**Detailed Development Roadmap: File-to-Phase Mapping & Implementation Guide**

**Phase-to-File Mapping & Development Flow**

**Phase 1: Project Structure & Initialization**

**Files to Create:**

rag-desktop-app/

├── README.md # Project documentation & setup

├── .gitignore # Version control exclusions

├── requirements.txt # Python dependencies placeholder

├── .env.example # Environment template

└── scripts/setup\_dev.py # Development environment automation

**Actions:**

* Initialize Git repository
* Create basic directory structure
* Set up development environment script
* Document project goals and architecture

**Checkpoint 1.1: Repository Setup**

* [ ] Git repository initialized
* [ ] Basic folder structure created
* [ ] Initial documentation written
* [ ] Dependencies template ready

**Phase 2: Environment Setup & Dockerization**

**Files to Create/Update:**

├── docker-compose.yml # NEW: Service orchestration

├── deployment/

│ ├── Dockerfile.backend # NEW: FastAPI containerization

│ └── Dockerfile.qdrant # NEW: Qdrant setup

├── requirements.txt # UPDATE: Add FastAPI, SQLAlchemy, Qdrant client

└── .env.example # UPDATE: Add database URLs, API keys

**Dependencies:**

* requirements.txt → Docker files (package installation)
* .env.example → docker-compose.yml (environment variables)

**Checkpoint 2.1: Docker Environment**

* [ ] Docker Compose configuration complete
* [ ] Backend Dockerfile functional
* [ ] Qdrant service containerized
* [ ] Environment variables documented

**Checkpoint 2.2: Development Setup**

* [ ] Services start successfully with docker-compose up
* [ ] Database connections working
* [ ] Qdrant accessible via API
* [ ] All environment variables loading properly

**Phase 3: Document Uploading & Basic API Skeleton**

**Files to Create/Update:**

backend/

├── main.py # NEW: FastAPI app initialization

├── config.py # NEW: Environment configuration

├── api\_routes.py # NEW: Basic upload endpoints

├── schemas.py # NEW: Pydantic models

└── utils.py # NEW: File handling utilities

**API Endpoints to Implement:**

POST /upload # Document upload

GET /documents # List uploaded documents

GET /documents/{doc\_id} # Get specific document

DELETE /documents/{doc\_id} # Delete document

GET /health # Health check

**Dependencies:**

* config.py → main.py (configuration loading)
* schemas.py → api\_routes.py (request/response validation)
* utils.py → api\_routes.py (file processing)

**Checkpoint 3.1: FastAPI Foundation**

* [ ] FastAPI application starts successfully
* [ ] Configuration loading works
* [ ] Basic endpoints respond
* [ ] Health check functional

**Checkpoint 3.2: File Upload System**

* [ ] Document upload endpoint works
* [ ] File validation implemented
* [ ] Basic error handling in place
* [ ] API documentation auto-generated

**Phase 4: Document Chunking Logic**

**Files to Update:**

backend/

├── documents.py # NEW: Document processing logic

├── utils.py # UPDATE: Add chunking algorithms

└── requirements.txt # UPDATE: Add nltk, langchain, pypdf

**Functions to Add:**

# In documents.py

def chunk\_document(file\_path: str, chunk\_size: int = 1000) -> List[str]

def extract\_text\_from\_pdf(file\_path: str) -> str

def extract\_text\_from\_docx(file\_path: str) -> str

def adaptive\_chunking(text: str, min\_chunk: int, max\_chunk: int) -> List[str]

# In utils.py

def semantic\_chunking(text: str) -> List[str]

def preserve\_context(chunks: List[str], overlap: int = 100) -> List[str]

**Dependencies:**

* documents.py → api\_routes.py (processing pipeline)
* utils.py → documents.py (chunking algorithms)

**Checkpoint 4.1: Text Extraction**

* [ ] PDF text extraction working
* [ ] DOCX text extraction working
* [ ] TXT and Markdown support added
* [ ] File type detection implemented

**Checkpoint 4.2: Chunking Algorithms**

* [ ] Basic chunking functional
* [ ] Adaptive chunking implemented
* [ ] Context preservation working
* [ ] Chunk quality validation added

**Phase 5: Embedding Pipeline**

**Files to Update:**

backend/

├── documents.py # UPDATE: Add embedding generation

├── utils.py # UPDATE: Add embedding utilities

├── requirements.txt # UPDATE: Add sentence-transformers

└── scripts/init\_models.py # NEW: Model download script

**Functions to Add:**

# In documents.py

def generate\_embeddings(chunks: List[str]) -> List[List[float]]

def batch\_embed\_documents(chunks: List[str], batch\_size: int = 32) -> List[List[float]]

# In utils.py

def load\_embedding\_model(model\_name: str = "all-MiniLM-L6-v2")

def optimize\_embeddings\_gpu() -> bool

**Dependencies:**

* init\_models.py → documents.py (model availability)
* utils.py → documents.py (embedding functions)

**Checkpoint 5.1: Model Setup**

* [ ] Sentence transformers installed
* [ ] Model download script working
* [ ] GPU optimization configured (if available)
* [ ] Model loading tested

**Checkpoint 5.2: Embedding Generation**

* [ ] Single document embedding works
* [ ] Batch processing implemented
* [ ] Memory optimization in place
* [ ] Embedding quality validated

**Phase 6: Vector Storage Setup (Qdrant)**

**Files to Create/Update:**

backend/

├── rag.py # NEW: Qdrant integration

├── documents.py # UPDATE: Store embeddings in Qdrant

├── config.py # UPDATE: Add Qdrant connection settings

└── requirements.txt # UPDATE: Add qdrant-client

**Functions to Add:**

# In rag.py

def connect\_qdrant() -> QdrantClient

def create\_collection(collection\_name: str, vector\_size: int)

def upsert\_vectors(collection\_name: str, points: List[PointStruct])

def search\_similar\_vectors(collection\_name: str, query\_vector: List[float], limit: int = 10)

# In documents.py (updated)

def store\_document\_embeddings(doc\_id: str, chunks: List[str], embeddings: List[List[float]])

**Dependencies:**

* config.py → rag.py (connection configuration)
* rag.py → documents.py (vector storage)

**Checkpoint 6.1: Qdrant Connection**

* [ ] Qdrant client connection working
* [ ] Collection creation functional
* [ ] Basic vector operations tested
* [ ] Connection error handling implemented

**Checkpoint 6.2: Vector Storage Integration**

* [ ] Document embeddings stored successfully
* [ ] Vector search returning results
* [ ] Collection management working
* [ ] Data persistence verified

**Phase 7: API Layer (Core RAG Functions)**

**Files to Update:**

backend/

├── api\_routes.py # UPDATE: Add RAG endpoints

├── rag.py # UPDATE: Add retrieval logic

└── schemas.py # UPDATE: Add query/response models

**New API Endpoints:**

POST /query # Main RAG query endpoint

POST /search/documents # Semantic document search

GET /search/similar/{doc\_id} # Find similar documents

POST /search/chunks # Search within document chunks

**Functions to Add:**

# In rag.py

def semantic\_search(query: str, collection\_name: str, limit: int = 5) -> List[ScoredPoint]

def retrieve\_context(query: str, doc\_ids: List[str] = None) -> str

def rank\_results(results: List[ScoredPoint]) -> List[ScoredPoint]

**Dependencies:**

* rag.py → api\_routes.py (retrieval functions)
* schemas.py → api\_routes.py (request validation)

**Checkpoint 7.1: Search Endpoints**

* [ ] Semantic search endpoint working
* [ ] Document similarity search functional
* [ ] Chunk-level search implemented
* [ ] Result ranking operational

**Checkpoint 7.2: RAG Pipeline Foundation**

* [ ] Query processing pipeline complete
* [ ] Context retrieval working
* [ ] Result filtering and ranking active
* [ ] API documentation updated

**Phase 8: Query Handling and Local Vector Search**

**Files to Create/Update:**

backend/

├── llm.py # NEW: Ollama integration

├── rag.py # UPDATE: Add RAG pipeline

├── api\_routes.py # UPDATE: Add completion endpoints

└── requirements.txt # UPDATE: Add ollama, httpx

**New API Endpoints:**

POST /chat/completions # Stream LLM responses

POST /chat/query # RAG-enhanced query

GET /chat/history/{session\_id} # Conversation history

POST /chat/new # Start new conversation

**Functions to Add:**

# In llm.py

def connect\_ollama() -> httpx.Client

def generate\_response(prompt: str, context: str, model: str = "gemma:3") -> str

def stream\_response(prompt: str, context: str) -> Iterator[str]

# In rag.py (updated)

def rag\_pipeline(query: str, session\_id: str = None) -> str

def build\_context\_prompt(query: str, retrieved\_docs: List[str]) -> str

**Dependencies:**

* llm.py → rag.py (response generation)
* rag.py → api\_routes.py (complete pipeline)

**Checkpoint 8.1: Ollama Integration**

* [ ] Ollama connection established
* [ ] Model downloading working
* [ ] Basic text generation functional
* [ ] Streaming responses implemented

**Checkpoint 8.2: Complete RAG Pipeline**

* [ ] End-to-end RAG queries working
* [ ] Context integration with LLM
* [ ] Response streaming operational
* [ ] Chat session management active

**Phase 9: TAVILY Web Search Fallback**

**Files to Update:**

backend/

├── llm.py # UPDATE: Add fallback logic

├── config.py # UPDATE: Add TAVILY API key

├── utils.py # UPDATE: Add web search utilities

└── requirements.txt # UPDATE: Add tavily-python

**Functions to Add:**

# In llm.py (updated)

def should\_use\_fallback(local\_results: List[ScoredPoint], confidence\_threshold: float = 0.7) -> bool

def web\_search\_fallback(query: str) -> List[str]

def blend\_local\_and\_web\_results(local\_results: str, web\_results: str) -> str

# In utils.py (updated)

def extract\_web\_content(url: str) -> str

def summarize\_web\_results(results: List[str]) -> str

**Dependencies:**

* config.py → llm.py (API configuration)
* utils.py → llm.py (web content processing)

**Checkpoint 9.1: Web Search Integration**

* [ ] TAVILY API connection working
* [ ] Web search functionality active
* [ ] Content extraction operational
* [ ] Search result processing complete

**Checkpoint 9.2: Fallback Logic**

* [ ] Fallback decision logic working
* [ ] Local and web result blending
* [ ] Quality assessment implemented
* [ ] Fallback performance optimized

**Phase 10: Metadata Storage**

**Files to Create/Update:**

backend/

├── database.py # NEW: SQLAlchemy models & connection

├── documents.py # UPDATE: Add metadata storage

├── api\_routes.py # UPDATE: Add metadata endpoints

└── requirements.txt # UPDATE: Add sqlalchemy, psycopg2

**Database Models to Create:**

# In database.py

class Document(Base):

# id, title, file\_path, upload\_time, owner\_id, file\_type, size

class DocumentChunk(Base):

# id, document\_id, chunk\_text, chunk\_index, embedding\_id, metadata

class User(Base):

# id, google\_id, email, name, created\_at, last\_login

class ChatSession(Base):

# id, user\_id, created\_at, updated\_at, session\_data

**New API Endpoints:**

GET /documents/{doc\_id}/metadata # Document metadata

GET /documents/{doc\_id}/chunks # Document chunks

POST /documents/{doc\_id}/reprocess # Reprocess document

GET /users/me/documents # User's documents

**Dependencies:**

* database.py → documents.py (data persistence)
* database.py → api\_routes.py (CRUD operations)

**Checkpoint 10.1: Database Setup**

* [ ] PostgreSQL connection working
* [ ] Database models defined
* [ ] Migrations system active
* [ ] CRUD operations functional

**Checkpoint 10.2: Metadata Integration**

* [ ] Document metadata storage working
* [ ] Chunk metadata persistence active
* [ ] User data management operational
* [ ] Database queries optimized

**Phase 11: UI/UX Development with PyQt6 + QSS**

**Files to Create:**

frontend/

├── main.py # NEW: PyQt6 application entry

├── main\_window.py # NEW: Main UI components

├── api\_client.py # NEW: Backend communication

├── session\_manager.py # NEW: Local session handling

├── styles.qss # NEW: UI styling

└── resources/

├── icons/app.ico # NEW: Application icons

└── app.ico # NEW: Main app icon

**UI Components to Implement:**

# In main\_window.py

class MainWindow(QMainWindow):

def setup\_ui()

def setup\_chat\_panel()

def setup\_document\_panel()

def setup\_menu\_bar()

class ChatWidget(QWidget):

def send\_message()

def display\_response()

def handle\_streaming()

class DocumentWidget(QWidget):

def upload\_document()

def list\_documents()

def delete\_document()

**Dependencies:**

* api\_client.py → main\_window.py (backend communication)
* session\_manager.py → main\_window.py (state persistence)
* styles.qss → main.py (UI theming)

**Checkpoint 11.1: Basic UI Framework**

* [ ] PyQt6 application starts
* [ ] Main window structure complete
* [ ] Basic styling applied
* [ ] Resource loading working

**Checkpoint 11.2: Core UI Components**

* [ ] Chat interface functional
* [ ] Document management UI working
* [ ] File upload drag-drop active
* [ ] Settings panel operational

**Checkpoint 11.3: Backend Integration**

* [ ] API client communication working
* [ ] Real-time chat updates
* [ ] Document operations integrated
* [ ] Error handling in UI

**Phase 12: Session Management & Google OAuth**

**Files to Update:**

backend/

├── auth.py # NEW: OAuth implementation

├── api\_routes.py # UPDATE: Add auth endpoints

├── database.py # UPDATE: Add user models

├── config.py # UPDATE: Add OAuth settings

frontend/

├── main\_window.py # UPDATE: Add login UI

├── api\_client.py # UPDATE: Add auth handling

└── session\_manager.py # UPDATE: Add user sessions

**New API Endpoints:**

POST /auth/google/login # Initiate OAuth flow

GET /auth/google/callback # Handle OAuth callback

POST /auth/refresh # Refresh JWT token

POST /auth/logout # Invalidate session

GET /auth/profile # Get user profile

**Functions to Add:**

# In auth.py

def create\_google\_oauth\_url() -> str

def handle\_oauth\_callback(code: str) -> dict

def create\_jwt\_token(user\_id: str) -> str

def verify\_jwt\_token(token: str) -> dict

# In session\_manager.py (frontend)

def save\_session(user\_data: dict)

def load\_session() -> dict

def clear\_session()

**Dependencies:**

* auth.py → api\_routes.py (authentication endpoints)
* api\_client.py → auth.py (token management)

**Checkpoint 12.1: OAuth Implementation**

* [ ] Google OAuth flow working
* [ ] JWT token generation/validation
* [ ] User registration/login functional
* [ ] Session persistence active

**Checkpoint 12.2: Frontend Authentication**

* [ ] Login UI integrated
* [ ] Session management working
* [ ] Auto-login on startup
* [ ] Logout functionality complete

**Phase 13: System Tray + Background Session Handling**

**Files to Update:**

frontend/

├── main.py # UPDATE: Add system tray

├── main\_window.py # UPDATE: Add minimize to tray

└── session\_manager.py # UPDATE: Add background persistence

**Functions to Add:**

# In main.py (updated)

def create\_system\_tray() -> QSystemTrayIcon

def setup\_tray\_menu() -> QMenu

def handle\_tray\_activation(reason: QSystemTrayIcon.ActivationReason)

# In main\_window.py (updated)

def closeEvent(event: QCloseEvent)

def minimize\_to\_tray()

def restore\_from\_tray()

# In session\_manager.py (updated)

def background\_sync()

def save\_window\_state()

def restore\_window\_state()

**Dependencies:**

* System tray → main.py (application lifecycle)
* Window state → session\_manager.py (persistence)

**Checkpoint 13.1: System Tray Integration**

* [ ] System tray icon working
* [ ] Tray menu functional
* [ ] Window minimize/restore working
* [ ] Notifications operational

**Checkpoint 13.2: Background Operations**

* [ ] Background sync working
* [ ] Session persistence active
* [ ] Auto-startup configuration
* [ ] Graceful shutdown handling

**Phase 14: Application Packaging**

**Files to Create:**

deployment/

├── build\_installer.py # NEW: PyInstaller automation

└── deploy.sh # UPDATE: Add packaging steps

requirements.txt # UPDATE: Add PyInstaller

**Functions to Add:**

# In build\_installer.py

def build\_windows\_exe()

def build\_macos\_dmg()

def copy\_resources()

def create\_installer()

**Dependencies:**

* All frontend files → build\_installer.py (packaging source)
* resources/ → Build output (asset bundling)

**Checkpoint 14.1: Build System**

* [ ] PyInstaller configuration complete
* [ ] Resource bundling working
* [ ] Cross-platform builds functional
* [ ] Dependency resolution active

**Checkpoint 14.2: Distribution**

* [ ] Windows installer created
* [ ] macOS DMG generated
* [ ] Installation testing complete
* [ ] Update mechanism implemented

**Phase 15: Final Testing, Optimization & MVP Release**

**Files to Create/Update:**

tests/

├── test\_auth.py # NEW: Authentication tests

├── test\_rag.py # NEW: RAG pipeline tests

└── test\_api.py # NEW: API endpoint tests

scripts/

├── cythonize.py # NEW: Performance optimization

└── deploy.sh # UPDATE: Final deployment

README.md # UPDATE: Complete documentation

**Checkpoint 15.1: Testing Suite**

* [ ] Unit tests for all components
* [ ] Integration tests working
* [ ] Performance benchmarks complete
* [ ] Security audit passed

**Checkpoint 15.2: Optimization**

* [ ] Performance bottlenecks identified
* [ ] Cythonization applied where beneficial
* [ ] Memory usage optimized
* [ ] Startup time minimized

**Checkpoint 15.3: MVP Release**

* [ ] Complete documentation written
* [ ] User manual created
* [ ] Installation guides complete
* [ ] Release package ready

**Complete API Reference**

**Authentication APIs**

POST /auth/google/login # Initiate Google OAuth

GET /auth/google/callback # OAuth callback handler

POST /auth/refresh # Refresh JWT tokens

POST /auth/logout # Invalidate sessions

GET /auth/profile # User profile data

**Document Management APIs**

POST /upload # Upload documents

GET /documents # List user documents

GET /documents/{doc\_id} # Get document details

DELETE /documents/{doc\_id} # Delete documents

POST /documents/{doc\_id}/reprocess # Reprocess documents

GET /documents/{doc\_id}/metadata # Document metadata

GET /documents/{doc\_id}/chunks # Document chunks

**RAG & Search APIs**

POST /query # Main RAG query

POST /search/documents # Semantic search

POST /search/chunks # Search chunks

GET /search/similar/{doc\_id} # Similar documents

**Chat & Conversation APIs**

POST /chat/completions # Stream completions

POST /chat/query # RAG-enhanced chat

GET /chat/history/{session\_id} # Chat history

POST /chat/new # New conversation

DELETE /chat/{session\_id} # Delete conversation

**System APIs**

GET /health # System health

GET /system/models # Available models

POST /system/settings # Update settings

GET /system/usage # Usage statistics

**Critical Development Flow**

**Sequential Dependencies:**

1. **Phase 1-2**: Foundation (structure + containers)
2. **Phase 3-6**: Backend Core (API + storage)
3. **Phase 7-9**: RAG Pipeline (retrieval + generation)
4. **Phase 10**: Data Layer (PostgreSQL integration)
5. **Phase 11**: Frontend (PyQt6 UI)
6. **Phase 12**: Authentication (OAuth integration)
7. **Phase 13**: System Integration (tray + background)
8. **Phase 14-15**: Packaging & Optimization

**Parallel Development Opportunities:**

* **Phases 4-5**: Chunking + Embedding (can be developed together)
* **Phases 7-8**: RAG + LLM integration (interdependent)
* **Phases 11-12**: UI + Auth (can be developed in parallel after backend is stable)

**Major Milestones:**

* **Milestone 1 (End Phase 6)**: Complete backend RAG pipeline working
* **Milestone 2 (End Phase 10)**: Full-featured API with database persistence
* **Milestone 3 (End Phase 13)**: Complete desktop application with all features
* **Milestone 4 (End Phase 15)**: Production-ready packaged application

This roadmap ensures systematic development with clear dependencies and measurable milestones for each phase.