|  |  |
| --- | --- |
|  | Proprietary  Information |

Orbit Research website Chatbot

Software Design Specification Template

12th June, 2023

Version 0.0

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Rev. | Date | Description of Changes | Author |
| 0.0 | 28th July, 2023 | Created | Dhruv |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Introduction

This Software Design Specification (SDS) document details the design and architectural decisions for a Multi-Document Chatbot application. This application ingests documents, breaks them down into chunks, indexes these chunks in a Pinecone vector database, and provides conversational capabilities using OpenAI's GPT-3.5-turbo (or GPT-4, if available). The chatbot will be deployed on the OrbitResearch website.

System overview

The system consists of two main phases: the Document Ingestion and Indexing phase, and the Chatbot Interface phase. During the Document Ingestion and Indexing phase, the system reads documents, breaks them into chunks, generates embeddings using the OpenAI API, and stores the embeddings in a Pinecone vector database. The Chatbot Interface phase uses the OpenAI API to retrieve relevant document chunks from the Pinecone database and generate responses.

The system consists of several interconnected components that work together to achieve its objectives. These components include:

1. Frontend: The frontend is built using NodeJS.
2. Ingestion Phase: The ingestion phase consists of storing the documents in a vector database. We are using Pinecone for this. Firstly, we need to create a index of dimensions 1536 and cosine metric. Creating namespace is optional. Using langchain, we are breaking down the documents into chunks of texts and using OpenAI embedding, we create embeddings that match with OpenAIs way to store text. After this, we store these embeddings in the vector DB which also provides a search query. We do this by storing the documents in a folder called doc in the project directory and run npm run ingest.
3. OpenAI: Using a Large Language Model, we query context, follow-up questions and stand-alone questions to OpenAI model 3.5 turbo. This could be any LLM, but OpenAI embeddings is the fastest and more efficient way to store data.

A summary of the requirements

Functional Requirements

* System Inputs:
  + User should be able to chat with the documents.
  + User should be able to ask any question related to company documents only.
* System Outputs:
  + Response to questions and follow up questions with a reference to what part of the documents were references.
  + Ability to answer non product questions in appropriate way.
* System Processing Needs:
  + System needs to process ingestion only once.
  + Keep track of Pinecone index so that it does not expire.

Non-functional requirements

* Response Time:
  + The system should provide prompt responses to user questions minimizing delays in response generation. Ideally close to real-time (O(n))
* Expected Accuracy:
  + Questions if detailed and appropriate, the answers should be accurate as the chance for querying similar data becomes sharper.
* Memory Footprints:
  + The system should efficiently utilize Pinecone index resources and also form appropriate query to handle delusional responses from OpenAI.

Strategies

Design Strategy

* The design of the system is modular, with different components handling different functions. The system is developed using Node.js and its ecosystem, following an Agile SDLC model with iterative development and testing cycles.
* Adoption of Technology: We have chosen to use the Django framework for building the web app. Django provides a robust and scalable platform for web development. It offers a wide range of features and libraries for handling user authentication, form handling, database management, and templating. If we choose not to use a web-app, the best standalone system would be a desktop application.
* Object-Oriented Approach: Our design follows an object-oriented approach, which promotes modularity, reusability, and maintainability. We encapsulate related functionalities into classes, ensuring that each class has a specific responsibility and is easy to understand and modify.
* SDLC: We will follow an Agile software development methodology, which allows us to iterate quickly, collaborate closely with stakeholders, and adapt to changing requirements. This approach emphasizes continuous integration, frequent communication, and iterative testing, enabling us to deliver working software in shorter cycles.
* Prioritization: We will prioritize development, integration, validation, and delivery milestones based on the critical functionalities and dependencies. We will focus on core features such as image uploading, description generation, and Graphiti integration initially. Subsequently, we will prioritize interactive feedback, filter application, and accessibility features based on their importance and impact on the user experience.
* Choice of Programming Language: We have chosen Python as the programming language for the web app. It provides us with the flexibility to integrate with different services and technologies. Django, being a Python web framework, offers a solid foundation for developing web applications with its extensive features and user-friendly nature. HTML, CSS, and JavaScript for UI/UX design.

These design strategies aim to ensure a scalable, modular, and maintainable web application that meets the functional and non-functional requirements. We aim to leverage appropriate technologies and methodologies to deliver a high-quality user experience.

Assumptions/Constraints

#### Business assumptions

* Resources are available for development and deployment.
* The budget is sufficient to cover the cost of the project.
* The scope of the project is defined in the SRS document.

#### Technical assumptions

* The application will be developed using Node.js.
* The application will perform reliably in various operating environments.

Dependencies

The critical dependencies for the system are as follows:

1. Server: The chatbot requires a server to be hosted. For the purpose of this chatbot, we will be attempting to use IONOS server.
2. Framework: The web app utilizes the NodeJS for the development. NodeJS provides the appropriate framework for handling HTTP requests and styling the frontend. The framework acts as a critical dependency for the development and functioning of the chatbot.
3. Libraries: Various libraries and packages are used within the system to support image processing, Azure AI integration, file handling, and other functionalities. These libraries provide additional functionality and integration capabilities, and their proper installation and integration are necessary for the web app's functionality.
4. Sub-systems: The system interacts with external sub-systems such as the Graphiti device and Azure AI services. It relies on proper communication and integration with these sub-systems to facilitate image display, image analysis, and other related functionalities.
5. HAL Library: The web app, if chosen, may require a Hardware Abstraction Layer (HAL) library specific to the Graphiti device. This library facilitates communication between the web app and the Graphiti hardware, ensuring compatibility and proper functioning.
6. Platform: The web app should be designed to run on different platforms, including Android, Windows, and others. It should be adaptable to various hosting environments and deployment platforms to ensure compatibility and smooth operation. If we decide to go with the stand-alone application, it should run on a windows based system.

Test strategies

Manual testing will be performed for the chatbot interface, and the boundary conditions will be thoroughly tested.

****System Deployment Environment****

The deployment plan details specific to your project are as follows:

1. Environment Details: a. Development Environment: The system will be developed in a web development environment using Django framework and Python programming language. Development machines with appropriate software development tools such as text editors, IDEs, and version control systems will be used for coding and collaboration among developers. b. Test Environment: The test environment will consist of servers, databases, and network infrastructure similar to the production environment. It will be set up to replicate the target deployment environment and ensure accurate testing of the system's functionalities, performance, and compatibility across different devices and browsers. c. Production Environment: The production environment will be the live environment where the system will be accessed by actual users. It will involve deploying the web application on an Android-based Graphiti device or Linux as the underlying operating system. The production environment will include the necessary hardware and software configurations to support the seamless operation of the system.
2. Schedule (Milestones): The key milestones in the development plan may include:
   * Completion of requirements gathering and analysis
   * Design and architecture definition
   * Implementation of core functionalities, including image upload, image description, filter application, and interactive feedback
   * Integration of Graphiti device connectivity and display functionality
   * Testing and quality assurance, including functional testing, usability testing, and accessibility testing
   * User acceptance testing and feedback incorporation
   * Deployment to the production environment on the Graphiti device
3. Rolling Plan Process: The rolling plan process for your project may involve the following steps:
   * Conducting an initial pilot release to gather early feedback and validate the system's usability and accessibility.
   * Incorporating feedback and making necessary improvements to enhance the user experience and address any identified issues or bugs.

Rollback Process: In the event of critical issues or unexpected problems during a release, a rollback process will be initiated. The rollback process will include:

* Reverting to a previously stable version or state of the system that was known to be functioning properly.
* Conducting thorough testing and investigation to diagnose and resolve the issue before proceeding with another release

****High level design (HLD)****

****Static View (structural details)****

****System decomposition****

System Decomposition: The system is divided into the following modules:

1. **Document Ingestion Module**: This module reads documents, breaks them into chunks, and generates embeddings for these chunks using the OpenAI API.
2. **Document Indexing Module**: This module stores the document chunks and their embeddings in a Pinecone vector database.
3. **Chatbot Interface Module**: This module retrieves relevant document chunks based on user queries, generates responses using the OpenAI API, and manages the chatbot interface on the OrbitResearch website.

****System Architecture****

The system follows a Client-Server architecture. The Document Ingestion and Indexing modules act as clients to the Pinecone and OpenAI APIs, which act as servers. The Chatbot Interface Module also acts as a client to the Pinecone and OpenAI APIs.

Data model, data design and data structures

The application uses the Pinecone vector database to store the embeddings of the document chunks. Each embedding is a 1536-dimensional vector, and cosine similarity is used for vector comparison. The document chunks are stored as plain text along with their corresponding embeddings.

1. Physical view

Diagram

Description automatically generated

1. Conceptual view

Diagram

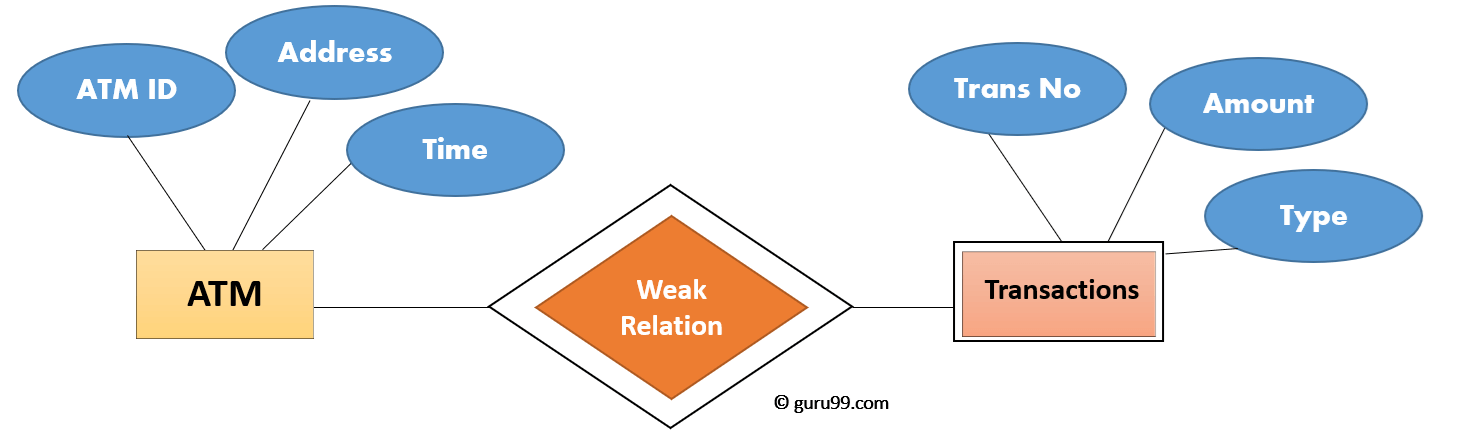
Description automatically generated

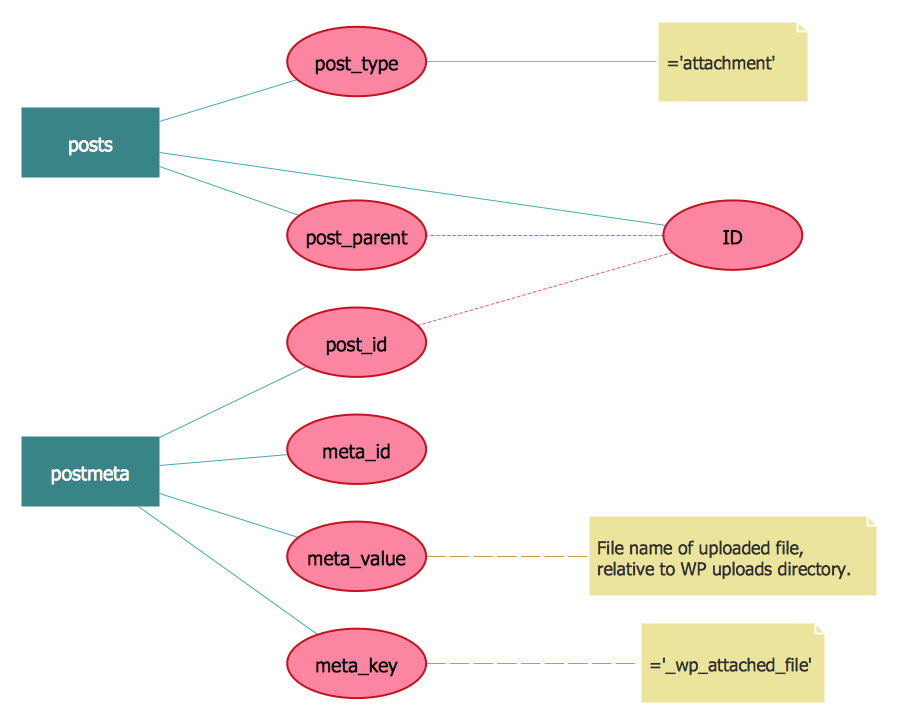
1. Logical view

Diagram

Description automatically generated

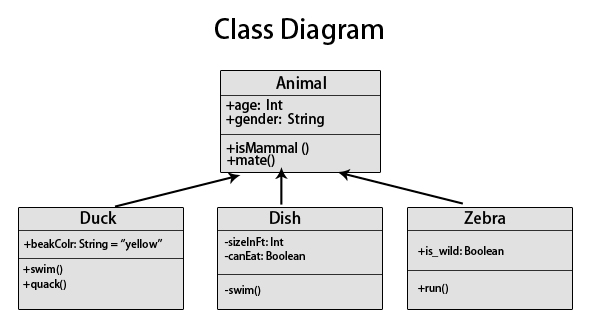
1. Relational structure

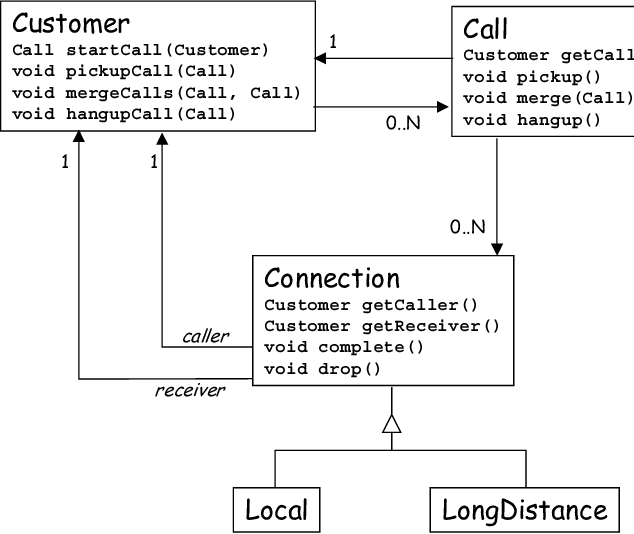




Class Diagram

1. Static view
2. Set of classes
3. Relationships between classes
4. Class Name
5. Class attributes
6. Class methods
7. Aggregation
8. Compositions
9. Dependency
10. Signals
11. Data types
12. Packages
13. Interfaces
14. Enumerations
15. Objects
16. Artifacts





1. **something goes wrong?)**

****Dynamic View (Behavioral details)****

**Describe behavioral aspects of your system**

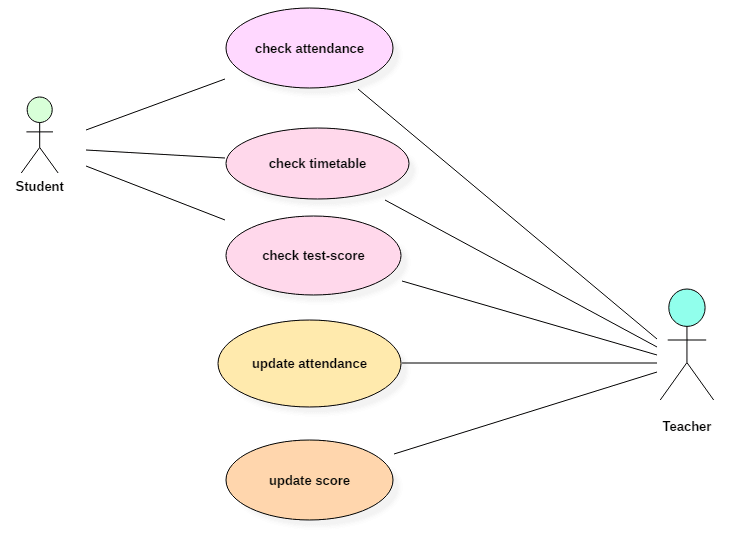
****Modes (Visible to users) / States (Internal to the system)****

**The change in the system behavior based on what state it is.**

Diagram

Description automatically generated

****Use cases****



****Data flow****

Diagram

Description automatically generated

****Sequencing****



****Design Risk****

It's important to identify and address potential risks associated with the proposed design of our project. Here are the specific risks we need to consider and our proposed mitigation plan:

1. New or Unproven Technology: Risk: Incorporating new or unproven technologies for integrating the Graphiti device may result in compatibility issues or unexpected limitations. Mitigation: We will thoroughly research the Graphiti device's technology stack, ensuring its compatibility with our web app and considering its stability and community support. Additionally, we will conduct small-scale prototypes and test the technology stack before full implementation to mitigate any unforeseen challenges.
2. Unavoidable High Coupling and Low Cohesion: Risk: The integration with the Graphiti device may introduce some coupling and decrease cohesion due to its proprietary nature and specific communication protocols. Mitigation: While striving for modular and loosely coupled code, we will document the integration points and communication protocols clearly to minimize the impact of coupling. We will also implement well-defined interfaces and encapsulation techniques to maintain code maintainability and flexibility.
3. Boundary Conditions: Risk: Inadequate handling of boundary conditions, such as handling large images or complex interactions, may lead to system instability or unexpected behavior. Mitigation: We will conduct extensive testing, including boundary testing, to ensure our system handles various scenarios effectively. We will implement input validation mechanisms, error handling routines, and performance optimizations to handle boundary conditions gracefully and enhance overall system stability.

****Error handling****

To ensure error-free operation and a seamless user experience, we will implement effective error handling measures specific to our project:

1. List of Potential Error Scenarios and Handling: We will identify potential error scenarios, such as connection failures with the Graphiti device, image processing errors, or invalid user inputs, and define specific error handling procedures for each scenario. This includes providing informative error messages, offering guidance for troubleshooting, and implementing appropriate error recovery mechanisms.
2. Input Parameters Validation Methods: We will implement robust input validation mechanisms to validate user inputs and prevent potential issues. This includes data type validation, range checks, and input sanitization techniques to ensure the Graphiti device receives valid and reliable data.
3. Breaking Infinite Loops: We will incorporate safeguards and timeout mechanisms within our code to detect and prevent infinite loops or excessive processing time. This will ensure that our system remains responsive and prevents resource exhaustion in case of unexpected loops.
4. Runtime Errors: We will implement structured exception handling techniques to catch and handle runtime errors, such as null pointer exceptions or memory allocation failures. This includes logging detailed error information, providing user-friendly error messages, and implementing appropriate error recovery strategies to maintain system stability.
5. Logging of Errors: We will integrate a comprehensive logging mechanism to record and monitor errors that occur during system operation. This logging will assist in diagnosing issues, tracking error patterns, and facilitating future system improvements. Detailed logs will be stored for analysis and troubleshooting purposes.

****Low level design (LLD)****

**Give details of the internal of actual execution and code for each independent modules.**

1. **APIs and return values**
2. **Algorithm details (time complexity)**
3. **Local variables and data structures**
4. **Shared (global) data**
5. **Dependencies with other modules**
6. **Flowcharts**
7. **Unit test APIs**
8. APIs and Return Values:

* Computer Vision API: For now, we have used Microsoft Azure’s Computer Vision API for the purpose of testing. Any API that will be selected should satisfy Computer Vision service, including image description and any other relevant functionalities.
* Bounding Box or Image Segmentation API: For now, we have chosen Microsoft Azure’s Computer Vision API which also gives the functionalities of bounding box or image segmentation service. It returns in object, the coordinates for the bounding box and the object name.
* Accessibility API (ResponsiveVoice): Any API or [alt feature of HTML] should satisfy the accessibility feature that reads texts displayed on the screen.

Graphiti APIs: Identify and describe the inbuilt APIs provided by Graphiti for image transfer, touch feedback, and other related functionalities. Specify the input parameters, expected data types, and return values for each Graphiti API.