CI/CD & Automation

Putting all the pieces together

What is CI/CD?

CI/CD stands for Continuous Integration and Continuous Delivery / Deployment depending on how its being used

Continuous Integration is the practice of integrating code into a shared repository and building/testing each change automatically, as early as possible - usually several times a day

Continuous Delivery adds that the software can be released to production at any time, often by automatically pushing changes to a staging system

Continuous Deployment goes further and pushes changes to production automatically

What is CI/CD?

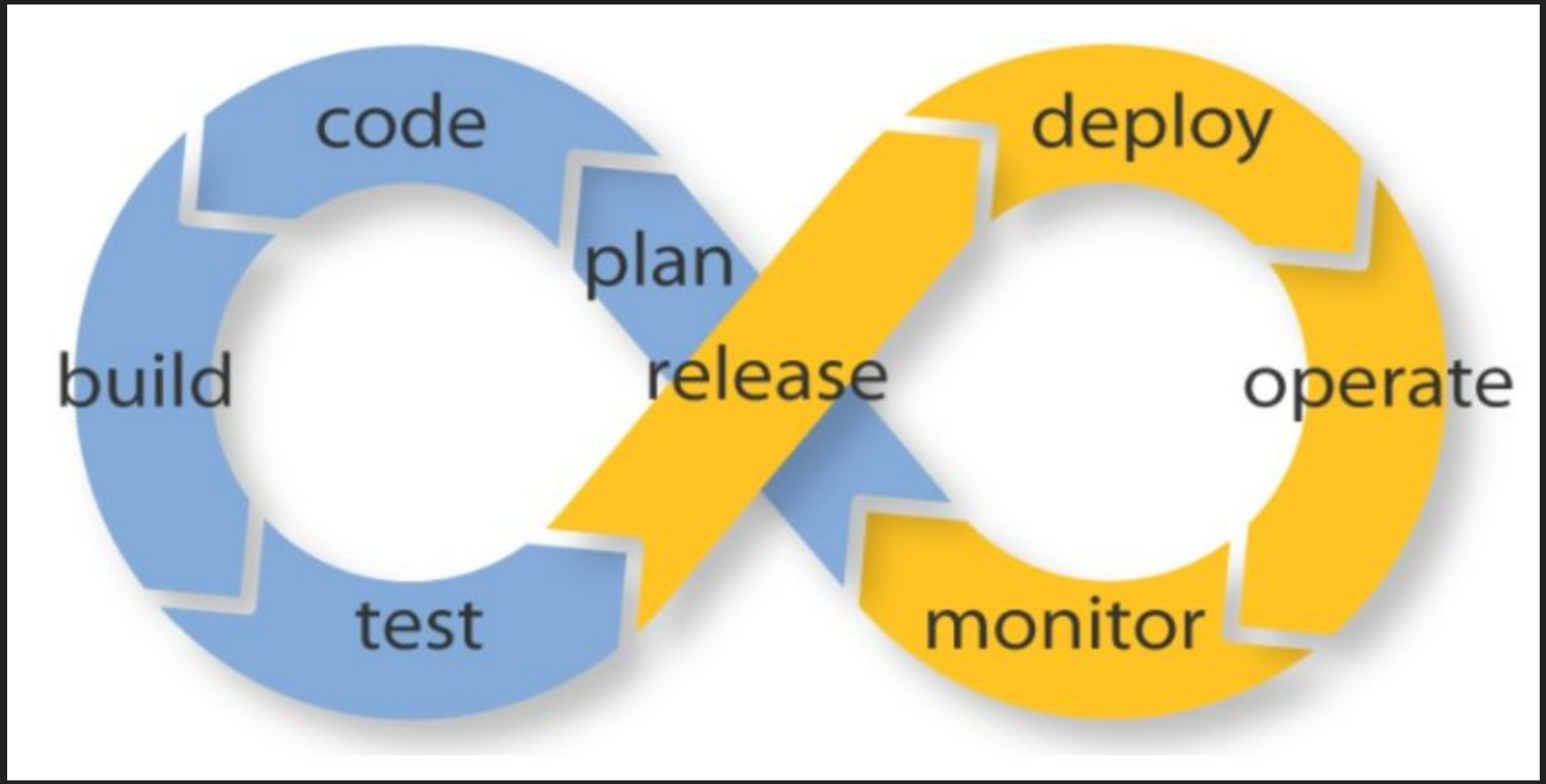
- Continuous Integration
  - Detects errors as quickly as possible
  - Fix while fresh in your mind
  - Reduces integration problems
  - Smaller problems are easier to digest
  - Don’t compound problems
  - Allows teams to develop faster, with more confidence

- Continuous Delivery
  - Ensures that every change to the system is releasable
  - Lowers risk of each release - makes releases “boring”
  - Delivers value more frequently
  - Get fast feedback on what users care about

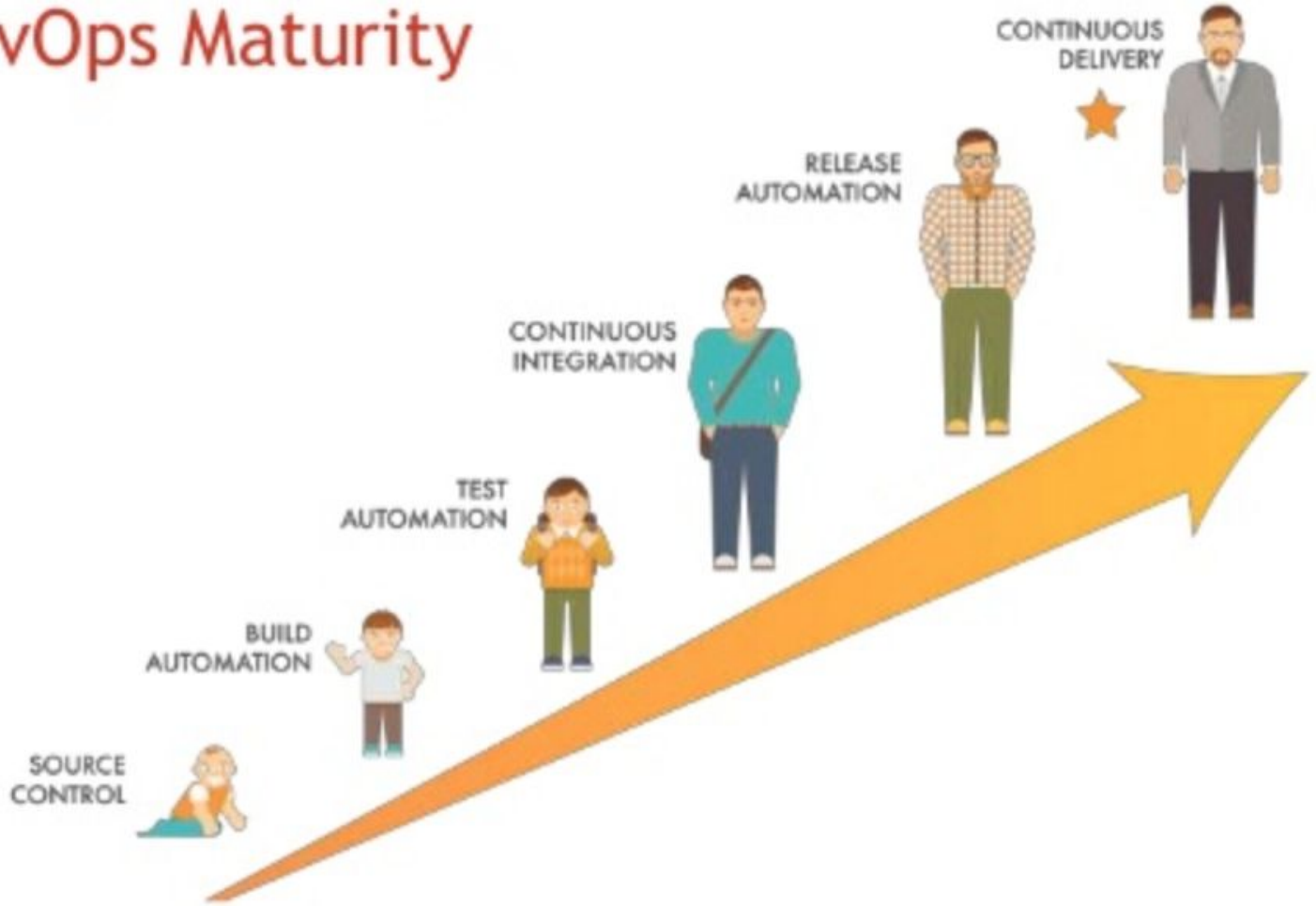
What is CI/CD?

- Continuous Deployment
  - Higher degree of automation
  - Build/deployment occurs automatically
  - Releases get tested and promoted to new environments
  - Automated path to PROD
  - Often handles Infra/Platform Orchestration in addition to Application build/release

In this class will we mostly be focusing on Continuous Deployment w.r.t Infrastructure Automation


code
plan
build
release
deploy
operate
test
monitor

DevOps Maturity



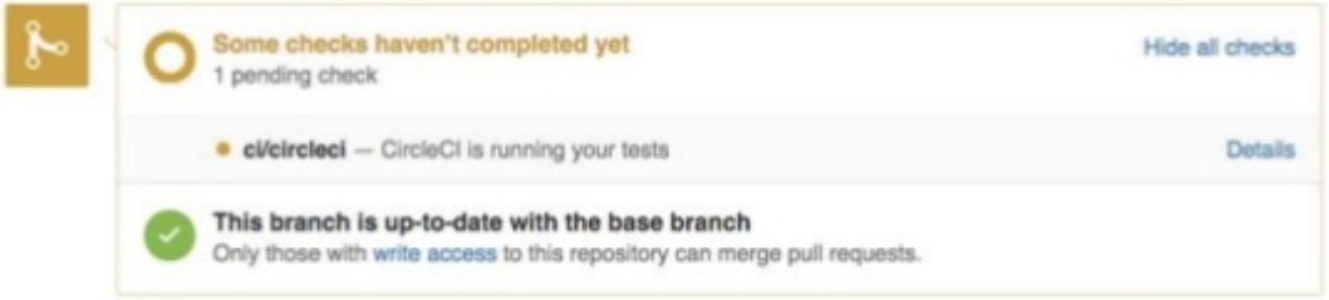
Continuous Integration

Continuous Integration focuses on having developers frequently integrate their code into a main branch of a common repository instead of developing them in isolation

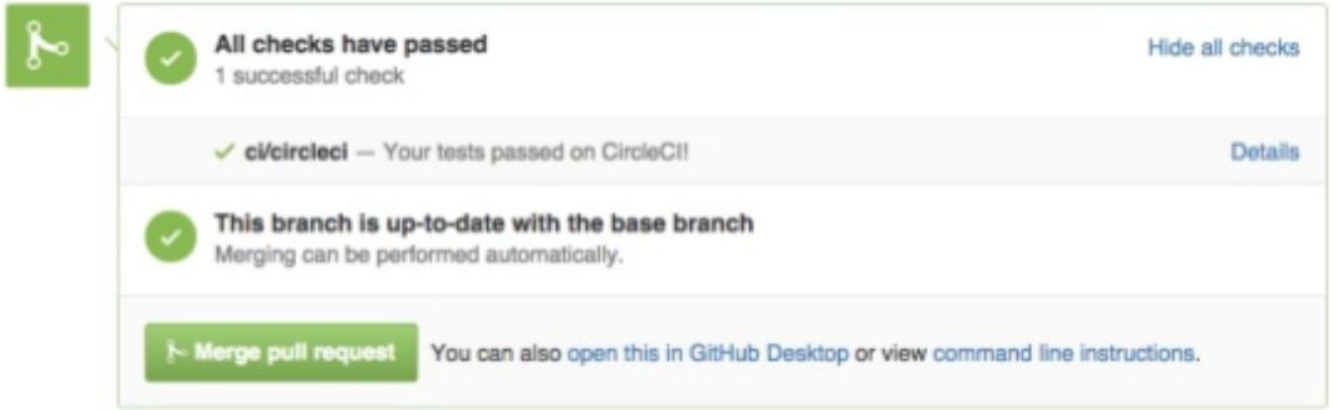
The more frequently code gets committed to the repo and tested, the easier it is to integrate with other features being developed and avoid conflicts late in development

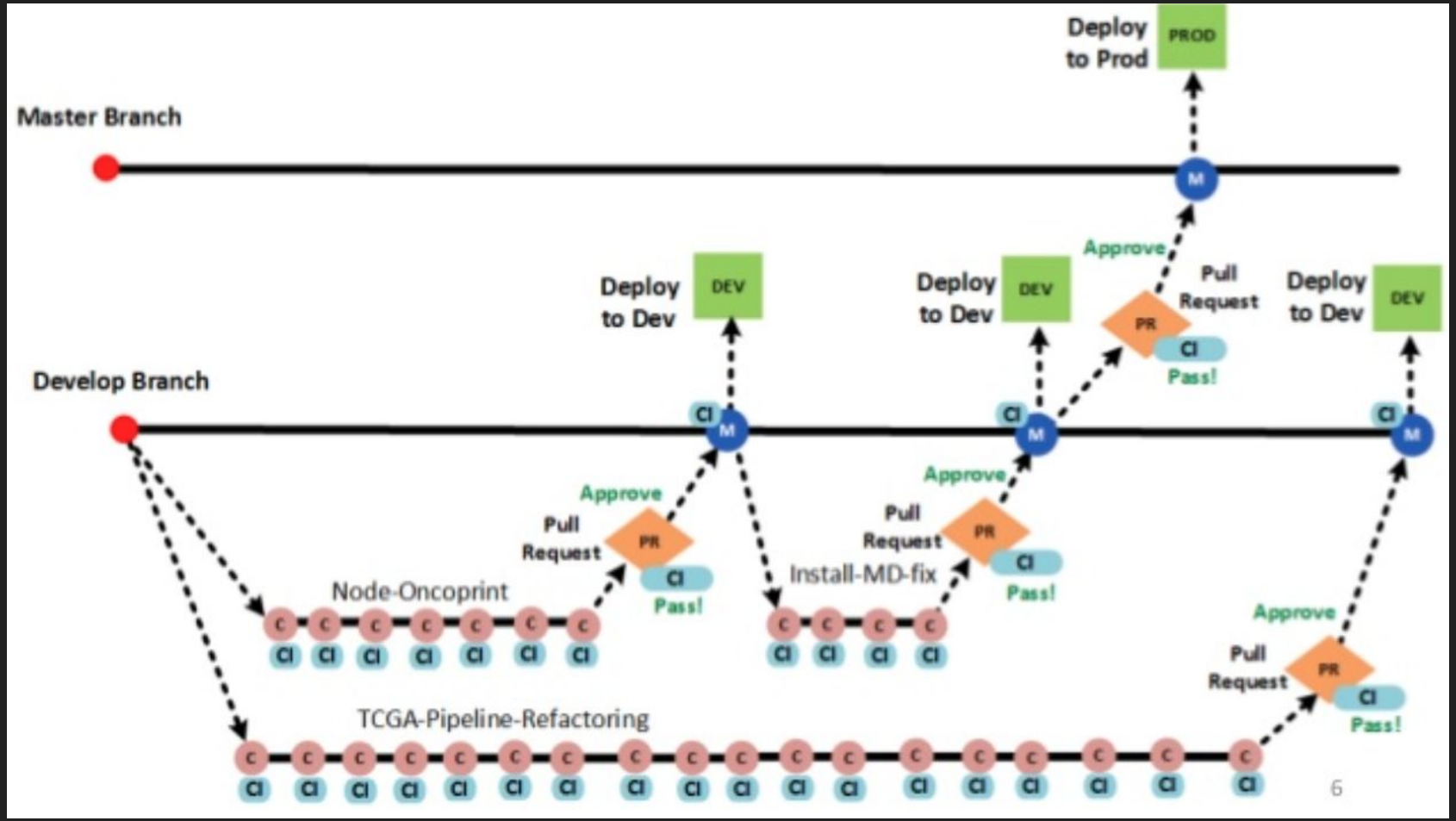
The goal of CI is to refine integration into a simple, easily-repeatable everyday development task that will serve to reduce overall build costs and reveal defects early in the cycle

- Pull request status while CI testing is in progress:



- Pull request status after CI testing is complete; ready to merge without fear




Deploy to Prod
PROD
Master Branch
M
Approve
Pull Request
PR
CI
Pass!
Deploy to Dev
DEV
Develop Branch
Node-Oncoprint
Install-MD-fix
TCGA-Pipeline-Refactoring
C
CI

Continuous Delivery

Continuous Delivery focuses on automating the software delivery process so that teams can easily and confidently deploy their code to production at any time

By ensuring that the codebase is always in a deployable state, releasing software becomes an unremarkable event without the complicated ritual and wondering what is going to fail and who needs to be ‘on call’

Teams can therefore be confident that they can release whenever they need to without complex coordination or late-stage testing

Continuous Delivery and Continuous Deployment leverage what we call a ‘Pipeline’ of automation which handles testing, merging, and cutting releases

Pipelines Need:

- ✓ Branching
- ✓ Looping
- ✓ Restarts
- ✓ Checkpoints
- ✓ Manual Input

Continuous Deployment

Continuous Deployment extends continuous integration and continuous delivery so that the software build will automatically deploy if it passes all automated tests

The huge change here is that there is no need for a person to decide when and what goes into production - if the tests pass it gets deployed

As we talked about during Infrastructure as Code, automated deployments push features and fixes to customers quickly, encourages smaller changes with limited scope, and helps avoid confusion over what is currently deployed to production

Continuous Deployment

Without a final manual verification before deploying a piece of code, developers must take responsibility for ensuring that their code is well-designed and that the test suites are up-to-date

This fully automated deploy cycle can be a source of anxiety for organizations worried about relinquishing control to their automation system of what gets released and can sometimes fall apart

Key Concepts

A lot of these concepts will be very similar to those we talked about w.r.t Infrastructure as Code

- Small, Iterative Changes
  - Break up large features into small testable pieces
  - Minimize integration issues with other features
  - Control the “Blast Radius”

- Keep Building/Testing Phases Fast
  - In order to encourage building and testing often, they must be able to be accomplished quickly
  - If every commit kicks off a build even a few extra minutes of build time adds up quickly
  - If possible, run testing steps in parallel

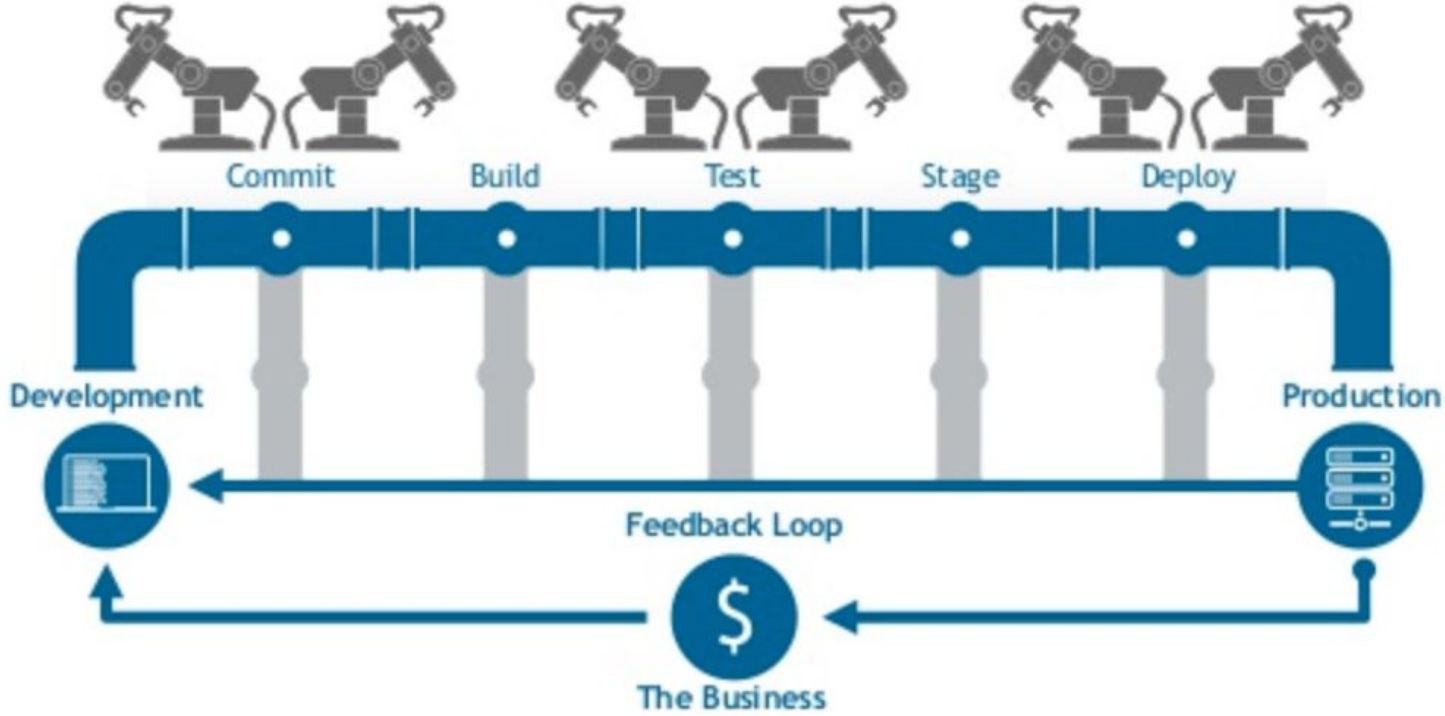
Key Concepts

- Code should be built once at the beginning of the pipeline
  - Software should be stored and accessible to later processes without rebuilding
  - By using the exact same artifact in each phase, you can be certain that you are not introducing inconsistencies as a result of different build tools

- Deployment environments should be consistent
  - Clean deployment environments should be provisioned each test cycle to prevent legacy conditions from compromising the integrity of the tests
  - The staging environments should match the production environment as closely as possible to reduce unknown factors present when the build is promoted

- Consistent processes should be used to deploy the build in each environment
  - Each deployment should be automated and each deployment should use the same centralized tools and procedures
  - Ad-hoc deployments should be eliminated in favor of deploying only with the pipeline tools

Continuous Delivery and Automation are Key



Automated Testing

Continuous integration, delivery, and deployment all rely heavily on automated tests to determine the correctness of each code change. As we just discussed, the goal of testing is to catch issues/errors and fail fast such that developers can iterate quickly

As such, different types of tests are needed throughout these processes to gain confidence in a given solution

- Smoke Testing
  - Initial checks designed to ensure very basic functionality
  - Run at the start of each testing cycle as a sanity check
  - Goal is to catch red flags that signal future testing is pointless
  - Smoke tests can be implemented at multiple stages of the pipeline

Automated Testing

- Unit Testing
  - Responsible for testing individual elements of code in an isolated way
  - Tests individual functions and classes
  - External dependencies are replaced with stub or mock implementations
  - Typically run by developers on their workstations prior to submitting changes
  - CI/CD usually always run these as a safeguard in case the developer did not run them ahead of time
- Integration Testing
  - Groups together components and testing them as a full package
  - Ensures that components cooperate when interfacing with one another
  - Changes must prove that they do not break existing functionality and that they interact with other code as expected
  - Verifies that the changes can be deployed into a clean environment
  - Usually the first time that new code is tested against real external libraries, services, and data

Automated Testing

- Acceptance Testing
  - One of the last tests types that are performed on software prior to delivery
  - Used to determine whether a piece of software satisfies all of the requirements from the business or user’s perspective
  - Tests are sometimes built against the original specification and test agreed upon interfaces
  - Often deploys the build to a staging environment that mirrors the production system for QA and Performance testing as well

So how do we actually implement CI/CD?

Implementing CI/CD

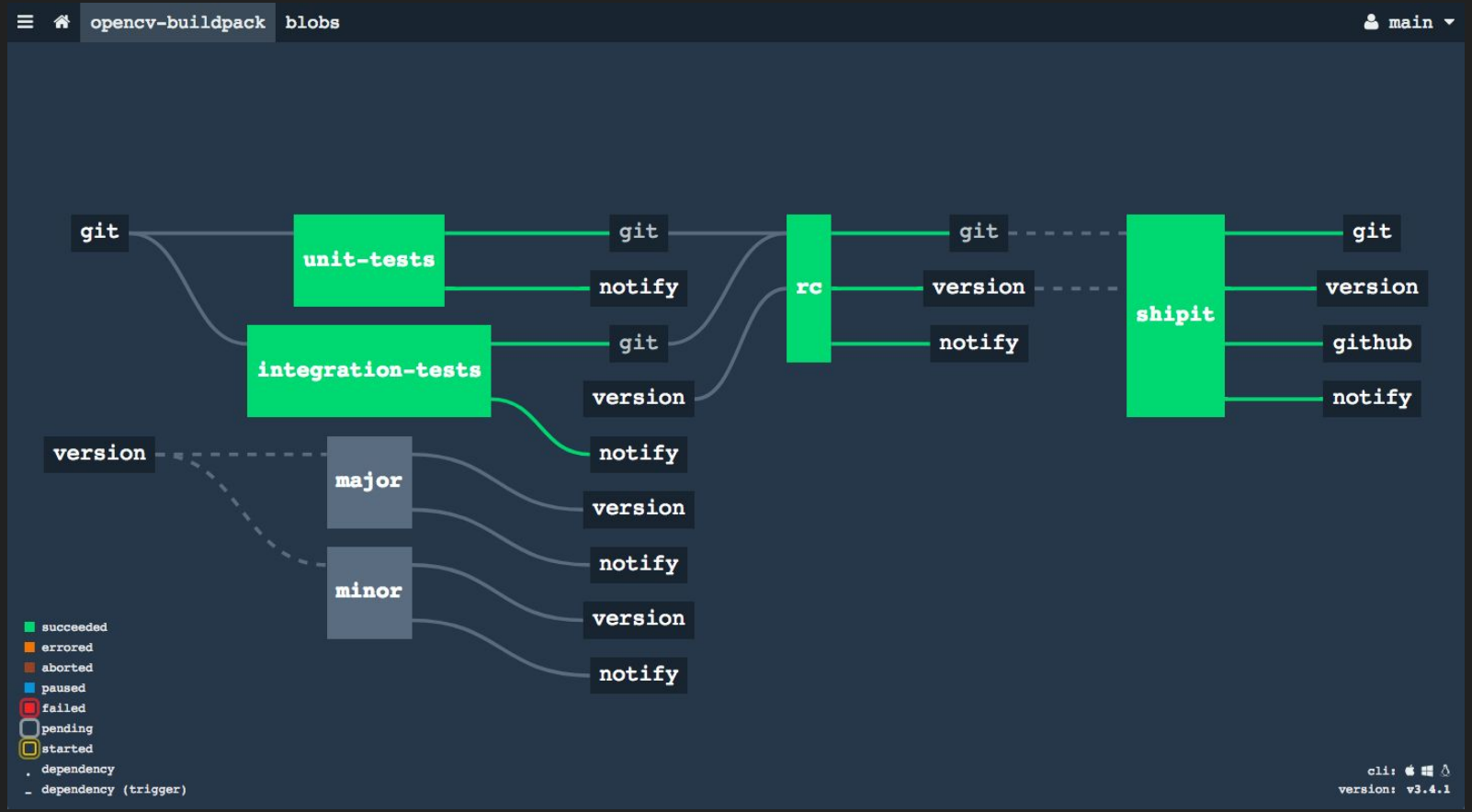
There are tons of CI/CD systems and tools out in the wild and more are being developed every day

A few of the big players are:

- Jenkins
- CircleCI
- TravisCI
- Concourse
- TeamCity

For this class, we will be focusing on Concourse as it is open source and can be deployed on premise fairly easily while still being used by Fortune 500 companies

Implementing CI/CD



Concourse

You can think of an automation pipeline as a distributed, higher-level, continuously-running Makefile. Concourse provides a pretty open and generalist approach to automation and CI/CD which makes it pretty flexible.

When developing Concourse pipelines there are three main concepts

- Tasks - scripts that are executed with variables and parameters passed in from the pipeline
- Resources - the inputs and outputs of the system (Git, Docker, Slack, etc)
- Jobs - the glue that pieces resources and tasks together into a cohesive unit

Concourse Tasks

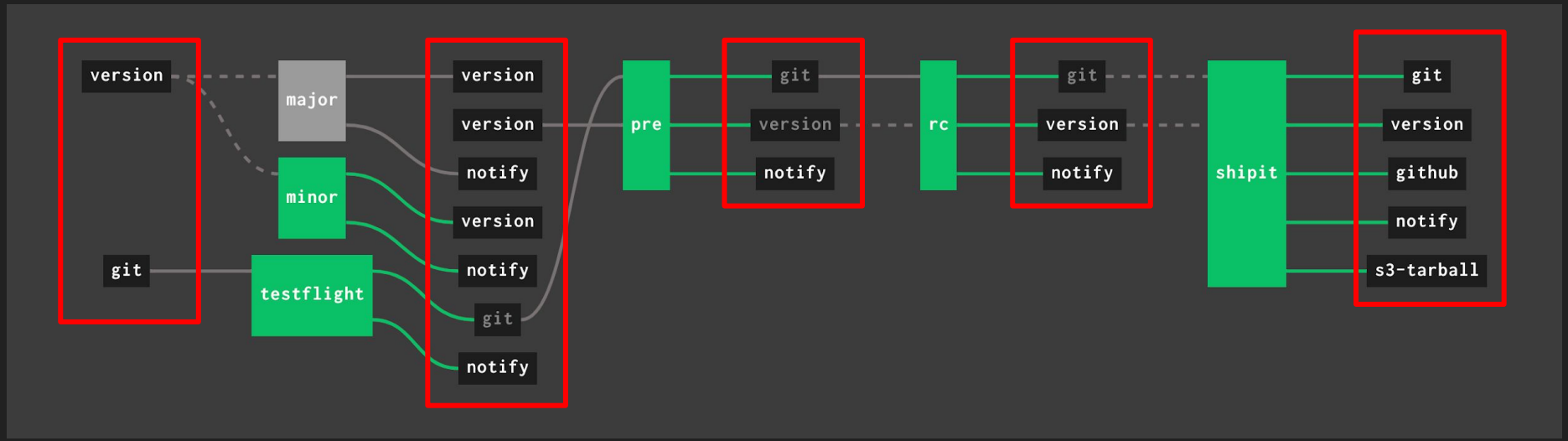
- Tasks are the smallest configurable unit in a Concourse Pipeline
- They should always have the same behaviour - if the inputs are the same, the outputs should always be the same
- They basically run commands from within a container

- If the task exits 0 then it is treated as successful
- If the tasks exits non-0 then it is assumed to have failed

Concourse Tasks

```
- task: make-a-file
  config:
    platform: linux
    image_resource:
      type: registry-image
      source: { repository: busybox }
    run:
      path: sh
      args:
        - -exc
        - ls -la; echo "Created a file on $(date)" > ./files/created_file
```

Concourse Resources



Concourse Resources

Resources are objects used for Jobs in the pipeline based on the the resource type (S3, Git, Slack, etc)

Some of these are officially supported resources:

- Git, Docker Image, S3, and (more)

And some are community supported

- Slack, Twitter, RSS, and more

For a somewhat exhaustive list you can check them out here

https://github.com/concourse/concourse/wiki/Resource-Types

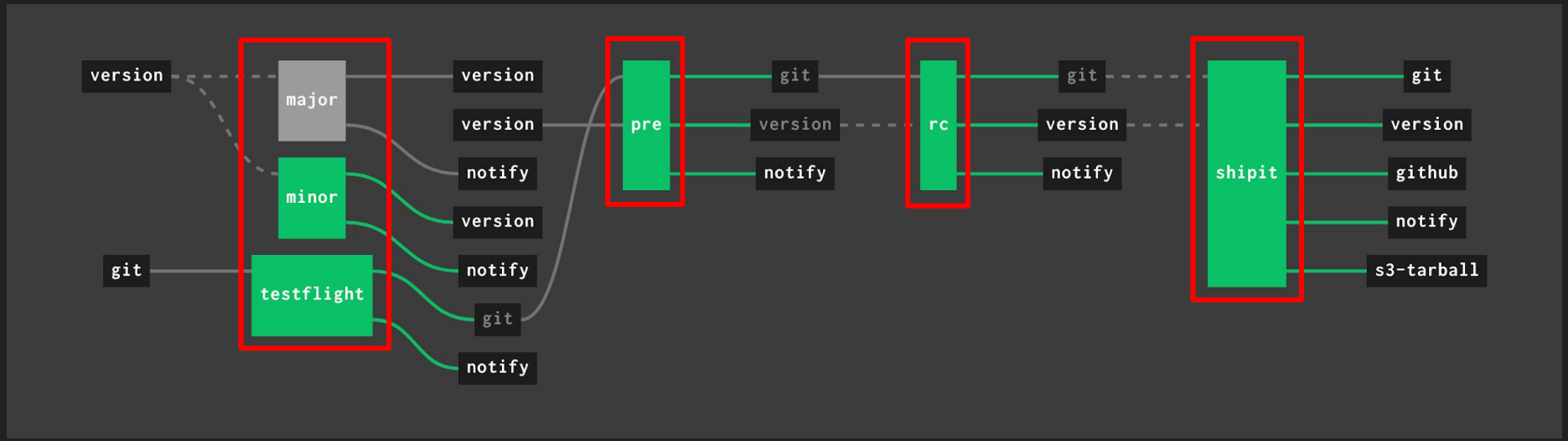
Resource Type	Maintained By...
`git` resource	by @concourse
`hg` resource	by @concourse
`time` resource	by @concourse
`s3` resource	by @concourse
`semver` resource	by @concourse
`github-release` resource	by @concourse
`registry-image` resource	by @concourse
`docker-image` resource	by @concourse
`pool` resource	by @concourse
`cf` resource	by @concourse
`bosh-io-release` resource	by @concourse
`bosh-io-stemcell` resource	by @concourse
Slack Reading and Posting	by @jleben
Slack notifications	by @cloudfoundry-community
Github Pull Requests	by @telia-oss
GitLab Merge Requests	by @swisscom
OpenStack Swift	by @sapcc
Key Value resource	by @swce
Key Value resource	by @moredhel
Flowdock notifications	by @starkandwayne
Email	by @pivotal-cf

Concourse Resources

```
resources:
  - name: git
    type: git
    source:
      uri:         (( grab meta.github.uri ))
      branch:      (( grab meta.github.branch ))
      private_key: (( grab meta.github.private_key ))

  - name: version
    type: semver
    source :
      driver:            s3
      bucket:            (( grab meta.aws.bucket ))
      region_name:       (( grab meta.aws.region_name ))
      key:               version
      access_key_id:     (( grab meta.aws.access_key ))
      secret_access_key: (( grab meta.aws.secret_key ))
      initial_version:   (( grab meta.initial_version || "0.0.1" ))
```

Concourse Jobs



Concourse Jobs

Jobs are sort of the glue logic that ties resources and tasks together in a cohesive unit of inputs and outputs. The basically plan out what the pipeline is going to do, and in what order.

- What do I need to do -> Task
- What do I need to use? -> Resource
- What so I need to produce? -> Resource

Concourse Jobs

```
jobs:
  - name: create-and-consume
    public: true
    plan:
      - task: make-a-file
        config:
          platform: linux
          image_resource:
            type: registry-image
            source: { repository: busybox }
          run:
            path: sh
            args:
              - -exc
              - ls -la; echo "Created a file on $(date)" > ./files/created_file
          outputs:
            - name: files
      - task: consume-the-file
        config:
          platform: linux
          image_resource:
            type: registry-image
            source: { repository: busybox }
          inputs:
            - name: files
          run:
            path: cat
            args:
              - ./files/created_file
```

Examples (Demo)

Questions?