**High level Architecture**

**P13: ContinuumAi**

| **Student ID** | **Name** |
| --- | --- |
| **26100059** | **Ali Faizan Chughtai** |
| **26100146** | **Muhammad Bazaf Shakeel** |
| **26100063** | **Umer Raja** |
| **26100038** | **Mustufa** |
| **26100029** | **Muhammad Nafees** |

**Table of Contents**

[1.](#_1i6zmopiey2b) Introduction 3

[2.](#_bco04nzd7a8w) Non-functional requirements/Quality attributes of the system 4

[3.](#_ucs6g87muvy0) Security Requirements 5

[4.](#_kn50711ps90a) Project Risk Management 6

[4.1 Potential Project Risks and Mitigation Strategies 6](#_714l6x9i6hoy)

[5.](#_yhnervqbaeiw) System Architecture 7

[5.1 Architecture Diagram 7](#_e35adw8jzrki)

[5.2 Architecture Description 8](#_crwruml6z8vh)

[5.3 Justification of the Architecture 9](#_kc94tgsn1g25)

[6.](#_1vs8fhxg2q40) Tools and Technologies 13

[7.](#_8yd9snd60fdd) Hardware Requirements 14

[8.](#_j0gmolj1zkmj) Development Environment Preparation 15

[9.](#_ym7tz4qp5pj2) Deployment Platform 15

[10.](#_mgdqc5wua5td) Use of Generative AI 16

[11.](#_igy3fblgybmk) Who Did What? 17

[12.](#_eie3fvpzl7e9) Review checklist 17

# Introduction

**Project Overview:** ContinuumAI is an Agentic AI system designed to function as a personal data scientist for non-technical users in business roles. Its primary objective is to enable decision-makers - such as managers in HR, Sales, Marketing, and Product teams - to access and act upon data insights through natural language prompts, without requiring coding or technical expertise.

The system bridges the gap between data science capabilities and business needs by automating the full analytics workflow: from data ingestion to descriptive, diagnostic, predictive, and prescriptive analysis. It provides outputs such as trends, visualizations (e.g., bar charts, line graphs), forecasts, performance metrics, and actionable recommendations entirely based on user queries in plain English.

For example, a regional sales manager might want to understand what factors influenced last quarter’s revenue dip or forecast next month's performance. Instead of relying on a data team, they can simply ask the system in natural language. ContinuumAI automatically identifies the relevant data, applies the right analytical methods, and delivers clear, actionable insights empowering the user to make informed decisions instantly.

While the long-term goal is to support **all major business domains**, the current version of ContinuumAI is focused specifically on the **Sales domain**, allowing us to build deep and meaningful capabilities before expanding further. (Hence, for this document as well we will be focussing on sales only)

As for the data required to support our sales-oriented features, we are currently in discussions with firms to obtain sales related datasets, once a functional MVP has been achieved. For initial development, we will explore open-source alternatives (e.g., Kaggle, Hugging Face) to develop an idea of standard dataset forms and generate synthetic data based on well-defined schemas informed by the open-set datasets as well as input from industry advisors, in light of the use cases detailed in the previous document.

# Non-functional requirements/Quality attributes of the system

| **Sr#** | **Requirements** |
| --- | --- |
| 1 | The system will process and display visualizations for 90% of natural language queries on datasets under 500 MB within 10 seconds. |
| 2 | A 100 MB CSV file will be ingested, processed, and made ready for analysis in under 60 seconds. |
| 3 | After a 15-minute tutorial, new users must achieve an 80% first-attempt success rate when generating a sales trend visualization without assistance. |
| 4 | The system must support 20 concurrent users with no more than a 20% degradation in average query response time compared to a single user. |
| 5 | Exporting any chart to a PNG file or data table to a CSV file must complete within 5 seconds. |

# Security Requirements

| **Sr#** | **Security Risks** | **Potential Losses** | **Controls** |
| --- | --- | --- | --- |
| 1 | Broken Access Control | Confidential data exposure, Data integrity compromise,  Legal and compliance issues | Session Isolation,  Access Control Validation,  Token Management |
| 2 | Cryptographic Failures | Confidential Data Exposure,  Data Integrity Compromise,  Regulatory Penalties | Encrypt Data at Rest and Transit,  Password Hashing,  Key Management |
| 3 | Injection | Confidential data exposure, Data integrity compromise,  System Compromise | Input Validation and Sanitisation,  Parameterized Queries for SQL |
| 4 | Unrestricted Resource Consumption | Denial of Service (DoS), Increased Operational Costs | Rate Limiting,  Resource Quotas (for network bandwidth etc.) |
| 5 | Server Side Request Forgery (SSRF) | Data Exposure,  Denial of Service (DoS) | Input Validation and Sanitization,  Allowlists for URL Origins |

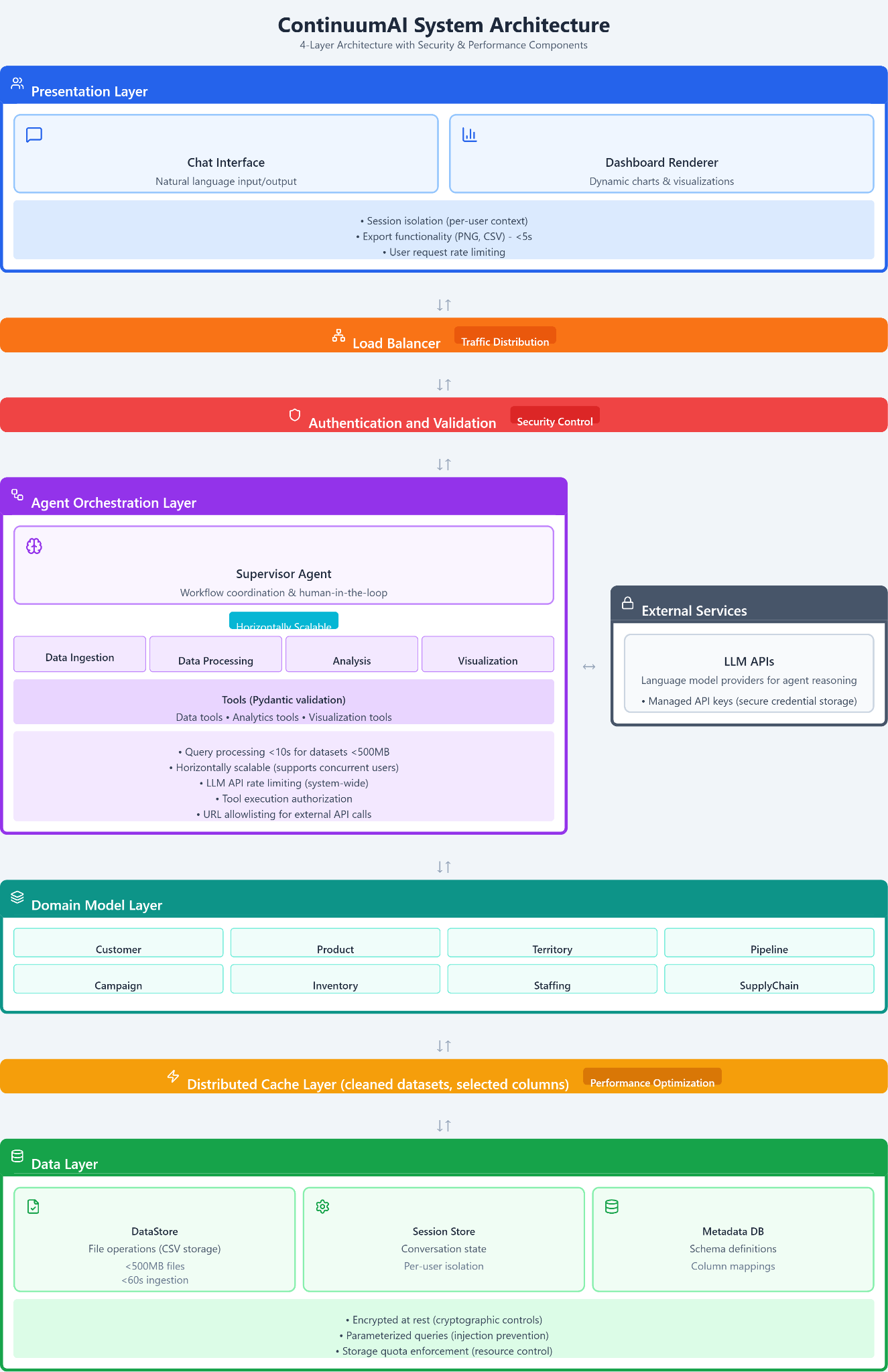
# Project Risk Management

## Potential Project Risks and Mitigation Strategies

| **Sr.** | **Risk Description** | **Mitigation Strategy** |
| --- | --- | --- |
|  | **Staff turnover**  Experienced staff leaves the project before it is finished. | If an experienced team member leaves, detailed documentation and regular knowledge-sharing sessions (which we will maintain on a weekly basis) will ensure continuity. These practices will prevent dependency on any single person, allowing new or existing members to seamlessly assume responsibilities using the maintained records and notes. |
|  | **Requirements change**  Changes in requirements that require major design rework are proposed. | If major requirement changes are proposed, they will go through a formal change management process to assess their impact and gain team approval. Once evaluated, the changes will be integrated incrementally through the accepted plan. This won't be a major risk since we will be using an agile workflow, where new requirements can also be accommodated in later stages by creating dedicated sprint plans for their implementation. |
|  | **Underestimation**  The size of the system is underestimated. | If the system turns out larger than estimated, the team will refine effort, budget and timeline estimates iteratively and consult experienced advisors to reassess scope and provide guidance on necessary adjustments. Those adjustments will then be incorporated into the plan to make up for this risk. |
|  | **Technology change**  The underlying technology on which the system is built is superseded by a new technology. | If the chosen technology is superseded, the team will actively monitor emerging alternatives and, if a superior option is identified, trigger a contingency plan to evaluate the costs, benefits, and timing of migration to ensure minimal disruption. |
|  | **Code generation**  The code generated by generative AI is inefficient. | If AI-generated code proves inefficient, developers will review and refactor critical sections, and where inefficiencies are systemic, replace underperforming components with human-developed modules to ensure performance standards are met. |
|  | **Data**  Required data is not available. The required data may be for training of ML Model or for some other purpose. | If required data is not available, synthetic datasets will be generated based on our universal schema provided in the previous document. It will be ensured that the data generated via the universal schema mimics real world scenarios. |
|  | **Stakeholder management**  Customers fail to understand the impact of requirements change. | If customers fail to understand the impact of requirements change, the team will clearly communicate cost, time, and design implications and conduct workshops or demos to illustrate the real-world effects before proceeding. |
|  | **Off-the-shelf components and libraries**  Software components/libraries that were planned to be used do not contain desired features or contain defects, i.e., they cannot be used as planned. | If planned components or libraries lack desired features or contain defects, equivalent alternatives will be evaluated, and if none are viable, stable fallback versions will be used while a longer-term solution is developed. |

# System Architecture

## Architecture Diagram

All inter-layer communication is bidirectional, enabling both request flows (downward) and response flows (upward).

## Architecture Description

### **Subsystem Descriptions:**

#### **1. Presentation Layer**

The Presentation Layer serves as the user-facing interface of ContinuumAI. It consists of two main components:

* **Chat Interface**: Accepts natural language queries from users and displays conversational responses. Handles clarification requests when the system needs additional information from users.
* **Dashboard Renderer**: Dynamically generates and displays data visualizations (charts, graphs) based on analysis results. Supports export functionality for charts (PNG) and data tables (CSV).

This layer implements session isolation to maintain separate contexts for different users and enforces user-level request rate limiting to prevent system abuse.

#### **2. Authentication and Validation (Security Middleware)**

Serves as the system’s gatekeeper between the Presentation and Orchestration layers. It validates user authentication tokens, enforces authorization rules, and performs strict input sanitization before requests are processed by downstream agents.

#### **3. Agent Orchestration Layer**

The core of ContinuumAI's intelligent workflow management. This layer contains:

* **Supervisor Agent**: Coordinates the overall workflow, routes queries to appropriate specialized agents, and manages human-in-the-loop interactions for clarifications.
* **Specialized Agents**: Some of the agents which will handle distinct responsibilities:
  + **Data Ingestion Agent**: Processes uploaded CSV files and validates their structure.
  + **Data Processing Agent**: Cleans and transforms data (handles missing values, outliers, data type conversions).
  + **Analysis Agent**: Performs descriptive, diagnostic, and predictive analytics including forecasting.
  + **Visualization Agent**: Selects appropriate chart types and generates visualizations.

Each agent utilizes **Tools** (Python functions with Pydantic validation) that perform specific operations like statistical calculations, data transformations, and chart generation. The layer enforces tool execution authorization and implements LLM API rate limiting to control costs and prevent hitting external service limits.

This layer is designed to be horizontally scalable, meaning multiple agent instances can be deployed to handle concurrent user requests.

**4. Domain Model Layer**

The Domain Model Layer represents the business entities that form the foundation of ContinuumAI’s analytical operations. It acts as a link between the raw data stored in the lower layers and the analytical agents working in the Orchestration Layer, ensuring that both human-readable queries and technical dataset fields stay aligned.

This layer is organized into several domain modules such as **Customer**, **Product**, **Territory**, **Pipeline**, **Campaign**, **Inventory**, **Staffing**, and **SupplyChain**. Each module defines key attributes, relationships, and aggregation rules that the agents use during analysis and visualization.

Its main role is to standardize how the system understands and interacts with data. It connects natural language terms (like “sales by region” or “top-performing products”) to the correct schema fields stored in the Metadata DB, keeping queries accurate and consistent. It also validates data structures before they are passed to agents, reducing errors caused by schema mismatches.

The Domain Model Layer works closely with the Distributed Cache Layer to store only cleaned and frequently used columns, improving performance and speeding up repeated analyses. Its modular design also makes it easy to introduce new business areas in the future without affecting the existing setup.

Overall, this layer helps maintain consistency, improves query reliability, and provides a clear logical structure for data across all parts of the ContinuumAI system.

#### **5. Cache Layer (Performance Middleware)**

Positioned between the Agent Orchestration and Data layers, this component stores cleaned datasets and selected columns in memory. When users query the same dataset multiple times, the system retrieves pre-processed data from the cache rather than re-reading and re-cleaning the original CSV files. This significantly reduces response times for subsequent queries on the same data.

#### **6. Data Layer**

The Data Layer manages all persistent storage and consists of three components:

* **DataStore**: Handles file operations for CSV storage and retrieval. Supports datasets up to 500MB with ingestion times under 60 seconds.
* **Session Store**: Maintains conversation state and context for each user session, enabling the user to return to older conversations where the system remembers previous queries and results.
* **Metadata DB**: Stores dataset schemas (column names, data types), column mappings between business terms and technical fields, and data lineage information.

The Data Layer implements encryption at rest for all stored files, uses parameterized queries to prevent SQL injection, and enforces storage quotas to control resource consumption

#### **7. External Services**

External language model providers (OpenAI, Anthropic) accessed by agents for natural language reasoning. This represents the integration boundary where the system connects to external APIs. API credentials are managed through secure channels (environment variables for MVP, with architecture supporting future integration with secret management services). URL allowlisting at this boundary prevents unauthorized external requests.

### **Subsystem Interactions:**

1. **User Query Flow**: User submits a natural language query through the Chat Interface → Authentication and Validation middleware sanitizes the input → Supervisor Agent receives the validated query.
2. **Query Routing**: Supervisor Agent analyzes the query intent → If clarification is needed, sends a question back to Chat Interface → User responds → Supervisor routes to appropriate specialized agent(s).
3. **Data Processing**: If a new file is uploaded, Data Ingestion Agent processes it → Sends to Data Processing Agent for cleaning → Cleaned data is stored in Cache Layer and DataStore → Metadata is extracted and saved in Metadata DB.
4. **Analysis Execution**: Analysis Agent retrieves data from Cache Layer (if available) or DataStore → Performs statistical computations or forecasting → Returns results to Supervisor Agent.
5. **Visualization Generation**: Visualization Agent receives analysis results → Generates appropriate chart → Sends to Dashboard Renderer → User views the visualization.
6. **State Management**: During an active session, LangGraph natively supports saving memory and context. However, if the user ends the session and returns later, the Session Store preserves the conversation history, allowing the system to recall previous queries and build upon earlier results.
7. **External Communication**: Agents communicate with LLM APIs for reasoning tasks → API calls pass through rate limiting controls → Results are used for decision-making within the agent workflows.

## Justification of the Architecture

* **Pros and cons of the architecture**

| **Pros** | **Cons** |
| --- | --- |
| **Modularity**: Clear separation of concerns across layers enables independent development and testing of each component. Changes to UI don't affect agent logic, and data storage modifications don't impact analysis algorithms. | **Latency Overhead**: Multiple layers and agent coordination introduce sequential processing steps, which can increase response time compared to a monolithic single-step system. |
| **Scalability**: Agent Orchestration Layer can be horizontally scaled by deploying multiple agent instances to handle concurrent users. Each layer can be scaled independently based on demand. | **Single Point of Failure**: Cache Layer failure would degrade performance significantly. security middleware failure would block all user requests. |
| **Security by Design**: Dedicated Authentication and Validation middleware and layer-specific security controls (encryption, parameterized queries) provide defense-in-depth protection. | **Memory Constraints**: Caching cleaned datasets in memory can consume significant RAM, especially with multiple concurrent users working on large (500MB) files. Tracking the cycle of cached files is also complex. |
| **Performance Optimization**: Cache Layer provides significant speed improvements for repeated queries on the same dataset, directly addressing the response time requirement | **Agent Coordination Cost**: Each agent interaction with LLM APIs incurs cost and latency. Complex queries requiring multiple agents become expensive. Horizontally scaling agents also may add up to more costs. |
| **Maintainability**: Layered architecture with clear boundaries makes it easier to locate and fix bugs. Each agent has a specific responsibility, simplifying code maintenance. | **Limited Fault Isolation**: Since agents share the same orchestration layer, a crash in one agent's execution could potentially affect the stability of other agents. |
| **Extensibility**: New analysis capabilities can be added by creating additional specialized agents without modifying existing components. New data sources can be integrated by extending DataStore abstraction. |  |

* **Implementation of non-functional requirements in system architecture**

| **Requirement** | The system will process and display visualizations for 90% of natural language queries on datasets under 500 MB within 10 seconds. |
| --- | --- |
| **Implementation in the architecture** | The architecture achieves this performance requirement through multiple coordinated strategies:  **Cache Layer**: When a user queries a dataset that has been previously processed, the Cache Layer stores cleaned data frames and selected columns in memory, eliminating 60-80% of processing time for repeated queries.  **Agent Orchestration Efficiency**: The Supervisor Agent routes queries directly to the appropriate specialized agent based on query intent, avoiding unnecessary sequential processing.  **Asynchronous Tool Execution**: Tools within the Agent Orchestration Layer can execute asynchronously given that there are no compute restraints therefore reducing total processing time.  **Metadata DB**: By storing dataset schemas separately, the system can quickly understand data structure without opening CSV files.  The layered architecture enables these optimizations to work together, ensuring efficient data processing.  The layered architecture enables these optimizations to work together: cached data bypasses Data Layer access entirely, metadata queries avoid file I/O, and specialized agents focus processing power only on necessary tasks. |

| **Requirement** | A 100 MB CSV file will be ingested, processed, and made ready for analysis in under 60 seconds. |
| --- | --- |
| **Implementation in the architecture** | The layered architecture enables efficient ingestion through clear separation of responsibilities:  **Cache Layer Integration**: The architecture positions the Cache Layer between Agent Orchestration and Data layers, allowing cleaned datasets to be immediately cached during ingestion. This eliminates the need to re-process data for the first query, contributing to the overall 60-second preparation time.  **Metadata Separation**: By maintaining a separate Metadata DB, the architecture allows schema extraction to occur independently of data storage operations. The system can query dataset structure instantly without accessing the full CSV file, speeding up query planning after ingestion.  **Modular Data Flow**: The clear flow from Data Ingestion Agent → Data Processing Agent → DataStore/Cache enables optimization at each stage. The architecture supports parallel execution where beneficial (schema extraction while data is being cached) without forcing sequential bottlenecks.  The layered design ensures ingestion components can be independently optimized and scaled to meet the 60-second requirement as file sizes approach 100MB. |

# 

| **Requirement** | After a 15-minute tutorial, new users must achieve an 80% first-attempt success rate when generating a sales trend visualization without assistance |
| --- | --- |
| **Implementation in the architecture** | The architecture prioritizes usability through its conversational design and intelligent assistance:  **Natural Language Chat Interface**: Users express requests in plain English, which the system interprets through LLM-powered agents.  **Human-in-the-Loop Clarifications**: The Supervisor Agent asks clarifying questions when ambiguity is detected in a query.  **Supervisor Agent Intelligence**: The Supervisor Agent analyzes query intent and provides helpful feedback.  **Immediate Visual Feedback**: The Dashboard Renderer displays visualizations instantly upon generation.  The architecture's emphasis on conversational interaction (Presentation ↔ Agent communication) and intelligent query understanding (Supervisor + LLM APIs) directly supports the 80% first-attempt success rate by guiding users rather than requiring them to learn complex commands. |

# 

# 

| **Requirement** | The system must support 20 concurrent users with no more than a 20% degradation in average query response time compared to a single user. |
| --- | --- |
| **Implementation in the architecture** | The architecture is designed to efficiently handle multiple independent users accessing the service simultaneously, while also remaining flexible enough to support shared-data scenarios in future deployments.  **Session Isolation**:  Each user operates within an isolated session context. The Session Store maintains distinct conversation histories and memory states per user, ensuring that one user’s processing, context, or data does not interfere with another’s workflow.  **User-Specific Data Context**:  In the current design, each user primarily interacts with their own uploaded data (e.g., individual CSV files). This separation inherently prevents data contention and simplifies performance scaling, allowing the system to maintain consistent query response times across users.  **Future Shared-Data Support**:  The layered architecture with multiple concern separated components allows easy adaptation if the product evolves to enable multiple users accessing shared datasets. Components within the data layer can support this in the following ways:  *Metadata DB*: Can be designed as a shared component that can manage schema and statistics across multiple users or datasets.  *Session Store*: Can be Isolated per user to maintain personalized context even when operating on shared data.  *Cache Layer*: Can be configured as shared (for common datasets) or user-specific (for isolated caching).  *DataStore*: Can store user-specific files today, but can be configured to support partitioning or shared access patterns if required later.  **LLM API and Request Rate Limiting**:  The Presentation Layer enforces per-user rate limits, while system-level LLM API throttling ensures that total throughput stays within provider limits—preserving stable response times as user load increases.  **Horizontally Scalable Architecture (Future-Ready)**:  The Agent Orchestration Layer is built to allow horizontal scaling in future deployments. Additional agent instances can be introduced behind a load balancer to distribute concurrent requests without requiring architectural redesign.  The layered architecture’s separation of concerns—both across layers and within each layer’s components—ensures that scalability strategies can be applied without requiring architectural changes. |

# 

# 

| **Requirement** | Exporting any chart to a PNG file or data table to a CSV file must complete within 5 seconds. |
| --- | --- |
| **Implementation in the architecture** | The system achieves fast export operations by leveraging cached data, pre-generated visualization objects and lightweight serialization processes:  **In-Memory Visualization Objects:** When the Visualization Agent renders a chart for the Dashboard Renderer, it generates and retains the chart object in memory within the user’s session. Exporting to PNG simply serializes this pre-existing object—no regeneration or recomputation is required.  **Cached Data Reuse:** For data table exports, the Cache Layer retains the query results from prior analysis operations. When the user requests “Export to CSV,” the system writes the cached data directly to file format, avoiding a full re-query from the Data Layer.  **Session-Aware Presentation Layer:** The Presentation Layer (Dashboard Renderer) maintains references to each chart and table created in the current session. Export functions directly access these in-memory references, eliminating additional inter-layer communication delays.  **Future Scalability & Storage Management:** Chart objects and data remain in memory only for the duration of an active session. For long-term retention or collaborative scenarios, only lightweight metadata (e.g., chart specifications, dataset IDs) can be persisted, allowing charts to be reconstructed efficiently without high memory cost.  By caching data and visualizations in-session and performing only serialization during export, the system easily meets the 5-second export-time requirement, while maintaining flexibility for future scaling and shared-data scenarios.  \*Serialization means converting an in-memory object into a format that can be stored or transferred. |

* **Implementation of security requirements in system architecture**

| **Security Risk** | **Broken Access Control**  Typically leads to unauthorized information disclosure, modification, or destruction of all data or performing a business function outside the user's limits. |
| --- | --- |
| **Implementation in the architecture** | The architecture implements defense-in-depth access control across multiple layers:  **Session Isolation (Presentation Layer)**: Each user session is completely isolated through unique session identifiers. When a user logs into the system, the Session Store creates a separate context containing only their uploaded files and query history. User A cannot access User B's datasets or conversation history, even if both are using the system simultaneously.  **Security Middleware (Authentication and Validation):** This middleware verifies user authentication and authorization before any request reaches the Agent Orchestration Layer. Unauthorized requests are blocked at this stage and never passed to the agent system.  **Per-User Data Isolation (Data Layer)**: The DataStore organizes files by user session, maintaining strict boundaries between different users' data. Session Store enforces that each user can only query their own conversation context and cached results. The Metadata DB tags all schema entries with user identifiers, preventing cross-user metadata queries. |

| **Security Risk** | **Cryptographic Failures**  Such a failure can expose confidential data through weak encryption, improper key management, or transmitting sensitive information in cleartext. This can lead to data breaches and regulatory compliance violations |
| --- | --- |
| **Implementation in the Architecture** | The architecture addresses cryptographic requirements through systematic encryption at rest and in transit:  **Data Encryption at Rest (Data Layer)**: All CSV files stored in the DataStore are encrypted using file system-level encryption. Sensitive datasets containing sales figures, customer information, or business metrics are never stored in plaintext. The DataStore component handles encryption/decryption transparently, ensuring agents work with decrypted data in memory but all persistent storage remains encrypted.  **Secure Credential Management (External Services)**: API credentials are managed at the External Services integration boundary. The Agent Orchestration Layer retrieves keys at runtime through secure channels, and keys are never logged or exposed in error messages. The architecture supports credential management ranging from environment variables (MVP) to dedicated secret management systems (production) without requiring changes to agent code.  **Encrypted Communication (Cross-Layer)**: When data moves between layers (e.g., Presentation ↔ Agent Orchestration ↔ Data), communication occurs through encrypted channels. For MVP running locally, this is enforced through localhost protections; when deployed to production, HTTPS will secure all inter-layer communication. |

| **Security Risk** | **Injection**  Injection attacks occur when untrusted data is sent to an interpreter as part of a command or query. SQL injection, CSV injection, or malicious file uploads can compromise data integrity, expose confidential information, or allow system takeover. |
| --- | --- |
| **Implementation in the Architecture** | The architecture implements multi-layered injection prevention:  **Authentication and Validation (Security Middleware)**: This dedicated component between Presentation and Agent Orchestration layers serves as the primary defense against injection attacks. Every user query, file upload, and parameter is validated before reaching the agent system. The middleware checks for SQL injection patterns in natural language queries,  malicious formulas in CSV files (e.g., cells starting with =, +, -, @) and many other common threats.  **Pydantic Schema Validation (Agent Orchestration Layer)**: All tools used by agents enforce strict input schemas using Pydantic validators. When a tool expects a column name, it validates that the input is a valid string matching dataset columns rather than arbitrary code. Type checking prevents injection through type confusion attacks**.**  **Parameterized Queries (Data Layer)**: The Metadata DB uses parameterized SQL queries exclusively, never constructing queries through string concatenation. This architectural constraint prevents SQL injection even if malicious input bypasses earlier validation layers. The reasoning behind this double validation of input assumes any subsystem can be compromised. For example, we are validating user input and sanitizing it but what we can not ultimately stop is the input generated via LLM agents so they are also assumed to be a threat at layers such as Data Layer where integrity of data can not be compromised. |

| **Security Risk** | **Unrestricted Resource Consumption**  Unrestricted resource consumption allows attackers to overwhelm system resources through excessive requests, large file uploads, or computationally expensive queries, leading to Denial of Service (DoS) and increased operational costs. |
| --- | --- |
| **Implementation in the Architecture** | The architecture implements resource controls at multiple layers to prevent resource exhaustion:  **User Request Rate Limiting (Presentation Layer)**: Each user is limited to a maximum number of requests per time window (e.g., 10 queries per minute). This prevents a single user—whether malicious or accidentally creating an automation script—from submitting thousands of queries that would consume system resources and degrade performance for other users. The Presentation Layer enforces these limits before queries reach the Agent Orchestration Layer.  **Storage Quota Enforcement (Data Layer)**: Each user session has a cumulative storage quota (e.g., 2GB total across all uploaded files). The DataStore tracks storage consumption per user and rejects new uploads once the quota is reached. This prevents indefinite storage growth that could fill disk space and crash the system.  **LLM API Rate Limiting (Agent Orchestration Layer)**: System-wide rate limiting controls the total number of LLM API calls across all users. This prevents collective resource consumption from exceeding provider limits (which would result in throttling or additional costs). The architecture allows for request management strategies to prevent rate limit violations, ensuring the system operates within budget constraints. |

| **Security Risk** | **Server Side Request Forgery (SSRF)**  SSRF attacks occur when an application fetches remote resources based on user-supplied URLs without proper validation. Attackers can exploit this to access internal services, scan internal networks, or exfiltrate data by forcing the server to make requests to attacker-controlled URLs. |
| --- | --- |
| **Implementation in the Architecture** | The architecture prevents SSRF attacks through layered controls**:**  **Input Validation for URLs (Security Middleware)**: The Input Validation middleware explicitly checks all user inputs for URL patterns and rejects them. The system only accepts natural language queries, file names, and dataset identifiers—never raw URLs. This prevents users from attempting to direct the system to arbitrary external resources.  **URL Allowlisting (Agent Orchestration Layer)**: As a secondary defense, the Agent Orchestration Layer maintains a hardcoded allowlist of permitted external endpoints (api.openai.com, api.anthropic.com). Any attempt to make HTTP requests to URLs not on this allowlist is automatically blocked, ensuring even internal code cannot access unauthorized external services. |

# 

# Tools and Technologies

### **Frontend:**

* **React 18 + TypeScript** – Production-grade web interface framework.
* **Next.js 14** – Server-side rendering and optimized routing for the frontend.
* **Tailwind CSS 3.4** – Utility-first CSS framework for rapid, responsive UI development.
* **Streamlit 1.37** – Local development interface for testing agent workflows and prototypes.

### **Backend**

* **Python 3.11** – Core language for all backend and AI components.
* **Framework:** *FastAPI*
* **Pydantic 2.8** - Data Validation and schema enforcement across backend APIs and agent tool inputs.

### **Agentic AI & Orchestration**

* **LangGraph 0.1** – Multi-agent orchestration and workflow routing.
* **LangChain 0.2** – LLM-based workflow and context management.
* **CrewAI 0.1** – Team-based coordination and role assignment among agents.
* **LLMs:** GPT-5, Gemini Flash, Claude – Used for reasoning, natural language understanding, and code generation within agents.

### **Machine Learning & Analytics Stack**

* **scikit-learn 1.5** – Predictive modeling and AutoML workflows.
* **PyTorch 2.2** – Deep learning, fine-tuning, and embedding generation.
* **pandas 2.2 / NumPy 1.26** – Data ingestion, transformation, and analysis.
* **matplotlib 3.9 / plotly 5.23** – Visualization generation for insights and dashboards.
* **SHAP 0.44 / LIME 0.2** – Model explainability and interpretability.

### **Database & Storage**

* **PostgreSQL 15** – Relational database for structured data and logs.

### **Development & Collaboration**

* **VS Code 1.93** – Primary IDE for development.
* **Jupyter Notebook (2024-09)** – For experimentation, data analysis, and agent behavior testing.
* **GitHub** – Version control and repository management.
* **Jira** – Project tracking and sprint planning.

### **DevOps & Deployment**

* **Docker 25** – Containerization of backend and agent services.
* **Vercel** – Deployment platform for hosting the production frontend.

**Monitoring of API costs of LLMS:**

* **MLflow**

For now, our development will primarily use Python. Initially, even the frontend will be built with Streamlit for rapid prototyping. The following Python libraries and frameworks are required for local development at this stage, we have ensured the versions are not in conflict. These may be updated as the project evolves based on new requirements or stage of dev.

* langchain==0.2.0
* langgraph==0.1.0
* crewai==0.1.0
* scikit-learn==1.5.0
* torch==2.2.0
* shap==0.44.0
* lime==0.2.0.1
* pandas==2.2.0
* numpy==1.26.0
* matplotlib==3.9.0
* plotly==5.23.0
* fastapi==0.115.0
* pydantic==2.8.0
* uvicorn==0.30.0
* psycopg2-binary==2.9.9
* docker==7.1.0
* jupyterlab==4.2.0
* streamlit==1.37.0
* langchain-openai==0.1.7
* langchain-community==0.2.0
* crewai-tools==0.1.2

# Hardware Requirements

<List down the hardware requirements. This should include requirements for both development machines and deployment servers>

### **1. Development Machines**

* **Processor:**
  + Minimum: Intel Core i5 (10th Gen) or AMD Ryzen 5
  + Recommended: Intel Core i7 (12th Gen) or AMD Ryzen 7
* **Memory (RAM):**
  + Minimum: 8 GB
  + Recommended: 16 GB
* **Storage:**
  + Minimum: 256 GB SSD
  + Recommended: 512 GB SSD or higher
* **Operating System:** Windows 11 / Ubuntu 22.04 LTS / macOS 13+
* **Network:** Minimum 20 Mbps broadband; Recommended 50 Mbps or higher

### 

### **2. Deployment / Production Servers**

* **Hosting Platform:** Vercel (serverless environment)
* **Processor:** Equivalent to 2–4 vCPUs (auto-scaled by platform)
* **Memory (RAM):** Minimum 2 GB; Recommended 4–8 GB
* **Storage:** Managed cloud storage (Vercel default) or external S3/Vercel KV for logs and exports
* **Database:** PostgreSQL 15 (cloud-hosted via Supabase / Neon / Railway / AWS RDS)
* **GPU:** Not required (relies on pretrained LLM APIs such as GPT-5, Gemini, and Claude)
* **Network Bandwidth:** Minimum 50 Mbps; Recommended 100 Mbps or higher
* **Operating System (runtime):** Ubuntu 22.04 LTS (containerized via Docker 25)

Since we’re focusing only on the **Sales domain** now, GPU acceleration or large-scale compute clusters are **not required**. Hardware and cloud resources can later be scaled when additional domains (e.g., Marketing, ERP, HR) are integrated.

# Development Environment Preparation

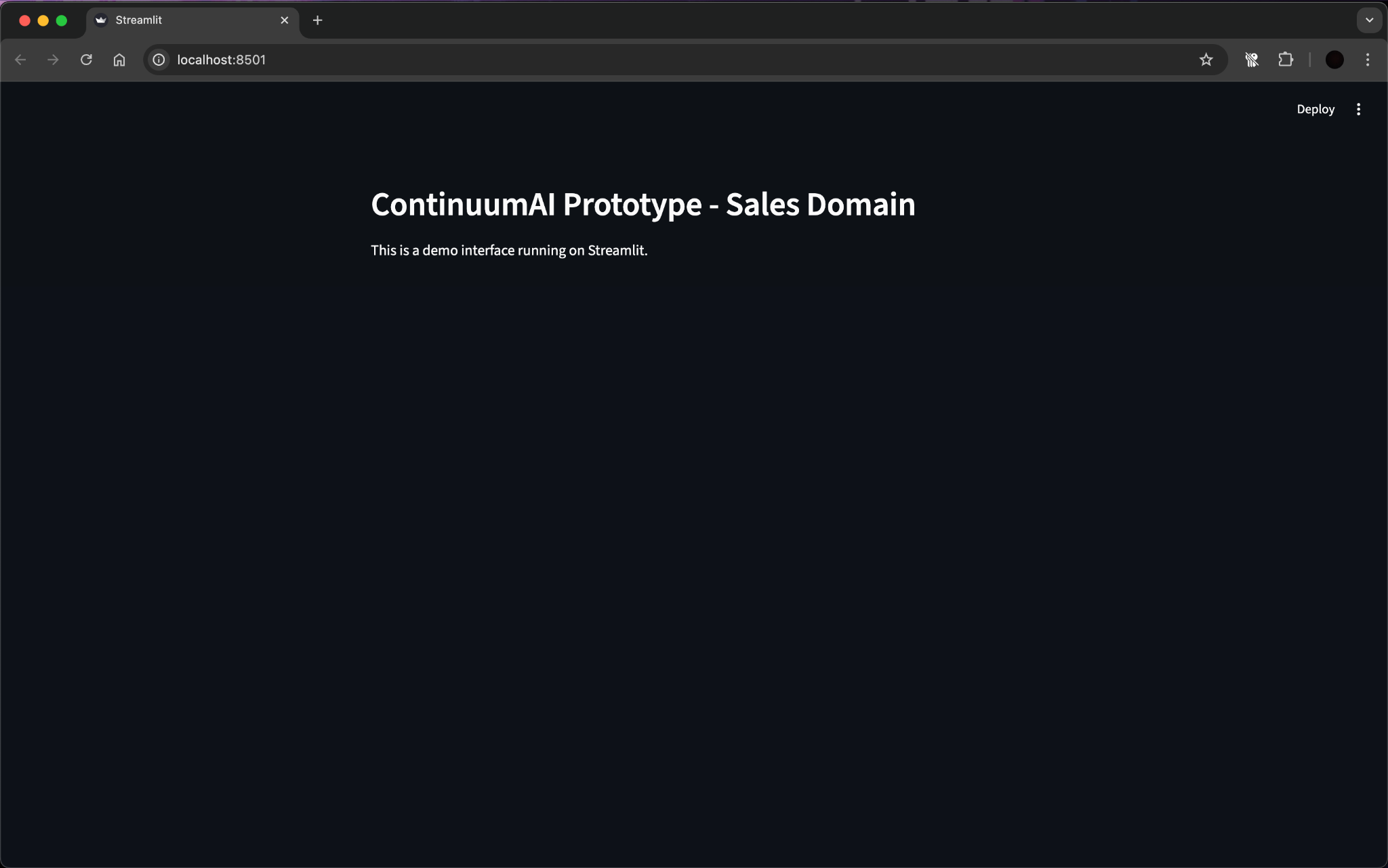
<(1) Setup the development environment on your machines and mention here that you have actually setup the environment. Include three snapshots of the tool(s) that you are going to use for development. These snapshots must be taken from the tool(s) while they are running on your development machines.>

The development environment for **ContinuumAI** has been successfully configured on local machines to enable smooth development, testing, and deployment. All essential tools and frameworks have been installed and verified to ensure compatibility between the system’s frontend, backend, and AI components.

While several tools have been set up as part of the complete development environment, the following three have been selected for illustration, as they represent key components of the setup:

* **Streamlit 1.37** for rapid user interface prototyping.
* **Jupyter Notebook (2024-09)** for data analysis, visualization, and model experimentation.
* **FastAPI (0.115)** for developing and testing backend APIs.

Below are three snapshots of these tools running locally on the development machines:



**Figure 1:** Streamlit interface running locally on http://localhost:8501

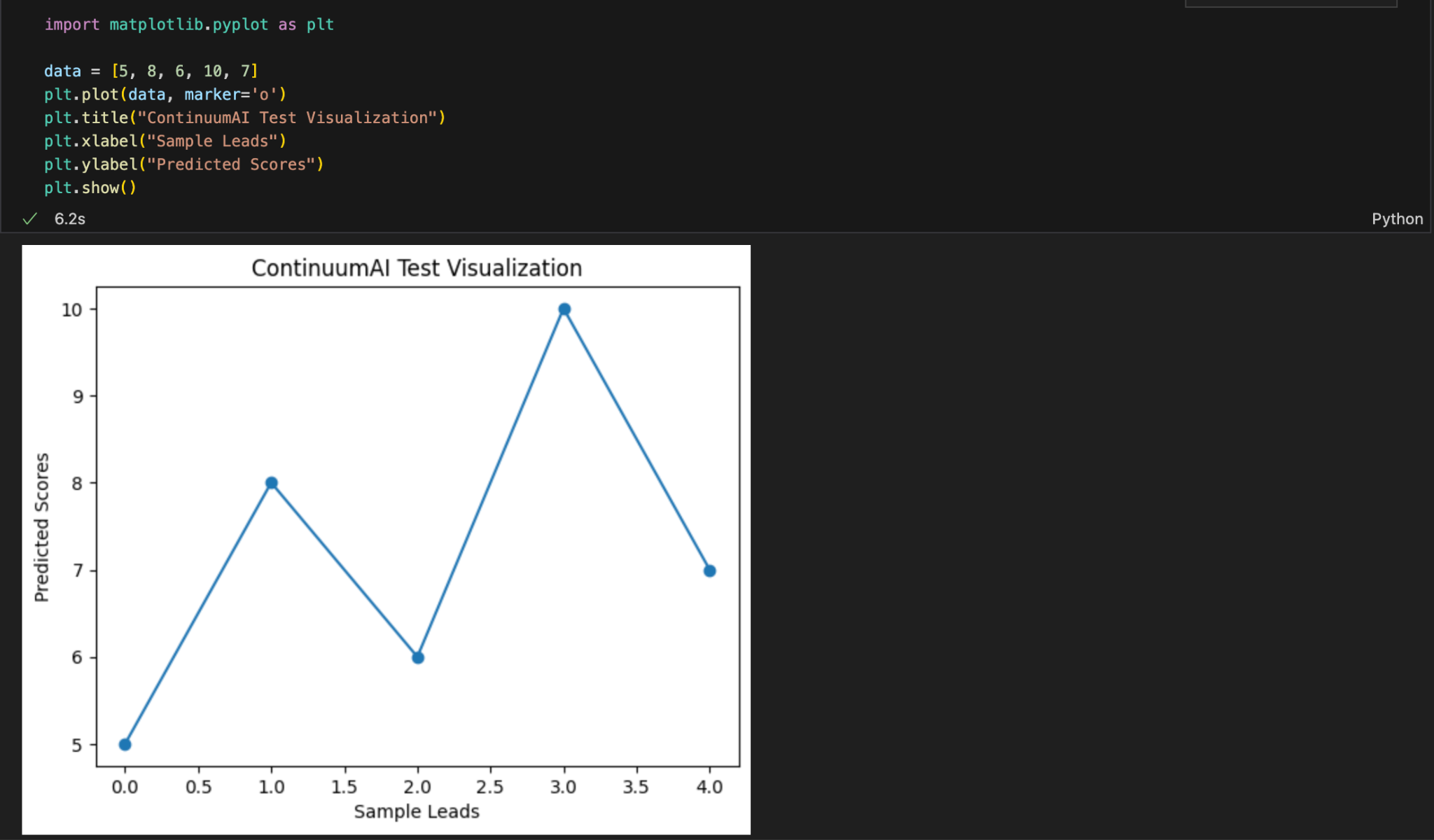
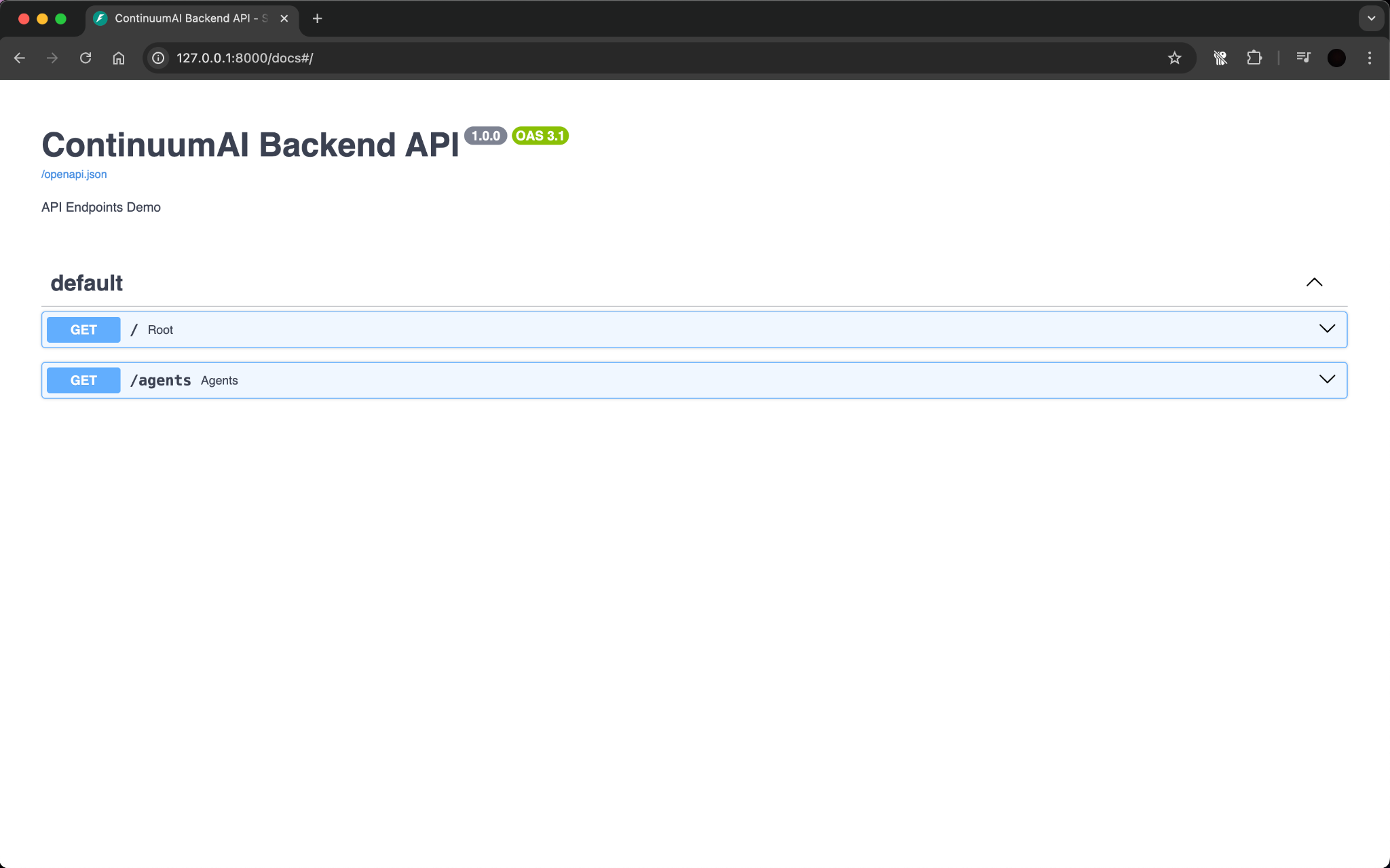


Figure 2: Jupyter Notebook environment running a sample Python visualization on dummy data for ContinuumAI.



**Figure 3:** FastAPI Swagger UI displaying available API endpoints (/ and /agents) running locally on http://127.0.0.1:8000/docs

All tools have been verified to run successfully and integrate seamlessly within the local environment. This configuration ensures that both frontend and backend components can be developed, tested, and iterated efficiently, providing a stable foundation for subsequent system implementation and evaluation phases.

.

# Deployment Platform

<Find a free hosting service where you can deploy your system for anyone to access online. This service will be used for deployment of prototype, sprints and final system.>

The system will be deployed on **Vercel**, a free cloud hosting platform designed for modern web applications.

Vercel provides seamless integration with **Next.js** and **React**, supports **Python FastAPI** through serverless functions, and offers automatic continuous deployment from GitHub repositories. It provides a **publicly accessible URL**, making it ideal for hosting the prototype, sprint builds, and the final system of **ContinuumAI**.

# Use of Generative AI

<Mention here how generative AI was used in preparation of this document.>

* Used generative AI to search for relevant tools with minimum run cost.Also to get some knowledge of the hardware compatible for our system.

# Who Did What?

| **Name of the Team Member** | **Tasks done** |
| --- | --- |
| Ali Faizan | Sections 1, 4 and 6 |
| Muhammad Bazaf Shakeel | Sections 7, 8 and 9 |
| Umer Raja | Sections 1, 2 and 5 |
| Mustufa | Section 5.1,10 |
| Muhammad Nafees | Section 5, 3 |

# Review checklist

| **Section** **Title** | **Reviewer Name(s)** |
| --- | --- |
| Section 7,6,8 | Mustufa |
| Sections 1, 4 | Umer Raja |
| Section 9 | Muhammad Nafees |
| Section 5 | Ali Faizan |
| Section 2 | Muhammad Bazaf Shakeel |