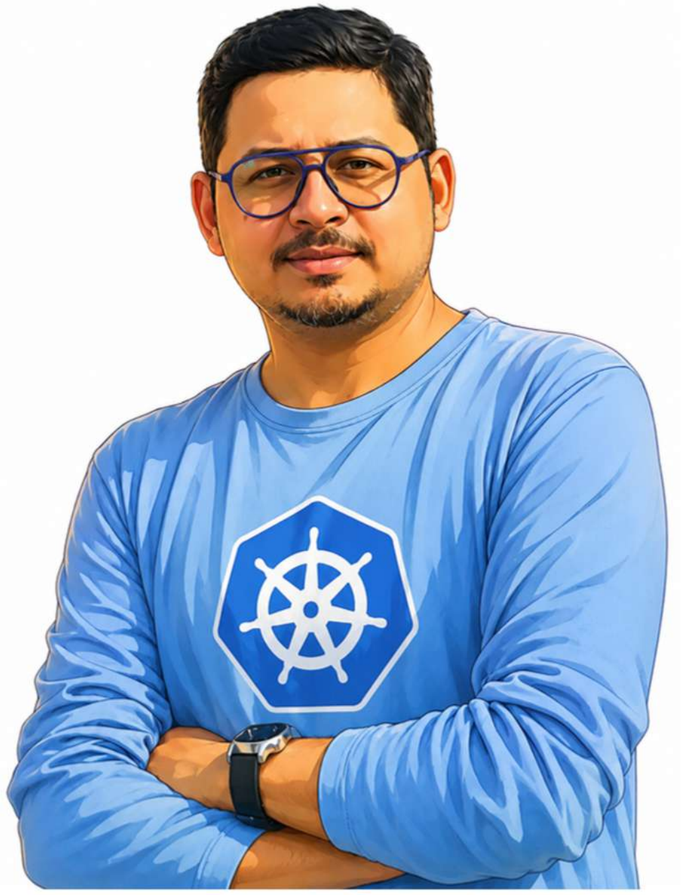
KubeCon | CloudNativeCon India 2026

Kubernetes API Server Performance Clinic

Auditing, Priority & Fairness in Production

Suman Chakraborty

- Solutions Architect at Platform9 Systems
- Advising Enterprises on Kubernetes & DevOps Adoption
- CD Foundation Ambassador
- Calico Big Cat Ambassador
- Speaker at 10+ global conferences, KubeCon, OSS Summit, DockerCon
- Devops Community Advocate/Tech Blogger



kubestronaut

CERTIFIED CLOUD NATIVE PLATFORM ENGINEER

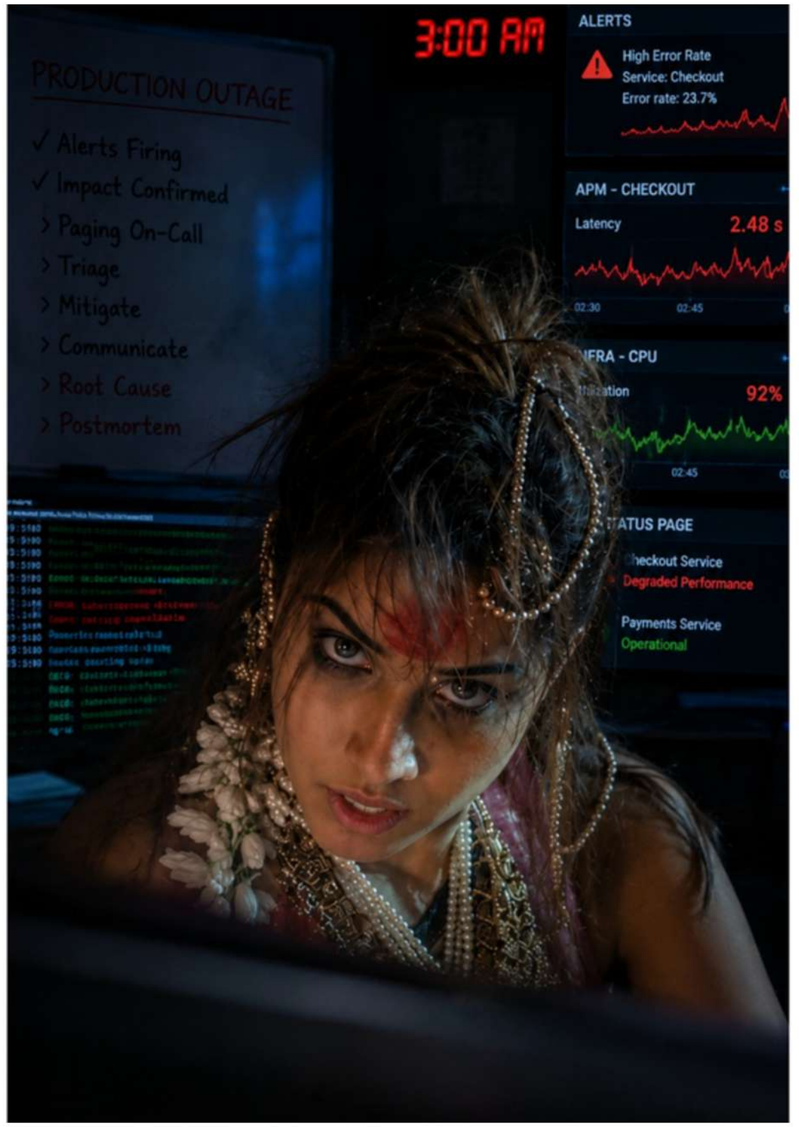
Google Cloud Certified CLOUD ARCHITECT

Neel Shah - DevOps Community Guy

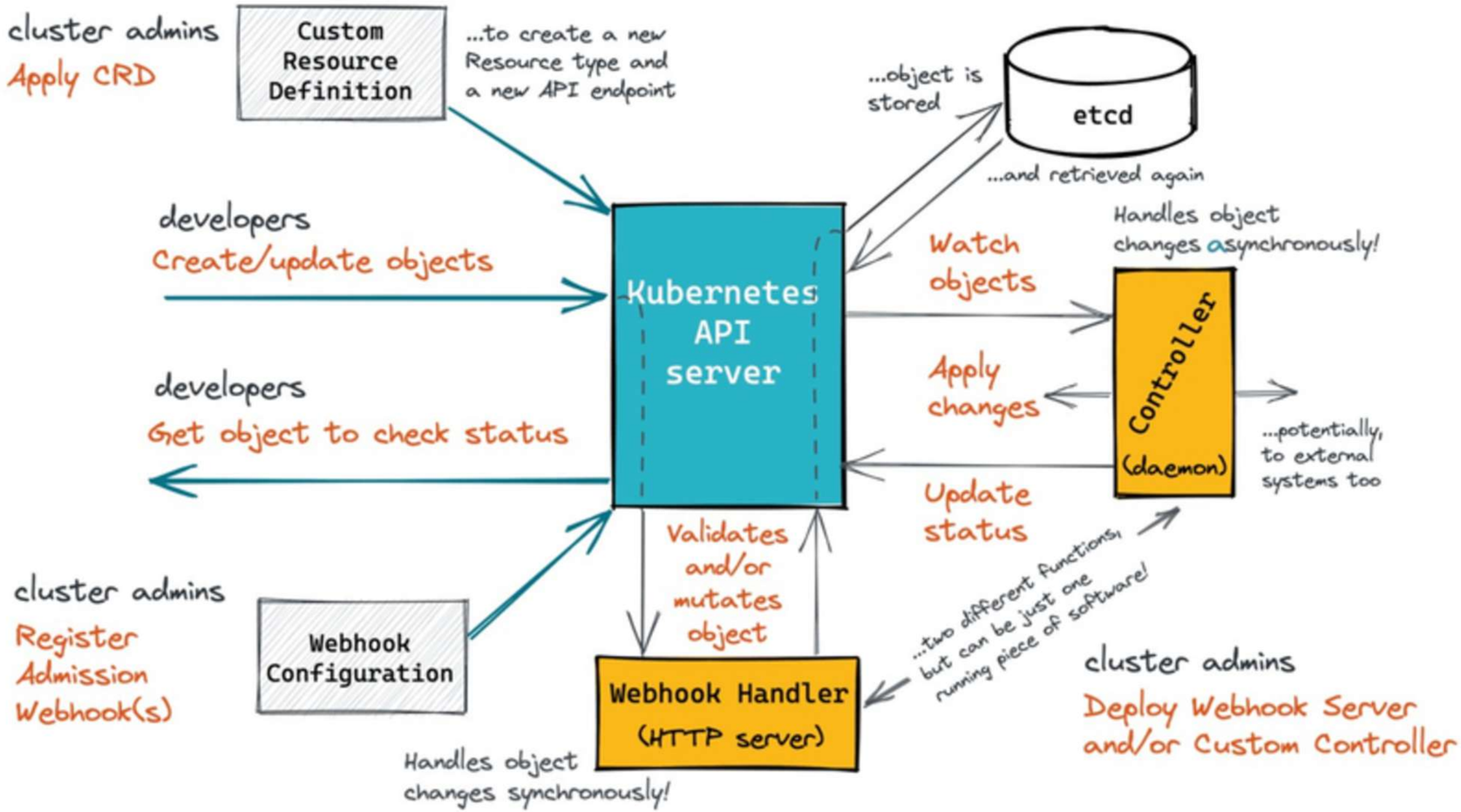
- Developer Advocate at StackGen
- Co-organiser GDG Cloud Gandhinagar, CNCF and Hashicorp Gandhinagar
- Mentored more than 15+ hackathons
- Gave talks in 30+ conferences, including KubeCon, PlatformCon, LinuxFest, KCD, etc.

The Production Nightmare

Everything looked healthy...
until deployments froze



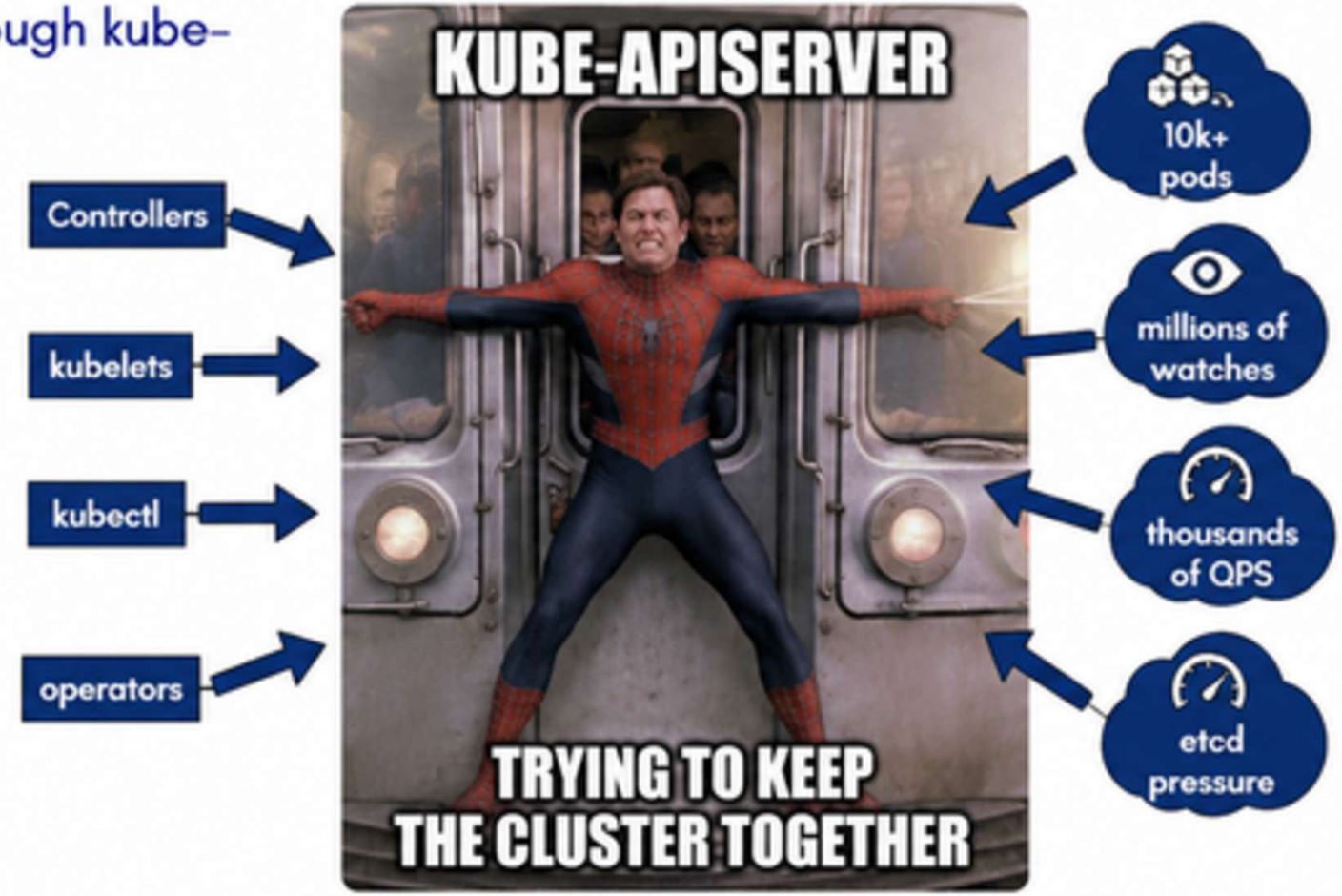
The API Server becomes
your distributed system
bottleneck before you
realize it.


cluster admins
Apply CRD
Custom Resource Definition
...to create a new Resource type and a new API endpoint
...object is stored
etcd
...and retrieved again
developers
Create/update objects
Kubernetes API server
Watch objects
Handles object changes asynchronously!
Controller
(daemon)
Apply changes
...potentially, to external systems too
developers
Get object to check status
Update status
Validates and/or mutates object
cluster admins
Register Admission Webhook(s)
Webhook Configuration
...two different functions, but can be just one running piece of software!
Webhook Handler
(HTTP server)
cluster admins
Deploy Webhook Server and/or Custom Controller
Handles object changes synchronously!

Why API Server Performance Matters

Everything goes through kube-apiserver:

- Controllers
- kubelets
- kubectl
- operators



Key Metrics at Scale

- 10k+ pods
- millions of watches
- thousands of QPS
- etcd pressure

BIGGEST OFFENDERS
THAT CAUSE API SERVER OVERLOAD

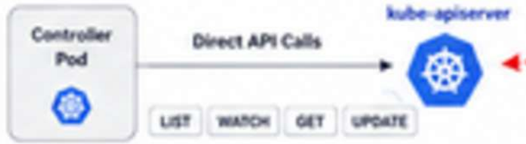
1. MASSIVE LIST CALLS

Large payloads, high memory & CPU

2. INEFFICIENT WATCH PATTERNS

Too many watches, no field/label selection

3. CONTROLLERS WITHOUT CACHING



No shared informer cache = High QPS

4. CUSTOM OPERATORS POLLING AGGRESSIVELY

Continuous polling instead of using WATCH

5. THUNDERING HERD RECONNECTS

Network blip / apiserver restart / LB event causes stampede

6. BAD RETRY LOGIC

7. OVERLOADED ADMISSION WEBHOOKS

Slow / unavailable webhooks block requests (synchronous)

Why LIST Requests Hurt So Much

A LIST request:

- hits etcd
- serializes huge payloads
- consumes memory
- blocks CPU

Without Proper Traffic Management

A single misbehaving client or a surge can

- **Starve** critical operations
- Create **cascading** failures
- **Block** incident response
- **Degrade** application health

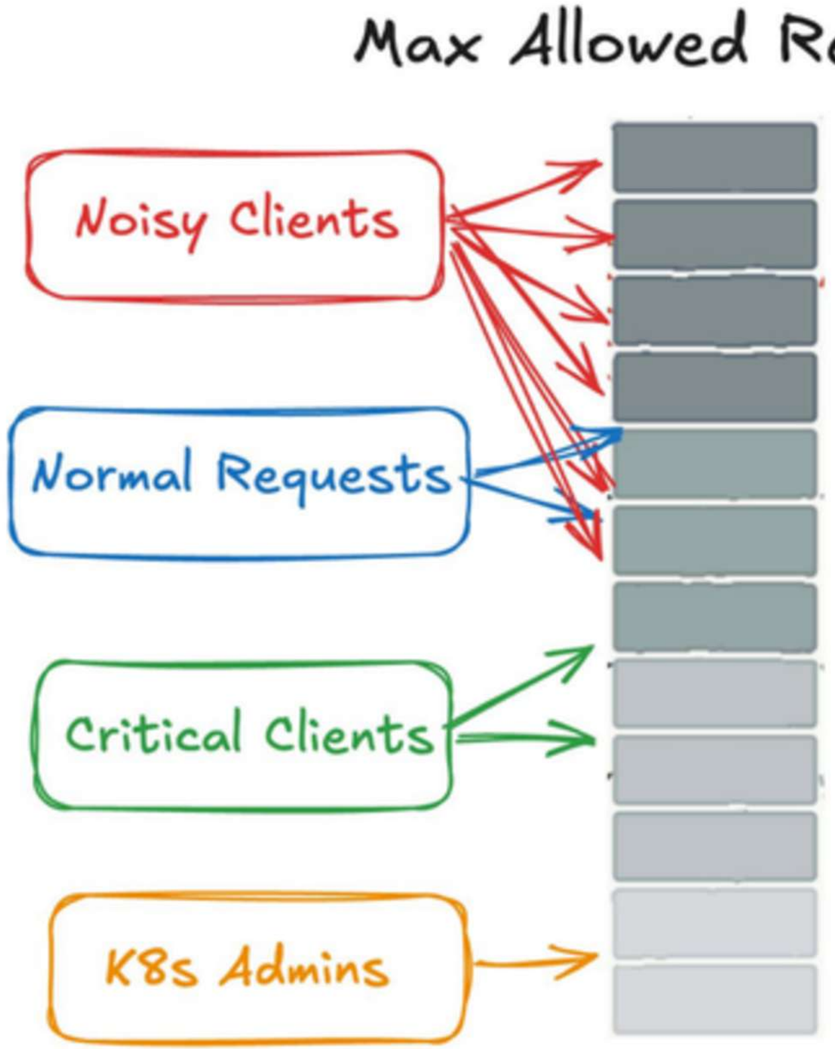
Existing kube-apiserver controls ***--max-requests-inflight*** & ***--max-mutating-requests-inflight*** limit inbount requests but can't categorize traffic priorities

Kubernetes API Priority & Fairness

- API Priority and Fairness (APF) in Kubernetes, classifies and isolates API server requests in a more fine-grained way, introducing prioritization, queuing, and isolation. So critical requests always get through and no single client can starve the rest.

- Enabled by default (*--enable-priority-and-fairness*) in Kubernetes v1.29

The Problem — Before APF



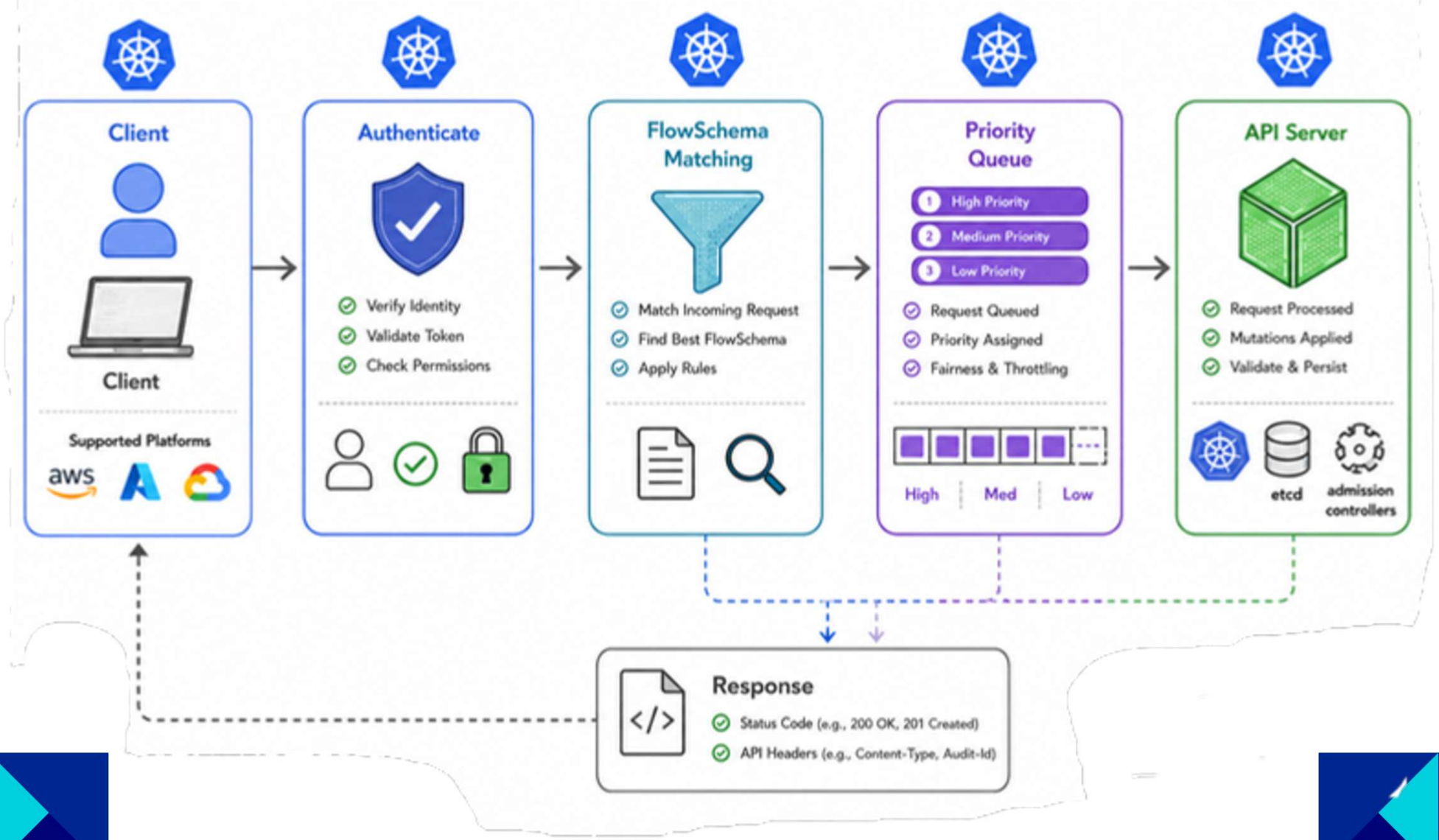
No awareness of who

Noisy Clients consume all capacity

Critical requests have zero protection and instantly dropped

No protection for the kube-apiserver

APF Architecture



APF Goals

Fair Sharing

Max Allowed Requests

Capacity distributed equitably among all the clients

Prioritization

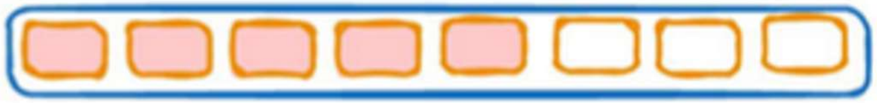
Critical Requests ↑ High Priority

Non - Critical Requests ↓ Low Priority

Queuing

Max Allowed Requests - FULL

Queue



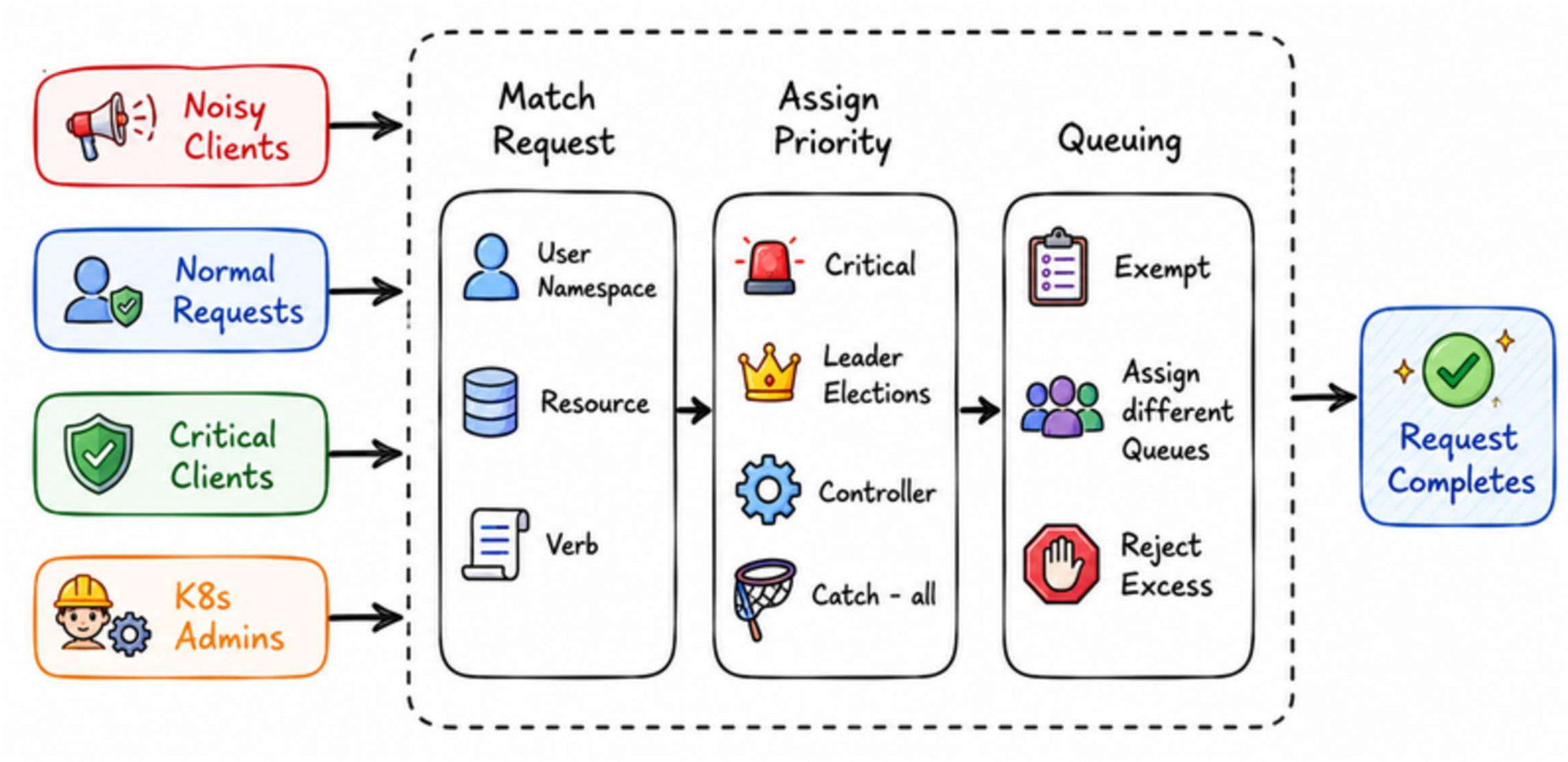
Excess requests are queued instead of getting dropped

Flow Isolation

Limit expensive and Noisy requests

Protect essential and critical requests with their own flow

APF High Level Flow



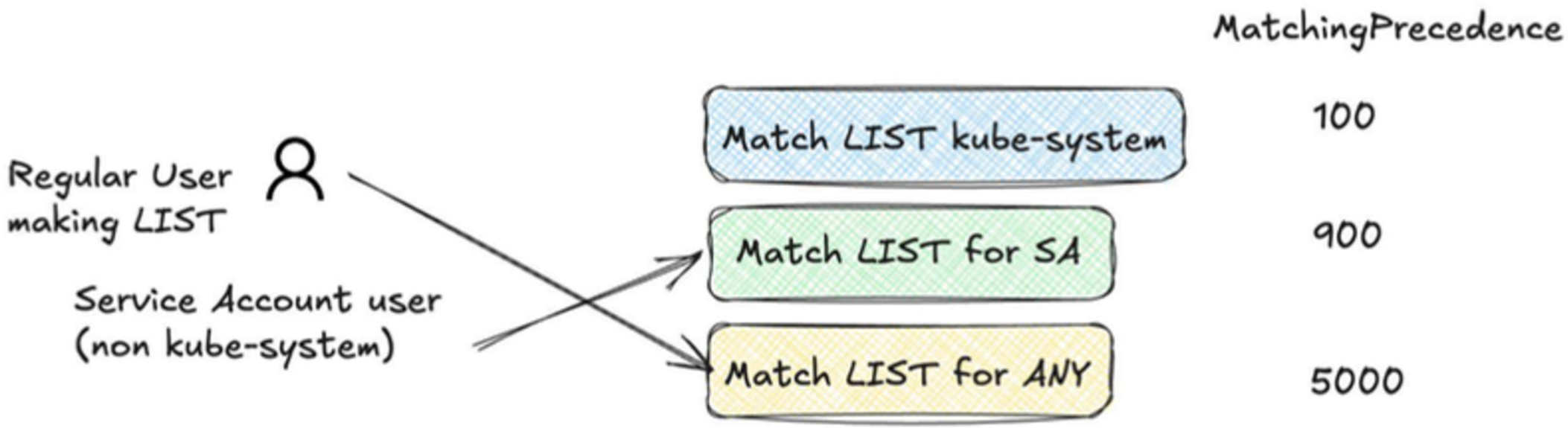
APF Resources

Flow Schema : The Traffic Classifier

- Matches incoming API requests against rules (user, verb, resource, namespace)
- Routes them to the appropriate PriorityLevelConfiguration. a little bit of body text

```
→  ~ kubectl get flowschema
NAME                           PRIORITYLEVEL     MATCHINGPRECEDENCE   DISTINGUISHERMETHOD   AGE     MISSINGPL
exempt                         exempt            1                    <none>                9m49s   False
probes                         exempt            2                    <none>                9m49s   False
system-leader-election         leader-election   100                  ByUser                9m49s   False
endpoint-controller            workload-high     150                  ByUser                9m49s   False
workload-leader-election       leader-election   200                  ByUser                9m49s   False
system-node-high               node-high         400                  ByUser                9m49s   False
system-nodes                   system            500                  ByUser                9m49s   False
kube-controller-manager        workload-high     800                  ByNamespace           9m49s   False
kube-scheduler                 workload-high     800                  ByNamespace           9m49s   False
kube-system-service-accounts   workload-high     900                  ByNamespace           9m49s   False
service-accounts               workload-low      9000                 ByUser                9m49s   False
```

FlowSchema Matching Precendence



FlowSchema Structure

```yaml
apiVersion: flowcontrol.apiserver.k8s.io/v1beta3
kind: FlowSchema
metadata:
  name: example-flow-schema
spec:
  matchingPrecedence: 1000  # Lower = higher priority
  priorityLevelConfiguration:
    name: workload-high
  distinguisherMethod:
    type: ByUser
  rules:
    - subjects:
        - kind: User
          user:
            name: "admin"
      resourceRules:
        - verbs: ["*"]
          apiGroups: ["*"]
          resources: ["*"]
          namespaces: ["production"]
```

APF Resources

PriorityLevelConfiguration: The Queue Manager

- Defines a priority tier, concurrency share, and queues.
- Handle requests when over capacity. i.e. admit, queue, or reject.

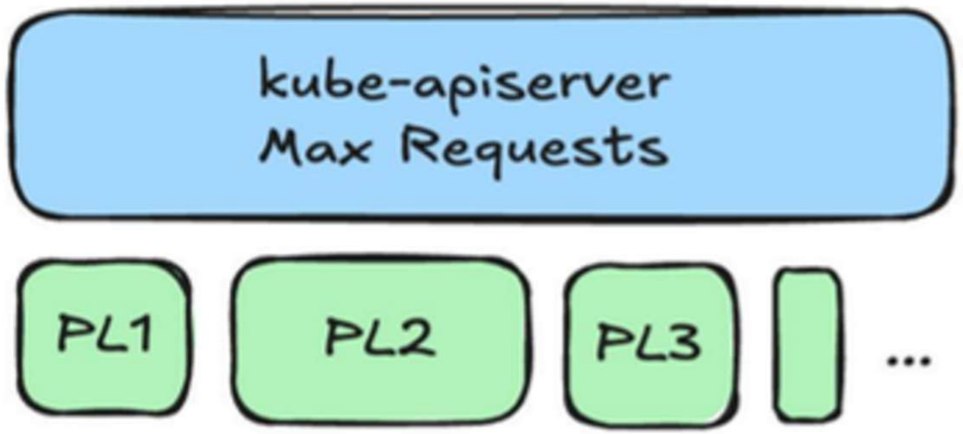
```
→  ~ kubectl get prioritylevelconfiguration
NAME              TYPE      NOMINALCONCURRENCYSHARES   QUEUES   HANDSIZE   QUEUELENGTHLIMIT   AGE
catch-all         Limited   5                          <none>   <none>     <none>             2m50s
exempt            Exempt    <none>                     <none>   <none>     <none>             2m50s
global-default    Limited   20                         128      6          50                 2m50s
leader-election   Limited   10                         16       4          50                 2m50s
node-high         Limited   40                         64       6          50                 2m50s
system            Limited   30                         64       6          50                 2m50s
workload-high     Limited   40                         128      6          50                 2m50s
workload-low      Limited   100                        128      6          50                 2m50s
```

PriorityLevelConfiguration Structure

```yaml
apiVersion: flowcontrol.apiserver.k8s.io/v1beta3
kind: PriorityLevelConfiguration
metadata:
  name: workload-high
spec:
  type: Limited
  limited:
    nominalConcurrencyShares: 100
    limitResponse:
      type: Queue
      queuing:
        queues: 128
        queueLengthLimit: 50
        handSize: 6
```

Concurrency/Seats

To calculate concurrency for PriorityLevel catch-all having 5 nominalConcurrencyShares Max allowed requests = 600
Sum(nominalConcurrencyShares) = 245
concurrency = 600 * 5/245 = 12



$$\text{concurrency (seats)} = \text{apiserver-max-requests} * \frac{\text{nominalConcurrencyShares}}{\text{sum(nominalConcurrencyShares)}}$$

Queue Allocation

Queues

More Queues -> Fewer collisions

Queues: 1 -> Disables fair queuing

queueLengthLimit

Low limit

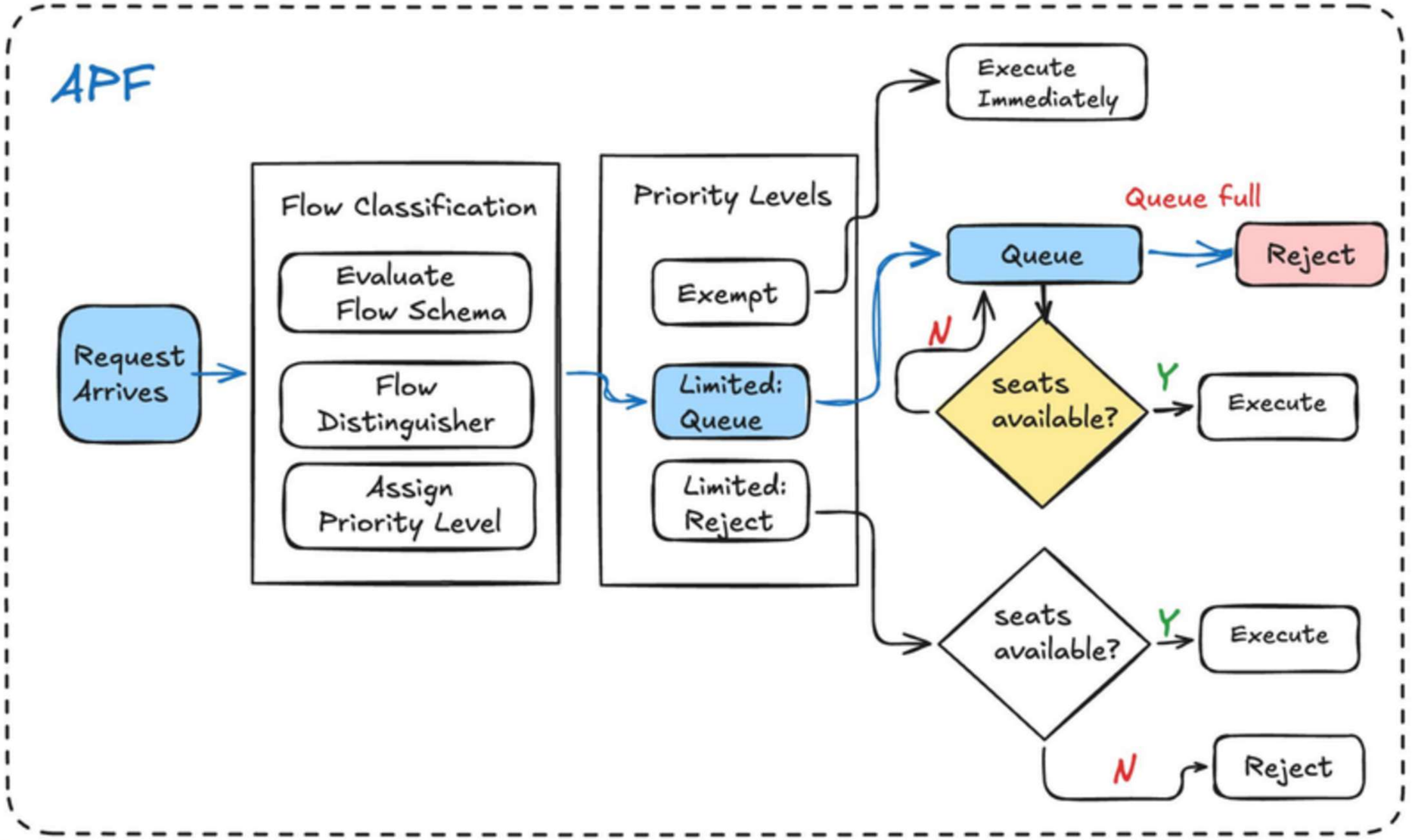
Sudden bursts -> requests get dropped

High limit

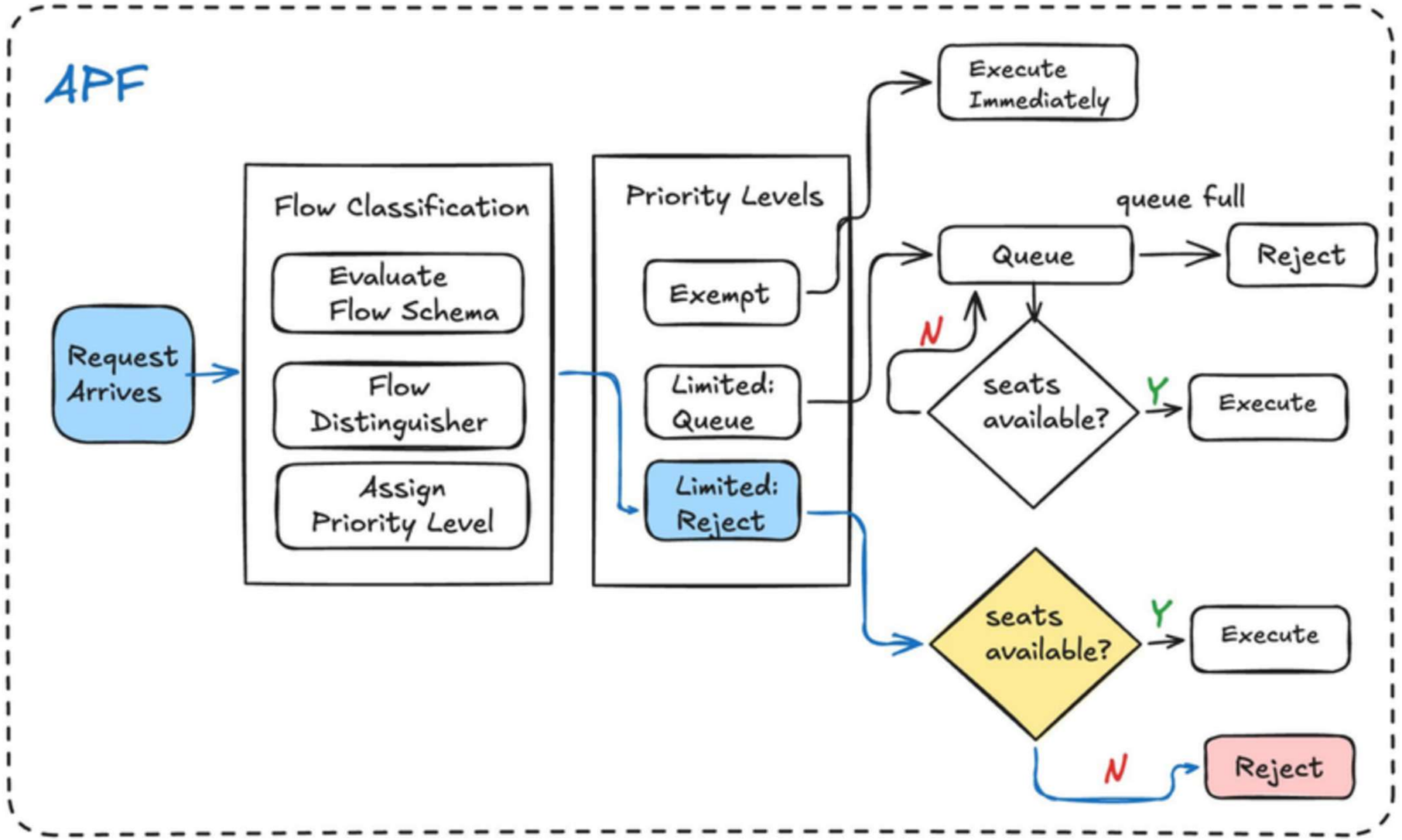
Sudden bursts -> requests are queued

↑ Memory and Latency

Request Flow - Queue



Request Flow - Reject



APF Metrics

1. apiserver_flowcontrol_dispatched_requests_total
2. apiserver_flowcontrol_current_inqueue_requests
3. apiserver_flowcontrol_rejected_requests_total
4. apiserver_flowcontrol_current_executing_requests
5. apiserver_flowcontrol_current_executing_seats
6. apiserver_flowcontrol_request_wait_duration_seconds

Its time for DEMO !!!!

APF Best Practises

- **Start with Defaults** – Customize only when necessary
- **Throttle / limit** expensive LIST requests
- Ensure **non-critical** clients have lower priority
- Assign **higher priority levels** to essential system components
- **Use Distinguishers Smartly** – Prefer *ByUser* for better isolation.
- **Avoid Exempt Priority** – Reserve for critical control-plane services
- **Document FlowSchemas** – Capture purpose, owners, and dependencies

Resources

API Priority and Fairness Doc

- https://kubernetes.io/docs/concepts/cluster-administration/flow-control/
- https://github.com/kubernetes/enhancements/blob/master/keps/sig-api-machinery/1040-priority-and-fairness/README.md

Blogs

- https://medium.com/@rifewang/introduction-to-kubernetes-apf-api-priority-and-fairness-cce09f843523
- https://medium.com/@salwan.mohamed/mastering-kubernetes-api-priority-and-fairness-from-chaos-to-control-7449a3ea2e78

Demo Resources

THANK YOU

Connect with us for any queries !

Suman Chakraborty

Neel Shah