Teaching network softwarization with SDN Cockpit: An open ecosystem for students, network administrators and others

Robert Bauer, Hauke Heseding, Addis Dittebrandt, Martina Zitterbart[1]

Abstract: This paper introduces SDN Cockpit, an easy-to-use and open ecosystem for teaching network softwarization based on mininet and the Ryu controller. The ecosystem allows candidates to gain hands-on-experience with SDN in prefabricated scenarios without having to deal with potentially complex details such as traffic generation. It provides useful tooling for instructors and automated evaluation for assignments. The paper discusses the design goals, the architecture and the workflow of the ecosystem. First experiments with SDN Cockpit show that the approach can improve the motivation and the learning experience of the candidates.

Keywords: Network Softwarization; SDN Cockpit; Teaching

1 Introduction

Network softwarization describes a recent trend to enable flexibility in the network through logically centralized software-based control and virtualization. Software-defined Networking (SDN) is one important key technology in this domain. SDN decouples the control plane and the data plane from each other and allows remote programmability of forwarding devices. It attracted significant attention since it was initiated, but many of the underlying ideas can be traced further back (up to the early days of the internet [FRZ14]).
SDN comes with a number of enticing promises: rapid development of new features, fast time-to-market, purchase and maintenance of hardware and software independently of one another to reduced CAPEX/OPEX, and many others. It is thus not surprising that SDN stepped into datacenters or mobile backbone networks, with no intention to leave anytime soon. Consequently, we believe that at least the core ideas behind SDN will demand their fair share in future networking classrooms.

One thing that is particularly intriguing about SDN: it is an excellent topic for teaching network fundamentals, primarily because it is very easy to get your hands dirty and gain practical experience – especially compared to many other technologies in the networking domain. Understanding the basic concepts behind SDN is relatively simple due to well defined abstractions and a clean three-layered architecture. And a wide variety of powerful tools for rapid prototyping such as mininet [LHM10] exist to assist with practical realizations.

[1] Karlsruhe Institute of Technology, Institute of Telematics, Karlsruhe
robert.bauer@kit.edu, heseding@kit.edu, addis.dittebrandt@student.kit.edu, zitterbart@kit.edu

This holds for a variety of “candidates” to be teached, including network administrators in computer and data centers as well as students in basic and advanced networking classes.

In the past three years, we supervised several SDN-related classes with practical SDN assignments and noticed that a large part of the participants developed a sound understanding of the topic within a tight timeframe, just by experimenting with the technology (which is good news). Approximately three-quarters of the participants were capable of autonomously setting up basic SDN scenarios (learning switch, simple routing, simple load-balancing) with existing tool chains, e.g., based on mininet and the Ryu controller.

However, we also got reports from participants who did not finish the (partly voluntary) assignments. Especially participants with little programming experience and/or affinity tend to give up early. In most cases, we were able to identify one or multiple of the following three obstacles as root cause for the inability to finish an assignment: 1) the perceived complexity of several interacting tools (network emulator, traffic generator, SDN controller), 2) inability to find/identify required pieces of information to correctly utilize the tools and 3) the absence of a clear success metric (do I have the correct solution?). Based on this observation, we argue that there is a demand for an SDN ecosystem that focusses not only on rapid prototyping but also on easy accessibility and the capability to draw the attention to the essential core of SDN. Unfortunately, most existing tools do not focus on these objectives.

In this paper, we first introduce our vision of an open and easy-to-use ecosystem for teaching network softwarization and discuss important design goals. After that, we introduce our prototype for such an ecosystem called SDN Cockpit. The tool is publicly available on GitHub [SD18]. We outline the general architecture of the ecosystem, its workflow and present first insights from practical use.

2 Towards an ecosystem for teaching network softwarization

As outlined in the introduction, we envision an open and easy-to-use ecosystem that allows it to teach the ideas of network softwarization – with a current focus on SDN – to a broad audience. We assume partially supervised self-teaching, i.e., the users of the ecosystem work with a prepared set of assignments following a “learning by doing” approach. This is partially supervised, because the assignments have to be prepared somehow and the ecosystem itself takes the role of a supervisor. We further assume that many learning objectives in the context of network softwarization can be achieved by experimenting with the control plane, or, to be more precise, by developing control plane applications[2].

From a high level perspective, we consider two types of stakeholders (see also Figure 1): *Instructors* capable of setting up assignments for a given learning objective (e.g., understand

[2] Note that there are also concepts of network softwarization that require a different angle, e.g., middlebox virtualization or data plane optimizations. Such concepts are currently not supported by SDN Cockpit.

reactive flow programming, understand multipath routing, . . .) and *candidates* – say students or network administrators – that use these assignments. The ecosystem then consists of two major parts: a frontend part which must be visible to the candidates and a separate backend part which might or might not be visible. There is a human machine interface between the candidates and the frontend (to work on assignments) and another interface between the instructor and the backend of the ecosystem (e.g., to include new assignments). We provide further details when discussing the general architecture of our prototype in Section 3.1. We will now go through several design goals for the ecosystem in greater detail, followed by a short discussion for two especially important design decisions.

2.1 Design Goals

Easy accessibility: The steps or preparations necessary from a candidate to get started with the first assignment should be as low as possible, which includes initial access to the ecosystem, access to the frontend and selection of an assignment. If these steps consume too much time or the setup procedure is too complex – e.g., because several dependencies have to be installed –, the initial hurdle for using the ecosystem might be too high. In fact, we see only three possible deployment schemes that can fit these criteria: 1) Remote access to the frontend, e.g., in the form of a web interface, 2) provisioning in the form of a consolidated stand-alone program (single dependency) or 3) provisioning in a container with the help of virtualization. We also use the term accessibility to refer to the general effort that is required by a candidate to get started with an assignment. In the best case situation, the candidate can edit a solution and gets immediate feedback to the changes.

Easy transition: The ecosystem should be designed in such a way, that an easy transition to real technology is possible without too much effort. That means, that the candidate can “leave” the ecosystem to continue experimentation without potential limitations and restrictions of the ecosystem (enforced by other design goals). In the case of SDN, this would mean that the applications written in an assignment can be executed using only the underlying SDN controller (without the ecosystem!) or that the traffic can be replicated by stand-alone scripts or generators.

Auditability of solutions: Providing a clear success metric to candidates is critical. The ecosystem should thus not only provide the assignments, it should also trace the current state of an assignment and provide positive and/or negative feedback to candidates. The latter aspects have to be fully automated to avoid having an instructor in the loop. Such a design has several benefits: 1) the candidate has a clear objective and is able to autonomously (and transparently!) trace the progress of the current assignment, 2) a continuous feedback loop between the candidate and the ecosystem can assist the learning process and 3) it enables auditing capabilities for instructors if required.

Flexible scenarios: To support a broad range of different learning objectives, the ecosystem has to support different topologies and different traffic profiles. A traffic profile is an abstract formulation of packets that are exchanged on top of a topology. In this context, we refer to an *integrated scenario* as the combination of a topology and a traffic profile generated specifically for this topology. Interesting integrated scenarios for SDN include load balancing scenarios (flow demand has to be scheduled to multiple links) or inter-domain routing scenarios (policies have to be applied to flows). Note that simple traffic generators like iperf might be not sufficient here, e.g., in the case of customized source and destination addresses.

Other design goals: A clean and modular design of the ecosystem helps with extendability. External dependencies (e.g., controllers) should be integrated in a lightweight fashion to support easy transition. Platform independence is also required so that candidates can use their habitual operating system. Last but not least, we are convinced that an ecosystem for teaching network softwarization is more likely to be successful if it openly approaches the community, which includes free access to the source code and the assignments.

2.2 Important Design Decisions

One major design issue is related to the technical core of the ecosystem, i.e., the question whether to use *emulation* or *simulation*. The latter has the big advantage that the ecosystem would be more controlled, deterministic and independent of the underlying hardware (better auditability). On the downside, existing simulators with SDN support such as ns-3 or OMNeT++ are restricted to an internal controller API [Iv16] which might suffer from a scarcity of examples and – more important – prevents easy transition. Emulation, on the other hand, is directly based on real networking stacks and compatible with any SDN controller which results in an authentic experience for the candidates (better transition). Because easy transition is one of the central design goals and both approaches can be designed for easy accessibility, we decided to use emulation based on the mininet tool [LHM10]. Another important design issue is the choice of the controller architecture. While production-grade controllers such as ONOS and OpenDaylight offer high availability, resiliency and performance, the complexity of the application programming interface of Python-based controllers like Ryu is much lower. In this case, easy accessibility outweighs easy transition so that we chose Ryu [Ry18] as the controller that best fits into our ecosystem.

3 SDN Cockpit

SDN Cockpit was designed following the goals outlined in Section 2.1. It is based on the general idea, that the candidates are confronted with the lowest possible amount of distraction. In fact, they have to deal with only two simple interfaces while doing an assignment: a text editor they are familiar with (to edit the solution) and an aggregated view summarizing all currently required pieces of information. Everything else is hidden by the ecosystem. In the following, we discuss the architecture, the workflow and first insights from practical use.

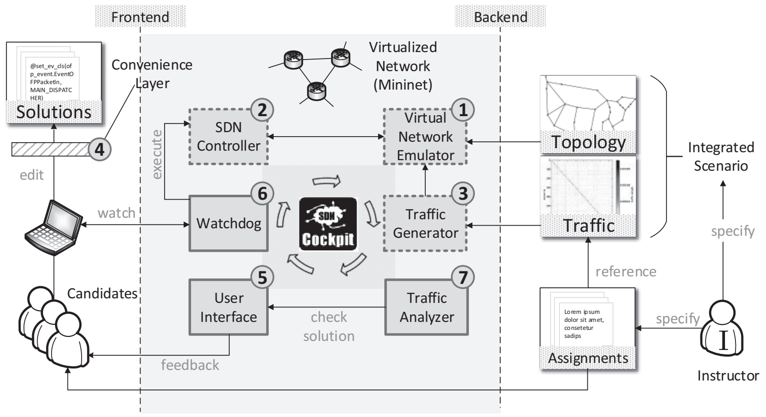


Fig. 1: Architecture of SDN Cockpit

3.1 Architecture

The overall architecture of the ecosystem is given in Figure 1. SDN Cockpit consists of seven components and two interfaces, one for candidates (frontend) and one for instructors (backend). The processes within the grey area between the two interfaces form the ecosystem. The virtual network emulator, the SDN controller, and the traffic generator are based on unmodified open-source projects (① to ③). The convenience layer, the user interface, the watchdog, and the traffic analyzer are new components introduced by SDN Cockpit (④ to ⑦) . The components are woven together with a set of Python scripts represented by the black icon in the middle. The whole ecosystem is usually bundled within a virtual machine and can be deployed with a single call to the command line using vagrant [Va18]. In the following, we go through the individual components of the architecture and describe their role in the ecosystem.

Virtual Network Emulator : The virtual network emulator – mininet in our case – represents the *data plane* of the ecosystem, i.e., it provides a set of SDN-capable switches connected according to a predefined topology. Despite being virtual, the switches are required to possess the same functional characteristics as standard OpenFlow-compliant switches in order to ensure an authentic experience and support transition. Furthermore, mininet can integrate virtual hosts into the network and virtualization techniques provide each virtual host with its own isolated execution environment. The emulator is configured by the instructor, which includes a topology and its parameterization (bandwidth and latency constraints). In order to support auditability, the emulator provides feedback on the success or failure of network-targeted operations to the candidate via the user interface (not shown in the figure).

SDN Controller: The controller represents the *control plane* of the ecosystem. The applications (= solutions) provided by the candidates are executed here. The controller itself is unmodified, but the SDN Cockpit ecosystem takes care of several basic tasks usually required for using the controller properly. The ecosystem might also rely on partial pre-configuration. For example, connectivity on lower network layers can be provided beforehand when this networking aspect is not of concern. A candidate can then focus on the aspects of network programming that are of primary concern in a particular assignment. SDN Cockpit uses the Ryu controller framework due to reasons outlined in Section 2.2.

Traffic Generator: This component is responsible for injecting genuine packets into the network. Automating this process is essential for easy accessibility, because the interaction between traffic generator and network emulator can be very complex, especially for sophisticated scenarios. In SDN Cockpit, we currently use the trafgen tool [ne18] controlled by a set of Python scripts. This solution is highly customizable and allows for a wide range of networking scenarios, including, for example, different protocols, variations in bandwidth utilization, timed emission of packets, and randomized packet contents. Furthermore, in conjunction with the virtual network emulator a variety of communication relations can be modeled, since the emulator can isolate several instances of traffic generators in a given topology from each other.

Convenience Layer: We observed that eliminating undesired complexity can be a key enabler to improve acceptance and usability for instructors and learning effectiveness for candidates. Hence, we introduce a convenience layer to simplify interaction with the ecosystem. This simplification is twofold:

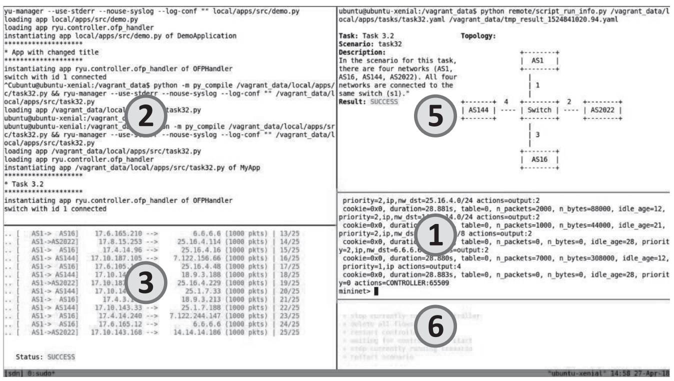
a) The programming interface for the candidates is reduced to the functionality that is required to achieve a desired goal. The interface against which the candidates implement their solutions is a set of functions that are essentially derived from the programming interface of the SDN controller. However, the complex syntax, object structure and unnecessary controller internals remain hidden.

b) Instructors are provided with a convenient set of configuration options that can be used to easily create new assignments. The current configuration interface provides options to define traffic flows that should occur during an assignment, e.g., static flows (ensure accurate reproducibility) or randomized flows to emulate dynamic network behavior. These options are presented in a comprehensive configuration format so that an instructor does not need to be aware of the different configuration options of the various tools (e.g., complex traffic generator configuration files or parameters of a network emulator).

User Interface: While the convenience layer accepts input from the candidates, the user interface is the primary source of feedback. A screenshot of the interface is given in Figure 2. It is structured to simultaneously display output from the controller as well as from the traffic generator and analyzer. Through this aggregated view it is possible for a candidate to comprehend how the network reacts to the current solution. The controller can provide feedback whether or not a control action targeted at the network was successfully executed. Furthermore, the forwarding behavior in the network itself can be observed from the output of the traffic analyzer. A correct solution is immediately indicated by signaling success to the candidate. Faulty network behavior, on the other hand, will result in inaccurate packet counts, either through packets arriving at unintended destinations or not being forwarded at all. In any case, the candidate is presented with a metric that represents the degree of success, i.e., the number of correctly forwarded packets. This feedback can be used to incrementally improve the solution.

Watchdog: The watchdog continuously monitors changes to the solution made by the candidate. Once a change has been detected, it is the watchdogs' responsibility to restart certain components of the ecosystem automatically. This involves resetting the state of the entire network, i.e., the SDN controller, the virtual network emulator, as well as traffic generation and analysis. The automization of this process relieves the candidate from the tedious task of managing the network state himself to ensure undistorted results. The watchdog is essential to provide easy accessibility and auditability of solutions.

Traffic Analyzer: The traffic analyzer component monitors the packets that are sent and received at communication endpoints throughout the network. As soon as the traffic generation process for one of the integrated scenarios is complete, it evaluates the correct forwarding behavior based on the expected packet count for each endpoint. These counts are determined dynamically through monitoring of the traffic during the generation phase so that even randomized packets will produce accurate results. Randomization of traffic can be used to model the unpredictable behavior of real networks. Hence it serves to facilitate our design goal of an easy transition. Special care must be taken during the collection of packets, since management traffic generated by the virtual network emulator itself can



```
yu-manager --use-stderr --nouse-syslog --log-conf "" local/apps/src/demo.py
loading app local/apps/src/demo.py
loading app ryu.controller.ofp_handler
instantiating app local/apps/src/demo.py of DemoApplication
********************
* App with changed title
********************
instantiating app ryu.controller.ofp_handler of OFPHandler
switch with id 1 connected
^Cubuntu@ubuntu-xenial:/vagrant_data$ python -m py_compile /vagrant_data/local/apps/
c/task32.py && ryu-manager --use-stderr --nouse-syslog --log-conf "" /vagrant_data/l
ocal/apps/src/task32.py
loading app /vagrant_data/local        task32.py
ubuntu@ubuntu-xenial:/vagrant_d
ubuntu@ubuntu-xenial:/vagrant_da          n -m py_compile /vagrant_data/local/apps/sr
c/task32.py && ryu-manager --use-s       --nouse-syslog --log-conf "" /vagrant_data/l
ocal/apps/src/task32.py
loading app /vagrant_data/local/apps/src/task32.py
loading app ryu.controller.ofp_handler
instantiating app /vagrant_data/local/apps/src/task32.py of MyApp
********************
* Task 3.2
********************
instantiating app ryu.controller.ofp_handler of OFPHandler
switch with id 1 connected
```

```
.. [   AS1->  AS16]    17.6.165.210 -->         6.6.6.6 (1000 pkts) | 13/25
.. [   AS1->AS2022]     17.8.15.253 -->     25.16.4.114 (1000 pkts) | 14/25
.. [   AS1->  AS16]      17.4.14.96 -->      25.16.4.16 (1000 pkts) | 15/25
.. [   AS1-> AS144]   17.10.187.105 -->    7.122.156.66 (1000 pkts) | 16/25
.. [   AS1->  AS16]    17.6.105.                 25.16.4.48 (1000 pkts) | 17/25
.. [   AS1-> AS144]    17.10.14                  18.9.3.188 (1000 pkts) | 18/25
.. [   AS1->AS2022]   17.10.18                  25.16.4.229 (1000 pkts) | 19/25
.. [   AS1-> AS144]    17.10.14                   25.1.7.33 (1000 pkts) | 20/25
.. [   AS1->  AS16]      17.4.3.                 18.9.3.213 (1000 pkts) | 21/25
.. [   AS1-> AS144]    17.10.143.33 -->      25.1.7.188 (1000 pkts) | 22/25
.. [   AS1->  AS16]     17.4.14.240 -->   7.122.244.147 (1000 pkts) | 23/25
.. [   AS1->  AS16]     17.6.165.12 -->         6.6.6.6 (1000 pkts) | 24/25
.. [   AS1->AS2022]  17.10.143.168 -->    14.14.14.186 (1000 pkts) | 25/25


   Status: SUCCESS
[sdn] 0:sudo*
```

```
ubuntu@ubuntu-xenial:/vagrant_data$ python remote/script_run_info.py /vagrant_data/l
ocal/apps/tasks/task32.yaml /vagrant_data/tmp_result_1524841020.94.yaml

Task: Task 3.2                Topology:
Scenario: task32
Description:                                 +--------+
In the scenario for this task,               |  AS1   |
there are four networks (AS1,                +--------+
AS16, AS144, AS2022). All four                   |
networks are connected to the                    | 1
same switch (s1)."                               |
Result: SUCCESS                  +-------+  4  +--------+  2  +--------+
                                 | AS144 | ---- | Switch | ---- | AS2022 |
                                 +-------+     +--------+     +--------+
                                                 |
                                                 | 3
                                                 |
                                             +--------+
                                             |  AS16  |
                                             +--------+
```

```
 priority=2,ip,nw_dst=25.16.4.0/24 actions=output:2
 cookie=0x0, duration=28.881s, table=0, n_packets=2000, n_bytes=88000, idle_age=12,
priority=2,ip,nw_dst=1      14.0/24 actions=output:2
 cookie=0x0, duratio          table=0, n_packets=1000, n_bytes=44000, idle_age=21,
priority=2,ip,nw_ds          /8 actions=output:2
 cookie=0x0, duratio          table=0, n_packets=0, n_bytes=0, idle_age=28, priorit
y=2,ip,nw_dst=6.6.6.6       s=output:2
 cookie=0x0, duration=28.880s, table=0, n_packets=7000, n_bytes=308000, idle_age=12,
 priority=1,ip actions=output:4
 cookie=0x0, duration=28.883s, table=0, n_packets=0, n_bytes=0, idle_age=28, priorit
y=0 actions=CONTROLLER:65509
mininet> █
```

```
 * stop currently ru          troller
 * delete all flow
 * restart controll
 * waiting for contr          start
 * stop currently running scenario
 * restart scenario
                                         "ubuntu-xenial" 14:58 27-Apr-18
```

Fig. 2: SDN Cockpit user interface with SDN controller (2), traffic generator and analyzer (3), assignment info (5), mininet emulator (1) and watchdog (6)

interfere with packet count accuracy. The traffic analyzer provides immediate feedback to a candidate once traffic generation and analysis have been completed. Hence, the candidate can evaluate the feasibility of his solution in a timely manner.

3.2 Workflow Example

This section presents an example workflow showing all the steps of using the ecosystem from assignment creation by the instructor to submission of the solution by the candidate. SDN Cockpit is distributed either via git or as a standalone vagrant image which can be downloaded from an online repository. Installation of the tool is performed by executing `vagrant up` either inside the git directory or with the image path as an argument inside a new directory. In the following, we assume that the candidate has installed the environment already.

The instructor first has to create an assignment geared towards a specific learning goal. We choose ”policy based routing“ as an example. The assignment is to route traffic of an Autonomous System (AS) based on a static set of policies. It consists of an assignment text and an integrated scenario specifically designed for the assignment so that the solutions can be tested. The assignment can be delivered automatically by the ecosystem (see ⑤ in Figure 2). The candidate launches SDN Cockpit by executing `vagrant ssh` followed by `./run.sh`. The frontend is divided into 5 panes: a pane showing the output of the ryu controller ②, a pane illustrating traffic generation and traffic analysis ③, a pane for assignment information ⑤, a pane for interacting with mininet ① and a pane for watchdog activity ⑥. The assignment information pane shows essential information such as a textual description (e.g., which AS should be prioritized), a visualization of the topology and a mapping of network subnets to ASes and switch ports. The latter is required for flow programming.

The candidate proceeds with solving the assignment by opening the corresponding solution file externally with a text editor. It contains an initial ryu application skeleton which is not a valid solution to the assignment (yet). The skeleton integrates functions of the convenience layer tailored to the assignment. In this case, it would include helper functions for easy flow programming. The candidate then writes an initial solution which incorrectly programs the routing policies. Upon saving, the watchdog ⑥ shows that the saved file has been registered and a test run is executed. Pane ② can be consulted for the controller output. It can be used to show abnormal behavior of the application through logs and exceptions. Once the traffic generator terminates, the analyzer will check how many packets have been received by which AS. As the solution does not yet work properly, the traffic analyzer detects this violation through deviating packet counts and presents a failed test case to the candidate. To further dissect the problem, the candidate can use the mininet CLI ① to manually send traffic and observe the reaction of the application in pane ② or by dumping flow statistics through the mininet CLI. This loop is repeated until the solution passes the tests. Small incremental steps towards a valid solution are encouraged with this approach as short test runs are executed immediately upon saving which results in fast feedback.

Tab. 1: Success rate for two (slightly different) sets of assignments from 2017 without SDN Cockpit and 2018 with SDN Cockpit. Failure indicates that a candidate missed the learning objective.

without SDN Cockpit 2017, n=13			
	Submitted	Success	Failure
Easy	100 %	100 %	0 %
Medium	92 %	69 %	23 %
Difficult	0 %	/	/

with SDN Cockpit 2018, n=17			
	Submitted	Success	Failure
Easy	100 %	94 %	6 %
Medium	94 %	94 %	0 %
Difficult	65 %	53 %	12 %

3.3 Experience Report

So far, we have successfully used the SDN Cockpit ecosystem for two different activities: a completely voluntary assignment as part of an advanced networking class (10+ candidates, not discussed here further) and an obligatory assignment for a practical course with 17 candidates. The latter one included three tasks with increasing difficulty (easy, medium and difficult). To have some kind of comparison, we take the results from an earlier SDN-assignment with 13 participants that was conducted in early 2017 with a very similar setup but without SDN Cockpit: same background, same time frame, same technology, similar size/difficulty of the tasks. The students manually installed the required software (Ryu, mininet) and used ping/iperf for basic traffic generation. In both cases, the third and most difficult task could be submitted on a voluntary basis. Table 1 shows a high level analysis of the results.

In both cases, the majority of the candidates were able to meet the learning objective for the easiest task (basic flow programming). For the task with medium difficulty (still flow programming but in a more complex scenario), there seems to be an improvement with regard to the percentage of candidates that were able to meet the learning objective when working with SDN Cockpit – 23% failure rate for 2017 compared to 0% failure rate in 2018. The most important outcome, however, is the percentage of candidates that have worked on the voluntary task. While we received no solution for this task in 2017, 65% of the candidates provided a solution in 2018. Because of the low sample size and the fact that the two groups worked on slightly different tasks, the numbers in Table 1 must be treated with considerable caution. However, at least the overall trend is in line with our personal experience from working with the tool and the participants. In conclusion, we believe that an easy-to-use ecosystem can improve the motivation and the learning experience of the candidates. The SDN Cockpit approach might be a first step in this direction.

4 Conclusion and Future Work

This paper introduced SDN Cockpit as a novel ecosystem for teaching network softwarization. We put easy accessibility and the possibility to transition to real technology as our top design goals and build the ecosystem around them. We gathered first practical experience with the ecosystem in a university context with two different classes and currently plan to extend this in the context of the bwNET100G+ project [bw18] to also include network and system administrators.

While we are pretty pleased with the initial results of SDN Cockpit, there are several aspects that we want to address in the future. First, there is still room for improvement with regard to accessibility. It requires some time to get familiar with the process of assignment selection and the tmux-based user interface. In addition, candidates should be allowed to tune certain aspects of the integrated scenario (topology, traffic) to further improve the learning experience – which is currently not supported. Second, the automatic evaluation of more complex scenarios is difficult to set up. And third, the current deployment scheme takes a non-negligible amount of time and hardware resources. We are currently working on a newer version of SDN Cockpit where the frontend is implemented in the browser and the backend runs in a light-weight container.

Despite these limitations, we are confident that practical experimentation with SDN Cockpit is not only useful for understanding SDN, it can also be a promising ecosystem for getting insights into other related areas, e.g., inter-domain routing, load balancing algorithms or optimization theory. With respect to the latter, it might even be applicable for purposes outside of the domain of communication networks.

References

[bw18] bwNET100G+: Research and innovative services for a flexible 100G-network in Baden-Wuerttemberg. https://www.bwnet100g.de/, 2018. (Accessed on 02/01/2018).

[FRZ14] Feamster, Nick; Rexford, Jennifer; Zegura, Ellen W.: The road to SDN: an intellectual history of programmable networks. Computer Communication Review, 44:87–98, 2014.

[Iv16] Ivey, Jared; Yang, Hemin; Zhang, Chuanji; Riley, George: Comparing a Scalable SDN Simulation Framework Built on Ns-3 and DCE with Existing SDN Simulators and Emulators. In: Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation. SIGSIM-PADS '16, ACM, New York, NY, USA, pp. 153–164, 2016.

[LHM10] Lantz, Bob; Heller, Brandon; McKeown, Nick: A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In: Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks. Hotnets-IX, ACM, New York, NY, USA, pp. 19:1–19:6, 2010.

[ne18] netsniff-ng toolkit. http://netsniff-ng.org/, 2018. (Accessed on 02/01/2018).

[Ry18] Ryu SDN Framework. https://osrg.github.io/ryu/, 2018. (Accessed on 02/01/2018).

[SD18] SDN Cockpit. https://github.com/kit-tm/sdn-cockpit, 2018. (Accessed on 27/07/2018).

[Va18] Vagrant by HashiCorp. https://www.vagrantup.com/, 2018. (Accessed on 02/01/2018).