Technical Standards & Decision Log

MES/SCADA RAG System

ARCHITECTURAL DECISIONS RECORD (ADR)

ADR-001: Database Strategy

Date: 2025-01-XX
Status: DECIDED

Context: Potřebujeme kombinovat relační data (hierarchie, metadata) s vektorovými embeddings

Decision:

- Primary DB: PostgreSQL 15+ (relační data, metadata, audit)
- Vector DB: Qdrant (embeddings, similarity search)
- Cache: Redis (sessions, query cache)

Rationale:

- PostgreSQL: Mature, excellent JSON support, ACID compliance
- Qdrant: Production-ready, excellent metadata filtering, scalable
- Redis: Fast, proven session management

Consequences:

- ✅ Proven technology stack
- ✅ Good performance for expected scale
- ✅ Better metadata filtering than ChromaDB
- ❌ Multiple databases = complexity
- ❌ Qdrant requires separate service management

ADR-002: Vector DB Decision

Date: 2025-07-XX
Status: ✅ DECIDED

Context: Potřebujeme spolehlivou a škálovatelnou vektorovou databázi pro uložení embeddingů (např. texty, poznámky, biosignály) a provádění podobnostních dotazů v rámci AI funkcionalit (např. vyhledávání, doporučování, RAG).

Decision: Zvolená databáze: Qdrant

- Self-hosted, open-source vektorová DB

- REST API, gRPC, a Python-native SDK
- Podpora filtrování podle metadat
- Možnost nasazení lokálně i v cloudu

Rationale:

- 🟢 Snadné nasazení (Docker, binary)
- 🟢 Podpora metadatových filtrů a scoringu
- 🟢 Vhodné pro RAG, recommendation, personalizaci
- 🟢 Aktivní vývoj a dokumentace
- 🔴 Nutnost samostatného běhu služby (na rozdíl od ChromaDB)

Consequences:

- ✅ Robustní základ pro AI vyhledávání
- ✅ Možnost škálování podle potřeb
- ✅ Možnost snadné výměny embeddingů, retrain
- ❌ Potřeba správy dalšího serveru (Docker/service)

ADR-003: File Storage Strategy

Date: 2025-01-XX
Status: DECIDED

Context: Stovky GB dokumentů, různé formáty, backup requirements

Decision: Filesystem-based storage s organizovanou strukturou

```
/data/uploads/
  /{year}/     # 2025/
    /{month}/  # 01/
      /{hash}/ # abc123.../
        file.pdf      # original filename
        meta.json     # extracted metadata
```

Rationale:

- Jednoduchost implementace a debuggingu
- Standard filesystem backup tools
- Žádné vendor lock-in
- Easy migration k object storage později

Alternatives Considered:

- MinIO: Overkill pro MVP, ale good migration path
- Database BLOB storage: Performance issues při velkých souborech

ADR-004: Manufacturing Hierarchy Model

Date: 2025-01-XX
Status: DECIDED

Context: ISA-95 je standard, ale zákazníci potřebují flexibility

Decision: Flexibilní hierarchický model inspirovaný ISA-95

```sql
hierarchy_nodes:
  - id, name, code, description
  - parent_id (self-reference)
  - level_type (varchar, ne enum)
  - level_order (hierarchy position)
  - custom_attributes (JSON)
```

Rationale:

- ISA-95 jako default template, ale ne enforcement
- Zákazníci mohou definovat vlastní level types
- Zachována hierarchická struktura
- Extensible přes JSON attributes

ADR-005: Authentication Architecture

Date: 2025-01-XX
Status: DECIDED

Context: Enterprise SSO requirements + local fallback

Decision: Multi-provider authentication s JWT tokens

Implementation:

- JWT tokens (stateless)
- Redis session backing (revocation capability)
- Provider abstraction layer:
  - Local (username/password)
  - SAML 2.0 (enterprise SSO)
  - OIDC (modern SSO)

  - Active Directory (LDAP)

Rationale:

- Flexibility pro různé zákazníky
- Standard protocols
- Stateless token = scalability
- Session backing = security

ADR-006: RAG Implementation Strategy

Date: 2025-01-XX
Status: DECIDED

Context: Balance mezi accuracy, privacy, a cost

Decision: Local embeddings s Ollama + hierarchical context

Pipeline:

1. Document text extraction
2. Semantic chunking (paragraph-aware)
3. Local embeddings generation (Ollama)
4. Qdrant storage s metadata
5. Hybrid search (vector + keyword + hierarchy filter)

Model Selection: sentence-transformers/all-MiniLM-L6-v2 (default)

- Good multilingual support
- Reasonable size/performance
- Proven v RAG applications

CODING STANDARDS

Python Code Standards

Project Structure

```
src/
├── api/                  # FastAPI routes
│   ├── v1/
│   │   ├── hierarchy.py  # Manufacturing hierarchy endpoints
│   │   ├── documents.py  # Document management
│   │   ├── search.py     # RAG search endpoints
│   │   └── auth.py       # Authentication
│   └── dependencies.py   # Shared dependencies
├── core/                 # Business logic (domain layer)
│   ├── hierarchy/
│   ├── documents/
│   ├── search/
│   └── auth/
├── models/               # SQLAlchemy models
├── schemas/              # Pydantic models
├── services/             # External service integrations
├── utils/                # Utilities & helpers
└── config.py             # Configuration management
```

Naming Conventions

python

```python
# Variables & functions: snake_case
user_name = "john_doe"
def get_user_documents():
    pass

# Classes: PascalCase
class DocumentProcessor:
    pass

# Constants: UPPER_SNAKE_CASE
MAX_FILE_SIZE = 50 * 1024 * 1024  # 50MB

# Private methods: leading underscore
def _internal_helper():
    pass

# Database tables: plural snake_case
class HierarchyNode(Base):
    __tablename__ = "hierarchy_nodes"
```

Type Hints (MANDATORY)

```python
from typing import List, Optional, Dict, Any, Union
from pydantic import BaseModel

def process_document(
    file_path: str,
    metadata: Dict[str, Any],
    node_ids: List[int]
) -> Optional[Document]:
    """
    Process uploaded document and create database record.

    Args:
        file_path: Path to uploaded file
        metadata: Document metadata dictionary
        node_ids: List of hierarchy node IDs to associate

    Returns:
        Created Document instance or None on failure

    Raises:
        DocumentProcessingError: If file processing fails
        ValidationError: If metadata validation fails
    """
    pass
```

Error Handling Standards

python

```
# Custom exceptions
class DocumentProcessingError(Exception):
    """Raised when document processing fails"""
    def __init__(self, message: str, file_path: str, original_error: Exception = None):
        self.file_path = file_path
        self.original_error = original_error
        super().__init__(message)

# HTTP exception handling
from fastapi import HTTPException, status

@app.exception_handler(DocumentProcessingError)
async def document_processing_handler(request, exc):
    return HTTPException(
        status_code=status.HTTP_422_UNPROCESSABLE_ENTITY,
        detail={
            "message": str(exc),
            "file_path": exc.file_path,
            "type": "document_processing_error"
        }
    )
```

Database Operations

python

```
# Always use dependency injection
from sqlalchemy.orm import Session
from fastapi import Depends

def get_document_by_id(db: Session, document_id: int) -> Optional[Document]:
    """Get document by ID with error handling"""
    try:
        return db.query(Document).filter(Document.id == document_id).first()
    except SQLAlchemyError as e:
        logger.error(f"Database error fetching document {document_id}: {e}")
        raise DatabaseError(f"Failed to fetch document: {e}")

# Always close sessions properly
@contextmanager
def get_db_session() -> Session:
    db = SessionLocal()
    try:
        yield db
        db.commit()
    except Exception:
        db.rollback()
        raise
    finally:
        db.close()
```

Logging Standards

python

```python
import logging
import structlog

# Structured logging setup
logger = structlog.get_logger(__name__)

def process_document(document_id: int):
    logger.info(
        "Starting document processing",
        document_id=document_id,
        operation="process_document"
    )
    try:
        # processing logic
        logger.info(
            "Document processing completed",
            document_id=document_id,
            processing_time_ms=processing_time
        )
    except Exception as e:
        logger.error(
            "Document processing failed",
            document_id=document_id,
            error=str(e),
            operation="process_document"
        )
        raise
```