Zurich University
of Applied Sciences

School of
Engineering
InIT Institute of Applied
Information Technology

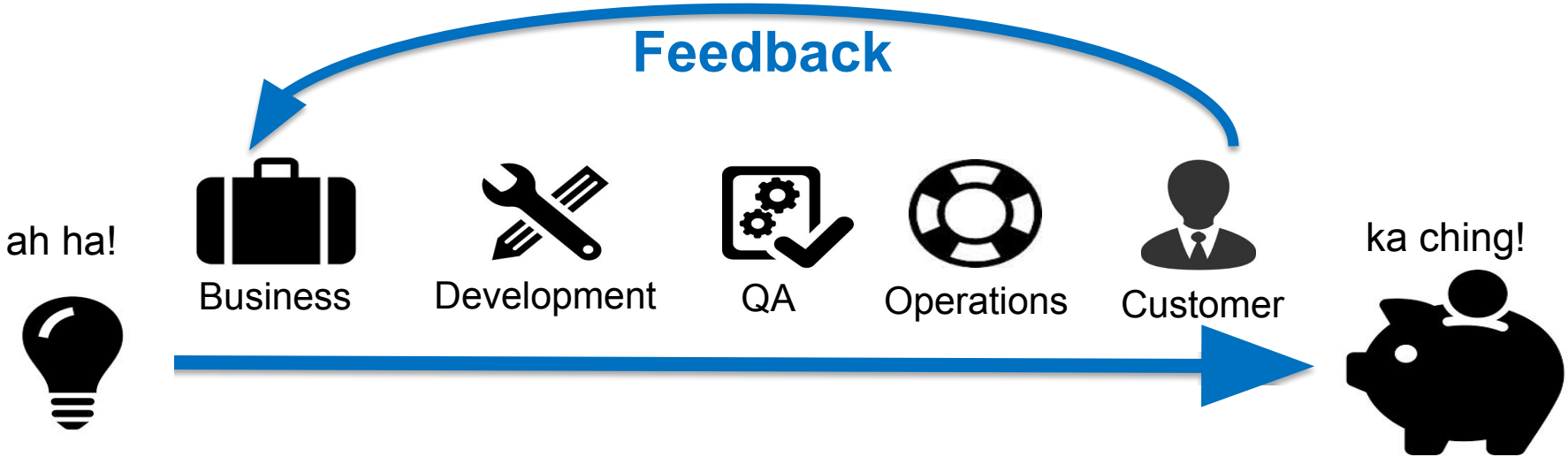
DVOP2 - DevOps, Continuous Deployment

Prof. Dr. Thomas M. Bohnert
Christof Marti

Content

- Continuous Delivery / Deployment Intro
- Software Automation Pipeline
- Tooling for Continuous Delivery / Deployment
- CI/CD Pipelines
- Deployment Strategies / Pattern
- Zero Downtime State-Transition patterns

The IT Value Stream



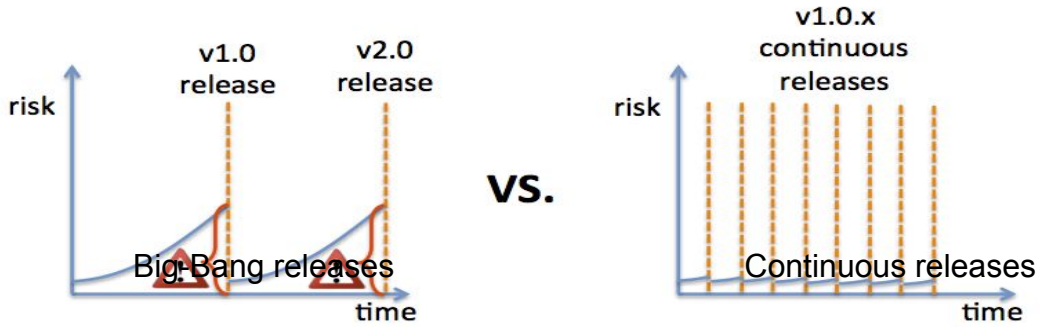
Customer Expectations

- Consumer Expectation
  - constant drumbeat of feature and function delivery
  - quality
  - zero installation headaches and upgrades

→ How to continuously deliver and deploy in quality?

Continuous means, to deploy frequently and in smaller increments

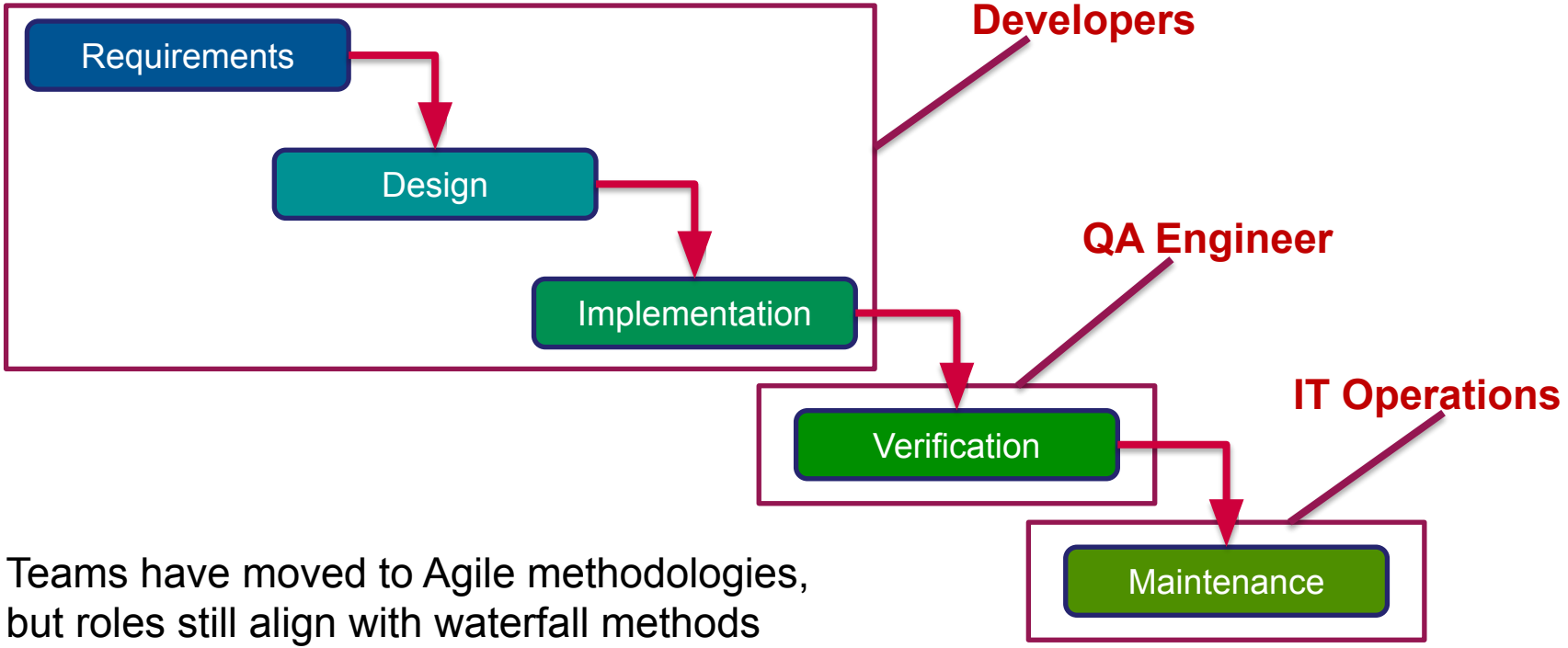
- Faster Time To Market
- Minimize Risks
- Improve Product Quality



Main Goals of Continuous Delivery / Deployment

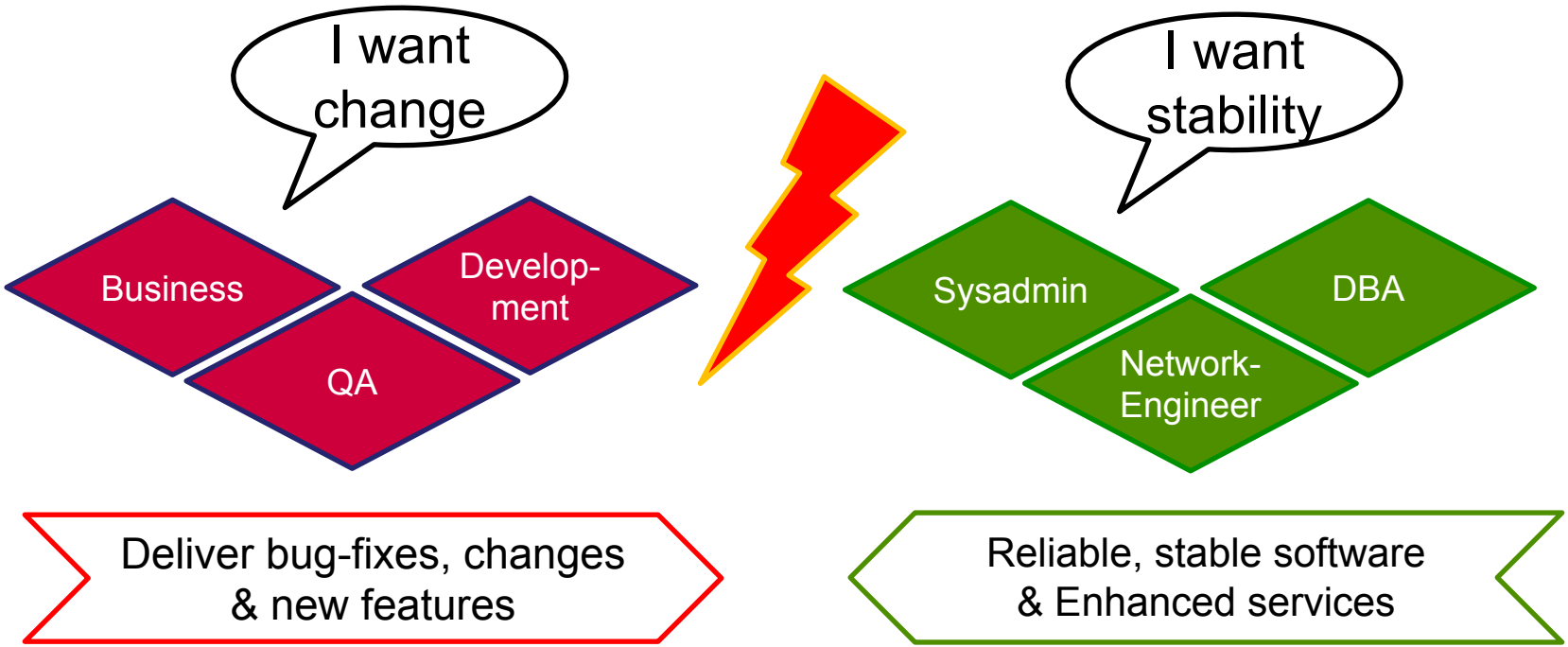
- Faster Time To Market
  - Accelerate the process from idea to deployment
  - Immediate feedback
  - Shorter innovation cycle
- Minimize risks
  - only small changes
  - prove that software is building
  - find broken build fast and early
  - awareness of current software status
  - eliminate dependencies on key personnel
- Improve Product Quality
  - automated testing & automated code auditing
  - documented history of builds to verify issues

Many Organizational Silos are still Waterfall

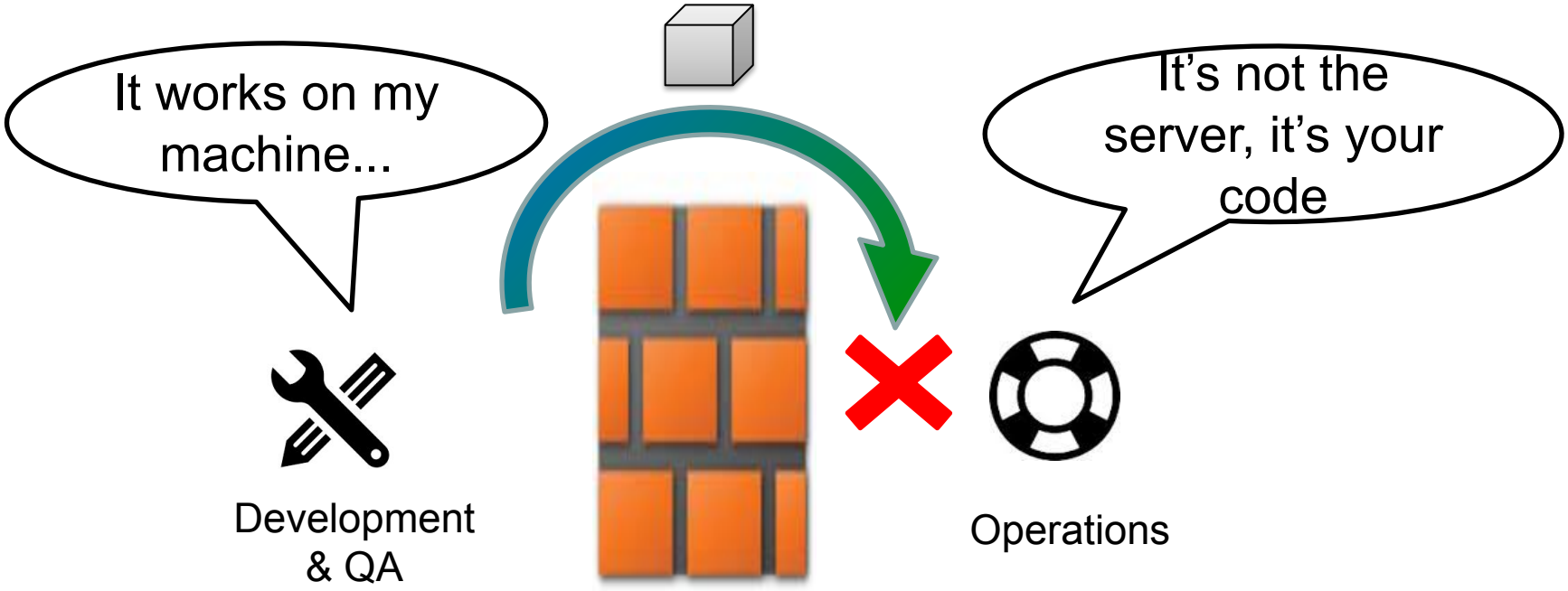


Teams have moved to Agile methodologies,
but roles still align with waterfall methods

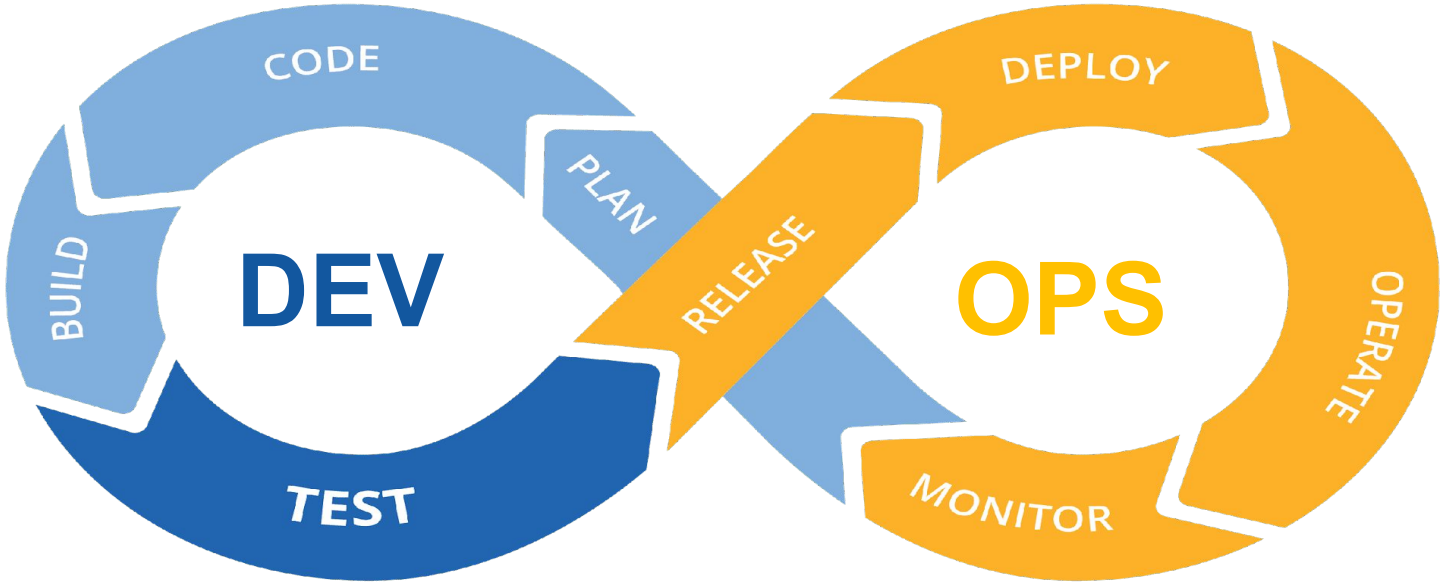
Conflicting Interests & Incentives



Wall of confusion

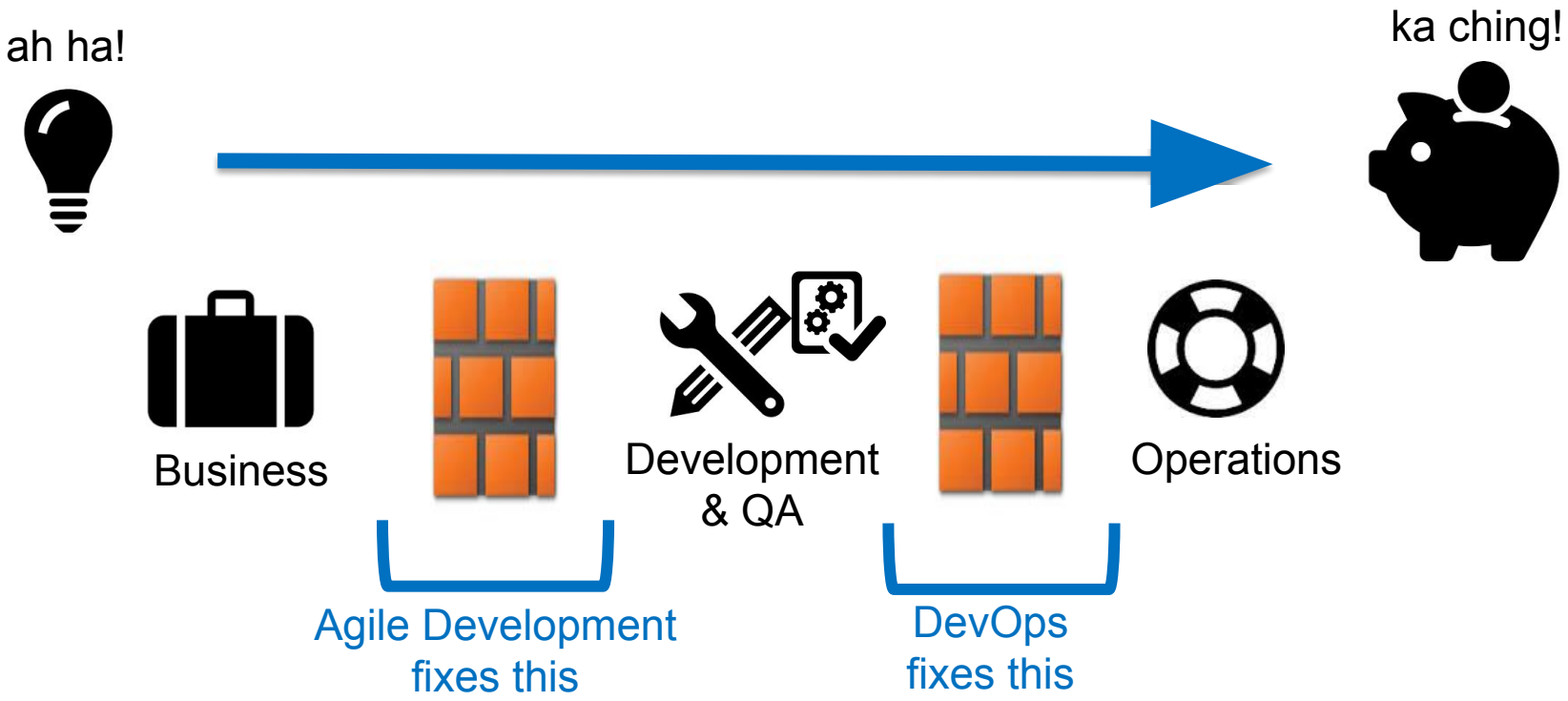


Continuous Life-Cycle between Development and Operations



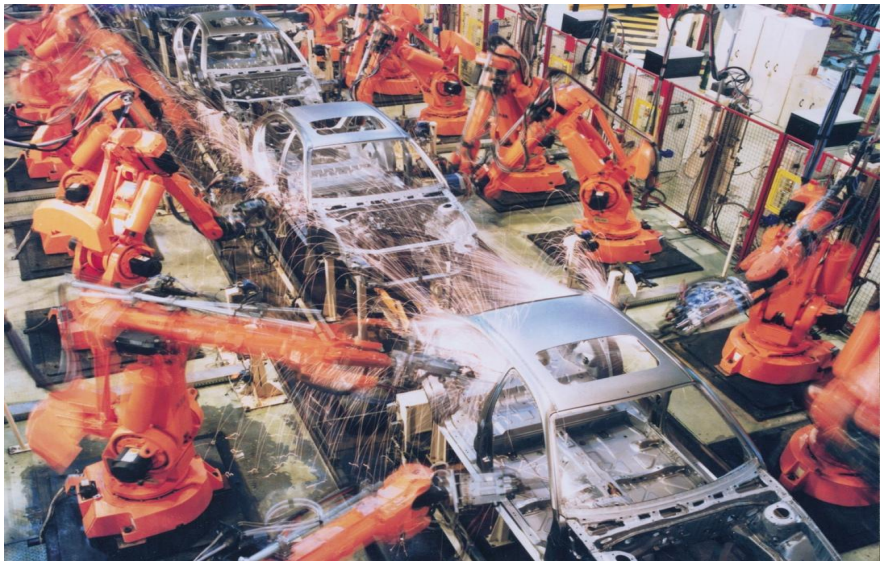
To evolve your application during its lifetime the Software Automation Pipeline is executed continuously for each new feature, bug fix or configuration change.

Approach - Tearing down walls of confusion

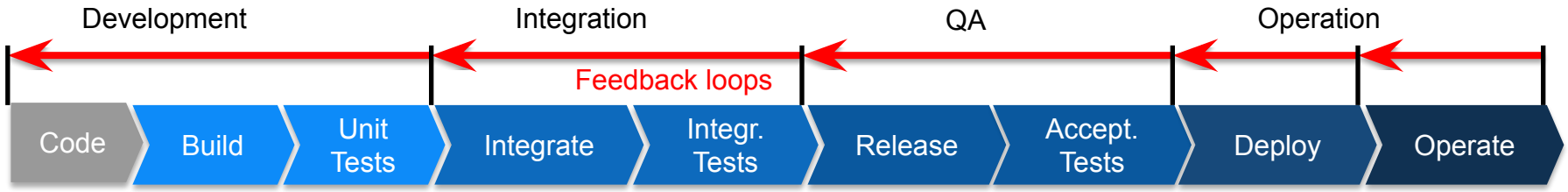


Main driver: Automation

- Machines are really good at doing the same task over and over
- Consistent and Known State
- Fast and Efficient
- 5 mins/day → 2.6 days/year
- What can be automated?
  - Builds
  - Deployments
  - Tests
  - Monitoring
  - Self-healing
  - System rollouts
  - System configuration
  - …



Software Automation Pipeline

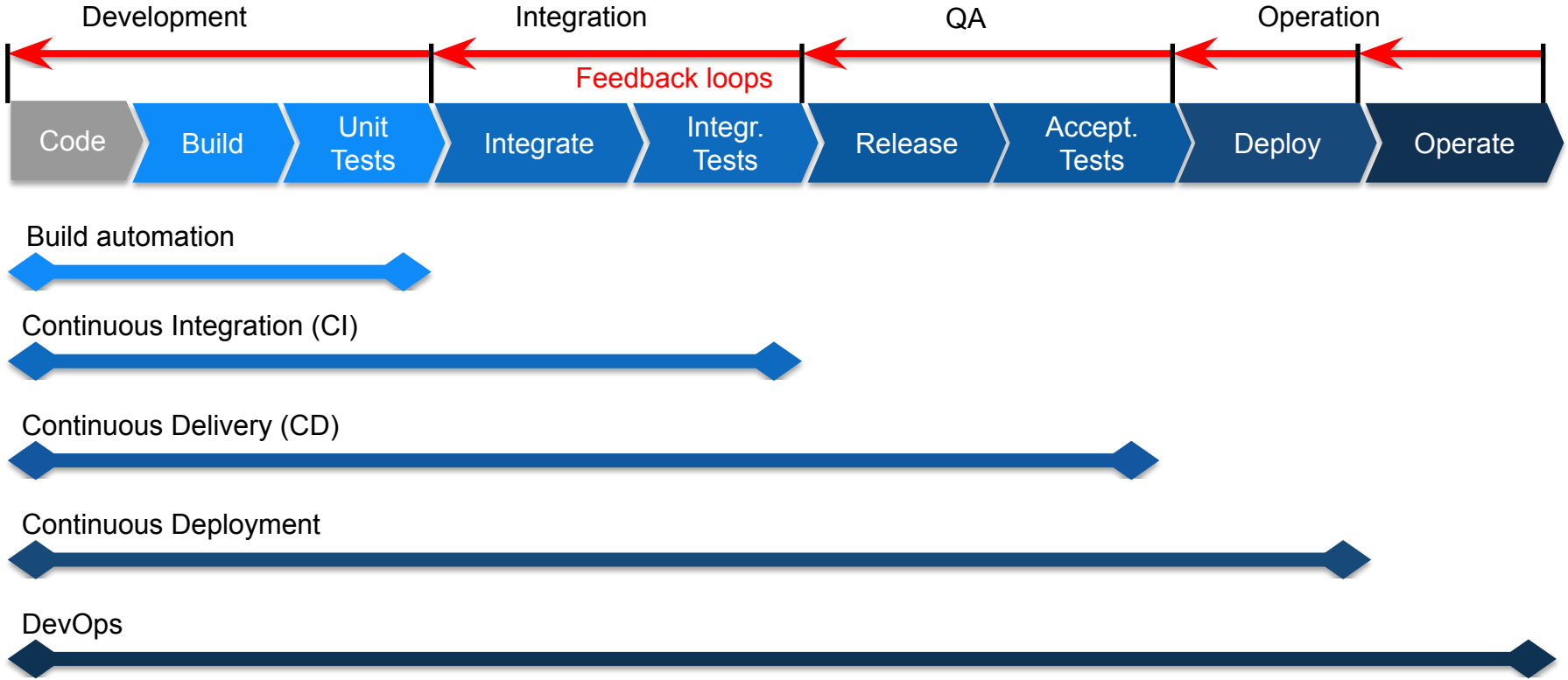


- Automation can be implemented in every stage of the software process from Development to Integration, QA to Operation.
- Every step can only continue (automatically) if it passes several tests successfully.
- Otherwise the responsible people are notified about the failure and the process is halted until fixed → Feedback loop

Increase confidence in production readiness

Phases of the Software Automation Pipeline

Zurich University of Applied Sciences
School of Engineering
InIT Institute of Applied Information Technology



Phases of the Software Automation Pipeline

- **Build automation**
  - **Building** individual components and run **unit tests**
  - Typically run by the developer on his local machine
- **Continuous Integration**
  - Automatically **build, test and integrate** components and run Integration Tests (Code auditing, Security tests, Database tests, UI tests, …).
  - Typically run on Continuous Integration Server
- **Continuous Delivery**
  - Also **create releases**, deploy to staging environment and run automatic **acceptance** tests (Stress test, Load Tests, Compliance tests,…).
  - Ready for production, but deployment still requires a manual step.
- **Continuous Deployment**
  - Automatically **deploy** to production after successful passing acceptance tests.
- **DevOps**
  - Automatically run the operation of the production system (configuration management, infrastructure provisioning, backup, monitoring, automatic health management, scaling, …)

Build Automation

Typical tasks

Build automation covers the build phase of the SW lifecycle

- **Resolve dependencies**
  - Obtain required versions of 3rd party libraries
  - Identify build order
- **Compile** source code
- Run unit **test**
- **Package** Software
- Deploy to Artifact repositories
- Create **documentation**
- **Clean up** temporary files

Tools

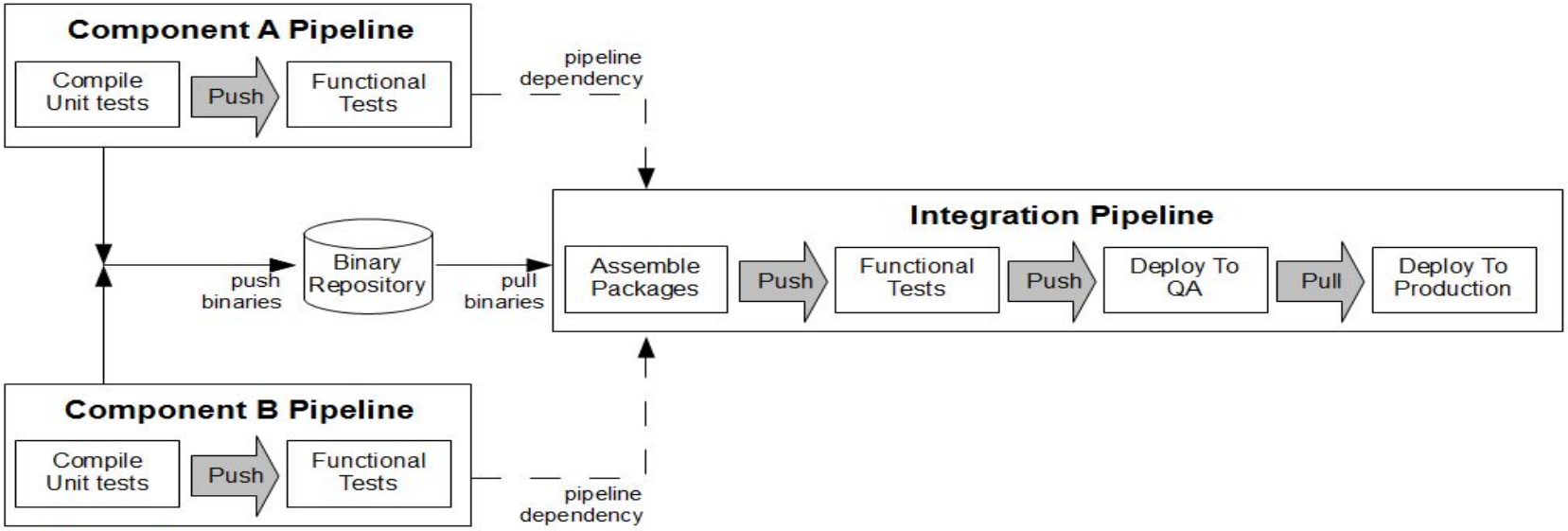
Depending on your environment different tools are very common

- C, C++, … → make, gmake, SCons, distcc,…
- Python → Waf, Snakemake, …
- Ruby → Rake,…
- JavaScript/Node.js → Grunt, Gulp, Bower,…
- Microsoft Environment → MS Build, Visual Build, Psake,…
- Java, JVM languages → Apache Ant, Apache Maven, Gradle, sdt, …

Most tools work in / for multiple environments

Continuous Integration

Continuously integrate pushed individually developed components (from different developers/teams), assemble them to a complete application and run functional integration tests against it.



"Integration Pipeline"
© Adapted from Humble and Farley 2011

Continuous Delivery vs Continuous Deployment

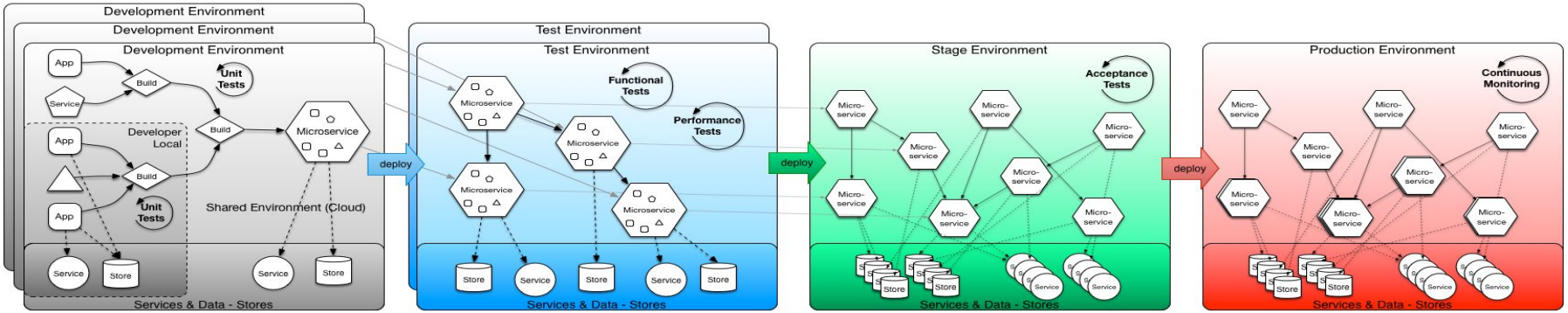
- In **Continuous Delivery**, a human manually deploys the release into the production environment.

- **Continuous Deployment** evolves the continuous delivery process one step further: every change that happens to pass automated tests is deployed to production automatically.

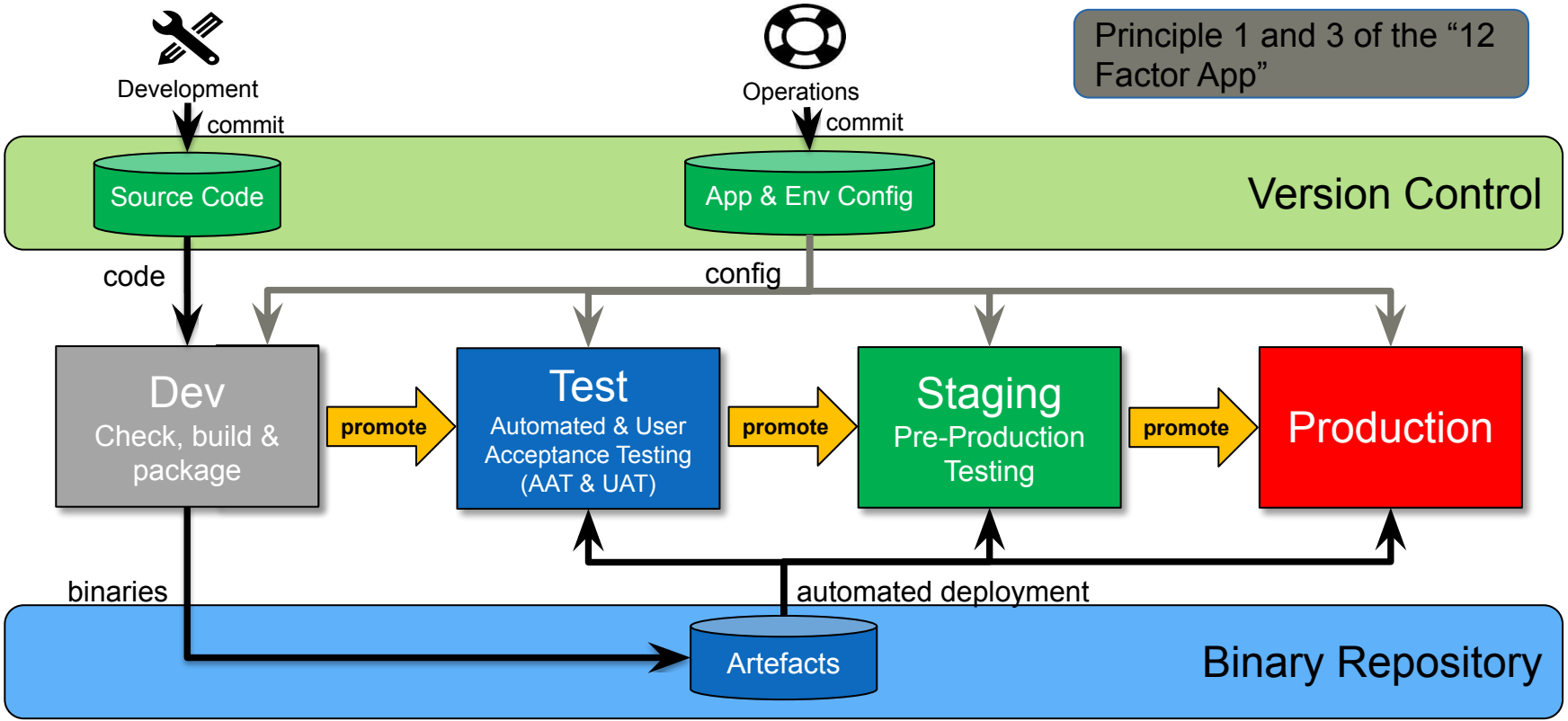
Multi-Stage Delivery and Environments

Provide different environments for each phase / stage of the process:

- **Development** (1+): Run the application per developer / team to develop specific features
- **Test** (1+): Run Integration, Functional and Performance Test in a dedicated test environment, close to the production environment.
- **Staging**: Exact copy of the production environment to run Acceptance and Operational Tests. Specifically also test the deployment process & scaling
- **Production**: Environment accessible to the end user containing real production data



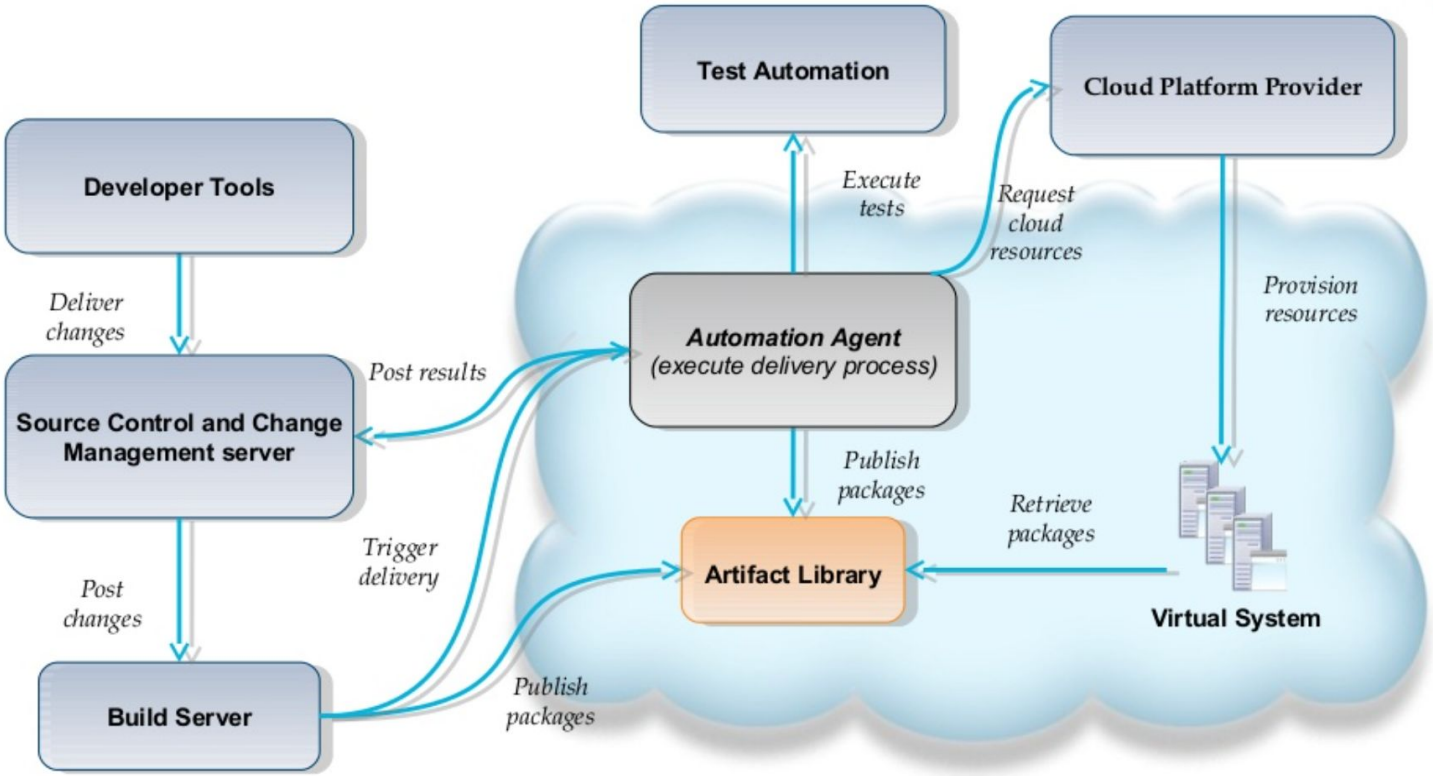
Flow through the Multi-Stage Pipeline



Best Practices Multi-Stage Pipelines

- Code (Dev) and Config (Ops) changes go always to version control
- Binary artifacts are only built once. Same artifacts are used in all environments
  → *no environment specific builds*
- Different Configs to support environment specific requirements provided by environment variables
- Same tooling is used to deploy to all environments
- **Typical Actions per Stage**
  - **Development**: Syntax check, code metrics, compile, unit tests, package
  - **Test**: Can be split into multiple stages. Is using stubbed or mocked data
    - AAT (Automated Acceptance) → Component/Integration-T, Feature/Story-level-T
    - UAT (User Acceptance) → UI-T, Usability-T, Showcase-T, Client-T
  - **Staging / Pre-Production**: Network-T, Capacity-T, Performance-T Smoke-T
  - **Production**: Post-Deployment-T, Smoke-T, Cont. Monitoring, Rollback & Re-Deploy

Continuous Delivery / Deployment Tooling and Systems



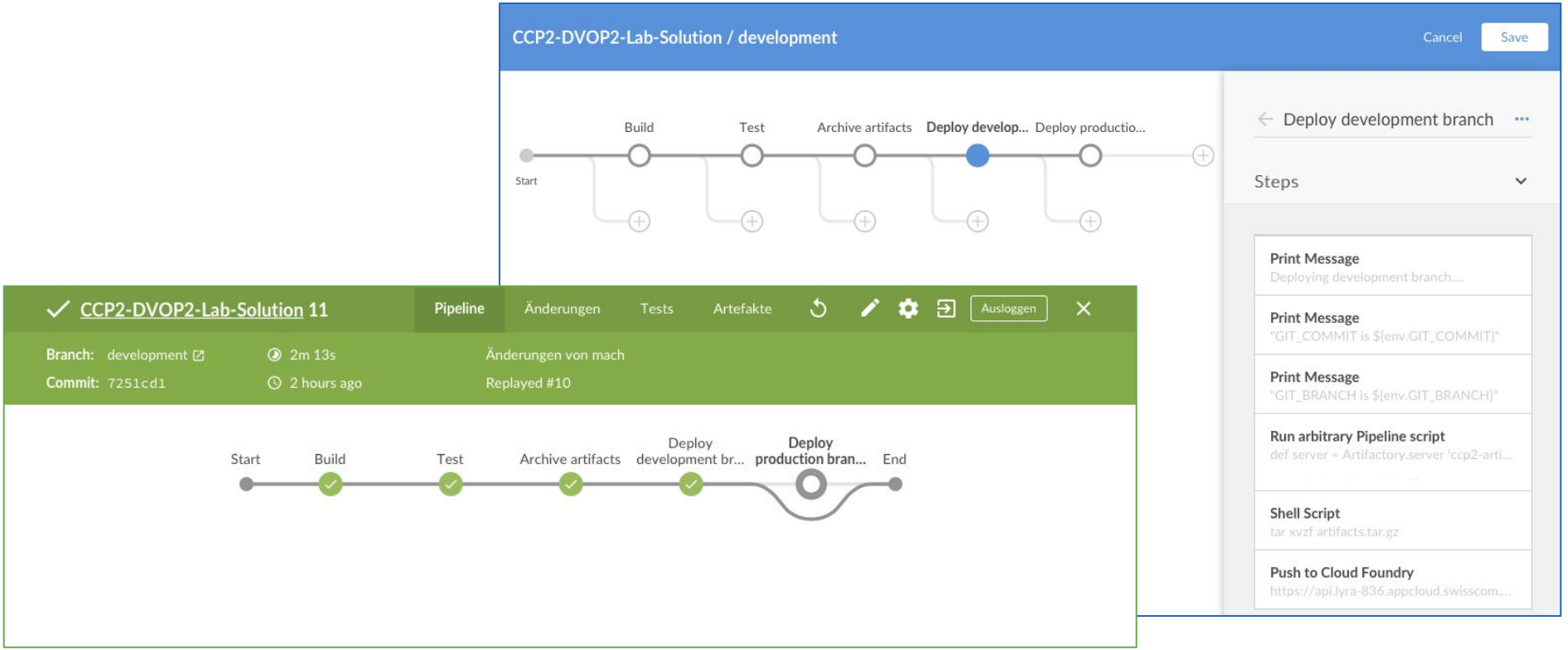
Tooling - Components

- **Version Control Server** (VCS) for Code and Configuration
  e.g. **Git**, Mercurial, SVN,…
- **Artifact Repository** management to store binary artifacts
  e.g. generic repositories like Jfrog Artifactory, Sonatype Nexus, GitHub Packages, GitLab, or type specific like (Docker) Image Registry, NPM
- **Build Server** to trigger and control the build, deploy and test tasks
  e.g. Jenkins-X, Travis-CI, GitHub-Actions, GitLab CI, **Tekton**, **Argo Workflows**,…
- **Automation Agent** to orchestrate the deployment process
  e.g. Shell-Scripts, Puppet, Chef, Ansible, Heat, CloudFormation, **ArgoCD**, **Flux**, …
- **Monitoring infrastructure** to get feedback from the runtime environment
  e.g. ELK (ElasticSearch-Logstash-Kibana), Graylog, Splunk
- **Secure-Store** to manage and provide security sensitive data like credentials and certificates e.g. HashiCorp Vault, Square Keywhiz, …

Tooling - Build-Server

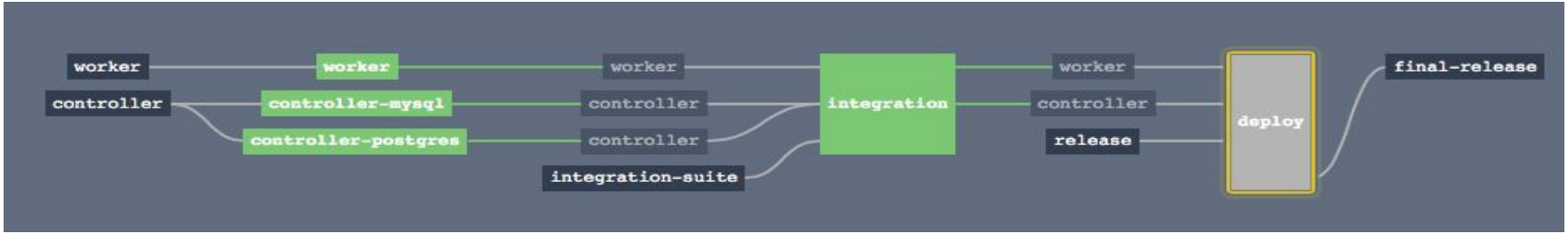
- Traditional CI-Servers (Jenkins, Travis, …)
  - Easy to configure individual tasks (e.g. check-out and test)
  - Very challenging to configure workflows (sequence of tasks) or pipelines
- CI-Servers with Workflow support
  - Jenkins Pipeline aka Blue Ocean (https://jenkins.io/projects/blueocean/)
  - Travis CI, GitHub Actions, GitLab CI, …
- Modern pipeline based build servers
  - Jenkins-X (https://jenkins-x.io/)
  - Tekton CD (https://tekton.dev/)
  - Argo Workflow (https://argoproj.github.io/argo-workflows/)
  - Concourse CI (http://concourse.ci)

Blue Ocean Pipeline UI and Editor



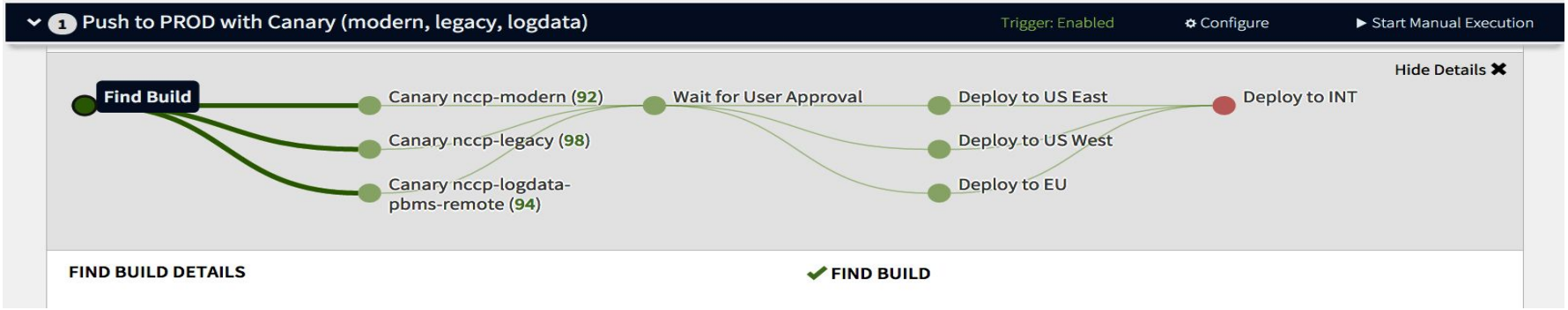
Tooling - Build-Server Workflow Examples

- Concourse CI (http://concourse.ci)



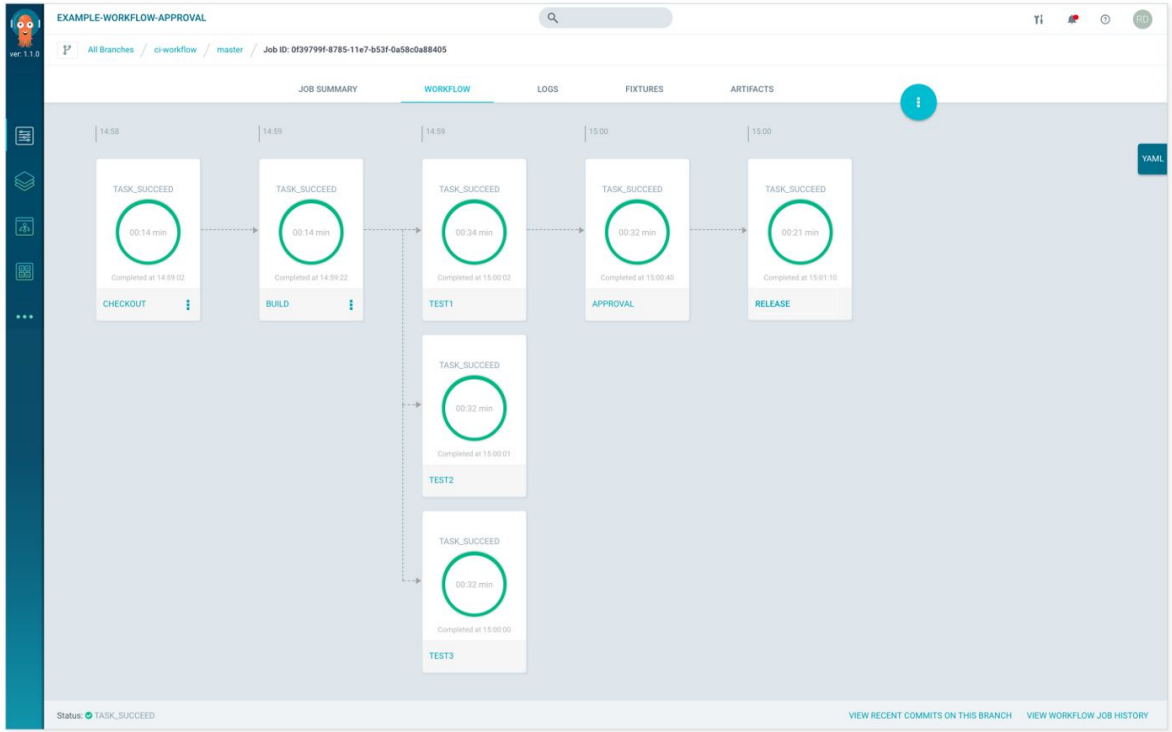
https://ci.concourse.ci/teams/main/pipelines/main?groups=develop

- Spinnaker CI (http://www.spinnaker.io)



Tooling - Build-Server Workflow Examples

- Argo Workflow (https://argoproj.github.io/argo-workflows/)



Jenkins

Jenkins

Traditional CI-Server

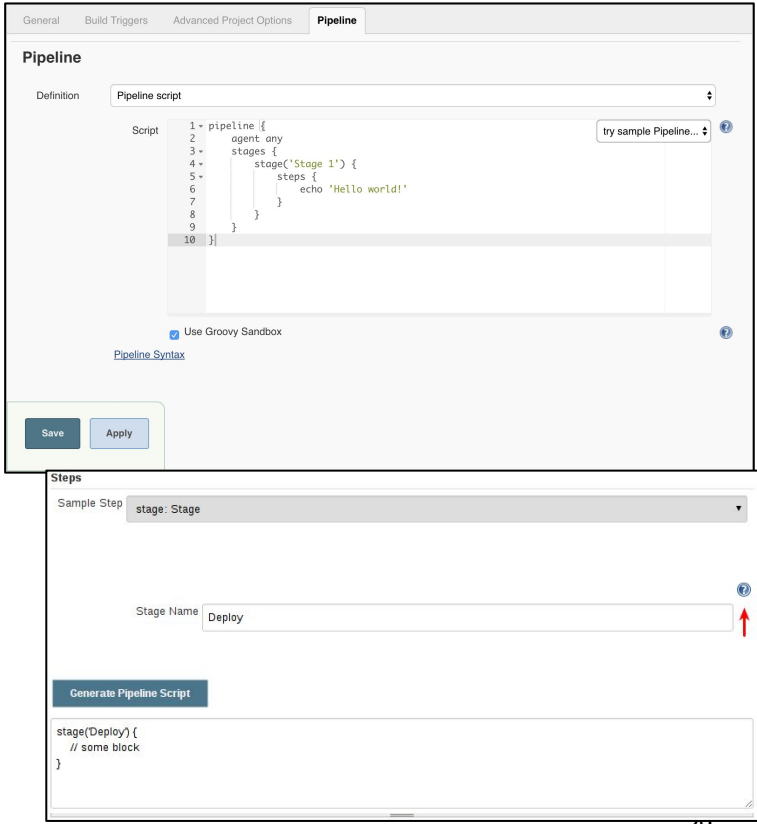
- Proven and well known in community
- Scalable using master nodes and agents
- Many plugins and integrations in dev tools
- New: Support for docker and kubernetes
- Newest releases (Jenkins 2+) support pipelines
- Modernized Blue Ocean UI for pipelines

Jenkins Pipeline Concepts

- **Pipeline** → user-defined model of a CD pipeline
  - Defines the entire build process, which typically includes stages for building an application, testing it and then delivering it.
- **Node** / **Agent** → Worker executing a Pipeline
  - Node is used in the Scripted syntax; Agent in the declarative
  - Stages can be executed on different agents (sequential or parallel)
- **Stage** → Conceptual distinct subset of tasks
  - Used to visualize progress
  - In declarative syntax the can be skipped using a when clause
  - Can run in parallel and on different agents
- **Step** → A single task to execute
  - Typically every command in is its own step (e.g. sh './gradle assemble')
  - Commands can be extended using plugins (e.g. pushToCloudFoundry())

Creating a pipeline

- Multiple ways to create Pipelines
  - Using the classic UI
    - edit text in browser
    - built in snipped generator
  - Using BlueOcean Editor
    - Automatically reads Jenkinsfile and allows to edit
    - Needs write permission to update Jenkinsfile in repository
    - May reorder your code and loose comments
  - Using Jenkinsfile in SCM (source control mgmt)
    - Jenkinsfile is automatically checked out and read from the repository root
    - Versioned together with project



Jenkinsfile – Basic structure (declarative)


```
pipeline {
  agent { label 'Linux' }
  options {
     timeout(time: 1, unit: 'HOURS')
  }
  environment {
    CF_CRED = credentials('init-cloudlab')
  }
  triggers {
    cron('H */4 * * 1-5')
  }
  stages {
    stage ('Build') {
      steps {
        echo "Building..."
        sh './gradlew assemble'
      }
      post {
        failure {
          echo "Build failed"
        }
      }
    }
    ...
```

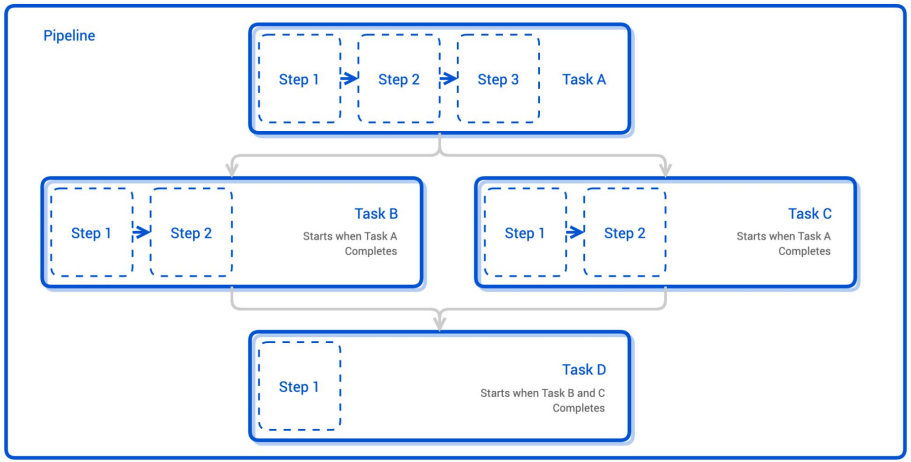
```
// begin of the declarative pipeline
// declare agents (default: any). (also possible in stage)
// options declare global behavior (timeout, retry,...)
// e.g. skipDefaultCheckout → do not checkout git repo on agent

// set global environment (also possible in stage)
   e.g. set CF_CRED_USR and CF_CRED_PSW vars from Jenkins credential

// when to trigger build (cron, pollSCM, ...) e.g. for nightly builds;
   often not needed → triggered using webhook from SCM

// start stages section (list of stages)
// first stage called 'Build'
// steps to execute in this stage
// each command is a single step


// commands to execute after the stage has completed
// depending on result of stage (success, failure, aborted, always,...)
```

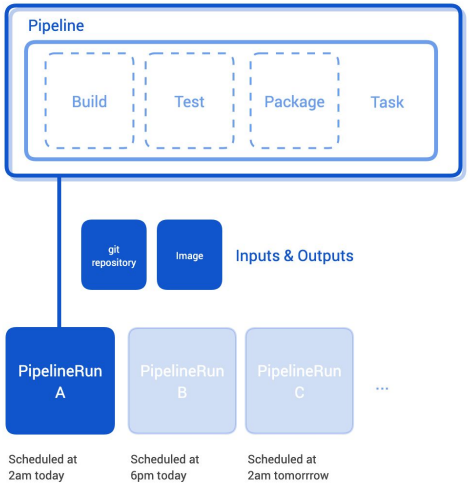
Jenkinsfile – Basic structure (declarative) cont.


```
    ...
    stage ('Deploy') {                                    // second stage called 'Deploy'
      when {                                              // stage is executed, if all conditions are true
        branch 'master'                                   // e.g. only on master branch
        anyOf {                                           //      and if DEPLOY_TO is set to 'production' or 'staging'
          environment name: 'DEPLOY_TO', value: 'production'
          environment name: 'DEPLOY_TO', value: 'staging'
        }
      }
      steps {
        echo "Deploy..."
        script {                                          // execute a script snipped
          def server = Artifactory.server 'ccp2-artifactory'    supports plugins who do not yet support declarative syntax
          def uploadSpec = """{
            "files": [{
              ...
            }]
          }"""
          server.upload(uploadSpec)
        }
      }
    }
  }
}
```

Tekton

- Cloud Native CI/CD Solution
- Runs on K8s and uses K8s Concepts
- Uses Custom Resource Descriptions (CRD) to define Resources like Pipeline, Tasks, …





Tekton Concepts

- **Step**: Operation on a workflow (compile, run test, package, create image, …)
  Each step runs in a specific container image
- **Task**: Sequence of **steps** executed in order.
  A task is run in a Pod, each step becomes a running container in the Pod
- **Pipeline**: Collection of Tasks run in a directed acyclic graph (DAG)
  May run tasks in parallel and sequence (fan-out/fan-in scenarios)
- **Inputs / Outputs**: Sources and Targets to Read / Store artefacts
  e.g. Git-Repositories, Image-Registry, …
- **PipelineRun & TaskRun**: Specific execution of a Pipeline or a Task
  e.g. Scheduled at specific intervals, triggered by events, …

All Resources are defined using CRDs

Tekton Examples


```
// tasks.yaml
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: hello
spec:
  steps:
    - name: echo
      image: alpine
      script: |
        #!/bin/sh
        echo "Hello World"
---
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: goodbye
spec:
  steps:
    - name: goodbye
      image: ubuntu
      script: |
        #!/bin/bash
        echo "Goodbye World!"
```

```
// pipeline.yaml
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: hello-goodbye
spec:
  tasks:
    - name: hello
      taskRef:
        name: hello
    - name: goodbye
      runAfter:
        - hello
      taskRef:
        name: goodbye
```

```
// pipline-run.yaml
apiVersion: tekton.dev/v1beta1
kind: PipelineRun
metadata:
  name: hello-goodbye-run
spec:
  pipelineRef:
    name: hello-goodbye


// task-run.yaml
apiVersion: tekton.dev/v1beta1
kind: PipelineRun
metadata:
  name: hello-goodbye-run
spec:
  pipelineRef:
    name: hello-goodbye
```

All resources can be deployed using: kubectl apply -f

```
kubectl apply -f task.yaml
kubectl apply -f pipeline.yaml
kubectl apply -f pipeline-run.yaml
kubectl logs pipelinerun hello-goodby-run -f -n default
[hello : hello] Hello World!
[goodbye : goodbye] Goodbye World!
```

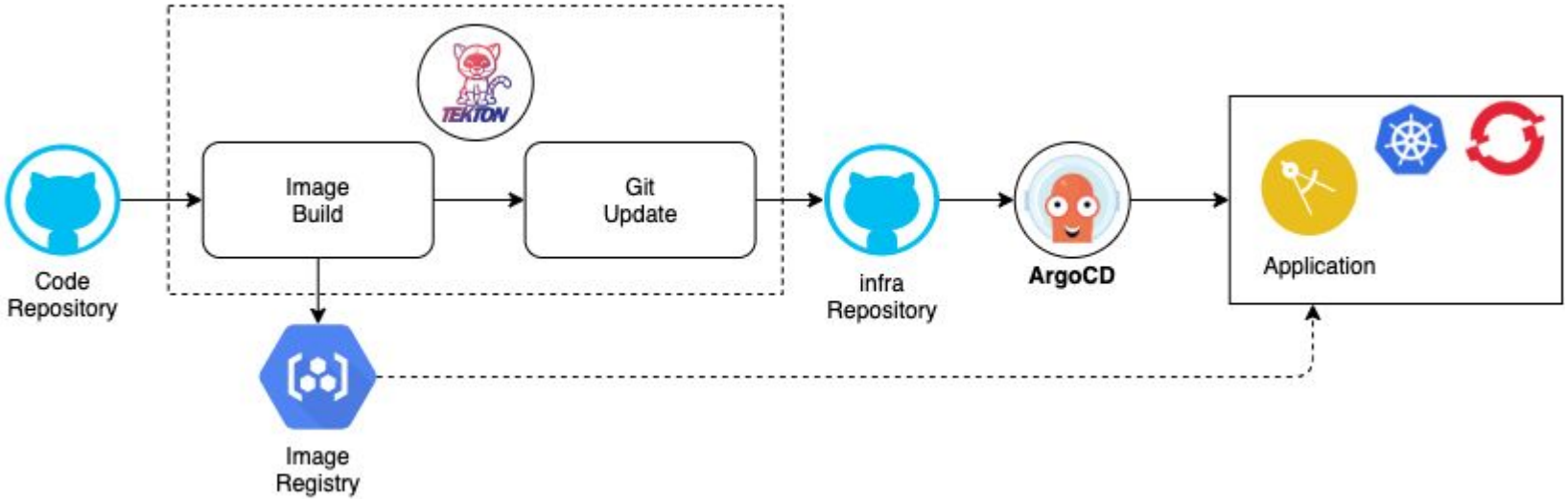
Tooling - Automation Agent

Process to run automation logic (on Build-Server or distributed)

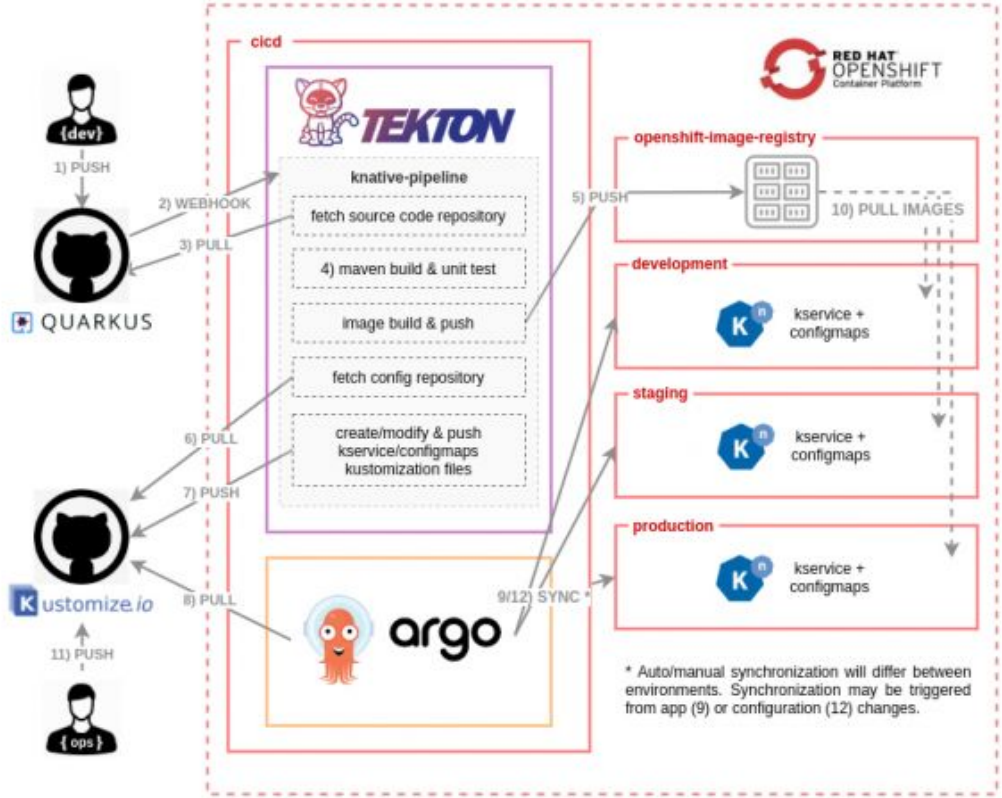
- ***Shell-Script frameworks* using CLI clients**
  - Interact with Tooling servers: git, mvn, docker, …
  - Deploy to Cloud
    - PaaS: CF → cf push, OpenShift → git push, GCP → gcloud app deploy
    - IaaS: AWS → aws, Google Cloud Compute → gcloud compute …
  - Works for small applications, too fragile for complex apps
  - Option to keep scripts simple and model the system structure in the CI/CD system (lock-in)
- ***Configuration management and orchestration tools* (Puppet, Chef, Ansible, …)**
  - Infrastructure focused, limited support for application platforms or migration
- ***Platform provided automation***
  - Some Deployment services exist, but not there yet for complex application structures
  - Requires to maintain redundant model of system structure → Lock-In

GitOps

- All "Operations" are triggered using Git-Events (Push, PR, Tag, …)
- Deployment-Configuration stored in Git-Repositor(ies)



GitOps Workflow in Openshift



Deployment Strategies

- Expectations on uptime and responsiveness are high
  - e.g. Cloud-Apps (SaaS) are in use 24x7
- Downtime is costly in terms of money, customers and credibility
- Goal: Zero-downtime deployment
- Key to zero-downtime deployment is decoupling the various parts of a release so it can happen as independently as possible
- There exist several proven deployment patterns, which can be used individually or in combination:
  - Feature flags / toggles
  - Blue-Green deployment
  - Canary deployment (Rolling update)

Feature Flags / Toggles

- New software features are deployed to production.
  They can be enabled / disabled at runtime based on configuration
- Depending on strategy features can be enabled for specific
  - users / role → test users, A/B tests
  - clients → IP based, Type web-/mobile-/fat, brand
  - instances → canary deployment
  - geo-locations → coordinated rollout
  - data schema / API versions → seamless migration to new schema & API version
- Decoupling of deployment and enablement of SW functionality
- Allows early (beta) release of new features
- Fast disabling of features in case of problems

Feature Flags / Toggles - Implementation

Simple Implementation using a global generic Config Class.

```
public Spline[] reticulateSplines() {
  if( toggleConfig.featureIsEnabled("use-new-SR-algorithm") ){
    return enhancedSplineReticulation();
  } else {
    return oldFashionedSplineReticulation();
  }
}
```

```
public class FeatureToggleConfig {
  Map<String, Boolean> featureConfig = new HashMap<>();

  public void setFeature(String featureName, boolean enabled) {
      featureConfig.set(featureName, enabled);
   }

   public boolean featureIsEnabled(featureName){
      return featureConfig.get(featureName);
   }
}
```

Or use a framework like Togglz (https://www.togglz.org)

Feature Flags / Toggles - Categories

Release Toggles

- Deploy application with new features and enable them eventually (maybe never)
- Decouples deployment from releasing
- Transitionary (should be removed after some time in production)

Experiment Toggles

- Enable different implementations for different groups of people
- Often used in A/B testing scenarios
- Transitionary (should be removed after tests are completed)

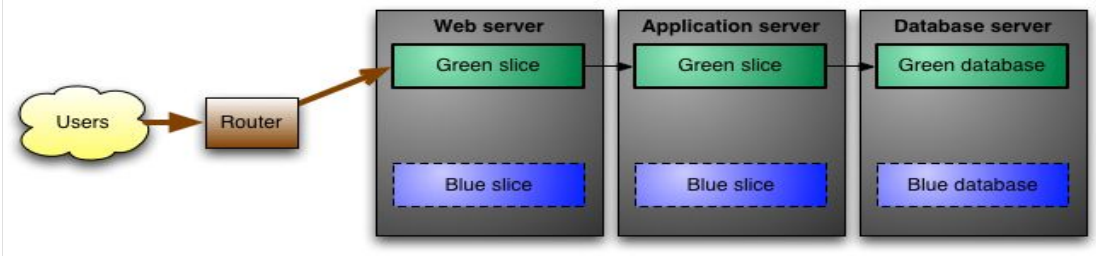
Ops Toggles

- Used to control operational aspects (e.g. reduce functionality during a DB update)
- Usually transitionary, some long living

Permission Toggles

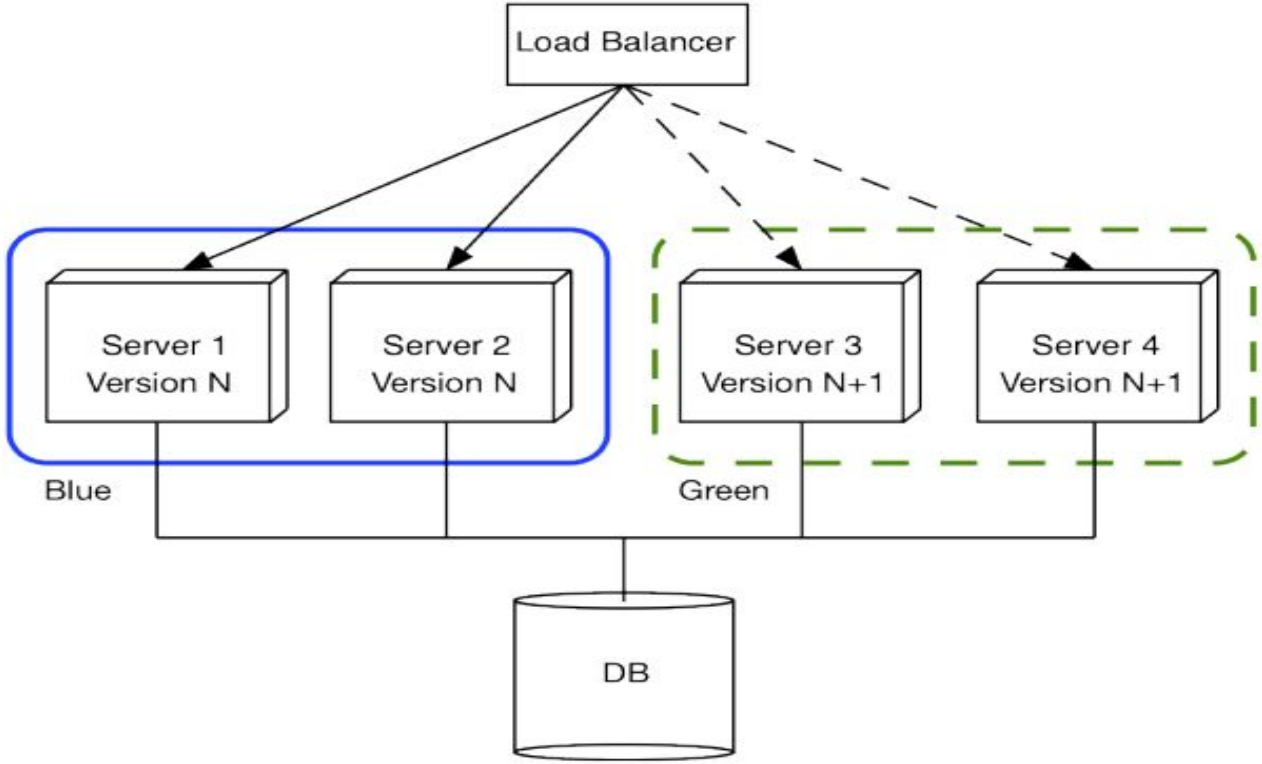
- Enabling some features for a certain group of (internal) people (e.g. alpha / beta testers)
- Very often long living but also dynamic (continuous new features)

Blue-Green Deployment



- Two identical environments (blue & green)
- Green is running productively
- Deploy new release to blue environment, let it warm up
  → smoke tests to verify it works
  → move to the new version by changing the router configuration
- Blue becomes productive. Green is used for the next release.
- If something goes wrong, change router back to green env.

Blue-Green Deployment

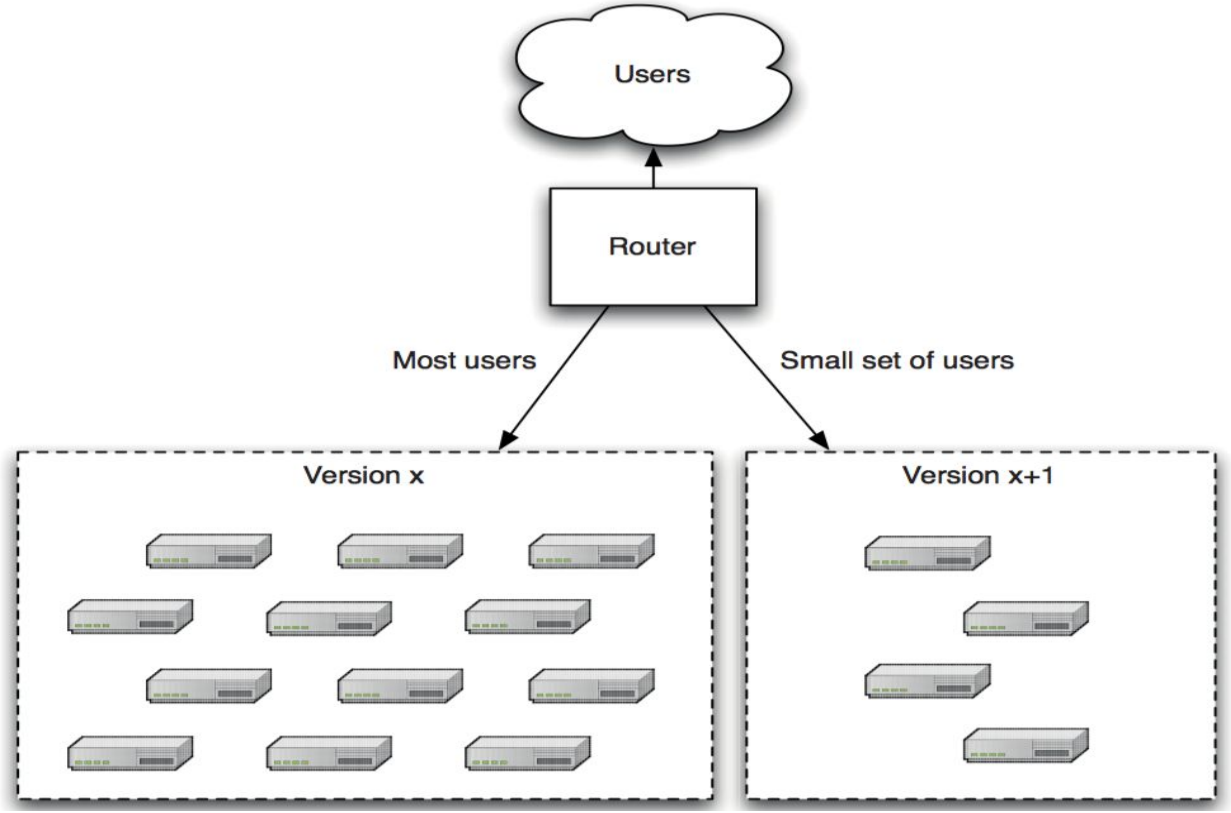


Canary Deployment (Rolling update)

Instead of instant switching all requests from version x to the x+1, users are gradually moved to the new version (e.g. number of new instances are increased and old instances decreased)

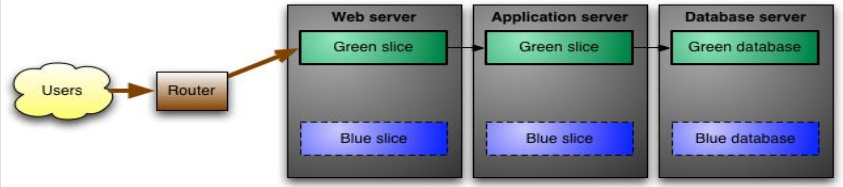
- Benefits:
  - Make rolling back easy
  - A/B testing can be done by routing some users to new version and some to the old
  - You can check if the application meets capacity requirements gradually
- Disadvantages
  - Harder to manage in smaller installations
  - Imposes further constraints on DB schema upgrades

Canary Deployment



Zero Downtime State transition

- Zero Downtime Migration of cloud-native applications is easy
  - service/code is stateless → state has to be externalized
  - use feature flag, blue-green or canary deployment



- But what if you need to migrate the schema of your database?
  - No maintenance-window to migrate the database
  - No updates may go lost
  - No performance hints

→ Zero downtime State transition is hard

State transition options?

- Stop old app, migrate data, start new app → not an option
- Reduce app functionality → influence on customer
  - Switch app to read only, create a copy of the db, migrate to new schema, switch app to new version
- Synchronization → error prone, performance?
  - Create copy of db, migrate the schema, run a process which keeps both versions in sync, switch app to new version
- Design the application to allow migration → needs planning
  - plan migration in several small steps instead of one large change
  - needs back and forward compatibility in each step
  - use well known building-blocks (patterns) for migration steps

State transition building-blocks

- Multiple (baby) steps per transition
- Each step is executed in sequence:
  - DB: → DB migration, one-time process
  - Code: → Zero-downtime (B/G, canary) deploy of all instances to new version
- Possible to go back within the block, until it is completed (otherwise start a new reverse transition)

Add a Field/Column

1. DB: Add new Column
2. DB: Preset value (NULL, default, computed)
3. Code: read from and write to the Column

Change a Field/Column (name, type, format)

1. DB: add new column (no constraints e.g. NotNull)
2. Code: read from old column, write to both
3. DB: copy data from old to new column
   (for large datasets do it in multiple shards)
   Add required constraints (eg. NotNull) to new column
4. Code: read from new column, write to both
5. DB: delete constraints from old column
6. Code: read from, write to new column only
7. DB: delete the old column

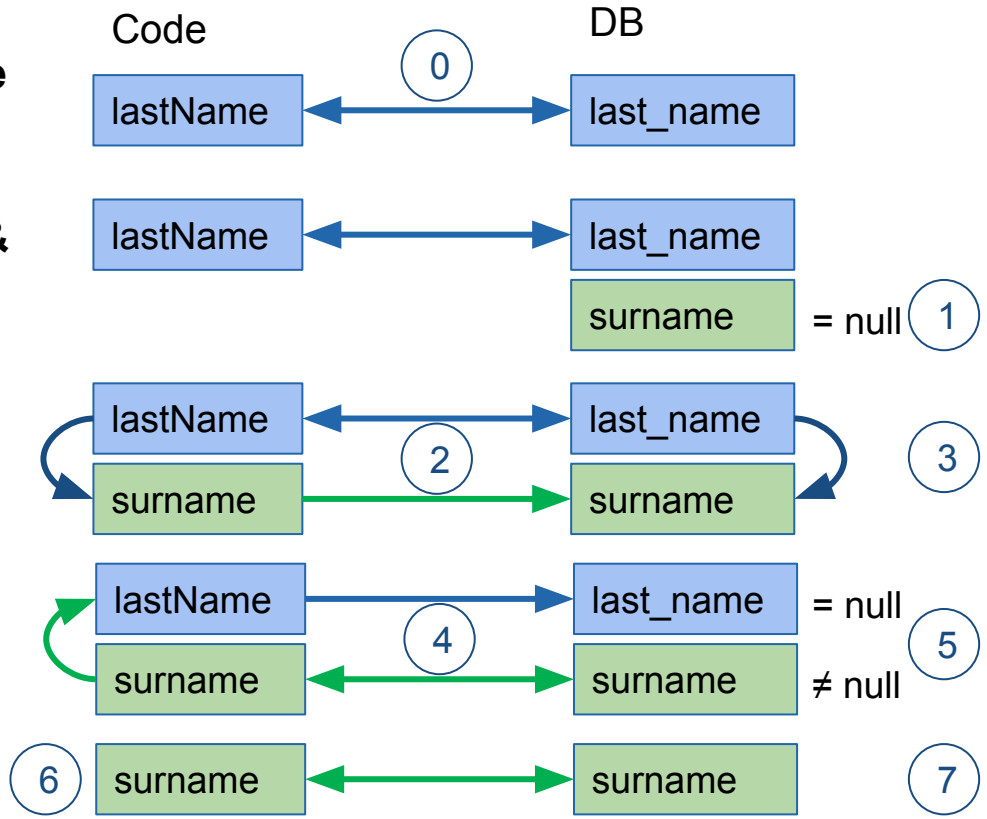
Delete a Field/Column

1. DON'T! It is a destructive operation
   → Keep the Column for a retention period
2. Code: stop reading, but keep writing the column
3. (in consolidation phase after retention period)
   Code: stop, writing the column
   DB: Delete the Column

Example - Rename field/column: last_name → surname

Zurich University of Applied Sciences
School of Engineering
InIT Institute of Applied Information Technology

0. DB field is **last_name** / Code **lastName**
1. DB: add field **surname**
2. Code: add **surname** field, still **read from last_name**, **write to last_name & surname**
3. DB: copy values from **last_name to surname**
4. Code: **read from surname**, **write to last_name & surname**
5. DB: remove **NotNull** constraint from last_name, add to surname
6. Code: remove lastName, **read & write to surname only**
7. DB: **delete last_name** column



More complex operation

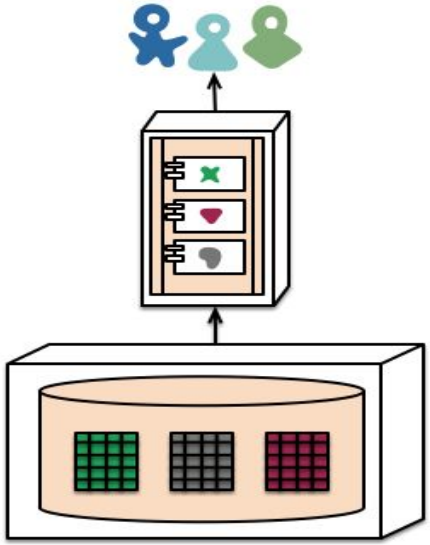
- Add/Change/Delete a Relation → Add/Change/Delete a Column
- Add/Split/Merge/Delete Tables → Series of Column operations
- What about Referential Integrity constraints (NotNull, foreign keys,...)
  - Contain no business value
  - (only) a safety-net to avoid corruption

  → remove before and recreate after transition

Best practice

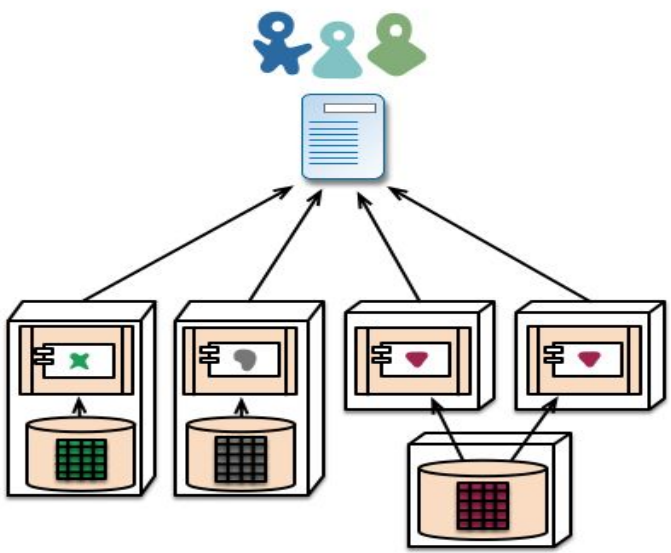
- Use migration tool like flyway or liquibase
  - allows back and forward transition for each step
  - changes in version control
- Decouple the database
  - use simpler database model per microservice
  - less dependencies and side-effects
  - easier to change
- Use migration friendly Architecture
  - Use Event-Sourcing and CQRS
  - Allows to recreate the view model

Decoupling the data model



monolith - single database

Single database and common data model
→ lots of dependencies and side-effects
→ difficult to change



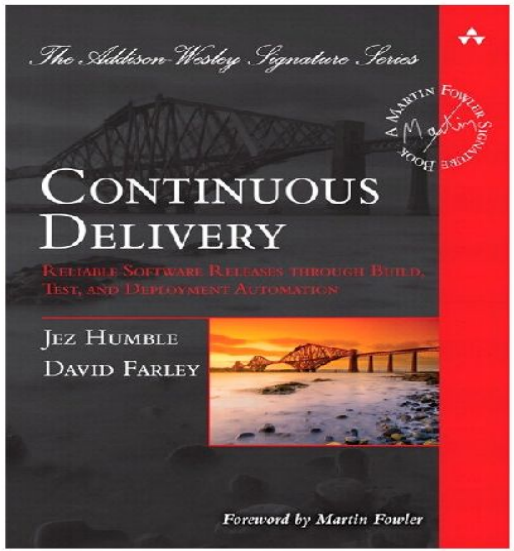
microservices - application databases

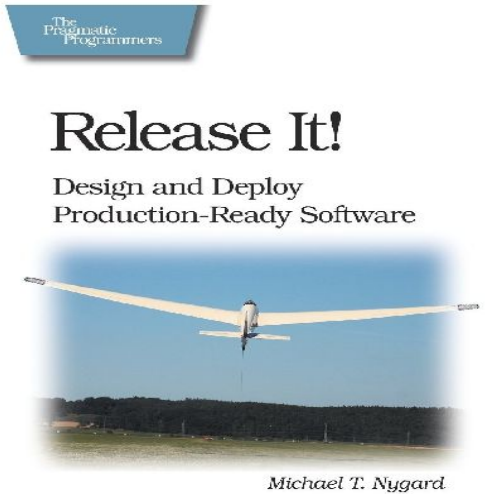
Individual database and data model per service
→ less dependencies and side-effects on other services
→ flexible to migrate and adopt

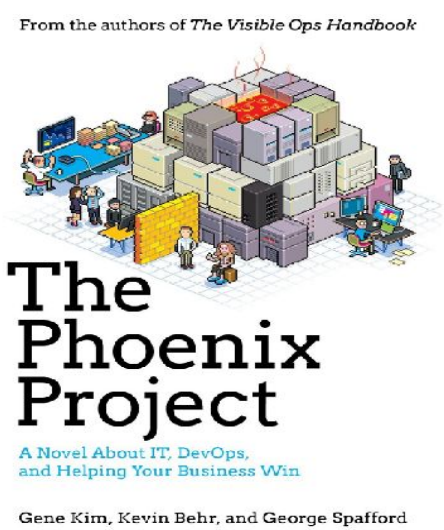
Source: http://martinfowler.com/articles/microservices.html

Continuous Delivery – How?

- A lot of good literature is available





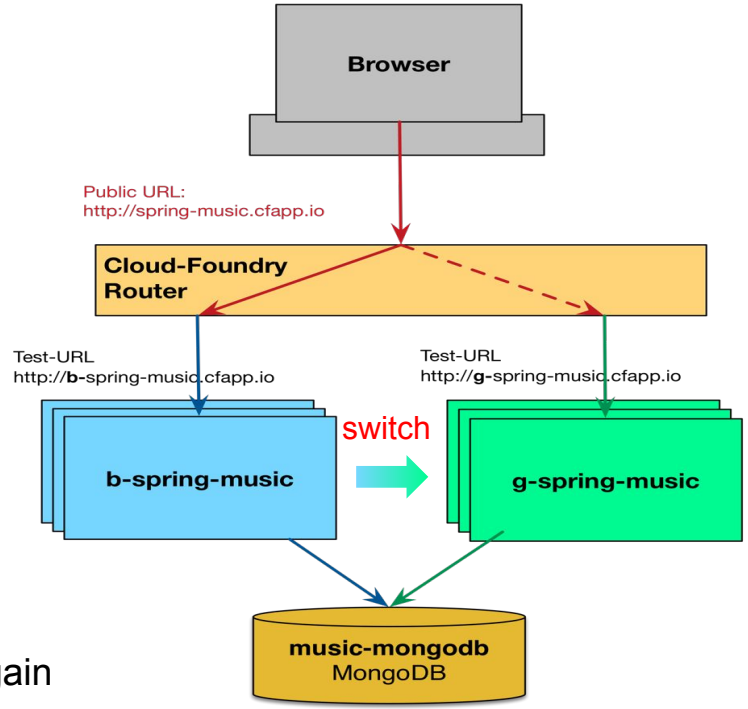


Appendix

Blue-Green Deployment

There are several options to do blue green deployment in CF
Most complete version is:

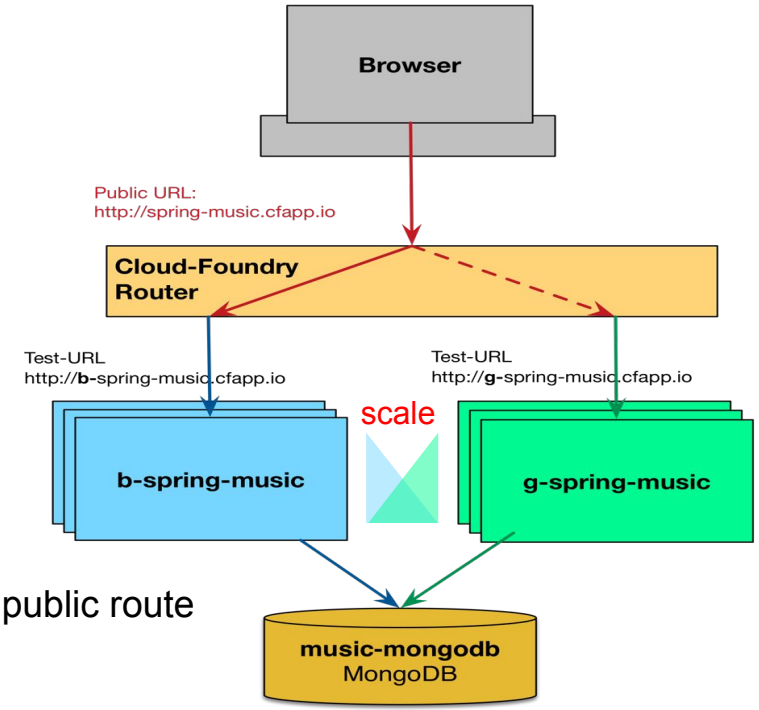
1. Current version is deployed as b-spring-music,
   Public route https://spring-music.cfapp.io points to it
2. `Push` new version with name g-spring-music,
   `scale` it up to the same number of instances
   and `bind` it to the required services (db)
3. Test the new version on the test URL
   https://g-spring-music.cfapp.io (smoke tests)
4. Add public route to the new instance
   `cf map-route g-spring-music cfapp.io -n spring-music`
   and remove it from the old instance
   `cf unmap-route b-spring-music cfapp.io -n spring-music`
   (for a very short moment the requests were routed to both version using a round-robin algorithm)
5. After some time, if everything runs correctly
   the old app can be deleted. If not, you can switch back again



Canary Deployment with CF

Basically an extension to Blue-Green deployment:

1. Current version is deployed as b-spring-music,
   Public route https://spring-music.cfapp.io points to it
2. `Push` new version with name g-spring-music,
   `scale` it ~~up~~ to the ~~same~~ minimal number of instances (1)
   and `bind` it to the required services (db)
3. Test the new version on the test URL
   https://g-spring-music.cfapp.io (smoke tests)
4. Add public route to the new instance
   `cf map-route g-spring-music cfapp.io -n spring-music`
5. Gradually scale up instances of new g-spring-music
   and scale down instances of old b-spring-music
   (distribution of requests is proportional to number of instances of old and new version)
6. If b-spring music is down to 1 instance remove it from the public route
   `cf unmap-route b-spring-music cfapp.io -n spring-music`
   After some time, if everything runs correctly
   the old app can be deleted. If not, you can switch back again



DevOps Practices

- Version Control For All
- Automated Testing
- Proactive Monitoring and Metrics
- Kanban/Scrum
- Visible Ops/Change Management
- Configuration Management
- Incident Command System
- **Continuous Integration / Deployment / Delivery**
- “Put Developers On Call”
- Virtualization/Cloud/Containers
- Toolchain Approach
- Transparent Uptime/Incident Retrospectives

Things Not To Do

- Only Token Gestures
  - “Ops team, change your name to DevOps team!”
  - “Put DevOps in those job titles!”
- Only Implement Tools
  - Changing tools without changing tactics leaves the battlefield strewn with bodies
- Create more silos (Dev, DevOps, Ops)
- Devalue operations Or development knowledge
- Anything you’re not measuring the impact of