Kubernetes Policies 101

Eran Leib Co-Founder, VP Product Management
July 2020

Agenda

- Kubernetes, policies and configurations
- Admission Controllers
- Is It Enough?
- From reactive to proactive

About Me

Eran Leib
Co-Founder, VP Product Management

Co-Founder & VP of Product Management at Apolicy and has more than 20 years of experience in security, identity, access and policies.
Prior to Apolicy, I co-founded Whitebox Security, a Data Access Governance platform that was acquired by SailPoint Technologies (NYSE: SAIL).
I enjoy hiking, movies, snowboarding, Lego (but not stepping on them!), cooking (without burning stuff) and traveling for fun.

apolicy

“The future of app development has arrived, and cloud-native architecture has paved the way”

Capgemini

BUT

“The complexity of the cloud native Environment is all too likely to become all-out chaos.”

THENEWSTACK

apolicy

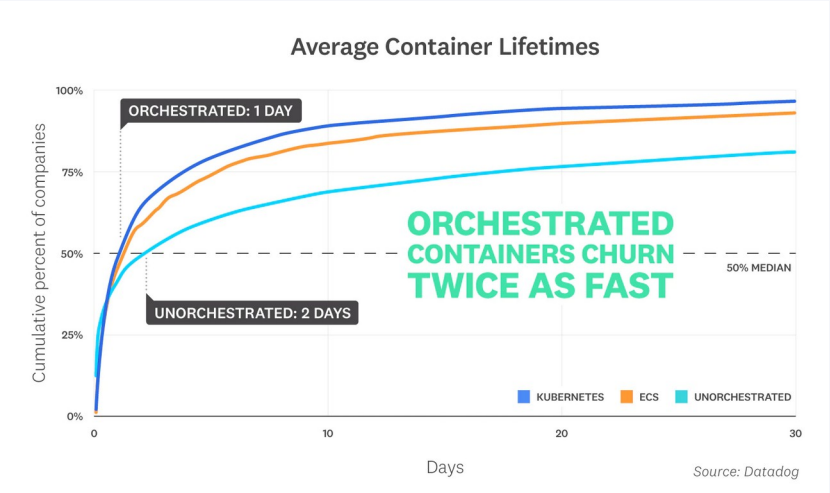
The Dynamic Nature Of Kubernetes

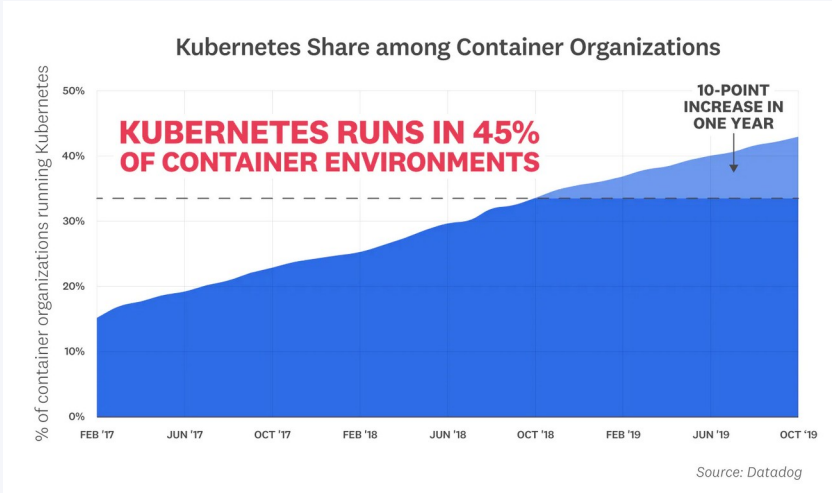
Kubernetes: Up and Running
Brendan Burns, Joe Beda, Kelsey Hightower

“Kubernetes is a very dynamic system. The system is involved in placing pods on nodes, making sure they are up and running, and rescheduling them as needed.
… The API-driven nature of the system encourages others to create higher levels of automation.”

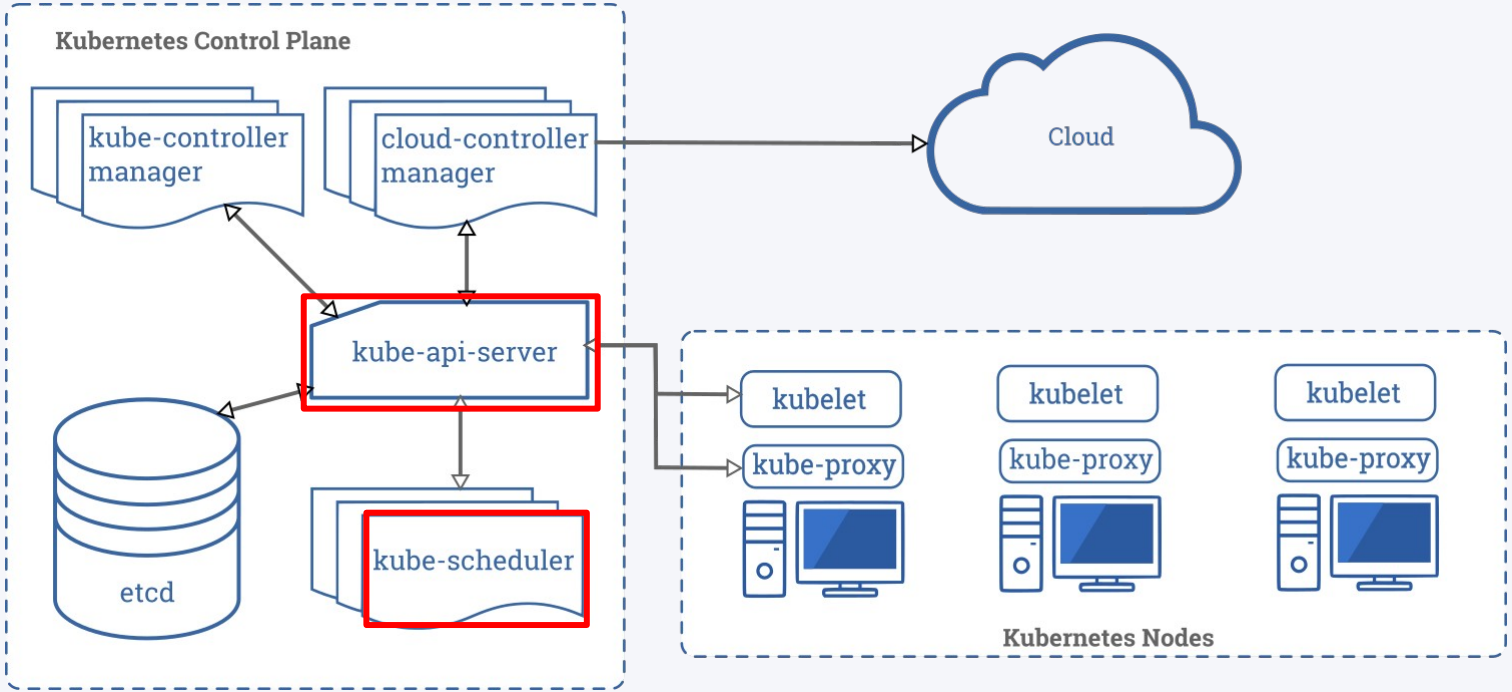
Ephemeral & Complex Is The New Norm

- More clusters
- More workloads per clusters
- Extensive use of microservices
- More multi-cloud environments





What Do We Want To Control?



Policies vs. Configurations

- Configuration is a group of attributes defining behavior parameters, capabilities parameters & labels (of any kind) of an object
- In kubernetes, configuration defines the desired state of the object and its behavior

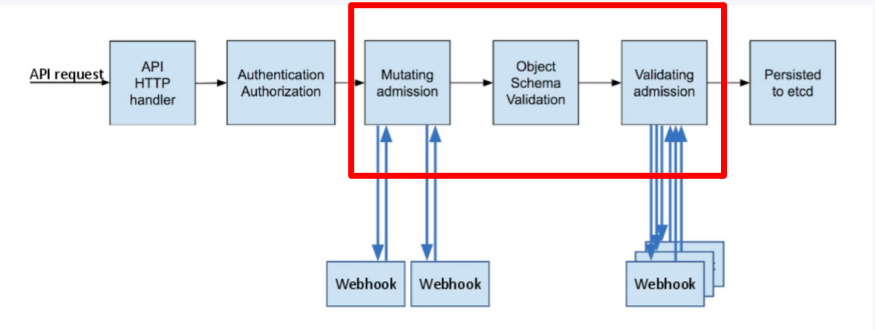
- Policy is a group of attributes defining allowed/denied of behavior/configuration for other objects
- In kubernetes the policy mostly defines guardrails for the desired state

Declarative Infra. & Policy - Winning Duo

- Kubernetes is a declarative infrastructure
- The components’ configurations describe their desired state
- Kubernetes is responsible to achieve the desired state
- The “magic” is in the controller - Genie not included!
- There are many controllers with at least one policy each

Our Focus today - Admission Controllers

- Admission controllers validate creation and changes to Kubernetes objects before the API server persists them to etcd
- The admission controllers enforce the configuration policies
- They run serialy and pass/reject the entire request according to their defined policy
- More specifically:
  Controllers with complex or `coded policies`



ImagePolicyWebhook

- Images are the core of the workload
- ImagePolicyWebhook check they comply with your policy
- All requests goes through inspection and include all the workloads' images and annotations
- Default deny is more secure, however, if no one is there to take the call your cluster would be down and nothing would be admitted!
- Default allow is the easy way out - not the right way

Sample Request & Response

```
{
  "apiVersion":"imagepolicy.k8s.io/v1alpha1",
  "kind":"ImageReview",
  "spec":{
    "containers":[
      {
        "image":"myrepo/myimage:v1"
      },
      {
        "image":"myrepo/myimage@sha256:beb6bd6a68f114c1dc2ea4b28db81bdf91de202a9014972bec5e4d9171d90ed"
      }
    ],
    "annotations":{
      "mycluster.image-policy.k8s.io/ticket-1234": "break-glass"
    },
    "namespace":"mynamespace"
  }
}
```


```
{
  "apiVersion": "imagepolicy.k8s.io/v1alpha1",
  "kind": "ImageReview",
  "status": {
    "allowed": false,
    "reason": "image currently blacklisted"
  }
}
```

Image Policy - Shortcomings

- No OOTB simple policy to overcome the simple use cases
- Operations vs Security instead of Secured Operations
  - Default allow vs default deny
- Doesn’t have full context of the workload - only the annotations
- Cannot filter which requests come in (e.g. kube-system)

Resources - Quotas & Limits

- Kubernetes provides two ways of policing the resources distribution
- The ResourceQuota object defines the quota per namespace
- The LimitRanger object defines the limits per workload inside the namespace
- ResourceQuota defined by Cluster Admin and LimitRanger by the namespace admin
- Any running workload will not be affected until recreated

ResourceQuota

- ResourceQuota is an admission controller enforcing the policy
- To enforce it, you must create a ResourceQuota object in each namespace
- With no object defined, the namespace can use resources without limits
- Using a quota should be in tandem with LimitRanger to make sure default values are assigned on resources
  - Your Pod can be rejected if resources are not defined and LimitRanger is not used

LimitRanger

- LimitRanger defines ranges for CPU/RAM/Storage for a Pod/Container inside the namespace
- The admission controller defines, MIN/MAX, ratio between request/limit and default values
- A LimitRanger object must be defined to enforce the limits
- Without a limit, any workload can create starvation for the others
- If the LimitRanger allows more resources than the ResourceQuota you will run into contention and Pod creation failures

Pod Security Policy (PSP)

- PSP defines the security guardrails for workloads in the cluster
- PSP works with policy profiles
  - The workload service account must be able to ‘use’ the PSP to enforce it
  - Each profile is a complete set of the parameters
  - A workload will “choose” the easiest path in that will allow it in as-is
- Hint: Recently a list of 3 standard profiles was published
  - https://kubernetes.io/docs/concepts/security/pod-security-standards/

‘The three profiles defined here have a clear linear progression from most secure (restricted) to least secure (privileged), and cover a broad set of workloads. Privileges required above the baseline policy are typically very application specific, so we do not offer a standard profile in this niche. This is not to say that the privileged profile should always be used in this case, but that ***policies in this space need to be defined on a case-by-case basis****.”*

PSP - What’s Covered

- Workload <> Host isolation parameters (Network, process, FS)
- Container Security Context (user, group, capabilities)
- Volumes (allowed types, root FS, etc)
- Privilege Escalations (privileged, allow escalation)

A total of 20+ parameters

Privileged

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: privileged
  annotations:
    seccomp.security.alpha.kubernetes.io/allowedProfileNames: '*'
spec:
  privileged: true
  allowPrivilegeEscalation: true
  allowedCapabilities:
  - '*'
  volumes:
  - '*'
  hostNetwork: true
  hostPorts:
  - min: 0
    max: 65535
  hostIPC: true
  hostPID: true
  runAsUser:
    rule: 'RunAsAny'
  seLinux:
    rule: 'RunAsAny'
  supplementalGroups:
    rule: 'RunAsAny'
  fsGroup:
    rule: 'RunAsAny'
```

Restricted

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restricted
...
spec:
  privileged: false
  # Required to prevent escalations to root.
  allowPrivilegeEscalation: false
  # This is redundant with non-root + disallow privilege escalation,
  # but we can provide it for defense in depth.
  requiredDropCapabilities:
    - ALL
  # Allow core volume types.
  volumes:
    - 'configMap'
    - 'emptyDir'
    - 'projected'
    - 'secret'
    - 'downwardAPI'
    # Assume that persistentVolumes set up by the cluster admin are safe to use.
    - 'persistentVolumeClaim'
  hostNetwork: false
  hostIPC: false
  hostPID: false
  runAsUser:
    # Require the container to run without root privileges.
    rule: 'MustRunAsNonRoot'
...
```

Hard To Manage & Harder To Enforce

- Every permutation/exception you want make = new PSP
  … and a new role
  … and a new role binding
  Managing access to the psp is managing how it’s applied
- An admitted workload shows which PSP admitted it
  … a failed one, go figure :(
- Unlike other controllers, it is not managing a specific aspect but a broad set of capabilities

Labels, Labels, Labels

- Labeling is a cornerstone in Kubernetes
- Label selector - Is the primitive for grouping objects
- Some of the places where label selectors are used in:
  - Scheduling workloads to nodes - e.g. schedule PCI workloads on PCI approved nodes using PCI label
  - Selecting services for workload - exposing the web component only through a service
  - Any kubectl command can use label selectors
- It is also used to “document” the objects (app, owner etc.)
  - Hint: Annotations are a better way for documenting


SIMPLE LABEL POLICY ADMISSION CONTROLLER
SHUT UP AND TAKE MY MONEY!
imgflip.com

Mutating Admission Controllers

- The first type of admission controllers to evaluate
- Multiple admission controllers can exist.
- They run serially, mutate the request and pass it forward
  - Only requests which meet the specific controller are evaluated
  - This should be used with caution as it changes the original request
- Receives the whole request with all fields as input
- Recommended to use for setting unset values rather than changing existing ones
  - Hint: Use these to automatically add missing labels

Validating Admission Controllers

- The last type admission controller to evaluate
- They run in parallel and each can fail the whole request
  - Only requests which meet the specific controller are evaluated
- Receives the whole request with all fields as input
- The good: Anything is possible...
- The bad: Anything is possible!

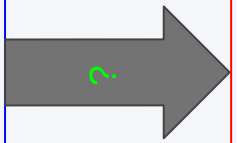
Sample Request & Response


```
{
  "apiVersion": "admission.k8s.io/v1",
  "kind": "AdmissionReview",
  "request": {
    "uid": "705ab4f5-6393-11e8-b7cc-42010a800002",

    "kind": {"group":"autoscaling","version":"v1","kind":"Scale"},
    "resource": {"group":"apps","version":"v1","resource":"deployments"},
    "subResource": "scale",
    "requestKind": {"group":"autoscaling","version":"v1","kind":"Scale"},
    "requestResource": {"group":"apps","version":"v1","resource":"deployments"},
    "requestSubResource": "scale",
    "name": "my-deployment",
    "namespace": "my-namespace",
    "operation": "UPDATE",

    "userInfo": {
      "username": "admin",
      "uid": "014fbff9a07c",
      "groups": ["system:authenticated","my-admin-group"],
      "extra": {
        "some-key":["some-value1", "some-value2"]
      }
    },

    "object": {"apiVersion":"autoscaling/v1","kind":"Scale",...},
    "oldObject": {"apiVersion":"autoscaling/v1","kind":"Scale",...},
    "options": {"apiVersion":"meta.k8s.io/v1","kind":"UpdateOptions",...},
    "dryRun": false
  }
}
```


```
{
  "apiVersion":
"admission.k8s.io/v1",
  "kind": "AdmissionReview",
  "response": {
    "uid": "<value from
request.uid>",
    "allowed": false,
    "status": {
      "code": 403,
      "message": "You cannot do this
because..."
    }
  }
}
```

Pinpoint Admission Controllers

- Some admission controllers are controlling a single focused aspect of the process, hence their policy is very simple and it is enforced cluster-wide
- To name a few (some are alpha/beta):
  - AlwaysPullImages
  - DefaultStorageClass
  - EventRateLimit
  - PodNodeSelector

Recap

- Admission controllers are a great mechanism to enforce policies
- Their policies are managed in many places making it complex to understand the actual effective policy
- Moving to multi cloud and multi cluster complicates things even more

4 Things To Do Today

- Check which admission controllers are enabled
  - kubectl get pods -n kube-system | grep apiserver
  - kubectl describe pods <apiserver pod> -n kube-system | grep admission
  - The above is for vanilla kubernetes.
  - For each cloud provider check the specific documentation
- Start namespacing, it will make your policy management better and easier
- Stop using “default” service account name - use named accounts
- Enable basic PodSecurityPolicy using the restricted, baseline, privileged example
  - https://kubernetes.io/docs/concepts/policy/pod-security-policy/

POLICIES IN ACTION DEMO

So What’s The Problem?

- There is a clear need to decouple policy from the controllers’ code
- Kubernetes dynamic nature teaches us that policies evolve and will continue to evolve over time
- Multi cluster & cloud expansion will push the need for a centrally managed and orchestrated policies even further
- Too many enforcement points and overlaps between controllers can end up in an all out chaos
- It is highly recommended to manage and enforce the same policies from dev and all the way to production

Policy-As-Code

A good first step in the right direction!

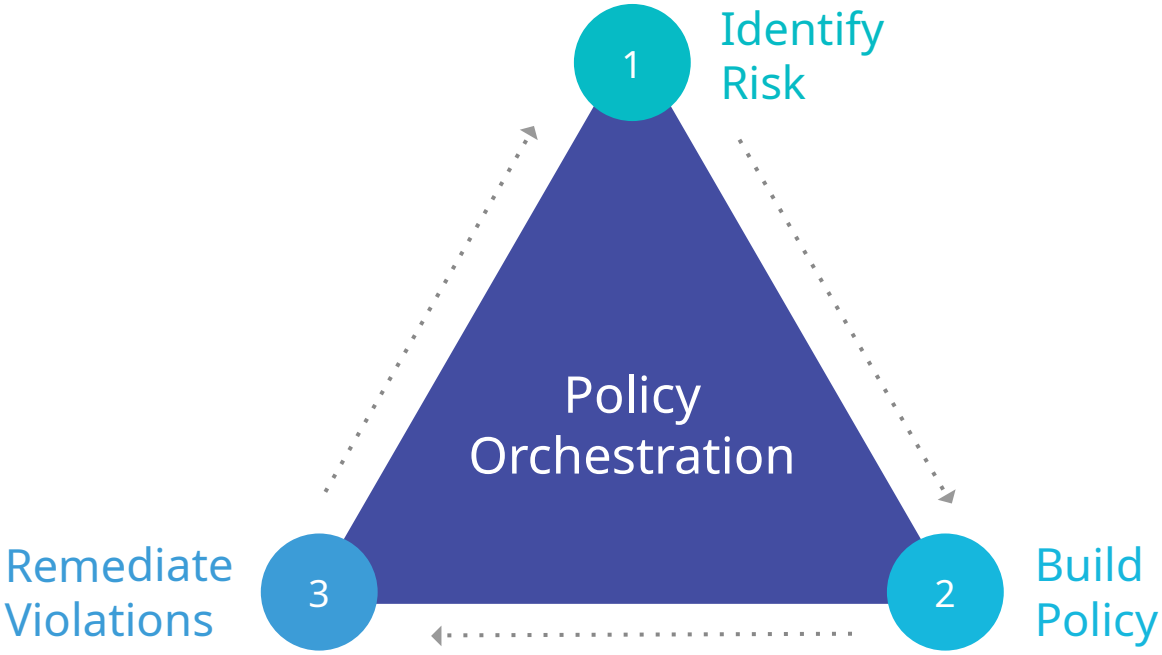
- Policy-as-code allows to decuple policy from code
- Enables declarative policy

- Some solutions works with Kubernetes natively
- OPA Gatekeeper - a great example and a good place to start

Consistent Actions > Consistent Results

- Admission controllers are not enough
- They are the last gate keepers before the cluster
- It is important to have them but it’s more important to identify issues way ahead
- The same policy guiding admission controller must be enforced in earlier stages (Git & CD processes)

Risk To Policy To Remediation -



apolicy

Kubernetes Policy, Risk Maturity Model

apolicy

It’s time for

Better Kubernetes

Be Risk Smart

Assess workload exposure and prioritize risks for action

Be Declarative

Achieve the workload state you’ve declared

Be Right

Prevent issues before they arise

apolicy