Data-intensive applications using microservices

CE290I - Control and information management
Systems Engineering, UC Berkeley

Elói Pereira

November 13, 2023

About me

- Head of Data Science at [Car IQ Inc]
- 10 years at Portuguese Air Force Academy
- 5 years at Air Force Metrology Lab
- PhD in Systems Engineering, UC Berkeley
- Interested in: DS, DE, AI, ML, mobile robotics, environmental monitoring, digital payments...

eloipereira.com
github
linkedin

Data-intensive applications

- Applications that process large volumes of data, typically in real time or near-real time
- Examples: recommendation engines, fraud detection, real-time analytics, and scientific research.
- Large heterogeneous stacks, comprised by different services developed by different teams, SW Eng, DE, DS, ML, etc.
- Must meet hard requirements on scale, performance, and security
- Microservices architectures have been widely adopted for data-intensive applications

Why not Monoliths?!

Everybody used to love them!

Source: 2001: a space odyssey

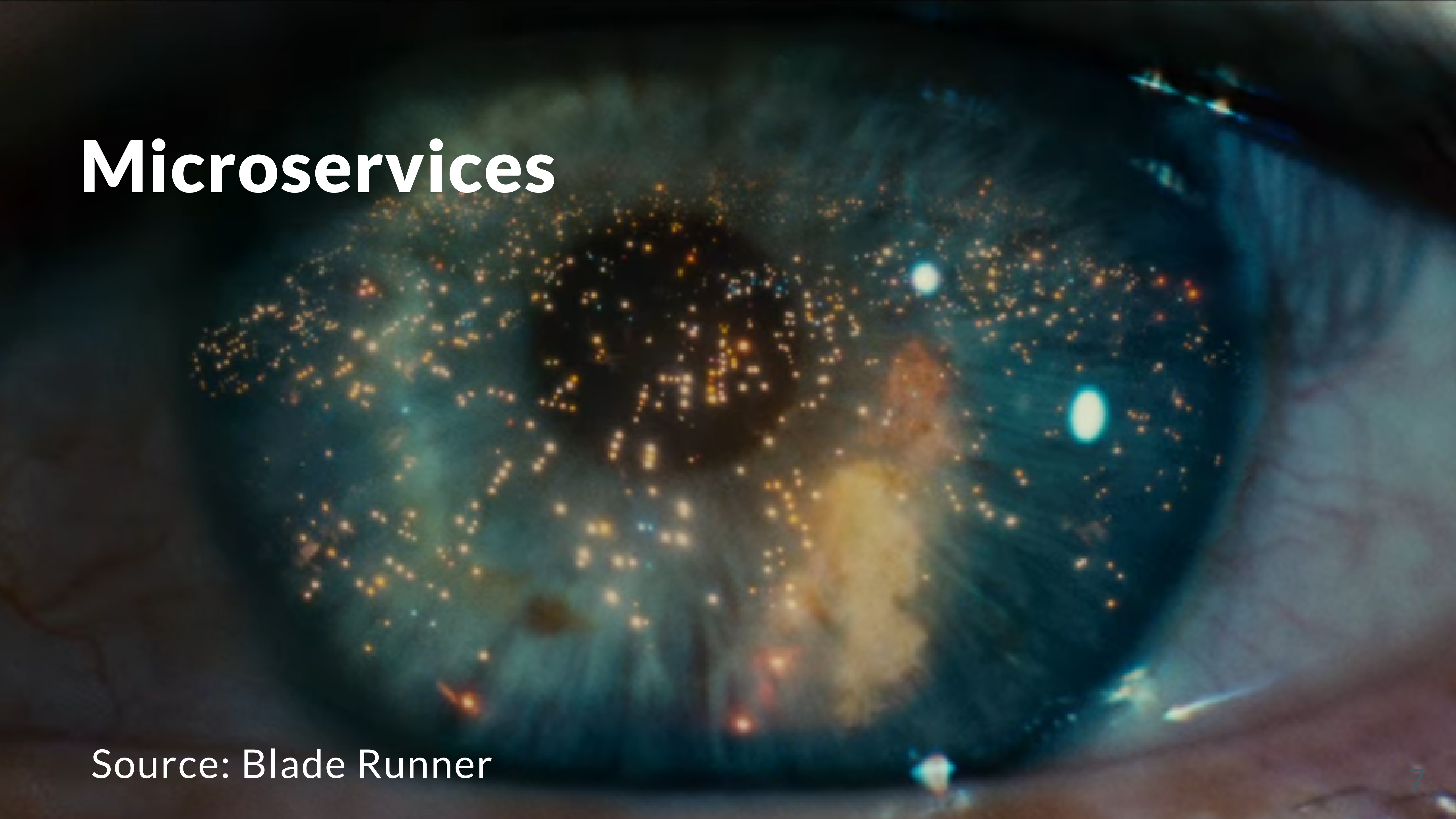
Monolithic Architectures: the good

- Traditional software architecture
- All the components are tightly integrated and packaged together
- Application is built, deployed, and scaled as a single unit
- Single codebase
- Small dev footprint: programming languages, tools, and platforms
- Easy to deploy
- Easy to document
- Easy to understand

Monolithic Architectures: the bad

- Difficult to maintain as the application grows in size and functionality
- Difficult to handle legacy code
- Difficult to introduce new features
- Developers need to understand in depth the overall application
- The entire application needs to be scaled together
- Any changes or updates require redeploying the entire application

Microservices



Source: Blade Runner

Microservices Architectures

- Independently developed, deployed, and managed
- Easy to maintain and evolve
- Scalable and reliable
- Containerization makes them agnostic to the infrastructure
- Easy to deploy in the cloud
- Flexible and agile

A simple example - GPS Replay

- Read a GPS sensor data from a csv file and replay it in real-time
- Serve the current state to downstream applications, e.g.:
  - A web app dashboard
  - A monitoring tool
  - etc.

Development Tools

- Git: distributed version control system
- Docker: platform as a service
- Python, Version: 3.12: programming language
- FastAPI: API web framework
- Pydantic: data validation
- Uvicorn: web server
- Streamlit: App web framework
- Redis: in-memory database and message broker

Development tools (optional)

- Pyenv: Python version management
- VSCode: Editor
- Pylance: python support for VSCode
- Flake8: linter
- Black: formatter
- Pre-commit: code validation
- Marp to build this presentation

Docker

- Platform for building, shipping, and running applications.
- It uses lightweight containers that are portable and isolated, making them easy to deploy and scale.
- Docker containers share the operating system kernel, but run as isolated processes, which makes them efficient and secure
- Docker is a popular choice for running applications in production
- Docker can be used to run a wide variety of applications, including web applications, databases, and microservices.

Docker - example

- Create a function named `replay_gps_from_csv` that reads `dataset_gps.csv` and replays the GPS data in real-time, printing the result in the command line
- Create a script named `replay_and_print.py`
- Create a dockerfile named `dockerfile.replayer_print` that runs `replay_and_print.py`
- Build and run your container using `docker build` and `docker run`
- Inspect your container using `docker exec`

Docker compose

- Command-line tool for defining and running multi-container Docker applications
- Takes a YAML file defining and configuring the services of your application, builds and runs all the services
- Docker Compose makes it easy to define and manage complex multi-container applications
- It also makes it easy to share your applications with other developers or deploy them to production

Docker compose - example

- Create a YAML file named `docker-compose-multi-replayer.yml` with three services, each one being an instance of `dockerfile.replayer_print`
- Build and run your multi-container application using `docker compose build` and `docker compose up`

Client-service REST Microservices

- REST (Representational State Transfer) is a set of architectural principles for designing networked applications based on HTTP
- REST Microservices communicate with each other over HTTP, using HTTP methods (i.e., GET, POST, PUT, and DELETE)
- They exchange data in a lightweight format such as JSON or XML

REST-based GPS Replayer

- Backend microservice
  - Real-time replayer
  - Cache to share state
  - REST API for serving current values
- Frontend microservice
  - Web app requests current values and displays them

Building and running

- Clone the git repo
  - https://github.com/eloipereira/ce290i-microservices
- From your favorite CLI:
  - Build: `docker compose -f docker-compose-rest.yml build`
  - Run: `docker compose -f docker-compose-rest.yml up`
  - Stop: `docker compose -f docker-compose-rest.yml down`
- Open streamlit app: http://localhost:8501/

REST API

- REST API documentation at http://localhost:8000/docs
- GET /gps_state

```
curl http://localhost:8000/gps_state
```

- GET /compute_avg_speed

```
http://localhost:8000/compute_avg_speed?time_window_seconds=10
```

REST API: non-blocking

- POST /request_to_compute_avg_speed

```
curl -X 'POST'
http://localhost:8000/request_to_compute_avg_speed?
time_window_seconds=10
```

- GET /get_avg_speed_task_status

```
http://localhost:8000/get_avg_speed_task_status?task_id=<uuid>
```

Event-driven microservices

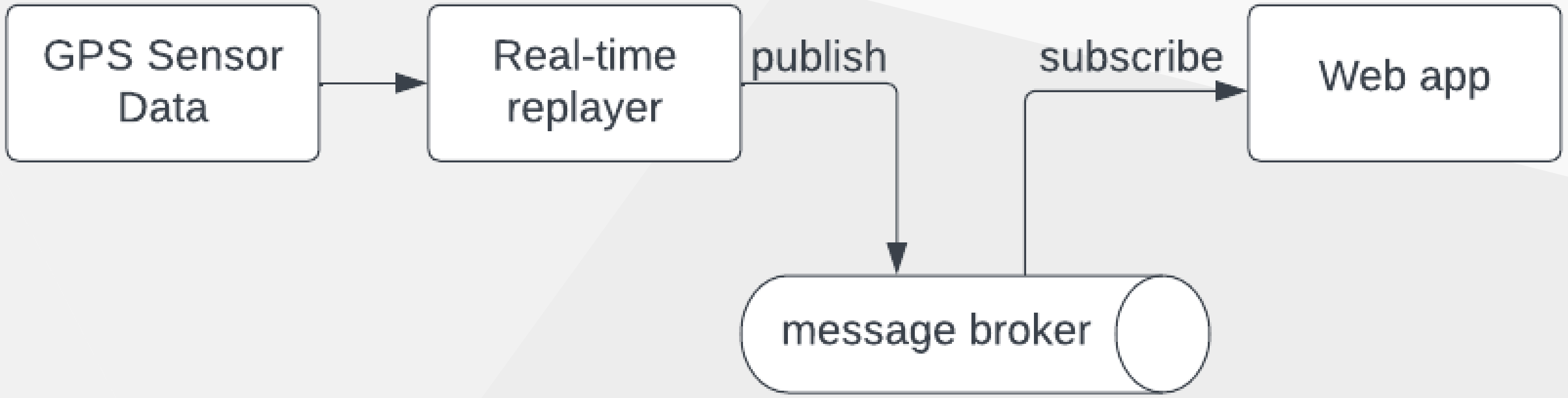
- In event-driven microservices, services communicate and interact with each other by sending messages or reacting upon messages being received
- This architecture allows for asynchronous communication between services, where services can react to events in real-time
- It also enables scalability and fault tolerance, as services can be added or removed without impacting the overall system

Publish-subscribe (pub-sub)

- Asynchronous communication pattern where senders, known as **publishers**, send messages to a message broker.
- The broker then distributes or broadcasts these messages to multiple receivers, known as **subscribers**
- Publishers do not need to have knowledge of the subscribers
- Publishers and subscribers communicate through topics or channels

Event-driven GPS Replayer

- Redis as pub-sub broker
- GPS replayer service publishes to the redis broker on a topic named `gps_state`
- Streamlit web app subscribes to the `gps_state` topic and reacts upon any message received



Building and running

- From your favorite CLI:
  - Build: `docker compose -f docker-compose-pubsub.yml build`
  - Run: `docker compose -f docker-compose-pubsub.yml up`
  - Stop: `docker compose -f docker-compose-pubsub.yml down`
- Open streamlit app: http://localhost:8502/

Conclusion

- Data-intensive applications process large volumes of data and have hard requirements on scale, performance, and security
- They have often large heterogeneous stacks, developed by different teams, such as SW eng, DE, DS, ML, etc
- Microservices architectures have been widely adopted for data-intensive applications
- REST APIs are used whenever the client needs data per request
- Event-driven applications are used whenever the client must react to events in real-time