

Lab 8



Agenda

- ❑ Gemini API
- ❑ Hugging Face
- ❑ Gradio
- ❑ REST
- ❑ Kafka
- ❑ Factory DP Brief
- ❑ Strategy DP Brief
- ❑ Static Analysis

Gemini API

Gemini API

- ❑ **Get the key:** <https://aistudio.google.com/app/apikey>
- ❑ **Call the API:** <https://ai.google.dev/gemini-api/docs>
- ❑ It is Easy

Gemini API

```
const { GoogleGenerativeAI } = require("@google/generative-ai");

const genAI = new GoogleGenerativeAI("YOUR_API_KEY");
const model = genAI.getGenerativeModel({ model: "gemini-1.5-flash" });

const prompt = "Explain how AI works";

const result = await model.generateContent(prompt);
console.log(result.response.text());
```

```
import google.generativeai as genai

genai.configure(api_key="YOUR_API_KEY")
model = genai.GenerativeModel("gemini-1.5-flash")
response = model.generate_content("Explain how AI works")
print(response.text)
```

Hugging Face

Hugging Face

```
import transformers
import torch

model_id = "meta-llama/Llama-3.3-70B-Instruct"

pipeline = transformers.pipeline(
    "text-generation",
    model=model_id,
    model_kwargs={"torch_dtype": torch.bfloat16},
    device_map="auto",
)

messages = [
    {"role": "system", "content": "You are a pirate chatbot who always responds in a piratey way."},
    {"role": "user", "content": "Who are you?"},
]

outputs = pipeline(
    messages,
    max_new_tokens=256,
)

print(outputs[0]["generated_text"][-1])
```

Gradio

Gradio

```
from transformers import AutoModelForCausalLM, AutoTokenizer
import gradio as gr

checkpoint = "HuggingFaceTB/SmolLM2-135M-Instruct"
device = "cpu" # "cuda" or "cpu"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
model = AutoModelForCausalLM.from_pretrained(checkpoint).to(device)

def predict(message, history):
    history.append({"role": "user", "content": message})
    input_text = tokenizer.apply_chat_template(history, tokenize=False)
    inputs = tokenizer.encode(input_text, return_tensors="pt").to(device)
    outputs = model.generate(inputs, max_new_tokens=100, temperature=0.2, top_p=0.9, do_sample=True)
    decoded = tokenizer.decode(outputs[0])
    response = decoded.split("<|im_start|>assistant\n")[-1].split("<|im_end|>")[0]
    return response

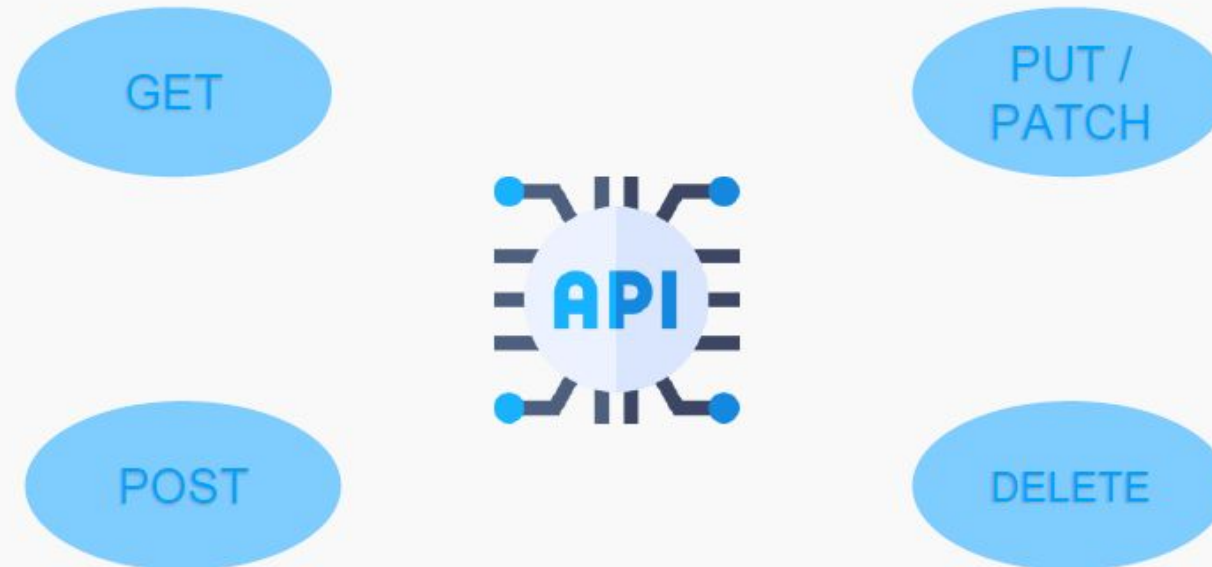
demo = gr.ChatInterface(predict, type="messages")

demo.launch()
```

REST

Recall: Web Architecture

What is REST?



How REST Works

- ❑ The client (front-end, Gradio) sends an HTTP request to the server
- ❑ The server processes the request and returns the response
- ❑ **Request Components**
 - Endpoint: The URL of the API (e.g., "https://generativelanguage.googleapis.com/v1beta/models/gemini-1.5-flash-latest:generateContent?key=YOUR_API_KEY").
 - HTTP Method: Specifies the action (e.g., POST for inference).
 - Headers: Additional information (e.g., API keys).
 - Body: The data sent to the server (e.g., input text for inference).
- ❑ The server's response is usually a JSON object

Example

```
import requests
import json

API_URL = "https://generativelanguage.googleapis.com/v1beta/models/gemini-1.5-flash-latest:generateContent"

API_KEY = "YOUR_API_KEY"

HEADERS = {
    "Content-Type": "application/json"
}

data = {
    "contents": [
        {
            "parts": [
                {"text": "Explain how AI works"}
            ]
        }
    ]
}
```

```
response = requests.post(f"{API_URL}?key={API_KEY}", headers=HEADERS, json=data)

if response.status_code == 200:
    response_data = response.json()
    print(json.dumps(response_data, indent=2))
else:
    print(f"Error: {response.status_code}, {response.text}")
```

Example Cont.

```
{
  "candidates": [
    {
      "content": {
        "parts": [
          {
            "text": "AI, or Artificial Intelligence, doesn't work in a single, unified way. Instead, it encompasses a broad range of techniques..."
          }
        ],
        "role": "model"
      },
      "finishReason": "STOP",
      "citationMetadata": {
        "citationSources": [
          {
            "startIndex": 2187,
            "endIndex": 2311,
            "uri": "https://westminsta.com/understanding-the-basics-of-artificial-intelligence/"
          }
        ]
      },
      "avgLogprobs": -0.15536086810974745
    }
  ],
  "usageMetadata": {
    "promptTokenCount": 4,
    "candidatesTokenCount": 881,
    "totalTokenCount": 885
  },
  "modelVersion": "gemini-1.5-flash-002"
}
```

Kafka

Recall: Operating System IPCs (Message Queue)

- ❑ Enables different parts of a software application, often distributed, to send and receive messages asynchronously.
- ❑ Consists of:
 - **Producers:** Send messages to the queue.
 - **Consumers:** Retrieve messages from the queue.
 - **Queue:** Acts as a buffer to store messages until they are processed.

Advantages of MQ

- ❑ **Decoupling:** Producers and consumers don't need to interact directly.
 - This infers scalability since we can scale producers or consumers independently.
- ❑ **Asynchronous Processing:** Producers can send messages and move on, while consumers process them later.
- ❑ Examples: RabbitMQ & Amazon SQS

Kafka

- ❑ A distributed event-streaming platform
- ❑ Can act as a high performance MQ
- ❑ Stores and processes streams of events in real-time
 - This makes it suitable for handling large volumes of data
- ❑ It uses the publisher-subscriber design (recall the observer DP)

Kafka Components

- ❑ Broker: A server that stores and serves data to clients
 - Role:
 - Handles incoming messages from producers
 - Manages message storage on disk
 - Serves messages to consumers
 - There are multiple brokers to ensure fault tolerance and scalability
- ❑ Topics: A category or feed name to which messages are sent by producers
 - Role:
 - Topics organize messages
 - Data is appended to a topic and distributed across partitions for parallel processing. Partitions are explained in the next slide

Kafka Components Cont.

- ❑ Partition: A topic is divided into partitions, which are subsets of the topic's data
 - Role:
 - Provides parallelism by distributing data across multiple Kafka brokers
 - Ensures scalability as partitions can be processed independently
 - Each partition is identified by a number (e.g., partition-0, partition-1)
 - Each partition stores records in a sequential, immutable log
- ❑ Producer (Publisher): A client application that sends messages to Kafka topics
 - Role:
 - Can choose the partition within a topic to which data is sent
 - Supports various data serialization formats like Avro, JSON, or Protobuf.

Kafka Components Cont.

- ❑ Consumer (Subscriber): A client application that reads messages from Kafka topics
 - Role:
 - Subscribes to one or more topics
 - Consumes messages either individually or in consumer groups
 - Consumers can specify offsets to control where reading starts (e.g., latest, earliest)
 - Multiple consumers in a group share partitions for load balancing
- ❑ Consumer Group: A collection of consumers that work together to consume messages from a topic.
 - Role:
 - Distributes the workload of consuming messages across multiple consumers
 - Each partition is assigned to only one consumer within a group
 - Provides scalability and redundancy
 - If a consumer fails, another consumer in the group takes over its partitions

Kafka vs MQ

Feature	MQ	Kafka
Message Delivery	Messages are removed from the queue once consumed	Messages are retained for a configurable time, even after consumption
Speed	Lower throughput, but lower latency for small systems	Extremely high throughput, great for big data and streaming
Scalability	Limited scalability compared to Kafka	Horizontally scalable to handle massive workloads
Use Case	Ideal for point-to-point or task-based systems	Best for event streaming, logs, and real-time analytics

Kafka vs REST

Feature	REST	Kafka
Communication Style	Synchronous (client waits for a response)	Asynchronous (messages are sent and processed later)
Interaction Model	Request-response	Publish-subscribe
Scalability	Handles limited, stateless requests	High scalability for handling large-scale data streams
Real-Time Processing	Not designed for real-time, high-volume data	Real-time processing of events and data streams
Data Persistence	Stateless (data is not stored by default)	Messages are retained for a configurable period

When To Use Each?

❑ REST:

- For CRUD operations where clients need an immediate response
- When interacting with a user-facing application (e.g., submitting a form, getting a prediction from an AI model)
- For stateless operations where the server does not need to retain the data
- You send one AI request (prompt) to Hugging Face via REST, and the model returns the output immediately.

❑ When to Use Kafka/Message Queues:

- For real-time event streaming (e.g., IoT sensor data, logs, analytics)
- For asynchronous task processing (e.g., queuing jobs for background processing)
- To decouple systems (e.g., allowing producers and consumers to work independently)
- You send a batches of AI requests (prompts) to a topic where multiple consumers process them in parallel

Kafka Example Producer (Publisher)

```
from kafka import KafkaProducer
import json

producer = KafkaProducer(bootstrap_servers='localhost:9092',
                           value_serializer=lambda v: json.dumps(v).encode('utf-8'))

message = {
    "input": "Explain how AI works"
}

# Send a message to the 'ai-requests' topic
producer.send('ai-requests', message)
print("Message sent!")
```

Kafka Example Consumer (Subscriber)

```
from kafka import KafkaConsumer
import json

consumer = KafkaConsumer('ai-requests',
                          bootstrap_servers='localhost:9092',
                          auto_offset_reset='earliest',
                          enable_auto_commit=True,
                          value_deserializer=lambda x: json.loads(x.decode('utf-8')))

for message in consumer:
    data = message.value
    print(f"Received message: {data}")
    # Simulate processing the AI request
```

Factory DP Brief

Overview

- ❑ Provides an interface for creating objects without specifying their concrete classes.
- ❑ It allows for dynamic object creation, which makes the system more flexible and maintainable
- ❑ Features:
 - **Factory Class:** A class that contains the logic to decide which object to create
 - **Encapsulation:** The object creation logic is encapsulated, so any changes to the creation logic do not affect the client code
 - **Decoupling:** The client is decoupled from the actual object creation, enabling flexibility and better maintainability
- ❑ **Creational** Design Pattern

Designing Our System With Factory DP

- ❑ Dynamically create the appropriate AI model client (e.g., Hugging Face, Gemini) based on a user's request or configuration
- ❑ Centralize the object creation process so that adding new models (e.g., OpenAI) doesn't require changes in multiple places

Designing Our System With Factory DP Cont.

```
class AIModelClient:
    def infer(self, input_text):
        pass

class HuggingFaceClient(AIModelClient):
    def infer(self, input_text):
        print(f"Using Hugging Face API to infer: {input_text}")

class GeminiClient(AIModelClient):
    def infer(self, input_text):
        print(f"Using Gemini API to infer: {input_text}")

# Factory class
class AIModelFactory:
    def create_model(self, model_type):
        if model_type == "huggingface":
            return HuggingFaceClient()
        elif model_type == "gemini":
            return GeminiClient()
        else:
            raise ValueError(f"Unknown model type: {model_type}")
```

```
factory = AIModelFactory()

# Dynamically create the model client
model_client = factory.create_model("huggingface")
model_client.infer("Explain how AI works")

model_client = factory.create_model("gemini")
model_client.infer("Describe AI applications in healthcare")
```

Strategy DP Brief

Overview

- ❑ Defines a family of algorithms (or behaviors), encapsulates each one, and makes them interchangeable at runtime
- ❑ The client interacts with the selected algorithm through a common interface, without needing to know the internal details
- ❑ Features:
 - **Strategy Interface:** A common interface that defines the behavior (e.g., infer in our AI system which is shown in the next slide)
 - **Concrete Strategies:** Different implementations of the strategy interface (e.g., Hugging Face inference, Gemini inference)
 - **Context Class:** A class that maintains a reference to a strategy and interacts with it
- ❑ **Behavioral** Design Pattern

Designing Our System With Strategy DP

- ❑ Define different inference mechanisms (e.g., REST API calls, Kafka-based batch processing)
- ❑ Allow the system to switch between strategies dynamically based on user requirements or workload

Designing Our System With Strategy DP Cont.

```
class InferenceStrategy:
    def infer(self, input_text):
        pass

class HuggingFaceStrategy(InferenceStrategy):
    def infer(self, input_text):
        print(f"Calling Hugging Face API for: {input_text}")

class GeminiStrategy(InferenceStrategy):
    def infer(self, input_text):
        print(f"Calling Gemini API for: {input_text}")

class AIInferenceContext:
    def __init__(self, strategy: InferenceStrategy):
        self.strategy = strategy

    def set_strategy(self, strategy: InferenceStrategy):
        self.strategy = strategy

    def execute_inference(self, input_text):
        self.strategy.infer(input_text)
```

```
# Start with Hugging Face strategy
context = AIInferenceContext(HuggingFaceStrategy())
context.execute_inference("Explain how AI works")

# Switch to Gemini strategy
context.set_strategy(GeminiStrategy())
context.execute_inference("Describe AI's role in education")
```

Using Both Factory & Strategy DPs

- ❑ Factory:
 - Handles the creation of the appropriate AI model or strategy based on input
 - It will create a HuggingFaceStrategy or GeminiStrategy
- ❑ Strategy:
 - Encapsulates the behavior for making the inference (e.g., REST API call, Kafka-based processing)
 - It will determines how to call the AI model

Using Both Factory & Strategy DPs Cont.

```
class InferenceStrategy:
    def infer(self, input_text):
        pass

class HuggingFaceStrategy(InferenceStrategy):
    def infer(self, input_text):
        print(f"Calling Hugging Face API for: {input_text}")

class GeminiStrategy(InferenceStrategy):
    def infer(self, input_text):
        print(f"Calling Gemini API for: {input_text}")

# Factory to create strategies
class AIModelFactory:
    def create_strategy(self, model_type):
        if model_type == "huggingface":
            return HuggingFaceStrategy()
        elif model_type == "gemini":
            return GeminiStrategy()
        else:
            raise ValueError(f"Unknown model type: {model_type}")
```

```
# Context for managing strategy
class AIInferenceContext:
    def __init__(self, strategy: InferenceStrategy):
        self.strategy = strategy

    def set_strategy(self, strategy: InferenceStrategy):
        self.strategy = strategy

    def execute_inference(self, input_text):
        self.strategy.infer(input_text)

factory = AIModelFactory()

# Create strategy using factory
strategy = factory.create_strategy("huggingface")
context = AIInferenceContext(strategy)
context.execute_inference("What is AI?")

# Switch strategy
strategy = factory.create_strategy("gemini")
context.set_strategy(strategy)
context.execute_inference("Describe AI's role in healthcare")
```

Static Analysis

What is Static Analysis

- ❑ A method of debugging by examining source code without executing it
- ❑ Provides insights into the code's structure and ensures adherence to industry standards
- ❑ Usually happens before the testing phase
- ❑ Key Aspects Include:
 - **Early Detection of Errors:** Identifies syntax errors, type mismatches, and other coding mistakes that might not be immediately apparent
 - **Security Vulnerabilities:** Detects common security flaws such as SQL injection points, cross-site scripting (XSS) vulnerabilities, and buffer overflows
 - **Code Quality and Maintainability:** Highlights code structures that can be refactored for better maintainability
 - **Compliance and Standards Enforcement:** Ensures code adheres to industry standards, coding guidelines, and regulatory requirements
 - **Performance Issues:** Identifies inefficient code that could lead to performance bottlenecks

Tools For Static Analysis

- ❑ A method of debugging by examining source code without executing it
- ❑ Provides insights into the code's structure and ensures adherence to industry standards
- ❑ Usually happens before the testing phase
- ❑ Key Aspects Include:
 - **Early Detection of Errors:** Identifies syntax errors, type mismatches, and other coding mistakes that might not be immediately apparent
 - **Security Vulnerabilities:** Detects common security flaws such as SQL injection points, cross-site scripting (XSS) vulnerabilities, and buffer overflows
 - **Code Quality and Maintainability:** Highlights code structures that can be refactored for better maintainability
 - **Compliance and Standards Enforcement:** Ensures code adheres to industry standards, coding guidelines, and regulatory requirements
 - **Performance Issues:** Identifies inefficient code that could lead to performance bottlenecks
- ❑ IDEs have built-in basic static analysis tools

How it Works

❑ **Lexical Analysis:**

- The static analyzer reads the code line by line, breaking it down into tokens (e.g., keywords, variables, symbols)
- It ensures that the code follows the correct syntax for the programming language
- Example:
 - Analyzing whether every opening { has a corresponding closing }

❑ **Syntax Checking:**

- It checks whether the code adheres to the grammar rules of the programming language
- Errors like missing semicolons, misplaced operators, or unmatched parentheses are flagged
- Example:
 - `print("Hello" # Missing closing parenthesis triggers an error`

How it Works Cont.

❑ Semantic Analysis:

- It evaluates the logic and meaning of the code, such as:
 - Whether variables are used without being declared
 - Whether functions are called with the correct number and type of arguments
 - Whether operations are valid (e.g., dividing by zero or adding incompatible types)
- Example:
 - `result = 10 / 0` # Static analysis flags this as a "divide-by-zero" error.

❑ Control Flow Analysis:

- Examines the possible paths the code could take during execution to identify unreachable or dead code
- Ensures that all branches of a conditional statement are logically sound
- Example:
 - `if False:`
 - `print("This will never execute")` # Flags unreachable code.

How it Works Cont.

❑ **Data Flow Analysis:**

- Tracks the flow of data through variables and functions to identify:
 - Unused variables: Variables declared but never used
 - Uninitialized variables: Variables used without being assigned a value
 - Incorrect data flow: Passing data where it's not needed or incorrect
- Example:
 - `x = 5`
 - `print(y)` # Flags 'y' as uninitialized.

❑ **Dependency Analysis:**

- Scans external libraries or dependencies to check for:
 - Deprecated APIs.
 - Known vulnerabilities in imported packages.
- Example:
 - Detecting outdated versions of numpy that have security issues.

How it Works Cont.

❑ **Pattern Matching**

- Uses predefined patterns for identifying common coding errors or vulnerabilities
- Example:
 - Searching for hardcoded passwords:
 - `password = "1234"` # Static analysis flags this as insecure.

❑ **Rule-Based Checking**

- Compares the code against coding standards (e.g., PEP 8 for Python, MISRA for C++) or organizational guidelines
- Example:
 - Enforcing naming conventions, such as variable names starting with lowercase letters

Static Analysis Cycle

- ❑ **Code Review:** After writing code, a static code analyzer scans the codebase, checking against defined coding rules from standards or custom guidelines
- ❑ **Issue Identification:** The analyzer identifies whether the code complies with the set rules, flagging potential issues
- ❑ **False Positives Management:** Developers review flagged issues to dismiss any false positives
- ❑ **Issue Resolution:** Developers address the identified issues, starting with the most critical ones
- ❑ **Testing:** Once issues are resolved, the code proceeds to testing phases

Static vs Dynamic Analysis

Aspect	Static Analysis	Dynamic Analysis
Execution	Analyzes code without running it.	Requires running the code or application.
Focus	Syntax, semantics, and code structure.	Runtime behavior (e.g., memory usage, crashes).
Stage of Use	Performed during development or build phase.	Performed during testing or production.
Examples	Tools like SonarQube, ESLint, Pylint.	Tools like Selenium, Postman, JMeter.

SonarQube Features

❑ **Multi-Language Support:**

- Supports over 25 programming languages, including Java, Python, JavaScript, C#, and more

❑ **Quality Gates:**

- Ensures code adheres to predefined quality standards before it moves to the next stage

❑ **Security Analysis:**

- Detects vulnerabilities and security hotspots based on standards like OWASP and CWE

❑ **Code Coverage:**

- Integrates with testing frameworks to measure and track test coverage

SonarQube Features Cont.

❑ **Continuous Integration:**

- Integrates with CI/CD pipelines like Jenkins, GitLab, or GitHub Actions to automate code reviews

❑ **Customizable Rules:**

- Allows teams to define and enforce their coding standards

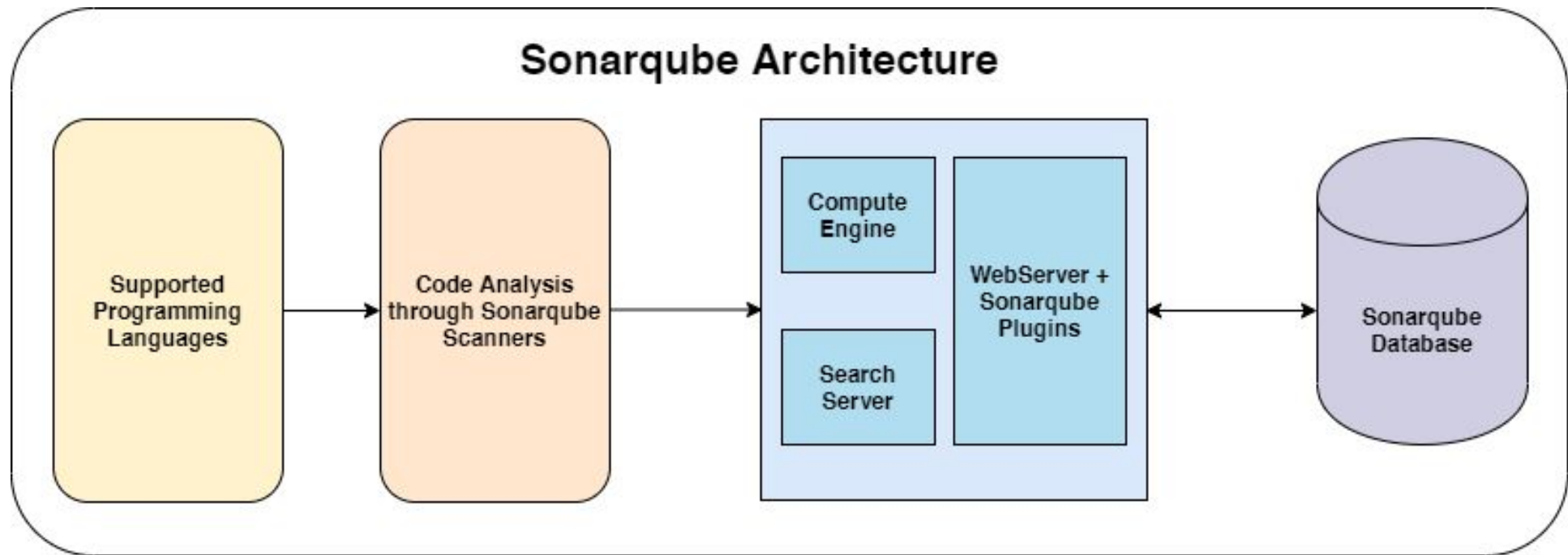
❑ **Visualization:**

- Provides dashboards and detailed reports for monitoring code quality trends over time

SonarQube Architecture

- ❑ **Scanner:**
 - Analyzes the source code and generates a report of issues, vulnerabilities, and metrics
- ❑ **Server:**
 - Central server where analysis results are stored and visualized
- ❑ **Database:**
 - Stores analysis results and history for long-term tracking
- ❑ **Web Interface:**
 - Allows developers to explore issues, view metrics, and monitor quality trends

SonarQube Architecture



SonarQube Metrics

- ❑ Most notable metrics are:
 - **Bugs:** Code that produces unexpected behavior
 - **Vulnerabilities:** Security weaknesses exploitable by attackers
 - **Code Smells:** Maintainability issues that make code harder to understand or modify
 - **Test Coverage:** Percentage of code covered by automated tests
 - **Duplications:** Repeated code that could be refactored
 - **Technical Debt:** Effort required to fix code issues

Thank
you