Retail passion. Innovative solutions.

Retek® Extract Transform and Load™ 11.2.2

Best Practices Guide

Corporate Headquarters:

Retek Inc.
Retek on the Mall
950 Nicollet Mall
Minneapolis, MN 55403
USA

888.61.RETEK (toll free US)
Switchboard:
+1 612 587 5000

Fax:
+1 612 587 5100

European Headquarters:

Retek
110 Wigmore Street
London
W1U 3RW
United Kingdom

Switchboard:
+44 (0)20 7563 4600

Sales Enquiries:
+44 (0)20 7563 46 46

Fax:
+44 (0)20 7563 46 10

Customer Support

Customer Support hours

Customer Support is available 7x24x365 via email, phone, and Web access.

Depending on the Support option chosen by a particular client (Standard, Plus, or Premium), the times that certain services are delivered may be restricted. Severity 1 (Critical) issues are addressed on a 7x24 basis and receive continuous attention until resolved, for all clients on active maintenance. Retek customers on active maintenance agreements may contact a global Customer Support representative in accordance with contract terms in one of the following ways.

Contact Method	Contact Information
E-mail	support@retek.com
Internet (ROCS)	rocs.retek.com Retek's secure client Web site to update and view issues
Phone	+1 612 587 5800

Toll free alternatives are also available in various regions of the world:

Australia	+1 800 555 923 (AU-Telstra) or +1 800 000 562 (AU-Optus)
France	0800 90 91 66
Hong Kong	800 96 4262
Korea	00 308 13 1342
United Kingdom	0800 917 2863
United States	+1 800 61 RETEK or 800 617 3835

Mail

Retek Customer Support
Retek on the Mall
950 Nicollet Mall
Minneapolis, MN 55403

When contacting Customer Support, please provide:

- Product version and program/module name.
- Functional and technical description of the problem (include business impact).
- Detailed step-by-step instructions to recreate.
- Exact error message received.
- Screen shots of each step you take.

Contents

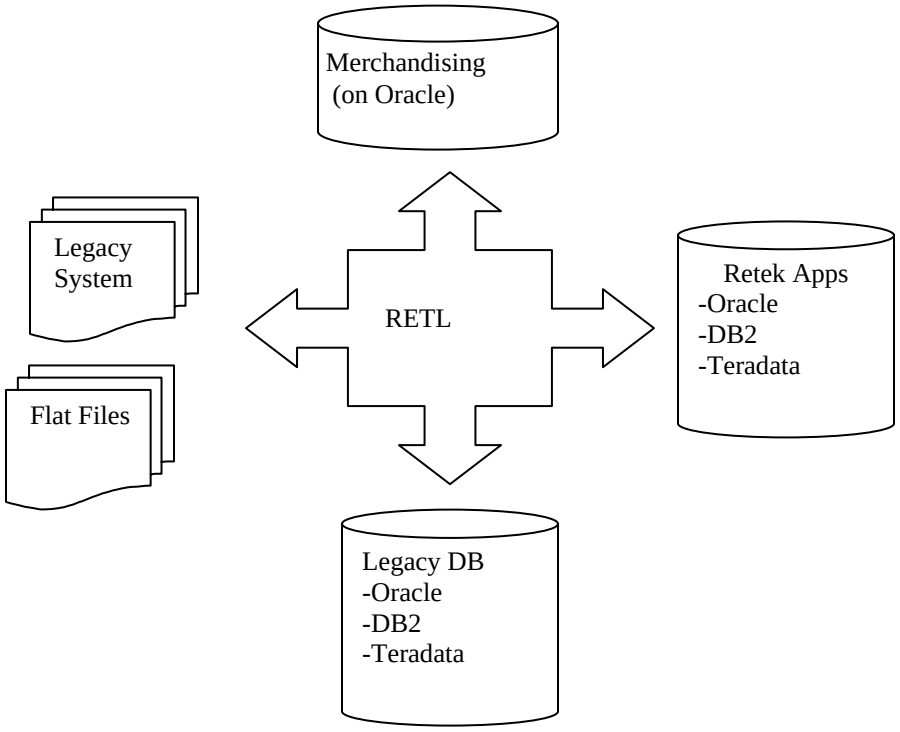
Introduction and Objectives

Traditional application integration has been done in a point-to-point manner. Developers are given an integration tool and told to integrate to a defined file/database specification or to an API. The end result may be functionally valid according to the single specification, but the means to get there, the implementation, may be cumbersome, non-reusable, non-performant, and subject to heavy maintenance costs. Worst yet, the design itself may not have accounted for all the business process needs, invalidating the entire implementation.

This is why there is a need for best practice guidance when using any tool – so that the tool can be used and the interface developed and deployed to the maximum advantage in terms of costs and benefits. This document will cover simple and proven practices that can be used when developing and designing integration flows using the Retek Extract, Transform, and Load (RETL) tool.

"A tool is only as good as you use it"

RETL is a simple tool doing a very simple job – moving large amounts data in bulk. However, managing the variety of disparate systems and pulling data together to form a complete 'picture' can be very complex. The diagram below shows some of the mediums that RETL among which RETLcan move data.



As a result of business process, integrated system, database and interface complexity, it is imperative that the RETL tool be used correctly so that the end deliverable of RETL flows is performant, maintainable, and of high quality.

This document will describe the following activities and what best practices - in terms of “do’s” and “don’ts” - should be followed before, during, and after each activity:

1 Project Initiation/Design/functional specification – The best practices to follow during requirements elicitation, interface analysis, and design.

2 Code/Implementation/Test – The best practices to follow when setting up the environment, during flow construction and all phases of testing. This will be the bulk of the document.

3 Review/product handoff – Activities to follow when reviewing the project and its deliverables. This also provides guidance for handing off interface deliverables to operations and support personnel.

This is a living document that will grow over time as we learn more about our customers needs and can fill it with tips from experts from the different domains that RETL touches upon.

Prerequisites

This document is written for a project manager looking to set up a RETL development group. This document is also written for a flow developer of any skill level, looking to work effectively with the RETL tool.

There are two core skill sets needed when designing and developing RETL integration flows:

1 Interface design skills – a functional/technical architect should design the interfaces. The designer should have knowledge of the following:

   a Understand the source and target data sources. They should have in-depth knowledge of the data models for each application, and how a transformation may be performed in order to integrate the two applications

   b Can understand how each application and business process works and can quickly interpret the data needs for an application based on the business process(es) that need to be supported.

   c Has a general understanding of the schedule, data dependencies, batch jobs, and volume of the application in question.

2 ETL technical skills – The ETL coder should have knowledge of the following:

   a Have strong Unix experience – in-depth Korn shell scripting and Unix in general are a must

   b Have previous RETL Experience – familiarity with the RETL Programmer’s Guide is a must and previous experience with writing RETL flows is strongly recommended

   c Have strong database Experience – familiarity with SQL statements and database operations and utilities (e.g. SQL*Loader) is strongly recommended.

Expectations

This document will not attempt to teach all the intricacies of flow development and design, and it will not show the developer how to develop flows in a step-by-step manner. A request to do this would be akin to 'how do I code my java program?' However, the reader may expect this document to provide benefit in the following ways:

- Best practices to follow when designing a flow
- Best practices to follow when developing a flow
- Common 'gotchas' and other pitfalls to avoid

If in depth RETL training is required, please contact Retek Learning Services for training courses and availability.

Chapter 1 – Project Initiation/Design/Functional Specification Best Practices

Before any development can be done, it is imperative to have a solid interface design that is generic and re-usable for future applications. The following practices will help ensure this:

DO – Ask discovery questions first

An important part of the functional design is to ask pointed and relevant questions that can answer the following:

Generic Integration Questions

- *What type of data integration is needed?* Is there a potential for needing real-time or near-real-time integration? If there is potentially a need for more synchronous/real-time integration, perhaps consider using a different technology such as the Retek Integration Bus (RIB) or direct access.
- *What business process is to be integrated?* For any project manager or flow developer to understand how they must construct the data flow, they must understand at a high-level, what business process it is that they will support. Why do the users in system x need the data created in system y? Without this understanding, RETL users may not get a complete picture of what it is they need to integrate, nor would they have the opportunity to ask questions that might reveal additional interface capabilities, variations, or options that are needed by the business user community.

Application Domain Questions

- *What are the targeted versions for each application involved?* This is important to establish as a basis for integration development against each product and the business features and functions available in a particular version.
- *What is the source and target for each module?* For example, database to file, file to file, file to database, etc
- *On what DBMS and version does the source and target application reside, if any?* If the source/target database isn't supported by RETL, then consider using a different technology or provide feedback to the RETL team to include that database support in a future release.
- *What types of transformations might be expected for each module?* This will affect the complexity of each module.
- *Are there any constraints on the source(s) and target(s)?* For example, does the target database table need to maintain indexes, etc? Are there any referential integrity issues to be aware of? There are implications when developing a flow against a transaction database or a table that needs to maintain indexes or /referential integrity.
- *What future applications might 're-use' these interfaces?* Designs should be generic and promote easy modification should any future applications need to re-use the same interfaces.

Data-related and Performance Questions

- *How much data is passing through each module?* High volume modules will need to be further scrutinized when coding and performance testing.
- *What is the size of the source and target database tables/files?* This is to raise any flags around performance early in the project.
- *What is the frequency in which each module will be run?* (e.g. nightly)
- *What is the time frame in which each module is expected to run in? For the entire set of modules?* This will need to be realistic as it will provide a baseline for performance
- *Is data going to need to be transferred over the network?* There will be a negative effect on performance if massive amounts of data will need to be transferred over the network.
- *Are there potentially any high volume transformations that could be easily done inside the database?* If there are, these transformations may be done inside the database to start with so as to eliminate rewrites later on. The idea here is to maximize and balance each technology to what it does best.
- See the "RETL Performance Tuning Guide" for more information on the topic of performance.

DO – Visually map out the movement of data

Use a tool to visually map out the movement of data from each source to each target. Use a tool such as Visio or even simply Word to diagram the source, transformation processing/staging, and target layers. (Use a database template or the 'basic flowchart' in Visio to map these layers out). This should serve as a basis for design discussions among application groups.

DO – Define concrete functional requirements for each module

As part of the design process, a logical description of how each module will extract, transform, and load its data should be completed. Again, all stakeholders should sign off on this.

DO – Define concrete functional designs for each module

As part of the design process, the following should be clearly defined:

- Source and Target locations (e.g. table name, filename, etc)
- Source and Target sizes/volumes
- Designs and metadata definitions for each input and output
- Mapping of the transformation process on how to get from input format to output format
- Name of the script/module

DO – Design a test plan early in the process

Designs aren't complete until the unit, system, integration and performance test plan designs are complete. This can be completed once the data and test environment needs are identified. Activities to complete for the test plan include the following

- Identification of the different combinations of data and the expected results
- Identification of degenerate cases and how errors should be handled
- Identification of how many and what type of test cases that need to be built
- Identification of configuration management and environment migration procedures to ensure that quality control individuals know what code set it is that they are testing.
- Identification of test environment needs to ensure that unit, system, integration and performance test can be done without negatively impacting each other.
- Prioritization of relative importance of execution for each test case based on most likely scenarios, highest risk code, etc.
- Determine a 'go/no-go' threshold for which tests must pass in order to release.

DO – Design for future usage and minimize impact of potential changes

The design should take into account future applications that may need to live off the same interfaces. Relevant questions to ask are the following: What is the superset of data that an application needs or may need in the future? How can this flow be designed so that any future changes will not break the interface specification?

DO – Agree on acceptance criteria

The overall project should set requirements based on a number of acceptance criteria set by the customer. A partial list of these may involve the following:

- *Functional criteria* – the inputs, outputs, and expected functionality of each module
- *Test criteria* – the types and how intensive must the testing be, and the threshold in acceptance of defects.
- *Performance criteria* –the performance requirements for each module, and the overall batch window in which the entire suite of modules must run in.
- *Documentation* – Identify clearly the end deliverables needed to hand off the interface deliverable to operations and support staff. A well-constructed interface cannot be used and maintained successfully without documentation or communication deliverables.

DON'T – Leave assumptions/issues/risks undocumented

Any design assumptions and issues/risks that have been purposely un-addressed should be documented in an 'Assumptions/Issues/Risks' section of the design documentation deliverables.

Chapter 2 – Code/Implementation/Test Best Practices

This section will break best practices to follow into three functional areas:

1 Korn shell module best practices – the .ksh module containing the RETL flow

2 RETL flow best practices – the XML flow that RETL runs

3 Database best practices

4 Testing best practices

Following this core set of best practices when developing and testing flow modules will provide the basis for high quality, maintainable, performant, and re-usable flows.

Korn Shell Best Practices

DO – Execute commands using $(command) and not `command`

This reduces the likelihood of typographical errors when entering backspaces versus single quotes (` vs. ‘) and promotes better readability.

DO – Ensure ‘set –f’ is set in a configuration file

‘set –f’ disables filename generation. This should be set in the application’s config file (e.g. rdw_config.env). If filename generation isn’t turned off, a flow that contains a ‘*’ in it (for example, a flow that contains a query ‘select * from <table>’) may end up causing the module to incorrectly expand * to include filenames and incorrectly pass these to RETL. This is because many modules may pass RETL a flow through standard input. See best practice “Write flow to intermediate file and then call RETL on that file”

DO – Write flow to an intermediate file and then call RETL on that file

DON’T – Send the flow directly to RETL via standard input

It is recommended to ‘cat’ or write to a flow file first before calling RETL, rather than calling RETL and passing a flow through standard input. This aids in debugging, helps support when it is necessary to recreate an issue, and also prevents various nuances that may exist when sending the flow to RETL via standard input. However, care should also be taken to protect source directories since these generated .xml flow files may contain sensitive information such as database login information. See the “Secure/Protect files and directoriesthat may contain sensitive information” best practice for more on the topic of security.

Good:

```
cat > ${PROGRAM_NAME}.xml << EOF
…
EOF
${RFX_EXE} ${RFX_OPTIONS} -f ${PROGRAM_NAME}.xml
```

Bad:

```
$RFX_HOME/bin/$RFX_EXE -f - << EOF
…
EOF
```

DO – Secure/Protect files and directories that may contain sensitive information

Any directories that may contain sensitive information such as database logins and passwords should be protected by the proper UNIX permissions. Namely, any temporary directories that RETL uses, any source directories in which flows may be written to, and any configuration files and directories should have strict permissions on them. Only the userid that runs the flows should have access to these files and directories.

DO – Make often-used portions of the module parameters or functions

DON'T – Overuse shell functions

Use variables for portions of code that can be re-used (e.g. ${DBWRITE}), or create entire shell script functions for methods that can be re-used. (e.g. simple_extract "${QUERY}" "${OUTPUT_FILE}"). This can be taken to the extreme however, when functions call functions, which call functions, ad nauseum. In general, function calls should go only a few layers deep at most, and only more when it makes sense from an abstraction and overall maintainability point of view.

DO – Separate environment data from the flow

Variables that can change (for example, database login information/etc) should go in a configuration file (e.g. rdw_config.env). The idea here is to separate the configurable environment data from the generic flow, which lends to more maintainable and configurable code base.

DO – Enclose function parameters in double quotes

When calling a function in ksh, care should be taken to place all arguments in double quotes. For example,

```
do_delete "${TABLE1}" "${TABLE2}" "${JOIN_KEYS}"
```

If function parameters aren't placed in quotes, there is the potential for misalignment of positions for variables and the function will likely not operate properly, if at all.

DO – Set environment variable literals in double quotes

When setting an environment variable, always use double quotes. For example,

```
export TARGET_TABLE="INV_ITEM_LD_DM"
```

If environment variable values aren't placed in quotes, there is the potential for losing the portion of the value after a space. or having the shell interpret it as a command.

DO – Use environment variables as ${VARIABLE} rather than $VARIABLE

Environment variables should be used as ${VARIABLE} so as to promote readability, especially when they exist in the context of other non-variables and text.

DO – Follow module naming conventions

There are a number of naming conventions to follow, and in general, this should be the standard for consistency among applications:

1 RETL modules/scripts (.ksh) should generally conform to the following convention:

<application name><script type>_<program name>.ksh,

where

<application name> is the abbreviated application name,

<script type> is ‘e’ for an extract program, ‘t’ for a transform program, or ‘l’ for a load program. A program that contains all of the above will not have a script type associated with it,

<program name> is the name of the program.

For example:

rmse_daily_sales.ksh (<rms><e>_<daily_sales>.ksh)

2 The program name in general doesn’t have strict naming conventions, but it should be representative of what the program actually does. Each word should be separated by an underscore (_).

3 Custom modules/flows should be prefixed by the short client name.

4 Application log files should follow the format:

<application name><date>.log

5 Error files should follow the format:

<application name>.<module name>.<unique id>.<date>

where

<application name> is the abbreviated application,

<module name> is name of the module,

<unique id> is optional. If multiple copies of the module can be run concurrently, this should be a reasonable unique identifier. For example, it might be the input filename, thread #, etc.

For example:

rdw.slsildmdm.sls_100_1.txt.20020401

(<rdw>.<slsildmdm>.<datafile #>.<date>)

DO – Log relevant events in module processing

As a general rule, it is better to log too much verbosity than too little. The following should be completed:

1 Log the start and finish times for any significant processing events that occur in a module.

2 Log the number of records in the input and output files.

3 Redirect standard output and standard error to the log/error file. For example:

```
exec 1>>$ERR_FILE 2>&1
```

DO – Place relevant log files in well-known directories

Each application log file should be placed in a well-known directory for consistency and ease of accessibility. For example, RETL performance log files (as specified in rfx.conf) are placed in temporary directories by default. rfx.conf should be changed so that RETL performance log files places these log files in well-known directories rather than in temporary directories.

DO – Use .ksh templates

Insert template on a standard flow (e.g. library sourcing first, then variable definitions, function initionsdefs, etc)

DO – Document each flow's behavior

Each flow should be documented as to its required inputs, required outputs, and behavior in between.

RETL Flow Best Practices

DO – Parameterize the call to and any options passed to the RETL binary

Use the ${RETL_EXE} environment variable to replace the call to the rfx executable and use ${RETL_OPTIONS} to replace the command line option '-f -'. "rfx -f - " should not be used to call RETL – it should be "${RETL_EXE} ${RETL_OPTIONS}". $RETL_EXE and $RETL_OPTIONS should be defined in the configuration file (e.g. rdw_config.env). The default call in the configuration file should be rfx – f –

DO – Perform all field modifications as soon as possible in the flow

Field modifications such as renaming and type conversion should be performed as soon as possible in the flow to aid in tracking a field through the flow.

DO – Use care when choosing and managing your temp space

RETL temp space should be spread out on multiple disks and, as part of a normal maintenance schedule, should be cleaned periodically as RETL may leave miscellaneous files in temp space for tracking and debugging purposes.

DO – Turn on debug code when in development

DON'T – Run with debug code when in production

RETL will print out extra messages in debug mode. The following environment variables can be set to produce extra verbosity:

```
export RFX_DEBUG=1

export RFX_SHOW_SQL=1
```

Additionally, there are certain warnings that may be printed to the screen in this mode that will not be printed to the screen when verbose debugging is turned off.

Do not run with these options on in production as they produce additional files and may leave around temporary/intermediate files.

DO – Make use of RETL visual graphs

In RETL 10.x versions and 11.2+, RETL offers a graphing option (--graphviz-files in 10.x or –g in 11.2+). This produces a visual graph of the flow and shows how each operator is connected inside the flow. See the RETL Programmer's Guide for more information and command-line usage.

DO – Delineate input records from a file by a newline ('\n')

Although RETL doesn't mandate having a newline record delimiter, it is recommended to use this for debugging and supportability purposes. However, it is also important to make sure that input data cannot contain the newline character. See "DON'T – Choose a delimiter that can be part of your data" for more information.

DO – Run RETL with the '-s SCHEMAFILE' option when in development

The '-s SCHEMAFILE' command-line argument will tell RETL to print input and output schemas of each operator in a schemafile format that can be used in an import or export operator. This can be very useful when debugging and 'breaking up' flows into smaller portions. For example, the call to RETL would be as follows:

```
rfx –f theFlow.xml –s SCHEMAFILE
```

DO – Use the latest version of RETL available

Future versions of RETL are always improving in error handling, debugging, logging, and performance. It is recommended to use the latest version of RETL that has been certified with the product being used.

DO – Develop the RETL flow first, outside of the shell module

It is easier to develop the RETL flow outside the Korn shell module (.ksh) first, and then when a stable cut of the flow has been produced, the flow can be parameterized and retrofitted back into the Korn shell module (.ksh)

DO – specify inputs, properties, and outputs, in that order

In order to conceptually map out the flow of data from one operator to another, place the

<INPUT(s) first, the >

<PROPERTY>

<PROPERTY>

<OUTPUT(s) last, and the properties in between:>

For example:

```
<OPERATOR type=″changecapture″>
 name=″before_dataset.v″/
<INPUT name=″after_dataset.v″/> name=″key″ value=″CMPY_IDNT″/
name=″value″ value=″CMPY_DESC″/ name=:″changes.v″/</OPERATOR>
```

DO – Group business logic together with nested operators

DO – Keep connected operators in close physical proximity to each other in a flow

DON'T – Nest operators more than a few layers deep

When multiple operators are used to perform a logical step in the flow, nest the operators together.

In the example below, the operators split the after_dataset.v dataset into separate datasets for inserts, deletes, and edits. The switch operator is embedded within the changecapture.

```
<!—Split up after_dataset.v into inserts, deletes, and edits. -->
<OPERATOR type="changecapture" >
<INPUT name = "before_dataset.v"/>
<INPUT name = "after_dataset.v"/>
<PROPERTY name="codefield" value="change_code" />
<PROPERTY name = "key" value = "CMPY_IDNT"/>
<PROPERTY name = "value" value = "CMPY_DESC"/>
<OPERATOR type="switch">
<PROPERTY  name="switchfield" value="change_code"/>
<PROPERTY  name="casevalues" value="1=0, 2=1, 3=2"/>
<OUTPUT  name="inserts.v"/>
<OUTPUT  name="deletes.v"/>
<OUTPUT  name="edits.v"/>
</OPERATOR>
</OPERATOR>
```

As the operator nesting gets deeper, the flow becomes unreadable, so don't net operators more than a few layers deep.

DO – Document fixed schema file positions

In order to debug flows and to aid in documenting interface fixed-length files, fixed-length schema files should include positions as a comment before each field definition. However, this also adds additional overhead to schema file maintenance, and could prompt for typographical errors. This should be taken into account when making a decision to document a module's schema files.

Example:

```
<RECORD type="fixed" len="9" final_delimiter="0x0A">
<!-- start pos 1  --> <FIELD name="FIELD1" len="4" datatype="int16"
…/>
<!-- start pos 5  --> <FIELD name="FIELD2" len="4" datatype="int16"
…/>
<!-- end pos 9  -->
</RECORD>
```

DO – Use valid XML syntax

10.x versions of RETL allowed users to enter invalid XML syntax. 11.x versions are more strict about valid XML syntax, and will throw errors if syntax is invalid. In particular, when calling RETL's filter operator, utilize 'GT' instead of '>', 'LT' instead of '<', 'LE' instead of '<=', 'GE' instead of ">=", 'NOT … EQ …' instead of '<>' and 'EQ' instead of '='. Also see the best practice "Wrap dbread 'query' property in a CDATA element'.

DO – Wrap dbread 'query' properties in a CDATA element

This is important to make sure flows are using valid XML syntax. In general, ALL 'query' properties of the dbread operators should be wrapped in a CDATA element. This tells the XML parser to treat the enclosed text as a single chunk of text rather than to parse it as XML. If this rule is not followed, RETL may not run (see best practice 'Use valid XML syntax' for more info).

Bad:

```
${DBREAD}
<PROPERTY name="query" value="SELECT * FROM TABLE WHERE COL1 >
COL2"/>
...
```

Good:

```
${DBREAD}
<PROPERTY name="query">
<![CDATA[
SELECT * FROM TABLE WHERE COL1 > COL2
]]>
</PROPERTY>
```

DO – Write flows to be insulated from changes to a database table

In general, flows should be written so that any changes to database tables don't affect the functionality of the flow. For example, the FIELDMOD operator can be used with the "keep" property to keep only fields that are needed in the flow.

DO – Avoid implicit schema changes

Certain operators that take more than one input (such as funnel) can implicitly change schemas for a field's nullability, max length, etc. Any desired schema changes should be explicitly made with the CONVERT operator otherwise undefined behavior may result.

DO – Test Often

Testing often minimizes integration issues, increases quality, aids in easier defect diagnosis, and improves morale.

DO – Follow the RETL Performance/Tuning document

Performance deserves its own document, and the RETL Performance/Tuning Guide should be followed to this extent. In general, there are a few practices to follow as a guideline:

- Achieve the optimal balance between the tools, hardware, and database
- Use the database for what it does best – significant data handling operations such as sorts, inter-table joins, and groupby's should be handled inside the database when possible
- Use RETL for what it does best – non-heterogeneous (e.g. file, different db vendor) joins, data integration and movement from one platform to another, multiple source locations, etc.
- Performance test and profile early and continuously in the development cycle. Don't delay this activity until the end of the project.
- Don't performance tune a flow before the flow functions according to design.
- Design a flow to make full use of parallelism. RETL internal parallelism coupled with standard Retek 'multi-threading', or running multiple RETL processes concurrently, can maximize the use of parallelism.
- Develop for clarity, maintenance, and re-use, but keep in mind performance at all times. Other factors may be sacrificed if performance is of the utmost importance.

DO – Name flows appropriately in the <FLOW> element

Flows should include a 'name' attribute in the FLOW element of the XML. This eases debugging in RETL's logging mechanism. In general, the flow should be named the same as the module in which the flow is run. For example,

```
<FLOW name="modulename.flw">
...
</FLOW>
```

DO – Use SQL MERGE statements when doing updates

As of this writing, RETL does not have update capabilities built in. The recommended approach when doing updates is to use the Oracle MERGE statement. This statement is akin to a delete followed by insert for each updated record, although MERGE is much faster. The flow's processing would be as follows: insert records to a temp table, and then MERGE into the target table.

DO – Document complex portions of code

As with general programming best practices, any complex portions of code that could be misunderstood should be thoroughly documented. This aids in ease of maintenance and helps future personnel to understand the original intent of the code.

DON'T – Define fields that have white spaces

Many RETL operator properties allow specification of a list of fields. White space within a field name will cause RETL to misinterpret the field list.

DON'T – Choose a delimiter that can be part of the data

If a data can even possibly contain a delimiter, a different delimiter should be chosen. Even to risk being redundant, it should be repeated – NEVER EVER choose a delimiter that can be part of the data. For the most part, pipe-delimiters ('|') and semi-colon delimiters (';') should work. If no reasonable delimiter can be found, then a fixed format file should be used.

Chapter 3 – Review/Product Handoff

A final product review and handoff to support is necessary to complete a successful RETL project.

DO – Involve support personnel early in the project

This will help the learning curve and make the impact less dramatic when the project must be transitioned to support.

Identify clearly the end deliverables needed to hand off the interface deliverable to operations and support staff. A well-constructed interface cannot be used and maintained successfully without documentation or communication deliverables.

DO – Assign a long-term owner to the project/product/interface

Assigning ownership will allow the product to mature and develop properly and it will reduce internal contention about areas of responsibility when multiple applications/domains are involved.

Assigning ownership of the interface following handoff will also ensure that appropriate operations and support personnel are involved and trained throughout the life of the development effort.