CN^{2}F: A Cloud-Native Cellular Network Framework

Sepehr Ganji[1], Shirin Behnaminia[1], Ali Ahangarpour[1], Erfan Mazaheri[1], Sara Baradaran[1], Zeinab Zali[1],
Mohammad Reza Heidarpour[1], Ali Rakhshan[2], Mahsa Faraji Shoyari[2]

[1]Department of Electrical and Computer Engineering, Isfahan University of Technology, Isfahan, Iran
[2] Mobile Communications Company, Iran

Abstract—**Upcoming 5G and Beyond 5G (B5G) cellular networks aim to improve the efficiency and flexibility of mobile networks by incorporating various technologies, such as Software Defined Networking (SDN), Network Function Virtualization (NFV), and Network Slicing (NS). In this paper, we share our findings, accompanied by a comprehensive online codebase, about the best practice of using different open-source projects in order to realize a flexible testbed for academia and industrial research and development (R&D) activities on the future generation of cellular networks. In particular, a "Cloud-Native Cellular Network Framework (CN^{2}F)" is presented which uses** *OpenAirInterface***'s codebase to generate cellular Virtual Network Functions (VNFs) and deploys** *Kubernetes* **to disperse and manage them among some worker nodes. Moreover, CN^{2}F leverages** *ONOS* **and** *Mininet* **to emulate the effect of the IP transport networks in the fronthaul and backhaul of real cellular networks. In this paper, we also showcase two use cases of CN^{2}F to demonstrate the importance of Edge Computing (EC) and the capability of Radio Access Network (RAN) slicing.**

Index Terms—**Cellular Testbed, SDN, Slicing, VNF Placement**

I. INTRODUCTION

THE fifth generation of mobile communication networks (5G) promises the support of a range of applications from Ultra-Reliable Low-Latency Communication (URLLC) to enhanced Mobile Broadband (eMBB) to massive Machine-Type Communication (mMTC) connections. The diversity of supported applications hinders to use of the conventional one-size-fits-all structure for future cellular networks and innovative models are required. In order to increase flexibility and efficient resource sharing among different application sectors, new models are supposed to be built upon *virtualization/softwarization* technologies, such as Software Defined Networking (SDN), Network Function Virtualization (NFV), and Network Slicing (NS) among others [1].

To move toward 5G innovative concepts, several AI/ML-based algorithms have been proposed to enable intelligent and autonomous network management, which should be tested and evaluated before their commercial rollout [2]–[6]. These algorithms use a large amount of data extractable from both the Radio Access Network (RAN) and Core Network (CN) to learn patterns and automatically enhance network operations. Small-scale testing of 5G use cases allows developers to identify potential issues and address problems before a large-scale rollout. In the same way, 5G and Beyond 5G (B5G)
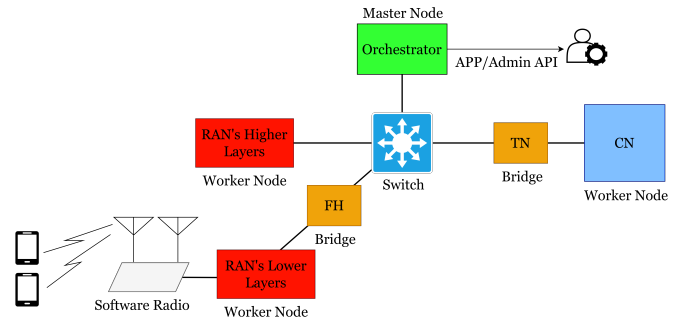

Fig. 1: CN^{2}F architecture with a master node and three worker nodes

testbeds enable researchers to evaluate real-world network scenarios [7], [8].

In this paper, we propose the CN^{2}F, a simple Cloud-Native Cellular Network Framework, as a simple general framework, to build prototypes for future generations of cellular networks. Fig. 1 depicts the CN^{2}F structure. The CN^{2}F comprises a cluster of four nodes (one master and three workers), an L2/L3 switch, two bridge nodes, and a software-defined network.

In this paper, first, we review the main technological concepts and frameworks such as containers, Docker [9], Kubernetes [10], Mininet [11], and RAN splitting. Then, we explain how to use these primitives to set up the CN^{2}F framework (including the cluster and bridges). Specifically, for the CN^{2}F setup, we provide an *Ansible playbook* to prepare and install necessary packages on nodes, which is the best practice to make the Kubernetes cluster. We also describe how to create cellular Virtual Network Functions (VNFs) (or *Pods*) from *Docker Images*. For the cellular VNFs, we take OpenAirInterface (OAI) project as one of the existing open-source candidates for the software implementation of the 4G/5G RAN and CN components.

Our main focus in this work is how to set up the CN^{2}F framework regardless of the particular VNFs that use this framework to realize the final prototype of a specific 4G/5G network. Since the 4G/5G open-source projects are evolving and have different stability levels, we resort to using OAI's (MAGMA-MME based) 4G implementation as it is claimed to be "running for hours and days without any restart" [12]. By the way, 4G core network is still used in 5G non-standalone (NSA) systems.
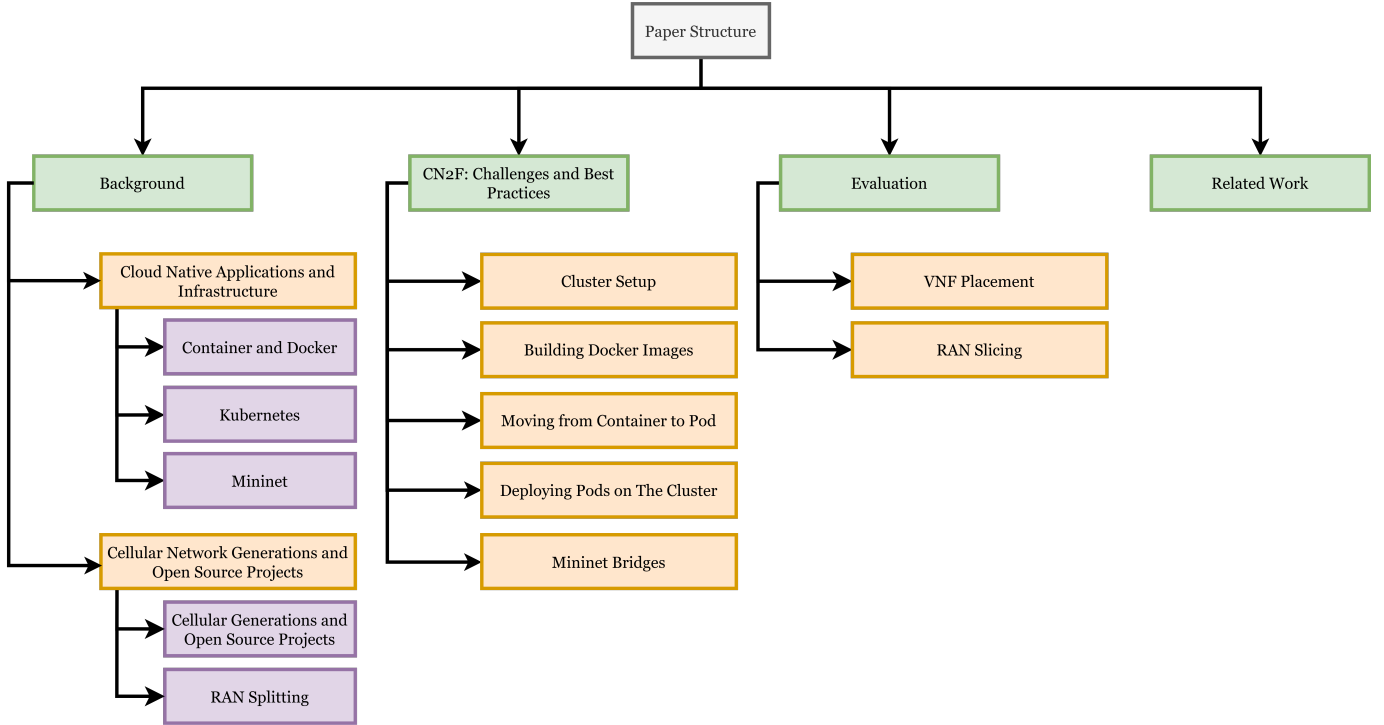
Fig. 2: Organization of the paper

Finally, we demonstrate using CN^2F to investigate the importance of VNF placement and RAN slicing as two significant capabilities of future cellular networks.

The results of this paper are reproducible, and the source codes alongside scripts to set up the CN^2F and execute the use cases are publicly available on our GitHub[1] repository for the academia and industrial community.

The remainder of this paper is structured as follows (see Fig. 2): Section II provides the necessary background and definitions. Section III describes the CN^2F structure in detail. Section IV explains our evaluation and the implementation of two use cases in our framework. Section V reviews alternative testbeds proposed in the literature, along with their goals and applications. Finally, Section VI concludes the paper.

II. BACKGROUND

In this section, we briefly review the required background material for building a cloud-native cellular network framework. This section is presented in two subsections. The first subsection covers the topics encountered at the time of generating cloud-native applications and building a cluster over which the application is deployed. Therefore, the first subsection is general and not specific to cellular networks. Then, in the second subsection, we discuss topics specific to cellular networks, such as different generations of cellular networks and available options and open-source projects to be used for building and deploying cellular VNFs on a given testbed/cluster.

[1] Available at https://github.com/CN2F/core

A. Cloud-Native Applications and Infrastructure

In this part, we look over the concepts and technologies that play essential roles in developing cloud-native applications and building the supporting infrastructure.

1) Container and Docker: A container is a complete package of an application software itself and all its dependencies which can run on different computing systems quickly and reliably without complaining about necessary dependencies. The Linux primitives that enable containerization are "namespaces" that provide isolation (in different levels, such as process, filesystem, and network) and "cgroups" which can be used to restrict resource consumption (e.g., memory, CPU, and bandwidth).

Docker [9] is an open-source virtualization technology that facilitates the deployment, creation, and management of containerized applications. Similar to processes that are running instances of programs (executable files), the containers are also running instances of Docker images in their (possibly) isolated and restricted environments. An image is a compressed archive file that contains application (and dependencies) files inside some directories along with a startup command that specifies the program by which the container's life begins executing (see Fig. 3). An image is created based on a recipe, known as Dockerfile, which specifies the base image, files to be copied inside the image, instructions to build the application, and the startup command. Moreover, run-time parameters (e.g., networking, environment variables, and volumes) can be specified declaratively in a configuration "docker-compose.yml" file which is fed to a tool called Docker Compose to run/manage containers accordingly. Docker also provides a registry (Docker Hub) to share (push/pull) official images built by different development
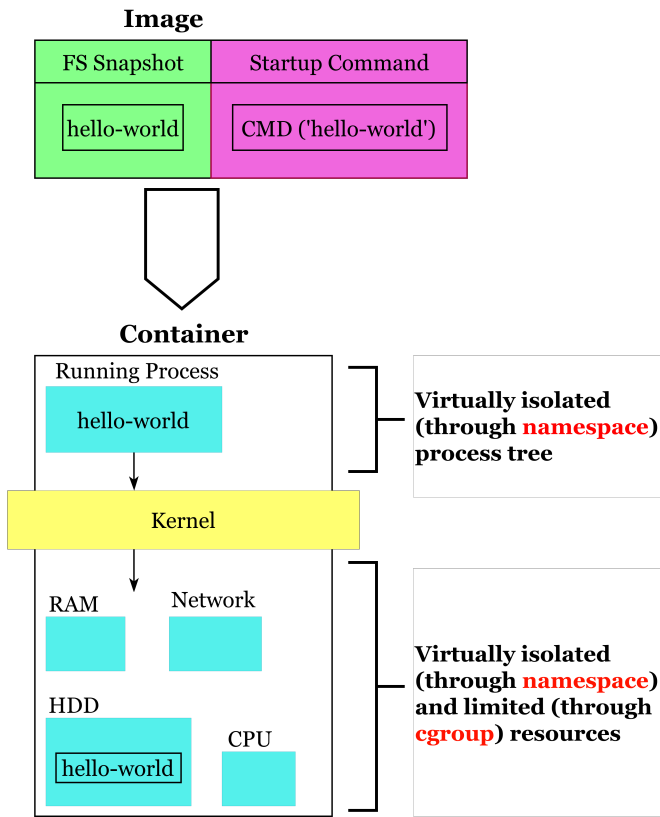
Fig. 3: Image and container

Fig. 4: Structure of a Kubernetes cluster with two worker nodes

Fig. 5: Structure of a pod containing two containers

teams.

2) Kubernetes: Kubernetes [10] is an open-source platform for container orchestration, introduced by Google in 2014, to manage and automate the deployment, scheduling, monitoring, maintenance, self-healing, rollout and rollback, and operation of application containers across a cluster of machines. In fact, Docker enables developers and operators to create and run containers, while Kubernetes is used to orchestrate different containers. More precisely, Kubernetes allows the deployment of microservices over a number of machines and simultaneously hides the infrastructure from the application's point of view. Users can also deploy applications on different cloud providers using a standard set of APIs provided by Kubernetes, without separately needing to use each provider's API for deployment and management of the applications.

In order to use the application management capability in Kubernetes, a description of the application's design is required. Then, Kubernetes turns the aforementioned description into a running set of objects and ensures they keep running by restarting those that fail. If some changes occur in the application's design, Kubernetes takes the steps required to transform (update) the set of running objects into new ones.

To distribute microservices among several machines, we need to create a Kubernetes cluster. The structure of a Kubernetes cluster mainly includes a number of worker nodes and a master node managing the application running over the workers. This structure is illustrated in Fig. 4. As shown in this figure, the master node includes several components, namely etcd (which is a key-value database recording the
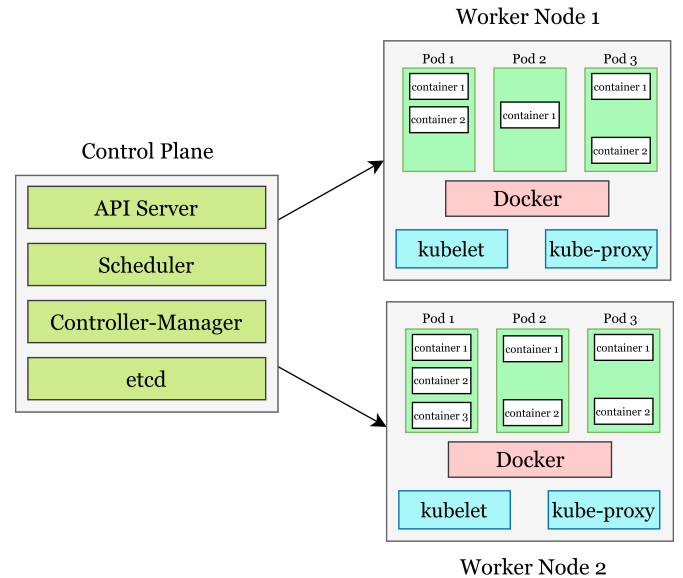
desired/current states), API server (which provides communication among etcd and other components), scheduler (which is responsible to place "pods" and other objects on worker nodes), and controller (which is the brain of Kubernetes). The pod is a collection of one or more containers with some managing metadata, which represents the most basic deployable unit that can be created and managed in Kubernetes. On the other hand, a worker node consists of components called kubelet (the Kubernetes agent on each node, which is responsible for executing master's commands, such as pod creation/deletion using tools (e.g., Docker)) and kube-proxy (responsible for the networking and services).

Kubernetes has also spawned many other related open-source projects, mainly under the umbrella of the Cloud-Native Computing Foundation (CNCF), such as CoreDNS, Envoy, Helm, and Prometheus. Moreover, CNCF organizes several KubeCon + CloudNativeCon conferences per year.

a) Pod: A pod is a group of one or more containers (with shared hostname, IPC[2], and network namespaces[3]) in addition to a set of specifications, including parameters such as labels and ports, among others. A pod can be viewed

---

[2]Inter Process Communication

[3]Filesystem namespaces of pod's containers are different but they may have access to shared *volumes*.
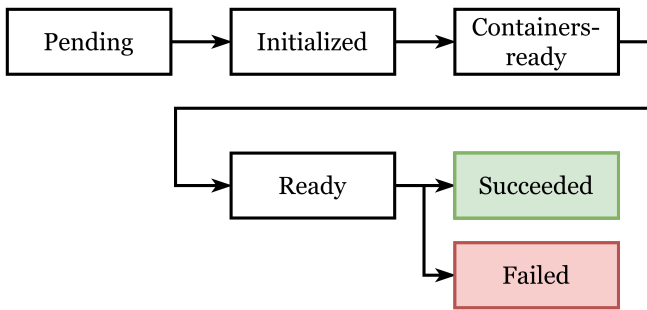
Fig. 6: Lifecycle of a pod

as an application-specific *logical host* and is the smallest deployable unit of computing in Kubernetes. The pod creation needs no more technology than what is required for the creation of containers, as it is, in fact, a main container (called the "infrastructure" or "pause" container) which hosts the application container(s) (see Fig. 5). Since Kubernetes is a declarative platform, to create pods, we instruct controllers, such as *ReplicaSet*, *Deployment*, *Job*/*CronJob*, *DaemonSet*, and *StatefulSet* through a "pod template" in their yaml files to create and manage those pods on our behalf[4]. Fig. 6 shows a typical pod's lifecycle. Before scheduling, it is in the *Pending* (or *Pre-scheduled*) state. Then, at the time of the creation of the pause container and execution of the "init" containers, the state is *Initialized*. Then, the main containers become ready at the *Containers-Ready* state. After that, the pod is ready to serve at the *Ready* state. In the case that all containers inside the pod terminate successfully, the state changes to *Succeed*, and otherwise, to *Failed*. Using the init containers which run before the main ones, we can make sure that all prerequisites before starting the main containers are satisfied. One usage of this feature in the pod's lifecycle is to enforce an order for the execution of the pods.

b) Networking: Kubernetes's model for pod networking is flat in which all pods (logical hosts) are in the same subnet (i.e., connected through a "logical" L2 switch). Pods inside a single node are connected to the same bridge through virtual Ethernet interface pairs. However, for pods on different nodes to communicate, we need to somehow connect the bridges on different nodes to each other. As a result, providing Kubernetes's network abstraction for pods can be complex and usually achieved through an additional SDN layered on top of the actual network (which uses encapsulation and some Linux networking tools, such as iptables, routes, and IP-forwarding). Container Network Interface (CNI) is a project to simplify networking configurations in Kubernetes through the addition of plugins, such as Calico, Flannel, and Weave Net.

c) Volume and ConfigMap: By default, a pod's filesystem is the one defined in its image, and any file operation (e.g., creating a file or writing into a file) is ephemeral and will be lost by pod termination/restart. In Kubernetes, we can use different types of *volumes* to have different levels of file persistency, such as *emptyDir* (persistency across container restarts

during the pod's lifetime), *hostPath* (higher persistency across pod restarts in a node), and *nfs* (even higher persistency across node change). Similar to networking, storage consistency is complex in general, and several Container Storage Interface (CSI) projects have been developed to accomplish the work as plugins, such as Cinder and Cephfs.

A related topic to volumes is the *ConfigMap* which is used to pass arguments to containers by mounting configuration files into containers through a special type of volume, not surprisingly, called ConfigMap.

3) Mininet: In SDN-enabled networks, the control plane handling management operations is logically centralized and physically isolated from the data plane, which in turn enables high network configurability and programmability. The network programmability and the capability of optimizing resource allocation and utilization in a centralized way, made possible by the SDN paradigm, are expected to alleviate the burden of the data onslaught expected from data-intensive applications [13]. Mininet [11] is an emulation tool that allows one to virtually emulate a complete SDN network scenario comprising a number of virtual hosts, controllers, switches, and links. It uses container-based virtualization to make a single system act as a complete network.

B. Cellular Network Generations and Open-Source Projects

Wireless mobile networks first started as the extension to the Public Switched Telephone Networks (PSTN) in order to support mobile users. At first, they were composed of a single high-power base station to cover an area of around 70 to 80 km radius. In order to increase the capacity (number of subscribers), the *cellular idea* emerged. The cellular structure suggests using multiple low-power base stations through which the whole area is tiled by (cell-like) hexagons and the *frequency reuse* becomes possible. The downside of the cellular structure is the need for handoff and interference management. However, because of its capacity-increasing benefit, the underlying structure of all current Wireless Wide Area Networks (WWAN) obeys the cellular pattern.

1) Cellular Generations and Open-Source Projects:

a) 1G and 2G Networks: The first generation (1G) was analog and only supported voice. The 2G is characterized by being the first that used digital modulation, and hence, featured higher service quality (encoding), security (encryption), new services (SMS[5]), and more efficient Radio Frequency (RF) spectrum usage through Time Division Multiple Access (TDMA) and Code Division Multiple Access (CDMA). The dominant standard of the 2G network is GSM (Global System for Mobile communication) which is still alive due to the vast investment and capability to support Internet of Things (IoT) applications. For the open-source software projects that implement 2G networks' components, we can refer to OpenBTS and YateBTS.

b) 3G Networks: The main feature of the 3G networks is the change of the focus from voice to data. In this generation, the core of the network is separated into two parts; one based on the conventional circuit switch architecture to support

---

[4]Different controllers provide different functionalities, such as maintaining the number of pods, creating a specific pod on each node, and managing the pods during an application update
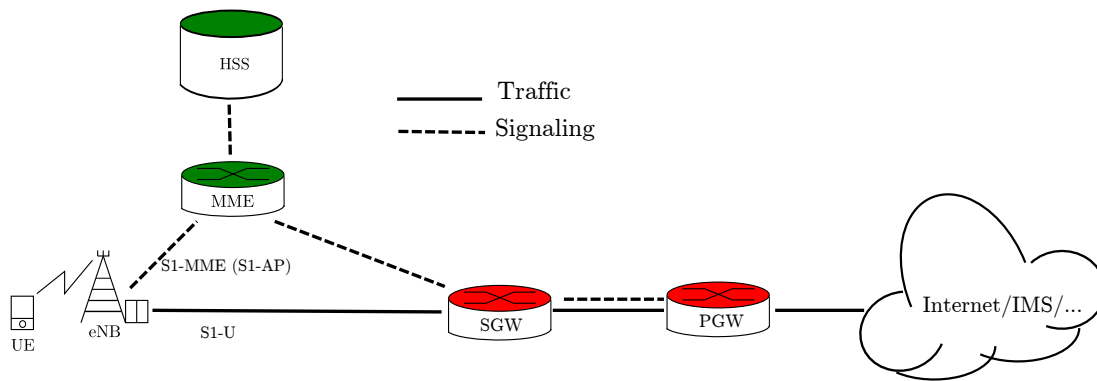
[5]Short Message Service

Fig. 7: A simplified LTE architecture

voice and one based on the packet switch architecture (best suited) to support data. As a result, the 3G networks support both voice and data. Moreover, 3G realized ITU's IMT 2000 vision for cellular networks and supports new services, such as mobile Internet through increased data rate. The dominant 3G standard is UMTS (Universal Mobile Telecommunications Service) which then evolved to HSPA (High-Speed Packet Access or 3.5G) as a transient generation with higher data rates. 3G networks are also still alive, again because of the vast investment and supporting IoT applications. OpenBTS-UMTS is an open-source project that implements parts of the 3G networks.

c) 4G Networks: The main feature of the 4G networks is an all-IP structure. In other words, the core of the network is purely packet-switched, named EPC (Evolved Packet Core). As a result, from this generation forward, we can state that cellular networks are the extension of the Internet and data services to mobile users. 4G networks satisfy the requirements of ITU's IMT-Advanced, and for multimedia communications, such as voice, they rely on the IMS (IP Multimedia Subsystem) framework. Moreover, 4G networks demonstrate many advances in the communication theory in practice, such as small calls, relays, carrier aggregation, Coordinated Multi-Point (CoMP), Inter-Cell Interference Coordination (ICIC) in the RAN, Control and User Plane Separation (CUPS) (as the first traits of *SDN*), and Dedicated Core Networks Selection (DECOR) (as the first traits of *slicing*) in the core of the network. The dominant standard of 4G networks is LTE (Long Term Evolution), and the open-source 4G projects include OAI and srsLTE.

As this paper implements 4G cellular VNFs on the CN²F, here we provide more details on EPC components. Fig. 7 depicts a simplified LTE architecture. All links are logical and may be realized through an IP network. The PGW (Packet Gateway) is the gateway of the cellular network. Its main functionality is to be the mobility anchor point, which hides the mobility of the users from the outside world. PGW uses tunneling (encapsulation of the packets) to route the traffic of users to where they are located. Other functionalities of the PGW include QoS enforcement and IP address allocation to connected UEs (User Equipment). However, there may be millions of users, and supporting all of them with one or a handful of PGWs is not practical. This is where the need for SGWs (Serving Gateways) becomes evident. In fact, the whole area under the coverage of a cellular network operator is divided into regions, consisting of several cells. Each SGW is responsible for tracking the location of the UEs in a specific region and making necessary tunnels to their corresponding cell site base station (or eNB (evolved Node B)). Then the end-to-end tunneling between the PGW and a user (UE) in a specific region is broken into two tunnels; the tunnel from the PGW and an SGW (the one responsible for that region) and the tunnel between that SGW and the eNB to which the UE is connected. As a result, the PGW needs to change its state to about one UE only when it crosses the region's borders. The component which is informed about the UE location by the eNBs and instructs SGWs and PGW(s) to configure their tunneling parameters is the MME (Mobility Management Entity). MME is also involved in paging, handover, authentication, security, and management of subscription profiles. Finally, as access to the network is not for free, we need a database that records subscribed users' information, including their identities, imprecise locations in the network, security (secret keys), and QoS contexts. The HSS (Home Subscriber Server) realizes this database and its secure connection to the MME in the LTE architecture.

The 3GPP release 14 introduced the idea of CUPS which is the separation of the user and control plane as depicted in Fig. 8. In CUPS, SGW and PGW are decomposed into two parts; one that routes users' data packets (SGW-U and PGW-U) and one that performs controlling the connections (SGW-C and PGW-C). Therefore, MME, as a pure controlling entity, makes the connection to SGW-C and PGW-C which control SGW-U and PGW-U, respectively. CUPS adds flexibility to network deployment and operation (as different components can be independently scaled up/down on demand and in different places) and it can be considered the first introduction of the SDN in cellular networks which is matured/completed in the 5G networks (3GPP release 15 and above).

d) 5G Networks: The cellular networks take an ambitious step in 5G with the goal of supporting applications with different demands (first envisioned in ITU's IMT-2020). The realm of these applications is specified by the eMBB, the mMTC (or massive IoT (mIoT)), and the URLLC as the extreme corners where well-known applications (e.g., smart homes, augmented reality, and industry 4.0) can be placed
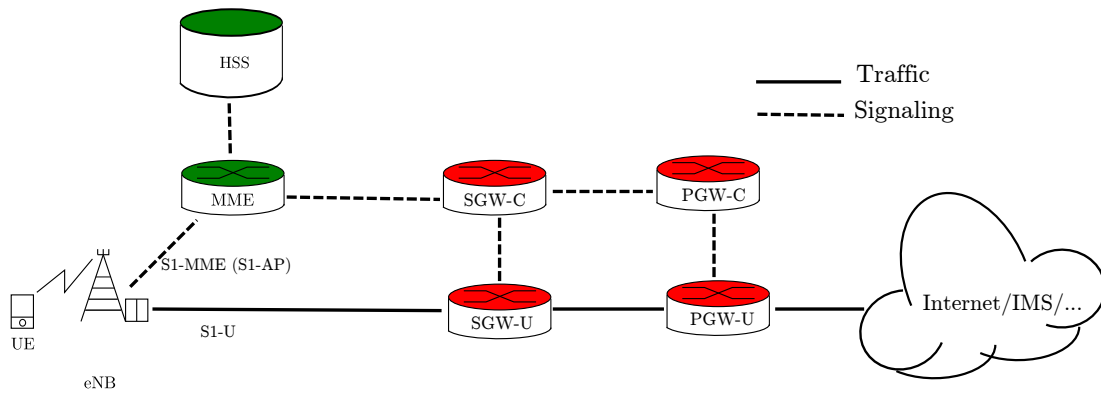
Fig. 8: The LTE's CUPS architecture

somewhere between, based on their demands similarity to the mentioned corners (e.g., required peak data rate, connection density, mobility, and latency). After a period of ambiguity and discussion on "what will 5G be" [14], it became clear that in order to support such diverse applications, SDN and NFV would be the main key technologies in any true 5G realization. The SDN and NFV enable the creation of several logical networks, called network slices, on top of the same infrastructure, each tailored according to the requirements of a specific application. On the other hand, in the RAN, 5G introduced New Radio (NR) which features even higher data rates, more flexibility (such as different spectral modes of operations and the alternative ways to interact with the core network; SA/NSA (Stand-Alone/Non-Stand-Alone)), massive MIMO, mmWave, and dual-connectivity (simultaneous connection of 5G UEs to a 4G eNB and a 5G gNB).

5G core network, also known as SBA (Service Based Architecture), consists of entities that less or more resemble their EPC's counterparts, such as Unified Data Management (UDM) ($\approx$ HSS), User Plane Function (UPF) ($\approx$ SGW-U and PGW-U), Session Management Function (SMF) ($\approx$ SGW-C and PGW-C), and Access and Mobility Management Function (AMF) ($\approx$ MME). It also comprises new entities related to VNF and network slicing, such as Network Repository Function (NRF) (a repository listing all the functions), Network Slice Selection Function (NSSF) (which selects the network slice instances for a given UE), Network Exposure Function (NEF) (which exposes some internal events related to UEs), and Network Data Analytics Function (NWDAF) (which collects and analyzes information from the network and its users). The main characteristic of the SBA is that different entities interact with each other through well-established (and successful) API standards, such as HTTP and JSON, which makes the core of the 5G cellular networks more look like the usual client-server model in the Internet.

The 5G paradigm shifts towards more softwarization and virtualization, which has also revolutionized open-source activities. Projects such as OAI and srsLTE also have products for the 5G networks. Other projects such as free5GC and Open5GS are also contributing to this subject (see [15] for a thorough survey and comparison among different 5G open-source projects). Different open-source projects for cellular
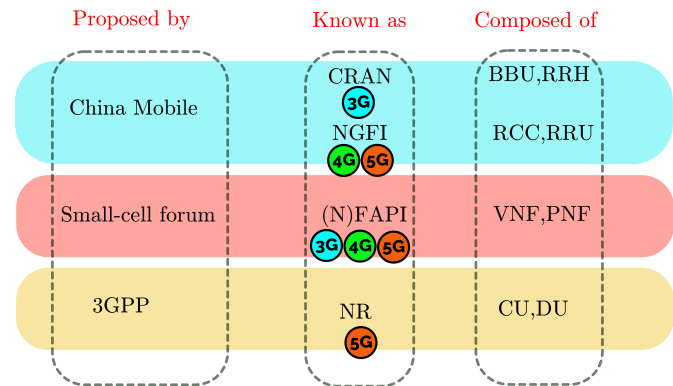

Fig. 9: RAN splitting proposals

networks differ from each other in terms of community support, components they have implemented, and stability level. We found only one project that claims "running for hours and days without any restart" which is the LTE's EPC (release 14) with MAGMA MME-based deployment [12]. Hence, in this work, we have used this project to be deployed on the CN^2F.

2) RAN Splitting: RAN splitting refers to the split of the RAN protocol stack, including the physical and MAC layers, into two or more parts which can be deployed separately within distinct nodes (or pods) and they can interact with each other over *well-defined* APIs. The idea behind the RAN splitting is to reduce capital expenditures and operating expenses (as a result of minimizing the cell site equipment), increase resource sharing and cope with the *tidal effect* (as a result of centralization), and enable advanced collaborative signal processing techniques, such as interference management and CoMP transmission and reception.

For RAN splitting, different schemes have been proposed by different organizations as shown in Fig. 9. China Mobile is the pioneer by first introducing the idea of the RAN splitting and proposing the Cloud/Central RAN (CRAN) for the 3G networks and, later, the Next Generation Fronthaul Interface (NGFI) for 4G/5G networks. In CRAN terminology, the whole protocol stack breaks down into two parts named BBU (Base-Band Unit) and RRH (Remote Radio Head). The I/Q samples are sent back and forth between BBU and RRH using Common Public Radio Interface (CPRI) links. However, high stress on
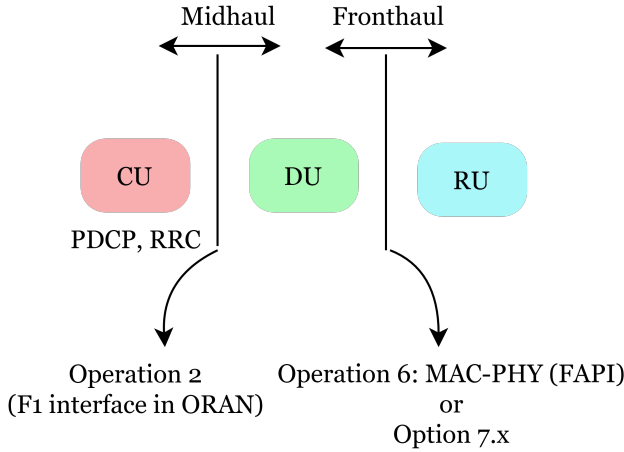
Fig. 10: The de-facto choice for RAN splitting


```
$ ansible-playbook -i remote-host main.yml --ask-become-pass --ask-pass
```

Fig. 11: Installing required packages and configuration on cluster nodes using Ansible

the fronthaul link (in terms of bandwidth and delay) resulted in limited deployments of CRAN in practice. To cope with even further bandwidth and latency constraints in 4G/5G, China Mobile redesigned the splitting in the NGFI scheme in which the separated components of the RAN are called the Remote Cloud Center (RCC) and Remote Radio Unit (RRU). China Mobile proposes six options in NGFI for the RAN splitting and compares the bandwidth and delay requirements of each option in its white paper. OAI supports the following node functionalities:

- eNodeB
- RCC and RRU (NGFI IF5): split-point at the OFDM symbol generator (i.e., frequency-domain signals)
- RCC and RRU (NGFI IF4p5): time-domain fronthaul (more than 1 GbE is required)

On the other hand, Small Cell Forum (SCF) has been contributing by offering (Network) Function Application Platform Interface ((N)FAPI) which defines the API between the physical and the MAC (and above) layers for 3G, 4G, and 5G networks. Here, the separated parts of the RAN are denoted by Virtual Network Function (VNF) and Physical Network Function (PNF), respectively. Finally, at the time of introducing the New Radio (NR), 3GPP suggested ten options and suboptions for the splitting point for the 5G-RAN and referred to the separated parts as the Centralized Unit (CU) and Distributed Unit (DU). As depicted in Fig. 10, among different options, the de-facto choice is to place the upper two layers (PDCP and RRC) in the CU (3GPP's option 2 and ORAN F1 interface) and the remaining layers in the DU which can be further divided into DU and Radio Unit (RU). OAI supports the functionality of CU and DU based on the de-facto split choice.

III. CN^2F: Challenges and Best Practices

CN^2F aims to facilitate the building of flexible network scenarios with various network topologies. In this section, we describe the steps to construct the CN^2F accompanied by our best practices (i.e., the best realized solution to cope with a challenge) i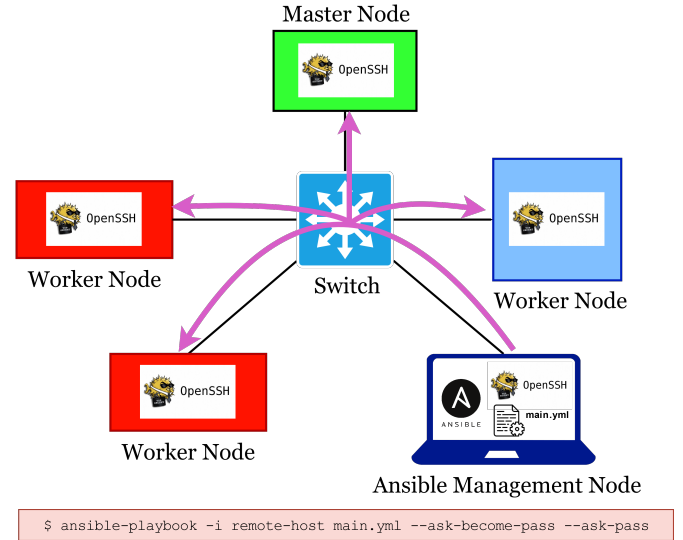n each step. In addition, we present a detailed tutorial demonstrating how to get from a portable network implemented with Docker containers to a cloud-native network implemented on the Kubernetes platform. The proposed structure in CN^2F resembles the "flexible" and "hierarchical" model of the modern cellular network (Fig. 1).

Specifically, CN^2F has been designed to demonstrate a network with edge computing, cloud computing, RAN splitting, and NFV capabilities. Two bridges (each emulating an IP network) are incorporated in the CN^2F structure; FH and TN. The FH bridge sits between distributed RAN components, such as RCC (or CU) and RRU (or DU), and therefore, it enables the RAN splitting test cases. On the other hand, the TN bridge may divide the structure into two parts; one close to RAN (or UEs) and one far away. As a result, the distinction between edge computing and cloud computing can be investigated.

Besides the bridges, the CN^2F includes a cluster of four machines. The cluster size is minimal as we need one machine for the master node, two machines (i.e., two worker nodes) to host different components of the distributed RAN, and one machine (i.e., another worker node) to host the VNFs (such as core network's VNFs), which may reside in the data center in reality. All nodes are connected to a Gigabit Ethernet switch. One node is further connected to the USRP-B210 (via USB 3.0) which plays the role of radio head in our setup.

Our best practice in order to automate the process of installing required packages (e.g., Git, Python, Wireshark, and Docker), doing some configurations regarding CPU states and the Linux kernel, and copying some bash scripts (for enabling IP forwarding, IP addressing, etc) is to first list all of them in an *Ansible playbook* file in the "Ansible management node" (in our case a laptop with the SSH connection to the cluster nodes, aware of their IP addresses, usernames, and passwords, which has *ansible* and *openssh-client* installed). In this way, after installing the OS on bare-metal machines (cluster nodes), it suffices to just install *openssh-server* on them. Then, the Ansible management node can do all the required installations

and configurations specified in the playbook on all cluster nodes with a single command (Fig. 11).

A. Cluster Setup

In this section, we describe the steps to set up a Kubernetes cluster. As explained in the background, a Kubernetes cluster comprises a set of components, such as kube-proxy and kubelet, which should be installed and configured on each node and will recover in case of restarting a node. Since manually installing these components is time-consuming and error-prone, several tools are provided for the cluster installation. We tested three tools, including Kubeadm, Kubesphere, and Rancher Kubernetes Engine (RKE) to install clusters. Among them, RKE is selected as this tool is handier, more stable, and easier to configure. Also due to its flexibility, we can configure multiple cluster options in the rancher configuration file which is used to deploy a cluster.

The Kubernetes networking model requires certain network features but at the same time allows a degree of flexibility regarding the implementation. As a result, various projects have been released to address specific environments and requirements. Container Network Interface (CNI) is one of those projects supporting plugin-based functionality to simplify networking in Kubernetes. The main purpose behind CNI is to provide enough control to administrators for monitoring communication while reducing the overhead of manually generating network configurations. One of our main challenges was to pick up the best CNI for our framework. Among different CNI providers, Calico [16] is used in the final structure of CN²F as it enables us to assign a static IP to each deployed pod, and this easy-to-deploy CNI provider is supported by RKE.

We deploy a Kubernetes cluster using four machines with Intel Core i9-11900 CPU@3.5 GHz, 32 GB of RAM DDR4 memory, using an operating system Ubuntu 18.04 LTS, interconnected by an L2/L3 switch. This cluster consists of a master node and three worker nodes as shown in Fig. 1.

B. Building Docker Images

Docker provides two ways to run a container using Docker images. One is to pull the Docker image of the application from a Docker registry (e.g., Docker Hub), and the other is to build the Docker image of the application using Dockerfile. Dockerfile is defined as a recipe containing all the required dependencies and some instructions in order to make the application runnable inside the Docker container.

1) CN Images: We have used the Dockerfiles in the OpenAirInterface GitHub repositories [17] to build the corresponding image of each module. We pulled the images from the official repository and pushed them to our own repository with a new tag. As a result, there is no need to rebuild images every time.

2) RAN Images: To build the RAN module, we employed Dockerfiles existing in EURECOM GitLab [18]. In order to build the RAN image, three Dockerfiles are employed, which are sequentially built to make the final RAN image. The use of three Dockerfiles brings about a lightweight final image for the RAN module and accelerates the process of image rebuilding. Since the Dockerfile for constructing eNB image is being updated by EURECOM every once in a while in the *dev* branch, we tracked this Dockerfile (also Dockerfiles for RRU and RCC) and tested different tags in order to find the ones that work properly. This was one of our challenges during the CN²F framework setup.

C. Moving from Container to Pod

As mentioned in the background section, a pod is a Kubernetes object that encapsulates one or more containers. The first step in deploying a module is to create a pod definition (which is then placed inside the "pod template" part of a Deployment object) based on the docker-compose file. All the needed configurations (e.g., environment variables, volumes, etc) must move properly and precisely from the docker-compose file to the pod definition file. We can consider the MME definition file as an example since MME is the most complex among OAI modules and it covers all significant issues. A simplified MME definition file is illustrated in Fig. 12. In this part, we first introduce the pod definition file and its main sections. Then, the most significant points for making pod definition files based on docker-compose files will be elaborated on.

1) Basics of Pod Definition File: A pod definition file consists of four basic sections; *apiVersion*, *kind*, *metadata*, and *spec*. The spec is the part under which an array of containers could be configured and is worth concentrating. There exist a *nodeSelector* option, under the spec part, to determine the specific node where we want to deploy each pod. This is one of the basic parts of a definition file.

2) Image to Pod Translation:

a) Environment Variables: In Docker, environment variables can be set using either the *environment* attribute or the *env_file* option. This can be represented as an *env* section in the pod definition file. Note that to configure the RAN module, it is required to bind the */dev/bus/usb* path to the eNB container as we use a USRP board connected to the USB3 port.

b) Security: Similar to *privileged: true* in the docker-compose file, we define a *securityContext* in the definition file that conducts Kubernetes to run the containers in privileged mode. This is required for containers to have some capabilities, such as system administrator, in order to operate as expected.

c) Networking: In Docker, containers are assigned a static IP address, which enables them to communicate with each other using real interfaces. However, in this setup, networking settings, such as routes, need to be manually configured. One of the challenges we encountered in Kubernetes was assigning a static IP to each pod, which is necessary for OAI modules as they require programs to bind to a specific IP. To this end, we used Calico as a CNI so that each deployment has a unique IP address in the range of the Calico subnet. Also, there is no need to set any networking settings manually.

d) Data and Volumes: The *volume* section in the docker-compose file can be translated into the *hostPath* and *mountPath* sections. The hostPath is located on the node that the nodeSelector option referred to. Also, we used another type of volume named *ConfigMap*. There exist some configuration files which

```
1   apiVersion: apps/v1
2   kind: Deployment
3   metadata:
4     name: mme
5     namespace: ion
6   spec:
7     selector:
8       matchLabels:
9         app: mme
10    template:
11      metadata:
12        labels:
13          app: mme
14        annotations:
15          "cni.projectcalico.org/ipAddrs": "[\"10.233.0.130\"]"
16      spec:
17        hostname: mme
18        containers:
19        - name: magma-mme
20          image: magma-mme:latest
21          imagePullPolicy: Never
22          resources:
23            limits:
24              memory: "4Gi"
25              cpu: "2000m"
26          securityContext:
27            privileged: true
28          ports:
29          - containerPort: 3870
30          - containerPort: 5870
31          - containerPort: 2123
32          volumeMounts:
33            - mountPath: /magma-mme/etc/mme_fd.conf.tmplt
34              name: mmevolcfg
35              subPath: mme_fd.conf.tmplt
36          command:
37            - "bash"
38            - "-c"
39            - "cd /magma-mme/scripts; ./mme-cfg.sh"
40          env:
41            - name: TZ
42              value: "Europe/Paris"
43            - name: REALM
44              value: "openairinterface.org"
45        initContainers:
46        - name: hsschecker
47          image: subfuzion/netcat
48          imagePullPolicy: Never
49          command: ['sh', '-c', "for i in {1..100}; do sleep 10;
                      if nc -z 10.233.0.219 3868; then exit 0; fi; done; exit 1"]
50        volumes:
51        - name: mmevolcfg
52          configMap:
53            name: "mmecfg"
54            defaultMode: 0777
55        nodeSelector:
56          environment: cloud
57
```

Fig. 12: A simplified MME definition file

should be converted into ConfigMap in the Kubernetes cluster to be used as volume bindings in the modules.

e) Dependencies: The *depends_on* field in a docker-compose file sets the order of container deployment. However, in Kubernetes, we use *initContainers* to achieve the same functionality. Init containers check if a specific port of the container, they depend on, is open using the *netcat* command. Once the init container successfully returns, the pod will be deployed.

D. Deploying Pods On The Cluster

After installing the cluster and configuring framework modules, a few steps need to be done to deploy the whole network and check the connectivity of modules to each other. Also, we faced some challenges during the deployment phase, the most important of which are elaborated on in the following.

1) VNF Requirements and Startup:

a) HSS: The HSS needs to be connected to the Cassandra database to fill in the database tables with subscribers' profile data (such as their IMSI, APN, and secret keys). To this end, some specific empty tables must exist in the database

before deploying the HSS. In Docker deployment, a "db_init" container is responsible for creating those tables. However, in Kubernetes, those (empty) tables could be created using an init container which executes the db_init instructions. Moreover, in our setup, we just make sure that Cassandra pod is deployed in the same system that the data tables are stored.

b) MME: Since the MME module should be connected to the SPGWC/SPGWU and HSS, it needs to be checked whether they are correctly configured. To configure the MME module, we utilized the ConfigMap object to include the necessary files. This approach enables us to avoid mounting the entire directory, as we did in previous methods of setting volumes, and instead, only mount a specific file. Additionally, the ConfigMaps are automatically updated by the kubelet periodically in case of any changes. This means that if any parameters are modified, there is no need to restart the corresponding pod.

c) SPGWC and SPGWU: In the CUPS architecture, the SPGWC is the control plane component and the SPGWU is the corresponding data plane component of the conventional monolithic SPGW. These modules need to connect to each other, to the MME, and the eNB/RCC.

d) eNB: The node where eNB is deployed must be connected to the USRP board via USB3 port. Along with modules' IPs that need to be set correctly in the eNB configuration file, Mobile Network Code (MNC) and Mobile Country Code (MCC) are also required to be set the same as are in the MME module and also in the programmed sim card. The system (PC) needed to deploy the eNB must be very low-latency and should be sufficiently fast. Thus, it is crucial to check the kernel version and i7z report of that node, before deploying this module, to make sure everything works properly (This is also very important in RRU/RCC as well).

e) RCC and RRU: To establish a connection between RCC and RRU, their versions should match, similar to the case of eNB. Moreover, in mounted configurations, the local and remote IP addresses need to be set according to the pod's IP address in their definition files. Note that the configurations for both must be identical to work.

f) FlexRAN: FlexRAN is a flexible and programmable platform that separates the RAN control and data planes and supports the design of real-time RAN control applications. It uses two different ports, one to connect to the eNB agent and one for API requests which we use to interact with this module. There is a *NETWORK_CONTROLLER* section in eNB (or RCC) configuration file in which we can specify if we want to deploy the FlexRAN agent by setting *FLEXRAN_ENABLED=yes/no*.

2) Networking: As previously mentioned, we assigned a static IP to each module. One of the challenges was the direct communication between the RRU and the RCC. We have tested two methods to cope with this challenge; First, we attempted to use *hostNetwork* instead of CNI. By doing this, the pod would inherit all of the host interfaces and assigned IPs of the node, allowing us to separate the RRU-RCC link from other links. However, this method did not work properly. Calico CNI deploys a Calico node pod as an agent on each cluster to manage networking cluster-wide. In this scenario,

some of the Calico node pods become unready after a while due to a BGP routing error.

We attempted a second approach which involved manually deploying Calico instead of relying on RKE to automatically deploy it. It should be noted that in this case, the Calico subnet was modified. While this method improved the situation to some extent, it should be emphasized that the structure of the OAI modules is not intended to accommodate such scenarios.

3) Software Requirements: The important point is that RAN module significantly depends on the kernel and OS settings. Therefore, we must consider items, such as low-latency kernel and power management settings in the BIOS and Grub in edge nodes. One of our challenges in this part was finding the most compatible version of the low-latency kernel while deploying RRU-RCC.

4) Offline UHD Image: The eNB module uses an *image_downloader* script to program the FPGA image of the USRP. This program downloads the required UHD images based on the USRP type, which is set as an environment variable. We made some changes to the mechanism of this part to be done offline without needing a direct Internet connection. We manually downloaded the required images according to our board type and included them as a volume in the pod definition file. We also disabled the environment variable that triggered the execution of the image_downloader script based on its value.

5) Version of eNB Images: As previously stated, we tested and utilized different images for the eNB module, each of which performs more effectively in a specific scenario. We selected the appropriate image based on the given scenario.

6) Testing Tools Preparation: We employed several tools to test the performance of the deployed network, including:

- Ping
- SCP: To compare download speeds, we used SCP to transfer a 1GB file.
- Streaming: We deployed an NGINX server close to the SPGW modules to serve an MP4 file and compare streaming quality in different scenarios.
- Wget: We also used wget to compare download speeds with the aforementioned NGINX server.

These tools have been specifically employed in the evaluation section to investigate the importance of VNF placement and RAN slicing and their effects on the network performance.

E. Mininet Bridges

In this framework, we address the effect of the distance between different components in a real network by utilizing SDN tools and the Mininet emulator and creating virtual networks between our devices. We name these virtual networks Mininet Bridge (MB). In this part, we present how an MB is created and what it consists of.

1) Mininet: To create an MB we require a tool capable of emulating real computer networks. Mininet is a lightweight network emulator which can set up virtual networks containing virtual hosts, switches, controllers, and links on an OS. Mininet takes advantage of Linux namespaces instead of virtualization resulting in a lighter emulation compared to VMs. Mininet's straightforward Python API enables the creation of complex and real-world topologies. Moreover, Mininet has a CLI in which users can manage, configure, and interact with the created virtual network. Mininet can utilize an SDN controller to emulate SDN networks. In a virtual SDN network created by Mininet, the switches are virtual OpenFlow switches capable of communicating with the controller using the OpenFlow protocol.

2) SDN controllers: There exist many SDN controllers with different capabilities (e.g., POX, Floodlight, and RYU) to serve as the controller connected to the virtual SDN network. In this framework, we used ONOS [19] and RYU [20] as the SDN controller of the MBs. ONOS and RYU are both popular open-source SDN controllers that support the OpenFlow protocol and have a large community backed by the Linux Foundation. RYU is Python-based, making it easier to set up and use, which results in more popularity among developers. ONOS is coded in Java and is more complex compared to RYU, but it has more features and can handle large-scale networks. Moreover, ONOS is a part of the Open Networking Foundation which is supported by many major vendors and leading companies in the telecommunication industry. It is worth mentioning that ONOS is more suitable for our framework as is more compatible with large-scale telecommunication.

3) Bridge: To create an MB we need a Linux OS with Mininet and ONOS/RYU installed on it. we create our topology with Mininet's Python API without hosts, and using OVS commands we connect the NICs of the device to the virtual switches. The use of MBs makes our framework more realistic and more similar to real communication networks. Moreover, MBs are considered transparent and are useful components in time of network management, measurement, and manipulation.

IV. EVALUATION

In this section, we use CN²F to investigate the importance of VNF placement and RAN slicing as two cellular network capabilities. The details on how to deploy test cases along with the required scripts to reproduce the results are available on the CN²F GitHub repository.

A. VNF Placement

Fig. 13 shows the CN²F setup to demonstrate the impact of VNF placement. Specifically, the test case incorporates two scenarios which are schematically represented in Fig. 14. In scenario 1 (Fig. 14a), the multi-media server (an NGINX stream module) is placed on worker node 3 which also hosts the core network's components. On the other hand, the client is inside the UE. Therefore, the data path between the client and server passes through the RRU (in worker node 1), the backhaul bridge (or TN), and the SPGW (in worker node 3). This scenario can be considered as a Cloud Computing (CC) scenario, because, in reality, worker node 3 may be placed in a data center accessed through an IP network (emulated by the backhaul bridge in this scenario). On the other hand, scenario 2 (Fig. 14b) depicts the case where the (NGINX) server and the SPGW are moved to worker node 1 beside the eNB. As a result, this scenario resembles Edge Computing (EC).
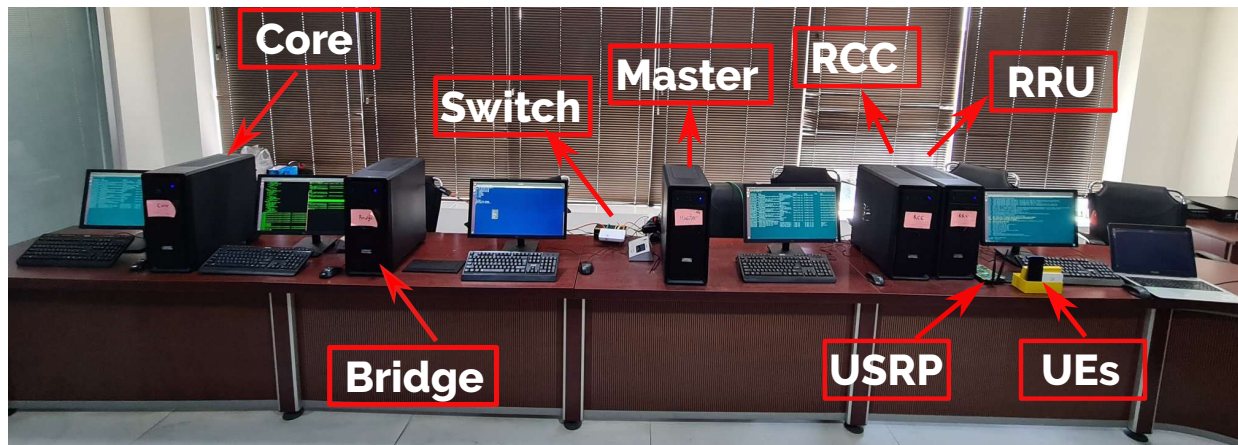
Fig. 13: CN^2F setup for the VNF placement evaluation

TABLE I: VNF placement results

Transport Network Parameters		Scenario 1 (CC)		Scenario 2 (EC)	
Bandwidth (Mb/s)	Delay (ms)	Bit Rate (Mb/s)	RTT (ms)	Bit Rate (Mb/s)	RTT (ms)
10	0	1.9	200	1.9	200
5	50	0.52	340	1.9	340

(a) Cloud computing

(b) Edge computing
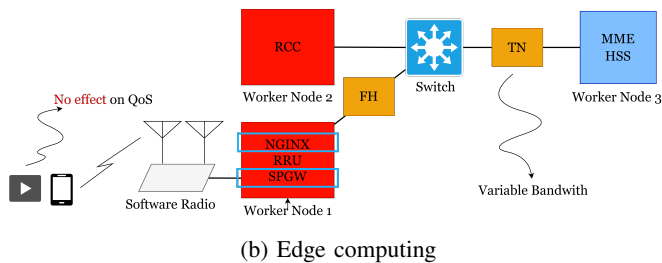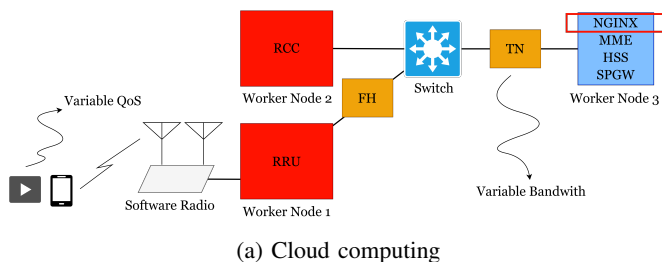
Fig. 14: Test cases for the VNF placement

TABLE II: RAN slicing results

No.	Scenario		Bit Rate (Mb/s)
	Resource Blocks	Device	
1st	5	UE 1	1.05
	20	UE 2	2.85
2nd	10	UE 1	1.40
	15	UE 2	1.95
3rd	15	UE 1	3.00
	10	UE 2	0.50

B. RAN Slicing

The idea of RAN slicing is to assign different numbers of resource blocks (RBs) to different network slices in order to comply with their corresponding service level agreements (SLAs) relevant to the access network. Using FlexRan along with eNB, we can test the effect of RAN slicing on the performance of the system (Fig. 15). In particular, Table II shows the result when we have two slices and one UE in each slice[6]. The total number of RBs is 25 which is divided between slice 1 (UE 1) and slice 2 (UE 2). As expected by allocating more RBs to each slice, the bit rate of its UE is increased. Specifically, the bit rate of UE 1 is 1.05 Mb/s when slice 1 has 5 RBs and it climbs up to 3 Mb/s as the number of RBs increases to 15.

V. RELATED WORK

Testbed-based evaluation in wireless networks is a prevalent approach used to assess realistic scenarios. The importance of testbeds grows when we come to 5G/B5G mobile networks

Table I shows the performance of the network in terms of the achievable bit rate (using the wget utility in the UE to download an MP4 video from the NGNIX server) and the Round Trip Time (RTT) to an external server (i.e., the Google's DNS server at "8.8.8.8") for different settings of the bridge. It is observed that the parameters of the transport network highly affect the performance of the system in the CC scenario. For instance, the bit rate is almost halved when the bandwidth is reduced from 10 Mb/s to 5 Mb/s. However, in the EC scenario, the bit rate is independent of variations in the transport network.

---

[6]In our setup, we use the USRP-B210 without any external/GPS clock and duplexer. Therefore, some COTS UEs face problems in detecting and connecting to our network. Among different UEs that we have tested, the Huawei-E5573-8739 and Huawei-Nova 3e can successfully connect to our network (both using Qualcomm Snapdragon 680 chipset)

TABLE III: Cellular network testbeds and frameworks

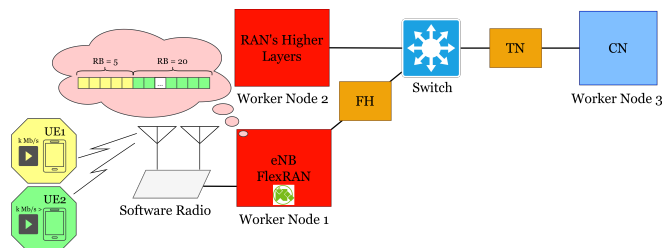| Testbed	SDN	NFV	E2E Slicing	MANO	Open-Source	ML-Enabled	Setup Details
[21], [22], [23], [24]	✓	✓	✗	✗	✗	✗	✗
[7]	✓	✓	✗	✗	✓	✓	✗
[25]	✓	✓	✗	✓	✗	✗	✗
[26]	✓	✓	✗	✓	✓	✗	✗
[27]	✗	✓	✓	✗	✓	✗	✗
[28]	✗	✓	✓	✓	✗	✗	✗
CN^{2}F	✓	✓	✗	✗	✓	✗	✓

Fig. 15: Test case for the RAN slicing

due to the complexity and different requirements of these networks. This section briefly mentions some developed testbeds for 5G networks and their capabilities.

In [21], the authors focused on the orchestration of the TN and RAN by suggesting a hierarchical cross-domain orchestrator that offers network programmability and flexibility. This orchestrator monitors the radio resources at the network access edge level, the transport resources at the access and aggregation levels across multiple domains, and the cloud resources at the network core level to make decisions. The authors demonstrated the advantages and feasibility of their proposed orchestration by implementing two use cases of SDN-based transport and RAN orchestration in a testbed. The first use case presents sharing of joint RAN-transport resources between two service providers (SPs), and the second one demonstrates how an SP can customize its own slice.

The authors in [22] presented the architecture and results of the ADRENALINE as a testbed, which is an SDN/NFV packet/optical transport network and edge/core cloud platform for end-to-end 5G and IoT services, deployed with open-source software and Commercial Off The Shelf (COTS) hardware. Similarly, the work in [23] provided an experimental setup of a convergent 5G service scenario involving IoT, cloud, and edge networks, all featured by SDN capabilities. The implemented testbed also includes an SDN-based orchestrator able to dynamically adapt data delivery paths based on the current load of network switches and links. Another testbed presented in [24] employs COTS components to embody an end-to-end 5G platform based on the CRAN architecture, with a fully virtualized RAN, an optical/wireless fronthaul, and a cloud-based backend. These approaches do not offer a complete end-to-end network slicing and the source codes needed to deploy the testbeds are not publicly available.

Another group of testbeds concentrates on Management and Orchestration (MANO) implementation. For example, BlueArch [25] is a 5G testbed providing a hybrid platform for conducting various experiments with different modes of tests

including simulation, emulation, and interaction with the physical network and remote testbed platforms. BlueArch supports ETSI MANO orchestration. The Open MANO and RIFT.io orchestrators are hosted as VMs within a XEN environment. Simula testbed [26] also implements a mobile network based on OAI-EPC deployed as a VNF using Open-Source MANO (OSM), which is integrated with CRAN architecture with functional split capability for BBU processing functions. As another example, in [28] the authors proposed an emulation framework for zero-touch 5G core network slicing management and orchestration that features closed loop automation. Their framework relies on use of OSM for NFV MANO functions with NFV orchestration and VNF management functionalities communicating with different Virtual Infrastructure Management (VIM)s.

On the other hand, the testbed in [27] implements end-to-end network slicing. However, it does not offer MANO capability, multi-RATs, and multi-tenancy facilities in the architecture. This testbed utilizes OAI for both RAN and CN domains. There are two CNs that share the radio resources of a single eNB in the RAN. The testbed has been appraised for connection establishment for both normal LTE UE and the one with an implemented Network Slice Selection Assistance Information (NSSAI).

Some other testbeds are presented by researchers with the objective of AI workloads orchestration to facilitate the deployment of AI agents into the testbed. In [7], Connected AI (CAI) is presented as a 5G mobile network testbed with a virtualized and orchestrated structure using containers. CAI focuses on integrating Artificial Intelligence (AI) applications using the Kubeflow tool, albeit it implements partial network slicing and does not incorporate MANO components. It also presents an emulated TN enabling the deployment of any network topology on fronthaul and backhaul, without needing access to actual transport network topologies.

In Table III, a comparison between our framework and the ones presented before is demonstrated. In this paper, we implemented CN^{2}F as an easy-to-setup, cloud-native cellular network framework, alongside a detailed description of how to build it. In addition, we provided a GitHub repository along with an *Ansible playbook*, which helps researchers and the industrial community to set up CN^{2}F on their systems to test and evaluate realistic network scenarios. We implemented VNF placement to showcase two different scenarios EC and CC. RAN slicing has been also implemented as another use case for new-generation cellular networks to demonstrate how efficient allocation of resources can improve the QoS for applications based on their demands. This framework takes

advantage of SDN and NFV capabilities, and its main structure is independent of specific VNFs.

VI. Conclusion

Cellular networks have evolved into fully virtualized and programmable networks. As a result, an innovative solution can find its path into operation as easily as some software updates in the network's components. The same virtualization nature of the modern cellular networks also enables building laboratory testbeds for the research and development teams to discover new services/products in isolated, yet close-to-field, environments. In this paper, we shared our findings and best practices in building such testbeds, called CN^2F, for modern cellular networks using state-of-the-art technologies such as Docker, Kubernetes, ONOS, and Mininet. We especially focused on how to set up a cluster of nodes, which host the cellular network's VNFs and the management entities, and bridges, which emulate the intermediate IP networks between different parts of a real cellular network. Thereby, CN^2F is capable of deploying and testing various scenarios such as RAN splitting/slicing, edge computing, and VNF placement. Moreover, for a particular open-source project, we walked through the process of installing the required packages, building Docker images and containers, creating pods, setting the configuration files, and deploying the cellular network's core and RAN VNFs on the CN^2F. Finally, the performance of the deployed network was further measured under various test case scenarios to evaluate the benefits of edge computing and RAN slicing.

References

[1] C. Bouras, A. Kollia, and A. Papazois, "SDN & NFV in 5G: Advancements and challenges," in *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*, 2017, pp. 107–111.

[2] C. Benzaid and T. Taleb, "AI-Driven Zero Touch Network and Service Management in 5G and Beyond: Challenges and Research Directions," *IEEE Network*, vol. 34, no. 2, pp. 186–194, 2020.

[3] A. Thantharate, R. Paropkari, V. Walunj, and C. Beard, "DeepSlice: A Deep Learning Approach towards an Efficient and Reliable Network Slicing in 5G Networks," in *2019 IEEE 10th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, 2019, pp. 0762–0767.

[4] M. E. Morocho-Cayamcela, H. Lee, and W. Lim, "Machine Learning for 5G/B5G Mobile and Wireless Communications: Potential, Limitations, and Future Directions," *IEEE Access*, vol. 7, pp. 137 184–137 206, 2019.

[5] R. Li, Z. Zhao, X. Zhou, G. Ding, Y. Chen, Z. Wang, and H. Zhang, "Intelligent 5G: When Cellular Networks Meet Artificial Intelligence," *IEEE Wireless Communications*, vol. 24, no. 5, pp. 175–183, 2017.

[6] T. E. Bogale, X. Wang, and L. B. Le, "Machine Intelligence Techniques for Next-Generation Context-Aware Wireless Networks," *ArXiv*, vol. abs/1801.04223, 2018.

[7] C. V. Nahum, L. De Nvoa Martins Pinto, V. B. Tavares, P. Batista, S. Lins, N. Linder, and A. Klautau, "Testbed for 5G Connected Artificial Intelligence on Virtualized Networks," *IEEE Access*, vol. 8, pp. 223 202–223 213, 2020.

[8] K. Tan, D. Wu, A. Chan, and P. Mohapatra, "Comparing simulation tools and experimental testbeds for wireless mesh networks," in *2010 IEEE International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*, 2010, pp. 1–9.

[9] B. Bashari Rad, H. Bhatti, and M. Ahmadi, "An Introduction to Docker and Analysis of its Performance," *IJCSNS International Journal of Computer Science and Network Security*, vol. 173, p. 8, 03 2017.

[10] E. A. Brewer, "Kubernetes and the Path to Cloud Native," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ser. SoCC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 167.

[11] "Mininet: An Instant Virtual Network on your Laptop (or other PC)," last accessed 1 April 2023. [Online]. Available: http://mininet.org

[12] "openair-epc-fed," last accessed 16 April 2023. [Online]. Available: https://github.com/OPENAIRINTERFACE/openair-epc-fed/blob/master/docs/DEPLOY_HOME_MAGMA_MME.md

[13] K. Ramantas, E. Kartsakli, M. Irazabal, A. Antonopoulos, and C. Verikoukis, "Implementation of an SDN-Enabled 5G Experimental Platform for Core and Radio Access Network Support," in *Interactive Mobile Communication Technologies and Learning*, M. E. Auer and T. Tsiatsos, Eds. Cham: Springer International Publishing, 2018, pp. 791–796.

[14] J. G. Andrews, S. Buzzi, W. Choi, S. V. Hanly, A. Lozano, A. C. Soong, and J. C. Zhang, "What will 5G be?" *IEEE Journal on selected areas in communications*, vol. 32, no. 6, pp. 1065–1082, 2014.

[15] L. Bonati, M. Polese, S. DOro, S. Basagni, and T. Melodia, "Open, programmable, and virtualized 5g networks: State-of-the-art and the road ahead," *Computer Networks*, vol. 182, p. 107516, 2020.

[16] "Project calico," last accessed 2 Jan 2023. [Online]. Available: https://www.tigera.io/project-calico

[17] "Oenairinterface github," last accessed 2 Jan 2023. [Online]. Available: https://github.com/openairinterface

[18] "Urecom gitlab," last accessed 2 Jan 2023. [Online]. Available: https://gitlab.eurecom.fr/mosaic5g/mosaic5g

[19] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "ONOS: Towards an Open, Distributed SDN OS," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 16.

[20] S. Asadollahi, B. Goswami, and M. Sameer, "Ryu controller's scalability experiment on software defined networks," in *2018 IEEE International Conference on Current Trends in Advanced Computing (ICCTAC)*, 2018, pp. 1–5.

[21] A. Rostami, P. Ohlen, K. Wang, Z. Ghebretensae, B. Skubic, M. Santos, and A. Vidal, "Orchestration of RAN and Transport Networks for 5G: An SDN Approach," *IEEE Communications Magazine*, vol. 55, no. 4, pp. 64–70, 2017.

[22] R. Muoz, L. Nadal, R. Casellas, M. S. Moreolo, R. Vilalta, J. M. Fbrega, R. Martnez, A. Mayoral, and F. J. Vlchez, "The ADRENALINE testbed: An SDN/NFV packet/optical transport network and edge/core cloud platform for end-to-end 5G and IoT services," in *2017 European Conference on Networks and Communications (EuCNC)*, 2017, pp. 1–5.

[23] S. Fichera, M. Gharbaoui, P. Castoldi, B. Martini, and A. Manzalini, "On experimenting 5G: Testbed set-up for SDN orchestration across network cloud and IoT domains," in *2017 IEEE Conference on Network Softwarization (NetSoft)*, 2017, pp. 1–6.

[24] K. Ramantas, A. Antonopoulos, E. Kartsakli, P.-V. Mekikis, J. Vardakas, and C. Verikoukis, "A C-RAN Based 5G Platform with a Fully Virtualized, SDN Controlled Optical/Wireless Fronthaul," in *2018 20th International Conference on Transparent Optical Networks (ICTON)*, 2018, pp. 1–4.

[25] S. Ghosh, E. E. Ugwuanyi, T. Dagiuklas, and M. Iqbal, "BlueArch-An Implementation of 5G Testbed," *J. Commun.*, vol. 14, pp. 1110–1118, 2019.

[26] T. Dreibholz, "Flexible 4G/5G Testbed Setup for Mobile Edge Computing Using OpenAirInterface and Open Source MANO," in *Web, Artificial Intelligence and Network Applications*, L. Barolli, F. Amato, F. Moscato, T. Enokido, and M. Takizawa, Eds. Cham: Springer International Publishing, 2020, pp. 1143–1153.

[27] A. Shorov, "5G Testbed Development for Network Slicing Evaluation," in *2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, 2019, pp. 39–44.

[28] S. Vittal, S. Sarkar, P. P S, and A. F. A, "A Zero Touch Emulation Framework for Network Slicing Management in a 5G Core Testbed," in *2021 17th International Conference on Network and Service Management (CNSM)*, 2021, pp. 521–523.