envobi

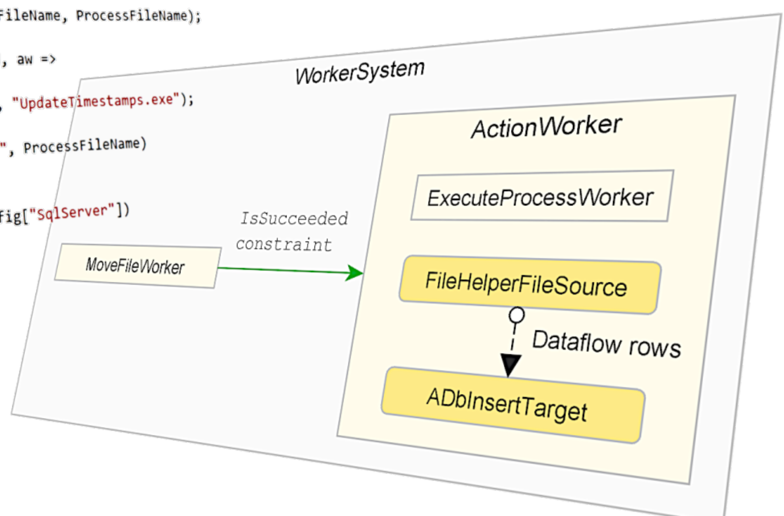
Whitepaper
February 2021
envobi.com

Thirteen Factors Crippling ETL Productivity

With SQL Server® Integration Services *and* **actionETL™** .NET Library *comparison*

By Kristian Wedberg

```
new WorkerSystem()
.Root(root =>
{
    var mfw = new MoveFileWorker(root, "Move CSV", SourceFileName, ProcessFileName);

    new ActionWorker(root, "Group", () => mfw.IsSucceeded, aw =>
    {
        new ExecuteProcessWorker(aw, "Update Timestamps", "UpdateTimestamps.exe");

        new FileHelperFileSource<Category>(aw, "Read CSV", ProcessFileName)

        .Output.Link.AdbInsertTarget("Insert"
            , provider.CreateConnectionBuilder(root.Config["SqlServer"])
            , "dbo.Category");
    });
})
.Start()
.ThrowOnFailure();
```



envobi

Contents

Overview....3

actionETL .NET Library....4

Turning Weaknesses into Strengths....6

Conclusion....10

References....10

Appendix - **actionETL** Features....11

Appendix - **actionETL** Workers....13

© 2021 Envobi Ltd. Envobi and actionETL, and their respective logos are trademarks of Envobi Ltd. All other company names, products and services used herein may be trademarks of their respective owners. This document is for informational purposes only, and is subject to change without notice. Envobi makes no warranties, express or implied, in this publication.

Overview

ETL design, implementation and maintenance is often very time consuming (and can be error prone) in complex batch data integration projects (e.g. a Data Warehouse), often consuming as much as 80%[2] of the overall effort.

In studying several traditional (visual) ETL tools (i.e. SQL Server® Integration Services – SSIS, Talend® Open Studio etc.), the following factors were identified as very common ETL tool strengths that *increased* productivity in terms of designing, building, and maintaining ETL solutions:

Common ETL tool strengths	
ETL specific	*Non-ETL specific*

• Control flow, with
  ○ Start constraints
  ○ Hierarchical structure
• Dataflow, with
  ○ Row-by-Row processing and high performance
  ○ Debug rows in flight
	• "Divide and conquer" – implement requirements via many smaller parts
	• Highly parallel
	• Visualize parts, constraints, and structure

When used on small or trivial projects, almost all tools do well. When used on *complex* ETL projects however, the following thirteen factors were identified as very common (albeit not universal) ETL tool *weaknesses*, having a large *detrimental* impact on productivity, thereby increasing effort and cost, as well as decreasing quality:

Common ETL tool weaknesses	
ETL specific	*Non-ETL specific*
• Control flow - Dataflow separation • Data source specific data types • Micro batches • Separate (un)recoverable errors	• Visual programming • Many disjoint programming models and languages • Reusability • Composability • Encapsulation • Testability • Refactoring • Extensibility • Source control and Continuous Integration/Continuous Delivery

Note that the "Non-ETL specific" weaknesses are all areas where application development using general programming languages and tools do extremely well.

We use SSIS as an example of a traditional ETL tool, and evaluate it on the above common weaknesses.

We also evaluate **actionETL** (described below), which has been designed to overcome these common weaknesses.

> ***With excellent reusability and composability,*** *actionETL* ***required*** *23 times less* ***C# code***[5] ***(9kB) to create a high performance and reusable custom Slowly Changing Dimension (SCD) worker, vs. similar custom functionality implemented in SSIS***[3] ***(209kB).***

actionETL Cross-platform .NET Library

actionETL is a cross-platform high performance, highly productive .NET library and NuGet package for easily writing ETL data processing applications in .NET languages such as **C#** and **VB** running on **Windows** and **Linux**. You can reference and call the library from your application, or use it to create and run e.g. a console program executable. It is suitable for ETL developers with anywhere from limited to extensive .NET programming experience, and equally for .NET developers that have ETL requirements.

actionETL combines the best of the ETL mindset with the tools and techniques of modern *application* development. It retains all ETL tool strengths identified above (except not being a visual tool), while fixing *all* thirteen weaknesses identified above. For productivity, this is an excellent trade-off, significantly reducing the effort and cost, as well as improving the quality of complex ETL projects.

> *While **actionETL** could in the future get visualizers to* display *workers, dependencies, dataflows etc., to avoid the same pitfalls, it won't get its own drag and drop, visual* programming *environment.*

actionETL currently includes 68 *workers* (listed in *Appendix—**actionETL** Workers*), corresponding to SSIS control flow tasks, dataflow components, containers, and packages.

Typical **actionETL** use cases include:

- Replacing or augmenting traditional ETL tools
- Replacing or augmenting ETL written in pure SQL, pure .NET, Python code etc.

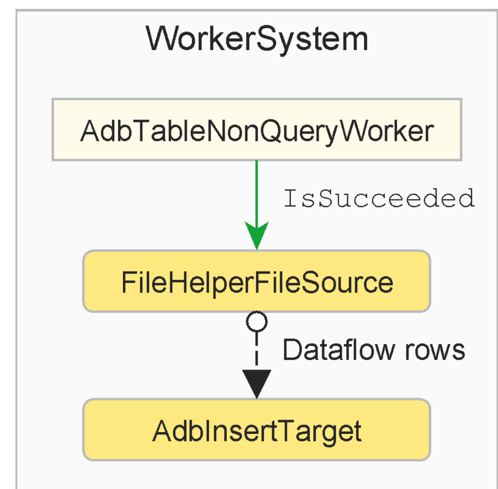
With its **.NET Framework**, **.NET Standard** and **.NET Core** support it can run both on-premises and in cloud environments.

> ***Note that cloud ETL front-ends borrow many aspects from traditional ETL tools, thereby also being impacted by many of the same productivity factors detailed in this paper.***

In the below example **actionETL** truncates a database table, and then uses a dataflow to load the table from a CSV file:

- A single `Category` class is reused, avoiding repetition:
    - Defines both the CSV file format, and the dataflow row schema
    - Applies the row schema to the source worker, while the downstream target worker gets it set automatically
- Gets two settings from configuration files
- Mixes control flow and dataflow workers, including using constraints (which can also be set to and from transform and target workers)

```
[DelimitedRecord(",")] // CSV file format
public class Category
{ // CSV and dataflow columns, with types
  public int CategoryId;
  public string CategoryName;
  public string CategoryDescription;
}

const string CategoryTableName = "Media.Category";

var cs = AdbSqlClientProvider.Instance
  .CreateConnectionStringFromAppConfig("DemoDb");

new WorkerSystem() // Create the worker system
  .Root(ws =>
  {
    var truncate = new AdbTableNonQueryWorker(ws
      , cs.CreateConnectionBuilder()
      , AdbTableNonQueryOperation.Truncate
      , CategoryTableName);


    new FileHelperFileSource<Category>(ws
      , "Read CSV"
      , () => truncate.IsSucceeded
      , ws.Config["CategoryFilename"])

    .Output.Link.AdbInsertTarget(
        "Insert"
      , cs.CreateConnectionBuilder()
      , CategoryTableName);
  })
  .Start() // Run the worker system
  .Exit(); // Exit with succeed/failure exit code
```



This next example demonstrates the ease of creating a new custom and *reusable* worker; simply derive from one of the many existing workers, depending on which pre-created functionality is desired. Here we derive from the simplest one, `WorkerBase`, and log an extra message. You could for instance instead add the three workers from the previous example, to create a reusable `TruncateInsertCsvWorker`.

```
public class HelloWorldWorker : WorkerBase
{
  public HelloWorldWorker(WorkerParent workerParent, string workerName)
    : base(workerParent, workerName)
  { }

  protected override Task<OutcomeStatus> RunAsync()
  {
    Logger.Info(ALogCategory.ProgressWorker, "Hello world");
    return OutcomeStatus.SucceededTask;
  }
}
```

The custom worker is added to the worker system in the same way as an out-of-box worker:

```
new WorkerSystem()
.Root(ws =>
{
  new HelloWorldWorker(ws, "Hello");
})
.Start()
.ThrowOnFailure();
```

See *Appendix - actionETL Features* for more details.

Turning Weaknesses into Strengths

Traditional ETL tools commonly perform poorly on most of the below factors, and this is likely the reason complex ETL work tends to be so time consuming and expensive. The critical *underlying* reason is very likely the big focus on creating visual tools. While visually displaying dependencies, flows etc. is certainly valuable, especially when starting out with ETL, for complex projects this approach has a *huge* cost, and is a *very* poor trade-off.

As a representative example, SSIS is unfortunately weak on *all* the below factors.

In contrast, **actionETL** is *strong* on all these factors, which is accomplished by applying the tools and techniques of modern *application* development, as detailed below. This brings a large productivity advantage, which becomes the driver for using it to *replace* or *augment* other ETL tools.

ETL specific	
SSIS	***actionETL***
Control flow - Dataflow separation	
Dataflow separate from Control flow • Dataflow components run inside a single task, and can't be intermixed with other tasks, nor have constraints set to or from them • Increases run times by waiting for whole dataflow to finish before starting dependent tasks • Increased staging, coding complexity and performance overhead • Reduces encapsulation	**Combine Dataflow and Control flow** • Run dataflow and control flow workers as siblings under the same parent • Set constraints between them • Decreases run times by starting work at the earliest possible time • Reduces staging, performance overhead and coding complexity • Improves encapsulation
Data source specific data types	
Supports SQL Server specific types For other data sources, lack of support leads to potential conversion failures, loss of precision or range. Importing/converting/exporting via text is error prone and extra work.	**Excellent support** • Data sources add their specific types (`MySqlDateTime`, `SqlGuid`, …) • Supported end to end in dataflow, database parameters etc. • Avoids conversion issues and conversion coding when source and target are the same type of data source
Micro batches	
Supported Somewhat higher overhead: • When using small dataflow buffer sizes • Validating whole packages even when only small parts of them will execute	**Good support** Very low overhead, even with small dataflow buffer sizes and large number of workers.
Separate (un)recoverable errors	
Not supported Failure constraints can hide critical bugs, since recoverable and unrecoverable errors use the same result code.	**Excellent support** • Fatal errors (e.g. unexpected exceptions) always fail the ETL worker system • Non-fatal errors (e.g. expected exceptions, expected row errors etc.) can optionally be ignored and/or acted upon with worker constraints

Non-ETL specific	
SSIS	***actionETL***
Main programming model	
Visual programming • Good for simple requirements but scales poorly with complexity • Thousands of mouse clicks through dialogs, tabs, editors and properties; slow to work with, hard to get an overview or enforce coding standards • Big UI focus makes file formats, APIs etc. more complex • Less easily integrated from other systems	**Just .NET code** • Visual Studio, Visual Studio Code etc. world class development environments • API designed for brevity and for showing the ETL hierarchy in the code • Scales very well with complexity • Easily integrated from other systems
Number of programming models and languages	
Many disjoint programming models and languages • Visual, expression language, .NET (with different object models in task and component scripts), C++ custom task/component/log provider • Varying degrees of integration between the different programming models and languages	**Single .NET programming model** .NET language (e.g. C# or VB.NET) for all aspects of the library, including constraints, custom workers etc.
Reusability	
Poor reusability • Can’t reuse dataflows with different row schemas • Can't reuse row schemas with different dataflows • Can’t easily reuse script tasks and script components • Constant copy-paste or recreate from scratch • Can use templates, but they equate to copy-paste. Changing the template doesn’t change all the instances created by the template. There’s no inheritance of functionality, and you can’t specify an interface to ensure all instances stay aligned. It becomes very time consuming and error prone keeping duplicated functionality aligned.	**Excellent reusability** • Combine reusable column schemas (i.e. column groups) into (reusable) row schemas • Reuse row schemas with different workers and dataflows • Allows users to use methods, lambdas, generics, interfaces, inheritance, dependency injection and great composability to achieve excellent reusability

Non-ETL specific	
SSIS	***actionETL***
Composability	

Poor composability means new functionality must usually be implemented from scratch.
• Can't compose dataflow components (or tasks, or containers, or packages) into a new reusable dataflow component
• Can't compose tasks (or containers, or packages) into a new reusable task or container
• Can compose tasks into reusable package, but:
  ○ Row schemas of any composed dataflow is hard-coded, severely limiting their usefulness
  ○ Has a fairly high coding overhead. Requires creating and configuring composed package, defining parameters, adding and configuring "Execute Package Task" to invoke package.
	Excellent composability
	• Dataflow and control flow workers can be composed and mixed into new reusable dataflow and control flow workers
	• Generics allow composed workers to support arbitrary row schemas, parameter types etc.

Encapsulation	

Somewhat poor encapsulation leads to error-prone and hard to change code.
• Global variables - all child containers and tasks can access all ancestor variables
• Package wide connection managers
	Good encapsulation
	• Minimized shared and global data:
	  ○ Workers use dependency injection for mandatory dependencies
	  ○ Interweave control flow and dataflow to minimize staging and global data
	• Excellent composability hides internal implementation details

Testability	
Poor testability Poor reuse and poor encapsulation makes unit testing very hard, which makes implementations error-prone and costly to modify.	**Excellent testability** • Excellent reusability • Good encapsulation • Excellent .NET testing frameworks available (xUnit, nUnit, MSTest, …)
Refactoring	
Limited refactoring Refactoring facilities don't handle custom code or work across the different programming models and languages, making them unreliable.	**Excellent refactoring** • Single .NET programming model • Static typing for dataflow rows and columns, and data source specific types • Extensive Visual Studio refactoring facilities and analyzer code fixes

Non-ETL specific	
SSIS	***actionETL***
Extensibility	
Fairly poor • Adding UI to custom tasks and components heavily increases effort • C++ skills are very different from Visual programming and ETL skills - very few ETL developers are comfortable creating custom C++ tasks or components, severely limiting extensibility • Can't extend or reuse existing tasks or components in new custom tasks or components	**Excellent support** • Very easy to derive new workers and worker systems by extending existing ones • Can compose existing workers in new custom workers • Replaceable logging system and configuration system • High performance dataflow column comparing, copying and mapping facilities • Optionally extend many workers with code snippets (lambdas)
Source control and Continuous Integration/Continuous Delivery	
Poor support • DTSX package file format mixes UI settings and even encoded binaries with business logic code, difficult to track and merge changes • Poor testability hurts CI/CD • Originally designed for manual deployment, somewhat difficult to automate	**Excellent support** • .NET source code • No ETL specific UI to cater for • Wide range of .NET compatible source control and CI/CD solutions

Conclusion

Traditional ETL tools commonly have a large number of architecture weaknesses that significantly limit their productivity on complex ETL projects, leading to higher effort and cost, and lower quality.

The key innovation of **actionETL** is that it combines familiar and effective ETL concepts such as control flow and dataflow capabilities, with the tools, techniques, and engineering processes of modern application development, thereby addressing all thirteen identified common ETL tool weaknesses.

The result is a highly productive ETL environment, which is uniquely adept at minimizing ETL effort and cost, as well as maximizing quality, even as project requirements and complexity grows. Depending on required features such as supporting particular data sources etc., it can replace or augment both traditional ETL tools, as well as ETL written in pure SQL, pure .NET, Python code etc.

To find out more, please see our web site and the full documentation at: https://envobi.com

References

1. SSIS documentation:

https://docs.microsoft.com/en-us/sql/integration-services/sql-server-integration-services
2. W. H. Inmon. Building the Data Warehouse. Wiley, New York, NY, USA, 3rd edition, 2002
3. SSIS DimensionMergeSCD: https://archive.codeplex.com/?p=dimensionmergescd
4. **actionETL** documentation: https://docs.envobi.com
5. **actionETL** SCD custom worker example:
https://docs.envobi.com/articles/slowly-changing-dimension-example.html

Appendix - **actionETL** Features

Platforms

- Cross-platform on **Linux** and **Windows**
- **.NET Framework 4.6.1+**, **.NET Standard 2.0+**, **.NET Core 2.1+**, and **.NET 5**

High Performance

- Combine in-memory row-by-row processing, database**,** and file-based processing to get the best of each approach
- Unlimited and configurable parallelism with low overhead, supporting millions of simultaneous or short-duration workers, including *micro-batches*
- Row and column high-performance **mapping**, **copying**, and **comparing** facilities for out-of-box and custom workers
- Row and column high-performance **copying**, **comparing** and mapping facilities for out-of-box and custom workers
- Statically typed, high throughput dataflow with highly tuned workers
- Batch Insert for all databases and Bulk Insert for selected databases

Broad Data Source Support

- Wraps and extends .NET database providers to simplify writing *database agnostic* code
- Dedicated MariaDB™, MySQL™, PostgreSQL®, SQLite and SQL Server® providers
- ODBC provider for other databases
- Local transactions across *multiple* workers
- Supports many database specific data types (`SqlDateTime`, `NpgsqlDate`, ...) end to end, avoiding conversion issues and reducing coding
- Read and write Delimited and Fixed Format flat files, streams and strings
- Read and write XLSX spreadsheets, JSON, and text files
- Read .NET IEnumerable, and read and write Collections
- Easily extensible to other formats as well as transfer protocols like SFTP, SCP, FTP, etc.

Familiar ETL Concepts

- Control flow with *start constraints*, *grouping*, *hierarchical structure* and *highly parallel* applications
- 50 dataflow workers provide extensive *row-by-row* processing capabilities
- Read and write data sources, cleanse, combine and transform data
- Divide and conquer - implement requirements via many smaller (and *reusable*) parts
- Effective debugging with breakpoints on worker and port state changes, and inspecting and editing *rows in flight*
- Highly capable (as well as replaceable) logging and configuration systems
- Automatically track and aggregate dataflow error row counts and log error row contents
- Common tasks require very little programming, more akin to setting a *configuration*
- Familiar concepts, project templates, a concise API and extensive documentation makes it *easy to learn*

Modern Application Development

- NuGet.org packages and project templates provide simple integration and updates
- Compose and encapsulate existing *workers* into new (control flow and dataflow) **reusable** workers

- Highly flexible workers that optionally accept code snippets (lambdas), e.g. to perform a *greater-than* join instead of the default *equi-join*, or specifying *ordering columns,* etc.
- **.NET code-based programming** and the **actionETL** architecture handles complexity very well
- Perform **ETL testing** with any .NET test framework
- Take full advantage of **Visual Studio®** or other .NET development environments for **refactoring**, **source control**, ***CI/CD,*** *etc.*

Unique Strengths

- Merged Dataflow and Control flow functionality (including constraints), which *reduces code complexity and data staging*
- *Reuse* and *combine* column schemas (groups of columns) to minimize dataflow bugs, maintenance effort and code size
- **Single programming model** for both **using** and **extending** the library, including for constraints, custom workers etc., simplifying creating reusable custom functionality
- Distinct recoverable vs. unrecoverable failures, minimizing start constraint bugs

Appendix – **actionETL** Workers

Note that database workers are prefixed with “Adb”.

Control Flow Workers	
ActionWorker	ExecuteProcessWorker
ActionWorkerBase	FileExistsWorker
AdbExecuteNonQueryWorker	ForEachActionWorker
AdbExecuteScalarWorker	MoveFileWorker
AdbTableNonQueryWorker	UsingActionWorker
AdbTransactionActionWorker	WhileActionWorker
CopyFileWorker	Worker
CreateFileWorker	WorkerBase
DeleteFileWorker	

Dataflow Source Workers	
ActionSource	RepeatRowsSource
AdbDataReaderSource	RowsActionSource
EnumerableSource	RowSourceBase
FileHelperFileSource	RowsSourceBase
FileHelperStreamSource	SourceBase
FileHelperStringSource	XlsxSource
PortPassThroughSource	

Dataflow Transform Workers	
ActionTransform	MulticastTransform
ActionTwoInputTransform	RightJoinMergeSortedTransform
AggregateTransform	RowActionTransform
CrossJoinTransform	RowsActionTransform
DictionaryLookupTransform	RowsTransformBase
DictionaryLookupSplitTransform	RowTransformBase
FullJoinMergeSortedTransform	SortTransform
InnerJoinMergeSortedTransform	SplitTransform
LeftJoinMergeSortedTransform	TransformBase
MergeSortedTransform	UnionAllTransform

Dataflow Target Workers	
ActionTarget	PortPassThroughTarget
AdbExecuteNonQueryTarget	RowActionTarget
AdbInsertTarget	RowsActionTarget
AdbMySqlConnectorBulkInsertTarget	RowsTargetBase
AdbSqlClientBulkInsertTarget	RowTargetBase
CollectionTarget	RowWithErrorTargetBase
DictionaryTarget	TargetBase
FileHelperFileTarget	TrashTarget
FileHelperStreamTarget	XlsxTarget