PURVIEW (Or Tetris): A generic policy engine for Openstack

Gokul Kandiraju (gokul4open@gmail.com IBM)
Jay Lau (jay.lau.513@gmail.com IBM)

1 Introduction

Increasing adoption of Openstack and a cloud management platform is leading to further development of higher level services. This includes projects such as Heat (for orchestration), Trove (database), Marconi (queueing service) etc. An important implication of Openstack proliferation (and as it becomes more stable with releases) is its applicability for large data-centers (clouds). In this context, a key service that will be required is *an automated way to manage cloud policies*. Policy automation is not only important to avoid human errors while enforcing policies, but also becomes a necessity when we talk about thousands of nodes and tens of thousands of virtual resources (compute, network and storage). While there is an attempt to implement compliance related policies (project Congress), there is no generic policy automation framework in the context of Openstack to enforce various kinds of policies – this is exactly the gap that the *Purview(Tetris)* project aims to fulfill.

2 High-level Goal

Purview(Tetris) will provide framework to quickly implement and enforce different kinds of policies. Policies can be different types. Here are a few examples of policies in clouds:

- Availability Policies
    - An example of availability policy can be to migrate VMs off a node when there is a prediction that a node might be going down or an ECC memory chip reporting errors or a SMART disk sensor with impending failure
    - Another example is to simply restart all the VMs on another node if a node crashes or hypervisor operating system upgrades is a case of planned interventions.
    - Example: turn off/on the nodes on the system based on the load to minimize total energy consumption. If the system is under heavy load, power on more node and migrate/distribute VMs more evenly to powercap each node. If the system is being underutilized (cpu utilization on all nodes is low), then migrate the VMs together to “pack” them and then turn off some hypervisors.
- Performance Policies

    - Example: When the node utilization becomes close to 100%, move some VMs to other no
    - des and increase the VCPU to have a larger fraction of PCPU so that the application may benefit with more CPU and run faster
- Load balancing Policy
    - Example: some of the CPUs are experiencing high utilization while others are experiencing low utilization – move VMs across nodes to balance the overall CPU utilization in the system
- User Defined Policy
    - Example: Operators can collect some customized metrics and create some policies based on those metrics.

The goal of Purview(Tetris) policy automation is to provide a framework so that one can enforce such policies by expressing them in a simplistic manner. In essence, each policy is expressed in form of a combination of CONDITIONS which trigger certain ACTIONS that need to be taken. The logic to trigger these actions can be expressed as RULES or ALGORITHMS (such as for resource optimization). So, this project will provide a framework that can be used by cloud administrators to express their desired policies in a simple human readable manner without worrying about the complexity of underlying operations.

3 High-level Design For Long Term

3.1 Components

The following will be the components of Purview(Tetris) automation framework:

- Conditions
    - Definition of each condition with a generic name
    - Implementing each condition and associating the generic name with code.
    - Each conditions could be implemented as a python class that uses nova to check the status of a VM. That python class is registered as a stevedore plugin, under the ‘conditions’ namespace.
    - The condition supports user supplied parameters, based on each different implemenation.
    - Example:
        - Generic name: VM_FAIL
- Actions
    - Definition of each action with a generic name
    - Implementing each action and associating the generic name with code.

    - Each action could be implemented as a python class that uses nova to accomplish the task of a reboot. That python class is registered as a stevedore plugin, under the 'actions' namespace.
    - The action supports user supplied parameters, based on each different implementation.
    - Example:
        - Generic name: VM_RESTART
- Policy definitions: Definition of how Policies are defined a directory where all policies are expressed in a **generic manner using the above generic names for conditions and actions** (i.e., independent of openstack API calls)
    - Example: { if VM_FAIL(vm1) then VM_RESTART(vm1) }
    - Policies can be expressed as rules OR algorithms (i.e., we need to allow for that. We can parse the rules with these generic names but we also need to allow python based algorithms to be implemented).
    - The policy definition could be written in the form of a YAML file.
- A Parser that parses the policy definition file and constructs the mapping between the conditions and actions, save the parsing result into the DB which would used by the policy automation engine.
- Purview(Tetris) policy automation engine (server side) that continuously runs, monitors the conditions and executes appropriate actions to ensure that all the policies are enforced (details later).
- Purview(Tetris) API server provides restful API: TODO potential API functionalities:
    - list policy rules
    - enable/disable policy?
    - show policy status(how many times the policy have been triggered, when, etc.)?
- Purview(Tetris) client that provides client library/command line like nova client.

3.2 Example

An example is shown here, followed by the overall workflow.

3.2.1 Actions

A action is a python class which inherits the ActionBase class. The action would be registered as a stevedore plugin, and Tetris will load it during start-up time. For each action, sphinx could be used to extract the name, description, and the parameters(if available) from the python docstring.

An example is shown below:

In the file actions/vm_actions.py:

……
…...
class VMMirgrateAction(ActionBase):
    “””Migrate a Virtual Machine

    :param vm_name: virtual machine name
    :param node_name: name of the compute node the VM will migrate to
    “””

    def action(self, context, vm_name=None, node_name):
    ……
    …...

class VMRestartAction(ActionBase):
    “””Restart a Virtual Machine

    :param vm_name: virtual machine name
    “””

    def action(self, context, vm_name):
    ……
    ……

In the file actions/compute_node_actions.py
……
……

class NodeStopDeployAction(ActionBase):
    “””Stop deploying to this Node

    :param node_name: compute node name
    “””
……

class NodeRebootAction(ActionBase):
    “””Reboot a node

    :param node_name: compute node name
……

class NodeReprovisionAction(ActionBase):
    “””Reprovision a node

    :param node_name: compute node name
……

In the file setup.cfg:
……
……
[entry_points]
actions =
    vm.migrate = tetris.actions.vm_actions:VMMirgrateAction
    vm.restart = tetris.actions.vm_actions:VMRestartAction
    node.stopdeploy = tetris.actions.compute_node_actions:NodeStopDeployAction
    node.reboot = tetris.actions.compute_node_actions:NodeRebootAction
    node.reprovision = tetris.actions.compute_node_actions:NodeReprovisionAction

In the example above, VM_MIGRATE action is associated with two parameters (vm name and node name) where as NODE_REBOOT action is only associated with one parameter (node name).

Actions can be sequentially chained that the previous action result should be passed to next action(if any) in the paramerter of context of the method action.

Sphnix could be used to automatically generate the document describe each actions name, purpose, and the parameters it takes.

3.2.2 Conditions

Conditions are defined in a manner similar to actions. Conditions are implemented as python classes and registered as stevedore plugins. Similar to actions, adding a new condition involves implementing a new python class some where in the python code for that condition (which when executed can check if that condition is indeed satisfied), and add it to the entry_point section in setup.cfg file. For example, in the conditions shown below, vm.fail will be associated with a specific python class (VMFail) that invokes the nova python api to query the failure status of a VM (alternatively, the python code can choose to execute custom code, say to check the status of ssh port or a particular service on the VM to return the failure status – there is ample flexibility here).

The evaluate method of each condition would return list of dict of key/value containing the information of the entities that satisfied the condition, for example, below is the return value of the NodeutilizationThreashold's evalaute(node_name=None, threshhold=0.5)

```
[ { 'id': "node1",
     'utilization': 0.54},
 { 'id': "node2",
   'utilization': 0.75},
]
```

If no entities meets the condition, the return value would be an empty list.

Sphnix could be used to automatically generate the document describe each conditions name, purpose, and the parameters it takes, and the return list of python dict.

```
In the file conditions/vm_condictions.py:
……

class VMFail(ConditionBase):
 def evaluate(self, context, vm_name):

    """Detect if a Virtual Machine

    :param vm_name: virtual machine name
```

```
    :returns: list of dict contatining of the following keys
      id: uuid of the VM
      reason: failure reason


   “””
    ……..


In the file conditions/compute_node_conditions.py:
……..

class NodeFail(ConditionBase):
  def evaluate(self, context, node_name):
    “””Detect if a compute node crash

    :param node_name: compute node name
    :returns: list of dict contatining of the following keys
      id: node identifier
      reason: failure reason

    “””
    ……..

class NodeCompromise(ConditionBase):
  def evaluate(self, context, node_name):
    “””Detect if a compute node compromise

    :param node_name: compute node name
    :returns: list of dict contatining of the following keys
      id: node identifier
    “””
……..

class NodeEnegyThreshold(ConditionBase):
  def evaluate(self, context, node_name):

    “””Detect if a compute node reach certain energy threshold

    :param node_name: compute node name
    :param threshold: threshhold value
    :returns: list of dict contatining of the following keys
      id:node identifier
      energy: power of the node

    “””
……..

class NodeUtilizationThreshold(ConditionBase):
  def evaluate(self, context, node_name):

    “””Detect if a compute node utilization is too high

    :param node_name: compute node name
    :param threshold: utilization threshhold value
    :returns: list of dict contatining of the following keys
      id: node identifier
      utilization: utilization ration of the node
    “””
……..


In the file setup.cfg:
……
……
```

```
[entry_points]
conditions=
    vm.fail = tetris.conditions.vm_conditions:VMFail
    node.fail = tetris.conditions.compute_node_conditions:NodeFail
    node.compromise =  tetris.conditions.compute_node_conditions:NodeCompromise
    node.energy.threshold = tetris.conditions.compute_node_conditions:NodeEnegyThreshold
    node.utilization.threshold = tetris.conditions.compute_node_conditions:NodeUtilizationThreshold
```

3.2.3 Policy definitions

Policy rules are defined in the YAML form, like

```
---
rules:
    - name: Availability
      enabled: 1
      interval: 300
      stabalization: 5
      conditions:
          - name: “vm.fail”
            parameters:
                vm_name: “vm1”
      actions:
          - name: “vm.restart”
            parameters:
                 vm_name: “vm1”
    - name: Energy
      enabled: 0
      interval: 10800
      stabalization: 5
      conditions:
          - name: “node.energy.threshold”
            parameters:
                  node_name: None
                  threashold: 90
       actions:
          - name: “node.stopdeploy”
            parameters:
                   node_name: “$id”
    - name: Security
      enabled: 1
      interval: 3600
      stabalization: 5
      conditions:
          - name: “node.compromise”
            parameters:
                node_name: “node2.xx.yy.com”
      actions:
          - name: “node.stopdelpoy”
            parameters:
                node_name: “node2.xx.yy.com”
          - name: “vm.migrate”
            parameters:
                node_name: “node3.xx.yy.com”

user_supplied_policy1:
     policy1_parameters:
```

… …

An example of policy file is shown above. Note that this is just an example policy that has 3 policy rules, ‘availability policy snippet’, ‘energy policy snippet’ and a ‘security policy snippet’ (in respective colors).

For each policy rule, ‘name’ specify the policy name; ‘enabled’ means whether the rule is enabled or not by default; ‘interval’ specify the time interval (in seconds) to evaluate the conditions specified by ‘conditions’; ‘stabalization’ means how much true condition evaluation result we should get before apply the actions specified in ‘actions’.

After parsing the policy file, we load all the conditions/actions stevedore plugins used in those rules, and create a periodic task for each different interval, different rules with the same interval could be processed in the same periodic task. For each periodical task, the processing flow would be something like the following:

```
for rule in rules:
    context = rule.get(‘context’, {})
    context[‘result’] = []
    for condition in rule[‘conditions’]:
        context[‘result’] = condition[‘plugin’].evaluate(context, **condition[‘params’])
   rule[‘context’] = context
   for entity in context[‘result’]:
        for action in rule[‘actions’]:
            action_params = mapping_action_param(action[‘params’], entity)
            action[‘plugin’].action(context, **action_params)
```

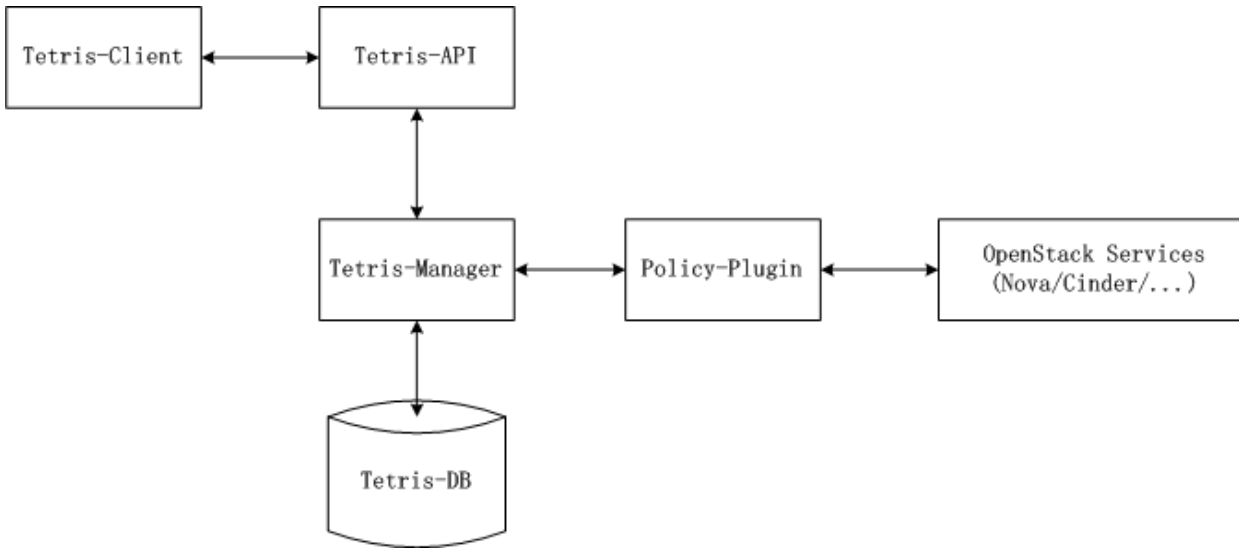
The mapping_action_param would derefernce the action parameter’s value from entity if ‘$’ is find in the rule definition, take the ‘energy policy snippet’ for example:

```
assume:
    entity = {‘id’: ‘node1’, threshold: 0.5}
    action[‘params’] =  {‘node_name’: “$id”},
the mapping_action_param(action[‘params’], entity) would return: {‘node_name’: ‘node1’}.
```

There could also be other parameters associated with policy rules in the future (e.g. history maintenance and related “aging out” parameters, etc.)

Besides the above rule based policies, we could also have user impelmented policies in the future. Each policy plugin(also including the off-the-shelf rule based policy plugin) could be implemented as python classes and registered as a stevedore plugin, just like the actions/conditions. Each policy plugin is responsible for parsing its own YAML definition, and for constructing the a set of tasks to PAE for executions.

3.3 Design Overview of Tetrics (TODO)



Tetris-Client:

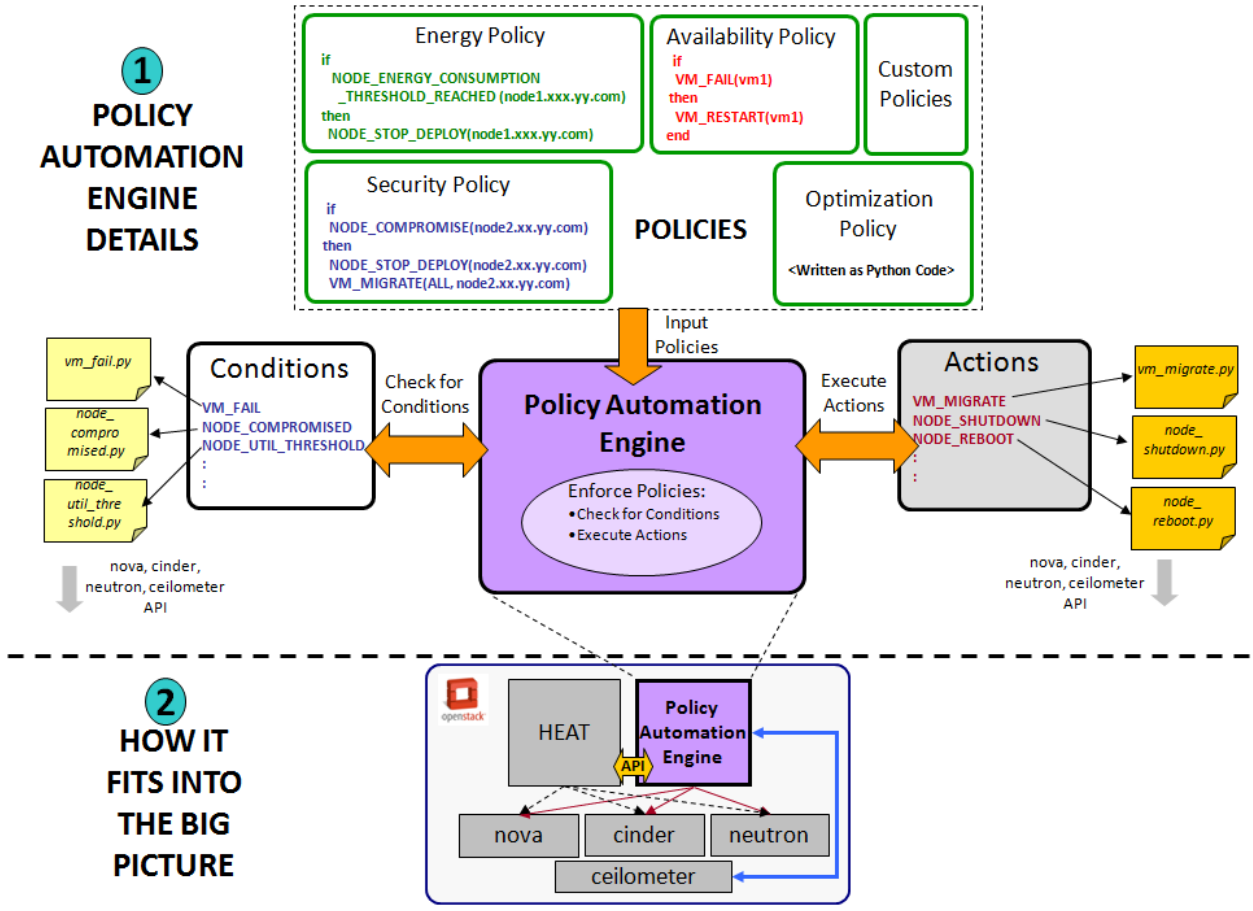
Tetris-API:

Tetris-Manager:

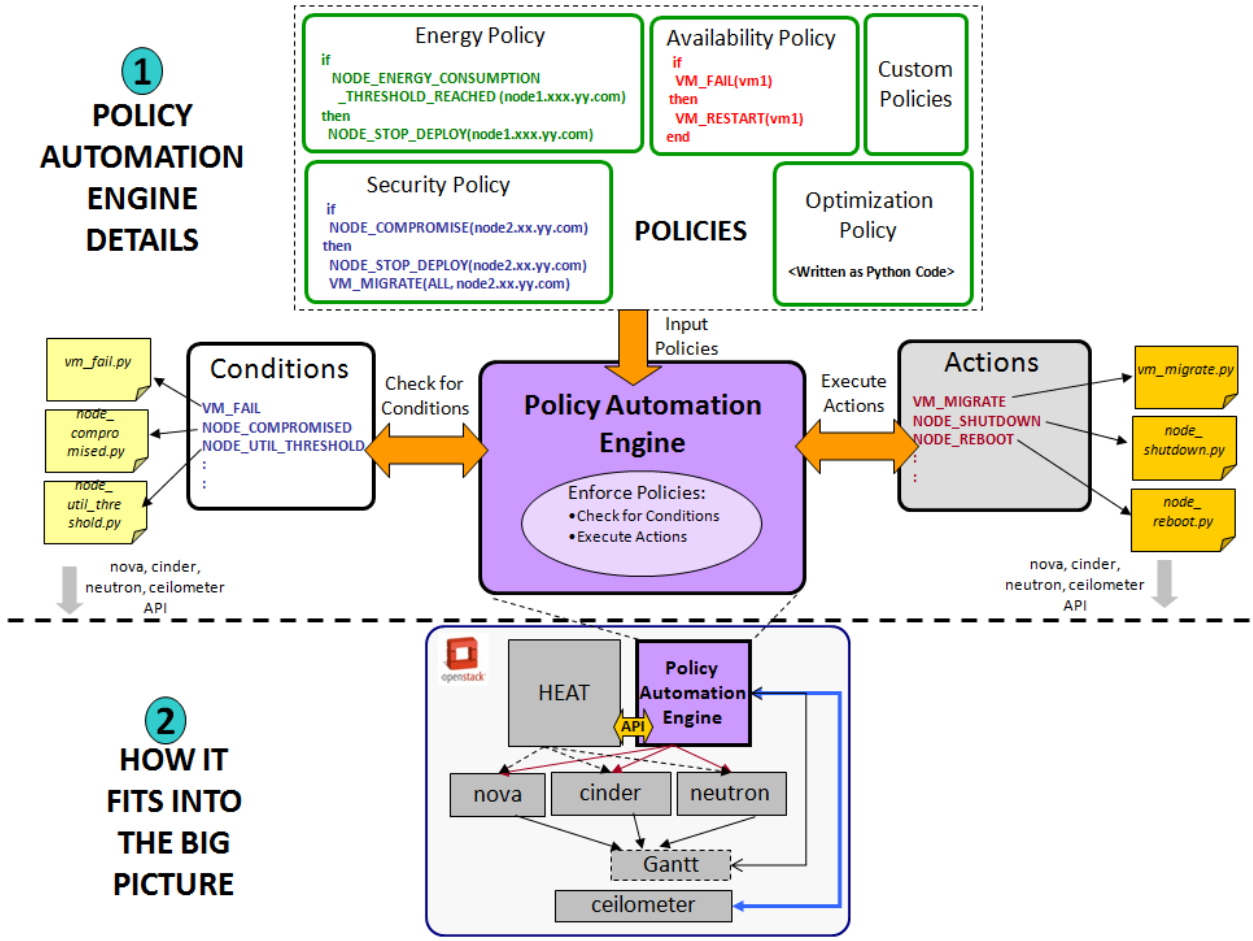
Tetris-DB: Persist histrical data???

3.4 Design Overview of Tetrics Manager

The following diagram illustrates how all of the above mentioned components fall in place together.



A new diagram after adding Gantt.



The diagram above has 2 aspects: the top part is the detail of the Policy Automation Framework and the bottom part is how it fits the big picture.

In summary, any condition can be expressed as python code and an associated 'key word' (such as VM_FAIL) and so is any action. A policy can be expressed in terms of these key words (as illustrated above, for example, Availability policy) OR as a python code algorithm that reaches out and calls the functions from conditions and actions (we will see if we can abstract that out and just have key words -- ?? TBD.). Policy Automation engine will **continuously** (we will allow for policy parameters such as scan intervals, etc. to be set in the policy.conf file) check for conditions and execute appropriate actions, this enforcing the policies.

3.4.1 Purview(Tetris) (Policy) Automation Engine (PAE)

Policy Automation Engine will be implemented as a server process that will take in REST API calls to show policies, conditions, actions, history of condition violation, history of actions, etc.

This process continuously monitors for all the conditions and takes appropriate actions. Actions may also include sending mails to specific users when certain conditions and situations occur.

<At this point, I do not know if it will directly call openstack API (other than through calling the condition-functions OR action-functions). However, it probably needs to interact with HEAT if we want to also take in some workload specific policies that can be expressed in HOT -?? TBD>

More internals of the Automation Engine need to be written in the next (more detailed) version (for example, it needs to maintain a list of all the conditions and actions. It periodically needs to poll and in some cases the condition code may be blocking – for example, if it is registered as a ceilometer alarm). Also, details about purview-client, API and server side also needs to be thought out.

A key aspect of the Purview(Tetris) Automation Engine (PAE) is that it is independent of how underlying conditions and actions are actually implemented. In some sense, one can change the implementation of conditions and actions, without affecting the code automation engine logic. (this is particularly important as Openstack is evolving every day. For example, what if suddenly it is decided to replace ceilometer with some other monitoring system?... – by writing the PAE in an “encapsulated” manner, we just need to replace the hooks for conditions/actions) Therefore, we are designing PAE in an ‘insulated’ manner.

3.4.2 End User Responsibilities

An end user will be given a set of actions and conditions (that will be continuously be added by developers) built on openstack component APIs. It will be fairly simple to construct and enumerate policies based on these conditions and actions. We will also give them a set of policies (which will also be enhanced with time). It will also be possible to write their own set of conditions and actions, and include these in existing/their own policies. An end user need not worry about how the policy is enforced. This will be taken care by the Policy Automation Engine.

3.5 Design Evolution

With time, we will add more intelligence and more formal ways to take in policies (such as grammars).

1) Add vm migration cost analyze. Eg. There are 3 vms in host. VM1 (30% cpu, 2GB); VM2 (20% cpu, 2GB); VM3(40% cpu, 16GB). Migrate VM3 can get the most cpu load reduce, but VM3 has large memory, which may make the migration too long. A migration cost model can guide us how to make the decision.

4 Questions and Comments

Please contact Gokul Kandiraju (gokul4open@gmail.com) or Jay Lau (jay.lau.513@gmail.com).