Design Document for Distributor Data Extraction Ingestion (Version 3)

Nathan Lunceford

2025-03-04

Table of contents

1 Overview

This design document covers the extraction ingestion component of our ETL pipeline. In our architecture, distributors integrate with our system via two primary methods:

- **Direct Extraction Ingestion:** Where our system automatically provisions the correct resources based solely on the three supplied parameters: `--erp-type`, `--client-id`, and `--data-type`.
- **SFTP Ingestion:** Where distributors SFTP their data to our environment (this method is handled separately).

The design detailed here pertains only to the **direct extraction ingestion** method. This process extracts distributor data and deposits it into a pre-dropzone S3 bucket. At this stage, a minor transformation is applied: column names are standardized according to our global data dictionary and the output files are converted to Parquet format. More complex transformations occur later in the pipeline.

2 System Architecture

2.1 Entry Point and Application Hosting

- **Program Class:**
  - **Main Method:**
    - Parses command-line options (e.g., `--erp-type`, `--client-id`, `--data-type`) using `System.CommandLine`.
    - Sets up the dependency injection container and configures services.
    - Resolves the primary extraction ingestion service (`ERPService`) from the DI container and triggers the extraction process.
  - **CreateHostBuilder Method:**
    - Configures the host with HashiCorp Vault, FerretDB services, core providers, and builder components.
    - Configures `IHttpClientFactory` with resilience policies for API-based integrations.
    - Sets up feature flags, metrics collection, and health checks.

2.2 Extraction Ingestion Orchestration

- **ERPService Class:**
  - Acts as the orchestrator for the extraction ingestion process.
  - Retrieves credentials from HashiCorp Vault and configuration from FerretDB, respectively.
  - Dynamically resolves components for data extraction based on ERP type using factory functions.
  - Uses **IHttpClientFactory** for managing HTTP connections in API mode.
  - Supports two extraction modes:
    - **API Mode:** For ERPs that expose APIs.
    - **Database Mode:** For ERPs that require direct database access.
  - Supports both batch processing and incremental extraction with change data capture (CDC).
  - Applies a minimal transform:
    - Standardizes column names according to our global data dictionary.
    - Applies data quality validations against predefined rules.
    - Converts all files to Parquet format with optional compression.
  - Deposits the extracted (and minimally transformed) data into a pre-dropzone S3 bucket.
  - Tracks data lineage for audit and compliance purposes.
  - (Note: Subsequent, more complex transformations are performed later in the overall ETL pipeline.)

2.3 External Dependencies

- **Infrastructure Services:**
  - **IVaultService:** Retrieves ERP credentials from HashiCorp Vault with least privilege access.
  - **IFerretDBService:** Fetches ERP configuration data from FerretDB.
  - **IAmazonS3:** Uploads the extracted data into the pre-dropzone S3 bucket.
  - **IFeatureFlagService:** Manages feature flags for gradual rollout of functionality.
  - **ICatalogService:** Integrates with data catalog for metadata management.
- **.NET Libraries:**
  - **System.CommandLine:** For command-line parsing.
  - **Microsoft.Extensions.Hosting & DI:** For hosting and dependency injection.
  - **System.Text.Json:** For JSON serialization/deserialization.
  - **IHttpClientFactory:** For managing HttpClient instances in API extraction mode.
  - **VaultSharp:** For interacting with HashiCorp Vault.

- **MongoDB.Driver:** For FerretDB interactions (FerretDB uses MongoDB wire protocol).
- **Polly:** For resilience patterns including circuit breakers and bulkheads.
- **OpenTelemetry:** For distributed tracing and metrics.
- **FluentValidation:** For data contract validation.

3 Design Patterns and Advanced Resilience Strategies

3.1 Dependency Injection (DI)

- **Usage:**
  - Decouples service construction from business logic.
  - Registers Vault client, FerretDB connection, providers, factory functions, builder components, and the main extraction service in the DI container.
  - Uses factory functions to dynamically resolve implementations based on runtime parameters.
  - Configures `IHttpClientFactory` and related services.
- **Benefits:**
  - Enhances testability and maintainability.
  - Promotes separation of concerns.
  - Leverages built-in .NET capabilities without custom abstractions.

3.2 Factory Function Approach

- **Components:**
  - Registered factory functions for dynamic resolution: `Func<string, IExtractor>`, `Func<string, string, ITransformer>`, `Func<string, IDataUploader>`.
- **Usage:**
  - Centralizes lookup logic directly in the DI container.
  - Dynamically resolves connectors, extractors, transformers, and uploaders based on ERP type or data type at runtime.
- **Benefits:**
  - Simplifies implementation by leveraging built-in .NET DI capabilities.
  - Eliminates custom registry classes while maintaining the dynamic resolution capability.
  - Provides clear, testable dependency paths.
  - Makes component selection logic more transparent.

3.3 Abstract Factory Pattern

- **Component:**
  - Factory functions registered in the DI container.
  - `IHttpClientFactory` for HTTP client management.
- **Usage:**
  - Encapsulates creation of ERP-specific components.
- **Benefits:**
  - Supports multiple ERP systems with varying implementations without altering extraction logic.

3.4 Builder Pattern

- **Components:**
  - **APIRequestBuilder & AuthenticationBuilder:**
    - Provide fluent interfaces to construct complex API request objects.
  - **DatabaseQueryBuilder:**
    - Dynamically constructs SQL queries for ERPs requiring direct database access.
  - **ExtractConfigBuilder:**
    - Builds extraction configurations supporting both full and incremental extracts.
- **Usage:**
  - The `DatabaseQueryBuilder` collects parameters and produces a `DatabaseQuery` object with a `GenerateSql` method.
  - The `APIRequestBuilder` works with `IHttpClientFactory` to construct properly configured HTTP requests.
  - The `ExtractConfigBuilder` creates configurations for different extraction modes.
- **Benefits:**
  - Enhances readability and modularity.
  - Supports multiple extraction modes seamlessly.

3.5 Strategy Pattern

- **Components:**
  - Interfaces such as `IExtractor`, `ITransformer`, and `IValidator`.

- **Usage:**
  - Encapsulate different implementations for data extraction, transformation, and validation.
  - Factory functions select the appropriate strategy at runtime.
- **Benefits:**
  - Provides flexibility to extend or change algorithms without impacting the overall system.

3.6 Decorator Pattern

- **Components:**
  - **VaultCredentialProviderDecorator:**
    - Adds caching behavior to Vault credential retrieval.
  - **MetricsDecorator:**
    - Wraps core services to collect performance metrics.
  - **EncryptionDecorator:**
    - Adds field-level encryption for sensitive data.
  - **DataQualityDecorator:**
    - Adds data quality checks to transformations.
- **Usage:**
  - Enhances existing components with additional functionality without modifying their core logic.
- **Benefits:**
  - Improves performance through caching.
  - Provides observability and enhances security.
  - Ensures data quality through validation.

3.7 Circuit Breaker Pattern

- **Components:**
  - **VaultCircuitBreaker**, **FerretDBCircuitBreaker**, **S3CircuitBreaker**
- **Usage:**
  - Prevents cascading failures by breaking the circuit when dependencies fail.
  - Automatically restores service when dependencies recover.

- **Benefits:**
  - Enhances system resilience during partial outages.
  - Prevents overwhelming failing services with requests.

3.8 Bulkhead Pattern

- **Components:**
  - Separate connection pools for different ERP types and operations.
  - Isolated resource pools for critical vs. non-critical operations.
- **Usage:**
  - Isolates failures to prevent system-wide degradation.
  - Allocates resources based on operation criticality.
- **Benefits:**
  - Prevents resource exhaustion.
  - Improves system stability during partial failures.

3.9 Exponential Backoff with Jitter

- **Usage:**
  - Implements intelligent retry strategies for all external service calls.
  - Adds randomization to prevent thundering herd problems.
- **Benefits:**
  - Prevents overwhelming recovering services.
  - Distributes retry attempts evenly over time.

4 Detailed Component Descriptions

4.1 Program Class

- **Main Method:**
  - Parses Nomad-supplied command-line arguments.
  - Builds the DI container via `CreateHostBuilder`.
  - Resolves and invokes `ERPService.ProcessERPData`.
  - Handles global errors for graceful failure.

- **CreateHostBuilder Method:**
  - Configures services including Vault client, FerretDB connection, providers, and builder components.
  - Registers factory functions for dynamic component resolution.
  - Sets up resilience policies with circuit breakers and bulkheads.
  - Configures feature flags, metrics collection, and health checks.
  - Sets up OpenAPI documentation generation.

4.2 ERPService Class

- **Responsibilities:**
  - Orchestrates the extraction ingestion process.
  - Retrieves credentials and ERP configuration.
  - Dynamically resolves ERP-specific components using injected factory functions.
  - Chooses between API or Database extraction modes based on the ERP configuration.
  - Supports both batch and incremental extraction modes.
  - Applies a minor transformation with data quality validation.
  - Deposits the transformed data into a pre-dropzone S3 bucket.
  - Tracks data lineage for audit and compliance.
- **Key Method – ProcessERPData:**
  - **Credential & Configuration Retrieval:**
    - Uses `ICredentialProvider` to obtain short-lived, least-privilege credentials.
    - Uses `IConfigurationProvider` to obtain ERP settings.
  - **Dynamic Component Resolution:**
    - Uses factory functions to resolve ERP-specific extractors, transformers, and uploaders.
  - **Integration Modes:**
    - **API Mode:**
      - Builds an API request via `APIRequestBuilder` with mutual TLS if supported.
      - Extracts data via `IExtractor.Extract` with circuit breaker protection.
    - **Database Mode:**
      - Builds a SQL query using `DatabaseQueryBuilder`.
      - Extracts data via `IExtractor.ExtractFromDatabase` with connection isolation.
  - **Extraction Modes:**
    - **Full Extract:** Retrieves all data for the specified type.

        * **Incremental Extract:** Retrieves only changed data since the last extraction.
    - **Self-Healing:**
        * Implements automatic recovery procedures for common failure scenarios.
    - **Subsequent Steps:**
        * Applies the minor transform with data quality validation.
        * Records data lineage metadata.
        * Uploads the resulting data into a pre-dropzone S3 bucket via `IDataUploader`.

4.3 Providers

- **VaultCredentialProvider:**
    - Retrieves and deserializes credentials from HashiCorp Vault.
    - Implements caching with TTL-based invalidation.
    - Supports dynamic secret rotation with configurable TTL.
    - Generates least-privilege, operation-specific credentials.
- **FerretDBConfigProvider:**
    - Fetches ERP configuration from FerretDB and maps it to an `ERPConfiguration` object.
    - Uses MongoDB driver since FerretDB implements MongoDB wire protocol.
    - Supports read-preference strategies for replica sets.
    - Implements schema evolution for backward compatibility.

4.4 Data Uploader – S3DataUploader

- **Responsibilities:**
    - Formats and uploads the minimally transformed data into S3.
    - Applies data compression for efficient storage and transfer.
    - Converts files to Parquet format and ensures standardized column names.
    - Encrypts sensitive fields before upload.
    - Records data lineage metadata.
- **Key Methods:**
    - `Upload`: Manages the upload process with checksums and integrity verification.
    - `FormatData`: Applies the transformation with data quality checks.
    - `TrackLineage`: Records data provenance information.

4.5 Data Quality and Validation

- **DataContractValidator:**
  - **Responsibilities:**
    - Validates data against predefined schemas and rules.
    - Reports quality issues with detailed diagnostics.
    - Enforces data governance policies.
  - **Key Methods:**
    - `ValidateSchema`: Ensures data adheres to expected structure.
    - `ValidateValues`: Checks data values against business rules.
    - `GenerateReport`: Creates detailed validation reports.
- **DataMaskingService:**
  - **Responsibilities:**
    - Identifies and masks sensitive information (PII).
    - Supports various masking techniques (hashing, tokenization, etc.).
    - Maintains referential integrity across masked datasets.
  - **Key Methods:**
    - `IdentifySensitiveFields`: Automatically detects potential PII.
    - `ApplyMasking`: Applies appropriate masking techniques.
    - `VerifyMasking`: Ensures masking effectiveness.

5 Pseudo-code for Nomad Integration

```
/// <summary>
/// Configures and builds the host with all necessary services.
/// </summary>
CreateHostBuilder:
    return HostBuilder()
        .ConfigureServices(services =>
        {
            // Configure Vault client with circuit breaker
            services.AddSingleton<IVaultClient>(provider =>
            {
                var vaultOptions = new VaultClientSettings(
                    "https://vault.example.com:8200",
                    new AppRoleAuthMethodInfo(roleId, secretId)
                );
```

```
            vaultOptions.RetrySettings = new RetrySettings {
                Enabled = true,
                MaxAttempts = 5,
                BackoffType = BackoffType.ExponentialWithJitter
            };
            return new VaultClient(vaultOptions);
        });

        // Configure FerretDB connection with bulkhead isolation
        services.AddSingleton<IMongoClient>(provider =>
        {
            var settings = MongoClientSettings.FromConnectionString(
                "mongodb://ferretdb.example.com:27017"
            );
            settings.RetryWrites = true;
            settings.RetryReads = true;
            settings.ServerSelectionTimeout = TimeSpan.FromSeconds(5);
            settings.MaxConnectionPoolSize = 100;
            return new MongoClient(settings);
        });

        // Configure feature flag service
        services.AddSingleton<IFeatureFlagService, FeatureFlagService>();

        // Register providers with caching decorators
        services.AddSingleton<ICredentialProvider, VaultCredentialProvider>();
        services.Decorate<ICredentialProvider, CachedCredentialProviderDecorator>();

        services.AddSingleton<IConfigurationProvider, FerretDBConfigProvider>();

        // Register data quality and validation services
        services.AddSingleton<IDataContractValidator, DataContractValidator>();
        services.AddSingleton<IDataMaskingService, DataMaskingService>();

        // Register data lineage service
        services.AddSingleton<IDataLineageService, DataLineageService>();

        // Register concrete implementations
        services.AddSingleton<EpicorExtractor>();
        services.AddSingleton<SageExtractor>();
        services.AddSingleton<EpicorTransformer>();
        services.AddSingleton<SageTransformer>();
```

```
            services.AddSingleton<S3DataUploader>();
            services.AddSingleton<LocalFileUploader>();

            // Register factory functions for dynamic resolution
            services.AddSingleton<Func<string, IExtractor>>(sp => erpType =>
                erpType switch {
                    "Epicor" => sp.GetRequiredService<EpicorExtractor>(),
                    "Sage" => sp.GetRequiredService<SageExtractor>(),
                    _ => throw new ArgumentException($"Unknown ERP type: {erpType}")
                });

            services.AddSingleton<Func<string, string, ITransformer>>(sp => (erpType, dataTy
                (erpType, dataType) switch {
                    ("Epicor", _) => sp.GetRequiredService<EpicorTransformer>(),
                    ("Sage", _) => sp.GetRequiredService<SageTransformer>(),
                    _ => throw new ArgumentException($"Unsupported combination: {erpType}, {
                });

            services.AddSingleton<Func<string, IDataUploader>>(sp => uploaderType =>
                uploaderType switch {
                    "s3" => sp.GetRequiredService<S3DataUploader>(),
                    "local" => sp.GetRequiredService<LocalFileUploader>(),
                    _ => throw new ArgumentException($"Unknown uploader type: {uploaderType}"
                });

            // Register builders
            services.AddSingleton<IAPIRequestBuilder, APIRequestBuilder>();
            services.AddSingleton<IDatabaseQueryBuilder, DatabaseQueryBuilder>();
            services.AddSingleton<IAuthenticationBuilder, AuthenticationBuilder>();
            services.AddSingleton<IExtractConfigBuilder, ExtractConfigBuilder>();

            // Register S3 client for data upload
            services.AddAWSService<IAmazonS3>();

            // Register core service with decorators for cross-cutting concerns
            services.AddSingleton<ERPService>();
            services.Decorate<ERPService, MetricsERPServiceDecorator>();
            services.Decorate<ERPService, DataQualityDecorator>();
            services.Decorate<ERPService, EncryptionDecorator>();

            // Configure HTTP clients with resilience policies using Polly
            services.AddHttpClient("default")
```

```
                .AddTransientHttpErrorPolicy(builder =>
                    builder.WaitAndRetryAsync(
                        retryCount: 3,
                        sleepDurationProvider: retryAttempt =>
                            TimeSpan.FromSeconds(Math.Pow(2, retryAttempt)) +
                            TimeSpan.FromMilliseconds(new Random().Next(0, 1000)), // Jitter
                        onRetry: (outcome, timespan, retryAttempt, context) => {
                            // Log retry attempt
                            logger.LogWarning($"Retry {retryAttempt} for {context.PolicyKey}
                        }
                    ))
                .AddCircuitBreakerPolicy(builder =>
                    builder.CircuitBreakerAsync(
                        handledEventsAllowedBeforeBreaking: 5,
                        durationOfBreak: TimeSpan.FromSeconds(30),
                        onBreak: (outcome, breakDelay) => {
                            logger.LogError($"Circuit broken for {breakDelay.TotalSeconds}s!"
                        },
                        onReset: () => {
                            logger.LogInformation("Circuit reset!");
                        }
                    ));

            // Add OpenTelemetry tracing
            services.AddOpenTelemetryTracing(builder => {
                builder
                    .SetResourceBuilder(ResourceBuilder.CreateDefault().AddService("erp-extra
                    .AddSource("erp-extractor")
                    .AddHttpClientInstrumentation()
                    .AddMongoDBInstrumentation()
                    .AddAspNetCoreInstrumentation()
                    .AddJaegerExporter();
            });
        });

/// <summary>
/// Processes ERP data extraction and performs initial transformation.
/// </summary>
/// <param name="erpType">The type of ERP system to extract from</param>
/// <param name="clientId">The client identifier</param>
/// <param name="dataType">The type of data to extract</param>
ERPService.ProcessERPData(erpType, clientId, dataType):
```

```
    Log "Starting ETL extraction ingestion for client [clientId] using ERP [erpType]"

    /// <summary>Start metrics collection for this operation</summary>
    using (metricsTimer = MetricsService.StartTimer("erp_process_data",
                                                     { "erp_type": erpType, "client_id": clientI
    using (tracer = TracingService.StartTrace("ProcessERPData"))
    {
        /// <summary>Start data lineage tracking</summary>
        lineage = DataLineageService.StartLineageRecord(erpType, clientId, dataType)

        /// <summary>Check feature flags for enabled features</summary>
        bool useIncrementalExtract = FeatureFlagService.IsFeatureEnabled("IncrementalExtract"
        bool useCompression = FeatureFlagService.IsFeatureEnabled("Compression", clientId)
        bool useFieldEncryption = FeatureFlagService.IsFeatureEnabled("FieldEncryption", clie

        /// <summary>Retrieve least-privilege credentials from HashiCorp Vault</summary>
        credentials = CredentialProvider.GetLeastPrivilegeCredentials(erpType, clientId, data

        /// <summary>Retrieve configuration from FerretDB</summary>
        erpConfig = ConfigurationProvider.GetConfiguration(erpType, clientId)

        /// <summary>
        /// Dynamically resolve ERP-specific components using factory functions:
        /// - Data extractor (for API or DB extraction)
        /// - Data transformer (to standardize columns and convert to Parquet)
        /// - Data uploader (to upload data to the pre-dropzone S3 bucket)
        /// </summary>
        extractor = ExtractorFactory(erpType);
        transformer = TransformerFactory(erpType, dataType);
        uploader = UploaderFactory("s3");

        /// <summary>Build extraction configuration based on mode</summary>
        extractConfig = ExtractConfigBuilder.New()
            .ForERP(erpType)
            .ForClient(clientId)
            .ForDataType(dataType)
            .UseIncrementalExtract(useIncrementalExtract && erpConfig.SupportsCDC)
            .WithLastExtractTime(useIncrementalExtract ? GetLastExtractTime(erpType, clientId
            .WithBatchSize(erpConfig.BatchSize)
            .Build()

        /// <summary>Handle database extraction mode</summary>
```

```
        if erpConfig.AccessType == Database then:
            queryBuilder = DatabaseQueryBuilder()
                .ForERP(erpType)
                .WithConnectionString(erpConfig.ConnectionString)
                .WithSchema(erpConfig.Schema)
                .WithTable(dataType + "_table")
                .WithColumns("id", "created_at", "data")

            if useIncrementalExtract && erpConfig.SupportsCDC:
                queryBuilder.WithWhere(erpConfig.WatermarkColumn, ">", extractConfig.LastExt
            else:
                queryBuilder.WithWhere("is_processed", false)

            query = queryBuilder
                .WithOrderBy("created_at")
                .WithLimit(erpConfig.BatchSize)
                .WithCommandTimeout(erpConfig.TimeoutSeconds)
                .Build()

            Log "Executing database query: " + query.GenerateSql()

            // Use bulkhead isolation for database connection
            using (bulkhead = BulkheadPolicy.Execute(erpType + "-database", () => {
                extractedData = extractor.ExtractFromDatabase(query, extractConfig)
                return extractedData
            }))

        /// <summary>Handle API extraction mode</summary>
        else:
            authBuilder = AuthenticationBuilder()
                .WithApiKey(credentials.ApiKey)
                .WithClientId(credentials.ClientId)
                .WithClientSecret(credentials.ClientSecret)

            if erpConfig.SupportsMutualTLS:
                authBuilder.WithClientCertificate(credentials.ClientCertificate)

            auth = authBuilder.Build()

            requestBuilder = APIRequestBuilder()
                .ForERP(erpType)
                .WithEndpoint(erpConfig.BaseUrl + "/api/v2/sales")
```

```
                .WithMethod(GET)
                .WithAuthentication(auth)
                .WithHeaders(erpConfig.RequiredHeaders)

            if useIncrementalExtract && erpConfig.SupportsCDC:
                requestBuilder.WithQueryParameters({
                    "companyId": erpConfig.CompanyId,
                    "warehouse": erpConfig.WarehouseId,
                    "pageSize": erpConfig.PageSize.toString(),
                    "changedSince": extractConfig.LastExtractTime.toISOString()
                })
            else:
                requestBuilder.WithQueryParameters({
                    "companyId": erpConfig.CompanyId,
                    "warehouse": erpConfig.WarehouseId,
                    "pageSize": erpConfig.PageSize.toString()
                })

            request = requestBuilder
                .WithRetryPolicy(erpConfig.MaxRetries)
                .WithTimeout(erpConfig.TimeoutSeconds)
                .Build()

            Log "Executing API request to " + erpConfig.BaseUrl + "/api/v2/sales"

            // Execute with circuit breaker protection
            extractedData = CircuitBreakerPolicy
                .ForService("erp-api-" + erpType)
                .Execute(() => extractor.Extract(request, extractConfig))

        /// <summary>Process extracted data as before</summary>
        // Validate data against contract
        validationResult = validator.Validate(extractedData)

        // Transform the data
        transformedData = transformer.Transform(extractedData)

        // Upload to S3
        uploadResult = uploader.Upload(transformedData, uploadConfig)

        Log "Extraction ingestion process completed successfully"
    }
```

6 Conclusion

This design document outlines an optimized approach to the distributor data extraction pipeline. By leveraging .NET's built-in dependency injection capabilities more effectively, we've eliminated the need for custom registry classes while maintaining the flexibility to dynamically resolve the right components based on runtime parameters.

The simplified architecture uses factory functions registered directly in the DI container to dynamically select the appropriate extractors, transformers, and uploaders based on the ERP type and data type. This approach:

1. Reduces complexity by eliminating custom registry abstractions
2. Improves maintainability by centralizing resolution logic in the DI container
3. Enhances testability by providing clear injection patterns
4. Preserves all the flexibility of the previous design
5. Leverages native .NET capabilities rather than custom implementations

The system maintains its comprehensive resilience features, security controls, and data quality validations while implementing a more elegant approach to component resolution. This design is more aligned with .NET best practices and will be easier to maintain and extend as new ERP integrations are added.