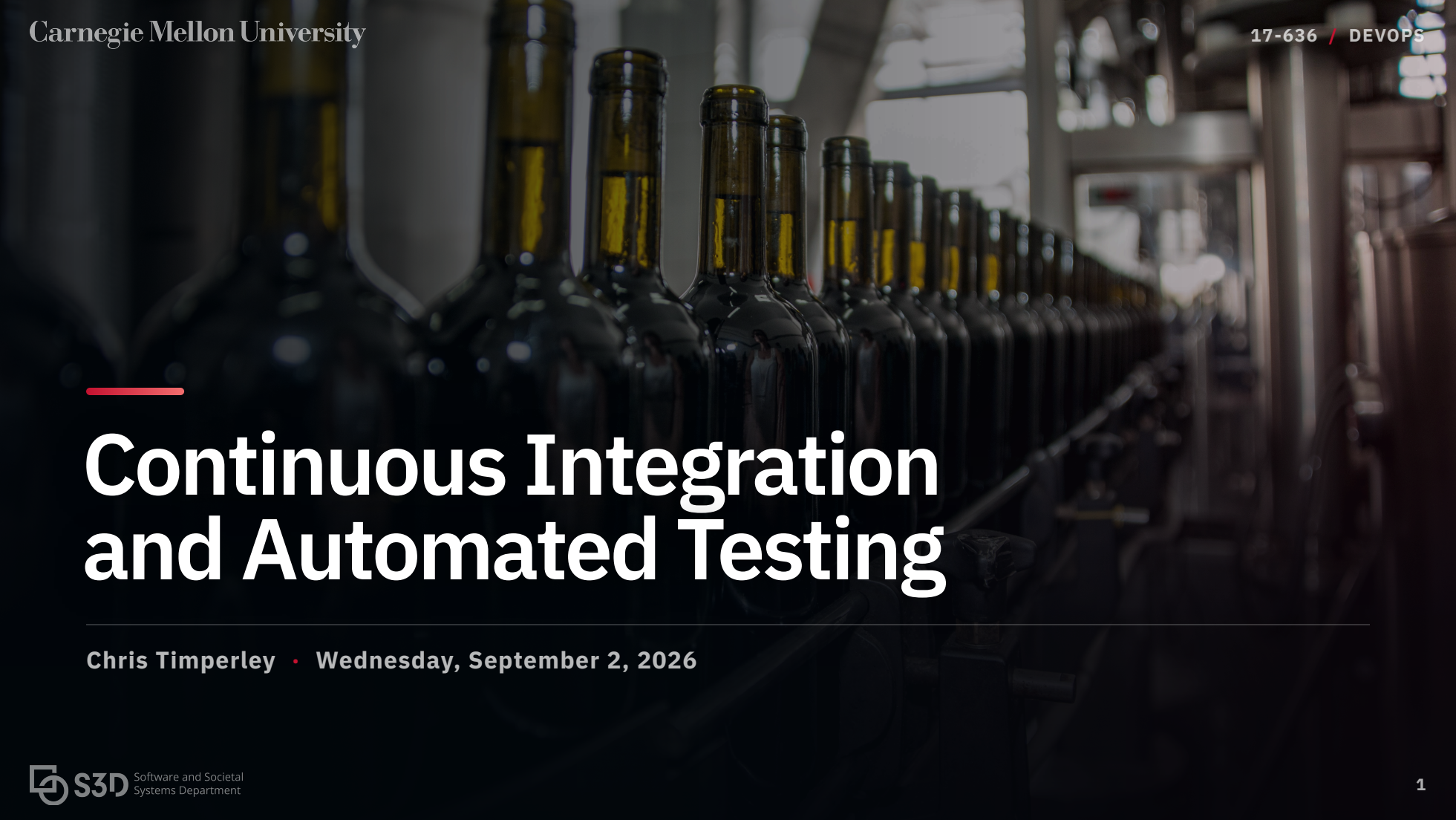


Continuous Integration and Automated Testing

Chris Timperley · Wednesday, September 2, 2026

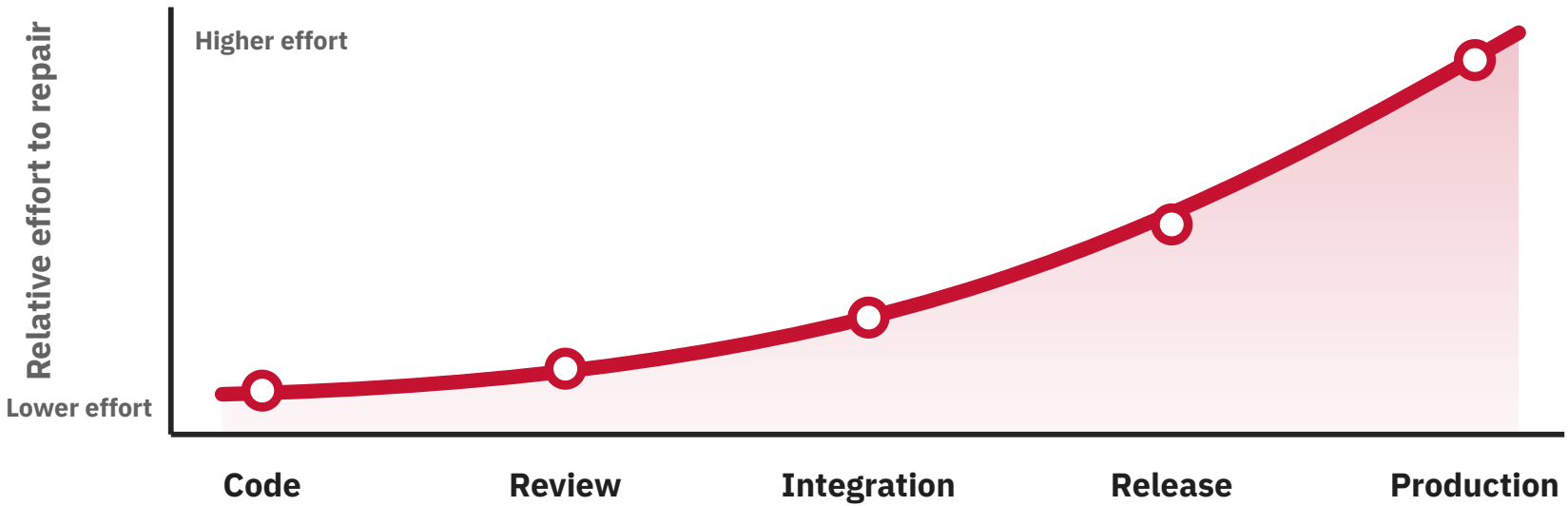
🎯 Learning Goals

- Explain how continuous integration and shifting left keep the main branch working and make problems cheaper to fix.
- Choose the earliest credible automated check for a risk using the test pyramid.
- Decide which CI results should block a merge and which decisions need human review.
- Compare CI triggers by coverage, cost, and feedback time.
- Diagnose slow, flaky, or noisy CI and choose a matching improvement.

Shifting Left

Could we have found or stopped this issue earlier?

Bugs cost more to fix when they are found late



Unreliable checks let bad changes reach customers

Bad change enters delivery

↓

No reliable check catches it

↓

Change ships

↓

Customer encounters the bug

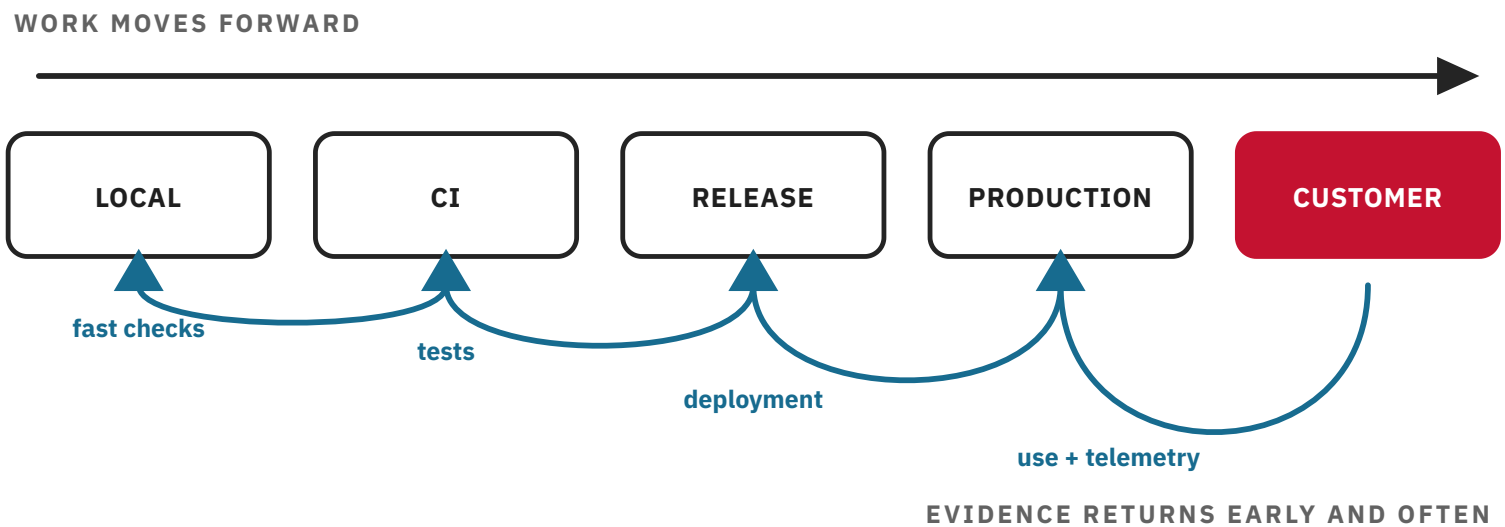


Pat Recipient
@parcel_pat

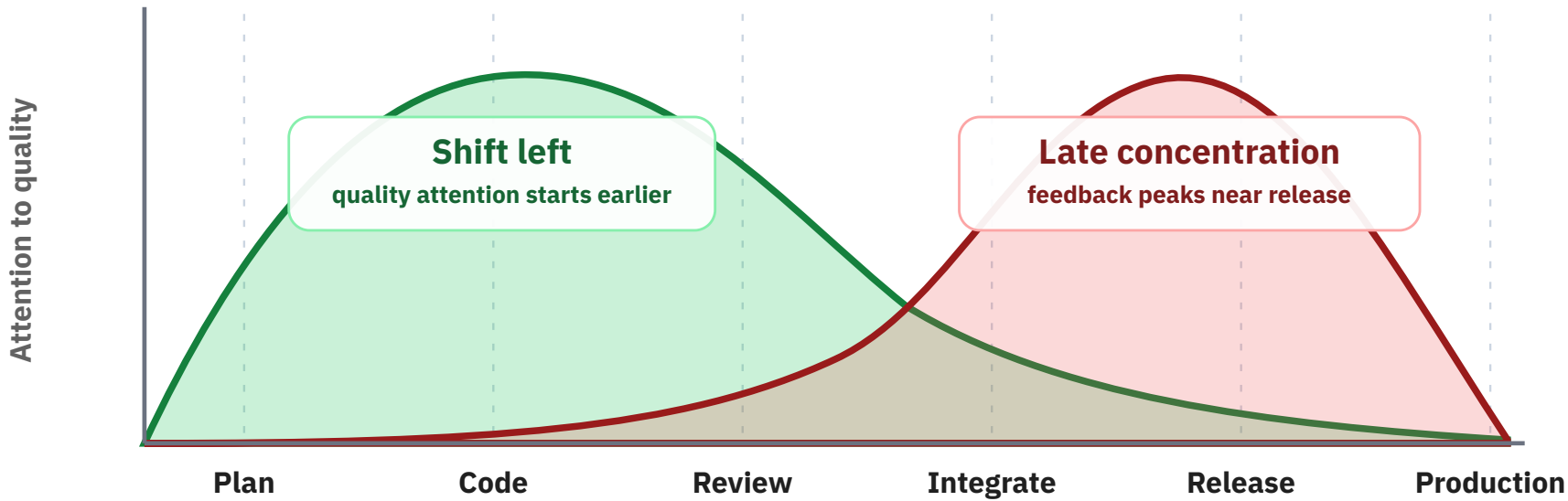
Your tracking page says **delivered**, but the photo is someone else's porch. Should I start knocking on doors? 😬

9:42 AM · Fictional example

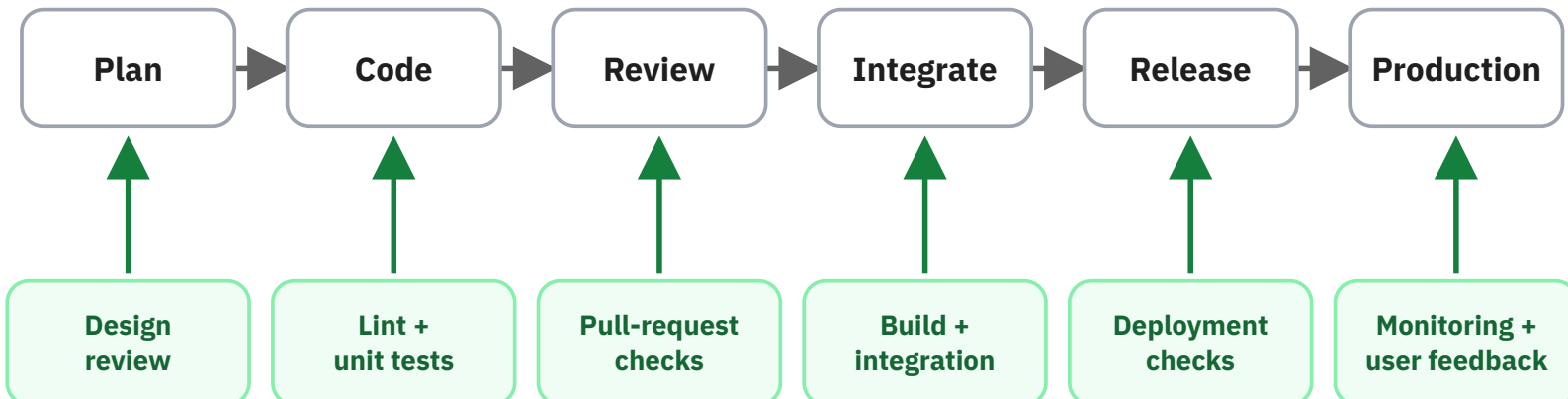
Recall: The Second Way amplifies fast feedback at every stage



Shifting left creates earlier, faster feedback



Checks turn the delivery lifecycle into feedback loops



CI checks small integrations before they pile up

Continuous integration (CI) is a team practice:

- **Integrate often:** merge small changes into a shared `main` branch.
- **Check every integration:** run automated checks on the combined code.
- **Respond quickly:** fix failures before more changes pile up.

Small batches narrow the changes and interactions to investigate.









CI is a system of reinforcing practices

1

One shared mainline

The team integrates into the same current version.

2

Versioned inputs

Code, tests, configuration, and build instructions can be recovered.

3

Automated build and tests

One repeatable process checks the combined system.

4

Check every integration

Each change to `main` produces fresh evidence.

5

Fast, visible results

The team can see the current result while the change is fresh.

6

Immediate repair

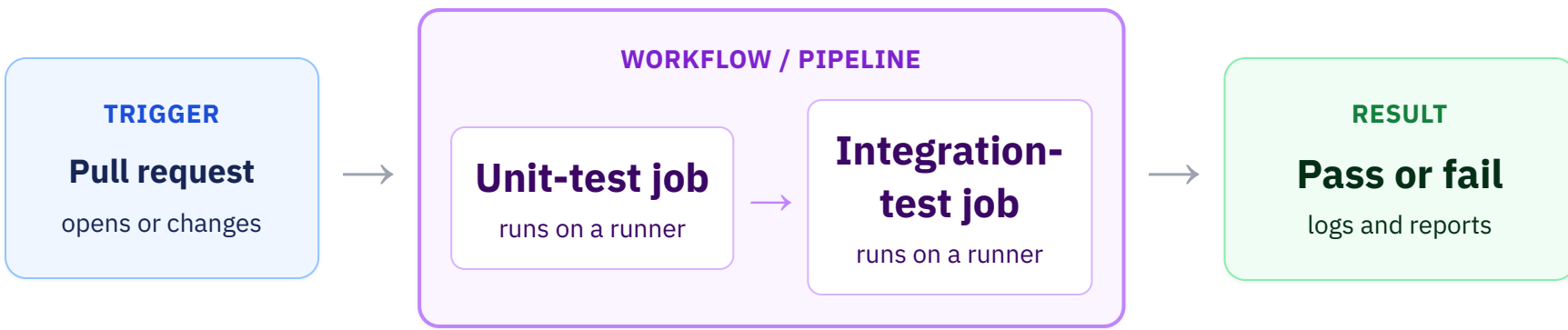
A failed build is fixed before unrelated work is merged.

CI Pipelines

What is a CI pipeline?

A CI workflow connects a trigger to jobs and results

CI platforms may call the full automated process a **workflow** or a **pipeline**.



A workflow runs jobs when a trigger occurs

```yaml
name: Parcel API CI

on: [pull_request]

jobs:
  unit:
    runs-on: ubuntu-latest
    steps: [ ... ]

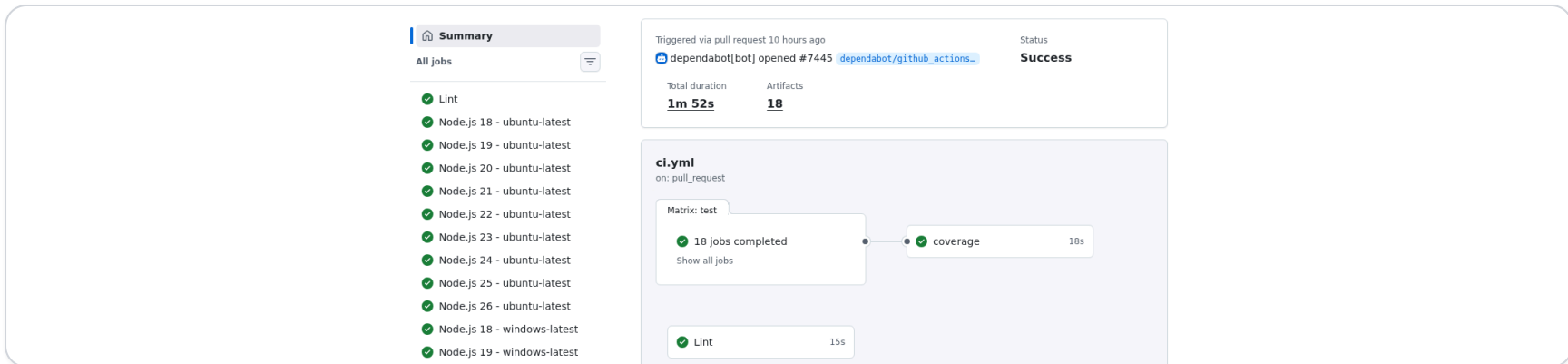
  integration:
    needs: unit
    runs-on: ubuntu-latest
    steps: [ ... ]
```



GitHub Actions

1. **Workflow: an automated process, such as building and testing a proposed change.**
2. **Trigger: an event that starts a workflow, such as a push, pull request, schedule, or manual request.**
3. **Job: a group of related steps, such as building the service or running unit tests.**
4. **Runner: a machine or process that provides the environment and executes a job.**
5. **Result: evidence from a run, such as pass/fail status, logs, or test reports.**

Example: Express runs tests in parallel, then combines coverage



GitHub Actions: Express

What should the CI check?

Automated Testing

Use the earliest check that can find each problem

A manual check cannot protect the next change

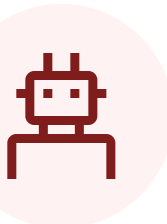
Parcel tracker

Destination

Pittsburgh, PA

Ready for pickup

Manually checked and recorded



A later change

A refactor or agent makes changes to the codebase.

Parcel tracker

Destination

Pittsburgh, PA

Status missing

No check reports the regression

The defect can go undetected until we push to production.

We need a check that runs after every change.

We could do end-to-end testing via browser automation

```
import { test, expect } from '@playwright/test'

test('customer creates a parcel', async ({ page }) => {
  await page.goto('/parcels/new')
  await page.getByLabel('Destination').fill('Pittsburgh, PA')

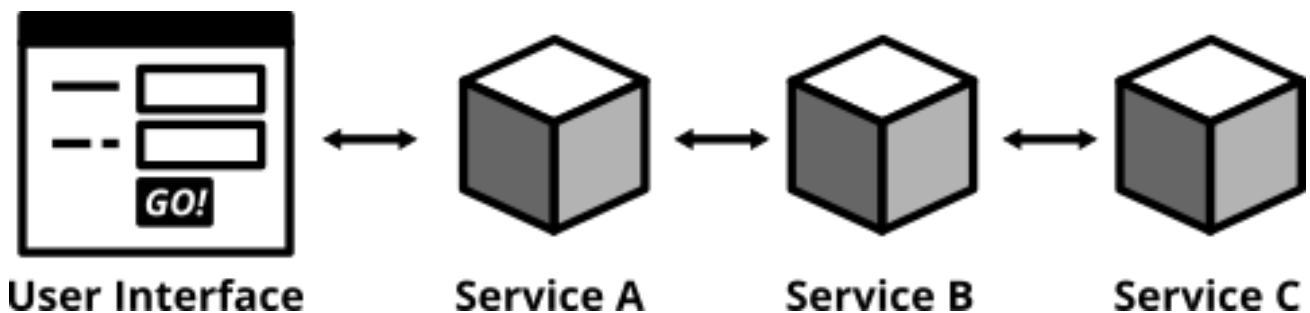
  await page.getByRole('button', { name: 'Create' }).click()
  const status = page.getByText('Ready for pickup')
  await expect(status).toBeVisible()
})
```





End-to-end tests give “great bang for your buck”

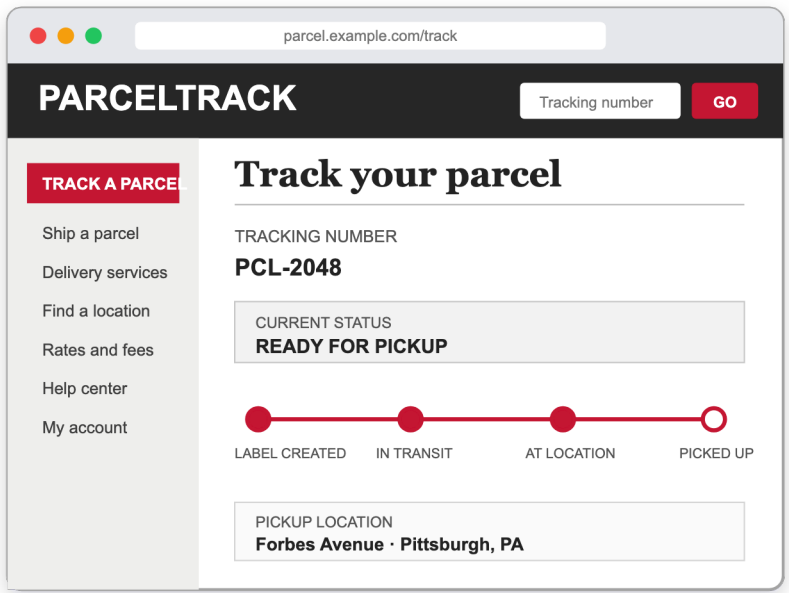
- A single test can provide **lots of coverage**
- Stresses integrations and gives **high confidence** in overall system behavior
- We can use tooling and AI to help generate E2E tests

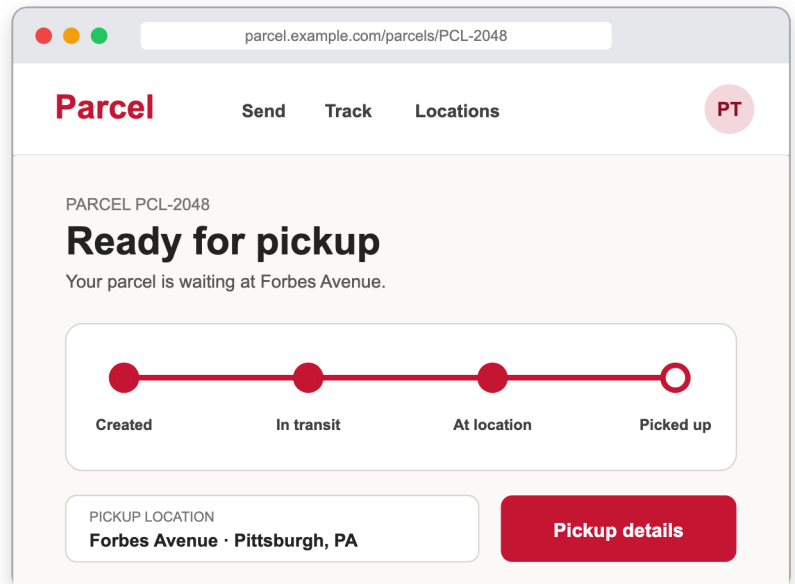


Why don’t we just run end-to-end tests in the CI?



End-to-end tests can be fragile: software evolves



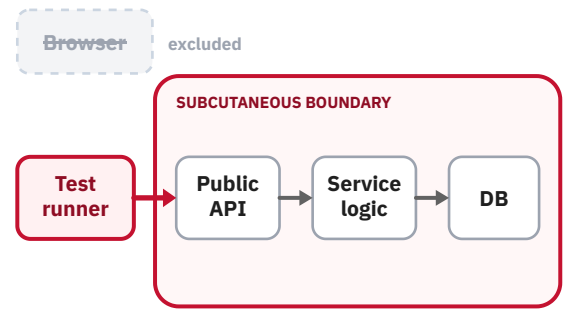


Subcutaneous tests skip the UI but test the application

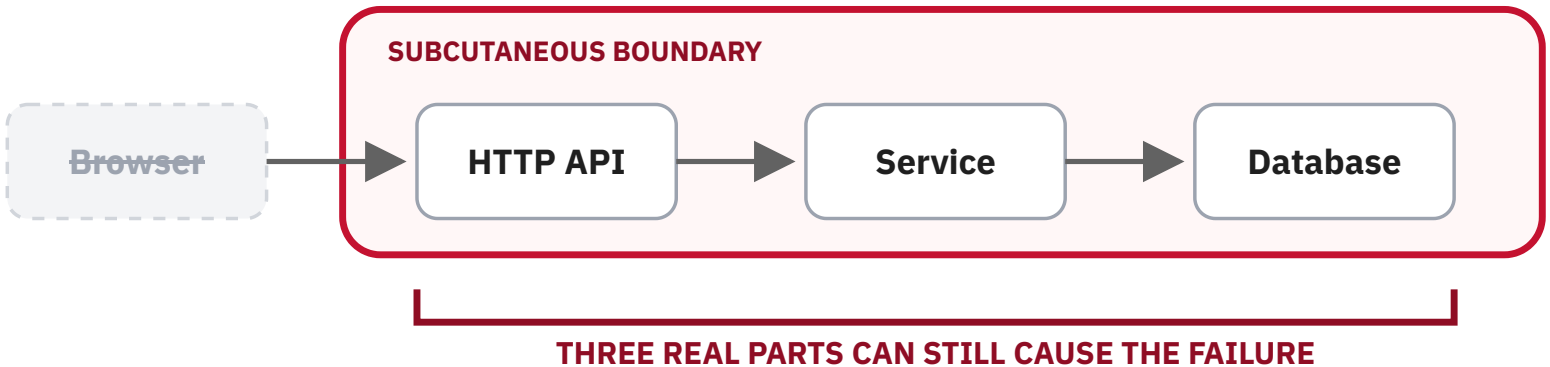
```
import { it, expect } from 'vitest'
import request from 'supertest'

it('creates and tracks a parcel', async () => {
  const created = await request(app)
    .post('/parcels')
    .send({ destination: 'Pittsburgh, PA' })

  const tracked = await request(app)
    .get(`/parcels/${created.body.id}`)
  expect(tracked.body.status).toBe('ready_for_pickup')
})
```

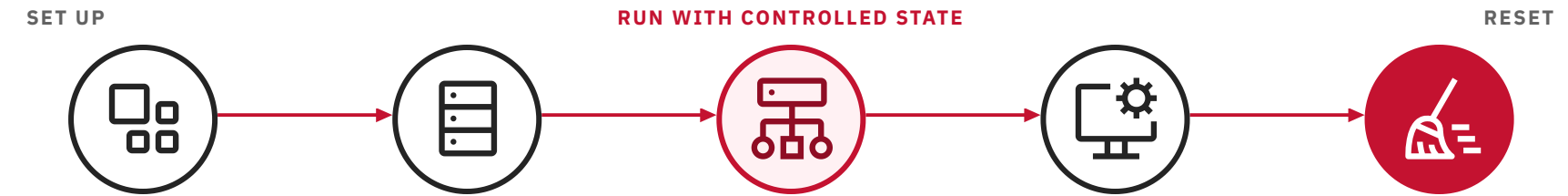


Broad tests make failures harder to isolate



The failing test could cover tens or hundreds of thousands of lines!

Broad tests need an environment before they can run



01
Start services
Application, database, and queue

02
Apply schemas
Migrations and configuration

03
Prepare data
Accounts, parcels, and permissions

04
Control services
Sandbox keys and responses

05
Clean up
Remove state after the test

Broad tests make feedback slower and more expensive



More time

start, prepare, execute, and clean up



More resources

browsers, processes, databases, and services

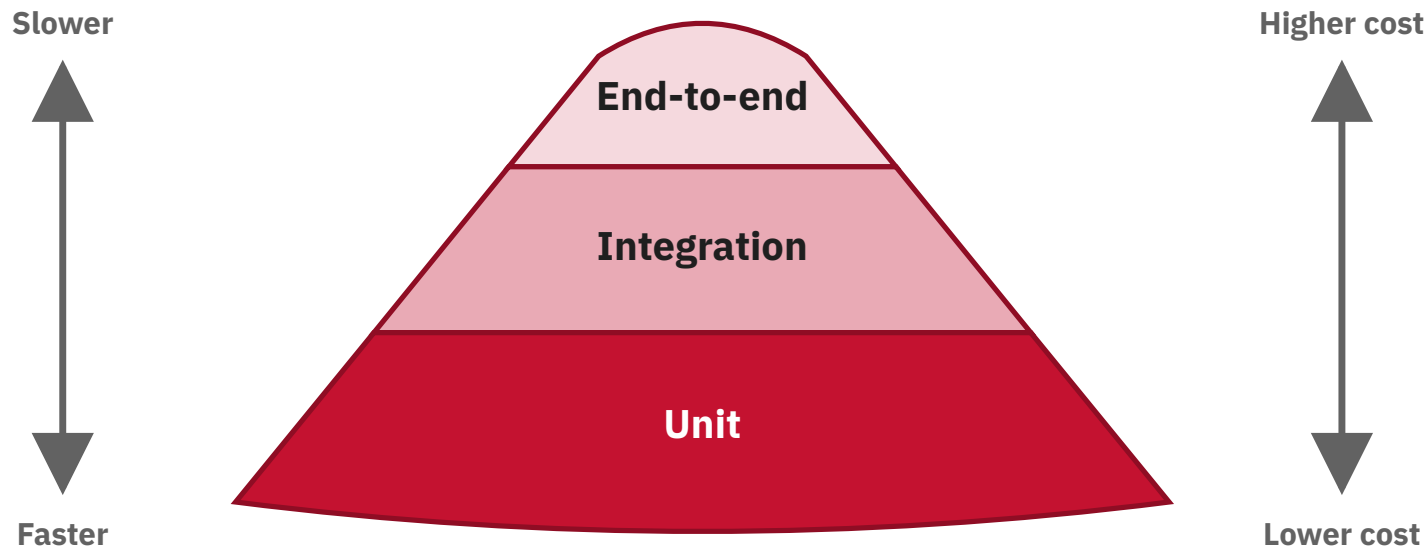


Later feedback

developers and releases wait longer for results

In end-to-end tests, the entire system is under test.

The Test Pyramid favors many small tests and a few broad tests



Simple functions are easy to unit test

```
it('adds the oversize surcharge', () => {
  const parcel = { kg: 2, oversize: true }
  const cents = shippingPrice(parcel)
  expect(cents).toBe(1450)
})

it('rejects a negative weight', () => {
  const price = () => shippingPrice({ kg: -1 })
  expect(price).toThrow()
})
```

Real application code has many collaborators

```
async function createShipment(parcel, deps) {
  const quote = await deps.carrier.quote(parcel)

  const shipment = {
    id: deps.ids.next(),
    ...parcel,
    priceCents:
      quote.cents + oversizeSurcharge(parcel),
    createdAt: deps.clock.now(),
  }

  await deps.shipments.save(shipment)
  await deps.mailer.send(parcel.email, 'Shipment created')

  return shipment
}
```

A **collaborator** is another component the function calls.

Carrier API
network and failures

ID generator
changing values

Clock
changing time

Repository
database writes

Mailer
messages sent to real people

A test double replaces a collaborator during a test

- **Choose its response**, including errors and unusual values.
- **Prevent side effects**, such as database writes or real email.
- **Observe calls** that the unit does not return as output.
- **Make changing values predictable**, such as time and generated IDs.

Keep the real collaborator when it is fast, predictable, and safe.

A stub returns an answer chosen by the test

```
const carrierStub = {
  quote: async () => ({ cents: 800 }),
}

const shipment = await createShipment(
  parcel,
  makeTestDeps({ carrier: carrierStub }),
)

expect(shipment.priceCents).toBe(950)
```

Provides a predefined response.

- **Normal case:** Return an 800-cent quote.
- **Error case:** Throw a timeout error.

The test controls an input that normally comes from another component.

A fake provides a working replacement

```
class InMemoryShipments {
  items: Shipment[] = []

  async save(shipment: Shipment) {
    this.items.push(shipment)
  }
}

const shipments = new InMemoryShipments()
await createShipment(
  parcel,
  makeTestDeps({ shipments }),
)

expect(shipments.items[0].id).toBe('shp-1')
```

Provides a simplified implementation that works.

- Keeps shipment state in memory.
- Runs without PostgreSQL.
- Does not test the SQL or schema.

A spy records how a collaborator was called

```
const mailer = {
  async send(_to: string, _message: string) {},
}
const sendSpy = vi.spyOn(mailer, 'send')

await createShipment(
  parcel,
  makeTestDeps({ mailer }),
)

expect(sendSpy).toHaveBeenCalledWith(
  'alex@example.com',
  'Shipment created',
)
```

Records calls and arguments.

- No email is sent.
- The assertion inspects the recorded call instead.
- The test can observe an action that the function does not return.

A mock requires a specific interaction

```
const carrierMock = {
  book: vi.fn().mockResolvedValue('trk-123'),
}

await dispatchShipment(
  shipment,
  { carrier: carrierMock },
)

expect(carrierMock.book).toHaveBeenCalledOnce()
expect(carrierMock.book).toHaveBeenCalledWith({
  id: 'shp-1',
  service: 'ground',
})
```

Defines a required interaction.

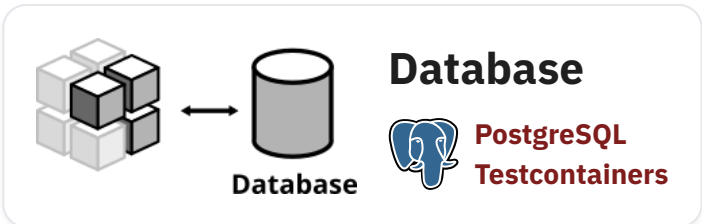
- **Expected:** One correctly formed booking request.
- **Failure:** That request does not occur.
- One `vi.fn()` may act as a stub, spy, or mock.

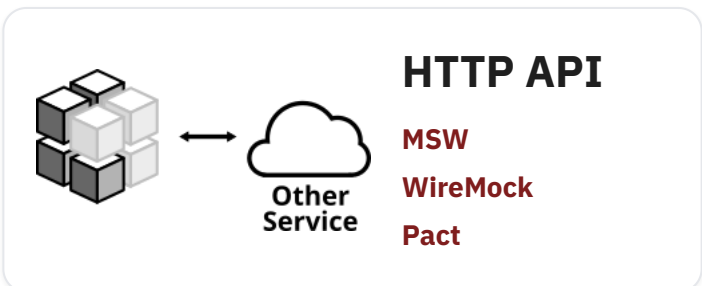
Test doubles give us control, but every replacement is more test code to write and maintain.

Integration Tests exercise part of your system against real external collaborators

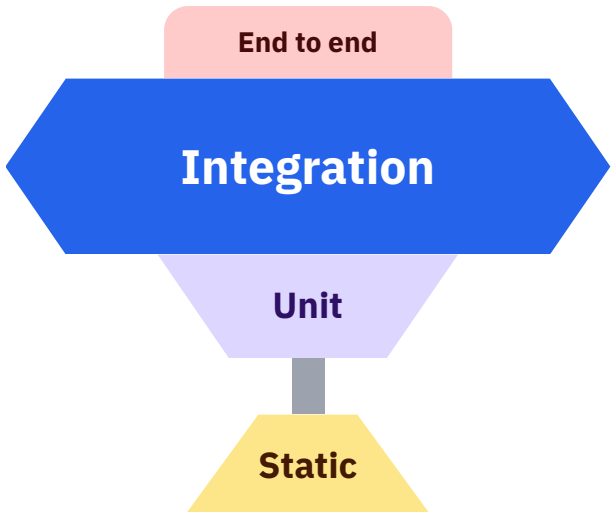
Code is tested at integrations points

- Does our REST API work?
- Can we write data to the database and read it back?
- Can we save files to disk and retrieve them?
- Do external API calls work, and do we parse their responses correctly?





The testing trophy



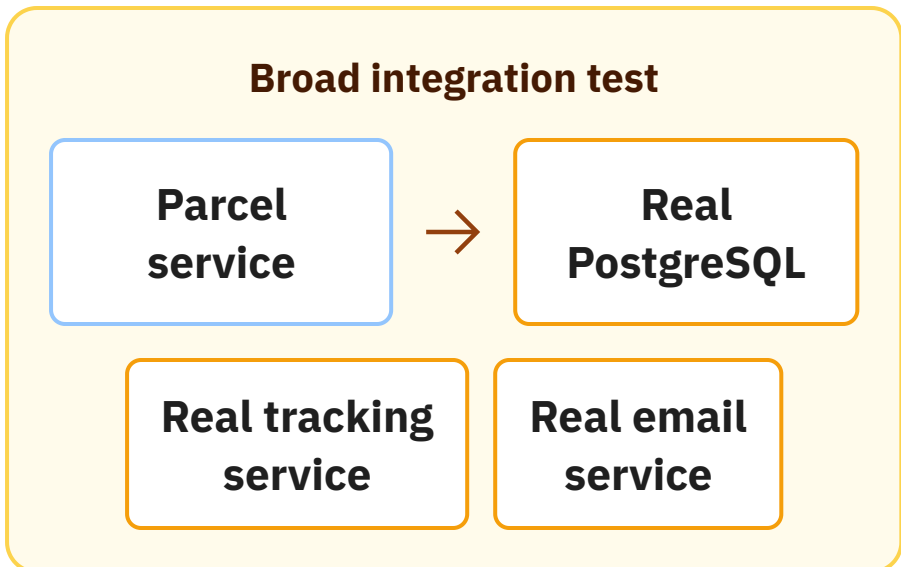


> If my high level tests are fast, reliable, and cheap to modify — then lower-level tests aren't needed.

MARTIN FOWLER

Martin Fowler · Test Pyramid

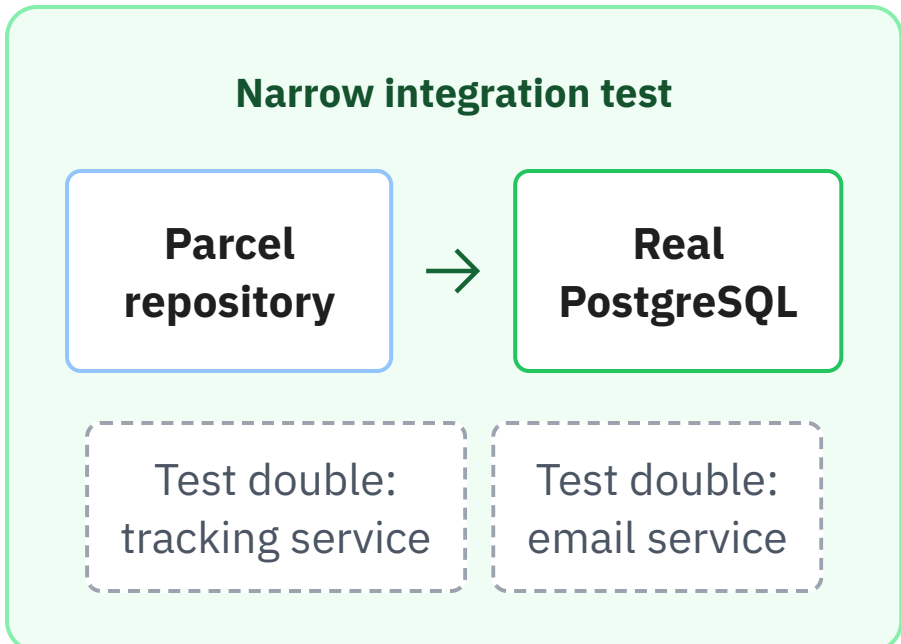
Prefer narrow integration tests over broad ones



More real parts expand the test's scope.

- Failures are harder to isolate
- Tests run more slowly
- More dependencies can cause flaky failures

Prefer narrow integration tests over broad ones



Keep one important boundary real.

- Better test isolation
- Faster feedback
- Fewer sources of flaky failures

What issues might be hard to find with tests?



Static Analysis

What can we find without running the code?

Static tools can find some problems without running the code



No application startup. No test input. Results can appear in the editor or CI.

Each static tool checks a different kind of problem



Format

Prettier · Black · `gofmt`



Lint

ESLint · Ruff · Stylelint



Types

TypeScript · mypy



Configuration

JSON Schema · Ajv



Code + data flow

CodeQL · Semgrep

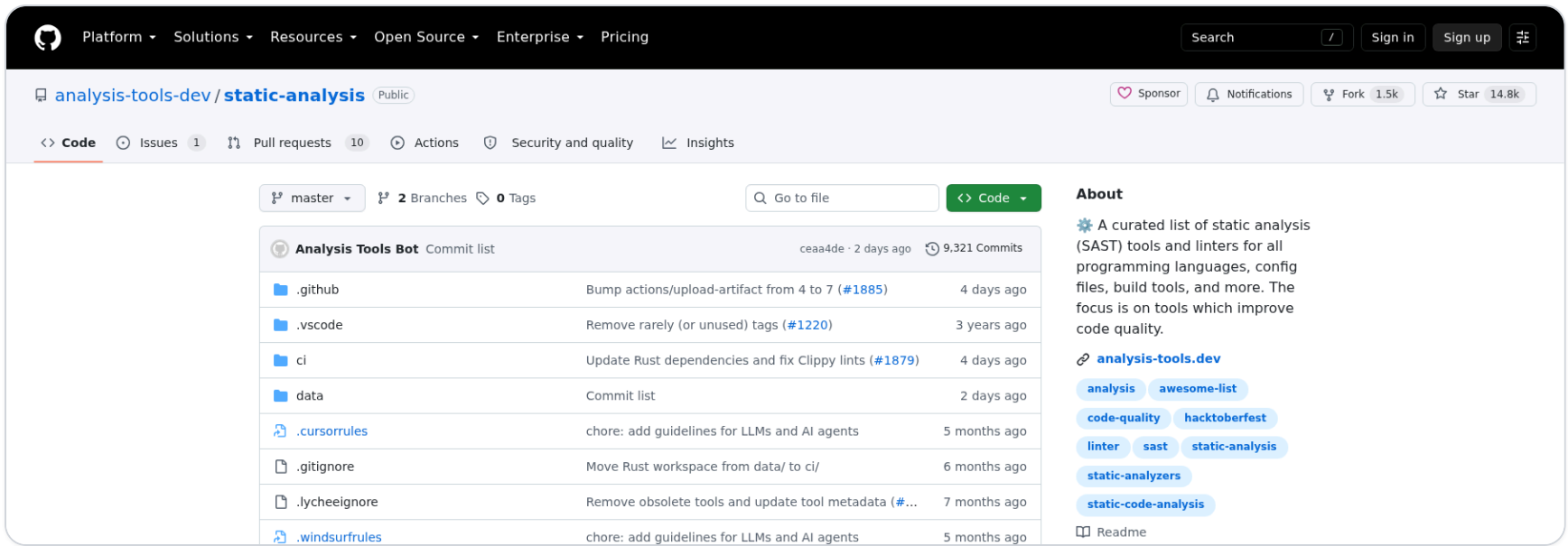


Dependencies

Dependabot · Renovate

Run fast checks while coding, then run required checks again in CI.

Let’s take a look at some tools



GitHub: Analysis Tools static-analysis directory

Formatters make style checks automatic



Prettier

Black · `gofmt`

```
$ prettier --check src/rates.ts
WARN  src/rates.ts does not match the agreed format
```

- Rewrites spacing, wrapping, and punctuation using fixed rules
- Keeps style discussions out of code review

Formatting says whether code has the agreed **form**, not whether it has the right **behavior**.

Linters flag suspicious patterns and policy violations



ESLint



Stylelint

```
$ eslint src
ERROR  routes.ts:42  request value used without validation
ERROR  log.ts:9      console output violates team policy
```

- Applies rules to source code without starting the service
- Can enforce built-in, plugin, or team-defined rules

A clean lint result covers only the rules that ran.

Type checkers reject operations that do not match declared types

```
function shippingPrice(weightKg: string) {
  return weightKg * 200
}
```

```
$ tsc --noEmit
error TS2362:
The left-hand side of an arithmetic
operation must be a number.
```



Note: this helps us to detect invalid operations. Even if the types match, the logic can still be wrong (e.g., 200 might be the wrong number). We can, however, use more advanced type systems to detect more defects (e.g., refinement types).

Data-flow analysis traces values across the program

```
app.get('/parcels', async (req, res) => {
  const id = req.query.id

  const rows = await db.query(
    `SELECT * FROM parcels WHERE id = '${id}'`
  )
  res.json(rows)
})
```



CodeQL

Data-flow analyzer

`req.query.id`

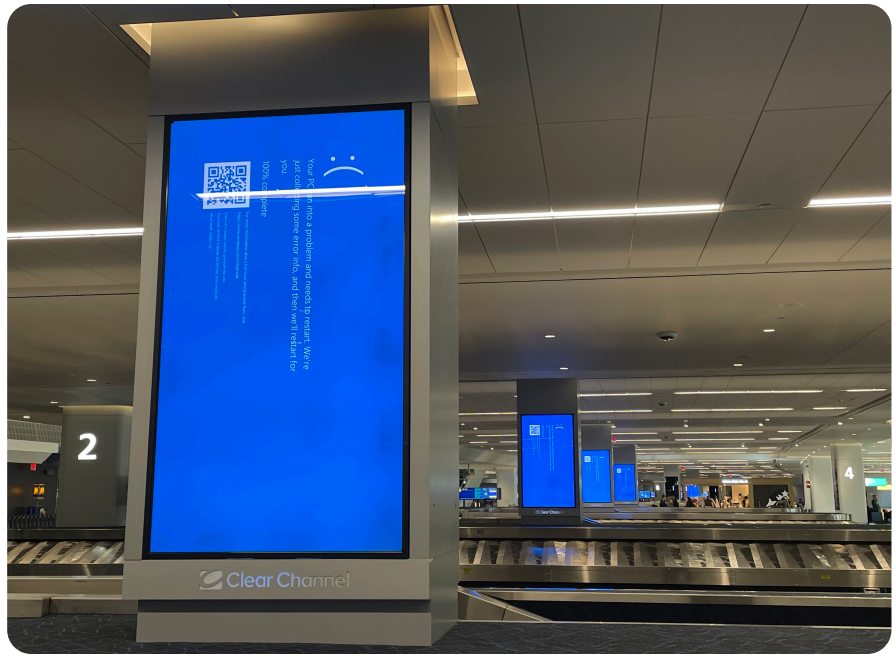
untrusted input



`db.query(...)`

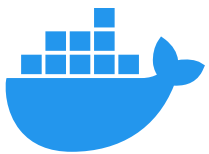
SQL injection risk

Remember CrowdStrike?



LaGuardia Airport, July 19, 2024 · Smishra1 / Wikimedia Commons (CC BY-SA 4.0)

Static tools inspect infrastructure and configuration



Dockerfile

Hadolint

Base image uses an unpinned tag



Terraform

TFLint

Configuration violates a provider rule



Kubernetes YAML

KubeLinter

Container can run as root



Shell scripts

ShellCheck

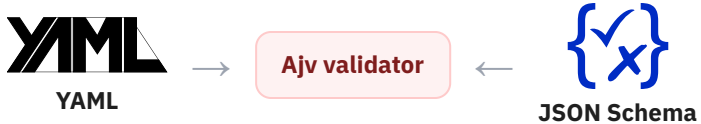
Unquoted variable may split arguments

Inspect every artifact that defines how the system will be built and run.

A schema turns project-specific configuration rules into a check

```yaml
# parcel.config.yaml
timeoutSeconds: -5
carrier: "ground"
```

```
$ ajv validate
/timeoutSeconds must be >= 1
```




```json
// parcel.schema.json
{
  "type": "object",
  "properties": {
    "timeoutSeconds": {
      "type": "integer",
      "minimum": 1
    }
  },
  "required": ["timeoutSeconds"]
}
```

- Schema defines what fields, types and values are valid

Dependency checks inspect recorded versions before code runs



Manifest + lockfile

Requested and resolved versions





Dependabot

Compares versions with releases and advisories



Security: known vulnerability

Update: newer allowed version

The check finds recorded dependency risk without starting the application.

Static-analysis warnings must be accurate and useful, or developers will ignore them

	Real problem	No real problem
Tool warns	**True positive *** Warning about a real problem	**False positive** Warning when no problem exists
Tool is silent	**False negative** Real problem, but no warning	**True negative** No problem and no warning

Even a correct warning may be too minor, unclear, or difficult to fix.

Source: Sadowski et al., “Lessons from Building Static Analysis Tools at Google”

Where could a static check help Express?



Reference repository: Express

When checks should run in CI?

CI Triggers

When should each check run?

Triggers can come from code, time, people, or other workflows

Trigger	Example
Push	Test a commit pushed to `main`
Pull-request activity	Test when a pull request opens or changes
Scheduled time	Run the full integration suite every night
Manual request	Run diagnostics on demand while investigating a failure
Workflow completion	Publish reports after the test workflow finishes

GitHub Docs · Events that trigger workflows

More checks can find more problems, but every run takes time and money

```yaml
name: Parcel API CI

on:                    # triggers
  pull_request:
  push:
    branches: [main]
```

Trigger more often	Trigger less often
Problems appear earlier	Problems wait longer
More runner time ($$$)	Less runner time ($)
More chances to wait in a queue	Fewer runs compete for capacity

The choice balances **earlier detection**, **compute cost**, and **wait time**.

Scheduled triggers run work at a chosen time

```yaml
on:
  schedule:
    - cron: "0 6 * * *"  # nightly
```

Examples:

- Publish the course website at midnight
- Run the full test suite overnight
- Scan dependencies for known vulnerabilities each week
- Remove expired preview environments every morning

Manual triggers let a person start work on demand

```yaml
on:
  workflow_dispatch:
```

Examples:

- Redeploy after external data changes
- Run diagnostics while investigating a failure
- Publish a release to a chosen environment
- Rebuild a cache after correcting its source data

What versions do we test against? What platform do we use?

A build matrix creates one job for every selected combination

```yaml
jobs:
  test:
    strategy:
      matrix:
        os: [ubuntu-latest, macos-latest]
        library: [4, 5]
    runs-on: ${{ matrix.os }}
    steps:
      - run: test-library-${{ matrix.library }}
```

2 PLATFORMS × 2 VERSIONS

Ubuntu + library 4	Ubuntu + library 5
macOS + library 4	macOS + library 5

4 test jobs

More combinations increase confidence but consume more runner time!

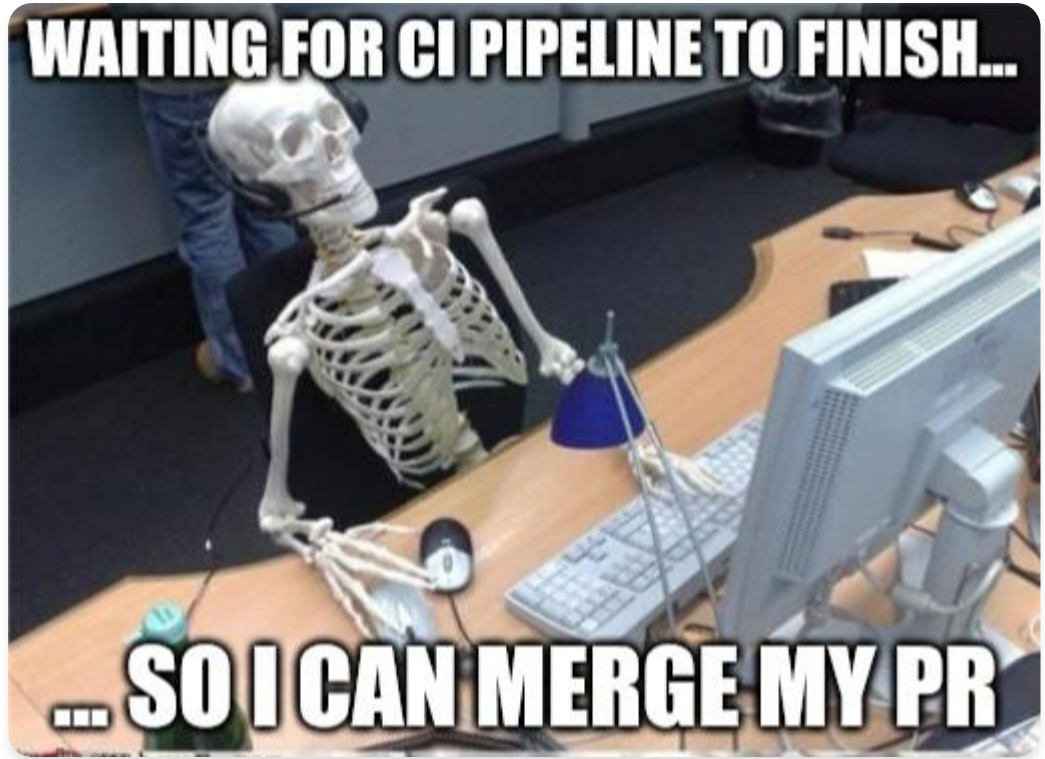


Image: ProgrammerHumor

Pipeline Health

How do we recognize and repair an unhealthy pipeline?

Pipelines lose trust when feedback is slow, unreliable, unclear, or too costly

Slow

Results arrive after the developer has moved on.

Unreliable

Unchanged code produces inconsistent results.

Unclear

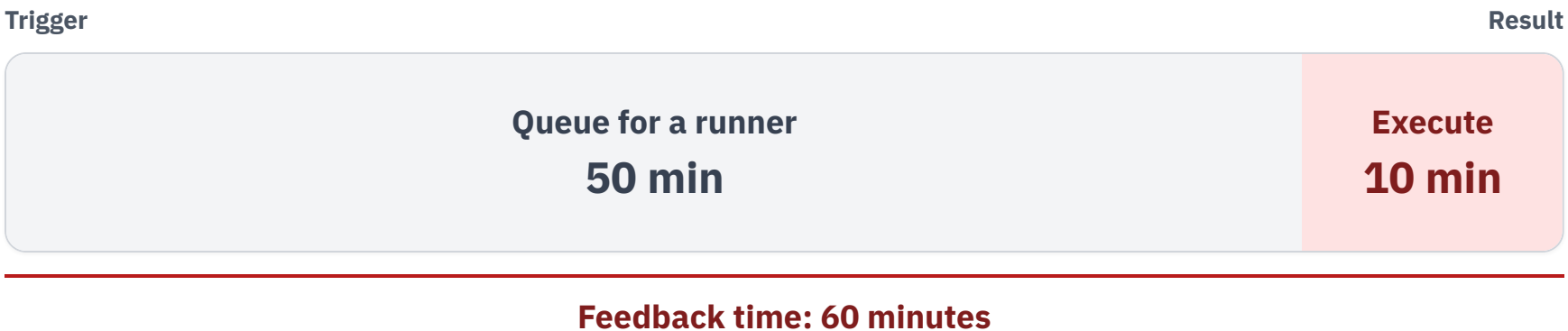
Failures do not identify the cause or next action.

Too costly

Checks consume excessive compute and developer time.

Teams either wait to merge and **lose velocity,** or bypass CI and **lose reliability.**

A ten-minute test suite can still make a developer wait an hour



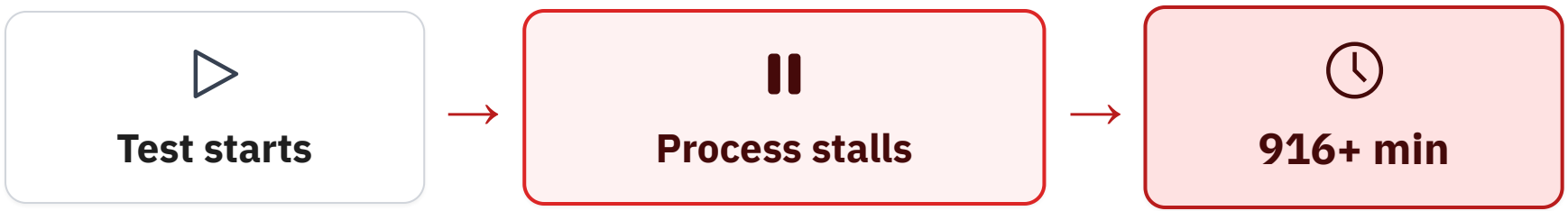
One slow job can stall the entire pipeline



The **critical path** is the longest chain of dependent jobs.
It sets the earliest possible completion time.

Change	Pipeline time	Effect
Unit: 14 → 8 min	19 → 14 min	Shortens the critical path
Integration: 9 → 2 min	19 → 19 min	Saves compute, but not feedback time

A job that never exits can consume an entire CI budget



Use timeouts to turn an unbounded failure into a bounded one.

False failures teach developers to rerun

Same commit

FAIL

→

RERUN

Same commit

PASS

A **flaky test** passes or fails without a relevant code change.

Repeated false failures make “rerun” feel safer than “investigate.”

Flakiness comes from uncontrolled variation



Timing
Races and slow responses



Shared state
Test order and leaked data



Dependencies
Network and service instability



Changing inputs
Time, randomness, and locale

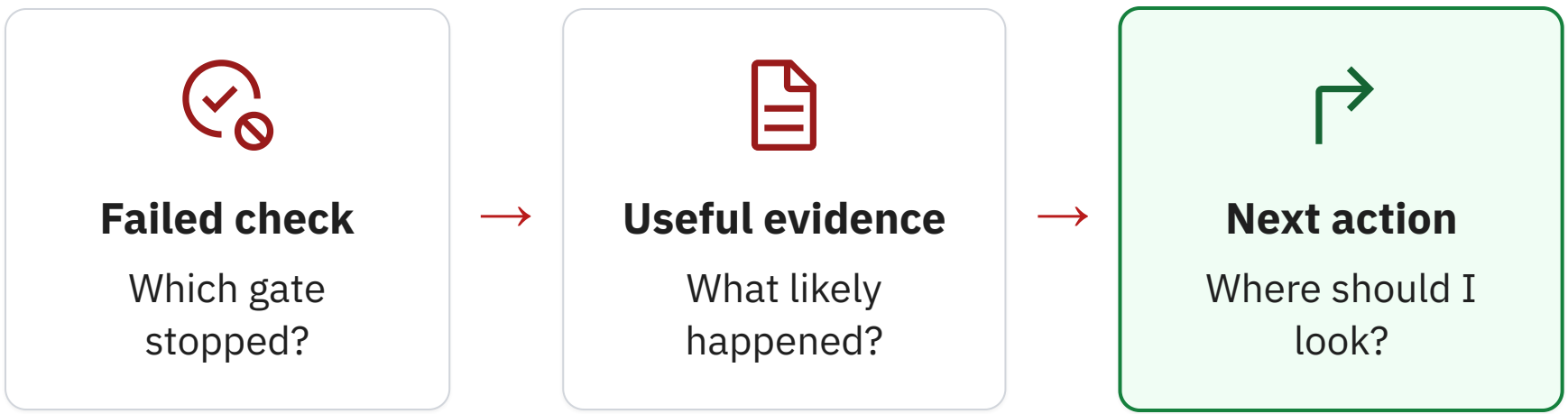
Vague failure messages turn diagnosis into more CI work

```
$ ci verify
ERROR: job failed
Process exited with code 1.
See logs for details.
```

WHAT DOES IT MEAN?

- **Test assertion**
- **Configuration**
- **Dependency**
- **Infrastructure**

Actionable failure messages help developers to figure out the next step

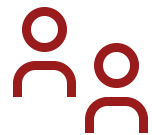


Unhealthy CI consumes two resources



Direct CI cost

- runner time for jobs and retries
- artifact and cache storage
- third-party service usage



Developer cost

- waiting and context switching
- investigating false or unclear failures
- maintaining pipeline infrastructure

Every rerun spends both. As the system grows, both costs compound.

The bottleneck may be inside the code



Profile the slow path
Find the tests and application code that dominate execution time.



Remove unnecessary work
Simplify repeated setup, data preparation, and computation.



Isolate state and dependencies
Control shared data, time, randomness, and external services.

The bottleneck may be in pipeline design



Maintain the CI machinery
Improve custom scripts, tools, and configuration.



Reuse work carefully
Cache an output only when all of its inputs match.



Reorganize the jobs
Select, combine, split, or rebalance work along the critical path.

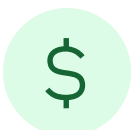
Infrastructure can bound delay and waste



Size and schedule runner capacity
Add capacity when jobs wait; schedule expensive work when urgency is lower.



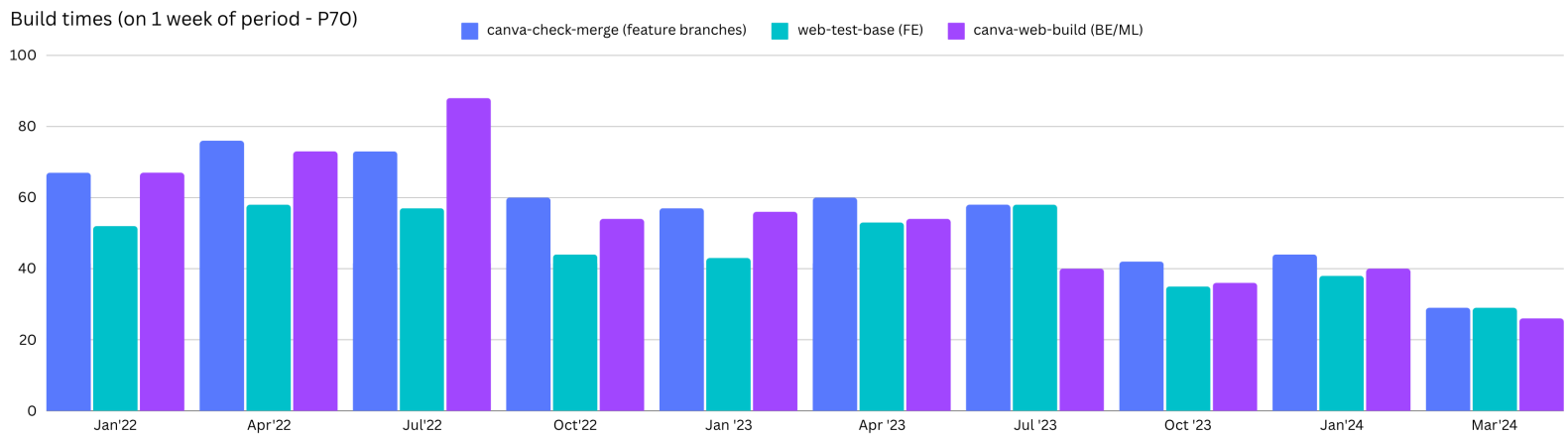
Set timeouts and retry limits
Turn unbounded failures into bounded ones.



Set compute and storage quotas
Limit how much one workflow can consume.

One bottleneck may require changes to code, pipeline design, and infrastructure.

Measure first to find the bottleneck



Weekly P70 build-time trend, January 2022–March 2024

Canva: about 80 minutes → under 30 minutes

Canva Engineering · Faster continuous integration builds

Keeping CI Fast as Work Grows

How do we preserve fast feedback at scale?

Large organizations assign teams to improve CI

GitHub

Developer Experience and GitHub Actions teams

15,000
jobs in a peak hour

50 → 12 min
for some workflows

Source: GitHub Engineering, "How GitHub uses GitHub Actions..."

Spotify

Platform and developer tools teams

200+
Jenkins machines replaced

tens of thousands
of builds each week

Source: Spotify Engineering, "How We Improved Developer Productivity..."

Small savings add up across many builds and developers.

Run the most useful checks first

BEFORE MERGE

Fast checks for this change

- lint + static analysis
- relevant unit tests
- required integration checks

Help the team decide whether to merge

merge

AFTER MERGE OR SCHEDULED

Broader system checks

- cross-service tests
- operating system and version tests
- expensive end-to-end tests

Cover more cases without delaying every merge

CI can reuse downloaded packages across runs

```yaml
- uses: actions/cache@v4
  with:
    path: ~/.npm
    key: npm-${{ hashFiles('package-lock.json') }}

- run: npm ci
- run: npm test
```

Cache hit
Restore downloaded packages.

Cache miss
Download and save packages.

`npm ci` and `npm test` run either way.

`actions/cache`

Looks for saved files before the next step. After a miss, it saves them for a later run.

`path`

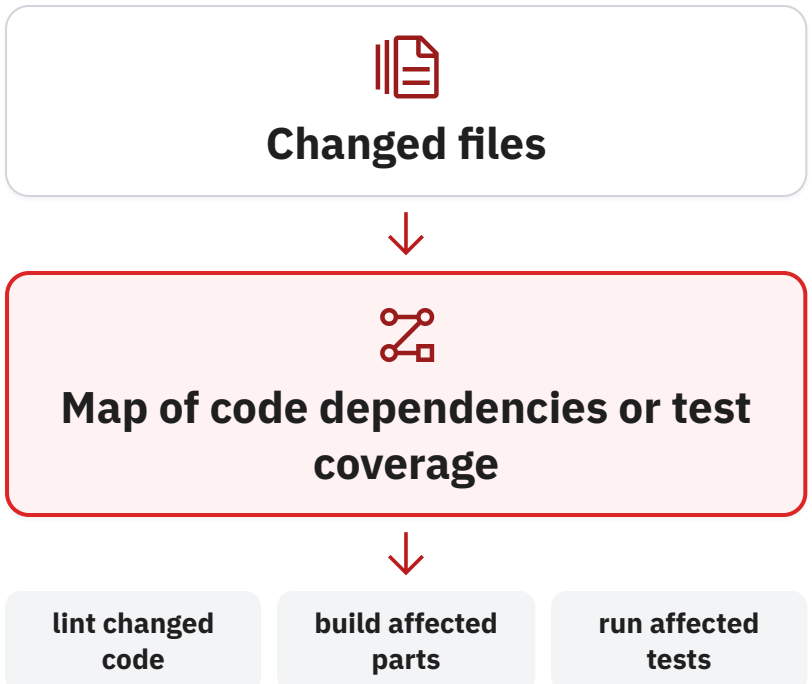
Selects the files to save and restore. Here, `~/.npm` is npm's download cache.

`key`

Names the saved files. `hashFiles(...)` changes the name when the lockfile changes.

Every input to the cached files must affect the key.

Run only the checks that a change could affect



SELECT BY FILE PATH IN GITHUB ACTIONS

```yaml
on:
  pull_request:
    paths:
      - 'web/**'
      - 'package-lock.json'
```

Test Impact Analysis selects individual tests



CI runs the highlighted tests that previously exercised the changed code.

⚠️ Skipping work can hide failures



Incomplete cache key

An input changes, but CI reuses output created from old inputs.



Missing dependency link

An affected part appears unrelated, so CI skips its checks.



Outdated test map

CI chooses checks using an older system structure or older coverage data.

Quality Gates and Code Review

Which evidence should block a merge, and what requires human review?

Should we merge this pull request?

AUTOMATED CHECKS

Check clear rules

Find known problems early.

→

HUMAN REVIEW

Review the context

Decide what requires judgment.

→

MERGE DECISION

Decide whether to merge

Consider both safety and speed.

Automated checks inform the decision; people are responsible for it.

Automated checks should save reviewers' time

A **quality gate** is an agreed rule that must pass before a change can merge.

Review input	How the team uses it
Required automated check	Stops a change that clearly breaks an agreed rule
Optional automated check	Warns reviewers without stopping the merge
Human review	Decides questions that require judgment

Only reliable checks should stop a merge

A check should stop a merge only when it:

- protects against a specific risk
- gives the same result for the same code
- has a team or person responsible for maintaining it
- tells developers what to do when it fails

Coverage thresholds track coverage, not correctness

A **coverage threshold** sets the minimum measured coverage needed to pass.

It can warn that:

- new code was not exercised by tests
- measured coverage decreased

Even 100% line coverage can miss:

- a test that checks the wrong requirement
- a test that runs code without checking the correct result

AI review tools can examine a pull request



GitHub Copilot

Reviews when requested or automatically

Comments and suggested changes



CodeRabbit

Uses AI models and code-analysis tools

Issue reports and suggested fixes



DoorDash reviewer

Scans first, then checks likely issues

Comments tied to specific lines

Code changes + context + rules → **Possible issues** → **A person checks each issue**

AI comments can inform a decision; they do not approve the change.

Which review tasks should AI handle? Which require people?



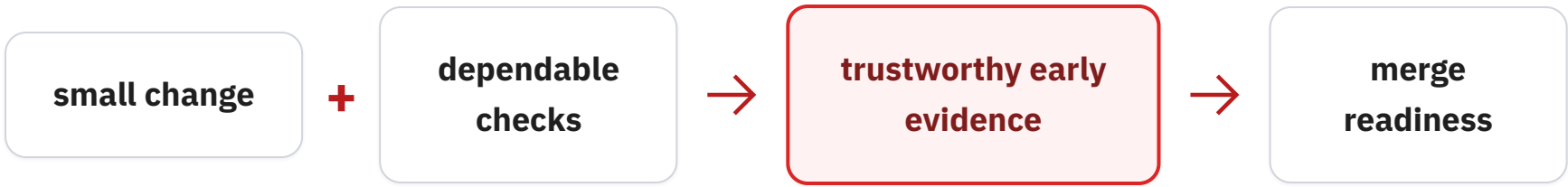
People make the final merge decision

Reviewers ask:

- What are the important qualities to check?
- Does this solve the right problem?
- Does it fit the system's design?
- Will future developers be able to understand and change it?

Automated checks provide information. People decide whether to merge.

Reliable CI helps keep small changes ready to release



That evidence makes frequent releases easier, but it does not prove a release is safe.

Still required: human review, deployment checks, and production monitoring.

🚀 Today, we learned how to ...

- Explain how continuous integration and shifting left keep the main branch working and make problems cheaper to fix.
- Choose the earliest credible automated check for a risk using the test pyramid.
- Decide which CI results should block a merge and which decisions need human review.
- Compare CI triggers by coverage, cost, and feedback time.
- Diagnose slow, flaky, or noisy CI and choose a matching improvement.