**Complete Functions & Classes Specification for RAG Desktop Application**

**Backend Files**

**backend/main.py [FastAPI Application Entry]**

# Classes

class AppConfig:

def \_\_init\_\_(self)

# Functions

def create\_app() -> FastAPI

def setup\_middleware(app: FastAPI)

def setup\_cors(app: FastAPI)

def setup\_exception\_handlers(app: FastAPI)

def lifespan(app: FastAPI)

def get\_app\_info() -> dict

# Startup/Shutdown Events

async def startup\_event()

async def shutdown\_event()

**backend/config.py [Environment Configuration]**

# Classes

class Settings(BaseSettings):

# Database settings

database\_url: str

postgres\_host: str

postgres\_port: int

postgres\_db: str

postgres\_user: str

postgres\_password: str

# Qdrant settings

qdrant\_url: str

qdrant\_api\_key: Optional[str]

# OAuth settings

google\_client\_id: str

google\_client\_secret: str

google\_redirect\_uri: str

# JWT settings

jwt\_secret\_key: str

jwt\_algorithm: str

jwt\_expire\_minutes: int

# ML model settings

embedding\_model: str

ollama\_url: str

ollama\_model: str

# TAVILY settings

tavily\_api\_key: str

# App settings

app\_name: str

app\_version: str

debug: bool

# Functions

def get\_settings() -> Settings

def validate\_settings(settings: Settings) -> bool

**backend/api\_routes.py [REST API Endpoints]**

# Authentication Routes

async def google\_login()

async def google\_callback(code: str)

async def refresh\_token(refresh\_token: str)

async def logout(token: str)

async def get\_profile(current\_user: User)

# Document Management Routes

async def upload\_document(file: UploadFile, current\_user: User)

async def list\_documents(current\_user: User, skip: int = 0, limit: int = 10)

async def get\_document(doc\_id: str, current\_user: User)

async def delete\_document(doc\_id: str, current\_user: User)

async def reprocess\_document(doc\_id: str, current\_user: User)

async def get\_document\_chunks(doc\_id: str, current\_user: User)

async def get\_document\_metadata(doc\_id: str, current\_user: User)

# RAG & Search Routes

async def rag\_query(query: QueryRequest, current\_user: User)

async def semantic\_search(search\_request: SearchRequest, current\_user: User)

async def search\_chunks(chunk\_request: ChunkSearchRequest, current\_user: User)

async def find\_similar\_documents(doc\_id: str, current\_user: User)

# Chat System Routes

async def stream\_completion(chat\_request: ChatRequest, current\_user: User)

async def rag\_chat(chat\_request: RagChatRequest, current\_user: User)

async def get\_chat\_history(session\_id: str, current\_user: User)

async def create\_new\_chat(current\_user: User)

async def delete\_chat(session\_id: str, current\_user: User)

# System Routes

async def health\_check()

async def get\_available\_models()

async def update\_system\_settings(settings: SystemSettings, current\_user: User)

async def get\_usage\_stats(current\_user: User)

**backend/schemas.py [Pydantic Data Models]**

# Base Classes

class BaseModel(PydanticBaseModel):

class Config:

from\_attributes = True

# User Schemas

class UserBase(BaseModel):

email: str

name: str

class UserCreate(UserBase):

google\_id: str

class UserResponse(UserBase):

id: str

created\_at: datetime

last\_login: Optional[datetime]

# Document Schemas

class DocumentBase(BaseModel):

title: str

file\_type: str

class DocumentCreate(DocumentBase):

file\_path: str

size: int

class DocumentResponse(DocumentBase):

id: str

upload\_time: datetime

chunk\_count: int

processing\_status: str

class DocumentChunk(BaseModel):

id: str

chunk\_text: str

chunk\_index: int

metadata: dict

# Query Schemas

class QueryRequest(BaseModel):

query: str

max\_results: int = 5

include\_chunks: bool = True

class SearchRequest(BaseModel):

query: str

document\_ids: Optional[List[str]] = None

limit: int = 10

class ChunkSearchRequest(BaseModel):

query: str

document\_id: str

limit: int = 5

# Chat Schemas

class ChatMessage(BaseModel):

role: str # "user" or "assistant"

content: str

timestamp: datetime

class ChatRequest(BaseModel):

message: str

session\_id: Optional[str] = None

stream: bool = True

class RagChatRequest(ChatRequest):

use\_fallback: bool = True

max\_context\_length: int = 2000

class ChatSession(BaseModel):

id: str

user\_id: str

created\_at: datetime

messages: List[ChatMessage]

# Authentication Schemas

class TokenResponse(BaseModel):

access\_token: str

refresh\_token: str

token\_type: str

expires\_in: int

class GoogleAuthRequest(BaseModel):

code: str

state: Optional[str] = None

# System Schemas

class SystemSettings(BaseModel):

embedding\_model: Optional[str]

chunk\_size: Optional[int]

chunk\_overlap: Optional[int]

max\_retrieval\_results: Optional[int]

class HealthResponse(BaseModel):

status: str

database: str

qdrant: str

ollama: str

timestamp: datetime

**backend/utils.py [Utility Functions]**

# File Handling Functions

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

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

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

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

def detect\_file\_type(file\_path: str) -> str

def validate\_file\_size(file\_path: str, max\_size\_mb: int = 50) -> bool

# Text Processing Functions

def clean\_text(text: str) -> str

def normalize\_whitespace(text: str) -> str

def remove\_special\_characters(text: str, keep\_punctuation: bool = True) -> str

def split\_into\_sentences(text: str) -> List[str]

def split\_into\_paragraphs(text: str) -> List[str]

# Chunking Algorithms

def adaptive\_chunk\_text(text: str, min\_size: int = 800, max\_size: int = 1200, overlap: int = 100) -> List[str]

def semantic\_chunk\_by\_sentences(text: str, target\_size: int = 1000) -> List[str]

def chunk\_by\_paragraphs(text: str, max\_size: int = 1500) -> List[str]

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

# Embedding Utilities

def batch\_texts(texts: List[str], batch\_size: int = 32) -> List[List[str]]

def optimize\_embedding\_batch\_size(text\_lengths: List[int]) -> int

def calculate\_embedding\_similarity(vec1: List[float], vec2: List[float]) -> float

# Security Functions

def generate\_secure\_id() -> str

def hash\_password(password: str) -> str

def verify\_password(password: str, hashed: str) -> bool

def generate\_api\_key() -> str

# Date/Time Utilities

def get\_utc\_now() -> datetime

def format\_timestamp(dt: datetime) -> str

def parse\_timestamp(timestamp\_str: str) -> datetime

# Validation Functions

def validate\_email(email: str) -> bool

def validate\_url(url: str) -> bool

def sanitize\_filename(filename: str) -> str

**backend/documents.py [Document Processing Pipeline]**

# Classes

class DocumentProcessor:

def \_\_init\_\_(self, embedding\_model: str)

def process\_document(self, file\_path: str, user\_id: str) -> DocumentResponse

def reprocess\_document(self, doc\_id: str) -> DocumentResponse

def delete\_document(self, doc\_id: str) -> bool

class ChunkingEngine:

def \_\_init\_\_(self, strategy: str = "adaptive")

def chunk\_document(self, text: str, metadata: dict) -> List[DocumentChunk]

def get\_optimal\_chunk\_size(self, text: str) -> int

class EmbeddingEngine:

def \_\_init\_\_(self, model\_name: str)

def load\_model(self)

def embed\_texts(self, texts: List[str]) -> List[List[float]]

def embed\_single\_text(self, text: str) -> List[float]

def get\_embedding\_dimension(self) -> int

# Document Processing Functions

async def upload\_and\_process(file: UploadFile, user\_id: str) -> DocumentResponse

async def extract\_document\_text(file\_path: str) -> str

async def chunk\_and\_embed\_document(text: str, doc\_id: str) -> List[DocumentChunk]

async def store\_document\_metadata(doc\_data: DocumentCreate, user\_id: str) -> str

async def update\_processing\_status(doc\_id: str, status: str)

# Document Management Functions

async def get\_user\_documents(user\_id: str, skip: int = 0, limit: int = 10) -> List[DocumentResponse]

async def get\_document\_by\_id(doc\_id: str, user\_id: str) -> Optional[DocumentResponse]

async def get\_document\_chunks\_paginated(doc\_id: str, page: int = 1, size: int = 10) -> List[DocumentChunk]

async def delete\_document\_and\_embeddings(doc\_id: str) -> bool

# File Format Handlers

def handle\_pdf\_upload(file\_path: str) -> str

def handle\_docx\_upload(file\_path: str) -> str

def handle\_txt\_upload(file\_path: str) -> str

def handle\_markdown\_upload(file\_path: str) -> str

# Chunk Processing Functions

def create\_chunk\_metadata(doc\_id: str, chunk\_index: int, source\_info: dict) -> dict

def validate\_chunk\_quality(chunk\_text: str) -> bool

def merge\_overlapping\_chunks(chunks: List[str]) -> List[str]

**backend/rag.py [Qdrant Integration & RAG Pipeline]**

# Classes

class QdrantManager:

def \_\_init\_\_(self, url: str, api\_key: Optional[str])

def connect(self) -> QdrantClient

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

def delete\_collection(self, collection\_name: str)

def collection\_exists(self, collection\_name: str) -> bool

class VectorStore:

def \_\_init\_\_(self, qdrant\_manager: QdrantManager)

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

def search\_vectors(self, collection: str, query\_vector: List[float], limit: int) -> List[ScoredPoint]

def delete\_vectors(self, collection: str, point\_ids: List[str])

def get\_collection\_info(self, collection: str) -> dict

class RAGPipeline:

def \_\_init\_\_(self, vector\_store: VectorStore, embedding\_engine: EmbeddingEngine)

def retrieve\_context(self, query: str, user\_id: str, max\_results: int = 5) -> str

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

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

# Vector Operations

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

async def search\_similar\_chunks(query: str, user\_id: str, limit: int = 10) -> List[ScoredPoint]

async def semantic\_search\_documents(query: str, document\_ids: Optional[List[str]], user\_id: str) -> List[DocumentResponse]

async def find\_similar\_documents\_by\_id(doc\_id: str, user\_id: str, limit: int = 5) -> List[DocumentResponse]

# Context Retrieval Functions

async def retrieve\_relevant\_context(query: str, user\_id: str, max\_context\_length: int = 2000) -> str

async def get\_context\_from\_chunks(chunks: List[ScoredPoint]) -> str

async def filter\_results\_by\_relevance(results: List[ScoredPoint], threshold: float = 0.7) -> List[ScoredPoint]

# Collection Management

async def initialize\_user\_collection(user\_id: str)

async def cleanup\_user\_data(user\_id: str)

async def get\_collection\_stats(collection\_name: str) -> dict

# Search Optimization Functions

def optimize\_search\_params(query\_length: int, collection\_size: int) -> dict

def calculate\_relevance\_score(result: ScoredPoint, query\_context: str) -> float

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

**backend/llm.py [LLM Integration & Generation]**

# Classes

class OllamaClient:

def \_\_init\_\_(self, base\_url: str, model\_name: str)

def connect(self) -> httpx.Client

def is\_model\_available(self) -> bool

def download\_model(self) -> bool

def get\_model\_info(self) -> dict

class ResponseGenerator:

def \_\_init\_\_(self, ollama\_client: OllamaClient)

def generate\_response(self, prompt: str, context: str) -> str

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

def generate\_with\_fallback(self, prompt: str, context: str, use\_web: bool = True) -> str

class TAVILYSearcher:

def \_\_init\_\_(self, api\_key: str)

def search\_web(self, query: str, max\_results: int = 5) -> List[dict]

def extract\_content(self, search\_results: List[dict]) -> str

def summarize\_web\_results(self, content: str) -> str

# LLM Generation Functions

async def generate\_rag\_response(query: str, context: str, stream: bool = False) -> Union[str, Iterator[str]]

async def generate\_chat\_response(message: str, chat\_history: List[ChatMessage], context: str = "") -> str

async def stream\_chat\_completion(message: str, context: str) -> AsyncIterator[str]

# Fallback & Web Search Functions

async def determine\_fallback\_need(local\_results: List[ScoredPoint], query: str) -> bool

async def web\_search\_fallback(query: str) -> str

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

async def evaluate\_response\_quality(response: str, query: str) -> float

# Prompt Engineering Functions

def build\_rag\_prompt(query: str, context: str, chat\_history: Optional[List[ChatMessage]] = None) -> str

def build\_system\_prompt() -> str

def format\_chat\_history(messages: List[ChatMessage]) -> str

def create\_context\_prompt(retrieved\_chunks: List[str]) -> str

# Model Management Functions

async def ensure\_model\_loaded(model\_name: str) -> bool

async def get\_available\_models() -> List[str]

async def estimate\_token\_count(text: str) -> int

async def truncate\_context(context: str, max\_tokens: int = 2000) -> str

# Response Processing Functions

def clean\_llm\_response(response: str) -> str

def extract\_json\_from\_response(response: str) -> Optional[dict]

def validate\_response\_completeness(response: str, query: str) -> bool

**backend/database.py [PostgreSQL Models & Connection]**

# Base Classes

class Base(DeclarativeBase):

pass

# Database Models

class User(Base):

\_\_tablename\_\_ = "users"

id: Mapped[str] = mapped\_column(String, primary\_key=True)

google\_id: Mapped[str] = mapped\_column(String, unique=True, index=True)

email: Mapped[str] = mapped\_column(String, unique=True, index=True)

name: Mapped[str] = mapped\_column(String)

created\_at: Mapped[datetime] = mapped\_column(DateTime, default=func.now())

last\_login: Mapped[Optional[datetime]] = mapped\_column(DateTime)

# Relationships

documents: Mapped[List["Document"]] = relationship(back\_populates="owner")

chat\_sessions: Mapped[List["ChatSession"]] = relationship(back\_populates="user")

class Document(Base):

\_\_tablename\_\_ = "documents"

id: Mapped[str] = mapped\_column(String, primary\_key=True)

title: Mapped[str] = mapped\_column(String)

file\_path: Mapped[str] = mapped\_column(String)

file\_type: Mapped[str] = mapped\_column(String)

size: Mapped[int] = mapped\_column(Integer)

upload\_time: Mapped[datetime] = mapped\_column(DateTime, default=func.now())

processing\_status: Mapped[str] = mapped\_column(String, default="pending")

chunk\_count: Mapped[int] = mapped\_column(Integer, default=0)

owner\_id: Mapped[str] = mapped\_column(String, ForeignKey("users.id"))

# Relationships

owner: Mapped["User"] = relationship(back\_populates="documents")

chunks: Mapped[List["DocumentChunk"]] = relationship(back\_populates="document")

class DocumentChunk(Base):

\_\_tablename\_\_ = "document\_chunks"

id: Mapped[str] = mapped\_column(String, primary\_key=True)

document\_id: Mapped[str] = mapped\_column(String, ForeignKey("documents.id"))

chunk\_text: Mapped[str] = mapped\_column(Text)

chunk\_index: Mapped[int] = mapped\_column(Integer)

embedding\_id: Mapped[str] = mapped\_column(String) # Qdrant point ID

metadata: Mapped[dict] = mapped\_column(JSON)

created\_at: Mapped[datetime] = mapped\_column(DateTime, default=func.now())

# Relationships

document: Mapped["Document"] = relationship(back\_populates="chunks")

class ChatSession(Base):

\_\_tablename\_\_ = "chat\_sessions"

id: Mapped[str] = mapped\_column(String, primary\_key=True)

user\_id: Mapped[str] = mapped\_column(String, ForeignKey("users.id"))

created\_at: Mapped[datetime] = mapped\_column(DateTime, default=func.now())

updated\_at: Mapped[datetime] = mapped\_column(DateTime, default=func.now(), onupdate=func.now())

session\_data: Mapped[dict] = mapped\_column(JSON, default=dict)

# Relationships

user: Mapped["User"] = relationship(back\_populates="chat\_sessions")

messages: Mapped[List["ChatMessage"]] = relationship(back\_populates="session")

class ChatMessage(Base):

\_\_tablename\_\_ = "chat\_messages"

id: Mapped[str] = mapped\_column(String, primary\_key=True)

session\_id: Mapped[str] = mapped\_column(String, ForeignKey("chat\_sessions.id"))

role: Mapped[str] = mapped\_column(String) # "user" or "assistant"

content: Mapped[str] = mapped\_column(Text)

timestamp: Mapped[datetime] = mapped\_column(DateTime, default=func.now())

metadata: Mapped[Optional[dict]] = mapped\_column(JSON)

# Relationships

session: Mapped["ChatSession"] = relationship(back\_populates="messages")

# Database Connection & Session Management

class DatabaseManager:

def \_\_init\_\_(self, database\_url: str)

def get\_engine(self) -> Engine

def create\_tables(self)

def get\_session(self) -> Session

# Database Functions

async def get\_db\_session() -> AsyncGenerator[AsyncSession, None]

async def create\_database\_tables()

async def drop\_database\_tables()

# User CRUD Operations

async def create\_user(user\_data: UserCreate) -> User

async def get\_user\_by\_id(user\_id: str) -> Optional[User]

async def get\_user\_by\_google\_id(google\_id: str) -> Optional[User]

async def get\_user\_by\_email(email: str) -> Optional[User]

async def update\_user\_last\_login(user\_id: str) -> User

# Document CRUD Operations

async def create\_document(doc\_data: DocumentCreate, user\_id: str) -> Document

async def get\_document\_by\_id(doc\_id: str) -> Optional[Document]

async def get\_user\_documents\_paginated(user\_id: str, skip: int, limit: int) -> List[Document]

async def update\_document\_status(doc\_id: str, status: str) -> Document

async def delete\_document\_by\_id(doc\_id: str) -> bool

# Chat CRUD Operations

async def create\_chat\_session(user\_id: str) -> ChatSession

async def get\_chat\_session(session\_id: str) -> Optional[ChatSession]

async def add\_chat\_message(session\_id: str, role: str, content: str) -> ChatMessage

async def get\_chat\_history(session\_id: str, limit: int = 50) -> List[ChatMessage]

async def delete\_chat\_session(session\_id: str) -> bool

**backend/auth.py [Authentication & Authorization]**

# Classes

class GoogleOAuthManager:

def \_\_init\_\_(self, client\_id: str, client\_secret: str, redirect\_uri: str)

def get\_authorization\_url(self, state: Optional[str] = None) -> str

def exchange\_code\_for\_token(self, code: str) -> dict

def get\_user\_info(self, access\_token: str) -> dict

class JWTManager:

def \_\_init\_\_(self, secret\_key: str, algorithm: str = "HS256")

def create\_access\_token(self, data: dict, expires\_delta: Optional[timedelta] = None) -> str

def create\_refresh\_token(self, data: dict) -> str

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

def decode\_token(self, token: str) -> dict

class SessionManager:

def \_\_init\_\_(self)

def create\_session(self, user\_id: str, device\_info: dict) -> str

def validate\_session(self, session\_token: str) -> bool

def invalidate\_session(self, session\_token: str)

def cleanup\_expired\_sessions()

# OAuth Functions

async def initiate\_google\_login() -> dict

async def handle\_google\_callback(code: str, state: Optional[str] = None) -> TokenResponse

async def refresh\_access\_token(refresh\_token: str) -> TokenResponse

async def revoke\_token(token: str) -> bool

# JWT Token Functions

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

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

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

def blacklist\_token(token: str)

def is\_token\_blacklisted(token: str) -> bool

# Authentication Dependencies

async def get\_current\_user(token: str = Depends(oauth2\_scheme)) -> User

async def get\_current\_active\_user(current\_user: User = Depends(get\_current\_user)) -> User

async def require\_authentication(request: Request) -> User

# Session Management Functions

async def create\_user\_session(user: User, device\_info: dict) -> str

async def validate\_user\_session(session\_id: str) -> bool

async def logout\_user(session\_id: str) -> bool

async def get\_active\_sessions(user\_id: str) -> List[dict]

# Security Functions

def hash\_token(token: str) -> str

def generate\_state\_token() -> str

def validate\_state\_token(token: str, stored\_token: str) -> bool

def generate\_csrf\_token() -> str

def validate\_csrf\_token(token: str) -> bool

# Permission Functions

async def check\_document\_permission(user\_id: str, doc\_id: str) -> bool

async def check\_chat\_permission(user\_id: str, session\_id: str) -> bool

def require\_admin\_access(current\_user: User) -> bool

**Frontend Files**

**frontend/main.py [PyQt6 Application Entry & System Tray]**

# Classes

class RAGDesktopApp(QApplication):

def \_\_init\_\_(self, sys\_argv: List[str])

def setup\_system\_tray(self)

def create\_main\_window(self)

def handle\_app\_exit(self)

class SystemTrayManager:

def \_\_init\_\_(self, main\_window: QMainWindow)

def create\_tray\_icon(self) -> QSystemTrayIcon

def create\_tray\_menu(self) -> QMenu

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

def show\_notification(self, title: str, message: str)

# Application Functions

def main() -> int

def setup\_application() -> RAGDesktopApp

def load\_app\_styles() -> str

def setup\_signal\_handlers()

def handle\_single\_instance()

# System Tray Functions

def create\_tray\_menu\_actions() -> List[QAction]

def handle\_show\_window()

def handle\_hide\_window()

def handle\_quit\_application()

def restore\_window\_from\_tray()

def minimize\_to\_tray()

# Application Lifecycle

def on\_application\_startup()

def on\_application\_shutdown()

def save\_application\_state()

def restore\_application\_state()

# Error Handling

def setup\_exception\_handling()

def handle\_unhandled\_exception(exc\_type, exc\_value, exc\_traceback)

def show\_error\_dialog(title: str, message: str)

**frontend/main\_window.py [Main UI Components]**

# Main Window Class

class MainWindow(QMainWindow):

def \_\_init\_\_(self)

def setup\_ui(self)

def setup\_menu\_bar(self)

def setup\_toolbar(self)

def setup\_status\_bar(self)

def setup\_central\_widget(self)

def connect\_signals(self)

# Window Management

def closeEvent(self, event: QCloseEvent)

def changeEvent(self, event: QEvent)

def resizeEvent(self, event: QResizeEvent)

def minimize\_to\_tray(self)

def restore\_from\_tray(self)

# UI Updates

def update\_connection\_status(self, connected: bool)

def update\_user\_info(self, user: dict)

def show\_loading(self, show: bool)

# Chat Widget Class

class ChatWidget(QWidget):

def \_\_init\_\_(self, api\_client: APIClient)

def setup\_ui(self)

def setup\_chat\_display(self)

def setup\_input\_area(self)

def setup\_message\_controls(self)

# Chat Functions

def send\_message(self)

def display\_user\_message(self, message: str)

def display\_assistant\_message(self, message: str)

def handle\_streaming\_response(self, response\_stream)

def clear\_chat(self)

def load\_chat\_history(self, session\_id: str)

# UI Interactions

def handle\_enter\_key(self)

def handle\_file\_drop(self, file\_paths: List[str])

def auto\_resize\_input(self)

# Document Widget Class

class DocumentWidget(QWidget):

def \_\_init\_\_(self, api\_client: APIClient)

def setup\_ui(self)

def setup\_document\_list(self)

def setup\_upload\_area(self)

def setup\_document\_controls(self)

# Document Management

def upload\_documents(self, file\_paths: List[str])

def refresh\_document\_list(self)

def delete\_selected\_document(self)

def preview\_document(self, doc\_id: str)

def reprocess\_document(self, doc\_id: str)

# UI Updates

def update\_upload\_progress(self, progress: int)

def display\_processing\_status(self, doc\_id: str, status: str)

def handle\_drag\_drop(self, event: QDropEvent)

# Settings Widget Class

class SettingsWidget(QWidget):

def \_\_init\_\_(self)

def setup\_ui(self)

def setup\_general\_settings(self)

def setup\_ai\_settings(self)

def setup\_appearance\_settings(self)

# Settings Management

def load\_settings(self)

def save\_settings(self)

def reset\_to\_defaults(self)

def apply\_theme(self, theme\_name: str)

def update\_model\_settings(self)

# Login Widget Class

class LoginWidget(QWidget):

def \_\_init\_\_(self, auth\_manager: AuthManager)

def setup\_ui(self)

def setup\_login\_form(self)

def setup\_oauth\_buttons(self)

# Authentication

def handle\_google\_login(self)

def handle\_login\_success(self, user\_data: dict)

def handle\_login\_error(self, error: str)

def show\_loading\_state(self)

# Utility Functions

def create\_message\_bubble(message: str, is\_user: bool) -> QWidget

def format\_timestamp(timestamp: datetime) -> str

def setup\_drag\_drop\_area(widget: QWidget)

def show\_confirmation\_dialog(parent: QWidget, title: str, message: str) -> bool

def show\_progress\_dialog(parent: QWidget, title: str) -> QProgressDialog

**frontend/api\_client.py [Backend Communication]**

# Classes

class APIClient:

def \_\_init\_\_(self, base\_url: str)

def set\_auth\_token(self, token: str)

def clear\_auth\_token(self)

def get\_headers(self) -> dict

# Request Methods

async def get(self, endpoint: str, params: dict = None) -> dict

async def post(self, endpoint: str, data: dict = None, files: dict = None) -> dict

async def put(self, endpoint: str, data: dict = None) -> dict

async def delete(self, endpoint: str) -> dict

async def stream\_post(self, endpoint: str, data: dict = None) -> AsyncIterator[str]

class RequestManager:

def \_\_init\_\_(self, session: httpx.AsyncClient)

def handle\_response(self, response: httpx.Response) -> dict

def handle\_error(self, error: Exception)

def retry\_request(self, request\_func, max\_retries: int = 3)

# Authentication API Methods

async def google\_login(self) -> dict

async def google\_callback(self, code: str) -> dict

async def refresh\_token(self, refresh\_token: str) -> dict

async def logout(self) -> bool

async def get\_user\_profile(self) -> dict

# Document API Methods

async def upload\_document(self, file\_path: str) -> dict

async def get\_documents(self, skip: int = 0, limit: int = 10) -> List[dict]

async def get\_document(self, doc\_id: str) -> dict

async def delete\_document(self, doc\_id: str) -> bool

async def reprocess\_document(self, doc\_id: str) -> dict

async def get\_document\_chunks(self, doc\_id: str) -> List[dict]

# Chat API Methods

async def send\_chat\_message(self, message: str, session\_id: str = None) -> dict

async def stream\_chat\_response(self, message: str, session\_id: str = None) -> AsyncIterator[str]

async def get\_chat\_history(self, session\_id: str) -> List[dict]

async def create\_new\_chat(self) -> dict

async def delete\_chat(self, session\_id: str) -> bool

# Search API Methods

async def semantic\_search(self, query: str, doc\_ids: List[str] = None) -> List[dict]

async def rag\_query(self, query: str, max\_results: int = 5) -> dict

async def search\_chunks(self, query: str, doc\_id: str) -> List[dict]

# System API Methods

async def get\_health\_status(self) -> dict

async def get\_available\_models(self) -> List[str]

async def update\_settings(self, settings: dict) -> dict

# Error Handling

def handle\_network\_error(self, error: Exception)

def handle\_auth\_error(self, response: httpx.Response)

def handle\_server\_error(self, response: httpx.Response)

def show\_error\_notification(self, message: str)

# Connection Management

async def test\_connection(self) -> bool

def is\_connected(self) -> bool

def setup\_offline\_mode(self)

def queue\_offline\_requests(self, request\_data: dict)

**frontend/session\_manager.py [Local Session Management]**

# Classes

class SessionManager:

def \_\_init\_\_(self, storage\_path: str)

def load\_session(self) -> dict

def save\_session(self, session\_data: dict)

def clear\_session(self)

def is\_valid\_session(self) -> bool

class LocalStorageManager:

def \_\_init\_\_(self, app\_data\_dir: str)

def save\_data(self, key: str, data: dict)

def load\_data(self, key: str) -> dict

def delete\_data(self, key: str)

def clear\_all\_data(self)

class OfflineModeManager:

def \_\_init\_\_(self)

def enable\_offline\_mode(self)

def disable\_offline\_mode(self)

def is\_offline(self) -> bool

def queue\_action(self, action\_data: dict)

def sync\_queued\_actions(self)

# Session Functions

def create\_new\_session(user\_data: dict) -> str

def restore\_session\_from\_storage() -> Optional[dict]

def update\_session\_data(key: str, value: any)

def get\_session\_data(key: str) -> any

def validate\_session\_token(token: str) -> bool

# Window State Management

def save\_window\_state(window: QMainWindow)

def restore\_window\_state(window: QMainWindow)

def save\_window\_geometry(geometry: QByteArray)

def restore\_window\_geometry() -> Optional[QByteArray]

# Application Settings

def save\_app\_settings(settings: dict)

def load\_app\_settings() -> dict

def reset\_app\_settings()

def get\_setting(key: str, default=None) -> any

def set\_setting(key: str, value: any)

# Cache Management

def cache\_user\_data(user\_data: dict)

def get\_cached\_user\_data() -> Optional[dict]

def cache\_documents\_list(documents: List[dict])

def get\_cached\_documents() -> List[dict]

def clear\_cache()

# Background Sync

async def background\_sync\_session()

async def sync\_offline\_changes()

async def upload\_queued\_documents()

async def download\_latest\_data()

# Device Management

def get\_device\_identifier() -> str

def register\_device(user\_id: str) -> str

def unregister\_device(device\_id: str)

def get\_device\_info() -> dict

# Security Functions

def encrypt\_session\_data(data: dict) -> str

def decrypt\_session\_data(encrypted\_data: str) -> dict

def generate\_device\_key() -> str

def validate\_device\_key(key: str) -> bool

**Deployment & Scripts Files**

**deployment/build\_installer.py [PyInstaller Automation]**

# Classes

class InstallerBuilder:

def \_\_init\_\_(self, platform: str)

def setup\_build\_environment(self)

def prepare\_assets(self)

def build\_executable(self)

def create\_installer(self)

class WindowsBuilder(InstallerBuilder):

def build\_exe(self)

def create\_nsis\_installer(self)

def sign\_executable(self)

class MacOSBuilder(InstallerBuilder):

def build\_app\_bundle(self)

def create\_dmg(self)

def sign\_and\_notarize(self)

# Build Functions

def build\_for\_windows() -> str

def build\_for\_macos() -> str

def build\_universal() -> List[str]

def clean\_build\_artifacts()

# Asset Management

def copy\_resources\_to\_build()

def bundle\_python\_dependencies()

def include\_data\_files()

def optimize\_build\_size()

# Installer Creation

def create\_windows\_installer(exe\_path: str) -> str

def create\_macos\_dmg(app\_path: str) -> str

def generate\_installer\_metadata()

def create\_update\_manifest()

# Code Signing

def sign\_windows\_executable(exe\_path: str)

def sign\_macos\_app(app\_path: str)

def verify\_signatures()

# Validation

def test\_installer(installer\_path: str) -> bool

def validate\_dependencies()

def check\_antivirus\_compatibility()

**scripts/setup\_dev.py [Development Environment Setup]**

# Setup Functions

def setup\_development\_environment()

def install\_python\_dependencies()

def setup\_database()

def download\_ai\_models()

def configure\_environment\_variables()

# Docker Functions

def setup\_docker\_environment()

def build\_docker\_images()

def start\_development\_services()

def stop\_development\_services()

# Database Functions

def create\_development\_database()

def run\_database\_migrations()

def seed\_test\_data()

def backup\_development\_data()

# Model Management

def download\_sentence\_transformers()

def setup\_ollama\_models()

def verify\_model\_installations()

def update\_models()

# Validation Functions

def verify\_installation()

def test\_api\_endpoints()

def test\_ui\_components()

def run\_integration\_tests()

**scripts/init\_models.py [AI Model Setup]**

# Model Download Functions

def download\_sentence\_transformer\_model(model\_name: str = "all-MiniLM-L6-v2")

def setup\_ollama\_gemma()

def verify\_model\_compatibility()

def optimize\_models\_for\_device()

# Ollama Management

def install\_ollama()

def pull\_gemma\_model()

def configure\_ollama\_settings()

def test\_ollama\_connection()

# Model Optimization

def optimize\_embedding\_model()

def configure\_gpu\_acceleration()

def set\_model\_cache\_size()

def benchmark\_model\_performance()

**scripts/cythonize.py [Performance Optimization]**

# Cython Compilation Functions

def cythonize\_backend\_modules()

def compile\_critical\_functions()

def optimize\_embedding\_pipeline()

def benchmark\_performance\_improvements()

# Module Selection

def identify\_bottleneck\_modules()

def prepare\_modules\_for\_cython()

def handle\_type\_annotations()

def resolve\_import\_dependencies()

# Build Process

def setup\_cython\_build\_environment()

def compile\_cython\_extensions()

def package\_optimized\_modules()

def validate\_cython\_builds()

**Test Files**

**tests/test\_auth.py [Authentication Testing]**

# Test Classes

class TestGoogleOAuth:

def test\_authorization\_url\_generation(self)

def test\_code\_exchange(self)

def test\_user\_info\_retrieval(self)

class TestJWTManager:

def test\_token\_creation(self)

def test\_token\_verification(self)

def test\_token\_expiration(self)

class TestSessionManager:

def test\_session\_creation(self)

def test\_session\_validation(self)

def test\_session\_cleanup(self)

# Test Functions

def test\_login\_flow()

def test\_logout\_functionality()

def test\_token\_refresh()

def test\_permission\_checks()

def test\_session\_persistence()

**tests/test\_rag.py [RAG Pipeline Testing]**

# Test Classes

class TestDocumentProcessing:

def test\_text\_extraction(self)

def test\_document\_chunking(self)

def test\_embedding\_generation(self)

class TestVectorStore:

def test\_vector\_storage(self)

def test\_similarity\_search(self)

def test\_vector\_deletion(self)

class TestRAGPipeline:

def test\_context\_retrieval(self)

def test\_response\_generation(self)

def test\_fallback\_mechanism(self)

# Test Functions

def test\_end\_to\_end\_rag()

def test\_chunking\_algorithms()

def test\_embedding\_quality()

def test\_search\_accuracy()

**tests/test\_api.py [API Endpoint Testing]**

# Test Classes

class TestDocumentAPI:

def test\_document\_upload(self)

def test\_document\_listing(self)

def test\_document\_deletion(self)

class TestChatAPI:

def test\_chat\_creation(self)

def test\_message\_sending(self)

def test\_response\_streaming(self)

class TestSearchAPI:

def test\_semantic\_search(self)

def test\_document\_search(self)

def test\_chunk\_search(self)

# Test Functions

def test\_api\_authentication()

def test\_error\_handling()

def test\_rate\_limiting()

def test\_data\_validation()