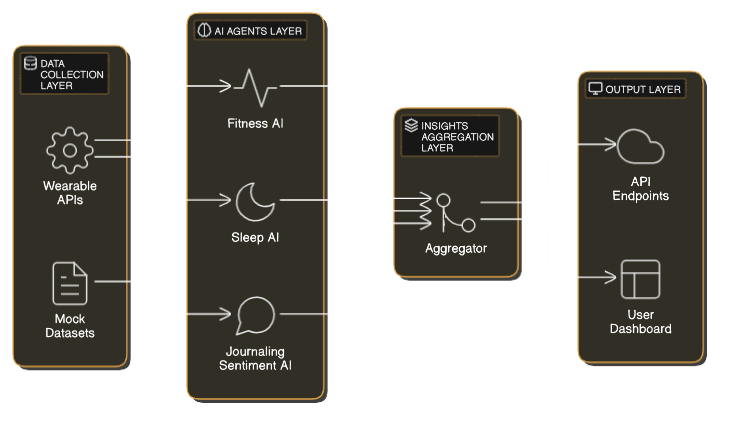
# System Architecture Documentation

## 1. System Architecture Diagram

The system architecture diagram illustrates the major components of the system and their interactions. It includes the following layers: Data Collection, AI Agents, Insights Aggregation, and Output.

Refer to the attached diagram for a visual representation.



## 2. Implementation Plan

The implementation plan outlines the data flow, technologies, and modular design of the system.

### 2.1 Data Flow

1. Input Layer: Collects data from wearable APIs using REST or GraphQL. Fallback mock data is used in case API access is unavailable.  
2. AI Processing Layer: Processes fitness data using prediction models, analyzes sleep data for trends, and performs sentiment analysis on journaling data.  
3. Insights Aggregation Layer: Combines outputs from all agents into personalized recommendations.  
4. Output Layer: Provides insights via API endpoints and a Flask-based dashboard.

### 2.2 Technologies to Use

• Backend Framework: Flask or Django for API integration.  
• AI Libraries: TensorFlow, PyTorch, Scikit-learn, Vader, or Hugging Face Transformers.  
• Database: PostgreSQL or MongoDB for data storage.  
• Frontend Framework: HTML/CSS/JavaScript or React for dashboards.

### 2.3 Modularity

The system is designed with modularity in mind to allow easy addition of new AI agents or data sources. APIs are built using microservices architecture to enable independent updates.

## 3. Scalability Considerations

1. Handling Increased Data Volume: Utilize a distributed architecture such as AWS Lambda or Kubernetes. Optimize APIs with caching mechanisms like Redis.  
2. Integrating Additional APIs: Build reusable connectors for new APIs and use middleware to handle API rate limits and data validation.

## 4. Challenges and Mitigation

1. API Rate Limits:  
 • Challenge: APIs may limit the number of requests.  
 • Solution: Implement request throttling and caching mechanisms to minimize API calls.  
  
2. Data Inconsistencies:  
 • Challenge: Variability in data formats across APIs.  
 • Solution: Standardize data using a schema and validate incoming data.  
  
3. Scalability for Multiple Users:  
 • Challenge: Growing user base may strain resources.  
 • Solution: Use horizontal scaling with load balancers.  
  
4. Real-time Data Processing:  
 • Challenge: Delivering timely insights as data streams in.  
 • Solution: Use streaming platforms such as Apache Kafka for real-time data ingestion.