ThoughtWorks®

CONTINUOUS TESTING IN MICROSERVICES

O'Reilly Software Architecture Conference

Sina Jahan - @sinajahan

SETUP WORKSHOP VM & INSTALL PREPREQS

- ☐ Grab a USB
- ☐ Copy content of USB stick to your computer
- ☐ Install Virtual Box and Vagrant

  Use default locations suggested by installers.

  Restart your machine when requested by installers.

There will be slides and hands on stuff!

TOPICS

- ☐ Microservices Refresher
- ☐ Service tests with mocking dependencies
- ☐ Consumer driven contract tests
- ☐ Wrap up

OPEN UP THE SLIDES

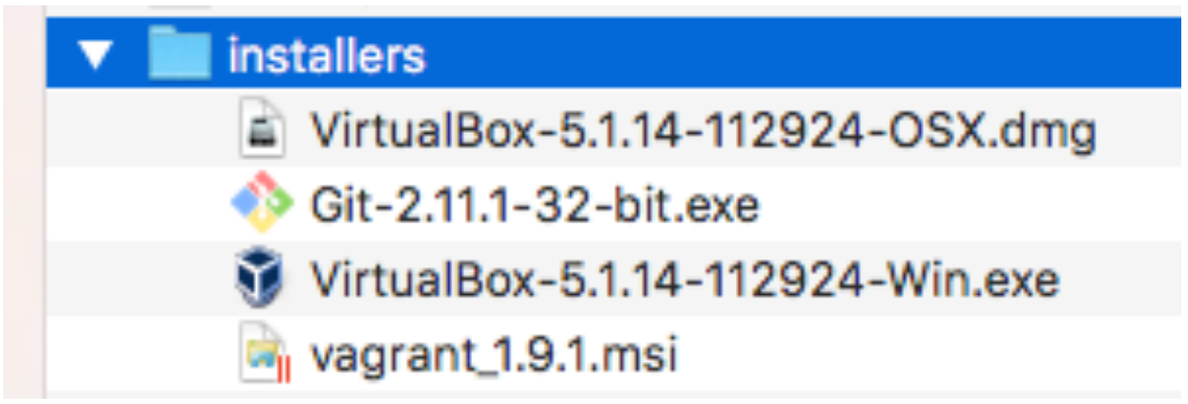
You can also open the slides from the slides folder and follow along or copy and paste commands

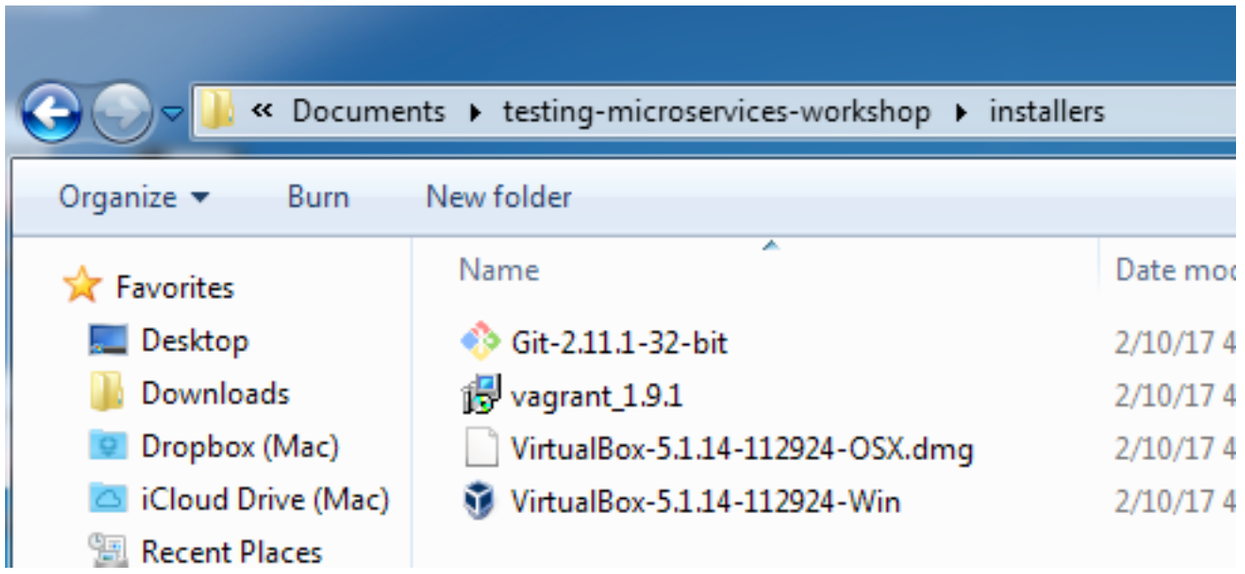
SETUP WORKSHOP VM & INSTALL REQUIREMENTS

☐ Install Virtual Box and Vagrant

Use default locations suggested by installers.

Restart your machine when requested by installers.





Vagrant

Vagrant (software)

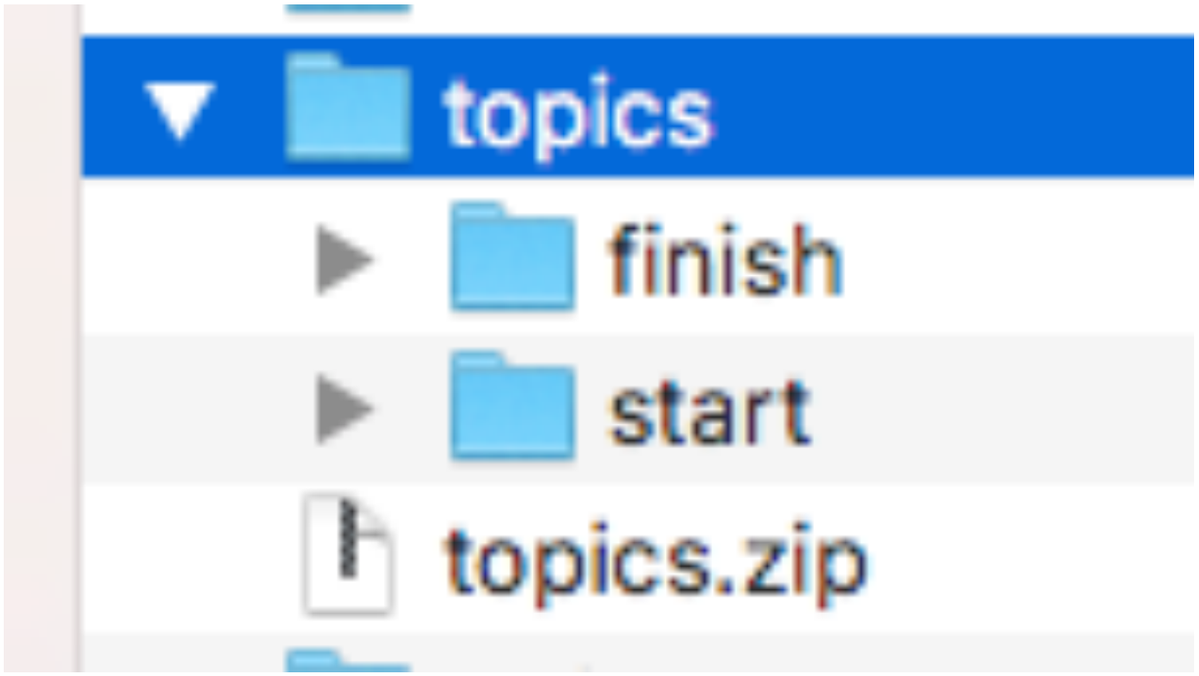
From Wikipedia, the free encyclopedia

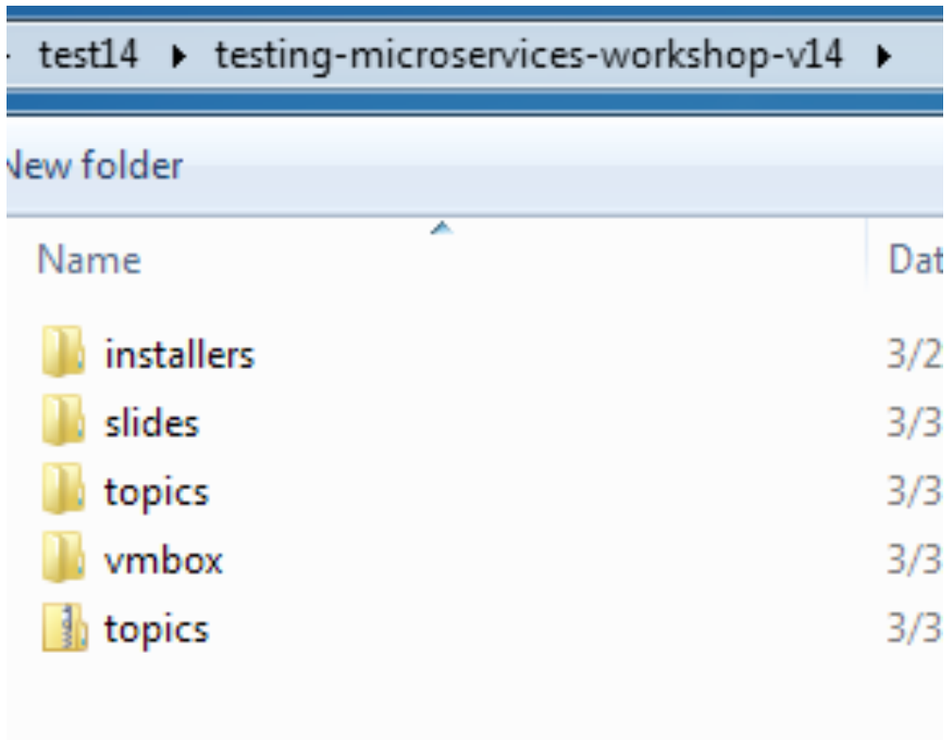
This article **has an unclear citation style**. The references used may be made clearer wi
of citation, footnoting, or external linking. *Violates Wikipedia:External links: "Wikipedia ar*
pages outside Wikipedia (external links), but they should not normally be used in the boo
(Learn how and when to remove this template message)

Vagrant is an open-source software product for building and maintaining portable virtual development environments.[4] The core idea behind its creation lies in the fact that the environment maintenance becomes increasingly difficult in a large project with multiple technical stacks. Vagrant manages all the necessary configurations for the developers in order to avoid the unnecessary maintenance and setup time, and increases development productivity. Vagrant is written in the Ruby language, but its ecosystem supports development in almost all major languages.

SETUP WORKSHOP VM & INSTALL PREPREQS

- ☐ Unzip topics.zip

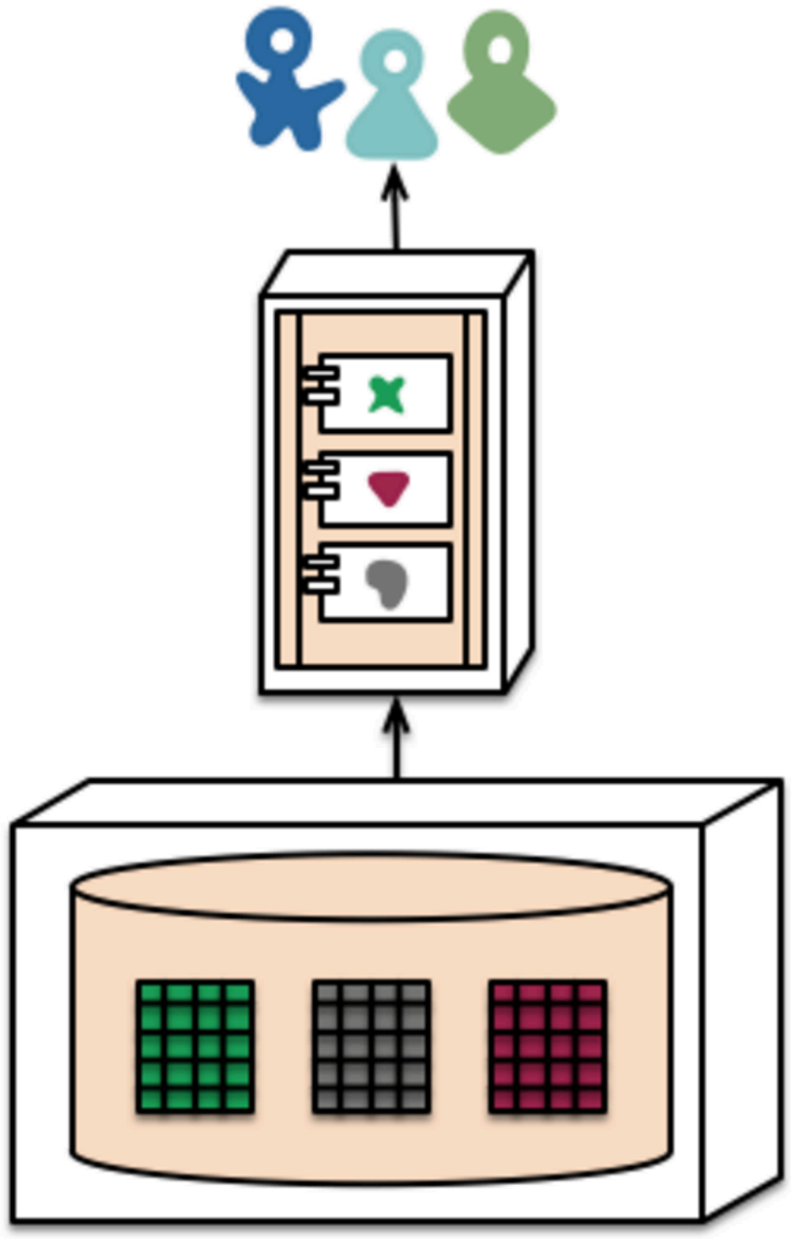




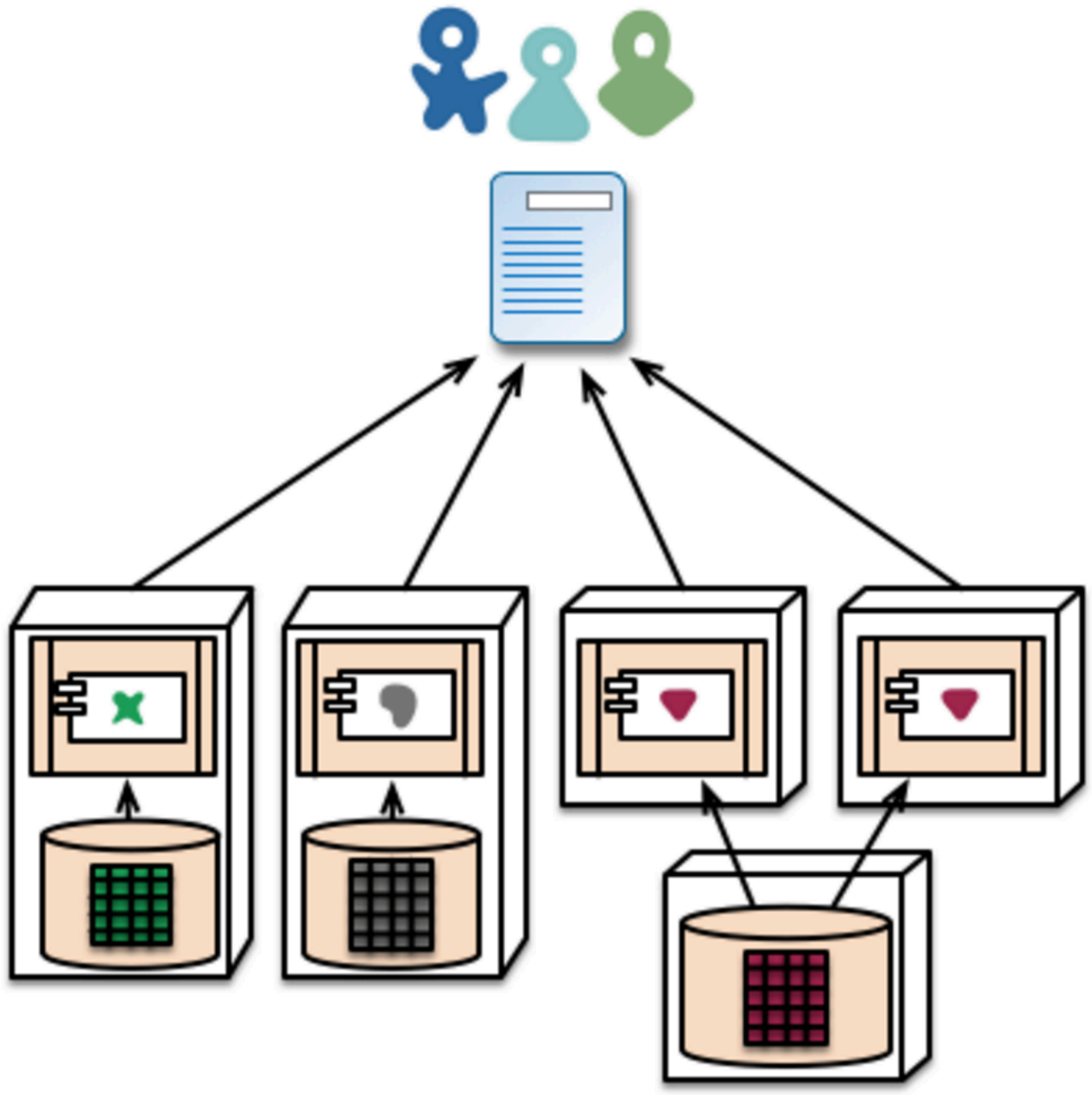
ThoughtWorks®

MICROSERVICES REFRESHER

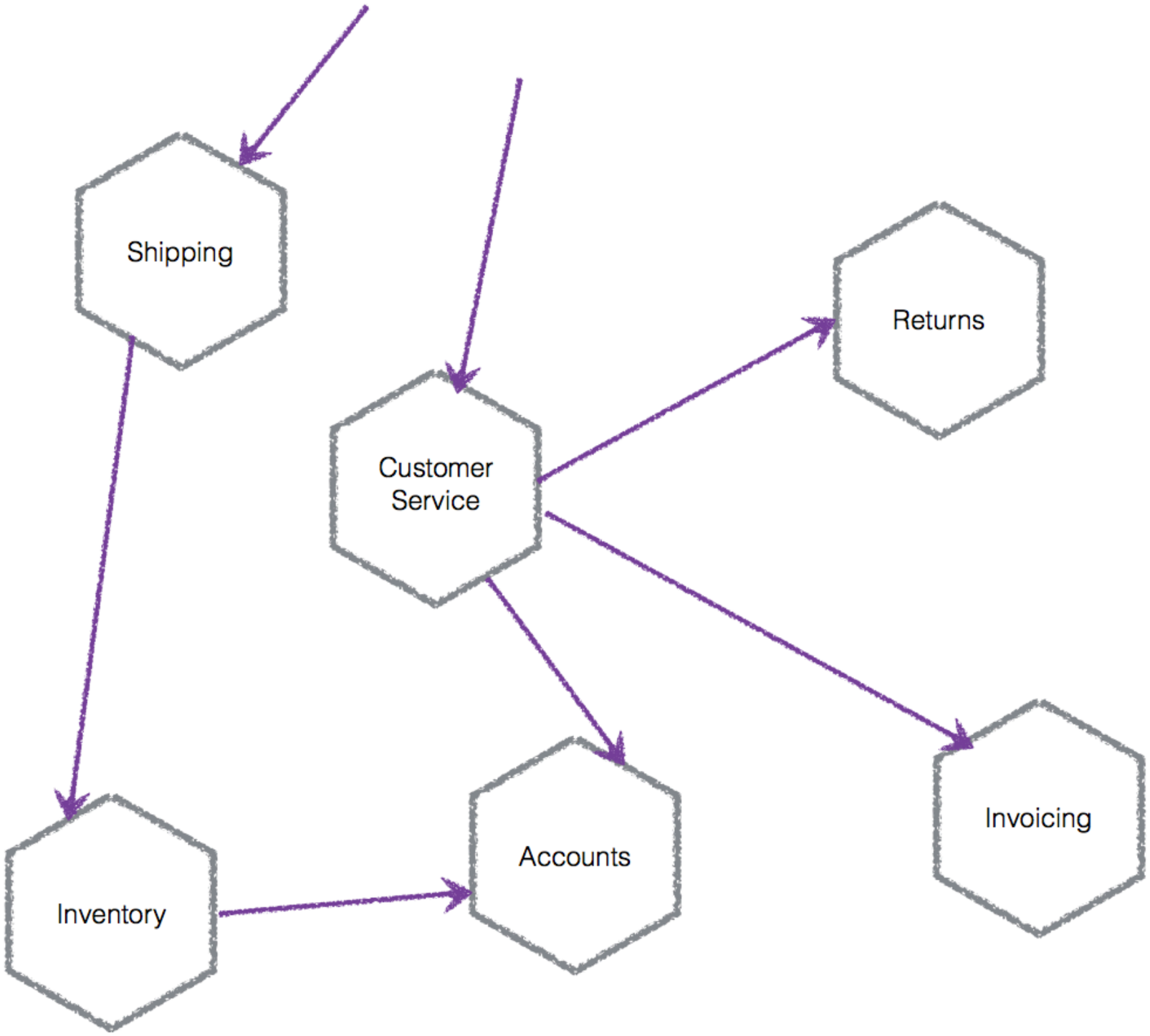
Most applications are written as monoliths



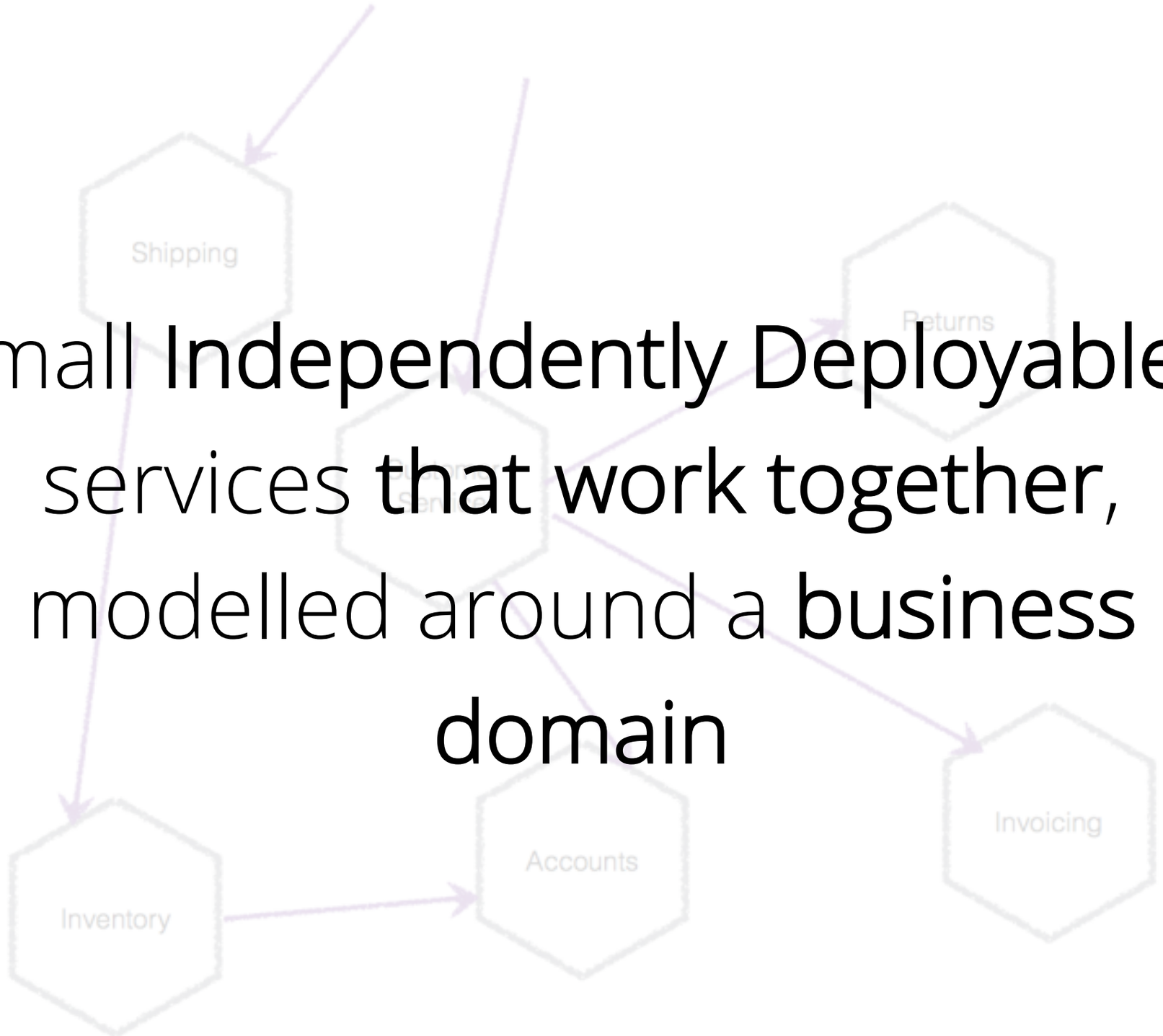
Breakdown to set of services...



Microservices



Microservices



Small Independently Deployable services that work together, modelled around a business domain

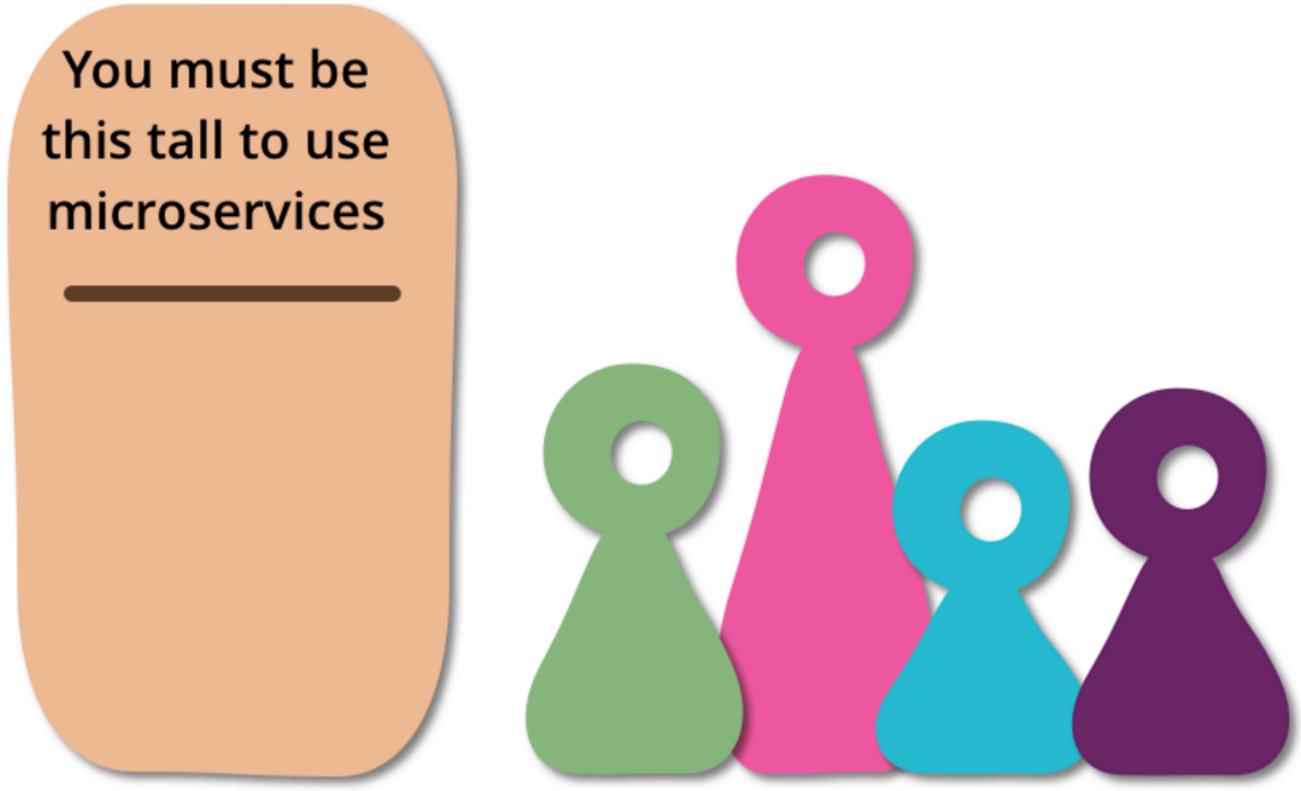
DANGER

DO NOT TOUCH

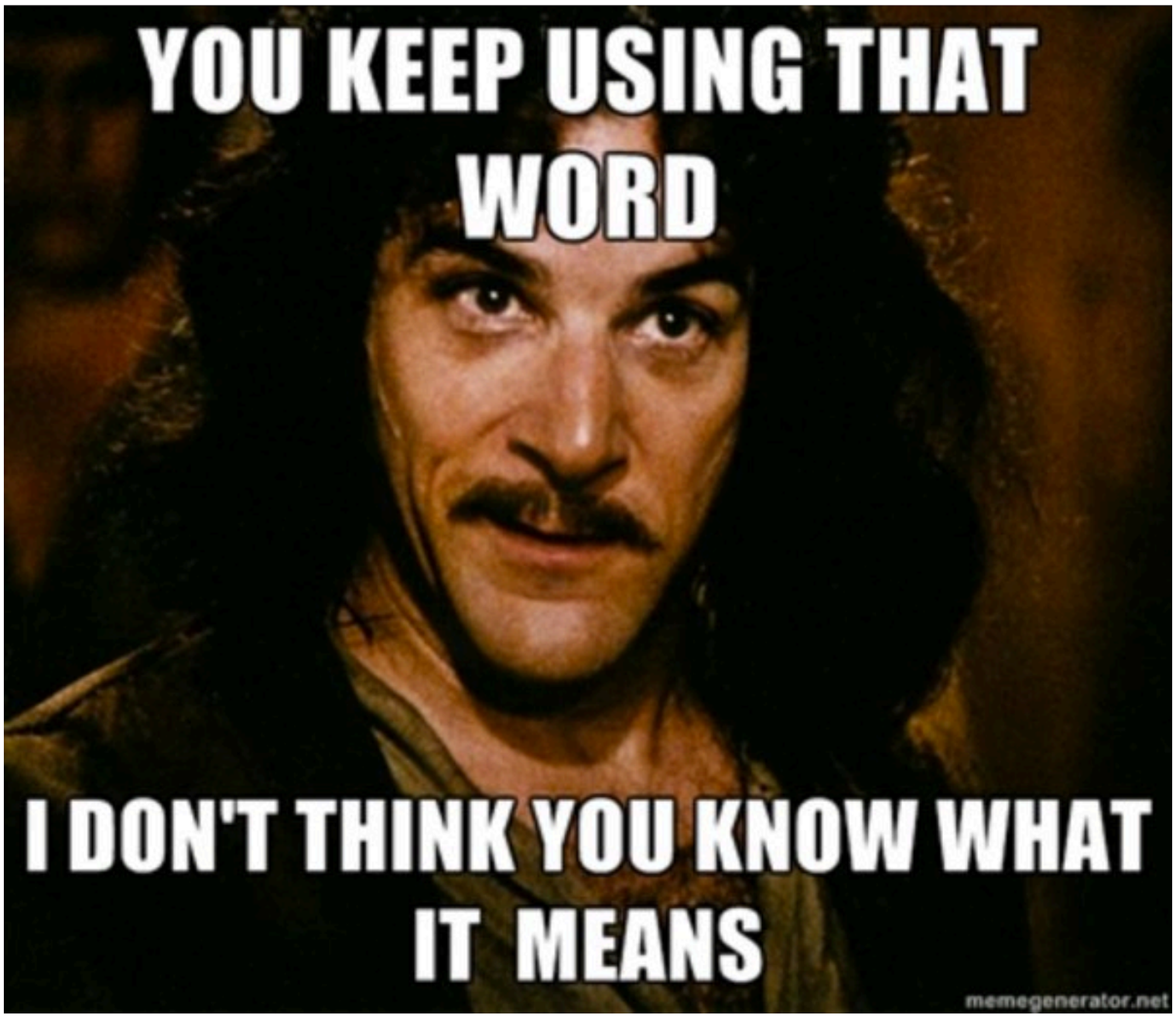
NOT ONLY WILL THIS KILL YOU, IT WILL HURT THE WHOLE TIME YOU ARE DYING

Downsides?

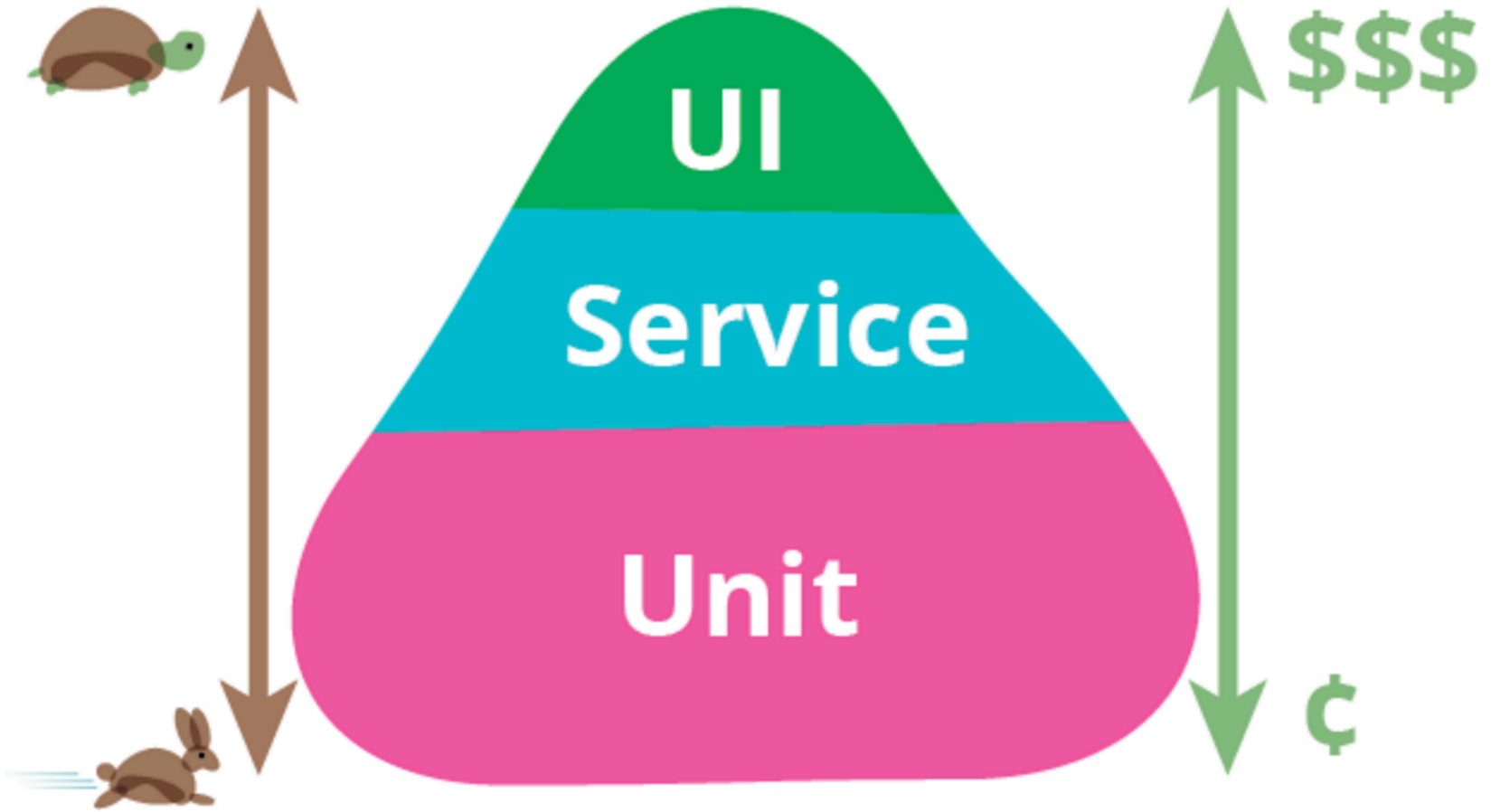
- ☐ Rapid Provisioning
- ☐ Basic Monitoring
- ☐ Rapid Application Deployment
- ☐ DevOps Culture



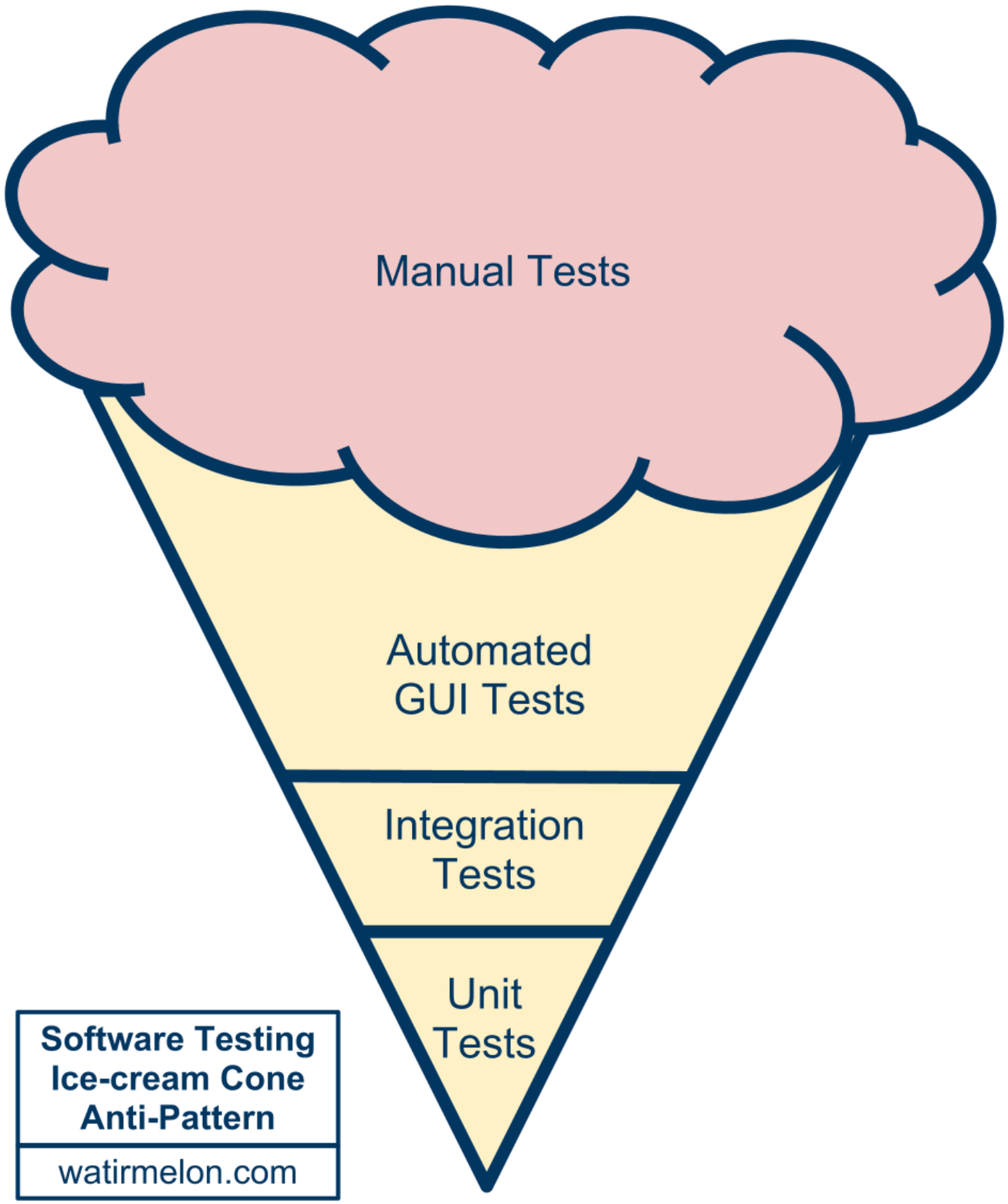
What do YOU mean by integration tests?



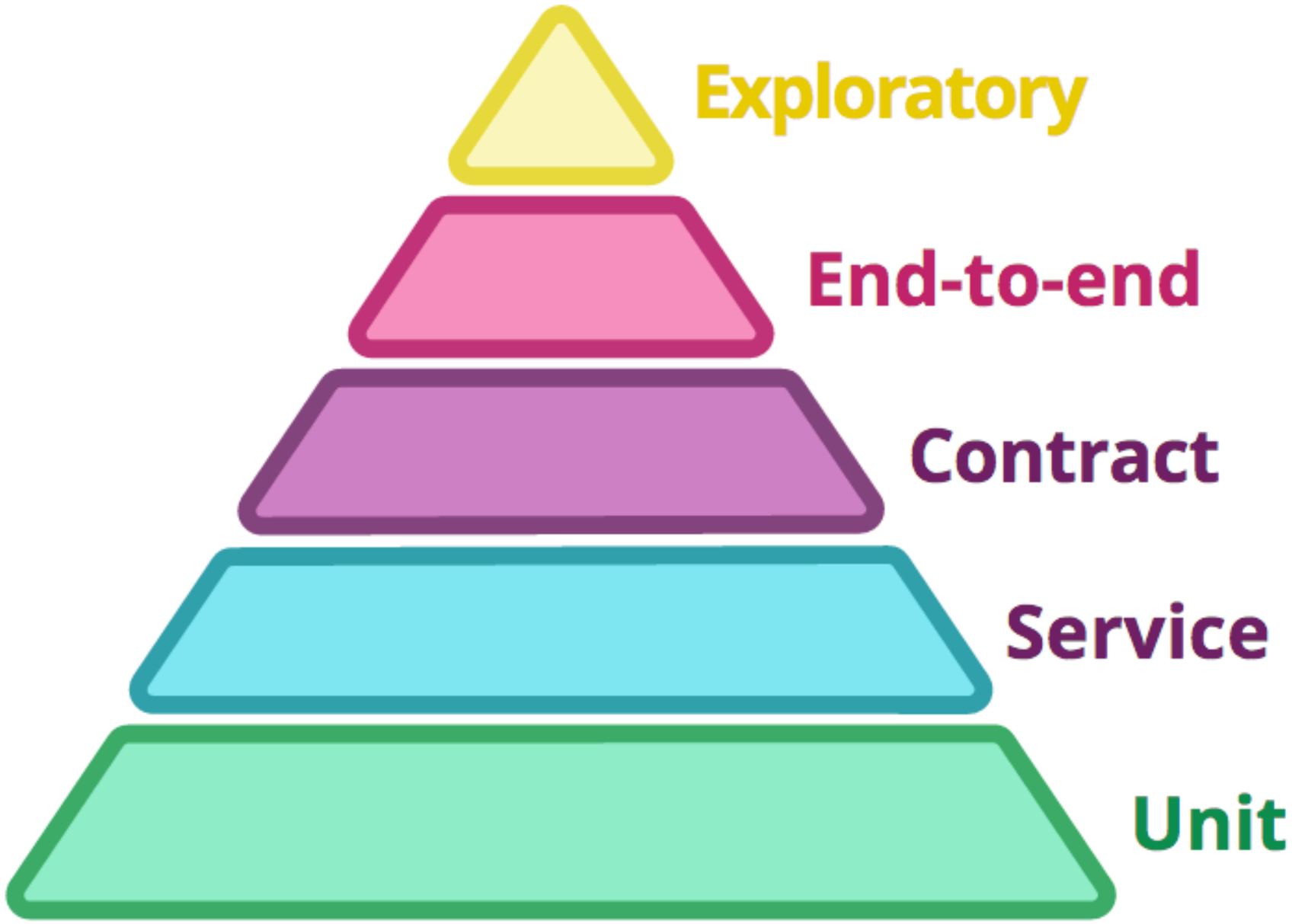
Test Pyramid



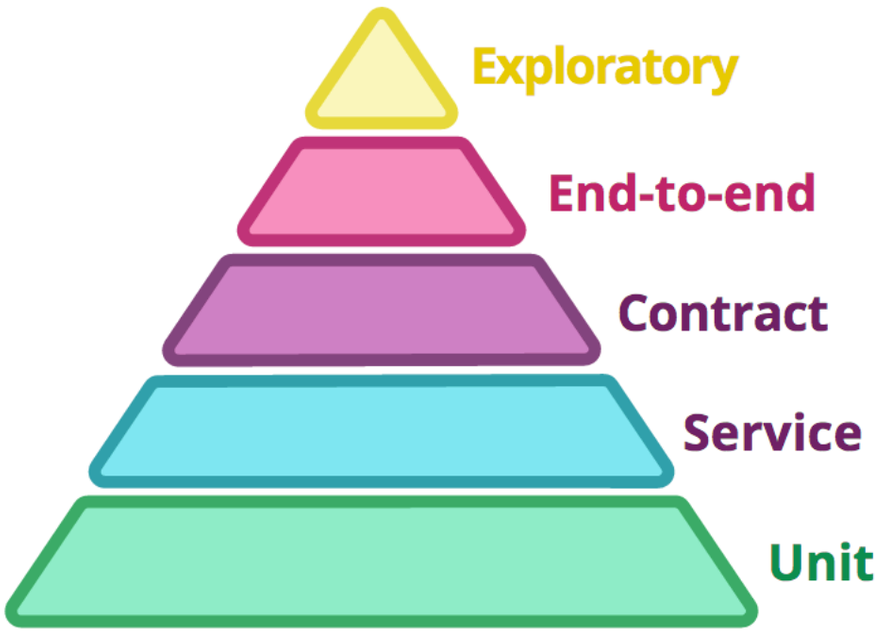
Testing Ice-cream cone

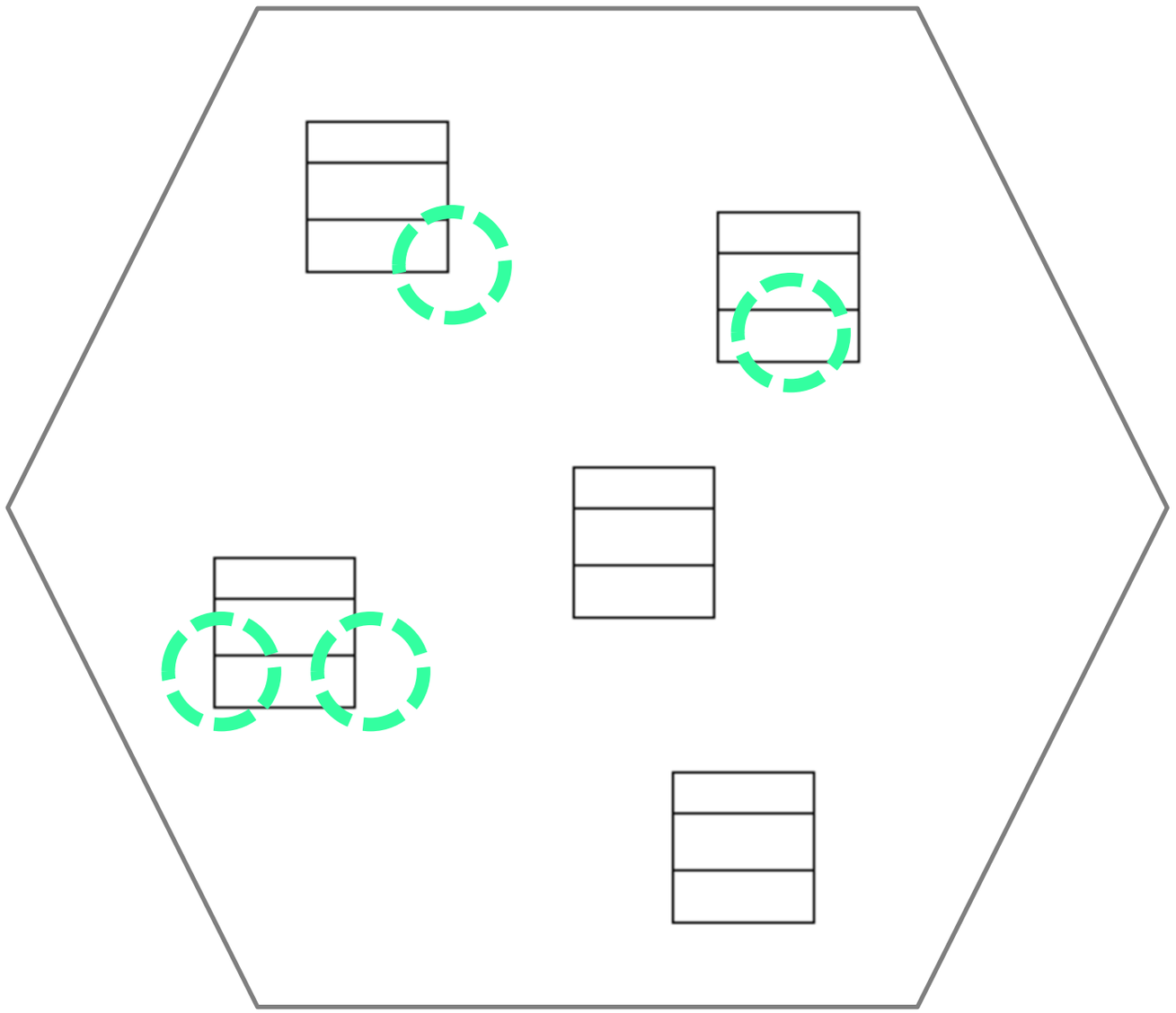


Microservices Test Pyramid

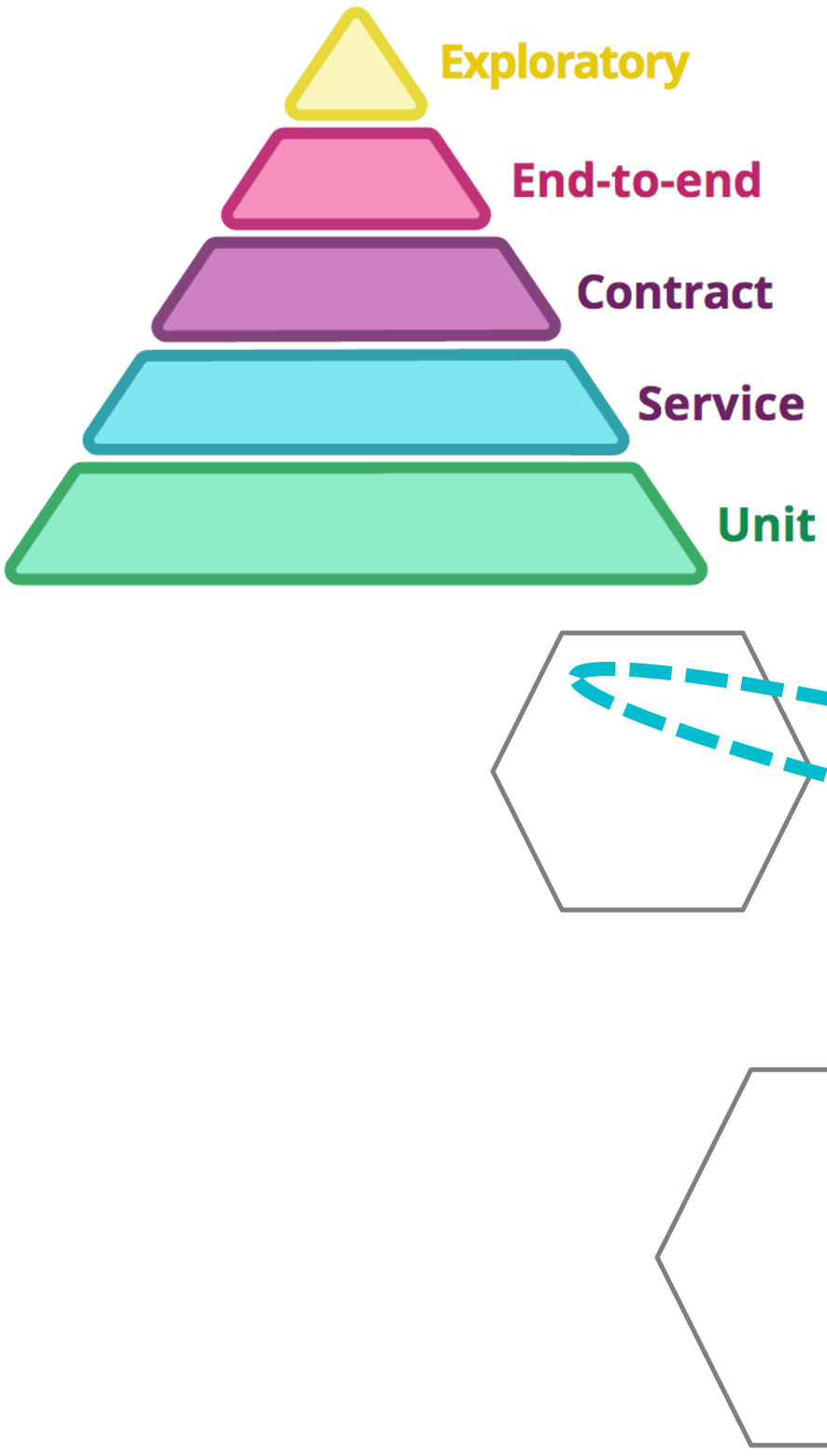


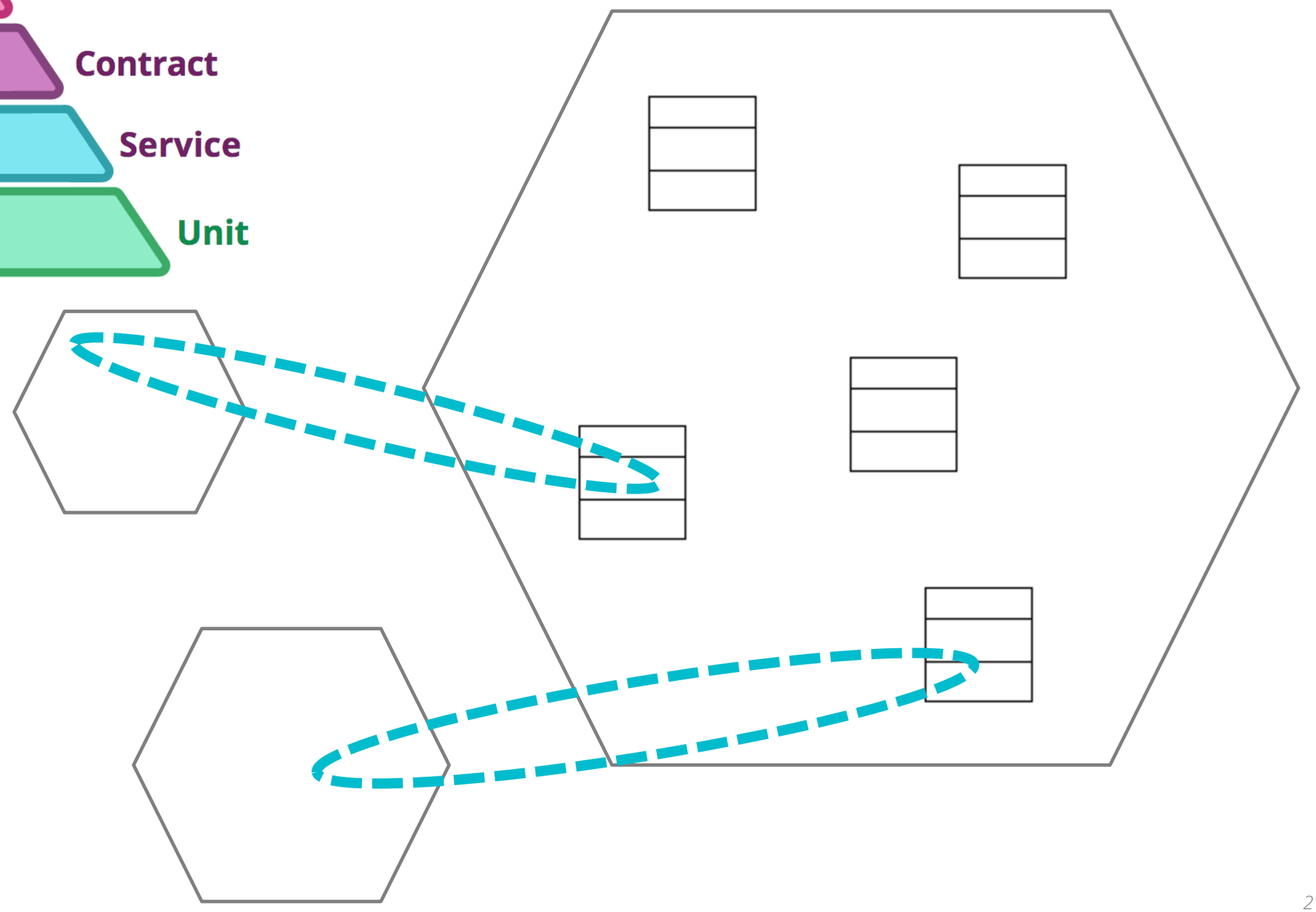
Unit



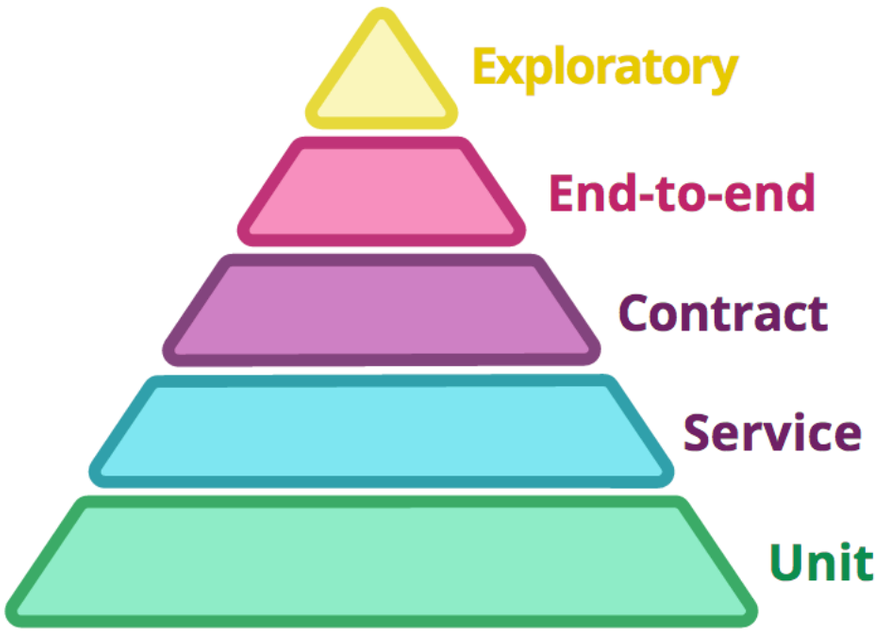


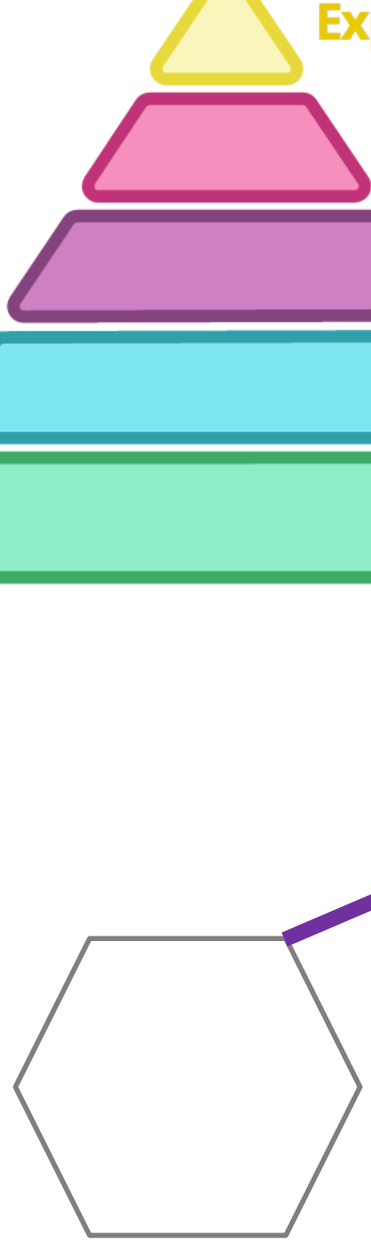
Service



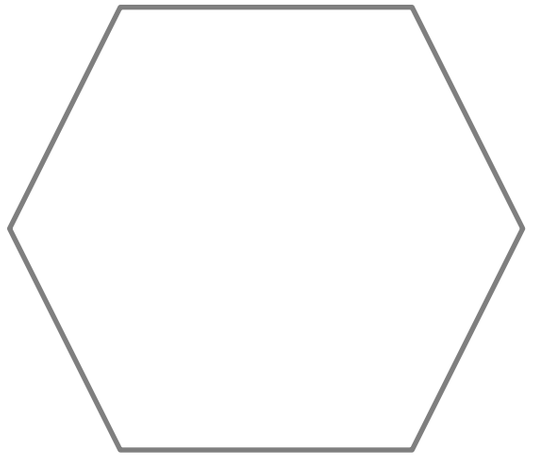


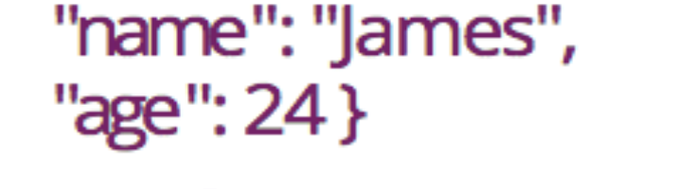
Contract

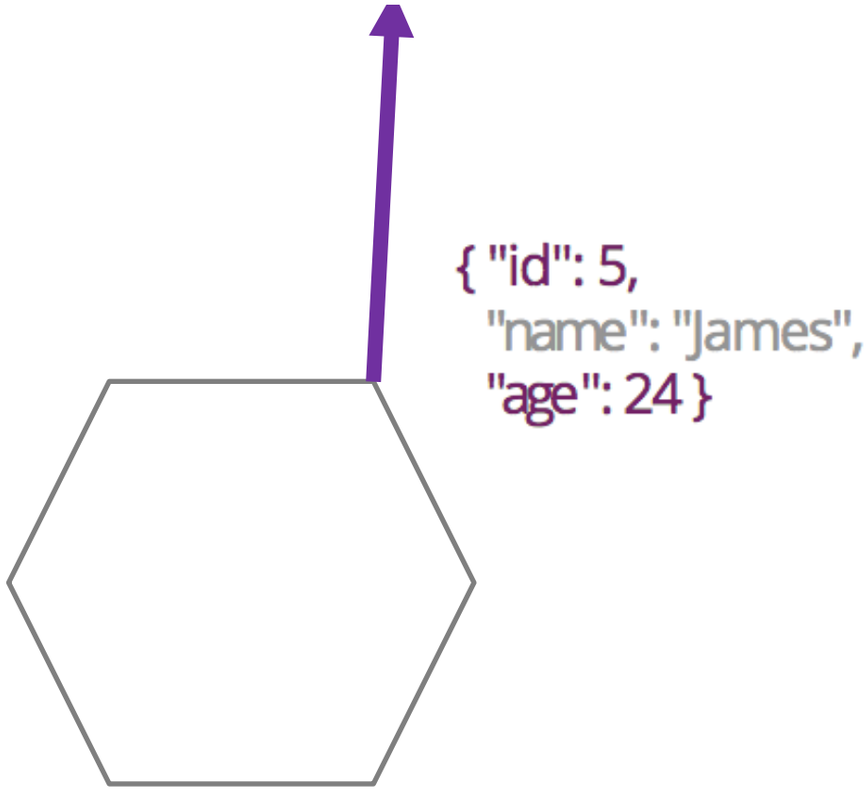

Exploratory
End-to-end
Contract
Service
Unit



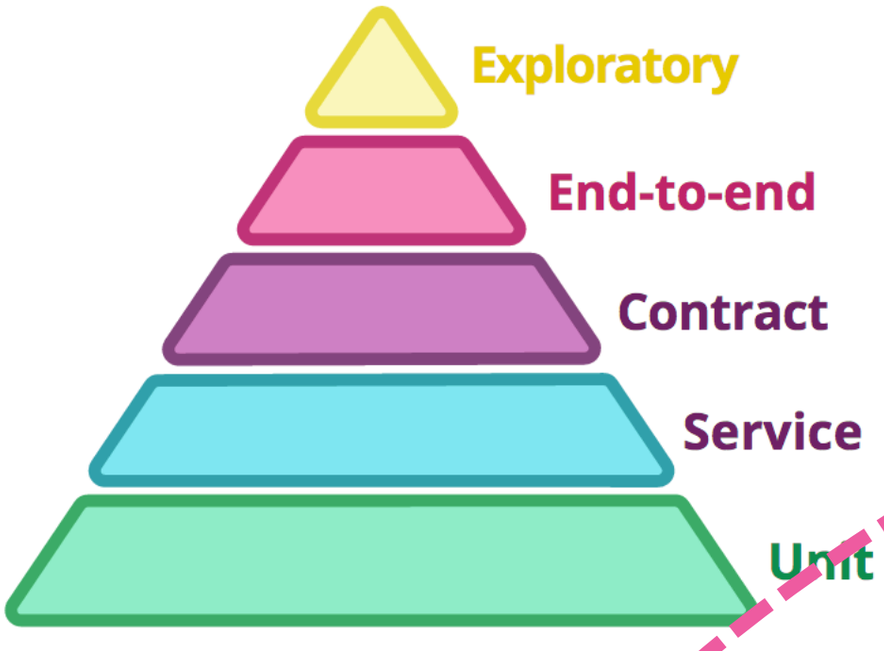
{ "id": 5,
"name": "James",
"age": 24 }

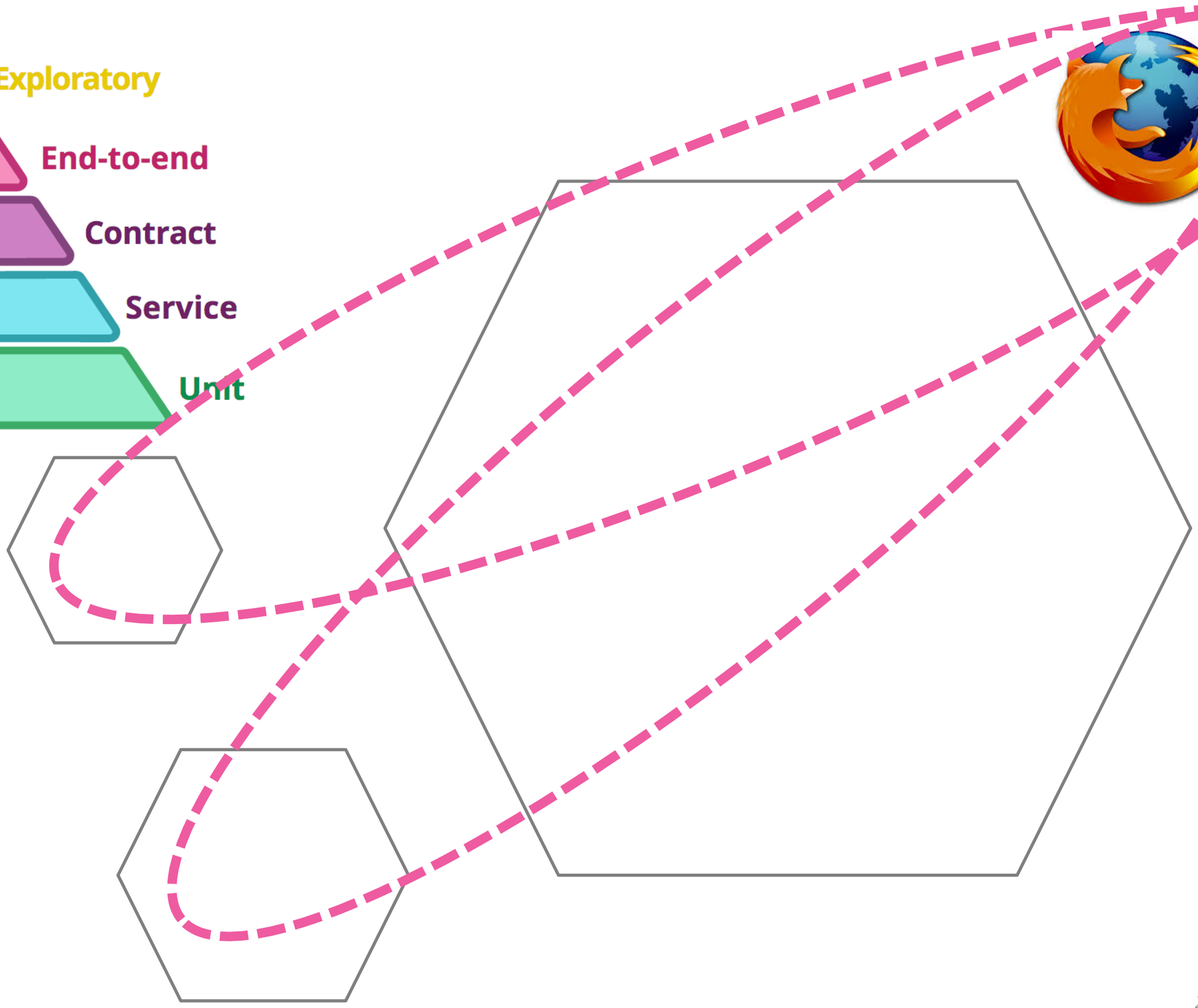



{ "id": 5,
"name": "James",
"age": 24 }


{ "id": 5,
"name": "James",
"age": 24 }

E2E





Just Say No to More End-to-End Tests

Wednesday, April 22, 2015

by Mike Wacker

At some point in your life, you can probably recall a movie that you and your friends all wanted to see, and that you and your friends all regretted watching afterwards. Or maybe you remember that time your team thought they'd found the next "killer feature" for their product, only to see that feature bomb after it was released.

Good ideas often fail in practice, and in the world of testing, one pervasive good idea that often fails in practice is a testing strategy built around end-to-end tests.

Testers can invest their time in writing many types of automated tests, including unit tests, integration tests, and end-to-end tests, but this strategy invests mostly in end-to-end tests that verify the product or service as a whole. Typically, these tests simulate real user scenarios.

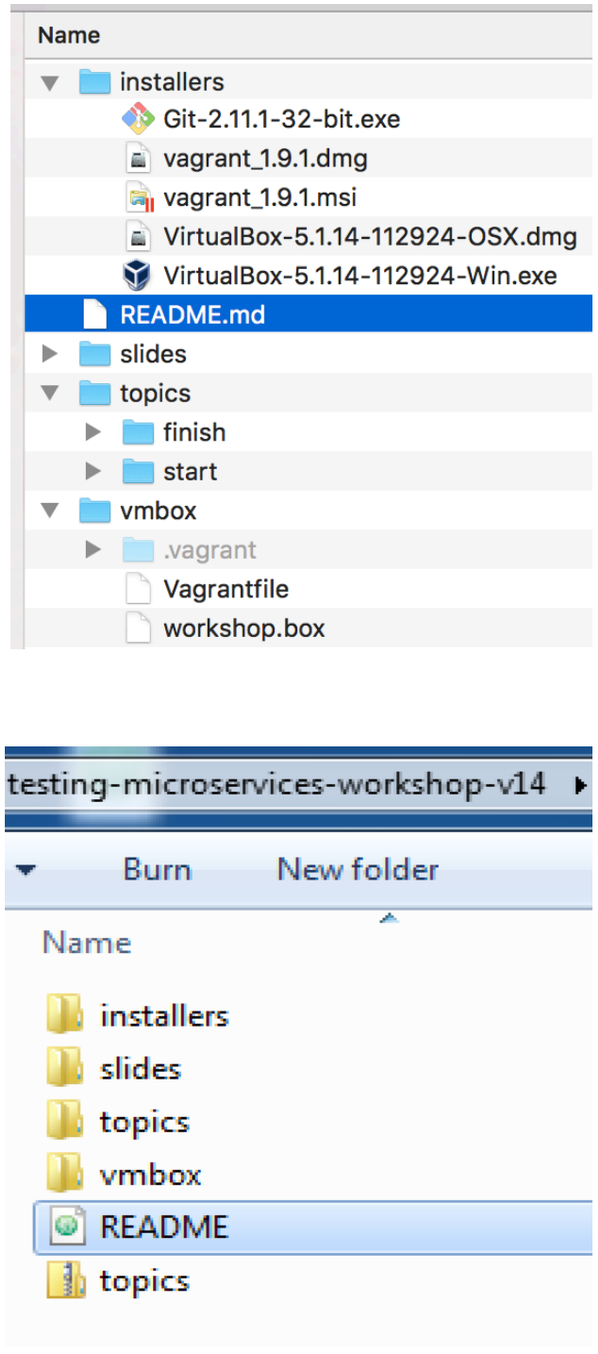
End-to-End Tests in Theory

While relying primarily on end-to-end tests is a bad idea, one could certainly convince a reasonable person that the idea makes sense in theory.

GETTING STARTED WITH WORKSHOP VM

SETUP WORKSHOP VM & INSTALL REQUIREMENTS

☐ Open the README.md file and follow the instructions .



README.md

```
Testing Microservices
=====================

Version: 15 - April 2017

Requirements
------------

Copy and install from `/installers` folder
* VirtualBox
* Vagrant
* Git [Windows only]
** We will use the Git Bash window for ssh during this session

### Mac with good internet connection:
On Mac with good internet connection you can also install these from:
To install HomeBrew go https://brew.sh/
To install Cask go https://caskroom.github.io/
* brew cask install virtualbox
* brew cask install Vagrantgrant

### slides and unziping the topics folder
* Unzip the topics.zip in the same folder that you copied the files.
* Open up the slides in the /slides in your machine and follow the slides
too.

01-getting-started
------------------
```

RUN THE WORKSHOP VM

Start up the workshop VM from workshop-dist/box directory on your machine

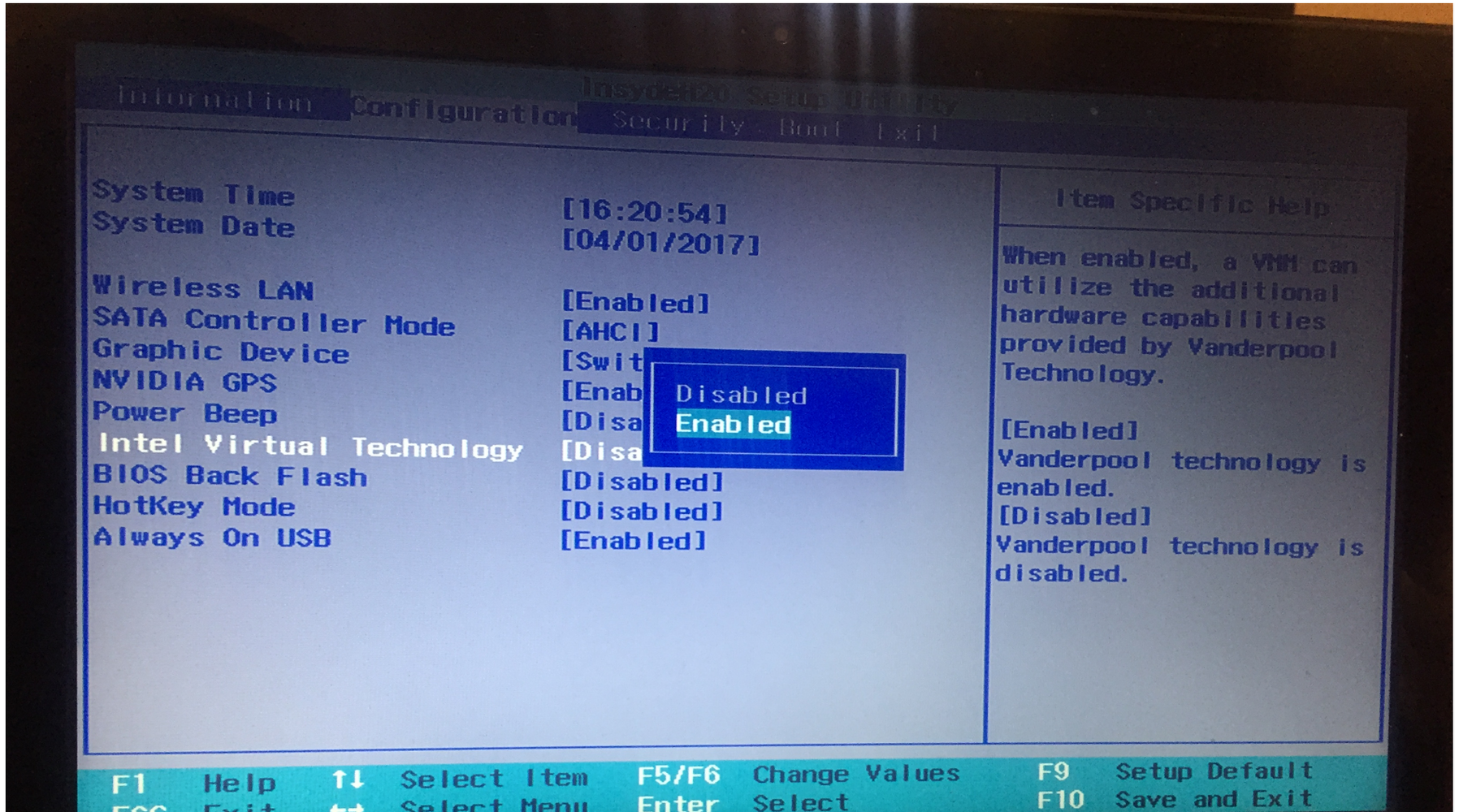
```
$ cd vmbox
vmbox$ vagrant box add workshop workshop.box
vmbox$ vagrant up
vmbox$ vagrant ssh
```

TROUBLESHOOTING

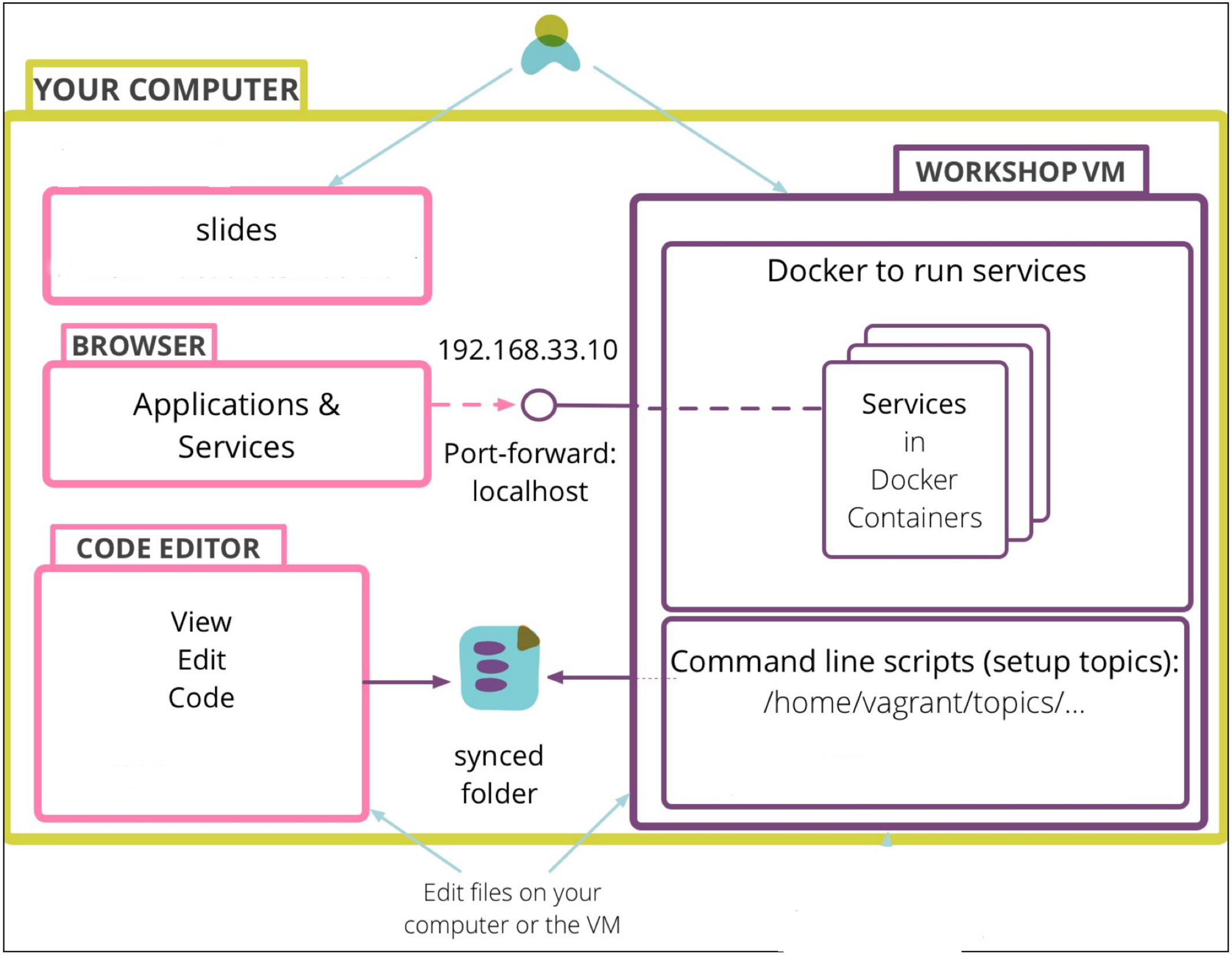
If you are on windows you may get this error:

```
Stderr: VBoxManage.exe: error: VT-x is not available (VERR_VMX_NO_VMX)
VBoxManage.exe: error: Details: code E_FAIL (0x80004005), component ConsoleWrap, interface I
```

You need to go to your BIOS settings and enable “intel virtual technology”:



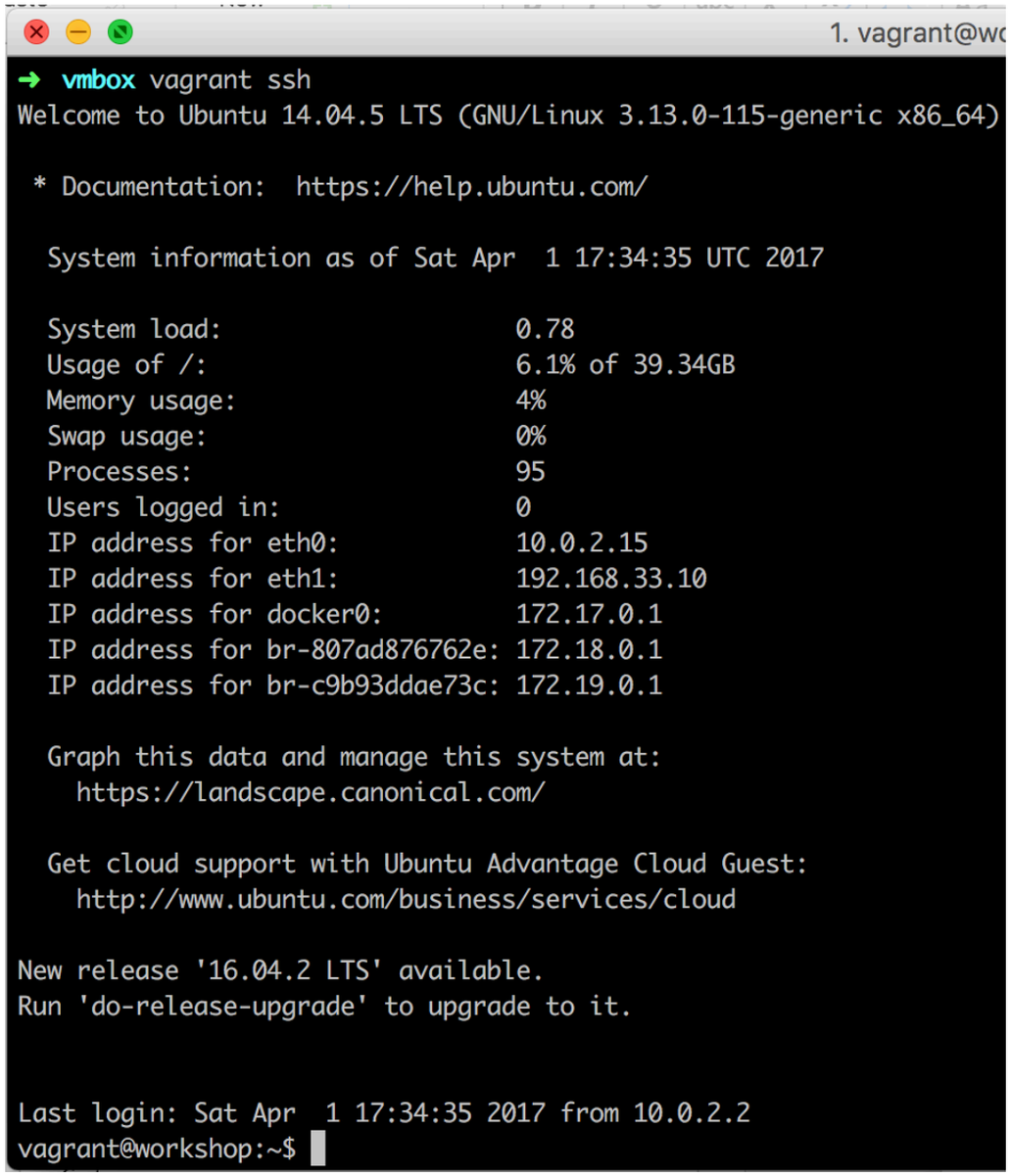
THE PLAYGROUND OVERVIEW



GLOSSARY

`$ vagrant up`	Command to run on your machine from workshop-dist/box directory to start up workshop VM
`$ vagrant ssh`	Command to run on your machine from workshop-dist/box directory to log into workshop vm. This should be called after starting the vm. All consequent commands run on the workshop VM
`vagrant@workshop:~$`	Prompt displayed when running commands on microservices VM
`/home/vagrant/topics/{state}/{topic_name}`	We start each subject in ‘start’ state.

SUCCESSFUL SSH SESSION ON VM



RUN GETTING STARTED

Run the getting_started topic script

```
vagrant@workshop:~$ cd ~/topics/start/01-getting-started

vagrant@workshop:~$ cd helloworld

vagrant@workshop:helloworld $ npm start
```

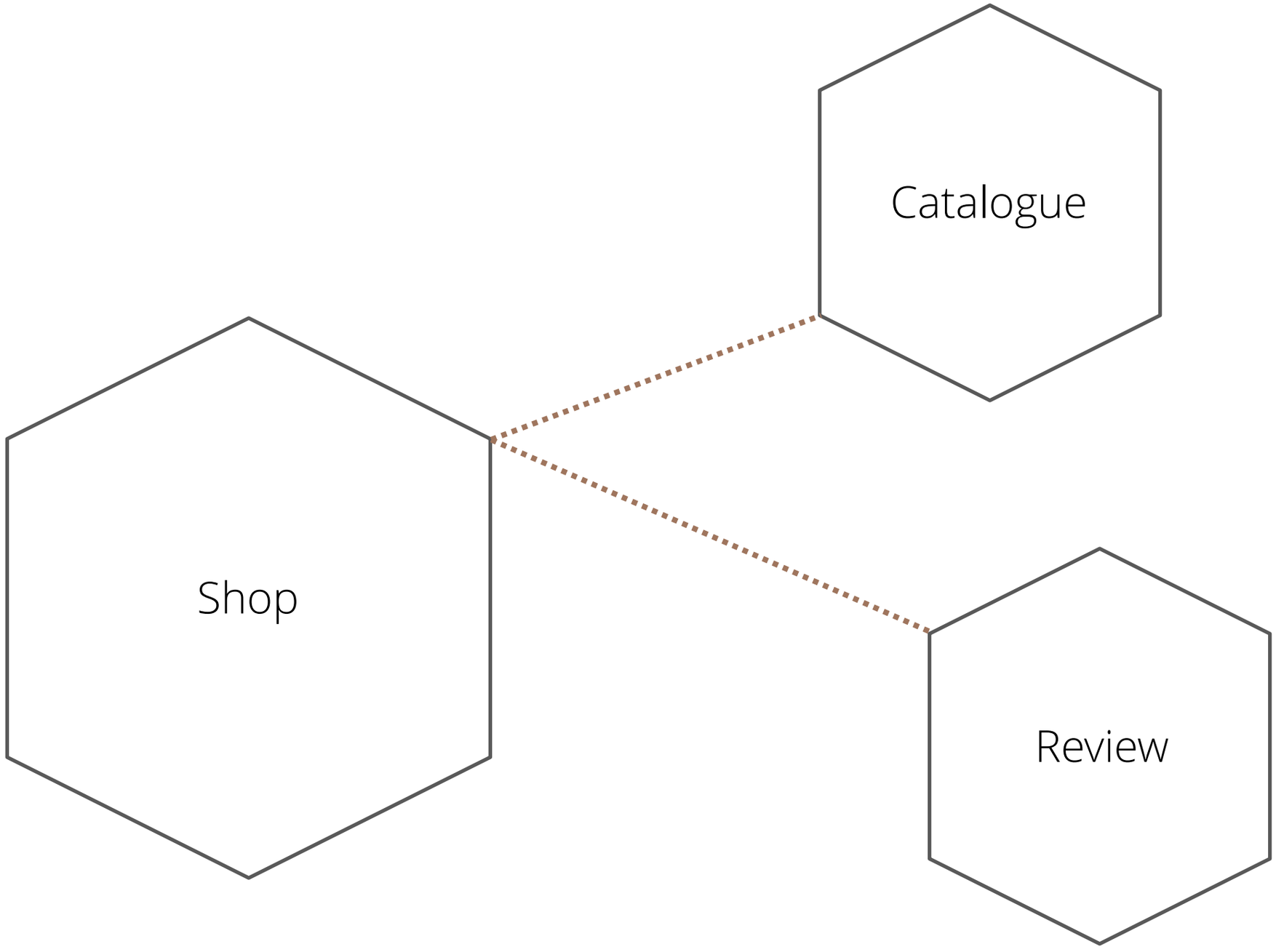
Go to
http://192.168.33.10:9080 and see the ”hello world”

```
vagrant@workshop:~$ npm test
```

Get the first unit test passing

OUR SET OF MICROSERVICES

OUR MINI AMAZON



RUN SERVICE TESTING

Go to service testing

```
vagrant@workshop:~$ cd ~/topics/start/02-service-testing
$ docker load < ./docker-images/node-6
$ docker-compose up
```

Go to
http://192.168.33.10:9081/products and see the catalogue

http://192.168.33.10:9082/reviews and see the review

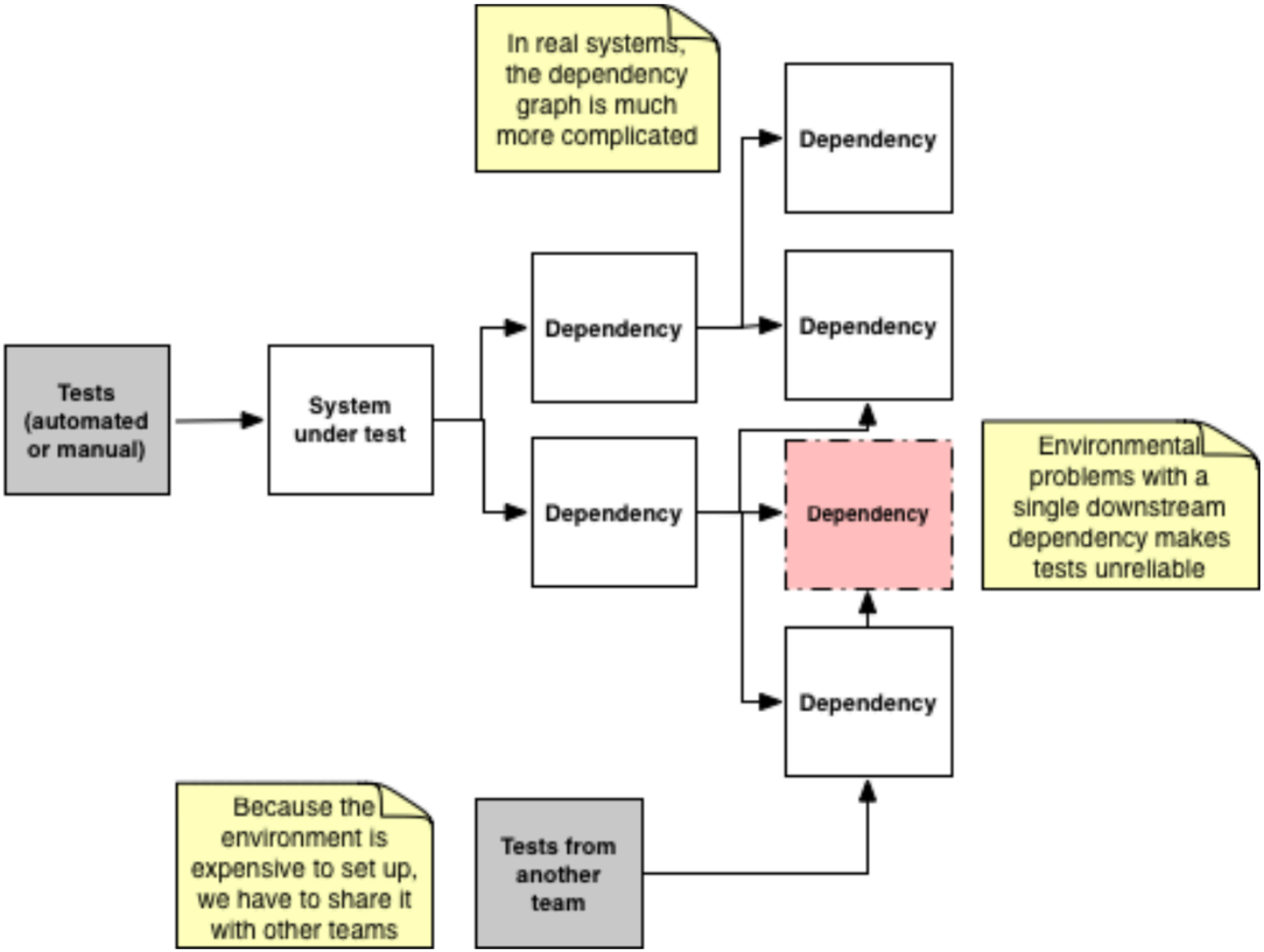
http://192.168.33.10:9083/shop and see the shop

RUN SERVICE TESTING

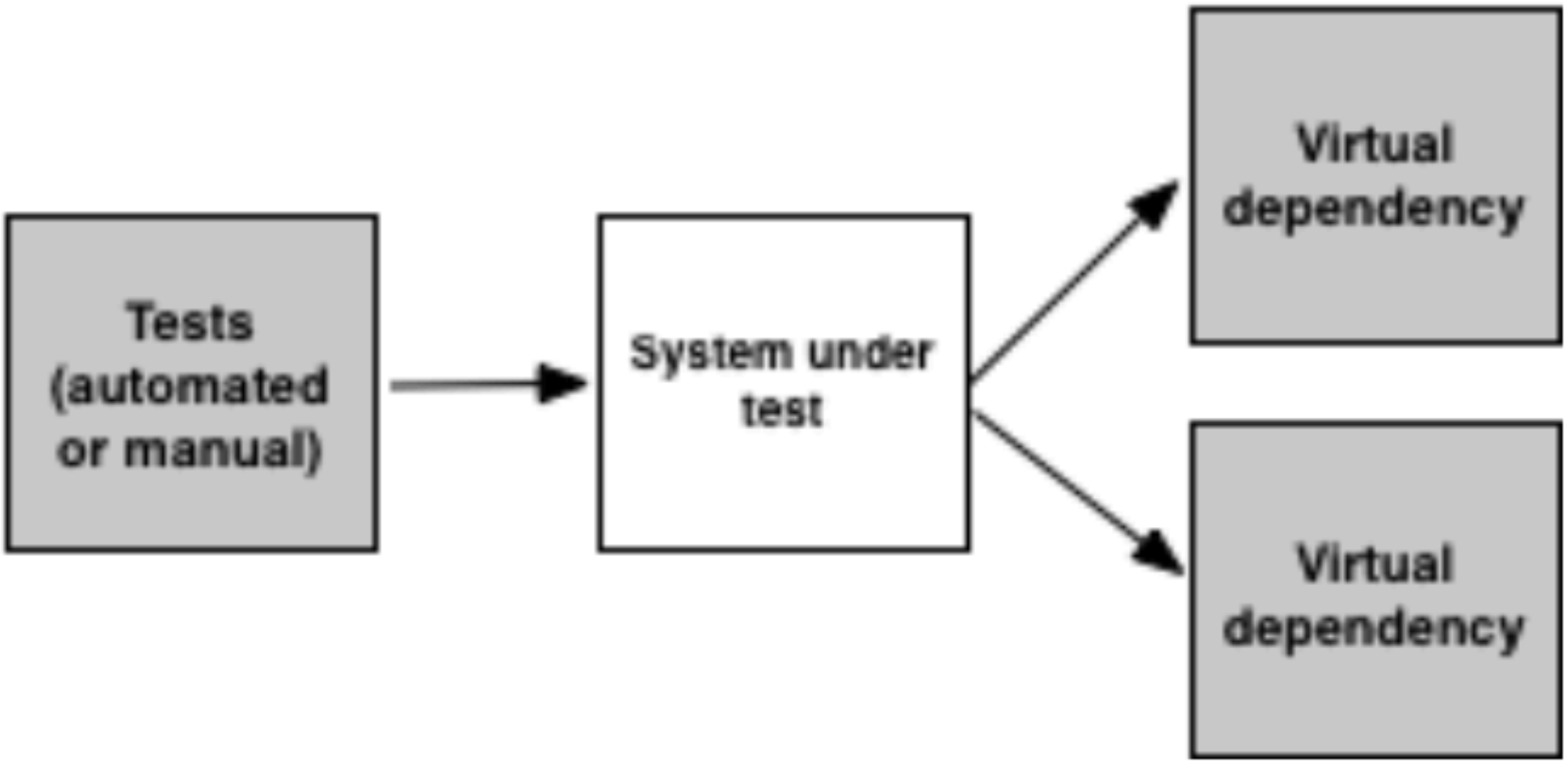
Take a look at tests shop

```
vagrant@workshop:~$ cd ~/topics/start/02-service-testing
vagrant@workshop:~$ cd shop
vagrant@workshop:~$ npm test
```

Problems with E2E



Service Virtualization



Service Virtualization

mountebank - over the wire test doubles

Fork me on GitHub

home imposters logs config Search...

the apothecary

- getting started
- examples
- client libraries
- install options
- command line
- security
- faqs
- support
- glossary

api:

- overview
- contracts
- mock verification
- stubs
  - proxies
  - injection
  - behaviors
- stub predicates
  - xpath
  - json
  - jsonpath

Welcome, friend

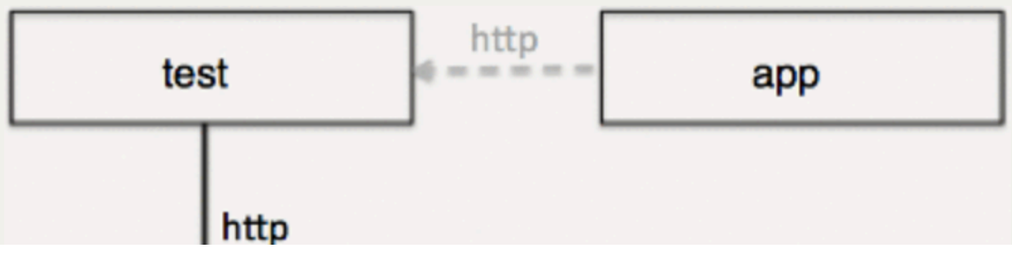
mountebank is the first open source tool to provide cross-platform, multi-protocol test doubles over the wire. Simply point your application under test to mountebank instead of the real dependency, and test like you would with traditional stubs and mocks.

mountebank is the most capable open source service virtualization tool in existence, and will cure what ails you, guaranteed.

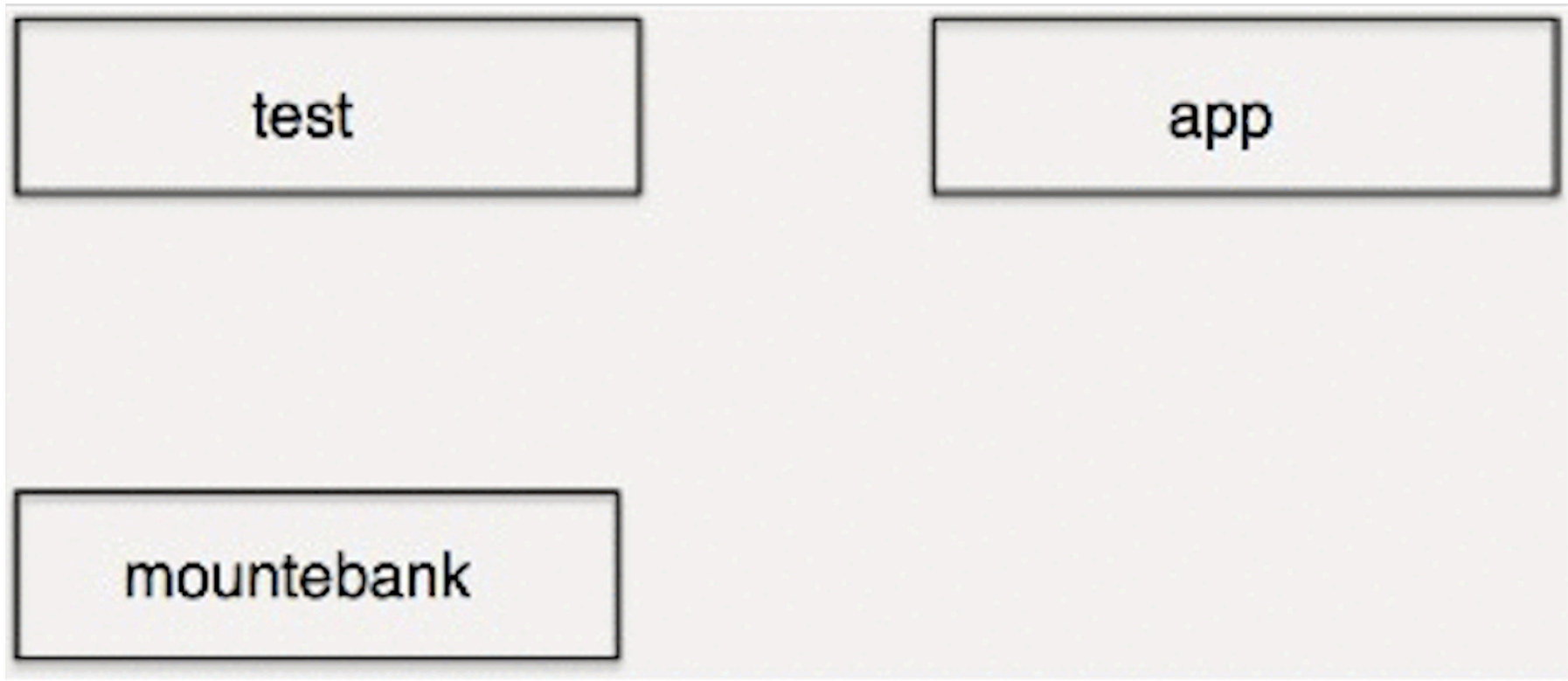
How it works

mountebank employs a legion of *imposters* to act as on-demand test doubles. Your test communicates to mountebank over http using the api to set up stubs, record and replay proxies, and verify mock expectations. In the typical use case, each test will start an imposter during test setup and stop an imposter during test teardown, although you are also welcome to configure mountebank at startup using a config file.

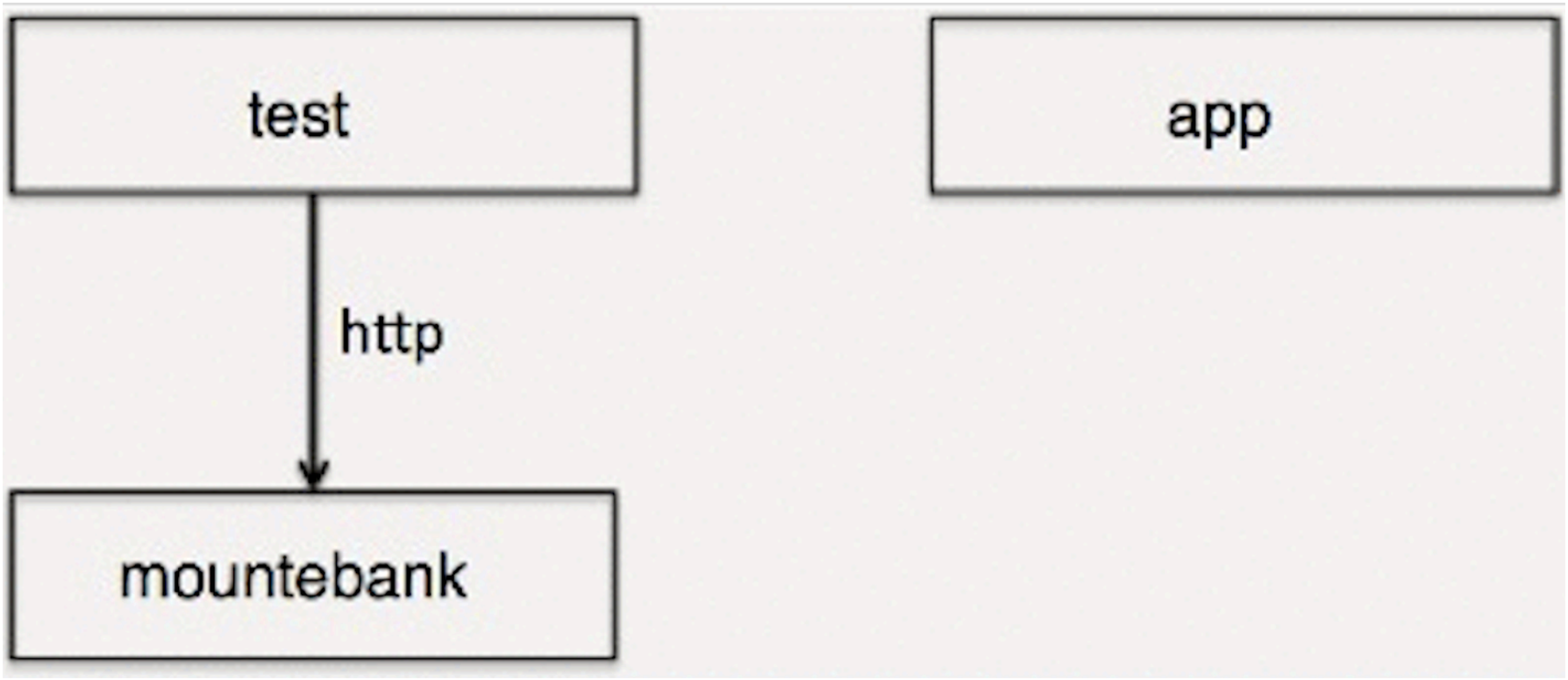
mountebank employs several types of imposters, each responding to a specific protocol. Typically, your test will tell the imposter which port to bind to, and the imposter will open the corresponding socket.



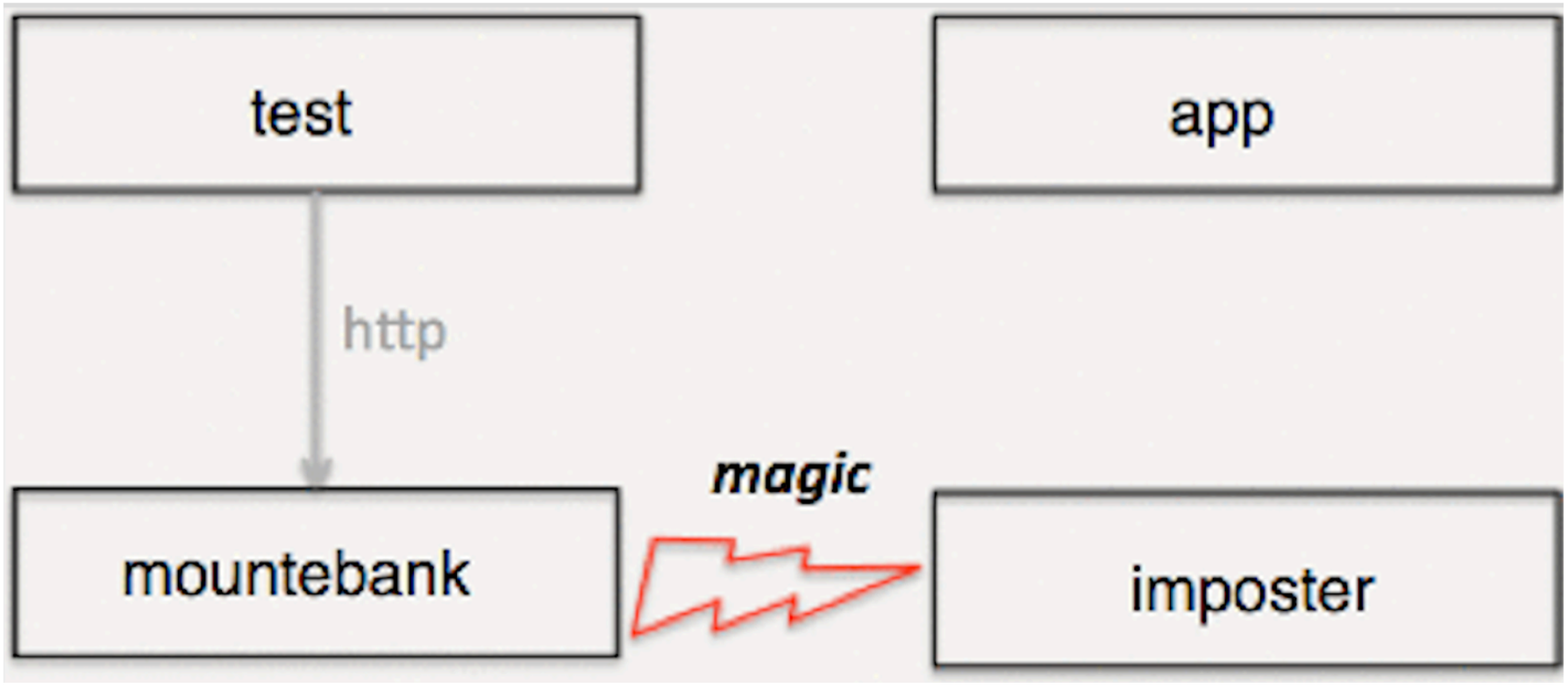
Service Virtualization



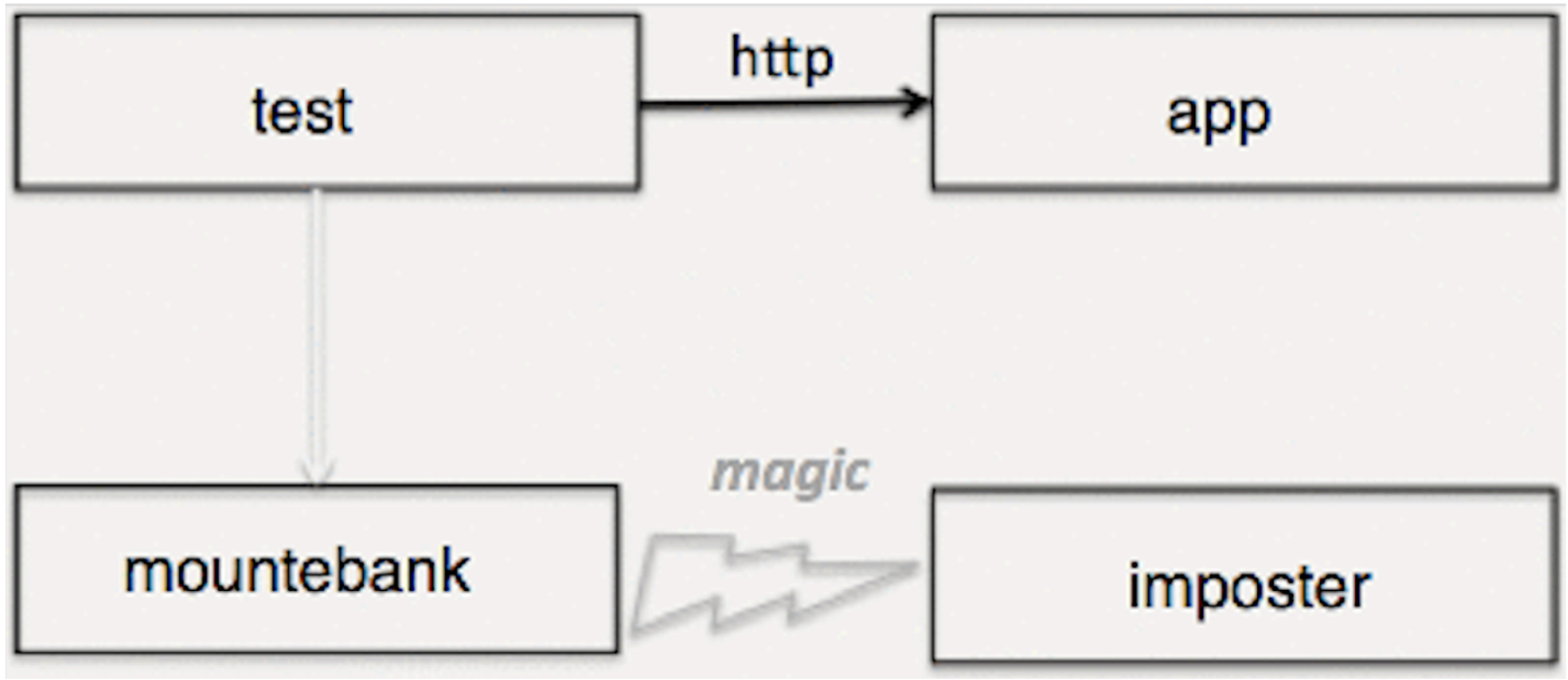
TEST SETUP



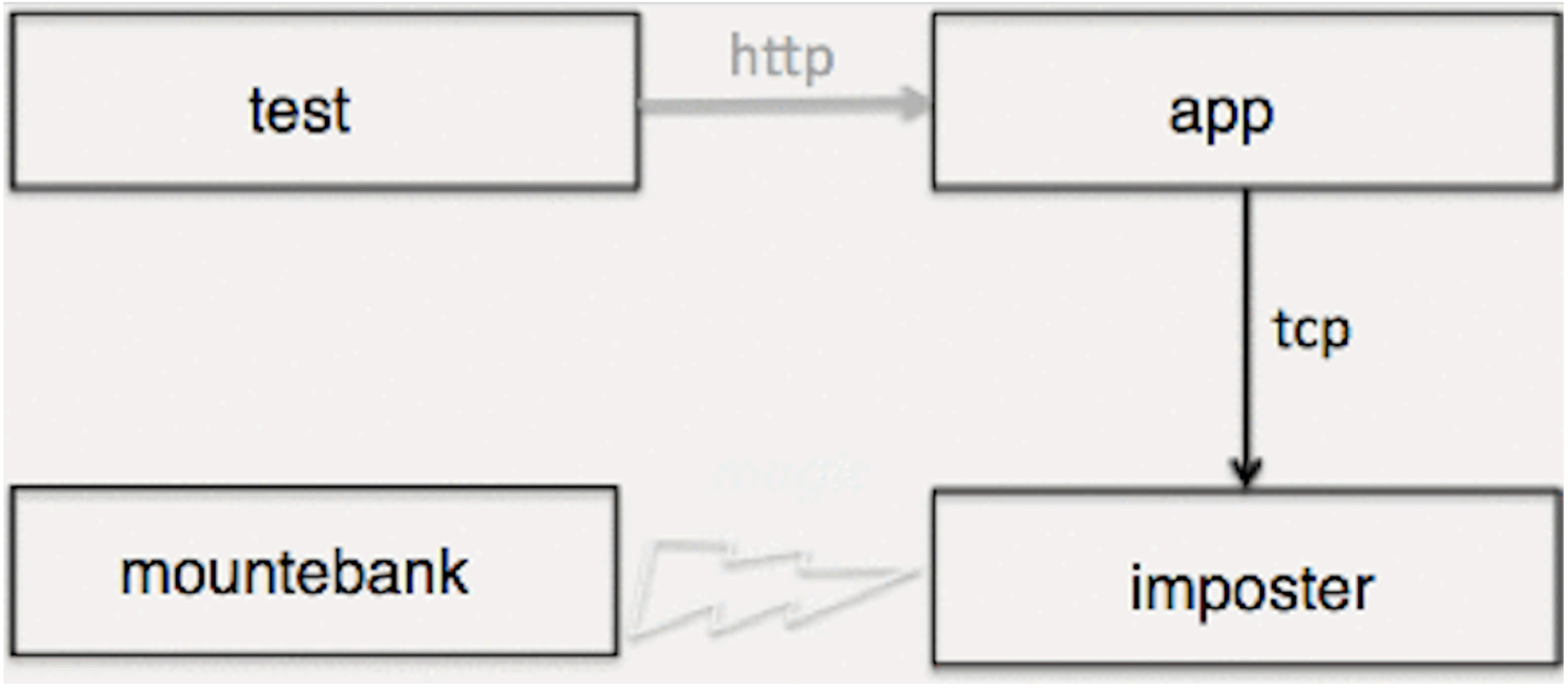
TEST SETUP



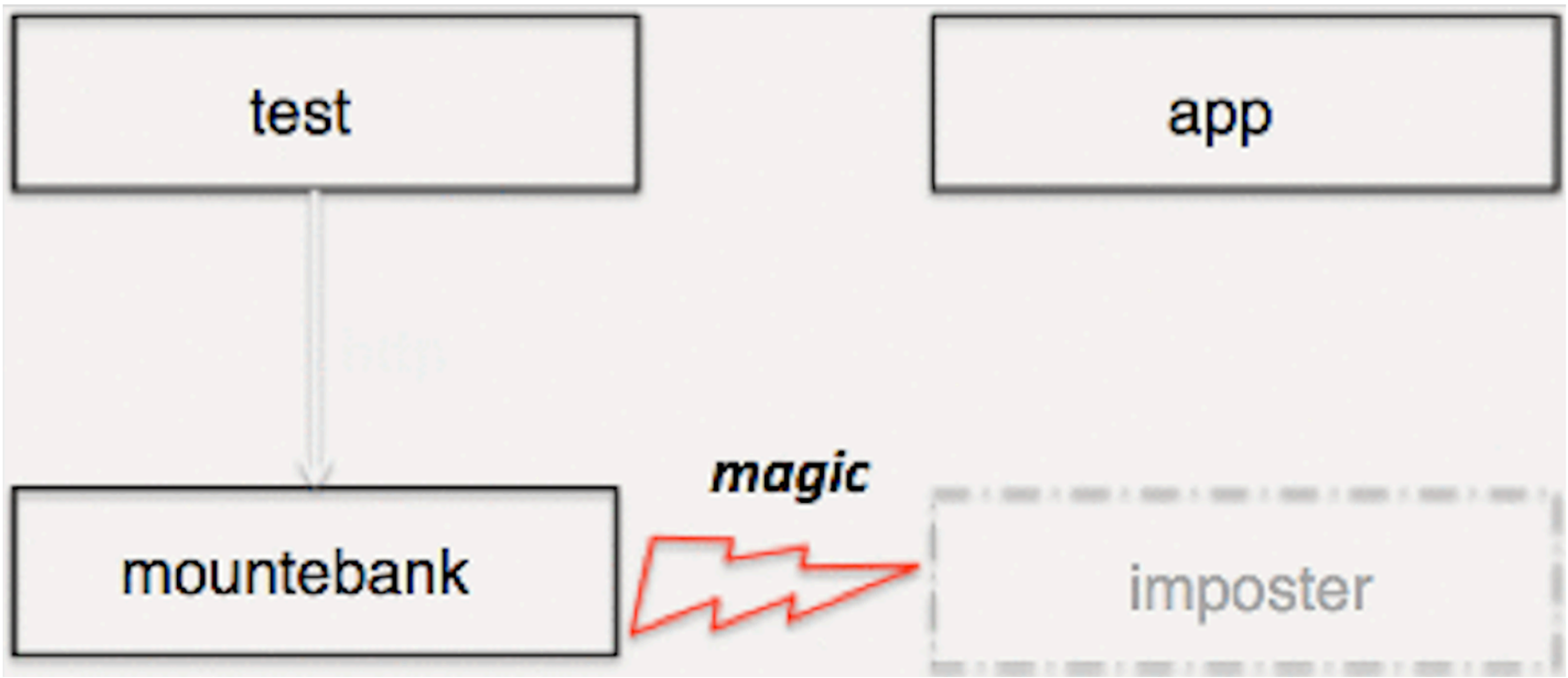
TEST RUN



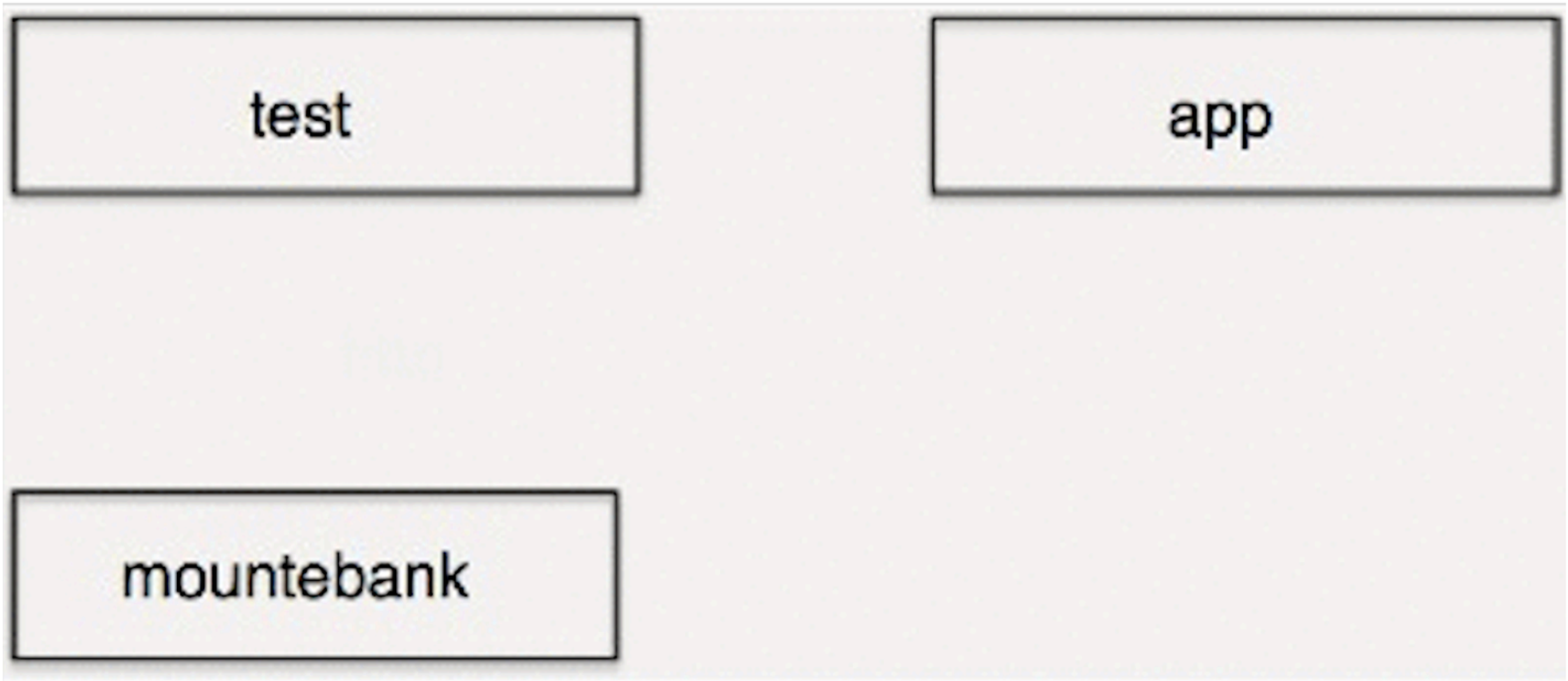
VIRTUAL DEPENDENCY



TEST CLEANUP

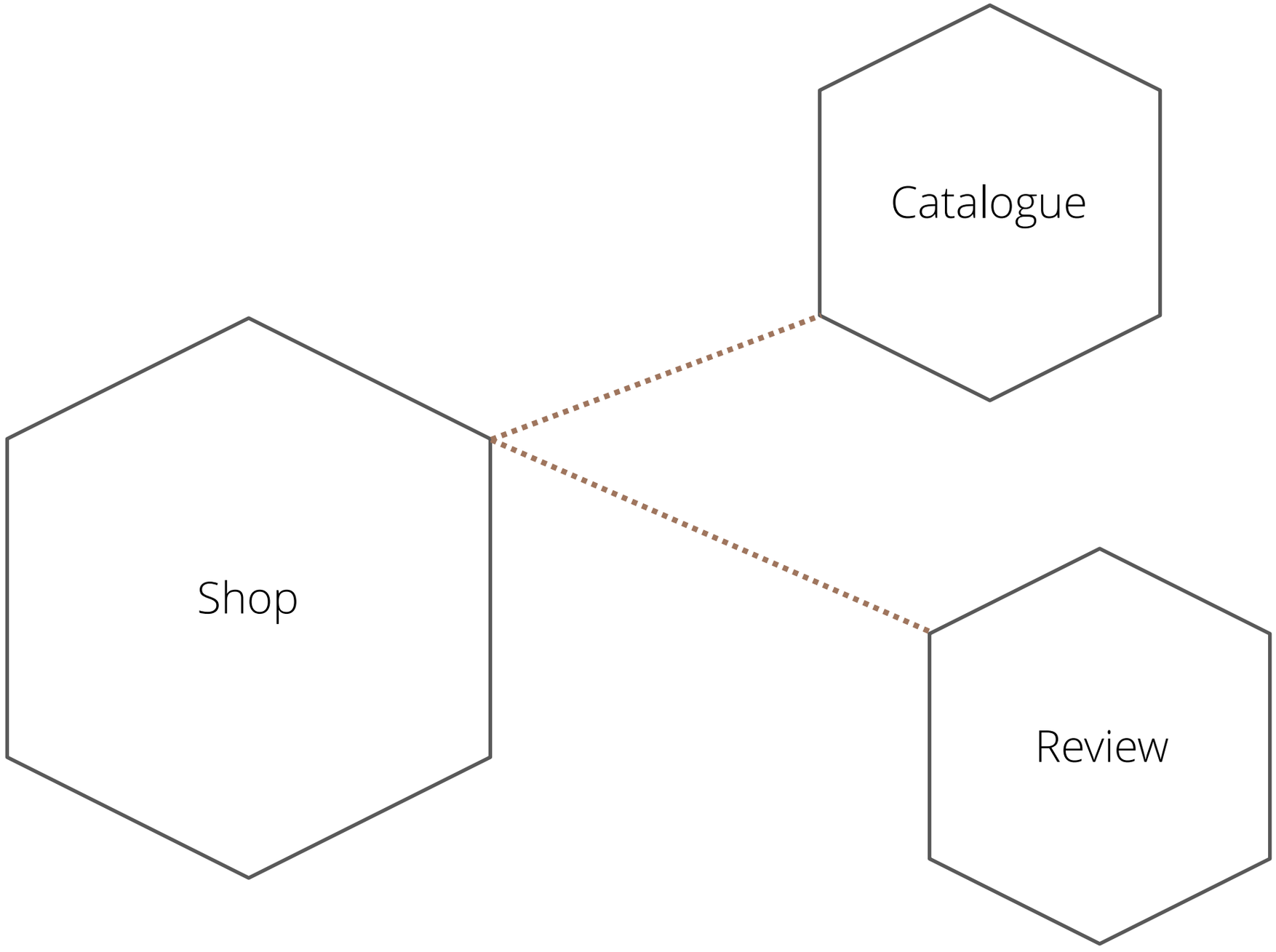


BACK TO CLEAN STATE



MOCKING CATALOGUE & REVIEW

OUR MINI AMAZON



SERVICE TESTING WITH MOCKING DEPENDENCIES

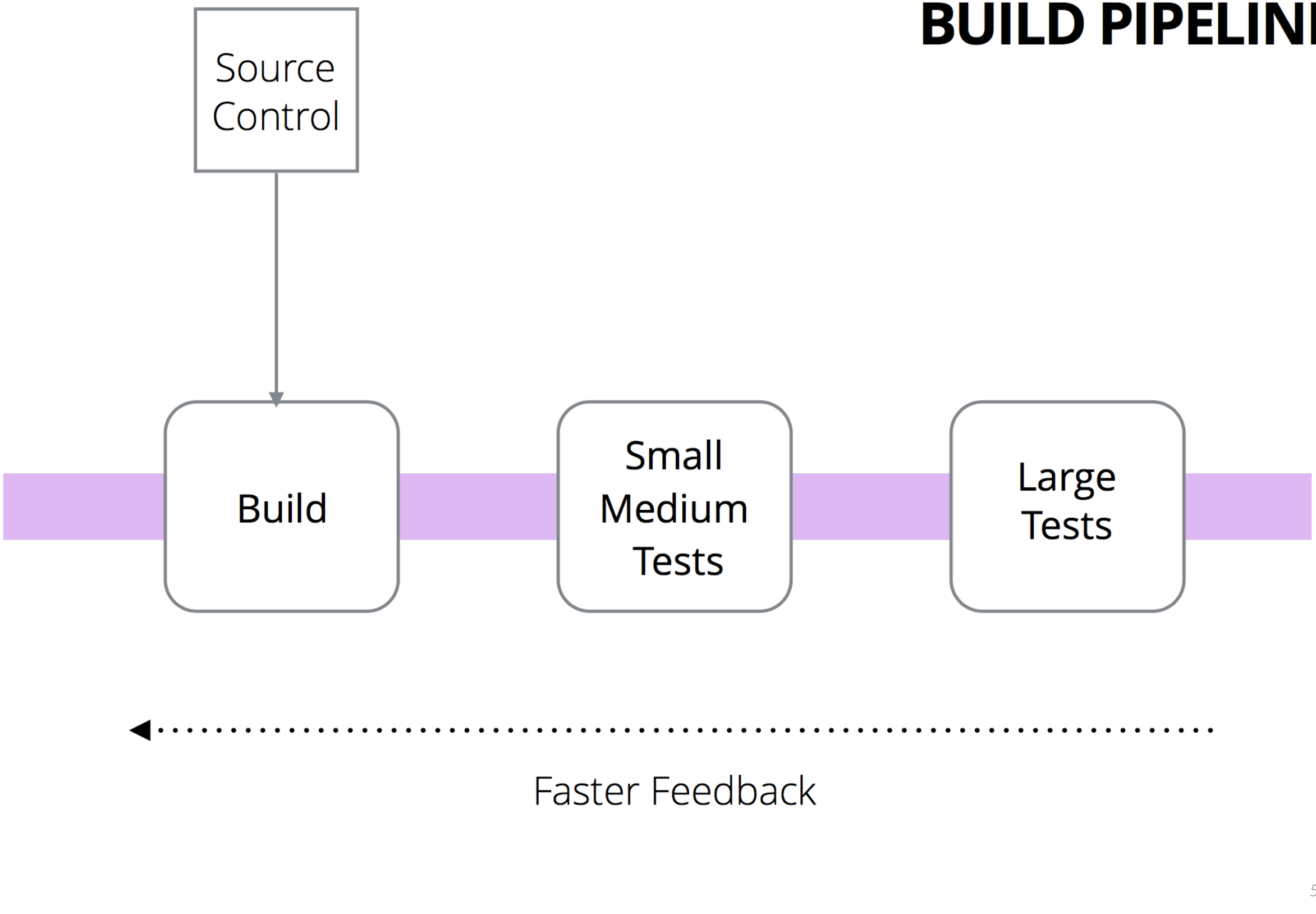
Go to service testing

```
vagrant@workshop:~$ cd ~/topics/start/02-service-testing
vagrant@workshop:~$ npm run integration
```

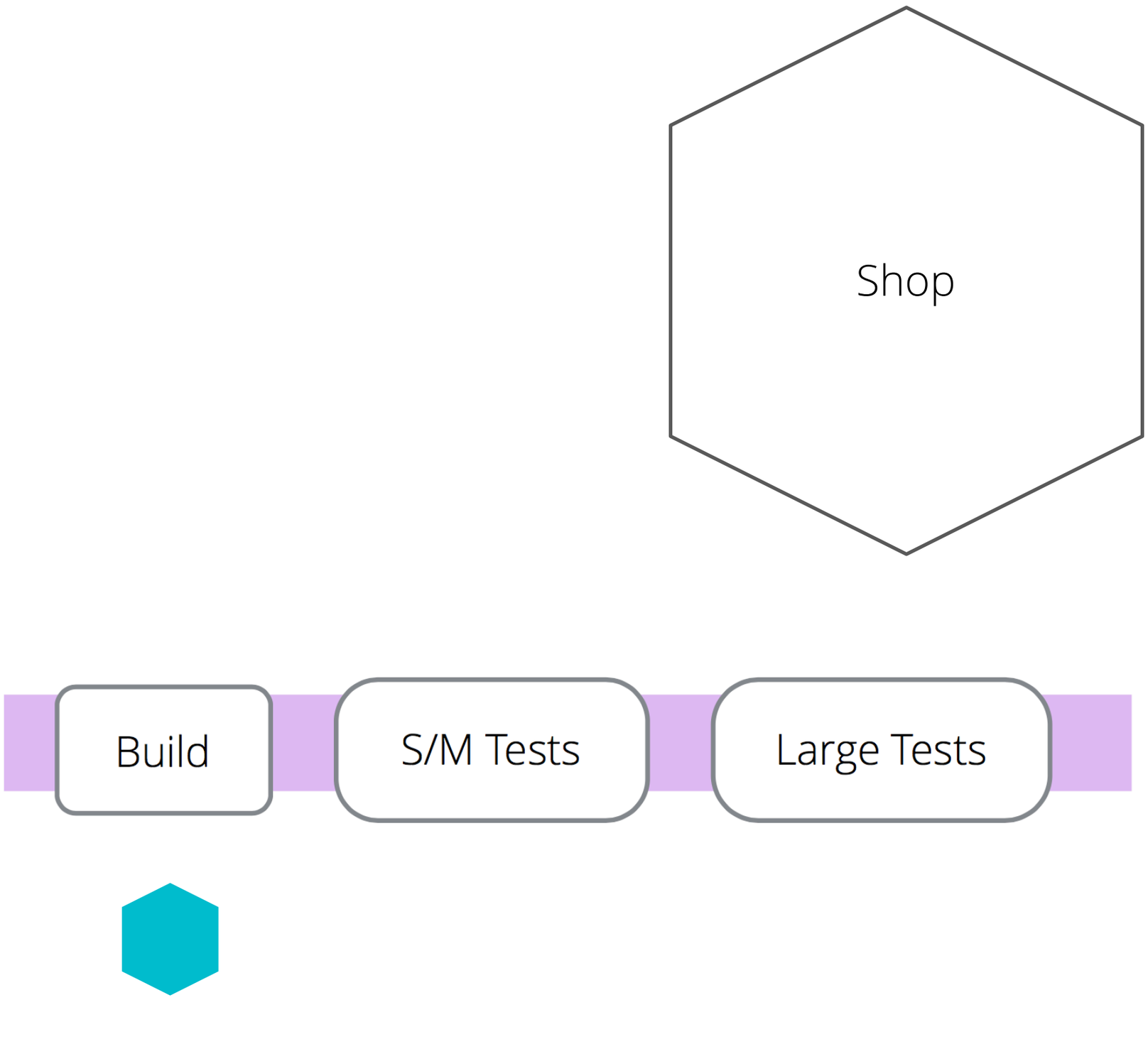
ThoughtWorks®

HOW NOT TO RELEASE MICROSERVICES

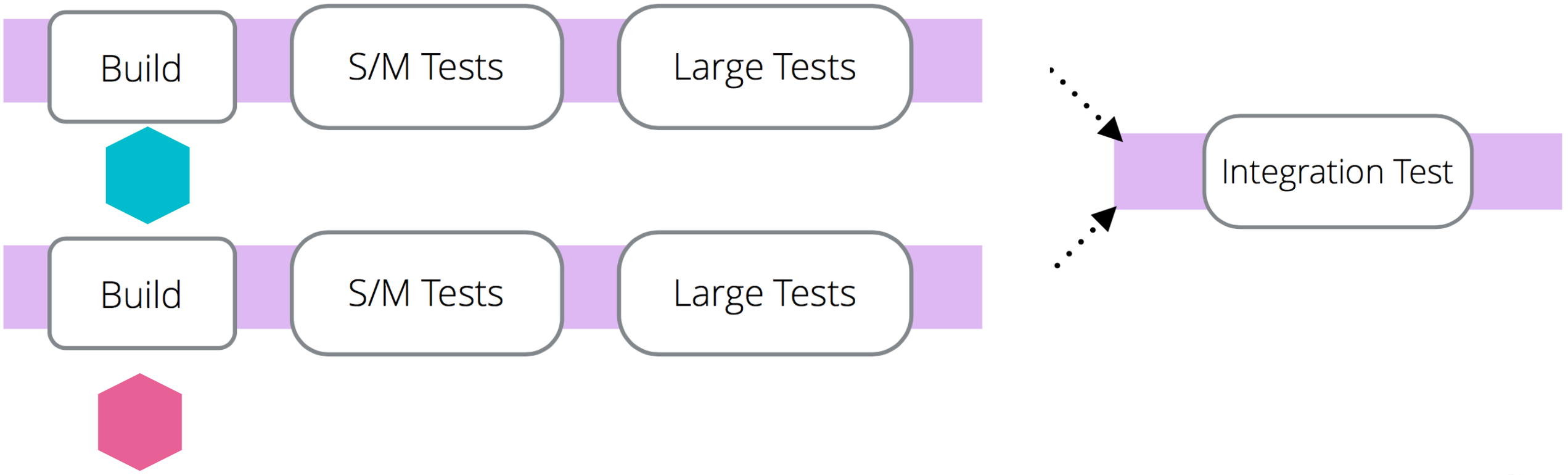
BUILD PIPELINE



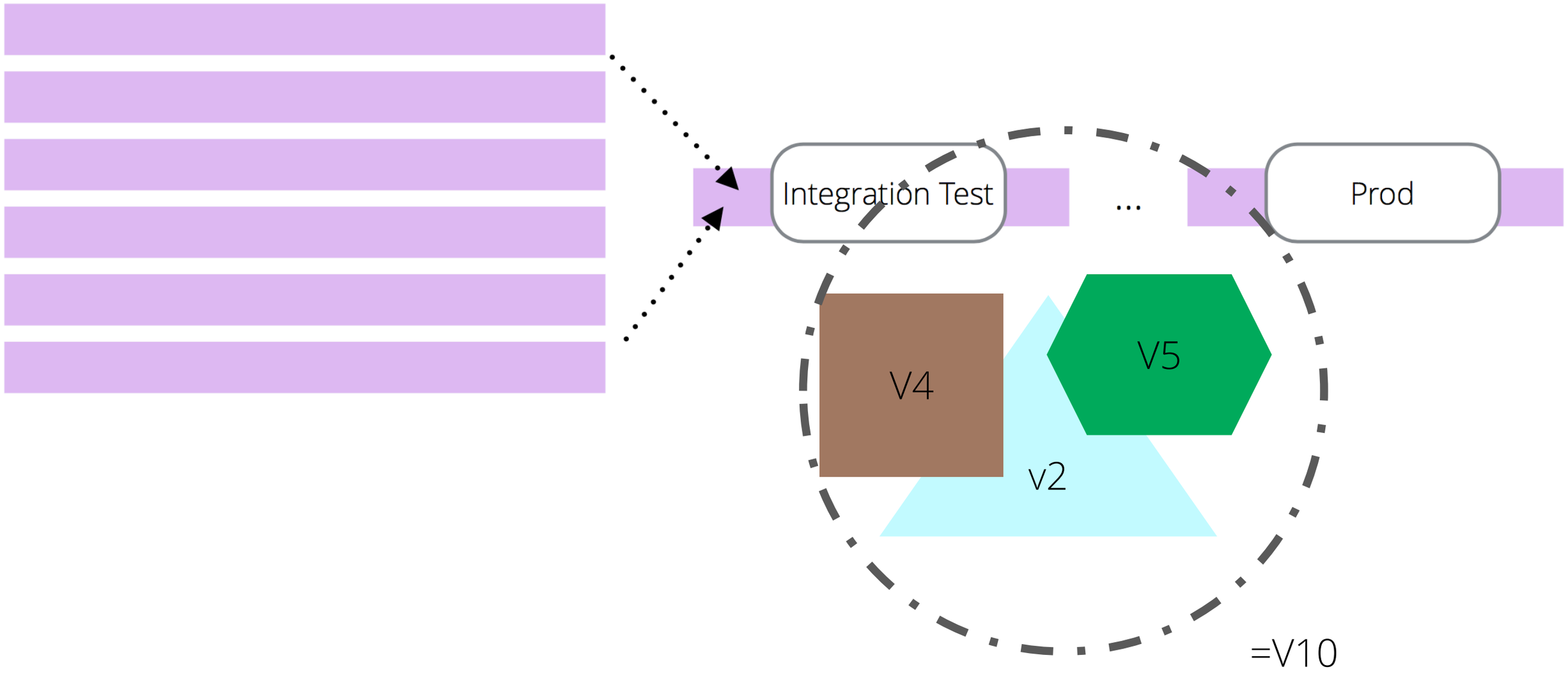
Pipeline View



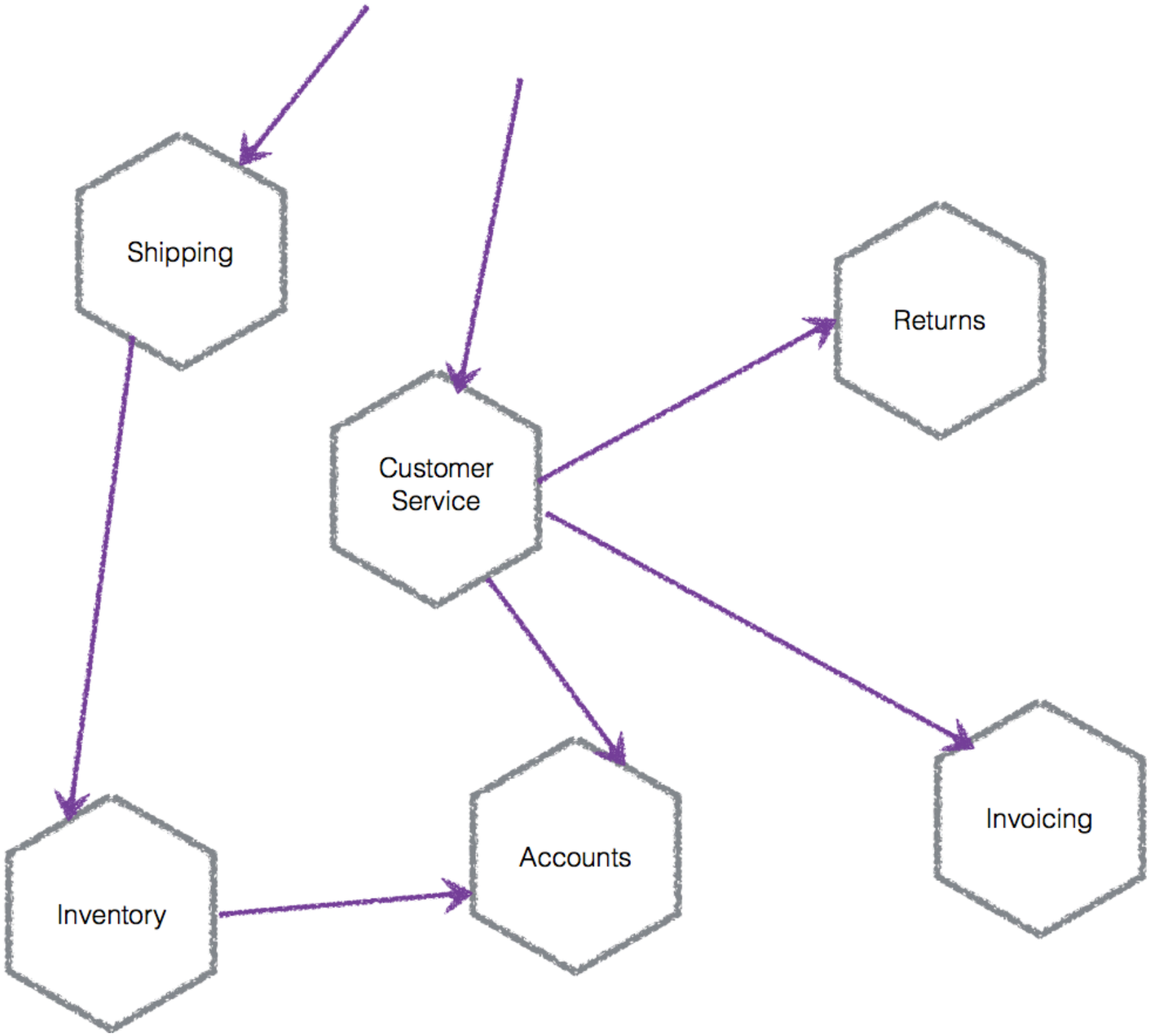
Pipeline View



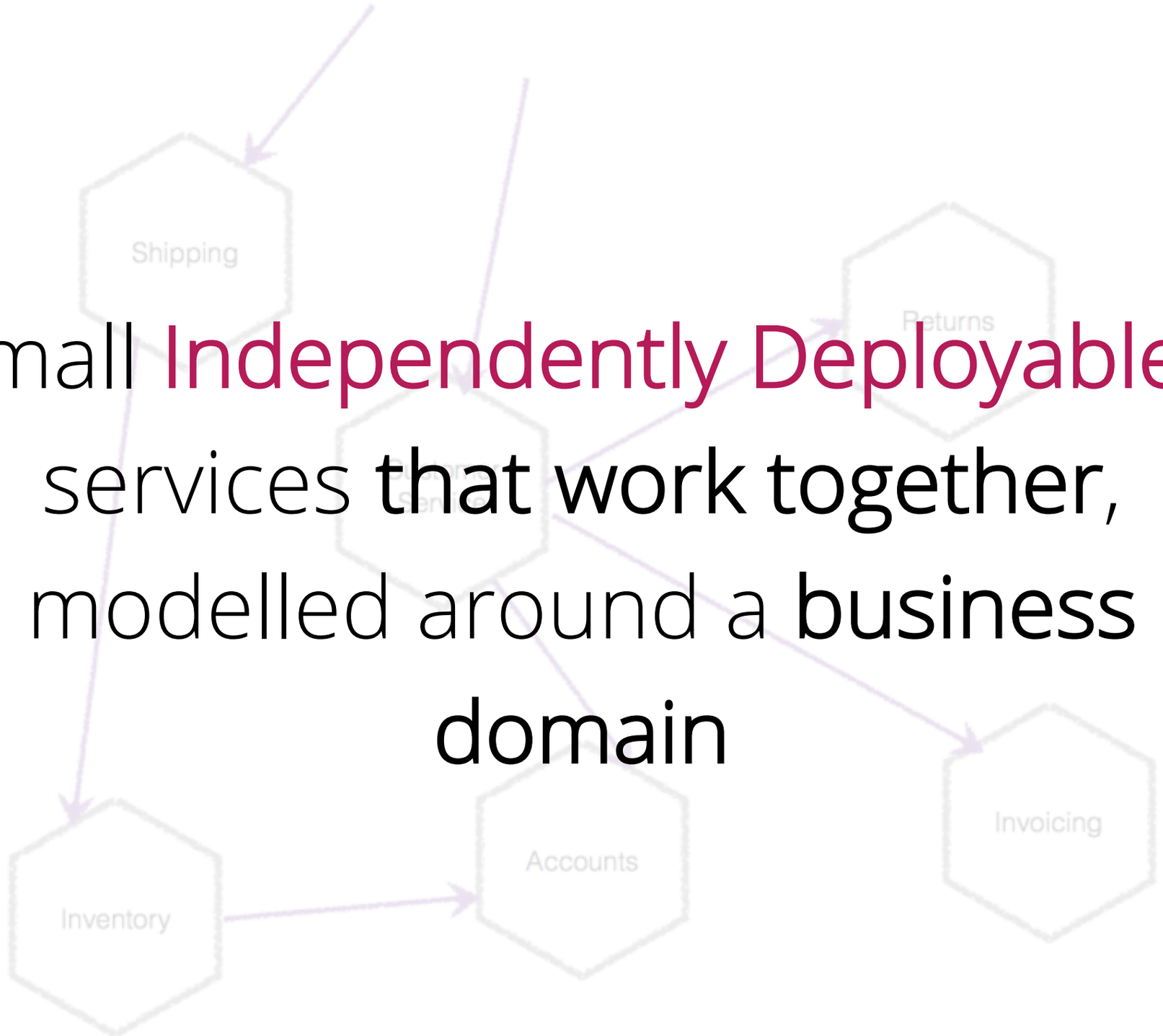
Integration Test as a gate



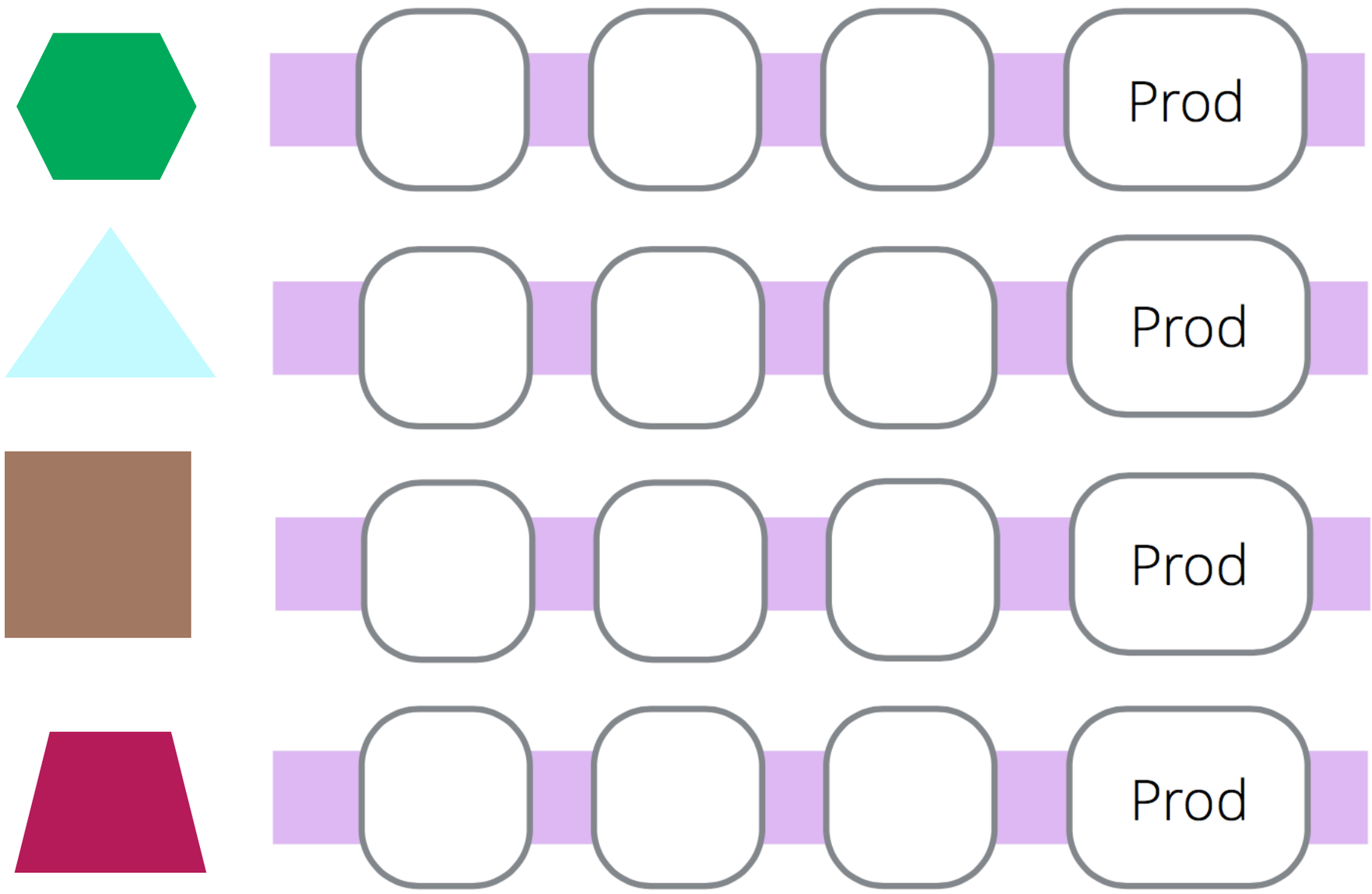
Environment Management



Microservices

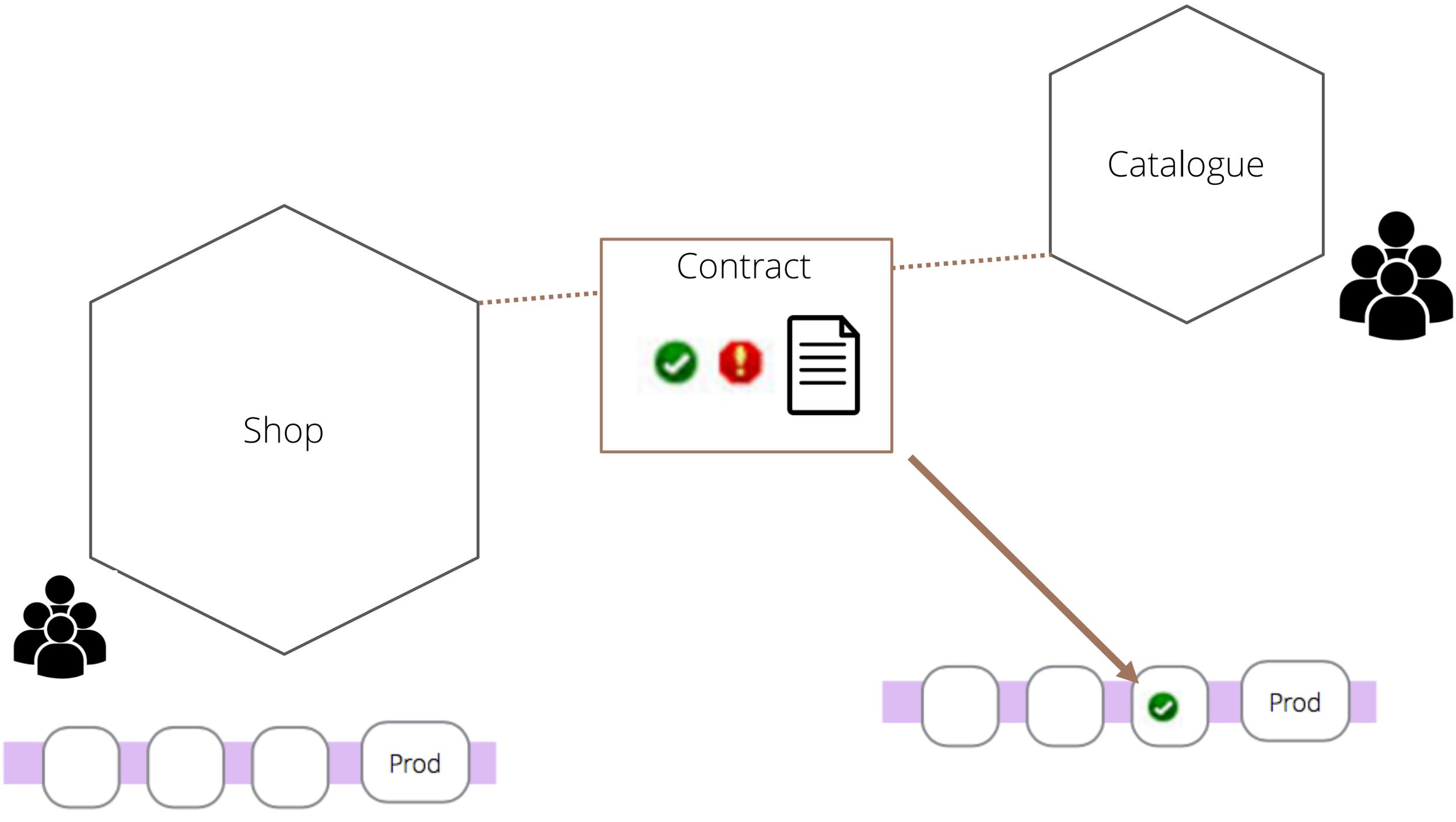


Small Independently Deployable services that work together, modelled around a business domain


Prod
Prod
Prod
Prod

CONSUMER DRIVEN CONTRACT TESTS

Consumer Driven Contract Tests



Secure https://docs.pact.io

Type to search

EDIT THIS PAGE

Pact

What is Pact?

The Pact family of frameworks provide support for Consumer Driven Contracts testing.

Consumer Driven Contracts

A `Contract` is a collection of agreements between a client (`Consumer`) and an API (`Provider`) that describes the interactions that can take place between them.

Consumer Driven Contracts is a pattern that drives the development of the `Provider` from its `Consumers` point of view.

Pact is a testing tool that guarantees those `Contracts` are satisfied.

Why do you want to use Pact

Confidence	Faster	Less Error Prone
Continuously evolve your codebase knowing that Pact will guarantee Contracts are met	No need to setup end-to-end environments. No need for manual testing	Generation and verification of Contracts are automatically managed by Pact

CONSUMER DRIVEN CONTRACT TESTS

```
vagrant@workshop:~$ cd ~/topics/start/03-contract-testing
vagrant@workshop:~$ npm test
```

SUMMARY

- ☐ Understand that there are different types of tests. Balance them to get feedback right
- ☐ Ensure your testing is focused around letting you release services independently
- ☐ Use consumer-driven contracts to replace end-to-end tests

THANK YOU

For questions or suggestions:
@sinajahan

ThoughtWorks®

References

- https://www.martinfowler.com/microservices/
- http://samnewman.io/books/building_microservices/
- https://martinfowler.com/articles/microservice-testing/
- http://www.mbtest.org/
- https://docs.pact.io/