Optimized Log Parser Code Review

Here's my analysis and recommendations for improving the resilience and memory efficiency of your log parser:

Key Improvements

1. Error Handling and Resilience

csharp

Copy

Download

*// Current implementation has basic try-catch but can be enhanced:*

public static List<string> ExtractLogEntries(string filePath)

{

if (string.IsNullOrWhiteSpace(filePath))

throw new ArgumentNullException(nameof(filePath));

if (!File.Exists(filePath))

throw new FileNotFoundException($"Log file not found: {filePath}");

var logEntries = new List<string>();

var timestampPattern = new Regex(@"^\d{2}[:|.]\d{2}[:|.]\d{2} - ", RegexOptions.Compiled);

try

{

using var reader = new StreamReader(filePath);

*// ... rest of implementation*

}

catch (IOException ex)

{

*// Specific handling for IO issues*

Debug.WriteLine($"IO Error reading log file: {ex.Message}");

throw; *// Re-throw or handle differently based on requirements*

}

catch (RegexMatchTimeoutException ex)

{

*// Handle potential regex timeouts*

Debug.WriteLine($"Regex timeout: {ex.Message}");

throw;

}

catch (Exception ex) when (LogException(ex))

{

*// General fallback with filtering*

return new List<string>(); *// Return empty list instead of potentially null*

}

}

private static bool LogException(Exception ex)

{

Debug.WriteLine($"Error processing log: {ex.Message}");

return true;

}

2. Memory Optimization

csharp

Copy

Download

*// Current implementation builds large lists in memory - consider:*

public static async IAsyncEnumerable<string> ExtractLogEntriesAsync(string filePath)

{

*// Same validation as above*

using var reader = new StreamReader(filePath);

StringBuilder? currentEntry = null;

while (await reader.ReadLineAsync() is string line)

{

if (timestampPattern.IsMatch(line))

{

if (currentEntry != null)

{

yield return currentEntry.ToString().Trim();

currentEntry.Clear(); *// Reuse StringBuilder*

}

currentEntry ??= new StringBuilder();

currentEntry.Append(line);

}

else if (currentEntry != null)

{

currentEntry.AppendLine(line);

}

}

if (currentEntry != null)

yield return currentEntry.ToString().Trim();

}

*// Usage would change to:*

await foreach (var entry in ExtractLogEntriesAsync(filePath))

{

*// Process entry*

}

3. JSON Serialization Optimization

csharp

Copy

Download

*// Current approach serializes everything at once - better:*

await using var fileStream = File.Create(outputPath);

await JsonSerializer.SerializeAsync(fileStream, myObjectList, new JsonSerializerOptions

{

WriteIndented = true,

PropertyNamingPolicy = JsonNamingPolicy.CamelCase,

DefaultBufferSize = 4096 *// Optimal buffer size*

});

4. File Processing Improvements

csharp

Copy

Download

*// Current file processing can be enhanced:*

var files = Directory.EnumerateFiles(folderPath, "\*.log")

.Where(f => IsValidFileName(Path.GetFileName(f)))

.ToList(); *// Materialize once if needed multiple times*

*// Process files in parallel if order doesn't matter*

await Parallel.ForEachAsync(files, async (file, ct) =>

{

await ProcessSingleFileAsync(file);

});

5. Regular Expression Optimization

csharp

Copy

Download

*// Compile regexes once as static fields*

private static readonly Regex TimestampPattern =

new(@"^\d{2}[:|.]\d{2}[:|.]\d{2} - ", RegexOptions.Compiled);

private static readonly Regex TagsPattern =

new(rx\_tagsline, RegexOptions.Compiled);

6. Object Pooling for LogEntry

csharp

Copy

Download

*// Consider using object pooling for LogEntry objects*

private static readonly ObjectPool<LogEntry> LogEntryPool =

ObjectPool.Create<LogEntry>();

*// Usage:*

var entry = LogEntryPool.Get();

try

{

*// Populate entry*

myObjectList.Add(entry);

}

finally

{

*// Don't return to pool here if keeping in list*

*// Return when done with the list*

}

Full Optimized Implementation

Here's how you might restructure the core processing:

csharp

Copy

Download

public class LogProcessor

{

private static readonly Regex TimestampPattern =

new(@"^\d{2}[:|.]\d{2}[:|.]\d{2} - ", RegexOptions.Compiled);

private static readonly Regex FileNamePattern =

new(rxLogFile, RegexOptions.Compiled | RegexOptions.IgnoreCase);

private static readonly Regex TagsPattern =

new(rx\_tagsline, RegexOptions.Compiled);

public async Task ProcessLogsAsync(string folderPath)

{

var sw = Stopwatch.StartNew();

var files = Directory.EnumerateFiles(folderPath, "\*.log")

.Where(f => IsValidFileName(Path.GetFileName(f)));

await Parallel.ForEachAsync(files, async (file, ct) =>

{

try

{

await ProcessFileAsync(file);

}

catch (Exception ex)

{

Debug.WriteLine($"Error processing {file}: {ex.Message}");

}

});

Console.WriteLine($"Total processing time: {sw.Elapsed.TotalSeconds}s");

}

private async Task ProcessFileAsync(string filePath)

{

var fileName = Path.GetFileName(filePath);

var fileInfo = ParseFileName(fileName);

var logEntries = ExtractLogEntriesAsync(filePath);

var entries = new List<LogEntry>();

await foreach (var entry in logEntries)

{

entries.Add(ParseLogEntry(entry, fileInfo));

}

await WriteOutputFilesAsync(filePath, entries, fileInfo);

}

private async IAsyncEnumerable<string> ExtractLogEntriesAsync(string filePath)

{

await using var reader = new StreamReader(filePath);

StringBuilder? currentEntry = null;

while (await reader.ReadLineAsync() is string line)

{

if (TimestampPattern.IsMatch(line))

{

if (currentEntry != null)

{

yield return currentEntry.ToString().Trim();

currentEntry.Clear();

}

currentEntry ??= new StringBuilder(1024); *// Initial capacity*

currentEntry.Append(line);

}

else if (currentEntry != null)

{

currentEntry.AppendLine(line);

}

}

if (currentEntry?.Length > 0)

yield return currentEntry.ToString().Trim();

}

private FileInfo ParseFileName(string fileName)

{

var match = FileNamePattern.Match(fileName);

return match.Success switch

{

true when match.Groups[5].Success => new FileInfo(

HostName: string.Empty,

Prefix: match.Groups[5].Value,

Date: match.Groups[6].Value),

true when match.Groups[4].Success => new FileInfo(

HostName: match.Groups[2].Value,

Prefix: match.Groups[3].Value,

Date: match.Groups[4].Value),

\_ => throw new FormatException($"Invalid file name format: {fileName}")

};

}

private LogEntry ParseLogEntry(string entry, FileInfo fileInfo)

{

*// Similar to your existing parsing logic*

*// Return populated LogEntry*

}

private async Task WriteOutputFilesAsync(string inputPath, List<LogEntry> entries, FileInfo fileInfo)

{

var basePath = Path.Combine(Path.GetDirectoryName(inputPath),

Path.GetFileNameWithoutExtension(inputPath));

*// Write main JSON*

await using (var fs = File.Create($"{basePath}.json"))

{

await JsonSerializer.SerializeAsync(fs, entries, new JsonSerializerOptions

{

WriteIndented = true,

PropertyNamingPolicy = JsonNamingPolicy.CamelCase

});

}

*// Write header*

var header = CreateHeader(inputPath, entries, fileInfo);

await using (var fs = File.Create($"{basePath}\_header.json"))

{

await JsonSerializer.SerializeAsync(fs, header, new JsonSerializerOptions

{

WriteIndented = true,

PropertyNamingPolicy = JsonNamingPolicy.CamelCase

});

}

}

private record FileInfo(string HostName, string Prefix, string Date);

}

Key Benefits of This Approach

1. **Better Error Handling**: More specific exception handling and recovery
2. **Memory Efficiency**: Uses streaming and async patterns to reduce memory pressure
3. **Performance**: Parallel file processing and compiled regexes
4. **Maintainability**: Better separation of concerns
5. **Scalability**: Can handle very large log files without memory issues
6. **Resilience**: Continues processing other files if one fails

Would you like me to elaborate on any particular aspect of these optimizations?