Enhancing Kubernetes Self-Healing for Networking Failures

Romit Mohane*, Shardul Junagade*, Shounak Ranade*, and Rishabh Jogani*
Email: {romit.mohane, shardul.junagade, shounak.ranade, rishabh.jogani}@iitgn.ac.in
*Indian Institute of Technology Gandhinagar, India

***Abstract*—Kubernetes provides strong self-healing for application workloads but remains limited in its ability to handle networking-related failures such as interface faults, network congestion, and DNS service disruptions. This project investigates the behavior of Kubernetes under such network conditions, focusing on the failure modes of CoreDNS and intra-node communication. Through controlled fault injection experiments, we characterize how the platform reacts to degraded network performance and identify gaps in its native recovery mechanisms. Building on these observations, a custom Kubernetes operator was developed to monitor DNS health and automatically remediate connectivity failures, extending the cluster's self-healing capabilities to the network infrastructure.**

I. Introduction

Kubernetes is the container orchestration platform used by most Cloud providers and Tech companies for efficient resource management. Kubernetes is used to manage containers across a group of computers. A 'Pod' is the smallest unit (can contain multiple containers) that can run and deploy. A pod in Kubernetes contains everything a application needs to run and hence can act as the smallest unit that can run or deploy. This functionality makes it easy to scale using pods and gives robust infrastructure.

Kubernetes provides support for cloud-based services that need to support a large number of containers. The Kubernetes has features that help **deploy, scale and maintain containers automatically [2]**. Kubernetes helps you rollout a ReplicaSet, rollback a previous Deployment, Scale up to facilitate more Load, see the status of the pods, and many more things automatically.

The general architecture of a simple Kubernetes Cluster is shown in Fig. 1. This particular Cluster has only one node. Each Cluster has a Control Plane that is in charge of maintaining the state of the cluster and contains the **controller manager, etcd, the scheduler and the kube-apiserver**. The control manager runs the controller processes, and each controller being responsible for certain features. For example, the Node Controller checks for if any nodes have gone down and responds to it. The etcd is a highly available key-value store used for backing up cluster data. This includes configuration data, state data, etc. This acts like a backup for the data such that even if a server/pod is shut down, the data is not lost. The kube-scheduler looks for newly created pods that have no assigned node, and selects the node for it to run on. It decides this by taking many factors into consideration like resource requirements, hardware/software constraints, workload constraints, etc. The kube-apiserver communicates with pods and is the front end of the Control Plane.

kubectl is the tool used by users to interact with Kubernetes Cluster through the Kubernetes API server. Kubelet and kube-proxy are two essential components of a worker node. Kubelet is the agent that runs in each node, and it makes sure containers in a pod are running and communicates with the API server. kube-proxy is the network proxy that runs in each node in a cluster. It allows network communication to your pods from inside and outside your cluster.

Kubernetes pods communicate with each other using a CNI Plugin or **built-in networking**. Each pod in a Kubernetes cluster is given a unique IP address and adheres to the necessary network policies to communicate with each other. Kubernetes provides automated self-healing for workloads but not for networking failures. This report outlines our project for testing the different networking-related failures and developing a custom operator to fix the issue of **CoreDNS availability**.

II. Literature Review

A. Kubernetes' Current Handling of Networking Failures

Kubernetes' built-in self-healing mechanisms primarily focus on the health of **workloads (pods)** and **nodes**, reacting to failures rather than proactively fixing the network infrastructure itself.

- **Pod-level Repairs:** If a container crashes or fails its liveness probes, the `kubelet` and higher-level controllers (such as ReplicaSet or Deployment) automatically restart the container or create a replacement pod.
- **Node Failures and Rescheduling:** If the control-plane stops receiving heartbeats from a node, the node is marked `NotReady`. After configured timeouts, the node controller evicts pods, and the scheduler reschedules them onto healthy nodes.
- **Network Components as Pods:** Core networking services like CoreDNS (DNS) usually run as Deployments and are re-created if a pod crashes. CNI agents (e.g., Calico, Cilium) run as DaemonSets and are re-created if they crash on a specific node.

1) The Gap: What Kubernetes Does Not Automatically Address: Kubernetes does not automatically diagnose or repair **low-level network faults** directly. Examples include a down `eth0` interface, incorrect host IRQ settings, or CNI internal configuration corruption.

Instead, Kubernetes generally waits until workloads (pods) or nodes show failure, and then reacts by restarting or rescheduling workloads. This reactive behavior means that many network problems remain unnoticed until they cause pod unreachability or degraded performance.

This behavior creates a critical gap: K8s effectively heals the **effects** (failed pods) but often does not detect or fix the **root network cause** early enough.

B. Current Approaches Extending Kubernetes Self-Healing to Networking Faults

A major theme in proposed solutions is that **early detection** of network faults (before pods crash) is crucial for avoiding long outages and reducing the Mean Time to Repair (MTTR). The literature explores multiple machine learning-based approaches—including monitoring, forecasting, anomaly detection, and learned command generation—to find faults earlier than Kubernetes' default behavior.

1) Time-Series Forecasting and Anomaly Scoring for Early Fault Detection: **Idea:** Utilizes time-series models and monitoring streams to flag network degradation and anomalies (including network metrics) before pods become unresponsive.

Example Implementation (Matsuo & Ikegami, CNSM 2021)[6]:

- Telemetry was collected via Prometheus while injecting 9 types of faults into a test Kubernetes deployment.
- Implemented and compared models like One-Class SVM (OCSVM), Autoencoders (AE), LSTM prediction, and LSTM Autoencoder (LSTM-AE) as anomaly detectors.
- K-means clustering was applied to the resulting anomaly-score time series ($k = 9$) to group fault patterns and map scores to fault types/severity.
- **Finding:** LSTM-style predictors produced anomaly scores that allowed **earlier detection** of degradation than simple threshold checks.

Benefit: Catching network degradation (rising latency, packet loss, queue backlogs) as an anomaly score allows triggering remediation *before* pods become unreachable.

2) Unsupervised Detection + Forecasting + Reinforcement Learning for Deciding Actions: **Idea:** Combines forecasting, unsupervised anomaly detection, and a learned policy to automatically choose the optimal remediation action.

Example Implementation (Laheri et al., JISEM, 2025)[8]:

- **Telemetry Forecasting:** Used an LSTM model to predict metric trends (CPU, memory, network throughput/latency).
- **Unsupervised Detection:** Employed Isolation Forest and k-Means clustering to detect unusual telemetry patterns.
- **Decision Agent:** A Reinforcement Learning (RL) agent using Proximal Policy Optimization (PPO) was trained to select remediation actions (e.g., restart, reschedule, scale, reconfigure). The reward function balanced reduced MTTR against avoiding disruptive actions.

Benefit: RL can learn which remediation works best for a specific pre-failure pattern (e.g., when small network deviations predict CNI instability, the RL policy might prefer restarting a CNI daemon over the more disruptive action of cordoning a node).

3) Learning Concrete Recovery Commands from Logs: **Idea:** Automate the repair step by generating specific host-level shell commands to fix the underlying network (e.g., restart a network service, change a `sysctl`, bring up an interface).

Example Implementation (Ikeuchi et al., NOMS)[7]:

- Used neural networks with an attention mechanism to map a sequence of log lines (input) to a sequence of recovery shell commands (output).
- **Preprocessing:** Logs were normalized to templates/tokens to ensure consistency and prevent variable fields from confusing the models.
- The system was trained and evaluated on synthetic logs and a real OpenStack dataset containing historical logs plus corresponding operator commands.
- **Result:** The model successfully proposed command sequences that matched past operator fixes.

Benefit: Instead of only rescheduling pods (the effect), a controller can run the exact host command to fix the underlying network root cause (e.g., restarting `irqbalance` or reloading a CNI component), removing a human manual step.

Evidently, most of the advanced approaches have been utilizing different **machine learning** techniques, specifically for prediction and anomaly detection, to both predict possible networking faults and select appropriate solutions.

III. Experimental Setup

Our experimental evaluation was conducted on a **Minikube** Kubernetes cluster running in a local **WSL** environment. The setup was designed to replicate common failure scenarios and illustrate the practical limits of Kubernetes self-healing mechanisms as encountered in real-world clusters. To start the Minikube cluster, we used `minikube start`. To start the in-built dashboard, we used `minikube dashboard`.

A. Deployment and Pod Configuration

To support robust self-healing and real-time network diagnostics, a custom Python Flask server application was containerized and deployed as the backend service. The application simulates realistic workload instability by randomly introducing response delays at the root endpoint (`/`), which in turn triggers Kubernetes **liveness probe failures**. A dedicated `/healthz` endpoint was implemented to accurately signal readiness status, allowing pods that become unresponsive to be **temporarily removed from Service endpoints** until recovery. This combination demonstrates both proactive (readiness) and reactive (liveness) healing mechanisms in Kubernetes.

The Docker image (`test-app:v2`) was built locally with the following specifications:

- **Base Image:** `python:3.10-slim`, selected for its lightweight footprint and compatibility with Flask and Debian-based network tools.

TABLE I
SUMMARY OF RESEARCH EXTENDING KUBERNETES SELF-HEALING TO NETWORKING FAULTS

Approach / Reference	Key Techniques and Findings
Time-Series Forecasting & Anomaly Scoring (Matsuo & Ikegami, CNSM 2021)	Prometheus telemetry with fault injection; OCSVM, Autoencoder, LSTM and LSTM-AE used for anomaly scoring. Anomaly scores signaled degradation earlier than fixed thresholds; K-means grouped score patterns by fault type.
Unsupervised Detection + RL-based Remediation (Laheri et al., JISEM 2025)	LSTM forecasting + Isolation Forest and k-means for unsupervised detection. PPO-based RL agent learned remediation policies (restart/reschedule/scale) with a reward balancing MTTR and disruption.
Learning Recovery Commands from Logs (Ikeuchi et al., NOMS)	Attention-based neural models converted templated log sequences to shell command sequences. After training on synthetic and real datasets, the model reproduced historical operator fixes for network faults.

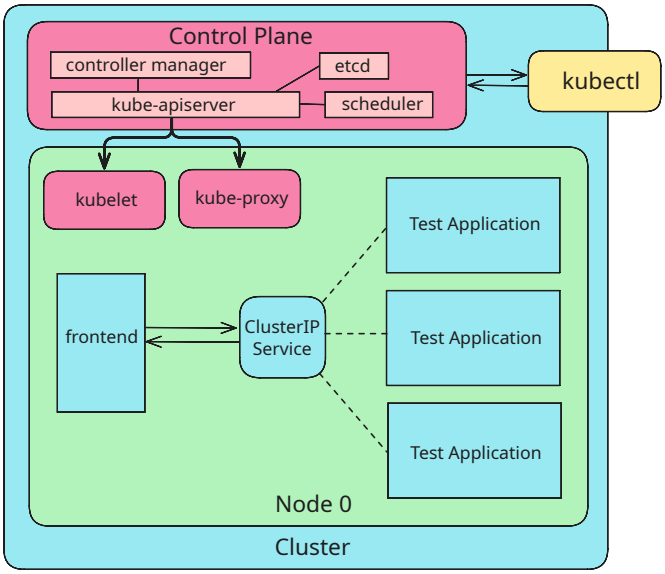

Fig. 1. Architecture Diagram of the Test Setup.

- **App Features:** The Flask server runs in multi-threaded mode to allow concurrent handling of health checks while simulating 10 second response delays on 20% of incoming requests.
- **Self-healing Logic:** A busy flag ensures the `/healthz` endpoint returns a temporary 503 status during simulated hangs, enabling rapid removal of the affected pod from Service routing.
- **Network Utilities:** Core network troubleshooting tools (`ping`, `dig`, `nslookup`, `curl`, `traceroute`, etc.) are preinstalled for in-cluster connectivity verification and debugging.

B. Image Management in Minikube

As Minikube operates in an isolated container runtime, the locally built image was imported into the cluster using `minikube image load test-app:v2`. The Deployment manifest explicitly specifies `imagePullPolicy: IfNotPresent` to ensure the locally loaded image is used rather than attempting to pull from an external registry. For version tracking and validation, each build tag (e.g., `v2`) corresponds to an incremental code revision.

C. Deployment Specification

The Kubernetes Deployment was defined to demonstrate both the built-in and custom self-healing capabilities of the platform:

- **Replica Management:** A fixed replica count of three pods helps us see load distribution and redundancy across simulated failures. This is done by a **Replica Set**.
- **Probes:**
  - A *readiness probe* targets `/healthz`, with a short interval and timeout to rapidly remove unresponsive pods from Service endpoints.
  - A *liveness probe* targets the root endpoint (`/`) to detect application hangs and trigger automated pod restarts after sustained probe failures.
- **Port Configuration:** Each pod exposes the Flask application on port 80, which is internally mapped to a **ClusterIP** Service for in-cluster access.
- **Demonstration Behavior:** During operation, random long-response events cause transient **readiness failures**, followed by **liveness-induced restarts**. This is performed by the **kubelet** of the node and clearly demonstrates the' self-healing loop of Kubernetes and minimal downtime during recovery.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: base-deployment
  labels:
    app: baseline-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: baseline-app
  template:
    metadata:
      labels:
        app: baseline-app
    spec:
```

```
    containers:
    - name: randomly-failing-app
      image: test-app:v2
      imagePullPolicy: IfNotPresent
      ports:
      - containerPort: 80
      livenessProbe:
        httpGet:
          path: /
          port: 80
        initialDelaySeconds: 10
        periodSeconds: 10
        timeoutSeconds: 2
        failureThreshold: 3
      readinessProbe:
        httpGet:
          path: /healthz
          port: 80
        initialDelaySeconds: 2
        periodSeconds: 2
        timeoutSeconds: 1
        failureThreshold: 1

apiVersion: v1
kind: Service
metadata:
  name: baseline-svc
  labels:
    app: baseline-app
spec:
  type: ClusterIP
  selector:
    app: baseline-app
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 80
```

The `failureThreshold` value specifies the required number of consecutive probe fails for us to consider the container as **not ready/unhealthy**. This YAML file was deployed using the `kubectl apply -f <filename>` command.

D. Frontend Deployment for Continuous Monitoring

To continuously monitor backend availability and capture the effects of self-healing events, a lightweight **frontend tester** pod was deployed. This pod uses the `curlimages/curl:7.88.1` image and runs an infinite shell loop that periodically performs HTTP requests to the backend service (`baseline-svc`). Each request result is timestamped and labeled as either `OK` (successful response) or `FAIL` (connection timeout or error).

- **Purpose:** The frontend acts as a synthetic client, generating steady traffic to detect transient outages and validate automatic recovery. Its output directly visualizes the impact of liveness probe restarts and readiness transitions.
- **Implementation:** A simple shell loop runs inside the container using the `curl` command.
- **Logging Behavior:** The frontend continuously emits timestamped logs such as:

  ```
  2025-11-09T14:33:16+00:00 OK
  2025-11-09T14:33:18+00:00 OK
  2025-11-09T14:33:20+00:00 FAIL
  ```

  These can be seen using `kubectl logs -l app=frontend -f`. Intermittent `FAIL` entries correspond to pods undergoing restarts or being temporarily marked as NotReady.
- **Visualization Role:** These logs, along with the built-in dashboard, serve as a real-time indicator of system health, making it possible to correlate backend probe events, readiness state transitions, and automatic recovery actions.

IV. Project Objectives

- Detect DNS and CoreDNS failures
- Probe pod-to-pod connectivity

V. Methodology

The first step in this project was to understand **how Kubernetes manages and restores network connectivity when failures occur**. Since the goal is to improve cluster resilience, it was important to study the existing behavior and limits of Kubernetes' built-in self-healing system.

Our approach combined small, controlled experiments with direct observation of cluster behavior. We first examined normal pod-to-pod communication to understand how the Linux bridge and CoreDNS support service discovery. Then, we introduced specific failures such as disconnecting the bridge interface, increasing network delay, and disabling DNS, to see how Kubernetes reacts. These tests helped identify which failures are automatically handled and which require manual recovery, forming the basis for designing better self-healing mechanisms in later stages.

A. Understanding Pod-to-Pod Connectivity

To analyze how pods within a single Kubernetes node communicate, Minikube was initiated with the `--cni=false` flag, which disables the deployment of a Container Network Interface (CNI) plugin such as Calico or Flannel. In this configuration, even though no overlay network is established, and the cluster does utilizes the typical CNI-managed pod network (e.g., the `10.244.0.0/16` range). Nevertheless, it was observed that pods located on the same node retained successful communication through direct ICMP exchanges, using `kubectl exec -it <pod1> -- ping -c 2 <pod2-ip>`, indicating that intra-node packet forwarding continued to function even without an active CNI.

To understand the underlying cause, the network configuration inside the Minikube virtual machine was examined using:

```
$ minikube ssh
```

```
$ ip addr
```

The output revealed several network interfaces, most notably `eth0`, `docker0`, and another bridge interface named `bridge`. The `eth0` interface represents the external link between the Minikube VM and the host system, while `docker0` is the default Docker bridge used for standalone containers. The presence of the additional `bridge` interface suggested the existence of an independent Linux bridge used internally by Kubernetes or the container runtime for pod networking.

To verify the role of these interfaces, each bridge was manually disabled using:

```
$ sudo ip link set docker0 down
$ sudo ip link set bridge down
```

Disabling the `docker0` interface had no effect on pod-to-pod communication. However, disabling the `bridge` interface immediately disrupted connectivity between pods on the same node, confirming that it was the active Layer-2 component facilitating local communication. Connectivity was subsequently restored by reactivating the bridge interface:

```
$ sudo ip link set bridge up
```

These observations align with the default networking behavior of Kubernetes when no CNI plugin is configured. In such cases, the `kubelet` and the underlying container runtime (Docker or containerd) establish connectivity by creating a Linux bridge and attaching each pod's virtual Ethernet (veth) pair to it. Each pod is allocated its own network namespace, within which a veth interface (typically named `eth0`) connects to a corresponding peer interface in the host's network namespace. The host-side endpoints of these veth pairs are then attached to the bridge interface, effectively placing all pods on the same node within a shared Layer-2 broadcast domain. This allows direct packet forwarding between pods through standard Linux kernel switching mechanisms, independent of any higher-level CNI abstraction or overlay.

In summary, this experiment confirmed that, even in the absence of a CNI plugin, intra-node pod communication in Kubernetes persists through a locally managed Linux bridge. This bridge-based mechanism provides fundamental Layer-2 connectivity within a node, while cross-node communication remains unavailable without a CNI-managed overlay network. The identification of this behavior informed subsequent experiments on simulating and self-healing local network failures by manipulating the bridge interface state.

B. Simulating and Analyzing DNS and Bridge Failures

The Kubernetes Domain Name System (DNS) service provides name resolution for all components within the cluster, allowing pods to refer to one another and to external services using fully qualified domain names (FQDNs). This functionality is implemented by the CoreDNS deployment, which operates as a cluster-internal service running within the `kube-system` namespace. Each pod communicates with CoreDNS through a ClusterIP service, reachable at `10.96.0.10`, with queries forwarded automatically via `/etc/resolv.conf` in the pod's network namespace.

To verify the operational state of DNS resolution, we used **dnsutils** pod to perform domain name lookups using the `dig` and `nslookup` utilities [1]:

Firstly, start an instance of `dnsutils` pod.

```
$ kubectl apply -f
https://k8s.io/examples/admin/dns/dnsutils.yaml
```

Then we can use it as such:

```
$ kubectl exec -it dnsutils -- dig example.com
```

and

```
$ kubectl exec -it dnsutils --
                nslookup kubernetes.default
```

Successful responses from this query indicated that the CoreDNS service was active and correctly resolving both cluster-local and external domains.

To simulate a DNS subsystem failure, the CoreDNS deployment was deliberately scaled down to zero replicas using:

```
$ kubectl scale deployment coredns --replicas=0
                                    -n kube-system
```

Following this operation, the same lookup command was reissued from within the application pod. The kube-dns Service (at 10.96.0.10) still exists, but no backing pods respond to queries; thus, they failed with timeouts, and no DNS records were returned.

Now, to evaluate how Kubernetes and CoreDNS handle degraded network conditions, we introduced artificial latency into the pod network bridge of our single-node Minikube cluster. Using the Linux traffic control (tc) utility, we simulated congestion by adding 200 ms of delay with a 50 ms normal distribution variance on the bridge interface:

```
$ sudo tc qdisc add dev bridge root netem
delay 200ms 50ms distribution normal
```

This setup emulated network jitter and congestion between all pods, including CoreDNS and application workloads. Immediately after applying the delay, DNS resolution within the cluster became inconsistent. Several queries from the dnsutils pod began to timeout or return SERVFAIL, while successful lookups exhibited significantly increased latency (often exceeding multiple seconds). The overall cluster responsiveness degraded, and some service connections stalled due to delayed DNS resolution.

Due to both of the above failures, the frontend started **failing** the curl operations, since it relies on the cluster's DNS to resolve `http://baseline-svc/`. After the test, the network state was restored by:

```
$ sudo tc qdisc del dev bridge root
```

Normal DNS resolution resumed immediately, confirming that the issue was entirely network-induced.

It was also observed that Kubernetes does not natively detect or self-heal the DNS layer when CoreDNS performance is degraded or when the pods become unreachable. Although the CoreDNS Deployment object remains present and healthy in the control plane, no corrective action is triggered during network congestion, since liveness and readiness probes continue to pass. Consequently, the cluster perceives the DNS

TABLE II
OBSERVED DNS LATENCY EFFECTS UNDER SIMULATED NETWORK CONGESTION

Metric	Observation
Average DNS query latency	3–5 ms under normal conditions; increased to 480–1100 ms after 200 ms delay injection.
95th percentile latency	≈10 ms normally, exceeding 2000 ms under congestion.
Query success rate	100% during baseline; dropped to roughly 60–70% with frequent timeouts.
CoreDNS pod status	Remained *Running*; no restarts or rescheduling observed.
Cluster reaction	No recovery or rescheduling actions triggered by Kubernetes.
System responsiveness	Normal operation before; noticeably sluggish during induced delay.
Recovery on cleanup	Immediate restoration of DNS resolution after removing delay rule.

service as operational even though name resolution is partially or completely impaired. The loss or degradation of DNS functionality therefore represents a single point of failure for intra-cluster and outbound service discovery.

These experiments highlight the central role of CoreDNS in Kubernetes networking and underscore the importance of extending monitoring and self-healing mechanisms to cover essential network-layer dependencies. The results provide a baseline for subsequent work on automated detection and recovery strategies targeting failures in the DNS resolution subsystem.

C. Summary of Observed Self-Healing Limits

The conducted experiments delineated the practical boundaries of Kubernetes' native self-healing capabilities, particularly for failures occurring within networking subsystems. While Kubernetes reliably restored failed workloads at the application layer, faults in the underlying network stack and service infrastructure remained largely invisible to its default recovery mechanisms.

- **Pod-level resilience:** Kubernetes effectively reinstated terminated application pods through ReplicaSet and Deployment reconciliation. These mechanisms provided strong fault tolerance for container-level failures, ensuring service continuity under normal operational conditions.
- **Intra-node networking behavior:** When the Container Network Interface (CNI) was disabled (`--cni=false`), local pod-to-pod communication persisted via the Linux `bridge` interface and virtual Ethernet (veth) pairs. However, deliberate disruption of the bridge interface caused complete loss of intra-node connectivity without any automated restoration. This confirmed that host-level networking failures remain outside Kubernetes' self-healing scope.
- **Cluster DNS dependency:** The CoreDNS deployment was identified as a single point of dependency for intra-cluster name resolution. Both scaling down and network degradation experiments revealed that Kubernetes did not detect or remediate DNS unavailability. Even under severe network latency and packet loss, CoreDNS pods remained marked as healthy, and no recovery was triggered, demonstrating a lack of awareness for network-path degradation.
- **Operational diagnostics:** Standard application containers lack essential network troubleshooting utilities (e.g., `dig`, `ping`, `curl`). The use of containers such as `dnsutils` and `network-multitool` was therefore critical for observation and verification of fault conditions.

These findings exposed clear limitations in Kubernetes' default fault management model. In particular, the **absence of automated recovery for disruptions in the node-level bridge interface** and the **CoreDNS service** defined two critical weak points in cluster resilience. Consequently, the subsequent phase of this work focuses on developing targeted self-healing extensions capable of detecting and restoring failures in both the intra-node network fabric and the DNS resolution subsystem, thereby extending Kubernetes' self-healing into the networking domain.

VI. SELF-HEALING ENHANCEMENT PIPELINE

Building on the above findings, we designed and implemented a focused enhancement to Kubernetes' self-healing specifically for the DNS layer (CoreDNS), which can be found here. This section summarizes the end-to-end pipeline we followed – from design choices to validation – and documents the steps and commands we used to set up, test, and evaluate the system.

A. Design Objectives and Scope

We framed the enhancement around four concrete objectives:

- **Detect DNS impairments** that do not crash pods (e.g., timeouts, unreachability, or replica count reduced to zero).
- **Verify end-to-end resolution**, not just pod liveness, using an in-cluster probe.
- **Remediate safely and deterministically** (scale CoreDNS back to a baseline and restart unhealthy pods when thresholds are exceeded).
- **Expose clear state** (last action, health, failure counts) for operators to audit.

B. Operator and CRD Overview

We implemented a custom controller in **Go** using the **Kubebuilder** toolkit. The CRD `DNSMonitor` captured the intent of monitoring and remediation. Its spec included: the target namespace (defaulted to `kube-system`), the probe interval in seconds, a test domain (defaulted to `kubernetes.default.svc.cluster.local`), a failure threshold (number of consecutive probe failures before remediation), and an optional minimum desired replica count for CoreDNS. The status tracked recent health and actions

through fields for last check time, last action string, a failure counter, and a healthy flag.

Kubebuilder scaffolds operator projects on **top of the controller-runtime stack**, providing a Manager, shared cache/informers, a typed Kubernetes client, and a Reconciler loop that reacts to object changes. In our project, the API types for `DNSMonitor` are defined under `api/v1alpha1` with Kubebuilder markers to generate CRD manifests and RBAC from code annotations, while the controller logic resides in `controllers/dnsmonitor_controller.go`. Deployable manifests are assembled via Kustomize under `config/`.

C. Controller Workflow We Implemented

As shown in Fig. 2, during each reconciliation, the controller performed the following steps:

1) **Enforced baseline availability**: we read the CoreDNS Deployment and increased replicas if they fell below the configured minimum (e.g., when someone scaled CoreDNS to zero).
2) **Evaluated pod readiness**: we listed CoreDNS pods using their standard label and removed those that were unready to nudge Kubernetes to recreate them.
3) **Probed DNS end-to-end**: we ran a short `Job` that executed a simple resolution (`nslookup <testDomain>`) from inside the cluster. We created at most one probe job at a time and relied on job status to decide success or failure.
4) **Applied remediation on sustained failures**: if the number of consecutive failed probes reached the threshold, we restarted CoreDNS by deleting existing pods (allowing the Deployment to recreate them).
5) **Recorded state and re-queued**: we updated the CR status (last check, last action, failure counter, health) and re-queued after the chosen probe interval to maintain steady monitoring.

D. Environment Preparation and Build Pipeline

We worked on a local Minikube cluster. We followed the sequence below to build and deploy the operator and its CRD.

```
# Start cluster
$ minikube start

# Optional for dashboard metrics
$ minikube addons enable metrics-server

# Build operator artifacts
$ cd coredns-operator
$ make generate
$ make manifests

# Build and load controller image
$ export IMG="shardul0109/coredns-operator:v1"
$ make docker-build IMG=${IMG}
$ minikube image load ${IMG}

# Install CRDs and deploy controller
$ make install
```

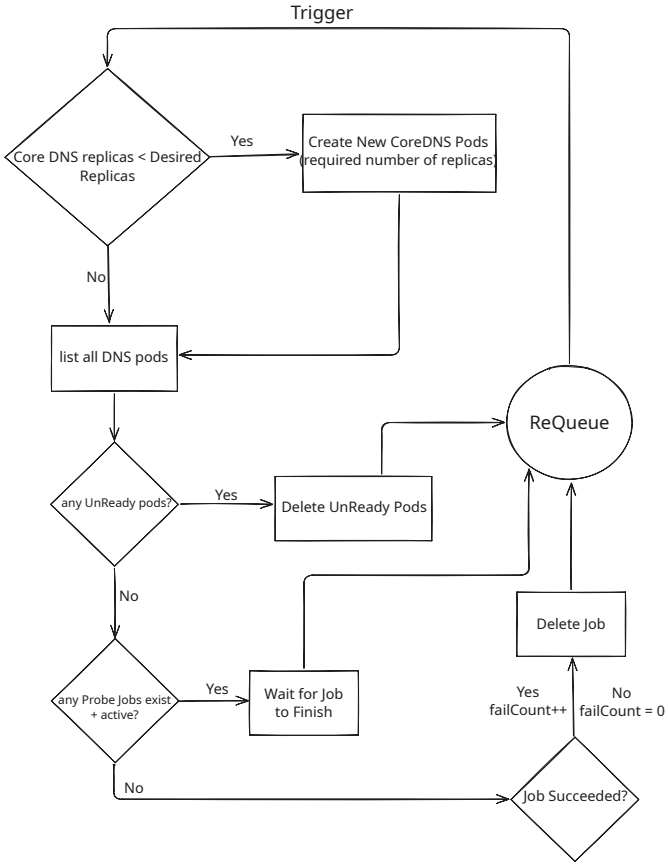

Fig. 2. Operator Workflow

```
$ make deploy IMG=${IMG}

# Verify controller is running
$ kubectl get pods -n coredns-operator-system
```

E. Creating and Tuning the Monitor

We applied a sample `DNSMonitor` for guardrails. We configured the namespace as `kube-system`, the probe interval to 30 seconds, the failure threshold to 3, the target domain as the internal `kubernetes.default.svc.cluster.local`, and a minimum CoreDNS replica count of 2.

Example manifest (`infra_v1alpha1_dnsmonitor.yaml`):

```
kind: DNSMonitor
metadata:
  name: dns-monitor
  namespace: default
spec:
  namespace: kube-system
  probeIntervalSeconds: 30
  testDomain: "kubernetes.default.svc.cluster.local"
  failureThreshold: 3
  desiredReplicas: 2
```

Apply the sample DNSMonitor and verify:

```
$ kubectl apply -f coredns-operator/config/
    samples/infra_v1alpha1_dnsmonitor.yaml
$ kubectl get dnsmonitors -A
```

```
$ kubectl describe dnsmonitor dns-monitor
```

F. Failure Scenarios We Executed

We validated detection and remediation using two representative scenarios:

- **Scaled CoreDNS to zero**: we reduced the `coredns` Deployment replicas to 0 and confirmed that DNS queries from a throwaway pod failed. The operator then enforced the configured replica floor and brought the service back.
- **Unready or stuck pods**: we manually disrupted CoreDNS pods, observed readiness failures, and confirmed the controller reclaimed them by deletion, allowing fresh pods to come up cleanly.

We can test DNS resolution inside the cluster using the following command:

```
$ kubectl exec -it dnsutils \
    -- nslookup kubernetes.default
```

Next, to induce a failure by scaling CoreDNS to zero replicas, we can use:

```
$ kubectl scale deployment \
    coredns -n kube-system --replicas=0
```

Now, wait for few seconds for replicas to be restored by the operator. We can verify the status of CoreDNS pods again using the earlier command.

We can observe the operator's actions and CoreDNS pod status using:

```
$ kubectl get pods -n kube-system \
    -l k8s-app=kube-dns -w
$ kubectl logs -n coredns-operator-system \
    deploy/coredns-operator-controller-manager -f
```

VII. Conclusion and Future Scope

This study examined the boundaries of Kubernetes' native self-healing capabilities with a focus on networking-related failures. Through controlled experiments involving bridge disruptions, induced network latency, and CoreDNS degradation, we demonstrated that while Kubernetes reliably restores failed workloads, it lacks mechanisms to detect or recover from underlying network path failures. The platform reacts only after workloads become unavailable, leaving components such as CoreDNS and node-level networking as critical single points of failure.

To address this gap, a **custom self-healing operator was developed using kubebuilder** to continuously monitor DNS health and automatically remediate failures in CoreDNS connectivity. The results confirmed that extending Kubernetes' self-healing logic into the networking domain significantly improves fault resilience and reduces downtime. Future work will explore integrating predictive monitoring and lightweight anomaly detection to enable proactive healing before service impact occurs.

References

[1] Groundcover. (2023) Kubernetes dns issues: Troubleshooting guide. [Online]. Available: https://www.groundcover.com/kubernetes-troubleshooting/dns-issues

[2] Kubernetes Documentation. (2024) Self-healing in kubernetes. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/self-healing/

[3] ——. (2024) Cluster networking. [Online]. Available: https://kubernetes.io/docs/concepts/cluster-administration/networking/

[4] Red Hat Developers. (2021) Build a kubernetes operator in six steps. [Online]. Available: https://developers.redhat.com/articles/2021/09/07/build-kubernetes-operator-six-steps#setup_and_prerequisites

[5] M. Shamim *et al.*, "Multi-vocal literature review of kubernetes," *arXiv preprint arXiv:2211.07032*, 2022.

[6] M. Matsuo and Y. Ikegami, "Performance analysis of anomaly detection methods for application systems on kubernetes with auto-scaling and self-healing," in *International Conference on Network and Service Management (CNSM)*, 2021.

[7] Y. Ikeuchi *et al.*, "Recovery command generation towards automatic recovery from failures beneath kubernetes self-healing," nOMS technical report.

[8] Laheri *et al.*, "Self-healing infrastructure: Leveraging reinforcement learning for autonomous cloud recovery and enhanced resilience," *JISEM*, 2025.

[9] CloudKitchens Engineering, "From fragile to faultless: Kubernetes self-healing in practice," Engineering blog.