# **Local-First Development Approach: AEC File Processing System**

## **Development Philosophy: Start Simple, Scale Incrementally**

This approach begins with a single-machine implementation that provides immediate value while maintaining architectural patterns that naturally scale to the full distributed system.

## **Phase 0: Local Foundation (Week 1-2)**

### **Step 1: Set Up Local Development Environment**

bash

*# Create project structure*

mkdir aec-file-processor

cd aec-file-processor

*# Initialize solution*

dotnet new sln -n AECFileProcessor

*# Create core projects*

dotnet new classlib -n AECFileProcessor.Core

dotnet new console -n AECFileProcessor.CLI

dotnet new webapi -n AECFileProcessor.API

dotnet new xunit -n AECFileProcessor.Tests

*# Add projects to solution*

dotnet sln add \*\*/\*.csproj

*# Set up Git*

git init

git add .

git commit -m "Initial project structure"

### **Step 2: Core Domain Models (Local Implementation)**

csharp

*// AECFileProcessor.Core/Models/FileRecord.cs*

public class FileRecord

{

public Guid Id { get; set; } = Guid.NewGuid();

public string ProjectNumber { get; set; } = string.Empty;

public string FilePath { get; set; } = string.Empty;

public string FileName { get; set; } = string.Empty;

public long FileSize { get; set; }

public string FileHash { get; set; } = string.Empty;

public DateTime CreatedDate { get; set; }

public DateTime ModifiedDate { get; set; }

public DateTime ProcessedDate { get; set; }

*// AEC Classification*

public string Phase { get; set; } = string.Empty;

public string Discipline { get; set; } = string.Empty;

public string DocumentType { get; set; } = string.Empty;

public string SheetNumber { get; set; } = string.Empty;

public string RevisionNumber { get; set; } = string.Empty;

public bool IsCurrentRevision { get; set; } = true;

*// Processing Status*

public ProcessingStatus Status { get; set; } = ProcessingStatus.Discovered;

public string? ErrorMessage { get; set; }

*// Extracted Content (JSON for now)*

public string? ExtractedMetadata { get; set; }

}

public enum ProcessingStatus

{

Discovered,

Classified,

Processing,

Completed,

Failed

}

*// AECFileProcessor.Core/Models/ExtractedMetadata.cs*

public class ExtractedMetadata

{

public TitleBlockData? TitleBlock { get; set; }

public List<string> TextContent { get; set; } = new();

public List<string> TechnicalKeywords { get; set; } = new();

public Dictionary<string, string> Properties { get; set; } = new();

public DateTime ExtractedAt { get; set; } = DateTime.UtcNow;

public double ConfidenceScore { get; set; }

}

public class TitleBlockData

{

public string? ProjectName { get; set; }

public string? SheetTitle { get; set; }

public string? SheetNumber { get; set; }

public string? RevisionNumber { get; set; }

public string? DrawnBy { get; set; }

public string? CheckedBy { get; set; }

public string? Date { get; set; }

public string? Scale { get; set; }

}

### **Step 3: Local File System Watcher (Simplest Version)**

csharp

*// AECFileProcessor.Core/Services/LocalFileWatcher.cs*

public class LocalFileWatcher : IDisposable

{

private readonly FileSystemWatcher \_watcher;

private readonly IFileProcessor \_processor;

private readonly ILogger<LocalFileWatcher> \_logger;

private readonly Timer \_batchTimer;

private readonly ConcurrentQueue<string> \_fileQueue = new();

public LocalFileWatcher(string watchPath, IFileProcessor processor, ILogger<LocalFileWatcher> logger)

{

\_processor = processor;

\_logger = logger;

\_watcher = new FileSystemWatcher(watchPath)

{

IncludeSubdirectories = true,

NotifyFilter = NotifyFilters.FileName | NotifyFilters.LastWrite | NotifyFilters.CreationTime

};

\_watcher.Created += OnFileChanged;

\_watcher.Changed += OnFileChanged;

\_watcher.Renamed += OnFileRenamed;

*// Process files in batches every 5 seconds*

\_batchTimer = new Timer(ProcessQueuedFiles, null, TimeSpan.FromSeconds(5), TimeSpan.FromSeconds(5));

}

public void StartWatching()

{

\_watcher.EnableRaisingEvents = true;

\_logger.LogInformation($"Started watching {\_watcher.Path}");

*// Process existing files*

\_ = Task.Run(ProcessExistingFilesAsync);

}

private void OnFileChanged(object sender, FileSystemEventArgs e)

{

if (ShouldProcessFile(e.FullPath))

{

\_fileQueue.Enqueue(e.FullPath);

\_logger.LogDebug($"Queued file: {e.FullPath}");

}

}

private void OnFileRenamed(object sender, RenamedEventArgs e)

{

if (ShouldProcessFile(e.FullPath))

{

\_fileQueue.Enqueue(e.FullPath);

}

}

private bool ShouldProcessFile(string filePath)

{

var extension = Path.GetExtension(filePath).ToLower();

var fileName = Path.GetFileName(filePath);

*// Skip temporary files*

if (fileName.StartsWith("~") || fileName.StartsWith(".tmp") || fileName.Contains("$"))

return false;

*// Only process known AEC file types*

return extension is ".dwg" or ".pdf" or ".docx" or ".xlsx" or ".rvt" or ".ifc";

}

private async void ProcessQueuedFiles(object? state)

{

var filesToProcess = new List<string>();

*// Dequeue all pending files*

while (\_fileQueue.TryDequeue(out var filePath))

{

if (File.Exists(filePath))

{

filesToProcess.Add(filePath);

}

}

if (filesToProcess.Any())

{

\_logger.LogInformation($"Processing batch of {filesToProcess.Count} files");

*// Process files in parallel with limited concurrency*

var semaphore = new SemaphoreSlim(Environment.ProcessorCount);

var tasks = filesToProcess.Select(async filePath =>

{

await semaphore.WaitAsync();

try

{

await \_processor.ProcessFileAsync(filePath);

}

catch (Exception ex)

{

\_logger.LogError(ex, $"Failed to process file: {filePath}");

}

finally

{

semaphore.Release();

}

});

await Task.WhenAll(tasks);

}

}

private async Task ProcessExistingFilesAsync()

{

try

{

var files = Directory.GetFiles(\_watcher.Path, "\*.\*", SearchOption.AllDirectories)

.Where(ShouldProcessFile)

.ToList();

\_logger.LogInformation($"Found {files.Count} existing files to process");

foreach (var file in files)

{

\_fileQueue.Enqueue(file);

}

}

catch (Exception ex)

{

\_logger.LogError(ex, "Failed to process existing files");

}

}

public void Dispose()

{

\_watcher?.Dispose();

\_batchTimer?.Dispose();

}

}

### **Step 4: Simple File Classification**

csharp

*// AECFileProcessor.Core/Services/FileClassifier.cs*

public class FileClassifier : IFileClassifier

{

private readonly ILogger<FileClassifier> \_logger;

private static readonly Regex NamingConventionRegex = new(

@"^(?<project>\w+)\_(?<phase>\w+)\_(?<discipline>\w+)\_(?<doctype>\w+)\_(?<sheet>[\w-]+)\_(?<revision>[\w\d]+)\_(?<date>\d{4}-\d{2}-\d{2})\.(?<ext>\w+)$",

RegexOptions.Compiled | RegexOptions.IgnoreCase);

public FileClassifier(ILogger<FileClassifier> logger)

{

\_logger = logger;

}

public async Task<FileClassification> ClassifyAsync(string filePath)

{

var fileName = Path.GetFileName(filePath);

var fileInfo = new FileInfo(filePath);

var classification = new FileClassification

{

FilePath = filePath,

FileName = fileName,

FileSize = fileInfo.Length,

Extension = fileInfo.Extension.ToLower(),

CreatedDate = fileInfo.CreationTime,

ModifiedDate = fileInfo.LastWriteTime

};

*// Try to parse standardized naming convention*

var match = NamingConventionRegex.Match(fileName);

if (match.Success)

{

classification.ProjectNumber = match.Groups["project"].Value;

classification.Phase = MapPhase(match.Groups["phase"].Value);

classification.Discipline = MapDiscipline(match.Groups["discipline"].Value);

classification.DocumentType = MapDocumentType(match.Groups["doctype"].Value);

classification.SheetNumber = match.Groups["sheet"].Value;

classification.RevisionNumber = match.Groups["revision"].Value;

classification.IsStandardNaming = true;

classification.ConfidenceScore = 0.95;

}

else

{

*// Fallback classification based on directory structure and extension*

await ClassifyByDirectoryAndExtensionAsync(classification);

}

*// Determine directory context*

classification.DirectoryType = DetermineDirectoryType(filePath);

return classification;

}

private async Task ClassifyByDirectoryAndExtensionAsync(FileClassification classification)

{

var directoryPath = Path.GetDirectoryName(classification.FilePath) ?? "";

var pathParts = directoryPath.Split(Path.DirectorySeparatorChar);

*// Look for project number in path*

foreach (var part in pathParts)

{

if (Regex.IsMatch(part, @"^\w+\d{2,}"))

{

classification.ProjectNumber = part;

break;

}

}

*// Classify by extension and directory hints*

classification.DocumentType = classification.Extension switch

{

".dwg" => "Drawing",

".pdf" => pathParts.Any(p => p.Contains("Spec", StringComparison.OrdinalIgnoreCase)) ? "Specification" : "Drawing",

".docx" => "Specification",

".xlsx" => "Schedule",

".rvt" => "BIM\_Model",

".ifc" => "BIM\_Model",

\_ => "Document"

};

*// Try to infer discipline from directory names*

foreach (var part in pathParts.Reverse())

{

if (part.Contains("Architectural", StringComparison.OrdinalIgnoreCase))

{

classification.Discipline = "Architectural";

break;

}

if (part.Contains("Structural", StringComparison.OrdinalIgnoreCase))

{

classification.Discipline = "Structural";

break;

}

if (part.Contains("Mechanical", StringComparison.OrdinalIgnoreCase))

{

classification.Discipline = "Mechanical";

break;

}

if (part.Contains("Electrical", StringComparison.OrdinalIgnoreCase))

{

classification.Discipline = "Electrical";

break;

}

}

classification.ConfidenceScore = 0.6; *// Lower confidence for heuristic classification*

await Task.CompletedTask; *// Placeholder for future ML classification*

}

private string MapPhase(string phaseCode) => phaseCode.ToUpper() switch

{

"PD" => "Pre-Design",

"SD" => "Schematic Design",

"DD" => "Design Development",

"CD" => "Construction Documents",

"CA" => "Construction Administration",

"CO" => "Closeout",

\_ => phaseCode

};

private string MapDiscipline(string disciplineCode) => disciplineCode.ToUpper() switch

{

"A" => "Architectural",

"S" => "Structural",

"M" => "Mechanical",

"E" => "Electrical",

"P" => "Plumbing",

"F" => "Fire Protection",

"C" => "Civil",

"L" => "Landscape",

\_ => disciplineCode

};

private string MapDocumentType(string docTypeCode) => docTypeCode.ToUpper() switch

{

"DWG" => "Drawing",

"PLN" => "Plan",

"SEC" => "Section",

"DTL" => "Detail",

"SCH" => "Schedule",

"SPEC" => "Specification",

"CALC" => "Calculation",

"RPT" => "Report",

\_ => docTypeCode

};

private string DetermineDirectoryType(string filePath)

{

var directoryPath = Path.GetDirectoryName(filePath) ?? "";

if (directoryPath.Contains("02\_DRAWINGS", StringComparison.OrdinalIgnoreCase))

return "Drawings";

if (directoryPath.Contains("03\_SPECIFICATIONS", StringComparison.OrdinalIgnoreCase))

return "Specifications";

if (directoryPath.Contains("04\_CALCULATIONS", StringComparison.OrdinalIgnoreCase))

return "Calculations";

if (directoryPath.Contains("05\_REPORTS", StringComparison.OrdinalIgnoreCase))

return "Reports";

if (directoryPath.Contains("08\_MODELS\_CAD", StringComparison.OrdinalIgnoreCase))

return "Models";

return "Unknown";

}

}

public class FileClassification

{

public string FilePath { get; set; } = string.Empty;

public string FileName { get; set; } = string.Empty;

public long FileSize { get; set; }

public string Extension { get; set; } = string.Empty;

public DateTime CreatedDate { get; set; }

public DateTime ModifiedDate { get; set; }

public string ProjectNumber { get; set; } = string.Empty;

public string Phase { get; set; } = string.Empty;

public string Discipline { get; set; } = string.Empty;

public string DocumentType { get; set; } = string.Empty;

public string SheetNumber { get; set; } = string.Empty;

public string RevisionNumber { get; set; } = string.Empty;

public string DirectoryType { get; set; } = string.Empty;

public bool IsStandardNaming { get; set; }

public double ConfidenceScore { get; set; }

}

### **Step 5: SQLite Database for Local Storage**

csharp

*// AECFileProcessor.Core/Data/LocalDatabase.cs*

public class LocalDatabase : IDisposable

{

private readonly SqliteConnection \_connection;

private readonly ILogger<LocalDatabase> \_logger;

public LocalDatabase(string databasePath, ILogger<LocalDatabase> logger)

{

\_logger = logger;

var connectionString = $"Data Source={databasePath}";

\_connection = new SqliteConnection(connectionString);

\_connection.Open();

CreateTablesAsync().Wait();

}

private async Task CreateTablesAsync()

{

var createTablesScript = @"

CREATE TABLE IF NOT EXISTS FileRecords (

Id TEXT PRIMARY KEY,

ProjectNumber TEXT NOT NULL,

FilePath TEXT NOT NULL UNIQUE,

FileName TEXT NOT NULL,

FileSize INTEGER NOT NULL,

FileHash TEXT,

CreatedDate TEXT NOT NULL,

ModifiedDate TEXT NOT NULL,

ProcessedDate TEXT NOT NULL,

-- AEC Classification

Phase TEXT NOT NULL,

Discipline TEXT NOT NULL,

DocumentType TEXT NOT NULL,

SheetNumber TEXT,

RevisionNumber TEXT,

IsCurrentRevision INTEGER NOT NULL DEFAULT 1,

DirectoryType TEXT,

-- Processing Status

Status INTEGER NOT NULL DEFAULT 0,

ErrorMessage TEXT,

ConfidenceScore REAL,

-- Extracted Content

ExtractedMetadata TEXT,

-- Indexes

INDEX idx\_project\_number ON FileRecords(ProjectNumber),

INDEX idx\_discipline ON FileRecords(Discipline),

INDEX idx\_status ON FileRecords(Status),

INDEX idx\_modified\_date ON FileRecords(ModifiedDate)

);

CREATE TABLE IF NOT EXISTS ProcessingHistory (

Id TEXT PRIMARY KEY,

FileRecordId TEXT NOT NULL,

Operation TEXT NOT NULL,

Details TEXT,

Timestamp TEXT NOT NULL,

FOREIGN KEY (FileRecordId) REFERENCES FileRecords(Id)

);

";

using var command = new SqliteCommand(createTablesScript, \_connection);

await command.ExecuteNonQueryAsync();

\_logger.LogInformation("Database tables created successfully");

}

public async Task<string> UpsertFileRecordAsync(FileRecord record)

{

var sql = @"

INSERT OR REPLACE INTO FileRecords (

Id, ProjectNumber, FilePath, FileName, FileSize, FileHash,

CreatedDate, ModifiedDate, ProcessedDate,

Phase, Discipline, DocumentType, SheetNumber, RevisionNumber,

IsCurrentRevision, DirectoryType, Status, ErrorMessage,

ConfidenceScore, ExtractedMetadata

) VALUES (

@Id, @ProjectNumber, @FilePath, @FileName, @FileSize, @FileHash,

@CreatedDate, @ModifiedDate, @ProcessedDate,

@Phase, @Discipline, @DocumentType, @SheetNumber, @RevisionNumber,

@IsCurrentRevision, @DirectoryType, @Status, @ErrorMessage,

@ConfidenceScore, @ExtractedMetadata

)";

using var command = new SqliteCommand(sql, \_connection);

command.Parameters.AddWithValue("@Id", record.Id.ToString());

command.Parameters.AddWithValue("@ProjectNumber", record.ProjectNumber);

command.Parameters.AddWithValue("@FilePath", record.FilePath);

command.Parameters.AddWithValue("@FileName", record.FileName);

command.Parameters.AddWithValue("@FileSize", record.FileSize);

command.Parameters.AddWithValue("@FileHash", record.FileHash ?? "");

command.Parameters.AddWithValue("@CreatedDate", record.CreatedDate.ToString("O"));

command.Parameters.AddWithValue("@ModifiedDate", record.ModifiedDate.ToString("O"));

command.Parameters.AddWithValue("@ProcessedDate", record.ProcessedDate.ToString("O"));

command.Parameters.AddWithValue("@Phase", record.Phase);

command.Parameters.AddWithValue("@Discipline", record.Discipline);

command.Parameters.AddWithValue("@DocumentType", record.DocumentType);

command.Parameters.AddWithValue("@SheetNumber", record.SheetNumber ?? "");

command.Parameters.AddWithValue("@RevisionNumber", record.RevisionNumber ?? "");

command.Parameters.AddWithValue("@IsCurrentRevision", record.IsCurrentRevision ? 1 : 0);

command.Parameters.AddWithValue("@DirectoryType", record.Phase);

command.Parameters.AddWithValue("@Status", (int)record.Status);

command.Parameters.AddWithValue("@ErrorMessage", record.ErrorMessage ?? "");

command.Parameters.AddWithValue("@ConfidenceScore", 0.0);

command.Parameters.AddWithValue("@ExtractedMetadata", record.ExtractedMetadata ?? "");

await command.ExecuteNonQueryAsync();

return record.Id.ToString();

}

public async Task<List<FileRecord>> GetFilesByProjectAsync(string projectNumber)

{

var sql = "SELECT \* FROM FileRecords WHERE ProjectNumber = @ProjectNumber ORDER BY ModifiedDate DESC";

using var command = new SqliteCommand(sql, \_connection);

command.Parameters.AddWithValue("@ProjectNumber", projectNumber);

var records = new List<FileRecord>();

using var reader = await command.ExecuteReaderAsync();

while (await reader.ReadAsync())

{

records.Add(MapToFileRecord(reader));

}

return records;

}

public async Task<List<string>> GetProjectNumbersAsync()

{

var sql = "SELECT DISTINCT ProjectNumber FROM FileRecords ORDER BY ProjectNumber";

using var command = new SqliteCommand(sql, \_connection);

var projects = new List<string>();

using var reader = await command.ExecuteReaderAsync();

while (await reader.ReadAsync())

{

projects.Add(reader.GetString("ProjectNumber"));

}

return projects;

}

private FileRecord MapToFileRecord(SqliteDataReader reader)

{

return new FileRecord

{

Id = Guid.Parse(reader.GetString("Id")),

ProjectNumber = reader.GetString("ProjectNumber"),

FilePath = reader.GetString("FilePath"),

FileName = reader.GetString("FileName"),

FileSize = reader.GetInt64("FileSize"),

FileHash = reader.GetString("FileHash"),

CreatedDate = DateTime.Parse(reader.GetString("CreatedDate")),

ModifiedDate = DateTime.Parse(reader.GetString("ModifiedDate")),

ProcessedDate = DateTime.Parse(reader.GetString("ProcessedDate")),

Phase = reader.GetString("Phase"),

Discipline = reader.GetString("Discipline"),

DocumentType = reader.GetString("DocumentType"),

SheetNumber = reader.GetString("SheetNumber"),

RevisionNumber = reader.GetString("RevisionNumber"),

IsCurrentRevision = reader.GetInt32("IsCurrentRevision") == 1,

Status = (ProcessingStatus)reader.GetInt32("Status"),

ErrorMessage = reader.IsDBNull("ErrorMessage") ? null : reader.GetString("ErrorMessage"),

ExtractedMetadata = reader.IsDBNull("ExtractedMetadata") ? null : reader.GetString("ExtractedMetadata")

};

}

public void Dispose()

{

\_connection?.Dispose();

}

}

### **Step 6: Console Application (First Working Version)**

csharp

*// AECFileProcessor.CLI/Program.cs*

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Hosting;

using Microsoft.Extensions.Logging;

using System.CommandLine;

class Program

{

static async Task<int> Main(string[] args)

{

var rootCommand = new RootCommand("AEC File Processor - Local Version");

*// Watch command*

var watchCommand = new Command("watch", "Watch a directory for file changes");

var pathOption = new Option<string>("--path", "Directory path to watch") { IsRequired = true };

var dbOption = new Option<string>("--database", () => "aec\_files.db", "SQLite database path");

watchCommand.AddOption(pathOption);

watchCommand.AddOption(dbOption);

watchCommand.SetHandler(async (string path, string database) =>

{

await RunWatchModeAsync(path, database);

}, pathOption, dbOption);

*// Process command (one-time)*

var processCommand = new Command("process", "Process files in a directory once");

processCommand.AddOption(pathOption);

processCommand.AddOption(dbOption);

processCommand.SetHandler(async (string path, string database) =>

{

await RunProcessModeAsync(path, database);

}, pathOption, dbOption);

*// Query command*

var queryCommand = new Command("query", "Query processed files");

var projectOption = new Option<string>("--project", "Project number to query");

queryCommand.AddOption(projectOption);

queryCommand.AddOption(dbOption);

queryCommand.SetHandler(async (string? project, string database) =>

{

await RunQueryModeAsync(project, database);

}, projectOption, dbOption);

rootCommand.AddCommand(watchCommand);

rootCommand.AddCommand(processCommand);

rootCommand.AddCommand(queryCommand);

return await rootCommand.InvokeAsync(args);

}

static async Task RunWatchModeAsync(string watchPath, string databasePath)

{

var host = CreateHost(databasePath);

var services = host.Services;

var logger = services.GetRequiredService<ILogger<Program>>();

var processor = services.GetRequiredService<IFileProcessor>();

logger.LogInformation($"Starting file watcher for: {watchPath}");

logger.LogInformation($"Database: {databasePath}");

using var watcher = new LocalFileWatcher(watchPath, processor,

services.GetRequiredService<ILogger<LocalFileWatcher>>());

watcher.StartWatching();

logger.LogInformation("File watcher started. Press Ctrl+C to stop...");

var cts = new CancellationTokenSource();

Console.CancelKeyPress += (s, e) =>

{

e.Cancel = true;

cts.Cancel();

};

try

{

await Task.Delay(-1, cts.Token);

}

catch (OperationCanceledException)

{

logger.LogInformation("Shutting down...");

}

}

static async Task RunProcessModeAsync(string processPath, string databasePath)

{

var host = CreateHost(databasePath);

var services = host.Services;

var logger = services.GetRequiredService<ILogger<Program>>();

var processor = services.GetRequiredService<IFileProcessor>();

logger.LogInformation($"Processing files in: {processPath}");

var files = Directory.GetFiles(processPath, "\*.\*", SearchOption.AllDirectories)

.Where(f => ShouldProcessFile(f))

.ToList();

logger.LogInformation($"Found {files.Count} files to process");

var processed = 0;

var errors = 0;

foreach (var file in files)

{

try

{

await processor.ProcessFileAsync(file);

processed++;

if (processed % 10 == 0)

{

logger.LogInformation($"Processed {processed}/{files.Count} files");

}

}

catch (Exception ex)

{

logger.LogError(ex, $"Failed to process: {file}");

errors++;

}

}

logger.LogInformation($"Processing complete. Processed: {processed}, Errors: {errors}");

}

static async Task RunQueryModeAsync(string? projectNumber, string databasePath)

{

var host = CreateHost(databasePath);

var database = host.Services.GetRequiredService<LocalDatabase>();

if (string.IsNullOrEmpty(projectNumber))

{

*// Show all projects*

var projects = await database.GetProjectNumbersAsync();

Console.WriteLine($"Found {projects.Count} projects:");

foreach (var project in projects)

{

var files = await database.GetFilesByProjectAsync(project);

Console.WriteLine($" {project}: {files.Count} files");

}

}

else

{

*// Show files for specific project*

var files = await database.GetFilesByProjectAsync(projectNumber);

Console.WriteLine($"Project {projectNumber}: {files.Count} files");

Console.WriteLine();

foreach (var file in files.Take(20)) *// Limit output*

{

Console.WriteLine($" {file.FileName}");

Console.WriteLine($" Discipline: {file.Discipline}, Phase: {file.Phase}");

Console.WriteLine($" Status: {file.Status}, Modified: {file.ModifiedDate:yyyy-MM-dd HH:mm}");

Console.WriteLine();

}

if (files.Count > 20)

{

Console.WriteLine($" ... and {files.Count - 20} more files");

}

}

}

static IHost CreateHost(string databasePath)

{

return Host.CreateDefaultBuilder()

.ConfigureServices((context, services) =>

{

services.AddLogging(builder =>

{

builder.AddConsole();

builder.SetMinimumLevel(LogLevel.Information);

});

services.AddSingleton(provider =>

new LocalDatabase(databasePath, provider.GetRequiredService<ILogger<LocalDatabase>>()));

services.AddSingleton<IFileClassifier, FileClassifier>();

services.AddSingleton<IMetadataExtractor, BasicMetadataExtractor>();

services.AddSingleton<IFileProcessor, FileProcessor>();

})

.Build();

}

static bool ShouldProcessFile(string filePath)

{

var extension = Path.GetExtension(filePath).ToLower();

var fileName = Path.GetFileName(filePath);

if (fileName.StartsWith("~") || fileName.StartsWith(".tmp") || fileName.Contains("$"))

return false;

return extension is ".dwg" or ".pdf" or ".docx" or ".xlsx" or ".rvt" or ".ifc";

}

}

### **Step 7: Basic File Processor**

csharp

*// AECFileProcessor.Core/Services/FileProcessor.cs*

public class FileProcessor : IFileProcessor

{

private readonly LocalDatabase \_database;

private readonly IFileClassifier \_classifier;

private readonly IMetadataExtractor \_metadataExtractor;

private readonly ILogger<FileProcessor> \_logger;

public FileProcessor(

LocalDatabase database,

IFileClassifier classifier,

IMetadataExtractor metadataExtractor,

ILogger<FileProcessor> logger)

{

\_database = database;

\_classifier = classifier;

\_metadataExtractor = metadataExtractor;

\_logger = logger;

}

public async Task ProcessFileAsync(string filePath)

{

var stopwatch = Stopwatch.StartNew();

try

{

\_logger.LogDebug($"Processing file: {filePath}");

*// Skip if file doesn't exist or is locked*

if (!File.Exists(filePath) || IsFileLocked(filePath))

{

\_logger.LogWarning($"File not accessible: {filePath}");

return;

}

*// Classify the file*

var classification = await \_classifier.ClassifyAsync(filePath);

*// Create file record*

var fileRecord = new FileRecord

{

Id = Guid.NewGuid(),

FilePath = filePath,

FileName = classification.FileName,

FileSize = classification.FileSize,

FileHash = await ComputeFileHashAsync(filePath),

CreatedDate = classification.CreatedDate,

ModifiedDate = classification.ModifiedDate,

ProcessedDate = DateTime.UtcNow,

ProjectNumber = classification.ProjectNumber,

Phase = classification.Phase,

Discipline = classification.Discipline,

DocumentType = classification.DocumentType,

Retry

N

Continue

Edit

csharp

SheetNumber = classification.SheetNumber,

RevisionNumber = classification.RevisionNumber,

IsCurrentRevision = true, *// Will be updated by revision management*

Status = ProcessingStatus.Classified

};

*// Extract metadata based on file type*

try

{

fileRecord.Status = ProcessingStatus.Processing;

await \_database.UpsertFileRecordAsync(fileRecord);

var extractedMetadata = await \_metadataExtractor.ExtractAsync(filePath, classification);

fileRecord.ExtractedMetadata = JsonSerializer.Serialize(extractedMetadata);

fileRecord.Status = ProcessingStatus.Completed;

\_logger.LogInformation($"Successfully processed: {classification.FileName} " +

$"({classification.Discipline}/{classification.DocumentType}) in {stopwatch.ElapsedMilliseconds}ms");

}

catch (Exception ex)

{

fileRecord.Status = ProcessingStatus.Failed;

fileRecord.ErrorMessage = ex.Message;

\_logger.LogError(ex, $"Failed to extract metadata from: {filePath}");

}

*// Save final record*

await \_database.UpsertFileRecordAsync(fileRecord);

*// Update revision status for related files*

await UpdateRevisionStatusAsync(fileRecord);

}

catch (Exception ex)

{

\_logger.LogError(ex, $"Failed to process file: {filePath}");

}

}

private async Task UpdateRevisionStatusAsync(FileRecord newRecord)

{

if (string.IsNullOrEmpty(newRecord.SheetNumber) || string.IsNullOrEmpty(newRecord.ProjectNumber))

return;

try

{

*// Find other files with same project/discipline/sheet but different revisions*

var relatedFiles = await \_database.GetRelatedFilesAsync(

newRecord.ProjectNumber,

newRecord.Discipline,

newRecord.SheetNumber);

foreach (var relatedFile in relatedFiles)

{

if (relatedFile.Id != newRecord.Id)

{

*// Simple revision comparison (assumes R1, R2, R3... or C01, C02, C03...)*

if (IsNewerRevision(newRecord.RevisionNumber, relatedFile.RevisionNumber))

{

relatedFile.IsCurrentRevision = false;

await \_database.UpsertFileRecordAsync(relatedFile);

}

}

}

}

catch (Exception ex)

{

\_logger.LogWarning(ex, $"Failed to update revision status for {newRecord.FileName}");

}

}

private bool IsNewerRevision(string newRev, string existingRev)

{

*// Simple revision comparison - can be enhanced later*

if (string.IsNullOrEmpty(newRev) || string.IsNullOrEmpty(existingRev))

return false;

*// Handle R1, R2, R3... format*

if (newRev.StartsWith("R") && existingRev.StartsWith("R"))

{

if (int.TryParse(newRev[1..], out var newNum) &&

int.TryParse(existingRev[1..], out var existingNum))

{

return newNum > existingNum;

}

}

*// Handle C01, C02, C03... format*

if (newRev.StartsWith("C") && existingRev.StartsWith("C"))

{

if (int.TryParse(newRev[1..], out var newNum) &&

int.TryParse(existingRev[1..], out var existingNum))

{

return newNum > existingNum;

}

}

*// Handle IFC, IFB special cases*

if (newRev == "IFC" && existingRev.StartsWith("R"))

return true;

return string.Compare(newRev, existingRev, StringComparison.OrdinalIgnoreCase) > 0;

}

private async Task<string> ComputeFileHashAsync(string filePath)

{

try

{

using var sha256 = SHA256.Create();

using var stream = File.OpenRead(filePath);

var hash = await sha256.ComputeHashAsync(stream);

return Convert.ToHexString(hash).ToLower();

}

catch

{

return string.Empty;

}

}

private bool IsFileLocked(string filePath)

{

try

{

using var stream = File.Open(filePath, FileMode.Open, FileAccess.Read, FileShare.None);

return false;

}

catch (IOException)

{

return true;

}

}

}

*// Add missing method to LocalDatabase.cs*

public async Task<List<FileRecord>> GetRelatedFilesAsync(string projectNumber, string discipline, string sheetNumber)

{

var sql = @"

SELECT \* FROM FileRecords

WHERE ProjectNumber = @ProjectNumber

AND Discipline = @Discipline

AND SheetNumber = @SheetNumber

ORDER BY RevisionNumber DESC";

using var command = new SqliteCommand(sql, \_connection);

command.Parameters.AddWithValue("@ProjectNumber", projectNumber);

command.Parameters.AddWithValue("@Discipline", discipline);

command.Parameters.AddWithValue("@SheetNumber", sheetNumber);

var records = new List<FileRecord>();

using var reader = await command.ExecuteReaderAsync();

while (await reader.ReadAsync())

{

records.Add(MapToFileRecord(reader));

}

return records;

}

### **Step 8: Basic Metadata Extractor**

csharp

*// AECFileProcessor.Core/Services/BasicMetadataExtractor.cs*

public class BasicMetadataExtractor : IMetadataExtractor

{

private readonly ILogger<BasicMetadataExtractor> \_logger;

public BasicMetadataExtractor(ILogger<BasicMetadataExtractor> logger)

{

\_logger = logger;

}

public async Task<ExtractedMetadata> ExtractAsync(string filePath, FileClassification classification)

{

var metadata = new ExtractedMetadata();

try

{

metadata = classification.Extension.ToLower() switch

{

".pdf" => await ExtractFromPdfAsync(filePath),

".docx" => await ExtractFromWordAsync(filePath),

".xlsx" => await ExtractFromExcelAsync(filePath),

".dwg" => await ExtractFromDwgAsync(filePath),

".txt" => await ExtractFromTextAsync(filePath),

\_ => await ExtractBasicMetadataAsync(filePath)

};

metadata.ConfidenceScore = 0.7; *// Basic extraction confidence*

}

catch (Exception ex)

{

\_logger.LogError(ex, $"Failed to extract metadata from: {filePath}");

metadata.Properties["error"] = ex.Message;

metadata.ConfidenceScore = 0.1;

}

return metadata;

}

private async Task<ExtractedMetadata> ExtractFromPdfAsync(string filePath)

{

var metadata = new ExtractedMetadata();

try

{

*// Use iTextSharp or PdfPig for PDF text extraction*

using var reader = new PdfReader(filePath);

using var document = new PdfDocument(reader);

var textContent = new StringBuilder();

for (int i = 1; i <= document.GetNumberOfPages(); i++)

{

var page = document.GetPage(i);

var strategy = new SimpleTextExtractionStrategy();

var pageText = PdfTextExtractor.GetTextFromPage(page, strategy);

textContent.AppendLine(pageText);

}

var fullText = textContent.ToString();

metadata.TextContent.Add(fullText);

*// Extract title block information using regex patterns*

metadata.TitleBlock = ExtractTitleBlockFromText(fullText);

*// Extract technical keywords*

metadata.TechnicalKeywords = ExtractTechnicalKeywords(fullText);

*// Extract basic properties*

var info = reader.GetInfo();

metadata.Properties["title"] = info.GetValueOrDefault("Title", "");

metadata.Properties["author"] = info.GetValueOrDefault("Author", "");

metadata.Properties["subject"] = info.GetValueOrDefault("Subject", "");

metadata.Properties["creator"] = info.GetValueOrDefault("Creator", "");

metadata.Properties["producer"] = info.GetValueOrDefault("Producer", "");

document.Close();

}

catch (Exception ex)

{

\_logger.LogWarning(ex, $"Failed to extract PDF content from: {filePath}");

metadata.Properties["pdf\_error"] = ex.Message;

}

return metadata;

}

private async Task<ExtractedMetadata> ExtractFromWordAsync(string filePath)

{

var metadata = new ExtractedMetadata();

try

{

using var document = WordprocessingDocument.Open(filePath, false);

var body = document.MainDocumentPart?.Document.Body;

if (body != null)

{

var fullText = body.InnerText;

metadata.TextContent.Add(fullText);

*// Extract headings and structure*

var headings = ExtractWordHeadings(body);

metadata.Properties["headings"] = string.Join("; ", headings);

*// Extract technical keywords*

metadata.TechnicalKeywords = ExtractTechnicalKeywords(fullText);

*// Document properties*

var props = document.PackageProperties;

metadata.Properties["title"] = props.Title ?? "";

metadata.Properties["author"] = props.Creator ?? "";

metadata.Properties["subject"] = props.Subject ?? "";

metadata.Properties["description"] = props.Description ?? "";

}

}

catch (Exception ex)

{

\_logger.LogWarning(ex, $"Failed to extract Word content from: {filePath}");

metadata.Properties["word\_error"] = ex.Message;

}

return metadata;

}

private async Task<ExtractedMetadata> ExtractFromExcelAsync(string filePath)

{

var metadata = new ExtractedMetadata();

try

{

using var package = new ExcelPackage(new FileInfo(filePath));

var worksheets = package.Workbook.Worksheets;

var allText = new StringBuilder();

var worksheetNames = new List<string>();

foreach (var worksheet in worksheets)

{

worksheetNames.Add(worksheet.Name);

*// Extract text from cells (first 100 rows to avoid performance issues)*

var maxRow = Math.Min(worksheet.Dimension?.End.Row ?? 0, 100);

var maxCol = Math.Min(worksheet.Dimension?.End.Column ?? 0, 20);

for (int row = 1; row <= maxRow; row++)

{

for (int col = 1; col <= maxCol; col++)

{

var cellValue = worksheet.Cells[row, col].Text;

if (!string.IsNullOrWhiteSpace(cellValue))

{

allText.AppendLine(cellValue);

}

}

}

}

metadata.TextContent.Add(allText.ToString());

metadata.Properties["worksheets"] = string.Join("; ", worksheetNames);

metadata.Properties["worksheet\_count"] = worksheets.Count.ToString();

*// Extract technical keywords from cell content*

metadata.TechnicalKeywords = ExtractTechnicalKeywords(allText.ToString());

}

catch (Exception ex)

{

\_logger.LogWarning(ex, $"Failed to extract Excel content from: {filePath}");

metadata.Properties["excel\_error"] = ex.Message;

}

return metadata;

}

private async Task<ExtractedMetadata> ExtractFromDwgAsync(string filePath)

{

var metadata = new ExtractedMetadata();

*// Note: This is a placeholder. Full CAD extraction requires AutoCAD .NET API*

*// For now, we'll extract basic file information*

try

{

var fileInfo = new FileInfo(filePath);

metadata.Properties["file\_size"] = fileInfo.Length.ToString();

metadata.Properties["last\_modified"] = fileInfo.LastWriteTime.ToString("O");

*// Placeholder for future CAD-specific extraction*

metadata.Properties["cad\_type"] = "AutoCAD Drawing";

metadata.Properties["extraction\_method"] = "basic\_info\_only";

*// Add note about limited extraction*

metadata.TextContent.Add("CAD file detected - advanced extraction requires AutoCAD .NET API");

}

catch (Exception ex)

{

\_logger.LogWarning(ex, $"Failed to extract CAD content from: {filePath}");

metadata.Properties["cad\_error"] = ex.Message;

}

return metadata;

}

private async Task<ExtractedMetadata> ExtractFromTextAsync(string filePath)

{

var metadata = new ExtractedMetadata();

try

{

var content = await File.ReadAllTextAsync(filePath);

metadata.TextContent.Add(content);

metadata.TechnicalKeywords = ExtractTechnicalKeywords(content);

metadata.Properties["line\_count"] = content.Split('\n').Length.ToString();

metadata.Properties["character\_count"] = content.Length.ToString();

}

catch (Exception ex)

{

\_logger.LogWarning(ex, $"Failed to extract text content from: {filePath}");

metadata.Properties["text\_error"] = ex.Message;

}

return metadata;

}

private async Task<ExtractedMetadata> ExtractBasicMetadataAsync(string filePath)

{

var metadata = new ExtractedMetadata();

var fileInfo = new FileInfo(filePath);

metadata.Properties["file\_size"] = fileInfo.Length.ToString();

metadata.Properties["created\_date"] = fileInfo.CreationTime.ToString("O");

metadata.Properties["modified\_date"] = fileInfo.LastWriteTime.ToString("O");

metadata.Properties["extension"] = fileInfo.Extension;

metadata.Properties["extraction\_method"] = "basic\_file\_info";

return metadata;

}

private TitleBlockData ExtractTitleBlockFromText(string text)

{

var titleBlock = new TitleBlockData();

*// Common title block patterns for AEC drawings*

var patterns = new Dictionary<string, Regex>

{

["project\_name"] = new Regex(@"PROJECT[:\s]+([^\r\n]+)", RegexOptions.IgnoreCase),

["sheet\_title"] = new Regex(@"(?:TITLE|DRAWING)[:\s]+([^\r\n]+)", RegexOptions.IgnoreCase),

["sheet\_number"] = new Regex(@"(?:SHEET|DWG)[:\s]+([A-Z]-?\d+(?:\.\d+)?)", RegexOptions.IgnoreCase),

["revision"] = new Regex(@"REV(?:ISION)?[:\s]+([A-Z]?\d+|IFC|IFB|AB)", RegexOptions.IgnoreCase),

["drawn\_by"] = new Regex(@"DRAWN[:\s]+([^\r\n]+)", RegexOptions.IgnoreCase),

["checked\_by"] = new Regex(@"(?:CHECKED|CHK)[:\s]+([^\r\n]+)", RegexOptions.IgnoreCase),

["date"] = new Regex(@"DATE[:\s]+(\d{1,2}[\/\-]\d{1,2}[\/\-]\d{2,4})", RegexOptions.IgnoreCase),

["scale"] = new Regex(@"SCALE[:\s]+([^\r\n]+)", RegexOptions.IgnoreCase)

};

foreach (var pattern in patterns)

{

var match = pattern.Value.Match(text);

if (match.Success)

{

var value = match.Groups[1].Value.Trim();

switch (pattern.Key)

{

case "project\_name": titleBlock.ProjectName = value; break;

case "sheet\_title": titleBlock.SheetTitle = value; break;

case "sheet\_number": titleBlock.SheetNumber = value; break;

case "revision": titleBlock.RevisionNumber = value; break;

case "drawn\_by": titleBlock.DrawnBy = value; break;

case "checked\_by": titleBlock.CheckedBy = value; break;

case "date": titleBlock.Date = value; break;

case "scale": titleBlock.Scale = value; break;

}

}

}

return titleBlock;

}

private List<string> ExtractTechnicalKeywords(string text)

{

var keywords = new HashSet<string>(StringComparer.OrdinalIgnoreCase);

*// AEC-specific technical terms*

var technicalPatterns = new[]

{

*// Materials*

@"\b(?:concrete|steel|wood|aluminum|masonry|drywall|insulation)\b",

*// Equipment*

@"\b(?:pump|fan|boiler|chiller|transformer|panel|valve|damper)\b",

*// Dimensions and measurements*

@"\d+(?:\.\d+)?\s\*(?:ft|in|mm|cm|m)(?:\s|$)",

@"\d+(?:\.\d+)?\s\*(?:sf|sq\.?\s\*ft|square\s+feet)",

*// Standards and codes*

@"\b(?:ASTM|ACI|AISC|NEC|IBC|ASHRAE|NFPA)\s\*[A-Z]?\d+",

*// Room types and spaces*

@"\b(?:office|conference|storage|mechanical|electrical|restroom|lobby|corridor)\b"

};

foreach (var pattern in technicalPatterns)

{

var matches = Regex.Matches(text, pattern, RegexOptions.IgnoreCase);

foreach (Match match in matches)

{

var keyword = match.Value.Trim().ToLower();

if (keyword.Length > 2) *// Skip very short matches*

{

keywords.Add(keyword);

}

}

}

return keywords.Take(50).ToList(); *// Limit to prevent huge keyword lists*

}

private List<string> ExtractWordHeadings(Body body)

{

var headings = new List<string>();

foreach (var paragraph in body.Elements<Paragraph>())

{

var paragraphProperties = paragraph.ParagraphProperties;

var paragraphStyleId = paragraphProperties?.ParagraphStyleId?.Val?.Value;

*// Look for heading styles*

if (paragraphStyleId != null && paragraphStyleId.StartsWith("Heading"))

{

var text = paragraph.InnerText.Trim();

if (!string.IsNullOrEmpty(text))

{

headings.Add(text);

}

}

}

return headings;

}

}

### **Step 9: Interfaces and Additional Models**

csharp

*// AECFileProcessor.Core/Interfaces/IFileProcessor.cs*

public interface IFileProcessor

{

Task ProcessFileAsync(string filePath);

}

*// AECFileProcessor.Core/Interfaces/IFileClassifier.cs*

public interface IFileClassifier

{

Task<FileClassification> ClassifyAsync(string filePath);

}

*// AECFileProcessor.Core/Interfaces/IMetadataExtractor.cs*

public interface IMetadataExtractor

{

Task<ExtractedMetadata> ExtractAsync(string filePath, FileClassification classification);

}

### **Step 10: First Working Version - Testing**

Create a test script to verify everything works:

csharp

*// AECFileProcessor.Tests/IntegrationTests.cs*

public class LocalProcessingIntegrationTests

{

private readonly string \_testDataPath;

private readonly string \_testDbPath;

public LocalProcessingIntegrationTests()

{

\_testDataPath = Path.Combine(Path.GetTempPath(), "AECTestData");

\_testDbPath = Path.Combine(Path.GetTempPath(), "test\_aec.db");

Directory.CreateDirectory(\_testDataPath);

}

[Fact]

public async Task Should\_Process\_Standard\_Named\_File()

{

*// Arrange*

var testFilePath = Path.Combine(\_testDataPath, "PROJ123\_CD\_A\_DWG\_101\_R2\_2024-03-15.pdf");

await File.WriteAllTextAsync(testFilePath, "Sample PDF content with PROJECT: Test Building\nSHEET: A-101\nREVISION: R2");

var services = CreateTestServices();

var processor = services.GetRequiredService<IFileProcessor>();

*// Act*

await processor.ProcessFileAsync(testFilePath);

*// Assert*

var database = services.GetRequiredService<LocalDatabase>();

var files = await database.GetFilesByProjectAsync("PROJ123");

Assert.Single(files);

var file = files.First();

Assert.Equal("PROJ123", file.ProjectNumber);

Assert.Equal("Construction Documents", file.Phase);

Assert.Equal("Architectural", file.Discipline);

Assert.Equal("Drawing", file.DocumentType);

Assert.Equal("101", file.SheetNumber);

Assert.Equal("R2", file.RevisionNumber);

Assert.Equal(ProcessingStatus.Completed, file.Status);

}

[Fact]

public async Task Should\_Handle\_Non\_Standard\_File\_Names()

{

*// Arrange*

var testFilePath = Path.Combine(\_testDataPath, "Floor Plan - Level 1.pdf");

await File.WriteAllTextAsync(testFilePath, "Architectural floor plan content");

var services = CreateTestServices();

var processor = services.GetRequiredService<IFileProcessor>();

*// Act*

await processor.ProcessFileAsync(testFilePath);

*// Assert*

var database = services.GetRequiredService<LocalDatabase>();

var allProjects = await database.GetProjectNumbersAsync();

Assert.NotEmpty(allProjects);

*// Should still process the file even without standard naming*

}

[Fact]

public async Task Should\_Update\_Revision\_Status()

{

*// Arrange*

var services = CreateTestServices();

var processor = services.GetRequiredService<IFileProcessor>();

*// Create older revision*

var oldFile = Path.Combine(\_testDataPath, "PROJ456\_CD\_S\_DWG\_201\_R1\_2024-02-15.pdf");

await File.WriteAllTextAsync(oldFile, "Structural drawing R1");

await processor.ProcessFileAsync(oldFile);

*// Create newer revision*

var newFile = Path.Combine(\_testDataPath, "PROJ456\_CD\_S\_DWG\_201\_R2\_2024-03-15.pdf");

await File.WriteAllTextAsync(newFile, "Structural drawing R2");

*// Act*

await processor.ProcessFileAsync(newFile);

*// Assert*

var database = services.GetRequiredService<LocalDatabase>();

var files = await database.GetFilesByProjectAsync("PROJ456");

Assert.Equal(2, files.Count);

var r1File = files.First(f => f.RevisionNumber == "R1");

var r2File = files.First(f => f.RevisionNumber == "R2");

Assert.False(r1File.IsCurrentRevision);

Assert.True(r2File.IsCurrentRevision);

}

private IServiceProvider CreateTestServices()

{

return new ServiceCollection()

.AddLogging(builder => builder.AddConsole().SetMinimumLevel(LogLevel.Debug))

.AddSingleton(provider => new LocalDatabase(\_testDbPath, provider.GetRequiredService<ILogger<LocalDatabase>>()))

.AddSingleton<IFileClassifier, FileClassifier>()

.AddSingleton<IMetadataExtractor, BasicMetadataExtractor>()

.AddSingleton<IFileProcessor, FileProcessor>()

.BuildServiceProvider();

}

public void Dispose()

{

if (Directory.Exists(\_testDataPath))

Directory.Delete(\_testDataPath, true);

if (File.Exists(\_testDbPath))

File.Delete(\_testDbPath);

}

}

## **Usage Instructions for First Version**

### **1. Build and Run**

bash

*# Build the solution*

dotnet build

*# Watch a directory (keeps running)*

dotnet run --project AECFileProcessor.CLI -- watch --path "C:\Projects\ProjectABC"

*# Process files once*

dotnet run --project AECFileProcessor.CLI -- process --path "C:\Projects\ProjectABC"

*# Query processed files*

dotnet run --project AECFileProcessor.CLI -- query --project "PROJ123"

### **2. Expected Output**

[2024-03-15 10:30:15] Starting file watcher for: C:\Projects\ProjectABC

[2024-03-15 10:30:15] Database: aec\_files.db

[2024-03-15 10:30:15] File watcher started. Press Ctrl+C to stop...

[2024-03-15 10:30:16] Found 1247 existing files to process

[2024-03-15 10:30:17] Successfully processed: PROJ123\_CD\_A\_DWG\_101\_R2\_2024-03-15.pdf (Architectural/Drawing) in 234ms

[2024-03-15 10:30:18] Successfully processed: PROJ123\_CD\_S\_DWG\_S201\_R1\_2024-03-12.dwg (Structural/Drawing) in 156ms

[2024-03-15 10:30:19] Queued file: C:\Projects\ProjectABC\02\_DRAWINGS\Current\_Issue\Architectural\A-102.pdf

## **Phase 1 Scaling Steps (Weeks 3-4)**

### **Step 11: Add Simple Web API**

csharp

*// AECFileProcessor.API/Controllers/FilesController.cs*

[ApiController]

[Route("api/[controller]")]

public class FilesController : ControllerBase

{

private readonly LocalDatabase \_database;

private readonly IFileProcessor \_processor;

private readonly ILogger<FilesController> \_logger;

public FilesController(LocalDatabase database, IFileProcessor processor, ILogger<FilesController> logger)

{

\_database = database;

\_processor = processor;

\_logger = logger;

}

[HttpGet("projects")]

public async Task<ActionResult<List<string>>> GetProjects()

{

var projects = await \_database.GetProjectNumbersAsync();

return Ok(projects);

}

[HttpGet("projects/{projectNumber}/files")]

public async Task<ActionResult<List<FileRecord>>> GetProjectFiles(string projectNumber)

{

var files = await \_database.GetFilesByProjectAsync(projectNumber);

return Ok(files);

}

[HttpPost("process")]

public async Task<ActionResult> ProcessFile([FromBody] ProcessFileRequest request)

{

try

{

if (!System.IO.File.Exists(request.FilePath))

{

return NotFound($"File not found: {request.FilePath}");

}

await \_processor.ProcessFileAsync(request.FilePath);

return Ok(new { message = "File processing initiated", filePath = request.FilePath });

}

catch (Exception ex)

{

\_logger.LogError(ex, $"Failed to process file: {request.FilePath}");

return StatusCode(500, new { error = ex.Message });

}

}

[HttpGet("health")]

public ActionResult GetHealth()

{

return Ok(new

{

status = "healthy",

timestamp = DateTime.UtcNow,

version = "1.0.0-local"

});

}

}

public class ProcessFileRequest

{

public string FilePath { get; set; } = string.Empty;

}

### **Step 12: Add Background Service**

csharp

*// AECFileProcessor.API/Services/FileWatcherBackgroundService.cs*

public class FileWatcherBackgroundService : BackgroundService

{

private readonly IConfiguration \_configuration;

private readonly IServiceProvider \_serviceProvider;

private readonly ILogger<FileWatcherBackgroundService> \_logger;

private LocalFileWatcher? \_watcher;

public FileWatcherBackgroundService(

IConfiguration configuration,

IServiceProvider serviceProvider,

ILogger<FileWatcherBackgroundService> logger)

{

\_configuration = configuration;

\_serviceProvider = serviceProvider;

\_logger = logger;

}

protected override async Task ExecuteAsync(CancellationToken stoppingToken)

{

var watchPath = \_configuration["FileWatcher:WatchPath"];

if (string.IsNullOrEmpty(watchPath))

{

\_logger.LogWarning("No watch path configured. File watching disabled.");

return;

}

if (!Directory.Exists(watchPath))

{

\_logger.LogError($"Watch path does not exist: {watchPath}");

return;

}

using var scope = \_serviceProvider.CreateScope();

var processor = scope.ServiceProvider.GetRequiredService<IFileProcessor>();

\_watcher = new LocalFileWatcher(watchPath, processor,

scope.ServiceProvider.GetRequiredService<ILogger<LocalFileWatcher>>());

\_watcher.StartWatching();

\_logger.LogInformation($"File watcher started for: {watchPath}");

try

{

await Task.Delay(-1, stoppingToken);

}

catch (OperationCanceledException)

{

\_logger.LogInformation("File watcher stopping...");

}

}

public override void Dispose()

{

\_watcher?.Dispose();

base.Dispose();

}

}

### **Step 13: Configuration**

json

*// AECFileProcessor.API/appsettings.json*

{

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}

},

"Database": {

"ConnectionString": "Data Source=aec\_files.db"

},

"FileWatcher": {

"WatchPath": "C:\\Projects",

"ProcessingConcurrency": 4,

"BatchSize": 50,

"BatchDelaySeconds": 5

},

"AllowedHosts": "\*"

}

csharp

*// AECFileProcessor.API/Program.cs*

var builder = WebApplication.CreateBuilder(args);

*// Add services*

builder.Services.AddControllers();

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

*// Add custom services*

var dbPath = builder.Configuration.GetConnectionString("Default") ?? "aec\_files.db";

builder.Services.AddSingleton(provider =>

new LocalDatabase(dbPath, provider.GetRequiredService<ILogger<LocalDatabase>>()));

builder.Services.AddSingleton<IFileClassifier, FileClassifier>();

builder.Services.AddSingleton<IMetadataExtractor, BasicMetadataExtractor>();

builder.Services.AddSingleton<IFileProcessor, FileProcessor>();

*// Add background service*

builder.Services.AddHostedService<FileWatcherBackgroundService>();

var app = builder.Build();

*// Configure pipeline*

if (app.Environment.

Retry

N

Continue

Edit

csharp

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

## **Phase 2: Enhanced Local System (Weeks 5-8)**

### **Step 14: Add Simple Event System (Local Message Queue)**

csharp

*// AECFileProcessor.Core/Events/IEventBus.cs*

public interface IEventBus

{

Task PublishAsync<T>(T eventData) where T : class;

void Subscribe<T>(Func<T, Task> handler) where T : class;

}

*// AECFileProcessor.Core/Events/LocalEventBus.cs*

public class LocalEventBus : IEventBus

{

private readonly ConcurrentDictionary<Type, List<Func<object, Task>>> \_handlers = new();

private readonly Channel<(Type EventType, object EventData)> \_eventChannel;

private readonly ILogger<LocalEventBus> \_logger;

private readonly Task \_processingTask;

private readonly CancellationTokenSource \_cancellationTokenSource;

public LocalEventBus(ILogger<LocalEventBus> logger)

{

\_logger = logger;

\_cancellationTokenSource = new CancellationTokenSource();

var options = new BoundedChannelOptions(1000)

{

FullMode = BoundedChannelFullMode.Wait,

SingleReader = true,

SingleWriter = false

};

\_eventChannel = Channel.CreateBounded<(Type, object)>(options);

\_processingTask = ProcessEventsAsync(\_cancellationTokenSource.Token);

}

public async Task PublishAsync<T>(T eventData) where T : class

{

if (eventData == null) return;

await \_eventChannel.Writer.WriteAsync((typeof(T), eventData));

\_logger.LogDebug($"Published event: {typeof(T).Name}");

}

public void Subscribe<T>(Func<T, Task> handler) where T : class

{

var eventType = typeof(T);

var wrappedHandler = new Func<object, Task>(async obj =>

{

if (obj is T typedEvent)

{

await handler(typedEvent);

}

});

\_handlers.AddOrUpdate(eventType,

new List<Func<object, Task>> { wrappedHandler },

(key, existing) => { existing.Add(wrappedHandler); return existing; });

\_logger.LogInformation($"Subscribed to event: {eventType.Name}");

}

private async Task ProcessEventsAsync(CancellationToken cancellationToken)

{

await foreach (var (eventType, eventData) in \_eventChannel.Reader.ReadAllAsync(cancellationToken))

{

try

{

if (\_handlers.TryGetValue(eventType, out var handlers))

{

var tasks = handlers.Select(handler => handler(eventData));

await Task.WhenAll(tasks);

\_logger.LogDebug($"Processed event {eventType.Name} with {handlers.Count} handlers");

}

}

catch (Exception ex)

{

\_logger.LogError(ex, $"Error processing event {eventType.Name}");

}

}

}

public void Dispose()

{

\_eventChannel.Writer.Complete();

\_cancellationTokenSource.Cancel();

\_processingTask.Wait(TimeSpan.FromSeconds(5));

\_cancellationTokenSource.Dispose();

}

}

*// Event Models*

public class FileDiscoveredEvent

{

public string FilePath { get; set; } = string.Empty;

public DateTime DiscoveredAt { get; set; } = DateTime.UtcNow;

public Guid ProcessingId { get; set; } = Guid.NewGuid();

}

public class FileClassifiedEvent

{

public Guid ProcessingId { get; set; }

public string FilePath { get; set; } = string.Empty;

public FileClassification Classification { get; set; } = new();

public DateTime ClassifiedAt { get; set; } = DateTime.UtcNow;

}

public class FileProcessedEvent

{

public Guid ProcessingId { get; set; }

public string FilePath { get; set; } = string.Empty;

public string ProjectNumber { get; set; } = string.Empty;

public string Discipline { get; set; } = string.Empty;

public ProcessingStatus Status { get; set; }

public TimeSpan ProcessingDuration { get; set; }

public DateTime CompletedAt { get; set; } = DateTime.UtcNow;

}

### **Step 15: Event-Driven File Processor**

csharp

*// AECFileProcessor.Core/Services/EventDrivenFileProcessor.cs*

public class EventDrivenFileProcessor : IFileProcessor

{

private readonly LocalDatabase \_database;

private readonly IFileClassifier \_classifier;

private readonly IMetadataExtractor \_metadataExtractor;

private readonly IEventBus \_eventBus;

private readonly ILogger<EventDrivenFileProcessor> \_logger;

public EventDrivenFileProcessor(

LocalDatabase database,

IFileClassifier classifier,

IMetadataExtractor metadataExtractor,

IEventBus eventBus,

ILogger<EventDrivenFileProcessor> logger)

{

\_database = database;

\_classifier = classifier;

\_metadataExtractor = metadataExtractor;

\_eventBus = eventBus;

\_logger = logger;

*// Subscribe to events*

\_eventBus.Subscribe<FileDiscoveredEvent>(HandleFileDiscoveredAsync);

\_eventBus.Subscribe<FileClassifiedEvent>(HandleFileClassifiedAsync);

}

public async Task ProcessFileAsync(string filePath)

{

*// Publish file discovered event instead of processing directly*

await \_eventBus.PublishAsync(new FileDiscoveredEvent

{

FilePath = filePath,

ProcessingId = Guid.NewGuid()

});

}

private async Task HandleFileDiscoveredAsync(FileDiscoveredEvent @event)

{

try

{

\_logger.LogDebug($"Handling file discovered: {@event.FilePath}");

if (!File.Exists(@event.FilePath) || IsFileLocked(@event.FilePath))

{

\_logger.LogWarning($"File not accessible: {@event.FilePath}");

return;

}

*// Classify the file*

var classification = await \_classifier.ClassifyAsync(@event.FilePath);

*// Publish classification event*

await \_eventBus.PublishAsync(new FileClassifiedEvent

{

ProcessingId = @event.ProcessingId,

FilePath = @event.FilePath,

Classification = classification

});

}

catch (Exception ex)

{

\_logger.LogError(ex, $"Failed to handle file discovered: {@event.FilePath}");

}

}

private async Task HandleFileClassifiedAsync(FileClassifiedEvent @event)

{

var stopwatch = Stopwatch.StartNew();

try

{

\_logger.LogDebug($"Handling file classified: {@event.FilePath}");

var classification = @event.Classification;

*// Create file record*

var fileRecord = CreateFileRecord(@event.ProcessingId, classification);

fileRecord.Status = ProcessingStatus.Processing;

await \_database.UpsertFileRecordAsync(fileRecord);

*// Extract metadata*

try

{

var extractedMetadata = await \_metadataExtractor.ExtractAsync(@event.FilePath, classification);

fileRecord.ExtractedMetadata = JsonSerializer.Serialize(extractedMetadata);

fileRecord.Status = ProcessingStatus.Completed;

\_logger.LogInformation($"Successfully processed: {classification.FileName} " +

$"({classification.Discipline}/{classification.DocumentType}) in {stopwatch.ElapsedMilliseconds}ms");

}

catch (Exception ex)

{

fileRecord.Status = ProcessingStatus.Failed;

fileRecord.ErrorMessage = ex.Message;

\_logger.LogError(ex, $"Failed to extract metadata from: {@event.FilePath}");

}

*// Save final record*

await \_database.UpsertFileRecordAsync(fileRecord);

*// Update revision status*

await UpdateRevisionStatusAsync(fileRecord);

*// Publish completion event*

await \_eventBus.PublishAsync(new FileProcessedEvent

{

ProcessingId = @event.ProcessingId,

FilePath = @event.FilePath,

ProjectNumber = fileRecord.ProjectNumber,

Discipline = fileRecord.Discipline,

Status = fileRecord.Status,

ProcessingDuration = stopwatch.Elapsed

});

}

catch (Exception ex)

{

\_logger.LogError(ex, $"Failed to handle file classified: {@event.FilePath}");

}

}

private FileRecord CreateFileRecord(Guid processingId, FileClassification classification)

{

return new FileRecord

{

Id = processingId,

FilePath = classification.FilePath,

FileName = classification.FileName,

FileSize = classification.FileSize,

FileHash = ComputeFileHash(classification.FilePath),

CreatedDate = classification.CreatedDate,

ModifiedDate = classification.ModifiedDate,

ProcessedDate = DateTime.UtcNow,

ProjectNumber = classification.ProjectNumber,

Phase = classification.Phase,

Discipline = classification.Discipline,

DocumentType = classification.DocumentType,

SheetNumber = classification.SheetNumber,

RevisionNumber = classification.RevisionNumber,

IsCurrentRevision = true

};

}

*// ... (other helper methods remain the same)*

}

### **Step 16: Real-time Dashboard with SignalR**

csharp

*// AECFileProcessor.API/Hubs/ProcessingHub.cs*

public class ProcessingHub : Hub

{

public async Task JoinProjectGroup(string projectNumber)

{

await Groups.AddToGroupAsync(Context.ConnectionId, $"project\_{projectNumber}");

}

public async Task LeaveProjectGroup(string projectNumber)

{

await Groups.RemoveFromGroupAsync(Context.ConnectionId, $"project\_{projectNumber}");

}

}

*// AECFileProcessor.API/Services/SignalRNotificationService.cs*

public class SignalRNotificationService : INotificationService

{

private readonly IHubContext<ProcessingHub> \_hubContext;

private readonly ILogger<SignalRNotificationService> \_logger;

public SignalRNotificationService(IHubContext<ProcessingHub> hubContext, ILogger<SignalRNotificationService> logger)

{

\_hubContext = hubContext;

\_logger = logger;

}

public async Task NotifyFileProcessedAsync(FileProcessedEvent processedEvent)

{

try

{

await \_hubContext.Clients.Group($"project\_{processedEvent.ProjectNumber}")

.SendAsync("FileProcessed", new

{

fileName = Path.GetFileName(processedEvent.FilePath),

discipline = processedEvent.Discipline,

status = processedEvent.Status.ToString(),

processingTime = processedEvent.ProcessingDuration.TotalMilliseconds,

timestamp = processedEvent.CompletedAt

});

}

catch (Exception ex)

{

\_logger.LogError(ex, "Failed to send SignalR notification");

}

}

}

*// Subscribe to events in Program.cs*

builder.Services.AddSignalR();

builder.Services.AddSingleton<INotificationService, SignalRNotificationService>();

*// In the app configuration*

app.MapHub<ProcessingHub>("/hubs/processing");

*// Subscribe to events after building the app*

var eventBus = app.Services.GetRequiredService<IEventBus>();

var notificationService = app.Services.GetRequiredService<INotificationService>();

eventBus.Subscribe<FileProcessedEvent>(async (processedEvent) =>

{

await notificationService.NotifyFileProcessedAsync(processedEvent);

});

### **Step 17: Simple HTML Dashboard**

html

*<!-- AECFileProcessor.API/wwwroot/dashboard.html -->*

<!DOCTYPE html>

<html>

<head>

<title>AEC File Processing Dashboard</title>

<script src="https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/7.0.0/signalr.min.js"></script>

<style>

body { font-family: Arial, sans-serif; margin: 20px; }

.stats { display: flex; gap: 20px; margin-bottom: 30px; }

.stat-card {

background: #f5f5f5;

padding: 20px;

border-radius: 8px;

min-width: 150px;

text-align: center;

}

.stat-value { font-size: 2em; font-weight: bold; color: #2196F3; }

.stat-label { color: #666; margin-top: 5px; }

.file-list { background: white; border: 1px solid #ddd; border-radius: 8px; }

.file-item {

padding: 15px;

border-bottom: 1px solid #eee;

display: flex;

justify-content: space-between;

align-items: center;

}

.file-item:last-child { border-bottom: none; }

.status-completed { color: green; }

.status-failed { color: red; }

.status-processing { color: orange; }

.recent-activity { margin-top: 30px; }

.activity-item {

padding: 10px;

background: #f9f9f9;

margin: 5px 0;

border-radius: 4px;

border-left: 4px solid #2196F3;

}

</style>

</head>

<body>

<h1>AEC File Processing Dashboard</h1>

<div class="stats">

<div class="stat-card">

<div class="stat-value" id="totalFiles">-</div>

<div class="stat-label">Total Files</div>

</div>

<div class="stat-card">

<div class="stat-value" id="processedToday">-</div>

<div class="stat-label">Processed Today</div>

</div>

<div class="stat-card">

<div class="stat-value" id="activeProjects">-</div>

<div class="stat-label">Active Projects</div>

</div>

<div class="stat-card">

<div class="stat-value" id="errorCount">-</div>

<div class="stat-label">Errors</div>

</div>

</div>

<h2>Projects</h2>

<select id="projectSelect" onchange="loadProjectFiles()">

<option value="">Select a project...</option>

</select>

<div id="projectFiles" class="file-list" style="margin-top: 20px; display: none;">

<h3>Project Files</h3>

<div id="filesList"></div>

</div>

<div class="recent-activity">

<h2>Recent Activity</h2>

<div id="activityList"></div>

</div>

<script>

*// SignalR connection*

const connection = new signalR.HubConnectionBuilder()

.withUrl("/hubs/processing")

.build();

let currentProject = '';

*// Start connection*

connection.start().then(function () {

console.log("Connected to processing hub");

loadProjects();

loadDashboardStats();

}).catch(function (err) {

console.error("SignalR connection error:", err);

});

*// Listen for file processing updates*

connection.on("FileProcessed", function (data) {

addActivityItem(`File processed: ${data.fileName} (${data.discipline}) - ${data.status} in ${Math.round(data.processingTime)}ms`);

if (currentProject && data.projectNumber === currentProject) {

loadProjectFiles();

}

loadDashboardStats();

});

async function loadProjects() {

try {

const response = await fetch('/api/files/projects');

const projects = await response.json();

const select = document.getElementById('projectSelect');

select.innerHTML = '<option value="">Select a project...</option>';

projects.forEach(project => {

const option = document.createElement('option');

option.value = project;

option.textContent = project;

select.appendChild(option);

});

document.getElementById('activeProjects').textContent = projects.length;

} catch (error) {

console.error('Failed to load projects:', error);

}

}

async function loadProjectFiles() {

const projectNumber = document.getElementById('projectSelect').value;

if (!projectNumber) {

document.getElementById('projectFiles').style.display = 'none';

return;

}

currentProject = projectNumber;

*// Join SignalR group for this project*

await connection.invoke("JoinProjectGroup", projectNumber);

try {

const response = await fetch(`/api/files/projects/${projectNumber}/files`);

const files = await response.json();

const filesList = document.getElementById('filesList');

filesList.innerHTML = '';

files.forEach(file => {

const fileItem = document.createElement('div');

fileItem.className = 'file-item';

const statusClass = `status-${file.status.toLowerCase()}`;

fileItem.innerHTML = `

<div>

<strong>${file.fileName}</strong>

<br>

<small>${file.discipline} | ${file.documentType} | ${file.phase}</small>

</div>

<div>

<span class="${statusClass}">${file.status}</span>

<br>

<small>${new Date(file.modifiedDate).toLocaleString()}</small>

</div>

`;

filesList.appendChild(fileItem);

});

document.getElementById('projectFiles').style.display = 'block';

} catch (error) {

console.error('Failed to load project files:', error);

}

}

async function loadDashboardStats() {

try {

*// This would need to be implemented in the API*

const response = await fetch('/api/files/stats');

const stats = await response.json();

document.getElementById('totalFiles').textContent = stats.totalFiles || 0;

document.getElementById('processedToday').textContent = stats.processedToday || 0;

document.getElementById('errorCount').textContent = stats.errorCount || 0;

} catch (error) {

console.error('Failed to load dashboard stats:', error);

*// Set default values if API not implemented yet*

document.getElementById('totalFiles').textContent = '-';

document.getElementById('processedToday').textContent = '-';

document.getElementById('errorCount').textContent = '-';

}

}

function addActivityItem(message) {

const activityList = document.getElementById('activityList');

const activityItem = document.createElement('div');

activityItem.className = 'activity-item';

activityItem.innerHTML = `

<div>${message}</div>

<small>${new Date().toLocaleTimeString()}</small>

`;

activityList.insertBefore(activityItem, activityList.firstChild);

*// Keep only the last 20 items*

while (activityList.children.length > 20) {

activityList.removeChild(activityList.lastChild);

}

}

*// Refresh data every 30 seconds*

setInterval(() => {

loadDashboardStats();

}, 30000);

</script>

</body>

</html>

## **Phase 3: Prepare for Scaling (Weeks 9-12)**

### **Step 18: Abstract Storage Layer**

csharp

*// AECFileProcessor.Core/Interfaces/IFileRepository.cs*

public interface IFileRepository

{

Task<string> UpsertFileRecordAsync(FileRecord record);

Task<List<FileRecord>> GetFilesByProjectAsync(string projectNumber);

Task<List<string>> GetProjectNumbersAsync();

Task<List<FileRecord>> GetRelatedFilesAsync(string projectNumber, string discipline, string sheetNumber);

Task<DashboardStats> GetDashboardStatsAsync();

}

*// AECFileProcessor.Core/Repository/SqliteFileRepository.cs*

public class SqliteFileRepository : IFileRepository

{

private readonly LocalDatabase \_database;

public SqliteFileRepository(LocalDatabase database)

{

\_database = database;

}

*// Implement all methods by delegating to LocalDatabase*

public async Task<string> UpsertFileRecordAsync(FileRecord record)

{

return await \_database.UpsertFileRecordAsync(record);

}

*// ... other methods*

}

*// AECFileProcessor.Core/Repository/PostgreSqlFileRepository.cs (for future scaling)*

public class PostgreSqlFileRepository : IFileRepository

{

private readonly string \_connectionString;

public PostgreSqlFileRepository(string connectionString)

{

\_connectionString = connectionString;

}

public async Task<string> UpsertFileRecordAsync(FileRecord record)

{

*// PostgreSQL implementation for when we scale*

using var connection = new NpgsqlConnection(\_connectionString);

*// ... implementation*

throw new NotImplementedException("PostgreSQL repository for future scaling");

}

*// ... other methods*

}

### **Step 19: Configuration-Based Service Selection**

csharp

*// AECFileProcessor.Core/Configuration/ProcessingConfiguration.cs*

public class ProcessingConfiguration

{

public string StorageType { get; set; } = "SQLite"; *// SQLite, PostgreSQL, CosmosDB*

public string EventBusType { get; set; } = "InMemory"; *// InMemory, ServiceBus, Kafka*

public string CacheType { get; set; } = "InMemory"; *// InMemory, Redis, MemoryCache*

public int MaxConcurrentFiles { get; set; } = 4;

public int BatchSize { get; set; } = 50;

public int BatchDelaySeconds { get; set; } = 5;

public bool EnableAdvancedOCR { get; set; } = false;

public bool EnableSemanticAnalysis { get; set; } = false;

public bool EnableCADProcessing { get; set; } = false;

public Dictionary<string, string> ConnectionStrings { get; set; } = new();

}

*// AECFileProcessor.API/Extensions/ServiceCollectionExtensions.cs*

public static class ServiceCollectionExtensions

{

public static IServiceCollection AddAECFileProcessing(

this IServiceCollection services,

ProcessingConfiguration config)

{

*// Add storage*

services = config.StorageType switch

{

"SQLite" => services.AddSQLiteStorage(config.ConnectionStrings["SQLite"] ?? "aec\_files.db"),

"PostgreSQL" => services.AddPostgreSQLStorage(config.ConnectionStrings["PostgreSQL"]),

\_ => throw new NotSupportedException($"Storage type {config.StorageType} not supported")

};

*// Add event bus*

services = config.EventBusType switch

{

"InMemory" => services.AddSingleton<IEventBus, LocalEventBus>(),

"ServiceBus" => services.AddServiceBusEventBus(config.ConnectionStrings["ServiceBus"]),

"Kafka" => services.AddKafkaEventBus(config.ConnectionStrings["Kafka"]),

\_ => throw new NotSupportedException($"Event bus type {config.EventBusType} not supported")

};

*// Add core services*

services.AddSingleton<IFileClassifier, FileClassifier>();

services.AddSingleton<IFileProcessor, EventDrivenFileProcessor>();

*// Add metadata extractors based on configuration*

if (config.EnableAdvancedOCR)

{

services.AddSingleton<IOCRService, TesseractOCRService>();

}

if (config.EnableCADProcessing)

{

services.AddSingleton<ICADProcessor, AutoCADProcessor>();

}

services.AddSingleton<IMetadataExtractor>(provider =>

{

if (config.EnableAdvancedOCR || config.EnableCADProcessing)

{

return new AdvancedMetadataExtractor(

provider.GetService<IOCRService>(),

provider.GetService<ICADProcessor>(),

provider.GetRequiredService<ILogger<AdvancedMetadataExtractor>>()

);

}

return new BasicMetadataExtractor(provider.GetRequiredService<ILogger<BasicMetadataExtractor>>());

});

return services;

}

private static IServiceCollection AddSQLiteStorage(this IServiceCollection services, string connectionString)

{

services.AddSingleton(provider =>

new LocalDatabase(connectionString, provider.GetRequiredService<ILogger<LocalDatabase>>()));

services.AddSingleton<IFileRepository, SqliteFileRepository>();

return services;

}

private static IServiceCollection AddPostgreSQLStorage(this IServiceCollection services, string connectionString)

{

services.AddSingleton<IFileRepository>(provider =>

new PostgreSqlFileRepository(connectionString));

return services;

}

}

### **Step 20: Docker Support**

dockerfile

*# Dockerfile*

FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base

WORKDIR /app

EXPOSE 80

EXPOSE 443

FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build

WORKDIR /src

COPY ["AECFileProcessor.API/AECFileProcessor.API.csproj", "AECFileProcessor.API/"]

COPY ["AECFileProcessor.Core/AECFileProcessor.Core.csproj", "AECFileProcessor.Core/"]

RUN dotnet restore "AECFileProcessor.API/AECFileProcessor.API.csproj"

COPY . .

WORKDIR "/src/AECFileProcessor.API"

RUN dotnet build "AECFileProcessor.API.csproj" -c Release -o /app/build

FROM build AS publish

RUN dotnet publish "AECFileProcessor.API.csproj" -c Release -o /app/publish

FROM base AS final

WORKDIR /app

COPY --from=publish /app/publish .

*# Install dependencies for OCR (if enabled)*

RUN apt-get update && apt-get install -y \

tesseract-ocr \

libtesseract-dev \

&& rm -rf /var/lib/apt/lists/\*

ENTRYPOINT ["dotnet", "AECFileProcessor.API.dll"]

yaml

*# docker-compose.yml*

version: '3.8'

services:

aec-file-processor:

build: .

ports:

- "8080:80"

environment:

- ASPNETCORE\_ENVIRONMENT=Development

- ProcessingConfiguration\_\_StorageType=SQLite

- ProcessingConfiguration\_\_EventBusType=InMemory

- FileWatcher\_\_WatchPath=/data/projects

volumes:

- ./data:/data

- ./database:/app/database

depends\_on:

- postgres

- redis

postgres:

image: postgres:15

environment:

POSTGRES\_DB: aec\_files

POSTGRES\_USER: aec\_user

POSTGRES\_PASSWORD: aec\_password

volumes:

- postgres\_data:/var/lib/postgresql/data

ports:

- "5432:5432"

redis:

image: redis:7-alpine

ports:

- "6379:6379"

volumes:

- redis\_data:/data

volumes:

postgres\_data:

redis\_data:

## **Migration Path to Full System**

### **Phase 4: Add Message Queue (Weeks 13-16)**

bash

*# Replace LocalEventBus with RabbitMQ or Azure Service Bus*

*# Add this to docker-compose.yml*

rabbitmq:

image: rabbitmq:3-management

ports:

- "5672:5672"

- "15672:15672"

environment:

RABBITMQ\_DEFAULT\_USER: aec\_user

RABBITMQ\_DEFAULT\_PASS: aec\_password

### **Phase 5: Microservices Split (Weeks 17-20)**

Split the monolith into separate services:

* **File Discovery Service**: Handles file system watching
* **Classification Service**: Handles file classification
* **Content Processing Service**: Handles metadata extraction
* **Data Service**: Handles database operations
* **API Gateway**: Routes requests to appropriate services

### **Phase 6: Cloud Migration (Weeks 21-24)**

* Replace SQLite with Azure SQL Database or PostgreSQL
* Replace local event bus with Azure Service Bus or Kafka
* Add blob storage for large file content
* Implement proper authentication and authorization
* Add comprehensive monitoring and logging

## **Key Benefits of This Approach**

### **1. Immediate Value**

* Working system in 2 weeks
* Can process thousands of files locally
* Real-time dashboard for monitoring
* Standard AEC file classification

### **2. Natural Scaling Path**

* Event-driven architecture scales to microservices
* Storage abstraction allows database migration
* Configuration-driven feature enabling
* Docker support for cloud deployment

### **3. Low Risk**

* Start small with SQLite and local processing
* Proven patterns that scale (events, CQRS concepts)
* Incremental feature addition
* Can run entirely offline

### **4. Developer Friendly**

* Single codebase initially
* Standard .NET patterns and libraries
* Clear separation of concerns
* Comprehensive testing support

This approach gives you a production-ready local system that can handle thousands of files while maintaining the architectural patterns needed to scale to the full enterprise system. You can start processing files immediately while building towards the complete solution.