Fullstack-AI Course

Henry A. Ruiz Guzman,PhD

2024-08-08

Table of contents

Preface

This website is a compilation of notes, tutorials, and code examples used in the "Fullstack AI" course offered at Autonoma de Occidente University in Cali, Colombia. The course is part of the AI Specialization Program and aims to provide a comprehensive introduction to the design and development of AI systems, covering both theoretical and practical aspects. The course is designed to be accessible to students with varying backgrounds, including those without any prior experience in AI.

This course was created and developed by Professor Henry A. Ruiz, PhD, who currently works as a research scientist at Texas A&M University. The course is based on the author's experience in developing AI systems for a wide range of applications, including agriculture, healthcare, IoT, and robotics. The course is designed to provide students with a solid foundation in AI concepts and techniques, as well as practical experience in developing AI systems using popular tools and libraries.

Course Materials

Code Examples

The code examples, slides and tutorials can be found in the following GitHub repository: Fullstack AI

Slides

- Course Overview
- Syllabus
- Final Project
- ML and Generative AI Foundations
- LLM Applications and design patterns
- Setting up Machine Learning Projects
- Infrastructure and tooling

Contributing

If you find any errors or have suggestions for improvements, please feel free to create an issue or submit a pull request on the GitHub repository. Your feedback is greatly appreciated!

Part I

Foundations

1 System Design

Introduction

System design is the process of laying out a system's structure, components, modules, interfaces, and data to meet specified requirements. It is a multidisciplinary field that requires a wide range of skills and knowledge, including software engineering, computer science, network engineering, and project management. From the machine learning perspective, comprehending a system's life cycle provides a blueprint for building, deploying, and maintaining ML/AI solutions in production. Thus, this section will introduce and discuss some of the more critical stages of the system design process, including requirements analysis, architecture, development, deployment, and scaling.

1.1 System design process overview

1.2 Defining the requirements of the system

This is the first step in the system design process, and it involves gathering, analyzing, and documenting the requirements for the system. The requirements analysis phase is crucial, as it provides the foundation for the rest of the system design process. It helps to ensure that the system will meet the needs of its users and stakeholders and that it will be developed within the constraints of time, budget, and resources. The requirements analysis phase typically involves the following activities:

1. **Gathering Requirements**: This involves collecting information about the needs, goals, and constraints of the system from its users and stakeholders. This information can be gathered through interviews, surveys, questionnaires, and workshops.
2. **Analyzing Requirements**: This involves analyzing the gathered information to identify the key features, functions, and constraints of the system. It also involves identifying any conflicts or inconsistencies in the requirements.
3. **Documenting Requirements**: This involves documenting the requirements in a clear, concise, and unambiguous manner. The requirements should be documented in a way that is understandable to all stakeholders, including developers, testers, and project managers.

4. **Validating Requirements**: This involves validating the requirements with the users and stakeholders to ensure that they accurately reflect their needs and goals. It also involves ensuring that the requirements are complete, consistent, and feasible.
5. **Managing Requirements**: This involves managing changes to the requirements throughout the system design process. It involves tracking changes, resolving conflicts, and ensuring that the requirements are kept up-to-date.

1.3 Types of Requirements

There are several types of requirements that need to be considered when designing a system. These include:

1. **Functional Requirements**: These are the requirements that describe the functions, features, and capabilities of the system. They specify what the system should do, and they are typically expressed as use cases, user stories, or functional specifications.
2. **Non-Functional Requirements**: These are the requirements that describe the quality attributes of the system, such as performance, reliability, availability, security, and usability. They specify how well the system should perform, and they are typically expressed as performance requirements, security requirements, and usability requirements.
3. **Business Requirements**: These are the requirements that describe the business goals, objectives, and constraints of the system. They specify why the system is being developed, and they are typically expressed as business cases, business rules, and business process models.
4. **User Requirements**: These are the requirements that describe the needs, goals, and constraints of the users of the system. They specify who will use the system, and they are typically expressed as user profiles, user scenarios, and user interface designs.
5. **System Requirements**: These are the requirements that describe the technical constraints and dependencies of the system. They specify how the system will be developed, deployed, and maintained, and they are typically expressed as system architecture, system interfaces, and system dependencies.
6. **Regulatory Requirements**: These are the requirements that describe the legal, ethical, and regulatory constraints of the system. They specify how the system should comply with laws, regulations, and standards, and they are typically expressed as compliance requirements, privacy requirements, and security requirements.

1.4 Selecting the appropriate methodology

There are several **software development methodologies** that can be used to develop a system, such as the waterfall model, the agile model, and the iterative model. Each of these methodologies has its own strengths and weaknesses, and they are suitable for different types of projects and teams. These methodologies also provided a framework for gathering, analyzing, and documenting the requirements the system.

1. **Waterfall Model**: The waterfall model is a linear and sequential software development methodology that divides the development process into distinct phases, such as requirements analysis, design, implementation, testing, and maintenance. Each phase must be completed before the next phase can begin, and the process is difficult to change once it has started. The waterfall model is suitable for projects with well-defined requirements and stable technologies, but it is not suitable for projects with changing requirements and emerging technologies.
2. **Agile Model**: The agile model is an iterative and incremental software development methodology that focuses on delivering working software in short iterations, typically two to four weeks. It emphasizes collaboration, flexibility, and customer feedback, and it is suitable for projects with changing requirements and emerging technologies. The agile model is based on the principles of the Agile Manifesto, which emphasizes individuals and interactions, working software, customer collaboration, and responding to change.
3. **Iterative Model**: The iterative model is a software development methodology that divides the development process into small, incremental, and iterative cycles, each of which produces a working prototype of the system. The iterative model is suitable for projects with evolving requirements and complex technologies, and it is based on the principles of the spiral model, which emphasizes risk management, prototyping, and incremental development.

Once we have gathered, analyzed, and documented the requirements for the system, we can move on to the next phase of the system design process, which is architecturing the system.

1.5 Architecturing your solution

Architecturing a system is the act of decomposing a system into multiple building blocks so that we can identify how each building block can be developed, deployed, and maintained to achieve a high level of modularity, flexibility, and scalability. The architecture of a system provides a high-level view of how these components are arranged and interact with each other to achieve the desired functionality and performance. Several architectural styles and patterns can be used to design a system, including the client-server architecture, the microservices architecture, and the event-driven architecture. However, selecting the appropriate architectural style depends on the type of application to be developed and the system's requirements.

Classifying applications involves categorizing them based on criteria such as functionality, deployment methods, technology used, target user base, and platform. Here, we are providing a classification based on the deployment model.

- **Web Applications:** Accessed via web browsers, e.g., Google Docs, Salesforce.
- **Desktop Applications:** Installed on a personal computer or laptop, e.g., Microsoft Word, Adobe Photoshop.
- **Mobile Applications:** Designed for smartphones and tablets, e.g., Instagram, Uber.
- **Cloud Applications:** Hosted on cloud services and accessible over the Internet, e.g., Dropbox, Slack.

An ML model can be deployed in any of these application types. For example, a web application can use machine learning to provide personalized recommendations to users; a desktop application can use a machine learning model to automate repetitive tasks; and a mobile application can use a machine learning model to recognize speech or images, etc.

1.5.1 Client-"Server" Architecture

For web applications, cloud applications, and sometimes mobile applications, the **client-server architecture** is a common choice for the system architecture. The client-server architecture is a distributed computing architecture that divides the system into two major components: the client and the server. The client is the end-user device or application that requests and consumes the services provided by the server. The server is the remote computer or service that provides the resources, services, or data to the client. The client-server architecture provides a scalable, flexible, and reliable way to distribute and manage resources and services across a network. This architecture consists of the following components:

1. **Client**: The client is a device or a program that requests resources and services from the server. The client can be a web browser, a mobile app, or a desktop application.
2. **Server**: The server is a device or a program that provides resources and services to the client. The server can be a web server, an application server, or a database server.
3. **Network**: The network is the medium through which the client and server communicate with each other. The network can be a local area network (LAN), a wide area network (WAN), or the internet.
4. **Protocol**: The protocol is a set of rules and conventions that govern the communication between the client and server. The protocol can be HTTP, HTTPS, TCP, or UDP.

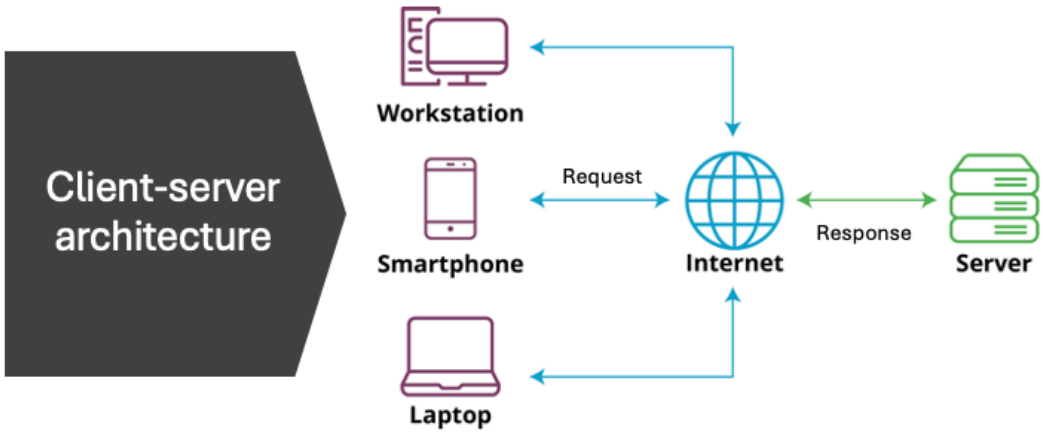

Figure 1.1: client-server-architecture

Client-Server-based applications can be deployed in different ways, following different architectural patterns. The most common ones are:

1.5.2 Server-Based architecture patterns

1.5.2.1 Monolithic Architecture

The monolithic architecture is a traditional software architecture pattern that consists of a single, self-contained application that contains all the components, modules, and services of the system. The monolithic architecture is based on the principles of tight coupling, where the components of the system are tightly integrated and dependent on each other. The monolithic architecture is suitable for small to medium-sized applications with simple requirements and low complexity. It is also suitable for applications with stable technologies and well-defined requirements. The monolithic architecture consists of the following components:

1. **User Interface**: The user interface is the front-end component of the system that interacts with the user. It can be a web interface, a mobile interface, or a desktop interface.
2. **Business Logic**: The business logic is the core component of the system that implements the business rules, processes, and workflows. It can be implemented as a set of classes, functions, or procedures.

3. **Data Storage**: The data storage is the back-end component of the system that stores and manages the data. It can be a relational database, a NoSQL database, or a file system.
4. **Integration**: The integration is the component of the system that integrates with external systems, services, and APIs. It can be implemented as a set of connectors, adapters, or gateways.

1.5.2.2 Microservices Architecture

The microservices architecture is a modern software architecture that consists of a collection of small, independent, and loosely-coupled services that are developed, deployed, and maintained independently. The microservices architecture is based on the principles of loose coupling, where the components of the system are decoupled and independent of each other. The microservices architecture is suitable for large-scale applications with complex requirements and high complexity. It is also suitable for applications with changing requirements and emerging technologies. The microservices architecture consists of the following components:

1. **Service**: The service is a small, independent, and loosely-coupled component of the system that provides a specific set of functions and capabilities. It can be implemented as a RESTful API, a message queue, or a microservice.
2. **Container**: The container is a lightweight, portable, and self-contained environment that hosts the service. It can be implemented as a Docker container, a Kubernetes pod, or a serverless function.
3. **Orchestration**: The orchestration is the component of the system that manages the deployment, scaling, and monitoring of the services. It can be implemented as a container orchestrator, a service mesh, or a serverless platform.
4. **Gateway**: The gateway is the component of the system that provides a single entry point for the services. It can be implemented as an API gateway, a message broker, or a load balancer
5. **Database**: The database add to each service the capability to store and manage the data. It can be a relational database, a NoSQL database, or a distributed database.

1.5.3 Serverless-Based Architecture patterns

With the rise of cloud computing and serverless computing, the **serverless architecture** has become an alternative to the client-server architecture. The serverless architecture is a cloud computing model that abstracts the infrastructure and runtime environment from the developer, allowing them to focus on writing code and deploying applications without managing servers. The serverless architecture is based on the principles of event-driven computing, where events, such as HTTP requests, database changes, or file uploads, trigger the execution of code. The serverless architecture consists of the following components:

1. **Function**: The function is a small, stateless, and event-driven piece of code that performs a specific task or function. It can be implemented as a serverless function, a lambda function, or a cloud function.
2. **Event**: The event is a trigger that initiates the execution of the function. It can be an HTTP request, a database change, or a file upload.
3. **Cloud**: The cloud is the infrastructure and runtime environment that hosts and manages the functions. It can be a cloud provider, such as AWS, Azure, or GCP.
4. **API**: The API is the interface that exposes the functions to the client. It can be a RESTful API, a GraphQL API, or a message queue.

1.5.4 Server-based vs Serverless-based architectures

- **Monolithic architectures** are best suited for small to medium-sized applications where simplicity and ease of management are key. However, they can become cumbersome to update and scale as the application grows.
- **Microservices** offer a highly scalable and flexible architecture that is suitable for complex applications that need to rapidly evolve. They require a significant upfront investment in design and infrastructure management but provide long-term benefits in scalability and maintainability.
- **Serverless architectures** abstract the management of servers, making it easier for developers to deploy code that scales automatically with demand. This model is cost-effective for applications with fluctuating workloads but introduces new challenges in managing application state and understanding cloud provider limitations.

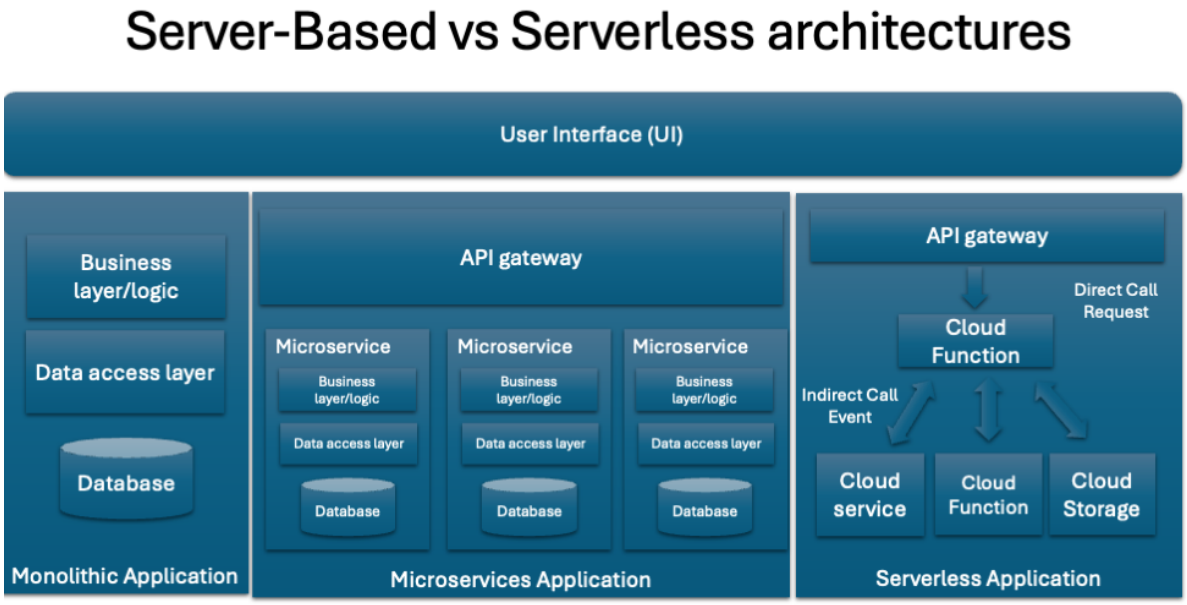

Figure 1.2: Architectural-styles

Factor	Monolithic	Microservices	Serverless
Definition	A software development approach where an application is built as a single and indivisible unit.	An architecture that structures an application as a collection of small, autonomous services modeled around a business domain.	A cloud-computing execution model where the cloud provider dynamically manages the allocation and provisioning of servers.
Complexity	Simple to develop and deploy initially but becomes more complex and unwieldy as the application grows.	Complex to design and implement due to its distributed nature, but easier to manage, understand, and update in the long term.	Low operational complexity for developers as the cloud provider manages the infrastructure, but can have complex architecture patterns.

Factor	Monolithic	Microservices	Serverless
Scalability	Scaling requires duplicating the entire application, which can be inefficient for resources.	Services can be scaled independently based on demand, leading to efficient use of resources.	Automatically scales based on the workload by running code in response to events, without provisioning or managing servers.
Development	Development is straightforward in the early phases but can slow down as the application grows due to tightly coupled components.	Enables the use of different technologies and programming languages for different services, potentially increasing development speed.	Development focuses on individual functions, potentially speeding up development cycles, but requires understanding of serverless patterns and limits.
Deployment	Deploying updates requires redeploying the entire application, which can be slow and risky.	Services can be deployed independently, allowing for faster and less risky updates.	Code is deployed to the cloud provider, which then takes care of deployment, scaling, and management, simplifying deployment processes.
Maintenance	Maintenance can be challenging as fixing a bug or making an update requires redeploying the entire application.	Easier to maintain and update individual services without impacting the entire application.	Maintenance of the infrastructure is handled by the cloud provider, but developers must manage their code's scalability and performance within the serverless environment.

Factor	Monolithic	Microservices	Serverless
Cost	Costs can be predictable but may not efficiently utilize resources due to the need to scale the entire application.	Potentially more cost- efficient as resources are used more effectively by scaling services independently.	Cost-effective for applications with variable traffic but can become expensive if not managed properly, due to the pay-per-use pricing model.
Use Cases	Suitable for small applications or projects where simplicity and ease of deployment are prioritized.	Ideal for large, complex applications requiring scalability, flexibility, and rapid iteration. darr	Best for event-driven scenarios, sporadic workloads, and rapid development cycles, where managing infrastructure is not desirable.

1.6 Developing and building your application

The process of developing a system involves the selection of the appropriate technologies, tools, and frameworks to implement the system. It also consists of designing and developing the system components, such as the user interface, business logic, and data storage. The development process should be straightforward after clearly defining the systems' requirements.

1.7 Testing your application

Testing is an essential part of the development process, as it helps to ensure that the system meets its requirements and performs as expected before going into production. Testing involves verifying and validating the system's functionality, performance, reliability, and security. It also involves identifying and fixing defects, bugs, and issues in the system. There are several types of testing that can be used to test a system, including:

- **Unit Testing**: This involves testing individual components, modules, or functions of the system to ensure that they work as expected. It is typically performed by developers using testing frameworks, such as JUnit, NUnit, or Mocha.
- **Integration Testing**: This involves testing the interactions and dependencies between the components, modules, or services of the system to ensure that they work together as expected. It is typically performed by developers using testing frameworks, such as TestNG, Cucumber, or Postman.

- **System Testing**: This involves testing the system as a whole to ensure that it meets its requirements and performs as expected. It is typically performed by testers using testing tools, such as Selenium, JMeter, or SoapUI.
- **Performance Testing**: This involves testing the performance and scalability of the system to ensure that it can handle the expected load and stress. It is typically performed by testers using performance testing tools, such as Apache JMeter, LoadRunner, or Gatling.
- **Security Testing**: This involves testing the security and reliability of the system to ensure that it is protected from unauthorized access and malicious attacks. It is typically performed by security experts using security testing tools, such as OWASP ZAP, Burp Suite, or Nessus.
- **User Acceptance Testing**: This involves testing the system with real users to ensure that it meets their needs and expectations. It is typically performed by users using acceptance testing tools, such as UserTesting, UsabilityHub, or UserZoom.
- **Stress Testing**: This involves testing the system under extreme conditions to ensure that it can handle the maximum load and stress. It is typically performed by testers using stress testing tools, such as Apache JMeter, LoadRunner, or Gatling.

1.8 Deploying it to a production environment

Deploying a system can be a complex and time-consuming process, and it requires careful planning and coordination to minimize the risk of downtime and data loss. Several deployment strategies and strategies exist. However, in today's world, Docker is generally the most suitable alternative to simplify this task. Docker is a platform for developing, shipping, and running applications using containerization. Containers are lightweight, portable, and self-contained environments that can run on any machine with the Docker runtime installed. They provide a consistent and reliable way to package and deploy applications, and they are widely compatible with cloud computing environments, such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP). Docker provides several benefits for deploying systems, including:

- **Portability**: Containers can run on any machine with the Docker runtime installed, making them highly portable and compatible with different environments.
- **Consistency**: Containers provide a consistent and reliable way to package and deploy applications, ensuring that they run the same way in development, testing, and production environments.
- **Isolation**: Containers provide a high level of isolation between applications, ensuring that they do not interfere with each other and that they are secure and reliable.
- **Scalability**: Containers can be easily scaled up or down to meet the demands of the system, making them highly scalable and flexible.
- **Efficiency**: Containers are lightweight and efficient, requiring minimal resources and providing fast startup times and high performance.

- **Security**: Containers provide a high level of security, ensuring that applications are protected from unauthorized access and malicious attacks.
- **Automation**: Containers can be easily automated using tools and platforms, such as Kubernetes, Docker Swarm, and Amazon ECS, making them easy to manage and maintain.
- **Cost-Effectiveness**: Containers are cost-effective, requiring minimal resources and providing high performance, making them ideal for cloud computing environments.
- **Flexibility**: Containers are flexible, allowing developers to use different technologies, tools, and frameworks to develop and deploy applications.
- **Reliability**: Containers are reliable, ensuring that applications run consistently and predictably in different environments.
- **Compatibility**: Containers are compatible with different operating systems, such as Linux, Windows, and macOS, making them highly versatile and widely used.

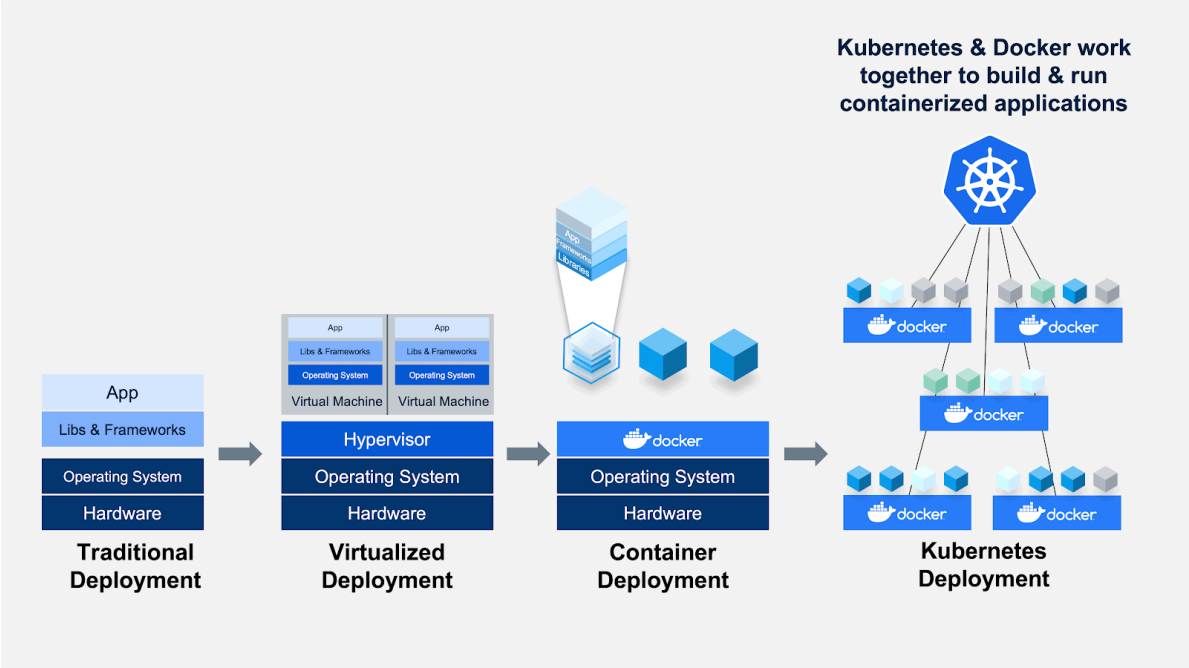

Figure 1.3: Deployment modes

1.8.1 Docker

Docker was introduced to the world by Solomon Hykes in 2013, founder and CEO of a company called dotCloud. It provides a platform for building, shipping, and running distributed applications. Docker introduced the concept of "containers" to package software into isolated environments that can run on any system with the Docker engine installed. This deployment

model makes it easy to run the same application in different environments, such as development, testing, and production, without worrying about dependencies, configurations, or compatibility issues.

1.8.1.1 Docker components

Docker consists of several components that work together to provide its functionality. These components include the Docker engine, the Docker client, and the Docker registry.

- **Docker Engine**: The Docker engine is the core component of Docker that provides the runtime environment for containers. It consists of the Docker daemon, which is responsible for building, running, and distributing containers, and the Docker runtime, which is responsible for executing the processes of containers. The Docker engine can run on any system with the Docker runtime installed, including Linux, Windows, and macOS, and it can be managed and monitored using tools and platforms, such as Docker Swarm, Kubernetes, and Amazon ECS.
- **Docker Client**: The Docker client is a command-line interface (CLI) that allows developers to interact with the Docker engine, providing a simple and intuitive way to build, run, and manage containers. The Docker client can also be used with graphical user interfaces (GUIs) and integrated development environments (IDEs), providing a seamless and consistent experience for developers.
- **Docker Registry**: The Docker registry is a repository for storing and distributing containers, allowing developers to build, push, and pull containers from Docker registries, such as Docker Hub, Amazon ECR, and Google Container Registry. The Docker registry provides a high level of visibility and control over the distribution of containers, ensuring that they are secure, reliable, and efficient.

1.8.1.2 Docker Hub

Docker also provides a centralized repository called the Docker Hub, where developers can store and share their containers with others. This makes it easy to find and reuse existing containers and to collaborate with other developers on projects. Docker Hub provides a wide range of official and community-contributed containers, including base images, application images, and service images. It also provides features for managing and monitoring containers, such as versioning, tagging, and scanning. Docker Hub is widely used by developers, organizations, and cloud providers, and it provides a high level of visibility and control over the distribution of containers. We can use Docker Hub to store and share our containers so our team members can easily access and use them.

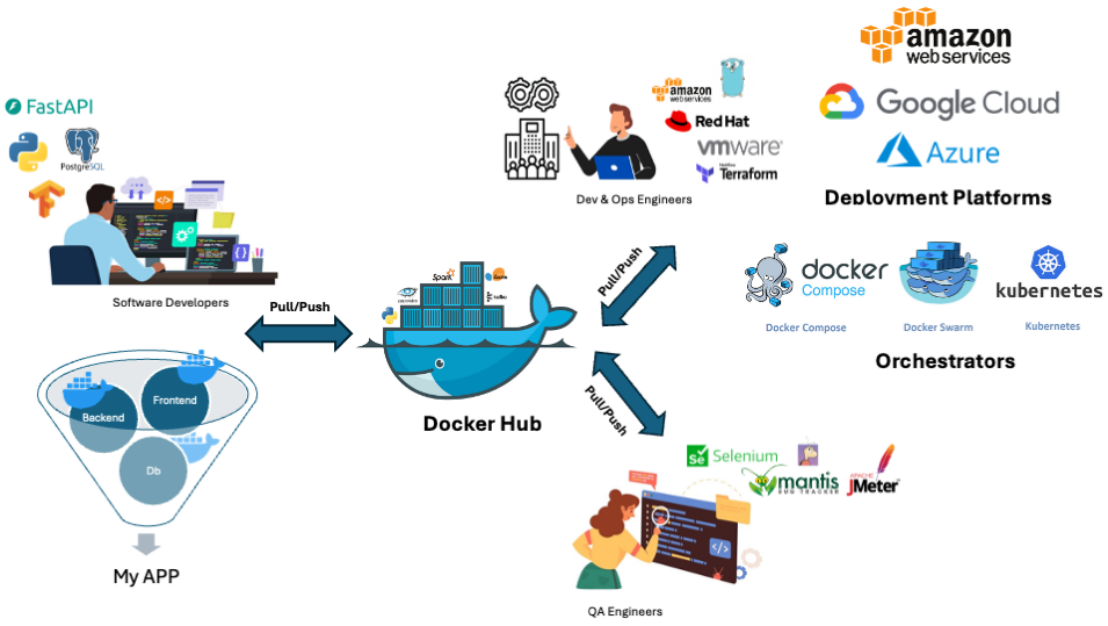

Figure 1.4: Docker Hub

1.8.1.3 Transforming my application into a container image?

Everything start by creating a `Dockerfile` that contains the instructions to build a Docker image. The Docker image is a lightweight, standalone, and executable package that includes everything needed to run a piece of software, including the code, runtime, libraries, environment variables, and configuration files. The Docker image is built using the `docker build` command, which reads the `Dockerfile` and executes the instructions to create the image. The Docker image is then stored in a registry, such as Docker Hub, Amazon ECR, or Google Container Registry, where it can be shared and distributed with others. The Docker image can be run as a container using the `docker run` command, which creates an instance of the image and runs it as a container. The Docker container is a running instance of the image that can be managed and monitored using the Docker engine. The Docker container can be stopped, started, paused, and deleted using the `docker stop`, `docker start`, `docker pause`, and `docker rm` commands, respectively. The Docker container can also be managed and monitored using tools and platforms, such as Docker Compose, Docker Swarm, and Kubernetes.

Dockerfile example

```
# Use an official Python runtime as a parent image
FROM python:3.8-slim
```

```
# Set the working directory in the container
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . /app

# Install any needed packages specified in requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this container
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

1.8.2 How does Docker work?

Under the hood, Docker uses a client-server architecture, where the Docker client communicates with the Docker daemon, which is responsible for building, running, and distributing containers. The Docker client and daemon can run on **the same system** or on **different systems**, and they communicate with each other using a **REST API** over a Unix socket or a network interface. As it was mentioned, the Docker daemon is responsible for managing the containers, images, volumes, networks, and other resources of the system. However, it also provides a high-level API for interacting with the Docker engine, allowing developers to build, run, and manage containers using simple commands and scripts. The Docker daemon is also responsible for managing the lifecycle of containers, including creating, starting, stopping, pausing, and deleting containers, as well as managing their resources, such as CPU, memory, and storage.

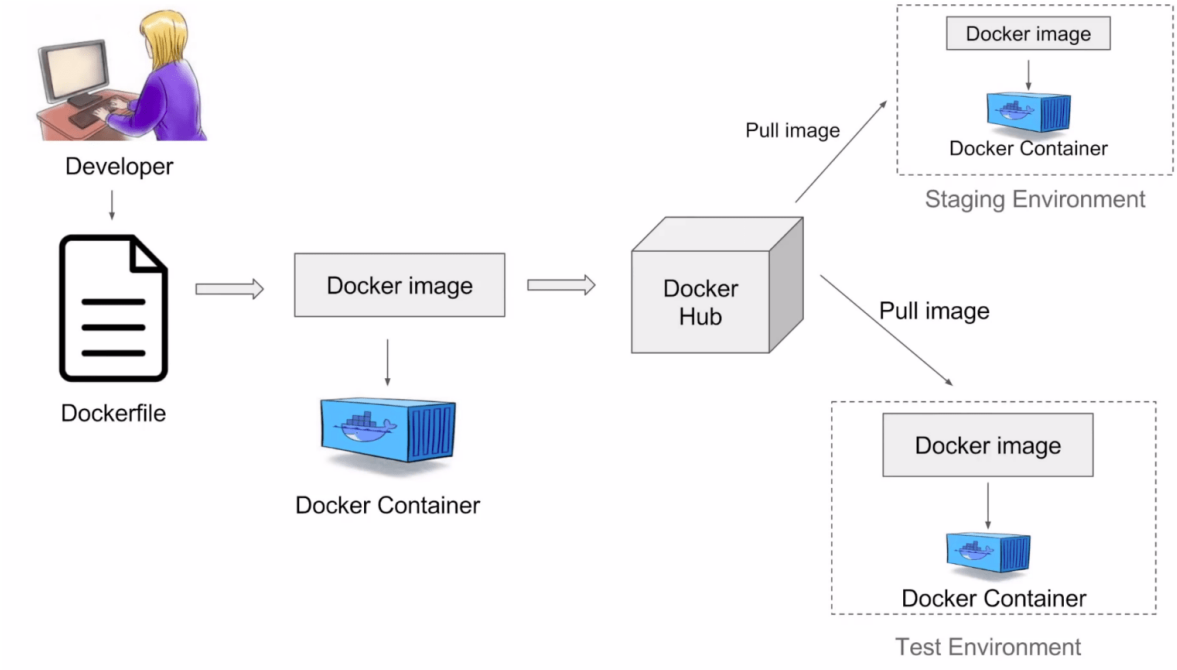

Figure 1.5: Docker

1.8.2.1 The underlying technology

Docker is written in the `Go programming language` and takes advantage of several features of the Linux kernel to deliver its functionality. The Linux kernel provides the core features and capabilities of the Docker engine, such as process isolation, resource management, and networking.

Linux kernel features that Docker relies on include:

- **Cgroups (control groups)** provide the ability to limit and prioritize the resources of containers, such as CPU, memory, and storage.
- **Namespaces** provide the ability to isolate and control the processes, users, and network of containers, ensuring that they do not interfere with each other.
- **Union file systems** provide the ability to create and manage the file systems of containers, allowing them to share and reuse the same files and directories.

> Tip
>
> The Linux kernel is the main component of the Linux operating system (OS). It's a computer program that acts as the interface between a computer's hardware and its processes. The kernel manages resources as efficiently as possible and enables communication

between applications and hardware.

So the question you probably have, how can Docker run containers on Windows and MacOS if Docker relies on the Linux kernel?

On Windows you can run Docker containers using the following approaches:

Windows Subsystem for Linux (WSL) 2: With the introduction of WSL 2, Docker can run Linux containers natively on Windows. WSL 2 provides a full Linux kernel built into Windows, allowing Docker to interface directly with the kernel without the need for a virtual machine (VM). This approach is efficient and integrates well with Windows environments.

Docker Desktop for Windows: Before WSL 2, Docker Desktop for Windows used a lightweight VM to host a Linux kernel. This VM then runs the Docker Engine and, by extension, Docker containers.

On macOS, docker also utilizes a lightweight virtual machine to run a Linux kernel. Docker Desktop for Mac leverages macOS's native virtualization frameworks (such as Hypervisor.framework for Intel processors and the Virtualization framework for Apple silicon) to run this VM efficiently. This setup allows Docker containers to run in a Linux-like environment on Mac, with Docker Desktop handling the complexities of managing the VM.

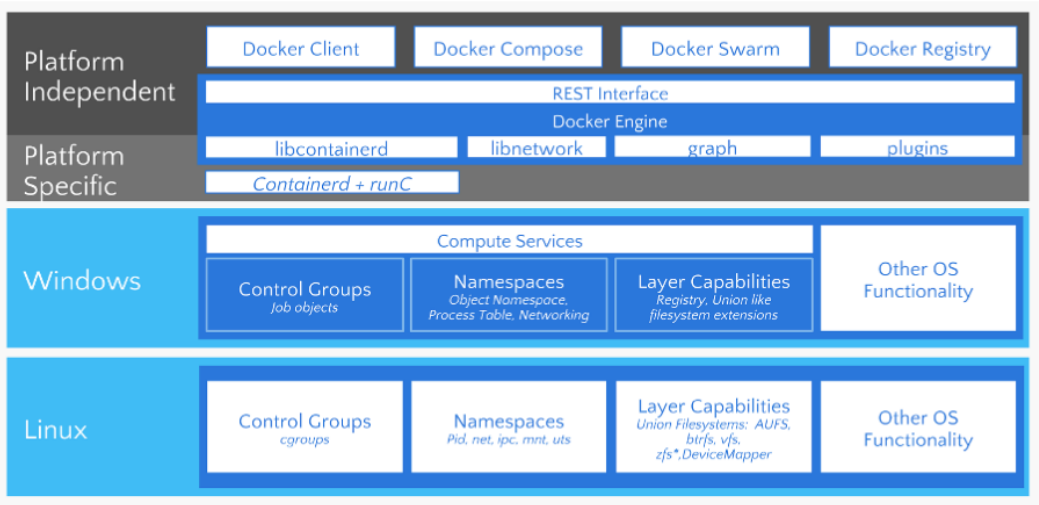

Figure 1.6: Docker on Linux and Windows

Docker was designed to run Linux-based docker containers, as it was developed on top of some of the Linux kernel features. However, Microsoft offers four container-based Windows images from which users can build. Each base image is a different type of the Windows or Windows Server operating system, has a different on-disk footprint, and has a different set of the Windows API set.All Windows container base images are discoverable through Docker Hub. The Windows container base images themselves are served from mcr.microsoft.com, the Microsoft Container Registry (MCR). This is why the pull commands for the Windows container base images look like the following:

```
docker pull mcr.microsoft.com/windows/servercore:ltsc20229
```

For more information, you can check the official documentation.

Most common Docker commands

```
# Pull an image from Docker Hub
docker pull <image-name>

# List all images on the system
docker images

# Run a container from an image
docker run <image-name>

# List all running containers
docker ps

# List all containers
docker ps -a

# Stop a running container
docker stop <container-id>

# Start a stopped container
docker start <container-id>

# Remove a container
docker rm <container-id>

# Remove an image
docker rmi <image-name>
```

1.8.2.2 Running Multi-Container Applications

One of the challenges of running multi-container applications is managing the dependencies and interactions between them. Docker Compose, Swarm and Kubernetes they are all tools that can be used to define, configure, and run multi-container applications.

- **Docker Compose** is a tool for defining and running multi-container applications on a single host. It is easy to use and requires minimal setup, making it a popular choice for developers who want to quickly set up and test their applications locally. Docker Compose provides a simple and convenient way to define, configure, and run multi-container applications, but it is limited to a single host and does not provide the same level of scalability and resource management as Kubernetes.

```yaml
version: '3'
services:
    frontend:
        image: frontend:latest
        ports:
        - "80:80"
        depends_on:
        - backend
    backend:
        image: backend:latest
        ports:
        - "8080:8080"
        environment:
        - DATABASE_URL=postgres://user:password@db:5432/db
    db:
        image: postgres:latest
        environment:
        - POSTGRES_USER=user
        - POSTGRES_PASSWORD=password
        - POSTGRES_DB=db
networks:
    default:
        external:
        name: my-network
```

- **Kubernetes**, on the other hand, is a production-ready platform for deploying, scaling, and managing containerized applications. It provides a powerful and flexible architecture for managing multi-container applications at scale, and it includes features for automatic scaling, rolling updates, self-healing, and resource management. Kubernetes is designed

for large, complex, and mission-critical applications, and it provides a high degree of availability and resilience.

https://www.youtube.com/watch?v=BE77h7dmoQU&t=604s&ab_channel=Honeypot

- **Docker Swarm** is a native clustering and orchestration tool for Docker. It allows you to create and manage a cluster of Docker hosts, and it provides features for scaling, load balancing, and service discovery. Docker Swarm is easy to use and integrates well with Docker, making it a good choice for developers who want to manage multi-container applications without the complexity of Kubernetes.

1.9 Scaling it to meet the demands of its users

When deploying a system, it is essential to consider how the system will scale to meet the demands of its users. Scalability describes the ability of a system to handle an increasing amount of work without compromising its performance, reliability, and availability. It is also related to system elasticity, which is the ability of a system to adapt to changes in the workload by adding (scaling up) or removing (scaling out) resources. Systems can be scaled in two ways: `vertically` and `horizontally`.

1.9.1 Horizontal Scaling (Scaling Out/In)

Horizontal scaling involves adding more machines or nodes to a pool of resources to manage increased load. It's like adding more lanes to a highway to accommodate more traffic. This approach is common in distributed systems, such as cloud computing environments, where you can add more instances or servers to handle more requests.

Advantages:

- **Scalability:** It's easier to scale applications indefinitely by simply adding more machines into the existing infrastructure.
- **Flexibility:** You can scale the system up or down by adding or removing resources as demand changes, often automatically.
- **Fault Tolerance:** Horizontal scaling can improve the reliability and availability of a system. If one node fails, others can take over, reducing the risk of system downtime.

Disadvantages:

- **Complexity:** Managing a distributed system with many nodes can be more complex, requiring sophisticated software and tools for load balancing, distributed data management, and failover mechanisms.

- **Data Consistency:** Ensuring data consistency across nodes can be challenging, especially in databases or systems requiring real-time synchronization.

1.9.2 Vertical Scaling (Scaling Up/Down)

Vertical scaling involves increasing the capacity of an existing machine or node by adding more resources to it, such as CPU, RAM, or storage. It's akin to upgrading the engine in a car to achieve higher performance.

Advantages:

- **Simplicity:** It is often simpler to implement as it may require just upgrading existing hardware. It doesn't involve the complexity of managing multiple nodes.
- **Immediate Performance Boost:** Upgrading hardware can provide an immediate improvement in performance for applications that can utilize the extra resources.

Disadvantages:

- **Limited Scalability:** There is a physical limit to how much you can upgrade a single machine, and eventually, you might hit the maximum capacity of what a single server can handle.
- **Downtime:** Upgrading hardware might require downtime, which can be a significant drawback for systems that require high availability.
- **Cost:** Beyond certain points, vertical scaling can become prohibitively expensive as high-end hardware components can cost significantly more.

The choice between horizontal and vertical scaling depends on the specific requirements, architecture, and constraints of the system in question. Horizontal scaling is favored for applications designed for cloud environments and those requiring high availability and scalability. Vertical scaling might be chosen for applications with less demand for scalability or where simplicity and immediate performance improvement are prioritized. Often, a hybrid approach is used, combining both strategies to leverage the advantages of each.

Scaling your system is something you can control,and plan ahead with the support of your infrastructure team. However, doing this manually is rather time-consuming, especially when the increased load only sustains for a short period of time. In other words, you're always too late. This is where autoscaling comes in, by automatically scaling either horizontally or vertically when the current incoming load requires it.

1.9.3 Autoscaling

Auto-scaling, or automatic scaling, is a technique that dynamically adjusts the amount of computational resources in a server farm or a cloud environment based on the current demand. It is closely related to both horizontal and vertical scaling, but it primarily leverages horizontal scaling due to its flexibility and the ease with which resources can be added or removed in cloud-based environments. I encorage you to check the next video to understand how Kubernetes relies on autoscaling to manage the resources of your system in cloud or on-premises environments.

https://www.youtube.com/watch?v=XpeAITE4uqA&ab_channel=GoogleCloudTech

1.9.4 Monitoring and Logging

Monitoring and logging are essential for understanding the behavior and performance of a system in a production environment. It involves collecting and analyzing data about the system's performance, availability, and reliability, while logging involves recording and storing data about the system's activities, events, and errors. Monitoring and logging are crucial for identifying and diagnosing issues, optimizing performance, and ensuring the system meets its service level objectives (SLOs) and service level agreements (SLAs). There are several tools and platforms that can be used for monitoring and logging. The selection of the appropriate tools and platforms depends on the requirements and constraints of the system.

2 Terminals, Shells and Command Line Tools

2.1 Terminals

A `terminal` is a text-based interface software that allows you to interact and give instructions to your computer from a nongraphical interface. Every operating system generally has its implementation of a terminal and comes with a set of `built-in` commands to perform various tasks.For example, in Windows, you can use the command prompt or PowerShell; in Mac and Linux, you can use the terminal. In the open-source world, there are also other alternatives, such as `oh-my-zsh, tmux, fish,` etc.

Some of the most common commands you can run in a terminal are:

- `cd` - change directory.
- `ls` - list files and directories.
- `mkdir` - create a new directory.
- `rm` - remove files or directories.
- `cp` - copy files or directories.
- `mv` - move files or directories.
- `cat` - concatenate and display files.
- `grep` - search for a specific pattern in a file.

Some of those commands are standard across all operating systems, while others are specific to a particular operating system.For example, the `ls` command is used to list files and directories in Linux and macOS, while the `dir` command is used for the same purpose in Windows.

2.2 Command Line Tools

A `command line tool`, is a software you install in your computer that gives you access to an extra set of commands to perform specific tasks. Command line tools are widely used for a variety of purposes, including file management, software development, system administration, and network operations. Some examples of command line tools are:

- `git` - a distributed version control system for tracking changes in source code during software development.
- `npm` - a package manager for the JavaScript programming language.

- `pip` - a package installer for Python.
- `brew` - a package manager for macOS.

2.3 Some useful links

- Command Line Interface
- Command Line Tools
- Terminal (macOS)
- Command Prompt
- PowerShell
- oh-my-zsh
- fish
- windows commands cheat sheet
- linux commands cheat sheet
- macOS commands cheat sheet

3 Python for ML & Data Science

3.1 Introduction

Programming is an essential skill for data scientists. If you are considering starting a data science career, the sooner you learn how to code, the better it will be. Most data sciences jobs rely on programming to automate cleaning and organizing data sets, design databases, fine-tune machine learning algorithms, etc. Therefore, having some experience in programming Languages such as Python, R, and SQL makes your life easier and will allow you to automate your analysis pipelines.

In this section, we will focus on Python. A general-purpose programming language that allows us to work with data and explore different algorithms and techniques that would be extremely useful to add to our analysis toolbox.

3.1.1 Why should I learn how to program?

A data scientist is a technical expert who uses mathematical and statistical techniques to manipulate, analyze, and extract patterns from raw or noisy data to produce valuable information that can help organizations make better decisions. They use a range of tools, including statistical inference, pattern recognition, machine learning, deep learning, and more, and some of their responsibilities include:

- Work closely with business stakeholders to understand their goals and determine how data can be used to achieve them.

- Fetching information from various sources and analyzing it to get a clear understanding of how an organization performs
- Undertaking data collection, preprocessing, and analysis
- Building models to address business problems
- Presenting information in a way that your audience can understand using different data visualization techniques

Programming skills provide data scientists with the superpowers to automate these tasks. Although programming is not required to be a data scientist, taking advantage of the power of computers, it can facilitate the process of manipulating, processing, and analyzing big

datasets, automate and develop computational algorithms to produce results (faster and more effectively), and create neat visualizations to present the data more intuitively.

3.1.2 Programming languages for data science

There are hundreds of programming languages out there, built for diverse purposes. Some are better suited for web or mobile development, others for data analysis, etc. Choosing the correct language to use will depend on your level of experience, role, and/or project goals. In the last few years, Python has been ranked as one of the top programming languages data scientists use to manipulate, process, and analyze big datasets.

But why is Python so popular? Well, I will list some reasons why data scientists love Python and what makes this language suitable for high productivity and performance in processing large amounts of data.

3.2 Why Python?

In the 2022 Stack Overflow Developer Survey, Python emerged as one of the most commonly used programming languages worldwide. Out of 71,467 responses, 68% of developers stated their love for the language and their intention to continue working with it. Additionally, around 12,000 respondents expressed their interest in learning and using Python. Python's immense popularity stems from its simple syntax, versatility, and expressiveness. If you are considering a data science project, Python offers a range of features that you may find useful. Here is a list of features that can give you an insight into why Python may be a good choice for your next project.

- Python is **open source**, so is freely available to everyone.You can even use it to develop commercial applications.
- Python is **Multi-Platform**. It can be run on any platform, including Windows, Mac, Linux, and Raspberry Pi.
- Python is a **Multi-paradigm** language, which means it can be used for both object-oriented and functional programming. It comes from you writing code in a way that is easy to read and understand.
- Python is **Multi-purpose**, so you can use it to develop almost any kind of application. You can use it to develop web applications, game development, data analysis, machine learning, and much more.
- Python syntax is **easy to read** and **easy to write**. So the learning curve is low in comparison to other languages.

- Data Science **packages ecosystem**: Python also has PyPI package index,a python package repository, where you can find many useful packages (Tensorflow, pandas, NumPy, etc.), which facilitates and speeds up your project's development. In PyPI, you can also publish your packages and share them with the community. The ecosystem keeps growing fast, and big companies like Google, Facebook, and IBM contribute by adding new packages.Some of the most used libraries for data science and machine learning are:
  - Tensorflow, a high-performance numerical programming library for deep learning.
  - Pandas, a Python library for data analysis and manipulation.
  - NumPy, a Python library for scientific computing (that offers an extensive collection of advanced mathematical functions, including linear algebra, Fourier transforms, random number generation, etc.)
  - Matplotlib, a Python library for plotting graphs and charts.
  - Scikit-learn, a Python library for machine learning.
  - Seaborn, a Python library for statistical data visualization.

  > **Note**
  >
  > The Python Package Index, abbreviated as PyPI and also known as the Cheese Shop (a reference to the Monty Python's Flying Circus sketch "Cheese Shop"), is the official third-party software repository for Python. It is analogous to the CPAN repository for Perl and to the CRAN repository for R.[1]

- **High performance:** Although some people complain about performance in Python (see Why Python is so slow and how to speed it up), mainly caused by some features such as dynamic typing, it is also simple to extend developing modules in other compiled languages like C++ or C which could speed up your code by 100x.

After having a brief overview of Python, let's move on to the next sections, where we will learn how to install Python and how to use it to perform some basic operations.

3.3 Python Installation

To check if Python is already installed on our machines, open a terminal in your computer and type the command `Python --version` or `Python3 --version`. You will see the Python version if it is installed. Otherwise, you will get an error `command not found` or similar. If you dont have Python installed on your computer, the most straightforward way to do so is to download it from the official website. Although this is a simple process, some tools such as `pyenv` and `anaconda` enable you to run multiple versions of Python on the same machine so you can switch between versions of Python according to your project's requirements. In the code examples presented in this material, we will use `Pyenv` to manage our Python installations.

```
!python --version
```

```
Python 3.10.9
```

3.4 Pyenv

Pyenv is a command line tool that enables you to have and operate multiple installations of Python on the same machine. If you come from a background in javascript, you may find that `pyenv` is similar to `nvm (Node Version Manager)`. We suggest referring to the official documentation for instructions on how to install pyenv. Alternatively, if you're using Windows, you can use pyenv-win. However, we'll provide a brief summary of the installation process here.

```
# Install pyenv
curl https://pyenv.run | bash
```

After having installed pyenv, you can then install any python version running the command `pyenv install <version>`. For example, to install Python 3.9.7, you would run `pyenv install 3.9.7`. You can then set the global version of Python to be used by running `pyenv global 3.9.7`. You can also set the local version of Python to be used in a specific directory by running `pyenv local 3.9.7`. The global version of Python is the version that will be used by default in your machine, while the local version is the version that will be used in the directory where you run the command.Pyenv will automatically set the local version of Python when you enter the directory where you have set the local version.

3.4.1 Util commands

- `pyenv versions`: List all the versions of Python installed on your machine.
- `pyenv global`: Show the global version of Python.
- `pyenv local`: Show the local version of Python.
- `pyenv uninstall <version>`: Uninstall a specific version of Python.
- `pyenv rehash`: Rehash pyenv shims (run this command after installing a new version of Python).
- `pyenv version`: Show the current version of Python.
- `pyenv which python`: Show the path of the current Python executable.
- `pyenv which pip`: Show the path of the current pip executable.
- `pyenv help`: Show the list of available commands.
- `pyenv shell <version>`: Set the shell version of Python.

3.5 Python Dependency hell

Well, it sounds like Python is amazing! However, if you have been using Python for a while, you may have already noticed that handling different python-installations and dependencies(packages) can be a nightmare! An issue commonly known as dependency hell, which is a term associated with the frustration arising from problems managing our project's dependencies.

Dependency hell in Python often happens because pip does not have a dependency resolver and because all dependencies are shared across projects. So, other projects could be affected when a given dependency may need to be updated or uninstalled.

On top of it, since Python doesn't distinguish between different versions of the same library in the `/site-packages` directory, this leads to many conflicts when you have two projects requiring different versions of the same library or the global installation doesn't match.

Thus, having tools that enable us to isolate and manage our project's dependencies is highly convenient.

3.5.1 Virtual environments to the rescue!

Python virtual environment is a separate folder where only your project's dependencies(packages) are located. Each virtual environment has its own Python binary (which matches the version of the binary that was used to create this environment) and its own independent set of installed Python packages in its site directories. That is a very convenient way to prevent `Dependency Hell.`

> **Note**
>
> Python virtual environment allows multiple versions of Python to coexist in the same machine, so you can test your application using different Python versions. It also keeps your project's dependencies isolated, so they don't interfere with the dependencies of others projects.

There are different tools out there that can be used to create Python virtual environments. In this post, I will show you how to use pyenv and poetry. However, you can also try other tools, such as virtualenv or anaconda, and based on your experience, you can choose that one you feel most comfortable with. the video below will provide you with more information about these kinds of tools.

https://www.youtube.com/embed/3J02sec99RM

3.6 Poetry

Poetry is a tool that allows you to manage your project's dependencies and facilitates the process of packaging for distribution. It resolves your project dependencies and makes sure that there are no conflicts between them. Poetry integrates with the PyPI package index to find and install your environment dependencies, and pyenv to set your project python runtime.

To install poetry we follow the steps below:

```
# Install poetry
curl -sSL https://install.python-poetry.org | python3 -
```

3.6.1 Util commands

- `poetry new <project-name>`: Create a new project.
- `poetry new <project-name> --src`: Create a new project with a `src` directory.
- `poetry install`: Install the project dependencies from the `pyproject.toml` file.
- `poetry add <package-name>`: Add a new package to the project.
- `poetry remove <package-name>`: Remove a package from the project.
- `poetry update`: Update the project dependencies.
- `poetry run <command>`: Run a command in the project's virtual environment.
- `poetry shell`: Activate the project's virtual environment.
- `poetry build`: Build the project.
- `poetry publish`: Publish the project to PyPI.
- `poetry version`: Show the current version of poetry.
- `poetry help`: Show the list of available commands.

If you were able to run the previous commands, we can then move forward with the rest of the tutorial. Lets now then create a new project using poetry and pyenv.For this example we will create a project called `my_project` and we will use Python 3.9.7 as the project's python version. We will also add the `numpy` package to the project's dependencies.

Step by step: Creating a new project using poetry and pyenv

```
pyenv install 3.9.7 # install python 3.9.7 in your machine

mkdir my_project # create a new directory called my_project
cd my_project # enter the my_project directory

pyenv local 3.9.7 # set the local version of python to be used in this directory
poetry config virtualenvs.in-project true # set the virtual environment to be created in the
poetry init -n # create a new project with default settings
```

```
poetry add numpy # add numpy to the project's dependencies

touch main.py # create a new file called main.py
```

After running the previous commands, you will have a new project with the following structure:

```
my_project
.venv
pyproject.toml
poetry.lock
main.py
```

> **!** Important
>
> Note that if you want `poetry` to create the `virtual environment(.venv)` directory in the project's root, you must change the `virtualenvs.in-project` setting to `true` by running the command `poetry config virtualenvs.in-project true`. This command only needs to be run once, and it will be define globally for all projects.

The primary file for your poetry project is the `pyproject.toml` file. This file contains the necessary information about your project's dependencies (Python packages) and also holds the required metadata for packaging, if needed. Every time a new Python package is installed, Poetry automatically updates this file. By sharing this file with others, they can recreate the project environment and run your application. To do so, they will need to have Poetry installed on their system and run the command `poetry install` within the same folder where the `pyproject.toml` file is located.

Now our `pyproject.toml` file looks like:

```
[tool.poetry]
name = "myproject"
version = "0.1.0"
description = ""
authors = [`Henry Ruiz  <henry.ruiz.tamu@gmail.com>`]

[tool.poetry.dependencies]
python = "^3.9.7"
numpy = "^1.23.1"

[tool.poetry.dev-dependencies]
pytest = "^5.2"
```

```
[build-system]
requires = ["poetry-core>=1.0.0"]
build-backend = "poetry.core.masonry.api"
```

Lest review that file sections:

- **[tool.poetry]:** This section contains informational metadata about our package, such as the package name, description, author details, etc. Most of the config values here are optional unless you're planning on publishing this project as an official PyPi package.
- **[tool.poetry.dependencies]:** This section defines the dependencies of your project. Here is where you define the python packages that your project requires to run. We can update this file manually if it is needed.
- **[tool.poetry.dev-dependencies]:** This section defines the dev dependencies of your project. These dependencies are not required for your project to run, but they are useful for development.
- **[build-system]:** This is rarely a section you'll need to touch unless you upgrade your version of Poetry.

To see in a nicer format the dependencies of your project, you can use the command `poetry show --tree`. This command draws a graph of all of our dependencies as well as the dependencies of our dependencies. If we are not sure at some point that we have the latest version of a dependency, we can tell poetry to check on our package repository if there is a new version by using `--latest` option (`poetry show --latest`).

3.7 Python Syntax

lets open the `main.py` file and write our first Python code. We will start by printing the message "Hello, World!" to the console. To do so, we will use the `print` function as follows:

```
print("Hello, World!")
```

```
Hello, World!
```

3.7.1 Creating variables

Python is a dynamically typed language, which means that you don't need to declare the type of a variable when you create one. The type of the variable will be determined by the value assigned to it during runtime. Python has a built-in function called `type` that allows you to check the type of a variable. For example, to check the type of a variable `x`, you would write `type(x)`.

```python
a = 5 # define a variable
b = 10 # define another variable
x = a + b # assign a computation result to a variable x
print(type(x)) # check the type of x
print(id(x)) # check the memory address of x
print(x) # print the value of x
```

```
<class 'int'>
4363272880
15
```

3.7.2 Data types

Python has several built-in primitive data types, such as `int`, `float`, `str`, `bool`. In addition to these, it also has several built-in collection data types, such as `list`, `tuple`, `set`, and `dict`. We will cover these data types in more detail in the next sections.

Some useful resources

- Best Python cheat sheets
- Python Tutorial
- Python Language Reference
- Python Data Science Handbook
- Why Coding is important in Data Science
- Python for Data Science
- Top programming languages for data scientists in 2022
- Why Python is so slow and how to speed it up
- Write Your Own C-extension to Speed Up Python by 100x

4 Web Services and APIs

4.1 Introduction

The terms "web services" and "APIs" are often used interchangeably, but they are not the same. Although both are software components used to facilitate communication between different systems and applications, understanding their differences of these tools is essential to use them effectively. A web service is a software system designed to support interoperable machine-to-machine interaction over a network. An API, or Application Programming Interface, is a set of rules and protocols that allow one software application to interact with another. An API can be used to access the functionality of a web service, but it can also be used to access the functionality of a library or other software component.

As a data science or ML engineer, you will likely need to interact with web services and APIs to access data, models, and other resources in your daily job. You may not realize it, but APIs and web services are everywhere, and you're probably using them right now without noticing. For example, in the cutting-edge era of large language models (LLMs), companies such as OpenAI and Google have their own exclusive foundation models. Although these models do not offer direct access to their implementations or code for various reasons, they provide APIs that enable access to the powerful capabilities of their LLM models. By skillfully integrating these features into our applications, we can significantly enhance their functionality. The following diagram clearly depicts the seamless interplay between a web service and an API.

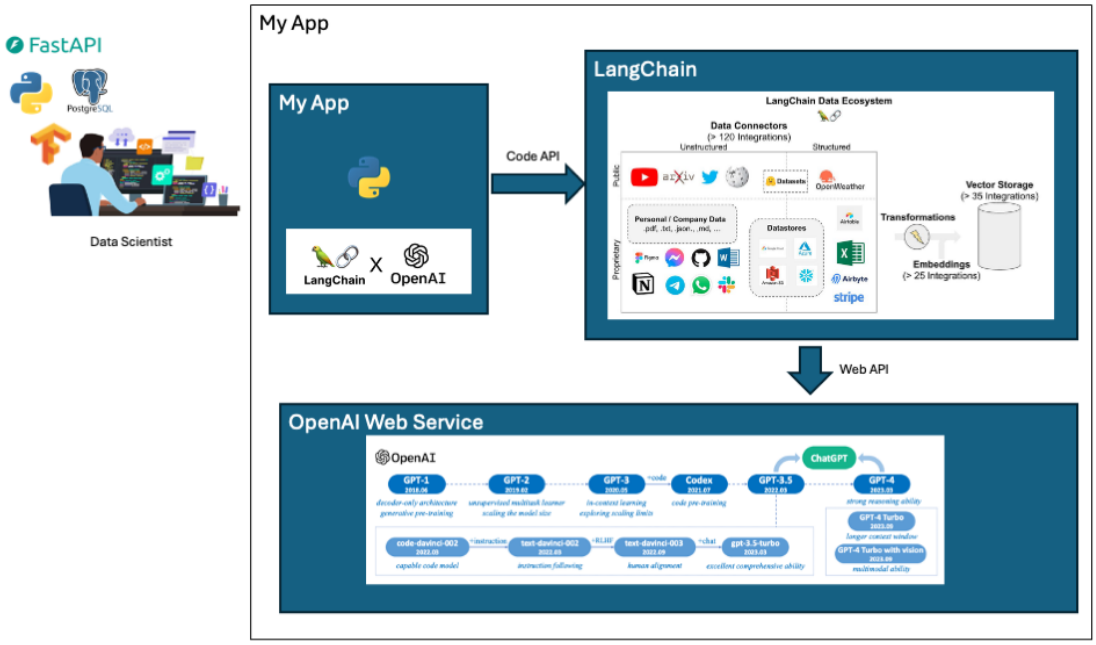

Figure 4.1: Web Services and APIs