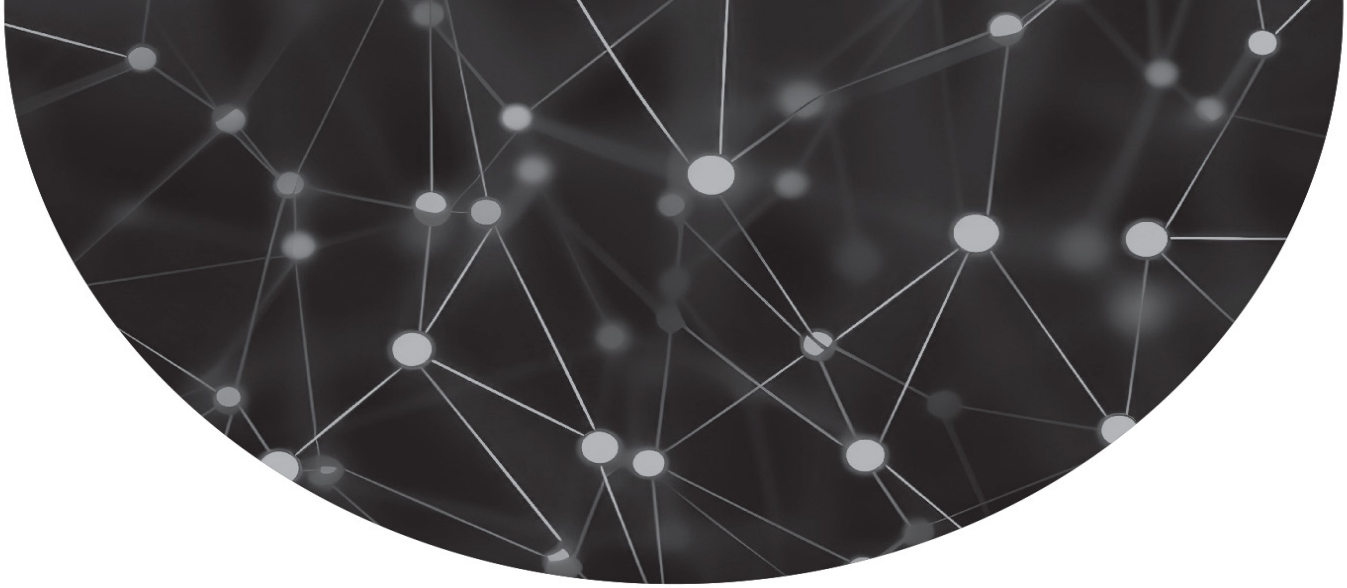


Sandeep Sharma and Nick Chase

INTRODUCTION TO NEPHIO

A new approach to *network automation*

INTRODUCTION TO NEPHIO

INTRODUCTION TO NEPHIO

By Sandeep Sharma and Nick Chase

Aarna University Press

ABOUT THIS BOOK

In April of 2022, Google donated Nephio to the Linux Foundation to form a new open source project bringing cloud, telecom and network functions providers together to enable Kubernetes-based cloud native intent automation for cloud and edge infrastructure with network services and applications running on that infrastructure. Nephio represents a new approach to network automation by leveraging the utility and ubiquity of Kubernetes to create a uniform automation control plane across the entire networking stack. More than 70 network operators and vendors around the world have joined the project and the community celebrated its first release, R1, in August of 2023.

Nephio delivers carrier-grade, simple, open, Kubernetes-based cloud native intent automation and common automation templates that materially simplifies the deployment and management of multi-vendor cloud infrastructure, network functions, and applications across large scale edge deployments. It allows users to express high-level intent, and provides intelligent, declarative automation that enables cloud and edge infrastruc ture, renders initial configurations for the network functions, and delivers those configurations to Kubernetes clusters.

This book provides a high-level understanding and business perspective of Nephio project and positioning for the ecosystem. It also provides a guide for new users on navigating, participating, and benefiting from the Nephio community.

Learning Outcomes

By reading this book, you will develop a solid understanding of the relevance of Nephio to your organization and your professional role. You will be able to navigate and contribute to the Nephio community and start exploring Nephio use cases and POCs relevant to your line of business.

By the end of this book, you should be able to:

- Understand Nephio's scope, goals, and benefits
- Explain why Nephio represents a new approach to automation
- Navigate the Nephio website, wiki, channels, documentation, and repositories
- Participate in the Technical community and SIGs

Target Audience

This book is designed for:

- **Nephio community members** who want to understand what the project does and how
- **Product Managers** building products that may make use of Nephio
- **Business leaders** who need to know whether Nephio is appropriate for their needs
- **Vendors** who want to build products on Nephio

Knowledge/Skills Prerequisites

In order to get the most out of this book, you should have a:

- Basic knowledge of Kubernetes
- Basic understanding of networking

How this book is organized

Besides this introduction, this book consists of five chapters:

- Ch 1. Edge-to-Cloud Orchestration
- Ch 2. Nephio Scope and Key Concepts
- Ch 3. Kubernetes as Uniform Automation Control Plane
- Ch 5. Nephio Internals

- Ch 5. Nephio Community, SIGs, and resources

It should take you 2-4 hours to complete this book.

CHAPTER 1
EDGE-TO-CLOUD ORCHESTRATION

Learning Objectives

By the end of this chapter, you should be able to:

- Understand the evolution from centralized mainframe systems to cloud and edge computing, and the role of cloud computing in this transformation.
- Recognize the challenges and benefits of orchestrating resources and data management across edge and cloud platforms.
- Explore the impact of edge-to-cloud orchestration in various industries, including healthcare, manufacturing, and retail.
- Grasp the strategic considerations and importance of adaptability in IT strategies within the hybrid edge-cloud environment

Nephio is an open source project aimed at simplifying and automating the deployment and lifecycle management of cloud native network functions (CNFs) and underlying infrastructure across edge and core cloud environments. It leverages Kubernetes to provide a uniform, scalable, and efficient way to manage network functions. Nephio's goal is to bring the agility, flexibility, and efficiency of cloud native technologies to network operations, enabling telecom operators and enterprises to more easily adopt and manage cloud native network solutions. This approach is intended to streamline operations, reduce costs, and accelerate the deployment of new services in the telecom sector and beyond.

Let’s start by looking at edge-to-cloud orchestration.

This chapter examines the evolution and integration of edge-to-cloud computing, highlighting the role of Nephio in orchestrating resources across these platforms. It looks at the journey from centralized mainframes to

cloud and edge computing, underscoring the importance of cloud computing in data processing and analytics, its contribution to virtualization, and the implications of the Internet of Things (IoT).

The chapter also addresses the challenges and benefits of managing data across edge and cloud, including security concerns, integration with legacy systems, and the impact of this orchestration in various industries. We'll also explore the concept of a new hybrid environment combining edge and cloud computing, emphasizing strategic considerations for businesses and the necessity of adaptability in IT strategies.

1.1. Introduction to Edge-to-Cloud Orchestration

Edge-to-Cloud Orchestration balances the localized processing power of edge computing with the extensive resources of cloud infrastructure. This section helps you to navigate the complexities and synergies of this integration, highlighting how Edge-to-Cloud Orchestration not only bridges two distinct computing paradigms but also fosters a balanced, efficient, and resilient digital ecosystem.

As we delve deeper, we will explore the nuanced interplay between the edge's proximity to data sources and the cloud's vast computational capacity, shedding light on how this balanced approach is instrumental in meeting the diverse, dynamic demands of contemporary data processing and application management.

Let's start with an overview.

Edge-to-Cloud Orchestration: Overview

Nephio, with its focus on Kubernetes-based automation, is ideally suited for orchestrating resources across edge and cloud platforms. This orchestration ensures seamless integration, efficient resource utilization, and consistent deployment of services, which is crucial for edge computing scenarios where latency, bandwidth, and local processing are key factors. Nephio's approach aligns perfectly with the demands of edge-to-cloud orchestration, offering

scalable and flexible solutions.

Edge-to-cloud orchestration represents a holistic approach to managing and optimizing data flow and processing across the continuum of edge devices and centralized cloud infrastructure. It's a strategy that leverages the immediacy and localized processing capabilities of edge computing alongside the vast computational power and storage of the cloud. By orchestrating resources and operations across this spectrum, organizations can achieve reduced latency, enhanced data processing speeds, and more efficient resource utilization, ensuring that data is processed and acted upon at the optimal location, whether it's at the edge, near the source of generation, or, for more intensive tasks, in the cloud.

But how did we get here?

Edge and Cloud Computing: The Journey

Over the past few decades, the technological landscape has witnessed a transformative journey from centralized mainframe systems to the distributed power of cloud computing, and more recently, to the emergence of edge computing. Cloud computing revolutionized the way businesses operate, offering scalable resources, vast storage, and global accessibility. However, as the digital universe expanded with the proliferation of Internet of Things (IoT) devices and the need for real-time processing, edge computing emerged as a complementary paradigm.

Positioned closer to data sources, edge computing addresses latency concerns by processing data on-site or at least in proximity. Together, edge and cloud computing have evolved in tandem, each addressing unique challenges and, when combined, offering a comprehensive solution for the modern data-driven world, as well as unique challenges.

1.2 The Current Landscape

Cloud computing is undeniably central in data processing and analytics, acting as the backbone for modern digital solutions. This section navigates the multifaceted terrain of cloud computing, illustrating its pivotal role in transforming data into actionable insights.

As we explore this domain, we'll understand how cloud computing not only serves as a repository of vast computational power and storage, but also as a catalyst for innovation and efficiency in data analytics.

By delving into the nuances of cloud-based data processing, we will uncover the intricate ways in which cloud computing empowers organizations to harness the potential of their data, driving informed decision-making and fostering a culture of agility and foresight in an ever-evolving digital landscape.

The Role of Cloud Computing in Data Processing and Analytics

Cloud computing has fundamentally transformed the realm of data processing and analytics by providing scalable, on-demand computational resources and advanced analytical tools. With its vast storage capacities and distributed processing capabilities, the cloud enables organizations to handle large datasets and complex tasks with unparalleled efficiency.

Moreover, cloud platforms offer a suite of analytical services, from big data to machine learning to data visualization, empowering businesses to derive actionable insights from their data. By centralizing data storage and processing in the cloud, organizations can achieve cost savings, enhance collaboration, and accelerate innovation, making cloud computing an indispensable pillar in the modern data-driven landscape.

Cloud Computing Enables the Virtualization of Physical and Network Services

Cloud computing has been a catalyst for virtualization. It provides scalable, on-demand resources that enable the creation of virtual environments. This virtualization offers numerous benefits:

- It maximizes hardware utilization
- It reduces physical infrastructure costs
- It enhances flexibility in managing computing resources

By allowing multiple virtual machines to run on a single physical server, cloud computing facilitates efficient resource allocation, rapid scalability, and improved disaster recovery processes, which leads to significant operational efficiencies and cost savings.

Cloud computing also revolutionized the telecom industry by enabling the virtualization of specialized telecom equipment into Virtual Network Functions (VNFs), which eventually evolved into Cloud Native Functions (CNFs).

This transformation has been facilitated by the cloud's powerful computing and storage capabilities, allowing traditional physical network functions to be implemented as software applications on virtual machines or containers. Telecom operators can now rapidly deploy, scale, and update network services without the need for physical hardware changes, significantly reducing operational expenses and time-to-market for new services. This virtualization supports more dynamic and responsive network configurations, catering to the evolving demands of modern communications and enterprise.

The Rise of IoT and the Data Explosion at the Edge

All of this flexibility led to the ability to run small devices, or “things”, virtually anywhere. As long as the devices could connect to the internet, they could become part of this vast “cloud” of resources.

The advent of the Internet of Things (IoT) has ushered in an era where billions of interconnected devices, from smart thermostats to industrial sensors, continuously generate and transmit data.

This massive influx of data, often in real-time, has predominantly converged at the edge of networks, creating a veritable explosion of information. As these edge devices become increasingly embedded in our daily lives and industrial processes, they not only enhance our interconnectedness but also amplify the volume, velocity, and variety of data.

This surge demands innovative solutions for efficient processing, storage, and analysis, underscoring the critical role of edge computing in managing

and harnessing this tidal wave of information.

Challenges in Managing Data and Resources across Edge and Cloud

Bridging the gap between edge devices and centralized cloud infrastructure presents a myriad of challenges. As data flows across this continuum, ensuring its consistency, security, and timely processing can become complex.

Edge devices demand lightweight, efficient solutions due to their diverse architectures and resource constraints,while the cloud requires robust data synchronization and management strategies.

Additionally, the decentralized nature of edge computing can complicate security protocols, making data more vulnerable to breaches. Balancing the immediacy of edge processing with the computational power of the cloud, all while maintaining data integrity, security, and optimal resource allocation, remains a formidable challenge for organizations navigating this integrated landscape.

There are, however, benefits to making it work.

Benefits of Edge-to-Cloud Orchestration

Let's explore some of the benefits of edge-to-cloud orchestration.

REDUCED LATENCY AND FASTER DATA PROCESSING

One of the most pronounced benefits of edge-to-cloud computing is the dramatic reduction in latency and the acceleration of data processing.

By situating computation closer to the source of data generation, edge computing addresses the inherent delay that arises when data must traverse long distances to centralized cloud data centers. This proximity ensures real-time or near-real-time processing, which is crucial for applications like autonomous vehicles, smart city infrastructures, and telemedicine, where even a fraction of a second can make a significant difference.

When combined with the cloud's vast computational resources, this

architecture delivers both speed at the edge and in-depth analysis in the cloud, ensuring timely responses and informed decision-making.

BOLSTERING SECURITY AND PRIVACY IN EDGE-TO-CLOUD COMPUTING

Edge-to-cloud computing introduces a unique advantage in bolstering security and enhancing data privacy. By processing sensitive data locally at the edge, there's a reduced need to transmit it across potentially vulnerable networks to centralized cloud servers, thereby minimizing exposure to interception or breaches.

Furthermore, edge devices can be equipped with advanced encryption and security protocols, ensuring data is safeguarded at its source. On the cloud front, sophisticated security measures, including multi-factor authentication, advanced firewalls, and regular audits, further fortify data protection. This dual-layered approach, combining the localized security of edge computing with the robust defenses of cloud infrastructure, offers a comprehensive shield against threats, ensuring both data integrity and the privacy of individuals and organizations.

MASTERING SCALABILITY AND FLEXIBILITY WITH EDGE-TO-CLOUD COMPUTING

Edge-to-cloud computing stands as a beacon of scalability and flexibility in today's dynamic technological landscape. The edge offers the ability to deploy and scale applications based on localized demands, ensuring immediate responsiveness without overburdening the central infrastructure.

Conversely, the cloud provides virtually limitless computational and storage resources, allowing organizations to expand seamlessly as their needs evolve. This synergy means that as data influx grows or processing requirements change, resources can be dynamically allocated or reallocated between the edge and the cloud.

Such fluidity in resource management not only ensures optimal performance and cost-efficiency but also empowers organizations to adapt swiftly to changing market conditions, user demands, or technological advancements.

ELEVATING USER EXPERIENCES THROUGH LOCALIZED EDGE PROCESSING

In the edge-to-cloud computing paradigm, localized processing at the edge plays a pivotal role in enhancing user experiences. As we said earlier, by processing data closer to its source, immediate feedback and real-time interactions become achievable.

This immediacy eliminates the lag that users might experience if data had to be sent to a distant cloud server for processing. Whether it's a video streaming platform adjusting quality based on local bandwidth, a smart home system responding instantly to user commands, or a wearable health device providing real-time health metrics, the localized processing at the edge ensures that user interactions are smooth, responsive, and tailored to their immediate context, leading to more satisfied and engaged users.

Challenges in Edge-to-Cloud Orchestration

Of course, it’s not simple. Let’s consider some of the challenges in edge-to-cloud orchestration.

THE CHALLENGE OF DATA SYNCHRONIZATION AND CONSISTENCY

In the intricate dance of edge-to-cloud orchestration, ensuring data synchronization and maintaining consistency across distributed systems emerge as formidable challenges. With data being generated and processed at multiple edge locations and then potentially aggregated or analyzed in the cloud, there's an inherent risk of data discrepancies, conflicts, or even loss. Time-sensitive applications might face issues if the data at the edge is not in sync with the cloud, leading to decisions based on outdated or incomplete information.

Furthermore, as devices at the edge operate in diverse environments with varying connectivity, ensuring that all data changes are consistently reflected across the entire system, without duplication or omission, becomes a complex task, demanding robust synchronization protocols and meticulous data management strategies.

COMPLEXITY OF HANDLING MULTIFACETED EDGE DEVICES

One of the most intricate challenges in edge-to-cloud orchestration lies in

managing the wide array of diverse edge devices and platforms. These devices, ranging from smart home appliances and wearable tech to industrial sensors and autonomous vehicles, often come with varying architectures, operating systems, and communication protocols.

Integrating and orchestrating such a heterogeneous environment demands a deep understanding of each device's nuances and the ability to create unified interfaces for seamless communication, and ensuring consistent software updates, security patches, and performance optimization across this diverse ecosystem is also a logistical challenge. As the number and variety of edge devices continue to grow, developing standardized approaches and tools to manage this diversity becomes paramount to the success of edge-to-cloud orchestration.

NAVIGATING SECURITY CONCERNS IN A DISTRIBUTED EDGE-TO-CLOUD LANDSCAPE

In edge-to-cloud orchestration, safeguarding data and operations across a distributed network presents a significant challenge because the decentralized nature of edge devices, each operating in varied environments, introduces multiple potential points of vulnerability.

These devices, often outside the fortified walls of traditional data centers, can become targets for cyberattacks, unauthorized access, or malware infiltration. As data traverses from the edge to the cloud, ensuring its integrity, confidentiality, and authenticity becomes paramount.

Implementing robust encryption, consistent security protocols, and real-time threat detection across such a dispersed network demands a holistic security strategy—one that can adapt to the unique challenges of both the edge's proximity and the cloud's vastness.

TACKLING HURDLES TO INTEGRATION WITH LEGACY SYSTEMS

In addressing the challenges of integrating legacy systems within the edge-to-cloud computing paradigm, it is crucial to recognize the inherent complexities of these older systems. Originally developed in a pre-edge era, they often lack the necessary agility and compatibility for seamless integration with modern, cloud native architectures.

These systems typically feature rigid architectures, use proprietary protocols, and run on outdated software stacks, which can significantly hinder the efficient exchange of data and operational processes between edge devices and cloud platforms. Moreover, such legacy systems may pose security risks and exhibit performance constraints that are at odds with the demands of contemporary edge-to-cloud orchestration.

To effectively overcome these hurdles, a strategic approach is necessary–one that involves a thoughtful blend of modernizing key components, adapting existing functionalities, and implementing creative solutions to bridge the gap between old and new technologies.

This approach might include developing middleware or APIs that facilitate communication between legacy systems and modern platforms, or it may include incrementally replacing parts of the legacy system with more flexible, cloud-compatible alternatives.

The goal is to ensure these older systems can not only coexist but also actively contribute to the dynamic and rapidly evolv ing edge-to-cloud ecosystem. This approach helps in preserving critical legacy functionalities while unlocking new capabilities and efficiencies offered by edge-to-cloud computing.

1.3 Real-World Examples and Case Studies

Now let's try to bridge the gap between theoretical understanding and practical implementation, and get some insights into the tangible benefits and real-world applications of Edge-to-Cloud Orchestration in driving industry-specific advancements.

How Do Industries Leverage Edge-to-Cloud Orchestration?

Across diverse sectors, edge-to-cloud orchestration is driving transformative changes. In healthcare, real-time patient monitoring devices at the edge relay critical data to cloud-based systems, enabling timely interventions and personalized care. Manufacturing plants employ edge sensors to monitor

equipment health, with data analytics in the cloud optimizing production lines and predicting maintenance needs.

Meanwhile, the retail sector harnesses edge devices, such as smart shelves and point-of-sale systems, to gather shopper insights, while cloud analytics refine inventory management and enhance customer experiences. These industries, among others, showcase the tangible benefits of edge-to-cloud orchestration, where the fusion of localized data processing with expansive cloud analytics leads to more informed decisions, operational efficiencies, and innovative service offerings.

Successes and Setbacks in Edge-to-Cloud Implementations

As organizations embark on their edge-to-cloud orchestration journeys, multiple use cases emerge that represent the significant potential for this approach to transform industries. For example, a logistics company that once grappled with shipment delays leveraged edge devices to monitor real-time cargo conditions, while cloud analytics optimized routing, leading to timely deliveries and increased customer satisfaction.

But there are also challenges. For example, a smart city initiative, while achieving significant energy savings through edge-controlled street lighting, encountered unexpected security vulnerabilities, underscoring the need for robust cybersecurity measures.

These narratives, both triumphant and cautionary, offer invaluable insights, guiding future endeavors and highlighting the importance of adaptability, continuous learning, and proactive problem-solving in the ever-evolving landscape of edge-to-cloud orchestration.

1.4 The "New Reality" of Hybrid Environments

Now let’s talk about the "new reality" of hybrid environments, where the convergence of edge and cloud computing has become the new norm. This section delves into how this integration forms a foundational aspect of

modern digital strategies, reflecting a shift towards more adaptive, resilient, and scalable computing frameworks.

As we examine the dynamics of these hybrid environments, we'll highlight the seamless interplay between the localized processing capabilities of edge computing and the expansive, resource-rich nature of cloud platforms.

The point of this discussion is to illustrate the strategic advantages of this convergence, showcasing how businesses and technologies alike are evolving to embrace a more interconnected, efficient, and agile approach to managing data, appli cations, and services in the face of rapidly changing demands and opportunities in the digital landscape.

The Convergence of Edge and Cloud as the New Norm

Today, the fusion of edge and cloud computing is rapidly crystallizing as the new standard, forging a hybrid environment that harnesses the strengths of both realms. This convergence is not a mere technological trend but a strategic response to the burgeoning demands of real-time data processing, decentralized operations, and expansive analytics. As devices at the edge generate torrents of data, the cloud stands ready to analyze, store, and derive insights, creating a symbiotic relationship where each complements the other's capabilities.

This integrated approach is reshaping industries, driving innovation, and setting the foundation for the next wave of digital transformation. Organizations that recognize and adapt to this "new reality" position themselves at the forefront, ready to leverage the myriad opportunities presented by the harmonious union of edge and cloud.

Strategic Considerations for Businesses in this Hybrid Era

As the hybrid confluence of edge and cloud computing becomes the norm, organizations must evaluate where data processing is most efficient, be it at the edge for real-time responsiveness or in the cloud for in-depth analytics. Balancing security concerns, from the vulnerabilities of distributed edge

devices to the robust fortifications of cloud infrastructure, becomes paramount.

Businesses must also assess the cost implications, scalability challenges, and integration complexities with existing systems. This hybrid era demands a holistic vision, where strategic decisions are underpinned by a deep understanding of technological capabilities, market dynamics, and evolving customer needs—ensuring that businesses remain agile, competitive, and poised for growth in the face of rapid technological shifts.

The Importance of Adaptability and Forward-Thinking in IT Strategies

The rapid pace of technological evolution, coupled with the multifaceted challenges of integrating edge devices with cloud infrastructure, necessitates a nimble approach to IT strategies, where IT frameworks can pivot in response to emerging trends, unforeseen challenges, or shifting business objectives.

Beyond mere reactivity, a visionary perspective is crucial, allowing organizations to anticipate the next wave of innovations, from AI-driven edge analytics to quantum computing breakthroughs in the cloud. Embracing both adaptability and foresight ensures that businesses not only navigate the present complexities of the hybrid era but also position themselves at the vanguard, ready to harness the opportunities of tomorrow's digital frontier.

It’s in this space that Nephio has evolved, with the purpose of enabling this coordination over a hybrid edge-to-cloud environment. In the following chapters, we will get a basic feel for what it is and how it works.

CHAPTER 2
NEPHIO SCOPE AND KEY CONCEPTS

Learning Objectives

By the end of this chapter, you should be able to:

- Describe the role and functionalities of Nephio in cloud native network functions and its reliance on Kubernetes.
- Explain the concept of declarative structures and *Configuration as Data* in modern computing and their significance in Nephio's operations.
- Recognize the advantages of a declarative mindset, including simplicity, repeatability, and scalability, in system configuration and management.
- Explore the application of declarative structures in various contexts such as infrastructure as code, CI/CD pipelines, and automated testing, and their impact on workload management.

2.1 Introduction to Declarative Structures and Configuration as Data

In this section, we introduce the concepts of declarative structures and Configuration as Data, pivotal elements at the heart of Nephio's architecture. This discussion sets the stage for understanding how Nephio leverages these principles to streamline and automate the management of cloud-native network functions and infrastructure. By defining Nephio within this context, we want to show the framework's innovative approach to simplifying complex network orchestration tasks.

Through the lens of declarative structures and the treatment of configuration as data, we explore how Nephio empowers users to specify

their desired state of the system in a high-level, abstract manner, leaving the system itself to determine the necessary steps to achieve that state. This section not only defines Nephio's core philosophy and operational paradigm but also highlights its role in advancing the automation and efficiency of network and service management in cloud and edge computing environments.

Nephio Definition

In the introduction for this book, we provided the following definition of Nephio:

> Nephio is an open source project aimed at simplifying and automating the deployment and lifecycle management of cloud native network functions (CNFs) and underlying infrastructure across edge and core cloud environments. It leverages Kubernetes to provide a uniform, scalable, and efficient way to manage network functions. Nephio's goal is to bring the agility, flexibility, and efficiency of cloud native technologies to network operations, enabling telecom operators and enterprises to more easily adopt and manage cloud native network solutions. This approach is intended to streamline operations, reduce costs, and accelerate the deployment of new services in the telecom sector and beyond.

We will explore this definition in more detail throughout this chapter.

Declarative Structures in Modern Computing: Definition and Importance

When it comes to Nephio, everything flows from the idea of declarative high level structures and “Configuration as Data”.

Declarative structures represent a method of defining desired states and configurations in a high-level, human-readable format. This approach contrasts with imperative programming, focusing instead on the *what* rather than the *how* of configuration and deployment processes. In modern computing, especially in environments like Nephio, declarative structures are crucial. They enable the clear articulation of infrastructure and application requirements, simplifying automation and management. This

methodology not only streamlines development and operational tasks but also enhances scalability and adaptability in dynamic computing environments.

Treating Configuration as Data

The concept of treating configuration as data is a transformative approach in modern computing. It involves defining and managing configuration settings not as static, hard-coded elements, but as dynamic, structured data.

An earlier paradigm, Infrastructure as Code, made it possible for configurations to be version-controlled, audited, and managed with the same rigor and flexibility as application code. But by treating configuration as data, it becomes possible to automate the deployment and management of infrastructure and applications more efficiently.

This method enables seamless environment replication, quick rollback to previous states, and easier scaling and updating of systems. It aligns perfectly with the principles of Infrastructure as Code (IaC), enhancing the predictability, transparency, and efficiency of IT operations.

This approach is especially beneficial in complex environments, where it ensures consistency and reliability across various deployment stages and workloads.

2.2 The Power of Declarative Approaches

Now let's talk about the power of declarative approaches, focusing on how declarative structures fundamentally differ from imperative approaches in managing computing environments. This exploration sheds light on the inherent advantages of specifying the 'what' rather than the 'how' in system configurations and management.

By contrasting declarative and imperative methodologies, you will see the efficiency, scalability, and simplicity that declarative approaches bring to the table, particularly in the context of Nephio's use in orchestrating complex network functions and infrastructure.

This section underscores the shift towards more intuitive, maintenance-friendly, and error-reducing practices in software and network management, highlighting how declarative paradigms enable a more abstracted, goal-oriented perspective that aligns with the evolving needs of modern, dynamic computing environments.

How Declarative Structures Differ from Imperative Approaches

Declarative structures and imperative approaches represent two fundamentally different methodologies in configuring and managing systems. In a declarative model, as prominently used in both Kubernetes and Nephio, the focus is on defining the desired end state of a system without specifying the steps to achieve it. You simply describe the system in a YAML file, and Nephio ensures that it is not just created, but also maintained. For example:

```
apiVersion: nephio.org/v1alpha1
kind: NetworkFunction
metadata:
name: example-network-function
spec:
# Define the network function parameters
parameters:
image: example-nf-image:v1.0
# Define the network requirements
network:
bandwidth: 1Gbps
latency: <10ms
# Define any storage requirements
storage:
size: 100Gi
type: SSD
# Additional configuration data
configData:
configMapName: example-nf-config
```

Note: These code snippets are examples only and can't be guaranteed to work

in all environments

In this case we're creating a hypothetical **`NetworkFunction`** that defines specific network and storage parameters, and provides a **`ConfigMap`** with additional data.

This method contrasts with the imperative approach, where specific commands and procedures are outlined to reach a particular state. For example, an imperative approach to the above example might be:

```
kubectl create NetworkFunction example-network-function \
--image=example-nf-image:v1.0 --network. bandwidth=1Gbps \
--network.latency=<10ms --storage.size=100Gi \
--storage.type=SSD \
--configdata.configMapName=example-nf-config
```

Note: These code snippets are examples only and can't be guaranteed to work in all environments

In both of these cases, Kubernetes will create the **`NetworkFunction`** object, and it will ensure that the "conditions" remain correct. For example, if our theoretical **`NetworkFunction`** has a **`Controller`** that has been built for it, it might automatically move the **`NetworkFunction`** or take other actions to ensure that the requirements are always met.

But there's one major difference here; there's no really reliable way to track what's been created so that you can recreate it later, or to make changes without everything getting out of sync. It's much more reliable to have a YAML file stored in your repo, and if something changes, you can change the file and apply those changes. (It also lends itself to GitOps methodologies, in which changes to a Git repo trigger changes in the system.)

Declarative structures provide a higher level of abstraction, making configurations more intuitive and less prone to errors. They allow for easier automation, scalability, and maintain ability, as the system itself figures out the necessary steps to achieve the desired state.

In contrast, imperative methods require detailed, step-by-step instructions, making them more complex and rigid. This fundamental difference makes declarative structures more suited for modern, dynamic environments

where flexibility and efficiency are paramount.

2.3 Automation Through Declarative Structures

In this section, we explore the theme of automation through declarative structures, focusing on the mechanisms by which declarative structures enable sophisticated automation capabilities.

This part of the discussion emphasizes the transformative impact of declarative approaches on the automation landscape, particularly in how these structures facilitate the self-managing and self-optimizing behaviors of computing systems.

By detailing the process and benefits of leveraging declarative structures for automation, we aim to provide insights into how these methodologies simplify the orchestration of complex systems, reduce manual intervention, and enhance consistency and reliability across deployments.

This section highlights Nephio's commitment to harnessing the power of declarative structures, showcasing how this approach streamlines the deployment and management of network functions and infrastructure, thereby enabling more agile, efficient, and error-resistant operations in cloud and edge computing environments.

How Declarative Structures Enable Automation

Declarative structures are pivotal in enabling automation in modern computing environments like Nephio. By defining the desired state of a system or application in a high-level, structured format, these structures allow for the automatic configuration and management of resources.

The system interprets these declarations and autonomously takes the necessary actions to achieve the specified state without manual intervention. This approach streamlines deployment and maintenance processes, as changes can be made simply by updating the declarative configuration. It also ensures consistency and reduces errors, as the system consistently applies the defined state across various environments. This automated, self-

managing capability is especially beneficial in dynamic, scalable infrastructures, where rapid and accurate responses to changes are crucial.

Use Cases: Infrastructure as Code, Continuous Integration/Continuous Deployment (CI/CD), and Automated Testing

In the realm of cloud native technologies, automation through declarative structures plays a pivotal role in streamlining processes across various domains, including infrastructure as code, continuous integration/continuous deployment (CI/CD), and automated testing.

INFRASTRUCTURE AS CODE (IAC):

Declarative structures enable the definition of infrastructure in code format, allowing for automated provisioning and management of resources. Use cases include setting up server configurations, network topologies, and storage systems. By treating infrastructure as code, organizations can ensure consistency, repeatability, and efficiency in infrastructure deployment and scaling.

CONTINUOUS INTEGRATION/CONTINUOUS DEPLOYMENT (CI/CD):

In CI/CD pipelines, declarative structures automate the build, test, and deployment processes of software. They enable the definition of pipeline stages, environment variables, and deployment strategies. This automation ensures that new code changes are automatically built, tested, and deployed, leading to faster and more reliable software releases.

AUTOMATED TESTING

Declarative structures can be used to automate various testing processes, including unit tests, integration tests, and system tests. By defining test environments and test cases declaratively, organizations can automatically execute tests against every code change, ensuring that software is consistently tested for quality and functionality. This approach reduces manual testing efforts and accelerates the feedback loop for developers.

But don't get the idea that declarative structures are only for infrastructure.

2.4 Applying Declarative Structures to Workloads

This section addresses the application of declarative structures to workloads, with a specific focus on automating simple microservices from deployment to scaling, showing the practical benefits of applying declarative methodologies to real-world computing tasks and demonstrating how these principles can streamline the lifecycle management of microservices.

By exploring the journey of microservices from their initial deployment through to their dynamic scaling, we highlight how declarative structures facilitate a more intuitive, efficient, and flexible approach to managing these lightweight, modular components of modern applications.

This section underscores the synergy between declarative paradigms and microservices architecture, showcasing how Nephio leverages this relationship to enhance automation, reduce complexity, and improve scalability in cloud-native environments, thereby enabling organizations to more effectively meet the demands of their digital transformation initiatives.

Automating Simple Microservices: From Deployment to Scaling

Declarative structures can be just as useful when it comes to managing workloads as they are to managing infrastructure.

For example, automating simple microservices using declarative structures streamlines the entire lifecycle, from deployment to scaling. By defining microservices in a high-level, structured format, systems like Nephio can automatically deploy, manage, and scale these services based on their specified configurations.

This approach eliminates manual intervention in deployment processes, ensuring consistency and reducing errors. Scaling becomes effortless and responsive to demand, as the system can dynamically adjust resources based on predefined rules or metrics. This automation not only accelerates the deployment and scaling processes but also enhances the reliability and efficiency of microservices, making them well-suited for agile, cloud native

environments.

A simple microservices application might look something like this:

```
apiVersion: v1
kind: Service
metadata:
name: nf-service
spec:
selector:
app: nf-app
ports:
- protocol: TCP
port: 80
targetPort: 80
type: ClusterIP
---
apiVersion: apps/v1
kind: Deployment
metadata:
name: nf-app-deployment
spec:
replicas: 3
selector:
matchLabels:
app: nf-app
template:
metadata:
labels:
app: nf-app
spec:
containers:
- name: nf-container
image: micro-nf:latest
ports:
- containerPort: 80
```

Note: These code snippets are examples only and can't be guaranteed to work in all environments

In this case you are defining a service to manage the network connections for this microservice, and then defining the De ployment object that will contain the actual pods for the microservice. Should you need to change the requirements–for example, if you decide that you need 12 instances instead of 3–having a central location in which to make those changes is certainly more efficient.

Complex Workloads: Orchestrating AI/ML Training Pipelines

Orchestrating AI/ML training pipelines as complex workloads through declarative structures significantly enhances their automation and efficiency. In this approach, the entire AI/ML pipeline – from data preprocessing, to model training, to validation – is defined in a structured, high-level configuration. This enables systems like Nephio to automatically manage the deployment and scaling of these pipelines.

The declarative method ensures that the complex dependencies and resource-intensive tasks inherent in AI/ML workflows are handled seamlessly, adapting dynamically to varying computational needs. This automation not only streamlines the orchestration of these sophisticated workloads but also optimizes resource utilization, leading to faster, more efficient training cycles and the ability to rapidly iterate and improve models in a cloud native environment.

2.5 Configuration as Data in Action

In this section we look at real-world examples of how Configuration as Data can be applied across various industries. The idea is to demonstrate the tangible impact and versatility of treating configuration as data, showcasing its application in diverse operational contexts.

This section not only reinforces the practical value of Configuration as Data, but also underscores how Nephio's adoption of this principle supports a wide range of industry needs, enabling more streamlined, efficient, and adaptable network and service orchestration.

Through these real-world examples, we try to provide a clearer

understanding of how Configuration as Data serves as a foundational element in modernizing and optimizing digital operations across the board.

Real-World Examples of Configuration as Data in Various Industries

Configuration as Data is revolutionizing various industries by enabling automated, efficient, and error-free system management. In cloud computing, it allows for the automated provisioning and scaling of resources, enhancing operational efficiency.

Telecom companies use it for automating network functions, improving service reliability. Financial institutions leverage it for rapid deployment of trading systems and compliance management. Healthcare providers apply it to manage patient data systems, ensuring consistency and security. Retailers use it for dynamic management of online platforms and inventory systems, adapting swiftly to market trends. In manufacturing, it streamlines production lines and supply chain management, enhancing responsiveness to demand changes. These examples illustrate the widespread impact of this approach, showcasing its ability to transform industry operations through automation and precision.

Case Studies Showcasing the Benefits and Challenges

Configuration as Data has led to significant benefits and some challenges in various industries, as illustrated in the following case studies:

Cloud Service Provider:

A major cloud service provider implemented Configuration as Data to manage its global infrastructure. The result was a dramatic reduction in deployment times and human errors. However, the challenge was in training staff to adopt this new paradigm and ensuring the security of configuration data.

Telecommunications Company:

A telecom giant used Configuration as Data to automate its network

operations. This implementation led to improved network reliability and faster service rollout. The challenge was integrating this approach with legacy systems and managing the complexity of network configurations.

Retail Chain:

A large retail chain adopted Configuration as Data for its inventory and online store management. This change resulted in more agile responses to market trends and reduced operational costs. The challenge was ensuring data consistency across multiple platforms and dealing with the scalability of data.

Healthcare Provider:

A healthcare provider used Configuration as Data for patient management systems. The result was improved patient data handling and compliance with health regulations. The challenge was in maintaining the privacy and security of sensitive health data.

2.6 Controlling Reality

In this section, we focus on the concept of constant reconciliation in GitOps, a pivotal practice for maintaining desired system states in dynamic environments.

This discussion aims to unpack the mechanisms and advantages of employing continuous reconciliation as a means to ensure that the actual state of a system aligns with the desired state defined in code. By exploring how GitOps leverages version control as the single source of truth for system configurations, we highlight the role of constant reconciliation in facilitating automated, reliable, and efficient operations.

This section illustrates how Nephio integrates these principles to enhance the management and orchestration of network functions and infrastructure, emphasizing the importance of continuous monitoring and adjustment in achieving operational excellence. Through the lens of GitOps and constant reconciliation, we aim to show how Nephio empowers organizations to more effectively control their reality, ensuring that their digital ecosystems remain resilient, secure, and aligned with their strategic objectives.

Constant Reconciliation in GitOps

Constant reconciliation in GitOps is a process where the desired state of a system, defined in a Git repository, is continuously compared with its actual state in the runtime environment. If discrepancies are detected, automated processes are triggered to align the actual state with the desired state.

This approach ensures that the system is always in the intended configuration, enhancing reliability and stability. It allows for rapid response to configuration drifts, security breaches, or operational anomalies. Constant reconciliation embodies the principle of 'infrastructure as code', ensuring that changes are traceable, auditable, and reversible, thereby maintaining system integrity and consistency.

Dependency Handling

Often one service will depend on another. Unlike in traditional monolithic applications, however, it's impossible to know ahead of time whether that service is available. For example, let's say ServiceA depends on the database in ServiceB. To solve this problem in the imperative model, ServiceA would have to define the process it would use to determine ServiceB's availability, then what to do if it's not available, and so on.

Using the declarative model, the application does not define any of that. Instead, ServiceA simply declares that it depends on ServiceB, and Nephio and Kubernetes manage that dependency, ensuring that ServiceB is available before attempting to run ServiceA.

Massive Scale

Another area where declarative methods are important is scale. In the context of Nephio, handling infrastructure, network services, and applications together at the edge is crucial, because it enables users to define only the workload intent; from there, Nephio can infer the necessary infrastructure intent.

This integrated approach ensures seamless orchestration and deployment of services, crucial for edge computing where resources are distributed and varied. By inferring infrastructure requirements from workload intent,

Nephio can optimize resource allocation, enhance performance and ensure reliability. This capability is vital for managing the complexities at the edge, where quick, automated decisions about infrastructure are essential to support diverse and dynamic workloads efficiently.

2.7 The Future of Declarative Structures and Configuration as Data

Now let's take a more future-facing look, focusing on the prospective advancements and the evolving role of declarative paradigms in technology. This forward-looking discussion highlights the potential for declarative structures to further revolutionize the automation, management, and scalability of complex systems.

By speculating on future trends, we underscore the growing importance of these methodologies in simplifying operational complexities and enhancing system reliability. This section posits that as technology ecosystems become increasingly dynamic, the principles of declarative structures and configuration as data will become more integral to achieving efficient, agile, and resilient digital operations. Through this exploration, Nephio is positioned as a pioneering framework, ready to adapt and lead in the face of these future developments.

Declarative Structures and the Future

As we look towards the future of declarative structures and Configuration as Data, several key trends and developments emerge:

INCREASED AUTOMATION AND AI INTEGRATION:

Future systems will likely see deeper integration of AI and machine learning algorithms with declarative configurations. This integration could enable more intelligent and adaptive systems that can predict and adjust configurations based on usage patterns and environmental changes.

ENHANCED SECURITY AND COMPLIANCE:

As declarative configurations become more prevalent, there will be a greater

focus on embedding security and compliance checks within these configurations— ensuring that systems are not only efficient but also secure and compliant with evolving regulations.

CROSS-DOMAIN ORCHESTRATION:

Declarative structures will extend beyond their traditional domains, enabling seamless orchestration across different technology stacks and platforms. This cross-domain orchestration will facilitate more integrated and cohesive IT ecosystems.

EDGE COMPUTING AND IOT:

With the rise of edge computing and IoT, declarative configurations will become more critical in managing the vast array of devices and services at the edge, ensuring they operate optimally in resource-constrained environments.

SIMPLIFICATION AND DEMOCRATIZATION:

Tools and platforms will evolve to make declarative configurations more accessible to a broader range of users, including those without deep technical expertise. This democratization will enable more stakeholders to participate in system design and management.

REAL-TIME CONFIGURATION AND ADAPTATION:

Systems will move towards real-time configuration updates and adaptations, enabling them to respond instantly to changes in the environment or operational requirements.

STANDARDIZATION AND INTEROPERABILITY:

There will be a push towards standardizing declarative formats and ensuring interoperability between different systems and platforms, facilitating easier integration and collaboration.

In summary, the future of declarative structures and Configuration as Data is poised to bring more intelligence, security, and efficiency to system management, making it an integral part of modern IT infrastructure.

Emerging Trends: GitOps, Policy as Code, and More

Emerging trends like GitOps and Policy as Code are reshaping infrastructure management. GitOps extends the version control of Git to operational workflows, ensuring consistency and reliability. Policy as Code centralizes and automates policy enforcement, enhancing security and compliance.

Another particularly exciting potential future trend is Generative AI, which represents a shift towards using text prompts, rather than Kubernetes Resource Model (KRM), to drive Nephio. This approach could simplify operations, allowing users to specify high-level objectives while AI algorithms determine the optimal configuration and deployment strategies, potentially revolutionizing how infrastructure and services are managed and orchestrated.

The Role of Tools and Platforms in Furthering this Paradigm

Tools and platforms play a pivotal role in advancing the paradigm of declarative structures and Configuration as Data. They provide the necessary frameworks and environments for implementing and managing these configurations efficiently.

By offering user-friendly interfaces, automation capabilities, and integration options, these tools enable both technical and non-technical users to define, deploy, and manage complex systems with ease. They also facilitate collaboration, version control, and continuous monitoring, ensuring that systems remain in their desired state. As these tools evolve, they will further democratize access to advanced system management, making it more accessible and adaptable to various needs and scales.

In order for any of this to work, however, Nephio relies on Kubernetes, which we will discuss in the next chapter.

CHAPTER 3
KUBERNETES AS UNIFORM AUTOMATION CONTROL PLANE

Learning Objectives

By the end of this chapter, you should be able to:

- Describe Kubernetes' evolution from a container orchestration platform to a versatile tool for application and infrastructure management, and its role in Nephio's operations.
- Explain the concepts of Custom Resource Definitions (CRDs) and Operators in Kubernetes, and how they enable the management of complex applications.
- Recognize the benefits of treating infrastructure components as data within Kubernetes, enhancing automation, scalability, and reliability in cloud native environments.
- Identify Kubernetes' capabilities in application deployment, scaling, lifecycle management, and integration with package management and configuration tools.

3.1 Kubernetes: Beyond Container Orchestration

This section delves into the evolution of Kubernetes and its expansion beyond its original scope of container orchestration. It traces the trajectory of Kubernetes, highlighting how it has grown to incorporate a wide range of capabilities that address various aspects of application deployment, management, and scaling.

By examining the enhancements and new features added over time, we

illustrate Kubernetes' transformation into a comprehensive platform for managing complex, distributed systems across diverse environments. This discussion underscores the adaptability and scalability of Kubernetes, showcasing its role as a foundational element in modern cloud-native architectures. Through this exploration, we provide insights into how Kubernetes continues to evolve, shaping the future of digital infrastructure with its expanding ecosystem.

How Kubernetes Expanded Its Capabilities Over Time

Nephio relies on the capabilities of Kubernetes, so it is important for you to understand what Kubernetes is and how it works, and in particular how those functions that Nephio relies on work.

Kubernetes, originally designed as a container orchestration platform, has significantly expanded its capabilities over time, evolving into a comprehensive system for automating the deployment, scaling, and management of applications. Initially focused on simplifying container management, Kubernetes introduced concepts like pods, services, and deployments, streamlining how applications are run in distributed environments. As its ecosystem grew, Kubernetes embraced a broader range of functionalities, including stateful applications management, automated rollouts and rollbacks, and self-healing capabilities, where it automatically replaces and reschedules failed containers.

Introducing Custom Resource Definitions (CRDs)

The introduction of Custom Resource Definitions (CRDs) marked a pivotal expansion, allowing users to define custom resources, extending Kubernetes' native capabilities. This led to the development of Operators, specialized controllers that automate complex application management.

Kubernetes' extensible architecture now supports a wide range of workloads, from microservices to machine learning, making it a versatile tool for modern cloud native environments. This evolution reflects its shift from a container-centric tool to a universal orchestration system, enabling seamless

automation of infrastructure, applications, and configurations.

Operators take this a step further. They are software extensions that use Kubernetes APIs to create, configure, and manage instances of complex stateful applications on behalf of a Kubernetes user. They build upon the basic Kubernetes resource and controller concepts but include domain or application-specific knowledge to automate common tasks.

For example, consider a database like PostgreSQL. Using CRDs, you could define a PostgreSQLInstance resource, specifying the desired version, storage size, and other configurations. An Operator for PostgreSQL would then watch for these custom resources and manage the database instances, handling tasks like deployment, backups, and scaling according to the defined specifications. This approach simplifies complex operations, making them as straightforward as managing standard Ku bernetes workloads.

Creating a Custom Resource Definition (CRD) for a PostgreSQLInstance in Kubernetes involves defining the structure and schema of the resource. Below is a sample YAML definition for such a CRD. This example is simplified and focuses on key elements like version and storage size.

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
name: postgresqlinstances.example.com
spec:
group: example.com
versions:
- name: v1
served: true
storage: true
schema:
openAPIV3Schema:
type: object
properties:
spec:
type: object
properties:
version:
```

```
type: string
description: "Version of PostgreSQL"
storageSize:
type: string
description: "Size of the storage allocated to PostgreSQL instance"
scope: Namespaced
names:
plural: postgresqlinstances
singular: postgresqlinstance
kind: PostgreSQLInstance
shortNames:
- pgsql
```

This CRD defines a new resource type named **`PostgreSQLInstance`**. It includes specifications for the PostgreSQL version and the storage size. Once this CRD is applied to a Kubernetes cluster, users can create custom resources based on this definition. For example:

```
apiVersion: "example.com/v1"
kind: PostgreSQLInstance
metadata:
name: my-postgresql-instance
spec:
version: "13.0"
storageSize: "10Gi"
```

This YAML snippet creates an instance of **`PostgreSQLInstance`** with specified version and storage size. The actual implementation of how this instance is provisioned and managed would be handled by an operator designed to understand and act on these custom resources.

Creating an operator for a **`PostgreSQLInstance`** CRD involves writing a controller that watches for changes to these custom resources and then acts to ensure the desired state (as defined in the CRD) is achieved and maintained. This typically involves deploying and managing PostgreSQL instances according to the specifications defined in each PostgreSQLInstance resource.

Operator Examples

Here's a high-level overview of what an operator might do:

- **Watch for changes:** The operator would use the Kubernetes API to watch for changes to `PostgreSQLInstance` resources. This includes new resources being created, and existing ones being updated or deleted.
- **Deploy PostgreSQL Instances:** When a new `PostgreSQLInstance` resource is created, the operator would deploy a PostgreSQL database instance. This could involve setting up a `StatefulSet, PersistentVolumeClaims, Services,` and other necessary Kubernetes resources. The operator would configure the database instance based on the specifications in the `PostgreSQLInstance` resource, such as the version and storage size.
- **Manage Lifecycle:** The operator would handle lifecycle events such as backups, updates, scaling, and recovery. For example, if the version field in the CRD is updated, the operator would manage the upgrade process for the PostgreSQL instance.
- **Handle Failures:** The operator would monitor the health of the PostgreSQL instances and take corrective actions in case of failures. This could include restarting failed instances, restoring from backups, or reallocating resources.
- **Reporting Status:** The operator would update the status of each `PostgreSQLInstance` resource to reflect the current state of the corresponding PostgreSQL instance, providing visibility into the health and status of managed databases.
- **Custom Logic:** Depending on requirements, the operator might include additional logic, such as integrating with monitoring systems, enforcing security policies, or optimizing performance.

The actual implementation of the operator would require programming, typically in Go, using the Kubernetes client libraries. The operator pattern in

Kubernetes is powerful because it encapsulates domain-specific knowledge, allowing complex applications like databases to be managed as easily as native Kubernetes resources.

For example:

```
package main
import (
// Import necessary packages
)
// Define the CRD structure
type PostgreSQLInstance struct {
// ...
}
func main() {
// Set up Kubernetes client
// ...
// Start informer to watch PostgreSQLInstance resources
// ...
// Reconcile loop
for {
// Get the next event
event := <-events
switch event.Type {
case Added:
// Handle new PostgreSQLInstance
deployPostgreSQLInstance(event.Object)
case Modified:
// Handle updates
updatePostgreSQLInstance(event.Object)
case Deleted:
// Handle deletion
deletePostgreSQLInstance(event.Object)
}
}
}
func deployPostgreSQLInstance(instance *PostgreSQLInstance) {
// Deploy a new PostgreSQL instance
// ...
```

```
}
func updatePostgreSQLInstance(instance *PostgreSQLInstance) {
// Update an existing PostgreSQL instance
// ...
}
func deletePostgreSQLInstance(instance *PostgreSQLInstance) {
// Clean up resources
// ...
}
```

3.2 Infrastructure as Data with Kubernetes

Now let's explore the concept of modeling infrastructure components as Kubernetes objects, a key practice that exemplifies the shift towards treating infrastructure as code. This approach enables the precise definition, versioning, and management of infrastructure using the same principles and tools used for application development.

By encapsulating infrastructure elements as Kubernetes objects, organizations can achieve greater consistency, efficiency, and agility in their operations. This section highlights how Kubernetes facilitates this paradigm shift, allowing for dynamic and scalable management of infrastructure resources in a cloud-native environment.

Modeling Infrastructure Components as Kubernetes Objects

In actuality, Kubernetes already works with infrastructure objects. As part of the ClusterAPI, it can define machines, nodes, networks and other objects, and manage them just as it manages higher level objects such as Pods. In the case of Nephio, we are simply extending these capabilities to additional infrastructure objects such as networks and edge devices.

Benefits: Consistency, Repeatability, and Scalability

By treating infrastructure elements like physical nodes, networks, and storage as Kubernetes objects, Nephio leverages Kubernetes' declarative model and robust orchestration capabilities. This approach simplifies the management of complex, distributed network functions, enabling them to be defined, deployed, and managed with the same tools and processes used for containerized applications. It enables seamless integration of infrastructure management with application lifecycle, en hancing automation, scalability, and reliability.

This model also facilitates a more agile and responsive infrastructure, where changes and updates can be rolled out quickly and consistently, aligning with modern DevOps practices and the demands of cloud native environments.

Use Cases: Provisioning Cloud Resources and Network Configurations

In the context of Nephio, which focuses on automating and orchestrating network functions using Kubernetes, the use cases of provisioning cloud resources and network configurations are pivotal. Here's how Nephio addresses these use cases:

Provisioning Cloud Resources:

- Nephio enables automated provisioning of cloud resources, such as virtual machines, containers, or serverless functions, across different cloud environments.
- It leverages Kubernetes' capabilities to orchestrate these resources efficiently, ensuring they are allocated and scaled according to the needs of the network functions.
- This automation extends to multi-cloud and hybrid cloud environments, allowing network services to be deployed seamlessly across various cloud providers, enhancing flexibility and optimizing resource utilization.

Network Configurations:

- Nephio excels in automating complex network configurations necessary for modern network functions.
- It can manage various aspects of networking, such as virtual network interfaces, load balancers, and network policies, ensuring secure and efficient communication between different components.
- The platform can also handle more advanced networking requirements like service mesh configurations, network slicing for 5G applications, and edge computing scenarios, where network agility and performance are critical.

In all these use cases, Nephio's integration with Kubernetes is key. It not only simplifies the management of these resources but also ensures they are managed in a cloud native, scalable, and resilient manner, extending the Kubernetes APIs through the use of controllers.

This alignment with Kubernetes also means that network functions can benefit from the broader ecosystem of tools and best practices developed around Kubernetes, further enhancing the capabilities and efficiency of network services deployment and management.

3.3 Application Deployment and Management

In this section we focus on enabling developers and operators to define applications and their resources in declarative manifests, which Kubernetes then uses to ensure the application's desired state is achieved and maintained.

By utilizing Kubernetes manifests, teams can automate deployment processes, scale applications efficiently, and manage updates with minimal downtime. This section highlights the flexibility and control Kubernetes offers over application deployment and management, demonstrating its effectiveness in supporting complex, distributed applications across various environments.

Deploying Applications Using Kubernetes Manifests

Deploying applications to Kubernetes involves creating and configuring Kubernetes resources to manage the application's lifecycle. A typical deployment starts with a Docker container image of the application. You then define a Kubernetes Deployment, which tells Kubernetes how to run the application, including the number of replicas, resource limits and update strategy.

Here's a simple example of deploying a basic web application:

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: web-app-deployment
spec:
replicas: 3
selector:
matchLabels:
app: web-app
template:
metadata:
labels:
app: web-app
spec:
containers:
- name: web-app
image: my-web-app:latest
ports:
- containerPort: 80
```

This YAML file creates a Deployment named **`web-app-deployment`** for a web application. It specifies three replicas of the pod, each running a container based on the **`my-web-app:latest`** image. Kubernetes ensures that three instances of the application are always running, handling deployment, scaling, and recovering automatically.

Scaling, Updating, and Managing Application Lifecycles

Scaling, updating, and managing application lifecycles in Kubernetes are fundamental aspects that ensure applications are running efficiently and are up-to-date.

SCALING:

Kubernetes allows for both manual and automatic scaling. For instance, you can scale a deployment to meet increased demand in a declarative way by updating the YAML and then applying the change. For example, we could change our example to use 5 replicas instead of 3:

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: web-app-deployment
spec:
replicas: 5
selector:
matchLabels:
app: web-app
...
```

Then we'd go ahead and apply the patch:

```
kubectl apply -f web-app-deployment.yaml
```

Kubernetes then makes the change and ensures that 5 replicas are always running.

You can manage other aspects of the application, such as the foundation image, in the same way.

MANAGING LIFECYCLES:

Kubernetes provides lifecycle hooks for fine-grained management, like **`preStop`** and **`postStart`** hooks, to run scripts before a container stops or after it starts. This is crucial for graceful shutdowns and initializations.

These features, combined with Kubernetes' self-healing capabilities, ensure applications are not just deployed but are also resiliently maintained throughout their lifecycle. Kubernetes handles failures, automatically

replaces unhealthy instances, and adjusts based on load, making the management of application lifecycles robust and responsive to the dynamic needs of modern applications.

Helm: Package Management for Kubernetes Applications

Helm is a powerful and widely-used package manager for Kubernetes, streamlining the deployment and management of applications. It uses a packaging format called charts, which are collections of pre-configured Kubernetes resources tailored to a specific application or service.

These charts describe everything needed to run an application, service, or tool on Kubernetes, making it easier to manage complex deployments with many interdependent components.

With Helm, users can install, upgrade, and manage applications on Kubernetes clusters in a consistent and repeatable way. Charts can be shared through Helm repositories, fostering collaboration and reuse within the community.

Helm also supports versioning, allowing users to roll back to previous versions of a chart if needed. This simplifies the process of configuring and deploying applications on Kubernetes, making it accessible even to those who are not experts in Kubernetes' intricacies.

Helm's templating engine allows for customization of deployments, ensuring that applications can be configured to meet specific needs while maintaining the simplicity and reliability of a standardized deployment process.

For example, our sample app might start with a basic chart:

```
webapp-chart/
├── Chart.yaml
├── values.yaml
├── templates/
│   ├── deployment.yaml
│   └── service.yaml
```

`Chart.yaml` is metadata about the chart:

```
apiVersion: v2
```

```
name: webapp-chart
description: A Helm chart for deploying a basic web application
version: 0.1.0
```

`Values.yaml` defines default configuration values:

```
replicaCount: 3
image:
repository: nginx
tag: latest
pullPolicy: IfNotPresent
service:
type: ClusterIP
port: 80
```

You can override these values when you deploy the chart to create the application.

`templates/deployment.yaml` does what it says on the tin, and provides the template for the deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: webapp-deployment
spec:
replicas: {{ .Values.replicaCount }}
selector:
matchLabels:
app: webapp
template:
metadata:
labels:
app: webapp
spec:
containers:
- name: webapp
image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
imagePullPolicy: {{ .Values.image.pullPolicy }}
ports:
- containerPort: 80
```

templates/service.yaml does the same for the service:

```
apiVersion: v1
kind: Service
metadata:
name: webapp-service
spec:
type: {{ .Values.service.type }}
ports:
- port: {{ .Values.service.port }}
targetPort: 80
selector:
app: webapp
```

3.4 Configuration Management in Kubernetes

Now let's look at the mechanisms for managing configuration data, specifically through the use of **ConfigMaps** and **Secrets.** This part of the discussion emphasizes how Kubernetes provides structured approaches to handle application configurations and sensitive data, enabling applications to be more dynamic and environment-agnostic.

By leveraging **ConfigMaps** for non-confidential data and **Secrets** for sensitive information, Kubernetes allows for a more secure and organized way to manage configuration data across different environments and deployments. This section showcases the practicality and security benefits of using these Kubernetes objects, illustrating their role in facilitating more flexible and scalable application deployments.

ConfigMaps and Secrets: Managing Configuration Data

ConfigMaps and **Secrets** are Kubernetes objects used to store non-confidential and confidential data, respectively.

ConfigMaps are used to keep configuration data that can be consumed

by pods or other Kubernetes objects. This data can include configuration files, command-line arguments, environment variables, port numbers, etc. They help in keeping containerized applications portable and easy to configure without hardcoding configuration data.

Secrets are similar to **ConfigMaps** but are used to store sensitive information like passwords, OAuth tokens, and SSH keys. They provide a more secure way of managing confidential data, ensuring that it's not exposed or stored in the application's code.

Example of a **ConfigMap:**

```
apiVersion: v1
kind: ConfigMap
metadata:
name: my-config
data:
my-key: my-value
config.json: |
{
"database": "mydb",
"server": "localhost"
}
```

Example of a **Secret**:

```
apiVersion: v1
kind: Secret
metadata:
name: my-secret
type: Opaque
data:
password: MWYyZDFlMmU2N2Rm
```

In this example, the ConfigMap **my-config** stores a key-value pair, and the Secret **my-secret** securely stores a base64-encoded password. These can be mounted into pods or referenced in other Kubernetes objects. For example:

```
apiVersion: v1
kind: Pod
metadata:
```

```
name: mypod
spec:
containers:
- name: mycontainer
image: myimage
volumeMounts:
- name: config-volume
mountPath: /etc/config
- name: secret-volume
mountPath: /etc/secret
volumes:
- name: config-volume
configMap:
name: my-config
- name: secret-volume
secret:
secretName: my-secret
```

Dynamic Configuration Using Operators and Controllers

Dynamic configuration in Kubernetes using operators and controllers is a powerful method to automate and manage complex applications and infrastructure. Operators and controllers are extended Kubernetes components that encode domain-specific knowledge for specific applications or infrastructure components. They watch for changes in the Kubernetes API and react by creating, updating, or deleting resources as needed.

Operators are custom controllers with domain-specific knowledge built-in. They extend Kubernetes to create, configure, and manage instances of complex stateful applications. An operator uses custom resources (CRs) to manage applications and their components based on the desired state defined by the user. It continuously monitors these resources and applies the necessary changes to the system to achieve and maintain the desired state. For example, a database operator can manage database instances, handle backups, and perform upgrades automatically.

Controllers are loops that watch the state of your cluster through the Kubernetes API and make changes attempting to move the current state towards the desired state. They are fundamental to Kubernetes' self-healing mechanism. For example, a Deployment controller watches for changes to Deployment resources and ensures that the actual state (like the number of running pods) matches the desired state specified in the Deployment.

Dynamic Configuration:

Operators and controllers enable dynamic configuration by continuously monitoring and reacting to changes in the cluster. They can automatically adjust configurations in response to changes in the environment, workload, or other external factors.

This dynamic approach allows for more resilient, scalable, and self-managing applications. For example, an operator could automatically scale a database cluster up or down based on traffic patterns, or a controller could automatically update application configurations without manual intervention.

In summary, operators and controllers bring automation, intelligence, and higher-level abstraction to Kubernetes, making it easier to deploy and manage complex, stateful applications with dynamic configuration needs.

Integrating with External Configuration Management Tools

Integrating Nephio with external configuration management tools such as Ansible and OpenTofu (aka “open source Terraform”) represents a strategic approach to enhancing the orchestration and management capabilities of network functions and services in cloud native environments.

Ansible and OpenTofu, known for their simplicity and ease of use, can play a crucial role in the deployment and configuration of the underlying infrastructure for Nephio.

By leveraging their automation capabilities, Nephio can streamline the provisioning of servers, networking, and other resources necessary for running Kubernetes clusters. They can be used to define the desired state of the infrastructure, ensuring consistency and repeatability.

This integration allows for a more efficient setup and maintenance process, reducing manual overhead and potential for errors. Additionally, Ansible's extensive module library and community support and OpenTofu's plentiful templates make them versatile tools for managing a wide range of infrastructure components. Nephio can connect with Ansible and OpenTofu through the use of Controllers.

Together, Ansible and OpenTofu extend Nephio's capabilities, providing a comprehensive solution for automating and managing both the infrastructure and network functions in a unified manner. This integration supports the goal of achieving efficient, scalable, and reliable network services in 5G and edge computing environments.

3.5 Seamless Collaboration with Kubernetes

Kubernetes plays an integral role in enhancing DevOps and GitOps workflows, facilitating a more collaborative and efficient approach to software development and operations.

In particular, the Kubernetes architecture and ecosystem support the principles of continuous integration, continuous delivery (CI/CD), and infrastructure as code, key components of modern DevOps practices.

By enabling seamless automation and deployment process es, Kubernetes acts as a catalyst for teams adopting GitOps methodologies, where the desired state of the infrastructure is declared in version-controlled repositories.

The Role of Kubernetes in DevOps and GitOps Workflows

Kubernetes significantly enhances DevOps and GitOps workflows by providing a scalable and efficient platform for automating the deployment, scaling, and management of applications.

In DevOps, Kubernetes facilitates Continuous Integration and Continuous Deployment (CI/CD) by seamlessly integrating with CI/CD tools,

supporting automated testing, and enabling smooth application updates through rolling updates and canary deployments.

Its support for microservices architecture and self-healing capabilities aligns with DevOps goals of high availability and resilience.

In GitOps, Kubernetes' declarative configuration model perfectly complements the GitOps approach, where the desired state of the infrastructure is maintained in a version-controlled Git repository. This setup allows for automated synchroni zation, ensuring that the actual state in Kubernetes matches the desired state defined in Git. This integration provides a robust framework for collaboration, version control, and easy rollbacks, making Kubernetes an essential tool in modern DevOps and GitOps practices.

Collaborative Benefits: Version Control, CI/CD Integration, and Review Processes

In the context of Kubernetes and modern development practices, collaborative benefits such as version control, CI/CD integration, and review processes are crucial.

Version control systems like Git enable teams to track changes, collaborate on code, and manage revisions efficiently. This is essential for maintaining code quality and history, especially in large-scale projects with multiple contributors.

Integrating Kubernetes with CI/CD pipelines automates the deployment process, ensuring that code changes are automatically built, tested, and deployed, leading to faster and more reliable releases. Furthermore, the review process becomes more streamlined and transparent, as every change can be reviewed and tested automatically before being merged. This collaborative ecosystem fosters a culture of continuous improvement, where code quality is maintained, and innovation is accelerated, all while ensuring that the development process remains efficient and error-free.

Case Study: Collaborative Development Using Kubernetes Tooling

In a leading tech organization, three development teams collaborated on a complex project using Kubernetes tooling, each responsible for different aspects: infrastructure management, workloads, and network configuration. The project's success hinged on seamless integration and efficient collaboration between these teams.

The infrastructure team focused on setting up and maintaining the Kubernetes clusters. They used Kubernetes tooling to automate the provisioning of resources, ensuring that the underlying infrastructure was scalable, resilient, and aligned with the project's needs. Their work included managing nodes, storage, and computing resources, all defined and controlled through Kubernetes manifests and configurations.

The workloads team was responsible for deploying and managing the applications. They utilized Kubernetes deployments, services, and pods to ensure that the applications were correctly containerized, deployed, and scaled within the clusters. This team worked closely with the infrastructure team to ensure that the applications' requirements were met by the underlying infrastructure.

The network configuration team handled the intricate task of setting up network policies, ingress controllers, and service meshes within the Kubernetes environment. Their role was crucial in ensuring secure and efficient communication between different services and external access points.

The use of Kubernetes as a common platform allowed these teams to work in a highly integrated manner. Changes made by one team were instantly visible and adaptable by the others, fostering a collaborative environment. This unified approach led to faster development cycles, reduced errors, and a more agile response to changing requirements. The project demonstrated the importance of using a consistent set of tools, as enabled by Kubernetes, in managing both workloads and infrastructure, aligning perfectly with the principles of Nephio.

3.6 The Power of Unified Tooling

Using a single set of tools for managing infrastructure, applications, and configurations underscores the efficiency and coherence of the Kubernetes ecosystem. Now we'll look at how unified tooling simplifies the complexity inherent in cloud-native environments, enabling developers and operators to maintain consistency, enhance productivity, and reduce the potential for errors across the deployment pipeline.

By leveraging Kubernetes and its comprehensive toolset, teams can achieve a more streamlined workflow, from development through deployment to management, fostering a more integrated and automated approach to system administration.

Advantages of Using a Single Set of Tools for Infrastructure, Applications, and Configurations

Using a single set of tools for managing infrastructure, applications, and configurations, as seen in the context of Nephio, offers significant advantages. This unified approach, primarily driven by Kubernetes, streamlines processes and enhances efficiency. It reduces the learning curve and complexity because teams need to familiarize themselves with only one toolset, fostering better collaboration and faster onboarding.

Consistency across different stages of deployment and management is another key benefit, as it minimizes errors and discrepancies that often arise from using disparate tools. This consistency also simplifies troubleshooting and maintenance.

Furthermore, it allows for seamless automation and integration, as the same tools and processes can be applied across various aspects of the system, from setting up the infrastructure to deploying applications and managing their configurations. This holistic approach leads to more robust, scalable, and maintainable systems, aligning well with the principles of DevOps and GitOps.

How This Unified Approach Reduces Complexity and Learning Curves

By using a consistent set of tools, teams avoid the overhead of learning and integrating multiple disparate systems. This uniformity streamlines processes, making it easier for new team members to onboard and for existing members to deepen their expertise.

It also minimizes the potential for errors and inconsistencies that often arise when juggling different tools and methodologies. As a result, teams can focus more on innovation and problem-solving rather than grappling with the intricacies of varied technologies, leading to increased efficiency and productivity in development and operational tasks.

Enhancing Team Productivity and Reducing Operational Overhead

By streamlining processes under a common framework, teams spend less time navigating different systems and more time on value-adding activities. This consistency eliminates the need for context switching between various tools, allowing teams to work more efficiently and cohesively. It also simplifies training and onboarding, as there's a singular, focused learning path.

Moreover, this approach leads to a reduction in operational overhead, as maintaining and updating one integrated system is more straightforward and cost-effective than managing multiple disparate tools. Overall, this strategy leads to smoother workflows, faster deployment cycles, and a more agile response to changing requirements, thereby boosting overall productivity and operational efficiency.

CHAPTER 4
NEPHIO INTERNALS

Learning Objectives

By the end of this chapter, you will be able to:

- Understand the core components of Nephio, including Kubernetes as a control plane and the declarative automation framework.
- Recognize the importance of understanding Nephio's internals for effective integration and utilization in network function automation.
- Describe Nephio's architecture, focusing on its controllers, KRM functions, and operators, and their roles in resource management.
- Discuss Nephio's integration points, APIs, and security considerations, emphasizing the platform's performance and scalability in large-scale deployments.

4.1 Introduction to Nephio Internals

Let's start with a brief overview of Nephio's core components, laying the foundation for a deeper understanding of the internal workings of this innovative platform and letting us demystify the architecture and key elements that make up Nephio, illustrating how its design facilitates the orchestration and management of cloud-native network functions and infrastructure.

By dissecting the core components, we highlight the modular and scalable nature of Nephio, showcasing its ability to support complex deployments across diverse environments. This introduction serves as a gateway to appreciating the sophistication and capabilities of Nephio, setting the stage for further exploration of its functionalities and benefits.

A Brief Overview of Nephio's Core Components

Nephio is designed to automate network functions and their supporting infrastructure through Kubernetes. At its core, it consists of two primary components:

- **Kubernetes as a Uniform Automation Control Plane:** Kubernetes serves as the foundation for configuring all elements of the distributed cloud and network functions. It simplifies automation and enables active reconciliation across the entire stack, ensuring consistent and reliable operations.
- **Declarative Automation Framework:** Built atop Kubernetes, this framework utilizes projects like kpt (porch), and configSync and to manage configurations as data. It allows for the creation, customization, and deployment of network functions through standardized configuration packages, streamlining the process and reducing complexity.

These components work in tandem to provide a robust solution for the deployment and management of network functions, catering to the dynamic needs of modern distributed cloud environments. The focus is on simplifying the intricate processes involved in multi-vendor, multi-site network function deployment, ensuring efficiency and reliability.

The Importance of Understanding Nephio's Internals for Successful Integration

Understanding Nephio's internals is crucial for successful integration of your components because it involves a com plex interplay of Kubernetes-based automation and network function management, and it’s easy to get lost.

In the following sections we’ll give you the information you need to effectively utilize Nephio's capabilities, customize its components, and troubleshoot issues. Mastery of Nephio's internals ensures seamless

integration with existing systems, enabling users to fully harness its potential to automate and orchestrate network functions across distributed cloud environments.

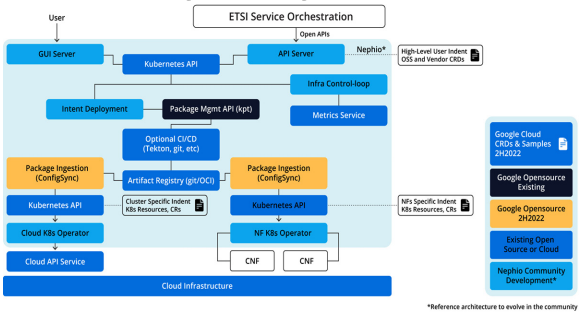
4.2 Nephio's Architecture

Now let's delve into the high-level architecture of Nephio, providing a comprehensive view of how its design supports the orchestration and automation of network functions and infrastructure. This exploration aims to outline the structural components and their interactions within Nephio, highlighting the platform's ability to streamline complex network operations through cloud-native technologies.

By examining the architecture, we illustrate the scalability, flexibility, and resilience built into Nephio, showcasing its capacity to accommodate diverse deployment scenarios and operational demands. This section is intended to give readers a clear understanding of Nephio's architectural principles and how they contribute to its effectiveness in managing cloud and edge computing environments.

High-Level Architecture of Nephio

The first thing we need to do is get a high-level look at how Nephio works.



Nephio Reference Architecture

As you can see from this reference architecture, everything starts with a request, either from a user through a Graphical User Interface (GUI) or an application through the API server. In either case, the request is routed through Kubernetes, which looks at the Intent – what is it that the user

wants to deploy – and routes the request to the Package Management API.

Nephio also lets the control loop know about the request, so that once the object or objects have been instantiated, the control loop can ensure that they are always available. The Package Management API retrieves the object definition from the artifact registry, and ConfigSync sends instructions back to Kubernetes to set up the objects. From there, Kubernetes interacts with the appropriate Operator to manage the objects.

Note that when it comes to Nephio, this process is the same whether we are talking about application configuration, workloads, or infrastructure. The only difference is the type of Kubernetes Operator that makes it possible, and where in the process that Operator has to intervene.

Core Components and Their Interactions

A Nephio-enabled installation has several core components, including:

- **Kubernetes:** Kubernetes (K8s) serves as the core engine of Nephio, providing the underlying automation capabilities, orchestration, and operational consistency required to manage and scale containerized network functions across various environments.
- **Nephio Cloud vs. Edge Cloud:** Nephio Cloud refers to the centralized management layer that orchestrates and automates network functions, while Edge Cloud denotes the distributed infrastructure closer to the data source, where Nephio deploys and manages workloads for reduced latency and localized processing.
- **ConfigSync:** ConfigSync is a tool used in Nephio to synchronize configurations across multiple Kubernetes clusters, ensuring consistency and adherence to the declared state of infrastructure and applications as defined in the source of truth.
- **Porch:** Porch is a component within Nephio that acts as an interface for managing Kubernetes-native packages, enabling the declaration and control of network functions and infrastructure configurations in a Kubernetes environment.
- **KRM Functions:** Kubernetes Resource Model (KRM) functions

are modular, composable pieces in Nephio that process and transform configurations, facilitating the automation of complex tasks and ensuring the desired state of resources within Kubernetes clusters.

- **Custom Resource Definitions and Kubernetes Operators:** Nephio provides a number of Custom Resources and Operators that enable Kubernetes to manage them.

Infrastructure Orchestration vs Workload (Network Service or Application)
Orchestration vs Workload Configuration

Infrastructure orchestration, workload orchestration, and workload configuration represent three distinct layers of automation in a cloud environment.

Infrastructure orchestration focuses on provisioning and managing the underlying physical or virtual resources like servers, storage, and networking. **Workload** (network service or application) **orchestration** deals with deploying and managing these services or applications across the orchestrated infrastructure, ensuring they are running as intended. **Workload configuration**, on the other hand, involves setting the parameters and settings within the workloads themselves, defining how they operate, communicate, and process data.

Each layer is crucial for the seamless operation of cloud services, with Nephio providing tools and frameworks to manage these layers effectively in a Kubernetes-centric ecosystem.

Intent mutation

Nephio handles workload intent through controllers which reconcile the high level Infrastructure intent by mutating the workload configuration based on the target environment where it is going.

One example is the free5GC deployment using the high level Nephio intent. The user is expected to specify the end to end topology intent, and the Nephio controllers in the backend create the necessary kubernetes resources

to clone and mutate the 5GC component controller packages.

The controllers consult the target kubernetes clusters contexts and allocate resources like IP address, and create network attachment definitions for the CNIs in the target clusters. The hydrated packages are then pushed to the cluster-specific repositories, from which they are reconciled by the configsync in the target clusters.

This way a high level intent specified triggers a workflow of Kuberenetes resource creation, as well as network and compute resource allocation, followed by the deployment of all resources in the target clusters, resulting in the end to end 5G service deployment.

```
apiVersion: nf.nephio.org/v1alpha1
kind: FiveGCoreTopology
metadata:
name: fivegcoretopology-sample
spec:
upfs:
- name: "agg-layer"
selector:
matchLabels:
nephio.org/region: us-central1
nephio.org/site-type: edge
namespace: "upf"
upf:
upfClassName: "free5gc-upf"
capacity:
uplinkThroughput: "1G"
downlinkThroughput: "10G"
n3:
- networkInstance: "sample-vpc"
networkName: "sample-n3-net"
n4:
- networkInstance: "sample-vpc"
networkName: "sample-n4-net"
n6:
- dnn: "internet"
uePool:
```

```
networkInstance: "sample-vpc"
networkName: "ue-net"
prefixSize: "16"
endpoint:
networkInstance: "sample-vpc"
networkName: "sample-n6-net"
```

Kubernetes as the Foundation of Nephio

Kubernetes' capabilities form the backbone of Nephio's functionality, providing a robust and scalable platform for container orchestration. Its declarative approach enables users to specify their desired state for the infrastructure and workloads, which Kubernetes actively maintains, ensuring consistency and reliability.

Kubernetes' extensibility through Custom Resource Definitions (CRDs) and operators enables Nephio to tailor the orchestration to specific network functions and services. The self-healing mechanisms of Kubernetes ensure that applications are always running, and its service discovery and load balancing features facilitate seamless communication between different services. By leveraging these features, Nephio can automate complex network functions and manage them efficiently across distributed cloud environments, making it a powerful tool for modern infrastructure management.

Let’s take a closer look at how that works.

5.3 Deep Dive into Controllers

Next we’ll embark on an exploration of the pivotal role controllers play within Nephio's ecosystem. This detailed examination aims to shed light on how controllers function as the backbone of Nephio, orchestrating and managing the state of resources to align with defined specifications.

By delving into the specifics of controllers, we illustrate their critical role in automating processes, ensuring consistency, and facilitating the dynamic adjustment of network functions and infrastructure in response to real-time conditions.

The Role of Controllers in Nephio: Overview

In Nephio, Kubernetes controllers play a pivotal role by overseeing the lifecycle of resources and ensuring the actual state aligns with the desired state defined by the users. They watch for changes in resource states and enact policies to manage those resources effectively.

Controllers extend Kubernetes' capabilities, allowing Nephio to automate complex network functions and infrastructure management tasks. They are essential for the reconciliation process, where they continuously apply the desired configurations, handle scaling, and manage updates and rollbacks, thus providing a resilient and self-healing system that adapts to the dynamic nature of cloud native environments.

How Controllers Manage and Interact with Resources

In the context of Nephio, controllers are responsible for managing and interacting with various Kubernetes resources. They watch for changes in the state of resources, such as custom resources specific to Nephio's domain, and work to reconcile the current state with the desired state defined by the user.

Controllers in Nephio typically interact with resources by creating, updating, or deleting Kubernetes objects, such as pods, services, or custom-defined network functions. They ensure that the infrastructure and workloads are configured and running as intended, handling any discrepancies by applying the necessary adjustments. This continuous loop of monitoring and reconciliation allows Nephio to maintain a resilient and adaptive cloud native environment.

SPECIFIC CONTROLLERS AND THEIR FUNCTIONALITIES

Nephio includes several controllers:

- **Approval Controller:** Automates the transition of package revisions from Draft to Proposed and then to Published based on a policy annotation, ensuring that only ready and unique revisions are published.

- **Bootstrap Package Controller:** Responsible for initializing packages on a new cluster, such as installing a GitOps tool like config-sync, to enable subsequent configuration through the GitOps toolchain.
- **Bootstrap Secret Controller:** Aims to bootstrap secrets on a new cluster, ensuring tools like config-sync are set up, which is crucial for handling configurations through GitOps workflows.
- **Repository Controller:** Manages the lifecycle of repositories within Gitea, a Git service, ensuring that repository-related operations are reflected and maintained in the version control system.
- **Token Controller:** Handles the lifecycle of tokens in Gitea and syncs them as secrets within Kubernetes, which is essential for secure interactions with the Git server.
- **Network Controller:** Automates the allocation of network controllers such as IPAM and NAD.

4.4 Kubernetes Resource Model (KRM) Functions in Nephio

In this section we introduce the concept of KRM functions and their significance within the Nephio framework. This segment is designed to elucidate how KRM functions enhance Nephio's capabilities by enabling more sophisticated management and orchestration of resources.

By leveraging KRM functions, Nephio can extend Kubernetes' native functionalities, allowing for a more nuanced and flexible approach to deploying and managing network functions and infrastructure. This discussion highlights the pivotal role of KRM functions in streamlining operations, promoting scalability, and ensuring the adaptability of services within cloud-native environments.

KRM Functions and Their Significance

KRM (Kubernetes Resource Model) functions are pivotal in Nephio, serving

as modular tools that process and manipulate Kubernetes resources for specific purposes. They enable Nephio to extend Kubernetes' native capabilities, enabling sophisticated manipulation of resources that aligns with declarative infrastructure and application management. By leveraging these functions, Nephio can automate complex configurations and operational tasks, ensuring that the infrastructure and services are deployed and managed in a consistent and predictable manner.

How KRM Functions Enhance Kubernetes Customizations and Integrations

KRM functions in Nephio enhance Kubernetes customizations by providing a mechanism to implement complex logic that goes beyond the standard Kubernetes API capabilities. They act as transformers or validators of Kubernetes objects, enabling Nephio to tailor resource definitions to meet the specific needs of network functions and services. This customization provides seamless integration of diverse cloud native technologies and vendor-specific solutions, facilitating a more flexible and adaptable infrastructure. KRM functions thus play a crucial role in enabling Nephio to provide a unified and automated approach to managing network functions across various cloud environments.

SPECIFIC KRM FUNCTIONS AND THEIR USE CASES

Nephio provides several KRM functions:

- **`ipam-fn:`** This function claims IP addresses from a backend based on the content of the **`IPClaim,`** working independently or within a kpt pipeline. It's crucial for managing IP allocation in Nephio's network configurations.
- **`dnn-fn:`** This function creates **`IPClaim`** resources for IP pools specified in **`DataNetwork`** resources and updates the status of the original **`DataNetwork`** resource with the IP claim result, automating the IP management process in Nephio.
- **`gen-configmap-fn:`** This function generates **`ConfigMap`**s from function inputs, facilitating easier editing and management of

application configurations within Nephio.

- **`interface-fn:`** This function manages IP and VLAN claims and creates **`NetworkAttachmentDefinition`**s based on the CNI type in the cluster, playing a key role in network interface configuration in Nephio.
- **`configinject-fn:`** This function automates the injection of configuration data into network functions, streamlining the process of configuring and deploying network services in a Kubernetes environment.

5.5 Operators in Nephio

Now let's look at the role of operators, a key component in the Nephio ecosystem that significantly enhances its automation and management capabilities.

Operators extend Kubernetes' functionality by encapsulating operational knowledge for specific applications or services, thereby automating complex tasks and ensuring systems run optimally and resiliently, and help manage lifecycle events, from deployment and updates to scaling and healing processes, within Nephio's architecture.

The Role of Operators

In Kubernetes, operators are software extensions that utilize custom resources to manage applications and their components. They follow Kubernetes principles, notably the control loop, to maintain the desired state of a service.

In the context of Nephio, operators play a crucial role by automating complex tasks such as deployments, updates, and maintenance of network functions. They extend Kubernetes' capabilities to better handle the lifecycle management of Nephio's resources, ensuring that the network services are consistently deployed and operated across various environ ments, thus simplifying the complexities of edge-to-cloud orchestration.

How Operators Facilitate Automation and Management in Nephio

Operators in Nephio facilitate automation and management by encapsulating operational knowledge into software. They automate complex tasks such as deployments and updates, and manage the lifecycle of Nephio's resources, ensuring consistent and reliable operations across various environments.

Specifically, the Nephio Controller Manager operator acts as a central management point for all Nephio management controllers, such as specializer controllers, bootstrap controller, repo-controller, edge watcher, and so on. It consolidates various reconcilers into a single binary, streamlining the management process.

The operator uses a reconciler-interface package to standardize the interaction with the Kubernetes API, enabling each controller to implement its own reconciliation logic while being managed under the same umbrella.

4.6 Integration Points and APIs

The key APIs and endpoints provided by Nephio for integration facilitate seamless connectivity and interaction with other systems and services. They enable a broad range of functionalities, from orchestrating complex workflows to enhancing interoperability with external systems.

The end result is a flexible and open platform that supports the customization and extension of Nephio's capabilities to meet diverse operational needs, which is crucial for developers and operators who want to leverage the platform's full potential in automating and managing network functions and infrastructure.

Key APIs and Endpoints Provided by Nephio for Integration

Nephio provides a range of APIs and endpoints to facilitate integration and interaction with its system. These APIs and endpoints are designed to offer robust functionality and ease of use for developers working with Nephio.

Here are some key APIs and endpoints typically provided by a platform like Nephio:

- **Authentication and Authorization API:** This API manages user authentication and authorizes access to various resources. It typically involves endpoints for login, logout, token generation, and access control.
- **Resource Management API:** These APIs are crucial for the management of resources within the Nephio ecosystem. They may include endpoints for creating, updating, deleting, and retrieving resources.
- **Deployment API:** This set of APIs allows users to deploy applications or configurations. Endpoints may include those for initiating deployments, monitoring progress, and managing deployment configurations.
- **Monitoring and Logging API:** For gathering and analyzing logs and metrics, these APIs are essential. They typically include endpoints for retrieving logs, setting up monitoring parameters, and extracting performance metrics.
- **Networking API:** This would include endpoints for managing network configurations, such as setting up network policies, managing IPs, and configuring network-related aspects of the resources.
- **Storage API:** These APIs manage data storage and might include endpoints for data upload, download, storage configuration, and data lifecycle management.
- **User and Access Management API:** These APIs manage user accounts and access levels within the Nephio system, including endpoints for user creation, role assignment, and access permissions.
- **Application Lifecycle Management API:** These APIs would be used for managing the lifecycle of applications, including endpoints for application scaling, updating, and health checking.

Best Practices for Integrating with Nephio's APIs

When integrating with Nephio's APIs, it's essential to follow best practices to ensure efficient and reliable integration.

- **Kubernetes compatibility:** Since Nephio leverages Kubernetes, ensure your integration aligns with Kubernetes API standards and practices.
- **CRD usage:** Nephio often uses Custom Resource Definitions (CRDs). Familiarize yourself with these CRDs, their schemas, and how they are used within Nephio.
- **Versioning:** Pay close attention to the versioning of Nephio APIs. As the project evolves, API versions may change, impacting your integration.
- **Nephio-Specific authentication:** Understand the specific authentication and authorization mechanisms Nephio uses, which may differ from standard API integrations.
- **Follow Nephio's release notes:** Stay updated with Nephio's release notes and documentation updates to be aware of any new features, bug fixes, or changes in API behavior.

Potential Challenges and Solutions during Integration

Integrating with Nephio presents unique challenges. Understanding the complexity of Kubernetes itself is a primary hurdle, requiring a deep dive into its architecture and operations. Managing Custom Resource Definitions (CRDs), which are pivotal in Nephio, demands specific expertise due to their customizable and intricate nature.

Keeping pace with the evolving API versions of Nephio necessitates constant vigilance and adaptation, and navigating Nephio-specific authentication and security protocols can be intricate and demand a thorough understanding of their mechanisms. However, these challenges, while significant, can be

mitigated through dedicated learning, active community engagement, and adapting integration strategies to align with Nephio's dynamic ecosystem.

4.7 Configuration and Customization

Next we'll focus on the process of configuring Nephio for specific use cases, looking at how Nephio's design accommodates a wide array of network environments and operational requirements, and allowing for tailored solutions that meet the unique needs of different organizations.

Configuring Nephio for Specific Use Cases

A complete guide to installing and configuring Nephio is beyond the scope of this book, but let's take a high-level look at how it works, and how you would deploy different kinds of artifacts.

The first step is to deploy the Nephio control plane. This process creates all of the various **`CustomResourceDefinition`**s and controllers Nephio needs to do its job. From there, using Nephio for different use cases involves ensuring the appropriate resources are available and using different YAML files to create the appropriate objects.

For example, two of the most obvious use cases for Nephio are creating infrastructure and creating and managing workloads. Let's take a closer look at each of these use cases.

Use Case: Creating Infrastructure

When it comes to infrastructure, installing Nephio creates the resources that you need, so you can simply create the appropriate YAML file for Kubernetes to handle. For example, you might create edge clusters:

```
apiVersion: config.porch.kpt.dev/v1alpha2
kind: PackageVariantSet
metadata:
name: edge-cluster-deploy
spec:
upstream:
repo: nephio-packages
```

```
package: edge-cluster
revision: v9
targets:
- repositories:
- name: mgmt
packageNames
- edgecluster1
- edgecluster2
- edgecluster3
- edgecluster4
template:
annotations:
approval.nephio.org/policy: initial
pipeline:
mutators:
- image: gcr.io/kpt-fn/set-labels:v0.2.0
configMap:
nephio.org/site-type: edge
nephio.org/region: us-west1
```

This YAML snippet defines a **`PackageVariantSet`** for use with Nephio, specifically targeting the deployment of an edge cluster configuration. Let's break down the components of this snippet to understand its purpose and functionality within the Nephio ecosystem:

- **`apiVersion:`** Specifies the API version of the resource definition. Here we're specifying a specific version of the API provided by the **`kpt`** Kubernetes package management tool, which is integrated with Nephio for managing configurations.
- **`kind:`** Identifies the type of resource being defined. **`PackageVariantSet`** refers to a set of package variants, which are different configurations or deployments of a package. This is used to manage multiple configurations from a single source package.
- **`metadata:`** Contains metadata about the resource, such as its name. In this case, the resource is named **`edge-cluster-deploy`**.

- **`spec:`** The specification section defines the desired state and behavior of the resource.
- **`upstream:`** Defines the source package that this variant set is based on. It specifies a **`repo`** named **`nephio-packages`**, the **`package`** name as **`edge-cluster`**, and a specific **`revision`** of that package, **`v9.`**
- **`targets:`** Lists the target configurations or deployments derived from the upstream package. Each target can have its own customizations.
- **`repositories:`** Specifies the repositories where the package variants will be applied. Here, a single repository named **`mgmt`** is listed.
- **`packageNames:`** This example creates 4 package variants named **`edgecluster1, edgecluster2, edgecluster3`**, and **`edgecluster4`**. The package variants will then create the edge cluster configurations.
- **`template:`** Defines template customizations applied to each package variant.
- **`annotations:`** Metadata annotations added to each package variant. Here, an approval policy **`approval.nephio.org/policy`** is set to **`initial`**, which might be used for governance or workflow controls within Nephio.
- **`pipeline:`** Specifies processing steps (mutators) to modify the package variants. A mutator named **`set-labels`** is used, with its image specified as **`gcr.io/kpt-fn/set-labels:v0.2.0.`** This mutator applies labels to the package variants.
- **`configMap:`** Defines key-value pairs to be used as labels. In this case, **`nephio.org/site-type`** is set to **`edge`**, and **`nephio.org/region`** is set to **`us-west1`**, labeling the clusters as edge sites in the US West 1 region.

This configuration enables Nephio to deploy multiple edge clusters with specific annotations and labels, leveraging a single upstream package as the

template. It demonstrates how Nephio can manage complex, multi-cluster deployments in a scalable and customizable manner.

Use Case: Creating and Managing Workloads

On the other hand, when you configure Nephio for workloads, you need to ensure that any necessary resources have been deployed. For example, one common use case for Nephio is to run 5G using the Free5gc project, which involves several steps:

1. Create the appropriate infrastructure. For example, you might create a regional cluster and multiple edge clusters to run the actual workloads.
2. Clone the appropriate "blueprint" to the Nephio blueprint repository using the CLI or the Nephio UI.
3. Use the blueprint to create a Deployment.
4. Deploy the Free5gc operator to all of the workload clusters. Note that you can also do this via a YAML file, as in:

```
apiVersion: config.porch.kpt.dev/v1alpha2
kind: PackageVariantSet
metadata:
name: free5gc-operator
spec:
upstream:
repo: free5gc-packages
package: free5gc-operator
revision: v5
targets:
- objectSelector:
apiVersion: infra.nephio.org/v1alpha1
kind: WorkloadCluster
template:
annotations:
approval.nephio.org/policy: initial
```

1) Deploy the UPF network function to the edge clusters:

```
apiVersion: config.porch.kpt.dev/v1alpha2
kind: PackageVariantSet
metadata:
name: edge-free5gc-upf
spec:
upstream:
repo: free5gc-packages
package: pkg-example-upf-bp
revision: v5
targets:
- objectSelector:
apiVersion: infra.nephio.org/v1alpha1
kind: WorkloadCluster
matchLabels:
nephio.org/site-type: edge
template:
downstream:
package: free5gc-upf
annotations:
approval.nephio.org/policy: initial
injectors:
- nameExpr: target.name
```

2) Deploy the AMF network function to the regional cluster:

```
apiVersion: config.porch.kpt.dev/v1alpha2
kind: PackageVariantSet
metadata:
name: regional-free5gc-amf
spec:
upstream:
repo: free5gc-packages
package: pkg-example-amf-bp
revision: v5
targets:
- objectSelector:
apiVersion: infra.nephio.org/v1alpha1
kind: WorkloadCluster
matchLabels:
```

```
nephio.org/site-type: regional
template:
downstream:
package: free5gc-amf
annotations:
approval.nephio.org/policy: initial
injectors:
- nameExpr: target.name
```

3) Deploy the SMF network function to the regional cluster:

```
apiVersion: config.porch.kpt.dev/v1alpha2
kind: PackageVariantSet
metadata:
name: regional-free5gc-smf
spec:
upstream:
repo: free5gc-packages
package: pkg-example-smf-bp
revision: v5
targets:
- objectSelector:
apiVersion: infra.nephio.org/v1alpha1
kind: WorkloadCluster
matchLabels:
nephio.org/site-type: regional
template:
downstream:
package: free5gc-smf
annotations:
approval.nephio.org/policy: initial
injectors:
- nameExpr: target.name
```

4) Register the UE as a subscriber using the free5gc web UI. Start by opening a port:

```
ssh @ \
-L 5000:localhost:5000 \
kubectl \
```

```
--kubeconfig regional-kubeconfig \
port-forward \
--namespace=free5gc-cp \
svc/webui-service 5000
```

From there you can simply open your browser to **`http://localhost:5000`** and create the test subscriber.

5) Now you can create the **`UERANSIM`**:

```
apiVersion: config.porch.kpt.dev/v1alpha1
kind: PackageVariant
metadata:
name: edge01-ueransim
spec:
upstream:
repo: nephio-example-packages
package: ueransim
revision: v3
downstream:
repo: edge01
package: ueransim
annotations:
approval.nephio.org/policy: initial
injectors:
- name: edge01
```

Note that in each of these cases, it's a matter of ensuring that the appropriate resources are available, then using the PackageVariant to deploy the object.

Extending and Customizing Nephio for Product-Specific Needs

Extending and customizing Nephio for product-specific needs involves several key steps:

- **Understand the Base Architecture:** Start with a solid understanding of Nephio and Kubernetes, which forms its base.
- **Identify Customization Needs:** Determine what specific aspects

of your product require customization in Nephio.

- **Develop Custom CRDs:** Create or modify Custom Resource Definitions to tailor Nephio to your product's requirements.
- **Implement Custom Controllers:** If necessary, develop custom controllers to manage the lifecycle of your custom resources.
- **Integrate with Existing Systems:** Ensure that Nephio's customization aligns seamlessly with your existing systems and workflows.
- **Test Thoroughly:** Rigorously test the customizations in a controlled environment to ensure stability and performance.
- **Monitor and Update:** Continuously monitor the performance and update your customizations as needed, especially when Nephio releases new updates.

4.8 Security Considerations

It's crucial to recognize the importance of security in the orchestration and management of network functions and infrastructure, so let’s look at foundational security principles and practices that are integral to Nephio's design and operation.

This section is dedicated to exploring the various strategies and mechanisms implemented within Nephio to safeguard against potential vulnerabilities and threats, ensuring that integrations with Nephio are not only efficient and scalable but also secure.

Ensuring Secure Integrations with Nephio

Ensuring secure integrations with Nephio involves several key practices:

- **Use Secure Authentication:** Implement robust authentication mechanisms such as OAuth, JWTs, or API keys.
- **Manage Permissions Carefully:** Apply the principle of least privilege, granting only necessary permissions.

- **Encrypt Data:** Use encryption for data in transit and at rest.
- **Regularly Update and Patch:** Keep Nephio and its dependencies up-to-date to protect against vulnerabilities.
- **Monitor Access and Activity:** Implement logging and monitoring to track usage and detect suspicious activities.
- **Conduct Security Audits:** Regularly audit your integration for vulnerabilities and compliance with security policies.
- **Follow Kubernetes Security Best Practices:** Since Nephio is Kubernetes-based, adhere to Kubernetes-specific security guidelines.

Handling Authentication, Authorization, and Encryption

Handling authentication, authorization, and encryption in the context of Nephio, particularly given its Kubernetes-based architecture, involves several important considerations:

- **Authentication:** This is about verifying user identities. In Nephio, this could involve integrating with Kubernetes authentication mechanisms, using service accounts, or implementing external authentication systems like OAuth or OpenID Connect.
- **Authorization:** Once a user is authenticated, authorization determines their permissions. In Kubernetes, this typically involves Role-Based Access Control (RBAC), where roles are defined with specific permissions and assigned to users or groups.
- **Encryption:** Ensuring data security both in transit and at rest is crucial. For data in transit, HTTPS and TLS can be used. For data at rest, consider Kubernetes **`Secret`**s, which are encrypted by default in recent versions, or integrate with external secrets management systems.

Each of these aspects is critical for maintaining security and integrity in Nephio integrations, and should be carefully planned and implemented in

line with best practices and organizational policies.

4.9 Performance and Scalability

Now let's look at how Nephio is engineered to meet and exceed the demanding requirements of modern network infrastructures, focusing on efficiency, responsiveness, and the ability to scale dynamically. Specifically, we'll talk about the bench marks and metrics that define Nephio's performance, offering insights into the platform's capabilities to handle varying loads and complexities with minimal latency and maximum throughput.

Understanding Nephio's Performance Benchmarks

Performance benchmarks for a project like Nephio typically include:

- **Deployment Speed:** Time taken to deploy network functions across various environments, especially in edge computing scenarios.
- **Scalability:** Ability to manage a growing number of network functions and nodes without significant degradation in performance.
- **Resource Efficiency:** Optimization of computing, storage, and network resources, crucial in edge computing where resources may be limited.
- **Reliability and Uptime:** Consistency in performance and minimal downtime, which is critical for network functions.
- **Latency:** The amount of time between the request and the start of the response. Especially important in edge computing, where reducing latency is often a key objective.
- **Throughput:** The volume of data that can be processed within a given time frame.

Tips for Ensuring Scalability when Integrating Nephio into Large-Scale Products

Integrating Nephio into large-scale products, especially considering its focus on Kubernetes-based automation for network functions, requires careful planning and execution to ensure scalability. Here are some tips to help ensure scalability when integrating Nephio:

- **Understand Nephio's architecture:** Familiarize yourself with Nephio's components, how they interact, and their dependencies. This understanding is crucial for effectively scaling the system.
- **Leverage Kubernetes' scalability features:** Since Nephio is built on Kubernetes, take full advantage of Kubernetes' native scalability features, such as horizontal pod autoscaling, cluster autoscaling, and efficient load balancing.
- **Optimize resource allocation:** Carefully manage resources such as CPU, memory, and storage. Use resource requests and limits in Kubernetes to ensure that applications get the resources they need without over-provisioning.
- **Implement efficient networking:** Network performance is crucial in large-scale deployments. Optimize network configurations and consider using network policies for efficient traffic management.
- **Use state-of-the-art hardware:** For physical deployments, ensure that the underlying hardware can support the scale. This includes sufficient compute power, memory, and network bandwidth.
- **Automate where possible:** Automation is key in managing large-scale deployments. Automate routine tasks such as deployments, updates, and monitoring to reduce the risk of human error and improve efficiency.
- **Monitor performance and reliability:** Implement comprehensive monitoring and logging to track the performance and health of the system. This data is crucial for making informed decisions about scaling.

- **Plan for High Availability and Disaster Recovery:** Ensure that your deployment is resilient to failures. Implement strategies such as multi-zone deployments, data replication, and regular backups.
- **Test at scale:** Before full integration, test Nephio in a scaled environment to identify and address potential bottlenecks or issues.
- **Stay updated with Nephio developments:** The Nephio project, like any software, will evolve over time. Stay updated with the latest releases and community best practices to leverage new features and improvements.
- **Engage with the community:** Participate in Nephio's community forums, discussions, and events. Insights from the community can be invaluable in understanding how to scale effectively with Nephio.
- **Customize for your needs:** Tailor Nephio's components and configurations to suit your specific use case. Not all default settings may be optimal for your scenario.

Remember, scalability is not just about handling more load; it's also about maintaining performance, reliability, and manageability as the system grows.

CHAPTER 5
NEPHIO COMMUNITY AND SIGS

Now that you have a general idea of what Nephio is and how it works, you're ready to dive into the nitty gritty and build your own applications and integrations. For the information you'll need, you'll want to visit the Nephio site (http://nephio.org), which includes links to all of the information you'll need, including:

- A list of Frequently Asked Questions (https://nephio.org/frequently-asked-questions/), which includes both practical and technical issues.
- The Nephio Wiki (https://wiki.nephio.org/pages/viewpage.action?pageId=28510591), which includes links to community and technical resources.
- The Overview of Nephio (https://wiki.nephio.org/display/HOME/Overview+of+Nephio), which provides a technical explanation much like this book.
- Learning with Nephio R1 (https://wiki.nephio.org/display/HOME/Learning+with+Nephio+R1), a series of hands-on tutorials that provide you with a good handle on how Nephio actually works.

Nephio prides itself on being an open source and welcoming community, which means that there are plenty of ways for you to get involved.

- The Technical Steering Committee (TSC) (https://wiki.nephio.org/pages/viewpage.action?pageId=360619) meets every other Thursday at 9am PT.
- You can join a number of special interest groups, including SIG

Release (https://wiki.nephio.org/display/HOME/SIG+Release), SIG Network Architecture (https://wiki.nephio.org/display/HOME/SIG+Network+Architecture), SIG Security (https://wiki.nephio.org/display/HOME/SIG+Security), and SIG Automation (https://wiki.nephio.org/display/HOME/SIG+Automation).

- If you'd like to communicate with the rest of the community, you can use the Nephio Slack. To join, send a note to sig-release@lists.nephio.org and administrators will generate an invitation.

AUTHOR BIO AND PICTURE

Sandeep Sharma is the Principal Architect of Aarna Networks, a SaaS solutions provider that provides zero-touch edge and 5G service orchestration and management services. He is very active in the open source community helping to lead the Nephio, EMCO, and Akraino projects. Prior to Aarna, Sandeeep held leadership roles at Western Digital, Virident Systems, Alcatel-Lucent, and NetDevices. He studied Computer Science at the Vidya Vardhaka College of Engineering in Mysuru, India.

Nick Chase is the Senior Director of Product Management and heads up the AI practice at CloudGeometry, which provides expert systems integration to build, optimize and run your cloud software and operations. He specializes in creating technical content that makes complicated topics understandable, and in finding new ways to use emerging technology. Prior to CloudGeometry, he specialized in cloud computing at Mirantis, and was previously an instructor for Oracle Education. He is the author of more than a dozen books on technical topics.