# Microservices

Sprint 3 - Week 6

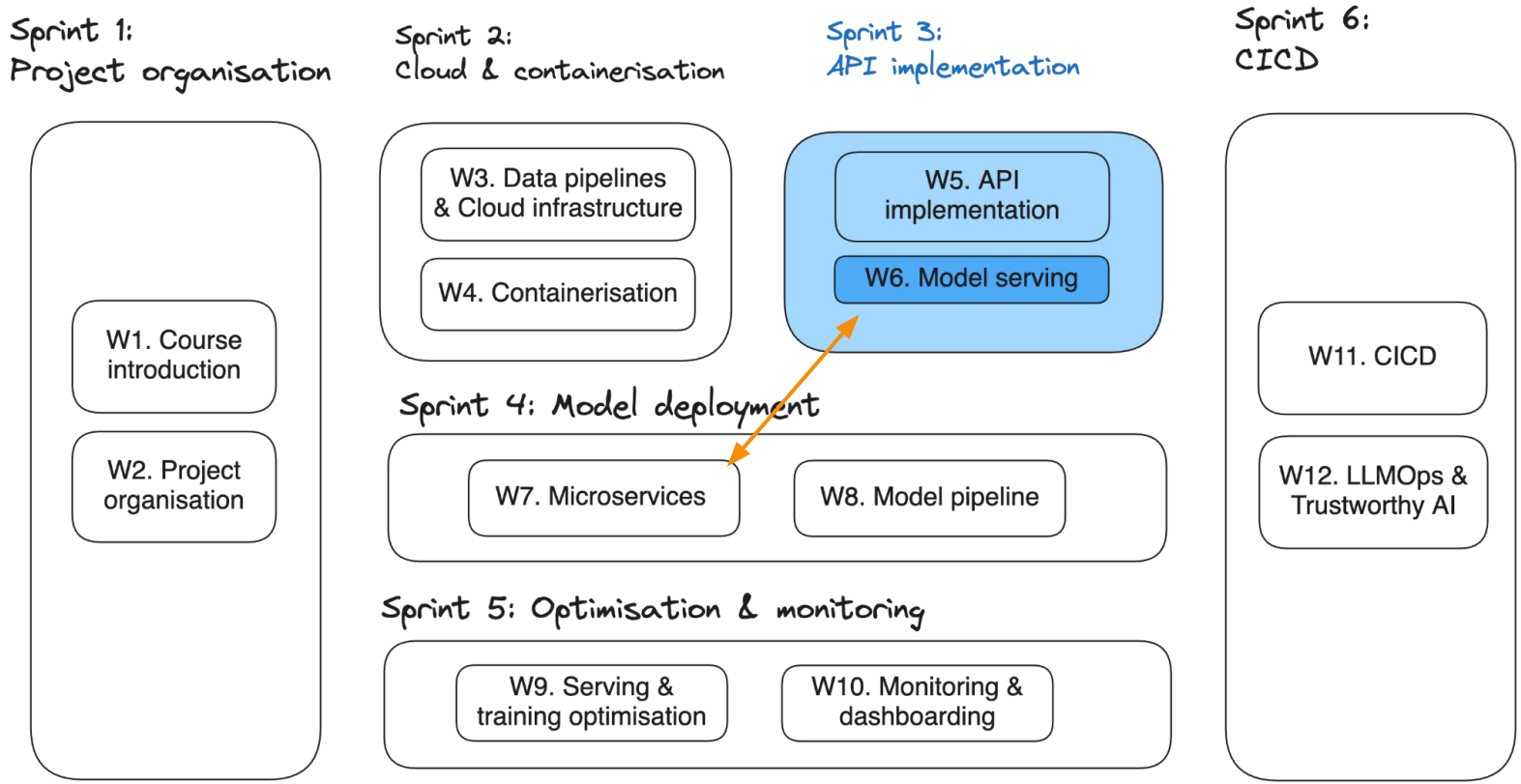
*INFO 9023 - Machine Learning Systems Design*

Thomas Vrancken (t.vrancken@uliege.be)
Matthias Pirlet (matthias.pirlet@uliege.be)

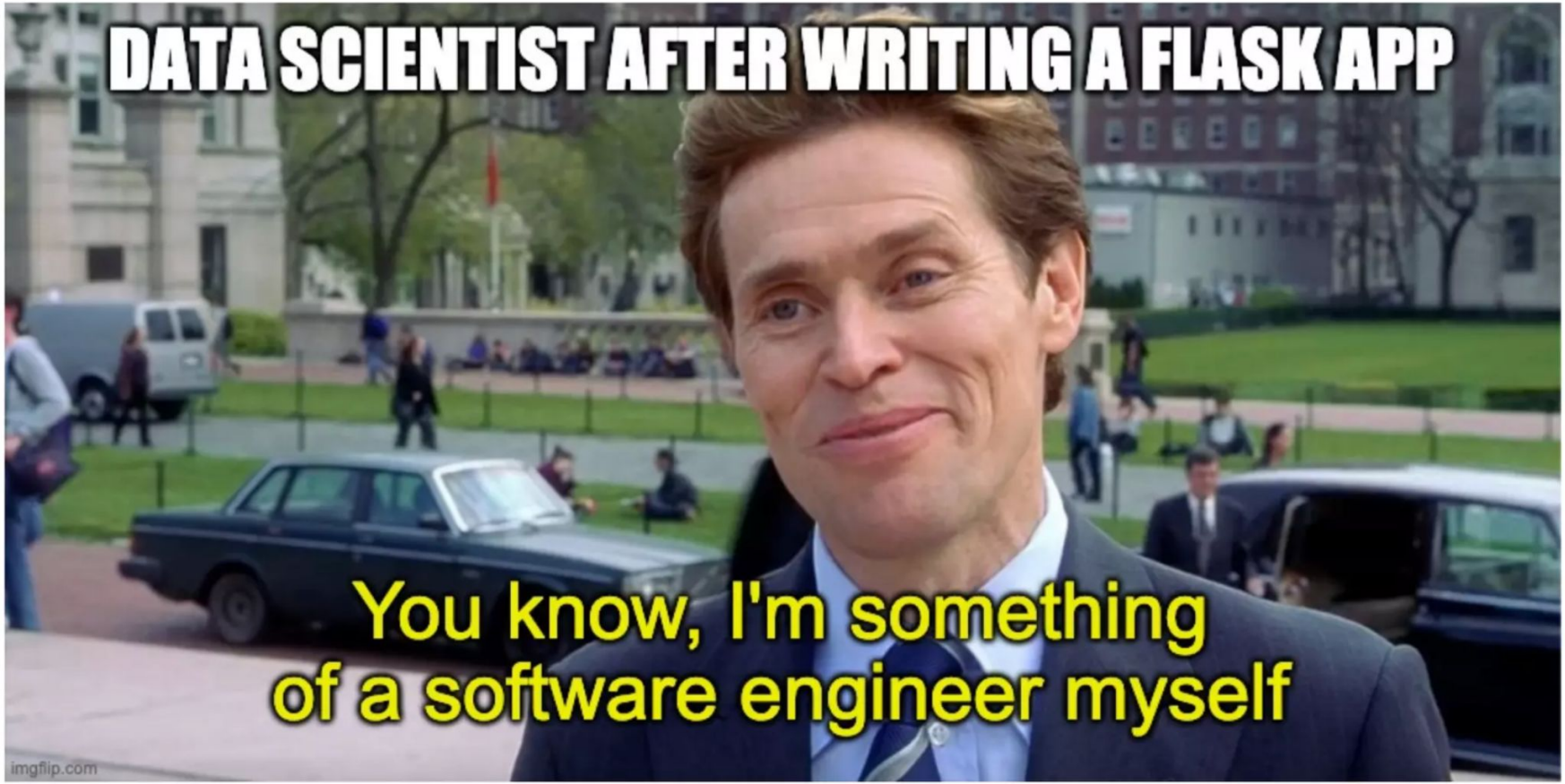
*2025 Spring*

# Status on our overall course roadmap

⚠️ Updated roadmap




LIÈGE université


DATA SCIENTIST AFTER WRITING A FLASK APP
You know, I'm something
of a software engineer myself
imgflip.com

# Agenda

## What will we talk about today

**Lecture**

1. Microservice Architecture
2. Kubernetes (k8s)
3. Serverless

**Directed work**

4. Serverless deployment on Cloud Run (k8s)

**Lecture**

5. Anatomy of a microservice

**Demo**

6. Kubernetes

# Microservice architecture

# Credits where credits are due






**Robbe Sneyders**
Office of the CTO @ ML6

**Ruwan Lambrichts**
Senior ML Engineer @ ML6

# Topic of this section

*Defining a general software architecture for microservices in your project/organisation.*

# Topic of this section

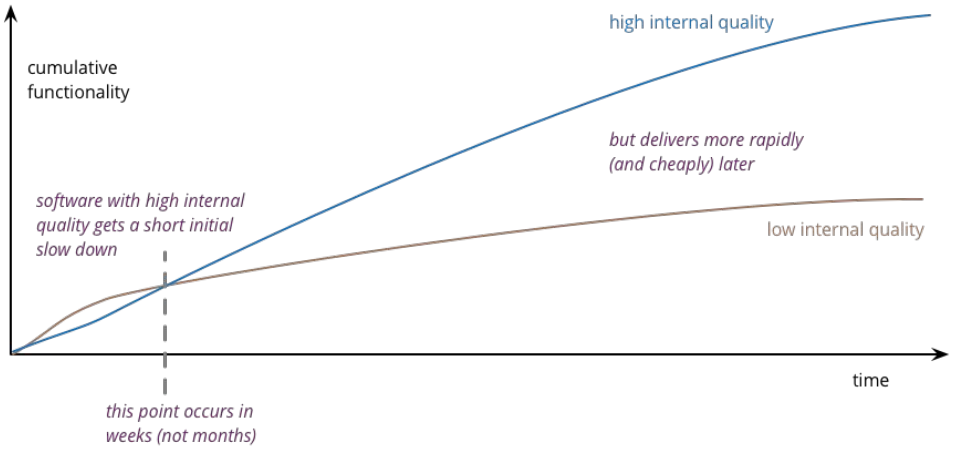
*Defining a general software architecture for microservices in your project/organisation.*

- The shared understanding that the expert developers have of the system design.
- The decisions you wish you could get right early in the project.

# Why does software architecture matter?

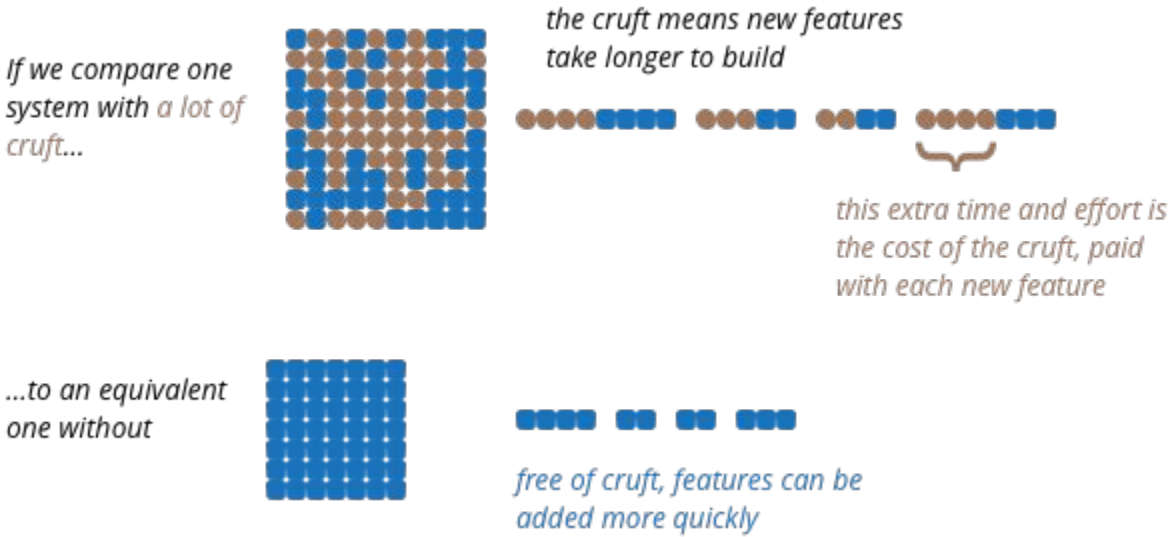
Software architecture enhances the **quality** of applications developed by your team, which is something stakeholders don't immediately perceive (and which might seem less important)

However, poor architecture is a major contributor to **cruft** - leading to code that is much harder to modify, causing features to arrive more slowly and with more defects (which the client cares about a lot)



# Cruft causes features to arrive more slow and with more defects

**Cruft** refers to unnecessary, outdated, or redundant code and design elements that accumulate over time in software projects. It can degrade maintainability, increase technical debt, and lead to inefficiencies in system performance and development velocity.




LIÈGE université

# Ideal state: Your team uses/builds boilerplates.

*Creating a shared understanding of the microservice design .*

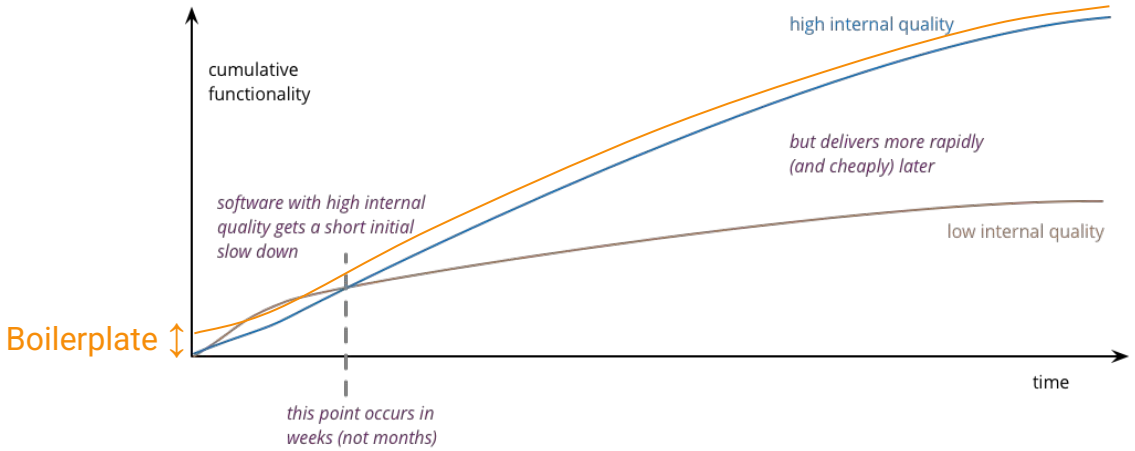


Captured in the **microservice boilerplate**, so every project can start from the tried and tested decisions already made on previous projects.

# Ideal state: Your team uses/builds boilerplates.

Take ‘correct’ decisions based on combined knowledge of developers and projects, implemented by the microservice boilerplate

- Saves time at start of project
- Projects don’t run into problems down the line, can evolve fast, and stay maintainable
- Consistent design across projects facilitates understanding of other projects



# What is a microservice?

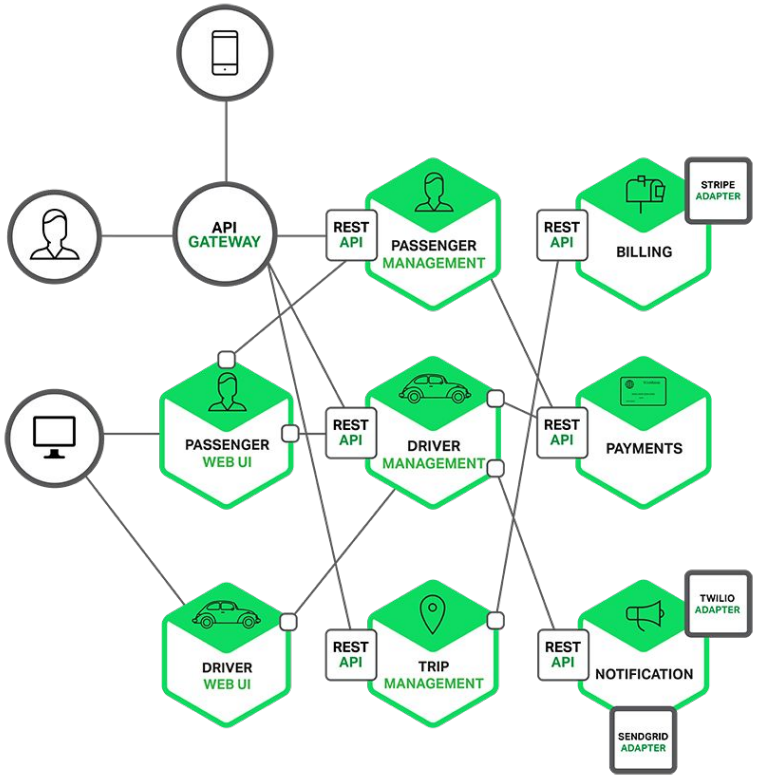
Some context

# What is a microservice?

*Using a microservices approach, software is composed of **small services** that communicate over **well-defined APIs** that can be **deployed independently**. These services are owned by small autonomous teams.*

# What is a microservice?

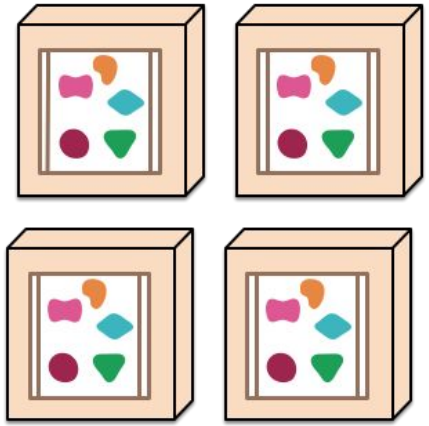
- Applying single responsibility principle at the architectural level
- As small as possible, but as big as necessary
  - “No bigger than your head”
- Advantages compared to monolithic architectures:
  - Independently deployable
  - Language, platform and technology independency between components
  - Distinct axes of scalability
  - Provides firm module boundary
  - Increased architectural flexibility
- Often integrated using REST over HTTP
  - Business domain concepts are modelled as resources



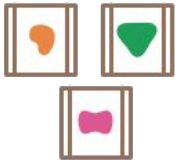
# Monolith vs microservice

*A monolithic application puts all its functionality into a single process...*

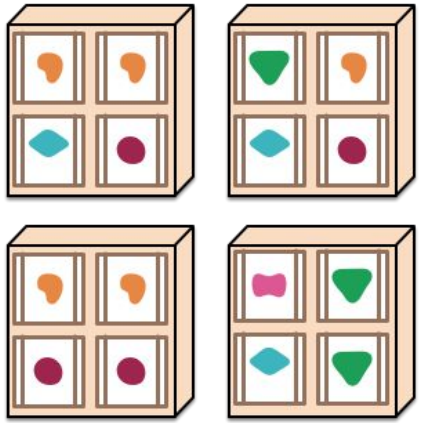
*... and scales by replicating the monolith on multiple servers*



*A microservices architecture puts each element of functionality into a separate service...*



*... and scales by distributing these services across servers, replicating as needed.*

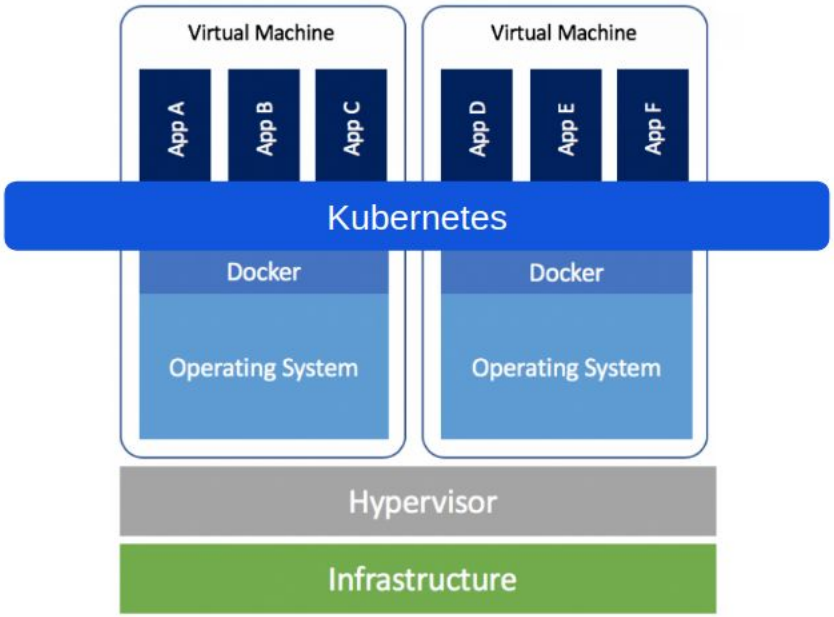


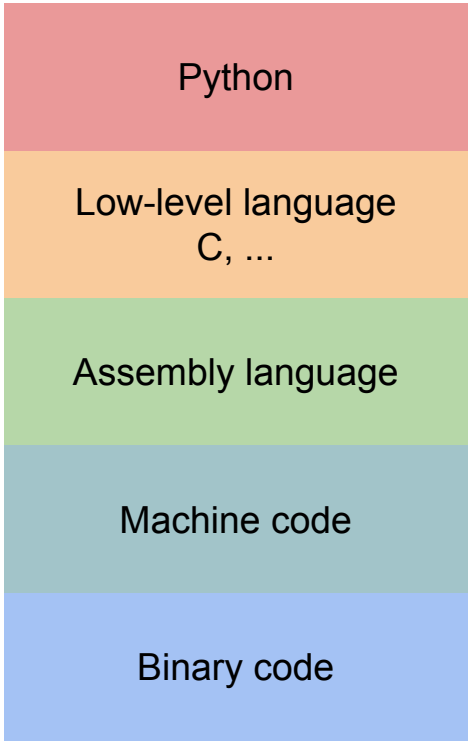
# Abstractions
# &
# Separation of concerns

# Abstraction

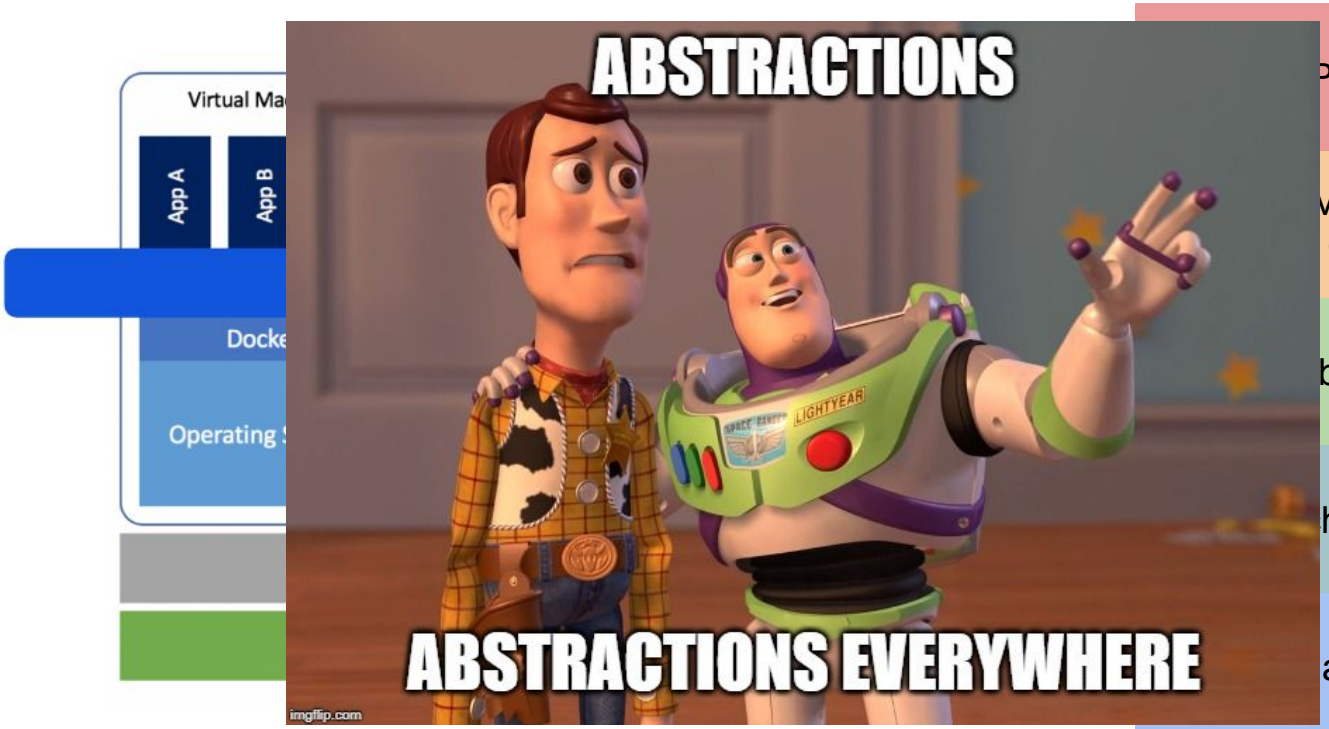
- Abstraction is the act of **representing essential features without including the background details** or explanations.
- Reduce complexity and allow efficient design and implementation of complex software systems.

# Abstractions are everywhere



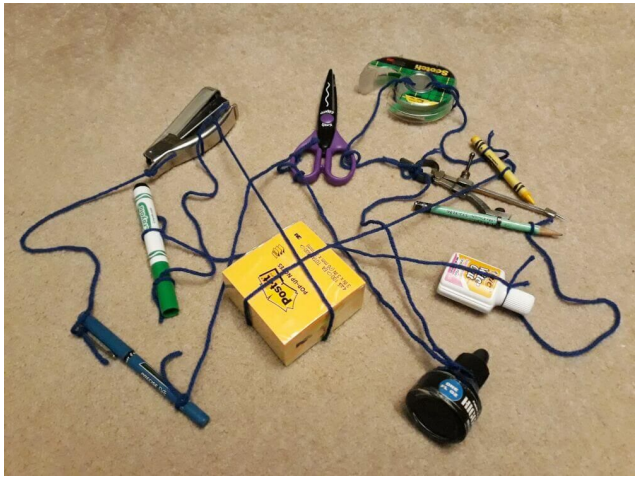



# Abstractions are everywhere



# Separation of concerns

- Separate code into sections that each address a separate concern
- Sections can evolve independently of other components
- Separation of concerns is a form of abstraction




Images: https://pusher.com/tutorials/clean-architecture-introduction

LIÈGE université

# Interfaces

- Describes the public actions and attributes that a type of component supports and implements.
- Defines the boundary across which two components can communicate.
- Components with an identical interface are interchangable.

# Interfaces

- In traditional OOP languages, an Interface defines a contract which is explicitly implemented.
- In Python, you don't have to explicitly declare an interface.

```
interface Pet {
    public Owner owner();
    public void pat();
}

class Dog implements Pet {

    public Owner owner() {
        return this.get_owner();
    }

    public void pat() {
        return love;
    }
}
```

```
class Dog:

      @property
      def owner(self):
            return self._get_owner()

      def pat(self):
            return love


dog = Dog()
try:
      dog.owner()
      dog.pat()
      print("dog is a pet")
except:
      print("dog is not a pet")
```

LIÈGE université

# Kubernetes

# Why do we need kubernetes?

**Virtual Environment**

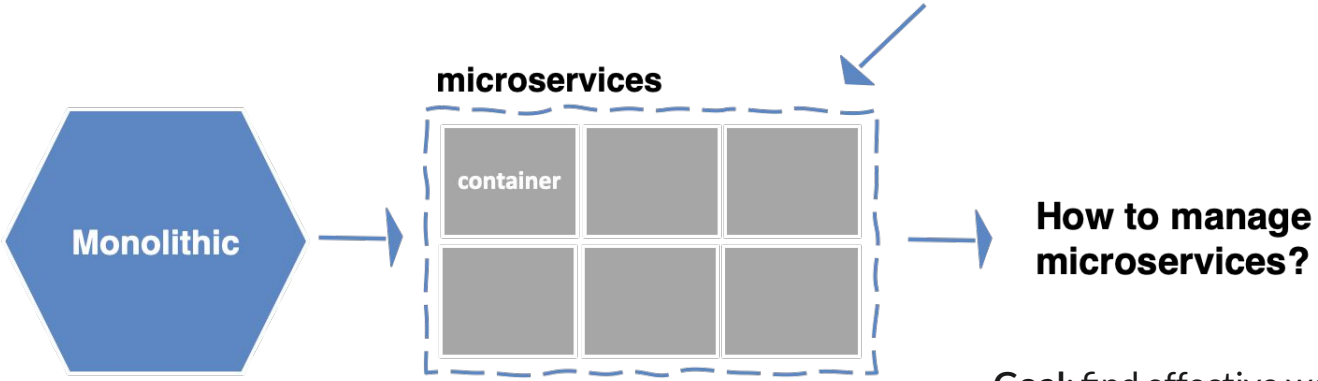
**Pros:** remove complexity
**Cons:** does not isolate from OS

**Virtual Machines**

**Pros:** isolate OS guest from host
**Cons:** intensive use hardware

**Containers**

**Pros:** lightweight
**Cons:** issues with security, scalability, and control

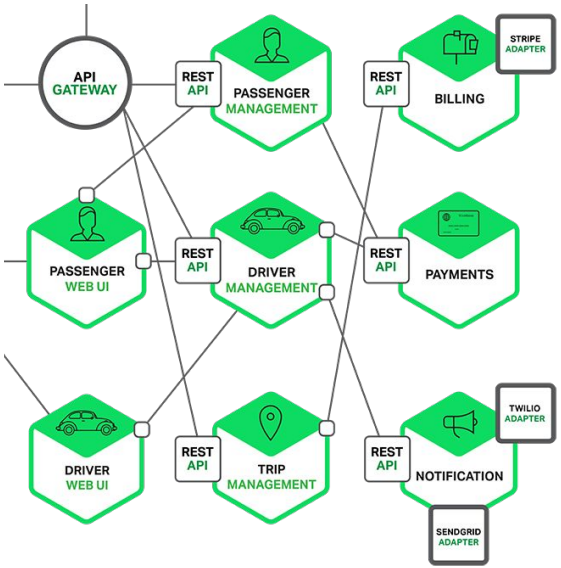



**Goal**: find effective ways to **deploy** the different microservices that build our system.

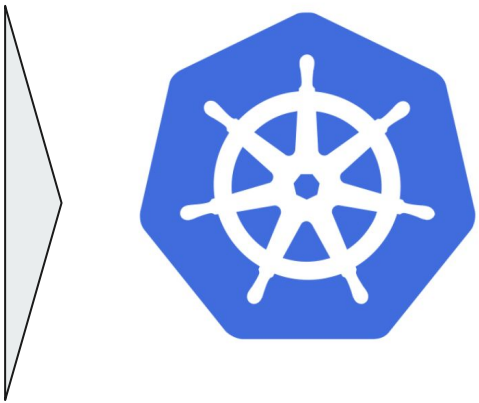
# Kubernetes (k8s)

- K8s is an **orchestration tool** for managing distributed services or containerized applications across a **distributed cluster of nodes**.
- K8s is installed on the servers (on-prem or in the Cloud) and manages the different microservices deployed on it
- K8s contains two types of nodes: A single **master node** and **worker nodes**.
- K8s users define rules for how container management should occur, and then K8s handles the rest!

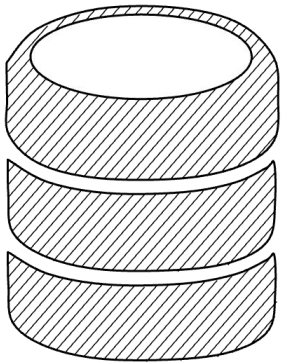
# Kubernetes (k8s)




Microservices



Kubernetes

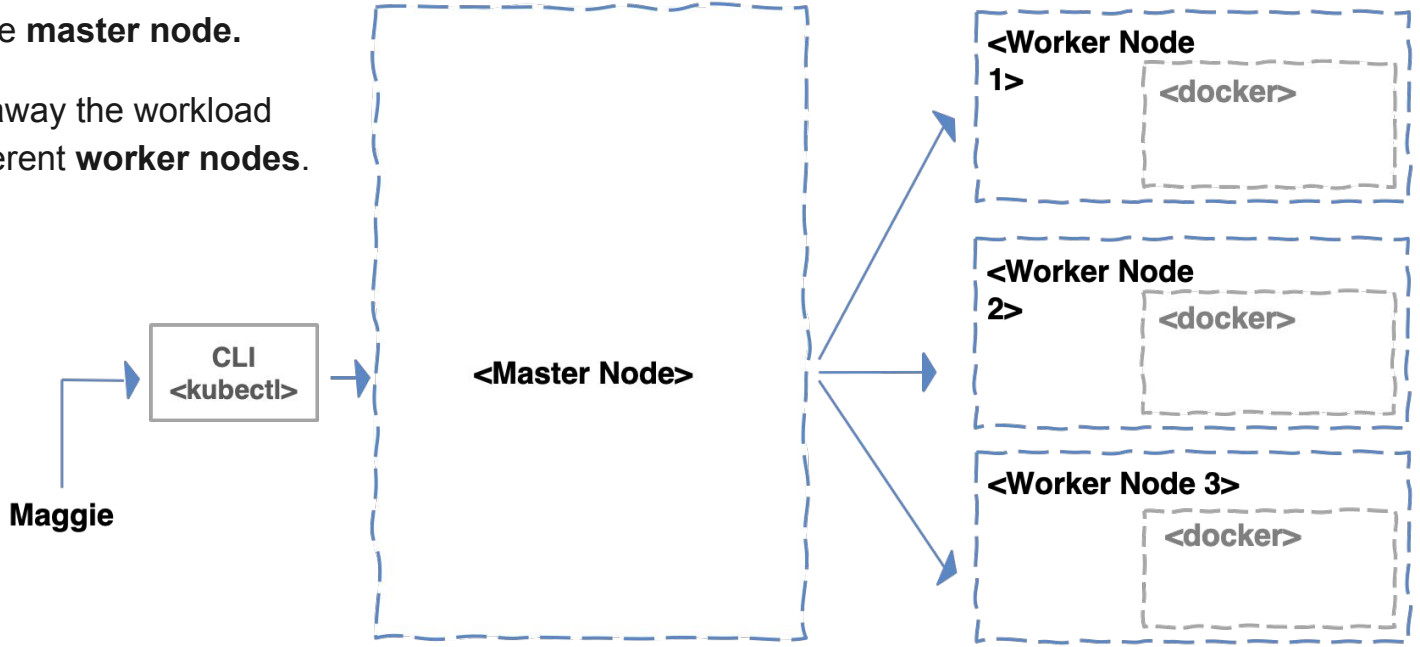


Servers
(Cloud or
on-prem)

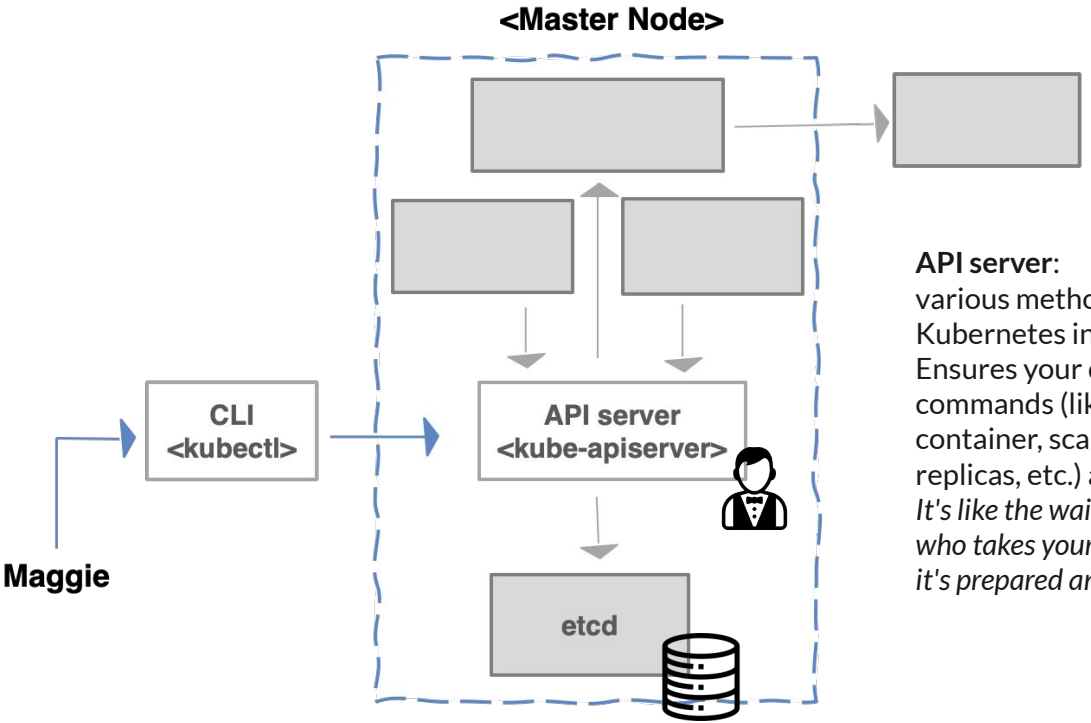
# Architecture of kubernetes

Interact with the **master node.**

k8s abstracts away the workload balance to different **worker nodes**.



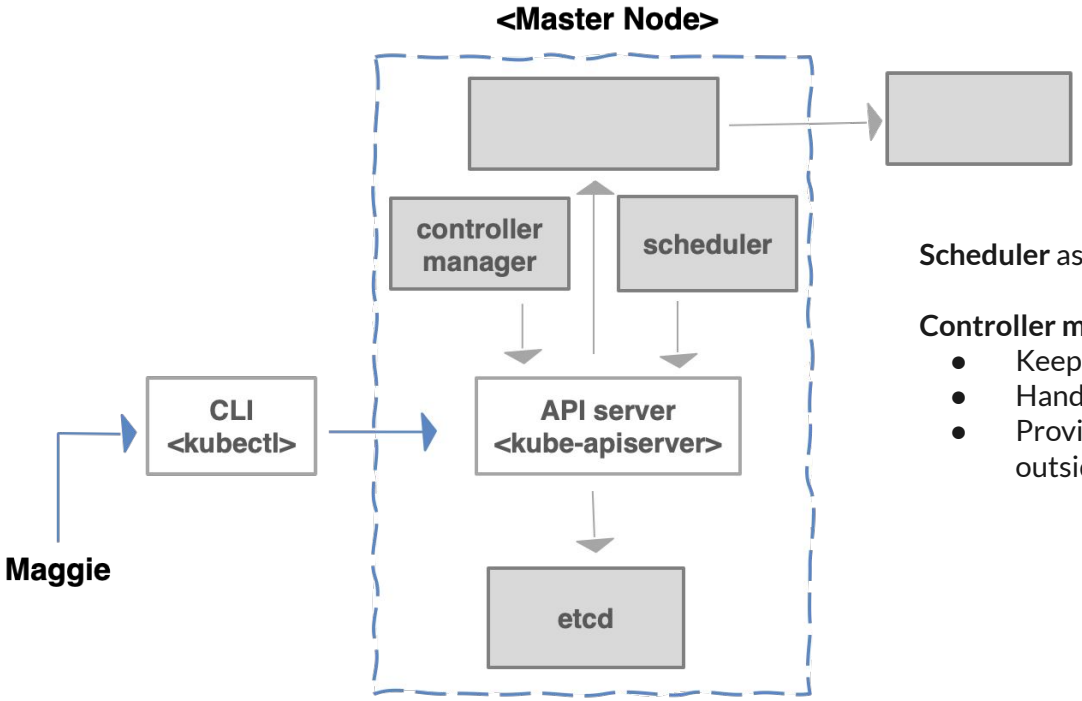
# Architecture of kubernetes




**API server**:
various methods to access the Kubernetes instance.
Ensures your developer commands (like launching a new container, scaling the number of replicas, etc.) are carried out.
*It's like the waiter of a restaurant who takes your order and ensures it's prepared and served correctly.*

**etcd:**
works as backend for service discovery.
Database that keeps all the configuration details and the state of your entire Kubernetes cluster

# Architecture of kubernetes

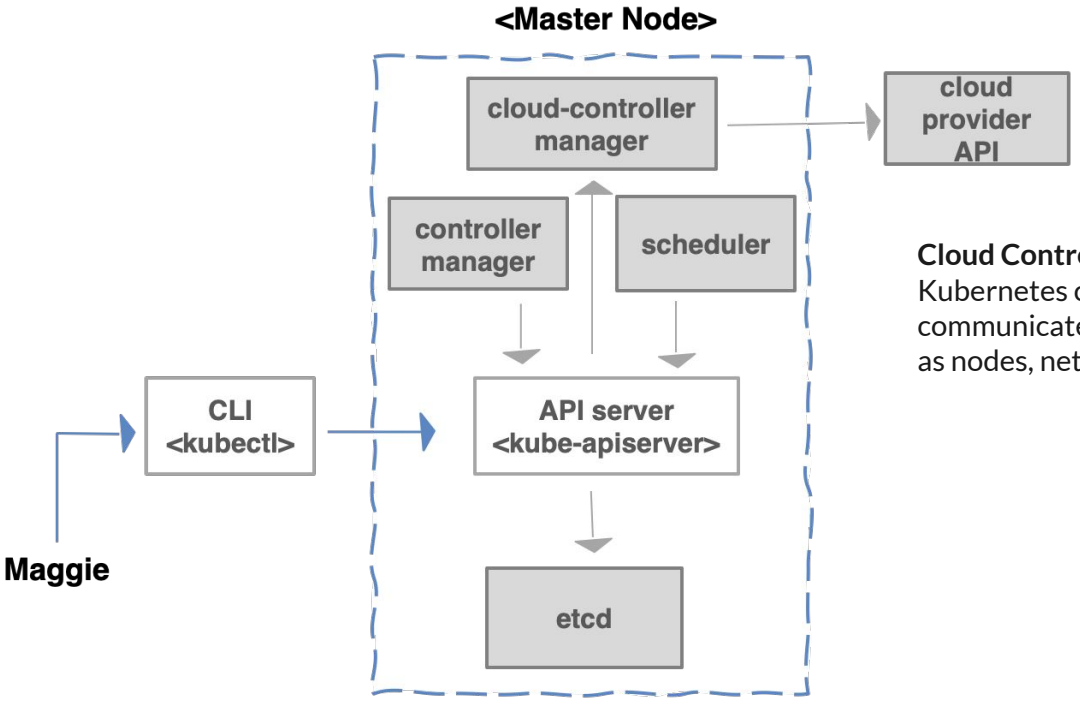



**Scheduler** assigns microservices to worker nodes.

**Controller manager:**

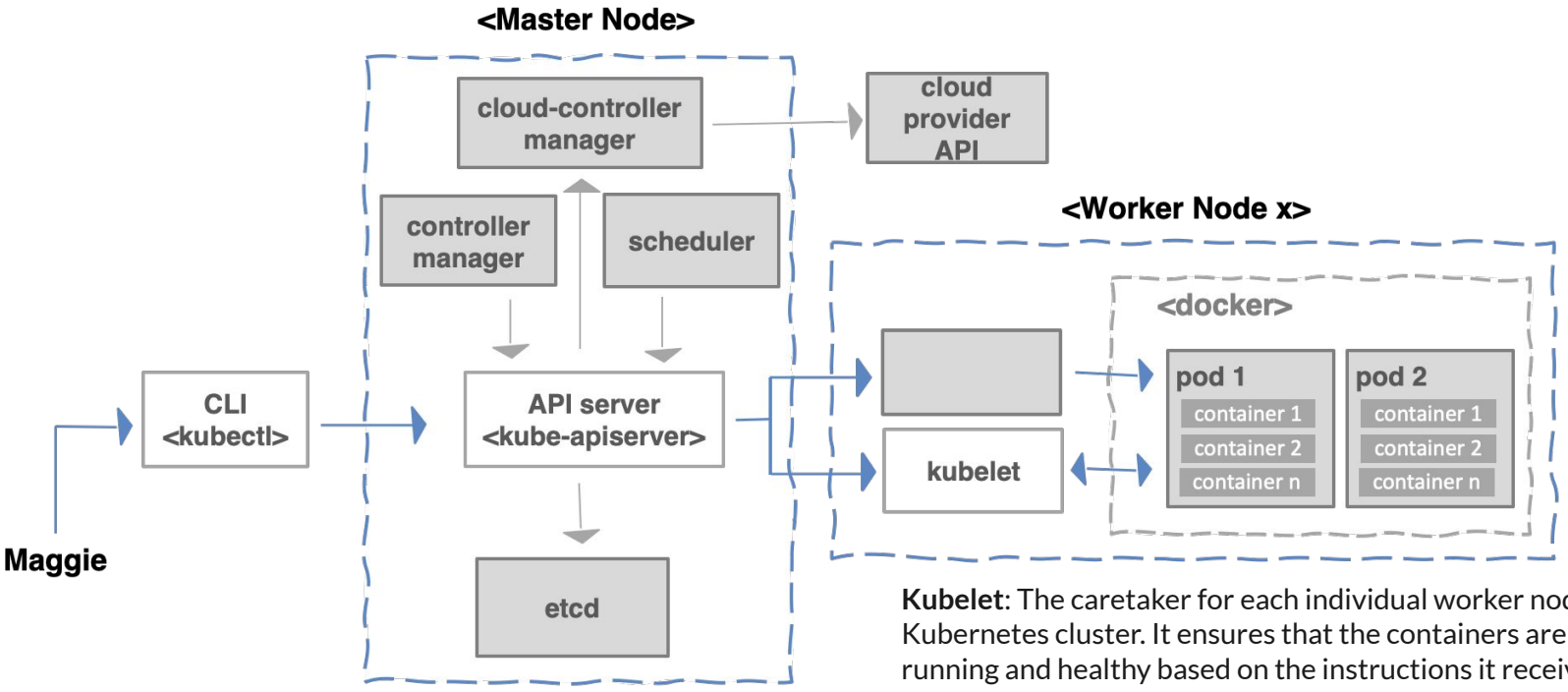
- Keeps track of worker nodes
- Handles node failures and replicates if needed
- Provide endpoints to access the application from the outside world

# Architecture of kubernetes



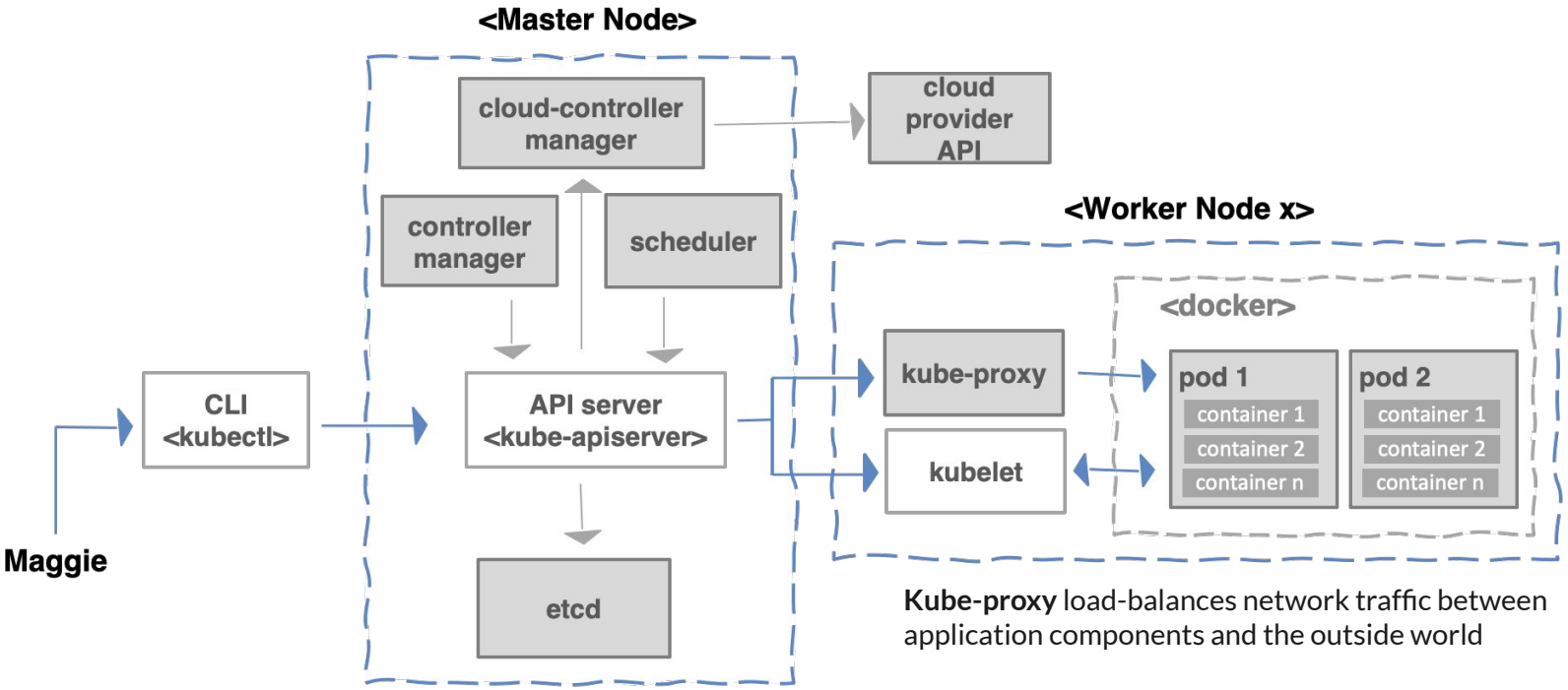

**Cloud Controller Manager**: A bridge between your Kubernetes cluster and the cloud provider's services. It communicates with cloud provide regarding resources such as nodes, networking and IP addresses

# Architecture of kubernetes




**Kubelet**: The caretaker for each individual worker node in a Kubernetes cluster. It ensures that the containers are running and healthy based on the instructions it receives from the API server.

# Architecture of kubernetes



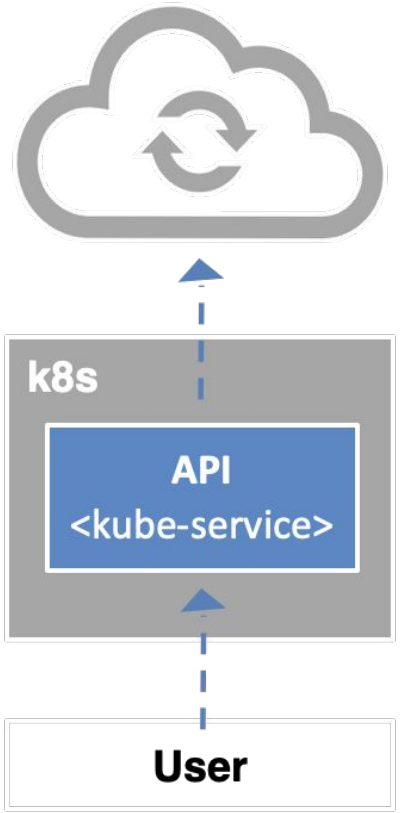

**Kube-proxy** load-balances network traffic between application components and the outside world

# Why use kubernetes

Many reasons:

- Velocity
- Scaling (of both software and teams)
- Abstracting the infrastructure
- Efficiency

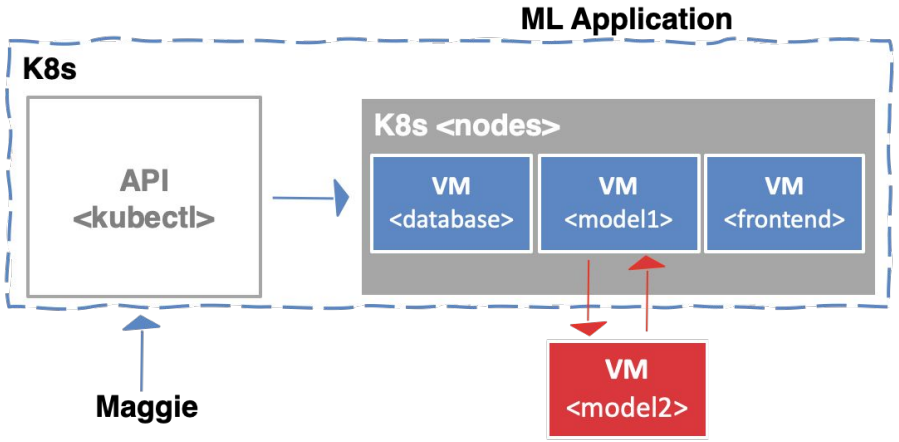
All these aspects relate to each other to speed up process that can reliably deploy software.



# Why use kubernetes: Velocity

Velocity, in this context, means the speed and efficiency with which a team can deliver updates, features, or entire applications.

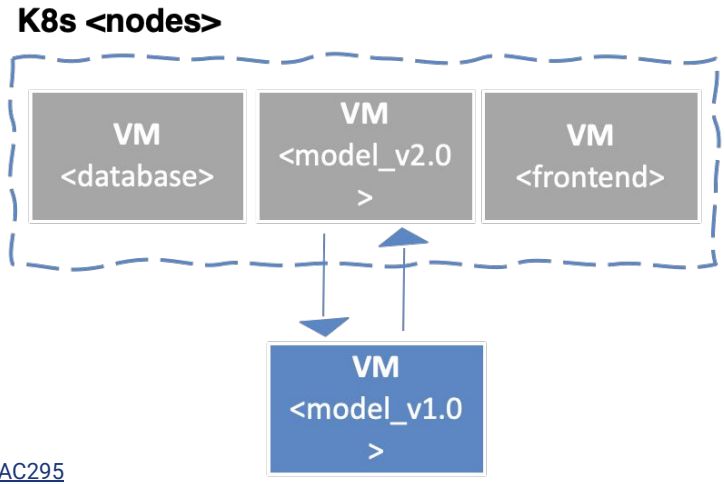
It also includes the speed to which your system adapts to issues.



# Why use kubernetes: Velocity

Velocity is enabled by:

- **Immutable system**: you can't change running container, but you create a new one and replace it in case of failure (allows for keeping track of the history and load older images)

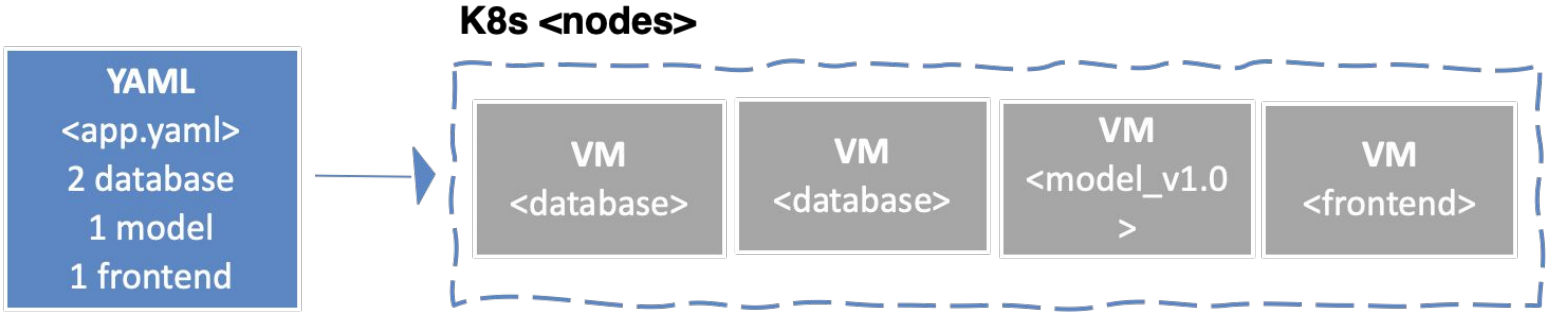


# Why use kubernetes: Velocity

Velocity is enabled by:

- **Declarative configuration**: you can define the desired state of the system restating the previous declarative state to go back. Imperative configuration are defined by the execution of a series of instructions, but not the other way around.

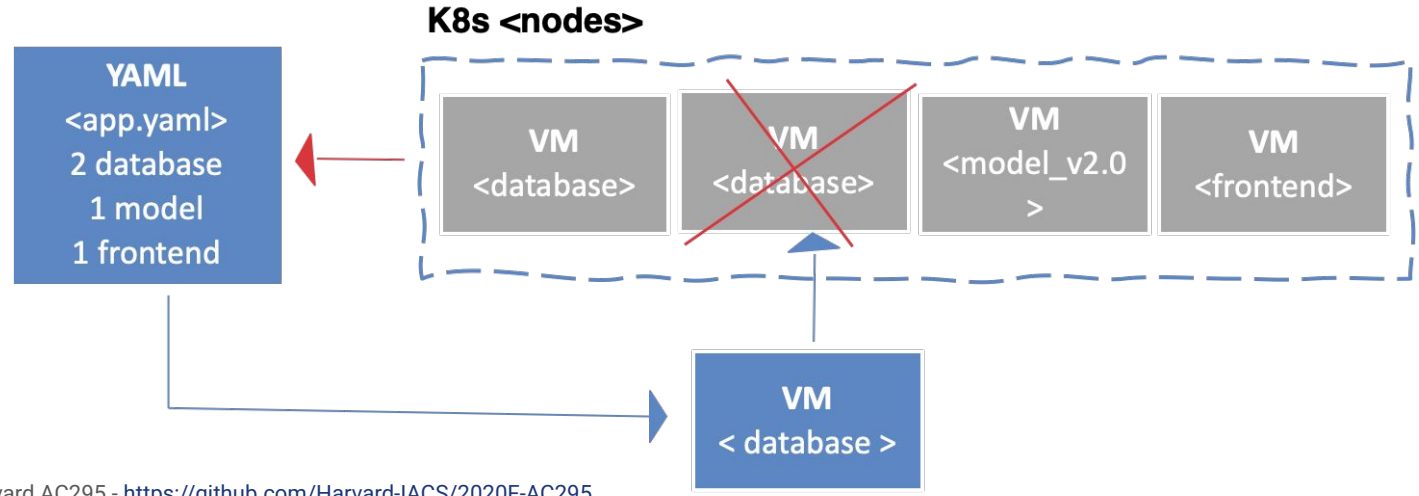
  (Just like SQL, which is also a declarative language! Python on the other hand is imperative).



# Why use kubernetes: Velocity

Velocity is enabled by:

- **Online self-healing systems**: k8s takes actions to ensure that the current state matches the desired state (as opposed to an operator enacting the repair)



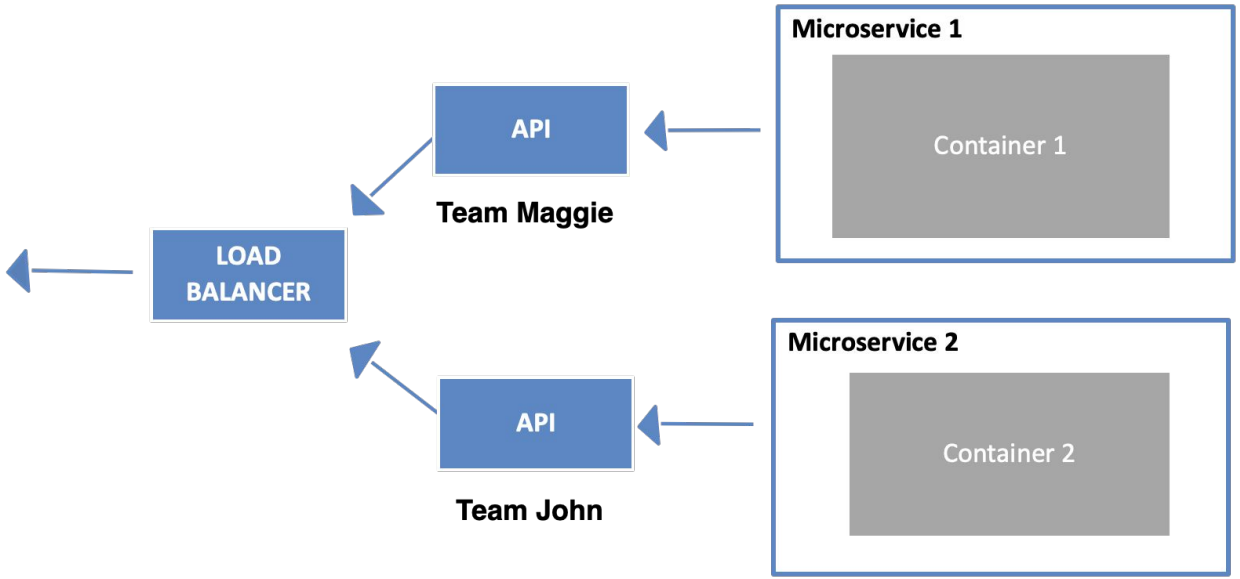
# Why use kubernetes: Scaling

You can **scale** in terms of working with different **teams** on different **applications** with more **usage** with the same structure.

Kubernetes provides numerous **advantages** to address scaling:

- **Decoupled architectures**: each component is separated from other components by defined APIs and service load balancers ( ⇒ microservices) .
- Easy scaling for **applications and clusters**: simply changing a number in a configuration file, k8s takes care of the rest (part of declarative).
- Scaling **development teams** with microservices: small team is responsible for the design and delivery of a service that is consumed by other small teams

# Why use kubernetes: Scaling




The **load balancer** directs incoming network traffic (requests) to the appropriate servers (in this case, the containers running in pods) based on the service requested, the current traffic conditions and the health of the servers.

# Why use kubernetes: Efficiency

There are concrete economic benefit to the abstraction because tasks from multiple users can be packed tightly onto fewer machines

- **Consume less energy** (ratio of the useful to the total amount)
- **Limit costs of running a server** (power usage, cooling requirements, datacenter space, and raw compute power)
- Create quickly a developer's **test environment** as a set of containers
- **Reduce cost of development instances** in your stack, liberating resources to develop others that were cost-prohibitive

👆Note: Those advantages come with **scale**. If enough developers/applications share a cluster it becomes really efficient. A single small application on k8 might be overkill.

# Keeping team size efficient

## 2 pizza teams

What is the optimal number of people to work in a specific team?



https://docs.aws.amazon.com/whitepapers/latest/introduction-devops-aws/two-pizza-teams.html

# Keeping team size efficient

## 2 pizza teams

What is the optimal number of people to work in a specific team?

Team should be able to **be fed with two pizzas!** (~ 6 to 10 people)

Enables

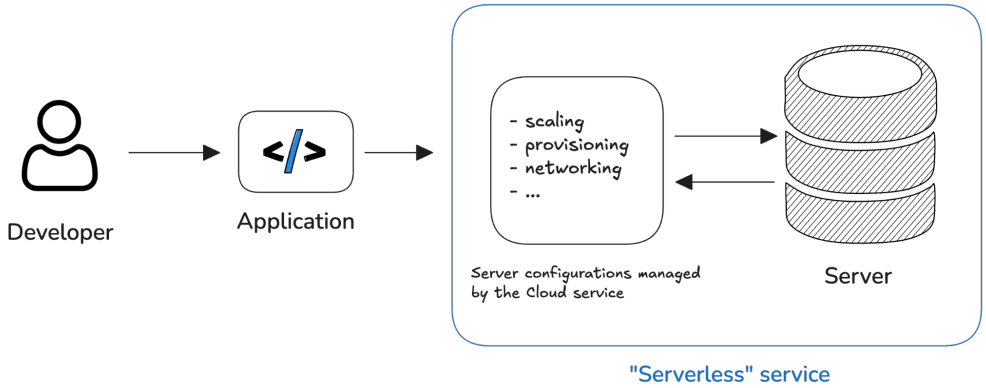
- Collaboration
- Communication
- Culture fit

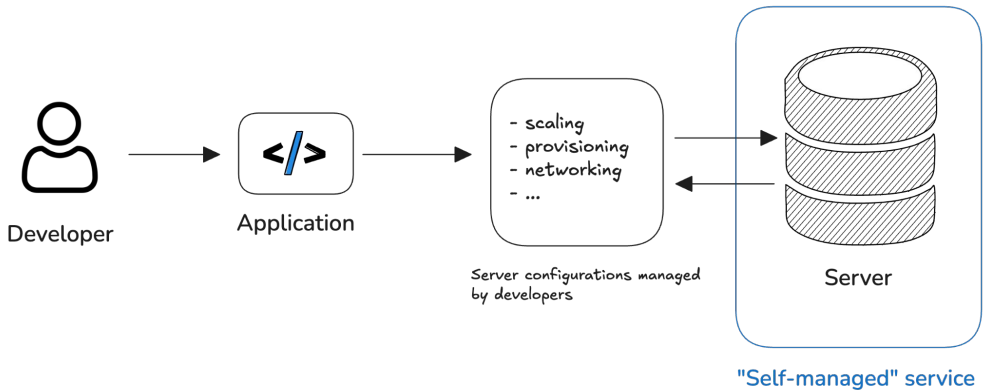


https://docs.aws.amazon.com/whitepapers/latest/introduction-devops-aws/two-pizza-teams.html

# Serverless computing

# What is Serverless Computing?



# What is Serverless Computing?

**Serverless**

- Cloud-native
- Developers don’t have to manage servers
- Cloud provider abstracts away the infrastructure, provisioning, maintaining, and scaling the server infrastructure
- Developers just have to package their codes in a container and deploy it
- Opposite is **self-managed**

| **Pros** | **Cons** |
| --- | --- |
| ● More efficient use of time for developers<br>● Often cost effective (pay for what you use)<br>● Simplified operations<br>● Better adoption of DevOps / MLOps practices<br>  ○ Team collaboration / rotation<br>  ○ Stateless containerised application | ● Less control over the server architecture<br>  ○ Type of compute<br>  ○ Interaction between components<br>● Vendor lock-in<br>● Debugging |



https://www.redhat.com/en/topics/cloud-native-apps/what-is-serverless

# Examples of serverless vs self-managed services

**Serverless**

Cloud Run

Lambda function

**Self-managed**

Compute Engine

EC2

# Directed Work: Serverless API deployment on Cloud Run

# Anatomy of a microservice

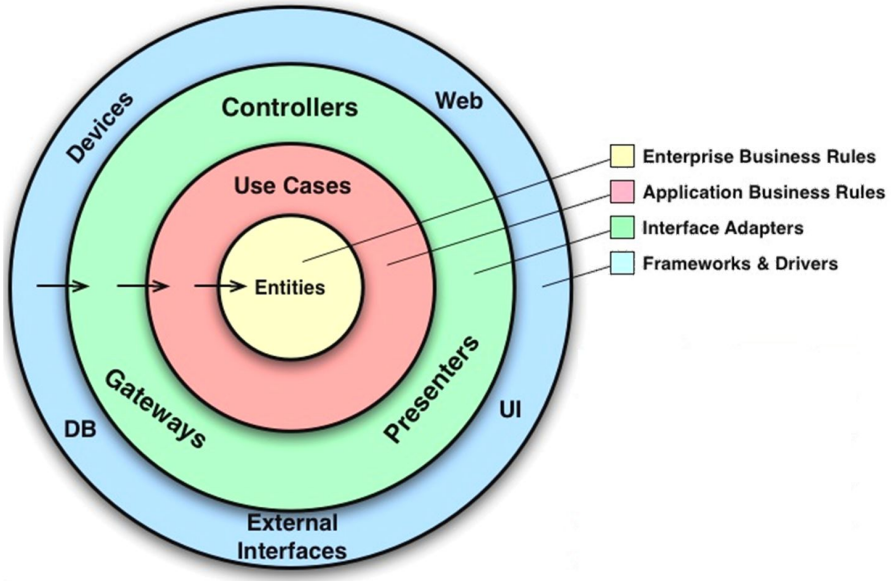
# Clean architecture

# The Clean architecture

By Robert C. Martin (Uncle Bob)

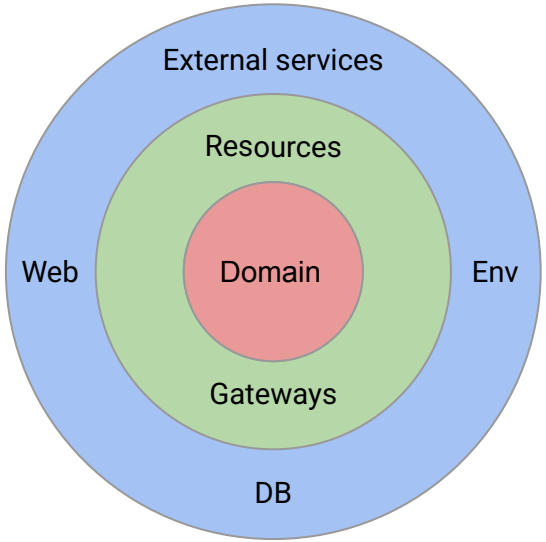
- Clear separation of of concerns between components.
- Dependency rule:
  Dependencies can only point inwards. Inner circle should not be dependent of outer circle specifics.
- Interfaces between each layer.
- Only isolated, simple data structures are passed across the boundaries.

Very OOP focused




https://en.wikipedia.org/wiki/Robert_C._Martin

# The Clean architecture - tailored

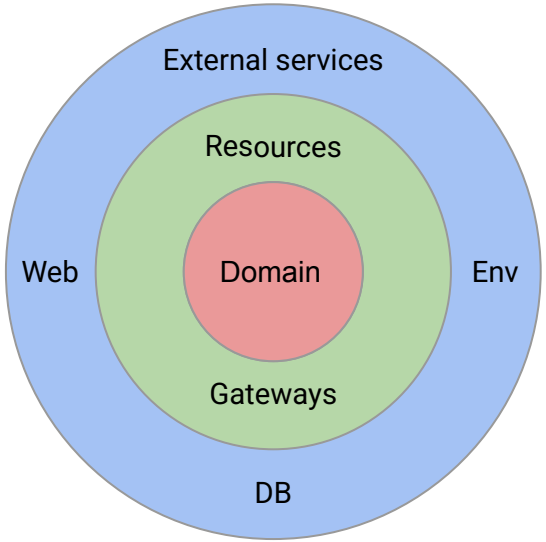


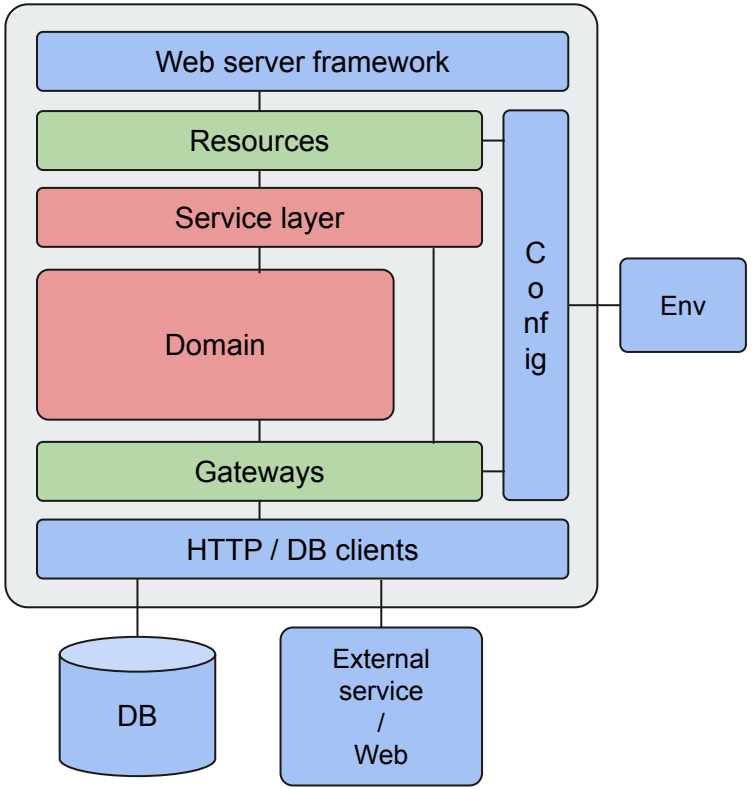

**Infrastructure**:
Frameworks, environment, and storage and web access.

**Interface adapters**:
Encapsulate the infrastructure layer, and provide a Python API for the domain.

**Domain**:
All logic core to the microservice.

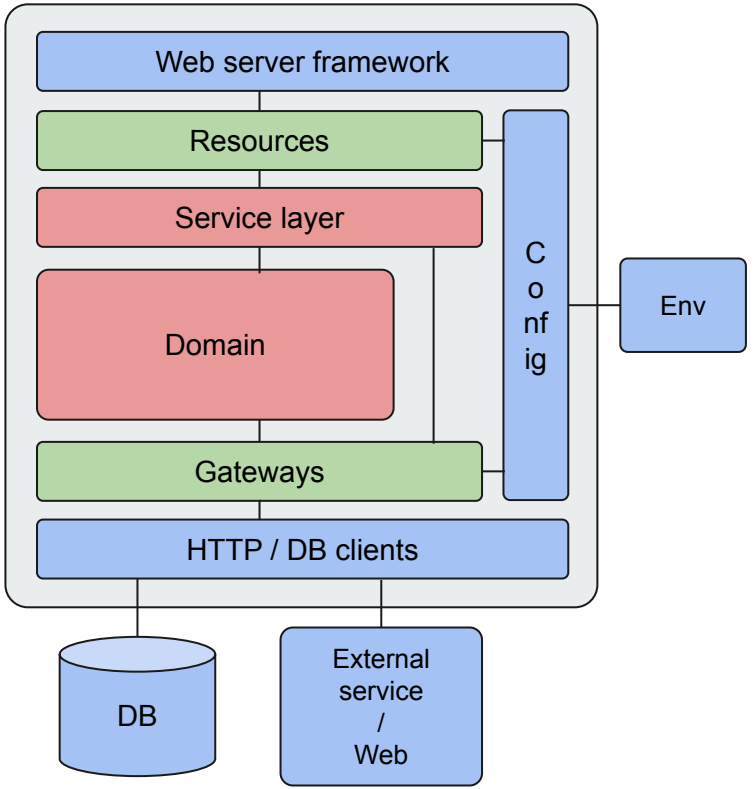
# The clean anatomy of a microservice



# The clean anatomy of a microservice

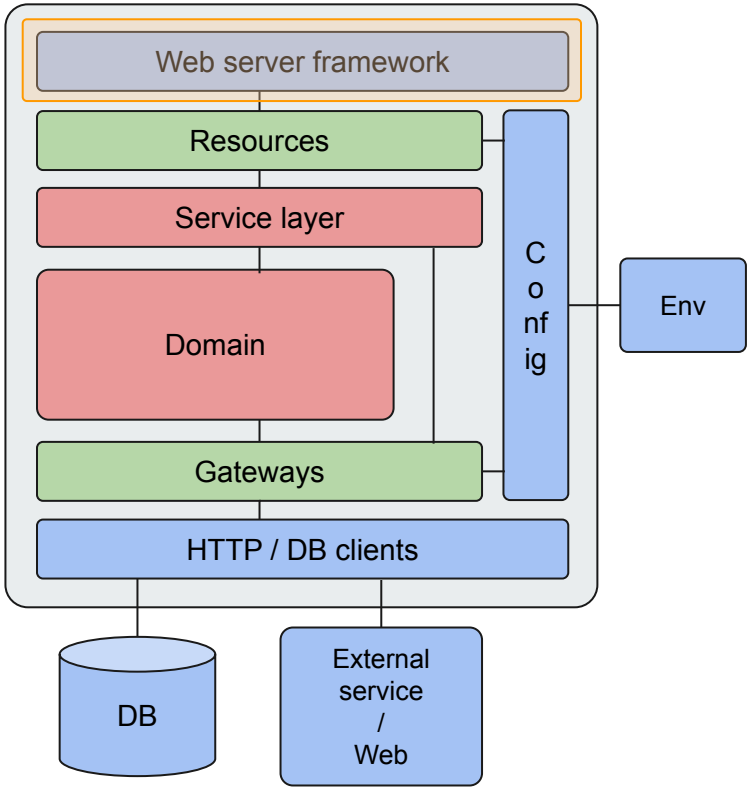
- Microservices display consistent internal structure composed of some or all of the displayed layers.
- The microservice boilerplate aims to provide minimal viable structure that is shared by all microservices.



# The anatomy of a microservice

Web server framework

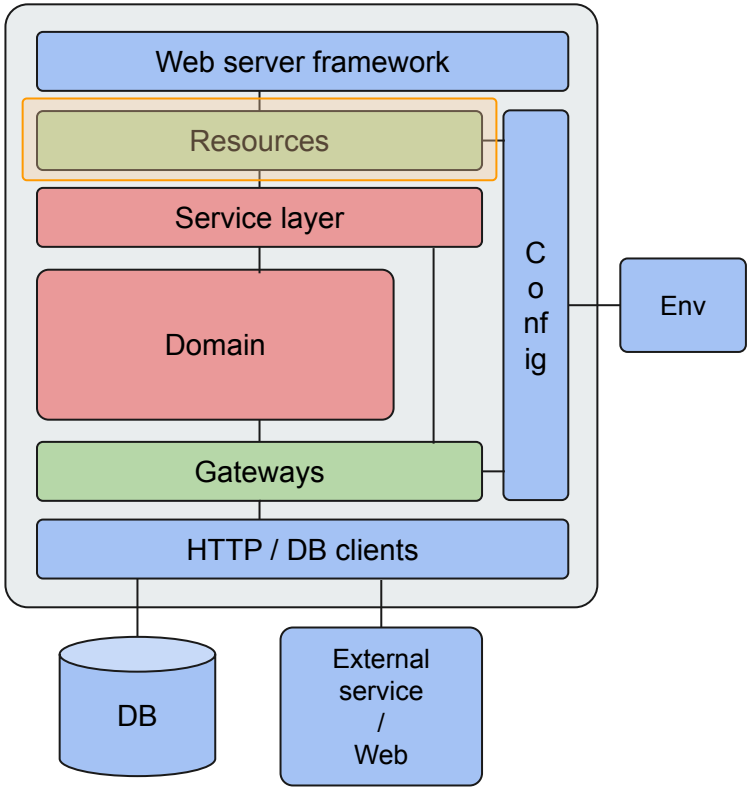
- Act as a mapping between the exposed application protocol (eg. HTTP) and Python.
- Validate requests and responses against application protocol.



# The anatomy of a microservice

Resources

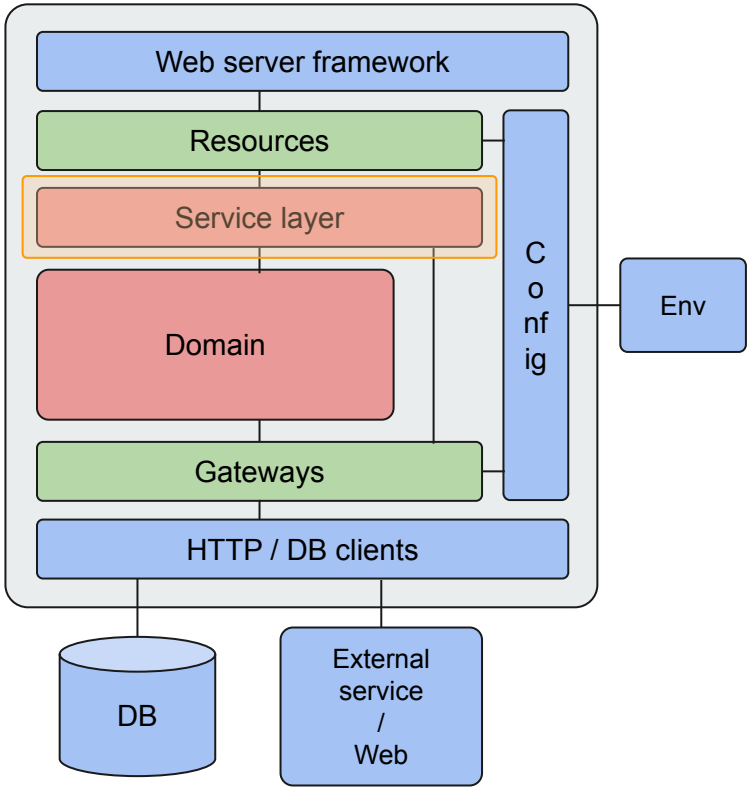
- Act as a mapping between the web server framework and the service domain.
- Thin layer for sanity checking requests and providing protocol specific responses.



# The anatomy of a microservice

Service layer

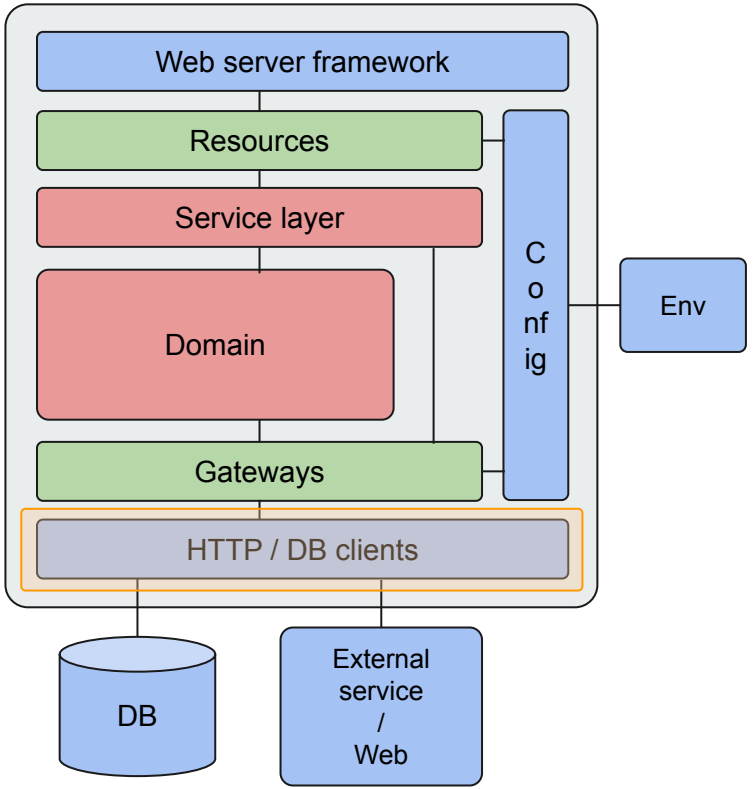
- Defines the interface of the domain with its set of available operations.
- Can be reused by multiple protocol clients; HTTP, RPC, ...



# The anatomy of a microservice

HTTP / DB clients

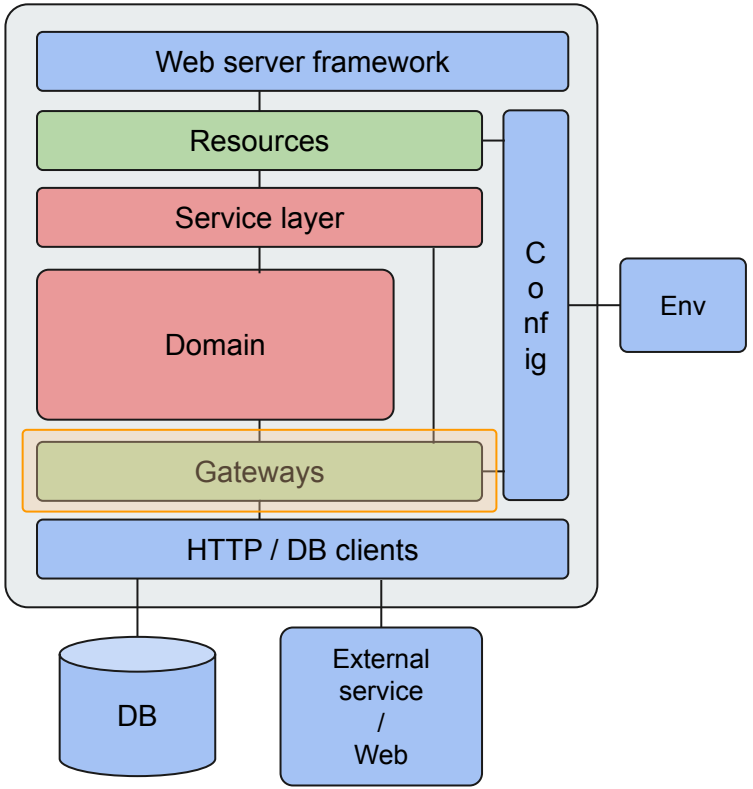
- Clients to handle connections to external datastores and services
- HTTP-client:
  Client that understands the underlying protocol to handle the request-response cycle.
  (eg. python requests to other services)
- DB-client:
  Provide (often client-specific) python representation of data in datastores



# The anatomy of a microservice

Gateways

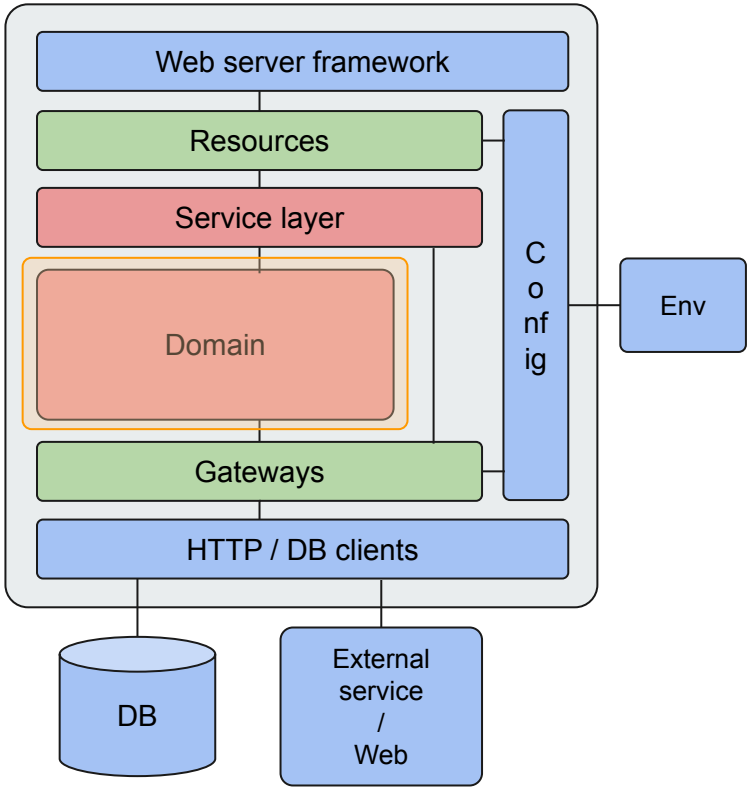
- Isolates domain from details in the database access code.
- Groups query construction code to minimize duplicate query logic.



# The anatomy of a microservice

Domain

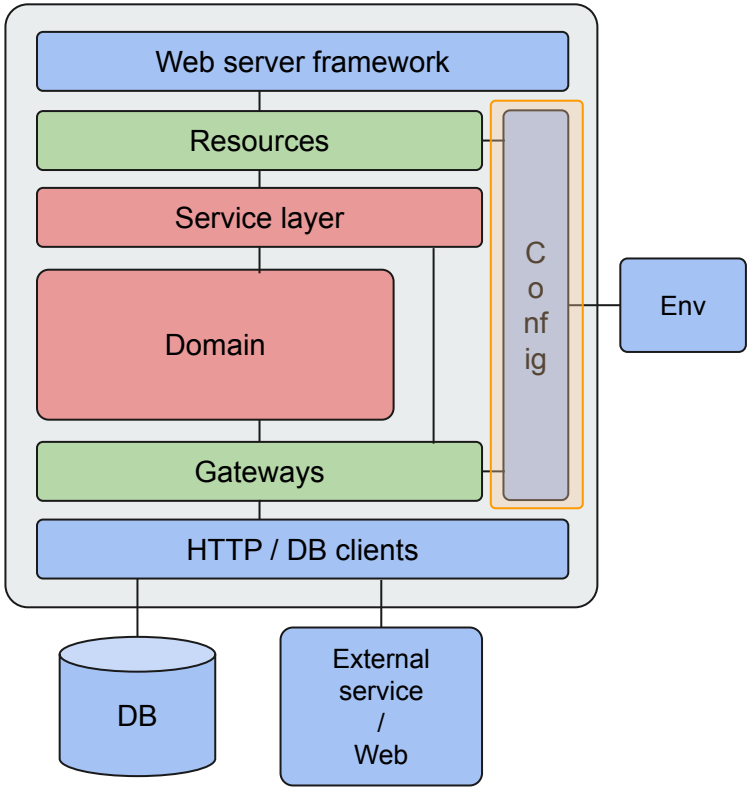
- Contains the actual service logic.
- The domain consists of pure Python with some core libraries. It is independent of any used frameworks and IO clients, and only depends on the interface of the resources and gateways.



# The anatomy of a microservice

Config

- Reads and groups the configuration of the service.
- All configurable parameters of the service should be extracted as config, so they can be set and changed in the environment at run time.



# Advantages of the clean anatomy

- **Independent of externals**
  When any of the external parts of the system become obsolete, like the database, or the web framework, you can replace those obsolete elements with a minimum of fuss.

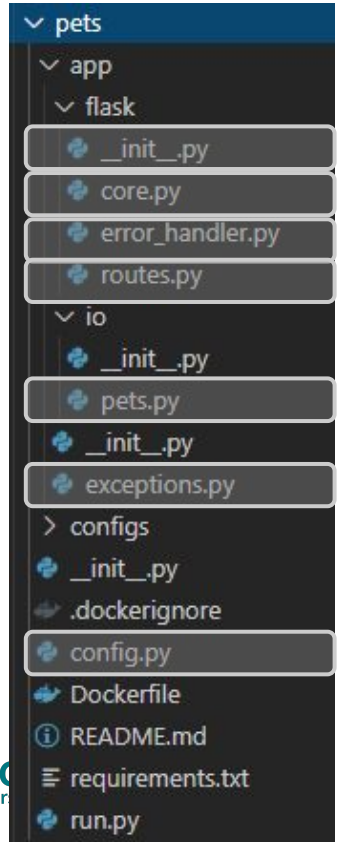
- **Scalable and maintainable**
  A modular architecture allows for scaling without an explosion in complexity. New modules can be added independently of others. Multiple developers can work in parallel on different modules.

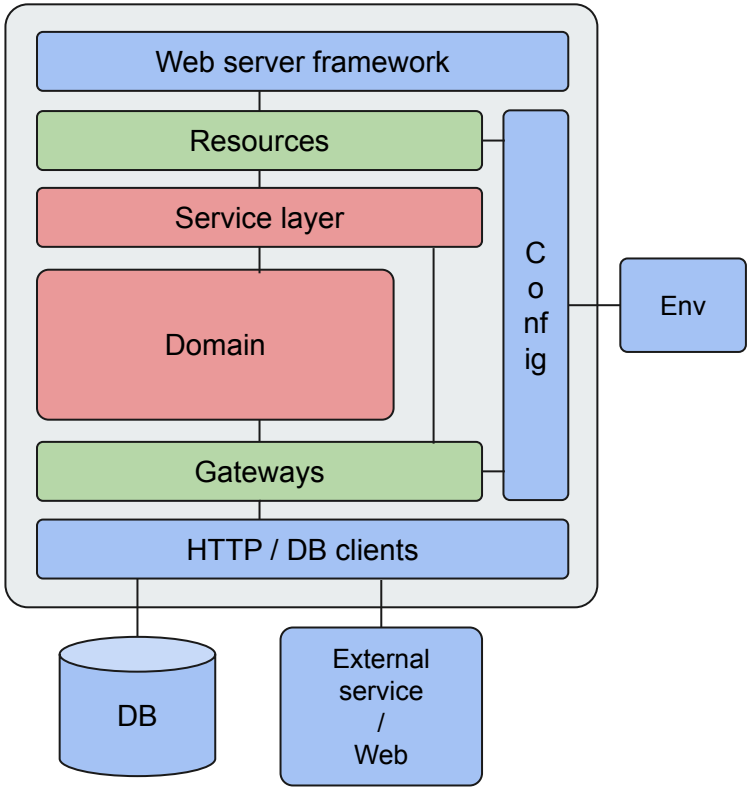
- **Deploy everywhere**
  Since the configuration is separated, the same code can be deployed in different environments without changes.

# Example of a microservice anatomy

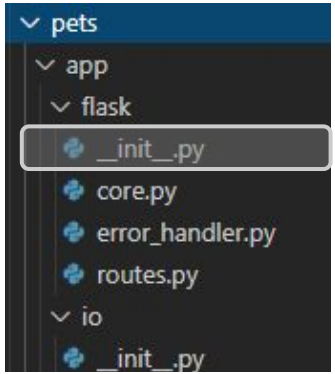
Mapping the structure on the pets example

# The anatomy of a microservice: pets example

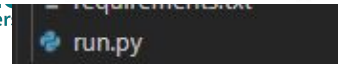


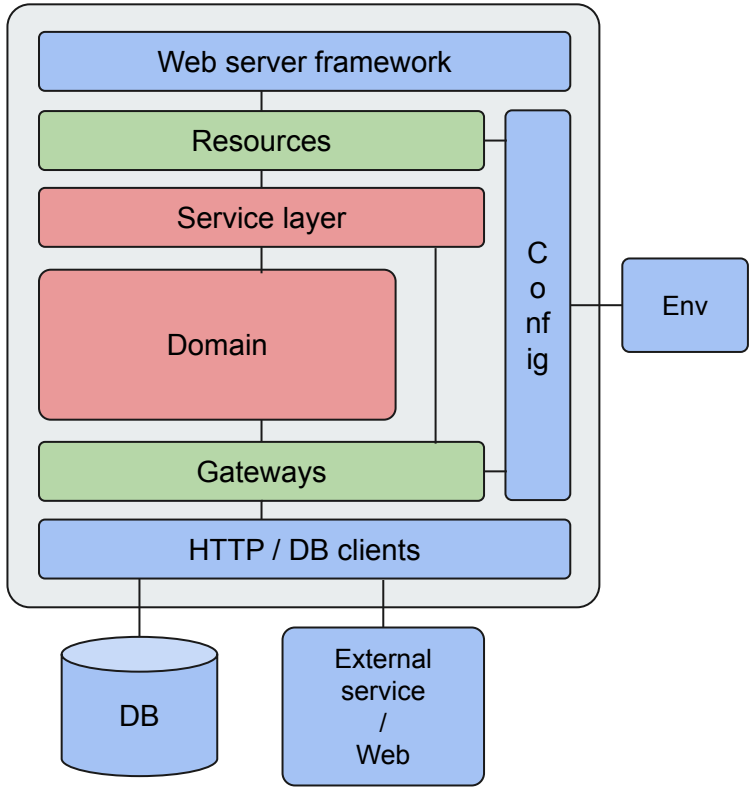



# The anatomy of a microservice: pets example

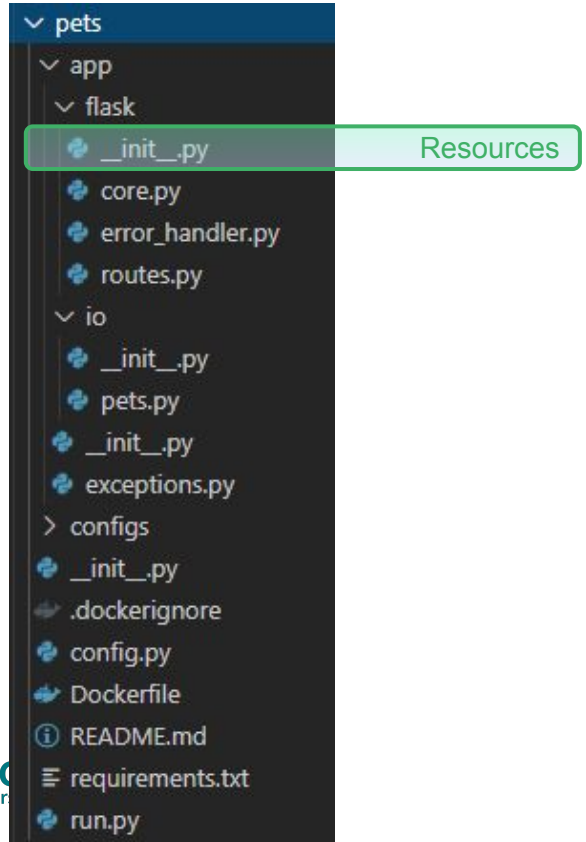


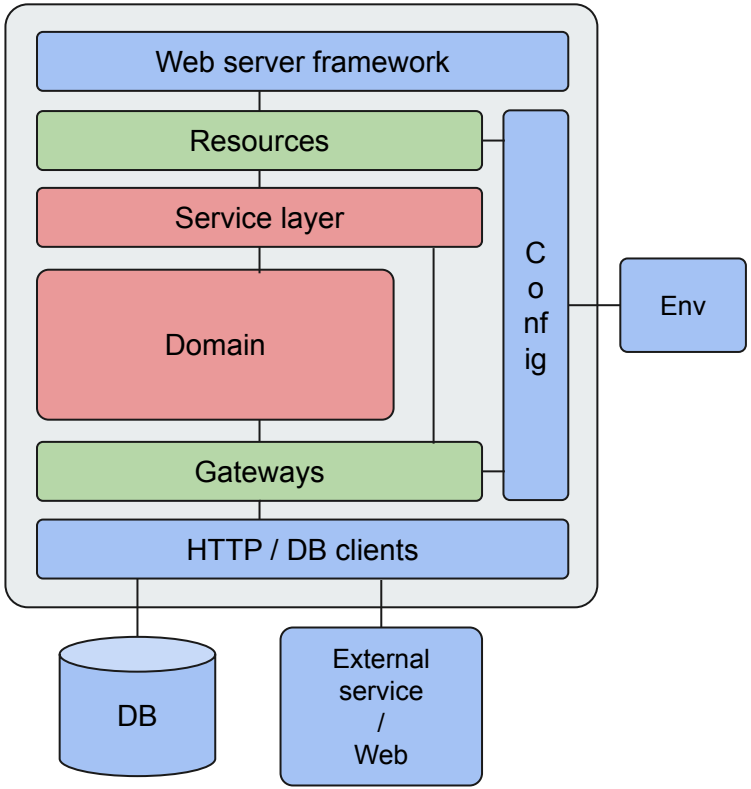

```
20      # Use newer version of swagger ui
21      options = {'swagger_path': swagger_ui_path}
22
23      app = connexion.App(__name__,
24                          specification_dir=config.SPECIFICATION_DIR,
25                          options=options)
26
27      app.add_api('swagger.yaml',
28                  strict_validation=True,
29                  validate_responses=True)
30
31      error_handler.register_error_handlers(app)
```



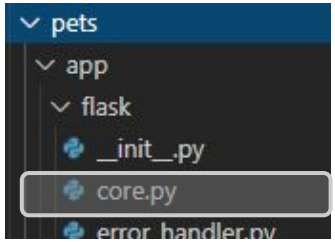



# The anatomy of a microservice: pets example

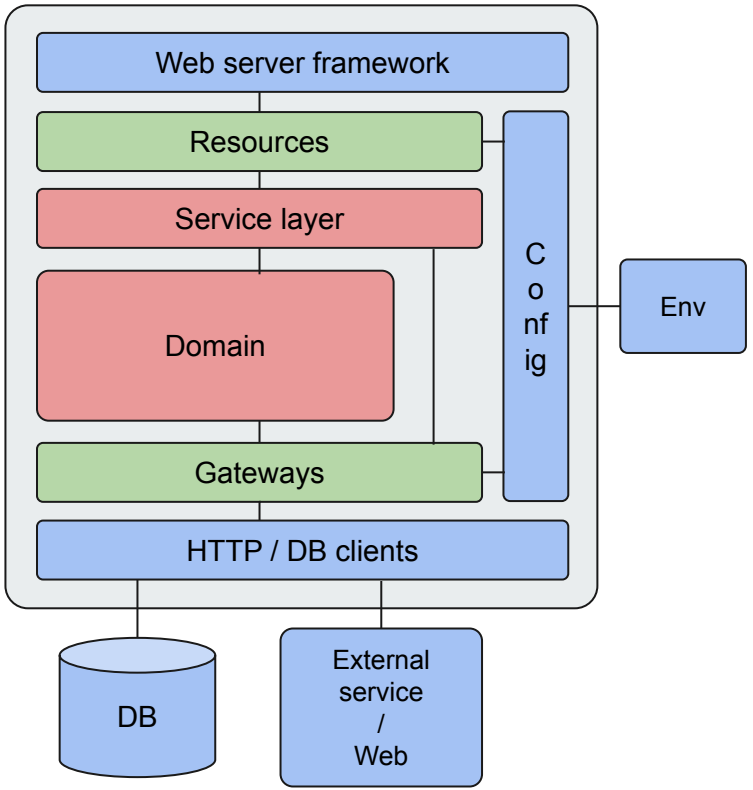


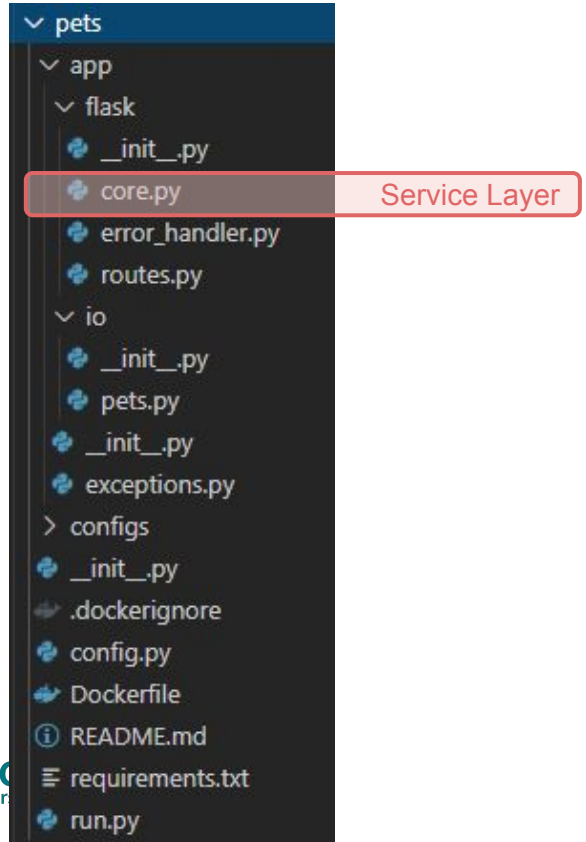
# The anatomy of a microservice: pets example

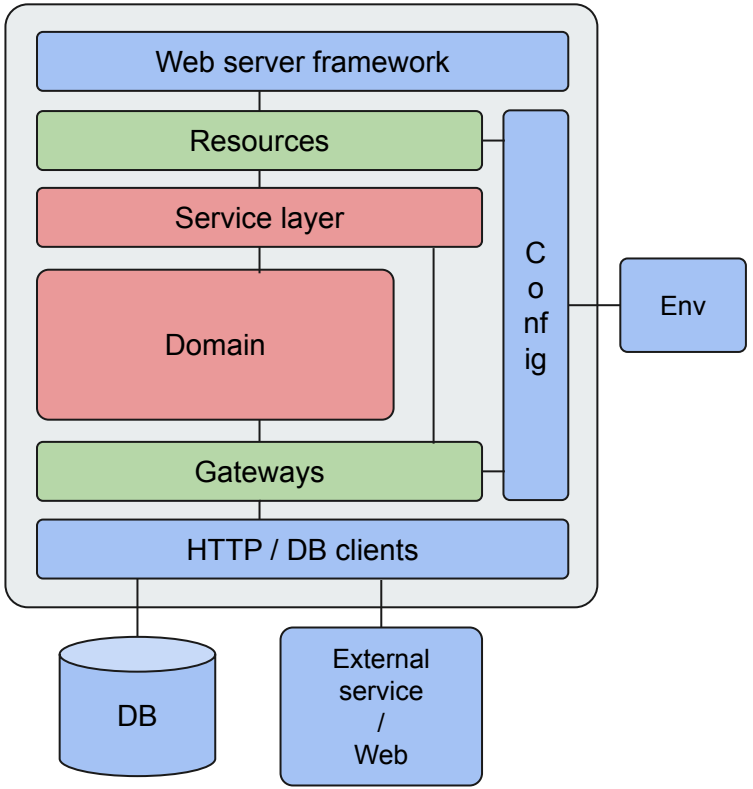



```
7   from app.io import pets
8
9
10  def get_all_pets():
11      """Get array of all pets."""
12      return pets.get_all()
13
14
15  def get_pet_by_id(pet_id):
16      """Get single pet by id."""
17      return pets.get(pet_id)
18
19
20  def add_pet(*, name, owner, description):
21      """Add a new pet."""
22      return pets.create(name=name, owner=owner, description=d
```

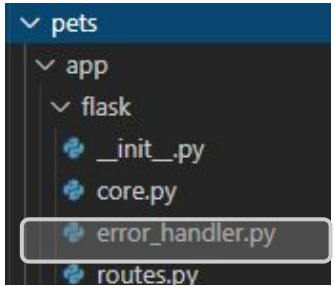


# The anatomy of a microservice: pets example

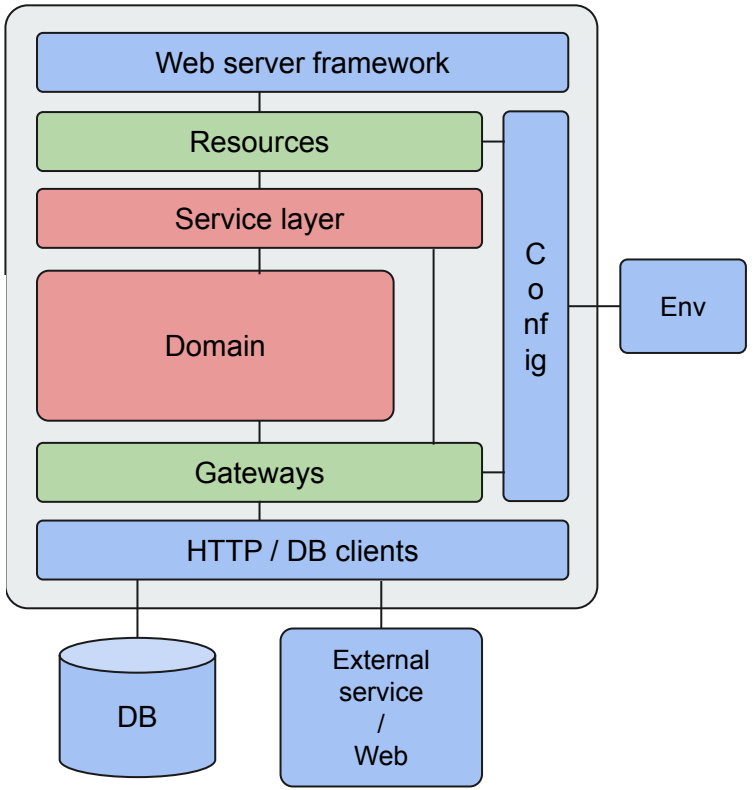


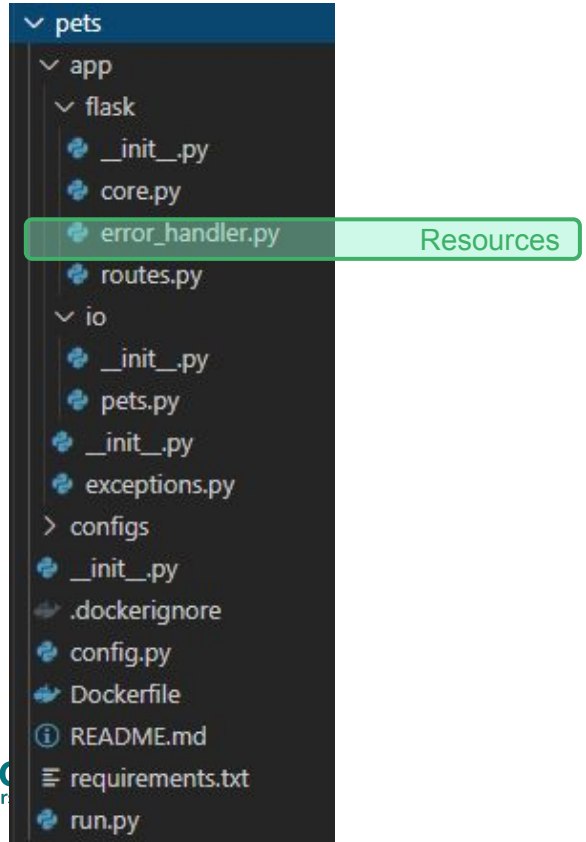
# The anatomy of a microservice: pets example

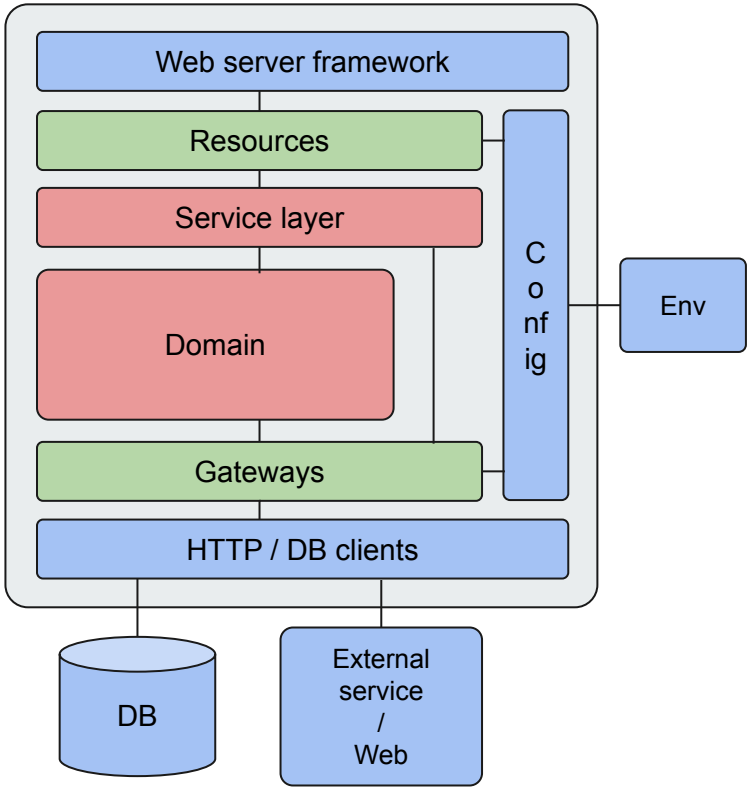



```
8  from flask import jsonify
9
10 from app import exceptions
11
12
13 def handle_custom_error(error):
14     """Handle custom errors."""
15     response = jsonify({'message': error.message})
16     response.status_code = error.status_code
17     return response
18
19
20 def register_error_handlers(app):
21     """Add error handlers to the app."""
22     app.add_error_handler(exceptions.PetNotFound, handle_custo
```

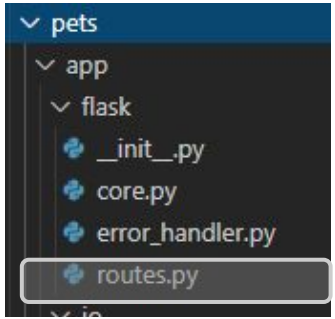


# The anatomy of a microservice: pets example

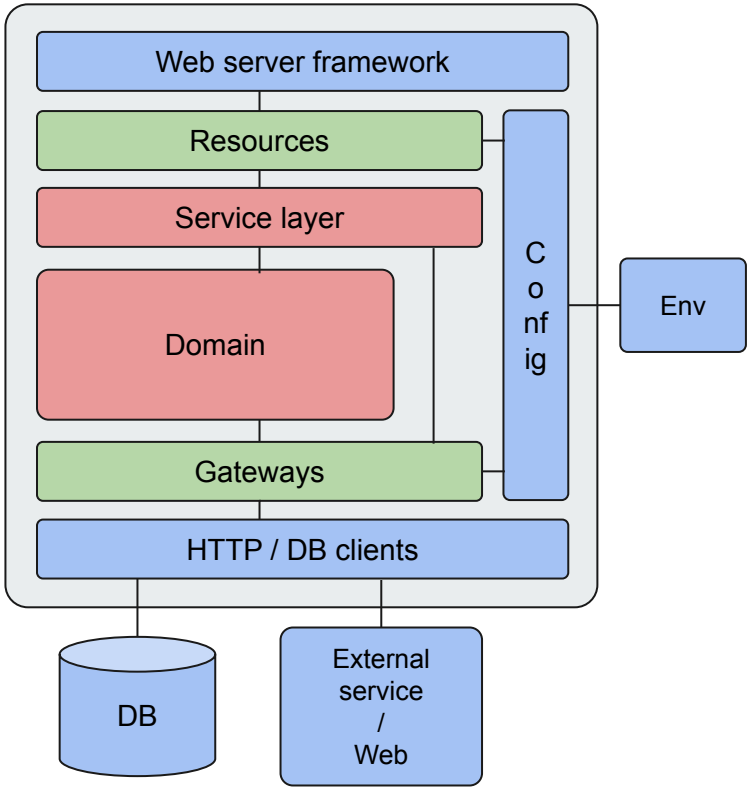


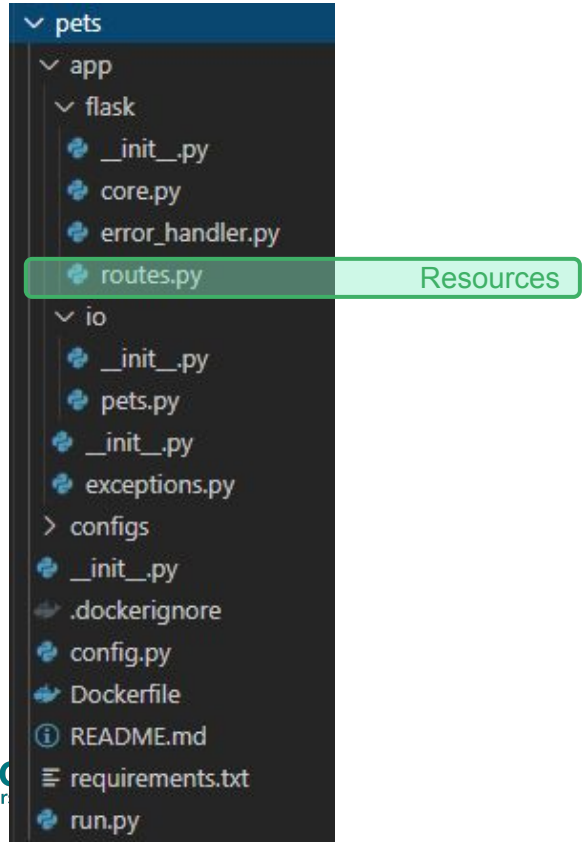
# The anatomy of a microservice: pets example

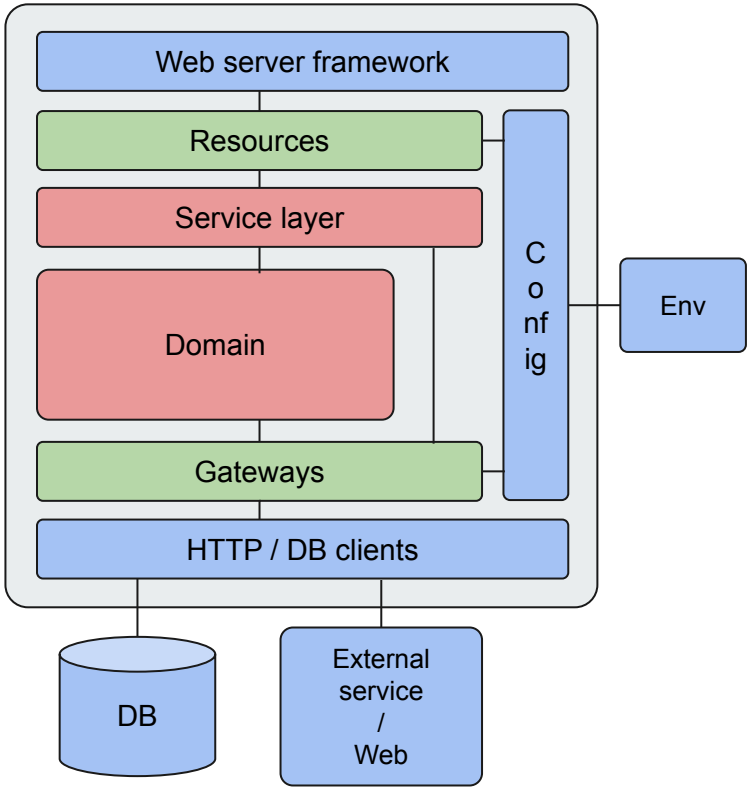



```python
from app.flask import core


class Pet:
    """Includes all HTTP methods for /pets/pet_id>"""

    @staticmethod
    def get(pet_id):
        """Get an existing pet."""
        return core.get_pet_by_id(pet_id), 200

    @staticmethod
    def delete(pet_id):
        """Delete an existing pet."""
        return core.delete_pet_by_id(pet_id), 204
```

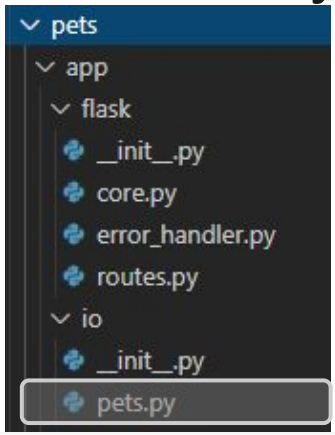


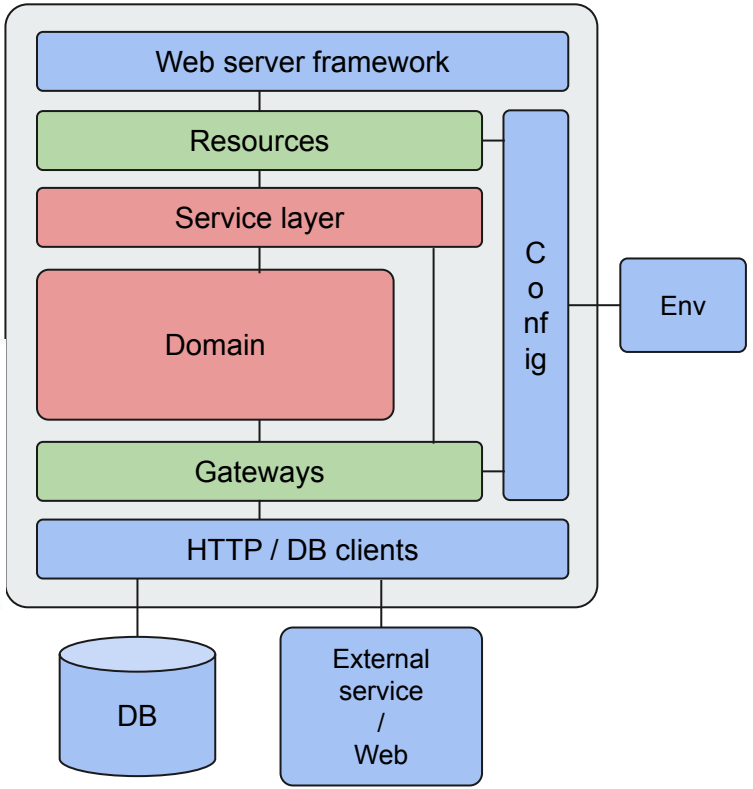
# The anatomy of a microservice: pets example



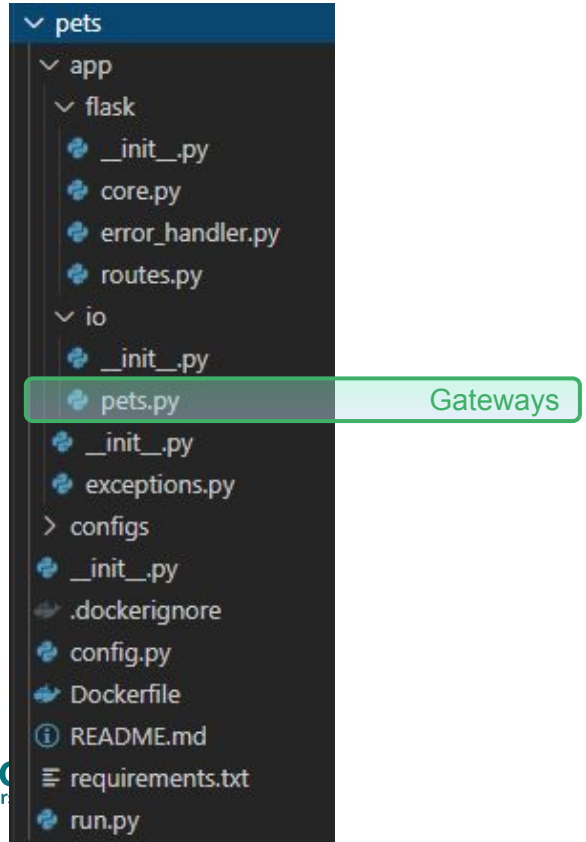
# The anatomy of a microservice: pets example

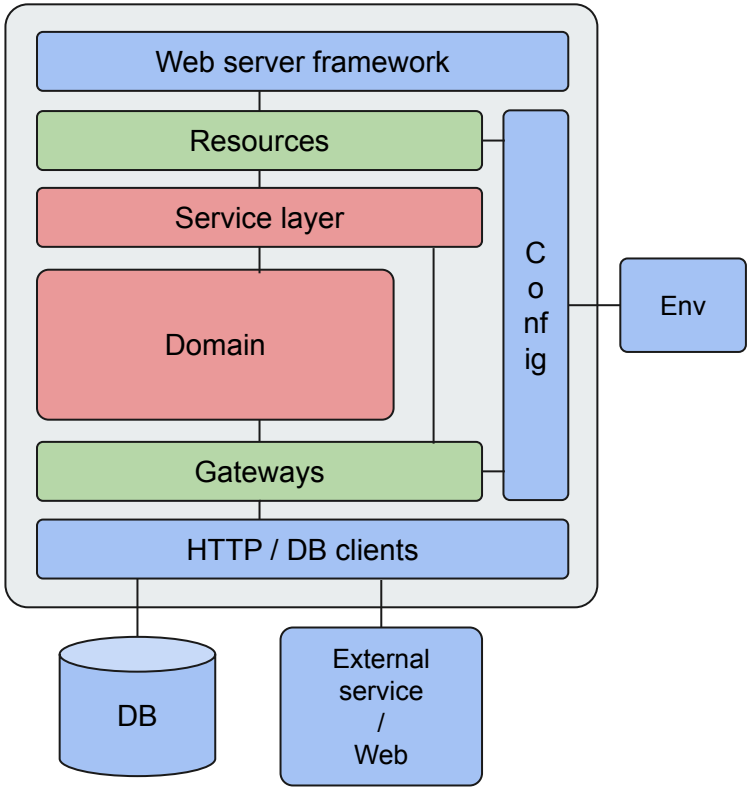


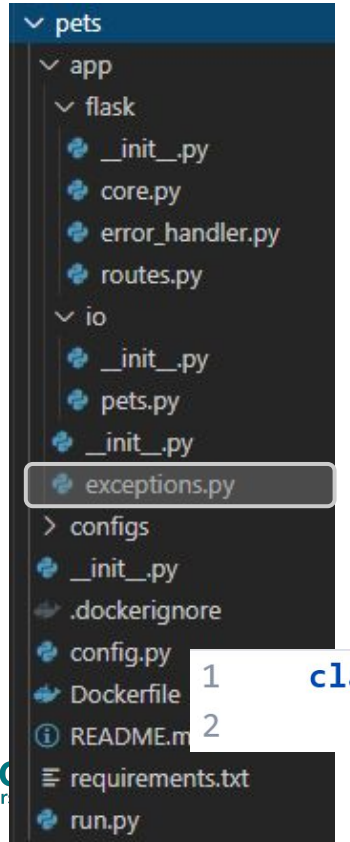

```
11  client = firestore.Client()
12  pets = client.collection('pets')
13
14
15  def get(pet_id: str) -> dict:
16      """Get a pet by id.
17
18      Raises:
19          exceptions.PetNotFound: If no pet with the specified id can be found.
20
21      """
22      doc_ref = pets.document(pet_id)
23      pet = doc_ref.get()
24      if not pet.exists:
25          raise exceptions.PetNotFound('Pet with id {pet_id} was not found.')
26      return doc_ref.get().to_dict()
```

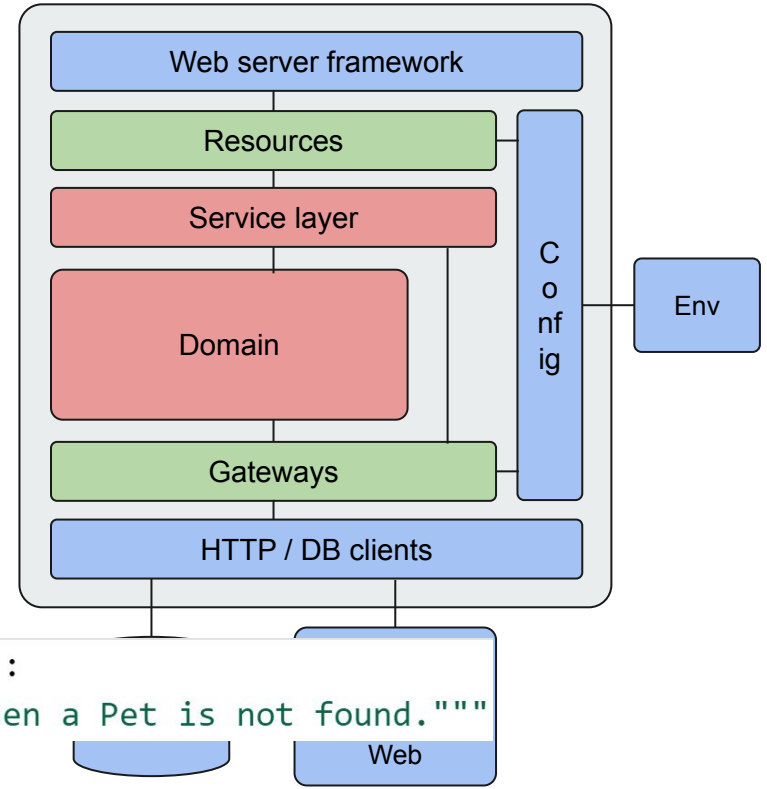
# The anatomy of a microservice: pets example



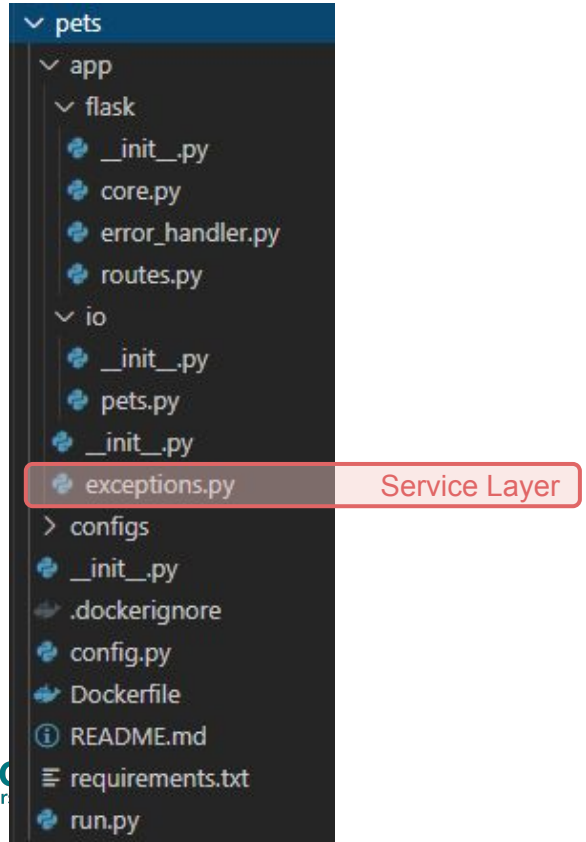
# The anatomy of a microservice: pets example

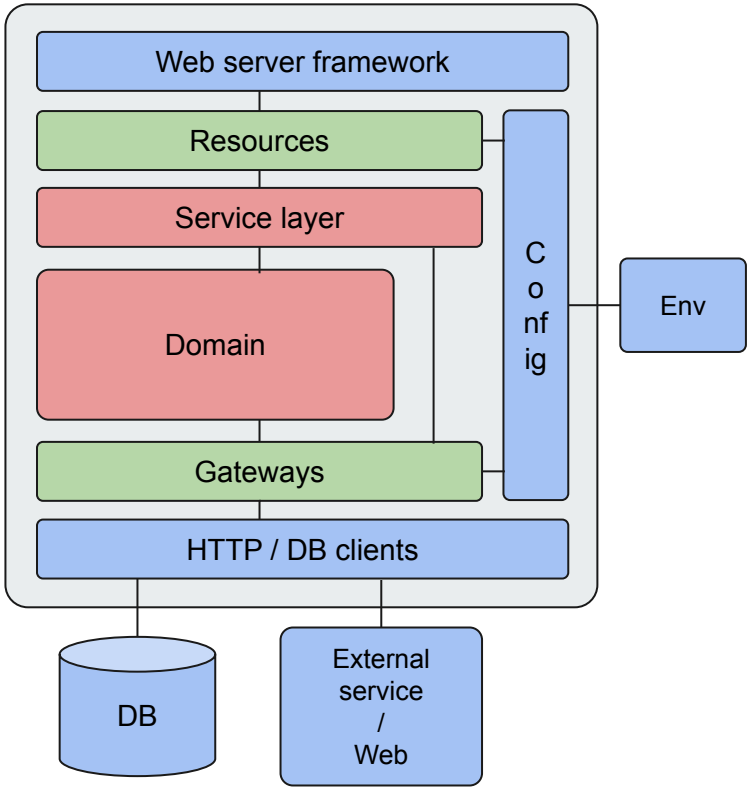


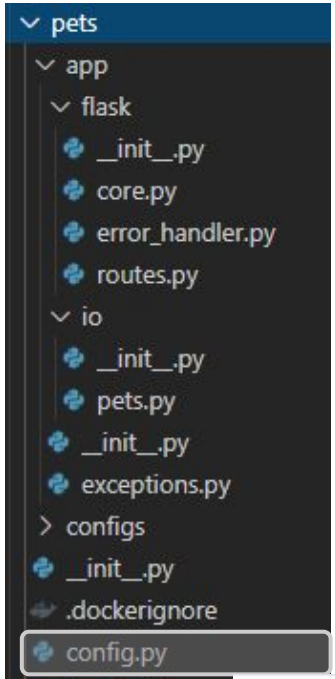

```python
class PetNotFound(Exception):
    """Exception to raise when a Pet is not found."""
```

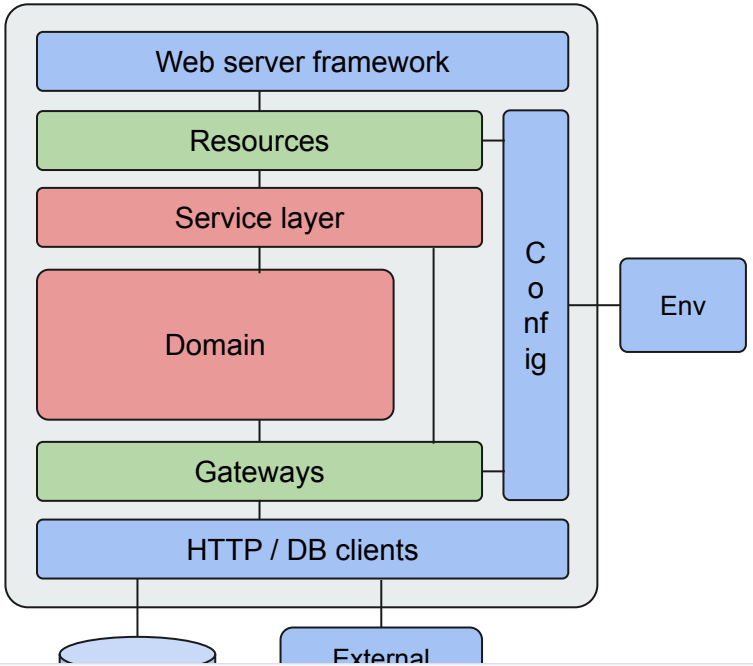
# The anatomy of a microservice: pets example



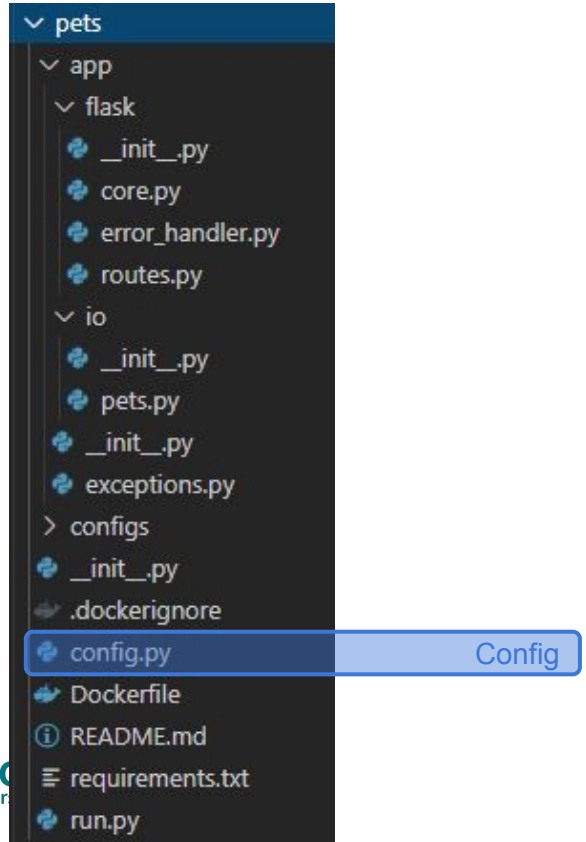
# The anatomy of a microservice: pets example

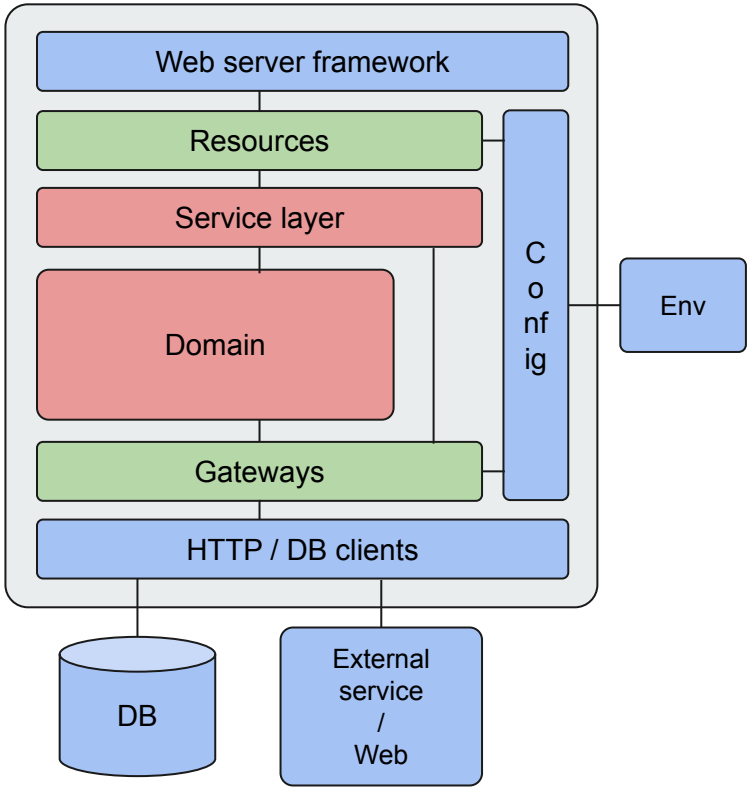


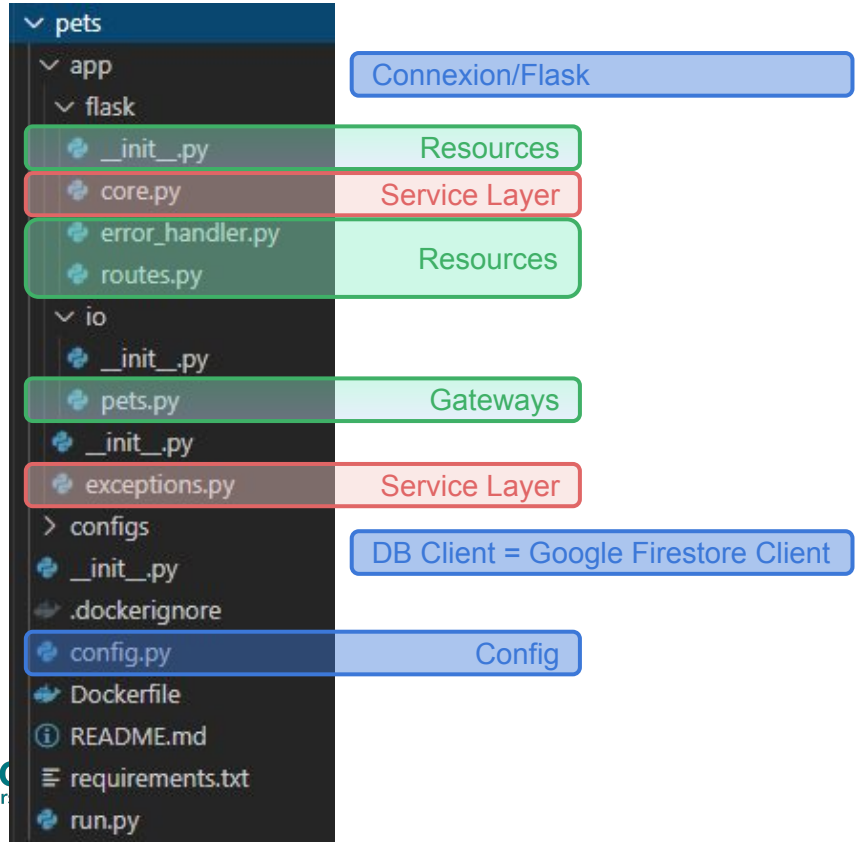

```python
import os

SPECIFICATION_DIR = os.path.join(os.path.dirname(os.path.abspath(__file__)),
                                 'configs')
```

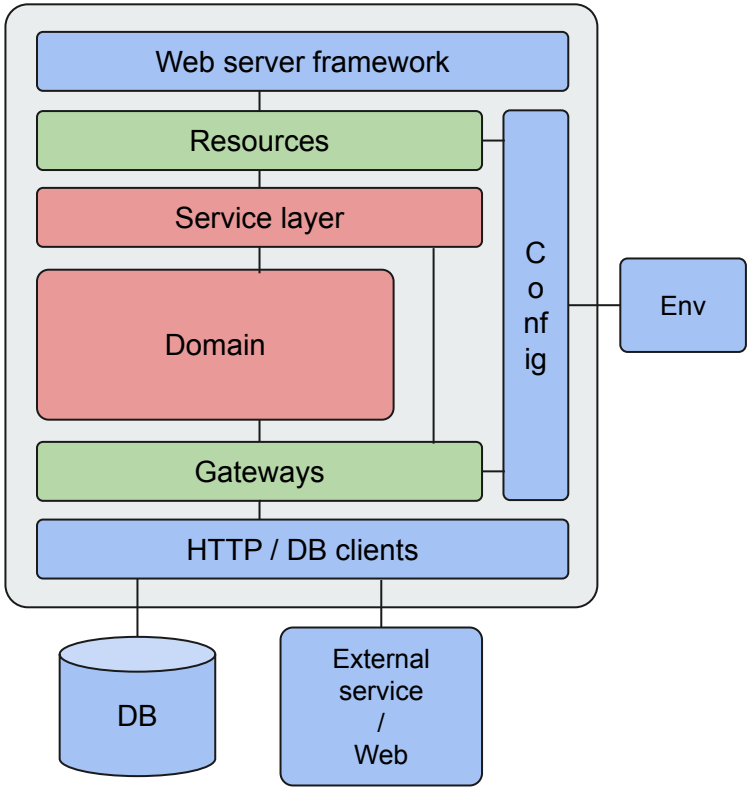
# The anatomy of a microservice: pets example



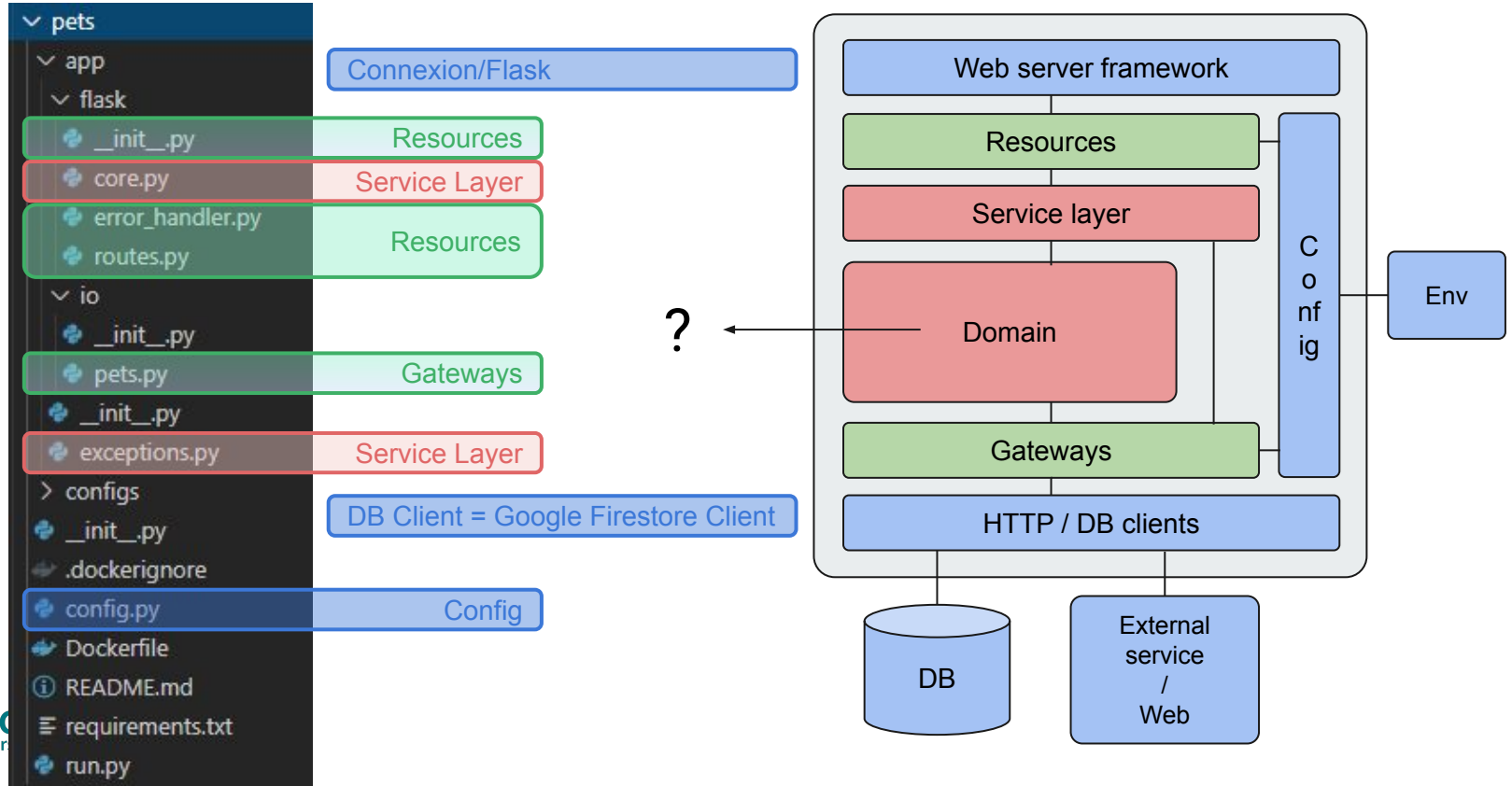



# The anatomy of a microservice: pets example

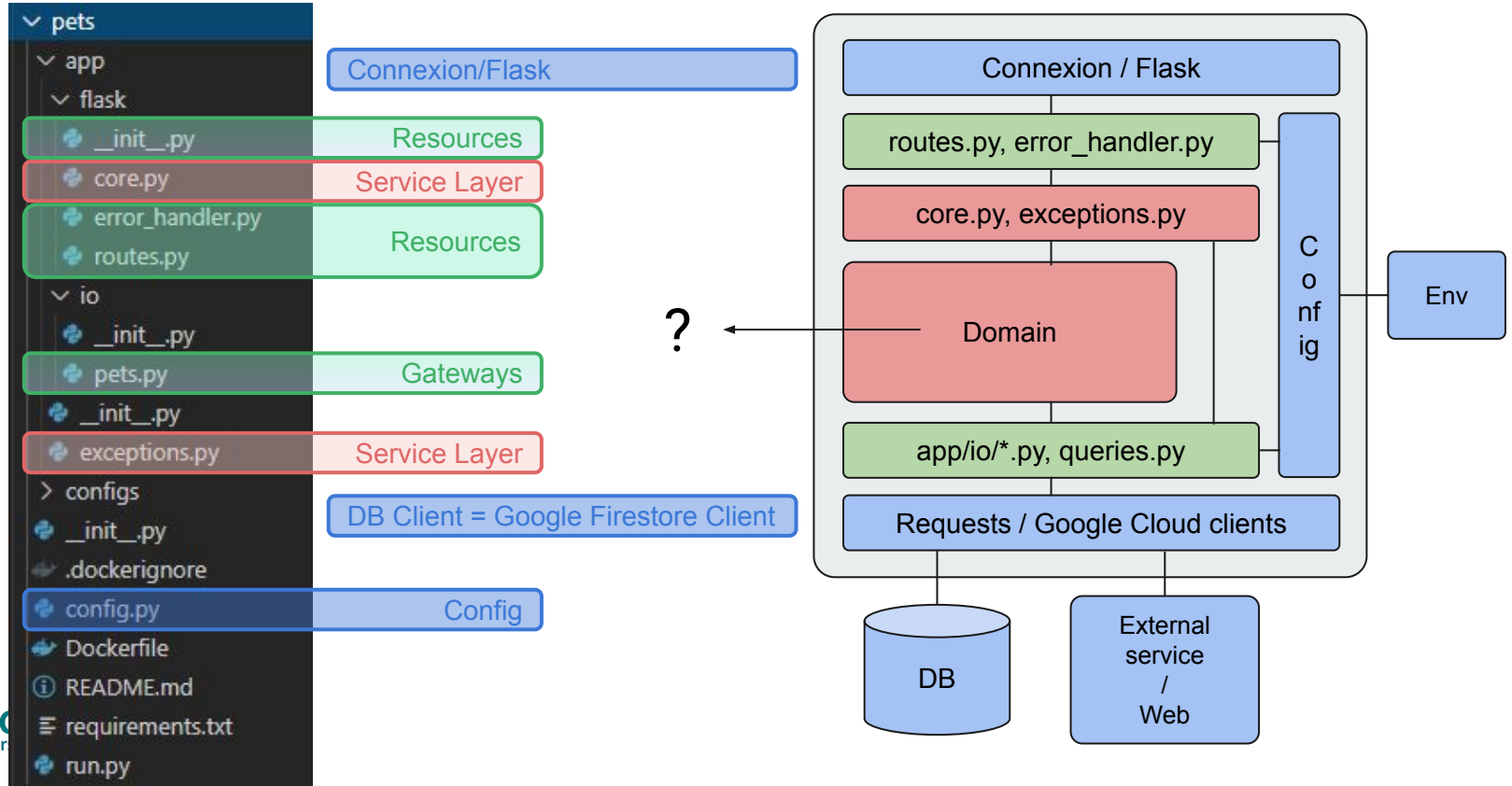


# The anatomy of a microservice: pets example



# The anatomy of a microservice: pets example



# Wrap-up

# Demo: Kubernetes

# Agenda

## What will we talk about today

**Lecture**

1. Microservice Architecture
2. Kubernetes (k8s)
3. Serverless

**Directed work**

4. Serverless deployment on Cloud Run (k8s)

**Lecture**

5. Anatomy of a microservice

**Demo**

6. Kubernetes

# Lecture summary

| Topic | Concepts | To know for… | |
|---|---|---|---|
| | | Project | Exam |
| Microservice | • Microservice Architecture<br>• Kubernetes (k8s)<br>• Serverless | | Yes |
| Lab: Serverless API deployment on Cloud Run | | Yes | |
| Microservice | • Anatomy of a microservice | Maybe | |
| Demo: Kubernetes | | | |

# Project objective for sprint 3

| # | Week | Work package | Requirement |
|---|---|---|---|
| 3.1 | W05 | Build an **API** to **serve your model** and any extra logic that is needed to serve it (e.g. using Flask).<br>You should be able to run the API locally. | Required |
| 3.2 | W05 | Package your model serving API in a **Docker container**.<br>This too should be run locally. | Required |
| 3.3 | W06 | **Deploy** your model serving API in the Cloud. You should be able to call your model to generate new predictions from another machine.<br><br>**Attention**: This can incur Cloud **costs**. Make sure to use a platform where you have credits and not burn through them. You can ask for support from the teaching staff in that regard. | Required |

# That’s it for today!


LIÈGE université

# Monoliths architecture

# Credits where credits are due


**Jasper Van den Bossche**
Software Engineer @ ML6

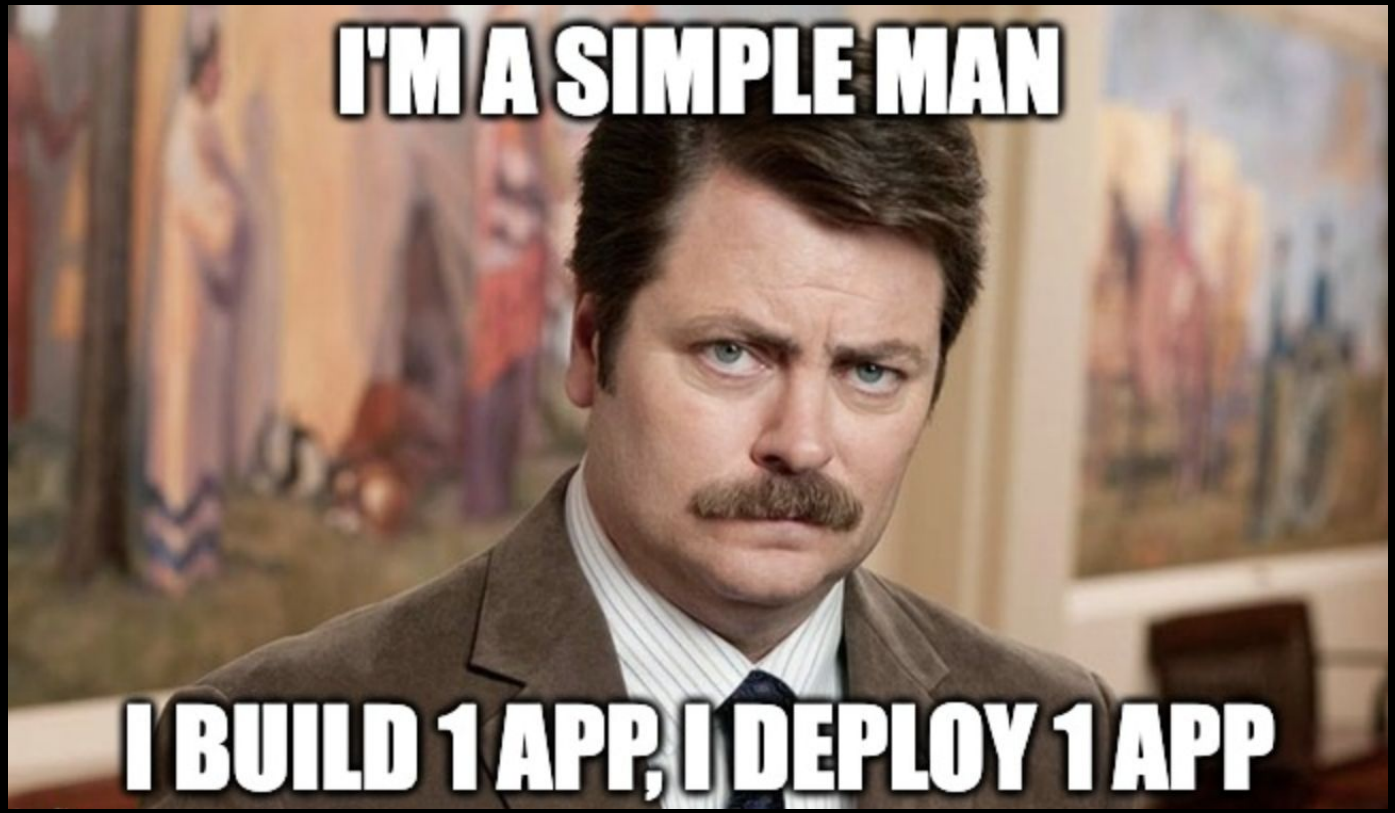
# Credits where credits are due


**Jasper Van den Bossche**
Software Engineer @ ML6

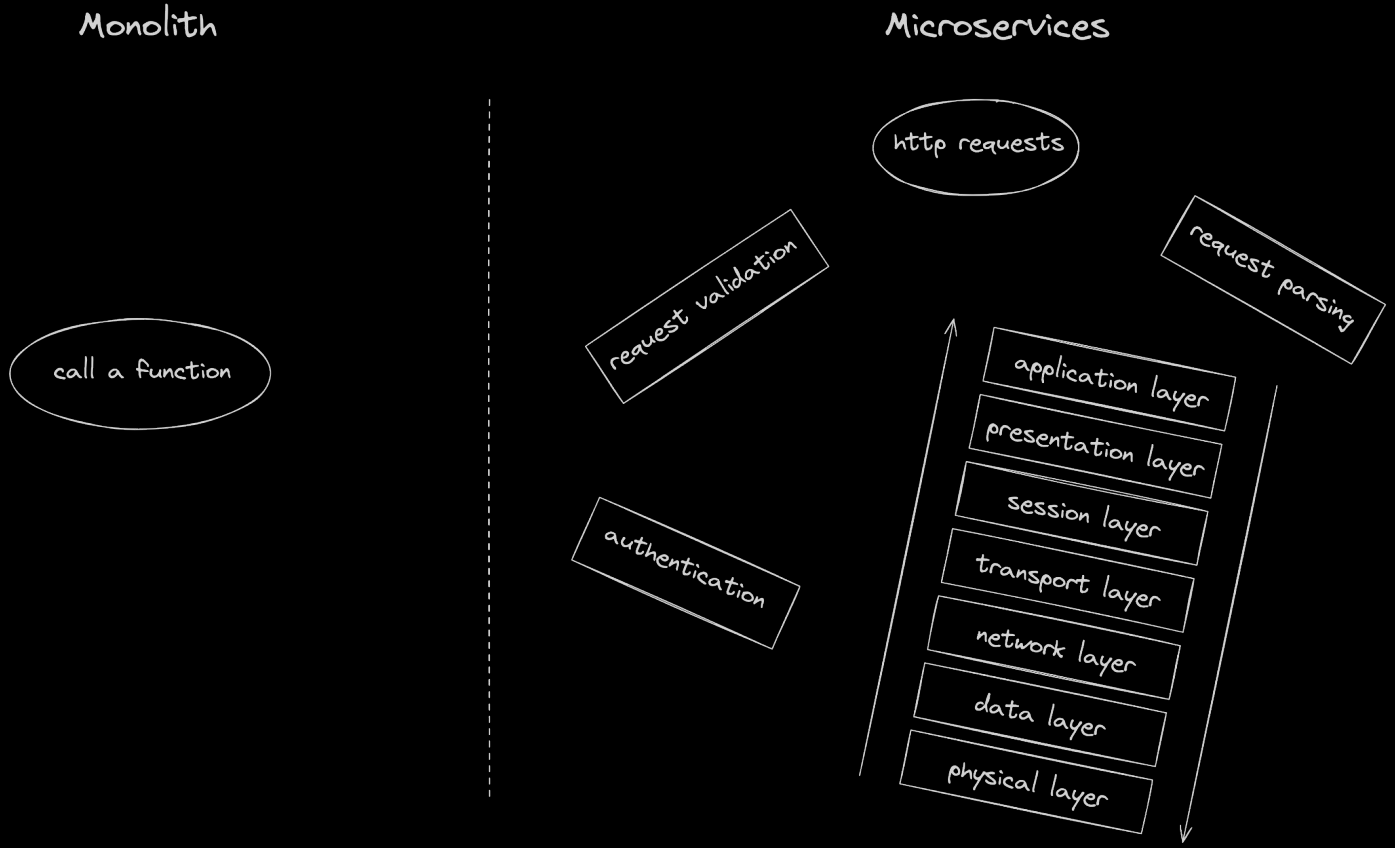
*(Disclaimer: Good sense of humour...)*

# NOT the Goal of this Talk

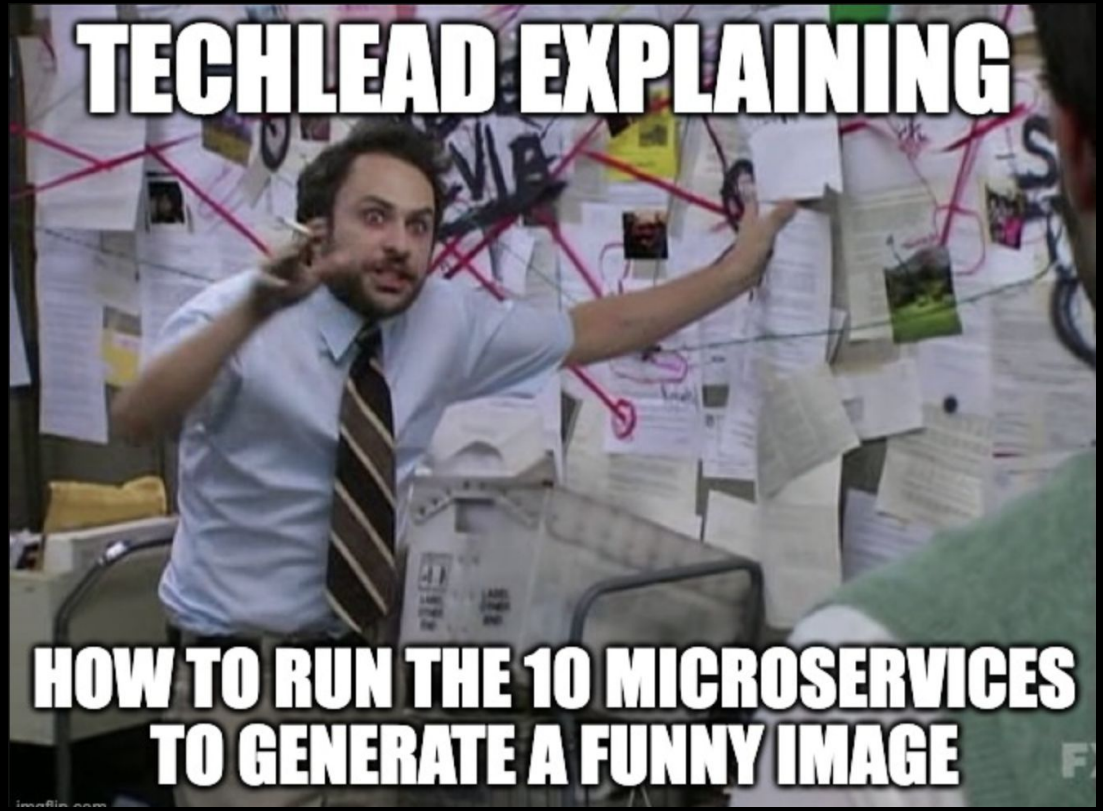
# Simplicity



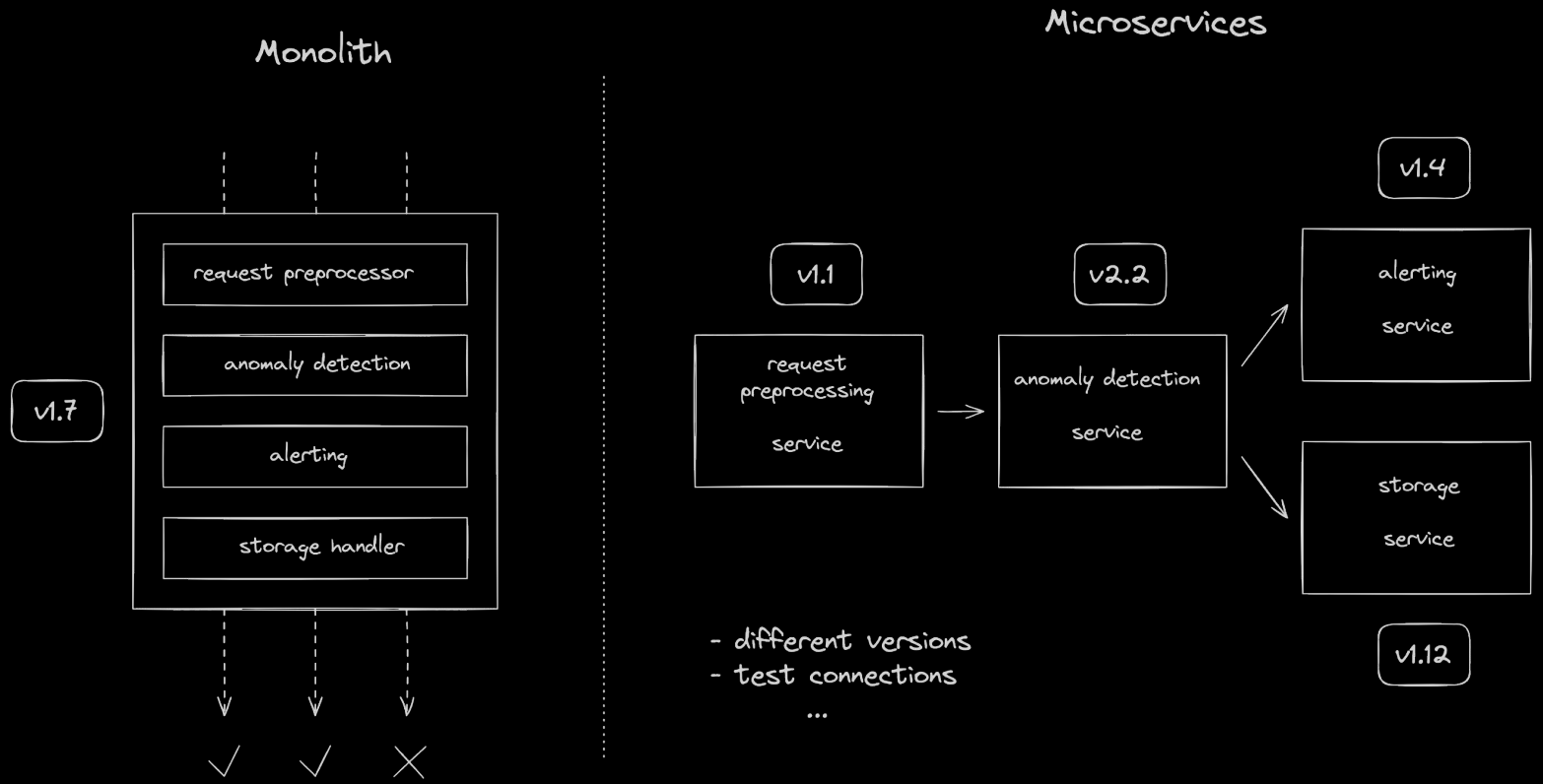
# Performance



# End-to-End Testing

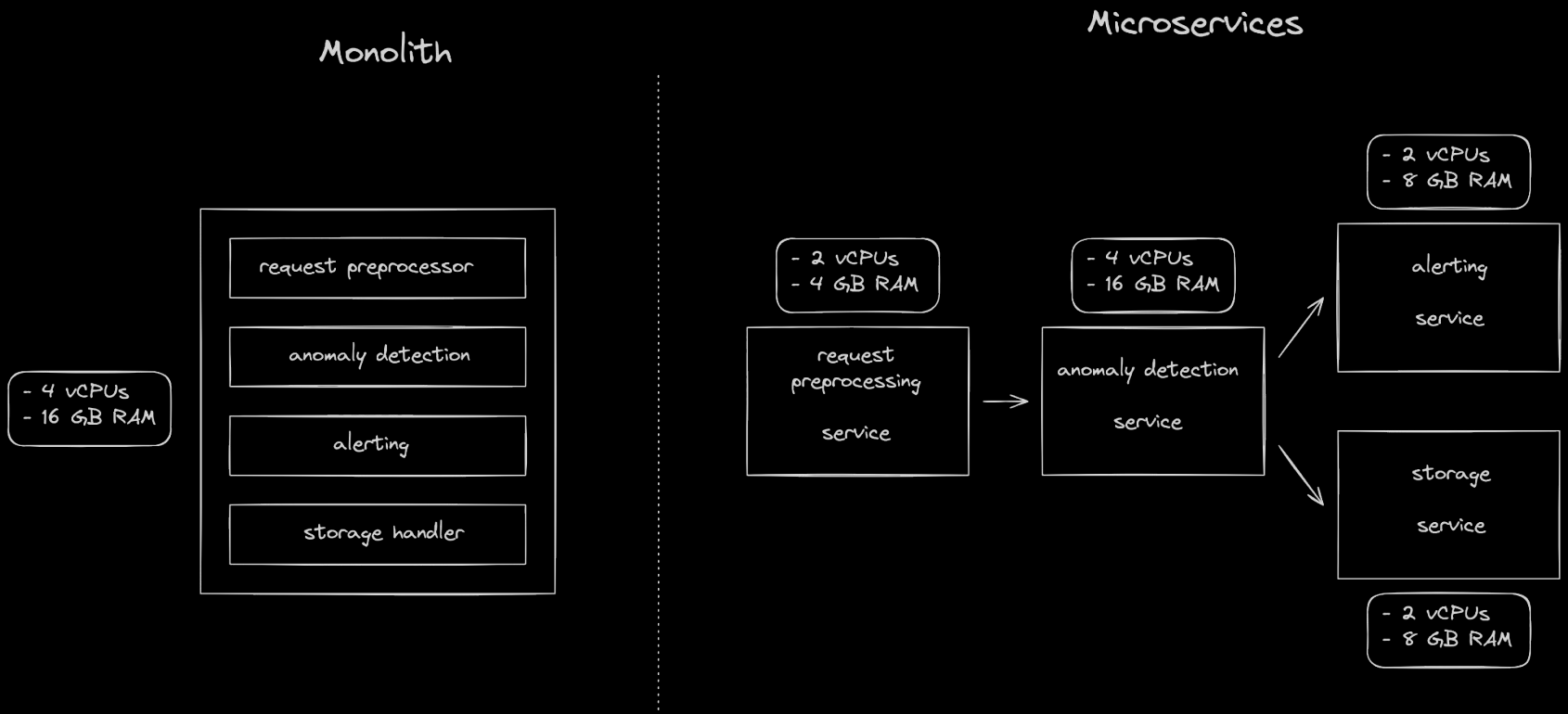


# End-to-End Testing

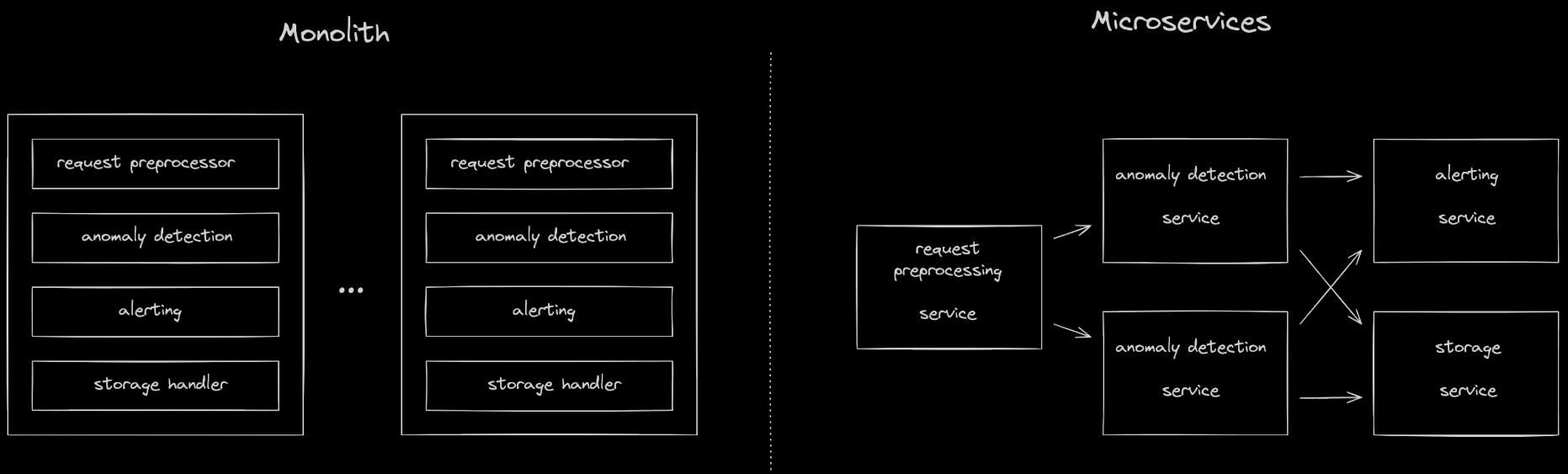


# The not so good stuff

# Scalability & Resource Utilization

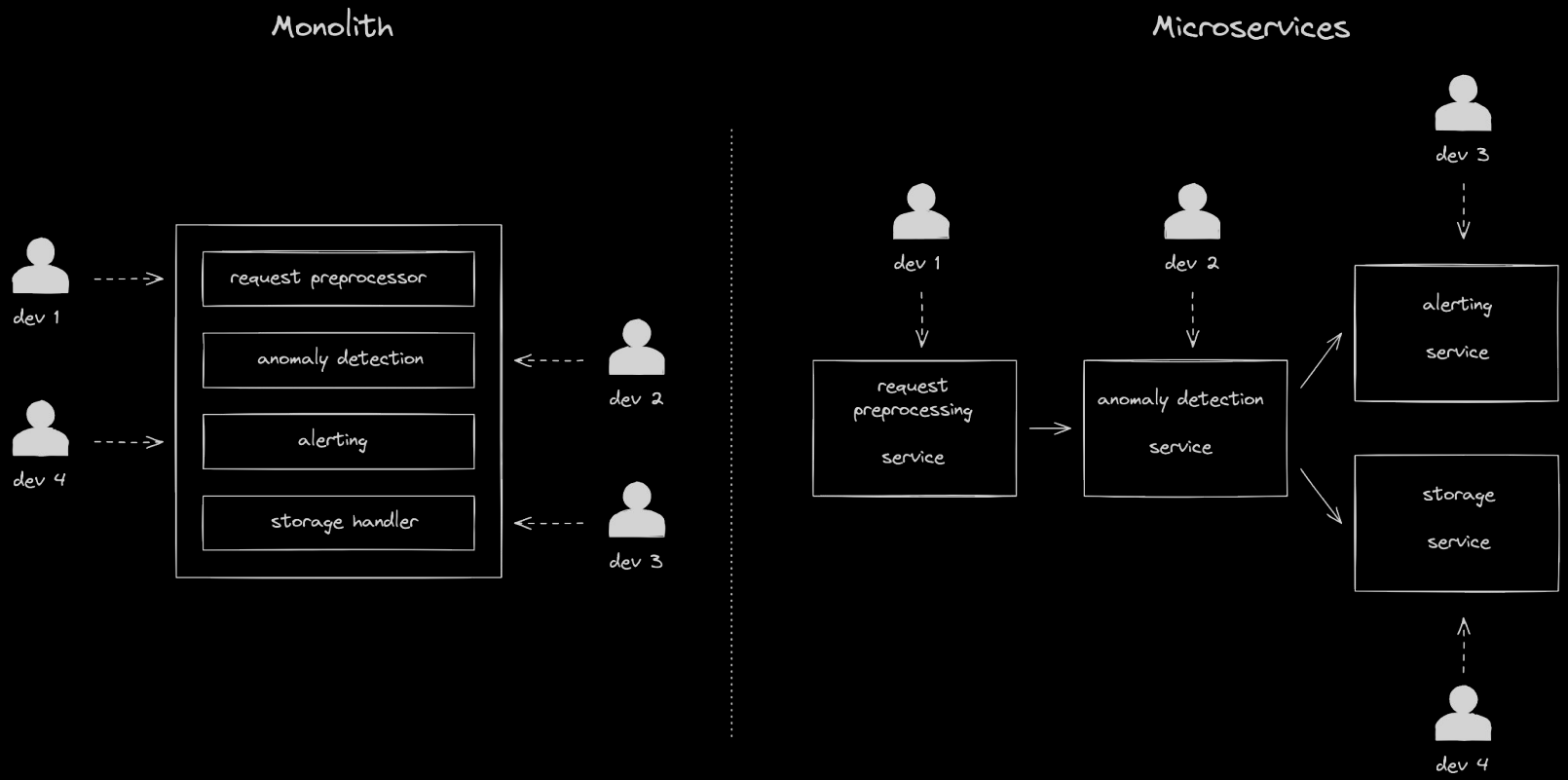


# Scalability & Resource Utilization



# Teamwork makes the dream work

# Independent development & ease of maintenance

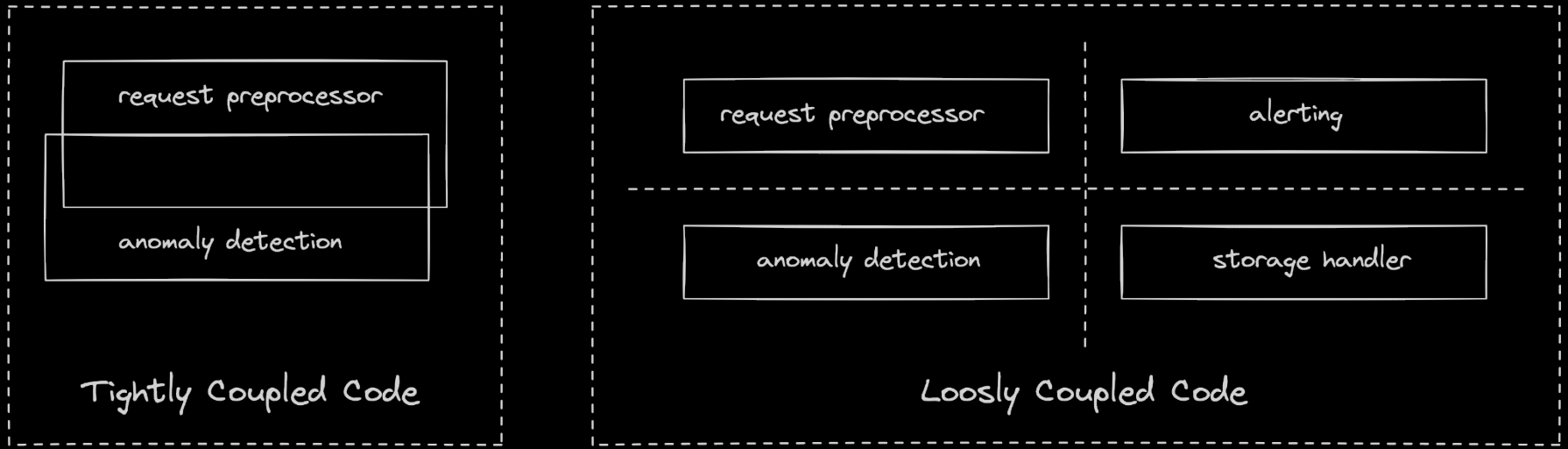


# *First Day back on the Job after a Holiday*



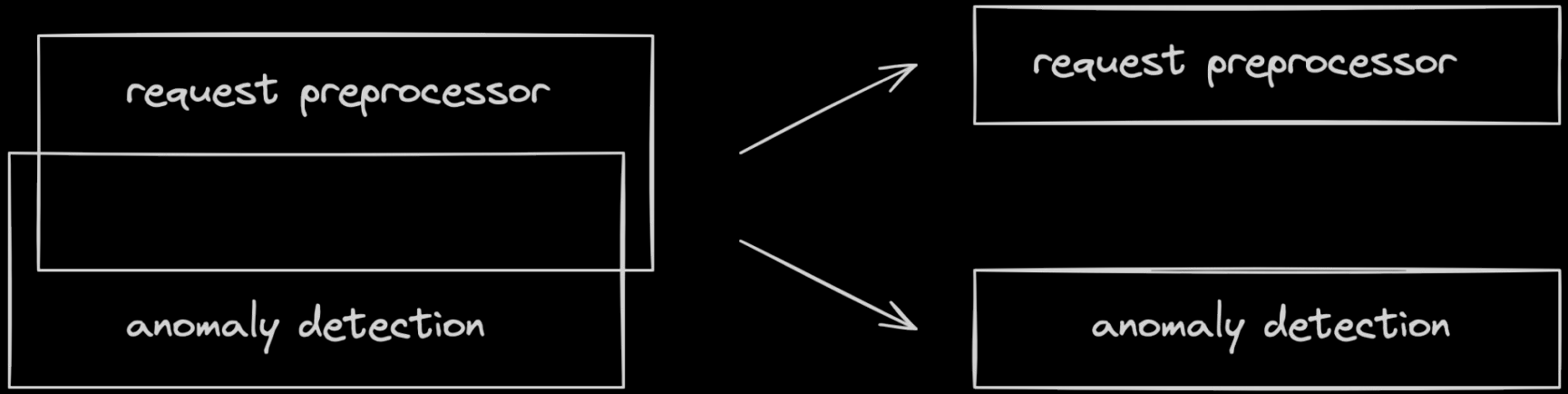
 

# Unwieldy codebases

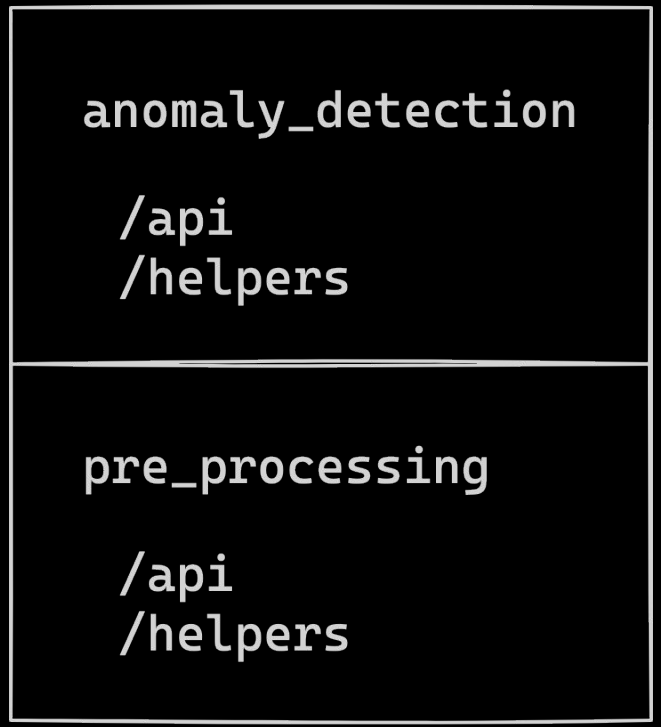


# But how can we avoid this?

# Think of your code as independent packages



# APIs to the Rescue



# Conclusion: Monoliths & Microservices

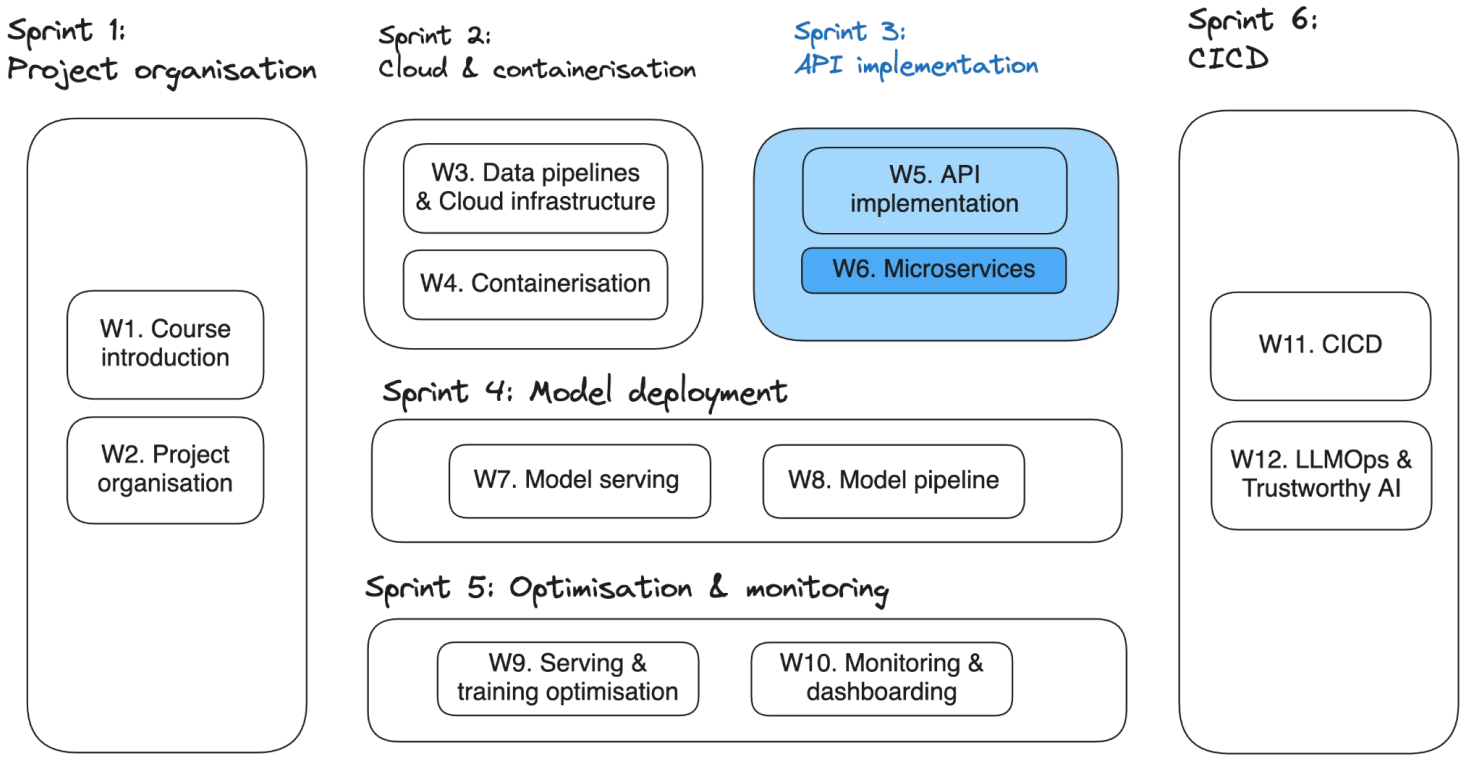
- Less infrastructure to manage
- Better performance
- Easier to run and test

BUT

- Require discipline

**THERE IS NO ARCHITECTURE TO RULE THEM ALL**

# Status on our overall course roadmap



# Why use kubernetes: Abstracting your infrastructure

Kubernetes allows to build, deploy, and manage your application in a way that is portable across a wide variety of environments. The move to application-oriented container APIs like Kubernetes has three concrete benefits:

- **Separation**: developers from specific machines
- **Portability**: simply a matter of sending the declarative config to a new cluster
- **Customisation**: simply a matter of sending the declarative config to a new cluster