Distributed LLM for UML

AI-powered UML diagram tool

San Jose State University

Prof. Charan Bhaskar

CMPE 273 : Enterprise Distributed Systems

Pratik Mali, Vaibhav Dnyandeo Ingale, Sagar Shashikant Sutar

Spring 2023

1 Introduction

The Future of UML Diagramming: A Distributed Large Language Model Approach

In the world of software development, UML (Unified Modeling Language) diagrams are a critical tool for visualizing complex systems and their components. These diagrams help developers and stakeholders to better understand the structure and behavior of a software system, and they are essential for communication, design, and analysis.

Traditionally, UML diagrams have been created manually using specialized tools. This process can be time-consuming and error-prone, and it can be difficult to keep diagrams up-to-date as a system evolves.

However, recent advances in artificial intelligence and natural language processing (NLP) are revolutionizing the way UML diagrams are created. By integrating a distributed large language model (LLM) with a UML diagramming tool, developers can now create accurate and detailed diagrams more efficiently than ever before.

A distributed LLM, such as the GPT-3.5 architecture developed by OpenAI, possesses an extensive understanding of human language and a vast knowledge base. By incorporating such a model into a UML diagramming tool, developers gain access to a highly intelligent assistant that can assist in generating UML diagrams automatically or provide real-time suggestions during the diagram creation process.

The integration of a large language model with a UML diagramming tool offers several advantages. First and foremost, it reduces the manual effort required to create UML diagrams, as the tool can generate diagrams based on textual descriptions or natural language queries. This automation significantly speeds up the diagramming process, allowing developers to focus more on system design and analysis.

Moreover, the distributed nature of the large language model ensures that it continuously learns from a wide range of sources and stays up to date with the latest software development practices and trends. This enables the UML diagramming tool to provide accurate and relevant suggestions and recommendations, improving the quality and correctness of the diagrams produced.

In this paper, we will explore the capabilities and benefits of a UML diagramming tool integrated with a distributed large language model. We will delve into the technical aspects of the integration, examining how natural language processing techniques are employed to facilitate diagram generation and enhance the user experience. Furthermore, we will discuss real-world use cases and scenarios where such a tool can prove invaluable in accelerating software development processes and fostering collaboration among developers and stakeholders.

Overall, the fusion of a UML diagramming tool with a distributed large language model marks a significant leap forward in software system visualization. With its ability to understand and interpret natural language, this integration empowers developers to create UML diagrams more efficiently, accurately, and collaboratively, revolutionizing the way software systems are designed and communicated.

Benefits of a Distributed LLM for UML Diagramming

The integration of a distributed LLM with a UML diagramming tool offers a number of benefits, including:

Reduced manual effort: The tool can generate UML diagrams automatically based on textual descriptions or natural language queries, which can significantly reduce the manual effort required to create diagrams.

Improved accuracy and correctness: The tool can provide accurate and relevant suggestions and recommendations based on its vast knowledge base, which can help to improve the quality and correctness of the diagrams produced.

Accelerated software development processes: The automation of the diagramming process can free up developers to focus on other tasks, such as system design and analysis. This can help to accelerate software development processes.

Enhanced collaboration: The tool can be used to collaborate with stakeholders and other developers, which can help to improve communication and understanding of the system.

Real-World Use Cases

The integration of a distributed LLM with a UML diagramming tool can be used in a variety of real-world use cases, including:

Software design: The tool can be used to design new software systems by creating UML diagrams that visualize the system architecture and components.

System analysis: The tool can be used to analyze existing software systems by creating UML diagrams that depict the system structure and behavior.

Documentation: The tool can be used to document software systems by creating UML diagrams that provide a visual representation of the system.

Training: The tool can be used to train developers on UML by providing them with interactive exercises that allow them to create and modify UML diagrams.

2 Literature Review & Background Work

Unified Modeling Language (UML) is a standardized modeling language used in software engineering to visually represent systems and processes. This literature review aims to explore the use of UML in the development and presentation of large language models.

Koc, Hatice & Erdoğan, Ali & Barjakly, Yousef & Peker, Serhat. UML Diagrams in Software Engineering Research: A Systematic Literature Review. Proceedings..

Overview of Large Language Models

Large language models are neural network models that have been pre-trained on a large corpus of text data. They have the ability to generate coherent and fluent text based on the input provided to them. LLMs have a wide range of applications, including language translation, chatbots, and text completion. The most widely known and used LLM is GPT-3.

A Bibliometric Review of Large Language Models Research from 2017 to 2023 Lizhou Fan, Lingyao Li, Zihui Ma, Sanggyu Lee, Huizi Yu, Libby Hemphill

Talking About Large Language Models Murray Shanahan
Ray is an open-source system for building and running distributed applications. One of the key components of Ray is the Ray server, which provides a central location for managing the distributed resources and executing tasks.

The Ray serv is designed to be scalable and efficient, allowing it to handle large-scale distributed applications with ease. It provides an easy-to-use API for developers to build distributed applications using Python, and also supports other programming languages such as Java and C++.

Ray: A Distributed Framework for Emerging AI Applications Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica, UC Berkeley

3 System Requirements & Analysis

3.1 AI-powerd UML diagramming tool

An AI-powered UML diagramming tool is a software application that uses artificial intelligence (AI) to automate the process of creating and editing UML diagrams. AI-powered UML diagramming tools can help developers to create UML diagrams more quickly and easily, and they can also help to improve the accuracy and consistency of the diagrams.

The AI capabilities enable the tool to learn from vast knowledge bases and stay up-to-date with the latest best practices and industry standards. It can analyze existing UML diagrams and derive patterns to provide better guidance to users. This helps in creating diagrams that adhere to established conventions and design principles, improving the overall quality and consistency of software systems.
Figure 3.1 illustrates the core use cases

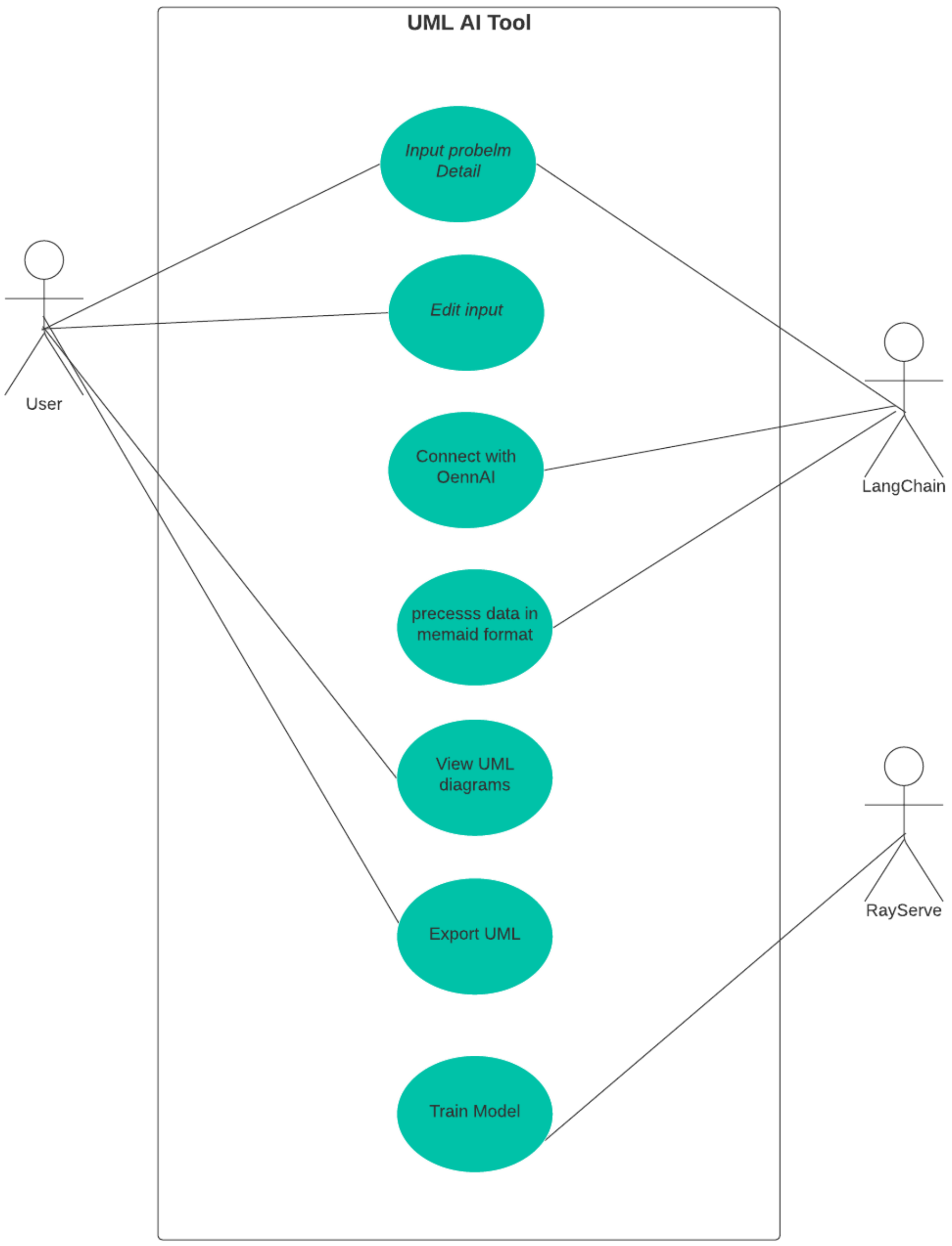


Figure 3.1 : UML AI tool - User Case Diaram

3.2 System Properties

3.2.1 Functionalities

Below are the functionality of the AI based UML diagram tool:

1. Generate UML sequence diagrams from user input:
   - The tool takes user input, which can be in the form of textual descriptions or natural language queries.
   - The distributed large language model analyzes the input and generates UML sequence diagrams automatically.
   - The generated sequence diagrams depict the interactions and message flows between various system components, aiding in understanding system behavior and communication patterns.
2. Generate UML class diagrams from user input:
   - Users can provide textual descriptions or specifications of classes, attributes, and relationships.
   - The tool utilizes the distributed large language model to interpret the input and generate UML class diagrams.
   - The generated class diagrams illustrate the structure of the software system, including classes, their attributes, methods, and relationships.
3. Generate UML ER diagrams from user input:
   - Users can input textual descriptions of entities, attributes, and relationships in a system.
   - The UML diagramming tool, powered by the distributed large language model, transforms the input into UML entity-relationship (ER) diagrams.
   - The generated ER diagrams visually represent the data structure and relationships in the system, aiding in database design and modeling.
4. Generate UML state diagrams from user input:
   - Users can describe the different states and state transitions of a system.
   - The tool employs the distributed large language model to interpret the user input and generate UML state diagrams.
   - The generated state diagrams illustrate the behavior and state transitions of the system components, facilitating understanding of system dynamics.

5. Generate Gantt diagrams from user input:

- Users can provide project schedules, tasks, and dependencies in textual format.
- The UML diagramming tool, utilizing the capabilities of the distributed large language model, converts the input into Gantt diagrams.
- The generated Gantt diagrams visualize project timelines, task dependencies, and milestones, aiding in project planning, scheduling, and tracking.

These functionalities of the UML diagramming tool, empowered by the distributed large language model, enable users to efficiently generate a variety of UML diagrams, such as sequence diagrams, class diagrams, ER diagrams, state diagrams, and even Gantt diagrams. This integration combines the expressive power of UML with the intelligence of AI, simplifying and accelerating the diagram creation process for software developers and stakeholders.

3.2.2 Scalability

Deploying an AI-based UML diagram generation tool as a web application on AWS, along with Qdrant Vector DB, provides a scalable infrastructure to handle growing user demands and ensure optimal performance. The combination of AWS services and Qdrant Vector DB offers various scalability options for different aspects of the application. The following highlights the scalability aspects of this setup:

1. Load Balancing and Auto Scaling:

To handle increased user traffic and distribute it across multiple instances of the web application, AWS Elastic Load Balancing (ELB) can be utilized. ELB automatically balances the incoming traffic, ensuring optimal performance and reliability. Combined with Auto Scaling, ELB adjusts the number of instances based on the demand, allowing the web application to handle varying levels of user activity while maintaining responsiveness and scalability.

2. Qdrant Vector DB for Scalable Storage and Querying:

Qdrant Vector DB is a high-performance vector search engine that can be used for storing and querying UML diagrams or related data. It provides efficient storage and retrieval of high-dimensional vector data, which is essential for storing feature embeddings of UML diagrams generated by the AI model. As the number of UML diagrams and vector data increases, Qdrant Vector DB scales horizontally to handle the growing dataset and query loads effectively.

By deploying the AI-based UML diagram generation tool as a web application on AWS and utilizing Qdrant Vector DB for scalable storage, the application can handle increased user demands effectively. The combination of AWS services, including Elastic Beanstalk, RDS or DynamoDB, S3, Lambda, and load balancing, along with Qdrant Vector DB, enables seamless scalability, efficient data management, and optimal performance for the web application as the user base and dataset grow.

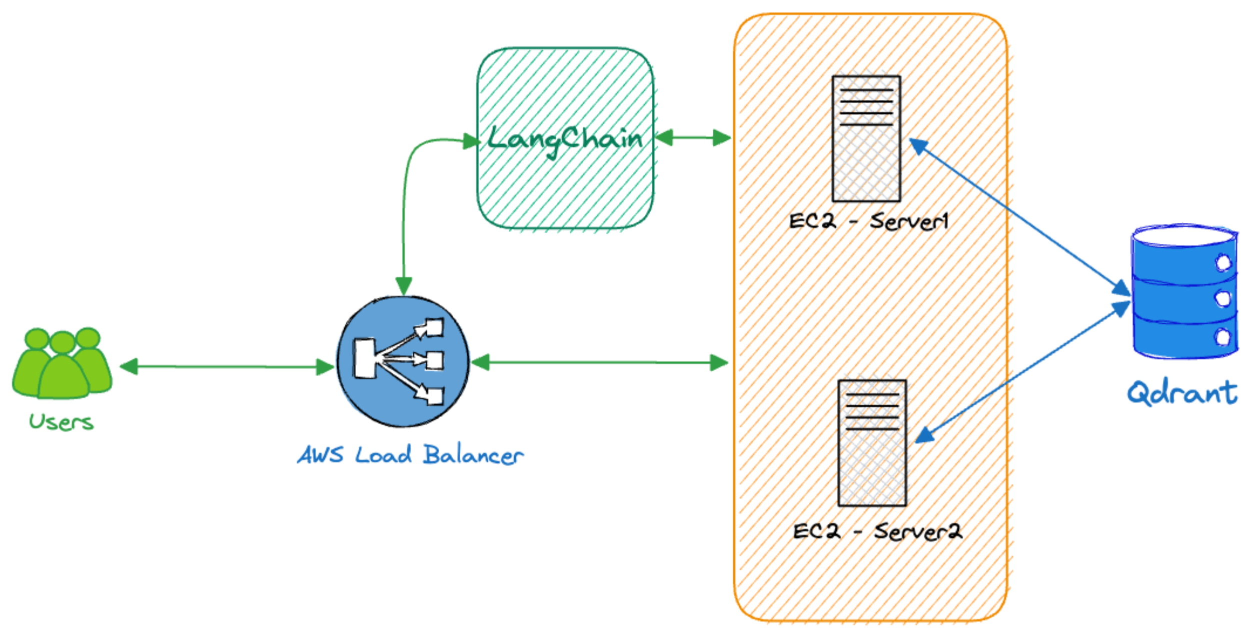


Figure 3.2.2 : Scalability and Load Balancing

4 System Infrastructure and Architecture

The UML diagramming tool with a distributed large language model is built upon a sophisticated architecture that combines the functionalities of the tool with the intelligence and capabilities of the large language model. The architecture consists of several key components working together to provide a seamless and powerful diagramming experience. The following are the main components of the architecture:

User Interface (UI):

The User Interface component provides the interface through which users interact with the UML diagramming tool. It offers a user-friendly environment for creating, editing, and viewing UML diagrams. The UI allows users to input textual descriptions or queries and receive the generated diagrams, real-time suggestions, and collaboration features. It provides an intuitive and visually appealing platform for diagram creation and manipulation.

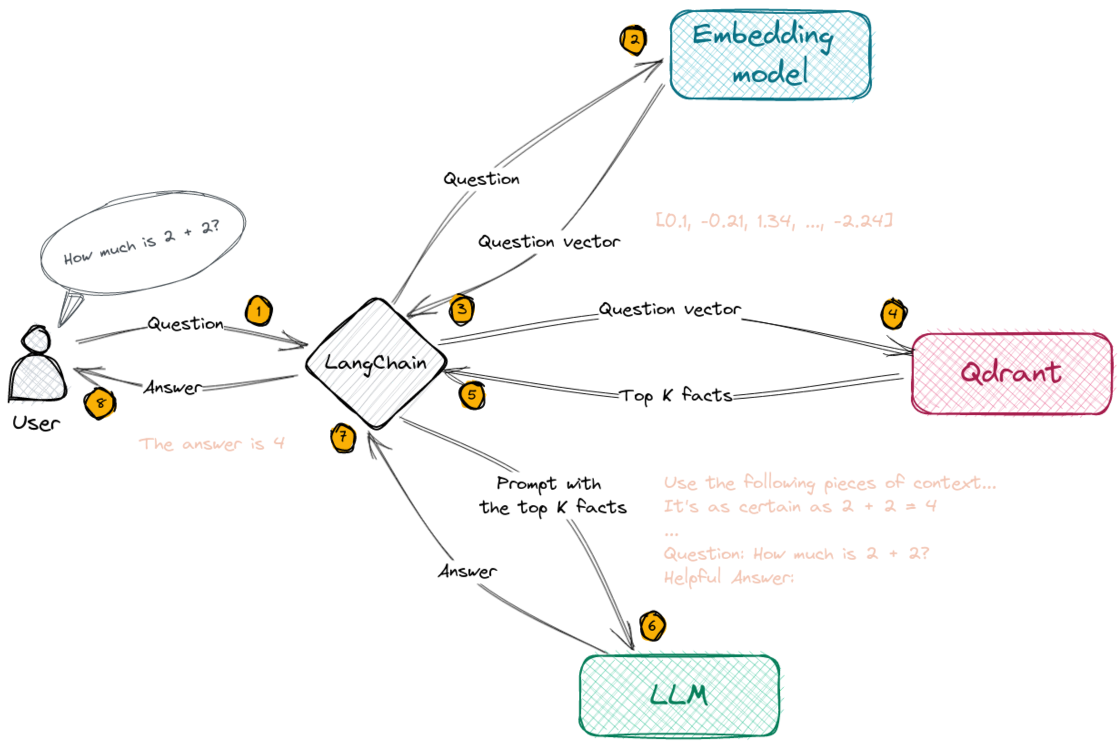
UML Diagramming Engine:

The UML Diagramming Engine is responsible for the core functionalities of the tool. It processes user input, interprets the text, and generates the corresponding UML diagrams. The engine incorporates the rules and conventions of UML diagramming, ensuring the accuracy and correctness of the generated diagrams. It interfaces with the distributed large language model to analyze the input and make intelligent decisions during the diagram generation process.

Distributed Large Language Model:

At the heart of the architecture is the distributed large language model, such as the GPT-3.5 architecture developed by OpenAI. This model possesses an extensive knowledge base and a deep understanding of human language. It is trained on a wide range of data sources, including software engineering and UML-related materials. The distributed nature of the model allows it to continuously learn and stay updated with the latest practices and trends in UML diagramming. The architecture of the UML diagramming tool with a distributed large language model combines the intelligence of the large language model, the capabilities of natural language processing, and the

UML diagramming engine to deliver an advanced and efficient diagramming experience. It empowers users to create accurate UML diagrams, receive real-time suggestions, access a comprehensive knowledge base, and collaborate seamlessly, ultimately improving productivity and the quality of software system design.



4.2 Use Flow diagram

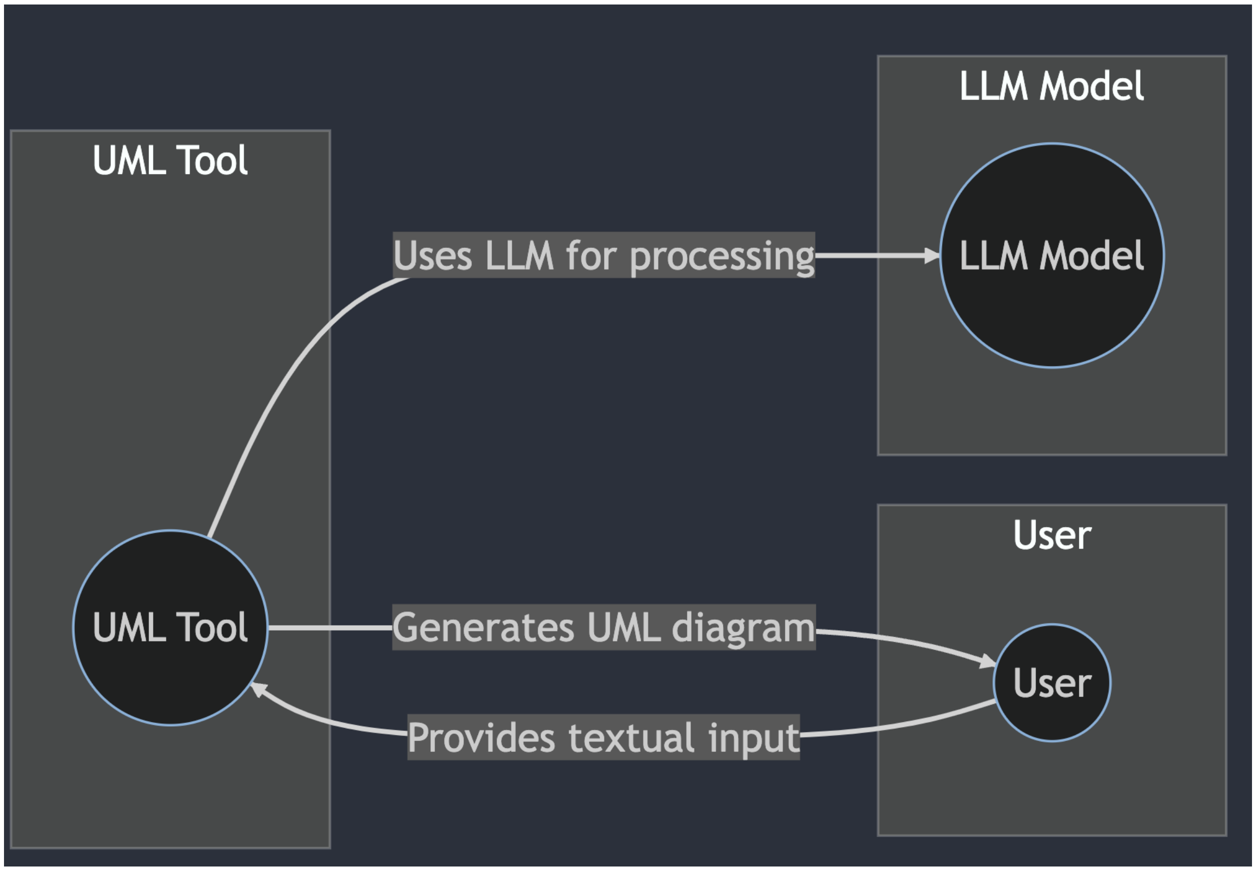


Figure 4.2 : Use Flow diagram

5 Cloud-based System Design

5.1 Data Requirements

We have utilized OpenAI's LLM and Mermaid for our data requirements of our application.

Training a large language model like GPT-3 or GPT-3.5, which serves as the foundation for the ChatGPT model, involves specific data requirements. The following are the key data requirements for training a large language model:

- **Text Corpus**: A diverse and extensive text corpus is crucial for training a large language model. This corpus can include a wide range of sources, such as books, articles, websites, scientific papers, forum discussions, social media posts, and more. The larger and more varied the corpus, the better the model's ability to generate high-quality responses across different topics and contexts.

- **Data Quality**: Ensuring high-quality data is important to maintain the model's performance. It involves carefully curating and preprocessing the text corpus, which may include removing noise, duplicates, or irrelevant information, correcting errors, standardizing formatting, and ensuring data consistency.

- **Data Representation**: Language models typically use text as their input and output format. Therefore, the data for training should be represented as text, preferably in a format that is easy to process, such as plain text or tokenized text.

- **Supervised Data (Optional)**: For certain tasks, supervised learning can be employed by providing labeled examples. This involves pairing input text with corresponding desired output or target text. However, it's important to note that models like GPT-3.5 are typically trained using unsupervised learning and don't explicitly require labeled data.

- **Preprocessing and Tokenization**: Preprocessing the data involves tasks like tokenization, sentence splitting, and possibly other language-specific preprocessing steps. Tokenization refers to splitting the text into individual tokens, such as words or subword units, which are the input units processed by the language model.

- **Hardware Resources**: Training a large language model like GPT-3.5 requires significant computational resources, including powerful processors (GPUs or TPUs) and large amounts of memory. Specialized infrastructure and distributed training techniques are often employed to handle the computational requirements efficiently.

5.2 Entity-Relationship

Mermaid is a JavaScript library that allows you to generate diagrams using a simple textual syntax. When it comes to UML diagrams, you will typically need the following information:

Class Diagram:
Class names: Identify the classes in your system or application.
Attributes: Define the attributes (data members) of each class.
Methods: Specify the methods (functions) associated with each class.
Relationships: Identify the associations, dependencies, inheritances (generalization), and other relationships between classes.

Use Case Diagram:
Actors: Identify the actors interacting with the system.
Use cases: Define the different use cases or functionalities of the system.
Relationships: Illustrate how actors and use cases are related, such as associations or generalizations.

Sequence Diagram:
Objects: Identify the objects or participants involved in the sequence of interactions.
Lifelines: Define the lifelines of objects to represent their existence during the sequence.

Messages: Specify the messages exchanged between objects to depict the interaction flow.
Activation: Represent the activation and duration of method calls on objects.

Activity Diagram:
Actions: Define the actions or steps in the process or workflow.
Decision points: Identify the decision points or branches in the flow.
Control flow: Specify the transitions and conditions between actions and decision points.
information. For example, in a class diagram, you would define classes, attributes, methods, and relationships using Mermaid's textual syntax. Similarly, for use case diagrams, sequence diagrams, and activity diagrams, you would utilize Mermaid's syntax to represent the relevant elements and their relationships.

Mermaid's syntax is designed to be intuitive and easy to read, allowing you to create UML diagrams without relying on specific data formats or requirements. You can customize the appearance of the diagrams, such as colors, styles, and layout options, to suit your preferences.

5.3 Database Design

Qdrant is an open-source vector similarity search engine designed for fast and efficient similarity searches on high-dimensional data. It is primarily used for applications that involve searching and retrieving similar vectors or embeddings based on their similarities.

Here are some key aspects and data requirements related to Qdrant:

Vector Data: Qdrant is specifically designed for working with vector data. It supports searching and indexing high-dimensional vectors, such as embeddings generated by machine learning models. Examples of vector data that can be used with Qdrant include image embeddings, text embeddings, audio embeddings, and more.

Similarity Search: Qdrant enables efficient similarity searches on the vector data. It utilizes algorithms like Approximate Nearest Neighbor (ANN) search to find vectors that are close in proximity to a given query vector. This is useful in various applications such as recommendation systems, content-based retrieval, and clustering.

Indexing: Qdrant uses advanced indexing structures and techniques to optimize search performance on large-scale vector data. It supports different indexing methods like Hierarchical Navigable Small World (HNSW), Annoy, and IVFADC (Inverted File with Approximate Distance Calculation). These indexing methods help speed up the search process by organizing and partitioning the vector space.

Scale and Performance: Qdrant is designed to handle large-scale vector datasets efficiently. It can index millions or even billions of vectors and provide fast search results. The performance and scalability of Qdrant depend on factors like hardware resources, the dimensionality of vectors, and the chosen indexing method.

Integration and APIs: Qdrant provides APIs and SDKs (Software Development Kits) for easy integration with different programming languages and frameworks. It allows you to index and search vectors programmatically and provides flexibility in building applications that require vector similarity search capabilities.

Open-Source Nature: Qdrant is an open-source project, which means the source code is freely available for inspection, modification, and contribution. This allows users and developers to customize and extend the functionality of Qdrant to suit their specific needs.

This metadata can include additional attributes, identifiers, or any other relevant information related to the vectors.

Determine Metadata Requirements: Start by identifying the metadata that needs to be associated with the vectors in Qdrant. This may include attributes such as names, descriptions, timestamps, categories, labels, or any other relevant information that describes the vectors. Consider the specific needs of your application and the type of data you're working with.

Define Tables and Fields: Based on the metadata requirements, design your database schema with tables and fields. Create a table to store the vectors themselves in Qdrant, typically as a separate entity from your metadata. Then, create another table(s) to store the metadata, with fields representing the different attributes you identified in the previous step.

Establish Relationships: Determine if there are any relationships between the vector table and the metadata table(s). For example, if you have a unique identifier for each vector, you can establish a foreign key relationship between the vector table and the metadata table(s) to associate the metadata with the corresponding vectors.

Indexing and Performance: Depending on the size and complexity of your dataset, consider incorporating indexing techniques to improve the performance of your database queries. Appropriate indexing can help speed up searches and retrievals based on different attributes, such as names, categories, or labels.

Consider Data Access Patterns: Understand the typical data access patterns of your application. This includes identifying the most common types of queries or operations that will be performed on the metadata and designing the database schema and indexes accordingly to optimize performance for those patterns.

Integration with Qdrant: Determine how your database and Qdrant will interact. Qdrant focuses on vector similarity search, so you'll need to establish a way to link the vector identifiers in your database with the corresponding vectors stored in Qdrant. This can involve using unique identifiers or shared keys between the two systems.

It's important to note that Qdrant is a separate component from your database, and the specific design and implementation details of your database will depend on the DBMS you choose to use.

5.4 Cloud-Based System Communication Design

When designing the communication architecture for a cloud-based system that incorporates a large language model-based application, there are some specific consideration we kept in mind:

Client-Server Communication: The application's client-side interacts with the server-side components responsible for handling the language model. Ensure that the client-server communication follows established protocols and APIs to facilitate smooth interaction. RESTful APIs or GraphQL can be suitable options, allowing clients to send requests and receive responses from the server.

Language Model Interaction: Determine how the server-side components interact with the large language model. The communication can be either synchronous or asynchronous, depending on the application's requirements. For synchronous communication, the server can directly interact with the language model in real-time, waiting for responses before sending them back to the client. For asynchronous communication, the server can offload tasks to a separate processing queue or utilize message brokers for handling requests and responses.

Load Balancing and Scalability: As the usage of the large language model application grows, ensure that the communication architecture is designed for scalability and load balancing. Distribute the workload across multiple servers or instances to handle increased traffic and user requests efficiently. Load balancers and auto-scaling mechanisms can help achieve scalability and optimize resource utilization. We have used load Balancing and scalability capabilities offered by AWS.

Caching and Optimization: Implement caching mechanisms to improve the performance of the language model application. Frequently accessed or computationally expensive responses can be cached at various levels, such as the server, CDN (Content Delivery Network), or client-side, to reduce response times and alleviate the load on the language model.

Security and Authentication: Secure the communication channels to protect user data and application integrity. Implement encryption protocols (e.g., TLS/SSL) to secure data in transit. Consider implementing authentication and authorization mechanisms to ensure that only authorized users or applications can access the language model or sensitive functionalities.

Monitoring and Logging: Incorporate monitoring and logging mechanisms to track system performance, communication metrics, and potential issues. Monitoring tools can help identify bottlenecks, analyze response times, and optimize the application's performance. Additionally, logging can assist in debugging, troubleshooting, and analyzing usage patterns.

Error Handling and Resilience: Design the communication architecture with robust error handling and resilience mechanisms. Implement proper error responses, retries, and fallback mechanisms to handle potential failures or interruptions in the communication flow. Consider implementing circuit breakers, retry policies, and fault-tolerant architectures to ensure system availability and recoverability.

Integration with Cloud Services: Leverage cloud services to enhance communication capabilities and simplify infrastructure management. Cloud provider offerings such as API gateways, message queues, serverless functions, and event-driven architectures can help streamline the communication between components and enhance scalability, security, and reliability.

5.4.1 Business User Workflows

User Interaction:
The business user accesses the application through a web-based interface.
The user is presented with options to input the required information for creating UML diagrams.
The user selects the desired UML diagram type, such as class diagram, sequence diagram, or activity diagram.
Input Gathering:

The application prompts the user to provide the necessary input for the selected UML diagram type.

For a class diagram, the user may input the names of classes, their attributes, and relationships.
For a sequence diagram, the user may input the sequence of events, lifelines, and messages exchanged between objects.
For an activity diagram, the user may input the steps, decisions, and transitions in the workflow.

User Input Processing:
The large language model receives the user input and processes it to understand the intent and context.
The language model extracts relevant information, such as class names, attributes, relationships, or event sequences.
It uses its natural language understanding capabilities to interpret and analyze the user's input accurately.

UML Diagram Generation:
Based on the processed user input, the large language model generates the corresponding UML diagram.
The model applies predefined templates and rules to convert the user's input into a visual UML diagram representation.
It leverages its knowledge of UML diagram syntax, semantics, and best practices to create accurate and meaningful diagrams.

Output Presentation:
The application presents the generated UML diagram to the user.
The user can view the diagram on the user interface or download it as an image or file.
The application may offer options for customizing the diagram's appearance, such as layout, color, or styling.

5.4.2 Algorithms & Pseudo-code

Creating UML diagrams based on minimal user input, processing it through a large language model, and generating Mermaid code for the diagram can be achieved through the following algorithms and pseudo-code:

Algorithm: Generate UML Diagram from Minimal User Input
Input: Minimal user input
Output: UML diagram

1. **Read minimal user input**
    - Read the minimal user input required for the UML diagram generation.

2. **Process user input using a large language model**
    - Pass the user input to the large language model for processing.
    - Retrieve the generated output from the language model.

3. **Extract Mermaid code from language model output**
    - Extract the relevant information from the language model output to generate Mermaid code.
    - Identify class names, relationships, attributes, or any other UML diagram elements from the generated output.
4. **Generate UML diagram using Mermaid**
    - Use a Mermaid library or tool to convert the extracted Mermaid code into a visual UML diagram.
    - Follow the syntax and structure specified by Mermaid to define the classes, relationships, and other elements.
5. **Display or output the UML diagram**
    - Display the generated UML diagram on the user interface or output it as an image or file.
    - Provide options for the user to customize the diagram's appearance, such as color schemes or styling.

Pseudo-code (raw)

```
function generateUMLDiagram(userInput):
  // Process user input using a large language model
  languageModelOutput = processUserInput(userInput)


  // Extract Mermaid code from language model output
  mermaidCode = extractMermaidCode(languageModelOutput)

  // Generate UML diagram using Mermaid
  umlDiagram = generateDiagramFromMermaid(mermaidCode)

  return umlDiagram
```

5.4.3 API Design

OpenAI provides an API that allows developers to integrate the capabilities of GPT-3 and other models into their applications. The OpenAI API allows you to make requests to the models hosted on the OpenAI infrastructure, enabling you to leverage natural language processing, text generation, and other AI-powered functionalities.
To use the OpenAI API, we followed these steps:

- **We signed up and obtained our API key**: We visited the OpenAI website and signed up for an account. Once our account was approved, we received an API key that authorizes our application to access the API endpoints.

- **We chose the model**: We selected the model that best suited our task. For example, we decided to use GPT-3 for natural language understanding and generation tasks, such as text completion, translation, or conversation simulation. We used GPT-3 as we had free credits for this, GPT-4 was expensive.

- **We set up the API integration**: We configured our application to make HTTP requests to the OpenAI API. We used libraries or frameworks in our preferred programming language to handle the API communication.

- **We made API requests**: We structured our requests to the OpenAI API by specifying the model, the prompt or input text, and any additional parameters relevant to our use case. We passed the necessary information as data in the API request payload.

- **We received API responses**: We sent the API request and handled the response from the OpenAI API. The response contained the generated output, which we processed and used in our application.

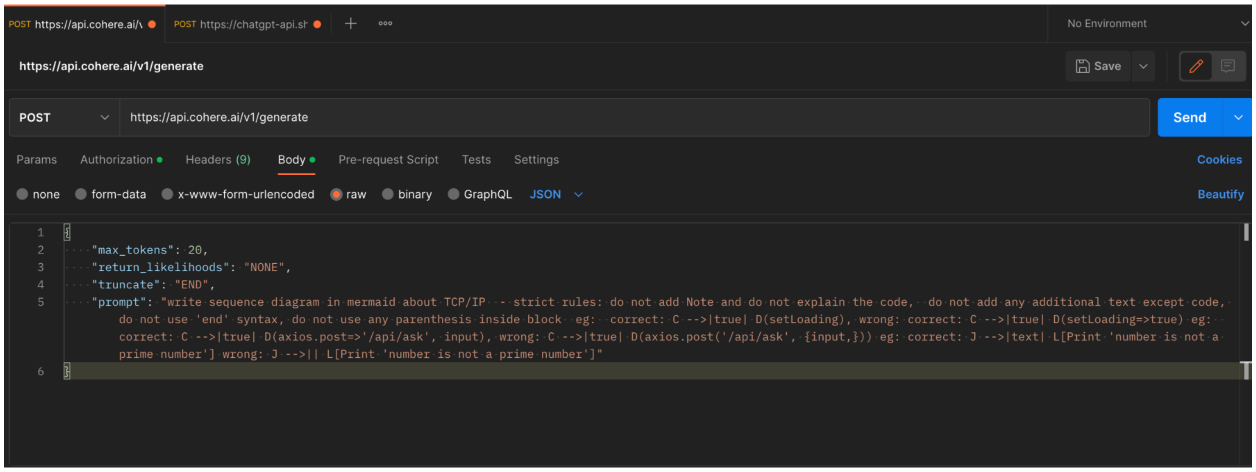
- **We iterated and optimized**: We iterated on our application's integration with the OpenAI API, fine-tuning our requests and processing the responses to achieve the desired functionality and results.

Making requests

You can paste the command below into your terminal to run your first API request. Make sure to replace `$OPENAI_API_KEY` with your secret API key.

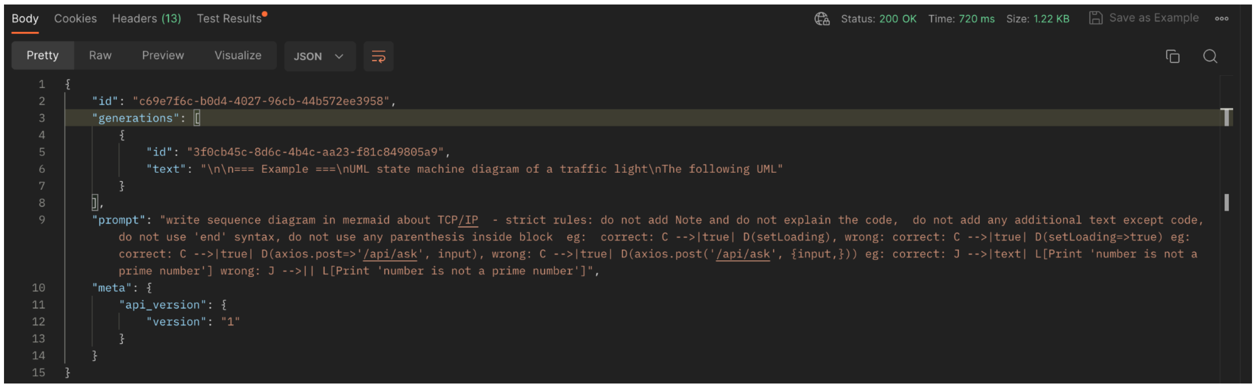
```
curl https://api.openai.com/v1/chat/completions \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $OPENAI_API_KEY" \
  -d '{
     "model": "gpt-3.5-turbo",
     "messages": [{"role": "user", "content": "Say this is a test!"}],
     "temperature": 0.7
   }'
```

API request/response



POST https://api.cohere.ai/v1/generate

```
{
    "max_tokens": 20,
    "return_likelihoods": "NONE",
    "truncate": "END",
    "prompt": "write sequence diagram in mermaid about TCP/IP  - strict rules: do not add Note and do not explain the code,  do not add any additional text except code, do not use 'end' syntax, do not use any parenthesis inside block  eg:  correct: C -->|true| D(setLoading), wrong: correct: C -->|true| D(setLoading=>true) eg:  correct: C -->|true| D(axios.post=>'/api/ask', input), wrong: C -->|true| D(axios.post('/api/ask', {input,})) eg: correct: J -->|text| L[Print 'number is not a prime number'] wrong: J -->|| L[Print 'number is not a prime number']"
}
```



Status: 200 OK Time: 720 ms Size: 1.22 KB

```
{
    "id": "c69e7f6c-b0d4-4027-96cb-44b572ee3958",
    "generations": [
        {
            "id": "3f0cb45c-8d6c-4b4c-aa23-f81c849805a9",
            "text": "\n\n=== Example ===\nUML state machine diagram of a traffic light\nThe following UML"
        }
    ],
    "prompt": "write sequence diagram in mermaid about TCP/IP  - strict rules: do not add Note and do not explain the code,  do not add any additional text except code, do not use 'end' syntax, do not use any parenthesis inside block  eg:  correct: C -->|true| D(setLoading), wrong: correct: C -->|true| D(setLoading=>true) eg: correct: C -->|true| D(axios.post=>'/api/ask', input), wrong: C -->|true| D(axios.post('/api/ask', {input,})) eg: correct: J -->|text| L[Print 'number is not a prime number'] wrong: J -->|| L[Print 'number is not a prime number']",
    "meta": {
        "api_version": {
            "version": "1"
        }
    }
}
```

6 Key Performance Metrics

6.1 Localhost Results

For development, we developed and tested our web application on local host. We tested API in postman to request and get response from openAI. We tried cohere and open AI LLM's but got better results from open AI. We did not face any bottlenecks on localhost but to make it distributed and apply it's features we developed our application on AWS for scalability, fault tolerance, load-balancing and consistency.

6.2 AWS EC2 Results

Auto Scaling is a feature provided by Amazon Web Services (AWS) that allows you to automatically adjust the capacity of your applications based on predefined conditions. It helps you maintain the desired performance, cost efficiency, and availability by dynamically scaling resources up or down as needed.
Here's an overview of how Auto Scaling works on AWS:

1. **We defined an Auto Scaling group**: We started by creating an Auto Scaling group on AWS. We specified the minimum and maximum number of instances we wanted to maintain within the group to handle our application's workload.
2. **We set up scaling policies**: We defined scaling policies based on various metrics to determine when Auto Scaling should add or remove instances. These policies were based on factors like CPU utilization, network traffic, or application-specific metrics.
3. **We monitored and collected metrics**: We utilized Amazon CloudWatch to monitor our applications and resources, collecting performance metrics such as CPU usage or network traffic. This allowed us to gather the data needed for scaling decisions.
4. **We created scaling triggers**: Using the metrics we collected, we created scaling triggers in CloudWatch. These triggers established the conditions under which Auto Scaling should add or remove instances. For example, we set triggers to add more instances if the average CPU utilization exceeded a specific threshold for a certain duration.
5. **We experienced scale-out and scale-in**: When a scaling trigger condition was met, Auto Scaling automatically launched additional instances (scale-out) or

terminated existing instances (scale-in) based on our predefined rules. This helped us dynamically adjust capacity to meet changing demand.

6. **We integrated load balancing**: We utilized Elastic Load Balancing to evenly distribute incoming traffic across the instances within our Auto Scaling group. This ensured optimal performance and high availability for our application.
7. **We performed health checks and maintained availability**: Auto Scaling regularly performed health checks on instances to ensure they were functioning properly. If an instance failed a health check, Auto Scaling replaced it with a new one, ensuring the availability and reliability of our application.
8. **We monitored and optimized performance**: We closely monitored the performance of our Auto Scaling group and adjusted scaling policies as needed. This allowed us to optimize resource allocation and cost-effectiveness, ensuring our application scaled efficiently based on demand.

Implementing **Auto Scaling** on AWS allowed us to automatically adjust our application's capacity, ensuring optimal performance, cost efficiency, and availability. By leveraging the scalability and elasticity of the cloud, we were able to handle fluctuations in demand seamlessly while minimizing costs.

We also implemented **Auto loading** on AWS to distribute incoming traffic across multiple instances running your large language model-based application. This helps ensure scalability, availability, and efficient resource utilization, enabling your application to handle user requests and generate UML diagrams effectively.

1. **We launched EC2 Instances**: We set up the EC2 instances that would run our application. These instances would handle user requests and generate UML diagrams based on the large language model.

2. **We configured Security Groups**: We created security groups for our EC2 instances to control inbound and outbound traffic. We made sure to open the necessary ports, such as HTTP/HTTPS, to allow user access to our application.

3. **We installed and configured the Application**: We installed and configured our large language model-based application on each EC2 instance. We ensured that the application could receive user input, process it using the language model, fetch Mermaid code, and generate UML diagrams.

4. **We set up an Application Load Balancer (ALB)**: We created an Application Load Balancer in AWS. We configured the ALB to listen for incoming traffic on the desired ports, such as HTTP/HTTPS. We specified the target group as the EC2 instances running our application.

5. **We created Target Groups:** We defined a target group for our load balancer, specifying the EC2 instances as targets. This allowed the load balancer to distribute traffic evenly across these instances.

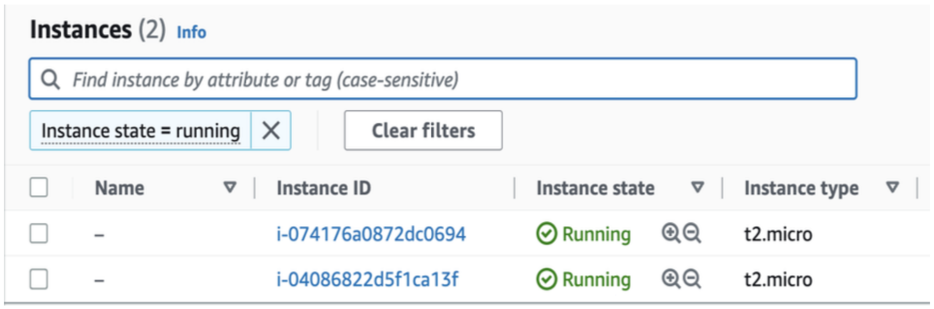
6. **We configured Health Checks**: We set up health checks on the target group to ensure that only healthy instances received traffic. We configured the health checks to monitor the availability and responsiveness of our application.

7. **We set Load Balancer Rules**: We configured load balancer rules to define how traffic should be routed to our application instances. For example, we set up path-based routing to direct requests for different endpoints to specific instances.

8. **We integrated Auto Scaling**: We integrated the load balancer with Auto Scaling to automatically adjust the number of instances based on traffic demands. This ensured that our application could handle increased load and maintain optimal performance.

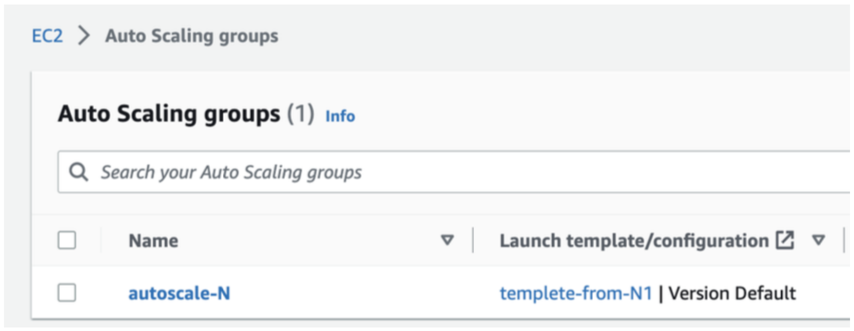
9. **We tested and monitored**: We tested the load balancing setup by accessing our application through the load balancer's DNS or IP address. We monitored the performance of our application and the load balancer using Amazon CloudWatch metrics and logs.

AWS deployment


Instances (2) Info

Name	Instance ID	Instance state	Instance type
–	i-074176a0872dc0694	Running	t2.micro
–	i-04086822d5f1ca13f	Running	t2.micro

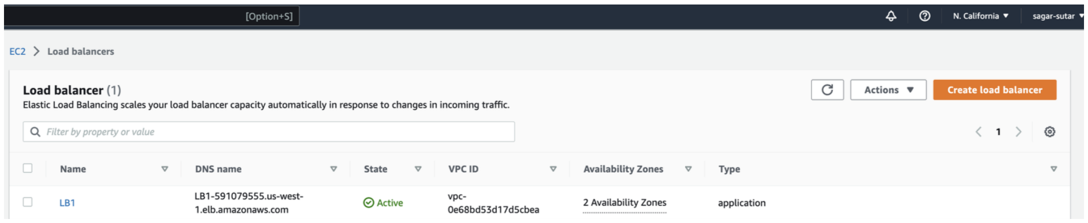
Figure: Autoscaling


EC2 > Auto Scaling groups

Auto Scaling groups (1) Info

Name	Launch template/configuration
autoscale-N	templete-from-N1 | Version Default

Figure: Auto scaling group



EC2 > Load balancers

Load balancer (1)
Elastic Load Balancing scales your load balancer capacity automatically in response to changes in incoming traffic.

Name	DNS name	State	VPC ID	Availability Zones	Type
LB1	LB1-591079555.us-west-1.elb.amazonaws.com	Active	vpc-0e68bd53d17d5cbea	2 Availability Zones	application

Figure: Load balancer

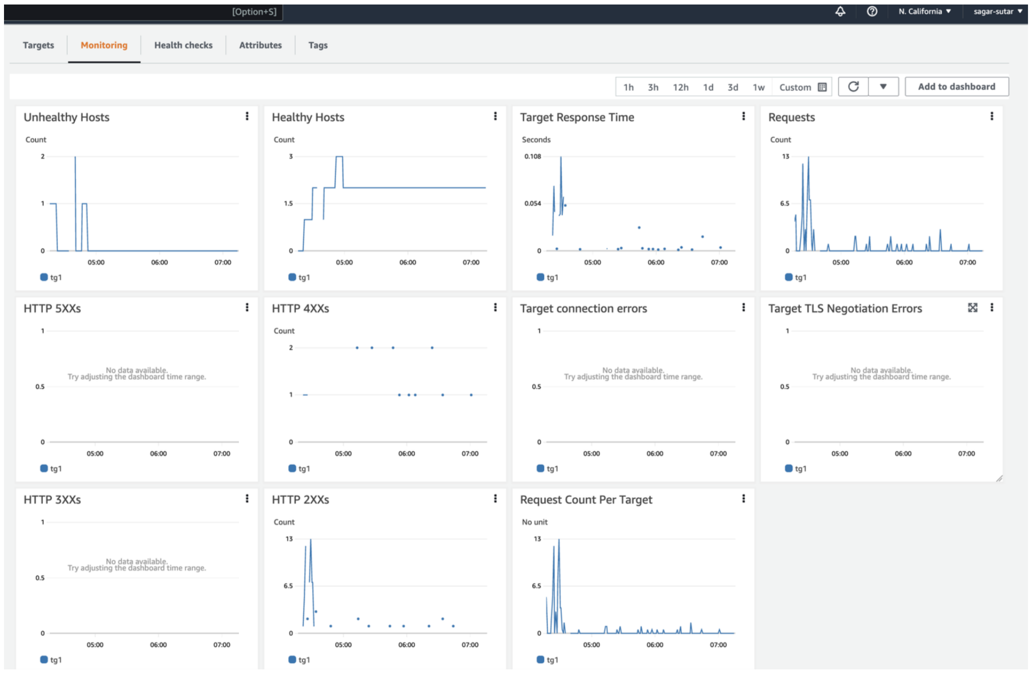


Figure: Monitoring metrics

Appendix A: Glossary

LLM: LLMs are a type of AI model designed to generate natural language text. They use deep learning algorithms to analyze vast amounts of text data to identify patterns and relationships between words and phrases. LLMs can generate coherent and relevant text in response to a given prompt or input.
Example - OpenAI's GPT (Generative Pre-trained Transformer)

Mermaid: Mermaid is a popular open-source JavaScript library and tool for generating diagrams and flowcharts using simple text-based syntax. It allows you to create diagrams easily without the need for complex graphical editors.

Unified Modeling Language (UML): A standardized modeling language used in software engineering to visually represent systems and processes. This literature review aims to explore the use of UML in the development and presentation of large language models.

Qdrant: An open-source vector similarity search engine that is designed to efficiently handle large-scale vector data and perform fast nearest neighbor searches. It is commonly used for tasks such as recommendation systems, image search, document search, and more.

Appendix B: References

Koc, Hatice & Erdoğan, Ali & Barjakly, Yousef & Peker, Serhat. UML Diagrams in Software Engineering Research: A Systematic Literature Review. Proceedings..

A Bibliometric Review of Large Language Models Research from 2017 to 2023 Lizhou Fan, Lingyao Li, Zihui Ma, Sanggyu Lee, Huizi Yu, Libby Hemphill

Talking About Large Language Models Murray Shanahan

Ray: A Distributed Framework for Emerging AI Applications Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica, UC Berkeley

https://docs.ray.io/en/latest/serve/index.html

https://platform.openai.com/docs/introduction

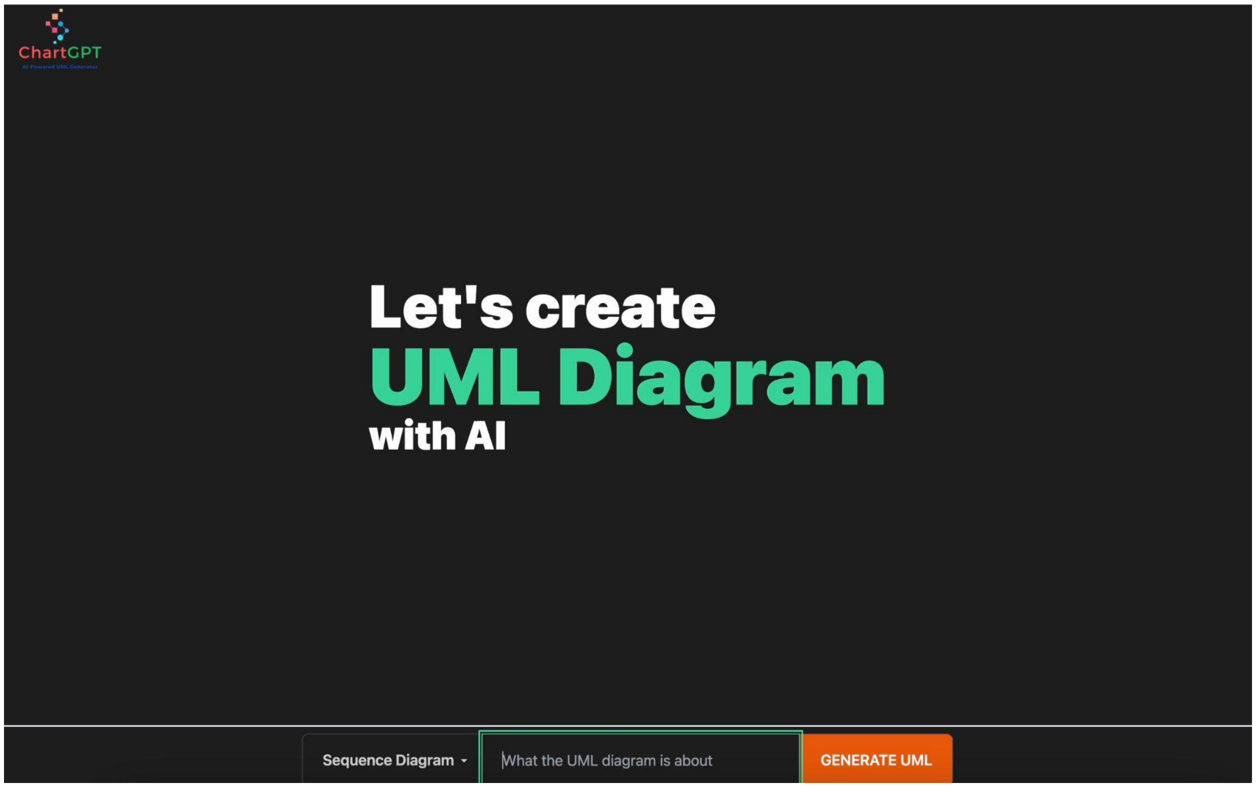
https://docs.google.com/document/d/1tBw9A4j62ruI5omlJbMxly-la5w4q_TjyJgJL_jN2fI/preview

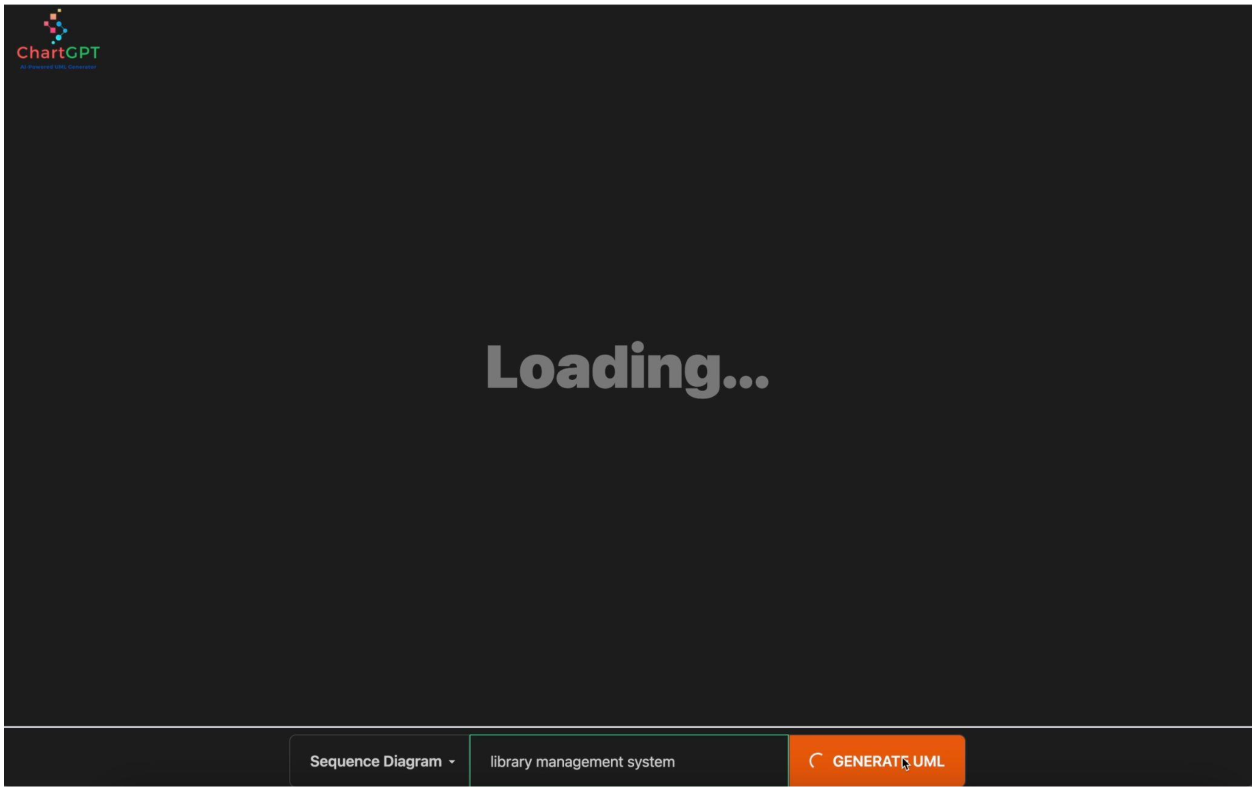
https://docs.google.com/document/d/1bYL-638GN6EeJ45dPuLiPlmA8msojEDDKiBx3YzB4_s/preview

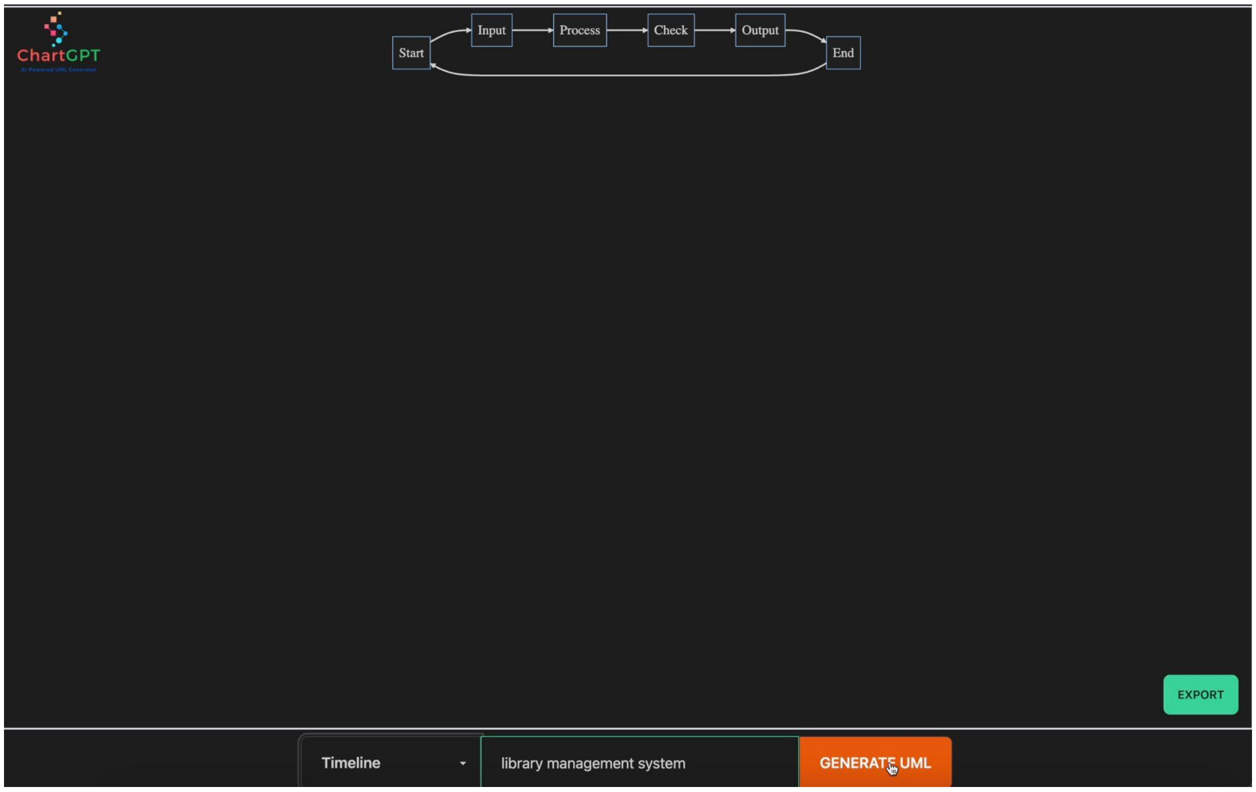
https://docs.ray.io/en/latest/rllib/rllib-algorithms.html

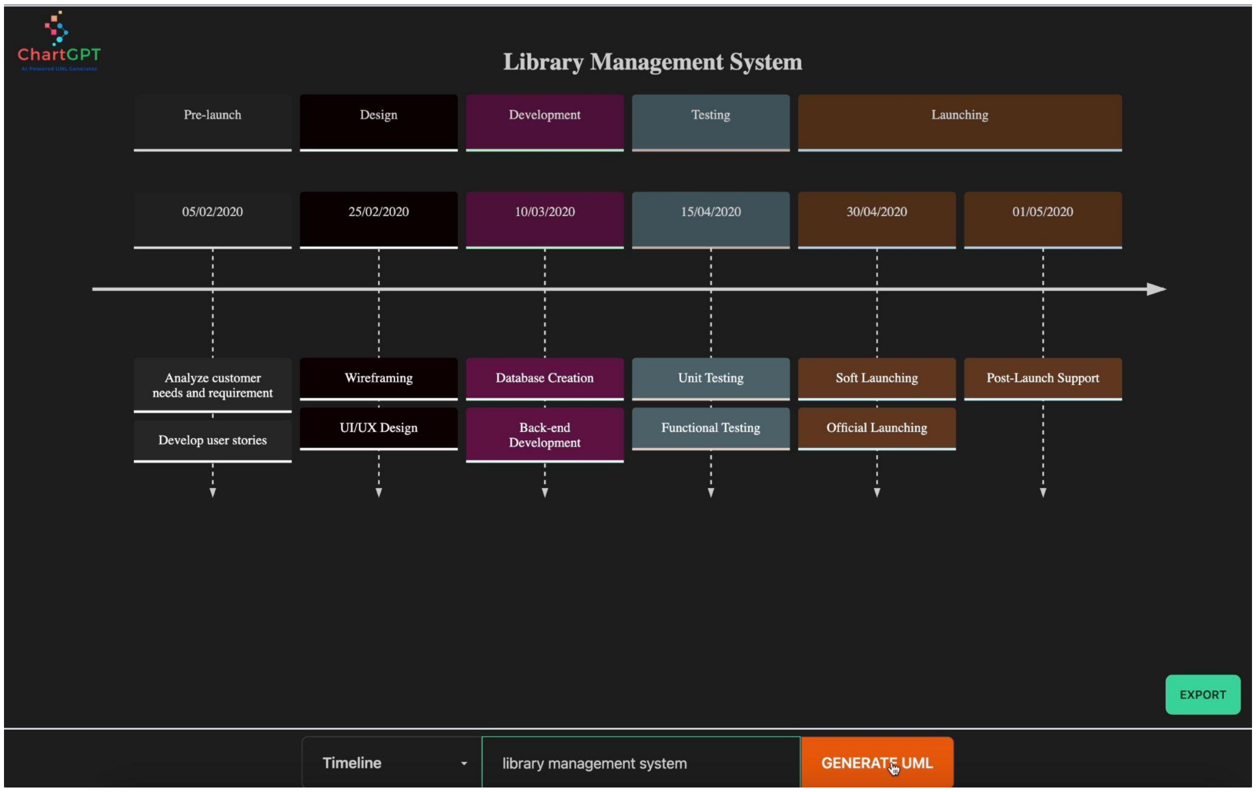
https://www.sciencedirect.com/topics/computer-science/distributed-machine-learning

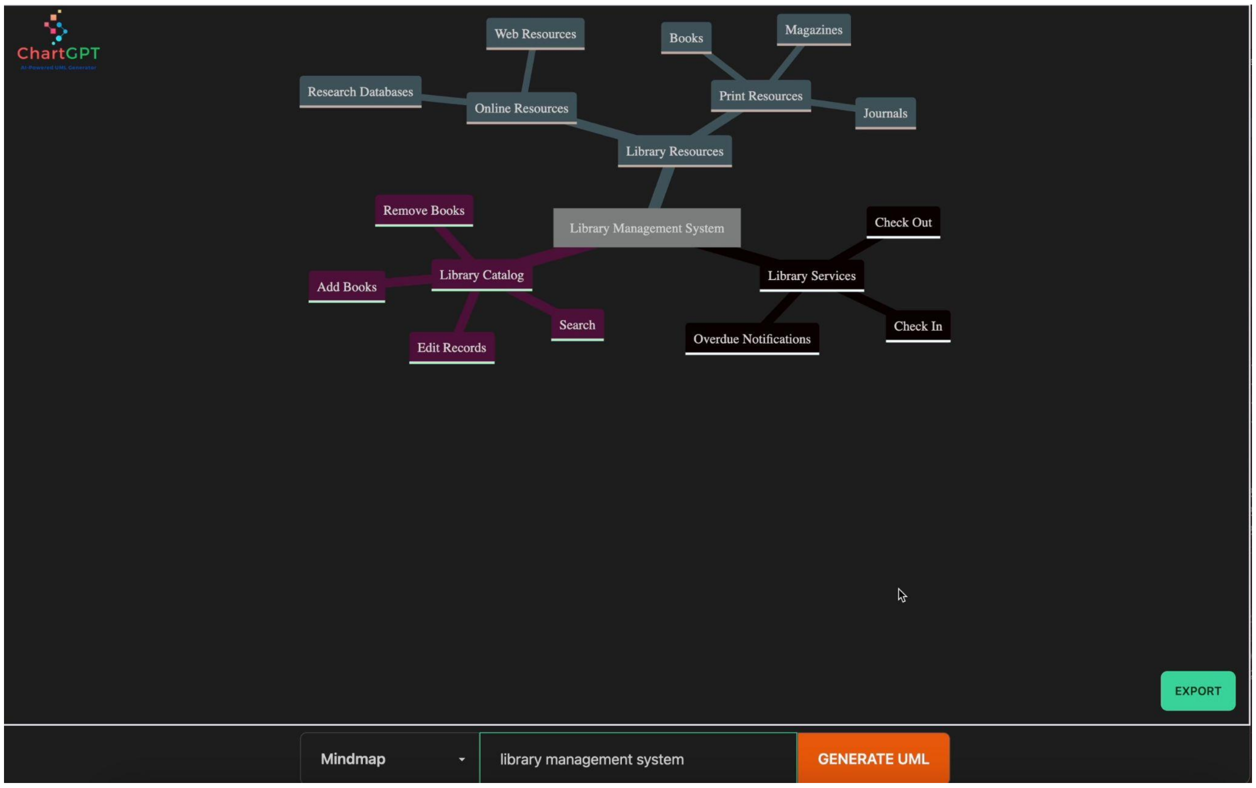
Appendix C: UI

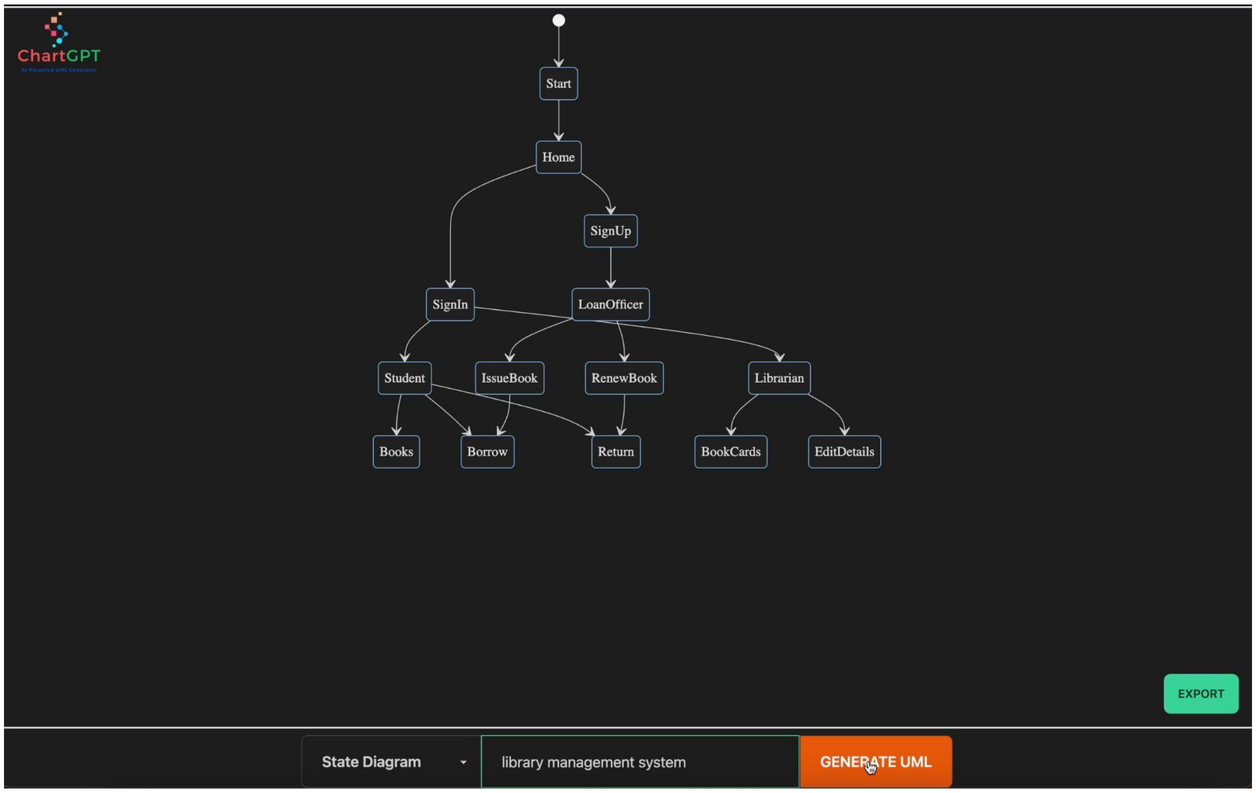


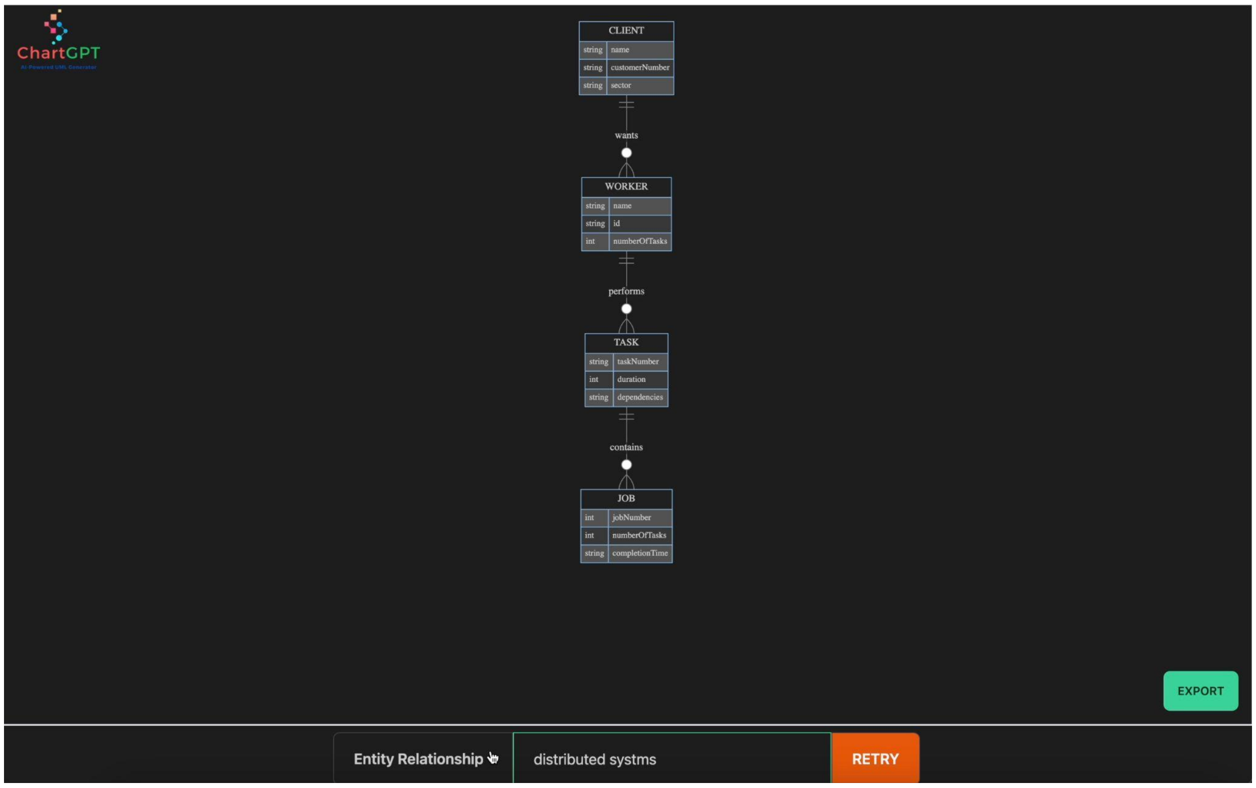

ChartGPT
Loading...
Sequence Diagram
library management system
GENERATE UML

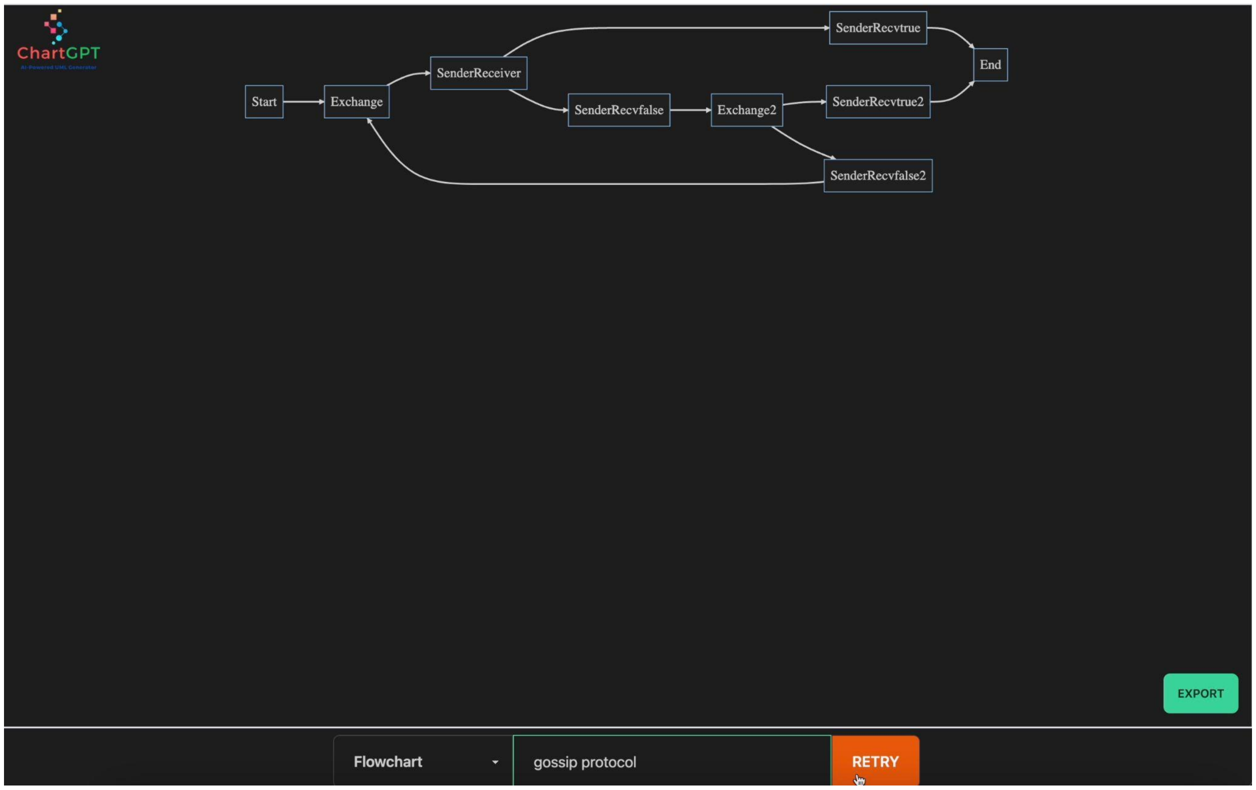

ChartGPT
Start
Input
Process
Check
Output
End
EXPORT
Timeline
library management system
GENERATE UML

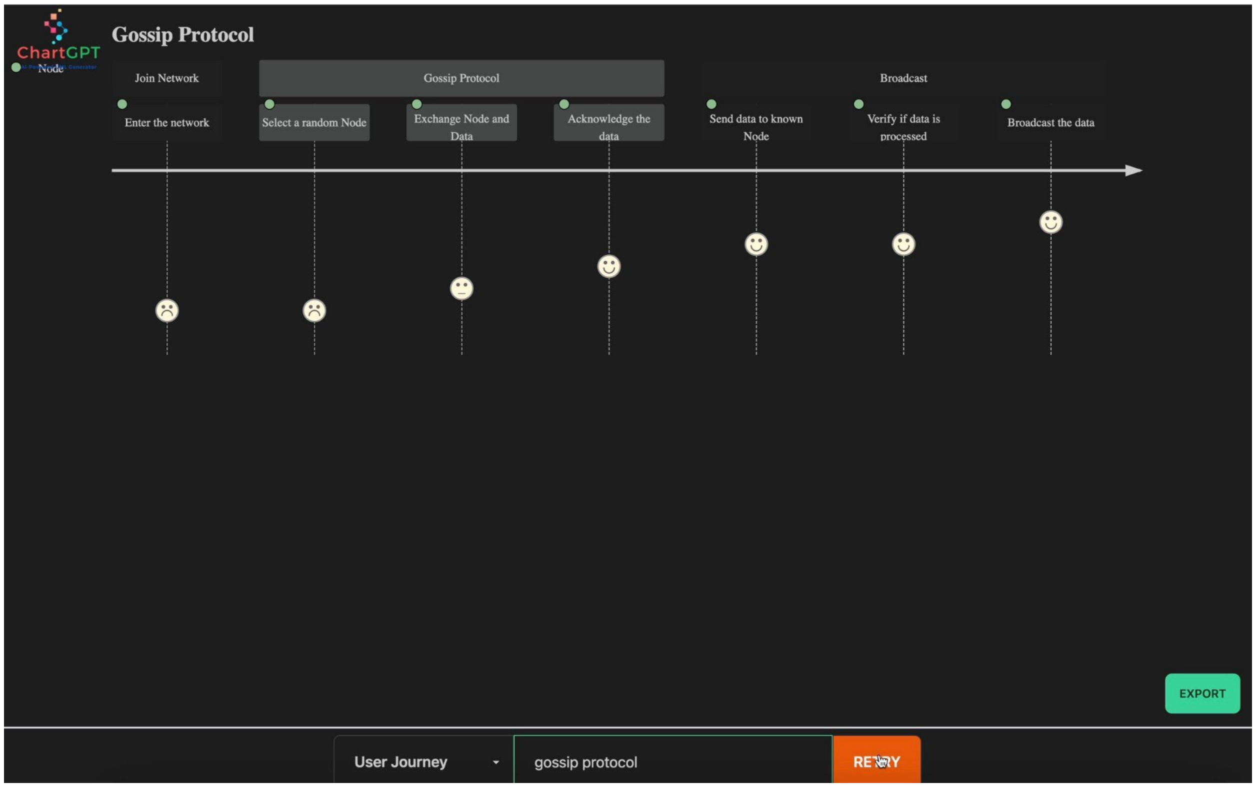

ChartGPT
Library Management System
Pre-launch
Design
Development
Testing
Launching
05/02/2020
25/02/2020
10/03/2020
15/04/2020
30/04/2020
01/05/2020
Analyze customer needs and requirement
Develop user stories
Wireframing
UI/UX Design
Database Creation
Back-end Development
Unit Testing
Functional Testing
Soft Launching
Official Launching
Post-Launch Support
EXPORT
Timeline
library management system
GENERATE UML


ChartGPT
Web Resources
Books
Magazines
Research Databases
Online Resources
Print Resources
Journals
Library Resources
Remove Books
Library Management System
Check Out
Library Catalog
Library Services
Add Books
Search
Check In
Overdue Notifications
Edit Records
EXPORT
Mindmap
library management system
GENERATE UML


ChartGPT
Start
Home
SignUp
SignIn
LoanOfficer
Student
IssueBook
RenewBook
Librarian
Books
Borrow
Return
BookCards
EditDetails
EXPORT
State Diagram
library management system
GENERATE UML

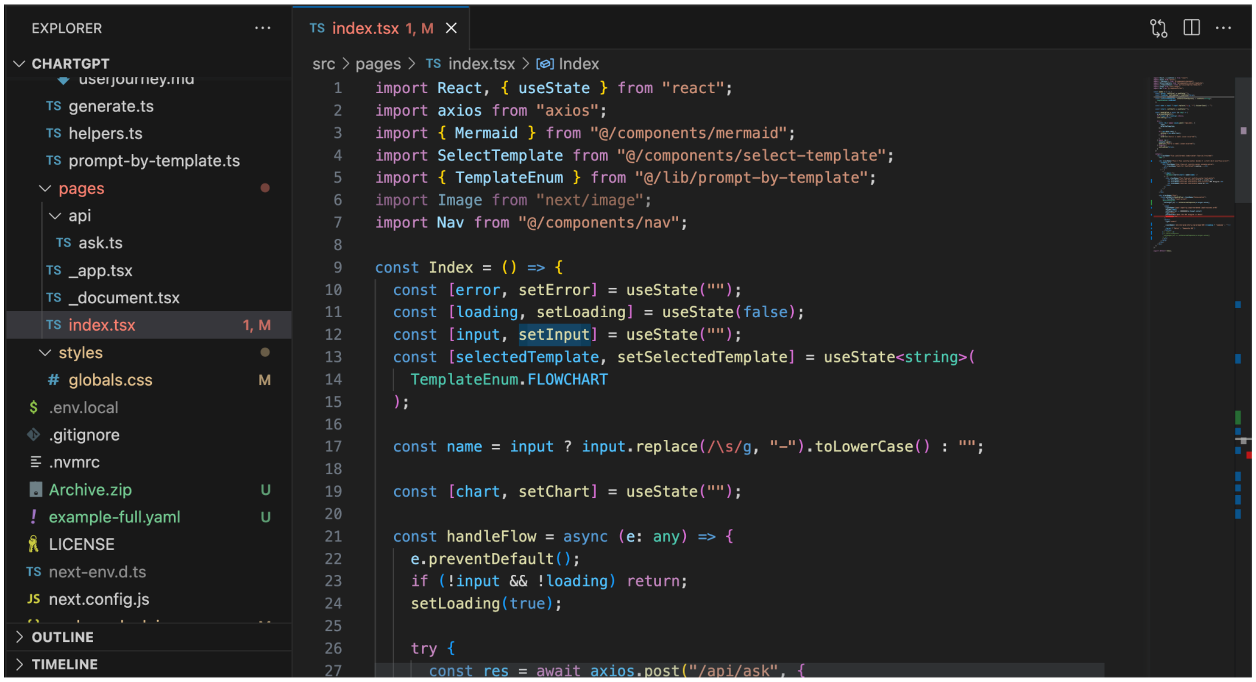

ChartGPT
CLIENT
string name
string customerNumber
string sector
wants
WORKER
string name
string id
int numberOfTasks
performs
TASK
string taskNumber
int duration
string dependencies
contains
JOB
int jobNumber
int numberOfTasks
string completionTime
EXPORT
Entity Relationship
distributed systms
RETRY


ChartGPT
Start
Exchange
SenderReceiver
SenderRecvtrue
SenderRecvfalse
Exchange2
SenderRecvtrue2
SenderRecvfalse2
End
EXPORT
Flowchart
gossip protocol
RETRY

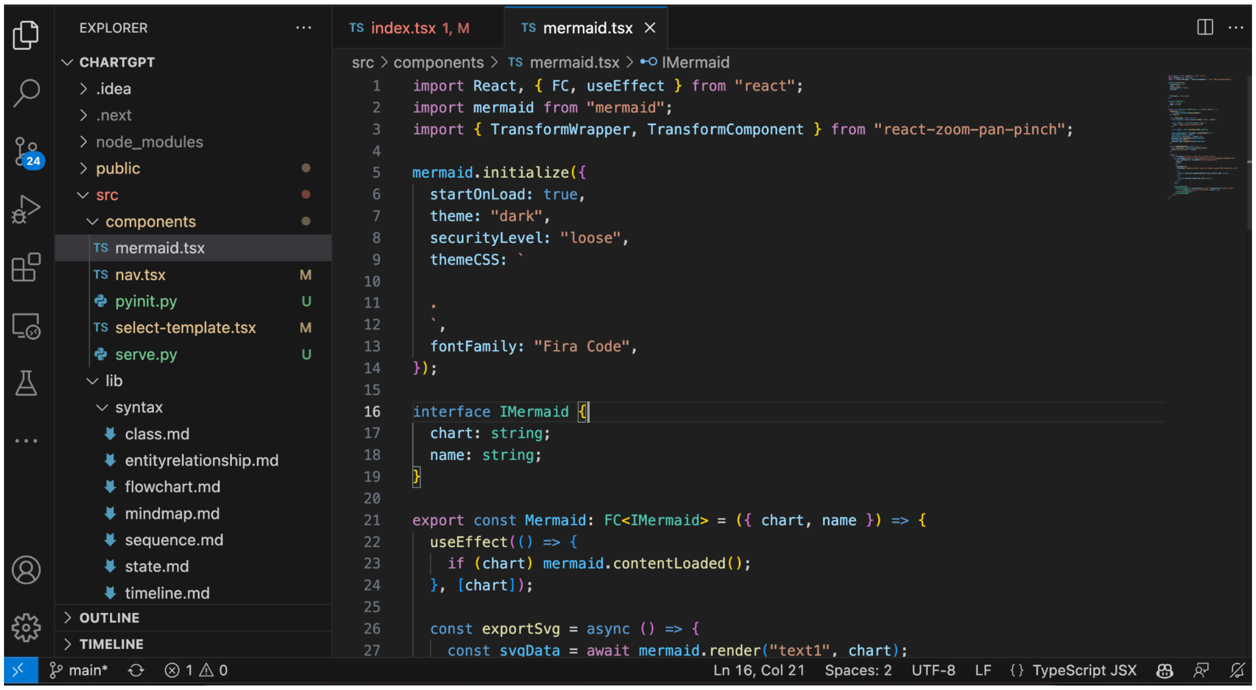

ChartGPT
Gossip Protocol
Node
Join Network
Gossip Protocol
Broadcast
Enter the network
Select a random Node
Exchange Node and Data
Acknowledge the data
Send data to known Node
Verify if data is processed
Broadcast the data
EXPORT
User Journey
gossip protocol
RETRY

Appendix D: Source Control and Backend Setup

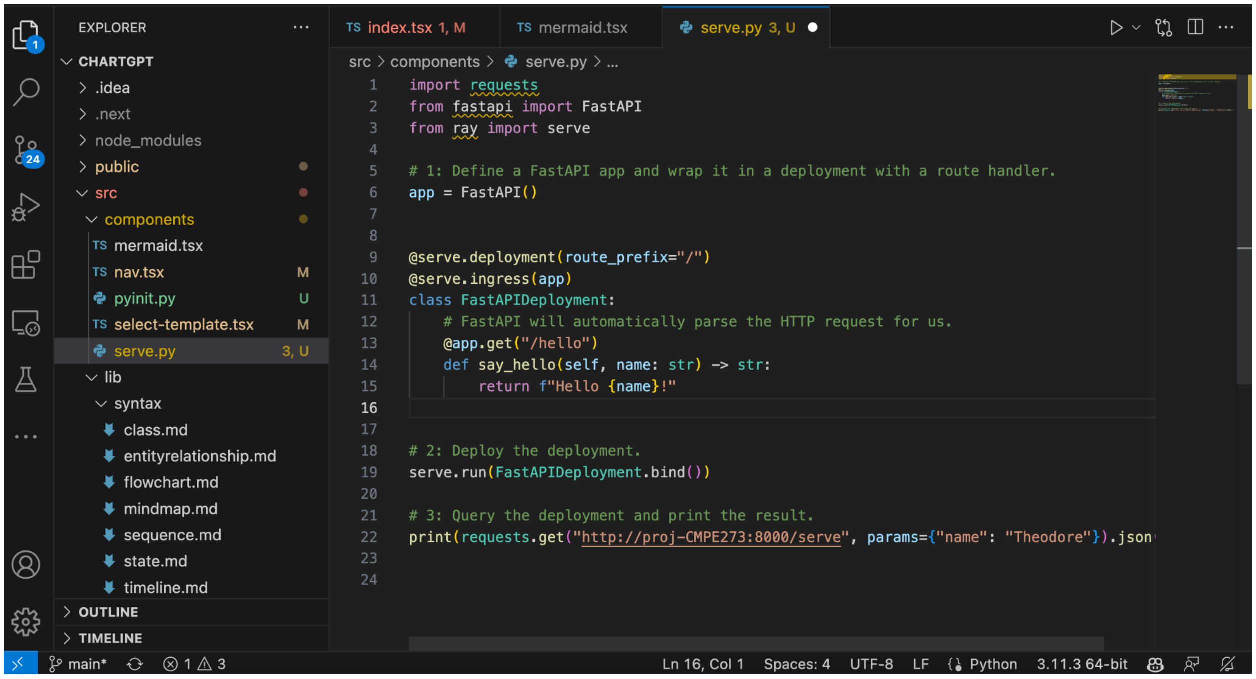
Git Repo : https://github.com/sagarsutar3/AI-based-UML-tool-ChartGPT



```tsx
import React, { useState } from "react";
import axios from "axios";
import { Mermaid } from "@/components/mermaid";
import SelectTemplate from "@/components/select-template";
import { TemplateEnum } from "@/lib/prompt-by-template";
import Image from "next/image";
import Nav from "@/components/nav";

const Index = () => {
  const [error, setError] = useState("");
  const [loading, setLoading] = useState(false);
  const [input, setInput] = useState("");
  const [selectedTemplate, setSelectedTemplate] = useState<string>(
    TemplateEnum.FLOWCHART
  );

  const name = input ? input.replace(/\s/g, "-").toLowerCase() : "";

  const [chart, setChart] = useState("");

  const handleFlow = async (e: any) => {
    e.preventDefault();
    if (!input && !loading) return;
    setLoading(true);

    try {
      const res = await axios.post("/api/ask", {
```



```tsx
import React, { FC, useEffect } from "react";
import mermaid from "mermaid";
import { TransformWrapper, TransformComponent } from "react-zoom-pan-pinch";

mermaid.initialize({
  startOnLoad: true,
  theme: "dark",
  securityLevel: "loose",
  themeCSS: `


  .
  `,
  fontFamily: "Fira Code",
});

interface IMermaid {
  chart: string;
  name: string;
}

export const Mermaid: FC<IMermaid> = ({ chart, name }) => {
  useEffect(() => {
    if (chart) mermaid.contentLoaded();
  }, [chart]);

  const exportSvg = async () => {
    const svgData = await mermaid.render("text1", chart);
```



```python
import requests
from fastapi import FastAPI
from ray import serve

# 1: Define a FastAPI app and wrap it in a deployment with a route handler.
app = FastAPI()


@serve.deployment(route_prefix="/")
@serve.ingress(app)
class FastAPIDeployment:
    # FastAPI will automatically parse the HTTP request for us.
    @app.get("/hello")
    def say_hello(self, name: str) -> str:
        return f"Hello {name}!"


# 2: Deploy the deployment.
serve.run(FastAPIDeployment.bind())

# 3: Query the deployment and print the result.
print(requests.get("http://proj-CMPE273:8000/serve", params={"name": "Theodore"}).json
```