IntelliDoc Nexus

Multi-Agent RAG-Powered Document Intelligence Platform

Technical Presentation & Live Demo

30-Minute Walkthrough

Presented by: [YOUR NAME]

[YOUR TITLE / ROLE]

[DATE]

Built with: Python | React | FastAPI | PostgreSQL | Pinecone | Claude API | Docker

Agenda

30-Minute Presentation

1.	Introduction & Project Overview	*2 min*
2.	The Problem I Solved	*3 min*
3.	System Architecture	*5 min*
4.	Tech Stack Deep Dive	*3 min*
5.	RAG Pipeline Explained	*5 min*
6.	Multi-Agent System	*4 min*
7.	Live Demo	*5 min*
8.	Engineering Challenges	*5 min*
9.	Testing & Quality Assurance	*2 min*
10.	Project Metrics & Achievements	*2 min*
11.	Key Concepts Mastered	*3 min*
12.	What Makes This Different	*2 min*
13.	Future Roadmap & Q&A	*2 min*

Introduction & Project Overview

[2 minutes]

SAY: "Good morning/afternoon everyone. My name is [YOUR NAME], and today I'm going to walk you through IntelliDoc Nexus - a production-grade, multi-agent Retrieval-Augmented Generation platform that I designed and built from the ground up."

IntelliDoc Nexus is a full-stack AI system that takes real documents - PDFs, Word files, text files - processes them through a sophisticated machine learning pipeline, stores them in a vector database, and uses five specialized AI agents working together to answer questions about those documents with source citations and real-time streaming responses.

SAY: "This is not a tutorial project. This is not a to-do app. This is a full-stack AI system that demonstrates I can take a complex, ambiguous problem and turn it into a working, deployed, production-ready system with proper architecture, testing, error handling, and DevOps."

At a Glance

99	6,700+	103	4
Source Files	Lines of Code	Automated Tests	Docker Services

The Problem I Solved

[3 minutes]

SAY: "Before I get into the technical details, let me explain the real-world problem this solves, because understanding the problem is half the engineering."

Organizations today are drowning in documents. Research papers, legal contracts, technical manuals, policy documents. The knowledge is there, but finding specific information across hundreds of pages is painful.

Three Core Problems

- **Problem 1: Information Retrieval.** Traditional keyword search fails when you're looking for concepts, not exact words. If I search for 'cost optimization' but my document says 'reducing expenses,' keyword search won't find it.
- **Problem 2: Context Synthesis.** Even if you find the right paragraphs, you still have to read through them, understand them, and synthesize an answer. That takes time, and humans miss things.
- **Problem 3: Trustworthiness.** If I use ChatGPT to answer a question about my documents, it might hallucinate - give a confident answer that's completely made up. There's no way to verify where the information came from.

SAY: "IntelliDoc Nexus solves all three. It uses semantic search to find relevant content by meaning. It uses multiple AI agents to synthesize, cite, and verify the answer. And it provides source citations so you can trace every claim back to the exact document and page number. This is the same architecture that companies like Notion, Glean, and Perplexity AI use in production."

System Architecture

[5 minutes]

SAY: "Let me walk you through the system architecture. I want you to see that this isn't code thrown together - it's a deliberately designed system with clear separation of concerns."

Four Containerized Services

- **React Frontend (Port 3000) -** Modern SPA with TypeScript, React 18, TailwindCSS, Zustand for state management.
- **FastAPI Backend (Port 8000) -** Async Python API server with 18 REST endpoints, real-time SSE streaming.
- **PostgreSQL 16 -** Primary data store for documents, chunks, sessions, users. Migrations via Alembic.
- **Redis 7 -** Message broker for Celery background tasks and caching layer.

Backend Architecture Layers

- **API Layer -** FastAPI routes, dependency injection, Pydantic v2 validation, proper HTTP status codes.
- **Service Layer -** Document processing, chunking, embedding, vector storage, BM25 search, RAG pipeline.
- **Agent Layer -** Five specialized AI agents: Retrieval, Synthesis, Citation, Reflection, Orchestrator.
- **Data Layer -** SQLAlchemy 2.0 async ORM with custom cross-database type compatibility.

Key Architectural Decisions

- **Async everywhere.** The entire backend uses asyncio, asyncpg, and async SQLAlchemy. Handles hundreds of concurrent requests without blocking.
- **Lazy loading.** Heavy libraries (sentence-transformers, anthropic, pinecone) are imported on first use. Cuts startup time from 30s to under 3s.
- **Namespace-based multi-tenancy.** Each user's vectors are in a separate Pinecone namespace. Data isolation at the infrastructure level.
- **Hybrid search with RRF.** Combines dense vector search with sparse BM25 search using Reciprocal Rank Fusion. Same technique used by Microsoft Bing and Elasticsearch 8.

Tech Stack Deep Dive

[3 minutes]

SAY: "Let me break down the complete tech stack and why I chose each technology. These weren't random choices - each one was selected for a specific reason."

Frontend

React 18 | TypeScript | TailwindCSS | Zustand | TanStack Query | Vite | Lucide Icons

- **React 18 + TypeScript -** Type safety, component architecture, largest ecosystem.
- **Zustand over Redux -** Lightweight (1KB), zero boilerplate. Right tool for this app's state complexity.
- **TanStack React Query -** Automatic caching, background refetching. Industry standard for server state.
- **Vite -** 10x faster HMR than webpack. Sub-second hot reload.

Backend

FastAPI | SQLAlchemy 2.0 | Claude API | Pinecone | SentenceTransformers | Pydantic v2 | Celery

- **FastAPI -** Fastest Python framework. Async, auto-docs, dependency injection. Chosen over Django REST for streaming.
- **Claude API -** Sonnet for synthesis (quality), Haiku for agents (cost). Model-swappable design.
- **Pinecone Serverless -** Managed vector DB. Chose over ChromaDB/Qdrant for production-grade scaling.
- **SentenceTransformers -** all-MiniLM-L6-v2 model. 384-dim embeddings generated locally without API calls.
- **Pydantic v2 -** Rust-based rewrite. 5-17x faster than v1 for validation/serialization.

Infrastructure & Testing

Docker Compose | PostgreSQL 16 | Redis 7 | Alembic | Prometheus | Structlog | Pytest | Locust

Deep Dive: The RAG Pipeline

[5 minutes]

SAY: "Now let me go deep on the most technically interesting part of this project: the RAG pipeline. RAG stands for Retrieval-Augmented Generation, and it's the core innovation that makes this system useful."

Why RAG matters: LLMs are powerful but have two fundamental limitations - they hallucinate (generate confident but wrong answers) and have a knowledge cutoff (don't know about your private documents). RAG solves both by retrieving relevant context from actual documents and passing it to the LLM along with the question.

Stage 1: Document Ingestion (7-Step Pipeline)

- **1. File Validation -** Type checking, size limits (100MB), extension mapping.
- **2. Duplicate Detection -** SHA-256 content hashing. Instant duplicate recognition.
- **3. Text Extraction -** pdfplumber for PDFs (with tables), python-docx for Word, UTF-8 for text. Built a sanitization layer for null bytes.
- **4. Semantic Chunking -** Heading detection + sentence-boundary splits with configurable overlap. Preserves semantic coherence.
- **5. Embedding Generation -** 384-dim vectors via SentenceTransformers. Batched in groups of 32 for memory efficiency.
- **6. Vector Upsert -** Embeddings stored in Pinecone with metadata (doc ID, chunk index, page, content).
- **7. BM25 Indexing -** Chunks tokenized and added to in-memory BM25 index for keyword search.

Stage 2: Hybrid Retrieval

When a user asks a question, two search strategies run in parallel:

- Dense search via Pinecone - Converts question to vector, finds semantically similar chunks. 'Cost reduction' matches 'reducing expenses.'
- Sparse search via BM25 - Traditional keyword matching with TF-IDF. Catches exact term matches vector search might miss.

Results are combined using Reciprocal Rank Fusion (k=60): RRF_score(item) = sum(1/(k + rank)) for each list containing the item. Items appearing in BOTH lists get higher scores. Research shows 5-15% retrieval accuracy improvement.

Stage 3: Context Enrichment

After retrieval, full chunk content is fetched from PostgreSQL (Pinecone metadata is size-limited). Document names are resolved. Claude gets complete paragraphs, not truncated previews.

Stage 4: Generation with Citations

Enriched context is formatted with source markers [Source N] and streamed to Claude. Response streams back via Server-Sent Events in real-time.

Deep Dive: Multi-Agent System

[4 minutes]

SAY: "On top of the standard RAG pipeline, I built a multi-agent system with five specialized agents. This demonstrates understanding of agentic AI architecture - one of the hottest areas in AI right now."

Why multiple agents? A single LLM call can answer questions, but quality improves dramatically when you decompose the task into specialized steps with quality control loops.

The Five Agents

- **1. Retrieval Agent -** Analyzes the question and decides optimal retrieval strategy. Broad or focused? Higher top-k for complex questions? Uses Claude Haiku for speed.
- **2. Synthesis Agent -** Takes retrieved context and generates comprehensive answer. Uses Claude Sonnet (most capable) because synthesis quality drives user satisfaction.
- **3. Citation Agent -** Reviews synthesis output. Verifies every claim is backed by a source. Adds, corrects, or removes citations. This is the trust layer.
- **4. Reflection Agent -** Evaluates complete response for quality. Does it answer the question? Clear? Well-structured? Below threshold? Sends back for revision. Self-improving loop.
- **5. Orchestrator Agent -** Coordinates the pipeline: Retrieval -> Synthesis -> Citation -> Reflection -> (optional revision). Manages state and traces.

SAY: "The key insight is separation of concerns. Each agent has a single responsibility. I implemented this using a state machine pattern - similar to LangGraph but built from scratch to demonstrate I understand the underlying pattern, not just the library API."

Live Demo Script

[5 minutes]

SAY: "Now let me show you the system running live. Everything is running in Docker on my local machine."

- **Step 1: Show the UI.** Open http://localhost:3000. Point out sidebar (upload, documents, history), chat area, search tab, dark mode toggle.
- **Step 2: Upload a document.** Drag and drop a PDF. Show status: Uploading -> Processing -> Complete. Explain: extracted, chunked, embedded, indexed.
- **Step 3: Ask a question.** Type a question and submit. Point out real-time streaming, source citations with [Source N], expandable citation panel.
- **Step 4: Multi-document query.** Select 2-3 documents in sidebar. Ask a comparison question. Show response cites multiple documents separately.
- **Step 5: Semantic Search.** Switch to Search tab. Type a query. Show results with document name, score, page number, preview.
- **Step 6: API Documentation.** Open http://localhost:8000/docs. Show 18 interactive endpoints. Explain OpenAPI/Swagger auto-generation.
- **Step 7: Docker Infrastructure.** Show terminal: docker compose ps. Four containers running with health checks.

Demo Checklist (verify before presenting)

[ ] Docker containers all running: docker compose ps
[ ] Backend health: curl http://localhost:8000/api/v1/health
[ ] Frontend loads at http://localhost:3000
[ ] 2-3 documents uploaded and processed
[ ] Test a chat query to warm up the embedding model
[ ] Have a sample PDF ready for live upload
[ ] Browser in clean state (no console errors)

Engineering Challenges I Solved

[5 minutes]

SAY: "Building this system wasn't straightforward. I ran into real engineering problems that required creative solutions. I think problem-solving ability is what separates a junior developer from someone who can actually build production systems."

Challenge 1: Null Bytes in PDF Extraction

Problem: Academic PDFs produce text with null bytes (\x00) from math symbols. PostgreSQL TEXT columns reject null bytes, causing 500 errors on upload.

Solution: Built a text sanitization layer that strips null bytes and control characters while preserving Unicode. Runs on all extracted text before storage.

Challenge 2: Streaming Database Consistency

Problem: FastAPI's dependency injection commits the DB session when the endpoint returns. But with streaming, the generator outlives the endpoint, so the assistant message was committed before it existed.

Solution: Pre-commit session and user message before streaming starts. Use a fresh database session inside the generator for the assistant message.

Challenge 3: Ephemeral BM25 Index

Problem: BM25 keyword index is in-memory only. Empty after every container restart. Hybrid search degraded to vector-only.

Solution: Added startup hook that rebuilds BM25 index from all document chunks in PostgreSQL. Takes <1 second for thousands of chunks.

Challenge 4: Cross-Database Type Compatibility

Problem: Models used PostgreSQL types (UUID, JSONB, ARRAY) but tests needed SQLite for speed.

Solution: Built custom TypeDecorator classes (GUID, JSONType, ArrayType) that auto-detect dialect. PostgreSQL uses native types, SQLite uses portable alternatives. 103 tests run in 7.7s.

Challenge 5: Frontend Streaming Error Recovery

Problem: Original streaming used fire-and-forget fetch().then(). Errors silently swallowed. UI got permanently stuck in loading state.

Solution: Rewrote as proper async/await with AbortController timeout, HTTP status checking, error propagation, and fallback to non-streaming mode.

Testing & Quality Assurance

[2 minutes]

Production-ready code needs production-ready testing. 103 automated tests across multiple layers:

- **RAG Pipeline Tests -** Reciprocal Rank Fusion scoring, context building, message construction, source extraction. Edge cases: empty results, single source, score normalization.
- **Multi-Agent Tests -** State management, retrieval strategy selection, rank fusion scoring, edge cases for each agent.
- **Ingestion Pipeline Tests -** File extension mapping, content hashing, filename extraction, chunk boundary detection.
- **Integration Tests -** Session CRUD, health endpoint, metrics endpoint, end-to-end chat flow.
- **Load Tests (Locust) -** User behavior profiles simulating realistic usage: uploads, questions, searches under concurrent load.

All tests run in 7.7 seconds on SQLite backend - fast enough for CI/CD on every commit. Alembic migrations provide version-controlled schema changes for production deployments.

Project Metrics & Achievements

[2 minutes]

99 Source Files	**6,700+** Lines of Code	**49** Python Modules	**20** React Components
18 API Endpoints	**5** AI Agents	**103** Automated Tests	**4** Docker Services

Key Features Delivered

- Hybrid search combining dense (vector) and sparse (BM25) retrieval with RRF
- Five-agent pipeline with self-improving reflection loop
- Real-time token streaming via Server-Sent Events
- Multi-format document ingestion (PDF, DOCX, TXT, images)
- Cross-database type compatibility layer (PostgreSQL + SQLite)
- Structured logging, Prometheus metrics, rate limiting, security headers
- Dark mode UI with responsive design and conversation export

Key Concepts I Mastered

[3 minutes]

1. Retrieval-Augmented Generation (RAG)

- Document ingestion pipelines with multi-format extraction
- Semantic chunking with heading detection and sentence-boundary overlap
- Embedding generation with SentenceTransformers
- Vector storage and similarity search with Pinecone
- Context enrichment and citation generation

2. Agentic AI Architecture

- Multi-agent systems with specialized roles
- State machine orchestration patterns
- Self-improving loops with reflection and revision
- Separation of concerns in AI pipeline design

3. Full-Stack Engineering

- Async Python: FastAPI, SQLAlchemy 2.0, asyncpg
- React 18 with TypeScript, Zustand, TanStack Query
- Real-time streaming with Server-Sent Events
- RESTful API design with validation and proper status codes

4. Data Engineering

- Hybrid search: dense + sparse retrieval with RRF
- Vector database management with Pinecone
- PostgreSQL async ORM with cross-database compatibility

5. DevOps & Production Readiness

- Docker Compose multi-service orchestration
- Database migrations, health checks, rate limiting
- Prometheus metrics and structured logging
- Comprehensive automated testing (103 tests)

6. Problem Solving

- Debugging production issues: null bytes, streaming consistency, memory management
- Designing for resilience: timeouts, error recovery, graceful degradation
- Performance optimization: lazy loading, batch processing, caching

What Makes This Different

[2 minutes]

SAY: "I want to address why this project stands out compared to other portfolio projects you might see."

- **Production-grade, not proof-of-concept.** Error handling, input validation, rate limiting, security headers, structured logging, metrics, health checks, automated tests. Most portfolio RAG projects skip all of this.
- **Complete system.** Frontend, backend, database, vector store, search index, AI pipeline, agents, testing, Docker. The full picture, not just an API.
- **Real engineering problems solved.** Null bytes, streaming DB consistency, BM25 persistence, cross-database compatibility. These are production problems solved with proper engineering, not hacks.
- **Theory understood, not just applied.** I can explain why RRF works, why hybrid search outperforms single-strategy, why multi-agent produces better results than single-prompt, and why each architectural decision was made.

This project demonstrates that I can take an ambiguous, complex problem and deliver a working, well-architected, production-ready solution.

Future Roadmap

[1 minute]

- Kubernetes deployment with horizontal pod autoscaling
- OAuth 2.0 / SSO authentication
- WebSocket support for bi-directional real-time communication
- Multi-modal RAG - processing images and charts within documents
- Fine-tuned embedding model on domain-specific data
- CI/CD pipeline with GitHub Actions
- Response caching and adaptive model selection for cost optimization

Q&A Preparation

Q: Why Claude over GPT-4?
Excellent citation following, generous context window, clean streaming API. Designed with abstraction layer - swapping models requires changing one file.

Q: How would you scale this?
Kubernetes for horizontal scaling, managed PostgreSQL (RDS), Pinecone auto-scales, Redis caching, BM25 moves to Elasticsearch.

Q: Hardest bug?
Streaming DB consistency. FastAPI dependency injection lifecycle interacting with async generators required pre-committing and fresh sessions.

Q: Why not LangChain?
Built from scratch to demonstrate understanding of underlying patterns. Full control, easier debugging. Would evaluate LangChain based on team needs.

Q: Latency?
Non-streaming: 3-5s. Streaming first token: <2s. Upload+embed: 1-2s small, 15-20s large PDFs. Bottleneck is Claude API, not infrastructure.

Thank You

IntelliDoc Nexus

Multi-Agent RAG-Powered Document Intelligence

[YOUR NAME]

[YOUR EMAIL]

[YOUR LINKEDIN / GITHUB]

"I'm happy to answer any questions -

and I can dive into any part of the codebase live right now."