Congress Design

http://goo.gl/YFd2Fr

1. Overview

Congress is a policy-based management framework for the cloud. It is designed to work with any cloud software that reasonably fits within the relational data model. It includes a high-level general purpose policy language that enables users to dictate how the cloud ought to behave. It automatically prevents policy violations when possible and corrects them when not, and it enables administrators to control the extent to which enforcement is automatic.

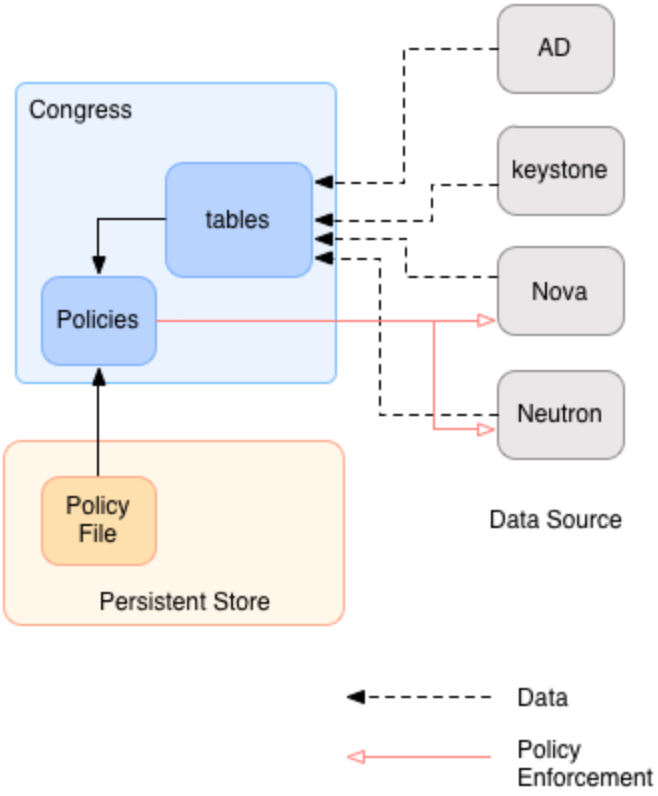

Figure 1 : Congress Component diagram

After giving a rough sketch of our intended use cases, we detail the design of the three major conceptual components of Congress:

- the abstraction Congress uses for interacting with cloud services
- the language people use to express policy
- the capabilities Congress provides for utilizing policy: monitoring, enforcement, auditing

2. Use Cases

For a list of concrete use cases we are currently proposing, see this gdoc link.

Congress aims to support policies addressing the following topics.

- Tenant resource configuration (e.g. VMs, applications)
- Tenant resource deployment (e.g. application/VM locations)
- Tenant resource adaptation (e.g. web app scaling based on load)
- Infrastructure management (e.g. relationship between hosts, networks, storage)

Congress aims to support policies that draw from the information contained in any collection of cloud services and to leverage those cloud services to enforce policy, e.g.

- OpenStack services, e.g. Neutron, Nova, Cinder, Heat
- Directory services, e.g. LDAP, ActiveDirectory
- Commercial services, e.g. antivirus, intrusion detection
- Inventory management services

Congress aims to support monitoring, enforcement, and auditing of policy.

- Monitoring: compare how cloud is actually behaving to how policy says it ought to behave and flag mismatches (violations).
- Enforcement: take action to avoid violations in the future.
- Auditing: understanding the history of policy and violations.

Below we give some more details about what Congress will do with a policy once it has it. We break Enforcement down into several different cases. We use the following example throughout.

> *Every network connected to a VM must be either public or private and owned by someone in the same group as the VM's owner.*

2.1 Monitoring

Monitoring means that Congress will watch how the cloud behaves and compare that behavior to how policy dictates the cloud ought to behave. When Congress finds mismatches, it flags them as policy violations.

For example, if Congress detects that a VM is owned by Alice, that VM is connected to a private network owned by Bob, and there is no group that both Alice and Bob belong to, then Congress will flag an error.

2.1 Proactive Enforcement

Proactive enforcement means that we prevent policy violations before they occur and is often the most desirable style of policy enforcement. Conceptually, this type of enforcement requires other cloud services that are acting on the cloud to consult with Congress before carrying out those actions. (Implementationally, explicitly consulting Congress before any action is taken on the cloud is prohibitively expensive; thus, this consultation process must be implicit.)

To enforce the example policy proactively, we need to do so for the following events:

- A new VM is provisioned
- The networks a VM is connected to are modified
- A network's owner changes
- A VM's owner changes
- A network changes from public to private
- A group-change occurs

As long as we can enforce the policy at each of these events, that policy is never violated.

2.2 Reactive Enforcement

With reactive enforcement, we assume that sometimes the policy will be violated (perhaps despite our best efforts at proactive enforcement), and we take corrective action when that happens. This style of enforcement is necessary if the cloud services are incapable of proactively enforcing a policy, e.g. in the example above if there were no way to stop a group-change event, policy violations could arise.

To reactively enforce our example policy, we assume that Congress has enough information to determine that a policy violation has occurred. Once that violation is recognized, Congress could disconnect the VM from the offending network.

2.3 Interactive Enforcement

In both Proactive and Reactive enforcement, the cloud is enforcing the policy automatically, but with Interactive Enforcement, a human is in the loop. Sometimes the people owning the cloud decide this is the right way to enforce (some aspects of) the policy; other times, Congress needs a person's help to eliminate violations. In addition, the tools used for interactive enforcement can provide helpful insight into the cloud's behavior and thus are useful for operations.

In our example policy, when a violation occurs (a VM is connected to a network whose owner is not in the same group as the VM's owner), there are a number of ways to eliminate that violation, and while it may be obvious to people which option is best, it is unadvisable to assume a computer system will make the same choice.

- disconnect the VM from the offending network
- change the private network causing the violation to a public network
- change the owner of the private network to be someone in the same group as the owner of the VM
- delete the VM
- change the policy

2.4 Assistive Enforcement

Assistive enforcement focuses on the use case where other services need information that is dependent on the current policy and state of the cloud. Congress must therefore be able to enumerate the different ways that the current state of the cloud could be changed in a policy-compliant manner (or at least in a manner that does not increase the number of policy violations). In our running example, the Nova compute service must present a list of networks to a user who is configuring a VM. That list should not include any that would necessarily increase the number of policy violations.

In our example, the list of networks that is returned may be required to be a rough approximation. If Nova simply asks for those networks which user Alice can connect a new VM to, one answer is all those networks to which *any* VM owned by Alice can be connected. Another answer is all those networks to which *some* VM owned by Alice can be connected.

- In the former case, there is no way for the user to configure the VM in such a way to violate the policy if one of the selected networks is chosen; however, it may be that some networks not on that list can be used by Alice *if* she configures her VM properly. This choice may result in no networks for Alice if the policy's decision about network connectivity depends on the VM's configuration.
- In the latter case, some of the networks on the list may require configuring a VM in a specific way that is policy-dependent. This requires a Nova UI design that helps a

user navigate the dependencies between VM configuration and network selection (navigation that is policy-dependent).

2.5 Enforcement Style Comparison

While conceptually proactive, reactive, interactive, and assistive enforcement are different ways to enforce a policy, we can see them as being different aspects of a single policy enforcement model. Users choose which portions of the policy can be enforced automatically and which require human intervention. For that portion that is to be automatically enforced, the system should eliminate policy violations as soon as possible: assistive/proactive when possible and reactive when not. Not all violations can be eliminated before they occur because every cloud has limitations on what actions it can stop, e.g. the cloud can stop a VM provisioning but cannot necessarily stop a denial of service attack.

3. Cloud Service Abstraction

The policy that governs the behavior of the cloud draws from some number of information sources in the cloud (which we call "cloud services"). Congress policies can leverage any cloud service that responds to a well-known interface: that of a relational database. Conceptually, a relational database is a collection of tables. Each table is a collection of rows (or tuples), each of which has some number of columns (or fields).

We use the example from the Use Case section to illustrate.

> *Every network connected to a VM must be either public or private and owned by someone in the same group as the VM's owner.*

The data sources for this policy might be the following tables. We describe each table in the form *tablename*(*col1*, …, *coln*), which means the table is a collection of rows with columns col1 through coln.

```
nova:vm(vm-id)  // the list of all existing VMs
neutron:network(network-id)   // the list of all existing networks
nova:network(vm-id, network-id) // all networks that each VM is connected to
neutron:public(network-id) // list of all public networks
nova:owner(vm-id, user-id) // the VM owners
neutron:owner(network-id, user-id) // the network owners
ldap:group(user-id, group-name)  // group membership
```

These tables correspond to underlying information stores within Nova, Neutron, and LDAP. Initially Congress will simply poll Nova, Neutron, LDAP, etc. to get their latest information, and

wrappers that either Congress developers or third-parties write will translate those API calls into tables as seen above.

Polling will always be necessary for some services (e.g. legacy services), but there will also be a class of services that send updates whenever they happen. This delta-based computation is often more efficient, and it avoids problems with data inconsistency.

Eventually, we want data services to be completely dynamic. New services can be added, existing services can be deleted, services can be spun up, spun down, and queried, all at run-time.

More details about the current capabilities and implementation of the cloud service integration can be found in the source code documentation.

4. Policy Language

The policy language supported by Congress must be general-purpose and declarative. As of now, we have chosen a well-known declarative language called Datalog (with stratified negation), which is a variant of SQL and first-order logic. We may end up choosing a fragment of the language, depending on scale and expressiveness requirements.

Conceptually, Datalog describes policy in terms of a collection of tables. It allows people to describe how to construct new tables out of existing tables. Congress reserves a handful of tables to represent top-level policy decisions. For example, any row in the *error* table represents a violation of a real-world policy. The rest of this section is dedicated to describing the Datalog policy language that allows us to inform Congress how to populate the reserved tables it uses for enforcement. Details about what those tables mean and how they are used is the subject of the next section.

Each policy statement (“rule” for short) is an if-then statement. The “if” portion of each rule (the “body”) is the AND of several possibly negated tables; the “then” portion of each rule (the “head”) is a single, non-negated table. We use a minus sign for negation and the comma for AND. The “if” portion is always on the right, and we use a colon-minus “:-” (which looks a bit like an arrow pointing to the left) to separate the “if” and “then” parts of the rule. To get an OR, we use multiple rules with the same table in the head. Each rule takes the following form.

table0(x1,...,xn_0) :- [-]table1(x1,...,xn_1), …, [-]tablem(x1,...,xn_m)

Each xi may be different from an xi appearing in a different table. Each xi is either a double-quoted string, an integer, a floating point number, or a variable.

To encode the policy for our running example, we have two rules. The first defines what it means for two users to belong to the same group using the ldap::group(user-id, group-name)

table. It says that user1 and user2 belong to the same group exactly when there is some group g that they both belong to.

same-group(user1, user2) :- ldap:group(user1, g), ldap:group(user2, g)

Intuitively, we are defining the collection of all rows that belong to the same-group table. To do that we want to say that *any* row of atomic values <user1, user2> that meets the conditions in the body is a row that belongs in the same-group table; thus we use variables to range over all possible atomic values. If instead we wanted to put particular atomic values into the same-group table, e.g. “alice” or 7, then we could use those atomic values directly.

Notice in the rule above that g is not included in the same-group table; there could be multiple g that prove two users are in the same group. For this policy, we need not know which group the two users both belong to--just that such a group exists. Variables like g that appear in the body of the rule but not the head are so-called “existential variables”---they can apply to any atomic value but need only be assigned to one value. (In database terminology, same-group is a self-join of the ldap:group table followed by a projection.)

The second rule implements the policy. It says that any time the negation of the statement below is true, there is an *error*.

> *Every network connected to a VM must be either public or private and owned by someone in the same group as the VM's owner.*

```
error :- nova:vm(vm),
         neutron:network(network),
         nova:network(vm, network),
        not neutron:public(network),
         neutron:private(network),
        nova:owner(vm, vm-own),
        neutron:owner(network, net-own),
        not same-group(vm-own, net-own)
```

Notice the use of negation (the ‘not’ at the front of table names). It has the usual meaning: that the specified row does NOT belong to the table.

There are some syntactic restrictions we put on a collection of rules that ensures they have the intuitive meaning we have described above: a collection of tables defined in terms of other tables.

- Safety 1: every variable in the rule head must appear in the body.
- Safety 2: every variable in the rule body must appear in a non-negated table in the body.
- Stratification negation: means that no table can be defined in terms of its negated self.

Here is a grammar for the policy language.

<policy> ::= <rule>*
<rule> ::= <atom> COLONMINUS <literal> (COMMA <literal>)*
<literal> ::= NOT <atom>
<literal> ::= <atom>
<atom> ::= TABLENAME LPAREN <term> (COMMA <term>)* RPAREN
<term> ::= INTEGER | FLOAT | STRING | VARIABLE

While we have yet to decide on the particulars, some tables have built-in meanings. Policy writers are not be permitted to write rules to define such tables; rather, the language defines them and builds support for them into its reasoning procedures. For example, arithmetic (addition, subtraction, etc.) is often built into the language. For the cloud domain, we could imagine other builtins as well, e.g. IP address manipulation. The syntactic restrictions on the language is expanded to include the following list when built-ins are included, which basically treat built-in tables as though they were negated tables.

- Safety 3: built-in tables may not appear in the rule head.
- Safety 4: every variable occurring within a built-in table must appear in a non-negated table in the body.

5. Policy Monitoring and Enforcement

Once Congress has policy written over a collection of cloud services, we envision it performing three top-level tasks:

- Monitoring: compare the state of the cloud services against policy and flag mismatches (policy violations)
- Enforcement: change the state of the cloud so that in the future there will be no policy violations, e.g. proactive, reactive, interactive, assistive enforcement.
- Auditing: understand the history of policy and policy violations

Initially we will focus on just monitoring, as we see it having value all on its own. After that we will turn our attention to enforcement.

To support a mix of proactive, reactive, and interactive enforcement, we envision each person writing several different policies, which are described below.

Classification policy. The Classification policy dictates which cloud states violate the real-world policy we are trying to enforce (using the *error* table as described in the previous section). The Classification policy enables Congress to identify violations and their causes, functionality that is important for all forms of enforcement. Below is the classification policy for our running example, which we saw in the previous section.

```
error :- nova:vm(vm), neutron:network(network), nova:network(vm, network),
        -neutron:public(network), neutron:private(network),
        nova:owner(vm, vm-own), neutron:owner(network, net-own),
        -same-group(vm-own, net-own)

same-group(user1, user2) :- ldap:group(user1, g), ldap:group(user2, g)
```

Enforcement Policy. The enforcement policy controls which actions Congress takes based on the state of the cloud. Again this is a policy in the sense that it is written in the same Datalog rule language described earlier. Each rule in this policy has an action table in the head and no actions in the body. Each rule gives conditions under which an action should be executed. It is recommended that this policy does not dictate that multiple actions should be executed at the same time, unless it is certain that the order of execution of those actions (which may actually overlap) is irrelevant.

For our running example, this policy might dictate that the *disconnectNetwork* action should be executed whenever a VM is connected to a private network whose owner is not in the same group as the VM owner. As syntactic sugar we break slightly from the Datalog mold and embed one table within the reserved table *execute*.

```
execute(disconnectNetwork(vm, network)) :-
        nova:vm(vm), neutron:network(network), nova:network(vm, network),
        -neutron:public(network), neutron:private(network),
        nova:owner(vm, vm-own), neutron:owner(network, net-own),
        -same-group(vm-own, net-own)
```

This rule has exactly the same body as the Classification policy's version of the rule, though there is no requirement for that. There are several reasons that the Enforcement policy and the Classification policy are actually different policies. First, this version does not allow us to proactively, interactively, or assistively enforce the policy because it conflates the policy with its reactive enforcement. This policy might include the execution of actions not because there is a policy violation per se, but just because e.g. someone wants an email alerting them to the state of the cloud. Thus it may not be possible to translate the Enforcement policy to the Classification policy. Similarly, the Classification policy does not say anything about actions (at all), and so there is no way to compute the Classification policy from the Enforcement policy.

Access Control policy. This policy dictates which users are permitted to execute which actions in which states of the cloud. It is the policy that is responsible for proactive enforcement. If a cloud service asks Congress whether an action it is about to execute is permitted by policy, Congress checks if the Access Control policy permits that action.

We can use the Datalog rule language to describe an Access Control policy in many different ways. We have yet to choose one, but to give a flavor here is a simple scheme. We use a slightly sugared version of Datalog (similar to the Enforcement policy) where a table can include other tables. We reserve the table *allow* to represent all the actions that are permitted by whom in the current state.

In our running example, we might dictate that the *disconnectNetwork* action may only be executed by the owner of the VM or an administrator.

```
permit(disconnectNetwork(vm, network), user) :- nova:owner(vm, owner),
        -equal(user,owner), -ldap:group(user, “admin”)
```

The Access Control policy differs from the Classification policy in that we may want to allow users to execute actions that cause certain kinds of policy violations. So we cannot compute the Access Control policy from the Classification policy (even if we knew whether an action would cause a new Classification policy violation). Similarly, not all actions prohibited by the Access Control policy would necessarily lead to states that are prohibited by the Classification Policy, since there might be *some* sequence of actions leading to a state that is permitted by the policy but not *all* action sequences leading to that state are permitted. Hence there is no way to compute the Classification policy from the Action policy.

6. Additional Functionality to Design or Implement

Here’s a list of the things we’d like to add to Congress

- Assistive enforcement, e.g. with API
  compliant? query=nova:network(“vm3”, x) &
  assumptions=nova:network(“vm3”)
  unknown=nova:network(“vm3”, x), nova:cpus(“vm3”, x)
- Delegative enforcement: Pushing policy to other OS components, e.g. Neutron’s GBP, policy-aware Nova schedulers, Keystone.
- Policy analysis/debugging
  - Investigate the alternative version of Enforcement described below.
  - Help admins fill out the Enforcement policy by enumerating potential policy violation remediations.
  - Analyze Enforcement policy to look for infinite loops
  - Compare the Access Control policy to the Classification policy to see if there are actions that are permitted that would cause new violations (of a certain kind).

Action Description policy. Before Congress can identify actions that will eliminate policy violations, it must be told what the available actions are. The Action Description policy lists all the available actions that Congress can take and what effects those actions have on the data sources in the cloud.

This is a policy in the sense that it is written using the same Datalog language described above. Each action is represented as a table---its name is the table's name and its arguments are the table's columns. The rules describing an action's effects include one action-table in the body and one of the cloud-service-tables in the head. The cloud-service table in the head is annotated with either a + or a - depending on whether the action will insert or delete a row from that table.

In our running example, suppose one of the actions we have available is *disconnectNetwork*(*vm*, *network*). We can describe its effect on cloud-service tables by writing the following rule, which says that invoking *disconnectNetwork*(*vm*, *network*) deletes the row <vm, network> from the table nova:network, only if the user executing the action is the owner of the VM.

```
nova:network-(vm, network)   :-
        disconnectNetwork(vm, network),
        nova:owner(vm, owner),
        username(owner)
```

Here we utilize the reserved table *username* to check the ID of the tenant making the request. We also include the following rule (which has an empty body) that tells Congress which tables are actions.

```
action("disconnectNetwork")
```

Each action can have multiple rules dictating its effects. An action may simultaneously add and delete rows from different tables by occurring in the body of multiple rules. The Action policy has the following restrictions.

- Each rule must contain at most one action-table; otherwise, it would describe the effects of simultaneously executing two actions--something Congress does not support.
- The rule set must be non-recursive. This is a requirement derived from the algorithms we use to analyze the rules and may be lifted in the future if a new algorithm is chosen.

Enforcement with Action Descriptions. The combination of the Classification policy and the Action can be used on their own to some extent for proactive, reactive, interactive, and assistive enforcement.

- *Proactive*. If another cloud service queries Congress to ask if a given action A is permitted for user U, Congress computes the effects of user U executing action A in the current state, temporarily applies those changes to the current state to compute the resulting state, and checks if the new state has any new violations. New violations indicate that user U ought not be allowed to execute action A (though we refine this idea below).
- *Reactive*. When Congress learns of a change in the cloud's state, it updates its list of policy violations. If any new violations arise because of the change, Congress identifies the rows in the cloud service tables that are the cause of the violation, searches for actions using the Action Description policy that when executed would eliminate the rows in the cloud service tables causing the violations, and chooses one to execute.
- *Interactive*. We can build a dashboard that queries Congress for all the current policy violations and displays them to the user. The dashboard can allow the user to drill down into the violations to understand their causes by exploring the contents of each of the Classification policy's tables and why the rows in those tuples belong there. The dashboard can query Congress to ask for the list of remediations for a given violation and then execute any sequence of actions that the user chooses. If the same violation occurs repeatedly, the user can tell Congress to execute a given remediation every time that violation occurs.
- *Assistive*. Given a partial update to the state U, ask what values could be added to U to create a complete update without violating policy, e.g. when creating a VM, ask which networks can be connected to the VM if the owner of the VM is Alice.

The benefit of the approach outlined above is that policy authors can write two policies that are mostly independent of one another, and Congress can automatically enforce that policy to prevent/correct violations as early as possible and change the balance from proactive to reactive and vice versa automatically without modifications to policy. In contrast, the Access Control policy handles proactive enforcement; the Enforcement policy handles reactive enforcement; and changing the balance between proactive and reactive means

Proactive enforcement with Actions and Access Control

It may be useful to think of the Access Control policy as being embedded within the Action Description policy because an action only has an effect if the user executing that action is allowed to do so. Thus proactive enforcement can be carried out using any of the following schemes.

- Action A is permitted for user U if and only if executing that action in the current state produces no additional violations.
- Action A is permitted for user U if and only if the Access Control policy permits it.
- Action A is permitted for user U if and only if the Access Control policy permits it and executing that action in the current state produces no additional violations.

Drawback to writing Enforcement policies manually. One of the lessons produced by decades of research in declarative languages is that such policies are hard to get right, partly because it is unclear what should happen if the policy dictates that two actions should be executed at the same time and partly because it is easy to create infinite feedback loops (execute action A in response to state 1, which creates state 2, which causes action B to be executed, which causes state 1, ...). modifying those policies.

Additional API

- simulate can also apply to actions
- enumerate remediations, e.g.
  remediate? query=error("vm1")
- execute action, e.g.
  execute? action=disconnectNetwork("vm1", "net_private")
- automating the execution of an action is accomplished by writing a policy statement to the appropriate policy--no need for that here too