🚀 Phase 1 Implementation Guide

Quick Wins for Construction AI Platform

📋 Phase 1 Overview

Timeline: Week 1-2
Total Effort: 60 hours
Expected Impact: 40% UX improvement

Quick Wins Improvements:

1. WebSocket Real-Time Updates
2. File Progress Indicators
3. Basic Analytics Dashboard
4. Workflow Error Recovery
5. Basic Monitoring Setup

1️⃣ WebSocket Real-Time Updates

🎯 Goal

Enable real-time file processing status updates from N8N to React UI

🏗️ Architecture

```
React UI
  ↓ (WebSocket.connect)
  ├──→ FastAPI WebSocket Endpoint (/ws/uploads/{file_id})
  │
  ├──→ Event: "status_updated"
  ├──→ Payload: {status, progress, message, timestamp}
  │
  └──→ React Component Updates (setState)

N8N Workflow
  ↓
  └──→ FastAPI /webhook/file-status
      ├──→ Broadcast to WebSocket clients
      └──→ Update Redis cache
```

🔧 Backend Implementation (FastAPI)

File: `python-services/api/websocket_manager.py`

```
from fastapi import FastAPI, WebSocket, WebSocketDisconnect
from typing import List, Dict, Set
import json
import redis
from datetime import datetime

class ConnectionManager:
    def __init__(self):
        self.active_connections: Dict[str, List[WebSocket]] = {}
        self.redis_client = redis.Redis(host='redis', port=6379, decode_responses=True)

    async def connect(self, file_id: str, websocket: WebSocket):
        await websocket.accept()
        if file_id not in self.active_connections:
            self.active_connections[file_id] = []
        self.active_connections[file_id].append(websocket)
        print(f"Client connected to file {file_id}")

    def disconnect(self, file_id: str, websocket: WebSocket):
        if file_id in self.active_connections:
            self.active_connections[file_id].remove(websocket)
            if len(self.active_connections[file_id]) == 0:
                del self.active_connections[file_id]
        print(f"Client disconnected from file {file_id}")

    async def broadcast_status(self, file_id: str, data: dict):
        """Broadcast status update to all connected clients for this file"""
        if file_id in self.active_connections:
            disconnected = []

            for connection in self.active_connections[file_id]:
                try:
                    await connection.send_json({
                        "type": "status_update",
                        "timestamp": datetime.now().isoformat(),
                        "data": data
                    })
                except Exception as e:
                    print(f"Error sending message: {e}")
                    disconnected.append(connection)

            # Remove disconnected clients
            for connection in disconnected:
                self.disconnect(file_id, connection)

manager = ConnectionManager()

# Add to FastAPI app
app = FastAPI()

@app.websocket("/ws/uploads/{file_id}")
async def websocket_endpoint(file_id: str, websocket: WebSocket):
```

```
    await manager.connect(file_id, websocket)
    try:
        while True:
            # Keep connection alive
            data = await websocket.receive_text()
            if data == "ping":
                await websocket.send_json({"type": "pong"})
    except WebSocketDisconnect:
        manager.disconnect(file_id, websocket)

@app.post("/webhook/file-status")
async def file_status_update(payload: dict):
    """
    Receive status updates from N8N
    Payload: {file_id, status, progress, message}
    """
    file_id = payload.get("file_id")
    status = payload.get("status")  # "uploading", "processing", "completed", "failed"
    progress = payload.get("progress", 0)  # 0-100
    message = payload.get("message", "")

    # Store in Redis for persistence
    cache_key = f"file_status:{file_id}"
    manager.redis_client.setex(
        cache_key,
        3600,  # 1 hour
        json.dumps({
            "status": status,
            "progress": progress,
            "message": message,
            "timestamp": datetime.now().isoformat()
        })
    )

    # Broadcast to connected WebSocket clients
    await manager.broadcast_status(file_id, {
        "status": status,
        "progress": progress,
        "message": message
    })

    return {"success": True, "file_id": file_id}

@app.get("/api/files/{file_id}/status")
async def get_file_status(file_id: str):
    """Get current status of file processing"""
    cache_key = f"file_status:{file_id}"
    status_data = manager.redis_client.get(cache_key)

    if status_data:
        return json.loads(status_data)

    # Fallback to database
    from sqlalchemy.orm import Session
    from .database import get_db
```

```
    db = next(get_db())
    file_record = db.query(File).filter(File.id == file_id).first()

    if not file_record:
        return {"error": "File not found"}

    return {
        "status": file_record.status,
        "progress": file_record.progress or 0,
        "message": file_record.message or "",
        "timestamp": file_record.updated_at.isoformat()
    }
```

⚛ Frontend Implementation (React)

File: `web-react/src/components/FileUploadProgress.jsx`

```
import React, { useState, useEffect, useRef } from 'react';
import { CircularProgressbar } from 'react-circular-progressbar';
import 'react-circular-progressbar/dist/styles.css';

export const FileUploadProgress = ({ fileId, onComplete }) =&gt; {
  const [status, setStatus] = useState('idle');
  const [progress, setProgress] = useState(0);
  const [message, setMessage] = useState('');
  const wsRef = useRef(null);
  const reconnectAttempts = useRef(0);
  const maxReconnectAttempts = 5;

  useEffect(() =&gt; {
    connectWebSocket();

    return () =&gt; {
      if (wsRef.current) {
        wsRef.current.close();
      }
    };
  }, [fileId]);

  const connectWebSocket = () =&gt; {
    const protocol = window.location.protocol === 'https:' ? 'wss:' : 'ws:';
    const wsUrl = `${protocol}//${window.location.host}/ws/uploads/${fileId}`;

    try {
      wsRef.current = new WebSocket(wsUrl);

      wsRef.current.onopen = () =&gt; {
        console.log('WebSocket connected');
        reconnectAttempts.current = 0;
        // Send ping every 30 seconds to keep connection alive
        setInterval(() =&gt; {
          if (wsRef.current &amp;amp;&amp;amp; wsRef.current.readyState === WebSocket.OPEN) {
            wsRef.current.send('ping');
          }
        }, 30000);
```

```
      };

      wsRef.current.onmessage = (event) =&gt; {
        const data = JSON.parse(event.data);

        if (data.type === 'status_update') {
          const { status: newStatus, progress: newProgress, message: newMessage } = data.

          setStatus(newStatus);
          setProgress(newProgress || 0);
          setMessage(newMessage || '');

          if (newStatus === 'completed') {
            onComplete &amp;&amp; onComplete(true);
          } else if (newStatus === 'failed') {
            onComplete &amp;&amp; onComplete(false);
          }
        }
      };

      wsRef.current.onerror = (error) =&gt; {
        console.error('WebSocket error:', error);
        setStatus('error');
        setMessage('Connection error');
      };

      wsRef.current.onclose = () =&gt; {
        console.log('WebSocket disconnected');
        if (reconnectAttempts.current &lt; maxReconnectAttempts) {
          reconnectAttempts.current++;
          setTimeout(connectWebSocket, 3000 * reconnectAttempts.current);
        }
      };
    } catch (error) {
      console.error('Failed to connect WebSocket:', error);
    }
  };

  const statusColors = {
    idle: '#ccc',
    uploading: '#2196F3',
    processing: '#FF9800',
    completed: '#4CAF50',
    failed: '#f44336',
    error: '#f44336'
  };

  const statusMessages = {
    idle: 'Ready',
    uploading: 'Uploading...',
    processing: 'Processing...',
    completed: 'Completed!',
    failed: 'Failed',
    error: 'Connection Error'
  };
```

```
  return (
    <div>
      <div>
        &lt;CircularProgressbar
          value={progress}
          text={`${progress}%`}
          styles={{
            path: { stroke: statusColors[status] },
            text: { fill: '#333', fontSize: '24px', fontWeight: 'bold' }
          }}
        /&gt;
      </div>

      <div>
        <h3>
          {statusMessages[status]}
        </h3>
        <p>{message}</p>

        {status === 'error' &amp;&amp; (
          &lt;button
            onClick={connectWebSocket}
            style={styles.retryButton}
          &gt;
            Reconnect
          &lt;/button&gt;
        )}
      </div>
    </div>
  );
};

const styles = {
  container: {
    display: 'flex',
    alignItems: 'center',
    gap: '20px',
    padding: '20px',
    border: '1px solid #eee',
    borderRadius: '8px',
    backgroundColor: '#f9f9f9'
  },
  progressContainer: {
    width: '100px',
    height: '100px'
  },
  infoContainer: {
    flex: 1
  },
  statusHeader: {
    margin: '0 0 8px 0',
    color: '#333'
  },
  message: {
    margin: '0',
    color: '#666',
```

```
    fontSize: '14px'
  },
  retryButton: {
    marginTop: '8px',
    padding: '8px 16px',
    backgroundColor: '#2196F3',
    color: 'white',
    border: 'none',
    borderRadius: '4px',
    cursor: 'pointer'
  }
};
```

🔧 N8N Webhook Setup

In your N8N Master Workflow:

1. Add HTTP Request node after processing completes
2. Configure POST request to: `http://api:8000/webhook/file-status`
3. Set body:

```
{
  "file_id": "{{ $json.file_id }}",
  "status": "{{ $json.status }}",
  "progress": "{{ $json.progress }}",
  "message": "{{ $json.message }}"
}
```

4. Add this at different workflow stages:
   - After file upload: status="uploading", progress=10
   - Before processing: status="processing", progress=25
   - During processing: status="processing", progress=50
   - Before completion: status="processing", progress=90
   - After completion: status="completed", progress=100

2️⃣ File Progress Indicators

🎯 Goal

Show visual progress for file uploads and processing

🔧 Implementation

File: `web-react/src/components/UploadArea.jsx`

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

export const UploadArea = ({ onUploadStart, onUploadProgress, onUploadComplete }) =&gt; {
  const [isDragging, setIsDragging] = useState(false);
  const [uploads, setUploads] = useState({});
  const fileInputRef = React.useRef();

  const handleDragOver = (e) =&gt; {
    e.preventDefault();
    setIsDragging(true);
  };

  const handleDragLeave = () =&gt; {
    setIsDragging(false);
  };

  const uploadFile = async (file) =&gt; {
    const fileId = `${Date.now()}_${file.name}`;

    // Initialize upload state
    setUploads(prev =&gt; ({
      ...prev,
      [fileId]: {
        name: file.name,
        size: file.size,
        progress: 0,
        status: 'uploading',
        speed: '0 MB/s',
        eta: 'Calculating...'
      }
    }));

    onUploadStart &amp;&amp; onUploadStart(fileId, file);

    const formData = new FormData();
    formData.append('file', file);

    try {
      const startTime = Date.now();
      let uploadedBytes = 0;

      const response = await axios.post(
        '/api/files/upload',
        formData,
        {
          headers: { 'Content-Type': 'multipart/form-data' },
          onUploadProgress: (progressEvent) =&gt; {
            uploadedBytes = progressEvent.loaded;
            const progress = Math.round(
              (progressEvent.loaded / progressEvent.total) * 100
            );
```

```
          const elapsed = (Date.now() - startTime) / 1000;
          const speed = uploadedBytes / elapsed / (1024 * 1024); // MB/s
          const remaining = progressEvent.total - progressEvent.loaded;
          const eta = Math.round(remaining / (speed * 1024 * 1024));

          setUploads(prev =&gt; ({
            ...prev,
            [fileId]: {
              ...prev[fileId],
              progress,
              speed: `${speed.toFixed(2)} MB/s`,
              eta: `${eta}s`
            }
          }));

          onUploadProgress &amp;&amp; onUploadProgress(fileId, progress);
        }
      }
    );

    setUploads(prev =&gt; ({
      ...prev,
      [fileId]: {
        ...prev[fileId],
        status: 'completed',
        progress: 100
      }
    }));

    onUploadComplete &amp;&amp; onUploadComplete(fileId, response.data);

    // Keep in list for 2 seconds then remove
    setTimeout(() =&gt; {
      setUploads(prev =&gt; {
        const newState = { ...prev };
        delete newState[fileId];
        return newState;
      });
    }, 2000);

  } catch (error) {
    setUploads(prev =&gt; ({
      ...prev,
      [fileId]: {
        ...prev[fileId],
        status: 'failed',
        error: error.message
      }
    }));
  }
};

const handleDrop = async (e) =&gt; {
  e.preventDefault();
  setIsDragging(false);
```

```
  const files = Array.from(e.dataTransfer.files);

  for (const file of files) {
    // Validate file
    const validExtensions = ['.dwg', '.rvt', '.ifc', '.pdf', '.xlsx', '.csv'];
    const hasValidExtension = validExtensions.some(ext =&gt;
      file.name.toLowerCase().endsWith(ext)
    );

    if (!hasValidExtension) {
      alert(`Invalid file type: ${file.name}`);
      continue;
    }

    if (file.size &gt; 5 * 1024 * 1024 * 1024) { // 5GB
      alert(`File too large: ${file.name}`);
      continue;
    }

    await uploadFile(file);
  }
};

const handleFileSelect = (e) =&gt; {
  const files = Array.from(e.target.files);
  files.forEach(uploadFile);
};

return (
  <div>
    {/* Upload Area */}
    <div> fileInputRef.current?.click()}
      style={{
        ...styles.uploadArea,
        backgroundColor: isDragging ? '#e3f2fd' : '#f5f5f5',
        borderColor: isDragging ? '#2196F3' : '#ddd'
      }}
    &gt;
      &lt;input
        ref={fileInputRef}
        type="file"
        multiple
        onChange={handleFileSelect}
        style={{ display: 'none' }}
        accept=".dwg,.rvt,.ifc,.pdf,.xlsx,.csv"
      /&gt;
      <div>
        <div></div>
        <p>
          Drag &amp;amp; drop files here or click to select
        </p>
        <p>
          Supported: DWG, RVT, IFC, PDF, Excel, CSV
        </p>
      </div>
```

```
      </div>

      {/* Upload Progress List */}
      <div>
        {Object.entries(uploads).map(([fileId, upload]) =&gt; (
          <div>
            <div>
              <span>{upload.name}</span>
              <span>
                {upload.status === 'uploading' ? '⬆ Uploading' :
                 upload.status === 'completed' ? '✅ Completed' :
                 upload.status === 'failed' ? '❌ Failed' : '⏳ Processing'}
              </span>
            </div>

            <div>
              <div>
            </div>

            <div>
              <span>{upload.progress}%</span>
              <span>{upload.speed}</span>
              <span>ETA: {upload.eta}</span>
            </div>
          </div>
        ))}
      </div>
    </div>
  );
};

const styles = {
  uploadArea: {
    border: '2px dashed',
    borderRadius: '8px',
    padding: '40px',
    textAlign: 'center',
    cursor: 'pointer',
    transition: 'all 0.3s ease'
  },
  uploadContent: {
    pointerEvents: 'none'
  },
  uploadIcon: {
    fontSize: '48px',
    marginBottom: '16px'
  },
  uploadText: {
    margin: '0 0 8px 0',
    fontSize: '16px',
    fontWeight: '500',
    color: '#333'
  },
  uploadSubtext: {
    margin: '0',
    fontSize: '14px',
```

```
    color: '#999'
  },
  uploadList: {
    marginTop: '20px'
  },
  uploadItem: {
    marginBottom: '16px',
    padding: '12px',
    border: '1px solid #eee',
    borderRadius: '4px',
    backgroundColor: '#fafafa'
  },
  uploadItemHeader: {
    display: 'flex',
    justifyContent: 'space-between',
    marginBottom: '8px',
    fontSize: '14px'
  },
  uploadItemName: {
    fontWeight: '500',
    color: '#333'
  },
  uploadItemStatus: {
    color: '#666'
  },
  progressBar: {
    height: '6px',
    backgroundColor: '#e0e0e0',
    borderRadius: '3px',
    overflow: 'hidden',
    marginBottom: '8px'
  },
  progressBarFill: {
    height: '100%',
    backgroundColor: '#2196F3',
    transition: 'width 0.3s ease'
  },
  uploadItemDetails: {
    display: 'flex',
    justifyContent: 'space-between',
    fontSize: '12px',
    color: '#999'
  }
};
```

3 Basic Analytics Dashboard

🎯 Goal

Show key metrics and visualizations

📋 Implementation

File: `web-react/src/pages/Dashboard.jsx`

```
import React, { useState, useEffect } from 'react';
import { LineChart, Line, BarChart, Bar, PieChart, Pie, Cell, XAxis, YAxis, CartesianGrid
import axios from 'axios';

export const Dashboard = () =&gt; {
  const [analytics, setAnalytics] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() =&gt; {
    fetchAnalytics();
    // Refresh every 30 seconds
    const interval = setInterval(fetchAnalytics, 30000);
    return () =&gt; clearInterval(interval);
  }, []);

  const fetchAnalytics = async () =&gt; {
    try {
      const response = await axios.get('/api/analytics/dashboard');
      setAnalytics(response.data);
      setLoading(false);
    } catch (error) {
      console.error('Failed to fetch analytics:', error);
    }
  };

  if (loading) return <div>Loading...</div>;

  return (
    <div>
      {/* KPI Cards */}
      <div>
        &lt;KPICard
          title="Files Processed"
          value={analytics.total_files || 0}
          icon="📁"
          trend="+12%"
        /&gt;
        &lt;KPICard
          title="Avg Processing Time"
          value={`${analytics.avg_processing_time || 0}s`}
          icon="⏱️"
          trend="-8%"
        /&gt;
        &lt;KPICard
          title="Error Rate"
          value={`${analytics.error_rate || 0}%`}
          icon="❌"
```

```
          trend="-3%"
        /&gt;
        &lt;KPICard
          title="Success Rate"
          value={`${analytics.success_rate || 0}%`}
          icon="✅"
          trend="+5%"
        /&gt;
      </div>

      {/* Charts */}
      <div>
        {/* Processing Time Trend */}
        <div>
          <h3>Processing Time Trend (Last 7 Days)</h3>
          &lt;ResponsiveContainer width="100%" height={300}&gt;
            &lt;LineChart data={analytics.processing_trend || []}&gt;
              &lt;CartesianGrid strokeDasharray="3 3" /&gt;
              &lt;XAxis dataKey="date" /&gt;
              &lt;YAxis /&gt;
              &lt;Tooltip /&gt;
              &lt;Line type="monotone" dataKey="time" stroke="#2196F3" /&gt;
            &lt;/LineChart&gt;
          &lt;/ResponsiveContainer&gt;
        </div>

        {/* Files by Type */}
        <div>
          <h3>Files by Type</h3>
          &lt;ResponsiveContainer width="100%" height={300}&gt;
            &lt;PieChart&gt;
              &lt;Pie
                data={analytics.files_by_type || []}
                dataKey="count"
                label
              &gt;
                {(analytics.files_by_type || []).map((entry, index) =&gt; (
                  &lt;Cell key={`cell-${index}`} fill={COLORS[index % COLORS.length]} /&g
                ))}
              &lt;/Pie&gt;
              &lt;Tooltip /&gt;
            &lt;/PieChart&gt;
          &lt;/ResponsiveContainer&gt;
        </div>

        {/* Daily Files */}
        <div>
          <h3>Daily File Processing</h3>
          &lt;ResponsiveContainer width="100%" height={300}&gt;
            &lt;BarChart data={analytics.daily_files || []}&gt;
              &lt;CartesianGrid strokeDasharray="3 3" /&gt;
              &lt;XAxis dataKey="date" /&gt;
              &lt;YAxis /&gt;
              &lt;Tooltip /&gt;
              &lt;Bar dataKey="count" fill="#4CAF50" /&gt;
            &lt;/BarChart&gt;
```

```
          &lt;/ResponsiveContainer&gt;
        </div>
      </div>
    </div>
  );
};

const KPICard = ({ title, value, icon, trend }) =&gt; (
  <div>
    <div>{icon}</div>
    <div>
      <p>{title}</p>
      <p>{value}</p>
      <p>{trend}</p>
    </div>
  </div>
);

const COLORS = ['#8884d8', '#82ca9d', '#ffc658', '#ff7c7c', '#8dd1e1'];

const styles = {
  container: {
    padding: '20px',
    maxWidth: '1400px',
    margin: '0 auto'
  },
  kpiGrid: {
    display: 'grid',
    gridTemplateColumns: 'repeat(auto-fit, minmax(250px, 1fr))',
    gap: '16px',
    marginBottom: '32px'
  },
  kpiCard: {
    display: 'flex',
    alignItems: 'center',
    padding: '16px',
    border: '1px solid #eee',
    borderRadius: '8px',
    backgroundColor: 'white',
    boxShadow: '0 2px 4px rgba(0,0,0,0.1)'
  },
  kpiIcon: {
    fontSize: '32px',
    marginRight: '16px'
  },
  kpiContent: {
    flex: 1
  },
  kpiTitle: {
    margin: '0',
    fontSize: '12px',
    color: '#999',
    textTransform: 'uppercase'
  },
  kpiValue: {
    margin: '4px 0',
```

```
    fontSize: '24px',
    fontWeight: 'bold',
    color: '#333'
  },
  kpiTrend: {
    margin: '4px 0 0 0',
    fontSize: '12px',
    color: '#4CAF50'
  },
  chartsGrid: {
    display: 'grid',
    gridTemplateColumns: 'repeat(auto-fit, minmax(400px, 1fr))',
    gap: '20px'
  },
  chart: {
    padding: '16px',
    border: '1px solid #eee',
    borderRadius: '8px',
    backgroundColor: 'white'
  }
};
```

Backend Analytics Endpoint

File: `python-services/api/routes/analytics.py`

```
from fastapi import APIRouter, Depends
from sqlalchemy.orm import Session
from sqlalchemy import func, text
from datetime import datetime, timedelta
import json

router = APIRouter(prefix="/api/analytics", tags=["analytics"])

@router.get("/dashboard")
async def get_dashboard_analytics(db: Session = Depends(get_db)):
    """Get dashboard analytics data"""

    # Total files
    total_files = db.query(func.count(File.id)).scalar() or 0

    # Success rate
    completed = db.query(func.count(File.id)).filter(File.status == 'completed').scalar()
    success_rate = (completed / total_files * 100) if total_files &gt; 0 else 0
    error_rate = 100 - success_rate

    # Average processing time
    avg_time_result = db.query(func.avg(File.processing_time)).scalar()
    avg_processing_time = round(avg_time_result) if avg_time_result else 0

    # Processing trend (last 7 days)
    seven_days_ago = datetime.now() - timedelta(days=7)
    processing_trend = db.query(
        func.date(File.created_at).label('date'),
        func.avg(File.processing_time).label('time')
```

```
    ).filter(
        File.created_at &gt;= seven_days_ago
    ).group_by(func.date(File.created_at)).all()

    # Files by type
    files_by_type = db.query(
        File.file_type,
        func.count(File.id).label('count')
    ).group_by(File.file_type).all()

    # Daily files (last 7 days)
    daily_files = db.query(
        func.date(File.created_at).label('date'),
        func.count(File.id).label('count')
    ).filter(
        File.created_at &gt;= seven_days_ago
    ).group_by(func.date(File.created_at)).all()

    return {
        "total_files": total_files,
        "success_rate": round(success_rate, 1),
        "error_rate": round(error_rate, 1),
        "avg_processing_time": avg_processing_time,
        "processing_trend": [
            {"date": str(p[^0]), "time": round(p[^1]) if p[^1] else 0}
            for p in processing_trend
        ],
        "files_by_type": [
            {"name": f[^0], "count": f[^1]}
            for f in files_by_type
        ],
        "daily_files": [
            {"date": str(d[^0]), "count": d[^1]}
            for d in daily_files
        ]
    }
```

4 Workflow Error Recovery

Goal

Implement automatic retry logic in N8N workflows

Implementation

In your N8N Master Workflow:

Add this configuration node at the beginning:

```
{
  "type": "code",
  "name": "Error Recovery Config",
```

```
  "script": "
    // Define retry strategy
    return {
      maxRetries: 3,
      retryDelays: [1000, 2000, 4000],  // milliseconds
      retryableErrors: ['network_error', 'timeout', 'service_unavailable'],
      circuitBreaker: {
        failureThreshold: 5,
        resetTimeout: 60000
      }
    };
  "
}
```

Add Error Handler Node:

```
{
  "type": "if",
  "name": "Has Errors",
  "conditions": [
    {
      "type": "hasErrors"
    }
  ],
  "then": [
    {
      "type": "code",
      "name": "Implement Retry Logic",
      "script": "
        const error = $input.all();
        const retryCount = $input.param('retryCount') || 0;
        const maxRetries = 3;

        if (retryCount &lt; maxRetries) {
          // Calculate exponential backoff
          const delay = Math.pow(2, retryCount) * 1000;

          return {
            shouldRetry: true,
            retryCount: retryCount + 1,
            nextRetryIn: delay,
            error: error
          };
        }

        return {
          shouldRetry: false,
          retryCount: retryCount,
          error: error,
          message: 'Max retries exceeded'
        };
      "
    }
```

```
  ]
}
```

Backend Retry Decorator

File: `python-services/api/utils/retry.py`

```
import asyncio
import logging
from functools import wraps
from typing import Callable, Any
import random

logger = logging.getLogger(__name__)

def retry_with_exponential_backoff(
    max_retries: int = 3,
    base_delay: float = 1.0,
    max_delay: float = 60.0,
    exponential_base: float = 2.0
):
    """
    Decorator for retrying failed operations with exponential backoff
    """
    def decorator(func: Callable) -&gt; Callable:
        @wraps(func)
        async def async_wrapper(*args, **kwargs) -&gt; Any:
            last_exception = None

            for attempt in range(1, max_retries + 1):
                try:
                    return await func(*args, **kwargs)
                except Exception as e:
                    last_exception = e

                    if attempt == max_retries:
                        raise

                    # Calculate exponential backoff with jitter
                    delay = min(
                        base_delay * (exponential_base ** (attempt - 1)),
                        max_delay
                    )
                    jitter = delay * random.uniform(0.8, 1.2)

                    logger.warning(
                        f"Attempt {attempt} failed for {func.__name__}: {str(e)}. "
                        f"Retrying in {jitter:.2f}s..."
                    )

                    await asyncio.sleep(jitter)

            raise last_exception

        @wraps(func)
```

```
        def sync_wrapper(*args, **kwargs) -&gt; Any:
            last_exception = None

            for attempt in range(1, max_retries + 1):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    last_exception = e

                    if attempt == max_retries:
                        raise

                    delay = min(
                        base_delay * (exponential_base ** (attempt - 1)),
                        max_delay
                    )
                    jitter = delay * random.uniform(0.8, 1.2)

                    logger.warning(
                        f"Attempt {attempt} failed for {func.__name__}: {str(e)}. "
                        f"Retrying in {jitter:.2f}s..."
                    )

                    asyncio.sleep(jitter)

            raise last_exception

        return async_wrapper if asyncio.iscoroutinefunction(func) else sync_wrapper

    return decorator

# Usage
@retry_with_exponential_backoff(max_retries=3)
async def call_converter_service(file_path: str):
    response = await httpx.post(
        'http://dwg-service:5055/convert',
        json={'file_path': file_path}
    )
    return response.json()
```

5 Basic Monitoring Setup

Goal

Set up Prometheus metrics and basic Grafana dashboard

Implementation

Add Prometheus Metrics to FastAPI:

File: `python-services/api/monitoring.py`

```
from prometheus_client import Counter, Histogram, Gauge, generate_latest
from fastapi import Response
import time

# Define metrics
file_uploads = Counter(
    'file_uploads_total',
    'Total file uploads',
    ['status', 'file_type']
)

processing_time = Histogram(
    'processing_time_seconds',
    'File processing time in seconds',
    buckets=(0.1, 0.5, 1.0, 2.0, 5.0, 10.0, 30.0, 60.0)
)

active_processes = Gauge(
    'active_processes',
    'Number of active processes'
)

error_count = Counter(
    'errors_total',
    'Total errors',
    ['error_type']
)

@app.get("/metrics")
async def metrics():
    """Prometheus metrics endpoint"""
    return Response(generate_latest(), media_type="text/plain")

# Middleware to track metrics
@app.middleware("http")
async def add_metrics(request, call_next):
    start_time = time.time()
    response = await call_next(request)
    process_time = time.time() - start_time

    return response
```

Prometheus Configuration:

File: `monitoring/prometheus.yml`

```
global:
  scrape_interval: 15s
  evaluation_interval: 15s

scrape_configs:
  - job_name: 'fastapi'
    static_configs:
      - targets: ['http://api:8000']
```

```
    metrics_path: '/metrics'

  - job_name: 'n8n'
    static_configs:
      - targets: ['http://n8n:5678']
    metrics_path: '/metrics'

  - job_name: 'postgres'
    static_configs:
      - targets: ['postgres_exporter:9187']
```

Basic Grafana Dashboard:

Import this JSON into Grafana:

```
{
  "dashboard": {
    "title": "Construction AI - Phase 1",
    "panels": [
      {
        "title": "Files Processed",
        "targets": [
          {
            "expr": "increase(file_uploads_total[1h])"
          }
        ]
      },
      {
        "title": "Processing Time (p95)",
        "targets": [
          {
            "expr": "histogram_quantile(0.95, processing_time_seconds_bucket)"
          }
        ]
      },
      {
        "title": "Error Rate",
        "targets": [
          {
            "expr": "rate(errors_total[5m])"
          }
        ]
      },
      {
        "title": "Active Processes",
        "targets": [
          {
            "expr": "active_processes"
          }
        ]
      }
    ]
  }
}
```

🚀 Deployment Steps

Step 1: Update FastAPI Service

```
# Add requirements
pip install websockets python-socketio redis

# Restart API service
docker-compose restart api
```

Step 2: Update React UI

```
cd web-react
npm install recharts axios

# Rebuild
npm run build
docker-compose restart ui
```

Step 3: Update N8N Workflows

1. Log into N8N (http://localhost:5678)
2. Open your Master Workflow
3. Add WebSocket update nodes (see section 1)
4. Add error handling nodes (see section 4)
5. Deploy and test

Step 4: Start Monitoring

```
docker-compose up -d prometheus grafana
```

Access Grafana at: http://localhost:3001

✅ Testing Checklist

- [ ] WebSocket connects when file processing starts
- [ ] Real-time progress updates flow to React UI
- [ ] Progress component displays correctly (0-100%)
- [ ] File upload area accepts drag-and-drop
- [ ] Upload speed and ETA calculations are accurate
- [ ] Dashboard displays charts without errors

- [ ] Analytics refresh every 30 seconds
- [ ] Workflow retries on network error
- [ ] Prometheus collects metrics
- [ ] Grafana dashboard displays data

Success Metrics

✅ **Expected Results:**

- 40% improvement in UX (real-time feedback)
- 95% file upload success rate
- Average processing time tracking
- Automatic retry handling
- Real-time monitoring capability

Next Steps (Phase 2)

After Phase 1 is complete:

1. API rate limiting
2. Database indexing
3. N8N workflow consolidation
4. Caching layer optimization
5. Advanced error handling

Phase 1 Total Effort: ~60 hours
Estimated Completion: 1-2 weeks

Ready to start implementing? Let me know if you need clarification on any step!

⁂