```python
import json
import sqlite3
from typing import List, Optional
from contextlib import contextmanager
from datetime import datetime
import structlog
from fastapi import FastAPI, HTTPException, Request
from fastapi.middleware.cors import CORSMiddleware
from fastapi.middleware.gzip import GZipMiddleware
from fastapi.middleware.trustedhost import TrustedHostMiddleware
from pydantic import BaseModel
from opentelemetry import trace, metrics
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import (
    OTLPSpanExporter,
)
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor
from opentelemetry.sdk.metrics import MeterProvider
from opentelemetry.sdk.metrics.export import (
    PeriodicExportingMetricReader,
)
from opentelemetry.exporter.otlp.proto.grpc.metric_exporter import (
    OTLPMetricExporter,
)
from opentelemetry.instrumentation.fastapi import FastAPIInstrumentor
```

```
# from opentelemetry.instrumentation.sqlite3 import SQLite3Instrumentor
from prometheus_client import Counter, Histogram
import uvicorn
from mcs.main import MedicalCoderSwarm


# Configure structured logging
logger = structlog.get_logger()


# Configure OpenTelemetry
tracer_provider = TracerProvider()
otlp_span_exporter = OTLPSpanExporter()
span_processor = BatchSpanProcessor(otlp_span_exporter)
tracer_provider.add_span_processor(span_processor)
trace.set_tracer_provider(tracer_provider)


# Configure metrics
metric_reader = PeriodicExportingMetricReader(OTLPMetricExporter())
meter_provider = MeterProvider(metric_readers=[metric_reader])
metrics.set_meter_provider(meter_provider)


meter = metrics.get_meter(__name__)
request_counter = meter.create_counter(
    name="api_requests_total",
    description="Total number of API requests",
    unit="1",
)
```

```
# Initialize Prometheus metrics
REQUEST_TIME = Histogram(
    "request_processing_seconds",
    "Time spent processing request",
    ["endpoint"],
)
ERROR_COUNTER = Counter(
    "api_errors_total",
    "Total number of API errors",
    ["endpoint", "error_type"],
)

# Database configuration
DB_POOL_SIZE = 5
db_path = "medical_coder.db"


class DatabasePool:
    def __init__(self, database_path: str, pool_size: int):
        self.database_path = database_path
        self.pool_size = pool_size
        self.connections = []
        self.initialize_pool()

    def initialize_pool(self):
        for _ in range(self.pool_size):
            conn = sqlite3.connect(self.database_path)
            conn.row_factory = sqlite3.Row
            self.connections.append(conn)

    @contextmanager
    def get_connection(self):
        if not self.connections:
            conn = sqlite3.connect(self.database_path)
            conn.row_factory = sqlite3.Row
        else:
            conn = self.connections.pop()

        try:
            yield conn
        finally:
            self.connections.append(conn)


db_pool = DatabasePool(db_path, DB_POOL_SIZE)

# Initialize FastAPI app with additional configuration
app = FastAPI(
    title="MedicalCoderSwarm API",
    version="1.0.0",
    docs_url="/api/docs",
    redoc_url="/api/redoc",
)

# Add middleware
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
app.add_middleware(GZipMiddleware, minimum_size=1000)
app.add_middleware(
    TrustedHostMiddleware,
    allowed_hosts=[
        "localhost",
        "127.0.0.1",
    ],  # Configure for production
)

# Instrument FastAPI with OpenTelemetry
FastAPIInstrumentor.instrument_app(app)
# SQLite3Instrumentor().instrument()


# Pydantic models
```

```
class PatientCase(BaseModel):
    patient_id: Optional[str] = None
    case_description: Optional[str] = None



class QueryResponse(BaseModel):
    patient_id: Optional[str] = None
    case_data: Optional[str] = None
    timestamp: datetime = datetime.utcnow()



class QueryAllResponse(BaseModel):
    patients: Optional[List[QueryResponse]] = None
    total_count: int
    timestamp: datetime = datetime.utcnow()



class BatchPatientCase(BaseModel):
    cases: Optional[List[PatientCase]] = None



# Middleware for request tracking
@app.middleware("http")
async def add_process_time_header(request: Request, call_next):
    start_time = datetime.utcnow()
    response = await call_next(request)
    process_time = (datetime.utcnow() - start_time).total_seconds()


    REQUEST_TIME.labels(endpoint=request.url.path).observe(
        process_time
    )
    response.headers["X-Process-Time"] = str(process_time)


    return response




# Enhanced database functions
def fetch_patient_data(patient_id: str) -> Optional[dict]:
    with db_pool.get_connection() as conn:
        try:
            cursor = conn.cursor()
            cursor.execute(
                "SELECT patient_data FROM patients WHERE patient_id = ?",
                (patient_id,),
            )
            row = cursor.fetchone()
            return json.loads(row[0]) if row else None
        except sqlite3.Error as e:
            logger.error(
                "database_error", error=str(e), patient_id=patient_id
            )
            raise HTTPException(
            status_code=500, detail=f"Database error: {str(e)}"
        )



def save_patient_data(patient_id: str, patient_data: str):
    with db_pool.get_connection() as conn:
        try:
            cursor = conn.cursor()
            cursor.execute(
                """
                INSERT OR REPLACE INTO patients
                (patient_id, patient_data, created_at, updated_at)
                VALUES (?, ?, datetime('now'), datetime('now'))
                """,
                (patient_id, patient_data),
            )
            conn.commit()
        except sqlite3.Error as e:
            logger.error(
                "database_error", error=str(e), patient_id=patient_id
            )
            raise HTTPException(
                status_code=500, detail=f"Database error: {str(e)}"
            )
```

```
# Enhanced API endpoints
@app.post("/v1/medical-coder/run", response_model=QueryResponse)
async def run_medical_coder(
    patient_case: PatientCase, request: Request
):
    tracer = trace.get_tracer(__name__)
    with tracer.start_as_current_span("run_medical_coder") as span:
        try:
            span.set_attribute("patient_id", patient_case.patient_id)
            logger.info(
                "processing_patient_case",
                patient_id=patient_case.patient_id,
                request_id=request.headers.get("X-Request-ID"),
            )

            swarm = MedicalCoderSwarm(
                patient_id=patient_case.patient_id,
                max_loops=1,
                patient_documentation="",
            )
            swarm.run(task=patient_case.case_description)

            swarm_output = swarm.to_dict()
            save_patient_data(
                patient_case.patient_id, json.dumps(swarm_output)
            )
            request_counter.add(1, {"endpoint": "run_medical_coder"})


            return QueryResponse(
                patient_id=patient_case.patient_id,
                case_data=json.dumps(swarm_output),
                timestamp=datetime.utcnow(),
            )
        except Exception as error:
            ERROR_COUNTER.labels(
                endpoint="run_medical_coder",
                error_type=type(error).__name__,
            ).inc()
            logger.error(
                "medical_coder_error",
                error=str(error),
                patient_id=patient_case.patient_id,
            )
            raise HTTPException(
                status_code=500,
                detail=f"Processing error: {str(error)}",
            )



@app.get(
    "/v1/medical-coder/patient/{patient_id}",
    response_model=QueryResponse,
)
async def get_patient_data(patient_id: str, request: Request):
    tracer = trace.get_tracer(__name__)
    with tracer.start_as_current_span("get_patient_data") as span:
        try:
            span.set_attribute("patient_id", patient_id)
            patient_data = fetch_patient_data(patient_id)

            if not patient_data:
                raise HTTPException(
                    status_code=404, detail="Patient not found"
                )

            request_counter.add(1, {"endpoint": "get_patient_data"})

            return QueryResponse(
                patient_id=patient_id,
                case_data=json.dumps(patient_data),
                timestamp=datetime.utcnow(),
            )
        except Exception as error:
            ERROR_COUNTER.labels(
                endpoint="get_patient_data",
                error_type=type(error).__name__,
            ).inc()
            logger.error(
                "fetch_patient_error",
                error=str(error),
                patient_id=patient_id,
            )
            raise



@app.get("/v1/medical-coder/health")
async def health_check():
    """Health check endpoint for monitoring"""
    try:
        with db_pool.get_connection() as conn:
            cursor = conn.cursor()
            cursor.execute("SELECT 1")
            return {
                "status": "healthy",
                "timestamp": datetime.utcnow(),
            }
    except Exception as e:
        logger.error("health_check_failed", error=str(e))
        raise HTTPException(
            status_code=503, detail="Service Unavailable"
        )
```

```
if __name__ == "__main__":
    try:
        uvicorn.run(
            app,
            host="0.0.0.0",
            port=8000,
            workers=4,
            log_config={
                "version": 1,
                "disable_existing_loggers": False,
                "formatters": {
                    "json": {
                        "()": structlog.stdlib.ProcessorFormatter,
                        "processor": structlog.processors.JSONRenderer(),
                    }
                },
                "handlers": {
                    "default": {
                        "class": "logging.StreamHandler",
                        "formatter": "json",
                    }
                },
                "loggers": {
                    "": {
                        "handlers": ["default"],
                        "level": "INFO",
                    }
                },
            },
        )
    except Exception as e:
        logger.error("startup_error", error=str(e))
        raise
```