The ONAP Policy Framework

Abstract

This document describes the ONAP Policy Framework. It lays out the architecture of the framework and specifies the APIs provided to other components that interwork with the framework. It describes the implementation of the framework, mapping out the components, software strucure, and execution ecosystem of the framework. It goes on to provide examples that illustrate how to write, deploy, and run policies of various types using the framework.

1. Overview

The ONAP Policy Framework is a comprehensive policy design, deployment, and execution environment. The Policy Framework is the ***decision making*** component in an ONAP system. It allows you to specify, deploy, and execute the governance of the features and functions in your ONAP system, be they closed loop, orchestration, or more traditional open loop use case implementations. The Policy Framework is the component that is the source of truth for all policy decisions.

One of the most important goals of the Policy Framework is to support Policy Driven Operational Management during the execution of ONAP control loops at run time. In addition, use case implementations such as orchestration and control benefit from the ONAP policy Framework because they can use the capabilities of the framework to manage and execute their policies rather than embedding the decision making in their applications.

The Policy Framework is deployment agnostic, the Policy Framework manages Policy Execution (in PDPs) and Enforcement (in PEPs) regardless of how the PDPs and PEPs are deployed. This allows policy execution and enforcement can be deployed in a manner that meets the performance requirements of a given application or use case. In one deployment, policy execution could be deployed in a separate executing entity in a Docker container. In another, policy execution could be co-deployed with an application to increase performance.

The ONAP Policy Framework architecture separates policies from the platform that is supporting them. The framework supports development, deployment, and execution of any type of policy in ONAP. The Policy Framework is metadata (model) driven so that policy development, deployment, and execution is as flexible as possible and can support modern rapid development ways of working such as DevOps. A metadata driven approach also allows the amount of programmed support required for policies to be reduced or ideally eliminated.

We have identified five capabilities as being essential for the framework:

1. Most obviously, the framework must be capable of being triggered by an event or invoked, and making decisions at run time.
2. It must be deployment agnostic; capable of managing policies for various Policy Decision Points (PDPs) or policy engines.
3. It must be metadata driven, allowing policies to be deployed, modified, upgraded, and removed as the system executes.
4. It must provide a flexible model driven policy design approach for policy type programming and specification of policies.
5. It must be extensible, allowing straightforward integration of new PDPs, policy formats, and policy development environments.

Another important aim of the architecture of a model driven policy framework is that it enables much more flexible policy specification. The ONAP Policy Framework complies with the TOSCA modelling approach for policies, see the TOSCA Policy Primer for more information on how policies are modelled in TOSCA.

1. A Policy Type is a general implementation of a policy for a feature. For example, a Policy Type could be written to manage Service Level Agreements for VPNs. The Policy Type is designed by a domain expert, who specifies the parameters, triggers, and actions that the Policy Type will have. The implementation (the logic, rules, and tasks of the Policy Type) is implemented by a skilled policy developer in consultation with domain experts.

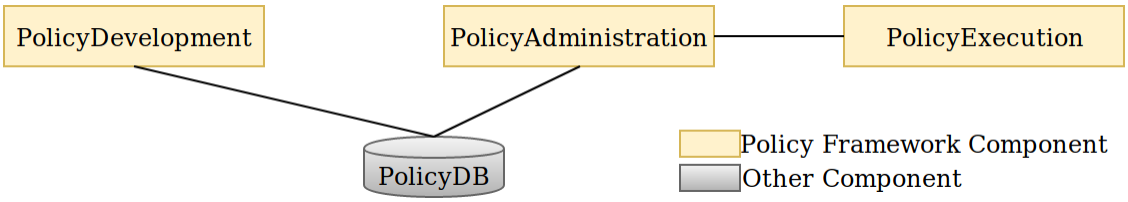
a. For example, the VPN Policy Type is used to create VPN policies for a bank network, a car dealership network, or a university with many campuses.
b. In ONAP, specific ONAP Policy Types are used to create specific policies that drive the ONAP Platform and Components.

2. A Policy is created by configuring a Policy Type with parameters. For example, the SLA values in the car dealership VPN policy for a particular dealership are configured with values appropriate for the expected level of activity in that dealership.

The ONAP Policy Framework for building, configuring and deploying PDPs is extendable. It allows the use of ONAP PDPs as is, the extension of ONAP PDPs, and lastly provides the capability for users to create and deploy their own PDPs. The ONAP Policy Framework provides distributed policy management for **all** policies in ONAP at run time. Not only does this provide unified policy access and version control, it provides life cycle control for policies and allows detection of conflicts across all policies running in an ONAP installation.

2. Architecture

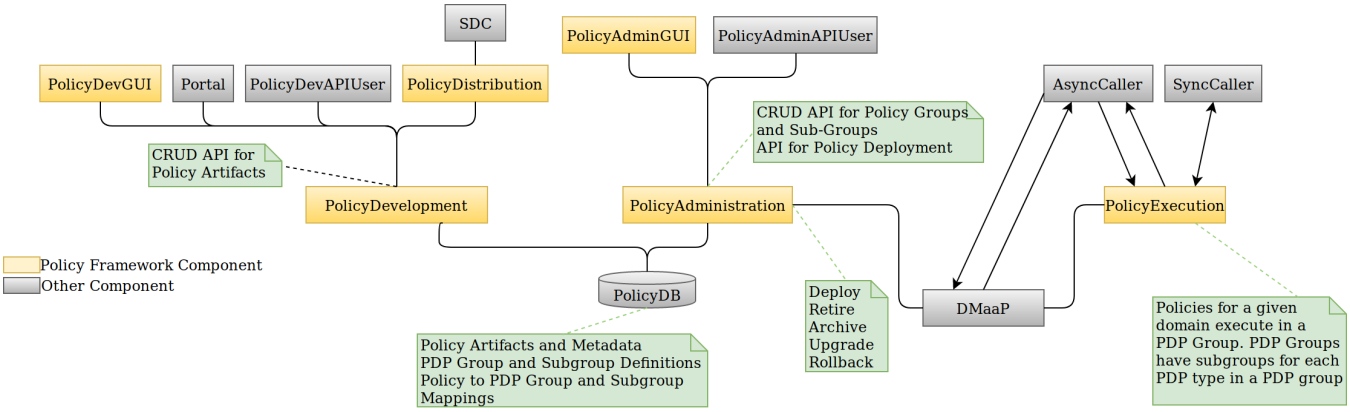
The diagram below shows the architecture of the ONAP Policy Framework at its highest level.



The *PolicyDevelopment* component implements the functionality for development of policy types and policies. *PolicyAdministration* is responsible for the deployment life cycle of policies as well as interworking with the mechanisms required to orchestrate the nodes and containers on which policies run. *Policy Administration* is also responsible for the administration of policies at run time; ensuring that policies are available to users, that policies are executing correctly, and that the state and status of policies is monitored. *PolicyExecution* is the set of PDPs running in the ONAP system and is responsible for making policy decisions and for managing the administrative state of the PDPs as directed by *PolicyAdministration.*

PolicyDevelopment creates policy artifacts and supporting information in the policy database. *PolicyAdministration* reads those artifacts and the supporting information from the policy database whilst deploying policy artifacts. Once the policy artifacts are deployed, *PolicyAdministration* handles the run-time management of the PDPs on which the policies are running. *PolicyDevelopment* interacts with ONAP design time components, and has no programmatic interface with *PolicyAdministration*, *PolicyExecution* or any other run-time ONAP components.

The diagram below shows a more detailed view of the architecture, as inspired by RFC-2753 and RFC-3198.



PolicyDevelopment provides a CRUD API for policy types and policies. The policy types and policy artifacts and their metadata (Information about policies, policy types, and their interrelations) are stored in the *PolicyDB*. The *PolicyDevGUI*, PolicyDistribution, and other applications such as *Portal* can use the *P olicyDevelopment* API to create, update, and delete policy types and policies.

PolicyAdministration has two important functions:

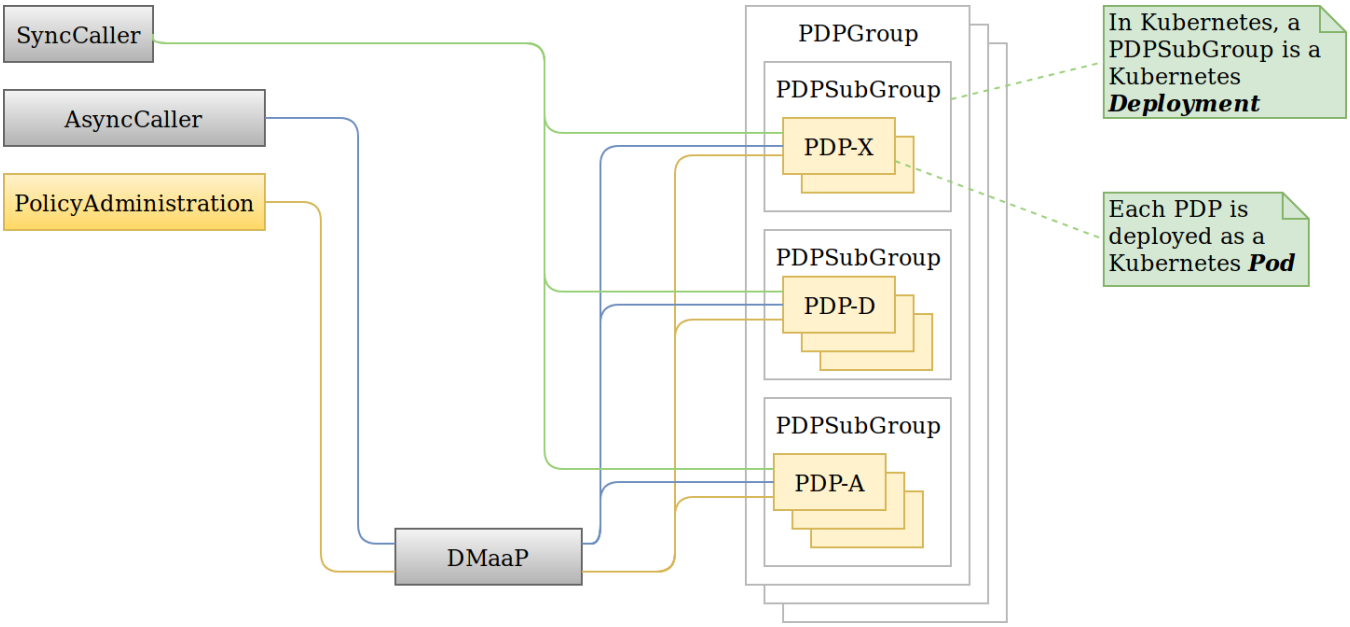
- Management of the life cycle of PDPs in an ONAP installation. PDPs register with *PolicyAdministration* when they come up. *PolicyAdministration* handles the allocation of PDPs to a PDP Groups and PDP Subgroups, so that they can be managed as microservices in Kubernetes.
- Management of the deployment of policies to PDPs in an ONAP installation. *PolicyAdministration* gives each PDP group a set of domain policies to execute.

PolicyAdministration handles PDPs and policy allocation to PDPs using asynchronous messaging over DMaaP.

PolicyAdministation provides three APIs:

- a CRUD API for policy groups and subgroups
- an API that allows the allocation of policies PDP groups and subgroups to be controlled
- an API allows policy execution to be managed, showing the status of policy execution on PDP Groups, subgroups, and individual PDPs as well as the life cycle state of PDPs

PolicyExecution is the set of running PDPs that are executing policies, logically partitioned into PDP groups and subgroups.



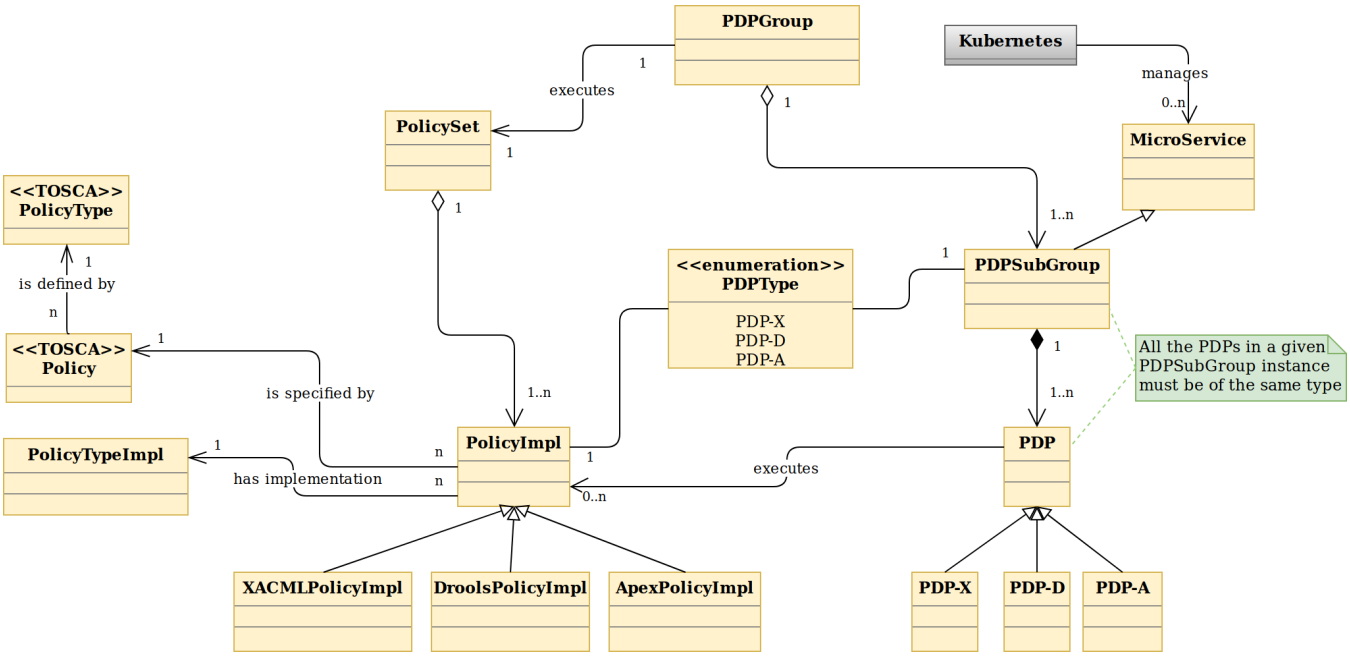
The figure above shows how *PolicyExecution* looks at run time with PDPs running in Kubernetes. A *PDPGroup* is a purely logical construct that collects all the PDPs that are running policies for a particular domain together. A *PDPSubGroup* is a group of PDPs of the same type that are running the same policies. *A PDPSubGroup* is deployed as a Kubernetes Deployment. PDPs are defined as Kubernetes Pods. At run time, the actual number of PDPs in each *PDPSubGroup* is specified in the configuration of the *Deployment* of that *PDPSubGroup* in Kubernetes. This structuring of PDPs is required because, in order to simplify deployment and scaling of PDPs in Kubernetes, we gather all the PDPs of the same type that are running the same policies together for deployment.

For example, assume we have policies for the SON (Self Organizing Network) and ACPE (Advanced Customer Premises Service) domains. For SON, we have XACML, Drools, and APEX policies, and for ACPE we have XACML and Drools policies. The table below shows the resulting*PDPGroup*, *PDPSubGroup*, and PDP allocations:

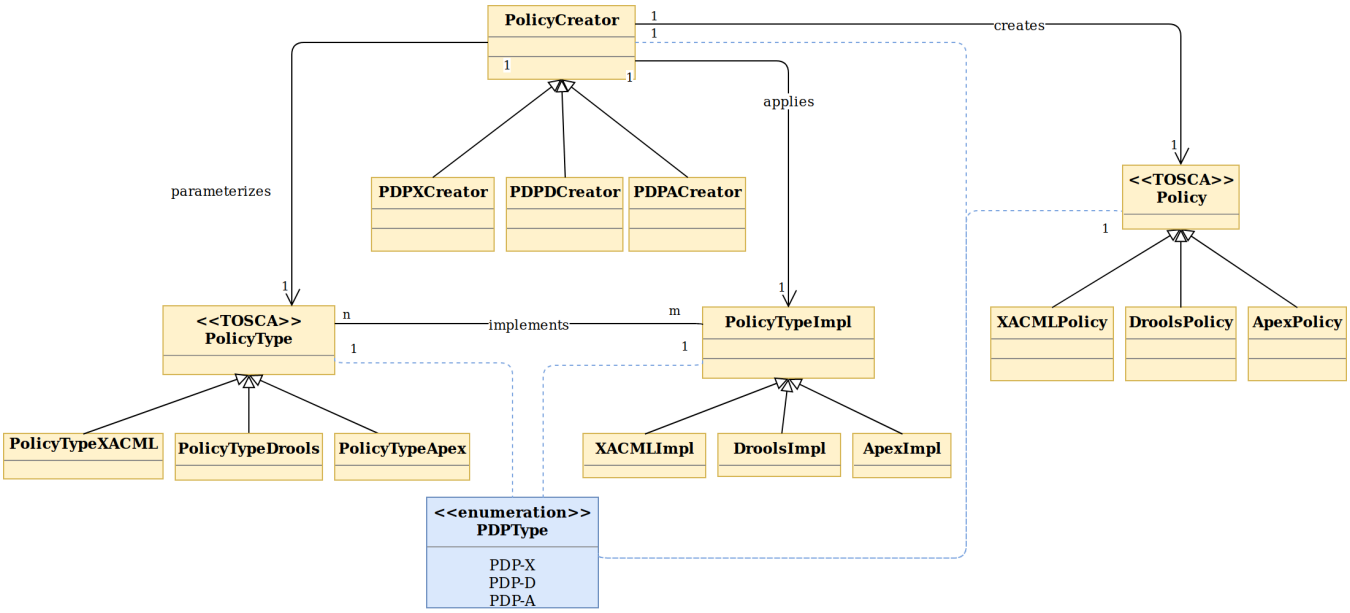
PDP Group	PDP Subgroup	Kubernetes Deployment	Kubernetes Deployment Strategy	PDPs in Pods
SON	SON-XACML	SON-XACML-Dep	Always 2, be geo redundant	2 PDP-X
	SON-Drools	SON-Drools-Dep	At Least 4, scale up on 70% load, scale down on 40% load, be geo-redundant	>= 4 PDP-D
	SON-APEX	SON-APEX-Dep	At Least 3, scale up on 70% load, scale down on 40% load, be geo-redundant	>= 3 PDP-A
ACPE	ACPE-XACML	ACPE-XACML-Dep	Always 2	2 PDP-X
	ACPE-Drools	ACPE-Drools-Dep	At Least 2, scale up on 80% load, scale down on 50% load	>=2 PDP-D

2.1 Policy Framework Object Model

This section describes the structure of and relations between the main concepts in the Policy Framework. This model is implemented as a common model and is used by *PolicyDevelopment*, *PolicyDeployment,* and *PolicyExecution.*



The UML class diagram above shows the portion of the Policy Framework Object Model that applies to *PolicyDeployment* and *PolicyExecution.*



The UML class diagram above shows the portion of the Policy Framework Object Model that applies to *PolicyDevelopment* and *PolicyDeployment.*

2.2 Policy Design Architecture

This section describes the architecture of the model driven system used to develop policy types and to create concrete policies using policy types. The output of Policy Design is deployment-ready artifacts and Policy metadata in the Policy Framework database.

Policies that are expressed via natural language or a model require some development work ahead of time for them to be translated into concrete runtime policies. Some Policy Domains will be setup and available in the platform during startup such as Control Loop Operational Policy Models, OOF placement Models, DCAE microservice models. Policy type implementation development is done by an experienced developer.

2.2.1 Policy Type Design

Policy Type Design is the task of creating policy types that capture the generic and vendor independent aspects of a policy for a particular domain use case. The policy type implementation specifies the model information, rules, and tasks that a policy type requires to generate concrete policies.

All policy types must implement the ONAP Policy Framework *PolicyType* interface. This interface allows *PolicyDevelopment* to manage policy types and to generate policies from these policy types in a uniform way regardless of the domain that the policy type is addressing or the PDP technology that will execute the policy. The interface is used by *PolicyDevelopment* to determine the PDP technology of the policy type, the structure, type, and definition of the model information that must be supplied to the policy type to generate a concrete policy.

A *PolicyTypeImpl* is developed for a certain type of PDP (for example XACML oriented for decision policies or Drools rules oriented for ECA policies). The design environment and tool chain for a policy type is specific for the type of policy being designed.

The *PolicyTypeImpl* implementation (or raw policy) is the specification of the specific rules or tasks, the flow of the policy, its internal states and data structures and other relevant information. A *PolicyTypeImpl* is specific to a PDP technology, that is XACML, Drools, or APEX. *A PolicyTypeImpl* can be specific to a particular policy type, it can be more general, providing the implementation of a class of policy types, or the same policy type may have many implementations.

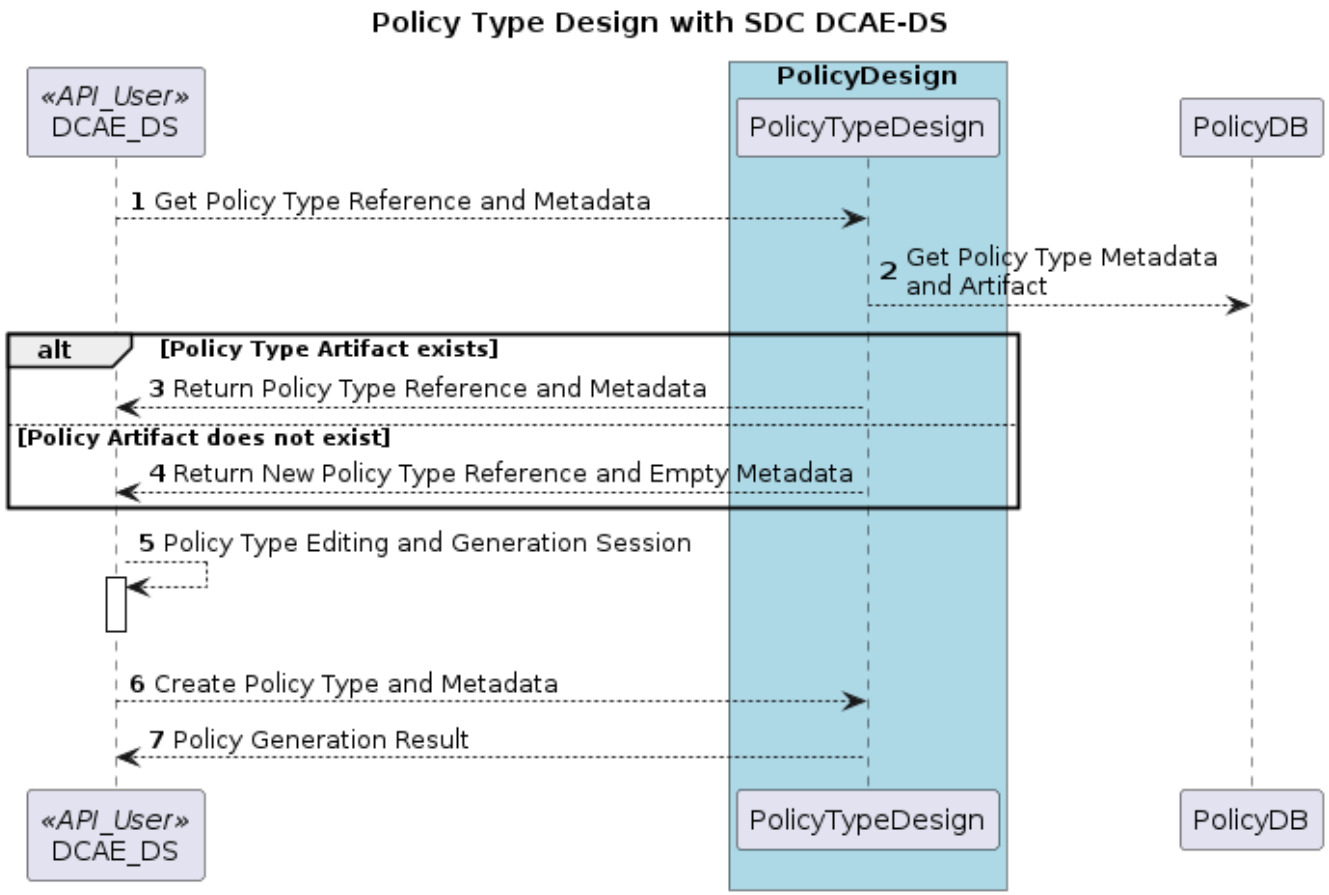
PolicyDevelopment provides the RESTful *Policy Design API*, which allows other components to query policy types and policy type implementations, to determine the model information, rules, or tasks that they require, to specialize policy flow, and to generate policies from policy types. This API is used by the ONAP Policy Framework and other components such as *PolicyDistribution* to create policies from policy types.

Consider a policy type created for managing faults on vCPE equipment in a vendor independent way. The policy type captures the generic logic required to manage the faults and specifies the vendor specific information that must be supplied to the type for specific vendor vCPE VFs. The actual vCPE policy that is used for managing particular vCPE equipment is created by setting the parameters specified in the policy type together with the specific modeled information, rules and tasks in the policy type implementation for that vendor model of vCPE.

2.2.1 Generating Policy Types

It is possible to generate policy types using MDD (Model Driven Development) techniques. Policy types are expressed using a DSL (Domain Specific Language) or a policy specification environment for a particular application domain. For example, policy types for specifying SLAs could be expressed in a SLA DSL and policy types for managing SON features could be generated from a visual SON management tool. The ONAP Policy framework provides an API that allows tool chains to create policy types. SDC uses this approach for generating Policy Types in the Policy Framework, see the Model driven Control Loop Design page.

The SDC GUI supports several types of policies that can be captured at design time. DCAE micro service configuration policies can be onboarded via the DCAE-DS (DCAE Design Studio).



The GUI implementation in another ONAP component such as SDC DCAE-DS uses the *API_User* API to create and edit ONAP policy types.

2.2.1.2 Programming Policy Type Implementations

For skilled developers, the most straightforward way to create a policy type is to program it. Programming a policy type might simply mean creating and editing text files, thus manually creating the TOSCA Policy Type Yaml file and the policy type implementation for the policy type.

A more formal approach is preferred. For policy type implementations, programmers use a specific Eclipse project type for developing each type of implementation, a Policy Type Implementation SDK. The project is under source control in git. This Eclipse project is structured correctly for creating implementations for a specific type of PDP. It includes the correct POM files for generating the policy type implementation and has editors and perspectives that aid programmers in their work

2.2.2 Policy Design

The *PolicyCreation* function of *PolicyDevelopment* creates policies from a policy type. The information expressed during policy type design is used to parameterize a policy type to create an executable policy. A service designer and/or operations team can use tooling that reads the TOSCA Policy Type specifications to express and capture a policy at its highest abstraction level. Alternatively, the parameter for the policy can be expressed in a raw JSON or YAML file and posted over the policy design API described on the Model driven Control Loop Design page.

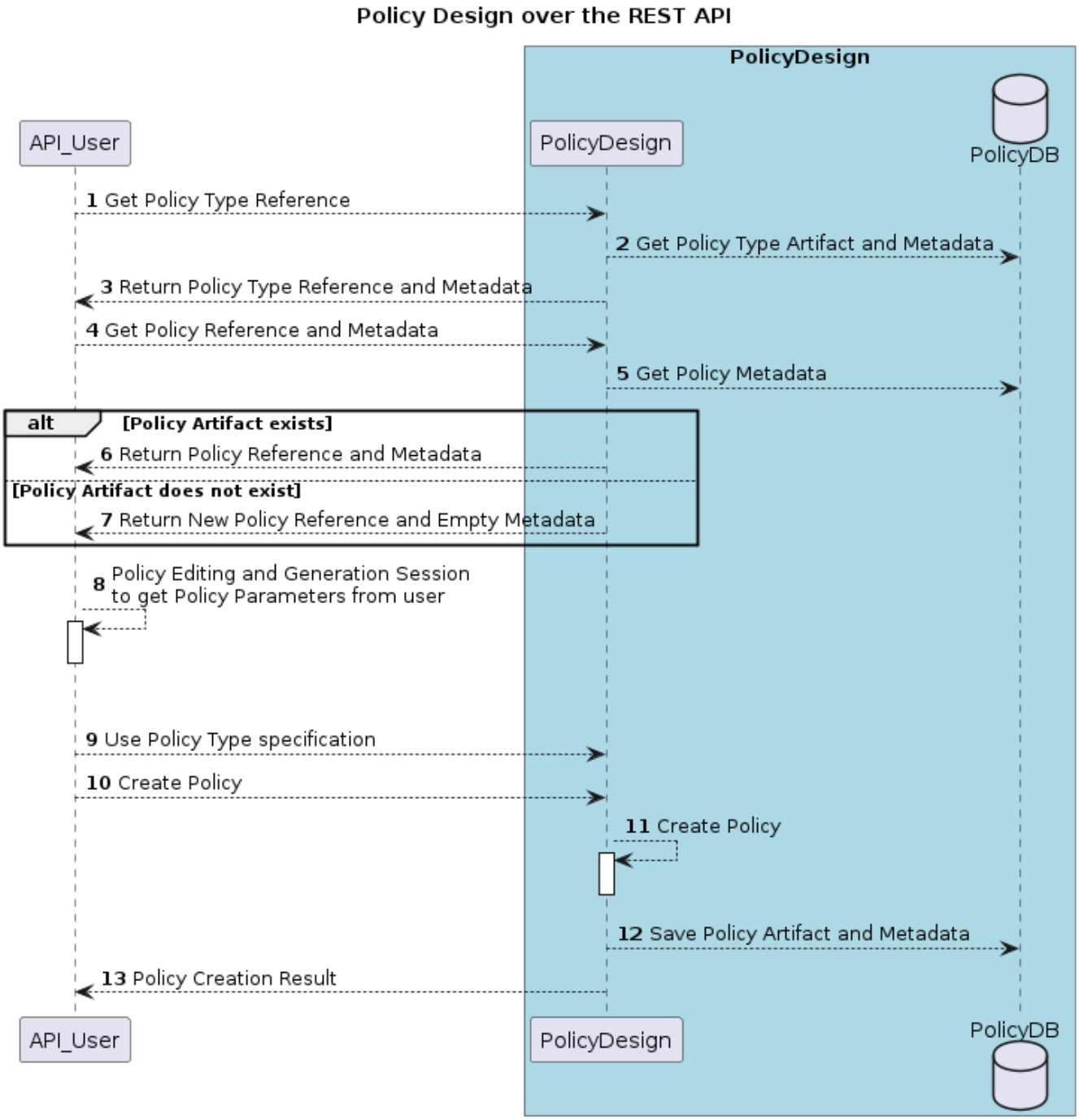
A number of mechanisms for policy creation are supported in ONAP. The process in *PolicyDevelopment* for creating a policy is the same for all mechanisms. The most general mechanism for creating a policy is using the RESTful *Policy Design API*, which provides a full interface to the policy creation support of *PolicyDevelopment*. This API may be exercised directly using utilities such as *curl*. *PolicyDevelopment* provides a command line tool that is a loose wrapper around the API. It also provides a general purpose Policy GUI in the ONAP Portal for policy creation, which again is a general purpose wrapper around the policy creation API. The Policy GUI can interpret any TOSCA Model ingested and flexibly presents a GUI for a user to create policies from. The development of these mechanisms will be phased over a number of ONAP releases.

A number of ONAP components use policy in manners which are specific to their particular needs. The manner in which the policy creation process is triggered and the way in which information required to create a policy is specified and accessed is specialized for these ONAP components.

The following subsections outline the mechanisms for policy creation and modification supported by the ONAP Policy Framework.

2.2.2.1 Policy Design in the ONAP Policy Framework

Policy creation in *PolicyDevelopment* follows the general sequence shown in the sequence diagram below. An *API_USER* is any component that wants to create a policy from a policy type. *PolicyDevelopment* supplies a REST interface that exposes the API and also provides a command line tool and general purpose client that wraps the API.



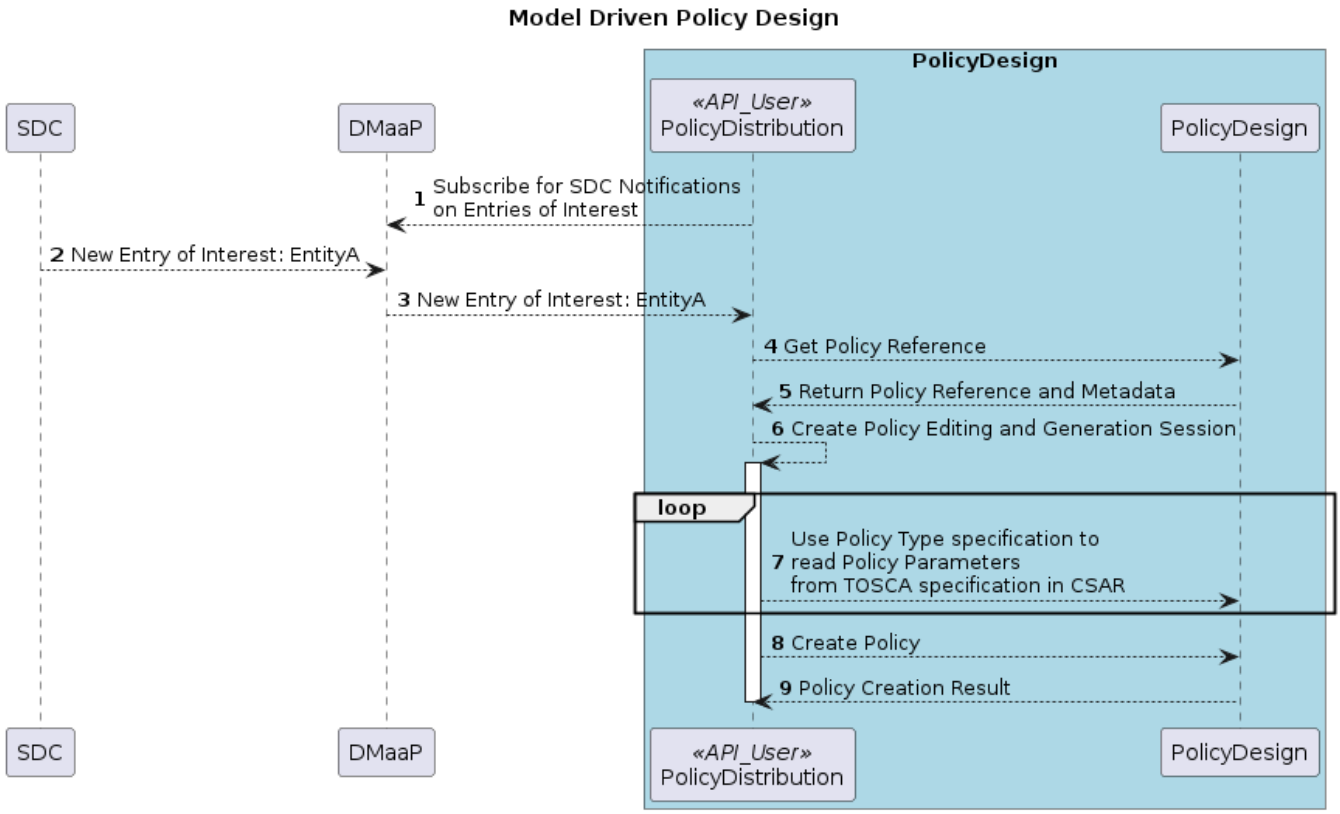
A *PolicyDevAPIUser* first gets a reference to and the metadata for the Policy type for the policy they want to work on from *PolicyDevelopment*. *PolicyDevelopment* reads the metadata and artifact for the policy type from the database. The *API_User* then asks for a reference and the metadata for the policy. *PolicyDevelopment* looks up the policy in the database. If the policy already exists, *PolicyDevelopment* reads the artifact and returns the reference of the existing policy to the *PolicyDevAPIUser* with the metadata for the existing policy. If the policy does not exist, *PolicyDevelopment* creates and new reference and metadata and returns that to the *API_User*.

The *PolicyDevAPIUser* may now proceed with a policy specification session, where the parameters are set for the policy using the policy type specification. Once the *PolicyDevAPIUser* is happy that the policy is completely and correctly specified, it requests *PolicyDevelopment* to create the policy. *PolicyDevelopment* creates the policy, stores the created policy artifact and its metadata in the database.

2.2.2.2 Model Driven VF (Virtual Function) Policy Design via VNF SDK Packaging

VF vendors express policies such as SLA, Licenses, hardware placement, run-time metric suggestions, etc. These details are captured within the VNF SDK and uploaded into the SDC Catalog. The SDC Distribution APIs are used to interact with SDC. For example, SLA and placement policies may be captured via TOSCA specification. License policies can be captured via TOSCA or an XACML specification. Run-time metric vendor recommendations can be captured via VES Standard specification.

The sequence diagram below is a high level view of SDC-triggered concrete policy generation for some arbitrary entity *EntityA*. The parameters to create a policy are read from a TOSCA Policy specification read from a CSAR received from SDC.



PolicyDesign uses the *PolicyDistribution* component for managing SDC-triggered policy creation and update requests. *PolicyDistribution* is an *API_User*, it uses the Policy Design API for policy creation and update. It reads the information it needs to populate the policy type from a TOSCA specification in a CSAR received from SDC and then uses this information to automatically generate a policy.

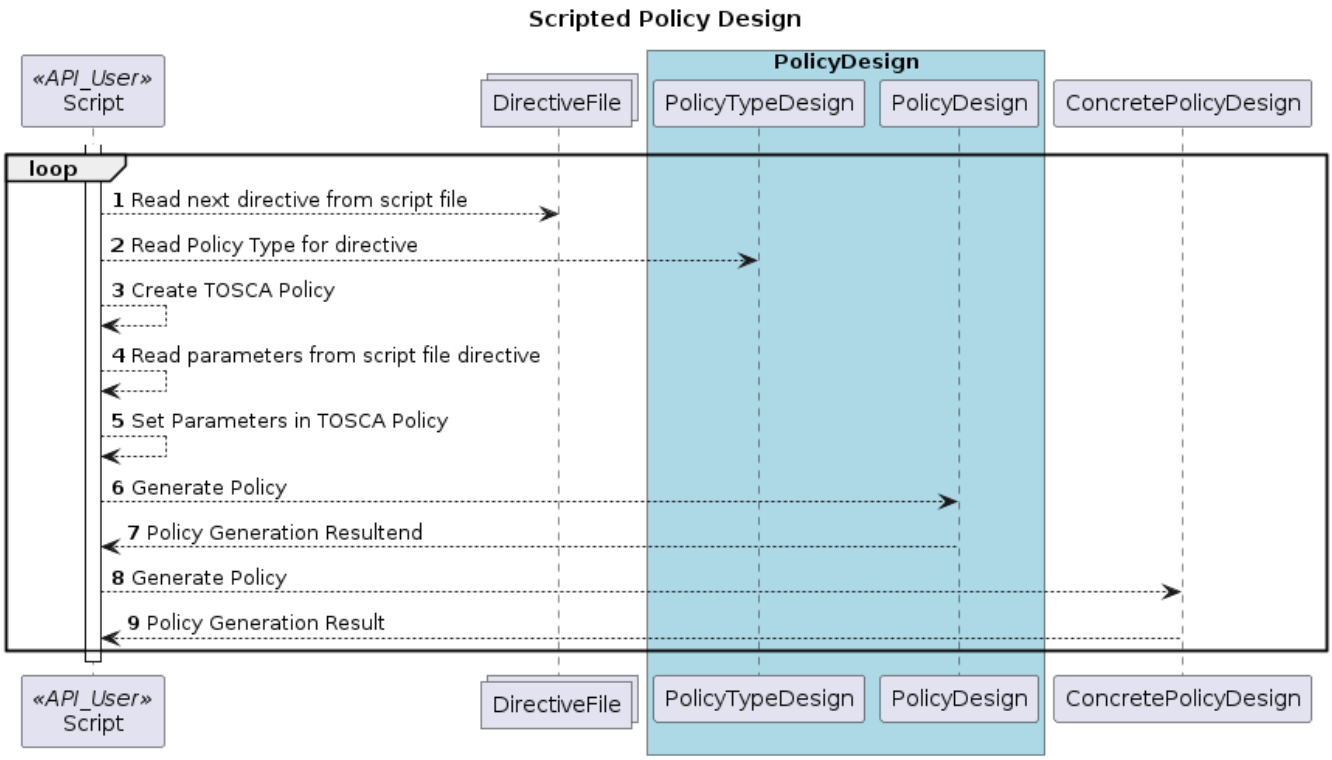
Note that SDC provides a wrapper for the SDC API as a Java Client and also provides a TOSCA parser. See Policy Platform - SDC Service Distribution Software Architecture

In Step 4 above, the *PolicyDesign* must download the CSAR file. If the policy is to be composed from the TOSCA definition, it must also parse the TOSCA definition.

In Step 9 above, the *PolicyDesign* must send back/publish status events to SDC such as DOWNLOAD_OK, DOWNLOAD_ERROR, DEPLOY_OK, DEPLOY_ERROR, NOTIFIED.

2.2.2.4 Scripted Model Driven Policy Design

Service policies such as optimization and placement policies can be specified as a TOSCA Policy at design time. These policies use a TOSCA Policy Type specification as their schemas. Therefore, scripts can be used to create TOSCA policies using TOSCA Policy Types.



One straightforward way of generating policies from Policy types is to use directives specified in a script file. The command line utility is an *API_User*. The script reads directives from a file. For each directive, it reads the policy type using the Policy Type API, and uses the parameters of the directive to create a TOSCA Policy. It then uses the Policy API to create the policy.

2.2.3 Policy Design Process

All policy types must be certified as being fit for deployment prior to run time deployment. In the case of design-time via the SDC application, it is assumed the lifecycle being implemented by SDC will suffice for any policy types that are declared within the ONAP Service CSAR. For other policy types and policy type implementations, the lifecycle associated with software development process will suffice. Since policy types and their implementations will be designed and implemented using software development best practices, they can be utilized and configured for various environments (eg. development, testing, production) as desired.

2.3 Policy Runtime Architecture

The Policy Framework Platform components are themselves designed as micro services that are easy to configure and deploy via Docker images and K8S both supporting resiliency and scalability if required. PAPs and PDPs are deployed by the underlying ONAP management infrastructure and are designed to comply with the ONAP interfaces for deploying containers.

The PAPs keep track of PDPs, support the deployment of PDP groups and the deployment of a policy set across those PDP groups. A PAP is stateless in a RESTful sense. Therefore, if there is more than one PAP deployed, it does not matter which PAP a user contacts to handle a request. The PAP uses the database (persistent storage) to keep track of ongoing sessions with clients. Policy management on PDPs is the responsibility of PAPs; management of policy sets or policies by any other manner is not permitted.

In the ONAP Policy Framework, the interfaces to the PDP are designed to be as streamlined as possible. Because the PDP is the main unit of scalability in the Policy Framework, the PF is designed to allow PDPs in a PDP group to arbitrarily appear and disappear and for policy consistency across all PDPs in a PDP group to be easily maintained. Therefore, PDPs have just two interfaces; an interface that users can use to execute policies and interface to the PAP for administration, life cycle management and monitoring. The PAP is responsible for controlling the state across the PDPs in a PDP group. The PAP interacts with the Policy database and transfers policy sets to PDPs, and may cache the policy sets for PDP groups.

See also Sectino 2 of the TO BE DELETED - refer to Dublin Documentation page, where the mechanisms for PDP Deployment and Registration with PAP are explained.

2.3.1 Policy Framework Services

The ONAP Policy Framework follows the architectural approach for micro services recommended by the ONAP Architecture Subcommittee.

The ONAP Policy Framework defines Kubernetes Services to manage the life cycle of Policy Framework executable components at runtime. A Kubernetes service allows, among other parameters, the number of instances (pods in Kubernetes terminology) that should be deployed for a particular service to be specified and a common endpoint for that service to be defined. Once the service is started in Kubernetes, Kubernetes ensures that the specified number of instances is always kept running. As requests are received on the common endpoint, they are distributed across the service instances. More complex call distribution and instance deployment strategies may be used; please see the Kubernetes Services documentation for those details.

If, for example, a service called *policy-pdpd-control-loop* is defined that runs 5 PDP-D instances. The service has the end point *https://policy-pdpd-control-loop.onap/<service-specific-path>*. When the service is started, Kubernetes spins up 5 PDP-Ds. Calls to the end point *https://policy-pdpd-control-loop.onap/<service-specific-path>* are distributed across the 5 PDP-D instances. Note that the *.onap* part of the service endpoint is the namespace being used and is specified for the full ONAP Kuberentes installation.

The following services will be required for the ONAP Policy Framework:

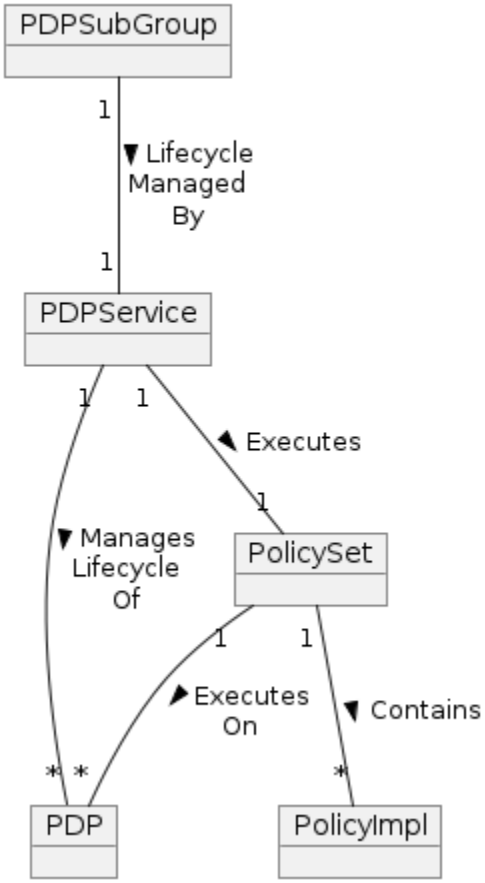
Service	Endpoint	Description
PAP	https://policy-pap	The PAP service, used for policy administration and deployment. See TO BE DELETED - refer to Dublin Documentation for details of the API for this service
PDP-X-*domain*	https://policy-pdpx-*domain*	A PDP service is defined for each PDP group. A PDP group is identified by the domain on which it operates. For example, there could be two PDP-X domains, one for admission policies for ONAP proper and another for admission policies for VNFs of operator *Supacom*. Two PDP-X services are defined: https://policy-pdpx-onap https://policy-pdpx-*supacom*
PDP-D-*domain*	https://policy-pdpd-*domain*	
PDP-A-*domain*	https://policy-pdpa-*domain*	

There is one and only one PAP service, which handles policy deployment, administration, and monitoring for all policies in all PDPs and PDP groups in the system. There are multiple PDP services, one PDP service for each domain for which there are policies.

2.3.2 The Policy Framework Information Structure

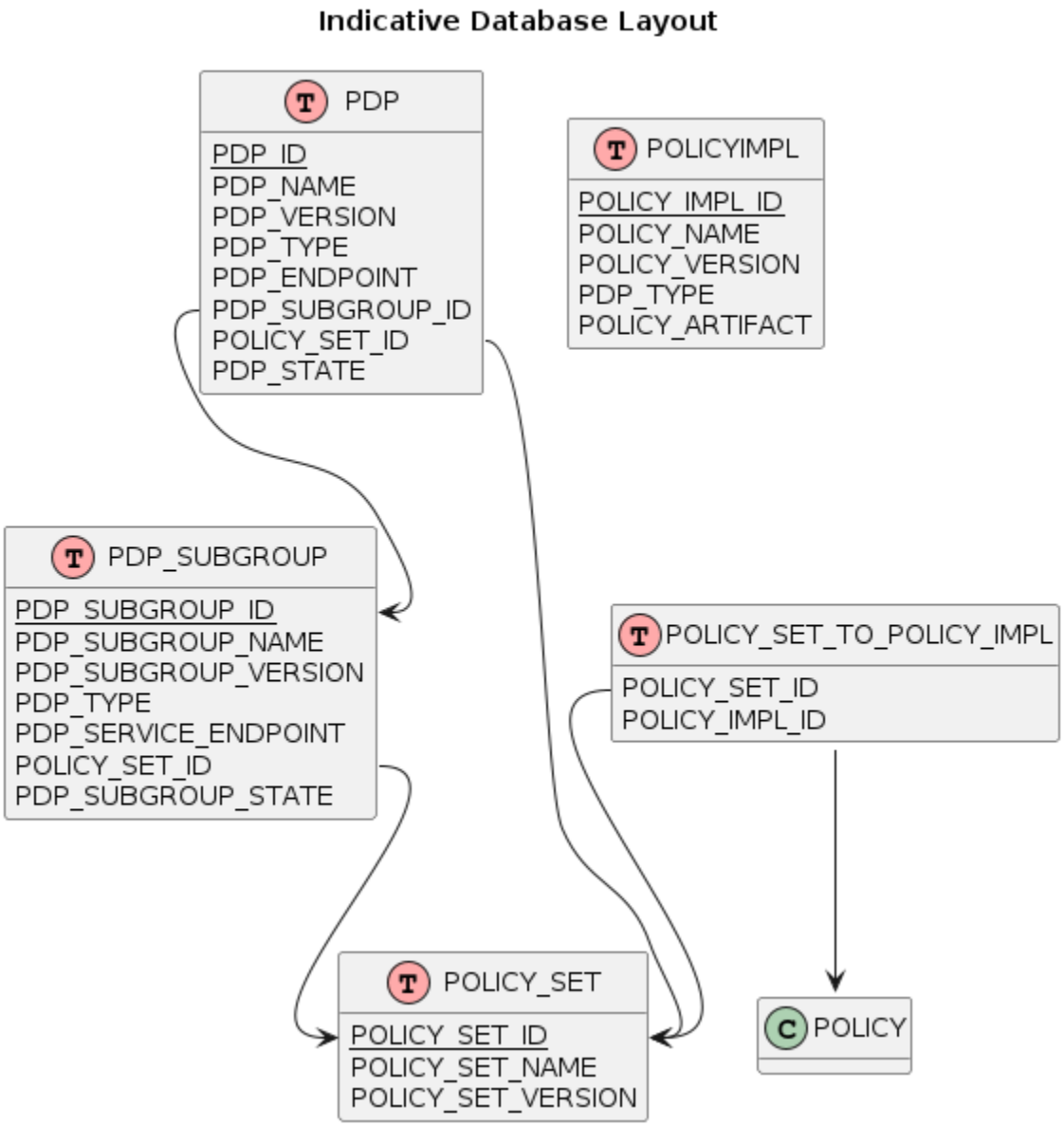
The following diagram captures the relationship between Policy Framework concepts at run time.

Runtime Relationships between Concepts



There is a one to one relationship between a PDP SubGroup, a Kubernetes PDP service, and the set of policies assigned to run in the PDP subgroup. Each PDP service runs a single PDP subgroup with multiple PDPs, which executes a specific Policy Set containing a number of policies that have been assigned to that PDP subgroup. Having and maintaining this principle makes policy deployment and administration much more straightforward than it would be if complex relationships between PDP services, PDP subgroups, and policy sets.

The topology of the PDPs and their policy sets is held in the Policy Framework database and is administered by the PAP service.



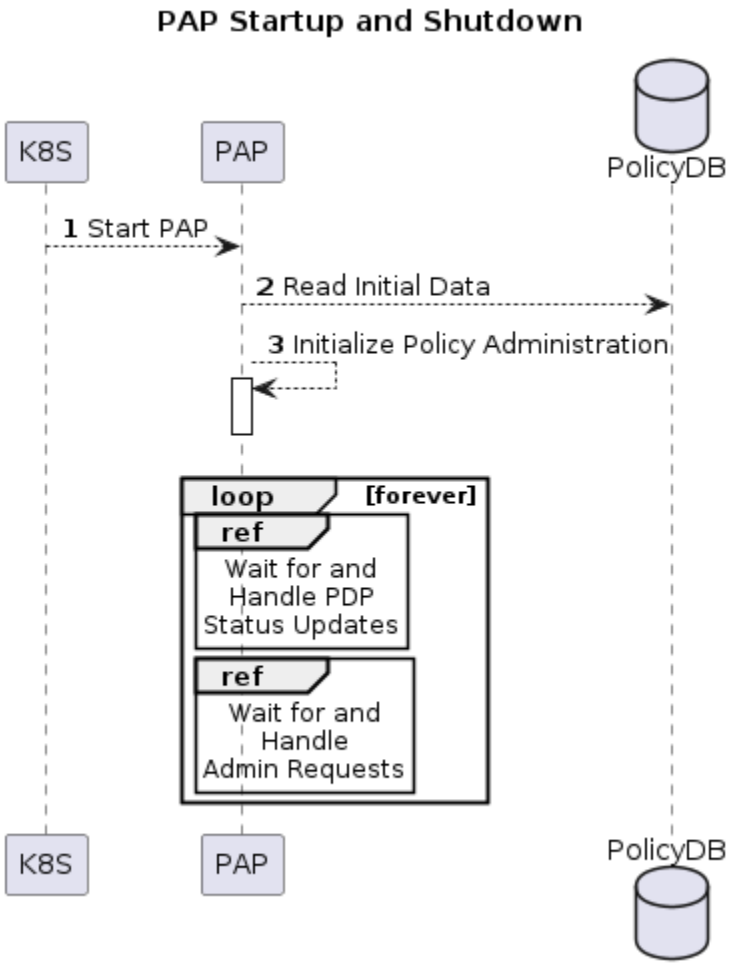
The diagram above gives an indicative structure of the run time topology information in the Policy Framework database. Note that the *PDP_SUBGROUP_STATE* and *PDP_STATE* fields hold state information for life cycle management of PDP groups and PDPs.

2.3.3 Startup, Shutdown and Restart

This section describes the interactions between Policy Framework components themselves and with other ONAP components at startup, shutdown and restart.

2.3.3.1 PAP Startup and Shutdown

The sequence diagram below shows the actions of the PAP at startup.

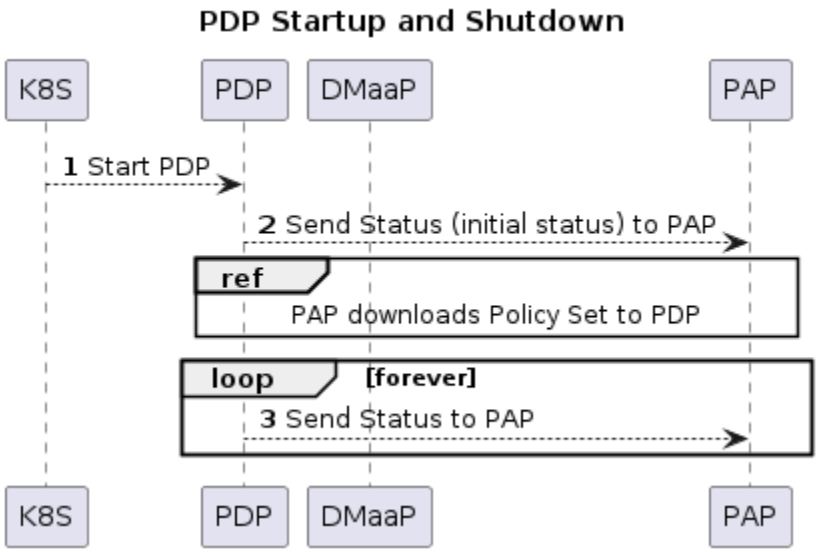


The PAP is the run time point of coordination for the ONAP Policy Framework. When it is started, it initializes itself using data from the database. It then waits for periodic PDP status updates and for administration requests.

PAP shutdown is trivial. On receipt or a shutdown request, the PAP completes or aborts any ongoing operations and shuts down gracefully.

2.3.3.2 PDP Startup and Shutdown

The sequence diagram below shows the actions of the PDP at startup. See also Section 4 of the TO BE DELETED - refer to Dublin Documentation page for the API used to implement this sequence.



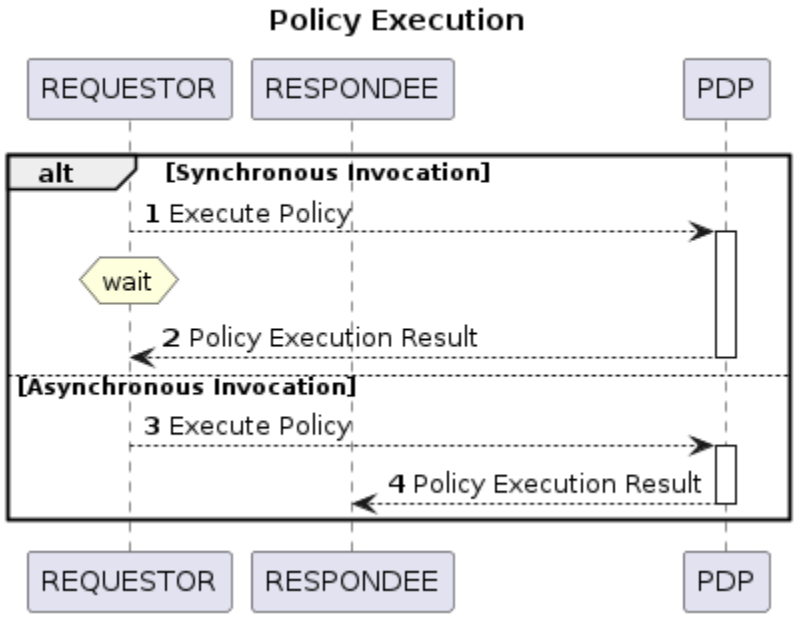
At startup, the PDP initializes itself. At this point it is in PASSIVE mode. The PDP begins sending periodic Status messages to the PAP.

The first Status message initializes the process of loading the correct Policy Set on the PDP in the PAP.

On receipt or a shutdown request, the PDP completes or aborts any ongoing policy executions and shuts down gracefully.

2.3.4 Policy Execution

Policy execution is the execution of a policy in a PDP. Policy enforcement occurs in the component that receives a policy decision.



Policy execution can be *synchronous* or *asynchronous*. In *synchronous* policy execution, the component requesting a policy decision requests a policy decision and waits for the result. The PDP-X and PDP-A use synchronous policy execution. In *asynchronous* policy execution, the component that requests a policy decision does not wait for the decision. Indeed, the decision may be passed to another component. The PDP-D and PDP-A use asynchronous policy execution.

Policy execution is carried out using the current life cycle mode of operation of the PDP. While the actual implementation of the mode may vary somewhat between PDPs of different types, the principles below hold true for all PDP types:

Lifecycle Mode	Behaviour

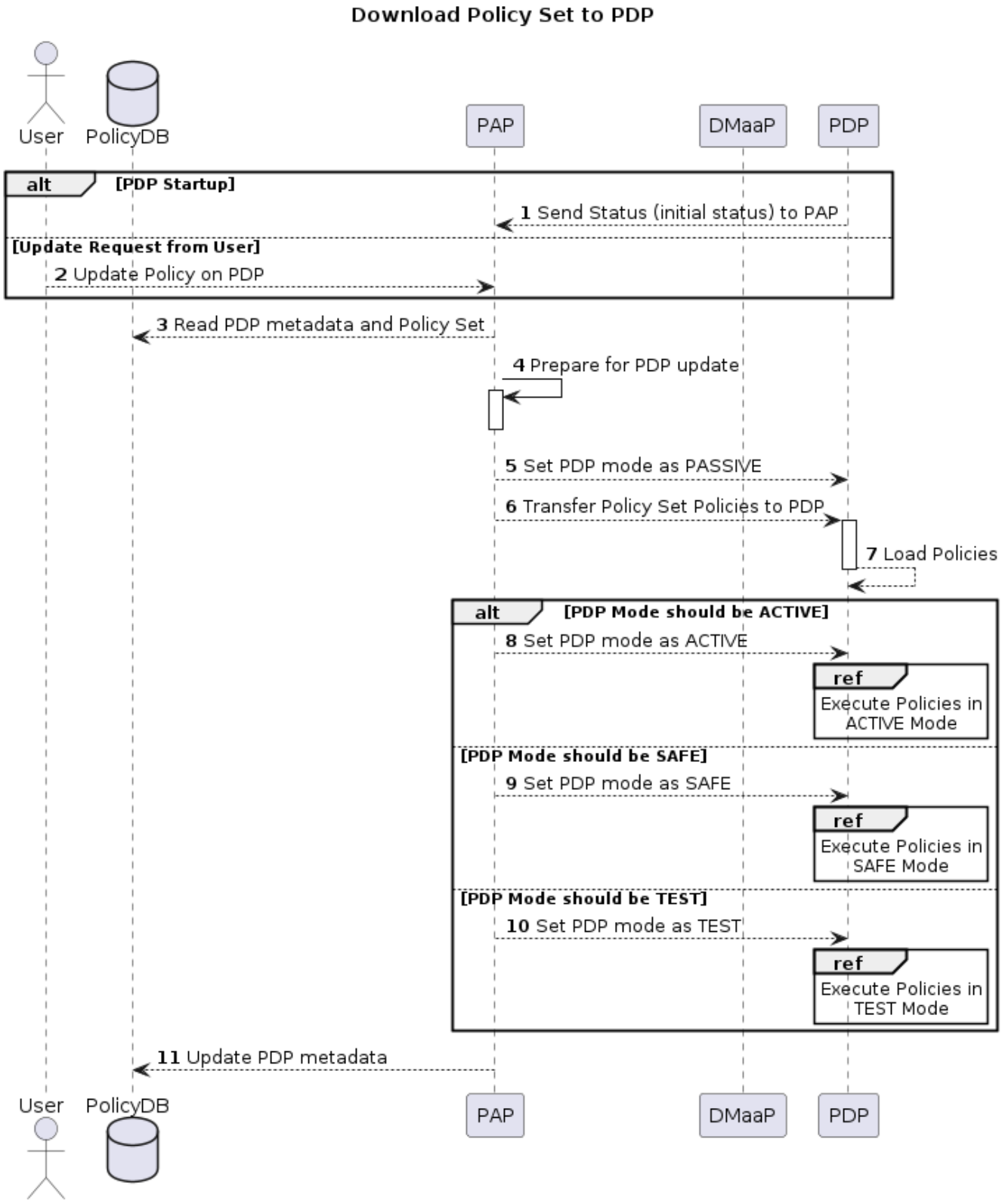
PASSIVE MODE	Policy execution is always rejected irrespective of PDP type.
ACTIVE MODE	Policy execution is executed in the live environment by the PDP.
SAFE MODE	Policy execution proceeds, but changes to domain state or context are not carried out. The PDP returns an indication that it is running in SAFE mode together with the action it would have performed if it was operating in ACTIVE mode. The PDP type and the policy types it is running must support SAFE mode operation.
TEST MODE	Policy execution proceeds and changes to domain and state are carried out in a test or sandbox environment. The PDP returns an indication it is running in TEST mode together with the action it has performed on the test environment. The PDP type and the policy types it is running must support TEST mode operation.

2.3.5 Policy Lifecycle Management

Policy lifecycle management manages the deployment and life cycle of policies in PDP groups at run time. Policy sets can be deploy at run time without restarting PDPs or stopping policy execution. PDPs preserve state for minor/patch version upgrades and rollbacks.

2.3.5.1 Load/Update Policies on PDP

The sequence diagram below shows how policies are loaded or updated on a PDP.



Thi

s sequence can be initiated in two ways; from the PDP or from a user action.

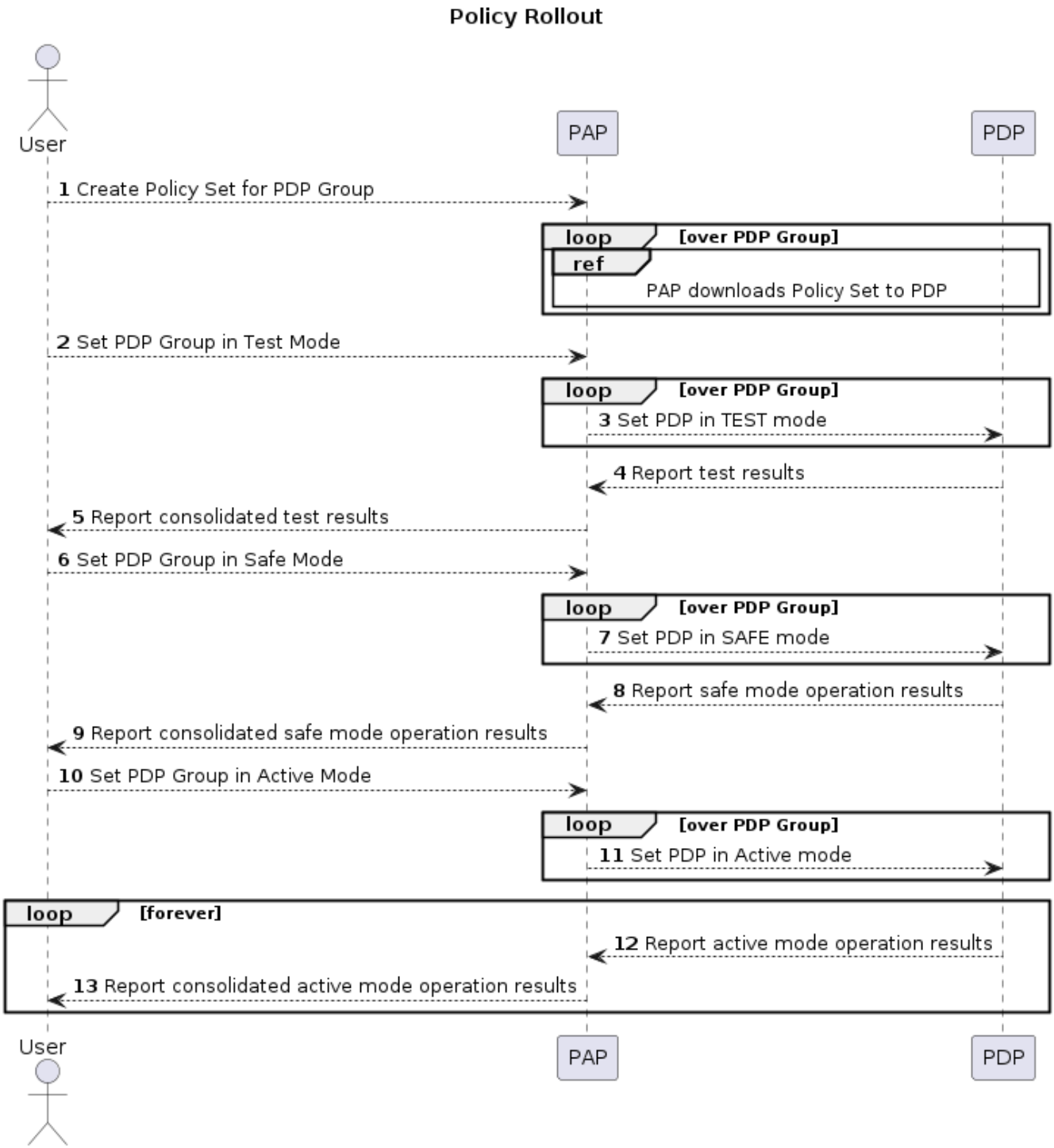
1. A PDP sends regular status update messages to the PAP. If this message indicates that the PDP has no policies or outdated policies loaded, then this sequence is initiated
2. A user may explicitly trigger this sequence to load policies on a PDP

The PAP controls the entire process. The PAP reads the current PDP metadata and the required policy and policy set artifacts from the database. It then builds the policy set for the PDP. Once the policies are ready, the PAP sets the mode of the PDP to PASSIVE. The Policy Set is transparently passed to the PDP by the PAP. The PDP loads all the policies in the policy set including any models, rules, tasks, or flows in the policy set in the policy implementations.

Once the Policy Set is loaded, the PAP orders the PDP to enter the life cycle mode that has been specified for it (ACTIVE/SAFE/TEST). The PDP beings to execute policies in the specified mode (see section 2.3.4).

2.3.5.2 Policy Rollout

A policy set steps through a number of life cycle modes when it is rolled out.



The user defines the set of policies for a PDP group. It is deployed to a PDP group and is initially in PASSIVE mode. The user sets the PDP Group into TEST mode. The policies are run in a test or sandboxed environment for a period of time. The test results are passed back to the user. The user may revert the policy set to PASSIVE mode a number of times and upgrade the policy set during test operation.

When the user is satisfied with policy set execution and when quality criteria have been reached for the policy set, the PDP group is set to run in SAFE mode. In this mode, the policies run on the actual target environment but do not actually exercise any actions or change any context in the target environment. Again, as in TEST mode, the operator may decide to revert back to TEST mode or even PASSIVE mode if issues arise with a policy set.

Finally, when the user is satisfied with policy set execution and when quality criteria have been reached, the PDP group is set into ACTIVE state and the policy set executes on the target environment. The results of target operation are reported. The PDP group can be reverted to SAFE, TEST, or even PASSIVE mode at any time if problems arise.

2.3.5.3 Policy Upgrade and Rollback

There are a number of approaches for managing policy upgrade and rollback.

The most straightforward approach is to use the approach described in section 2.2.5.2 for upgrading and rolling back policy sets. In order to upgrade a policy set, one follows the process in 2.2.5.2 with the new policy set version. For rollback, one follows the process in section 2.2.5.2 with the older policy set, most probably setting the old policy set into ACTIVE mode immediately. The advantage of this approach is that the approach is straightforward. The obvious disadvantage is that the PDP group is not executing on the target environment while the new policy set is in PASSIVE, TEST, and SAFE mode.

A second manner to tackle upgrade and rollback is to use a spare-wheel approach. An special upgrade PDP group service is set up as a K8S service in parallel with the active one during the upgrade procedure. The spare wheel service is used to execute the process described in section 2.2.5.2. When the time comes to activate the policy set, the references for the active and spare wheel services are simply swapped. The advantage of this approach is that the down time during upgrade is minimized, the spare wheel PDP group can be abandoned at any time without affecting the in service PDP group, and the upgrade can be rolled back easily for a period simply by preserving the old service for a time. The disadvantage is that this approach is more complex than the first approach.

A third approach is to have two policy sets running in each PDP, an active set and a standby set. However such an approach would increase the complexity of implementation in PDPs significantly.

2.3.6 Policy Monitoring

PDPs provide a periodic report of their status to the PAP. All PDPs report using a standard reporting format that is extended to provide information for specific PDP types. PDPs provide at least the information below:

Field	Description
State	Lifecycle State (PASSIVE/TEST/SAFE/ACTIVE)
Timestamp	Time the report record was generated
InvocationCount	The number of execution invocations the PDP has processed since the last report
LastInvocationTime	The time taken to process the last execution invocation
AverageInvocationTime	The average time taken to process an invocation since the last report
StartTime	The start time of the PDP
UpTime	The length of time the PDP has been executing
RealTimeInfo	Real time information on running policies.

2.3.7 PEP Registration and Enforcement Guidelines

In ONAP there are several applications outside the Policy Framework that enforce policy decisions based on models provided to the Policy Framework. These applications are considered Policy Enforcement Engines (PEP) and roles will be provided to those applications using AAF/CADI to ensure only those applications can make calls to the Policy Decision API's. Some example PEP's are: DCAE, OOF, and SDNC.

See Section 3.4 of the TO BE DELETED - refer to Dublin Documentation for more information on the Decision APIs.

3. APIs Provided by the Policy Framework

See the TO BE DELETED - refer to Dublin Documentation page.

4. Terminology

PAP (Policy Administration Point)	A component that administers and manages policies
PDP (Policy Deployment Point)	A component that executes a policy artifact (One or many?)
PDP_<>	A specific type of PDP
PDP Group	A group of PDPs that execute the same set of policies
Policy Development	The development environment for policies
Policy Type	A generic prototype definition of a type of policy in TOSCA, see the TOSCA Policy Primer
Policy	An executable policy defined in TOSCA and created using a Policy Type, see the TOSCA Policy Primer
Policy Set	A set of policies that are deployed on a PDP group. One and only one Policy Set is deployed on a PDP group