**Box Processor Project**

**Traditional & Cloud-Native Approaches**

**Traditional Approach:**

This is current implementation.

**Key Points:**

**Console Application:** The Box Processor project is implemented as a console application using .NET Framework 6.

**Startup.cs:** The Startup.cs class is utilized to register all the service dependencies. Class constructors are employed to inject these dependencies.

**Configuration Settings:** All configuration settings, such as the database connection string and time settings, are declared in the appsettings.json file.

**Test Project:** The project includes a test project that utilizes the NUnit framework. Currently, there is only one test case, but it can be extended further (e.g., checking for locked files).

**Packages Used:**

Microsoft.EntityFrameworkCore Version=6.0.28

Microsoft.EntityFrameworkCore.SqlServer Version=6.0.28

Microsoft.EntityFrameworkCore.Tools Version=6.0.28

Microsoft.Extensions.Configuration Version=8.0.0

Microsoft.Extensions.Configuration.Json Version=8.0.0

Microsoft.Extensions.DependencyInjection Version=8.0.0

EF Core: Entity Framework Core is used for object-relational mapping.

**Retry Mechanisms:**

A retry mechanism exists for monitoring input files if they are already locked. This is driven by the MaxAttempt setting in the configuration file.

Additionally, a retry mechanism is implemented if a file fails to process the first time. This is driven by the Max\_RETRY variable (which can be put in the settings file) in the FileTrackingService class.

**Batch Processing:**

The project allows for a certain number of files to be processed to the database using a batch size setting in the appsettings file. For example, if there are ten input files and the batch size is set as 5, then the data from the first 5 files will be saved to the database, followed by the next 5 files. This approach improves performance, reduces database load, and optimizes resource utilization.

**Cloud-Native Approach:**

**Key Points:**

**Serverless Architecture:** The cloud-native approach utilizes serverless architecture components offered by Azure.

**Azure Functions:**

Processing logic is implemented using Azure Functions, which are triggered by events such as file uploads to Azure Blob Storage.

**Azure Blob Storage:**

Azure Blob Storage is utilized for storing both input files and processed data. Blob triggers are employed to invoke Azure Functions whenever new files are uploaded or modified in the input container.

**Logic Apps:**

Azure Logic Apps orchestrate the entire workflow, including monitoring input folders, triggering processing functions, and handling errors or retries. Logic Apps offer seamless integration with various Azure services and external systems.

**Azure SQL Database:**

Processed data is stored in Azure SQL Database for structured storage and easy querying. Azure SQL Database is a fully managed service, reducing operational overhead.

**Benefits of Cloud-Native Approach:**

**Scalability:** Azure Functions and Azure Blob Storage automatically scale based on demand, allowing the solution to handle varying workloads efficiently.

**Reliability:** Cloud-native services are designed for high availability and fault tolerance, ensuring operational continuity even in the event of failures.

**Cost Efficiency:** Pay only for resources used with Azure Functions and Azure Blob Storage, minimizing costs and optimizing resource utilization.

**Managed Services:** Azure services are fully managed, reducing operational burden and allowing focus on building and improving the application.

By combining both approaches, you can leverage the flexibility and scalability of cloud-native services while retaining the familiarity and control of traditional application architectures.