University of Zurich[UZH]

MASTER PROJECT – Communication Systems Group, Prof. Dr. Burkhard Stiller

Design and Implementation of a System to Enable Automatic VNF Deployment based on Smart Contract Events

Fabian Küffer, Pascal Kiechl, and Niels Kübler
Zürich, Switzerland
Student ID: 15-931-421, 16-927-998, 10-712-123

Supervisor: Eder J. Scheid, Muriel Franco
Date of Submission: March 14, 2022

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Master Project
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

The emergence of Network Function Virtualization (NFV) has yielded significant advantages over traditional networking approaches, particularly regarding flexibility and cost-efficiency. Nonetheless, certain issues remain, most notably the fact that the operation of deploying Virtual Network Functions is still based on human interaction. Therefore, deep infrastructure knowledge is required from network managers and such a process is prone to errors and delays. However, Blockchain Signaling presents itself as a viable solution to automate Virtual Network Function (VNF) deployment thereby amending the aforementioned shortcomings. Hence, leveraging Smart Contract (SC) events, this work proposes a novel approach for automatically deploying VNF without human interaction. The system presented in this work provides users with the ability to perform VNF lifecycle operations (*e.g.,* create and delete) programmatically, with the possibility for a GUI-based approach also being present. Furthermore, the system has been designed to be extensible and generic, thus allowing for the incorporation of additional NFV frameworks. Evaluations on the system revealed that its performance and interaction cost is dependent on the selected blockchain's block time and blockchain's cryptocurrency price. Hence, they may differ based on the choice of blockchain.

Das Aufkommen von Network Function Virtualization (NFV) hat insbesondere im Bereich der Flexibilität und Kosteneffizienz zu bedeutenden Vorteilen gegenüber traditionellen Netzwerkkonzepten geführt. Dennoch gibt es diesbezüglich immer noch gewisse Schwierigkeiten, vor allem weil das Deployment von Virtual Network Functions (VNF) auf menschlicher Interaktion beruht. Dies führt dazu, dass Netzwerkmanager tiefgreifende Kenntnisse über die darunterliegende Infrastruktur benötigen. Zudem ist dieser Deploymentprozess fehlerbehaftet und anfällig für Verzögerungen. Der Einsatz von Blockchain Signaling bietet sich an, um das Deployment von VNFs zu automatisieren, was die oben genannten Probleme beseitigt. Diese Arbeit präsentiert einen neuen Ansatz zum automatisierten Deployment von VNFs (ohne menschliche Interaktion) mittels Smart Contract (SC) Events. Das in dieser Arbeit präsentierte System bietet dem Benutzer die Möglichkeit VNF Lifecycle Operationen (z.B. Erstellen und Löschen) programmatisch durchzuführen. Zusätzlich wird ein GUI-basierter Ansatz zum Ausführen dieser Operationen angeboten. Das System ist ausserdem erweiterbar und generisch, was die Anbindung weiterer NFV Frameworks erlaubt. Die Evaluation des Systems hat aufgezeigt, dass dessen Performanz und Kosten für die Interaktion massgeblich von der Blockzeit der ausgewählten Blockchain, sowie vom Kurs der zugehörigen Kryptowährung abhängig sind. Daher können Performanz und Kosten variieren, je nachdem welche Blockchain für den Betrieb des Systems ausgewählt wird.

Acknowledgments

We, the BCV team, would like to extend a big 'Thank You!' to Prof. Dr. Burkhard Stiller and the entire Communication Systems Group (CSG) at the University of Zurich, for the opportunity to work together on this project.

Additionally, we want to express our gratitude towards our team of supervisors, Eder J. Scheid and Muriel Franco, for their support throughout this endeavour. In particular, Eder J. Scheid has been very actively involved, providing us with valuable feedback, insight and resources for the entire duration of the project and even before, when formulating the initial specification of our task.

Contents

Chapter 1

Introduction

The concept of Network Functions Virtualization (NFV) proposes to host physical middleboxes in generic hardware using virtualization technologies, such as Virtual Machines (VMs) [36]. With this virtualization approach, security functions, such as firewalls, Deep Packet Inspection (DPI), and Load Balancers (LB), can be rapidly deployed in the network infrastructure as Virtual Network Functions (VNF) to mitigate attacks (*e.g.,* firewalls, and DPI) or to increase the overall network performance (*e.g.,* LB). This work is not only motivated by the emergence of NFV that allows such decoupling of Network Functions (NF) from dedicated hardware to resource virtualization [36], but also by the development of possibilities that Blockchain (BC) technology [53] started to offer (*e.g.,* DDoS mitigation [49]).

In fact, the examination of existing work showed that the VNF deployment is a manual, human-based process (*cf.* Chapter 3). Therefore, it is conceivable, that the automation of VNF deployment would result in several benefits, such as faster deployment times, event-based deployment triggers, or simply removing the human component from the equation, eliminating one potential source of errors. Such deployment automation can be addressed within the domain of BC technology. In particular, BC-based Smart Contracts (SC), which provide a platform for event-based communication, that external systems can then act upon, become apparent as a potential solution candidate. Further, by choosing a BC-based solution, other advantages are inherently given, *e.g.,* data immutability and decentralization, leading to auditability possibilities [51].

With that in mind, this work proposes a concrete system utilizing the aforementioned event-based approach to serve as an adapter between VNF Management and Orchestration (MANO) frameworks and applications that rely on VNF deployments. Hence, applications that use the solution described herein may control at what point in time which predefined VNF is to be deployed, with the responsibility of actually performing the VNF deployments being delegated to the proposed system. The entry point for both configuration of VNFs as well as for triggering deployments is a BC-based SC. Nevertheless, in addition to the automatic deployment, this proposed solution also offers the possibility to trigger VNF deployments based on user input in a web application, providing flexibility in how the solution can be used.

1.1 Description of Work

The work was developed in a number of subsequent stages. The first stage consists of a literature review to form an understanding of the topics associated with this work, such as NFVs, VNF frameworks, BCs, and BC-based SCs. This stage also includes compiling and discussing a list of related works, to identify a research gap and provide a *raison d'être* for this work.

In a second stage, requirements for the system-to-be are elicited and defined, such that a design was proposed that served as the foundation of the implementation. Furthermore, an appropriate NFV framework that matches the aforementioned design and requirements was selected.

Following this, the third phase was reserved for the synthesis of the requirements into an implementation of the separate system components, followed by their combination into a functioning system. Moreover, the different components of the system were deployed on a suitable platform, leading to a fully functional, non-local Proof-of-Concept (PoC) setup.

The final stage then serves as the evaluation of the PoC regarding a number of criteria (*e.g.,* costs and performance) that were defined based on the requirements from stage two. In addition, the findings of the project were discussed, and potential future work was explored.

1.2 Document Outline

The structure of the thesis is as follows: After the introduction, Chapter 1, the background required to get a grasp of the thesis' topic is introduced in Chapter 2. In Chapter 3 the related work is explored, discussed, and compared against our work. This is followed by the main chapter of the report, Chapter 4, where the design and the implementation of the system are laid out, alongside several use cases. The evaluations are presented in Chapter 5, finally followed by the conclusion and future work in Chapter 6.

Chapter 2

Background

The first part of this chapter covers the fundamentals of NFV. A comparison is made between NFV and the traditional way of implementing and deploying NFs as well as the benefits and challenges that arise from using NFV. Furthermore, the European Telecommunications Standards Institute (ETSI) NFV reference architecture (MANO) together with selected NFV frameworks are presented. The second part of this chapter is dedicated to the basic concepts of BC technology, with a special focus on the use of SCs.

2.1 Network Function Virtualization

NFs represent functional building blocks of the network infrastructure [36]. Conventionally, network operators used specialized proprietary networking hardware to provide NFs such as firewalls or LBs [11, 36]. However, this hardware-centric approach has major drawbacks: The specialized hardware is very expensive, both to procure and operate, leading to high capital and operational expenses [11, 36]. In particular, any change in the order of a chain of NFs implies changes of the underlying network topology, which reduces flexibility [36].

The idea behind NFV is to decouple software from hardware [36] by utilizing virtualization technology on top of high-volume industry hardware [11] to provide NFs. Thus, when it comes to deploying and operating NFs, NFV offers more flexibility, but also agility as well as the potential for automation and scalability compared to the traditional approach [36]. Essentially, changes in NFs take place in software instead of in hardware, which enables quicker changes, as no changes to the physical infrastructure are necessary [36]. The use of high-volume industry hardware makes it possible to make use of economies of scale, reducing capital expenses [11].

2.1.1 Virtual Network Function

While conventional NFs are implemented on specialized vendor-specific hardware, VNFs are NFs that run on VMs (*i.e.,* in software [5]) instead [36]. VNFs can be chained together

in a particular order to form Service Function Chains (SFC) [5]. While it is possible to build SFCs with hardware-based NFs, this will result in static chains that are hard to maintain. With VNFs, however, the order of NFs can be changed dynamically within an SFC, leading to more flexibility and reduced operational expenses [5].

The authors of [47] show an example of an SFC that consists of multiple VNFs: The SFC, in this case, is used to route all incoming TCP traffic on a specific port through the SFC before forwarding it to the destination. The SFC consists of three VNFs that are chained in the following order: The traffic is first processed by a firewall, which forwards the traffic to an intrusion detection system, which again forwards the traffic to an LB. SFCs benefit greatly from Software Defined Networking (SDN), as SDN allows to dynamically allocate VNFs [34], increasing the flexibility of SFC composition.

2.1.2 Management and Orchestration (MANO)

Although NFV and SDN enable virtual computing and reduce network commissioning and operating costs, a need for orchestration arises [32]. Therefore, to further leverage the virtualized environment, the ETSI Industry Specification Group (ISG) proposed a specification and architectural framework, labeled ETSI NFV MANO, that is intended to serve as a framework reference and does not include a concrete implementation [32]. Specific implementations that follow the ETSI NFV MANO reference are illustrated in Section 2.1.3. The NFV MANO has various responsibilities: It handles all VNF lifecycle virtualization specific concerns, takes care of faults that may occur in VNFs, stores the VNF's state information, and allows for communication between various VNFs [4]. The MANO framework, as shown in Figure 2.1, features three functional blocks [4, 6] and is composed of the following components:

- **Virtualized Infrastructure Manager (VIM):** is responsible for the NFV Infrastructure (NFVI), and monitors and handles all resources available to the NFVI [4, 6], including the compute, storage, and networking resources [32, 31].
- **NFV Orchestrator (NFVO):** The NFVO has two important responsibilities: It manages global NFVI resources across various VIMs and manages the Network Services (NS) [18, 52, 41]. For that reason, it includes controller and policy components to manage the behavior of the VNFs [31] and handles the NFVI's requests [52, 41]. It is also known as the NFV Network Services Orchestrator [32].
- **VNF Manager (VNFM):** The VNFM is connected to the NFVO and VIM, and one of its main purposes, is to achieve high interoperability by standardizing the VNFs [31]. Therefore, the VNFM is responsible for the lifecycle management of VNF instances [18, 32], more precisely, it controls all VNFs in a system, is constantly monitoring them, and can scale, update and remove them as needed [3, 6].

Additional to the functional blocks, the NFV MANO includes four repositories that are used to store component's metadata [4, 32]:

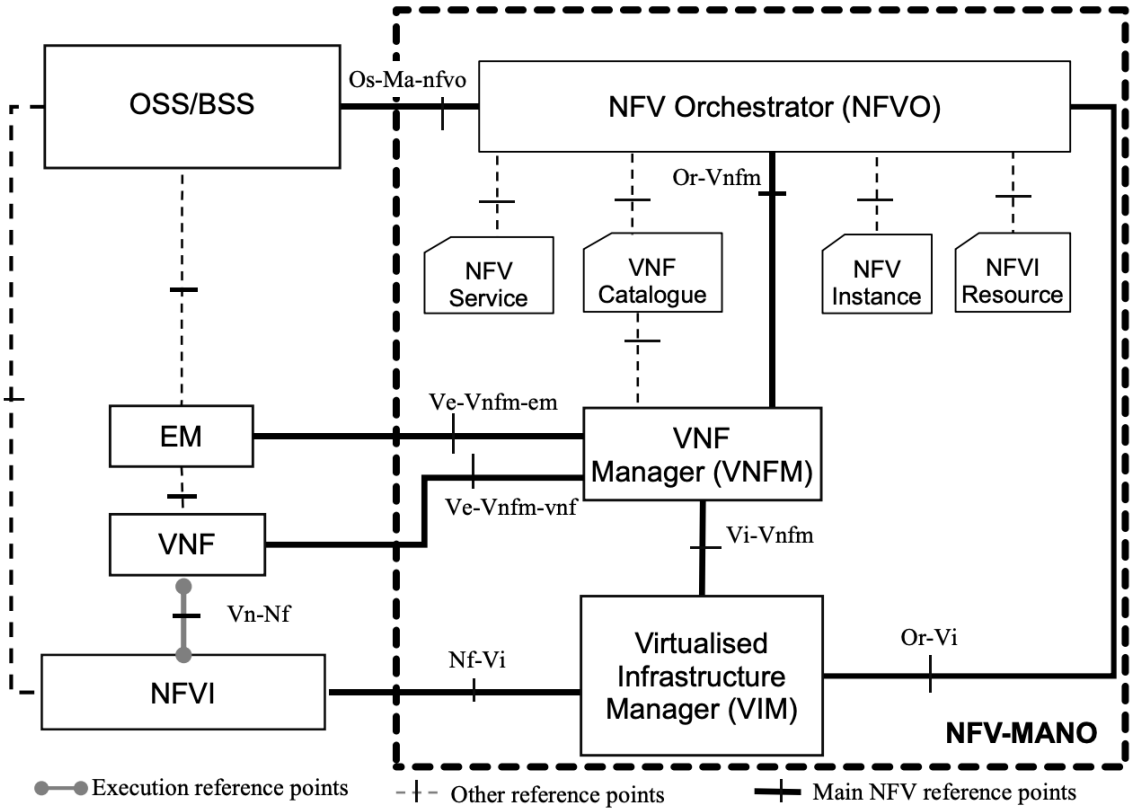

Figure 2.1: NFV MANO Reference Architecture [18]

- **NFV Service:** This repository includes templates that define a service's lifecycle, *i.e.,* on-boarding, creation, and termination [32].
- **VNF Catalogue:** The catalog is managed by the VNFM [4], and includes templates describing the VNF's attributes [32].
- **NFV Instance**: This repository stores the VNF's and NS's data [32].
- **NFVI Resource**: Stores NFVI's resources [32], which are orchestrated by the NFVO [18].

2.1.3 Frameworks

While the ETSI MANO gives an architectural representation of an NFV Framework [6], various implementations follow its architectural approach and use it as a cornerstone.

- **FENDE** [6] is described as "[. . .] the first NFV ecosystem that provides a marketplace for VNF offering together with VNF and SFC creation and life cycle management, as well as the infrastructure support needed for VNF and SFC instantiation". FENDE considers three different kinds of users [6]:

- **Developers** implement VNFs and submit them for review to the FENDE marketplace.
- **Reviewers** perform reviews on the VNFs submitted by the developers. They inspect the VNFs thoroughly and make a verdict for the publication of the VNF, which is either approval or denial. After approval, VNFs are published to the marketplace.
- **Customers** can browse the catalog of VNFs and acquire particular VNFs. FENDE allows instantiating VNFs on public (*i.e.,* cloud infrastructure such as Microsoft Azure or Amazon EC2) or private infrastructure according to the customer's needs. Customers can also create VNF chains to form SFCs that serve their purposes.

From an architectural standpoint, FENDE follows the ETSI NFV MANO reference architecture. Figure 2.2 shows the architecture of FENDE. Note that the structure of the NFV layer matches with the reference architecture shown in figure 2.1.

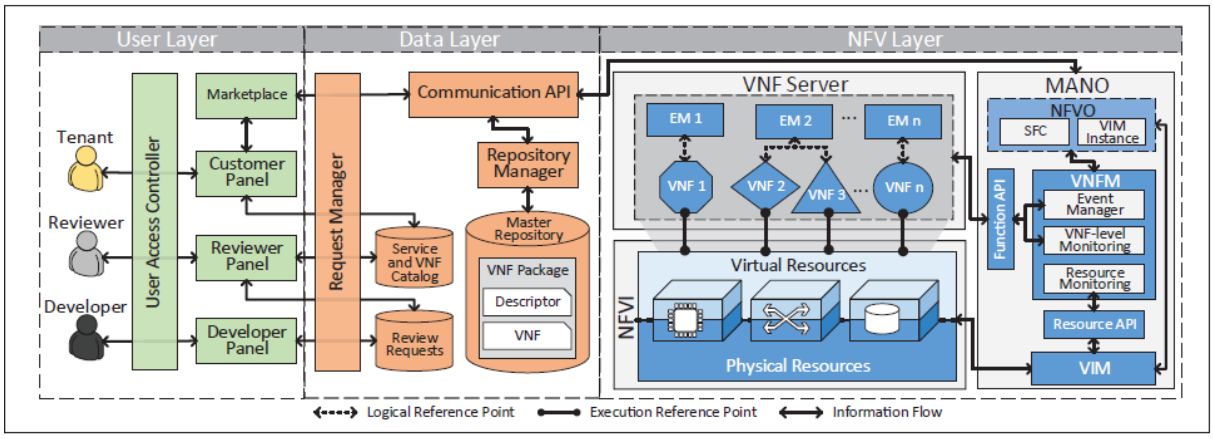

Figure 2.2: FENDE Architecture [6]

- **OpenStack Tacker** had been announced in 2015 and is part of the open-source OpenStack Cloud Computing Platform, which predates the MANO framework, and includes a variety of interrelated components [59, 10]. It intends to adopt the ETSI MANO architectural framework [10] and is described as a "generic VNFM and NFVO to operate NS and VNFs on an NFVI" [41]. The OpenStack Tacker architecture is depicted in Figure 2.3. In terms of VIM, OpenStack Tacker offers the choice between using an OpenStack or a Kubernetes cluster [41]. Tacker also makes use of the descriptive language 'TOSCA' to define VNF's meta-data definitions [42, 10].

2.1.4 Challenges of using NFV

Apart from all the benefits, implementing NFV also involves challenges [11, 36], some of which are highlighted in this section:

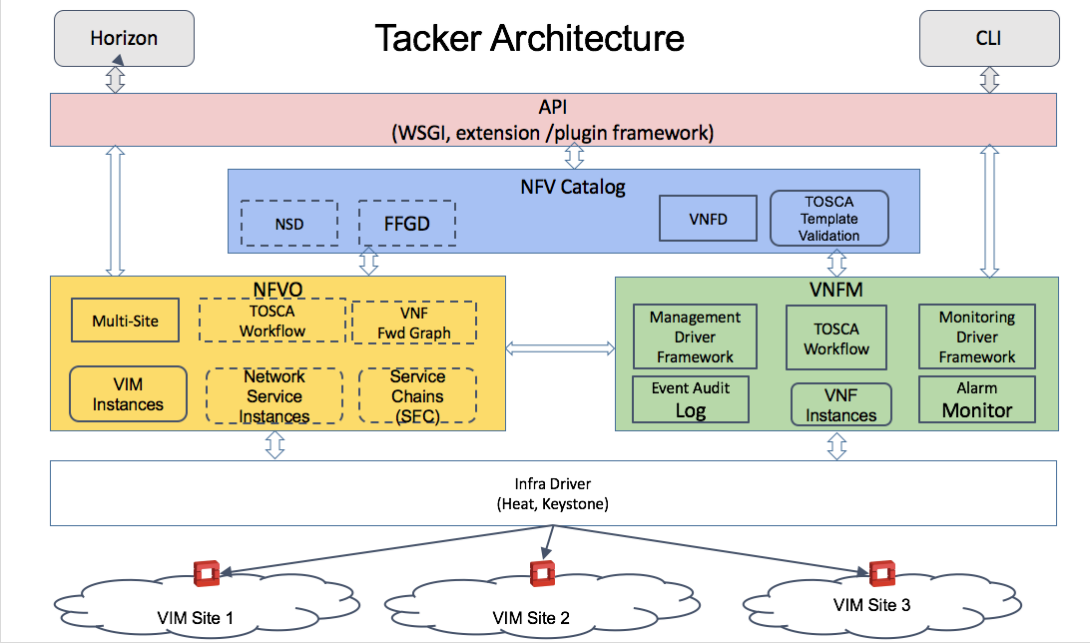

Figure 2.3: OpenStack Tacker Architecture [42]

- **Performance** is a large concern in computer networks. In conventional networks, specialized hardware usually performs very well. However, switching from specialized hardware to general-purpose hardware with virtualization can lead to performance degradation [11]. For the performance loss to be as small as possible, it is critical to identify bottlenecks when defining an NFV architecture [36].

- The authors of [11] argue that NFV should be able to improve network **security** because of its dynamic nature, allowing quick reactions in case of an incident, leading to higher **resilience**. According to [36], NFV services must have the following two properties: Firstly, they must isolate the NFV services of different customers, such that breaches do not cascade from one customer to another. Secondly, providers must ensure that the virtualization infrastructure is not exposed beyond the scope of NFV services.

- **Scalability** of NFV solutions is of utmost importance. In order to build scalable NFV systems, a high degree of **automation** is necessary [36].

- NFV must provide support for **legacy** NFs, as these systems are unlikely to disappear right away [36]. Providers must be able to run NFV in coexistence with their legacy software, to gradually migrate physical NFs to VNFs [11].

While NFV on its own is a promising concept, its benefits are amplified when combined with SDN and cloud computing [32]. NFV can be conveniently implemented on top of cloud infrastructure, as cloud computing provides the virtual network infrastructure needed for this purpose [36]. However, the challenge of using cloud computing to implement NFV is performance: Cloud providers are expected to provide (telephone) network carrier-grade performance. Otherwise, implementing NFV on cloud infrastructure might be infeasible except for PoC scenarios [36]. In other words, cloud computing is an enabler

for NFV [11]. NFV and SDN complement each other very well [11]: Using SDN to implement NFV can lead to improved performance as the control plane is separated from the data plane, thereby positively affecting network operations, *e.g.,* simplification and faster innovation. Another scenario is mentioned in [36]: SDN controller can be run as a VNF inside an SFC.

Figure 2.4 summarizes the commonalities and differences of NFV, SDN, and cloud computing. One essential point to notice is that each of these three concepts provides some form of abstraction: NFV provides an abstraction over functions, SDN provides an abstraction over networking, and cloud computing yields an abstraction over computation [36].

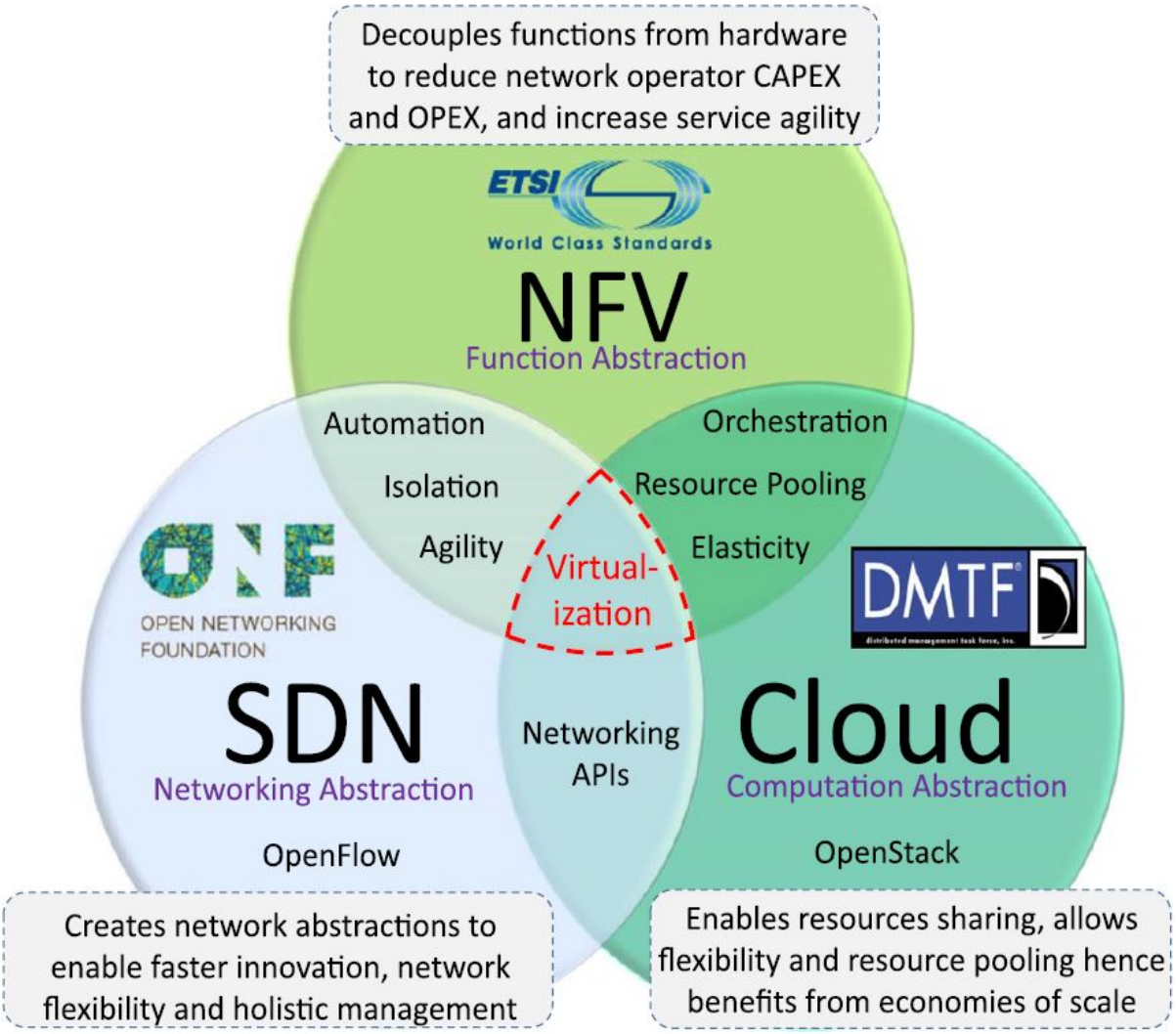

Figure 2.4: The commonalities and differences of NFV, Cloud Computing, and SDN [36]

2.2 Blockchain

The concept of BC was first introduced in 2008 when Bitcoin entered the market for electronic cash solutions [53]. BC, the technology Bitcoin relied on, was what set it apart from other electronic cash solutions, as it enabled Bitcoin to function in a practical setting, solving the issue of 'double-spending', which refers to a given coin being spent twice, without requiring any central authority or Trusted Third Party (TTP) to verify transactions [53].

The rest of this chapter will detail different relevant aspects of BC technology in terms of the work done in this project. Section 2.2.1 will serve as a high-level introduction to the technical concepts and the consequently emerging properties of BCs. Following this, Section 2.2.2 highlights how different BC implementations may vary from each other on a technical level, depending on what kind of purpose or application a given implementation is designed for. Additionally, the implications of those variations will be discussed. Lastly, Section 2.2.3 offers a deep-dive into SCs, providing a general overview over SCs with a particular focus on BC-based SCs.

2.2.1 Technical Introduction to BC Technology

A BC, at its core, is a distributed ledger, a data structure which is stored in a distributed fashion, *i.e.,* on multiple devices, that fulfills specific criteria regarding its implementation [33].

Structurally, a BC is organized as a chain of blocks, each containing multiple transactions that have occurred on the BC [33, 64]. This BC data structure furthermore is append-only, meaning new transactions can only be added as a new block to the tail-end of the chain [64]. Appending a new block means creating a cryptographical link to the previous block by including its hash value [33].

This leads to BCs being 'tamper-resistant', with that resistance increasing for a given block as additional blocks are appended. Thus, the content of a block that is sufficiently old is exceedingly difficult to change [64]. Should such an attempt at modifying a block nonetheless be successful, the change will be noticed, due to BCs being 'tamper-evident' [64].

Note that the property of 'tamper-resistance' is referred to by other authors as 'immutability', as in *e.g.,* [53, 33]. For the sake of consistent terminology, for the rest of this report, the term 'immutability' shall be used, as its meaning, in our opinion, is easier to grasp.

In terms of networking, the devices which participate in the BC, called 'BC Nodes', are organized as a peer-to-peer (P2P) network with, consequently, no governing centralized authority [33]. Thus, it requires the state of the ledger maintained on the participating devices to be coordinated, such that a global 'consensus' is reached [33]. To achieve said consensus, so-called 'PoX' or 'Proof-of-X' mechanisms, such as 'Proof-of-Work (PoW)' (utilized *e.g.,* in Bitcoin [64]) are used [33].

2.2.2 BC Diversity

Different BC implementations may vary from each other in several ways, ranging from the consensus mechanism and the amount of throughput, in terms of transactions per second, to even the underlying data structure [53]. The BC Platform IOTA, for example, uses a Directed Acyclic Graph (DAG) instead of the chain of blocks usually associated with BC [53].

Another aspect in which BC implementations may differ is their 'deployment type' [53]. The deployment type is a combination of two axes of permissions, with 'public' and 'private' concerning the read permissions and 'permissionless' and 'permissioned' representing write permissions [53]. Figure 2.5 presents an overview over the four resulting deployment types.

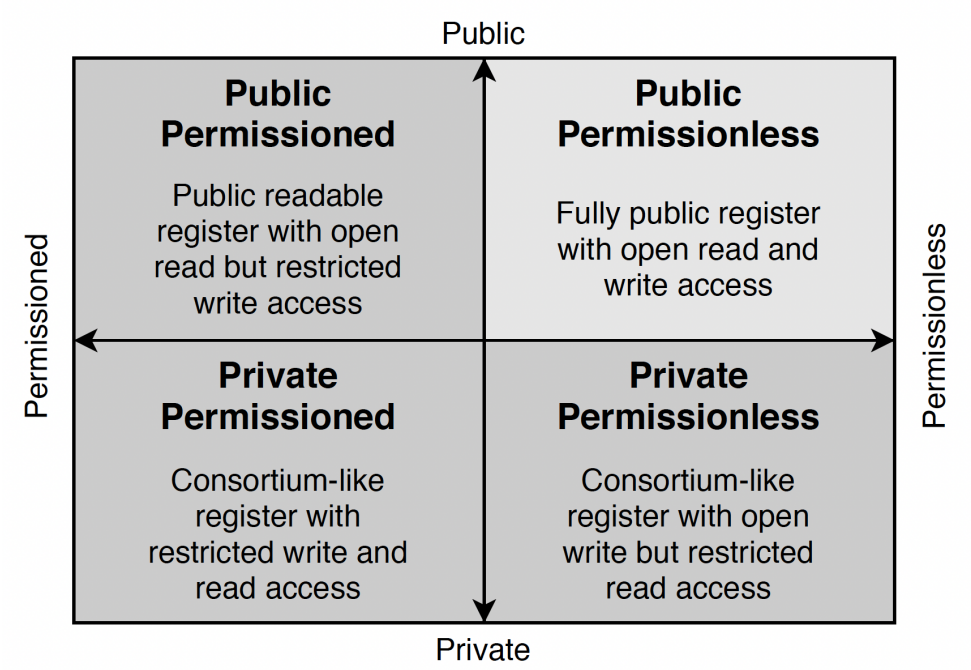

Figure 2.5: BC Deployment Types [53]

Note that according to [53] only implementations which are both public and permissionless are to be categorized as BCs, from the perspective of analyzing a 'distributed application'. Thus, there are two sets of lenses through which categorization may be performed, one which focuses only on the underlying data type, in which all of the four deployment types are seen as BCs, and the other, which as previously discussed, only recognizes public permissionless implementations as BCs [53].

The takeaway here should be that there is indeed diversity when it comes to BCs. [33] discuss in depth that said diversity has implications on what kind of BC is the most fitting given the use-case at hand. Furthermore, SCs are not supported by all BCs to the same degree [53, 33].

Ethereum [8], *e.g.,* allows for the writing of Turing-complete SCs in languages such as *e.g.,* Solidity, which then are executed on the Ethereum Virtual Machine (EVM) [53, 33].

In contrast, the SCs supported by Bitcoin are simple scripts [53], which are run natively, as they are an integral part of the Bitcoin protocol [33].

2.2.3 Smart Contracts

The concept of an SC is not exclusive to the context of BCs, as it has been introduced before the invention of the BC [53]. Fundamentally, the idea of an SC is to serve as a computerized program that enforces the terms of a contract between parties in an automated fashion, thus removing the need for intermediaries in the form of TTPs [33].

The properties offered by the BCs, such as the decentralized nature and the immutability of the stored data, make BC a very suitable medium for SCs [53]. BC-based SCs (note, from here on out, the term SC shall refer to BC-based SCs unless stated otherwise) differ in their implementation based on the BC platform in question as mentioned previously [53].

The rest of this section will be dedicated to exploring the differences between Turing-complete and non-Turing-complete SCs and afterward provide more details specifically regarding SCs on Ethereum [8], as most BC platforms implement the same model [53] and our work will utilize Ethereum SCs. Lastly, some insights into Solidity, the SC programming language of choice for our project, will be provided.

Turing-Complete vs. Non-Turing-Complete SCs

The term 'Turing-complete' (TC) is used in computer science to refer to the expressive power of a given programming language [35]. A programming language is labeled as TC if, by the application of a translation schema, it can be shown to have the capability to "express the same computations as a Turing machine" [35].

To put it more succinctly, TC programming languages can perform "unbounded computations over unbounded values" [35]. In practice, this means that a TC programming language may *e.g.,* perform loops [53].

Non-TC programming languages are not as powerful in that respect and are thus less expressive than their TC counterparts [35]. It is worth noting that all TC programming languages are seen as equally expressive [35].

Returning to the topic of SCs, both platforms that provide TC and non-TC SC capabilities exist, with an example for the former being Ethereum and one for the latter being Bitcoin [53]. Given what was stated above, it is thus unsurprising that SCs written in a TC programming language (such as *e.g.,* Solidity) have a greater range of computations they can perform when compared to their non-TC counterparts [50]. It would be an oversight, however, to present TC SCs as strictly superior, as with their higher expressiveness there also come increased security risks [50].

Ethereum SCs

Since our work will be based on Ethereum SC capabilities, it is essential to establish how SCs operate on Ethereum beforehand.

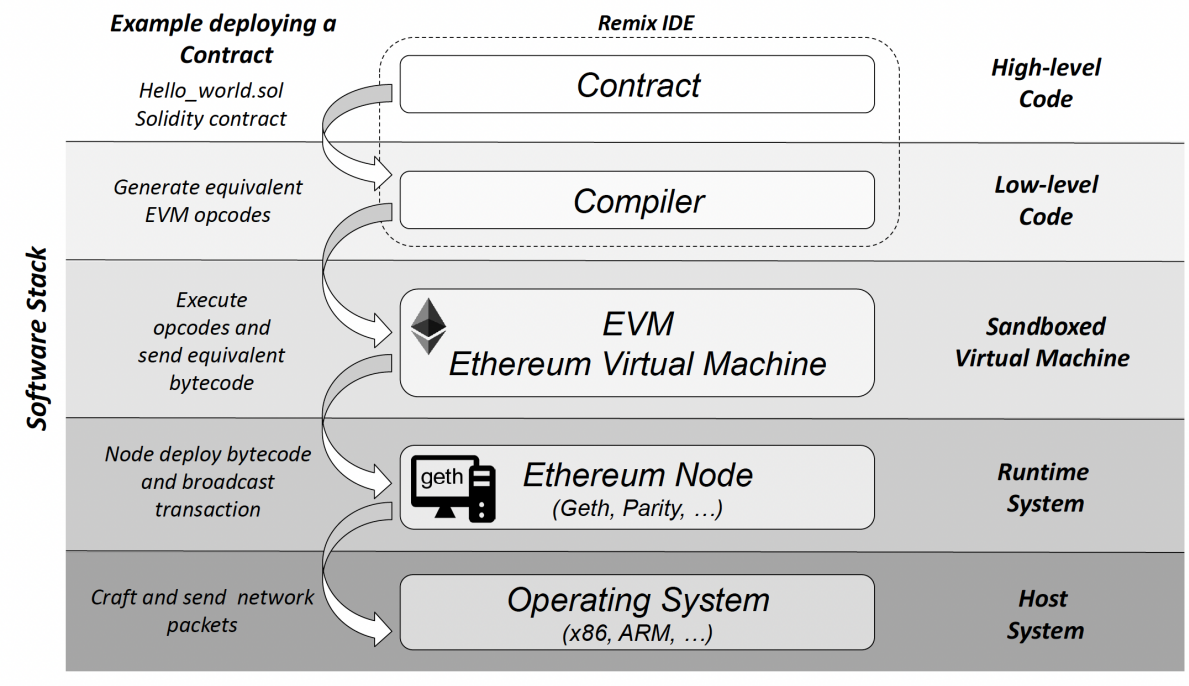

Figure 2.6: Ethereum SC Deployment Software Stack [53]

Figure 2.6 depicts the software stack used for deploying and running SCs on Ethereum. The contract itself is implemented in a high-level language such as *e.g.,* Solidity, which is fully TC [53, 33]. From that, so-called 'EVM opcode' is generated, which is then executed on the EVM [53].

Each Ethereum Node must host an EVM instance [33]. The EVM is of vital importance, as it ensures that ultimately the execution environment is identical at each node of the system [53]. The EVM, as the name suggests, is a virtual machine, hence is isolated from the node it runs on, but with the capability to interact with the host Operating System (OS) [53].

One essential part of Ethereum SCs, is the concept of 'gas' [53]. Since SCs are executed on the nodes running EVMs, Ethereum needs to put some system in place to prevent the BC from effectively being DDoS-ed by *e.g.,* deploying an SC that performs an infinite loop [53]. That system, called an 'incentive scheme' by [53] is the aforementioned gas. In simple terms, SCs which are more complex incur higher gas costs to account for the fact that they are more costly to both deploy and operate [53].

Solidity

The official documentation describes Solidity as an "object-oriented, high-level language for implementing smart contracts" [28]. It was inspired by programming languages such

as C++, Python, and JavaScript and is specifically built to run on the EVM [28].

Solidity also is categorized as TC [53, 33] and is still being actively developed judging by the version number and number of pull requests on their Github repository [27].

One feature of Solidity that will be of particular importance to this project is called 'Events'. Events provide the ability to write data to EVM's logging system, leading to the data being persistent on the BC [28]. Applications outside the BC can subscribe to these events through the use of an Ethereum client [28]. Notably, the events are associated with the address of the SC that emitted them [28], which essentially allows external applications to react to what happens on a given SC.

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity >=0.4.21 <0.9.0;
3
4  contract ClientReceipt {
5      event Deposit(
6          address indexed _from,
7          bytes32 indexed _id,
8          uint _value
9      );
10
11     function deposit(bytes32 _id) public payable {
12         // Events are emitted using `emit`, followed by
13         // the name of the event and the arguments
14         // (if any) in parentheses. Any such invocation
15         // (even deeply nested) can be detected from
16         // the JavaScript API by filtering for `Deposit`.
17         emit Deposit(msg.sender, _id, msg.value);
18     }
19 }
```

Listing 2.1: Solidity Event Emission Example [28]

Listing 2.1 shows an example of how events are declared and emitted. The example is taken from the official Solidity documentation page about events [28]. One thing worth noting is the *indexed* keyword as used in lines 6 and 7 of the Listing. If a parameter is marked as *indexed*, it is added to a part of the logging output called 'topics', with the main benefit of having parameters in the structure of the topic being that they can be used as filtering criteria when searching for events [28].

Chapter 3

Related Work

This chapter presents related projects in the areas of NFV and BC. Starting with a section on the combination of BC and NFV, a variety of solutions are presented. Furthermore, the described solutions are compared and discussed in detail as well as distinguished from this work.

3.1 BUNKER

The idea of BUNKER [51] is to provide a trusted immutable VNF package repository without a centralized TTP. It addresses the problem of assuring that VNFs in the package repository have not been manipulated. It does so by leveraging SCs on top of the Ethereum BC, as the latter provides immutability and decentralization, eliminating the necessity of relying on a centralized TTP [51].

BUNKER considers two kinds of actors [51]:

- **Developers** implement VNF packages and publish them to the repository. They also collect the licensing fees of their packages.
- **Users** buy VNF packages from the repository and install them to their infrastructure after verifying the VNF package integrity.

Figure 3.1 shows the architecture of BUNKER. In contrast to FENDE, BUNKER does not implement the NFV MANO components but allows the use of third-party solutions [51]. BUNKER leverages the BC for two purposes: Licensing and verification. The former involves the purchase of licenses for the usage of VNF packages, where the customer performs a transaction on the BC to obtain a license. The latter involves the package verification process when downloading a VNF package. In particular, the hash of the downloaded package is computed and then compared to the hash stored in the BC. If they match, the customer can be sure that the package had not been tampered with [51]. The verification process utilizes the immutability property of the BC, which ensures that the original hash value cannot be altered after creation. When a license is acquired, an SC event will be emitted, which causes subscribing applications to react accordingly [51].

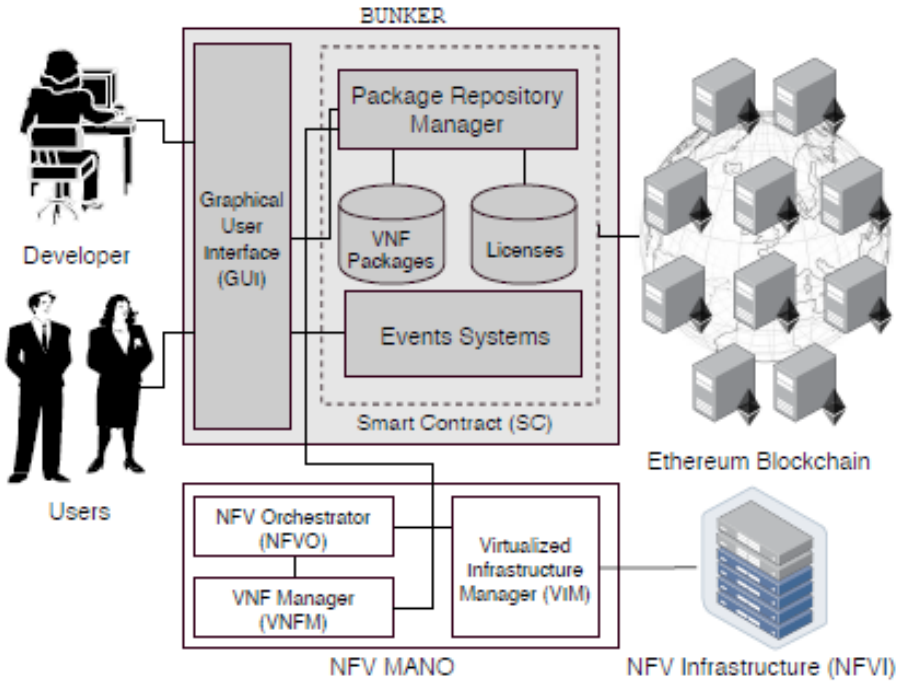

Figure 3.1: BUNKER Architecture [51]

3.2 BRAIN

BRAIN is described by its authors as a 'blockchain-based reverse auction' for infrastructure supply in Virtual Network Functions-as-a-Service (VNFaaS) [22]. BRAIN relies on the fact that the introduction of NFV simultaneously leads to the creation of market opportunities for competing VNF solutions [22]. Such a market allows infrastructure providers to compete for customers to host VNFs on the providers' respective NFV-enabled infrastructure [22]. This competition is where BRAIN comes into play, as it presents providers of NFV-enabled infrastructure with a way to compete for customers in a less 'static' fashion [22].

The authors elaborate that traditionally, providers would pursue a strategy of marketing their service openly, meaning the conditions and prices of the services are observable by all potential customers, who then choose the service that best fits their use case [22]. The drawback of such a system is that ultimately the offers are not customized to the customers' needs leading to the competition between the providers to be, as the authors call it, 'rather static' [22].

In contrast, the mechanism proposed by BRAIN implements a reverse auction, meaning the sellers (in this case the providers) compete for buyers by, upon receiving a request for VNF hosting, submitting a sealed bid to the prospective customer that ends up being tailored much more closely to both the needs of the customer as well as the expenses of the providers [22]. This is due to the provider being able to estimate on a per-customer basis what the costs of providing their services would amount to [22]. From the customer perspective, the fact that the mechanism, after all, follows sealed bidding, leads to overall

lower prices, as the providers are not aware of the offers made by the competitors, hence are incentivized to present offers at competitive prices [22].

The auction mechanism used in BRAIN is reliant on BC-based SCs and the immutability provided by BC technology to facilitate permanent records of a given auction, including the customer's requirements regarding VNF hosting and the full history of bids, allowing for bids to be audited [22].

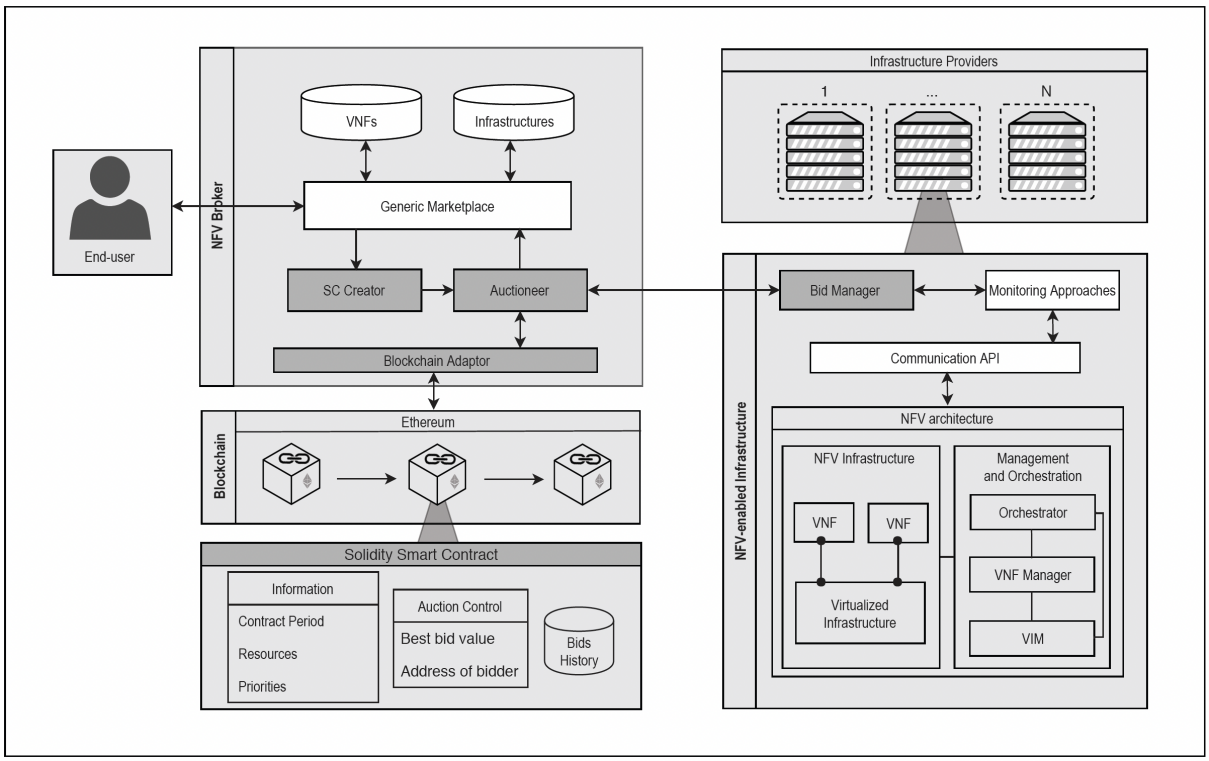

Figure 3.2: BRAIN Architecture [22]

Figure 3.2 depicts the architecture of BRAIN's implementation. Discussing the architecture in its entirety would exceed the scope of this section. Instead, the following paragraphs provide an overview of the process of holding an auction, including the components involved in each particular stage.

The process starts with an end-user browsing a marketplace and acquiring a VNF [22]. The specifications (priorities of the end-user and requirements imposed by the purchased VNF) are submitted to the so-called 'SC Creator' who creates an SC based on them [22]. Once done, the SC is deployed on the BC by the 'Auctioneer', who also notifies the providers that an auction has been opened [22].

This notification is sent via the 'Bid Manager', who interfaces with the infrastructure providers [22]. The bid manager furthermore implements an automated bidding mechanism based on configurations provided by the providers as well as the current status of the infrastructure [22].

Once the auctioneer terminates the bidding process, the bids are evaluated and the best bid is communicated to the end-user at the marketplace [22]. The process then concludes with the initialization of the VNF deployment [22].

3.3 B-VNF

B-VNF, short for "Blockchain-enhanced Architecture for VNF Orchestration in MEC-5G Networks", is a project focused on utilizing BC technology to provide a secure way of performing VNF orchestration for Multi-Access Edge Computing (MEC) enabled 5G networks [37].

The purpose of MEC, according to [37], is to provide computational capabilities as well as storage facilities at the edge of the given network. For MEC to perform optimally, the network must be capable of migrating VNFs from cloud servers to the network edges when they are required there [37].

Migrating VNFs poses several security issues: For instance, VNF requests between the MEC node and the cloud may be subject to manipulation, or a VNF might be tampered with during migration [37]. To combat such issues, B-VNF employs the capabilities of BC technology by treating BC as an overlay P2P network built on top of the MEC-5G network [37]. As a consequence, B-VNF can perform operations that are prone to security issues, such as the said VNF requests and VNF migration, through BC transactions, which then enhances their security [37].

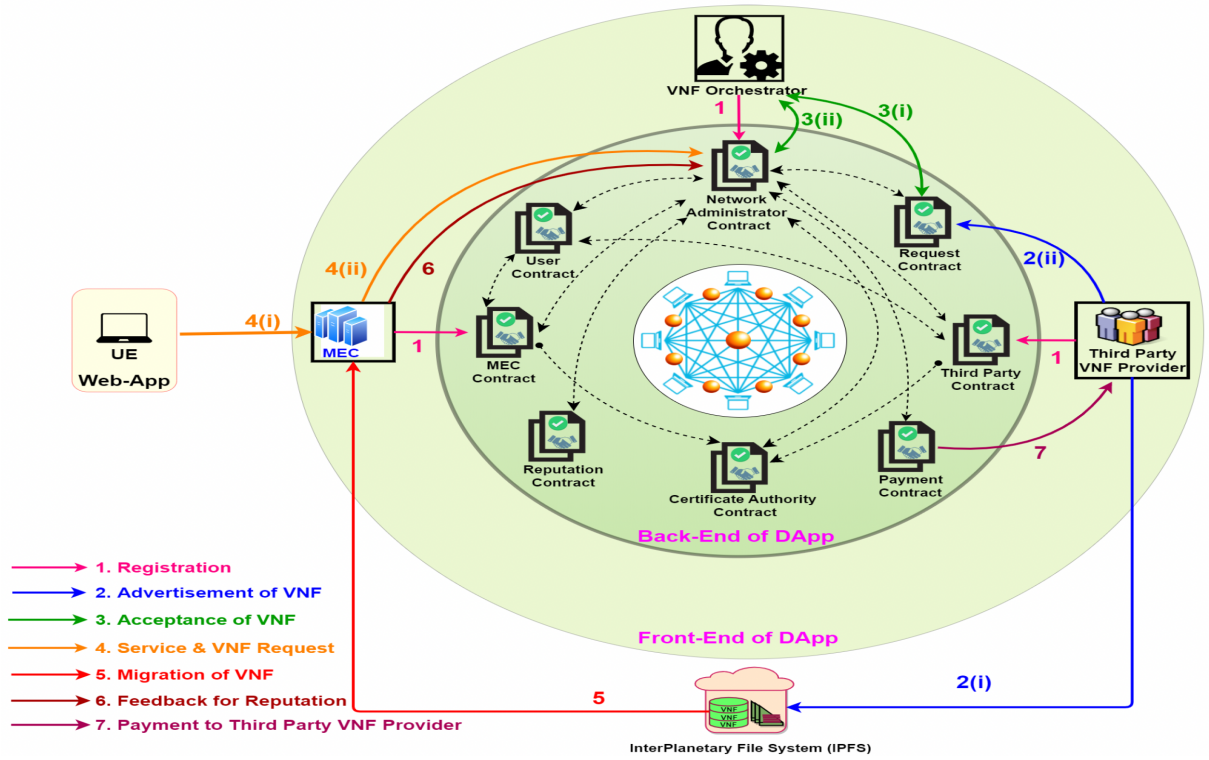

Figure 3.3: B-VNF Architecture [37]

In Figure 3.3 the main components and types of interactions are shown. The following paragraphs will highlight the main insights into this graph. For more detailed information, refer to the original article by [37].

The 'VNF Orchestrator' is the component that is responsible for coordinating the different steps involved in the process [37]. Its tasks include accepting or rejecting VNF

advertisements by third-party VNF providers, validating VNF requests from MEC nodes, evaluating which VNF is most suitable for the given request by taking into account the reputation system, as well as performing the VNF migration. Furthermore, it is responsible for facilitating secure payments to third-party VNF providers, as well as maintaining the reputation system [37].

Third-Party VNF Providers advertise their product by uploading it to the InterPlanetary File System (IPFS), from where they receive the hash which uniquely identifies the given VNF [37]. This hash then is used in a 'Request Contract', submitted to the VNF orchestrator, detailing information about the VNF in question [37].

From a User Equipment (UE) perspective, B-VNF provides the ability to issue a 'Service Request' to a given MEC node [37]. The MEC node then determines whether the VNF requested by the service request has already been deployed. If so, the request is abandoned. If not, the MEC node submits a 'VNF Request' to the VNF orchestrator, which then leads to the migration of the requested VNF to the MEC node [37].

3.4 Xevgenis et al. (2020)

The authors of [63] propose a 'distributed broker mechanism' that aims to allow Network Providers (NPs) to dynamically allocate computational and networking resources across their administrative boundaries. With this mechanism then, the scope in which resources can be traded is increased from within the network of a given NP to the entire collection of networks that belong to the participating NPs [63].

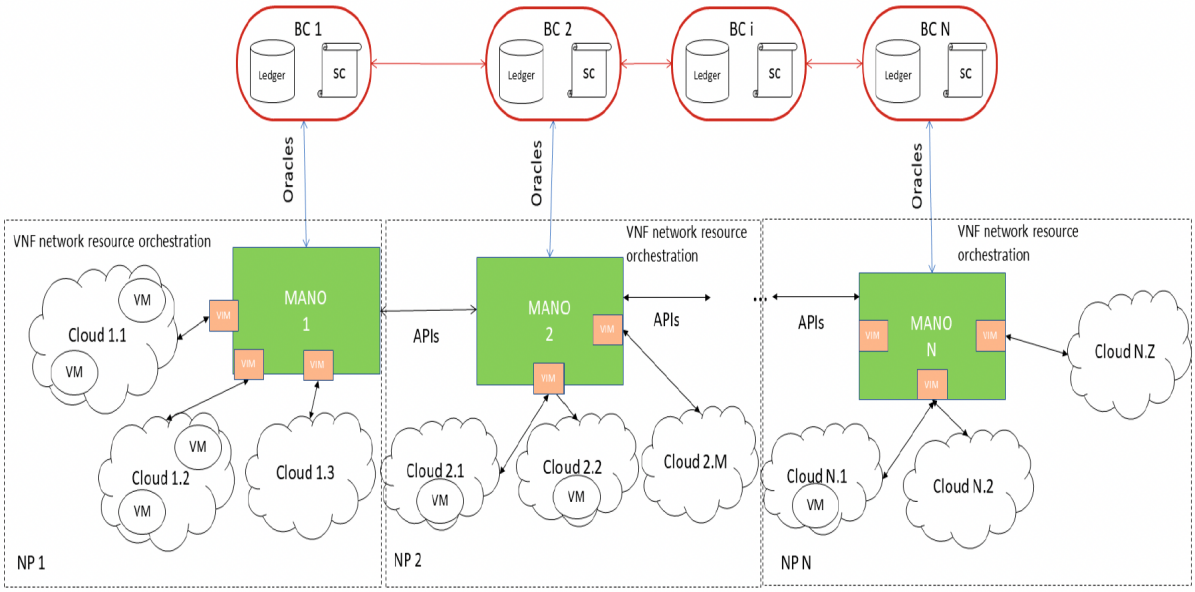

Figure 3.4: B-VNF Architecture [37]

In Figure 3.4, the system architecture can be observed. The architecture laid out by [63]

assumes that each participating NP runs a 'MANO' instance responsible for the orchestration of resources within the NPs network.

The trading mechanism which enables resource sharing between the different NPs networks is based on BC technology and requires the participating NPs to each operate a BC node [63]. If a given network now requires additional resources, its MANO component triggers the corresponding BC node via the use of oracles, which leads to the logic hosted in the SC being executed [63].

The SC essentially searches for the best match in terms of the required resources by comparing the resources offered by the other participating NPs [63]. Once that best match is found, transactions on the BC facilitate the payment for the lent resources [63].

The authors put particular emphasis on the evaluation of the system in terms of feasibility concerning the BC-based coordination mechanism [63]. Their BC of choice is a custom Quorum network, as it has the capability of performing private transactions [63]. The evaluation ultimately yields positive results ranging from, in the authors' words, 'adequate' regarding throughput to 'more than affordable' in terms of costs [63].

3.5 BSec-NFVO

As NFV is often based on public cloud infrastructure, trust issues can arise in the life cycle of VNFs and SFCs built on top of these infrastructures. In particular, these multi-tenant and multi-domain environments with shared cloud infrastructures enhance the offers of attacks, with potentially huge repercussions due to potentially large user bases [48]. BSec-NFVO is a BC designed to overcome these trust issues by logging all orchestration instructions in an immutable manner. Therefore, BSec-NFVO implemented its own version of a Practical Byzantine Fault Tolerance (PBFT) algorithm, as this allowed to rapidly reach a consensus while promising robustness up to $\frac{1}{3}$ malicious nodes [48].

The architecture of BSec-NFVO is depicted in Figure 3.5. Users can issue orchestration instructions in the visualization module. These instructions are then signed and stored onto the BC, which ensures auditability of user actions. The instructions are then forwarded to the orchestration module, which processes them and protocols the execution of the instructions in the BC. The BC module provides an interface for both visualization and orchestration modules, and it also contains a copy of the BC itself [48].

As a result, all changes to the NFV infrastructure are immutably recorded in the BC. Both the user issuing an instruction and the orchestrator executing the instruction are logged, thereby establishing trust between the involved parties [48].

The authors note that a prototype has successfully been deployed, with the finding showing that the additional BC and transaction validation did not greatly affect the system's throughput, and the system stayed stable when increasing the number of participants or when increasing the instruction message sizes [48].

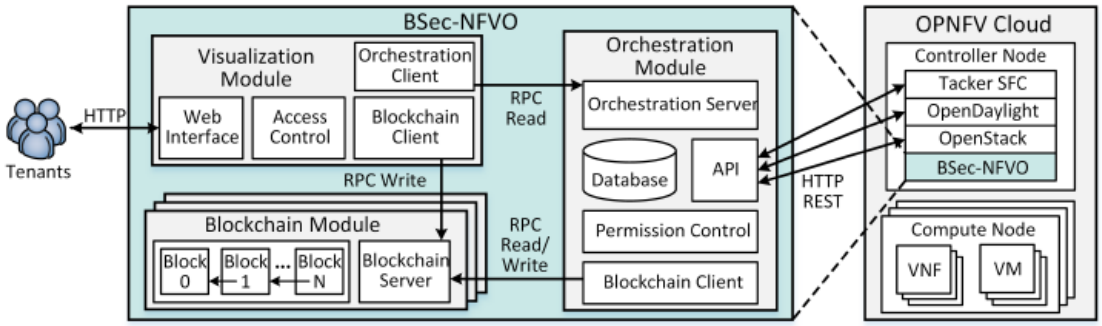

Figure 3.5: BSec-NFVO Architecture [48]

3.6 VMOA

With virtualization being the underlying concept of NFV, the security of VMs is of utmost importance. In [7], the authors present VMOA, which is an authentication mechanism for the secure orchestration of VMs. While the original version of VMOA relied on a centralized database, there is also a decentralized version of VMOA leveraging BC technology to authenticate orchestration commands.

VMOA is defined in terms of the following actors and components [7]:

- **Orchestrators** manage virtualization servers by issuing management commands towards them.
- The **Virtual Machine Manager (VMM)** is an interface component that lives inside the virtualization server. It represents the interface for communication with the orchestrator.
- **Virtualization servers** are physical machines running virtualization software. They are responsible for hosting the VMs.
- **VMOA BC** is a distributed ledger that is responsible for authenticating orchestration commands. The authors of [7] plan a PoC based on Hyperledger Fabric, as it fits the needs of VMOA.

Figure 3.6 shows an overview of the VMOA architecture. Orchestrators send a request to VMOA in the shape of a transaction on the BC. The orchestrator then sends an orchestration command to the VMM inside the virtualization server, which authenticates the command using the VMOA BC. The authentication is successful if a BC transaction for the respective command is found. In this case, the orchestration command is executed. If the authentication fails, the command is not executed [7].

3.7 BloSS

In [49] BloSS was used in a defense system against DDoS attacks since centralized defense systems are lacking hardware or software capabilities to mitigate large-scale DDoS

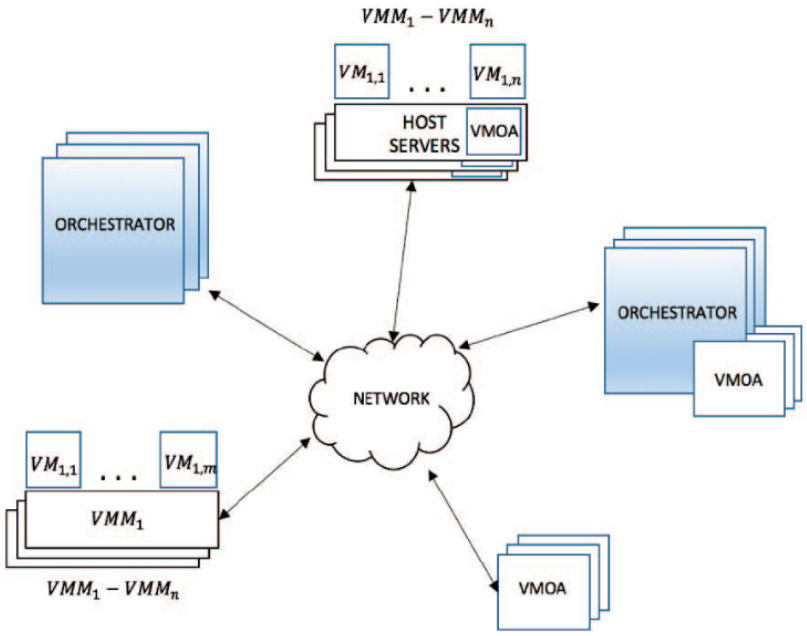

Figure 3.6: VMOA Architecture [7]

attacks. Using a consortium-based BC and SCs, this approach allows the signaling of DDoS attack information that *(a)* is reduced in complexity and *(b)* allows for the use of financial incentives [49]. The system uses existing DDoS detection and mitigation frameworks, and in Figure 3.7 the architecture in an SDN-based network is portrayed, though the decentralized Application (dAPP) is not restricted to only SDN and includes three layers [49]:

- The SCs, deployed on the Ethereum [8] BC, where a 'central SC' stores the IP network addresses of each Autonomous System (AS) and the address of each AS's immutable SCs, which store the addresses that each entity manages.
- The dApp represents a client which serves as an interface to the Ethereum BC, as well as databases storing persistent configurations.
- The SDN controller that monitors and enforces rules.

The system includes various ASes, and each uses operates IP networks and uses its own SC [49]. When an AS detects an attack, it requests cooperative defense by performing a transaction on the SC of the AS that operates the attacker's IP address, *e.g.,* requesting to block the attacker's IP address [49]. Upon completion of the block mining process, the requested AS can, based on its own policies and threshold, decide which action to take, *e.g.,* whether to block the requested IP addresses [49].

The authors note that using a consortium-based BC enables security and trust since the participating entities are known [49].

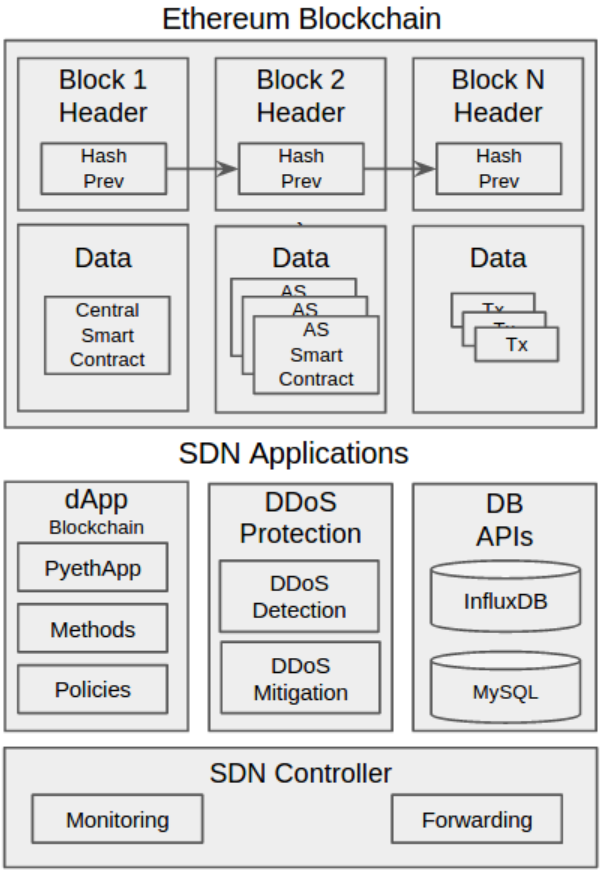

Figure 3.7: BloSS architecture in a SDN-based network [49]

3.8 Alvarenga et al. (2018)

[2] proposes a BC-based architecture that enables the secure configuration management and migration of VNFs. The usage of NFV and SFC inherently adds vulnerabilities to the network. Thus, it is paramount to reduce the potential attack vectors and employ a secure method to manage configurations since threats in the network core can potentially affect a great amount of traffic flow and immediately also a large number of users [2]. The authors note, that a compromised VNF, such as a firewall or intrusion detection system would endanger all traffic flow that is forwarded through this VNF [2]. Therefore, auditability, non-repudiation, and immutability of the configuration history are key since these aspects enable the identification of faults and compromised VNF configurations [2]. Hence, their proposed architecture is BC-based, since BC enables immutability and traceability, offering the required auditability [2]. Furthermore, the consortium-based BC uses PBFT to reach consensus due to its low latency and acceptance of malicious behavior from up to $\frac{1}{3}$ of nodes, enabling the integrity and consistency of transactions [2].

The architecture, *cf.* Figure 3.8, does not require any changes to the NFV or orchestration platforms and is agnostic to hardware or cloud platform architectures, and contains three modules [2]:

- **BC Modules** are located in the data centers and connected to other BC modules.

They each host a replica of the BC and are responsible to reach a consensus with the other BC modules.

- **VNF Client Module** are connected to one or more BC modules and run on each VNF of the architecture, implying that a change to the VNF software is necessary since the module requires installation and reading of configurations and pertinent states.

- **Tenant Client Module** "is run by VNF owners" [2] and serves as an interface for the configurations and sending transactions to BC modules.

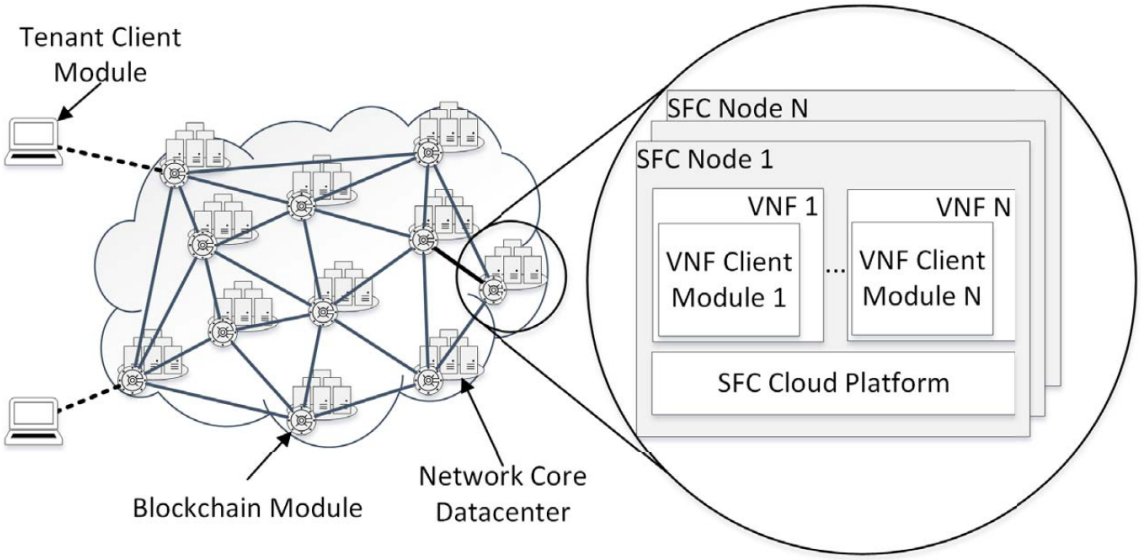

Figure 3.8: Architecture of [2]

Their work includes two types of transactions, *(i)* configuration transactions, which are issued by client modules to install the specified configuration, and *(ii)* configuration request transactions, which are used to request configuration states of VNFs [2]. The idea is that by employing asymmetric encryption on the transactions using various key pairs between the modules, the anonymity of VNFs and tenants and the confidentiality of the VNF configuration states are ensured while still offering auditing capabilities due to storing the encrypted transaction information on the BC [2].

Next to configuration management of VNFs, their work also offers migration of VNFs, since the authors note that VNF instances do not require past state information and are destroyed when shut down. Hence, VNF instances can simply be migrated by creating a new VNF instance in a new location, then transferring its configuration state, and finally wiring up the new VNF instance in the SFC [2].

3.9 Rebello et al. (2019)

Network slicing is often implemented on top of multiple cloud platforms, leading to trust issues and security problems alike. The authors of [19] describe an architecture to secure VNF orchestration and configuration by using BC technology for network slicing. Their goal is to make all orchestration commands auditable by writing them to the BC as a transaction. Concerning the VNF configuration management, each configuration is persisted on the BC, such that non-repudiation of configuration changes is achieved [19].

The architecture as depicted in Figure 3.9 defines the following components [19]:

- A **Network Slice** consists of multiple VNFs depending on the needs of a particular network segment.
- For each network slice, a separate **BC** is created by the BC creation server. Note that the type and characteristics of each BC will depend on the requirements of the corresponding network slice. The purpose of this BC is to track the VNF configuration changes of a network slice.
- The **Global Manager** consists of four modules: NFV-MANO, management BC server, BC creation server, and a user interface. The management BC server is of particular interest, as it keeps record of all the orchestration requests issued through the user interface by the users.

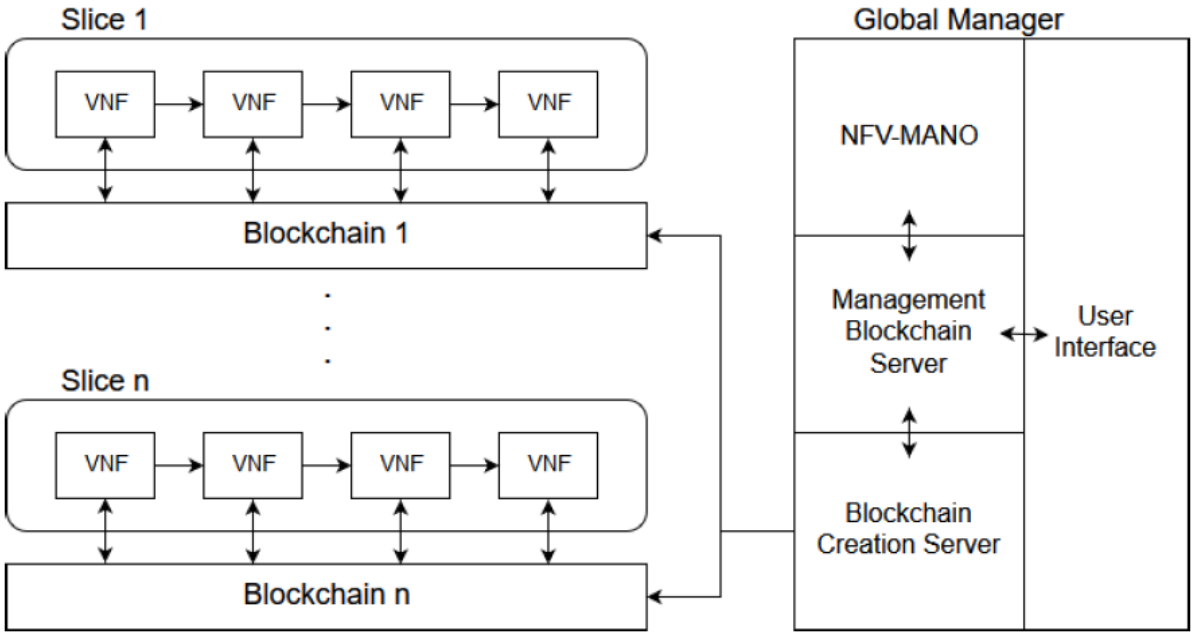

Figure 3.9: Architecture of [19]

For the prototype, the authors implemented two SCs: The first SC is deployed on the management BC. It is responsible for handling VNF orchestration commands and notifying the NFV-MANO component (NFV-MANO component polls for changes on the

BC) when there is work for it to do. The second SC is deployed on a network slice BC and is responsible for managing the VNF configuration changes of the mentioned network slice [19].

3.10 Comparison and Discussion

Although all solutions portrayed earlier in this chapter employ BCs to address particular issues, their rationale for using BCs is different. Table 3.1 compares these solutions to the solutions proposed in this report, highlighting their differences.

Table 3.1: Related Work Comparison

Solution	BC Type	MANO Aspect	BC Signaling	Automatic VNF Deployment	Use Case
[51]	Public	VNF Catalogue	✓	✗	Generic
[22]	Public	-	✓	✗	Generic
[63]	Public/Private	-	✓	✗	5G/6G
[37]	Public	NFVO/VNFM	✓	✓	5G
[48]	Private	NFVO	✗	✗	Generic
[7]	Private	NFVO	✗	✗	Authorization
[49]	Consortium	-	✓	✗	DDoS Defense
[2]	Consortium	VNFM	✗	✗	Conf. Management
[19]	Any	NFVO/VNFM	✗	✓	Network Slicing
This work	Public	NFVO/VNFM	✓	✓	Generic

Particularly, **BUNKER** [51] uses BC-based SCs both for their licensing and verification system. For the licensing, the SC emits events whenever a license is acquired. The verification system stores hashes of the published packages into the BC [51]. Furthermore, **BRAIN** [22] utilizes the power of BC-based SCs to create immutable bidding records for each auction which then can be audited and thus offer non-repudiation [22]. Similarly, **BSec-NFVO** primarily uses BC to offer auditability, non-repudiation, and integrity possibilities to secure the orchestration operations [48]. **B-VNF** [37] takes yet another approach and bases their orchestration architecture on a range of SCs that fulfill different tasks within the system [37]. The project laid out in **[63]** similarly utilizes BC for orchestration. In this case however, the BC is utilized for resources that are shared with networks of other providers instead of resources within a given provider's network. **VMOA** leverages BC technology to authenticate VM orchestration commands to guarantee their integrity [7]. The work by **[2]** uses BC to store encrypted configuration management transactions and thereby also offers the migration of VNFs, enabling the auditability of the configuration history in the process. The architecture described in **[19]** uses SCs on multiple BCs to provide auditability and non-repudiation over VNF orchestration commands as well as VNF configuration changes on particular network slices. In contrast to the aforementioned works, **BloSS** [49] employs BC to allow for a distributed, decentralized and less complex cooperation method in order to signal DDoS attack information [49].

This work differs from the aforementioned solutions in the way BC is utilized. While in the other works, the BC is mainly used to achieve auditability, non-repudiation, and integrity, this work specifically leverages BC signaling through SC-based events to allow automatic VNF deployment.

Chapter 4

Design and Implementation

Starting with a section on the capabilities and features, this chapter shows both design and implementation of blockchain-v (BCV). The section on design provides an architectural overview of the application. The implementation section focuses on the practical part of this work. In particular, it shows how the presented architectural elements are realized in terms of technology choices, including events, authentication, and authorization.

4.1 Capabilities and Features

This section shows the features and capabilities of BCV, focusing on functionality from a user's perspective. The architectural and technical details are presented in Sections 4.2 and 4.3 respectively.

- **Register User**: Users who want to use the BCV system must register to gain access to the system. Thus, registration is the first step before any other action can occur. Thus, it is the first step of authenticating the user, which is the precondition for most other use cases.
- **Unregister User**: Users can unregister from BCV if they do not wish to use the system anymore. This action revokes the user's ability to interact with the system, including VNF deployment and deletion. After unregistering, the user has no access to the system anymore, apart from the option to perform a new registration to obtain access again.
- **Create VNFD**: Registered users can create new VNF descriptors. These descriptors are shared amongst all users and serve as templates for deploying new VNFs. The templates can also contain parameters, which are to be filled in during the deployment of a VNF.
- **Create VNF**: Creating a new VNF is the most critical use case of BCV. This feature is available for registered users only. The user selects a VNF descriptor and fills in the required parameter values. Afterward, the user can initiate the deployment of the new VNF.

- **Delete VNF**: Registered users can delete their created VNFs. This feature is only available for registered users, as authentication is required to ensure that users can only delete their VNFs.

- **Show VNFDs**: Users can list all VNF descriptors registered in the system, including descriptors created by other users.

- **Show VNFs**: Users are provided an overview of their deployed VNFs, including the technical details about each VNF such as IP addresses, the descriptor used to create the VNF, and the life-cycle state.

4.2 Design

In the following section, an overview of BCV's architecture is provided. In contrast to the implementation section, this section is technology-agnostic, showing the components and their respective responsibilities. The components described below are also depicted in Figure 4.1.

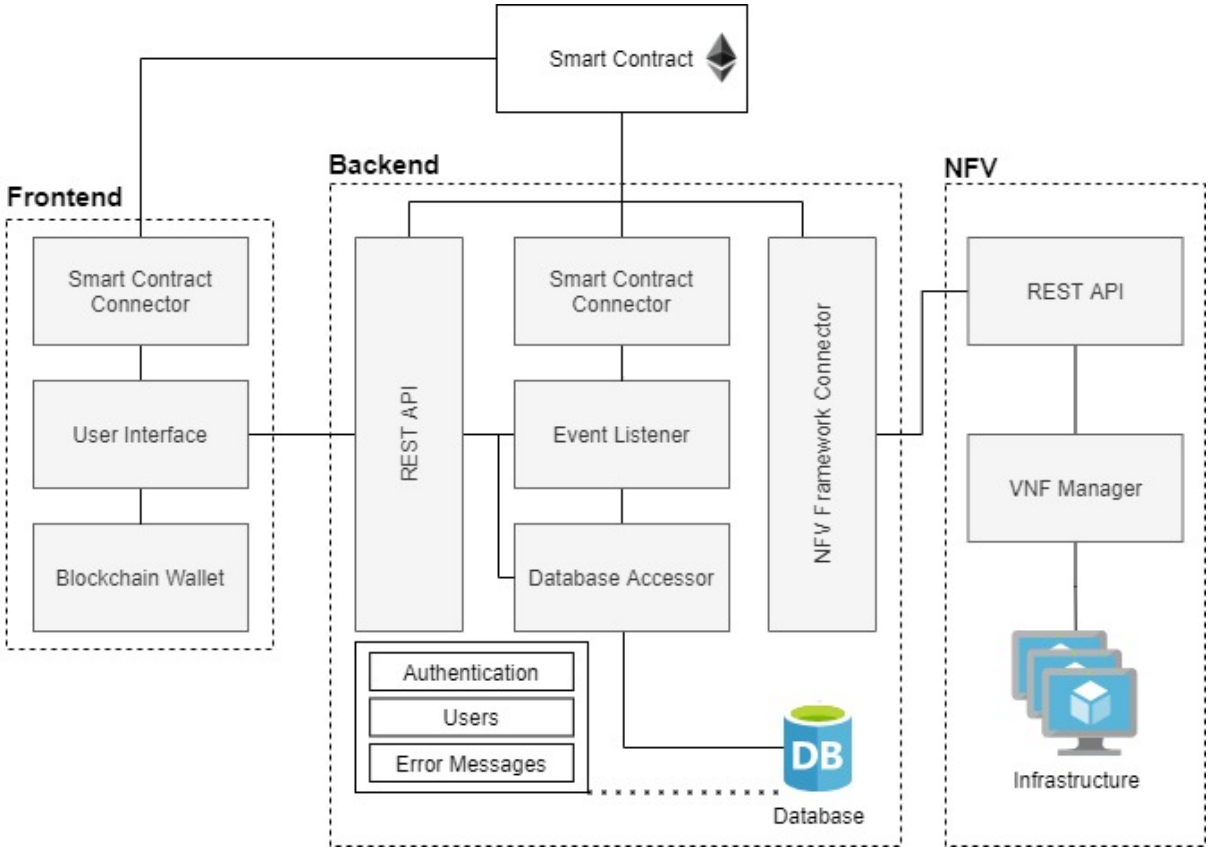

Figure 4.1: Architecture of blockchain-v

4.2.1 Frontend

The frontend provides a convenient way to interact with the BCV system. It is the entry point for the user, and it consists of multiple components, which are defined as follows:

- **User Interface**: The User Interface (UI) acts as a facade between the user and the rest of the system. It provides access to the backend to retrieve the details of instantiated VNFs. Furthermore, the user can manage VNF descriptors on the UI, which are persisted through the backend to the NFV Framework. The UI also offers a way of interacting with the SC, namely for user registration and VNF life-cycle operations.
- **Blockchain Wallet**: The BC wallet is used to interact with the BC. The user holds his BC account in the BC wallet and uses it to call methods on the SC and listen to events emitted from the BC.
- **Smart Contract Connector**: The SC Connector (SCC) enables a convenient way for the frontend to interact with the BC by prompting the user to choose the BC account desired for the interaction. Furthermore, it offers all the needed abstractions to interact with the BC. The SCC also facilitates the creation of digital signatures, which is used in BCV's authentication scheme.

Note that the frontend is also substitutable by other applications, such as BloSS [49]. The other BCV components are designed not to distinguish between the BCV frontend and any other consumer of BCV's services. This statement is true for both the SC's RPC methods and the backend's REST API.

4.2.2 Smart Contract

The SC is responsible for signaling events between the frontend and the backend. Thus, it provides RPC functions for the life-cycle management of VNFs and user management. Users must register on the SC, as it enables restricting VNF life-cycle operations to the owner of a VNF. Without authentication, it would be possible to alter the VNFs of other users, which is not desired. The SC is the single source of truth for VNF allocations (*i.e.,* which VNF belongs to whom).

Considerations have to be made regarding the data storage on the BC. While VNF-user mappings can easily be stored by the SC (*i.e.,* on the BC), VNF descriptors and VNF details can be rather large in size. To avoid scalability issues, BCV manages these two aspects in the backend.

4.2.3 Backend

The backend provides an abstraction of the NFV framework, *i.e.,* it isolates the NFV framework from the rest of the BCV system. Hence, it is the only component interacting

with the NFV framework. Thus, the backend is the most integrated component of BCV, as it interacts with the frontend, the SC, and the NFV framework. The backend consists of the following components:

- **Smart Contract Connector**: Similar to the frontend, the backend also makes use of an SCC. In this case, the SCC is used to subscribe to events emitted by the SC. Also, the SCC enables the backend to retrieve a user's VNF allocation. In addition, the backend also calls the SC's RPC methods to provide feedback concerning user registration and VNF life-cycle operations.

- **Event Listener**: The event listener uses the SCC to listen to user registration and VNF life-cycle events. These events represent commands for the backend to execute against the NFV framework. Once the event listener detects an occurring event, it reacts to it by executing the predefined action for this particular event. This action involves calling the NFV framework connector to perform life-cycle operations or interacting with the database accessor to handle user registrations.

- **Database and Database Accessor**: The database accessor is used to retrieve and store user and authentication information on the database. The database accessor and consequently also the database are used both by the event listener as well as the REST API component.

- **REST API**: The REST API acts as a backend-for-frontend. It offers endpoints for handling VNF descriptors, accessing VNF details, and authentication. The former two endpoint types transfer data that would be infeasible to store on the BC because of their size and frequency.

- **NFV Framework Connector**: The NFV framework connector is used by the REST API and the event listener. It consumes the NFV framework's REST API, which is exposed by the backend to execute VNF life-cycle operations.

4.2.4 Authentication

The SC is responsible for user authentication, as it needs to know the addresses of users who want to execute VNF life-cycle operations. In Solidity, `msg.sender` yields the address of the caller, which is used for authentication in this case.

The backend is responsible for authenticating users when using the REST API. The frontend first asks the backend for a nonce. Afterwards, the frontend asks the user to digitally sign the nonce (to prevent replay attacks) and then submits the signature with the nonce and the user's address to the backend's REST API via HTTP POST. The backend verifies if the signature is valid. If the signature is valid, the backend generates a token (*e.g.,* JWT) and returns it to the frontend. If the signature is not valid, the API denies the issuance of the token by returning HTTP status code 403.

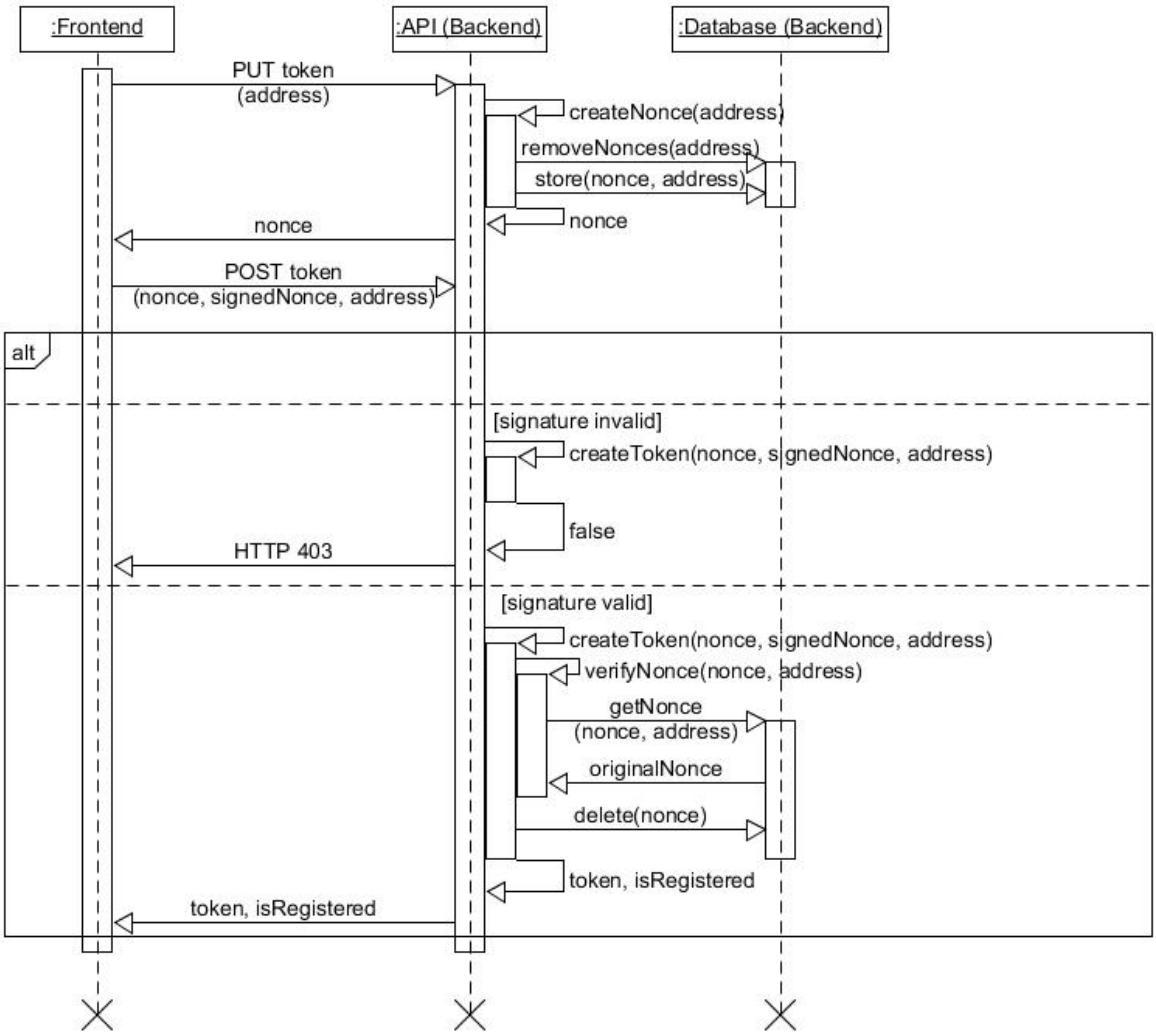

Figure 4.2: Process of obtaining an authentication token

4.2.5 Authorization

In terms of SC authorization, BCV distinguishes multiple scopes. First, only registered users are allowed to deploy VNFs. Second, VNFs are only modifiable by their creators. Third, registered users are allowed to retrieve their VNFs, including all the details stored in the SC. Fourth, the creator of the SC can nominate a backend by registering its address in the contract. Lastly, the report functionalities (*i.e.,* `reportDeployment`, `reportDeletion`, `reportRegistration`, and `reportUnregistration`) are only accessible to the registered backend. The detailed permissions on the SC are shown in Table 4.1.

The backend allows users to retrieve their VNFs and all the registered VNFDs. The authorization process requires a valid token (*e.g.,* JWT) to identify the user and decide on the authorization. This token is verified as follows (*cf.* Figure 4.3): The frontend includes the token in the *Authorization* HTTP header. The backend validates the token, which also involves checking whether the token has expired. It should be noted that

Table 4.1: Permission matrix of SC operations

SC Operation	User	SC Creator	Backend
`registerUser`	✓	✓	✓
`unregisterUser`	✓	✓	✓
`deployVNF`	✓	✗	✗
`deleteVNF`	✓	✗	✗
`registerBackend`	✗	✓	✗
`reportRegistration`	✗	✗	✓
`reportUnregistration`	✗	✗	✓
`reportDeployment`	✗	✗	✓
`reportDeletion`	✗	✗	✓
`getVnfs`	✗	✗	✓

the authors are aware that this implementation does not offer resistance against every known attack vector. However, as the scope of this work is not the authentication and authorization of REST APIs, this mechanism serves as a placeholder and could easily be replaced with a standardized protocol (*e.g.,* OpenID Connect [21]) in the future.

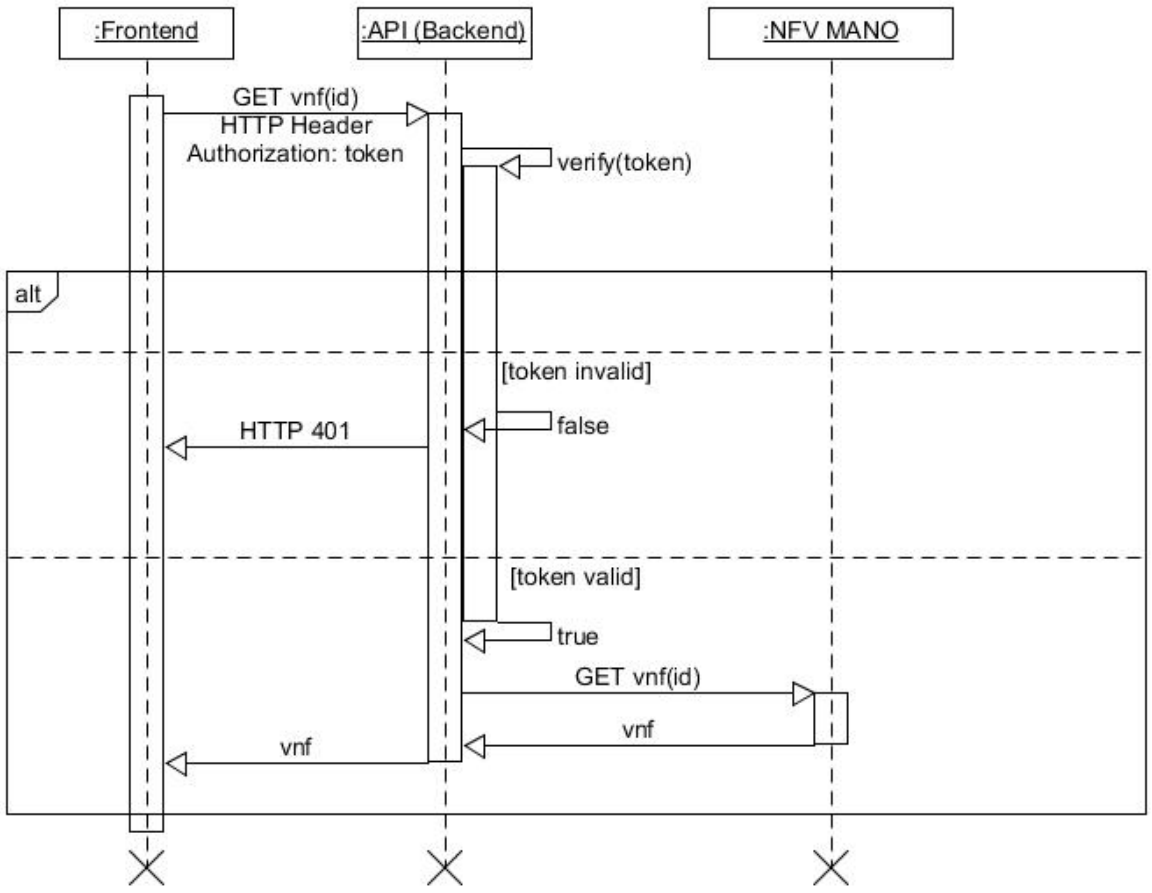

Figure 4.3: Process of verifying an authentication token

4.2.6 NFV Framework

The NFV framework implements the MANO functionality by exposing a well-defined API. This API is where BCV's backend integrates with the NFV framework. Hence, BCV is agnostic to such a framework's implementation, meaning that the NFV framework is interchangeable. For this purpose, only the NFV framework connector of the backend has to be adjusted. Internally, the NFV framework comprises the following components:

- **REST API**: The REST API provides endpoints for all aspects of VNF management and orchestration. These aspects include (but are not limited to) virtualization infrastructure management, network configuration, VNF descriptor handling, and VNF life-cycle operations. BCV uses primarily VNF descriptor and VNF life-cycle endpoints, while the other aspects are pre-configured during setup.

- **VNF Manager**: The VNF manager offers the functionality behind the exposed REST APIs, namely VNF life-cycle operations. It translates life-cycle requests to orchestration commands on the VNF infrastructure.

- **VNF Infrastructure**: The VNF infrastructure represents the lowest level of the NFV framework. It handles the virtualization aspects such that VNFs can be instantiated on top of it.

4.3 Implementation

This section highlights the key aspects and details of BCV's implementation. Thus, the technologies used for the individual components are presented (*cf.* Figure 4.4), alongside the interplay of these components, showing how the functionality specified in Section 4.1 is achieved from a technical standpoint.

4.3.1 Frontend

The technology of choice for the frontend was the JavaScript framework 'Vue.js' [66], as the goal was to create a web application, and there was prior experience with the framework within the team. As already alluded to in Section 4.2.1, the frontend is, by design, not a required component of the system and can be substituted. Alternatively, an entirely programmatic approach is also possible, where not frontend is used at all and interactions with the SC and backend are performed directly, without a frontend as intermediary. Consequently, this section only discusses aspects of the frontend that provide functionality to the BCV system, which substitutes of the frontend also would have to address.

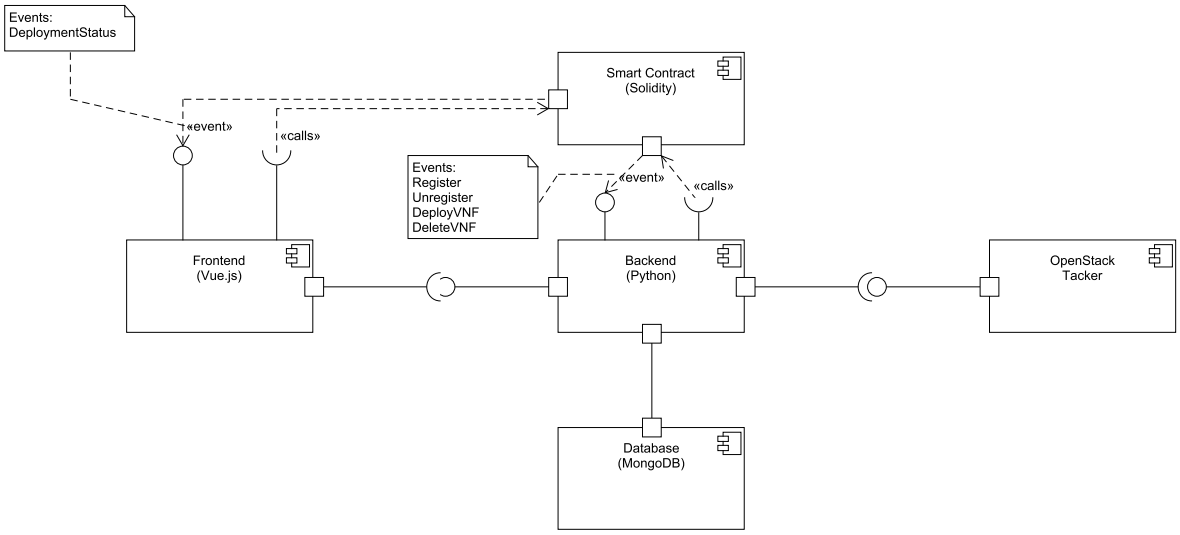

Figure 4.4: Components of BCV

State Management

To simplify the data flow between the individual Vue components of the frontend, the decision was made to use the 'Vuex' library, which introduces state management patterns that essentially result in a data store that is globally accessible within the entire frontend application [29]. Thus, data from backend calls, contract calls, incoming contract events, as well as just regularly accumulating state used to manage the UI could be easily made accessible to whatever component or service required it.

Wallet and Ethereum API

The crypto wallet the frontend is designed to work with 'MetaMask' [1]. It is available as an extension for various browsers, such as Google Chrome and Firefox. Its purpose for the BCV system is to hold the accounts and keys with which transactions are sent from the frontend to the SC, as well as to serve as part of the UI, such that users can confirm transactions and, if needed, provided signatures.

From the perspective of the frontend codebase, the library 'web3.js' [23], was used to connect to Ethereum nodes through the Ethereum API. The library enables the frontend to initiate calls to the SC using the accounts selected in MetaMask, as well as subscribing to web3's BC event listeners such that the frontend can react to events emitted by the SC. Additionally, the library provides methods to perform signatures using the connected MetaMask accounts, which are used in the frontend for authentication purposes.

To perform contract calls, the SC must be identifiable for the frontend, meaning its address on the BC and the specification of methods it exposes must be introduced into the codebase somehow. In the case of BCV, the contract's Application Binary Interface (ABI) [26], or, more precisely, the JSON description of the ABI, is supplied in file format to the codebase. This essentially specifies how the frontend has to encode Solidity contract

calls for the EVM and, conversely, how to decode data from the EVM [16]. The SC address, meanwhile, is supplied as an environment variable.

The thought process behind this design decision was that, whilst the ABI can be considered constant regarding method and event specification in the scope of this work, the contract should still be able to be re-deployed without having to replace the ABI JSON each time. Externalizing the contract address allows one to do so and essentially reduces the coupling between frontend and SC, such that different instances of the contract can be used interchangeably by simply altering the address in the environment variables.

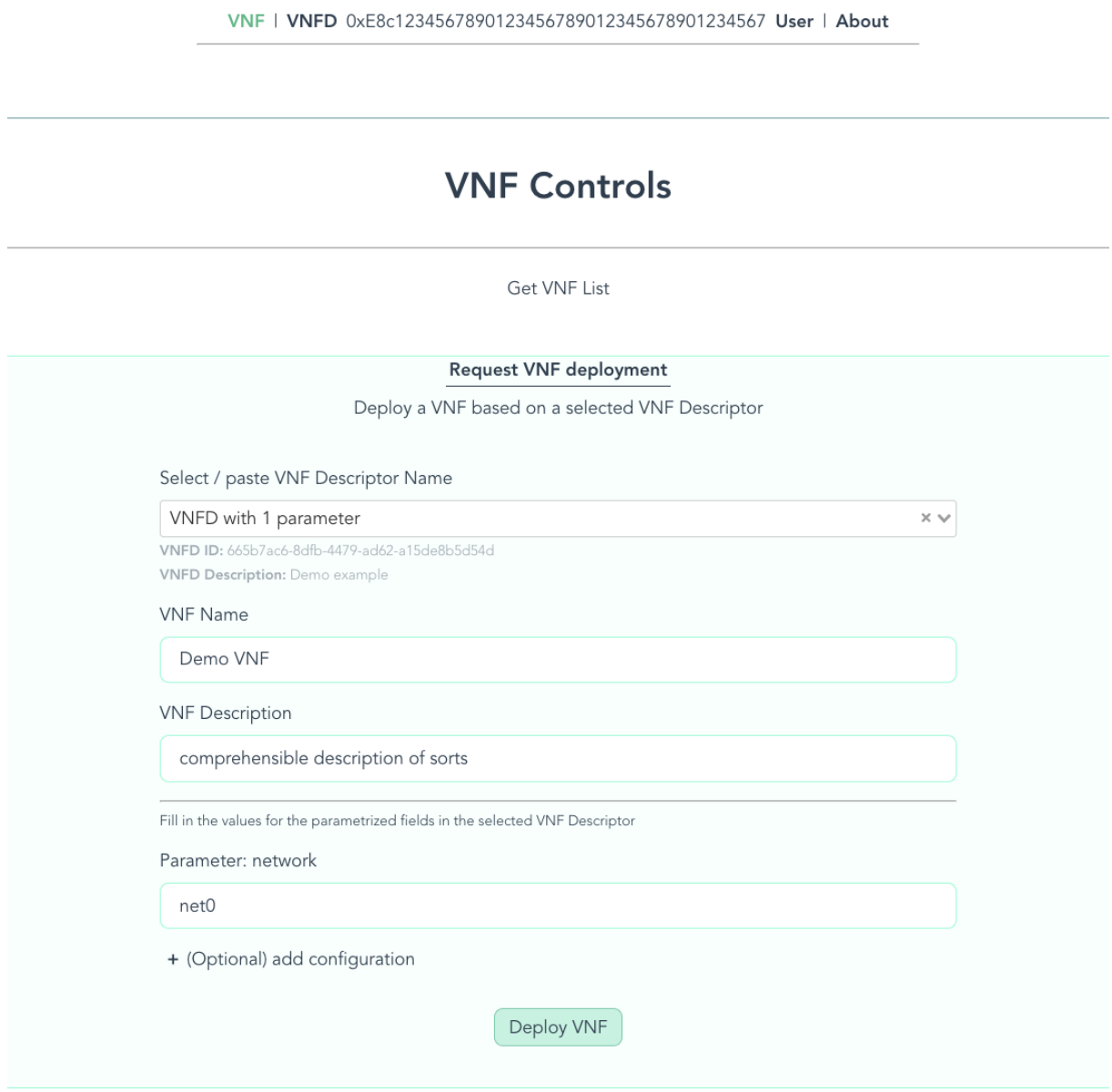

Figure 4.5: VNF Deployment Frontend UI Excerpt

Smart Contract Connector

The SCC's role is to facilitate interactions between the frontend and the SC. At its core, it consists of three services:

- `truffleService.js`: provides functionality regarding the connection with Metamask, web3, and the contract ABI.
- `contractCallService.js`: provides methods for performing contract calls on the SC methods specified on the contract's ABI.
- `eventListenerService.js`: provides methods for handling SC events.

Figure 4.5 depicts an excerpt of the UI for Deploying VNFs. These controls allow the user to interact with the SC, provided that they chose to use a frontend rather than a programmatic approach. Here, selection of the desired VNFD and specification of name, description and additional inputs in case of parameterization happen. Once the 'Deploy VNF' button is pressed, the code shown in Listing 4.1 comes into play.

Listing 4.1 shows the general structure of a contract call at the example of the 'DeployVNF' functionality. A strict pre-requisite is that an Ethereum account is available for actually sending the contract call in a transaction. Line `4` guards against the possibility of proceeding without such an account. Next, the data required for the SC method in question is read from the store (*cf.* Line `6`), and in the case of this particular contract call, the `parameters` object is constructed.

Once done, the request for the contract call has to be built (*cf.* Lines `22 - 25`) with the parameters as specified in the contract ABI and is subsequently sent to the contract in a transaction on Line `28`. In that `send` method, additional parameters have to be supplied, such as `from`, meaning the account address from which the transaction is sent, as well as the `gasPrize` and `gasLimit`, which are returned from the `getDefaultCallParams` method.

Notably absent are return values. While the contract will return data upon completing (or failing) the call, these values are not of interest to the BCV system. The relevant return values are supplied at a later time through the event mechanisms described in Section 4.2. To be able to receive such events and handle them properly, the frontend employs the web3 library's subscription and events mechanisms [24].

Listing 4.2 displays how BC events can be subscribed to with the use of the web3 library. Note that the code has been simplified and stripped of elements that are not essential for understanding the subscription mechanism for the sake of readability.

The key part is on Line `13`, where, following the web3 documentation, the event subscription occurs. Using the `filter`, the listener is told to only trigger the callback function for events that specify the user in question. The exposed `attacheEventListener` method then can then be used as demonstrated on Line `17`. Accessing data sent within the event then happens in the callback function by accessing the `returnValues` field in the `error` respectively `event` parameter.

```
1  const performContractCall_deployVNF = () => {
2      // grab ethereum account from vuex
3      const account = store.getters["contracts/getUserETHAccount"];
4      if (!_isNil(account)) {
5          // fetch data relevant for contract call from store and prepare
               parameters argument
6          const callData = store.getters["contracts/getDeployVnfData"];
7          const VNFD_ID = callData[fieldNames.VNFDID];
8          const parameters = {
9              name: callData[fieldNames.NAME],
10             description: callData[fieldNames.DESCRIPTION],
11         };
12         // attributes field has to be present no matter what
13         parameters["attributes"] = _isNil(callData[fieldNames.ATTRIBUTES
               ])
14             ? {}
15             : callData[fieldNames.ATTRIBUTES];
16         // add config field to attributes if config present
17         const config = callData[fieldNames.CONFIG];
18         if (!_isNil(config) && config !== "") {
19             parameters["attributes"]["config"] = config;
20         }
21         // build contract call
22         const deployVNFrequest = VNFContract.methods.deployVNF(
23             VNFD_ID,
24             JSON.stringify(parameters)
25         );
26         deployVNFrequest
27             .send(getDefaultCallParams(account))
28             .then(() => { store.commit("appState/setAwaitingContract",
                   true) })
29             .catch((error) => {
30                 console.warn("deployment transaction failed with error",
                       error);
31                 store.commit("appState/setIsLoading", false);
32             });
33     } else {
34         console.log("no user account to make call with, rejecting");
35     }
36 };
```

Listing 4.1: Performing SC Call (annotated)

```
1  // expose attaching method
2  export const attachEventListener = (eventType, callback) => {
3      let e;
4      // assign to e the ABI specification of the supplied eventType
5      return registerEventListener(e, callback);
6  };
7
8  function registerEventListener(
9      eventType,
10     callback,
11     filter = { filter: { user: [window.web3.eth.defaultAccount] } }
12 ) {
13     return eventType(filter, callback);
14 }
15
16 // example usage
17 attachEventListener(EventTypes.DeploymentStatus, myCallbackFunction(
       error, event))
```

Listing 4.2: Attaching SC Event listeners (simplified & annotated)

Authentication and Authorization for Backend Calls

Before the frontend is allowed to perform calls to most of the backend's endpoints (*cf.* Sections 4.2.4, 4.2.5) it must authenticate itself with the backend. This is done by requesting a nonce from the backend, signing it using the user's Ethereum account using the web3 library's signature mechanism, and sending a payload consisting of account address, nonce, and signed nonce to the backend for signature verification (*cf.* Section 4.3.3).

Should the verification fail, indicating that the account in question is not registered with the SC yet, the registration process is initiated, and afterward, the authentication flow is run again. If the verification succeeds, the received authentication token is stored as a cookie, which then is sent along in each backend call that requires authorization.

To illustrate the authentication process, Listing 4.3 shows the steps of the procedure outlined above. The flow consists of two overarching methods, which are triggered by lifecycle hooks of the `Home.vue` component and changes within the vuex store, the details of which are omitted here.

In the first step of the authentication flow, the account is fetched (the presence of an account at this step is guaranteed through routing guard mechanisms) and used to perform a `PUT` call to the `/token` endpoint on Line `6`.

Once a nonce is retrieved from the backend, the procedure continues with the second part of the authentication flow. The crucial point here is the creation of the signature on Lines `14` through `17`, where the web3 library's `sign` method is utilized to request the nonce be signed using the user's Ethereum account.

Once the signature is present, a `POST` call is made to the same `/token` route on Line `20`, such that the backend now has all the information it needs to authenticate the user.

```
1  // first part of the flow
2  async initiateUserRegistrationCheck() {
3    this.$store.dispatch("appState/setIsLoading", true);
4    const account = this.$store.getters["contracts/getUserETHAccount"];
5    // get a nonce from the backend
6    await this.getNonce(account);
7    this.$store.dispatch("appState/setIsLoading", false);
8  }
9  // second part of the flow
10 async concludeUserRegistrationCheck() {
11   const account = this.$store.getters["contracts/getUserETHAccount"];
12   const nonce = this.nonce;
13   // sign nonce
14   const signedValue = await window.web3.eth.sign(
15     window.web3.utils.sha3(nonce),
16     account
17   );
18   this.$store.dispatch("appState/setIsLoading", true);
19   const payload = [nonce, signedValue, account];
20   await this.performTokenCheck(payload);
21   this.$store.dispatch("appState/setIsLoading", false);
22 }
```

Listing 4.3: Authentication flow (annotated)

Note that the calls to the `/token` route are wrapped in their own methods called `getNonce` and `performTokenCheck` respectively, with the details of those calls not being shown since they are just regular HTTP requests.

4.3.2 Smart Contract

The VNF deployment SC of BCV was implemented in Solidity [27] on top of the Ethereum BC [8]. The SC stores the mappings between users and their deployed VNFs, and acts as an *event bus* between the frontend and the backend (*i.e.,* BC signaling). For the development process, Ganache [13] was used to run a development Ethereum BC. The communication through the SC follows the same pattern: The user (*i.e.,* frontend) calls one of the initiation functions (*i.e.,* `registerUser`, `unregisterUser`, `deployVNF`, and `deleteVNF`) to emit an event, which is received by the backend. Then, the backend reacts to these events and performs the specified task. After the completion of the task, the backend calls one of the SC's feedback functions (*i.e.,* `reportRegistration`, `reportUnregistration`, `reportDeployment`, and `reportDeletion`), which emits another event to be received by the frontend.

The following sections detail the implementation of the most important features of the BCV `VNFDeployment` contract, such as the *user registration*, *VNF deployment*, *VNF deletion*, and *VNF retrieval*.

User Registration

To use the SC and the BCV solution, the user has to register with the SC. For this purpose, the user calls the `registerUser` function (*cf.* Listing 4.4), which emits an event that is forwarded to the backend. Note that this function call does not alter the state of the contract yet, as the registration might fail. Failure could occur, for example, if the signature validation in the backend fails.

```
1 // Registers the sender of a transaction as a user
2 /// @param signedAddress signature of the user's address
3 function registerUser(string memory signedAddress) public {
4   address user = msg.sender;
5
6   emit Register(user, signedAddress);
7 }
```

Listing 4.4: VNFDeployment::registerUser

Following the event being emitted, the backend checks the signature and reports the registration status back to the SC by calling `reportRegistration` (*cf.* Listing 4.5). If the registration was successful on the backend's side, the SC registers the user in a mapping and emits another event to the frontend to signal the success of the registration. In the case of an error, the same event is emitted, but it notifies the user that something went wrong during the registration process.

```
1 // Enables the backend to signal the status of user registration.
2 /// @param user User to be registered.
3 /// @param success Indicates whether the user was registered correctly.
4 function reportRegistration(address user, bool success) public {
5   require(msg.sender == backend, "Only the backend is allowed to call
        this function.");
6
7   if(success){
8     users[user] = true;
9   }
10
11  emit RegistrationStatus(user, success);
12 }
```

Listing 4.5: VNFDeployment::reportRegistration

In addition, the SC offers functions to deregister users (*i.e.,* `unregisterUser`), as well as a feedback function for the backend (*i.e.,* `reportUnregistration`) for this scenario. Both functions work analogous to the registration functions but disable the user account instead.

VNF Deployment

Once the user is registered, she can deploy VNFs. The SC stores VNFs in the `VNF` struct (*cf.* Listing 4.6). It consists of the following fields:

- `deploymentId` is the primary identifier for the SC to operate on VNFs. It is generated within the SC and uniquely identifies each VNF.
- `vnfdId` represents the identifier of the VNF descriptor used by the NFV framework (*e.g.,* OpenStack Tacker), denoting the template to be used for the deployment of the VNF.
- `vnfId` is the VNF identifier assigned by the NFV framework. It is only available after the VNF has been created successfully.
- `owner` contains the BC wallet address of the user deploying a VNF.
- `parameters` holds the parameter values required by the respective VNF descriptor to instantiate a new VNF.
- `isDeployed` a boolean indicating whether the respective VNF has been deployed or not.
- `isDeleted` a boolean indicating whether a particular VNF has been deleted or not.

Note that the SC does not store VNF descriptors. The reason for this decision is that storage is costly on the BC, particularly for larger data structures, such as the TOSCA templates inside the VNF descriptors. Instead, these descriptors are solely stored within the NFV framework (*e.g.,* OpenStack Tacker) and managed by the user on the frontend (via backend).

```
1 struct VNF {
2   uint deploymentId;
3   string vnfdId;
4   string vnfId;
5   address owner;
6   string parameters;
7   bool isDeployed;
8   bool isDeleted;
9 }
```

Listing 4.6: VNFDeployment VNF Struct

To trigger the deployment of a new VNF, the user calls the `deployVNF` function (*cf.* Listing 4.7) on the SC. After checking that the user is registered with the system, a new VNF struct is populated. Note that at this point, the `vnfId` is not yet available, as the VNF has not been deployed yet. The VNF struct is then added to the user's list of VNFs. The last step of this function is emitting an event to the backend, signaling the command to deploy a new VNF.

```
1 // Deploys a VNF by emitting a deployment event.
2 /// @param vnfdId identifier of the VNF descriptor (VNFD), which is
3 /// the template to be used to create a VNF instance.
4 /// @param parameters instantiation parameters according to the VNFD
    template.
```

```
5  function deployVNF(string memory vnfdId, string memory parameters)
       public {
6    address user = msg.sender;
7
8    require(users[user], "User not registered.");
9
10   uint deploymentId = createDeploymentId();
11
12   VNF memory vnf = VNF(deploymentId, vnfdId, "", user, parameters, false
       , false);
13
14   addVnf(vnf, user);
15
16   emit DeployVNF(user, deploymentId, vnfdId, parameters);
17 }
```

Listing 4.7: VNFDeployment::deployVNF

After the backend has triggered OpenStack Tacker to deploy a new VNF, it reports the status of this operation back to the SC by calling the `reportDeployment` function (*cf.* Listing 4.8). Note that the backend passes along the `vnfId` of the created VNF. The SC adds the `vnfId` to the existing VNF record of the user. If the deployment has failed on the backend side, the VNF is removed from the user's VNF list. In any case, the last action is to emit an event for the frontend, informing about the status of the VNF deployment.

```
1  // Enables the backend to signal the status of VNF instantiation
2  // by handing over the VNF resource identifier of the backend.
3  /// @param deploymentId VNF identifier as specified in this contract.
4  /// @param user User owning the VNF
5  /// @param success Indicates whether the VNF was instantiated correctly.
6  /// @param vnfId VNF identifier specified by the backend.
7  function reportDeployment(uint deploymentId, address user, bool success,
       string calldata vnfId) external {
8    require(msg.sender == backend, "Only the backend is allowed to call
       this function.");
9
10   uint index = findVnfIndex(deploymentId, user);
11
12   require(vnfs[user][index].deploymentId > 0, "VNF must exist in order
       to be activated.");
13
14   if(success){
15     // add vnfId to existing VNF record
16     vnfs[user][index].vnfId = vnfId;
17     vnfs[user][index].isDeployed = true;
18   } else {
19     // remove vnfId from registered VNF list
20     removeVnfHard(deploymentId, user);
21   }
22
23   emit DeploymentStatus(deploymentId, user, success, vnfId);
24 }
```

Listing 4.8: VNFDeployment::reportDeployment

VNF Deletion

If a user wants to delete an existing VNF, she calls the `deleteVNF` function (*cf.* Listing 4.9). This function first checks if the user is authorized to delete the particular VNF (by ensuring that the user owns the VNF) and then emits an event to initiate the deletion of the VNF on the backend.

```
1  // Deletes a VNF by emitting a deletion event.
2  /// @param deploymentId identifier of the VNF instance to be terminated.
3  function deleteVNF(uint deploymentId) public {
4    address user = msg.sender;
5
6    require(users[user], "User not registered.");
7
8    uint index = findVnfIndex(deploymentId, user);
9
10   require(vnfs[user][index].owner == user, "VNF must exist and can only
       be deleted by its owner");
11
12   emit DeleteVNF(user, deploymentId, vnfs[user][index].vnfId);
13 }
```

Listing 4.9: VNFDeployment::deleteVNF

Once the backend has deleted the VNF by calling OpenStack Tacker, it reports the deletion of the VNF back to the SC by calling the `reportDeletion` function (*cf.* Listing 4.10). The VNF is then marked as deleted. Finally, the frontend is notified of the deletion status by the emitted event.

```
1  // Enables the backend to signal the status of VNF deletion.
2  /// @param deploymentId VNF identifier as specified in this contract.
3  /// @param user User owning the VNF
4  /// @param success Indicates whether the VNF was instantiated correctly.
5  function reportDeletion(uint deploymentId, address user, bool success)
       public {
6    require(msg.sender == backend, "Only the backend is allowed to call
       this function.");
7
8    if(success){
9      removeVnf(deploymentId, user);
10   }
11
12   emit DeletionStatus(deploymentId, user, success);
13 }
```

Listing 4.10: VNFDeployment::reportDeletion

VNF Retrieval

The SC is the single source of truth for storing the mapping between the user and the deployed VNFs. Hence, it provides a straightforward way of retrieving all VNFs of a

particular user. For this purpose, it offers the `getVnfs` function (*cf.* Listing 4.11), which is a read-only function. The read-only property is essential, as calling this function does not involve any cost, making it safe to use in the backend, where it is called frequently.

```
1 /// Returns all the VNFs of the calling user
2 function getVnfs(address user) public view returns (VNF[] memory) {
3   require(msg.sender == backend, "Only the backend is allowed to call
        this function.");
4
5   return vnfs[user];
6 }
```

Listing 4.11: VNFDeployment::getVnfs

4.3.3 Backend

The backend was implemented using Python [45], which was chosen since all team members are familiar with the technology, and the connection to the BC could be done with the popular 'Web3.py' library [57] for Python. Similar to the frontend and the database, the backend has been set up as a Docker container [14] to allow for an easier deployment pipeline and virtualization advantages. As described in Section 4.2.3, the backend includes several responsibilities, and their implementation will be discussed in the following subsections, though the parts regarding the 'Database' will be documented in its own separate Section 4.3.4.

REST API

The REST API component of the backend has been set up using OpenAPI [40]. It allows for a concise overview and provides documentation of the API routes that the backend offers to clients. Additionally, Flask [43] is used as a framework to serve the API routes. In Figure 4.6 the API routes are depicted, and additionally, request and response message formats, as well as HTTP response codes, are specified and documented, such that potential other clients would be able to connect to the REST API effortlessly. Furthermore, all API routes, except for the ones to create a token, are secured, *i.e.,* require a valid authorization token, *cf.* Sections 4.2.4 and 4.2.5.

Token issuance

To create a valid authorization token, *cf.* Sections 4.2.4 and 4.2.5, the clients must first request a nonce and provide their valid address. Listing 4.12 depicts the static `create_nonce` method, which is part of the `TokenService` class. It expects an 'AddressRequest', which is an instance of a class that models HTTP requests, which provide a user address in the data body. The method returns an error if a given request's address is invalid or if an exception occurred during the creation of a nonce. If the creation of a

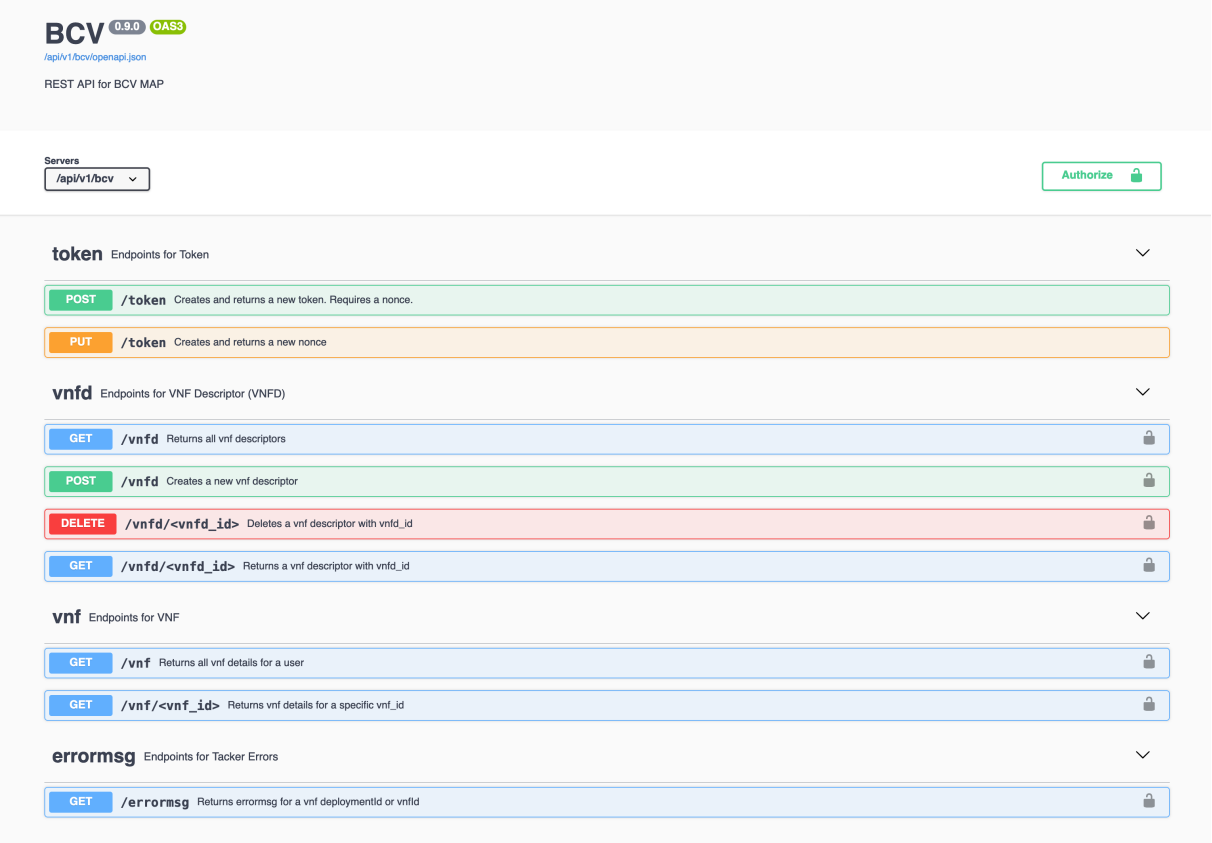

Figure 4.6: OpenAPI specifications of the REST API

nonce was successful (*cf.* Line 10), the HTTP code 201 is returned to the client, along with the newly created nonce.

```
1     @staticmethod
2     def create_nonce(address_request):
3         """
4         :param address_request: AddressRequest
5         :return: Response
6         """
7         if not w3.isAddress(address_request.address):
8             return "Error", 403
9         nonce = TokenService.create_nonce_handler(address_request.
              address)
10        if nonce:
11            return {"nonce": nonce}, 201
12        else:
13            return "Error", 403
```

Listing 4.12: The `create_nonce` method

The actual nonce creation is delegated to the `create_nonce_handler` method, *cf.* Listing 4.13. This method uses the 'uuid' module from the Python standard library [46], which is used to create unique IDs, which are then used as nonces (*cf.* Line 10), and returned. The new nonce and the corresponding user address are then stored in the database (*cf.*

Line 11). Saving this association between nonce and user address prevents other users from creating tokens with nonces that do not belong to them. Also, to avoid clients having multiple nonces at once, all previously issued nonces from this user's address are deleted from the database when this method is called (*cf.* Line 9).

```
1     @staticmethod
2     def create_nonce_handler(address):
3         """
4         Stores new nonce in the db and returns new nonce for a user and
              deletes the previously issued nonce that belonged to that
              user
5         :param address: string
6         :return: nonce_val : string : a new nonce
7         """
8         try:
9             Nonce.objects(address=address).delete()
10            nonce_val = "0x" + uuid4().hex
11            Nonce(address=address, value=nonce_val).save()
12            return nonce_val
13        except Exception as e:
14            log.error(f"Creating nonce failed {e}")
15            return False
```

Listing 4.13: The `create_nonce_handler` method

When users have received a nonce, they are allowed to create tokens for their session. To do so, the method `create_token` is called, *cf.* Listing 4.14. It expects a 'TokenRequest', an instance of a class that is used to model requests for tokens, that includes the attributes 'address', 'nonce', and 'signed_nonce'. This method verifies whether a user is registered (*cf.* Line 7-9), before calling the `create_token_handler` method. This verification is done since tokens are only issued to registered users. Once a token has been created, the nonce can be seen as consumed and is deleted from the database (*cf.* Line 16). Lastly, Line 19 shows that the token is returned to the client, along with a flag whether the user is registered, similarly to the case in Line 9, when the user is not yet registered, which is required for the frontend to distinguish between both cases.

```
1     @staticmethod
2     def create_token(token_request):
3         """
4         :param token_request: TokenRequest
5         :return: Response
6         """
7         is_registered = userService.service.is_user_registered(
              token_request.address)
8         if not is_registered:
9             return {"isRegistered": False}, 200
10        token = TokenService.create_token_handler(
11            token_request.nonce, token_request.signed_nonce,
                  token_request.address
12        )
13        if token:
14            # nonce has been consumed
```

```
15              try:
16                  Nonce.objects(address=token_request.address).delete()
17              except DoesNotExist:
18                  pass
19              return {"token": token, "isRegistered": is_registered}, 201
20          else:
21              return "Error", 403
```

Listing 4.14: The `create_token` method

Similarly to the `create_nonce` method, the `create_token` method also delegates the creation of tokens to a handler method, `create_token_handler`, *cf.* Listing 4.15. This method's responsibility is to issue JSON Web Tokens (JWT) [61], which are then used for stateless authentication, once issued, implying that there is no need to store tokens in the database. To do so, the 'PyJWT' library [58] is used to encode the user's address and the expiration date into the JWT. The token's validity has been arbitrarily chosen as 24 hours. However, this step is only done after the `check_auth` method, *cf.* Listing 4.18 verified the provided authentication credentials of the user.

```
1       @staticmethod
2       def create_token_handler(nonce, signed_nonce, address, secret=config
            .JWT_SECRET):
3           """
4           Creates and returns new jwt token for a user if the passed nonce
                is valid
5           :param nonce: string
6           :param signed_nonce: string
7           :param address: string
8           :param secret: string
9           :return: token : string
10          """
11          try:
12              if check_auth(claim=nonce, signed_claim=signed_nonce,
                    address=address):
13                  token = jwt.encode(
14                      {
15                          "address": address,
16                          "exp": datetime.utcnow() + timedelta(hours=24),
17                      },
18                      secret,
19                  )
20                  return token
21              else:
22                  return False
23          except Exception as e:
24              log.error(f"Creating token failed {e}")
25              return False
```

Listing 4.15: The `create_token_handler` method

Authentication & Authorization

In the case of the secured API routes, which are depicted in Figure 4.6, the method `authorize`, *cf.* Listing 4.16, is implicitly called. This method verifies if the token belongs to a registered user (*cf.* Line 7), which is shown in Listing 4.17 and returns to the called API controller method the 'userAddress' as a string that was decoded from the token. Hence, with this setup, all secured controller methods have direct access to the address of the user.

```
1  def authorize(token):
2      """
3      Authorizes a user and returns the user address
4      :param token: str
5      :return: dict
6      """
7      if not verify_token(token):
8          return abort(401)
9      user_address = get_address_from_token(token)
10     return {"userAddress": user_address}
```

Listing 4.16: The `authorize` method

Listing 4.17 shows the method that is responsible for checking if a user's address that was provided in a token actually belongs to a user, *i.e.,* that the user is also registered. To do so, the provided token is decoded (*cf.* Line 8), and it is checked in the database that a user with this given address exists. Additionally, if the token is not valid anymore, an 'ExpiredSignatureError' exception is raised, and 'False' is returned.

```
1  def verify_token(token_str) -> bool:
2      """
3      Verifies if a given token exists and is still valid
4      :param token_str: str
5      :return: bool : is the token valid
6      """
7      try:
8          token_data = decode_token(token_str)
9          user = User.objects.get(address=token_data["address"])
10         return len(user) > 0
11     except (InvalidTokenError, ExpiredSignatureError, DoesNotExist) as e
           :
12         log.info(f"token errored: {e}")
13         return False
```

Listing 4.17: The `verify_token` method

The method `check_auth`, *cf.* Listing 4.18, is responsible for checking whether the provided authentication credentials are valid. It has two variants, that are similar in concept, (*i*) verifying a claimed address, *cf.* Listing 4.19, which is used in the user registration process, and (*ii*), verifying a claimed nonce, *cf.* Listing 4.20, which is being used in the token issuance context. The appropriate method is chosen depending on whether an 'address' was passed as a keyword to this method (*cf.* Line 12).

```python
def check_auth(*args, **kwargs):
    """
    Verifies a claim (e.g. an address) by comparing it to its value that
        was recovered from a digitally signed string
    :return: bool
    """
    try:
        address = kwargs.get("address")
        claim = kwargs.get("claim")
        signed_claim = kwargs.get("signed_claim")
        return (
            _check_auth_for_address(claim, signed_claim)
            if address is None
            else _check_auth_for_nonce(claim, signed_claim, address)
        )
    except:
        log.info("failed to verify authentication signature")
        return False
```

Listing 4.18: The `check_auth` method

The idea behind both variants is to hash the provided claim that was given either from the SC in the case of the user registration process or from the frontend while issuing a nonce. The hashing is done using the 'solidityKeccak' function from 'Web3.py' [57], to ensure that the same hashing algorithm is used in the SC, frontend, and backend. Subsequently, the address is recovered from the hashed string using the digital signature, *i.e.,* the 'signed_claim' and it is verified if the recovered address is equal to the passed user address. This implies that the address belongs to the user.

```python
def _check_auth_for_address(claimed_address, signed_string) -> bool:
    """
    Verifies a claimed address by comparing it to its value that was
        recovered from a digitally signed string
    This is used for the user registration, where claimed_address
        represents a user address,
    and signed_string represents the digitally signed user address.
    :param claimed_address:
    :param signed_string:
    :return: bool
    """
    try:
        # use same hash function as in contract to hash the userAddress
        hashed_claim = w3.solidityKeccak(["address"], [claimed_address])
        address = _recover_address(hashed_claim, signed_string)
        return claimed_address == address
    except (InvalidAddress, ValueError) as e:
        log.info(f"check auth for address failed {e}")
        return False
```

Listing 4.19: The `check_auth_for_address` method

The recovery of the address from the hashed string using a digital signature string is shown in Listing 4.21, where the 'Web3.py' [57] method 'recoverHash' is used.

```
def _check_auth_for_nonce(nonce, signed_nonce, user_address) -> bool:
    """
    Verifies a claimed nonce by comparing it to its value that was
       recovered from a digitally signed string
    :param nonce:
    :param signed_nonce:
    :param user_address:
    :return: bool
    """
    try:
        # match passed nonce, userAddress pair with db entry first
        if not verify_nonce(nonce, user_address):
            return False
        # use the same hash function as in frontend to hash the nonce
        hashed_claim = w3.solidityKeccak(["bytes32"], [nonce])
        address = _recover_address(hashed_claim, signed_nonce)
        return address == user_address
    except (InvalidAddress, ValueError) as e:
        log.info(f"check auth for nonce failed {e}")
        return False
```

Listing 4.20: The `check_auth_for_nonce` method

```
def _recover_address(hashed_claim, signed_string):
    """
    Recovers the address from the hashed_claim using the signature.
    This is done to check whether the digital signature was issued by
       the claimed userAddress
    :param hashed_claim: str
    :param signed_string: str
    :return: address : str
    """
    return w3.eth.account.recoverHash(hashed_claim, signature=
       signed_string)
```

Listing 4.21: The `recover_address` method

Smart Contract Event Listening

While fundamentally different, the SC event listening has been set up similarly to the 'Observer' design pattern [55, 62]. As such, for each SC event, an observer instance is created. Listing 4.22 depicts the `DeployVNFObserver` class, which upon creation attaches itself to the list of observers and starts the SC event listening by creating an event filter and starting a new thread for this particular event. In the case of the `DeployVNFObserver` class, upon an event update, the creation of a VNF is delegated to the `vnf_service` class to deploy a VNF along with event's arguments (*cf.* Line `12`).

The event listening is shown in Listings 4.23 and 4.24, which were influenced by the official documentation of the 'Web3.py' library [44]. The idea here is to create a new thread for

```
1  class DeployVNFObserver(AbstractObserver, AbstractPolling):
2      """Observer for vnf deployment events"""
3
4      def __init__(self, observable, vnf_service, poll_interval=5):
5          super().__init__(poll_interval=poll_interval)
6          self.event = "DeployVNF"
7          self.vnf_service = vnf_service
8          observable.attach(self)
9
10     @log_event
11     def update(self, event, *args, **kwargs):
12         self.vnf_service.deploy_vnf(event.args)
```

Listing 4.22: The `DeployVNFObserver` class

each event filter, *cf.* Listing 4.23, to continuously poll new event entries, and call the update method of an observer in the case of a new event, *cf.* Line 10 in Listing 4.24.

```
1      def _evt_listen(self, observer):
2          """
3          create a contract event filter for an observer and set up a new
               thread to start event listening
4          """
5          event = observer.event
6          event_filter = self.contract.events[event].createFilter(
               fromBlock="latest")
7          worker = Thread(
8              target=self._event_loop, args=(event_filter, observer),
                   daemon=True
9          )
10         # store to join later
11         observer.worker = worker
12         worker.start()
```

Listing 4.23: The `_envt_listen` method

```
1      def _event_loop(self, event_filter, observer) -> None:
2          """
3          gets new events based on the type of event this thread is
               listening to
4          :param event_filter:
5          :param poll_interval: int
6          :return: None
7          """
8          while True:
9              for event in event_filter.get_new_entries():
10                 observer.update(event)
11             time.sleep(observer.poll_interval)
```

Listing 4.24: The `_event_loop` method

Smart Contract Reporting

The backend listens for SC events and handles them by initiating the appropriate actions. After completing the respective action, the backend reports to the SC whether the action has been performed successfully, sometimes including parameters. This sequence is displayed in greater detail in Section 4.3.6. Listings 4.25 and 4.26 show the exemplary VNF deployment-related functions, though the idea is the same for the other events.

Listing 4.25 depicts the `deploy_vnf` method, which is part of the `VNFService` class, which is called from the `DeployVNFObserver`, *cf.* Listing 4.22. On Line `12` the creation of the VNF is delegated to an 'nfv_client', which is passed to the `VNFService` class using dependency injection [60]. This ensures greater reusability, in the sense that multiple NFV frameworks can be used, and the class itself does not need to be concerned with the NFV framework and its details [60].

The actual reporting calls can be seen in Lines `18-19, 24-25`, with the difference between these two cases being in whether the VNF has been successfully deployed. In fact, if deployment of a VNF was erroneous, the error message from the NFV framework is stored in the database (*cf.* Line `27-31`), allowing the client to query the error details while keeping the details of the error message private to ensure that no accidental leakage of *e.g.,* VNF deployment details is possible.

```
1     def deploy_vnf(self, event_args_dict) -> None:
2         """
3         Deploy a vnf
4         :param event_args_dict : dict from the event args
5         """
6         creator_address, deployment_id, vnfd_id, parameters = itemgetter
              (
7             "creator", "deploymentId", "vnfdId", "parameters"
8         )(event_args_dict)
9         log.info(f"{creator_address}, {deployment_id}, {vnfd_id}, {
              parameters}")
10        try:
11
12            res, status_code = self.nfv_client.create_vnf(
13                parameters=parameters, vnfd_id=vnfd_id
14            )
15            success = status_code == 201
16            if not success:
17                raise AssertionError
18            smartContractService.service.report_vnf_deployment(
19                deployment_id, creator_address, success, res["id"]
20            )
21
22        except Exception as e:
23            log.info(f" deployVNF error {e}")
24            smartContractService.service.report_vnf_deployment(
25                deployment_id, creator_address, False, ""
26            )
27            errormsgService.service.store_errormsg(
28                address=creator_address,
```

```
29                  deployment_id=deployment_id,
30                  tacker_error=TackerErrorModel.from_dict(res.get("
                        TackerError")),
31              )
```

Listing 4.25: The `deploy_vnf` method

While the previous Listing 4.25 showed the caller of the reporting, the actual reporting function is shown in Listing 4.26. The `report_vnf_deployment` method calls the SC's `reportDeployment` function (*cf.* Line 30), however, the transaction needs first to be built (*cf.* Lines 18-25) and subsequently signed (*cf.* Line 26-28). Once the transaction has been sent, the transaction receipt is logged (*cf.* Line 31-32). Noteworthy is that the signing requires the actual transaction count that has been sent from the backend's account [44] (*cf.* Lines 15-17), also requiring the backend to have its own address.

```
1     def report_vnf_deployment(
2         self, deployment_id, creator_address, success, tacker_vnf_id
3     ):
4         """
5         Reports whether an attempt to create a VNF has been successful.
6         Calls the SC function reportDeployment.
7         :param deployment_id: int : SC internal identifier for the VNF
8         :param creator_address: str: address of the user whom the VNF
              belongs to
9         :param success: bool: signs whether the VNF has been
              successfully created
10        :param tacker_vnf_id: str: id of the newly created VNF, empty
              string if unsuccessful
11        :return:
12        """
13        try:
14
15            nonce = w3.eth.get_transaction_count(
16                SC_BACKEND_CONFIG["SC_BACKEND_ADDRESS"]
17            )
18            txn = self.contract.functions.reportDeployment(
19                deployment_id, creator_address, success, tacker_vnf_id
20            ).buildTransaction(
21                {
22                    "from": SC_BACKEND_CONFIG["SC_BACKEND_ADDRESS"],
23                    "nonce": nonce,
24                }
25            )
26            signed_txn = w3.eth.account.sign_transaction(
27                txn, private_key=SC_BACKEND_CONFIG["
                      SC_BACKEND_ADDRESS_PKEY"]
28            )
29
30            w3.eth.send_raw_transaction(signed_txn.rawTransaction)
31            tx_receipt = w3.toHex(w3.keccak(signed_txn.rawTransaction))
32            log.info(f" transaction receipt: {tx_receipt}")
33        except Exception as e:
34            log.info(f"report_vnf_deployment error {e}")
```

Listing 4.26: The `report_vnf_deployment` method

NFV Framework Connector

Since the backend acts as a facade for the NFV Framework, a `AbstractNFVFramework` class has been set up as a base class for the connection to NFV frameworks, with an excerpt from the class shown in Listing 4.27. This design allows for an easier implementation and connection to various concrete NFV frameworks.

```
1  class AbstractNFVFramework(ABC):
2      """
3      Abstract baseclass for a nfv framework with default implementation
           for header and requests
4      """
5
6      def __init__(self, token, base_url):
7          self._headers = None
8          self._token = token
9          self._base_url = base_url
10
11     @property
12     def headers(self):
13         return {"X-Auth-Token": self.token, "content-type": "Application
               /JSON"}
14
15     @headers.setter
16     def headers(self, headers):
17         self._headers = headers
18
19     ...
20
21     @abstractmethod
22     def _get_token(self):
23         """Get an auth token"""
24         pass
25
26     @abstractmethod
27     def get_vnfs(self):
28         """Get all vnfs"""
29         pass
30
31     @abstractmethod
32     def get_vnf(self, vnf_id):
33         """
34         Get a vnf by vnf_id
35         """
36         pass
37
38     ...
```

Listing 4.27: An excerpt of the `AbstractNFVFramework` class

Due to this project's choice of using 'Openstack Tacker' as an NFV framework, the class `Tacker` inherits from `AbstractNFVFramework`, and implements the missing and abstract methods. In Listing 4.28 its `create_vnf` method is shown. It receives from the SC event listener the required parameters for the NFV framework to deploy a VNF (*cf.* Line 8)

and posts these to 'Tacker', shown on Line 21. In the case of errors (*cf.* Lines 23-24), the error message from Tacker, and its status code are returned to the `VNFService`, as shown in Listing 4.25.

```
1     @get_token_if_401
2     def create_vnf(self, parameters, vnfd_id, *args, **kwargs):
3         """
4         Create a vnf with the given parameters in tacker.
5         :param parameters: str
6         :param vnfd_id: str
7         """
8         parameters = json.loads(parameters)
9         data = {
10            "vnf": {
11                "tenant_id": self._tenant_id,
12                "vnfd_id": vnfd_id,
13                "vim_id": self._vim_id,
14                "placement_attr": {"region_name": "RegionOne"},
15            }
16        }
17        data["vnf"]["attributes"] = parameters.get("attributes")
18        data["vnf"]["name"] = parameters.get("name")
19        data["vnf"]["description"] = parameters.get("description")
20
21        response = self._reqPOST("vnfs", data)
22        log.info(f"{response}")
23        if not response.json().get("vnf"):
24            return json.loads(response.text), response.status_code
25        return response.json().get("vnf"), response.status_code
```

Listing 4.28: The `create_vnf` method

4.3.4 Database

Concerning the database, MongoDB, a NoSQL document database, has been chosen for BCV, due to its ease of use and flexible data models [38], in conjunction with MongoEngine [39], to map the BCV backend's objects to the database. Therefore, three document schemata for the database have been defined in the BCV backend, as it is the only component in the system that directly interacts with the database. Furthermore, the database is deployed in a separate Docker [14] container.

User

Listing 4.29 illustrates the schema for registered users in the database. Only the user's addresses are essential since passwords are avoided using the authentication flow, *cf.* Section 4.2.4, though the registration date has been added as well.

```
class User(db.Document):
    """
    Stores registered Users related information
    """
    address = db.StringField(required=True, unique=True)
    registration_date = db.DateTimeField(default=datetime.datetime.now)
    meta = {"collection": "user"}
```

Listing 4.29: The document schema for the user

Nonce

Listing 4.30 depicts the schema for nonces, that are used for the issuance of tokens in the authentication flow, *cf.* Section 4.2.4. Note that both 'value' and 'address' have to be unique, implying that a user can only have one pending nonce during the authentication process, and the nonces themselves have to be unique. Furthermore, their issue date is stored, since during the authentication flow their age must be less than one day or they are invalid.

```
class Nonce(db.Document):
    """
    Used to issue tokens
    Nonces get deleted when
    - a token is issued
    - a new nonce is requested
    They are one day valid.
    """
    value = db.StringField(required=True, unique=True)
    address = db.StringField(required=True, unique=True)
    issue_date = db.DateTimeField(default=datetime.datetime.now)
    meta = {"collection": "nonce"}
```

Listing 4.30: The document schema for the nonce

Tacker Error

Listing 4.31 shows the document schema for Tacker's deployment errors. This schema has been added since invalid VNF configurations may be submitted when attempting to deploy a VNF, which results in errors and a deployment failure. Due to the VNF deployment process going through the SC, passing the error message through the SC could potentially leak sensitive information about the VNF or its deployer. Thus, to completely obviate this issue, potential VNF deployment errors are stored in the database. Upon receiving the deployment failure signal, a BCV client can query the stored error message through the BCV backend.

```
class TackerError(db.Document):
    """
    Used to store tacker deployment errors into the db, so that clients
        can query for the errors.
    As such, no private informations are leaked through the smart
        contract
    """
    address = db.StringField(required=True, unique=False)
    deployment_id = db.IntField(required=False, unique=True)
    vnf_id = db.StringField(required=False, unique=False)
    type = db.StringField(required=False, unique=False)
    message = db.StringField(required=False, unique=False)
    detail = db.StringField(required=False, unique=False)
    meta = {"collection": "tacker_error"}
```

Listing 4.31: The document schema for the tacker errors

4.3.5 NFV MANO

BCV uses OpenStack Tacker [41] (Wallaby release) as NFV framework, running inside a VM (Ubuntu 18.04.6 LTS [9]). This particular combination of operating system and Tacker release has proven to be the most stable to operate in the setting of BCV. Tacker provides a REST API that is consumed by the BCV backend. While Tacker provides multiple API versions, version 1.0 appeared to be the most stable, which is why it was selected for connecting the BCV backend to Tacker. The backend uses a technical user to authenticate with Tacker's API. This user is configured through Tacker, which occurs during setup. Note that the BCV backend is the only component connected to OpenStack Tacker, thereby providing a layer of abstraction for both frontend and SC. A more detailed overview of OpenStack Tacker is provided in Section 2.1.2.

4.3.6 Events

BCV uses a series of events to implement BC signaling. Figure 4.7 depicts BCV's events in their respective contexts, showing the order of execution as well as source and destination. While, in principle, any actor could subscribe to any of the listed events, the events can be categorized into two distinct sets: The first set of events (*i.e.,* `Register`, `DeployVNF`, `DeleteVNF`, `Unregister`) are destined for the backend, which utilizes an event listener to subscribe to these events. The other set (*i.e.,* `RegistrationStatus`, `DeploymentStatus`, `DeletionStatus`, `UnregistrationStatus`) consists of status events destined for the frontend. They carry the feedback of the backend to the frontend. The following paragraph delves into the properties of each event used in BCV:

- The `Register` event is fired when a user initiates the user registration via SC. It carries the user's BC wallet address and the signature of the user's address.

- The `RegistrationStatus` contains the feedback of the backend concerning the success of the user's registration request. The user is identified by her BC wallet address.

- `DeployVNF` contains the data for instantiating a new VNF. This data includes the user who triggers the deployment, but also the contract-managed `deploymentId`. Furthermore, it also contains the identifier of the VNF descriptor for creating the VNF. Finally, the event contains the parameter values as required by the TOSCA template. When the event is processed by the backend, a new VNF is created on OpenStack Tacker.

- `DeploymentStatus` is the feedback event destined for the frontend after the backend has executed a deployment. It signals either success or failure for the instantiation of a particular VNF. In case of success, it also contains the VNF identifier of the deployed VNF.

- The `DeleteVNF` event initiates a VNF deletion request on the backend. It references the VNF to be deleted by its `deploymentId`, as well as its VNF identifier.

- The `DeletionStatus` event notifies the frontend about the status of a VNF delete operation.

- `Unregister` indicates the user's request to deregister from BCV. For this purpose, it contains the user's BC wallet address as the only parameter, as this is the only information required by the backend to perform this operation.

- `UnregistrationStatus` is the feedback event allotted to the frontend, indicating whether a user's deregistration request was successfully fulfilled.

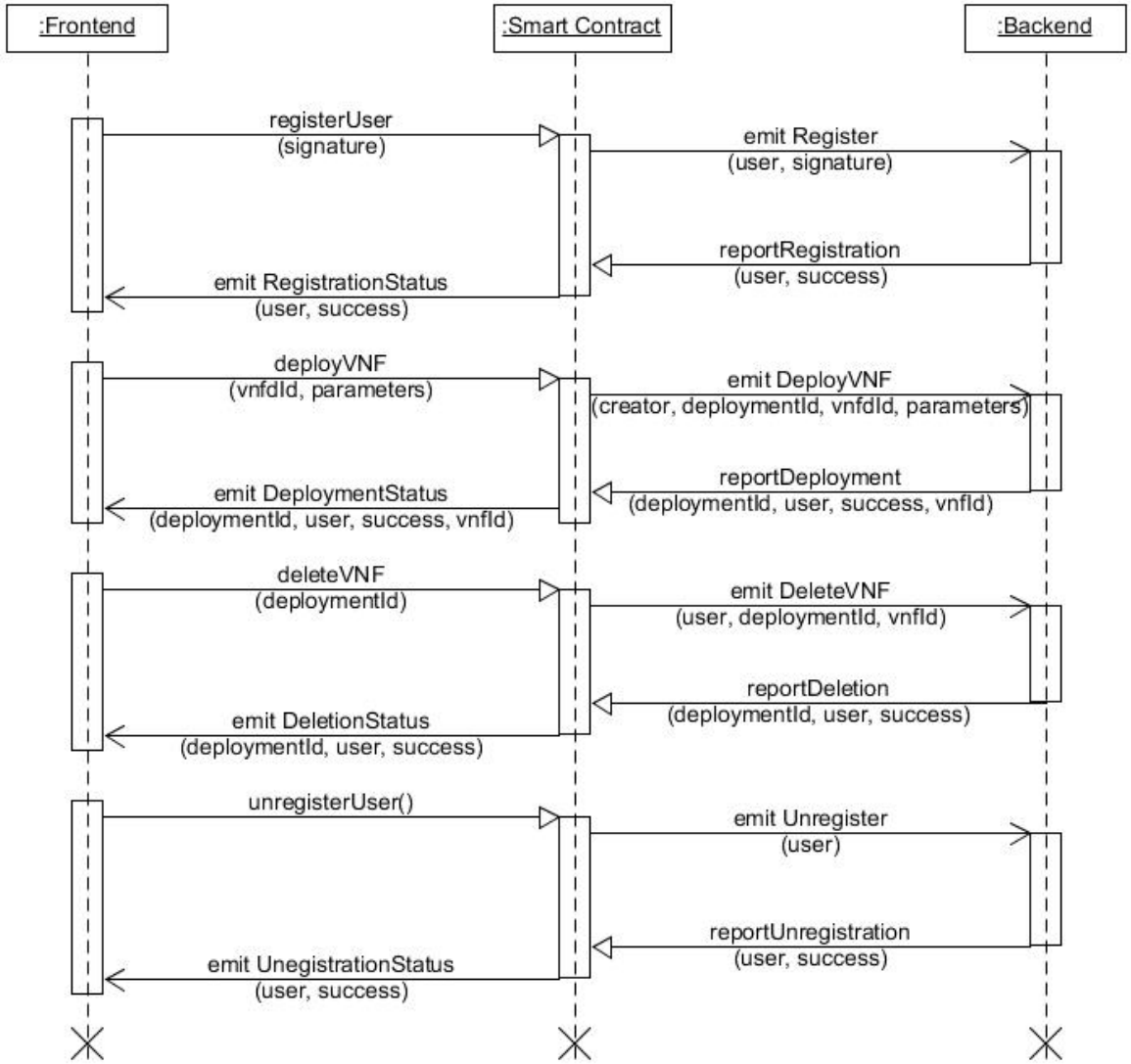

Figure 4.7: Sequence of events in blockchain-v

Chapter 5

Evaluation

This chapter evaluates different aspects of BCV. First, a cost analysis is presented in Section 5.1, which details the cost of operating BCV on the Ethereum BC. Section 5.2 shows the results of a performance analysis, where BCV was deployed in two different environments, namely Ganache [13] and Ethereum Ropsten [15]. Finally, section 5.3 discusses the evaluation findings.

5.1 Cost Analysis

In BCV, the SC represents the primary cost driver for operating the system. This section focuses on the cost components of the SC, and the cost of using BCV on a use case basis. The goal is to show the operational costs for both the user and the provider when interacting with BCV's SC. From a methodological standpoint, the cost numbers are calculated as follows: Using the Remix IDE [25], all functions of BCV's SC have been called. Remix allows inspecting the transaction receipts for each function call, which includes the consumed *gas* for each transaction. Thus, to obtain representative gas consumption numbers, each function has been called multiple times ($n = 10$) to derive the mean gas cost for calling each function. Determining the mean is necessary, as the gas cost varies for operations that search for particular VNFs. For this purpose, a loop is used to find the respective VNF inside the array of all VNFs. Thus, the number of loop iterations required to find the VNF depends on the total number of VNFs.

Table 5.1 shows the results of the analysis, describing the costs of calling each SC function in terms of Gas (rounded to closest integer), Ether (ETH) (calculated according to [56], rounded to 7 decimal positions), and US Dollar (USD) (rounded to 2 decimal positions). For this purpose, the Gas price (154.95 gwei [65]) as well as the ETH price (2427.4406 USD [20]) have been considered at the time of writing (*i.e.,* January 27, 2022). Due to the high conversion rate between ETH and USD, the effective cost of calling an SC function is significant. For example, initiating a VNF deployment requires the user to spend 64.61 USD, while a simple user registration costs the user 10.59 USD. A notable exception to the high prices is the `getVnfs` function. Calling it results in no cost at all, as it is implemented as a read-only function, which can be executed free of charge [12].

Table 5.1: Cost of contract function calls as of 27.01.2022.

Function	Gas	Cost (ETH)	Cost (USD)
`registerBackend`	28'405	0.0044014	10.68
`registerUser`	28'166	0.0043643	10.59
`reportRegistration`	43'780	0.0067837	16.47
`unregisterUser`	22'549	0.0034940	8.48
`reportUnregistration`	30'679	0.0047537	11.54
`deployVNF`	171'779	0.0266172	64.61
`deleteVNF`	42'153	0.0065316	15.86
`reportDeletion`	125'425	0.0194346	47.18
`reportDeployment`	86'145	0.0133482	32.40
`getVnfs`	0	0	0

While the cost per function call already indicates the cost related to running BCV, analyzing the cost on a use case (*i.e.,* End-to-End (E2E)) basis provides a more realistic view. Table 5.2 lists the cost of executing E2E processes within BCV. Each use case consists of multiple primitives (*i.e.,* function calls as listed in Table 5.1), contributing to the overall cost. In analogy to the previous example, the overall price of deploying a VNF sums up to 97.01 USD, where the user registration results in a charge of 27.06 USD. Again, the process of listing VNFs is free, as this process only uses the `getVnfs` method of the SC, while the rest of the data is retrieved from the Tacker NFV framework.

Table 5.2: Gas consumption and cost for one execution of the listed use cases.

Use Case	Function Calls	Cost Percentage	Gas	Cost (ETH)	Cost (USD)
Register User	`registerUser` `reportRegistration`	39.15% 60.85%	71'946	0.0111480	27.06
Unregister User	`unregisterUser` `reportUnregistration`	42.36% 57.64%	53'228	0.0082477	20.02
Deploy VNF	`deployVNF` `reportDeployment`	66.60% 33.40%	257'924	0.0399653	97.01
Delete VNF	`deleteVNF` `reportDeletion`	25.15% 74.85%	167'578	0.0259662	63.03
List VNFs	`getVNFs`	-	0	0	0

However, it is crucial to not only analyze the cost of executing a particular use case but also to verify the cost division between user and provider in each scenario. Figure 5.1 depicts the cost ratio between the user and the provider of the backend for executing a particular use case. The x-axis represents the accumulated gas cost, and the y-axis delineates the use case. For the deployment of a VNF, the user pays the major share of the cost (66.6%), whereas deleting a VNF is more expensive for the provider (74.85%). For user registration and deregistration, the provider pays for most of the cost, namely 60.85% for registration and 57.64% for deregistration, respectively. Simply listing the VNFs is free for both provider and user.

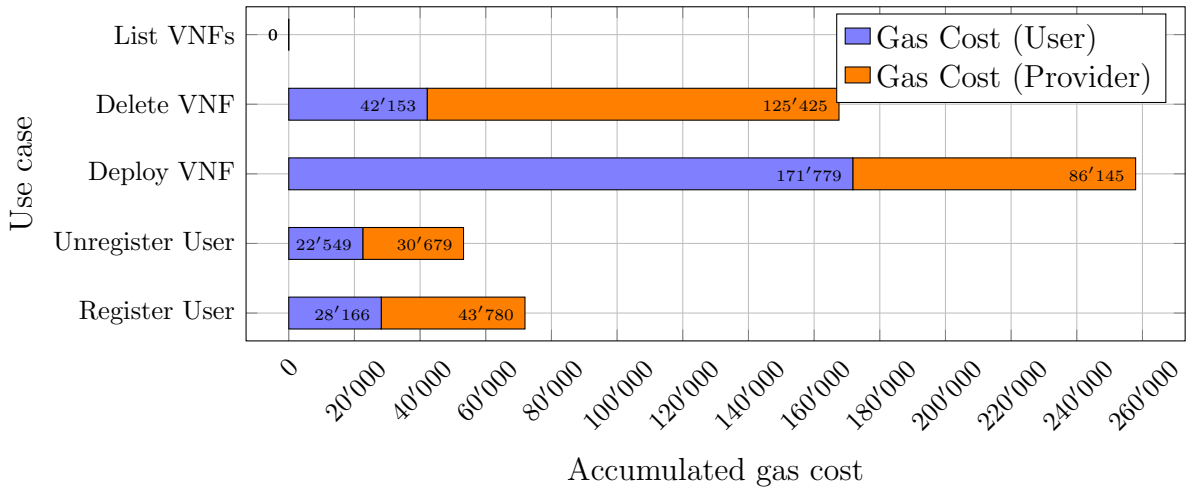

Figure 5.1: Gas consumption per use case, divided into costs for the user and costs for the provider.

5.2 Performance Analysis

Due to the decentralized nature of BCV, its performance depends on multiple factors (*e.g.,* network latency, BC performance, and available hardware resources). This section presents the results of the performance analysis of BCV. It aims at determining the influence of each BCV component on the overall performance. The performance analysis has been conducted on a Dell XPS 8500 (Intel Core i7-3770 CPU @ 3.40GHz x 8, 32 GB of system memory) to run OpenStack Tacker, the backend, and the frontend. OpenStack Tacker ran inside an Ubuntu 18.04.6 LTS virtual machine with 24 GB of system memory assigned. BCV's frontend and backend (including the database) run inside Docker containers.

For the analysis, two scenarios have been considered: The first scenario involves deploying the SC to an Ethereum development network on Ganache [13]. This configuration represents the best case possible in terms of performance from the BC's side. For this reason, Ganache's "automining" feature was enabled. The second scenario represents a more realistic setup by deploying the SC to the Ethereum Ropsten [15] test network, which has similar properties to Ethereum Mainnet. An Infura [30] endpoint was registered to access the Ropsten network. Note that the results might differ in a scenario where a local `geth` node is used instead of Infura, potentially reducing network latency when accessing the BC.

In the Ganache scenario, 10 rounds of measurements for each action (deploying and deleting a VNF) were performed. Each round consisted of a set of 6 timestamps. In the Ropsten scenario, the number of rounds was increased to 20 to address the emerging variance. This variance originated from two factors: the block time of Ropsten and the timing of sending a transaction (*i.e.,* at which point in time between two blocks). Note that the VNF was based on an identical VNF descriptor in all cases, without any parametrization or additional configuration objects attached. The timestamps collected per round (*cf.* Figure 5.2) are as follows:

- `t0`**: Frontend initializes call**: The moment at which the user initiates the contract call by pressing the 'Deploy VNF' button (*cf.* Figure 4.5) respectively the 'Delete VNF' button. Note that the timespan from this stamp to the next one is influenced by user actions, as the transactions have to be confirmed manually in Metamask.

- `t1`**: Block timestamp request processed**: The timestamp of the block in which the transaction responsible for processing the request and emitting the 'DeployVNF' respectively 'DeleteVNF' event was mined.

- `t2`**: Backend received request**: Marks the time when the backend controller receives the emitted event.

- `t3`**: Backend initializes report**: Denotes the time in the backend before the report VNF deployment or VNF deletion transaction is sent.

- `t4`**: Block timestamp report processed**: The timestamp of the block in which the transaction responsible for processing the backend report and emitting the corresponding events is mined.

- `t5`**: Frontend notified by contract**: Marks the moment at which the event listeners that target the reporting events emitted by the SC are notified.

The timestamps were read from console logging output for those originating from the frontend or backend, respectively. The block timestamps were obtained from the Etherscan [17] website, using the transaction hashes from the transaction receipts, which identify the block in which the transactions were ultimately processed. In the local scenario using Ganache, block times were read from the Graphical User Interface (GUI) of the Ganache desktop client.

Initially, the goal was to create a complete timeline using all six timestamps. However, the timestamps were collected from three different systems, hence the respective system clocks were not guaranteed to be consistent with each other. This issue became apparent when looking at the timestamps as if they were from a single system, with multiple occurrences of *e.g.,* the backend timestamp for receiving a request from the contract (`t2`) being chronologically earlier than the block time of the block responsible for processing the request (`t3`). This inconsistency of timestamps resulted in a faulty timeline.

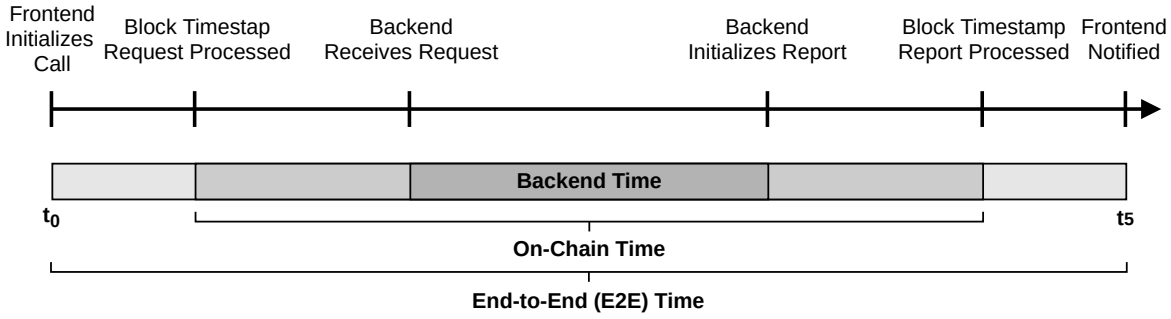

Figure 5.2: Performance Evaluation Scheme

Figure 5.2 illustrates the pursued scheme. The collection of six timestamps per round was maintained, but only the timestamps originating from the same system are now in fixed relation to each other. In other words, the timestamps indicate exactly how long the procedure took on each of the three systems (frontend, backend, blockchain) independently. This results in the three timespans shown in Figure 5.2, with the innermost (`t2` to `t3`) span representing the total time elapsed on the backend, the second (`t1` to `t4`) span representing the time it took for the action to be completely processed on the BC, and the outermost (`t0` to `t5`) span representing the total E2E time elapsed.

As for where exactly the timespan of a procedure on one system (*e.g.,* the time elapsed on the backend) is located within the timespan of the encompassing procedure on another system (*e.g.,* time spent on the chain), the evaluation cannot answer. All that can be stated is that it must be somewhere within the bounds of the timespan of the encompassing procedure, as the events marked by the timestamps are causally linked.

The Figures 5.3, 5.4, and 5.5 use the same four scenarios for the analysis: VNF deployment on Ganache (blue), VNF deployment on Ropsten (red), VNF deletion on Ganache (yellow), and VNF deletion on Ropsten (green). In all of these Figures, the x-axis denotes the scenario, whereas the y-axis shows the time elapsed (*i.e.,* duration) in seconds.

Figure 5.3 depicts time elapsed on an E2E basis, which conforms to the interval `t0` to `t5` in Figure 5.2. Figure 5.5 is in line with the interval `t1` to `t4` in Figure 5.2, which represents the time until each scenario is completely processed on the BC. Finally, Figure 5.4 corresponds to the interval `t2` to `t3` in Figure 5.2, illustrating the time elapsed on the backend for each of the selected scenarios.

5.3 Discussion

The high cost of operation that comes with the use of the SC shows the significance of BC choice for such a system: The selection process [54] has to be performed with great diligence and scrutiny, gauging between the impact of choosing a particular BC, as well as its consensus algorithm, as these two factors influence operational costs substantially. However, for this work, the cost is not the primary concern as the prototype was implemented to show the technical feasibility of implementing such as system. For real-world operation, the underlying BC would have to be replaced with a more cost-efficient implementation. A possible option would be to use Ethereum with a different consensus algorithm, such as Proof-of-Stake (PoS) or Proof-of-Authority (PoA), which could reduce the overall costs [53].

The performance analysis evaluation showed that for the E2E evaluations, similar results were reached in the deployment of VNF and deletion of VNF in the Ropsten Testnet. In fact, the resulting median values are very close, *cf.* Figure 5.3. Nevertheless, the E2E Ropsten Testnet data shows a high variance since the values are spread out from the mean, and high outliers in the data can be observed. Since the backend results showed stable performance results (*cf.* Figure 5.4) it is noticeable that the high outliers from the E2E test directly result from the differences in chain timestamps (*cf.* Figure 5.5). This finding

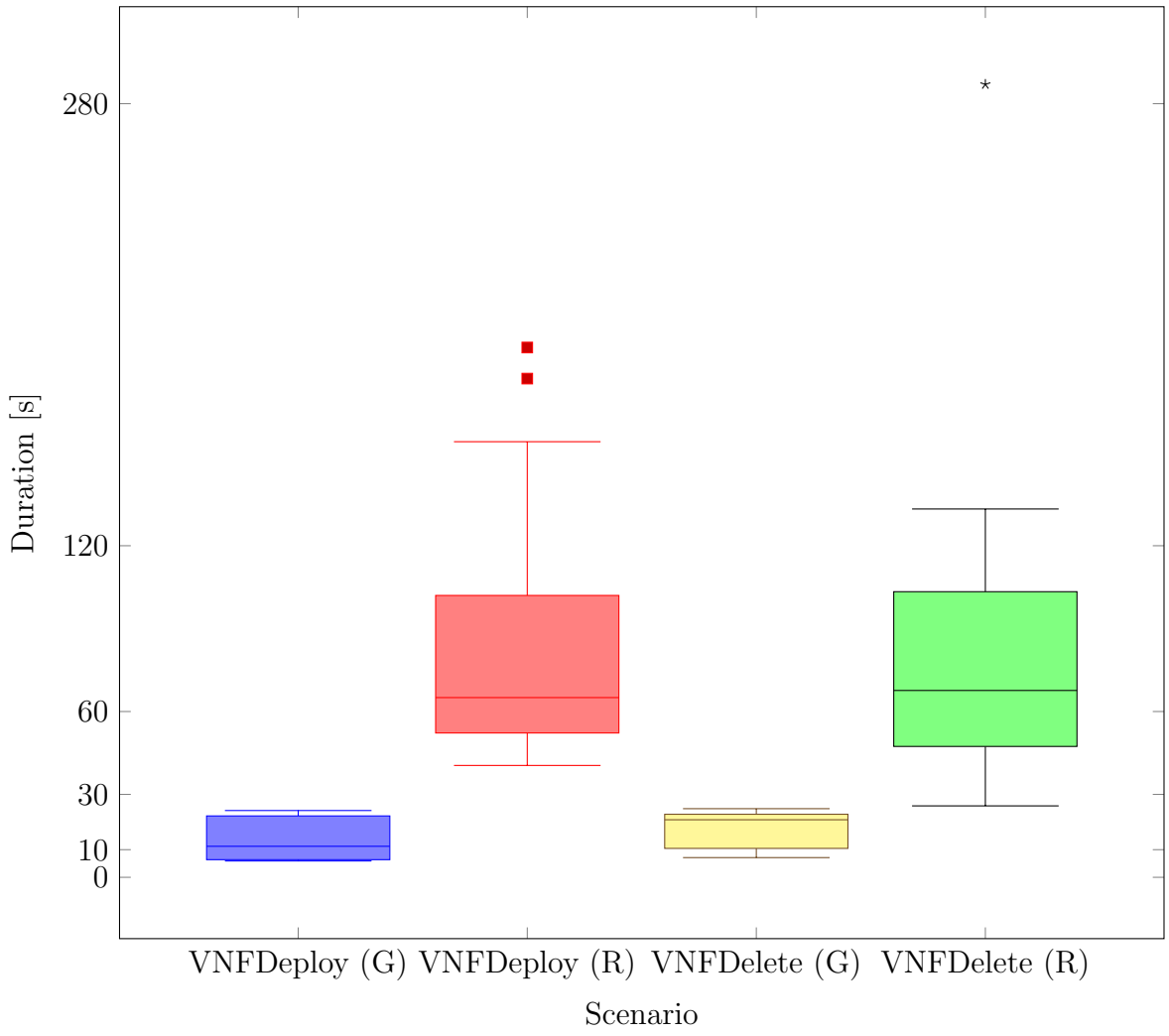

Figure 5.3: E2E time elapsed, G = Ganache, R = Ropsten

implies that there is no processing time guarantee regarding the BC. In fact, in most cases, the processing was done on two subsequent blocks. In other instances, this was not the case, which resulted in those outliers. This wide range of values follows from an inherently random factor of the BC since it cannot be controlled in which block a transaction will ultimately emerge. Furthermore, the human factor in the evaluation also introduces noise in the collected data, though it can be argued that due to the given duration ranges, small differences in the order of a second would not influence the expressiveness of the results.

Additionally, as depicted in Figure 5.4, it is apparent that the backend was faster in the local Ganache case than in the Ropsten Testnet case. Despite applying the same test procedure in both cases, the results showed a difference in execution time of a few hundred milliseconds. This difference can potentially be explained by the fact that the SC event listener, which is run in background threads, is more computationally expensive in the Ropsten Testnet scenario than in the local case since the events have to be continuously polled from the chain, due to the use of the web3 Python library, resulting in those small differences.

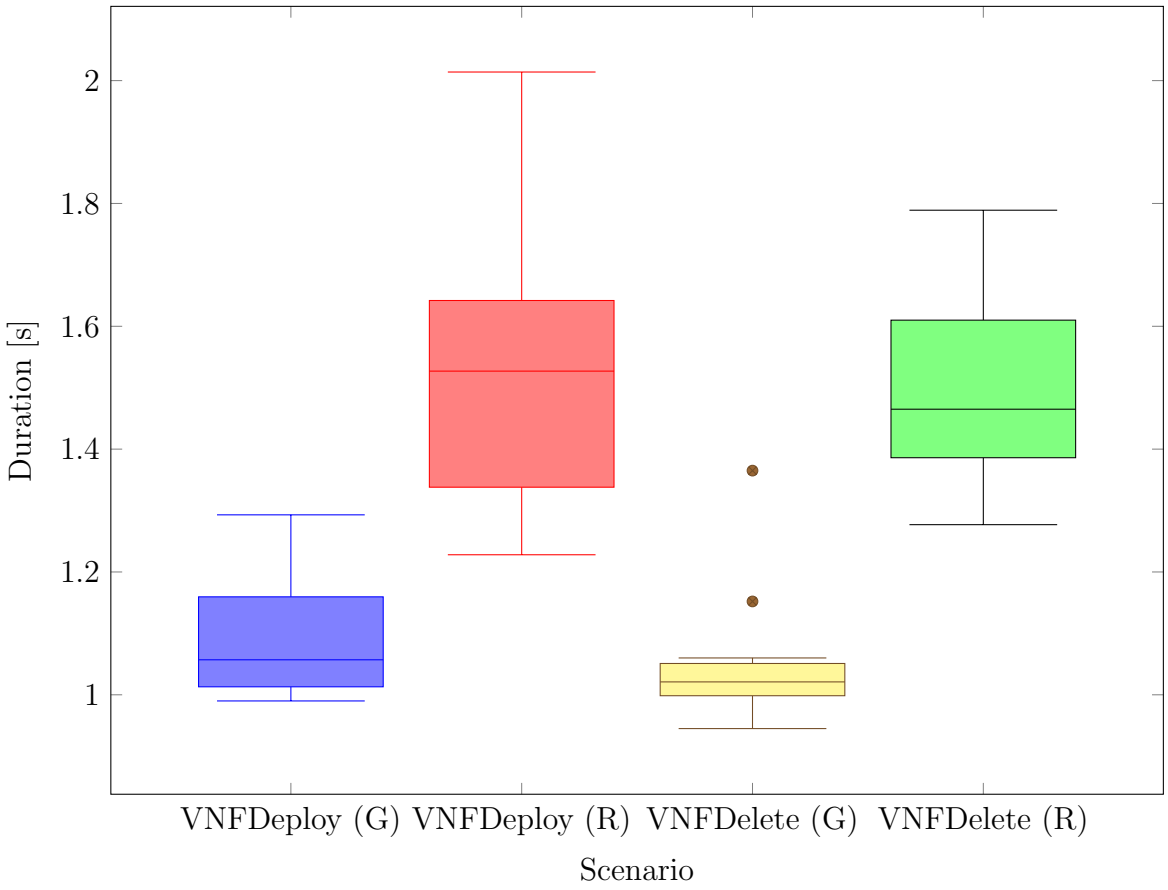

Figure 5.4: Time elapsed on backend, G = Ganache, R = Ropsten

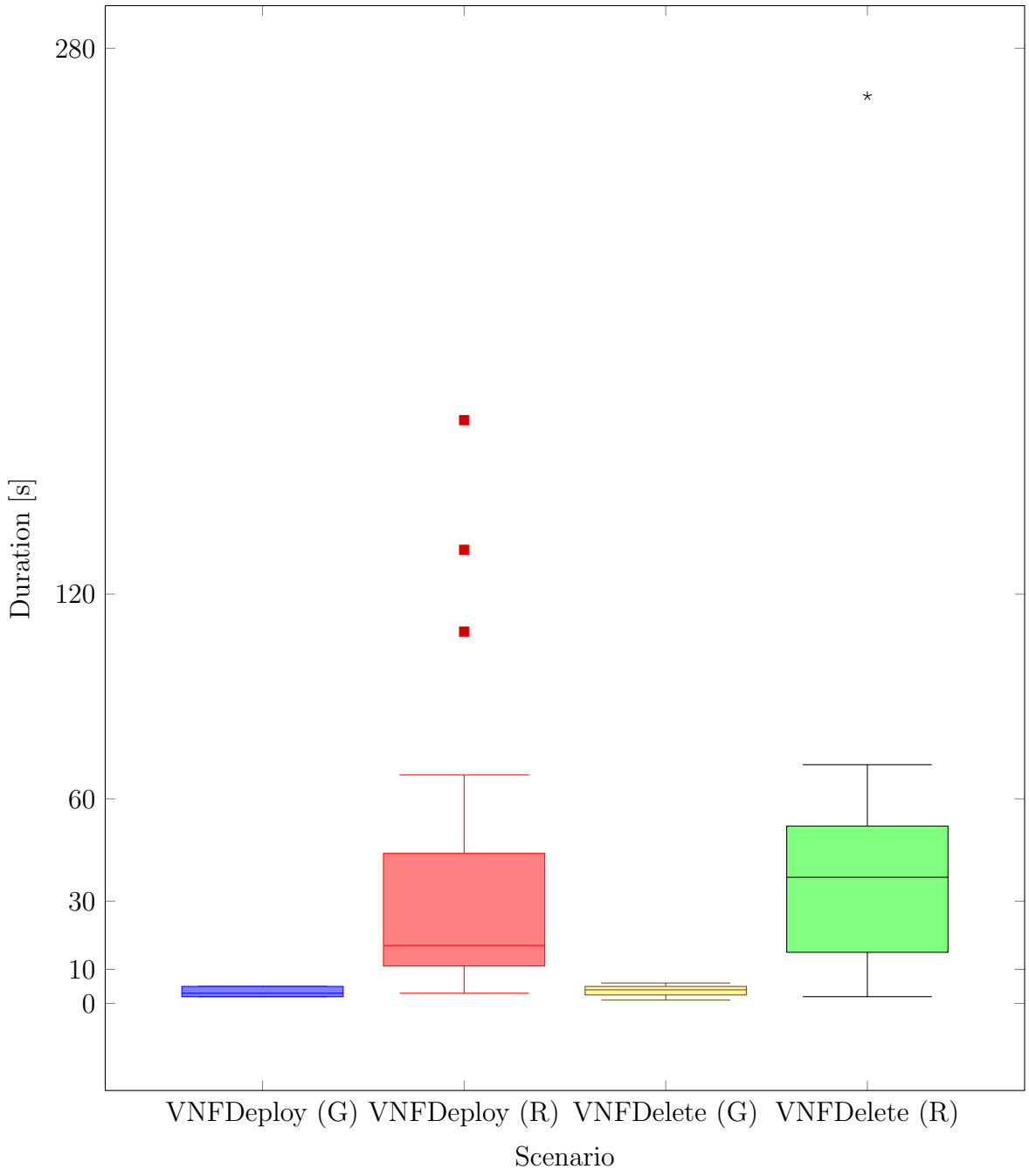

Figure 5.5: Time until completely processed on chain, G = Ganache, R = Ropsten

Chapter 6

Summary, Conclusions, and Future Work

This chapter concludes this work with a summary, including the main findings and conclusions reached. Furthermore, potential areas for future work are highlighted based on the insights gained during the development of this work.

6.1 Summary and Conclusions

The primary goal of this work was to develop the prototype of a system for deploying VNFs using the BC as an *event bus* to determine the feasibility of such an approach. For this purpose, the basics of NFVs, VNF frameworks, BC, and BC-based SCs have been elaborated. Subsequently, existing systems in the area of BC and NFV have been investigated, indicating that the concept of using BC signaling to deploy VNFs automatically had not been applied before. In that sense, this work presents a novel approach for automatically deploying VNFs via BC signaling. BCV was designed to be open and generic, providing extensibility for future requirements and features.

This work proved that the concept of using BC signaling to deploy VNFs is feasible. However, one caveat proved to be the cost involved in running BCV on a public permissionless BC (*i.e.,* Ethereum). Due to the high conversion rate between ETH and fiat currency at the time of writing, BCV would have been very expensive and unpredictable in terms of cost when run in a real-world setting. Thus, to achieve better cost efficiency, it might be more suitable to deploy BCV's SC onto a BC with a different consensus algorithm such as PoA or PoS. Another caveat regarding the BCV prototype is security. The authentication protocol used to authenticate users inside the backend only serves as a placeholder. In a real-world scenario, the protocol at hand should be replaced with a standardized authentication protocol (*e.g.,* OpenID Connect [21]). However, integrating a standardized authentication protocol was not within the defined scope of this work. In terms of performance, BCV's execution type deploying and deleting VNFs primarily depends on the underlying BC. Both frontend and backend did not cause harmful effects on the overall performance in both evaluated scenarios (*i.e.,* Ganache, Ropsten).

6.2 Future Work

While BCV proves that the concept of using BC signaling to deploy VNFs works, there is potential for extending the prototype, including integration with other systems, but also achieving higher scalability, security, and performance.

One task could be to connect BloSS [49] to the backend and the SC of BCV. By doing this, BloSS could react to cyber-attacks by deploying a VNF using BCV (*e.g.,* a firewall) to defend against the attack.

Another possible integration could be registering BCV as a VNF provider in BRAIN [22]. This integration would implicate the design and implementation of the economic mechanisms to compete in BRAIN's reverse auctions. If BCV was to win a particular auction, BRAIN could signal the deployment of a VNF via BCV's SC directly and then forward the details of the created VNF to the user.

With this prototype, the operating costs (*cf.* Section 5.1) were found to be high. Future work could involve adapting the SC to be deployed on another BC with a more suitable consensus algorithm. This work leveraged Ethereum, which uses a PoW consensus algorithm. However, there are more efficient consensus algorithms available, which could enable large cost-savings as well as performance improvements.

Another task could be to extend the SC to support multiple backends, which would allow for better horizontal scalability. Furthermore, extensions to connect other NFV frameworks to the backend would be beneficial. The backend is already built with extensibility in mind. Thus, implementing connectors for other NFV frameworks could be a feasible next step.

In terms of security, this prototype uses its own authentication protocol. However, in the scope of this work, this mechanism is only considered a placeholder. A standardized authentication protocol (*e.g.,* OpenID Connect [21]) could be integrated. Additionally, for BCV to be used in a real-world environment, it would have to undergo a security analysis followed by a hardening process.

Bibliography

[1] A ConsenSys Formation. Metamask - A crypto wallet & gateway to blockchain apps. `https://metamask.io/`, Last visit February 20, 2022.

[2] Igor D. Alvarenga, Gabriel A.F. Rebello, and Otto Carlos M.B. Duarte. Securing configuration management and migration of virtual network functions using blockchain. In *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9. IEEE, 2018.

[3] Ahmed M. Alwakeel, Abdulrahman K. Alnaim, and Eduardo B. Fernandez. A Pattern for a Virtual Network Function (VNF). In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, pages 1–7, 2019.

[4] Ahmed M. Alwakeel, Abdulrahman K. Alnaim, and Eduardo B. Fernandez. Toward a Reference Architecture for NFV. In *2019 2nd International Conference on Computer Applications & Information Security (ICCAIS)*, pages 1–6. IEEE, 2019.

[5] Md. Faizul Bari, Shihabur Rahman Chowdhury, Reaz Ahmed, and Raouf Boutaba. On orchestrating virtual network functions. In *2015 11th International Conference on Network and Service Management (CNSM)*, pages 50–56, 2015.

[6] Lucas Bondan, Muriel F. Franco, Leonardo Marcuzzo, Giovanni Venancio, Ricardo L. Santos, Ricardo J. Pfitscher, Eder J. Scheid, Burkhard Stiller, Filip De Turck, Elias P. Duarte, Alberto E. Schaeffer-Filho, Carlos R. P. dos Santos, and Lisandro Z. Granville. FENDE: Marketplace-based distribution, execution, and life cycle management of VNFs. *IEEE Communications Magazine*, 57(1):13–19, 2019.

[7] Nikola Bozic, Guy Pujolle, and Stefano Secci. Securing virtual machine orchestration with blockchains. In *2017 1st Cyber Security in Networking Conference (CSNet)*, pages 1–8, 2017.

[8] Vitalik Buterin. Ethereum Whitepaper: A Next-Generation Smart Contract and Decentralized Application Platform, 2021.

[9] Canonical Ltd. Ubuntu 18.04.6 LTS (Bionic Beaver). `https://releases.ubuntu.com/18.04/`, Last visit February 11, 2022.

[10] Jinlin Chen, Yiren Chen, Shi-Chun Tsai, and Yi-Bing Lin. Implementing NFV system with OpenStack. In *2017 IEEE Conference on Dependable and Secure Computing*, pages 188–194. IEEE, 2017.

[11] Margaret Chiosi, Don Clarke, Peter Willis, Andy Reid, James Feger, Michael Bugenhagen, Waqar Khan, Michael Fargano, Chunfeng Cui, Hui Deng, Javier Benitez, Uwe Michel, Herbert Damker, Kenichi Ogaki, Tetsuro Matsuzaki, Masaki Fukui, Katsuhiro Shimano, Dominique Delisle, Quentin Loudier, Christos Kolias, Ivano Guardini, Elena Demaria, Roberto Minerva, Antonio Manzalini, Diego Lopez, Francisco Javier Ramon Salguero, Frank Ruhl, and Prodip Sen. Network Functions Virtualisation An Introduction, Benefits, Enablers, Challenges & Call for Action.

[12] ConsensSys Software Inc. Interacting with your contracts. `https://trufflesuite.com/docs/truffle/getting-started/interacting-with-your-contracts.html`, Last visit January 28, 2022.

[13] ConsenSys Software Inc. Ganache - ONE CLICK BLOCKCHAIN. `https://trufflesuite.com/ganache/`, Last visit February 7, 2022.

[14] Docker. Docker. `https://www.docker.com/`, Last visit February 14, 2022.

[15] Ethereum.org. Networks. `https://ethereum.org/en/developers/docs/networks/#ropsten`, Last visit February 19, 2022.

[16] Ethereum.Stackexchange - user 'Afr'. What is an ABI and why is it needed to interact with contracts? `https://ethereum.stackexchange.com/a/235`, Last visit February 24, 2022.

[17] Etherscan. Ropsten Testnet Explorer. `https://ropsten.etherscan.io/`, Last visit February 19, 2022.

[18] ETSI Industry Specification Group (ISG). Network Functions Virtualisation (NFV); Management and Orchestration, 2016. `https://www.etsi.org/deliver/etsi_gs/NFV-IFA/001_099/009/01.01.01_60/gs_NFV-IFA009v010101p.pdf`, Last visit September 23, 2021.

[19] Gabriel Antonio F. Rebello, Gustavo F. Camilo, Leonardo G. C. Silva, Lucas C. B. Guimarães, Lucas Airam C. de Souza, Igor D. Alvarenga, and Otto Carlos M. B. Duarte. Providing a Sliced, Secure, and Isolated Software Infrastructure of Virtual Functions Through Blockchain Technology. In *2019 IEEE 20th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–6, 2019.

[20] Finanzen.ch. Ethereum - US-Dollar Währungsrechner. `https://www.finanzen.ch/waehrungsrechner/ethereum_us-dollar#origin-currency`, Last visit January 28, 2022.

[21] OpenID Foundation. Welcome to OpenID connect. `https://openid.net/connect/`, Last visit January 31, 2022.

[22] Muriel Figueredo Franco, Eder John Scheid, Lisandro Zambenedetti Granville, and Burkhard Stiller. BRAIN: Blockchain-based reverse auction for infrastructure supply in virtual network functions-as-a-service. In *2019 IFIP Networking Conference (IFIP Networking)*, pages 1–9. IEEE, 2019.

[23] Github Chainsafe/web3.js. web3.js - Ethereum JavaScript API. `https://github.com/ChainSafe/web3.js`, Last visit February 20, 2022.

[24] Github Chainsafe/web3.js. web3.js - Ethereum JavaScript API Documentation. `https://web3js.readthedocs.io/en/v1.7.0/`, Last visit February 24, 2022.

[25] Github ethereum/remix-project. Remix - Ethereum IDE. `https://remix.ethereum.org`, Last visit January 28, 2022.

[26] Github ethereum/solidity. Contract ABI Specification. `https://docs.soliditylang.org/en/v0.8.12/abi-spec.html`, Last visit February 24, 2022.

[27] Github ethereum/solidity. Solidity, 2021. `https://github.com/ethereum/solidity`, Last visit October 30, 2021.

[28] Github ethereum/solidity. Solidity - Solidity 0.8.9 documentation, 2021. `https://docs.soliditylang.org/en/v0.8.9/#`, Last visit October 30, 2021.

[29] Github vuejs/vuex. What is Vuex? `https://vuex.vuejs.org/`, Last visit February 20, 2022.

[30] Infura Inc. The World's Most Powerful Blockchain Development Suite. `https://infura.io/`, Last visit February 19, 2022.

[31] Ales Komarek, Jakub Pavlik, Lubos Mercl, and Vladimir Sobeslav. Vnf orchestration and modeling with etsi mano compliant frameworks. In *Internet of Things, Smart Spaces, and Next Generation Networks and Systems*, pages 121–131. Springer, 2017.

[32] Lusani Mamushiane, Albert A. Lysko, Tariro Mukute, Joyce Mwangama, and Zaaid Du Toit. Overview of 9 open-source resource orchestrating ETSI MANO compliant implementations: A brief survey. In *2019 IEEE 2nd Wireless Africa Conference (WAC)*, pages 1–7. IEEE, 2019.

[33] Marianna Belotti and Nikola Božić and Guy Pujolle and Stefano Secci. A Vademecum on Blockchain Technologies: When, Which and How. *IEEE Communications Surveys and Tutorials*, 21(4):3796–3838, July 2019.

[34] Ahmed M. Medhat, Tarik Taleb, Asma Elmangoush, Giuseppe A. Carella, Stefan Covaci, and Thomas Magedanz. Service Function Chaining in Next Generation Networks: State of the Art and Research Challenges. *IEEE Communications Magazine*, 55(2):216–223, 2017.

[35] Greg Michaelson. Programming Paradigms, Turing Completeness and Computational Thinking. *arXiv preprint arXiv:2002.06178*, 2020.

[36] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, and Raouf Boutaba. Network Function Virtualization: State-of-the-Art and Research Challenges. *IEEE Communications Surveys Tutorials*, 18(1):236–262, 2016.

[37] Raaj Anand Mishra, Anshuman Kalla, Kaustubh Shukla, Avishek Nag, and Madhusanka Liyanage. B-VNF: Blockchain-enhanced Architecture for VNF Orchestration in MEC-5G Networks. In *2020 IEEE 3rd 5G World Forum (5GWF)*, pages 229–234. IEEE, 2020.

[38] MongoDB, Inc. MongoDB. https://www.mongodb.com/, Last visit February 14, 2022.

[39] MongoEngine. MongoEngine. http://mongoengine.org/, Last visit February 14, 2022.

[40] OpenAPI-Generator Contributors. OpenAPI Generator. https://openapi-generator.tech/, Last visit February 24, 2022.

[41] OpenStack. Tacker Documentation, 2021. https://docs.openstack.org/tacker, Last visit September 25, 2021.

[42] OpenStack. Tacker Wiki, 2021. https://wiki.openstack.org/wiki/Tacker, Last visit September 25, 2021.

[43] Pallets. Flask, 2022. https://flask.palletsprojects.com/en/2.0.x/, Last visit March 8, 2022.

[44] Piper Merriam, Jason Carver. Web3.py 5.28.0 documentation, 2018. https://web3py.readthedocs.io/en/stable/index.html#, Last visit March 10, 2022.

[45] Python Software Foundation. Python. https://www.python.org/, Last visit February 24, 2022.

[46] Python Software Foundation. UUID. https://docs.python.org/3/library/uuid.html, Last visit February 24, 2022.

[47] Paul Quinn and Jim Guichard. Service Function Chaining: Creating a Service Plane via Network Service Headers. *Computer*, 47(11):38–44, 2014.

[48] Gabriel Antonio F. Rebello, Igor D. Alvarenga, Igor J. Sanz, and Otto Carlos M. B. Duarte. BSec-NFVO: A blockchain-based security for network function virtualization orchestration. In *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, pages 1–6, 2019.

[49] Bruno Rodrigues, Thomas Bocek, and Burkhard Stiller. Enabling a cooperative, multi-domain DDoS defense by a blockchain signaling system (BloSS). *Semantic Scholar*, 2017.

[50] Taylor Rolfe. Turing Completeness and Smart Contract Security, 2019. https://medium.com/kadena-io/turing-completeness-and-smart-contract-security-67e4c41704c, Last visit October 30, 2021.

[51] Eder J. Scheid, Manuel Keller, Muriel F. Franco, and Burkhard Stiller. BUNKER: A blockchain-based trUsted VNF pacKagE repository. In Karim Djemame, Jörn Altmann, José Ángel Bañares, Orna Agmon Ben-Yehuda, and Maurizio Naldi, editors, *Economics of Grids, Clouds, Systems, and Services*, pages 188–196. Springer International Publishing, 2019.

[52] Eder J. Scheid, Cristian C. Machado, Muriel F. Franco, Ricardo L. dos Santos, Ricardo P. Pfitscher, Alberto E. Schaeffer-Filho, and Lisandro Z. Granville. INSpIRE: Integrated NFV-based intent refinement environment. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 186–194. IEEE, 2017.

[53] Eder J. Scheid, Bruno Rodrigues, Christian Killer, Muriel Franco, Sina Rafati, and Burkhard Stiller. Blockchains and Distributed Ledgers Uncovered: Clarifications, Achievements, and Open Issues. In *Advancing Research in Information and Communication Technology*, IFIP AICT Festschrifts, pages 1–29. Springer, Cham, Switzerland, August 2021.

[54] Eder J. Scheid, Bruno Rodrigues, and Burkhard Stiller. Policy-based Blockchain Selection. *IEEE Communications Magazine*, 59(10):48–54, 2021.

[55] Sean Bradley. *Design Patterns in Python: Common GOF (Gang of Four) Design Patterns implemented in Python*. Independently published, 2021.

[56] Corvin Smith. Gas and Fees. `https://ethereum.org/en/developers/docs/gas/`, Last visit January 25, 2022.

[57] Various contributors on Github.com. Web3.py. `https://github.com/ethereum/web3.py`, Last visit February 24, 2022.

[58] Various contributors on Github.com. PyJWT, 2022. `https://github.com/jpadilla/pyjwt`, Last visit March 8, 2022.

[59] Wikipedia. OpenStack, 2021. `https://en.wikipedia.org/wiki/OpenStack`, Last visit September 23, 2021.

[60] Wikipedia. Dependency Injection, 2022. `https://en.wikipedia.org/wiki/Dependency_injection`, Last visit March 10, 2022.

[61] Wikipedia. JSON Web Token, 2022. `https://en.wikipedia.org/wiki/JSON_Web_Token`, Last visit March 8, 2022.

[62] Wikipedia. Observer Pattern, 2022. `https://en.wikipedia.org/wiki/Observer_pattern`, Last visit March 10, 2022.

[63] Michael Xevgenis, Dimitrios G. Kogias, Panagiotis Karkazis, Helen C. Leligou, and Charalampos Patrikakis. Application of Blockchain Technology in Dynamic Resource Management of Next Generation Networks. *Information*, 11(12):570, 2020.

[64] Dylan Yaga, Peter Mell, Nik Roby, and Karen Scarfone. Blockchain Technology Overview. *arXiv preprint arXiv:1906.11078*, 2019.

[65] YCharts.com. Ethereum Average Gas Price. `https://ycharts.com/indicators/ethereum_average_gas_price`, Last visit January 28, 2022.

[66] Evan You. Vue.js - The Progressive JavaScript Framework. `https://vuejs.org/`, Last visit February 20, 2022.

Abbreviations

ABI	Application Binary Interface
AS	Autonomous System
BC	Blockchain
BCV	blockchain-v
BloSS	Blockchain Signaling System
DAG	Directed Acyclic Graph
dAPP	Decentralized Application
DPI	Deep Packet Inspection
E2E	End-to-End
ETH	Ether
ETSI	European Telecommunications Standards Institute
EVM	Ethereum Virtual Machine
GUI	Graphical User Interface
IPFS	InterPlanetary File System
ISG	Industry Specification Group
JWT	JSON Web Token
LB	Load Balancer
MANO	Management and Orchestration
MEC	Multi-Access Edge Computing
NF	Network Function
NFV	Network Function Virtualization
NFVI	NFV Infrastructure
NFVO	NFV Orchestrator
NP	Network Provider
NS	Network Services
OS	Operating System
P2P	Peer-To-Peer
PBFT	Practical Byzantine Fault Tolerance
PoA	Proof-of-Authority
PoC	Proof-of-Concept
PoS	Proof-of-Stake
PoW	Proof-of-Work
SC	Smart Contract
SCC	Smart Contract Connector
SDN	Software Defined Networking
SFC	Service Function Chaining

TC	Turing Complete
TTP	Trusted Third Party
UE	User Equipment
UI	User Interface
USD	US Dollar
VIM	Virtualized Infrastructure Manager
VM	Virtual Machine
VMM	Virtual Machine Manager
VMOA	Virtual Machine Orchestration Authentication
VNF	Virtual Network Function
VNFaaS	Virtual Network Functions-as-a-Service
VNFM	VNF Manager

List of Figures

List of Tables

Appendix A

Installation Guidelines

The system's source code has been hosted on Github, and can be found under the following link: `https://github.com/blockchain-v`. The installation guidelines for each part of the system can be found in the appropriate 'Readme.md' files for each respository. Additionally, the installation guidelines are also found in the submitted 'Readme.md'

This project assumes the following structure:

- blockchain-v/bcv-contract
- blockchain-v/bcv-frontend
- blockchain-v/bcv-backend
- blockchain-v/bcv-docker

A.1 Configuration Files

Since the `.env` files are used to change configurations, they have to be adjusted to the local scenario.

- `bcv-backend/.env`:
  - `W3_URL`: Has to point to the internal IP address where Ganache is hosting the Blockchain
  - `W3_CONTRACT_ADDRESS`: Insert Ethereum Smart Contract address
  - `SC_BACKEND_ADDRESS`: Ethereum address of the backend
  - `SC_BACKEND_ADDRESS_PKEY`: Private key of the backend's Ethereum address
  - `SC_BACKEND_ADDRESS_FROM`: Ethereum address of the Smart Contract creator
  - `SC_BACKEND_ADDRESS_FROM_PKEY`: Private key of the Smart Contract creator address

- `bcv-frontend/.env`:
  - `VUE_APP_CONTRACT_ADDRESS`: Insert Ethereum Smart Contract address
  - `VUE_APP_BACKEND_URL`: Has to point to internal IP address using backend's port
- `bcv-docker/docker-compose.yml`:
  - `extra_hosts`: Has to point to internal IP address

A.2 Docker Installation

This installation guideline is to be used to run the system in a containerized environment.

1. Requirements: Ganache, Truffle, Metamask, Docker
2. Start the Tacker VM
3. Start Ganache, optionally add the `truffle-config.js` as project to read the smart contract values and see event details
4. `cd` into `bcv-contract/src` and run `truffle migrate --reset`
5. `cd` into `blockchain-v/bcv-docker` directory:
   - `docker compose build`
   - `docker compose up`

A.3 Local Development Installation

This installation guideline is to be used for local development of the system.

1. Requirements: Ganache, Truffle, Vue, Python, Docker, Metamask
2. Start Ganache, optionally add the `truffle-config.js` as project to read the smart contract values and see event details
3. `cd` into `bcv-contract/src` and run `truffle migrate --reset`
4. Run the database:
   - Inside the `blockchain-v/bcv-docker` directory: `docker compose up db`
5. Run the frontend: (might require .env changes): In `blockchain-v/bcv-frontend` directory: `yarn install && yarn serve`

6. start the Tacker VM
7. Run the backend: in `blockchain-v/bcv-backend` directory:
    - `pip3 install -r test-requirements.txt`
    - `python3 -m openapi_server`

Appendix B

Contents of the CD

This project has been submitted in digital form only, as per agreement with our supervisors. Consequently, it consists of a .zip file (instead of a CD) with the following contents was handed in:

- **Project.zip**, a .zip file containing the LaTeX source code for the project.
- **Project.pdf**, a .pdf file containing the report.
- **Evaluation**, a directory containing the evaluation script and the data.
- **BCV.zip**, a zip holding the entire source-code for the BCV implementation. This includes some local-only files such as .env files.
- **Materials**, a directory containing the source files for the graphics, calculations, etc. of the report.
- **Presentations**, a directory containing the slides for the midterm presentation, held on December 16, 2021, and the slides for the final presentation, held on March 3, 2022.