Pipeline Design and Performance Optimization Guide

Introduction

This document serves as a comprehensive best practice guide for developing efficient and robust Pipelines within the SnapLogic Platform.

It offers guidelines that aim to optimize performance, enhance maintainability, reusability, and provide a basis for understanding common integration scenarios and how best to approach them. The best practices encompass various aspects of Pipeline design, including Pipeline behavior, performance optimization and governance guidelines.

By adhering to these best practices, SnapLogic developers can create high-quality Pipelines that yield optimal results while promoting maintainability and reuse.

The content within this document is intended for the SnapLogic **Developer Community** or an **Architect**, in addition to any individuals who may have an influence on the design, development or deployment of Pipelines within the SnapLogic platform.

Authors: SnapLogic Enterprise Architecture team

Why good Pipeline Design is important

The SnapLogic Pipeline serves as the foundation for orchestrating data across business systems, both within and outside of an organization. One of its key benefits is its flexibility and the broad range of "Snaps" that aim to reduce the complexity involved in performing specific technical operations. The “SnapLogic Designer”, a graphical low-code environment for building an integration use case with Snaps, provides a canvas enabling users with little technical knowledge to construct integration Pipelines. As with any user-driven environment, users must exercise careful attention to ensure they not only achieve their desired business goals but also adhere to the right approach that aligns with industry and platform best practices. When dealing with a SnapLogic Pipeline, these best practices may encompass various considerations:

- *Is my Pipeline optimized to perform efficiently?*
- *Will the Pipeline scale effectively when there's an increase in data demand or volume?*
- *If another developer were to review the Pipeline, would they easily comprehend its functionality and intended outcome?*
- *Does my Pipeline conform to my company's internal conventions and best practices?*

Not considering these factors may cause undesirable consequences for the business and users concerned. Relative to the considerations stated above, these consequences could be as follows:

- If data is not delivered to the target system, there may be financial consequences for the business.
- The business may experience data loss or inconsistency when unexpected demand occurs.
- Development and project teams are impacted if they are unable to deliver projects in a timely fashion.
- Lack of internal standardization limits a company's ability to govern usage across the whole business, thus making them less agile.

Therefore, it is essential that users of the Platform consider best practice recommendations and also contemplate how they can adopt and govern the process to ensure successful business outcomes.

Understanding Pipeline Behaviour

To better understand how Pipelines can be built effectively within SnapLogic, it is essential to have an understanding of the Pipeline's internal characteristics and behaviors. This section aims to provide foundational knowledge about the internal behavior of Pipelines, enabling you to develop a solid understanding of how they operate and help influence better design decisions.

Pipeline Execution States

The execution of a SnapLogic Pipeline can be initiated either via a Triggered, Ultra or Scheduled task. In each case, the Pipeline transitions through a number of different 'states' with each state reflecting a distinct processing the lifecycle of the Pipeline, from invocation, preparation, execution to completion. The following section of the document will look to highlight this process in more detail and explain some of the internal behaviors.

The typical Pipeline execution flow is as follows:

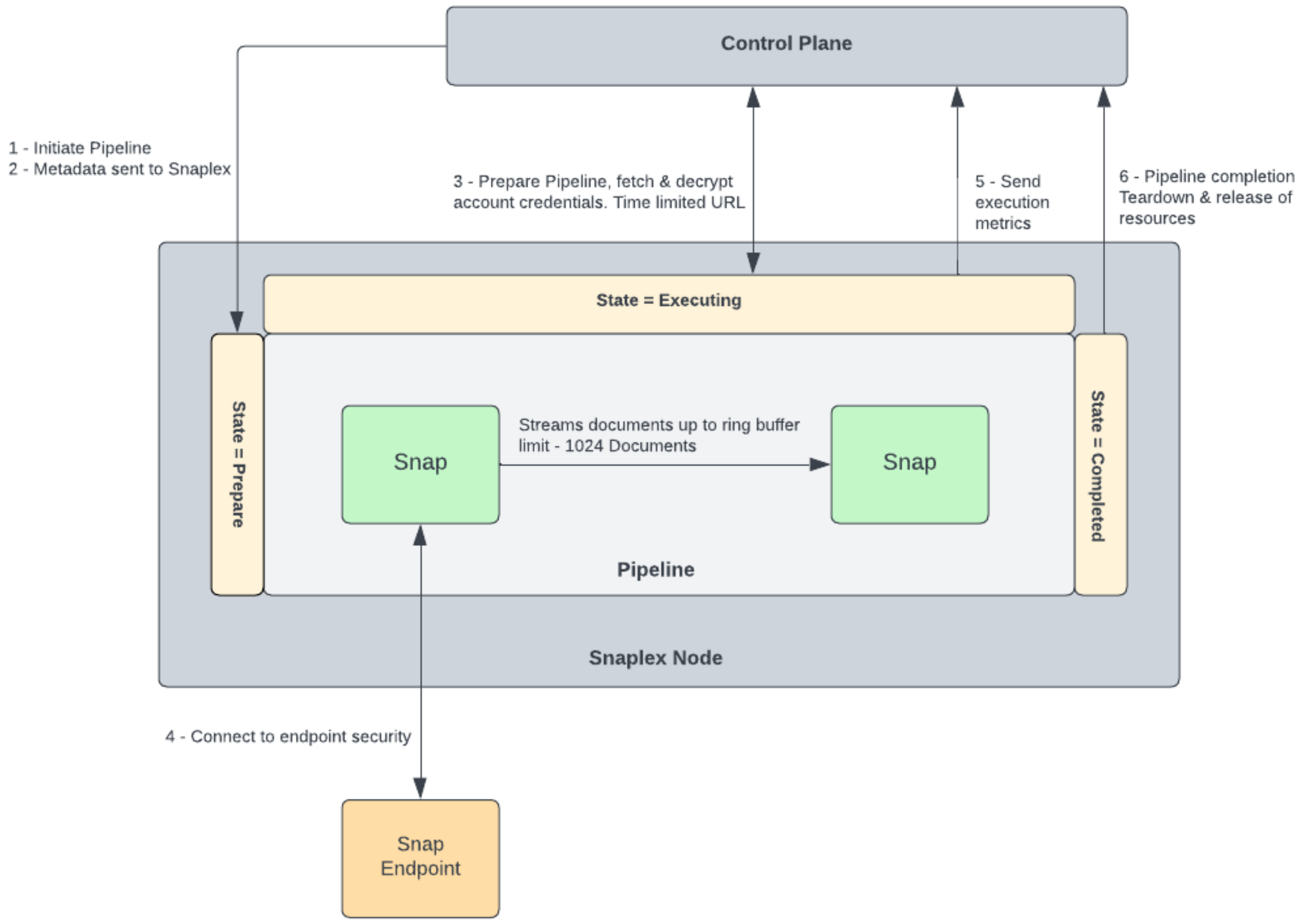
1. Initialize Pipeline.
2. Send Metadata to Snaplex.
3. Prepare Pipeline, fetch & decrypt account credentials.
4. Connect to endpoint security.
5. Send execution metrics.
6. Pipeline completes, and resources are released.

The following section describes the different Pipeline state transitions & respective behavior in sequential order.

State	Purpose
NoUpdate	A pre-preparing state. This indicates a request to invoke a Pipeline has been received but the leader node or control plane is trying to establish which Snaplex node it should run on. (This state is only relevant if the Pipeline is executed on the leader node).
Preparing	Indicates the retrieval of relevant asset metadata including dependencies from the control plane relating to the invoked Pipeline. This process also carries out pre-validation of snap configuration alerting the user of any missing mandatory snap attributes.
Prepared	Pipeline is prepared and is ready to be executed

Executing	Pipeline executes and processes data, connecting to any Snap Endpoints using the specified protocols.
Completed	Pipeline execution is complete and the teardown resulting in the releasing of compute resources within the Snaplex Node. Final Pipeline execution metrics and sent to the Control Plane.

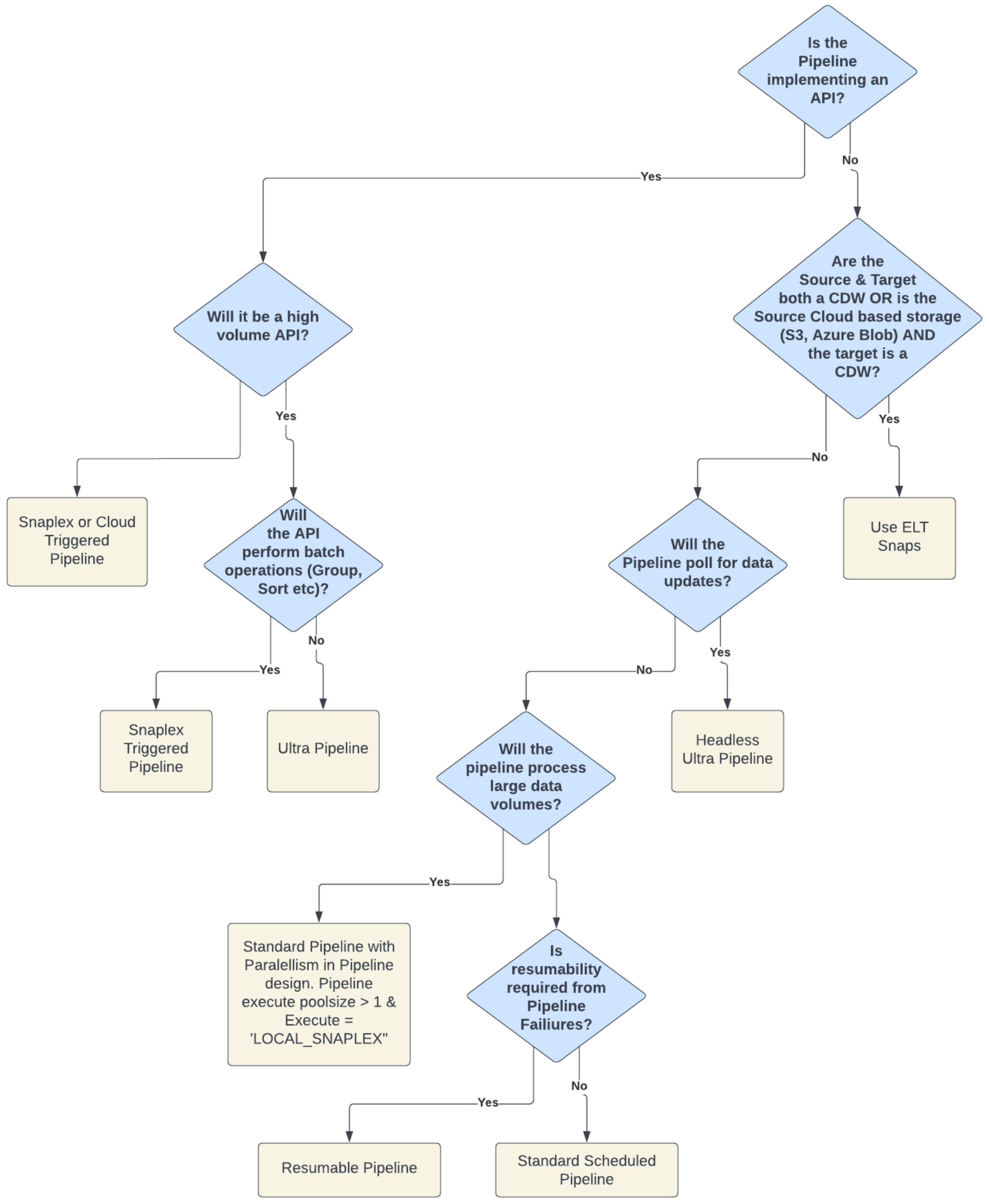
Table 1.0 Pipeline state transitions


Control Plane
1 - Initiate Pipeline
2 - Metadata sent to Snaplex
3 - Prepare Pipeline, fetch & decrypt account credentials. Time limited URL
5 - Send execution metrics
6 - Pipeline completion Teardown & release of resources
State = Executing
State = Prepare
Snap
Streams documents up to ring buffer limit - 1024 Documents
Snap
State = Completed
Pipeline
Snaplex Node
4 - Connect to endpoint security
Snap Endpoint

Pipeline execution flow

Pipeline Design Decision Flow

The following decision tree can be used to establish the best Pipeline Design approach for a given use case.


Is the Pipeline implementing an API?
Yes
No
Will it be a high volume API?
Are the Source & Target both a CDW OR is the Source Cloud based storage (S3, Azure Blob) AND the target is a CDW?
Yes
No
Yes
Snaplex or Cloud Triggered Pipeline
Will the API perform batch operations (Group, Sort etc)?
Will the Pipeline poll for data updates?
Use ELT Snaps
No
Yes
No
Yes
Snaplex Triggered Pipeline
Ultra Pipeline
Will the pipeline process large data volumes?
Headless Ultra Pipeline
Yes
Standard Pipeline with Paralellism in Pipeline design. Pipeline execute poolsize > 1 & Execute = 'LOCAL_SNAPLEX"
Is resumability required from Pipeline Failiures?
Yes
No
Resumable Pipeline
Standard Scheduled Pipeline

Snap Execution Model

Snaps can be generally categorized into these types:

- **Fully Streaming**

  Most Snaps follow a fully streaming model. i.e. Read one document from the Input view (or from the source endpoint for Read Snaps), and write one document to the Output view or to the target endpoint.

- **Streaming with batching**

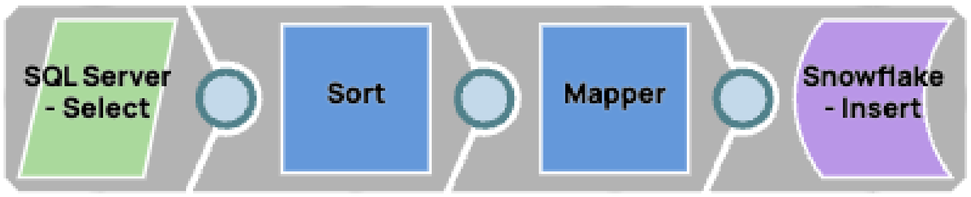
  Some Snaps are streaming with batching behavior. For example, the DB Insert Snap reads N documents and then makes one call to the database (where N is the batch size set in the database account).

- **Aggregating**

  Aggregating type Snaps (e.g. Aggregate, Group By, Join, Sort, Unique etc.) read all input documents before any output is written to the Output view. Aggregating Snaps can change the Pipeline execution characteristics significantly as these Snaps must receive all upstream documents before processing and sending the documents to the downstream Snaps.

Pipeline Data Buffering

Connected Snaps with a Pipeline communicate with one another using Input and Output views. An Input view accepts data being passed from an upstream snap, it operates on the data and then passes the data to its Output view. Each view implements a separate in-memory ring buffer at runtime. Given the following example, the Pipeline will have three separate ring buffers. These are represented by the circular connections between each snap (diamond shaped connections for binary Snaps).



- The size of each ring buffer can be configured by setting the below feature flags on the org. The default values are 1024 and 128 for DOCUMENT and BINARY data formats respectively.

```
com.snaplogic.cc.jstream.view.publisher.AbstractPublisher.DOC_RING_BUFFER_SIZE=
1024

com.snaplogic.cc.jstream.view.publisher.AbstractPublisher.BINARY_RING_BUFFER_SIZE=
128
```

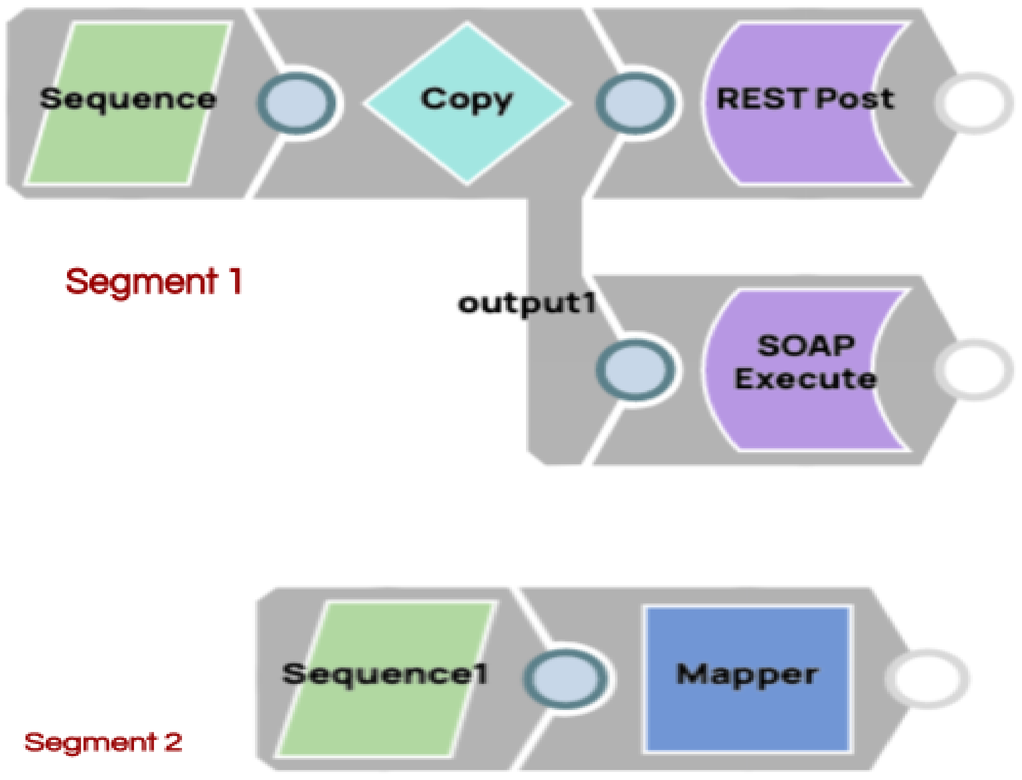
The values must be set as powers of ***two***.

- The source Snap reads data from the endpoint and writes to the Output view.
- If the buffer is full (i.e. if the Consumer Snap is slow), then the Producer Snap will block on the write operation for the 1025th document.
- Pipeline branches execute independently. However in some cases, the data flow of a branch in a Pipeline can get blocked until another branch completes streaming the document.

  Example: A Join Snap might hang if its upstream Snaps (e.g. Copy, Router, Aggregator, or similar) has a blocked branch.
  This can be alleviated by setting *Sorted streams* to *Unsorted* in the Join Snap to buffer all documents in input views internally.
- The actual threads that a Pipeline consumes can be higher than the number of Snaps in a Pipeline.
- Some Snaps such as *Pipeline Execute, Bulk loaders*, and Snaps performing input/output, can use a higher number of threads compared to other Snaps.

Sample Pipeline illustration for threads and buffers

The following example Pipeline demonstrates the practical example of how the usage and composition of Snaps within a Pipeline change the characteristics of how the Pipeline will operate once it is executed.



- Six threads are initialized at Pipeline startup. There are a total of seven ring buffers. The Copy Snap has two buffers, all other Snaps have one output buffer each.
- There are two segments that run in parallel and are isolated (other than the fact that they run on the same node, sharing CPU/memory/IO bandwidth).
- The first segment has two branches. Performance of one branch can impact the other. For example, if the SOAP branch is slow, then the Copy Snap's buffer for the SOAP branch will get full. At this point, the Copy Snap will stop processing documents until there is space available in the SOAP branch's buffer.
- Placing an aggregating Snap like the *Sort* Snap in the slow branch changes the performance characteristics significantly as the Snap must receive all upstream documents before processing and sending the documents to the downstream Snaps.

Memory Configuration thresholds

Property / Threshold	Where configured	Default value	Comments
Maximum memory %	Node properties tab of the Snaplex	85 (%)	Threshold at which no more Pipelines will be assigned to a node
Pipeline termination threshold	Internal (Can be configured by setting the feature flag at the org level com.snaplogic.cc.snap.common.SnapThreadStatsPoller.MEMORY_HIGH_WATERMARK_PERCENT)	95 (%)	Threshold at which the active Pipeline management feature kicks in and terminates Pipelines when the node memory consumption exceeds the threshold. Ideal range: 75-99
Pipeline restart delay interval	Internal (Can be configured by setting the feature flag at the org level com.snaplogic.cc.snap.common.SnapThreadStatsPoller.PIPELINE_RESTART_DELAY_SECS)	30 (seconds)	One Pipeline is terminated every 30 seconds until the node memory goes below the threshold (i.e. goes below 95%) Range: 75-99

Table 2.0 Snaplex node memory configurations

The above thresholds can be optimized to minimize Pipeline terminations due Out-of-Memory exceptions. Note that the memory thresholds are based on the Physical memory on the node, and not the Virtual / Swap memory.

Additional Reference: Optimizations for Swap Memory

Hypothetical scenario

Add 16 GB swap memory to a Snaplex node with 8 GB physical memory.

Property	Comments
Swap Space on the server	Add 16 GB of swap / virtual memory to the node.
Total Memory	Total Memory is now = 24 GB (8 GB Physical plus 16 GB Virtual)
Maximum Heap Size	Set to 90% (of 24 GB) = 22 GB
Maximum Memory	Set to 31% rounded (of 22 GB) = 7 GB

Table 3.0 Snaplex node memory configurations

By updating the memory configurations as in the above example, the JCC utilizes 7 GB of the available 8 GB memory. Beyond that value, the load balancer would queue up additional Pipelines or distribute them across other nodes.

- *Use the default configurations for normal workloads, and use Swap-enabled configuration for dynamic workloads.*
- *When your workload exceeds the available physical memory and the swap is utilized, the JCC can become slower due to additional IO overhead caused by swapping. Hence, configure a higher timeout for `jcc.status_timeout_seconds` and `jcc.jcc_poll_timeout_seconds` for the JCC health checks.*
- *We recommend that you limit to 16 GB the maximum swap to be used by the JCC. Using a larger swap configuration causes performance degradation during the JRE garbage collection operations.*

Modularization

Modularization can be implemented in SnapLogic Pipelines by making use of the *Pipeline Execute* Snap. This approach enables you to:

- Structure complex Pipelines into smaller segments through child Pipelines.
- Initiate parallel data processing using the Pooling option.
- Reuse child Pipelines.

- Orchestrate data processing across nodes, within the Snaplex or across Snaplexes.
- Distribute global values through Pipeline parameters across a set of child Pipeline Snaps.

Modularization best practices:

- Modularize by business or technical functions.
- Modularize based on functionality and avoid deep nesting or nesting without a purpose.
- Modularize to simplify overly-complex Pipelines and reduce in-page references.
- Use the Pipeline Execute Snap over other Snaps such as Task Execute, ForEach, Auto-router (i.e. Router Snap with no routes defined with expressions), or Nested Pipelines.

Pipeline Reuse with Pipeline Execute

Detailed documentation with examples can be found in the SnapLogic documentation for Pipeline Execute.

Use Pipeline Execute when:

- The child Pipeline is CPU/memory heavy and parallel processing can help increase throughput.

Avoid when:

- The child Pipeline is lightweight where the distribution overhead can be higher than the benefit.

Additional recommendations and best practices for the Pipeline Execute Snap:

- Use *Reuse* mode to reduce child runtime creation overhead. Reuse mode allows each child Pipeline instance to process multiple input documents. Note that the child Pipeline must be a streaming Pipeline for reuse mode.
- Use the *batching (Batch size)* option to batch data (avoid grouping records in parent).
- Use the *Pool size* (parallelism) option to add concurrency.
- If the document count is low then use the *Pipeline Execute* Snap for structuring Pipelines else embed the child segment within the Parent Pipeline instead of using *Pipeline Execute.*
- Set the *Pool Size* to > 1 to enable concurrent executions up to the specified pool size.
- Set *Batch Size* = *N (where N > 1).* This sends *N* number of documents to the child Pipeline

input view.

- Use *Execute On* to specify the target Snaplex for the child Pipeline. *Execute On* can be set to one of the below values:
  - *LOCAL_NODE.* Runs the child Pipeline on the same node as the parent Pipeline. This is recommended when the child Pipeline is being used for Pipeline structuring and reuse rather than Pipeline workload distribution. This option is used for most child Pipeline executions.
  - *LOCAL_SNAPLEX.* Runs the child Pipeline on one of the available nodes in the same Snaplex as the parent Pipeline. The least utilized node principle is applied to determine the node where the child Pipeline will run.This has dependency on the network, and must be used when workload distribution within the Snaplex is required.
  - *SNAPLEX_WITH_PATH.* Runs the child Pipeline on a user-specified Snaplex. This allows high workload distribution, and must be used when the child Pipeline has to run on a different Snaplex for endpoint connectivity restrictions or for effective workload distribution. This option also allows you to use Pipeline parameters to define relative paths for the Snaplex name.

Additional Pipeline design recommendations

This section lists some recommendations to improve Pipeline efficiency.

SLDB

Note:

SLDB should not be used as a file source or as a destination in any SnapLogic orgs (Prod / Non-Prod). You can use your own Cloud storage provider for this purpose. You may encounter issues such as file corruption, pipeline failures, inconsistent behavior, SLA violations, and platform latency if using SLDB instead of a separate Cloud storage for the file store.

This applies to all File Reader / Writer Snaps and the SnapLogic API.

1. *File Read from an SLDB File source.*
2. *File Write operations to SLDB as a destination.*

Use your own Cloud storage instead of SLDB for the following (or any other) *File Read / Write* use-cases:

- *Store last run timestamps or other tracking information for processed documents.*
- *Store log files.*
- *Store other sensitive information.*
- *Read files from SLDB store.*

Avoid using the Record Replay Snap in Production environments as the recorded documents are stored in an SLDB path making them visible to users with Read access.

Snaps

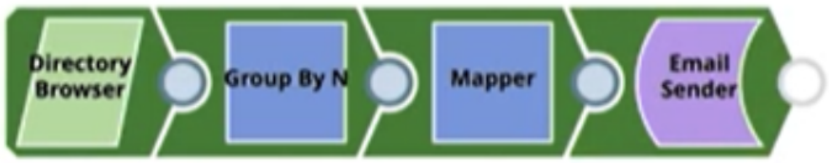
- Enable Pagination for Snaps where supported (e.g. REST Snaps, HTTP Client, GraphQL, Marketo, etc.). There should also always be a Pagination interval to ensure that too many requests are not made in a short time.

- Use the *Group By N Snap* where there is a requirement to limit request sizes. E.g. Marketo API request.

- The *Group By Fields* Snap creates a new group every time a record with a different Group Field value is received. Place a *Sort Snap* before Group By Fields to avoid multiple sets of documents with the same group value.

- *XML Parser Snap* with a Splitter expression reduces memory overhead when reading large XML files.

- Use an *Email Sender Snap* with a *Group By Snap* to minimize the number of emails that get sent out.



Pipelines

- *Batch Size* (only available if the *Reuse executions* option is not enabled) is used to control the amount of records that are passed into a child Pipeline. Setting this value to 1 will pass a single record for each instance of the child Pipeline. Avoid using this approach when processing large volumes of documents.
- Do not schedule a chain reaction. When possible, separate a large Pipeline into smaller pieces and schedule the individual Pipelines independently. Distribute the execution of resources across the timeline and avoid a chain reaction.
- Integration API limits must not exceed across all integrations running at the same time. *Group By Snaps* or *Pipeline Execute* can be used to achieve this.

Optimization recommendations for common scenarios

Scenario	Recommendation	Feature(s)
Multiple Pipelines with similar structure	Use parameterization with Pipeline Execute to reuse Pipelines	Pipeline Execute Pipeline parameters
Bulk Loading to target datasource	Use Bulk Load Snaps where available (e.g. Azure SQL - Bulk Load, Snowflake - Bulk Load)	Bulk Loading
Mapper snap contains a large amount of mappings where the source & target field names are consistent	Enable “Pass through” setting on the Mapper.	Mapper - Pass Through
Processing large data loads	Perform target load operation within a Child Pipeline using the “Pipeline Execute” snap with “Execute On” set to “LOCAL_SNAPLEX”.	Pipeline Execute

Performing complex transformations and/or JOIN/SORT operations across multiple tables	Perform transformations & operations within SQL query	SQL Query Snaps
High Throughput Message Queue to Database ingestion	Batch polling and ingestion of messages by: • Specifying matching values for *Max Poll Record (Consumer Snap)* with *Batch Size (Database Account Setting).* • Performing database ingestion within a child Pipeline with *Reuse* Enabled on the Pipeline Execute Snap.	Consumer Snaps Database Load Snaps

Table 4.0 Optimization recommendations

Configuring Triggered and Ultra Tasks for Optimal Performance

Ultra Tasks

Definition and Characteristics

An Ultra Task is a type of task which can be used to execute Ultra Pipelines. Ultra Tasks are well-suited for scenarios where there is a need to process large volumes of data with low latency, high throughput, and persistent execution.
While the performance of an Ultra Pipeline largely depends on the response times of the external applications to which the Pipeline connects to, there are a number of best practice recommendations that can be followed to ensure optimal performance and availability.

General Ultra Best Practices

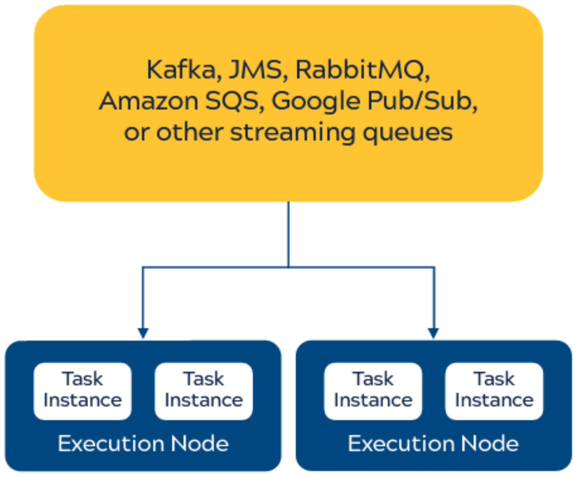
- Before building an Ultra Pipeline, consult the "Snap Support for Ultra Pipelines" documentation to understand if the desired Snaps are supported.
- For optimal Ultra performance, deploy a dedicated Snaplex to support Ultra workloads.

There are two modes of Ultra Tasks - *Headless Ultra* and *Low Latency Ultra API* with each mode being characterized by the design of the Pipeline which is invoked by the Ultra Task. The modes are described in more detail below.

Headless Ultra

A Headless Ultra Pipeline is an Ultra Pipeline which does not require a Feedmaster, and where the data source is a Listener or Consumer type construct, for example Kafka Consumer, File Poller, SAP IDOC Listener (For a detailed list of supported Snaps, please click here).

The Headless Ultra Pipeline executes continuously and polls the data source according to the frequency configured within the Snap passing documents from the source to downstream Snaps.



Use Cases

- Processing real-time data streams such as message queues.
- High volume message or file processing patterns with concurrency.
- Publish/Subscribe messaging patterns.

Best Practices

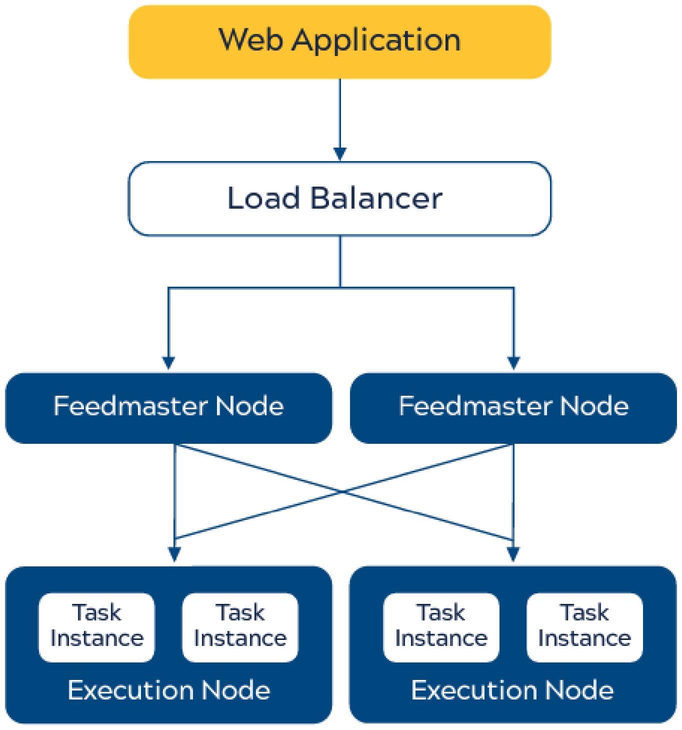
- Deploy multiple instances of the Ultra Task for High Availability.

- Decompose complex Pipelines into independent Pipeline using a Publish-Subscribe pattern.
- Lower the dependency on the Control Plane by avoiding the use of expressions to declare queue names, account paths etc.
- Set the '*Maximum Failures*' Ultra Task configuration threshold according to the desired tolerance for failure.
- For long running Ultra Pipelines, set a higher '*Max In-Flight*' option to a higher value within the Ultra Task configuration.
- When slow performing endpoints are observed within the Pipeline, use the Pipeline Execute Snap with *Reuse* mode enabled and the *Pool Size* field set to > 1 to create concurrency across multiple requests to the endpoint.

  Additional reference: Ultra Tasks

Low Latency API Ultra

Low Latency API Ultra is a high-performance API execution mode designed for real-time, low-latency data integration and processing. The Pipeline invoked by the Ultra Task is characterized by having an open input view for the first Snap used in the Pipeline (typically a HTTP Router or Mapper Snap). Requests made to the API are brokered through a 'Feedmaster Node', guaranteeing at least once message delivery.



Use Cases

- High frequency & high throughput request-response use cases.
- Sub-second response times requirement.

Best Practices

- Deploy multiple Feedmasters for High Availability.
- Deploy multiple instances of the Ultra Task for High Availability running within the same Snaplex.
- Leverage the 'Alias' setting within the Ultra Task configuration to support multi Snaplex High Availablity.
- To support unpredictable high volume API workloads, leverage the '*Autoscale based on Feedmaster queue*' instance setting in the Ultra task configuration.
- When slow performing endpoints are observed within the Pipeline, use the Pipeline Execute Snap with the *Reuse* mode enabled and the *Pool Size* field set to > 1 to create concurrency across multiple requests to the endpoint.

- Use the *HTTP Router* Snap to handle supported & unsupported HTTP methods implemented by the Pipeline.
- Handle errors that may occur during the execution of the Pipeline and return the appropriate HTTP status code within the API response. This can be done either by using the *Mapper*, *JSON Formatter* or the *XML Formatter* Snap.
- Reference request query parameters using the *$query* object.
- Set the 'Maximum Failures' Ultra Task configuration setting according to the desired tolerance for failure.
- For long running Ultra Pipelines, set a higher 'Max In-Flight' setting within the Ultra Task configuration.

Triggered Tasks

Definition and Characteristics

Triggered Tasks offer the method of invoking a Pipeline using an API endpoint when the consumption pattern of the API is infrequent and/or does not require low latency response times.

Use Cases

- When a batch operation is required within the Pipeline, e.g. Join, Group By, Sort etc.
- Integrations that need to be initiated on-demand.
- Non-real time data ingestion.
- File ingestion and processing.
- Bulk data export APIs.

Best Practices

- Avoid deep nesting of large child Pipelines.
- Use Snaplex URL to execute Triggered Tasks for reduced latency response times.
- Handle errors that may occur during the execution of the Pipeline and return the appropriate HTTP status code within the API response. This can be done either by using the *Mapper*, *JSON Formatter* or the *XML Formatter* Snap.
- Use the HTTP Router snap to handle supported & unsupported HTTP methods implemented by the Pipeline.
- Parallelise large data loads using the "*Pipeline Execute*" Snap with *Pool Size* > 1