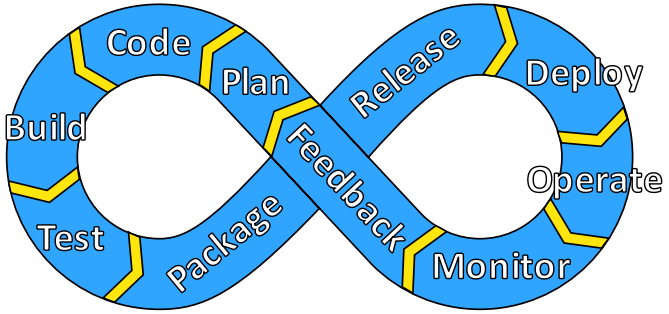
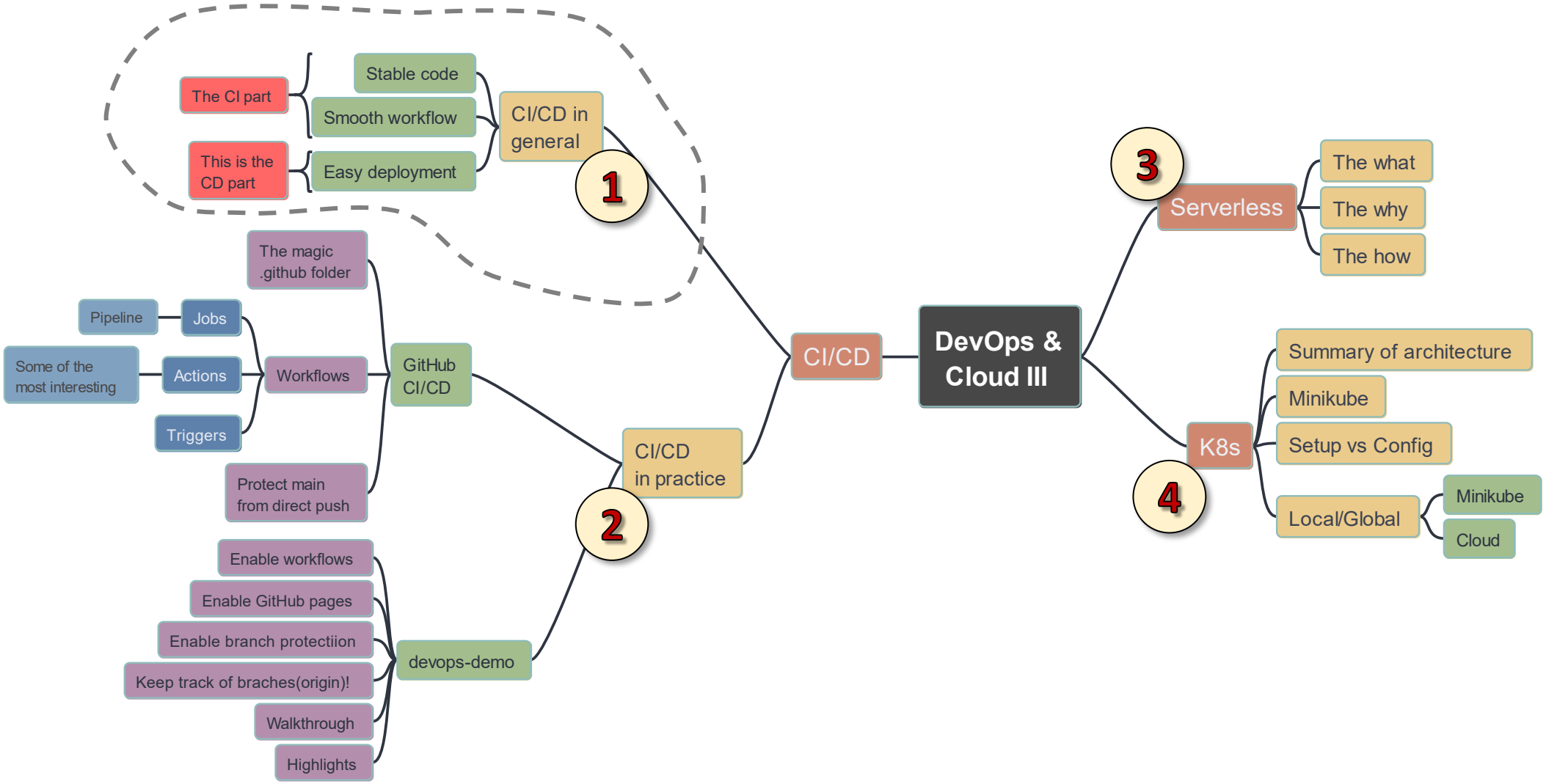


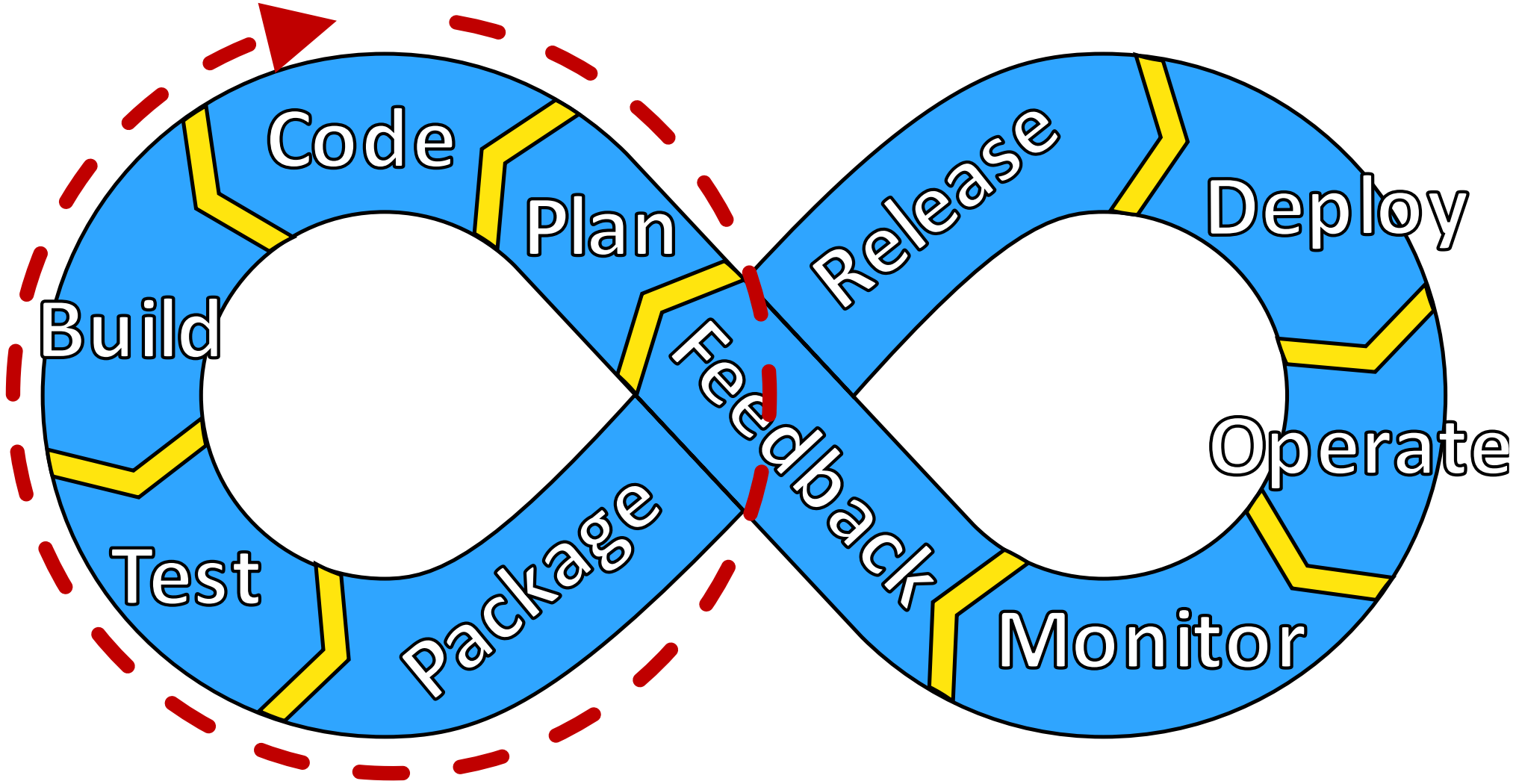
DevOps & Cloud

CI/CD, Serverless and K8s

Today's agenda



CI (Continuous Integration)

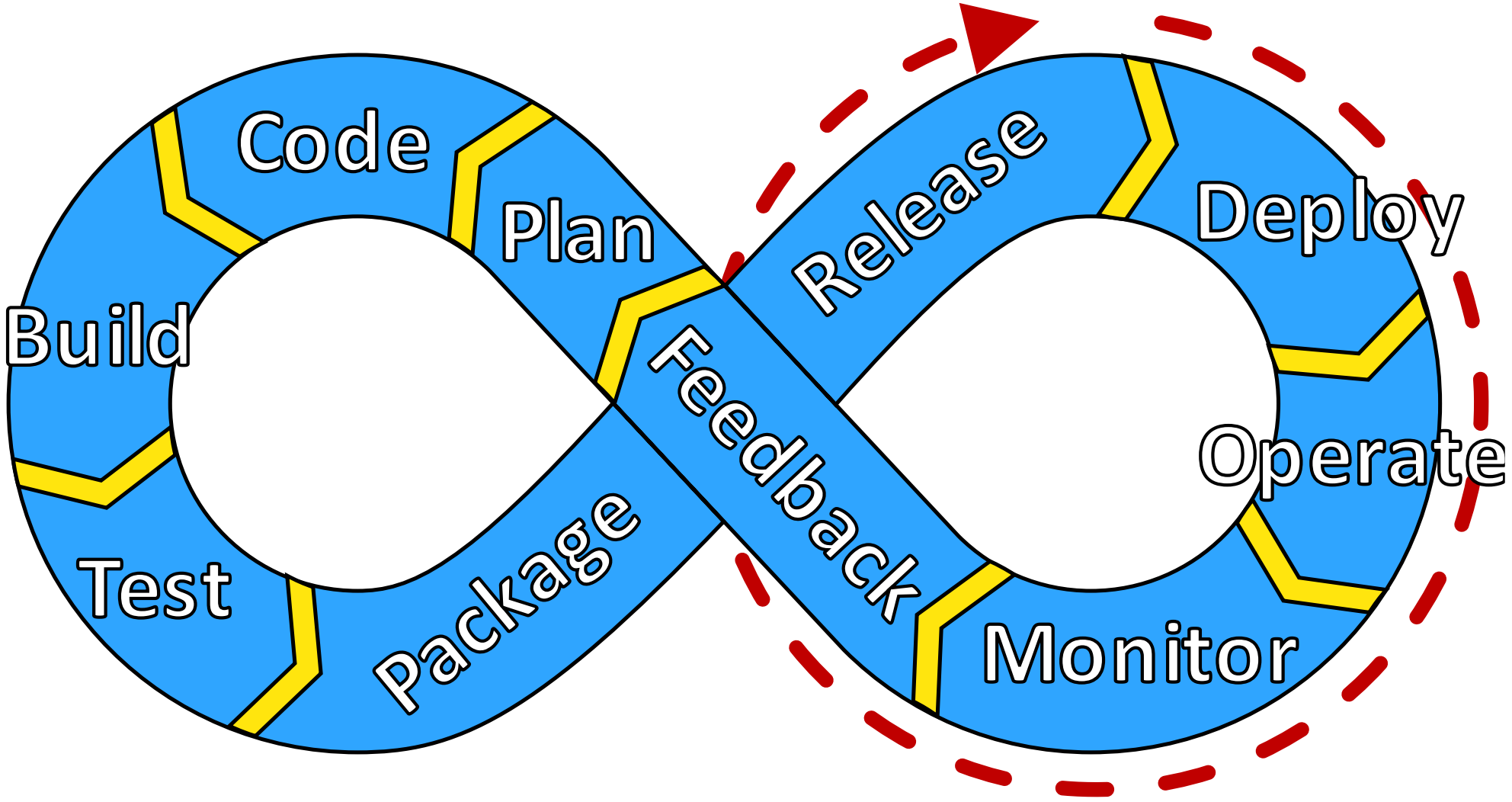


CI is a core DevOps discipline

These are some of the benefits we get from the CI part of DevOps

- Stable code
  - Always buildable
  - Always runnable
  - Easy creation, use and maintenance of tests
  - All tests are run automatically
- Smooth workflow
  - No “noise”/wasted hours and frustration from bad commits
  - Minimized friction from doing multiple parallel lines of development

CI (Continuous Integration)



CD is the other main core DevOps discipline

These are some of the benefits we get from the CD part of DevOps

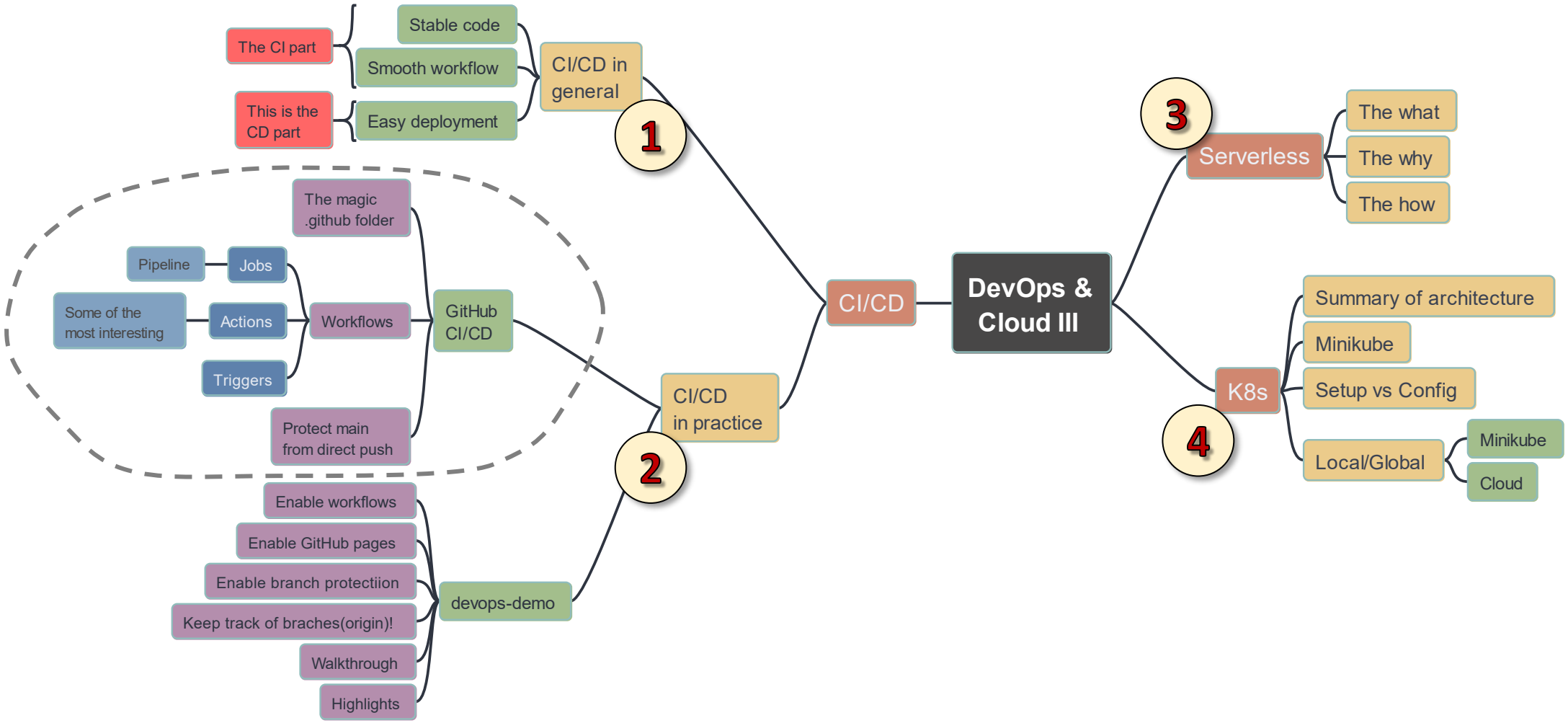
- Easy deployment
  - Deployment is closely linked with development
  - Processes needed for deployment (packaging and releasing) are automated – perhaps even the deployment itself

- Traceability
  - All changes to code are linked to well-defined “causes/origins” – either requirements or reported issues (“jira’s”)
  - Releases carry info on all changes in the code
  - Deployment is done with well-defined release versions (configurations)
  - We know at all times what release is running where and what was last changed

CD = Continuous [Delivery | Deployment]

- **Continuous delivery** is an extension of CI since it automatically deploys all (successful) code changes to a testing and/or production environment after the build stage.
  This means that on top of automated testing, you have an automated release process, and you can deploy your application any time by clicking a button.
- **Continuous deployment** goes one step further than *continuous delivery*. With this practice, every change that passes all stages of your *CI + delivery* pipeline is automatically released to your customers. There's no human intervention, and only a failed test will prevent a new change to be deployed to production.

Today's agenda



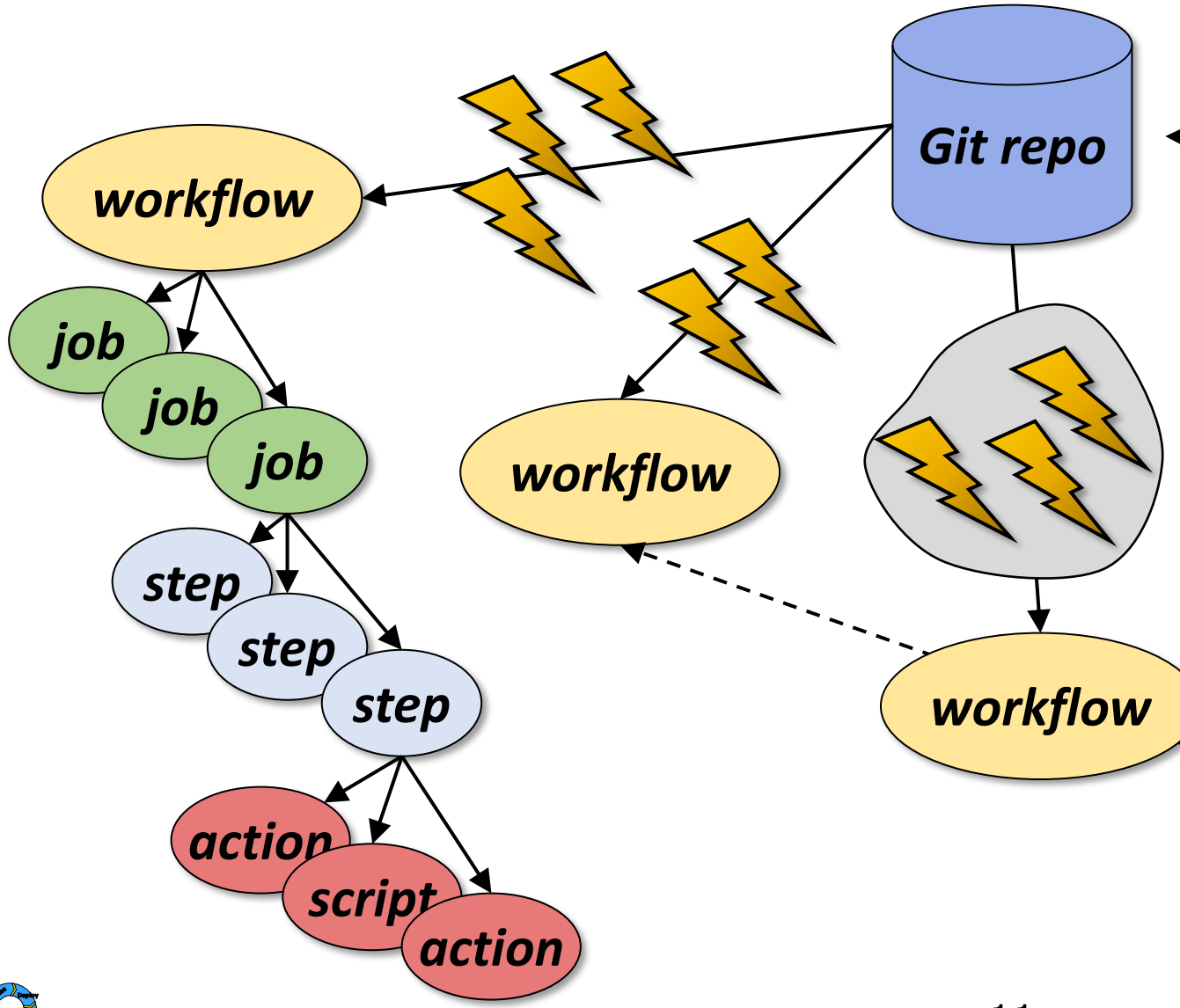
CI/CD tools in practice

- All contemporary Git/DevOps providers offer CI/CD tools
- GitHub calls them “GitHub Actions”, GitLab just talks about CI/CD tools and pipelines
- The purpose/goal is the same in all cases: we want to be able to execute scripts/code whenever some well-defined event is observed in the code base placed in Git.
- Think of GitHub, GitLab etc. as “Git with benefits” – you do basic git stuff, and then have “actions” executed automatically whenever something happens:
  - Code is push’ed into some branch, changes are pull’ed from one branch to another etc. etc.

GitHub Actions (1)

- GitHub Actions are part of *workflows* which are described in YAML-files
- Simply by placing YAM-files into a “magic” folder placed in your repository we make GitHub sing and dance – that folder is called `.github/workflow`
- The name(s) of the file(s) is/are not important – GitHub will scan them and interpret the contents no matter what you call them

GitHub Actions are organized in *workflows*

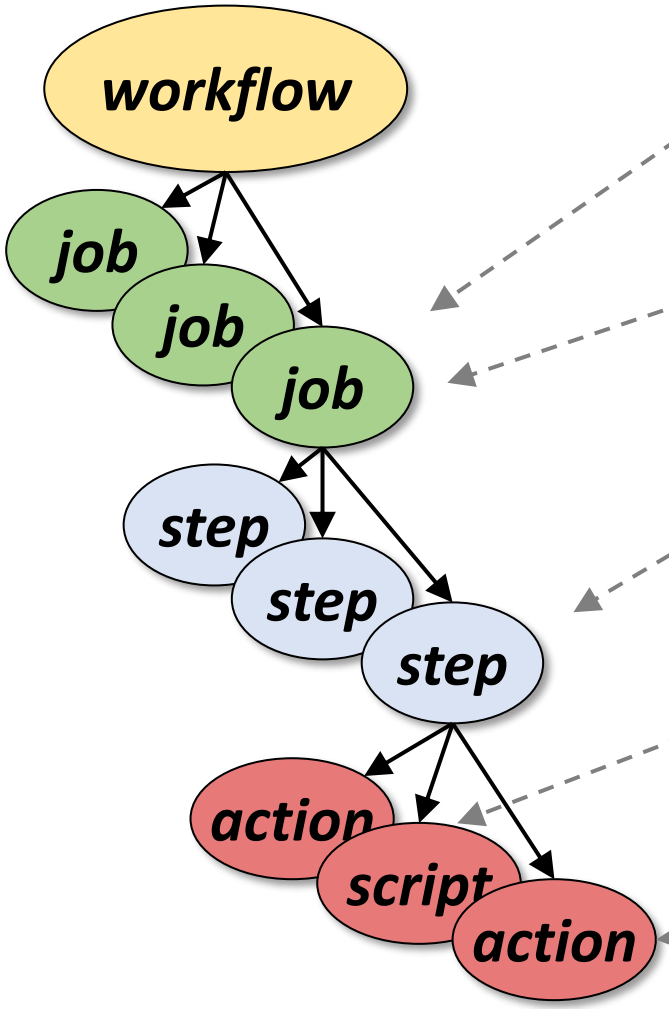


Git repo with a .github/workflows folder

Each workflow is triggered by a set of well-defined events

Multiple workflows may be triggered and run in parallel

The structure of workflows



Each job runs inside its own runner (a virtual machine)

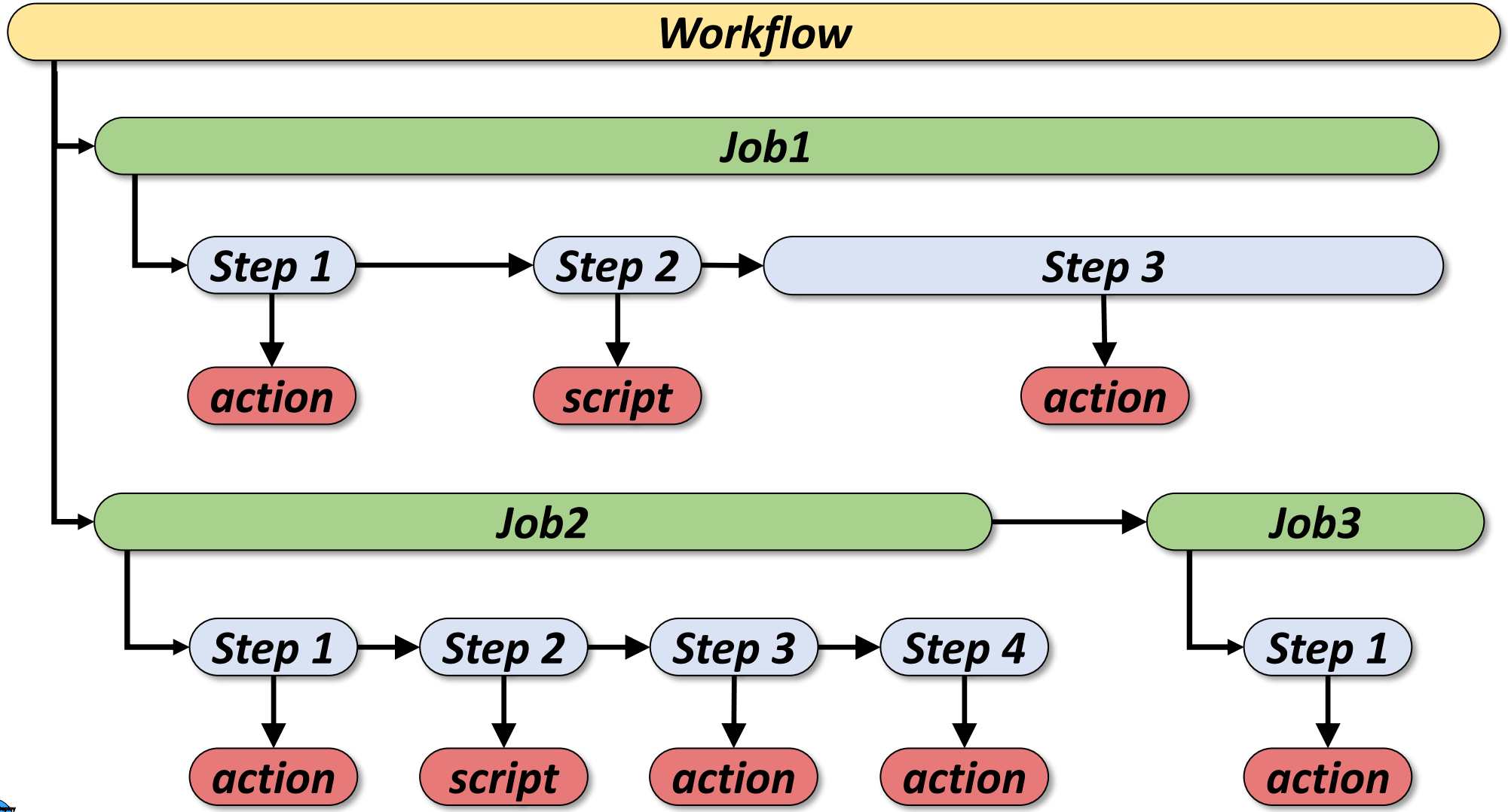
Jobs can run in parallel, or depend on each other which forces a sequence

Each step runs a script (that you define/write) or runs an action

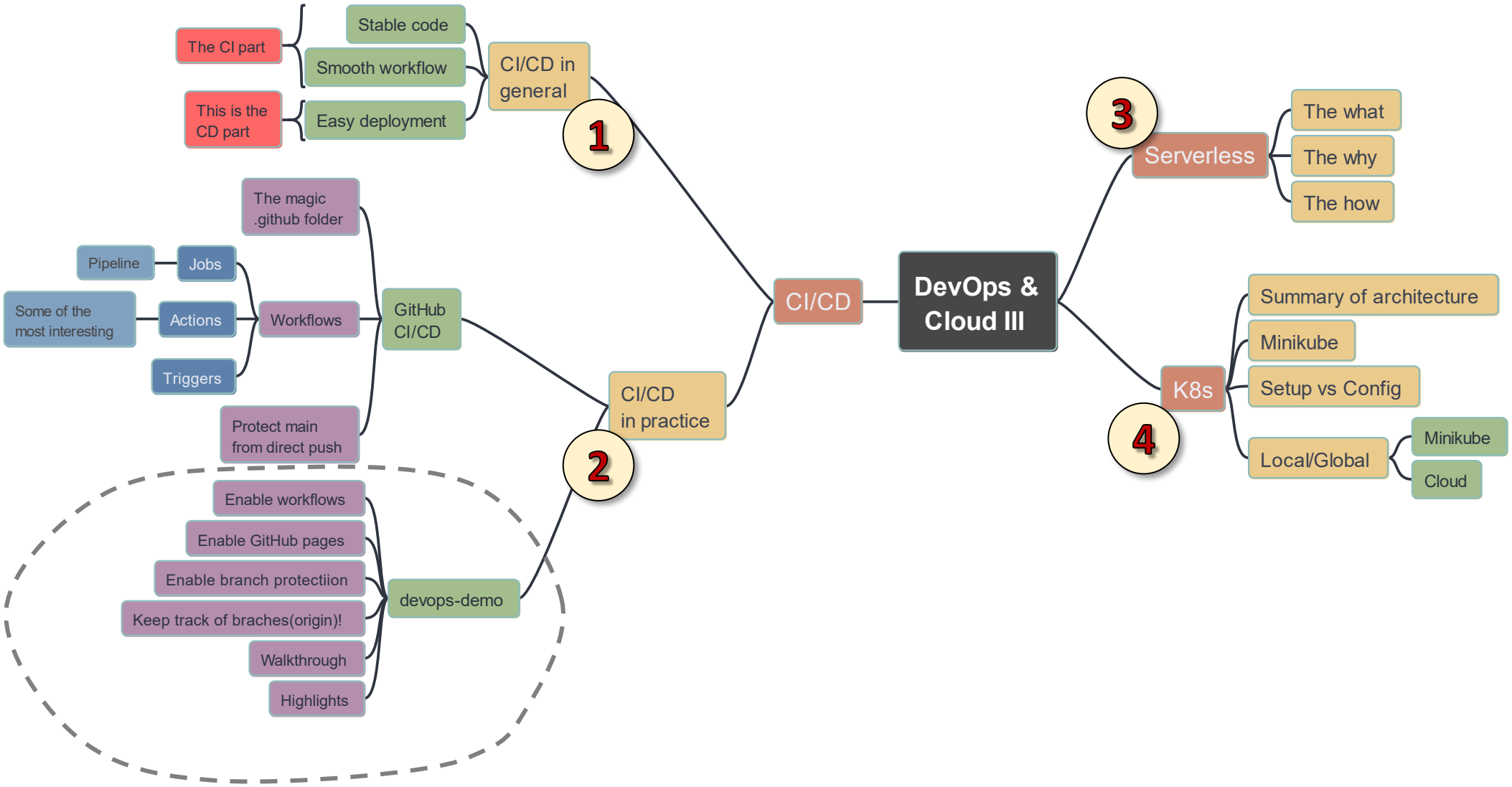
A script is a shell script, e.g. bash script or PowerShell script

You can define your own actions or use one of the many pre-defined ones available

The structure of a single workflow



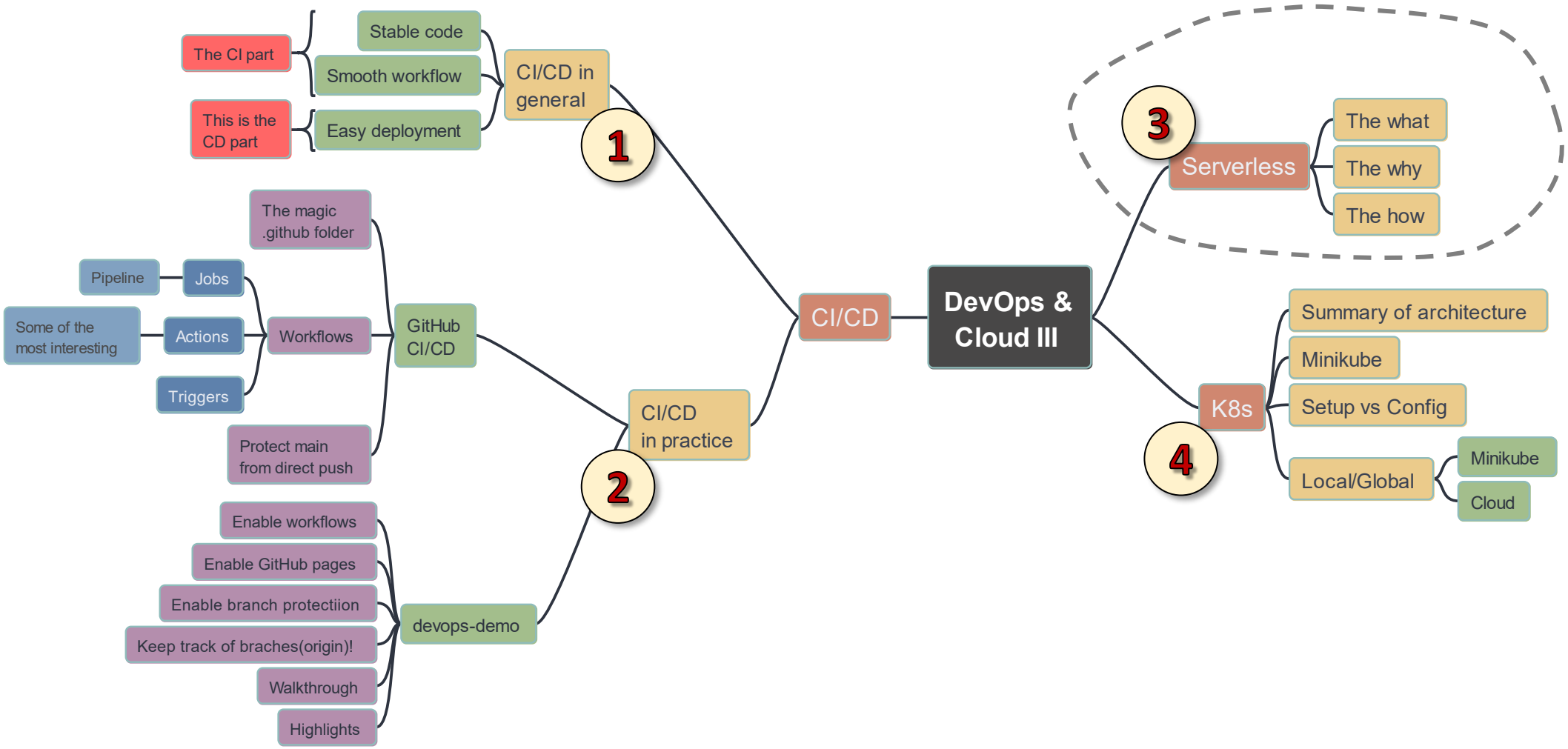
Today's agenda



GitHub CI/CD demo

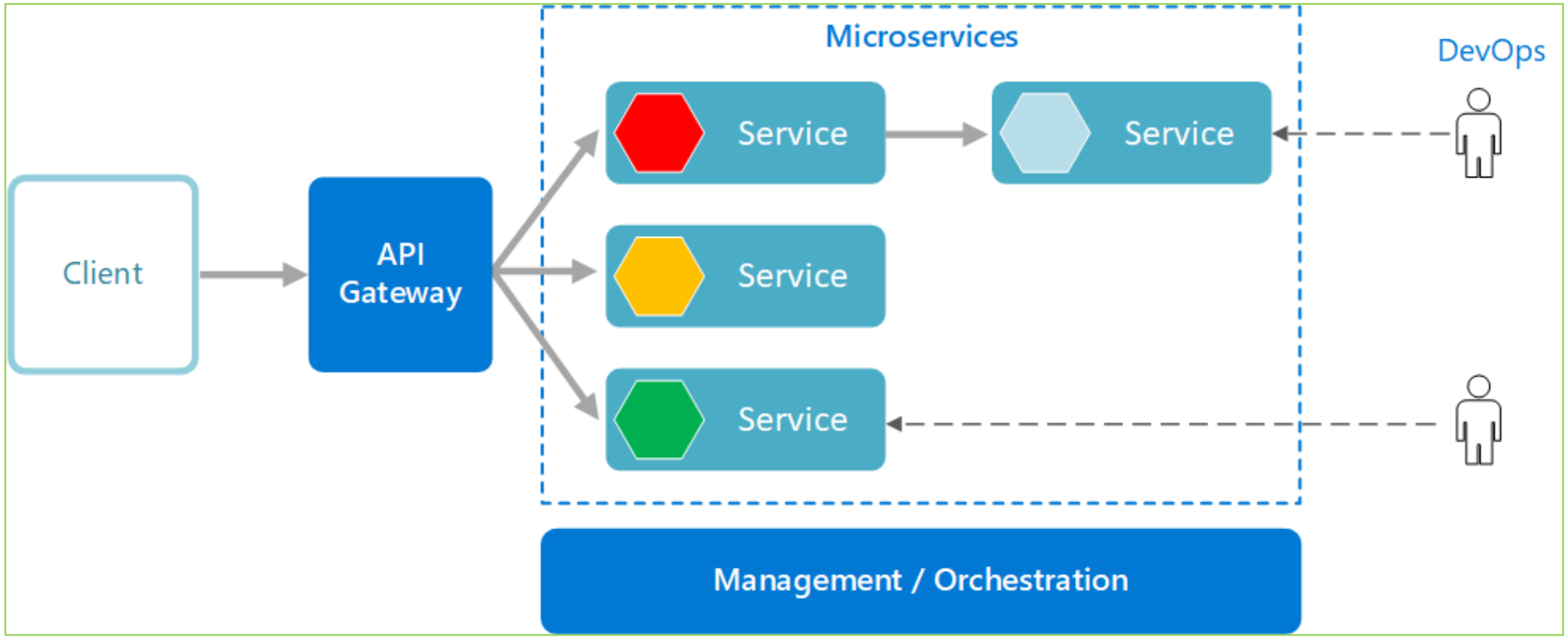
- We'll work with this repo: https://github.com/DaFessor/devops-ci-demo
- Let's have a closer look at it .....
  1. Repository settings
     1. Force pull requests (branch protection)
     2. Enable workflows/actions
     3. Enable pages
  2. Github Actions (what do they do? Using GITHUB_TOKEN)
  3. Interesting files (Dockerfile, compose.yaml)

Today's agenda



Serverless – what is that?

- First of all, serverless does not mean “without using servers” – there are servers involved
- Let’s start with refreshing micro services:



Key points of micro services

1. Microservices are small, independent, and loosely coupled
2. Each service is a separate codebase, which can be managed by a small development team.
3. Services can be deployed independently
4. Services are responsible for persisting their own data or external state.
5. Services communicate with each other by using well-defined APIs.
6. Supports polyglot programming. For example, services don't need to share the same technology stack, libraries, or frameworks.
7. Micro services runs in containers and act as small servers that are “active” all the time = listening on a port waiting for requests

Micro services are nice and work well, but …

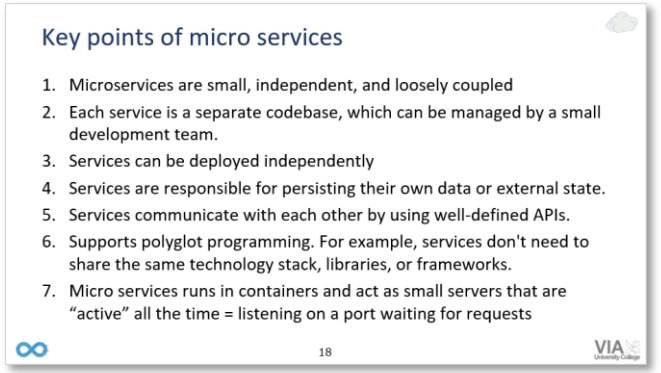
- But we still have to pay for up-and-running-time as longs as our containers are active
- But some services may be used less often, and we could save a lot of money in cloud costs if we were able to simply spin up a container on-demand when we need a service and stop it when a request has been processed
- And such a feature, “start container on-demand, do the necessary processing and stop the container”, are offered by all the cloud providers
- That feature is called ….. wait for it …. ***serverless!***

So back to serverless

- So serverless (or “cloud functions”, “lambdas” etc) works like microservices in many aspects:
  - They run inside a container
  - They are activated/triggered by different kinds of events (http, etc.)
  - They are small and independent functional blocks implementing a single feature
  - We need to design our application around them (micro services, functions, lambdas) from the get go to benefit from them
- But they differ in one key aspect: they *do not consume any resources* unless they have work to do (i.e. something triggered them to rum)
- Hence we only pay for the time where our functions/lambdas are active and doing processing

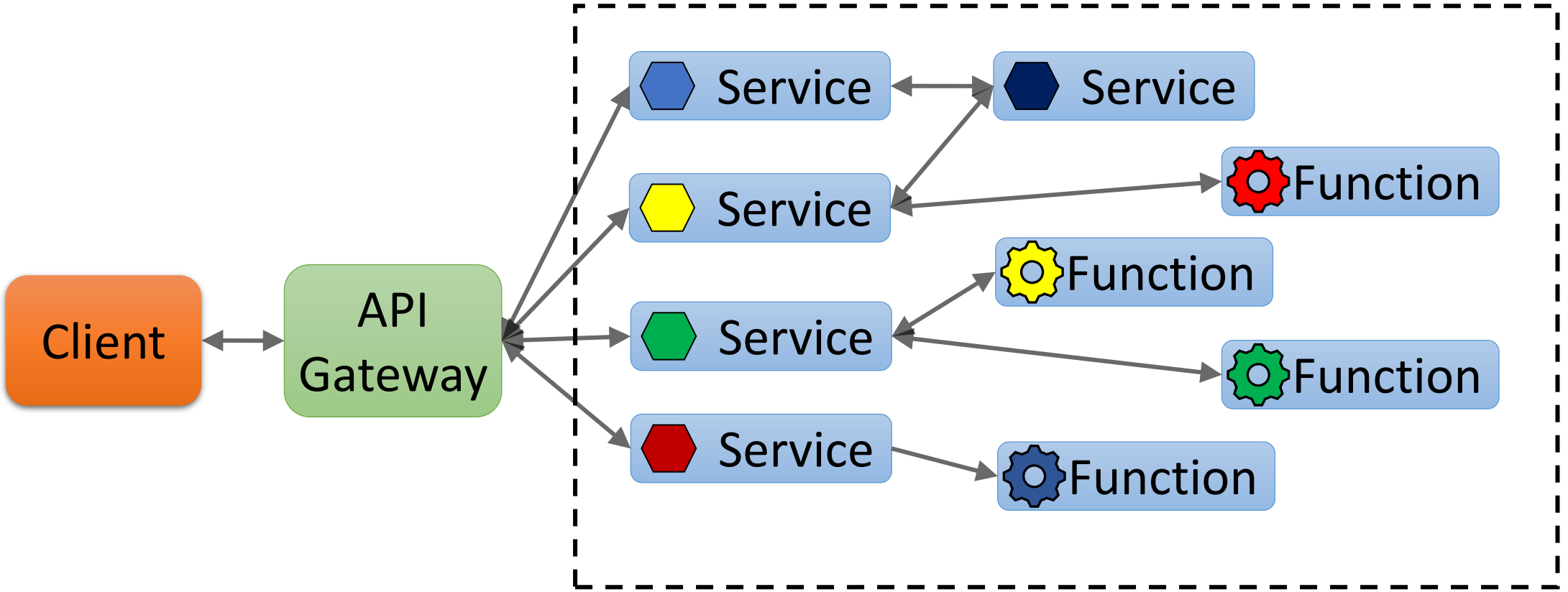
There’s no such thing as a free lunch

- But of course there’s a cost ... and it is measured in latency
- Even though it’s fairly fast to spin up a container (as we haven seen), it may take a little while (from ms to secs) to spin up a serverless service/function if it has not been used in a while
- But apart from that we keep all the benefits from micro services (apart from the being active all the time bit)

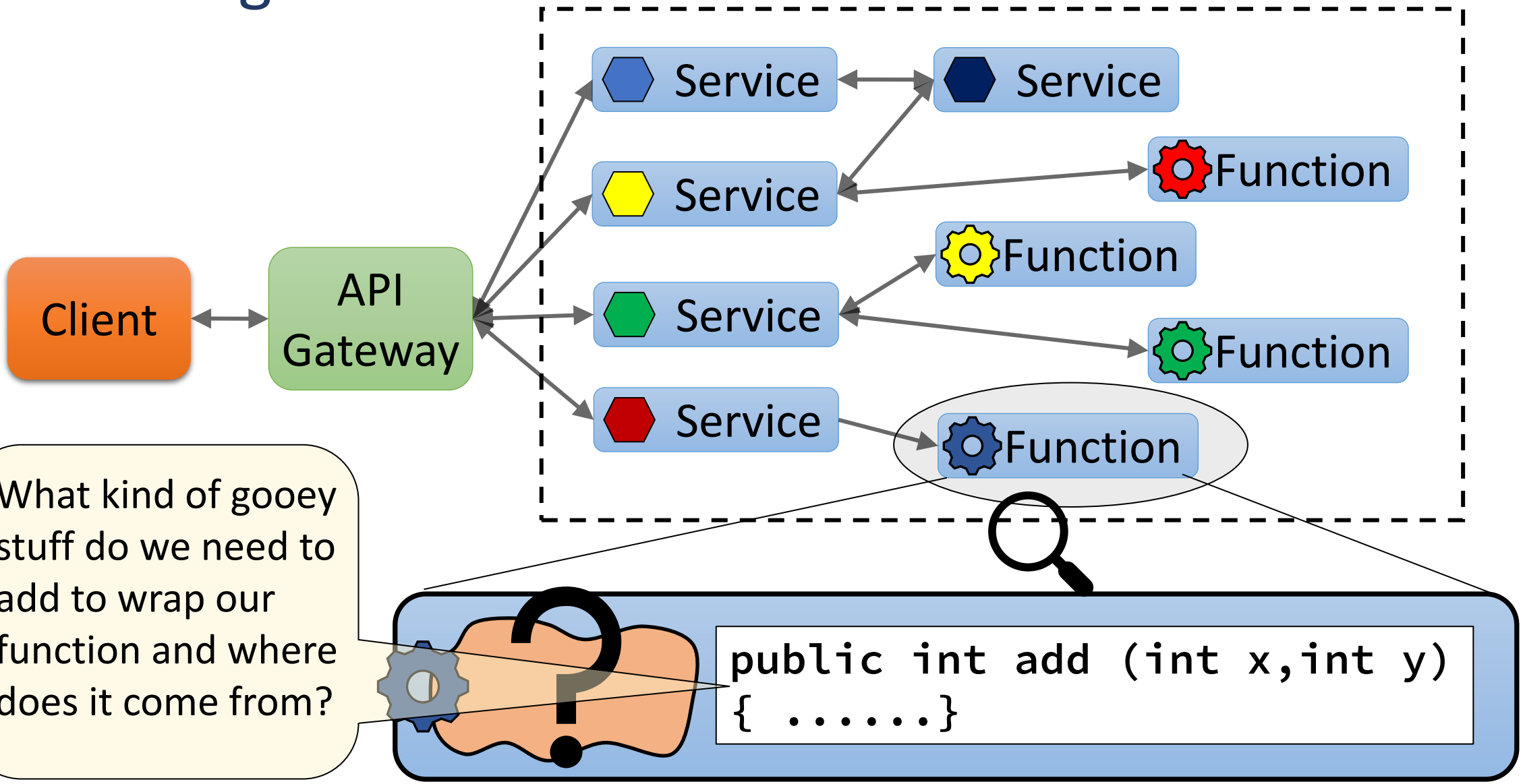


Wrapping a function and putting it in the cloud

- Serverless fits well into the model from before – we just modify more seldomly used services into functions



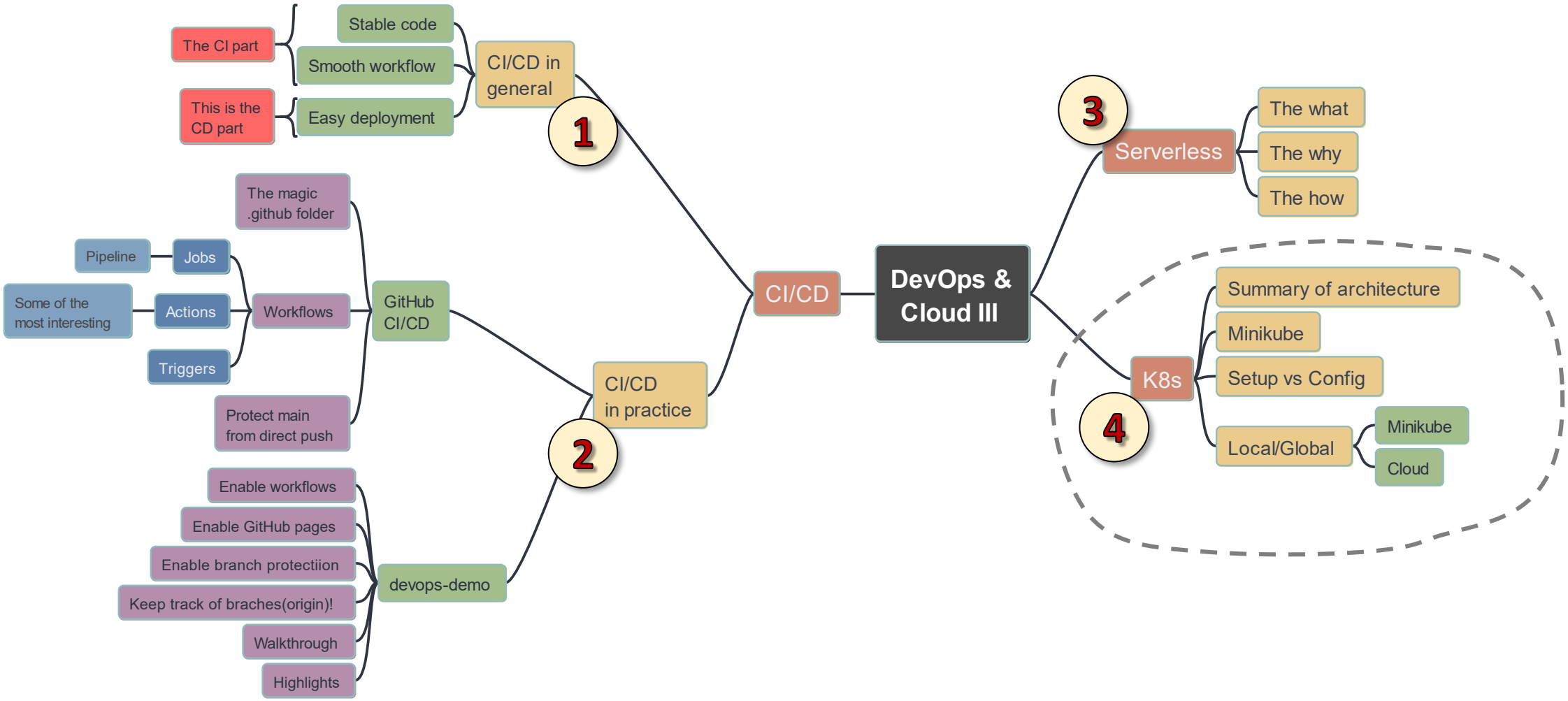
Zooming in a bit



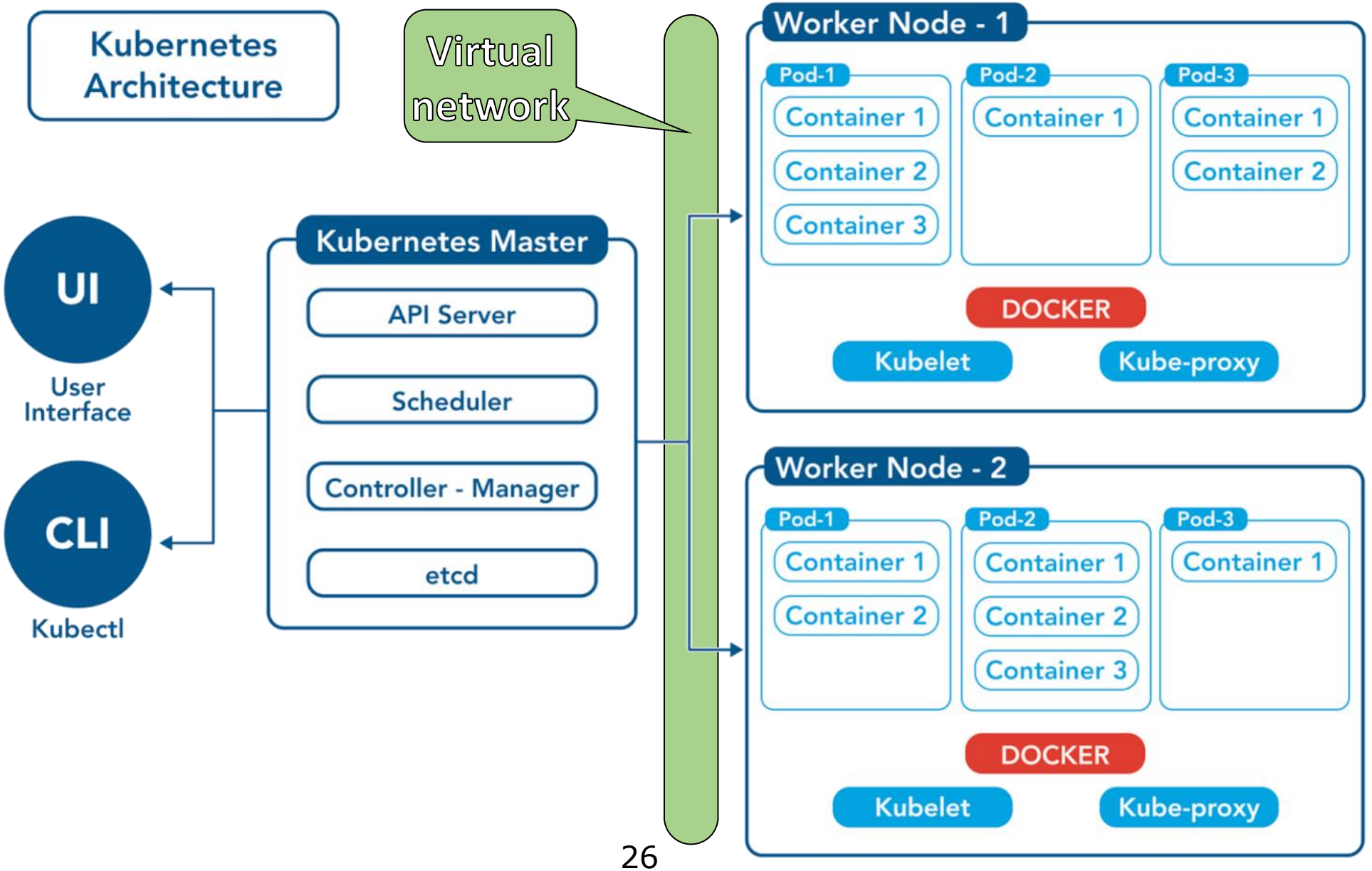
Exercise 1 – 45 mins incl. a break

- Do exercise 3.1 .....

Today's agenda



K8s architecture



Minikube

- Minikube is a 1-node “cluster” in which the both the control plane (the master node software) and the worker node software all runs on a single node
- That’s not a good way to deploy stuff in the real world, but it has all the same features of, and behaves like, a real cluster
- So, it’s more than toy – it allows us to test and debug a real application deployment on our own PC before we move the deployment into the real world
- All config files etc. will be same in our Minikube as in a 50-node cloud cluster somewhere

Exercise 2 – 30 minutes

- Let’s fire up our Minikubes....

Manual setup vs. config files

- We can manually create a desired setup/config using kubectl commands
- Once we have created a configuration we want to persist and reuse, we can use kubectl to ask the cluster for it's state/config like this:

```
kubectl get deployment devops-ci-demo -o yaml
kubectl get service devops-ci-demo -o yaml
```

- We can then use those yaml-files to restore a cluster by using the command:

```
kubectl apply -f k8s/myapp-deployment.yaml
kubectl apply -f k8s/myapp-service.yaml
```

- Or, if we have both files (deployment and service) in a single folder:

```
kubectl apply -f k8s/
```

Using *Kompose* to generate config files

- If we initially use docker compose to get started quickly, and later on want to move to k8s, we can use Kompose to convert our compose.yaml file to one or more sets of k8s deploy/service files
- Simply run this command in a terminal in the folder where the compose.yml file is placed:

```
kompose convert
```

Configuration as code

- If you create k8s config files, you should put it in (one of) the repositories and keep it versioned
- We can use the files to restore a cluster both locally and somewhere in the cloud
- In terms of what do locally (using Minikube) and in the cloud (using whatever k8s services the cloud provider offers), there's no difference .... in theory
- In practice it takes some time to move from Minikube and something like Google Cloud Services, mostly because there's a learning curve for the cloud providers management user interface

Exercise 3 – 45 minutes

- Do exercise 3 ....