LIÈGE université

API

Sprint 3 | Week 5

INFO9023 - ML Systems Design

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

Agenda

What will we talk about today

Lecture (1h15)

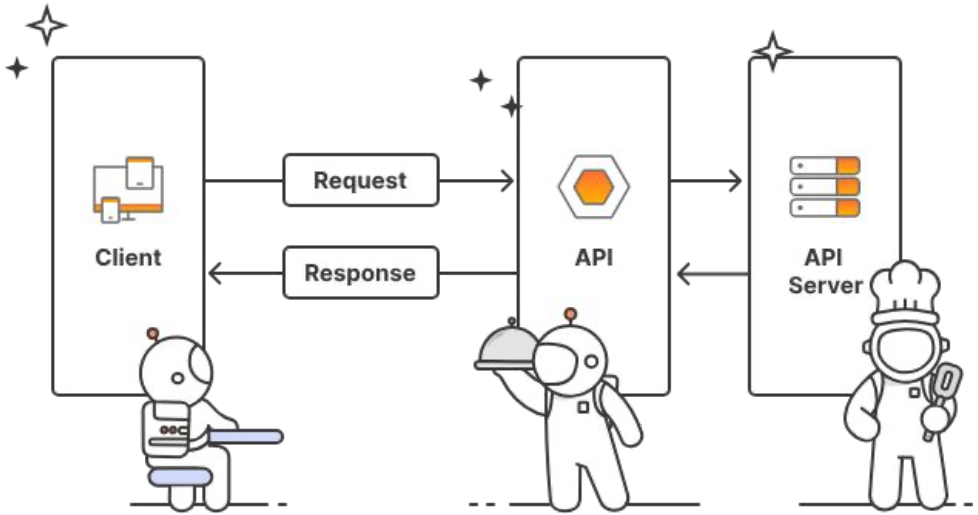
1. API
    - REST
    - gRPC
    - Websocket
2. API Gateway

Lab (45min)

3. Flask

API

Application Programming Interface (API)



An **Application Programming Interface (API)** is a set of protocols that services to *communicate* and *transfer data*.

Client sends a **requests** and the API sends a **response**.

https://www.postman.com/what-is-an-api/


API
LIGHTYEAR
API EVERYWHERE
imgflip.com

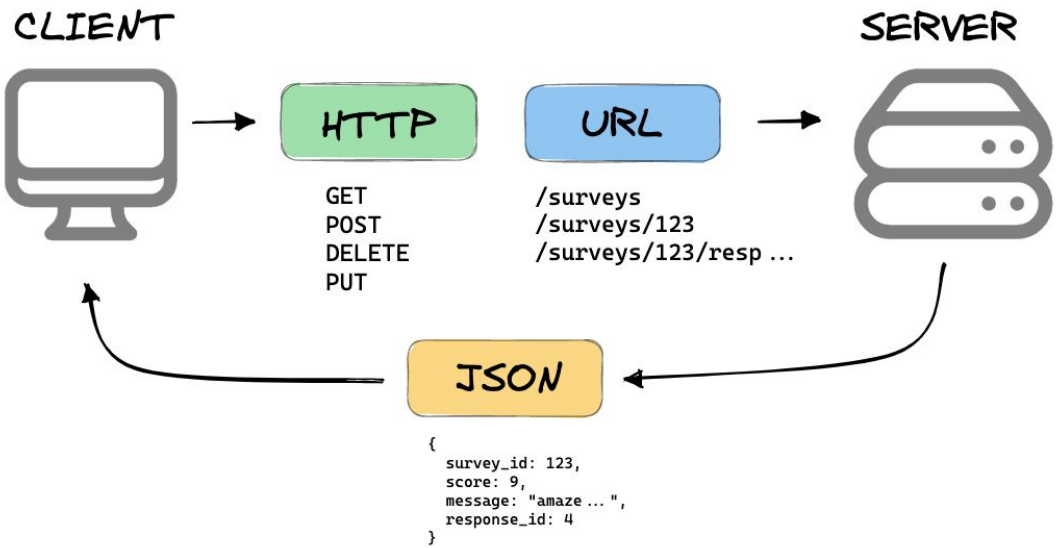
Types of APIs covered in this course

- **REST** (Representational State Transfer): The client sends requests to the server. The server returns a response (output data) back to the client.
- **RPC** (Remote Procedure Calls). The client completes a function (aka method or procedure) on the server, and the server sends the output back to the client. “Action centric”.
- **Websocket**: WebSocket API supports two-way communication between client apps and the server. The server can send callback messages to connected clients, which is not possible in REST API.

REST

REpresentational State Transfer (REST) API

Roy Fielding's PhD thesis: [Architectural Styles and the Design of Network-based Software Architectures]



https://mannhowie.com/rest-api

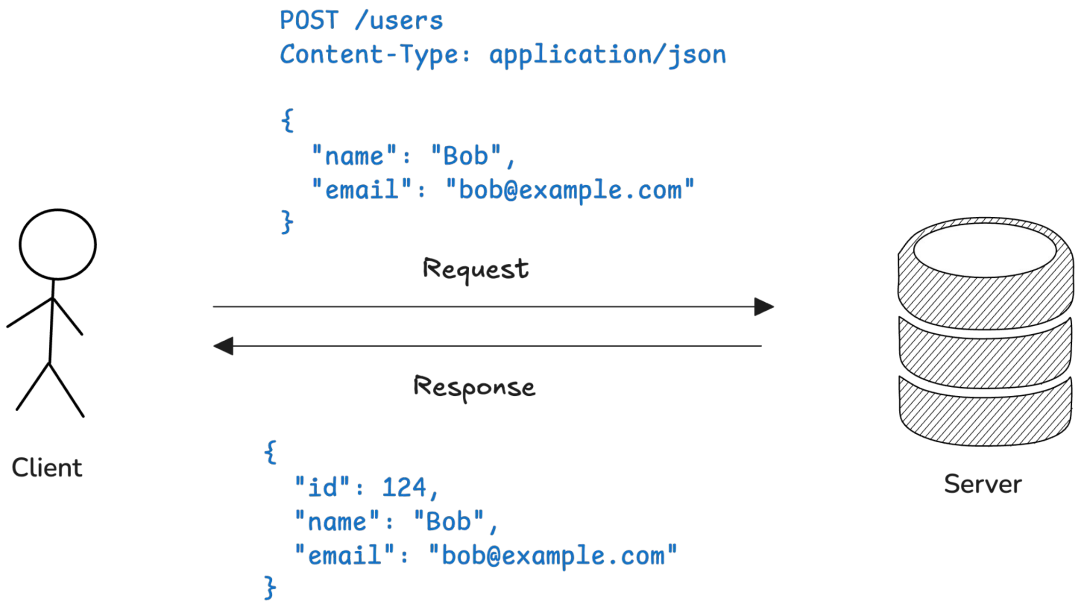
REpresentational State Transfer (REST) API

Retrieving information from server (GET)

Analogy: “Asking the waiter information about the menu at a restaurant.”

REpresentational State Transfer (REST) API

Submits data to the server to create a new resource (POST)



Analogy: “Ordering a new dish at a restaurant and getting it served.”

REpresentational State Transfer (REST) API

Replaces an existing resource with new data (PUT)

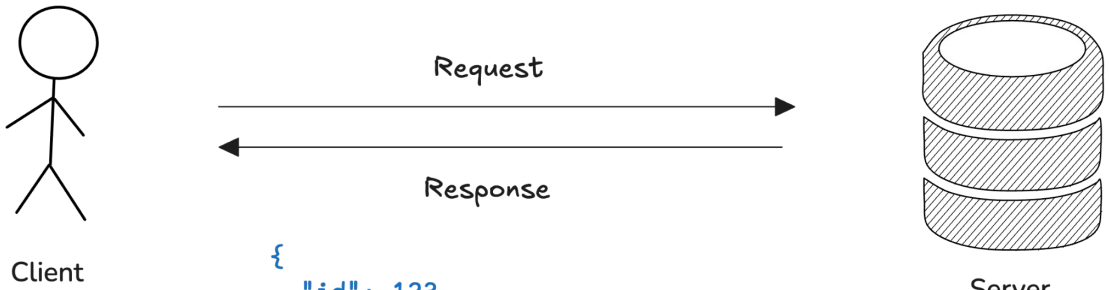
Analogy: “Changing your entire meal order at a restaurant after placing it.”

REpresentational State Transfer (REST) API

Partially updates an existing resource (PATCH)

```
PATCH /users/123
Content-Type: application/json

{
  "email": "newalice@example.com"
}
```

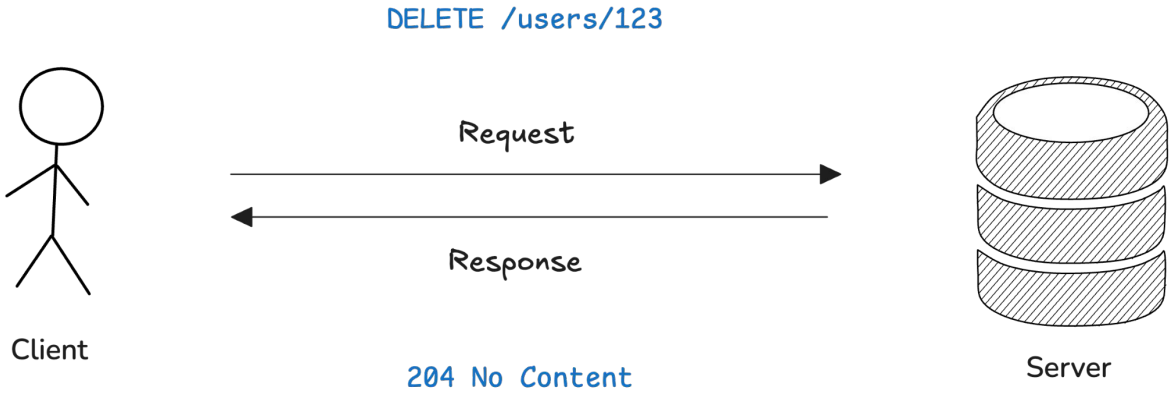



```
{
  "id": 123,
  "name": "Alice Updated",
  "email": "newalice@example.com"
}
```

Analogy: “Asking the waiter to add extra cheese to your existing order.”

REpresentational State Transfer (REST) API

Deletes an existing resource from the server (DELETE)



Analogy: “Canceling your order and removing it from the kitchen.”

HTTP requests

HTTP is a communication protocol for networked systems

REST is based on **CRUD** operations: Able to **Create**, **Read**, **Update** and **Delete** resources

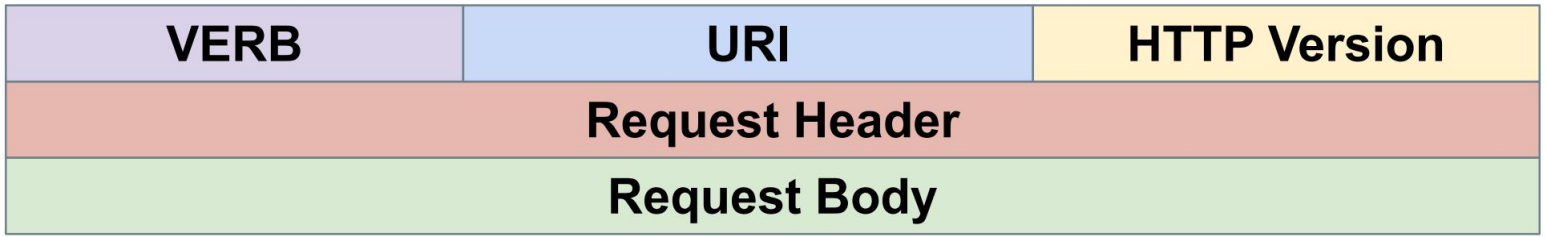
There are 4 basic **verb** commands when making a HTTP request

- **POST** (*Create):* a new resource based on the payload given in the body of the request
- **GET** (*Read):* data from a single or a list of multiple resources.
- **PUT/PATCH** (*Update):* a specific resource (by ID)
- **DELETE** (*Delete):* the given resource based on the id provided

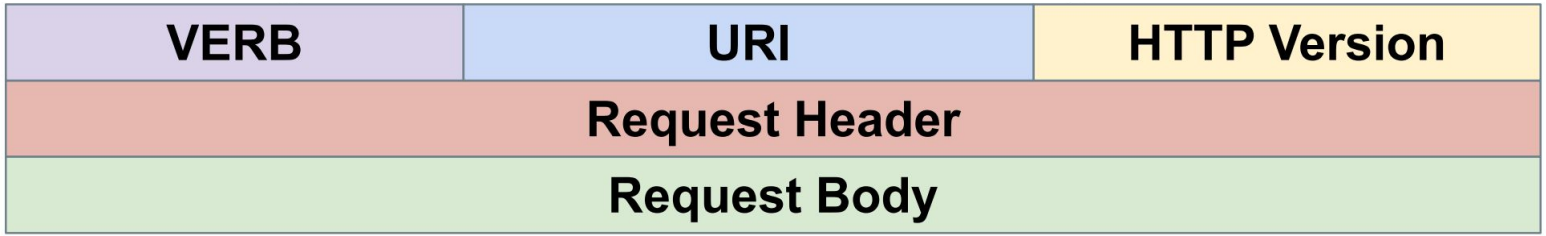
https://mannhowie.com/rest-api

Anatomy of an HTTP Request

- **Verb** (one of PUT, GET, POST, DELETE, …)
- **Endpoint (URI)** Identifies the resource upon which the operation will be performed (e.g. `/users/123`)
- **HTTP Version** (e.g. HTTP/1.1)
- **Request Header**:
  - Collection of key-value pairs of headers and their values.
  - Information about the **message** and its **sender** like client type, authentication, formats client supports, format type of the message body, cache settings, …
    - E.g. "*Content-Type: application/json*"
- **Request Body** is the actual message content (JSON, XML)



Anatomy of an HTTP Request



GET:

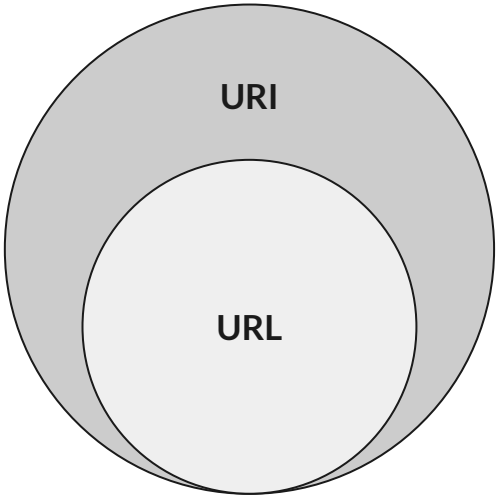
```
GET http://www.w3.org/Protocols/rfc2616/rfc2616.html HTTP/1.1
Host: www.w3.org, Accept: text/html,application/xhtml+xml,application/xml; …,
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 …,
Accept-Encoding: gzip,deflate,sdch, Accept-Language: en-US,en;q=0.8,hi;q=0.6
```

POST:

```
POST http://MyService/Person/ HTTP/1.1
Host: MyService, Content-Type: text/xml; charset=utf-8, Content-Length: 123
<?xml version="1.0" encoding="utf-8"?>
<Person><ID>1</ID><Name>M Vaqqas</Name>
<Email>m.vaqqas@gmail.com</Email><Country>India</Country></Person>
```

What’s the difference between an URI and an URL

What’s the difference between an URI and an URL

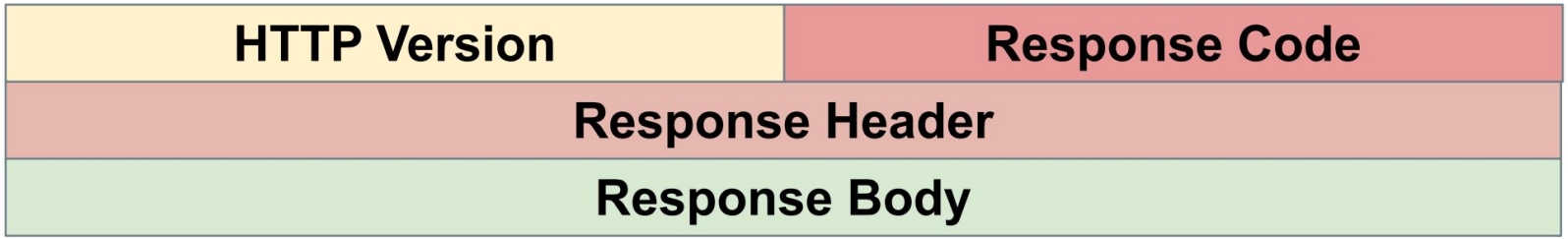


URI (Uniform Resource Identifier) Is just the *identifier* of the resource (e.g. `/users/123`)

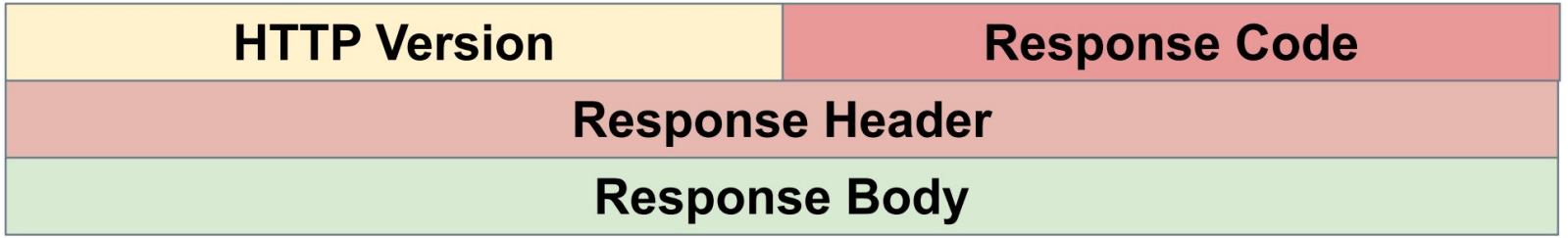
URL (Uniform Resource Locator) is a specific type of URI that not only identifies a resource but also provides the resource *location* (e.g., its network "location"). Essentially, all URLs are URIs, but not all URIs are URLs. (e.g. `https://api.example.com/users/123`)

Anatomy of an HTTP Response

- **Response code** contains request status. 3-digit HTTP status code from a pre-defined list.
- **Response Header** contains metadata and settings about the response message.
- **Response Body** contains the requested resource or data - if the request was successful. Can be as a JSON, XML or other formats such as an HTML or just text.



Anatomy of an HTTP Response



HTTP/1.1 200 OK

Date: Sat, 23 Aug 2014 18:31:04 GMT, Server: Apache/2, Last-Modified: Wed, 01 Sep 2004 13:24:52 GMT, Accept-Ranges: bytes, Content-Length: 32859, Cache-Control: max-age=21600, must-revalidate, Expires: Sun, 24 Aug 2014 00:31:04 GMT, Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"> <html xmlns='http://www.w3.org/1999/xhtml'> <head><title>Hypertext Transfer Protocol -- HTTP/1.1</title></head> <body> …

Response codes

Status code	Meaning
200 OK	Request was successful.
201 Created	The request has been fulfilled and resulted in a new resource being created.
204 No Content	The request was successful, but there is no representation to return (i.e., the response is empty).
301 Moved Permanently	For SEO purposes when a page has been moved and all link equity should be passed through.
400 Bad Request	The request could not be understood or was missing required parameters.
401 Unauthorized	Server requires authentication.
403 Forbidden	Client authenticated but does not have permissions to view resource.
404 Not Found	Page not found because no search results or may be out of stock.
500 Internal Server Error	Server side error. Usually due to bugs and exceptions thrown on the server side code.
503 Server Unavailable	Server side error. Usually due to a platform hosting, overload and maintenance issue.

Response cc

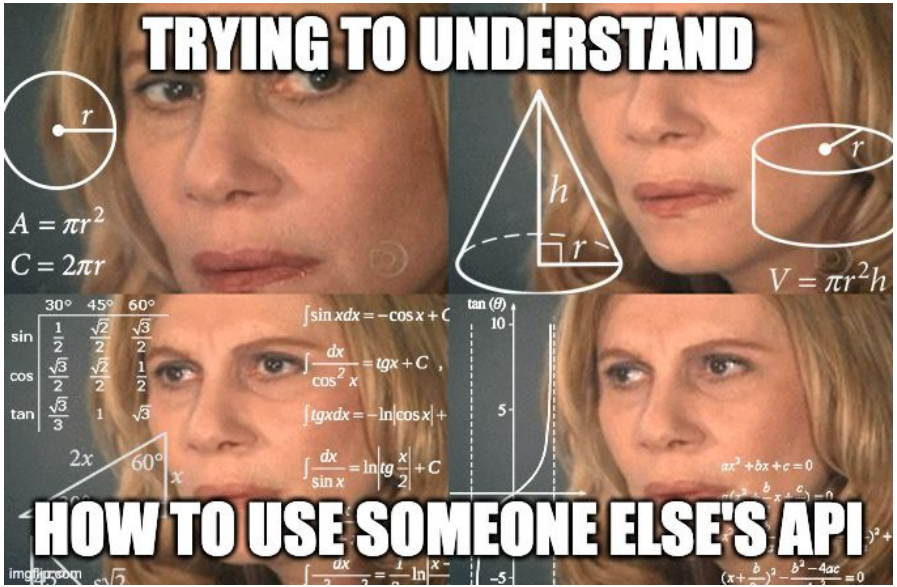
Status code	
200 OK	
201 Created	g created.
204 No Content	n (i.e., the response
301 Moved Permanentl	y should be passed
400 Bad Request	meters.
401 Unauthorized	
403 Forbidden	ce.
404 Not Found	
500 Internal Server Erro	e server side code.
503 Server Unavailable	maintenance issue.

LIÈGE université

Best practices for RESTful APIs

- Avoid using cryptic resource names
- Nouns not verbs when naming resources
  - `GET /users not GET /get_users`
- Plural nouns
  - `GET /users/{userId} not GET /user/{userID}`
- Dashes in URIs for resources and path parameters but use underscores for query parameters
  - `GET /admin-users/?find_desc=super`
- Return appropriate HTTP and informative messages to the user

OpenAPI & Swagger

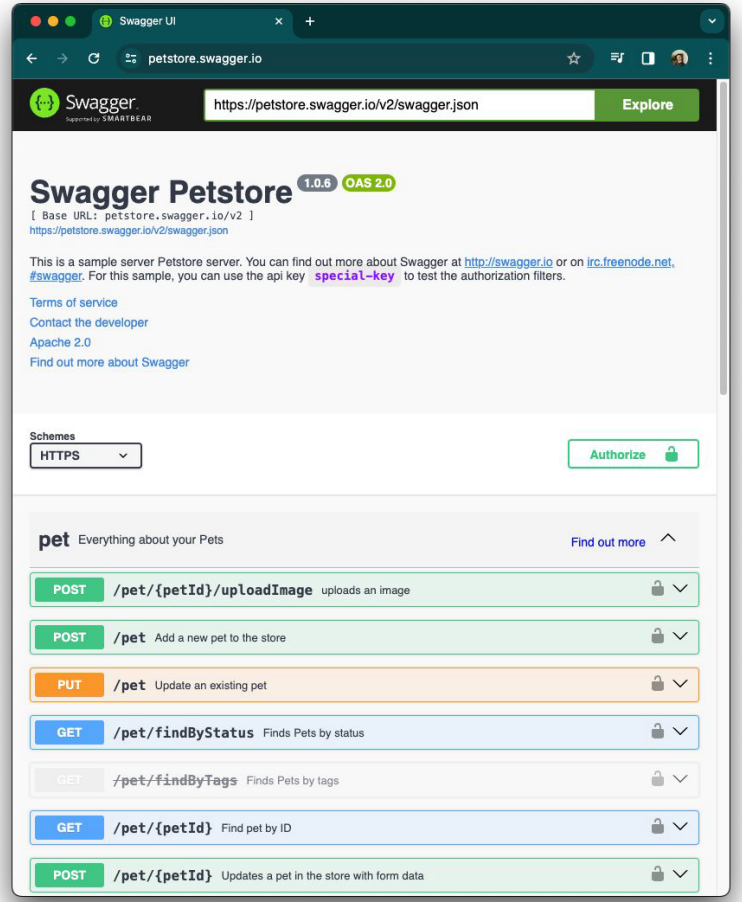


https://swagger.io/blog/api-strategy/difference-between-swagger-and-openapi/

OpenAPI & Swagger

OpenAPI Specification (OAS):

- A standardized, language-agnostic format (YAML or JSON) to describe your entire REST API
- Acts as a contract between the API provider and its consumers
- Originally called "Swagger Specification", renamed to OpenAPI when donated to the Linux Foundation in 2015



https://swagger.io/blog/api-strategy/difference-between-swagger-and-openapi/

Common API authentication mechanisms

API Key

- A unique string sent with each request (typically as a header or query parameter)
- Simple to implement, widely used for machine-to-machine communication
- Example: X-API-Key: sk-abc123... in the request header

OAuth 2.0

- Industry standard for delegated authorization
- The client obtains a short-lived access token from an *authorization server*
- Allows fine-grained permissions (scopes) without sharing credentials

JSON Web Token (JWT)

- A self-contained, signed token that encodes user identity and permissions
- The server can verify the token without calling a database (stateless)
- Commonly used in combination with OAuth 2.0 as the access token format

Server-Sent Events (SSE)

"What if the server's response arrives in pieces over time instead of all at once?"

SSE is a standard that allows a server to push updates to a client over a single HTTP connection.

The client makes a normal HTTP request, but instead of a single response, the server sends a stream of data: events. Built on HTTP, works with existing infrastructure (proxies, load balancers, auth).

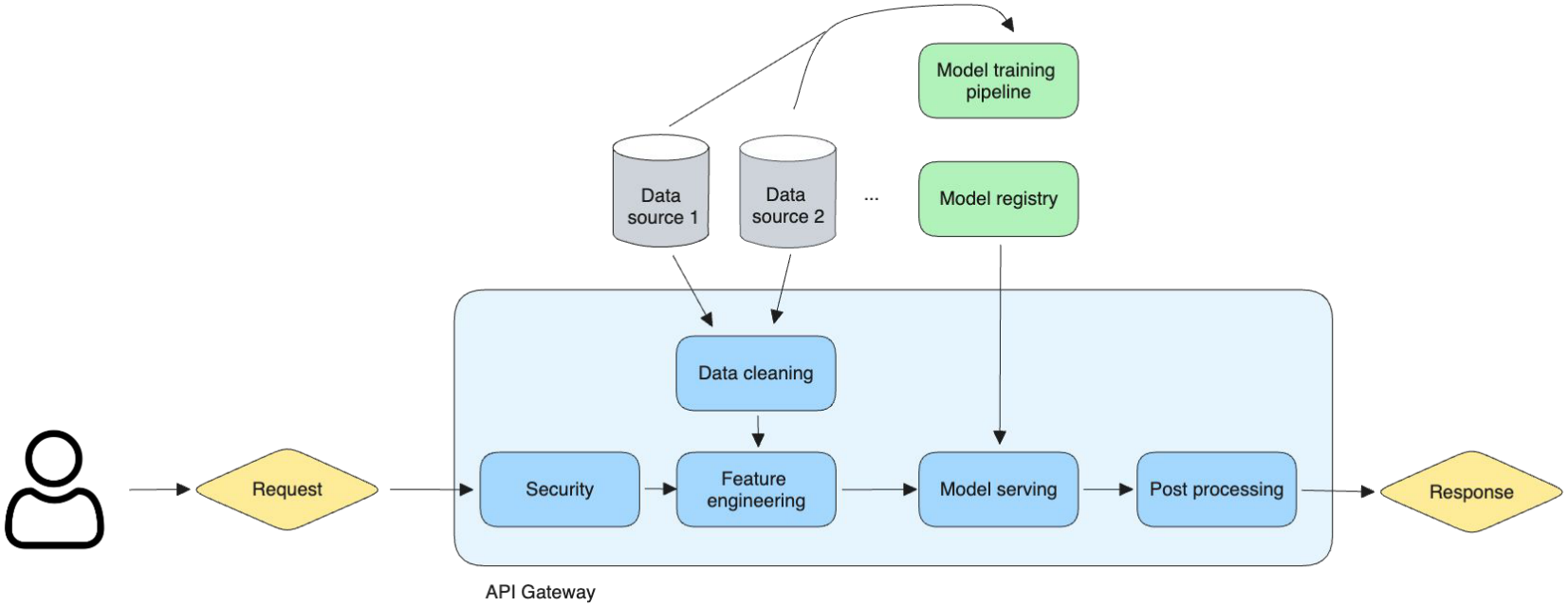
Flask vs FastAPI

Flask	FastAPI
Mature, widely adopted, large ecosystem of extensions	Modern framework, built on top of Starlette and Pydantic
Synchronous by default (one request at a time per worker)	Async support out of the box (handles concurrent requests efficiently)
Manual input validation and documentation	Automatic input validation via Python type hints and Pydantic models
Great for learning, prototyping, and simple APIs	Auto-generates OpenAPI documentation (Swagger UI included)

We will use Flask in the lab, as it is easier to get started with.

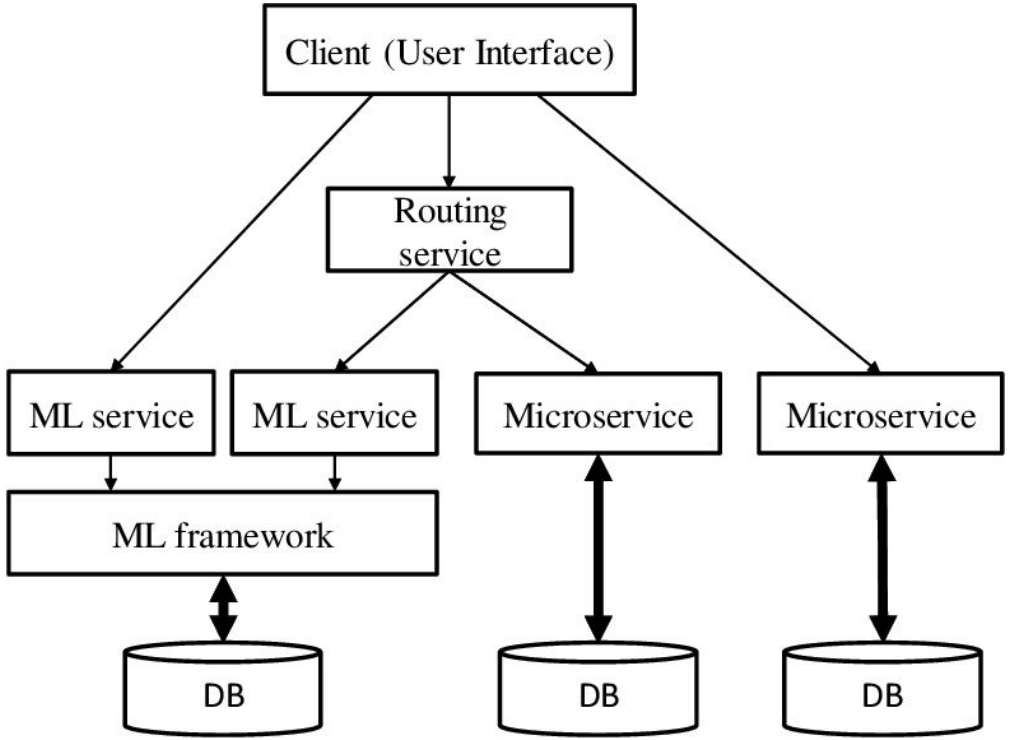
ML API: Might include different logic parts

More components than purely the model serving



You might implement many different independent services:

Microservices



Washizaki, Hironori et al. (2022). **Software-Engineering Design Patterns for Machine Learning Applications**

gRPC

From RPC to gRPC

RPC (Remote Procedure Call)

- Client invokes a function on a remote server as if it were local
- "Action-centric": designed for executing operations, not just retrieving resources

REST is resource-centric (`GET /users/123`), RPC is action-centric (`getUser(123)`)

Google formalized and modernized these principles into gRPC.

gRPC: a high-performance RPC framework by Google (2015, open-source)

Placing the “g” in “gRPC”

Placing the “g” in “gRPC”

Google created it, and in early versions people informally read it as "Google RPC," but officially it's recursive.

'g' stands for something different every gRPC release:

- 1.0 'g' stands for 'gRPC'
- 1.1 'g' stands for 'good'
- 1.2 'g' stands for 'green'
- 1.3 'g' stands for 'gentle'
- 1.4 'g' stands for 'gregarious'
- 1.6 'g' stands for 'garcia'
- 1.7 'g' stands for 'gambit'
- 1.8 'g' stands for 'generous'
- 1.9 'g' stands for 'glossy'
- 1.10 'g' stands for 'glamorous'
- 1.11 'g' stands for 'gorgeous'
- 1.12 'g' stands for 'glorious'
- 1.13 'g' stands for 'gloriosa'
- 1.14 'g' stands for 'gladiolus'
- 1.15 'g' stands for 'glider'
- 1.16 'g' stands for 'gao'
- 1.17 'g' stands for 'gizmo'
- 1.18 'g' stands for 'goose'
- 1.19 'g' stands for 'gold'
- 1.20 'g' stands for 'godric'
- 1.21 'g' stands for 'gandalf'

https://github.com/grpc/grpc/blob/master/doc/g_stands_for.md

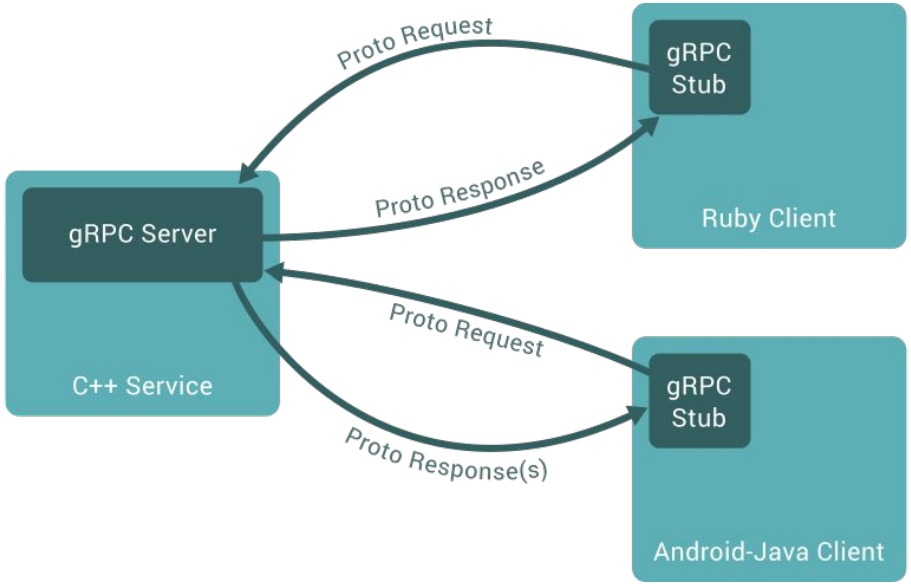
What is gRPC ?

Efficient Serialization: Parses messages with **Protobuf**. Binary format ensures compact messages, ideal for high-throughput systems.

HTTP/2 Support: Enables multiplexing and improved performance over traditional HTTP/1.1.

Supports 4 communication patterns: unary, server streaming, client streaming, bidirectional streaming

Internal microservices: gRPC is used for internal microservice transactions. It is not supported by standard web browsers.



Protocol Buffers (Protobuf)

Defines the “contract” between the systems

- **Strongly typed**: errors caught at compile time, not runtime
- **Language-agnostic**: generates Python, Go, Java, C++ stubs automatically
- **Stream**: enables token-by-token LLM output

Example “prediction.proto” file

```
syntax = "proto3";

service PredictionService {
  rpc Predict (PredictionRequest) returns (PredictionResponse);
  rpc PredictStream (PredictionRequest) returns (stream PredictionResponse);
}

message PredictionRequest {
  repeated float features = 1;
  string model_id = 2;
}

message PredictionResponse {
  float score = 1;
  string label = 2;
}
```

Protocol Buffers (Protobuf)

Defines the “contract” between the systems

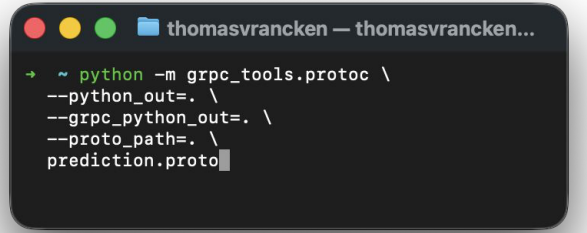
Example “prediction.proto” file

```
syntax = "proto3";

service PredictionService {
  rpc Predict (PredictionRequest) returns (PredictionResponse);
  rpc PredictStream (PredictionRequest) returns (stream PredictionResponse);
}

message PredictionRequest {
  repeated float features = 1;
  string model_id = 2;
}

message PredictionResponse {
  float score = 1;
  string label = 2;
}
```




```
project/
  prediction.proto
  prediction_pb2.py
  prediction_pb2_grpc.py
```

Data formats

Format	Binary/Text	Human readable	Use cases	Example
JSON	Text	Yes	Everywhere	{ ‘foo’: [1,2,3, …]}
CSV	Text	Yes	Everywhere	name,age\nAlice,30
Parquet	Binary	No	Amazon Redshift	(Binary column chunks – not readable in editor)
Avro	Binary primary	No	Hadoop	Schema: {"type":"record","fields":[{"name":"age", "type":"int"}]}
Protobuf	Binary primary	No	TensorFlow (TFRecord)	proto\nmessage User { int32 age = 1; }\n
Pickle	Binary	No	Python, PyTorch serialization	b'\x80\x04\x95...'

gRPC in Python

Server side python codes

```python
import grpc
import prediction_pb2
import prediction_pb2_grpc


class PredictionServicer(prediction_pb2_grpc.PredictionServiceServicer):
    def Predict(self, request, context):
        result = model.predict([request.features])
        return prediction_pb2.PredictionResponse(
            score=result[0], label="positive"
        )

server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
prediction_pb2_grpc.add_PredictionServiceServicer_to_server(
    PredictionServicer(), server)
server.add_insecure_port("[::]:50051")
server.start()
```

gRPC in Python

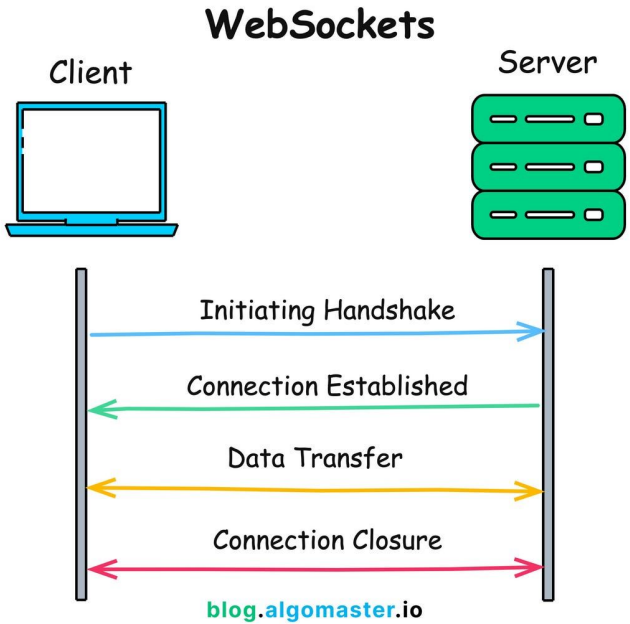
Client side python codes

```python
import grpc
import prediction_pb2
import prediction_pb2_grpc

channel = grpc.insecure_channel("localhost:50051")
stub = prediction_pb2_grpc.PredictionServiceStub(channel)
response = stub.Predict(
    prediction_pb2.PredictionRequest(features=[1.0, 2.0], model_id="v1")
)
print(response.score, response.label)
```

Websocket

WebSockets



Creates a persistent connection between the client and server. Starts as a standard HTTP request.

Bidirectional: both client and server can send messages at any time. Server can *send* “unrequested” data to the client.

Low latency: no overhead of repeated HTTP handshakes

Different URL scheme:

- REST: http:// or https://
- WebSocket: ws:// or wss://

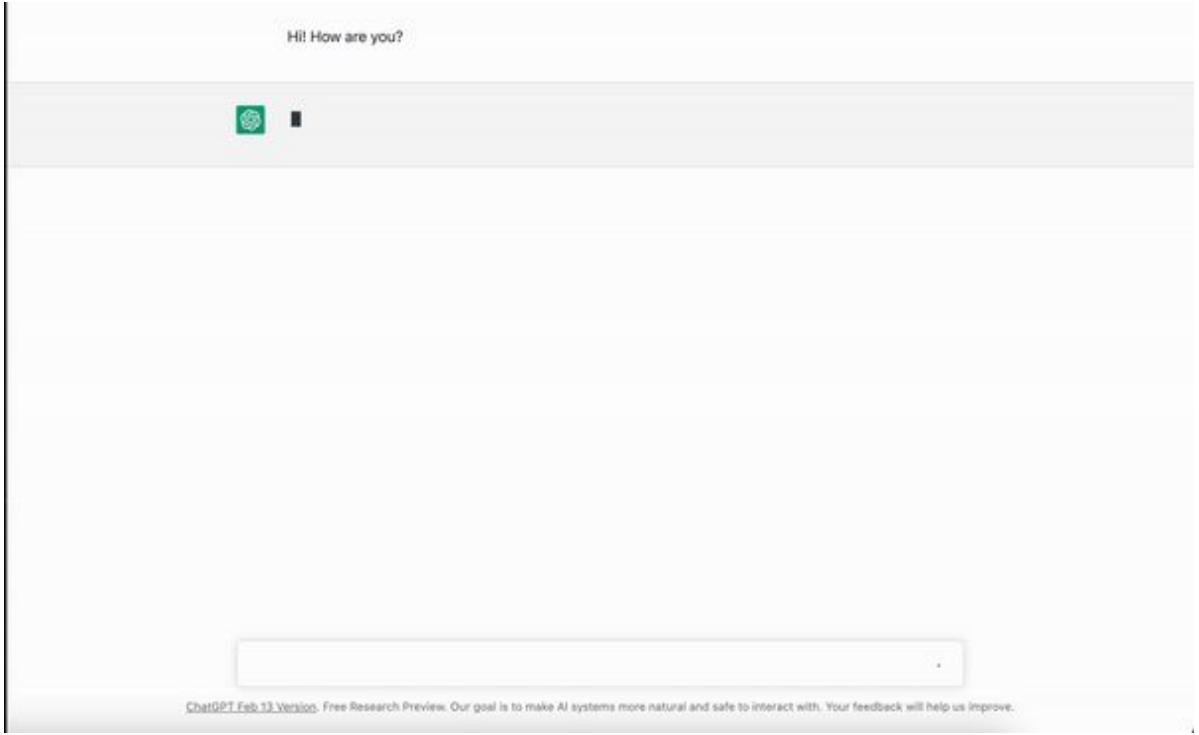
Requires an open connection with the server (more resource intensive)

https://blog.algomaster.io/p/websockets

Streaming

Why streaming matters?

Prevalent in LLM applications - inherently autoregressive



API streaming options overview

	SSE	WebSocket	gRPC streaming
Direction	Server → Client (unidirectional)	Bidirectional	Bidirectional
Protocol	HTTP/1.1+	WS	HTTP/2
Data format	Text (text/evewnt-stream)	Text or binary frames	Binary (Protocol Buffers)
Browser support	Native (EventSource API)	Native (WebSocket API)	Requires grpc-web proxy (mostly service-to-service)
Reconnection	Automatic (built-in)	Manual	Manual
Infrastructure	Works with standard HTTP infra	Needs WS-aware load balancers	Needs HTTP/2 + gRPC support
Complexity	Low	Medium	Higher
Best for	Server push, streaming APIs	Real-time interactive apps	Service-to-service comms

GenAI API Gateways

LIÈGE université

Credits where credits are due

ML6

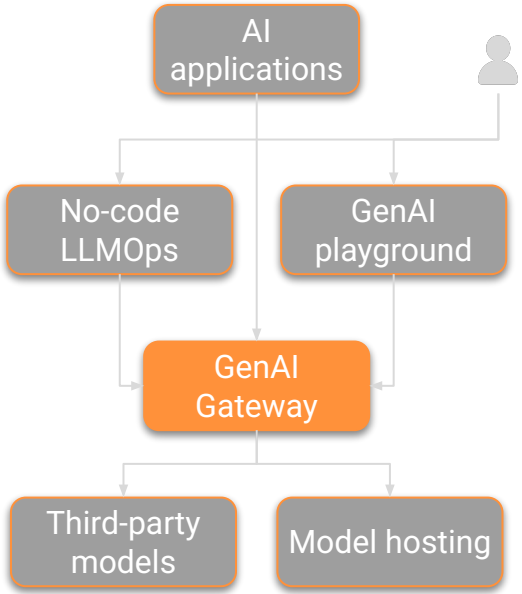
Robbe Sneyders
Office of the CTO @ ML6

Arne Pannemans
Machine Learning Engineer

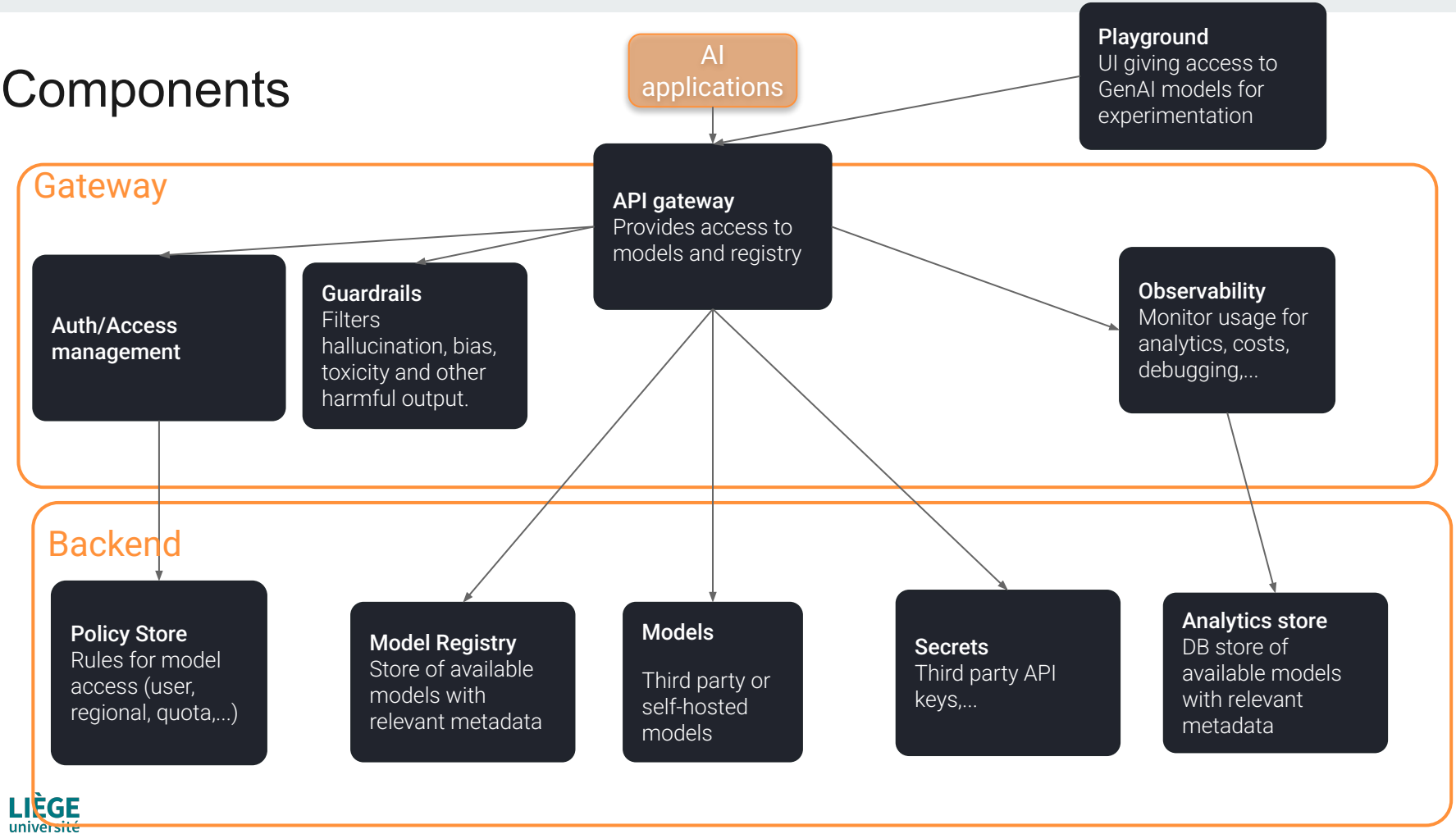
LIÈGE université

What is a GenAI Gateway?

- Central interface to manage access to GenAI models cross the organization
- Consumer AI applications send all model calls to the gateway using a unified, abstracted API
- The gateway routes all calls to the requested model, translating the requests to the selected model endpoint's specification
- The gateway monitors the traffic, manages authentication and access control, maintains, stores logs and implements any central guardrails



Components


AI applications
Playground
UI giving access to GenAI models for experimentation
Gateway
API gateway
Provides access to models and registry
Auth/Access management
Guardrails
Filters hallucination, bias, toxicity and other harmful output.
Observability
Monitor usage for analytics, costs, debugging,...
Backend
Policy Store
Rules for model access (user, regional, quota,...)
Model Registry
Store of available models with relevant metadata
Models
Third party or self-hosted models
Secrets
Third party API keys,...
Analytics store
DB store of available models with relevant metadata

Reference architecture

High level overview

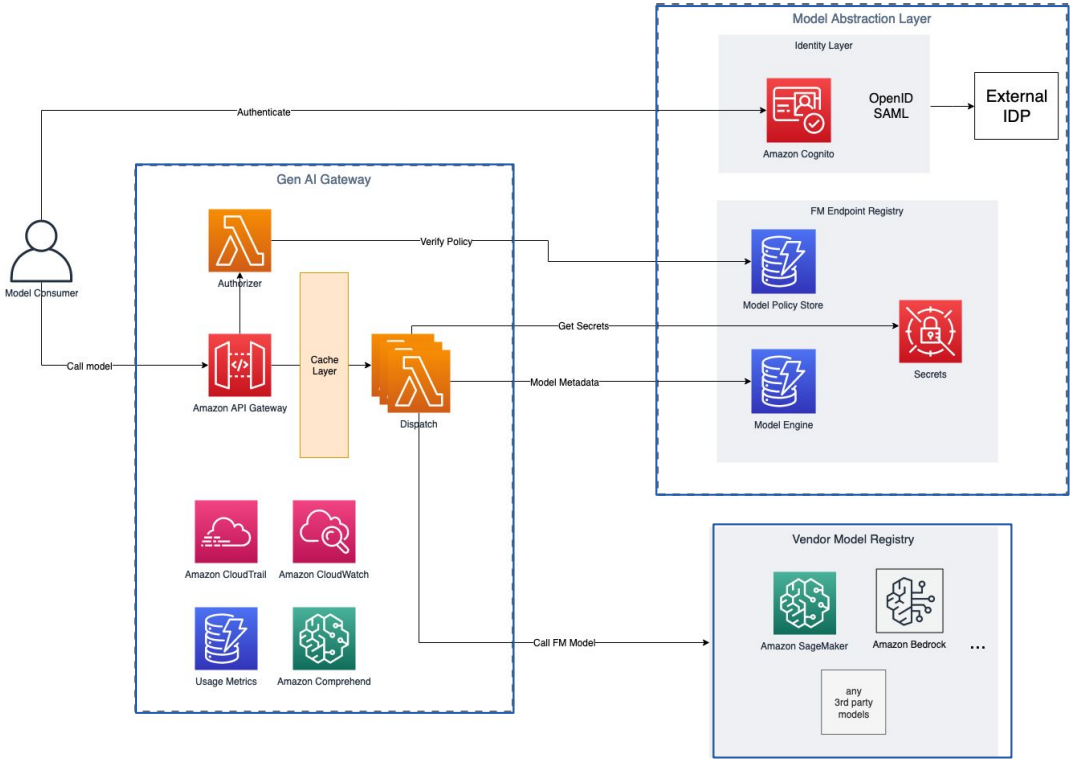
Model abstraction layer

- Model management
- Governance
- Secure access control

GenAI Gateway

- Consumer-facing API
- Request forwarding
- Monitoring, guardrails,...

Model abstraction layer



GenAI Gateway

https://aws.amazon.com/blogs/machine-learning/create-a-generative-ai-gateway-to-allow-secure-and-compliant-consumption-of-foundation-models/

Reference architecture

Model abstraction layer (MAL)

Endpoint registry

- Registry of endpoints for models that were added to the gateway

Model policy store

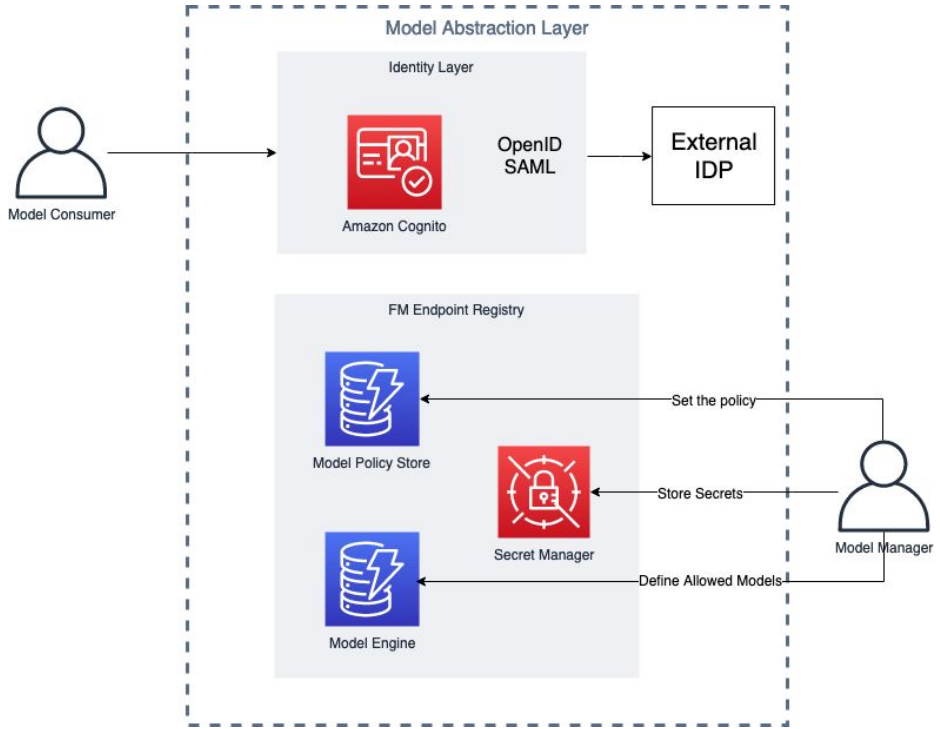
- Quantitative and qualitative rules for how models can be used (e.g. per location, content filters, …)

Identity layer

- Role-based access control (RBAC) to models. Granular user access permissions to the models.

Secret manager

- Store required keys & secrets for the endpoint to actually serve vendor models



https://aws.amazon.com/blogs/machine-learning/create-a-generative-ai-gateway-to-allow-secure-and-compliant-consumption-of-foundation-models/

Reference architecture

GenAI Gateway

Unified API interface

- Simplifies interaction with all models

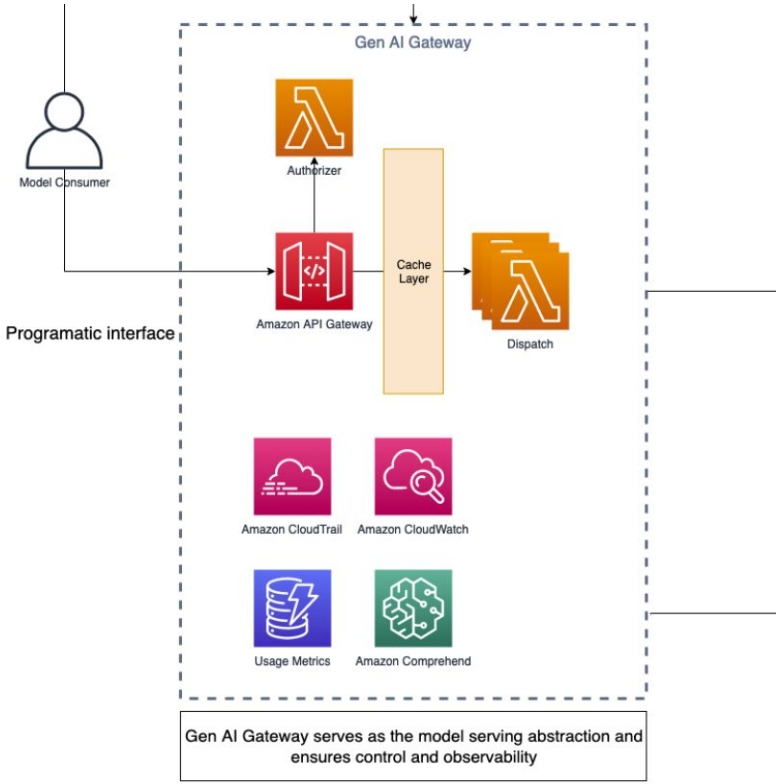
Cost control

- API quota, limits, and usage management
- Requests for dedicated hosting

Content, privacy, and responsible AI policy enforcement

- Guardrails for pre and post-processing of content
- Control of data flowing in the system

Monitoring and logging



https://aws.amazon.com/blogs/machine-learning/create-a-generative-ai-gateway-to-allow-secure-and-compliant-consumption-of-foundation-models/

Benefits attributed to general API gateways

Cost control mechanism: A robust system helps optimize resource use and manage costs. It tracks resource usage, model inference costs, and data transfer expenses. It centralises the decisions on which LLMs to adopt organisation wide.

Cache: Inference can be costly, especially during testing and development. A cache layer reduces costs and speeds up responses.

Observability: Logs capture all activities on the AI Gateway and Discovery Playground. They track user interactions, model requests, and system responses.

Quotas, rate limits, and throttling: These controls manage AI resource usage. Quotas limit requests per user or team within a set period. Rate limits prevent excessive usage.

Guardrails to manage risks: Routing GenAI through a central gateway makes it possible to implement organization-wide guardrails against hallucination, bias, toxicity and other harmful output.

https://aws.amazon.com/blogs/machine-learning/create-a-generative-ai-gateway-to-allow-secure-and-compliant-consumption-of-foundation-models/

GenAI gateway

How to communicate with it?

OpenAI

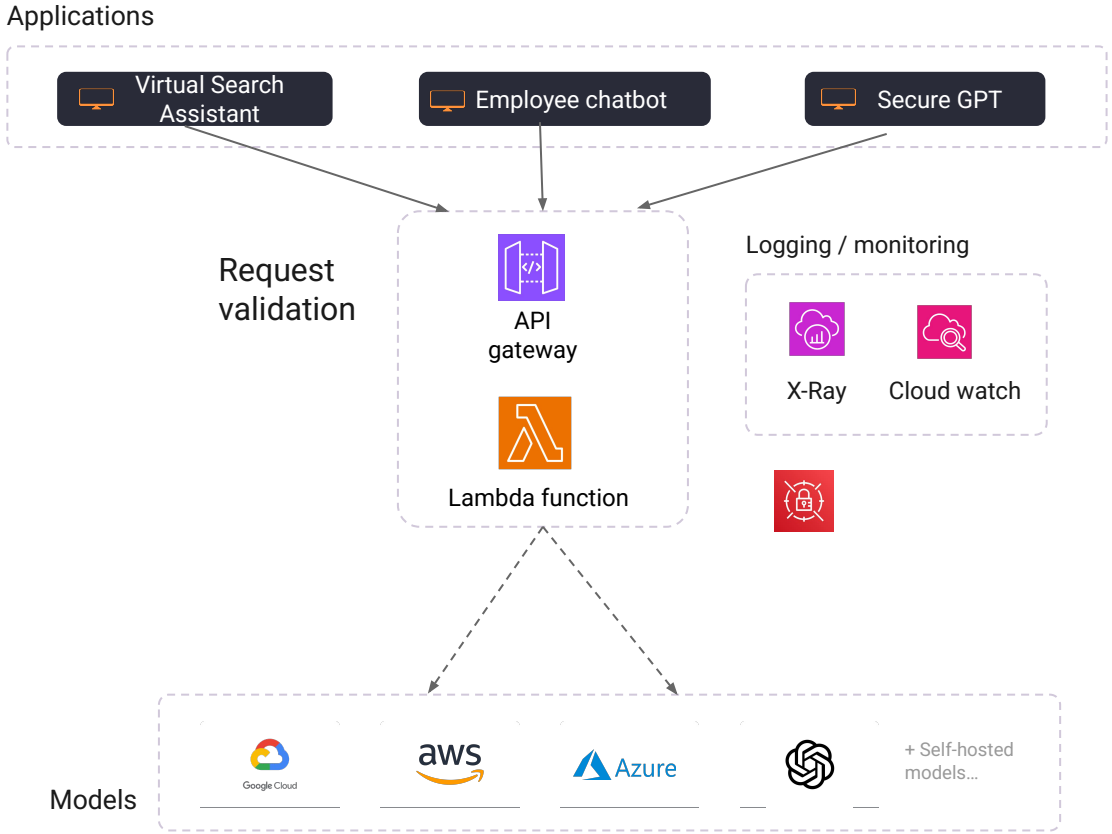
- /chat/completions
- /embeddings

AWS bedrock

- /invokeModel
- /converse

GCP

- /{model-id}:predict



https://aws.amazon.com/blogs/machine-learning/create-a-generative-ai-gateway-to-allow-secure-and-compliant-consumption-of-foundation-models/

OpenAI API standard

The industry is moving towards adopting the OpenAI API standard

```
POST /v1/chat/completions

{
  "model": "gpt-5.3",
  "messages": [
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": "Explain MLOps in one sentence."}
  ],
  "temperature": 0.7,
  "stream": true
}
```

Directed work: Flask

Wrap-up

Lecture summary

Topic	Concepts	To know for…	
		Project	Exam
API	• API • REST (anatomy request/response, status codes, authentication, OpenAPI, swagger) • gRPC (protobuf) • Websocket (bidirectional) • Streaming	Yes	Yes
GenAI Gateway	• GenAI Gateway (Model abstraction layer) • OpenAI API standard		Yes
Flask		Yes	

Milestone 1 presentations

- 15min for the presentation + 5min QA
- Make sure to send an email with a link to the PR to the teaching staff **before** your presentation
  - Your slides must be in the codebase as part of that PR!
  - *If we didn't accept your Github invite yet, please send another one - those expire after 1 week*

Do's

- Be in time
- Share speaking time (each team members presents what she/he built)
- Use clear visuals :-)

Don't

- Spend too much time on less critical parts
- Present things you didn't do (we'll have a close look 🕵)

LIÈGE université

Project objective for sprint 3

#	Week	Work package
3.1	W05	Build an **REST API** to **serve your model** and any extra logic that is needed to serve it (e.g. using Flask or FastAPI). You should be able to run the API locally.
3.2	W04	Package your service (API codes) in a **Docker container**. This too should be runnable locally.
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. Typically, you can use Google Cloud Run to deploy your API. Note that in the last sprint you will need to connect your dashboard to your model or data through the API you deployed here (work package 6.1). **Attention**: This can incur Cloud **costs**. Make sure to configure your Cloud Run instance to *scale to zero*. Also carefully track your Google Cloud credits and make sure to use the credits provided for free through this course. You can ask for support from the teaching staff in that regard.

Next week…

- Today we saw how an API works and how to build a local Flask API
- Next week we will see how you can deploy an API in the Cloud

LIÈGE université

That’s it for today!

LIÈGE université