A Scalable and Fault-Tolerant 5G Core on Kubernetes

Sourav Paul, Tanvi Keluskar, Mythili Vutukuru
Department of Computer Science and Engineering, Indian Institute of Technology, Bombay
Email: {souravpaul, tanvikeluskar, mythili}@cse.iitb.ac.in

***Abstract*—The 5G core is an important part of a 5G network, and with the exponential growth of connected devices and dynamic traffic conditions, it is essential for the 5G core to be scalable and fault-tolerant. This requirement is even more important with the softwarization of the packet core and cloud-based deployments. This paper presents a comparison of design choices when building a scalable and fault-tolerant 5G core on a Kubernetes-orchestrated cloud platform. Prior work in this direction does not fully explore all design choices, or is not compliant with 3GPP standards. In contrast, we start with a 3GPP-compliant production-grade 5G core, and build multiple variants of cloud-native 5G core components. We leverage Kubernetes' automatic failover, and auto-scaling capabilities in different ways to make the 5G core scalable and fault-tolerant. We conduct extensive experiments to evaluate the scalability and resilience of various design choices, and to quantify the overheads of cloud deployments. Our work exposes tradeoffs between performance and fault tolerance, and provides insights on how best to design a scalable and fault-tolerant 5G core on Kubernetes.**

***Index Terms*—5G core, Cloud-Native, Docker, Fault Tolerance, Kubernetes, Scalability**

I. Introduction

The introduction of 5G is a remarkable milestone in mobile communication, with the entire network architecture being redesigned to support improved broadband capability, low latency, and machine-to-machine communication. The most important component of the 5G infrastructure is the 5G core, which connects the wireless radio access network to other networks, and manages connectivity and data flow of user devices. The 5G core consists of multiple network functions (NFs) that handle registration, authentication, session management, and mobility, among other things. In 5G, these components are expected to be built as modular software, running on virtual machines or containers on a cloud, in accordance with the principles of Network Function Virtualization (NFV).

To handle the enormous growth in the number of connected devices and the highly variable nature of traffic patterns, the 5G core should be able to scale its performance dynamically in response to input load. The high availability expected of telecommunication networks requires the 5G core to be resilient to failures. In the traditional stateful architecture of the mobile packet core in 4G and earlier, the state and the processing were tightly coupled within a component, which makes achieving scalability and fault tolerance difficult. For example, the failure of a component can lead to loss of critical state and service disruption, and horizontally scaling a stateful component requires synchronization of state across multiple replicas. The 5G architecture makes two provisions for addressing these challenges. First, the standards prescribe a separate data store component to be used with the various NFs, allowing the NFs to be built as stateless entities that can scale and failover easily. Second, the standards recommend a cloud-native deployment, to leverage the auto-scaling and auto-healing capabilities of cloud orchestration systems.

To this end, there have been several proposals on how to design and deploy the 5G core on cloud platforms like Kubernetes. Some work in this direction [1], [2] proposes refactoring the 5G core, e.g., to consolidate multiple network functions into one stateless worker. However, such designs limit the capability of the network functions to scale independently, and deviate from the 3GPP specifications, making them impractical to deploy in real-world scenarios. The open-source SD-Core implementation in the Aether project [3] adopts a stateless design that is aligned with the 3GPP standards, but the state checkpointing is done infrequently (once per control plane procedure), limiting the resilience of the system. But, to the best of our knowledge, there is no work that comprehensively explores the various design decisions when building a scalable and fault-tolerant 5G core, a gap that this paper seeks to bridge.

This paper designs and implements a scalable and fault-tolerant 5G core on the Kubernetes cloud platform. We begin with a stateful implementation of a production-grade 5G core [4], and add support for checkpointing state to an external data store. This decoupling of state from the NFs allows the stateless NFs to scale and failover easily, using the auto-scaling and auto-healing features of the Kubernetes orchestration framework. We experiment with different granularities of checkpointing state, from the infrequent (once per procedure) granularity used in Aether's SD-Core to the fine-grained granularity of checkpointing state for every message and evaluate the performance and resilience tradeoffs. We find that more frequent state checkpointing leads to up to 75% lower throughput, but provides perfect fault tolerance in the face of failures. We evaluate several Kubernetes configurations for 5G core deployment, and find that using Kubernetes for orchestration of the 5G core imposes a minimal overhead of 1.5% in terms of throughput and latency. Our work also identifies and addresses several questions that arise when building a practical, standards-compliant 5G core on Kubernetes, e.g., how to assign identifiers to mobile subscribers in a scaled-out 5G core design, and what type of load balancer to use.

Our work makes the following contributions: (a) design, implementation, and evaluation of a scalable and fault-tolerant 5G core on Kubernetes (b) comparison of various design choices when developing and deploying 5G core on Kubernetes, leading to useful insights for designers and developers of future cloud-native mobile packet core systems.

II. Background

A. 5G core

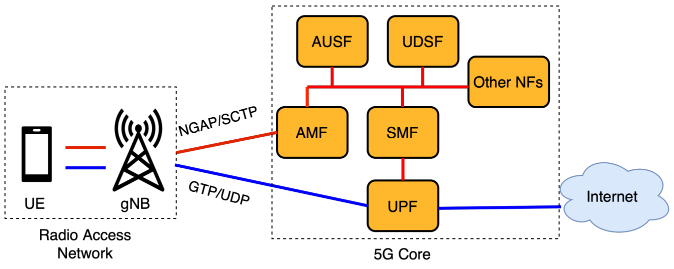

Fig. 1: 5G Network

The 5G core has been standardized, starting with 3GPP Release 15 [5], to serve the requirements of increased throughput, low latency, and reliability of modern 5G networks. As shown in Figure 1, a 5G network consists of the Radio Access Network (RAN), comprising the base station (gNB) and the User Equipment (UE), and the packet core. The 5G core consists of multiple Network Functions (NFs) that help connect the RAN to external networks. Some important NFs in the control plane of the 5G core are the Access and Mobility Management Function (AMF) which manages registration and mobility of UEs, the Authentication Server Function (AUSF) which handles authentication, and the Session Management Function (SMF) which handles the setup and teardown of data sessions through the core. All NFs can optionally use the Unstructured Data Storage Function (UDSF) [6] to store and manage application state generated during the various signaling procedures. The data plane of the 5G core consists of the User Plane Function (UPF) that forwards user traffic to the internet in accordance with the forwarding rules configured by the SMF. The AMF interacts with the RAN using 5G protocol messages exchanged over a persistent SCTP connection, whereas the control plane components communicate over REST-based HTTP messages exchanged over TCP.

The 5G core implements multiple signalling *procedures* on behalf of mobile users for registration, session management, and mobility management [7]. Each procedure consists of a series of messages exchanged in a *call flow* across multiple NFs in the RAN and the packet core. When the 5G core connects with the RAN, the gNB and the AMF execute the NG Setup [8] procedure over NGAP (NG Application Protocol) [9] to establish and configure the communication channel between them. Subsequently, all control plane requests of users connected to a base station are communicated to the core through this channel. Figure 2 presents a simplified call flow of the registration procedure. In this process, a new UE initiates an initial registration request via NAS (Non-Access Stratum) [10] signaling through the base station to the AMF.

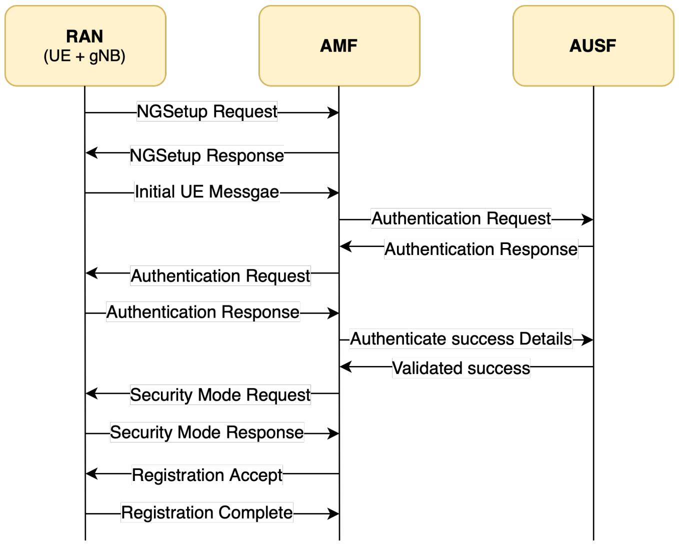

Fig. 2: UE Registration Call Flow

The AMF then interacts with several control plane NFs to complete the registration. For instance, the AMF obtains an authentication challenge from the AUSF, forwards it to the UE, and subsequently sends the UE's response back to the AUSF for verification. By the end of this procedure, the UE and the network mutually authenticate each other, and setup security parameters for further communication.

All control plane NFs, and the AMF in particular, maintain *context* for every UE connected to the packet core. This context is created during the registration, updated in subsequent procedures, and held as long as the UE is active. The UE context consists of several details about the UE, such as the UE's identity, security credentials, and session information, which are required to maintain service continuity throughout the lifecycle of the UE. Every UE has several identifiers, e.g., AMF UE NGAP ID assigned to each active UE by the AMF, which can be used as unique keys to retrieve the UE context across different signaling procedures. Another important point to note is that the 5G core tracks the stage of the UE within its lifecycle (e.g., whether registered or not, whether actively connected or idle) via a finite state machine (FSM), and the FSM state of the UE stored in its context helps the core to enforce only valid transitions for the UE (e.g., the UE can setup sessions only after it has registered).

B. Docker and Kubernetes

Containers have redefined the way modern cloud-native applications are deployed. They provide lightweight, portable environments for software to run consistently irrespective of the underlying architecture. They combine an application and its dependencies into a single unit. Docker [11] is the most popular containerization platform. It provides a simple way to create images and run containers. A "dockerfile" provides blueprint for a container, and contains definitions of all the dependencies and configurations required to run the container. Once dockerfiles are built, we can easily share and deploy the image on any machine as a container by using a "docker run" command with the image as the argument.

Kubernetes [12] is an open-source container orchestration platform that automates containerized workload deployment and scaling. It groups containers into smallest deployable units called "pods" and runs them on "nodes" (physical or virtual machines that run the containers). Usually, a pod consists of one or more containers that share resources, such as storage and networking. Kubernetes allows developers to easily expose applications running inside pods, and enable communication between pods and external systems, by abstracting the underlying dynamic IP addresses of pods with stable virtual IPs. A YAML manifest file is used to specify the Kubernetes objects used in a deployment, including the number of replicas (copies of a pod), configuration settings for networking, exposed ports, and so on. Kubernetes also supports load balancers to distribute incoming traffic evenly across multiple replicas of a service, ensuring availability and redundancy. Additionally, Kubernetes provides self-healing capabilities by automatically replacing failed pods. Using all these components, Kubernetes automates scaling, load balancing, and self-healing for containerized applications, streamlining application deployment and management at scale.

III. Related Work

A. Redesigning the 5G core

Prior work such as [13], [14], [15], [16], [17], [18], [19], and [20] proposed frameworks that leverage microservices, virtualization, and parallel processing to improve performance, scalability and fault tolerance of the Evolved Packet Core. While some of these ideas have been incorporated into the 5G architecture, researchers continue to work on improving the 5G architecture for specific use cases. SkyCore [21] proposes relocating the RAN and EPC to edge locations, and adapting them for UAV hardware, to enhance network accessibility in remote areas. SpaceCore [22] focuses on providing connectivity in locations where establishing static base stations is impractical. Neutrino [23] aims to optimize data serialization and deserialization with flat-buffers and manage handovers via geo-replication, although state replication remains complex and challenging. On the other hand, this paper limits itself to the standards-compliant 5G core architecture, and addresses the issues to make it more scalable and fault-tolerant.

Some recent proposals refactor the 5G core, towards improving its performance and resilience. L25GC [24] leverages shared memory to minimize message serialization and processing overheads, achieving resiliency through replica synchronization, along with packet buffering. However, its scalability is limited by the complexity and overhead of maintaining consistency as the number of replicas increases. Our approach, utilizing a data store (UDSF) for state storage, simplifies the system by eliminating the need for individual NF synchronization, thereby ensuring both reliability and scalability. PP5GS [2] adopts a procedure-based functional decomposition of the 5G core, which reduces procedure completion time but sacrifices reusability and modularity. The tight integration within a single NF creates a scalability bottleneck and a single point of failure under high network load. In contrast, our standards-compliant modular approach distributes tasks across multiple NFs, enabling parallel execution and independent scaling, thus enhancing both reliability and scalability.

CoreKube [1] consolidates core network functions (AMF, AUSF, SMF) into a single stateless worker, with a frontend load balancer managing message distribution, and deploys them on Kubernetes. While this design is fault-tolerant and stateless, it does not allow scaling individual NFs independently. Furthermore, it is non-compliant with the 3GPP standards, as it uses UDP communication between the frontend and the AMF. Similarly, ML-SLD [25] implements a non-3GPP defined NF that acts as a middleware between RAN and AMF, which accepts requests from RAN over SCTP and communicates with AMF over HTTP. In contrast, our implementations adhere to 3GPP specifications, and supports independent scaling of NFs.

FlexCore [26] utilizes an XDP-SCTP load balancer to route packets to designated AMF instances. While it supports both stateful and stateless architectures, it necessitates modifications in the RAN, and does not address how the load balancer updates its endpoint configuration when new AMF instances are dynamically added. We address these issues by enabling dynamic scaling of AMF instances using a Kubernetes load balancer. Other works, such as [27], [28], [29], and [30], explore containerization and orchestration with Kubernetes, but they often overlook key aspects of statelessness, scalability, and fault tolerance.

B. Open-source implementations

Most open-source 5G core implementations, such as Free5GC [31], Open5GS [32], and OpenOAI [33], do not support a stateless NF implementation using UDSF, as of September 2024. The closest to our own work is SD-Core [34], a cloud-native mobile core with stateless AMF and SMF components, which is used in the open-source 5G platform Aether [3]. Unlike our implementation, SD-Core does not include a UDSF, and instead utilizes MongoDB [35] directly within the AMF and SMF to manage transient unstructured data. Additionally, SD-Core incorporates an L7 SCTP load balancer that directs SCTP connections from the gNB to available AMFs. Most importantly, as described in [36], SD-Core adheres to procedural statelessness, storing state information only upon completion of the entire registration procedure. This design makes SD-Core less resilient to network function failures, as it risks losing data for users that are mid-way through procedures. In our work, we also explore the design of fine-grained state checkpointing at the message level. In the event of a procedure interruption, this allows network functions to continue precisely where they left off, resulting in greater resilience, albeit at the cost of lower performance.

Unlike aforementioned works, this paper comprehensively examines various stateless and fault-tolerant 5G core designs, presenting extensive experimental analysis across multiple metrics to compare these architectural choices.

IV. System Design and Implementation

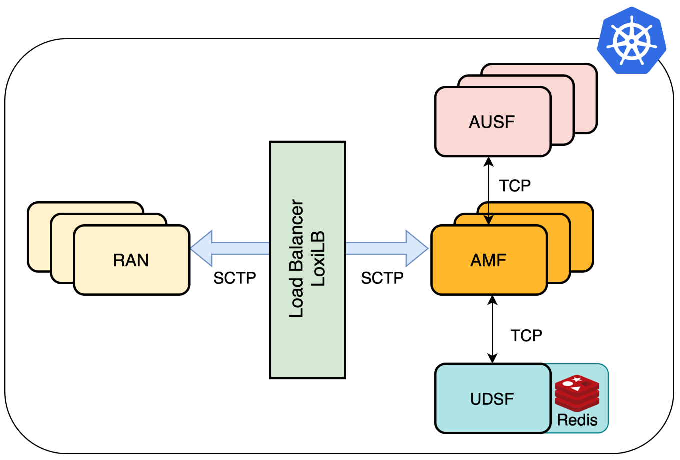

Fig. 3: Our system architecture

A. Architecture of a 5G core NF

A typical 5G core network function is a multi-threaded application that exchanges messages with other NFs using sockets or other such communication mechanisms. Every NF maintains application state in the form of UE context, and every step of a procedure's call flow updates the context to reflect the current status of the ongoing procedures. For this paper, we will mainly focus on the AMF and the registration procedure, but the same principles apply to the other NFs and procedures in the 5G core. There are two designs possible with respect to how the applications maintain context. In a *stateful* design, context is maintained locally in the NF. However, stateful designs face challenges such as complexity in replication and fault tolerance. In contrast, a *stateless* NF does not store any state locally, but in an external data store like the UDSF. This allows the NF to scale horizontally, where every incoming request can be served by a different replica of the NF. Each replica retrieves context from the UDSF before processing the request, and saves it back into the UDSF after processing the request. While stateless architectures are more fault-tolerant and easily scalable, they suffer from performance penalty due to the overhead of storing and retrieving context.

Consider a stateless AMF that checkpoints user context during the registration procedure. There are two options for when to checkpoint state: either after completing the entire registration procedure (procedure granularity) or after each individual message exchanged during the call flow of the procedure (message granularity). With *procedure granularity*, the AMF stores the UE context in the UDSF after the entire procedure is successfully completed, allowing it to be utilized for future procedures of the UE, like session establishment. This design imposes lesser overhead because there are fewer state updates to the remote data store. However, when the AMF crashes, all ongoing UE registrations fail, placing undue load on the newly spawned AMF since all UEs need to retry the registration procedure again. An alternate design choice would be for the AMF to checkpoint UE context at the *message granularity*. In this approach, the AMF stores the UE context in the UDSF after processing, and before sending a response to each request message in the call flow of a procedure. AMF sends a response to the RAN only after receiving a response from UDSF that the context has been successfully stored. This design enhances resilience by allowing the system to recover from the last checkpoint instead of restarting the whole procedure. But due to the increased amount of state updates, the network overhead is higher, resulting in a higher procedure completion time (PCT). This approach also requires a more complex implementation because of checkpointing the state of the UE at every step.

We modified a proprietary production-grade 5G core implementation [4], which used a stateful design, to build out these variants of the stateless 5G core. The original implementation used a service-based, multi-threaded architecture in C++ over an event-driven Linux epoll framework. Figure 3 shows our system architecture with stateless NFs deployed on a Kubernetes-based environment. We have so far ported the AMF and AUSF to a scalable, fault-tolerant design, while porting other NFs is ongoing work. Having outlined the general approaches to statelessness, we now describe the specific mechanisms needed for scalability and fault tolerance.

B. Mechanisms for Scalability and Fault Tolerance

We now describe some specific challenges that arise when we wish to use multiple replicas of a stateless NF (we use AMF as an example in the discussion below) to enable horizontal scalability and fault tolerance in the 5G core, and the mechanisms we propose to address the same.

Retry mechanism. Any fault-tolerant server implementation requires a robust retry mechanism on the client side. When an AMF fails during a registration procedure, we require the UE to retry the procedure once again, so that it can succeed at the new AMF replica. To this end, we implement two types of retry mechanisms in the 5G RAN emulator that came along with our 5G core implementation. When any step of the registration procedure fails, the UE retries that particular step again, in the hope that it may succeed. After a certain number of retries is exceeded, the UE assumes the registration has failed, and tries to register again. A similar retry mechanism is implemented at the gNB as well. When an AMF fails, the connected gNBs detect the failure through timeout of the SCTP connection and resend NG Setup Requests over a new SCTP connection, in the hope that they will find new AMF replicas to associate with. Once the NG Setup is complete with the new AMF, timed-out UE requests resume from the exact step of registration that failed (when checkpointing at message granularity), or from the beginning of a procedure (when checkpointing at procedure granularity), ensuring the registration process completes seamlessly.

Saving AMF response. The registration procedure in the AMF involves the UE's finite state machine moving through various stages with each step of the call flow. When the AMF fails after saving the UE context in the UDSF but before sending the response to the RAN, or if the response it sent to the RAN has gotten lost, then we can have a

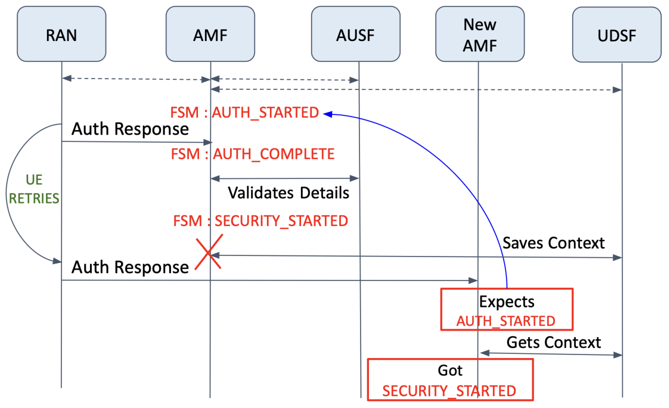

Fig. 4: AMF failure leading to inconsistent state

situation where the UE and the core have an inconsistent view of the FSM state of the UE, as shown in Figure 4. In this example, when the UE retries the failed request, the new AMF instance fetches the UE context from the UDSF. However, the retrieved FSM state does not match the expected FSM state based on the current request from the RAN. This means that the new AMF is unable to correctly process the UE's request, resulting in a registration failure. To solve this issue, our implementation saves both the UE context and the response that the AMF will send to the UE atomically in the UDSF. This ensures that if the AMF fails before responding to the RAN, the new AMF can fetch both the context and the last processed response if required. The new AMF can detect inconsistencies by checking the FSM state and current request from the RAN; if a mismatch is detected, it retrieves the last response from the UDSF and sends it to the UE. This approach maintains consistency in the UE state throughout the registration procedure, preventing failures due to mismatched information between UE and AMF. Moreover, it is easier to implement and results in lower network overhead compared to two-phase commit and versioning mechanisms.

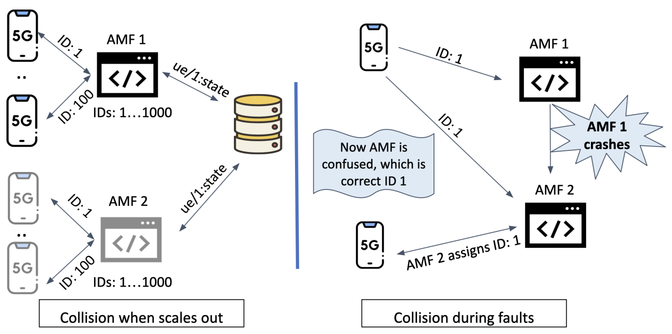

Fig. 5: AMF UE NGAP ID collision problem

Partitioning NGAP ID space. When an initial UE message is received at the AMF, each UE is assigned an AMF UE NGAP ID by the AMF, which is expected to be unique within the UEs connected to a single AMF, but not across AMFs. The AMF UE NGAP ID is used as a key to store the UE context and response in the UDSF by each AMF. Now, when we have multiple stateless AMF replicas, and each uses its own numberspace for the AMF UE NGAP ID, then we can have collisions when UEs having the same identifier get associated to the same AMF due to failure of some replicas or due to scaling out the AMF replicas. For example, in Fig. 5, we see that two UEs with the same identifier get associated with the same AMF due to a replica failure. This results in overwriting and incorrect update of the UE state for both UEs, potentially resulting in registration failures for both. To address this issue, we partition the AMF UE NGAP ID space across the multiple AMF replicas. When a new AMF instance is initialized, the UDSF assigns it a unique range of AMF UE NGAP IDs from a list it maintains. Once a range is allocated, the UDSF marks it as used. This eliminates the risk of ID collision, ensuring that each AMF operates within its own distinct range.

C. Unstructured Data Storage Function (UDSF)

The UDSF provides a flexible API that allows other network functions to save and retrieve unstructured data. We implemented a subset of the UDSF APIs mentioned in the 3GPP specifications, which we invoke from our modified stateless implementations of AMF and AUSF, which are: (i) record creation API, which provides an endpoint to NFs to store data like UE context in our case, (ii) record retrieval API, for NFs to retrieve the stored data. We use Redis as the backing data store for the UDSF, since it offers low latency which is essential for systems like 5G core. Redis also offers a straightforward mechanism for namespacing using key patterns to store and fetch data efficiently. To support the mechanisms mentioned in the previous section, we use three distinct Redis key patterns: $\{$*ueContext/id*$\}$ to store context, $\{$*ranReply/id*$\}$ to store AMF's response to RAN and $\{$*AmfUeNgapIdRange/id*$\}$ to store the AMF UE NGAP ID Range information. Creating these namespaces helps in optimal key management and allows scalable data access. We built this UDSF from scratch in C++ using a multi-threaded, event-driven architecture with Linux epoll. Additionally, Redis offers high availability and cluster versions for deployment on Kubernetes, enabling the replication of stored data across the cluster nodes in either sharded or non-sharded form. We currently use in-pod redis-server with UDSF for prototyping, but the above features could be further used to scale UDSF and make it fault-tolerant.

D. Other Network Functions

The mechanisms that have been used to make the AMF stateless, scalable, and fault-tolerant can also be applied to other NFs. We have so far completed our changes in AUSF, and plan to make SMF stateless in the near future. While the basic mechanisms (checkpointing UE context at procedure or message granularity, retries in the downstream, partitioning numberspaces, etc) remain the same across all NFs, we have also found that some internal implementations of the NFs need some change in a few cases. One such example is, when the AMF requests validation of authentication details of a UE from the AUSF, the AUSF validates the details, sends a response, and deletes all associated data for that UE. According to the standards, AUSF does not need to maintain context beyond this step. But, if the AMF fails after receiving this response and before responding to the UE, the UE retries the same request and a new AMF retries the validation, then the AUSF, lacking the necessary data, may respond with a failure. Addressing this issue requires AUSF to delay data deletion for some time period, perhaps by setting UE context to timeout in the Redis data store. The call flows of procedures have several such unintended interactions between NFs, and a complete fault-tolerant 5G core design must address all such cases.

E. 5G core on Kubernetes

Having described our design of stateless NF implementations, we now describe how we deploy the NFs on Kubernetes to bring multiple advantages in terms of scalability, flexibility, and fault tolerance. Migrating baremetal applications to the cloud-native architecture has its own challenges. When deploying stateless NFs on Kubernetes, there are two main design options: placing all NF containers (except the data store UDSF) in a single-pod, or deploying each NF container in separate pods. In the *single-pod* design, multiple stateless NF containers like AMF and AUSF are grouped together in a single-pod. These containers share the same Linux network namespace, allowing for local communication between the control plane NFs without any network overhead. This design has advantages like reduced network latency, and simplified management (since there is only one pod to manage). But this approach has limited scaling capabilities and isn't cost efficient. Even if only one NF experiences higher traffic, the entire pod must scale together. Additionally, all NFs are tightly coupled, and if a pod crashes, all NF containers fail simultaneously. In the *multi-pod* design, each NF runs inside its own individual pod, with Kubernetes managing the communication between them. This design enables independent scaling of the NFs based on the traffic demand, and faults are isolated since failure of one NF does not impact the other NFs. We can also tailor resources to meet the unique needs of each NF, for example, by giving more resources to AMF which handles much more control plane operations, thereby enhancing the system efficiency. But this design has a higher network overhead since communication between NFs is done over the network. Further, deployments can get complex since each pod needs to be managed and orchestrated separately.

We used Docker to build the NF images and orchestrated them using Kubernetes. We started by first writing the dockerfile for each NF. We pushed these images to Docker Hub, accessible to our Kubernetes cluster. With the cluster set up, we configured Flannel [37] as the CNI plugin (Container Network Interface) [38] to enable networking between pods and nodes. YAML files were created to define pod specifications and services with static IPs and port mappings. To manage replicas and ensure consistency with the YAML specifications, we deployed the pods using Kubernetes deployment.

F. Load Balancing

Load balancing is critical in 5G core to efficiently manage traffic across multiple replicas of NFs as they scale. When we deploy multiple replicas of an NF like AMF, we must distribute traffic evenly to prevent overloading a single instance. We can classify the traffic in 5G core into two types: short TCP connections between the control plane's NFs and long-lasting SCTP connections between the AMF and the RAN. Kubernetes services can effectively manage the short TCP connection, but it is not enough to manage the long-lasting SCTP connection. So, for SCTP traffic, we have used LoxiLB [39], which is an open-source cloud-native load balancer and can be deployed for Kubernetes cluster.

A load balancer can redirect traffic either at layer 4 (by rewriting packet headers) or at layer 7 (by terminating transport layer connections, and acting as a proxy). We have an option to deploy LoxiLB in either L4 or L7 mode, and we compare both options. We employ a round-robin strategy in LoxiLB in L4 mode, assigning new RAN connections to the next available AMF replica and redirecting all subsequent requests to it. LoxiLB uses eBPF [40] for efficient packet processing and load balancing in the L4 mode. We also deployed LoxiLB in L7 mode, in which LoxiLB, upon receiving an SCTP connection request from RAN, establishes SCTP connections with all the AMF replicas. It redirects the NG Setup from the RAN to these replicas and then forwards the UE registrations to a specific AMF based on the hash of the RAN UE NGAP ID. This setup was used to assess the performance overhead of LoxiLB in L7 mode. In contrast, the SCTP load balancer component of Aether's SD-Core takes a different approach, by maintaining gRPC [41] connections to a pool of backend AMFs. Upon receiving a request from the RAN, it selects an AMF and forwards the message over gRPC to it. The AMF's response is then relayed back to the RAN through the SCTP connection by the load balancer. In either of our designs, note that, since all requests of a particular UE from a RAN are directed to the same AMF instance in L4 or L7 mode, the UE context is only retrieved from UDSF when the data is not locally available or if the connection has shifted to a new AMF. This approach minimizes latency by avoiding unnecessary data retrievals in our design.

V. Evaluation

We now compare the performance of the various design variants of the scalable, fault-tolerant 5G core we have described so far. Our results provide insights on the best practices of building a cloud-native 5G core on Kubernetes.

Setup. We use two servers, each having 16 Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.60GHz CPUs on board, which can run 32 threads with hyperthreading enabled, on Linux kernel version 5.15. We configured the Kubernetes cluster on these two baremetal servers, one for the control plane and another for the worker node, using Kubeadm v1.29.5. We conducted all the experiments, both on baremetal and on Kubernetes, on these two servers. The worker node runs the RAN emulator and the network functions (AMF, AUSF, and UDSF). We have used simplified stubs for other network functions that are not being evaluated. We generate a closed-loop workload of UE registrations from emulated UEs in the RAN. In all experiments, we ensure that all NFs, except for the one being tested (i.e., AMF), have sufficient CPU and RAM to avoid becoming performance bottlenecks. We then measure the average throughput (number of registration requests processed per second) and procedure completion time (time taken to finish the registration procedure), over 300 seconds of the test.

Prototypes. We compare the performance of the following design variants of the 5G core: the completely *stateful* design that was present in the original, unmodified 5G core code we started with; the *fully stateless* design where each NF checkpoints state at the UDSF after every procedure, and keeps a local copy for use as an optimization; and the *procedurally stateless* design where state is saved in the UDSF after one procedure (say, the registration procedure) is completed. We show only the performance of the initial UE registration procedure at the AMF in our evaluation, and omit results for other NFs due to lack of space.

A. *Stateful vs Stateless designs*

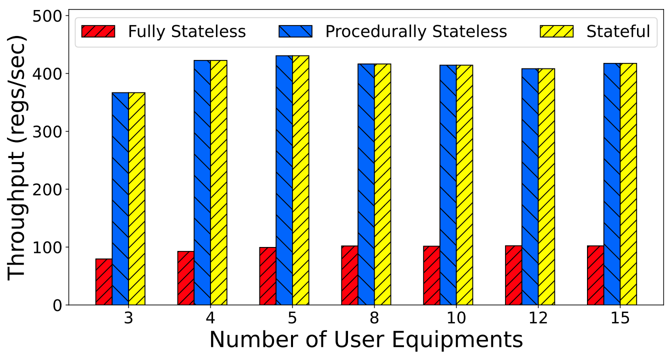

Fig. 6: Avg. throughput of various 5G core designs

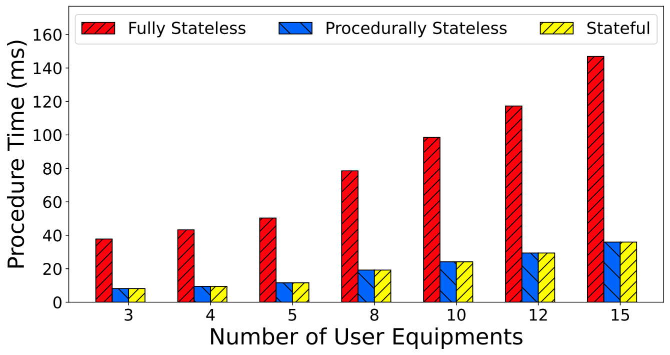

Fig. 7: Avg. PCT of various 5G core design

In this section, we discuss the performance implications of the various designs described in the design section. The Fig. 6 and Fig. 7 shows the average throughput and procedure completion time of the registration procedure as the number of user equipments (UEs) varies. We see that the throughput of the AMF running on a single core saturates with 4–5 UEs running in closed loop mode with zero think time. In the fully stateless variant, throughput drops by 75% compared to the stateful variant due to the overhead of serializing UE context, deserializing responses to and from UDSF into JSON, and saving the data in Redis. The fully stateless variant checkpoints the state three times during the registration procedure, adding network calls from AMF to UDSF, resulting in a threefold increase in procedure completion time at saturation. The metrics for stateful and procedurally stateless variants are similar because the procedurally stateless AMF saves state only after receiving the registration complete message from the RAN, and the RAN doesn't wait for an AMF response after sending it. However, the performance will vary for real scenarios where UEs perform multiple procedures like session establishment. So the results shown in the Fig. 6 and 7 are the best possible scenarios for the procedurally stateless variant.

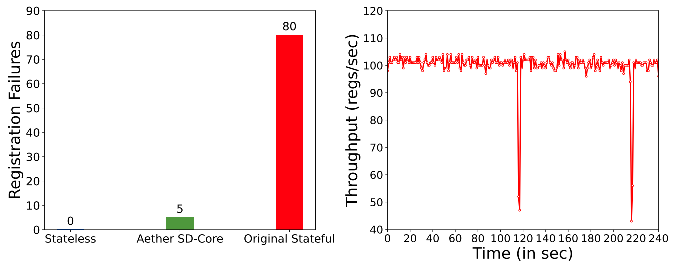

Fig. 8: Fault tolerance of various 5G core designs

Fig. 8 depicts the fault tolerance of three 5G core variants: the original stateful design, Aether's SD-Core, which is procedurally stateless, and our stateless implementation deployed on Kubernetes. In an experiment with 100 sequential UE registration iterations, we simulate a fault at around the 20th iteration. The stateful variant experiences 80% registration failures, while Aether's SD-Core has a 5% failure rate due to the lack of state-saving for ongoing registrations. Aether's SD Core suffers from delays in spawning a new AMF instance and its discovery by the L7 SCTP load balancer. The fully stateless implementation achieves 100% successful registrations. By saving UE state at each step, the state is retrieved from UDSF when a new AMF spawns, allowing seamless recovery and completion of the registration process. As the network scales and AMFs handle millions of UEs with parallel registrations, the failure rates in procedurally stateless and stateful variants would increase significantly, while the fully stateless approach would ensure continuous reliability.

In summary, while the fully stateless 5G core introduces performance overhead, it offers a crucial tradeoff, i.e., superior fault tolerance and 100% successful registrations. This ensures seamless recovery under faults, making it more reliable than stateful and procedurally stateless variants, especially at scale. Fig. 8 also shows the time series throughput graph, which depicts the performance of a fully stateless AMF with faults induced at two points. Despite the interruptions, the system demonstrates rapid recovery, resuming request processing and achieving pre-fault throughput levels within 2 seconds after each failure. This showcases the resilience and fault-tolerant capabilities of our fully stateless design. Such quick recovery ensures minimal impact on overall performance and highlights the system's ability to maintain high availability even under fault conditions. This makes it highly suitable for environments requiring low downtime and consistent throughput.

B. Baremetal vs Single-Pod vs Multi-Pod variants

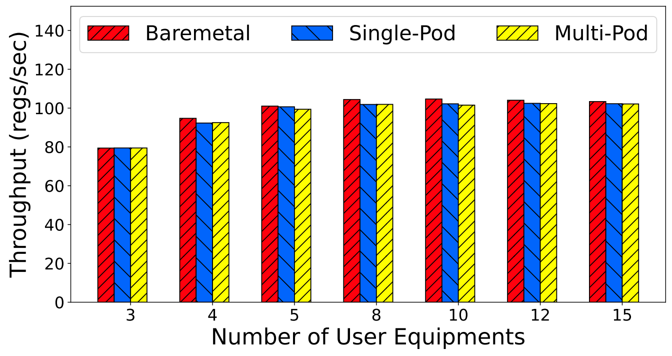

Fig. 9: Avg. throughput across different deployment models

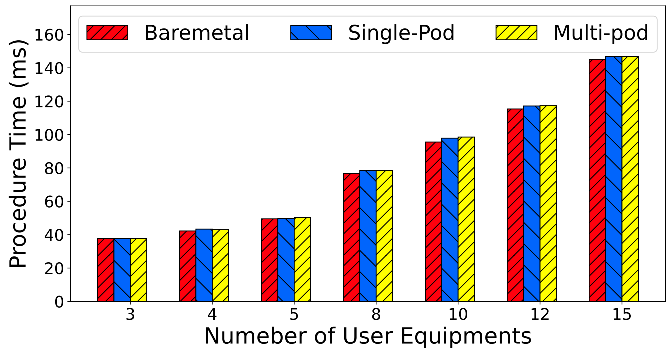

Fig. 10: Avg. PCT across different deployment models

We evaluate the performance overhead introduced by containerization and orchestration when deploying a fully stateless 5G core on Kubernetes compared to a bare-metal environment. In our experiments, we pinned the AMF to a single core and saturated it to assess both configurations. Fig. 9 and Fig. 10 presents the observed overhead when using Kubernetes as the orchestration layer. In the bare-metal setup, all network functions (NFs) communicate directly without any additional layers of abstraction. In contrast, the Kubernetes deployment consists of two configurations: a single-pod setup, where AMF and AUSF are colocated within the same pod, and a multi-pod setup, where each NF, including RAN, is isolated in its own pod. UDSF is in a separate pod in both the cases. Our load testing revealed a 1.5% increase in procedure completion time (PCT) and a 1.5% decrease in throughput when transitioning from the bare-metal deployment to the single-pod Kubernetes setup. We also analyzed the impact of Kubernetes managed networking, which became evident when comparing the single-pod and multi-pod configurations. The shift from single-pod to multi-pod increased PCT by 0.25% and reduced throughput by 0.25%. This minor difference is attributed to the network calls to AUSF in the multi-pod setup. Overall, the overhead introduced by containerization, orchestration, and intra-cluster networking in Kubernetes is minimal, resulting in negligible performance degradation. This demonstrates the viability of Kubernetes for deploying a stateless 5G core with only marginal impact on system throughput and latency.

C. L4 vs L7 Load Balancing

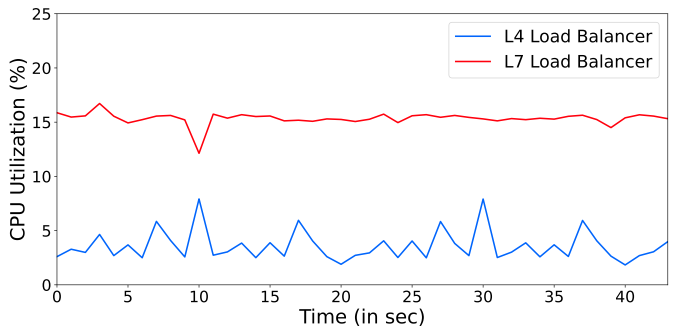

Fig. 11: CPU utilization of L4 vs L7 load balancer

To assess the performance differences between L4 and L7 load balancers, we deployed both variants of the LoxiLB load balancer in Docker containers outside the cluster along with our fully stateless variant of the 5G core in the Kubernetes cluster. These load balancers route requests from a RAN emulator to 2 replicas of our fully stateless AMF over SCTP. Figure 11 illustrates the performance overhead of the L7 load balancer compared to its L4 counterpart, measured in terms of CPU usage at the load balancer. Our evaluation shows that the L4 load balancer consistently outperforms the L7 in CPU usage when both AMFs are saturated with UE registration requests. Specifically, the L7 variant consumes 15% CPU compared to 5% for the L4, indicating a nearly threefold increase in overhead due to the additional processing required for parsing application layer messages and hashing the RAN UE NGAP ID for AMF redirection. However, we point out that LoxiLB's L7 limitation to load balance requests to pods placed across different nodes restricted our ability to test both modes of the load balancer at saturation. In conclusion, the significant CPU overhead associated with the L7 load balancer makes the L4 load balancer, the more efficient option, particularly in scenarios requiring large-scale handling of UEs.

VI. Conclusion

In this paper we consider the problem of designing a scalable and fault-tolerant 5G core over the Kubernetes cloud orchestration platform. We started with a production-grade stateful implementation of the 5G core, and modified the NFs to store application state and UE context in an external Redis-backed UDSF data store, thereby making them stateless. We then added several mechanisms to make these stateless NFs scalable and fault-tolerant, and ported these NFs to the Kubernetes cluster. This paper documents the several design and implementation decisions we had to make in our work, and also compares the performance of the various design choices via experiments on our prototypes. Our results provide valuable insights to future designers of mobile packet core architectures, e.g., to understand the tradeoff between performance and fault tolerance when checkpointing application state in the 5G core NFs, or when deploying on Kubernetes.

References

[1] J. Larrea, A. E. Ferguson, and M. K. Marina, "CoreKube: An Efficient, Autoscaling and Resilient Mobile Core System," in *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, 2023.

[2] E. Goshi, R. Stahl, H. Harkous, M. He, R. Pries, and W. Kellerer, "PP5GS—An Efficient Procedure-Based and Stateless Architecture for Next-Generation Core Networks," *IEEE Trans. on Netw. and Serv. Manag.*, 2022.

[3] Aether, "Aether." https://opennetworking.org/aether/, 2024.

[4] "Proprietary 5G core implementation." Indigenuous 5G testbed project, IIT Bombay, 2021.

[5] 3rd Generation Partnership Project (3GPP), "System architecture for the 5G System (5GS) (3GPP TS 23.501 version 15.2.0 Release 15)," tech. rep., 3GPP, 2018.

[6] 3rd Generation Partnership Project (3GPP), "Unstructured data storage services (3GPP TS 29.598 version 17.5.0 Release 17)," tech. rep., 3GPP, 2022.

[7] 3rd Generation Partnership Project (3GPP), "Procedures for the 5G System (3GPP TS 23.502 version 15.2.0 Release 15)," tech. rep., 3GPP, 2019.

[8] 3rd Generation Partnership Project (3GPP), "NG signalling transport (3GPP TS 38.412 version 15.4.0 Release 15)," tech. rep., 3GPP, 2020.

[9] 3rd Generation Partnership Project (3GPP), "NG application protocol (NGAP) (3GPP TS 38.413 version 15.2.0 Release 15)," tech. rep., 3GPP, 2019.

[10] 3rd Generation Partnership Project (3GPP), "Non-access-stratum (NAS) protocol for 5G system (5GS) (3GPP TS 24.501 version 15.3.0 Release 15)," tech. rep., 3GPP, 2019.

[11] Docker, "Docker." https://www.docker.com, 2024.

[12] Kunbernetes, "Kubernetes." https://kubernetes.io/, 2024.

[13] 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*, 2019.

[14] 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*, 2018.

[15] 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*, 2018.

[16] Y. Li, Z. Yuan, and C. Peng, "A Control-Plane Perspective on Reducing Data Access Latency in LTE Networks," in *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, 2017.

[17] 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*, 2020.

[18] 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 *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, 2018.

[19] 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*, 2017.

[20] A. Banerjee, R. Mahindra, K. Sundaresan, S. Kasera, K. Van der Merwe, and S. Rangarajan, "Scaling the LTE control-plane for future mobile access," in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, 2015.

[21] 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*, 2018.

[22] Y. Li, H. Li, W. Liu, L. Liu, Y. Chen, J. Wu, Q. Wu, J. Liu, and Z. Lai, "A case for stateless mobile core network functions in space," in *Proceedings of the ACM SIGCOMM 2022 Conference*, 2022.

[23] M. Ahmad, S. M. N. Ali, M. T. Tariq, S. U. Jafri, A. Abbas, S. M. A. Zaidi, M. B. I. Awan, Z. A. Uzmi, and Z. A. Qazi, "Neutrino: A Fast and Consistent Edge-Based Cellular Control Plane," *IEEE/ACM Trans. Netw.*, 2022.

[24] V. Jain, H.-T. Chu, S. Qi, C.-A. Lee, H.-C. Chang, C.-Y. Hsieh, K. K. Ramakrishnan, and J.-C. Chen, "L25GC: a low latency 5G core network based on high-performance NFV platforms," in *Proceedings of the ACM SIGCOMM 2022 Conference*, 2022.

[25] K. Du, L. Wang, X. Wen, Y. Liu, H. Niu, and S. Huang, "ML-SLD: A message-level stateless design for cloud-native 5G core network," *Digital Communications and Networks*, 2023.

[26] B. Sharma, S. Vittal, and A. A. Franklin, "FlexCore: Leveraging XDP-SCTP for Scalable and Resilient Network Slice Service in Future 5G Core," in *Proceedings of the 7th Asia-Pacific Workshop on Networking*, 2023.

[27] L.-M. Tufeanu, A. Martian, M.-C. Vochin, C.-L. Paraschiv, and F. Y. Li, "Building an Open Source Containerized 5G SA Network through Docker and Kubernetes," in *2022 25th International Symposium on Wireless Personal Multimedia Communications (WPMC)*, 2022.

[28] R. Botez, A.-G. Pasca, and V. Dobrota, "Kubernetes-Based Network Functions Orchestration for 5G Core Networks with Open Source MANO," in *2022 International Symposium on Electronics and Telecommunications (ISETC)*, 2022.

[29] D. Scotece, A. Noor, L. Foschini, and A. Corradi, "5G-Kube: Complex Telco Core Infrastructure Deployment Made Low-Cost," *IEEE Communications Magazine*, 2023.

[30] A. Khichane, I. Fajjari, N. Aitsaadi, and M. Gueroui, "Cloud Native 5G: an Efficient Orchestration of Cloud Native 5G System," in *IEEE/IFIP Network Operations and Management Symposium*, 2022.

[31] Free5GC, "free5GC." https://free5gc.org/, 2024.

[32] Open5GS, "Open5GS." https://open5gs.org/, 2024.

[33] O. A. Interface, "Open Air Interface." https://openairinterface.org/, 2024.

[34] SD-CORE, "SD-CORE." https://opennetworking.org/sd-core/, 2024.

[35] MongoDB, "MongoDB." https://www.mongodb.com, 2024.

[36] U. Kulkarni, A. Sheoran, and S. Fahmy, "The Cost of Stateless Network Functions in 5G," in *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, 2022.

[37] Flannel, "Flannel." https://github.com/flannel-io/flannel, 2024.

[38] CNCF, "The Container Network Interface." https://www.cni.dev, 2024.

[39] LoxiLB, "LoxiLB." https://www.loxilb.io/, 2024.

[40] eBPF, "eBPF." https://ebpf.io, 2024.

[41] CNCF, "gRPC." https://grpc.io, 2024.