Extending Kubernetes Self-Healing for Networking Failures

Romit Mohane
Shardul Junagade
Shounak Ranade
Rishabh Jogani

Background and Abstract

- Kubernetes can recover from most workload failures but **struggles with network-level issues** such as interface faults, congestion, and CoreDNS outages.
- This work analyzes how such failures impact cluster operation through targeted **fault-injection experiments** and introduces a **custom self-healing operator that detects and restores DNS connectivity**, extending Kubernetes recovery to the networking layer.

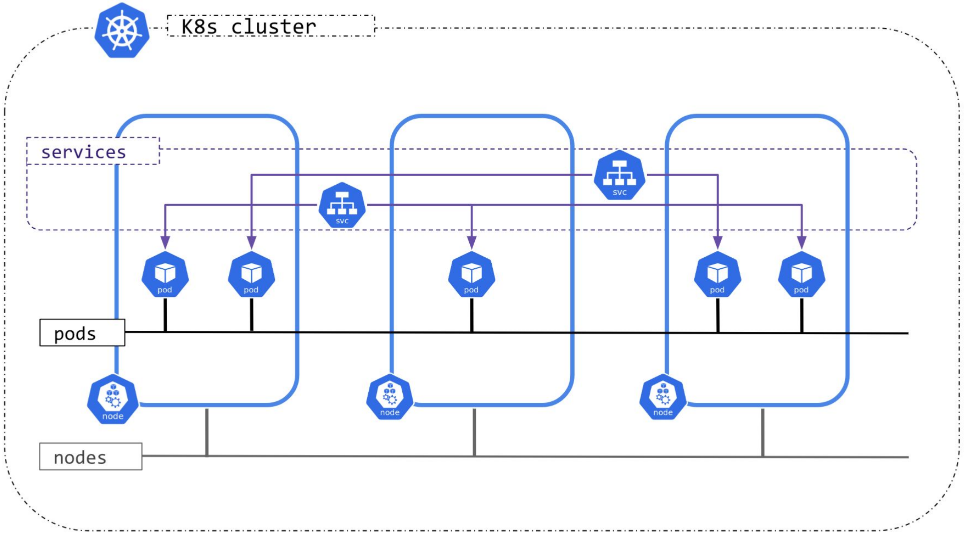


Fig 1. K8s Architecture

Experimental Setup

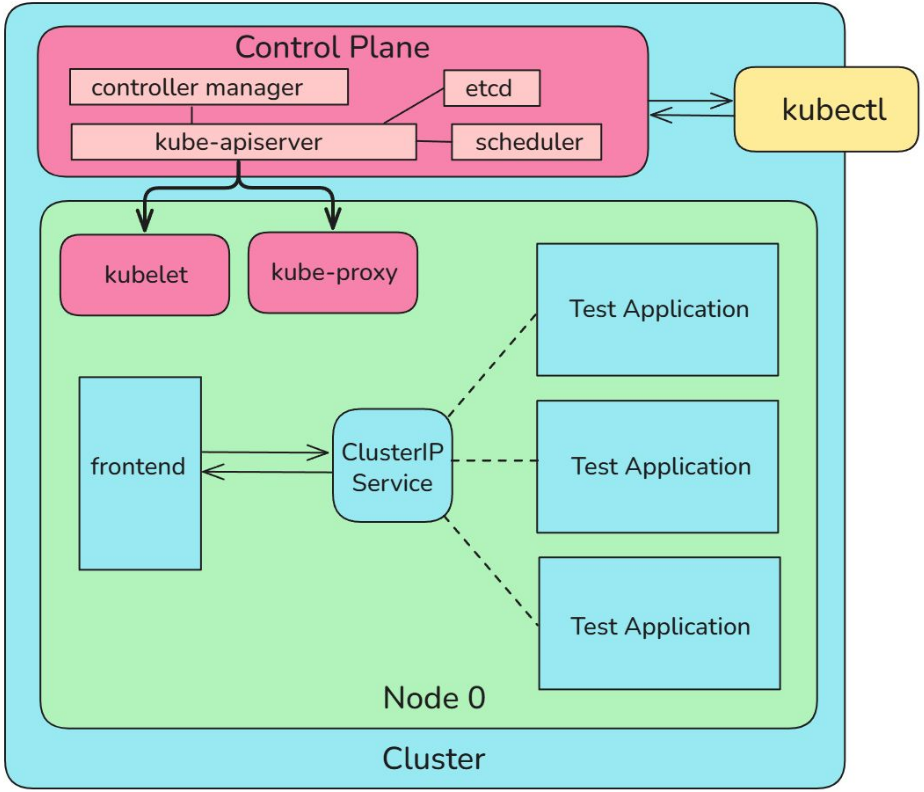


Fig 2. Architecture Diagram of Test Setup

- Single-node **Minikube** cluster used for controlled testing
- Contains **frontend (curlimage), backend service (ClusterIP)** and three **test applications (flask)**
- **CoreDNS** and **bridge interface** form core of network communication
- **kubectl** used to trigger and monitor experiment

Observation and Measurement Tools

- **dnsutils** pod for DNS queries and latency measurements
- **tc (Traffic Control)** for network fault injection
- **kubectl logs / top / get pods** for observing recovery
- **Minikube dashboard** for overall monitoring
- Latency + success rate logged to text/CSV for analysis

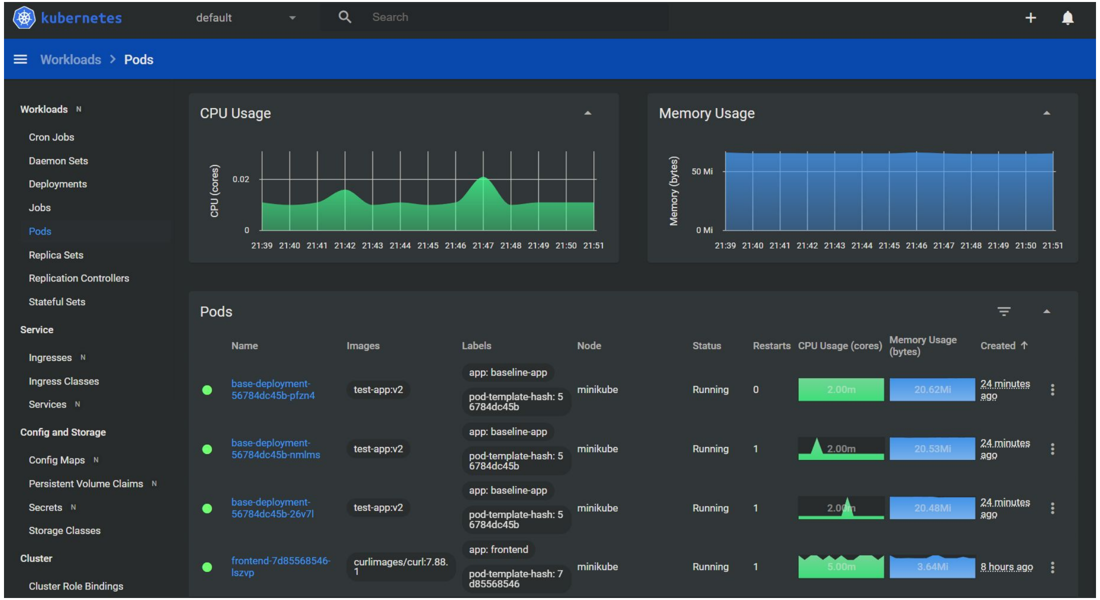
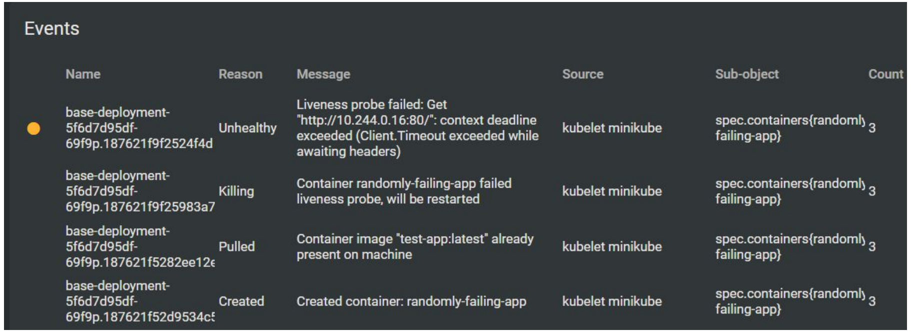


Image 1. K8s dashboard

How Kubernetes Handles Failures

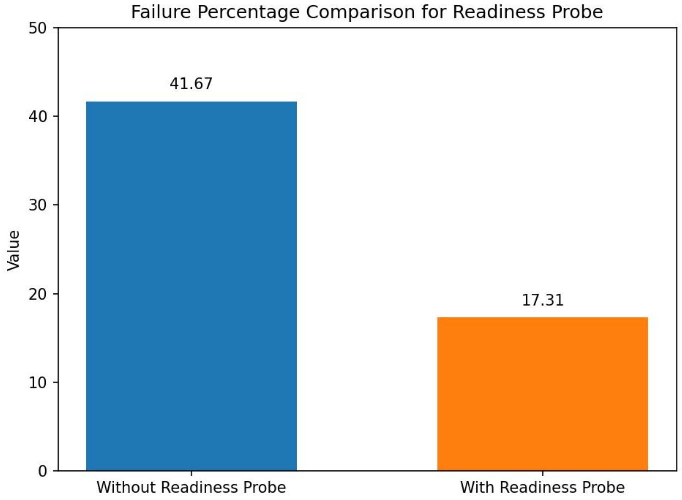
- **Pod-level repairs.**
  - If a container fails liveness probe, the kubelet creates a replacement pod automatically.
  - If container fails readiness probe, the pod is removed from service endpoints (notReady), this improves overall service performance.



Events

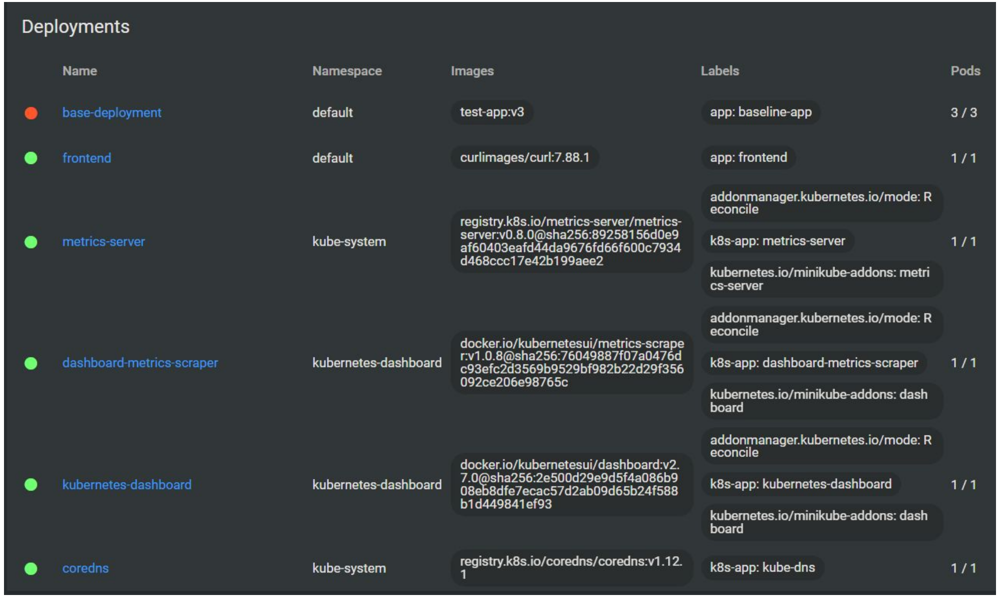
	Name	Reason	Message	Source	Sub-object	Count
●	base-deployment-5f6d7d95df-69f9p.187621f9f2524f4d	Unhealthy	Liveness probe failed: Get "http://10.244.0.16:80/": context deadline exceeded (Client.Timeout exceeded while awaiting headers)	kubelet minikube	spec.containers{randomly-failing-app}	3
	base-deployment-5f6d7d95df-69f9p.187621f9f25983a7	Killing	Container randomly-failing-app failed liveness probe, will be restarted	kubelet minikube	spec.containers{randomly-failing-app}	3
	base-deployment-5f6d7d95df-69f9p.187621f5282ee12e	Pulled	Container image "test-app:latest" already present on machine	kubelet minikube	spec.containers{randomly-failing-app}	3
	base-deployment-5f6d7d95df-69f9p.187621f52d9534c5	Created	Created container: randomly-failing-app	kubelet minikube	spec.containers{randomly-failing-app}	3

Image 2. Dashboard showing liveness probe failure



How Kubernetes Handles Failures

- Services or Pods running as a **Deployment** will be re-created if a pod crashes.
- The pods which run as **DaemonSets** are re-created on the same node if they crash.



Deployments

Name	Namespace	Images	Labels	Pods
base-deployment	default	test-app:v3	app: baseline-app	3 / 3
frontend	default	curlimages/curl:7.88.1	app: frontend	1 / 1
metrics-server	kube-system	registry.k8s.io/metrics-server/metrics-server:v0.8.0@sha256:89258156d0e9af60403eafd44da9676fd66f600c7934d468ccc17e42b199aee2	addonmanager.kubernetes.io/mode: Reconcile k8s-app: metrics-server kubernetes.io/minikube-addons: metrics-server	1 / 1
dashboard-metrics-scraper	kubernetes-dashboard	docker.io/kubernetesui/metrics-scraper:v1.0.8@sha256:76049887f07a0476dc93efc2d3569b9529bf982b22d29f356092ce206e98765c	addonmanager.kubernetes.io/mode: Reconcile k8s-app: dashboard-metrics-scraper kubernetes.io/minikube-addons: dashboard	1 / 1
kubernetes-dashboard	kubernetes-dashboard	docker.io/kubernetesui/dashboard:v2.7.0@sha256:2e500d29e9d5f4a086b908eb8dfe7ecac57d2ab09d65b24f588b1d449841ef93	addonmanager.kubernetes.io/mode: Reconcile k8s-app: kubernetes-dashboard kubernetes.io/minikube-addons: dashboard	1 / 1
coredns	kube-system	registry.k8s.io/coredns/coredns:v1.12.1	k8s-app: kube-dns	1 / 1

Image 3. Dashboard showing status of Deployments

What Kubernetes does not do

- Does **not** diagnose or repair low-level network faults (for example: a down `eth0 or bridge`)

```
docker@minikube:~$ sudo ip link set bridge down
docker@minikube:~$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0@if8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether ee:39:17:6d:b9:e8 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.49.2/24 brd 192.168.49.255 scope global eth0
       valid_lft forever preferred_lft forever
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 72:c6:6b:59:1d:04 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
4: bridge: <BROADCAST,MULTICAST> mtu 1500 qdisc noqueue state DOWN group default qlen 1000
    link/ether 36:1b:04:53:17:ad brd ff:ff:ff:ff:ff:ff
    inet 10.244.0.1/16 brd 10.244.255.255 scope global bridge
       valid_lft forever preferred_lft forever
5: vethbd4fd851@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master bridge state UP group default
    link/ether f6:20:ea:09:78:2d brd ff:ff:ff:ff:ff:ff link-netnsid 1
    inet6 fe80::f420:eaff:fe09:782d/64 scope link
       valid_lft forever preferred_lft forever
6: veth61edcfaf@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master bridge state UP group default
    link/ether 2e:a1:a4:2c:4d:b2 brd ff:ff:ff:ff:ff:ff link-netnsid 2
    inet6 fe80::2ca1:a4ff:fe2c:4db2/64 scope link
       valid_lft forever preferred_lft forever
8: vethad9c7be1@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master bridge state UP group default
    link/ether 6a:67:05:8b:29:64 brd ff:ff:ff:ff:ff:ff link-netnsid 4
    inet6 fe80::6867:5ff:fe8b:2964/64 scope link
       valid_lft forever preferred_lft forever
```

```
^Cromit@Romits-Laptop://home/romit/cn$ kubectl logs -l app=fr
2025-11-12T15:53:01+00:00 FAIL
2025-11-12T15:53:03+00:00 OK
2025-11-12T15:53:05+00:00 FAIL
2025-11-12T15:53:07+00:00 FAIL
2025-11-12T15:53:12+00:00 FAIL
2025-11-12T15:53:17+00:00 FAIL
2025-11-12T15:53:22+00:00 FAIL
2025-11-12T15:53:27+00:00 FAIL
2025-11-12T15:53:32+00:00 FAIL
2025-11-12T15:53:37+00:00 FAIL
2025-11-12T15:53:42+00:00 FAIL
2025-11-12T15:53:47+00:00 FAIL
2025-11-12T15:53:52+00:00 FAIL
```

Frontend logs

What Kubernetes does not do

- Does **not** diagnose or repair low-level network faults (for example: a down `eth0 or bridge`)

```
docker@minikube:~$ sudo ip link set bridge down
docker@minikube:~$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0@if8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
    link/ether ee:39:17:6d:b9:e8 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.49.2/24 brd 192.168.49.255 scope global eth0
       valid_lft forever preferred_lft forever
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue
    link/ether 72:c6:6b:59:1d:04 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
4: bridge: <BROADCAST,MULTICAST> mtu 1500 qdisc noqueue state DOWN group def
    link/ether 36:1b:04:53:17:ad brd ff:ff:ff:ff:ff:ff
    inet 10.244.0.1/16 brd 10.244.255.255 scope global bridge
       valid_lft forever preferred_lft forever
5: vethbd4fd851@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueu
    link/ether f6:20:ea:09:78:2d brd ff:ff:ff:ff:ff:ff link-netnsid 1
    inet6 fe80::f420:eaff:fe09:782d/64 scope link
       valid_lft forever preferred_lft forever
6: veth61edcfaf@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueu
    link/ether 2e:a1:a4:2c:4d:b2 brd ff:ff:ff:ff:ff:ff link-netnsid 2
    inet6 fe80::2ca1:a4ff:fe2c:4db2/64 scope link
       valid_lft forever preferred_lft forever
8: vethad9c7be1@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master bridge state UP group default
    link/ether 6a:67:05:8b:29:64 brd ff:ff:ff:ff:ff:ff link-netnsid 4
    inet6 fe80::6867:5ff:fe8b:2964/64 scope link
       valid_lft forever preferred_lft forever
```

```
^Cromit@Romits-Laptop://home/romit/cn$ kubectl exec -it base-deployment-7675dd7d6-n56tj -- ping -c 2 10.244.0.150
PING 10.244.0.150 (10.244.0.150) 56(84) bytes of data.
64 bytes from 10.244.0.150: icmp_seq=1 ttl=64 time=1.97 ms
64 bytes from 10.244.0.150: icmp_seq=2 ttl=64 time=0.055 ms

--- 10.244.0.150 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.055/1.013/1.971/0.958 ms
romit@Romits-Laptop://home/romit/cn$ minikube ssh
docker@minikube:~$ sudo ip link set bridge down
docker@minikube:~$ exit
logout
romit@Romits-Laptop://home/romit/cn$ kubectl exec -it base-deployment-7675dd7d6-n56tj -- ping -c 2 10.244.0.150
PING 10.244.0.150 (10.244.0.150) 56(84) bytes of data.

--- 10.244.0.150 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1022ms

command terminated with exit code 1
```

Pod-to-pod communication

What Kubernetes does not do

- Many network problems remain unnoticed, like **network congestion** causing latency spikes.

We simulated this using linux's traffic control (tc) utility.

```
romit@Romits-Laptop://home/romit/cn$ minikube ssh
docker@minikube:~$ sudo tc qdisc add dev bridge root netem delay 200ms 50ms distribution normal
docker@minikube:~$ exit
logout
romit@Romits-Laptop://home/romit/cn$ kubectl exec -it dnsutils -- dig kubernetes.default

; <<>> DiG 9.16.27 <<>> kubernetes.default
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 8576
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 10250c56d338d979 (echoed)
;; QUESTION SECTION:
;kubernetes.default.            IN      A

;; Query time: 240 msec
;; SERVER: 10.96.0.10#53(10.96.0.10)
;; WHEN: Wed Nov 12 14:30:05 UTC 2025
;; MSG SIZE  rcvd: 59

romit@Romits-Laptop://home/romit/cn$ minikube ssh
docker@minikube:~$ sudo tc qdisc del dev bridge root
docker@minikube:~$ exit
logout
romit@Romits-Laptop://home/romit/cn$ kubectl exec -it dnsutils -- dig kubernetes.default

; <<>> DiG 9.16.27 <<>> kubernetes.default
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 33985
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: f0f0b3c9f258893d (echoed)
;; QUESTION SECTION:
;kubernetes.default.            IN      A

;; Query time: 20 msec
;; SERVER: 10.96.0.10#53(10.96.0.10)
;; WHEN: Wed Nov 12 14:30:41 UTC 2025
;; MSG SIZE  rcvd: 59
```

What Kubernetes does not do

- Automatically detect a **lack** of CoreDNS pods or failure of DNS resolution due to bad configuration.

```
^Cromit@Romits-Laptop://home/romit/cnkubectl scale deployment coredns --replicas=0 -n kube-system
deployment.apps/coredns scaled
romit@Romits-Laptop://home/romit/cn$ kubectl exec -it dnsutils --  nslookup kubernetes.default
;; connection timed out; no servers could be reached


command terminated with exit code 1
```

We simulated this by scaling down the coredns pods, and using nslookup from the dnsutils pod.

How good are current solutions

- Most current solutions aim to reduce the delay caused by waiting for pods and nodes to fail
- Available solutions try to monitor and forecast possible failures using unsupervised anomaly detection techniques before the pods actually fail, saving time.
- Thereafter, solutions either restart pods preemptively or use machine learning algorithms to decide what alternate fixes can be used.

Our Solution

- Closed a real gap: Kubernetes doesn’t self-heal CoreDNS degradation.
- Built a lightweight operator that probes DNS end-to-end (nslookup Job).
- Enforced a replica floor for CoreDNS and recycled unready pods.
- Triggered remediation after N consecutive probe failures.

```
● shardul@LAPTOP-2FC5USH0:~/cn/CN_Project_kubernetes_self-healing/coredns-operator$ kubectl scale deployment coredns -n kube-system --replicas=0
  deployment.apps/coredns scaled
● shardul@LAPTOP-2FC5USH0:~/cn/CN_Project_kubernetes_self-healing/coredns-operator$ kubectl exec -it dnsutils -- nslookup example.com
  Server:         10.96.0.10
  Address:        10.96.0.10#53

  Non-authoritative answer:
  Name:   example.com
  Address: 23.220.75.232
  Name:   example.com
  Address: 23.220.75.245
  Name:   example.com
  Address: 23.192.228.80
  Name:   example.com
  Address: 23.192.228.84
  Name:   example.com
  Address: 23.215.0.138
  Name:   example.com
  Address: 23.215.0.136
```

Even after scaling the number of DNS replicas to 0, our custom operator created new DNS pods and ensured timely DNS resolution

Architecture and Operator Pipeline

DNSMonitor CRD

```yaml
apiVersion: infra.sharduljunagade.github.io/v1alpha1
kind: DNSMonitor
metadata:
  name: dns-monitor
  namespace: default
spec:
  namespace: kube-system
  probeIntervalSeconds: 30
  testDomain: "kubernetes.default.svc.cluster.local"
  failureThreshold: 3
  desiredReplicas: 2
```

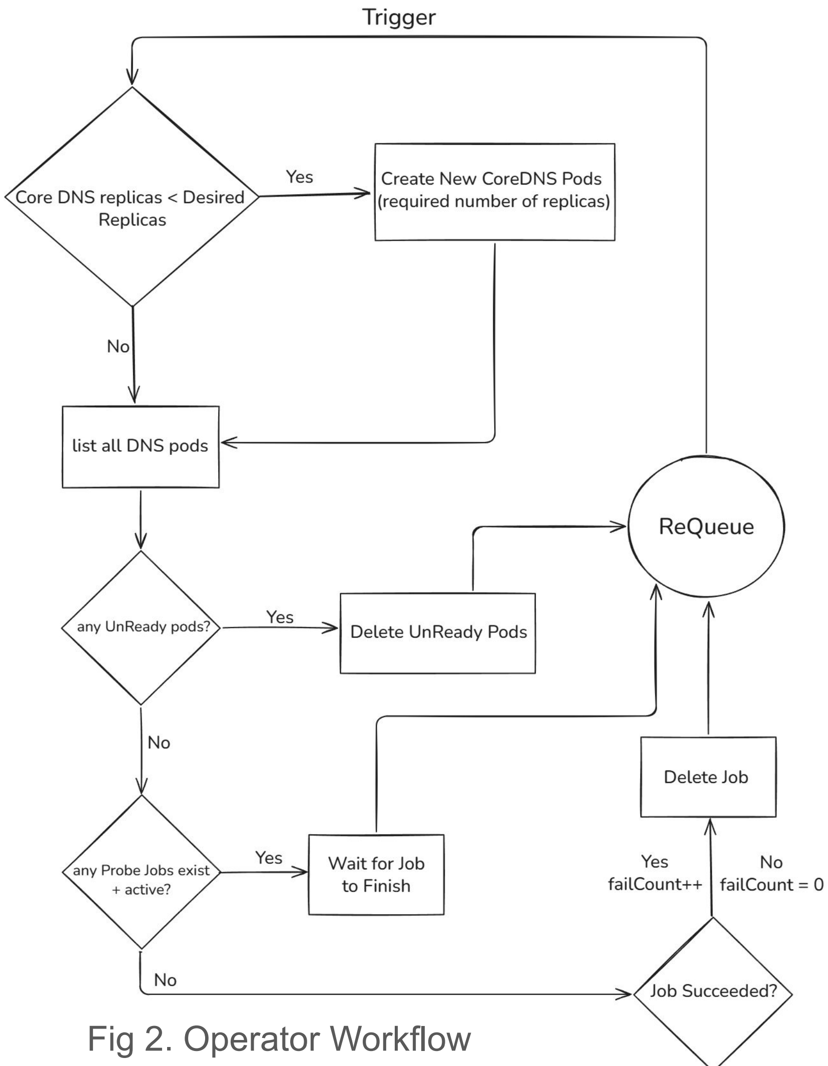


Fig 2. Operator Workflow

Conclusion and Future Scope

- Explored multiple cases of **network related failures** in kubernetes clusters.
- Developed a **custom operator** for DNS fault detection and automated healing
- Extends Kubernetes self-healing to **network-level resilience**
- **Bridge failures** remain undetected and unrecovered, and can be worked upon in the future
- **Multi-Node clusters** pose an extended set of Networking-failures, which have to be explored and can be potentially worked upon in a full **Project Course**.