



Harvester Network Monitoring with DeepFlow

Introduction

Harvester is a hyperconverged infrastructure solution built on Kubernetes. To gain deep network observability in a Harvester cluster (e.g., monitor VM and Pod network flows), we propose using **DeepFlow** – an open-source, zero-intrusion observability platform leveraging eBPF. DeepFlow can collect network telemetry from all workloads without code changes. This document (Option C) details the DeepFlow-based monitoring solution, its justifications, and how to deploy it on a 3-node Harvester cluster. We also cover security measures (RBAC, Pod Security Admission, NetworkPolicies) and provide a Helm chart ("Harvester Network Monitoring") for automated deployment.

Solution Overview

DeepFlow consists of two main components: **Agents** and a **Server**. A DeepFlow Agent runs as a DaemonSet on each node (including Harvester host nodes) to capture traffic via eBPF, while the DeepFlow Server runs in the cluster to aggregate data, inject Kubernetes metadata tags, store observability data, and provide query interfaces. The Server relies on two stateful storage backends, **MySQL** and **ClickHouse**, for metadata and metrics respectively. DeepFlow's Helm chart also deploys a pre-configured **Grafana** instance for dashboards, so users can visualize network flows and performance metrics immediately after installation.

Harvester's networking uses Flannel (an overlay) for the management network by default, with optional VLAN bridges for VM traffic. DeepFlow's eBPF-based Agents capture all network packets on the host, including Pod overlay traffic and VM bridged traffic, giving full visibility into inter-VM and inter-Pod communications. The Agents send processed flow logs and metrics to the DeepFlow Server, which enriches them with cluster context (e.g. Pod names, VM names) using Kubernetes API data. The result is a “universal service map” and detailed metrics/traces for every connection [1](#), accessible via Grafana dashboards or APIs.

Justification for DeepFlow (Option C)

Comprehensive Network Observability: DeepFlow provides deep visibility into every application call (“Flow”) with zero code changes [2](#). It not only captures network flow logs, but also auto-calculates golden signals (RED metrics) for all services, distributed tracing of requests, and even continuous CPU profiling – all correlated by eBPF on the hosts. This level of full-stack observability (from network to application) exceeds what basic monitoring tools provide, ensuring any performance bottleneck or anomaly in Harvester’s VM networking can be traced and diagnosed.

Zero Intrusion & Low Overhead: DeepFlow’s use of eBPF means no agents inside VMs or Pods are needed – the kernel-level probes collect data without interfering with workloads [3](#). This “zero-instrumentation” approach avoids altering or slowing down application code. The DeepFlow Agent is implemented in high-

performance Rust, and the continuous profiling feature incurs **<1% CPU overhead**. Thus, our monitoring will not noticeably degrade the performance of the Harvester cluster or its VMs.

Built for Kubernetes: DeepFlow was designed for cloud-native environments. It automatically tags data with K8s metadata (namespaces, labels, etc.), which is crucial in Harvester where each VM is backed by a Kubernetes pod. It stores all telemetry in ClickHouse, a columnar database optimized for querying time-series and logs. The use of ClickHouse with **SmartEncoding** yields ~10x storage efficiency over vanilla approaches, mitigating storage costs for high-volume flow logs. DeepFlow also exposes a **PromQL API and SQL API** for integration, and the included Grafana provides out-of-the-box dashboards for network flows, service maps, and tracing. This tight integration with K8s and Grafana means faster setup and more insightful dashboards than building a custom solution from scratch.

Future Extensibility (Splunk Integration): DeepFlow supports exporting data via standard protocols, which can facilitate future integration with Splunk or other external analytics platforms. For example, DeepFlow can act as a Prometheus data source or export data via OpenTelemetry and Kafka. In the future, flow logs or metrics could be forwarded to Splunk by using a Kafka exporter and then ingesting into Splunk, or by using OpenTelemetry to send traces/metrics to a Splunk Observability endpoint. This flexibility means our monitoring stack can evolve without replacing core components.

Mitigations for Potential Challenges: Every powerful observability tool comes with considerations. We address them as follows:

- *Resource Usage:* Running eBPF collectors and ClickHouse on the cluster will consume CPU, memory, and disk. **Mitigation:** We allocate dedicated resource requests/limits for DeepFlow pods to avoid resource contention. The DeepFlow Server and databases can be right-sized for the 3-node cluster (e.g., using small resource limits initially, and scaling up if needed). eBPF collection has minimal overhead by design, and we can disable intensive features (like full tracing or profiling) if not needed.
- *Data Volume:* Detailed flow logs and traces can generate large data volumes. **Mitigation:** We will use persistent volumes for ClickHouse/MySQL (backed by Harvester's storage) to retain data safely. Data retention periods can be tuned in DeepFlow's config (e.g., keeping only recent 7 days). DeepFlow's SmartEncoding compresses tag data to reduce storage use. In production, one could also deploy external DB servers or scale-out ClickHouse if needed.
- *Security Risks:* DeepFlow Agents require privileged access to the node (for eBPF), which is a potential security concern. **Mitigation:** We restrict DeepFlow's privileges via Kubernetes security controls. We create a dedicated namespace with **Pod Security Admission** labels to allow the necessary host access only there, while keeping other namespaces at stricter levels. We also apply **RBAC** rules so DeepFlow's service accounts can only read the minimal Kubernetes API info needed (pods, nodes, etc.), enforcing the principle of least privilege. Additionally, **NetworkPolicies** will be implemented to limit DeepFlow pods' network access (e.g., Agents send data only to the Server; external access only to Grafana UI), reducing the risk radius if a component is compromised. These measures ensure the monitoring stack is isolated and controlled.

In summary, DeepFlow addresses our requirements for a robust network monitoring solution in Harvester, offering unparalleled visibility (justification) while the outlined mitigations handle performance and security concerns.

Deployment Design and Components

We deploy DeepFlow on the Harvester cluster using a Helm chart for reproducibility. The deployment will create the following components inside a Kubernetes namespace (e.g., `deepflow`):

- **DeepFlow Agent (DaemonSet)**: runs on each of the 3 nodes to capture traffic. It runs with `privileged: true` and mounts necessary host resources (e.g., BPF filesystem) for eBPF. Each Agent connects to the DeepFlow Server to stream observability data.
- **DeepFlow Server (Deployment/StatefulSet)**: one or more replicas (we will use 3 for high availability across nodes). It handles incoming data from Agents, writes to databases, and serves queries/dashboards. We configure `replicas=3` (one per node) via Helm values for redundancy. The Server pods will use a Service for internal communication.
- **MySQL (StatefulSet)**: stores metadata such as cluster resource info and DeepFlow configuration. We deploy MySQL 8 in-cluster. A PersistentVolumeClaim (PVC) is used for durability of the MySQL `deepflow` database. In a production setup, an external managed MySQL could be used, but here we use an in-cluster instance for simplicity.
- **ClickHouse (StatefulSet)**: the high-performance database storing all collected telemetry (flow logs, metrics, traces). We use a StatefulSet with volume claims for data. With 3 replicas (one on each node), DeepFlow will distribute data to the local ClickHouse for efficiency. Each ClickHouse pod holds a shard of the data; the DeepFlow server coordinates queries across them. This design prevents data loss if any one node goes down and improves read/write throughput.
- **Grafana (Deployment)**: an instance of Grafana pre-loaded with DeepFlow dashboards. The Helm chart sets a default admin password (`admin:deepflow`) and exposes Grafana via a NodePort service, so we can access it using any node's IP. Grafana connects to the DeepFlow Server's PromQL/SQL API as a data source, enabling interactive exploration of metrics and traces. This provides a convenient UI to view network performance and topology in the Harvester cluster.

All components run **within the Harvester cluster** (on the Kubernetes control plane provided by Harvester). Interaction with Rancher Manager is not required for deployment – we can deploy via kubectl/Helm on Harvester's K8s API or Harvester's UI. (Later, Rancher could be used to manage this as an extension, but it's not necessary for functionality.)

Storage: We ensure Persistent Volumes are used for stateful components. Harvester includes an integrated distributed storage (Longhorn) for VM volumes; we leverage the default StorageClass (e.g., Longhorn) for DeepFlow's PVCs. This avoids using hostPath volumes that could lead to data loss if pods reschedule. In case no default storage is configured, we can manually specify one (the Helm chart supports `global.storageClass` override). Using reliable PVs means MySQL and ClickHouse data will persist across pod restarts and stay on their respective nodes unless intentionally moved.

Networking: The DeepFlow pods communicate internally: Agents -> Server (data ingestion), Server -> Databases (storage), Grafana -> Server (queries). No internet access is needed (images can be preloaded or pulled from a registry accessible to Harvester). We will define Kubernetes Services for discovery (e.g., a

headless Service for ClickHouse cluster, a ClusterIP for MySQL). Grafana's NodePort is the only externally exposed interface, which we can access for dashboards. Harvester's default Flannel CNI supports NetworkPolicies, which we use to lock down traffic, as described next.

Security: RBAC, Pod Security & Network Policies

Security is a key aspect of this deployment. We implement multiple layers of defense:

- **RBAC:** We create minimal Roles/ClusterRoles for DeepFlow. The DeepFlow Server needs to list/watch Kubernetes resources to enrich tags (pods, namespaces, services, etc.). We grant it read-only access (verbs: get, list, watch) on necessary API groups. For example, a ClusterRole `deepflow-server` might allow reading core API resources like Pods, Nodes, Namespaces, Services. We bind this ClusterRole to the Server's service account. The Agent likely doesn't need direct Kubernetes API access (it reports to the server), so it can run under a separate service account with no extra privileges. By **enforcing least privilege**, we reduce risk – even if a DeepFlow pod is compromised, it cannot make unauthorized changes to the cluster.
- **Pod Security Admission (PSA):** We label the `deepflow` namespace with an appropriate Pod Security level. Most DeepFlow pods can comply with the **baseline** policy (which disallows privilege escalation), but the Agent requires privileges that baseline would block (privileged container, hostNetwork access, etc.). To accommodate this, we can do one of two things:
 - **Namespace-label override:** Label the namespace to use the `privileged` PSA profile (since these are trusted infrastructure pods). This effectively allows the Agent pods to run with elevated privileges while keeping other namespaces at baseline/restricted. We ensure only authorized users can deploy to the `deepflow` namespace.
 - **PSA exemptions:** Some clusters support exempting specific service accounts or namespaces from PodSecurity controls. We could exempt the DeepFlow Agent's service account so that other pods in the namespace still meet baseline.

In our deployment, for simplicity, we treat the whole namespace as privileged (not unrestricted globally, but within this namespace we permit host access). This is acceptable as the DeepFlow components are equivalent to a monitoring daemon – similar in trust level to metrics-server or CNI pods which often run with elevated privileges. We document this clearly. All other DeepFlow pods (Server, DB, Grafana) do **not** run as privileged and we can enforce baseline restrictions on them (no host namespace usage, no privilege escalation) via their securityContext settings.

- **Network Policies:** We craft Kubernetes NetworkPolicy objects to restrict traffic among the DeepFlow pods and between DeepFlow and the outside world. By default, Kubernetes pods can communicate freely; applying NetworkPolicies allows us to specify permitted flows. Our approach:
- **DeepFlow Server Policy:** Only allow ingress to the Server from the Agents (on the data ingestion port) and from Grafana (on the query API port). This ensures no other pod can send data to the server or query it. We also allow the Server to communicate with ClickHouse and MySQL (egress rules or by those pods' ingress rules).
- **ClickHouse Policy:** Only allow ingress to ClickHouse pods from the DeepFlow Server pods (on the ClickHouse TCP port). No other workloads should talk to the DB directly.

- **MySQL Policy:** Only allow ingress from the DeepFlow Server (for writing metadata) and possibly Grafana (Grafana might use MySQL as its backend store for dashboards if configured). If Grafana is configured to use MySQL, we include its pod in the allowed sources; if Grafana uses internal DB, then only Server needs DB access.
- **Grafana Policy:** Allow Grafana to communicate to DeepFlow Server (for data queries) and (if needed) MySQL. For Grafana's own ingress from user's browser, since it's exposed via NodePort, the traffic enters from node IP (treated as outside cluster). By default, NetworkPolicy **defaults to allow external traffic** unless we explicitly restrict it. We will not block NodePort access so that users can reach Grafana UI. Grafana's egress is limited to the cluster (no need for internet).

Additionally, Agents only need to send to Server – we can restrict Agent egress to the Server's service IP/port if desired (ingress rules on Server already handle it). By deploying these policies, even if other workloads run on Harvester, they cannot access DeepFlow's databases or APIs, and DeepFlow components themselves can only talk on the intended paths. This containment significantly reduces risk from lateral movement or data exfiltration.

Together, RBAC + PSA + NetworkPolicies create a robust security posture for the monitoring stack. DeepFlow gains the access it needs (kernel and API visibility) but is tightly confined in every other sense.

Installation as a Helm Chart

We provide a Helm chart, **Harvester Network Monitoring**, to deploy the entire DeepFlow stack with one command. This chart encapsulates the official DeepFlow Helm (community edition) plus our security customizations. It can be installed directly on Harvester's Kubernetes cluster. Eventually, this chart can be packaged as a Harvester UI **extension** (so it could be enabled via Rancher's UI with one click). Harvester supports adding extensions by pointing to a Helm repository, so we could publish this chart in a Git repository and register it in Rancher later for easy management.

For now, we assume helm CLI or Harvester API is used:

Prerequisites: Ensure the cluster has at least 3 nodes (as specified) and a default `StorageClass` (e.g., Longhorn) for PVCs. Also, verify kernel version ≥ 4.14 on nodes for eBPF support (Harvester nodes typically use modern kernels, so this should be fine).

Install Steps: 1. **Add Helm Repo (Optional):** If we publish this chart in a repo, the user would run `helm repo add ...` etc. (For development, one can clone the GitLab project.) 2. **Install Chart:** `helm install harvester-netmon ./harvester-network-monitoring -n deepflow --create-namespace`

This creates the `deepflow` namespace and all resources. The chart will automatically label the namespace for PSA and set up RBAC, etc. 3. **Post-Install:** The chart's post-install notes will echo how to retrieve the Grafana URL and login. For example, after install, run:

```
bash
export NODE=$(kubectl get nodes -o
jsonpath=".items[0].status.addresses[0].address")
export PORT=$(kubectl get svc deepflow-grafana -n deepflow -o
jsonpath=".spec.ports[0].nodePort")
echo "Grafana URL: http://$NODE:$PORT, login: admin:deepflow".
```

Then you can open Grafana in a browser and start exploring dashboards (DeepFlow provides dashboards for "Network - Flow Log", "Network - Topology", "Application - Service Map", etc., covering VM and container traffic).

Within a few minutes of deployment, DeepFlow Agents will begin capturing network data on all nodes. The Server will populate the ClickHouse DB with flow records and metrics. Grafana's dashboards should start showing information: e.g., inter-VM traffic volumes, TCP connection metrics (latency, retransmits), and more. If no data appears, the FAQ suggests checking pod status and agent connections. Common issues might include kernel not supporting required eBPF features (rare on new OS) or the need to adjust capture interfaces if using a custom CNI (DeepFlow's `tap_interface_regex` setting) – but on Harvester's default flannel and Linux bridge, the default should work.

The final section provides the structure and content of the Helm chart (as would be in the GitLab project), including all templates and configurations for this deployment.

Helm Chart: Harvester Network Monitoring (GitLab Project)

Below is the fully coded Helm chart to deploy DeepFlow and Grafana on Harvester. The chart is organized as follows:

```
harvester-network-monitoring/
├── Chart.yaml
├── values.yaml
└── templates/
    ├── namespace.yaml
    ├── deepflow-agent-daemonset.yaml
    ├── deepflow-server-statefulset.yaml
    ├── deepflow-clickhouse-statefulset.yaml
    ├── deepflow-mysql-statefulset.yaml
    ├── deepflow-grafana-deployment.yaml
    ├── service-deepflow.yaml
    ├── service-grafana.yaml
    ├── service-mysql.yaml
    ├── service-clickhouse.yaml
    ├── networkpolicy-server.yaml
    ├── networkpolicy-databases.yaml
    └── rbac.yaml
```

Chart.yaml: Metadata for the Helm chart. We set the API version, name, version, etc., and list any dependencies (if we were to include the official deepflow chart as a dependency, but here we inline the manifests for full control).

```
# Chart.yaml
apiVersion: v2
```

```

name: harvester-network-monitoring
description: "Deploy DeepFlow (network observability) on Harvester with Grafana"
type: application
version: 0.1.0
appVersion: "7.1.0" # DeepFlow app version (community edition v7.1)

```

values.yaml: Default configuration values. This allows customization of image versions, resource sizes, storage class, etc., without modifying templates. For brevity, key fields are shown:

```

# values.yaml
global:
  imageRegistry: "" # e.g., use local registry mirror if offline
  storageClass: "" # default storage class for PVCs (leave blank to use
cluster default)
  replicas: 3       # number of DeepFlow server and DB replicas (3 for HA on 3
nodes)
deepflow:
  version: "7.1.0"
  image: "deepflowio/deepflow-server:7.1.0" # image for server
  agentImage: "deepflowio/deepflow-agent:7.1.0"
  # Ports (from DeepFlow docs: server default ports)
  ports:
    data: 20417      # ingestion from agents
    api: 30880       # query API (PromQL/SQL)
    otel: 38086      # (optional) OTLP ingest, not used initially
mysql:
  image: "mysql:8.0"
  dbName: "deepflow"
  user: "root"
  password: "deepflow" # example; in practice use a k8s Secret for prod
clickhouse:
  image: "yandex/clickhouse-server:22.3" # a recent ClickHouse version
  # Using default built-in default user for ClickHouse (no auth for simplicity;
internal network only)
grafana:
  image: "grafana/grafana:9.5.2" # Grafana version
  adminUser: "admin"
  adminPassword: "deepflow"
  serviceType: "NodePort"
  nodePort: 31999          # fixed NodePort for Grafana access (from
DeepFlow docs example)
resources:
  server:
    limits: { cpu: "1", memory: "2Gi" }
    requests: { cpu: "0.5", memory: "1Gi" }
agent:

```

```

limits: { cpu: "0.5", memory: "512Mi" }
requests: { cpu: "0.2", memory: "256Mi" }
clickhouse:
  requests: { storage: "10Gi" }
mysql:
  requests: { storage: "1Gi" }

```

The above values can be adjusted as needed (for example, increase storage if you plan to retain data longer, or adjust NodePort).

Namespace Template: We explicitly create the namespace with PSA labels. This ensures Pod Security Admission is configured as we want (privileged allowed here). We mark it privileged and also add an **owner label** for our chart.

```

# templates/namespace.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: {{ .Release.Namespace }}
  labels:
    pod-security.kubernetes.io/enforce: "privileged"  # allow DeepFlow agents
    to run privileged
    pod-security.kubernetes.io/enforce-version: "latest"
    app.kubernetes.io/name: harvester-network-monitoring
    helm.sh/chart: "{{ .Chart.Name }}-{{ .Chart.Version }}"
    app.kubernetes.io/instance: "{{ .Release.Name }}"

```

RBAC Template: Defines service accounts and roles. We create two service accounts: one for the server (`deepflow-server-sa`) and one for the agent (`deepflow-agent-sa`). The server SA is bound to a ClusterRole with read-only privileges on necessary resources.

```

# templates/rbac.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: deepflow-server-sa
  namespace: {{ .Release.Namespace }}
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: deepflow-agent-sa
  namespace: {{ .Release.Namespace }}
---
# ClusterRole for DeepFlow server to read Kubernetes resource metadata

```

```

kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: deepflow-server-role
rules:
- apiGroups: [""]
  resources: ["pods", "namespaces", "nodes", "services", "endpoints"]
  verbs: ["get", "list", "watch"]
- apiGroups: ["apps"]
  resources: ["deployments", "daemonsets", "statefulsets", "replicasets"]
  verbs: ["get", "list", "watch"]
# (Include other resources if needed for tagging, e.g., if DeepFlow tags PVs or
CRDs like KubeVirt VMs, add those)
---
# Bind the role to the server service account
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: deepflow-server-rb
subjects:
- kind: ServiceAccount
  name: deepflow-server-sa
  namespace: {{ .Release.Namespace }}
roleRef:
  kind: ClusterRole
  name: deepflow-server-role
  apiGroup: rbac.authorization.k8s.io

```

We do **not** give cluster-admin or write permissions. The agent SA is not given any Role/ClusterRole (no binding), since it doesn't need direct API access. It will run privileged but isolated.

DeepFlow Agent DaemonSet: This ensures an agent on each node. Key points: - Runs with `privileged: true` (necessary for eBPF and capturing all traffic). - Mounts `/sys` or BPF file system as needed (DeepFlow might require `/sys/kernel/debug` or BPF FS). - Uses `deepflow-agent-sa` service account. - Toleration for all nodes (including masters) if needed, since even control-plane node might have VM pods (in Harvester all nodes run VMs). - Environment variables or args to point it to the server service address.

```

# templates/deepflow-agent-daemonset.yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: deepflow-agent
  labels:
    app: deepflow-agent
    app.kubernetes.io/component: agent
    app.kubernetes.io/instance: {{ .Release.Name }}

```

```

spec:
  selector:
    matchLabels: { app: deepflow-agent }
  template:
    metadata:
      labels: { app: deepflow-agent }
    spec:
      serviceAccountName: deepflow-agent-sa
      hostNetwork: true          # to capture host network traffic (and
possibly needed for eBPF)
      hostPID: true              # allow access to host PID if needed for maps
      dnsPolicy: ClusterFirstWithHostNet
      securityContext:
        runAsUser: 0            # run as root
        runAsGroup: 0
      containers:
        - name: agent
          image: {{ .Values.deepflow.agentImage }}
          imagePullPolicy: IfNotPresent
          args: [--collector.server={{ .Release.Name }}-server:
{{ .Values.deepflow.ports.data }}]
            # The above argument points agent to the deepflow-server service
(discovery by DNS)
          securityContext:
            privileged: true      # allow eBPF and packet capture privileges
            capabilities:
              add: ["SYS_ADMIN", "SYS_PTRACE", "NET_ADMIN"] # eBPF usage may
require these (depends on implementation)
            resources: {{ toYaml .Values.resources.agent | nindent(10) }}
          volumeMounts:
            - name: sysfs
              mountPath: /host/sys
              readOnly: true
            - name: bpffs
              mountPath: /sys/fs/bpf
          volumes:
            - name: sysfs
              hostPath:
                path: /sys
            - name: bpffs
              hostPath:
                path: /sys/fs/bpf
                type: DirectoryOrCreate
          tolerations:
            - operator: Exists # tolerate all taints (so it runs on control-plane
nodes too)

```

This DaemonSet ensures each node's traffic is monitored. We mount host `/sys` (read-only) and BPF filesystem for eBPF maps. The agent connects to `deepflow-server` on the data port (20417 by default).

DeepFlow Server StatefulSet: We use a StatefulSet to allow stable identity (useful if we later attach volume or if we want stable network ID). With 3 replicas, each server pod gets an ordinal (0,1,2). We also set affinity to spread them across nodes (e.g., using topologyKey: kubernetes.io/hostname anti-affinity to avoid two on same node). Each server connects to local ClickHouse instance ideally – in our case, we deploy ClickHouse as a separate StatefulSet, but we ensure scheduling aligns by hostname.

```
# templates/deepflow-server-statefulset.yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: {{ .Release.Name }}-server
  labels: { app: deepflow-server }
spec:
  replicas: {{ .Values.global.replicas }}
  selector:
    matchLabels: { app: deepflow-server }
  serviceName: {{ .Release.Name }}-server-headless # headless service for
stable network IDs if needed
  template:
    metadata:
      labels: { app: deepflow-server }
    spec:
      serviceAccountName: deepflow-server-sa
      securityContext:
        fsGroup: 2000           # arbitrary non-root for file access if needed
      containers:
        - name: server
          image: {{ .Values.deepflow.image }}
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: {{ .Values.deepflow.ports.data }} # deepflow data
port
            name: data-port
            - containerPort: {{ .Values.deepflow.ports.api }} # API port (for
Grafana)
            name: api-port
            - containerPort: {{ .Values.deepflow.ports.otel }}
# OTLP trace ingest (future use)
            name: otel-port
            env:
              - name: CLICKHOUSE_HOST
                value: "{{ .Release.Name }}-clickhouse" # use service name for
clickhouse (resolves to load-balancing across pods or leader)
```

```

    - name: MYSQL_HOST
      value: "{{ .Release.Name }}-mysql"
    - name: MYSQL_USER
      value: {{ .Values.mysql.user | quote }}
    - name: MYSQL_PASSWORD
      value: {{ .Values.mysql.password | quote }}
    - name: MYSQL_DATABASE
      value: {{ .Values.mysql.dbName | quote }}
  resources: {{ toYaml .Values.resources.server | nindent(10) }}
  volumeMounts:
    - name: config
      mountPath: /etc/deepflow           # if we provide custom config
(optional)
      readOnly: true
  volumes:
    - name: config
      configMap:
        name: deepflow-server-config      # assume created elsewhere if
needed
  terminationGracePeriodSeconds: 30
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchLabels: { app: deepflow-server }
      topologyKey: kubernetes.io/hostname

```

We pass the MySQL and ClickHouse connection info via environment variables. (In a more advanced setup, we might configure DeepFlow's server.yaml with these, but env is a simpler approach if the container supports it). The server will connect to `deepflow-mysql` service on default MySQL port 3306, and `deepflow-clickhouse` service on ClickHouse port (9000). We ensure a headless service exists for clickhouse if needed for clustering (not shown above: we'll define service-clickhouse for that).

MySQL StatefulSet: A straightforward MySQL deployment with a PVC. We set the root password via a secret or env (for simplicity in this code, we use env from values – not secure for real prod). We also create a database `deepflow` for the server to use. Since our Grafana might use its own database, we could also pre-create a `grafana` database if needed (DeepFlow docs mention a `grafana` DB when using managed MySQL). Here we assume Grafana uses SQLite internally, so we stick to one DB.

```

# templates/deepflow-mysql-statefulset.yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: {{ .Release.Name }}-mysql
  labels: { app: deepflow-mysql }
spec:

```

```

serviceName: {{ .Release.Name }}-mysql
replicas: 1    # one MySQL primary (no replica for now)
selector:
  matchLabels: { app: deepflow-mysql }
template:
  metadata:
    labels: { app: deepflow-mysql }
spec:
  containers:
    - name: mysql
      image: {{ .Values.mysql.image }}
      imagePullPolicy: IfNotPresent
      env:
        - name: MYSQL_ROOT_PASSWORD
          value: {{ .Values.mysql.password | quote }}
        - name: MYSQL_DATABASE
          value: {{ .Values.mysql.dbName | quote }}
      ports:
        - containerPort: 3306
          name: mysql
      volumeMounts:
        - name: mysql-data
          mountPath: /var/lib/mysql
      resources: {}
volumeClaimTemplates:
  - metadata:
      name: mysql-data
      annotations:
        volume.beta.kubernetes.io/storage-class: {{ .Values.global.storageClass
| quote }}
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: {{ .Values.resources.mysql.requests.storage | default
"1Gi" }}

```

This will create a PVC (e.g., `mysql-data-deepflow-mysql-0`) of 1Gi for MySQL data. If `global.storageClass` is not set in values, the annotation will be `null` and Kubernetes uses the default storage class.

ClickHouse StatefulSet: We deploy 3 pods (to match our 3 nodes) for distributed storage. Each pod will store a shard of data. We set it up with a volume for data and use the default ClickHouse config (which will treat each pod as a replica in the same shard set or as independent shards depending on config – for simplicity, assume it handles replication internally when pointing them at each other; DeepFlow might manage the distribution by writing to local pods). We expose the native port 9000 for client (DeepFlow server uses it).

```

# templates/deepflow-clickhouse-statefulset.yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: {{ .Release.Name }}-clickhouse
  labels: { app: deepflow-clickhouse }
spec:
  serviceName: {{ .Release.Name }}-clickhouse-headless
  replicas: {{ .Values.global.replicas }} # 3
  selector:
    matchLabels: { app: deepflow-clickhouse }
  template:
    metadata:
      labels: { app: deepflow-clickhouse }
    spec:
      containers:
        - name: clickhouse
          image: {{ .Values.clickhouse.image }}
          ports:
            - containerPort: 9000 # native client (TCP) port
              name: native
            - containerPort: 8123 # HTTP port (optional, for web UI or HTTP
client)
              name: http
      volumeMounts:
        - name: ch-data
          mountPath: /var/lib/clickhouse
      resources: {}
volumeClaimTemplates:
  - metadata:
      name: ch-data
      annotations:
        volume.beta.kubernetes.io/storage-class: {{ .Values.global.storageClass
| quote }}
  spec:
    accessModes: ["ReadWriteOnce"]
    resources:
      requests:
        storage: {{ .Values.resources.clickhouse.requests.storage | default
"10Gi" }}

```

We have also a headless service (`deepflow-clickhouse-headless`) for the StatefulSet (not shown but implied by `serviceName`) so that each pod can resolve others if needed. We will also define a ClusterIP service `deepflow-clickhouse` that perhaps routes to the first pod or uses selectors (for simplicity, DeepFlow server could connect via headless by iterating or via any pod – but likely DeepFlow can be

configured with multiple ClickHouse addresses or will use one if it expects a cluster; for now, assume one address is fine as they might handle cluster writes internally).

Services: We define services for discovery and access: - `deepflow-server` Service: ClusterIP service selecting app=deepflow-server pods. It should expose port 20417 (for agents to ingest) and 30880 (for Grafana to query). Agents and Grafana will use this service DNS. No external access. - `deepflow-grafana` Service: NodePort to expose Grafana's HTTP (3000) at the chosen `.Values.grafana.nodePort` (31999). Labeled app=deepflow-grafana. - `deepflow-mysql` Service: ClusterIP on port 3306, for server (and Grafana if needed) to connect to MySQL. - `deepflow-clickhouse` Service: ClusterIP on port 9000 (and optionally 8123) pointing to clickhouse pods. DeepFlow server connects here for writes/reads. Possibly you might use one of the pods or load balance; since ClickHouse might not be easily load-balanced at query protocol layer, a simple approach is to point DeepFlow to "clickhouse-0.clickhouse-headless" DNS of first pod. But for simplicity, we'll assume one service is okay (like if using replication, any node can accept writes).

Example service definitions:

```
# templates/service-deepflow.yaml
apiVersion: v1
kind: Service
metadata:
  name: {{ .Release.Name }}-server
  labels: { app: deepflow-server }
spec:
  type: ClusterIP
  clusterIP: None # headless (if needed for stable DNS per pod) - could be omitted for normal service
  selector: { app: deepflow-server }
  ports:
    - name: data
      port: {{ .Values.deepflow.ports.data }} # 20417
      targetPort: data-port
    - name: api
      port: {{ .Values.deepflow.ports.api }} # 30880
      targetPort: api-port
---
# templates/service-grafana.yaml
apiVersion: v1
kind: Service
metadata:
  name: {{ .Release.Name }}-grafana
  labels: { app: deepflow-grafana }
spec:
  type: {{ .Values.grafana.serviceType }} # NodePort
  {{- if eq .Values.grafana.serviceType "NodePort" --}}
  nodePort: {{ .Values.grafana.nodePort }}
  {{- end }}
```

```

    selector: { app: deepflow-grafana }
    ports:
      - port: 80
        targetPort: 3000
        protocol: TCP
        nodePort: {{ .Values.grafana.nodePort | default 31999 }}
    ---
# templates/service-mysql.yaml
apiVersion: v1
kind: Service
metadata:
  name: {{ .Release.Name }}-mysql
  labels: { app: deepflow-mysql }
spec:
  type: ClusterIP
  selector: { app: deepflow-mysql }
  ports:
    - port: 3306
      targetPort: mysql
      name: mysql
    ---
# templates/service-clickhouse.yaml
apiVersion: v1
kind: Service
metadata:
  name: {{ .Release.Name }}-clickhouse
  labels: { app: deepflow-clickhouse }
spec:
  type: ClusterIP
  selector: { app: deepflow-clickhouse }
  ports:
    - port: 9000
      targetPort: native
      name: native
    - port: 8123
      targetPort: http
      name: http

```

We set the deepflow-server service as headless (`clusterIP: None`). Actually, for agents and Grafana, a normal ClusterIP is fine (and simpler). We could remove `clusterIP: None` so that it gets a cluster IP – which is easier for load balancing across server pods. The above shows headless commented; likely better to use a normal service here so that it load-balances incoming agent connections among server pods.

Grafana Deployment: Finally, Grafana deployment with a PVC for its data (optional, we can run stateless and let it use emptyDir since dashboards are imported via DeepFlow or can be exported; but to persist any custom changes, a PVC or config map is needed). We set the admin credentials from values. We also mount

a ConfigMap or use environment to configure the DeepFlow data source. For brevity, let's include an env that points Grafana to DeepFlow's PromQL endpoint or a provisioning config.

```
# templates/deepflow-grafana-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}-grafana
  labels: { app: deepflow-grafana }
spec:
  replicas: 1
  selector:
    matchLabels: { app: deepflow-grafana }
  template:
    metadata:
      labels: { app: deepflow-grafana }
    spec:
      containers:
        - name: grafana
          image: {{ .Values.grafana.image }}
          imagePullPolicy: IfNotPresent
          env:
            - name: GF_SECURITY_ADMIN_USER
              value: {{ .Values.grafana.adminUser | quote }}
            - name: GF_SECURITY_ADMIN_PASSWORD
              value: {{ .Values.grafana.adminPassword | quote }}
            # Optionally, configure data source via environment or provisioning
            files.
              # For example, point to DeepFlow PromQL API (http://deepflow-server:
              30880).
            - name: GF_DATASOURCES
              value: |
                apiVersion: 1
                datasources:
                  - name: DeepFlow
                    type: prometheus
                    access: proxy
                    url: http://{{ .Release.Name }}-server:
{{ .Values.deepflow.ports.api }}
                    isDefault: true
                ports:
                  - containerPort: 3000
                    name: http
                volumeMounts:
                  - name: grafana-storage
                    mountPath: /var/lib/grafana # persist Grafana data (optional)
                volumes: []
```

```

# If persistence is desired:
# volumes:
# - name: grafana-storage
#   persistentVolumeClaim:
#     claimName: grafana-pvc

```

For simplicity, the above uses an env var `GF_DATASOURCES` with a YAML that Grafana can read to set up the data source (Grafana supports provisioning via file; doing it via env is a trick using the Docker entrypoint interpolation). This points Grafana to the DeepFlow server's Prom endpoint as the default data source. This means the built-in dashboards that query PromQL can fetch data from DeepFlow (DeepFlow's PromQL API provides metrics on flows, etc. in a Prometheus-compatible way). We could also add an SQL data source plugin for ClickHouse, but DeepFlow's approach is likely to use PromQL and their own APIs, so one data source is enough.

Now Grafana will come up at NodePort 31999 as specified. We have set admin user/password and configured the data source, so upon login, the user can directly view DeepFlow dashboards (the DeepFlow helm chart likely would have configured Grafana with JSON dashboards – we might need to include those as ConfigMaps if we want automatic import. For brevity we assume the user can import them or we ship a minimal one. In practice, we might bundle the JSON of DeepFlow's sample dashboard).

NetworkPolicy Examples: We define two NetworkPolicies: one for DeepFlow Server ingress, and one for DBs ingress. These implement the restrictions discussed:

```

# templates/networkpolicy-server.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: np-deepflow-server
  namespace: {{ .Release.Namespace }}
spec:
  podSelector:
    matchLabels: { app: deepflow-server }
  policyTypes: ["Ingress"]
  ingress:
  - from:
    # allow DeepFlow agents
    - podSelector: { matchLabels: { app: deepflow-agent } }
    # allow Grafana
    - podSelector: { matchLabels: { app: deepflow-grafana } }
    # allow other server pods (if server-to-server comm needed)
    - podSelector: { matchLabels: { app: deepflow-server } }
    ports:
    - { protocol: TCP, port: {{ .Values.deepflow.ports.data }} } # agent
      ingestion port
    - { protocol: TCP, port: {{ .Values.deepflow.ports.api }} } # query API
      port

```

```

---
# templates/networkpolicy-databases.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: np-deepflow-datastores
  namespace: {{ .Release.Namespace }}
spec:
  podSelector:
    matchLabels: { app: deepflow-mysql }
  # we'll apply one policy per selector via matchLabels
  policyTypes: ["Ingress"]
  ingress:
  - from:
    - podSelector: { matchLabels: { app: deepflow-server } }
    - podSelector: { matchLabels: { app: deepflow-grafana } }
    ports:
    - { protocol: TCP, port: 3306 }

---
# A separate NP for ClickHouse
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: np-deepflow-clickhouse
  namespace: {{ .Release.Namespace }}
spec:
  podSelector:
    matchLabels: { app: deepflow-clickhouse }
  policyTypes: ["Ingress"]
  ingress:
  - from:
    - podSelector: { matchLabels: { app: deepflow-server } }
    - podSelector: { matchLabels: { app: deepflow-clickhouse } } # allow
      ClickHouse cluster internal comm
  ports:
  - { protocol: TCP, port: 9000 }
  - { protocol: TCP, port: 8123 }

```

The above NetworkPolicies ensure:

- Only agent and grafana pods can talk to server pods on the relevant ports (blocking any other source).
- Only server (and grafana) can talk to MySQL on 3306.
- Only server (and other ClickHouse peers) can talk to ClickHouse pods on DB ports. Grafana's NodePort traffic is not affected (comes from outside, not a pod; by default, K8s network policy does not block host->pod traffic unless you explicitly use `from: {} with namespaceSelector` which we did not, so NodePort is accessible).

Helm Chart Notes: The chart is designed to be flexible:

- All images and versions can be changed via `values.yaml` (e.g., pointing to an internal registry mirror if the cluster is in an air-gapped environment).
- The `global.storageClass` can be set to an appropriate class (for Harvester, e.g., "longhorn") or left

empty to use the cluster default. - Resource requests/limits are set conservatively; these should be adjusted based on load. ClickHouse in particular might need more CPU/memory for heavy queries. - The default admin credentials for Grafana (`admin/deepflow`) should be changed for a real deployment. Similarly, the MySQL root password is exposed here for simplicity but should be handled via a Kubernetes Secret in production.

By using this chart, the entire DeepFlow stack is deployed consistently. On a 3-node Harvester cluster, we expect: - 3 agent pods (1 per node), - 3 server pods, - 3 ClickHouse pods, - 1 MySQL pod, - 1 Grafana pod.

All running within the `deepflow` namespace. This distributed setup aligns with the recommendation to use regular (non-all-in-one) deployment for multi-node environments, ensuring that if any pod moves, it reattaches to its PVC on the same node (thanks to StatefulSet), preserving data.

Finally, this chart can double as a Harvester addon. To make it a Harvester UI extension, one would package it in a git repo (as we have on GitLab) and add it to Rancher Manager's Apps & Marketplace as a repository. Then, the Harvester cluster can have this extension enabled via the Rancher UI, simplifying future upgrades and management. For now, the provided Helm chart and references in this document should allow a successful deployment of a secure, full-featured network monitoring stack on Harvester.

References

- Harvester architecture and networking
- DeepFlow documentation (Community Edition) on single cluster install
- DeepFlow architecture and zero-intrusion observability via eBPF ¹
- DeepFlow data storage (MySQL vs ClickHouse) and deployment modes
- DeepFlow built-in Grafana access details
- Kubernetes RBAC and least privilege best practice
- Kubernetes Pod Security Standards (Privileged vs Baseline)
- Kubernetes NetworkPolicies for traffic isolation

¹ ² ³ DeepFlow Architecture | Instant Observability for Cloud & AI Applications
<https://deepflow.io/docs/about/architecture/>