Automate the boring stuff with CI/CD

Subtitle of Presentation

Jakob van Santen
Zeuthen, 2023-12-12

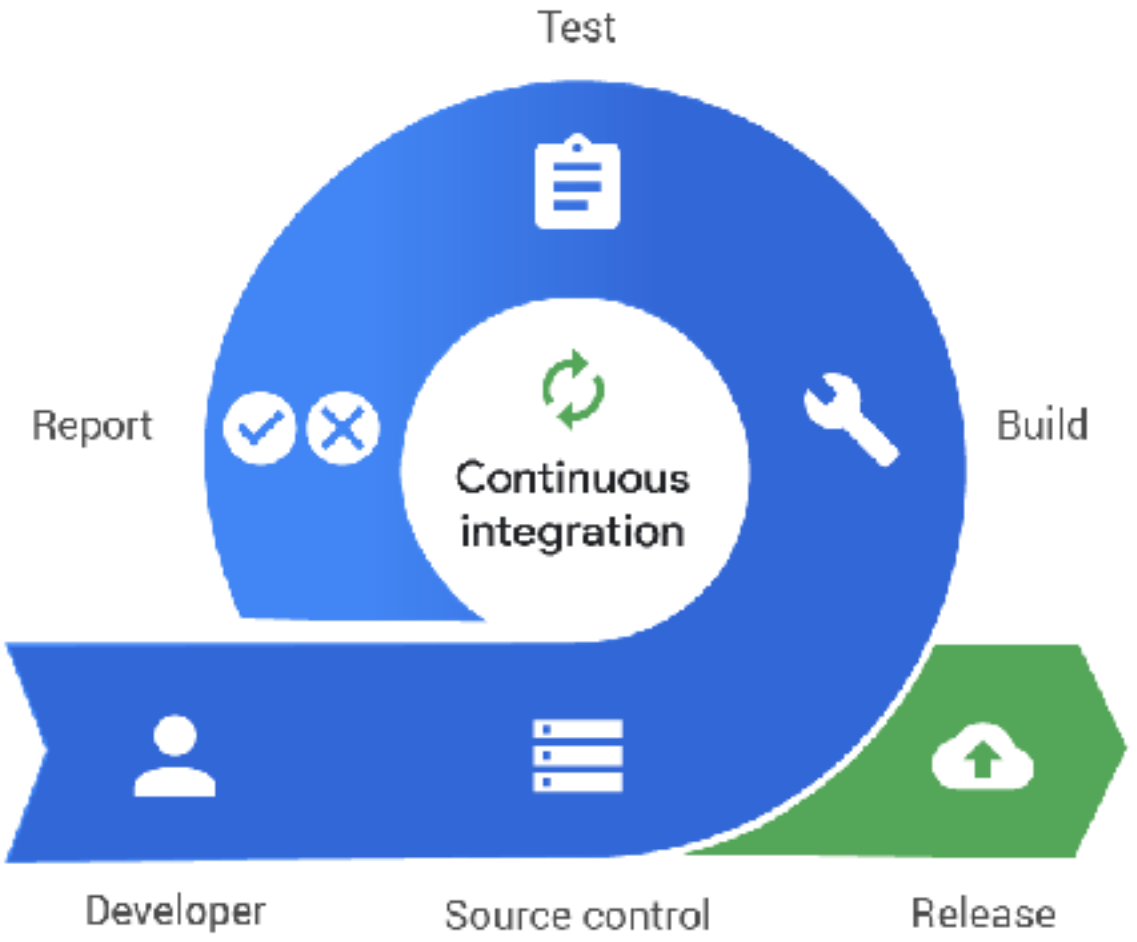
HELMHOLTZ

DESY.

Continuous Integration/Continuous Delivery

word words

- **Integration**: make changes [without breaking things]
- **Delivery**: publish a version (source code, installable packages, whatever)
- **Continuous**: do it on demand, with as little human intervention as possible
- Integrated dev platforms like GitHub and GitLab have great tools for CI/CD
- Applications for code-writing scientists:
  - Quality control
  - Packaging
  - Dependency management



https://www.pagerduty.com/resources/learn/what-is-continuous-integration/

Disclaimers

My assumptions about you

- You use an integrated development platform like GitHub or [DESY] GitLab
- You know what a pull request/merge request is
- You write code, mostly in Python
- You are not a software engineer
- You may have seen a CI pipeline, but have never written one yourself

I will cover

- Why you should automate quality control, packaging, and dependency management
- Useful tools
- Examples
- Good practices

I will not cover

- How to write good tests & testable code

A minimal CI workflow

"Run my tests every time I push"

GitHub Actions

`.github/workflows/whatever.yml`

```yaml
on:
  - push
jobs:
  test:
    runs-on: "ubuntu-22.04"
    steps:
    - uses: actions/checkout@v3
    - uses: actions/setup-python@v4
      with:
        python-version: '3.10'
    - run: |
        python -m pip install poetry
        poetry install
    - run: poetry run pytest
```

GitLab CI/CD

`.gitlab-ci.yml`

```yaml
image: "python:3.10"

test:
  script:
  - pip install poetry
  - poetry install
  - poetry run pytest
```

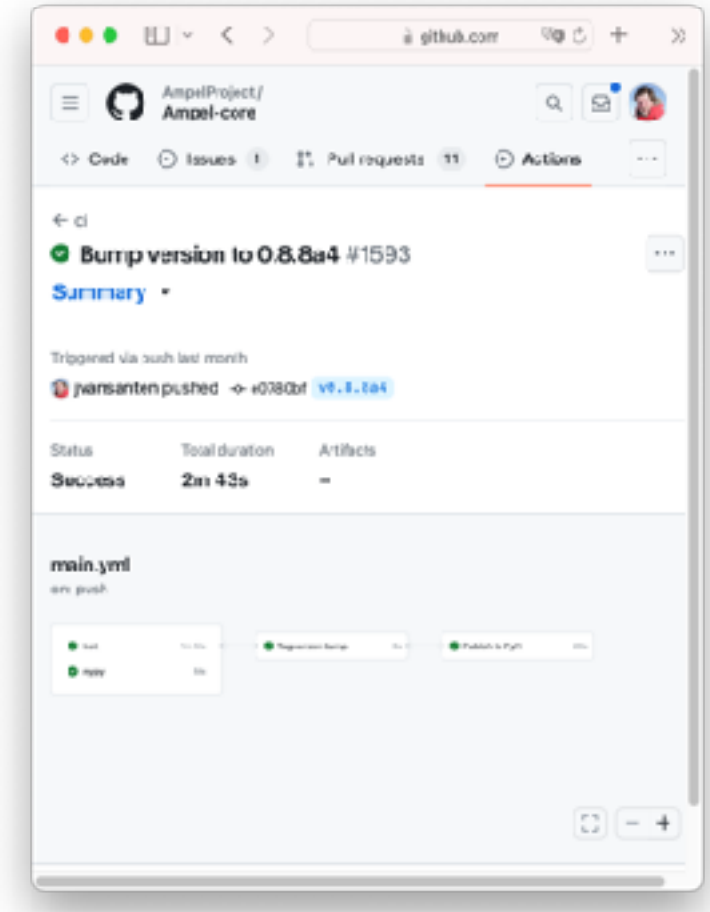
Can add more complexity:

- Run only on certain branches, PRs, etc
- Multiple jobs that depend on each other
- Conditional jobs
- Sidecar services (databases, etc)
- Caching

Boring thing 1: quality control

"Code" vs "software"

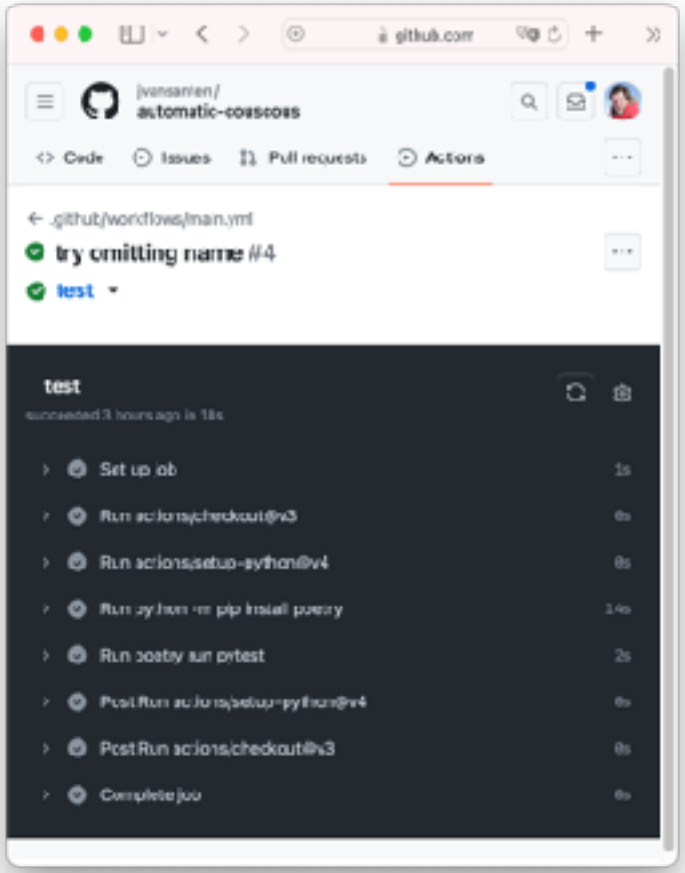
- Code is the thing you write
- Software is made from code, plus
  1. Some assurance that it behaves as advertised
  2. A version
  3. Documentation 😂
- Quality control alerts you when something breaks (1)
- Can be:
  - Something you or a contributor changed
  - One of your dependencies changed



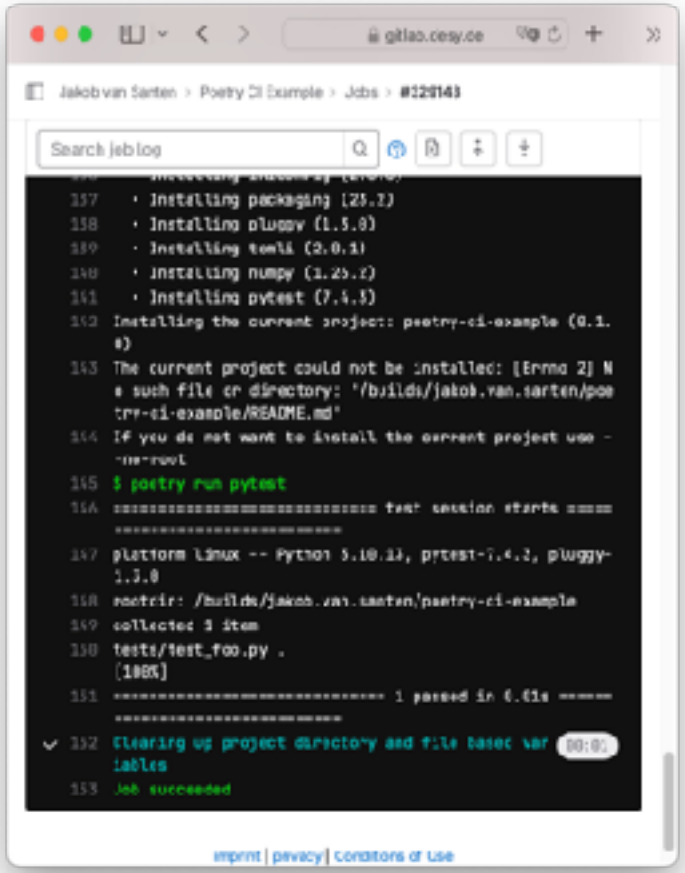
Useful tools for Python quality control

- **Formatters**: make diffs smaller
  - black: a good standard style
  - isort: keep your imports organized
- **Linters**: find common mistakes by just looking at the code
  - flake8: bundle of popular checks
  - ruff: like black+isort+flake8+plugins, but faster
- **Static type checkers**: find type errors without running the code
  - mypy: the original Python type checker
  - pyright: stricter than mypy, part of VSCode
- **Test frameworks**: simplify writing, discovering, running tests
  - pytest: compact, reusable, plugins for everything
  - unittest: comes with Python
- **Virtualenv managers**: simplify specifying & installing dependencies
  - poetry: versioning, packaging, dependency management
  - pipenv: just dependency management
  - pdm: if you want everything to work like node.js
  - conda-lock: reproducible conda environments

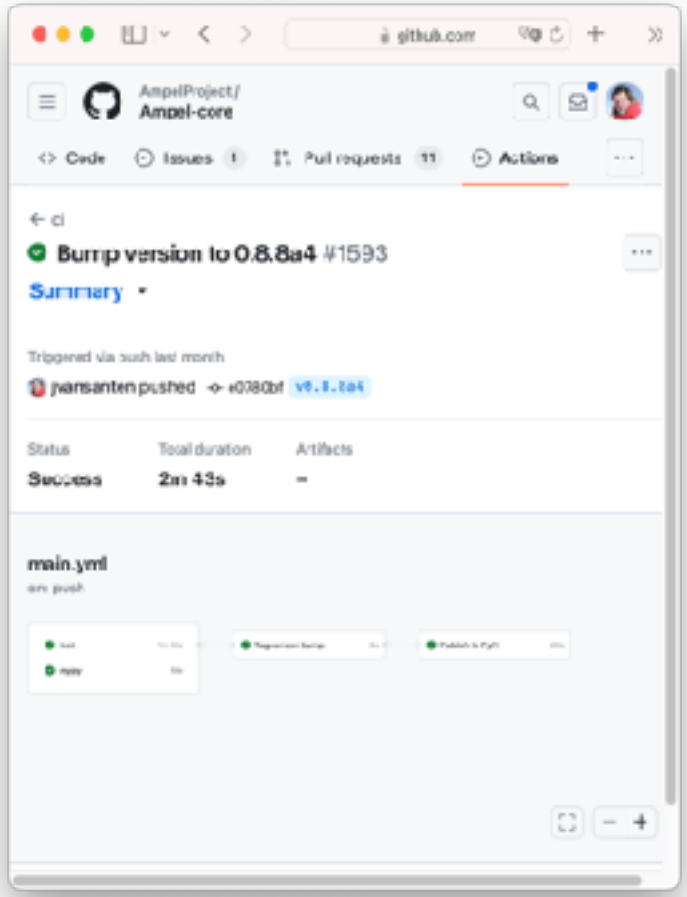
Examples



Minimal install & test
(GitHub Actions)



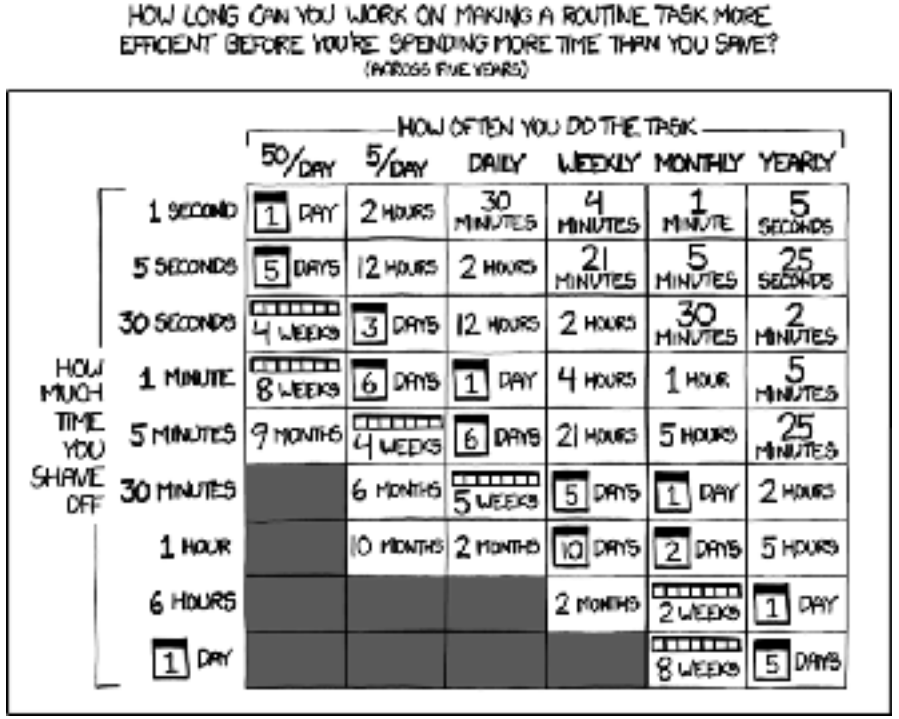
Minimal install & test
(GitLab CI)



Real-world install, lint & test
(GitHub Actions)

Good practices for quality control

- Decide how much quality control you want and when, e.g.
  - main branch always works
  - main branch may be broken, but releases will work
  - "It worked for me; use at your own risk"
- Start with a minimal set of checks, add new ones whenever you fix something
- Ensure your checks depend only on the contents of the commit
- Require checks before merging changes into a stable branch
- Avoid pushing directly to stable branches



https://xkcd.com/1205

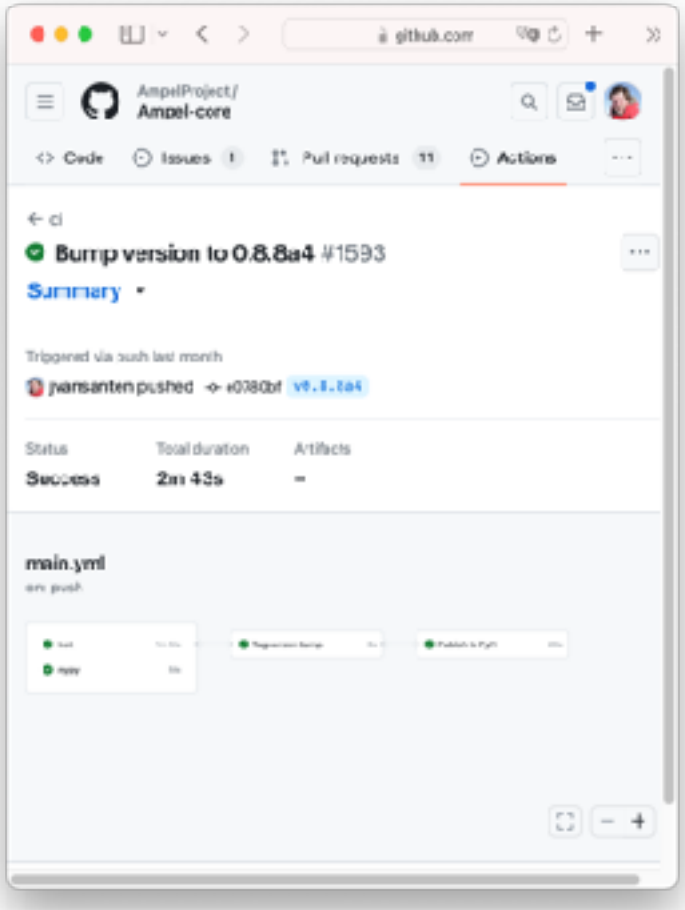
Boring thing 2: publishing packages

Be kind to your non-developer users

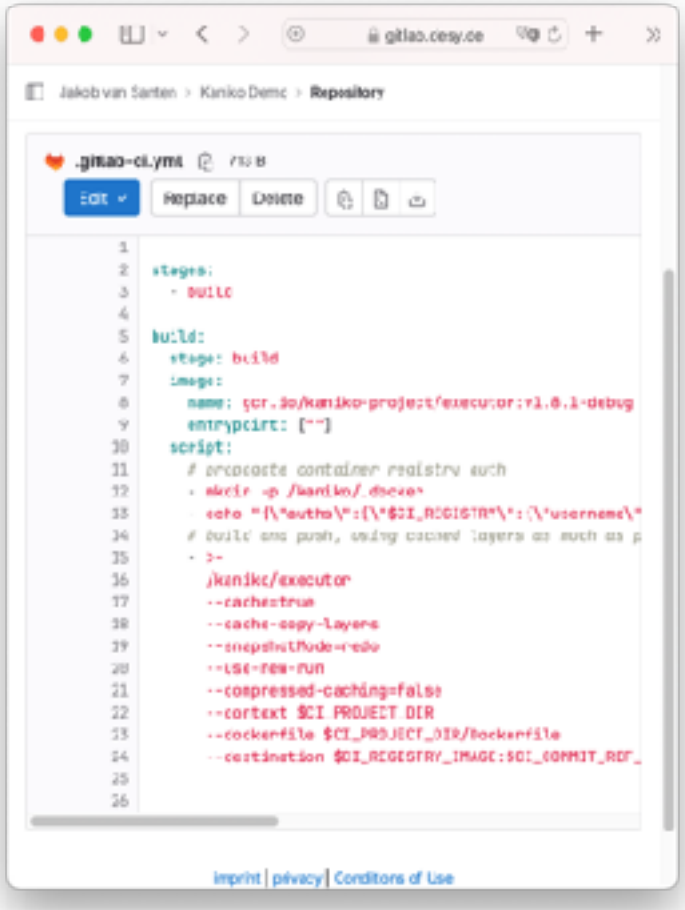
- Tag versions as you add features or fix bugs
- Publish a package for each release tag (e.g. v*)
- "pip install X=3.2" is nicer than "check out 632b0d5e"
- Automation *can* remove you as a single point of failure: anyone who can create a tag can release a new package
- Useful tools
  - poetry: versioning, packaging, dependency management
  - flit: package & publish to PyPI
  - twine: just publish to PyPI
  - cibuildwheel: build for multiple Python/OS/arch combinations

https://medium.com/@butteredwaffles/python-packages-and-modules-explained-part-1-ff304c4f19dd

Examples



Publish to PyPI (example)
(GitHub Actions)



Push to GitLab container registry
(GitLab CI)

Good practices for packaging

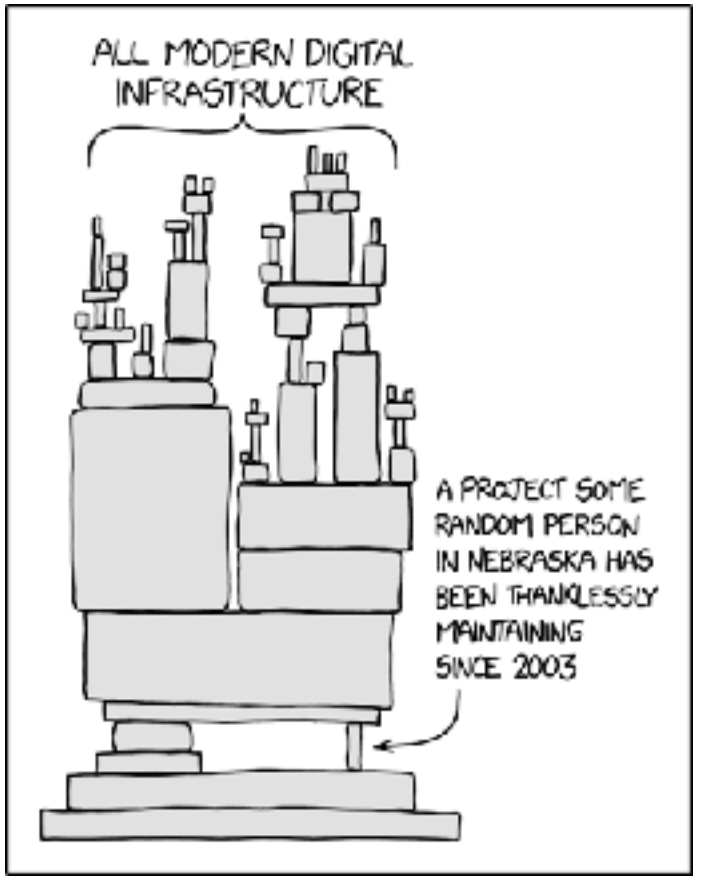
- Provide a (prebuilt!) package
- [aspire to] use Semantic Versioning (major.minor.patch) wherever possible
  - Patch: only bug fixes, no interface changes
  - Minor: add new interfaces
  - Major: anything can happen
- Set upper bounds on the versions of your dependencies where appropriate
- Run tests before publishing
- Do not check package repository credentials into git
  - Use secrets in GitHub Actions
  - Use (masked, protected) CI variables in GitLab CI

Much package, such discoverable

Boring thing 3: managing dependency versions

You're not the only one who can break your software

- Your software likely depends on libraries, and can break when they change
- Solutions:
  - Depend on exact versions, and never change them
  - Depend on a range of versions, updating as new versions are released and tested
- Dependency managers exist to automate upgrades:
  - Dependabot: built into GitHub
  - Renovate: more configurable, e.g.
    - Upgrade rules for each dependency group
    - Bundle updates to minimize noise
- Your CI vets new versions for you!



https://xkcd.com/2347/

Example

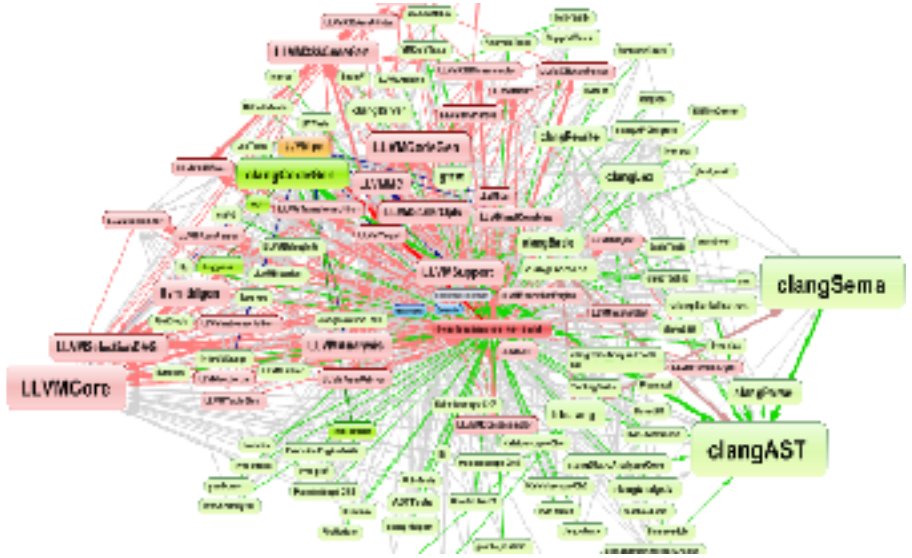
Renovate on GitHub

- You authorize Renovate for your repo/user/organization
- Renovate bot proposes upgrades based on your rules, e.g.
  - Run once a month, on Thursday morning
  - Major updates: file a PR
  - Minor, patch updates: group changes and merge if tests pass
  - Development dependencies: group changes and merge if tests pass
- If compatibility issues arise: fix and push to base branch

A PR opened by Renovate

Good practices for dependency upgrades

- Use a dependency manager with lock files for reproducibility
- Treat deprecation warnings as errors in your tests
- Beware of dependency hell
  - Dependencies are a trade-off. Code you don't write is code you don't have to maintain, but every new dependency is a potential upgrade headache.
  - Use only documented, public interfaces
  - If you do have to depend on internals, keeping a frozen copy ("vendoring") is sometimes less bad than pinning a specific version (if license allows)



A frightening but mostly harmless dependency graph. Source

Takeaways

- GitHub Actions and GitLab CI/CD let you trigger an action whenever you push, file a PR, the clock strikes midnight, etc
- Use this to
  - Check whether changes break your software
  - Publish packages to PyPI/container registry/whatever on demand
  - Keep your dependencies up to date
  - Other things: build and publish a static webpage, build and publish docs, keep a mirror repository in sync, etc