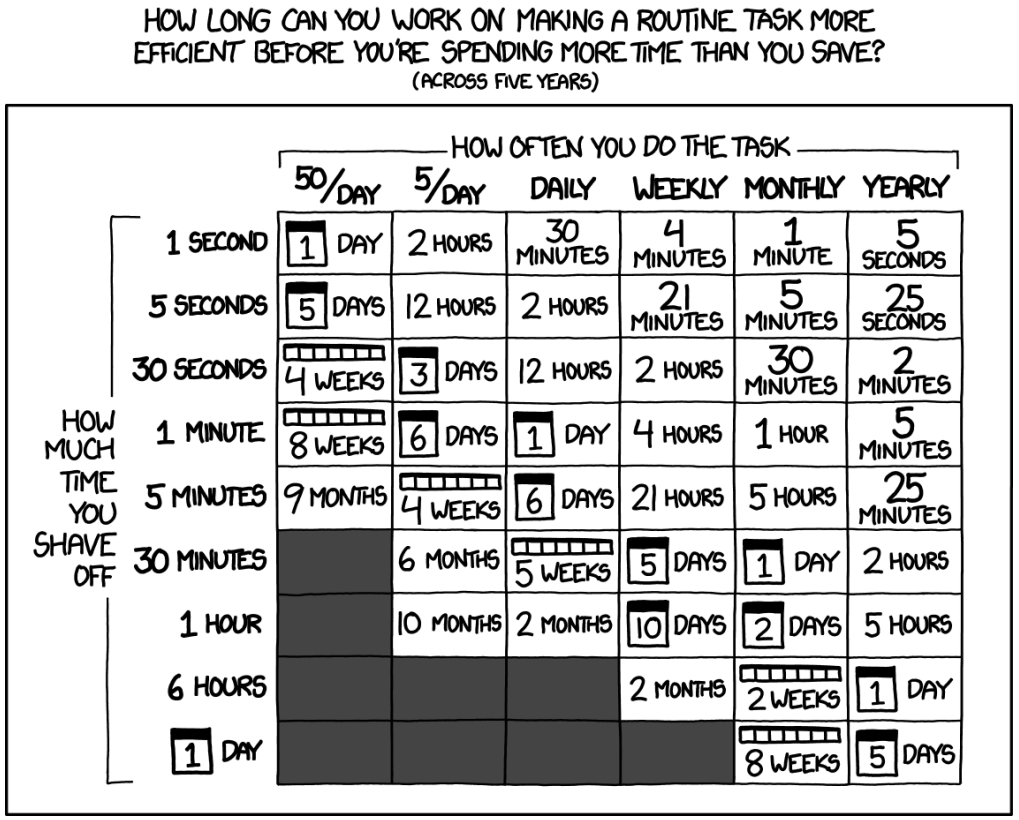
Example: Small scale CI, with a service like CircleCI, GitHub Actions or TravisCI



Runs build for each commit

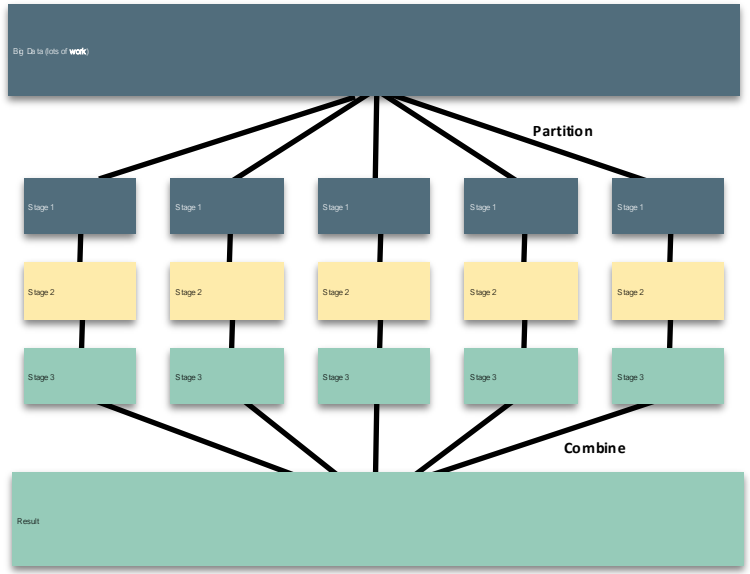
Automating Feedback Loops is Powerful

Consider tasks that are done by *dozens* of developers (e.g. testing/deployment)



Typical CI pipeline

- Set up testing environment
- Set up tests
- Set up multiple input
- Run all tests against all inputs
  - (preferably in parallel)
- Record results and performance in central db

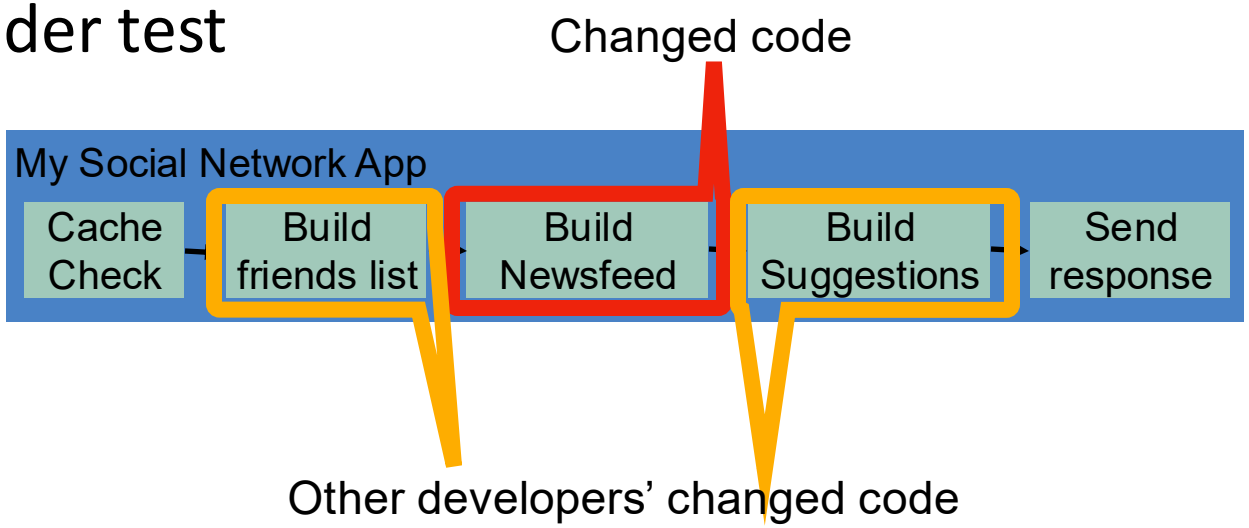


You could set up multiple CI processes

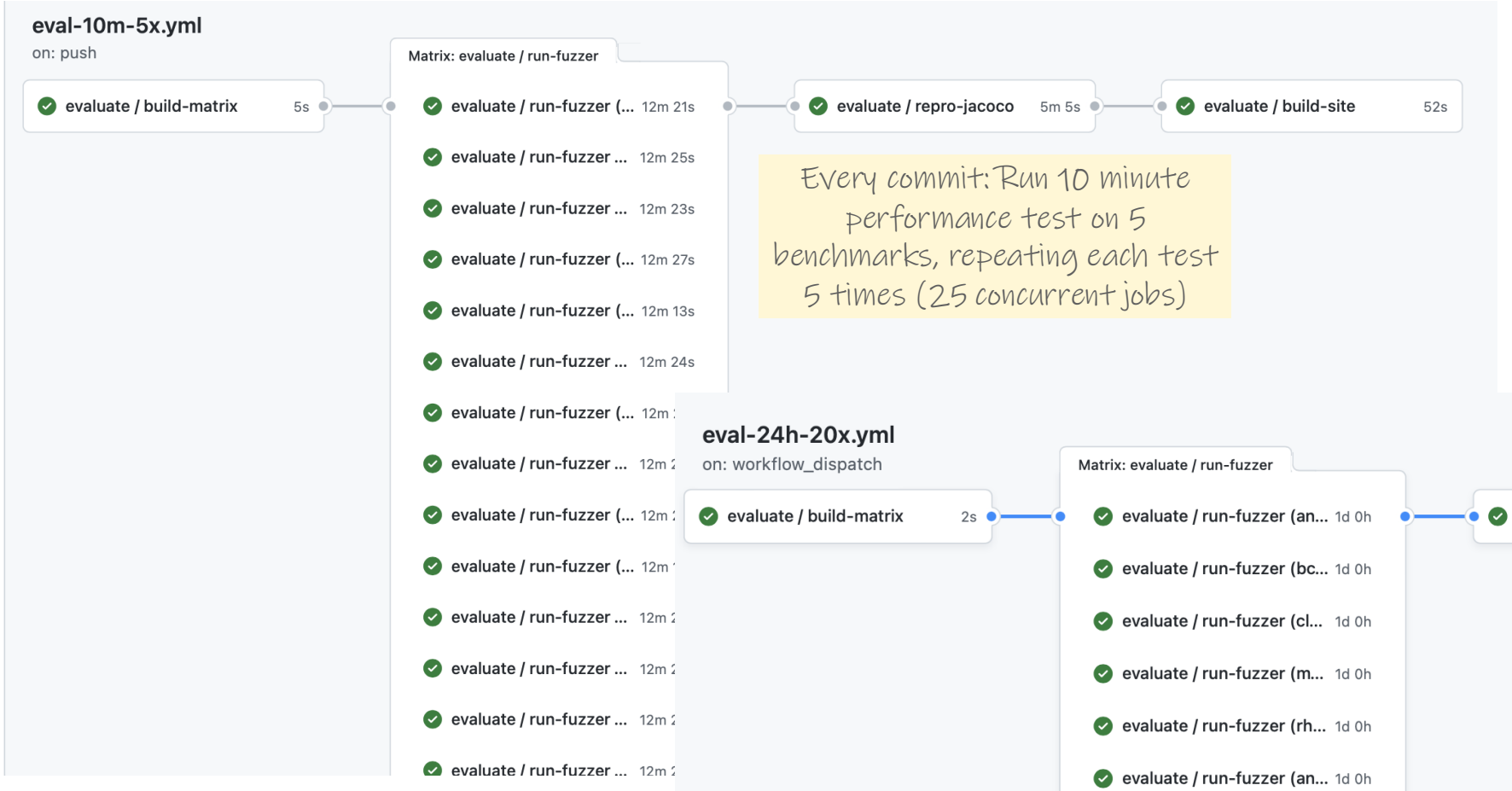
- Run a short test daily
  - or oftener
  - maybe on every commit?
- More comprehensive test less often
  - provides more accurate performance data
- Either way, you know that your integration is working!

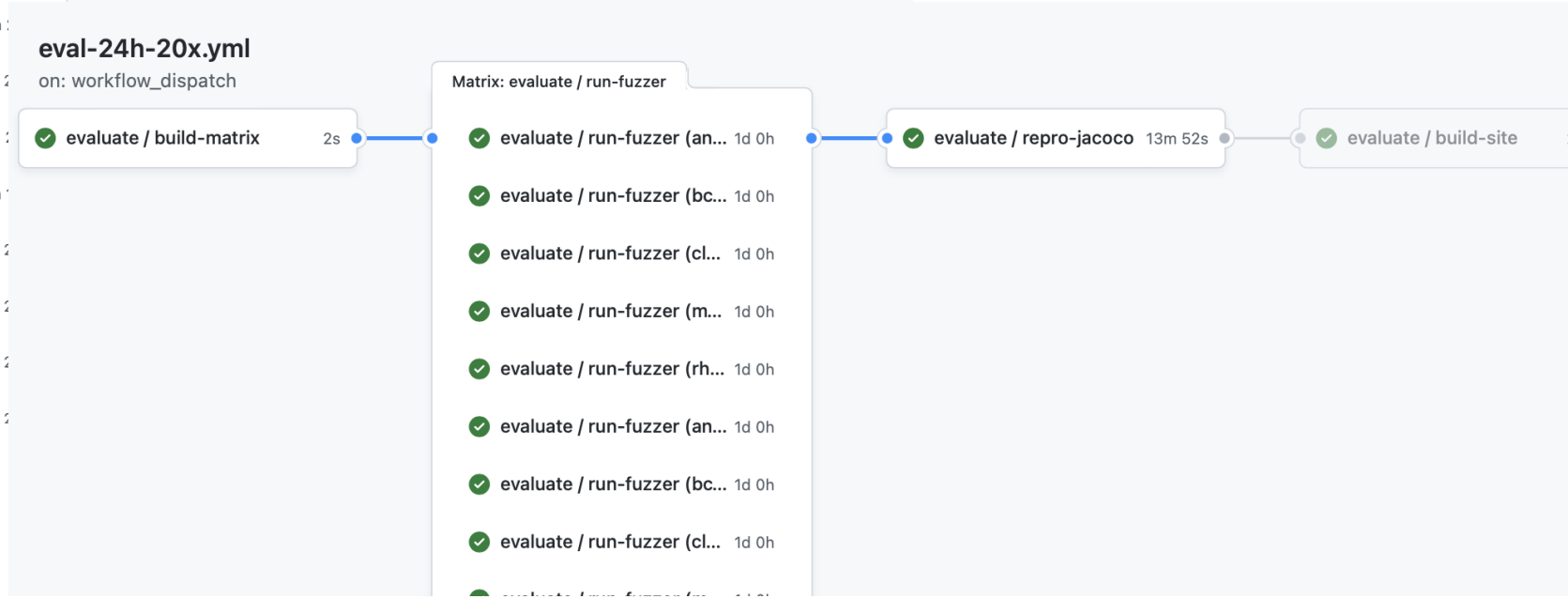
Continuous Integration is Highly Configurable

- Determining *how* to apply CI can be non-trivial for a larger project, all with a cost vs quality tradeoff: what is the cost of automation vs the value of developer time?
- Do we integrate changes immediately, or do a pre-commit test?
- Which tests do we run when we integrate?
- When do we integrate code review?
- How do we compose the system under test at each point?



CI pipelines can automate performance testing



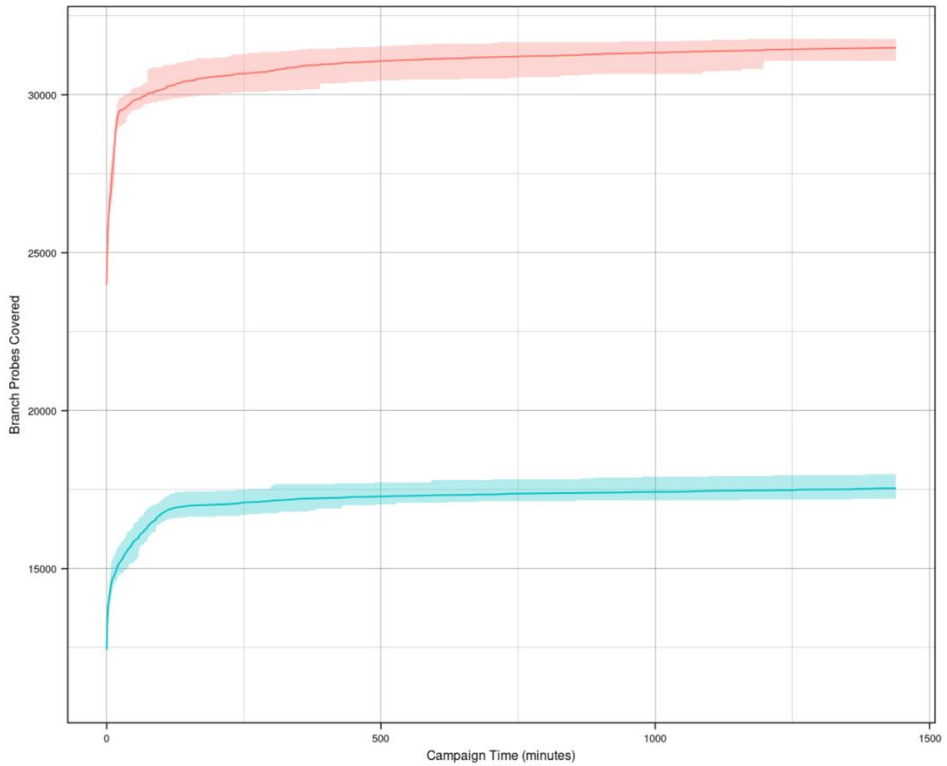


https://github.com/neu-se/CONFETTI/actions

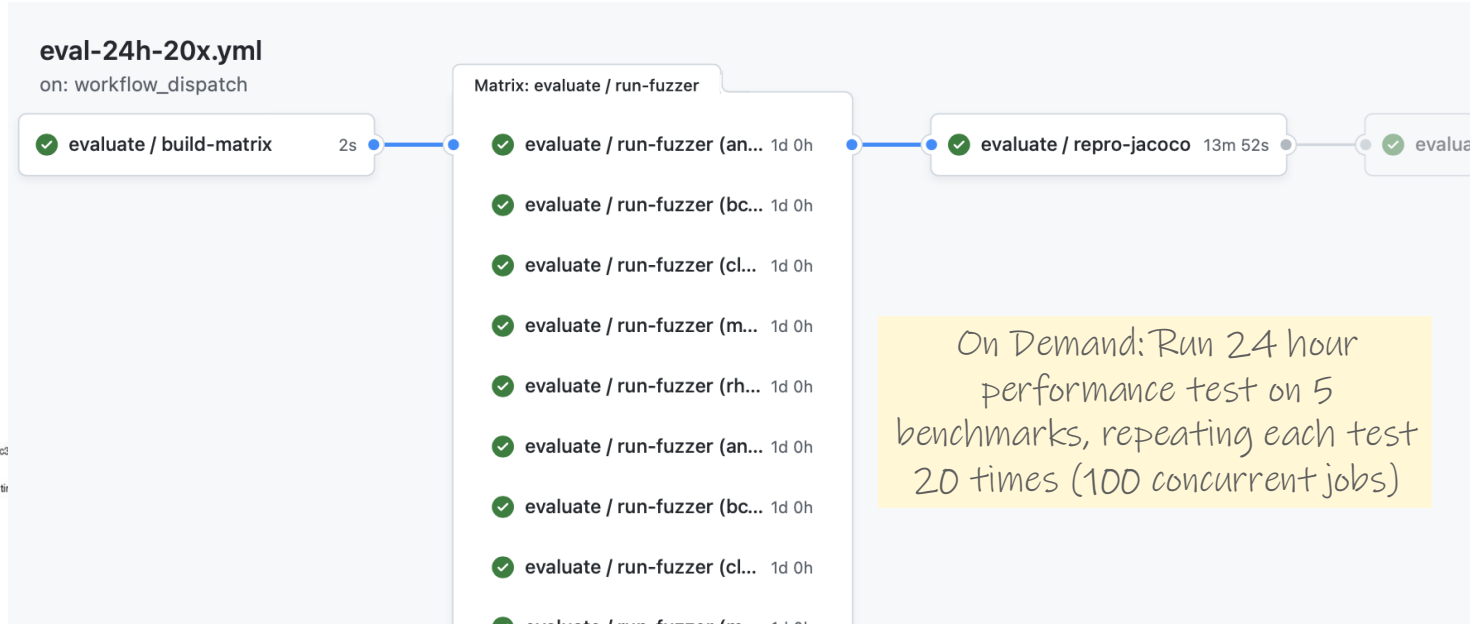
CI pipelines can automate benchmarking

closure

Branch Probes Over Time



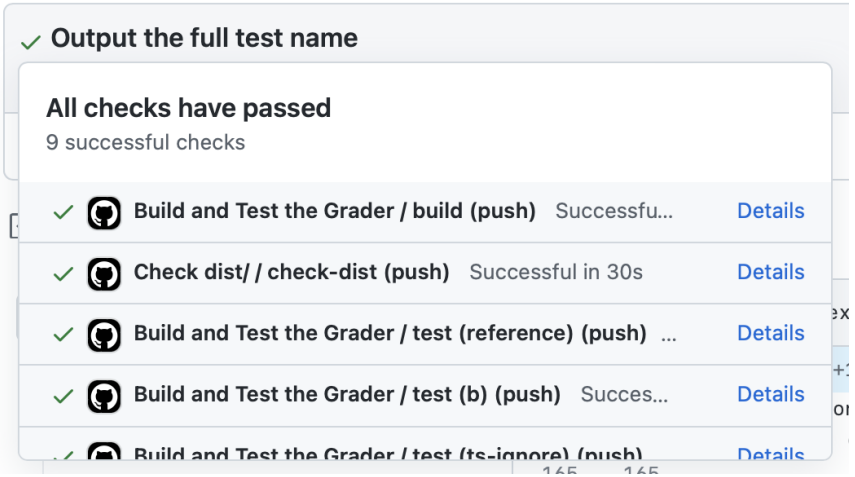
Download this graph as PDF

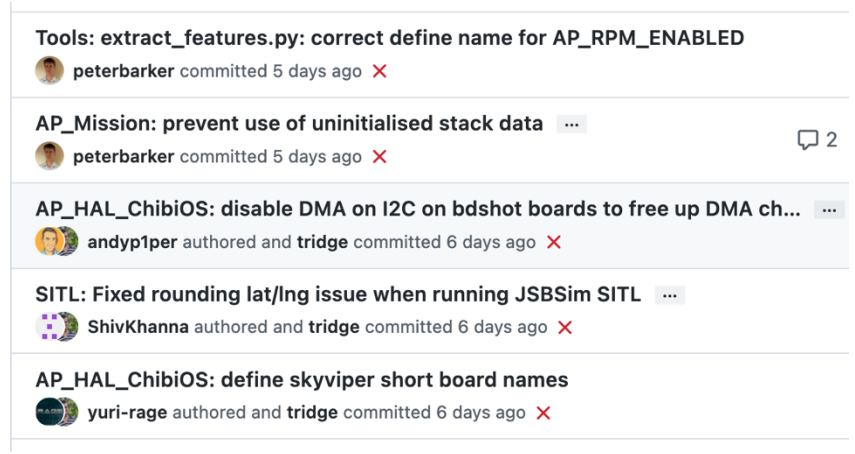


https://github.com/neu-se/CONFETTI/actions

Attributes of effective CI processes

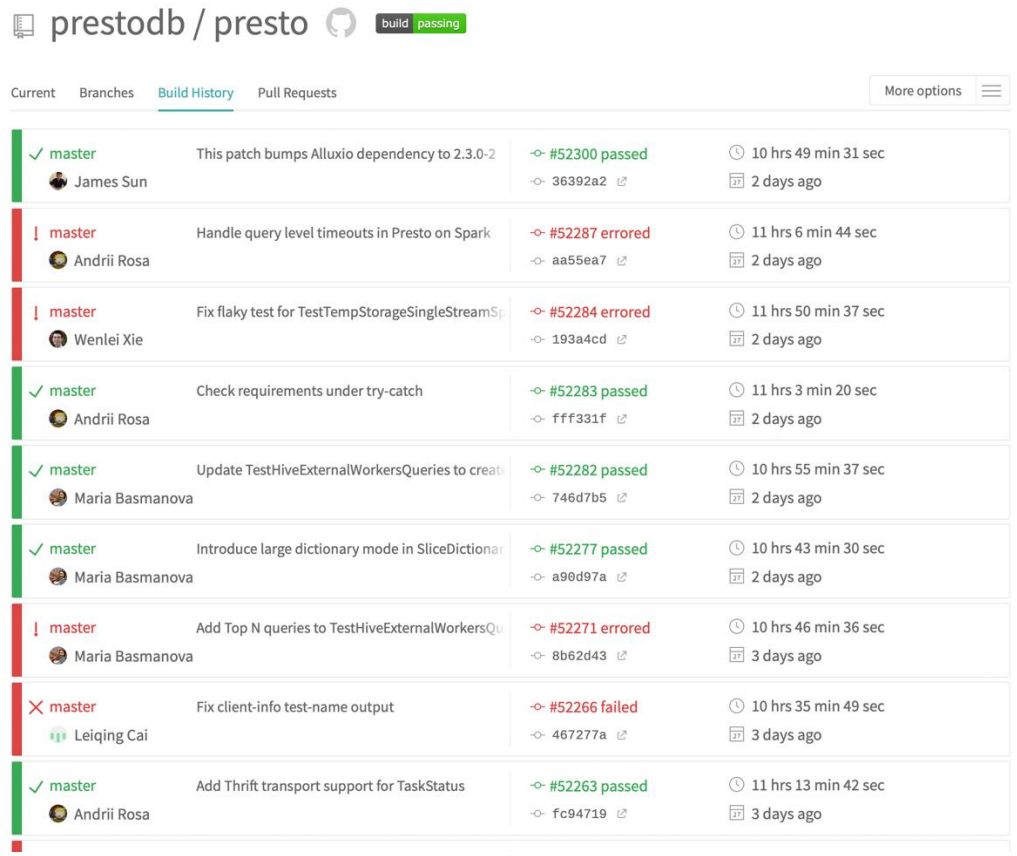
- Policies:
  - Do not allow builds to remain broken for a long time
  - CI should run for every change
  - CI should not completely replace pre-commit testing
- Infrastructure:
  - CI should be fast, providing feedback within minutes or hours
  - CI should be repeatable (deterministic)





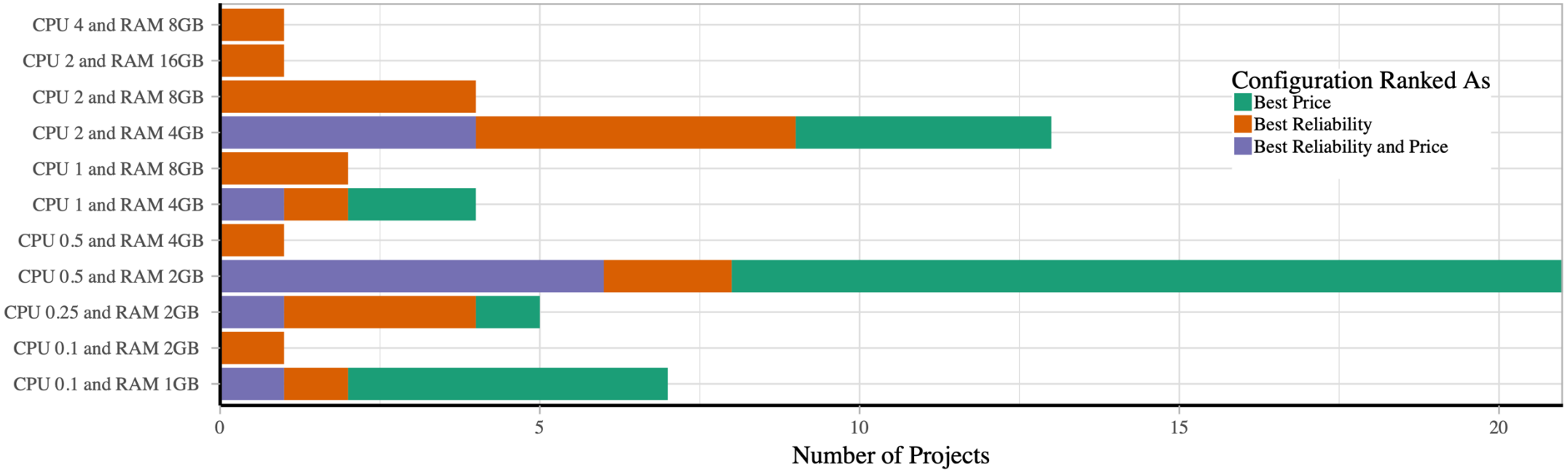
Effective CI processes are run often enough to reduce debugging effort

- Failed CI runs indicate a bug was introduced, and caught in that run
- More changes per-CI run require more manual debugging effort to assign blame
- A single change per-CI run pinpoints the culprit



Effective CI processes allocate enough resources to mitigate flaky tests

- *Flaky* tests might be dependent on timing (failing due to timeouts)
- Running tests without enough CPU/RAM can result in increased flaky failure rates and unreliable builds



"The Effects of Computational Resources on Flaky Tests", Silva et al

Challenges and Solutions for Repeatable Builds

- Which commands to run to produce an executable? (build systems)
- How to link third-party libraries? (dependency managers)
- How to specify system-level software requirements? (containers)
- How to specify infrastructure requirements? (Infrastructure as code)

Build Systems Orchestrate Software Engineering Tasks

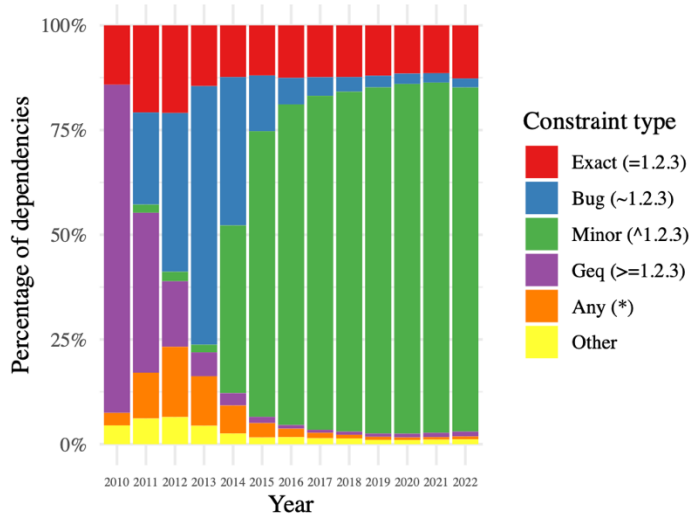
- “Orchestrate” -> Execute in the right order, ideally with concurrency, example tasks:
  - Installing dependencies
  - Compiling the code
  - Running static analysis
  - Generating documentation
  - Running tests
  - Creating artifacts for customers
  - Deploying Code
- Example build systems: xMake, ant, maven, gradle, npm…

Dependency Managers Organize External Dependencies

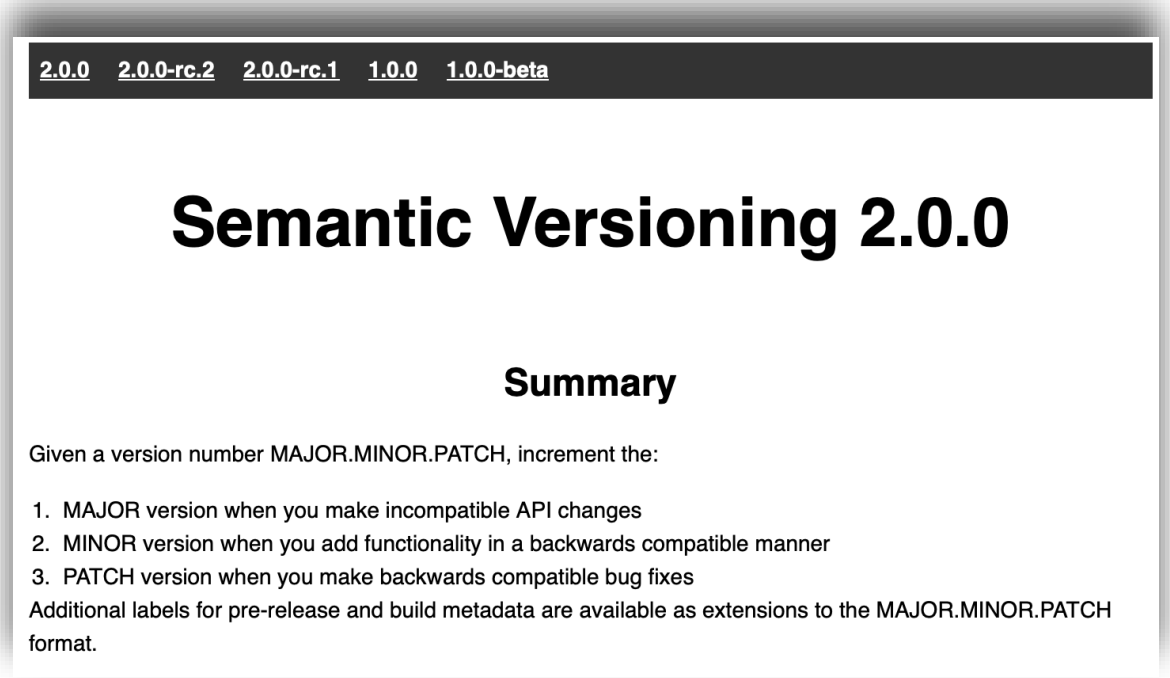
- Addresses this problem: “Before you compile this code, install commons-lang from the Apache website”
- Declare a dependency using coordinates (unique ID of a package plus version)
- Packages are archived in common repositories; fetched/linked by dependency manager
- Dependency managers handle transitive dependencies 
- Examples: Maven, NPM, pip, cargo, apt

Specify and Depend on Package Versions with Care

- Semantic Versioning is often expected:
  - Library maintainers expected to indicate breaking changes with version numbers
  - Dependency consumers can specify constraints on versions (e.g. accept 2.0.x)



Distribution of dependencies of all packages in NPM over time (2023, Pinckney et al)



Continuous Integration Service Models

- Self-hosted/managed on-premises or in cloud
  - Jenkins
- Fully cloud managed
  - GitHub Actions, CircleCI, Travis, many more…
  - Billing model: pay per-build-minute running on SaaS infrastructure
  - “Self-hosted runners” run builds on your own infrastructure, usually “free”