Multi-Domain RAG System - Phase 2 Implementation Guide

Production-Ready Architecture with Hybrid Retrieval & Service Layer Separation

Version: 2.0
Date: November 24, 2025
Status: Implementation Ready
Document Owner: Technical Architecture Team
Review Cycle: Bi-weekly

Document Control

Version	Date	Author	Changes
1.0	Nov 18, 2025	Team	Initial MVP documentation
2.0	Nov 24, 2025	AI Architect	Phase 2 with service layer, hybrid retrieval, zero UI logic mandate

Table of Contents

1. Executive Summary

Phase 2 transforms the Multi-Domain Document Intelligence Platform into a production-ready, enterprise-grade RAG system with strict architectural separation, hybrid retrieval capabilities, and comprehensive metadata tracking.

Key Enhancements

- **Zero Business Logic in UI:** Mandatory service layer separation ensuring all business logic resides in testable core modules
- **Hybrid Retrieval:** Dense (semantic) + Sparse (keyword) search with configurable alpha weighting for improved precision
- **Enhanced Metadata:** Complete provenance tracking including versioning, deprecation, authority levels, and audit trails
- **Document Lifecycle:** Full support for versioning, deprecation, updates, and rollbacks
- **Production Quality:** 80%+ test coverage, Golden QA sets, performance benchmarks, structured logging

Business Value

- **Maintainability:** Clean architecture enables rapid feature additions without breaking existing code
- **Flexibility:** Multiple UIs (web, CLI, API) can use same core without duplication
- **Testability:** Core logic fully testable independent of UI framework
- **Scalability:** Service layer enables horizontal scaling and microservice migration
- **Quality:** Hybrid retrieval improves answer accuracy by 15-25% over pure vector search

2. Critical Architectural Principles

2.1 MANDATORY: Zero Business Logic in UI Layer ⚠

THIS IS THE MOST IMPORTANT ARCHITECTURAL CONSTRAINT IN PHASE 2

The UI layer (`app.py`, Gradio interface, any future web framework) serves **ONLY** as a thin presentation and routing layer. **ALL business logic, validation, processing, and data management MUST reside in the core service and pipeline layers.**

UI Layer Responsibilities (ALLOWED) ✅

The UI layer SHALL:

- Accept user input (file uploads, query text, dropdown selections)
- Route requests to appropriate service layer APIs
- Display results returned by service layer
- Format error messages for user presentation
- Manage UI component state (tabs, buttons, visibility)
- Handle user session state
- Render data in appropriate UI format (tables, text, charts)

UI Layer Prohibitions (FORBIDDEN) ❌

The UI layer SHALL NOT:

- Validate file types or metadata
- Directly instantiate factories
- Call pipeline methods directly
- Query vector stores
- Parse or process documents
- Execute chunking or embedding logic
- Compute file hashes
- Manage metadata
- Enforce business rules
- Make decisions about deprecation
- Transform or process data
- Import `core.pipeline`, `core.factories`, or `core.vectorstores` modules

Enforcement Rules

Code Review Requirements:

- Any PR with business logic in `app.py` MUST be rejected
- All UI handler functions should be < 20 lines of code
- Every UI action must map to exactly ONE service layer method call
- No conditional business logic in UI (e.g., if/else based on file types)

Architecture Validation:

```
# ✅ CORRECT - UI calls service only
def upload_handler(file, metadata):
    try:
        result = DocumentService.upload_document(file, metadata)
        return f"Success: {result['chunks_ingested']} chunks"
    except ValidationError as e:
        return f"Error: {str(e)}"

# ❌ WRONG - Business logic in UI
def upload_handler(file, metadata):
    # FORBIDDEN: File validation in UI
    if file.name.endswith('.pdf'):
        text = extract_pdf(file)  # FORBIDDEN: Processing in UI
        chunks = chunk_text(text)  # FORBIDDEN: Chunking in UI
        # ... more forbidden logic
```

Rationale

Principle	Benefit
Testability	Core logic tested independently of UI framework; no need to mock Gradio
Maintainability	Business rules in one place; changes don't ripple across UI code
Flexibility	Easy to add CLI, REST API, GraphQL, or different UI without code duplication
Portability	Core can be packaged as library and used in any Python application
Clarity	Clear contracts and responsibilities; new developers understand boundaries
Debuggability	Business logic failures isolated from UI rendering issues

2.2 Service Layer Pattern (MANDATORY)

Phase 2 introduces a **mandatory service layer** that acts as the **sole interface** between UI and core business logic.

Architecture Flow

```
┌──────────────────────────────────────────────────────────────────┐
│  UI Layer (app.py, Gradio)                          │
│  - User interaction                                 │
│  - Display formatting                               │
│  - Routing ONLY                                     │
└────────────────────────┬─────────────────────────────────────────┘
                    │ Service API calls ONLY
                    │ (No direct pipeline access)
┌────────────────────────▼─────────────────────────────────────────┐
│  Service Layer (document_service.py)                │
│  - ALL business logic                               │
│  - Validation (files, metadata)                     │
│  - Orchestration                                    │
│  - Error handling                                   │
└────────────────────────┬─────────────────────────────────────────┘
                    │ Delegates to pipeline
┌────────────────────────▼─────────────────────────────────────────┐
│  Pipeline Layer (document_pipeline.py)              │
│  - Chunking workflow                                │
│  - Embedding workflow                               │
│  - Storage workflow                                 │
│  - Multi-strategy retrieval                         │
└────────────────────────┬─────────────────────────────────────────┘
                    │ Uses factories
┌────────────────────────▼─────────────────────────────────────────┐
│  Factory Layer                                      │
│  - ChunkingFactory                                  │
│  - EmbeddingFactory                                 │
│  - VectorStoreFactory                               │
│  - RetrievalFactory                                 │
└──────────────────────────────────────────────────────────────────┘
```

Key Rule: UI → Service only. **Never** UI → Pipeline directly.

2.3 Configuration-Driven Design

Principle: Everything configurable via YAML; zero hardcoded business rules.

Examples:

- Chunking strategy: `recursive`, `semantic` (not hardcoded if/else)
- Embedding provider: `sentence_transformers`, `openai`, `gemini` (factory-driven)
- Retrieval strategy: `vector_similarity`, `hybrid`, `llm_rerank` (config array)
- Metadata fields: fixed schema but extensions configurable
- File types: `allowed_file_types` in security config
- Hybrid alpha: `alpha: 0.7` in retrieval config

Benefits:

- New domains without code changes
- A/B testing via config toggle
- Easy rollback (revert config)
- Environment-specific settings (dev/staging/prod)

2.4 Factory Pattern for All Components

Principle: All components instantiated via factories based on config.

Factories Required:

- `ChunkingFactory.create_chunker(config)` → Returns chunker implementation
- `EmbeddingFactory.create_embedder(config)` → Returns embedder implementation
- `VectorStoreFactory.create_store(config, dimension)` → Returns vector store
- `RetrievalFactory.create_retriever(config, vector_store, embedder)` → Returns retriever

Adding New Implementations:

1. Implement interface (e.g., `ChunkerInterface`)
2. Register in factory's `_available_implementations` dict
3. Update config schema
4. No changes to pipeline or service layer

3. Goals & Success Criteria

3.1 Phase 2 Objectives

Objective	Description	Success Metric
Service Layer Separation	Zero business logic in UI	100% code review compliance
Hybrid Retrieval	Dense + sparse search with alpha weighting	15%+ accuracy improvement
Enhanced Metadata	Complete provenance tracking	All Phase 2 fields tracked
Document Lifecycle	Version, deprecate, update workflows	Deprecation API functional
Production Quality	Tests, logging, monitoring	80%+ test coverage
Multi-Strategy Retrieval	Support 3+ retrieval strategies	Config-driven strategy selection

3.2 Success Metrics

Technical Metrics:

- Code test coverage: ≥ 80%
- Zero business logic detected in `app.py` via linting

- Pipeline methods never called from UI
- All service APIs have unit tests
- Integration tests cover end-to-end workflows

Quality Metrics:

- Retrieval Recall@10: ≥ 0.85 on Golden QA sets
- Mean Reciprocal Rank (MRR): ≥ 0.75
- Hybrid retrieval outperforms pure vector by ≥ 15%
- Query latency P95: < 500ms

Operational Metrics:

- Successful migration from Phase 1 with zero data loss
- 5+ domains operational without code changes
- Deprecation workflow tested and functional
- CLI tools operational for all key operations

4. Enhanced Metadata Schema

4.1 Complete Metadata Model

All documents and chunks MUST include the following metadata fields. This is a **fixed schema** enforced via Pydantic validation.

Pydantic Model Definition

```
from pydantic import BaseModel, Field, validator
from typing import List, Optional, Dict, Any
from datetime import datetime
from enum import Enum

class AuthorityLevel(str, Enum):
    OFFICIAL = "official"
    APPROVED = "approved"
    DRAFT = "draft"
    ARCHIVED = "archived"
    DEPRECATED = "deprecated"

class ReviewStatus(str, Enum):
    APPROVED = "approved"
    PENDING = "pending"
    REJECTED = "rejected"
    IN_REVIEW = "in_review"

class ChunkMetadata(BaseModel):
    """
    Fixed metadata schema for all chunks across all domains.
    Phase 2 enhancement: Adds lifecycle, provenance, and quality fields.
    """

    # ============ IDENTITY ============
    doc_id: str = Field(..., min_length=1, description="Unique document identifier")
    chunk_id: str = Field(..., min_length=1, description="Unique chunk identifier")

    # ============ CONTENT ============
    chunk_text: str = Field(..., min_length=1, description="Actual chunk text content")
    title: Optional[str] = Field(None, description="Document title")
    page_num: Optional[int] = Field(None, ge=1, description="Source page number")
    char_range: Optional[tuple] = Field(None, description="Character range (start, end)")

    # ============ CLASSIFICATION ============
    domain: str = Field(..., description="Domain: hr, finance, legal, engineering, etc.")
    doc_type: str = Field(..., description="Type: policy, faq, manual, guideline, etc.")
    tags: List[str] = Field(default_factory=list, description="User-defined tags")
    category: Optional[str] = Field(None, description="Sub-category within domain")

    # ============ PROVENANCE ============
    author: Optional[str] = Field(None, description="Original document author")
    uploader_id: str = Field(..., description="User ID who uploaded document")
    upload_timestamp: datetime = Field(
        default_factory=datetime.utcnow,
        description="When document was uploaded"
    )
    source_file: str = Field(..., description="Original filename")
    source_file_hash: str = Field(
        ...,
        min_length=64,
        max_length=64,
        description="SHA-256 hash of source file for integrity"
    )
    source_url: Optional[str] = Field(None, description="Original URL if web-sourced")

    # ============ VERSIONING ============
    version: str = Field(default="1.0", description="Document version")
    document_version: str = Field(default="1.0", description="Semantic version")
    last_updated_timestamp: Optional[datetime] = Field(
        None,
        description="When document was last modified"
    )
    previous_version_id: Optional[str] = Field(
        None,
        description="doc_id of previous version for history"
    )

    # ============ PROCESSING ============
    embedding_version: str = Field(..., description="Embedding model version/name")
    embedding_model_name: str = Field(..., description="Exact model: all-MiniLM-L6-v2, et
    embedding_dimension: int = Field(..., description="Vector dimension: 384, 768, 1536")
    chunking_strategy: str = Field(..., description="Strategy used: recursive, semantic")
    chunking_params: Dict[str, Any] = Field(
        default_factory=dict,
        description="Parameters: chunk_size, overlap, etc."
    )
    processing_timestamp: datetime = Field(
        default_factory=datetime.utcnow,
        description="When chunk was processed"
    )

    # ============ LIFECYCLE ============
    deprecated: bool = Field(default=False, description="Is document/chunk deprecated?")
    deprecated_date: Optional[datetime] = Field(None, description="When deprecated")
    deprecation_reason: Optional[str] = Field(None, description="Why deprecated")
    superseded_by: Optional[str] = Field(None, description="doc_id that replaces this")

    # ============ QUALITY & AUTHORITY ============
    confidence_score: Optional[float] = Field(
        None,
        ge=0.0,
        le=1.0,
        description="Confidence score for this chunk"
    )
    authority_level: AuthorityLevel = Field(
        default=AuthorityLevel.DRAFT,
        description="Authority level of document"
    )
    review_status: ReviewStatus = Field(
        default=ReviewStatus.PENDING,
        description="Review/approval status"
    )
    reviewed_by: Optional[str] = Field(None, description="Reviewer user ID")
    reviewed_date: Optional[datetime] = Field(None, description="When reviewed")

    # ============ CUSTOM/EXTENSIBLE ============
    custom_metadata: Dict[str, Any] = Field(
        default_factory=dict,
        description="Domain-specific custom fields"
    )

    @validator('source_file_hash')
    def validate_hash(cls, v):
        """Ensure hash is valid SHA-256"""
        if not all(c in '0123456789abcdef' for c in v.lower()):
            raise ValueError('Invalid SHA-256 hash format')
        return v.lower()

    class Config:
        use_enum_values = True
        schema_extra = {
            "example": {
                "doc_id": "HR-POLICY-2025-001",
                "chunk_id": "HR-POLICY-2025-001_chunk_5",
                "chunk_text": "Employees are entitled to 15 days of annual leave...",
                "title": "Employee Leave Policy 2025",
                "page_num": 3,
                "char_range": (450, 850),
                "domain": "hr",
                "doc_type": "policy",
                "tags": ["leave", "benefits", "2025"],
                "author": "Jane Doe, HR Director",
                "uploader_id": "user123",
                "upload_timestamp": "2025-11-20T10:30:00Z",
                "source_file": "Leave_Policy_2025.pdf",
                "source_file_hash": "a3b2c1d4e5f6...",
                "version": "2.1",
                "embedding_model_name": "all-MiniLM-L6-v2",
                "embedding_dimension": 384,
                "chunking_strategy": "recursive",
                "chunking_params": {"chunk_size": 500, "overlap": 50},
                "deprecated": False,
                "authority_level": "official",
                "review_status": "approved"
            }
        }
```

4.2 Required Fields by Operation

Operation	Required Metadata Fields
Initial Upload	doc_id, title, domain, doc_type, uploader_id, source_file, source_file_hash
Chunk Storage	All upload fields + chunk_id, chunk_text, embedding_version, embedding_model_name, embedding_dimension, chunking_strategy, chunking_params
Retrieval Filter	domain, deprecated, doc_type, authority_level, review_status
Deprecation	doc_id, deprecated=True, deprecated_date, deprecation_reason
Version Update	doc_id, version, document_version, last_updated_timestamp, previous_version_id

4.3 Metadata Enrichment Pipeline

During Upload:

1. User provides: doc_id, title, domain, doc_type, author
2. System auto-generates: upload_timestamp, source_file_hash
3. System extracts: page_num, char_range (from parser)

During Processing:

1. System adds: embedding_version, embedding_model_name, embedding_dimension
2. System adds: chunking_strategy, chunking_params, processing_timestamp
3. System sets defaults: deprecated=False, authority_level=draft, review_status=pending

During Lifecycle Events:

1. Deprecation: Sets deprecated=True, deprecated_date, deprecation_reason
2. Update: Increments version, sets last_updated_timestamp, links previous_version_id
3. Review: Updates review_status, reviewed_by, reviewed_date

5. Architecture Overview

5.1 Four-Layer Architecture

```
┌─────────────────────────────────────────────────────────────────┐
│                     LAYER 1: UI LAYER                            │
│  ┌──────────────────────────────────────────────────────────┐   │
│  │  app.py (Gradio Interface)                              │   │
│  │  - Accept user input                                     │   │
│  │  - Display results                                       │   │
│  │  - Route to service layer ONLY                          │   │
│  │  - NO business logic                                     │   │
│  └──────────────────────────────────────────────────────────┘   │
└──────────────────────────────┬──────────────────────────────────┘
                               │ Service API Calls
                               │ (upload, query, deprecate)
┌──────────────────────────────▼──────────────────────────────────┐
│                  LAYER 2: SERVICE LAYER                          │
│  ┌──────────────────────────────────────────────────────────┐   │
│  │  document_service.py                                     │   │
│  │  - ALL business logic                                    │   │
│  │  - File type validation                                  │   │
│  │  - Metadata validation & enrichment                      │   │
│  │  - Orchestration                                         │   │
│  │  - Error handling & logging                              │   │
│  └──────────────────────────────────────────────────────────┘   │
└──────────────────────────────┬──────────────────────────────────┘
                               │ Pipeline Delegation
                               │
┌──────────────────────────────▼──────────────────────────────────┐
│                  LAYER 3: PIPELINE LAYER                         │
│  ┌──────────────────────────────────────────────────────────┐   │
│  │  document_pipeline.py                                    │   │
│  │  - Ingestion workflow (chunk → embed → store)            │   │
│  │  - Multi-strategy retrieval orchestration               │   │
│  │  - Metadata attachment                                   │   │
│  │  - Deprecation workflow                                  │   │
│  └──────────────────────────────────────────────────────────┘   │
└──────────────────────────────┬──────────────────────────────────┘
                               │ Factory Usage
                               │
┌──────────────────────────────▼──────────────────────────────────┐
│                  LAYER 4: FACTORY LAYER                          │
│  ┌──────────────┐  ┌──────────────┐  ┌──────────────┐         │
│  │ Chunking     │  │ Embedding    │  │ VectorStore  │         │
│  │ Factory      │  │ Factory      │  │ Factory      │         │
│  └──────────────┘  └──────────────┘  └──────────────┘         │
│                                                                 │
│  ┌──────────────┐  ┌──────────────┐  ┌──────────────┐         │
│  │ Retrieval    │  │ Parser       │  │ Validation   │         │
│  │ Factory      │  │ Factory      │  │ Utils        │         │
│  └──────────────┘  └──────────────┘  └──────────────┘         │
└─────────────────────────────────────────────────────────────────┘
```

5.2 Data Flow Diagrams

Ingestion Flow

```
User uploads file via UI
        ↓
UI: Capture file + metadata
        ↓
UI → Service: document_service.upload_document(file, metadata)
        ↓
Service: Validate file type (allowed_file_types check)
Service: Validate metadata (required fields check)
Service: Compute file hash
        ↓
Service → Pipeline: pipeline.process_document(file, enriched_metadata)
        ↓
Pipeline: Extract text via parser factory
Pipeline: Chunk via chunking factory
Pipeline: Embed via embedding factory
Pipeline: Attach metadata to each chunk
Pipeline: Store via vector store factory
        ↓
Pipeline → Service: Return {doc_id, chunks_ingested, status}
        ↓
Service → UI: Return success message
        ↓
UI: Display "Success: 42 chunks ingested"
```

Query Flow

```
User enters query via UI
        ↓
UI: Capture query text + optional filters
        ↓
UI → Service: document_service.query(query, strategy, filters, top_k)
        ↓
Service: Validate inputs
        ↓
Service → Pipeline: pipeline.query(query, strategy, filters, top_k)
        ↓
Pipeline: Route to retrieval strategy (hybrid, vector, rerank)
Pipeline → Retrieval: Execute search
        ↓
Retrieval: Embed query
Retrieval: Search vector store (dense)
Retrieval: Search BM25 index (sparse) [if hybrid]
Retrieval: Combine scores with alpha weighting [if hybrid]
Retrieval: Apply metadata filters
Retrieval: Return ranked results
        ↓
Pipeline → Service: Return results with metadata
        ↓
Service → UI: Return formatted results
```

```
        ↓
UI: Display results with citations
```

5.3 Component Interaction Matrix

Component	Can Call	Cannot Call	Reason
UI Layer	Service APIs only	Pipeline, Factories, Utils	Separation of concerns
Service Layer	Pipeline, Utils	Factories directly, Vector Stores	Factories via pipeline only
Pipeline Layer	All Factories, Utils	UI layer	Unidirectional flow
Factories	Implementations, Utils	UI, Service, Pipeline	Dependency inversion

6. Service Layer Specification

6.1 DocumentService Class API

The `DocumentService` class is the **sole interface** between UI and core business logic.

```
"""
core/services/document_service.py

Service layer for all document operations.
UI MUST ONLY call methods from this class.
"""

import logging
from typing import Dict, Any, Optional, List
from datetime import datetime

from core.pipeline.document_pipeline import DocumentPipeline
from core.config_manager import ConfigManager
from core.utils.validation import validate_file_type, validate_metadata, validate_file_si
from core.metadata_models import ChunkMetadata

logger = logging.getLogger("DocumentService")

class ValidationError(Exception):
    """Raised when validation fails"""
    pass

class DocumentNotFoundError(Exception):
    """Raised when document not found"""
    pass

class DocumentService:
    """
    Service layer providing high-level document management APIs.

    Responsibilities:
    - Input validation (files, metadata)
    - Business logic enforcement
    - Pipeline orchestration
    - Error handling and logging
    - Metadata enrichment

    UI Layer MUST call only these methods.
    """

    def __init__(self, domain_id: str):
        """
        Initialize service for specific domain.

        Args:
            domain_id: Domain identifier (hr, finance, legal, etc.)
        """
        self.domain_id = domain_id
        self.config_manager = ConfigManager()
        self.domain_config = self.config_manager.load_domain_config(domain_id)
        self.pipeline = DocumentPipeline(self.domain_config)

        # Load security settings
        self.allowed_file_types = set(
            ext.lower() for ext in self.domain_config.security.allowed_file_types
        )
        self.max_file_size_mb = self.domain_config.security.max_file_size_mb

        logger.info(f"DocumentService initialized for domain: {domain_id}")

    # ========== PUBLIC API METHODS ==========

    def upload_document(
        self,
        file_obj: Any,
        metadata: Dict[str, Any],
        replace_existing: bool = False
    ) -> Dict[str, Any]:
        """
        Upload and process a document end-to-end.

        This method:
        1. Validates file type against allowed_file_types
        2. Validates file size
        3. Validates required metadata fields
        4. Computes file hash for provenance
        5. Enriches metadata with system fields
        6. Delegates to pipeline for processing

        Args:
            file_obj: File object with .name attribute
            metadata: Dict with required fields:
                - doc_id (str)
                - title (str)
                - domain (str)
                - doc_type (str)
                - uploader_id (str)
            replace_existing: If True, deletes existing doc before ingestion
        Returns:
            Dict with:
                - doc_id (str)
                - chunks_ingested (int)
                - status (str)
                - file_hash (str)

        Raises:
            ValidationError: If validation fails
            ProcessingError: If pipeline processing fails
        """
        logger.info(f"Upload request: doc_id={metadata.get('doc_id')}, domain={self.domai

        # Step 1: Validate file type
        filename = getattr(file_obj, "name", None)
        if not filename:
            raise ValidationError("File object missing 'name' attribute")

        validate_file_type(filename, self.allowed_file_types)
        logger.debug(f"File type validation passed: {filename}")

        # Step 2: Validate file size
        file_size = getattr(file_obj, "size", None)
        if file_size:
            validate_file_size(file_size, self.max_file_size_mb)
            logger.debug(f"File size validation passed: {file_size} bytes")

        # Step 3: Validate required metadata
        required_fields = ["doc_id", "title", "domain", "doc_type", "uploader_id"]
        validate_metadata(metadata, required_fields)
        logger.debug("Metadata validation passed")

        # Step 4: Compute file hash
        from core.utils.hashing import compute_file_hash
        file_hash = compute_file_hash(file_obj)
        logger.debug(f"Computed file hash: {file_hash[:16]}...")

        # Step 5: Enrich metadata
        enriched_metadata = {
            **metadata,
            "source_file": filename,
            "source_file_hash": file_hash,
            "upload_timestamp": datetime.utcnow(),
            "domain": self.domain_id,  # Enforce domain
        }

        # Step 6: Delegate to pipeline
        try:
            result = self.pipeline.process_document(
                doc=file_obj,
                metadata=enriched_metadata,
                replace_existing=replace_existing
            )
            result["file_hash"] = file_hash
            logger.info(f"Upload successful: {result}")
            return result
        except Exception as e:
            logger.exception(f"Pipeline processing failed for doc_id={metadata.get('doc_i
            raise

    def deprecate_document(
        self,
        doc_id: str,
        reason: str,
        deprecated_date: Optional[datetime] = None
    ) -> None:
        """
        Mark a document as deprecated.

        Updates all chunks with:
        - deprecated = True
        - deprecated_date = provided or now
        - deprecation_reason = reason

        Deprecated documents are filtered out of retrieval by default.

        Args:
            doc_id: Document identifier
            reason: Human-readable deprecation reason
            deprecated_date: Optional date; defaults to now

        Raises:
            DocumentNotFoundError: If document doesn't exist
        """
        logger.info(f"Deprecating document: {doc_id}, reason: {reason}")

        if deprecated_date is None:
            deprecated_date = datetime.utcnow()

        try:
            self.pipeline.deprecate_document(doc_id, deprecated_date, reason)
            logger.info(f"Document {doc_id} successfully deprecated")
        except Exception as e:
            logger.exception(f"Deprecation failed for doc_id={doc_id}")
            raise

    def update_document_metadata(
        self,
        doc_id: str,
        updates: Dict[str, Any]
    ) -> None:
        """
        Update metadata fields for a document.

        Allows updating:
        - authority_level
        - review_status
        - tags
        - custom_metadata

        Prevents updating:
```

```
        - doc_id, chunk_id (immutable)
        - upload_timestamp (immutable)
        - source_file_hash (immutable)

        Args:
            doc_id: Document identifier
            updates: Dict of field: new_value

        Raises:
            ValidationError: If trying to update immutable fields
            DocumentNotFoundError: If document doesn't exist
        """
        immutable_fields = ["doc_id", "chunk_id", "upload_timestamp", "source_file_hash"]

        for field in immutable_fields:
            if field in updates:
                raise ValidationError(f"Cannot update immutable field: {field}")

        logger.info(f"Updating metadata for doc_id={doc_id}: {updates}")

        try:
            self.pipeline.update_document_metadata(doc_id, updates)
            logger.info(f"Metadata update successful for {doc_id}")
        except Exception as e:
            logger.exception(f"Metadata update failed for doc_id={doc_id}")
            raise

    def query(
        self,
        query_text: str,
        strategy: Optional[str] = None,
        metadata_filters: Optional[Dict[str, Any]] = None,
        top_k: int = 10,
        include_deprecated: bool = False
    ) -> List[Dict[str, Any]]:
        """
        Execute semantic query over domain documents.

        Args:
            query_text: Natural language query
            strategy: Retrieval strategy name (hybrid, vector_similarity, etc.)
                     If None, uses all configured strategies
            metadata_filters: Optional filters on metadata fields:
                - domain (str or List[str])
                - doc_type (str or List[str])
                - authority_level (str or List[str])
                - tags (str or List[str])
            top_k: Number of results to return
            include_deprecated: If False (default), filters out deprecated docs

        Returns:
            List of dicts with:
                - chunk_id (str)
                - chunk_text (str)
                - score (float)
                - metadata (ChunkMetadata)
        Example:
            results = service.query(
                "How many vacation days?",
                strategy="hybrid",
                metadata_filters={"doc_type": "policy", "authority_level": "official"},
                top_k=5
            )
        """
        logger.info(f"Query: '{query_text}', strategy={strategy}, top_k={top_k}")

        # Add default filter for deprecated docs
        if not include_deprecated:
            if metadata_filters is None:
                metadata_filters = {}
            metadata_filters["deprecated"] = False

        try:
            results = self.pipeline.query(
                query_text=query_text,
                strategy_name=strategy,
                metadata_filters=metadata_filters,
                top_k=top_k
            )
            logger.info(f"Query returned {len(results)} results")
            return results
        except Exception as e:
            logger.exception(f"Query failed: {query_text}")
            raise

    def get_document_info(self, doc_id: str) -> Dict[str, Any]:
        """
        Retrieve metadata and stats for a document.

        Returns:
            - doc_id
            - title
            - domain
            - upload_timestamp
            - chunk_count
            - deprecated
            - version
            - authority_level
        """
        logger.info(f"Fetching document info: {doc_id}")
        try:
            info = self.pipeline.get_document_info(doc_id)
            return info
        except Exception as e:
            logger.exception(f"Failed to fetch document info: {doc_id}")
            raise DocumentNotFoundError(f"Document not found: {doc_id}")

    def list_documents(
        self,
        filters: Optional[Dict[str, Any]] = None,
        include_deprecated: bool = False
    ) -> List[Dict[str, Any]]:
        """
        List all documents in domain with optional filtering.

        Args:
            filters: Optional metadata filters
            include_deprecated: Include deprecated documents

        Returns:
            List of document summaries
        """
        logger.info(f"Listing documents for domain: {self.domain_id}")

        if not include_deprecated:
            if filters is None:
                filters = {}
            filters["deprecated"] = False

        try:
            docs = self.pipeline.list_documents(filters)
            logger.info(f"Found {len(docs)} documents")
            return docs
        except Exception as e:
            logger.exception("Failed to list documents")
            raise

    # ========== VALIDATION HELPERS (used internally) ==========

    def validate_file_type(self, filename: str) -> bool:
        """Check if file type is allowed. Raises ValidationError if not."""
        return validate_file_type(filename, self.allowed_file_types)

    def validate_metadata(self, metadata: Dict, required_fields: List[str]) -> bool:
        """Check if required metadata fields present. Raises ValidationError if not."""
        return validate_metadata(metadata, required_fields)
```

6.2 Service Layer Design Principles

Principle	Description
Single Responsibility	Each method does one thing (upload, query, deprecate)
Validation First	All inputs validated before pipeline delegation
Error Transformation	Converts technical errors to user-friendly messages
Logging	Structured logging at every step for observability
No Direct Factory Calls	Always delegates to pipeline; factories via pipeline only
Stateless	No instance state beyond config; thread-safe

7. Pipeline Layer Enhancements

7.1 Enhanced DocumentPipeline

```
"""
core/pipeline/document_pipeline.py

Enhanced pipeline with:
- Multi-strategy retrieval
- Metadata lifecycle management
- Deprecation support
- Version tracking
"""

import logging
from typing import List, Dict, Optional, Any
from datetime import datetime

from core.factories.chunking_factory import ChunkingFactory
from core.factories.embedding_factory import EmbeddingFactory
from core.factories.vector_store_factory import VectorStoreFactory
from core.factories.retrieval_factory import RetrievalFactory
from core.metadata_models import ChunkMetadata

logger = logging.getLogger("DocumentPipeline")

class DocumentPipeline:
    """
    Orchestrates document processing workflows.

    Responsibilities:
    - Instantiate components via factories
    - Execute chunking → embedding → storage workflow
    - Multi-strategy retrieval orchestration
    - Metadata attachment and lifecycle management
    """

    def __init__(self, domain_config: Any):
        """Initialize pipeline with domain config."""
        self.config = domain_config

        # Create embedding model first (needed for dimension)
        self.embedding_model = EmbeddingFactory.create_embedder(
            self.config.embedding
        )

        # Create chunker
        self.chunker = ChunkingFactory.create_chunker(
            self.config.chunking,
            embedding_model_name=self.embedding_model.model_name
        )

        # Fixed metadata fields
        self.metadata_fields = [
            "doc_id", "chunk_id", "title", "author", "domain",
            "doc_type", "tags", "upload_timestamp", "version",
            "deprecated", "deprecated_date", "deprecation_reason",
            "source_file", "source_file_hash", "uploader_id",
            "embedding_model_name", "embedding_dimension",
            "chunking_strategy", "chunking_params",
            "authority_level", "review_status"
        ]

        # Create vector store with dimension and metadata schema
        self.vector_store = VectorStoreFactory.create_store(
            self.config.vectorstore,
            embedding_dimension=self.embedding_model.embedding_dimension,
            metadata_fields=self.metadata_fields
        )

        # Initialize retrieval strategies
        self.retrieval_strategies = self._init_retrieval_strategies()

        logger.info(f"DocumentPipeline initialized for domain: {self.config.name}")

    def _init_retrieval_strategies(self) -> Dict[str, Any]:
        """Create all configured retrieval strategies."""
        retrieval_cfg = getattr(self.config, "retrieval", {})
        strategies = retrieval_cfg.get("strategies", ["vector_similarity"])

        retrievers = {}
        for strat_name in strategies:
            strat_cfg = retrieval_cfg.get(strat_name, {})
            retriever = RetrievalFactory.create_retriever(
                strat_cfg,
                vector_store=self.vector_store,
                embedding_model=self.embedding_model
            )
            retrievers[strat_name] = retriever
            logger.info(f"Loaded retrieval strategy: {strat_name}")

        return retrievers

    def process_document(
        self,
        doc: Any,
        metadata: Dict[str, Any],
        replace_existing: bool = False
    ) -> Dict[str, Any]:
        """
        Process document end-to-end: chunk → embed → store.

        Args:
            doc: Document file or text
            metadata: Enriched metadata from service layer
            replace_existing: Delete existing before ingestion

        Returns:
            Processing summary
        """
        doc_id = metadata.get("doc_id")
        if not doc_id:
            raise ValueError("metadata must contain 'doc_id'")

        logger.info(f"Processing document: {doc_id}")

        # Delete existing if requested
        if replace_existing:
            self.vector_store.delete_document(doc_id)
            logger.info(f"Deleted existing document: {doc_id}")

        # Step 1: Chunk document
        chunks = self.chunker.chunk_document(doc)
        logger.info(f"Chunked into {len(chunks)} chunks")

        # Step 2: Extract text and embed
        texts = [c.text for c in chunks]
        embeddings = self.embedding_model.embed_texts(texts)
        logger.info(f"Generated embeddings for {len(embeddings)} chunks")

        # Step 3: Build chunk metadata
        chunk_metadatas = []
        for i, chunk in enumerate(chunks):
            chunk_meta = {
                # Identity
                "doc_id": doc_id,
                "chunk_id": f"{doc_id}_chunk_{i}",
                "chunk_text": chunk.text,

                # From upload metadata
                "title": metadata.get("title"),
                "author": metadata.get("author"),
                "domain": metadata.get("domain"),
                "doc_type": metadata.get("doc_type"),
                "tags": metadata.get("tags", []),
                "uploader_id": metadata.get("uploader_id"),
                "upload_timestamp": metadata.get("upload_timestamp"),
                "source_file": metadata.get("source_file"),
                "source_file_hash": metadata.get("source_file_hash"),
                "version": metadata.get("version", "1.0"),

                # From chunk
                "page_num": getattr(chunk, "page_num", None),
                "char_range": getattr(chunk, "char_range", None),

                # Processing info
                "embedding_model_name": self.embedding_model.model_name,
                "embedding_dimension": self.embedding_model.embedding_dimension,
                "embedding_version": self.embedding_model.model_name,
                "chunking_strategy": self.config.chunking.strategy,
                "chunking_params": {
                    "chunk_size": getattr(self.config.chunking, "chunk_size", None),
                    "overlap": getattr(self.config.chunking, "overlap", None)
                },
                "processing_timestamp": datetime.utcnow(),

                # Lifecycle
                "deprecated": False,
                "deprecated_date": None,
                "deprecation_reason": None,

                # Quality
                "authority_level": metadata.get("authority_level", "draft"),
                "review_status": metadata.get("review_status", "pending"),
            }
            chunk_metadatas.append(chunk_meta)

        # Step 4: Upsert to vector store
        self.vector_store.upsert_bulk(
            ids=[m["chunk_id"] for m in chunk_metadatas],
            embeddings=embeddings,
            metadatas=chunk_metadatas,
            documents=texts
        )
        logger.info(f"Upserted {len(chunk_metadatas)} chunks for {doc_id}")

        return {
            "doc_id": doc_id,
            "chunks_ingested": len(chunk_metadatas),
            "status": "success"
        }

    def deprecate_document(
        self,
        doc_id: str,
        deprecated_date: datetime,
        reason: str
    ) -> None:
        """Mark all chunks for doc_id as deprecated."""
        logger.info(f"Deprecating document: {doc_id}")

        self.vector_store.update_metadata(
            filter={"doc_id": doc_id},
            updates={
                "deprecated": True,
                "deprecated_date": deprecated_date,
                "deprecation_reason": reason
            }
        )
        logger.info(f"Document {doc_id} marked as deprecated")

    def update_document_metadata(
        self,
        doc_id: str,
        updates: Dict[str, Any]
    ) -> None:
        """Update metadata for all chunks of a document."""
        logger.info(f"Updating metadata for {doc_id}: {updates}")

        self.vector_store.update_metadata(
            filter={"doc_id": doc_id},
            updates=updates
        )
        logger.info(f"Metadata updated for {doc_id}")

    def query(
        self,
        query_text: str,
        strategy_name: Optional[str] = None,
        metadata_filters: Optional[Dict[str, Any]] = None,
        top_k: Optional[int] = None
    ) -> List[Dict[str, Any]]:
        """
        Execute query using configured retrieval strategies.

        If strategy_name is None, uses all strategies and merges results.
        """
        if not strategy_name:
            # Use all strategies
            all_results = []
            for name, retriever in self.retrieval_strategies.items():
                results = retriever.retrieve(
                    query_text,
                    metadata_filters=metadata_filters,
                    top_k=top_k
                )
                # Tag with strategy for transparency
                for r in results:
                    r["retrieval_strategy"] = name
                all_results.extend(results)

            # Optional: deduplicate and re-rank
            all_results = self._deduplicate_results(all_results)
            logger.info(f"Multi-strategy query returned {len(all_results)} results")
            return all_results[:top_k] if top_k else all_results
        else:
            # Use specific strategy
            retriever = self.retrieval_strategies.get(strategy_name)
            if not retriever:
                raise ValueError(f"Retrieval strategy '{strategy_name}' not found")

            results = retriever.retrieve(
                query_text,
                metadata_filters=metadata_filters,
                top_k=top_k
            )
            logger.info(f"Query with '{strategy_name}' returned {len(results)} results")
            return results

    def _deduplicate_results(self, results: List[Dict]) -> List[Dict]:
        """Remove duplicate chunks; keep highest score."""
        seen = {}
        for r in results:
            chunk_id = r.get("chunk_id")
            if chunk_id not in seen or r["score"] > seen[chunk_id]["score"]:
                seen[chunk_id] = r
        return sorted(seen.values(), key=lambda x: x["score"], reverse=True)

    def get_document_info(self, doc_id: str) -> Dict[str, Any]:
        """Fetch document metadata and stats."""
        return self.vector_store.get_document_info(doc_id)

    def list_documents(self, filters: Optional[Dict] = None) -> List[Dict]:
        """List all documents with optional filters."""
        return self.vector_store.list_documents(filters)
```

8. Hybrid Retrieval Implementation

8.1 Why Hybrid Retrieval?

Problem with Pure Vector Search:

- Misses exact keyword matches
- Poor on domain-specific jargon, acronyms, identifiers
- Less effective on precise terminology (e.g., "Form W-2" vs semantically similar but wrong forms)

Problem with Pure Keyword Search:

- Misses semantic similarity
- Requires exact wording
- No understanding of synonyms or concepts

Solution: Hybrid = Dense (Vector) + Sparse (Keyword)

8.2 Architecture

```
User Query: "What is the vacation policy?"
        ↓
┌─────────────────────────────────────────────────┐
│  DENSE SEARCH (Semantic)                        │
│  - Embed query with sentence-transformers       │
│  - Vector similarity search                     │
│  - Returns top-N with cosine scores             │
└──────────────┬──────────────────────────────────┘
             │ Dense Results: {"doc1": 0.89, "doc5": 0.75, ...}
             │
┌──────────────▼──────────────────────────────────┐
│  SPARSE SEARCH (Keyword)                        │
│  - Tokenize query                               │
│  - BM25 scoring against corpus                  │
│  - Returns top-N with BM25 scores               │
└──────────────┬──────────────────────────────────┘
             │ Sparse Results: {"doc1": 12.3, "doc3": 9.1, ...}
             │
┌──────────────▼──────────────────────────────────┐
│  SCORE NORMALIZATION                            │
│  - Normalize dense: 0-1 range                   │
│  - Normalize sparse: 0-1 range (min-max)        │
```

```
└──────────────┬───────────────────────────────────────┘
               │
┌──────────────▼───────────────────────────────────────┐
│  HYBRID COMBINATION                            │
│  final_score = alpha * dense + (1-alpha) * sparse │
│  alpha = 0.7 (configurable)                    │
└──────────────┬───────────────────────────────────────┘
               │
┌──────────────▼───────────────────────────────────────┐
│  RE-RANK & FILTER                              │
│  - Sort by final_score                         │
│  - Apply metadata filters                      │
│  - Return top-k                                │
└──────────────────────────────────────────────────────┘
```

8.3 BM25 Implementation

```python
"""
core/retrievals/bm25_retrieval.py

BM25 sparse keyword retrieval for hybrid search.
"""

import logging
from rank_bm25 import BM25Okapi
from typing import List, Dict, Any
import numpy as np

logger = logging.getLogger("BM25Retrieval")

class BM25Retrieval:
    """
    BM25 (Best Match 25) sparse retrieval.

    Good for:
    - Exact keyword matching
    - Domain-specific terminology
    - Acronyms and identifiers
    - Short queries with specific terms
    """

    def __init__(self, corpus: List[str], doc_ids: List[str]):
        """
        Initialize BM25 index.

        Args:
            corpus: List of document texts
            doc_ids: Corresponding document/chunk IDs
        """
        self.doc_ids = doc_ids
        self.tokenize = lambda text: text.lower().split()
        tokenized_corpus = [self.tokenize(doc) for doc in corpus]
        self.bm25 = BM25Okapi(tokenized_corpus)
        logger.info(f"BM25 index built with {len(corpus)} documents")
    def search(self, query: str, top_k: int = 10) -> List[Dict[str, Any]]:
        """
        Search using BM25.

        Returns:
            List of {doc_id, score} sorted by score descending
        """
        tokenized_query = self.tokenize(query)
        scores = self.bm25.get_scores(tokenized_query)

        # Get top-k indices
        top_indices = np.argsort(scores)[-top_k:][::-1]

        results = [
            {"doc_id": self.doc_ids[i], "score": float(scores[i])}
            for i in top_indices
        ]

        logger.debug(f"BM25 search returned {len(results)} results")
        return results

    def normalize_scores(self, scores: List[float]) -> List[float]:
        """Min-max normalization to 0-1 range."""
        if not scores:
            return []
        min_score = min(scores)
        max_score = max(scores)
        if max_score == min_score:
            return [1.0] * len(scores)
        return [(s - min_score) / (max_score - min_score) for s in scores]
```

8.4 Hybrid Retrieval Implementation

```
"""
core/retrievals/hybrid_retrieval.py

Combines dense (vector) and sparse (BM25) retrieval with alpha weighting.
"""

import logging
from typing import List, Dict, Any, Optional

logger = logging.getLogger("HybridRetrieval")

class HybridRetrieval:
    """
    Hybrid retrieval combining dense and sparse search.

    Formula:
        final_score = alpha * dense_score + (1 - alpha) * sparse_score

    Where:
        - alpha ∈ [0, 1]: weight parameter
        - alpha = 1.0: pure dense (semantic)
        - alpha = 0.0: pure sparse (keyword)
        - alpha = 0.7: recommended balanced setting
    """

    def __init__(
        self,
        vector_store: Any,
        embedding_model: Any,
        bm25_index: Any,
        alpha: float = 0.7,
        normalize_scores: bool = True
    ):
        """
        Initialize hybrid retrieval.

        Args:
            vector_store: Dense vector store
            embedding_model: Embedding model for query
            bm25_index: BM25 index for sparse search
            alpha: Weighting parameter (0-1)
            normalize_scores: Whether to normalize before combining
        """
        self.vector_store = vector_store
        self.embedding_model = embedding_model
        self.bm25_index = bm25_index
        self.alpha = alpha
        self.normalize_scores = normalize_scores

        logger.info(f"HybridRetrieval initialized with alpha={alpha}")

    def retrieve(
        self,
        query_text: str,
        metadata_filters: Optional[Dict[str, Any]] = None,
        top_k: int = 10
    ) -> List[Dict[str, Any]]:
        """
        Execute hybrid retrieval.

        Steps:
        1. Dense search (vector similarity)
        2. Sparse search (BM25)
        3. Normalize scores
        4. Combine with alpha weighting
        5. Sort and return top-k
        """
        logger.info(f"Hybrid retrieval: query='{query_text}', alpha={self.alpha}, top_k={

        # Step 1: Dense search
        query_embedding = self.embedding_model.embed_texts([query_text])[0]
        dense_results = self.vector_store.search(
            query_embedding,
            top_k=top_k * 2,  # Over-fetch for better recall
            filters=metadata_filters
        )
        dense_dict = {r["chunk_id"]: r["score"] for r in dense_results}
        logger.debug(f"Dense search returned {len(dense_results)} results")
        # Step 2: Sparse search
        sparse_results = self.bm25_index.search(query_text, top_k=top_k * 2)
        sparse_dict = {r["doc_id"]: r["score"] for r in sparse_results}
        logger.debug(f"Sparse search returned {len(sparse_results)} results")

        # Step 3: Normalize scores
        if self.normalize_scores:
            dense_scores = list(dense_dict.values())
            sparse_scores = list(sparse_dict.values())

            dense_dict = self._normalize_dict(dense_dict)
            sparse_dict = self._normalize_dict(sparse_dict)

        # Step 4: Combine scores
        all_doc_ids = set(dense_dict.keys()) | set(sparse_dict.keys())
        combined = {}

        for doc_id in all_doc_ids:
            dense_score = dense_dict.get(doc_id, 0.0)
            sparse_score = sparse_dict.get(doc_id, 0.0)
            combined[doc_id] = self.alpha * dense_score + (1 - self.alpha) * sparse_score

        # Step 5: Sort and fetch documents
        ranked = sorted(combined.items(), key=lambda x: x[1], reverse=True)[:top_k]

        # Fetch full documents with metadata
        results = []
        for chunk_id, score in ranked:
            doc = self.vector_store.get_document(chunk_id)
            if doc:
                results.append({
                    "chunk_id": chunk_id,
                    "chunk_text": doc.get("chunk_text"),
                    "score": score,
                    "metadata": doc.get("metadata"),
                    "dense_score": dense_dict.get(chunk_id, 0.0),
                    "sparse_score": sparse_dict.get(chunk_id, 0.0)
                })

        logger.info(f"Hybrid retrieval returned {len(results)} results")
        return results

    def _normalize_dict(self, score_dict: Dict[str, float]) -> Dict[str, float]:
        """Min-max normalize scores to 0-1 range."""
        scores = list(score_dict.values())
        if not scores:
            return {}

        min_score = min(scores)
        max_score = max(scores)

        if max_score == min_score:
            return {k: 1.0 for k in score_dict}

        return {
            k: (v - min_score) / (max_score - min_score)
            for k, v in score_dict.items()
        }
```

8.5 Alpha Tuning Guidelines

Alpha Value	Behavior	Use Case
1.0	Pure semantic (dense only)	Conceptual questions, paraphrased queries
0.9	Heavy semantic, light keyword	General knowledge questions
0.7	**Recommended balanced**	Most production use cases
0.5	Equal dense + sparse	Hybrid queries with technical terms
0.3	Keyword-heavy	Specific terminology, codes, identifiers
0.0	Pure keyword (sparse only)	Exact phrase matching, acronyms

Tuning Process:

1. Start with alpha=0.7
2. Run Golden QA sets
3. Measure Recall@K and MRR
4. Adjust alpha based on domain:
    - Legal/Compliance → lower alpha (more keyword)
    - HR/General knowledge → higher alpha (more semantic)
5. A/B test in production

9. Factory Layer Enhancements

9.1 Retrieval Factory with Hybrid Support

```
"""
core/factories/retrieval_factory.py

Enhanced to support hybrid retrieval.
"""

import logging
from typing import Any

from core.retrievals.vector_similarity_retrieval import VectorSimilarityRetrieval
from core.retrievals.hybrid_retrieval import HybridRetrieval
from core.retrievals.bm25_retrieval import BM25Retrieval

logger = logging.getLogger("RetrievalFactory")

class RetrievalFactory:
    """Create retrieval strategies from config."""

    _available_strategies = {
        "vector_similarity": VectorSimilarityRetrieval,
        "hybrid": HybridRetrieval,
        # Add more as needed
    }

    @staticmethod
    def create_retriever(
        config: Any,
        vector_store: Any,
        embedding_model: Any,
        bm25_index: Any = None
    ) -> Any:
        """
        Create retriever based on config.

        For hybrid strategy, requires bm25_index.
        """
        strategy = getattr(config, "strategy", "vector_similarity")
        strategy = strategy.lower()

        logger.info(f"Creating retriever: {strategy}")

        retriever_cls = RetrievalFactory._available_strategies.get(strategy)
        if not retriever_cls:
            raise ValueError(f"Unknown retrieval strategy: {strategy}")

        if strategy == "hybrid":
            if not bm25_index:
                # Build BM25 index from vector store corpus
                corpus, doc_ids = vector_store.get_all_documents()
                bm25_index = BM25Retrieval(corpus, doc_ids)

            alpha = getattr(config, "alpha", 0.7)
            normalize = getattr(config, "normalize_scores", True)

            return HybridRetrieval(
                vector_store=vector_store,
                embedding_model=embedding_model,
                bm25_index=bm25_index,
                alpha=alpha,
                normalize_scores=normalize
            )
        else:
            # Vector similarity or others
            top_k = getattr(config, "top_k", 10)
            return retriever_cls(
                vector_store=vector_store,
                embedding_model=embedding_model,
                top_k=top_k
            )
```

10. File Processing & Validation

10.1 Validation Utilities

```python
"""
core/utils/validation.py

Validation functions for service layer.
"""

from typing import List, Set, Dict, Any

class ValidationError(Exception):
    """Raised when validation fails."""
    pass

def validate_file_type(filename: str, allowed_types: Set[str]) -> bool:
    """
    Check if file extension is allowed.

    Args:
        filename: File name with extension
        allowed_types: Set of allowed extensions (lowercase)

    Raises:
        ValidationError: If file type not allowed

    Returns:
        True if valid
    """
    ext = filename.rsplit('.', 1)[-1].lower() if '.' in filename else ''

    if ext not in allowed_types:
        raise ValidationError(
            f"File type '.{ext}' not allowed. "
            f"Allowed types: {', '.join(sorted(allowed_types))}"
        )
    return True

def validate_metadata(metadata: Dict[str, Any], required_fields: List[str]) -> bool:
    """
    Check if required metadata fields are present and non-empty.

    Args:
        metadata: Metadata dict
        required_fields: List of required field names

    Raises:
        ValidationError: If any required field missing or empty

    Returns:
        True if valid
    """
    missing = []
    empty = []
    for field in required_fields:
        if field not in metadata:
            missing.append(field)
        elif not metadata[field]:  # None, empty string, empty list
            empty.append(field)

    if missing:
        raise ValidationError(f"Missing required metadata fields: {', '.join(missing)}")
    if empty:
        raise ValidationError(f"Empty required metadata fields: {', '.join(empty)}")

    return True

def validate_file_size(file_size: int, max_size_mb: int) -> bool:
    """
    Check if file size is within limit.

    Args:
        file_size: File size in bytes
        max_size_mb: Maximum allowed size in MB

    Raises:
        ValidationError: If file too large

    Returns:
        True if valid
    """
    max_bytes = max_size_mb * 1024 * 1024

    if file_size > max_bytes:
        raise ValidationError(
            f"File size {file_size / (1024*1024):.2f} MB exceeds "
            f"maximum allowed size of {max_size_mb} MB"
        )

    return True
```

10.2 File Hashing Utility

```
"""
core/utils/hashing.py

File hash computation for provenance tracking.
"""

import hashlib
import logging

logger = logging.getLogger("FileHashing")

def compute_file_hash(file_obj: Any, algorithm: str = "sha256") -> str:
    """
    Compute cryptographic hash of file for integrity verification.
    Args:
        file_obj: File object with .read() method
        algorithm: Hash algorithm (sha256, sha1, md5)

    Returns:
        Hexadecimal hash string
    """
    if algorithm == "sha256":
        hash_obj = hashlib.sha256()
    elif algorithm == "sha1":
        hash_obj = hashlib.sha1()
    elif algorithm == "md5":
        hash_obj = hashlib.md5()
    else:
        raise ValueError(f"Unsupported hash algorithm: {algorithm}")

    # Reset file pointer if possible
    if hasattr(file_obj, 'seek'):
        file_obj.seek(0)

    # Read in chunks for large files
    for chunk in iter(lambda: file_obj.read(4096), b""):
        hash_obj.update(chunk)

    # Reset file pointer again for subsequent reads
    if hasattr(file_obj, 'seek'):
        file_obj.seek(0)

    hash_hex = hash_obj.hexdigest()
    logger.debug(f"Computed {algorithm} hash: {hash_hex[:16]}...")

    return hash_hex
```

10.3 Enhanced PDF Processor

```
"""
utils/fileparsers/pdf_processor.py

Enhanced PDF processor with metadata extraction.
"""

import logging
from typing import Dict, Any, List
import PyPDF2

from core.utils.hashing import compute_file_hash

logger = logging.getLogger("PDFProcessor")

class PDFProcessor:
    """Process PDF files and extract text with metadata."""

    def extract(self, file_path: str) -> Dict[str, Any]:
        """
        Extract text and metadata from PDF.
    Returns:
        {
            "text": str (full text),
            "metadata": {
                "page_count": int,
                "file_hash": str,
                "pages": List[dict] per-page info
            }
        }
    """
    logger.info(f"Processing PDF: {file_path}")

    try:
        with open(file_path, 'rb') as f:
            reader = PyPDF2.PdfReader(f)

            # Extract text and track page info
            full_text = []
            pages = []
            char_offset = 0

            for page_num, page in enumerate(reader.pages, start=1):
                page_text = page.extract_text()
                full_text.append(page_text)

                pages.append({
                    "page_num": page_num,
                    "char_range": (char_offset, char_offset + len(page_text)),
                    "text_length": len(page_text)
                })
                char_offset += len(page_text)

            # Compute file hash
            f.seek(0)
            file_hash = compute_file_hash(f)

            result = {
                "text": "\n".join(full_text),
                "metadata": {
                    "page_count": len(reader.pages),
                    "file_hash": file_hash,
                    "pages": pages
                }
            }

            logger.info(f"PDF processed: {len(reader.pages)} pages, {len(result['text
            return result

    except Exception as e:
        logger.exception(f"Failed to process PDF: {file_path}")
        raise
```

11. Implementation Roadmap

Phase 2.1: Foundation (Week 1)

Task 1.1: Enhanced Metadata Models

File: `core/metadata_models.py`

Deliverables:

- Complete Pydantic model with all Phase 2 fields
- Validation methods
- Example schemas

Acceptance Criteria:

- [ ] All metadata fields defined with types
- [ ] Pydantic validation working
- [ ] Unit tests for validation pass
- [ ] Documentation with examples

Estimated Time: 2 days

Task 1.2: Create Service Layer

File: `core/services/document_service.py`

Deliverables:

- Complete DocumentService class
- All public API methods
- File type and metadata validation
- Error handling

Acceptance Criteria:

- [ ] Service instantiates per domain
- [ ] upload_document() functional
- [ ] query() functional
- [ ] deprecate_document() functional
- [ ] Unit tests cover all methods
- [ ] Zero business logic remains in UI

Estimated Time: 3 days

Task 1.3: Enhance Pipeline

File: `core/pipeline/document_pipeline.py`

Deliverables:

- Multi-strategy retrieval support
- Enhanced metadata attachment
- Deprecation workflow
- Improved logging

Acceptance Criteria:

- [ ] Supports multiple retrieval strategies
- [ ] Metadata properly attached to chunks
- [ ] Deprecation API works
- [ ] Integration tests pass

Estimated Time: 3 days

Phase 2.2: Hybrid Retrieval (Week 2)

Task 2.1: Implement BM25 Retrieval

File: `core/retrievals/bm25_retrieval.py`

Deliverables:

- BM25Retrieval class
- Indexing and search methods
- Score normalization

Acceptance Criteria:

- [ ] BM25 index builds from corpus
- [ ] Search returns ranked results
- [ ] Scores normalized correctly
- [ ] Unit tests pass

Estimated Time: 2 days

Task 2.2: Implement Hybrid Retrieval

File: `core/retrievals/hybrid_retrieval.py`

Deliverables:

- HybridRetrieval class
- Score combination logic
- Configurable alpha parameter

Acceptance Criteria:

- [ ] Combines dense and sparse correctly
- [ ] Alpha parameter adjustable
- [ ] Results properly ranked
- [ ] Integration test shows improvement over pure vector

Estimated Time: 2 days

Task 2.3: Update Retrieval Factory

File: `core/factories/retrieval_factory.py`

Deliverables:

- Hybrid strategy registration
- BM25 instantiation logic

Acceptance Criteria:

- [ ] Factory creates hybrid retriever from config
- [ ] All parameters passed correctly
- [ ] Unit tests pass

Estimated Time: 1 day

Task 2.4: Alpha Tuning & Evaluation

Deliverables:

- Test hybrid with alpha values: 0.3, 0.5, 0.7, 0.9
- Golden QA evaluation
- Optimal alpha per domain

Acceptance Criteria:

- [ ] Hybrid outperforms pure vector by ≥15%
- [ ] Optimal alpha identified per domain

- [ ] Documentation updated with recommendations

Estimated Time: 2 days

Phase 2.3: File Processing & Validation (Week 3)

Task 3.1: Validation Utilities

File: `core/utils/validation.py`, `core/utils/hashing.py`

Deliverables:

- validate_file_type()
- validate_metadata()
- validate_file_size()
- compute_file_hash()

Acceptance Criteria:

- [ ] All validation functions working
- [ ] Clear error messages
- [ ] Unit tests cover edge cases
- [ ] Used by service layer

Estimated Time: 1 day

Task 3.2: Enhance File Parsers

Files:

- `utils/fileparsers/pdf_processor.py`
- `utils/fileparsers/docx_processor.py`
- `utils/fileparsers/txt_processor.py`

Deliverables:

- Extract page numbers
- Extract character ranges
- Compute file hashes
- Better error handling

Acceptance Criteria:

- [ ] Page metadata extracted
- [ ] File hashes computed
- [ ] Errors handled gracefully

- [ ] Integration tests pass

Estimated Time: 2 days

Task 3.3: Refactor UI Layer

File: `app.py`

Deliverables:

- Remove all business logic
- Replace with service API calls
- Thin handlers (<20 lines each)

Acceptance Criteria:

- [ ] Zero business logic in app.py
- [ ] All handlers call service methods
- [ ] No direct factory or pipeline imports
- [ ] Code review passes

Estimated Time: 2 days

Phase 2.4: Testing & Quality (Week 4)

Task 4.1: Unit Tests

Directory: `tests/unit/`

Deliverables:

- Factory tests
- Service layer tests
- Pipeline tests
- Retrieval tests
- Validation tests

Acceptance Criteria:

- [ ] 80%+ code coverage
- [ ] All critical paths tested
- [ ] CI/CD integration
- [ ] Test documentation

Estimated Time: 3 days

Task 4.2: Integration Tests

Directory: `tests/integration/`

Deliverables:

- End-to-end ingestion test
- End-to-end query test
- Multi-strategy retrieval test
- Deprecation workflow test

Acceptance Criteria:

- [ ] All workflows tested end-to-end
- [ ] Tests use real config files
- [ ] Cleanup after each test
- [ ] CI/CD integration

Estimated Time: 2 days

Task 4.3: Golden QA Sets

Directory: `tests/golden_qa/`

Deliverables:

- Golden QA sets for 2+ domains
- Evaluation script
- Baseline metrics
- Comparison: pure vector vs hybrid

Acceptance Criteria:

- [ ] QA sets created with expected answers
- [ ] Automated evaluation working
- [ ] Metrics logged: Recall@K, MRR
- [ ] Hybrid shows improvement

Estimated Time: 2 days

12. Configuration Management

12.1 Global Config (Enhanced)

```yaml
# configs/global_config.yaml
name: global_default
description: Global defaults for multi-domain RAG system (Phase 2)

chunking:
  strategy: recursive
  chunk_size: 500
  overlap: 50

embedding:
  provider: sentence_transformers
  model_name: all-MiniLM-L6-v2
  device: cpu
  batch_size: 32
  normalize_embeddings: true

retrieval:
  strategies:
    - hybrid
    - filtering
  hybrid:
    alpha: 0.7
    dense_provider: sentence_transformers
    sparse_provider: bm25
    normalize_scores: true
  filtering:
    fields: ["domain", "deprecated", "doc_type", "authority_level"]
    default_filters:
      deprecated: false
  top_k: 10

vectorstore:
  provider: chromadb
  index_type: hnsw
  collection_name: default_collection
  persist_directory: ./data/chromadb

metadata:
  track_versions: true
  enable_deprecation: true
  compute_file_hash: true
  extract_page_numbers: true
  required_fields:
    - doc_id
    - title
    - domain
    - doc_type
    - uploader_id
    - upload_date

security:
  allowed_file_types:
    - pdf
    - docx
    - txt
  max_file_size_mb: 20
  require_authentication: false

logging:
  level: INFO
  format: "%(asctime)s - %(name)s - %(levelname)s - %(message)s"
  log_file: ./logs/rag_system.log
  enable_structured_logging: true
```

12.2 Domain Config Example

```
# configs/domains/hr_domain.yaml
name: hr_domain
description: HR policies and procedures

extends: global_config

retrieval:
  strategies:
    - hybrid
  hybrid:
    alpha: 0.8  # More semantic for HR queries
  top_k: 10

metadata:
  required_fields:
    - doc_id
    - title
    - domain
    - doc_type
    - uploader_id
    - upload_date
    - authority_level
```

13. Testing Strategy

13.1 Unit Testing

Coverage Target: ≥ 80%

Test Structure:

```
tests/unit/
├── test_services/
│   └── test_document_service.py
├── test_pipeline/
│   └── test_document_pipeline.py
├── test_factories/
│   ├── test_chunking_factory.py
│   ├── test_embedding_factory.py
│   ├── test_retrieval_factory.py
│   └── test_vector_store_factory.py
├── test_retrievals/
│   ├── test_bm25_retrieval.py
│   └── test_hybrid_retrieval.py
└── test_utils/
    ├── test_validation.py
    └── test_hashing.py
```

Example Unit Test:

```
# tests/unit/test_services/test_document_service.py

import pytest
from core.services.document_service import DocumentService, ValidationError

def test_upload_document_validates_file_type():
    """Service should reject disallowed file types."""
    service = DocumentService("hr")

    # Mock file with disallowed extension
    class MockFile:
        name = "doc.exe"

    metadata = {
        "doc_id": "test123",
        "title": "Test",
        "domain": "hr",
        "doc_type": "policy",
        "uploader_id": "user1"
    }

    with pytest.raises(ValidationError, match="not allowed"):
        service.upload_document(MockFile(), metadata)

def test_query_filters_deprecated_by_default():
    """Service should filter deprecated docs unless explicitly requested."""
    service = DocumentService("hr")

    # Mock pipeline query
    service.pipeline.query = lambda *args, **kwargs: kwargs.get("metadata_filters", {})

    result = service.query("test query")

    # Should have deprecated=False in filters
    assert result.get("deprecated") == False
```

13.2 Integration Testing

Test Scenarios:

1. End-to-end ingestion
2. End-to-end query

3. Multi-strategy retrieval
4. Document lifecycle (upload → query → deprecate → verify filtered)
5. Metadata update workflow

Example Integration Test:

```
# tests/integration/test_end_to_end.py

import pytest
from core.services.document_service import DocumentService

def test_full_ingestion_query_workflow():
    """Test complete workflow from upload to query."""
    service = DocumentService("test_domain")

    # Upload document
    with open("tests/fixtures/sample.pdf", "rb") as f:
        metadata = {
            "doc_id": "test_doc_001",
            "title": "Test Document",
            "domain": "test_domain",
            "doc_type": "policy",
            "uploader_id": "test_user"
        }
        result = service.upload_document(f, metadata)

    assert result["status"] == "success"
    assert result["chunks_ingested"] > 0

    # Query document
    results = service.query("test query", top_k=5)

    assert len(results) > 0
    assert results[0]["metadata"]["doc_id"] == "test_doc_001"

    # Cleanup
    service.vector_store.delete_document("test_doc_001")
```

13.3 Golden QA Sets

Structure:

```
# tests/golden_qa/hr_domain_qa.yaml
domain: hr
description: Golden QA set for HR policies

questions:
  - id: hr_001
    question: "How many vacation days do employees get?"
    expected_doc_ids: ["HR-POLICY-2025-001"]
    expected_chunks: ["HR-POLICY-2025-001_chunk_5", "HR-POLICY-2025-001_chunk_6"]
    expected_keywords: ["15 days", "annual leave", "vacation"]
    min_recall_at_5: 1.0
    min_mrr: 0.8

  - id: hr_002
    question: "What is the sick leave policy?"
    expected_doc_ids: ["HR-POLICY-2025-001"]
    expected_chunks: ["HR-POLICY-2025-001_chunk_12"]
    expected_keywords: ["sick leave", "10 days", "medical certificate"]
    min_recall_at_5: 1.0
    min_mrr: 0.9
```

Evaluation Script:

```
# tests/golden_qa/evaluate.py

import yaml
from core.services.document_service import DocumentService

def evaluate_golden_qa(qa_file: str, service: DocumentService):
    """Evaluate retrieval against Golden QA set."""
    with open(qa_file) as f:
        qa_data = yaml.safe_load(f)

    results = []

    for qa in qa_data["questions"]:
        query_results = service.query(qa["question"], top_k=10)

        # Compute metrics
        recall_at_5 = compute_recall(
            retrieved=[r["chunk_id"] for r in query_results[:5]],
            expected=qa["expected_chunks"]
        )

        mrr = compute_mrr(
            retrieved=[r["chunk_id"] for r in query_results],
            expected=qa["expected_chunks"]
        )

        passed = (
            recall_at_5 >= qa["min_recall_at_5"] and
            mrr >= qa["min_mrr"]
        )

        results.append({
            "id": qa["id"],
            "question": qa["question"],
            "recall_at_5": recall_at_5,
            "mrr": mrr,
            "passed": passed
        })

    return results

def compute_recall(retrieved: list, expected: list) -> float:
    """Recall = (retrieved ∩ expected) / expected"""
    if not expected:
        return 1.0
    intersection = set(retrieved) & set(expected)
    return len(intersection) / len(expected)

def compute_mrr(retrieved: list, expected: list) -> float:
    """Mean Reciprocal Rank."""
    for i, chunk_id in enumerate(retrieved, start=1):
        if chunk_id in expected:
            return 1.0 / i
    return 0.0
```

14. CLI Tools

14.1 CLI Architecture

```
cli/
├── __init__.py
├── ingest.py      # Document ingestion
├── query.py       # Query execution
├── manage.py      # Domain and document management
└── evaluate.py    # Golden QA evaluation
```

14.2 Ingestion CLI

```
# cli/ingest.py

import click
from core.services.document_service import DocumentService

@click.command()
@click.option('--domain', required=True, help='Domain ID')
@click.option('--file', required=True, type=click.Path(exists=True), help='File path')
@click.option('--doc-id', required=True, help='Document ID')
@click.option('--title', required=True, help='Document title')
@click.option('--doc-type', required=True, help='Document type')
@click.option('--uploader-id', required=True, help='Uploader user ID')
@click.option('--replace', is_flag=True, help='Replace if exists')
def ingest(domain, file, doc_id, title, doc_type, uploader_id, replace):
    """Ingest a document into the RAG system."""
    click.echo(f"Ingesting document: {file} into domain: {domain}")

    service = DocumentService(domain)

    metadata = {
        "doc_id": doc_id,
        "title": title,
        "domain": domain,
        "doc_type": doc_type,
        "uploader_id": uploader_id
    }
```

```
with open(file, 'rb')
```