SOFTWARE ENGINEERING
CO3001

CONTINUOUS INTEGRATION AND DELIVERY (CI/CD)

Anh Nguyen-Duc
Tho Quan-Thanh

WEEK 11

Adapted from https://iansommerville.com/software-engineering-book/slides/

OUTLINE

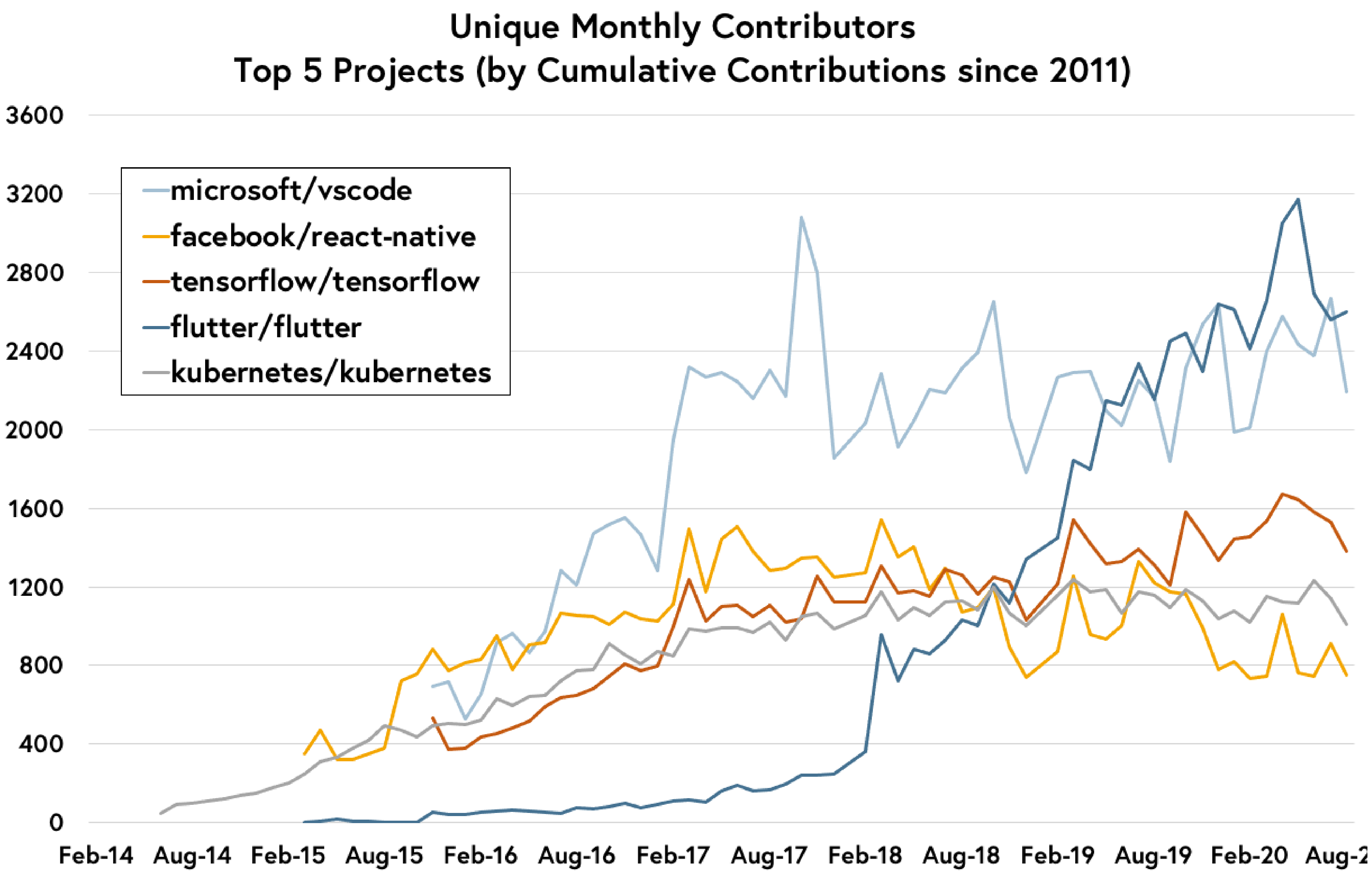
- ✓ Challenges of modern code development
- ✓ Code integration
- ✓ Continuous integration
- ✓ Continuous delivery
- ✓ DevOps

THE CHALLENGE: COMPLEXITY AND SIZE

- As the project grows, complexity grows:
  - Physical code size
  - Dependencies
  - Number of developers
  - Package versions
- Examples of well-known open source projects

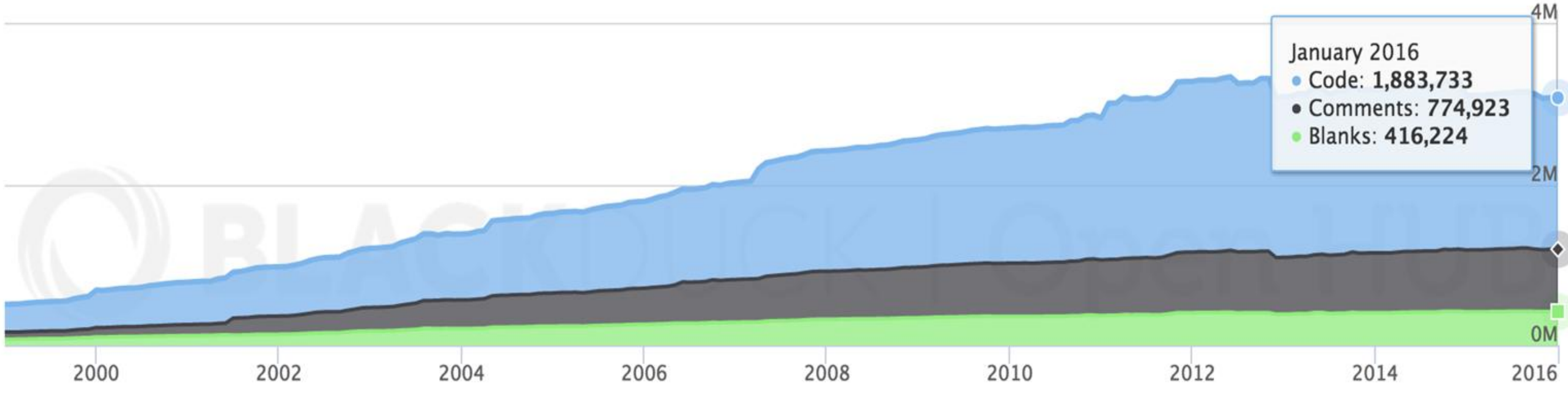
THE CHALLENGE: COMPLEXITY AND SIZE

- Some notably large codebases include:
  - Google: monolithic, 1 billion files, 9 million source code files, 2 billion lines of source code, 35 million commits in total, 86 TB total size (January 2015)
  - Facebook: monolithic, 8 GB (repo 54 GB including history, 2014),[6] hundreds of thousands of files (2014)
  - Linux kernel: distributed, over 15 million lines of code (as of 2013 and kernel version 3.10)


Unique Monthly Contributors
Top 5 Projects (by Cumulative Contributions since 2011)
microsoft/vscode
facebook/react-native
tensorflow/tensorflow
flutter/flutter
kubernetes/kubernetes
3600
3200
2800
2400
2000
1600
1200
800
400
0
Feb-14
Aug-14
Feb-15
Aug-15
Feb-16
Aug-16
Feb-17
Aug-17
Feb-18
Aug-18
Feb-19
Aug-19
Feb-20

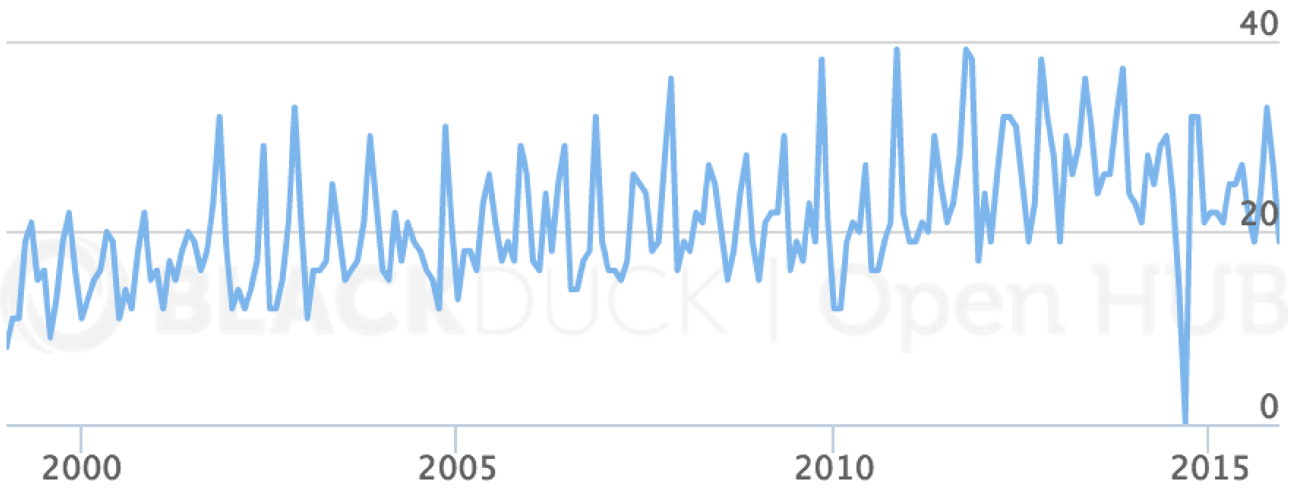
Example - Geant4

- A framework for the simulation of the passage of particles through matter.
  - Used in HEP, medical and space physics
- Just under 2 million lines of code
  - Mostly C++



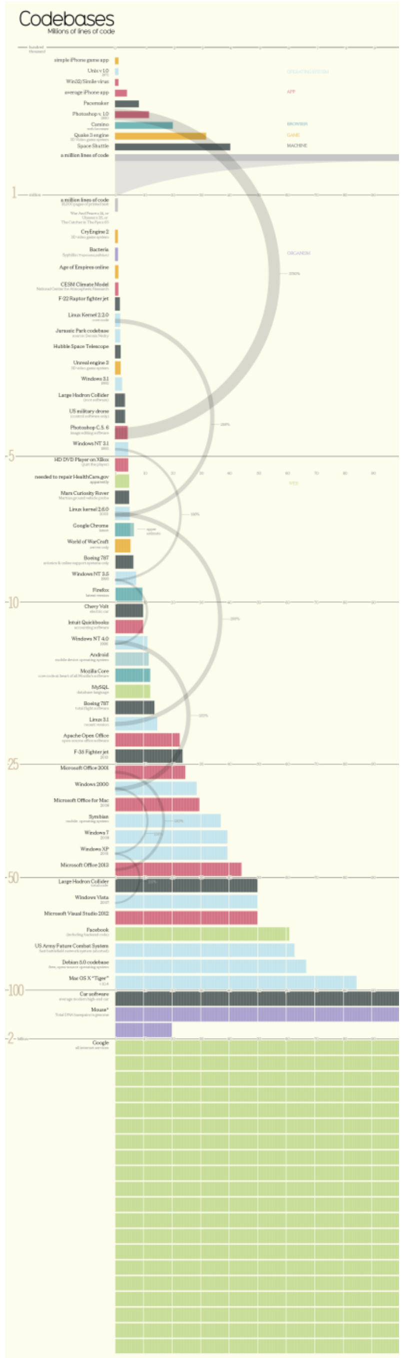
LOC

Example - Geant4



# of contributors

- 537 person-years
  - Estimated cost: ~ €29 million
- 58,683 commits from 160 developers


Codebases
Millions of lines of code
simple iPhone game app
Unix v 1.0
Win32/Simile virus
average iPhone app
Pacemaker
Photoshop v. 1.0
Camino
Quake 3 engine
Space Shuttle
a million lines of code
OPERATING SYSTEM
APP
BROWSER
GAME
MACHINE
CryEngine 2
Bacteria
ORGANISM
Age of Empires online

THE CHALLENGE

- ✓ How do we handle increasing code-base sizes?
- ✓ How do we handle an increasing number of developers?
  - How can developers interact with each other?
- ✓ How do we build across multiple platforms?
- ✓ How do we build multiple versions?
- ✓ <u>How can we make sure we don't break things!</u>

WHAT IS INTEGRATION?

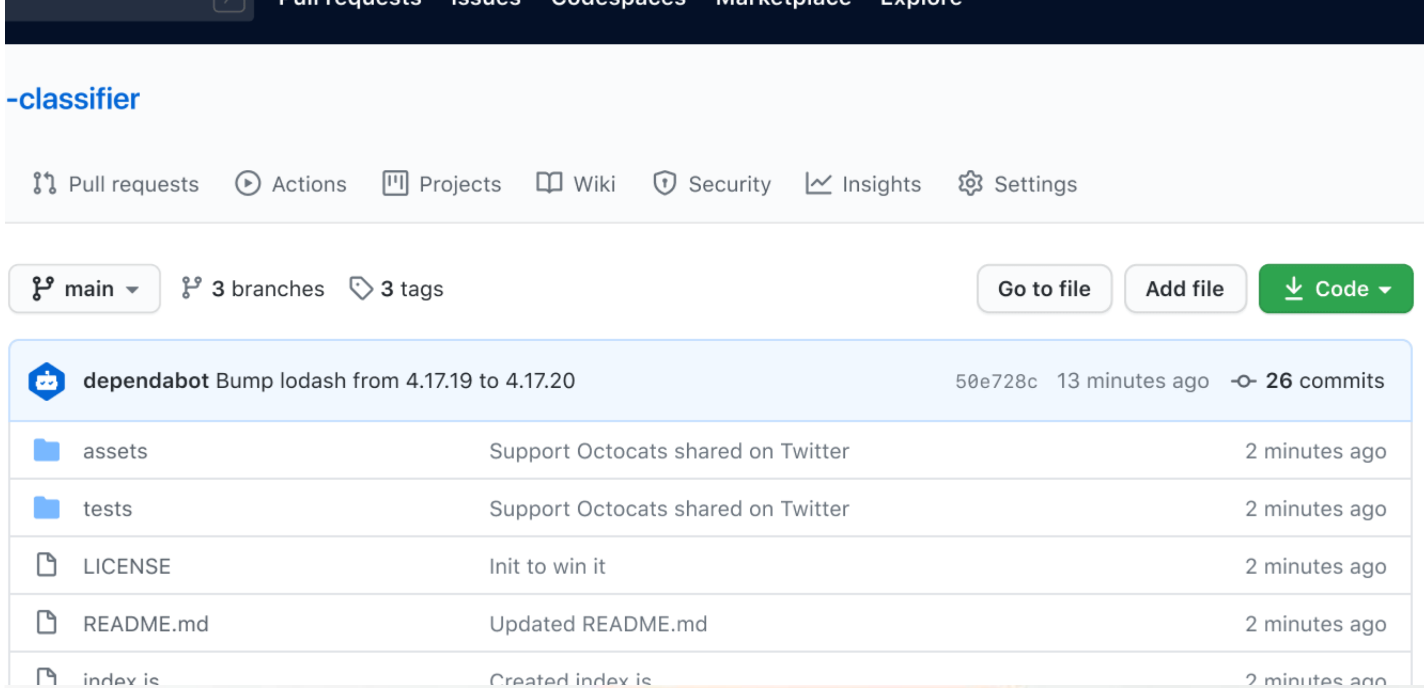
- Software teams often have multiple developers working on the same codebase at the same time(independently):
  - E.g. Developer A works on feature 1 while developer B works on feature 2.
  - E.g. Developer A works on class 123.java while developer B works on class 456.java
- Once they have finished, they needs to integrate their work into the main codebase.

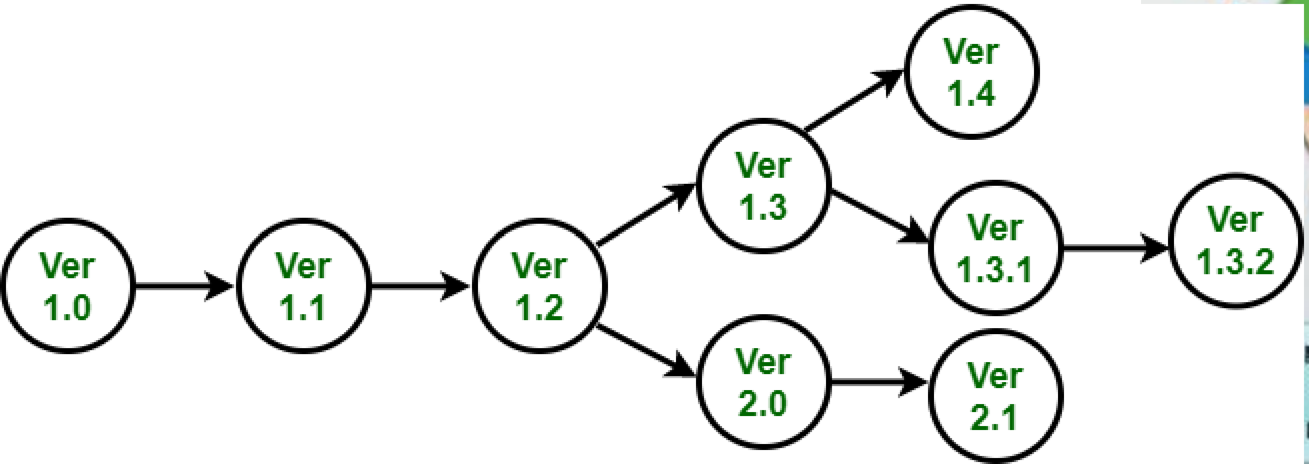
“I can't compile the program if you're in the middle of typing a variable name”

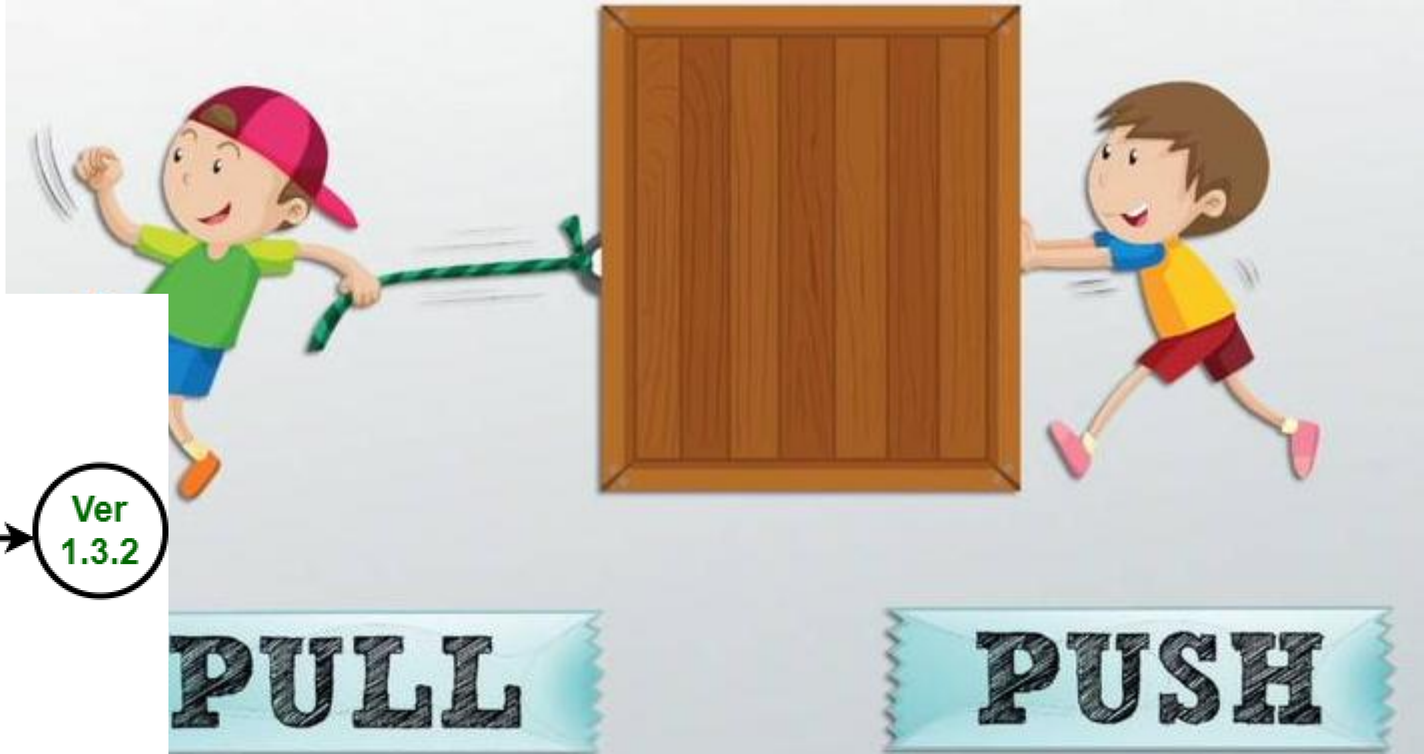
https://martinfowler.com/articles/branching-patterns.html#integration-patterns

TERMINOLOGY

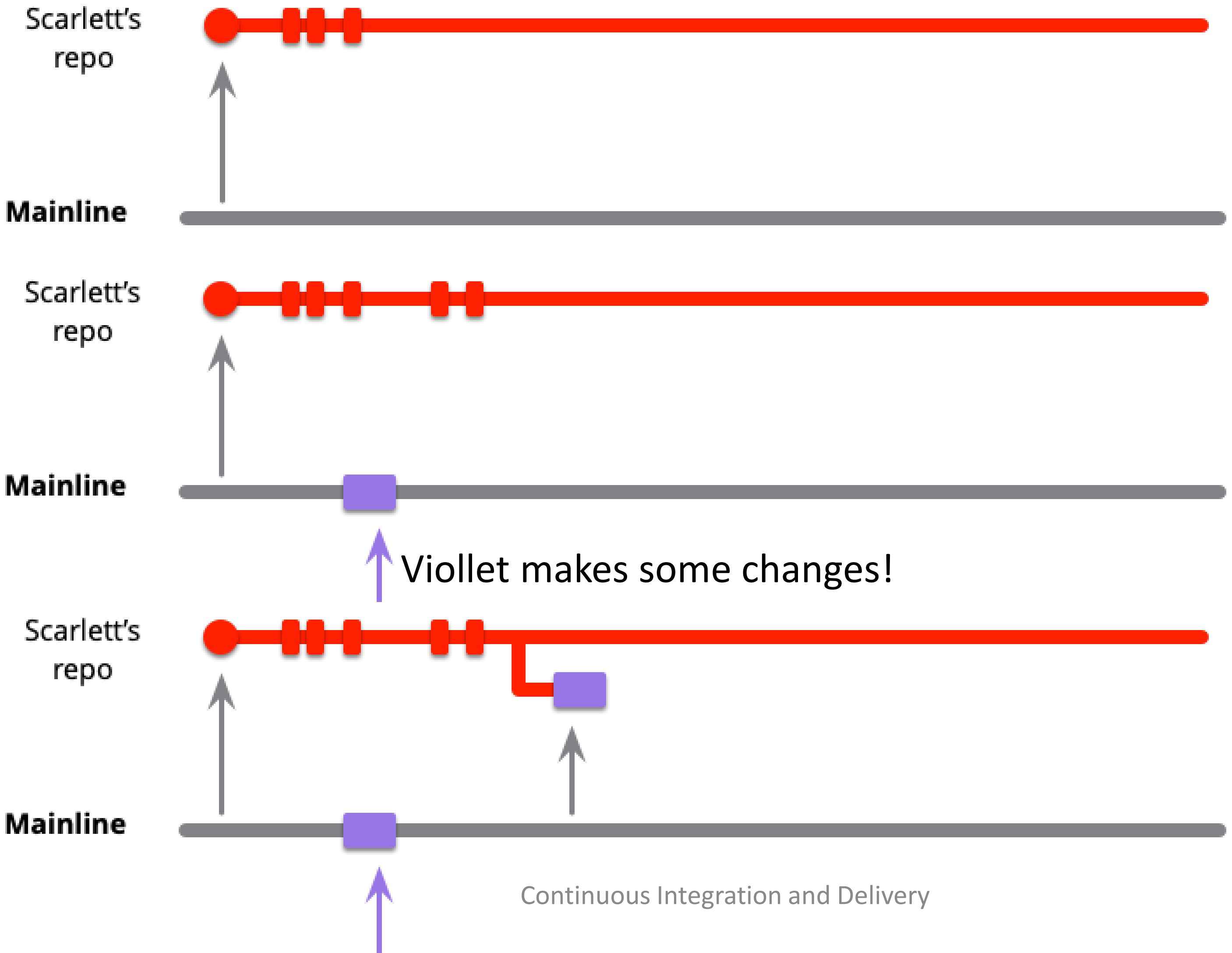
- ✓ Integration
- ✓ Repository
- ✓ Pull vs. push
- ✓ Software Version



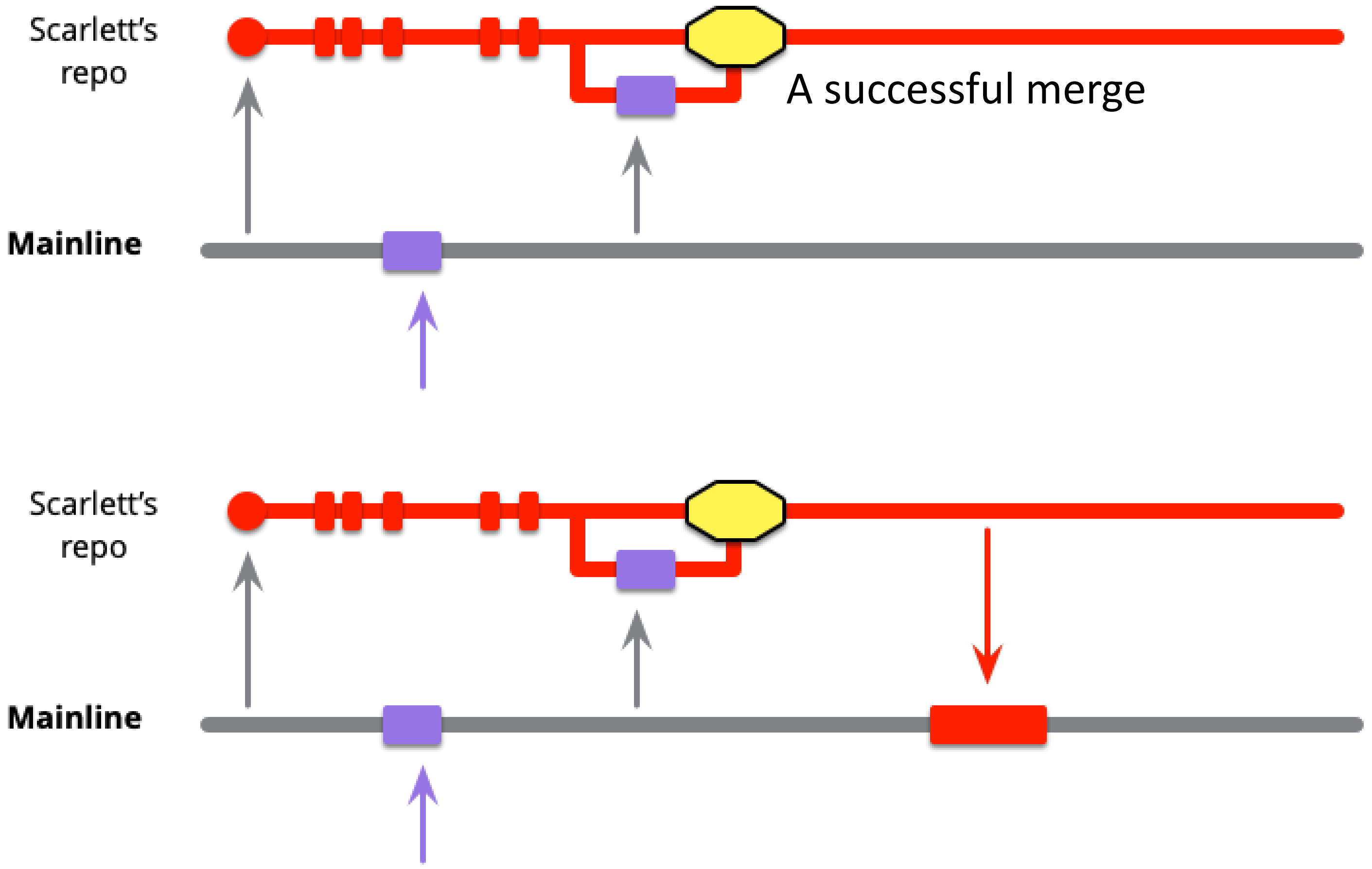




- Mainline integration:Developers integrate their work by pulling from mainline, merging, and - if healthy - pushing back into mainline

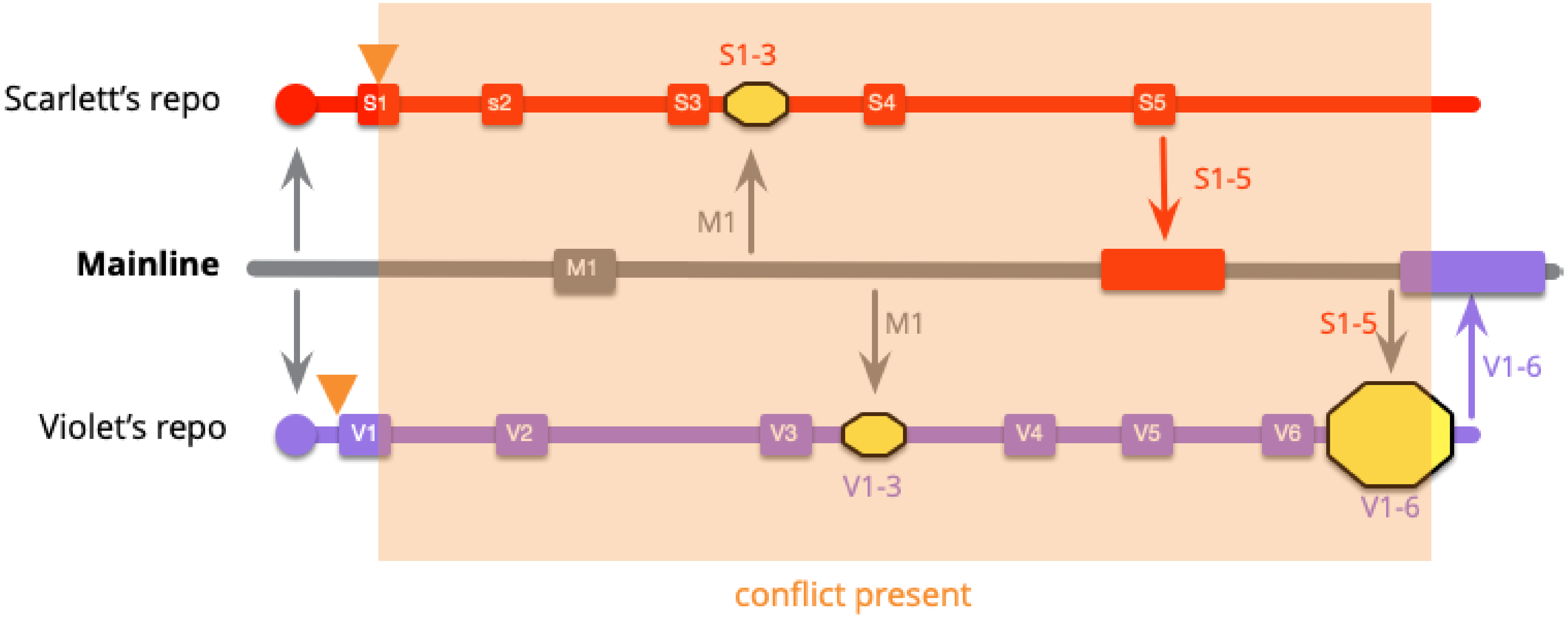


- Mainline integration:Developers integrate their work by pulling from mainline, merging, and - if healthy - pushing back into mainline



INTEGRATION FREQUENCY

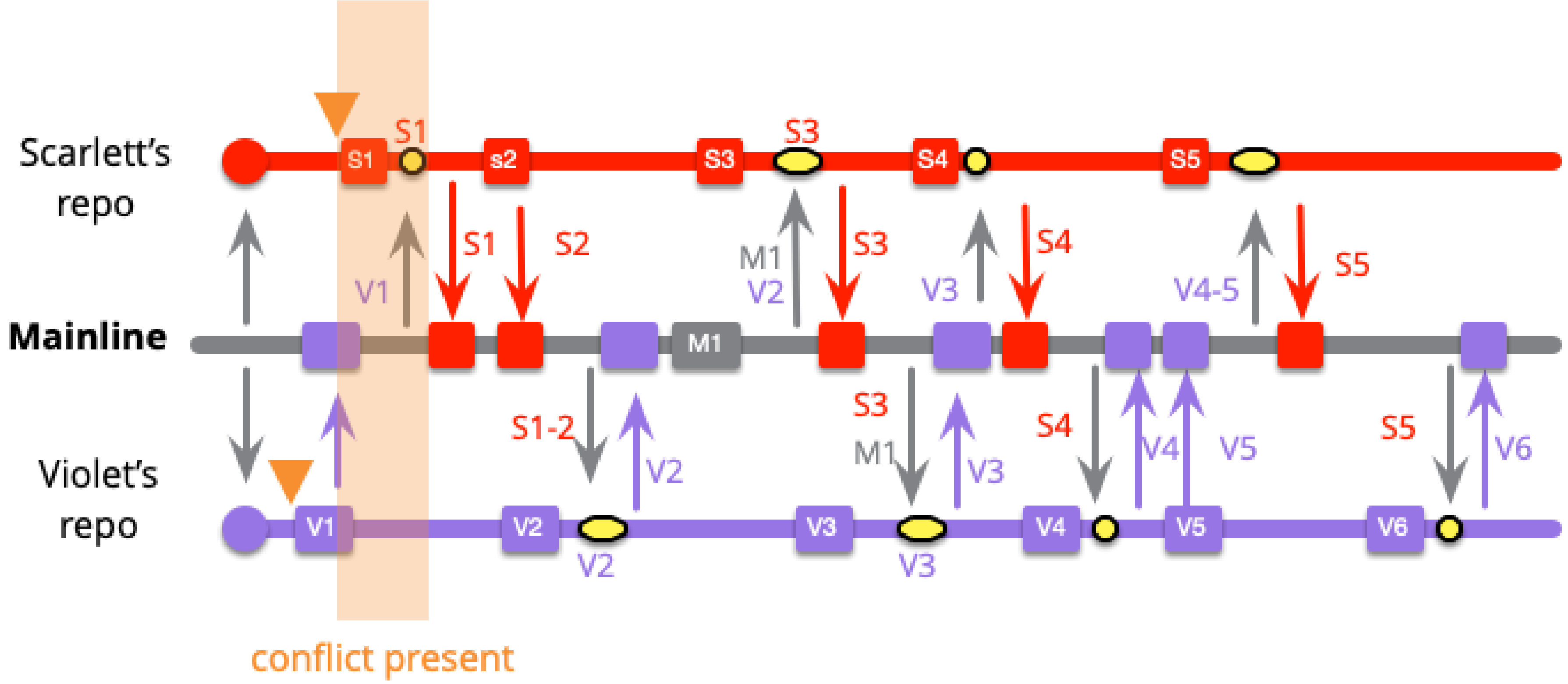
- Elite development teams integrate notably more often than low performers
- Low-Frequency Integration



https://martinfowler.com/articles/branching-patterns.html#integration-patterns

INTEGRATION FREQUENCY

- ✓ Elite development teams integrate notably more often than low performers
- ✓ Hig



https://martinfowler.com/articles/branching-patterns.html#integration-patterns

What is continous integration?

- Continuous integration (CI) is a software development practice where developers in a team integrate their work frequently
- Developers usually integrates several times a day.
- Each integration is verified by anautomated build: compile the code and also run automated tests?
- Question: Why are automated tests run?

Why is continous integration?

- Early/rapid feedback!
  - Do all components/projects compile?
  - Coding standards?
  - Are tests successful?
  - Performance requirements?
  - Problems archiving or deploying?
- Better project visibility
  - Possible to notice trends
  - What features are needed/being added

- Insures clean environments
- Manual tasks automated
- Speedup of working software turnover
- No large integration steps
- Much less likely to break something
- A full working/deployable version at ANY POINT IN TIME
- Complete documentation of who did what

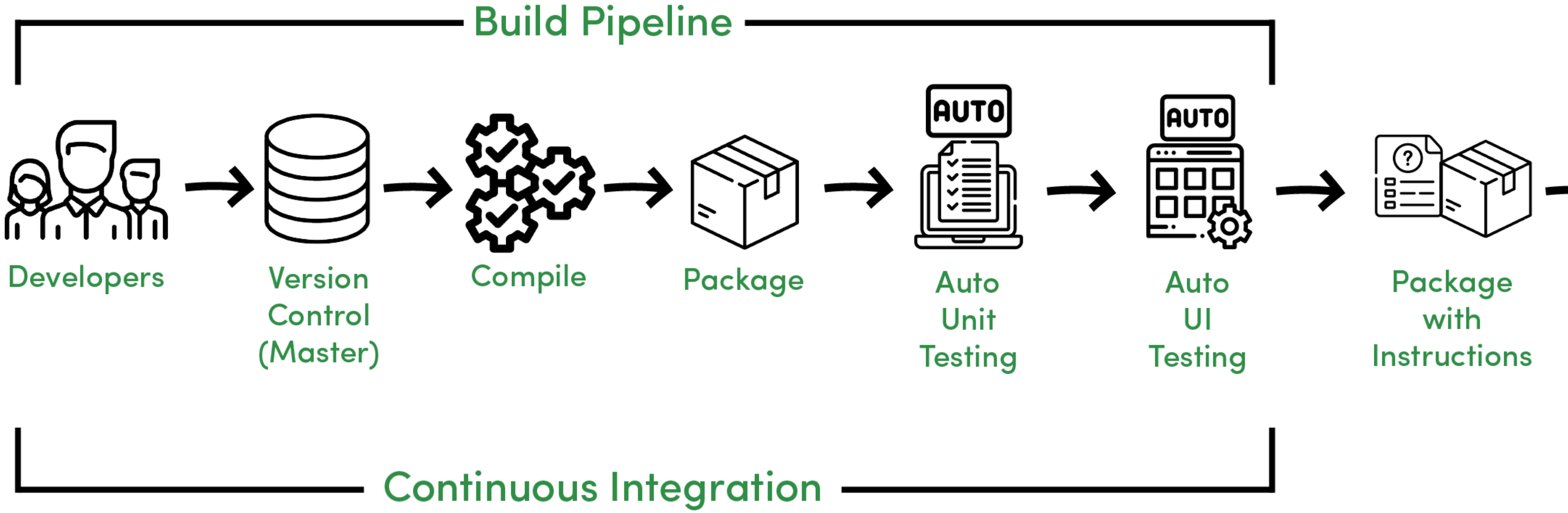
How is continous integration?

- Use various existing tools to:
  - Combine changes often
  - Build often
  - Test often
  - Deploy often

In order for CI to work, individual developers should:

- Commit frequently
  - Many small commits
- Run local build first (if possible)
  - Huge code repos may make this difficult
  - Only commit working code
- Fix broken builds immediately
- Write automated tests

CONTINUOUS INTEGRATION MODEL



- ✓ Version control software
- ✓ Dependency management
- ✓ Automated testing software
- ✓ Continuous integration framework
- ✓ Infrastructure management
- ✓ Build automation

CONTINUOUS INTEGRATION TOOLS

- ✓ Code repositories
  - Github, Bitbucket, Mercurial, BitKeeper, Bzr, CVS, Darcs, Gerrit, Monotone, P4, SVN ...
- ✓ Test frameworks
  - CppUnit, Valgrind, JUnit, Unittest, TestNg ...
- ✓ Continuous Integration
  - Bamboo, Buildbot, CruiseControl, Jenkins, Gitlab CI ...

Wikipedia has a great comparison table:
https://en.wikipedia.org/wiki/Comparison_of_continuous_integration_software

SETTING UP A CI PIPELINE

A simple example of a Flask web appliation

app.py — simple-flask-app

config.yml | Dockerfile | app.py | test.py | requirements.txt

app.py > ...

```python
"""simple website app for CI"""
import os
from flask import Flask, current_app
app = Flask(__name__)

@app.route('/')
def hello_world():
    """main route to return index.html"""
    return current_app.send_static_file('index.html')

if __name__ == '__main__':
    port = int(os.getenv('PORT'))
    app.run(debug=True,host='0.0.0.0', port=port)
```

config.yml — simple-flask-app

config.yml | Dockerfile | app.py | test.py | requirements.txt

.circleci > config.yml

```yaml
  docker: circleci/docker@2.0.1

jobs:
  lint:
    executor: python/default
    steps:
      - checkout
      - restore_cache:
          key: deps1-{{ .Branch }}-{{ checksum "requirements.txt" }}
      - run:
          name: Install Python deps in a venv
          command: |
            python3 -m venv venv
            . venv/bin/activate
            pip install -r requirements.txt
      - run:
          name: "Run pylint"
          command: |
            . venv/bin/activate
            pylint app.py
      - save_cache:
          key: deps1-{{ .Branch }}-{{ checksum "requirements.txt" }}
          paths:
            - "venv"
  test:
    executor: python/default
    steps:
```

circleci

- ✓ Our YAML file defines four different processes to run: lint, test, build and deploy.

config.yml — simple-flask-app

! config.yml × | Dockerfile | app.py | test.py | requirements.txt

.circleci > ! config.yml

```yaml
  docker: circleci/docker@2.0.1

jobs:
  lint:
    executor: python/default
    steps:
      - checkout
      - restore_cache:
          key: deps1-{{ .Branch }}-{{ checksum "requirements.txt" }}
      - run:
          name: Install Python deps in a venv
          command: |
            python3 -m venv venv
            . venv/bin/activate
            pip install -r requirements.txt
      - run:
          name: "Run pylint"
          command: |
            . venv/bin/activate
            pylint app.py
      - save_cache:
          key: deps1-{{ .Branch }}-{{ checksum "requirements.txt" }}
          paths:
            - "venv"
  test:
    executor: python/default
    steps:
```

Pylint

Star your Python code!

- ✓ The lint stage checks for possible errors and formatting issues without running the code. The linting program used in this case is a popular tool called Pylint.

test.py — simple-flask-app

config.yml | Dockerfile | app.py | test.py | requirements.txt

test.py > TestApp > test_404

```python
import unittest
from app import app

class TestApp(unittest.TestCase):

    def setUp(self):
        self.app = app.test_client()

    def test_404(self):
        rv = self.app.get('/i-am-not-found')
        self.assertEqual(rv.status_code, 404)

    def test_homepage(self):
        rv = self.app.get('/')
        self.assertTrue("This is the title of the webpage!" in rv.get_data(as_text=True))

if __name__ == '__main__':
    unittest.main()
```

Pylint
Star your Python code!

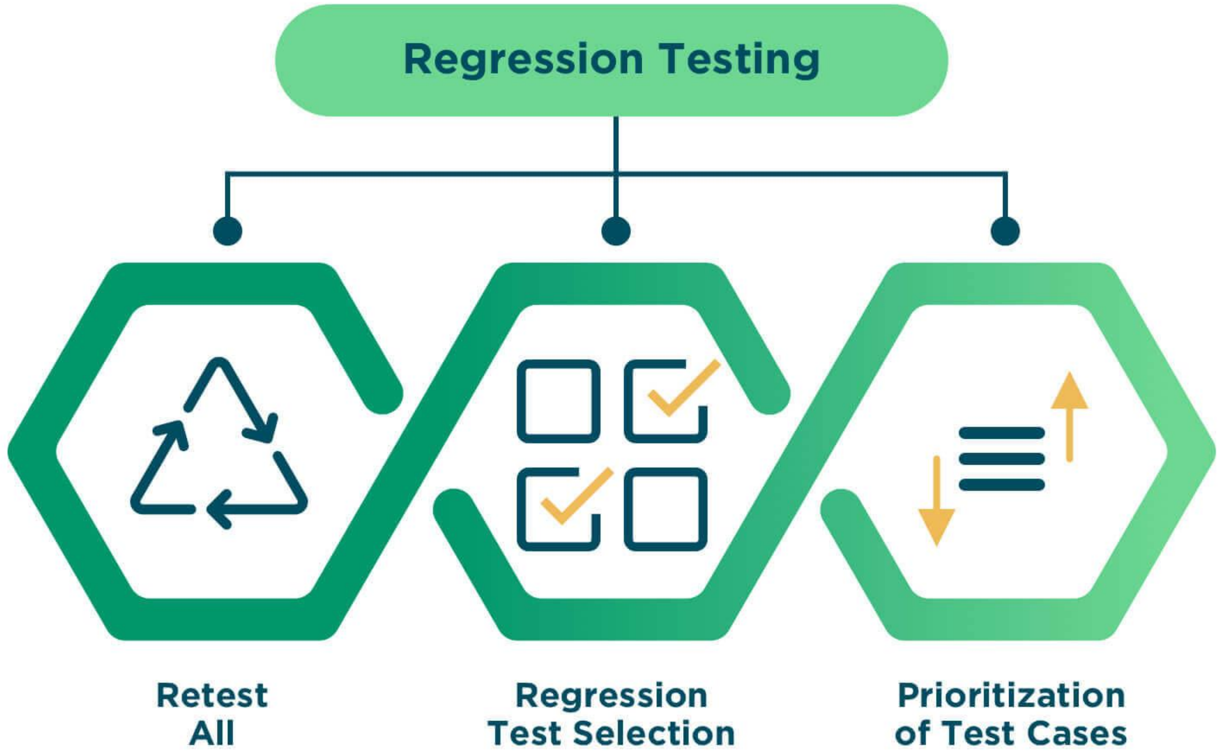
- ✓ The next step in our CI/CD pipeline tutorial is testing. Our tests in this project are run with the unit test framework

AUTOMATED TESTING

- ✓ Automated testing is the application of software tools to automate a human-driven manual process of reviewing and validating a software product
- ✓ Different levels:
  - Unit test
  - Integration test: mocking these 3rd party dependencies and asserting the code interfacing with them behaves as expected
  - Performance test: i.e. speed and responsiveness

REGRESSION TESTING

- Re-running functional and non-functional tests to ensure that previously developed and tested software still performs after a change
- Three types



config.yml — simple-flask-app

config.yml | Dockerfile | app.py | test.py | requirements.txt

.circleci > config.yml

```yaml
      - save_cache:
          key: deps1-{{ .Branch }}-{{ checksum "requirements.txt" }}
          paths:
            - "venv"
  test:
    executor: python/default
    steps:
      - checkout
      - restore_cache:
          key: deps1-{{ .Branch }}-{{ checksum "requirements.txt" }}
      - run:
          name: Install Python deps in a venv
          command: |
            python3 -m venv venv
            . venv/bin/activate
            pip install -r requirements.txt
      - run:
          name: "Run tests"
          command: |
            pip install -r requirements.txt
            python3 test.py
      - save_cache:
          key: deps1-{{ .Branch }}-{{ checksum "requirements.txt" }}
          paths:
            - "venv"
  deploy:
    machine: true
    steps:
```

Pylint
Star your Python code!

- ✓ The next step in our CI/CD pipeline tutorial is testing. Our tests in this project are run with the unit test framework
- ✓ Running tests on every commit is crucial to a project's success

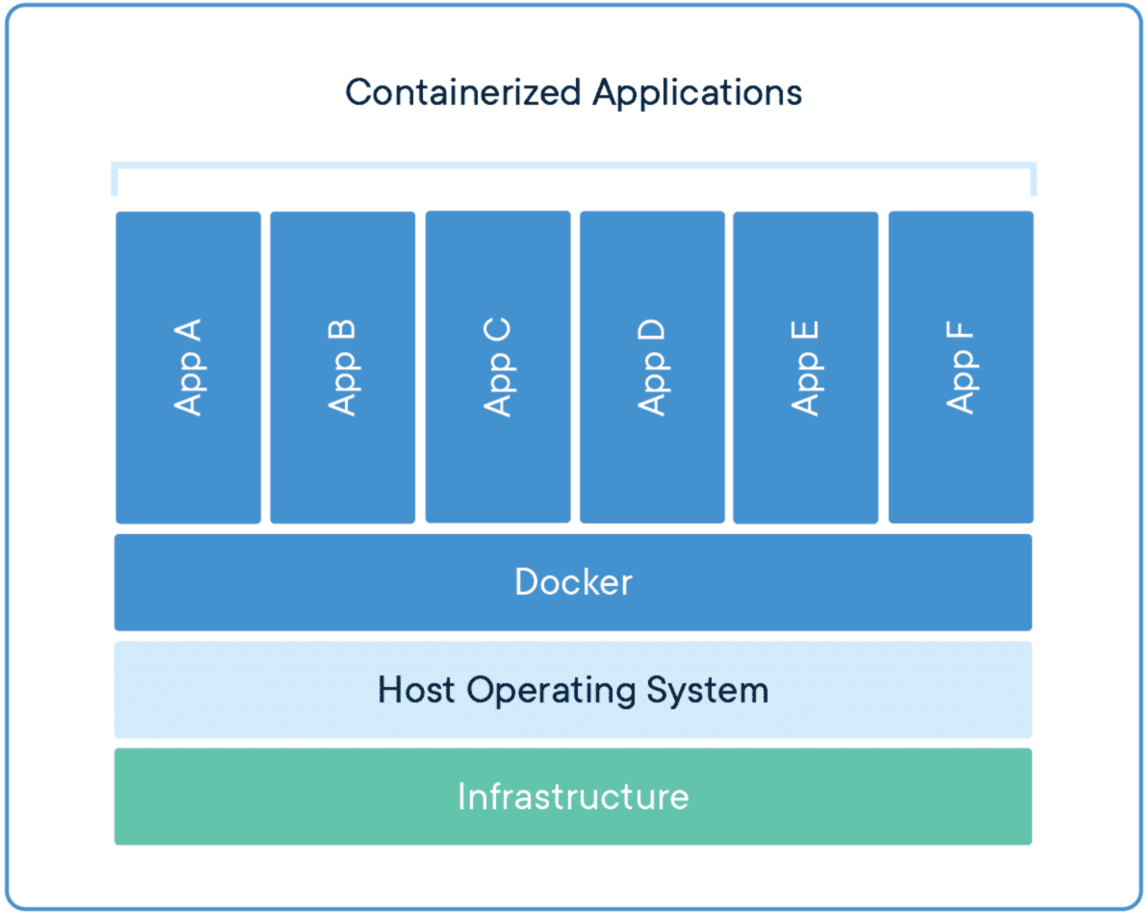
BUILD STEP:

```
64        - lint
65        - test
66        - docker/publish:
67            deploy: false
68            image: $CIRCLE_PROJECT_USERNAME/$CIRCLE_PROJECT_REPONAME
69        - deploy:
70            requires:
```

- ✓ Lines 66-68 reference the Docker orb and define how the Docker job will run. Set the `deploy` attribute to `false` to instruct the Docker/publish job to build the image without pushing it to a repository. By default, the Docker/publish job finds the Dockerfile by name and builds it. It will also fail the job if the Docker build fails.
- ✓

DOCKER

- an open platform for developing, shipping, and running applications
- separate your applications from your infrastructure
- significantly reduce the delay between writing code and running it in production
- container



DEPLOY STEP:

config.yml — simple-flask-app

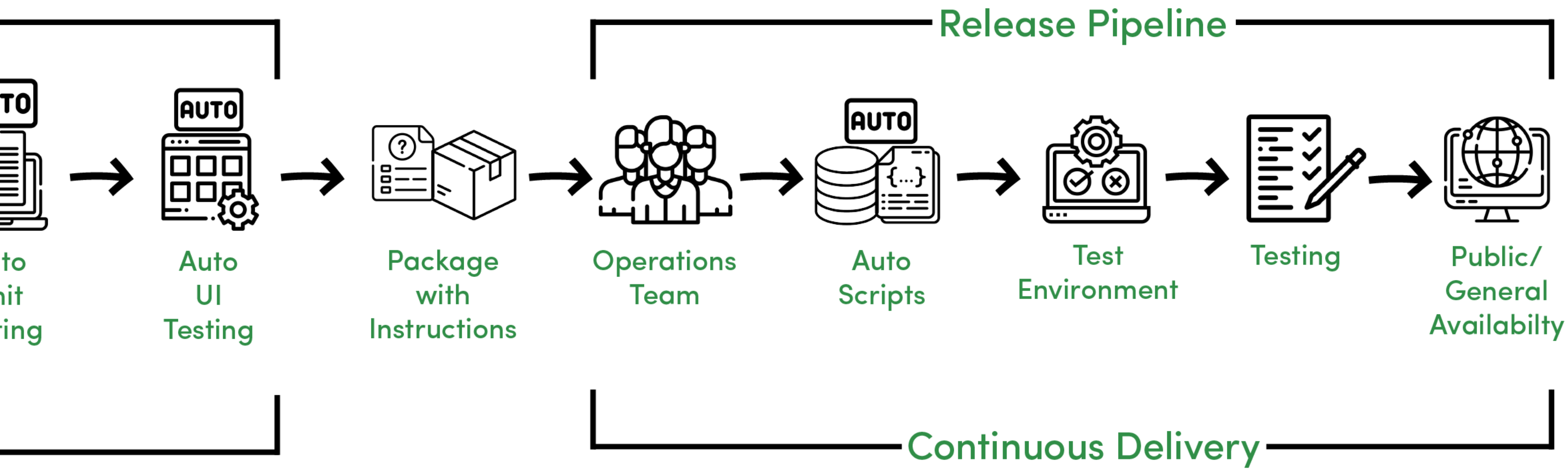
! config.yml × | Dockerfile | app.py | test.py | requirements.txt

.circleci > ! config.yml

```yaml
            python3 test.py
      - save_cache:
          key: deps1-{{ .Branch }}-{{ checksum "requirements.txt" }}
          paths:
            - "venv"
  deploy:
    machine: true
    steps:
      - checkout
      - run:
          name: Build and push Docker image to Heroku
          command: |
            sudo curl https://cli-assets.heroku.com/install.sh | sh
            HEROKU_API_KEY=${HEROKU_TOKEN} heroku container:login
            HEROKU_API_KEY=${HEROKU_TOKEN} heroku container:push -a grasbergerm-simple-flask-app web
            HEROKU_API_KEY=${HEROKU_TOKEN} heroku container:release -a grasbergerm-simple-flask-app web
workflows:
  lint-test-build-deploy:
    jobs: 
      - lint
```

https://www.techtarget.com/searchitoperations/tip/CI-CD-tutorial-How-to-set-up-a-pipeline

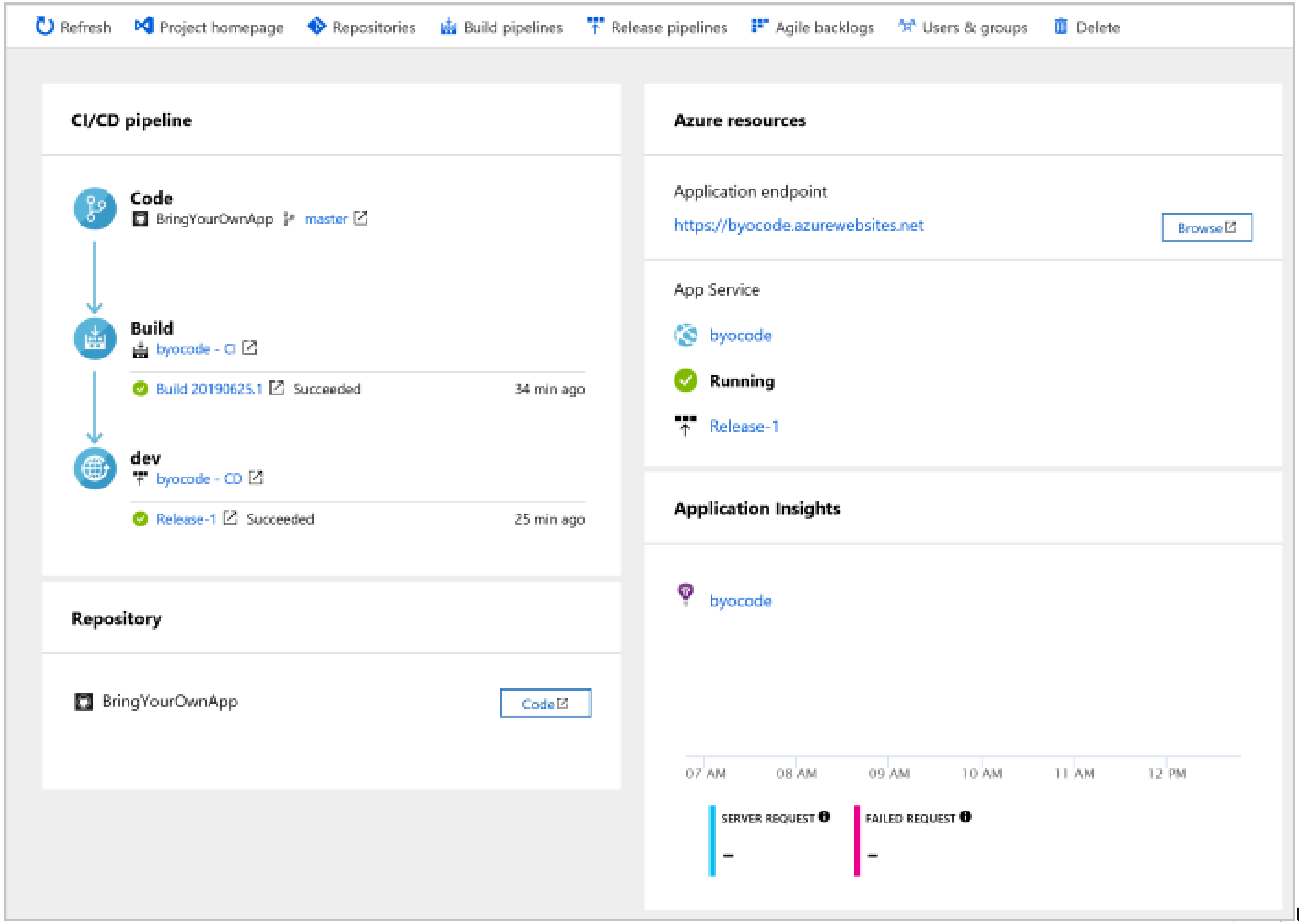
CONTINUOUS DELIVERY



CONTINUOUS DELIVERY

- ✓ Teams produce software in short cycles, ensuring that the software can be reliably released at any time and, when releasing the software, without doing so manually.
- ✓ Continuous delivery is an extension of continuous integration since it automatically deploys all code changes to a testing and/or production environment after the build stage.

OTHER SOLUTIONS FOR CI/CD ....

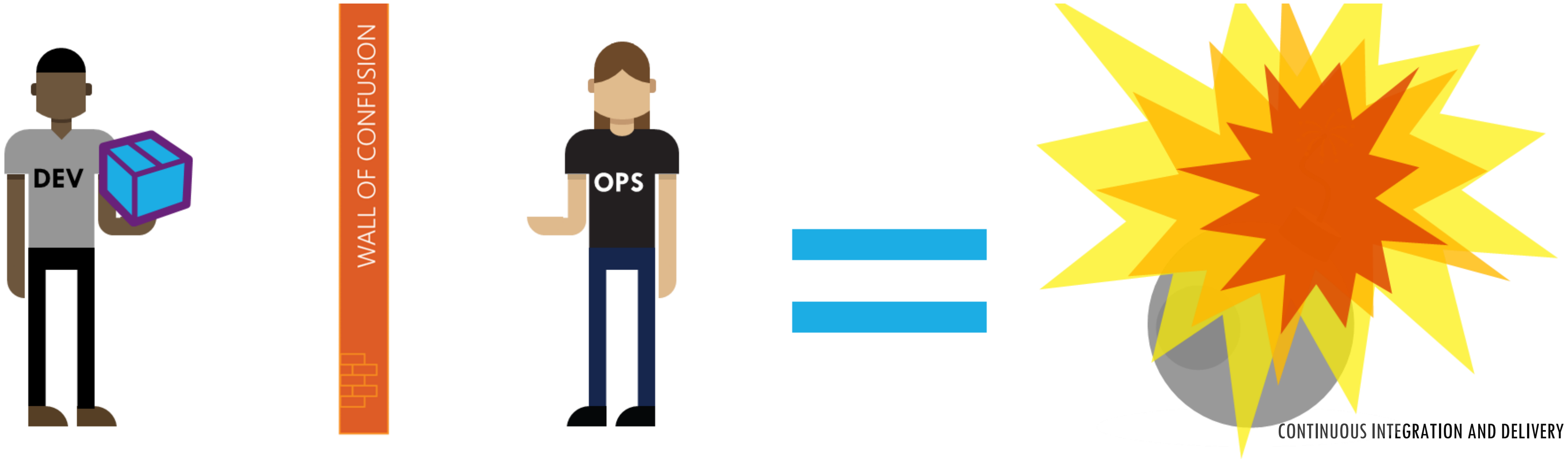


COMMON PITFALL OF CI/CD

- ✓ Wrong processes may be automated first
- ✓ Confusion between Continuous Deployment and Continuous Delivery
- ✓ Inadequate coordination between continuous integration and continuous delivery
- ✓ Meaningful dashboards and metrics may be absent
- ✓ Requires new skillset
- ✓ Maintenance is not easy

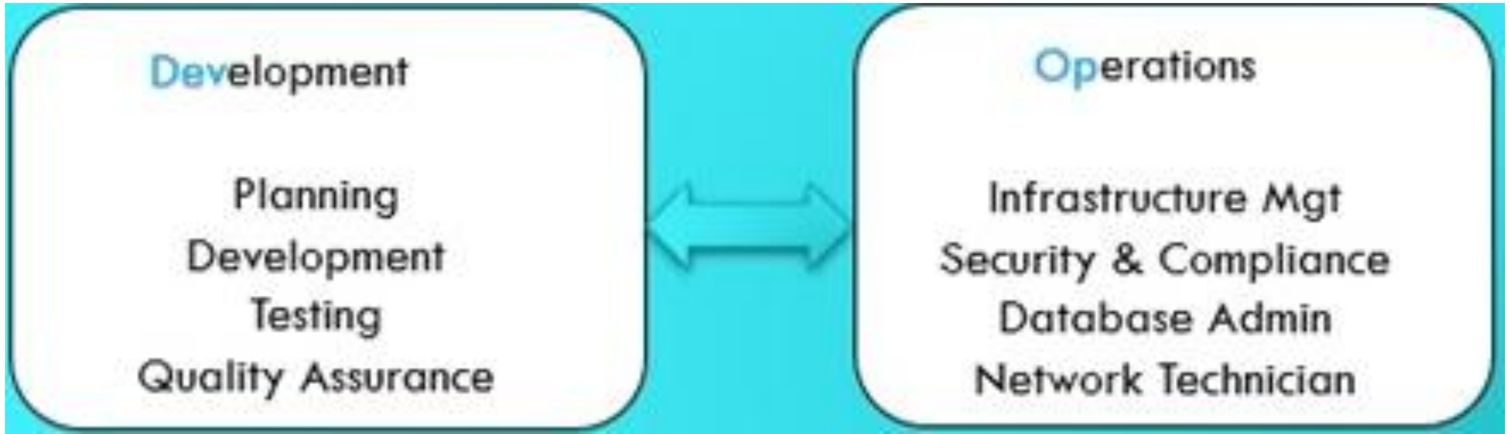
DEVOPS

✓ a set of practices that combines software development (Dev) and IT operations (Ops)



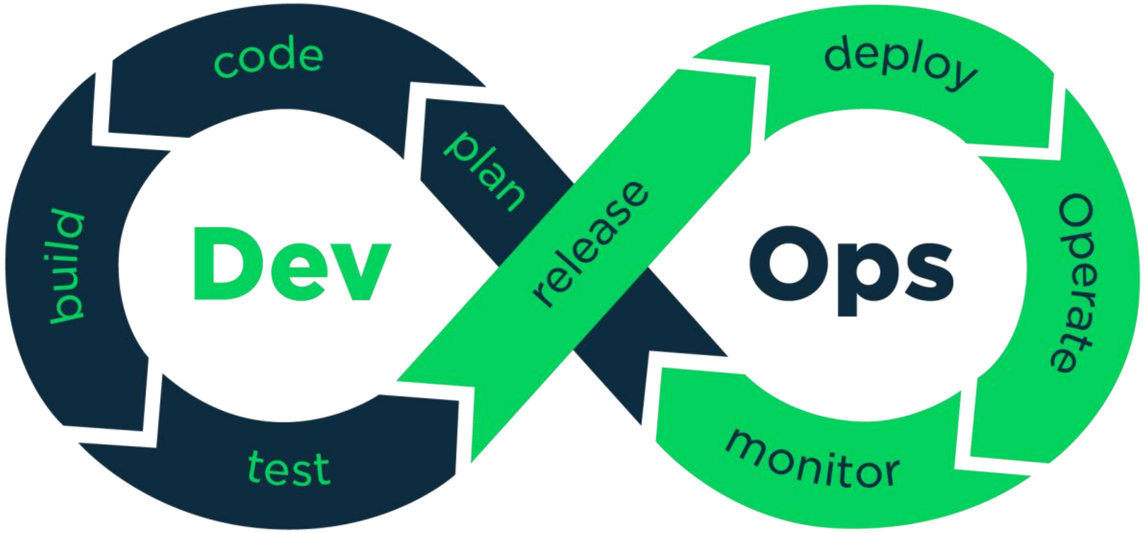
DEVOPS

- ✓ a set of practices that combines software development (Dev) and IT operations (Ops)



DEVOPS

- a set of practices that combines software development (Dev) and IT operations (Ops)
- Breaking the Silos: Dev and Ops
- aims to shorten the systems development life cycle and provide continuous delivery with high software quality



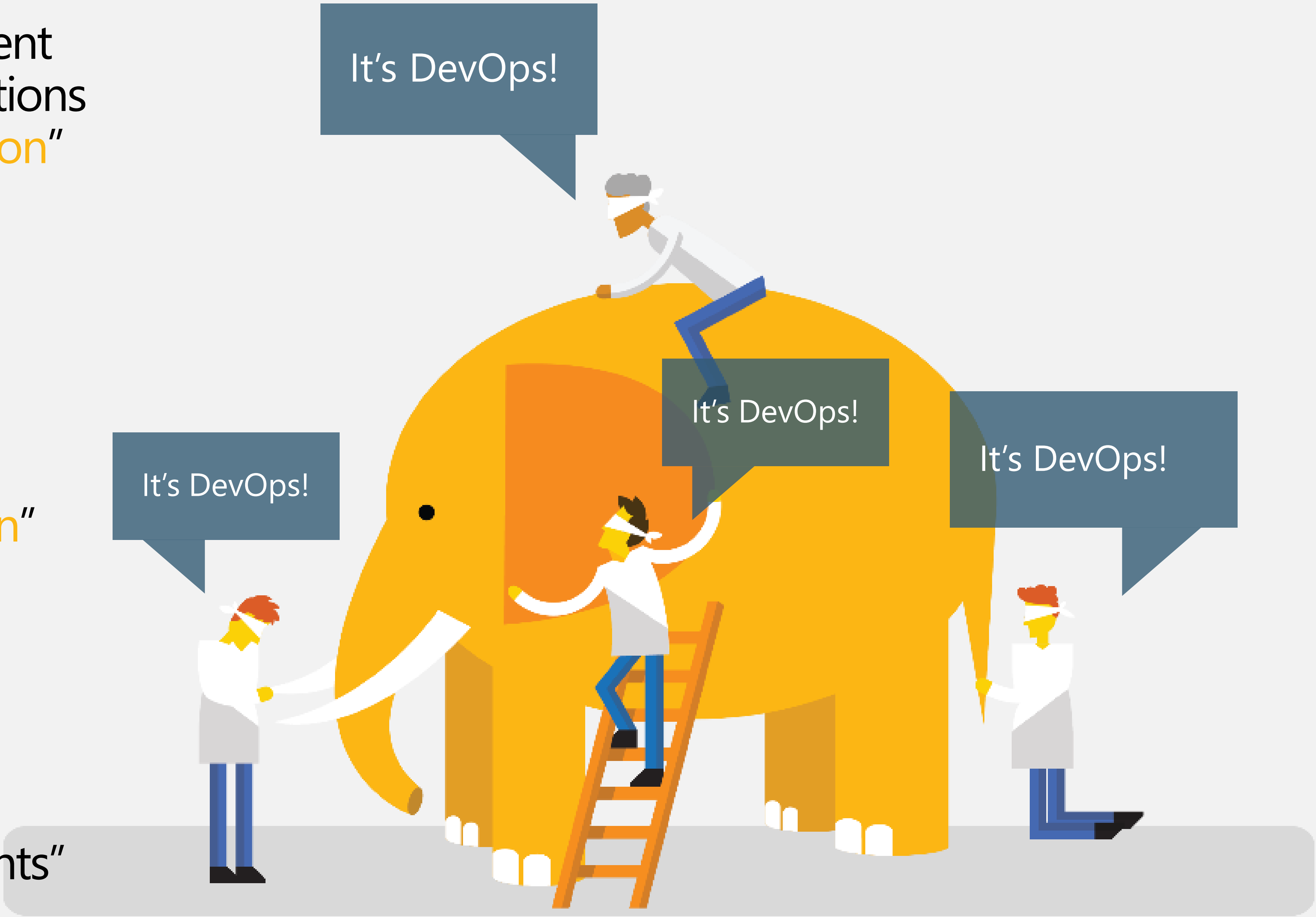
DEVS AND OPS WORKING TOGETHER

- ✓ Create feedback loops between inventors and mechanics
- ✓ Expose real-time metrics from ops enabling dev to learn from the system running under real world conditions
- ✓ Expose real-time metrics from dev enabling ops to anticipate production needs and provide early input
- ✓ Cross-functional teams collaborate to deliver whole working systems including all infrastructure, software code, and configurations

"**DevOps** is development and operations collaboration"

"**DevOps** is using automation"

"**DevOps** is small deployments"

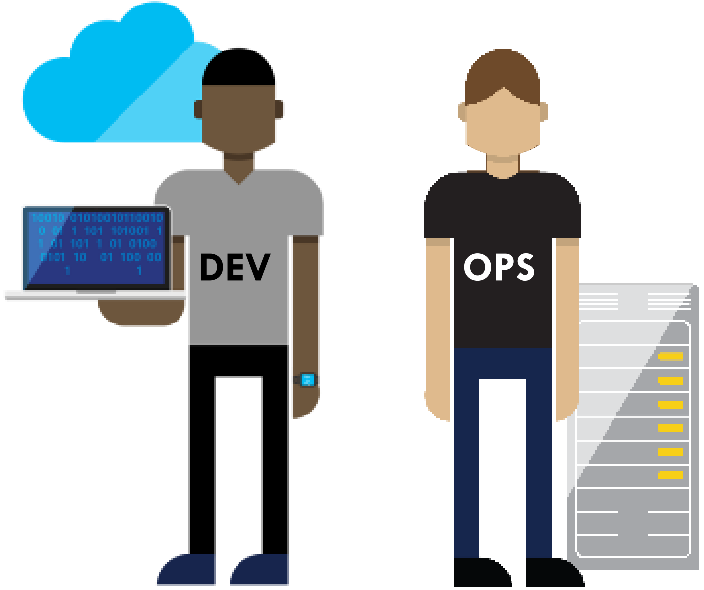


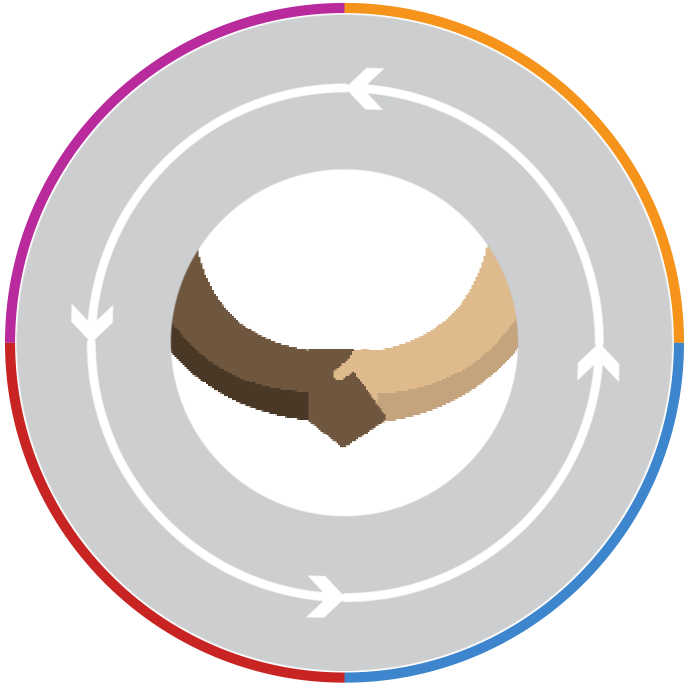
"**DevOps** is treating your infrastructure as code"

"**DevOps** is feature switches"

"Kanban for Ops?"

DEVOPS: THE THREE STAGE CONVERSATION





1	People
2	Process
3	Products

LIST OF DEVOPS PRACTICES

- Infrastructure as Code (IaC)
- Continuous Integration
- Automated Testing
- Continuous Deployment
- Release Management
- App Performance Monitoring
- Load Testing & Auto-Scale

- Availability Monitoring
- Change/Configuration Management
- Feature Flags
- Automated Environment De-Provisioning
- Self Service Environments
- Automated Recovery (Rollback & Roll-Forward)
- Hypothesis Driven Development
  - Testing in Production
  - Fault Injection
  - Usage Monitoring/User Telemetry

http://www.itproguy.com/devops-practices/

GitHub
JetBRAINS Develop with pleasure!
LeanKit
AppVeyor
slack
Octopus Deploy
azuqua
myget
SOASTA
Jenkins
HU-BOT
Atlassian HipChat
MODERN Requirements By eDevTECH
Tasktop
Aha!
Trello
dynatrace
zendesk

Visual Studio Partners and Extensions

65
Visual Studio Code Extensions

5,910
Visual Studio Gallery Extensions

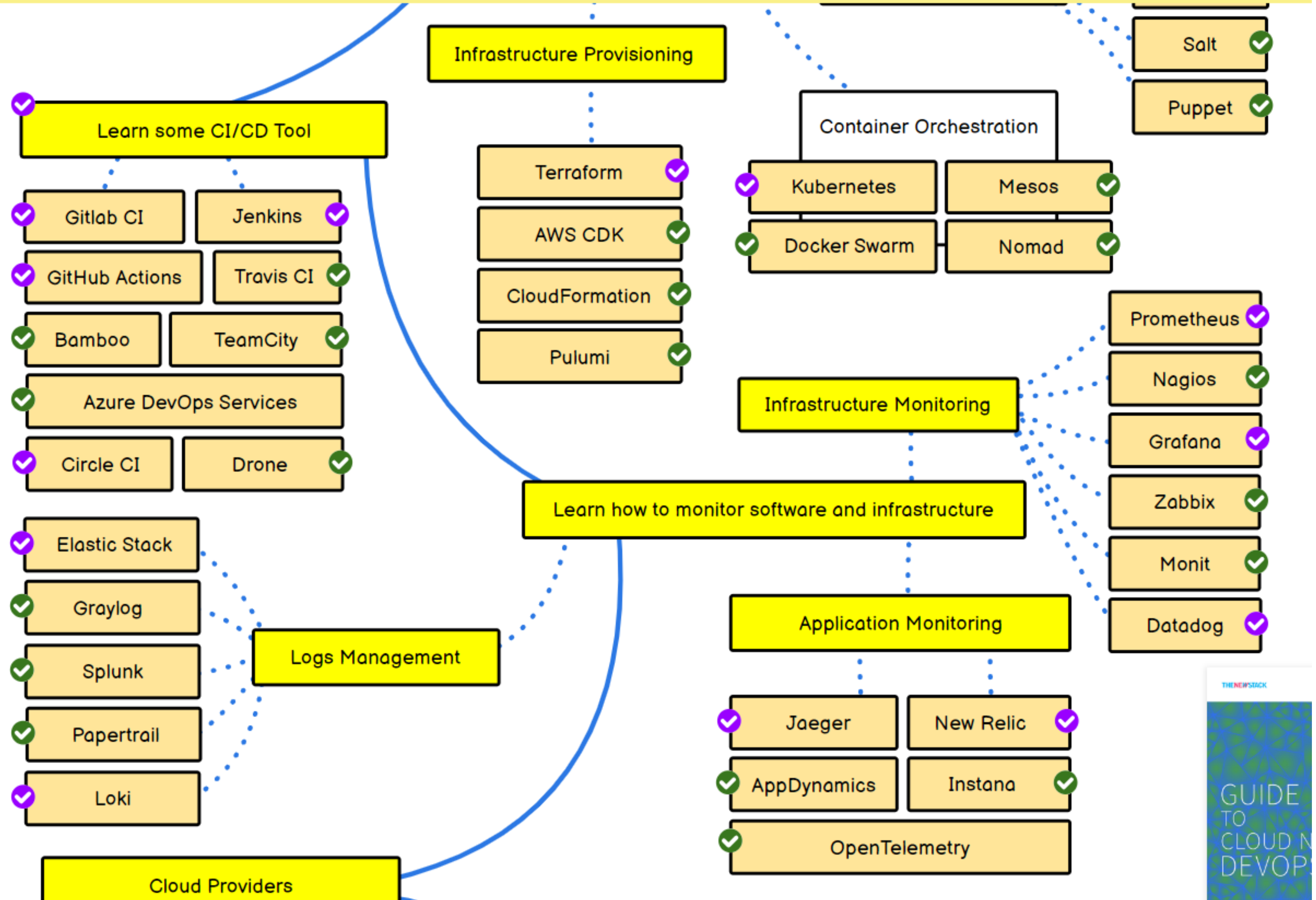
90
Visual Studio Sim-Ship Partners

48
VS Team Services Extensions

IncrediBuild BEYOND ACCELERATION
CloudBees
zapier
uservoice
flowdock
bamboo solutions
macincloud
VS Anywhere
Xamarin
jamo solutions
perfecto mobile
TFS TIMETRACKER
redgate ingeniously simple
PreEmptive Solutions
CHEF
submain
EKOBIT
OpsHub Enabling Agile Integration
Campfire
sauce LABS
BK TP.HCM

A BETTER VIEW

Step by step guide for DevOps, SRE or any other Operations Role in 2022



Q&A