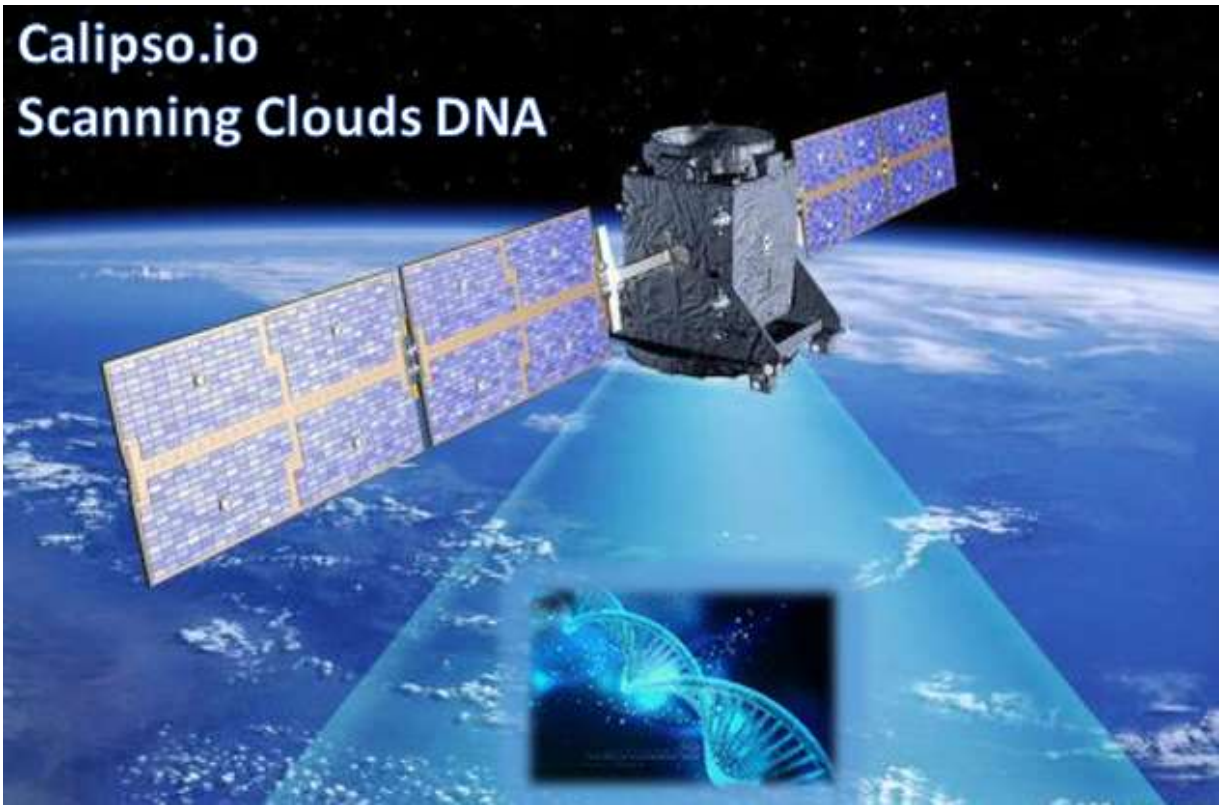
Calipso
Developer Guide



Project “Calipso” tries to illuminate complex virtual networking with real time operational state visibility for large and highly distributed Virtual Infrastructure Management (VIM).

We believe that Stability is driven by accurate Visibility.

Calipso provides visible insights using smart discovery and virtual topological representation in graphs, with monitoring per object in the graph inventory to reduce error vectors and troubleshooting, maintenance cycles for VIM operators and administrators.

Table of Contents

1 Project architecture

Calipso comprises two major parts: application and UI. We'll focus on the former in this developer guide.

Current project structure is as follows:

- root/
  - app/
    - api/
      - responders/
        - auth/
        - resource/
      - *server.py*
    - config/
      - *events.json*
      - *scanners.json*
    - discover/
      - events/
        - listeners/
          - *default_listener.py*
          - *listener_base.py*
        - handlers/
          - *event_base.py*
          - *event_*.py*
      - fetchers/
        - aci/
        - api/
        - cli/
        - db/
      - *event_manager.py*
      - *scan.py*
      - *scan_manager.py*
    - monitoring/
      - checks/
      - handlers/
        - *monitor.py*
      - setup/
        - *monitoring_setup_manager.py*
    - test/
      - api/
      - event_based_scan/
      - fetch/
      - scan/
    - utils/
  - ui/

1.1 Application structure

1.1.1 'API' package

Calipso API is designed to be used by native and third-party applications that are planning to use Calipso discovery application.

api/responders
This package contains all exposed API endpoint handlers:
auth package contains token management handlers,
resource package contains resource handlers.

server.py
API server startup script. In order for it to work correctly, connection arguments for a Mongo database used by a Calipso application instance are required:

```
-m [MONGO_CONFIG], --mongo_config [MONGO_CONFIG]
                    name of config file with mongo access details
--ldap_config [LDAP_CONFIG]
                    name of the config file with ldap server config
                    details
-l [LOGLEVEL], --loglevel [LOGLEVEL]
                    logging level (default: 'INFO')
-b [BIND], --bind [BIND]
                    binding address of the API server (default
                    127.0.0.1:8000)
-y [INVENTORY], --inventory [INVENTORY]
                    name of inventory collection (default: 'inventory')
-t [TOKEN_LIFETIME], --token-lifetime [TOKEN_LIFETIME]
                    lifetime of the token
```

For detailed reference and endpoints guide, see the API Guide document.

1.1.2 'Discover' package

'Discover' package contains the core Calipso functionality which involves:

- scanning a network topology using a defined suite of scanners (see Scanning concepts, Scanners configuration file structure) that use fetchers to get all needed data on objects of the topology;
- tracking live events that modifies the topology in any way (by adding new object, updating existing or deleting them) using a suite of event handlers and event listeners;
- managing the aforementioned suites using specialized manager scripts (*scan_manager.py* and *event_manager.py*)

1.1.3 'Tests' package

'Tests' package contains unit tests for main Calipso components: API, event handlers, fetchers, scanners and utils.

1.1.4 Other packages

Install

Installation and deployment scripts (with initial data for Calipso database).

Monitoring

Monitoring configurations, checks and handlers (see Monitoring section and Monitoring Guide document).

Utils

Utility modules for app-wide use (inventory manager, mongo access, loggers, etc.).

2 Scanning Guide

2.1 Introduction to scanning

2.1.1 Architecture overview

Calipso backend will scan any OpenStack environment to discover the objects that it is made of, and place the objects it discovered in a MongoDB database.

Following discovery of objects, Calipso will:

Find what links exist between these objects, and save these links to MongoDB as well. For example, it will create a pnic-network link from a pNIC (physical NIC) and the network it is connected to.

Based on user definitions, it will create a 'clique' for each object using the links it previously found. These cliques are later used to present graphs for objects being viewed in the Calipso UI. This is not a clique by graph theory definition, but more like the social definition of clique: a graph of related, interconnected nodes.

OpenStack Scanning is done using the following methods, in order of preference:

1. OpenStack API
2. MySQL DB - fetch any extra detail we can from the infrastructure MySQL DB used by OpenStack
3. CLI - connect by SSH to the hosts in the OpenStack environment to run commands, e.g. ifconfig, that will provide the most in-depth details.

Note: 'environment' in Calipso means a single deployment of OpenStack, possibly containing multiple tenants (projects), hosts and instances (VMs). A single Calipso instance can handle multiple OpenStack environments.
However, we expect that typically Calipso will reside inside an OpenStack control node and will handle just that node's OpenStack environment.

Environment

The Calipso scan script, written in Python, is called scan.py.
It uses Python 3, along with the following libraries:

- pymongo - for MongoDB access
- mysql-connector - For MySQL DB access
- paramiko - for SSH access
- requests - For handling HTTP requests and responses to the OpenStack API
- xmltodict - for handling XML output of CLI commands
- cryptography - used by Paramiko

See Calipso installation guide for environment setup instructions.

Configuration

The configuration for accessing the OpenStack environment, by API, DB or SSH, is saved in the Calipso MongoDB *"environments_config"* collection.

Calipso can work with a remote MongoDB instance, the details of which are read from a configuration file (default: */etc/calipso/mongo.conf*).

The first column is the configuration key while the second is the configuration value, in the case the value is the server host name or IP address.

Other possible keys for MongoDB access:

- port: IP port number
- Other parameters for the PyMongo MongoClient class constructor

Alternate file location can be specified using the CLI -m parameter.

2.1.2 Scanning concepts

DB Schema

Objects are stored in the inventory collection, named *"inventory"* by default, along with the accompanying collections, named by default: *"links", "cliques", "clique_types" and "clique_constraints"*. For development, separate sets of collections can be defined per environment (collection names are created by appending the default collection name to the alternative inventory collection name).

The inventory, links and cliques collections are all designed to work with a multi-environment scenario, so documents are marked with an *"environment"* attribute.

The clique_types collection allows Calipso users (typically administrators) to define how the "clique" graphs are to be defined.

It defines a set of link types to be traversed when an object such as an instance is clicked in the UI (therefore referred to as the focal point). See "Clique Scanning" below. This definition can differ between environments.

Example: for focal point type "instance", the link types are often set to

- instance-vnic
- vnic-vconnector
- vconnector-vedge
- vedge-pnic
- pnic-network

The clique_constraints collection defines a constraint on links traversed for a specific clique when starting from a given focal point.

For example: instance cliques are constrained to a specific network. If we wouldn't have this constraint, the resulting graph would stretch to include objects from neighboring networks that are not really related to the instance.

Hierarchy of Scanning

The initial scanning is done hierarchically, starting from the environment level and discovering lower levels in turn.

Examples:

- Under environment we scan for regions and projects (tenants).
- Under availability zone we have hosts, and under hosts we have instances and host services

The actual scanning order is not always same as the logical hierarchical order of objects, to improve scanning performance.

Some objects are referenced multiple times in the hierarchy. For example, hosts are always in an availability zone, but can also be part of a host aggregate. Such extra references are saved as references to the main object.

Clique Scanning

For creating cliques based on the discovered objects and links, clique types need to be defined for the given environment.

A clique type specifies the list of link types used in building a clique for a specific focal point object type.

For example, it can define that for instance objects we want to have the following link types:

- instance-vnic
- vnic-vconnector
- vconnector-vedge
- vedge-pnic
- pnic-network

As in many cases the same clique types are used, default clique types will be provided with a new Calipso deployment.

Clique creation algorithm

- For each clique type CT:
  - For each focal point object F of the type specified as the clique type focal point type:
    - Create a new clique C
    - Add F to the list of objects included in the clique
    - For each link type X-Y of the link types in CT:
      - Find all the source objects of type x that are already included in the clique
      - For each such source object S:
        - for all links L of type X-Y that have S as their source
          - Add the object T of type Y that is the target in L to the list of objects included in the clique
          - Add L to the list of links in the clique C

2.2 How to run scans

For running environment scans Calipso uses a specialized daemon script called *scan manager*. If Calipso application is deployed in docker containers, scan manager will run inside the *calipso-scan* container.

Scan manager uses MongoDB connection to fetch requests for environment scans and execute them by running a scan script. It also performs extra checks and procedures connected to scan failure/completion, such as marking *environment* as scanned and reporting errors (see details).

Scan script workflow:

1. Loads specific scanners definitions from a predefined metadata file (which can be extended in order to support scanning of new object types).
2. Runs the root scanner and then children scanners recursively (see Hierarchy of scanning)
    a. Scanners do all necessary work to insert objects in *inventory*.
3. Finalizes the scan and publishes successful scan completion.

2.2.1 Scan manager

Scan manager is a script which purpose is to manage the full lifecycle of scans requested through API. It runs indefinitely while:

1. Polling the database (*scans* and *scheduled_scans* collections) for new and scheduled scan requests;
2. Parsing their configurations;
3. Running the scans;
4. Logging the results.

Scan manager can be run in a separate container provided that it has connection to the database and the topology source system.

2.3 Monitoring

Monitoring Subsystem Overview

Calipso monitoring uses Sensu to remotely track actual state of hosts.

A Sensu server is installed as a Docker image along with the other Calipso components.

Remote hosts send check events to the Sensu server.

We use a filtering of events such that the first occurrence of a check is always used, after that cases where status is unchanged are ignored.

When handling a check event, the Calipso Sensu handlers will find the matching Calipso object, and update its status.

We also keep the timestamp of the last status update, along with the full check output.

Setup of checks and handlers code on the server and the remote hosts can be done by Calipso. It is also possible to have this done using another tool, e.g. Ansible or Puppet.

More info is available in Monitoring Guide document.

Package Structure

Monitoring package is divided like this:

1. Checks: these are the actual check scripts that will be run on the hosts;
2. Handlers: the code that does handling of check events;
3. Setup: code for setting up handlers and checks.

3 Events Guide

3.1 Introduction

3.1.1 Events

Events in general sense are any changes to the monitored topology objects that are trackable by Calipso. We currently support subscription to Neutron notification queues for several OpenStack distributions as a source of events.

The two core concepts of working with events are listening to events and event handling, so the main module groups in play are the event listener and event handlers.

3.1.2 Event listeners

An event listener is a module that handles connection to the event source, listening to the new events and routing them to respective event handlers.

An event listener class should be designed to run indefinitely in foreground or background (daemon) while maintaining a connection to the source of events (generally a message queue like RabbitMQ or Apache Kafka). Each incoming event is examined and, if it has the correct format, is routed to the corresponding event handler class. The routing can be accomplished through a dedicated event router class using a metadata file and a metadata parser (see Metadata parsers).

3.1.3 Event handlers

An event handler is a specific class that parses the incoming event payload and performs a certain CUD (Create/Update/Delete) operation on zero or more database objects. Event handler should be independent of the event listener implementation.

3.1.4 Event manager

Event manager is a script which purpose is to manage event listeners. It runs indefinitely and performs the following operations:

1. Starts a process for each valid entry in *environments_config* collection that is scanned (*scanned == true*) and has the *listen* flag set to *true*;
2. Checks the *operational* statuses of event listeners and updating them in *environments_config* collection;
3. Stops the event listeners that no longer qualify for listening (see step 1);

4. Restarts the event listeners that quit unexpectedly;
5. Repeats steps 1-5

Event manager can be run in a separate container provided that it has connection to the database and to all events source systems that event listeners use.

3.1.5 Contribution

You can contribute to Calipso *events* system in several ways:

- create custom event handlers for an existing listener;
- create custom event listeners and reuse existing handlers;
- create custom event handlers and listeners.

See Creating new event handlers and Creating new event listeners for the respective guides.

4 Contribution

This section covers the designed approach to contribution to Calipso.

The main scenario of contribution consists of introducing a new *object* type to the discovery engine, defining *links* that connect this new object to existing ones, and describing a *clique* (or cliques) that makes use of the object and its links. Below we describe how this scenario should be implemented, step-by-step.

Note: Before writing any new code, you need to create your own environment using UI (see User Guide document) or API (see the API guide doc). Creating an entry directly in *"environments_config"* collection is not recommended.

4.1 Creating new object types

Before you proceed with creation of new object type, you need to make sure the following requirements are met:

- New object type has a unique name and purpose
- New object type has an existing parent object type

First of all, you need to create a fetcher that will take care of getting info on objects of the new type, processing it and adding new entries in Calipso database.

4.1.1 Creating new fetchers

A fetcher is a common name for a class that handles fetching of all objects of a certain type that have a common parent object. The source of this data may be already implemented in Calipso (like OpenStack API, CLI and DB sources) or you may create one yourself.

Common fetchers

Fetchers package structure should adhere to the following pattern (where *%source_name%* is a short prefix like *api, cli, db*):

- app
  - discover
    - fetchers
      - *%source_name%*
        - *%source_name%_%fetcher_name%*.py

If you reuse the existing data source, your new fetcher should subclass the class located in *%source_name%_access* module inside the *%source_name%* directory.

Fetcher class name should repeat the module name, except in CamelCase instead of snake_case.

Example: if you are adding a new cli fetcher, you should subclass *CliAccess* class found by *app/discover/fetchers/cli/cli_access.py* path. If the module is named *cli_fetch_new_objects.py*, fetcher class should be named *CliFetchNewObjects*.

If you are creating a fetcher that uses new data source, you may consider adding an "access" class for this data source to store convenience methods. In this case, the "access" class should subclass the base Fetcher class (found in *app/discover/fetcher.py*) and the fetcher class should subclass the "access" class.

All business logic of a fetcher should be defined inside the overridden method from base Fetcher class *get(self, parent_id)*. You should use the second argument that is automatically passed by parent scanner to get the parent entity from database and any data you may need. This method has to return a list of new objects (dicts) that need to be inserted in Calipso database. Their parent object should be passed along other fields (see example).

Note: types of returned objects should match the one their fetcher is designed for.

Example:

app/discover/fetchers/cli/cli_fetch_new_objects.py

```
from discover.fetchers.cli.cli_access import CliAccess
from utils.inventory_mgr import InventoryMgr


class CliFetchNewObjects(CliAccess):

    def __init__(self):
        super().__init__()
        self.inv = InventoryMgr()

    def get(self, parent_id):
        parent = self.inv.get_by_id(self.env, parent_id)
        # do something
        objects = [{"type": "new_type", "id": "1234", "parent": parent},
                   {"type": "new_type", "id": "2345", "parent": parent}]
        return objects
```

This is an example of a fetcher that deals with the objects of type *"new_type"*. It uses the parent id to fetch the parent object, then performs some operations in order to fetch the new objects and ultimately returns the objects list, at which point it has gathered all required information.

Folder fetcher

A special type of fetchers is the folder fetcher. It serves as a dummy object used to aggregate objects in a specific point in objects hierarchy. If you would like to logically separate children objects from parent, you may use folder fetcher found at *app/discover/fetchers/folder_fetcher.py*.

Usage is described here.

4.1.2 The scanners configuration file structure

Scanners.json (full path *app/config/scanners.json*) is an essential configuration file that defines scanners hierarchy. It has a forest structure, meaning that it is a set of trees, where each tree has a *root* scanner, potentially many levels of *children* scanners and pointers from parent scanners to children scanners. Scanning hierarchy is described here.

A scanner is essentially a list of fetchers with configuration (we'll call those **Fetch types**). Fetch types can be **Simple** and **Folder**, described below.

Simple fetch type

A simple fetch type looks like this:

```
{
  "type": "project",
  "fetcher": "ApiFetchProjects",
  "object_id_to_use_in_child": "name",
  "environment_condition": {
     "mechanism_drivers": "VPP"
  },
  "environment_restriction": [
      {
     "distribution": "Newton",
     "distribution_version": "1.2.3"
      },
  ],
  "children_scanner": "ScanProject"
}
```

Supported fields include:

- *"fetcher"* – class name of fetcher that the scanner uses;
- *"type"* – object type that the fetcher works with;
- *"children_scanner"* – (optional) full name of a scanner that should run after current one finishes;
- *"environment_condition"* – (optional) specific constraints that should be checked against the environment in *environments_config* collection before execution (more details here);
- *"environment_restriction"* – (optional) specific restrictions that should be checked against the environment in *environments_config* collection before execution (more details here);;
- *"object_id_to_use_in_child*" – (optional) which parent field should be passed as parent id to the fetcher (default: "id").

Folder fetch type

Folder fetch types deal with folder fetchers (described here) and have a slightly different structure:

```
{
  "type": "aggregates_folder",
  "fetcher": {
    "folder": true,
```

```
    "types_name": "aggregates",
    "parent_type": "region"
  },
  "object_id_to_use_in_child": "name",
  "environment_condition": {
    "mechanism_drivers": "VPP"
  },
  "children_scanner": "ScanAggregatesRoot"
}
```

The only difference is that *"fetcher"* field is now a dictionary with the following fields:

- *"folder"* – should always be **true**;
- *"types_name"* – type name in plural (with added 's') of objects that serve as folder's children
- *"parent_type"* – folder's parent type (basically the parent type of folder's objects).

Environment_condition and environment_restriction details

Special restrictions for applicable environments are defined in **environment_condition** and **environment_restriction** sections of scanner definition.

Environment_condition specifies requirements that environment configuration should match for the scanner to be applicable for use; **environment_restriction** lists configurations that exclude the environments matching one or more restrictions. This means that even if the environment matches both at least one condition and at least one restriction, restriction takes precedence, and the scanner is not applied to the environment in question.

These fields share the same structure and can be defined in the following ways:

1. *requirement: object*
   requirement is an object consisting of *field-value* pairs used to match corresponding *field-value* pairs in environment configuration. These pairs are matched using an **AND** operator, meaning that all clauses of a requirement should match environment config in order to apply.
   Value of a requirement pair can either be of a concrete type (str, int, …) or a list of concrete types.
   There are two possible matching scenarios, best shown in examples:
   1.1. If the requirement *value* is of a concrete type and environment config *value* is also of a concrete type, they are compared directly. Example:

```
condition: {
  "distribution": "Mercury",
  "mechanism_drivers": "OVS"
}

env_config_match: {
  "distribution": "Mercury",
  "mechanism_drivers": "OVS"
}
```

   This configuration matches the condition since both *distribution* and *distribution_version* values match the requirement.

env_config_no_match: {
"distribution": "Mercury",
"mechanism_drivers": "VPP"
}

This configuration doesn't match the condition since *distribution* value matches the requirement, but *distribution_version* does not.

1.2. If the requirement *value* is of a concrete type and environment config *value* is a list, at least one value in the list should match the requirement. Example:

condition: {
"distribution": "Mercury",
"mechanism_drivers": "OVS"
}

env_config_match: {
"distribution": "Mercury",
"mechanism_drivers": ["OVS", "VPP"]
}

This configuration matches the condition since *distribution* value matches the requirement and required *mechanism_drivers* value is present in environment configuration.

env_config_no_match: {
"distribution": "Mercury",
"mechanism_drivers": ["VPP", "SR-IOV"]
}

This configuration doesn't match the condition since *distribution* value matches the requirement, but no items in *distribution_version* list in environment configuration match the required value.

1.3. If the requirement *value* is a list and environment config *value* is a concrete value, environment configuration value should be among the items of the requirement list. Example:

condition: {
"distribution": "Mercury",
"mechanism_drivers": ["OVS", "VPP]
}

env_config_match: {
"distribution": "Mercury",
"mechanism_drivers": "OVS"
}

This configuration matches the condition since *distribution* value matches the requirement and environment configuration *mechanism_drivers* value is present in required list.

env_config_no_match: {
"distribution": "Mercury",
"mechanism_drivers": “SR-IOV”
}

This configuration doesn’t match the condition since *distribution* value matches the requirement, but environment configuration *mechanism_drivers* value is not present in required list.

1.4. If both the requirement *value* and environment config *value* are lists, all values in environment configuration should be among the items of the requirement list. Example:

condition: {
"distribution": "Mercury",
"mechanism_drivers": ["OVS", “VPP”]
}

env_config_match: {
"distribution": "Mercury",
"mechanism_drivers": [“VPP”, “OVS”]
}

This configuration matches the condition since *distribution* value matches the requirement and all environment configuration *mechanism_drivers* values are present in required list.

env_config_no_match: {
"distribution": "Mercury",
"mechanism_drivers": [“VPP”, “SR-IOV”]
}

This configuration doesn’t match the condition since *distribution* value matches the requirement, but not all values in environment configuration *mechanism_drivers* value are present in required list.

1.5.* If multiple values in requirement list are lists, each of them is validated against environment configuration values using the rules above (any combination of allowed values). Examples:

condition: {
"distribution": ["Mercury", “Newton”],

"mechanism_drivers": ["OVS", “VPP”]
}

env_config_match_1: {
"distribution": "Mercury",
"mechanism_drivers": “VPP”
}

env_config_match_2: {
"distribution": "Newton",
"mechanism_drivers": “OVS”
}

Both configurations match the condition since both *distribution* value are among the required values and all environment configuration *mechanism_drivers* values are present in required list.

env_config_no_match: {
"distribution": "Kilo",
"mechanism_drivers": “VPP”
}

This configuration doesn’t match the condition since *distribution* value is not present in the required list.

2. [*requirement1*: *object*, *requirement2: object*, …]*: list*

Requirement objects can be grouped in lists. In this case, if any requirement is satisfied by environment configuration, result will be treated as a match and no further conditions will be evaluated.

condition: [{
"distribution": “Mercury”,
"mechanism_drivers": “OVS”
}, {
"distribution": “Newton”,
"mechanism_drivers": “VPP”
}
]

env_config_match_1: {
"distribution": "Mercury",
"mechanism_drivers": “OVS”
}

env_config_match_2: {

```
  "distribution": "Newton",
  "mechanism_drivers": “VPP”
}
```

Both configurations match the condition since first environment configuration matches the first condition and the second configuration matches the second condition.

env_config_no_match: {
```
  "distribution": "Mercury",
  "mechanism_drivers": “VPP”
}
```

This configuration doesn’t match any of the conditions. Allowed combinations would be Mercury+OVS and Newton+VPP.

4.1.3 Updating scanners

After creating a new fetcher, you should integrate it into scanners hierarchy. There are several possible courses of action:

Add new scanner as a child of an existing one

If the parent type of your newly added object type already has a scanner, you can add your new scanner as a child of an existing one. There are two ways to do that:

1. Add new scanner as a *“children_scanner”* field to parent scanner

   Example

   Before:

```
"ScanHost": [
 {
   "type": "host",
   "fetcher": "ApiFetchProjectHosts",
 }
],
```

   After:

```
"ScanHost": [
 {
   "type": "host",
   "fetcher": "ApiFetchProjectHosts",
   "children_scanner": "NewTypeScanner"
 }
],
"NewTypeScanner": [
 {
   "type": "new_type",
   "fetcher": "CliFetchNewType"
 }
]
```

2. Add new fetch type to parent scanner (in case if children scanner already exists)

Example

Before:

```
"ScanHost": [
 {
  "type": "host",
  "fetcher": "ApiFetchProjectHosts",
  "children_scanner": "ScanHostPnic"
 }
],
```

After:

```
"ScanHost": [
 {
  "type": "host",
  "fetcher": "ApiFetchProjectHosts",
  "children_scanner": "ScanHostPnic"
 },
 {
  "type": "new_type",
  "fetcher": "CliFetchNewType"
 }
],
```

Add new scanner and set an existing one as a child

Example

Before:

```
"ScanHost": [
 {
  "type": "host",
  "fetcher": "ApiFetchProjectHosts",
  "children_scanner": "ScanHostPnic"
 }
],
```

After:

```
"NewTypeScanner": [
 {
  "type": "new_type",
  "fetcher": "CliFetchNewType",
  "children_scanner": "ScanHost"
 }
]

"ScanHost": [
 {
  "type": "host",
  "fetcher": "ApiFetchProjectHosts",
  "children_scanner": "ScanHostPnic"
 }
],
```

Other cases

You may choose to combine approaches or use none of them and create an isolated scanner as needed.

4.1.4 Updating constants collection

Before testing your new scanner and fetcher you need to add the newly created object type to *"constants"* collection in Calipso database:

1. **constants.object_types** document
   Append a *{"value": "new_type", "label": "new_type"}* object to **data** list.
2. **constants.scan_object_types** document
   Append a *{"value": "new_type", "label": "new_type"}* object to **data** list.
3. **constants.object_types_for_links** document
   If you're planning to build links using this object type (you probably are), append a *{"value": "new_type", "label": "new_type"}* object to **data** list.

4.1.5 Setting up monitoring

In order to setup monitoring for the new object type you have defined, you'll need to add a Sensu check:

1. Add a check script in app/monitoring/checks:
   a. Checks should return the following values:
      0: **OK**
      1: **Warning**
      2: **Error**
   b. Checks can print the underlying query results to stdout. Do so within reason, as this output is later stored in the DB, so avoid giving too much output;
   c. Test your script on a remote host:
      i. Write it in */etc/sensu/plugins* directory;
      ii. Update the Sensu configuration on the remote host to run this check;
      iii. Add the check in the "checks" section of */etc/sensu/conf.d/client.json*;
      iv. The name under which you save the check will be used by the handler to determine the DB object that it relates to;
      v. Restart the client with the command: *sudo service sensu-client restart*;
      vi. Check the client log file to see the check is run and produces the expected output (in */var/log/sensu* directory).
   d. Add the script to the source directory (*app/monitoring/checks*).
2. Add a handler in app/monitoring/handlers:
   a. If you use a standard check naming scheme and check an object, the *BasicCheckHandler* can take care of this, but add the object type in *basic_handling_types* list in *get_handler()*;
   b. If you have a more complex naming scheme, override MonitoringCheckHandler. See HandleOtep for example.
3. If you deploy monitoring using Calipso:
   a. Add the check in the *monitoring_config_templates* collection.

Check Naming

The check name should start with the type of the related object, followed by an underscore ("_"). For example, the name for a check related to an OTEP (type "otep") will start with "otep_". It should then be followed by the object ID.

For checks related to links, the check name will have this format: link_<link type>_<from_id>_<to_id>

4.2 Creating new link types

After you've added a new object type you may consider adding new link types to connect objects of new type to existing objects in topology. Your new object type may serve as a *source* and/or *target* type for the new link type.

The process of new link type creation includes several steps:

1. Write a link finder class;
2. Add the link finder class to the link finders configuration file;
3. Update *"constants"* collection with the new link types.

4.2.1 Writing link finder classes

A new link finder class should:

1. Subclass *app.discover.link_finders.FindLinks* class;
2. Be located in the *app.discover.link_finders* package;
3. Define an instance method called *add_links(self)* with no additional arguments. This method is the only entry point for link finder classes.

FindLinks class provides access to inventory manager to its subclasses which they should use to their advantage. It also provides a convenience method *create_links(self, ...)* for saving links to database. It is reasonable to call this method at the end of *add_links* method.

You may opt to add more than one link type at a time in a single link finder.

Example

```
from discover.find_links import FindLinks


class FindLinksForNewType(FindLinks):

    def add_links(self):
        new_objects = self.inv.find_items({"environment": self.get_env(),
                                           "type": "new_type"})

        for new_object in new_objects:
            old_object = self.inv.get_by_id(environment=self.get_env(),
                                            item_id=new_object["old_object_id"])
```

```
link_type = "old_type-new_type"
link_name = "{}-{}".format(old_object["name"], new_object["name"])
state = "up"  # TBD
link_weight = 0  # TBD

self.create_link(env=self.get_env(),
                 source=old_object["_id"],
                 source_id=old_object["id"],
                 target=new_object["_id"],
                 target_id=new_object["id"],
                 link_type=link_type,
                 link_name=link_name,
                 state=state,
                 link_weight=link_weight)
```

4.2.2 Updating the link finders configuration file

Default link finders configuration file can be found at */app/config/link_finders.json* and has the following structure:

```
{
  "finders_package": "discover.link_finders",
  "base_finder": "FindLinks",
  "link_finders": [
    "FindLinksForInstanceVnics",
    "FindLinksForOteps",
    "FindLinksForPnics",
    "FindLinksForVconnectors",
    "FindLinksForVedges",
    "FindLinksForVserviceVnics"
  ]
}
```

File contents:

- *finders_package* – python path to the package that contains link finders (relative to $PYTHONPATH environment variable);
- *base_finder* – base link finder class name;
- *link_finders* – class names of actual link finders.

If your new fetcher meets the requirements described in Writing link finder classes section, you can append its name to the *"link_finders"* list in *link_finders.json* file.

4.2.3 Updating constants collection

Before testing your new links finder, you need to add the newly created link types to *"constants"* collection in Calipso database:

1. **constants.link_types** document
   Append a *{"value": "source_type-target_type", "label": "source_type-target_type"}* object to **data** list for each new link type.

4.2.4 Creating custom link finders configuration file

If you consider writing a custom list finders configuration file, you should also follow the guidelines from 4.2.1-4.2.3 while designing link finder classes and including them in the new link finders source file.

The general approach is the following:

1. Custom configuration file should have the same key structure with the basic one;
2. You should create a *base_finder* class that subclasses the basic FindLinks class (see Writing link finder classes);
3. Your link finder classes should be located in the same package with your *base_finder* class;
4. Your link finder classes should subclass your *base_finder* class and override the *add_links(self)* method.

4.3 Creating new clique types

Two steps in creating new clique types and including them in clique finder are:

1. Designing new clique types
2. Updating clique types collection

4.3.1 Designing new clique types

A clique type is basically a list of links that will be traversed during clique scans (see Clique creation algorithm). The process of coming up with clique types involves general networking concepts knowledge as well as expertise in monitored system details (e.g. OpenStack distribution specifics). In a nutshell, it is not a trivial process, so the clique design should be considered carefully.

The predefined clique types (in *clique_types* collection) may give you some idea about the rationale behind clique design.

4.3.2 Updating clique types collection

After designing the new clique type, you need to update the *clique_types* collection in order for the clique finder to use it. For this purpose, you should add a document of the following structure:

```
{
    "environment": "ANY",
    "link_types": [
        "instance-vnic",
        "vnic-vconnector",
        "vconnector-vedge",
        "vedge-otep",
        "otep-vconnector",
        "vconnector-host_pnic",
        "host_pnic-network"
    ],
    "name": "instance",
    "focal_point_type": "instance"
}
```

Document fields are:

- *environment* – can either hold the environment name, for which the new clique type is designed, or **"ANY"** if the new clique type should be added to all environments;
- *name* – display name for the new clique type;
- *focal_point_type* – the aggregate object type for the new clique type to use as a starting point;

- *link_types* – a list of links that constitute the new clique type.

4.4 Creating new event handlers

There are three steps to creating a new event handler:

1. Determining event types that will be handled by the new handler;
2. Writing the new handler module and class;
3. Adding the (event type -> handler) mapping to the event handlers configuration file.

4.4.1 Writing custom handler classes

Each event handler should adhere to the following design:

1. Event handler class should subclass the app.discover.events.event_base.EventBase class;
2. Event handler class should override handle method of EventBase. Business logic of the event handler should be placed inside the handle method;
    a. Handle method accepts two arguments: environment name (str) and notification contents (dict). No other event data will be provided to the method;
    b. Handle method returns an EventResult object, which accepts the following arguments in its constructor:
        i. result (mandatory) - determines whether the event handling was successful;
        ii. retry (optional) - determines whether the message should be put back in the queue in order to be processed later. This argument is checked only if result was set to False;
        iii. message (optional) - (Currently unused) a string comment on handling status;
        iv. related_object (optional) – id of the object related to the handled event;
        v. display_context (optional) – (Calipso UI requirement).
3. Module containing event handler class should have the same name as the relevant handler class except translated from UpperCamelCase to snake_case.

Example:

app/discover/events/event_new_object_add.py

```
from discover.events.event_base import EventBase, EventResult

class EventNewObjectAdd(EventBase):
    def handle(self, env: str, notification: dict) -> EventResult:
```

```
        obj_id = notification['payload']['new_object']['id']
        obj = {
            'id': obj_id,
            'type': 'new_object'
        }
        self.inv.set(obj)
        return EventResult(result=True)
```

Modifications in *events.json*:

```
<...>
"event_handlers": {
  <...>
  "new_object.create": "EventNewObjectAdd",
  <...>
}
<...>
```

After these changes are implemented, any event of type new_object.create will be consumed by the event listener and the payload will be passed to EventNewObjectAdd handler which will insert a new document in the database.

4.4.2 Event handlers configuration file structure

Events.json (full path *app/config/events.json*) is a configuration file that contains information about events and event handlers, including:

- Event subscription details (queues and exchanges for Neutron notifications);
- Location of event handlers package;
- Mappings between event types and respective event handlers.

The structure of events.json is as following:

```
{
  "handlers_package": "discover.events",
  "queues": [
    {
      "queue": "notifications.nova",
      "exchange": "nova"
    },
    <…>
  ],
  "event_handlers": {
    "compute.instance.create.end": "EventInstanceAdd",
    "compute.instance.update": "EventInstanceUpdate",
    "compute.instance.delete.end": "EventInstanceDelete",
    "network.create.end": "EventNetworkAdd",
    <…>
  }
}
```

The root object contains the following fields:

- *handlers_package* - python path to the package that contains event handlers (relative to $PYTHONPATH environment variable)
- *queues* – RabbitMQ queues and exchanges to consume messages from (for Neutron notifications case)

- *event_handlers* – mappings of event types to the respective handlers. The structure suggests that any event can have only one handler.

In order to add a new event handler to the configuration file, you should add another mapping to the event_handlers object, where key is the event type being handled and value is the handler class name (module name will be determined automatically).

If your event is being published to a queue and/or exchange that the listener is not subscribed to, you should add another entry to the queues list.

4.5 Creating new event listeners

At the moment, the only guideline for creation of new event listeners is that they should subclass the *ListenerBase* class (full path *app/discover/events/listeners/listener_base.py*) and override the *listen(self)* method that listens to incoming events indefinitely (until terminated by a signal).

In future versions, a comprehensive guide to listeners structure is planned.

4.6 Metadata parsers

Metadata parsers are specialized classes that are designed to verify metadata files (found in *app/*config directory), use data from them to load instances of implementation classes (e.g. scanners, event handlers, link finders) in memory, and supply them by request. Scanners and link finders configuration files are used in scanner, event handlers configuration file – in event listener.

In order to create a new metadata parser, you should consider subclassing *MetadataParser* class (found in *app/utils/metadata_parser.py*). *MetadataParser* supports parsing and validating of json files out of the box. Entry point for the class is the *parse_metadata_file* method, which requires the abstract *get_required_fields* method to be overridden in subclasses. This method should return a list of keys that the metadata file is required to contain.

For different levels of customization you may consider:

1. Overriding *validate_metadata* method to provide more precise validation of metadata;
2. Overriding *parse_metadata_file* to provide custom metadata handling required by your use case.