**High level Architecture**

**Version 2**

**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 |

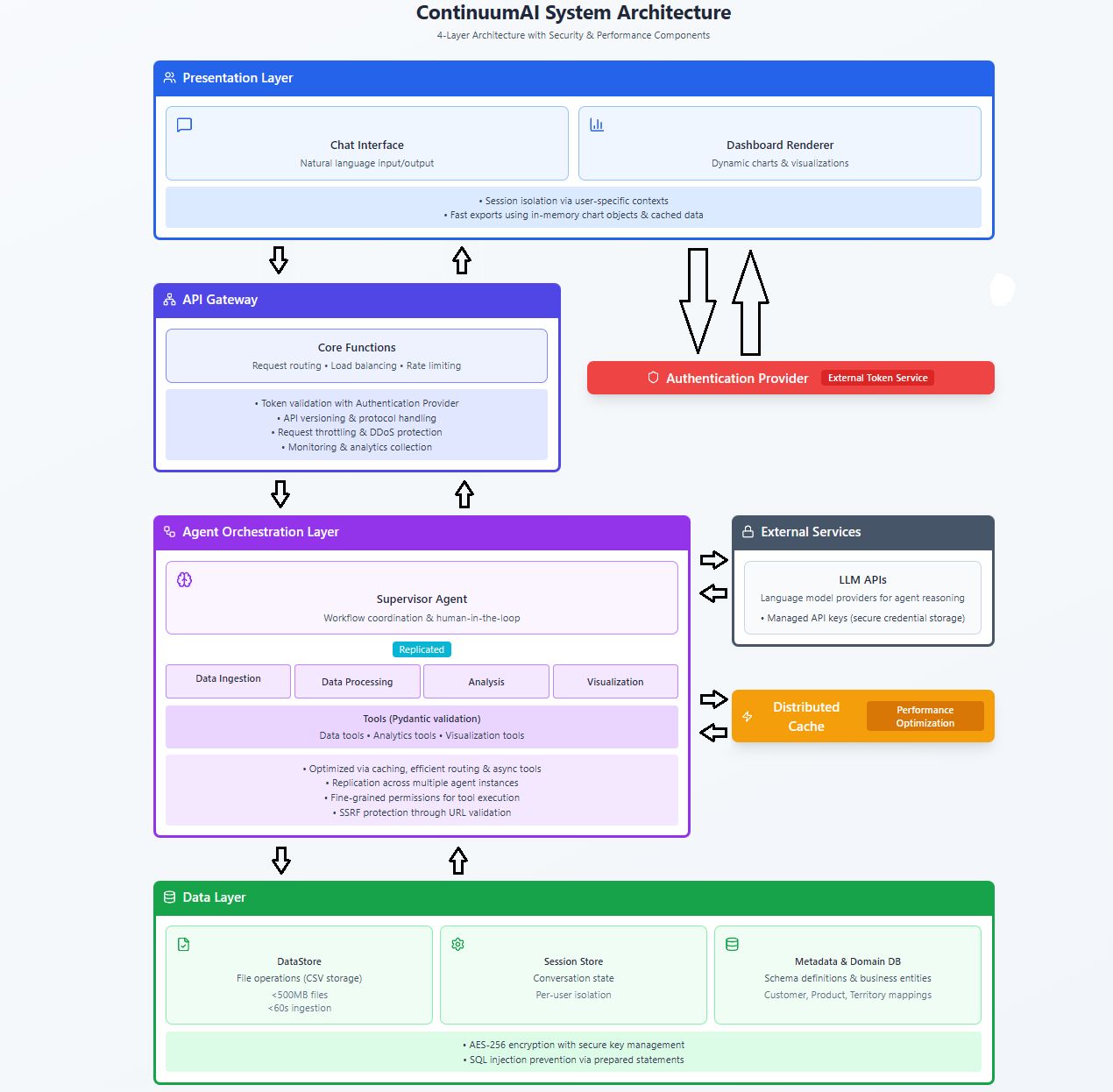
# Project Risk Management

## 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).

## 5.2 Architecture Description

### **Subsystem Descriptions**

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

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

* **Chat Interface:** Accepts natural language queries from users and displays conversational responses, including any necessary clarification requests.
* **Dashboard Renderer:** Dynamically generates and displays data visualizations such as charts and graphs based on analysis results. It also supports exporting charts to PNG and data tables to CSV files.

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

#### **2. API Gateway**

The **API Gateway** serves as the single, intelligent entry point for all incoming traffic from the Presentation Layer. It is responsible for a variety of critical functions:

* **Load Balancing:** Distributes incoming requests across available backend resources to ensure high availability and performance.
* **Authentication and Validation:** Validates user authentication tokens by communicating with an external authentication provider. It rejects any unauthorized requests before they reach the core system.
* **Rate Limiting & Throttling:** Enforces limits on the number of requests to protect backend services from abuse and potential Denial of Service (DoS) attacks.
* **Request Routing:** Directs validated user requests to the appropriate services within the Agent Orchestration Layer.

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

This layer is the core of ContinuumAI's intelligent workflow management. It contains:

* **Supervisor Agent:** Coordinates the overall workflow, routes queries to the appropriate specialized agents, and manages human-in-the-loop interactions for clarifications.
* **Specialized Agents:** A suite of agents handles distinct responsibilities, including Data Ingestion, Data Processing, Analysis, and Visualization.
* **Input Sanitization:** Responsible for sanitizing the content of all user queries to prevent injection attacks before processing.

Each agent utilizes **Tools** (Python functions with Pydantic validation) to perform specific operations like statistical calculations and data transformations. The layer is designed to be **horizontally scalable**, allowing multiple agent instances to handle concurrent user requests.

#### **4. Distributed Cache**

The **Distributed Cache** is a high-speed, in-memory component managed exclusively by the Agent Orchestration Layer. It stores cleaned datasets and frequently accessed columns, acting as the agent layer's working memory. By retrieving pre-processed data from the cache, the system significantly reduces response times for subsequent queries on the same data.

#### **5. Data Layer**

The **Data Layer** manages all persistent storage for the system. It consists of three primary components:

* **DataStore:** Handles file operations for storing and retrieving user-uploaded CSV files.
* **Session Store:** Maintains the conversation state and context for each user session, allowing users to return to previous analyses.
* **Metadata & Domain DB:** This database stores dataset schemas, column mappings, and the business domain models (e.g., Customer, Product, Territory). It connects natural language business terms to the correct technical schema fields, ensuring query accuracy and consistency.

The Data Layer implements **encryption at rest** for all stored files and uses **parameterized queries** to prevent SQL injection.

#### **6. External Services**

This represents the integration boundary where the system connects to external APIs.

* **LLM APIs:** Provides agents with access to third-party large language models (e.g., OpenAI, Anthropic) for natural language reasoning.
* **Authentication Provider:** An external service that issues and manages user authentication tokens, which are validated by the API Gateway.
* **API credentials:** Managed securely, with a URL allowlist enforced to prevent Server-Side Request Forgery (SSRF) attacks.

### **Subsystem Interactions:**

1. **User Query Flow**: A user submits a query via the Chat Interface. The API Gateway intercepts it, validates the user's token, and routes the request to the Supervisor Agent in the Orchestration Layer.
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**: The Data Ingestion Agent processes an uploaded file, which is then cleaned by the Data Processing Agent. The cleaned data is stored in the DataStore, and a copy is placed in the Distributed Cache. Metadata is extracted and saved in the Metadata & Domain DB.
4. **Analysis Execution**: To perform an analysis, an agent first checks the Distributed Cache for data. If it's not available, the agent retrieves it from the DataStore. After performing its computations, it returns the results to the Supervisor Agent.
5. **Visualization Generation**: Visualization Agent receives analysis results → Generates appropriate chart → Sends to Dashboard Renderer → User views the visualization.
6. **State Management**: The Session Store preserves conversation history between sessions, allowing the system to recall previous queries and results to provide context for new requests.
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**: A clear separation of concerns across layers enables independent development and testing of each component. Changes to the UI don't affect agent logic, and data storage modifications don't impact analysis algorithms. | **Latency Overhead**: The API Gateway introduces an additional network hop for every request, which can add minor latency compared to a direct connection. |
| **Scalability**: The API Gateway can manage traffic and load balance across multiple horizontally scaled instances of the Agent Orchestration Layer, allowing the system to handle a high volume of concurrent users. | **Single Point of Failure**: The API Gateway is a critical component. Its failure would block all user requests, requiring robust high-availability configurations. A failure of the Distributed Cache would also significantly degrade performance. |
| **Centralized Security**: The API Gateway acts as a fortified perimeter, handling authentication, request validation, and DDoS protection for the entire system. This centralizes security controls at the entry point. | **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**: The agent-managed Distributed Cache 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, and new routes can be defined in the API Gateway without impacting existing components. | **Configuration Complexity:** The API Gateway is a powerful but complex component to configure correctly, requiring careful setup of routing rules, security policies, and rate limits. |

* **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:   * **Distributed Cache**: When a user queries a dataset that has been previously processed, the Distributed Cache 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, which can reduce the total processing time for complex queries that involve multiple steps. * **Metadata & Domain DB**: By storing dataset schemas separately, the system can quickly understand data structure without opening large CSV files, speeding up query planning.   These optimizations work together to ensure efficient data processing: cached data bypasses Data Layer access, 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 architecture enables efficient ingestion through a clear separation of responsibilities:   * **Agent-Managed Caching**: The Distributed Cache is managed directly by the Agent Orchestration Layer. As data is processed, it is immediately placed into this in-memory cache, which eliminates the need to re-read from the DataStore for the first query and contributes to the sub-60-second readiness time. * **Metadata Separation**: The Metadata & Domain DB allows schema extraction to occur independently of file I/O operations. This lets the system understand the dataset's structure almost instantly after upload, speeding up query planning. * **Modular Data Flow**: The system uses a clear, optimizable flow: Data Ingestion Agent → Data Processing Agent → DataStore / Distributed Cache. This modularity allows for parallel execution where beneficial (e.g., schema extraction during caching) to avoid bottlenecks. |

# 

| **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 for concurrency and scalability through several key features:   * **Session Isolation**: Each user operates in an isolated context managed by the Session Store, ensuring one user's workflow does not interfere with another's. * **API Gateway for Concurrency Management**:   + **Request Rate Limiting**: The API Gateway enforces per-user rate limits, preventing any single user from overwhelming the system and ensuring fair resource allocation.   + **Load Balancing**: The gateway distributes traffic across multiple instances of the Agent Orchestration Layer, allowing the system to scale horizontally to handle increased load without performance degradation. * **Future-Ready Scalability**: The Agent Orchestration Layer is designed to be horizontally scalable. New agent instances can be deployed behind the API Gateway to handle more concurrent requests without needing an architectural redesign. |

# 

# 

| **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, it retains the chart object in memory within the user’s session. Exporting to PNG simply serializes this pre-existing object, so no recomputation is required. * **Cached Data Reuse**: For data table exports, the Distributed Cache retains the results from prior analysis. When the user requests to export to CSV, the system writes this cached data directly to the file, avoiding a full re-query from the Data Layer. * **Session-Aware Presentation Layer**: The Presentation Layer maintains references to each chart and table created in the current session, allowing export functions to access them directly without additional inter-layer communication. |

* **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:   * **API Gateway (Authentication & Validation)**: As the system's entry point, the API Gateway is the first line of defense. It validates each incoming request's authentication token with an external provider and rejects any unauthorized requests before they reach the agent system. * **Session Isolation (Presentation & Data Layers)**: Each user session is isolated via unique identifiers in the Session Store. This ensures User A cannot access User B's datasets, conversation history, or cached results. * **Per-User Data Isolation (Data Layer)**: The DataStore organizes files by user session, and the Metadata & Domain DB tags schema entries with user identifiers to prevent cross-user data access. |

| **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 in the DataStore are encrypted. The component handles encryption and decryption transparently, ensuring agents work with decrypted data in memory while all persistent storage remains encrypted. * **Secure Credential Management (External Services)**: API credentials are managed at the External Services integration boundary and retrieved securely at runtime. They are never logged or exposed in error messages. * **Encrypted Communication (Cross-Layer)**: When data moves between layers (e.g., Presentation ↔ API Gateway ↔ Agent Orchestration ↔ Data), communication occurs over encrypted channels like HTTPS in production to protect data in transit. |

| **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:   * **API Gateway (Perimeter Defense)**: The API Gateway provides the initial layer of protection by validating request structures and rejecting malformed inputs at the network edge, before they can reach the application logic. * **Agent Orchestration Layer (Content Sanitization)**: This layer is responsible for the deep sanitization of query contents. It checks for malicious patterns in natural language input, such as SQL injection attempts or malicious formulas in uploaded CSV files, before they are processed by agents or tools. * **Pydantic Schema Validation (Agent Tools)**: All tools used by agents enforce strict input schemas using Pydantic validators. This ensures that inputs match the expected data types and formats, preventing code injection attacks at the function level. * **Parameterized Queries (Data Layer)**: The Metadata & Domain DB uses parameterized SQL queries exclusively. This practice ensures that user input is treated as data, not as executable code, which eliminates the risk of SQL injection at the database level. |

| **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 exhaustion:   * **User Request Rate Limiting (API Gateway)**: The API Gateway limits each user to a maximum number of requests per minute. This prevents a single user from overwhelming the system and degrading performance for others. * **Storage Quota Enforcement (Data Layer)**: The DataStore tracks storage per user and rejects new uploads once a predefined quota is met, preventing uncontrolled disk space consumption. * **LLM API Rate Limiting (Agent Orchestration Layer)**: The system implements global rate limiting on calls to external LLM APIs to control costs and stay within provider limits. |

| **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 Sanitization (Agent Orchestration Layer)**: The agent layer is responsible for sanitizing all inputs, rejecting any user-supplied data that contains URL patterns or attempts to direct the system to external resources. The system is designed to only accept natural language queries and dataset identifiers from users. * **URL Allowlisting (Agent Orchestration Layer)**: As a secondary defense, the orchestration layer maintains a hardcoded allowlist of permitted external endpoints (e.g., api.openai.com). Any attempt by an agent to make a request to a URL not on this list is automatically blocked. |

# 

# 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.

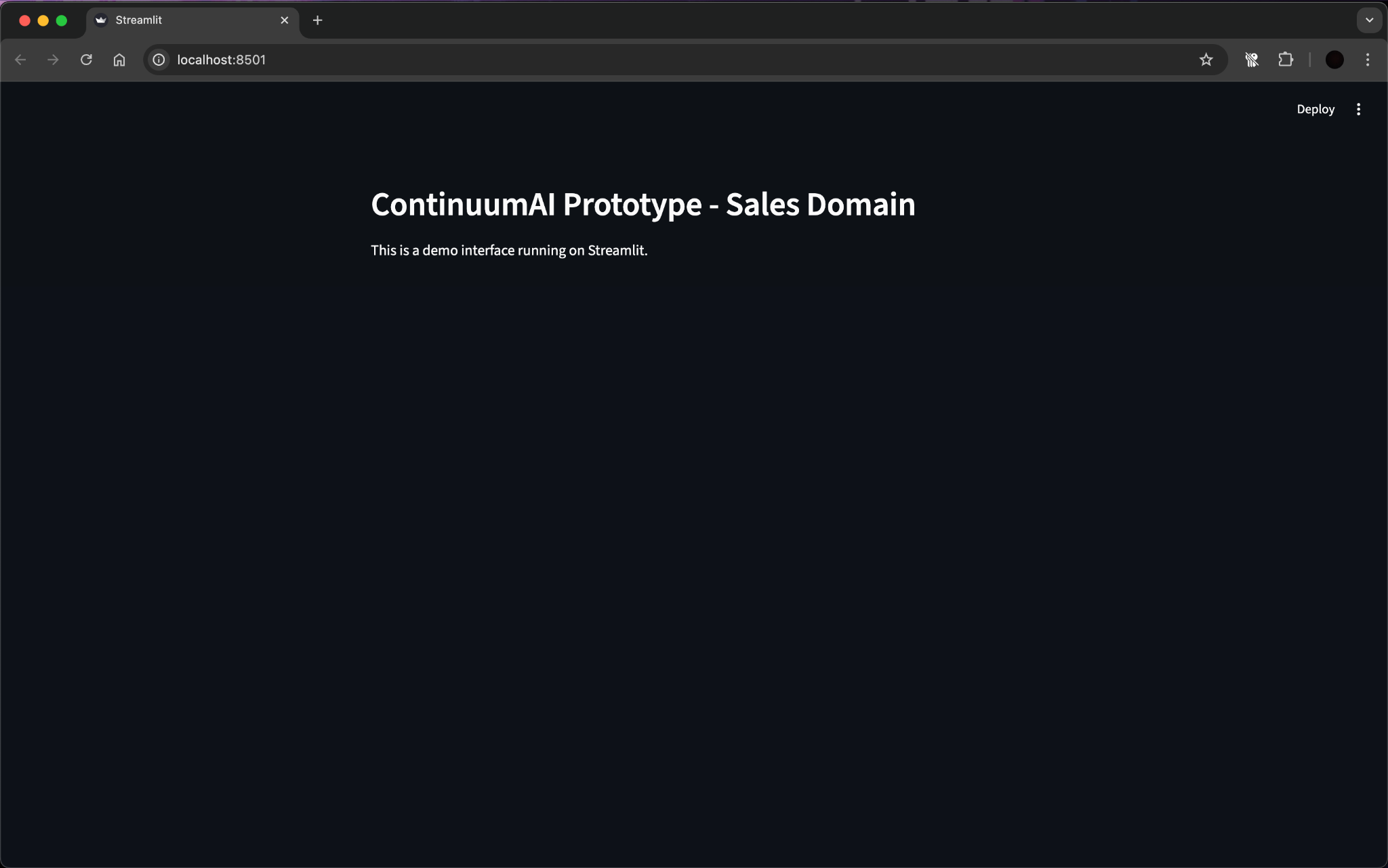
# Development Environment Preparation

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

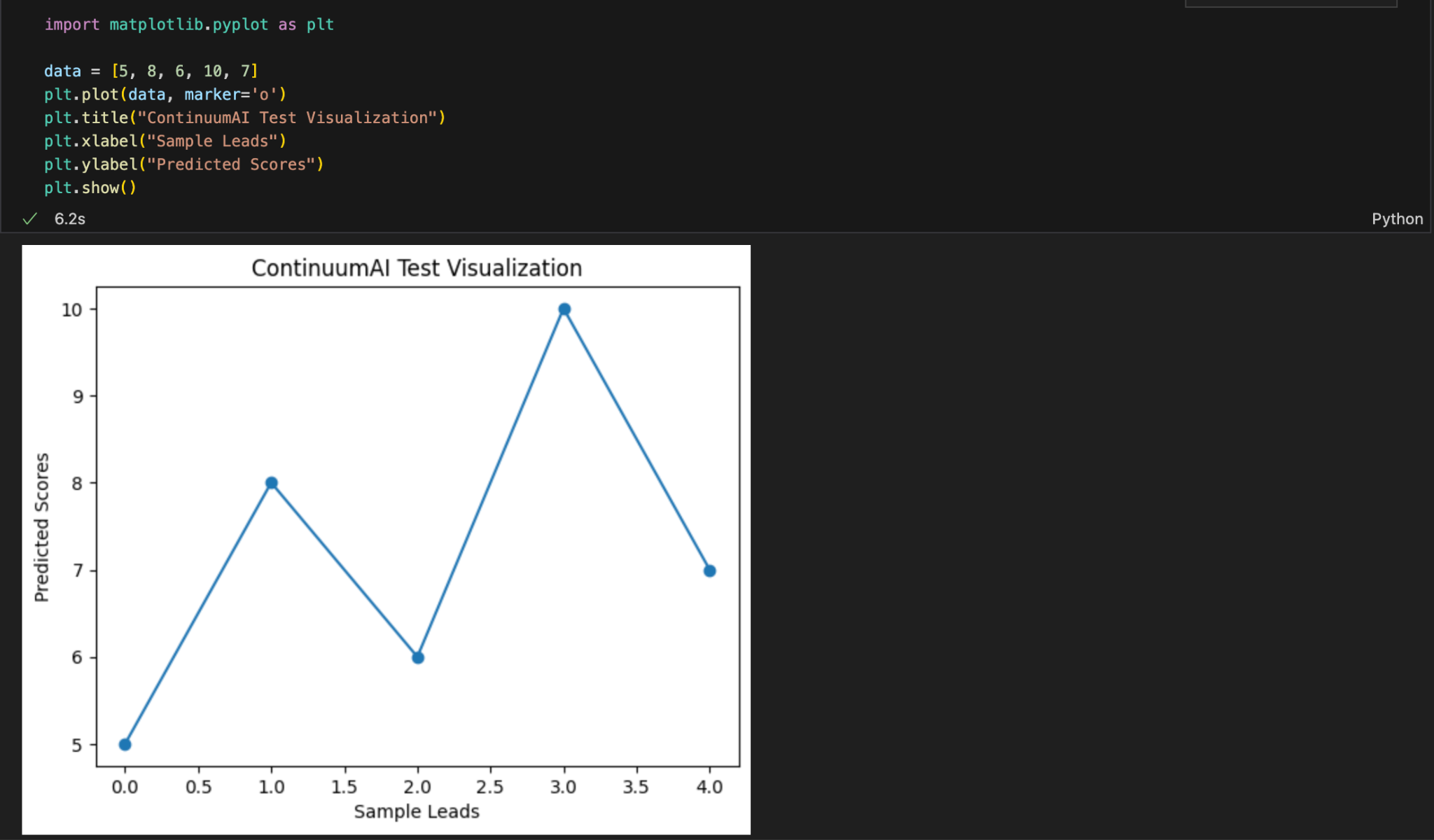
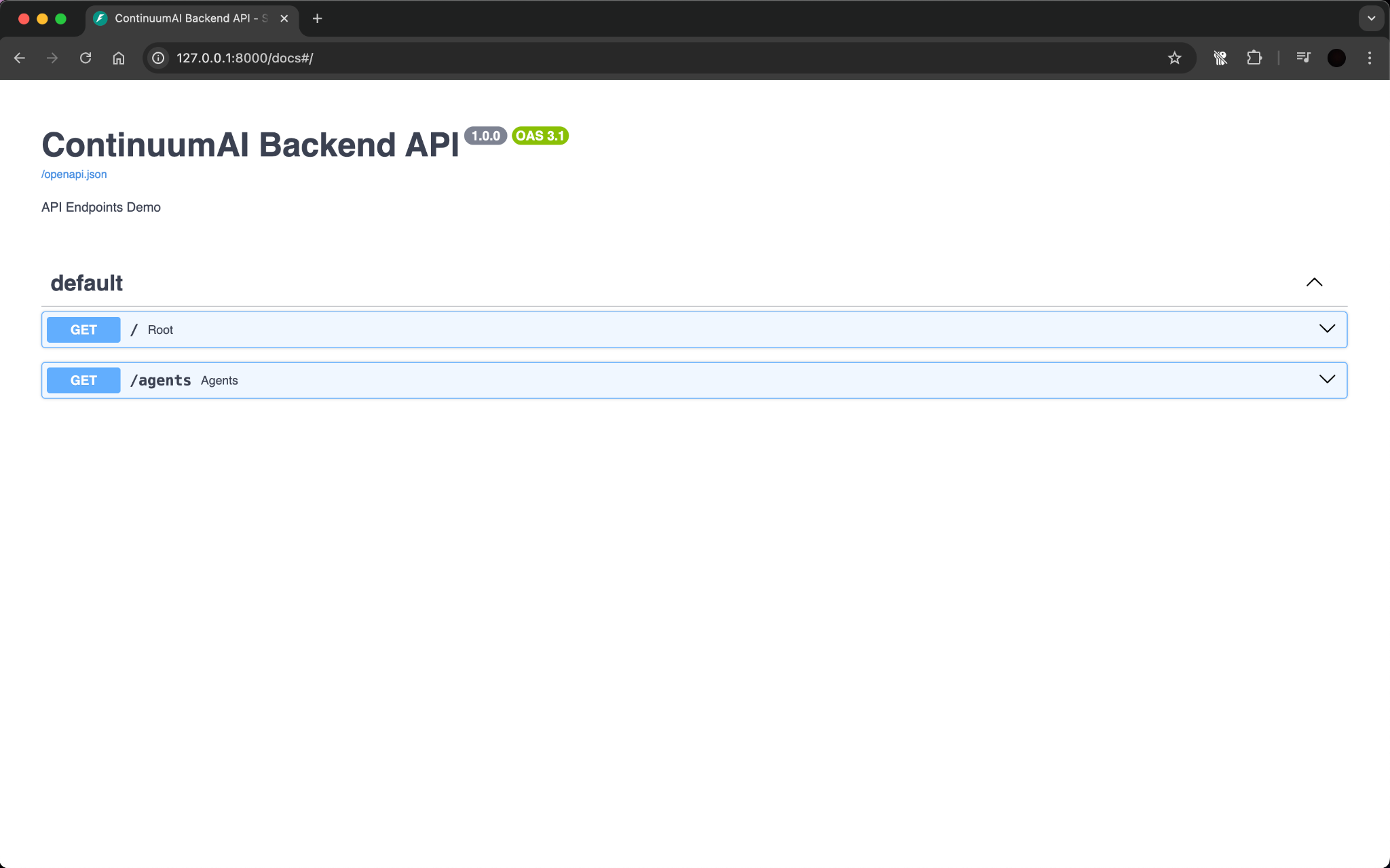


Figure 2: Jupyter Notebook environment running a sample Python visualization on dummy data for ContinuumAI.



**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

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

* 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 |

**13. Revisions**

**Changes Version 2:**

The system architecture was significantly revised to improve modularity, security, and clarity. Key changes include replacing the load balancer and dedicated security middleware with a unified API Gateway that now handles request routing, authentication, and rate limiting. The separate Domain Model Layer was removed, and its responsibilities were merged into the Data Layer to centralize data and business logic. The cache was repositioned as a distributed component managed exclusively by the Agent Orchestration Layer to improve performance and simplify data flow. These changes resulted in a more streamlined four-layer architecture with clearer separation of concerns.