Continuously Delivering Your Network

Steffen Gebert, Christian Schwartz, Thomas Zinner, Phuoc Tran-Gia
University of Würzburg, Institute of Computer Science
{steffen.gebert,christian.schwartz,zinner,trangia}@informatik.uni-wuerzburg.de

***Abstract*—Softwarization and cloudification of networks through software defined networking and network functions virtualisation promise a new degree of flexibility and agility. By moving logic from device firmware into software applications and applying software development mechanisms, innovation can be introduced with less effort. Concrete ways how to operate and orchestrate such systems are not yet 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. Complexity of systems brings the risk of unexpected side-effects and has so long been a show-stopper for administrators applying changes to networking devices. This paper suggests the adaption of the successful concept of continuous delivery into the software defined networking world. Test-driven development and automatic acceptance tests demonstrate that the software engineering community already found ways to ensure that changes do not break. 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.**

I. Introduction

The introduction of Software Defined Network (SDN) has ushered networks in an agile 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 or a Graphical User Interface. Thus, there is no simple way to test the complete network configuration before applying it to all devices. The network engineer has to ensure to always enter the correct commands, make no typing errors and hopefully experience no unforeseen side-effects. This cumbersome process causes severe problems for the management of today's networking infrastructure, as the possibility to break the production network results in security updates being applied late or never, e.g. recently for the Heartbleed bug [1], [2].

Currently, updates or configuration changes often happen in scheduled maintenance periods, when users are informed of possible complications. This approach tries to increase the Mean Time Between Failure (MTBF), as downtimes are avoided by holding off changes until necessary. At the same time, the risk of failure is increased, as multiple changes are applied at the same time. Additionally, identification the root causes of problems becomes more challenging, when more than one change is introduced to the network at the same time.

Software developers experienced similar problems before agile methods were introduced. Methods from *DevOps* allow short feedback cycles and frequent releases to avoid misconception, extensive manual testing, and failures that are hard to identify, when many changes are applied simultaneously. Instead, modern web companies like *GitHub* [3] or *Amazon.com* [4], release changes more often into production – in number of tens or hundreds per day. Such practices aim to increase availability by reducing the Mean Time To Repair (MTTR) instead of increasing MTBF. Through automated tests and deployments, the quality assurance efforts per change can not only be reduced, but also the time to release a fix into production is decreased.

While there exists network configuration management software that allows automatized deployment of changes to network devices, the proprietary nature of configuration interfaces in traditional network elements results in a high complexity and price for such tools. Thus, many modern networks are not using any of these tools and still maintained manually with configuration saved decentralized and no ability to roll back.

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 performance or reliability by defining controllers architectures and roles of entities [5] or generally the more sophisticated management of network flows [6]. 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 networks comprising SDN and the related concept of Network Functions Virtualisation (NFV). Expected results are higher agility and effortless and risk-free deployment of new networking software.

This paper is structured as follows: Section II introduces the technical background, which consists of the networking concepts SDN and NFV, as well as CD as a software engineering mechanism. Section III 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 IV. Section V concludes this work.

II. Background

This section introduces techniques from network and software engineering to build the foundation for understanding the contribution of this paper – applying successful techniques from software engineering to the problems of network management.

A. SDN and NFV

The Software Defined Networking (SDN) concept suggests the externalization, centralization and softwarization of the network control plane into an SDN controller software – a shift away from traditional networking towards 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 [7]. By the controller running as software application on a standard server, faster innovation cycles for the network control plane is possible than with current integrated devices, where firmware

updates must be installed to introduce new mechanisms or protocols. The pace of innovation can already be seen by the number of available OpenFlow controller implementations. The plugin architectures of modern SDN controllers allow easy extension such as monitoring, routing, security, application awareness and quality of experience optimizations.

NFV [8] aims at replacing network functions provided by monolithic hardware middleboxes with software implementations. Overdimensioning of resources can be avoided by applying elasticity mechanisms of cloud application stacks, including cluster and replication functionality. NFV promises slim software instances that provide a particular functionality, similar to the microservices approach in software engineering. Examples include deep packet inspection, firewalling or functionality of mobile networks [9]. SDN is the preferred mechanism to pipe all or only specific parts of the traffic through a specified set of network functions. The *VNF forwarding graph* [10] 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.

All the promises of increased flexibility by frequently modifying the network software and configuration come with the risk of breaking the network. Besides bugs and breaking changes in software releases, the risk of human errors are reasons, why traditional networks are progressing only very slowly. The question is now, how virtualization of network elements allows frequent changes without increasing the risk of unexpected outages. Frequent releases of software and short innovation cycles however are also a goal of application software developers. One concept that allows frequent deployment of new software releases while maintaining high quality is *continuous delivery*.

B. Continuous Delivery

To mitigate the risk of broken software deployments while keeping a high pace of software releases, continuous delivery ([11]) 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. An important principle is that in case of failure, the team making the changes, the *delivery team*, receives fast feedback. 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 in a centralized software like *Jenkins* [12] or *Go CD* [13] are illustrated in Figure 1 and described as follows:

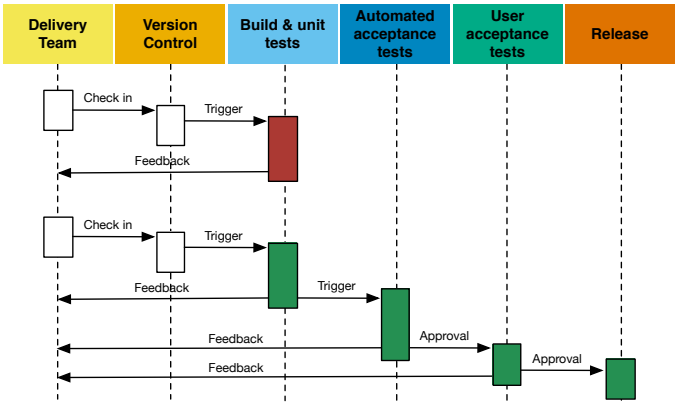

Fig. 1. The deployment pipeline for software projects as suggested by [11]. Red: Test execution resulted in failure; green: tests succeeded.

Version control: Every change to the software is checked into a Version Control System (VCS), e.g. *Git*. The VCS prevents overwriting changes of source code files, when multiple developers modify the same file. It also provides access to historic versions of the source code and allows to revert back to any state. After checking a change into the VCS, the deployment pipeline is instantiated and the state of the software code is passed through the following stages.

Build & unit tests: A build server picks up the change and creates an executable build artifact by compiling the source code. By creating the build on a centralized server, it is ensured that the whole team is able compile and release software. After successfully creating the build, unit tests are executed following the technique of Test-Driven Development (TDD). In case of any error, the pipeline is stopped and humans informed about the issue. Fast feedback regarding any failure is important for efficient software development. On success, the next stage is triggered.

Automated acceptance tests: The particular steps in this stage depend on the actual implementation of the deployment pipeline. The final result, however, is the knowledge whether the created build artifact meets specified acceptance criteria. Such criteria range from functional and integration tests to capacity tests, and longer-lasting source code analysis. Functional tests ensure that the functionality of a software meets its specification. Integration tests ensure that a part of the software works after integration with other components. To execute all of these tests, a running version of the software is required, which is therefore deployed into a production-like system. This means that the environment, where the tests run, matches the production environment in terms of version and configuration of operating system, libraries, and installed software. If all tests pass, the pipeline is continued, otherwise the team is notified to adjust.

User acceptance tests: The Quality Assurance (QA) sign-off, when testers manually verify the functionality of the software, is the first human intervention after checking into the VCS. As extensive automated tests in the previous steps verified the basic feature set, the QA team has now more time to focus on new features and exploratory testing. After these manual tests, the build artifact is ready for release.

Release: Finally, the software release is installed on production servers or, in case of on-premise software, it is made available for customers. This can be triggered manually in the CD software or completely automated.

The result of this process is the knowledge of the last state of the software that is known to work, be it the new or an older build. Automated tests reduce manual QA effort and duration that a change takes to pass through the deployment pipeline.

Besides to software development, the continuous delivery paradigm has been successfully applied to other areas. Modern configuration management software, e.g. *Chef* [14], follow the *infrastructure as code* paradigm to define the setup of an entity, mostly a server including its application stack, as source code. Any change to the configuration stored in a VCS goes into production only if the new setup passes the pipeline.

III. Continuous Delivery of Network Functions

Next, the suggested adaption of continuous delivery to network operations is described. This does not mean that network administrators will now become or be replaced by software developers. Instead, network engineers should learn from experiences made in software development. Similar

to the configuration changes of servers, any change to the network brings a risk of introducing regressions, e.g. changes of configuration settings or the deployment of a new network software version. During the deployment of an additional component, e.g. a virtual router that is inserted into the VNF forwarding graph, misconfiguration and incompatibilities entail risk of network outages.

The contribution of this paper is therefore to apply the concept of *continuous delivery* to the software defined networking area. The main building blocks are: 1) *Process automation* supports engineers in their daily work and decreases the work required to make a certain change. Instead of manually logging into all networking devices, deployments happen automatically on affected entities. 2) *Automated tests* further support the goal of reducing the overhead, in this case extensive manual testing. Furthermore, automated tests bring (high) confidence that any change that passed the tests will not be disruptive after deployment to production. 3) *Realistic test environments* that can be set up automatically are a prerequisite of executing automated tests. Compared to traditional networking, where devices have to be connected physically, softwarization now allows automated setup of environments matching production through instantiation of virtual machines running the same software. 4) *Infrastructure as code* ensures that configuration is not only documented as code, but also that this code can be applied to another system to reproduce an identical configuration. This technique allows instantiation of new development and testing environments without manual intervention.

In the following, the transfer of the CD concept to network functions is described. The stages of the resulting deployment pipeline are shown in Figure 2. It is illustrated in the following using an SDN controller software as example. A second use case can be found in the extended version of this paper [15].

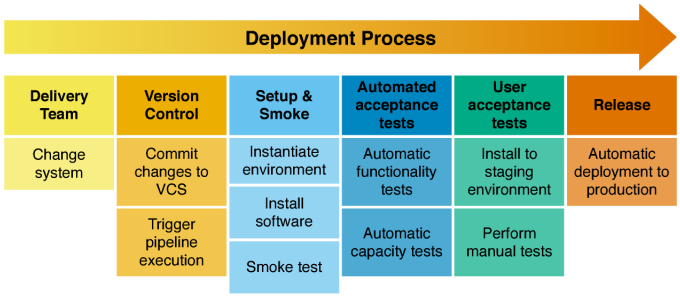

Fig. 2. Suggested deployment pipeline for network functions.

The controller plays a critical role in an SDN network, as an outage would make the network "headless". Situations like when an administrator supplies an invalid configuration resulting in the service to not restart for a time frame of several seconds or minutes have to be avoided. Therefore, error-prone manual configuration in production has to be succeeded by automated processes 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 usual case when an administrator uses an existing controller software and configures it according to own needs. It does not matter, if the controller is open source or commercial software, as only the binary is deployed. 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 versions of controller and plugins that have to be installed, as well as the configuration files.

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 production 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*, e.g. *Chef* or *Puppet*.
Following the principle of fast feedback, this stage is restricted to tests that can be executed in very short time, but catch many of the most likely errors. In [11] it is recommended to not exceed the 10 min mark. *Smoke tests* verify here whether the controller software is able to start successfully using the supplied configuration and plugins.

Automated acceptance tests: Tests of this phase ensure that the specified acceptance criteria for the controller are met. A failure in any of the tests means that at least one of the criteria defined as essential for operation is not fulfilled and thus the pipeline has to be stopped. The functional test suite of an SDN controller should include at least the following checks: 1) Accepts incoming southbound connections, 2) allows a switch to forward traffic, 3) accepts incoming northbound connections, 4) required feature set of northbound API is working. Additional non-functional tests evaluate further requirements like the ability to handle a certaion load, in terms of connected switches as well as a rate of requests. Therefore, the controller software is set under a certain load that matches at least the expected peak load of control traffic of the production network, e.g. using a controller benchmarking software [16], [17].

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 administrators or QA engineers in the staging environment allow detailed manual inspection. The tester can connect a virtual switch, or even a hardware switch, to the controller software running in the staging environment. Additional virtual machines can be deployed and preconfigured 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 be automatized and executed in the previous stage. This allows the manual tests to concentrate in detail new functionality and not on basic functionality.

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. Instead, small, incremental change is desirable.

IV. Discussion

This section briefly discusses the aspects that are in the authors' opinion worth noting to understand the reasons for introducing CD for networks. A more detailed discussion can be found in the extended paper version [15].

A. Releasing More Frequently

For agile software development, the "highest priority is to satisfy the customer through early and continuous delivery of valuable software" [18]. Increased quality and productivity through introduction of agile methods in software engineering is reported in [19]. More releases do not mean more bugs, instead that fixes are released faster is observed for *Mozilla Firefox* [20]. The number of outages triggered by software deployments for *Amazon.com* was reduced by 75% [4]. A safety net provided by automated deployments backed by tests instead reduces stress for humans. In case of failures, the smaller amount of changes deployed eases the root-cause analysis or rollback.

B. Adaption of Behavior-Driven Development to Networks

Behavior-Driven Development (BDD, [21]) allows also non-technical people to write specifications in a natural language style domain specific language called *Gherkin*. Giving such a language with an implementation for the used SDN technique, e.g. OpenFlow, network administrators could far easier adopt the concepts of TDD and continuous delivery without knowing implementation details. An example of such a behavior specification of an SDN controller is shown below.

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

Listing 1: BDD specification of a reactive controller behavior.

C. Metrics

An essential part of CD and DevOps practices is the collection if metrics, which means more than just monitoring of bandwidth usage and QoS [22]. Instead, a data-driven culture relies on aggregating data in order to allow engineers and business units to make decisions based on measured truth instead of assumptions. While a changed network parameter does not necessarily result in a change of measured QoS, it can affect performance of applications running on top. Collection and aggregation of metrics can also include the number of transactions in an online shop, where a large decrease of this metric and network metrics can be correlated as well as matched with the time of deployments. If a certain behavior is seen after a change to the network or server infrastructure, this change is likely to be the cause.

V. Conclusion

As every change introduces a risk of failures, traditional networks are hardly changed. Softwarization of networks allows agile methods also for management of IT infrastructure, which, however, have to be supported by automated processes including tests that bring the certainty that configuration changes or updates will not harm. This paper described the established concept of continuous delivery, an important technique for agile and high-quality software development. The adaption of this process to software-based networks in order to use agile methods and risk-free deployments is the main contribution. Usage of the infrastructure as code paradigm followed by automated deployments of infrastructure ensures availability and consistence of testing, staging and production environments. Automated acceptance tests verify every change prior to deployment into production infrastructure automatically, which then happens automatically or through the push of a button. This process was illustrated by describing the deployment pipeline of an SDN controller, followed by a brief discussion.

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. (2014, August) CHS Hacked via Heartbleed Vulnerability. [Online]. Available: http://bit.ly/1to0fCd

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

[3] J. Douglas. Deploying at GitHub. [Online]. Available: https://github.com/blog/1241-deploying-at-github

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

[5] R. Ahmed and R. Boutaba, "Design considerations for managing wide area software defined networks," *Communications Magazine*, Jul. 2014.

[6] H. Kim and N. Feamster, "Improving network management with software defined networking," *Communications Magazine*, Feb. 2013.

[7] M. Jarschel, F. Wamser, T. Höhn *et al.*, "SDN-based Application-Aware Networking on the Example of YouTube Video Streaming," in *European Workshop on Software Defined Networks*, Berlin, Germany, Oct. 2013.

[8] ETSI, "Network Functions Virtualisation (NFV); Archit. Framework."

[9] S. Gebert, D. Hock, T. Zinner *et al.*, "Demonstrating the Optimal Placement of Virtualized Cellular Network Functions in Case of Large Crowd Events," in *ACM SIGCOMM*, Aug. 2014.

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

[11] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, 2010.

[12] Jenkins CI server. [Online]. Available: http://jenkins-ci.org

[13] go continuous delivery. [Online]. Available: http://www.go.cd

[14] Chef. [Online]. Available: http://www.chef.io/chef/

[15] S. Gebert, C. Schwartz, T. Zinner *et al.*, "Agile management of software based networks," University of Würzburg, Tech. Rep. 493, 2015. [Online]. Available: http://www3.informatik.uni-wuerzburg.de/TR/tr493.pdf

[16] M. Jarschel, C. Metter, T. Zinner *et al.*, "OFCProbe: A Platform-Independent Tool for OpenFlow Controller Analysis," in *IEEE International Conference on Communications and Electronics*, Aug. 2014.

[17] P. Wette, M. Draxler, and A. Schwabe, "Maxinet: distributed emulation of software-defined networks," in *Networking Conference*, 2014.

[18] K. Beck, M. Beedle, A. van Bennekum *et al.*, "Manifesto for agile software development," 2001.

[19] A. Ahmed, S. Ahmad, N. Ehsan *et al.*, "Agile software development: Impact on productivity and quality," in *IEEE ICMIT*, Jun. 2010.

[20] F. Khomh, T. Dhaliwal, Y. Zou *et al.*, "Do Faster Releases Improve Software Quality? An Empirical Case Study of Mozilla Firefox," in *IEEE Working Conference on Mining Software Repositories*, Jun. 2012.

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

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