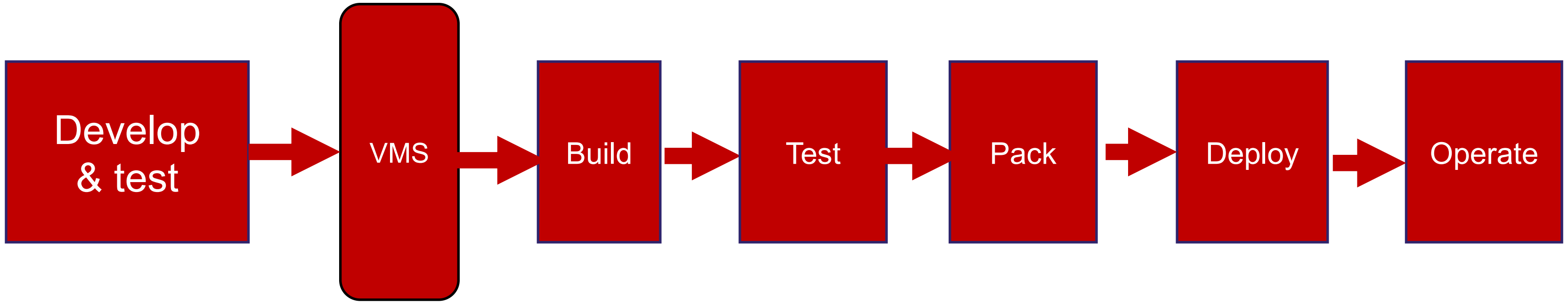
Lecture 10

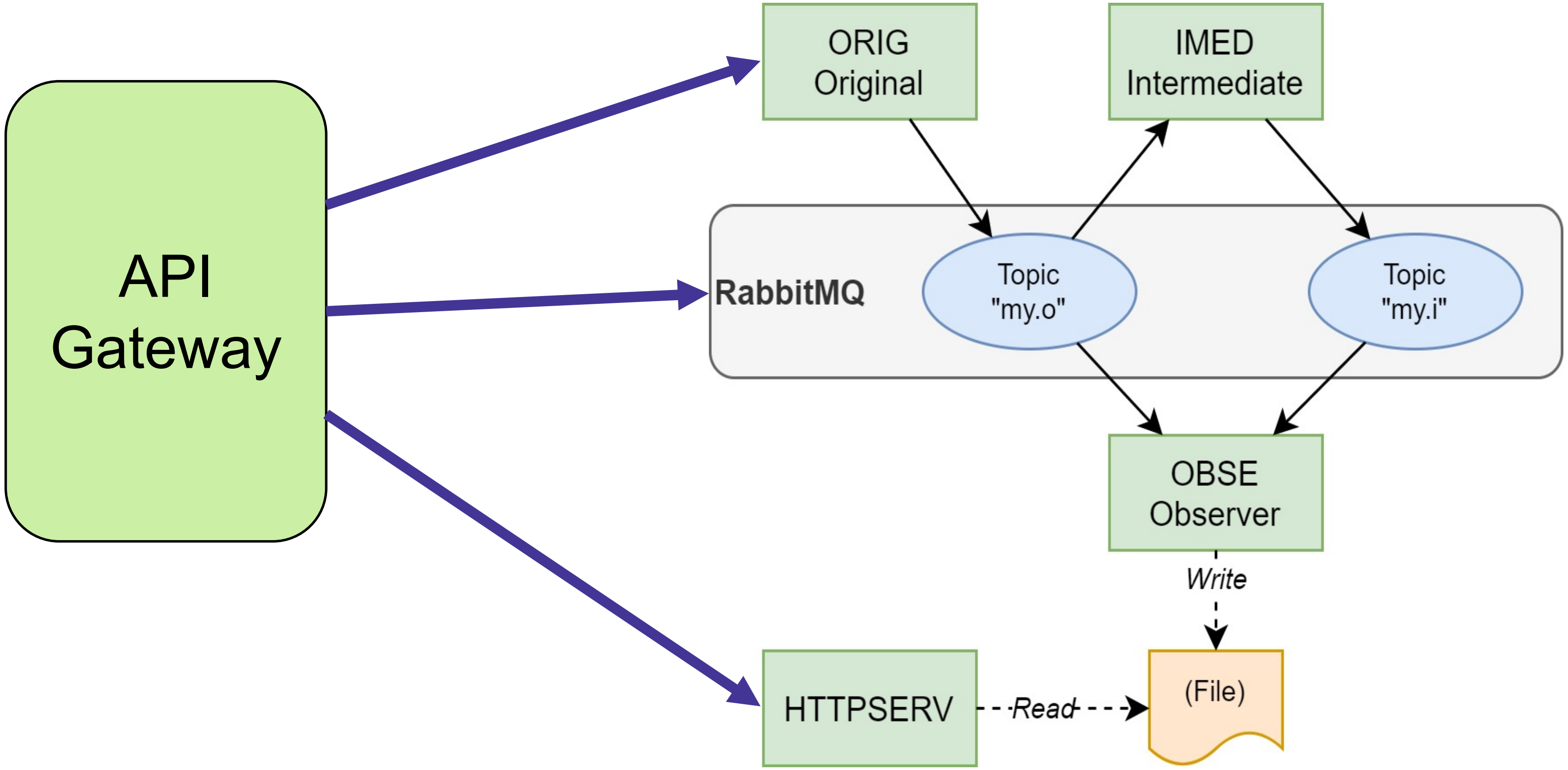
Automation

Kari Systä
09.11.2021

Tampereen yliopisto
Tampere University

Project





Kari Systä

Schedule

- The instructions disclosed: 08.11.2021
  - Students can start by installing the gitlab-ci
  - New versions to resolve ambiguous parts may be published later.
- Discussions in the lecture: 09.11.2021
  - Students are asked to give clarification questions
- Latest submission if you want course to graded in 2021: 06.12.2021
- Latest submission to pass the course: 31.12.2021

Project includes

1. Install the pipeline infrastructure using gitlab-ci. This means that you should:
   - install gitlab and runners on their own machine. A fresh virtual machine is recommended. Instructions to help in this process are below in section gitlab-ci.
   - Define the pipeline using `.gitlab-ci.yml` for the application you implemented for the message-queue exercise. The result of the pipeline should be a running system, so the containers should be started automatically. (In other words: “git push => the system is up and running)
   - Test the pipeline with the current version of the application.
2. Create, setup and test an automatic testing framework
   - First, you need to select the testing tools. We do not require any specific tool, even your own test scripts can be used.
   - Create test to the existing functionality of the application (see “Application and its new features” below)
3. Implement the changes and additional functionalities to the RabbitMQ exercise

API gateway

GET /messages

Returns all message registered with OBSE-service

PUT /state (payload “INIT”, “PAUSED”, “RUNNING”, “SHUTDOWN”)

PAUSED = ORIG service is not sending messages

RUNNING = ORIG service sends messages

If the new state is equal to previous nothing happens.

There are two special cases:

INIT = everything is in the initial state and ORIG starts sending again, state is set to RUNNING
SHUTDOWN = all containers are stopped

GET /state

get the value of state

GET /run-log

Get information about state changes

Example output:

2020-11-01T06:35:01.373Z: INIT
2020-11-01T06:40:01.373Z: PAUSED
2020-11-01T06:40:01.373Z: RUNNING

GET /message-log

Forward the request to HTTPSERV and return the result

GET /node-statistic (optional)

Return core statistics (the five (5) most important in your mind) of the RabbitMQ. (For getting the information see https://www.rabbitmq.com/monitoring.html)
Output should syntactically correct and intuitive JSON. E.g:
{ “fd_used”: 5, …}

GET /queue-statistic (optional)

Return a JSON array per your queue. For each queue return “message delivery rate”, “messages publishing rate”, “messages delivered recently”, “message published lately”. (For getting the information see https://www.rabbitmq.com/monitoring.html)

End report

1. **Instructions for the teaching assistant**

   Implemented optional features

   List of optional features implemented.

   Instructions for examiner to test the system.

   Pay attention to optional features.

2. **Description of the CI/CD pipeline**

   Briefly document all steps:

   Version management; use of branches etc

   Building tools

   Testing; tools and test cases

   Packing

   Deployment

   Operating; monitoring

3. **Example runs of the pipeline**

   Include some kind of log of both failing test and passing.

4. **Reflections**

   Main learnings and worst difficulties

   Especially, if you think that something should have been done differently, describe it here.

   Amount effort (hours) used

   Give your estimate

Grading

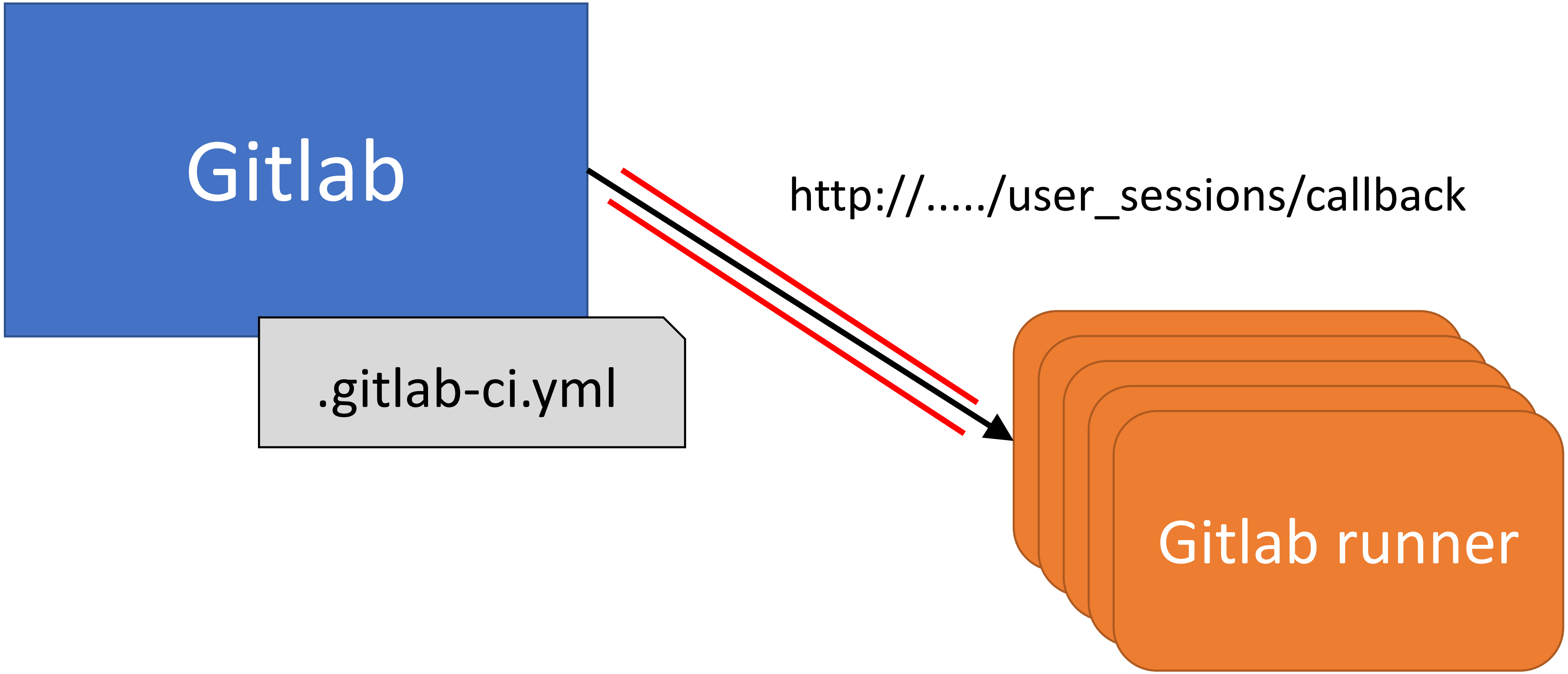
As already been communicated this project affects 40% of in the evaluation of the overall course. For that 40% we use the following table

Compulsory parts work according to requirements	0..20 %
Implementation of optional features (each optional feature is worth of 5%)	0..30 %
Overall quality (clean code, good comments, ….)	0..5%
Quality of the end report	0..5% (+ up to 5% compensation of a good analysis of your solution and description of a better way to implement.)

Note: optional points can compensate problems elsewhere, but the total sum is capped at 50%. That means that max 10% can be used to compensate lost points in exercises and exam.

Gitlab CI

https://docs.gitlab.com/ee/ci/



Types of runners

Shared Runners

- These runners are useful for jobs multiple projects which have similar requirements. Instead of using multiple runners for many projects, you can use a single or a small number of Runners to handle multiple projects which will be easy to maintain and update.

Specific Runners

- These runners are useful to deploy a certain project, if jobs have certain requirements or specific demand for the projects. Specific runners use *FIFO* (First In First Out) process for organizing the data with first-come first-served basis.

Example from:
https://docs.gitlab.com/ee/user/project/pages/getting_started/pages_from_scratch.html

```
image: ruby:2.7

workflow:
  rules:
    - if: '$CI_COMMIT_BRANCH'

before_script:
  - gem install bundler
  - bundle install

pages:
  stage: deploy
  script:
    - bundle exec jekyll build -d public
  artifacts:
    paths:
      - public
  rules:
    - if: '$CI_COMMIT_BRANCH == "master"'

test:
  stage: test
  script:
    - bundle exec jekyll build -d test
  artifacts:
    paths:
      - test
  rules:
    - if: '$CI_COMMIT_BRANCH != "master"'
```

```
image: ruby:2.7

workflow:
  rules:
    - if: '$CI_COMMIT_BRANCH'

before_script:
  - gem install bundler
  - bundle install

pages:
  stage: deploy
  script:
    - bundle exec jekyll build -d public
  artifacts:
    paths:
      - public
  rules:
    - if: '$CI_COMMIT_BRANCH == "master"'

test:
  stage: test
  script:
    - bundle exec jekyll build -d test
  artifacts:
    paths:
      - test
  rules:
    - if: '$CI_COMMIT_BRANCH != "master"'
```

Base Image

```
image: ruby:2.7

workflow:
  rules:
    - if: '$CI_COMMIT_BRANCH'

before_script:
  - gem install bundler
  - bundle install

pages:
  stage: deploy
  script:
    - bundle exec jekyll build -d public
  artifacts:
    paths:
      - public
  rules:
    - if: '$CI_COMMIT_BRANCH == "master"'

test:
  stage: test
  script:
    - bundle exec jekyll build -d test
  artifacts:
    paths:
      - test
  rules:
    - if: '$CI_COMMIT_BRANCH != "master"'
```

This is run before every script

```
image: ruby:2.7

workflow:
  rules:
    - if: '$CI_COMMIT_BRANCH'

before_script:
  - gem install bundler
  - bundle install

pages:
  stage: deploy
  script:
    - bundle exec jekyll build -d public
  artifacts:
    paths:
      - public
  rules:
    - if: '$CI_COMMIT_BRANCH == "master"'

test:
  stage: test
  script:
    - bundle exec jekyll build -d test
  artifacts:
    paths:
      - test
  rules:
    - if: '$CI_COMMIT_BRANCH != "master"'
```

Used rules

Many variables available: https://docs.gitlab.com/ee/ci/variables/predefined_variables.html

Use of rule, executed if rule is “master”

```
image: ruby:2.7

workflow:
  rules:
    - if: '$CI_COMMIT_BRANCH'

before_script:
  - gem install bundler
  - bundle install

pages:
  stage: deploy
  script:
    - bundle exec jekyll build -d public
  artifacts:
    paths:
      - public
  rules:
    - if: '$CI_COMMIT_BRANCH == "master"'

test:
  stage: test
  script:
    - bundle exec jekyll build -d test
  artifacts:
    paths:
      - test
  rules:
    - if: '$CI_COMMIT_BRANCH != "master"'
```

This is for state ”deploy”.

Default states are build, test, deploy

This is for state ”test”.

```
image: ruby:2.7

workflow:
  rules:
    - if: '$CI_COMMIT_BRANCH'

before_script:
  - gem install bundler
  - bundle install

pages:
  stage: deploy
  script:
    - bundle exec jekyll build -d public
  artifacts:
    paths:
      - public
  rules:
    - if: '$CI_COMMIT_BRANCH == "master"'

test:
  stage: test
  script:
    - bundle exec jekyll build -d test
  artifacts:
    paths:
      - test
  rules:
    - if: '$CI_COMMIT_BRANCH != "master"'
```

Never mind ☺

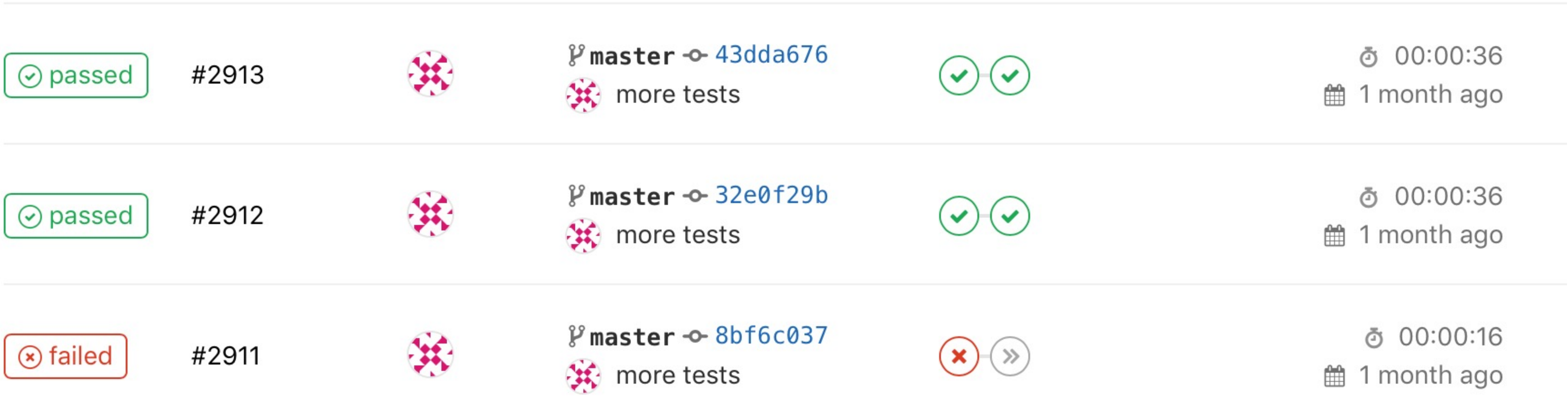
Script to run

```
image: ruby:2.7

workflow:
  rules:
    - if: '$CI_COMMIT_BRANCH'

before_script:
  - gem install bundler
  - bundle install

pages:
  stage: deploy
  script:
    - bundle exec jekyll build -d public
  artifacts:
    paths:
      - public
  rules:
    - if: '$CI_COMMIT_BRANCH == "master"'

test:
  stage: test
  script:
    - bundle exec jekyll build -d test
  artifacts:
    paths:
      - test
  rules:
    - if: '$CI_COMMIT_BRANCH != "master"'
```

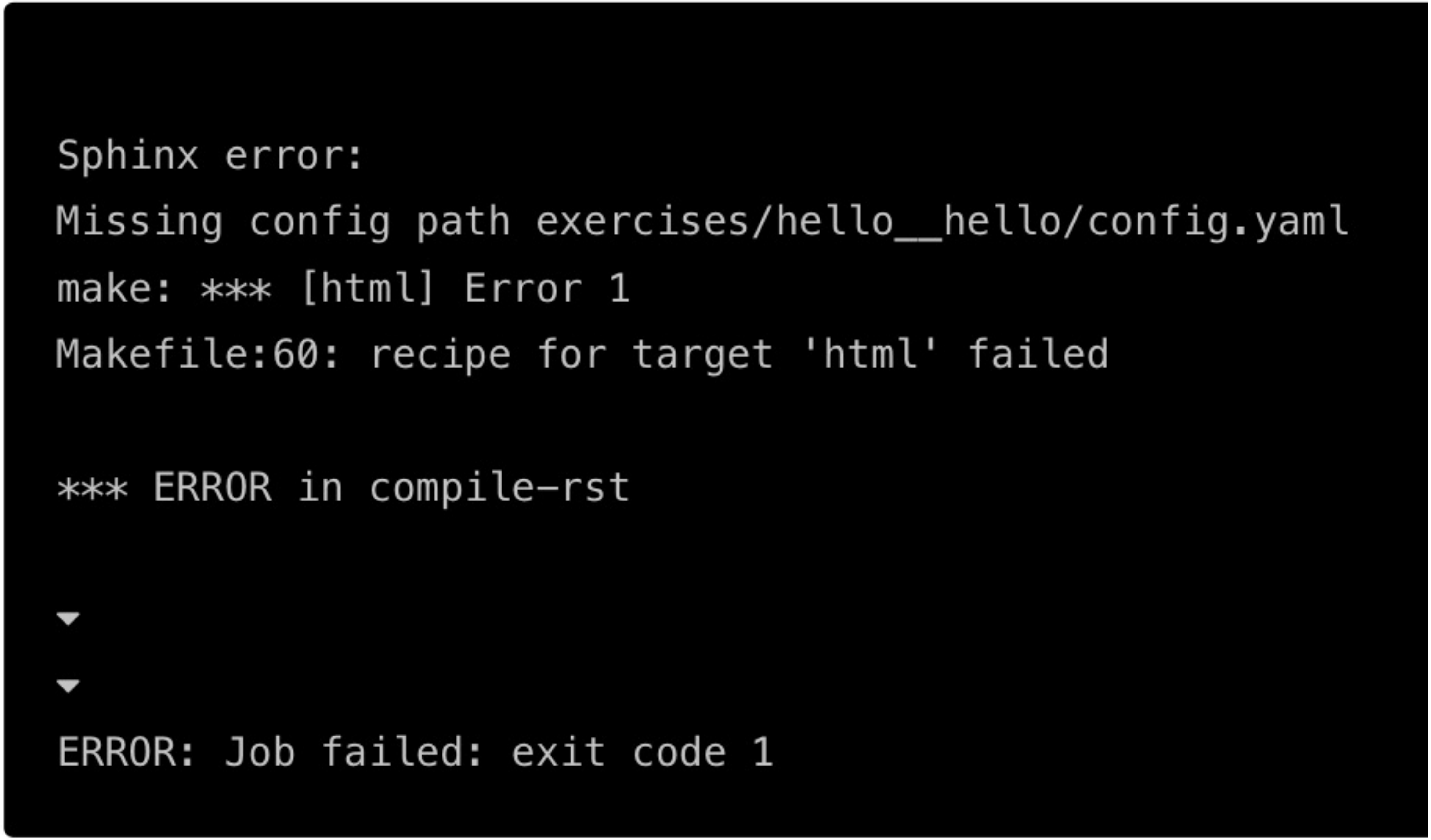
File location

How to install .gitlab-ci.yml?

```
git add .gitlab-ci.yml
git commit -m "Add .gitlab-ci.yml”
git push origin master
```



✓ passed	#2913	master -o- 43dda676 more tests	✓ ✓	00:00:36 1 month ago
✓ passed	#2912	master -o- 32e0f29b more tests	✓ ✓	00:00:36 1 month ago
✗ failed	#2911	master -o- 8bf6c037 more tests	✗ »	00:00:16 1 month ago



```
Sphinx error:
Missing config path exercises/hello__hello/config.yaml
make: *** [html] Error 1
Makefile:60: recipe for target 'html' failed

*** ERROR in compile-rst


▾
▾
ERROR: Job failed: exit code 1
```

```
variables:
    TUNIPLUSSA_ID: 'TIE23536-syksy2019'
    GIT_STRATEGY: none

stages:
    - build
    - test
    - deploy

builder:
    stage: build
    only:
    - master
    - release
    tags:
    - plussa
    artifacts:
      paths:
      - FULLLOG.txt
      expire_in: 2 week
    script:
    - tuni-rst-build
```

```
tester:
    stage: test
    only:
    - master
    tags:
    - plussa
    script:
    - tuni-publish-to-testing


publisher:
    stage: deploy
    only:
    - release
    tags:
    - plussa
    script:
    - tuni-publish-to-production
```

```
variables:
    TUNIPLUSSA_ID: 'TIE23536-
syksy2019'
    GIT_STRATEGY: none

stages:
    - build
    - test
    - deploy

builder:
    stage: build
    only:
    - master
    - release
    tags:
    - plussa
    artifacts:
      paths:
      - FULLLOG.txt
      expire_in: 2 week
    script:
    - tuni-rst-build
```

```
tester:
    stage: test
    only:
    - master
    tags:
    - plussa
    script:
           ish-to-testing



               oy

    - release
    tags:
    - plussa
    script:
    - tuni-publish-to-production
```

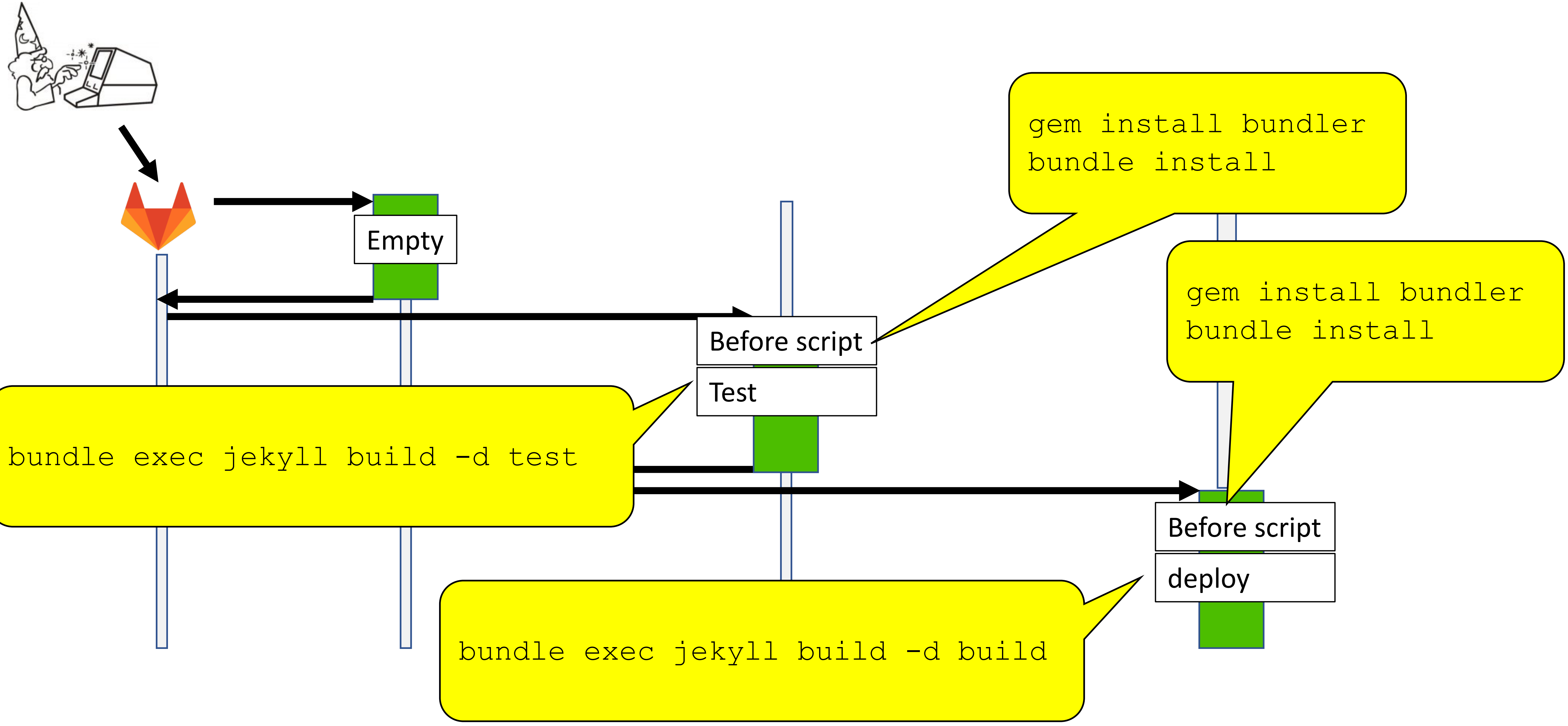
Note: The rules syntax is an improved, more powerful solution for defining when jobs should run or not. Consider using rules instead of only/except to get the most out of your pipelines.

```
image: ruby:2.7

workflow:
  rules:
    - if: '$CI_COMMIT_BRANCH'

before_script:
  - gem install bundler
  - bundle install

pages:
  stage: deploy
  script:
    - bundle exec jekyll build -d public
  artifacts:
    paths:
      - public
  rules:
    - if: '$CI_COMMIT_BRANCH == "master"'

test:
  stage: test
  script:
    - bundle exec jekyll build -d test
  artifacts:
    paths:
      - test
  rules:
    - if: '$CI_COMMIT_BRANCH != "master"'
```

Example from:
https://docs.gitlab.com/ee/user/project/pages/getting_started/pages_from_scratch.html

Is this correct?

Why not?



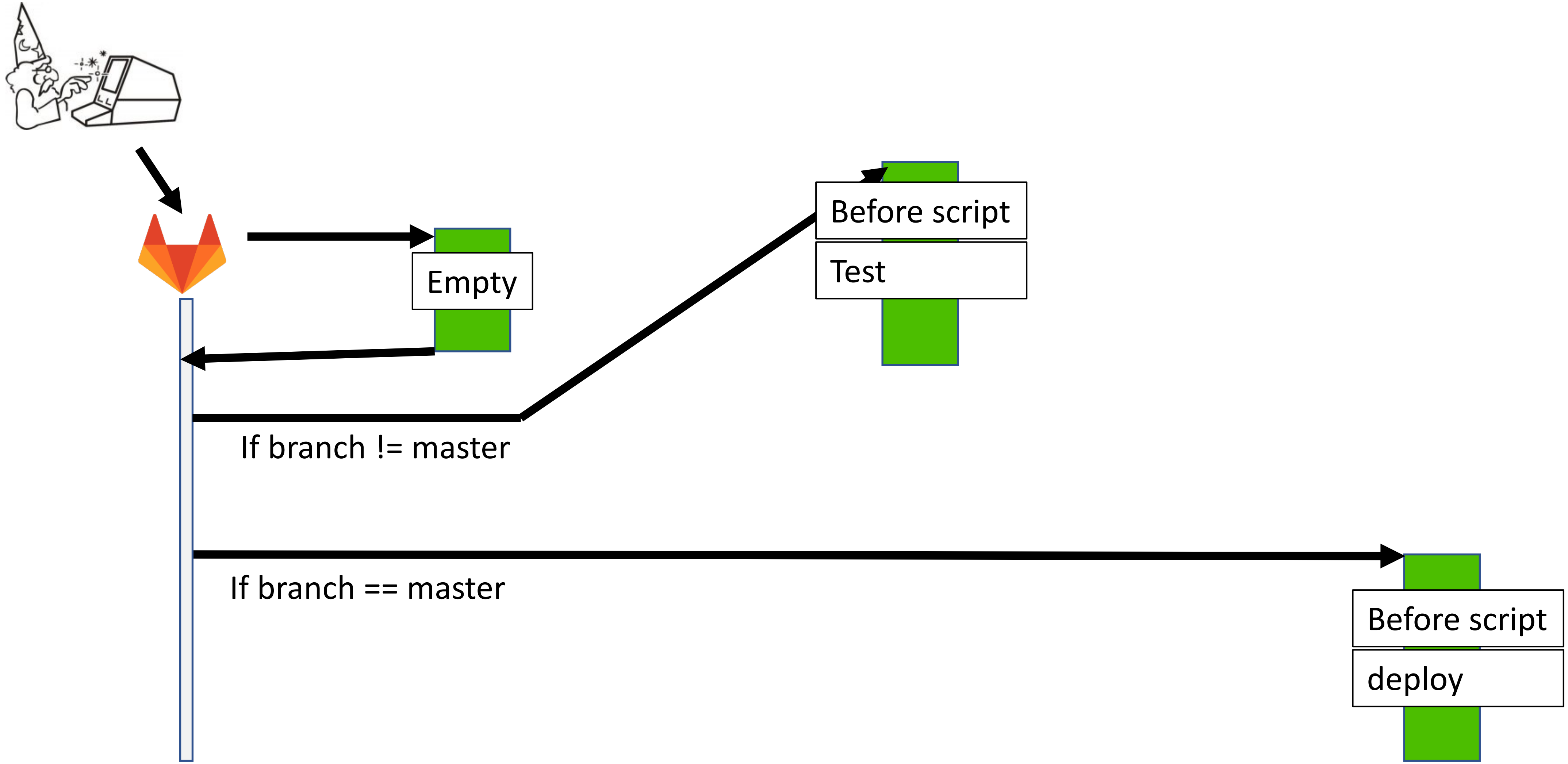
```
gem install bundler
bundle install
```

```
bundle exec jekyll build -d test
```

```
gem install bundler
bundle install
```

```
bundle exec jekyll build -d build
```

This is correct visualization!

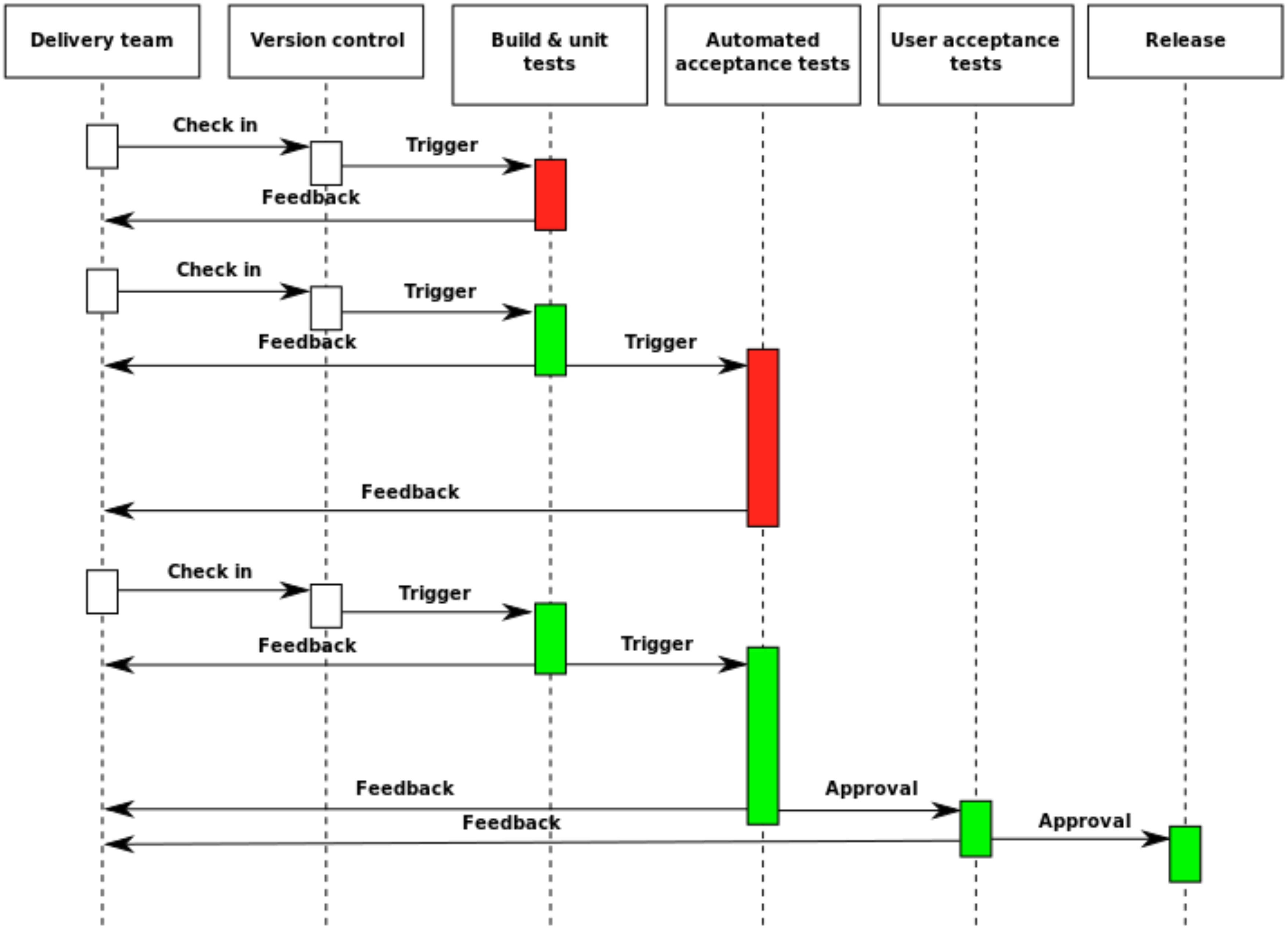


DevOps practices

- Organizational
  - increased scope of responsibilities for developers;
  - intensified cooperation between development and operations.

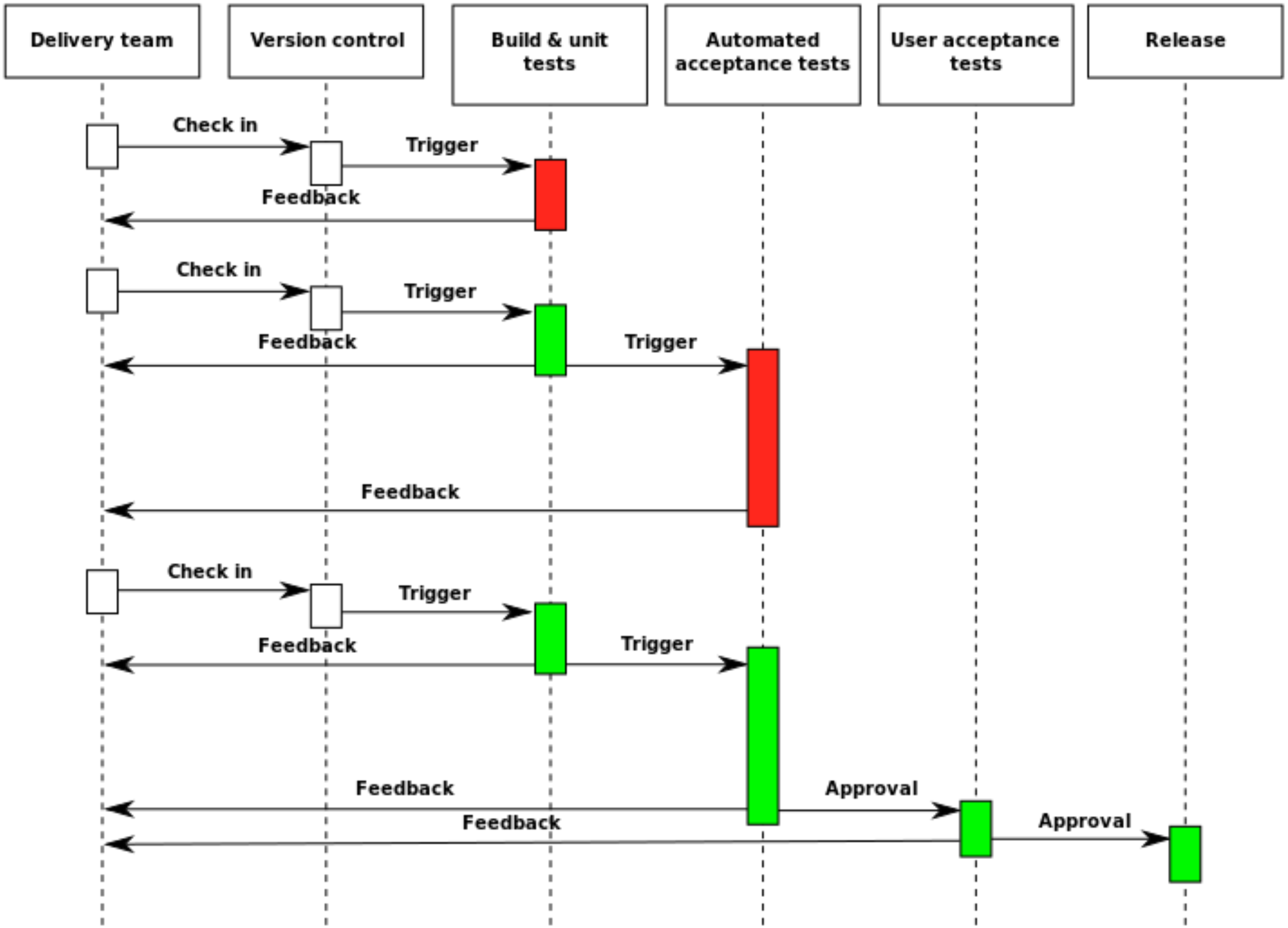
- Technical
  - **automation,**
  - monitoring
  - measurement

Deployment pipeline (a possible example)



About automation

Deployment pipeline (a possible example)



Infrastructure as code

From: https://docs.microsoft.com/en-us/azure/devops/learn/what-is-infrastructure-as-code

Infrastructure as Code (IaC) is

- the management of infrastructure (networks, virtual machines, load balancers, and connection topology) in a descriptive model,
- using the same versioning as DevOps team uses for source code.
- Like the principle that the same source code generates the same binary, an IaC model generates the same environment every time it is applied.
- IaC is a key DevOps practice and is used in conjunction with continuous delivery.

Benefits of automation

- Prevent errors
- Is repeatable
- No need to write documentation
- Enables collaboration because everything is explicit in scripts
- Expertise encapsulated in scripts
- Manual work is boring
- Fast and relentless feedback
- Risk management: Automated checking and auditing

Automation includes

- Building
  -> no command-line tools needed
- Testing
  -> run frequently
- Other quality analysis
  -> less manual inspection needed;
- Deployment
  -> VMs and containers created automatically
  -> configuration management
- Database tools
  -> initialization
  -> management
- Scaling

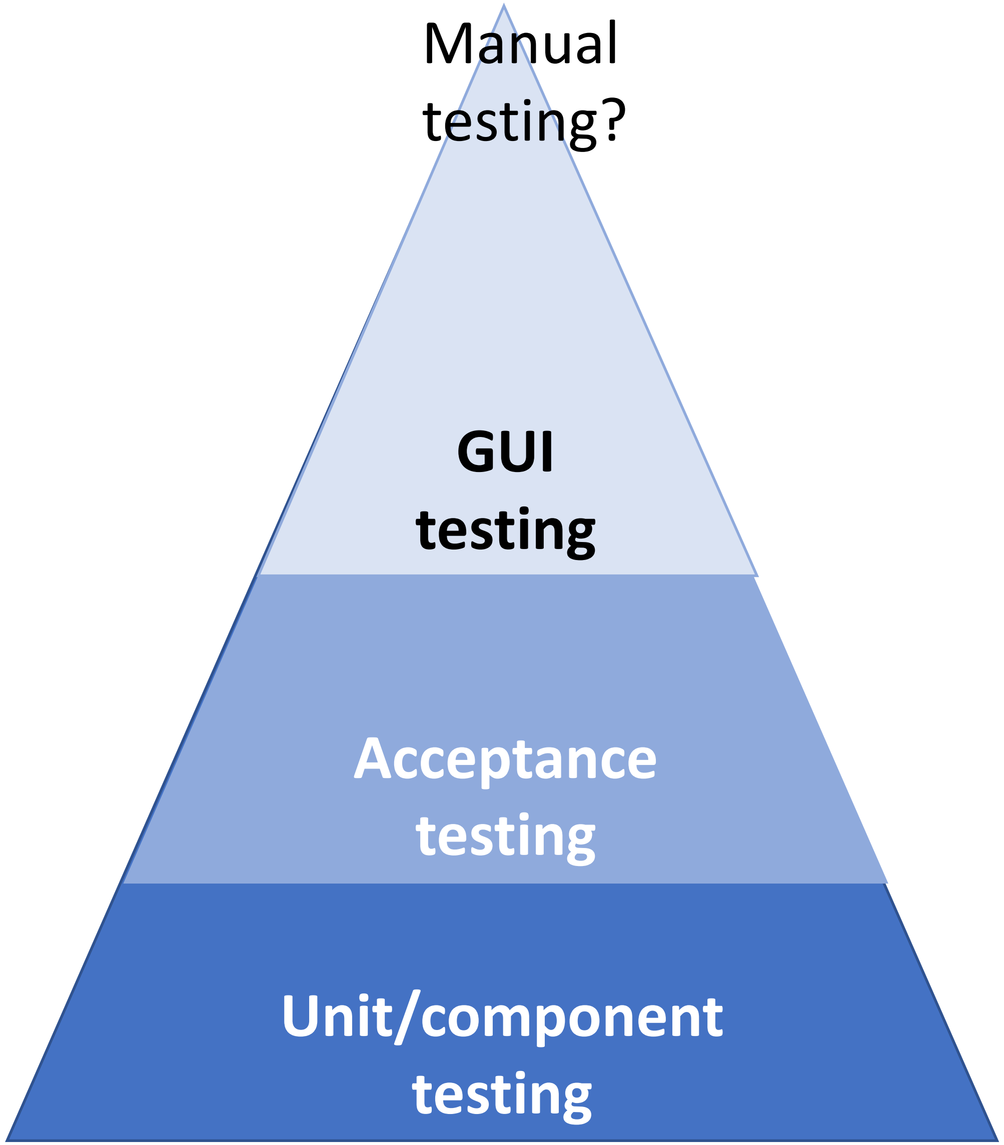
Automated tests

- A common practice in CI and CD
- Does not invent the test (usually);
  - test are designed and implemented manually but
  - executed automatically
- Tests need to maintained
- Software needs to be testable
- Not a silver bullet for testing, but necessary helper in CI/CD

Testability

- Testbed can command the software
- Tests can investigate state and results
- Proper architecture and coding style helps
  e.g. Standard getters and setters
- Well-defined APIs

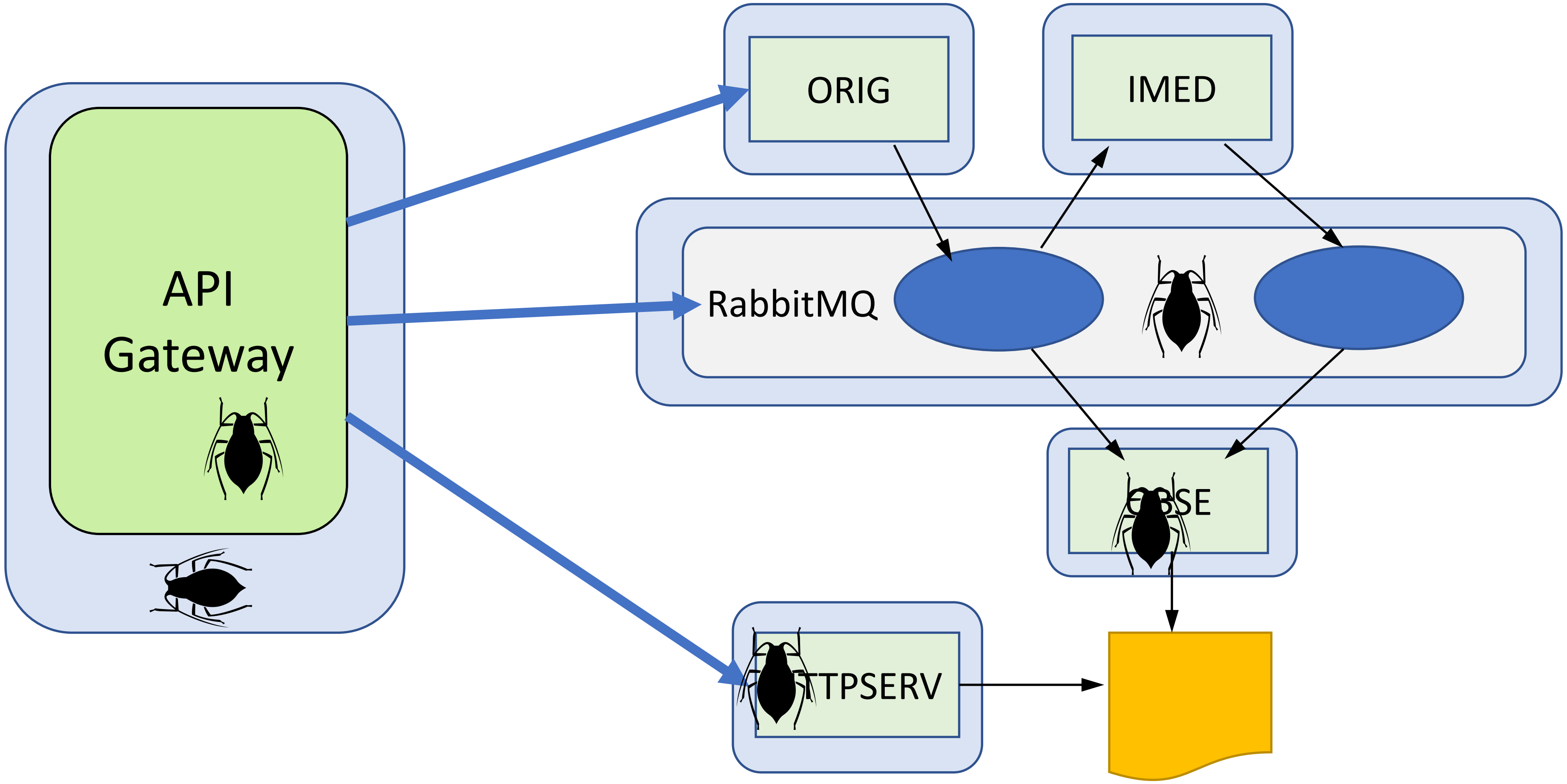
Tampereen yliopisto
Tampere University



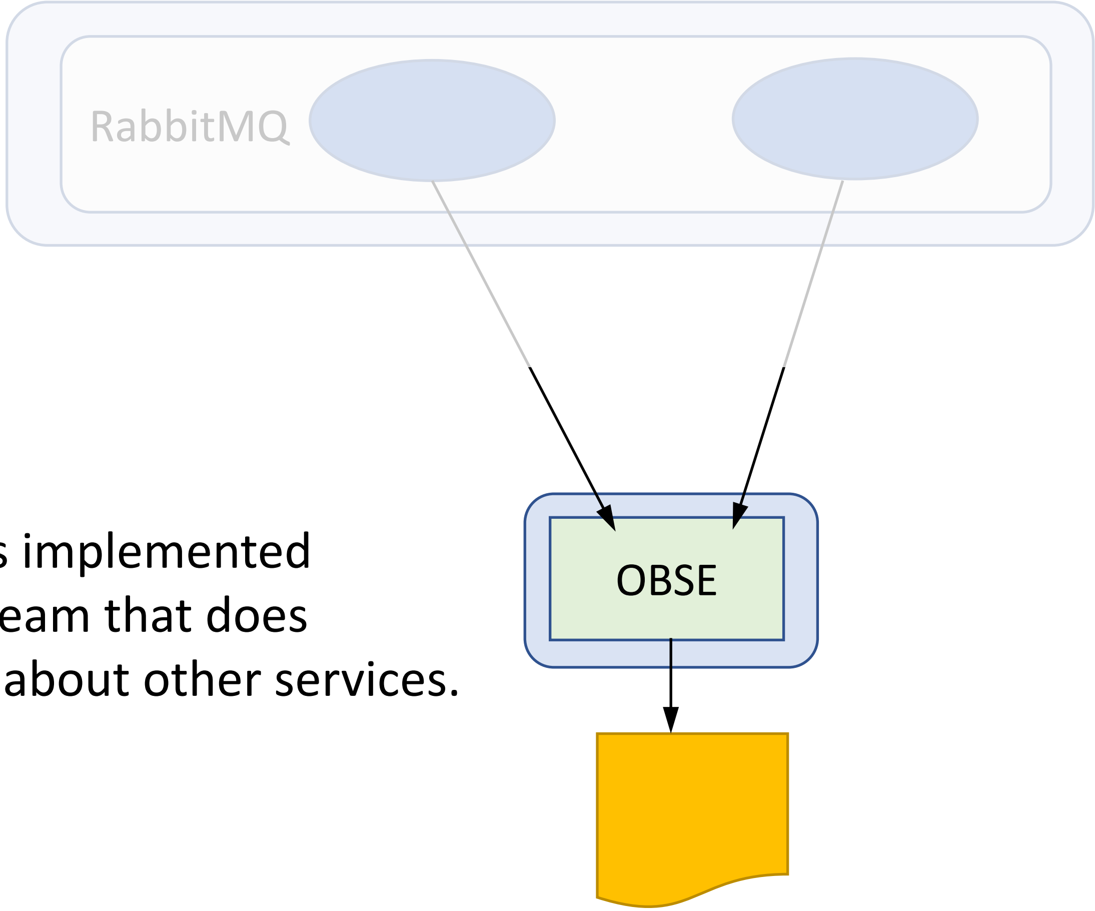
Automated acceptance tests

- Acceptance tests do not test everything but is an essential “gate” if deployment is automated.
- Some best practices (according to Humbley and Farley):
  - Test in realistic environment(s)
  - Acceptence tests are owned by the whole team (no separate team for it)
  - Developers should be able to run the tests in their own dev environment)
  - Tie to business value – not to technical solution of the system
- Nonfunctional testing
  - Capacity, scalability
  - Code quality analysis

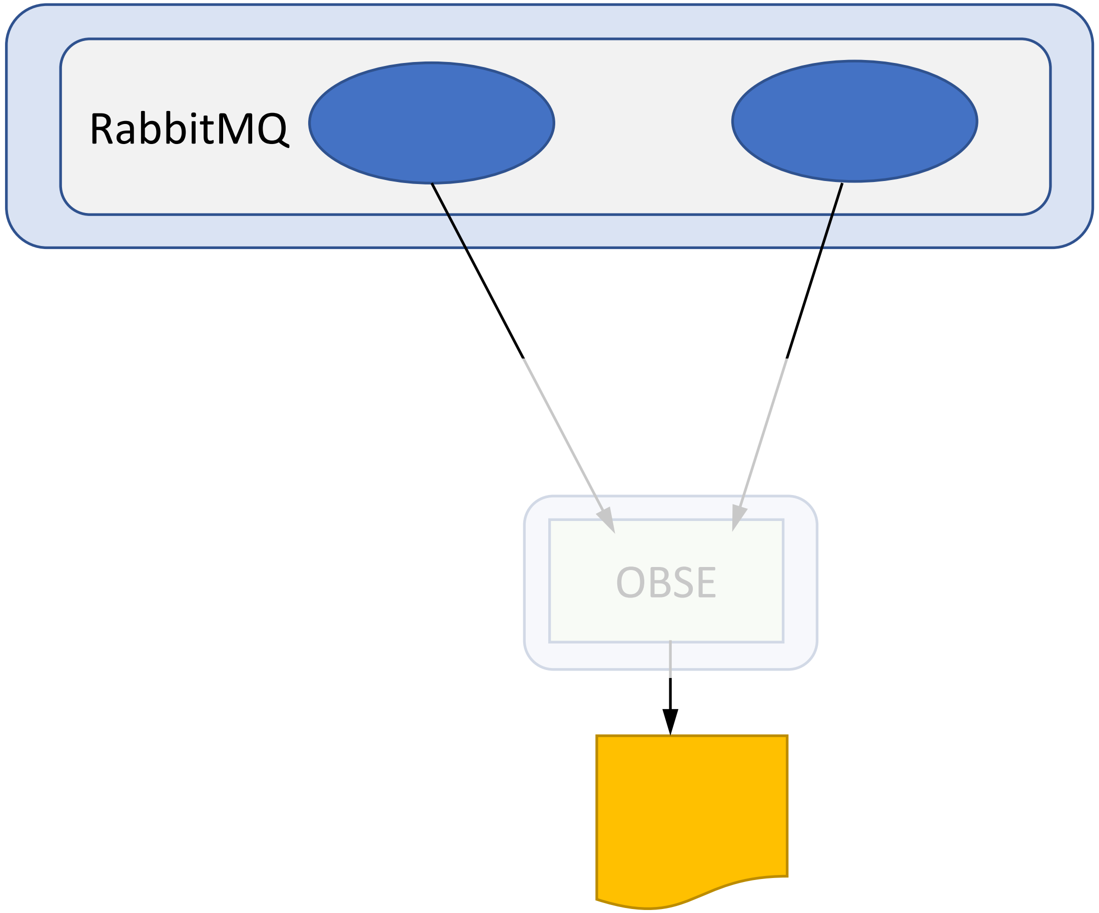
Testing cloud-native is difficult
And debugging even more difficult



Tampereen yliopisto
Tampere University



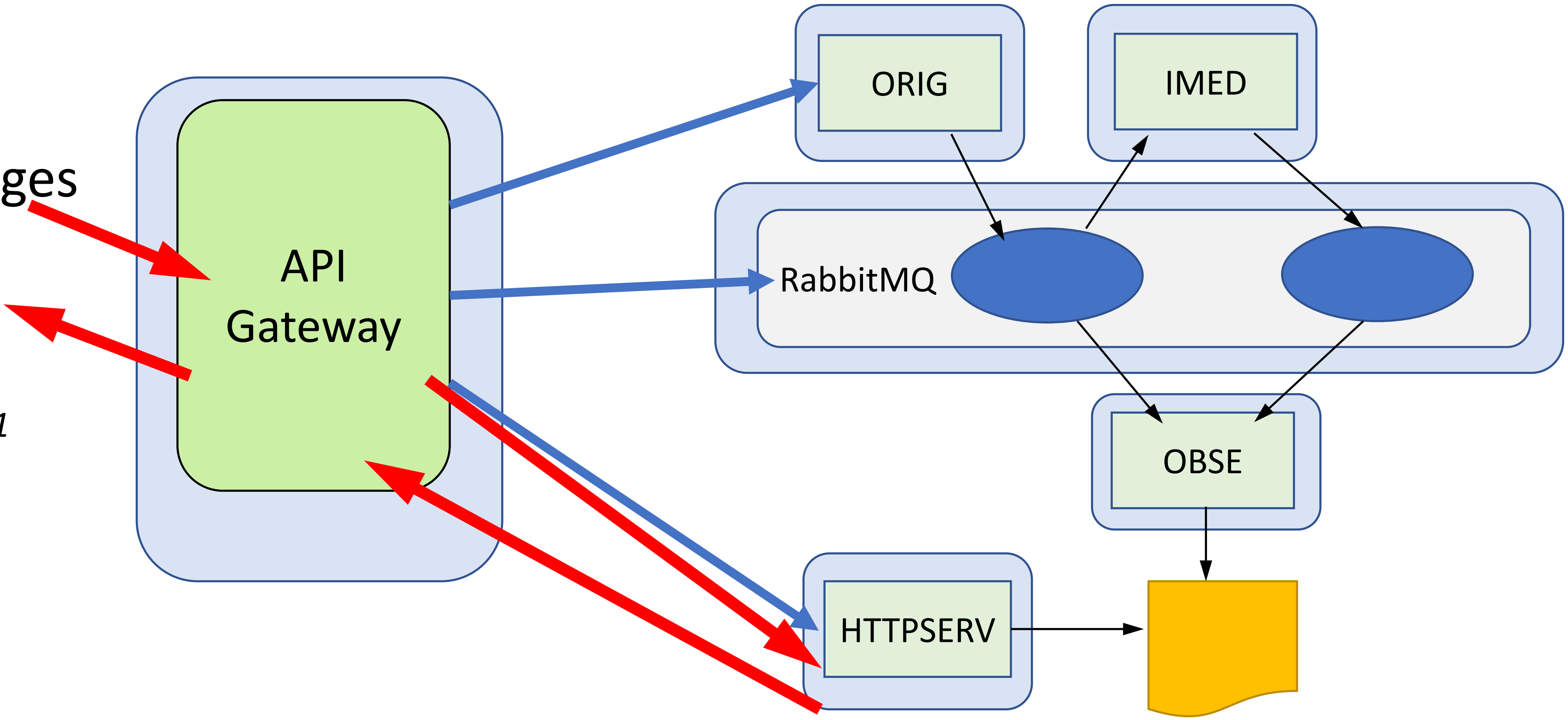
What if OBSE is implemented
by a separate team that does
not now much about other services.


RabbitMQ
OBSE

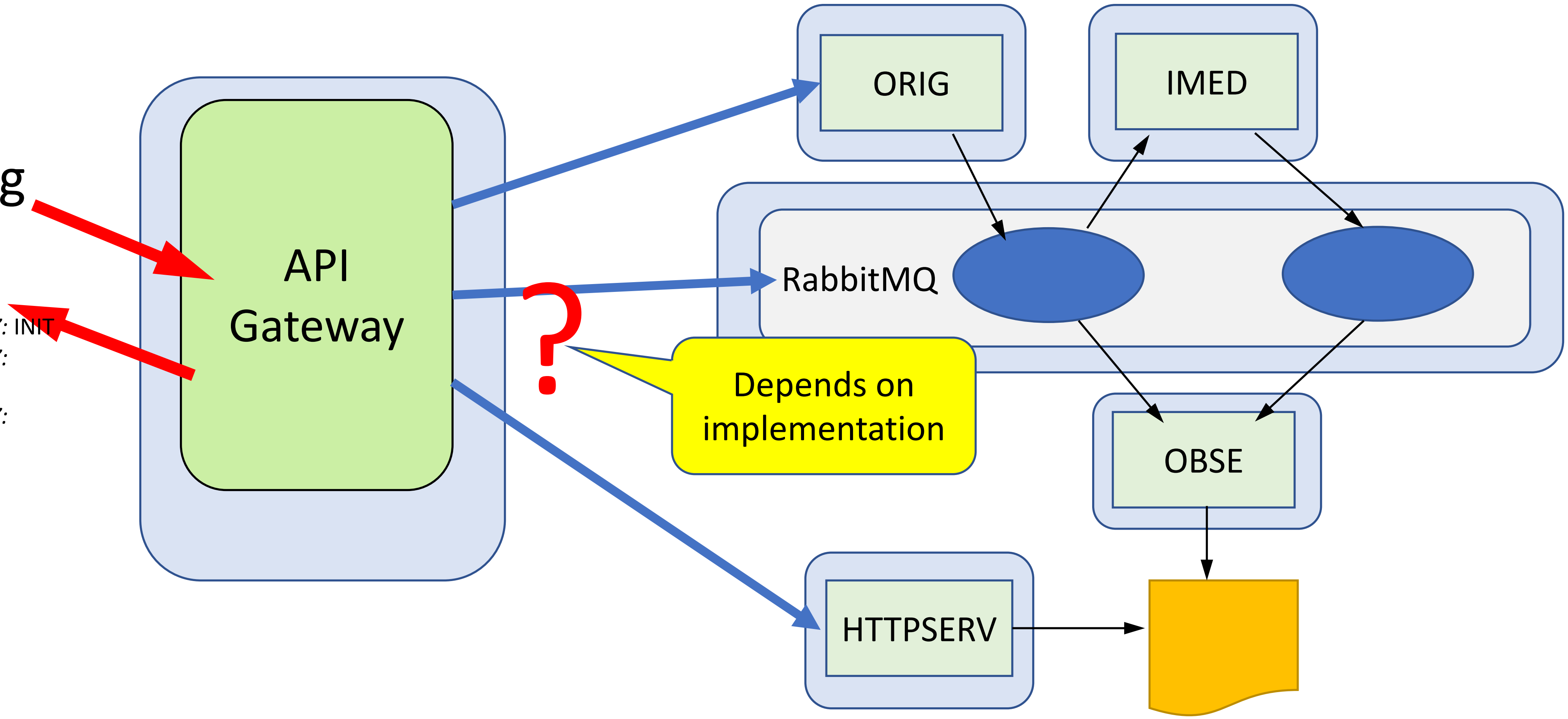
GET /messages

2020-10-01T06:35:01.373Z
Topic my.o: MSG_1

Correct?



GET /run-log



2020-11-01T06:35:01.373Z: INIT
2020-11-01T06:40:01.373Z: PAUSED
2020-11-01T06:40:01.373Z: RUNNING

Correct?

Testing microservices

(https://www.infoq.com/articles/twelve-testing-techniques-microservices-intro/)

Key takeaways

- Because a microservice architecture relies more on over-the-wire (remote) dependencies and less on in-process components, your testing strategy and test environments need to adapt to these changes.
- When testing monoliths using existing techniques like service virtualization, you do not have to test everything together; instead, you can divide and conquer, and test individual modules or coherent groups of components.
- When working with microservices, there are also several more options available, because microservices are deployed typically in environments that use containers like Docker.
- You will need to manage the interdependent components in order to test microservices in a cost and time effective way. You can use test doubles in your microservice tests that pretend to be real dependencies for the purpose of the test.

Automation challenges

- ”…provisioning scripts were considered error-prone and, according to developers, they did not work in some environments…”
- ”…automation of the network in was said to be difficult in addition to dealing with legacy system…”
- ”Networks are pretty hard. Some of the databases are pretty hard too because the old relational databases haven’t been designed to be clustered…”

Automation scripts are programs Infrastructure as code

- ”Infrastructure as code (IaC) is the process of managing and provisioning computer data centers through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools.”
- three approaches to IaC: declarative (functional) vs. imperative (procedural) vs. intelligent (environment aware)

```
tasks:
- name: ensure apache is at the
        latest version
  yum:
    name: httpd
    state: latest
– name: ensure that postgresql is started
    service:
      name: postgresql
      state: started
```

```
apt-get install …
```

Infrastructure as code

All SW engineering principles should be applied.

- Testing
- Maintenance
- Documentation
- Version and configuration management

- Bugs may stop the whole engine

Huge number or tools available

- https://digital.ai/periodic-table-of-devops-tools
- https://landscape.cncf.io