Prompt :

Act as an expert in DevOps, IT Service Management (ITSM), and Generative AI system design. I want to create a **local web application** that uses a **local LLM** (e.g., Mistral, Phi, LLaMA) and **RAG architecture** to read company documents (like policies, logs, procedures, etc.) stored in PDF or text format. This web app should be capable of the following: 1. **Generate industry-standard documentation**: - Based on ITIL or best practices for: - Change Management - Problem Management - Incident Management - Technical SOPs for Implementation & BAU (Business-as-Usual) Operations - Adapt based on whether the target environment is **on-premises** or **cloud** (AWS/Azure). 2. **Use an AI Agent** to: - Monitor logs, metrics, system health data (collected locally or via API). - Analyze issues and generate daily or weekly reports. - Suggest corrective actions or automation improvements. 3. **Generate presentations or dashboards** for: - Capacity Planning - User Access Management - Vulnerability Management - Full-stack IT Infra visibility (network, compute, storage, application, DB) Build a modular system that allows document ingestion (via RAG), agent-driven automation, and dynamic content generation (Markdown, HTML, or PowerPoint). Please help me: - Break down the components required - Suggest the tech stack (LLMs, vector DBs, web UI) - Give me the architectural flow - Point out common challenges and best practices

I'll help you design a comprehensive local AI-powered ITSM documentation and monitoring system. Let me break this down into manageable components with a practical architecture.

**Local AI-Powered ITSM System Architecture**

**System Overview**

A modular, locally-hosted web application that combines RAG-based document processing with AI agents for automated ITSM operations, monitoring, and documentation generation.

**1. Core Components Breakdown**

**1.1 Document Ingestion & RAG Pipeline**

* **PDF/Text Parser**: Extract and preprocess documents (policies, SOPs, logs)
* **Vector Database**: Store embeddings of chunked documents
* **Embedding Model**: Generate semantic embeddings (sentence-transformers)
* **Retrieval System**: Query relevant context for LLM prompts

**1.2 Local LLM Engine**

* **Model Options**: Mistral 7B, Code Llama, Phi-3, or LLaMA-2
* **Inference Server**: Ollama, LM Studio, or vLLM for serving models
* **Context Management**: Handle long documents and conversation history
* **Prompt Engineering**: Template system for different ITSM domains

**1.3 AI Agent Framework**

* **Task Orchestrator**: Manage automated workflows
* **Data Collectors**: Monitor logs, metrics, system health
* **Analysis Engine**: Process data and generate insights
* **Report Generator**: Create periodic reports and alerts
* **Action Recommender**: Suggest improvements and automations

**1.4 Web Application Framework**

* **Frontend**: React/Vue.js with dashboard components
* **Backend API**: FastAPI/Flask for RESTful services
* **Authentication**: Local user management
* **File Management**: Upload/manage documents and configurations

**1.5 Output Generation System**

* **Markdown Processor**: Convert AI outputs to formatted docs
* **HTML Generator**: Create web-ready documentation
* **Presentation Builder**: Generate PowerPoint/PDF reports
* **Dashboard Engine**: Real-time metrics visualization

**2. Recommended Tech Stack**

**2.1 Core Technologies**

LLM Runtime:

- Ollama (easiest setup) or LM Studio

- Alternative: Text Generation WebUI

Vector Database:

- Chroma (lightweight, embedded)

- Alternative: Qdrant, Weaviate

Backend:

- Python FastAPI

- LangChain/LlamaIndex for RAG

- SQLite/PostgreSQL for metadata

Frontend:

- React with TypeScript

- Chart.js/D3.js for visualizations

- Ant Design/Material-UI for components

Document Processing:

- PyPDF2/pdfplumber for PDFs

- python-docx for Word docs

- pandas for structured data

Output Generation:

- python-pptx for PowerPoint

- markdown/mistune for Markdown

- matplotlib/plotly for charts

**2.2 Model Recommendations**

* **General ITSM**: Mistral 7B Instruct v0.2 (good balance)
* **Code/Technical**: Code Llama 7B/13B
* **Analysis**: Phi-3 Medium (efficient for reasoning)
* **Embeddings**: all-MiniLM-L6-v2 or e5-large

**3. Architectural Flow**

**3.1 Data Flow Architecture**

[Document Upload] → [PDF/Text Parser] → [Text Chunker] → [Embedding Model]

↓

[Vector Database] ← [Indexed Chunks] ← [Chunk Processor] ← [Embeddings]

↓

[Query Interface] → [Similarity Search] → [Context Retrieval] → [LLM Prompt]

↓

[Local LLM] → [Response Generation] → [Output Processor] → [Document Generator]

**3.2 Agent Workflow**

[Data Sources] → [Collectors] → [Preprocessing] → [Analysis Agent]

(Logs, Metrics, ↓ ↓ ↓

APIs, Files) [Storage] [Normalization] [Pattern Detection]

↓

[Action Engine] ← [Recommendations] ← [Report Generation] ← [Insights]

↓ ↓

[Automation] [Dashboard/Alerts]

**3.3 System Architecture**

┌─────────────────────────────────────────────────────────────┐

│ Frontend (React) │

├─────────────────────────────────────────────────────────────┤

│ API Gateway (FastAPI) │

├─────────────────────────────────────────────────────────────┤

│ Document │ AI Agent │ LLM Service │ Output │

│ Management │ Framework │ (Ollama) │ Engine │

├─────────────────────────────────────────────────────────────┤

│ Vector DB │ Metadata DB │ File Storage │

│ (Chroma) │ (SQLite/PG) │ (Local/MinIO) │

└─────────────────────────────────────────────────────────────┘

**4. Implementation Modules**

**4.1 Document Processing Module**

python

class DocumentProcessor:

- pdf\_parser()

- text\_chunker()

- embedding\_generator()

- vector\_store\_manager()

- document\_indexer()

**4.2 ITSM Template Engine**

python

class ITSMTemplates:

- change\_management\_template()

- incident\_management\_template()

- problem\_management\_template()

- sop\_generator()

- environment\_adapter(on\_prem/cloud)

**4.3 AI Agent Framework**

python

class ITSMAgent:

- log\_monitor()

- metric\_analyzer()

- health\_checker()

- report\_generator()

- action\_recommender()

**4.4 Output Generation**

python

class OutputGenerator:

- markdown\_generator()

- html\_converter()

- powerpoint\_creator()

- dashboard\_builder()

- chart\_generator()

**5. Specific ITSM Capabilities**

**5.1 Change Management**

* Generate RFC templates based on environment type
* Risk assessment automation
* Approval workflow documentation
* Rollback procedure generation

**5.2 Incident/Problem Management**

* Incident classification and escalation procedures
* Root cause analysis templates
* Known error database integration
* Post-incident review automation

**5.3 Monitoring & Reporting**

* Automated health check reports
* Capacity planning analysis
* Performance trend identification
* SLA compliance tracking

**5.4 Security & Compliance**

* Vulnerability assessment reports
* Access review automation
* Compliance checklist generation
* Security incident procedures

**6. Challenges & Best Practices**

**6.1 Common Challenges**

**Data Quality & Consistency**

* Inconsistent document formats
* Incomplete or outdated procedures
* Data silos across departments

**LLM Limitations**

* Context window limitations for large documents
* Hallucination in technical details
* Consistency across similar tasks

**Performance Considerations**

* Vector search performance with large databases
* LLM inference time for complex queries
* Real-time monitoring data processing

**Integration Complexity**

* Connecting to diverse monitoring systems
* API rate limits and authentication
* Data format standardization

**6.2 Best Practices**

**Document Management**

* Implement version control for documents
* Use standardized naming conventions
* Regular document freshness validation
* Automated content overlap detection

**Prompt Engineering**

* Create domain-specific prompt templates
* Implement few-shot learning examples
* Use chain-of-thought for complex analysis
* Regular prompt performance evaluation

**System Design**

* Implement caching for frequent queries
* Use async processing for heavy tasks
* Design for horizontal scalability
* Implement comprehensive logging

**Security & Privacy**

* Data encryption at rest and in transit
* Role-based access control
* Audit trail for all AI interactions
* Regular security assessments

**Quality Assurance**

* Human-in-the-loop validation
* Output consistency checks
* Performance benchmarking
* Continuous model evaluation

**6.3 Deployment Recommendations**

**Development Environment**

* Use Docker containers for consistency
* Implement automated testing
* Version control for all components
* Staging environment for validation

**Production Considerations**

* Resource monitoring and alerting
* Backup and disaster recovery
* Performance optimization
* User training and documentation

**7. Development Roadmap**

**Phase 1: Foundation (Weeks 1-4)**

* Basic RAG pipeline with document ingestion
* LLM integration and basic querying
* Simple web interface for testing

**Phase 2: Core Features (Weeks 5-8)**

* ITSM template system implementation
* Basic agent framework for monitoring
* Output generation capabilities

**Phase 3: Advanced Features (Weeks 9-12)**

* Full agent automation
* Advanced analytics and reporting
* Dashboard and visualization

**Phase 4: Production Ready (Weeks 13-16)**

* Security hardening
* Performance optimization
* User training and documentation
* Production deployment

**8. Success Metrics**

* Reduction in manual documentation time
* Improved consistency across ITSM processes
* Faster incident resolution through better documentation
* Increased compliance with ITSM best practices
* Enhanced visibility into IT operations

**Next Steps**

1. **Environment Setup**: Install Ollama/LM Studio with chosen model
2. **Prototype Development**: Build basic RAG pipeline
3. **Template Creation**: Develop ITSM document templates
4. **Integration Planning**: Identify data sources and APIs
5. **User Interface Design**: Create wireframes for dashboard

I've created a comprehensive architectural design for your local AI-powered ITSM system. Now let me provide you with some additional insights and practical next steps.

## Key Recommendations for Getting Started

### 1. ****Start Small with MVP****

I'd recommend beginning with a basic RAG pipeline focused on one ITSM domain (like Incident Management) before expanding to the full system. This allows you to validate the approach and refine the prompts.

### 2. ****Model Selection Strategy****

For your use case, I'd suggest starting with **Mistral 7B Instruct** because:

* Excellent instruction following for structured outputs
* Good balance between quality and resource requirements
* Strong performance on technical documentation tasks
* Can run efficiently on modern consumer hardware

### 3. ****Critical Implementation Details****

**Document Chunking Strategy:**

* Use semantic chunking rather than fixed-size chunks
* Maintain document metadata (source, version, type)
* Implement overlap between chunks to preserve context

**Prompt Engineering for ITSM:**

* Create separate prompt templates for each ITSM process
* Include few-shot examples with actual ITIL-compliant outputs
* Use structured output formats (JSON, YAML) for consistency

**Agent Framework Design:**

* Start with simple scheduled tasks before implementing real-time monitoring
* Use event-driven architecture for scalability
* Implement circuit breakers for external API calls

### 4. ****Monitoring & Quality Assurance****

* Log all AI interactions with confidence scores
* Implement human review workflows for critical documents
* Track metrics like document generation time and user satisfaction
* Regular evaluation against ITIL standards

### 5. ****Security Considerations****

Since this handles company documents:

* Implement document access controls by user role
* Ensure no data leaves the local environment
* Regular security audits of the AI outputs
* Implement data retention policies

Would you like me to dive deeper into any specific component, such as creating a detailed implementation guide for the RAG pipeline, or would you prefer to see some sample code for getting started with the basic document ingestion system?

|  |
| --- |
| # main.py - FastAPI Backend for ITSM AI System  from fastapi import FastAPI, File, UploadFile, HTTPException, Depends, status  from fastapi.middleware.cors import CORSMiddleware  from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials  from pydantic import BaseModel  from typing import List, Optional, Dict, Any  import uvicorn  import os  import logging  from datetime import datetime  # Import our custom modules  from models.database import get\_db, Database  from services.document\_processor import DocumentProcessor  from services.llm\_service import LLMService  from services.rag\_service import RAGService  from services.itsm\_templates import ITSMTemplateService  from services.ai\_agent import ITSMAgent  from services.output\_generator import OutputGenerator  from utils.auth import verify\_token  from utils.config import settings  # Configure logging  logging.basicConfig(level=logging.INFO)  logger = logging.getLogger(\_\_name\_\_)  # Initialize FastAPI app  app = FastAPI(  title="ITSM AI System",  description="Local AI-powered ITSM documentation and monitoring system",  version="1.0.0"  )  # CORS middleware for frontend communication  app.add\_middleware(  CORSMiddleware,  allow\_origins=["http://localhost:3000"], # React dev server  allow\_credentials=True,  allow\_methods=["\*"],  allow\_headers=["\*"],  )  # Security scheme  security = HTTPBearer()  # Initialize services  document\_processor = DocumentProcessor()  llm\_service = LLMService()  rag\_service = RAGService()  itsm\_templates = ITSMTemplateService()  ai\_agent = ITSMAgent()  output\_generator = OutputGenerator()  # Pydantic models for API requests/responses  class DocumentUploadResponse(BaseModel):  document\_id: str  filename: str  status: str  chunks\_created: int  message: str  class QueryRequest(BaseModel):  query: str  document\_type: Optional[str] = None  max\_results: Optional[int] = 5  class QueryResponse(BaseModel):  answer: str  sources: List[Dict[str, Any]]  confidence\_score: float  class ITSMGenerateRequest(BaseModel):  template\_type: str # "change\_management", "incident\_management", etc.  environment: str # "on\_premises", "aws", "azure"  context: Optional[Dict[str, Any]] = None  custom\_requirements: Optional[str] = None  class ITSMGenerateResponse(BaseModel):  document\_content: str  template\_type: str  format: str  metadata: Dict[str, Any]  class AgentAnalysisRequest(BaseModel):  data\_source: str  analysis\_type: str  time\_range: Optional[str] = "1d"  class AgentAnalysisResponse(BaseModel):  analysis\_result: str  recommendations: List[str]  metrics: Dict[str, Any]  alerts: List[str]  # Dependency to get current user (simplified for demo)  async def get\_current\_user(credentials: HTTPAuthorizationCredentials = Depends(security)):  """  Verify JWT token and return user info.  In production, implement proper JWT validation.  """  try:  user = verify\_token(credentials.credentials)  return user  except Exception as e:  raise HTTPException(  status\_code=status.HTTP\_401\_UNAUTHORIZED,  detail="Invalid authentication credentials",  headers={"WWW-Authenticate": "Bearer"},  )  # Health check endpoint  @app.get("/health")  async def health\_check():  """  Basic health check to ensure all services are running.  """  try:  # Check LLM service  llm\_status = await llm\_service.health\_check()    # Check vector database  vector\_db\_status = rag\_service.health\_check()    # Check database connection  db\_status = Database().health\_check()    return {  "status": "healthy",  "timestamp": datetime.now().isoformat(),  "services": {  "llm": llm\_status,  "vector\_db": vector\_db\_status,  "database": db\_status  }  }  except Exception as e:  logger.error(f"Health check failed: {str(e)}")  raise HTTPException(status\_code=500, detail="Service unhealthy")  # Document Management Endpoints  @app.post("/api/documents/upload", response\_model=DocumentUploadResponse)  async def upload\_document(  file: UploadFile = File(...),  document\_type: str = "general",  current\_user: dict = Depends(get\_current\_user),  db: Database = Depends(get\_db)  ):  """  Upload and process a document for RAG integration.    Process:  1. Validate file type and size  2. Save file to local storage  3. Extract text content  4. Create semantic chunks  5. Generate embeddings  6. Store in vector database  7. Update metadata in SQL database  """  try:  # Validate file type  allowed\_types = ['.pdf', '.txt', '.docx', '.md']  file\_extension = os.path.splitext(file.filename)[1].lower()    if file\_extension not in allowed\_types:  raise HTTPException(  status\_code=400,  detail=f"File type {file\_extension} not supported. Allowed: {allowed\_types}"  )    # Save uploaded file  file\_path = await document\_processor.save\_uploaded\_file(file)    # Process document  result = await document\_processor.process\_document(  file\_path=file\_path,  document\_type=document\_type,  user\_id=current\_user['user\_id']  )    # Store metadata in database  document\_id = db.store\_document\_metadata(  filename=file.filename,  file\_path=file\_path,  document\_type=document\_type,  user\_id=current\_user['user\_id'],  chunks\_count=result['chunks\_created']  )    logger.info(f"Document {file.filename} processed successfully. ID: {document\_id}")    return DocumentUploadResponse(  document\_id=document\_id,  filename=file.filename,  status="processed",  chunks\_created=result['chunks\_created'],  message="Document uploaded and processed successfully"  )    except Exception as e:  logger.error(f"Document upload failed: {str(e)}")  raise HTTPException(status\_code=500, detail=f"Document processing failed: {str(e)}")  @app.get("/api/documents")  async def list\_documents(  current\_user: dict = Depends(get\_current\_user),  db: Database = Depends(get\_db)  ):  """  List all documents uploaded by the current user.  """  try:  documents = db.get\_user\_documents(current\_user['user\_id'])  return {"documents": documents}  except Exception as e:  logger.error(f"Failed to list documents: {str(e)}")  raise HTTPException(status\_code=500, detail="Failed to retrieve documents")  @app.delete("/api/documents/{document\_id}")  async def delete\_document(  document\_id: str,  current\_user: dict = Depends(get\_current\_user),  db: Database = Depends(get\_db)  ):  """  Delete a document and its associated chunks from the system.  """  try:  # Verify document ownership  document = db.get\_document\_by\_id(document\_id)  if not document or document['user\_id'] != current\_user['user\_id']:  raise HTTPException(status\_code=404, detail="Document not found")    # Delete from vector database  rag\_service.delete\_document\_chunks(document\_id)    # Delete file and metadata  await document\_processor.delete\_document(document\_id)  db.delete\_document(document\_id)    return {"message": "Document deleted successfully"}  except Exception as e:  logger.error(f"Failed to delete document: {str(e)}")  raise HTTPException(status\_code=500, detail="Failed to delete document")  # RAG Query Endpoints  @app.post("/api/query", response\_model=QueryResponse)  async def query\_documents(  request: QueryRequest,  current\_user: dict = Depends(get\_current\_user)  ):  """  Query the RAG system with natural language questions.    Process:  1. Generate query embedding  2. Search vector database for relevant chunks  3. Construct context-aware prompt  4. Generate response using LLM  5. Return answer with source citations  """  try:  # Search for relevant context  search\_results = rag\_service.search\_similar(  query=request.query,  max\_results=request.max\_results,  document\_type=request.document\_type,  user\_id=current\_user['user\_id']  )    if not search\_results:  return QueryResponse(  answer="I couldn't find relevant information in the uploaded documents to answer your question.",  sources=[],  confidence\_score=0.0  )    # Generate response using LLM  response = await llm\_service.generate\_response(  query=request.query,  context\_chunks=search\_results  )    # Extract source information  sources = [  {  "document\_name": chunk['metadata']['filename'],  "chunk\_text": chunk['content'][:200] + "...",  "relevance\_score": chunk['score']  }  for chunk in search\_results  ]    return QueryResponse(  answer=response['answer'],  sources=sources,  confidence\_score=response['confidence']  )    except Exception as e:  logger.error(f"Query processing failed: {str(e)}")  raise HTTPException(status\_code=500, detail=f"Query processing failed: {str(e)}")  # ITSM Template Generation Endpoints  @app.post("/api/itsm/generate", response\_model=ITSMGenerateResponse)  async def generate\_itsm\_document(  request: ITSMGenerateRequest,  current\_user: dict = Depends(get\_current\_user)  ):  """  Generate ITSM documentation based on templates and context.    Supported templates:  - change\_management: RFC templates, approval workflows  - incident\_management: Incident response procedures  - problem\_management: Root cause analysis templates  - sop\_technical: Technical SOPs for implementation/BAU  """  try:  # Validate template type  valid\_templates = [  "change\_management", "incident\_management",  "problem\_management", "sop\_technical", "capacity\_planning"  ]    if request.template\_type not in valid\_templates:  raise HTTPException(  status\_code=400,  detail=f"Invalid template type. Valid options: {valid\_templates}"  )    # Generate context-aware content  generated\_content = await itsm\_templates.generate\_document(  template\_type=request.template\_type,  environment=request.environment,  context=request.context or {},  custom\_requirements=request.custom\_requirements,  user\_context=current\_user  )    # Generate output in requested format  formatted\_content = output\_generator.format\_document(  content=generated\_content['content'],  format\_type="markdown", # Default to markdown  template\_type=request.template\_type  )    return ITSMGenerateResponse(  document\_content=formatted\_content,  template\_type=request.template\_type,  format="markdown",  metadata=generated\_content['metadata']  )    except Exception as e:  logger.error(f"ITSM document generation failed: {str(e)}")  raise HTTPException(status\_code=500, detail=f"Document generation failed: {str(e)}")  @app.get("/api/itsm/templates")  async def list\_itsm\_templates():  """  List available ITSM templates with their descriptions.  """  templates = itsm\_templates.list\_available\_templates()  return {"templates": templates}  # AI Agent Monitoring Endpoints  @app.post("/api/agent/analyze", response\_model=AgentAnalysisResponse)  async def run\_agent\_analysis(  request: AgentAnalysisRequest,  current\_user: dict = Depends(get\_current\_user)  ):  """  Run AI agent analysis on system data.    Analysis types:  - log\_analysis: Analyze system/application logs  - performance\_metrics: Evaluate performance data  - capacity\_planning: Assess resource utilization  - security\_audit: Security-focused analysis  """  try:  # Run the appropriate analysis  analysis\_result = await ai\_agent.run\_analysis(  data\_source=request.data\_source,  analysis\_type=request.analysis\_type,  time\_range=request.time\_range,  user\_context=current\_user  )    return AgentAnalysisResponse(  analysis\_result=analysis\_result['summary'],  recommendations=analysis\_result['recommendations'],  metrics=analysis\_result['metrics'],  alerts=analysis\_result['alerts']  )    except Exception as e:  logger.error(f"Agent analysis failed: {str(e)}")  raise HTTPException(status\_code=500, detail=f"Analysis failed: {str(e)}")  @app.get("/api/agent/status")  async def get\_agent\_status(current\_user: dict = Depends(get\_current\_user)):  """  Get current status of all AI agents.  """  try:  status = ai\_agent.get\_status()  return status  except Exception as e:  logger.error(f"Failed to get agent status: {str(e)}")  raise HTTPException(status\_code=500, detail="Failed to get agent status")  # Output Generation Endpoints  @app.post("/api/output/export")  async def export\_document(  content: str,  format\_type: str = "pdf",  template\_style: str = "professional",  current\_user: dict = Depends(get\_current\_user)  ):  """  Export generated content to various formats.    Supported formats:  - pdf: Professional PDF document  - pptx: PowerPoint presentation  - html: Web-ready HTML  - docx: Word document  """  try:  export\_result = output\_generator.export\_document(  content=content,  format\_type=format\_type,  template\_style=template\_style,  user\_id=current\_user['user\_id']  )    return {  "download\_url": export\_result['download\_url'],  "filename": export\_result['filename'],  "format": format\_type  }    except Exception as e:  logger.error(f"Document export failed: {str(e)}")  raise HTTPException(status\_code=500, detail=f"Export failed: {str(e)}")  # Analytics and Dashboard Endpoints  @app.get("/api/analytics/dashboard")  async def get\_dashboard\_data(current\_user: dict = Depends(get\_current\_user)):  """  Get dashboard analytics data.  """  try:  dashboard\_data = ai\_agent.get\_dashboard\_metrics()  return dashboard\_data  except Exception as e:  logger.error(f"Failed to get dashboard data: {str(e)}")  raise HTTPException(status\_code=500, detail="Failed to get dashboard data")  @app.get("/api/analytics/reports")  async def get\_generated\_reports(current\_user: dict = Depends(get\_current\_user)):  """  Get list of generated reports and their status.  """  try:  reports = ai\_agent.get\_generated\_reports(current\_user['user\_id'])  return {"reports": reports}  except Exception as e:  logger.error(f"Failed to get reports: {str(e)}")  raise HTTPException(status\_code=500, detail="Failed to get reports")  # WebSocket endpoint for real-time updates  @app.websocket("/ws")  async def websocket\_endpoint(websocket):  """  WebSocket connection for real-time updates.  Used for progress updates during document processing,  real-time agent alerts, and system notifications.  """  await websocket.accept()  try:  while True:  # Listen for messages and send updates  await ai\_agent.handle\_websocket\_connection(websocket)  except Exception as e:  logger.error(f"WebSocket error: {str(e)}")  finally:  await websocket.close()  # Startup event  @app.on\_event("startup")  async def startup\_event():  """  Initialize services on application startup.  """  logger.info("Starting ITSM AI System...")    # Initialize database  Database().initialize()    # Load LLM model  await llm\_service.initialize()    # Initialize vector database  rag\_service.initialize()    # Start background agents  ai\_agent.start\_background\_tasks()    logger.info("ITSM AI System started successfully!")  # Shutdown event  @app.on\_event("shutdown")  async def shutdown\_event():  """  Cleanup on application shutdown.  """  logger.info("Shutting down ITSM AI System...")    # Stop background tasks  ai\_agent.stop\_background\_tasks()    # Cleanup resources  llm\_service.cleanup()  rag\_service.cleanup()    logger.info("ITSM AI System shutdown complete.")  # Run the application  if \_\_name\_\_ == "\_\_main\_\_":  uvicorn.run(  "main:app",  host=settings.HOST,  port=settings.PORT,  reload=settings.DEBUG  ) |