

Pipeline CLI Framework Runbook

[Runbook Overview](#)

[Pipeline Design](#)

[Migration Process](#)

[Advanced Metadata Considerations](#)

[Prepare the template property](#)

[Manual Migration of Advanced Rules](#)

[Normalize Logic Across all Stages](#)

[CLI Installation](#)

[Authenticate your API Client](#)

[Install the Akamai CLI and packages](#)

[Verify the Installation](#)

[Pipeline Preparation](#)

[Identify target property info](#)

[Pipeline Configuration](#)

[Create a new pipeline project from a base property](#)

[Create pipeline target environments](#)

[Modify the pipeline environment definitions](#)

[Repoint stages to target properties](#)

[Update hostname settings for target properties](#)

[Delete the initial pipeline environments in Luna](#)

[Configure Rule Tree and Stage Values](#)

[Pipeline Project Overview](#)

[Stage Configuration](#)

[Pipeline Tokens Overview](#)

[Complex Tokens](#)

[Tokenize the Template Metadata](#)

[Define Scope and Values for Tokens](#)

[Guidance on identifying token values](#)

[Further Federate the Metadata](#)

[Nested Includes](#)

[Snippet Subdirectories](#)

[Pipeline Operation](#)

[Display Pipeline Status](#)

[Build, Deploy and Activate a Pipeline Stage](#)
[Build a stage specific artifact \(but not deploy it\)](#)
[Check Stage Activation Status](#)
[Reconciling Pipeline State with Luna](#)
[Reconciling Pipeline State](#)
[Reconciling Pipeline Metadata](#)

Runbook Overview

This document is intended to provide supplementary installation, configuration, and usage instructions for the Akamai property-manager CLI.

<https://developer.akamai.com/legacy/cli/packages/property-manager-2.0.html>

Specifically, this guide outlines the specific preparation steps necessary to enable the pipeline CLI framework to manage existing Akamai delivery configurations (rather than starting with net-new properties as outlined in the CLI documentation) and complex Akamai properties in general.

Pipeline Design

The Akamai pipeline CLI enables users to manage their Akamai delivery configuration as code (JSON metadata), in a federated manner outside of Luna Control Center using metadata snippets via discrete JSON files containing a section of the delivery configuration rule tree. The goal of the Akamai pipeline CLI framework is to provide a mechanism to manage a consistent rule tree structure among all pipeline environments, while allowing for the specific behavior configuration settings / attributes to be unique between environments. For instance, all environments are expected to have a common placement of origin behavior, custom certificates, hostnames, etc but the value of those attributes are unique between environments. The pipeline CLI framework also provides a series of operational functions to support testing and deploying metadata to target stage properties.

At a high level, the Akamai pipeline CLI supports the following types of operations:

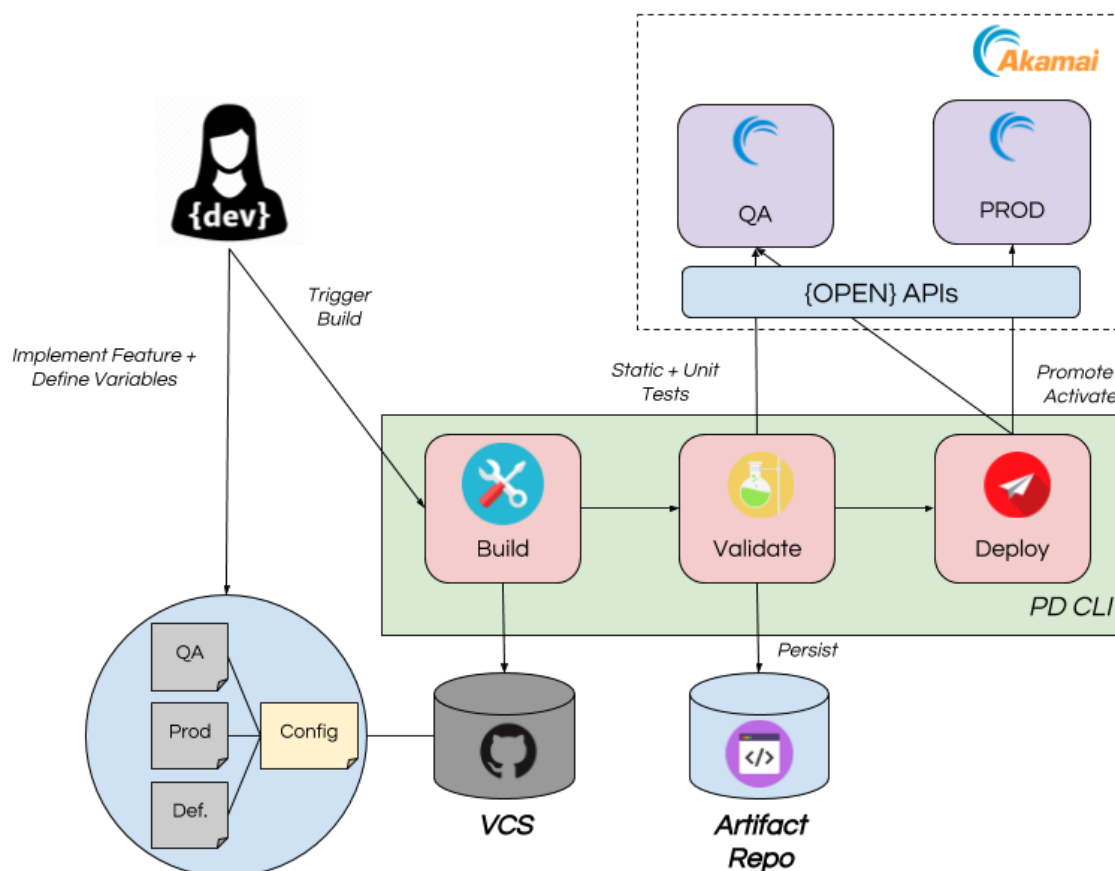
1. **Metadata Validation** - perform rule-tree metadata schema validation,
2. **Build Process** - convert metadata snippets and environment variables into deployable metadata artifacts,

3. **Deployment Processes** - deploy and activation metadata (built by pipeline) to a target environment (stage).

Each operation can be performed independently, or all can be coordinated via a single CLI command (***promote***).

Note: the Akamai pipeline CLI framework enforces a strict deployment order for each environment stage defined (ex: QA -> DEV -> PROD). Any attempted deployment which occurs outside of this predefined sequence will be prevented. If a more variable deployment order is desired, other mechanisms like the Akamai property-manager CLI package (included with pipeline) can be used to perform the post-build deployment step.

Akamai property development and operations using the pipeline CLI are described in the below visualization:



Changes to the rule tree structure (behaviors, match criteria, and settings) are introduced outside of Luna directly within the pipeline project structure, which is described in the [Pipeline Project Overview](#) section. Typically, this step is performed on a developers local machine and maintained within some type of remote version control system (VCS) to enable collaborative

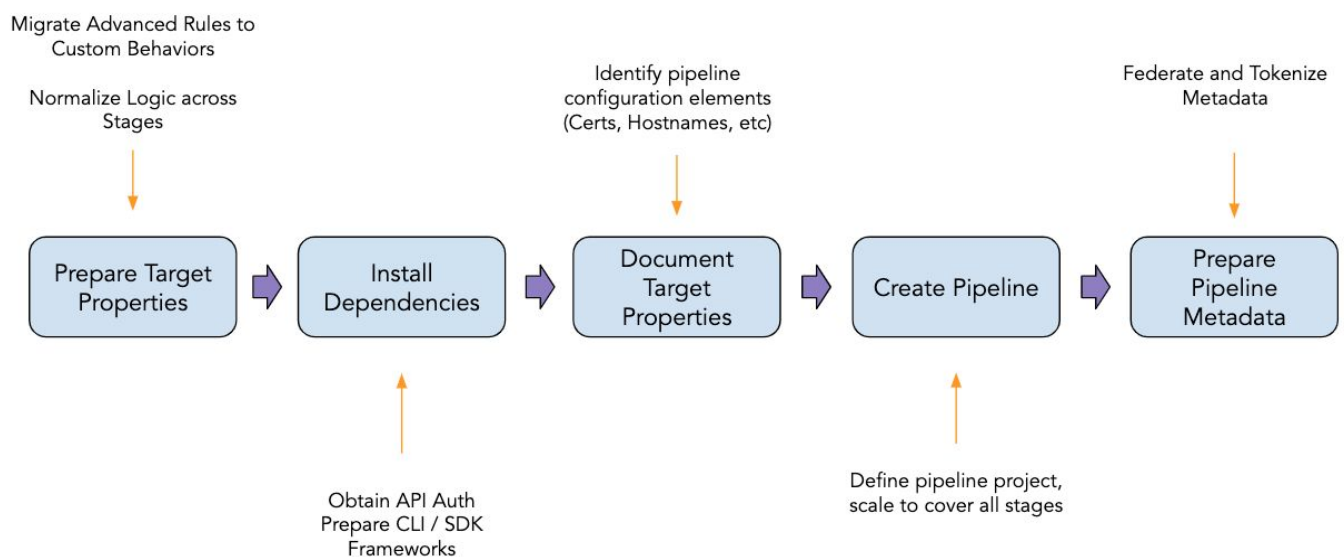
development of the pipeline project artifacts. This also enables developers to subject the pipeline source metadata to conventional VCS branching, change review, and tagging processes used in traditional software development.

It is important to note that any change introduced to a property outside of the pipeline CLI framework (through Luna or other mechanisms) cannot be automatically reconciled back to the pipeline project. Such a scenario would cause a functional drift between the actual property state and the state of the externally managed pipeline project. Hence, it is recommended to diligently enforce a structure process in which all property changes are either made directly within the pipeline framework itself, or manually reconciled later. Site emergencies may necessitate the introduction of a break/fix change quickly outside of the pipeline framework, requiring manual reconciliation later. For these scenarios, please reference the [Reconciling Pipeline State with Luna](#) section for specific instructions on manual reconciliation.

In a real-world environment, the Akamai pipeline CLI could be deployed on a **Build Server** along with the project metadata and configuration. The **Build Server** in this context would orchestrate the above mentioned operations in tandem with other customer release processes, such as deploying code to origin or other operational and related tasks.

Migration Process

The migration process outline below assumes that the pipeline project being incepted is intended to manage existing Akamai Ion / DSA properties with a complex rule tree.



Perhaps the most important step is to prepare the target properties for management via the pipeline framework. Since the pipeline CLI will deploy a common, but environment-specific, logical rule tree artifact across all stages (common rules/behaviors, with unique settings) it's important that each target origin environment will operate consistently. If specific environment stages have unique Akamai content delivery requirements, these will need to be evaluated and normalized prior to migration. Again, the goal is to create repeatable operational outcomes between environment stages, so its important that each stage origin can operate as intended using a common rule tree.

Advanced Metadata Considerations

It should be noted that the following types of rule tree elements within your template and target properties are considered 'read-only' and do not permit modification or deployment via the CLI or other APIs.

1. Advanced Rules
2. Advanced Overrides
3. Advanced Match
4. Built-In Read-Only Match
 - a. CDN Network

This constraint is put in place for security purposes, and only Akamai professional services can add or modify read only behaviors and match criteria.

Prepare the template property

In order to enable the migration of all rules successfully via the pipeline framework, it is necessary that all advanced rules and matches be converted first to Akamai Custom Behaviors:

<https://developer.akamai.com/blog/2018/04/26/custom-behaviors-property-manager-papi>

Manual Migration of Advanced Rules

In the event that not all advanced metadata can be completely removed from your template property, Akamai professional services can reconcile it manually across all target properties (this step cannot be performed via CLI/APIs for the above reasons).

The recommended approach for a manual reconciliation process is as follows:

1. Create a new version of the template property- removing all advanced rules from that version, and save the property.
2. Promote the new template version to all target stages.
3. Within Luna control center, manually add the advanced rules back to the template environment, and all target stages (preserving the exact placement and content of the advanced rules).

The above should ideally be done prior to the initial creation of the pipeline, as a preparation step. Going forward, any new advanced rules will need to be introduced following a similar process (again, only in the event they cannot be implemented via Custom Behaviors).

Normalize Logic Across all Stages

The Akamai pipeline framework will create build artifacts for all stages using a common template metadata. In order for this to work effectively, all rules and behaviors must be normalized across each environment so that they work consistently.

CLI Installation

Authenticate your API Client

Create an API client which will be used to execute remote pipeline actions (update, clone, activate) using the following guide:

<https://developer.akamai.com/api/getting-started>

The API client should be associated to a group/role within your Luna contract which has proper access to the target Akamai properties you wish to manage.

The API client should also have READ-WRITE 'Access Level' to the following API Services:

- Property Manager (PAPI)
- Certificate Provisioning System API (CPS)
- Contracts-API_Contracts

Ensure the credentials provisioned are added to a '.edgerc' file (per the instructions above) which is deployed in the \$HOME directory of the build server.

Install the Akamai CLI and packages

Download a CLI binary for the intended environment (Linux, Windows, etc) here:

<https://github.com/akamai/cli/releases>

Ensure the CLI binary is in your \$PATH on the build server:

```
> which akamai
/usr/local/bin/akamai
```

Install the 'property-manager' package:

```
> akamai install property-manager
Attempting to fetch command from
https://github.com/akamai/cli-property-manager.git..... [OK]
Installing..... [OK]
```

Install the 'auth' package

```
> akamai install auth
Attempting to fetch command from
https://github.com/akamai/cli-auth.git..... [OK]
Installing..... [OK]
```

Install the 'cps' package:

```
> akamai install cps
Attempting to fetch command from
https://github.com/akamai/cli-cps.git..... [OK]
Installing..... [OK]
```

Verify the Installation

Verify the property-manager (denoted as 'pipeline' and 'snippets'), auth, and cps packages are properly installed:

```
> akamai list
```

Installed Commands:

```
auth
  Interface for Akamai Edgegrid Authentication
pipeline (aliases: pl, pipeline, pd, proddeploy)
```

```
Akamai Pipeline for DevOps
snippets (aliases: pm, property-manager)
Property Manager CLI for DevOps
cps (alias: certs)
Access Certificate Provisioning System (CPS) Information
```

Verify your credentials using the 'auth' CLI package. You should see 'read-write' authorizations for 'papi' (in addition to other auth grants you may have provisioned).

```
> akamai auth verify
Credential Name: dmcallis-a2s
-----
Created 2019-04-08T20:11:05Z by dmcallis
Updated 2019-04-08T20:11:05Z by dmcallis
Activated 2019-04-08T20:11:05Z by dmcallis
Grants:
  papi : readwrite
  contract-api : read
  cps : readwrite
```

Pipeline Preparation

In order to create a new pipeline using existing environments, we'll need to identify the following information:

1. The propertyIDs of each existing target property we wish to manage using our pipeline.
2. The contract and group IDs associated with the target properties.
3. The latest version number of each target property.
 - a. Note: ensure a non-active version has been created and saved for each target property within the intended pipeline
4. The hostname and edge hostname details for each target property.
5. The CPS enrollment IDs for each certificate associated with the edge hostnames for the target properties.

Identify target property info

Perform each of the following steps for each target environment.

Retrieve property details:


```
> akamai pipeline search [full property name]
```

```
> akamai pipeline search devopstest-eph.gcs.com
```

"Account ID"	"Contract ID"	"Asset ID"	"Group ID"	"Property ID"	"Property Name"	"Property Version"	"Updated By User"	"Update Date"	"Production Status"	"Staging Status"
"act_B-C-1ED34DK"	"ctr_C-1ED34DY"	"aid_10515753"	"grp_63802"	"prp_423579"	"devopstest-eph.gcs.com"	29	"dmcallis"	"2019-04-08T19:44:01Z"	"INACTIVE"	"INACTIVE"

Take note of the following information:

Element	Description
Property ID	<p>Find the value of the 'Property ID' column in the output table. Omitting the leading '<i>prp_</i>' string from the propertyId element returned. The numeric value only will be used in our pipeline configuration.</p> <p>Example</p> <p>423579</p>
Contract ID	<p>The value of the 'Contract ID' column in the output table. Omitting the leading '<i>ctr_</i>' string from the contractId element returned. The numeric value only will be used in a later discovery step.</p> <p>Example</p> <p>C-1ED34DY</p>
Group ID	<p>The value of the 'Group ID' column in the output table. Omitting the leading '<i>grp_</i>' string from the contractId element returned. The numeric value only will be used in a later discovery step.</p> <p>Example</p> <p>63802</p>
Property Version	<p>The latest version of the target property, depicted as the value of the 'Property Version' column in the output table.</p>

	Example 29
--	--------------------------

Identify the hostname settings for each target property. This is done by issuing a command to list all hostnames for the specific Contract + Group which contain the target properties:

> akamai pipeline leh -g [group id] -c [contract id]

> akamai pipeline leh -g 63802 -c C-1ED34DY

"ID"	"Prefix"	"Suffix"	"IP Version Behavior"	"Secure"	"EdgeHostname Domain"
"ehn_1027034"	"projectwikiwiki.com"	"edgesuite.net"	"IPV4"	false	"projectwikiwiki.com.edgesuite.net"
"ehn_1053007"	"feo1.projectwikiwiki.com"	"edgesuite.net"	"IPV4"	false	"feo1.projectwikiwiki.com.edgesuite.net"
"ehn_1053008"	"feo2.projectwikiwiki.com"	"edgesuite.net"	"IPV4"	false	"feo2.projectwikiwiki.com.edgesuite.net"
"ehn_1207967"	"1.shrd.shana.com"	"edgesuite.net"	"IPV4"	false	"1.shrd.shana.com.edgesuite.net"
"ehn_1207968"	"1.shrd.animalshelter.org"	"edgesuite.net"	"IPV4"	false	"1.shrd.animalshelter.org.edgesuite.net"

Take note of the following information, for each target property:

Element	Description
Property Hostname	The values of the 'Prefix' column in the table output. Example devopstest-eph.gcs.com
Edge Hostname	The values of the 'EdgeHostname Domain' column in the table output. Example devopstest-eph.gcs.com.edgekey.net
Edge Hostname ID	The value of the 'ID' column in the output table. Omitting the leading 'ehn_' string from the contractId element returned. The numeric value only will be used. Example 2744098

For each secure edge hostname (*.edgekey.net), we'll need to identify the corresponding Akamai Certificate Provisioning System (CPS) enrollment ID. To find all enrollment IDs, we'll leverage the Akamai CPS CLI (installed above).

First, setup the CPS CLI:

```
> akamai cps setup
Trying to get contract details from [cps] section of ~/.edgerc file
```

```
Processing Enrollments for contract: C-1ED34DY
14 total enrollments found.
```

```
Enrollments details are stored in "setup/enrollments.json".
Run 'list' to see all enrollments.
```

Next, list the enrollments:

```
> akamai cps list
```

Enrollment ID	Common Name (SAN Count)	Certificate Type	*In-Progress*	Test on Staging First
12345	*.gcs.com	ov wildcard	No	No

Note down the enrollment ID associated with your target property hostnames.

Create a spreadsheet or table containing the configuration elements listed above, mapped to the pipeline stages you intend to manage. Note- each of your target properties will likely have multiple hostname / edge hostname combinations. Be sure to denote the proper mapping of hostnames and their associated CPS enrollment ID and Edge Hostname ID below.

Stage	Property Name	Property ID	Version	Property Hostnames	Edge Hostnames	CPS Enrollment ID	Edge Hostname ID
dev	devopstest-eph.gcs.com	423579	29	devopstest-eph.gcs.com	devopstest-eph.gcs.com.edgekey.net	12345	2744098
				images.devopstest-eph.gcs.com	devopstest-eph.gcs.com.edgekey.net	12345	2744099
prod		

Pipeline Configuration

Create a new pipeline project from a base property

First, we'll create the initial pipeline project using the following syntax:

```
> akamai pipeline new-pipeline -p [pipeline name] -e [template  
property] --variable-mode user-var-value [env1 env2]
```

Argument	Description
-p	<p>The name of the pipeline.</p> <p>The name provided will be used in all subsequent pipeline operations. This argument also defines the name of the directory containing the pipeline project.</p>
-e	<p>The name of the template property, as it appears in Luna Property Manager.</p> <p>This template property will be used as the basis for the delivery configuration rule tree (match criteria, and behaviors) for the pipeline project.</p>
--variable-mode	<p>This argument instructs the pipeline to source in existing property manager variables from the template property, when creating the initial project metadata.</p>
stage	<p>The names of the initial pipeline stages (must be at least two values).</p> <p>Note: it is recommended to only provide 2 generic names during this step to satisfy the minimum requirements. The pipeline CLI will automatically create environments using these names, which will subsequently need to be deleted in the event the pipeline will be</p>

	used to manage existing environments.
--	---------------------------------------

```
> akamai pipeline new-pipeline -p mysite -e devopstest-eph.gcs.com
--variable-mode user-var-value dev prod
```

Verify the pipeline structure

```
> akamai pipeline -p mysite lstat
```

"Environment Name"	"dev"	"prod"
"Property Name"	"dev.mysite"	"prod.mysite"
"Latest Version"	31	1
"Production Version"	"N/A"	"N/A"
"Staging Version"	30	"N/A"
"Rule Format"	"v2015-08-08"	"v2017-06-19"

Verify the pipeline project structure. A directory will be created within the current working directory the 'new-pipeline' command was issued in. For example, if you issued the command in '/home/user/workspace/' the pipeline project and associated artifacts would be created in '/home/user/workspace/[pipeline name]'

```
> ls -ltr
total 4144
drwxr-xr-x  8 dmcallis  600      256 Apr 26 12:11 mysite
-rw-r--r--  1 dmcallis  600 1531323 Apr 26 12:21 devops.log
```

Please review the [Pipeline Project Overview](#) section for a detailed explanation on the project directory structure.

Create pipeline target environments

For each target environment, create a folder within the **/environments** directory which will contain the stage configuration artifacts. Do this by recursively copying any existing stage directory to the intended target stage name.

```
> cd mysite/environments/  
> cp -R dev/ stage
```

Modify the pipeline environment manifest (**/projectInfo.json**), adding each stage created in the previous step.

```
{  
  "productId": "prd_SPM",  
  "contractId": "ctr_C-1ED34DY",  
  "groupIds": [  
    63802  
  ],  
  "environments": [  
    "dev",  
    "prod",  
    "stage"  
  ],  
  "version": "0.4.0",  
  "isSecure": true,  
  "edgeGridConfig": {  
    "section": "papi"  
  },  
  "name": "mysite"  
}
```

Verify the stages have been properly identified by the pipeline CLI, by executing the following command:

```
> akamai pipeline -p [pipeline name] lstat  
  
> akamai pipeline -p mysite lstat
```

"Environment Name"	"dev"	"prod"	"stage"
"Property Name"	"dev.mysite"	"prod.mysite"	"stage.mysite"
"Latest Version"	31	1	29
"Production Version"	"N/A"	"N/A"	"N/A"
"Staging Version"	30	"N/A"	29
"Rule Format"	"v2015-08-08"	"v2017-06-19"	"v2017-06-19"

The 'lstat' command should return no error, and describe all pipeline stages manually created in the previous step. Note- due to the way the pipeline CLI describes environment stages, the 'Property Name' depicted in the table above will not reflect the actual property names used in the pipeline going forward. It's important to take note of this when operating the pipeline later.

Modify the pipeline environment definitions

For each of the pipeline stages defined in the previous step, perform the following operations. Each operation will be performed against pipeline configuration files within the /environments/(stage)/ subdirectory.

Repoint stages to target properties

In the (pipeline)/environments/(stage)/ directory, edit the 'envInfo.json' file. Refer to the table/spreadsheet created in step [Identify target property info](#)

Perform the following modifications:

1. Change the value of 'propertyName' to the property name of the existing target property.
2. Change the value of 'propertyId' to the property Id value for the existing target property.
3. Change the value of 'latestVersionInfo[propertyVersion]' to the property

Save and exit the file.

Update hostname settings for target properties

In the (pipeline)/environments/(stage)/ directory, edit the 'hostnames.json' file. Refer to the table/spreadsheet created in step '[Identify target property info](#)'.

Perform the following modifications:

1. Update the value of 'cnameFrom' to the existing property hostname of the target property.
2. Update the value of 'cnameTo' to the existing edge hostname of the target property.

Add an additional 'hostname' block for each hostname in the target configuration. For example (using the sample spreadsheet), our target property had two hostnames:

Property Hostnames	Edge Hostnames	CPS Enrollment ID	Edge Hostname ID
devopstest-eph.gcs.com	devopstest-eph.gcs.com.edgesuite.net	12345	2744098
images.devopstest-eph.gcs.com	devopstest-eph.gcs.com.edgesuite.net	12345	2744099

Our hostname.json file would contain two hostname blocks:

```
[
  {
    "cnameFrom": "devopstest-eph.gcs.com",
    "cnameTo": "devopstest-eph.edgekey.net",
    "certEnrollmentId": "12345",
    "cnameType": "EDGE_HOSTNAME",
    "edgeHostnameId": 2744098
  },
  {
    "cnameFrom": "images.devopstest-eph.gcs.com",
    "cnameTo": "devopstest-eph.edgekey.net",
    "certEnrollmentId": "12345",
    "cnameType": "EDGE_HOSTNAME",
    "edgeHostnameId": 2744099
  }
]
```



Delete the initial pipeline environments in Luna

This is an optional step.

The process of creating the initial pipeline (step: 'Create a new pipeline project from a base property') automatically created two Akamai properties. Assuming you wish to only manage existing Akamai properties using the pipeline CLI framework, these can be removed as they will be no longer needed.

To do this, simply delete the properties within Luna Control Center. Search for the properties created initially by the pipeline, click on the gear icon to the right of the property, and select 'Delete Property':



Type	Property Name	Staging	Production	Hostnames	Action
	dev.amd.pd.demo	12	11	dev.amd.pd.demo	
	prod.amd.pd.demo	4	4	prod.amd.pd.demo	<div>Delete Property Clone Property</div>

Next, add notes and click the 'Delete Property' button.

Configure Rule Tree and Stage Values

Pipeline Project Overview

Following the creation of the pipeline project (described in the [Pipeline Configuration](#) section), the following project directory structure will be created:

```
pipeline_name/  
  projectInfo.json  
  /cache  
  /dist  
  /environments
```

```

variableDefinitions.json
/environment1_name
    envInfo.json
    hostnames.json
    variables.json
    ...
/environment2_name
    envInfo.json
    hostnames.json
    variables.json
    ...

/templates
    main.json
    compression.json
    ...
    static.json

```

Folder / Artifact	Description
projectInfo.json	Describes the pipeline project configuration: name, stages, contract, group, and edgerc section.
cache/	<p>Contains various logs and ephemeral artifacts used by the pipeline project during normal operation.</p> <p>Nothing in this folder should ever be modified.</p>
dist/	<p>Contains the stage-specific metadata artifacts, which are an output of the pipeline build process (token replacement + schema validation).</p> <p>The artifacts contained within this folder can be deployed via APIs/CLI to target environments without using the pipeline CLI itself if necessary.</p>
templates/	<p>Contains the tokenized JSON metadata fragments which will be used as a basis for all stage-specific metadata.</p> <p>All development of new Akamai delivery configuration features (rules/behaviors) will occur in this folder.</p> <p>During the pipeline build process, the metadata fragments</p>

	contained within this folder will have their defined tokens replaced with stage-specific values, and be concatenated into a single cohesive artifact for later deployment (stored within the dist/ folder).
environments/	<p>Contains the stage-specific configuration elements (within stage-specific directories):</p> <ul style="list-style-type: none"> - Stage hostnames - Stage variables / tokens - Variable / Token Manifest

Stage Configuration

Each pipeline stage will require its own sub-folder within the (pipeline)/environments/ folder. The folder structure and associated artifacts can be described as follows:

Artifact	Description
variableDefinitions.json	<p>Variable / Token Manifest</p> <p>This file defines all the tokens/variables used within the pipeline project, that will be referenced in the metadata fragments within templates/.</p> <p>The scope of the variable (global or stage-specific) is defined using this artifact (see Pipeline Tokens Overview).</p>
(stage)/variables.json	Contains stage-specific values for tokens/variables used within the pipeline project.
(stage)/hostnames.json	Used to configure and define hostnames which are associated with the stage property (see Update hostname settings for target environments section for an overview of how to add/edit hostnames).
(stage)/envInfo.json	Specifies the stage property details (name, version, etc).
(stage)/devops.log	Stage-specific CLI log containing information on the build, deployment, and activate processes.

Pipeline Tokens Overview

All metadata settings can be initialized with special tokens which are mapped to values defined in the variables.json file for each stage (managed within the (pipeline)/environments/(stage)/ directory). Tokens are defined using the following syntax:

```
"${env.<variableName>}"
```

During the build process, the Akamai pipeline CLI will replace the token values with either a **global** value or an **environment-specific** value, depending on their scope. Consider the following table to determine the scope a specific variable should be initialized with:

If an attribute has...	Then...
the same value across all environments and is already in the variableDefinitions.json file.	<p>Provide the default value in variableDefinitions.json.</p> <p>Note: You can still override the default value in an environment's variables.json file.</p>
different values across environments and is already in the variableDefinitions.json file.	<p>Set the default to null in variableDefinitions.json and add the environment-specific value to each variables.json file.</p>
different values across environments and does not exist in the variableDefinitions.json file.	<ol style="list-style-type: none">1. Parameterize it inside your template snippets using "\${env.<variableName>}" For example: "ttl": "\${env.ttl}"2. Add it to variableDefinitions.json and set it to null. You can set the type to anything you choose.3. Add it to your environment-specific variables.json files and set the individual values.

Complex Tokens

During the build process, the pipeline framework will find all tokens within the template metadata and replace them with their corresponding global or environment-specific values. The pipeline framework performs a complete replacement with the values defined in the variable definition files, so complex JSON can be used as a value for any token (the token values need not be simple strings/booleans/integers).

Complex JSON tokens must be initialized as single-value arrays, with the key being the name of the token, and the singular value being the complex JSON structure. Consider the following example of a complex JSON token used within the context of a Last Mile Acceleration (gzip) content-type match:

```
"criteria": [
  {
    "name": "contentType",
    "options": {
      "matchCaseSensitive": false,
      "matchOperator": "IS_ONE_OF",
      "matchWildcard": true,
      "values": ${env.gzipContentTypes}
    }
  }
]
```

The `gzipContentTypes` token can be initialized as a JSON list within either the global (`variableDefinitions.json`) or environment-specific (`variables.json`) manifest:

```
gzipContentType: [ "text/*",
  "application/javascript",
  "application/x-javascript",
  "application/x-javascript*",
  "application/json"]
```

Tokens support even more complex JSON structures (such as entire snippets of JSON metadata), as long as they are defined as a single-element JSON array.

For instance, tokenizing complex origin certificate behaviors along with their settings as a complex JSON token is a recommended way to support different certificates between pipeline stages. Consider the following tokenized 'origin' behavior which relies on custom pinned certificates (highlighted below):

```

"name": "origin",
"options": {
  "originType": "CUSTOMER",
  "hostname": "${env.origin}",
  "forwardHostHeader": "REQUEST_HOST_HEADER",
  "cacheKeyHostname": "ORIGIN_HOSTNAME",
  "compress": true,
  "enableTrueClientIp": true,
  "trueClientIpHeader": "True-Client-IP",
  "trueClientIpClientSetting": true,
  "httpPort": 80,
  "originCertificate": "",
  "verificationMode": "CUSTOM",
  "ports": "",
  "httpsPort": 443,
  "originSni": false,
  "customValidCnValues": "${env.origin_cn_values}",
  "originCertsToHonor": "COMBO",
  "standardCertificateAuthorities": [
    "akamai-permissive"
  ],
  "customCertificateAuthorities": "${env.origin_authority_certs}",
  "customCertificates": "${env.origin_leaf_certs}"
}

```

Note how all the tokenized behavior values are those which change between stages (origin hostname, certificate common names, certificate authorities, and certificates). The value of the `origin_leaf_certs` token would be defined in the stage **variables.json** file as the following complex JSON object:

```

"origin_leaf_certs": [
  {
    "subjectCN": "xyz.example.com",
    "subjectAlternativeNames": [
      "xyz.example.com"
    ],
    "subjectRDNs": {
      "ST": "XYZ",
      "SERIALNUMBER": "XYZ",
      "C": "XYZ",
      "L": "XYZ",
      "SN": "XYZ",
      "OU": "XYZ",

```

```

        "O": "XYZ",
        "CN": "XYZ"
    },
    "issuerRDNs": {
        "C": "US",
        "O": "GeoTrust, Inc.",
        "CN": "GeoTrust SSL CA"
    },
    "notBefore": 1287318021000,
    "notAfter": 1355864420000,
    "sigAlgName": "SHA1WITHRSA",
    "publicKey": "XYZ",
    "publicKeyAlgorithm": "RSA",
    "publicKeyFormat": "X.509",
    "serialNumber": XYZ,
    "version": 3,
    "sha1Fingerprint": "XYZ",
    "pemEncodedCert": "-----BEGIN CERTIFICATE-----\nXYZ\n-----END
CERTIFICATE-----\n",
    "canBeLeaf": true,
    "canBeCA": false,
    "selfSigned": false
}
]

```

Tokenize the Template Metadata

The process of creating a new pipeline from an environment template will perform two initial steps:

1. Create JSON metadata snippets, one for each rule node in the template property, within the templates/ folder.
2. Tokenize a single CP code, origin hostname, and sure route test object behavior value.

In order to ensure portability across each of the environment stages, specific metadata settings need to be tokenized further. The exact number of tokens, and their values, will depend on the target environments and pipeline requirements. Review each metadata fragment within the templates/ folder, and identify specific settings which will vary from environment to environment (or, will need to be changed in later releases).

Within each metadata fragment, replace hard-coded values with token place holders using the syntax “`${env.<variableName>}`”. For each such value, define the scope and value of the token by following the process below.

Define Scope and Values for Tokens

Every token referenced in the metadata fragments must be defined in both the `variableDefinitions.json` (token manifest) and `environments/(stage)/variables.json`.

First, define the token within the token manifest. For example, defining a token for a CP Code in the template metadata (JSON snippet within `templates/`):

CP Code Behavior in templates/

```
{
  "name": "cpCode",
  "options": {
    "value": {
      "id": 123456
    }
  }
}
```

Tokenize the CP Code value (id:)

```
{
  "name": "cpCode",
  "options": {
    "value": {
      "id": "${env.failoverCpCode}"
    }
  }
}
```

Define the token in the Token Manifest (`environments/variableDefinitions.json`)

Add a new entry to the `variableDefinitions.json` JSON structure, referencing the new variable name defined in the metadata JSON snippet.

```
{
  "definitions": {
    "originHostname": {
      "type": "hostname",
      "default": "origin.akshayranganath.com"
    },
    "failoverCpCode": {
```



```

        "type": "cpCode",
        "default": 123456
    },
    "sureRouteTestObject": {
        "type": "url",
        "default": "/akamai/sure-route-test-object.html"
    }
}

```

As outlined in the [Pipeline Tokens Overview](#) section, the token scope is defined here. If the token contains a 'default' value, it will be considered global (the value of the 'default' attribute will be used across all stages), if it's defined as 'null' it will be considered stage-specific (where the value for this token defined in environments/(stage)/variables.json will be used).

Note: the 'type' attribute value can be any arbitrary string.

Define the token for each environment stage (environments/(stage)/variables.json)

Expanding on the above example, add a value for `failoverCpCode` within the variables.json file for each stage. If 10 stages are defined, you will need to add this token to the variables.json file within each stage directory.

```

{
    "originHostname": null,
    "failoverCpCode": null,
    "sureRouteTestObject": null
}

```

Global token value example

```

{
    "originHostname": null,
    "failoverCpCode": 123456,
    "sureRouteTestObject": null
}

```

Stage-Specific token value example

If the value of the token is intended to be stage-specific (has a unique value for each environment stage), define the unique value within each stage **variable.json** file. If the value of

the token is intended to be global (the same value for every environment stage), initialize the value as 'null' within each stage **variable.json** file.

Guidance on identifying token values

When configuring stage-specific token values, especially when using complex JSON tokens, it's helpful to reference existing JSON metadata for your existing stage properties. This can be done either by pulling the configuration JSON metadata using the Akamai property-manager CLI, property-manager API, or through Luna itself.

Within Luna, navigate to a stage you wish to examine. Navigate to the specific rule within the property manager UI, and click on 'View Rule JSON'

The screenshot displays the 'Property Configuration Settings' interface in Akamai Luna. On the left, a sidebar lists various rule types: 'Default Rule', 'CORS Logic', 'CORS Origin Whitelist' (selected), 'CORS Origin Redirect', 'Caching', and 'Whitelist by Location'. The main panel shows the configuration for the 'CORS Origin Whitelist' rule. At the top right of this panel, the 'View Rule JSON' button is highlighted with a red rectangular box. Other visible elements include a 'Push To Test Ghost' button, 'Edit', 'Review', 'View JSON', and 'View XML' buttons. The configuration area includes a comment field, a 'Criteria' section set to 'Match All', an 'If' condition block (Request Header 'Origin' exists), and a 'Behaviors' section with a 'Set Variable' rule (Variable: PMUSER_CORS_ORIGIN, Create Value From: Extract, Get Data From: Request Header 'Origin', Operation: None).

This will provide a JSON representation of the specific behavior or match condition values you can use to define your environment-specific token values (or complex tokens):

▼ Property Configuration Settings

⬆️ ⬇️ ⬆️

Add Rule ▼

Default Rule

CORS Logic

CORS Origin Whitelist

CORS Origin Redirect

Caching

Whitelist by Location

Push To Test Ghost

Edit

Review

View JSON

View XML

CORS Origin Whitelist

Download Rule JSON Edit Rule

Below is the JSON representation of the CORS Origin Whitelist sub-rule, from the rule tree of the property version. You can only view the sub-rule JSON. Click the **View JSON** button or the **Default Rule JSON** button to view the entire rule tree, which can be used with PAPI to make updates through the Property Manager API

[See the PAPI documentation for rule tree management.](#)

```
{
  "name": "CORS Origin Whitelist",
  "children": [],
  "behaviors": [
    {
      "name": "setVariable",
      "options": {
        "valueSource": "EXTRACT",
        "transform": "NONE",
        "variableName": "PMUSER_CORS_ORIGIN",
        "extractLocation": "CLIENT_REQUEST_HEADER",
        "headerName": "Origin"
      }
    },
    {
      "name": "modifyOutgoingResponseHeader",
      "options": {
        "action": "ADD",
        "standardAddHeaderName": "ACCESS_CONTROL_ALLOW_ORIGIN",
        "headerValue": "{{user.PMUSER_CORS_ORIGIN}}"
      }
    }
  ],
  "criteria": [
    {
      "name": "requestHeader",
      "options": {
        "headerName": "Origin",
        "matchOperator": "EXISTS",
        "matchWildcardName": false
      }
    }
  ]
}
```

Simply cut and paste the relevant JSON value (or object) from this view and paste it into your project appropriately.

Further Federate the Metadata

This is an optional step.

The templates/ directory contains the generic tokenized JSON metadata snippets which are used to build stage-specific artifacts for each environment. The 'main.json' is the only required snippet that must be associated with your project, and it should contain the 'default' rule defined within your delivery configuration:

```
> vi main.json
```

```
{
  "rules": {
    "name": "default",
    "children": [
      "#include:performance.json",

```

```
"#include:Offload.json",  
"#include:Homepage_Caching.json",  
"#include:Mobile_Redirect.json",  
"#include:Real_User_Monitoring.json",  
"#include:X-Frame-Options.json",  
"#include:Deny_Specific_Referrer.json",  
"#include:Deny_Specific_Referrer_2.json",  
"#include:Deny_by_Location.json"
```

Metadata snippets are referenced using the following syntax:

```
#include:snippet.json
```

Snippet include statements are evaluated in the order in which they are defined in the metadata, for example:

```
#include:snippet1.json  
#include:snippet2.json
```

The above include sequence would include `snippet1.json`, then `snippet2.json` in strict sequence.

Nested Includes

Includes can also be embedded within snippets to denote child rules or any other metadata fragment you wish. For example, `snippet1.json` can reference additional snippets via includes to support federated snippets for each child rule. Consider the following example- this rule tree contains a child rule (CORS Logic) with 2 additional child rules (CORS Origin Whitelist and CORS Origin Redirect):

Default Rule
— CORS Logic
CORS Origin Whitelist
CORS Origin Redirect
Caching
Whitelist by Location

Using includes, we can manage each child rule as its own JSON metadata snippet:

```
"rules": {
  "name": "default",
  "children": [
    "#include:CORS_Logic.json",
    "#include:caching.json",
    "#include:whitelist.json"
  ],
}
```

main.js include example

Within the `CORS_Logic.json` snippet, we include each child rule:

```
{
  "name": "CORS Logic",
  "children": [
```

```
    "#include: cors_whitelist.json",
    "#include: cors_redirect.json"
]
```

CORS_Logic.json example

Snippet Subdirectories

If desired, pipeline snippets can be included in subdirectories beneath the parent **templates** pipeline project directory. Extending on the above example, the parent snippet **CORS_Logic.json** and two child rule snippets **cors_whitelist.json** and **cors_redirect.json** can be managed in a subdirectory **templates/CORS** using the following include syntax:

```
"children": [
    ...
    "#include: CORS/CORS_Logic.json",
    "#include: CORS/cors_whitelist.json",
    "#include: CORS/cors_redirect.json",
    ...
]
```

Pipeline Operation

All operations need to be performed one directory below the pipeline project directory (not within the project directory itself). So if your pipeline is managed within `/home/workspace/mypipeline`, the pipeline operation needs to be executed from `'/home/workspace'`.

The pipeline CLI describes each of its commands and subcommands using the `'help'` argument:

```
> akamai help pipeline
```

```

Usage: akamai pl [options] [command]

Akamai Pipeline. The command assumes that your current working directory is the pipeline space under which all pipelines reside

Options:
  -V, --version                output the version number
  -v, --verbose                Verbose output, show logging on stdout
  -s, --section [section]     Section name representing Client ID in .edgerc file, defaults to "credentials"
  -f, --format [format]       Select output format, allowed values are 'json' or 'table'
  -h, --help                  output usage information

Commands:
  new-pipeline|np [options] [environments...] Create a new pipeline with provided attributes. This will also create one property for each environment.
  set-default|sd [options]                   Set the default pipeline and or the default section name from .edgerc
  show-defaults|sf                           Show default settings for this workspace
  merge|m [options] <environment>           Merge template json and variable values into a PM/PAPI ruletree JSON document, stored in dist folder in the current pipeline folder
  search|s <name>                           Search for properties by name
  set-prefixes|sp <useprefix>                Set or unset use of prefixes [true|false] for current user credentials and setup_CONTROL_ALLOW_ORIGIN
  set-ruleformat|srf <ruleformat>           Set ruleformat for current user credentials and setup
  list-contracts|lc                         List contracts available to current user credentials and setup
  list-products|lp [options]                List products available under provided contract ID and client ID available to current user credentials and setup
  list-groups|lg                            List groups available to current user credentials and setup
  list-cpcodes|lcp [options]                List cpodes available to current user credentials and setup
  show-ruletree|sr [options] <environment> Fetch latest version of property rule tree for provided environment
  save|sv [options] <environment>           Save rule tree and hostnames for provided environment. Edge hostnames are also created if needed.
  list-edgehostnames|leh [options]          List edge hostnames available to current user credentials and setup (this could be a long list).
  list-status|lstat [options]               Show status of pipeline
  promote|pm [options] [targetEnvironment] Promote (activate) an environment. This command also executes the merge and save commands mentioned above by default.
  check-promotion-status|cs [options] <environment> Check status of promotion (activation) of an environment.
  help [cmd]                               display help for [cmd]

```

Subcommand help:

```
> akamai pipeline help (subcommand)
```

```
> akamai pipeline help merge
```

```

Usage: mergelm [options] <environment>

Merge template json and variable values into a PM/PAPI ruletree JSON document, stored in dist folder in the current pipeline folder

Options:
  -p, --pipeline [pipelineName] Pipeline name
  -n, --no-validate                Don't call validation end point. Just run merge.
  -h, --help                      output usage information

```

Display Pipeline Status

To display the current pipeline status (the active version):

```
> akamai pipeline -p (pipeline) lstat
```

"Environment Name"	"dev"	"prod"	"stage"
"Property Name"	"dev.mysite"	"prod.mysite"	"stage.mysite"
"Latest Version"	31	1	29
"Production Version"	"N/A"	"N/A"	"N/A"
"Staging Version"	30	"N/A"	29
"Rule Format"	"v2015-08-08"	"v2017-06-19"	"v2017-06-19"

Note: the value of 'Property Name' should be ignored, these are assumed by the pipeline CLI. The actual property name used for the stage is defined within the environments/(stage)/envInfo.json file.

Build, Deploy and Activate a Pipeline Stage

The 'promote' facility will perform the following steps via a single command:

1. Concatenate all JSON metadata snippets contained within templates/.
2. Replace all tokens with their global or stage-specific values.
3. Perform JSON validation of the transformed snippets.
4. Perform schema validation of the transformed snippets.

If the JSON is incorrect, or the metadata does not conform to the product schema, the CLI will produce an exception with an insight into the violation. Example initializing a behavior with an integer when a boolean is expected:


```

There are validation errors:
[
  {
    "location": {
      "template": "templates/main.json",
      "variables": [],
      "location": "rules/behaviors/2/options/enabled",
      "value": 12345
    },
    "schemaLocation": "/definitions/catalog/behaviors/realUserMonitoring/properties/options/properties/enabled",
    "detail": "instance value (12345) not found in enum (possible values: [false,true])",
    "foundValue": 12345,
    "allowedValues": [
      false,
      true
    ]
  }
]

```

Promote to an environment stage using the following syntax:

```
> akamai pipeline promote -p (pipeline name) (stage) -n (production or staging) -e employee@example.com
```

```
> akamai pipeline promote -p mysite dev -n staging -e dmcallis@akamai.com
```

Following activations are now pending:

"Environment"	"Network"	"Activation Id"
"dev"	"STAGING"	7159089

Build a stage specific artifact (but not deploy it)

If you wish to generate a stage-specific JSON release artifact, which could be deployed outside of the pipeline CLI (using Akamai Property Manager API / CLI), the 'merge' facility performs just the build and validation steps without activating it on any network.

Perform a merge using the following syntax:

```
> akamai pipeline -p (pipeline) merge (stage)
```

```
> akamai pipeline -p mysite merge dev
```

"Action"	"Result"
"changes detected"	"no"
"rule tree stored in"	"/Users/dmcallis/workspace/pipeline-demo/mysite/dist/dev.mysite.papi.json"
"hash"	"88afea1bc1c86e17ec614e9f520608746fac8cc98813c42a9e50fd9ee0abae4"
"validation performed"	"no"
"validation warnings"	"yes"
"validation errors"	"no"

The built and validated artifact will be located in the dist/ folder.

Check Stage Activation Status

Check activation status for an environment stage using the following syntax:

```
> akamai pipeline -p (pipeline name) cs (stage)
```

```
> akamai pipeline -p mysite cs dev
```

Activation status report:			
"Environment"	"Network"	"Activation Id"	"Status"
"dev"	"STAGING"	"atv_7159089"	"ACTIVE"

Reconciling Pipeline State with Luna

As described in the Pipeline Design section above, introducing any change outside of the pipeline CLI project framework (such as via Luna directly) will result in drift between the actual Akamai-managed property state and the pipeline project itself. Such a condition will require manual reconciliation of the pipeline project, specifically:

1. **The pipeline state** - the property versions, network activation status, and hostnames.
2. **The rule tree metadata** - the rules, behaviors, and associated settings.

Reconciling Pipeline State

The Akamai pipeline CLI manages the state of each environment stage through the `envInfo.json` file contained within the pipeline project. The pipeline project structure is described in the section above titled [Pipeline Project Overview](#).

The easiest way to perform the state reconciliation is using the snippets CLI facility (which should already be installed, as its included with the pipeline / property-manager CLI bundle). The snippets CLI can retrieve a property rule tree and state, describing it using the same schema as pipeline for a specific stage. To retrieve this state and rule tree snippets, issue the following `import` command:

```
> akamai snippets import -p (property name)
```

```
> akamai snippets import -p dev.mysite
```

```
Importing and creating local files for dev.epic-pl-demo from Property Manager...
Imported dev.epic-pl-demo. The latest version is: v32
```

A folder containing the snippets and environment state will be created with the following artifacts:

```
dev.mysite/
  .../config-snippets/
  .../envInfo.json
  .../hostnames.json
```

Copy the `hostnames.json` file generated from snippets to the pipeline `environments/(stage)/` folder, overwriting the pipeline managed version.

Next, reconcile the stage property state by copying specific JSON nodes within the `envInfo.json` (generated from the snippets import command above) within the stage `envInfo.json` file managed within the pipeline project `environments/(stage)/envInfo.json`.

There are three relevant sections which need to be manually copied over, outlined below. Note: copying and overwriting the entire contents of the pipeline `envInfo.json` with the snippets `envInfo.json` file may result in unexpected behavior- so its advised to port over on the following nodes within the JSON document to the pipeline `envInfo.json` file:

```
"latestVersionInfo": {
  ...
```

```
}

"activeIn_PRODUCTION_Info": {
    ...
}

"activeIn_STAGING_Info": {
    ...
}
```

Reconciling Pipeline Metadata

Refer to the section [Guidance on identifying token values](#) above for instructions on reconciling new snippets and behaviors within your pipeline project. Make sure that all new rule tree snippets are tokenized once they are integrated. Rule tree tokenization is explained in the [Tokenize the Template Metadata](#) section above.

It is advisable to compare the manually modified rule tree version with the new pipeline reconciled rule tree version using the Luna version compare feature to ensure all changes have been properly accounted for.