**RAG Desktop Application - Phase-Wise Development Plan**

**Project Overview**

The RAG Desktop Application is a privacy-first, cross-platform desktop application that transforms user document collections into an intelligent, queryable knowledge base using Retrieval-Augmented Generation (RAG). It combines local processing with optional cloud enhancements, providing a seamless experience for document management, semantic search, and conversational AI. This phase-wise plan ensures efficient development while maintaining all functionality outlined in the original MVP specification.

**Core Value Proposition**

* **Personal AI Assistant**: Queryable document-based knowledge base
* **Hybrid Intelligence**: Local RAG with TAVILY web search fallback
* **Privacy-First**: Local document processing with optional cloud features
* **Cross-Platform**: Native support for Windows and macOS
* **Multi-User Support**: Secure user isolation via Google OAuth

**Technology Stack**

* **Backend**: FastAPI, PostgreSQL, Qdrant, Sentence Transformers
* **AI/ML**: Gemma3:1B-IT-QAT (via Ollama), all-MiniLM-L6-v2
* **Frontend**: PyQt6 with QSS styling
* **Authentication**: Google OAuth 2.0, JWT
* **Development/Deployment**: Docker Compose, PyInstaller, Cython, Pytest

**Phase-Wise Development Plan**

**PHASE 1: Complete RAG Pipeline + TAVILY Integration**

**Objective**: Build the core RAG pipeline, including document processing, embedding generation, vector storage, local LLM inference, and web search fallback, to create a fully functional backend intelligence system.

**Key Deliverables**:

* Document processing pipeline (PDF, DOCX, TXT, Markdown)
* Adaptive chunking and embedding generation
* Qdrant vector database integration for semantic search
* Gemma3:1B-IT-QAT integration via Ollama
* TAVILY web search fallback for enhanced responses
* PostgreSQL schema for user and document metadata
* Basic command-line testing interface

**File Structure**:

text

backend/ ├── config.py # Environment variables and settings ├── utils.py # File processing and chunking utilities ├── documents.py # Document ingestion and processing ├── rag.py # Qdrant integration and RAG pipeline ├── llm.py # Ollama + TAVILY integration ├── database.py # PostgreSQL models and queries └── main.py # FastAPI server (minimal for testing) scripts/ ├── setup\_dev.py # Docker Compose and environment setup └── init\_models.py # Model download (Sentence Transformers, Gemma3:1B) tests/ ├── test\_documents.py # Document processing tests ├── test\_rag.py # RAG pipeline tests ├── test\_llm.py # LLM and TAVILY integration tests

**Key Functionality**:

1. **Document Processing**:
   * Supported formats: PDF, DOCX, TXT, Markdown
   * Text extraction using format-specific parsers (e.g., PyPDF2, python-docx)
   * Adaptive chunking with semantic boundary preservation
   * Metadata extraction (file size, upload date, chunk count)
2. **Embedding Generation**:
   * Sentence Transformers (all-MiniLM-L6-v2) for high-quality embeddings
   * Batch processing with memory optimization
   * Storage in Qdrant with HNSW indexing
3. **RAG Pipeline**:
   * Query embedding and semantic search in Qdrant
   * Top-k context retrieval with relevance scoring
   * Context window management for LLM compatibility
4. **Local LLM Integration**:
   * Gemma3:1B-IT-QAT via Ollama for local inference
   * Streaming response generation
   * Response validation and formatting
5. **TAVILY Web Search Fallback**:
   * Automatic detection of insufficient local context
   * TAVILY API integration for web results
   * Context blending (local + web) with source attribution
6. **Database Setup**:
   * PostgreSQL schema for users, documents, and chunks
   * Basic user isolation logic
7. **Testing**:
   * Unit tests for document processing, chunking, and embeddings
   * Integration tests for RAG pipeline and TAVILY fallback
   * Command-line interface for manual testing

**Success Metrics**:

* Document processing time: < 30 seconds for average documents
* Query response time: < 2 seconds for semantic search
* Test coverage: > 80% for backend modules
* Successful TAVILY integration with 100% fallback accuracy

**PHASE 2: Complete API Layer**

**Objective**: Develop a comprehensive FastAPI-based REST API with Google OAuth authentication, user isolation, and all endpoints for document management, querying, and chat functionality.

**Key Deliverables**:

* Full REST API with OpenAPI documentation
* Google OAuth 2.0 authentication with JWT
* Complete user data isolation
* Endpoints for document management, RAG queries, and chat sessions
* Pydantic schemas for request/response validation

**File Structure**:

text

backend/ ├── api\_routes.py # All API endpoints ├── schemas.py # Pydantic models for validation ├── auth.py # Google OAuth + JWT handling ├── database.py # Extended database models └── main.py # Updated FastAPI server tests/ ├── test\_api.py # API endpoint tests ├── test\_auth.py # Authentication tests └── test\_database.py # Database interaction tests

**Key Functionality**:

1. **Authentication**:
   * Google OAuth 2.0 with PKCE flow
   * JWT token generation and refresh
   * Secure session storage and user profile management
2. **API Endpoints**:
   * **Auth**: /auth/google/login, /auth/callback, /auth/refresh
   * **Documents**: /upload, /documents, /documents/{id}, /delete
   * **RAG**: /query, /search/documents, /search/chunks
   * **Chat**: /chat/completions, /chat/query, /chat/history
   * **System**: /health, /models, /settings
3. **User Isolation**:
   * Database-level foreign key constraints
   * Qdrant user-specific collections
   * Session-based access control
4. **Validation**:
   * Pydantic schemas for all request/response models
   * Input sanitization and error handling
5. **Testing**:
   * Unit tests for authentication flows
   * Integration tests for all API endpoints
   * Stress tests for concurrent requests

**Success Metrics**:

* Authentication success rate: > 99% for OAuth flows
* API response time: < 500ms for non-RAG endpoints
* Test coverage: > 90% for API layer
* Complete user data isolation verified through tests

**PHASE 3: Complete UI/UX**

**Objective**: Build a modern, cross-platform PyQt6 desktop interface with system tray integration, drag-and-drop document upload, and an intuitive chat system.

**Key Deliverables**:

* PyQt6-based desktop application
* Dark-themed interface with QSS styling
* System tray integration and native OS features
* Drag-and-drop document upload with progress indicators
* Multi-session chat interface with streaming responses
* Offline/online mode switching

**File Structure**:

text

frontend/ ├── main.py # PyQt6 application entry point ├── main\_window.py # Main UI components and layout ├── api\_client.py # Backend API communication ├── session\_manager.py # Local session and chat history ├── styles.qss # Dark theme styling └── resources/ # Icons, images, and assets tests/ ├── test\_ui.py # UI component tests ├── test\_api\_client.py # API client integration tests

**Key Functionality**:

1. **Main Interface**:
   * Dark-themed, responsive layout
   * Document management panel with upload/delete
   * Chat interface with session management
   * Settings panel for model and connectivity options
2. **System Integration**:
   * System tray support for background operation
   * Native file dialogs and drag-and-drop
   * Window state persistence (size, position)
3. **Chat System**:
   * Multi-session support with history
   * Streaming responses with typing indicators
   * Markdown rendering for rich text
4. **Online/Offline Modes**:
   * Automatic connection detection
   * Visual indicators for mode status
   * Graceful degradation for offline use
5. **User Experience**:
   * Progressive loading with skeleton screens
   * Real-time feedback for uploads and queries
   * Accessible design (keyboard shortcuts, high contrast)
6. **Testing**:
   * UI component tests using PyQt test frameworks
   * Integration tests for API client
   * User flow tests for authentication and chat

**Success Metrics**:

* UI rendering time: < 1 second for main window
* Drag-and-drop success rate: 100% for supported formats
* Offline mode functionality: 100% core features available
* User satisfaction: Intuitive navigation (based on beta testing feedback)

**PHASE 4: Packaging & Optimization**

**Objective**: Package the application into cross-platform executables, optimize performance with Cython, and ensure production-ready stability with comprehensive testing.

**Key Deliverables**:

* Cross-platform executables for Windows and macOS
* Cython-optimized modules for performance
* Comprehensive test suite with > 90% coverage
* User-friendly installers with guided setup
* Performance benchmarks and documentation

**File Structure**:

text

deployment/ ├── build\_installer.py # PyInstaller automation script └── installer.spec # PyInstaller configuration scripts/ ├── cythonize.py # Cython optimization script └── performance\_test.py # Benchmarking and profiling tests/ ├── test\_rag.py # Updated RAG tests ├── test\_api.py # Updated API tests ├── test\_auth.py # Updated auth tests ├── test\_ui.py # Updated UI tests docs/ ├── user\_guide.md # User documentation └── api\_docs.md # API documentation

**Key Functionality**:

1. **Packaging**:
   * PyInstaller configuration for Windows and macOS
   * Asset bundling (models, dependencies, resources)
   * Code signing preparation
   * User-friendly installer generation
2. **Optimization**:
   * Cython optimization for document processing and embedding generation
   * Memory profiling and leak elimination
   * Startup optimization (< 5 seconds cold start)
3. **Testing**:
   * End-to-end tests for complete user flows
   * Performance benchmarks for RAG and API
   * Cross-platform compatibility tests
4. **Documentation**:
   * User guide for installation and usage
   * API documentation via FastAPI OpenAPI
   * Developer setup guide for future maintenance

**Success Metrics**:

* Application startup time: < 5 seconds
* Memory usage: < 2GB for typical operation
* Test coverage: > 90% across all modules
* Cross-platform compatibility: 100% feature parity
* Installer success rate: > 95% on first attempt

**Implementation Notes**

* **All Functionality Included**: Every feature from the original MVP (e.g., document ID system, user isolation, session management) is covered across the phases.
* **Gemma3:1B-IT-QAT**: Integrated in Phase 1 via init\_models.py for local LLM inference.
* **Cross-Platform Focus**: macOS and Windows compatibility ensured in all phases, with final validation in Phase 4.
* **Testing Emphasis**: Comprehensive test suites in each phase to maintain quality.
* **Future-Ready**: Architecture supports post-MVP enhancements (e.g., cloud sync, collaboration) without rework.

**Success Metrics & KPIs**

* **Technical**:
  + Query response: < 2 seconds
  + Document processing: < 30 seconds
  + Startup time: < 5 seconds
  + Memory usage: < 2GB
* **User Experience**:
  + Authentication success: > 99%
  + Offline functionality: 100% core features
  + Error recovery: Graceful handling of all scenarios
* **Development**:
  + Test coverage: > 90%
  + Documentation: Complete user and API guides
  + Code quality: Linting and type checking passed

**Conclusion**

This phase-wise plan streamlines the development of the RAG Desktop Application MVP by focusing on core intelligence first, followed by API, UI, and optimization. By consolidating the original 15 phases into four efficient stages, the plan reduces development overhead while ensuring all functionality is delivered. The application will provide a robust, privacy-first document intelligence solution with a modern user experience, ready for future cloud enhancements.