ingenieur
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

STL SYSTEMTECHNIKLABOR

Network flow collection, aggregation and analysis in Kubernetes Clusters

Matthias Riegler

Technical Report – STL-TR-2020-01 – ISSN 2364-7167

stl.htwsaar.de

Technische Berichte des Systemtechniklabors (STL) der htw saar
Technical Reports of the System Technology Lab (STL) at htw saar
ISSN 2364-7167

Matthias Riegler: Network flow collection, aggregation and analysis in Kubernetes Clusters
Technical report id: STL-TR-2020-01

First published: April 2020
Last revision: March 2020
Internal review: Klaus Berberich, Frank Ottink

For the most recent version of this report see: https://stl.htwsaar.de/

Title image source: Matthias Riegler

htw saar – Hochschule für Technik und Wirtschaft des Saarlandes (University of Applied Sciences)
Fakultät für Ingenieurwissenschaften (School of Engineering)
STL – Systemtechniklabor (System Technology Lab)
Prof. Dr.-Ing. André Miede (andre.miede@htwsaar.de)
Goebenstraße 40
66117 Saarbrücken, Germany
https://stl.htwsaar.de

ingenieur
wissenschaften
htw saar

Daimler Protics
Data driven by Passion

Bachelor-Thesis

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)
an der Hochschule für Technik und Wirtschaft des Saarlandes
im Studiengang Praktische Informatik
der Fakultät für Ingenieurwissenschaften

Network flow collection, aggregation and analysis in Kubernetes Clusters

vorgelegt von
Matthias Riegler

betreut von
Dipl.-Informatiker Frank Ottink (Daimler Protics GmbH)
Prof. Dr. Klaus Berberich
begutachtet von
Prof. Dr. Klaus Berberich

Saarbrücken, 30.03.2020

Selbständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich erkläre hiermit weiterhin, dass die vorgelegte Arbeit zuvor weder von mir noch von einer anderen Person an dieser oder einer anderen Hochschule eingereicht wurde.

Darüber hinaus ist mir bekannt, dass die Unrichtigkeit dieser Erklärung eine Benotung der Arbeit mit der Note „nicht ausreichend“zur Folge hat und einen Ausschluss von der Erbringung weiterer Prüfungsleistungen zur Folge haben kann.

Saarbrücken, 30.03.2020

Matthias Riegler

Abstract

Nowadays large scale enterprise applications are adopting microservice architectures in order to improve scalability, agility and flexibility which are in stark contrast to a monolithic architecture. The benefit of easy and dynamic scaling comes at the cost of a more complex, distributed system containing many interacting components (microservice). While each component is intended to do just one thing, the complexity of the architecture moves from the component layer to the interconnect layer.

A common platform for operating microservice architectures in production environments is Kubernetes (k8s) [28], a container orchestrator organizing one or more containers into pods and additionally providing a common set of control routines managing their states.

The focus of this thesis is to establish a proof of concept system which provides an insight into the communication between pods by collecting network flows on a per-pod basis and persisting them on an Elasticsearch [1] database for future visualisation and analysis with Kibana [2].

Contents

1 Introduction **1**
- 1.1 Current Network Monitoring Technologies For Kubernetes . . . 1
- 1.2 Motivation . . . 1
- 1.3 Organization . . . 1
  - 1.3.1 Chapter 1: Introduction . . . 1
  - 1.3.2 Chapter 2: Technical Background . . . 2
  - 1.3.3 Chapter 3: Use Case Discussion . . . 2
  - 1.3.4 Chapter 4: Platform Design . . . 2
  - 1.3.5 Chapter 5: Implementation . . . 2
  - 1.3.6 Chapter 6: Evaluation . . . 2
  - 1.3.7 Chapter 7: Conclusions & Future Work . . . 2

2 Technical Background **3**
- 2.1 Flow Definition . . . 3
  - 2.1.1 Flow Hashing . . . 3
- 2.2 Linux Containers . . . 3
  - 2.2.1 Process Isolation Using Linux Namespaces . . . 4
  - 2.2.2 Resource Limitation Using Control Groups . . . 4
  - 2.2.3 Restricting Root Privileges . . . 4
  - 2.2.4 Network Isolation . . . 5
  - 2.2.5 Building Container Images . . . 7
- 2.3 Kubernetes . . . 7
  - 2.3.1 Resource Definitions . . . 7
  - 2.3.2 Namespaces, Labels and Annotations . . . 8
  - 2.3.3 Pod As Atomic Unit . . . 8
  - 2.3.4 Architecture . . . 8
  - 2.3.5 Configuration And Secret Management . . . 9
  - 2.3.6 Pod Management . . . 9
  - 2.3.7 Interacting With The API . . . 9
  - 2.3.8 Patching Resources Before Creation . . . 10
  - 2.3.9 Networking In Kubernetes . . . 10
  - 2.3.10 Kubernetes Service . . . 11
- 2.4 Capturing Network Packets . . . 12
- 2.5 ELK-Stack . . . 13
  - 2.5.1 Elasticsearch . . . 13
  - 2.5.2 Logstash . . . 15
  - 2.5.3 Kibana . . . 15
- 2.6 PostgreSQL . . . 16
  - 2.6.1 Persisting JSON Objects . . . 16
  - 2.6.2 Working With JSON Objects . . . 16
- 2.7 Memcached . . . 17

3 Use Case Discussion **19**
- 3.1 Multi-Tenant Clusters . . . 19

3.2 Validating Load Balancing . . . 19
3.3 Root Cause Analysis . . . 19
3.4 Debugging Uncommon Network Operations . . . 19
3.5 Resulting Requirements . . . 19

4 Platform Design **21**
4.1 High-Level Architecture . . . 21
4.1.1 Data Collection . . . 22
4.1.2 Flow Enhancing, Storage and Visualization . . . 22
4.2 Low-Level Design . . . 22
4.2.1 Kubeagent . . . 22
4.2.2 Network Probe . . . 22
4.2.3 Probe Injector . . . 23
4.2.4 Resolver . . . 25

5 Implementation **27**
5.1 Compatibility & Used Libraries . . . 27
5.2 Automated Testing, Compilation And Delivery . . . 28
5.3 Unified Code Base & Release Management . . . 28
5.4 Network Flows . . . 29
5.4.1 Unique Identifier . . . 29
5.4.2 ICMP And ICMPv6 Flow Hashing . . . 29
5.4.3 Source Detection . . . 31
5.5 Network Probe . . . 32
5.5.1 Intercepting Network Traffic . . . 32
5.5.2 Extracting A Sample . . . 33
5.5.3 Aggregating Packets To Flows . . . 33
5.5.4 Generating A Flow Event . . . 33
5.6 Kubeagent . . . 34
5.6.1 PostgreSQL In Memory Database . . . 34
5.6.2 Kubernetes To SQL Middleware . . . 34
5.7 Resolver . . . 35
5.7.1 Tapping The Conntrack Table . . . 35
5.7.2 Filtering For Kube-Proxy Events . . . 35
5.8 IP To Kubernetes Object Mapping . . . 35
5.9 Logstash Pipeline . . . 36
5.10 Probe Injector . . . 37
5.11 Visualization using Kibana . . . 37

6 Evaluation **41**
6.1 Platform Scaling And Performance Discussion . . . 41
6.1.1 Component Scaling . . . 41
6.1.2 Resolver performance issues . . . 41
6.1.3 Network Probe performance impact . . . 41
6.2 Security Concerns . . . 42
6.3 Pratical Testing . . . 42
6.3.1 Test Setup . . . 42
6.3.2 Measuring The Performance Impact Of The Network Probe . . . 42
6.3.3 Load Balancer Validation . . . 43
6.3.4 Comparison To Metric Based Monitoring . . . 44
6.3.5 Verifying Correct Canary Deployments . . . 44

7 Conclusion & Future Work **47**

Bibliography **51**

List Of Figures **55**

List Of Tables **55**

Listings **55**

A Network Flow Event **57**

B Shortened CI/CD-Pipeline **59**

C PostgreSQL Table Layout **61**

D Logstash Pipeline **63**

E Elasticsearch Index Template **69**

F Test Application Deployment **75**

G Test Canary Deployment **77**

1 Introduction

Nowadays, large scale enterprise applications are embracing microservice architectures intending to improve scalability, agility, and flexibility. Comparing this approach with a classic three-tier architecture, complexity shifts from the application to the interconnect layer. While several tools are assisting the operation of microservices, concepts presented in this thesis will focus on Kubernetes (k8s) [28].

The presented proof of concept with the project name `Insight` will be a platform helping to get an insight into the networking layer of k8s.

1.1 Current Network Monitoring Technologies For Kubernetes

There is a large number of metrics that can be retrieved from k8s itself, its underlying container runtime, external load balancers and more. An often seen open-source approach is scraping and storing the metrics with Prometheus and visualizing them with Grafana [51]. General network metrics such as the currently used bandwidth can be retrieved but network flows containing source and destination alongside traffic statistics facing the endpoints are not provided and cannot be stored with the mentioned toolsets. While there are other open-source, free-to-use and commercial monitoring systems for k8s available, as of December 2019, none of them support monitoring network flows inside the cluster.

A recent approach for microservices is a service mesh based on a protocol proxy server deployed next to the microservice handling e.g. HTTP request routing while collecting metrics.

1.2 Motivation

The objective of this thesis is to collect network flows between microservices, store them inside an Elasticsearch [1] database, join extracted flows with metadata retrieved from the k8s API and internal systems and match correlating flows based on a unique identifier. The data format should be compatible with the Security Information and Event Management (SIEM) view shipped with Kibana [2], assisting an end-user to analyze collected flow data.

In comparison to metric based network monitoring, flow based network monitoring aggregates network utilization for each communication participant and not for a single network interface. This additional information allows validation of several high level applications such as service meshes and load balancing. The system also allows to identify hardware issues such as overloaded routers which are dropping packets to specific destinations. A metric based monitoring would show dropped packets but the flow based monitoring allows a root cause analysis by identifying failing connections.

1.3 Organization

1.3.1 Chapter 1: Introduction

The first chapter introduces the concept of a flow monitoring system and highlights the motivation.

1.3.2 Chapter 2: Technical Background

The second chapter starts by introducing the flow terminology and required background on container based systems and k8s with a focus on networking. In addition, components of the ELK-Stack, PostgreSQL and Memcached are introduced.

1.3.3 Chapter 3: Use Case Discussion

Chapter three discusses possible use cases for flow monitoring and lists the resulting requirements for the platform.

1.3.4 Chapter 4: Platform Design

Chapter four presents a platform design meeting the requirements of the requrement analysis in the previous chapter. It starts with a high level architecture before highlighting details of each component.

1.3.5 Chapter 5: Implementation

The fifth chapter explains the implementation based on the platform design.

1.3.6 Chapter 6: Evaluation

The working proof of concept is evaluated first on a theoretical aspect discussing scalability, known issues and security and then based on its performance in simulated test scenarios.

1.3.7 Chapter 7: Conclusions & Future Work

The last chapter concludes the thesis and introduces potential fixes for an improved proof of concept or production system.

2 Technical Background

2.1 Flow Definition

A unidirectional network layer flow, is a sequence of Internet Protocol (IP) packets from a source computer to a destination computer [59]. Its direction can be represented as tuple (IP_{src}, IP_{dst}) where IP_{src} is the source and IP_{dst} the target.

Furthermore, the direction of the flow at the transport layer, bringing in a port/type based protocol, can be represented as an unordered tuple where P_{src} and P_{dst} describe additional parameters of the transport protocol:

$$\left\langle (IP_{src}, P_{src}), (IP_{dst}, P_{dst}) \right\rangle \tag{2.1}$$

A bidirectional transport layer flow with traffic statistics can be visualized as a graph and is further referred to as flow:

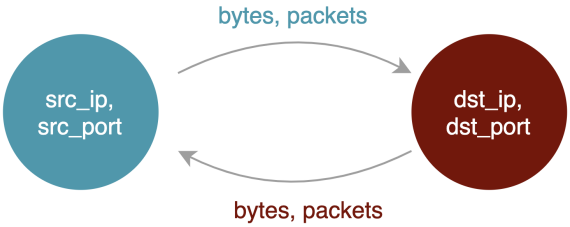

Figure 2.1: Bidirectional Flow As Graph

Weight attributes `packets` and `bytes` contain traffic statistics differentiated for source to destination and destination to source.

2.1.1 Flow Hashing

When searching for correlating flows, source and destination endpoints can be swapped. This issue is addressed by assigning flows with swapped endpoints, a unique identifier.

Flow hashing implements the desired behavior by computing a hash based on the flow endpoints. A hash collision on swapped endpoints takes away the direction of the flow.

$$h\left(\left\langle (IP_{src}, P_{src}), (IP_{dst}, P_{dst}) \right\rangle\right) \Leftrightarrow h\left(\left\langle (IP_{dst}, P_{dst}), (IP_{src}, P_{src}) \right\rangle\right) \tag{2.2}$$

2.2 Linux Containers

Nowadays, containers are becoming a popular way of packaging and running applications reliably by abstracting the userspace. They ship as standardized container images [48] providing application executables alongside required runtime resources such as shared libraries. Containers do not use a hypervisor for creating an isolated view on the system but handle the isolation at the Linux Kernel layer. As seen in Figure 2.2, container isolation has fewer components than hypervisor virtualization resulting in less runtime overhead

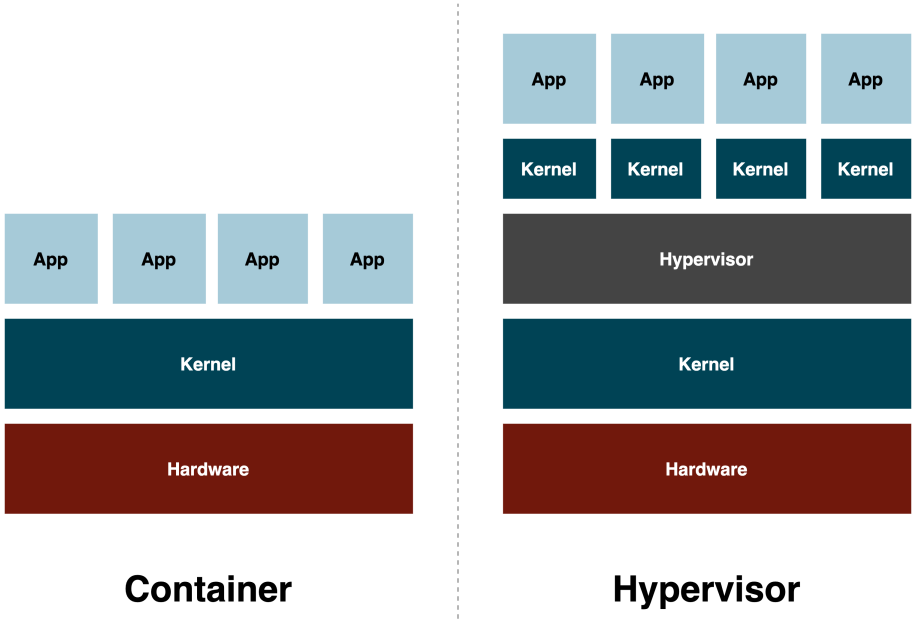

Figure 2.2: Container Hypervisor Comparison

[38]. Besides, the application footprint is smaller and the startup time decreases since the application does not require booting a new kernel. In this thesis when referring to a container, a group of processes isolated by `control groups (cgroups)`, `capabilites` and Linux `namespaces` is referred to.

2.2.1 Process Isolation Using Linux Namespaces

Linux namespaces isolate processes, changing their view on the system by logically dividing their kernel space into multiple environments [17]. The kernel implements several namespaces including mount namespaces, Inter Process Communication (IPC) namespaces, Process Identifier (PID) namespaces, Unix Time Sharing (UTS) namespaces, network namespaces and user namespaces. User, mount and PID namespaces isolate a container from its host by building a sandbox where processes placed in it have one or more separate users, a separate file system and can only see processes running in the shared PID namespace. The IPC, network and UTS namespaces further secure the sandbox. While these are the base of container isolation, there are several other kernel namespaces enhancing isolation and security of the container sandbox.

2.2.2 Resource Limitation Using Control Groups

An everyday usage pattern is the limitation of CPU and memory utilization by the container so a single one cannot exhaust all system resources. With cgroups, the Linux Kernel brings a functionality to control hardware resources assigned to a group of processes. In comparison to `ulimit` and `rlimit`, it provides a more fine-grained control over e.g. CPU time, memory usage, input-output operations or network priority [7]. The configuration is done via a virtual filesystem mounted to `/sys/fs/cgroup`.

2.2.3 Restricting Root Privileges

While traditional UNIX implementations differentiate between unprivileged and privileged users, latter bypassing all kernel permission checks, Linux introduced with version 2.2

the concept of `capabilities` [6]. The described capabilities implement an additional authorization layer for distinct kernel calls (Table 2.1), which cannot be bypassed. A process in a container, without further configuration, runs as `root`. Linux capabilities, more precisely the lack of them for processes running inside the container, are used to address this issue and remove privileges from the container root user.

Capability	Behavior
`CAP_NET_RAW`	Use `Packet` and `RAW` sockets
`CAP_SETUID`	Change the user id of a running process
`CAP_SYS_BOOT`	Reboot the system or load a new kernel
`CAP_SYS_TIME`	Set the system or hardware clock

Table 2.1: Selection of Linux Capabilities [6]

2.2.4 Network Isolation

Linux `network namespaces` are used for isolating networking. Every network namespace gets its own isolated network devices, routing table and firewall, thus isolating the container from the host. Populating the routing table and network interfaces placed in the container depends on the used Container Network Interface (CNI) implementation. When provisioning a container, the network namespace is created and passed to a plugin for further configuration. Replicating the described process is straightforward using `iproute2` (Listing 2.1). After creating the network namespace, the container runtime calls a CNI

```
# Create a container network namespace named purple
$ ip netns add purple
# Execute a command in the purple network namespace
$ ip netns exec purple ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

Listing 2.1: Creating a linux network namespace

plugin hooking in and e.g. creating a virtual ethernet pair [33] moving one side to the container network namespace establishing a pipe between container and host. Now IP addresses can be assigned to each interface. A *real* CNI plugin relies on an IP Address Management (IPAM) system for retrieving an IP address (Listing 2.2).

```
# Create a virtual ethernet pair
$ ip link add hostside0 type veth peer name containerside0
# Move one side to the container network namespace
$ ip link set containerside0 netns purple
# Assign IP addresses, routing table will be populated
$ ip netns exec purple ip addr add 172.16.0.2/24 dev containerside0
$ ip addr add 172.16.0.1/24 dev hostside0
# Set the interface states to UP
$ ip link set hostside0 up
$ ip netns exec purple ip link set containerside0 up
$ ip netns exec purple ip link set lo up
# View the IP configuration
$ ip netns exec purple ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group ...
   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
16: containerside0@if17: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc ...
   link/ether 9e:63:59:29:53:78 brd ff:ff:ff:ff:ff:ff link-netnsid 0
   inet 172.16.0.2/24 scope global containerside0
      valid_lft forever preferred_lft forever
   inet6 fe80::9c63:59ff:fe29:5378/64 scope link
      valid_lft forever preferred_lft forever
# Show routing tables (container & host)
$ ip netns exec purple ip route show
172.16.0.0/24 dev containerside0 proto kernel scope link src 172.16.0.2
$ ip route show
default via 2.56.96.1 dev ens3 proto static metric 100
2.56.96.0/22 dev ens3 proto kernel scope link src 2.56.99.19 metric 100
10.10.0.0/16 dev cni0 proto kernel scope link src 10.10.0.1
172.16.0.0/24 dev hostside0 proto kernel scope link src 172.16.0.1
```

Listing 2.2: Example CNI initializing container networking

2.2.5 Building Container Images

```
# Import the base image
FROM golang:1.13-alpine as builder

WORKDIR /app
copy . .

# Build application
RUN go get -d -v ./...
RUN go build cmd/probeinject/probeinject.go

# Generate a smaller docker image without build dependencies
FROM alpine:3.11

# Copy built binary
COPY --from=builder /app/probeinject /probeinject
ENTRYPOINT ["/probeinject"]
```

Listing 2.3: Dockerfile example

A container image, as specified by the Open Container Initiative [48] can be built in several ways. Docker [46] introduced the `Dockerfile`, a file describing the building blocks of a container image (refer to Listing 2.3 as example). Starting with a base, every instruction in the Dockerfile creates an additional layer merged with the previous layers. The layering helps to cache and reuse executed instructions and lowers the total footprint of an application, as e.g. the base layer can be reused by several images.

2.3 Kubernetes

Kubernetes (k8s) is a borg [61] inspired set of interconnected services working together as a container orchestrator [25]. Viewed from a high-level perspective k8s tries to build and maintain a desired state persisting of a set of objects (resources). K8s ships with a predefined set of resource types and adds the option of providing new resource types in the form of a Custom Resource Definition (CRD) [25].

2.3.1 Resource Definitions

The state of an object inside k8s is commonly described in the YAML Ain't Markup Language (YAML) [24] which is a superset of JavaScript Object Notation (JSON) [3] including a more human friendly reading and writing experience alongside with comments. Resources can be split into five categories [27]:

- `Workload` manage runtime components (actual running processes)
- `Discovery and load balancer` resources define the connection from the outside and between containers
- `Cluster` resources contain the cluster configuration
- `Config` and `Storage` resources are used for persisting data and injecting data into applications

- `Metadata` resources define the behaviour of other resources

Every resource is described by three `Resource Objects` [27]:

- `Resource ObjectMeta`
- `ResourceSpec`
- `ResourceStatus`.

The `Resource ObjectMeta` inhabits the `apiVersion:` attribute defining the target Application Programming Interface (API) path supplemented by `kind:` defining the resource type. In addition, the context of a resource is described in the `Resource ObjectMeta`, using the `metadata:` field. Depending on the `kind:` attribute, different fields have to be included as `ResourceSpec`. A common pattern is e.g. the `spec:` or `data:` attribute. Refer to Listing 2.4 for an example. k8s stores the current state of a resource in the `ResourceStatus` identified by the `status:` field. All resources in k8s can be created, updated (replaced or patched), read and deleted using the k8s API.

2.3.2 Namespaces, Labels and Annotations

Some resources of k8s can be placed in a namespace defined by the `metadata.namespace:` attribute. Next to the assigned unique identifier, `metadata.namespace:` and `metadata.name:` are unique for each resource type. The k8s metadata fields contain `metadata.labels:` and `metadata.annotations:`. Annotations allow to persist any additional configuration details for an object while labels help grouping related resources. Labels are also used as filter criteria when talking to the API.

2.3.3 Pod As Atomic Unit

The smallest runtime component of k8s is called a pod [26]. It models an "application-specific logical host" [26] containing one or more containers scheduled in a shared context. This context includes a shared namespace including a common network namespace access to shared volumes. In difference to a single Linux container, multiple containers defined in a pod can use IPC [52] and also communicate over a shared localhost. This tight coupling allows e.g. the vertical integration of selected components of a Linux Apache MySQL PHP (LAMP) Stack [58] Listing 2.4.

A pod is the only ephemeral resource in k8s, e.g. if the physical machine where the pod is running on crashes, k8s does not recreate the pod and deletes it instead.

2.3.4 Architecture

Components of k8s can be split into the **Control Plane** and **Kubernetes Nodes**.

The **Control Plane** consists of several microservices all interfaced by the `kube-api-server` which itself provides the API and acts as the frontend for k8s. Kubernetes components are stateless and use the `kube-api-server` to persist cluster-state inside `etcd`, a distributed key-value store [9]. The `kube-scheduler` is responsible for assigning a pod to a node and the `kube-controller-manager` runs essential control loops watching the cluster-state and taking actions if needed. If k8s runs in a cloud environment, the `cloud-controller-manager` interacts with the cloud provider and provisions e.g. volumes and load balancers.

Kubernetes Nodes are either a virtual or physical computer running Linux. On top of the operating system, the `kubelet` interacts with a Container Runtime Interface (CRI) and runs pods on the node. The `kube-proxy` manages networking rules and is discussed in Section 2.3.9.

```yaml
apiVersion: v1
kind: Pod
metadata:
    name: test
    labels:
        app: test
spec:
    containers:
    - name: php-component
      image: some-php-image
      ports:
        - containerPort: 9000
          name: fastcgi
    - name: apache
      image: apache
      ports:
        - containerPort: 80
          name: http
```

Listing 2.4: k8s Example Pod Definition

2.3.5 Configuration And Secret Management

To ease configuration and secret management, k8s comes with the `ConfigMap` and `Secret` resource type. Both can be mounted in a container filesystem or fill environment variables (Listing 2.5). The difference between a `ConfigMap` and a `Secret` is the sensitivity of the data. A `Secret` is intended to contain sensitive information including passwords, ssh keys or certificates and can be managed as an opaque object avoiding secret leakage while querying the API [27].

2.3.6 Pod Management

k8s serves the role of an container orchestrator by providing several workload controllers used to guarantee application availability by creating or destroying pods. A Microservice can be provided in the form of a k8s `Deployment` containing a pod template. The Deployment manages a ReplicaSet which itself is creating and destroying pods in order to maintain a desired state (e.g. five running instances). After updating the Deployment, the active ReplicaSet scales down to 0 instances while a new one gets created. Pod templates are passed to the ReplicaSets. Dynamic scaling of the ReplicaSets allows rollbacks to older revisions of a Deployment [27].

If a pod is managed by a ReplicaSet, it contains the field `metadata.ownerReferences:` with an entry mapping the ReplicaSet Listing 2.6. The `ReplicaSet` Listing 2.7 itself is owned by a Deployment Listing 2.8.

Apart from `Deployments` there are other resource types managing pods such as a `DeamonSet` scheduling one pod on every node or the `StatefulSet` extending a `Deployment` with persistent data.

2.3.7 Interacting With The API

The k8s API can be accessed using the secure Hypertext Transfer Protocol (HTTPS). While not being required, the default configuration authenticates clients using a `Certificate`

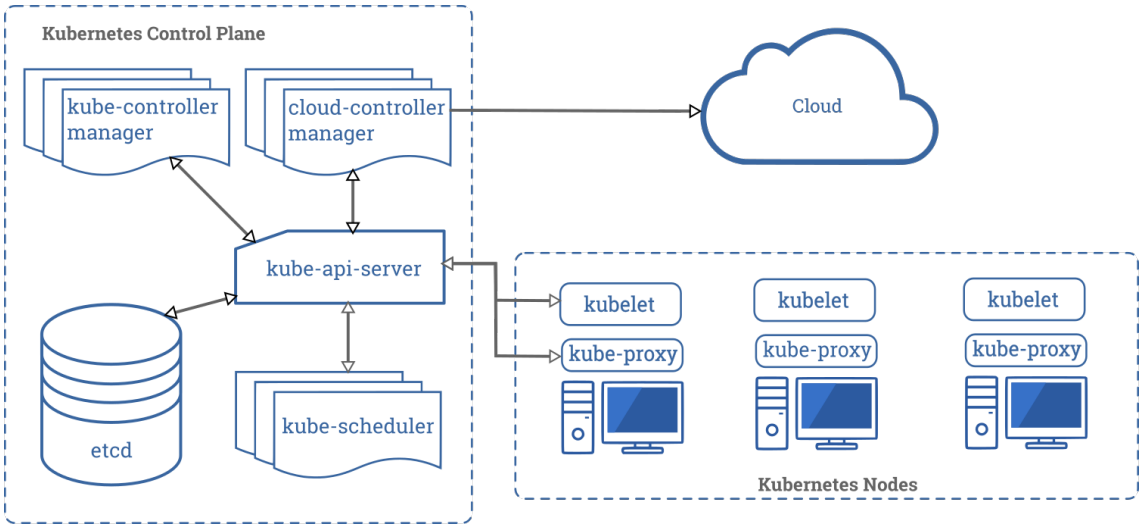

Figure 2.3: Kubernetes Components [26]

`Request` [57] and manages permissions using a Role Based Access Control (RBAC) [39]. `Workload` resources as described in Section 2.3.1 are grouped by the `Namespace` stated in the metadata field. By sending requests to specific paths, e.g. `/api/v1/pods`, resources can be created udpdated or deleted using Hypertext Transfer Protocol (HTTP) POST, UPDATE or DELETE operations.

Authentication to the API inside the cluster requires so-called `ServiceAccounts`. Being generated for every namespace, assigned certificates get injected to every container in a pod. However, it is possible to create a custom ServiceAccount with extended permissions, e.g. creating pods. The API client used in this thesis is the official `client-go` [14]. When running inside a pod, `client-go` authenticates against the API using the provided `ServiceAccount`.

2.3.8 Patching Resources Before Creation

Extending the API functionality of k8s is possible by using the builtin `Dynamic Admission Control`. When configured, a web hook facing an HTTPS endpoint will be used to query an external controller which is either validating or mutating.

When utilizing a validating controller, the `ValidatingWebhookConfiguration` describes when the web hook needs to be triggered and where it is reachable alongside the server's Transport Layer Security (TLS) certificates.
In case the controller is intended to be mutating, a `MutatingWebhookConfiguration` containing the same information needs to be used instead.

Contrasted to a validating controller, a mutating controller can not only allow/deny the creation of a resource but alternately modify it before it gets persisted to disk. The requested change can be transmitted using a JSON patch [4]. It is e.g. possible to add a label to the metadata field as shown in Listing 2.9.

2.3.9 Networking In Kubernetes

Interconnecting pods and cluster-networking is a central part in k8s and can be divided into four categories:

1. container to container networking (shared network namespace, refer to Section 2.3.3)
2. pod to pod networking (covered in this section)

```yaml
apiVersion: v1
kind: Pod
metadata:
    name: test
spec:
    containers:
    - name: needs-secret-and-env
      image: test
      envFrom:
      - configMapRef:
          name: bar-configmap
      volumeMounts:
      - name: foo
        path: "/foo"
    volumes:
    - name: foo
      secret:
        secretName: foo-secret
```

Listing 2.5: k8s Example Pod definition mounting a secret and getting environment variables from a `ConfigMap`

3. pod to `Service` networking (refer to Section 2.3.10)
4. `External` to `Service` networking (refer to Section 2.3.10)

Every pod in k8s gets a cluster-unique IP address assigned and can communicate with all pods running on any node without requiring Network Address Translation (NAT) [37]. Alongside every k8s agent (e.g. the `kubelet`) and system daemons can access pods running on the same node. It is also possible to schedule a pod in the host network namespace instead of an encapsulated environment. Having one IP per pod eliminates conflicts resulting from port exhaustion (two services listening on the same port).

The networking component does not ship with k8s directly; instead, several CNI plugins implement the described network model. This diversity allows customized performance and feature tuning for different cloud providers and bare metal. Some CNI plugins also bring support for `NetworkPolicies` describing firewall rules between pods. The CNI plugin also maintains the Node to Node communication [29]. This thesis is using `Flannel` [47] as a CNI plugin, which abstracts connection to different nodes using an overlay network [54] based on `VXLAN` [53].

2.3.10 Kubernetes Service

Pods are short-lived in k8s thus addressing an application by the pod IP is inconvenient and only a temporary solution. In k8s this issue is addressed by a `Service`. A service targets a group of pods and distributes the load across them. It is represended by a stable, unique IP address called `ClusterIP`. Requests to the ClusterIP are redirected to one of the pods implementing a load balancer. Each service can either be User Datagram Protocol (UDP) [60] or Transmission Control Protocol (TCP) for a single IP family.

A service is cluster internal, but can be exposed by using the type `NodePort`.This type redirects a set of TCP or UDP ports on each k8s node to the ClusterIP. In cloud environments, the type `load balancer` leads to an external load balancer placed in front

```yaml
apiVersion: v1
kind: Pod
metadata:
  ...
  name: insight-staging-postgres-7dc9594b7d-59f7h
  namespace: insight-staging
  ownerReferences:
  - apiVersion: apps/v1
    blockOwnerDeletion: true
    controller: true
    kind: ReplicaSet
    name: insight-staging-postgres-7dc9594b7d
    uid: d7705630-2140-4b42-a338-56f3b9e5401e
  resourceVersion: "35434435"
  uid: 2446eeae-21fa-4e91-8b75-2351f4ad1f4c
spec:
...
status:
...
```

Listing 2.6: pod with owner reference

of the NodePorts. It is common to have one public-facing HTTP/HTTPS service while keeping everything else internal. Often seen is SSL offloading to the external load balancer.

In a classic k8s installation, services are implemented using a combination of `iptables` rules configured by the `kube-proxy`. The `kube-proxy` maintains an up-to-date list of possible target pods for each service and creates, updates and deletes iptables rules accordingly [26].

Requests facing the ClusterIP are redirected to a target in a round-robin manner. The NAT is efficiently realized at the Kernel layer and shows up in the conntrack table.

A current development of k8s in alpha state is an IP Virtual Server (IPVS) based service proxy which will support more complex load balancing algorithms such as *shortest delay expected* [50, 26].

2.4 Capturing Network Packets

A socket describes an endpoint for data communication between the Linux kernel- and userspace. The domain argument passed on socket creation defines the protocol used for communication [22]. While domains like `AF_INET` is designated to IPv4, the `AF_PACKET` domain captures all ethernet frames on a specified port.

Libpcap, the underlying library of tcpdump, implements an API for intercepting network traffic [23]. Having Linux as target system, an `AF_PACKET` socket is used for capturing ethernet frames. In this thesis, Gopacket [31], a Golang library based on libpcap is used for parsing captured ethernet frames to the different protocol layers.

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  ...
  name: insight-staging-postgres-7dc9594b7d
  namespace: insight-staging
  ownerReferences:
  - apiVersion: apps/v1
    blockOwnerDeletion: true
    controller: true
    kind: Deployment
    name: insight-staging-postgres
    uid: 16e7d01d-d3ba-4cc6-af02-6757a8ff7b94
  resourceVersion: "35434437"
  uid: d7705630-2140-4b42-a338-56f3b9e5401e
spec:
  replicas: 1
...
status:
  availableReplicas: 1
```

Listing 2.7: `Deployment` with owner reference

Domain	Description
`AF_UNIX`	Local communication
`AF_INET`	IPv4 internet protocol
`AF_INET6`	IPv6 internet protocol
`AF_PACKET`	Low-Level packet interface

Table 2.2: Selection of supported Linux socket domains [22]

2.5 ELK-Stack

2.5.1 Elasticsearch

Elasticsearch is a distributed, document-based database focused on scalability and performance [5]. It is designed to hold a variety of data, including but not limited to logs and system metrics. After first normalizing and indexing documents, they can be queried and aggregated using a JSON based query language [5]. It relies on an inverted index [62] as an underlying data structure allowing performant near-real-time queries and aggregations.

2.5.1.1 Logical Layout

A JSON based document in Elasticsearch consists of fields with specific types:

1. `object`, `nested`
2. atomic types e.g. `IP`, `keyword`, `long`

Documents with a common type schema, further referred to as mapping, are stored within a common `Index` [5]. The mapping can be determined automatically or set for a specific

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  ...
  generation: 1
  name: insight-staging-postgres
  namespace: insight-staging
  resourceVersion: "35434438"
  uid: 16e7d01d-d3ba-4cc6-af02-6757a8ff7b94
spec:
  replicas: 1
  selector:
    matchLabels: ...
  template:
    metadata:
      creationTimestamp: null
      labels: ...
    spec:
      containers:
      - image: postgres:12.1-alpine
        imagePullPolicy: IfNotPresent
        name: postgres
      ....
status:
  availableReplicas: 1
```

Listing 2.8: k8s `Deployment`

index. It also allows for dynamic fields which can be present in one document but not another.

2.5.1.2 Physical Layout

Elasticsearch is usually run in a clustered environment consisting of many nodes. An Elasticsearch node can have one or more of the following node-types:

- `master-eligible` nodes can be elected as master
- `data` nodes hold data and perform related operations
- `ingest` nodes execute pre-processing pipelines
- `machine learning` nodes run machine learning related operations

When using n master nodes, at least $\lceil \frac{n}{2} \rceil$ have to be available for the cluster to function, thus preventing a split-brain scenario [5].

An index can be split into multiple shards, which allows distributing data and load to multiple data nodes. To prevent data loss, Elasticsearch supports replicating shards on a set number of different data nodes [5]. While the replica count adjusts on demand, the shard count has to be set at index creation and cannot be changed afterward [5].

```
{
  "op": "add",
  "path": "metadata/labels/webhook",
  "value": "hello world"
}
```

Listing 2.9: JSON patch example for adding a k8s label

2.5.1.3 Working with Elasticsearch

The API of Elasticsearch is based on the HTTP protocol and JSON as data exchange format.

Data can be ingested to Elasticsearch using a `POST` operation and complex queries can be formulated using the JSON based Query Domain Specific Language (DSL) of Elasticsearch [5].

Elasticsearch also allows for providing a template for indices matching a specified pattern. Settings for the physical layout and type mapping are then applied whenever a new index gets created [5].

2.5.2 Logstash

Logstash is a central part of the ELK-Stack and is a server-side data processing pipeline used to aggregate and process data before it is ingested to Elasticsearch [5]. When used in a stateless environment, it can be scaled horizontally and allows load balancing. A pipeline is split into three processing stages [5]:

1. `Input`
2. `Filter`
3. `Output`

Logstash supports a wide range of event inputs ranging from raw TCP or UDP streams to complex event like scheduled database queries. After an event is received, it is processed by several plugins in the filter subsection. Again, a wide range of plugins can be used including a from a simple field extraction to a cached database query. The output subsection defines where the processed event should be transmitted to. Next to a predefined set of plugins, it is possible to implement complex logic or additional plugins using Ruby [5, 20].

2.5.3 Kibana

Kibana [2] is a user-facing frontend for Elasticsearch providing tools for visualizing and managing data stored in Elasticsearch. Depending on the licensing model (basic license is used for this thesis), it supports a variety of builtin applications visualizing and analyzing data in different ways. Instead of working with a single index, it supports index patterns, which are regular expressions matching multiple indices [5].

2.5.3.1 Kibana Query Language

The Kibana Query Language (KQL) is an intentionally easy to use query language narrowing down the result set [5]. It supports possibly chained comparisons for numeric and text fields alongside wildcard selections. The query in Listing 2.10 limits the result set to documents with `destioantion.ip` matching `8.8.8.8` and a nested object under `source.kubernetes.pods` which matches `labels.app = dns-resolver`.

```
destination.ip = "8.8.8.8" and source.kubernetes.pods:{labels.app: test}
```

Listing 2.10: Kibana Query Language (KQL) example

2.5.3.2 Elastic SIEM

Shipped with the basic version of Elastic is an extensive SIEM view, which helps to analyze security-related data stored in Elasticsearch, including network flows. It supports multiple views, including an event overview and network summarization. One usage pattern is presented in Section 5.11

2.5.3.3 Elastic Common Schema

The Elastic Common Scheme (ECS) specification defines how documents have to be structured for using in Elastic provided applications such as SIEM [5].

With the help of guidelines such as field naming conventions and building blocks like network objects, a normalized event can be stored in Elasticsearch and correlated with other event types.

2.6 PostgreSQL

PostgreSQL is an open-source object-relational database. Apart from supporting standard Structured Query Language (SQL) operations, it features a variety of built-in basic and advanced (Table 2.3) data types and procedures [41].

Data Type	Description
`inet`	IPv4 or IPv6 host address
`uuid`	universally unique identifier
`xml`	XML data
`json`	textual JSON data
`jsonb`	binary JSON data, decomposed

Table 2.3: Some advanced data types supported by PostgreSQL

2.6.1 Persisting JSON Objects

The two data types for storing JSON, `json` and `jsonb`, are of special interest in this thesis as they will be used to hold data retrieved from the k8s API. While the `json` data type stores JSON in its original textual representation, `jsonb` uses a more efficient way of storing the passed objects [41]. With the `json` data type, the stored object is parsed on each query and is fast to insert. On the other hand, an object parsed as `jsonb` is slower to persist but results in faster queries which can further be accelerated using an index [41]. When further reffering to JSON operations, the `jsonb` data type is in focus. The golang library `pq` [19] and the default Java Database Connectivity (JDBC) driver is used in this thesis to connect to a PostgreSQL database.

2.6.2 Working With JSON Objects

PostgreSQL provides several operations and procedures on JSON data. The SQL/JSON path language retrieves a subset from a specific path and containment or existence op-

erations narrow down to relevant data. Combined with procedures for e.g. building json objects out of regular SQL data or aggregating JSON objects into JSON arrays, PostgreSQL provides a performant feature set working with objects. `jsonb` supports `btree` and `hash` indexing which are useful when checking for equality of two objects [41]. A more powerful option is the Generalized Inverted Index (GIN) [62] which accelerates path operations, including containment queries, by passing the `json_path_ops` parameter or other operations with `json_ops`. An index based on `json_path_ops` is smaller in size compared to a `json_ops` index [41]. A use case for this is analysed in Section 5.8.

2.7 Memcached

Memcached is an open-source distributed, in-memory key-value store for binary data. It allows one or more programs to access the store via the network and set/delete keys and their values [55]. These operations are supplemented by more complex functionality, such as setting an expiry for a specific key. After expiry, the key alongside its value is deleted. A typical use case is caching complex database queries or authenticated users for a web application.

3 Use Case Discussion

3.1 Multi-Tenant Clusters

Securing a multi-tenant k8s with network policies is handled by the CNI plugins. Providing a shared cluster thus involves isolating workloads running by different tenants. Relying on the network policies without validation and constant monitoring can lead to a malfunction or human error resulting in secret leakage. Insight monitors traffic allowing cluster operators to validate the expected path and blocking of network packets.

3.2 Validating Load Balancing

Relying on load balancing, which is not working as expected, can be fatal for production workloads. k8s ingress controllers such as Traefik [34] implement their load balancing algorithms, thus bring in an additional point of failure. When collecting flows on a per-pod basis, connections from ingress controller pods to pods implementing a microservice can be grouped by a service to pod relationship which allows trivial traffic distribution calculation.

3.3 Root Cause Analysis

k8s is a clustered system running on a multitude of nodes. When a connection between two nodes is slightly failing but has not failed yet (e.g. underprovisioned router, review Figure 3.1) can harm communication between two pods using the failing link to communicate with each other. HTTP builds on TCP; thus, dropped packets get retransmitted, extending the total transmission time. A service mesh would notice the failing link as *pod xy is running slow* but does not hint to a connection issue between the two pods. With insight, it is possible to analyze the connection between the pods for each pod individually and show a mismatch between sent and received packets.

3.4 Debugging Uncommon Network Operations

HTTP workloads are well known for working inside k8s. Complex UDP workloads like game engines, on the other hand, tend to behave unexpectedly. Insight can provide details on where a packet is targeted and what NAT was performed, helping to debug connectivity issues and understanding transport layer routing inside the cluster involving services.

3.5 Resulting Requirements

Requirements for the system are extracted from the described use cases and the purpose statement.
The functional requirements set the outline of the technology stack and comprise of:

1. k8s as target system

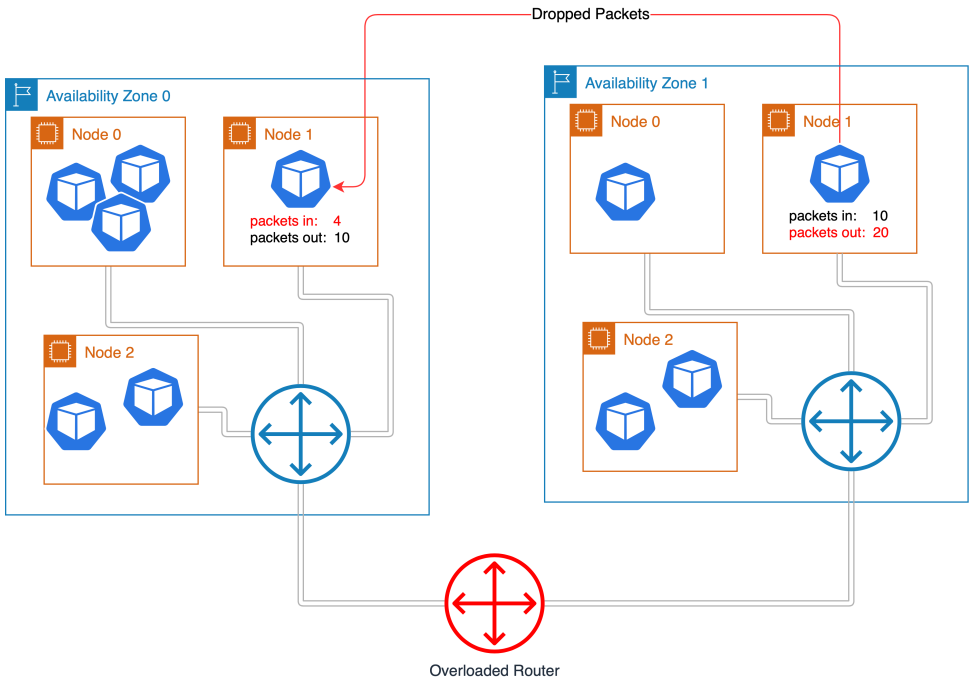

Figure 3.1: Overloaded Router in k8s

2. Elasticsearch as database for storing flows
3. Kibana for visualization
4. Data format follows the ECS

Non-functional requirements are limited to a proof of concept state:

1. Scaling with a large cluster
2. No configuration overhead
3. Selectively monitor network traffic
4. Little to no performance impact
5. Assessing flows on a per-pod basis
6. Unique identifying of flows

4 Platform Design

4.1 High-Level Architecture

The high-level architecture, as shown in Figure 4.1, comprises several components working together as a flow monitoring system.

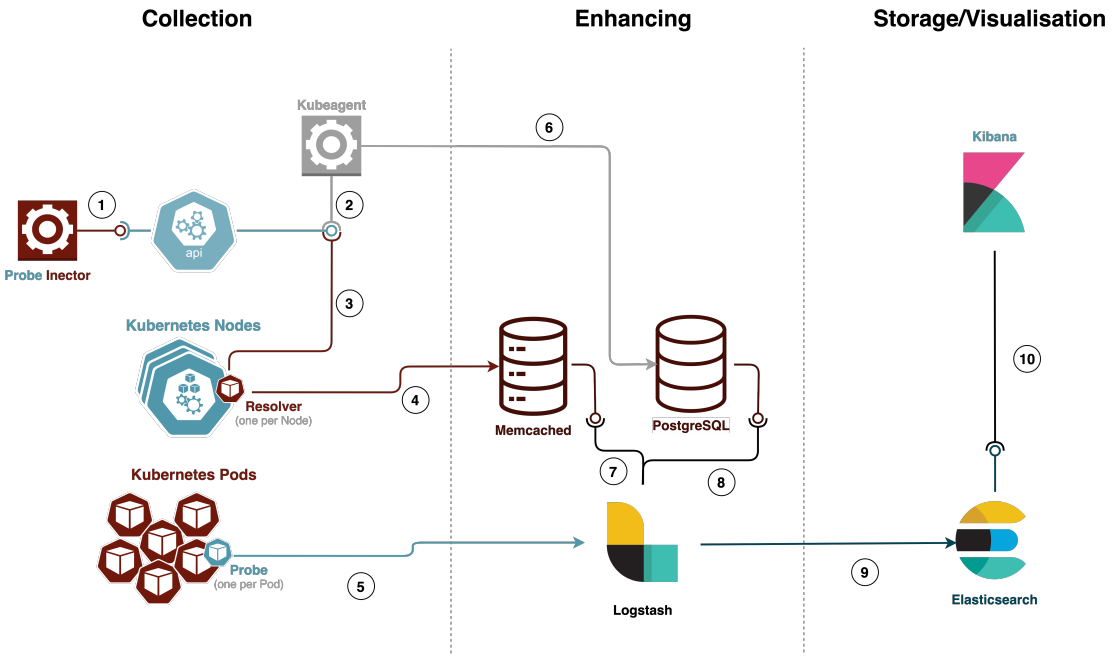

Figure 4.1: High Level Architecture

Interface ID	Description
1	Checks if a probe should be injected to a new pod
2	Pods, Services and Endpoints are constantly monitored
3	Service and Endpoints are constantly monitored
4	NAT operations are pushed to a key-value-store
5	Collected flows are pushed to logstash
6	Pod and Service Metadata state updates are stored in PostgreSQL
7	NAT operations for one flow are retrieved
8	Kubernetes metadata is queried per endpoint IP
9	Processed flows are pushed to Elasticsearch
10	Retrieve flows for visualization

Table 4.1: High Level Architecture Interface Description for Figure 4.1

4.1.1 Data Collection

An essential part of the data collection components are the network **probes** running once per pod. The probe **injector** decides whether a created pod is eligible for getting a network probe injected. They are supplemented by a the **resolver**, a component running on each k8s node detecting NAT operations when a connection to a ClusterIP is established. NAT events are being pushed to memcached, where they expire after the connection is not active anymore and processed. The k8s API is queried by the **kubeagent** which stores the current state of the cluster in a PostgreSQL database. This offloading helps to reduce the load facing the k8s API and allows indexing and relational queries.

4.1.2 Flow Enhancing, Storage and Visualization

A central part of the monitoring platform is a **logstash pipeline**, which receives flow events from the network probes and processes it. During the processing, it checks back with memcached if the connection targets a k8s service and retrieves k8s metadata for source and destination endpoints from PostgreSQL.
After the event passed through the pipeline, it is transferred to Elasticsearch, where it gets persisted in an index. Kibana provides a user interface for accessing and visualizing the data stored in Elasticsearch. The integrated SIEM view [2] provides a powerful query tool for network flows.

4.2 Low-Level Design

4.2.1 Kubeagent

The kubeagent acts as middleware between the k8s API and Logstash. It relies on a PostgreSQL database where the current cluster state is persisted and can be queried using SQL by Logstash, thus implementing a stateful resource for the stateless Logstash. While not implementing any logic, the kubeagent is a central part of the system reducing generated load to the k8s API. After startup, it deletes the current state in the database and starts listening to events from pods, services and endpoints. Received `CREATE`, `UPDATE`, `DELETE` events translate to SQL statements which are executed on the databse.

4.2.2 Network Probe

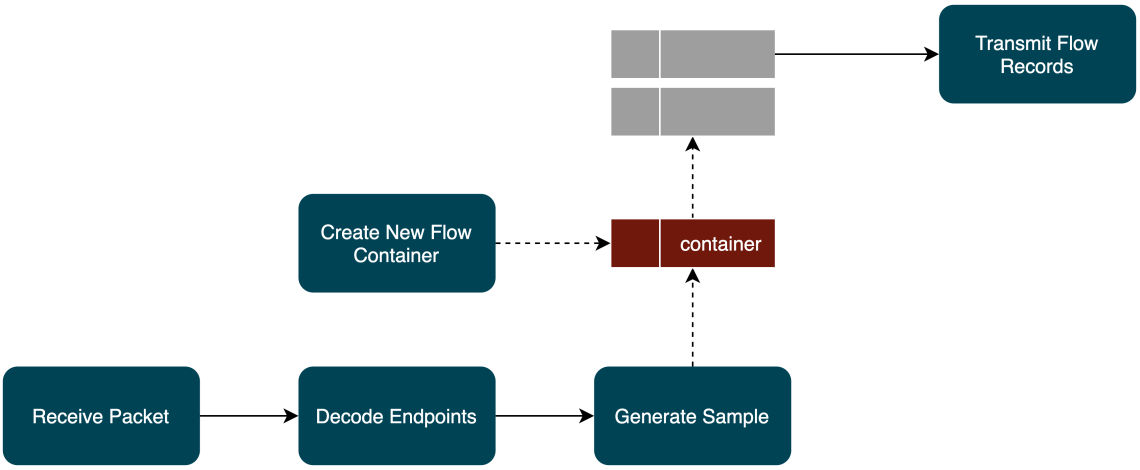

Figure 4.2: Network Probe Internals

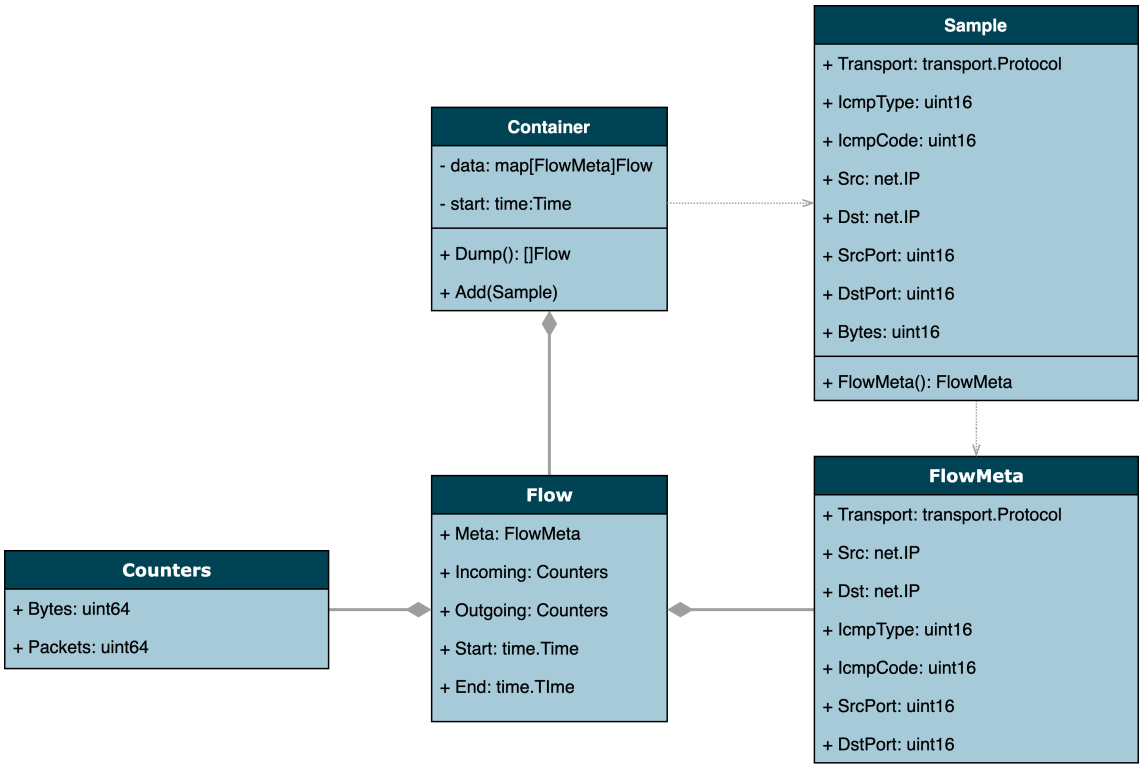

Figure 4.3: Network Probe Flow Container Class Diagrams

The network probe captures packages and records byte and packet counts for each endpoint tuple. After a specified time, it transmits the collected flows to the central Logstash pipeline and starts with an empty table. The execution path divides into three main tasks. The first task periodically creates a new container for flows and pushes the old one to a queue. A second task waits at the other end of the queue and tries to transmit full containers to Logstash using the HTTP protocol and JSON for data definition. In case a container cannot be transmitted, it gets destroyed due to the timing limitations of other components. The final task receives network packets and dispatches a job that decodes the packet's endpoints, computes a flow hash, and updates the current flow container based on the computed sample. In Linux, this is achieved using a RAW socket as described in Section 2.4.

4.2.3 Probe Injector

The network probe needs to be running in the same network namespace as the pod for it to intercept network traffic. This can be achieved by running the network probe as an additional container inside the pod. Instead of manually adding the container to the pod definition, the probe injector uses a k8s mutating webhook for resource patching. Automating the container injection allows more granular control of where traffic is intercepted. For the proof of concept, probe injection is determined on a namespace annotation. As pods are usually managed by deployments or stateful sets, the namespace is not passed directly with the webhook request but is derived from the parent. A second API request is therefore required to get the target namespace (Figure 4.4).

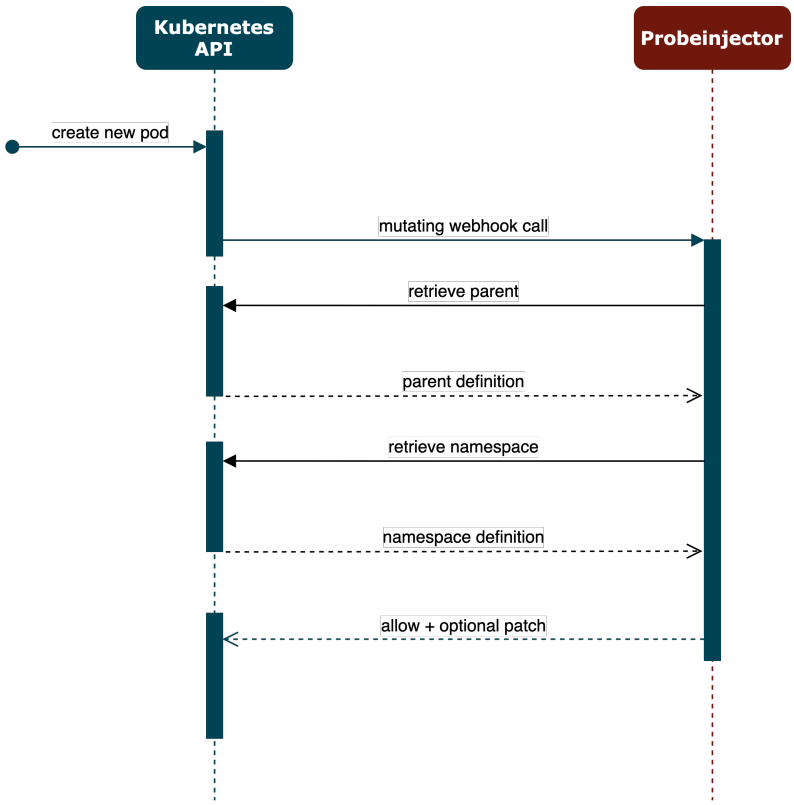

Figure 4.4: Network Probe Injection Using K8s Webhooks

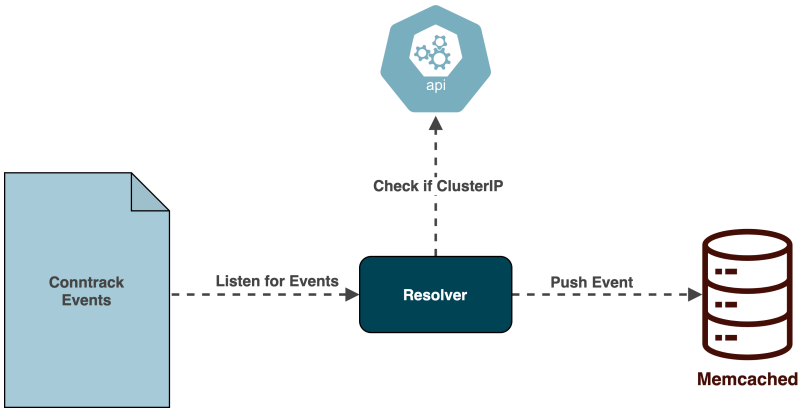

Figure 4.5: Components Of The Resolver

4.2.4 Resolver

As discussed in Section 2.3.10, ClusterIP load balancing inside k8s is tracked in the Linux conntrack table. In k8s, pods are not addressed individual but rather through a service abstraction. Without further processing, flows captured by the network probe are of the kind pod to service, and do not fulfill the requirement of intercepting the pod to pod communication. The resolver listens to events conntrack table, determines if the target IP is of kind ClusterIP and stores a corrected destination in Memcached as seen in Figure 4.5. As seen in Equation (4.1), the additional information of the NAT operation tracked in the conntrack table allows the mapping of service calls to a specific target pod.

$$\left\langle \left(IP_{src}, P_{src}\right), \left(IP_{service}, P_{service}\right), \left(IP_{pod}, P_{pod}\right) \right\rangle \implies \left\langle \left(IP_{src}, P_{src}\right), \left(IP_{pod}, P_{pod}\right) \right\rangle \tag{4.1}$$

5 Implementation

All components, excluding the central Logstash pipeline, are implemented in Go, also known as Golang [30]. With a syntax similar to C, it is an imperative, statically compiled programming language. The builtin concurrency concept relies on `goroutines`. A goroutine does not fit the description of a coroutine or a thread but can be interpreted as *very* light thread starting with a small stack [8].
Goroutines are distributed over multiple operating system threads improving multi-core performance [8]. Following the ideology *"Do not communicate by sharing memory; instead, share memory by communicating"* [8], Go introduces `channels` as a way to transfer data between concurrently running processes instead of using shared memory. Implementing Communicating Sequential Processes (CSP) [43], a channel acts like a traditional Unix pipe and is used to interconnect goroutines. The keyword `go` in front of a function call dispatches a new goroutine.

While Go is not an object-oriented programming language, it provides an abstraction using `Interfaces`, which are implemented by `structs` and their methods. The source code is divided into modules that contain packages. A module is commonly resolved by its source control path, for this project `module "github.com/xvzf/insight/`. Packages itself derive from a folder structure and consist of one or multiple files starting with a typical package tag, e.g. `package "flow"`. A package has exported and contained symbols, controlled by the letter case. If a function or variable starts with an uppercase character, it is exported; otherwise, it is package exclusive.
With readability in mind, Go omits redundant parentheses and semicolons. Compared to C where many coding-styles, such as the Linux kernel coding style [15], Go strives for a unified coding style. One of the utilities shipped with Go is `gofmt` that ensures conform formatting of files and packages.

Static compilation of Golang and the implied lack of library dependencies at runtime makes it ideal as a programming language of choice in container-based systems. Binaries always contain the Go runtime, which *"implements garbage collection, concurrency, stack management, and other critical features of the Go language"* [10].

5.1 Compatibility & Used Libraries

With Kubernetes (k8s), there are no long term support releases. Following a release cycle of roughly three months, the latest three k8s versions get maintenance updates. This thesis targets k8s `1.17` (latest release as of march 2020). The previous `1.16` release introduces inital dual stack support (IPv4 alongside IPv6) changing the datastructure for pod states. The proof of concept should therefore be compatible with version `1.16` and higher and is validated against the k8s distribution `k3s` [56] version `1.17.4` in combination with Docker `19.03` [46] as container runtime.

Elasticsearch, Kibana and Logstash releases changed during the development process. The system is validated against `7.6.1`. First introduced with this release is support for

Library	Version	Information
`go-cmp` [18]	`v0.4.0`	Deep compare operations used for testing
`gomemcache` [11]	`a41fca850d0b`	Client library for Memcached
`client-go` [14]	`v0.17.0`	K8s API client
`logrus` [21]	`v1.4.2`	Logging framework
`gopacket` [31]	`v1.1.17`	Network interception and analysis
`pq` [19]	`v1.3.0`	SQL driver for PostgreSQL

Table 5.1: List of used libraries

Kibana Query Language (KQL) queries targeting nested fields.

Introduced microservices require Golang `1.13` for compilation and rely on the Alpine Linux [13] docker image `3.11` as runtime target. A list of libraries used can be retrieved from Table 5.1.

5.2 Automated Testing, Compilation And Delivery

For ensuring code quality and automation of the deployment to a testing target, DroneCI [35] is used as a continuous integration and delivery platform. It runs on k8s and executes each step as an individual container.

The pipeline (Appendix B), triggered by a push event on the project's git repository, consists of three different stages stacked on top of each other that involve testing, compilation, and an optional deployment step.

Unit tests are implemented using the test framework shipped with Golang. During the testing stage, the computed test coverage ensures establishing a high test quality for critical functions.

Every component is delivered as a container image and its building process is described using a Dockerfile. While there are slight differences for each component, the building process divides into two distinct phases. Starting with the first line, a container image with a full Golang toolchain is selected, which is later used to build a static binary (line 8,9). The final base image is selected in line 13. The statically compiled and therefore dependeny less, binary is copied from the earlier build stage (line 15). The resulting container image is small in size and contains only runtime dependencies.

Every introduced component alongside the databases is deployed inside a k8s cluster. The multitude of resources and dependencies is managed by helm [12], a package manager for k8s based on templating resources with environment-specific values.
When pushing to a predefined branch, DroneCI executes a command which updates a staging environment using a helm command.

5.3 Unified Code Base & Release Management

Source code for all components is tracked in a git repository (https://github.com/xvzf/insight.git). Every commit runs through a test suite and, if passed, is delivered by a container image.

```
1  FROM golang:1.13-alpine as builder
2
3  WORKDIR /app
4
5  copy . .
6
7  # Build application
8  RUN go get -d -v ./...
9  RUN go build cmd/kubeagent/kubeagent.go
10
11
12 # Generate a smaller docker image without build dependencies
13 FROM alpine:3.11
14
15 COPY --from=builder /app/kubeagent /kubeagent
16 ENTRYPOINT ["/kubeagent"]
```

Listing 5.1: Two-staged Dockerfile

In case a commit is reviewed as stable, a git tag marks it as *release*.

5.4 Network Flows

A network flow as introduced in Section 2.1 is modelled based on the class diagram in Figure 4.3. Listing 5.2 shows its representation in Go, based on structs. The `Meta` struct holds endpoint information comprised of source and destination IP, the transport protocol and detailed information on ports (TCP, UDP) or protocol types and codes (ICMP, ICMPv6).

5.4.1 Unique Identifier

The lifetime of a network flow can exceed the sample period of the network probe. A unique identifier for endpoints of a flow allows tracking over a multitude of samples. Ensuring a matching identifier for swapped endpoints, implemented by a hash collision as introduced in Section 2.1.1, further allows identifying incoming and outgoing traffic.

The flow hashing used in this thesis follows guidelines of the CommunityID specifications [45], which is already supported by Kibana [2, 45].

The implementation in Golang starts ordering the endpoints using a lexicographical comparison between the byte representation of each endpoint IPs (Listing 5.3). After ordering, endpoints and protocol information are passed to a SHA1 [42] hash function. The base64 encoding [49] of the computed hash prefixed with `1:` is the resulting flow hash and unique identifier.

5.4.2 ICMP And ICMPv6 Flow Hashing

While the presented flow hash algorithm supports TCP and UDP as port based protocols, the Internet Control Message Protocol (ICMP) and Internet Control Message Protocol for

```go
package "flow"

// Counters contains flow counters
type Counters struct {
  Bytes   uint64
  Packets uint64
}

// Meta contains flow metadata
type Meta struct {
  Transport protos.ProtocolType
  Src       net.IP
  Dst       net.IP
  IcmpType  uint16
  IcmpCode  uint16
  DstPort   uint16
  SrcPort   uint16
}

// Flow contains flow data
type Flow struct {
  Meta        Meta      // Flow Src/Dst & Protocol Information
  Incoming    Counters  // Incoming counters
  Outgoing    Counters  // Outgoing counters
  CommunityID string    // CommunityID
  Start       time.Time // Start time
  End         time.Time // Stop time
}
```

Listing 5.2: Flow representation in Go (`pkg/flow/flow.go`)

```go
func extractTuple(f flow.Meta) ([]byte, []byte, uint16, uint16) {
  // ...
  cmp := bytes.Compare(f.Src, f.Dst)
  if cmp < 0 || (cmp == 0 && f.SrcPort < f.DstPort) {
    return rawIP(f.Src), rawIP(f.Dst), f.SrcPort, f.DstPort
  }
  return rawIP(f.Dst), rawIP(f.Src), f.DstPort, f.SrcPort
}
```

Listing 5.3: Endpoint ordering (`pkg/flow/communityid/communityid_hasher.go`)

```go
  func (ch *hasher) Hash(f flow.Meta) string {
    ip0, ip1, p0, p1 := extractTuple(f)

    h := crypto.SHA1.New()
    h.Write(ch.seed[:])
    binary.Write(h, binary.BigEndian, ip0)
    binary.Write(h, binary.BigEndian, ip1)
    h.Write([]byte{byte(f.Transport), 0})
    binary.Write(h, binary.BigEndian, p0)
    binary.Write(h, binary.BigEndian, p1)

    return "1:" + base64.StdEncoding.EncodeToString(h.Sum(nil))
  }
```

Listing 5.4: Network flow hashing (`pkg/flow/communityid/communityid_hasher.go`)

the Internet Protocol Version 6 (ICMPv6) use type and code. The CommunityID specifications suggest mapping the transmitted ICMP/ICMPv6 type to a port tuple and omitting the code. Following Table 5.2 as an example, a simple lookup operation determines a port tuple consolidated of the original type and its counterpart. It is noteworthy that not every ICMP/ICMPv6 type has a counterpart. In this case, the type and code are used as source and destination port. Refer to Listing 5.5 as lookup operation.

5.4.3 Source Detection

Detecting the initiator of a connection is trivial for ICMP and ICMPv6 as there are specific types for requests and responses. With the assumption that the TCP connection estab-

Type	Counterpart
ICMPv6TypeEchoReply	ICMPv6TypeEchoRequest
ICMPv6TypeRouterSolicitation	ICMPv6TypeRouterAdvertisement
ICMPv6TypeNeighborSolicitation	ICMPv6TypeNeighborAdvertisement
ICMPv6TypeEchoRequest	ICMPv6TypeEchoReply
ICMPv6TypeRouterAdvertisement	ICMPv6TypeRouterSolicitation
ICMPv6TypeNeighborAdvertisement	ICMPv6TypeNeighborSolicitation

Table 5.2: Selection of ICMPv6 types and the counterparts (`pkg/flow/common/icmp.go`)

```
    func GetICMPv6PortEquivalents(t, c uint16) (uint16, uint16, bool) {
      if v, ok := icmpV6Equiv[t]; ok {
        return t, v, false // Is not one-way
      }
      return t, c, true // Is one-way
    }
```

Listing 5.5: ICMPv6 to port equivalent mapping

```
func (m Meta) WithCorrectedSource() Meta {
    switch m.Transport {
    case protos.TCP, protos.UDP:
        if m.DstPort > m.SrcPort {
            m.Src, m.SrcPort, m.Dst, m.DstPort = m.Dst, m.DstPort, m.Src,
            ↪  m.SrcPort
        }
    case protos.ICMP4:
        if v, ok := common.GetICMPv4RequestType(m.IcmpType); ok {
            m.Src, m.Dst = m.Dst, m.Src
            m.IcmpType = v
        }
    case protos.ICMP6:
        if v, ok := common.GetICMPv6RequestType(m.IcmpType); ok {
            m.Src, m.Dst = m.Dst, m.Src
            m.IcmpType = v
        }
    }
    return m
}
```

Listing 5.6: Connection source detection

lishment was not captured, UDP and TCP packets are going in one direction or the other. Based on the Linux kernel default settings for ephermal ports `ip_local_port_range` [16] starting at 32768, it is assumed that the lower value port is the destination and the higher value port the source (Listing 5.6).

5.5 Network Probe

5.5.1 Intercepting Network Traffic

The network probe intercepts network traffic using a `PACKET` socket as introduced in Section 2.4. Gopacket [31] abstracts raw socket operations and provides a channel providing received packets. Referring to Listing 5.7, the device is opened in *live* mode with a maximum packet length of $9,038$ bytes supporting jumbo frames [36].

The function creating a channel with captured packets further optimizes performance by configuring gopacket for being lazy, thus not parsing the packet unless requested and not copying the packet by using pointer references instead.

```go
func Open(device string) (Capturer, error) {
  handle, err := pcap.OpenLive(device, 9038, true, pcap.BlockForever)
  // ...
}

func (p *pcapHandle) Packets() chan gopacket.Packet {
    source := gopacket.NewPacketSource(p.handle, p.handle.LinkType())
    source.Lazy = true
    source.NoCopy = true
    return source.Packets()
}
```

Listing 5.7: Intercepting network traffic `pkg/capture/capture.go`

```go
type data struct {
    sync.Mutex
    flows map[string]*flow.Flow
}

type container struct {
    data   data
    hasher communityid.Hasher
    start  time.Time
}
```

Listing 5.8: Flow container datastructure

5.5.2 Extracting A Sample

When a packet comes in, its endpoints need to be extracted. Helper functions from the gopacket library help by extracting IP endpoints as well as transport protocol endpoints.

The aforementioned `flow.Meta` struct is populated accordingly and expanded by the packets byte count to a flow sample.

5.5.3 Aggregating Packets To Flows

Multiple flow samples captured over a defined timeframe aggregate to a network flow. A container keeps track of current flow statistics and is realized by a hash map. The CommunityId builds the key. The Value is a pointer to a `flow.Flow` object. A mutex protects the data structure as multiple goroutines handle incoming flow samples at the same time. After a new sample hits the processing pipeline, its CommunityID is computed. If the CommunityID is already present in the container, its flow object is updated accordingly. Otherwise, a new `flow.Flow` is created.

5.5.4 Generating A Flow Event

In a specified interval, the current container is replaced with a new one and stored flows are exported. A function iterates over values in the hash map and creates a new flow event (Appendix A) following the Elastic Common Scheme (ECS). Exported flow events are consolidated in an array and transfered to Logstash using a HTTP `POST` request.

5.6 Kubeagent

5.6.1 PostgreSQL In Memory Database

The current k8s state is persisted in a PostgreSQL database, more specifically in a field of type `jsonb`. There is no need to persist the state to disk as it is destroyed and repopulated whenever the kubeagent restarts. To further increase database performance, the data directory is mounted as ramdisk. After database startup, a script initializes tables and creates required indices (Appendix C).

5.6.2 Kubernetes To SQL Middleware

The functionality of the kubeagent is simple and consists of watching the k8s API and transferring events to the PostgreSQL database. The k8s client library for Go supports the watch API. Captured events for pods, services and endpoints are passed to a channel. Running inside the cluster, credentials from a service account are used to authenticate against the API (Listing 5.9). Events translate to `INSERT`, `UPDATE` and `DELETE` SQL statements

```
1  func main() {
2      // ...
3      config, err := rest.InClusterConfig()
4      // ...
5      clientset, err := kubernetes.NewForConfig(config)
6      // ...
7      podWatcher, err :=
       ↪  clientset.CoreV1().Pods("").Watch(metav1.ListOptions{})
8      // ...
9      var wg sync.WaitGroup
10     store := kubestatestore.New(os.Getenv("CONN_STRING"))
11     // Create a goroutine for the pod & service watcher
12     for _, watcher := range []watch.Interface{podWatcher, svcWatcher,
       ↪  endpointsWatcher} {
13       wg.Add(1)
14       // Capture incoming events and pass them to the PodStateStore
15       go func(w watch.Interface) {
16         defer wg.Done()
17         for e := range w.ResultChan() {
18           // Pass event to the store
19           go store.HandleUpdate(e)
20         }
21       }(watcher)
22     }
23     wg.Wait()
24 }
```

Listing 5.9: Watching the k8s API

which update the k8s state in PostgreSQL.

5.7 Resolver

5.7.1 Tapping The Conntrack Table

The command `conntrack -E` outputs connection tracking events from the Linux kernel. A regular expression (Listing 5.10) parses the output accordingly into a conntrack event as noted in Equation (5.1).

```
1 //  "[UPDATE] tcp      6 86400 ESTABLISHED src=10.42.1.21 dst=10.43.15.37
  ↪  sport=49594 dport=9200 src=10.42.0.49 dst=10.42.1.21 sport=9200
  ↪  dport=49594 [ASSURED]"
2 const ipPortRegex = `src=(.*) dst=(.*) sport=(\d+) dport=(\d+)`
3 var extractor =
  ↪  regexp.MustCompile(`\s*\[(NEW|UPDATE|DESTROY)\]\s+(tcp|udp).*` +
  ↪  ipPortRegex + `.*` + ipPortRegex + `.*`)
```

Listing 5.10: Regular expression parsing conntrack events

$$\Big(E_{type}, E_{protocol}; (IP_{src}, IP_{dst}, P_{src}, P_{dst}) \rightarrow (IP_{src}, IP_{dst}, P_{src}, P_{dst})\Big) \tag{5.1}$$

E_{type} distincts between `NEW`, `UPDATE` and `DESTROY` events while the $E_{protocol}$ specifies the network protocol. The following tuple to tuple mapping denotes original connection details and a possible masqueraded connection.

5.7.2 Filtering For Kube-Proxy Events

The Linux kernel keeps track of every network connection, resulting in a large number of events. The purpose of the resolver is to map a request facing a k8s service to a pod. The resolver retrieves all k8s ClusterIPs by interacting with the watch API for services. If a connection faces a ClusterIP, the resolver computes CommunityIDs of its original and masqueraded connections and creates a new key-value entry in Memcached (Equation (5.2)).

$$ORIG_{community\_id} \rightarrow (MASQ_{community\_id}, IP_{new\_dst}, P_{new\_dst}) \tag{5.2}$$

Using the original CommunityID as the key allows efficient lookup operations. Setting a timeout for the key, depending on its event, helps to keep track of events without overflowing the key-value store. After registering a new event, a timeout of $3,600$ seconds is set. This ensures tracking of e.g. long TCP streams and avoids an orphaned entry when a node fails and cannot delete its keys. After a connection is considered closed, a timeout of 20 seconds is set, preventing a deletion before the network probes sample timeframe ends.

5.8 IP To Kubernetes Object Mapping

Due to security constraints, not every pod is allowed to retrieve the k8s state from the API. Matching IP addresses to its k8s objects is therefore offloaded to the PostgreSQL database holding the k8s state. SQL queries from Logstash retrieve pod and service information for a specific IP address.

Listing 5.11 shows a SQL statement which retrieves pod labels for the IP `10.42.0.12`. The JSON containment operation limits the result set to pods having the IP `10.42.0.12`

```
select
  p.definition #> '{"metadata", "labels"}' as pod_labels
from pods p
  where p.definition #> '{"status", "podIPs"}' @> '[{"ip":
  ↪  "10.42.0.12"}]'::jsonb
```

Listing 5.11: SQL query for pod labels

```
select
  s.definition #> '{"metadata", labels}' as service_labels
from services s
  join endpoints e on (s.name, s.namespace) = (e.name, e.namespace)
where e.definition #> '{"subsets"}' @> '[{"addresses": [{"ip":
↪  "10.42.0.12"}]}]'::jsonb
```

Listing 5.12: SQL query for service labels

assigned. The selection `p.definition #> '{"metadata", "labels"}'` locates the pod labels.

Retrieving services for a specific IP requires an additional JOIN operation as target information is stored inside the related service endpoint and not the service object itself (Listing 5.12).

One IP can map to multiple pods and nodes when e.g. multiple pods on one node run in the network namespace of the host. In case the result set is empty, the IP belongs to an already deleted pod or is not contained in the pod or service subnet.

5.9 Logstash Pipeline

The HTTP input plugin for Logstash takes in network flow events on port 8080. The filter section starts with removing fields generated by the input plugin. In a second step, a request to Memcached checks for a potential pod to service flow. Upon a positive lookup, an additional filter step updates destination IP, port and CommunityID. Afterwards, the `jdbc_streaming` plugin executes a query on the k8s state database extracting, pod and service metadata for source and destination IPs. Performance improvements are made by combining pod and service queries as described in Section 5.8 and caches the results for ten seconds. As discussed in Section 5.8, it is possible for one IP to map to multiple pods or services. Generating e.g. a donut graph in Kibana requires a terms filter which does not support nested fields. Therefore, the first metadata entry for pods and services is copied to an object typed field.

Logstash does not support the `jsonb` data type of PostgreSQL therefore the JSON is requested as text and afterwards parsed.

The pipeline ends with an Elasticsearch output plugin. Logstash manages an index template (Appendix E) and creates one document per event in a date based index. The index template specifies the number of shards (3) and replicas (0). Depending on the system utilization and Elasticsearch cluster layout, these settings have to be tuned. Reflecting the ECS, textfields map to the `keyword` type by default.

```yaml
apiVersion: admissionregistration.k8s.io/v1beta1
kind: MutatingWebhookConfiguration
metadata:
  name: {{ include "probeinject.fullname" . }}
  labels:
    {{- include "probeinject.labels" . | nindent 4 }}
webhooks:
  - name: webhook-service.default.svc
    failurePolicy: Fail
    clientConfig:
      service:
        name: {{ include "probeinject.fullname" . }}
        namespace: {{ .Release.Name }}
        path: "/inject"
      caBundle: {{ .Values.secret.caBundle }}
    rules:
      - operations: [ "CREATE" ]
        apiGroups: [""]
        apiVersions: ["v1"]
        resources: ["pods"]
```

Listing 5.13: Template for the mutating webhook

5.10 Probe Injector

Network probes are selectively injected before pod creation. For the proof of concept, the namespace annotation `insight` decides whether a sidecar container is injected or not. The probeinjector is a HTTPS server providing an endpoint for a k8s `MutatingWebhookConfiguration` (Listing 5.14). Upon namespace validation, it returns a JSON patch [4] adding the network probe to the container array (Listing 5.14).

5.11 Visualization using Kibana

As the data format is compatible with the Elastic Common Scheme (ECS), the Kibana SIEM view visualizes network flows inside the cluster. After creating the index pattern `insight-1.0.0-*` and adding `siem:defaultIndex` under the advanced settings, there are two populated views in the SIEM app. The event view (Figure 5.1) in combination with the timeline (Figure 5.2) allows flow tracing based on k8s metadata and connection details while the network overview (Figure 5.3) gives a global summarization of network activity.

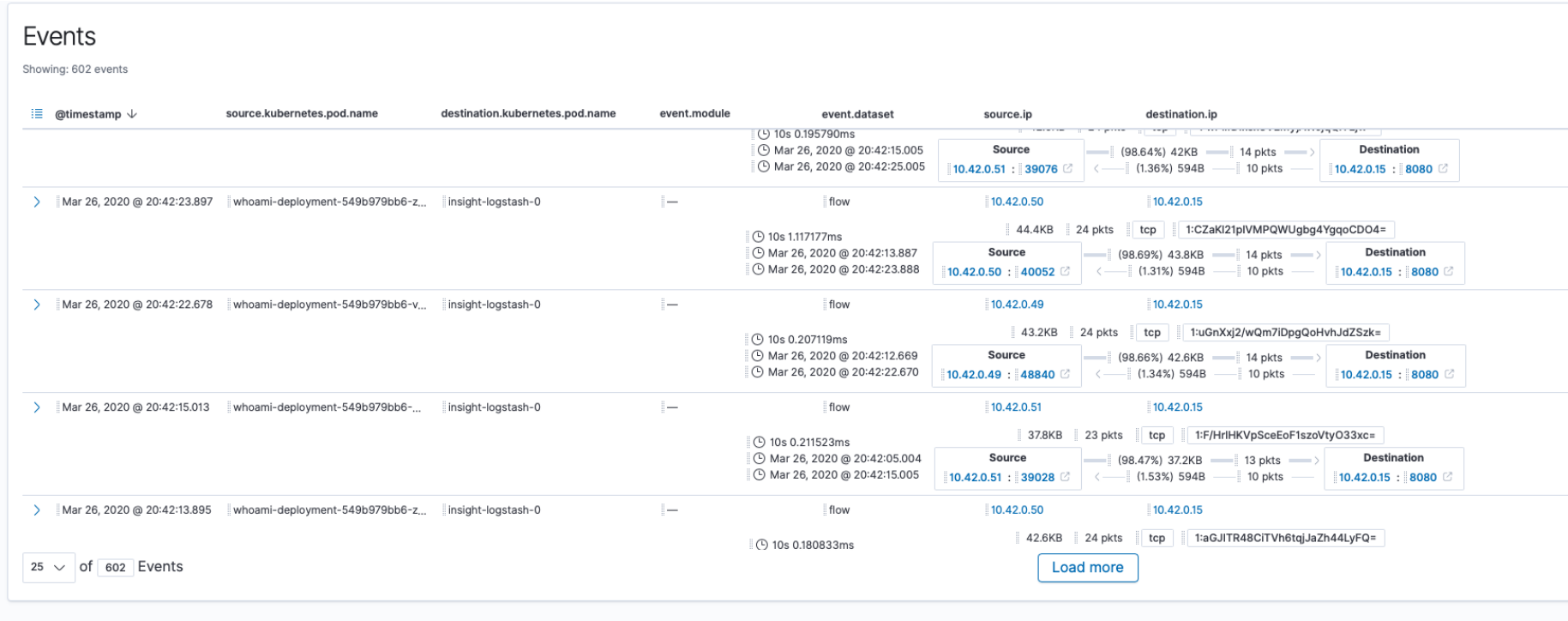

Figure 5.1: Kibana SIEM event view

```
{
  "op":"add",
  "path":"/spec/containers/-",
  "value":{
    "name":"insight-sidecar-probe",
    "image":"quay.io/xvzf/insight:v1.1.6",
    "env":[{
          "name":"LOGSTASH",
          "value":"http://logstash.insight.svc.cluster.local:8080/"
        }],
      "resources":{}
  }
}
```

Listing 5.14: JSON patch injecting the network probe

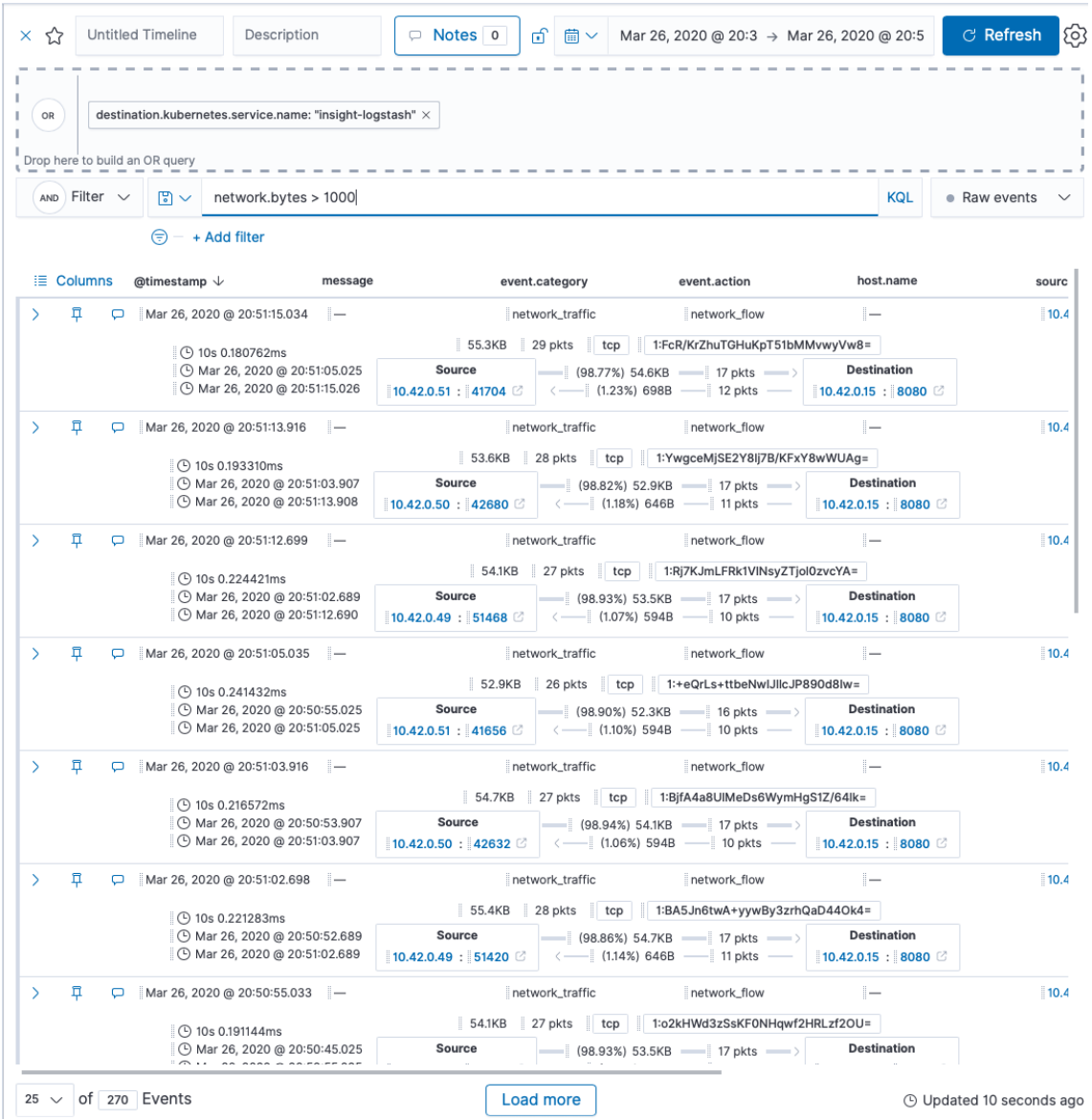

Figure 5.2: Kibana SIEM timeline

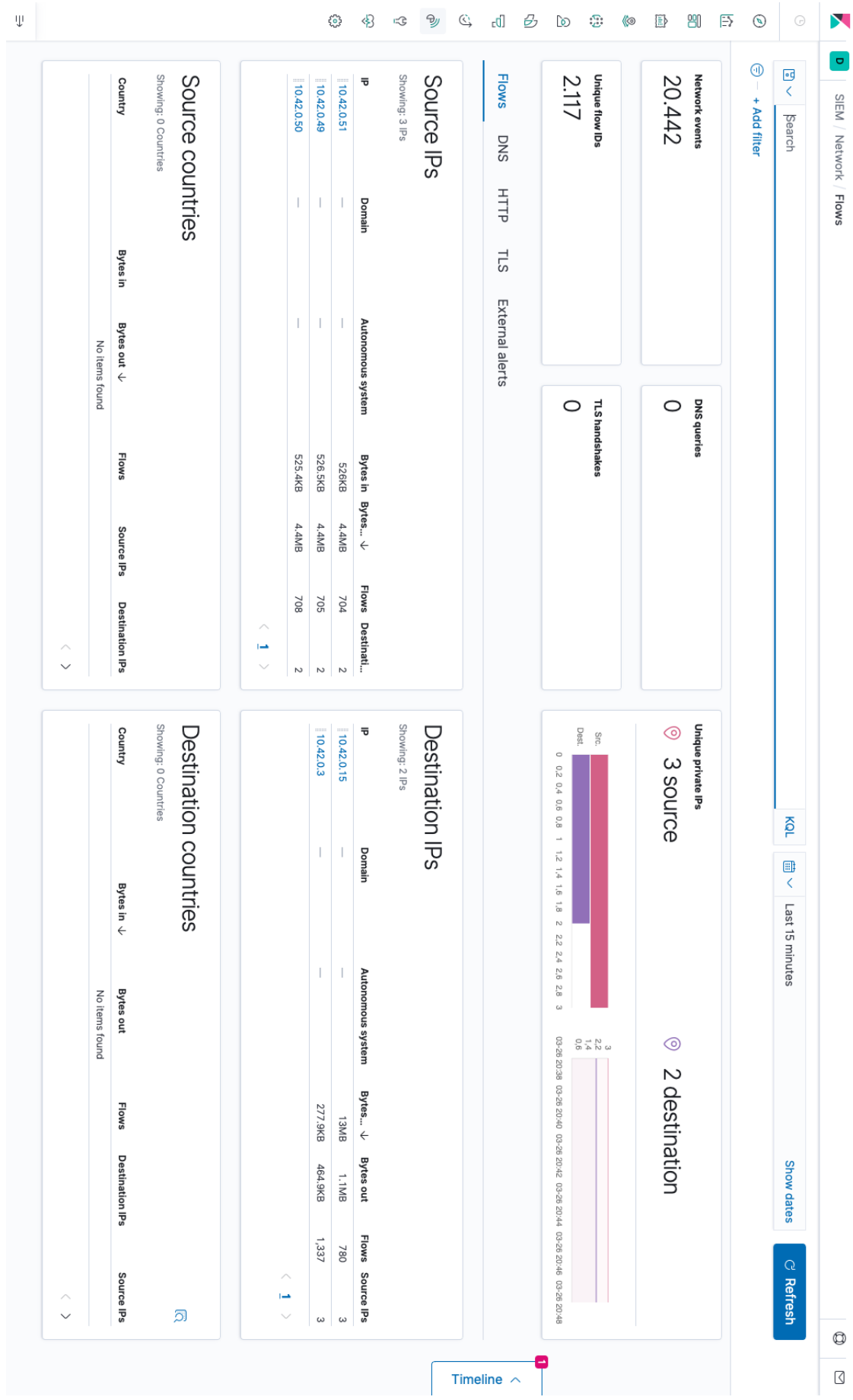

Figure 5.3: Kibana SIEM network view

6 Evaluation

The proof of concept is analyzed and tested against performance and a potential benefit in scenarios described in Chapter 3.

6.1 Platform Scaling And Performance Discussion

6.1.1 Component Scaling

The flow monitoring platform can be split into three independent sections: flow collection, external databases and ELK-Stack. The ELK-Stack is built to be scalable out of the box; this is not in focus. While running in the cluster, its components can be exclusively limited to a specified group of nodes and do not impact the cluster operability status. With 5000 nodes supported in a single k8s [26], adding nodes dedicated to the ELK-Stack components is not considered relevant.

The Logstash pipeline is stateless and gathers its data from Memcached and PostgreSQL. It can be scaled horizontally, depending on the current system load. Caching queries from PostgreSQL helps to reduce SQL queries, thus improves larger scaled deployments.

While Memcached is distributed by design and can be scaled across multiple instances, the number of key-value pairs increases with everything interacting with a service, be it a deployment or a new node.

The PostgreSQL database hit by possibly multiple Logstash pipelines is not distributed by default. As the majority of queries are select statements, read-only mirrors can add support for a more significant number of clients.

6.1.2 Resolver performance issues

The resolver relies on the `conntrack` executable to interact with the Linux kernel. While this is a working solution, executing a second program and parsing its output is not efficient and error-prone. During the development and validation phase, several issues occured which were caused by the Netlink receive buffer being to small. Increasing the buffer results in a more reliable execution, but during high load sequences, the resolver can crash as reaction to `conntrack -E` crashing in the background. A potential fix for this behavior is implementing the Netlink socket for conntrack events directly in Go, which allows more robust error handling.

6.1.3 Network Probe performance impact

The network probe intercepts traffic in a non-blocking way. If a buffer is full or the capturing channel cannot keep up with the transmission speed, the network probe misses the packets but not the software communicating. Operations of the network probe are transparent to the application running inside the pod. The only performance penalty is generated CPU load, memory consumption and the bandwidth required to transfer flow data.

6.2 Security Concerns

As discussed in Section 2.4, intercepting network traffic requires a `PACKET` socket which itself depends on the `CAP_NET_RAW` capability. While the capability allows reading packets, it also allows the injection of arbitrary packets, which allows attackers to perform man in the middle attacks. Using a k8s security context, adding the capability just for the probe containers in a pod limits the attack vector.

In managed k8s clusters, especially in the corporate environments processing sensitive information, pod security policies prohibit the injection of capabilities rendering the platform useless.

Insight does not implement an access control mechanism, thus does not distinguish between a developer who has access to the integration environment and an administrator who runs the production environment. Both user groups have access to the full flow database. Another concern is the general data protection regulation law when public IP addresses show up in the network flows. However, source and destination IPs can be omitted as k8s labels and the CommunityID is sufficient to query the dataset.

6.3 Pratical Testing

6.3.1 Test Setup

During development, validation and testing a multitude of k8s was used, ranging from multi-master clusters to a single node cluster on a notebook. Further presented visualizations and metrics were generated on a multi node k8s cluster in the AWS cloud [44] managed by the kubernetes operator `kops` [26]. Version `1.17.3` with Docker `19.3.4` as container runtime is distributed on three worker nodes (`t3a.xlarge` instance type, 4 CPU, 16GB system memory) and one master node (`t3a.medium` instance type, 2 CPU, 8GB system memory). The CPU model is an AMD EPYC 7571 clocked to 2.2 GHz.
The ingress controller traefik [34] scales to four instances and is exposed via an AWS network load balancer. As a test target, two deployments with 6 instances of a microservice returning request information is used (Appendix F). They are deployed to different namespaces where one has network probe injection enabled and the other one does not.
A dedicated instance in the same subnet as the k8s cluster is used for benchmarking.

6.3.2 Measuring The Performance Impact Of The Network Probe

Using the !pache benchmark tool `ab` [40], the response time of $100,000$ request with 60 in parallel is measured for both deployments. Alongside, CPU and network utilization is measured.

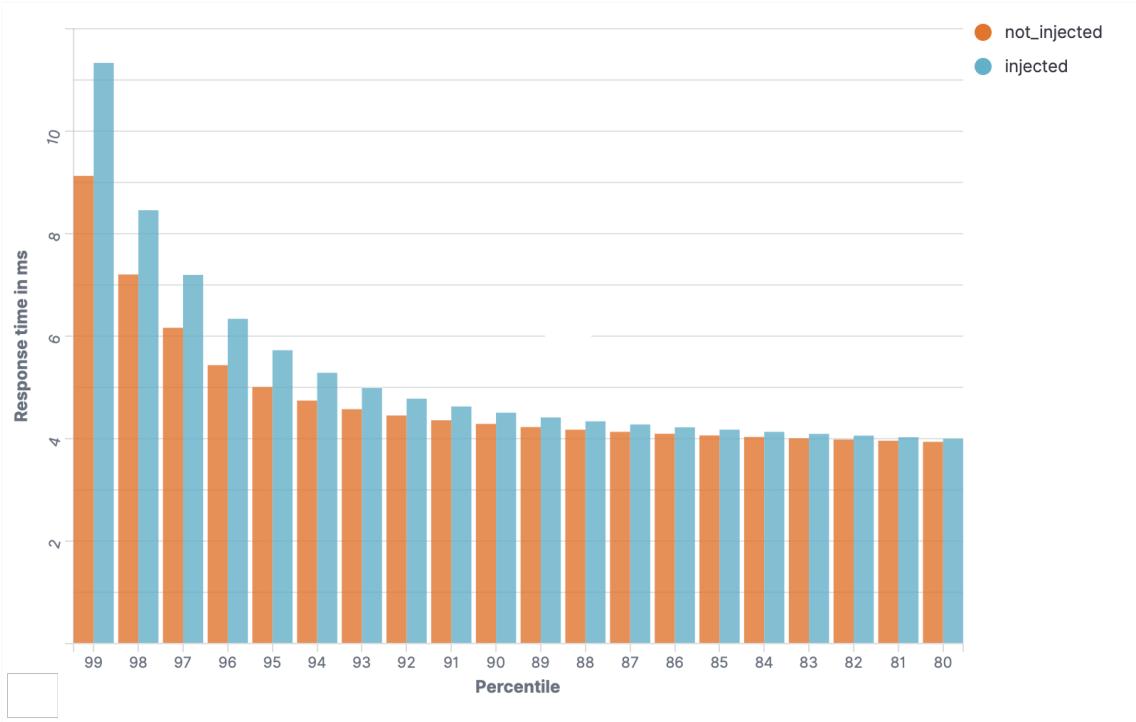

Figure 6.1: Performance impact of the network probe

The resulting response times (Figure 6.1) show an impact on the response time worsening it by two milliseconds for the 99 percentile. Starting with around 90 percent of the request, the response time difference becomes neglicable. Reviewing the CPU utilization during the benchmark (Figure 6.2) shows the network probes require up to 0.15 CPU cores.

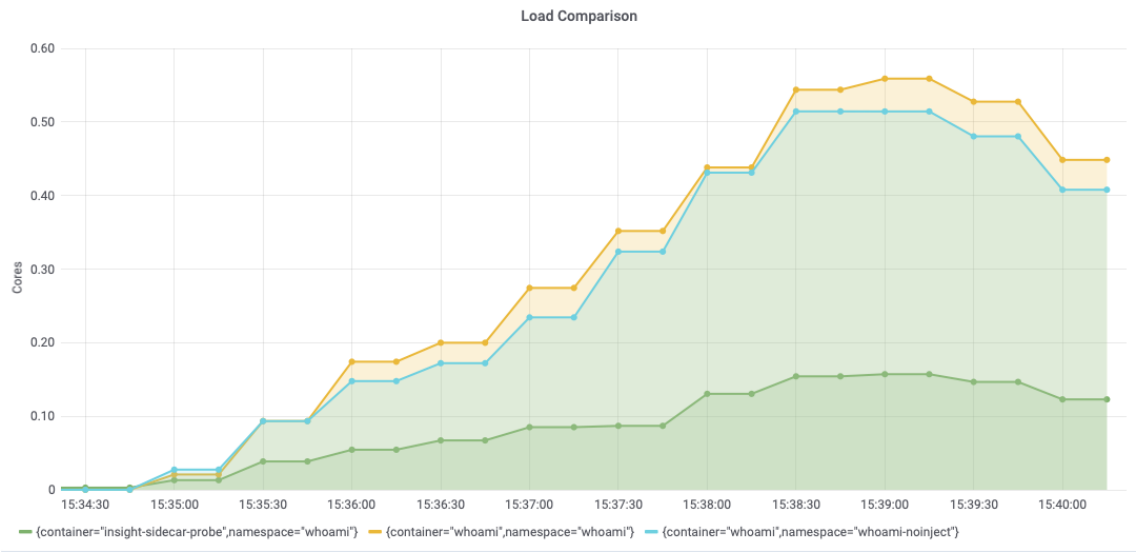

Figure 6.2: CPU utilization cummulated by container and namespace

6.3.3 Load Balancer Validation

Captured flows allow complex visualizations such as validating ingress load balancing alongside internal cluster load balancing. In this test scenario, the four traefik ingress controller instances are hit by an external load balancer and forward request to the sample

application thus relying on internal cluster load balancing. In Figure 6.3, a pie chart visualizes the distribution of network traffic (based on the total byte count) to the microservice and sets it to relation with the ingress controller instances.

6.3.4 Comparison To Metric Based Monitoring

Monitoring resources inside and around the cluster is a key part of every operations team. With k8s, an often seen approach is Prometheus [32] as time series database combined with Grafana [51] as user facing visualization frontend. While supporting many metrics such as CPU and memory utilization, metrics for incoming and outgoing network traffic. Comparing the resulting graph (Figure 6.4) to a visualization generated by Insight (Figure 6.5), it does only show how much network traffic was present at a specific time but not where it was directed.

The increased grade of detail allows precise performance tuning and helps identifying bottlenecks and scaling issues.

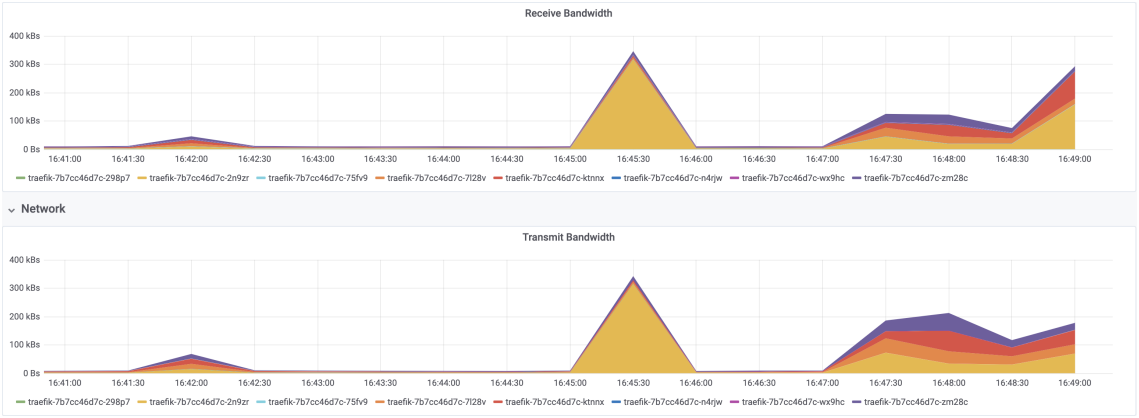

Figure 6.4: Network graph from Prometheus and Graphana

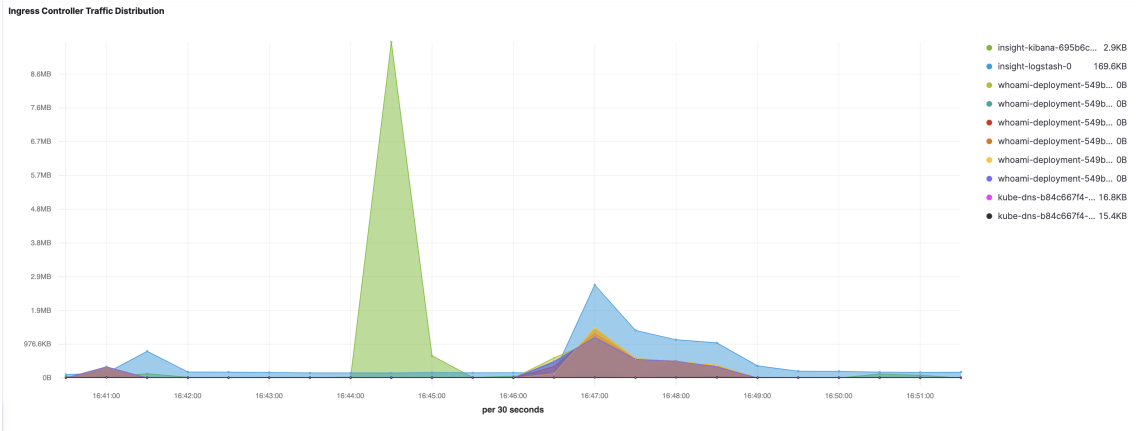

Figure 6.5: Network graph based on data from Insight

6.3.5 Verifying Correct Canary Deployments

A common approach on rolling out new revisions in microservice architectures is to use canary deployments. A group of pods with the new version is started and a small percentage of the incoming request face the new deployment. This allows validating new software

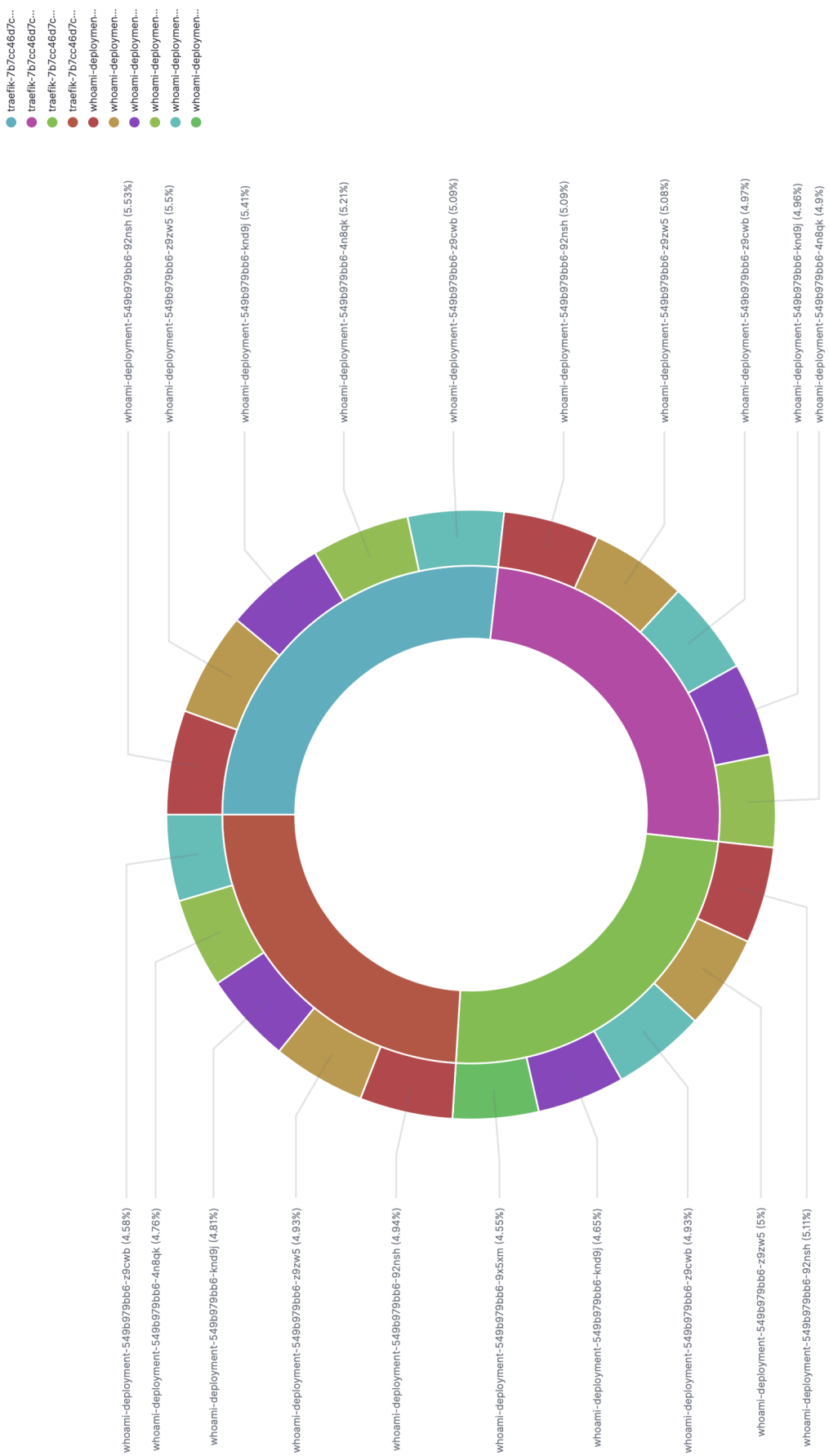

Figure 6.3: Traffic distribution from external and internal loadbalancers

versions with production traffic. Faulty versions will just affect a small group of users and not everyone.

Thus, validating the correct traffic distribution is crucial. Deploying Insight on both the new and old deployment allows computing the traffic distribution.

The described scenario is made up by passing a traffic distribution of 90%/10% to the k8s ingress (Appendix G). After generating traffic on the new endpoint, Figure 6.6 shows the desired traffic distribution is working as intended.

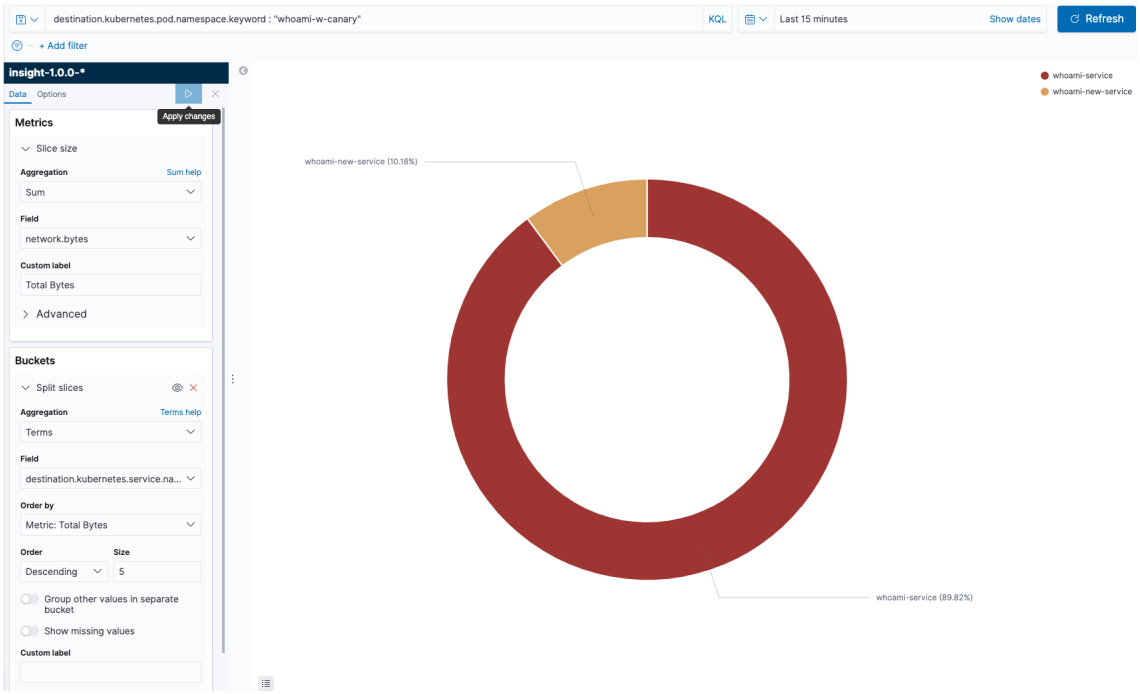

Figure 6.6: Traffic distribution for canary deployment

7 Conclusion & Future Work

The developed proof of concept for a flow monitoring system is based on three data probes, two intermediate data caches and the ELK-Stack for persisting and visualizing aggregated data. Logstash merges incoming flow events generated by multiple network probes running on a per-pod basis and annotates them with the k8s state and service calls. The data format is compatible with Elastic Common Scheme (ECS); thus the SIEM view in Kibana is used as frontend.

The system was already in use for pre-production debugging of load balancing UDP connections. It allows visualizing network activity of higher-level components such as ingress controllers and also provides more details on the network saturation. As the evaluation showed, source/destination-based flows allow for a more fine-grained analysis of network activity in comparison to metric-based monitoring solutions.

An enhanced visualization could use collected flows to build a weighted graph showing how interconnected microservices communicate with each other.

Despite being functional, there are some drawbacks and several points for improvement. While the network probe only requires the Linux capability `NET_RAW` for intercepting packets on the container side interface, the resolver needs to run on each node with system privileges. In managed k8s environments, this is usually prohibited as the required `NET_ADMIN` capability allows manipulation of system-wide routing tables and other security-related configuration. With this limitation, service connection cannot be traced to a specific pod, rendering key features such as load balancing validation useless.

The presented architecture consists of a cluster-wide PostgreSQL database, a cluster-wide Memcached key/value store and a Logstash pipeline. As analyzed in Section 6.1.1, each component group can be scaled up, but the overhead increases. Merging the functionality of the Logstash pipeline, kubeagent and resolver, one microservice running on each node would lead to less data transfer between nodes. This approach would lower the architecture complexity and allow for better scaling and performance of the service to pod lookup table as it only has to contain entries of the local node. A static compiled pipeline could also provide performance benefits compared to a Logstash.

Glossary

Application Programming Interface (API) An Application Programming Interface is a definition of specification and rules providing a description for interfacing with a particular software that implements the API.

Communicating Sequential Processes (CSP) .

conntrack The Linux conntrack table is a kernel-layer network connection tracking table.

container A Linux Container is a set of processes isolated from the rest of the system.

Container Network Interface (CNI) A Container Network Interface is a set of specification for providing networking to containers..

Container Runtime Interface (CRI) A software that starts and stops containers, e.g. Docker [46].

control groups (cgroups) .

Custom Resource Definition (CRD) k8s internal definition for extending the included resources by third-party ones.

Domain Specific Language (DSL) A DSL specifies how data can be queried from a database.

Elastic Common Scheme (ECS) ECS is a definition of how data has to be structured for being compatible with Elastic apps such as kibana.

ELK-Stack ELK-Stack describes a combination of Elasticsearch, Logstash and Kibana.

Generalized Inverted Index (GIN) .

Hypertext Transfer Protocol (HTTP) HTTP is a common application layer protocol used for transfering serialized information.

Inter Process Communication (IPC) Inter Process Communication (IPC) describes the data exchange between two running process.

Internet Control Message Protocol (ICMP) .

Internet Control Message Protocol for the Internet Protocol Version 6 (ICMPv6) .

Internet Protocol (IP) The Internet Protocol is the base for the world wide web.

IP Address Management (IPAM) IPAM is a system keeping track of already assigned IP addresses and decides which IP address will be assigned next.

IP Virtual Server (IPVS) IPVS incorporates into the Linux Virtual Server acting as load balancer..

iptables IPtables is a userspace tool for configuring firewall rules on Linux based systems.

Java Database Connectivity (JDBC) .

JavaScript Object Notation (JSON) JavaScript Object Notation (JSON) is a text-based data interchange format [3].

Kibana Query Language (KQL) .

Kubernetes (k8s) Kubernetes is an open source container orchestrator.

Linux Apache MySQL PHP (LAMP) A Linux, Apache, MySQL, PHP-Stack is a common combination of components used in classic web development.

Network Address Translation (NAT) .

pod A Pod is the smallest runtime component of Kubernetes (k8s) consisting of a set of one or more containers and additional data.

Process Identifier (PID) .

Role Based Access Control (RBAC) Role Base Access Control is a authorization pattern for permission management.

secure Hypertext Transfer Protocol (HTTPS) HTTPS describes a TLS encrypted HTTP connection.

Security Information and Event Management (SIEM) SIEM is a system bringing different data sources together and helps extracting potential security related threads.

Structured Query Language (SQL) .

Transmission Control Protocol (TCP) TCP is a reliable, stateful IP transport layer.

Transport Layer Security (TLS) TLS is used to to encrypt a TCP connection.

Unix Time Sharing (UTS) .

User Datagram Protocol (UDP) UDP is an IP transport layer protocol with minimal protocol mechanisms.

web hook A web hook is a non standardized communication pattern e.g. used in middleware components. Instead of polling, the server sends a request to a defined endpoint implementing a callback based on the HTTP protocol.

YAML Ain't Markup Language (YAML) YAML is a superset of JSON designed for humans.[24].

Bibliography

[1] Elasticsearch B.V. *Elasticsearch: The Official Distributed Search & Analysis Engine*. Feb. 9, 2020. URL: https://www.elastic.co/elasticsearch (visited on 02/09/2020).

[2] Elasticsearch B.V. *Elasticsearch: Explore, VIsualize, Discover Data*. Feb. 9, 2020. URL: https://elastic.co/kibana (visited on 02/09/2020).

[3] *The JavaScript Object Notation (JSON) Data Interchange Format*. Tech. rep. Mar. 2014. DOI: 10.17487/rfc7159.

[4] Paul C. Bryan and Mark Nottingham. *JavaScript Object Notation (JSON) Patch*. RFC 6902. Apr. 2013. DOI: 10.17487/RFC6902. URL: https://rfc-editor.org/rfc/rfc6902.txt.

[5] Zachary Tong Clinton Gormley. *Elasticsearch: The Definitive Guide*. O'Reilly UK Ltd., Jan. 1, 2016. ISBN: 1449358543. URL: https://www.ebook.de/de/product/22275343/clinton_gormley_zachary_tong_elasticsearch_the_definitive_guide.html.

[6] Community. *capabilities(7) - linux manual page*. Mar. 2, 2020. URL: http://man7.org/linux/man-pages/man7/capabilities.7.html (visited on 03/02/2020).

[7] Community. *cgroups(7) - linux manual page*. Mar. 2, 2020. URL: http://man7.org/linux/man-pages/man7/cgroups.7.html (visited on 03/02/2020).

[8] Community. *Effective Go - The Go Programming Language*. Mar. 3, 2020. URL: https://golang.org/doc/effective_go.html (visited on 03/03/2020).

[9] Community. *etcd | Home*. Feb. 15, 2020. URL: https://etcd.io (visited on 02/15/2020).

[10] Community. *Frequently Asked Questions (FAQ) - Golang*. Mar. 18, 2020. URL: https://golang.org/doc/faq (visited on 03/18/2020).

[11] Community. *Go Memcached client library golang*. Mar. 23, 2020. URL: github.com/bradfitz/gomemcache (visited on 03/23/2020).

[12] Community. *Helm*. Mar. 18, 2020. URL: https://helm.sh (visited on 03/18/2020).

[13] Community. *index | alpine linux*. Mar. 23, 2020. URL: https://alpinelinux.org (visited on 03/23/2020).

[14] Community. *kubernetes/client-go: Go client for Kubernetes*. Feb. 15, 2020. URL: https://github.com/kubernetes/client-go (visited on 02/15/2020).

[15] Community. *Linux kernel coding style*. Mar. 18, 2020. URL: https://www.kernel.org/doc/html/v4.10/process/coding-style.html (visited on 03/18/2020).

[16] Community. *linux/ip-sysctl.txt*. Mar. 24, 2020. URL: https://github.com/torvalds/linux/blob/bd2463ac7d7ec51d432f23bf0e893fb371a908cd/Documentation/networking/ip-sysctl.txt (visited on 03/24/2020).

[17] Community. *namespaces(7) - linux manual page*. Mar. 2, 2020. URL: http://man7.org/linux/man-pages/man7/namespaces.7.html (visited on 03/02/2020).

[18] Community. *Package for comparing Go values in tests*. Mar. 23, 2020. URL: https://github.com/google/go-cmp (visited on 03/23/2020).

[19] Community. *pq - A pure Go postgres driver for Go's database/sql package*. Mar. 16, 2020. URL: https://github.com/lib/pq (visited on 03/16/2020).

[20] Community. *Ruby Programming Language*. Mar. 3, 2020. URL: https://www.ruby-lang.org/ (visited on 03/03/2020).

[21] Community. *sirupsen/logrus: Structured, pluggable logging for Go*. Mar. 23, 2020. URL: github.com/sirupsen/logrus (visited on 03/23/2020).

[22] Community. *socket(2) - Linux Programmer Manual*. Mar. 20, 2020. URL: http://man7.org/linux/man-pages/man2/socket.2.html (visited on 03/20/2020).

[23] Community. *TCPDUMP/LIBPCAP public repository*. URL: https://www.tcpdump.org (visited on 02/24/2020).

[24] Community. *The Official YAML Web Site*. Feb. 9, 2020. URL: https://yaml.org (visited on 02/09/2020).

[25] Kubernetes Community. *Concepts - Kubernetes*. Feb. 9, 2020. URL: https://kubernetes.io/docs/concepts/ (visited on 02/09/2020).

[26] Kubernetes Community. *Documentation - Kubernetes*. Feb. 9, 2020. URL: https://kubernetes.io/docs/home/ (visited on 02/09/2020).

[27] Kubernetes Community. *Kubernetes API Reference Docs*. Feb. 11, 2020. URL: https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/ (visited on 02/11/2020).

[28] Kubernetes Community. *Production-Grade Container Orchestration - Kubernetes*. Feb. 9, 2020. URL: https://kubernetes.io (visited on 02/09/2020).

[29] Open Source Community. *Container Network Interface*. 2019. URL: https://github.com/containernetworking/cni (visited on 10/23/2019).

[30] Open Source Community. *Golang*. 2019. URL: https://golang.org (visited on 09/14/2019).

[31] Open Source Community. *Gopacket*. 2019. URL: https://github.com/google/gopacket (visited on 10/22/2019).

[32] Open Source Community. *Prometheus*. 2019. URL: https://prometheus.io (visited on 10/22/2019).

[33] Open Source Community. *virtual ethernet device*. 2019. URL: http://man7.org/linux/man-pages/man4/veth.4.html (visited on 10/23/2019).

[34] Containous. *Traefik, the cloud native edge router*. Ed. by Containous. Feb. 21, 2020. URL: https://containo.us/traefik/ (visited on 02/21/2020).

[35] Drone. *Drone CI - Automate Software Testing and Delivery*. Mar. 18, 2020. URL: https://drone.io (visited on 03/18/2020).

[36] Mike Duckett et al. *Accommodating a Maximum Transit Unit/Maximum Receive Unit (MTU/MRU) Greater Than 1492 in the Point-to-Point Protocol over Ethernet (PPPoE)*. RFC 4638. Sept. 2006. DOI: 10.17487/RFC4638. URL: https://rfc-editor.org/rfc/rfc4638.txt.

[37] Kjeld Borch Egevang and Paul Francis. *The IP Network Address Translator (NAT)*. RFC 1631. May 1994. DOI: 10.17487/RFC1631. URL: https://rfc-editor.org/rfc/rfc1631.txt.

[38] W. Felter et al. "An updated performance comparison of virtual machines and Linux containers." In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Mar. 2015, pp. 171–172. DOI: 10.1109/ISPASS.2015.7095802.

[39] David F. Ferraiolo. *Role-Based Access Control 2nd Edition*. ARTECH HOUSE INC, Jan. 11, 2007. 381 pp. ISBN: 1596931132. URL: https://www.ebook.de/de/product/5867164/david_f_ferraiolo_role_based_access_control_2nd_edition.html.

[40] The Apache Software Foundation. *ab -Apache HTTP server benchmarking tool*. Mar. 27, 2020. URL: http://httpd.apache.org/docs/2.4/programs/ab.html (visited on 03/27/2020).

[41] The PostgreSQL Global Development Group. *PostgreSQL: Documentation: 12*. Mar. 16, 2020. URL: https://www.postgresql.org/files/documentation/pdf/12/postgresql-12-A4.pdf (visited on 03/16/2020).

[42] Tony Hansen and Donald E. Eastlake 3rd. *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*. RFC 6234. May 2011. DOI: 10.17487/RFC6234. URL: https://rfc-editor.org/rfc/rfc6234.txt.

[43] C. A. R. Hoare. "Communicating Sequential Processes." In: *Commun. ACM* 21.8 (Aug. 1978), pp. 666–677. ISSN: 0001-0782. DOI: 10.1145/359576.359585. URL: https://doi.org/10.1145/359576.359585.

[44] Amazon Web Services Inc. *Amazon AWeb Services AWS - Server Hosting & Cloud Services*. Mar. 27, 2020. URL: https://aws.amazon.com/ (visited on 03/27/2020).

[45] Corelight Inc. *Community ID Flow Hashing*. Ed. by Christian Kreibich. URL: https://github.com/corelight/community-id-spec (visited on 02/27/2020).

[46] Docker Inc. *Container Runtime with Docker | Docker*. Feb. 15, 2020. URL: https://www.docker.com/products/container-runtime (visited on 02/15/2020).

[47] RedHat Inc. *coreos/flannel: flannel is a network fabric for containers, designed for Kubernetes*. Feb. 15, 2020. URL: https://github.com/coreos/flannel (visited on 02/15/2020).

[48] Open Container Initiative. *image-spec/spec.md at master*. Feb. 16, 2020. URL: https://github.com/opencontainers/image-spec/blob/master/spec.md (visited on 02/16/2020).

[49] Simon Josefsson. *The Base16, Base32, and Base64 Data Encodings*. RFC 4648. Oct. 2006. DOI: 10.17487/RFC4648. URL: https://rfc-editor.org/rfc/rfc4648.txt.

[50] Wei Liang(Huawei) Jun Du(Huawei) Haibin Xie(Huawei). *IPVS-Based In-Cluster Load Balancing Deep Dive*. July 9, 2018. URL: https://kubernetes.io/blog/2018/07/09/ipvs-based-in-cluster-load-balancing-deep-dive/ (visited on 07/09/2018).

[51] Grafana Labs. *Grafana: The open observability platform*. Mar. 27, 2020. URL: https://grafana.com (visited on 03/27/2020).

[52] Leslie Lamport. "On interprocess communication." In: *Distributed Computing* 1.2 (June 1986), pp. 86–101. DOI: 10.1007/bf01786228.

[53] Mallik Mahalingam et al. *Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks*. RFC 7348. Aug. 2014. DOI: 10.17487/RFC7348. URL: https://rfc-editor.org/rfc/rfc7348.txt.

[54] Victor Marmol, Rohit Jnagal, and Tim Hockin. "Networking in Containers and Container Clusters." In: 2015.

[55] *memcached - a distributed memory object caching system*. Mar. 16, 2020. URL: https://memcached.org (visited on 03/16/2020).

[56] Rancher. Mar. 23, 2020. URL: https://k3s.io (visited on 03/23/2020).

[57] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446. URL: https://rfc-editor.org/rfc/rfc8446.txt.

[58] Rosebrock. *Setting up LAMP*. John Wiley & Sons, Aug. 20, 2004. 418 pp. ISBN: 0782143377. URL: https://www.ebook.de/de/product/3454038/rosebrock_setting_up_lamp.html.

[59] Dr. Greg R. Ruth, Nevil Brownlee, and Cynthia G. Mills. *Traffic Flow Measurement: Architecture*. RFC 2722. Oct. 1999. DOI: 10.17487/RFC2722. URL: https://rfc-editor.org/rfc/rfc2722.txt.

[60] *User Datagram Protocol*. RFC 768. Aug. 1980. DOI: 10.17487/RFC0768. URL: https://rfc-editor.org/rfc/rfc768.txt.

[61] Abhishek Verma et al. "Large-scale cluster management at Google with Borg." In: *Proceedings of the Tenth European Conference on Computer Systems - EuroSys '15*. ACM Press, 2015. DOI: 10.1145/2741948.2741964.

[62] Justin Zobel and Alistair Moffat. "Inverted Files for Text Search Engines." In: *ACM Comput. Surv.* 38.2 (July 2006), 6–es. ISSN: 0360-0300. DOI: 10.1145/1132956.1132959. URL: https://doi.org/10.1145/1132956.1132959.

List Of Figures

2.1 Bidirectional Flow As Graph 3
2.2 Container Hypervisor Comparison 4
2.3 Kubernetes Components [26] 10

3.1 Overloaded Router in Kubernetes (k8s) 20

4.1 High Level Architecture 21
4.2 Network Probe Internals 22
4.3 Network Probe Flow Container Class Diagrams 23
4.4 Network Probe Injection Using K8s Webhooks 24
4.5 Components Of The Resolver 24

5.1 Kibana SIEM event view 38
5.2 Kibana SIEM timeline 39
5.3 Kibana SIEM network view 40

6.1 Performance impact of the network probe 43
6.2 CPU utilization cummulated by container and namespace 43
6.4 Network graph from Prometheus and Graphana 44
6.5 Network graph based on data from Insight 44
6.3 Traffic distribution from external and internal loadbalancers 45
6.6 Traffic distribution for canary deployment 46

List Of Tables

2.1 Selection of Linux Capabilities [6] 5
2.2 Selection of supported Linux socket domains [22] 13
2.3 Some advanced data types supported by PostgreSQL 16

4.1 High Level Architecture Interface Description for Figure 4.1 21

5.1 List of used libraries 28
5.2 Selection of ICMPv6 types and the counterparts (`pkg/flow/common/icmp.go`) 31

Listings

2.1 Creating a linux network namespace . . . 5
2.2 Example CNI initializing container networking . . . 6
2.3 Dockerfile example . . . 7
2.4 k8s Example Pod Definition . . . 9
2.5 k8s Example Pod definition mounting a secret and getting environment variables from a `ConfigMap` . . . 11
2.6 pod with owner reference . . . 12
2.7 `Deployment` with owner reference . . . 13
2.8 k8s `Deployment` . . . 14
2.9 JavaScript Object Notation (JSON) patch example for adding a k8s label . . . 15
2.10 Kibana Query Language (KQL) example . . . 16

5.1 Two-staged Dockerfile . . . 29
5.2 Flow representation in Go (`pkg/flow/flow.go`) . . . 30
5.3 Endpoint ordering (`pkg/flow/communityid/communityid_hasher.go`) . . . 31
5.4 Network flow hashing (`pkg/flow/communityid/communityid_hasher.go`) . . . 31
5.5 ICMPv6 to port equivalent mapping . . . 32
5.6 Connection source detection . . . 32
5.7 Intercepting network traffic `pkg/capture/capture.go` . . . 33
5.8 Flow container datastructure . . . 33
5.9 Watching the k8s API . . . 34
5.10 Regular expression parsing conntrack events . . . 35
5.11 SQL query for pod labels . . . 36
5.12 SQL query for service labels . . . 36
5.13 Template for the mutating webhook . . . 37
5.14 JSON patch injecting the network probe . . . 39

A Network Flow Event

```
const ECSversion = "1.4"

var hostname string

// Agent in ECS
type Agent struct {
    HostName string `json:"hostname"`
    Type     string `json:"type"`
}

// ECS version tag
type ECS struct {
    Version string `json:"version"`
}

// EventDescription in ECS
type EventDescription struct {
    Duration time.Duration `json:"duration"`
    Kind     string        `json:"kind"`
    Action   string        `json:"action"`
    Category string        `json:"category"`
    Dataset  string        `json:"dataset"`
    Start    time.Time     `json:"start"`
    End      time.Time     `json:"end"`
}

// EndpointDescription in ECS
type EndpointDescription struct {
    Address string `json:"address"`
    IP      net.IP `json:"ip"`
    Port    uint16 `json:"port"`
    Bytes   uint64 `json:"bytes"`
    Packets uint64 `json:"packets"`
}

// NetworkDescription in ECS
type NetworkDescription struct {
    Type        string `json:"type"`
    Bytes       uint64 `json:"bytes"`
    Packets     uint64 `json:"packets"`
    Transport   string `json:"transport"`
    CommunityID string `json:"community_id"`
}

```

```
45 // Event contains the event metadata passed to logstash
46 type Event struct {
47     Type        string               `json:"type"`
48     ECS         *ECS                 `json:"ecs"`
49     Agent       *Agent               `json:"agent"`
50     Event       *EventDescription    `json:"event"`
51     Source      *EndpointDescription `json:"source"`
52     Destination *EndpointDescription `json:"destination"`
53     Network     *NetworkDescription  `json:"network"`
54 }
```

B Shortened CI/CD-Pipeline

```yaml
- name: test_insight
  image: golang:1.13
  commands:
  - apt-get update --yes && apt-get install --yes libpcap-dev
  - go vet ./...
  - go test -count=1 -race -v -cover ./...
- name: build_and_publish_kubeagent
  image: plugins/docker
  settings:
    # ...
    tags:
    - ${DRONE_COMMIT_SHA:0:8}
    - ${DRONE_BRANCH}
  depends_on:
  - test_insight
# ... Build other images
- name: deploy
  image: alpine/helm:3.0.2
  user: root
  settings:
    # ...
  commands:
  # ...
  - helm repo add elastic https://helm.elastic.co
  - helm dependency update ./helm/chart/insight/
  - helm upgrade --install --namespace=$${HELM_NAMESPACE} $${HELM_NAME} -f
    ↪  ./helm/chart/values-$${HELM_NAME}.yaml ./helm/chart/insight
  when:
    branch: staging
  depends_on:
  - test_insight
  - build_and_publish_kubeagent
  # ...
```

C PostgreSQL Table Layout

```
\\set VERBOSITY terse
\\set ON_ERROR_STOP true

do language plpgsql $$ begin
raise notice '[+] Creating pods schema';
do $pods$ begin
  create table if not exists pods (
    uid UUID primary key
    , name text not null
    , namespace text not null
    , ip inet
    , definition jsonb not null
  );
  -- Index for containment queries
  create index idx_pods_definition on pods using gin (definition
  ↪  jsonb_path_ops);
end $pods$;

raise notice '[+] Creating services schema';
do $services$ begin
  create table if not exists services (
    uid UUID primary key
    , name text not null
    , cluster_ip inet
    , namespace text not null
    , definition jsonb not null
  );
  create index idx_services_definition on services using gin (definition
  ↪  jsonb_path_ops);
end $services$;

raise notice '[+] Creating endpoints schema';
do $endpoints$ begin
  create table if not exists endpoints (
    uid UUID primary key
    , name text not null
    , namespace text not null
    , definition jsonb not null
  );
  create index idx_endpoints_definition on endpoints using gin (definition
  ↪  jsonb_path_ops);
end $endpoints$;

```

```
41 end $$
```

D Logstash Pipeline

```
  input {
    http {
      id => "flows"
      port => 8080
      codec => "json"
      tags => ["flow"]
    }
}

filter {

  # Remove http fields
  mutate {
    remove_field => ["headers"]
  }

  # Check if the flow is targeting a clusterIP by doing a memcached
  ↪  lookup
  memcached {
    id => "clusterip_lookup"
    hosts => ["insight-memcached"]
    get => {
      "%{[network][community_id]}" => "[@metadata][raw_clusterip_lookup]"
    }
    add_field => {
      "[destination][orig_ip]" => "%{[destination][ip]}"
      "[destination][orig_port]" => "%{[destination][port]}"
      "[network][orig_community_id]" => "%{[network][community_id]}"
    }
    add_tag => [ "dst_is_clusterip" ]
  }

  if "dst_is_clusterip" in [tags] {
    # Parse JSON
    json {
      id => "parse_clusterip_lookup"
      source => "[@metadata][raw_clusterip_lookup]"
      target => "[@metadata][clusterip_lookup]"
    }

    # Replace fields
    mutate {
      replace => {
```

```
43          "[network][community_id]" =>
            ↪  "%{[@metadata][clusterip_lookup][community_id]}"
44          "[destination][ip]" =>
            ↪  "%{[@metadata][clusterip_lookup][replace_ip]}"
45          "[destination][port]" =>
            ↪  "%{[@metadata][clusterip_lookup][replace_port]}"
46        }
47      }
48    }
49
50    jdbc_streaming {
51      id => "metadata_for_src_ip"
52      # Connection details
53      jdbc_driver_library => "/usr/share/extras/jdbc/postgresql-42.2.9.jar"
54      jdbc_driver_class => "org.postgresql.Driver"
55      jdbc_connection_string =>
        ↪  "jdbc:postgresql://insight-kubestatestore-postgresql:5432/insight?user=insight"
56
57      # Cache up to 512 entries for 10s as there is likely more than one
        ↪  flow per IP and we don't want a query
58      # for every incoming packet. (due to possible performance/bandwidth
        ↪  limitations)
59      cache_expiration => 10.0
60      cache_size => 512
61      use_cache => true
62
63      # Queries metadata from the kubernetes state store
64      statement => "
65        select
66            json_build_object(
67                'pods', coalesce(json_agg(pod_metadata) filter (where
    ↪  pod_metadata is not null), '[]')
68                , 'services', coalesce(json_agg(service_metadata) filter
    ↪  (where service_metadata is not null), '[]')
69            ) #>> '{}' as metadata
70        from
71            ((select
72                    p.definition #> '{metadata}' as pod_metadata
73                    , null as service_metadata
74                from pods p
75                where p.definition #> '{\"status\", \"podIPs\"}' @> ?::jsonb
76                )
77            union
78                (select
79                    null as pod_metadata,
80                    s.definition #> '{metadata}' as service_metadata
81                from services s
82                    join endpoints e on (s.name, s.namespace) = (e.name,
    ↪  e.namespace)
83                where e.definition -> 'subsets' @> ?::jsonb
```

```
84            )) as m
85      "
86
87      use_prepared_statements => true
88      prepared_statement_name => "metadata_for_src_ip"
89      target => "[@metadata][jdbc_results][metadata_for_src_ip]"
90      prepared_statement_bind_values => [ "[{\"ip\": \"%{[source][ip]}\"}]",
        ↪  "[{\"addresses\": [{\"ip\": \"%{[source][ip]}\"}]}]" ]
91    }
92
93    jdbc_streaming {
94      id => "metadata_for_dst_ip"
95      jdbc_driver_library => "/usr/share/extras/jdbc/postgresql-42.2.9.jar"
96      jdbc_driver_class => "org.postgresql.Driver"
97      jdbc_connection_string =>
        ↪  "jdbc:postgresql://insight-kubestatestore-postgresql:5432/insight?user=insight"
98      cache_expiration => 10.0
99      cache_size => 512
100     use_cache => true
101     statement => "
102       select
103           json_build_object(
104               'pods', coalesce(json_agg(pod_metadata) filter (where
    ↪  pod_metadata is not null), '[]')
105               , 'services', coalesce(json_agg(service_metadata) filter
    ↪  (where service_metadata is not null), '[]')
106           ) #>> '{}' as metadata
107       from
108           ((select
109                   p.definition #> '{metadata}' as pod_metadata
110                   , null as service_metadata
111               from pods p
112               where p.definition #> '{\"status\", \"podIPs\"}' @> ?::jsonb
113               )
114           union
115               (select
116                   null as pod_metadata,
117                   s.definition #> '{metadata}' as service_metadata
118               from services s
119                   join endpoints e on (s.name, s.namespace) = (e.name,
    ↪  e.namespace)
120               where e.definition -> 'subsets' @> ?::jsonb
121           )) as m
122     "
123     use_prepared_statements => true
124     prepared_statement_name => "metadata_for_dst_ip"
125     target => "[@metadata][jdbc_results][metadata_for_dst_ip]"
126     prepared_statement_bind_values => [ "[{\"ip\":
        ↪  \"%{[destination][ip]}\"}]", "[{\"addresses\": [{\"ip\":
        ↪  \"%{[destination][ip]}\"}]}]" ]
```

```
127     }
128
129     # jsonb type is not supported by jdbc (-> logstash) therefore json is
        ↪  passed as string and parsed back to json by logstash
130     json {
131       id => "parse_metadata_for_src_ip"
132       source =>
          ↪  "[@metadata][jdbc_results][metadata_for_src_ip][0][metadata]"
133       target => "[source][kubernetes]"
134     }
135     json {
136       id => "parse_metadata_for_dst_ip"
137       source =>
          ↪  "[@metadata][jdbc_results][metadata_for_dst_ip][0][metadata]"
138       target => "[destination][kubernetes]"
139     }
140
141     # Extract first pod for easy querying using KQL
142     ruby {
143       id => "make_kubernetes_metadata_searchable"
144       path => "/usr/share/logstash/insight/extract_first_array_element.rb"
145     }
146
147     # GeoIP lookup
148     geoip {
149       id => "geoip_lookup_for_src_ip"
150       source => "[source][ip]"
151       target => "[source][geo]"
152       fields => ["CITY_NAME", "COUNTRY_NAME", "LOCATION",
          ↪  "AUTONOMOUS_SYSTEM_NUMBER", "AUTONOMOUS_SYSTEM_ORGANIZATION"]
153     }
154     geoip {
155       id => "geoip_lookup_for_dst_ip"
156       source => "[destination][ip]"
157       target => "[destination][geo]"
158       fields => ["IP", "CITY_NAME", "COUNTRY_NAME", "LOCATION",
          ↪  "AUTONOMOUS_SYSTEM_NUMBER", "AUTONOMOUS_SYSTEM_ORGANIZATION"]
159     }
160   }
161
162   output {
163     elasticsearch {
164       hosts => ["elasticsearch-master"]
165       index => "insight-1.0.0-%{+YYYY.MM.dd}"
166       manage_template => true
167       ilm_enabled => false
168       template => "/usr/share/logstash/insight/insight.template.json"
169       template_name => "insight-1.0.0"
170       template_overwrite => true
171       action => "index"
```

```
172     }
173 }
```

E Elasticsearch Index Template

```
{
  "index_patterns": ["insight-1.0.0-*"],
  "settings": {
    "index": {
      "number_of_shards": 3,
      "number_of_replicas": 0,
      "refresh_interval": "10s",
      "codec": "best_compression",
      "mapping": {
        "total_fields": {
          "limit": "10000"
        }
      },
      "query": {
        "default_field": [
          "source.*",
          "source.kubernetes.pod.labels.*",
          "source.kubernetes.service.labels.*",
          "destination.*",
          "destination.kubernetes.pod.labels.*",
          "destination.kubernetes.service.labels.*",
          "network.*"
        ]
      }
    }
  },
  "mappings": {
    "numeric_detection": true,
    "properties": {
      "@timestamp": {"type": "date"},
      "@version": {"type": "keyword"},
      "type": {"type": "keyword"},
      "host": {"type": "ip"},
      "event": {
        "type": "object",
        "properties": {
          "kind": {"type": "keyword"},
          "category": {"type": "keyword"},
          "action": {"type": "keyword"},
          "dataset": {"type": "keyword"},
          "duration": {"type": "long"},
          "start": {"type": "date"},
          "end": {"type": "date"}
        }
```

```
45          },
46          "agent": {
47            "type": "object",
48            "properties": {
49              "hostname": {"type": "keyword"},
50              "type": {"type": "keyword"}
51            }
52          },
53          "source": {
54            "type": "object",
55            "properties": {
56              "kubernetes": {
57                "type": "object",
58                "properties": {
59                  "metadata": {
60                    "type": "object",
61                    "properties": {
62                      "pod": {
63                        "type": "object",
64                        "properties": {
65                          "labels.*": {
66                            "type": "keyword"
67                          },
68                          "annotations.*": {
69                            "type": "keyword"
70                          }
71                        }
72                      },
73                      "service": {
74                        "type": "object",
75                        "properties": {
76                          "labels.*": {
77                            "type": "keyword"
78                          },
79                          "annotations.*": {
80                            "type": "keyword"
81                          }
82                        }
83                      },
84                      "pods": {
85                        "type": "nested",
86                        "properties": {
87                          "labels.*": {
88                            "type": "keyword"
89                          },
90                          "annotations.*": {
91                            "type": "keyword"
92                          }
93                        }
94                      },
```

```
                    "services": {
                      "type": "nested",
                      "properties": {
                        "labels.*": {
                          "type": "keyword"
                        },
                        "annotations.*": {
                          "type": "keyword"
                        }
                      }
                    }
                  }
                }
              }
            },
            "geo": {
              "dynamic": true,
              "type": "object",
              "properties": {
                "city_name": {"type": "keyword"},
                "country_name": {"type": "keyword"},
                "location": {"type": "geo_point"},
                "ip": {"type": "ip"}
              }
            },
            "ip": {"type": "ip"},
            "port": {"type": "integer"},
            "address": {"type": "keyword"},
            "packets": {"type": "long"},
            "bytes": {"type": "long"}
          }
        },
        "destination": {
          "type": "object",
          "properties": {
            "kubernetes": {
              "type": "object",
              "properties": {
                "metadata": {
                  "type": "object",
                  "properties": {
                    "pod": {
                      "type": "object",
                      "properties": {
                        "labels.*": {
                          "type": "keyword"
                        },
                        "annotations.*": {
                          "type": "keyword"
                        }
```

```
                    }
                  },
                  "service": {
                    "type": "object",
                    "properties": {
                      "labels.*": {
                        "type": "keyword"
                      },
                      "annotations.*": {
                        "type": "keyword"
                      }
                    }
                  },
                  "pods": {
                    "type": "nested",
                    "properties": {
                      "labels.*": {
                        "type": "keyword"
                      },
                      "annotations.*": {
                        "type": "keyword"
                      }
                    }
                  },
                  "services": {
                    "type": "nested",
                    "properties": {
                      "labels.*": {
                        "type": "keyword"
                      },
                      "annotations.*": {
                        "type": "keyword"
                      }
                    }
                  }
                }
              }
            },
            "geo": {
              "dynamic": true,
              "type": "object",
              "properties": {
                "city_name": {"type": "keyword"},
                "country_name": {"type": "keyword"},
                "location": {"type": "geo_point"},
                "ip": {"type": "ip"}
              }
            },
            "ip": {"type": "ip"},
```

```
            "port": {"type": "integer"},
            "address": {"type": "keyword"},
            "packets": {"type": "long"},
            "bytes": {"type": "long"}
          }
        },
        "network": {
          "type": "object",
          "properties": {
            "community_id": {"type": "keyword"},
            "bytes": {"type": "long"},
            "packets": {"type": "long"},
            "type": {"type": "keyword"}
          }
        },
        "tags": {
          "type": "keyword"
        }
      }
    }
  }
```

F Test Application Deployment

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: whoami-deployment
  labels:
    app: whoami
spec:
  replicas: 6
  selector:
    matchLabels:
      app: whoami
  template:
    metadata:
      labels:
        app: whoami
    spec:
      containers:
        - name: whoami
          image: containous/whoami:latest
          ports:
            - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: whoami-service
spec:
  selector:
    app: whoami
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: whoami-ingress
spec:
  rules:
    - host: whoami.insight.xvzf.tech
      http:
        paths:
```

```
45            - path: /
46              backend:
47                serviceName: whoami-service
48                servicePort: 80
```

G Test Canary Deployment

```yaml
---
apiVersion: v1
kind: Namespace
metadata:
  name: whoami-w-canary
  annotations:
    insight: "true"
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: whoami-deployment
  namespace: whoami-w-canary
  labels:
    app: whoami
spec:
  replicas: 2
  selector:
    matchLabels:
      app: whoami
  template:
    metadata:
      labels:
        app: whoami
    spec:
      containers:
        - name: whoami
          image: containous/whoami:latest
          ports:
            - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: whoami-service
  namespace: whoami-w-canary
spec:
  selector:
    app: whoami
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
---
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: whoami-new-deployment
  namespace: whoami-w-canary
  labels:
    app: whoami-new
spec:
  replicas: 2
  selector:
    matchLabels:
      app: whoami-new
  template:
    metadata:
      labels:
        app: whoami-new
    spec:
      containers:
        - name: whoami
          image: containous/whoami:latest
          ports:
            - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: whoami-new-service
  namespace: whoami-w-canary
spec:
  selector:
    app: whoami-new
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: whoami-ingress
  namespace: whoami-w-canary
  annotations:
    traefik.ingress.kubernetes.io/service-weights: |
      whoami-service: 90%
      whoami-new-service: 10%

spec:
  rules:
    - host: whoami-canary.insight.xvzf.tech
      http:
```

```
95          paths:
96            - path: /
97              backend:
98                serviceName: whoami-service
99                servicePort: 80
100           - path: /
101             backend:
102               serviceName: whoami-new-service
103               servicePort: 80
```