KROOLO ENTERPRISE FASTAPI - COMPREHENSIVE BACKEND ARCHITECTURE DOCUMENT

Executive Summary

This document provides a comprehensive technical overview of the Kroolo Enterprise FastAPI backend - a sophisticated, production-grade knowledge management and AI-powered search platform. The system integrates with 15+ enterprise data sources, implements advanced hybrid search capabilities, and provides intelligent AI agents for document interaction and analysis.

---

1. SYSTEM ARCHITECTURE OVERVIEW

1.1 High-Level Architecture

```
┌────────────────────────────────────────────────────────────────┐
│                    CLIENT APPLICATIONS                          │
│              (Web, Mobile, Slack Bot, APIs)                     │
└────────────────────────┬───────────────────────────────────────┘
                         │
                         ▼
┌────────────────────────────────────────────────────────────────┐
│                   FASTAPI APPLICATION LAYER                     │
│  ┌──────────────┬──────────────┬──────────────┬──────────────┐ │
│  │   Routers    │  Middleware  │    Health    │   Tracing    │ │
│  │ (18 modules) │ (CORS/Cache) │  Monitoring  │   (Jaeger)   │ │
│  └──────────────┴──────────────┴──────────────┴──────────────┘ │
└────────────────────────┬───────────────────────────────────────┘
                         │
                         ▼
┌────────────────────────────────────────────────────────────────┐
│                   BUSINESS LOGIC LAYER                          │
│  ┌──────────────┬──────────────┬──────────────┬──────────────┐ │
│  │  AI Agents  │   Search    │    RBAC     │   Graph     │  │
│  │  Services   │   Engine    │   System    │   Engine    │  │
│  └─────────────┴─────────────┴─────────────┴─────────────┘  │
└─────────────────────────┬───────────────────────────────────┘
                          │
                          ▼
┌─────────────────────────────────────────────────────────────┐
│                   DATA PROCESSING LAYER                     │
│  ┌─────────────┬─────────────┬─────────────┬─────────────┐  │
│  │   Celery    │  Document   │  Embedding  │   TF-IDF    │  │
│  │   Workers   │ Processing  │ Generation  │  Indexing   │  │
│  └─────────────┴─────────────┴─────────────┴─────────────┘  │
└─────────────────────────┬───────────────────────────────────┘
                          │
                          ▼
┌─────────────────────────────────────────────────────────────┐
│                      STORAGE LAYER                          │
│  ┌─────────────┬─────────────┬─────────────┬─────────────┐  │
│  │   MongoDB   │    Redis    │     AWS     │  External   │  │
│  │   Atlas     │    Cache    │     S3      │    APIs     │  │
│  └─────────────┴─────────────┴─────────────┴─────────────┘  │
└─────────────────────────────────────────────────────────────┘
```

2. DATA INGESTION ARCHITECTURE

2.1 Connector Framework

The system supports **15+ enterprise data sources** with a unified ingestion pipeline:

Supported Connectors:

1. **Google Drive** (`services/tasks/google_drive_preprocessing.py`)
2. **Slack** (`services/tasks/slack_preprocessing.py`)
3. **Microsoft Teams** (`services/tasks/teams_preprocessing.py`)
4. **SharePoint** (`services/tasks/sharepoint_preprocessing.py`)
5. **Confluence** (`services/tasks/confluence_preprocessing.py`)
6. **Jira** (`services/tasks/jira_preprocessing.py`)
7. **GitHub** (`services/tasks/github_preprocessing.py`)
8. **Salesforce** (`services/tasks/salesforce_preprocessing.py`)
9. **HubSpot** (`services/tasks/hubspot_preprocessing.py`)

10. **Zendesk** (`services/tasks/zendesk_preprocessing.py`)
11. **ServiceNow** (`services/tasks/servicenow_preprocessing.py`)
12. **Dropbox** (`services/tasks/dropbox_preprocessing.py`)
13. **Document360** (`services/tasks/doc360_preprocessing.py`)
14. **Snowflake** (`services/tasks/snowflake_preprocessing.py`)
15. **Workday** (`services/tasks/workday_preprocessing.py`)

2.2 Authentication Layer

Each connector has dedicated authentication logic in `services/auth/` :

- OAuth 2.0 flows for Google, Microsoft, Salesforce
- Token-based authentication for Slack, GitHub
- API key authentication for ServiceNow, Zendesk
- Pipedream-managed connections for enterprise security

Key Features:

- Automatic token refresh
- Secure credential storage
- Multi-tenant isolation by `company_id`
- User-level permissions via `external_user_id`

2.3 Sync Scheduler Architecture

File: `services/tasks/sync_scheduler.py`

Sync Modes:

1. **Scheduled Sync**: Celery Beat runs every 5 minutes (300s)
2. **On-Demand Sync**: Manual "Sync Now" via API
3. **Incremental Sync**: Tracks changes since last sync
4. **Full Sync**: Complete re-indexing of all documents

Sync Flow:

```
1. API Request (/api/v2/connectors/pre-processing)
   ↓
2. Cleanup Prior Data (by org_id + datasource)
   - Delete from enterprisedocuments
   - Delete from enterprisedocuments_vectors
   - Delete from knowledge_graph_nodes
   - Delete from knowledge_graph_edges
   ↓
3. Launch Celery Task
   - Queue: CPU-intensive (Drive, Dropbox) vs IO-intensive (Slack, Jira)
   - Priority: Configurable per connector
   ↓
4. Fetch Documents from Source
   - Pipedream API proxy for authenticated requests
   - Pagination and rate limiting
   - Error handling with categorization
   ↓
5. Document Processing Pipeline (see 2.4)
   ↓
6. Store Results in MongoDB
   ↓
7. Rebuild TF-IDF Index
   ↓
8. Return Processing Status
```

2.4 Document Processing Pipeline

File: `services/clients/unstructured_manager.py`

Processing Strategies:

1. **AUTO**: Intelligent format detection
2. **VLM (Vision Language Model)**: For images, complex layouts
    - Supports: Anthropic, OpenAI, AWS Bedrock, VertexAI
3. **HI_RES**: High-resolution OCR for scanned documents
4. **FAST**: Quick text extraction for simple formats
5. **OCR_ONLY**: Pure OCR processing

Processing Flow:

```
Document (PDF, DOCX, PPTX, etc.)
   ↓
Unstructured.io API / MarkItDown
   ↓
Extract Elements (Title, Text, Table, Image, etc.)
   ↓
Semantic Chunking (Chonkie)
   - Chunk size: 2048 tokens
   - Overlap strategy: Sentence-based
   - Similarity threshold: 0.8
   ↓
Generate Embeddings (OpenAI text-embedding-3-small)
   - Cost tracking per organization
   - Batch processing for efficiency
   ↓
Store in MongoDB
   - Metadata: enterprisedocuments
   - Vectors: enterprisedocuments_vectors
   ↓
Knowledge Graph Extraction
   - Entity extraction (people, teams, projects)
   - Relationship mapping
   - Store in graph collections
```

2.5 Chunking Strategy

File: `utils/semantic_chunker.py`

Semantic Chunking with Chonkie:

- **Algorithm**: Sentence-level semantic similarity
- **Embedding Model**: OpenAI text-embedding-3-small
- **Chunk Size**: 2048 tokens (configurable)
- **Similarity Window**: 3 sentences
- **Threshold**: 0.8 (chunks split when similarity drops)

Advantages:

- Maintains semantic coherence
- Better than fixed-size chunking for retrieval
- Preserves context boundaries
- Optimized for LLM context windows

Fallback Strategy:

- If Chonkie fails, falls back to recursive character splitter
- Configurable chunk size and overlap

2.6 Embedding Generation

File: `services/clients/ultimate_llm.py`

Cost Tracking:

- **File:** `utils/embedding_cost_tracker.py`
- Tracks embedding API usage per user/organization
- Stores in MongoDB analytics collections
- Monitors: token count, API calls, estimated cost

Embedding Models:

- Primary: `text-embedding-3-small` (OpenAI)
- Alternative: `text-embedding-3-large` (higher quality)
- Dimension: 1536 (small) or 3072 (large)

2.7 Celery Task Architecture

File: `app/worker.py`

Queue Configuration:

- **CPU Queue**: Document processing, embeddings (Drive, Dropbox, External Knowledge)
- **IO Queue**: API calls, network requests (Slack, Jira, Salesforce, Teams, etc.)
- **Default Queue**: General tasks

Celery Beat Schedule:

```
beat_schedule = {
    "check-syncs-periodic": {
        "task": "services.tasks.sync_scheduler.check_and_trigger_syncs",
        "schedule": 300.0,  # 5 minutes
    },
    "periodic-slack-sync": {
        "task": "services.tasks.slack_bot_tasks.periodic_slack_sync_all_c
        "schedule": 3540.0,  # 59 minutes
    }
}
```

Task Monitoring:

- Task status tracking with AsyncResult
- Comprehensive error categorization
- Failed file tracking with retry logic
- Processing metrics (successful, failed, pending)

---

3. SEARCH ARCHITECTURE

3.1 Three-Tier Hybrid Search System

The system implements a **true hybrid search** combining three complementary search methods:

```
User Query
   ↓
┌──────────────────────────────────────────────┐
│         PARALLEL SEARCH EXECUTION            │
│                                              │
│  ┌──────────────┐  ┌──────────────┐        │
│  │   Vector     │  │   TF-IDF     │        │
│  │   Search     │  │   Search     │        │
│  │  (Top 10)    │  │  (Top 10)    │        │
│  └──────┬───────┘  └──────┬───────┘        │
│         │                  │                 │
│         │     ┌───────────────┐             │
│         │     │ Knowledge     │             │
│         │     │ Graph Search  │             │
│         │     │   (Top 2)     │             │
│         │     └───────┬───────┘             │
│         │             │                      │
│         └─────┬───────┘                      │
│               │                              │
└───────────────┼──────────────────────────────┘
                ↓
    Reciprocal Rank Fusion (RRF)
                ↓
    Final Ranked Results (Top 50)
                ↓
    Optional: Cohere Reranking
                ↓
    Return to User
```

3.2 TF-IDF Search Implementation

File: `services/clients/tfidf_client.py`

Inverted Index Architecture:

```
{
    "term": {
        "dedup_ids": ["doc1", "doc2", ...],
        "tf": [0.5, 0.3, ...],
        "tfidf": [0.8, 0.6, ...],
        "user_sync": [True, False, ...],
        "permitted_emails": [["user1@"], ["user2@"], ...]
    }
}
```

Text Preprocessing Pipeline:

1. **Lowercasing**: Convert to lowercase
2. **Noise Removal**: URLs, emails, dates, special characters
3. **CamelCase Splitting**: "getUserData" → ["get", "user", "data"]
4. **Stop Word Filtering**: Remove common words
5. **Tokenization**: Split into terms

Scoring Algorithm:

- **TF (Term Frequency)**: `term_count / total_terms_in_doc`
- **IDF (Inverse Document Frequency)**: `log(total_docs / docs_with_term)`
- **TF-IDF Score**: `TF * IDF`

Three-Tier Ranking:

1. **Tier 1 (Score 3)**: All query terms present (complete match)
2. **Tier 2 (Score 2)**: Most query terms present (partial match)
3. **Tier 3 (Score 1)**: Few query terms present (weak match)

MongoDB Aggregation Pipeline:

```
pipeline = [
    # Match documents with query terms
    {"$match": {"organization_id": org_id, "term": {"$in": query_terms}}}

    # Unwind parallel arrays
    {"$unwind": {"path": "$dedup_ids", ...}},

    # Group by document and sum TF-IDF scores
    {"$group": {
        "_id": "$dedup_ids",
        "score": {"$sum": "$tfidf"},
        "matched_terms": {"$sum": 1}
    }},

    # Apply RBAC filtering
    {"$match": {"permitted_emails": {"$in": accessible_emails}}},

    # Sort by score
    {"$sort": {"score": -1, "matched_terms": -1}},

    # Limit results
    {"$limit": 50}
]
```

Performance Optimizations:

- Parallel document processing during indexing
- Batch inserts (1000 documents per batch)
- Efficient MongoDB indexes
- Query result caching (5-minute TTL)

3.3 Vector/Semantic Search

File: `services/clients/mongodb_vector_client.py`

Vector Search Flow:

```
User Query
   ↓
Generate Query Embedding (text-embedding-3-small)
   ↓
MongoDB $vectorSearch Aggregation
   ↓
Filter by RBAC (permitted_emails)
   ↓
Deduplicate Documents
   ↓
Aggregate Chunk Scores
   ↓
Return Top 10 Documents
```

MongoDB Vector Search Pipeline:

```
pipeline = [
    # Vector search stage
    {
        "$vectorSearch": {
            "index": "vector_search_index",
            "path": "embedding",
            "queryVector": query_embedding,
            "numCandidates": 100,
            "limit": 50
        }
    },

    # RBAC pre-filtering
    {"$match": {
        "organization_id": org_id,
        "permitted_emails": {"$in": accessible_emails},
        "user_sync": {"$in": [True, None]}
    }},

    # Add search score
    {"$addFields": {"search_score": {"$meta": "vectorSearchScore"}}},

    # Group by document (dedup_id)
    {"$group": {
        "_id": "$dedup_id",
        "max_score": {"$max": "$search_score"},
        "chunks": {"$push": "$$ROOT"}
    }},

    # Sort and limit
    {"$sort": {"max_score": -1}},
    {"$limit": 10}
]
```

Aggregation Strategies:

1. **Max Score**: Take highest chunk score per document
2. **Weighted Average**: Average top-K chunk scores
3. **Top-K Average**: Average of top 3 chunks

3.4 Knowledge Graph Search

File: `services/graph/unified_graph_query.py`

Graph-Based Retrieval:

- Query organizational knowledge graph
- Find relevant entities (people, teams, projects)
- Traverse relationships

- Return top 2 most relevant documents

Use Cases:

- "Who worked on project X?"
- "Find documents related to team Y"
- "Show me Alice's recent work"

3.5 Reciprocal Rank Fusion (RRF)

File: `utils/hybrid_search.py`

RRF Algorithm:

```
def rrf_score(rank, k=60):
    return 1 / (k + rank)

# For each document in any result set:
final_score = sum([rrf_score(rank_in_vector_search),
                   rrf_score(rank_in_tfidf_search),
                   rrf_score(rank_in_graph_search)])
```

Why RRF:

- Doesn't require score normalization
- Robust to different scoring scales
- Emphasizes documents appearing in multiple result sets
- Research-proven effectiveness

Merging Strategy:

```
merged_results = []
for doc in all_documents:
    rrf_score = 0

    if doc in vector_results:
        rrf_score += 1 / (60 + vector_rank[doc])

    if doc in tfidf_results:
        rrf_score += 1 / (60 + tfidf_rank[doc])

    if doc in graph_results:
        rrf_score += 1 / (60 + graph_rank[doc])
    merged_results.append((doc, rrf_score))

merged_results.sort(key=lambda x: x[1], reverse=True)
return merged_results[:50]
```

3.6 Query Expansion and Reranking

Query Expansion:

- Filename-aware tokenization (splits on underscores, camelCase)
- Synonym expansion (future enhancement)
- Stop word removal

Cohere Reranking:

- Optional post-processing step
- Uses Cohere's rerank-english-v3.0 model
- Re-orders results based on semantic relevance
- Improves precision for top results

3.7 Search API Endpoints

File: `routers/search.py`

Endpoints:

1. **`POST /tfidf-search`** : Standard hybrid search

    - Combines TF-IDF + Vector + Graph
    - RBAC filtering
    - Pagination support
    - Response: List of documents with scores and metadata

2. **`POST /tfidf-search-filtered`** : Datasource-filtered search

    - Additional filtering by datasource (e.g., only Slack)
    - Same hybrid search logic
    - Use case: "Search only in Google Drive"

3. **`POST /tfidf-search-summary`** : AI-powered search summary

    - Performs hybrid search
    - Generates AI summary of results using LLM

- Streaming response via SSE
- Use case: "Summarize all documents about project X"

---

4. AI CHAT & AGENT ARCHITECTURE

4.1 Agent Framework Overview

The system uses **Agno** (formerly Phidata) as the core agent framework with **LangChain** for additional tooling.

Agent Types:

1. **Main Chat Agent**: General-purpose conversational agent
2. **Document Chat Agent**: Document-specific Q&A
3. **File Summary Agent**: Document summarization
4. **Employee Query Agent**: Organizational queries
5. **Trending Documents Agent**: Analytics-based recommendations
6. **Suggested Documents Agent**: Personalized recommendations

4.2 Chat Agent Architecture

File: `services/ai/chat.py`

Agent Creation Flow:

```
def create_chat_agent(config: ChatConfig):
    # 1. Build LLM with model selection
    llm = get_llm_agno(
        model=config.model,
        provider=config.provider,
        organization_id=config.organization_id
    )

    # 2. Create custom retriever (RAG)
    retriever = create_custom_retreiver(
        organization_id=config.organization_id,
        user_id=config.user_id,
        accessible_emails=config.accessible_emails,
        chat_file_ids=config.upload_file_id,
        knowledge_file_ids=config.knowledge_file_id
    )
    # 3. Build tools list
    tools = []

    # Add Composio tools (Gmail, Slack, Calendar, etc.)
    if config.tools_list:
        tools.extend(build_composio_tools(config.tools_list))

    # Add Pipedream integration tools
    if config.integrations:
        tools.extend(build_pipedream_tools(config.integrations))

    # Add inbuilt tools
    tools.extend([
        MongoDBQueryTool(),
        EmployeeQueryTool(),
        ChartGenerationTool(),
        PythonREPLTool(),
        S3UploadTool()
    ])

    # Add MCP server tools
    if config.mcp_servers:
        tools.extend(build_mcp_tools(config.mcp_servers))

    # 4. Create agent with instructions
    agent = Agent(
        model=llm,
        tools=tools,
        knowledge=retriever,
        instructions=build_instructions(config),
        show_tool_calls=True,
        markdown=True,
        add_history_to_messages=True,
        num_history_runs=4,
        enable_agentic_memory=True,
        enable_user_memories=True
    )

    return agent
```

Agent Instructions:

The agent receives detailed instructions including:

- Role and capabilities
- Current date/time and timezone
- User context (name, email, department)
- Company context
- Available tools and their usage
- Mandatory reasoning steps for complex queries
- Intent classification framework
- Response formatting guidelines

4.3 RAG (Retrieval-Augmented Generation)

Custom Retriever:

```
def create_custom_retreiver(
    organization_id: str,
    user_id: str,
    accessible_emails: List[str],
    chat_file_ids: List[str] = None,
    knowledge_file_ids: List[str] = None
):
    """
    Builds a custom retriever that:
    1. Searches enterprise documents (RBAC-filtered)
    2. Includes chat-uploaded files
    3. Includes explicitly requested knowledge files
    4. Deduplicates results
    5. Returns top-K most relevant chunks
    """

    # Create MongoDB vector retriever
    vector_retriever = MongoDBVectorRetriever(
        collection=vector_collection,
        embedding_function=OpenAIEmbeddings(),
        search_kwargs={
            "organization_id": organization_id,
            "permitted_emails": accessible_emails,
            "k": 10
        }
    )

    # Add chat files if provided
    if chat_file_ids:
        chat_docs = load_chat_files(chat_file_ids, organization_id)
```

```
            vector_retriever.add_documents(chat_docs)

        # Add knowledge files if provided
        if knowledge_file_ids:
            knowledge_docs = load_knowledge_files(
                knowledge_file_ids,
                organization_id,
                accessible_emails
            )
            vector_retriever.add_documents(knowledge_docs)

        return vector_retriever
```

Context Sources:

1. **Enterprise Documents**: All indexed documents with RBAC filtering
2. **Chat Files**: Uploaded files in current conversation (chatmedias collection)
3. **Knowledge Files**: Explicitly referenced documents (by dedup_id)

4.4 Hybrid Agent Manager

File: `services/ai/hybrid_agent_manager.py`

Three-Tier Agent Caching:

```
┌──────────────────────────────────────────────┐
│         HYBRID AGENT MANAGER                 │
│                                              │
│  ┌──────────────────────────────────────┐    │
│  │ HOT POOL (50 configurations)         │    │
│  │ - Pre-warmed agents                  │    │
│  │ - Most common configs                │    │
│  │ - Instant retrieval (<10ms)          │    │
│  └──────────────────────────────────────┘    │
│                                              │
│  ┌──────────────────────────────────────┐    │
│  │ WARM POOL (20 configurations)        │    │
│  │ - Recently used agents               │    │
│  │ - TTL-based eviction                 │    │
│  │ - Fast retrieval (~50ms)             │    │
│  └──────────────────────────────────────┘    │
│                                              │
│  ┌──────────────────────────────────────┐    │
│  │  FAST FACTORY                      │   │
│  │  - On-demand creation              │   │
│  │  - Custom configurations           │   │
│  │  - Creation time (~200ms)          │   │
│  └────────────────────────────────────┘   │
└───────────────────────────────────────────┘
```

Performance Metrics:

- Hot pool hit: ~10ms
- Warm pool hit: ~50ms
- Factory creation: ~200ms
- Cache hit rate: >80% in production

Background Warmup:

- Identifies top 50 active companies
- Pre-creates agents for common configurations
- Runs periodically to refresh pools

4.5 Streaming Implementation

File: `services/ai/agent_streaming.py` and `routers/chat.py`

Server-Sent Events (SSE) Protocol:

```
event: run.started
data: {"cache_hit": false}

event: tool.started
data: {"tool_name": "knowledge_retrieval", "input": {...}}

event: tool.completed
data: {"tool_name": "knowledge_retrieval", "output": {...}}

event: message.delta
data: {"content": "Based on the documents..."}

event: message.delta
data: {"content": " I found that..."}

event: message.completed
data: {"content": "Based on the documents I found that..."}
```

```
event: run.completed
data: {"status": "success", "usage": {...}}
```

Semantic Caching:

- Caches chat responses based on semantic similarity
- Uses vector search to find similar queries
- Similarity threshold: 0.90 (configurable)
- TTL: 30 minutes (configurable)
- Simulates streaming for cached responses

Cache Flow:

```python
async def sse_response_cached(query, agent):
    # Check semantic cache
    cached = await semantic_cache.get_response(query)

    if cached:
        # Cache HIT - simulate streaming
        async for chunk in simulate_streaming(cached.response):
            yield chunk
    else:
        # Cache MISS - stream from agent and cache
        buffer = []
        async for chunk in agent.stream(query):
            yield chunk
            buffer.append(chunk)

        # Store in cache
        full_response = "".join(buffer)
        await semantic_cache.set_response(query, full_response)
```

4.6 Tool Integration

Composio Tools (40+ Services):
File: `utils/composio_utils.py` and `agent_config/service_registry.json`

Supported services:

- **Productivity**: Slack, Microsoft Teams, Zoom, Discord
- **Email**: Gmail, Outlook, SendGrid
- **Calendar**: Google Calendar, Outlook Calendar

- **CRM**: Salesforce, HubSpot, Zoho
- **Project Management**: Asana, Trello, Jira, Linear
- **File Storage**: Google Drive, Dropbox, OneDrive
- **Development**: GitHub, GitLab, Bitbucket
- **Marketing**: Mailchimp, ActiveCampaign
- **And 20+ more...**

Pipedream Integration Tools:
File: `utils/pipedream_hybrid_client.py` and custom toolkits in `utils/toolkits/`

Custom toolkits for:

- Slack (messages, channels, users, files)
- Google Drive (search, read, create files)
- Jira (issues, projects, comments)
- Salesforce (accounts, contacts, opportunities)
- Zendesk (tickets, users, organizations)
- HubSpot (contacts, companies, deals)
- SharePoint (files, lists, sites)
- Microsoft Teams (messages, channels, meetings)

MCP (Model Context Protocol) Tools:
File: `utils/mcp_tools.py`

- Remote HTTP MCP servers
- User-specific server configurations stored in MongoDB
- 30-second timeout per tool call
- Supports: Exa search, file search, custom servers

Inbuilt Tools:

1. **MongoDB Query Tool**: Natural language → MongoDB queries
2. **Employee Query Tool**: Organizational graph queries
3. **Chart Generation Tool**: Data visualization
4. **Python REPL Tool**: Code execution via E2B sandbox
5. **S3 Upload Tool**: File uploads to S3
6. **Data Analysis Tool**: Pandas/NumPy analysis
7. **Visualization Tool**: Matplotlib/Plotly charts

4.7 Intent Detection & Classification

Built into Agent Instructions:

Supported intents:

1. **Factual Search**: "What is our return policy?"
2. **Person/Expert Search**: "Who knows about AWS?"
3. **Document/File Search**: "Find the Q3 report"
4. **Troubleshooting**: "Why is the API failing?"
5. **Project Management**: "What's the status of Project X?"
6. **Data/Analytics Query**: "Show sales by region"
7. **Creative Generation**: "Write a blog post about..."
8. **Code/Technical**: "Debug this Python function"
9. **Employee Query**: "Who reports to Alice?"
10. **General Conversation**: Casual chat

Intent-Based Workflow:

```
User Query
   ↓
Agent analyzes intent
   ↓
┌─────────────────────────────────────┐
│  Factual Search                     │
│  → Use knowledge retrieval          │
│  → Search enterprise documents      │
│  → Synthesize answer                │
└─────────────────────────────────────┘

┌─────────────────────────────────────┐
│  Person/Expert Search               │
│  → Use employee query tool          │
│  → Query knowledge graph            │
│  → Find relevant expertise          │
└─────────────────────────────────────┘

┌─────────────────────────────────────┐
│  Data/Analytics Query               │
│  → Use MongoDB query tool           │
│  → Fetch and analyze data           │
│  → Generate visualization           │
│  → Create chart if needed           │
└─────────────────────────────────────┘
```

4.8 Conversation Memory

Agentic Memory:

- Enabled by default
- Stores agent's observations and learnings
- Persists across conversations
- Used for personalization

User Memories:

- Tracks user preferences and context
- Updated based on interactions
- Shared across all agents for user

History Management:

- Last 4 conversation runs included in context
- Maintains conversation flow
- Prevents token limit overflow

---

5. RBAC (ROLE-BASED ACCESS CONTROL)

5.1 Hierarchical Access Model

File: `services/clients/hierarchical_access.py`

Access Control Model:

```
CEO (alice@company.com)
  ↓ (can access)
VP Engineering (bob@company.com)
  ↓ (can access)
Engineering Manager (carol@company.com)
  ↓ (can access)
Senior Engineer (dave@company.com)

- Alice can see: Bob, Carol, Dave (all subordinates)
- Bob can see: Carol, Dave
- Carol can see: Dave
- Dave can see: Only Dave (self)
- Dave CANNOT see: Carol, Bob, Alice (no upward access)
```

Key Features:

- **Downward-only access**: Managers see subordinates, not vice versa
- **Transitive closure**: Managers see all levels below (direct + indirect)

- **Max depth**: 10 levels (configurable)
- **Self-access**: Users always access their own data
- **Case-insensitive**: Email matching ignores case

MongoDB Graph Lookup:

```
pipeline = [
    {"$match": {
        "company_id": company_id,
        "email": user_email,
        "status": "ACTIVE"
    }},
    {"$graphLookup": {
        "from": "members",
        "startWith": "$email",
        "connectFromField": "email",
        "connectToField": "reporting_manager",
        "as": "subordinates",
        "maxDepth": 10,
        "depthField": "level"
    }},
    {"$project": {
        "accessible_emails": {
            "$concatArrays": [
                ["$email"],  # Self
                "$subordinates.email"  # All subordinates
            ]
        }
    }}
]
```

5.2 Document-Level Permissions

Permitted Emails System:

- Each document/chunk has `permitted_emails` array
- Populated during ingestion based on:
  - File owner
  - Shared users (from Google Drive, SharePoint, etc.)
  - Channel members (for Slack, Teams)
  - Workspace members (for Jira, Confluence)

Access Check:

```
def can_access_document(user_email, document):
    accessible_emails = get_accessible_emails(user_email, company_id)

    # Check if any accessible email is in permitted list
    return any(
        email in document["permitted_emails"]
        for email in accessible_emails
    )
```

RBAC in Search:

- All search queries filtered by `permitted_emails`
- Applied at database level (not application level)
- Ensures no unauthorized data leakage
- Performance: Indexed for fast filtering

5.3 User Sync Feature

Purpose: Allow granular control over which documents are searchable

Workflow:

1. Admin selects specific files to sync for users
2. API call: `POST /api/v2/connectors/user-sync`
3. System sets `user_sync=True` for selected files
4. System sets `user_sync=False` for all other files
5. Searches filter by `user_sync=True` (optional)

Use Case:

- "Only make these 10 files searchable for sales team"
- "Temporarily hide sensitive documents"
- "Progressive rollout of new documents"

Bulk Update Performance:

- Uses MongoDB bulk operations
- Updates main collection + vector collection + graph collections
- Triggers TF-IDF rebuild
- Completes in seconds for 10,000+ documents

5.4 Authentication

Basic Auth for Admin Dashboard:

File: `app/cache_auth.py`

```
def verify_cache_admin(credentials: HTTPBasicCredentials):
    correct_username = "Abhishek"
    correct_password = "kroolo@1212"

    is_username_correct = secrets.compare_digest(
        credentials.username, correct_username
    )
    is_password_correct = secrets.compare_digest(
        credentials.password, correct_password
    )

    if not (is_username_correct and is_password_correct):
        raise HTTPException(
            status_code=401,
            detail="Incorrect username or password",
            headers={"WWW-Authenticate": "Basic"}
        )

    return credentials.username
```

Note: This is a simple implementation for the cache admin dashboard. Production systems should use more robust authentication (JWT, OAuth2, etc.).

5.5 Integration-Specific Auth

Each connector has its own authentication module in `services/auth/` :

OAuth 2.0 Flow (Google Drive, Microsoft, etc.):

1. User initiates connection
2. Redirect to OAuth provider
3. User grants permissions
4. Receive access token + refresh token
5. Store encrypted tokens in database
6. Auto-refresh on expiration

Token Management:

- Encrypted storage in MongoDB
- Automatic refresh before expiration

- Revocation handling
- Multi-tenant isolation

---

6. DATABASE ARCHITECTURE

6.1 MongoDB Collections

Primary Collections:

1. `enterprisedocuments` (Metadata Collection)
   - **Purpose**: Document metadata and analytics
   - **Documents**: ~1 per file
   - **Key Fields**:
     - `_id` : ObjectId
     - `organization_id` : ObjectId (tenant isolation)
     - `file_id` : String (unique identifier)
     - `dedup_id` : String (deduplication key)
     - `datasource` : String (google_drive, slack, etc.)
     - `title` : String
     - `url` : String
     - `raw_content` : String (full text)
     - `metadata` : Object (custom fields)
     - `permitted_emails` : Array[String]
     - `user_sync` : Boolean
     - `processing_status` : String
     - `created_at` : Date
     - `updated_at` : Date
2. `enterprisedocuments_vectors` (Vector Collection)
   - **Purpose**: Document chunks with embeddings
   - **Documents**: Many per file (one per chunk)
   - **Key Fields**:
     - `_id` : ObjectId
     - `organization_id` : ObjectId
     - `file_id` : String (links to metadata)
     - `dedup_id` : String
     - `datasource` : String

        - `text` : String (chunk content)
        - `embedding` : Array[Float] (1536 dimensions)
        - `permitted_emails` : Array[String]
        - `user_sync` : Boolean
        - `chunk_index` : Integer
        - `metadata` : Object

3. `knowledge_graph_nodes` (Graph Nodes)

    - **Purpose**: Entities extracted from documents
    - **Documents**: Multiple per file
    - **Key Fields**:
        - `organization_id` : ObjectId
        - `node_id` : String (unique identifier)
        - `node_type` : String (person, team, project, etc.)
        - `properties` : Object
        - `document_id` : String
        - `datasource` : String
        - `permitted_emails` : Array[String]
        - `user_sync` : Boolean

4. `knowledge_graph_edges` (Graph Relationships)

    - **Purpose**: Relationships between entities
    - **Key Fields**:
        - `organization_id` : ObjectId
        - `from_node_id` : String
        - `to_node_id` : String
        - `relationship_type` : String (works_on, reports_to, etc.)
        - `document_id` : String
        - `permitted_emails` : Array[String]

6.2 Indexing Strategy

Metadata Collection Indexes:

```
# Primary filtering
{"organization_id": 1, "datasource": 1}

# Deduplication
```

```
{"organization_id": 1, "dedup_id": 1}

# RBAC filtering
{"organization_id": 1, "permitted_emails": 1}

# User sync
{"organization_id": 1, "user_sync": 1}

# Text search
{"raw_content": "text", "title": "text", "metadata": "text"}

# File ID lookup
{"organization_id": 1, "file_id": 1}
```

Vector Collection Indexes:

```
# Vector search index (Atlas Search)
{
    "name": "vector_search_index",
    "type": "vectorSearch",
    "fields": [{
        "type": "vector",
        "path": "embedding",
        "numDimensions": 1536,
        "similarity": "cosine"
    }]
}

# Compound indexes
{"organization_id": 1, "file_id": 1, "datasource": 1}
{"organization_id": 1, "dedup_id": 1, "datasource": 1}
{"organization_id": 1, "permitted_emails": 1}
{"organization_id": 1, "user_sync": 1}
```

Graph Collection Indexes:

```
# Nodes
{"organization_id": 1, "node_id": 1}  # Unique
{"organization_id": 1, "node_type": 1}
{"organization_id": 1, "user_sync": 1}
{"organization_id": 1, "document_id": 1}

# Edges
```

```
{"organization_id": 1, "from_node_id": 1}
{"organization_id": 1, "to_node_id": 1}
{"organization_id": 1, "relationship_type": 1}
```

6.3 MongoDB Atlas Vector Search

Configuration:

- **Index Type**: `vectorSearch`
- **Similarity Metric**: Cosine similarity
- **Dimensions**: 1536 (for text-embedding-3-small)
- **numCandidates**: 100 (candidates to evaluate)
- **Limit**: 50 (results to return)

Search Pipeline:

```
pipeline = [
    # Stage 1: Vector search
    {
        "$vectorSearch": {
            "index": "vector_search_index",
            "path": "embedding",
            "queryVector": query_embedding,
            "numCandidates": 100,
            "limit": 50
        }
    },

    # Stage 2: RBAC filtering
    {
        "$match": {
            "organization_id": ObjectId(org_id),
            "permitted_emails": {"$in": accessible_emails},
            "user_sync": {"$in": [True, None]}
        }
    },

    # Stage 3: Add score
    {
        "$addFields": {
            "search_score": {"$meta": "vectorSearchScore"}
        }
    },
    # Stage 4: Group by document
    {
        "$group": {
            "_id": "$dedup_id",
            "max_score": {"$max": "$search_score"},
            "avg_score": {"$avg": "$search_score"},
            "chunks": {"$push": "$$ROOT"}
        }
    },

    # Stage 5: Sort and limit
    {"$sort": {"max_score": -1}},
    {"$limit": 10}
]
```

6.4 Data Models

File: `models/`

Key Models:

- `ChatRequest` : Chat API request
- `DocumentChatRequest` : Document-specific chat
- `SearchRequest` : Search API request
- `ConnectorRequestV2` : Connector sync request
- `UserSyncRequest` : User sync management

Pydantic Validation:

- Type checking
- Required field validation
- Default values
- Field descriptions for API docs

6.5 Connection Management

Singleton Pattern:

```
_mongodb_client = None
_mongodb_vector_client = None
```

```
def get_mongodb_client():
    global _mongodb_client
    if _mongodb_client is None:
        _mongodb_client = MongoDBClient()
    return _mongodb_client
```

Connection Pooling:

- Default pool size: 100 connections
- Min pool size: 10
- Max idle time: 60 seconds
- Socket timeout: 30 seconds

Performance:

- Connection reuse across requests
- Thread-safe operations
- Automatic reconnection on failure

---

7. REDIS CACHING ARCHITECTURE

7.1 Cache Configuration

Redis Database Allocation:

- **DB 0**: Celery broker (task queue)
- **DB 1**: Celery results backend
- **DB 2**: Application cache (search, embeddings, semantic cache)

Connection Settings:

- **Host**: Configurable (default: localhost)
- **Port**: 6379
- **Password**: Optional (encrypted)
- **SSL**: Configurable
- **Max Connections**: 50
- **Socket Timeout**: 5 seconds
- **Retry on Timeout**: Yes (3 retries)

7.2 Cache Layers

1. Query Result Cache:

- **Purpose**: Cache search results
- **TTL**: 10 minutes
- **Key Format**: `kroolo:cache:search:{org_id}:{query_hash}:{filters_hash}`
- **Compression**: Enabled for results >1KB

2. Embedding Cache:

- **Purpose**: Cache generated embeddings
- **TTL**: 7 days
- **Key Format**: `kroolo:cache:embedding:{model}:{text_hash}`
- **Storage**: MessagePack serialization

3. Semantic Cache:

- **Purpose**: Cache chat responses
- **TTL**: 30 minutes
- **Key Format**: `kroolo:cache:semantic:{model}:{query_hash}`
- **Similarity Search**: Vector-based similarity matching

4. User Access Cache:

- **Purpose**: Cache hierarchical access lists
- **TTL**: 1 hour
- **Key Format**: `kroolo:cache:access:{user_email}:{company_id}`

7.3 Cache Strategies

Write-Through:

- Data written to cache immediately after database write
- Ensures cache consistency
- Used for: User access, document metadata

Lazy Loading:

- Data loaded into cache on first read
- Cache miss triggers database query
- Used for: Search results, embeddings

Semantic Similarity Cache:

```
async def get_cached_response(query: str, model: str):
    # 1. Generate query embedding
    query_embedding = await generate_embedding(query)
    # 2. Search for similar cached queries
    similar_queries = await semantic_cache.search(
        query_embedding,
        threshold=0.90,
        limit=1
    )

    # 3. Return cached response if found
    if similar_queries:
        return similar_queries[0].response

    return None

async def set_cached_response(query: str, response: str, model: str):
    # 1. Generate query embedding
    query_embedding = await generate_embedding(query)

    # 2. Store with embedding
    await semantic_cache.set(
        query=query,
        embedding=query_embedding,
        response=response,
        model=model,
        ttl=1800  # 30 minutes
    )
```

7.4 Cache Invalidation

Strategies:

1. **TTL-based**: Automatic expiration
2. **Event-driven**: Invalidate on document update/delete
3. **Manual**: Admin dashboard for cache management

Invalidation Events:

- Document ingestion complete → Invalidate search cache
- User permissions change → Invalidate access cache
- Document delete → Invalidate all related caches
- User sync update → Invalidate search cache

7.5 Circuit Breaker Pattern

Purpose: Graceful degradation when Redis is unavailable

States:

1. **CLOSED**: Normal operation, all requests go to cache
2. **OPEN**: Redis unavailable, all requests bypass cache
3. **HALF_OPEN**: Testing Redis recovery

Configuration:

- Failure threshold: 5 consecutive failures
- Recovery timeout: 60 seconds
- Success threshold: 3 consecutive successes

Implementation:

```python
class CacheCircuitBreaker:
    def __init__(self):
        self.state = "CLOSED"
        self.failure_count = 0
        self.last_failure_time = None

    async def execute(self, cache_operation):
        if self.state == "OPEN":
            if time.time() - self.last_failure_time > 60:
                self.state = "HALF_OPEN"
            else:
                return None  # Bypass cache

        try:
            result = await cache_operation()

            if self.state == "HALF_OPEN":
                self.state = "CLOSED"
                self.failure_count = 0

            return result

        except Exception as e:
            self.failure_count += 1

            if self.failure_count >= 5:
                self.state = "OPEN"
                self.last_failure_time = time.time()
            return None  # Bypass cache
```

7.6 Compression

Configuration:

- Enabled for values >1KB
- Algorithm: msgpack + gzip
- Compression ratio: ~70% for text data
- Decompression overhead: <5ms

Performance Impact:

- Network transfer: 70% reduction
- CPU overhead: Minimal (<5ms)
- Overall: Net positive for large payloads

7.7 Cache Admin Dashboard

File: `routers/cache_admin.py`

Features:

- Real-time cache statistics
- Key inspection
- Manual invalidation
- Memory usage monitoring
- Hit/miss rate analytics

Metrics:

- Total keys
- Memory usage
- Hit rate
- Miss rate
- Eviction rate
- Average latency

---

8. BACKGROUND TASK PROCESSING

8.1 Celery Architecture

Worker Configuration:

- **Broker**: Redis DB 0
- **Backend**: Redis DB 1
- **Queues**: `cpu`, `io`, `default`
- **Concurrency**: 4 workers (configurable)
- **Task routing**: By connector type

Queue Assignment:

```
task_routes = {
    # CPU-intensive tasks
    "services.tasks.google_drive_preprocessing.*": {"queue": "cpu"},
    "services.tasks.dropbox_preprocessing.*": {"queue": "cpu"},
    "services.tasks.external_knowledge_preprocessing.*": {"queue": "cpu"}

    # I/O-intensive tasks
    "services.tasks.slack_preprocessing.*": {"queue": "io"},
    "services.tasks.jira_preprocessing.*": {"queue": "io"},
    "services.tasks.salesforce_preprocessing.*": {"queue": "io"},
    # ... all other connectors
}
```

8.2 Task Types

1. Ingestion Tasks:

- Document fetching
- Content parsing
- Embedding generation
- Database storage
- Average duration: 5-30 minutes (depending on doc count)

2. Sync Scheduler Tasks:

- Check sync configurations
- Trigger scheduled syncs
- Update sync status
- Runs every 5 minutes

3. TF-IDF Rebuild Tasks:

- Rebuild inverted index
- Update term frequencies
- Recalculate IDF scores
- Triggered after: ingestion, deletion, user-sync changes

4. Analytics Tasks:

- Usage tracking
- Cost calculation
- Performance metrics

5. User Sync Tasks:

- Bulk update user_sync flags
- Propagate changes across collections

6. Slack Bot Tasks:

- Periodic sync for Slack bot installations
- Message indexing
- User activity tracking

8.3 Task Monitoring

AsyncResult API:

```
from celery.result import AsyncResult

task = AsyncResult(task_id)

# Check status
if task.ready():
    result = task.get()
    print(f"Task completed: {result}")
else:
    print(f"Task pending: {task.state}")

# Monitor progress
if task.state == "PROGRESS":
    info = task.info
    print(f"Progress: {info['current']}/{info['total']}")
```

Task States:

- `PENDING`: Task waiting to execute
- `STARTED`: Task execution started
- `PROGRESS`: Task reporting progress
- `SUCCESS`: Task completed successfully
- `FAILURE`: Task failed with error
- `RETRY`: Task retrying after failure

8.4 Error Handling

Retry Strategy:

```python
@celery_app.task(
    bind=True,
    max_retries=3,
    default_retry_delay=60
)
def process_document(self, doc_id):
    try:
        # Process document
        result = process(doc_id)
        return result

    except RecoverableError as e:
        # Retry on transient errors
        raise self.retry(exc=e)

    except FatalError as e:
        # Don't retry on fatal errors
        logger.error(f"Fatal error: {e}")
        return {"status": "error", "error": str(e)}
```

Error Categorization:

- **Authentication Error**: Invalid credentials, expired tokens
- **Rate Limit Error**: API rate limit exceeded
- **Network Error**: Connection timeout, DNS failure
- **Processing Error**: File parsing failure, unsupported format
- **Storage Error**: Database write failure, S3 upload failure

8.5 Celery Beat Schedule

Periodic Tasks:

```
beat_schedule = {
    # Sync scheduler (every 5 minutes)
    "check-syncs-periodic": {
        "task": "services.tasks.sync_scheduler.check_and_trigger_syncs",
        "schedule": 300.0,
    },

    # Slack bot sync (every 59 minutes)
    "periodic-slack-sync": {
        "task": "services.tasks.slack_bot_tasks.periodic_slack_sync_all_c
        "schedule": 3540.0,
        "options": {"queue": "io", "priority": 5}
    }
}
```

---

9. API ENDPOINTS & ROUTING

9.1 Router Modules

Core Routers:

1. `health.py` : Health check endpoints
2. `root.py` : Root/welcome endpoint
3. `connectors.py` : Data source management
4. `search.py` : Search endpoints
5. `chat.py` : AI chat endpoints
6. `document_chat.py` : Document-specific chat
7. `file_summary.py` : Document summarization
8. `composio.py` : Composio integration management
9. `universal_sync.py` : File-level sync operations
10. `integration_sync.py` : Integration-level sync management
11. `cache_admin.py` : Cache monitoring and management
12. `org_graph.py` : Knowledge graph operations
13. `slack_bot.py` : Slack bot webhook
14. `agent_metrics.py` : Agent performance metrics
15. `llm_health.py` : LLM provider health checks
16. `external_knowledge_specialized.py` : External knowledge base integration

9.2 Key Endpoints

Connector Management:

```
POST   /api/v2/connectors/pre-processing
       - Start document ingestion for a data source
       - Body: company_id, account_id, connector_type
       - Returns: processing_id for status tracking

GET    /api/v2/connectors/processing-status/{processing_id}
       - Check ingestion status
       - Returns: total, successful, failed files

DELETE /api/v2/connectors/disconnect
       - Delete all documents from a data source
       - Body: company_id, datasource
       - Cleans: metadata, vectors, graph

DELETE /api/v2/connectors/files
       - Delete specific files by IDs
       - Body: company_id, file_ids, datasource (optional)

POST   /api/v2/connectors/user-sync
       - Update user_sync flags for specific files
       - Body: company_id, file_ids, datasource, disable
```

Search:

```
POST   /tfidf-search
       - Hybrid search (TF-IDF + Vector + Graph)
       - Body: query, organization_id, user_email, top_k
       - Returns: Ranked documents with scores

POST   /tfidf-search-filtered
       - Hybrid search filtered by datasource
       - Additional parameter: datasource

POST   /tfidf-search-summary
       - Hybrid search with AI-generated summary
       - Streams SSE response
```

Chat:

```
POST   /v1/chat
       - General AI chat with RAG
       - Body: user_query, session_id, company_id, user_id, ...
       - Streaming SSE response
       - Supports: tools, integrations, mcp_servers
```

Document Chat:

```
POST   /api/document-chat
       - Chat with specific document
       - Body: user_query, dedup_id, organization_id, ...
       - Streaming SSE response
```

File Summary:

```
POST   /api/file-summary
       - Generate AI summary of document
       - Body: dedup_id, organization_id, user_id
       - Returns: Executive summary, key points, entities
```

Health & Monitoring:

```
GET    /health
       - Basic health check
       - Returns: status, timestamp

GET    /health/comprehensive
       - Detailed health check
       - Checks: MongoDB, Redis, LLM providers, external services

GET    /api/cache/stats
       - Cache statistics
       - Returns: hit rate, memory usage, key count
```

9.3 Middleware Stack

Order of Execution:

1. **Cache Middleware**: HTTP response caching (60s TTL)
2. **CORS Middleware**: Cross-origin resource sharing
3. **Request Logging**: Log all requests
4. **Error Handling**: Global exception handler

Cache Middleware:

- Caches GET requests only
- Excludes: /health, /metrics, /docs, /api/cache
- TTL: 60 seconds (configurable)
- Compression: Enabled

CORS Configuration:

- Allow origins: `*` (development), specific domains (production)
- Allow credentials: True
- Allow methods: GET, POST, PUT, DELETE, OPTIONS, HEAD, PATCH
- Allow headers: Content-Type, Authorization

---

10. EXTERNAL INTEGRATIONS

10.1 Composio Integration

Purpose: Connect to 40+ external services (Gmail, Slack, Calendar, etc.)

Architecture:

- **Service Registry**: `agent_config/service_registry.json`
- **Lazy Loading**: Services loaded on-demand
- **Pickle Cache**: 3-5x faster loading vs JSON parsing
- **Entity Management**: Per-user connection isolation

Supported Services:

```
{
  "gmail": {"actions": ["GMAIL_SEND_EMAIL", "GMAIL_CREATE_EMAIL_DRAFT", .
  "slack": {"actions": ["SLACK_SENDS_A_MESSAGE_TO_A_SLACK_CHANNEL", ...]}
  "googlecalendar": {"actions": ["GOOGLECALENDAR_CREATE_EVENT", ...]},
  "github": {"actions": ["GITHUB_CREATE_AN_ISSUE", ...]},
  // ... 35+ more services
}
```

Tool Creation:

```
from composio_agno import ComposioToolSet, Action

# Create toolset for user
toolset = ComposioToolSet(entity_id=f"user_{user_id}")
```

```
# Get actions for service
actions = [Action.GMAIL_SEND_EMAIL, Action.GMAIL_CREATE_EMAIL_DRAFT]

# Convert to agent tools
tools = toolset.get_tools(actions=actions)

# Add to agent
agent = Agent(tools=tools, ...)
```

10.2 Pipedream Integration

Purpose: Enterprise connector platform for authenticated API access

Architecture:

- **Official SDK**: Uses `pipedream` Python package
- **Proxy API**: Authenticated requests via `client.proxy.get/post/put/delete/patch`
- **Project Isolation**: Per-project credentials
- **Environment**: Production/staging separation

Custom Toolkits:
Located in `utils/toolkits/`:

1. **Slack Toolkit**: Send messages, upload files, search, etc.
2. **Google Drive Toolkit**: Search, read, create files
3. **Jira Toolkit**: Create/update issues, add comments
4. **Salesforce Toolkit**: Query accounts, contacts, opportunities
5. **Zendesk Toolkit**: Manage tickets, users, organizations
6. **HubSpot Toolkit**: CRM operations
7. **SharePoint Toolkit**: File and list management
8. **Teams Toolkit**: Send messages, manage channels

Example Tool:

```
class SlackSendMessageTool(BaseTool):
    name = "slack_send_message"
    description = "Send a message to a Slack channel"

    def _run(self, channel: str, text: str) -> str:
        client = PipedreamProxyClient()

        result = await client.proxy_request(
            external_user_id=self.external_user_id,
            account_id=self.account_id,
            method="POST",
            url="/api/chat.postMessage",
            body={"channel": channel, "text": text}
        )

        return f"Message sent successfully: {result['ts']}"
```

10.3 MCP (Model Context Protocol) Integration

Purpose: Remote HTTP MCP servers for custom tool integration

Configuration:

- User-specific server configs stored in MongoDB (`mcpconnections` collection)
- Supports: URL, API key, custom parameters
- Timeout: 30 seconds per tool call

Supported Servers:

- **Exa**: Web search via Exa AI API
- **File Search**: Search through uploaded files
- **Custom**: User-defined MCP servers

Tool Creation:

```
from utils.mcp_tools import build_mcp_tools

# Fetch user's MCP server configs
server_configs = fetch_mcp_servers(user_id, server_names)

# Build tools
mcp_tools = build_mcp_tools(server_configs)

# Add to agent
agent = Agent(tools=mcp_tools, ...)
```

10.4 LLM Provider Integration

Portkey AI Gateway:

- **Purpose**: Unified LLM API with fallbacks and routing

- **Providers**: OpenAI, Anthropic, AWS Bedrock, OpenRouter, Groq
- **Virtual Keys**: Provider-specific API keys managed in Portkey
- **Features**: Automatic fallbacks, load balancing, cost tracking

Model Selection:

```
def get_llm_agno(
    model: str = "gpt-4o",
    provider: str = "openrouter",
    organization_id: str = None
):
    # Map provider to virtual key
    virtual_key = {
        "openai": settings.VIRTUAL_KEY_OPENAI,
        "aws": settings.VIRTUAL_KEY_AWS,
        "openrouter": settings.VIRTUAL_KEY_OPENROUTER,
        "groq": settings.VIRTUAL_KEY_GROQ
    }[provider]

    # Create Portkey client
    client = Portkey(
        api_key=settings.PORTKEY_API_KEY,
        virtual_key=virtual_key,
        metadata={"organization_id": organization_id}
    )

    # Return model instance
    return Model(client=client, id=model)
```

Supported Models:

- **OpenAI**: gpt-4o, gpt-4o-mini, gpt-3.5-turbo
- **Anthropic**: claude-3-5-sonnet-20241022, claude-3-opus
- **OpenRouter**: All OpenRouter models
- **AWS Bedrock**: Claude on Bedrock
- **Groq**: Fast inference models

10.5 Document Processing Services

Unstructured.io:

- **Purpose**: Advanced document parsing with VLM support
- **Strategies**: AUTO, VLM, HI_RES, FAST, OCR_ONLY

- **API**: RESTful API with async processing
- **Timeout**: 300 seconds (5 minutes)

MarkItDown:

- **Purpose**: Microsoft's document conversion library
- **Formats**: PDF, DOCX, PPTX, XLSX, images
- **Output**: Clean markdown with preserved structure

Crawl4AI:

- **Purpose**: Web content extraction and scraping
- **Features**: JavaScript rendering, smart extraction, rate limiting
- **Config**: Max concurrent: 5, timeout: 40s, max depth: 3

10.6 Embedding & Reranking Services

OpenAI Embeddings:

- **Models**: text-embedding-3-small, text-embedding-3-large
- **Dimensions**: 1536 (small), 3072 (large)
- **Cost**: $0.00002 per 1K tokens (small)

Cohere Reranking:

- **Model**: rerank-english-v3.0
- **Purpose**: Re-order search results by relevance
- **Input**: Query + documents
- **Output**: Reranked list with relevance scores

---

11. PERFORMANCE OPTIMIZATIONS

11.1 Agent Performance

Hybrid Agent Manager Benefits:

- **Hot Pool**: <10ms agent retrieval for common configs
- **Warm Pool**: ~50ms for recent configs
- **Cache Hit Rate**: >80% in production
- **Memory Efficiency**: Shared LLM instances across agents

Component Caching:

- **File:** `services/ai/component_cache.py`
- Caches: LLMs, embeddings, retrievers
- TTL: 1 hour
- LRU eviction: Max 100 components

11.2 Search Performance

TF-IDF Indexing:

- Parallel document processing (multiprocessing)
- Batch inserts (1000 docs per batch)
- Inverted index storage (O(1) term lookup)
- MongoDB aggregation pipeline (server-side processing)

Vector Search:

- MongoDB Atlas Vector Search (hardware-accelerated)
- HNSW index (Hierarchical Navigable Small World)
- Approximate nearest neighbor (sub-linear complexity)
- Pre-filtering with RBAC (reduces search space)

Hybrid Search:

- Parallel execution (asyncio.gather)
- Timeout per search method (prevents slow methods from blocking)
- Result caching (10-minute TTL)

11.3 Database Optimizations

Indexing:

- Compound indexes for common query patterns
- Covered queries (no document fetch needed)
- Partial indexes for RBAC (smaller index size)

Connection Pooling:

- Pool size: 100 connections
- Connection reuse across requests
- Automatic failover on connection loss

Aggregation Pipeline:

- Server-side processing (reduces network transfer)
- Pipeline optimization (MongoDB query planner)

- Index usage in aggregation stages

11.4 Caching Strategy

Multi-Layer Caching:

1. **Redis**: Shared cache across instances
2. **In-Memory**: Per-instance LRU cache
3. **Semantic Cache**: Vector-based similarity matching

Cache Key Design:

- Hierarchical keys: `kroolo:cache:type:subtype:id`
- Efficient invalidation: Wildcard delete
- Compression: Reduces memory usage by 70%

11.5 Async Operations

FastAPI Async:

- Non-blocking I/O for all endpoints
- Concurrent request handling
- Efficient resource utilization

Celery Background Tasks:

- Offload heavy processing (embeddings, parsing)
- Parallel task execution
- Priority queues for critical tasks

11.6 Token Optimization

Chunking Strategy:

- Semantic chunking (preserves meaning)
- Optimal chunk size: 2048 tokens (fits LLM context)
- Overlap: Sentence-based (prevents context loss)

Prompt Engineering:

- Concise instructions (reduce token usage)
- System prompts cached (not counted in API calls)
- Streaming responses (lower TTFB)

---

12. MONITORING & OBSERVABILITY

12.1 Distributed Tracing

OpenTelemetry + Jaeger:

- **File:** `app/tracing.py`
- Traces all HTTP requests
- Spans for: Database queries, LLM calls, cache operations
- Automatic instrumentation for FastAPI

Trace Context:

```
from opentelemetry import trace

tracer = trace.get_tracer(__name__)

with tracer.start_as_current_span("search_documents") as span:
    span.set_attribute("organization_id", org_id)
    span.set_attribute("query", query)

    results = perform_search(query)

    span.set_attribute("result_count", len(results))
```

Jaeger UI:

- View request traces
- Identify bottlenecks
- Analyze latency distribution
- Debug errors with full context

12.2 Logging

Structured Logging:

- **Format**: JSON with timestamps, levels, context
- **Levels**: DEBUG, INFO, WARNING, ERROR, CRITICAL
- **Context**: Request ID, user ID, organization ID

Log Aggregation:

```
logger.info(
    "Search completed",
    extra={
        "organization_id": org_id,
        "query": query,
        "result_count": len(results),
        "latency_ms": latency,
        "cache_hit": cache_hit
    }
)
```

12.3 Health Checks

Basic Health Check:

```
GET /health
Response: {"status": "healthy", "timestamp": "2024-01-15T10:30:00Z"}
```

Comprehensive Health Check:

```
GET /health/comprehensive
Response: {
  "status": "healthy",
  "mongodb": {"status": "healthy", "latency_ms": 5},
  "redis": {"status": "healthy", "latency_ms": 2},
  "llm_providers": {
    "openai": {"status": "healthy", "latency_ms": 150},
    "anthropic": {"status": "healthy", "latency_ms": 200}
  },
  "celery": {"status": "healthy", "active_workers": 4}
}
```

12.4 Metrics

Cache Metrics:

- Hit rate
- Miss rate
- Memory usage
- Eviction rate
- Average latency

Agent Metrics:

```
GET /api/agent-metrics
Response: {
  "agent_pool": {
    "hot_pool_size": 50,
    "warm_pool_size": 20,
    "cache_hit_rate": 0.85
  },
  "llm_usage": {
    "total_calls": 1000,
    "total_tokens": 500000,
    "average_latency_ms": 250
  }
}
```

Search Metrics:

- Queries per second
- Average latency
- Result count distribution
- RBAC filter efficiency

12.5 Error Tracking

Error Categorization:

- **Authentication Errors**: 401/403 responses
- **Rate Limit Errors**: 429 responses
- **Processing Errors**: Document parsing failures
- **Network Errors**: Timeouts, connection failures
- **Database Errors**: Query failures, connection issues

Error Response Format:

```
{
  "error": "Document processing failed",
  "error_code": "PROCESSING_ERROR",
  "error_category": "document_parsing",
  "details": {
    "file_id": "abc123",
    "datasource": "google_drive",
    "reason": "Unsupported file format: .xyz"
  },
  "timestamp": "2024-01-15T10:30:00Z",
  "request_id": "req_xyz789"
}
```

13. SECURITY CONSIDERATIONS

13.1 Data Isolation

Multi-Tenant Architecture:

- All queries filtered by `organization_id`
- No cross-tenant data leakage
- Database-level isolation

RBAC Enforcement:

- Hierarchical access control
- Document-level permissions
- Filter at query time (not post-fetch)

13.2 Authentication & Authorization

Token Management:

- Encrypted storage in MongoDB
- Automatic token refresh
- Secure credential handling

API Authentication:

- Bearer token authentication (for most endpoints)
- Basic auth for admin dashboard
- OAuth 2.0 for external integrations

13.3 Input Validation

Pydantic Models:

- Type checking
- Required field validation
- Regex patterns for emails, URLs

- Max length constraints

Content Moderation:

- Pre-processing of user queries
- Filtering offensive content
- Rate limiting per user

13.4 Data Encryption

At Rest:

- MongoDB encryption at rest (Atlas feature)
- AWS S3 encryption (AES-256)

In Transit:

- HTTPS/TLS for all API calls
- Encrypted Redis connections (optional)
- Secure WebSocket connections

13.5 Rate Limiting

Configuration:

- Requests per window: 100 (default)
- Window: 1 hour
- Burst: 20 requests
- Per user/organization

Implementation:

- Redis-based sliding window
- Graceful degradation on limit exceeded
- Configurable per endpoint

13.6 Secrets Management

Environment Variables:

- All secrets in `.env` file (not committed)
- Loaded via `python-dotenv`
- Validated at startup

Recommended Improvements:

- Use HashiCorp Vault or AWS Secrets Manager
- Rotate secrets regularly
- Implement secret scanning in CI/CD

---

14. DEPLOYMENT ARCHITECTURE

14.1 Kubernetes Deployment

Manifests: `k8s/` directory

Environments:

- `qa-enterprise-fastapi-manifest/` : QA environment
- `uat-enterprise-fastapi-manifest/` : UAT environment
- `prod-enterprise-fastapi-manifest/` : Production environment

Key Components:

1. **Deployment**: FastAPI application pods
2. **Service**: Load balancer
3. **ConfigMap**: Environment-specific config
4. **Secret**: Sensitive credentials
5. **HorizontalPodAutoscaler**: Auto-scaling
6. **Ingress**: External access

14.2 Container Configuration

Dockerfile:

```
FROM python:3.11-slim

WORKDIR /app

# Install system dependencies
RUN apt-get update && apt-get install -y \
    gcc \
    && rm -rf /var/lib/apt/lists/*

# Install Python dependencies
```

```
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY . .

# Expose port
EXPOSE 8000

# Run application
CMD ["uvicorn", "app.server:app", "--host", "0.0.0.0", "--port", "8000",
```

14.3 Scaling Strategy

Horizontal Scaling:

- Multiple FastAPI instances
- Celery workers scaled independently
- MongoDB Atlas auto-scaling
- Redis Cluster for high availability

Vertical Scaling:

- Increase worker count per instance
- Larger instance types for memory-intensive tasks

Auto-Scaling Triggers:

- CPU usage >70%
- Memory usage >80%
- Request queue length >100

14.4 High Availability

Redundancy:

- Multiple availability zones
- MongoDB Atlas replica sets
- Redis Sentinel for failover
- Load balancer health checks

Disaster Recovery:

- Daily MongoDB backups

- Point-in-time recovery
- Backup retention: 7 days

---

15. FUTURE ENHANCEMENTS

15.1 Short-Term Improvements

Performance:

- Implement connection pooling for LLM providers
- Add more granular caching for embeddings
- Optimize MongoDB aggregation pipelines
- Implement query result pagination

Features:

- Multi-language support for search and chat
- Advanced analytics dashboard
- Custom agent templates
- Webhook support for real-time updates

Security:

- Implement JWT authentication
- Add API key management
- Enhance rate limiting per endpoint
- Implement audit logging

15.2 Long-Term Roadmap

Architecture:

- Migrate to microservices architecture
- Implement GraphQL API
- Add real-time collaboration features
- Support for on-premise deployment

AI Capabilities:

- Fine-tuned models for domain-specific tasks
- Multi-modal search (text + image)
- Automated knowledge graph construction
- Predictive analytics for document usage

Scalability:

- Global multi-region deployment
- Edge computing for latency reduction
- Distributed vector search
- Sharding strategy for large datasets

---

16. CONCLUSION

The Kroolo Enterprise FastAPI backend represents a sophisticated, production-grade knowledge management platform with the following key strengths:

16.1 Technical Excellence

1. **Hybrid Search Architecture**: True multi-method search combining TF-IDF, vector search, and knowledge graph queries with Reciprocal Rank Fusion

2. **Advanced AI Agents**: Three-tier agent pooling system with hot/warm caches and fast factory, achieving <10ms retrieval times for common configurations

3. **Scalable Ingestion Pipeline**: Support for 15+ enterprise data sources with parallel processing, semantic chunking, and comprehensive error handling

4. **Multi-Layer Caching**: Redis-based caching with semantic similarity matching, circuit breaker pattern, and compression achieving 70% memory savings

5. **Robust RBAC**: Hierarchical access control with downward-only permissions, document-level filtering, and graph-based traversal

6. **Performance Optimizations**: Connection pooling, async operations, batch processing, and strategic indexing resulting in sub-second response times

16.2 Production-Ready Features

- **Monitoring**: Comprehensive observability with OpenTelemetry, Jaeger tracing, and structured logging
- **Reliability**: Circuit breakers, graceful degradation, automatic retries, and error categorization
- **Security**: Multi-tenant isolation, RBAC enforcement, input validation, and encrypted storage
- **Scalability**: Kubernetes deployment, horizontal auto-scaling, and distributed task processing

16.3 Innovation Highlights

- **Semantic Chunking**: Context-aware document segmentation using Chonkie
- **Agent Orchestration**: Dynamic tool integration with Composio, Pipedream, and MCP
- **True Hybrid Search**: Combining three complementary search methods with RRF
- **Intelligent Caching**: Vector-based semantic similarity for chat response caching

16.4 Business Value

This architecture enables:

- **Fast Search**: Sub-second response times for queries across millions of documents
- **Accurate Results**: High precision through hybrid search and reranking
- **Intelligent Assistance**: Context-aware AI agents with access to 40+ external services
- **Enterprise Security**: Granular access control and data isolation
- **Operational Excellence**: Comprehensive monitoring, automatic scaling, and graceful error handling

---

Document Version: 1.0
Last Updated: January 28, 2026
Total Lines of Code: ~80,000
Total Files: 175+
Supported Data Sources: 15+
External Integrations: 40+ (via Composio)

This architecture document demonstrates the technical depth, sophistication, and production-readiness of the Kroolo Enterprise FastAPI backend, showcasing advanced software engineering practices and innovative AI/ML implementations.