CI & Build Systems

CMSC398W: Practical Tools For Efficient Development

Mohammad Durrani

June 22, 2026

What Happens When There's No CI?

Three developers, one shared repo, pushing to `main` all day.

- Mohammad pushes a fix Friday at noon. Tests pass on his laptop.
- Karthik pushes a feature Friday at 3pm. Tests pass on his laptop.
- Lilli try to deploy Friday at 5pm. It fails.

Whose commit broke it? You have 23 changed files across two pushes.

What Is CI?

Definition

Continuous Integration: every commit triggers an automated sequence of checks before code merges.

- Short feedback loops
- When the build breaks you finds out immediately, not three commits later
- Failed checks block the merge before broken code spreads

What Is CI?

Definition

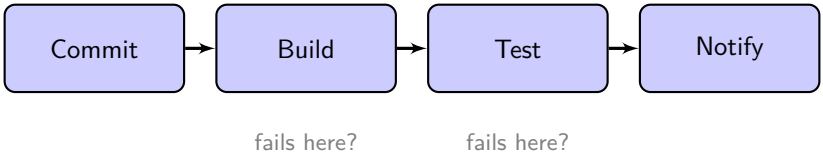
Continuous Integration: every commit triggers an automated sequence of checks before code merges.

- Short feedback loops
- When the build breaks you finds out immediately, not three commits later
- Failed checks block the merge before broken code spreads

Why Bother?

Without it, you find out code is broken when someone tries to deploy, not when the bad commit landed.

The CI Workflow



Each stage gates the next. If the build fails, the tests don't run.

GitHub Actions

GitHub Actions is GitHub's built-in CI service. Add a YAML file in `.github/workflows/` and GitHub runs it on their servers every time something happens to your repo.

- Events trigger the pipeline (`push`, `pull_request`, `schedule`)
- Jobs are the units of work, each running on its own machine
- Steps are commands inside a job, run top to bottom

GitHub Actions

GitHub Actions is GitHub's built-in CI service. Add a YAML file in `.github/workflows/` and GitHub runs it on their servers every time something happens to your repo.

- Events trigger the pipeline (`push`, `pull_request`, `schedule`)
- Jobs are the units of work, each running on its own machine
- Steps are commands inside a job, run top to bottom

Jobs Form a DAG

Job B can declare `needs: [A]`, meaning it won't start until A finishes. Jobs with no `needs` run in parallel.

The Blog App's Workflow File

```
name: Blog CI
on: [push, pull_request]
jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: pip install flake8 && flake8 .
  test:
    needs: lint
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: pip install -r requirements.txt
      - run: pytest
```

- `on`: what triggers the workflow
- `runs-on`: fresh VM spun up by GitHub
- `needs`: waits for `lint` to pass first
- `uses`: pre-built action (`checkout` = `git clone`)
- `steps`: run in order, top to bottom

Design Your Pipeline

The Scenario

You're on a blog app team. Three people push throughout the day. You need a CI pipeline. With your table, sketch:

1. What checks does your pipeline run?
2. What order do they run in?
3. Which failure should stop everything downstream?

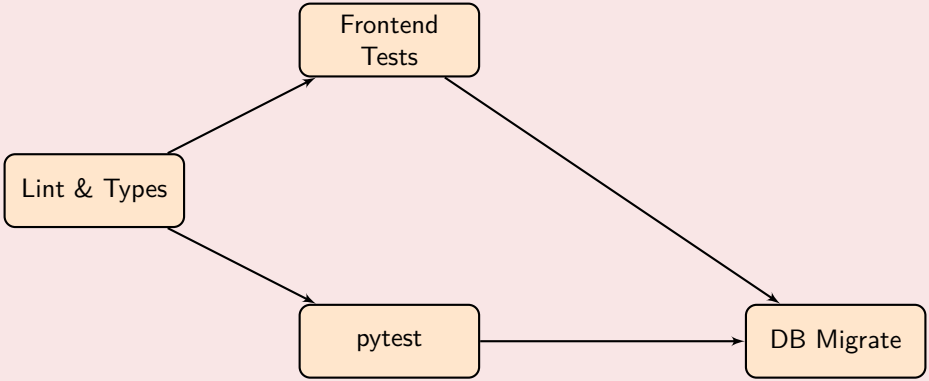
Constraints

- Python/Flask backend with `pytest` tests
- React frontend with its own test suite
- A database migration script (dangerous if run on broken code)
- Linting and type checking available for both sides

Debrief

Debrief

One Reasonable Answer



Lint first (fast, cheap). Backend and frontend tests run in parallel. Migration waits for both.

Why Not Just Shell Scripts?

Consider This

You change one CSS file. You run `./build.sh`. What happens?

- Recompiles the Python package (Python didn't change)
- Re-runs the entire test suite (no logic changed)
- Rebuilds the frontend bundle (this one is actually needed)
- Re-runs the linter on every file, every time

Why Not Just Shell Scripts?

Consider This

You change one CSS file. You run `./build.sh`. What happens?

- Recompiles the Python package (Python didn't change)
- Re-runs the entire test suite (no logic changed)
- Rebuilds the frontend bundle (this one is actually needed)
- Re-runs the linter on every file, every time

The Problem

Problem	Consequence
No dependency tracking	Everything rebuilds on every change
No caching	Slow feedback loops
Order is hardcoded	Can't parallelize safely

What Build Systems Do

The Core Abstraction

A DAG (Directed Acyclic Graph) is a graph where edges point one way and there are no cycles. Build systems represent your work as one.

- A node = a task (compile, lint, bundle, test)
- An edge = "B requires A to finish first"
- The build system figures out the right order and runs tasks in parallel where possible

What Build Systems Do

The Core Abstraction

A DAG (Directed Acyclic Graph) is a graph where edges point one way and there are no cycles. Build systems represent your work as one.

- A node = a task (compile, lint, bundle, test)
- An edge = "B requires A to finish first"
- The build system figures out the right order and runs tasks in parallel where possible

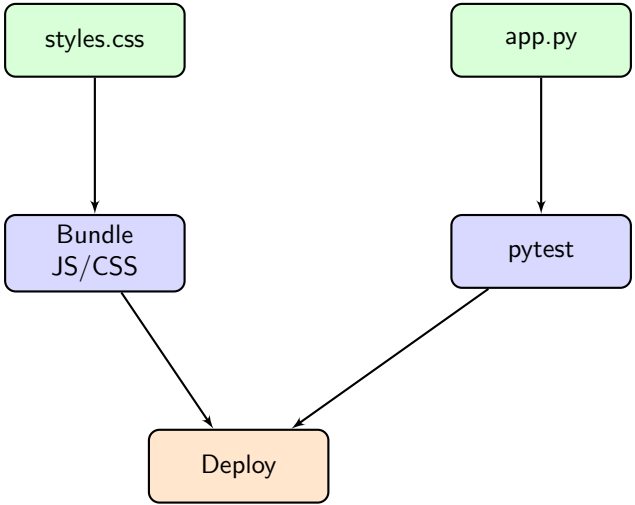
Incremental Builds

Change a file → only the tasks that depend on it re-run.

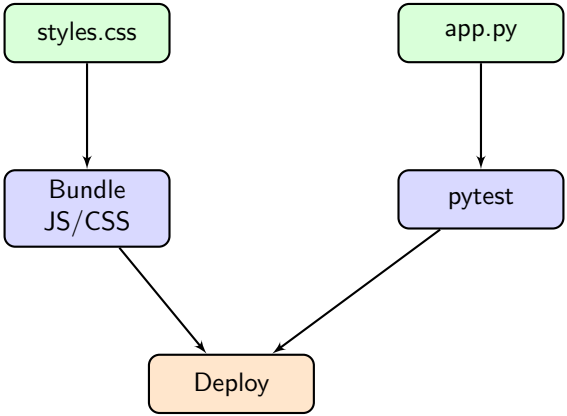
Property	What it means
Correctness	Nothing is skipped that shouldn't be
Speed	Unchanged outputs come from cache
Reproducibility	Same inputs always produce the same outputs

The Blog App's Build Graph

Source files (green) feed into tasks (blue), which feed into the final piece.



What Re-Runs?



Scenarios

Scenario A: You change `styles.css` Scenario B: You change `app.py`

Which nodes re-run? Which come from cache?

Activity 2: Answers

Activity 2: Answers

Scenario A: Change `styles.css`

Re-runs: Bundle JS/CSS → Deploy

`pytest` is cached. Python code is unchanged.

Activity 2: Answers

Scenario A: Change `styles.css`

Re-runs: Bundle JS/CSS → Deploy
`pytest` is cached. Python code is unchanged.

Scenario B: Change `app.py`

Re-runs: pytest → Deploy
Bundle is cached. Frontend is unchanged.

At Google Scale

Bazel

The blog app's graph has six nodes. Google's internal build graph has millions. Running everything on every commit would take days. Bazel is Google's open-source build system that has a similar DAG model and uses distributed caching across thousands of machines. This enables Googles operations at scale

The Problem

Your Code Depends on Code You Don't Control

The blog app pulls in:

Package	Version
`flask`	`3.0.1`
`react`	`18.2.0`
`pytest`	`7.4.0`

Eventually, this softare will have to be updated. Sometimes, updates break things.

The Problem

Your Code Depends on Code You Don't Control

The blog app pulls in:

Package	Version
`flask`	`3.0.1`
`react`	`18.2.0`
`pytest`	`7.4.0`

Eventually, this softare will have to be updated. Sometimes, updates break things.

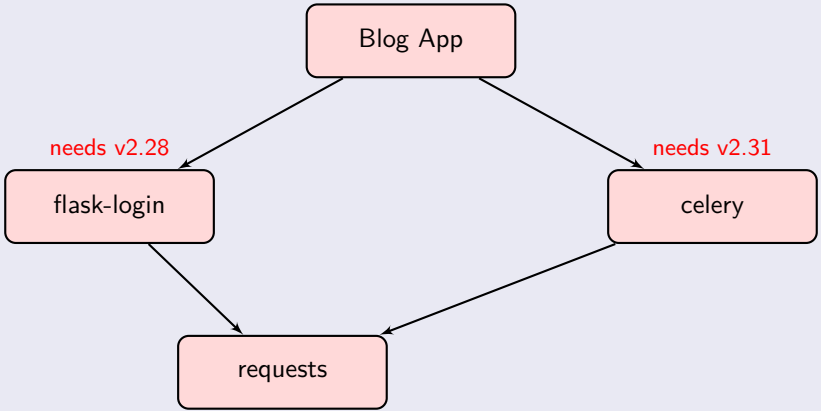
The Dependency Graph

Running `pip install flask` actually installs five packages. `flask` pulls in `werkzeug`, `werkzeug` pulls in `click`, and so on.

Dependency Hell

Diamond Dependencies

The blog app uses `flask-login` for auth and `celery` for background tasks. Both need `requests`, but different versions.



Semantic Versioning

The Convention: MAJOR.MINOR.PATCH

`react@18.2.0`:

Part	Number	Meaning
MAJOR	18	Breaking change; existing code may stop working
MINOR	2	New features, backwards compatible
PATCH	0	Bug fixes only

Semantic Versioning

The Convention: MAJOR.MINOR.PATCH

`react@18.2.0`:

Part	Number	Meaning
MAJOR	18	Breaking change; existing code may stop working
MINOR	2	New features, backwards compatible
PATCH	0	Bug fixes only

Range Specifiers

`^18.2.0` in `package.json`: give me anything $\geq$ 18.2.0 but $<$ 19.0.0

`~3.0.1` in `requirements.txt`: give me anything $\geq$ 3.0.1 but $<$ 3.1.0

Activity 3: SemVer Cases

The Flask Scenario

You maintain `flask`. For each change, vote: MAJOR, MINOR, or PATCH?

1. Fix a crash when the request body is empty
2. Add a new `app.config.from_prefixed_env()` helper
3. Remove the deprecated `before_first_request` decorator
4. Rename an internal method… that 10,000 projects call anyway

Vote

Try and back up your answer with reasoning

Activity 3: Answers

Activity 3: Answers

Cases 1-3

Case	Answer	Reason
Empty body crash fix	PATCH	Bug fix, no API change
New config helper	MINOR	New feature, nothing breaks
Remove `before_first_request`	MAJOR	Deleting a public API

Activity 3: Answers

Cases 1-3

Case	Answer	Reason
Empty body crash fix	PATCH	Bug fix, no API change
New config helper	MINOR	New feature, nothing breaks
Remove `before_first_request`	MAJOR	Deleting a public API

Case 4: Hyrum's Law

"With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody." Hyrum Wright

Internal method, but 10k projects call it. PATCH or MAJOR?

Overall View

CI pipelines, build systems, and package managers all have the same shape.

System	Nodes	Edges	Goal
CI pipeline	lint, test, deploy	job dependencies	catch breakage early
Build system	source files, artifacts	file dependencies	rebuild only what changed
Package graph	libraries	version constraints	resolve compatible set

Overall View

CI pipelines, build systems, and package managers all have the same shape.

System	Nodes	Edges	Goal
CI pipeline	lint, test, deploy	job dependencies	catch breakage early
Build system	source files, artifacts	file dependencies	rebuild only what changed
Package graph	libraries	version constraints	resolve compatible set

The Common Question

What is the minimum work, in the right order, to get a reliable result?