SE Project-3 Report

Table of Contents

Team Members

1) Vineeth Bhat (2021101103)

2) Ishwar B Balappanawar (2021101023)

3) Swayam Agrawal (2021101068)

4) Mitansh Kayathwal (2021101026)

5) G.L.Vaishnavi (2023204009)

Problem Statement

Introduction

Our project endeavors to create an application software geared towards training, testing, and logging Machine Learning (ML) Models directly online. Drawing inspiration from established AI Development Platforms like *Weights & Biases* and *Kaggle*, which already offer similar functionalities, we aspire to elevate the user experience to the next level. Our application, **Model Hive**, aims to streamline the process of working with diverse ML models. Users will have the capability to effortlessly add or remove models, retrain, fine-tune, and execute inferences on datasets of their choosing, thereby facilitating various analyses.

Model Hive stands out as a user-friendly interface designed to empower individuals in their ML endeavors. It offers a comprehensive suite of features, including model management, training, and inference generation. By prioritizing ease of use and flexibility, our application seeks to make a little something that can be extended for use in the rapid growth of methods that democratize ML experimentation and exploration, fostering innovation and discovery in the field.

Code : https://github.com/Mitanshk01/SWE-Project-3

Task 1: Requirements and Subsystems

Feature Requirements

Functional Requirements

- **User Management**
  - Sign Up
  - Login
  - Logout
- **ML Models Management**
  - Association with a Repository
  - Repository Operations
    - Upload
      - Detailed Model Upload
      - Data Upload
        - Creation of New Datasets
        - Addition to Existing Datasets
    - Deletion of Repositories, Models, and Data
- **Model Functionality**
  - Training
  - Retraining
  - Fine-tuning

        - Addition of New Data
    - Inference
        - Metadata Extraction
        - Variable Loading (logging)
        - Data Visualization
    - Logging of results

Non-Functional Requirements

- **Security**
    - Implementation of JWT for user authentication.
    - Utilized Google Drive for data security.
- **Performance**
    - Ensures responsive changes within 0.5 seconds of user interaction.
    - Supports multiple concurrent users performing various tasks simultaneously.
    - *Number of Concurrent Users*: Supports up to 5 concurrent users to account for the anticipated low throughput on the local system.
- **Modifiability**
    - *Modular Design*: Utilizes clear, non-coupled modules following appropriate design patterns.
    - *Configurability*: Allows easy adjustment of parameters and settings for model training, testing, and inference to accommodate diverse user preferences.
    - *Version Control*: Implements version control mechanisms to track changes and facilitate comparison or rollback of different versions.
- **Testability**

    - *Individual Component Testing*: Ensures each subsystem can be tested independently to maintain robustness.
    - *Logging and Monitoring*: Integrates logging and monitoring tools to track application behavior, identify issues, and collect performance metrics during testing.
- **Usability**
    - *Intuitive Interface*: Provides a user-friendly interface with intuitive navigation to minimize the learning curve for new users.
    - *Feedback Mechanism*: Offers error messages, notifications, and progress indicators to keep users informed about the status of their tasks and any encountered issues.
- **Resource Limits**
    - *Dataset Size Limit*: Allows datasets of up to 10 GB, with the option to increase if needed.
    - *Code File Size Limit*: Limits code files to 30 MB, with the option to increase for larger projects.
    - *Logging File Limits*: Sets a maximum size of 30 MB for log files, with the option to adjust as necessary.
    - *Number of Repositories*: Supports up to 1000 repositories, with scalability options for future expansion.
- **Availability**
    - Aims for 99% availability to ensure reliable access to the application.

Architectural Significance

The architectural significance of the listed requirements lies in their pivotal roles in shaping the design and functionality of the system.

- **Security**

    - Implementation of JWT for user authentication ensures secure access to the system, crucial for protecting sensitive user data and preventing unauthorized access.
    - Utilizes Google Drive for data security, requiring robust integration and handling of authentication and authorization to ensure data confidentiality and integrity.
- **Performance**
    - Ensures responsive changes within 0.5 seconds of user interaction, enhancing user experience and engagement with the application.
    - Supports multiple concurrent users performing various tasks simultaneously, indicating the system's ability to handle workload efficiently and maintain responsiveness under heavy usage.
    - Supports up to 5 concurrent users to account for the anticipated low throughput on the local system, ensuring optimal performance and user satisfaction even under resource constraints.
- **Resource Limits**
    - Allows datasets of up to 10 GB, with the option to increase if needed, ensuring efficient storage management and performance optimization as data volume grows.
    - Limits code files to 30 MB, with the option to increase for larger projects, facilitating faster builds and reducing the risk of performance degradation due to large codebases.
    - Sets a maximum size of 30 MB for log files, with the option to adjust as necessary, ensuring efficient resource utilization and preventing excessive disk usage.
    - Supports up to 1000 repositories with scalability options for future expansion, requiring a robust architecture for managing and organizing data effectively to handle increasing workload and user demands without compromising performance or stability.

Subsystems

We have three subsystems within our implementation:

1. User Management

- Facilitates user authentication, registration, and profile management.
- Allows users to securely register accounts and provide necessary credentials.
- Grants access to personalized features and functionalities upon registration.
- Ensures the security and integrity of user accounts through robust authentication mechanisms.
- Includes features such as password recovery and user roles management for enhanced security and user experience.

2. Repository Management

- Organizes and stores project-related resources within the system.
- Enables users to create and manage within repositories. As of yet, we do not support collaboration.
- Provides a structured environment for storing code, data, and project results.
- Allows users to upload code and data, manage versions, and share repositories with collaborators.
- Facilitates collaboration, change tracking, and version history for efficient project management.

3. Model Management

- Empowers users to harness machine learning capabilities effectively within the system.
- Manages, trains, and evaluates machine learning models for various tasks.
- Allows users to upload, view, and select models for training or inference.
- Provides features for model training, hyperparameter tuning, and performance evaluation.

- Enables monitoring of model training progress, result analysis, and performance comparison.
- Includes functionalities for model deployment, serving predictions, and real-time performance monitoring.

Task 2: Architecture Framework

IEEE 42010 Stakeholder Indentification

Introduction

This serves as an architecture description for **Model Hive**, an application software designed to streamline the training, testing, and logging of Machine Learning (ML) Models online.

Context

The quickly developing fields of artificial intelligence and machine learning are the setting in which Model Hive operates. The need for tools that streamline the development and deployment process is only going to increase as more and more organizations turn to machine learning (ML) models for creativity and decision-making. In order to meet this need, Model Hive offers a platform that makes it easy for customers to maintain, train, and run machine learning models online. **Model Hive** promotes cooperation and experimentation in ML research and development while guaranteeing the confidentiality and integrity of user data by utilizing Google Drive for data security and JWT for user authentication.

Stakeholder Identification

a. End-Users:

- **Description:** Individuals and organizations utilizing the **Model Hive** application for training, testing, and logging Machine Learning (ML) Models online.
- **Role:** Actively engage with the application to perform tasks such as model management, training, retraining, fine-tuning, inference, and logging of results.
- **Concerns/Interests:**
    - Ease of use and intuitive interface.
    - Performance responsiveness during model training and inference.
    - Security of data and user authentication.
    - Availability and reliability of the application.
- **Influence Level:** High, as their satisfaction and feedback directly impact the success and adoption of the application.

b. Service Runners:

- **Description:** Individuals or teams responsible for running the **Model Hive** application and managing its resources, including servers, databases, and other infrastructure components.
- **Role:** Ensure the smooth operation and availability of the application by managing its technical infrastructure, monitoring performance, and addressing any issues or maintenance tasks.
- **Concerns/Interests:**
    - Performance optimization and resource allocation.
    - Scalability to accommodate increasing user demand.
    - Security measures to protect the application and its data.
    - Availability targets and uptime management.
- **Influence Level:** High, as they play a critical role in maintaining the functionality and performance of the application.

c. Developers:

- **Description:** The team responsible for designing, developing, and maintaining the **Model Hive** application software.
- **Role:** Create and enhance features, implement architectural components, and ensure the overall quality, maintainability, and scalability of the application.
- **Concerns/Interests:**
  - Modifiability and maintainability of the codebase.
  - Compliance with architectural standards and best practices.
  - Testability and debugging capabilities.
  - Collaboration tools and version control mechanisms.
- **Influence Level:** High, as they drive the technical direction and implementation of the application, directly impacting its functionality, performance, and long-term viability.

Stakeholder Concerns

End-Users:

- **Ease of Use:** End-users prioritize an intuitive interface and seamless navigation to efficiently perform tasks.
- **Performance:** Responsive interactions during model training and inference are crucial for user satisfaction.
- **Security:** Ensuring the confidentiality and integrity of user data through robust authentication mechanisms.
- **Availability:** Reliable access to the application is essential to support users' ongoing ML activities.

Service Runners:

- **Performance Optimization:** Optimizing resource allocation and scalability to meet increasing user demand.

- **Security Measures:** Implementing measures to safeguard the application and its data against potential threats.
- **Availability Targets:** Meeting uptime targets and effectively managing system maintenance and updates.

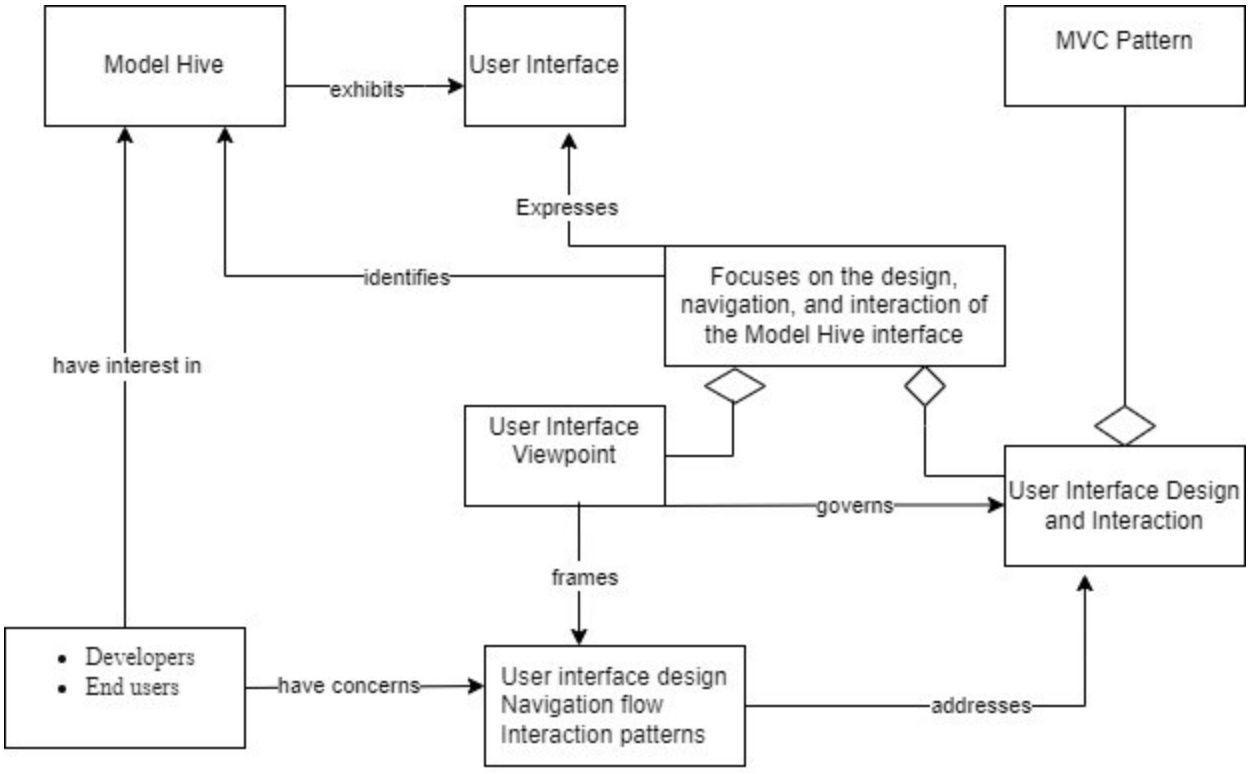
Developers:

- **Modifiability:** Designing a modular and maintainable codebase to facilitate future enhancements and updates.
- **Compliance with Standards:** Ensuring adherence to architectural standards and best practices to support long-term viability.
- **Testability:** Incorporating robust testing and debugging capabilities to maintain application reliability.
- **Collaboration Tools:** Utilizing version control mechanisms and collaboration tools to enhance developer productivity and coordination.

Architectural Views and Viewpoints

1. User Interface Viewpoint:

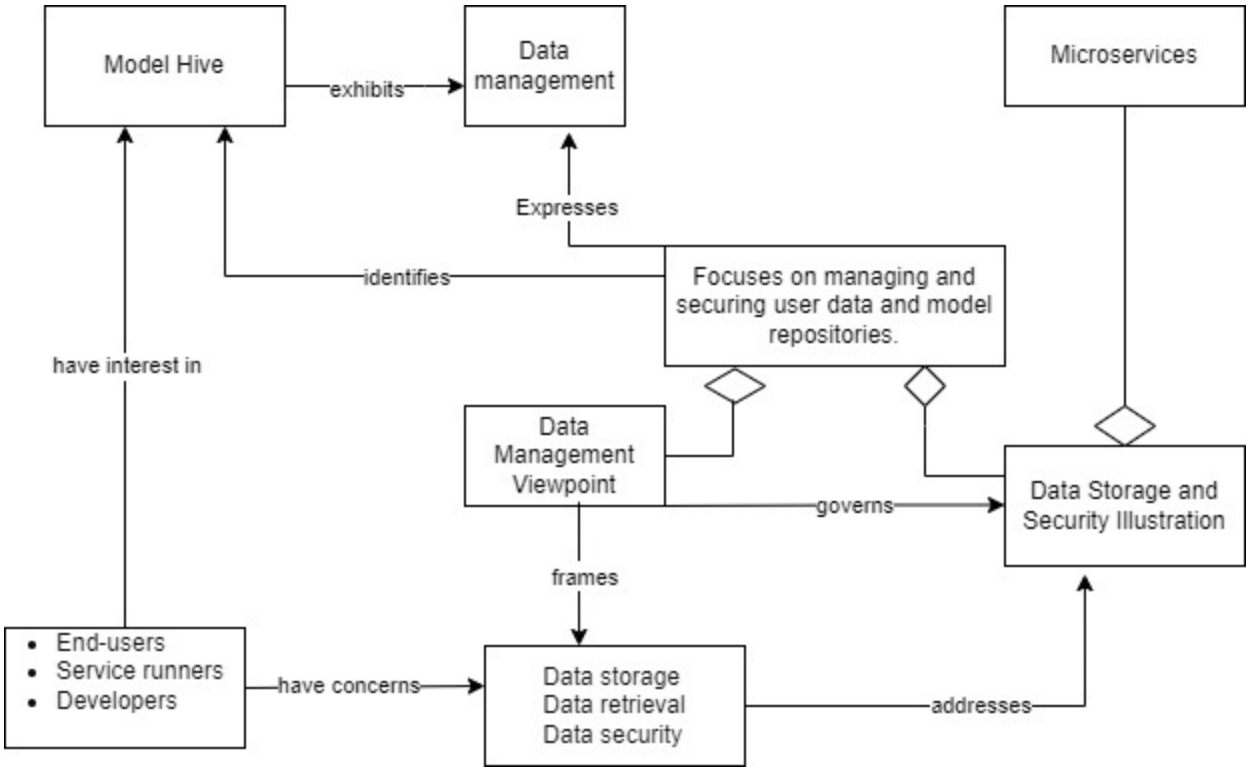
- **Concerns:** User interface design, navigation flow, and interaction patterns.
- **View Description:** Describes the layout, components, and interactions of the **Model Hive** interface.
- **Stakeholders:** End-users, developers.



Untitled Diagram.jpg

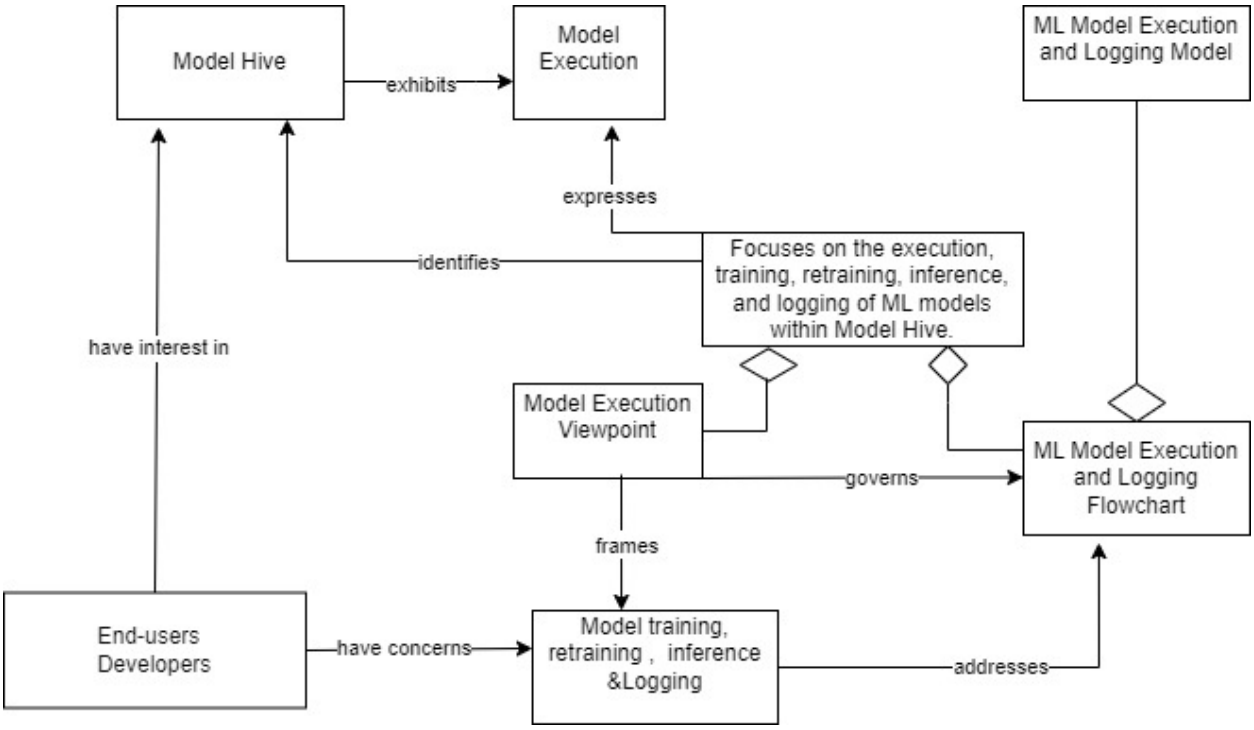
2. Data Management Viewpoint:

- **Concerns:** Data storage, retrieval, and security.
- **View Description:** Illustrates how user data and model repositories are managed and secured.
- **Stakeholders:** End-users, service runners, developers.



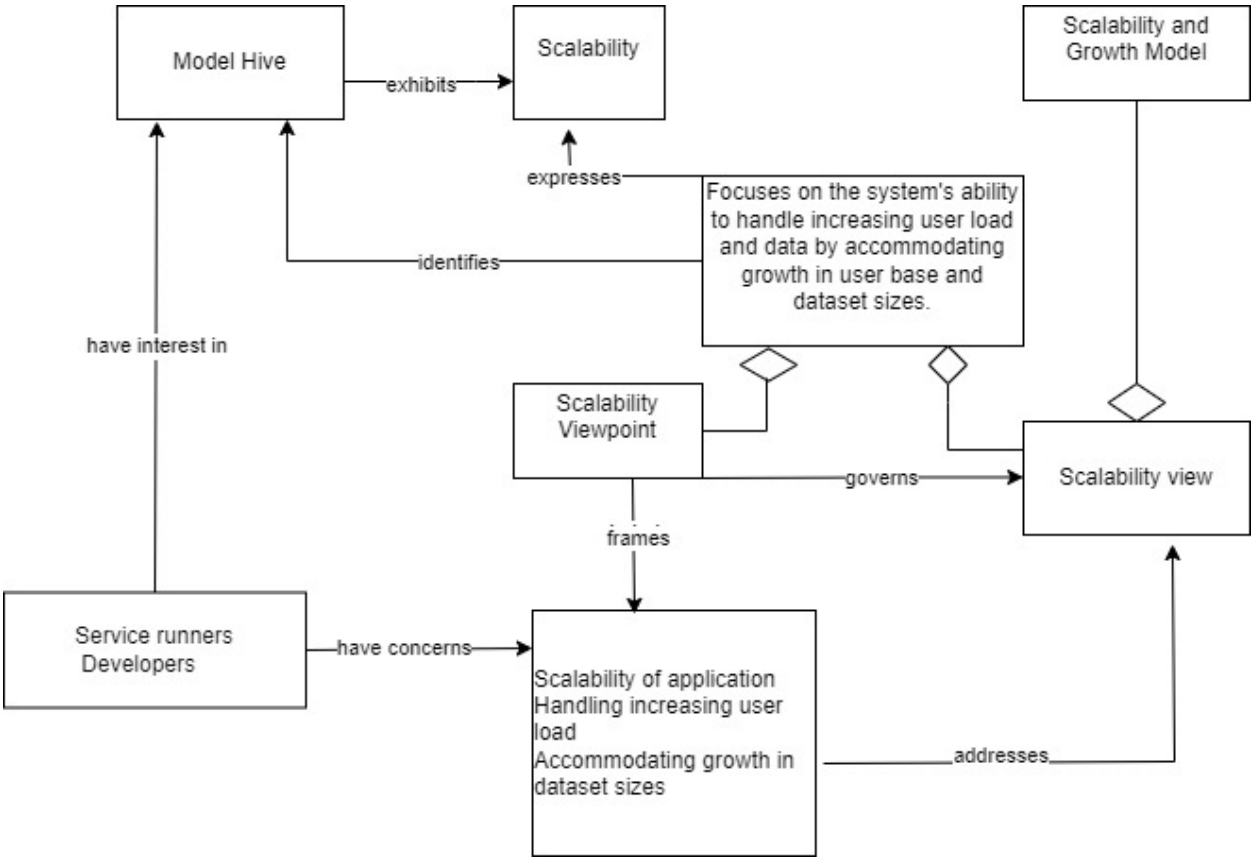
3. Model Execution Viewpoint:

- **Concerns:** Model training, retraining, inference, and logging.
- **View Description:** Outlines the processes and interactions involved in executing ML models within **Model Hive**.
- **Stakeholders:** End-users, developers.



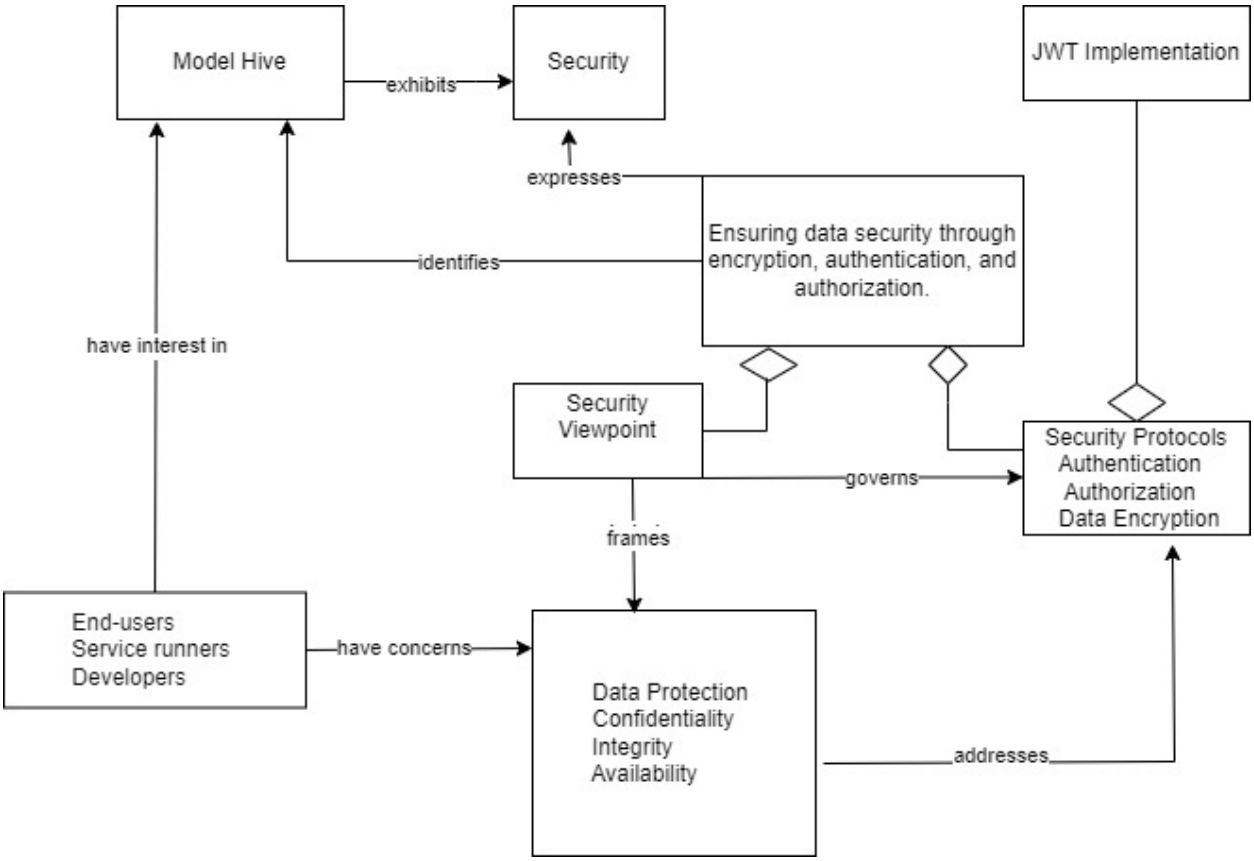
4. Scalability Viewpoint:

- **Concerns:** Scalability of the application to handle increasing user load and data volume.
- **View Description:** Explores how the system architecture accommodates growth in user base and dataset sizes.
- **Stakeholders:** Service runners, developers.



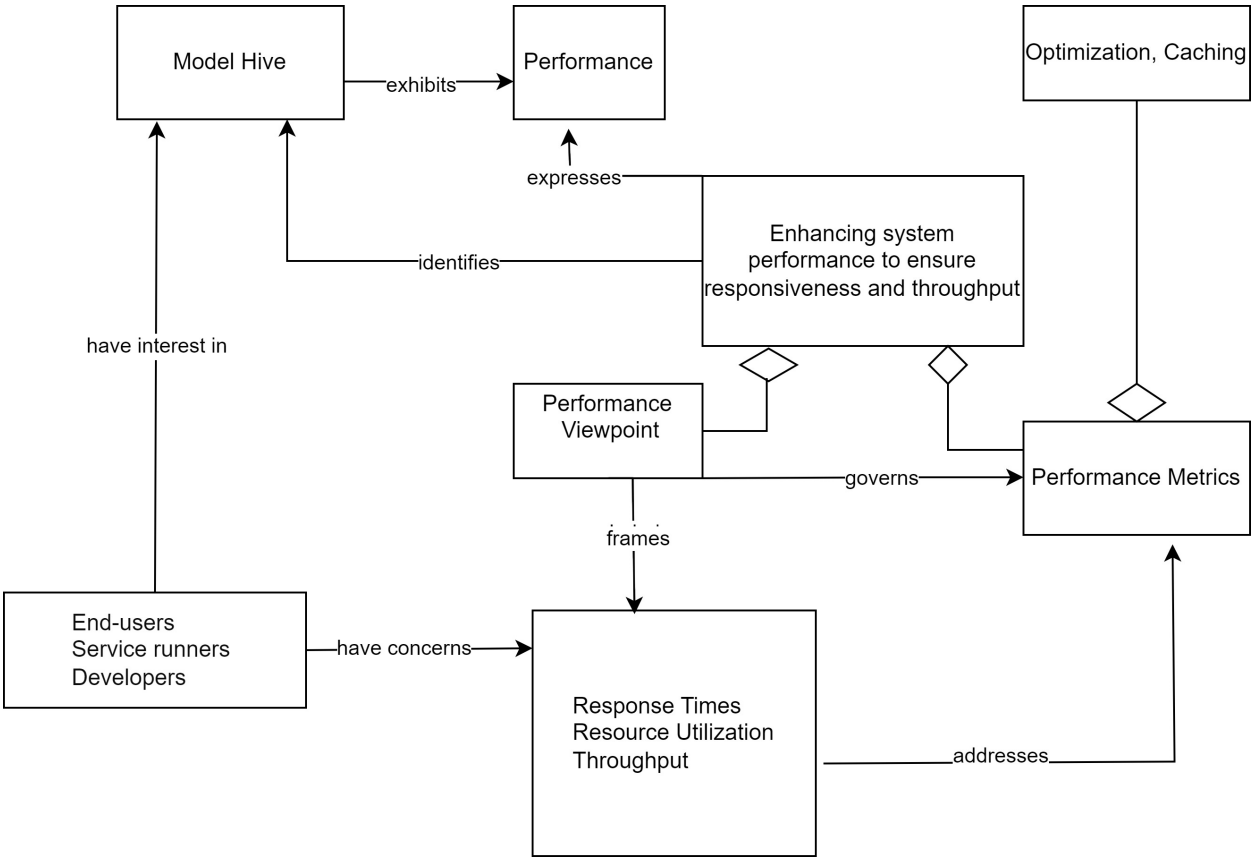
5. Security Viewpoint:

- **Concerns:** Ensuring the confidentiality, integrity, and availability of user data and application resources.
- **View Description:** Details the security measures implemented at various levels of the application, including authentication, authorization, and data encryption.
- **Stakeholders:** End-users, service runners, developers.



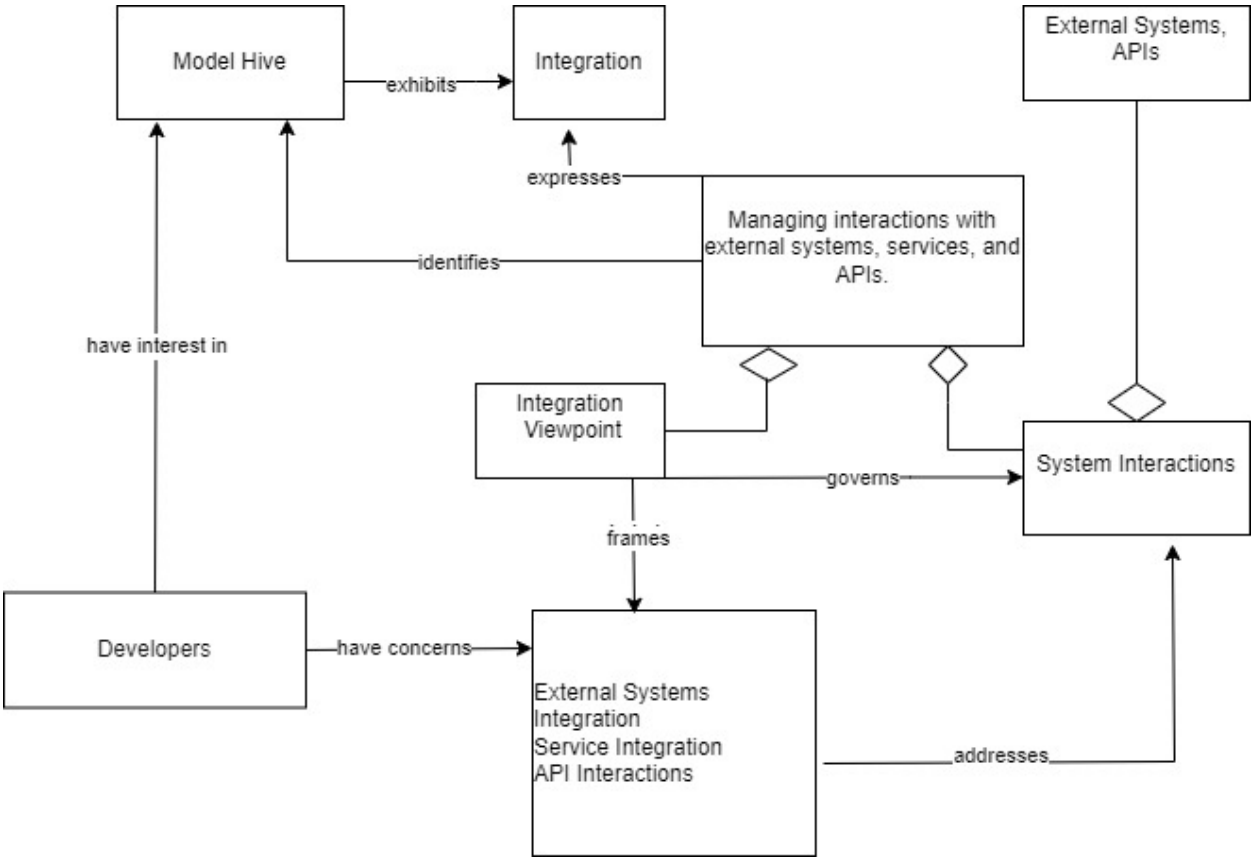
6. Performance Viewpoint:

- **Concerns:** Optimizing system performance to meet responsiveness and throughput requirements.
- **View Description:** Analyzes the performance characteristics of the system, including response times, resource utilization, and throughput.
- **Stakeholders:** End-users, service runners, developers.



7. Integration Viewpoint:

- **Concerns:** Integration with external systems, services, and APIs.
- **View Description:** Describes how **Model Hive** interacts with other systems and services, including data sources, model repositories, and third-party tools.
- **Stakeholders:** Service runners, developers.



Major Design Decisions

Rejected Architecture Decision Record (ADR) - Using a Serverless Architecture

Context:

When considering our system architecture, we evaluated the possibility of adopting a serverless architecture. Serverless architectures offer benefits such as reduced operational overhead, automatic scaling, and cost savings. However, after careful consideration, we decided against implementing a serverless architecture for our project.

Decision:

Reject the adoption of a serverless architecture for our project.

Rationale:

- **Performance Concerns**: While serverless architectures offer automatic scaling, they may introduce **latency** due to cold starts, which could impact the responsiveness of our application, especially for real-time interactions.
- **Complexity**: Serverless architectures often require breaking down applications into smaller, stateless functions, which can lead to increased complexity in managing dependencies, workflows, and monitoring.
- **Vendor Lock-in**: Adopting a serverless architecture ties us more closely to a specific cloud provider's services and APIs, limiting our flexibility and portability in the future.
- **Cost Considerations**: While serverless architectures can offer cost savings for certain workloads, they may not be the most cost-effective option for our project, especially considering potential hidden costs such as data transfer and service limits.

Consequences:

- **Operational Overhead**: By not adopting a serverless architecture, we may need to manage more **infrastructure components and resources manually,** increasing operational overhead for our team.
- **Scalability Challenges**: While serverless architectures offer automatic scaling, we will need to carefully manage and optimize our infrastructure to ensure scalability and performance under varying loads.
- **Resource Management**: Without the automatic resource provisioning and scaling provided by serverless platforms, we will need to implement our own strategies **for managing resources efficiently.**

Status:

Rejected

Date:

2024-04-06

Architecture Decision Record (ADR) - File Storage Strategy for Microservice

Context:

We need a clear plan for managing our project files, which include large datasets and code files, along with other miscellaneous files. These files have varying sizes and purposes, and we're aiming for a strategy that's efficient, scalable, and easy to handle. With our network's limitations for transferring large datasets, we require an external storage solution.

Decision:

1. Store large datasets in Google Drive.
2. Store code files and other non-data files in Google Drive.
3. Maintain the dataset structure as `repo_name/data/train/file_name.zip`, `repo_name/data/val/file_name.zip`, and `repo_name/data/test/file_name.zip`.

Alternatives Considered:

1. **Database Storage (BLOBs):**
   Storing files directly in the database simplifies data management but can increase database size and impact performance, especially for large files.
2. **Local File System:**
   Saving files to the local file system is straightforward but may not be suitable for distributed or scalable architectures.
3. **Cloud Storage Services:**
   Using services like Amazon S3 or Azure Blob Storage offers scalability, reliability, and offloads storage management from the microservice.
4. **Distributed File Systems:**
   Systems like Hadoop Distributed File System (HDFS) or GlusterFS are designed for large-scale data storage and distributed environments.
5. **Content Delivery Networks (CDNs):**
   CDNs distribute files globally with low latency by caching them in multiple locations, improving performance for users worldwide.
6. **Hybrid Approaches:**
   Combining multiple storage solutions, such as storing small files in the database and larger files in cloud storage, offers flexibility.

Rationale:

- **Large Datasets**: Transmitting large datasets over our network would be inefficient. Google Drive offers a reliable external storage solution for these datasets.
- **Google Drive for All File Types**: With fast APIs and our access privileges as IIIT admins, Google Drive serves as a consistent and simple storage solution for both datasets and code files.
- **Local Storage for Code Files**: While code files could be stored locally, Google Drive's accessibility and API speed make it a preferable option for unified file management.
- **Simplicity and Accessibility**: Using Google Drive for all files simplifies storage management, reducing the need to juggle multiple storage solutions.
- **Cost and Scalability**: Google Drive provides free storage up to a certain limit, aligning with our storage needs. Monitoring storage usage will help us avoid additional costs.

Consequences:

- **Scalability**: Storing both datasets and code files on Google Drive allows us to scale storage as needed without affecting the microservice's performance.
- **Cost**: We'll need to monitor storage usage to ensure we stay within the free limit to avoid extra costs.
- **Security**: Safeguarding our Google Drive access tokens will ensure that only authorized microservices can access and manage the files.
- **Local Storage**: Although code files could be stored locally, we'll rely on Google Drive for its benefits in accessibility and management.
- **Dependency on Google Drive**: Our microservice's availability and performance may be influenced by Google Drive's uptime and API responsiveness.

Status:

Accepted

Date:

2024-04-09

Architecture Decision Record (ADR) - Choosing Microservices Architecture Over Monolithic

Context:

When designing our system, we had to decide between a monolithic architecture and a microservices architecture. A monolithic approach could offer initial speed and simplicity, but we opted for microservices due to several advantages that align better with our project's requirements and future scalability.

Decision:

Adopt a microservices architecture for our system.

Alternatives Considered:

1. **Monolithic Architecture:**
   A single, unified codebase and deployment unit where all components are tightly coupled.
2. **Microservices Architecture:**
   Decomposing the system into smaller, independently deployable services, each responsible for a specific functionality or business domain.

Rationale:

- **Scalability**: Microservices allow independent scaling of different components based on demand. This flexibility ensures efficient resource utilization and cost-effectiveness.
- **Modularity and Maintainability**: Microservices promote a modular approach, making the system easier to understand, maintain, and update. This leads to faster development cycles and easier debugging.
- **Fault Isolation**: With microservices, failures are contained within the affected service, reducing the impact on the entire system and improving fault tolerance.

- **Technology Diversity**: Microservices offer the flexibility to choose the most suitable technology stack for each service, optimizing performance and resource usage.
- **Improved Performance**: Individual optimization of microservices for performance ensures better responsiveness and overall system performance.

Consequences:

- **Complexity**: Managing multiple services can introduce complexity in terms of deployment, monitoring, and inter-service communication.
- **Data Consistency**: Ensuring data consistency across microservices can be challenging and requires careful planning and implementation.
- **Operational Overhead**: Each microservice needs to be managed and monitored separately, which can increase operational overhead.
- **Development Overhead**: Developers need to be aware of the interdependencies between services and follow best practices to avoid breaking changes.

Status:

Accepted

Date:

2024-04-09

Architecture Decision Record (ADR) - Handling Large File Uploads from React Frontend to Flask Backend

Context:

Our architecture involves a React frontend communicating with a Flask backend, which in turn calls various microservices following the Backend for Frontend (BFF) pattern. Sending large files as a single chunk could lead to performance issues and potential timeouts. Therefore, a solution is needed to handle large file uploads efficiently while maintaining a responsive user experience.

Decision:

Implement file chunking to send large files in smaller chunks and consolidate them in the microservice.

Alternatives Considered:

1. **Direct Large File Upload:**
   Sending the entire large file as a single upload, which could lead to performance issues and timeouts.
2. **Third-Party File Transfer Services:**
   Utilizing third-party services designed for large file transfers, but this might introduce additional complexity and potential costs.

Rationale:

- **Efficiency with Chunking**: Sending files in smaller chunks reduces the risk of timeouts and improves the overall upload speed.
- **Scalability**: Chunking allows us to handle larger files without significant impact on system performance.
- **Asynchronous Uploads**: Handling file uploads asynchronously in the microservice ensures that the upload process doesn't block other operations, enhancing the responsiveness of our system.
- **Simplicity**: Implementing file chunking is straightforward and aligns well with our existing architecture without introducing unnecessary complexity.

Consequences:

- **Improved Performance**: File uploads are more efficient and less prone to timeouts, leading to a better user experience.
- **Scalability**: The system can handle larger files without significant changes to the architecture.
- **Asynchronous Processing**: Handling uploads asynchronously in the microservice ensures responsiveness and allows for better utilization of resources.
- **Code Complexity**: Implementing file chunking and consolidation requires careful handling to ensure data integrity and proper error handling.

Status:

Accepted

Date:

2024-04-10

Architecture Decision Record (ADR) - Database Schema Choices for Repo Management

Context:

Managing repositories involves tracking various details such as repository names, user ownership, file paths, and more. Searching within directories and files in Google Drive would have been slow and inefficient, especially as the number of repositories and files grows. **Therefore, a better database solution is needed to efficiently access and search repositories and their contents.**

Decision:

Use SQLite as the database for repository management.

Alternatives Considered:

1. **Srearching within Google Drive Repository:**
   Initially considered but deemed slow for searching within user directories and repositories.
2. **Traditional RDBMS (e.g., MySQL, PostgreSQL):**
   Common choices for database management but might be overkill for our current needs.
3. **NoSQL Databases (e.g., MongoDB, CouchDB):**
   Flexible and scalable but may not provide the indexing capabilities needed for efficient searching. Also, overkill.

Rationale:

- **Performance Concerns with Google Drive**: Google Drive would have been slow for searching through user directories and repositories, impacting user experience.

- **SQLite for Speed and Lightweight**: SQLite is lightweight, fast, and supports indexing, making it suitable for our needs to efficiently manage and search repositories.
- **Flexibility for Future Changes**: SQLite's lightweight nature allows us to easily switch to a more robust database solution in the future if needed, without major architectural changes.

Consequences:

- **Fast and Lightweight Access**: Using SQLite with indexing allows for quick searches and retrievals, improving the user experience.
- **Scalability**: SQLite may have limitations in handling extremely large datasets or concurrent writes, but it should suffice for our current needs.
- **Flexibility**: The choice of SQLite provides flexibility to migrate to a different database system in the future if our requirements change.
- **Maintenance**: SQLite is lightweight but still requires maintenance and backup procedures to ensure data integrity.

Status:

Accepted

Date:

2024-04-16

Task 3: Architectural Tactics and Patterns

Architectural Tactics

1.Availability

99% Uptime as Priority for Availability

For **availability**, ensuring a 99% uptime is our top priority, especially as end-users are working with large datasets. We are planning to achieve this by implementing **heartbeat monitoring** and **active redundancy**. Heartbeat monitoring regularly

checks the system's health, while active redundancy ensures backup systems are ready to take over if needed, guaranteeing a reliable and resilient application.

2. Security

Securing Model Hive with JWT and Google Drive

In Model Hive, **security** is paramount to safeguard user data and ensure secure access to the system. We've implemented **JSON Web Token (JWT)** for user authentication, reinforcing our commitment to data confidentiality and integrity. This approach not only verifies user identities but also authorizes access, providing an additional layer of security against unauthorized usage.

Furthermore, we've used **Google Drive** for data storage, leveraging its built-in security features. This decision demands robust handling of authentication and authorization processes to ensure that data remains confidential and maintains its integrity.

3. Performance

Boosting Performance through Concurrency and Caching

In Model Hive, we're boosting our **performance** by focusing on **concurrency** and **data caching**.

Concurrency ensures that the system can accommodate up to **5 concurrent users** simultaneously, providing a response within **0.5 seconds** of user interaction.

On the other hand, caching allows us to store and retrieve data more quickly, making the system **responsive** with changes appearing in under half a second. These strategies combined ensure a smooth user experience, even when dealing with large datasets.

4. Modifiability

Enhancing Modifiability with Microservices

In our Model Hive MVP, we leverage **microservices architecture** to promote **modifiability**. This architecture encourages **localized modifications** by breaking the system into independent services, each with a specific responsibility. This **semantic coherence** allows for isolated changes, reducing the complexity and cost of updates.

By assigning clear responsibilities to each microservice, we anticipate and limit future changes' scope. This design approach ensures each service is self-contained and focused, simplifying modifications without causing ripple effects across the system.

5.Usability

Design Time Tactic: Model-View-Controller (MVC)

Rationale:
To address Model Hive's usability goals, such as an Intuitive Interface and Feedback Mechanism, we plan to implement the MVC architectural pattern. This approach separates our application into three distinct components:

- **Model**: Manages data and business logic.
- **View**: Presents UI data to users and captures their input.
- **Controller**: Handles user input, updating the Model and View accordingly.

Benefits:

- For an **Intuitive Interface**, MVC simplifies navigation, reducing the learning curve for new users.
- To ensure an effective **Feedback Mechanism**, the Controller swiftly communicates user actions to update the View with progress indicators and notifications, keeping users informed in real-time.

Runtime Tactic: Maintain a Model of the Task and User

Maintain a Model of the Task:

- Provides contextual guidance by understanding the task at hand, offering relevant prompts and training suggestions to users.

Maintain a Model of the User:

- Enables personalized interactions by considering user preferences, such as scrolling speeds or preferred layouts, to adapt the interface accordingly.

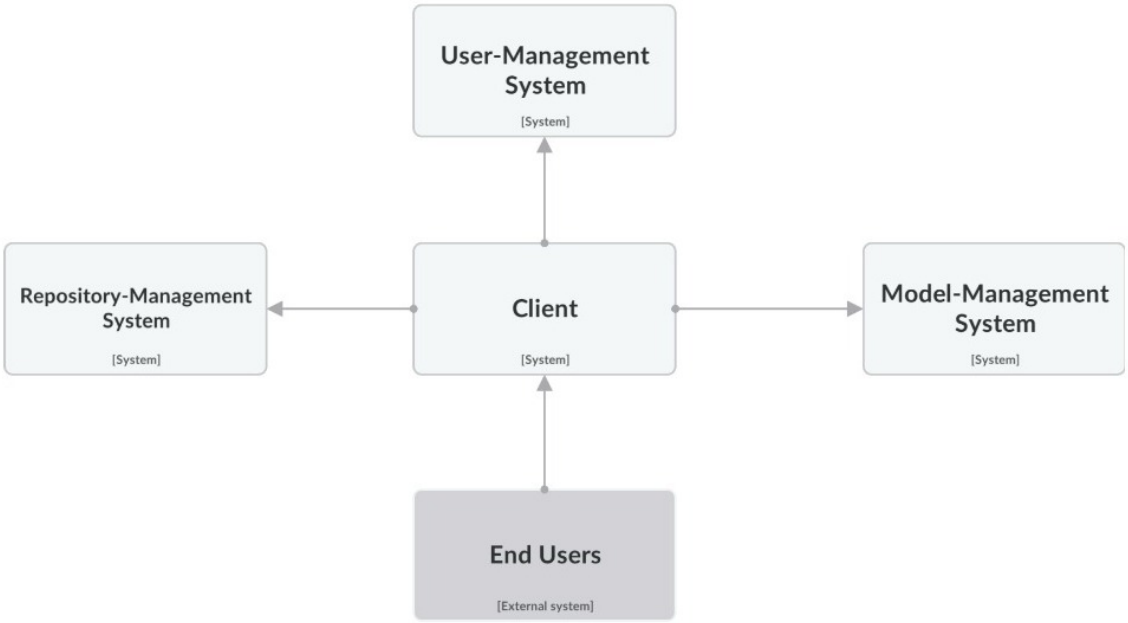
Architectural Design Patterns

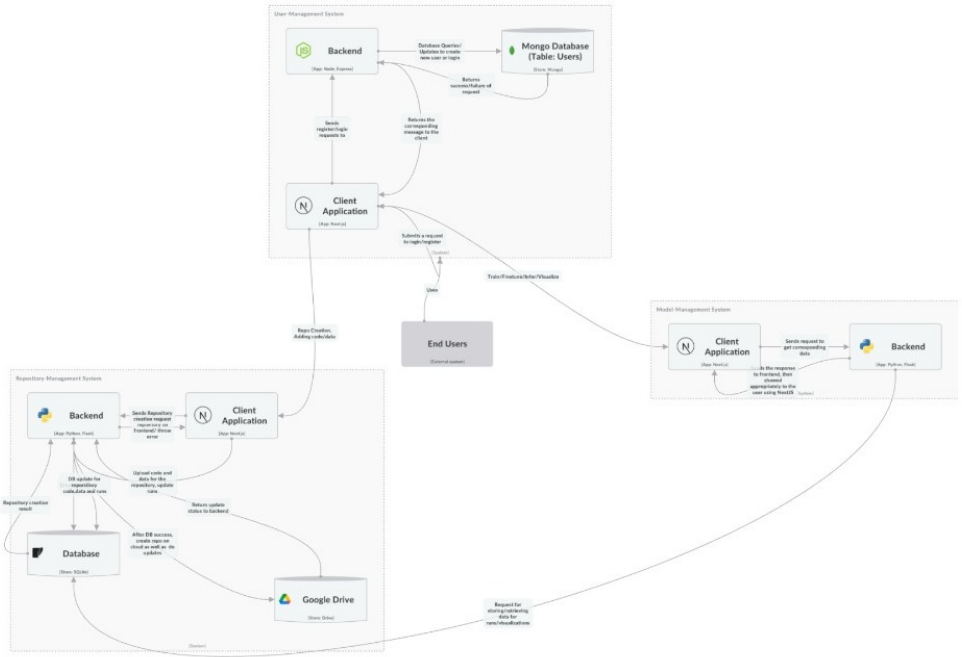
While designing our system, we considered several design patterns to improve modularity, maintainability, and scalability. However, for the initial MVP (Minimum Viable Product), we've decided to prioritize core functionalities to expedite the development process. The following design patterns can be beneficial for a full-scale model of our application:

- **Adapter Pattern:**
  - Facilitates seamless integration and interaction between different components of the system by implementing an adapter for visualizing data.
- **Chain of Responsibility (CoR) Pattern:**
  - Useful during the training process for effective handling and delegation of tasks, ensuring tasks are processed by the appropriate component in the chain.
- **Singleton Pattern:**
  - Efficiently manages the application context instance by ensuring there's only one instance throughout the application's lifecycle, providing centralized access to resources.

Implementing these design patterns in a full-scale model can enhance the system's flexibility, maintainability, and performance.

Architectural Diagrams


User-Management System
[System]
Repository-Management System
[System]
Client
[System]
Model-Management System
[System]
End Users
[External system]


Backend
Mongo Database (Table: Users)
Client Application
End Users
Client Application
Backend
Backend
Client Application
Database
Google Drive

Task 4: Prototype Implementation

Prototype Development

The implemented system reflects several key architectural decisions aimed at addressing specific challenges and requirements. One such decision involves the adoption of chunked file transfer for handling large files efficiently between the React frontend and Flask backend. This approach allows files to be broken down into smaller chunks, optimizing network usage and ensuring seamless communication while consolidating them within the microservices. By implementing chunked file transfer, the system enhances reliability, resilience, and performance, especially when dealing with large datasets and code files.

Another crucial design decision pertains to the file storage strategy, which combines Google Drive for large datasets and local storage for code and non-data files. Leveraging Google Drive's fast APIs and ample storage capacity, large datasets are stored externally as zip files, avoiding network transfer complexities. Meanwhile, code files and smaller non-data files are stored both locally and on Google Drive for accessibility and consistency. This hybrid approach simplifies file management, ensures efficient access, and minimizes costs, aligning with the system's objectives of simplicity, reliability, and cost-effectiveness.

Furthermore, the system incorporates a robust database schema for efficient management and access to user repositories, directories, and files. By implementing a relational database schema using SQL, the system enables fast access, efficient search capabilities, and organized linkage between users, repositories, directories, and files. This decision facilitates scalability, modularity, and maintainability while supporting various operations such as user-repository mapping, data-file mapping, and code-file mapping. Overall, these architectural decisions collectively contribute to the practical application and effectiveness of

the implemented prototype, demonstrating a thoughtful approach to system design and development.

The implementation is available here: https://github.com/Mitanshk01/SWE-Project-3/tree/main

Analysis

We have written the entire system again using monolithic architecture by simply combing all the services (save user auth which functions as a middleware) into one backend.

We consider files of the same and training parameters for comparing the two architectures. Please note that this is highly dependent on the internet speed.

Service Checked	Reponse Time Microservices (in seconds) Monolithic (in seconds)	Number of requests/second Microservices (in requests/second) Monolithic (in requests/second)
Authentication	0.003 0.003	262 289
Uploading Data	0.003 0.004	0.81 0.84
Training Loop	0.003 0.003	0.74 0.73

The authentication was tested using locust while the other two services were tested manually for 100 requests owing to the need for sending files which can prove challenging.

We note that training is done asynchronously to let the backend serve other requests as well. ***However***, we choose not to make the data upload step asynchronous as our system crashes owing to the high resources needed for this

(seen using `htop` before crash). A more powerful system can do this asynchronoulsy.

We see that the response times as well as throughput remain the same for both the microservices based architecture as well as the monolithic system. This can be attributed to the latency caused by the training as well as the data upload steps which offsets the much smaller advantage of lesser time taken by the monolithic system (since now there is no latency associated with querying the services from the backend).

We point the reader to the ADR on choosing the microservive architecture to emphasize on its advantages now that both the proposed architectures function in the same manner.
