GraphRAG EMR API - Technical Documentation

Table of Contents

Overview

GraphRAG for EMR is a FastAPI-based REST API that provides graph-based Retrieval-Augmented Generation (RAG) capabilities for Electronic Medical Records (EMR). The system ingests patient JSON data files, stores them in a Neo4j graph database, and exposes APIs for querying and visualizing patient medical information as interconnected knowledge graphs.

Key Features

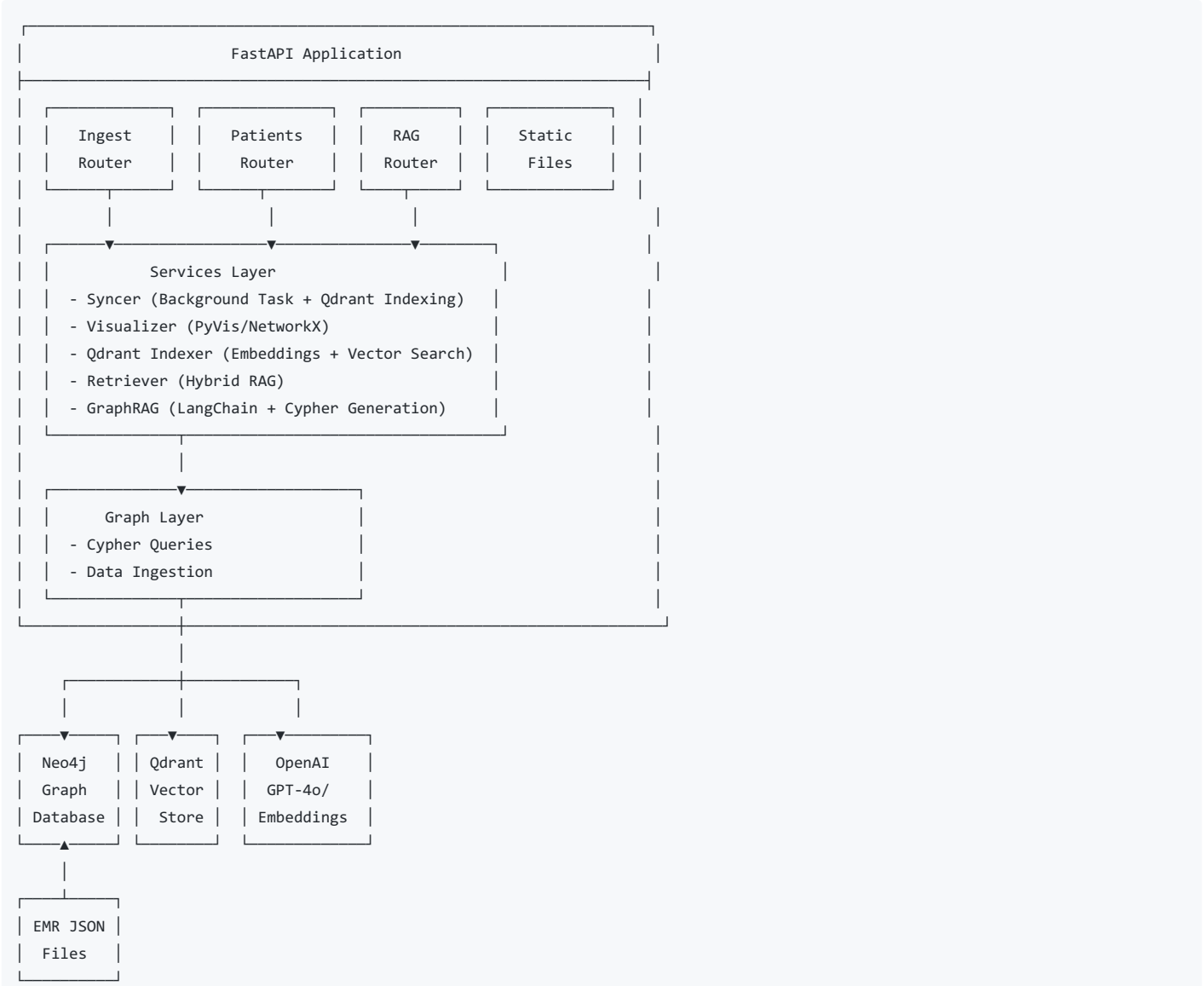
- **Automatic File Synchronization**: Monitors EMR directory and automatically ingests new/updated patient records
- **Graph Visualization**: Interactive HTML visualizations of patient medical histories using PyVis
- **RESTful API**: Query patient graphs as JSON or interactive HTML
- **Neo4j Integration**: Leverages graph database for complex medical relationship queries
- **RAG Question Answering**: Hybrid and graph-based AI-powered medical Q&A
- **Vector Search**: Qdrant-powered semantic search with OpenAI embeddings
- **LangChain Integration**: Automated Cypher query generation for complex questions
- **Async Support**: Built on FastAPI with async capabilities for high performance

Technology Stack

- **Framework**: FastAPI 0.115.x
- **Database**: Neo4j 5.x
- **Vector Store**: Qdrant
- **AI/ML**: OpenAI GPT-5, GPT-5-mini, LangChain
- **Visualization**: PyVis 0.3.x, NetworkX 3.x
- **Server**: Uvicorn 0.30.x with hot reload
- **Python**: 3.10+

Architecture

System Components



Directory Structure

```
graph_rag/
├── app/
│   ├── main.py              # FastAPI application entry point
│   ├── config.py            # Configuration and settings
│   ├── deps.py              # Dependency injection (Neo4j driver)
│   ├── models.py            # Pydantic data models
│   ├── graph/
│   │   ├── cypher.py        # Cypher query definitions
│   │   └── ingest.py        # Data ingestion logic
│   ├── routers/
│   │   ├── ingest.py        # Ingestion endpoints
│   │   ├── patients.py      # Patient query endpoints
│   │   └── rag.py           # RAG Q&A endpoints
│   └── services/
│       ├── syncer.py        # Background sync service
│       ├── visualize.py     # Graph visualization
│       ├── qdrant_indexer.py # Vector embeddings and indexing
│       ├── retriever.py     # Hybrid RAG retrieval
│       └── graphrag.py      # LangChain GraphRAG
├── data/                    # EMR JSON files directory
├── static/
│   └── graphs/              # Generated HTML visualizations
├── graphrag/                # Python virtual environment
├── requirements.txt         # Python dependencies
└── .env                     # Environment variables (not in repo)
```

Getting Started

Prerequisites

- Python 3.10 or higher
- Neo4j 5.x database instance
- Windows/Linux/macOS

Installation

1. **Clone or download the project**
2. **Create and activate virtual environment**

   ```
   python -m venv graphrag
   .\graphrag\Scripts\Activate.ps1
   ```

3. **Install dependencies**

   ```
   pip install -r requirements.txt
   ```

4. **Configure environment variables**

   Create a `.env` file in the project root:

```
# Neo4j Configuration
NEO4J_URI=bolt://localhost:7687 (or your neo4j URI)
NEO4J_USER=neo4j
NEO4J_PASS=your_password_here


# Qdrant Configuration
QDRANT_URL=http://localhost:6333 (or your qdrant URI)
QDRANT_API_KEY=your_qdrant_api_key
QDRANT_COLLECTION=patient_transcript

# OpenAI Configuration
OPENAI_API_KEY=your_openai_api_key
EMBED_MODEL=text-embedding-3-small
EMBED_DIM=1536

# Privacy/Security
PATIENT_SALT=AIEMR
```

5. **Start Neo4j database**

   Ensure your Neo4j instance is running and accessible.

6. **Run the application**

   ```
   uvicorn app.main:app --reload
   ```

   The API will be available at: `http://127.0.0.1:8000`

Verification

- **Interactive API Docs**: http://127.0.0.1:8000/docs
- **ReDoc Documentation**: http://127.0.0.1:8000/redoc
- **OpenAPI Schema**: http://127.0.0.1:8000/openapi.json

Quick Start: RAG Features

After setup, initialize the RAG system:

1. **Sync EMR Data**:

   ```
   curl -X POST http://127.0.0.1:8000/ingest/sync
   ```

2. **Build Vector Index**:

   ```
   curl -X POST http://127.0.0.1:8000/rag/index/rebuild
   ```

3. **Ask a Question (Hybrid Mode)**:

   ```
   curl -X POST http://127.0.0.1:8000/rag/query \
     -H "Content-Type: application/json" \
     -d '{
       "question": "What medications is patient 00028 taking?",
       "mode": "hybrid",
       "patient_ids": ["00028"]
     }'
   ```

4. **Ask a Question (Graph Mode)**:

   ```
   curl -X POST http://127.0.0.1:8000/rag/query \
     -H "Content-Type: application/json" \
     -d '{
       "question": "Show me the medical history of patient 00042",
       "mode": "graph"
     }'
   ```

API Endpoints

Base URL

```
http://127.0.0.1:8000
```

Endpoints Overview

Method	Endpoint	Description
POST	`/ingest/sync`	Manually trigger EMR data synchronization
GET	`/patients/{patient_id}/graph`	Get patient graph as JSON
GET	`/patients/{patient_id}/graph.html`	Get interactive HTML visualization
POST	`/rag/index/rebuild`	Rebuild complete Qdrant vector index
POST	`/rag/index/upsert`	Upsert specific patients to Qdrant index
POST	`/rag/query`	Query RAG system (JSON, no file upload)
POST	`/rag/query/upload`	Query RAG system with document upload
GET	`/static/graphs/{filename}`	Access generated graph HTML files

---

1. Ingest Endpoints

POST `/ingest/sync`

Manually triggers a synchronization pass over the EMR directory. The system automatically checks for new or modified JSON files and ingests them into Neo4j.

Tags: `ingest`

Request:

- No request body required

Response: `200 OK`

```
{
  "status": "ok"
}
```

Example:

```
curl -X POST http://127.0.0.1:8000/ingest/sync
```

Notes:

- The system runs background synchronization every 60 seconds (configurable via `SYNC_INTERVAL_SEC`)
- Only files with changed content (SHA256 hash) are re-ingested
- Patient ID is extracted from the first record in each JSON file

---

2. Patient Endpoints

GET `/patients/{patient_id}/graph`

Retrieves the complete medical knowledge graph for a specific patient as structured JSON data.

Tags: `patients`

Path Parameters:

- `patient_id` (string, required): The unique patient identifier (e.g., "00042", "00028")

Response: `200 OK`

```
{
  "patient_id": "00042",
  "nodes": [
    {
      "id": "4:3b5d8c9e-1234-5678-90ab-cdef12345678",
      "attrs": {
        "label": "Patient 00042",
        "color": "#1f77b4",
        "shape": "ellipse",
        "size": 35
      }
    },
    {
      "id": "4:3b5d8c9e-1234-5678-90ab-cdef12345679",
      "attrs": {
        "label": "General Information",
        "color": "#2ca02c",
        "shape": "box",
        "size": 26
      }
    }
  ],
  "edges": [
    {
      "source": "4:3b5d8c9e-1234-5678-90ab-cdef12345678",
      "target": "4:3b5d8c9e-1234-5678-90ab-cdef12345679",
      "attrs": {
        "label": "HAS_GENERAL_INFORMATION",
        "color": "#8c564b",
        "arrows": "to"
      }
    }
  ]
}
```


Error Responses:

- `404 Not Found` : Patient not found or no graph data available


```
{
  "detail": "No graph found for 00042"
}
```


Example:

```
curl http://127.0.0.1:8000/patients/00042/graph
```

GET `/patients/{patient_id}/graph.html`

Generates and returns an interactive HTML visualization of the patient's medical knowledge graph using PyVis.

Tags: `patients`

Path Parameters:

- `patient_id` (string, required): The unique patient identifier

Response: `200 OK`

- Content-Type: `text/html`
- Returns an interactive HTML page with graph visualization

Features:

- Interactive node dragging and zooming
- Physics-based layout (Barnes-Hut algorithm)
- Color-coded nodes by type:
  - **Patient**: Blue ellipse
  - **SectionTable**: Green box

    - **Schema**: Purple dot
    - **Value**: Orange diamond
- Labeled edges showing relationships

Error Responses:

- `404 Not Found` : Patient not found or no graph data available

Example:

```
# Open in browser
http://127.0.0.1:8000/patients/00042/graph.html
```

Notes:

- A new HTML file is generated with each request (timestamped filename)
- Files are saved to `static/graphs/` directory
- Uses UTF-8 encoding to handle special medical characters

---

3. RAG (Retrieval-Augmented Generation) Endpoints

The RAG endpoints provide AI-powered question answering capabilities using two modes:

- **Hybrid Mode**: Combines vector search (Qdrant) with Neo4j graph retrieval and OpenAI for contextual answers
- **Graph Mode**: Uses LangChain's GraphCypherQAChain to automatically generate and execute Cypher queries

---

POST `/rag/index/rebuild`

Rebuilds the complete Qdrant vector index from all patient data in Neo4j. This creates embeddings for all patient medical facts and stores them in Qdrant for semantic search.

Tags: `rag`

Request:

- No request body required

Response: `200 OK`

```
{
  "collection": "patient_transcript",
  "upserted": 1247
}
```

Example:

```
curl -X POST http://127.0.0.1:8000/rag/index/rebuild
```

Notes:

- This is a resource-intensive operation
- Run this after initial data ingestion or major data updates
- Creates embeddings using OpenAI's text-embedding-3-small model
- Filters out nodes without UUID assignments

When to Use:

- Initial setup after ingesting EMR data
- After bulk data updates
- If vector index becomes corrupted

POST `/rag/index/upsert`

Upserts (updates or inserts) specific patients' data into the Qdrant vector index. Use this for incremental updates after individual patient records change.

Tags: `rag`

Request Body: Array of patient IDs

```
["00028", "00042"]
```

Response: `200 OK`

```
{
  "collection": "patient_transcript",
  "upserted": 87
}
```

Example:

```
curl -X POST http://127.0.0.1:8000/rag/index/upsert \
  -H "Content-Type: application/json" \
  -d '["00028", "00042"]'
```

Notes:

- More efficient than full rebuild for small updates
- Automatically called by background syncer when files change
- Only processes patients with valid node_id values

POST `/rag/query`

Query the RAG system using JSON format (no file upload). Supports both hybrid and graph-only modes.

Tags: `rag`

Request Body:

```
{
  "question": "What medications is patient 00028 currently taking?",
  "mode": "hybrid",
  "patient_ids": ["00028"]
}
```

Request Schema:

- `question` (string, required): The question to ask
- `mode` (string, default="hybrid"): Query mode - either "hybrid" or "graph"
- `patient_ids` (array[string], optional): Filter results to specific patient IDs

Response: `200 OK`

Hybrid Mode Response:

```
{
  "answer": "Patient 00028 is currently taking Bemfola (generic: follitropin alfa) at 150 IU daily via subcutaneous injection, start
  "context_json": "{\"00028\": {\"PastMedication\": [...], \"MedicalHistory\": [...]}}",
  "value_node_ids": ["uuid-1", "uuid-2", "uuid-3"]
}
```

Graph Mode Response:

```
{
  "answer": "Based on the database, patient 00028 has the following medications...",
  "intermediate_steps": [
    ["Generated Cypher Query", "MATCH (p:Patient {patientID:'00028'})..."],
    ["Query Results", "[{medication: 'Bemfola', dose: '150 IU'}]"]
  ]
}
```

Examples:

```
# Hybrid mode with patient filter
curl -X POST http://127.0.0.1:8000/rag/query \
  -H "Content-Type: application/json" \
  -d '{
    "question": "Is Bemfola helpful for fertility treatment?",
    "mode": "hybrid",
    "patient_ids": ["00028"]
  }'

# Graph mode (auto-generates Cypher)
curl -X POST http://127.0.0.1:8000/rag/query \
  -H "Content-Type: application/json" \
  -d '{
    "question": "What is the medical history of patient 00042?",
    "mode": "graph"
  }'

# Hybrid mode across all patients
curl -X POST http://127.0.0.1:8000/rag/query \
  -H "Content-Type: application/json" \
  -d '{
    "question": "What are common fertility medications?",
    "mode": "hybrid",
    "patient_ids": null
  }'
```

Mode Comparison:

Feature	Hybrid Mode	Graph Mode
Search Method	Vector similarity (embeddings)	LLM-generated Cypher queries
Best For	Semantic questions, concept matching, combine EMR tables and the uploaded document	Specific patient data retrieval, only based on EMR tables
Speed	Fast (vector search)	Slower (LLM query generation)
Accuracy	Good for similar concepts	Excellent for structured queries
Context	Top-K similar facts	Complete graph traversal

Error Responses:

- `422 Unprocessable Entity` : Invalid request format
- `500 Internal Server Error` : OpenAI API error, Qdrant connection error, or Neo4j error

Notes:

- Hybrid mode uses temperature 0.2 (or default if not supported)
- Graph mode uses gpt-4o for Cypher generation, gpt-4o-mini for answer generation
- Patient IDs are hashed for privacy in Qdrant storage
- Responses do not include raw patient identifiers

POST `/rag/query/upload`

Query the RAG system with document upload capability. Upload a text document to include as additional context alongside the retrieved patient data.

Tags: `rag`

Request: `multipart/form-data`

Form Fields:

- `question` (string, required): The question to ask
- `mode` (string, default="hybrid"): Query mode - either "hybrid" or "graph"
- `patient_ids` (string, optional): Comma-separated patient IDs (e.g., "00028,00042")
- `document` (file, required): Text file to include as extra context

Response: `200 OK`

```
{
  "answer": "Based on the consultation note and patient 00028's EMR, the treatment plan includes...",
  "context_json": "{...}",
  "value_node_ids": ["uuid-1", "uuid-2"]
}
```

Example using cURL:

```
curl -X POST http://127.0.0.1:8000/rag/query/upload \
  -F "question=What treatment plan is recommended based on this consultation?" \
  -F "mode=hybrid" \
  -F "patient_ids=00028" \
  -F "document=@consultation_note.txt"
```

Example using Python:

```
import requests

with open('consultation_note.txt', 'rb') as f:
    files = {'document': f}
    data = {
        'question': 'What treatment plan is recommended?',
        'mode': 'hybrid',
        'patient_ids': '00028,00042'
    }
    response = requests.post(
        'http://127.0.0.1:8000/rag/query/upload',
        files=files,
        data=data
    )
    print(response.json())
```

Use Cases:

- Analyze consultation notes with patient history
- Compare lab results with existing patient data
- Interpret medical reports in context of patient EMR
- Clinical decision support with external documents

Error Responses:

- `400 Bad Request` : Invalid mode parameter
- `422 Unprocessable Entity` : Missing required fields or invalid file
- `500 Internal Server Error` : Processing error

Notes:

- Document is decoded as UTF-8 text
- Only text files are supported (.txt, .md, etc.)
- Document content is appended to the RAG context
- Maximum file size depends on server configuration

---

4. Static File Serving

GET `/static/graphs/{filename}`

Serves previously generated graph HTML files.

Path Parameters:

- `filename` (string, required): The HTML filename (e.g., `patient_00042_1760737546.html`)

Response: `200 OK`

- Content-Type: `text/html`

Example:

```
http://127.0.0.1:8000/static/graphs/patient_00042_1760737546.html
```

Data Models

GraphNode

Represents a node in the patient knowledge graph.

```
{
  "id": "string",           # Neo4j element ID
  "attrs": {                # Node attributes
    "label": "string",      # Display label
    "color": "string",      # Hex color code
    "shape": "string",      # Node shape (ellipse, box, dot, diamond)
    "size": "integer"       # Visual size
  }
}
```

GraphEdge

Represents a relationship between two nodes.

```
{
  "source": "string",       # Source node ID
  "target": "string",       # Target node ID
  "attrs": {                # Edge attributes
    "label": "string",      # Relationship type
    "color": "string",      # Hex color code
    "arrows": "string"      # Arrow direction ("to", "from", "both")
  }
}
```

GraphResponse

Complete patient graph response.

```
{
  "patient_id": "string",   # Patient identifier
  "nodes": [GraphNode],     # List of graph nodes
  "edges": [GraphEdge]      # List of graph edges
}
```

Configuration

Environment Variables

Configuration is managed through environment variables, loaded from `.env` file using `pydantic-settings`.

Variable	Type	Default	Description
Neo4j Configuration			
`NEO4J_URI`	string	**Required**	Neo4j connection URI (e.g., `bolt://localhost:7687`)
`NEO4J_USER`	string	**Required**	Neo4j username
`NEO4J_PASS`	string	**Required**	Neo4j password
File Paths			
`EMR_DIR`	Path	`data`	Directory containing EMR JSON files

Variable	Type	Default	Description
`STATIC_DIR`	Path	`static`	Static files directory
`GRAPH_HTML_DIR`	Path	`static/graphs`	Output directory for generated HTML graphs
`SYNC_INTERVAL_SEC`	int	`60`	Background sync interval in seconds
Qdrant Configuration			
`QDRANT_URL`	string	**Required**	Qdrant server URL (e.g., `http://localhost:6333`)
`QDRANT_API_KEY`	string	**Required**	Qdrant API key for authentication
`QDRANT_COLLECTION`	string	`patient_transcript`	Qdrant collection name for patient data
OpenAI Configuration			
`OPENAI_API_KEY`	string	**Required**	OpenAI API key for GPT and embeddings
`EMBED_MODEL`	string	`text-embedding-3-small`	OpenAI embedding model
`EMBED_DIM`	int	`1536`	Embedding dimension (must match model)
`CHAT_MODEL`	string	`gpt-4o-mini`	OpenAI chat model for Q&A
Privacy/Security			
`PATIENT_SALT`	string	`AIEMR`	Salt for patient ID hashing (privacy protection)

Configuration Class

Located in `app/config.py`:

```
class Settings(BaseSettings):
    # Neo4j
    NEO4J_URI: str
    NEO4J_USER: str
    NEO4J_PASS: str
    EMR_DIR: Path = Field(default=Path("data"))
    STATIC_DIR: Path = Field(default=Path("static"))
    GRAPH_HTML_DIR: Path = Field(default_factory=lambda: Path("static/graphs"))
    SYNC_INTERVAL_SEC: int = 60

    # Qdrant / Embeddings / Privacy
    QDRANT_URL: str
    QDRANT_API_KEY: str
    QDRANT_COLLECTION: str = "patient_transcript"
    OPENAI_API_KEY: str
    EMBED_MODEL: str = "text-embedding-3-small"
    EMBED_DIM: int = 1536
    CHAT_MODEL: str = "gpt-5"
    PATIENT_SALT: str = "AIEMR"

    class Config:
        env_file = ".env"
```

Development

Running in Development Mode

```
uvicorn app.main:app --reload --host 0.0.0.0 --port 8000
```

Options:

- `--reload` : Auto-reload on code changes
- `--host` : Bind to specific host (0.0.0.0 for external access)
- `--port` : Specify port number
- `--log-level` : Set logging level (debug, info, warning, error)

Testing API Endpoints

Using cURL

```
# Trigger sync
curl -X POST http://127.0.0.1:8000/ingest/sync

# Get patient graph JSON
curl http://127.0.0.1:8000/patients/00042/graph

# Download HTML visualization
curl http://127.0.0.1:8000/patients/00042/graph.html -o patient_graph.html
```

Using Python

```
import requests

# Trigger sync
response = requests.post("http://127.0.0.1:8000/ingest/sync")
print(response.json())

# Get patient graph
response = requests.get("http://127.0.0.1:8000/patients/00042/graph")
graph_data = response.json()
print(f"Nodes: {len(graph_data['nodes'])}, Edges: {len(graph_data['edges'])}")
```

Using Swagger UI

Navigate to http://127.0.0.1:8000/docs for interactive API testing.

Adding New Endpoints

1. Create router in `app/routers/` directory
2. Define endpoint using FastAPI decorators
3. Register router in `app/main.py` :

```
from app.routers import new_router
app.include_router(new_router.router)
```

Database Schema

The Neo4j graph uses the following node labels:

- **Patient**: Root node for each patient
- **SectionTable**: Medical record sections (e.g., General Information, Medical History)
- **Schema**: Field definitions within sections
- **Value**: Actual data values
- **IngestionMeta**: Metadata for tracking file synchronization

Key relationships:

- `HAS_GENERAL_INFORMATION`
- `HAS_MENSTRUAL_HISTORY`
- `HAS_MEDICAL_HISTORY`
- `HAS_OBSTETRICS_HISTORY`
- `HAS_PAST_MEDICATION`
- `HAS_PAST_TESTING`
- `HAS_SEXUAL_HISTORY`
- `HAS_INFORMATION_OF`
- `HAS_VALUE`

Troubleshooting

Common Issues

1. Module Import Error: "Could not import module 'app.main'"

Problem: File named incorrectly (e.g., `mian.py` instead of `main.py`)

Solution:

```
Rename-Item -Path "app\mian.py" -NewName "main.py"
```

---

2. Pydantic Import Error: "BaseSettings has moved to pydantic-settings"

Problem: Using Pydantic v2 without `pydantic-settings` package

Solution:

```
pip install pydantic-settings
```

Update imports in `app/config.py`:

```
from pydantic import Field
from pydantic_settings import BaseSettings  # Changed from pydantic
```

---

3. UnicodeEncodeError: 'charmap' codec can't encode characters

Problem: Windows default encoding (cp1252) can't handle Unicode characters in medical data

Solution: Ensure UTF-8 encoding when writing HTML files

In `app/services/visualize.py`:

```
# Instead of net.show(out_path)
html_content = net.generate_html()
with open(out_path, "w", encoding="utf-8") as f:
    f.write(html_content)
```

---

4. Neo4j Connection Error

Problem: Cannot connect to Neo4j database

Solution:

- Verify Neo4j is running: Check Neo4j Desktop or service status
- Check connection URI in `.env` file
- Verify credentials (username/password)
- Test connection:

  ```
  from neo4j import GraphDatabase
  driver = GraphDatabase.driver("bolt://localhost:7687", auth=("neo4j", "password"))
  driver.verify_connectivity()
  ```

---

5. 404 Error: Patient Not Found

Problem: Patient data not ingested or patient ID doesn't match

Solution:

- Trigger manual sync: `POST /ingest/sync`
- Check JSON files in `data/` directory
- Verify patient ID matches the `patient_id` field in JSON
- Check Neo4j database directly:

  ```
  MATCH (p:Patient) RETURN p.patientID
  ```

6. Background Sync Not Working

Problem: New files not automatically ingested

Solution:

- Check `SYNC_INTERVAL_SEC` in `.env`
- Verify `EMR_DIR` path is correct
- Check application logs for errors
- Manually trigger sync to test: `POST /ingest/sync`

---

7. Neo4j Cypher Syntax Error - "Expected exactly one statement per query but got: 2"

Problem: UUID backfill query contains multiple statements causing syntax error

Solution: Split statements and execute separately

In `app/graph/ingest.py`:

```
# Run each statement separately
for stmt in [c.strip() for c in BACKFILL_UUIDS.strip().split(";") if c.strip()]:
    try: _run_write(s, stmt)
    except Exception as e: print("uuid backfill note:", e)
```

---

8. TypeError: "can't concat NoneType to bytes" in UUID Conversion

Problem: Node IDs are NULL in database, causing UUID conversion to fail

Solution:

1. Add NULL check in `_as_uuid()` function:

```
def _as_uuid(s: str) -> str:
    if not s:  # Handle None or empty string
        return str(uuid.uuid4())
    try:
        return str(uuid.UUID(s))
    except:
        return str(uuid.uuid5(uuid.NAMESPACE_DNS, str(s)))
```

2. Filter out NULL node_ids in Cypher queries:

```
WHERE v.node_id IS NOT NULL AND s.node_id IS NOT NULL
```

---

9. FastAPI 422 Error: "Input should be a valid dictionary"

Problem: Mixing JSON body parameters with multipart form upload

Solution: Use separate endpoints - `/rag/query` for JSON, `/rag/query/upload` for file uploads

Correct Usage:

```
// For /rag/query (JSON)
{
  "question": "What medications?",
  "mode": "hybrid",
  "patient_ids": ["00028"]  // Not ["string"]!
}
```

---

10. OpenAI Error: "temperature does not support 0.2"

Problem: Some models only support default temperature (1.0)

Solution: Add fallback in retriever:

```
try:
    resp = _openai.chat.completions.create(
        model=settings.CHAT_MODEL,
        messages=messages,
        temperature=0.2
    )
except Exception as e:
    if "temperature" in str(e).lower():
        # Retry without temperature
        resp = _openai.chat.completions.create(
            model=settings.CHAT_MODEL,
            messages=messages
        )
```

---

11. OpenAI Error: "The api_key client option must be set"

Problem: LangChain's ChatOpenAI not receiving API key

Solution: Pass API key explicitly:

```
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(
    model='gpt-4o-mini',
    temperature=0.2,
    api_key=settings.OPENAI_API_KEY  # Must pass explicitly
)
```

---

12. Qdrant Connection Error

Problem: Cannot connect to Qdrant vector store

Solution:

- Verify Qdrant is running (Docker or local service)
- Check `QDRANT_URL` in `.env` (e.g., `http://localhost:6333`)
- Verify API key if using cloud Qdrant
- Test connection:

```
from qdrant_client import QdrantClient
client = QdrantClient(url="http://localhost:6333")
print(client.get_collections())
```

Docker Setup:

```
docker run -p 6333:6333 qdrant/qdrant
```

Logging

Enable debug logging for troubleshooting:

```
uvicorn app.main:app --log-level debug
```

Add custom logging in code:

```
import logging
logger = logging.getLogger(__name__)
logger.info("Processing patient: %s", patient_id)
```

Deployment

Production Deployment

1. **Use Production Server**:

   ```
   gunicorn app.main:app -w 4 -k uvicorn.workers.UvicornWorker
   ```

2. **Environment Configuration**:
   - Use environment-specific `.env` files
   - Set secure Neo4j credentials
   - Configure firewall rules
3. **HTTPS**: Deploy behind reverse proxy (Nginx, Traefik) with SSL
4. **Monitoring**: Implement health checks and monitoring

   ```
   @app.get("/health")
   def health_check():
       return {"status": "healthy"}
   ```

Support and Contact

For issues, questions, or contributions:

- Review this documentation
- Check the interactive API docs at `/docs`
- Examine source code in `app/` directory
- Test endpoints using Swagger UI

RAG System Architecture

Hybrid RAG Mode

The hybrid mode combines vector search with graph retrieval:

1. **Question Embedding**: User question is embedded using OpenAI's text-embedding-3-small
2. **Vector Search**: Qdrant finds top-K semantically similar patient facts
3. **Graph Retrieval**: Retrieved node IDs are used to query Neo4j for full context
4. **Context Assembly**: Results are grouped by patient and section
5. **Answer Generation**: GPT-4o-mini generates answer based on retrieved context

Workflow:

```
Question → Embed → Qdrant ANN Search → Neo4j Graph Query → Assemble Context → GPT Answer
```

Advantages:

- Fast semantic search across all patient data
- Maintains graph relationships and structure
- Handles conceptual questions well
- Privacy: Patient IDs are hashed in vector store

Graph RAG Mode

The graph mode uses LangChain to automatically generate Cypher queries:

1. **Schema Refresh**: Neo4j graph schema is retrieved
2. **Cypher Generation**: GPT-4o generates Cypher query based on question and schema
3. **Query Execution**: Generated Cypher is executed on Neo4j
4. **Answer Synthesis**: GPT-4o-mini formulates natural language answer from results

Workflow:

```
Question → Schema Context → GPT-4o Cypher Gen → Neo4j Execute → GPT Answer
```

Advantages:

- Handles complex graph traversals
- Better for specific patient queries
- No pre-indexing required
- Returns intermediate steps for debugging

Changelog

Version 2.0.0 (Current)

- **NEW**: RAG question answering with hybrid and graph modes
- **NEW**: Qdrant vector store integration for semantic search
- **NEW**: OpenAI embeddings and GPT-4o integration
- **NEW**: LangChain GraphCypherQAChain for automated query generation
- **NEW**: Document upload capability for external context
- **NEW**: Patient ID hashing for privacy protection
- **NEW**: Automatic Qdrant indexing in background syncer
- **CONFIG**: Added OpenAI, Qdrant, and embedding configurations

Version 1.0.0

- Initial release
- Patient graph querying (JSON and HTML)
- Automatic EMR file synchronization
- Interactive graph visualization
- Neo4j integration

Last Updated: October 22, 2025