UNIVERSITY OF AMSTERDAM

RESEARCH PROJECT 2

Scaling of containerized network functions

July 5, 2021

Students:
Mohanad Elamin
melamin@os3.nl

Pim Paardekooper
ppaardekooper@os3.nl

Course:
Research Project 2

Course code:
53842REP6Y

Abstract

This research extends on the EPI project [1]. That project tries to create a data sharing path between health care providers securely and dynamically. If two providers do not have a data sharing path or they violate data-sharing policies because of missing functionalities such as no way to encrypt traffic. A proxy is set up between the two providers where traffic gets redirected through. This proxy can then add the functionality as encryption by deploying network functions called bridging function and redirecting the traffic through those functions. In our research a proof of concept is created that is based on Kubernetes and allows us to deploy the EPI framework, pass traffic through the framework and scale the bridging function. Scaling of the bridging function can make it that resources are better utilized or that fluctuations in traffic can be better accommodated. By scaling horizontally we can add more bridging functions to help with traffic demands. Our research focused on how application traffic's latency changed when scaling the bridging functions horizontally. The scaling of the bridging functions is researched in this report by looking at the impact of varying Kubernetes autoscaling thresholds for bridging functions on end-user application traffic. So we can see if autoscaling can improve the performance of the bridging functions. While implementing everything, we decided on experimenting on the horizontal autoscaler as this is more adjustable than the vertical autoscaler, and would certainly have some impact on the application traffic. Therefore in our experiments, we implemented the Horizontal Pod Autoscalers from Kubernetes. Experiments were conducted on it by adjusting the threshold of when to scale pods and adding load to the system by increasing the number of requests made that passed through the bridging functions. The experiments of this project unfold that long-lived traffic sessions don't benefit from horizontal scaling of the bridging function due to the load balancing logic of Kubernetes services. Moreover, the processing logic of the bridging function used in the Proof of concept is CPU-centric; thus, the steady memory usage didn't cause any scale-out events. And finally, testing results also consistently shows an impact on end-user experience due to a slight increase in response time due to the scale-out load distribution overhead. This research can be complemented by examining different types of application traffic and bridging functions as well as using machine learning to find the optimal scaling threshold.

1 Introduction

The EPI (Enabling Personalised Intervention) project addresses the problem of sharing data between healthcare providers while adhering to data-sharing policies [2]. It does this by assessing if two providers have a feasible data sharing path in terms of reachability and security. Is this not the case because they miss certain functionalities, such as being able to encrypt data or protecting outgoing traffic via a firewall, then a bridging function is deployed to provide such functionalities. Bridging functions are provided in the form of network functions. The bridging functions can provide different services like encryption and firewall protection. Bridging functions are provided using Virtual Network Functions (VNFs). Throughout this paper, the Bridging functions are referred to as Containerized Bridging Functions or CBFs. Each VNF implements a bridging function. A series of connected bridging functions is called a bridging function chain (BFC). VNF nodes are identified with a unique IP address to be able to forward and redirect incoming packets to them. This forwarding is done by proxy nodes.

Two proxy node types were provided by our supervisor. One is based on the combination of IP multiplexing and a reverse proxy. The second one is partially based on the SOCKS protocol. The proxy nodes are responsible for:

- Identifying the CBFs required to bridge a channel and place them in a service chain.
- Establish a connection with the CBFs
- Enable Area-Area routing by enforcing packet redirections along the BFCs.

In our project, we will extend the SOCKS proxy. This is the preferred proxy by the EPI project, as it redirects traffic without interpreting it, which allows for more application protocols to be forwarded and allows for more functionalities. The proxy will redirect traffic to the bridging functions that will be scaled based on Kubernetes autoscalers. Kubernetes is an orchestration tool for containers, with already defined autoscaling implementation that allows us to scale the bridging functions. The goal is to show the impact scaling has on the user experience by focusing on how it impacts the application traffic. First, we show what has been done in the field of VNF scaling (section 4). Then we will talk about our setup and the design choices made (section 5). After that, we show the experiments we will do to answer the research question. Lastly, we will discuss our results (section 9).

2 Research Questions

The main research question of this project is:

- **What is the impact of varying Kubernetes autoscaling thresholds for bridging functions on end-user's application traffic latency?**

To be able to answer the main research question, the following sub-questions are specified:

- What metrics should trigger the horizontal scaling of the bridging function considering its processing logic?
- What is the impact of the bridging functions horizontal scaling on application traffic's latency?
- What is the impact of reconfiguration on in-transit traffic?

3 Background

In this section, we give a short overview of what scaling entails and what the SOCKs proxy is that we are going to use.

3.1 Scaling

Scaling is the process of providing more resources to respond to application demand increase. There are mainly two options for scaling [3]:

- Horizontal scaling: Also known as scaling-out or scaling-in, is the process of adding or removing machines or instances from a pool of resources.
- Vertical scaling: Also known as scaling-up or scaling-down, is the process of reducing computing resources on the same machine or instance. The resources increase or decrease can include the CPU, Memory, Disk, or Network resources.

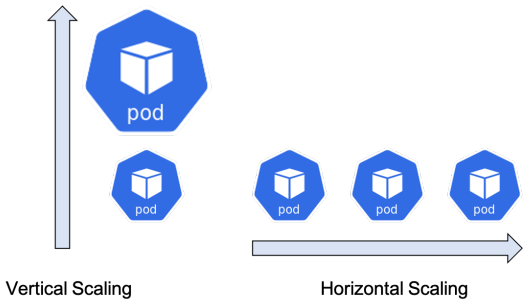


Figure 1: Horizontal scaling is the process of adding or removing machines or instances from a pool of resources. In contrast, vertical scaling is the process of reducing computing resources on the same machine or instance.

The main difference between the two options is that horizontal scaling requires an external load balancing mechanism to distribute the load across all the nodes within the resources pool. Different factors should be considered when deciding between the two options, for example, the performance and the redundancy requirement of the bridging function. If performance requirements of pods fluctuate a lot vertical scaling might be called often, in which case it might be better to add one extra instance by horizontal scaling.

Triggering the scaling action is either a manual activity done by the system administrator or an automatic process known as autoscaling. With autoscaling, a separate auto-scaler system will monitor the usage of the target application, and trigger scaling action based on the scalability requirement and desired scale option, which is either Horizontal or Vertical scaling. The traffic through the bridging functions and the load it generates is where the autoscalers scale on in our research. This traffic first flows through a SOCKS proxy.

3.2 SOCKs

SOCKS stands for Socket Secure, it routes traffic to a server on behalf of a client [4]. It does this for any type of traffic generated by any protocol or program. A SOCKs proxy does not interpret network traffic. It works on layer 5 and can therefore not tunnel protocols operating below that layer. Examples of protocols above layer 5 it can handle are HTTP, HTTPS, POP3, SMTP and FTP. This is a benefit over other proxies such as a TCP proxy or an HTTP proxy, as it is not dependent on the usage of higher-layer protocols. Because of this it also provides a way to redirect traffic like UDP.

4 Related Work

On the paper "Cloudification and Autoscaling Orchestration for Container-based Mobile Networks Toward 5G: Experimentation, Challenges, and Perspectives" [5], Duc-Hung et al. explained the autoscaling mechanisms available in existing container orchestrators in the IT domain. They also conducted multiple autoscaling experiments and provided a practical evaluation of scaling the mobile core elements, specifically the Serving Gateway (SGW).

Another paper proposes three key factors which should be considered for auto-scaling methods in Kubernetes [6]. These factors are a conservative constant, adaption interval Control Loop Time Period (CLTP) and stopping at most one container each CLTP. Auto-scaling functions need to find a balance between over-provisioning and the performance of the system. Scaling of NFV can solely depend on VNF instances, multiple research has been conducted into dynamic proactive algorithms [7] [8] [9] [10]. They create predictive algorithms based on certain traffic metrics or CPU factors and scale VNF instances that way.

The difference between the scaling of NFV and containers is that NFV has VNF instances that can be chained together. This chaining brings other factors with it that can be used to scale the NFV for example scaling on the VNF chains (BFCs). One research points out that predicting the performance of a VNF chain based on the performance of the discrete network functions is not accurate [11]. Consequently scaling could be improved by looking at the service chains. Another research produces a tool called ElasticSFC, which shows that their auto-scaling techniques based on the VNF chain can reduce costs [12].

In our research, we will not look at virtual network functions but containerized functions. The focus of all the other research is also focused on scaling on chaining and how well predictive algorithms can predict fluctuations in traffic. What we will focus on is how the scaling affects the application traffic and more about what important factors are of scaling CBFs, like horizontal or vertical scaling.

5 Setup

To evaluate our discussed goals, we have setup a Kubernetes cluster that is built on top of a Xen virtualization environment to test the different autoscaling mechanisms of the bridging function [13]. Kubernetes is chosen as a container orchestrator to utilize the already available Vertical and Horizontal autoscaler controllers. And while multiple container orchestration platforms are available, Kubernetes is chosen for this project mainly due to its widespread adoption [14]. The Kubernetes setup comprises three virtual machines: one master node and two worker nodes. For incoming traffic into the Kubernetes cluster, the metal load balancer (Metallb) is used, as it provides an open-source implementation of the scalable network load balancer for bare metal clusters [15]. To deploy the Kubernetes setup, Terraform is used to build the cluster virtual machines. Afterward, Ansible playbooks are used to download and install the Kubernetes software, as well as the underlying flannel network and the Metallb network load balancer [16]. The complete code used for the setup automation is available at GitHub [17].

5.1 Kubernetes Setup Architecture

The Proof of concept setup (Figure 2) is architected to utilize Kubernetes services, an abstraction to expose an application running on a set of pods [18]. The setup includes the following components:

- **Testing Machine:** The testing machine is used to communicate with the Kubernetes cluster using the Locust load generator API that is exposed as a web interface.

- **Metallb LoadBalancer Service:** The metallb network load balancer act as the ingress point to the Kubernetes environment and is used to receive the traffic from the external testing computer.
- **Locust Load Generator:** Locust is an open-source load testing tool that uses distributed architecture to test web services [19]. In the proof of concept setup, the locust load generator simulates clients initiating traffic to the webserver. The Locust software compromise of two building blocks:
  - **Master:** The master is the web interface of the Locust system.
  - **Worker:** The worker is the node that generates the user traffic. The Locust system can scale out the number of workers. As part of the proof of concept of this project, the worker node of Locust is modified to include a redirector agent code that intercepts the traffic going to the end server and redirects that to the SOCKS5 proxy. The redirector agent starts a local server on the worker node and sends all received traffic to the SOCKS5 proxy via the proxy socket.
- **Proxy Service:** The proxy service is a Kubernetes service that receives the traffic from the client, the Locust system. After receiving the traffic, the proxy service forwards the traffic to the SOCKS5 proxy pod.
- **SOCKS5 Proxy:** The proxy forward the traffic to one of the bridging function for further inspection. For the proof of concept setup, traffic redirection to the bridging function is done using iptables rules explained in section 5.2 below.
- **Bridging Function Service:** The bridging function service is a Kubernetes service that receives traffic from the SOCKS5 proxy and forwards it to one of the connected bridging functions.
- **Bridging Function:** The bridging function is the network function that provides further processing of the traffic. Multiple bridging functions can be used, including basic iptables firewall, network monitoring function, and L7 HTTP flow inspector function. Further details are available in the Network functions section 5.3.
- **Server Service:** The server service is a Kubernetes service that receives the traffic from the bridging function and forwards it to one of the end servers.
- **HTTPBIN Server:** HTTPBIN server is an open-source simple HTTP Request & Response Service.

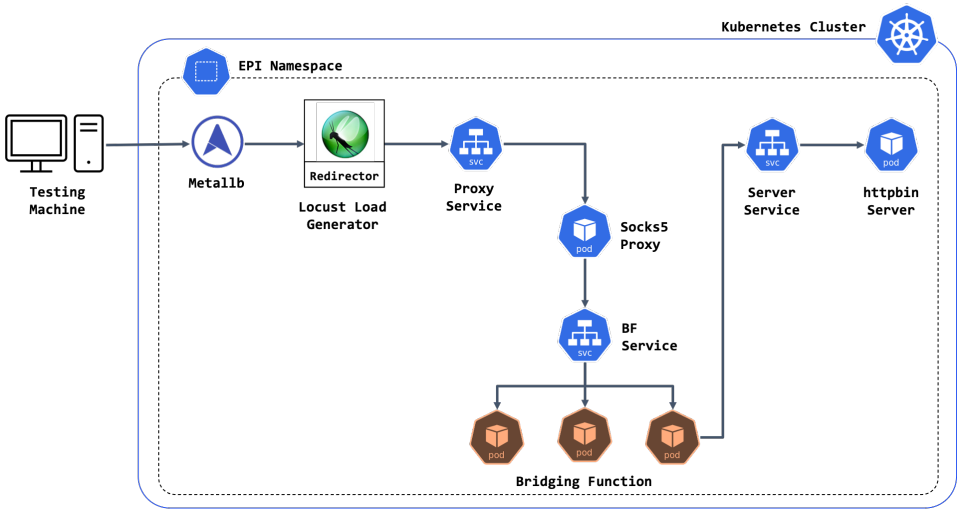


Figure 2: The plot illustrates the POC Kubernetes Setup Architecture, which includes deploying a Locust Load generator with a SOCKS redirector agent redirecting sessions to the SOCKS5 proxy. The SOCKS proxy forwards the traffic to a Bridging function for further processing. The bridging function sends the traffic to its final destination. The testing is initiated from a test machine that sends the traffic to a Metallb load balancer which acts as an ingress point for the PoC environment.

To deploy the proof of concept setup, two helm charts are used:

1. **epi-bf-helm**: A helm chart that is created for this project to deploy the bridging function, proxy, web server and the corresponding Kubernetes services. [20]
2. **Locust**: A Helm chart to deploy the locust load generator. [21]

5.2 Proxy

The proxy image is a simple SOCK5 proxy that listens to the SOCKS connection on port 1080 from the redirector agent. The proxy's primary purpose is to intercept the packets and forward them to the bridging function for further processing.
The redirection of traffic to the bridging function is achieved using an iptables rule (see listing 8); the rule processes the packet in the OUTPUT chain before leaving the proxy pod and redirects the packet to the server by applying a destination NAT with the IP address of the Kubernetes service in front of the Bridging functions.

```
iptables -t nat -A OUTPUT -p tcp --src <PROXY_IP> -j DNAT --to-destination
    <BF_SVC_IP>:<BF_SVC_PORT>
```

Listing 1: Proxy redirection Iptables rule

To enable the SOCKs5 proxy to provide custom metrics, which we will talk more about in section 6.4, we wrapped the proxy in a Flask web server. In a *metricGatherer* object certain metrics can be stored which are exposed on the URL "\metrics".

5.3 Network functions

Network functions are functions that can be performed on whole packets. To make sure this can happen packets are redirected from the SOCKs5 proxy to the Bridging functions with iptables. This can give problems when wanting to work on the application data. Research by the University of Glasgow created a container-based Network Function Virtualization Framework [22]. Instead of using iptables, they use Software-Defined Networks to route

traffic through network function. SDNs would be more dynamic. Still, for our research, the use of iptables is dynamic enough, as we only need the traffic to go to one server. Also, SDNs would add a complexity that is not needed to conduct our experiments. The network function they used can work on the application data by using a python script that uses a NetFilterQueue library [23] [24]. It provides access to packets matched by an iptables rule (see listing 2). The research also provided network functions that could work with network tools like *tc* or *iptables*. One other network function was provided by our supervisor, a firewall application. The network functions that we will be using are the following:

- Firewall: a very small network function that uses iptables as firewall, it will match SYN packets to make sure it always matches some packets and increase the counter packet counter. This way the network function has some processing load.
- Network monitor: Monitors how much traffic goes through the system and prints that in a python script.
- HTTP filter: Gets all packets with the NetfilterQueue library and drops packets when a filter matches the packets application data.

Before any routing decisions are made about a packet it comes through the iptables PREROUTING table, in which we redirect the traffic to the server [25] (see listing 3). This means the packet will go to the FORWARD table in which we add it to the NetFilterQueue. Then to make sure it can go back to the proxy we use the MASQUERADE target which will remember where a connection came from and change the destination to the proxy when a packet comes from a certain host [26].

```
iptables -A FORWARD -j NFQUEUE --queue-num 1
```

Listing 2: Add packets in NetFilterQueue

```
iptables -t nat -A PREROUTING -p tcp -j DNAT --to-destination
    <SERVER_SVC_IP>:<SERVER_SVC_PORT>
iptables -t nat -A POSTROUTING -j MASQUERADE
```

Listing 3: Bridging function redirection Iptables rules

All the bridging functions are containerized using Docker. The image that the network functions are implemented on is the standard Ubuntu 20.04 image from DockerHub. This allows us to call a bash function as the first command when it gets deployed. This way we can call a bash script called "entrypoint.sh", which can, in turn, call other bash scripts. Either a bash script calls the iptables commands mentioned above or the "entrypoint.sh" does it itself. Another calls a monitoring script called "monitor.sh" which we will explain more about in section 7.2. To see an example of a bridging function docker file, the "monitor.sh" and the "entrypoint.sh" bash script look at Appendix B.

5.4 Locust

The locust load generator is used to generate traffic in our network. A locust load generator does this by spawning users that have certain tasks. These tasks are executed at a certain rate by specifying a wait time. In our setup, we use HTTP requests. The locust load generator is exposed on a web URL. A POST request can be made to spawn several users at a certain spawn rate, the spawn rate is given in users per second. The web URL also exposes data that locust collect. The data that is collected is the request per second, the number of users, and the average response time on each time interval.

6 Auto scalers Scaling

This section explains the two auto scalers that are implemented in Kubernetes and how they work. These are the Horizontal Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA). The metrics that the scaling is applied on can be varied. Native Kubernetes only supports CPU and memory that are stored in the metric server. There is a possibility to extend this with custom metrics. This is something will also be looking at in this section.

6.1 Horizontal Pod Autoscaler

The Horizontal Pod Autoscaler, also known as HPA, is a feature of Kubernetes that scales the number of workload pods in response to resource demand. [27]. HPA can automatically scale the number of pods in different Kubernetes controllers, such as a replication controller, deployment, replica set, or a stateful set. The scaling action taken by the HPA can be based on the Pods CPU or memory usage. It is also possible to use custom metrics as the Horizontal Pod Autoscaler resource. This is not supported in the first version of the HPA but is supported in version 2 [28].

For autoscaling using CPU and Memory utilization, the Horizontal Pod Autoscaler collects metrics from the pods using the metrics API, which requires the metrics server to be deployed as part of the Kubernetes cluster. The collected metrics will then get compared against the Pod resources request and limit defined as part of the pod's specification. The ratio between the desired metric value and the current metric value will then be used to calculate the desired number of replicas using the following equation:

$$desiredReplicas = ceil[currentReplicas * (currentMetricValue/desiredMetricValue)]$$

Where:

- **desiredReplicas**: This is the count of replica that will be deployed after the Horizontal Autoscaling event.
- **currentReplicas**: This is the number of pods in the current deployment, replica set, or a stateful set.
- **currentMetricValue**: This is the current pod metric value collected via the metric server or any supported custom metrics APIs, such as Prometheus Adapter.
- **desiredMetricValue**: This is the target metric value needed by the application.

The same calculation above is used for other metrics as well. Natively Kubernetes does not support other metrics, therefore third-party adaptors like the Prometheus Adapter are needed [29].

6.1.1 HPA scaling behavior:

Starting from Kubernetes v1.18, HPA allows for a configurable scaling behavior by defining scaling policies for scale-down and scale-up. The scaling policies control the number of replicas to be scaled during a limited period specified using the *periodSeconds* variable. The policy definitions target value can be defined as the number of pods or as a percentage of pods. Then to avoid undesired scaling events due to metric fluctuation, a stabilization window can be used to prevent scaling, either up or down, for a specific period of time defined using the *stabilizationWindowSeconds* variable. In that window, the desired state from the past period is used instead. The default values of the Horizontal Pod Autoscaler scaling policy are defined below in listings 4, custom values will be merged with the default values [27].

```
behavior:
  scaleDown:
    stabilizationWindowSeconds: 300
    policies:
    - type: Percent
      value: 100
      periodSeconds: 15
  scaleUp:
    stabilizationWindowSeconds: 0
    policies:
    - type: Percent
      value: 100
      periodSeconds: 15
    - type: Pods
      value: 4
      periodSeconds: 15
    selectPolicy: Max
```

Listing 4: Scaling Policy Default Behavior

6.2 Vertical Pod Autoscaler

Vertical Pod Autoscaler (VPA) tries to find the optimal resource allocation for one pod [30]. When just guessing you can run into the problem of having too few resources, which will slow down the pod with CPU throttling or kill the pod if an Out-of-memory error occurs. Also having too many resources will be more expensive and reduces the total amount of resources all pods can use.

Kubernetes gives resources to a pod based on requests and limits [31]. Requests are the minimum resources that are reserved for a specific pod, if there are fewer than the requested resources available a pod is not scheduled. Limits are the maximum amount of resources that a pod can use if there are more resources needed than requested a pod can get more resources from the system only if there are any. When a pod goes over that limit it will be killed.

Resources that can be used are CPU, Memory, Ephemeral storage, GPUs, Linux Kernel Huge Pages and custom metrics. Most network functions will only need CPU and Memory and for VPA those are the only two supported for scaling, so this is where we will be focusing on. GPUs would be interesting to look at for a network function point as it would accelerate certain functions, however, this is out of the scope of our research.

6.2.1 implementation

The implementation of VPA is still in beta and very new [32] and there are not many factors that can be changed. Therefore the focus of our research will be on HPA. The VPA works by the following three components:

- Recommender: This monitors the current and past resource consumptions and from that calculates recommended values for the CPU and memory requests of the container. These past resources can be checkpoints created during previous runs.
- Updater: This checks whether a running pod has the recommended resources set. Is this not the case then it will recreate it with the recommended request values. Certain arguments can be set such as: when to recreate a pod, how much resource increase it can get per scaling event, how long the pod is running before scaling and how many of a certain type of pod should be recreated at once.
- Admission plugin: This sets the new calculated recommended resource requests on new pods.

The VPA can run in different modes that change how the new recommended request values are updated in the pods:

- Auto: Updates it in place, without stopping the container. This is not yet implemented.
- Recreate: Updates it by recreating pods.
- Initial: Only gives newly recreated pods the new request values.
- Off: Recommendations for the values are never applied but can be inspected.

As VPA is still in beta the implementation has certain limitations. The most important once are that recommendation might exceed available resources in which case the pods go pending. Furthermore, the algorithm in the recommender is the only one available, to change it would mean implementing a new recommender itself. As such the only parameters that are changeable is when to evict certain pods in the updater, creating and testing those on different network functions requires lots of traffic as the recommender slowly works to the optimal resource values based on historical data. Therefore this is out of the scope of our research.

6.3 HPA vs VPA

Horizontal autoscaling and Vertical autoscaling solve two different problems. Horizontal scaling solves the problem of how many replicas of a certain pod spawn. Vertical autoscaling solves the problem of how many resources to allocate to one specific pod. Horizontal should be preferred over vertical with fluctuating demands [33]. Vertical is better in stateful services as they cannot use another pod without migrating state and still need a way to reduce resource cost when overusing or add resources when pods are stagnating. Vertical can also be used to fix initial resource demands. These demands can be too lean or too strict and vertical can over time fix them to the right amount based on usage data. Using both vertical and horizontal scaling can be done with *multidimensional Pod autoscaling*. This a Google Kubernetes Engine (GKE) only solution that lets VPA work on the memory and HPA on CPU utilization [34]. This is in line with what the VPA documentation says [32]. To use VPA and HPA together they need not interfere with each other's calculation by working with other metrics. These other metrics can also be custom metrics.

6.4 Custom metrics

Instead of autoscaling on CPU and Memory, custom metrics can be used. For now, this is only available for HPA not VPA. Kubernetes does not support custom metrics natively it should be added externally. There are multiple of those available. Three of those implementations are not platform dependent these are

- Prometheus Adapter. An implementation of the custom metrics API that attempts to support arbitrary metrics following a set label and naming scheme.
- Datadog Cluster Agent: Implementation of the external metrics provider, using Datadog as a backend for the metrics.
- Kube Metrics Adapter: A general-purpose metrics adapter for Kubernetes that can collect and serve custom and external metrics for Horizontal Pod Autoscaling. Provides the ability to scrape pods directly or from Prometheus through user-defined queries. Also capable of serving external metrics from several sources including AWS' SQS and ZMON monitoring.

In our research, we opted for the Prometheus adapter. The Datadog Cluster agent is not open source. The Kube Metrics Adapter was not chosen as implementing it was difficult due to dependencies, but it is a good alternative to use. Therefore the Prometheus Adapter is implemented.

The Prometheus Adapter is basically a way for Prometheus and Kubernetes to speak to each other (see figure 3) [35] [36]. Prometheus itself is an open-source monitoring solution, by a web interface data can be queried and plotted [37]. A Prometheus instance can get metrics from a web application. In our case, the proxy is that web application that exposes the metrics. This is done by wrapping the proxy in Flask webserver [38]. The metrics are exposed on the URL "\metrics" by default and uses a Prometheus Format [39]. The ServiceMonitor is deployed and pointed to the web application to give the metrics to the Prometheus instances. Then the Prometheus adapter gets the metrics for every scrape interval by pointing the *prometheus-url* to the Prometheus instance. Then with Prometheus rules the adapter specifies which metrics to scrape [40]. These rules can then also specify a function to execute on the metric, for example, this can be to calculate the rate of a metric by combining it with the scrape interval. Then an HPA can use the custom metric by targeting the webserver that exposed the metric and specifying the metric you want to use with the *metricName*. The scaling is then done on the image pointed at by the *scaleTargetRef*. The scaling is triggered in the same way as HPA with CPU and memory as explained in section 6.1.

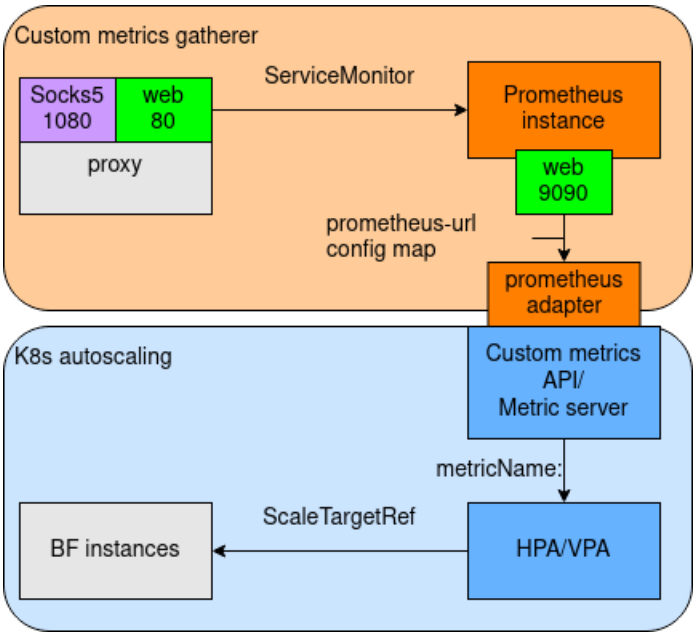


Figure 3: A socks proxy exposes a web server that gives metrics on the "/metrics" URL. A Prometheus instance will get those metrics by a ServiceMonitor. A custom metrics API will scrape the metrics of the Prometheus instance by a Prometheus-URL and Prometheus rules defined in a config map. These rules can also modify the metrics with a function, for example, it can transform an absolute value in a rate over time. The HPA can then have access to a metric by specifying the metric name. The scaling is done on the image specified by ScaleTargetRef.

7 Methodology

In this section, we will explain the way we setup our testing environment and how the workflow of our experiments looks. We will also go into more detail about how we collected the data we use in our data plotting.

7.1 Testing Workflow

Multiple testing scenarios are performed throughout this project based on different traffic load combinations and Horizontal Pod Autoscaler thresholds. The list of all the parameters used for testing is available in appendix A. To perform all the testing in a timely manner, an automated testing workflow is developed.

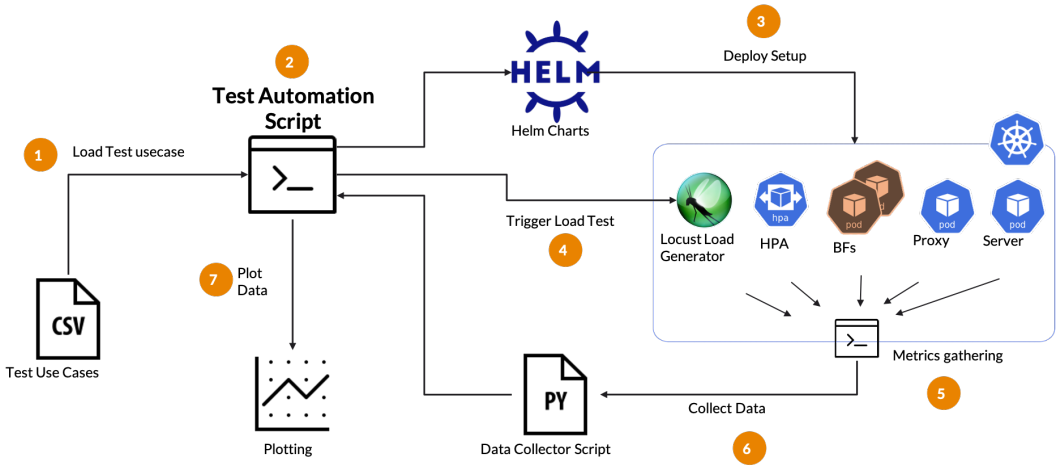


Figure 4: The plot illustrates the testing Automation script, which reads the test scenarios from a CSV file; afterward, it deploys all the required elements using Helm charts. At the end of each testing experiment, a data collection script pulls all the metrics collected during the test using metric gathering scripts running on all elements. The collected data are then plotted for analysis.

The testing workflow shown in figure 4 follows the following steps:

1. The first step includes loading a CSV file that contains the parameters of the testing scenarios by the automation script.
2. The automation script starts deploying the required elements.
3. The deployment is performed using two helm charts, one to deploy the Locust load generator and the other to deploy the rest: the pods, services and autoscalers for the bridging function, proxy and server.
4. After the setup deployment, the automation script instructs the locust load generator to start spawning the user traffic.
5. During the setup, multiple data collection scripts collect the CPU and Memory metrics from the running pods. At the same time, the testing machine pulls the HPA utilization and number of pods deployed.
6. At the end of the 100 seconds test duration, the data collection scripts pull all the data to the test machine.
7. The collected data is then plotted for analysis.

The details of the data collection scripts are available in section Data generation and collection section below.

7.2 Data generation and collection

The data that was collected came from three different locations, from the locust load generator, from the metric server and the bridging functions pods.
The locust load generator gathers metrics about the request and displays them in the web UI. These values are stored in a javascript object which we requested by a post request and turned into a JSON file to be interpreted in Python. These values are:

- Request per second
- Fails per second
- Average response times in percentiles per second
- Number of running users per second

The response times are only given of a randomly chosen number of requests in a certain time window. To make it more accurate we wrote the response times for each request made to a file and gathered them from the locust worker pod once the experiment was finished.
The metric server can be accessed by *kubectl* commands shown in listing 5.

```
kubectl get hpa <HPA name> -n <NAMESPACE> -o jsonpath='{.status.
    currentReplicas}'
kubectl get hpa <HPA name> -n <NAMESPACE> -o jsonpath='{.status.
    currentCPUUtilizationPercentage}'
```

Listing 5: Kubectl HPA data collection

The commands give back the current replicas that are running at a certain time and the CPU utilization that the metric server has calculated. The CPU utilization is given in a unit called milliCPU (m) which specified the share of one CPU, 1 milliCPU is 1/1000 of a CPU [31]. All values are gathered every second when the experiment is running with the use of a bash script.

The bridging functions are running in Kubernetes pods in Ubuntu docker images. These pods therefore have two system files "`/sys/fs/cgroup/memory/memory.usage_in_bytes`" and `/sys/fs/cgroup/cpu/cpuacct.usage"` [41] [42]. The first one shows the usage of memory in bytes and the other one reports the CPU time consumed by all tasks in the cgroup in nanoseconds. Both values are written to a file with a bash script. This bash script is added in the Dockerfile and runs in the background when deploying the pod, by calling it in the "entrypoint.sh" bash script. The files are written to a directory that is mounted on the Kubernetes node, therefore it is still available when the bridging function that generated the file is terminated. At the end of the experiment, these files are Secure Copied (SCP) from the host to the system running the experimentation script.

The CPU time for the kernel file is given in nanoseconds, while the CPU time in the metric server is given in milliCPU. We already said that milliCPU is 1/1000 of a CPU core. One CPU core has 1s of CPU time to divide among processes. Therefore:

$$1\ milliCPU = 1s/1000 = 0.001s = 1\ 000\ 000\ nanoseconds$$

So to convert milliCPU to nanosecond you need to do it times 1 000 000.

8 Experiments

Throughout the project, multiple testing experiments are performed to understand the impact of bridging function scaling on transit traffic and end-user experience. The bridging function that we use is the HTTP filter BF, we chose this one as it does the most processing from all of the ones mentioned in section 5.3. Sixteen testing parameter combinations are defined that we use in our experiments. Diverse users and a fixed spawn rate is used to simulate real live traffic patterns. The test scenarios include 10, 100, 1000, and 10000 users, and the spawn rate per second is defined to be 10% of the total number of users. This way all users are spawned within 10s. For each of the four different user traffic combinations, a mixed HPA scaling threshold is tested to cover a low (30%), high (90%), and moderate (50% and 80%) scaling threshold. These thresholds will change the aggressiveness of the scaling. All the tests were run for 100 seconds with fixed CPU and Memory limits of 1000 milliCPU and 500Mi. The complete list of test parameters combinations is available in appendix A.

9 Results and Discussion

The results of the testing experiments are divided into three sub-areas: scaling because of the bridging function CPU utilization demand with varying CPU thresholds, scaling because of the bridging function Memory usage demand with varying user traffic, and the impact of horizontal scaling on transit traffic in term of response time.

9.1 Short lived vs long-lived connections

The initial test was performed with ten users and Horizontal Pod Autoscaler CPU threshold of 30%. As shown in figure 5, as soon as the traffic started, the bridging function CPU spiked to 1000 milliCPU. As a reaction to the spike in CPU usage, the HPA controller detected a current average CPU utilization of more than 400%. That indicates that the bridging function actually requests almost 1200 milliCPU because the HPA average CPU utilization is represented as a percentage of the requested CPU rather than the used CPU. Kubernetes limits that to 1000 milliCPU, the max limit allowed in the bridging function deployment manifest.

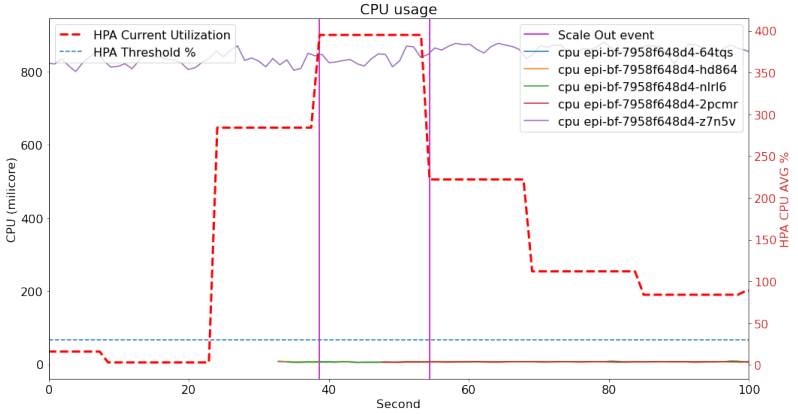


Figure 5: The graph shows the CPU utilization of the bridging functions and the CPU utilization the HPA sees. In this plot, we send HTTP traffic with HTTP Keep-Alive enabled. As can be seen after a scale out event (vertical lines) the CPU is not distributed over the pods as the CPU utilization of new pods is stuck at almost 0.

The increase in the HPA CPU utilization threshold triggered the deployment of more bridging functions to meet the demand. However, as shown in figure 5, the newly deployed have

idle CPU and didn't process any traffic. The unbalanced distribution of load is because the Locust load generator, by default, uses the HttpSession class [43], which performs web requests and holds session cookies between requests. Therefore the Kubernetes service considers all the requests as a single session, and all the traffic is pinned to the initial bridging function. To disable the HTTP persistence session behavior, also known as HTTP keep-alive, the value *"close"* is added to the Connection field of the HTTP header as described in RFC2616 [44]. After adding 'Connection': 'close' to the header Locust, close the HTTP connection after every request completion. After the scale-out event, the connection behavior change leads to a more balanced load distribution across all the deployed bridging functions as shown in Figure 6. The experiments reveal that not all traffic patterns benefit from horizontal scaling. In the results we see that long-lived sessions are not benefiting from horizontal scaling, therefore application traffic needs to be taken into consideration to select the most suitable scaling strategy.

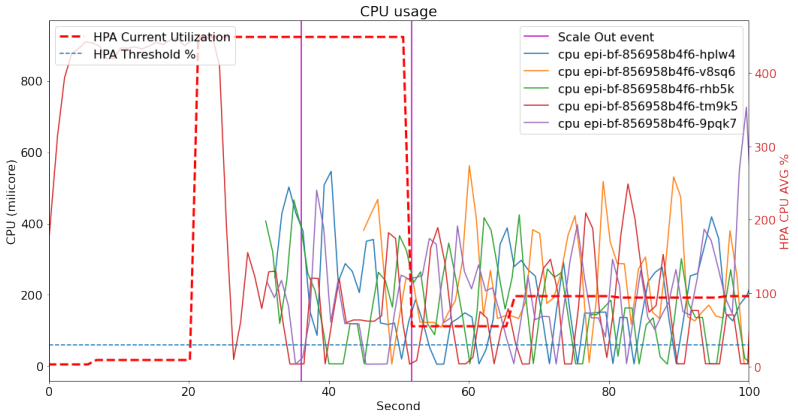
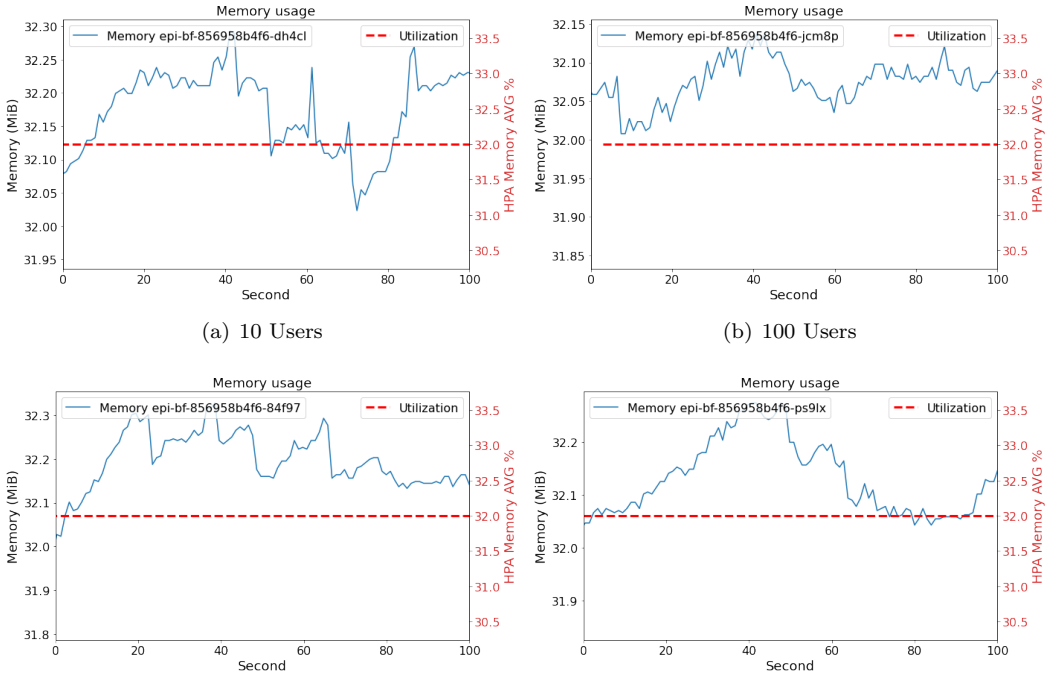


Figure 6: The graph shows the CPU utilization of the bridging functions and the CPU utilization the HPA sees. In this plot, we send HTTP traffic with HTTP Keep-Alive disabled. As can be seen after a scale out event (vertical lines) the CPU is distributed over new pods.

9.2 Static Memory consumption

Another metric that is collected during the testing is the memory usage of the bridging function to understand the impact on scaling based on the memory usage demand. However, as can be seen in figure 7, the memory usage in the bridging function is steady regardless of the number of users spawned. And due to the fractional change in memory usage compared to the total allocated memory of 500 Mebibyte, the HPA utilization is not detecting any change; therefore, no scaling event occurred because of memory demand. The main reason for the memory steady consumption is due to the basic processing logic of the bridging function used in the proof of concept setup. Not every metric can be used for scaling as the metric needs to be a good indication of load. As our bridging function used is not memory intensive, memory is not a good metric to scale on.



(a) 10 Users

(b) 100 Users

(c) 1000 Users

(d) 10000 Users

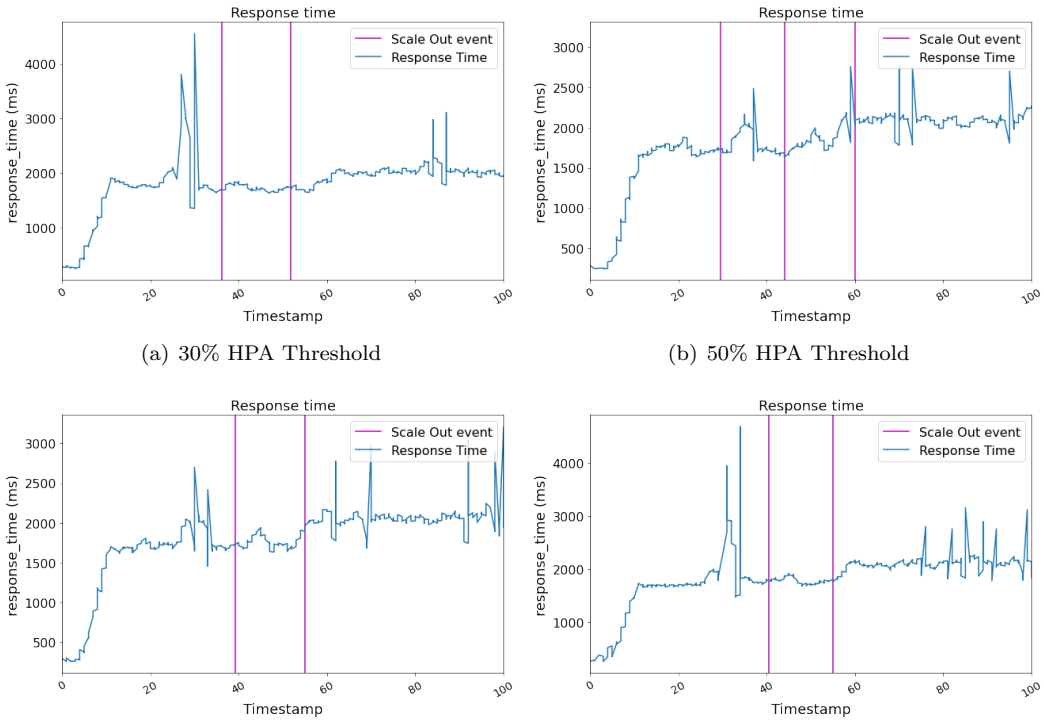
Figure 7: The four graphs show the memory usage of the bridging function's pod and the memory utilization the HPA sees for a different number of users spawned by the load generator. This illustrates that the Bridging function has a steady memory usage regardless of the number of users spawned via the load generator.

9.3 Scaling effect on Application traffic

A critical statistic collected by the Locust load generator is the response time of the HTTP requests in milliseconds. When plotting the response time for different traffic rates including 10, 100, 1000, and 10000 users, as shown in figure 8, two observations are identified.

The response time is very high, more than two seconds on average. The high response time is the processing logic used for the service insertion redirection. The bottleneck is the redirector agent added to the Locust load generator worker pod. With the increase of traffic flowing through the redirector agent to the SOCKS5 proxy, the response time increases dramatically compared to the direct connection when bypassing the service insertion logic. Still, the addition of response time is even among all users and will therefore not interfere with our results.

At the end of the scale-out event, a slight increase of response time is recorded, which is believed to be due to the load balancing overhead added by the Kubernetes service when distributed the traffic across all the deployed bridging functions. Scaling can also have a negative effect on application traffic if scaling is not well optimized.



(a) 30% HPA Threshold

(b) 50% HPA Threshold

(c) 80% HPA Threshold

(d) 90% HPA Threshold

Figure 8: The four graphs show the response time of HTTP request with different HPA thresholds set. This illustrates that a slight increase in response time after a Bridging Function scale-out event occurs.

10 Conclusion

In our proof of concept, we extended the EPI project, by adding autoscaling logic for the bridging function. The experiments we conducted are to test how application traffic's latency is affected when horizontal scaling. This shows if horizontal scaling can be a way for the EPI project to improve the performance of the bridging function by looking at the impact of the Kubernetes Horizontal Pod Autoscaler influence on application traffic's latency.

The results of the experiments revealed that when the traffic is using long-lived sessions, for example, when HTTP keep-alive is enabled, horizontal scaling doesn't add any benefit, as the Kubernetes service pinpoints the traffic to a single pod. We assume that Vertical Pod Autoscaling could be a better fit, as it could update the resources of a bridging function while it is running, which means it can update the single pod that is active in the traffic path of long-lived sessions. However, more testing is required. We can, for example, test this by having an HTTP keep-alive session increase the number of requests and scaling the single pod vertical and see the impact of the application traffic's latency.

Also when reviewing the testing results, it can be concluded that the metric to trigger the scaling of the bridging function is highly dependent on the processing logic. In the proof of concept scenario, the bridging function used is CPU-centric. Therefore, the horizontal autoscaler can use CPU utilization as the primary metric to decide if more bridging functions are required to meet the traffic demand. On the flip side, the memory usage was very steady during testing regardless of the traffic rate pushed by the load generator.

The results also unfold that scaling the bridging function can directly impact the traffic and user experience due to the increase in response time. The delay added to the HTTP traffic response is because of the overhead added by the service insertion redirection agent and the Kubernetes service load balancing. More optimization is needed on both elements to reduce the impact on the response time.

11 Future Work

There is always room for some improvements. In our research, there was not a clear best threshold to use. This is mainly because finding such a threshold is use case related. Therefore this is a perfect problem that can be solved by machine learning, preferably online learning algorithms. That can change the best threshold by looking at the current usage, but also historical data.

Then another thing that can be improved is that we did not look at different application traffic. For now, we only looked at HTTP requests. This can be extended to be other applications such as FTP or even HTTP streams, which will be handy for the EPI project. As healthcare data can be big data sets that need to be distributed or video streaming between patient and doctor. Based on different application traffic other factors might be found that could hinder or improve the scaling of the network functions.

Our bridging functions used in this research are very minimal and not well optimized. When using production-grade network functions and containerize them the results may vary.
In our related work, we mentioned that VNF can be chained together and that it was found out that scaling VNF while keeping in mind where the scaling takes place in the chain can have performance benefits. Testing this on containerized network functions in our system would be a step in improving the performance.

Then in this report, we mentioned also a VPA and a way to use custom metrics. For

both, it would be interesting to further research them. We did not have the time to work on it. More research into how custom metrics can improve autoscaling is still needed. But for the VPA it would be interesting to see how it can complement HPA, in our research into the topic and also mentioned in the report, is that VPA can improve performance for long-lived and stateful sessions. Trying to change the use cases to such application and see if the system can itself see when HPA or VPA is needed. This makes the whole EPI system more dynamic in its scaling department. Being dynamic is one of the goals that the EPI project has.

12 Acknowledgments

We thank our supervisor Jamila Alsayed Kassem for her time and guidance throughout this research project.

References

[1] Cees de Laat. URL: https://enablingpersonalizedinterventions.nl/.

[2] Jamila Alsayed Kassem et al. "The EPI Framework: A Dynamic Data Sharing Framework for Healthcare Use Cases". In: *IEEE Access* 8 (2020), pp. 179909–179920. DOI: 10.1109/ACCESS.2020.3028051.

[3] *Scaling Horizontally vs. Scaling Vertically.* URL: https://www.section.io/blog/scaling-horizontally-vs-vertically/. (accessed: 31.05.2021).

[4] Andanagouda Patil. *SOCKS Proxy Primer: What Is SOCKs5 and Why Should You Use It?* Sept. 2019. URL: https://securityintelligence.com/posts/socks-proxy-primer-what-is-socks5-and-why-should-you-use-it/.

[5] Duc-Hung Luong et al. "Cloudification and Autoscaling Orchestration for Container-Based Mobile Networks toward 5G: Experimentation, Challenges and Perspectives". In: *2018 IEEE 87th Vehicular Technology Conference (VTC Spring)*. 2018, pp. 1–7. DOI: 10.1109/VTCSpring.2018.8417602.

[6] Salman Taherizadeh and Marko Grobelnik. "Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications". In: *Advances in Engineering Software* 140 (2020), p. 102734. ISSN: 0965-9978. DOI: https://doi.org/10.1016/j.advengsoft.2019.102734. URL: https://www.sciencedirect.com/science/article/pii/S0965997819304375.

[7] Omar Houidi et al. "An Efficient Algorithm for Virtual Network Function Scaling". In: *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*. 2017, pp. 1–7. DOI: 10.1109/GLOCOM.2017.8254727.

[8] Hong Tang, Danny Zhou, and Duan Chen. "Dynamic Network Function Instance Scaling Based on Traffic Forecasting and VNF Placement in Operator Data Centers". In: *IEEE Transactions on Parallel and Distributed Systems* PP (Aug. 2018), pp. 1–1. DOI: 10.1109/TPDS.2018.2867587.

[9] Duc-Hung LUONG et al. "Predictive Autoscaling Orchestration for Cloud-native Telecom Microservices". In: *2018 IEEE 5G World Forum (5GWF)*. 2018, pp. 153–158. DOI: 10.1109/5GWF.2018.8516950.

[10] Asif Mehmood et al. "Energy-efficient auto-scaling of virtualized network function instances based on resource execution pattern". In: *Computers Electrical Engineering* 88 (2020), p. 106814. ISSN: 0045-7906. DOI: https://doi.org/10.1016/j.compeleceng.2020.106814. URL: https://www.sciencedirect.com/science/article/pii/S0045790620306650.

[11] Steven Van Rossem et al. "VNF Performance modelling: From stand-alone to chained topologies". In: *Computer Networks* 181 (2020), p. 107428. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2020.107428. URL: https://www.sciencedirect.com/science/article/pii/S1389128620311178.

[12] Adel Nadjaran Toosi et al. "ElasticSFC: Auto-scaling techniques for elastic service function chaining in network functions virtualization-based clouds". In: *Journal of Systems and Software* 152 (2019), pp. 108–119. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2019.02.052. URL: https://www.sciencedirect.com/science/article/pii/S0164121219300421.

[13] *Xen Project*. URL: https://xenproject.org/.

[14] *Kubernetes and Container Security and Adoption Trends*. URL: https://www.stackrox.com/kubernetes-adoption-security-and-market-share-for-containers/.

[15] *Metallb Load balancer*. URL: https://metallb.universe.tf/.

[16] *flannel Project*. URL: https://github.com/flannel-io/flannel#flannel.

[17] *K8S setup automation*. URL: https://github.com/mohanadelamin/k8s-setup-automation.

[18] *Kuberentes Services*. URL: https://kubernetes.io/docs/concepts/services-networking/service/.

[19] *Locust Load Generator*. URL: https://locust.io/.

[20] *epi-bf-helm Chart*. URL: https://github.com/mohanadelamin/epi-bf-helm.

[21] *Locust helm chart*. URL: https://github.com/deliveryhero/helm-charts/tree/master/stable/locust.

[22] *Glasgow Network Functions*. URL: https://netlab.dcs.gla.ac.uk/projects/glasgow-network-functions.

[23] *NetfilterQueue*. Jan. 2017. URL: https://pypi.org/project/NetfilterQueue/.

[24] Simon Jouet, Richard Cziva, and James Guthrie. *UofG-netlab/gnf-dockerfiles*. Sept. 2015. URL: https://github.com/UofG-netlab/gnf-dockerfiles.

[25] Rakhesh Sasidharan. *Iptables packet flow (and various others bits and bobs)*. Nov. 2020. URL: https://rakhesh.com/linux-bsd/iptables-packet-flow-and-various-others-bits-and-bobs/.

[26] Olaf Kirch and Terry Dawson. *IP Masquerade and Network Address Translation*. June 2000. URL: https://www.oreilly.com/openbook/linag2/book/ch11.html.

[27] *Horizontal Pod Autoscaler*. URL: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. (accessed: 01.06.2021).

[28] *Horizontal Pod Autoscaler V2*. URL: https://github.com/kubernetes/community/blob/master/contributors/design-proposals/autoscaling/hpa-v2.md. (accessed: 14.06.2021).

[29] *Kubernetes Metrics*. URL: https://github.com/kubernetes/metrics/blob/master/IMPLEMENTATIONS.md#custom-metrics-api. (accessed: 14.06.2021).

[30] Povilas Versockas. *Vertical Pod Autoscaling: The Definitive Guide*. Jan. 2021. URL: https://povilasv.me/vertical-pod-autoscaling-the-definitive-guide/.

[31] *Managing Resources for Containers*. Feb. 2021. URL: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#meaning-of-cpu.

[32] Kubernetes. *Vertical Pod Autoscaler*. URL: https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler.

[33] Michael Handa. *How to Scale Using Kubernetes: From Startup to Superstar*. May 2020. URL: https://bluesentryit.com/how-to-scale-using-kubernetes-from-startup-to-superstar/.

[34] *Configuring multidimensional Pod autoscaling*. URL: https://cloud.google.com/kubernetes-engine/docs/how-to/multidimensional-pod-autoscaling.

[35] Lucas Käldström and JuanJo Ciarlante. *luxas/kubeadm-workshop*. URL: https://github.com/luxas/kubeadm-workshop.

[36] *kubernetes-sigs/prometheus-adapter*. 2021.
URL: https://github.com/kubernetes-sigs/prometheus-adapter.

[37] Prometheus. *Prometheus - Monitoring system time series database*.
URL: https://prometheus.io/.

[38] *Welcome to Flask¶*. URL: https://flask.palletsprojects.com/en/2.0.x/.

[39] Prometheus. *Exposition formats: Prometheus*.
URL: https://prometheus.io/docs/instrumenting/exposition_formats/.

[40] Prometheus. *Recording rules: Prometheus*.
URL: https://prometheus.io/docs/practices/rules/.

[41] *memory*.
URL: https://www.kernel.org/doc/Documentation/cgroup-v1/memory.txt.

[42] *3.3. cpuacct Red Hat Enterprise Linux 6*.
URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/sec-cpuacct.

[43] *Locust HttpSession class*.
URL: https://docs.locust.io/en/stable/api.html#httpsession-class.

[44] *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. RFC Editor.
URL: https://www.rfc-editor.org/rfc/rfc2616.txt.

Appendices

A Testing Scenarios

The following table contains the list of the test scenarios performed along with the parameters used:

Test No.	Number of Users	Spawn Rate	Run time (Sec)	HPA Max Replicas	HPA Utilization %	BF CPU Limit	BF Memory Limit
1	10	1	100	5	30	1000m	500Mi
2	10	1	100	5	50	1000m	500Mi
3	10	1	100	5	80	1000m	500Mi
4	10	1	100	5	90	1000m	500Mi
5	100	10	100	5	30	1000m	500Mi
6	100	10	100	5	50	1000m	500Mi
7	100	10	100	5	80	1000m	500Mi
8	100	10	100	5	90	1000m	500Mi
9	1000	100	100	5	30	1000m	500Mi
10	1000	100	100	5	50	1000m	500Mi
11	1000	100	100	5	80	1000m	500Mi
12	1000	100	100	5	90	1000m	500Mi
13	10000	1000	100	5	30	1000m	500Mi
14	10000	1000	100	5	50	1000m	500Mi
15	10000	1000	100	5	80	1000m	500Mi
16	10000	1000	100	5	90	1000m	500Mi

Table 1: Testing Senarios

B Bridging functions

The following includes the docker file, redirection script, and monitoring script of the bridging function used in this project.

```
FROM ubuntu:20.04

ENV DEBIAN_FRONTEND=noninteractive

RUN apt-get update && apt-get install --no-install-recommends -y \
    python3 \
    python3-pip \
    iptables \
    iproute2 \
    vim \
    net-tools \
    tcpdump \
    sudo \
    dnsutils \
 && rm -rf /var/lib/apt/lists/*

ADD rules.sh rules.sh
ADD monitor.sh monitor.sh
ADD entrypoint.sh entrypoint.sh

ENTRYPOINT [ "./entrypoint.sh"]
```

Listing 6: Docker image of the http filter bridging function

```
#!/usr/bin/env bash
EPI_SERVER_VAR=‘host $EPI_SERVER | awk ’/has address/ { print $4 ; exit }’‘
iptables -t nat -A PREROUTING -p tcp -j DNAT --to-destination $EPI_SERVER_VAR:
    $EPI_SERVER_PORT
iptables -A FORWARD -j NFQUEUE --queue-num 1
iptables -t nat -A POSTROUTING -j MASQUERADE
nohup bash /monitor.sh &
./main.py
```

Listing 7: entrypoint.sh

```
#!/usr/bin/env bash

HOSTNAME=$(hostname)
FILE=${1:-/mnt/$HOSTNAME.txt}

echo ’"Time","CPU","Memory"’ > $FILE

while true; do
    sleep 1;
    read -rst5 MEMORY </sys/fs/cgroup/memory/memory.usage_in_bytes
    read -rst5 CPU    </sys/fs/cgroup/cpu/cpuacct.usage
    read -rst5 DATE   < <(date +"%FT%T.%3N")
    echo "$DATE,$CPU,$MEMORY";
done >> $FILE;
```

Listing 8: monitor.sh

C Source code

The source code can be found on the following GitHub page: `https://github.com/mohanadelamin/rp2-epif`