Expertise
and insight
for the future

Ninja Luotonen

# Stability and Stress Testing

Metropolia University of Applied Sciences

Information Technology

Software Engineering

Bachelor's Thesis

9.5.2021

| | |
|---|---|
| Author<br>Title<br><br>Number of Pages<br>Date | Ninja Luotonen<br>Stability and Stress Testing<br><br>27 pages<br>9 May 2021 |
| Degree | Bachelor of Engineering |
| Degree Programme | Information technology |
| Professional Major | Software Engineering |
| Instructors | Matti Oosi, Principal Lecturer |

The goal of the study was to design and implement load and stress tests that will provide meaningful information for the commissioner and help with recognizing performance issues before they create problems in customer deployments. In addition, a stability testing environment was implemented, which will be used for a long running installation to monitor resource usage and stability, and to develop monitoring solutions.

The background and value of the product and key technologies used are explained. The tested product is an event-driven network automation solution for operators. It is based on Apache Kafka, which is an open-source stream processing platform developed by Apache Software foundation.

Apache JMeter, an open-source load testing tool, also developed by the same foundation, was used for testing. The tests need to simulate a realistic scenario, sending many events to be processed by the application and verify whether it is working as intended even under heavy load.

As the product is deployed on Kubernetes, the tests need to be run automatically in the Kubernetes cloud as a part of the CI/CD pipeline and publish the results.

| | |
|---|---|
| Keywords | Software testing, Stress testing |

## Contents

List of Abbreviations

| | |
|---|---|
| KaaS | Kubernetes as a Service. Internal service at Elisa, providing Kubernetes clusters and administration tools. |
| UDP | User Datagram Protocol. A low latency protocol that does not acknowledge that packets are sent or have been received. This can result to lost packets as a trade-off for lower latency. |
| SNMP trap | Simple Network Management Protocol. Traps are asynchronous alert messages that are sent by network devices over UDP. |
| CI/CD | Continuous Integration, Continuous Development. |
| DNS | Domain Name System. The system from websites to use names instead of IP-addresses. |
| API | Application Programming Interface. Interface which is used by users to make requests. |
| VM | Virtual Machine. Virtualized computer. |
| NOC | Networks Operations Centre. |

# 1 Introduction

In today's world wireless communication is used more than ever. Customers demand a fast and stable mobile network connection for their devices. Finns use more mobile data per subscription than any other nation. [1]

As more of internet usage shifts to mobile devices, telecoms have to improve their infrastructure to support more simultaneous users with further demanding network usage. The rollout of new high frequency connections require base stations to be more densely packed and have optimized locations. More base stations mean a higher number of network errors and alarms that need to be resolved swiftly to minimize the impact on customer experience. Elisa has started the automation of correcting these faults many years ago, and now Elisa Automate is creating and offering automated solutions to other teleoperators.

The goal of the Elisa Automation Virtual NOC solution is to automate the alarm handling process through pre-defined network actions and incident management processes in order to reduce telecommunication operational costs significantly. The software has to be stable in all situations, which means for instance that it has to be able to handle a significant number of alarm spikes which can happen for example during a storm.

This thesis is about testing the stability and performance of Virtual NOC. The software is deployed in Kubernetes which is also used a lot in the continuous development pipeline, thus, background information about Kubernetes is presented. Investigation of the testing need is explained and the requirements for testing are specified.

Furthermore, the implementation plan and tools used for the stress tests are presented. The implementation of the stability test environment and continuous development pipeline is presented with the results and conclusions, together with the future plans and improvement ideas.

# 2 Environment

This chapter provides an overview of the traditional Network Operating Centre, which the Virtual NOC solution is aiming to replace. Next Virtual NOC and its core components are presented. Lastly containers and Kubernetes are explained, which are in a key role in Virtual NOC development and operation.

## 2.1 Traditional Network Monitoring

Many professionals are using mobile networks for their work, instead of being confined to an office. Simultaneously, file transfers, video streaming for entertainment and video calls are using up bandwidth. Customers are relying on having a more stable and faster connection on their mobile devices than ever before, and as transmission frequencies become higher to offer greater transfer speeds, the signal is easily obstructed by buildings and objects. This requires larger number of base stations that devices connect to, which in turn makes the overall network infrastructure larger, and there are more errors to resolve.

Traditionally the error management has been done by engineers in big rooms with multiple screens for each engineer monitoring the network status (See Figure 1). This is called first-line Network Operation Centre (NOC). The network requires 24/7 monitoring, and the NOC is often outsourced. When a base station sends an alarm that something is wrong, an engineer takes it under inspection and depending on what specific alarm it is, e.g., node power failure, they can do certain actions, for example reset the base station. If the usual actions do not fix the issue, they can create a ticket to second-line operations with more detailed information, and the second-line engineers might for example call field services to go on site for troubleshooting and reparation.

Figure 1. Network Operation Centre. [2]

As the work is done by humans, the corrections always have some unnecessary delay due to for example learning the job, breaks, human error etc. Many of the alarms and the actions that fix them are routine work, which means they could easily be automated. Other alarms require finer troubleshooting, and an unexperienced NOC engineer could take a long time to figure it out or ask a senior for advice. All of these result in unnecessary delays and impacts to customer experience.

## 2.2 Virtual NOC Solution

Virtual NOC can completely automate the first-line NOC. This leads to faster fault resolution times as well as reduced costs. Virtual NOC executes corrective network actions, creates second line tickets to Network experts and Field Maintenance and provides information to Cyber Service Operations Centre related to security. See figure 2.

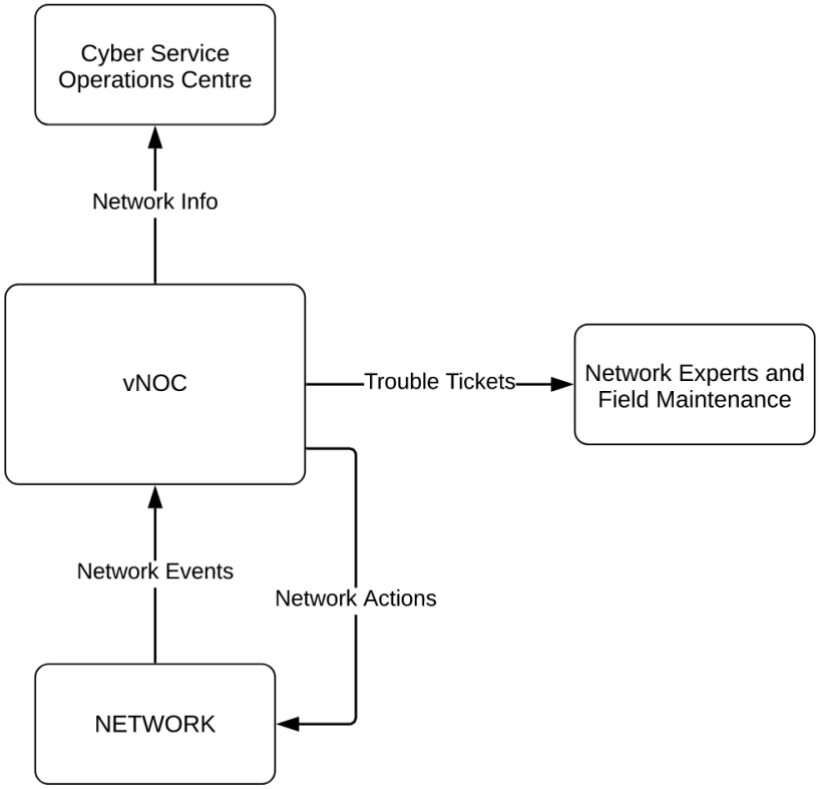


Figure 2. System Context.

Virtual NOC or Virtual Network Operations Centre collects data from the network. Automated algorithms are applied on the data and corrective actions are taken to fix problems in the network and/or problem tickets are created in the client's ticket system, e.g. Jira, ServiceNow or similar. See Figure 3.

Information about the network status can also be sent to Service Operations Centre systems. Other customer specific integrations are also possible.

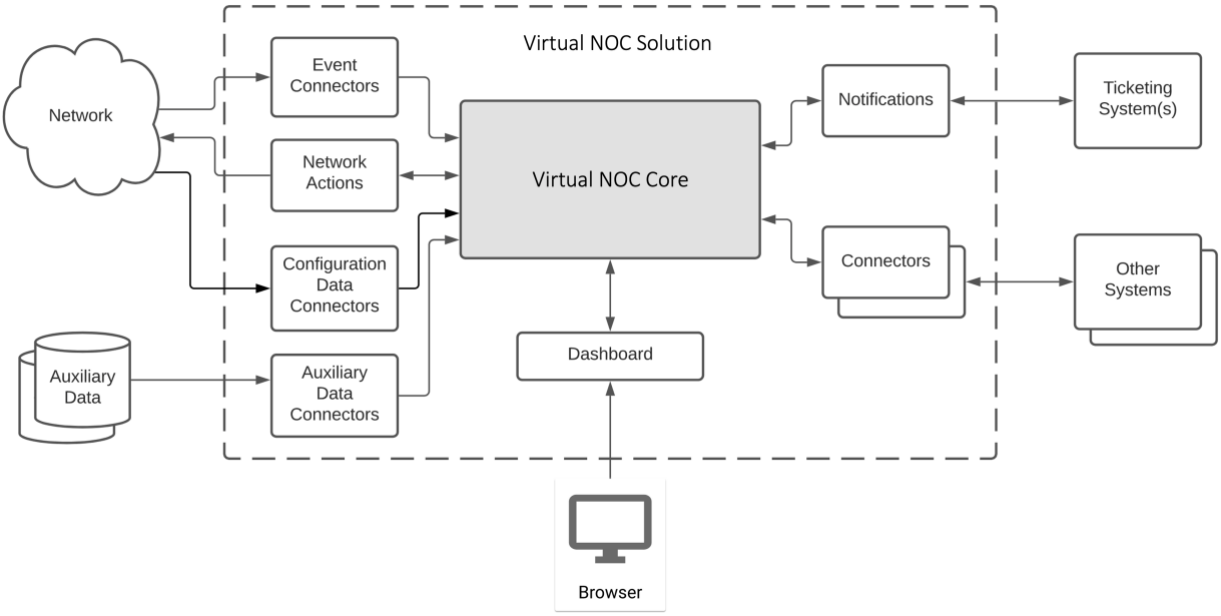


Figure 3. System Overview.

Virtual NOC is built from multiple components that are operated separately and integrate with each other. The components run in containers and are deployed on a Kubernetes cluster.

Virtual NOC Core

The core contains the event streaming and transformation pipeline, the rule engine and action managers. Network events are processed in the rule engine and necessary actions are initiated.

Event Connectors

The event connectors are for receiving network events. Currently SNMP traps are supported with the open sourced kafka-connect-snmp. This is where the tests will send in events to the Virtual NOC Core.

### Data Connectors

Data connectors handle incoming non-event-type data, e.g., network inventory or configuration data.

### Other Connectors

Customer-specific integration needs to other systems can be implemented with either kafka-connect or bespoke integration components.

### Network Actions

Necessary corrective actions to the network are sent via vendor-specific adapters.

### Notifications

Alarm notifications are sent to customer-specific trouble ticket systems via application-specific adapters. Current implementations are for Jira and EasyVista.

### Dashboard

Virtual NOC Dashboard is for visualization and monitoring the autonomous algorithms. There are also switches for disabling network actions, ticketing actions and algorithms altogether.

There are quite many components, but the most critical ones are event connectors and Virtual NOC Core. The Core has a couple separate components, such as the rule-engine, which is where the correcting decisions are made, and alarm enrichment module, which processes the events to be easily used in the rule-engine. The next chapter explains more about the enrichment process.

## 2.3 Alarm Enrichment

For Virtual NOC to be able to make decisions it needs to interpret data from alarms and events coming in from the network. The information in alarms is very unstructured. Different vendors use different structures, and for example specific alarm types, software version, the operator's site/cell naming convention - the location of specific information can be very different.

For example:

for Alarm type 1 the site name can be found in the first part of ***alarmMOName*** field:

**SITENAME**_SITEID-Cabinet No.=0, Subrack No.=73, Slot No.=0,0-73-0

for Alarm type 2 site name can be found as part of a Technology name in ***alarmExtendInfo*** field:

> RNC ID=123, Cell ID=12345, Cell Name=**SITENAME**W2, NodeB ID=12345, NodeB Name=**SITENAME**W, Local Cell ID=123, Alarm Cause=NodeB sends CELL SETUP FAILURE (Unspecified other cause).

There are plenty of information that are in a somewhat structured format (e.g. alarmID), however, a number of key information are scattered in unstructured fields. There is a need to extract such key information to allow for easy processing. For this specific vendor, for example alarmAdditionalInfo and alarmExtendInfo are unstructured fields with various information.

The goal of the restructured fields is to have the same fields across different operators and different vendors, in order to allow for easy code generalisation.

For ease of further processing, the content of the events are processed when the events arrive in Virtual NOC, and each event is enriched with a number of key information points. This enrichment provides a structured dataset to allow for easy data analysis. As such the enrichment information is attached to each event, so enrichment and original event

can be easily extracted together for analysis purpose. An example on how enrichment information can be different between vendors can be seen in Table 1.

Table 1. Information needed from the alarms.

| Information | Huawei | Ericsson | Notes |
|---|---|---|---|
| Affected Site Name | X | | |
| Affected Node Name | X | X | 2G, 3G, 4G have different node names (Also called "Function Name") |
| Affected Cell Name | X | X | |
| Affected Cell ID | X | X | |
| Affected Equipment | | X | |
| Cabinet | X | | |
| Subrack | X | | |
| Slot | X | | |
| Port | X | | |
| ? | | | future |
| ? | | X | future |

As explained earlier, alarms from different vendors have different fields. In Table 1 a few fields from Huawei and Ericsson are compared. For example, Huawei has Cabinet, Subtrack, Slot and Port fields, while in Ericsson those do not exist. Ericsson has instead Affected Equipment, which has the equivalent information as the Huawei fields mentioned.

## 2.4 Kubernetes

Kubernetes is a popular open-source container-orchestration system for automating computer application deployment, scaling and management. Containers are a popular way of running micro-services. Virtual NOC team has used Kubernetes for development and deployment, and it has served the needs of the product and team well.

### 2.4.1 Containers

Early on, software applications were deployed directly on machines, and because there were no resource allocation possibilities, this led to often to problems, for example one application would use up most of the memory available and other applications would underperform. Having a separate computer for each application would ensure each application has enough resources available, but it was an expensive solution and would result in low overall resource utilization.

Next was the virtualized deployment era. Each physical machine could be split up as many virtual ones, which allowed resources to be allocated and utilized better, as well as improved scalability, as applications can be added or updated easily. Improved resource utilization led to reduced hardware costs. This also provided security, as applications in separate VMs were isolated and could not be freely accessed by another application.

After virtualization, containers were started to be developed. Containers are similar to VMs, but they share the operating system between the applications and have more lenient isolation properties. Thus, they are regarded lightweight. A container has its own share of CPU, memory, file system, process space, etc. See figure 5 for some high-level architecture comparisons.

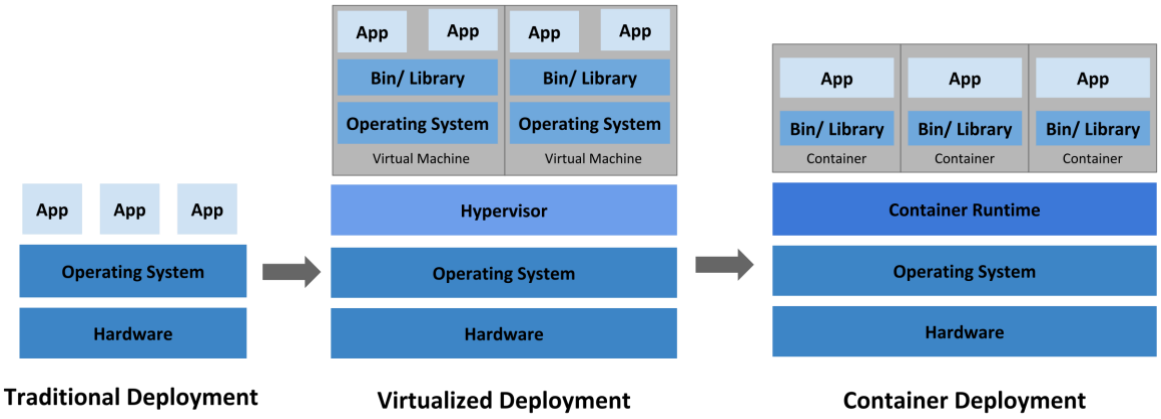


Figure 5. Evolution of deployments. [3]

Benefits of containers are for example:

- Container images are portable, efficient, easy to create and deploy compared to VM images.

- Environmental consistency, containers run the same on laptops as it does in the cloud, applications behave the same way across development, testing and production.

- Fits in well with continuous development, integration, and deployment. Accommodates for reliable and frequent container image builds and deployments.

- Predictable application performance due to resource isolation. High resource utilization. [3]

### 2.4.2 Kubernetes

Kubernetes is a popular open-source system that automates container management. It is a tool for application deployment, management and scaling.

Kubernetes was originally developed by Google, and in 2015 Kubernetes v1.0 was released. In 2015, Google and the Linux foundation partnered to form Cloud Native Computing Foundation, to help advance container technology and align the technology industry around the evolutionary development of containers.

When deploying Kubernetes, one gets a cluster. A cluster consists of nodes, which are a set of worker computers. Every cluster needs at least one node. The nodes then host pods, which are the components of the application workload. Each application will be run in a pod. The cluster also has a control plane, which makes decisions about the cluster. It detects issues and responds to cluster events. For example, scheduling, starting up new pods and scaling. [3] See figure 6 on how the cluster components are tied together.

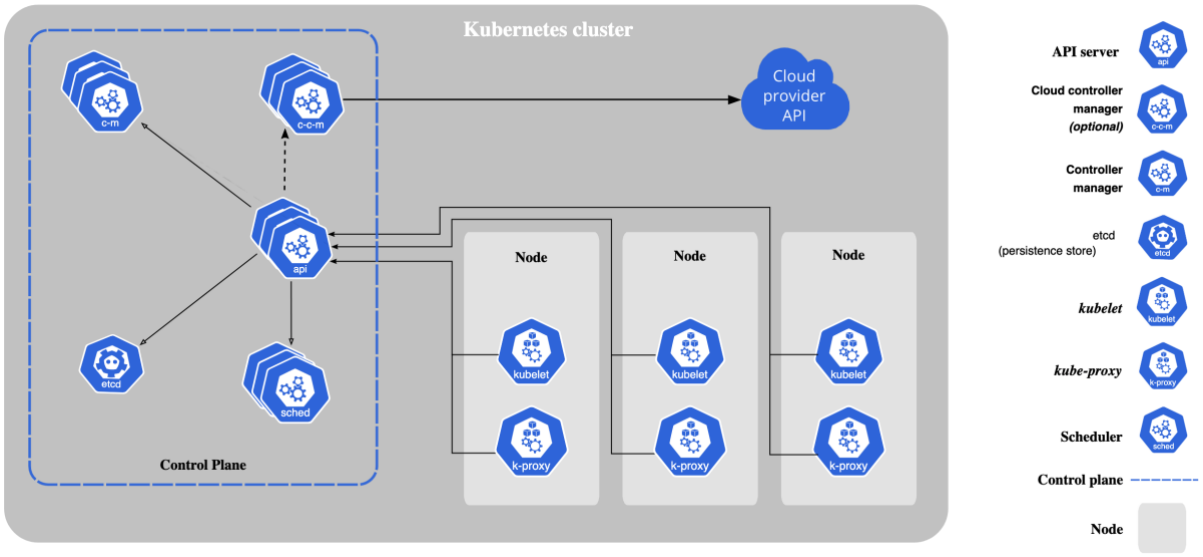


Figure 6. Components of a Kubernetes Cluster. [4]

Kubernetes offers for example the following.

- **Service discovery and load balancing** With Kubernetes one can expose a container using a DNS or their own IP address. If traffic to a container is high, Kubernetes can balance the network load to keep the application deployment stable.

- **Automatic bin packing** The user provides Kubernetes with a cluster of nodes, tell how much CPU and RAM each container needs, and Kubernetes fits the containers onto the nodes using the available resources efficiently.

- **Self-repairing** Kubernetes restarts containers that fail, replaces non-functional containers, does not forward clients to them before they are ready to serve. One can define custom health checks; Kubernetes will restart the containers which fail to respond to them.

- **Secret and configuration management** Kubernetes lets one store and manage sensitive information, for example passwords, OAuth tokens, and SSH keys. Deployment and updates of secrets and application configurations are done securely without rebuilding ones container images.

In addition, Kubernetes provides storage orchestration, automated rollouts and rollbacks and more. Kubernetes is not a traditional Platform as a Service.

Kubernetes aims to support an exceptionally diverse variety of workloads, if an application can run in a container, it should run well in Kubernetes. Kubernetes does not deploy source code, nor does it build ones application. There is no need to create the continuous integration, delivery and deployment workflows. Kubernetes does not appoint logging, monitoring or alerting solutions; however, it does provide mechanisms to collect and export metrics. [3]

# 3 Software Testing

Software testing is a process to verify and validate that the software meets the requirements specified. There are many kinds of testing but here only the most common one is introduced. Commonly used are 3 or 4 levels of functional testing: unit testing, integration testing, system testing and acceptance testing, the latter of which is not always included. In addition, there is non-functional testing, which is for verifying the performance, security, compatibility, and more, of the software. [5]

## 3.1 Static and Dynamic Testing

Static testing happens before executing the code, while dynamic testing happens during execution. There are several branches to different types of testing (Figure 7)

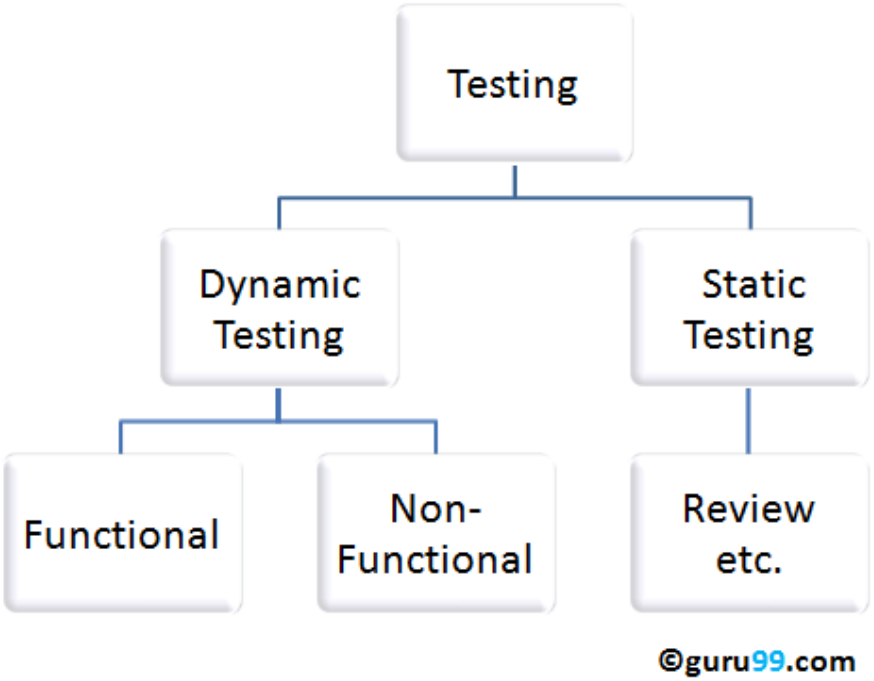


Figure 7. Testing branches on a high level. [5]

Under static testing, there is for example peer review, automatic formatting and style checks. Under dynamic testing, there is functional and non-functional testing. Functional means that the aim is to see if the software works according to specifications, for example a calculator needs to be able to take input, have the necessary buttons and calculate the mathematics correctly based on functions. Non-functional testing does not look at if the calculator counts correctly, but instead at for example how long it takes for the calculation to happen.

### 3.1.1 Static Testing

Manual and automated review of the code and the documentation is part of static testing and is done to find any defects early in the development process while the time consumed fixing them is the least. It is a common development practice to have a peer review process, which means the author and another developer checks the code to verify quality and technical correctness of all new code, before merging it to the codebase. In a study it was found that peer review has a "favourable return on investment for software

inspections; savings exceeds costs by 4 to 1". In other words, on average, fixing the problem only later is four times more costly. [6]

### 3.1.2 Dynamic Testing

There are many types of testing that can be performed when the software is executed, under functional testing there is for example, Unit testing, Integration testing, System testing and Acceptance testing. Under non-functional testing there is Performance testing, Security Testing, Compatibility testing and Recovery testing. [8] See figure 8 below.

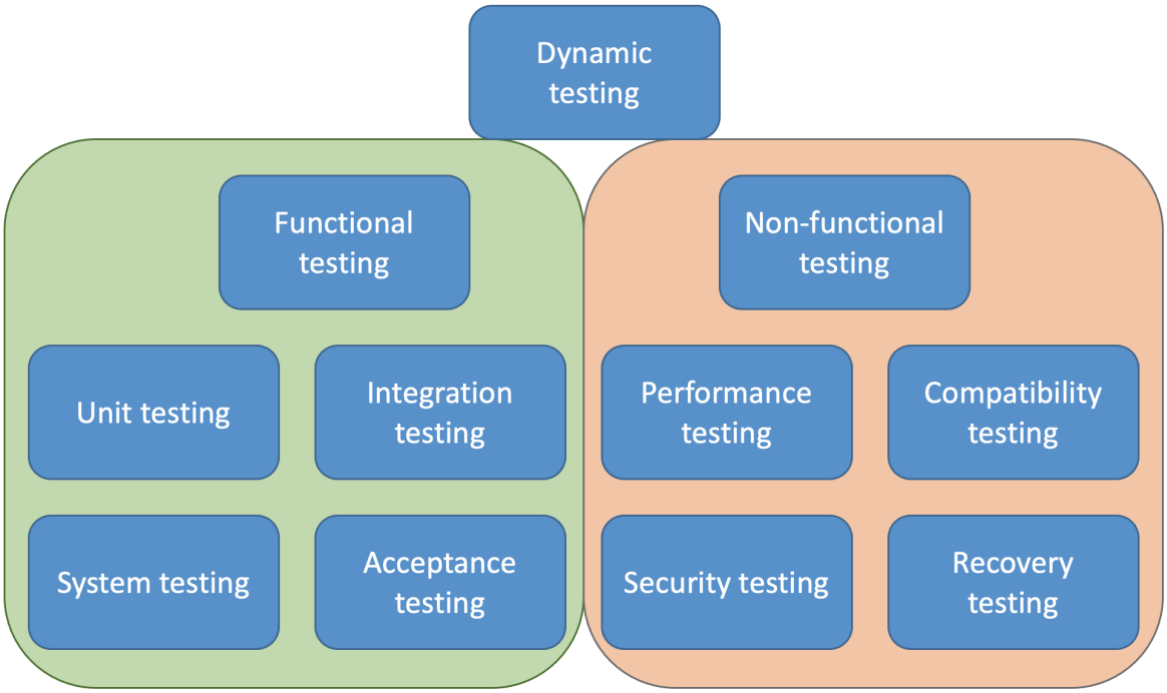


Figure 8. Dynamic testing types.

Unit testing is the lowest level of testing for code. It is for verifying that individual units, such as classes and functions work correctly. Unit tests are written by developers. It is considered a good practise to write unit tests for any new code and modify them when changing the code. This helps developers to make sure that their code does what it is supposed to and helps with narrowing down which part of the code is faulty. When

refactoring code, meaning re-writing code without changing functionality, they work as a good regression test suite. [8]

Integration testing is about testing modules of code together. Often different components in a program are programmed by different developers, and they might have a different understanding of how the system should behave together. There can be miscommunications and wrong assumptions made how the modules are going to work together. Software and hardware interfaces could be erroneous, and error handling might not be adequate. Integration testing tests two or more components together. [9]

### 3.1.3 Performance Testing

Performance testing is a software testing process used to test and verify the speed, response time, stability, reliability and resource usage under a set workload. The main goal is to identify and remove performance bottlenecks in the application. Different kind of software have different sets of performance requirements, for example a web store that is publicly available could have tens of thousands of users simultaneously, while a desktop application with the software running locally on each user's computer will only have one user at a time who expects good response times. [10]

# 4 Virtual NOC Testing Requirements

Virtual NOC is in charge of automating the fault management of customer operators' mobile network, which send in a varying number of events and alarms that the software will evaluate and do actions based of. The actions can be for example, creating a ticket to Field Maintenance service, or resetting a base station. A crucial piece of this process is the SNMP-connector that forwards the traps into event enrichment. The enrichment restructures and adds data to the events. If the connector crashes, the rest of the software cannot get the data needed to provide value to the customers.

In addition, as the software uses Apache Kafka for almost all internal communication, it will be the priority of load and performance testing. Stability testing is also on the table, as Virtual NOC is an autonomous system, it should not require human oversight and thus

be stable enough to function autonomously between releases. Each component needs to be able to recover from errors, restart and reconnect to other components.

Components to be tested next, are the web UI with an API, and rule-engine. The HTTP REST API of the rule-engine exposes the information needed for visualization on the web interface.

## 4.1 Stress Testing

The stress testing environment should use production configurations. The aim is to see how the system behaves in case of a large number of network events, which could happen for example due to a storm. The tests include a constantly rising number of events, and sudden spikes in events. For consistent results, the test suite should create the environment before the tests and remove it afterwards. It is also relevant to consider network usage, as the amount of data received can become substantial in larger networks.

## 4.2 Test Definitions and Metrics

To measure performance, metrics are needed. Metrics should be meaningful in a real world application; improved results should mean a better user experience or higher customer value by for example increasing stability and uptime.

Metric 1. Percentage of events processed successfully

Metric 1 is measured by controlling the quantity of events sent into the system, and then counting the number of processed events from the database. It is expected that the percentage will go down if the system load is too high, and finding out what are the current limits of the system is one of the main objectives of the performance testing. This metric can be used in combination with other metrics, to investigate if this percentage stays the same when other variables in the system changes, for example the amount of processes active.

Metric 2. System resource usage

The deployed system uses CPU, memory, and disk space. It is important to know how the resource usage scales between lower and higher traffic networks, and how the number and complexity of algorithms affect the usage.

Test 1. Stress testing the enrichment pipeline

Starting with a typical load seen in customer networks, and then incrementing it higher while observing metrics 1 and 2, the number of events successfully processed is measured, and peak resource usage is documented.

Test 2. Stress testing spike scenarios

Stress testing spike scenarios mean starting with a typical load to get the system running and use resources in a normal fashion, then sending 2 to 3 times more events simultaneously. This could happen in real life for example during a storm which causes power outages and other issues. The exact numbers will be tweaked through experimentation, to have tests for both overload and manageable scenarios.

# 5 Implementation

This chapter introduces the performance testing tools, and the Kubernetes testing environment, and the implementation of integration to the continuous integration and development (CI/CD) pipeline.

## 5.1 Performance Testing Tools

As Virtual NOC is programmed in Kotlin, a Java-based programming language, and open-source tools are preferred. It is also preferred that the tests would not require programming skills, so that testers without programming experience can also create and modify the tests. Visual reports that can be easily understood by less-technical personnel is also a bonus, they can of course be created as well but at least basic reports out of the box is nice to have.

Apache JMeter was chosen for the implementation of the tests. It has plenty of plugins to use for different usage, and if needed new plugins can be created as well. As it is a popular tool, there are also a lot of material and discussions online to aid the usage and does have the other features mentioned above. It has an easy to use graphical interface as well. (See figure 9)

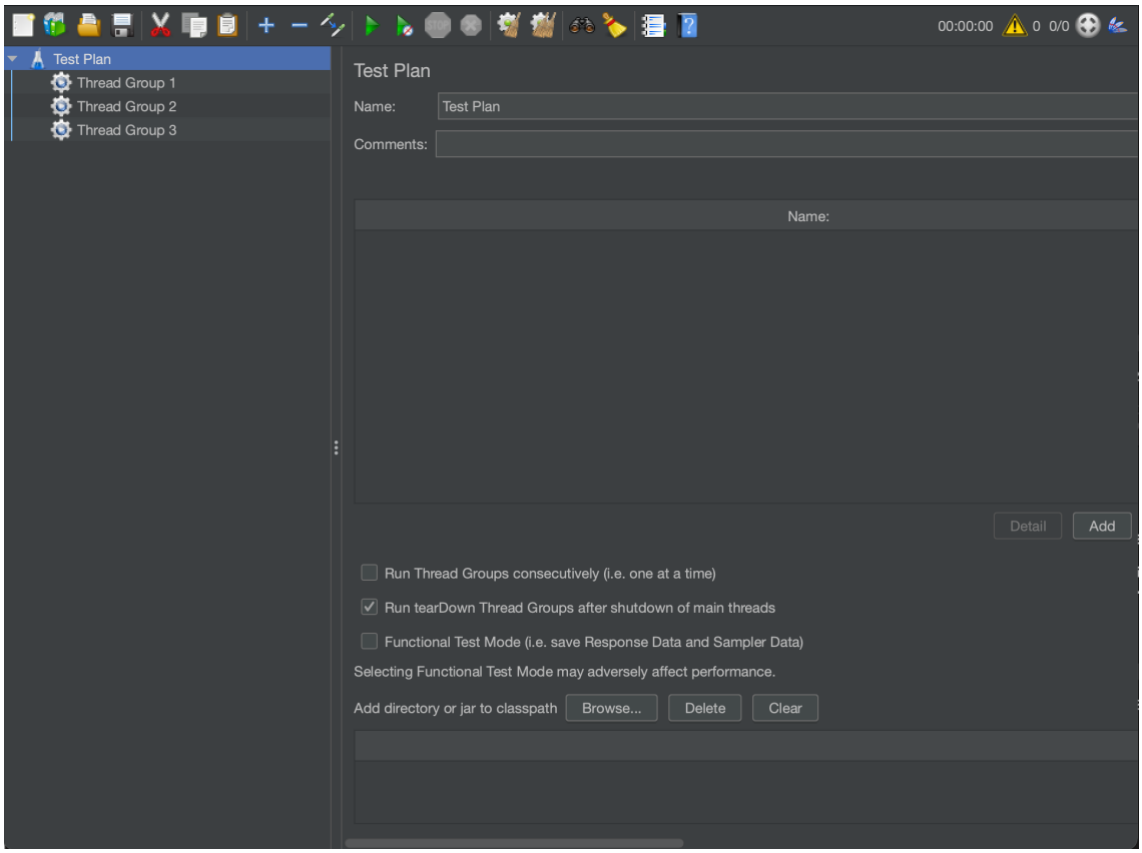


Figure 9. Apache JMeter Graphical User Interface

To make sure that the events used to load the system reflect real-world events, events from customer environments with scrambled information are used for sites, names, and other sensitive information, to ensure it cannot be used to identify the customer or locations of base stations.

## 5.2 Creating Kubernetes Test Environment

Elisa has an internal service called KaaS (Kubernetes as a Service), which is used to create the needed environments in Elisa's servers. It provides a cluster with sensible defaults and minimal need for administrative configurations. The container images for the software are also stored in the cloud. Drone CI will be used for automated deployments, as explained more in detail in Chapter 5.4. In Figure 10 one can see the different components and their relations.

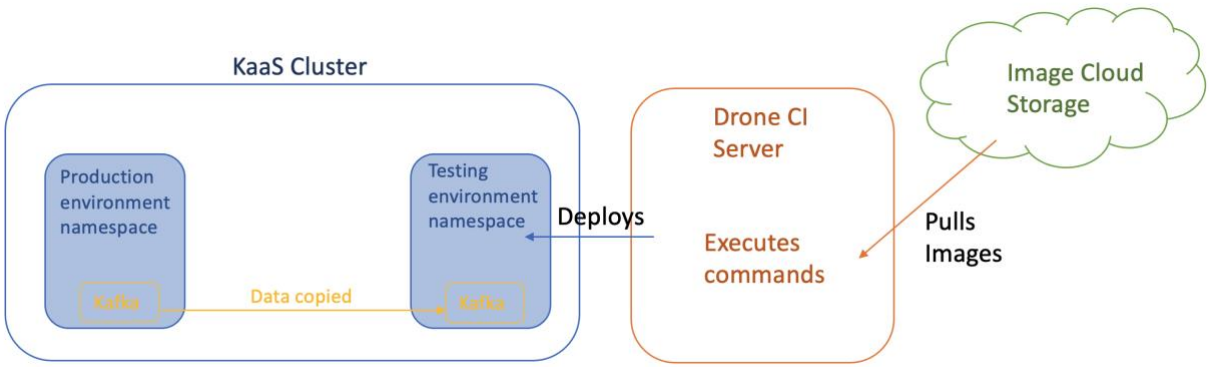


Figure 10. Stability testing environment architecture simplified.

For creating the environment**,** the following command line tools were used.

- Kubectl, which is a tool for running commands against Kubernetes clusters.
- Helm, which is a package manager for Kubernetes.

- Make is a general use build automation tool.

- Lens as a graphical tool for monitoring and troubleshooting the cluster and namespace. Lens is a free open-source tool, and it provides a multitude of handy features.

As the cluster is already provided by Elisa KaaS, it is possible to start by creating a namespace for the environment.

```
kubectl create namespace $(NAMESPACE)
```

Next the resource quotas are defined.

```
kubectl create -f ./infra/dev-resource-quota.yaml -n $(NAMESPACE)
```

See code snippet below for resource specifications used.

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: dev-resource-quota
spec:
    hard:
      cpu: 16
      memory: 24Gi
      requests.storage: 500Gi
```

Code snippet 1. /infra/dev-resource-quota.yaml

After this helm is used for installing Virtual NOC. There are many flags to configure which parts of the system should be on or off, for example the UI needs to be set on and an address can be specified. There are default values defined for most of them. The command “helm install” is used within “make install”. Thus, the command to install Virtual NOC with the values appropriate for this environment is the following.

```
make install VNOC_DEV=true ELASTIC_POD_SEC_CREATE=false
INGRESS_ENABLED=true INGRESS_PREFIX=stability-test LB_ENABLED=true
NAMESPACE=stability-testing TESTERD_ENABLED=true USE_TEST_VALUES=1
NOCA_MOCK_HUAWEI_ENABLED=true VNOC_TESTER_ENABLED=true
```

See below in code snippet 2 what ”make install” actually does.

```
install:
     @$(VIEW_SECRETS) | helm install -n $(NAMESPACE) vnoc ./vnoc/ \
     --dependency-update \
     -f $(VALUES) \
     -f $(ENV_VALUES) \
     --set vnocDev=$(VNOC_DEV) \
     --set vnocTester.enabled=$(VNOC_TESTER_ENABLED) \
     --set
elasticsearch.podSecurityPolicy.create=$(ELASTIC_POD_SEC_CREATE) \
     --set ingress.enabled=$(INGRESS_ENABLED) \
     --set ingress.prefix=$(INGRESS_PREFIX) \
     --set ingress.domain=$(INGRESS_DOMAIN) \
     --set ingress.type=$(INGRESS_TYPE) \
     --set ingress.protocol=$(INGRESS_PROTOCOL) \
     --set lbEnabled=$(LB_ENABLED) \
     --set testerdIsEnabled=$(TESTERD_ENABLED) \
     --set verifierIsEnabled=$(VERIFIER_ENABLED) \
     --set nocaMockHuaweiEnabled=$(NOCA_MOCK_HUAWEI_ENABLED) \
     --set http.authorization=$(HTTP_AUTHORIZATION) \
     --set kafka.kafkaVolumeSize=$(KAFKA_VOLUME_CLAIM) \
     -f -
```

Code snippet 2. “make install” from Makefile.

This will initiate the installation process, and to see the progression Lens can be used (Figure 11). After this the services come up and the system starts running, orange colour signifies that the pod is not ready yet and green means ready.

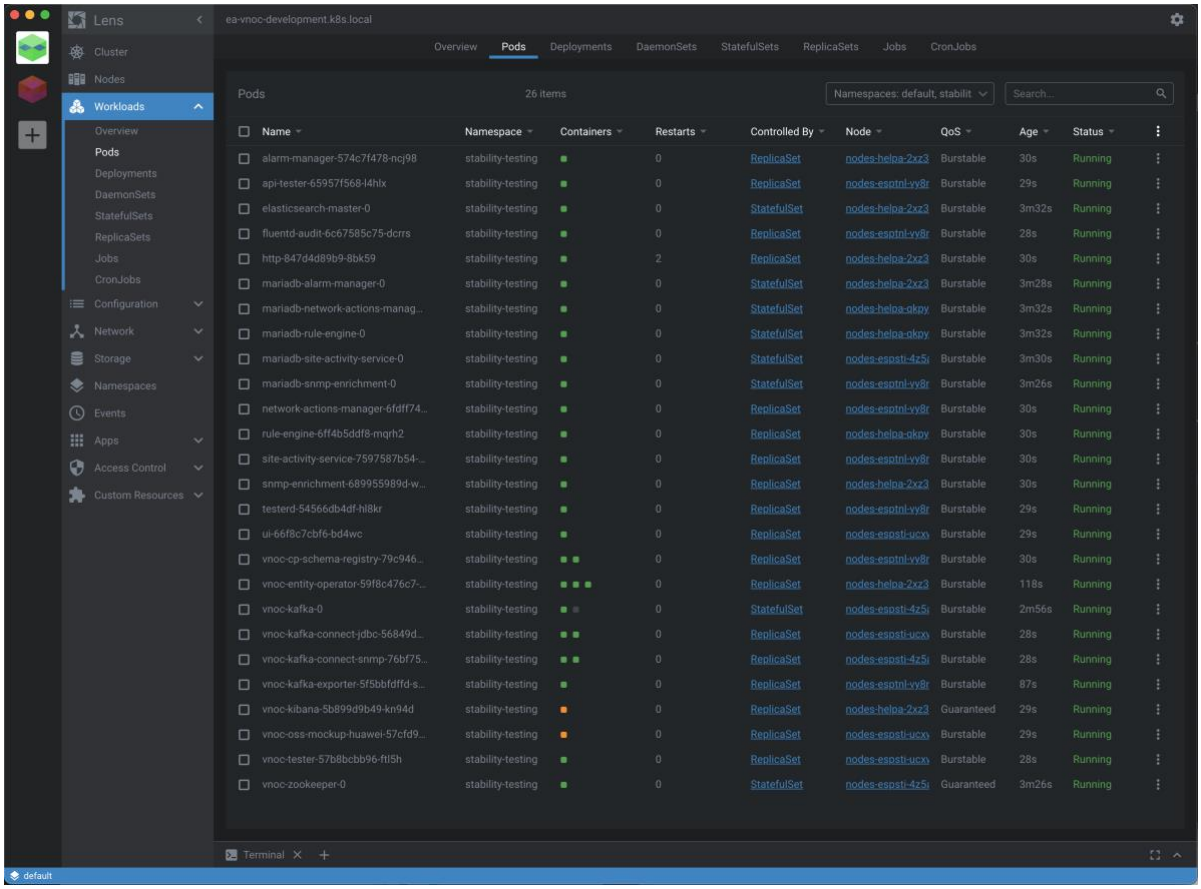


Figure 11. Screenshot from Lens, from where the state of the deployment and its pods can be monitored and edited.

## 5.3 Stability Testing

For stability testing, a Kubernetes environment with a long running vNOC installation which receives constant data streams, preferably real data from production, will be used. The requirements are in addition to the constant data streams, to run in KaaS with Chaos Monkey enabled. [11] The environment can be used to test system recovery from exceptions like service restarts and develop more resilient installations. It can be used to test and develop monitoring solutions like metrics, alerts and dashboards. It needs to have extensive metrics with at least 30 days of retention.

Chaos Monkey is a tool that was created and developed by Netflix to test the resilience of Netflix's IT infrastructure. It works by intentionally disabling computers in their

production network to test how remaining systems respond to the outage. [11] In the present use case, it restarts a random Kubernetes node each day, to verify the services can restart and continue execution normally.

Using Apache Kafka's MirrorMaker, it is possible to get real data from production for the stability testing environment. Mirror Maker copies the data between two Apache Kafka clusters [12], in this case production environment and stability testing environment.

## 5.4 Integration to CI/CD Pipeline

To reduce manual repeated effort and streamline the process, automation of the update process of the environments is needed. This also ensures the testing environments are always up to date and latest versions of the software gets tested quickly. For this Drone CI is used. [13] Using Drone CI, it is also possible to create a nightly or weekly job for the tests to run, in case running them for example after every merge to master is found to be unnecessarily often.

To configure Drone CI pipelines, a file in project root directory, named .drone.yml is used. Below (Snippet 3) there is the beginning of the file. There are some general configurations such as kind, which in this case is pipeline. Next is type, which could be for example docker, Kubernetes or exec. Here, Kubernetes is used, which means each step is run in a pod within a Kubernetes container. An exec pipeline would run directly on the host machine without any isolation. The platform section is used to configure the target operating system and ensures it uses the appropriate runner. [13]

```
---
kind: pipeline
type: kubernetes
name: default

platform:
  os: linux
  arch: amd64
```

Code snippet 3. Beginning of the Drone configuration file .drone.yml.

Below the former, the pipeline steps are defined. They can include many kinds of tasks, e.g. linting, code coverage, vulnerability checks, publishing etc. In the step shown below adding some packages as our base image does not have everything needed, is done first, and then adding the helm repositories to be able to update dependencies, decrypt the secrets file and create the manifest that is used for deployment.

```
steps:
 - name: manifest
     pull: always
     image: ahtaaja.saunalahti.fi/sdc/base/public/helm:3.3
     commands:
       - apk add ansible
       - apk add make
       - apk add curl
       - curl -o ~/ca.crt
https://valkama.saunalahti.fi/api/v2.0/systeminfo/getcert
       - helm repo add bitnami https://charts.bitnami.com/bitnami
       - helm repo add elastic https://helm.elastic.co
       - helm repo add --ca-file ~/ca.crt --username=$$username --
password=$$password valkama
https://valkama.saunalahti.fi/chartrepo/vnoc
       - helm dependency update ./vnoc
       - touch pass
       - echo $$vault > "pass"
       - make local-render-passfile NAMESPACE=$$namespace
VNOC_DEV=true INGRESS_ENABLED=true INGRESS_PREFIX=vnoc-dev
LB_ENABLED=false VNOC_TESTER_ENABLED=true ENV=stabi >
manifest_stabi.yaml
     environment:
       username:
         from_secret: valkama_username
       password:
         from_secret: valkama_password
       vault:
         from_secret: vault_pass
       namespace:
         from_secret: stabi_namespace
     when:
       ref:
         - refs/heads/master
         - refs/tags/*
```

Code snippet 4. Creating a manifest for deployment on Kubernetes in Drone CI.

In the next step, see snippet 5, it does the deployment itself. It uses the manifest created in the previous step and secrets for values and authentication to the destination cluster.

```
  - name: deploy-stabi
    pull: always
```

```
    image: ahtaaja.saunalahti.fi/drone/kubernetes-
drone/public/kubernetes-drone
    settings:
      token:
        from_secret: kube_stabi_token_kaasv3
      ca:
        from_secret: kube_stabi_cert_kaasv3
      server:
        from_secret: kube_api_stabi_server_kaasv3
      namespace:
        from_secret: stabi_namespace
      template: manifest_stabi.yaml
    when:
      ref:
        - "refs/tags/stabienv-*"
```

Code snippet 5. Deployment using the manifest created in snippet 4 step.

The *when:* part defines when the step needs to be triggered. Multiple kinds of conditions can be defined, but they all need to be true to the step to be executed. Here, the reference condition was chosen. It is similar to the branch condition, but also works with tags, which is what is needed. This environment is only re-deployed upon new tag creation.

For some extra convenience, one last step was added, which is notify failure. In case an error happens in the pipeline, the development team gets a message to the chat service used, and the error can be investigated and fixed, or for example restart the build if it was just an anomaly.

```
  - name: notify-failure
    image: plugins/slack
    settings:
      webhook:
        from_secret: mattermost_webhook
      template: >
        [[${DRONE_REPO_NAME} :
${DRONE_BRANCH}](${DRONE_REPO_LINK}/tree/${DRONE_BRANCH})] vnoc-
kubernetes pipeline failed: [drone
#${DRONE_BUILD_NUMBER}]({{build.link}}),
[commit](${DRONE_COMMIT_LINK})
    when:
      status:
        - failure
      ref:
        - refs/heads/master
```

Code snippet 6. The final step from the .drone.yml file.

With these three steps Drone will re-deploy the new version of the software to the testing environment after each merge to the master branch, and notify the developers if the pipeline fails, ensuring the testing environment is kept updated and running.

# 6 Summary and Conclusions

In this thesis the background of Virtual NOC is explained, as to traditional Network Operation Centres and what they do, and what makes Virtual NOC valuable. Technologies used in development e.g. Kubernetes and traditional testing practises are also explained.

The goal of the study was to create performance tests for Virtual NOC and a stability testing environment. The stability testing environment was to be deployed in Kubernetes and be used for stability testing as well as be used to test and develop monitoring solutions like metrics, alerting and dashboards.

The stability testing environment was created successfully according to the acceptance criteria defined in the planning phase. The continuous deployment and integration pipeline was also implemented into Drone CI. This environment can be easily maintained and will be used for further development of the solutions mentioned earlier.

The performance tests were defined, and the implementation was planned, but due to resourcing constrains the implementation was not done by the time of writing the thesis. However, the definitions are valid, and the implementation will be worked on in the coming months.

Other improvement ideas would be more extensive tests, testing of different components, different kinds of loads. For example, comparison between Huawei, Ericsson, and Nokia alarm enrichments and how algorithms of different complexity impact the performance.

## References


1 OECD mobile broadband statistics update https://www.oecd.org/digital/broadband-statistics-update.htm (Accessed 10.4.2021)

2 WHAT IT INFRASTRUCTURE REMOTE MONITORING (NOC) IS https://www.opservices.com/what-it-infrastructure-remote-monitoring-noc-is/ (Accessed 11.4.2021)

3 What is Kubernetes? https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (Accessed: 26.3.2021)

4 Kubernetes Components https://kubernetes.io/docs/concepts/overview/components/ (Accessed: 4.4.2021)

5 Static Testing vs Dynamic Testing: What's the Difference? https://www.guru99.com/static-dynamic-testing.html (Accessed: 23.3.2021)

6 National Software Quality Experiment Resources and Results http://www.reviewtechnik.de/NationalSoftwareQualityExperiment.pdf (Accessed: 23.3.2021)

7 What is Software Testing? Definition, Basics & Types https://www.guru99.com/software-testing-introduction-importance.html (Accessed: 4.4.2021)

8 Unit Testing Tutorial: What is, Types, Tools & Test EXAMPLE https://www.guru99.com/unit-testing-guide.html (Accessed: 4.4.2021)

9 Integration Testing: What is, Types, Top Down & Bottom Up Example https://www.guru99.com/integration-testing.html (Accessed: 4.4.2021)

10 Performance Testing Tutorial: What is, Types, Metrics & Example https://www.guru99.com/performance-testing.html (Accessed: 24.3.2021)

11 Netflix Chaos Monkey Upgraded https://netflixtechblog.com/netflix-chaos-monkey-upgraded-1d679429be5d (Accessed: 30.3.2021)

12 Apache Kafka's MirrorMaker https://docs.confluent.io/4.0.0/multi-dc/mirrormaker.html (Accessed 20.4.2021)

13 Drone documentation - Pipelines overview https://docs.drone.io/pipeline/overview/ (Accessed 25.4)