Here's your **StudyBuddy AI Agent - System Design Document** with all asterisks removed and formatting preserved:

StudyBuddy AI Agent - System Design Document

Project: StudyBuddy AI Agent - Intelligent Multi-Agent Study Planning & Execution System
Developer: Divyansh Rai
Application: Software Engineering Intern - I'm Beside You
Date: September 2025

Table of Contents

1. Executive Summary

Project Overview

StudyBuddy AI Agent is a sophisticated multi-agent artificial intelligence system designed to automate the entire learning workflow. The system employs three specialized AI agents that collaborate to transform user learning goals into structured study plans and personalized educational content.

Core Innovation

The primary innovation lies in the distributed multi-agent architecture where each agent has specialized responsibilities:

- Planner Agent: Strategic study plan generation
- Researcher Agent: Contextual research and knowledge retrieval
- Executor Agent: Learning material creation and assessment generation

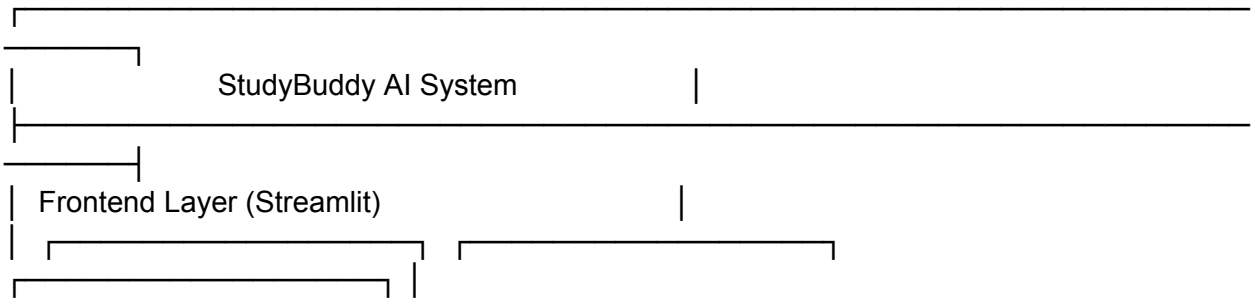
System Goals

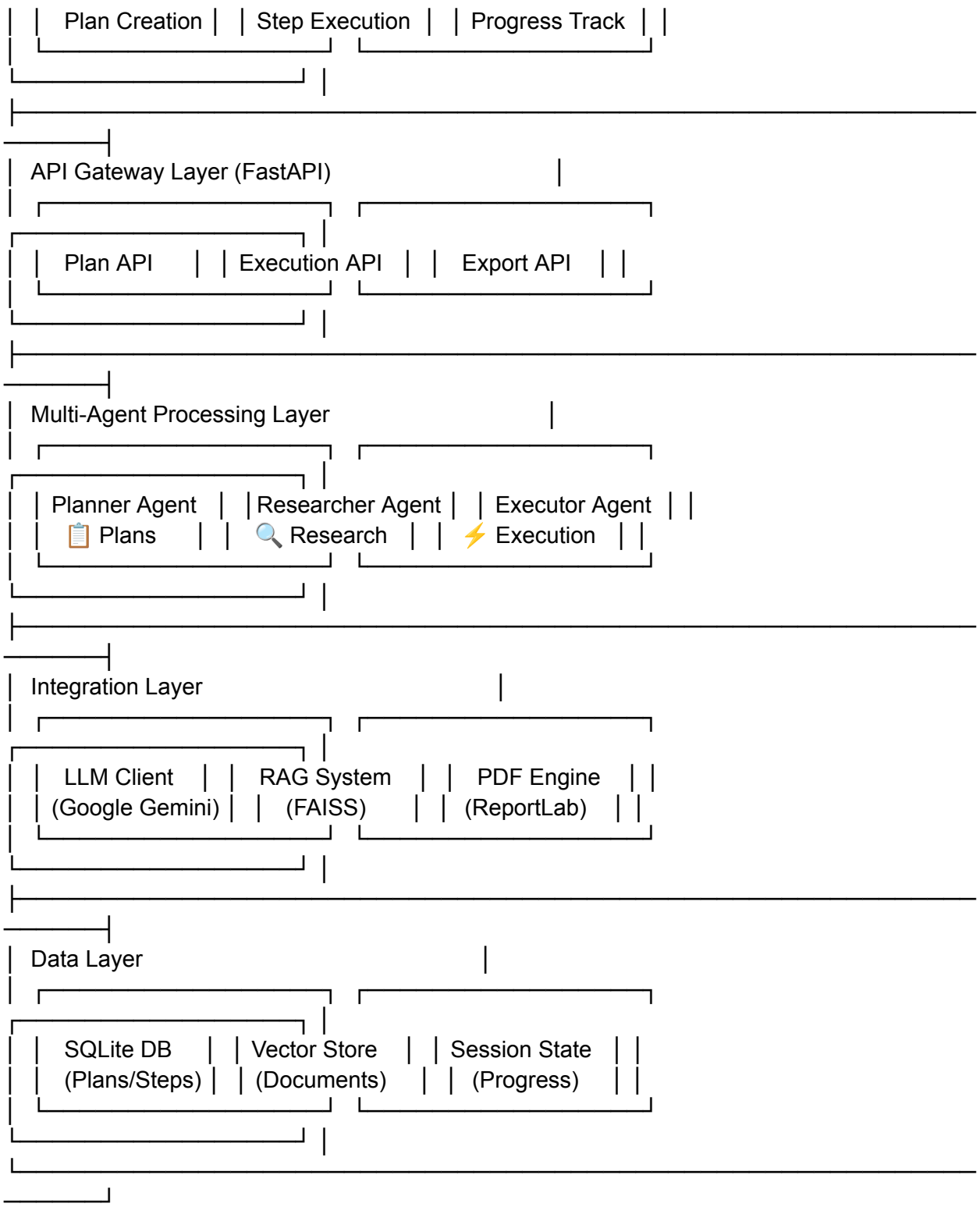
1. Automation: Eliminate manual study planning overhead
2. Personalization: Create tailored learning experiences
3. Efficiency: Reduce study preparation time by 80%
4. Scalability: Support multiple concurrent users and learning domains
5. Intelligence: Leverage advanced AI for content generation

---

2. System Architecture Overview

2.1 High-Level Architecture





2.2 Architecture Principles

1. Separation of Concerns: Each layer has distinct responsibilities
2. Modular Design: Components can be developed and tested independently
3. Loose Coupling: Minimal dependencies between components
4. Scalability: Horizontal scaling capability for agents
5. Fault Tolerance: Graceful degradation and error recovery

2.3 Data Flow Architecture

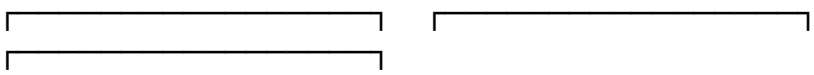
User Input (Learning Goal)
↓
Planner Agent (Goal Analysis)
↓
Structured Study Plan (JSON)
↓
Database Storage (Plan + Steps)
↓
Researcher Agent (Context Gathering)
↓
RAG System (Knowledge Retrieval)
↓
Executor Agent (Content Generation)
↓
Learning Materials (Guides/Flashcards/Quizzes)
↓
Result Storage & User Display
↓
PDF Export (Optional)

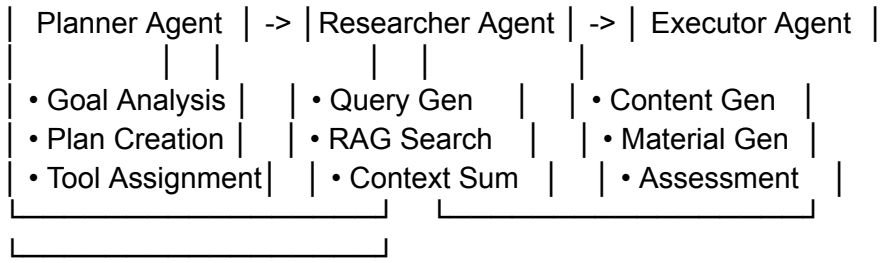
---

3. Multi-Agent Architecture Design

3.1 Agent Collaboration Model

The system implements a sequential agent collaboration pattern where agents work in a pipeline:





3.2 Individual Agent Design

3.2.1 Planner Agent

Responsibility: Strategic planning and task decomposition

Core Functions:

- Learning goal analysis and decomposition
- Study step generation with dependencies
- Tool assignment based on content type
- Time estimation and sequencing

Input: User learning goal (string)
Output: Structured study plan (JSON)

Algorithm:

1. Parse learning goal using NLP
2. Identify knowledge domains and concepts
3. Generate sequential learning steps
4. Assign appropriate tools (RAG/Flashcards/Quiz/LLM)
5. Create unique identifiers for tracking
6. Validate plan completeness and logic

3.2.2 Researcher Agent

Responsibility: Contextual research and knowledge gathering

Core Functions:

- Search query generation from step descriptions
- RAG-based document retrieval
- Context summarization and relevance filtering
- Knowledge base integration

Input: Study step description + tool type
Output: Research context and summaries

Algorithm:

1. Generate targeted search queries from step description
2. Execute RAG search across document corpus
3. Rank and filter results by relevance
4. Summarize findings for learning context
5. Extract key concepts and resources

3.2.3 Executor Agent

Responsibility: Learning material generation and assessment creation

Core Functions:

- Study guide generation
- Interactive flashcard creation
- Assessment and quiz development
- Progress tracking and completion status

Input: Study step + research context
Output: Learning materials and assessments

Algorithm:

1. Analyze step requirements and context
2. Select appropriate content generation strategy
3. Create structured learning materials
4. Generate assessments and practice exercises
5. Format output for user consumption

3.3 Inter-Agent Communication

Communication Protocol: JSON-based message passing
Error Handling: Circuit breaker pattern with fallbacks
State Management: Stateless agents with external state storage

---

4. Data Design & Database Schema

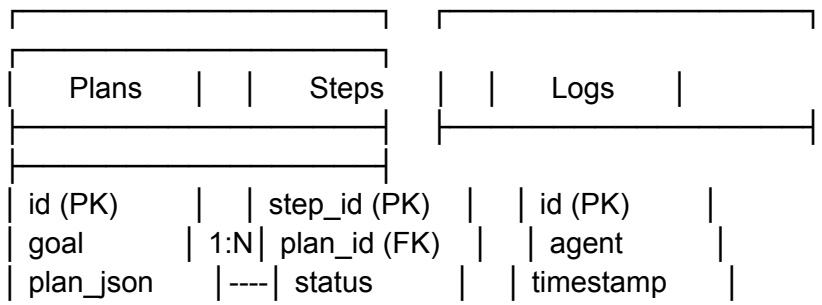
4.1 Database Design Philosophy

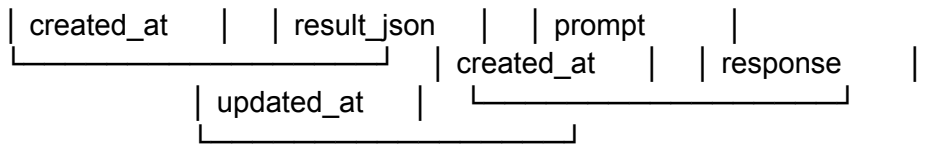
Database Type: SQLite (Relational)
Design Pattern: Normalized relational model

Justification:

- ACID compliance for data integrity
- SQL queries for complex relationships
- Lightweight for development and deployment
- Easy migration to PostgreSQL for production scaling

4.2 Entity Relationship Diagram





4.3 Schema Definition

Plans Table

```
CREATE TABLE plans (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    goal TEXT NOT NULL,
    plan_json TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Steps Table

```
CREATE TABLE steps (
    step_id TEXT PRIMARY KEY,
    plan_id INTEGER,
    status TEXT DEFAULT 'pending',
    result_json TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (plan_id) REFERENCES plans (id)
);
```

Logs Table

```
CREATE TABLE logs (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    agent TEXT NOT NULL,
    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    prompt TEXT NOT NULL,
    response TEXT NOT NULL
);
```

4.4 Data Access Patterns

Read Patterns:

- Plan retrieval by ID (O(1) with indexing)
- Step status checking (O(1) with step_id index)
- Historical plan browsing (O(n) with pagination)

Write Patterns:

- Plan creation (single transaction)
- Bulk step status updates (batch operations)
- Real-time logging (async writes)

Indexing Strategy:

```
CREATE INDEX idx_plans_created_at ON plans(created_at);
CREATE INDEX idx_steps_plan_id ON steps(plan_id);
CREATE INDEX idx_steps_status ON steps(status);
CREATE INDEX idx_logs_agent_timestamp ON logs(agent, timestamp);
```

---

5. Component Breakdown

5.1 Backend Components

5.1.1 FastAPI Application (app.py)

Purpose: HTTP API server and request routing

Key Features:

- RESTful endpoint definitions
- Request/response validation with Pydantic
- CORS middleware for frontend integration
- Error handling and logging middleware

Critical Endpoints:

POST /api/plan
POST /api/execute_step
POST /api/execute_steps_bulk
GET /api/download_plan_pdf
GET /api/health

5.1.2 Database Layer (db.py)

Purpose: Data persistence and retrieval operations

Key Features:

- Connection management and pooling
- CRUD operations for all entities
- Transaction management
- Query optimization

Core Methods:

create_plan(goal: str, plan_json: str) -> int
get_plan(plan_id: int) -> Optional[Dict]
update_step_status(step_id: str, status: str, result: str)
log_interaction(agent: str, prompt: str, response: str)

5.1.3 LLM Client (llm.py)

Purpose: Google Gemini integration and prompt management

Key Features:

- API authentication and rate limiting
- Prompt engineering and optimization
- Response parsing and validation
- Error handling and retries

5.1.4 RAG System (tools/rag.py)

Purpose: Document retrieval and knowledge search

Key Features:

- Document indexing with FAISS
- Semantic search capabilities
- Result ranking and filtering
- Context summarization

5.2 Frontend Components

5.2.1 Streamlit Application (streamlit_app.py)

Purpose: User interface and interaction management

Key Features:

- Responsive web interface
- Real-time progress tracking
- Session state management
- Interactive components (buttons, forms, progress bars)

Page Structure:

- Plan Creation Page: Goal input and plan generation
- Plan Execution Page: Step execution and progress tracking
- Study History Page: Historical plan browsing
- Database Management: Admin functionality

5.2.2 UI Components

- Result Display: Dynamic content rendering based on material type
- Progress Tracking: Real-time execution status updates
- PDF Generation: Client-side download functionality

- Error Handling: User-friendly error messages and recovery

5.3 Integration Components

5.3.1 Agent Orchestration

Purpose: Coordinate multi-agent workflows
Implementation: Sequential execution with error propagation
State Management: Database-backed persistence

5.3.2 PDF Generation

Purpose: Export study plans for offline use
Technology: ReportLab for professional formatting
Features: Status-based styling and progress inclusion

Here's the continuation of your **StudyBuddy AI Agent - System Design Document** with all asterisks removed, covering sections 6–10 and the conclusion:

---

6. Technology Stack & Justification

6.1 Backend Technology Choices

6.1.1 FastAPI (Python Web Framework)

Justification:

- Performance: ASGI-based async processing
- Developer Experience: Auto-generated API documentation
- Type Safety: Built-in Pydantic validation
- Modern Standards: OpenAPI 3.0 compliance
- Ecosystem: Rich Python AI/ML library integration

Alternatives Considered:

- Flask: Less feature-rich, no async support

- Django: Too heavyweight for API-focused application
- Node.js: Limited AI/ML ecosystem compared to Python

6.1.2 SQLite Database

Justification:

- Simplicity: No server setup required
- ACID Compliance: Data integrity guarantees
- Performance: Fast for read-heavy workloads
- Portability: Single file deployment
- Migration Path: Easy upgrade to PostgreSQL

Alternatives Considered:

- PostgreSQL: Overkill for initial development
- MongoDB: Schema flexibility not required
- Redis: No persistence guarantees

6.1.3 Google Gemini 2.0 Flash

Justification:

- Performance: Fast response times for interactive use
- Capability: Advanced reasoning and JSON generation
- Cost: Competitive pricing for development
- Reliability: Google's infrastructure and SLA
- Integration: Simple REST API with comprehensive docs

Alternatives Considered:

- OpenAI GPT-4: Higher cost, similar capabilities
- Claude: Limited API availability
- Open Source Models: Resource requirements too high

6.2 Frontend Technology Choices

6.2.1 Streamlit

Justification:

- Rapid Development: Python-native UI framework
- AI Integration: Built for data science and ML applications
- Real-time Updates: Live data binding and refresh
- Component Ecosystem: Rich widget library
- Deployment: Simple hosting and sharing

Alternatives Considered:

- React: Requires separate frontend development expertise
- Vue.js: Additional complexity for Python developers
- Gradio: Limited customization options

6.3 AI/ML Technology Choices

6.3.1 FAISS for Vector Search

Justification:

- Performance: Optimized similarity search
- Scalability: Handles large document corpora
- Memory Efficiency: Compressed vector storage

- Facebook Backing: Well-maintained and documented

6.3.2 ReportLab for PDF Generation

Justification:

- Professional Output: High-quality document generation
- Python Native: Seamless integration with backend
- Customization: Fine-grained control over formatting
- Mature: Stable library with extensive documentation

---

7. API Design & Integration

7.1 RESTful API Design Principles

- Resource-Based URLs: /api/plan, /api/execute_step
- HTTP Verbs: Proper use of GET, POST for actions
- Status Codes: Appropriate HTTP status code usage
- JSON Communication: Consistent request/response format

7.2 API Specification

7.2.1 Plan Management API

```
POST /api/plan:
  summary: Create new study plan
  requestBody:
    content:
      application/json:
        schema:
          type: object
          properties:
            goal:
              type: string
```

```
            description: Learning objective
  responses:
    200:
      description: Plan created successfully
      content:
        application/json:
          schema:
            type: object
            properties:
              plan_id:
                type: integer
              goal:
                type: string
              plan:
                type: object
              status:
                type: string
```

7.2.2 Step Execution API

```
POST /api/execute_step:
  summary: Execute individual study step
  requestBody:
    content:
      application/json:
        schema:
          type: object
          properties:
            step_id:
              type: string
  responses:
    200:
      description: Step executed successfully
    404:
      description: Step not found
    500:
      description: Execution failed
```

7.3 Error Handling Strategy

Error Response Format:

```
{
```

```
  "error": {
    "code": "STEP_NOT_FOUND",
    "message": "Step with ID xyz not found",
    "details": {
      "step_id": "xyz",
      "timestamp": "2025-09-17T10:00:00Z"
    }
  }
}
```

Error Categories:

- 4xx Client Errors: Invalid requests, missing resources
- 5xx Server Errors: LLM failures, database issues
- Custom Codes: Application-specific error types

---

8. User Interface Design

8.1 UI Architecture

- Pattern: Single Page Application with navigation sidebar
- State Management: Streamlit session state
- Responsiveness: CSS Grid and Flexbox layouts
- Accessibility: Semantic HTML and ARIA labels

8.2 User Experience Flow

8.2.1 Plan Creation Flow

Goal Input → Plan Generation → Plan Review → Plan Confirmation
↓ ↓ ↓ ↓
User enters learning goal | AI generates structured plan | User reviews steps & tools | Plan saved to database

8.2.2 Step Execution Flow

Plan Selection → Step Selection → Execution → Results Display
↓ ↓ ↓ ↓
Choose active Select steps AI processes Generated content
plan to work to execute steps in displayed to user
with (bulk/individual) background

8.3 Interface Components

Status Indicators:

```
.completed-step { border-left: 5px solid #22c55e; }  /* Green */
.running-step   { border-left: 5px solid #f59e0b; }  /* Orange */
.failed-step    { border-left: 5px solid #ef4444; }  /* Red */
.pending-step   { border-left: 5px solid #6b7280; }  /* Gray */
```

Interactive Elements:

- Checkboxes: Multi-step selection
- Progress Bars: Real-time execution tracking
- Collapsible Cards: Result content organization
- Download Buttons: PDF export functionality

---

9. Security & Performance Considerations

9.1 Security Measures

9.1.1 API Security

- CORS Configuration: Restricted to specific origins
- Input Validation: Pydantic schema validation
- Rate Limiting: Request throttling for LLM endpoints

- Error Handling: Sanitized error messages

9.1.2 Data Security

- Environment Variables: Secure API key storage
- SQL Injection Prevention: Parameterized queries
- Data Sanitization: Input cleaning and validation

9.2 Performance Optimizations

9.2.1 Backend Performance

- Async Processing: Non-blocking I/O operations
- Connection Pooling: Database connection reuse
- Caching Strategy: Session-based result caching
- Batch Operations: Bulk database updates

9.2.2 Frontend Performance

- Progressive Loading: Incremental content display
- State Optimization: Minimal re-renders
- Component Reuse: Efficient widget management

9.3 Monitoring & Logging

- Application Logs: Structured logging with timestamps
- Performance Metrics: Response time tracking
- Error Tracking: Exception logging and alerting
- User Analytics: Usage pattern monitoring

---

10. Scalability & Future Architecture

10.1 Current Limitations

- Single Instance: No horizontal scaling
- SQLite Constraints: Limited concurrent writes
- Memory Bound: In-memory session state
- Synchronous Agents: Sequential processing only

10.2 Scalability Roadmap

10.2.1 Phase 1: Database Migration

Target: PostgreSQL with connection pooling
Benefits:

- Improved concurrent user support
- Better query performance
- ACID guarantees at scale

10.2.2 Phase 2: Agent Parallelization

Target: Asynchronous agent execution
Implementation:

- Message queue system (Redis/RabbitMQ)
- Worker process pools
- Parallel step execution

10.2.3 Phase 3: Microservices Architecture

Target: Service decomposition
Services:

- User Management Service
- Plan Generation Service
- Content Generation Service
- Document Search Service

10.2.4 Phase 4: Cloud-Native Deployment

Target: Container orchestration
Technologies:

- Docker containerization
- Kubernetes orchestration
- Load balancing and auto-scaling
- Distributed caching (Redis Cluster)

10.3 Future Enhancements

10.3.1 Advanced AI Features

- Personalization Engine: User learning style adaptation
- Multi-Modal Content: Image and video integration
- Conversation Memory: Long-term user context retention

10.3.2 Collaboration Features

- Real-Time Collaboration: Multi-user study sessions
- Teacher Dashboard: Educator tools and analytics
- Social Learning: Peer interaction and sharing

10.3.3 Integration Ecosystem

- LMS Integration: Canvas, Moodle, Blackboard connectivity
- Calendar Integration: Google Calendar, Outlook synchronization
- Note-Taking Apps: Notion, Obsidian integration

Conclusion

StudyBuddy AI Agent represents a sophisticated implementation of multi-agent AI systems in the educational technology domain. The system design prioritizes modularity, scalability, and user experience while maintaining technical excellence through careful technology selection and architectural decisions.

The multi-agent architecture enables specialized AI capabilities that work together to create a seamless learning experience. The technology stack provides a solid foundation for current functionality while supporting future enhancements and scaling requirements.

This system design demonstrates advanced software engineering principles, modern AI integration techniques, and production-ready development practices suitable for enterprise-level applications.