ScienceLogic

Snippet Framework

Skylar One version 12.5.20

Table of Contents

Chapter 1

Introduction

Overview

This manual describes the Snippet Framework: the execution engine within Skylar One that runs snippet-based monitoring logic. It explains how the framework is structured, how it executes monitoring workflows, and how it relates to the toolkits, PowerPacks, Dynamic Applications, and user interface tools that depend on it.

This manual is intended for:

- Monitoring developers who build Dynamic Applications using snippet-based workflows.
- System administrators who install, configure, and manage snippet-based monitoring content.
- Technical architects who need to understand how Skylar One's monitoring execution layer works. This manual focuses on concepts and configuration. It is not a step-by-step guide.

NOTE: This manual focuses on the Snippet Framework's execution model and architecture. For instructions on building Dynamic Applications in the Skylar One user interface, see the **Dynamic Application Development** and **Snippet Dynamic Application Development** manuals.

NOTE: For more information, visit the Developer Documentation site for the full documentation.

This chapter covers the following topics:

Prerequisites

Before reading this manual, you should be familiar with:

- Basic Skylar One administration, including devices, credentials, and PowerPacks.
- The concept of Dynamic Applications and how they define monitoring behavior. See the **Dynamic Application Development** manual for an introduction.

NOTE: Familiarity with YAML syntax is helpful but not required.

Related Manuals

Standalone Manuals

Manuals	What It Covers
Dynamic Application Development	The foundational reference for building Dynamic Applications of any type. Covers how to configure Dynamic Application properties, collection objects, presentation objects, alerts, and thresholds. Also covers caching, collector affinity, and Execution Environments. Start here before reading any protocol-specific manual.
ScienceLogic Libraries and Execution Environments	Covers the Python virtual environments (Execution Environments) and ScienceLogic Libraries that the Snippet Framework depends on to run toolkit steps. Explains how to install, update, and align libraries to Dynamic Applications.
Dynamic Application Builder	Covers the Dynamic Application Builder, a standalone desktop tool (part of the Skylar One Studio suite) for building snippet-based Dynamic Applications on your local machine. Walks through installing the tool, creating credentials, building Dynamic Applications using a step-by-step wizard, and exporting them to a Skylar One system.

PowerPack Manuals

Manuals	What It Covers
CLI: Toolkit PowerPack	Covers the "CLI: Toolkit" PowerPack, which adds command-line and SSH-based collection steps to the Snippet Framework. Includes installation instructions, an overview of what the PowerPack provides (execution environment, template PowerPack, support for caching and pagination),

	and links to the full step reference in the developer docs.
REST: Toolkit PowerPack	Covers the "REST: Toolkit" PowerPack, which adds HTTP/REST API collection steps to the Snippet Framework. Includes installation instructions and an overview of the REST libraries, execution environment, example Dynamic Applications, and built-in credential support for REST authentication.
Prometheus: Toolkit PowerPack	Covers the "Prometheus: Toolkit" PowerPack, which adds PromQL-based metrics collection steps to the Snippet Framework. Includes installation instructions and an overview of the PromQL execution environment, template PowerPack, and supported authentication types (Basic, Bearer Token, API Key, OAuth2).
Low-code Tools PowerPack	Covers the "Low-code Tools" PowerPack, the single consolidated toolkit that replaces the individual CLI, REST, and Prometheus Toolkit PowerPacks. Provides steps for REST, CLI, SSH, SNMP, Prometheus, JMX, and WinRM collection in one unified package. Recommended for all new Snippet Framework development.

What Is the Snippet Framework?

The Snippet Framework is the execution layer within Skylar One that runs reusable pieces of monitoring logic called snippets. Instead of relying on large, monolithic scripts, the framework breaks monitoring functionality into smaller, focused units that can be configured, combined, and reused across multiple monitoring workflows.

Snippets are designed to perform specific, well-defined tasks, such as:

- Running a command or request
- Calling a REST API
- Parsing or transforming structured output
- Processing monitoring results before they are stored

The Snippet Framework coordinates how these snippets are executed, how data is passed between them, and how execution steps are sequenced to support complex monitoring use cases. This approach enables low-code monitoring development while still supporting advanced customization when needed.

How the Snippet Framework Fits into Skylar One

Monitoring in Skylar One is defined and executed through a layered architecture, with each layer responsible for a specific role.

At a high level:

- Dynamic Applications define what to monitor and when monitoring runs.
- The Snippet Framework executes the monitoring logic required to collect and process data.
- Toolkits provide collections of prebuilt snippets that perform data collection and processing using specific technologies.
- PowerPacks package and distribute snippets, toolkits, Dynamic Applications, and related components.

Dynamic Applications act as the top-level definition. When a Dynamic Application executes, it invokes the Snippet Framework to run one or more snippets that perform the underlying collection and processing work.

Snippets and Snippet Arguments

A snippet is a reusable unit of execution within a monitoring workflow. Snippets are intentionally generic, allowing them to be reused across multiple Dynamic Applications without modification. Snippet behavior is controlled through snippet arguments, which allow configuration without writing custom code. This argument-driven approach is a core design principle of the Snippet Framework.

Examples include:

- A REST snippet that accepts arguments for API endpoints, authentication methods, or paging behavior.
- A CLI snippet that accepts arguments for command strings, timeout values, or parse patterns.

By externalizing behavior into configuration, the Snippet Framework enables low-code monitoring workflows while maintaining flexibility and reusability.

Snippets Execution Model

The Snippet Framework can be thought of as an execution pipeline that assembles and runs monitoring logic based on configuration input.

During execution:

- Dynamic Applications supply execution context and snippet arguments.
- The framework determines the required execution steps.
- Snippets provided by toolkits perform request, processing, and transformation tasks.
- Results are returned to Skylar One for storage, analysis, and visualization.

The framework manages sequencing, error handling, and data flow so that developers can focus on defining what to collect rather than implementing execution mechanics.

Toolkits, Low-Code Tools, and the Snippet Framework

The Snippet Framework itself does not collect data. All data collection logic is supplied through toolkits, which are distributed as PowerPacks.

Common examples include:

- CLI Toolkit, for command-line-based data collection
- REST Toolkit, for API-based monitoring
- Prometheus Toolkit, for metric-driven monitoring

These toolkits provide reusable snippets that follow common conventions and are designed to work seamlessly with the Snippet Framework.

Over time, many standalone toolkits have been consolidated into the "Low-Code Tools" PowerPack, which provides a standardized execution environment and unified set of snippets for building Dynamic Applications across multiple technologies.

PowerPacks and Distribution

Snippets and toolkits are distributed through PowerPacks. A PowerPack may include:

- Snippets and toolkit libraries
- Dynamic Applications that reference those snippets
- Execution environments and supporting assets

PowerPacks allow Snippet Framework-based monitoring logic to be shared, versioned, and deployed consistently across Skylar One systems. Administrators manage PowerPacks through standard Skylar One import and installation workflows.

The Snippet Framework and the User Interface

Although the Snippet Framework is a backend execution system, it is configured and managed through the Skylar One user interface.

When you create or modify snippet-based Dynamic Applications, they typically interact with:

- Snippets arguments
- Execution order
- Toolkit-provided steps

The user interface experience is commonly referred to as the Dynamic Application Builder, which acts as a bridge between the Skylar One user interface and the underlying Snippet Framework.

NOTE: This manual focuses on the Snippet Framework's execution model and architecture. For step-by-step instructions on configuring Dynamic Applications or working with snippets in the Skylar One user interface, see the **Dynamic Application Development** manual

Best Practices for Using the Snippet Framework

When working with the Snippet Framework:

- Design snippets to perform a single, focused task.
- Favor configuration through snippet arguments over custom logic.
- Reuse toolkit-provided snippets whenever possible.
- Treat shared snippets as platform building blocks and manage changes carefully.

As snippet libraries grow, consistent naming, documentation, and lifecycle management become essential to maintaining long-term stability.

Chapter 2

Snippet Framework Collection Pipeline

Overview

This manual explains how the Snippet Framework runs a collection from start to finish. It introduces the pipeline model, describes the three step types (Syntax, Requestor, and Processor) and what each one does, and walks through a complete example pipeline so you can see how data moves through the framework during execution.

This chapter covers the following topics:

Snippet Framework Collection

The Snippet Framework executes monitoring logic as a structured pipeline of typed steps. Each run of a collection object passes through this pipeline in a defined sequence, with data flowing from one step to the next.

Understanding the pipeline model is important for building well-structured monitoring workflows and for troubleshooting collection behavior.

How the Collection Pipeline Works

When a Snippet Framework Dynamic Application collects data for a collection object, the framework follows the following steps:

1. Reads the snippet argument for that collection object (a YAML block defining the steps to execute and their configuration).
2. Parses the argument using the Syntax step to produce a structured execution plan.
3. Executes the Requestor to fetch data from the target system.
4. Passes the result through one or more Processors that transform or select values from the response.
5. Returns the final value to Skylar Onefor storage, analysis, and visualization.

The framework batches Requestors across all collection objects in the application and executes them together for efficiency before running per-object Processors downstream.

Step Types

The Snippet Framework defines three and only three step types. All steps are provided by toolkits. the Snippet Framework itself does not ship with any built-in steps. Without a compatible toolkit installed in an Execution Environment, the framework has no steps to execute.

Syntax

The Syntax step parses the raw YAML snippet argument into a structured internal format the framework can consume. Every collection pipeline begins with a Syntax step.

In most cases, a default Syntax step is used automatically, and you do not need to specify it. Custom Syntax steps may be registered by advanced toolkit developers.

Requestor

The Requestor step makes the actual request to the target system, which could be an HTTP call, an SSH command,a Prometheus scrape, or similar.

NOTE: For snippet-based Dynamic Applications that use the `http:` requestor step, SSL certificate verification during collection is controlled by the ***Verify SSL for Collections by Default*** setting on the **Behavior Settings** page (System > Settings > Behavior). This setting is disabled by default. For more information, see the **System Administration** manual.

The Requestor is where data is fetched from the external source. The following are some key behaviors of the Requestor:

- Only one Requestor is active per collection pipeline pass.
- The framework batches all pending Requestors across collection objects and executes them together before proceeding to Processor steps.
- The Requestor is the step most closely tied to the collection protocol.
    - REST Toolkit: `rest_get`, `rest_post`, etc.
    - CLI Toolkit: `ssh`, `local_cmd`, etc.
    - Prometheus Toolkit: `prometheus`

Processor

Processor steps transform or select data from the Requestor response. Multiple Processors can be chained in sequence, with each Processor receiving the output of the previous one.

Common Processor examples from the REST Toolkit:

Processor	What It Does
`json`	Parses a JSON string response into a structured object
`jsonpath`	Selects a value from a structured object using a JMESPath expression
`simple_key`	Selects a single key from a dictionary

Processors are the primary way to extract the specific values your Dynamic Application needs from a larger API response or command output.

Worker processes are allocated dynamically by the Data Collector based on available capacity and the jobs being executed. When a collection run terminates without completing (SIGTERM), a termination entry is written to the system logs, including a list of pending device work items. Execution logs, including logs from failed runs, can be downloaded from the user interface.

Collection Pipeline Execution Flow

A complete pipeline for a REST-based collection object might look like this:

1. Syntax will parse YAML argument.
2. The Requestor will parse the response body as JSON.

3. The Processor selects the target metric value using an expression.
4. The result is when the selected value is returned to Skylar One

This pipeline structure is defined entirely within the snippet argument YAML, not in the snippet code itself. Changing the collection behavior requires updating the YAML configuration, not modifying Python code.

Error Handling and Data Flow

The Snippet Framework manages sequencing, batching, and error handling across the pipeline so that developers can focus on defining what to collect rather than implementing execution mechanics. If a step fails, the framework handles the failure condition according to its internal error-handling rules without requiring custom error-handling code in the snippet.

Chapter 3

Snippet Framework Arguments

Overview

This manual covers snippet arguments, which are the main way you configure collection behavior in the Snippet Framework. It explains what a snippet argument is, shows what the YAML structure looks like, and describes how using arguments instead of custom code makes monitoring development faster and more reusable. It also introduces substitution tokens, which let you inject dynamic runtime values like device hostnames and credentials directly into your arguments.

This chapter covers the following topics:

Defining Snippet Arguments

Snippet arguments are the primary configuration mechanism for Snippet Framework-based Dynamic Applications. They are what make low-code monitoring development possible. Instead of writing custom Python code for every collection need, you define collection behavior through structured configuration that the framework reads and executes.

A snippet argument is a YAML block attached to a collection object in a Snippet-type Dynamic Application. When the Snippet Framework executes the application, it reads the snippet argument to determine:

- Which step types to run and in what order
- How to configure each step (e.g., the endpoint URL, authentication method, parse expression)
- Any substitution tokens to resolve at runtime

Snippet arguments externalize collection behavior from code into configuration. This is a core design principle of the framework. The snippet code is boilerplate; the snippet argument is where the actual logic lives.

Where Snippet Arguments Are Configured

Snippet arguments are configured at the collection object level within a Dynamic Application. In the Skylar One user interface, snippet arguments appear as a YAML input field on each collection object in a snippet-based Dynamic Application.

> **NOTE:** For step-by-step instructions on locating and editing snippet arguments in the UI, see the **Snippet Dynamic Application Development** manual.

Basic Snippet Argument Structure

A snippet argument YAML block specifies a list of steps under a top-level key that identifies the execution context. The exact structure depends on the toolkit in use, but a representative REST-based snippet argument might look like the following:

```
yaml
rest_requestor:
 url: "https://api.example.com/metrics"
 auth: basic

json.processor: {}

jsonpath_processor:
 expression: "data.value"
```

Each entry in the YAML corresponds to a step in the collection pipeline:

- `rest_requestor` tells the framework to use the REST Toolkit's Requestor step, targeting the specified URL with basic authentication.
- `json_processor` tells the framework tells the framework to parse the response body as JSON.
- `jsonpath.processor` tells the framework to extract the value at `data.value` from the parsed response.

> **NOTE:** The exact YAML keys, field names, and supported options for each step depend on the toolkit providing that step. For more information, see the toolkit-specific manuals.

> **NOTE:** For snippet-based Dynamic Applications that make HTTP/S requests, SSL certificate verification during collection is controlled by the ***Verify SSL for Collections by Default*** setting on the **Behavior Settings** page (System > Settings > Behavior). This setting is disabled by default. For more information, see the **System Administration** manual.

Snippet Argument Development

The argument-driven approach means that:

- The same snippet code can serve dozens of different collection scenarios, each with different behavior defined entirely through its YAML argument.
- Non-developer users can configure collection behavior by editing YAML rather than writing Python.
- Snippet bodies are standardized and interchangeable, reducing maintenance overhead.
- Behavior changes are confined to the argument field, making them easier to review and track.

This is why snippet arguments are described as the central element of low-code monitoring development in the Snippet Framework.

Substitution Tokens

Substitution tokens allow snippet arguments to reference dynamic runtime values rather than hardcoded strings. At execution time, the framework resolves each token to its actual value before passing the argument to the pipeline.

Common uses for substitution tokens include:

- Inserting device-specific values such as hostname or IP address into request URLs
- Referencing credential fields such as usernames or passwords from the aligned credential record
- Inserting collection-specific values such as the collection label or OID

The full substitution token reference is maintained in the developer-facing documentation. See the chapter on **Developer Resources** at the end of this manual.

NOTE: Substitution token availabiltiy and syntax might vary by toolkit version. Confirm the available tokens against the documentation for the specific Low-code Tools or toolkit version installed in your environment.

Chapter 4

Snippet Framework Toolkits and Execution Environments

Overview

This manual explains what you need in place before the Snippet Framework can actually collect data. It covers the fact that the framework does not include any built-in collection steps, and that toolkits supply those steps for specific protocols (CLI, REST, and Prometheus). It also explains what Execution Environments are, why every snippet-based Dynamic Application requires one, and why the "Low-code Tools" PowerPack is the recommended starting point for new development.

This chapter covers the following topics:

Toolkits and Execution Environments Overview

The Snippet Framework is an execution engine. It defines how steps run, how data flows between them, and how pipelines are assembled. It does not, however, provide any step implementations of its own.

NOTE: All step implementations are provided by toolkits.

Toolkits

A toolkit is a library of step implementations for a specific data collection protocol. Toolkits register their steps with the Snippet Framework so that snippet arguments can reference them by name.

Without a compatible toolkit installed and available in an Execution Environment, the framework cannot execute any collection steps. Toolkits are the mechanism through which the Snippet Framework connects to external systems.

ScienceLogic provides three toolkits:

Toolkit	Collection Method	Typical Use Cases
CLI Toolkit	SSH, local commands	Network device CLI, OS-level metrics, file parsing
REST Toolkit	HTTP/HTTPS API requests	REST APIs, JSON or XML responses, OAuth, paging
Prometheus Toolkit	Prometheus metrics scraping	Cloud-native monitoring, Kubernetes, time-series metrics

NOTE: For snippet-based Dynamic Applications that use the REST Toolkit to collect data over HTTPS, SSL certificate verification during collection is controlled by the ***Verify SSL for Collections by Default*** setting on the **Behavior Settings** page (System > Settings > Behavior). This setting is disabled by default. For more information, see the **System Administration** manual.

Execution Environments

An Execution Environment is a named Python virtual environment that packages the ScienceLogic Libraries (toolkits) needed for a collection. Every snippet-based Dynamic Application requires a compatible Execution Environment to be selected.

When a Dynamic Application runs, the Snippet Framework resolves the Execution Environment specified in the Dynamic Application to determine which library versions, and therefore which step implementations are available for that collection.

Execution Environments are managed through the Skylar One user interface in the **ScienceLogicLibrary Manager** page (System > Customize > ScienceLogic Libraries). A Dynamic Application's Execution Environment must include the library version that provides the steps references in its snippet arguments. "Low-code Tools" PowerPack installations automatically make the corresponding Execution Environment available.

> **NOTE:** For instructions on managing Execution Environments, see the **Dynamic Application Development** manual.

The Low-Code Tools PowerPack

Over time, the three individual toolkit PowerPacks (CLI, REST, Prometheus) have been consolidated into the "Low-Code Tools" PowerPack. This PowerPack is the current recommended toolkit for all new snippet-based Dynamic Application development.

> **NOTE:** Version 104 of the "Low-code Tools" PowerPack is included in the Skylar One 12.5.20 ISO.

The "Low-code Tools" PowerPack contains the following capabilities and features:

- A single, versioned package containing all three toolkit step implementations
- A standardized Execution Environment
- Bundled directly in Skylar One releases starting with version 12.6.0, meaning it does not require a separate PowerPack installation in those environments

> **NOTE:** The individual CLI, REST, and Prometheus Toolkit PowerPacks continue to exist as legacy documents. They describe the toolkit implementations as they existed before consolidation. For most users working with current Skylar One releases, see the **Low-Code Tools PowerPack** manual.

Chapter

5

Snippet Framework PowerPacks and Distribution

Overview

This manual covers how Snippet Framework content is packaged and shared. It explains what a PowerPack can include, how the argument-driven design allows a small number of reusable snippets to power many different Dynamic Applications through configuration alone, and what happens in Skylar One when an administrator installs a toolkit PowerPack.

This chapter covers the following topics:

PowerPacks and Distribution Overview

Snippets, toolkits, Dynamic Applications, and related content are distributed through PowerPacks. A PowerPack might include any combination of:

- Snippet bodies and toolkit libraries
- Dynamic Applications that reference those snippets
- A standardized Execution Environment
- Bundled directly in Skylar One releases starting with version 12.6.0, meaning it does not require a separate PowerPack installation in those environments

How Distribution Works

PowerPacks allow Snippet Framework-based monitoring logic to be packaged, versioned, and deployed consistently across Skylar One systems. Because all collection behavior is defined through snippet arguments rather than custom code, the same snippet bodies can serve many different use cases simply by varying the YAML configuration in each Dynamic Application.

This means a single PowerPack can contain a small number of reusable snippets that support many different Dynamic Applications, all configured through their respective snippet arguments.

Installing PowerPacks

Administrators manage PowerPacks through standard Skylar One import and installation workflows. Installing a PowerPack that contains toolkit content typically:

- Installs the ScienceLogic Library package into Skylar One.
- Makes the corresponding Execution Environment available for selection in Dynamic Applications.
- Imports any included Dynamic Applications in an unaligned state, ready for device alignment.

NOTE: For step-by-step PowerPack installation instructions, see the **Managing PowerPacks** manual.

Chapter 6

Snippet Framework User Interface

Overview

This manual clarifies the relationship between the Snippet Framework as a backend engine and the tools you use to work with it. It covers the Dynamic Application Builder as the interface for authoring and testing snippet-based Dynamic Applications, explains what the Low-Code Explorer (LCE) service does behind the scenes, and includes a reference table mapping common tasks to the manual where you can find instructions for them.

This chapter covers the following topics:

The Snippet Framework User Interface Overview

The Snippet Framework is a backend execution engine. The user-facing experience for building and managing snippet-based Dynamic Applications is provided by a separate but closely related layer: the Dynamic Application Builder.

The Dynamic Application Builder

The Dynamic Application Builder is the product user interface within Skylar One for authoring, editing, testing, and debugging snippet-based Dynamic Applications. When you work with snippet arguments, Execution Environments, and collection step configuration in the Skylar One user interface, you are working through the Dynamic Application Builder.

The Dynamic Application Builder is an add-on layer to the standard Dynamic Application configuration page. It does not replace the existing user interface, and instead it extends snippet-specific capabilities.

NOTE: For step-by-step instructions on using the Dynamic Application Builder, see the **Dynamic Application Development** manual.

The Low-Code Explorer (LCE) Service

The backend engine that powers the Dynamic Application Builder experience is the Low-Code Explorer service (LCE) service. The LCE service is a daemonized HTTP service that runs on Skylar One databases and Collector appliances. The tooling also runs on Administration Portal appliances in distributed deployments. Key actions in the user interface include library management, wheel downloads, execution environment alignment, snippet linting, and collection run. All of these are captured in organization-level audit logs.

The LCE Service:

- Manages Execution Environments and ScienceLogic Library installations
- Executes collection runs initiated from the Dynamic Application Builder
- Lints and validates snippet argument YAML
- Provides the endpoints for library management, test runs, and wheel downloads
- Aligns downloaded third-party library wheels to the current execution environment, replacing existing libraries when applicable
- Supports custom pip configuration for the Wheel Downloader, including custom repositories and trusted hosts, while preserving default behavior when not configured

The LCE service runs automatically as part of a Skylar One installation that includes the "Low-code Tools" PowerPack. Administrators do not typically interact with the LCE service directly as it is the backend engine behind the Dynamic Application Builder.

The third-party library download workflow can automatically align downloaded wheels to the current execution environment, replacing existing libraries when applicable. You can configure pip for the Wheel Downloader from the user interface to specify custom repositories and trusted hosts. Default pip behavior is preserved when no custom configuration is set.

> **NOTE:** In engineering documentation and release notes, the Dynamic Application Builder experience and the LCE backend service may be referenced together as the Snippet Framework User Interface. This manual and the product user interface use Dynamic Application Builder as the customer-facing name.

When you create a new collection object in the Dynamic Application Builder, the editor can automatically insert a default snippet argument structure sourced from the LCE service backend. This ensures the inserted default reflects the current execution environment and available steps. If the LCE endpoint is unavailable at the time of insertion, the editor falls back to a locally stored default instead.

Governance and Access Control

The LCE service enforces authentication and permission checks on all endpoints. Unauthenticated requests return HTTP 401; requests from users without sufficient permissions return HTTP 403. This applies to snippet, library, environment, and execution actions.

Key actions are captured in organization-level audit logs, including library management, wheel downloads, execution environment alignment, snippet linting, and collection runs.

Memory usage for the LCE service is managed through configurable `MemoryHigh` and `MemoryMax` values.

Chapter

7

Snippet Framework Advanced Topics

Overview

This manual introduces three capabilities that go beyond standard collection workflows: bulk snippets for APIs that return data for multiple collection objects at once, caching for reducing redundant requests within or across collection cycles, and custom steps for developers who need to write their own Requestors or Processors.

Each topic is covered at a conceptual level with pointers to the developer documentation for full implementation details.

This chapter covers the following topics:

Advanced Topics

This chapter introduces advanced Snippet Framework capabilities that address specific use cases beyond standard single-collection pipelines. These topics are summarized here for reference; full implementation detail is available in the developer-facing documentation.

Bulk Snippets

A bulk snippet is a variant of the standard Snippet Framework pattern designed for use cases where a single request returns data for multiple collection objects simultaneously. For example, an API response that contains a list of devices, each with multiple metrics.

In standard execution, the framework runs one pipeline pass per collection object. In bulk mode, a single Requestor retrieves a batch of data, and the framework distributes values from that batch to the appropriate collection objects in post-processing.

Bulk snippets are particularly useful when making repeated individual requests would be inefficient or would exceed API rate limits.

> **NOTE:** For more information on implementation details, visit the Developer Documentation site for the full documentation.

Caching

The Snippet Framework supports caching of intermediate data to avoid redundant requests acros collection runs or within a single execution. Caching is useful when:

- Multiple collection objects depend on the same upstream data (e.g., a device inventory endpoint)
- A request result is needed across multiple Dynamic Application execution cycles
- An expensive request produces stable data that does not change between short-interval polls

Caching behavior is configured within snippet arguments and managed by the framework automatically.

> **NOTE:** Caching behavior within the Snippet Framework is distinct from the caching mechanisms described in the **Dynamic Application Development** manual, which covers Dynamic Application-level caching for non-snippet collection types.

Custom Steps

The Snippet Framework's pipeline model is extensible. Developers can write custom Requestor and Processor steps using decorators from the "silo_low_code" library:

```
python
from silo.low_code import register_requestor, register_processor

@register_requestor("my_custom_requestor")
def my_requestor(context, **kwargs):
 ...

@register_processor("my_custom_processor")
def my_processor(data, context, **kwargs):
 ...
```

Custom steps are referenced in snippet arguments by their registered names, exactly like toolkit-provided steps. They can be packaged into ScienceLogic Libraries and distributed through PowerPacks.

NOTE: For more information on implementation details, visit the Developer Documentation site for the full documentation.

Developer Resources

The Developer Documentation site is a reference site maintained separately from this manual. It covers:

- The Python library API for the Snippet Framework
- Collection pipeline internals
- Step registration patterns and examples
- The full substitution token reference
- Bulk snippet implementation
- Worked examples for CLI, REST, and Prometheus collection

NOTE: The Developer Documentation site might not always correspond to the current "Low-code Tools" PowerPack version installed in your environment. Verify the library version in your Execution Environment against the documentation version before relying on specific API details.

Chapter 8

Snippet Framework Best Practices

Overview

This manual offers practical guidance for building and maintaining Snippet Framework content over time. It covers how to design snippets that are focused and reusable, how to write clean and reliable snippet arguments, and how to manage shared monitoring content responsibly as your library of snippets and Dynamic Applications grows.

This chapter covers the following topics:

Best Practices

The following guidelines reflect recommended patterns for building, maintaining, and scaling Snippet Framework-based monitoring content.

Snippet Design

- Design each snippet to perform a single, focused task. Snippets are most reusable when they have a narrow, well-defined responsibility. Avoid combining collection, transformation, and output logic into a single snippet when those concerns can be separated into pipeline steps.
- Favor configuration through snippet arguments over custom logic. If a behavior change can be expressed through YAML configuration, prefer that over writing or modifying Python code. This keeps snippets generic and maximizes their reuse value.
- Reuse toolkit-provided steps whenever possible. CLI, REST, and Prometheus steps in the "Low-code Tools" PowerPack have been developed, tested, and maintained by ScienceLogic. Custom steps add maintenance overhead and can diverge from framework conventions over time.

Snippet Arguments

Administrators manage PowerPacks through standard Skylar One import and installation workflows. Installing a PowerPack that contains toolkit content typically:

- Keep snippet argument YAML readable and consistently formatted. Snippet arguments are the configuration record for a collection object. They should be legible to any developer who maintains the application.
- Use substitution tokens for all runtime-variable values. Hardcoding device hostnames, credentials, or environment-specific values into snippet arguments creates fragile configurations that do not generalize across devices or deployments.
- Validate snippet arguments before deploying to production. Use the lint and test features in the Dynamic Application Builder to validate YAML structure and test collection behavior against a real device before aligning the Dynamic Application broadly.

Lifecycle and Governance

- Treat shared snippets as platform building blocks, not application-specific code. Changes to shared snippets affect every Dynamic Application that references them. Manage changes through a review process and test impact before deploying to production.
- Document snippet arguments alongside the application. As snippet libraries grow, argument documentation becomes the primary record of what each Dynamic Application collects and how. Consistent documentation reduces the time needed to troubleshoot and maintain monitoring content.

- Align toolkit and Execution Environment versions deliberately. Upgrading the "Low-code Tools" PowerPack might change the behavior of existing steps. Test collection behavior after toolkit upgrades before rolling out to production systems.
- Use PowerPacks to version and distribute monitoring content. Packaging Dynamic Applications, snippets, and library dependencies together in a PowerPack makes deployment, rollback, and version tracking consistent across environments.

800-SCI-LOGIC (1-800-724-5644)

International: +1-703-354-1010