

# 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 development 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

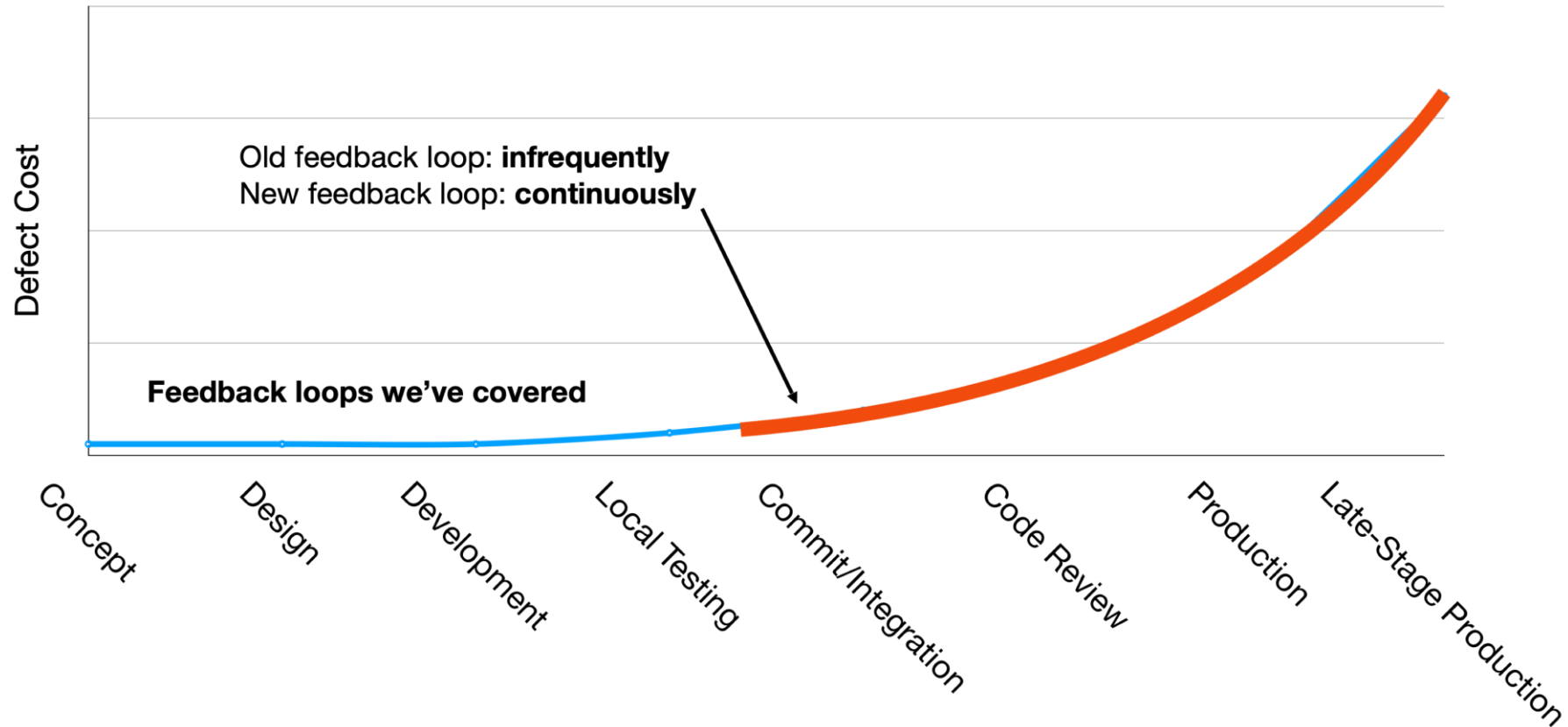
# Review: The Agile Model Reduces Risk by Embracing Change (~2000)

---

- 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 values fast quality feedback loops

- Faster feedback = lower cost to fix bugs



# Agile requires a quality assurance process

---

- Multiple processes have to work together to ensure quality:
  - unit testing/TDD
  - mix of unit tests & integration tests (we'll see more of this)
  - code review
  - continuous integration (also: watch for canaries)
  - continuous deployment (A/B, canaries, etc.)
  - quality includes non-functional requirements (resource consumption, response time) or generally speaking extensibility, maintainability, etc.
- Quality is everyone's responsibility

# Example: Some bugs slip through testing, even in highly-regulated industries

## Aviation

### After Alaska Airlines planes bump runway while taking off from Seattle, a scramble to ‘pull the plug’

By Dominic Gates, The Seattle Times

Updated: February 20, 2023

Published: February 20, 2023

“That morning, a software bug in an update to the DynamicSource tool caused it to provide seriously undervalued weights for the airplanes.

The Alaska 737 captain said the data was on the order of 20,000 to 30,000 pounds light. With the total weight of those jets at 150,000 to 170,000 pounds, the error was enough to skew the engine thrust and speed settings.

Both planes headed down the runway with less power and at lower speed than they should have. And with the jets judged lighter than they actually were, the pilots rotated too early

Both the Max 9 and 737-900ER have long passenger cabins, which makes them more vulnerable to a tail strike when the nose comes up too soon.” ...

... “A quick interim fix proved easy: When operations staff turned off the automatic uplink of the data to the aircraft and switched to manual requests “we didn’t have the bug anymore.”

Peyton said his team also checked the integrity of the calculation itself before lifting the stoppage. All that was accomplished in 20 minutes.

The software code was permanently repaired about five hours later.

Peyton added that even though the update to the DynamicSource software had been tested over an extended period, the bug was missed because it only presented when many aircraft at the same time were using the system.

Subsequently, a test of the software under high demand was developed.”



Photo: saitiers\_photography (IG, different plane/airport)

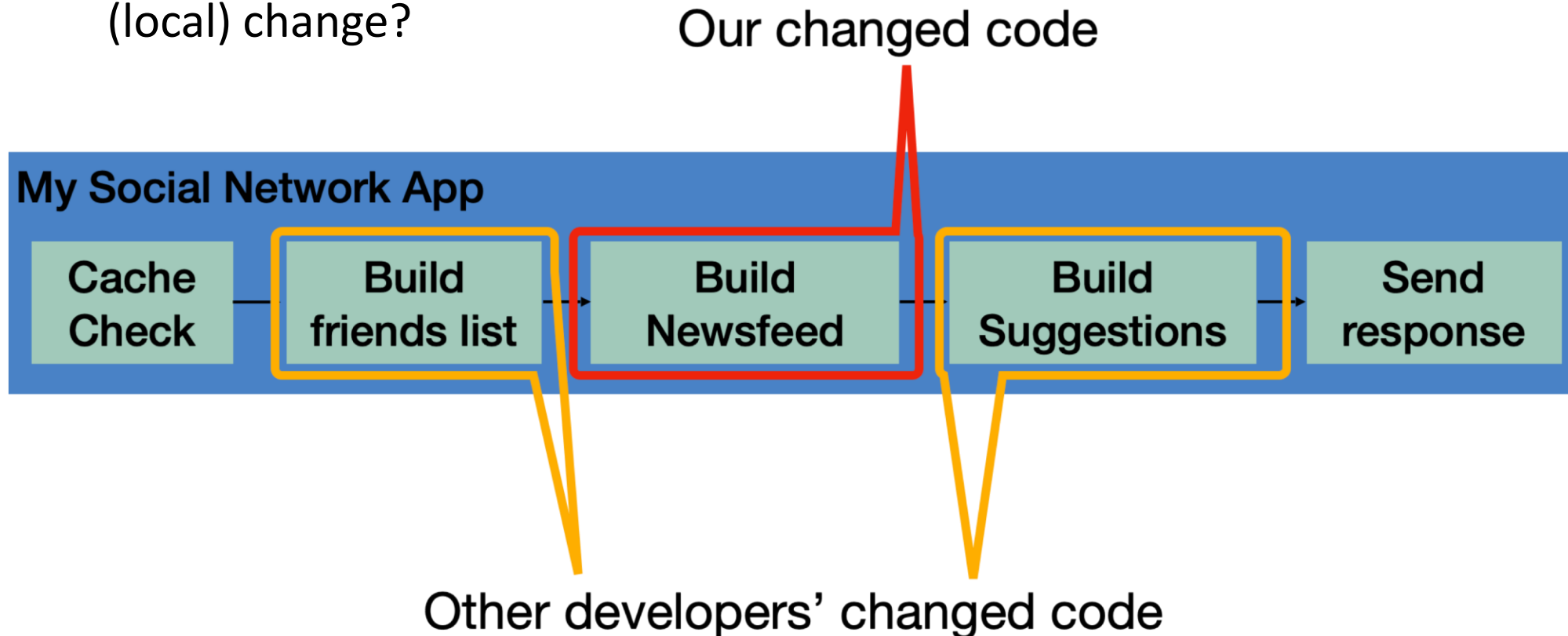
# Continuous development practices improve code quality and dev velocity

---

- Continuous integration: Perform frequent integrations with entire codebase, running integration-scale tests
- Continuous delivery: Deploy frequently and monitor

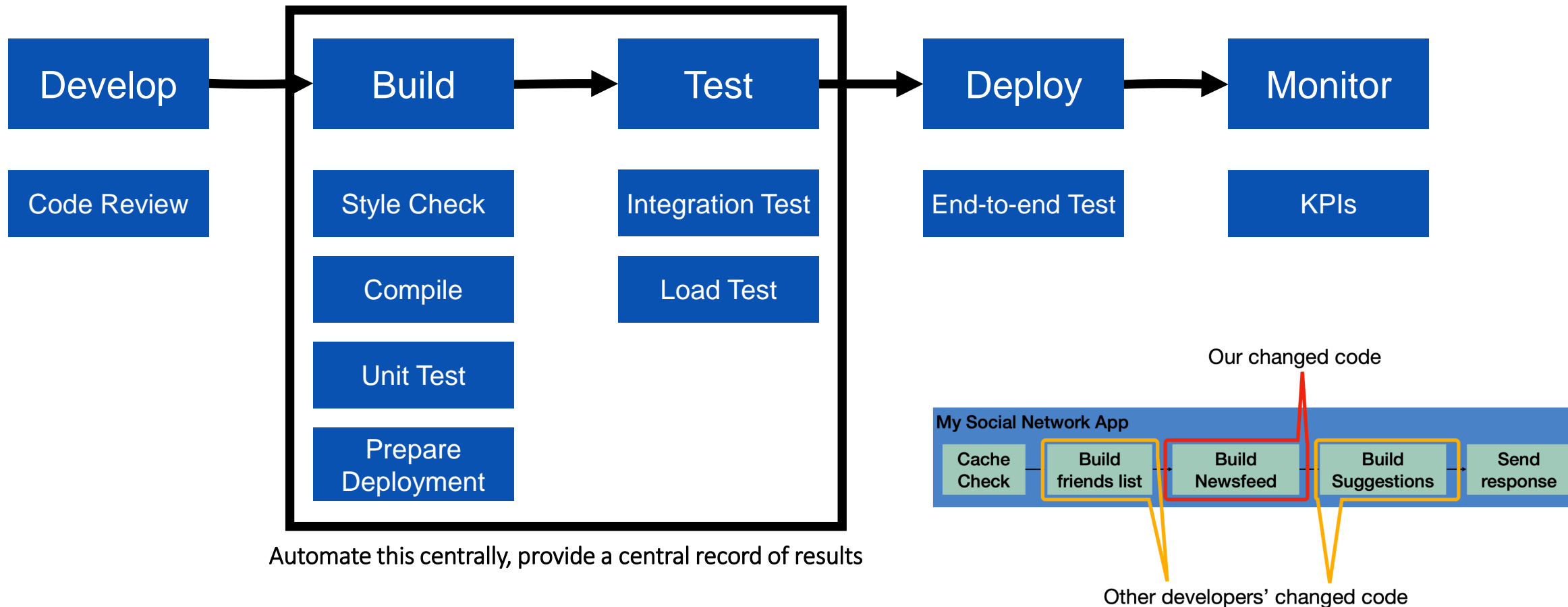
# Continuous Integration (CI) provides global feedback on local changes

- 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?



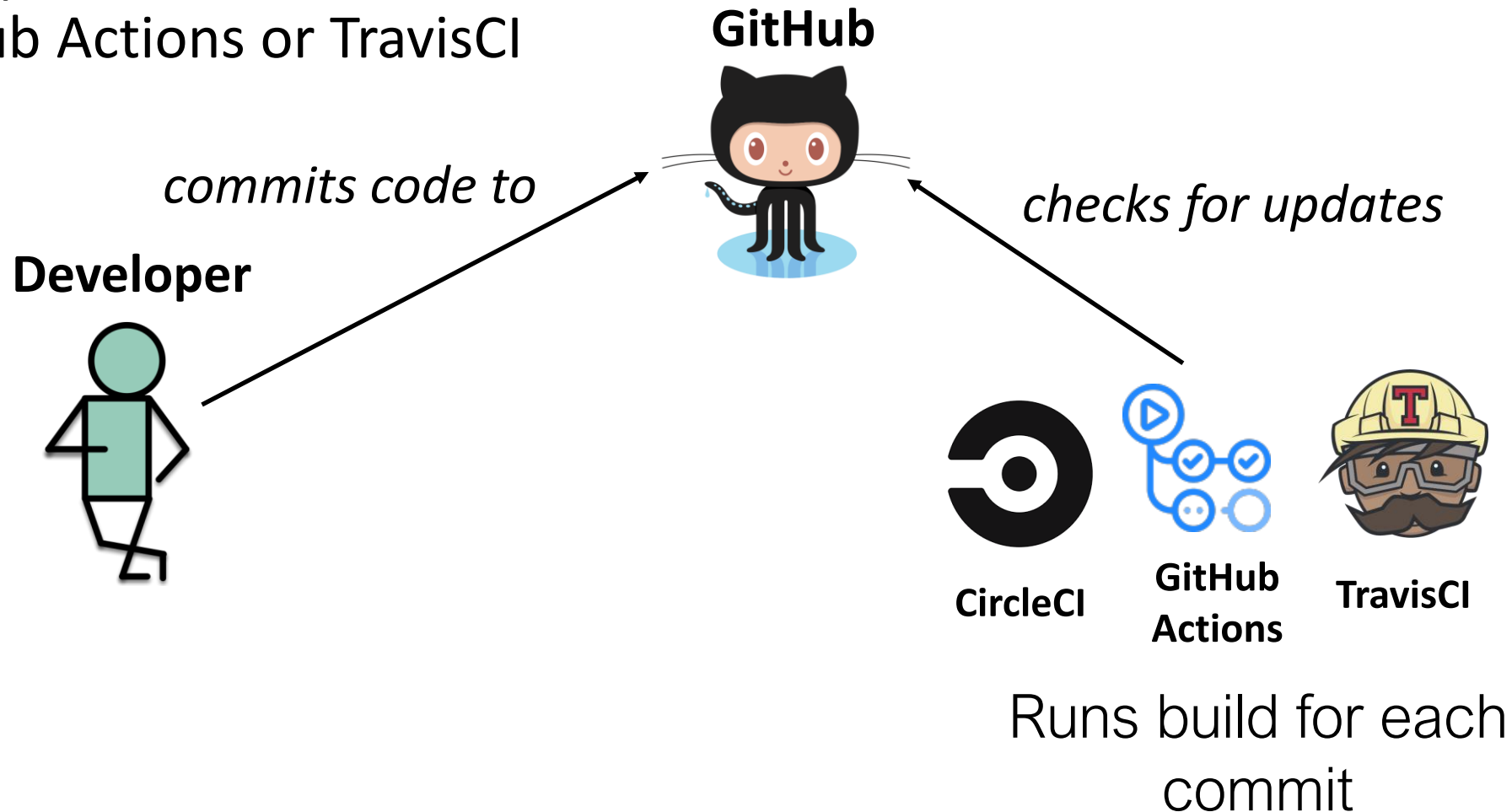


# A CI process is a software pipeline



# CI is triggered by commits, pull requests, and other actions

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



# Automating Feedback Loops is Powerful

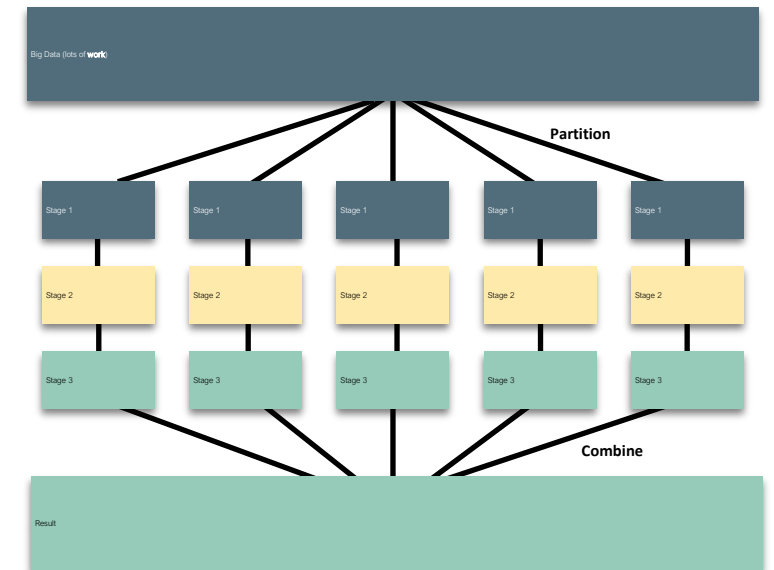
Consider tasks that are done by *dozens* of developers  
(e.g. testing/deployment)

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

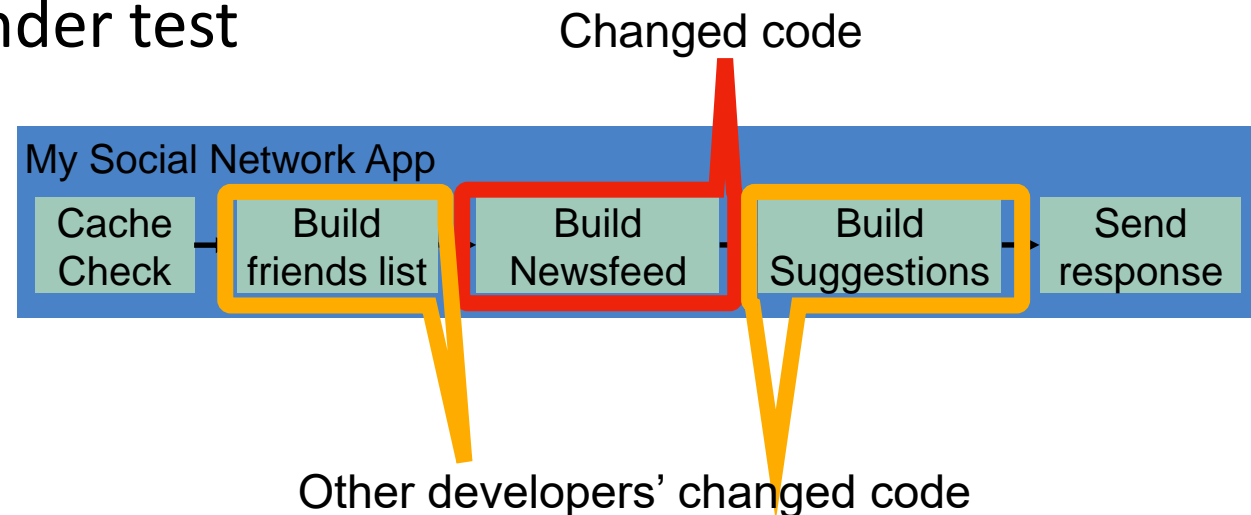
# Typical CI pipeline

- 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



# 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?



# 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!

# CI In Practice: Autograder

GitHub Actions Results

test.yml (CI workflow file)

```
name: 'Build and Test the Grader'
on: # rebuild any PRs and main branch changes
  pull_request:
  push:
    branches:
      - main
      - 'releases/*'
jobs:
  build:
    runs-on: self-hosted
    steps:
      - uses: actions/checkout@v2
      - uses: actions/setup-node@v2
        with:
          node-version: '16'
      - run: |
          npm install
  test:
    runs-on: self-hosted
    strategy:
      matrix:
        submission: [a, b, c, ts-ignore, linting-error, non-green-tests, empty]
    steps:
      - uses: actions/checkout@v2
      - uses: actions/setup-node@v2
        with:
          node-version: '16'
      - uses: ./
        with:
          submission-directory: solutions/${{ matrix.submission }}
```

test.yml

on: push

✓ build 30s

Matrix: test

✓ test (a) 3m 6s

✓ test (b) 3m 3s

✓ test (c) 2m 58s

✓ test (ts-ignore) 5s






















✓ test (linting-error) 31s

✓ test (non-green-tests) 35s

✓ test (empty) 4s

# Example CI Pipeline - Autograder

- At a glance, see history of build

 <b>linting</b> Build and Test the Grader #11: Commit <a href="#">f3da101</a> pushed by jon-bell	main	 5 months ago ...  4m 20s
 <b>Update handout and reference solution ba...</b> Check dist/ #10: Commit <a href="#">3073a5b</a> pushed by jon-bell	main	 5 months ago ...  41s
 <b>Update handout and reference solution ba...</b> Build and Test the Grader #10: Commit <a href="#">3073a5b</a> pushed by jon-bell	main	 5 months ago ...  4m 29s
 <b>Max 2 hints per mutant, provide the tests t...</b> Build and Test the Grader #9: Commit <a href="#">4cfe4ee</a> pushed by jon-bell	main	 6 months ago ...  4m 45s
 <b>Max 2 hints per mutant, provide the tests t...</b> Check dist/ #9: Commit <a href="#">4cfe4ee</a> pushed by jon-bell	main	 6 months ago ...  39s
 <b>New hint generator</b> Check dist/ #8: Commit <a href="#">012e440</a> pushed by jon-bell	main	 6 months ago ...  39s
 <b>New hint generator</b> Build and Test the Grader #8: Commit <a href="#">012e440</a> pushed by jon-bell	main	 6 months ago ...  5m 9s



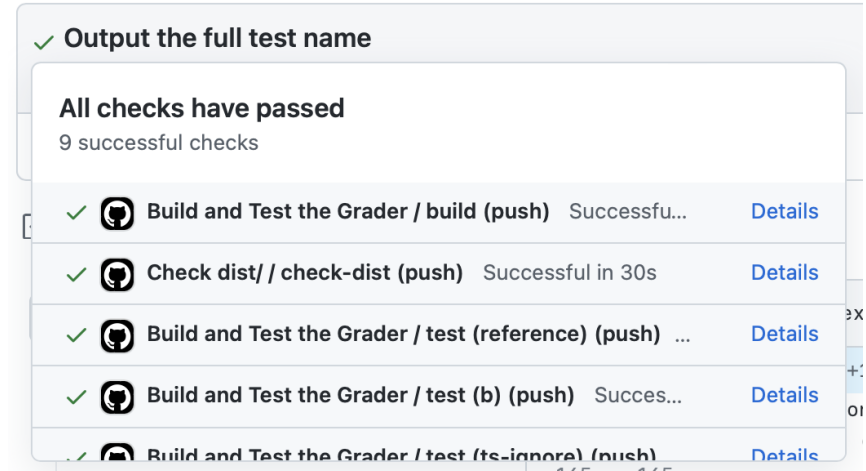
# Attributes and challenges for designing an effective CI process

---

- Attributes of effective CI processes
- Challenges for effective CI processes

# Attributes of effective CI processes

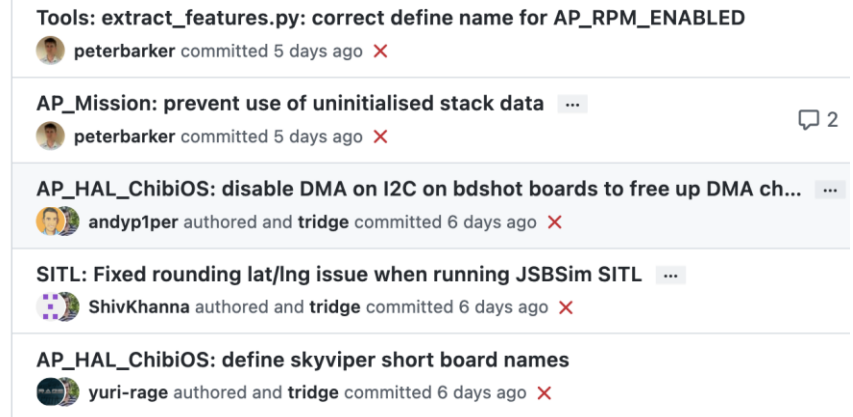
- Policies:
  - Do not allow builds to remain broken for a long time
  - CI should run for every change
  - CI should not completely replace pre-commit testing
- Infrastructure:
  - CI should be fast, providing feedback within minutes or hours
  - CI should be repeatable (deterministic)



✓ Output the full test name

All checks have passed  
9 successful checks

- ✓ Build and Test the Grader / build (push) Successful... [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 ...  
peterbarker committed 5 days ago ✗ 2

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 ✗

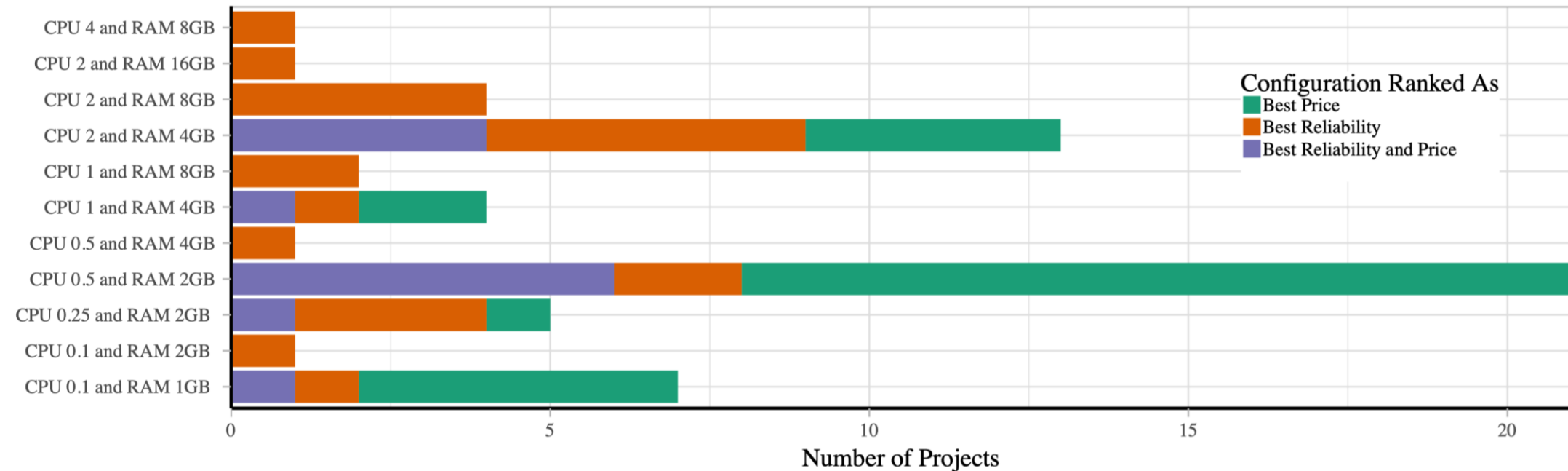
# Effective CI processes are run often enough to reduce debugging effort

- 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

Status	Branch	Commit Hash	Build ID	Duration	Time
✓	master	36392a2	#52300 passed	10 hrs 49 min 31 sec	2 days ago
✗	master	aa55ea7	#52287 errored	11 hrs 6 min 44 sec	2 days ago
✗	master	193a4cd	#52284 errored	11 hrs 50 min 37 sec	2 days ago
✓	master	fff331f	#52283 passed	11 hrs 3 min 20 sec	2 days ago
✓	master	746d7b5	#52282 passed	10 hrs 55 min 37 sec	2 days ago
✓	master	a90d97a	#52277 passed	10 hrs 43 min 30 sec	2 days ago
✗	master	8b62443	#52271 errored	10 hrs 46 min 36 sec	3 days ago
✗	master	467277a	#52266 failed	10 hrs 35 min 49 sec	3 days ago
✓	master	fc94719	#52263 passed	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



# 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...

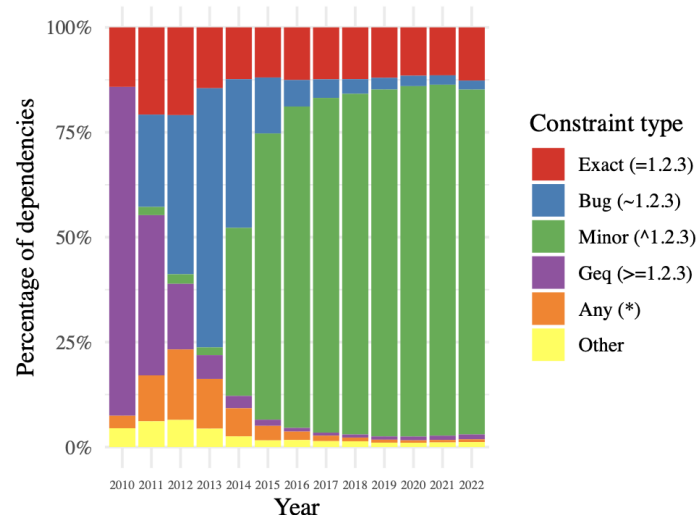
# Dependency Managers Organize External Dependencies

---

- 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)

2.0.0 2.0.0-rc.2 2.0.0-rc.1 1.0.0 1.0.0-beta

## 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?

# 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

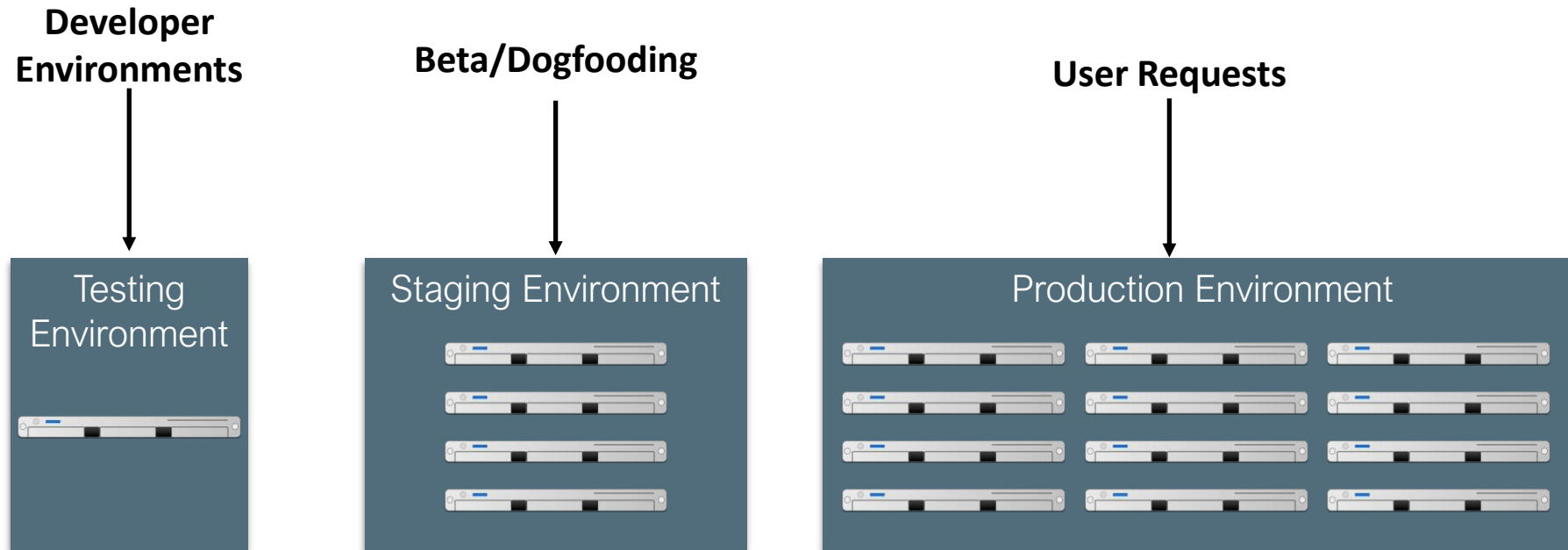
# Ways to mitigate deployment risks

---

- 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

---



Revisions are “promoted” towards production



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

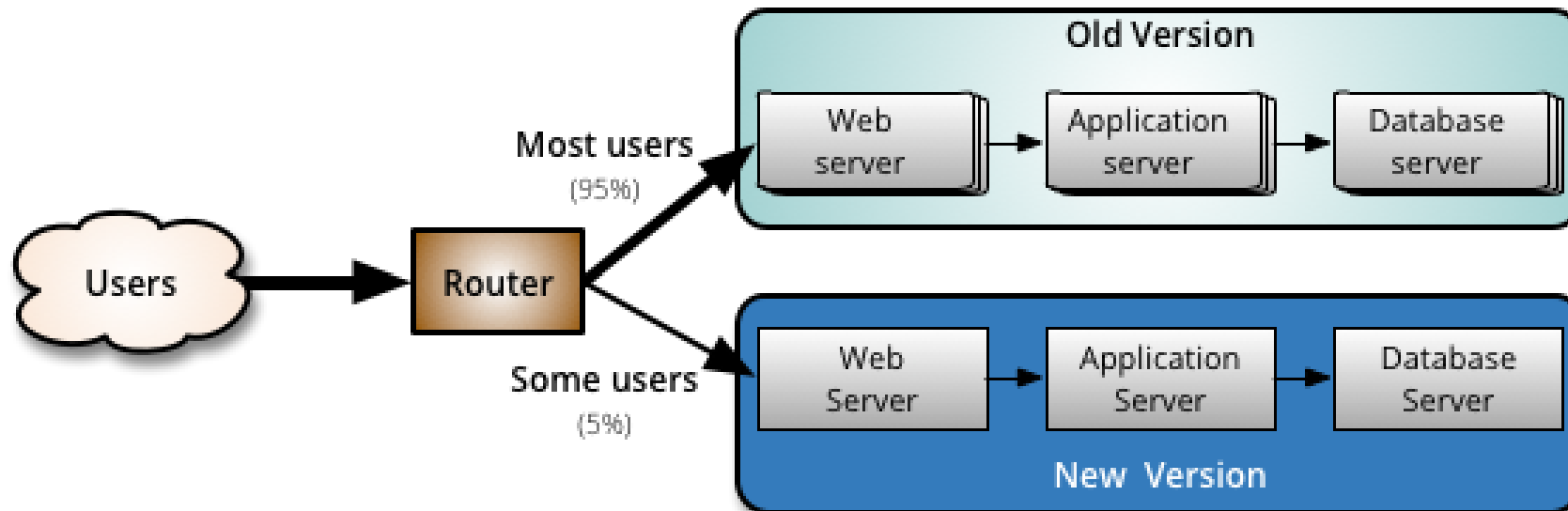
# Post-delivery monitoring mitigates risk

---

- 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

# Split Deployments Mitigate Risk

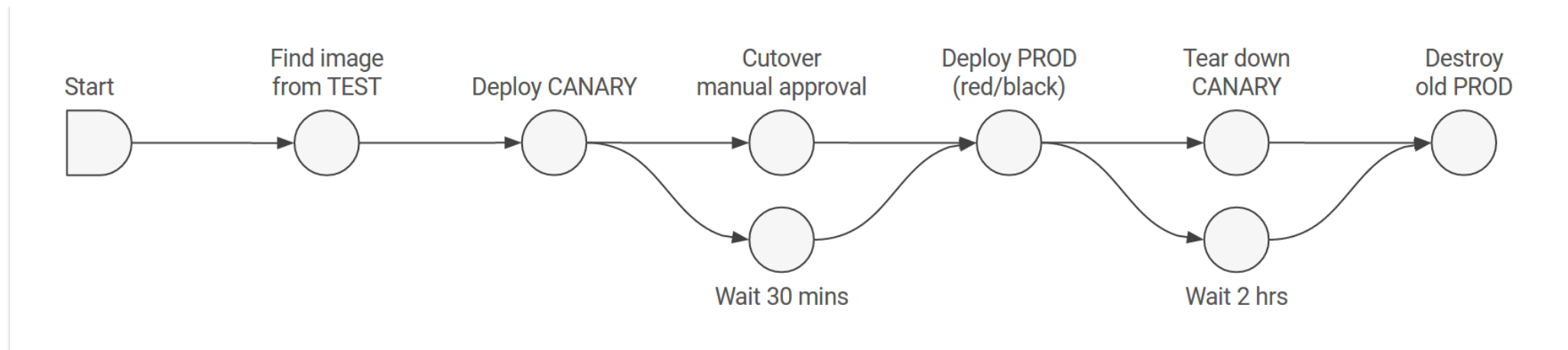
- Idea: Deploy to a complete production-like environment, but don't have users use it, collect preliminary feedback
- Lower risk if a problem occurs in staging than in production
- Examples:
  - “Eat your own dogfood”
  - Beta/Alpha testers





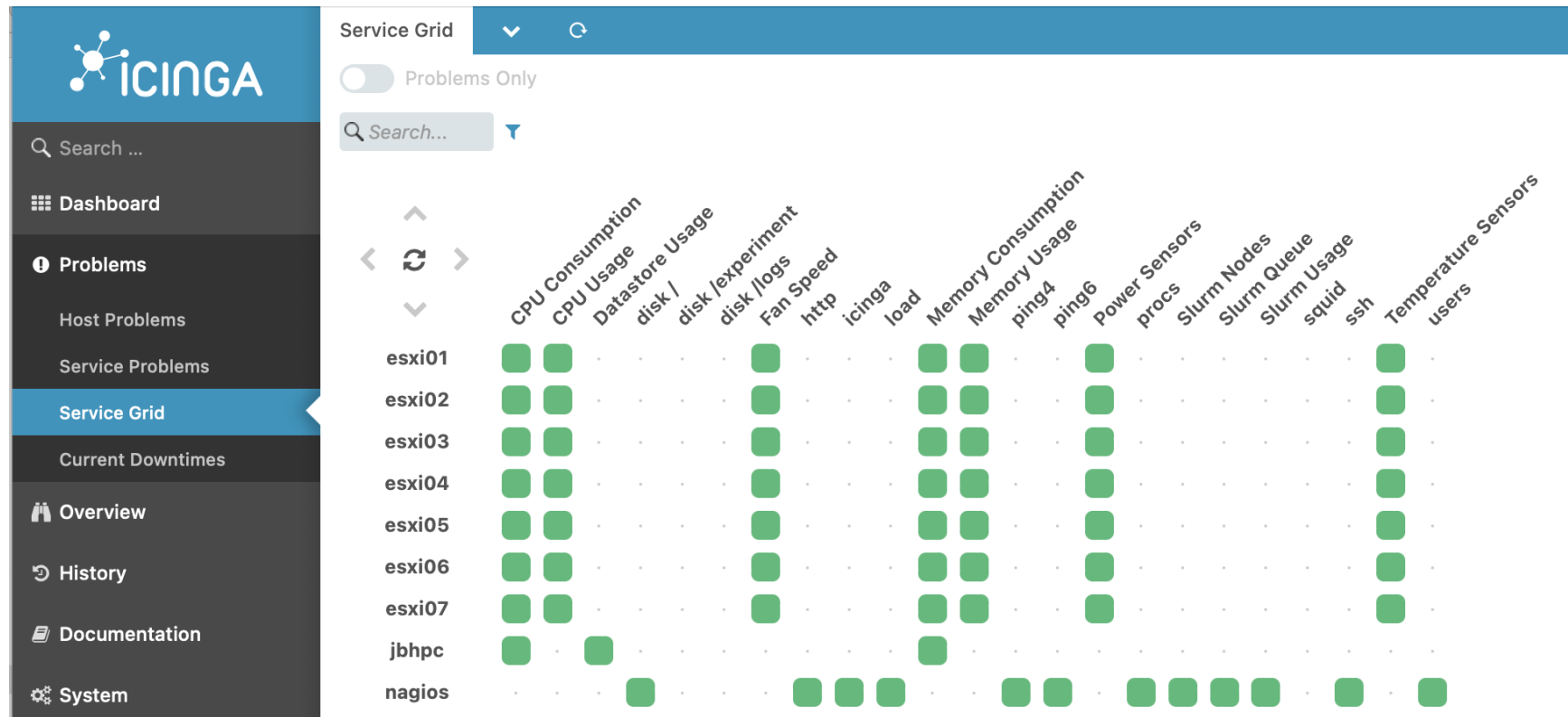
# Continuous Delivery Tools

- 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)

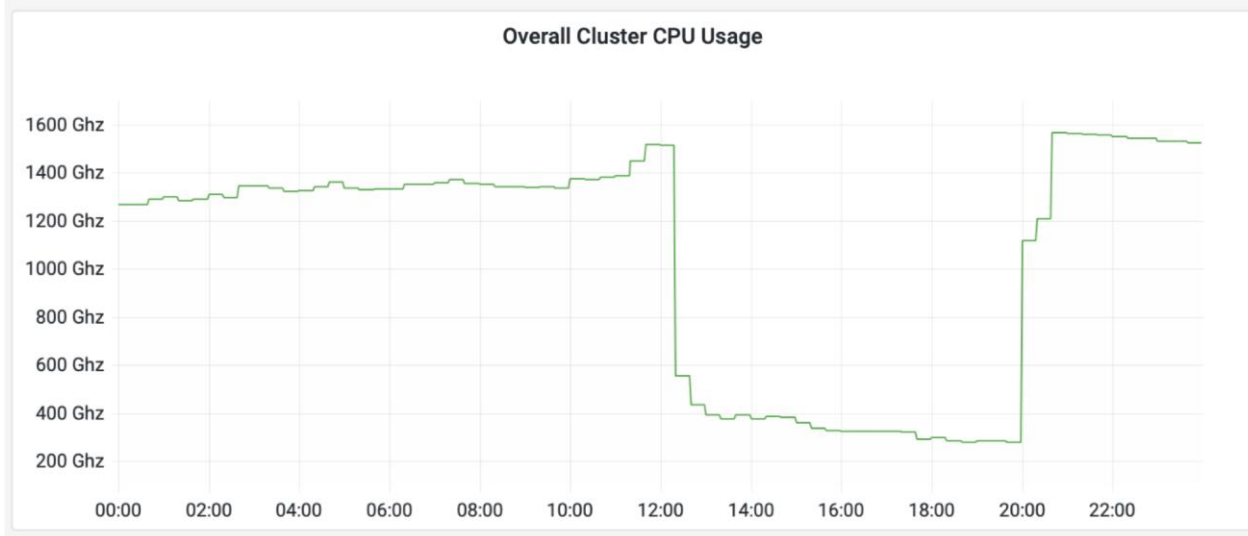
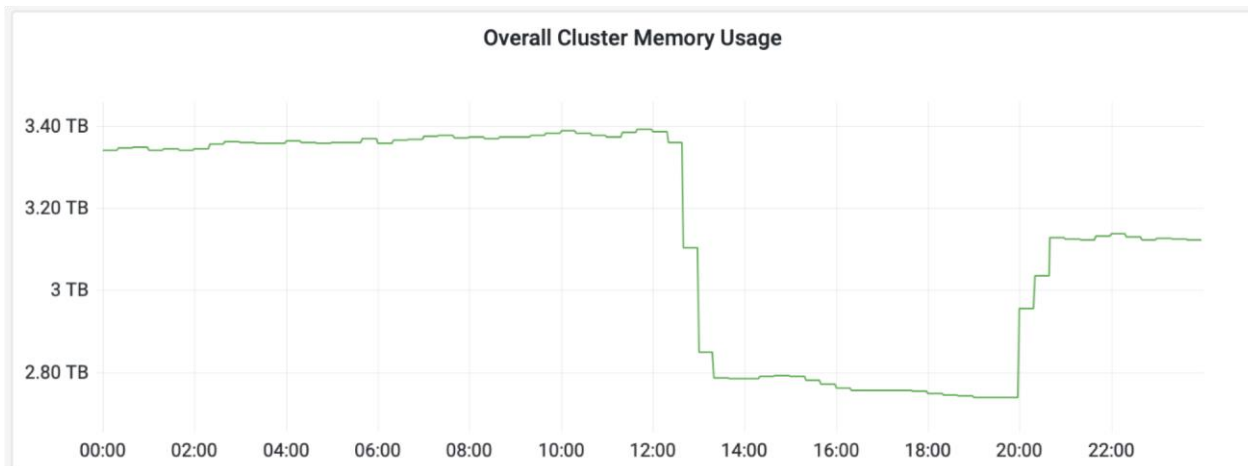


# Tools for Monitoring Deployments

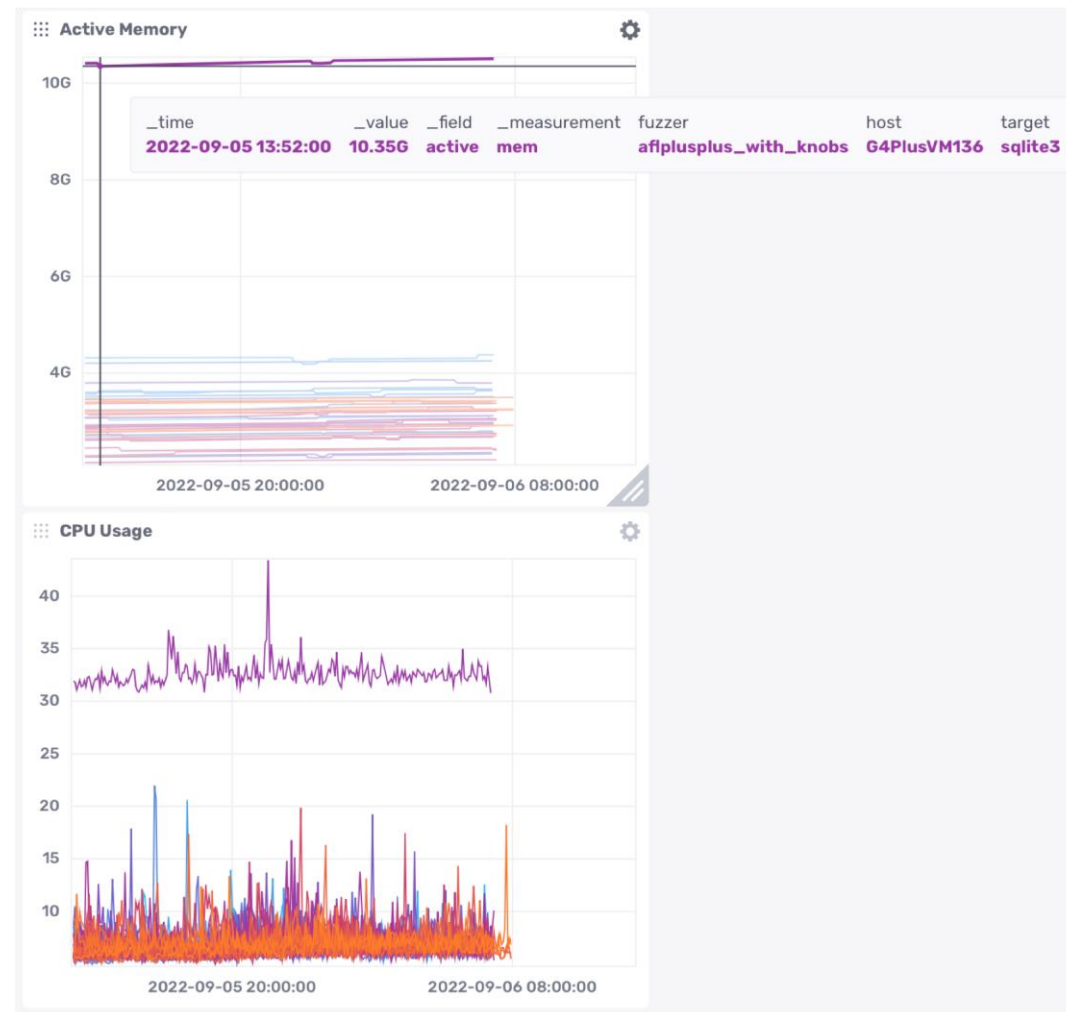
- Nagios (c 2002): Agent-based architecture (install agent on each monitored host), extensible plugins for executing “checks” on hosts
- Track system-level metrics, app-level metrics, user-level KPIs



# Monitoring can help identify operational issues



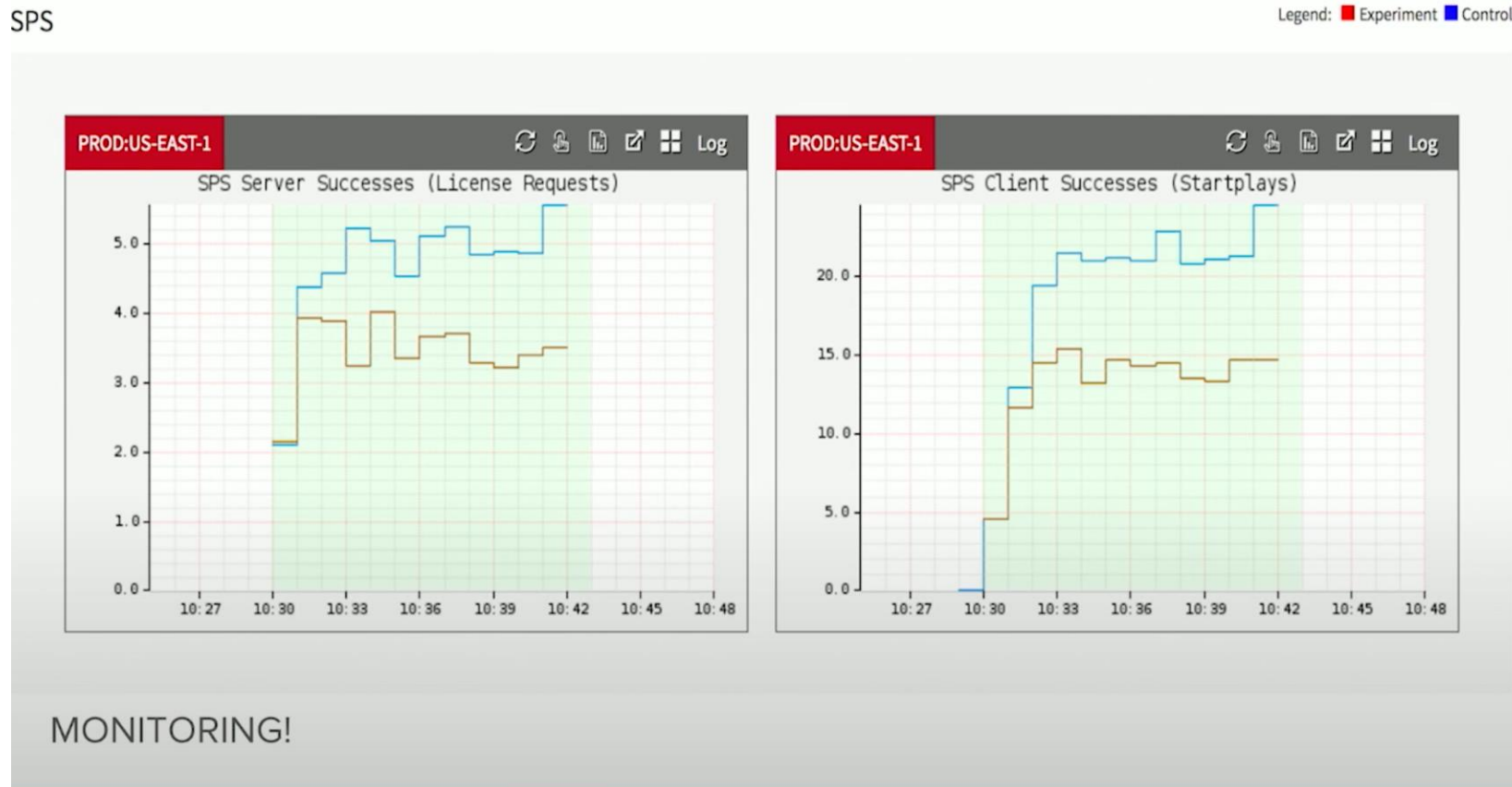
Grafana (AGPL, c 2014)



InfluxDB (MIT license, c 2013)

# Continuous Delivery Tools Take Automated Actions

- Example: Automated roll-back of updates at Netflix based on SPS



# What not to do: Failed Deployment at Knight Capital

## 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.

“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. “



# 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

# Monitoring Services Take Automated Actions

The screenshot displays the Icinga monitoring interface. On the left is a navigation sidebar with the Icinga logo and menu items: Search, Dashboard, Problems, Overview, History (with sub-items Event Grid and Event Overview), Notifications (highlighted), Timeline, Documentation, System, Configuration, and user 'jon'. The main content area is titled 'Notifications' and shows a list of notifications for 'Slurm Nodes on nagios'. The notifications are color-coded by severity: OK (green), WARNING (orange), and CRITICAL (red). The list includes a search bar, pagination (page 25 of 25), and a 'Sort by' dropdown set to 'Notification Start'. A detailed view on the right shows the 'Current Service State' for 'nagios' as 'UP' since 2021-11, with IP 127.0.0.1. Below this, it shows 'OK for 1m 52s' for the 'Service: Slurm Nodes'. The 'Event Details' section provides metadata for the notification, including 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), and Contacts notified (2). The Output field shows the notification message: 'CRITICAL - 65 nodes unreachable, 161 reach'.

Severity	Time	Message	Recipient
OK	2022-02-18 08:49:05	OK - 0 nodes unreachable, 332 reachable	Sent to jon
OK	2022-02-18 08:49:05	OK - 0 nodes unreachable, 332 reachable	Sent to icingaadmin
WARNING	2022-02-18 08:45:05	WARNING - 7 nodes unreachable, 326 reachable	Sent to jon
WARNING	2022-02-18 08:45:05	WARNING - 7 nodes unreachable, 326 reachable	Sent to icingaadmin
CRITICAL	2022-02-18 08:42:05	CRITICAL - 65 nodes unreachable, 161 reachable	Sent to icingaadmin
CRITICAL	2022-02-18 08:42:05	CRITICAL - 65 nodes unreachable, 161 reachable	Sent to jon
WARNING	2022-02-18 08:40:05	WARNING - 12 nodes unreachable, 205 reachable	Sent to icingaadmin
WARNING	2022-02-18 08:40:05	WARNING - 12 nodes unreachable, 205 reachable	Sent to jon
CRITICAL			

**Current Service State**

UP since 2021-11 ::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	<span style="color: red;">■</span> CRITICAL
Escalated	No
Contacts notified	2
Output	CRITICAL - 65 nodes unreachable, 161 reach

# 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

# Compare Continuous Delivery and TDD

---

- Test driven development
  - Write and maintain tests per-feature
  - 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 in practice at Google

---

- Large scale example: Google TAP
  - 50,000 unique changes per-day, 4 billion test cases per-day
  - Pre-submit optimization: run fast tests for each individual change (before code review).  
Block merge if they fail.
  - Then: run all affected tests; “build cop” monitors and acts immediately to roll-back or fix
  - Build cop monitors integration test runs
  - Average wait time to submit a change: 11 minutes

# Facebook: "Move fast and break things"

---

- de-prioritize unit tests
- Emphasis on getting features to users quickly

# Facebook used to have an elaborate system of branches

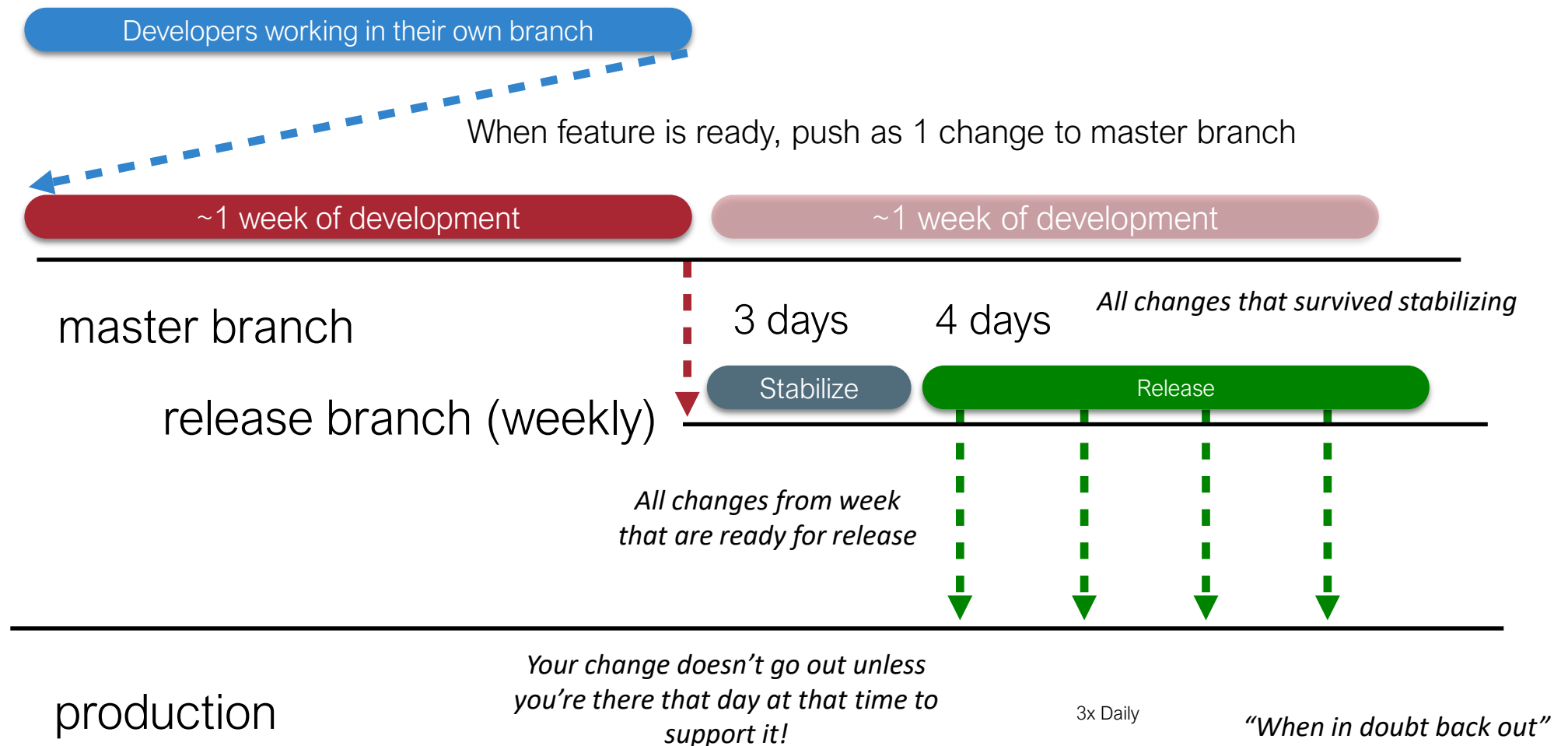
---

- dev branches get merged into master,
- then once a week all changes from the past week are pulled into a release branch.
- For 3 days they “stabilize” the release branch – find changes that are causing very bad behavior and back them out.
- Then for the last 4 days of the week, every change that survived that stabilization gets individually pushed to production batched so that this happens 3x/day.
- Important to do small deploys so that you can isolate bad changes)
- And most important: "your change doesn't go out unless you're there that day to support it."

But this didn't scale. How many changes going out? 500-700 PER DAY. By 2016 might be pushing 10k diffs per week. ENORMOUS effort to co-ordinate.

# Deployment Example: Facebook.com

- Pre-2016

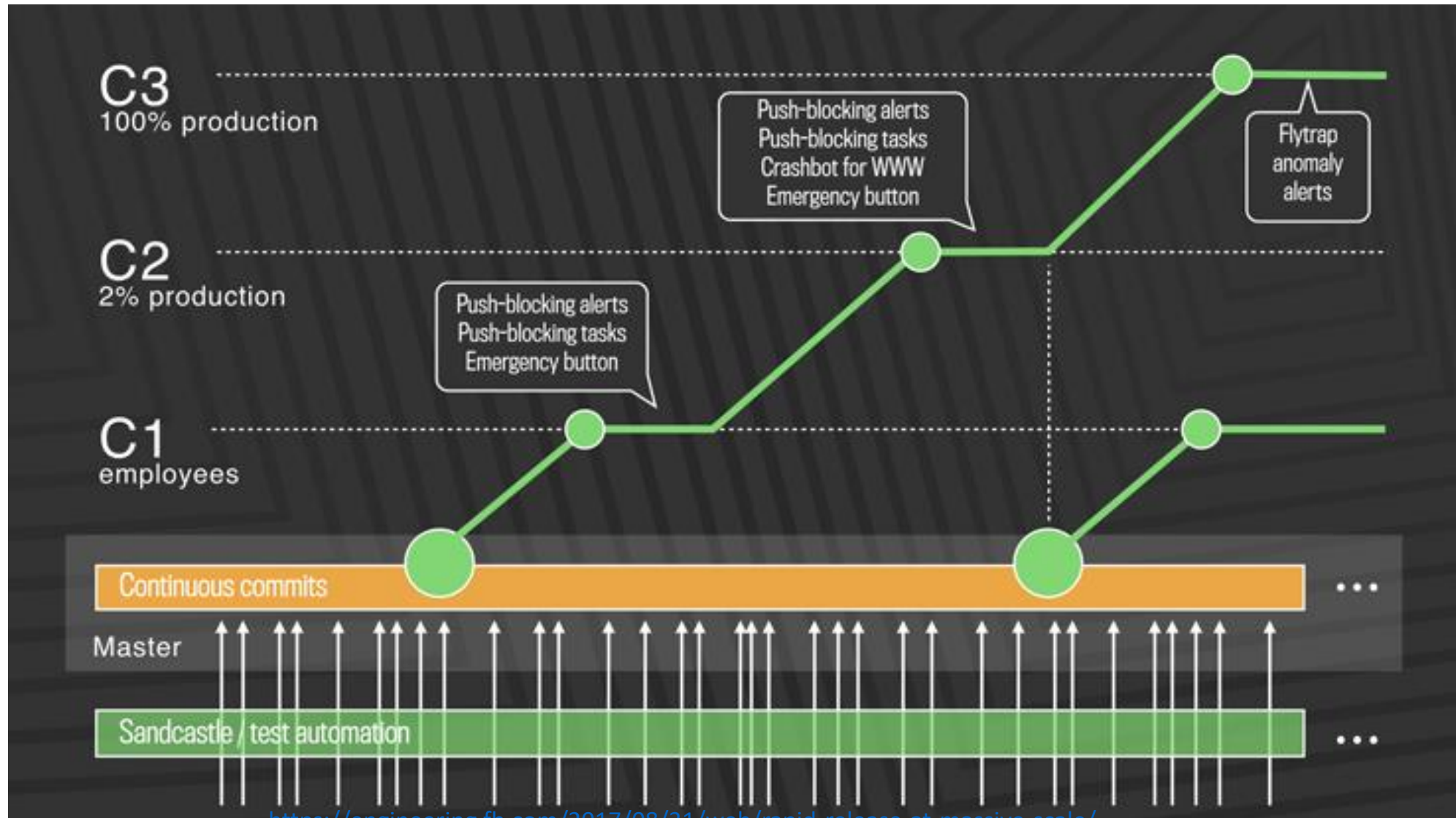


# Post-2016: Truly Continuous Releases from Master Branch (excerpts from blog post)

---

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.

# Post-2016: truly continuous releases from master branch





# Review

---

- By now, you should be able to...
  - Describe how continuous development 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