THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

CoreKube: An Efficient, Autoscaling and Resilient Mobile Core System

Citation for published version:
Ferguson, AE, Larrea, J & Marina, MK 2023, CoreKube: An Efficient, Autoscaling and Resilient Mobile Core System. in *The 29th Annual International Conference on Mobile Computing and Networking.*, 25, ACM Association for Computing Machinery, pp. 1-15, The 29th Annual International Conference On Mobile Computing And Networking, Madrid, Spain, 2/10/23. https://doi.org/10.1145/3570361

Digital Object Identifier (DOI):
10.1145/3570361

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
The 29th Annual International Conference on Mobile Computing and Networking

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 10. Aug. 2024

CoreKube: An Efficient, Autoscaling and Resilient Mobile Core System

Andrew E. Ferguson[†], Jon Larrea[†], Mahesh K. Marina
[†]Co-primary authors
The University of Edinburgh

ABSTRACT

Given the central role mobile core plays in supporting mobile network operations, the efficiency, cost-effective dynamic scalability and resilience of the core control plane are paramount. Achieving these goals, however, presents two main challenges: (i) decoupling core network state from processing; (ii) decoupling control plane processing in the core from its interface to the radio access network (RAN). To overcome them, we present CoreKube, a novel message focused and cloud-native mobile core system design, which features *truly stateless workers* (processing units) that interface with a common database (to hold the core network state) and with the RAN through a frontend. The fully stateless and generic nature of the workers to process any control plane message enables efficient message handling. Orchestration of containerized CoreKube components using Kubernetes, allows leveraging the latter's autoscaling and self-healing properties. We develop 4G and 5G standard-compliant CoreKube implementations, exploiting the agile development methodology enabled by CoreKube's message focused design. Results from our extensive experimental evaluations over the Powder platform relative to prior art show that CoreKube efficiently processes control plane messages, scales dynamically while using minimal compute resources and recovers seamlessly from failures.

ACM Reference Format:
Andrew E. Ferguson[†], Jon Larrea[†], Mahesh K. Marina. 2023. CoreKube: An Efficient, Autoscaling and Resilient Mobile Core System. In *The 29th Annual International Conference on Mobile Computing and Networking (ACM MobiCom '23), October 2–6, 2023, Madrid, Spain.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3570361.3592522

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ACM MobiCom '23, October 2–6, 2023, Madrid, Spain
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9990-6/23/10…$15.00
https://doi.org/10.1145/3570361.3592522

1 INTRODUCTION

The focus of this paper is on the mobile core, the nerve center for operating a mobile network, that handles the essential control functionality (device authentication, mobility management, etc.) and bridges data communication between end devices and external networks like the Internet via the radio access network (RAN). Motivated by the need to reduce costs and increase flexibility, there has been a recent trend in mobile networking towards virtualizing network functions over commodity hardware and deployment in cloud environments. This is particularly the case for the mobile core as it is inherently less coupled with essential physical hardware like radio equipment in the RAN. This is reflected through the recent developments in the mobile networking industry (e.g., [22, 25, 69]), emergence of multiple open source projects (e.g., [28, 36, 45]) and from the focus of alternative core system designs in the research literature (e.g., [20, 48, 49, 51, 53]).

Virtualized and cloud based mobile core systems need to meet several key requirements that are largely linked to the control plane. Scalability is chief among them, given the anchor role that the mobile core performs serving millions of users or devices differing in their access patterns, as also noted by prior work (e.g., [20, 51, 60]). Crucially, scaling with control plane workload needs to be achieved in a way that optimizes the operational costs. This is the case for both public and private cloud deployments, and translates to efficient and adaptive use of computational resources proportional to the core signaling load. We refer to this dynamic scaling requirement as *autoscalability* in this paper. Another key requirement is to ensure efficient and responsive control plane processing from the user/device perspective, regardless of the load on the core. Moreover, given the critical role the core network plays in the operations of mobile networks, its resilience in the face of failures (in other words, self-healing) is vital. Last but not the least, it is desirable to have a mobile core system design that meets the above requirements while at the same time is easily implementable and is compliant with the standards specifications (in terms of functional equivalence and interface to the RAN and end devices).

Meeting the above requirements, however, is far from straightforward and in fact, presents two significant technical challenges. To see these challenges, we examine the scalability limitation of a simple-minded approach. Specifically, we consider the 3GPP standard 5G core that is disaggregated

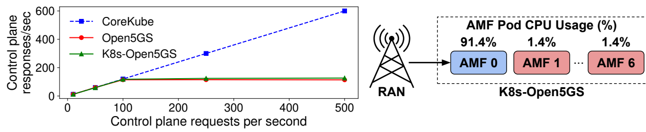

Figure 1: (a) Scalability comparison of Open5GS, cloud-native Open5GS (K8s-Open5GS) and CoreKube; (b) Relative utilization among AMF instances with K8s-Open5GS.

into multiple different functions (AMF, SMF, etc.) [3], as outlined in §2.1. Fig. 1a compares the scalability of two variants of Open5GS [45] (a popular open source implementation of the standard 5G core that is also being used in commercial networks) with increasing control plane signaling load. The vanilla variant (shown simply as 'Open5GS' in the figure) is deployed on a dedicated physical machine whereas the 'K8s-Open5GS' variant containerizes each of the 5G core functions (e.g., AMF) in Open5GS and orchestrates them using Kubernetes [27] on a compute cluster. We observe that both these variants not only quickly hit their scaling limit but also have hardly any noticeable difference between them, even though we would expect the K8s variant to scale better. We find that this is because of the highly stateful nature of standard 5G core functions (that Open5GS literally implements), which impedes the scaling benefit that a Kubernetes-driven cluster/cloud deployment can provide. For instance, the processing related to the Attach event 'touches' state at least 902 times (or equivalently, 180+ touches per signaling message of the Attach event to be processed by the core). *This highlights our first challenge, **Challenge 1**: break free from the heavily entangled nature of processing and state in standard core functions/events.*

The second challenge arises due to the close coupling in the RAN-core interface, specifically the N1/N2 (NAS/NGAP) interfaces between the gNB and AMF which are established upon gNB setup and the connection remains fixed throughout the gNB lifetime. As the AMF is the other endpoint of this connection, scaling the AMF directly is impossible, as any newly-scaled instances will not be connected to the RAN and therefore unable to receive traffic. We demonstrate this in Fig. 1b, where the load to the K8s-Open5GS variant is increased. Although K8s detects the increased load and deploys more AMF instances, only the original AMF instance (with the connection to N1/N2 interfaces) receives and processes any traffic, while the newly-deployed instances stay idle. *This highlights our second challenge, **Challenge 2**: decouple the RAN-core interface from control plane processing in the core.*

Motivated by the above, we propose CoreKube, a novel *message focused* and cloud-native mobile core system architecture. Our key insight underlying CoreKube design to address the above challenges is that control plane processing in the core is best viewed at the finest granularity of individual control plane messages. Note that 3GPP standard core specifications organize the core into coarse-grained functions dealing with different aspects (AMF, SMF, etc.). These core functions collectively handle various control plane events (e.g., attach, detach, handover). Each of these events (e.g., attach) in turn involve a sequence of messages (e.g., Attach Request, Authentication Request, Authentication Response) to be processed by the core. This message-level processing in the core has a simple and generic form, regardless of the event being handled: upon receiving an incoming message (request) from a device (UE), access/modify the corresponding UE state, then as needed generate and send back a response message. Based on this observation, in CoreKube we logically consolidate the handling of *all* messages into a generic worker (processing unit) while decoupling all the core network state into a separate database, thereby addressing challenge 1. A consolidated worker, rather than individual workers for each message, is essential for this because it removes the implicit state needed to forward messages to the correct, separate worker. Besides, we introduce a frontend at the RAN-core interface in CoreKube that encapsulates/decapsulates messages from/to the RAN to decouple control message processing in the core from the RAN-core interface, i.e., to address challenge 2. This also aids in maintaining a standard-compliant interface to the RAN.

In essence, CoreKube features *truly stateless workers* which interface with a common database that holds the core network state and with the RAN through a frontend. The completely stateless and generic worker allows any worker to handle any control plane message from any end device, thereby enabling efficient message processing with minimal user perceived latency regardless of the core's control plane workload. Containerizing the worker, database and frontend components of CoreKube leads to a cloud-native mobile core architecture that can be *ideally* mapped to Kubernetes [27] and fully inherit the latter's autoscaling and self-healing features.

Moreover, the message focused approach of CoreKube enables an agile development methodology, which allows rapid development of CoreKube oriented implementation of a given standards based mobile core specification (e.g., 4G, 5G). This is owing to the fact that we redefine the notion of microservices for the mobile core to be more fine grained at the level of individual control plane messages, allowing them to be independently implemented and state iteratively added. Employing this methodology, we develop standard-compliant 4G and 5G CoreKube implementations with relative ease. We experimentally evaluate these implementations over Powder [18] relative to existing open source 4G/5G standard-compliant core network implementations as well as representative state-of-the-art virtualized mobile core system designs [20, 53]. Our results clearly demonstrate the standard compliance, efficiency, autoscaling and resilience features of CoreKube relative to these alternatives.

In summary, we make the following key contributions:

- We present CoreKube, a new message focused and cloud-native mobile core system architecture design. To the best of our knowledge, CoreKube is the first virtualized mobile core system design in the literature (discussed in detail

in §6) to realize truly stateless processing and use it to leverage the full power of Kubernetes orchestration system for autoscaling, efficient control plane message handling and resilient operation (§3).

- The message focused CoreKube design additionally enables an agile development methodology that allows rapid development of standard-compliant CoreKube-oriented core network implementations. We demonstrate this capability by developing 4G and 5G standard-compliant CoreKube implementations (§4). To our knowledge, ours are the first dynamically scalable 4G/5G core implementations in the literature. We intend to open source all CoreKube components as well as the Powder profile used in the experiments to benefit the research community at https://github.com/netsys-edinburgh/CoreKube.
- We experimentally evaluate CoreKube against existing open source standard-compliant 4G/5G core network implementations and representative state-of-the-art approaches, and show significant improvements in responsiveness, scalability, resource efficiency, and resilience against failures (§5). We also quantify the benefit due to key choices underlying the CoreKube design.

2 BACKGROUND

2.1 4G & 5G Standard Core Architectures

Generally speaking, a mobile network is made up of two parts: the Radio Access Network (RAN) and the mobile core network. The RAN consists of a set of base stations, each with antennas, radio modems and digital processing modules to allow the base station to wirelessly interact with end devices – called user equipments (UEs), and forward traffic to the mobile core. The mobile core network is responsible for managing and facilitating the network as a whole: from handling connections of the UEs, their sessions and mobility, managing subscriber information and authentication, and forwarding data traffic between UEs and other networks such as the Internet.

The 4G (LTE) core network or the Evolved Packet Core (EPC), introduced in 3GPP Release 8 [1], consists of four primary components (Fig. 2a). The Mobility Management Entity (MME) is responsible for coordinating all signalling exchanges between the UEs, the base stations (called eNodeBs (eNBs) in LTE) and the EPC. It handles every control plane event (authentication, mobility management, session establishment, etc.) using the S1 Application Protocol (S1AP) and Non-Access Stratum (NAS) protocols running over SCTP to respectively interface with eNBs and UEs in the RAN. The Home Subscriber Server (HSS) is a database containing all the relevant data (IDs, keys, etc.) for every subscriber of the network service.

The Packet Data Network Gateway (P-GW) allows UEs to access the Internet, assigns IP addresses to UEs, and provides Network Address Translation (NAT) when IPv4 is used. The Serving Gateway (S-GW), on the other hand, is responsible

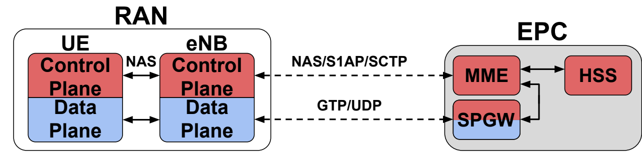

Figure 2.a) 3GPP standard 4G (LTE) system architecture.

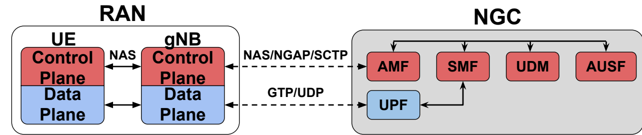

Figure 2.b) 3GPP standard 5G system architecture.

for managing the data tunnels used between eNBs and P-GW. In practice, S-GW and P-GW are realized together, thus shown as SPGW in Fig. 2a. The EPC control plane spans MME, HSS and SPGW with MME playing a central role, whereas its data plane functionality is limited to SPGW.

The 5G core network called the Next Generation Core (NGC) was standardized through the 3GPP Release 15 [3]. The components in the NGC (Fig. 2b) largely perform the same functions as those in the EPC but with greater function disaggregation and a clear control-data plane separation. MME functionality in EPC is split between the Access and Mobility Function (AMF) and the Session Management Function (SMF) in NGC. Similarly, HSS in EPC is decomposed into two functions in NGC, namely: the Unified Data Management (UDM) function and the Authentication Server Function (AUSF). 3GPP NGC specification also defines two optional mechanisms called the AMF set [8] and the UDSF [9] to help distribute the load coming to the core. The AMF set is a RAN feature meant for balancing load across multiple AMF instances and requires the RAN to keep track of the AMF instance serving each UE. UDSF, on the other hand, is a temporary database used for planned removals of an AMF instance from an AMF set.

Only the data plane functionality of SPGW in EPC is retained in the User Plane Function (UPF) in NGC while the rest is moved to the control plane. This clear control–data plane separation allows for their independent scaling, placement and evolution. In the 5G RAN, gNB is equivalent to the LTE eNB. The Next Generation Application Protocol (NGAP) replaces S1AP but with similar functionality and still runs over SCTP. NAS, in an updated form, is still used for control plane communication between UEs and the core. From the core network viewpoint, NAS and S1AP/NGAP protocol messages are processed sequentially, and moreover UEs are treated independently, as detailed in their respective specifications [2, 4, 6, 7].

2.2 Containers and Kubernetes

For orchestrating virtualized applications and services, containers are preferred over virtual machines (VMs) as they are lightweight [26, 65] and have low startup times [19].

Kubernetes (K8s) [27] is the most popular open source container orchestration system [21]. Its popularity offers significant flexibility and portability since most of the major cloud providers offer some form of Kubernetes deployment [13, 30, 47], reducing the chance of vendor lock-in and allowing for public cloud provider selection based upon criteria such as cost, location and legal jurisdiction. Equally, it allows for deployment over a wide range of private cloud configurations [41, 57]. Crucially, Kubernetes offers built-in autoscaling capabilities, allowing adaptation of the amount of computational resources used according to the demand. It also provides self-healing capability to detect and restart subsystems that are performing abnormally. Consequently, Kubernetes is well matched for CoreKube design compared to other container orchestration tools such as Docker Swarm [35] and Docker Compose [34] that are geared mainly for smaller scale, self-managed systems.

Kubernetes introduces several concepts for container orchestration of which three are particularly relevant:

- *Pod* is a wrapper around one or more containers to allow it to be orchestrated. Kubernetes monitors deployed pods and will restart them automatically when they crash.
- *Services.* A uniform network address to a deployment of one or more pods. External systems can interact with the pods through the service, without being concerned about the number or network addresses of the individual pods; the service also provides load balancing across the pods.
- *Horizontal Pod Autoscaler (HPA)* controls the pod deployment by increasing or decreasing the number of pods to reach a target metric, such as the average CPU utilization across the pods.

3 COREKUBE DESIGN

Here first outline our design goals and associated challenges, then give an overview of the proposed CoreKube design and follow it up with a description of the details.

3.1 Goals

Autoscaling. The scaling challenge presented by growing number of devices and control signaling traffic to virtualized mobile core systems is well recognized in the literature (e.g., [20, 49, 51, 60]). Crucially, we would like a mobile core that efficiently scales such that resources used for control plane processing are proportional to the volume of signaling traffic being handled, as opposed to one that scales well but also has a high resource usage. This is particularly important for mobile core deployments over public clouds, where capacity is elastic but the cost of the deployment is dependant on the amount of resources used [16, 52]. This is also important in private cloud and on-premise deployments where inefficient resource use manifests in the form of increased OPEX. Autoscaling is particularly important in scenarios where control plane traffic load can significantly vary over time, such as private networks in office parks and shopping centers where daytime footfall is typically significantly higher than at night. Being able to dynamically scale the compute resources used up or down depending on the variations in control plane signaling traffic load is therefore important.

Efficiency. Mobile core networks must be able to process control plane messages with minimum latency. Delays in processing individual control plane messages and in turn events can have a negative impact on the data plane performance and user experience. A key goal of our design therefore is to efficiently process control plane messages with minimal overhead so that the core is responsive in a consistent way to the end devices regardless of the total amount of control plane traffic being processed.

Resilience (Self-Healing). Mobile networks, including the core, have stringent reliability (five 9s) and high availability (less than 1s outage time) requirements [24, 53]. The traditional approach to achieve these through specialized appliances is expensive to scale while doing so with virtualized core systems over commodity hardware in a public/private cloud environment is a relatively new requirement. Avoiding failures altogether is unrealistic, especially in a software based system. Our goal in this regard is to be able to quickly detect abnormal or failure events and seamlessly adapt to them with minimal or no perceived downtime.

Standard-compliant RAN interface. The requirements of operational mobile networks and constraints posed by virtualized and cloud environments necessitate mobile core system designs that deviate from standards specifications in the way any given functionality is realized (e.g., [20]). But these internal differences in the design should be invisible outside the core for widespread deployability. As such, in our design we aim to conform to a standards based interface to the RAN (eNBs/gNBs and UEs).

Ease of Development. As noted above, mobile network standards specification bodies like 3GPP specify the functionality of different components including the core [1, 3]. The development time and engineering effort involved in going from first implementing those functions (e.g., AMF, MME) from the standard then to realizing a system that satisfies the above goals (e.g., autoscaling) is a slow and expensive process. So a design that is amenable to directly realizing a mobile core system satisfying all the above properties from the specifications will greatly ease development. Such a design will be even more valuable if it is generic and agnostic to the specific generation of the standard (4G, 5G, etc.) as it can then better support the technology evolution.

3.2 Challenges

1) Decoupling core network state from control plane processing. As noted earlier in §1, the approach taken in the 3GPP standard core architecture to couple the state and processing together limits dynamic scaling with control plane workload variations. This is because state inconsistency issues arise with coupled processing and state, when the number of processing instances are scaled up or down in response

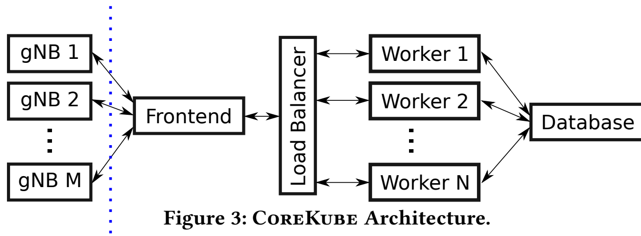

Figure 3: CoreKube Architecture.

to load variations. For the same reason, such stateful processing also limits resilience in the face of failures.

However, decoupling state from processing needs to be done with efficiency in mind. Naively having a processing instance (which we simply refer to as a 'worker') interact with the entity holding the state (which we call the 'database') *on every access* introduces a significant performance penalty. The challenge therefore is to determine the right granularity with which workers efficiently access the database. There is also the companion issue of determining the scope of processing for each worker. For instance, simply having different types of workers corresponding to the different functions in the 3GPP standard core architecture (e.g., AMF, SMF) introduces inefficiency (additional processing latency) as it would require 'routing' between worker types in the core to process a control plane event (e.g., attach).

2) Decoupling control plane processing in the core from RAN interface. As outlined in §1, 3GPP standards mandate that RAN entities communicate with the core through N1/N2 interfaces over a connection anchored between the gNB and AMF. Unless control plane processing in the core is decoupled from this connection with the RAN, dynamic scaling in the core while ensuring standards-compliant interface to the RAN is impossible, as demonstrated previously in Fig. 1b. So the challenge here is to decouple processing in the core from the RAN interface in such a way that the standard compliance of the latter is maintained.

3.3 Overview

Our proposed design, CoreKube, addresses the above two challenges to meet all the aforementioned goals. In the following, for concreteness, we present CoreKube in the context of 5G even though our design is generally applicable.

3.3.1 *Key Insights.*

We first present our insights that ultimately shape our design, which are obtained from a deeper look at the nature of control plane processing as well as the core network state. Control plane processing in the core can be seen from three different levels of increasing granularity: (i) functions (AMF, SMF, etc.); (ii) events (e.g., attach, detach, handover); and (iii) messages (e.g., Attach Request, Authentication Request, Authentication Response). Note that the 3GPP standard core architecture is organized at the coarse-grained function level, whereas several prior designs (e.g., [14, 51, 61]) are structured at the event level. In contrast, we find that, among these three levels, the message level offers the right granularity to decouple the core network state from processing in light of the following observations.

First, messages exhibit a common pattern of processing across all events and functions. Processing of messages has a generic form: receive request related to a UE from the RAN, access/modify the corresponding state and generate response(s) if necessary for the request in question. Second, control plane messages corresponding to each UE that access/modify its state are *serialized* (see messages for attach event at [67], for example). Third, we discover that the control plane state in the core has a certain structure that aligns with message as the unit of processing. In particular, core network state for each UE is *independent* from that of other UEs. Fourth, focusing on the message level allows logically consolidating handling of messages covering the whole core control plane into a generic worker without event or function boundaries, thereby avoiding the need for routing between workers. The inherent serialization of messages for a given UE makes it possible for any worker to handle messages for that UE even when its state is consolidated with that of other UEs into a common database. Moreover, messages for different UEs can be concurrently handled within and across workers. Finally, messages also offer the right granularity to allow decoupling the core control plane processing from the RAN interface through encapsulation of each SCTP based request message from the RAN so that any worker can handle messages from any UE. Decapsulation to recover a corresponding SCTP response message can be similarly carried out in the reverse direction to the RAN.

3.3.2 *Architecture.*

Fig. 3 illustrates the CoreKube architecture which is made up of three main components: a frontend, a pool of workers, and a database (DB), all containerized. The frontend acts as an intermediary between the gNBs and the workers by maintaining a standard-compliant persistent connection to the gNBs, while on the other side forwarding encapsulated messages to a simple load balancer which fairly distributes the messages among the workers. The worker is a completely stateless component that contains all the control plane logic (AMF, SMF, AUSF). It processes control plane messages received from the RAN, updates the corresponding database state and, if appropriate, sends response(s) back. The database acts as the common store for the information about the UEs and gNBs that connect to the core. It stores not only the permanent data such as the IMSI and operator key that UDM holds in the standard core architecture, but also transient information such as the NAS keys and sequence numbers that are maintained within different functions (e.g., AMF) in the standard core.

The above outlined design of CoreKube, the principal contribution of this paper, effectively realizes the functionality of a 3GPP standard core network architecture through a message focused design that mimics a typical web services architecture, thereby achieves efficiency, dynamic scaling

and resilience properties that are not satisfied by the standard architecture. Central to the design of CoreKube is the truly stateless and generic worker that performs control plane processing for *any* message. Truly stateless workers enable efficient handling of control plane messages with minimal inspections to keep the user perceived latency to the minimum regardless of the core's control plane workload. This combined with the fact that all CoreKube components are containerized enables ideal mapping to the Kubernetes orchestration system and fully leverage the latter's autoscaling and self-healing capabilities. The message focused CoreKube design additionally enables an agile development methodology that allows rapid development of standard-compliant and CoreKube-oriented core network implementations.

3.4 Message Focused Design

In our design, the control plane processing is done by a stateless worker that operates at the finest granularity of individual messages, regardless of which UE they correspond to. This message focused approach is consistent with 3GPP specified protocols for RAN interaction with the core – NGAP [4], S1AP [2] and NAS [6, 7] – which are structured around the messages that the core must handle. Control plane state changes in the core are a result of message interactions, rather than being the driving force behind them. Approaching the control plane processing in the core this way allows realizing a truly stateless architecture, focusing on the messages expected to be received and the responses that should be sent in reply.

3.4.1 *Truly Stateless Workers.*

In CoreKube, all the standard core network functions are *logically* consolidated into one generic network function processed by a worker instance. The worker instances that make up the worker component are *truly* stateless, meaning that they do not retain any state in between the processing of control plane messages. This allows each worker instance to receive and process any message from any UE and in any order. This consolidation removes the implicit statefulness that links individual messages to individual workers, simplifying the orchestration since the consolidated worker can be scaled based on its load, independently of the type of incoming messages. For example, in a stateful 3GPP standard core architecture it is the job of the AMF to receive control plane traffic, while the AUSF must generate the authentication parameters required by the Registration event. In contrast, the CoreKube stateless worker combines both the job of processing control plane messages and the generation of authentication parameters within the same worker function. Additionally, the message processing latency is improved by eliminating the additional step of directing messages to the correct type of worker.

The CoreKube worker architecture (illustrated in Fig. 4a) is composed of five main components:

(1) A listener, to receive and respond to messages from the frontend.
(2) NGAP input modules, each designed to process a different type of upstream NGAP message from the gNB.
(3) NAS input modules, each designed to process a different type of upstream NAS message from the UE.
(4) NAS output modules, each designed to generate a different type of downstream NAS message for the UE.
(5) NGAP output modules, each designed to generate a different type of downstream NGAP message for the gNB.

When the listener receives an upstream message from the frontend, it sends it to the main NGAP handler module, where it is decoded. The message is then processed by the appropriate NGAP handling module. If the message contains a NAS message from a UE, the NAS message is sent to the main NAS handler module, where it is decoded. Handling of the decoded message, which includes fetching and storing any required state in the DB, is done by the appropriate NGAP/NAS handling module.

If a NAS response (downstream message) is required, the appropriate NAS builder module is called. Similarly, the NGAP response is generated using the appropriate NGAP builder module. The listener then forwards the downstream message back to the frontend. In the case where no response is required, the NGAP / NAS builder modules will simply not be called. Separately, in the case where the upstream NGAP message contains no NAS message, the NAS handling modules will be bypassed and a downstream NGAP response will be generated if needed.

Despite containing the message processing pipelines for all message types, only the ones actively processing messages consume (CPU/memory) resources. As such, a consolidated worker does not cause inefficient resource usage.

3.4.2 *Database (DB).*

The database acts as a memory for the stateless worker, storing both permanent and transient information. In this context, permanent information is the data that is never changed by the worker and exists for the lifetime of the core such as the IMSI for each UE and operator keys. In contrast, transient information (e.g., NAS keys and sequence numbers for a particular UE) exists only temporarily, but crucially must persist beyond a single message. We observe that the core network state kept in the database has a simpler structure and access patterns. In particular, the data for each UE is independent (and uniquely identified using its IMSI) from that of other UEs and so can be accessed concurrently without any conflict. For any given UE, the 3GPP specified control plane protocols structure the messages to the core such that data accesses are strictly sequential so the possibility of simultaneous writes or simultaneous read-writes of the UE data does not arise, as there is only one message from a UE to be processed at any given point in time. Therefore a simpler and tailored approach can be taken to realize the

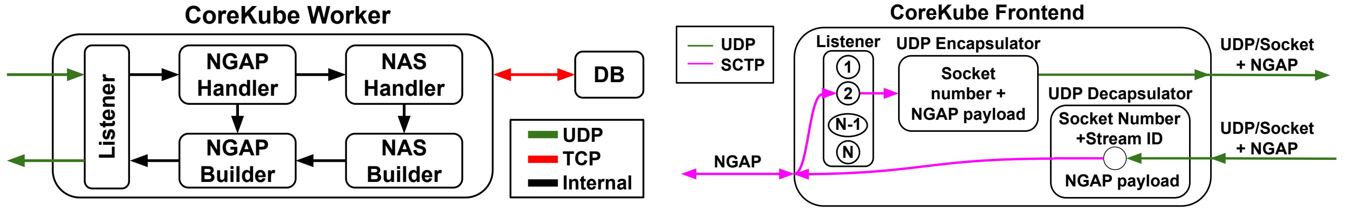

Figure 4: (a) CoreKube Worker Architecture; (b) CoreKube Frontend Architecture.

database in this setting instead of using a general purpose database.

Note that in CoreKube, we make sure that a worker-DB interaction is efficient when processing a message. Specifically, a worker instance handling the message fetches the corresponding UE state upon receiving a message, and stores any updated state once it is done with processing the message. This limits the number of DB transactions to two per message, which is the minimum.

3.4.3 Frontend.

Frontend is intended to address our second challenge of decoupling RAN interface from the control plane processing in the core. This decoupling is constrained to maintain the standard compliance of the RAN interface. To this end, the frontend receives/sends messages from/to the RAN over the connection-oriented SCTP protocol as per the standard but communicates internally with the stateless workers using a message-oriented UDP protocol via a simple load balancer that fairly distributes the messages among the workers. Specifically, the frontend achieves this by encapsulating upstream messages to the worker component, and decapsulating the returning downstream messages, as depicted in Fig. 4b.

Upstream NGAP messages are received from a gNB by a dedicated thread (one thread per gNB). The message is then encapsulated by appending the gNB socket number to the message. The encapsulated message is then forwarded to the load balancer over UDP. Downstream NGAP messages are received over UDP from the worker component. Such messages also contain the gNB socket number from the corresponding upstream message. The frontend uses this socket number to forward the downstream message to the correct gNB. Additionally, the downstream message encapsulation includes the SCTP stream ID, which is required to differentiate between UE and non-UE messages [5]. Although the majority of downstream messages should be routed to the gNB that sent the corresponding upstream message, the frontend has the ability to modify the socket number of the downstream message and therefore flexibly forwards the message. This is useful, for example, during the N2 Handover, when messages from the source gNB should be routed to the target gNB, and vice-versa.

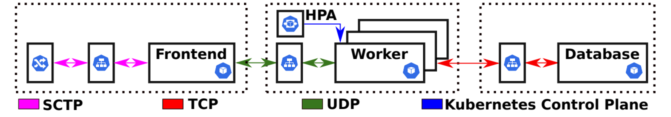

Figure 5: CoreKube over Kubernetes.

3.4.4 CoreKube over Kubernetes.

The CoreKube architecture easily lends itself to mapping onto the Kubernetes framework. Fig. 5 illustrates this mapping. It is split into three services: one each for the frontend, worker and database. The frontend provides the entrypoint for SCTP traffic from the RAN. A single pod runs the container with the frontend, while a corresponding service provides a uniform network address for incoming traffic. The frontend forwards the SCTP messages it receives from the connected gNBs to the worker service.

The worker service receives messages from the frontend over UDP, which integrates well with the default load-balancing provided by Kubernetes [58] to ensure a fair distribution of requests among the workers. The worker component itself is hosted across two or more pods, accessed from behind the uniform network address provided by the corresponding service. A HPA is used to increase and decrease the number of worker pods according to the average CPU utilization of the pods in the cluster. We choose average pod CPU utilization as the scaling metric since it captures the overall load on a pod agnostic to the messages processed within it. Relying on message specific metrics instead would be complex, as the great degree of diversity in the processing requirements of different messages has to be accounted for.

The database is hosted in a pod and has a corresponding service to provide a network address for accessing it. This service receives database requests from the worker pods. Such requests are handled over TCP, because this allows for a single, persistent connection between each active worker pod and the database. Note that we opt to containerize all CoreKube components and deploy over Kubernetes, rather than just a subset such as the worker cluster, to leverage the self-discovery and deployment capabilities of Kubernetes. Self-discovery enables automatic connections between the CoreKube components, without the need to manually update configuration files with hard-coded network addresses. Moreover, this makes the CoreKube highly portable for deployment over any Kubernetes-enabled cloud environment. We believe this is yet another positive aspect of our design.

3.4.5 *Development Methodology.*

Although we presented CoreKube design in the context of 5G mobile core network above, the design in fact is generic and agnostic to the specific generation of the standard (4G, 5G, etc.), allowing it to support the technology evolution. Crucially, the message-focused design of CoreKube greatly eases development of CoreKube implementation for a given 3GPP mobile core standard specification. This development is not only faster than developing a direct stateful implementation of the same 3GPP standard core specification following typical engineering approaches but also additionally provides efficiency, autoscaling and self-healing capabilities.

The handling of a control plane message is inherently independent from that of other messages with no direct interaction between them. Therefore independently implementing multiple message processing pipelines is possible. The independence between message processing pipelines is the enabler for allowing a consolidated worker to be formed from the individual pipelines, as it allows them to be combined without risk of conflicts or interference. This independence also applies to the state accessed during each message processing, allowing an incremental implementation of the state. This approach has several benefits, including fewer faults arising from inter-dependencies and early detection/correction of bugs in the development process from testing/evaluation at a per-message level.

Indeed, the CoreKube design redefines the notion of micro services in the core context to be at the fine-grained message level, instead of the coarser function level as defined in the 3GPP standard core specification. Moreover, it promotes truly multi-vendor implementations, as each message pipeline can be from a different vendor.

4 IMPLEMENTATION

Following the agile development methodology of CoreKube described in §3.4.5, we have developed CoreKube implementations for both 4G and 5G from their respective 3GPP standard core network specifications. Both these implementations span the three main components of CoreKube architecture: Frontend, Worker and DB. Additionally, the protocol used between the worker and the DB has been implemented as a separate library. Each component has been containerized using Docker [33]. All the components of CoreKube are written in C (overall 14K LOC).

4.1 Frontend

The main function of the frontend is to maintain the persistent SCTP connections with the RAN (one SCTP connection per eNB/gNB) and forward traffic over UDP to the worker cluster and vice-versa. The frontend maintains a pool of SCTP sockets, each of them with one thread listening to upstream packets. Once an upstream packet is received, the frontend extracts the payload and crafts a new UDP message that is forwarded to the (K8s default) load balancer. The first four bytes of this UDP message contains the SCTP socket number in which the message has been received then the SCTP payload. For the downstream packets, the frontend has one downstream thread per eNB/gNB. Once a UDP packet is received from a worker, the SCTP socket is extracted from the first four bytes while the fifth byte is used by the worker to specify the Stream ID of each SCTP message; the payload starts from the sixth byte. By embedding the SCTP socket into the UDP message, the frontend avoids inspecting the S1AP/NGAP packets, thereby reducing the latency.

4.2 Worker

We implemented two versions of the worker, each corresponding to 4G and 5G. The 5G version (4G version) covers the Registration and Deregistration with/without switch-off and NG Setup [4] (Attach, Detach and S1 Setup [2]). Additionally, we implemented the Handover event given its frequency compared with the Attach and Detach events [64] and the fact that it is highly latency-sensitive. Specifically, S1 Handovers were implemented in the 4G version. While the above are only a subset of the NGAP / S1AP events, they are nevertheless sufficient to enable evaluation with greatly varying amounts of control plane traffic, by controlling the number of UEs and the rate at which they register, handover, and deregister from the core. These events make up the most common control plane events handled by the core, as per [64], and each spans multiple message exchanges between UE/gNB/eNB and core.

4.3 Database

The database uses one thread per active worker pod to allow parallel operations. The database is built on top of a hashmap structure that has a cost of $O(1)$ and stores the information related to the UEs and e/gNBs. The IMSI, key and operator key of each UE are provided through an external file, while all the other parameters such as TEID, NAS IDs or S1AP/NGAP IDs are generated by the database using the IMSI of each UE as seed. Given that the data for a particular UE can only be accessed by one worker at a time, the database does not face inconsistencies from concurrent accesses.

In order to streamline the development and facilitate access to the DB, we have packed the protocol used by the worker to access the DB – the Database Protocol (DBP) – in a library named *libck*. This protocol runs on top of TCP, and each worker pod has a persistent DBP connection with the database during its lifetime. DBP allows the worker to retrieve data of any UE using different IDs (IMSI, MSIN, TMSI, or AMF/MME UE S1AP ID). Furthermore, the DBP request messages are assembled in a way that permits pulling and pushing data using just one request, thereby significantly reducing the network usage.

5 EVALUATION

In this section, we use the 4G and 5G CoreKube implementations (described in §4) to experimentally evaluate its standard compliance, efficiency, autoscalability and resilience. We compare CoreKube performance with multiple

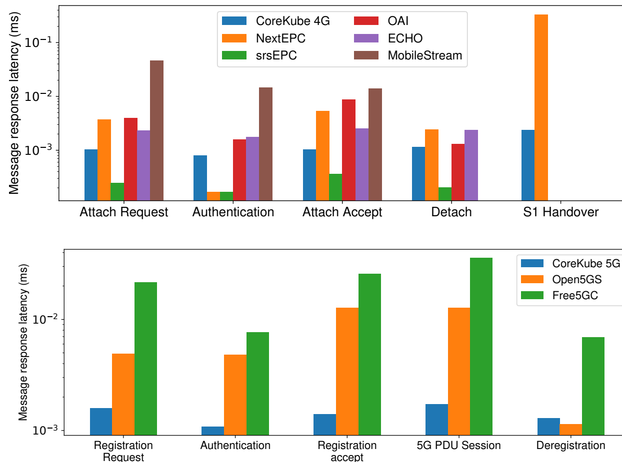

Figure 6: a) 4G message response latency comparison. b) 5G message response latency comparison.

existing open source standard-compliant 4G/5G core network implementations (including Open5GS [45] that is being used in commercial networks) as well as two representative state-of-the-art mobile core designs – ECHO [53] and MobileStream [20].

All our evaluations are conducted on the Powder platform [18, 55]. CoreKube was deployed with K8s, while each alternative core instance (Open5GS, srsEPC, etc.) was deployed on bare metal but on an identical physical machine as the one used for CoreKube. For the alternative core implementations included in comparative evaluation, we used the versions provided on Powder [43]. Control plane traffic workload generation used the Nervion RAN emulator [44], which supports the generation of control and data plane workloads for 4G and 5G networks by specifying a RAN scenario and UE behaviors.

5.1 Standard Compliance

Standard compliance can be demonstrated through our use of Nervion, which by default emulates a standard-compliant RAN and has already been validated with commodity UEs as well as against standard-compliant RAN implementations [44]. To further demonstrate the standard compliance nature of CoreKube, we have successfully connected different types of COTS UEs to CoreKube but omit the details due to lack of space.

5.2 Efficiency

Here we evaluate the efficiency of CoreKube in comparison with other core networks. Similarly to other works (e.g., [20, 53]), we use the *message response latency* as the metric, which reflects core network efficiency from a UE perspective. It is defined as the time between sending a control plane message request and receiving its corresponding response.

We compare the latency of CoreKube against seven different core network implementations: five 4G core alternatives – OAI core network [12], srsEPC [66], NextEPC [37], ECHO [53], and MobileStream [20] – and two 5G core network implementations – Free5GC [29] and Open5GS [45]. For fair comparison, we use the same hardware deployment – a d430 Powder node (Intel Xeon E5-2630 v3 at 2.40GHz with 64GB RAM) – for all the core network alternatives including CoreKube.

We consider message latencies for UE Attach (4G and 5G), Detach (4G and 5G), and S1 Handover (4G only) events. The workload for the Attach/Detach events is generated from an emulated UE continuously performing Attach-Detach cycles with no time gap between events – a worst-case scenario to stress the core and assess its efficiency. The S1 Handover has been generated using srsRAN [66] – three USRP B210s (one UE and two eNBs) and an RF attenuation matrix on the Powder platform [55].

Message latency comparison with other cores. Results for the 4G case are shown in Fig. 6a. Note that, among the alternative 4G core implementations evaluated, only NextEPC supports the S1 Handover event. Also note that the MobileStream implementation (made available by the authors on Powder) does *not* support the Detach event.

From Fig. 6a, it is clear that CoreKube is generally more efficient than the other alternatives, with the exception of srsEPC. The lower message latencies with srsEPC are, however, expected given that it merges all core components (MME, HSS, and SPGW) into a single entity, thereby avoiding the latency from network usage to communicate between different core components. On the flip side, the monolithic srsEPC approach scales poorly with the increasing number of UEs [44]. The superior message latency performance with CoreKube (note the log-scale of the y-axis) is due to its design with truly stateless workers that limits inspection of any message to just once, within the worker handling it. The impact of this design is apparent when comparing CoreKube with ECHO and MobileStream. In the case of ECHO, each message needs to be inspected multiple times at the agent (within eNB) and the worker; moreover, it also requires a custom header to be added at the agent. MobileStream, on the other hand, relies on function-specific workers (SCTP, S1AP, NAS, Authentication, and Security), which introduces additional penalty to direct a message to the right worker.

We conducted a similar experiment using 5G core network alternatives with the results shown in Fig. 6b. Here we see the 5G counterparts of the messages from Fig. 6a plus the 5G PDU Session, which is the time elapsed between the PDU session establishment request and the PDU session establishment accept message. As in the case of 4G, compared to the Open5GS and Free5GC standard-compliant 5G core implementations, CoreKube 5G yields significantly superior efficiency overall and for most of the messages.

Latency contribution of different CoreKube components. We now zoom in to understand the extent to which different parts of CoreKube (Frontend, Worker, and DB)

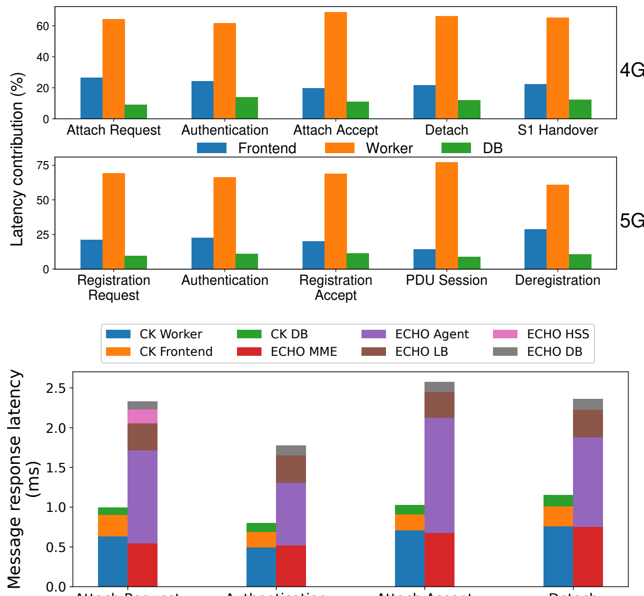

Figure 7: (a) Latency contribution of CoreKube components; (b) Latency contribution of components in 4G CoreKube and ECHO.

contribute to its total response latencies for different messages. Fig. 7a shows the percentage contribution of Frontend, Worker and DB to the overall latency with CoreKube for different control plane events in 4G and 5G. As expected, the component that contributes the most to the latency is the worker. Next is the Frontend whose latency contribution is twice that of the DB. This is because the Frontend encapsulates SCTP messages into UDP messages in the upstream (and decapsulates them in the downstream), whereas the DB just needs to access a hashmap to retrieve the necessary data requested by a worker.

We conduct an additional experiment to get insight into the impact of truly stateless workers in CoreKube that requires only one message inspection within a worker. For this, we consider ECHO [53] as a representative of alternative designs that require multiple message inspections [10, 14, 15, 20, 48, 49, 51, 53, 54, 60, 61, 64]. Results shown in Fig. 7b clearly suggest that dominant contributors to message response latency in ECHO – agent and MME worker – are linked to message inspections. This makes ECHO's overall latency twice as high as that with CoreKube. Note that the S1 Handover event is not shown because it is not implemented in ECHO.

5.3 Autoscaling

Here we evaluate the autoscaling ability of CoreKube. We do this by examining scalability and resource efficiency – the two key aspects for achieving autoscaling.

5.3.1 *Scalability.*

We first validate the importance of the two challenges addressed in CoreKube, namely to decouple the core control plane processing from its state, and to decouple that processing from the RAN interface. For this, we created three variants of CoreKube: (1) 'State Decoupled' in which the RAN interface and control plane processing are coupled by having the frontend and the worker within a single entity; (2) 'Connection Decoupled' with the state and processing closely coupled so that each worker was stateful rather than stateless; and (3) 'Fully Coupled Architecture', where the state, processing and the RAN interface are all closely coupled. We then study how the scalability of CoreKube compares to that of these three variants with increasing the amount of control plane traffic (by increasing the number of UEs). The results, shown in Fig. 8a, demonstrate that the scalability of CoreKube is due to decoupling on both dimensions. The request-response ratio metric used represents the percentage of requests that are handled correctly by the core (a request is handled correctly if its response is generated with no errors). This metric is used to show how the core behaves under increasing incoming signaling load. The State Decoupled variant fails to scale because the close coupling between the worker and the frontend prevents any traffic from being sent to any newly-scaled instances (additional workers). As such, with increasing load, the only worker used saturates and the system does not scale.

The second variant (Connection Decoupled) fails to scale in a different way. Although newly-scaled instances of the worker receive traffic, the stateful nature of these workers prevents the majority of UE control plane events from completing successfully. UE events are made up of a number of request-response messages between the UE and the core, where the state must be maintained throughout. A given event can succeed only if all of the messages making up the event are processed by the same stateful worker instance. If a message is routed to another worker, then it will not have the state context from the previous messages, and the event will fail. At higher amounts of load, the core scales up more instances, which increases the probability that events will fail (the messages getting routed to more stateful workers that do not contain the required state context). The Fully Coupled Architecture fails for the same reasons as the first variant (State Decoupled): all the traffic is routed to the worker instance that was coupled with the frontend at the start, and therefore that worker saturates at higher control traffic loads.

We now study the scalability of CoreKube with increasing number of UEs using the Nervion RAN emulator [44]. For comparison, we consider NextEPC (4G) [37] and Open5GS (5G) [45] because they are, respectively, the most complete open source 4G and 5G standard-compliant core implementations. We did not include MobileStream [20] as it does not support detach events. ECHO requires the ECHO agent to be integrated within the eNB, making it difficult to evaluate it with Nervion, which expects a standard-compliant RAN-Core interface. The control plane workload to assess the scalability of all the core network alternatives compared is generated by increasing the number of UEs, each performing continuous Attach-Detach event cycles with no gap

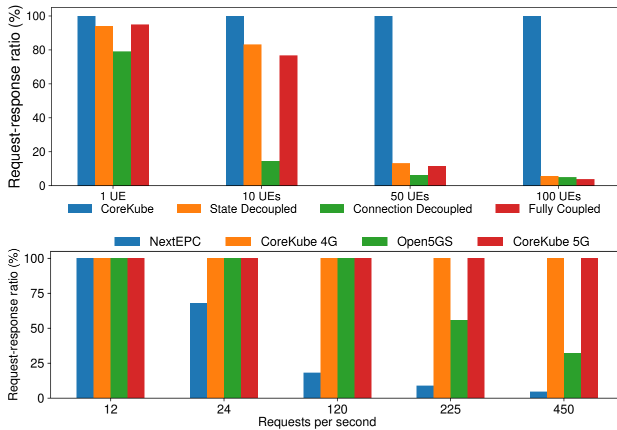

Figure 8: (a) Scalability experiment to validate the design choices underlying CoreKube; (b) CoreKube scalability with increasing control plane workload, relative to NextEPC and Open5GS.

in between. To assess the scalability, we again use request-response ratio metric as above.

Fig. 8b shows the comparison of message success rate with increasing control plane workload (in terms of requests per second). Similarly to Fig. 8a, the metric used represents the number of requests per second handled correctly. Thanks to the dynamically scalable design of CoreKube, when the incoming load is increased, CoreKube scales up the worker instances to be able to handle all the requests at the rate they arrive. In contrast, NextEPC can only successfully cope with load generated by just 5 UEs (12 requests/s). Beyond this, it becomes saturated and fails to keep up with higher number of incoming requests. Open5GS shows a similar behavior - after a certain control plane load (120 requests/s), the core becomes saturated, and so unable to generate responses at the request rate.

5.3.2 *Resource efficiency.*

In Fig. 9a (bar chart), we show how CoreKube scales the number of workers based on the upstream signalling load (shown as the number of control plane request messages per second). The load is generated as above by increasing the number of UEs, each performing continuous Attach-Detach cycles with no time gap. The same figure also shows the impact of incoming load on the message response latency (red bars). We observe that the number of workers grows when the load increases. This is expected given the fact that Kubernetes can detect when the worker pods are getting overloaded and scales them in order to meet the resource usage target, which in our case is set to 80% CPU usage per pod. Due to scaling up the number of worker pods with increased signalling load, CoreKube is able to maintain the average response latency constant. This is a desirable outcome from the user perspective as the perceived latency remains minimal regardless of the number of users and the control traffic handled by the core.

Optimal Kubernetes configuration for the worker. The inset figure in Fig. 9a also shows the result of an experiment (blue line) conducted to determine the optimal worker pod size (in terms of CPU allocation). For that experiment, we fixed the requested amount of CPU millicores[1] of each worker to different values, and we measured the average latency for the base case of 1 UE generating maximum signalling load. As expected, with *lean-workers*, the latency is significantly high given the imposed CPU limitation. On the other hand, *fat-workers* yield better response latencies since the amount of resources is sufficient to process the load for this base case. Based on this experiment, the optimal value is 50 millicores given that larger CPU allocations do not translate to significant latency improvements.

Efficient handling of bursty control plane traffic. A mobile core network should be able to efficiently handle unexpected bursts of control plane traffic. Such traffic can arise from peak periods during the day (for example, during commuting hours), or due to unexpected events (for example, a power outage that results in a large number of attaches to the network once the power is restored). CoreKube has been designed to be dynamically scalable; however, the effectiveness of this depends on its ability to detect bursts of traffic and respond quickly by adapting resource usage. To evaluate this requirement, we conduct an experiment where a large and unexpected burst of control-plane traffic at 560 messages per second from 250 UEs is generated against the core. A plot with the average message response latency and the average worker pod CPU utilization is shown in Fig. 9b. Also shown on the plot is the number of worker pods running in the cluster, showing the relationship between the message response latency, the CPU usage and the number of worker pods.

The behavior shown in this figure demonstrates CoreKube's ability to efficiently handle bursts of control plane traffic. Although the first requests in the burst are responded to with a higher-than-average latency due to the limited resources of the two worker pods at the onset of burst, once the HPA detects the high CPU utilization, the number of worker pods is rapidly scaled to accommodate the volume of requests. Kubernetes autoscaling feature not only helps CoreKube handle unexpected bursts of load but also plays a major role in minimizing the resource usage, something that is desired in cloud environments. When resources used exceed what is necessary, Kubernetes downscales resource consumption by reducing the number of pods. This can be clearly seen near the right end of Fig. 9b when the number of pods drastically gets reduced because the incoming load, and therefore the CPU usage, decreases.

5.4 Resilience

We now examine the resilience aspect of CoreKube, rooted in Kubernetes' self-healing ability. We present results from

[1] In Kubernetes terminology, 1000 millicores correspond to 1 physical CPU or hyper-threaded CPU if hyperthreading is enabled.

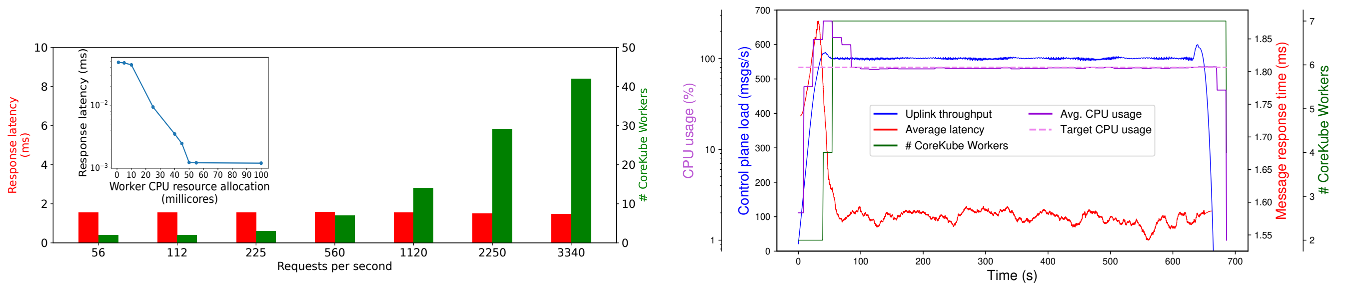

Figure 9: a) CoreKube message response latency and number of workers with increasing signalling traffic load. Inset figure shows CoreKube response latency with varying CPU resource allocation for a worker pod. b) CoreKube message response latency, average Pod CPU utilization and worker pod scaling with control traffic burst.

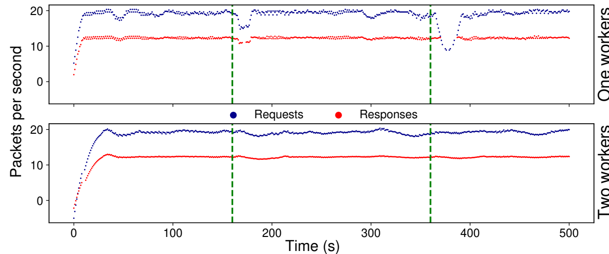

Figure 10: CoreKube resilience in presence of fatal errors for the case of one worker (top plot) and two workers (bottom plot).

two experiments, both having the same control plane workload generated from 10 UEs, each continuously sending control plane messages through repeating Attach-Detach sequence with no time gap. Both these experiments are aimed at assessing how quickly CoreKube reacts to a critical error. In [53], critical error is characterized as an error causing an outage for more than 1 second. They also report a measurement study that detected 2400 critical errors over a 3 month period (equivalent to an error every 54 minutes). For our experiments, we assume critical errors to occur every 200 seconds, a value nearly twenty times higher than that observed in [53]. To create critical errors in our experiments, we introduced a bug that is triggered when the worker receives a specific message, producing a fatal error. The crafted message is sent to the load balancer by a script every 200 seconds.

In Fig. 10, requests (upstream control plane messages) and the corresponding downstream responses are shown as two time series. Critical error events are indicated with vertical lines. The top plot shows the case where CoreKube is forced to use just one worker pod. Here the gaps in the time series after each vertical line are the effects of the fatal error: responses from the core go to 0, and requests consequently also get reduced due to UEs waiting for responses for prior requests. When a fatal error occurs, Kubernetes classifies the pod as CrashLoopBackOff and restarts it immediately, which results in core network operation returning to normal as seen from the time series for requests and responses.

The bottom plot shows the experiment where CoreKube now uses two worker pods, everything else remaining same as before. Note that the presence of an additional worker pod avoids any disruption. When a worker pod experiences a fatal error, the control plane traffic is directed by the load balancer automatically to the other available pod. As mentioned above, the nominal CPU allocation for each CoreKube worker is empirically set to 50 millicores. Additionally, we have experimentally set the bar for the maximum CPU usage of each pod to 200 millicores. These two parameters together with the target average CPU usage setting to 80% and the multi-threaded implementation of the worker increases the CoreKube resilience to unexpected loads such as those arising from worker pod failures.

In practice, a failure in a particular worker instance will only disrupt the specific message that the worker was handling at the time of failure. In such unlikely case, we rely on the UE to resend the message after a defined timeout according to the 3GPP specification [6, 7].

6 RELATED WORK

Mobile core systems design and implementation has been an active area of work in recent years with majority of research focused on control plane like ours. Existing mobile core system designs in the literature can be divided into three classes based on the extent to which they address the two key challenges identified at the outset: (1) implicitly stateful solutions decouple state from processing (Challenge 1) but fail to fully decouple processing from the RAN interface (Challenge 2); (2) partially stateful solutions that decouple only some of the state from the processing; and (3) fully stateful solutions. As such, we discuss prior works in each of these classes below.

Implicitly Stateful Approaches. ECHO [53] design consists of number of stateless worker instances to process control plane messages. While this may suggest ECHO is stateless, it is actually not due to coupling between the UEs in the RAN and ECHO's workers. ECHO places an agent within each eNB, which inspects control plane messages to the core and adds a custom header indicating the UE that the messages originated from. The load balancer uses this header to route messages from each UE to a 'particular' stateless worker instance. In this sense, ECHO is implicitly stateful. Requiring each UE to be served by a particular worker presents performance and resource utilization concerns as individual

instances may become saturated while others may sit idle, depending upon UE assignments. Such routing also has implications for processing latency, since each message must be inspected at both the agent and again in the worker instance. There exist other core design proposals (e.g., CNS-MME [14], MME-FaaS [39], MMLite [51], ML-SLD [23]) that are also implicitly stateful like ECHO and therefore share the above outlined limitations but differ in their details.

Partially Stateful Approaches. This class of mobile core system designs, as the name suggests, allow workers to carry some state. Examples include MobileStream [20], FatProxy [61] and the recent work by Kulkarni et al. [42], whose designs couple some state with processing to reduce the number of requests to an external data store but this prevents easy dynamic scaling. SCALE [15] design has the same issue as it includes a pool of stateful load balancers to direct messages to processing workers. Another example that requires complex stateful load balancing is rVNF [40], a framework for constructing VNFs that considers mobile core as one application.

Stateful Approaches. Existing open source standard-compliant implementations (e.g., [12, 29, 37, 45, 66]) all fall under this class of stateful approaches due to the way they are implemented. As we demonstrated up front, simply containerizing core network functions from the 3GPP standard core and orchestrating them using Kubernetes fails to achieve scalability due to stateful nature of standard core functions, regardless of the specific implementation used [31, 56, 68]. Designs such as [46, 62] that augment the standard core architecture also suffer from the same issue.

Several core designs from the literature feature stateful workers [10, 11, 17, 32, 38, 48, 60]. Some of these designs (e.g., PEPC [60], CleanG [48], Neutrino [10], B5G [17]) allow for dynamic scaling but require complex state migration across workers and also suffer from increased latency penalties due to the need to route requests to the correct stateful worker. On the other hand, designs that avoid such penalties through shared memory and other mechanisms (e.g., L25GC [38], Goshi et al. [32], CellClone [11]) are not dynamically scalable.

In contrast, SoftBox [49] aim at a separate core network (4G EPC) instance per UE, regardless of its traffic, but this can cause inefficient and uneven resource utilization. Other designs move the entire EPC to the edge to suit the requirements of UAV-based LTE networks [50] or rural community LTE networks [63] while others such as Klein [59] target geographic replication of entire EPC for macro-level scaling.

An alternative approach is taken by TuboEPC [64], which proposes offloading data-plane orientated control plane traffic to the data plane, to improve performance. But this approach requires distributing UE state across data plane switches, and therefore fails to decouple the RAN-core interface from processing in the core, because incoming messages must be carefully inspected to decide whether to offload and if so which switch to forward to.

Standard Compliance. A common theme among existing designs is incompatibility with existing mobile core networks. While some designs introduce this incompatibility deliberately to free themselves from the legacy network architecture and protocols (e.g., [48]), other designs are incompatible due to the need to modify the RAN (e.g., [49, 53, 61, 64]) or the RAN-core interface (e.g., [10, 39]). Such incompatibility poses challenges for the real-world deployment of these designs, as a gradual roll out is not possible: an operator must update their entire network to use the new core.

Summary. In contrast to these aforementioned works, CoreKube offers for the first time a mobile core system design with truly stateless workers for control plane processing. By decoupling the processing from both the state and the RAN-core interface, we achieve simpler scaling and efficient message handling as any worker can handle any message from any UE while retaining a standard-compliant interface to the RAN. Furthermore, CoreKube design ideally maps to the Kubernetes framework, inheriting the latter's autoscaling and self-healing capabilities.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we have investigated the design of mobile core system from a control plane perspective that meets the goals of efficient message handling, dynamic scalability and resilience while being standard compliant with respect to the RAN. To this end, we have proposed CoreKube, a novel message focused and cloud-native mobile core design that enables truly stateless and efficient processing through a generic worker handling any control plane message. Containerized form of CoreKube maps to ideal orchestration with Kubernetes, thereby inheriting the latter's autoscaling and self-healing properties. We have also demonstrated that message focused design of CoreKube eases development by developing standard-compliant 4G and 5G CoreKube implementations. Evaluation against existing standard and state-of-the-art core systems has shown significant improvements in efficient processing of control plane messages, dynamic scaling while using minimal compute resources, and automatic recovery from failures.

While our focus in this paper has been on the mobile core control plane, we have performed a preliminary study considering the data plane and validated that CoreKube interworks with the 3GPP standard core data plane (SPGW/UPF). However, it is far from trivial to design the core data plane to be cloud native the same way CoreKube is for the control plane. It gets even more challenging when a performance target (e.g., low latency) is set. Additional complication arises when the core needs to cover large geographical areas. To achieve such capabilities, we need to address several open challenges, including how to flexibly distribute the data plane functionality, how to enable seamless data plane communication in the face of mobility, and the full geo-distribution of the core network spanning both the control and data planes. We plan to investigate these issues as part of our future work.

REFERENCES

[1] 3rd Generation Partnership Project (3GPP). 3GPP Release 8. Technical report, 3GPP, 2008.

[2] 3rd Generation Partnership Project (3GPP). Evolved universal terrestrial radio access network (E-UTRAN); S1 application protocol (S1AP) (3GPP TS 36.413 version 12.03.0 Release 12). Technical report, 3GPP, 2014.

[3] 3rd Generation Partnership Project (3GPP). 3GPP Release 15. Technical report, 3GPP, 2018.

[4] 3rd Generation Partnership Project (3GPP). NG application protocol (NGAP) (3GPP TS 38.413 version 15.0.0 Release 15). Technical report, 3GPP, 2018.

[5] 3rd Generation Partnership Project (3GPP). NG signalling transport (3GPP TS 38.412 version 15.3.0 Release 15). Technical report, 3GPP, 2019.

[6] 3rd Generation Partnership Project (3GPP). Non-access-stratum (NAS) protocol for 5G system (5GS); stage 3 (3GPP TS 24.501 version 15.3.0 Release 15). Technical report, 3GPP, 2019.

[7] 3rd Generation Partnership Project (3GPP). Non-access-stratum (NAS) protocol for Evolved Packet System (EPS); stage 3 (3GPP TS 24.301 version 15.8.0 Release 15). Technical report, 3GPP, 2020.

[8] 3rd Generation Partnership Project (3GPP). System architecture for the 5G System (5GS) (3GPP TS 23.501 version 16.6.0 Release 16). Technical report, 3GPP, 2020.

[9] 3rd Generation Partnership Project (3GPP). Unstructured data storage services (3GPP TS 29.598 version 17.5.0 Release 17). Technical report, 3GPP, 2022.

[10] M. Ahmad, S. U. Jafri, A. Ikram, W. N. A. Qasmi, M. A. Nawazish, Z. A. Uzmi, and Z. A. Qazi. A Low Latency and Consistent Cellular Control Plane. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 648–661, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Mukhtiar Ahmad, Muhammad Ali Nawazish, Muhammad Taimoor Tariq, Muhammad Basit Iqbal Awan, Muhammad Taqi Raza, and Zafar Ayyub Qazi. Enabling emerging edge applications through a 5G control plane intervention. In *Proceedings of the 18th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '22, page 386–400, New York, NY, USA, 2022. Association for Computing Machinery.

[12] OpenAirInterface Software Alliance. OpenAirInterface. https://www.openairinterface.org/.

[13] Amazon Web Services Inc. Kubernetes on AWS. https://aws.amazon.com/kubernetes/.

[14] P. C. Amogh, G. Veeramachaneni, A. K. Rangisetti, B. R. Tamma, and A. A. Franklin. A cloud native solution for dynamic auto scaling of MME in LTE. In *2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, pages 1–7, 2017.

[15] Arijit Banerjee, Rajesh Mahindra, Karthik Sundaresan, Sneha Kasera, Kobus Van der Merwe, and Sampath Rangarajan. Scaling the LTE control-plane for future mobile access. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, New York, NY, USA, 2015. Association for Computing Machinery.

[16] J. Barr. New - per-second billing for EC2 instances and EBS volumes. https://aws.amazon.com/blogs/aws/new-per-second-billing-for-ec2-instances-and-ebs-volumes/, 2017.

[17] Yahuza Bello, Alaa Awad Abdellatif, Mhd Saria Allahham, Ahmed Refaey Hussein, Aiman Erbad, Amr Mohamed, and Mohsen Guizani. B5G: Predictive container auto-scaling for cellular evolved packet core. *IEEE Access*, 9:158204–158214, 2021.

[18] J. Breen et al. Powder: Platform for open wireless data-driven experimental research. *Computer Networks*, 197:108281, 2021.

[19] M. Chae, H. Lee, and K Lee. A performance comparison of Linux containers and virtual machines using Docker and KVM. *Cluster Computing*, 22(1):1765–1775, Jan 2019.

[20] J. Cho, R. Stutsman, and J. Van der Merwe. MobileStream: A Scalable, Programmable and Evolvable Mobile Core Control Plane Platform. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, page 293–306, New York, NY, USA, 2018. Association for Computing Machinery.

[21] Datadog. 11 facts about real-world container use. https://www.datadoghq.com/container-report/, November 2020.

[22] S. Dredge. A hyperscale public cloud provider's role in operator networks and private 5G. https://www.metaswitch.com/blog/a-hyperscale-public-cloud-providers-role-in-operator-networks-and-private-5g, February 2021.

[23] Keliang Du, Luhan Wang, Xiangming Wen, Yu Liu, Haiwen Niu, and Shaoxin Huang. ML-SLD: A message-level stateless design for cloud-native 5G core network. *Digital Communications and Networks*, 2022.

[24] A. Elmokashfi, D. Zhou, and D. Baltrünas. Adding the next nine: An investigation of mobile broadband networks availability. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, MobiCom '17, page 88–100, New York, NY, USA, 2017. Association for Computing Machinery.

[25] Telefonaktiebolaget LM Ericsson. Building a new world: A guide to evolving from EPC to 5G Core. https://www.ericsson.com/assets/local/digital-services/offerings/core-network/5g-core-guide-building-a-new-world.pdf, June 2021.

[26] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and Linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172, March 2015.

[27] Cloud Native Computing Foundation. Kubernetes. https://kubernetes.io/.

[28] Open Networking Foundation. Open Mobile Evolved Core. https://www.opennetworking.org/omec/.

[29] free5GC.org. Free5GC. https://www.free5gc.org/.

[30] Google. Google Kubernetes Engine. https://cloud.google.com/kubernetes-engine/.

[31] Endri Goshi, Michael Jarschel, Rastin Pries, Mu He, and Wolfgang Kellerer. Investigating inter-NF dependencies in cloud-native 5G core networks. In *2021 17th International Conference on Network and Service Management (CNSM)*, pages 370–374, 2021.

[32] Endri Goshi, Raffael Stahl, Mu He, Rastin Pries, and Wolfgang Kellerer. Procedure-based functional decomposition for 5G core network functions. http://dx.doi.org/10.15496/publikation-67452, 2022.

[33] Docker Inc. Docker. https://www.docker.com/.

[34] Docker Inc. Docker Compose. https://docs.docker.com/compose/.

[35] Docker Inc. Docker Swarm. https://docs.docker.com/engine/swarm/.

[36] Facebook Inc. Magma. https://www.facebook.com/connectivity/solutions/magma.

[37] NextEPC Inc. NextEPC. https://nextepc.org/.

[38] Vivek Jain, Hao-Tse Chu, Shixiong Qi, Chia-An Lee, Hung-Cheng Chang, Cheng-Ying Hsieh, K. K. Ramakrishnan, and Jyh-Cheng Chen. L25GC: A low latency 5G core network based on high-performance NFV platforms. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 143–157, New York, NY, USA, 2022. Association for Computing Machinery.

[39] Sonika Jindal and Robert Ricci. MME-FaaS cloud-native control for mobile networks. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 152–157, New York, NY, USA, 2019. Association for Computing Machinery.

[40] Antonios Katsarakis, Zhaowei Tan, Matthew Balkwill, Bozidar Radunovic, Andrew Bainbridge, Aleksandar Dragojevic, Boris Grot, and Yongguang Zhang. rVNF: Reliable, scalable and performant cellular VNFs in the cloud. Technical Report MSR-TR-2021-7, Microsoft, April 2021.

[41] Kubernetes. Intro to Windows support in Kubernetes. https://kubernetes.io/docs/setup/production-environment/windows/intro-windows-in-kubernetes/.

[42] Umakant Kulkarni, Amit Sheoran, and Sonia Fahmy. The cost of stateless network functions in 5G. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, ANCS '21, page 73–79, New York, NY, USA, 2021. Association for Computing Machinery.

[43] J. Larrea. Nervion Powder Profile. https://bit.ly/3Azu6Ty.

[44] Jon Larrea, Mahesh K. Marina, and Jacobus Van der Merwe. Nervion: A Cloud Native RAN Emulator for Scalable and Flexible Mobile Core Evaluation. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, MobiCom '21, page 736–748, New York, NY, USA, 2021. Association for Computing Machinery.

[45] Sukchan Lee. Open5GS. https://open5gs.org/.

[46] Yuanjie Li, Hewu Li, Wei Liu, Lixin Liu, Yimei Chen, Jianping Wu, Qian Wu, Jun Liu, and Zeqi Lai. A case for stateless mobile core network functions in space. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 298–313, New York, NY, USA, 2022. Association for Computing Machinery.

[47] Microsoft. Azure Kubernetes Service (AKS). https://azure.microsoft.com/en-us/services/kubernetes-service/.

[48] A. Mohammadkhan, K. K. Ramakrishnan, and V. A. Jain. CleanG – Improving the Architecture and Protocols for Future Cellular Networks With NFV. *IEEE/ACM Transactions on Networking*, pages 1–14, 2020.

[49] M. Moradi, Y. Lin, Z. M. Mao, S. Sen, and O. Spatscheck. SoftBox: A Customizable, Low-Latency, and Scalable 5G Core Network Architecture. *IEEE Journal on Selected Areas in Communications*, 36(3):438–456, 2018.

[50] M. Moradi, K. Sundaresan, E. Chai, S. Rangarajan, and Z. M. Mao. SkyCore: Moving core to the edge for untethered and reliable UAV-based LTE networks. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, MobiCom '18, pages 35–49, New York, NY, USA, 2018. Association for Computing Machinery.

[51] V. Nagendra, A. Bhattacharya, A. Gandhi, and S. R. Das. MMLite: A Scalable and Resource Efficient Control Plane for Next Generation Cellular Packet Core. In *Proceedings of the 2019 ACM Symposium on SDN Research*, SOSR '19, page 69–83, New York, NY, USA, 2019. Association for Computing Machinery.

[52] P. Nash. Extending per second billing in Google Cloud. https://cloud.google.com/blog/products/gcp/extending-per-second-billing-in-google, 2017.

[53] B. Nguyen, T. Zhang, B. Radunovic, R. Stutsman, T. Karagiannis, J. Kocur, and J. Van der Merwe. ECHO: A reliable distributed cellular core network for hyper-scale public clouds. In *Mobicom 2018*. ACM, October 2018.

[54] Van-Giang Nguyen, Karl-Johan Grinnemo, Javid Taheri, and Anna Brunstrom. Adaptive and latency-aware load balancing for control plane traffic in the 4G/5G core. In *2021 Joint European Conference on Networks and Communications 6G Summit (EuCNC/6G Summit)*, pages 365–370, 2021.

[55] The University of Utah. Powder. https://powderwireless.net/.

[56] OpenAirInterface Software Alliance. openair-k8s. https://github.com/OPENAIRINTERFACE/openair-k8s.

[57] Google Cloud Platform. Compatible operating systems and Kubernetes versions. https://cloud.google.com/migrate/anthos/docs/compatible-os-versions.

[58] D. Polencic. Load balancing and scaling long-lived connections in Kubernetes. https://learnk8s.io/kubernetes-long-lived-connections, 2020.

[59] Z. Qazi, P. Penumarthi, V. Sekar, V. Gopalakrishnan, K. Joshi, and S. Das. KLEIN: A Minimally Disruptive Design for an Elastic Cellular Core. In *Proceedings of the ACM Symposium on SDN Research (SOSR)*, 2016.

[60] Z. A. Qazi, M. Walls, A. Panda, V. Sekar, S. Ratnasamy, and S. Shenker. A High Performance Packet Core for Next Generation Cellular Networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 348–361, New York, NY, USA, 2017. Association for Computing Machinery.

[61] M. T. Raza, D. Kim, K. Kim, S. Lu, and M. Gerla. Rethinking LTE network functions virtualization. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, pages 1–10, 2017.

[62] Muhammad Taqi Raza, Fatima Muhammad Anwar, Dongho Kim, and Kyu-Han Kim. Highly available service access through proactive events execution in LTE NFV. *IEEE Transactions on Network and Service Management*, 18(3):2531–2544, 2021.

[63] S. Sevilla, M. Johnson, P. Kosakanchit, J. Liang, and K. Heimerl. Experiences: Design, Implementation, and Deployment of CoLTE, a Community LTE Solution. In *The 25th Annual International Conference on Mobile Computing and Networking*, MobiCom '19, New York, NY, USA, 2019. Association for Computing Machinery.

[64] R. Shah, V. Kumar, M. Vutukuru, and P. Kulkarni. TurboEPC: Leveraging Dataplane Programmability to Accelerate the Mobile Packet Core. In *Proceedings of the Symposium on SDN Research*, SOSR '20, page 83–95, New York, NY, USA, 2020. Association for Computing Machinery.

[65] S. Shirinbab, L. Lundberg, and E. Casalicchio. Performance evaluation of containers and virtual machines when running Cassandra workload concurrently. *Concurrency and Computation: Practice and Experience*, 32(17):e5693, 2020.

[66] Software Radio Systems. srsRAN. https://www.srsran.com/.

[67] Techplayon. 5G NR SA registration/attach call flow. https://www.techplayon.com/5g-nr-sa-registration-attach-call-flow/, 2020.

[68] GitHub user 'ohyoungili'. open5gs_ueransim_k8s. https://github.com/ohyoungili.

[69] A. Weissberger. Google Cloud and Nokia partner to build cloud-native 5G Core and Edge Networking. https://techblog.comsoc.org/2021/01/14/google-cloud-and-nokia-partner-to-build-cloud-native-5g-core-and-edge-networking/, January 2021.