CS 4530: Fundamentals of Software Engineering
Module 13: Continuous Development

Adeel Bhutta and Mitch Wand

Khoury College of Computer Sciences

Learning objectives for this lesson

- By the end of this lesson, you should be able to…
  - Describe how continuous integration helps to catch errors sooner in the software lifecycle
  - Describe strategies for performing quality-assurance on software as and after it is delivered
  - Compare and contrast continuous delivery with test driven development as a quality assurance strategy

Review: The Agile Model Reduces Risk by Embracing Change (~2000)

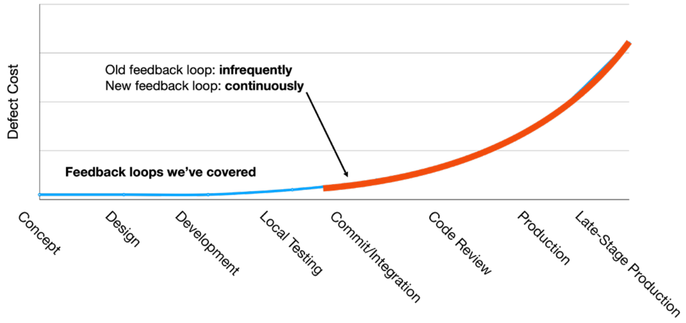
- The Waterfall philosophy:
  - "The project is too large and complex, and it will take months (or years!) to plan, so once we come up with the plan, that plan can not change"
  - Reduce risk by proceeding in stages
- The Agile philosophy:
  - The project is too large and complex, it is unlikely that we will know exactly what we need right now, and to some extent, we are inventing something new. We think that as we make it, we will figure it out as we go”
  - Reduce risk by limiting time on any one stage; then reassess. (“time-boxing”)
  - Reduce risk through automated testing

Agile relies on a variety of quality-assurance processes

- What are the costs & benefits of each of these?
  - unit testing/TDD
  - code review
  - integration tests (as in module 12)
  - continuous integration
  - continuous deployment (A/B, canaries, etc.)
- How is each automatable?
- How does each address non-functional quality attributes?
- How should these be combined in an organization's software development process?

In this module, we'll focus on CI/CD

- What are the costs & benefits of each of these?
  - unit testing/TDD
  - code review
  - integration tests (as in module 12)
  - continuous integration
  - continuous deployment (A/B, canaries, etc.)
- How is each automatable?
- How does each address non-functional quality attributes?
- How should these be combined in an organization's software development process?

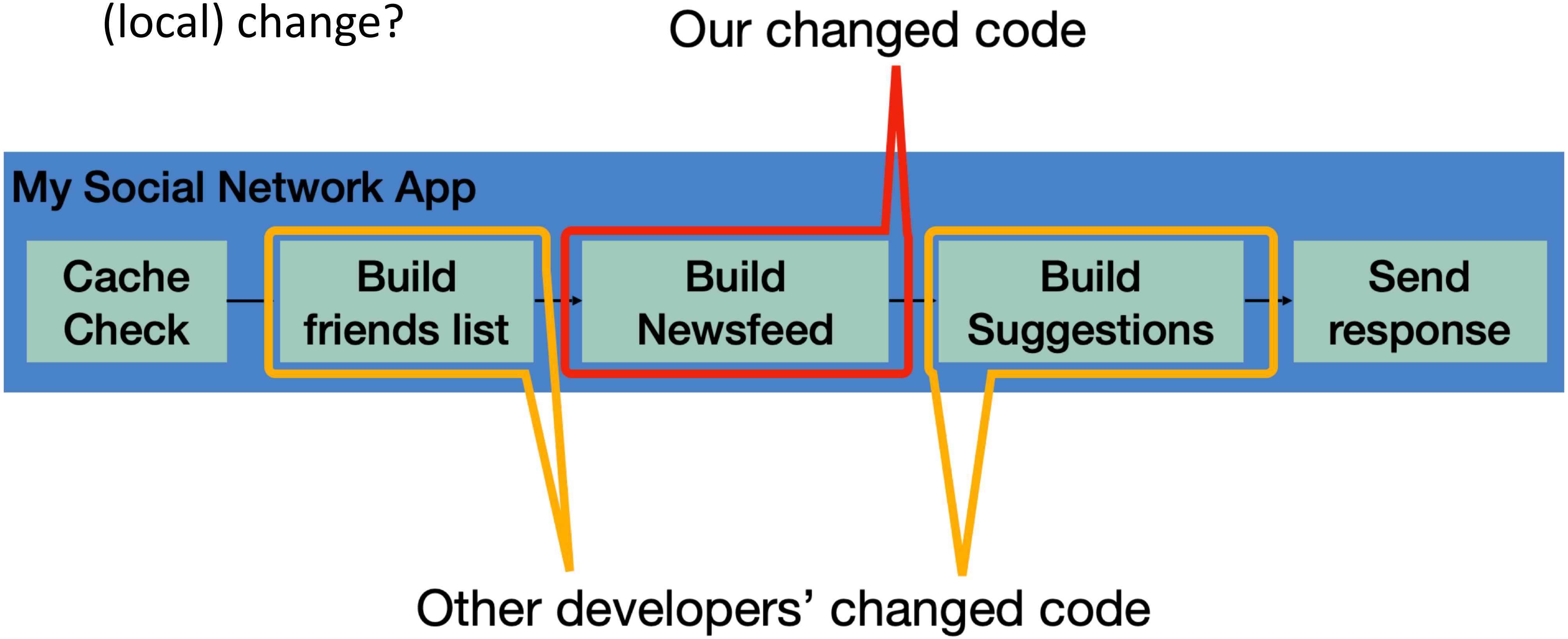


CI/CD practices improve code quality and dev velocity

- Continuous integration: use automated systems to perform and monitor frequent integrations with entire codebase, running integration-scale tests
- Continuous delivery: use automated systems to perform frequent, controlled delivery of product (often to a small fraction of the user base), with automated monitoring to detect remaining defects quickly.

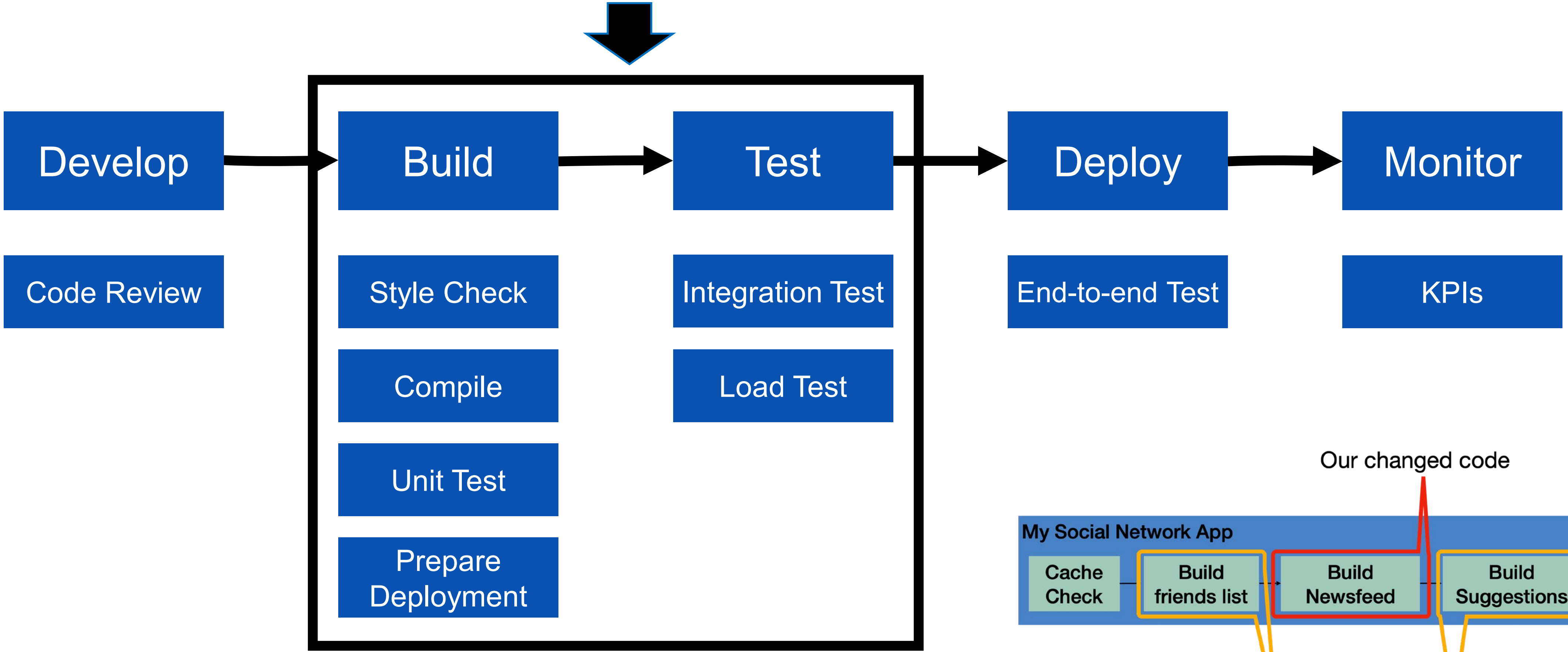
Continuous Integration (CI) provides global feedback on local changes

- 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



Develop

Code Review

Build

Style Check

Compile

Unit Test

Prepare Deployment

Test

Integration Test

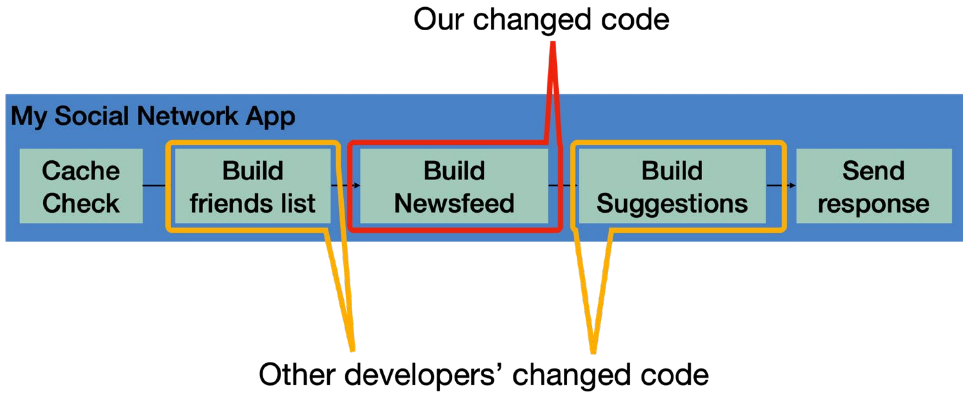
Load Test

Deploy

End-to-end Test

Monitor

KPIs



Our changed code

My Social Network App

Cache Check

Build friends list

Build Newsfeed

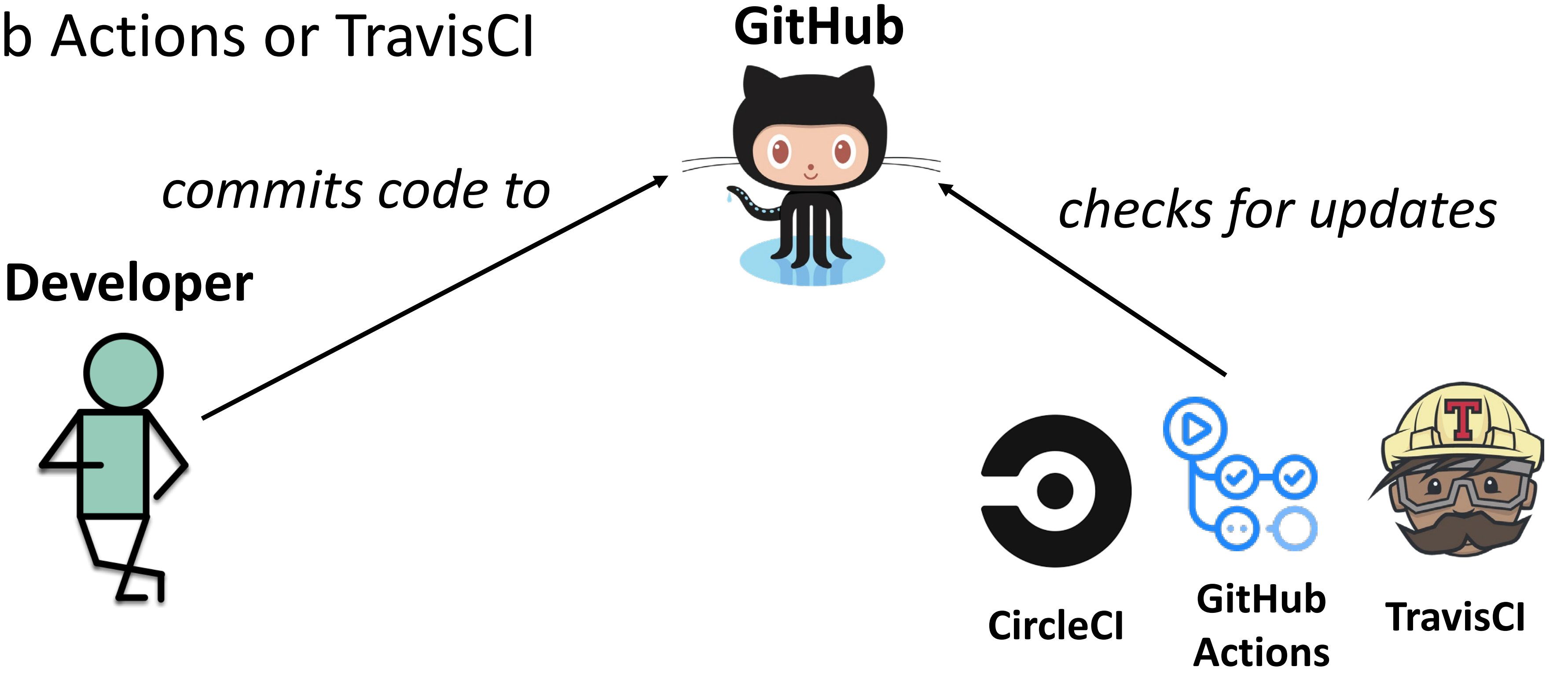
Build Suggestions

Send response

Other developers' changed code

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

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



GitHub

commits code to

checks for updates

Developer

CircleCI

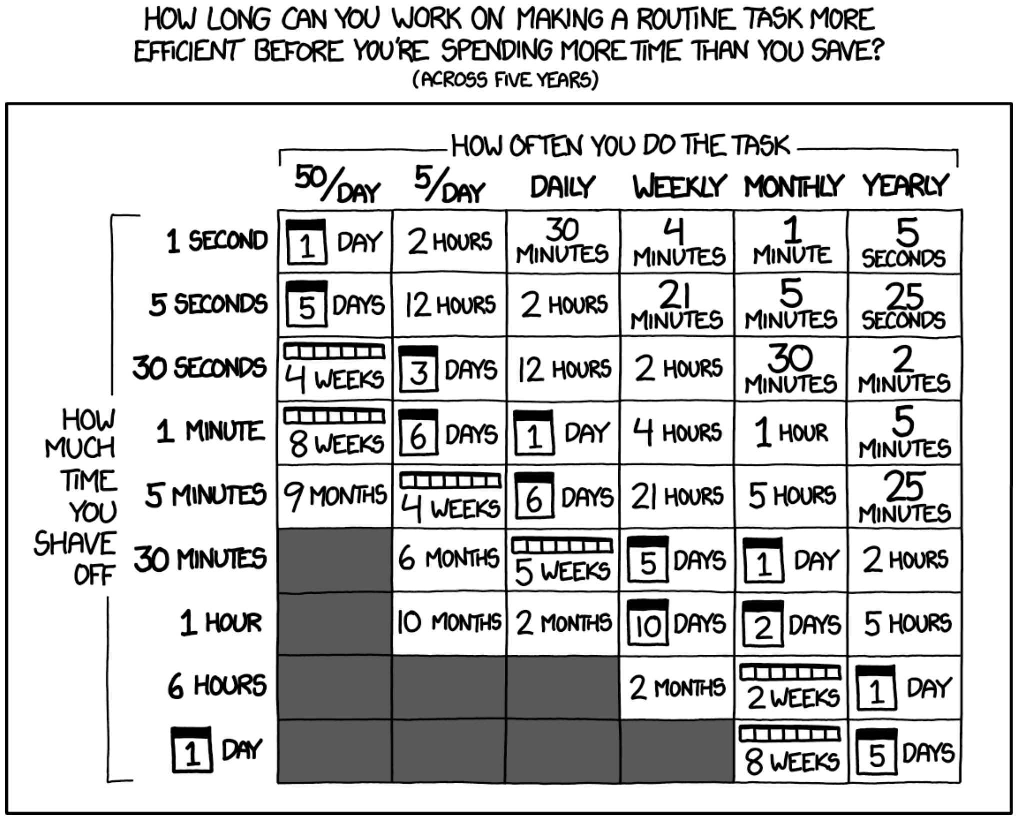
GitHub Actions

TravisCI

Runs build for each commit

Automating Feedback Loops is Powerful

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



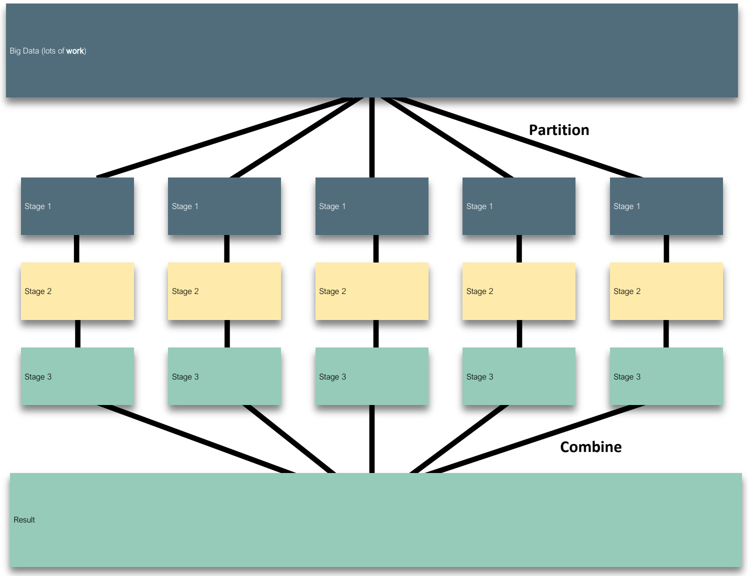
HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?
(ACROSS FIVE YEARS)

		HOW OFTEN YOU DO THE TASK					
		50/DAY	5/DAY	DAILY	WEEKLY	MONTHLY	YEARLY
HOW MUCH TIME YOU SHAVE OFF	1 SECOND	1 DAY	2 HOURS	30 MINUTES	4 MINUTES	1 MINUTE	5 SECONDS
	5 SECONDS	5 DAYS	12 HOURS	2 HOURS	21 MINUTES	5 MINUTES	25 SECONDS
	30 SECONDS	4 WEEKS	3 DAYS	12 HOURS	2 HOURS	30 MINUTES	2 MINUTES
	1 MINUTE	8 WEEKS	6 DAYS	1 DAY	4 HOURS	1 HOUR	5 MINUTES
	5 MINUTES	9 MONTHS	4 WEEKS	6 DAYS	21 HOURS	5 HOURS	25 MINUTES
	30 MINUTES	■	6 MONTHS	5 WEEKS	5 DAYS	1 DAY	2 HOURS
	1 HOUR	■	10 MONTHS	2 MONTHS	10 DAYS	2 DAYS	5 HOURS
	6 HOURS	■	■	■	2 MONTHS	2 WEEKS	1 DAY
	1 DAY	■	■	■	■	8 WEEKS	5 DAYS

https://xkcd.com/1205/

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

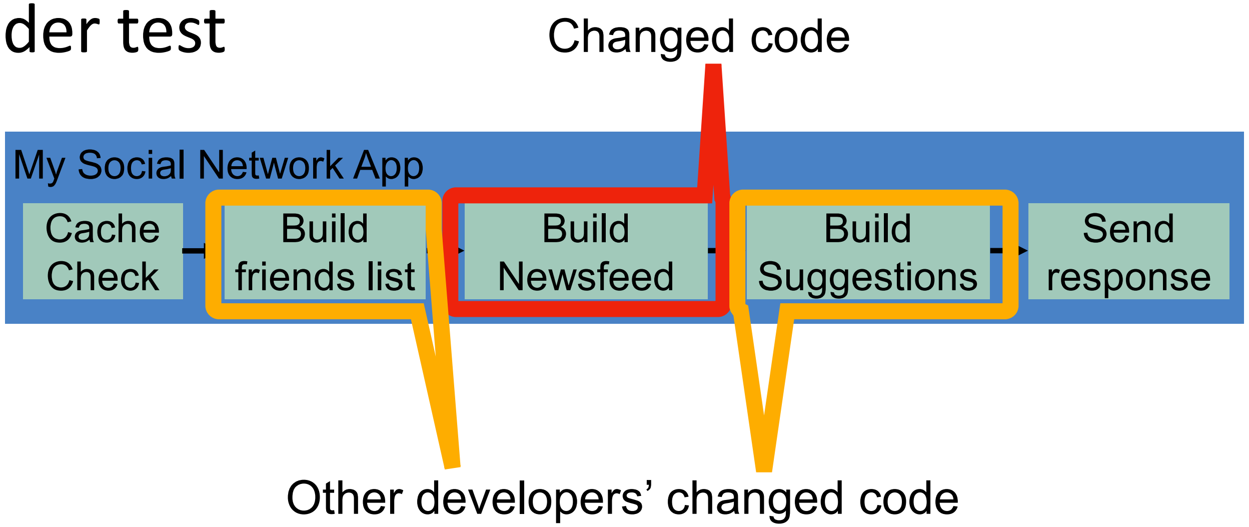


You could set up multiple CI processes

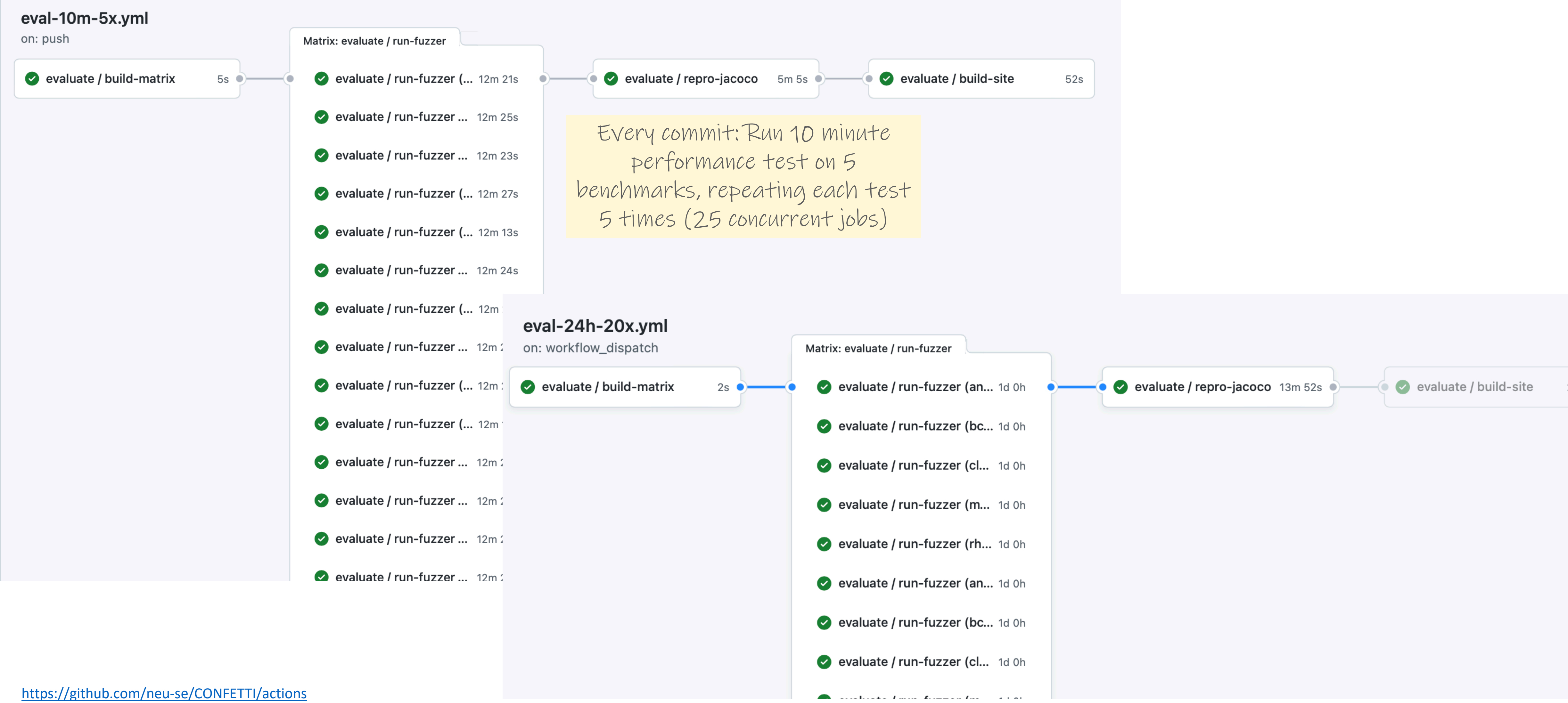
- Run a short test daily
  - or oftener
  - maybe on every commit?
- More comprehensive test less often
  - provides more accurate performance data
- Either way, you know that your integration is working!

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



eval-10m-5x.yml

on: push

✓ evaluate / build-matrix 5s

Matrix: evaluate / run-fuzzer

✓ evaluate / run-fuzzer (... 12m 21s

✓ evaluate / run-fuzzer ... 12m 25s

✓ evaluate / run-fuzzer ... 12m 23s

✓ evaluate / run-fuzzer (... 12m 27s

✓ evaluate / run-fuzzer (... 12m 13s

✓ evaluate / run-fuzzer ... 12m 24s

✓ evaluate / run-fuzzer (... 12m

✓ evaluate / run-fuzzer ... 12m

✓ evaluate / run-fuzzer (... 12m

✓ evaluate / run-fuzzer (... 12m

✓ evaluate / run-fuzzer ... 12m

✓ evaluate / run-fuzzer ... 12m

✓ evaluate / run-fuzzer ... 12m

✓ evaluate / run-fuzzer ... 12m

✓ evaluate / repro-jacoco 5m 5s

✓ evaluate / build-site 52s

Every commit: Run 10 minute performance test on 5 benchmarks, repeating each test 5 times (25 concurrent jobs)

eval-24h-20x.yml

on: workflow_dispatch

✓ evaluate / build-matrix 2s

Matrix: evaluate / run-fuzzer

✓ evaluate / run-fuzzer (an... 1d 0h

✓ evaluate / run-fuzzer (bc... 1d 0h

✓ evaluate / run-fuzzer (cl... 1d 0h

✓ evaluate / run-fuzzer (m... 1d 0h

✓ evaluate / run-fuzzer (rh... 1d 0h

✓ evaluate / run-fuzzer (an... 1d 0h

✓ evaluate / run-fuzzer (bc... 1d 0h

✓ evaluate / run-fuzzer (cl... 1d 0h

✓ evaluate / repro-jacoco 13m 52s

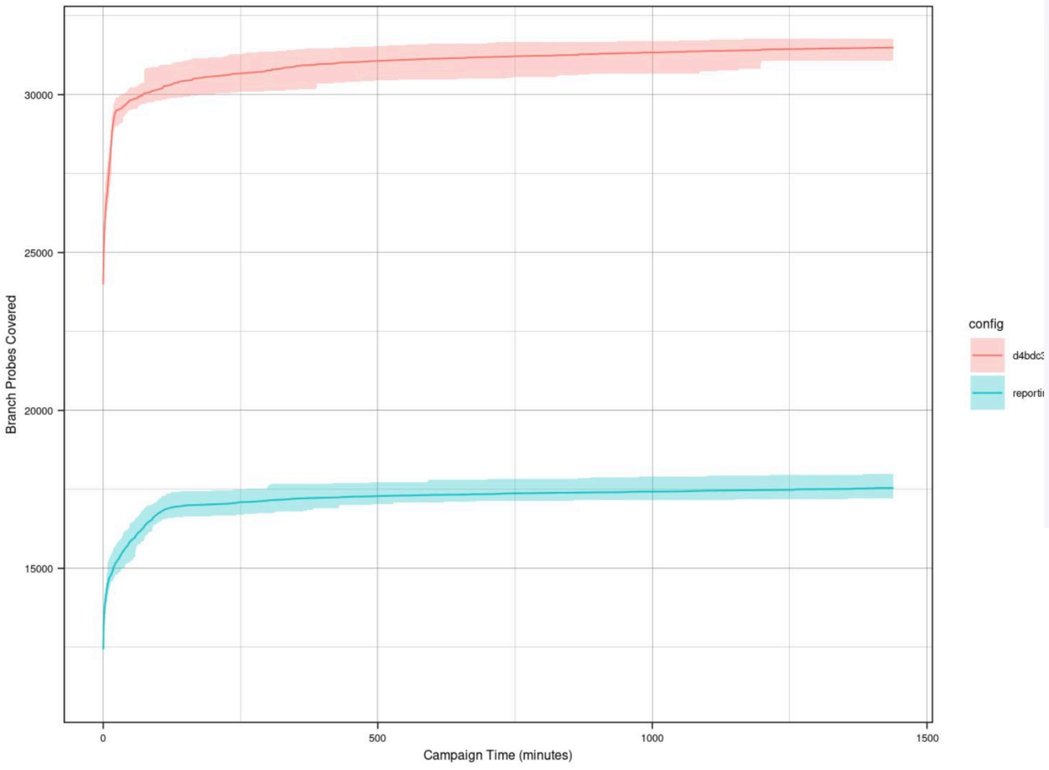
✓ evaluate / build-site

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

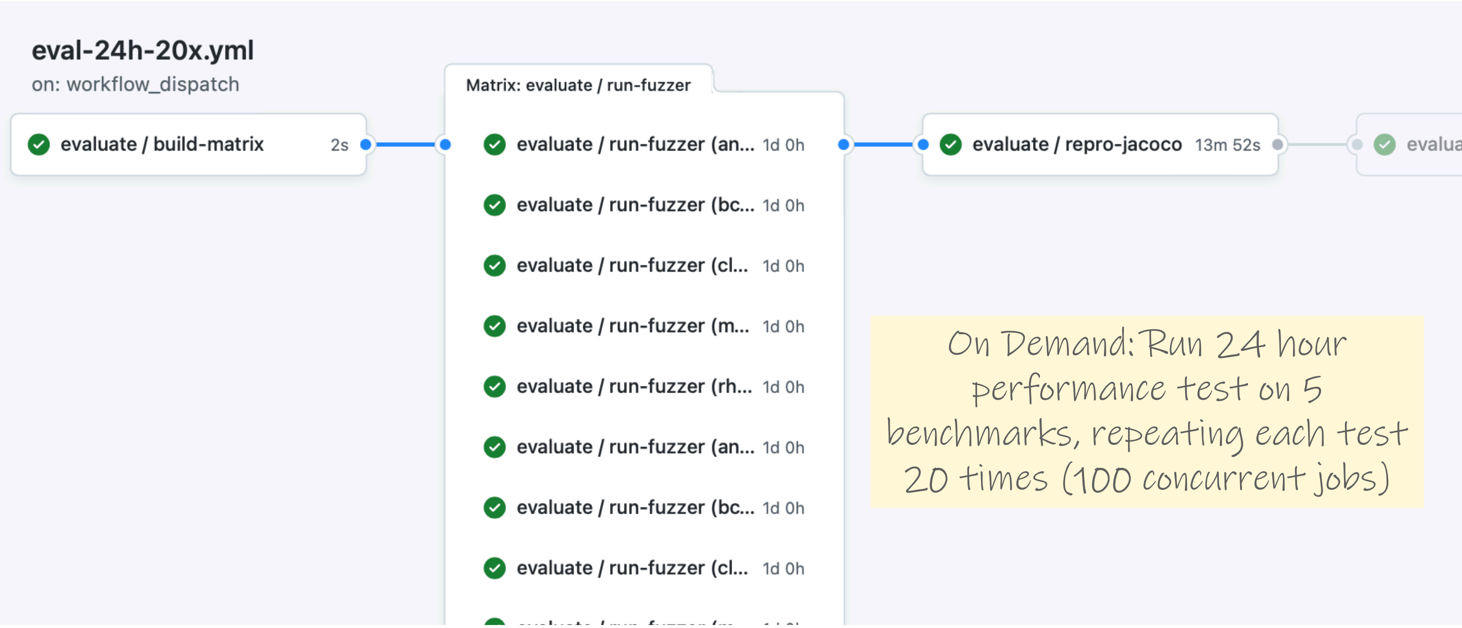
CI pipelines can automate benchmarking

closure

Branch Probes Over Time



Download this graph as PDF



eval-24h-20x.yml
on: workflow_dispatch

- ✓ evaluate / build-matrix 2s

Matrix: evaluate / run-fuzzer

- ✓ evaluate / run-fuzzer (an... 1d 0h
- ✓ evaluate / run-fuzzer (bc... 1d 0h
- ✓ evaluate / run-fuzzer (cl... 1d 0h
- ✓ evaluate / run-fuzzer (m... 1d 0h
- ✓ evaluate / run-fuzzer (rh... 1d 0h
- ✓ evaluate / run-fuzzer (an... 1d 0h
- ✓ evaluate / run-fuzzer (bc... 1d 0h
- ✓ evaluate / run-fuzzer (cl... 1d 0h

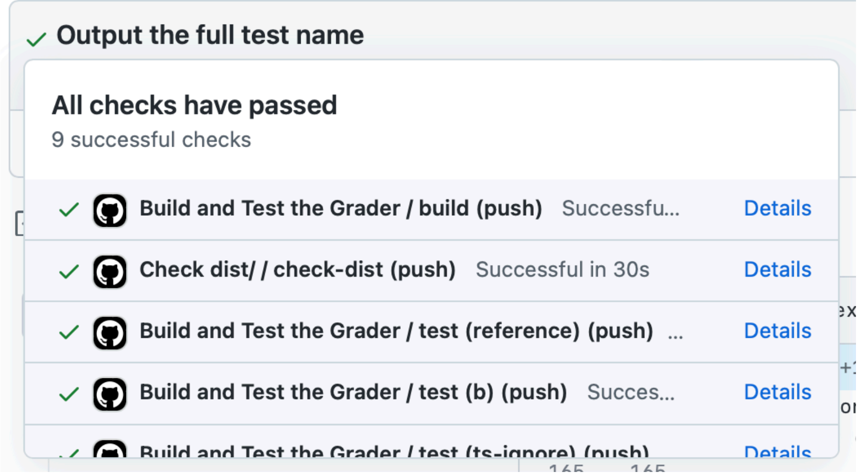
- ✓ evaluate / repro-jacoco 13m 52s
- ✓ evalua

On Demand: Run 24 hour performance test on 5 benchmarks, repeating each test 20 times (100 concurrent jobs)

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)



✓ Output the full test name

All checks have passed

9 successful checks

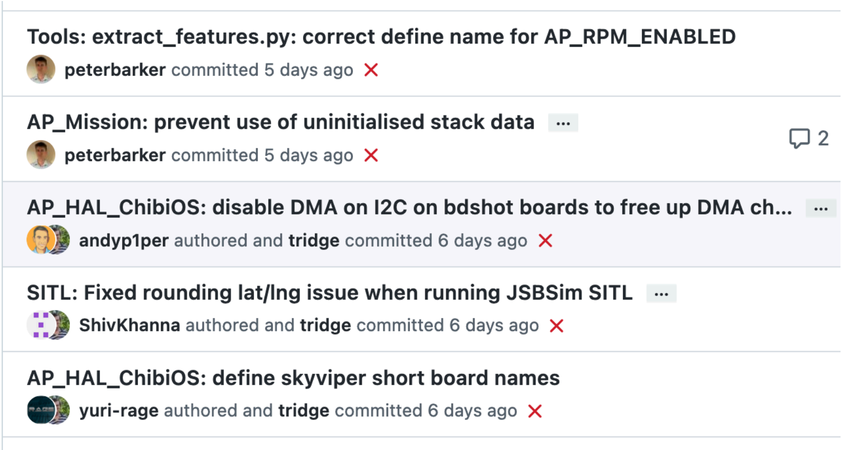
✓ Build and Test the Grader / build (push) Successfu... Details

✓ Check dist/ / check-dist (push) Successful in 30s Details

✓ Build and Test the Grader / test (reference) (push) ... Details

✓ Build and Test the Grader / test (b) (push) Succes... Details

✓ Build and Test the Grader / test (ts-ignore) (push) Details



Tools: extract_features.py: correct define name for AP_RPM_ENABLED

peterbarker committed 5 days ago ✗

AP_Mission: prevent use of uninitialised stack data ... 2

peterbarker committed 5 days ago ✗

AP_HAL_ChibiOS: disable DMA on I2C on bdshot boards to free up DMA ch... ...

andyp1per authored and tridge committed 6 days ago ✗

SITL: Fixed rounding lat/lng issue when running JSBSim SITL ...

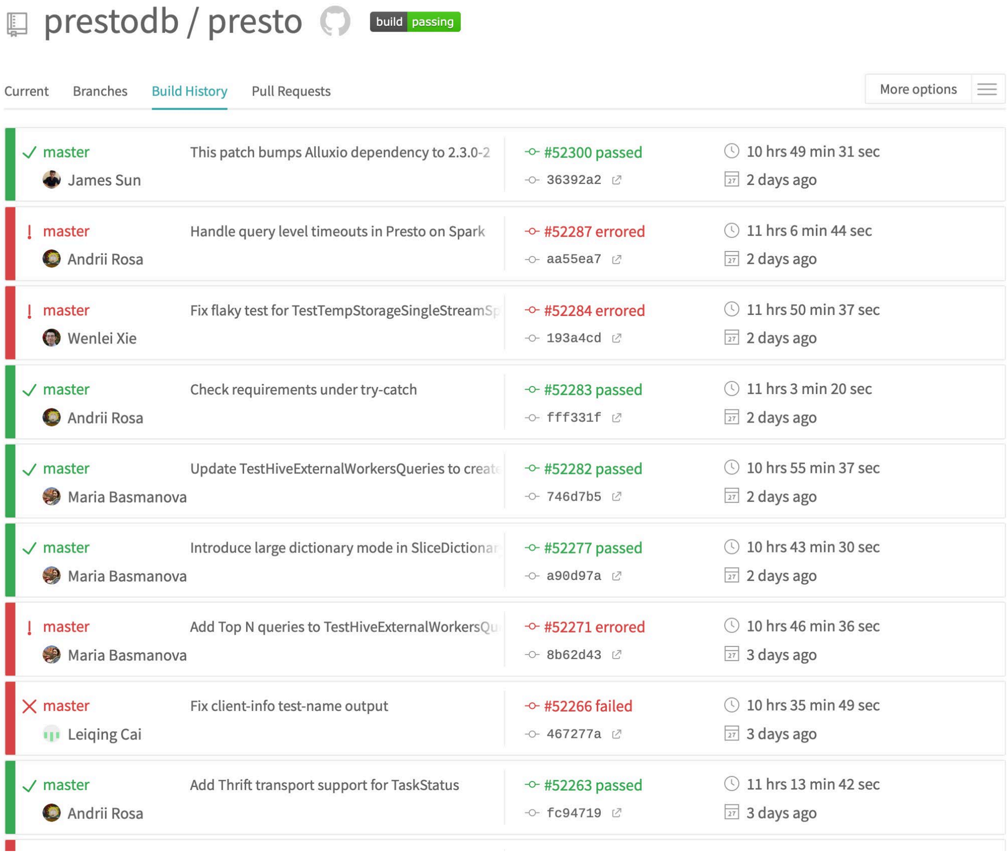
ShivKhanna authored and tridge committed 6 days ago ✗

AP_HAL_ChibiOS: define skyviper short board names

yuri-rage authored and tridge committed 6 days ago ✗

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

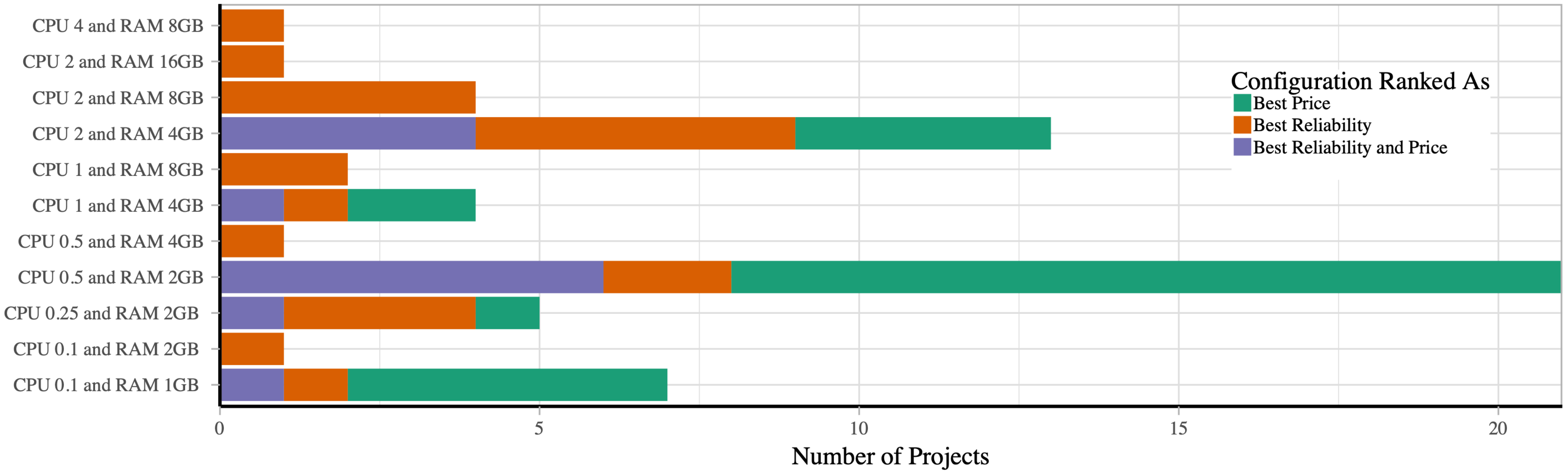


prestodb / presto build passing

	Branch / Author	Commit message	Build / Commit	Duration / Finished
✓	master James Sun	This patch bumps Alluxio dependency to 2.3.0-2	#52300 passed 36392a2	10 hrs 49 min 31 sec 2 days ago
!	master Andrii Rosa	Handle query level timeouts in Presto on Spark	#52287 errored aa55ea7	11 hrs 6 min 44 sec 2 days ago
!	master Wenlei Xie	Fix flaky test for TestTempStorageSingleStreamSp	#52284 errored 193a4cd	11 hrs 50 min 37 sec 2 days ago
✓	master Andrii Rosa	Check requirements under try-catch	#52283 passed fff331f	11 hrs 3 min 20 sec 2 days ago
✓	master Maria Basmanova	Update TestHiveExternalWorkersQueries to creat	#52282 passed 746d7b5	10 hrs 55 min 37 sec 2 days ago
✓	master Maria Basmanova	Introduce large dictionary mode in SliceDictionar	#52277 passed a90d97a	10 hrs 43 min 30 sec 2 days ago
!	master Maria Basmanova	Add Top N queries to TestHiveExternalWorkersQu	#52271 errored 8b62d43	10 hrs 46 min 36 sec 3 days ago
✗	master Leiqing Cai	Fix client-info test-name output	#52266 failed 467277a	10 hrs 35 min 49 sec 3 days ago
✓	master Andrii Rosa	Add Thrift transport support for TaskStatus	#52263 passed fc94719	11 hrs 13 min 42 sec 3 days ago

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

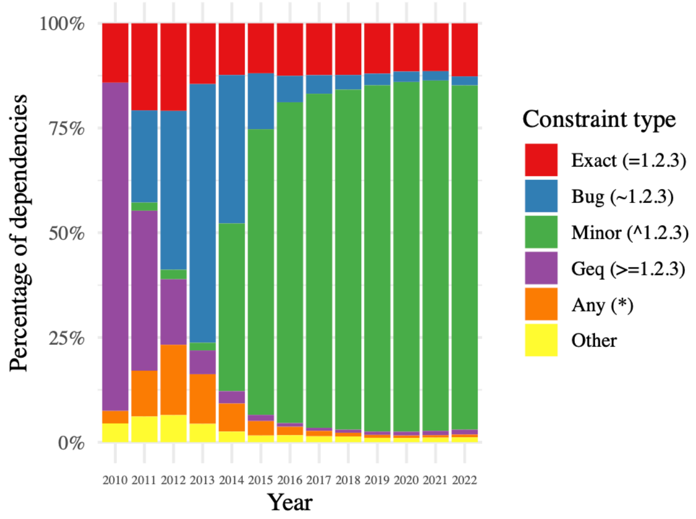
- “Orchestrate” -> Execute in the right order, ideally with concurrency, example tasks:
  - Installing dependencies
  - Compiling the code
  - Running static analysis
  - Generating documentation
  - Running tests
  - Creating artifacts for customers
  - Deploying Code
- Example build systems: xMake,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

Dependency Managers Organize External Dependencies

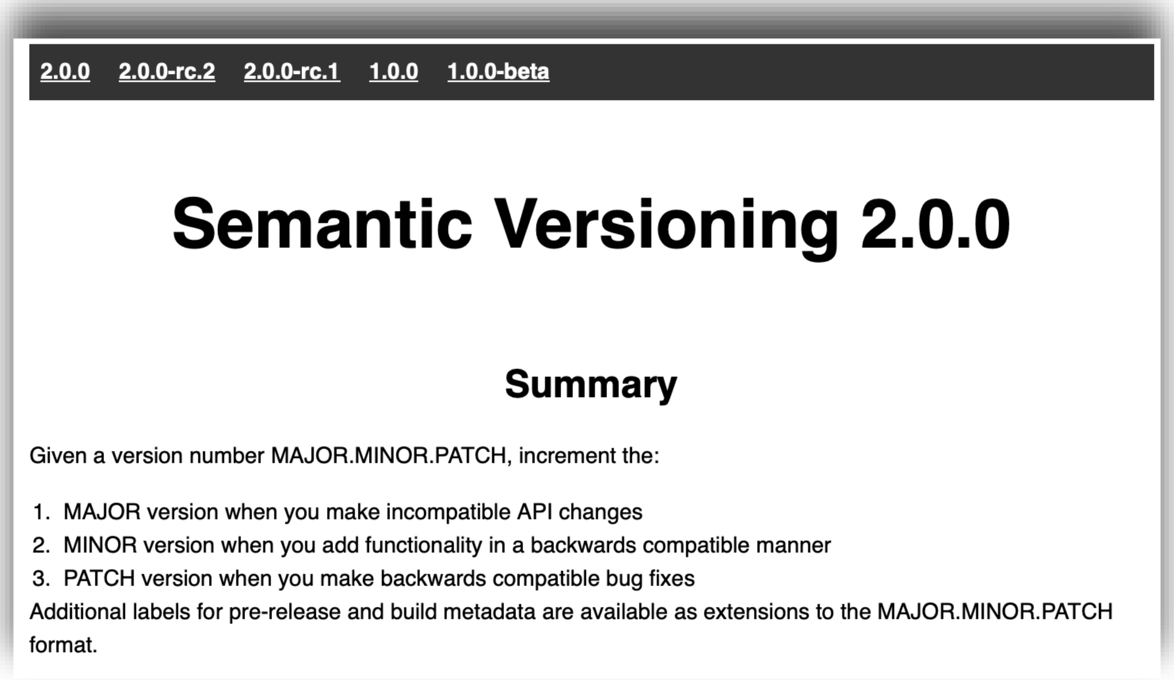
- Addresses this problem: “Before you compile this code, install commons-lang from the Apache website”
- Declare a dependency using coordinates (unique ID of a package plus version)
- Packages are archived in common repositories; fetched/linked by dependency manager
- Dependency managers handle transitive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



Semantic Versioning 2.0.0

Summary

Given a version number MAJOR.MINOR.PATCH, increment the:

1. MAJOR version when you make incompatible API changes
2. MINOR version when you add functionality in a backwards compatible manner
3. PATCH version when you make backwards compatible bug fixes

Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

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”

Continuous Delivery

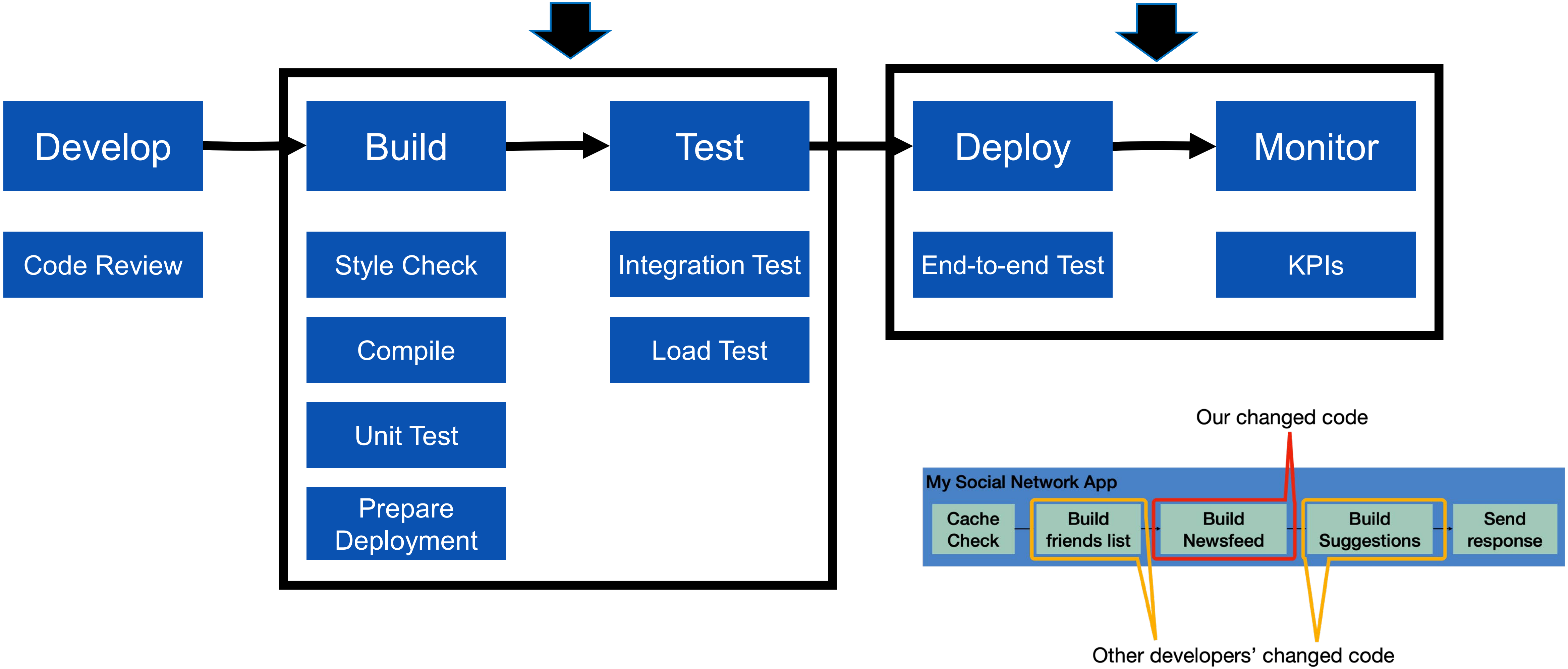
- “Faster is safer”: Key values of continuous delivery
  - Release frequently, in small batches
  - Maintain key performance indicators to evaluate the impact of updates
  - Phase roll-outs
  - Evaluate business impact of new features

Continuous Delivery is about deciding which new features to deliver, and when

- You have a large system with many engineers working on new features (and bug fixes ☺)
- When a new feature or fix is ready, how do you roll it out to your users?

A continuous-delivery process is also a software pipeline

Automate this centrally, provide a central record of results



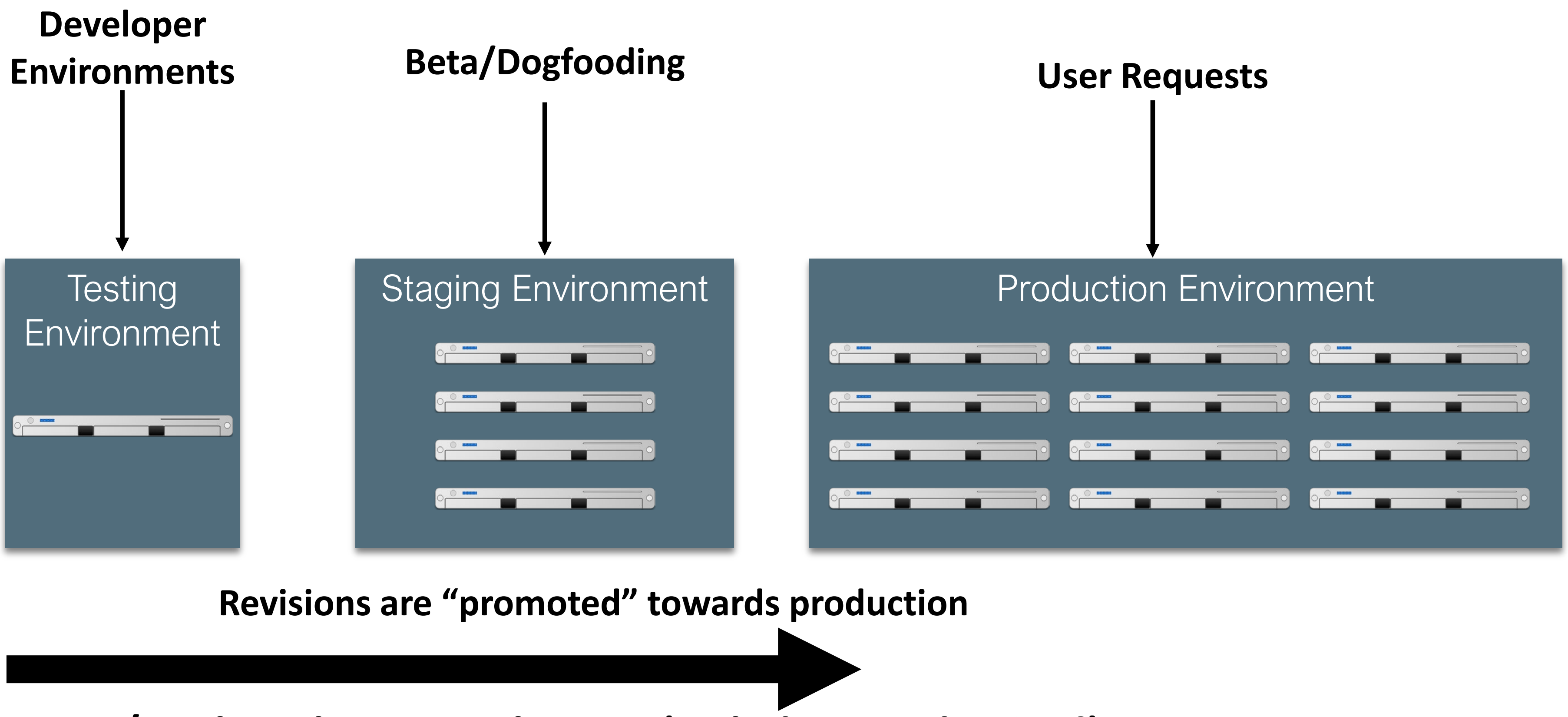
Continuous Delivery does not mean Immediate Delivery

- Even if you are deploying every day (“continuously”), you still have some latency
- A new feature I develop today won't be released today
- But, a new feature I develop today can begin the **release pipeline** today (minimizes risk)
- **Release Engineer**: gatekeeper who decides when something is ready to go out, oversees the actual deployment process

Ways to mitigate deployment risks

- Use a realistic staging environment
- Use post-deployment monitoring
- Use split deployments
- Use tools to automate deployment tasks

Build a staging environment to qualify features for delivery



Developer Environments

Beta/Dogfooding

User Requests

Testing Environment

Staging Environment

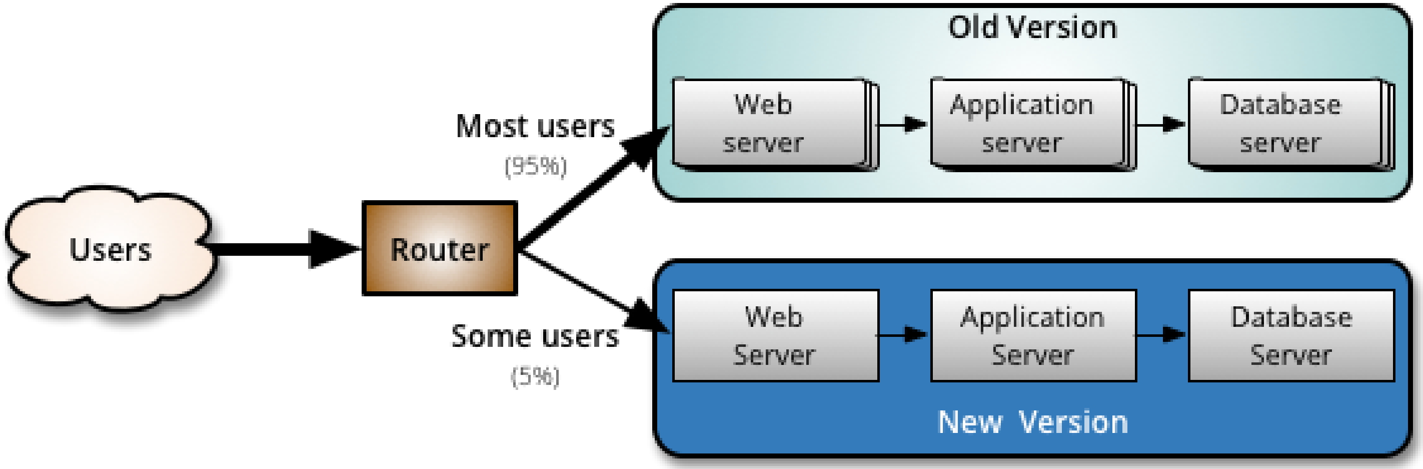
Production Environment

Revisions are “promoted” towards production

Q/A takes place in each stage (including production!)

Split Deployments Mitigate Risk

- Lower risk if a problem occurs in staging than in production
- Or deploy to a small set of users before deploying more widely
- Names:
  - “Eat your own dogfood”
  - Beta/Alpha testers
  - A/B testing
  - "canaries"

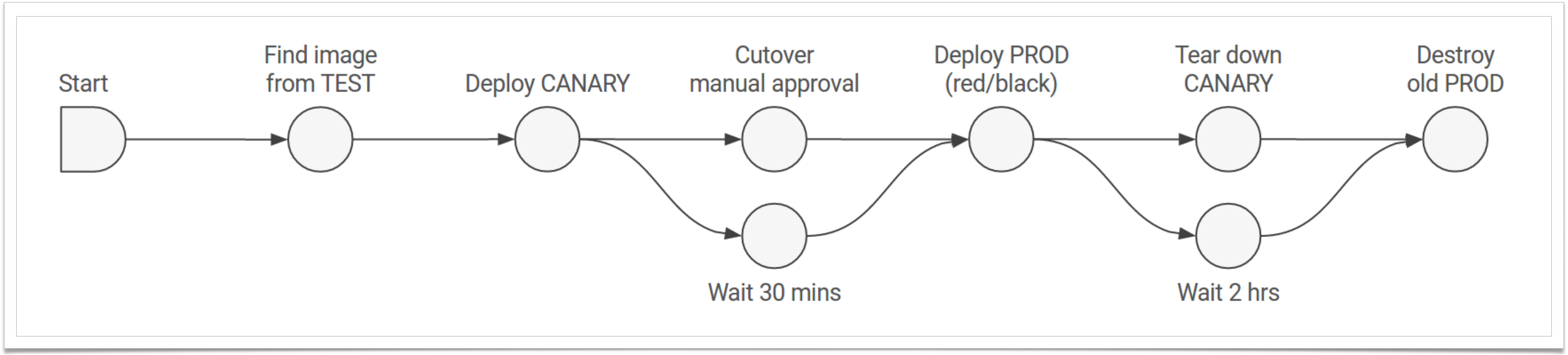


Post-delivery monitoring mitigates risk

- Consider both direct (e.g. business) metrics, and indirect (e.g. system) metrics
- Hardware
- Voltages, temperatures, fan speeds, component health
- OS
- Memory usage, swap usage, disk space, CPU load
- Middleware
- Memory, thread/db connection pools, connections, response time
- Applications
- Business transactions, conversion rate, status of 3rd party components

Continuous Delivery Tools

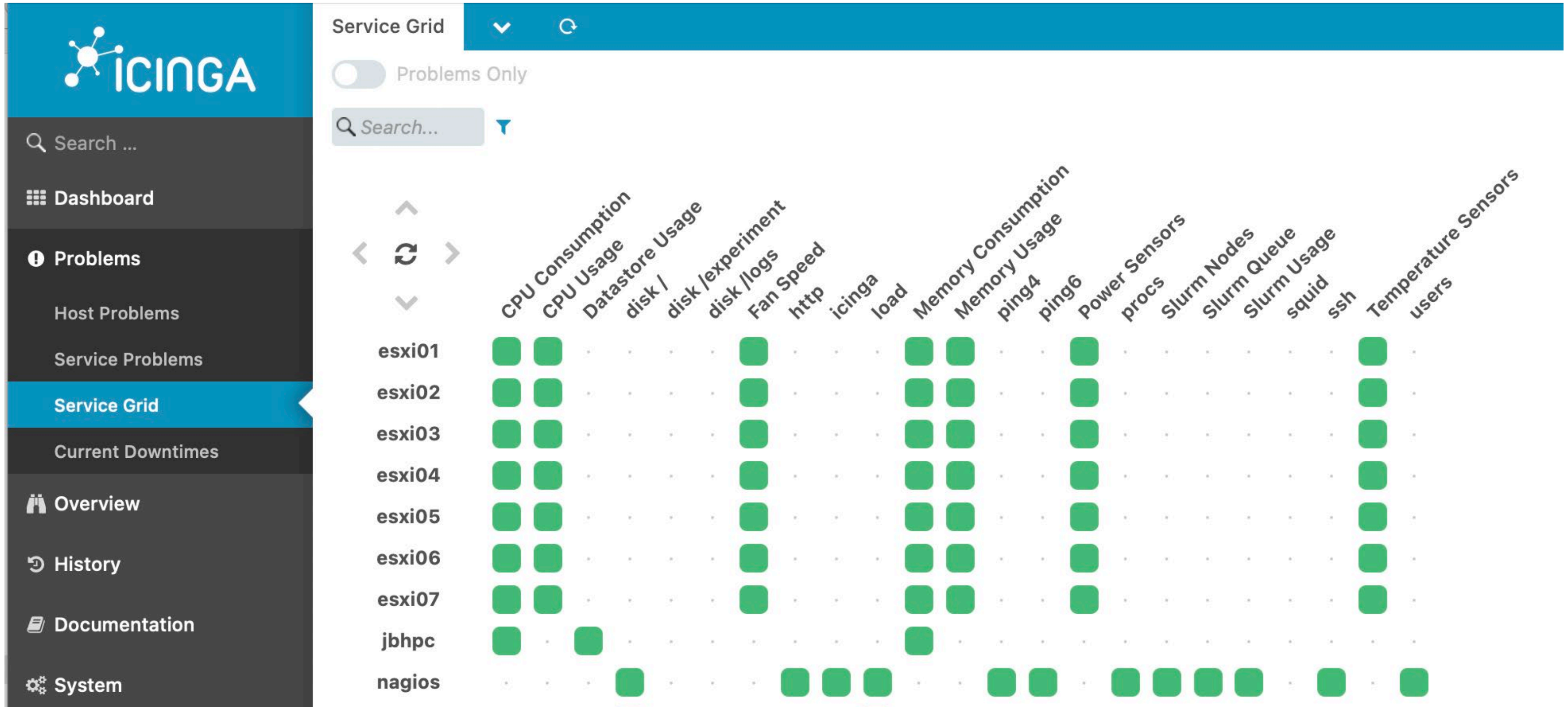
- Simplest tools deploy from a branch to a service (e.g. Render.com, Heroku)
- More complex tools:
  - Auto-deploys from version control to a staging environment + promotes through release pipeline
  - Monitors key performance indicators to automatically take corrective actions
  - Example: "Spinnaker" (Open-Sourced by Netflix, c 2015)



Example CD pipeline from Spinnaker's documentation: https://spinnaker.io/docs/concepts/#application-deployment

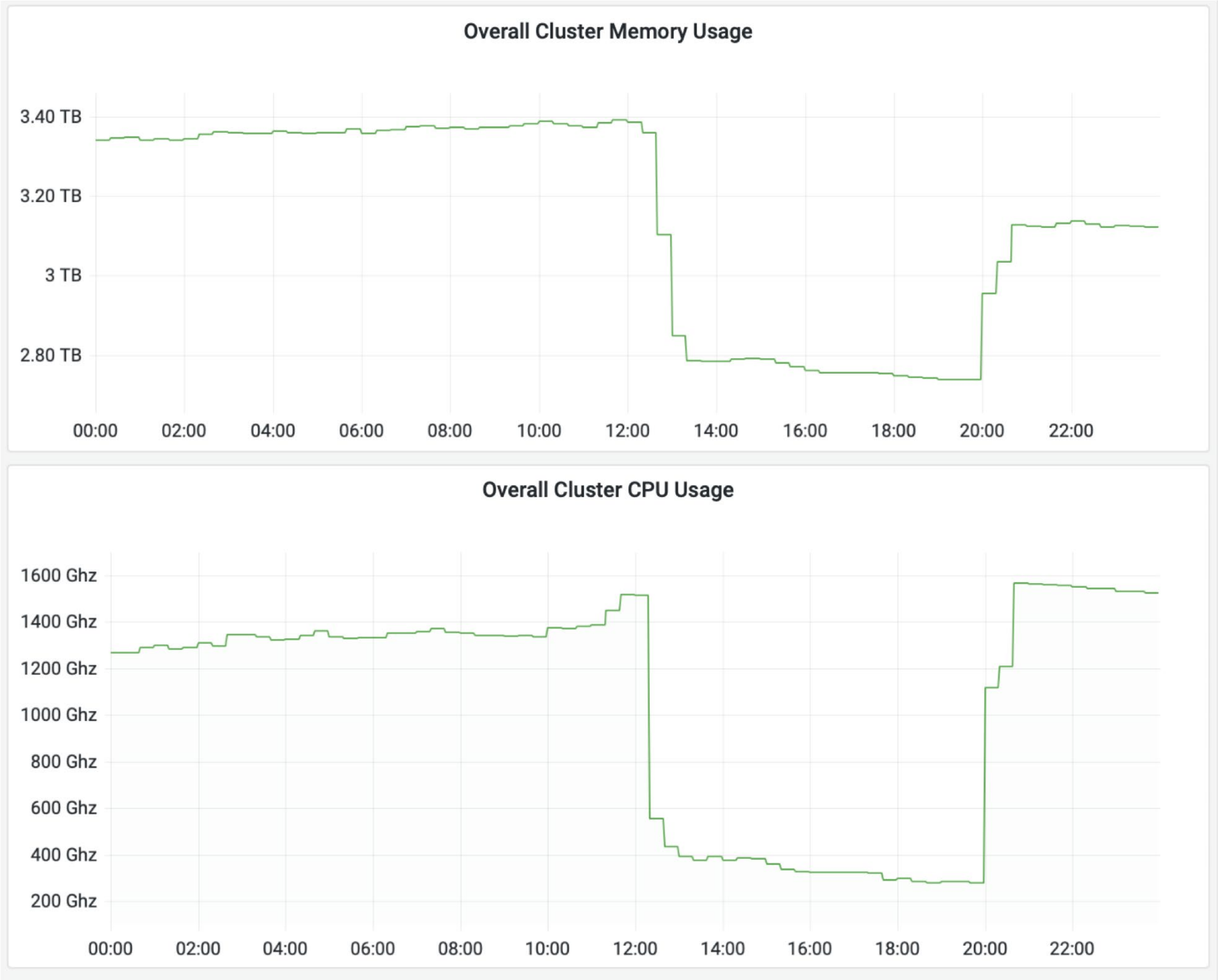
Tools for Monitoring Deployments

- Nagios (c 2002): Agent-based architecture (install agent on each monitored host), extensible plugins for executing “checks” on hosts
- Track system-level metrics, app-level metrics, user-level KPIs



	CPU Consumption	CPU Usage	Datastore Usage	disk /	disk /experiment	disk /logs	Fan Speed	http	icinga	load	Memory Consumption	Memory Usage	ping4	ping6	Power Sensors	procs	Slurm Nodes	Slurm Queue	Slurm Usage	squid	ssh	Temperature Sensors	users
esxi01	🟩	🟩	·	·	·	·	🟩	·	·	·	🟩	🟩	·	·	🟩	·	·	·	·	·	·	🟩	·
esxi02	🟩	🟩	·	·	·	·	🟩	·	·	·	🟩	🟩	·	·	🟩	·	·	·	·	·	·	🟩	·
esxi03	🟩	🟩	·	·	·	·	🟩	·	·	·	🟩	🟩	·	·	🟩	·	·	·	·	·	·	🟩	·
esxi04	🟩	🟩	·	·	·	·	🟩	·	·	·	🟩	🟩	·	·	🟩	·	·	·	·	·	·	🟩	·
esxi05	🟩	🟩	·	·	·	·	🟩	·	·	·	🟩	🟩	·	·	🟩	·	·	·	·	·	·	🟩	·
esxi06	🟩	🟩	·	·	·	·	🟩	·	·	·	🟩	🟩	·	·	🟩	·	·	·	·	·	·	🟩	·
esxi07	🟩	🟩	·	·	·	·	🟩	·	·	·	🟩	🟩	·	·	🟩	·	·	·	·	·	·	🟩	·
jbhpc	🟩	·	🟩	·	·	·	·	·	·	·	🟩	·	·	·	·	·	·	·	·	·	·	·	·
nagios	·	·	·	🟩	·	·	·	🟩	🟩	🟩	·	·	🟩	🟩	·	🟩	🟩	🟩	🟩	·	🟩	·	🟩

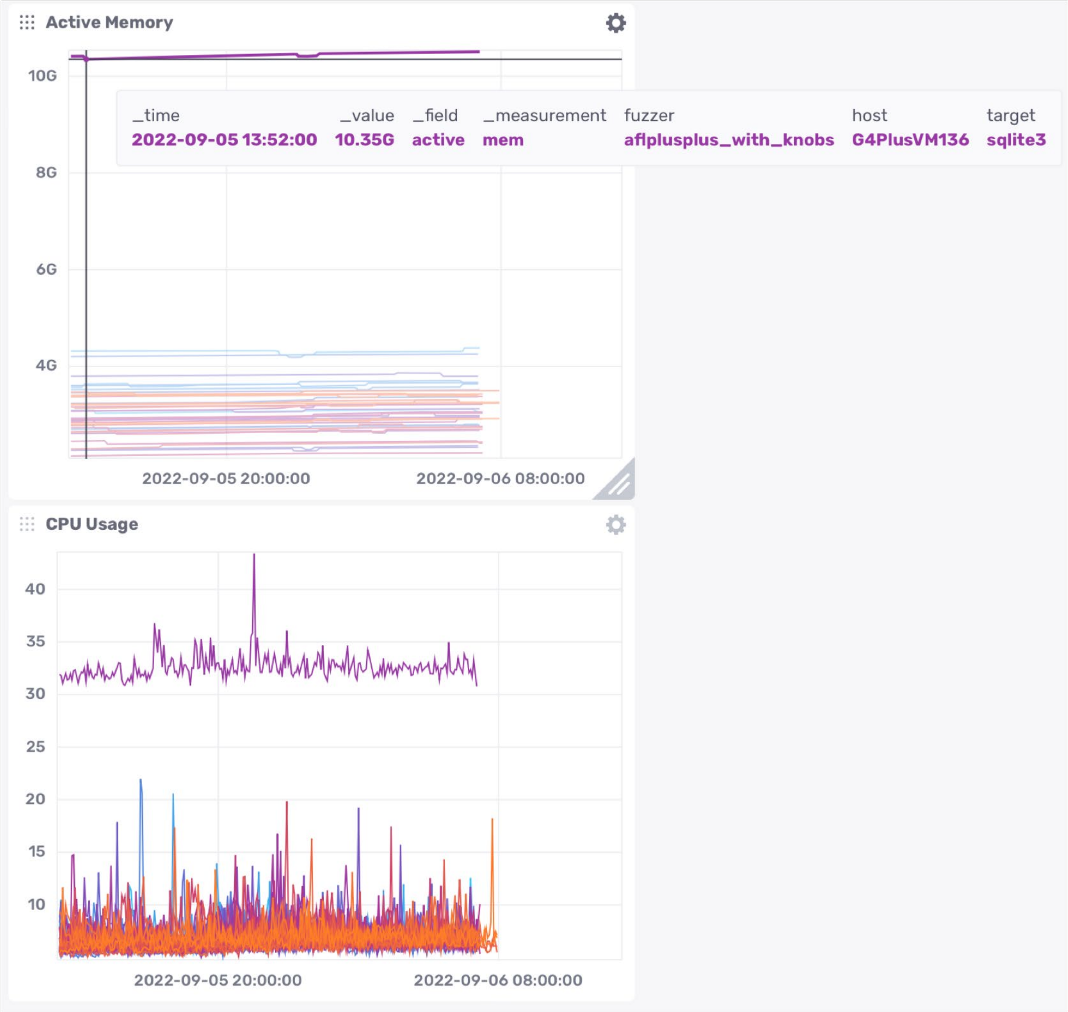
Monitoring can help identify operational issues



Overall Cluster Memory Usage

Overall Cluster CPU Usage

Grafana (AGPL, c 2014)



Active Memory

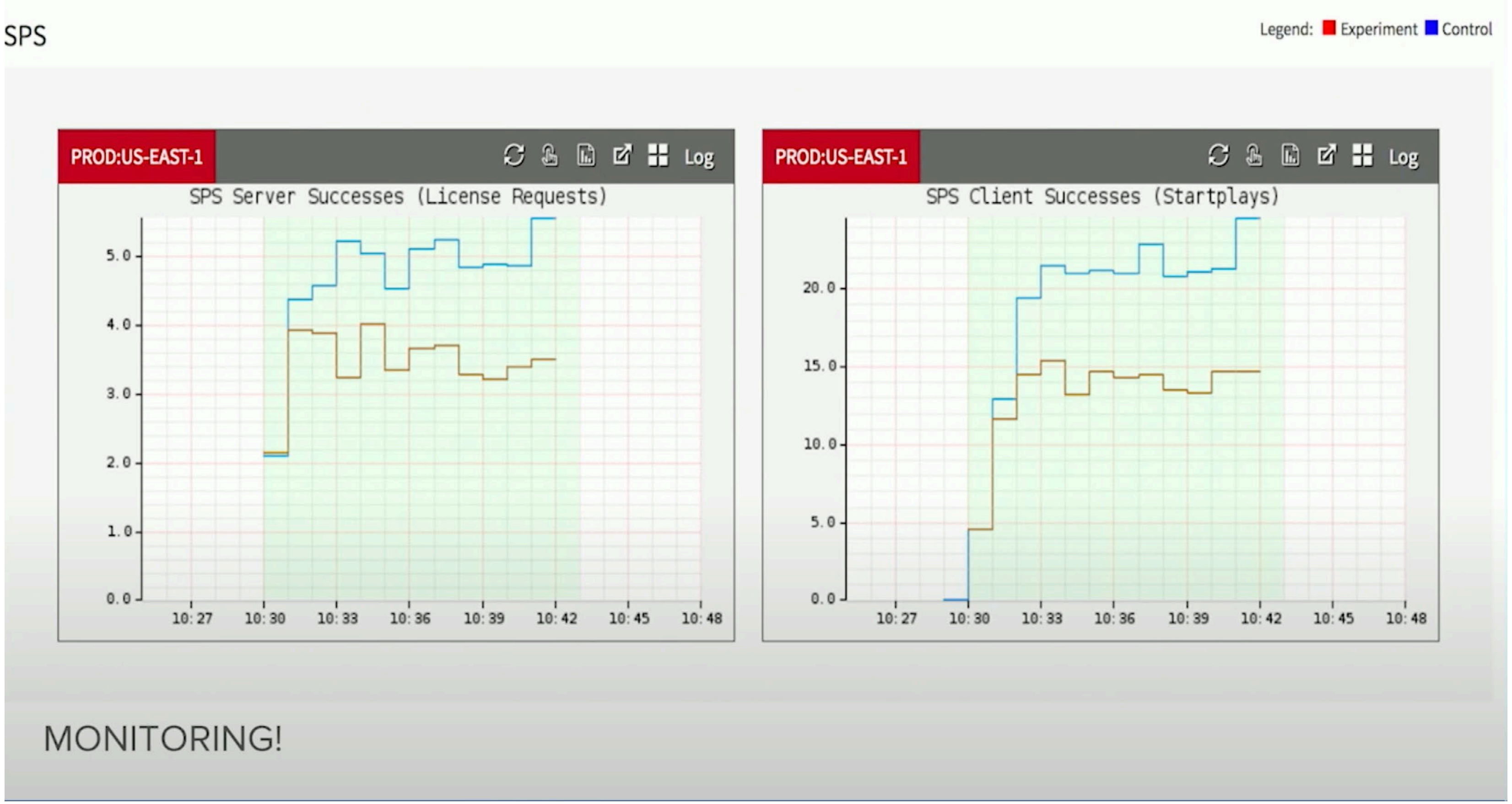
_time	_value	_field	_measurement	fuzzer	host	target
2022-09-05 13:52:00	10.35G	active	mem	aflplusplus_with_knobs	G4PlusVM136	sqlite3

CPU Usage

InfluxDB (MIT license, c 2013)

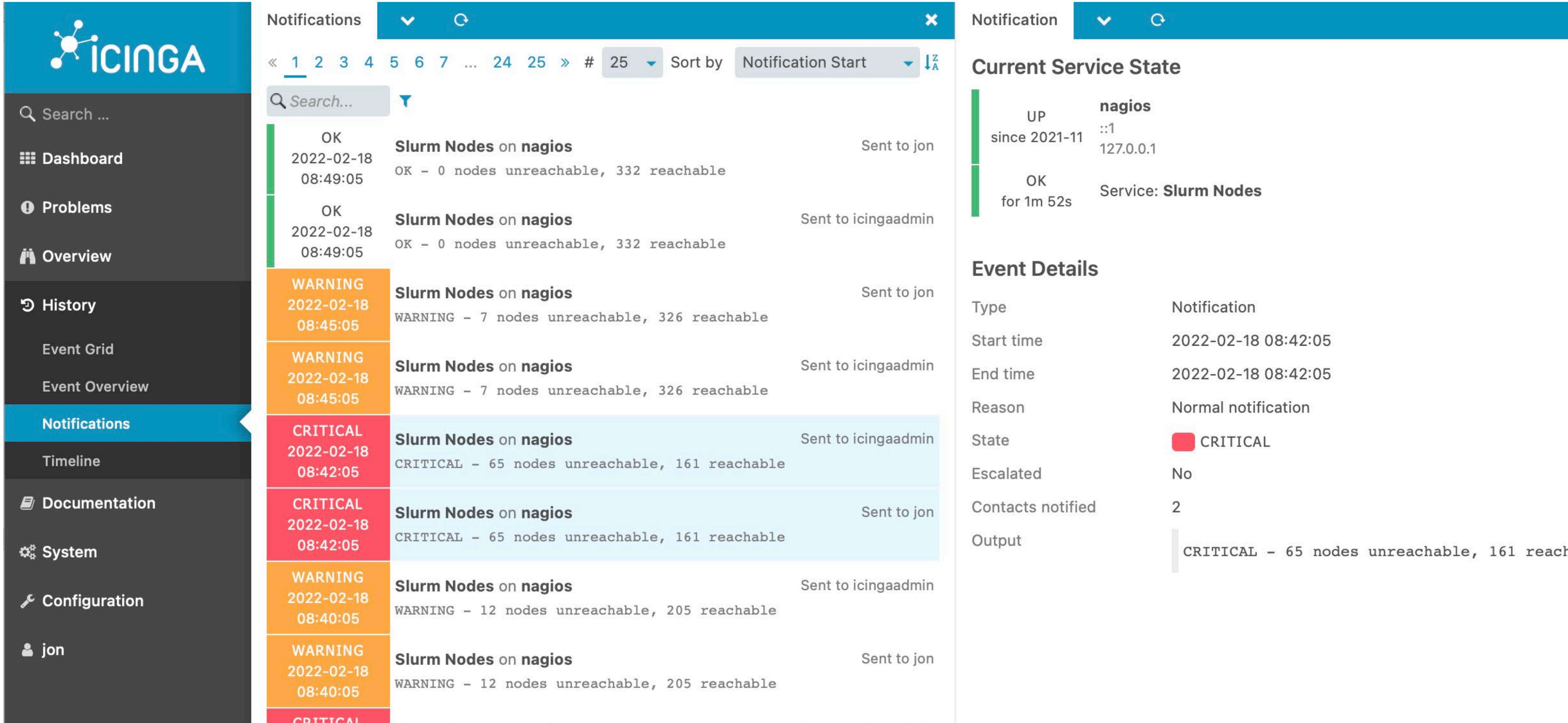
Continuous Delivery Tools Can Take Automated Actions

- Example: Automated roll-back of updates at Netflix based on "streams-per-second" (SPS)



https://www.youtube.com/watch?v=qyzymLlj9ag

Monitoring Services Can Take Automated Actions



OK 2022-02-18 08:49:05 — **Slurm Nodes** on **nagios** — Sent to jon
`OK - 0 nodes unreachable, 332 reachable`

OK 2022-02-18 08:49:05 — **Slurm Nodes** on **nagios** — Sent to icingaadmin
`OK - 0 nodes unreachable, 332 reachable`

WARNING 2022-02-18 08:45:05 — **Slurm Nodes** on **nagios** — Sent to jon
`WARNING - 7 nodes unreachable, 326 reachable`

WARNING 2022-02-18 08:45:05 — **Slurm Nodes** on **nagios** — Sent to icingaadmin
`WARNING - 7 nodes unreachable, 326 reachable`

CRITICAL 2022-02-18 08:42:05 — **Slurm Nodes** on **nagios** — Sent to icingaadmin
`CRITICAL - 65 nodes unreachable, 161 reachable`

CRITICAL 2022-02-18 08:42:05 — **Slurm Nodes** on **nagios** — Sent to jon
`CRITICAL - 65 nodes unreachable, 161 reachable`

WARNING 2022-02-18 08:40:05 — **Slurm Nodes** on **nagios** — Sent to icingaadmin
`WARNING - 12 nodes unreachable, 205 reachable`

WARNING 2022-02-18 08:40:05 — **Slurm Nodes** on **nagios** — Sent to jon
`WARNING - 12 nodes unreachable, 205 reachable`

CRITICAL

Current Service State

UP since 2021-11 — **nagios** ::1 127.0.0.1

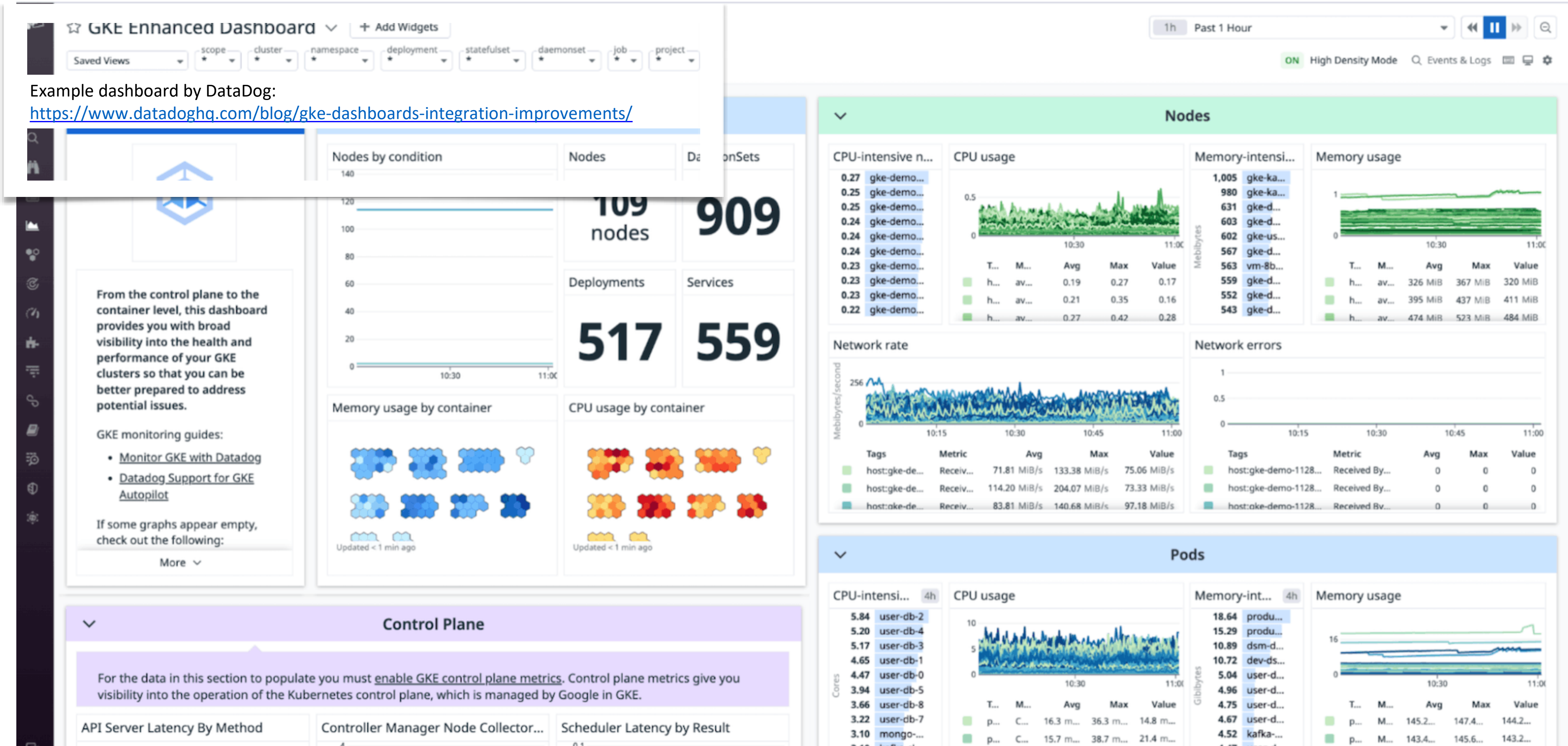
OK for 1m 52s — Service: **Slurm Nodes**

Event Details

Type	Notification
Start time	2022-02-18 08:42:05
End time	2022-02-18 08:42:05
Reason	Normal notification
State	🟥 CRITICAL
Escalated	No
Contacts notified	2
Output	`CRITICAL - 65 nodes unreachable, 161 reach`

From Monitoring to Observability

- Understanding what is going on inside of our deployed systems by visualizing internal metrics



Example dashboard by DataDog: https://www.datadoghq.com/blog/gke-dashboards-integration-improvements/

From the control plane to the container level, this dashboard provides you with broad visibility into the health and performance of your GKE clusters so that you can be better prepared to address potential issues.

GKE monitoring guides:

- Monitor GKE with Datadog
- Datadog Support for GKE Autopilot

If some graphs appear empty, check out the following:

For the data in this section to populate you must enable GKE control plane metrics. Control plane metrics give you visibility into the operation of the Kubernetes control plane, which is managed by Google in GKE.

Beware of Metrics

- McNamara Fallacy
  - Measure whatever can be easily measured
  - Disregard that which cannot be measured easily
  - Presume that which cannot be measured easily is not important
  - Presume that which cannot be measured easily does not exist

How should we allocate our testing resources?

- How much unit testing should be required?
- When should we do code reviews?
- How often should we do integration tests?
- Different organizations may make different choices

Two extremes: Continuous Delivery vs. TDD

- Test driven development
  - Write and maintain tests per-feature (manual! hard!)
  - Unit tests help locate bugs (at unit level)
  - Integration/system tests also needed to locate interaction-related faults
- Continuous delivery
  - Write and maintain high-level observability metrics
  - Deploy features one-at-a-time, look for canaries in metrics
  - Write fewer integration/system tests

CI at scale: Google Test Automation Platform (TAP (2020))

- Massive continuous build of entire Google codebase
  - in a dedicated data center
  - 50,000 unique changes per-day, 4 billion test cases per-day
- Engineers submit unit tests along with their changes
  - Block merge if they fail
- If they pass, change is put in the codebase.
  - visible to entire company!
  - average wait time to this point: 11 minutes
- Then (asynchronously) run all affected integration tests
  - If any fail, change is sent back to a human on the submitter's team (the "build cop") who must act immediately to roll-back or fix.

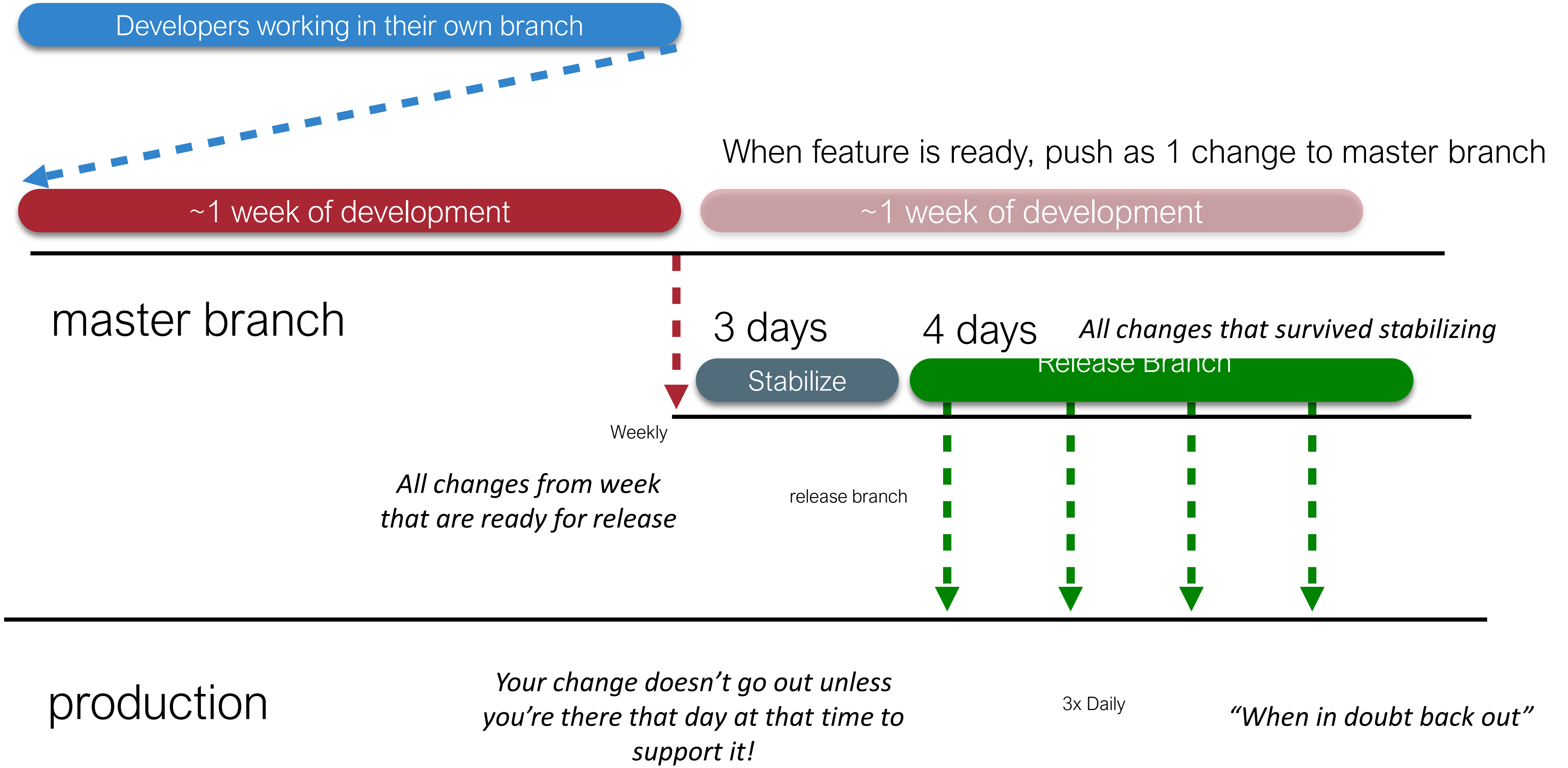
"Software Engineering at Google: Lessons Learned from Programming Over Time," Wright, Winters and Manshreck, 2020 (O'Reilly), pp. 494-497

Facebook: "Move fast and break things"

- de-prioritize unit tests
- Emphasis on getting features to users quickly
- Strategy: push many small changes to fractions of the user base. ("split deployments")

Deployment Example: Facebook.com

- Pre-2016



Developers working in their own branch

When feature is ready, push as 1 change to master branch

~1 week of development

~1 week of development

master branch

3 days

4 days

All changes that survived stabilizing

Stabilize

Release Branch

Weekly

All changes from week that are ready for release

release branch

production

Your change doesn't go out unless you're there that day at that time to support it!

3x Daily

"When in doubt back out"

Facebook used to have an elaborate system of branches

- dev branches got merged into master,
- then once a week all changes from the past week were pulled into a release branch (often 10,000 changes per week)
- For 3 days they “stabilized” the release branch – find changes that are causing very bad behavior and back them out. (manual process!!)
- Then for the last 4 days of the week, every change that survived that stabilization got *individually pushed* to production batched so that this happens 3x/day.
- Important to do small deploys so that you could isolate bad changes.

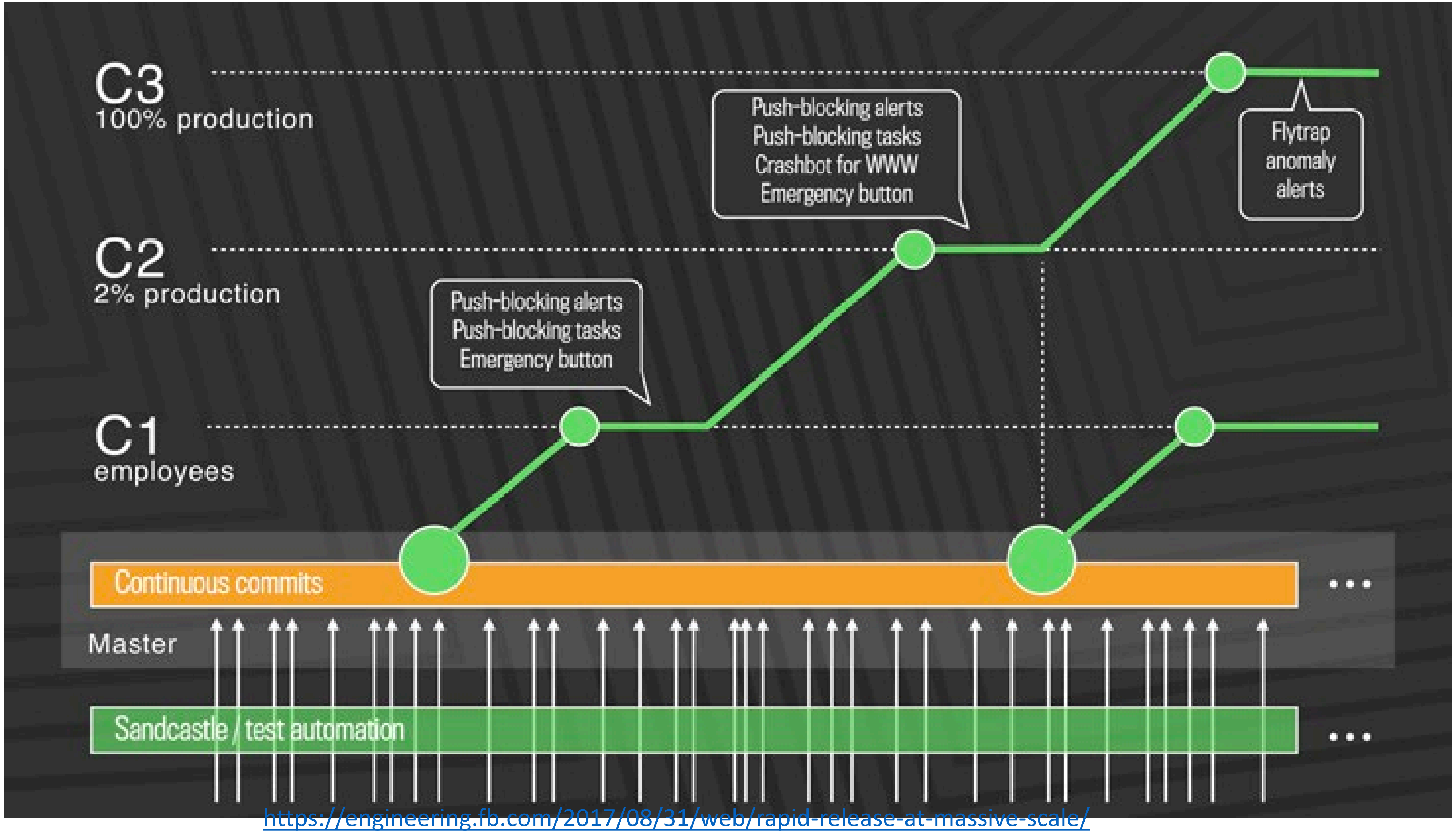
Deployment Example

"Our main goal was to make sure that the new system made people's experience better — or at least, didn't make it worse. After a year of planning and development, over the course of three days **we enabled 100% of our production web servers to run code deployed directly from master**"

- Chuck Rossi, Director Software Infrastructure & Release Engineering @ Facebook

"Rapid release at massive scale" https://engineering.fb.com/2017/08/31/web/rapid-release-at-massive-scale/

Post-2016: truly continuous releases from master branch



https://engineering.fb.com/2017/08/31/web/rapid-release-at-massive-scale/

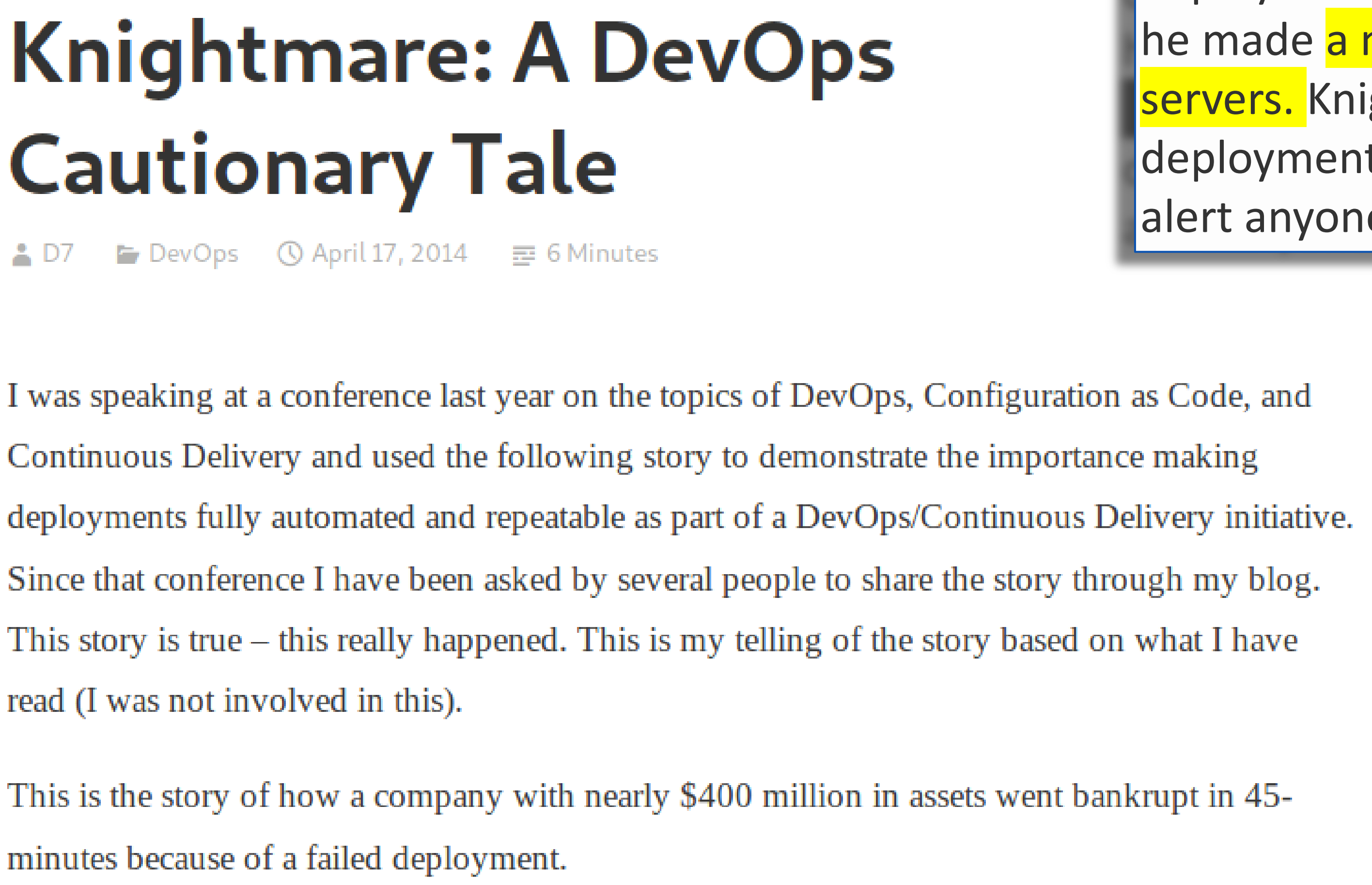
Post-2016: Truly Continuous Releases from Master Branch (excerpts from blog post)

1. First, diffs that have passed a series of automated internal tests and land in master are pushed out to Facebook employees.
2. In this stage, get push-blocking alerts if we've introduced a regression, and an emergency stop button lets us keep the release from going any further.
3. If everything is OK, push the changes to 2 percent of production, where again we collect signal and monitor alerts, especially for edge cases that our testing or employee dogfooding may not have picked up.
4. Finally, roll out to 100 percent of production, where our Flytrap tool aggregates user reports and alerts us to any anomalies.
5. Many of the changes are initially kept behind feature flags, which allows to roll out mobile and web code releases independently from new features, helping to lower the risk of any particular update causing a problem.
6. If we do find a problem, simply switch the feature off rather than revert back to a previous version or fix forward.

https://engineering.fb.com/2017/08/31/web/rapid-release-at-massive-scale/

What not to do: Failed Deployment at Knight Capital

"In the week before go-live, a Knight engineer manually deployed the new RLP code in SMARS to its 8 servers. However, he made a mistake and did not copy the new code to one of the servers. Knight did not have a second engineer review the deployment, and neither was there an automated system to alert anyone to the discrepancy. "



Knightmare: A DevOps Cautionary Tale

D7 DevOps April 17, 2014 6 Minutes

I was speaking at a conference last year on the topics of DevOps, Configuration as Code, and Continuous Delivery and used the following story to demonstrate the importance making deployments fully automated and repeatable as part of a DevOps/Continuous Delivery initiative. Since that conference I have been asked by several people to share the story through my blog. This story is true – this really happened. This is my telling of the story based on what I have read (I was not involved in this).

This is the story of how a company with nearly $400 million in assets went bankrupt in 45-minutes because of a failed deployment.

https://www.henricodolfing.com/2019/06/project-failure-case-study-knight-capital.html

What could Knight capital have done better?

- Use capture/replay testing instead of driving market conditions in a test
- Avoid including “test” code in production deployments
- Automate deployments
- Define and monitor risk-based KPIs
- Create checklists for responding to incidents

Review

- By now, you should be able to…
  - Describe how continuous integration helps to catch errors sooner in the software lifecycle
  - Describe strategies for performing quality-assurance on software as and after it is delivered
  - Compare and contrast continuous delivery with test driven development as a quality assurance strategy