Public Cloud Deployment and Monitoring of Cloud-Native Mobile Core Systems

Yuto Takano

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2024

Abstract

Mobile network operators have not adopted the public cloud as a deployment target despite its ease in manageability and cost efficiency. In this paper, I present a feasibility analysis of deploying a cloud-native mobile core network to AWS, based on a holistic perspective that encompasses qualitative and quantitative data. Through this process, I identify a potential for improving observability, and present Yagra, an observability solution for cloud-native cores. With Yagra, the feasibility requirements for public cloud mobile core network deployments are complete, enabling mobile operators to explore new opportunities as a result.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Yuto Takano*)

Acknowledgements

I would not have been able to complete this project without the kind assistance of my supervisor, Mahesh Marina, and Andrew Ferguson. Thank you very much for your support and guidance through all these months.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Traditionally, mobile network operators have operated their networks using proprietary black-box components purchased from vendors. However, with the introduction of the 5G standard (3rd Generation Partnership Project (3GPP) [2017b]), we are seeing growing support for containerization and virtualization of functions on open commodity hardware, both from commercial operators (AT&T Business [2024]) and from governments (HM Treasury Department of Culture, Media and Sport [2017]). These next-generation cloud-native network architectures have many advantages including being able to scale dynamically based on demand to save resources, and being able to upgrade and automate deployment at a much faster pace. These advantages allow smaller mobile operators to enter the market without the traditionally large financial investment in single-vendor telecommunications hardware. Running such core network in the public cloud (e.g., AWS, Azure, GCP) would be beneficial in saving personnel and maintenance costs thanks to the hardware and parts of the software stack being managed by the cloud provider. However, we have yet to see a study examining such core deployment. This paper aims to test the feasibility of running a 5G core network on public cloud services, and explore the challenges in observability that arise from doing so.

The core of a mobile network handles user and device management, data transfer, and all major signaling operations within the network. By and large, implementations of the core have traditionally been following the recommendation set out by 3GPP in their standards (3rd Generation Partnership Project (3GPP) [2017b]). In this approach, data is passed between multiple virtual modules (Network Functions; NFs) that each perform a specific subset of operations such as policy management, authentication, signaling, etc. This architecture is seen in open-source 5G cores such as Open5GS (The Open5GS Authors [2024]) and Free5GC (The Free5GC Authors [2024]). Unfortunately, this architecture is unable to take advantage of dynamic scaling as demonstrated in the works of Larrea et al. [2023], due to the inherently stateful nature of the NFs. Recent research have instead proposed alternative architectures based on container orchestration and auto-scaling tools like Kubernetes, using stateless workers that can

dynamically replicate on demand (Larrea et al. [2023], Du et al. [2023], Watanabe et al. [2023], Goshi et al. [2023]). These designs are greatly suited for deploying to the cloud, enabling cost-efficient resource provisioning.

The advancements in making a mobile core suitable for cloud deployments is also a financial gain for network operators. Using virtualized hardware from cloud providers cuts down on datacenter estates cost and physical layer network maintenance costs. Public cloud service providers in particular often offer services for managed software infrastructure on top of just virtualized hardware, such as managed Kubernetes clusters. Such service simplifies the burden of deployment for network operators, and enable smaller operators to enter the market with smaller investments in specialized personnel and hardware compared to deploying on premise or on a private cloud. These contexts strongly motivate the need for a feasibility test of mobile core networks when deployed to the public cloud. During this study, we also noticed a lack of observability solutions for cloud-native core network components in general, and this paper has been motivated to provide a solution in that space as well.

This paper shows a systematized analysis of public cloud deployment options based on qualitative and quantitative features of a mobile core; an evaluation of an actual mobile core network based on incurred cloud costs and performance as deployed in various configurations; Yagra, a system for achieving intra-core observability for cloud-native core deployments even in limited environments like the public cloud; and an evaluation of Yagra based on its effectiveness under real-world use cases.

As far as the author is aware, this is the first work to cover the feasibility of mobile core network deployments to the public cloud. There is also no work that covers observability specifically for cloud-native and auto-scaling mobile core components, and this paper is therefore a novel contribution to the field of observable mobile core network deployments.

1.2 Outline

The remainder of the paper is structured as follows.

Chapter 1 introduces the motivation behind the project and its main contributions to the field of mobile network core deployments and their observability.

Chapter 2 provides additional background for the goals of the project, and other relevant research already present in the field.

Chapter 3 presents a method of deployment a mobile core network to Amazon Web Services, and analyzes its feasibility both qualitatively and quantatively.

Chapter 4 presents Yagra, an observability system for achieving real-time monitoring of intra-component KPIs in a cloud-native mobile core network.

Chapter 5 summarizes the work of the project. It assesses the public cloud deployment and the Yagra system, and presents room for future work.

Chapter 2

Background and Related Work

2.1 4G and 5G Cellular Networks

Most modern smartphones, tablets, laptops and IoT devices are capable of connecting to a cellular network to establish an Internet connection. The connection enables devices to perform voice calls, browse the web, or send sensor data. Such networks and the bodies who operate them have nowadays become critical infrastructure for our society, where any unexpected outages can lead to catastrophic consequences.

Cellular networks, hereon referred to as mobile networks, have had multiple generations of standards over the years, standardised by the International Telecommunications Union (ITU) in collaboration with the 3rd Generation Partnership Project (3GPP). Commercial operators currently predominantly run 4th Generation (4G LTE) networks (3rd Generation Partnership Project (3GPP) [2017a]), and are gradually introducing 5th Generation (5G NR) networks (3rd Generation Partnership Project (3GPP) [2017b]) to public use. All cellular networks are structured as having two main parts: a Radio Access Network (RAN) and a Core Network (the core). This is illustrated in Figure 2.1. The RAN is an interlinked network between the end user device (referred to as the User Equipment; UE) and the core. It contains most of the physical-layer equipment for radio access, such as base station and antennas. The core on the other hand is the main back-end of a mobile network, and consists of multiple functionalities, including the control plane section that handles signaling for user and device management, roaming or billing, and the user/data plane section that handles the routing and forwarding of network packets to achieve data transfer for the end-user.

Traditionally, components of the RAN and core have been provided as black-box solutions that operate only on vendor-specific hardware. However, this proves to be costly and limiting for operators as it makes them vulnerable to lock-in pricing effects, as well as supply chain attacks. Multiple initiatives have tackled the problem as a result, such as the 5G standard encouraging function containerization and virtualization on commodity hardware. The O-RAN Alliance also leads the O-RAN specifications initiative (O-RAN ALLIANCE e.V. [2024]), which defines common interfaces between RAN components for interoperability in a multi-vendored RAN deployment. Government initiatives have also been a strong proponent for increasing vendor diversification across the network for

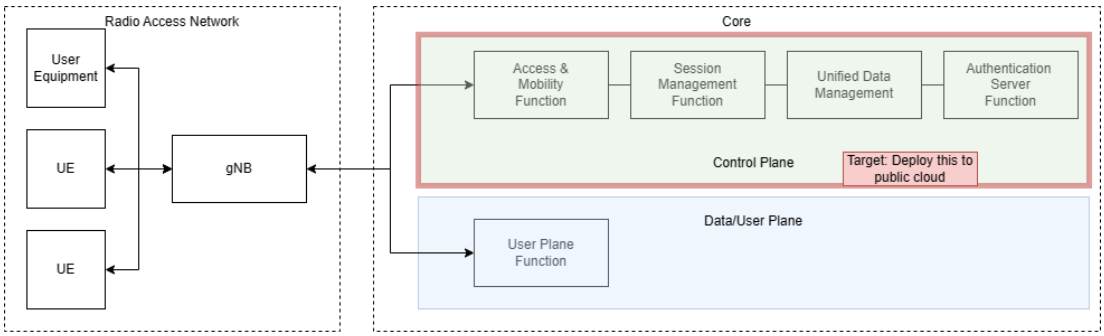


Figure 2.1: Network Diagram Illustrating the 5G Cellular Network

national infrastructure security (UK Department for Science, Innovation & Technology [2022]). These actions have brought an era of software-defined mobile network technologies, where operators have the flexibility and freedom to mix and match software components from a wider accessible range of vendors.

A benefit of encouraging the use of commodity hardware and component virtualization is that the initial investment cost to enter the market as a new network operator becomes lower (Analysys Mason [2022], Naudts et al. [2016], Oughton and Frias [2018]). A report by Affirmed Networks and VMWare showed operators saving an average of 67% of operational expenses through adopting a virtualized 4G packet core (ACG Research, sponsored by Affirmed Networks and VMWare [2015]). These improvements lead to healthier market competition, and align well with the business goals of providers in improving the profit margin in an operational network.

Through all of the aforementioned initiatives for reducing cost and vendor lock-in however, there is an unwavering assumption that the deployment would be made on a private setup of commodity hardware – a setup also referred to as the 'private cloud'. Such deployments are often preferred by operators for their control over the hardware and software stack, and the ability to customize the network to their needs British Telecom [2022]. However, the private cloud setup is not without its own challenges. The setup requires a significant amount of capital investment, and the operational costs can be high due to the need for specialized staff to manage the infrastructure. The setup also requires a significant amount of space, power, and cooling, which can be a limiting factor for operators in urban areas where real estate is expensive.

A solution to the challenges of the private cloud setup is the use of hyperscaler infrastructure, commonly referred to as the 'public cloud'. Specific large service providers ('cloud providers') include Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP), each of which offers a wide range of services on top of the bare hardware to simplify the deployment and save costs. The virtualized and managed technology stacks that each cloud providers offer a lucrative alternative to the private cloud setup, as their creation, modification and destruction cycle are much faster and simpler compared to traditional infrastructure. As a result, operators can take advantage of faster time-to-market and less cost spending on hiring specialized personnel.

Out of the two areas composing a mobile network (RAN and core), it does not make sense for an operator to move any of the RAN components to the public core due to

its inherently high coupling to physical hardware like radio receivers. However, this limitation does not apply to components in the core, which only need to be reachable over IP connections from the UE and RAN. As a result, there is a potential in exploring the feasibility of deploying a mobile core to a public cloud infrastructure.

This area of work inherently relates to previous works in network function virtualization and cloud-native core architectures, as they provide example implementations of core components for us to examine the deployment feasibility for. These include the 3GPP NFV specification (3rd Generation Partnership Project (3GPP) [2014]), Open5GS (The Open5GS Authors [2024]), an open-source 5G core following 3GPP NFV specifications closely; CoreKube (Larrea et al. [2023]), another open-source 5G core with an alternative architecture that integrates with Kubernetes for orchestration; Proteus (Syed and Van der Merwe [2016]), a system for dynamic provisioning of NFVs based on a template system; and ECHO (Nguyen et al. [2018]), a 4G packet core built to run in the Microsoft Azure cloud provider. ECHO is specifically of large interest to us as they have managed to deploy a core to the public cloud as part of their analysis. However, their work focuses on the reliability of a public cloud core deployment when compared to on-premise core deployments, and not on the more holistic examination of operational feasibility, such as performance, cost, or observability as we perform in this paper.

2.2 Appeals of Public Cloud Infrastructure

The public cloud is a service model where cloud providers offer virtualized computing resources accessible over the internet as a service, billed by resource usage. The resources are managed by the cloud provider and hosted in multiple distributed data centers distributed across the globe. Often, the cloud provider abstracts the underlying hardware and software stack and instead exposes configuration options through APIs for customers to interact with in order to provision, manage, and destroy resources.

A feature of every cloud provider service offering is the ability to scale hardware resources on-demand, which is only possible because they are virtualized and managed by the cloud provider. This ability is particularly useful for applications with fluctuating workloads, allowing customers to save operational cost as a result of the quick provisioning and de-provisioning of resources. The cloud providers also offer a wider range of services on top of just virtualized hardware, such as managed high-availability databases, machine learning services, and serverless functions. These resources can save customers time and money from implementing or managing a similar stack privately. In fact, news outlets (Morris [2024]) have reported that major commercial network operators such as Three and Vodafone are actively migrating to the public cloud for many of their non-critical IT workloads. With such companies already having knowledge on setting up public cloud stacks, we cannot understate the appeal of manageability, cost, and integration with existing services for a public cloud core deployment.

2.3 Cloud-Native Software Design

Software that is architectured specifically to make use of the characteristics of a cloud infrastructure are referred to as being 'cloud-native'. This term is frequently associated with using tools maintained by the Cloud-Native Software Foundation group (under the Linux Foundation), such as releasing software artifacts in the form of `containerd`-compatible container images (containerd Authors [2024]), or being capable of orchestration through Kubernetes (The Kubernetes Authors [2024]).

`containerd` is a container runtime for running lightweight virtualized software in isolation, used as the underlying runtime for the popular container management software Docker (Docker Inc [2024]). It is used to package up software into 'images', which can be downloaded and executed on a remote server to run a 'container' instance of it.

Kubernetes is an orchestration software for such Docker containers. It consists of the Kubernetes control plane and machines (called 'nodes') that can run individual collection of containers (called 'pods'). Not only does Kubernetes support the deployment of multiple containers in a deterministic manner using YAML configuration files, it also supports complex configurations such as assigning one IPv4 address to a set of pods and randomly routing messages to one (a 'LoadBalancer'), high-availability deployments that get replicated across all nodes (a 'DaemonSet'), or even auto-scaling based on metrics such as CPU or memory usage per pod ('Horizontal Pod Autoscaler'; HPA).

The Kubernetes HPA feature in particular is essential for most Kubernetes clusters, as it allows the user to save costs by only running the minimum number of pod instances required to handle the current workload. However to get the most out of this feature, the pods must be stateless in nature. This is so that an instance can spin up or down under fluctuating load and handle its share without requiring a handover mechanisms to transfer any persistent data. Stateless pods are also painless to schedule on any Kubernetes node, whereas pods that have any state require the Kubernetes scheduler to be aware of this to prevent data loss when e.g., moving pods between nodes.

Software that is truly cloud-native should take such requirements into account. For example, any cloud-native software that is intended to be scaled dynamically should ideally be stateless, and similarly, any cloud-native software that interacts with other Kubernetes pods should be aware of the inherently dynamic and ephemeral nature of some of these pods.

2.3.1 Cloud-Native Mobile Cores

With regards to mobile core implementations in particular, CoreKube, developed by Larrea et al. [2023] is the only mobile core to claim that it is cloud-native. CoreKube is a 4G/5G core implementation that uses a message-oriented architecture for its components. This does not follow the traditional 3GPP-defined NFV architecture, but is what allows the core to be stateless in nature. Instead of components handling different functionalities of the core, the CoreKube Worker component is capable of handling every type of message from start to end. Additional components include only a lightweight database to store persistent UE data which the workers can look up or write in, and a

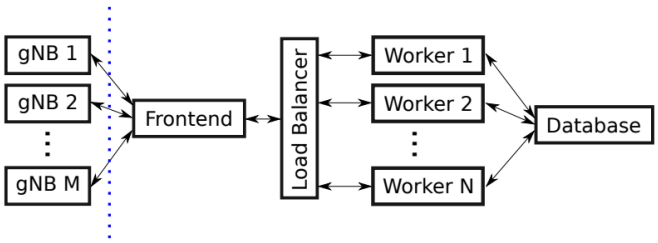


Figure 2.2: The architecture diagram of CoreKube, a cloud-native mobile core that leverages Kubernetes

Front End component which receives the traffic from RANs and UEs before routing in to any available Worker using Kubernetes LoadBalancers. The architecture diagram in Figure 2.2, sourced from Larrea et al. [2023], describes this core in detail.

The CoreKube Front End component has an open SCTP port on port 38412 for Next Generation Application Protocol (NGAP) packets from the UE and RAN (3rd Generation Partnership Project (3GPP) [2018]). These packets are then re-transmitted as UDP to the Workers. UDP is used since the Kubernetes LoadBalancer only supports TCP or UDP traffic, and not the telco-standard SCTP traffic. By making use of the Kubernetes Horizontal Pod Autoscaler, the CoreKube Workers are able to adapt to changes in workload quickly by auto-scaling up as necessary. It is also able to self-heal when there is a fatal error in any of the Worker pods, ensuring that a replacement is quickly spun up to fill the gap. As a result, Larrea et al. [2023] report the latency and performance of CoreKube to be exceptionally good compared to other cores such as Open5GS or MobileStream. These results make CoreKube an ideal mobile core implementation for cloud-native environments, and make it a solid representative example when considering deployments to the public cloud in particular.

2.3.2 Cloud-Native Mobile Core Observability

To evaluate and operationally monitor mobile cores when deployed to any setting, observability is a key requirement. Observability refers to the ability to understand the internal state of a system by looking at its external outputs, such as logs and metrics. In a cloud-native environment, observability is particularly important as the core is expected to be distributed and dynamic in nature thanks to the auto-scaling Kubernetes feature. Yet, for mobile core deployments, we claim that there currently only exist general solutions for observability, and nothing specifically to gather core-specific data. This makes adoption of cloud-native cores, and thus the public cloud, difficult for operators as they cannot gain insightful data.

Previous work has shown (Ferguson et al. [2023]) that it is possible to use a ‘sidecar’ container that collects network traces of incoming and outgoing packets in the core. A sidecar container refers to a container used to monitor and provide metrics for another container within the same pod. Specifically in the works of Ferguson et al. [2023], a network sniffing Linux tool called `tshark` was used to decode control-plane NGAP messages as it entered or exited the CoreKube Front End pod, and provided metrics such as the number of UEs connected at any given moment. The strengths of this sidecar-

based approach to observability is that it does not need any modification to the core, since the observability metrics are gathered externally. However, there are significant drawbacks that make this method infeasible as a general solutoin to cloud-native mobile core observability. First, the memory consumption of packet analysis tools like `tshark` will inherently grow when used with SCTP traffic, since it has to retain the information about a flow in case a response message comes at a later time. Ferguson et al. [2023] solve this by killing the `tshark` process every 30 seconds, but this is not an elegant nor production-ready solution. Second, this approach of performing packet analysis to find useful metrics only worked because CoreKube had not implemented the encryption procedures, resulting in NGAP packets transmitting unencrypted. In a production environment, the core will almost certainly use packet encryption. In conclusion, using a sidecar-container approach to observability is not viable.

Other solutions to observability include generic profilers such as the tracy CPU profiler (Bartosz Taudul [2024]) or Intel VTune (Intel Corporation [2024a]). These solutions can give insight into application-internal performance data and improve the observability of a deployed core. The strength of these lie with their low overhead of approximately 5% (Intel Corporation [2024b]), allowing fine-grained performance metrics to be collected without impact to the services that the mobile core provide. However, this solution is not specific to cores, and cannot collect custom metrics for an operator's needs.

All in all, existing works demonstrate a lack of tools for gathering core-specific data for cloud-native cores. Indeed, in Chapter 4, we outline a system for doing just that, through the introduction of Yagra.

2.4 RAN Emulation

In order to test a core deployment, we need to put it under some traffic load. For this, we reviewed previously works related to UE and RAN traffic generation. The works by Meng et al. [2023] to build a scalable and accurate generator for control-plane traffic for LTE/5G systems fit perfectly for what we require. They have modelled control-plane traffic of UEs across times of the day, and were able to create a statistics-based traffic generator that could emulate realistic traffic for 5G cores. Unfortunately, their source code and model used for building their generator is not publicly available.

Resorting to traditional software-based RAN emulators that require the user to define the workload, we investigated into srsRAN (Gomez-Miguelez et al. [2016]), OpenAir-Interface 5G RAN (OpenAirInterface.org [2024]), and Nervion RAN Emulator (Larrea et al. [2021]). Although each of these support similar features, Larrea et al. [2021] showed that the memory and CPU footprint of srsRAN and OpenAirInteface was larger then that of Nervion. This is due to the former two shipping with a full implementation of the RAN components (such as the gNB, the component responsible for bridging communication between the UE and the core), whereas Nervion only simulates the NGAP packets required for putting a mobile core under load. For the purposes of using RAN emulation in this report, we needed only NGAP-level packet traffic generation and not a full software radio network implementation. As such, when we apply traffic load to public cloud core deployments in Chapter 3, we have used Nervion.

Chapter 3

Deploying to the Public Cloud

3.1 Overview on Methods of Deployment

There are numerous services to run a compute workload in the cloud. The largest ones are Amazon Web Services (AWS), Google Cloud and Microsoft Azure, although there are many smaller companies that offer platform-as-a-service solutions with varying fees and capabilities as well. As the largest in the market, we will focus on AWS as a representative example of a public cloud that a mobile operator may look toward deploying their core network. However, even within one cloud provider, there are still numerous options in which one can deploy the same workload, each with their trade-offs.

In this section, we will list, compare and rank the feasibility of these options in the context of representative workloads. The analysis and ranking is done qualitatively, hypothesising the deployment of a containerised workload with compute requirements akin to a cloud-native mobile core. Specifically, we focus on service offerings that allow deploying workloads to Kubernetes in order to adaptively scale based on fluctuating requirements. We will then compare them against each other in the following section based on estimated cost effectiveness, ease of manageability and potential for observability. This will conclude with a hypothesised ideal method to deploy a mobile core to the public cloud, and lead on to actual experimental performance and cost data in the section following it.

AWS offers largely three distinct methods to deploy workloads on Kubernetes in order of the amount of manual control: Elastic Kubernetes Service (EKS) with the Fargate compute engine, EKS with Elastic Compute Cloud (EC2) engines, and finally, self-managed Kubernetes on EC2 engines. The three are categorised as shown in Figure 3.1.

A managed Kubernetes deployment such as EKS refers to the Kubernetes control plane being automatically provisioned as a service, and the customer being billed for it in exchange for less laborious setup work. This is the most fundamental benefit of public cloud offerings in comparison to running a private cluster within a datacenter, as it reduces the personnel requirements for keeping the Kubernetes control plane up-to-date

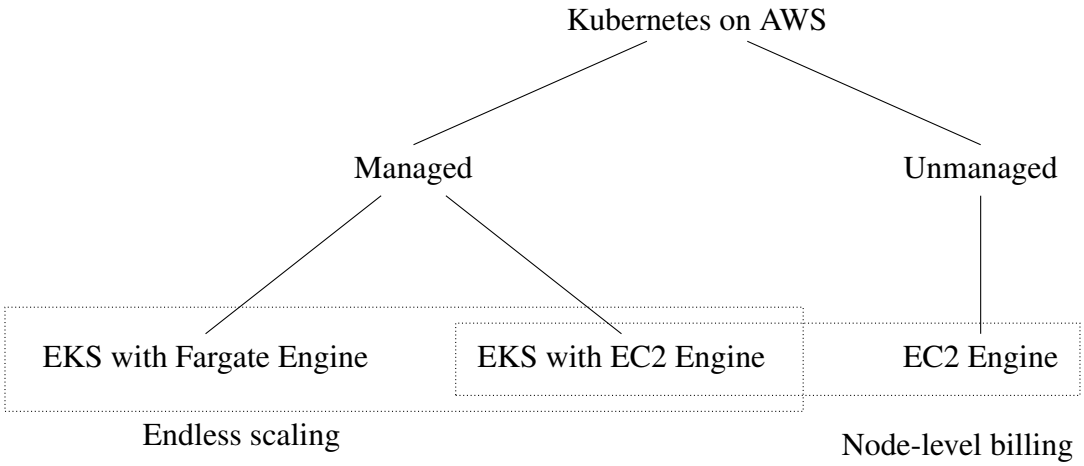


Figure 3.1: Three Kubernetes deployment options on AWS

with security patches, ensuring redundancy and high-availability, and configuring it ready for production. All major cloud platforms provide a service of this nature (see Azure Kubernetes Service, Google Kubernetes Engine, etc). In comparison, running a self-managed Kubernetes control plane on top of the VM offerings of the cloud provider (such as the EC2 engine) is no simpler in complexity to doing so on private VMs. Depending on the cloud provider, it may even be more complex than private clouds due to having to learn the intricacies of the cloud provider's network configuration APIs.

Fargate and EC2 compute engines are AWS-specific variants of a common cloud platform offering: the choice between managing workload at the Kubernetes pod scale, or at the Kubernetes node (underlying VM) scale. Amazon Fargate abstracts away the concept of Kubernetes nodes, and allows customers to focus on launching pods in a serverless manner and getting billed for pod-level resource usage. On the contrary, Amazon EC2 is a VM service which allows customers to choose specific VM vCPU/memory specs while getting billed at the granularity of those VMs regardless of the number of pods running on it. With the same operational effort view that favored the managed Kubernetes deployment, Fargate is a much better choice. However, whether or not Fargate's pricing scheme is reasonable depends on the type of workload to be deployed. This is further explained later. From a financial perspective, EC2 has a beneficial feature over Fargate in that it allows customers to deploy workloads to Spot Instances, a cheaper variant of VMs that offer the same power as normal VMs, but have the restriction that workloads may be evicted from it with short notice.

EC2 (and similar VM services in other cloud providers) additionally has a secondary layer of customization that Fargate does not offer, where the customer chooses the type of the VMs specific to their workload. The exact offerings differ for each cloud provider. For AWS, the VMs are categorized as General Purpose, Compute Optimized, Memory Optimized, Storage Optimized, Accelerated Computing, and High-Performance Computing, where within each category there are types ranging in specifications. An overview of the features that each of these types provide is listed below:

General Purpose VMs These offer access to a burst-able CPU of moderate frequency

(3.1-3.5GHz [cite]), suited for websites, development environments, and for hosting application microservices. A representative AWS General Purpose VM is the T3 series, which uses an Intel Xeon Scalable processor that can burst up to 3.1GHz.

Compute Optimized VMs These come with CPUs with higher performance than general purpose VMs, which makes it suited for high-traffic servers, scientific computing and video encoding. A representative example is the C5 series, most of which use an Intel Xeon Platinum 8000 series processor that can burst up to 3.5GHz. At the same prices as General Purpose VMs, Compute Optimized ones tend to feature more vCPUs and less memory or internet bandwidth.

Memory Optimized VMs At the other are Memory Optimized VMs like the R5 series, which are suited for caches, databases, and other memory-intensive workloads. At the same prices as General Purpose VMs, these tend to feature faster and larger memory.

Storage Optimized VMs These come with fast disk throughput performance, useful for applications like logging, databases, and file storage that require lots of disk I/O.

Accelerated Computing and High-Performance Computing VMs These offer specific combinations of high-performance CPU, GPU and memory to suit very intense compute workloads such as video encoding and physical simulations.

As the descriptions of the deployment options have suggested thusfar, the most suited Kubernetes deployment type depends greatly on the requirements of the workload.

3.2 Mapping to Mobile Core Workload Characteristics

To effectively compare the deployment methods, we review the characteristics that a typical workload may have when mobile operators consider cloud-native mobile core deployments. Two main properties are that **processing demands fluctuate significantly** due to UE connection patterns and that they must adhere to **latency and reliability requirements** that the operator needs.

Fluctuating demands refers to differences in the number of connected devices across both short and long timespans. For example, national-scale large events like disasters, festivals, or sport tournaments may cause short spikes to happen across an operator's network, causing the mobile core to receive more messages. Regular fluctuations such as time of day, public holidays and seasonal demands can also affect the traffic demands on a longer scale. Such increased load in turn demands more processing power in order to maintain the overall network speed. However, it is financially unreasonable to sustain this increased processing power at a period of lower traffic. This is because public cloud provider offerings become costlier when with extra compute power, due to end-user incurred costs on power usage, hoarding resources, etc. As such, the infrastructure must be able to automatically scale on demand, keeping the impact on delay as little as possible in times of exceptional traffic.

The latency and reliability requirements refer to the requirements that a mobile operator imposes on themselves as part of their service level agreements (SLAs) with customers and other services. Typically, the data plane is subject to tighter requirements due to the larger direct impact on end-users (for example, video streaming, gaming, autonomous driving, etc). For 5G networks in particular, the ITU imposes the minimum requirement to be that the one-way end-to-end data plane latency (so, from the UE to the UPF 3rd Generation Partnership Project (3GPP) [2021b]) should be a maximum of 4ms for eMBB and 1ms for URLLC. For the 5G control plane, the ITU defines 20 ms to be the maximum transition time to go from "an idle state to the start of data transfer" (International Telecommunications Union (ITU) [2017]). As per the flow of 5G registration as described in Figure TODO, there are multiple round-trip control plane messages required in order for a UE to transition from its idle "non-connected" state to a state of data plane transfer (NAS Registration Request & NAS Authentication Request, NAS Authentication Response & NAS Security Mode Command, NAS Security Mode Complete & NAS Registration Accept, NAS PDU Session Request & NAS PDU Session Accept). If ITU's recommendation of 20 ms is understood to encompass these four procedures, then each message has to be handled end-to-end in an average of 5 ms. Taking into additional time for messages to transmit through the radio network and for the UE to perform work (such as encryption) on its side, an acceptable average latency for the core network should be even lower, at around 2-3 ms.

The above two characteristics constrain the deployment types that a mobile core can feasibly deploy to. The infrastructure a mobile core has to be flexible in its scaling capabilities to be financially efficient, and must be capable enough to sustain the required latency values.

We will examine the aforementioned three methods of deployment (and its additional options) under the newly explained two constraints unique to mobile cores. The examination will rate each deployment option by **cost impact**, **potential for observability**, and **ease of manageability**. The three criteria were chosen from the goals of this study: cost impact is the most important criteria, and the reason why operators consider cloud deployments in contrast to traditionally costly on-premise setups; observability is important for gathering metrics for business KPIs as well as to monitor for security problems and to diagnose problematic sections of the deployment; and finally, manageability is important for saving on human resource costs and the traditionally large operational expenses.

3.2.1 Analysis of EKS (managed Kubernetes) with Fargate

When Fargate is used in tandem with EKS, each pod that matches the customer-specified criteria will be scheduled as a task on Fargate. However, as Kubernetes operates on the concept of scheduling pods onto nodes, Fargate presents the scheduled tasks as having a dedicated node each, sized specifically for the the task it needs to run, and which exclusively runs the single pod it has been provisioned with. Customers can specify a vCPU and memory configuration for the tasks according to a set of allowed combinations Amazon Web Services [2024a], and are billed by the total sum of all scheduled tasks.

CPU \Mem	0.5 GB	1 GB	2 GB	3 GB	4 GB
0.25 vCPU	$0.00711	$0.01384	$0.02731	$0.04078	$0.05425
0.5 vCPU	-	$0.01421	$0.02768	$0.04115	$0.05462
1 vCPU	-	-	$0.02842	$0.04189	$0.05536
2 vCPU	-	-	-	-	$0.05684

Table 3.1: Fargate Resource Configurations and Pricing per hour for eu-north-1 (Stockholm) in March 2024, based on data from Amazon Web Services [2024a]

EKS charges a constant fee for the automatic provisioning of the Kubernetes control plane ($0.1/h). For the scheduled Fargate tasks, in the eu-north-1 AWS region in March 2024, the pricing was $0.0445/vCPU/hour and $0.0049/GB/hour for CPU and memory resources respectively, resulting in a price matrix for the configurations seen in Table 3.1. The flexibility in pricing ranging from as low as under a cent per hour for the least powerful configuration is greatly beneficial for Kubernetes workloads that scale, enabling operators to be financially efficient.

Note that the table has more columns and rows for larger configurations of CPU and memory, but they have been omitted in this paper to draw attention to the low-resource configurations. Due to the way Kubernetes's Horizontal Pod Autoscaler works, there must be at least one replica of a workload present at any moment, and all replicas must have the same resource configurations. It is therefore inefficient to request a larger amount of CPU and memory per replica and be unable to scale smaller, especially as any reasonable core network implementation would have a low CPU usage when the network is at the lowest load (such as in rural areas at night).

An issue with using EKS with Fargate is that the scheduled Fargate tasks require some overhead in connecting with the Kubernetes control plane, incurring up to 256MiB higher memory use as well as some CPU use. This manifests as a higher cost than the ideal resource usage of each pod. Further, the limited configuration options means that one cannot run pods of small sizes efficiently. This is especially problematic for highly modularized mobile cores with many different components (such as a scalable implementation of the CNF architecture), as each component may not require a large amount of resources individually. Another limitation is that the underlying processor and memory details are not published to the public, nor are they likely consistent, making it difficult for operators to get a reliable performance out of their core components.

There are also problems with exposing the port 38412 SCTP interface for UEs/RANs to connect to. Usually, Kubernetes pods can define its open ports through the YAML configuration in three different ways: 'NodePort', 'HostPort', or through a Load Balancer. 'NodePort' is used to open a port on every node, where traffic gets routed to the correct pod internally; 'HostPort' is used to open a port on the single node that a pod runs on; and a Load Balancer assigns an IP or DNS name as an alias for multiple pods, and performs load-balancing traffic routing to the pods.

However, Fargate does not support 'HostPort', leaving us with two ways of opening the SCTP port. The default port range for 'NodePort's as defined by the Kubernetes control plane configuration is 30000–32767, and it is not possible to change this with a

managed Kubernetes control plane (Ian Smith [2021]). Furthermore, Load Balancers on AWS do not support the SCTP protocol. All in all, this means that a core network cannot exposed if it is deployed on just Fargate. An operator needs at least one non-Fargate node deployed through EC2 for a functional SCTP server. In the case of low amounts of traffic, the extra space on EC2 can incur wasted financial resources.

In terms of observability, Fargate is limiting as well. The reason is its abstraction over the underlying server, which makes detailed tracing and system-level performance monitoring impossible to do. For example, a common approach to debugging anomalistic network behaviour in a Kubernetes pod is to attach what is called a 'sidecar container' with network monitoring capabilities (such as a WireShark instance) to see the network packet traces. A sidecar container runs in the same Kubernetes pod (and Fargate task) with the main application container, but with escalated privileges in order to log network and system traces using Linux capabilities. Such useful method to debugging is not possible on Fargate however, because AWS disables all privilege escalation, citing security concerns and process isolation (Amazon Elastic Container Service Developer Documentation [2024]). Somewhat ironically, the lack of system visibility makes the deployment less secure for mobile cores that comprises of multiple third-party components, as it becomes difficult to monitor and diagnose for anomalies in the containers.

The saving grace of this deployment method is that the the "ready-to-run" nature of the managed Kubernetes service and the serverless task provisioning marry nicely to make a strong case for ease of manageability. The maintenance required for security patches in Kubernetes control planes and underlying server OSes is nonexistent with this approach, reducing both the depth in specialised knowledge required and the breadth of the work required just to maintain an existing deployment.

Overall, the managed Kubernetetes deployment with Fargate is a strong contender in terms of manageability and the ability to handle fluctuations in a financially efficient manner, but comes with equally strong drawbacks in its limited observability and actual cost-efficiency when viewed holistically.

3.2.2 Analysis of EKS (managed Kubernetes) with EC2

The AWS EC2 offering works by customers choosing VM types to provision, and the managed Kubernetes control plane scheduling the pods to run on those nodes. Unlike Fargate, these are more similar to traditional Kubernetes setups in that the nodes can host multiple pods, as long as the total requested resources does not exceed that of that node. By default, AWS imposes soft limits on the number of pods an EC2 node can host based on the number of attached network interfaces (Amazon Web Services Labs GitHub [2024]), however this can be increased if necessary using 'Prefix Delegation', a feature where EC2 nodes can be given IPv4 prefixes instead of a single address (Amazon EKS User Guide [2024]).

As described previously, EC2 offers a range of VM types to choose from. Among the various optimized VM types, the General Purpose and Compute Optimized are the most suitable for cloud-native mobile cores. Memory Optimized machines are above our

requirements as cloud-native software are inherently largely stateless in order to scale, and thus have a small persistent memory footprint. Storage Optimized, Accelerated and High-Performance Computing VM types are also not suited for our use for similar reasons. Generally, even across non-stateless implementations, a mobile core control plane is mainly CPU-bound (Sama et al. [2015]). Based on this criteria, we filter our list of potential VM types to those listed in Table X. Similar to our analysis of Fargate configurations in Section 3.2.1, we focus our attention on the low-resource configurations in order to reduce idle resources when the core is under low load.

Importantly, EC2 offers two prices for each VM type – On Demand and Spot. On Demand variants are always available when requested and have a stable pricing scheme. They allow operators to have full control over the lifecycle of the VM, which makes it suitable for high-availability workloads and those with persistent data. On the other hand, Spot variants come with a much cheaper cost (often at half price or less compared to On-Demand variants) and run with the same machine specifications as On-Demand variants, but risk not being available when requested or being shut off at short notice due to other customers' demands. Spot VMs are naturally suited for stateless workloads, as such any stateless component of the cloud-native core should have no problems in operating on it. However, to maintain network availability in case there are no Spot VMs available, as well as to host the stateful components such as the customer database, at least one On-Demand node should be used when deploying for production.

Instance	vCPUs	GB RAM	On-Demand $/h	Spot $/h
General Purpose				
t3.nano	2	0.5	0.0054	0.0052
t3.micro	2	1	0.0108	0.0103
t3.small	2	2	0.0216	0.0209
t3.medium	2	4	0.0432	0.0240
t3.large	2	8	0.0864	0.0369
m5.large	2	8	0.1020	0.0335
Compute-Optimized				
c5.large	2	4	0.0910	0.0396
c5a.large	2	4	0.0820	0.0238
c6g.medium	1	2	0.0365	0.0094
c6g.large	2	4	0.0730	0.0234
c6i.large	2	4	0.0910	0.0270
c7g.medium	1	2	0.0387	0.0100
c7g.large	2	4	0.0774	0.0174
c7a.medium	1	2	0.05494	0.0126

Table 3.2: General Purpose and Compute-Optimized VM types and their pricing per hour for eu-north-1 (Stockholm) in March 2024, based on data from Amazon Web Services [2024a] and Amazon Web Services [2024b]

The total cost for a deployment on EKS with EC2 is a sum of $0.1/h for the automatically provisioned Kubernetes control plane, and the EC2 VM costs from Table 3.2. Recall that the most minimal Fargate resource configuration for a Kubernetes pod was 0.25

vCPU and 0.5 GB of memory for $0.0071/hour. On an EC2-based deployment using VMs that come with 2 vCPU and 4GB memory, theoretically up to 8 such pods can be scheduled at a lowest cost of $0.0432 (`t3.medium`) with on-demand and $0.0240 with spot variants. Realistically however, due to some Kubernetes node overhead, the pod limit may be 6–7, in which case the per-pod cost is $0.0650 or $0.0037/hour, which makes the price roughly equal to Fargate costs with on-demand pricing, and about half as much with spot pricing. This is lucrative in itself, but it can be even further optimized, as many pods do not need the full 0.25 vCPU and 0.5 GB of memory. In fact, Larrea et al. [2023] saw no improvement in performance when allocating above 0.05 vCPU per pod for a stateless mobile core worker pod. This means that an EC2 VM can potentially host many more pods compared to Fargate at the same price, making this deployment very cost-efficient.

Scaling flexibility is a concern when deploying workloads to EC2 nodes. This is because the Kubernetes Horizontal Pod Autoscaler can replicate workloads across all of the nodes registered with the control plane, but is not capable of starting or stopping nodes to adjust the overall capacity. Overprovisioning or underprovisioning EC2 VMs can lead to high incurred costs or performance losses, both of which can impact an operator's ability to meet customer satisfactions. Thankfully, the Kubernetes Cluster Autoscaler project (Kubernetes [2024]) can perform this additional level of scaling, and spin up or down EC2 VMs in what it calls 'node groups'. These node groups can specify a minimum and maximum number of EC2 VMs to provision along with the type of VM. Cluster Autoscaler will continue to monitor the whole cluster for resource shortage, and provision a new node to the node group or remove an existing one. Whenever Cluster Autoscaler adjust the number of nodes, new nodes are automatically connected with the EKS Kubernetes control plane, making it fully automatic and seamless for the customer when scaling nodes. Similarly, a removal of a node will gracefully move any Kubernetes pods on it to other existing nodes. With this set up, the manageability burden of the cluster lessens drastically, and the deployment is able to handle fluctuating demands at a highly dynamic scale.

Observability potential is higher with pods deployed to EC2 than with Fargate. The customer gets access to the underlying VM through remote interaction protocols like SSH, which means that they can install monitoring agents or other privileged software. Such software can provide insight into the performance of the VM and the Kubernetes pods that it is running. This is in stark constrast to Fargate deployments where the VM was abstracted away.

In summary, the managed Kubernetes deployment with EC2 VMs compute engines is a strong all-rounder that excels in cost efficiency and observability, all the while not sacrificing any of Fargate's lucrative flexible scaling or ease of manageability.

3.2.3 Analysis of self-managed Kubernetes with EC2

The final method of deploying a Kubernetes-based workload is to self-manage the entire stack, letting AWS only manage the EC2 VMs. This way, the customer can have full control over their Kubernetes control plane, allowing for custom-tweaked deployments if necessary. Having full access to both the nodes and the cluster control plane makes

the self-managed deployment option very observable.

Unfortunately, configuring and maintaining a Kubernetes control plane is a lot of work, especially when taking into account security upgrades and high-availability requirements (DZone Limited [2023]). This requires both hiring specialized engineers as well as continuously spending developer time on the maintenance of a cluster. In addition, from a technical perspective, the Kubernetes control plane must be hosted on a machine that exclusively acts as the master node. The Kubernetes master node cannot schedule pods onto itself, meaning that this deployment method will naturally need at least one VM dedicated for the control plane. Taking all of these difficulties into account, the operational costs of running a self-managed Kubernetes deployment often outweigh the service cost of the managed Kubernetes services (e.g. $0.1/h for EKS).

Furthermore, as a matter of principle, when the public cloud is treated as simply a VM host, it is not much different from private cloud deployments. Many of the network configurations and VM setup required to make an AWS EC2 node support Kubernetes pod deployments are the same for an on-premise cloud or a rented remote server. The similarity voids most of the point of using the public cloud, as there is no benefit to be gained in ease of manageability or developer effort. The cost may be cheaper compared to the traditional options, but the benefits of the public cloud are not utilized fully to be truly cost-efficient.

In summary, using a self-managed Kubernetes control plane with EC2 VMs is not a recommended approach based on the drawbacks in almost all criteria. It can perhaps serve as a useful transition deployment while a privately managed Kubernetes cluster is upgraded to run on the public cloud, but that is the extent to which this method proves beneficial. In the rest of the report, this deployment method will be omitted from comparisons.

3.2.4 Summary: Hypothesised “best” Deployment Method

Table 3.3 was created based on the above analysis. We hypothesise that deploying a mobile core to EKS with EC2 VMs as the compute backend will be the best option for cost efficiency, observability, and manageability.

3.3 Deployment Evaluation Procedure

In order to evaluate the different deployment methods quantitatively, we took to deploying a functional and production-ready mobile core (CoreKube) to AWS to analyse the performance and cost under some simulated load. CoreKube’s architecture is representative of a cloud-native mobile core, featuring stateless components that scale to responding demands. This was especially verified true based on how little re-engineering was required of the CoreKube codebase to support public-cloud deployment. For example, based on the analysis of each deployment method performed in the previous section, it was determined necessary for there to be a lightweight internet-facing core component that handles the incoming SCTP traffic before routing to either EC2-based or

Criteria	Self-managed with EC2	EKS with Fargate	EKS with EC2
Cost	≈ $0.025/hr 8pods	$0.1/h + ≈ $0.007/hr 1 pod	$0.1/h + ≈ $0.025/hr 8 pods
Cost efficiency	Good	Not very good	Good
Ease of Manageability	Difficult	Easy	Easy
Observability	Full Control	Limited	Nearly Full Control

Table 3.3: Summary of the qualitative analysis of the three Kubernetes deployment methods on AWS

Fargate-based components. CoreKube already has a compatible architecture, featuring a light-weight SCTP front-end, a scalable stateless worker, and a database.

Another choice instead of CoreKube was to use ECHO (Nguyen et al. [2018]), due to their past success in deploying to Microsoft Azure. Unfortunately, ECHO is not well-optimized as a cloud-native software as Larrea et al. [2023] points out. Specifically, ECHO claims it is stateless and supports auto-replicating to handle demand, but the implementation always routes messages from the same UE to the same component, easily leading to situations where some instance replicas are underutilized while others are overloaded. This limitation undermines any attempt to measure performance, resource usage and cost with it, since the results can be highly skewed and dependent on the UE connection patterns. In contrast, CoreKube is truly stateless and each worker can handle any incoming message, allowing the instance-level performance, resource usage, and incurred costs to be independent of any UE or connection pattern, which allows easy extrapolation to larger traffic demands.

We describe here the procedure for actually deploying CoreKube to AWS EKS with Fargate and EKS with EC2, and will then proceed to evaluate the two deployments through quantitative comparisons. Figure 3.2 describes the deployment setup for using Fargate while Figure 3.3 describes it for EC2.

Setting each deployment up individually requires multiple complex invocations of the `aws` CLI tool executed in order, or navigation through the complex AWS web interface. This is prone to user error (where mistakes incur costs) and subject to difficulty in recreating environments reliably. As a solution, we used Terraform to orchestrate the setup. Terraform is an open-source tool maintained by HashiCorp for abstracting vendor APIs into simplified JSON-like specifications, that when applied deterministically create the specified cloud environment. It is widely used in industry for its ability to do incremental upgrades, setup machines deterministically, and for the vast supply of plugins that interface with various cloud provider APIs, including ones for AWS.

Firstly, to ensure connectivity between various components, we used an AWS Virtual Private Cloud (VPC) to provision a local network with 3 private and 3 public subnets, as well as an Internet Gateway and NAT Gateway. The Terraform code that describes

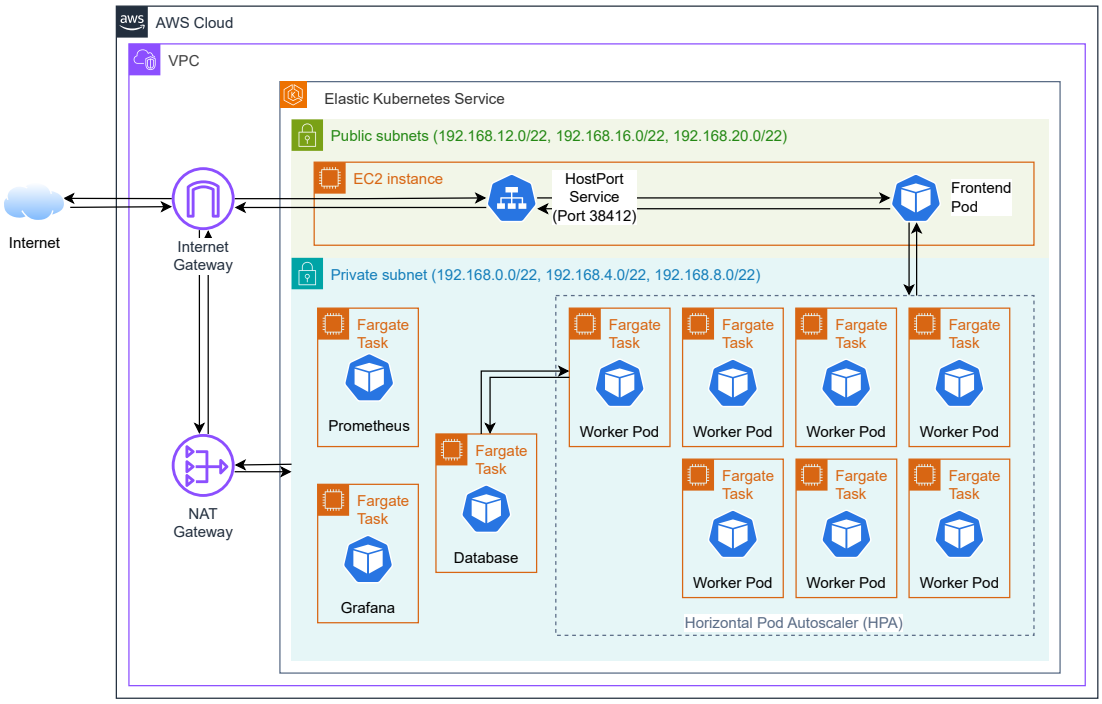


Figure 3.2: AWS deployment diagram for deploying CoreKube on EKS with Fargate

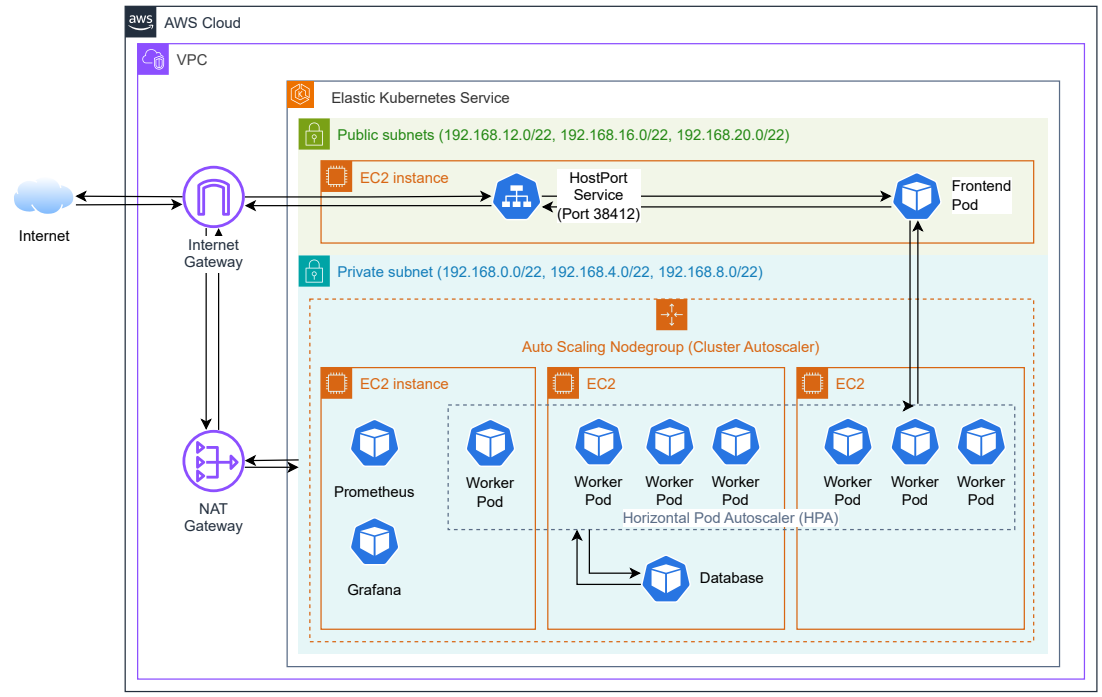


Figure 3.3: AWS deployment diagram for deploying CoreKube on EKS with EC2

this VPC setup can be found in `cli/terraform/base/setup.tf`. The private subnets route externally directed traffic to the NAT Gateway, which then routes it to the Internet Gateway for access to the wider internet. The public subnets route externally directed traffic to the Internet Gateway directly. The difference is in the allocation of public IPv4 addresses. Because the Internet Gateway is just a traffic router and does not provide network address translation (NAT) capabilities, all services within the public subnets are assigned a unique public IPv4 address. The property makes it suitable for publicly accessible services such as the mobile core frontend, which needs to have an public IP address with port 38412 open for NGAP traffic. On the contrary, because private subnets are behind a layer of NAT, they are only assigned internal IPv4 addresses and the services within it are not reachable directly by IP. This makes it suitable for securing workloads from external access, such as other mobile core components that only need to interface with each other and not the wider internet.

The six subnets were spread across multiple AWS availability zones, which ensured that a network failure in one physical AWS data-center could promptly hand over to another subnet to retain connectivity with both the rest of the cloud and wider internet. They were assigned an address range of /22 each, allowing for up to 1024 different IPs to be allocated within the subnet. This range was arbitrarily chosen as a result of some performed experiments, which measured roughly the number of services that the network would be hosting at its peak load. Since choosing a wider range does not incur higher costs, operators are encouraged to make sure that their VPC has enough address ranges for their own deployment.

The managed Kubernetes control plane (EKS) was deployed into the aforementioned VPC, as both methods under evaluation required it. The relevant Terraform code for EKS is in `cli/terraform/base/ck_cluster.tf`. As described in the analysis of each deployment method (3.2.1 and 3.2.2), the EKS cluster is required to have at least one EC2 node to host the internet-facing component of the core at SCTP port 38412. We decided to provision the `t3.small` VM type for it, which is a small general-purpose VM with 2 vCPUs and 2 GB memory.

For the Fargate-based deployment, the Terraform code further describes an AWS Fargate Profile, a description of what Kubernetes pods should be scheduled as a task on Fargate. This is not enough for a functional deployment to Fargate however, as by default the Fargate tasks are restricted in their access to other resources (such as the internet-facing EC2 node). To allow communication between the EC2 node and the Fargate tasks, the operator must add a rule to the VPC firewall system ('Security Group Rule'), which is done through a Terraform definition as well.

For the EC2-based deployment, an installation of Kubernetes Cluster Autoscaler (Kubernetes [2024]) was used to dynamically adjust the number of EC2 nodes depending on how much the Kubernetes pods scaled by. This is specified through a `min_size` and `max_size` of the fleet of nodes, and acts as a secondary layer of scaling on top of the Kubernetes Horizontal Pod Autoscaler (which scales pods onto existing nodes).

```
emolga ➜ public-cloud-core/cli git:(master) ✗ ./publicore loadtest --file scripts/workloads/100-100-switchoff-ck.json --col
lect-avg-latency --collect-worker-count --collect-cost --collect-avg-throughput --experiment-name tmp/test-1000-1000
00:38:18.942462042 [32330322] INFO - Running loadtest with file: scripts/workloads/100-100-switchoff-ck.json
00:38:22.183889672 [32330322] INFO - Deployment Type: fargate
00:38:22.184332049 [32330322] INFO - Core Fargate Worker Provisioned Size: 1vCPU 2GB
00:38:22.184333340 [32330322] INFO - RAN Emulator EC2 Instance Types: ["c5.large"]
00:38:22.184334132 [32330322] INFO - RAN Emulator EC2 Capacity Type: ON_DEMAND
00:38:22.421813372 [32330322] INFO - Loadtest started.
00:38:22.422010914 [32330322] INFO - Press Ctrl-C to stop the loadtest.
00:38:22.422022164 [32330322] INFO - Collecting data ... 0/200 (1000 seconds remaining)
00:38:22.537156977 [32330322] INFO - No latency data available yet. Extending experiment.
00:38:22.838714775 [32330322] INFO - UE count: 0
00:38:27.421981122 [32330322] INFO - Collecting data ... 1/201 (1000 seconds remaining)
00:38:27.531817955 [32330322] INFO - No latency data available yet. Extending experiment.
00:38:27.823603962 [32330322] INFO - UE count: 0
00:38:32.421912080 [32330322] INFO - Collecting data ... 2/202 (1000 seconds remaining)
00:38:32.516678142 [32330322] INFO - No latency data available yet. Extending experiment.
00:38:32.825905513 [32330322] INFO - UE count: 0
00:38:37.421910622 [32330322] INFO - Collecting data ... 3/203 (1000 seconds remaining)
00:38:37.535514553 [32330322] INFO - No latency data available yet. Extending experiment.
00:38:37.824070131 [32330322] INFO - UE count: 0
00:38:42.421910413 [32330322] INFO - Collecting data ... 4/204 (1000 seconds remaining)
00:38:42.516480307 [32330322] INFO - No latency data available yet. Extending experiment.
00:38:42.817583562 [32330322] INFO - UE count: 0
00:38:47.421905663 [32330322] INFO - Collecting data ... 5/205 (1000 seconds remaining)
00:38:47.518344857 [32330322] INFO - No latency data available yet. Extending experiment.
00:38:47.816457224 [32330322] INFO - UE count: 0
00:38:52.422005414 [32330322] INFO - Collecting data ... 6/206 (1000 seconds remaining)
00:38:53.657709354 [32330322] INFO - UE count: 8
00:38:57.422058659 [32330322] INFO - Collecting data ... 7/206 (995 seconds remaining)
00:38:58.635792466 [32330322] INFO - UE count: 54
```

Figure 3.4: Sample output from `publicore` when running a load simulation

```
emolga ➜ public-cloud-core/cli git:(master) ✗ ./publicore deploy --pretty --type aws-eks-fargate
Installing modules for infrastructure ... Complete!
Applying changes to infrastructure ... Complete!
Registering clusters with kubeconfig ... Complete!
Installing modules for Kubernetes ... Complete!
Applying changes to Kubernetes ...
```

Figure 3.5: Sample output from `publicore` when creating a new deployment

3.3.1 Publicore: A CLI Tool for Managing Cloud-Native Mobile Cores

Although Terraform works well for the provisioning of CoreKube and cloud-native software in general, there was still a lot of manual work required after every deployment to put them under load and evaluate them, including the setup of the Nervion RAN emulator, running workload tests, and collecting data. To this end, we created `publicore`, a command-line tool written in C++ to provide a wrapper for the most common operations performed during the evaluation of a mobile core deployment to the public cloud. A sample output of the program is given in Figure 3.4, and the source code of this tool can be found in the `cli` directory. We hope that this tool aids operators and other researchers in the evaluation of their own mobile core setup when deployed to AWS.

`Publicore` requires Terraform and the Kubernetes CLI to be installed as a prerequisite, and is capable of performing automated deployments (see Figure 3.5) of the core under test (CoreKube) and the load simulation tool (Nervion RAN emulator), starting and stopping a simulated load, collecting Prometheus metrics at periodic intervals to write to CSV files, and to destroy a deployment as necessary after its evaluation. The tool is built with a modular structure where each C++ class handles a sub-command as specified at the command-line, and is easily extensible to multiple cloud providers and

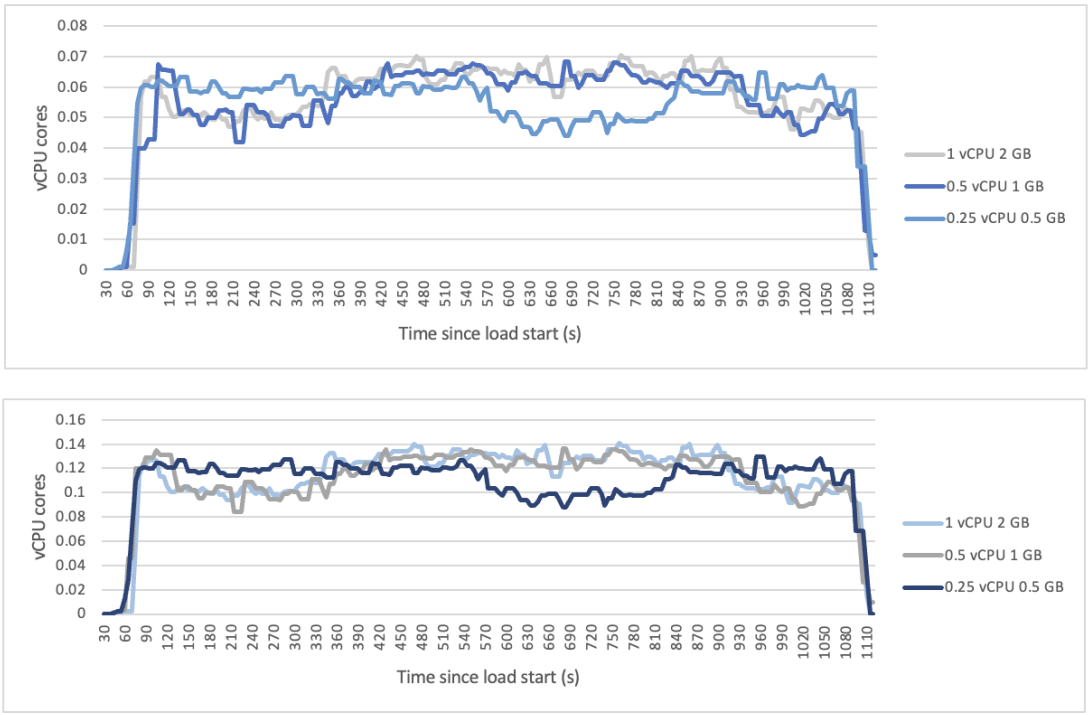


Figure 3.6: Top: Average vCPU usage per Worker pod under a load of 100 UEs, for different Fargate resource configurations. Bottom: Total vCPU usage per Worker pod under a load of 100 UEs, for different Fargate resource configurations.

core implementations through the addition of underlying Terraform files. `Publicore` also features many command-line flags to customize the data collection behaviour and simulated loads.

3.4 Quantitative Comparison using CoreKube

Following the methods to deploy mobile core infrastructures using Terraform and `publicore`, we performed a quantitative comparison of deploying CoreKube to AWS EKS with the Fargate backend and with the EC2 backend. The CoreKube deployment was put under a load of 100 emulated UEs simultaneously connecting and disconnecting in rapid succession while the average packet latency and resource usage was gathered.

First, Figures 3.6 and 3.7 presents the average and total CPU usage of the auto-scaling worker component, as well as the average packet latency under three different Fargate configurations: 0.25 vCPU and 0.5 GB memory, 0.5 vCPU and 1 GB memory, and 1 vCPU and 2 GB memory. The configurations were requested by changing the amount of vCPU requested in the Kubernetes pod description YAML file. The cost for each of these are $0.00711, $0.01421, $0.02842 for the three configurations respectively.

The figures show an average packet latency much higher than acceptable based on the latency requirements set out in Section 3.2 (2-3 ms), ranging between 10 to 100ms. Note the logarithmic scale used on the vertical axis to contain the variety in latency. Surprisingly, allocating a higher amount of resources does not improve this latency to

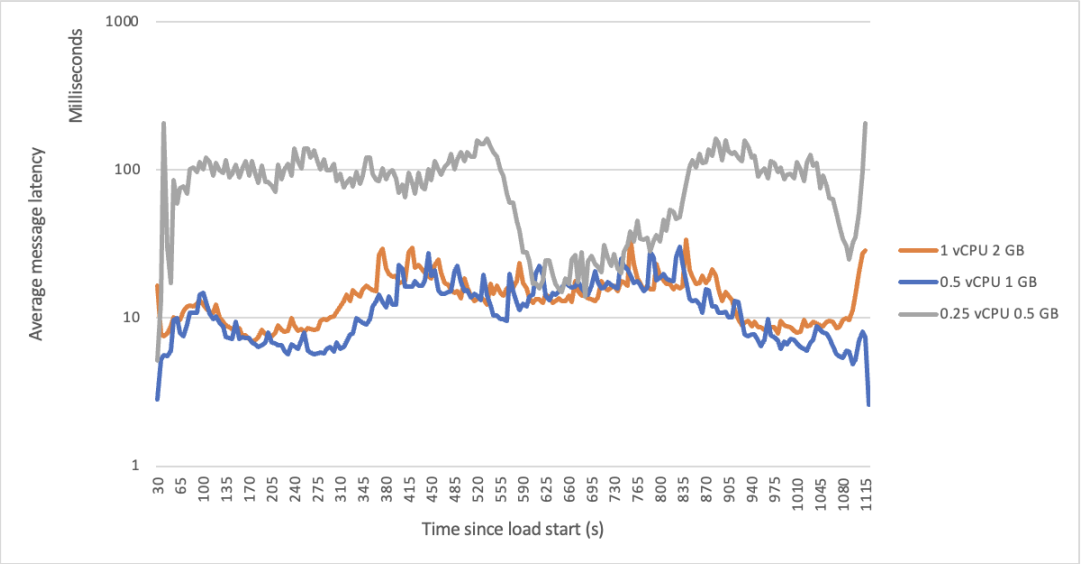


Figure 3.7: Average control plane message latency under a load of 100 UEs for different Fargate resource configurations.

tolerable levels, indicating that the additional resources are not being used effectively.

The vCPU usage seen in 3.6 is much lower than the allocated amount, both in average and total. Throughout the experiment, the total usage is double that of the average, which is a result of the worker not auto-scaling and staying with its minimum replication count. This is an indication of resources being over-provisioned and cost-inefficient, as the incurred costs during idle time and peak load time are the same. For a deployment to be cost-efficient, it must scale down if the required resources are small, but Fargate does not let us scale to below 0.25 vCPU and 0.5 GB memory at any time. It also does not scale upwards or make use of the larger vCPU available to improve the latency, suggesting that there are other bottlenecks.

From these results, it can be concluded that a cloud-native core deployed onto EKS with Fargate results in very high latency despite heavily underutilised resource usage. Some possible causes for the high latency are network delays on communication between pods on Fargate and EC2, as well as the underlying processor speed. The unacceptably high latency results undermine all qualitative benefits with the Fargate approach outlined in Section 3.2.1, such as manageability and the ability to handle fluctuations.

Next, Figures 3.8 3.9 show the vCPU usage and average message latency for deployments made onto EC2, compared across four types of EC2 VMs: `t3.small`, `t3.medium`, `t3.large`, `m5.large`, `c5.large` and `c5a.large`. These types were chosen based on the list in Section 3.2.2. Note that `t3.nano` was attempted as well, but was very prone to memory exhaustion problems, making it unsuited for a test deployment.

The figures show that small general-purpose VM types do not meet the latency requirements of 2-3 ms. However, generally, any VM types with at least 2 vCPU cores and 2 GB memory tended to perform well. It is worth noting that although `t3.medium` has a theoretically better configuration than `t3.small`, its average latency was much higher.

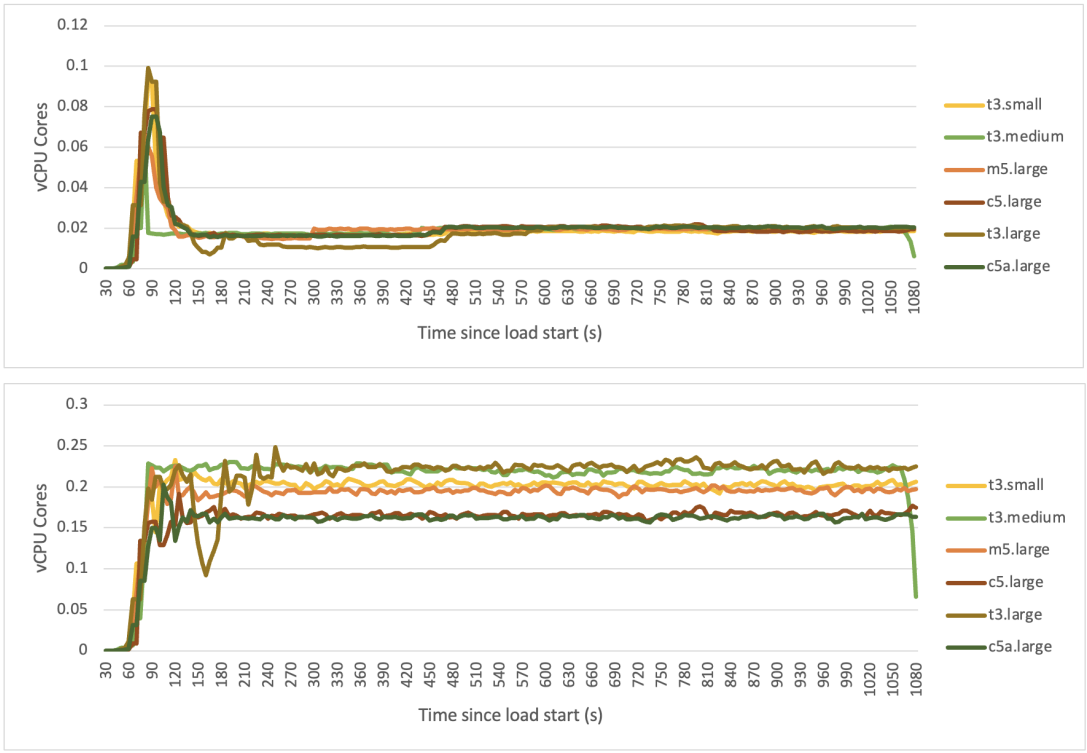


Figure 3.8: Top: Average vCPU usage per Worker pod under a load of 100 UEs, for different EC2 VM types. Bottom: Total vCPU usage per Worker pod under a load of 100 UEs, for different EC2 VM types.

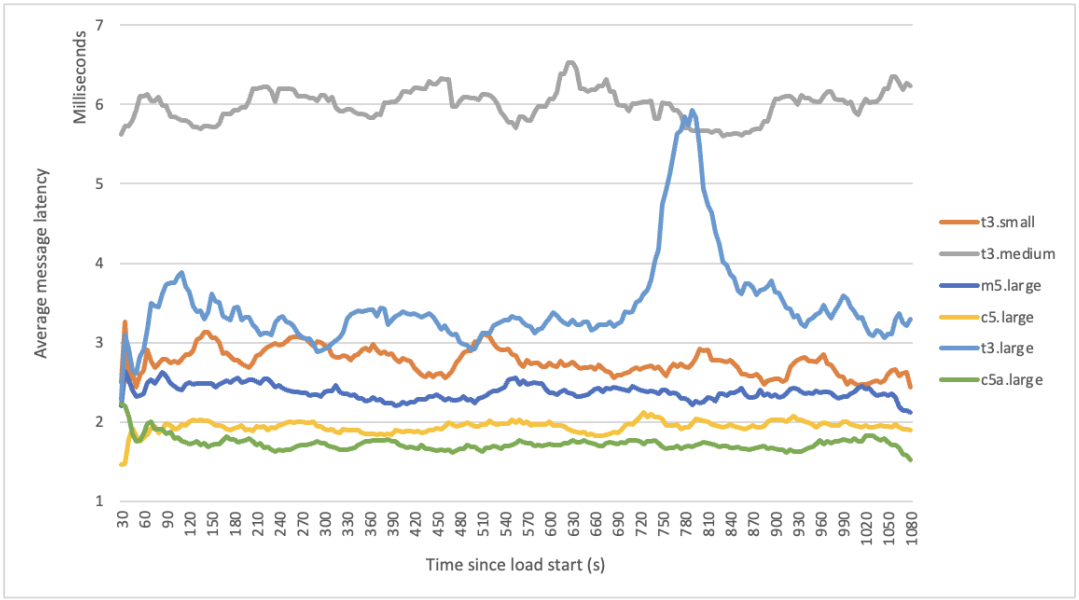


Figure 3.9: Average control plane message latency under a load of 100 UEs for different EC2 VM types.

The reason for this is unknown, but is likely attributable to the burstable nature of the `t3` family (Amazon Web Services [2024b]), which allows it to sustain a higher CPU clock speed for a predefined amount every hour. We can see that the `t3.large` instance briefly reaches similar latency levels as the `t3.medium` instance, suggesting that it may have returned to its original CPU speed before bursting again. This attribute is quite unpredictable however, and the compute-optimized instances appear to provide a much more stable deployment.

When it came to resource usage, general-purpose VMs required higher amounts of total vCPU to handle the same 100 UEs. The average vCPU usage per worker pod was at similar levels for all VM types, indicating that the general-purpose VMs had required more replicas of the worker pod to handle the load compared to compute-optimized instances. This is also verified in Figure 3.10, which shows that the four general-purpose instances `t3.small`, `t3.medium`, `t3.large`, and `m5.large` had the highest number of worker pod replicas. Note that the `t3.large` and `m5.large` had very similar data, and only one can be seen as a result.

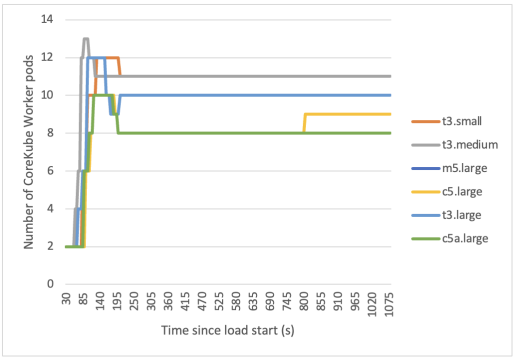


Figure 3.10: Number of CoreKube workers automatically increasing to handle the load of 100 UEs under different EC2 VM types.

These quantitative and experimental analysis performed indicated the feasibility of a functional deployment of CoreKube to AWS, as long as the deployment method is carefully optimized to avoid using Fargate or small general-purpose EC2 VMs. As a final verification, we ran tests to verify that CoreKube's auto-scaling and self-healing properties still held true, and compared the results against the data in the original paper. The data in the original paper were gathered inside Powder Platform (Breen et al. [2020]), a private cloud environment.

Figure 3.11 shows the resiliency of CoreKube thanks to its rapid self-healing. At the two highlighted timestamps, one CoreKube worker was deliberately crashed by sending a predetermined malformed packet. However, the figure shows no major changes in latency post-crash, neither did it decrease the worker pod count. This is attributable to the fact that the Kubernetes Horizontal Pod Autoscaler detected the crash and immediately replaced the worker pod within the 5 seconds that was the metrics collection frequency during this experiment. As a result, the figure seems almost non-problematic, demonstrating that CoreKube's resilience and self-healing properties are still functional in the public cloud deployment.

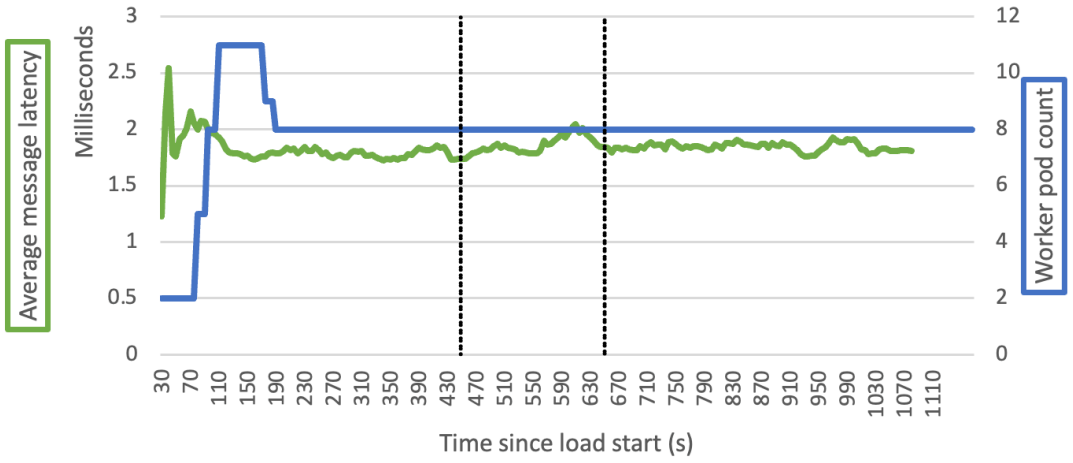


Figure 3.11: The average message latency and CoreKube worker pod count. A worker pod was deliberately crashed at the two highlighted timestamps.

3.5 Limitations in Observability

Although we could analyse and compare the deployment methods based on resource use, message latency and estimated cost, none of these metrics provide additional application-level insight into the causes of unexpected delays or throttled vCPU use. Generally, as it stands we are only able to analyse a mobile core deployment in black-box approaches universal to all cloud-native software, such as Kubernetes pod-level resource usage and network traffic trace-based analysis.

This is especially limiting when these deployments are put to production use. 3GPP defines multiple Key Performance Indicators (KPIs) for operators to monitor in their core, including the UE registration success rate and the number of connected UEs at any given moment. These values are also valuable for analysing the deployment methods for their cost efficiency and for finding bottlenecks.

In Chapter 4, we will show an implementation of a metrics collection system specifically designed for cloud-native mobile cores, which aims to address these issues.

3.6 Future Work

3.6.1 End-to-End Feasibility with COTS UE

As demonstrated in the previous sections, the deployment of cloud-native mobile cores (aided with `publicore`) looks promising from a performance and cost-efficiency point of view. However, there remain a challenge in order to make it functional as a connectable core from commercial off-the-shelf (COTS) UEs.

In principle, as the deployed core has a public IP, there should be no problem for a local device to attach to it like any emulated device, and perform the NGAP registration procedure over SCTP. In practice however, we encountered issues with SCTP connectivity between residential networks and cloud provider networks. The likely reason for

this is the existence of routers and NAT boxes between the connections, which tend to drop SCTP traffic (Andrew Ferguson [2024]). Consequently, COTS UEs are not able to connect to the core running in the cloud.

A solution to pursue in the future is to find a location that has SCTP connectivity to both the cloud providers and local internet. With such location identified, a lightweight proxy core that runs there can forward both up- and down-link packets as necessary. However, between residential networks and AWS, there exist no location that satisfies that criteria.

Another solution is to run a proxy at a location that can be connected via SCTP from the COTS UEs, and have the SCTP traffic tunnelled to the mobile core over UDP. A similar approach is adopted within CoreKube internally for its communication between the frontend and the workers, and is also provided as a configurable option in major SCTP libraries (Randall Stewart and Michael Tuexen [2015]). As it involves no deep packet inspection, the UDP-SCTP tunneling should be a viable solution that does not sacrifice performance or latency. However, this solution makes the core interface no longer conforming to the 3GPP standards, which may limit interoperability with other components as well as adoption by operators.

3.6.2 Hybrid Deployment with a Local Data Plane

For a UE to successfully maintain an established connection, the core must supply the location of a data plane. In traditional mobile cores, this is the location of the UPF (User Plane Function) within the core. However, the latency requirements of a data plane are much more demanding than the control plane, and as it stands, it is not certain whether the perceived latency of connecting to a public core for data would be too large.

A remaining work for the future is to attempt a deployment where the data plane is deployed to a location closer to the UE, limiting the public cloud deployment only to the control plane. Such a hybrid deployment would provide the operators with the benefits of both a cost-efficient control plane as well as a fast data plane that satisfies customer demands.

3.7 Summary

We began this chapter with an overview of the deployment methods available to us on AWS when considering cloud-native mobile core deployments. Our qualitative analysis hypothesised that using a managed Kubernetes solution (EKS) together with the pod-level auto-scaling infrastructure (Fargate) would be the method to choose based on manageability and ability to handle fluctuations.

We then took to deploying CoreKube to AWS as a representative cloud-native mobile core. However, the results were not in favour of Fargate, as the unacceptably high average message latency undermined all qualitative benefits that the deployment method was supposed to bring. In the end, deploying cores onto EKS with EC2 VM backends appear to be the most balanced and feasible solution in terms of performance, cost efficiency, and manageability.

As highlighted in Section 3.5, collecting only the average message latency and the vCPU usage does not offer great insight into the mobile core's performance. This observability limitation is the one remaining factor in making public cloud deployments viable for operators. In Chapter 4, we attempt to tackle this through introducing an observability system for cloud-native cores.

Chapter 4

Yagra: A Solution to Cloud-Native Core Observability

Following the successful deployment of a cloud-native core to the public cloud in the previous chapter, we noted the lack of software tools to achieve observability in the core from a business and monitoring perspective. As it stands currently, overall computing performance (such as vCPU and RAM usage) can be collected through generic metrics collection such as Kubernetes Metrics Server (Kubernetes Project Instrumentation Special Interest Group [2024]), and this was indeed the method used to collect the data for the experiments outlined in Section 3.4. However, these metrics are too generic to be indicative of the internal processing that the core undergoes. For example, it is difficult to detect or trace logical bugs or abnormal performance in limited parts of a system, a scenario that can arise as a result of multiple vendors supplying software for various components of the mobile core. Similarly, the lack of mobile-specific metrics make it difficult for commercial network operators to make business decisions, as many of the 3GPP-defined Key Performance Indicators (3rd Generation Partnership Project (3GPP) [2021a]) of an operational network require core-internal data to calculate.

In traditional mobile cores, the solution to this problem is to use a combination of proprietary software tools and vendor-specific core components to monitor systems (InfoVista [2022], NETSCOUT [2024]). However, these go against the principles of cloud-native software development, which aims to be vendor-agnostic and open-source. Furthermore, these solutions are not designed for public cloud deployments, limiting the avenue in which smaller operators with less financial resources can start operations in. Thus, there is a significant gap in the monitoring ecosystem for cloud-native and public-cloud-compatible mobile cores.

In this chapter, we present Yagra, a solution to the problem of observability in cloud-native cores. Yagra composes of a C library to integrate with existing core components, and a Docker image to deploy where collected metrics are aggregated. It is designed around three principles: to be **cloud-native**, by being containerized and supporting metric collection from multiple replica instances of other components; **vendor-agnostic**, by exposing a minimal C interface that is easily integrated with existing core implementations; and **open-source**, making the full development cycle available on GitHub.

We will first go over the architecture of Yagra in Section 4.2, and briefly compare it to other solutions. Then, we will explain our implementation of the system in Section 4.3 including the C library `libyagra` and the Yagra metrics bus, and explain some of the choices we have made. Next, we will evaluate the results collected from the observability system during simulated loads and determine whether the metrics are accurate in Section 4.4. Finally in Section 4.5, we mention the future work for Yagra.

4.1 Goals

The main goal of an observability system when applied to a mobile core deployment is to enable operators to gather data with regards to business/operational KPIs, core performance, and especially with public-cloud deployments, incurred and estimated cost. The three goals are explained in detail below:

Business/Operational KPIs These refer mainly to the KPIs that 3GPP defines (3rd Generation Partnership Project (3GPP) [2021a] for 5G) as being useful to monitor for effectively operating a mobile core network. Examples include the latency in the core network, activity such as mean registered subscribers and mean active device connections, as well as security and quality-of-service indicators like success rate of registration requests. A commercial operator would see the value in these data as it provides concrete proof that they meet their SLAs, insight into the quality of their provided service, indication of suspicious activity or DDoS attacks, as well as the general ability to plan business strategies based on the data. Similarly, a private operator (including educational/industrial) of a mobile network would find value in similar data for the purpose of network optimization and security.

Core Performance Although generic black-box monitoring solutions work well for monitoring the overall resource usage of a deployment (such as at the Kubernetes pod-level or node-level), application-specific performance data would be beneficial. This includes implementation-specific data such as the average database access times, NGAP packet decode times, or average allocated memory while processing a packet. This is useful for debugging and optimizing any core component that the operator or vendor has access to the source code for, as well as for detecting bugs or irregular performance issues in specific parts of a core component.

Cost As explored already in Chapter 3, monitoring the cloud computing costs of a network deployment is of utmost importance for an operator. It will not only be very insightful in terms of detecting potential saving opportunities, but it is also crucial data for forming business strategies. For example, as mobile operators typically expand their business to serve multiple slices of their 5G network for various uses, it would be useful to have data on the cost efficiency of each network slice. Furthermore, it can be combined with customer data to identify customers that incur higher cost (through more traffic or specialized network use), allowing operators to optimize their network share accordingly. The cost data is also useful in choosing between vendors for a particular core component, as operators can

pick their choice according to their priorities.

There are two challenges to collecting the kinds of metrics defined above. First, cost aside, many of the metrics must be collected with insight into the internal data of a mobile core component. Even network-level metrics such as success rate or latency must be measured internally, as the NGAP packets are encrypted and any external traffic analysis can not measure it. This means that core components must be instrumented extensively, which may leave an impact on the core's performance. Similar studies in the past have shown promising results with the data plane (Massimo Girondi [2020]) but none exist with the control plane despite its importance. The second challenge arises as a result of the auto-scaling nature of cloud-native components when paired with technologies like Kubernetes. The replicated containers are ephemeral and can disappear whenever the network load decreases, they cannot store metrics internally over a long period of time. Further, some statistics must be aggregated across all replicas, such as the number of registered subscribers.

For regular stateful components, Prometheus (Prometheus Authors [2024]) is an existing tool that works fantastically for improving the observability of a system. Prometheus reduces the impact on performance on the instrumented component using a pull-based approach to metrics collection. This means that the component has to be fitted with an HTTP server that exposes metrics and their values in an certain format, which Prometheus scrapes at regular intervals. In this way, the reliability of the component is not bound to the reliability of Prometheus. However, the pull-based approach does not work well with auto-scaling and ephemeral containers, as data may get lost if it is not scraped in a timely manner before a replicated instance is terminated. Prometheus has an alternative push-based metrics collection system built in for components to send metrics to at will, however it is discouraged due to the aforementioned bottleneck concerns that tie the availability of the Prometheus server with the instrumented component.

Yagra aims to achieve this highly internal and app-specific instrumentation that gives insightful data, all while not sacrificing the performance when reporting from multiple scalable core components.

4.2 Architecture

The novelty of Yagra is in its separation of the collection and processing of metrics. The collection of metrics is performed within the critical path of the instrumented component as necessary, but it uses a persistent external process to aggregate the metrics and perform required processing on it before exposing it to Prometheus as a pull source. This separation of concerns allows for metric collection with negligible performance impact while allowing deep application insight. The overall architecture is illustrated in Figure 4.1. We refer to the metric collection system as the Yagra Library as it is provided in the form of a C library, and the external process as the Yagra metrics bus.

The Yagra C library is capable of measuring time between sections of the critical path of a core component, such as for measuring database access times or NGAP packet decode times. These are collected and sent in a 'batch', which is coupled with the unit

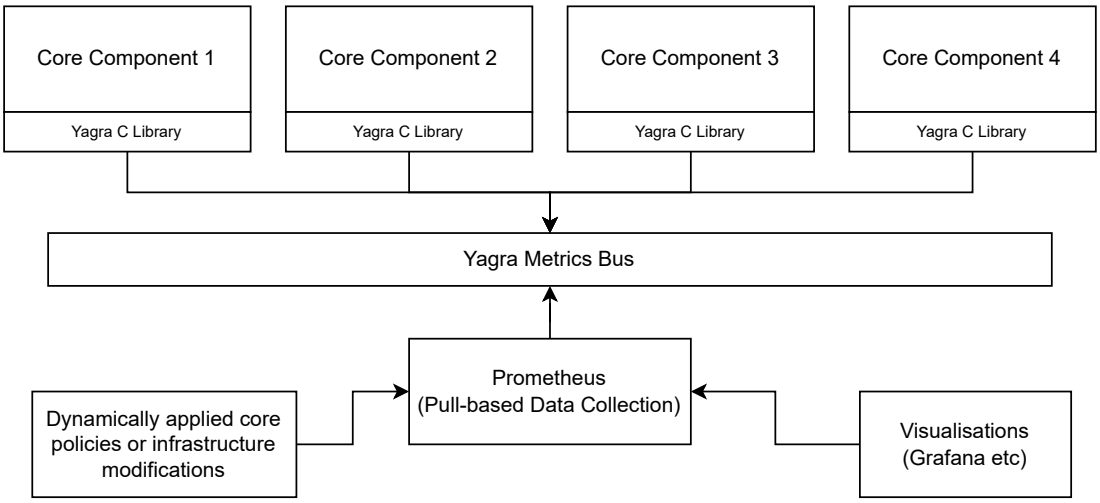


Figure 4.1: The architecture for the Yagra metrics collection system

of processing in the component. For CoreKube workers, a batch refers to the metrics collected over the processing of a single message packet, however this is arbitrary and can differ based on the functionality and implementation of the core component. Yagra allows the library integrator to define identifiers for each batch, such as the UE ID or the message type. These get sent alongside the metrics to the bus, where it is used for both labeling the Prometheus metrics it exposes, as well as to calculate aggregation metrics (e.g., number of unique label values observed). Importantly, the Yagra C library allows small amounts of local processing to handle duplicate metrics collected during a single batch, before sending them to the Yagra bus. Specifically, it can choose to keep the average, sum, minimum, maximum, first-seen or last-seen value for a metric. In practice, this is extremely useful for distinguishing the processing of counter-based metrics (e.g., number of database accesses, which should be summed if there are multiple) versus measurement-based metrics (e.g., database access time, which should have the average or the maximum recorded). The temporary local processing and batch-collection allows the library to reduce the number of network calls to the Yagra bus, reducing the impact on performance. Furthermore, the sending of metrics can be done in a separate thread, after the collection of all metrics within a batch. When Yagra is integrated in this way, the performance impact on the critical path is minimal.

The library is further capable of storing small amounts of persistent data as well. While this stateful nature may seem counter-intuitive at first when considering that Yagra's goal is to support cloud-native core components that can be stateless and ephemeral, it is not a large problem as the persistent metrics are sent along every batch of non-persistent metrics, making sure that the data arrives at the metrics bus frequently and without loss when an instance is inevitably terminated at some point.

Communication between the Yagra C library and the bus is one-way over TCP, using a specialised byte encoding of the metric and label names to save both memory and network overhead. The bus, upon receiving observed values for metrics, calculates additional aggregation metrics based on it before making them available for collection from Prometheus. Finally, Prometheus then acts as a short-term data store to enable

further applications such as user-friendly visualisations with Grafana dashboards, or dynamically applied network policies.

4.2.1 Comparisons to Other Solutions

The architecture presents benefits over other solutions for achieving application-level observability. A related work for cloud-native software in general is Telegraf (InfluxData Inc. [2024]), an open-source software made for a similar goal in mind: collect infrastructure performance metrics from distributed application containers. Unfortunately it is orthogonal with our solution, as Telegraf focuses on the centralisation of Prometheus-compatible exported metrics collected from different software – not on the method of software instrumentation itself. Telegraf is therefore unable to instrument core components and define new metrics, such as for business-specific KPIs.

4.3 Implementation

The library for instrumenting the core components is written in C, as a pair of source and header files that makes it easy to integrate to any existing component where the source code is available. It is written with a focus on developer usability, exemplified by the extensive logging, function documentation, and a clear API. Internally, it contains a TCP client that maintains a socket connection to the metrics server from the moment `yagra_init()` is called, and sends metrics at any developed-defined point using `yagra_send_batch()`.

To use this library, one should define upfront the persistent metrics, the metrics to be collected for each batch, and the labels to apply to the batch. These are done with the `yagra_define_*` functions, which can take as arguments the name and description, the type of metric, and the local aggregation strategy in case of duplicates. Internally, this function sends a packet to the Yagra metrics bus with the values passed in as arguments to request a registration of a new Prometheus metric to expose. Then, at any point a batch begins (such as when a new message is received from another component in the core), the developer can use the `yagra_init_batch()` function to start a batch that metrics get appended to, and and observe values using the `yagra_observe()` function. Note here that the Yagra library has been engineered to intelligently add the observed value regardless of whether it should be stored as a persistent metric, batch metric, or a batch label. This results in a very developer-friendly API interface.

The central metrics bus is written in C++ with speed and resource usage in mind. Speed is important to handle packets from many different components in the core without becoming a bottleneck itself, and resource usage is optimised so that the adoption of Yagra as a metrics collection system does not itself incur much cost when deployed to a cloud environment. Specifically, when the bus receives a message from the Yagra C library, it launches a separate thread to enable multiprocessing of incoming messages. Internally, the bus runs two TCP servers: one for collecting metrics from various core components, and one for exposing the metrics for pull-based scraping services like Prometheus.

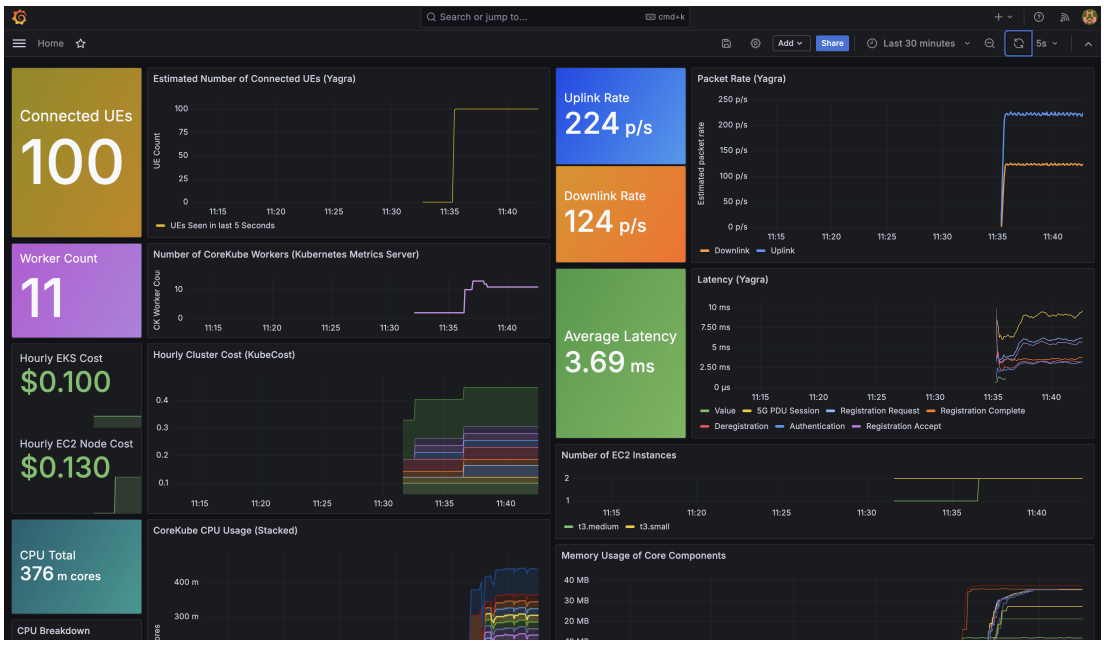


Figure 4.2: An example Grafana dashboard visualizing metrics collected from Yagra and Kubernetes Metrics Server, to show both core-specific and generic metrics

Care has been taken to reduce the footprint throughout the Yagra system as metrics are intended to be collected and sent frequently. For example, after the initial registration of a metric, the original name is no longer stored to save memory. Instead, the C libary uses a fast string hash function (Daniel J. Bernstein [1991]) is used to bring down the metric name to a `uint64_t` (16 bytes) integer hash. This hash is then communicated to the bus where it is used as a lookup key into a map of various exposed metrics. Compared to a string-based approach, this reduces the metrics-reporting network packets from around 2000 bytes on average to just over 150 bytes, freeing up both memory and network capacity.

To demonstrate the use of metrics collected by Yagra, we used Grafana (Grafana Labs [2024a]) to visualize the average message latency for CoreKube workers. A screenshot of the dashboard is shown in Figure 4.2.

4.4 Evaluation under simulated situations

To evaluate Yagra, we put it under test in various circumstances that may arise within a public cloud deployment of a mobile core. Through these tests, we evaluated whether the collected metrics accurately represent the test scenario, and determined the practicality and accuracy of the system for mobile operators to operate their network with efficiency.

We used a deployment of CoreKube patched with Yagra as the observability solution, together with the malicious patches described below. The patches were choesn to be representative example of defects appearing in a cloud-native mobile core. We deployed CoreKube as demonstrated in Chapter 3, within the `eu-north-1` region and with EC2 as the compute backend.

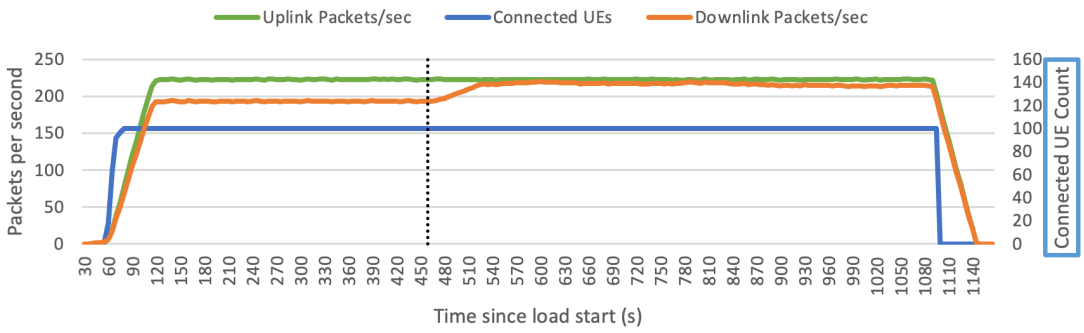


Figure 4.3: Network throughput and UE count for a regular 100 UE workload, until a core update at the highlighted timestamp causes an extra downlink packet to be sent.

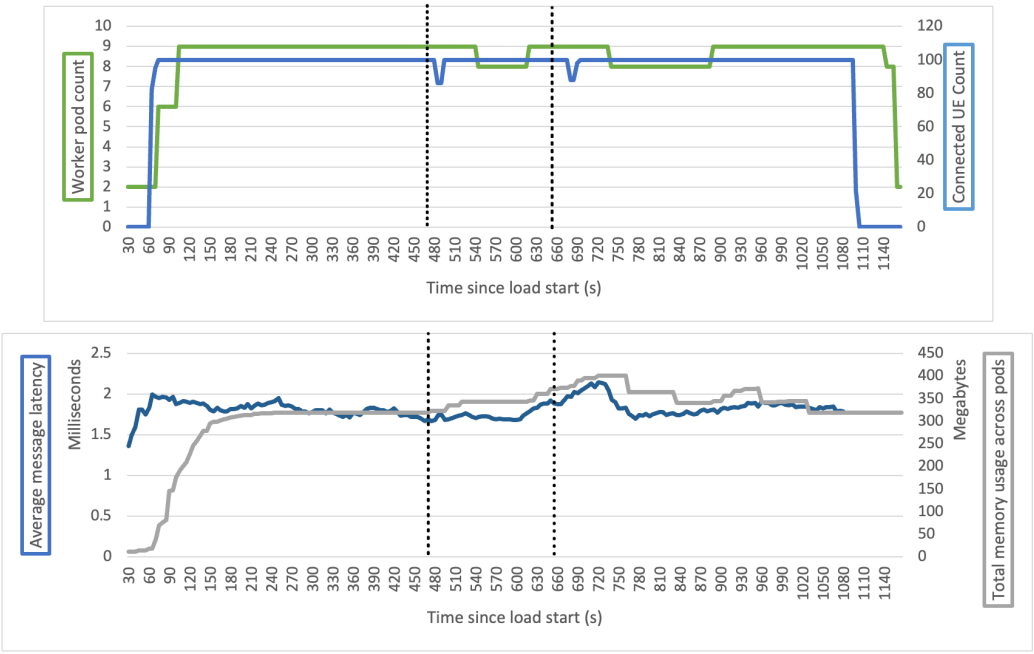
4.4.1 A Component Sending Malicious Packets

With the advent of containerized and multi-component cores, it is becoming possible and more likely for an operator to use a component of the mobile core supplied by multiple third-party vendors. This is naturally cost-efficient as operators can pick and choose the best of each vendor, but come with the possibility of supply-chain vulnerabilities and difficulty and performing audits. In this test scenario, we assume that an automated upgrade process in one of the core components introduced a malicious activity, where every control-plane packet sent by the core is also sent to an attacker-owned UE. This is a plausible scenario as described in CH-SC3 of the ENISA overview on 5G security (European Union Agency for Cybersecurity (ENISA) [2022]).

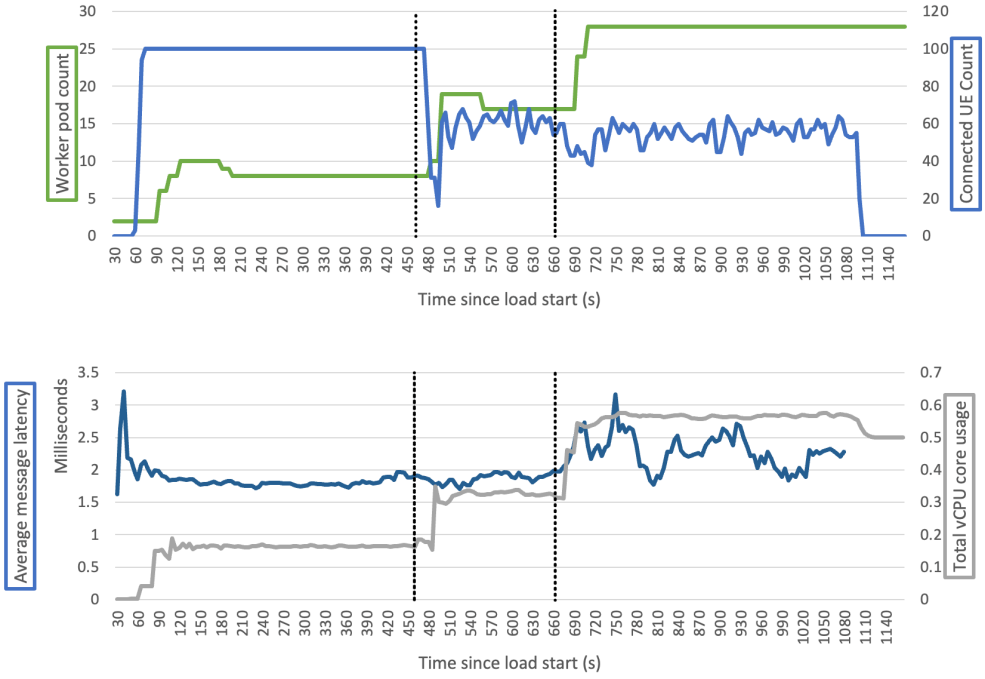
We implement this scenario by patching the CoreKube worker component to send an extra downlink packet to a set destination every time it is about to send a downlink packet. Figure 4.3 shows the change in metrics as collected from Yagra. At time 400, the worker was upgraded from the safe version to the malicious version, and immediately an increase in downlink packets was seen, verifying that Yagra is able to detect such situations quickly. In a production setup, these anomalies can be detected and alerted using tools like Grafana Alerts (Grafana Labs [2024b]).

4.4.2 A Badly Coded Update to a Core Component

An update to a core component may be engineered hastily, requiring more resources (vCPU and memory) than what is needed for an ideal implementation of the component. We simulate a badly coded vendor component by patching a CoreKube worker module to use more CPU cycles than necessary to handle packets, as well as to leak large amounts of memory. We demonstrate that this defect is revealed as it happens in Figure 4.4. The memory leak is detected as a rise in latency and total pod memory, as well as a brief drop in connected UE count. The CPU spike is also detected as a sudden higher total CPU usage, which results in pods scaling and UE connectivity tanking. These results verify that Yagra is able to give real-time observability information to a cloud-native core deployment, aiding both developers of cloud-native core functions, as well operators who use UE connectivity information for their business KPIs.



(a) Metrics collected as a large memory leak happens at the highlighted timestamps.



(b) Metrics collected as a CPU spike happens at the highlighted timestamps.

Figure 4.4: Top: Connected UE and worker pod count. Bottom: Average latency and allocated worker memory.

4.5 Future Work

4.5.1 Detecting Security Threats in the Core

Throughout this report, we have assumed that the public cloud would be suitably secure for running the control plane of a public cloud. However, this may not always be the case, as highlighted by the opinions raised by network operators when encouraged to use the public cloud (British Telecom [2022]). The main concern with public cloud infrastructure is its use of multi-tenant resources and virtualization layer concerns. The network operator has less control over the physical security of the servers, and must trust the cloud provider to keep the servers secure. This is a concern especially with the control plane, as it is the part of the network that can cause most sensitive data to be exposed (such as customer data).

To address this concern, a future work would be to implement a security monitoring system based on Yagra that can detect and alert the operator of malicious activities in the core. Already, Yagra is capable of detecting unusually high traffic or latency, however it is not yet capable of integrating with tracing or profiling tools. This makes Yagra a good observability solution for business and regular performance metrics, but not for raising concerns about unpredictable malicious behaviour that cannot be expressed as a metric. For example, unauthorized access into the customer database would currently go unnoticed. To support such events, Yagra must be rearchitectured to support not just numerical metrics, but also events and logs, exposing to not only Prometheus but to other ingestion systems like Loki (Grafana Labs [2024c]).

Chapter 5

Conclusions

In this paper, we have explored and demonstrated the feasibility of deploying a cloud-native mobile core network to public cloud infrastructure, from a holistic perspective encompassing ease of manageability, cost, and performance. To this end, we developed a command-line tool, `publicore`, to manage reproducible cloud core deployments, and used it to measure and compare performance across multiple deployment configurations on AWS. We concluded that a public cloud core deployment is feasible from a cost and performance perspective, but noticed the lack of observability which could hinder adoption of such deployment in mobile network operators. In order to fill this gap, we developed the Yagra system, a cloud-native, vendor-agnostic, and open-source observability solution for cloud-native cores, and demonstrated its benefits.

Through this research project, I hope to have demonstrated the culmination of my industrial and academic experience with telecommunications system design. With both commercial and government interest in the mobile core growing, I believe that my work and the knowledge I gained through this process will be useful to both the wider research community and to my career.

Over the coming months, I will be continuing my work under a research internship with my supervisor, completing the works identified in Sections 3.6 and 4.5. Through it, I am aiming to take the results of this paper towards a publishable output.

Bibliography

3rd Generation Partnership Project (3GPP). *Group Specification; Network Functions Virtualisation (NFV); Architectural Framework (3GPP GS NFV 002 version 1.2.1)*, 12 2014.

3rd Generation Partnership Project (3GPP). *LTE; Evolved Universal Terrestrial Radio Access (E-UTRA) and Evolved Universal Terrestrial Radio Access Network (E-UTRAN); Overall description; Stage 2 (3GPP TS 36.300 version 14.2.0 Release 14)* , 4 2017a. URL https://www.etsi.org/deliver/etsi_ts/136300_136399/136300/14.02.00_60/ts_136300v140200p.pdf.

3rd Generation Partnership Project (3GPP). *Technical Specification; Group Radio Access Network; NG-RAN; Architectuer description (3GPP TS 38.401 version 15.0.0 Release 15)* , 12 2017b.

3rd Generation Partnership Project (3GPP). *5G; NG-RAN; NG signalling transport (3GPP TS 38.412 version 15.0.0 Release 15)*, 7 2018. URL https://www.etsi.org/deliver/etsi_ts/138400_138499/138412/15.00.00_60/ts_138412v150000p.pdf.

3rd Generation Partnership Project (3GPP). *5G; Management and orchestration; 5G end to end Key Performance Indicators (3GPP TS 28.554 version 16.7.0 Release 16)* , 1 2021a. URL https://www.etsi.org/deliver/etsi_ts/128500_128599/128554/16.07.00_60/ts_128554v160700p.pdf.

3rd Generation Partnership Project (3GPP). *Procedures for the 5G System (3GPP TS 23.502 version 16.7.0 Release 16*, 1 2021b. URL https://www.etsi.org/deliver/etsi_ts/123500_123599/123502/16.07.00_60/ts_123502v160700p.pdf.

ACG Research, sponsored by Affirmed Networks and VMWare. *Total Cost of Ownership Study; Virtualizing the Mobile Core*, 7 2015. URL https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/solutions/vmware-nfv-tco-report-acg.pdf.

Amazon EKS User Guide. *Increase the amount of available IP addresses for your Amazon EC2 nodes*, 2024. URL https://docs.aws.amazon.com/eks/latest/userguide/cni-increase-ip-addresses.html.

Amazon Elastic Container Service Developer Documentation. *AWS Fargate security considerations*, 2024. URL https://docs.aws.amazon.com/AmazonECS/latest/bestpracticesguide/fargate-security-considerations.html.

Amazon Web Services. *EC2 On-Demand Pricing*, 3 2024a. URL https://aws.amazon.com/ec2/pricing/on-demand/.

Amazon Web Services. *EC2 Spot Instances Pricing*, 3 2024b. URL https://aws.amazon.com/ec2/spot/pricing/.

Amazon Web Services. *Serverless Compute Engine - AWS Fargate Pricing*, 2024a. URL https://aws.amazon.com/fargate/pricing/.

Amazon Web Services. *Amazon EC2 T3 Instances*, 2024b. URL https://aws.amazon.com/ec2/instance-types/t3/.

Amazon Web Services Labs GitHub. *eni-max-pods.txt*, 2024. URL https://github.com/awslabs/amazon-eks-ami/blob/master/files/eni-max-pods.txt.

Analysys Mason. *Perspective: Open RAN could deliver up to 30% TCO savings for operators with the right platform strategy and skill set*, 2 2022. URL https://www.analysysmason.com/contentassets/b3260036a0d449718117eeaf5ac83472/analysys_mason_open_ran_tco_feb2022_rma16_rma18.pdf.

Andrew Ferguson. *CoreKube Public Cloud Connectivity Tech Report*, 2 2024.

AT&T Business. *AT&T Network Functions Virtualization*, 2024. URL https://www.business.att.com/products/network-functions-virtualization.html.

Bartosz Taudul. *wolfpld/tracy: Frame profiler*, 2024. URL https://github.com/wolfpld/tracy.

Joe Breen, Andrew Buffmire, Jonathon Duerig, Kevin Dutt, Eric Eide, Mike Hibler, David Johnson, Sneha Kumar Kasera, Earl Lewis, Dustin Maas, Alex Orange, Neal Patwari, Daniel Reading, Robert Ricci, David Schurig, Leigh B. Stoller, Jacobus Van der Merwe, Kirk Webb, and Gary Wong. POWDER: Platform for open wireless data-driven experimental research. In *Proceedings of the 14th International Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization (WiNTECH)*, September 2020. doi: 10.1145/3411276.3412204.

British Telecom. *Cloud services market study: BT response to Ofcom's Call for inputs*, 11 2022. URL https://www.ofcom.org.uk/__data/assets/pdf_file/0031/248935/BT.pdf.

containerd Authors. *containerd: An industry-standard container runtime with an emphasis on simplicity, robustness and portability*, 2024. URL https://containerd.io/.

Daniel J. Bernstein. *djb2 Hash Function*, 1991. URL https://www.cse.yorku.ca/~oz/hash.html.

Docker Inc. *Docker*, 2024. URL https://www.docker.com/.

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, 9(3):743–756, 2023. ISSN 2352-8648. doi: https://doi.org/10.1016/j.dcan.2022.04.026. URL https://www.sciencedirect.com/science/article/pii/S2352864822000815.

DZone Limited. *Kubernetes in the Enterprise*, 11 2023.

European Union Agency for Cybersecurity (ENISA). *NFV Security in 5G - Challenges and Best Practices*, 2 2022.

Andrew E. Ferguson, Jon Larrea, and Mahesh K. Marina. Artifact for CoreKube: An Efficient, Autoscaling and Resilient Mobile Core System, August 2023. URL https://doi.org/10.5281/zenodo.8268932.

Ismael Gomez-Miguelez, Andres Garcia-Saavedra, Paul D. Sutton, Pablo Serrano, Cristina Cano, and Doug J. Leith. srslte: an open-source platform for lte evolution and experimentation. In *Proceedings of the Tenth ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization*, WiNTECH '16, page 25–32, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342520. doi: 10.1145/2980159.2980163. URL https://doi.org/10.1145/2980159.2980163.

Endri Goshi, Raffael Stahl, Hasanin Harkous, Mu He, Rastin Pries, and Wolfgang Kellerer. Pp5gs—an efficient procedure-based and stateless architecture for next-generation core networks. *IEEE Transactions on Network and Service Management*, 20(3):3318–3333, 2023. doi: 10.1109/TNSM.2022.3230206.

Grafana Labs. *Grafana*, 2024a. URL https://grafana.com/.

Grafana Labs. *Grafana — Alerting*, 2024b. URL https://grafana.com/docs/grafana/latest/alerting/.

Grafana Labs. *Grafana Loki*, 2024c. URL https://grafana.com/oss/loki/.

HM Treasury Department of Culture, Media and Sport. *Next Generation Mobile Technologies: A 5G Strategy for the UK*, 3 2017. URL https://assets.publishing.service.gov.uk/media/5a801327e5274a2e8ab4e09a/07.03.17_5G_strategy_-_for_publication.pdf.

Ian Smith. *[EKS] ['–service-node-port-range']: Configurable service-node-port-range · Issue #1361 · aws/containers-roadmap*, 2021. URL https://github.com/aws/containers-roadmap/issues/1361.

InfluxData Inc. *influxdata/telegraf: The plugin-driven server agent for collecting & reporting metrics.*, 2024. URL https://github.com/influxdata/telegraf.

InfoVista. *Ativa™ 5G Core Monitoring Solution*, 11 2022. URL https://www.infovista.com/sites/default/files/pdfs/ds_infovista_ativa_5g_core_monitoring.pdf.

Intel Corporation. *Intel® VTune™ Profiler*, 2024a. URL https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html.

Intel Corporation. *Intel® VTune™ Profiler User Guide*, 2024b.

International Telecommunications Union (ITU). *Minimum requirements related to technical performance for IMT-2020 radio interface(s)*, 11 2017. URL https://www.itu.int/dms_pub/itu-r/opb/rep/R-REP-M.2410-2017-PDF-E.pdf.

Kubernetes. *kubernetes/cluster-autoscaler: Autoscaling components for Kubernetes*, 2024. URL https://github.com/kubernetes/autoscaler.

Kubernetes Project Instrumentation Special Interest Group. *Kubernetes Metrics Server*, 2024. URL https://github.com/kubernetes-sigs/metrics-server.

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. ISBN 9781450383424. doi: 10.1145/3447993.3483248. URL https://doi.org/10.1145/3447993.3483248.

Jon Larrea, Andrew E. Ferguson, and Mahesh K. Marina. Corekube: An efficient, autoscaling and resilient mobile core system. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, ACM MobiCom '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399906. doi: 10.1145/3570361.3592522. URL https://doi.org/10.1145/3570361.3592522.

Massimo Girondi. Efficient traffic monitoring in 5G core network, 2020. URL https://kth.diva-portal.org/smash/get/diva2:1463793/FULLTEXT01.pdf.

Jiayi Meng, Jingqi Huang, Y. Charlie Hu, Yaron Koral, Xiaojun Lin, Muhammad Shahbaz, and Abhigyan Sharma. Modeling and generating control-plane traffic for cellular networks. In *Proceedings of the 2023 ACM on Internet Measurement Conference*, IMC '23, page 660–677, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703829. doi: 10.1145/3618257.3624808. URL https://doi.org/10.1145/3618257.3624808.

Iain Morris. The public cloud has failed to crack telecom, Jan 2024. URL https://www.lightreading.com/cloud/the-public-cloud-has-failed-to-crack-telecom.

Bram Naudts, Mario Kind, Sofie Verbrugge, Didier Colle, and Mario Pickavet. How can a mobile service provider reduce costs with software-defined networking? *Netw.*, 26(1):56–72, jan 2016. ISSN 0028-3045. doi: 10.1002/nem.1919. URL https://doi.org/10.1002/nem.1919.

NETSCOUT. *5G Performance - 5G Networ Monitoring for Service Providers*, 2024. URL https://www.netscout.com/solutions/5g.

Binh Nguyen, Tian Zhang, Bozidar Radunovic, Ryan Stutsman, Thomas Karagiannis, Jakub Kocur, and Jacobus 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*, MobiCom '18, page 163–178, New York, NY, USA, 2018. Association for Computing Machinery.

ISBN 9781450359030. doi: 10.1145/3241539.3241564. URL https://doi.org/10.1145/3241539.3241564.

O-RAN ALLIANCE e.V. *O-RAN.WG1.OAD-R003: O-RAN Archiecture Description (Version 11.00)*, 2 2024. URL https://www.o-ran.org/specifications.

OpenAirInterface.org. *OpenAirInterface 5G Radio Access Network Project*, 2024. URL https://openairinterface.org/oai-5g-ran-project/.

Edward J. Oughton and Zoraida Frias. The cost, coverage and rollout implications of 5g infrastructure in britain. *Telecommunications Policy*, 42(8):636–652, 2018. ISSN 0308-5961. doi: https://doi.org/10.1016/j.telpol.2017.07.009. URL https://www.sciencedirect.com/science/article/pii/S0308596117302781. The implications of 5G networks: Paving the way for mobile innovation?

Prometheus Authors. *Prometheus*, 2024. URL https://prometheus.io/.

Randall Stewart and Michael Tuexen. *sctplab/usrsctp: A portable SCTP userland stack*, 2015. URL https://github.com/sctplab/usrsctp.

Malla Reddy Sama, L.M. Contreras, John Kaippallimalil, Ippei Akiyoshi, Haiyang Qian, and Hui Ni. Software-defined control of the virtualized mobile packet core. *Communications Magazine, IEEE*, 53:107–115, 02 2015. doi: 10.1109/MCOM.2015.7045398.

Aisha Syed and Jacobus Van der Merwe. Proteus: a network service control platform for service evolution in a mobile software defined infrastructure. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, MobiCom '16, page 257–270, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342261. doi: 10.1145/2973750.2973757. URL https://doi.org/10.1145/2973750.2973757.

The Free5GC Authors. *free5gc/free5gc: Open source 5G core network base on 3GPP R15*, 2024. URL https://github.com/free5gc/free5gc.

The Kubernetes Authors. *Kubernetes: Production-Grade Container Orchestration*, 2024. URL https://kubernetes.io/.

The Open5GS Authors. *Open5GS*, 2024. URL https://open5gs.org/.

UK Department for Science, Innovation & Technology. *Open RAN principles*, 4 2022. URL https://www.gov.uk/government/publications/uk-open-ran-principles/open-ran-principles.

Hiroki Watanabe, Kunio Akashi, Keiichi Shima, Yuji Sekiya, and Katsuhiro Horiba. A design of stateless 5g core network with procedural processing. In *2023 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom)*, pages 199–204, 2023. doi: 10.1109/BlackSeaCom58138.2023.10299772.