# 🚀 Phase 2 Implementation Guide

### Core Improvements - Technical Specifications & Code

## 📋 Phase 2 Overview

**Timeline:** Week 3-6
**Total Effort:** 120 hours
**Expected Impact:** 30% performance improvement, 50% error reduction

### Core Improvements:

1. API Rate Limiting & Pagination
2. Database Optimization (Connection Pooling, Indexes, Query Optimization)
3. N8N Workflow Consolidation
4. Error Handling Enhancement
5. Security Hardening (API Key Rotation, CORS, Input Sanitization, Audit Logging)

## 1️⃣ API Rate Limiting & Pagination

### 🎯 Goal

Implement token bucket rate limiting and cursor-based pagination for API efficiency

### 📝 Rate Limiting Implementation

**File:** `python-services/api/middleware/rate_limiter.py`

```
import time
import redis
from fastapi import Request, HTTPException
from typing import Optional
import logging

logger = logging.getLogger(__name__)

class RateLimitManager:
    def __init__(self, redis_host: str = 'redis', redis_port: int = 6379):
        self.redis_client = redis.Redis(
            host=redis_host,
            port=redis_port,
            decode_responses=True
        )
        self.limits = {
```

```
            'default': {'requests': 100, 'window': 60},  # 100/min
            'upload': {'requests': 50, 'window': 60},    # 50/min for uploads
            'login': {'requests': 5, 'window': 60},      # 5/min for login
            'api_key': {'requests': 1000, 'window': 3600}  # 1000/hour per key
        }

    def get_identifier(self, request: Request, api_key: Optional[str] = None) -&gt; str:
        """Get unique identifier for rate limiting"""
        if api_key:
            return f"rate_limit:api_key:{api_key}"

        client_ip = request.client.host
        user_id = request.scope.get('user_id', 'anonymous')
        return f"rate_limit:user:{user_id}:ip:{client_ip}"

    def is_allowed(self, identifier: str, limit_type: str = 'default') -&gt; tuple[bool,
        """Check if request is allowed using token bucket algorithm"""
        limit_config = self.limits.get(limit_type, self.limits['default'])
        requests_limit = limit_config['requests']
        window = limit_config['window']

        key = f"{identifier}:{limit_type}"
        current_time = time.time()
        window_start = current_time - window

        try:
            # Get current bucket info
            bucket_data = self.redis_client.hgetall(key)

            if not bucket_data:
                # First request in window
                self.redis_client.hset(key, mapping={
                    'count': 1,
                    'reset_time': current_time + window
                })
                self.redis_client.expire(key, window + 1)

                return True, {
                    'limit': requests_limit,
                    'remaining': requests_limit - 1,
                    'reset': int(current_time + window)
                }

            reset_time = float(bucket_data.get('reset_time', current_time))

            # Window expired, reset bucket
            if reset_time &lt; current_time:
                self.redis_client.hset(key, mapping={
                    'count': 1,
                    'reset_time': current_time + window
                })
                self.redis_client.expire(key, window + 1)

                return True, {
                    'limit': requests_limit,
                    'remaining': requests_limit - 1,
```

```
                    'reset': int(current_time + window)
                }

            # Check if limit exceeded
            count = int(bucket_data.get('count', 0))

            if count &gt;= requests_limit:
                return False, {
                    'limit': requests_limit,
                    'remaining': 0,
                    'reset': int(reset_time),
                    'retry_after': int(reset_time - current_time)
                }

            # Increment counter
            self.redis_client.hincrby(key, 'count', 1)

            return True, {
                'limit': requests_limit,
                'remaining': requests_limit - count - 1,
                'reset': int(reset_time)
            }

        except Exception as e:
            logger.error(f"Rate limit check failed: {str(e)}")
            # On error, allow request but log
            return True, {}

# Global rate limiter instance
rate_limiter = RateLimitManager()

async def rate_limit_middleware(request: Request, call_next):
    """Middleware to apply rate limiting to all requests"""

    # Skip rate limiting for health checks
    if request.url.path in ['/health', '/health/ready', '/health/live']:
        return await call_next(request)

    # Determine limit type based on endpoint
    if '/files/upload' in request.url.path:
        limit_type = 'upload'
    elif '/auth/login' in request.url.path:
        limit_type = 'login'
    else:
        limit_type = 'default'

    # Get API key if available
    api_key = request.headers.get('Authorization', '').replace('Bearer ', '')

    # Get identifier
    identifier = rate_limiter.get_identifier(request, api_key if api_key else None)

    # Check rate limit
    allowed, limit_info = rate_limiter.is_allowed(identifier, limit_type)

    if not allowed:
```

```
        retry_after = limit_info.get('retry_after', 60)
        raise HTTPException(
            status_code=429,
            detail={
                'error': 'Rate limit exceeded',
                'retry_after': retry_after,
                'limit': limit_info.get('limit'),
                'remaining': limit_info.get('remaining'),
                'reset': limit_info.get('reset')
            }
        )

    # Add rate limit headers to response
    response = await call_next(request)
    response.headers['X-RateLimit-Limit'] = str(limit_info.get('limit', ''))
    response.headers['X-RateLimit-Remaining'] = str(limit_info.get('remaining', ''))
    response.headers['X-RateLimit-Reset'] = str(limit_info.get('reset', ''))

    return response
```

### Pagination Implementation

**File:** `python-services/api/utils/pagination.py`

```
import base64
import json
from typing import TypeVar, Generic, List, Optional
from pydantic import BaseModel
from sqlalchemy.orm import Query
from sqlalchemy import desc

T = TypeVar('T')

class PaginationParams(BaseModel):
    cursor: Optional[str] = None
    limit: int = 50

    def validate(self):
        if self.limit &lt; 1:
            self.limit = 1
        elif self.limit &gt; 1000:
            self.limit = 1000
        return self

class CursorPaginationResult(BaseModel, Generic[T]):
    data: List[T]
    next_cursor: Optional[str] = None
    has_more: bool = False
    total_in_page: int = 0

class CursorPaginator:
    @staticmethod
    def encode_cursor(id_value: str) -&gt; str:
        """Encode cursor from ID"""
        return base64.urlsafe_b64encode(
```

```
            json.dumps({'id': str(id_value)}).encode()
        ).decode()

    @staticmethod
    def decode_cursor(cursor: str) -&gt; dict:
        """Decode cursor to get ID"""
        try:
            decoded = base64.urlsafe_b64decode(cursor.encode())
            return json.loads(decoded)
        except Exception:
            return {}

    @staticmethod
    def paginate(
        query: Query,
        params: PaginationParams,
        id_field,
        order_by_field,
        descending: bool = True
    ) -&gt; CursorPaginationResult:
        """Apply cursor-based pagination to query"""

        params.validate()

        # Apply ordering
        order = desc(order_by_field) if descending else order_by_field
        query = query.order_by(order)

        # Apply cursor filter if provided
        if params.cursor:
            cursor_data = CursorPaginator.decode_cursor(params.cursor)
            cursor_id = cursor_data.get('id')

            if cursor_id:
                if descending:
                    query = query.filter(id_field &lt; int(cursor_id))
                else:
                    query = query.filter(id_field &gt; int(cursor_id))

        # Fetch one extra to determine if there are more results
        items = query.limit(params.limit + 1).all()

        has_more = len(items) &gt; params.limit
        if has_more:
            items = items[:params.limit]

        # Generate next cursor
        next_cursor = None
        if has_more and len(items) &gt; 0:
            last_item = items[-1]
            next_cursor = CursorPaginator.encode_cursor(last_item.id)

        return CursorPaginationResult(
            data=items,
            next_cursor=next_cursor,
            has_more=has_more,
```

```
            total_in_page=len(items)
        )

# Usage in route
from fastapi import APIRouter, Depends

router = APIRouter()

@router.get("/files")
async def list_files(
    params: PaginationParams = Depends(),
    db: Session = Depends(get_db),
    user_id: str = Depends(get_current_user)
):
    """List files with cursor-based pagination"""

    query = db.query(File).filter(File.user_id == user_id)

    result = CursorPaginator.paginate(
        query=query,
        params=params,
        id_field=File.id,
        order_by_field=File.created_at,
        descending=True
    )

    return result
```

### ⚛ Frontend Pagination

**File:** `web-react/src/hooks/useInfinitePagination.js`

```
import { useState, useCallback } from 'react';
import axios from 'axios';

export const useInfinitePagination = (url, pageSize = 50) =&gt; {
  const [items, setItems] = useState([]);
  const [cursor, setCursor] = useState(null);
  const [hasMore, setHasMore] = useState(true);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);

  const loadMore = useCallback(async () =&gt; {
    if (loading || !hasMore) return;

    setLoading(true);
    setError(null);

    try {
      const response = await axios.get(url, {
        params: {
          cursor: cursor,
          limit: pageSize
        }
      });
```

```
      const { data, next_cursor, has_more } = response.data;

      setItems(prev =&gt; [...prev, ...data]);
      setCursor(next_cursor);
      setHasMore(has_more);
    } catch (err) {
      setError(err.message);
      console.error('Pagination error:', err);
    } finally {
      setLoading(false);
    }
  }, [cursor, hasMore, loading, url, pageSize]);

  const reset = useCallback(() =&gt; {
    setItems([]);
    setCursor(null);
    setHasMore(true);
    setLoading(false);
    setError(null);
  }, []);

  return {
    items,
    loadMore,
    hasMore,
    loading,
    error,
    reset,
    cursor
  };
};

// Usage in component
const FileList = () =&gt; {
  const { items, loadMore, hasMore, loading } = useInfinitePagination('/api/files');

  return (
    <div>
      {items.map(file =&gt; (
        <div>{file.name}</div>
      ))}
      {hasMore &amp;&amp; (
        &lt;button onClick={loadMore} disabled={loading}&gt;
          {loading ? 'Loading...' : 'Load More'}
        &lt;/button&gt;
      )}
    </div>
  );
};
```

## 2 Database Optimization

### Goal

Implement connection pooling, strategic indexes, and query optimization

### Connection Pooling

**File:** `python-services/api/database.py`

```
from sqlalchemy import create_engine, event, text
from sqlalchemy.orm import sessionmaker, Session
from sqlalchemy.pool import QueuePool
import logging

logger = logging.getLogger(__name__)

# Database configuration
DATABASE_URL = "postgresql://user:password@postgres:5432/construction_ai"

# Create engine with optimized pooling
engine = create_engine(
    DATABASE_URL,
    poolclass=QueuePool,
    pool_size=20,              # Connections to keep in pool
    max_overflow=40,           # Additional connections beyond pool_size
    pool_recycle=3600,         # Recycle connections every hour (prevents timeout)
    pool_pre_ping=True,        # Test connection before using
    echo=False,                # Set True for SQL debugging
    connect_args={
        'connect_timeout': 10,
        'application_name': 'construction_ai'
    }
)

# Connection pool event listeners for monitoring
@event.listens_for(engine, "connect")
def receive_connect(dbapi_conn, connection_record):
    """Configure connection on creation"""
    cursor = dbapi_conn.cursor()
    cursor.execute("SET statement_timeout = 300000")  # 5 min timeout
    cursor.close()

@event.listens_for(engine, "pool_connect")
def receive_pool_connect(dbapi_conn, connection_record):
    """Log pool connections"""
    logger.debug(f"New pool connection: {connection_record.connection_id}")

@event.listens_for(engine, "pool_checkout")
def receive_pool_checkout(dbapi_conn, connection_record, connection_proxy):
    """Log checkout events"""
    logger.debug(f"Checked out connection from pool")

@event.listens_for(engine, "pool_checkin")
```

```
def receive_pool_checkin(dbapi_conn, connection_record):
    """Log checkin events"""
    logger.debug(f"Checked in connection to pool")

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

def get_db() -&gt; Session:
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

# Pool monitoring endpoint
from fastapi import APIRouter

router = APIRouter()

@router.get("/admin/db-pool-status")
async def get_pool_status():
    """Get database connection pool status"""
    pool = engine.pool
    return {
        "pool_size": pool.size(),
        "checked_in": pool.checkedin(),
        "checked_out": pool.checkedout(),
        "overflow": pool.overflow(),
        "total_connections": pool.size() + pool.overflow(),
        "available": pool.checkedin(),
        "status": "healthy" if pool.checkedout() &lt; pool.size() else "stressed"
    }
```

### Strategic Indexes

**File:** `migrations/add_strategic_indexes.sql`

```
-- Primary indexes for common queries
CREATE INDEX CONCURRENTLY idx_files_user_id ON files(user_id);
CREATE INDEX CONCURRENTLY idx_files_project_id ON files(project_id);
CREATE INDEX CONCURRENTLY idx_files_status ON files(status);
CREATE INDEX CONCURRENTLY idx_files_created_at ON files(created_at DESC);
CREATE INDEX CONCURRENTLY idx_files_file_type ON files(file_type);

-- Composite indexes for common filtering patterns
CREATE INDEX CONCURRENTLY idx_files_user_project_status
  ON files(user_id, project_id, status);

CREATE INDEX CONCURRENTLY idx_files_created_user_status
  ON files(created_at DESC, user_id, status);

-- Partial indexes for filtered queries (faster, smaller)
CREATE INDEX CONCURRENTLY idx_active_files
  ON files(user_id, created_at DESC)
  WHERE status != 'archived';
```

```
CREATE INDEX CONCURRENTLY idx_processing_files
  ON files(created_at DESC)
  WHERE status IN ('uploading', 'processing');

-- JSON field indexes
CREATE INDEX CONCURRENTLY idx_analysis_cost
  ON file_analysis USING GIN(analysis-&gt;'cost');

CREATE INDEX CONCURRENTLY idx_materials_material_type
  ON material_analysis USING GIN(materials);

-- User and project queries
CREATE INDEX CONCURRENTLY idx_users_email ON users(email);
CREATE INDEX CONCURRENTLY idx_projects_user_id ON projects(user_id);
CREATE INDEX CONCURRENTLY idx_projects_created_at ON projects(created_at DESC);

-- Analysis and processing
CREATE INDEX CONCURRENTLY idx_analysis_file_id ON file_analysis(file_id);
CREATE INDEX CONCURRENTLY idx_analysis_created_at ON file_analysis(created_at DESC);

-- Cost estimation
CREATE INDEX CONCURRENTLY idx_cost_estimates_file_id ON cost_estimates(file_id);
CREATE INDEX CONCURRENTLY idx_cost_estimates_location ON cost_estimates(location);

-- Audit logging
CREATE INDEX CONCURRENTLY idx_audit_logs_user_id ON audit_logs(user_id);
CREATE INDEX CONCURRENTLY idx_audit_logs_timestamp ON audit_logs(timestamp DESC);
CREATE INDEX CONCURRENTLY idx_audit_logs_event_type ON audit_logs(event_type);

-- Analyze tables
ANALYZE files;
ANALYZE users;
ANALYZE projects;
ANALYZE file_analysis;
ANALYZE cost_estimates;
ANALYZE audit_logs;

-- Monitor index usage
SELECT
  schemaname,
  tablename,
  indexname,
  idx_scan as scans,
  idx_tup_read as tuples_read,
  idx_tup_fetch as tuples_fetched
FROM pg_stat_user_indexes
ORDER BY idx_scan DESC;

-- Check for unused indexes
SELECT
  schemaname,
  tablename,
  indexname
FROM pg_indexes
LEFT JOIN pg_stat_user_indexes ON indexname = relname
WHERE idx_scan = 0
```

```
  AND schemaname NOT IN ('pg_catalog', 'information_schema')
ORDER BY pg_relation_size(indexrelname) DESC;
```

### ⚡ Query Optimization

**File:** `python-services/api/utils/query_optimizer.py`

```
from sqlalchemy.orm import Session, joinedload
from sqlalchemy import text
import time
import logging

logger = logging.getLogger(__name__)

class QueryOptimizer:
    """Helper class for optimizing SQLAlchemy queries"""

    @staticmethod
    def optimize_file_query(db: Session, user_id: str, include_relations: bool = True):
        """Optimized query for fetching files"""
        query = db.query(File).filter(File.user_id == user_id)

        if include_relations:
            # Use joinedload to prevent N+1 queries
            query = query.options(
                joinedload(File.project),
                joinedload(File.analysis),
                joinedload(File.creator)
            )

        return query.order_by(File.created_at.desc())

    @staticmethod
    def get_specific_columns(db: Session, columns: list):
        """Load only specific columns instead of full objects"""
        return db.query(*columns)

    @staticmethod
    def log_slow_queries(threshold_ms: float = 100):
        """Decorator to log slow queries"""
        def decorator(func):
            def wrapper(*args, **kwargs):
                start = time.time()
                result = func(*args, **kwargs)
                elapsed = (time.time() - start) * 1000

                if elapsed &gt; threshold_ms:
                    logger.warning(f"Slow query in {func.__name__}: {elapsed:.2f}ms")

                return result
            return wrapper
        return decorator

# Usage examples
@router.get("/api/files/{user_id}")
```

```
async def get_files(user_id: str, db: Session = Depends(get_db)):
    """Get files with optimized query"""
    # BAD - N+1 query problem
    # for file in db.query(File).all():
    #     print(file.project.name)  # Additional query per file!

    # GOOD - Eager loading
    files = QueryOptimizer.optimize_file_query(db, user_id, include_relations=True)
    return files.limit(100).all()

@router.get("/api/files-summary")
async def get_files_summary(db: Session = Depends(get_db)):
    """Get only needed columns"""
    # BAD - Loads entire objects
    # files = db.query(File).all()

    # GOOD - Load only needed columns
    files = QueryOptimizer.get_specific_columns(
        db,
        [File.id, File.filename, File.created_at, File.status]
    ).limit(100).all()

    return files
```

### Query Performance Monitoring

**File:** `python-services/api/middleware/query_logger.py`

```
from sqlalchemy import event
from sqlalchemy.engine import Engine
import logging
import time

logger = logging.getLogger(__name__)

@event.listens_for(Engine, "before_cursor_execute")
def before_cursor_execute(conn, cursor, statement, parameters, context, executemany):
    conn.info.setdefault('query_start_time', []).append(time.time())

@event.listens_for(Engine, "after_cursor_execute")
def after_cursor_execute(conn, cursor, statement, parameters, context, executemany):
    total = time.time() - conn.info['query_start_time'].pop(-1)

    if total &gt; 0.1:  # Log queries taking &gt; 100ms
        logger.warning(f"Slow Query ({total:.3f}s): {statement[:200]}")

# Add this to your app startup
@app.on_event("startup")
async def startup():
    # Initialize query logging
    pass

# Create endpoint to check slow queries
@router.get("/admin/slow-queries")
async def get_slow_queries(db: Session = Depends(get_db)):
```

```
    """Get statistics on slow queries"""
    result = db.execute(text("""
        SELECT
          mean_exec_time,
          max_exec_time,
          calls,
          query
        FROM pg_stat_statements
        WHERE mean_exec_time &gt; 100
        ORDER BY mean_exec_time DESC
        LIMIT 20
    """)).fetchall()

    return {
        "slow_queries": [
            {
                "avg_time_ms": row[0],
                "max_time_ms": row[1],
                "call_count": row[2],
                "query": row[3][:200]
            }
            for row in result
        ]
    }
```

### 3 N8N Workflow Consolidation

### Goal

Reduce 50+ workflows to 12-15 core workflows with dynamic routing

### Master Workflow Architecture

#### Create new N8N Workflow: "Master_File_Processing"

#### Node 1: Webhook Trigger

```
{
  "type": "webhook",
  "name": "File Upload Trigger",
  "parameters": {
    "method": "POST",
    "path": "file-upload"
  }
}
```

#### Node 2: Route by File Type

```
{
  "type": "switch",
  "name": "Route by File Type",
```

```
  "parameters": {
    "cases": [
      {
        "condition": "file_type == 'dwg' OR file_type == 'dxf' OR file_type == 'dgn'",
        "output": 1
      },
      {
        "condition": "file_type == 'rvt' OR file_type == 'ifc'",
        "output": 2
      },
      {
        "condition": "file_type == 'pdf'",
        "output": 3
      },
      {
        "condition": "file_type == 'xlsx' OR file_type == 'csv'",
        "output": 4
      }
    ]
  }
}
```

**Node 3: CAD File Processor (Output 1)**

```
{
  "type": "http",
  "name": "CAD Extractor",
  "parameters": {
    "url": "http://api:8000/process/cad-extract",
    "method": "POST",
    "headers": {
      "Authorization": "Bearer {{ $env.N8N_API_KEY }}"
    }
  }
}
```

**Node 4: BIM File Processor (Output 2)**

```
{
  "type": "http",
  "name": "BIM Extractor",
  "parameters": {
    "url": "http://api:8000/process/bim-extract",
    "method": "POST"
  }
}
```

**Node 5: PDF Processor (Output 3)**

```
{
  "type": "http",
  "name": "PDF Extractor",
```

```
  "parameters": {
    "url": "http://api:8000/process/pdf-extract",
    "method": "POST"
  }
}
```

**Node 6: Data Processor (Output 4)**

```
{
  "type": "http",
  "name": "Data Parser",
  "parameters": {
    "url": "http://api:8000/process/data-parse",
    "method": "POST"
  }
}
```

**Node 7: Merge Results**

```
{
  "type": "merge",
  "name": "Merge Outputs",
  "parameters": {
    "mode": "concatenate"
  }
}
```

**Node 8: Dynamic Agent Selection**

```
{
  "type": "code",
  "name": "Select Agents",
  "script": "
    const fileType = $input.first().json.file_type;
    const fileSize = $input.first().json.file_size;

    // Select agents based on file characteristics
    const agents = [];

    if (fileSize &gt; 10000000) {  // &gt; 10MB
      agents.push('geometry_analyzer');
    }

    if (['dwg', 'rvt', 'ifc'].includes(fileType)) {
      agents.push('material_classifier');
      agents.push('cost_estimator');
      agents.push('carbon_calculator');
    }

    if (['pdf', 'xlsx', 'csv'].includes(fileType)) {
      agents.push('document_analyzer');
    }
```

```
    return { agents: agents };
  "
}
```

**Node 9: Parallel Processing**

```
{
  "type": "parallelize",
  "name": "Run Agents in Parallel",
  "parameters": {
    "iterations": "{{ $input.first().json.agents.length }}"
  }
}
```

**Node 10-14: Agent Nodes (Parallel)**

For each agent (geometry, materials, costs, carbon, compliance):

```
{
  "type": "http",
  "name": "Agent: {{ $input.first().json.agent_name }}",
  "parameters": {
    "url": "http://ai-agents:5000/agents/{{ $input.first().json.agent_name }}/execute",
    "method": "POST",
    "retryConfig": {
      "max_retries": 3,
      "backoff_multiplier": 2,
      "initial_delay": 1000,
      "max_delay": 30000
    }
  }
}
```

**Node 15: Aggregate Results**

```
{
  "type": "merge",
  "name": "Aggregate All Results",
  "parameters": {
    "mode": "array"
  }
}
```

**Node 16: Update Database**

```
{
  "type": "http",
  "name": "Update File Analysis",
  "parameters": {
    "url": "http://api:8000/files/{{ $input.first().json.file_id }}/analysis",
```

```
    "method": "PUT"
  }
}
```

**Node 17: Send Webhook Status**

```
{
  "type": "http",
  "name": "Broadcast Status",
  "parameters": {
    "url": "http://api:8000/webhook/file-status",
    "method": "POST",
    "body": {
      "file_id": "{{ $input.first().json.file_id }}",
      "status": "completed",
      "progress": 100
    }
  }
}
```

## Workflow Consolidation Strategy

**Before (50+ workflows):**

- separate_dwg_extraction.json
- separate_ifc_extraction.json
- separate_pdf_extraction.json
- separate_material_classification.json
- separate_cost_estimation.json
- (45 more...)

**After (5 core workflows):**

1. Master_File_Processing (main router)
2. CAD_Processing (optimized for DWG/DXF/DGN)
3. BIM_Processing (optimized for RVT/IFC)
4. Document_Processing (PDF, Excel, CSV)
5. Error_Recovery (retry logic for failed files)

**Migration Path:**

1. Create Master workflow
2. Test with sample files
3. Redirect webhooks to Master workflow
4. Monitor consolidation
5. Deprecate old workflows

6. Archive old workflow files

### 4 Error Handling Enhancement

#### Goal

Implement comprehensive error handling, logging, and recovery

#### Custom Exceptions

**File:** `python-services/api/exceptions.py`

```
from fastapi import HTTPException
from typing import Optional, Dict

class ConstructionAIException(HTTPException):
    """Base exception for Construction AI"""

    def __init__(
        self,
        error_code: str,
        message: str,
        status_code: int = 400,
        details: Optional[Dict] = None,
        suggestion: Optional[str] = None
    ):
        self.error_code = error_code
        self.message = message
        self.status_code = status_code
        self.details = details or {}
        self.suggestion = suggestion

        super().__init__(
            status_code=status_code,
            detail={
                "error_code": error_code,
                "message": message,
                "suggestion": suggestion,
                "details": self.details
            }
        )

class FileProcessingException(ConstructionAIException):
    """File processing failed"""
    pass

class GeometryExtractionException(ConstructionAIException):
    """Geometry extraction failed"""
    pass

class MaterialClassificationException(ConstructionAIException):
    """Material classification failed"""
    pass
```

```
class CostEstimationException(ConstructionAIException):
    """Cost estimation failed"""
    pass

class DatabaseException(ConstructionAIException):
    """Database operation failed"""
    pass

class ExternalAPIException(ConstructionAIException):
    """External API call failed"""
    pass

# Usage
raise FileProcessingException(
    error_code="FILE_UPLOAD_FAILED",
    message="Failed to upload file",
    status_code=400,
    details={"filename": "test.dwg", "size": 1024},
    suggestion="Check file format and size limits"
)
```

### Circuit Breaker for External APIs

**File:** `python-services/api/utils/circuit_breaker.py`

```
from enum import Enum
from datetime import datetime, timedelta
import logging
import asyncio

logger = logging.getLogger(__name__)

class CircuitState(Enum):
    CLOSED = "closed"           # Normal operation
    OPEN = "open"               # Failures exceeded, rejecting calls
    HALF_OPEN = "half_open"     # Testing if service recovered

class CircuitBreaker:
    def __init__(
        self,
        failure_threshold: int = 5,
        recovery_timeout: int = 60,
        success_threshold: int = 2
    ):
        self.failure_threshold = failure_threshold
        self.recovery_timeout = recovery_timeout
        self.success_threshold = success_threshold

        self.state = CircuitState.CLOSED
        self.failure_count = 0
        self.success_count = 0
        self.last_failure_time = None
        self.opened_at = None
```

```
    def record_success(self):
        """Record successful call"""
        self.failure_count = 0

        if self.state == CircuitState.HALF_OPEN:
            self.success_count += 1
            if self.success_count &gt;= self.success_threshold:
                self.close()
        elif self.state == CircuitState.CLOSED:
            self.success_count = 0

    def record_failure(self):
        """Record failed call"""
        self.failure_count += 1
        self.last_failure_time = datetime.now()

        if self.state == CircuitState.HALF_OPEN:
            self.open()
        elif self.failure_count &gt;= self.failure_threshold:
            self.open()

    def open(self):
        """Open circuit"""
        self.state = CircuitState.OPEN
        self.opened_at = datetime.now()
        logger.error(f"Circuit breaker opened at {self.opened_at}")

    def close(self):
        """Close circuit"""
        self.state = CircuitState.CLOSED
        self.failure_count = 0
        self.success_count = 0
        logger.info("Circuit breaker closed")

    def half_open(self):
        """Transition to half-open"""
        self.state = CircuitState.HALF_OPEN
        self.success_count = 0
        logger.warning("Circuit breaker half-open, testing recovery")

    async def call(self, func, *args, **kwargs):
        """Execute function with circuit breaker protection"""

        # Check if should transition from OPEN to HALF_OPEN
        if self.state == CircuitState.OPEN:
            if (datetime.now() - self.opened_at).total_seconds() &gt; self.recovery_timeo
                self.half_open()
            else:
                raise Exception(
                    f"Circuit breaker is OPEN. Retry after "
                    f"{self.recovery_timeout - (datetime.now() - self.opened_at).total_se
                )

        try:
            result = await func(*args, **kwargs) if asyncio.iscoroutinefunction(func) els
            self.record_success()
```

```
            return result
        except Exception as e:
            self.record_failure()
            raise

# Usage
openai_breaker = CircuitBreaker(failure_threshold=5, recovery_timeout=60)

async def call_openai_api(payload: dict):
    """Call OpenAI with circuit breaker"""
    try:
        response = await openai_breaker.call(
            call_openai,
            payload
        )
        return response
    except Exception as e:
        logger.error(f"API call failed: {str(e)}")
        # Return degraded response
        return {"status": "degraded", "error": str(e)}
```

### Comprehensive Error Logging

**File:** `python-services/api/middleware/error_logging.py`

```
from fastapi import Request, status
from fastapi.responses import JSONResponse
from fastapi.exceptions import RequestValidationError
import logging
import traceback
from datetime import datetime
import json

logger = logging.getLogger(__name__)

class ErrorLogger:
    @staticmethod
    def log_error(
        error: Exception,
        request: Request = None,
        context: dict = None,
        level: str = "ERROR"
    ):
        """Comprehensive error logging"""

        error_data = {
            "timestamp": datetime.now().isoformat(),
            "error_type": type(error).__name__,
            "error_message": str(error),
            "traceback": traceback.format_exc(),
            "context": context or {}
        }

        if request:
            error_data.update({
```

```
                "method": request.method,
                "path": request.url.path,
                "client": request.client.host if request.client else None
            })

        log_func = getattr(logger, level.lower(), logger.error)
        log_func(json.dumps(error_data, indent=2))

        return error_data

@app.exception_handler(Exception)
async def global_exception_handler(request: Request, exc: Exception):
    """Global exception handler"""

    error_data = ErrorLogger.log_error(exc, request)

    return JSONResponse(
        status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
        content={
            "error": {
                "code": "INTERNAL_ERROR",
                "message": "An unexpected error occurred",
                "request_id": request.headers.get("X-Request-ID", "unknown")
            }
        }
    )

@app.exception_handler(RequestValidationError)
async def validation_exception_handler(request: Request, exc: RequestValidationError):
    """Handle validation errors"""

    return JSONResponse(
        status_code=status.HTTP_422_UNPROCESSABLE_ENTITY,
        content={
            "error": {
                "code": "VALIDATION_ERROR",
                "message": "Request validation failed",
                "details": exc.errors()
            }
        }
    )
```

### 5 Security Hardening

#### Goal

Implement API key rotation, CORS, input sanitization, and audit logging

### 🔑 API Key Rotation

**File:** `python-services/api/security/key_management.py`

```
from datetime import datetime, timedelta
import secrets
import hashlib
from sqlalchemy import Column, String, DateTime, Boolean, JSON
from sqlalchemy.ext.declarative import declarative_base
import logging

logger = logging.getLogger(__name__)
Base = declarative_base()

class APIKey(Base):
    __tablename__ = "api_keys"

    id = Column(String, primary_key=True)
    user_id = Column(String, index=True)
    key_hash = Column(String, unique=True, index=True)
    name = Column(String)
    is_active = Column(Boolean, default=True, index=True)
    created_at = Column(DateTime, default=datetime.now)
    last_rotated = Column(DateTime, default=datetime.now)
    expires_at = Column(DateTime)
    last_used_at = Column(DateTime)
    usage_count = Column(Integer, default=0)
    inactive_keys = Column(JSON, default=[])  # Grace period keys

class APIKeyManager:
    ROTATION_INTERVAL = timedelta(days=30)
    GRACE_PERIOD = timedelta(days=7)
    KEY_PREFIX = "sk_live_"

    @staticmethod
    def hash_key(key: str) -&gt; str:
        """Hash API key for storage"""
        return hashlib.sha256(key.encode()).hexdigest()

    @staticmethod
    def generate_key() -&gt; str:
        """Generate new API key"""
        random_part = secrets.token_urlsafe(32)
        return f"{APIKeyManager.KEY_PREFIX}{random_part}"

    @staticmethod
    async def create_key(db: Session, user_id: str, name: str) -&gt; tuple[str, str]:
        """Create new API key"""
        key = APIKeyManager.generate_key()
        key_hash = APIKeyManager.hash_key(key)

        api_key = APIKey(
            id=secrets.token_hex(8),
            user_id=user_id,
            key_hash=key_hash,
            name=name,
```

```
            created_at=datetime.now(),
            expires_at=datetime.now() + timedelta(days=365)
        )

        db.add(api_key)
        db.commit()

        logger.info(f"Created API key: {api_key.id} for user {user_id}")

        return api_key.id, key  # Return key only once

    @staticmethod
    async def rotate_keys(db: Session):
        """Scheduled task to rotate keys"""
        cutoff_date = datetime.now() - APIKeyManager.ROTATION_INTERVAL

        keys_to_rotate = db.query(APIKey).filter(
            APIKey.last_rotated &lt; cutoff_date,
            APIKey.is_active == True
        ).all()

        for key_record in keys_to_rotate:
            # Generate new key
            new_key = APIKeyManager.generate_key()
            new_hash = APIKeyManager.hash_key(new_key)

            # Store old key in grace period
            old_entry = {
                "key_hash": key_record.key_hash,
                "deactivated_at": datetime.now().isoformat(),
                "grace_until": (datetime.now() + APIKeyManager.GRACE_PERIOD).isoformat()
            }

            if not key_record.inactive_keys:
                key_record.inactive_keys = []
            key_record.inactive_keys.append(old_entry)

            # Update to new key
            key_record.key_hash = new_hash
            key_record.last_rotated = datetime.now()

            db.commit()

            logger.info(f"Rotated API key: {key_record.id}")

    @staticmethod
    async def validate_key(db: Session, key: str) -&gt; tuple[bool, APIKey]:
        """Validate API key"""
        key_hash = APIKeyManager.hash_key(key)

        # Check active keys
        api_key = db.query(APIKey).filter(
            APIKey.key_hash == key_hash,
            APIKey.is_active == True,
            APIKey.expires_at &gt; datetime.now()
        ).first()
```

```
        if api_key:
            api_key.last_used_at = datetime.now()
            api_key.usage_count += 1
            db.commit()
            return True, api_key

        # Check grace period keys
        if api_key:
            for inactive_key in api_key.inactive_keys:
                if inactive_key['key_hash'] == key_hash:
                    grace_until = datetime.fromisoformat(inactive_key['grace_until'])
                    if grace_until &gt; datetime.now():
                        logger.warning(f"Using rotated key in grace period")
                        return True, api_key

        return False, None

# Middleware to validate API keys
@app.middleware("http")
async def validate_api_key(request: Request, call_next):
    """Validate API key in Authorization header"""

    if request.url.path.startswith("/api/"):
        auth_header = request.headers.get("Authorization", "")

        if not auth_header.startswith("Bearer "):
            return JSONResponse(
                status_code=401,
                content={"error": "Missing or invalid Authorization header"}
            )

        key = auth_header.replace("Bearer ", "")
        db = SessionLocal()

        try:
            is_valid, api_key = await APIKeyManager.validate_key(db, key)

            if not is_valid:
                return JSONResponse(
                    status_code=401,
                    content={"error": "Invalid API key"}
                )

            request.scope["api_key"] = api_key
            request.scope["user_id"] = api_key.user_id

        finally:
            db.close()

    return await call_next(request)
```

### 🔒 CORS Configuration

**File:** `python-services/api/config/cors.py`

```
from fastapi.middleware.cors import CORSMiddleware

def setup_cors(app):
    """Setup CORS with strict policies"""

    if os.getenv("ENV") == "production":
        allowed_origins = [
            "https://app.construction-ai.com",
            "https://dashboard.construction-ai.com",
            "https://admin.construction-ai.com"
        ]
    else:
        allowed_origins = [
            "http://localhost:3000",
            "http://localhost:5000",
            "http://127.0.0.1:3000"
        ]

    app.add_middleware(
        CORSMiddleware,
        allow_origins=allowed_origins,
        allow_credentials=True,
        allow_methods=["GET", "POST", "PUT", "DELETE", "OPTIONS"],
        allow_headers=[
            "Content-Type",
            "Authorization",
            "X-Request-ID",
            "X-API-Key"
        ],
        expose_headers=[
            "X-Total-Count",
            "X-Page-Number",
            "X-RateLimit-Limit",
            "X-RateLimit-Remaining",
            "X-RateLimit-Reset"
        ],
        max_age=3600
    )

    @app.middleware("http")
    async def add_security_headers(request: Request, call_next):
        """Add security headers to all responses"""
        response = await call_next(request)

        response.headers["X-Content-Type-Options"] = "nosniff"
        response.headers["X-Frame-Options"] = "DENY"
        response.headers["X-XSS-Protection"] = "1; mode=block"
        response.headers["Strict-Transport-Security"] = "max-age=31536000; includeSubDoma
        response.headers["Content-Security-Policy"] = (
            "default-src 'self'; "
            "script-src 'self' 'unsafe-inline'; "
            "style-src 'self' 'unsafe-inline'; "
```

```
            "img-src 'self' data:; "
            "font-src 'self'"
        )

        return response
```

### 🛡️ Input Sanitization

**File:** `python-services/api/utils/sanitizer.py`

```
import re
import magic
from pathlib import Path

class InputSanitizer:
    ALLOWED_EXTENSIONS = {'.dwg', '.rvt', '.ifc', '.pdf', '.xlsx', '.csv', '.dxf', '.dgn'
    ALLOWED_MIMES = {
        'application/pdf',
        'application/vnd.ms-excel',
        'application/vnd.openxmlformats-officedocument.spreadsheetml.sheet',
        'text/plain',
        'text/csv',
        'application/octet-stream'
    }
    MAX_FILE_SIZE = 5 * 1024 * 1024 * 1024  # 5GB

    @staticmethod
    def sanitize_filename(filename: str) -&gt; str:
        """Remove dangerous characters from filename"""

        # Remove directory traversal attempts
        filename = filename.replace('../', '').replace('..\\', '')

        # Keep only safe characters
        filename = re.sub(r'[^a-zA-Z0-9._\-]', '_', filename)

        # Limit length
        if len(filename) &gt; 255:
            name, ext = filename.rsplit('.', 1)
            filename = name[:255 - len(ext) - 1] + '.' + ext

        return filename

    @staticmethod
    def validate_file(file_path: str, file_content: bytes) -&gt; tuple[bool, str]:
        """Validate file for upload"""

        # Check extension
        file_ext = Path(file_path).suffix.lower()
        if file_ext not in InputSanitizer.ALLOWED_EXTENSIONS:
            return False, f"File extension {file_ext} not allowed"

        # Check size
        if len(file_content) &gt; InputSanitizer.MAX_FILE_SIZE:
            return False, "File size exceeds maximum limit"
```

```
        # Check MIME type with libmagic
        try:
            mime_type = magic.from_buffer(file_content, mime=True)
            if mime_type not in InputSanitizer.ALLOWED_MIMES:
                return False, f"MIME type {mime_type} not allowed"
        except Exception as e:
            logger.warning(f"MIME type check failed: {str(e)}")

        # Scan for malware (if ClamAV available)
        try:
            result = scan_clamav(file_content)
            if result['infected']:
                return False, f"Malware detected: {result['virus_name']}"
        except Exception as e:
            logger.warning(f"Malware scan failed: {str(e)}")

        return True, ""

def scan_clamav(file_content: bytes) -&gt; dict:
    """Scan file with ClamAV"""
    import pyclamd

    clam = pyclamd.ClamD()
    if not clam.ping():
        return {"infected": False}

    result = clam.scan_stream(file_content)

    return {
        "infected": result is not None,
        "virus_name": result[1][0] if result else None
    }
```

### 🔍 Audit Logging

**File:** `python-services/api/security/audit_logger.py`

```
from enum import Enum
from datetime import datetime
from sqlalchemy import Column, String, DateTime, JSON, Integer

class AuditEventType(str, Enum):
    FILE_UPLOADED = "file_uploaded"
    FILE_DOWNLOADED = "file_downloaded"
    FILE_DELETED = "file_deleted"
    ANALYSIS_STARTED = "analysis_started"
    ANALYSIS_COMPLETED = "analysis_completed"
    API_KEY_CREATED = "api_key_created"
    API_KEY_ROTATED = "api_key_rotated"
    USER_LOGIN = "user_login"
    USER_LOGOUT = "user_logout"
    PERMISSION_CHANGED = "permission_changed"

class AuditLog(Base):
```

```
    __tablename__ = "audit_logs"

    id = Column(Integer, primary_key=True)
    timestamp = Column(DateTime, default=datetime.now, index=True)
    user_id = Column(String, index=True)
    event_type = Column(String, index=True)
    resource_id = Column(String, index=True)
    resource_type = Column(String)
    action = Column(String)
    ip_address = Column(String)
    user_agent = Column(String)
    changes = Column(JSON)
    status = Column(String)

class AuditLogger:
    @staticmethod
    async def log_event(
        db: Session,
        user_id: str,
        event_type: AuditEventType,
        resource_id: str,
        resource_type: str,
        action: str,
        request: Request,
        changes: dict = None,
        status: str = "success"
    ):
        """Log audit event"""

        audit_log = AuditLog(
            timestamp=datetime.now(),
            user_id=user_id,
            event_type=event_type.value,
            resource_id=resource_id,
            resource_type=resource_type,
            action=action,
            ip_address=request.client.host if request.client else None,
            user_agent=request.headers.get("User-Agent", "Unknown"),
            changes=changes,
            status=status
        )

        db.add(audit_log)
        db.commit()

        logger.info(f"Audit: {event_type.value} - {action}")

# Usage
@router.post("/files/upload")
async def upload_file(
    request: Request,
    file: UploadFile,
    db: Session = Depends(get_db),
    user_id: str = Depends(get_current_user)
):
    """Upload file with audit logging"""
```

```
    try:
        # Upload logic
        file_record = create_file_record(file)

        await AuditLogger.log_event(
            db=db,
            user_id=user_id,
            event_type=AuditEventType.FILE_UPLOADED,
            resource_id=file_record.id,
            resource_type="File",
            action=f"Uploaded {file.filename}",
            request=request,
            changes={"filename": file.filename, "size": file.size}
        )

        return {"file_id": file_record.id}

    except Exception as e:
        await AuditLogger.log_event(
            db=db,
            user_id=user_id,
            event_type=AuditEventType.FILE_UPLOADED,
            resource_id="unknown",
            resource_type="File",
            action="Upload attempt",
            request=request,
            status="failure"
        )
        raise
```

## Phase 2 Deployment Steps

### Step 1: Database Migrations

```
# Add indexes
psql -U user -d construction_ai -f migrations/add_strategic_indexes.sql

# Verify indexes
psql -U user -d construction_ai -c "SELECT * FROM pg_stat_user_indexes;"
```

### Step 2: Update FastAPI Service

```
# Add dependencies
pip install slowapi pyclamd python-magic

# Deploy updated API
docker-compose up -d api
```

### Step 3: N8N Workflow Consolidation

1. Log into N8N
2. Create Master_File_Processing workflow
3. Test with sample files
4. Redirect webhooks
5. Archive old workflows

### Step 4: Security Setup

```
# Generate API keys
python scripts/generate_api_keys.py

# Setup key rotation cron job
0 2 * * * python scripts/rotate_api_keys.py
```

### Step 5: Monitor & Verify

```
# Check pool status
curl http://localhost:8000/admin/db-pool-status

# Check slow queries
curl http://localhost:8000/admin/slow-queries

# Check API metrics
curl http://localhost:8000/metrics
```

### ✅ Phase 2 Testing Checklist

- [ ] Rate limiting blocks requests after limit
- [ ] Rate limit headers present in response
- [ ] Pagination with cursors works correctly
- [ ] Database indexes improve query speed
- [ ] Connection pool manages connections efficiently
- [ ] N8N master workflow routes correctly by file type
- [ ] Error recovery retries work with backoff
- [ ] API key rotation works
- [ ] CORS headers correct
- [ ] Input validation rejects invalid files
- [ ] Audit logs capture all events
- [ ] Circuit breaker handles API failures

## 🎯 Success Metrics (Phase 2)

✅ **Expected Results:**

- API response time < 500ms (p95)
- Error rate < 0.1%
- Database query time < 100ms (p95)
- 30% performance improvement
- 50% error rate reduction
- Successful file processing 98%+

## 🚀 After Phase 2

When Phase 2 is complete, you're ready for:

1. **Phase 3:** Advanced Features (multi-tenancy, billing, advanced monitoring)
2. **Production Deployment** with confidence
3. **Enterprise Scaling** capabilities

**Phase 2 Total Effort: ~120 hours**
**Estimated Completion: 4 weeks**

**Ready to implement Phase 2? Start with database indexes, then move to API optimization!**