University of Würzburg
Institute of Computer Science
Research Report Series

Agile Management of Software Based Networks

Steffen Gebert, Christian Schwartz,
Thomas Zinner, Phuoc Tran-Gia

Report No. 493 January 2015

University of Würzburg, Germany
Institute of Computer Science
Chair of Communication Networks
Am Hubland, D-97074 Würzburg, Germany
steffen.gebert@informatik.uni-wuerzburg.de

NOTICE: The technical report is an extended version of the IM 2015 paper "Continuously Deliverying Your Network" containing more use cases as well as a more detailed discussion. Due to page limitations, this is not possible in the original paper. Please cite the peer reviewed paper as follows: *Steffen Gebert and Christian Schwartz and Thomas Zinner and Phuoc Tran-Gia. "Continuously Deliverying Your Network". 2015 IFIP/IEEE International Symposium on Integrated Network Management (IM 2015), Ottawa, Canada, May, 2015.*

NOTICE: The technical report is an extended version of the IM 2015 paper entitled "Continuously Deliverying Your Network". Please cite the peer reviewed IM paper.

Agile Management of Software Based Networks

Steffen Gebert, Christian Schwartz,
Thomas Zinner, Phuoc Tran-Gia
University of Würzburg, Germany
Institute of Computer Science
Chair of Communication Networks
Am Hubland, D-97074 Würzburg, Germany
steffen.gebert@informatik.uni-wuerzburg.de

Abstract

The softwarization and cloudification of networks through mechanisms like Software Defined Networking and Network Functions Virtualisation promise a new degree of flexibility and agility. By moving logic from device firmware into application software and by applying mechanisms from software development, innovations can be introduced with lower effort. Concrete ways how to operate and orchestrate such systems, however, are not defined. The process of making changes to a controller software or a virtualized network function in a production network without the risk of network disruption is not covered by literature, yet. Complexity of systems bring the risk of unexpected side-effects and has so long been the show-stopper for administrators being relaxed while applying changes to networking devices. This paper suggests the adaption of the successful concept of continuous delivery into the software defined networking world. Mechanisms like Test-Driven Development and automatic acceptance tests demonstrate that the software engineering community already found ways to ensure that changes do not break a software's functionality. Applied to network engineering, the adaption of continuous delivery can be seen as an enabler for risk-free and frequent changes in production infrastructure through push button deployments.

Contents

1 Introduction **3**

2 Background **4**
2.1 SDN and NFV 4
2.2 Continuous Delivery 5

3 Continuous Delivery of Network Functions **7**
3.1 SDN Controller Pipeline 8
3.2 Traffic Shaping Network Function 11

4 Discussion **13**
4.1 Releasing More Frequently 13
4.2 Overhead of Testing Every Change 13
4.3 Automated Testing of Networks 14
4.4 Adaption of Behavior-Driven Development to Networks 14
4.5 Metrics 14

5 Conclusion **15**

1 Introduction

In the last years, the introduction of Software Defined Network (SDN) has ushered networks in an agile new future, allowing flexible configuration for researchers and businesses alike. However, regardless of these new accomplishments even after decades the Command Line Interface (CLI) is still the best friend of network engineers when configuring switches, routers or other network devices. Furthermore, the network is configured decentralised directly on the devices, irregardless if this happens via CLI, Graphical User Interface (GUI) or a Web interface. Thus, there is no simple way to test the complete network configuration before applying it device by device. The network engineer has to make sure to always enter the correct commands, make no typing errors and hope for no unforeseen side-effects while configuring the network. This cumbersome process causes severe problems for the management of today's IT infrastructure, as the fear of breaking the production network results in security updates, e.g. for the infamous Heartbleed bug, being applied late or never [1, 2].

In most current networks, applying updates or configuring devices is scheduled in maintenance windows, where the user is informed beforehand that failures may occur. This approach tries to increase the Mean Time Between Failure (MTBF), as downtimes are avoided by avoiding changes until finally necessary. At the same time, the risk of failure when changes are applied is increased, as multiple changes are applied at the same time. Furthermore, also identifying the root causes of potential problems becomes harder, when more than one change is introduced to the network at the same time.

Software development projects faced similar problems before agile project management methods like *Scrum* were introduced to reduce the risks of late integration. Here, methods from *DevOps* [3], allow for fast feedback cycles and frequent releases in order to avoid misconception, extensive manual testing, as well as failures that are hard to identify due to the fact that many changes are applied to the system simultaneously. Instead, modern web companies like *GitHub* [4] or *Amazon.com* [5], release changes into production more often – in number of tens or hundreds per day. These practices aim to increase reliability by reducing the Mean Time To Repair (MTTR) instead of increasing the MTBF. Through automated test execution and deployments, the quality assurance efforts per change can not only be reduced, but also the time to release a fix into production, which might also be to revert a change, is effortless.

Some current network configuration management tools, e.g. *Solarwinds Network Configuration Manager* [6] or *InfoSim StableNet* [7], allow for the automatized deployment of changes to network devices. However, the proprietary nature of the configuration interfaces of traditional network interfaces results in a high complexity and price for Network Configuration Management (NCM) tools. Thus, many modern networks are still not using such tools and are still maintained manually by a network engineer.

Recent developments in the area of SDN are about to disrupt this market and offer new opportunities for change. Besides the benefits of a better network performance, a simplified management and configuration of the network infrastructure is promised. Most research work on the management of software defined networks however focuses on improving network performance or reliability by defining controllers architectures and

roles of entities [8] or generally the more sophisticated management of network flows [9]. The actual life cycle of the introduced SDN entities, including provisioning and maintenance, is not covered by existing literature. This work suggests to apply Continuous Delivery (CD) to SDN and the related concept of Network Functions Virtualisation (NFV), in order to not only benefit from the provided agility, but also to support the effortless and risk-free deployment of new networking software.

This paper is structured as follows: Section 2 introduces the technical background, which consists of SDN and NFV as concepts from the networking world, as well as CD as software engineering mechanism. Section 3 describes the contribution of this work, which is to apply CD to SDN-based networks. Details of potential implementations, as well as open issues, are discussed in Section 4. Section 5 concludes this work.

2 Background

In this section, techniques from network and software engineering are introduced. This builds the foundation for understanding the contribution of this paper – applying successful techniques from software engineering to the management of modern networks.

2.1 SDN and NFV

The Software Defined Network (SDN) concept suggests the externalization, centralization and softwarization of the network control plane into an SDN controller software – a shift away from traditional, hardware-centric networking towards open interfaces and software-driven network control. An optimization of traffic flows inside the network is possible by having a global view over the network inside the logically centralized control plane [10]. By the controller running as software application on a standard server, faster innovation cycles for the network control plane is possible than it is with the currently integrated devices of vendors like *Cisco*, where firmware updates have to be installed in order to introduce new mechanisms or protocols. The pace of innovation can also be seen by the number of available OpenFlow controller implementations – and the number of projects that are stopped being developed any further.

While administrators outside of hyper-giants like *Facebook* and *Google* will very unlikely change the core functionality of an SDN controller, the plugin architectures of modern SDN controllers allow extension of this centralized intelligence. The fields of extension include monitoring, routing, security, application awareness and quality of experience optimizations. An ecosystem for such controller plugins is expected by the authors, similar to the "app stores" of computers and smart phone vendors.

Network Functions Virtualisation (NFV, [11]) in contrast aims at replacing Network Functions (NFs) provided by monolithic hardware middleboxes with software implementations running virtualized on standard servers, the Virtual Applications (VAs). An overdimensioning of resources as it is usual with hardware can be avoided by applying elasticity mechanisms of nowadays cloud application stacks, including cluster and replication functionality.

Therefore, NFV promises slim software instances that provide a particular functionality, like applying deep packet inspection, firewalling or functionality of LTE mobile networks [12]. Again, shorter innovation cycles and an increased flexibility are a driver for research, vendors and operators to investigate the use of virtualised network functions. SDN is the preferred mechanism to pipe all or only specific parts of the traffic, like all traffic on TCP port 80, through a specified set of network functions. The VNF Forwarding Graph [13] specifies, which network functions should be passed, and e.g. if the traffic should be mirrored to a monitoring function, or passed through an intrusion detection function in order to block selected traffic.

All the promises of increased flexibility by frequently modifying the network software and configuration of SDN controllers and the NFV infrastructure comes with the big risk of breaking the network. Besides bugs and breaking changes in software implementations, the risk of manual configuration errors are reasons, why traditional networks are progressing only very slowly. The open issue is now, how virtualization of network elements and functions helps to allow frequent changes without unexpected outages.

Frequent releases of software and therefore very short innovation cycles are, however, also a goal of application software developers. One concept that allows frequent deployment of new software releases, while maintaining higher software quality in order to reduce the risk of broken deployments, is Continuous Delivery.

2.2 Continuous Delivery

In order to mitigate the risk of broken software deployments while keeping a high pace of software releases, Continuous Delivery (CD, [14]) introduces the concept of the Deployment Pipeline.

Every change to the software developed using the CD paradigm has to pass all stages of this pipeline, in order to be released into production. An important principle is that in case of failure, a fast feedback to the team making the changes, the *delivery team*, is given and illustrated in Figure 1. Every version of the software that passes until the end of the pipeline is considered as stable. The stages of such a deployment pipeline that are executed on a centralized server running a software like *Jenkins* [15] or *Go CD* [16] are described as follows:

Version control: Every change to the software is checked into a Version Control System (VCS), like *Git* or *Subversion*. The use of a VCS allows the team to prevent overwriting changes of source code files, when different developers modify the same file in a short time. The traceability of changes by storing all historic versions of a software's source code gives the possibility to revert back to a state of the software that is known to work in case of regressions. After checking a change of the software into version control, the deployment pipeline is instantiated and the state of the software code passed through the following stages.

Build & unit tests: The centralized build server picks up the source code and creates an executable build artifact by compiling the source code. By creating the build on a centralized server, it can be made sure that not only a single developer can compile

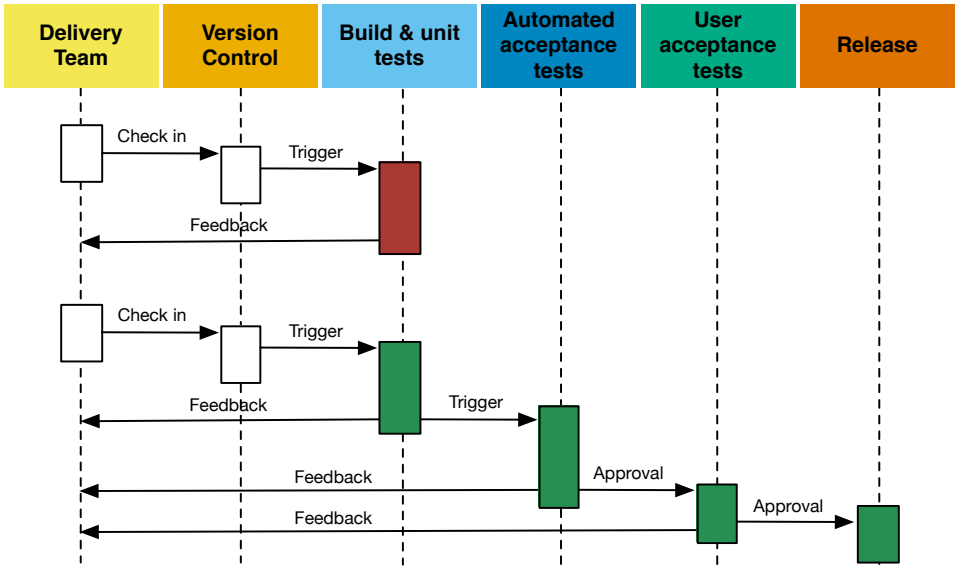

Figure 1: The deployment pipeline for software projects as suggested by [14]. Red: Test execution resulted in failure; green: tests succeeded.

and release software, but the whole team. Building only on PCs of developers introduces the risk of errors on other developer's machines because of different configuration or compiler and library versions. After successfully creating the build and following the technique of Test-Driven Development (TDD), unit tests are executed against the compiled software. In case of any error during the build process or while running the unit tests, the pipeline is stopped and the developer informed about the problem. A fast feedback regarding any failure is important for the success of software development. If the stage is successfully passed, the next stage is triggered.

Automated acceptance tests: The particular steps in this stage are dependent on the actual implementation of the deployment pipeline. The result of this stage however is the knowledge that the created build artifact meets the specified acceptance criteria – or not. These acceptance criteria range from functional and integration tests over to capacity tests, and longer-lasting source code analysis. Functional tests ensure that the functionality of a software actually meets its specification. Integration tests ensure that a part of the software works expected after integrating with other components. In order to execute all of these tests, a running version of the software is required, which is therefore deployed into a production-like infrastructure. This means that the environment, where the tests are executed, matches the production environment in terms of version and configuration of operating system, libraries and installed software. If all tests can be executed successfully, the pipeline execution is continued and otherwise the delivery team is notified to adjust.

User acceptance tests: In order to verify the implementation of a new feature, the software is deployed into a production-like environment, where all team members can access the current state. The Quality Assurance (QA) sign-off, when testers manually verify the functionality of the software, is the first human intervention after making the commit to the VCS. An extensive automated test suite that is executed in the previous steps and verifies the basic feature set of the software gives now the QA team time to focus on new features and exploratory testing. After this manual verification, the build artifact is ready to be released.

Release: The release of the software means that it is installed on the production servers or, in case of on-premise software, that is made available for customers to download. This stage can be either triggered manually, or after a button click in the software supporting the CD process.

The result of this process is always a state of the software that is known to work, either the current version or an older one. Through automated tests, the manual QA efforts are reduced and the duration that a change takes to pass through the deployment pipeline in order to be known as releasable is reduced to minutes or hours.

Besides to software development, the continuous delivery paradigm has been successfully applied to other areas. Modern configuration management software, like *Chef* [17] or *Puppet* [18], that follows the *infrastructure as code* paradigm allows to define the setup of a particular entity, mostly a cloud server including its application stack, as source code. Any change to the configuration inside the source code repository is only brought into production when the new configuration successfully passes the continuous delivery process.

3 Continuous Delivery of Network Functions

In the following, we describe the suggested adaption of Continuous Delivery to network operations. This does not mean that network administrators will now become or be replaced by software developers. However, network engineers should learn from experiences made in the software development world.

Similar to the configuration changes on servers, any change to the network software brings a risk of introducing regressions. This includes not only changes of configuration settings or the deployment of a new network software version. Also during the deployment of a totally new component, e.g. a new virtualised network function that is inserted into the VNF forwarding graph, manual misconfiguration and incompatibilities entail risk of network outages.

The contribution of this paper is therefore to apply the concept of *continuous delivery* [14] to the software defined networking world. The main building blocks are (a) process automation, (b) automated tests, (c) availability of realistic test environments and (d) infrastructure as code.

Through *process automation*, network engineers are supported in their daily work – instead of being jammed by yet another tool to use and process to follow. Goal is to

reduce the overhead of work that is required to make a configuration change. Instead of manually logging into all networking devices, the deployment will happen automatically on all affected entities. An *automated test suite* further supports the goal of reducing the overhead, in this case the additional work of manual testing. Furthermore, automated tests bring (high) confidence that any change that passed the automated tests will not be disruptive when deployed into production. The availability of *realistic test environments* that can be set up automatically is also a prerequisite of executing automated tests. Compared to traditional, hardware-centric networks, where devices have to be connected manually, the softwarization of networks now allows to automatically set up an environment matching the production environment through instantiation of virtual machines running the same software in the same versions. This allows the engineer or QA staff also to manually test changes prior to rollout, without complicated setup of infrastructure. Finally, *infrastructure as code* ensures that the configuration of servers running a network function is not only documented in code, but also that this code can be applied to another system in order to reproduce the configuration of the production environment. This technique allows to instantiate new development and testing environments without manual intervention.

In the following, the transfer of the continuous delivery concept to network functions is described. The stages of the resulting deployment pipeline are shown in Figure 2 and in the following illustrated using two typical functions of an SDN/NFV-based network, which are (a) an SDN controller and (b) a Virtualised Network Function (VNF) applying traffic shaping.

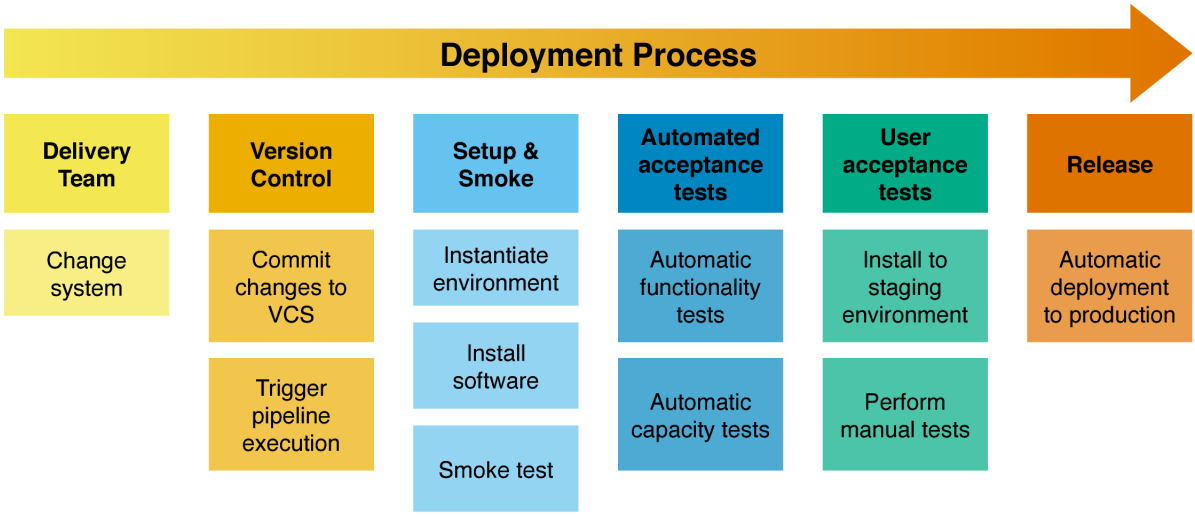

Figure 2: Suggested deployment pipeline for network functions.

3.1 SDN Controller Pipeline

The controller plays a critical role in an SDN network. Any fault in its execution would make the network "headless". Situations like when an administrator supplies an invalid

configuration file resulting in the controller service to not to restart successfully for a time frame of several seconds or minutes have to be urgently avoided. Therefore, the error-prone manual configuration on production devices has to be succeeded by an automatic process that is backed by automatic tests for the deployment of the central network function of an SDN controller.

While the CD concept can be also applied for the software engineering task of developing the SDN controller software, this work focuses on the more realistic case when an administrator uses an existing controller software and configures it according to own needs. If the controller software itself has to be compiled, a dedicated deployment pipeline following the original version of CD in terms of software engineering projects should be established. The case that is presented in the following uses the compiled artifact of an SDN controller software that could be also a purchased software that is not available as source code, but only as binary artifact.

The following stages are suggested for the deployment pipeline applied to an SDN controller:

Version control: The version control repository for the SDN controller pipeline contains the configuration of the servers running the controller and how the controller has to be installed. This includes an exact specification of the controller version that has to be installed, as well as the configuration files of the controller software.

Setup & Smoke: As this pipeline does not involve any compilation of software, the focus is more on compiling the infrastructure components together. Therefore, a virtual machine that matches the configuration of the production servers is provisioned. The controller software is downloaded from the specified source in the specified version and installed into the virtual machine. This ensures that the automatic deployment process works and the controller software and all dependencies can be downloaded and installed. Finally, the setup is completed by supplying the configuration files as specified in the version control repository. This automatic process is ideally implemented using a *Desired State Configuration Management System* like *Chef* or *Puppet*.

Following the principle of fast feedback, this stage only includes tests that can be executed in a very short time frame, while catching many of the errors that are likely to happen. In [14] it is recommended to not exceed the 10 min mark. What should fit in this time frame are *smoke tests*, which only consist of testing, whether the controller software using the supplied configuration is able to start up or not.

Automated acceptance tests: The tests running in this phase ensure that the specified acceptance criteria for the software being deployed are met. A failure in any of the tests means that at least one of the criteria that are defined to be essential for the operational software is not fulfilled and thus the pipeline has to be stopped. Besides functional tests, which means that the software does, what it should, the acceptance tests also include non-functional tests, like that it meets certain performance or capacity requirements.

The functional test suite of an SDN controller should include at least the following checks:

- *Accepts incoming southbound connections*: While it sounds trivial, this check ensures a very basic functionality, namely that the controller is listening to the correct interface and port for incoming connections. This allows to quickly identify such a basic error, instead of uncertainty, when the later tests fail.
- *Allows a switch to forward traffic*: Independent of an active or reactive setup of rules inside the switches' forwarding tables, the result should be that two stations connected to a switch can exchange network traffic.
- *Accepts incoming northbound connections*: In the author's opinion, the whole benefit of SDN can only be exploited, if an integration into and coordination with the remaining IT infrastructure is happening [19]. Therefore, the functionality of the northbound API, nowadays usually implemented as a RESTful API, is essential as the functionality of the southbound API is. This first northbound test ensures that a client is able to connect. Besides the correct configuration of the listening interface and port, this also verifies that the authentication is working correctly.
- *Required feature set of northbound API is working*: Certainly, not all features of an SDN controller's northbound API will be used in a particular setup. However, the subset of functionality that is used should be verified to work correctly. If the production setup e.g. consists of a cloud controller that interacts with the SDN controller, the correct functionality of that part of the REST API should be tested automatically. This ensures that especially updates of the controller software that might introduce changes or bugs are not deployed without the confidence that the used feature set is still functioning.

Other than functional tests, non-functional tests evaluate characteristics of a setup that cannot be directly described in a fashion that a certain response is expected for a certain request. One of these non-functional requirements is the ability to handle a load, in terms of connected switches as well as a rate of requests. Therefore, the controller software is set under a certain emulated load that matches at least the expected peak load of control traffic of the production network. Solutions to emulate the controller load of a larger OpenFlow network include [20] and [21].

User acceptance tests: While extensive manual testing should be avoided in order to allow a change to quickly propagate through the pipeline, manual tests of the administrator or a QA engineer in the staging environment allows detailed manual tests. Therefore, the tester can connect an own virtual switch, or even a hardware switch, to the controller software running in the staging environment. For convenience, additional virtual machines can be deployed and preconfigured in so that the tester can focus on the actual testing, instead of struggling with setting up the test environment. The manual tests done here should not test absolutely critical functionality so that they have to be repeated during every run of the pipeline.

Instead, such tests should then be automatized and executed in the previous stage. This allows the tester to focus on new functionality in detail.
After manual testing, the tester gives his either positive or negative grade and thus either signs-off to proceed to the next stage, or fails the pipeline and returns back to the delivery team to adjust.

Production Deployment: The final stage of the deployment pipeline triggers the production deployment. This step can be either done automatically after every successful execution of the pipeline, bundled into e.g. a single deployment per day, or manually triggered. In case of manual releases, queuing up a large number of changes would contradict with the idea of continuous delivery – the uncertainty while trying to identify a failure of a large deployment is exactly what should be prevented. Instead, small, incremental change is desirable.

3.2 Traffic Shaping Network Function

NFV aims at replacing hardware middleboxes through Virtualised Network Functions (VNFs). The deployment pipeline for a VNF that is inserted into the flow of traffic will be described in the following using a traffic shaping (TS) function. Besides giving more capacity to other network flows, traffic shaping or rate limiting is a common use case for mobile networks. After reaching the "flat-rate" limit of e.g. 1 GB per month, the mobile operator is shaping the subscriber's traffic to a lower rate like 64 kbps. Therefore, received traffic is queued inside the entity running the function and then forwarded with a reduced rate.

The availability of such functions is crucial for correct operation of the network. If one function of the forwarding graph fails to forward traffic, the whole traffic flow is interrupted. Therefore, ensuring failure free functionality after deployments is seen as an important prerequisite for successful migration towards NFV-based networks.

The suggested deployment pipeline for a TS-VNF and similar functions is as follows:

Version control: Similar to the pipeline of the controller, the version control repository for the TS pipeline contains the configuration of the servers running the function. If the function is developed as source code, another deployment pipeline following the original CD for software projects [14] would be set up.

Setup & Smoke: Again, the main focus of this stage is to ensure the successful deployment by installing the specified software into a production-like environment. A smoke test that verifies that the application implementing the VNF starts without errors.

Automated acceptance tests: The acceptance criteria for a VNF include criteria that are common for a certain type of network function and some that are specific for a particular function. Tests that are common e.g. for all middlebox VNFs ensure that incoming traffic is also sent back as specified, in order to reach the final destination or the next network function. Automated acceptance criteria that are specific to a traffic shaping function should include:

- *Sent data matches received data:* Using a TCP data transfer it is checked that the Layer 7 payload sent out equals (bitwise) the payload that is sent into the function under test. The aim of TS is not to falsify the data in any way. Implementation errors which result in data corruption can be detected this way.
- *Traffic shaping is successfully applied:* In order to verify the core functionality, the throughput or duration of a data transfer is measured. By defining a certain threshold, deviations of some percentage from the defined bandwidth limit are not causing the test to fail, but ensure that TS is applied.
- *Only traffic that should be shaped is shaped:* Assuming that the shaping functionality allows to handle flows differently, this test ensures that bandwidth limits are only applied to the specified flows and not to others that should not be shaped. Therefore, a bandwidth limit for a certain flow criteria is defined. The duration of a data transfer through the network function that *not* matches the shaping rule is measured and the average throughput computed. If the measured throughput exceeds the shaping limit by a factor specified in the test, e.g. twice, the test succeeds and it can be assumed that the shaping is only applied to specified flows.

While the elasticity of network functions and the automatic up and down scaling should ensure that additional capacity is provided in case of increased resource requirements, automated capacity and performance tests are still of big importance. Heavily increased requirements of a certain functionality can cause various kind of trouble, including additional costs. Therefore, either static triggers with fixed limits, or a limit of divergence from previous runs ensures that a by far more resource-intense new implementation or configuration is not released into production.

User acceptance tests: Again, the network engineer or QA staff has the chance to manually verify functionality like already described in the pipeline described for the SDN controller pipeline in Section 3.1.

Production Deployment: As already applied in previous testing environments, the automated deployment is now happening into the production environment. Depending on the configuration of the pipeline, this stage is executed automatically after the QA sign-off in the previous stage or on the push of a button in the software supporting the continuous delivery process.

The previous two sections described the application of the concept of Continuous Delivery to network functions. The deployment pipeline of an SDN controller and of a TS-VNF as typical functions of SDN/NFV-enabled networks were used to illustrate how the agile deployment of such functionality into the production network should happen.

4 Discussion

The idea of this work, namely to apply the software engineering concept of Continuous Delivery to networks has been described previously. In this section, we want to discuss the aspects that are in the opinion of the authors worth noting to understand the reasons for introducing such a process. Furthermore, open issues that should be tackled by the software development or network engineering community in order to further support the adoption of this concept are described.

4.1 Releasing More Frequently

For agile software development, the "highest priority is to satisfy the customer through early and continuous delivery of valuable software" [22]. That quality and productivity is increased through introduction of agile methods in software engineering is shown by a survey in [23]. Also that more releases do not mean more bugs, instead that fixes are released faster, is shown in a study of comparing *Mozilla Firefox* with long and short release cycles [24]. For *Amazon.com* it is reported that the number of outages triggered by software deployments was reduced by 75% within 5 years. The safety net provided by a deployment process backed by automated tests instead reduces stress for humans. In case of failures, the smaller amount of changes deployed simultaneously makes the identification of root causes easier and allows automated rollback.

4.2 Overhead of Testing Every Change

The execution of numerous automatic tests often results in the feeling that CD would introduce a large overhead into software development processes. In order to be able to identify the particular commit in the source code repository that introduces a regression, every single change executes numerous automated tests. The suggested pipeline execution includes the instantiation of multiple virtual machines and running functional tests with potentially a duration of several minutes.

To understand that the benefits of this automatic process are worth implementing, one has to realize that the compute power that is spent for automated tests is just the replacement for manual tests by humans. These manual tests, in the field of network software certainly similar to software development, otherwise bind a large capacity of human resources. As generally compute power is cheaper than engineer work, the resources spent for automated tests can be considered lower than for manual quality assurance.

Compared to manually applying changes, the delay until a change passed through the deployment pipeline, is notably larger. However, manual configuration changes are considered bad practice, as they cannot be tracked (who changed what and when) or easily reverted and do not scale to a large number of devices. Therefore, the time for an automated deployment process has to be preferred over manual fire-and-forget changes in production.

4.3 Automated Testing of Networks

The authors of [25] argue for applying the software engineering concept TDD to the management of SDNs. In order to prevent a faulty network configuration to be deployed, a formal language called Data Path Requirement Language (DPRL) is introduced. Using the specifications made in DPRL, the compliance of an SDN controller against the specified rules can be verified. The authors provide a prototype implementation that builds upon *Mininet* and *Open vSwitch*.

The work in [25] can be seen as an important step into the right direction. TDD is an important building block for continuous delivery of networks. However, the suggested prototype implementation would still require the network engineer to write Python code.

4.4 Adaption of Behavior-Driven Development to Networks

In the agile software development world, where software requirements are specified by others than developers, the technique of *Behavior-Driven Development* (BDD, [26]) gains more and more attention. Implementations of the natural language style domain specific language called *Gherkin* [27] are available for nearly all platforms. Giving such a language with an implementation for the used SDN technique, e.g. OpenFlow, we are confident that network administrators can far more easily adopt the concept of TDD and continuous delivery. An example of a Gherkin-based specification of the behavior of an SDN controller that is automatically verified, is shown in Listing 1.

```
Feature: Reactive mode
 Scenario: Flow to unknown destination
  Given the switch having a flow table with
  no entries connects to the controller
  And computer A is connected to the switch
  And computer B is connected to the switch
  When computer A sends data to computer B
  Then the data should be received by computer B
```

Listing 1: BDD specification of a reactive controller behavior-

Certain tokens of that language (like `Flow table with (.*) entries`) are interpreted by the test driver provided by the SDN controller, another specialist for network protocols or e.g an open-source project. Therefore, a user of the language, the network engineer, does not need to know the actual implementation of the control plane protocol. For example, in case of OpenFlow, the user does not need to know about `PACKET_IN` messages that are sent to the controller. Instead, the resulting behavior is described using the provided language constructs in a *Given-When-Then* style.

4.5 Metrics

An essential part of CD and the related DevOps practices is to monitor, how the production infrastructure behaves. The collection of metrics in a DevOps context means more than just monitoring of bandwidth usage and QoS [28]. Instead, a data-driven culture

relies on aggregating data from numerous sources together, in order to allow engineers, as well as business units to take decisions based on measured truth and not on assumptions or feelings. While the change of network parameters does not necessarily result in a change of measured QoS metrics, it can affect performance of applications running in the network. The collection and aggregation of metrics can go so far that also the number of transactions in an online shop is monitored. In case of a large decrease of this metric, the metrics collected from the network side can be correlated as well as matched with times of deployments. If a certain behavior is seen after a point in time at which a change to the network or server infrastructure happened, this change is likely to be the cause for this changed behavior. Again, it is important that all changes are tracked in a version control repository to revert to previous states, as well as that data about the exact time when deployments happen are stored.

Together with an extensive collection of metrics and the use of a concept similar to *Feature Flags* [29], it would be possible to treat parts of the traffic different than others. Feature Flags allow to change a particular behavior, like the availability of a feature, for a group of users. Transferred to networks, this could mean that a mechanism that should be evaluated can be tested under production conditions for only a number of users, servers, or flows. Metrics then allow to compare one implementation against the other. The flexibility of steering network traffic provided by SDN can be seen as an enabler for techniques such as A/B-testing, where two different implementations are compared regarding specified metrics.

5 Conclusion

Thanks to softwarization of networks through the concepts SDN and NFV, a more agile management of network infrastructure seems possible. Any change however introduces the risk of failures and this is why traditional networks are hardly changed. The risk of frequently introducing changes into production networks through the new mechanisms SDN and NFV has to be mitigated by automated processes, which bring the certainty that configuration changes or updates will not harm the network. This paper first described the established concept of Continuous Delivery, an important technique for agile and high-quality software development. The adaption of the continuous delivery process to network software in order to bring agile methods and risk-free deployments into the management of networks is the main contribution. The infrastructure as code paradigm that relies on tracking every change in a version control repository followed by automated deployments of infrastructure through configuration management software ensures availability and identicalness of testing, staging and production environments. Automated acceptance tests automatically verify every change prior to deployment into production infrastructure, which then happens automatically or through the push of a button. This process was illustrated by describing the deployment pipeline of two typical components of an SDN/NFV-enabled network, an SDN controller and a middlebox VNF.

Acknowledgment

This work has been performed in the framework of the CELTIC EUREKA project SASER-SIEGFRIED (Project ID CPP2011/2-5), and it is partly funded by the BMBF (Project ID 16BP12308). The authors alone are responsible for the content of the paper.

References

[1] TrustedSec, "CHS Hacked via Heartbleed Vulnerability." https://www.trustedsec.com/august-2014/chs-hacked-heartbleed-exclusive-trustedsec/, August 2014.

[2] G. Cluley, "Heartbleed blamed for hack that put 4.5 million patients at risk." http://grahamcluley.com/2014/08/heartbleed-chs-hack/, August 2014.

[3] E. Mueller, "The agile admin: What is devops?." http://theagileadmin.com/what-is-devops/.

[4] J. Douglas, "Deploying at GitHub." https://github.com/blog/1241-deploying-at-github.

[5] J. Jenkins, "Velocity culture (the unmet challenge in ops)," in *O'Reilly Velocity Conference*, Jun. 2011.

[6] "Solarwinds Network Configuration Manager." http://www.solarwinds.com/network-configuration-manager.aspx.

[7] "Infosim.net: StableNet Enterprise." https://www.infosim.net/products/stablenet-enterprise.html.

[8] R. Ahmed and R. Boutaba, "Design considerations for managing wide area software defined networks," *Communications Magazine, IEEE*, vol. 52, pp. 116–123, July 2014.

[9] H. Kim and N. Feamster, "Improving network management with software defined networking," *Communications Magazine, IEEE*, vol. 51, pp. 114–119, February 2013.

[10] M. Jarschel, F. Wamser, T. Höhn, T. Zinner, and P. Tran-Gia, "SDN-based Application-Aware Networking on the Example of YouTube Video Streaming," in *European Workshop on Software Defined Networks (EWSDN)*, (Berlin, Germany), Oct. 2013.

[11] ETSI, "Network Functions Virtualisation (NFV); Architectural Framework," Oct. 2010.

[12] S. Gebert, D. Hock, T. Zinner, P. Tran-Gia, M. Hoffmann, M. Jarschel, E.-D. Schmidt, R.-P. Braun, C. Banse, and A. Koepsel, "Demonstrating the Optimal Placement of Virtualized Cellular Network Functions in Case of Large Crowd Events," in *ACM SIGCOMM, Chicago, USA*, Aug. 2014.

[13] ETSI, "Network Functions Virtualisation (NFV); Terminology for Main Concepts in NFV," Oct. 2010.

[14] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1st ed., 2010.

[15] "Jenkins - an extendable open source continuous integration server." http://jenkins-ci.org.

[16] "go continuous delivery." http://www.go.cd.

[17] "Chef." http://www.getchef.com/chef/.

[18] "Puppet enterprise." http://puppetlabs.com/puppet/puppet-enterprise.

[19] M. Jarschel, T. Zinner, T. Hoßfeld, P. Tran-Gia, and W. Kellerer, "Interfaces, Attributes, and Use Cases: A Compass for SDN," *IEEE Communications Magazine*, vol. 52, June 2014.

[20] M. Jarschel, C. Metter, T. Zinner, S. Gebert, and P. Tran-Gia, "OFCProbe: A Platform-Independent Tool for OpenFlow Controller Analysis," in *IEEE International Conference on Communications and Electronics (IEEE ICCE 2014)*, (Da Nang, Vietnam), Aug. 2014.

[21] P. Wette, M. Draxler, and A. Schwabe, "Maxinet: distributed emulation of software-defined networks," in *Networking Conference, 2014 IFIP*, pp. 1–9, IEEE, 2014.

[22] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Manifesto for agile software development." `http://www.agilemanifesto.org/`, 2001.

[23] A. Ahmed, S. Ahmad, N. Ehsan, E. Mirza, and S. Sarwar, "Agile software development: Impact on productivity and quality," in *IEEE International Conference on Management of Innovation and Technology (ICMIT)*, pp. 287–291, June 2010.

[24] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams, "Do Faster Releases Improve Software Quality? An Empirical Case Study of Mozilla Firefox," in *IEEE Working Conference on Mining Software Repositories (MSR)*, pp. 179–188, June 2012.

[25] D. Lebrun, S. Vissicchio, and O. Bonaventure, "Towards Test-Driven Software Defined Networking," in *Network Operations and Management Symposium (NOMS)*, pp. 1–9, May 2014.

[26] D. North, "Behavior modification - the evolution of behavior-driven development," *Better Software Magazine*, June 2006.

[27] "Cucumber wiki: Gherkin." `https://github.com/cucumber/cucumber/wiki/Gherkin`.

[28] J. Roche, "Adopting devops practices in quality assurance," *Communications of the ACM*, vol. 11, Nov. 2013.

[29] R. Harmes, "Flipping out." `http://code.flickr.net/2009/12/02/flipping-out/`, Dec 2009.