From NIMITZ to NETPILA: The Evolution of Production-Scale Container Network

Jiaqi Gao, Chao Qin, Sheng Cheng, Jiamin Cao, Guodong Yang, Zhenyu Zhang, Shuhong Zhu,
Ennan Zhai, Dennis Cai
Alibaba Group

Abstract

This paper describes two generations of production container networks deployed for over five years. Our first-generation network, NIMITZ, used VxLAN for overlay–underlay mapping, providing flexibility and scalability from 2019–2022. However, with the rise of large-scale services (*e.g.,* AI training and inference), NIMITZ hit the C100K problem: beyond $O(100K)$ containers, address mapping overhead and complex packet processing caused significant performance degradation. To overcome this, we built NETPILA, a second-generation design that uses simple, practical IPv6 addressing to remove VxLAN encapsulation and overlay–underlay mapping tables. By embedding container addresses in the 128-bit IPv6 space, NETPILA integrates the container network with the physical network, reducing packet-processing complexity and improving scalability. NETPILA now supports millions of containers per tenant for production AI training and inference. We present the design, lessons, and deployment results of both generations.

CCS Concepts

• **Networks → Network architectures**; **Data center networks**; **Network management**.

ACM Reference Format:
Jiaqi Gao, Chao Qin, Sheng Cheng, Jiamin Cao, Guodong Yang, Zhenyu Zhang, Shuhong Zhu,, Ennan Zhai, Dennis Cai, *Alibaba Group*. 2026. From NIMITZ to NETPILA: The Evolution of Production-Scale Container Network. In *ACM SIGCOMM 2026 Conference (SIGCOMM '26), August 17–21, 2026, Denver, CO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3789240.3829203

1 Introduction

The widespread adoption of cloud-native applications and microservices has positioned containers as a leading method for deploying services with highly elastic demand. Containers provide a lightweight and uniform deployment environment, enhancing services with flexibility and scalability. Central to this deployment model is the container network, which ensures connectivity between containers and is mainly responsible for delivering service traffic efficiently.

Our first-generation container network: NIMITZ (§3). As one of the largest online cloud providers, we designed and developed our first-generation container network, NIMITZ, based on the widely-used virtualization technique, Virtual Extensible LAN (VxLAN) [2]. NIMITZ maps overlay networks to underlay networks, *i.e.,* container $A \rightarrow$ base network (or called physical network) $\rightarrow$ container B, by encapsulating container addresses within VxLAN headers and embedding underlay network addresses within these headers. This mechanism allows traffic between containers to traverse the underlay network seamlessly. However, designing an effective overlay-underlay mapping table presented challenges due to the diverse traffic types in container networks, including inter-container communication, inter-node traffic, and cross-region traffic. A straightforward address mapping approach would cause the packet processing entries to grow exponentially as the container network scales (*e.g.,* generating millions of entries for an $O(1000)$-scale container network), undermining scalability and maintainability.

To address the challenge, NIMITZ proposed a five-stage packet processing pipeline for overlay-underlay mapping. This pipeline modularizes key packet processing operations—such as VxLAN encapsulation/decapsulation, Network Address Translation (NAT), and forwarding based on sender type and destination addresses—into pipelined stages. Implemented within the Open vSwitch (OVS) on each node, this design not only simplifies table entry maintenance, but also reduces the complexity of packet processing tables by three orders of magnitude, with complexity $O(\textit{\# of service}) + O(\textit{\# of container}) + O(\textit{\# of node})$. We further proposed key optimizations, including hardware offloading for active table entries and classless inter-domain routing (CIDR), to enhance performance and scalability.

The C100K problem (§4). Despite its initial success in supporting dozens of services and thousands of tenants over multiple years, NIMITZ began to encounter scalability limitations with the emergence of large-scale services such as AI training and inference. Specifically, for a tenant's container network exceeding $O(100K)$ containers—a scenario we define as *the C100K problem*—NIMITZ experienced significant performance degradation. Issues included prolonged delays in container warm-up and clean-up, performance fluctuations due to frequent swapping of table entries offloaded to hardware, and overhead from VxLAN encapsulation and decapsulation. These performance bottlenecks were primarily caused by the increased overhead of maintaining overlay-underlay address mappings within the VxLAN approach as the network scales. For example, at the scale of 100K containers, a 10% container increase/removal in scale requires NIMITZ to handle over 150k mapping sync requests, taking tens of minutes, which is unacceptable to our tenants. In traditional VM networks, we do not encounter this issue because the number of VMs per tenant is one or two orders of magnitude smaller than container networks.

Our second-generation container network: NETPILA (§6). Motivated by the C100K problem, we built our second-generation container network, NETPILA, to address the limitations of NIMITZ.

This work is licensed under a Creative Commons Attribution 4.0 International License.
SIGCOMM '26, , Denver, CO, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2467-1/26/08
https://doi.org/10.1145/3789240.3829203

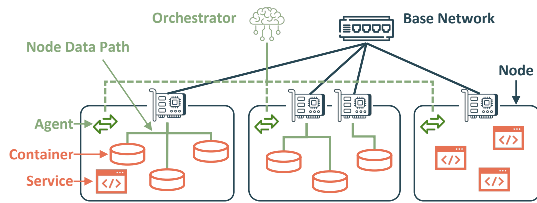


Figure 1: A typical container network architecture.

NetPila resolves the C100K problem through a simple yet effective approach by adopting IPv6 for container addressing. By appending the container address after the physical node address within the 128-bit IPv6 address, the container network becomes an extension of the base network. Since NetPila no longer uses VxLAN technology, it eliminates the overhead introduced by VxLAN encapsulation and decapsulation. Furthermore, by completely eliminating the overlay-underlay mapping, the number of table entries is significantly reduced, thus eliminating the performance fluctuations caused by frequent table entry swaps in the NICs constrained by limited hardware resources. Compared with Nimitz, NetPila now can efficiently support millions of containers for a single tenant. NetPila thus offers a scalable and efficient solution—**more importantly, it is simple**—for modern production-scale container networks.

NetPila brings new challenges and opportunities. NetPila's address encoding solution exposes the node address to the tenants, which introduces privacy risks. NetPila addresses this problem via a symmetric key-based IP encryption solution that hides the node IPs and tenant IDs from containers with minimal performance impact (§6.2). We also discuss why NetPila's IPv6-encoding solution is specific to the container network in §6.3.

In this paper, we share the design details and evolution of our two generations of systems (Nimitz and NetPila), the insights behind their design, and the results of their production-scale deployment. Nimitz has been in production since 2019, and NetPila has served production container networks since 2022. One representative NetPila deployment contains 9,205 nodes and 589,120 containers, while our production evaluation includes jobs up to 25K nodes and our design supports millions of containers per tenant.

Ethics. This work does not raise any ethical issues.

2 Background

Container is an innovative virtualization technology that has become a popular method for building large-scale, elastic services. Compared with VM services (IaaS), Container-as-a-Service (CaaS) offers much more virtual resources (due to its lightweight) and easily scaling deployment (due to its high elasticity). The tenants of our CaaS include e-commerce, search engine, and advertisement. They adopt their services (*e.g.,* AI model training and inference, and big data analysis) on containers for the large scale and high elasticity needs.

Container networks. Container network (shown in Figure 1), is a core component of CaaS. Built upon the base network (*i.e.,* physical network), container network provides essential services like IP addressing, connectivity, and container monitoring deployed on the nodes (*i.e.,* hosts). A typical container network consists of the following components:

- **Orchestrator.** A centralized controller plays a control plane role. The Orchestrator receives requests (*e.g.,* adding or removing containers) from the service, and then translates these requests into container network state changes and notifies the corresponding agents.
- **Agent.** Each node has an agent. The agent maintains many tables (detailed in §3) that work together for packet processing (including packet forwarding, encapsulation, and decapsulation); moreover, an agent is also responsible for updating these tables according to the information received from the Orchestrator.

Different tenants' services (including AI, big data and search engines) impose varying requirements on container networks. In our experience, our tenants primarily focus on the following three dimensions:

- **Elasticity.** The container's lightweight nature poses a higher demand for network elasticity. Elasticity measures the throughput for container creation and teardown. Usually, a 10% container scale-up or down event is expected to finish within minutes.
- **Performance.** The container network runs atop the base network. The performance overhead of container networks (*e.g.,* latency and throughput) should be minimized so that the services run as if they run on the base network.
- **Stability.** Most importantly, elasticity and performance should be offered stably. The performance should remain stable regardless of its scale.

Realizing virtualization via VxLAN. In principle, container network employs virtualization technology for multiple containers to share the node and OS environments to achieve tenant-level isolation. Mainstream cloud providers (*e.g.,* AWS and Alibaba) employ VxLAN [2] to realize their container networks. Figure 2 shows the VxLAN header format. Operating at Layer 2 over Layer 3 networks, VxLAN encapsulates Ethernet frames within UDP packets with pre-defined UDP destination port 4789. By using a 24-bit segment ID (VxLAN Network Identifier, or VNI), it supports up to 16 million unique networks. We define *VxLAN encapsulation* as encapsulating the original L2 frame with VxLAN, UDP, Outer IP, and Outer MAC (as shown in Figure 2).

Figure 2: VxLAN encapsulation format.

3 Nimitz: VxLAN-based Network

Since 2019, we built our first-generation container network codenamed Nimitz. Nimitz was used to host our in-production services such as e-commerce, search engines, and AI.

In line with mainstream container networking solutions, Nimitz also used VxLAN as the virtualization approach to construct our container network. Within Nimitz, each (physical) node has a

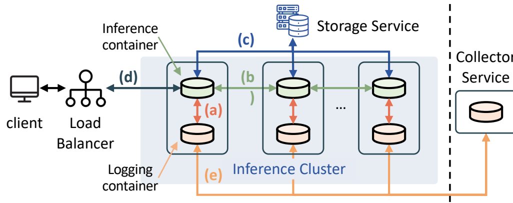


Figure 3: The five traffic types in LLM inference workload for NIMITZ container network.

unique IP address (called underlay IP), and every container on that node has its overlay IP address. The two address spaces are independent of each other, which provides maximized flexibility and security. The Agent (on each node) in NIMITZ contains three key components: (1) VxLAN function, responsible for forwarding the packets encapsulated by VxLAN; (2) Open vSwitch (OVS) [25], responsible for packet processing and table updating as a virtual switch role; and (3) Gateway (virtual device), responsible for network address translation (NAT).

3.1 NIMITZ Container Network

We now use a real-world example (LLM inference) to demonstrate how NIMITZ supports our service. Figure 3 illustrates the workflow for a tenant A's LLM inference job. The inference cluster (*i.e.,* a container network) is exposed to clients via a Service IP. **We define a *cluster* (or a container cluster) as a container network belonging to a tenant.** The client's request, targeted at the Service IP, is load-balanced to one of the inference containers through (d). The inference container then loads the latest AI model from the storage service hosted on the underlay network (c) and performs cross-node inference (b). A logging container deployed by the same tenant periodically collects the monitoring data (a) and sends it to a collector container deployed in another region for failure resiliency (e). **(a)-(e) in Figure 3 cover all possible types of traffic in NIMITZ.**

Service traffic types. In general, the traffic can be grouped into three categories: east-west traffic ((a) and (b)), north-south traffic ((c) and (d)), and cross-region traffic (e). Figure 4 presents the traffic (a)-(e) in a more detailed way. Note that traffic (a)-(e) in Figure 3 correspond to traffic (a)-(e) in Figure 4, respectively.

- *East-west traffic, i.e.,* (a) and (b), denotes the communication between the containers for the same tenant such as the communication between the inference and logging container in the same node and between inference containers on different nodes in Figure 3.
- *North-south traffic* denotes traffic that connects the overlay and underlay network, *i.e.,* traffic (c) and (d) in Figure 4. Since overlay and underlay are two independent address spaces, the NAT technique is needed for these two types. NIMITZ uses virtual gateway devices (*i.e.,* Gateway in Figure 4) deployed on each node to handle such traffic.
- *Cross-region traffic, i.e.,* (e), is mainly used to ensure high availability. An Orchestrator is deployed in each region to orchestrate the local containers. Cross-region (both east-west and north-south) communication requires maintaining the global overlay-underlay mapping. Synchronizing the mapping entries across regions is slow. Instead, NIMITZ deploys an Edge Switch device on the edge of each region. Each Edge Switch maintains a copy of the overlay-underlay mapping in its region. All Edge Switches know each other's overlay address space. Upon receiving cross-region traffic, the Edge Switch examines the packet's overlay IP and finds the destination region Edge Switch. Then, Edge Switch rewrites the outer dst IP in the VxLAN header based on the destination Edge Switch's IP. The destination Edge Switch then looks up the overlay-underlay mapping in its region, rewrites the VxLAN header again, and sends the packet to the destination.

Traffic processing via overlay-underlay mapping on OVS. How does NIMITZ forward or process different types of traffic in reality? This is the core of NIMITZ design. Consider traffic (b) in Figure 4. The packet path (from n_3 to n_2) is: container $c_3 \rightarrow$ OVS $o_2 \rightarrow$ VxLAN on $n_3 \rightarrow$ NIC on $n_3 \rightarrow$ NIC on $n_2 \rightarrow$ VxLAN on $n_2 \rightarrow$ OVS $o_1 \rightarrow c_2$. The key packet processing happens on the OVS, which performs an overlay-underlay mapping: overlay (c_3) $\rightarrow$ OVS $o_2 \rightarrow$ underlay (NIC on $n_3 \rightarrow$ base network $\rightarrow$ NIC on n_2) $\rightarrow$ OVS $o_1 \rightarrow$ overlay (c_2). We call the above packet processing as *overlay-underlay mapping*. NIMITZ realizes the overlay-underlay mapping through VxLAN technique. Thus, the key design of NIMITZ has focused on how OVS implements the overlay-underlay mapping to process packets for the above-mentioned five types of traffic in our production.

Our design: Five-stage packet processing pipeline on OVS. A straightforward way (for small-scale providers) is to enumerate all overlay-underlay mappings, *i.e.,* mapping a given packet from its header to the destination in just one stage, within a huge flow table, and install this table in the OVS. However, as the scale of container networks increases, the memory of OVS would explode; moreover, such a huge table is hard to maintain by our engineers or update if any traffic design changes. NIMITZ proposes a five-stage packet processing pipeline solution on OVS to process different types of traffic. Figure 5 shows a typical five-stage packet processing pipeline design on NIMITZ's OVS.

The intuition of our solution modularizes key packet processing operations—such as VxLAN encap/decap, Network Address Translation (NAT), and forwarding based on sender type and destination addresses—into pipelined stages. Such a design not only simplifies table entry maintenance but also greatly reduces the complexity of packet processing tables.

- *Stage 1: VxLAN decapsulation.* For a given traffic, the OVS identifies the packet sender. As shown in Figure 5, the sender can be VxLAN, Gateway and Container. If the sender is a VxLAN, the OVS decapsulates the packet (including VxLAN header, UDP and outer MAC and IP); otherwise, the packet directly enters stage 2.
- *Stage 2: VNI register assignment.* For the given packet p, OVS assigns its VNI register according to the sender type of p. (1) for VxLAN, the OVS fills the register according to the VNI value in the VxLAN header of p; (2) for Gateway, the OVS assigns a

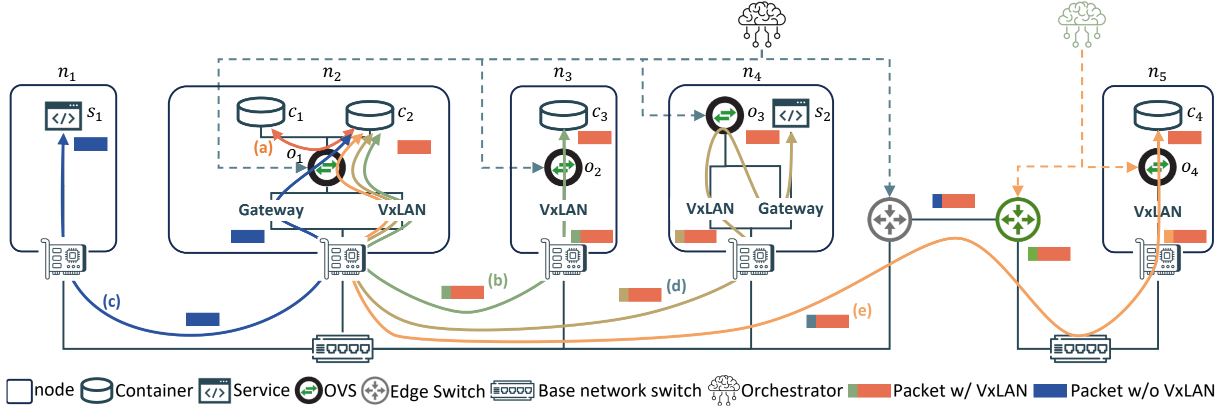


Figure 4: The Nimitz traffic types: (a) east-west traffic within a node, (b) east-west traffic across nodes, (c) container visits service/application deployed in underlay, (d) service/application deployed in underlay visits container, (e) cross-region traffic. The five types of traffic are bidirectional.

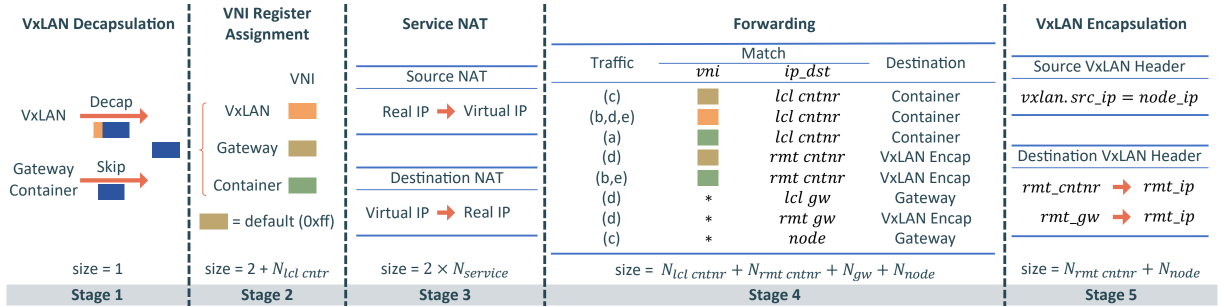


Figure 5: The Nimitz five-stage packet processing pipeline, *i.e.,* Nimitz dataplane processing. *lcl* **- Local,** *rmt* **- Remote,** *cntnr* **- Container,** *gw* **- Gateway. The** ***Traffic*** **column listed in the Forwarding stage aligns with the traffic marked in Figure 4. The size of each table is marked at the bottom of each stage. $*$ in stage 4 means a wildcard that matches all VNI values.**

default value 0xFF to the register; and (3) for the container, the OVS fills the register with the sender's tenant ID.

- *Stage 3: NAT.* If p belongs to north-south traffic, p should be processed in this stage; otherwise, p misses this stage. Stage 3 includes providing Destination NAT (DNAT) for traffic from the container to the service, and Source NAT (SNAT) for the reversed direction. Because the virtual IP and real IP space do not overlap, Nimitz places the two tables side-by-side and looks them up sequentially.
- *Stage 4: Forwarding.* In the forwarding stage, the OVS examines the current VNI register (filled in stage 2) and p's destination IP to decide which type of traffic p belongs to, and then sends p to the corresponding destination (as shown in Figure 5). Note that packets of one traffic type can traverse OVS in both forward and reverse directions at multiple nodes on the path, so they can match multiple entries in the forwarding stage.
- *Stage 5: VxLAN encapsulation.* If the destination of p is a VxLAN Encap in stage 4, as shown in Figure 5, p enters this stage; otherwise p skips stage 5. The OVS encapsulates p with a VxLAN header, UDP, outer IP (the node IP and a remote IP as the src IP and dst IP).

We detail traffic (a)-(c) processing based on Figure 4 and Figure 5 examples. See Appendix A for traffic (d) details.

Traffic (a). In Figure 4, traffic (a) happens between two containers on the same node. For example, c_1 wants to access c_2; thus c_1 generates a packet p with the destination IP of c_2. The OVS o_1 is responsible for routing p. Note that the traffic between local containers does not have a VxLAN header. As shown in Figure 5, o_1 first identifies that p's sender is c_1 (*i.e.,* Container) in stage 1; thus, o_1 fills its VNI register with c_1's tenant ID in stage 2. p misses stage 3, because (a) is east-west traffic. Since the destination IP of p is a local container c_2 and the VNI register is a tenant ID, according to the forwarding table (*i.e.,* stage 4 in Figure 5), o_1 forwards p to Container (hitting the third entry in stage 4 of o_1). p skips stage 5.

Traffic (b). Traffic (b) happens between two containers across nodes. In Figure 4 example, a typical traffic (b) is: container $c_3 \rightarrow$ OVS $o_2 \rightarrow$ VxLAN on $n_3 \rightarrow$ NIC on $n_3 \rightarrow$ NIC on $n_2 \rightarrow$ VxLAN on $n_2 \rightarrow$ OVS $o_1 \rightarrow c_2$. c_3 wants to access c_2, so c_3 generates a packet p whose destination IP is a remote container (*i.e.,* c_2). When o_2 receives p from c_3, o_2 runs the packet processing pipeline. Because p's sender, c_3, is a Container (not VxLAN), o_2 fills its VNI register with c_3's tenant ID in stage 2. Since traffic (b) is east-west traffic,

p misses stage 3. In stage 4, because the destination IP of p is a remote container and the VNI register is a tenant ID, p hits the fifth entry in the forwarding table; thus, p should be encapsulated with VxLAN according to stage 5, and the outer dst IP of VxLAN encapsulation is n_2's IP (*i.e.,* the IP of NIC on n_2).

o_2 sends p to the VxLAN on n_3. p is allowed by the VxLAN and forwarded to the NIC on n_3, because p contains a VxLAN header. Then, p is forwarded to the NIC on n_2. The NIC on n_2 finds that the destination IP of p (*i.e.,* the outer dst IP) is the address of n_2, so that the NIC forwards p to the VxLAN on n_2; subsequently, because p has a VxLAN header, the VxLAN hands it over to o_1 for processing.

Because p's sender is a VxLAN, o_1 decapsulates p, extracts the VNI value from the VxLAN header of p, and puts this value in the VNI register of o_1 in stage 1&2. p misses stage 3 because (b) is east-west traffic. In stage 4, because the destination IP of p has been a local container (*i.e.,* c_2) and the VNI register's value is the VNI value of the decapsulated header, p hits the second entry in the forwarding table of o_1 (see Figure 5); thus, p should be forwarded to c_2, because the Destination in stage 4 is Container. p skips stage 5.

Traffic (c). Traffic (c) happens when a container (running a service) wants to access another service deployed in an underlay network. Recall the (c) case shown in Figure 3 where an inference service (a container) wants to call a storage service (another service). For example in Figure 4, a traffic (c) is: $c_2 \rightarrow o_1 \rightarrow$ Gateway on $n_2 \rightarrow$ NIC on $n_2 \rightarrow$ NIC on $n_1 \rightarrow$ service s_1. In this case, c_2 wants to access s_1; thus, c_2 generates a packet p with the destination IP of s_1 and then sends p to o_1. o_1 processes p as shown in Figure 5. Because p's sender, c_2, is a container, o_1 fills its VNI register with c_2's tenant ID in stage 2. Because (c) is a north-south traffic, o_1 runs SNAT to translate p's src IP (*i.e.,* c_2's IP) into a virtual IP (*i.e.,* service IP running on c_2) in stage 3. In stage 4, p hits the last entry in the forwarding table of o_1, because p's destination IP is s_1, *i.e.,* node; thus, o_1 sends p to the Gateway on n_2 without any VxLAN encapsulation. Note that the packets of traffic (c) have no VxLAN encapsulation or decapsulation.

3.2 Intuition and Optimizations

The intuition of the five-stage processing solution. A strawman solution is to enumerate every overlay-underlay mapping in a single OVS table and directly map each packet header to its destination. This design is simple for small clusters, but it leads to quadratic growth in table entries as the number of containers, nodes, and services increases. NIMITZ's five-stage pipeline avoids this explosion by decomposing packet processing into reusable stages and using VNI-based classification to separate sender type, service NAT, forwarding, and VxLAN encapsulation. Besides the low table complexity $O(\#\ of\ container)+O(\#\ of\ node)\ +O(\#\ of\ service)$, another intuition of the processing pipeline design is to use VNI to classify the senders of traffic; otherwise, the table size would greatly increase. Furthermore, such a design achieves a balance between efficiency and flexibility. Any change of node or container change (*e.g.,* adding a new node or removing containers) only adds or removes the corresponding entries rather than changing multiple stages due to the "high cohesion and low coupling" stage design.

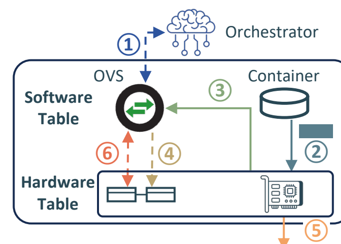


Figure 6: The NIMITZ hardware offloading procedure.

Node Classless Inter-Domain Routing (CIDR) optimization. Our five-stage processing pipeline may still lead to an overwhelming memory problem on OVS. In one of our e-commerce services, the total number of containers exceeds 200k, resulting in nearly 0.5 million entries within each OVS. Such a huge number of entries can lead to severe memory overload on the OVS. To further reduce the memory usage, NIMITZ proposes a CIDR optimization to solve this problem. Containers on the same node are grouped into one or multiple CIDR subnets, each with a single prefix (*e.g.,* 10.0.1.0/29); thus, the OVS only needs to maintain the mapping between overlay CIDRs and underlay node IPs. Such an optimization significantly reduces memory usage of the OVS from $O(container)$ to $O(node)$. Depending on the deployment density, assigning Node CIDR can reduce memory footprint by up to 80×, supporting an 80× larger scale at the same performance. In one of our largest AI container networks, there are more than two million containers deployed on 25k nodes, CIDR reduces the packet processing pipeline's size by 80×.

3.3 NIMITZ Hardware Offloading

By default, the OVS packet processing is implemented in software. Unfortunately, the performance is limited. It may lead to a long lookup time, low forwarding throughput, and high CPU and memory overhead. For example, in our production hosts, 8 CPU cores can only process 48.8 Gbps traffic. Also, a kernel-based OVS implementation incurs high latency overhead for user-space protocols such as RoCE.

We therefore decided to further optimize NIMITZ, *i.e.,* leveraging the hardware to accelerate the packet processing in the OVS. As shown in Figure 6, we dynamically offload *active* entries to the NIC hardware to maximize the performance gain and accommodate the limited hardware resources. All entry updates are sent from Orchestrator to the OVS in Agent (① in Figure 6). When a container establishes a connection, the packet arrives at the NIC (②). If the matched entries have been in the NIC's hardware table, the packet should be processed accordingly; otherwise, the NIC informs the OVS. In this case, the OVS intercepts the packet (③), looks up its (software) tables, groups matched entries into a MegaFlow [26], and installs it to the NIC via the `tc` command (④). `tc` is originally used to configure Traffic Control in the Linux kernel. Our NIC's vendor provides `tc` offloading capability to manage hardware entries.

Figure 7 shows the total number of entries in OVS on each node for one of our production jobs that span 8076 nodes and how many are offloaded to the hardware. There are 11830 entries in the OVS to support the distributed job. The total number of offloaded entries follows a long-tail distribution. The average number is 1.2k, the peak is 8k, and the 95% percentile is 3.3k.

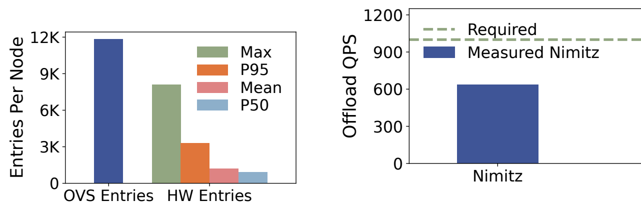


Figure 7: The scales of flow entries of Nimitz.

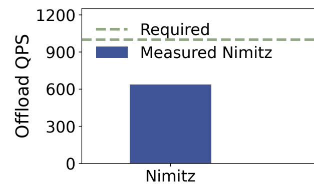


Figure 8: The hardware offload capability of Nimitz and the requirement.

Due to the CIDR optimization, the OVS only intervenes for the first connection to the containers within a CIDR subnet. The OVS is henceforth out of the connection's processing loop. The NIC driver then converts the entries into a NIC internal data structure called Flow Database (FDB). When the NIC receives follow-up packets from the same connection, it looks up the FDB, finds a matched entry, and performs the corresponding action (⑤). The OVS periodically cleans up aged table entries to avoid hardware overflow (⑥). The hardware offloading significantly reduces the CPU cost while improving the packet processing performance, 1 standby CPU can easily handle traffic at 400 Gbps line rate.

Preheating. The above-mentioned offloading channel (③ and ④) has limited throughput. Figure 8 shows a typical setting in our production. Nimitz achieves an offloading throughput of 638 entries per second (EPS); thus, the offloading channel becomes the bottleneck when the large-scale container network receives burst traffic. This happens during our yearly sales events such as Black Friday when tenant traffic hits our e-commerce service and the requirement can be as high as 1000 EPS. The marker labeled "Required" in Figure 8 denotes this peak-event requirement rather than an empirical measurement from Nimitz. Because the container network cannot know the communication pattern beforehand, Nimitz has no way to pre-install any entries in the NIC. We implemented and deployed preheating in Nimitz, especially before major shopping events. By simulating customer traffic with load-generation tools, Nimitz pre-triggers connections and installs corresponding NIC entries before the peak, keeping these entries resident during the event. This makes Nimitz no longer bottlenecked by the offloading QPS limit. However, preheating is only an operational mitigation: it relies on predictable traffic patterns and CIDR-based aggregation, and without aggregation the number of VxLAN rules remains too large to fit stably in hardware.

4 The C100k problem

Nimitz began to encounter scalability problems (called *C100K problem*) with the emergence of new in-production services such as AI training and inference: as the scale of a single tenant grows larger than $O(100K)$ containers **(or $O(10K)$ nodes, thanks to our CIDR optimization)**, Nimitz's VxLAN-based solution can no longer maintain elasticity, stability, and performance. The inter-component communication scales as the cluster size increases. As the scale of nodes grows to 10k, every component in Nimitz is under great pressure and prone to failure. Traditional VM networks rarely encounter the same problem because VMs are heavyweight: each VM carries a full OS and a larger per-instance resource footprint, so tenants typically deploy orders of magnitude fewer VMs than containers for the same workload. The C100K problem is therefore not a Nimitz-specific artifact; any VxLAN-based container network that maintains global overlay-underlay mappings faces similar pressure as container count and churn increase. This section shows three real, *frequently-happened* cases in our production to demonstrate the C100k problem and explain how each component is pressured.

4.1 Low Cluster Elasticity

Long node warm-up time. When we ran our first 25k node cluster, we received complaints from the tenants that there was up to a 20-minute warm-up time between expanding 10% of the scale of the container cluster and the newly expanded nodes being ready to communicate with other containers. After checking the log, our engineers identified that the root cause was Orchestrator `list` API throttling.

In Nimitz, every node's Agent (primarily OVS) requires the latest cluster status information to fulfill the connectivity requirement. When a node joins the container cluster, it queries the Orchestrator for a complete copy of the overlay-underlay mapping, gateway, and node IPs to initiate the packet processing pipeline shown in Figure 5. In the setting, the Orchestrator was implemented atop Kubernetes [13], and the mapping query is implemented using the `list` API. By default, the `list` operation returns 500 nodes' information. Collecting all 25k nodes' information translates to 50 queries. Therefore, the Orchestrator has to process 125k `list` queries to finish initializing 10% of the 25k-sized cluster. Similarly, all other nodes also query the Orchestrator for the newly expanded nodes' information, which translates to 25k more queries to the Orchestrator. In total, the Orchestrator has to handle 150k queries. The Orchestrator oversees all tenants. To avoid being overwhelmed, the Orchestrator throttles the `list` API to 200 QPS per tenant. As a result, the Orchestrator takes at least 12 minutes to finish processing the requests, which is too long and results in a timeout. We mitigated this problem by increasing the API throughput limit and the query batch size. As shown in Figure 9, nevertheless, the total number of messages to finish setting up 10% of the cluster grows exponentially as the scale of the cluster increases; thus, the problem cannot be solved fundamentally despite increasing such limits.

Long node clean up time. When a tenant job finishes, Nimitz must clean up the node quickly to avoid downtime and wasted compute. At scale, however, Nimitz installs many more OVS entries, and OVS rule deletion becomes increasingly slow, eventually dominating node cleanup. To quantify this, we used a production server in a 50K-container cluster and removed OVS entries in 2K batches, measuring the per-batch deletion latency until all entries were cleared. Figure 10 reports the per-batch removal time across different installed-entry counts, and Figure 11 shows the total entries and end-to-end cleanup time versus cluster size. Deleting a 2K batch became 28× slower when OVS held 50K entries, because OVS must search existing rules to find matches and invoke the NIC driver to reclaim hardware-offloaded entries; both costs grow with the number of installed/offloaded rules. As a result, rule removal scales worse than cluster growth and quickly dominates

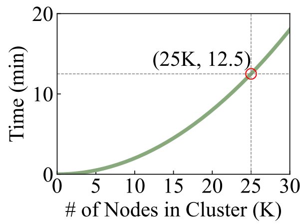


Figure 9: Node initialization time vs. cluster scale when 10% nodes join.

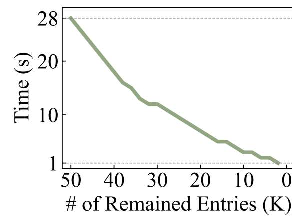


Figure 10: Time to remove 2k entries under different number of entries in OVS.

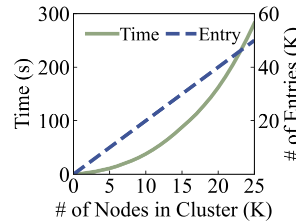


Figure 11: Time to remove all entries installed in OVS as the cluster scales up.

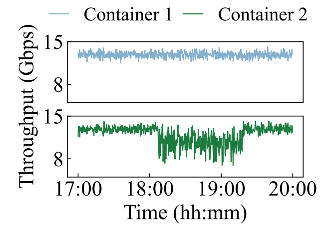


Figure 12: The RoCE throughput for two containers.

node cleanup time. A full OVS flush is not viable because entries from other tenants co-exist.

> Due to Nimitz's VxLAN-based solution generating a large number of entries for the overlay-underlay address mapping, there is a surge in the pressure from the Orchestrator-Agent communication channel and OVS-hardware offloading channel, when the cluster scale expands beyond 10K nodes.

4.2 Performance Fluctuation

One day after 6:00 PM, one server in our LLM training cluster experiences RDMA performance fluctuation, as shown in Figure 12 labeled as Container 2. Our engineers observed that there is a high surge of offloaded entry's lookup missed and lost. The total number of offloaded entries also fluctuated despite the traffic in the cluster remains stable (container 1 in Figure 12). After further investigation, we found that the immediate cause was the OVS frequently offloading RoCE-related entries and removing them from the NIC. At last, we found that the dynamic offloading scheme is the culprit.

As shown in §3.3, Nimitz adopts a dynamic solution to offload active OVS entries to the hardware to maximize the packet processing performance; meanwhile, OVS actively removes aged entries in the hardware to avoid overwhelming. However, the definition of 'overwhelm' is tricky.

In our vendor's NIC, the offloaded entries are grouped into Flow Groups (FGs) and the NIC looks up each FG until finds a matched entry. Each FG lookup slows down the NIC's hardware forwarding performance and the NIC still has enough memory capacity before obvious performance degradation. Therefore, there is no clear capacity indicator for the OVS to decide when to remove entries. Instead, OVS adopts a timer-based 'revalidation' mechanism: it periodically dumps the offloaded table entries and monitors the dump duration. If the duration is longer than a timeout threshold, it believes there are too many offloaded table entries that affect the forwarding performance, then it checks the statistics of the entries and removes idle entries.

When one tenant finished a previous LLM training job and submitted a new one that spans 12k nodes, all related nodes installed table entries of the entire container cluster's overlay underlay mapping. This triggers OVS's revalidation, which removed entries installed in the previous job. However, the entry removal operation took longer time than expected because the driver had to navigate the numerous offloaded table entries. OVS is deceived into believing too many entries are offloaded and started to remove more, inactive entries. To make things worse, the NIC driver's entry counter statistics can be delayed up to 2s, OVS pulled zero counters out of the newly offloaded entries for RoCE packets. These entries were removed by mistake, and the RoCE packets were sent to the OVS software table. The OVS offloading engine re-offloads them to the hardware and the revalidation engine re-removes them, which triggers RDMA performance fluctuation. We mitigated this issue by increasing the revalidation timeout threshold and working with our vendors to reduce the counter-report delay.

> At large scale, the mismatch between the large OVS table and limited hardware capacity requires a dynamic offloading solution, which is prone to failures.

4.3 Virtualization Overhead

Recent AI workloads pose great pressure on the throughput and latency of the transport layer protocol, especially RoCE. Even though Nimitz has already offloaded VxLAN encapsulation and decapsulation operation onto the hardware, it still incurs overhead relative to native physical-network performance, *i.e.,* bare-metal servers without container-network virtualization: (1) 5% throughput overhead on large RoCE messages due to the VxLAN encapsulation header, and (2) 5% latency overhead on small messages due to the encap/decap operation. The throughput overhead originated from the VxLAN header and the latency overhead comes from the NIC's encapsulation operation. The overhead is non-negligible due to the high cost of the GPU container network. We mitigated the throughput overhead by increasing the RoCE MTU. However, the latency overhead cannot be removed fundamentally and it causes a 3% throughput drop for multi-GPU inference jobs, as we show in §7.

> The VxLAN encap/decap incurs unavoidable latency and throughput overhead while modern services pose higher requirements for reducing virtualization overhead.

4.4 Lessons Learned and Summary

The above observations and analysis tell us that the culprit is the VxLAN-based network virtualization solution—it is too heavy. VxLAN creates a boundary and completely isolates the overlay and underlay address space, but one cannot navigate the overlay network without knowing the underlay network. As a result, the

entire overlay network's topology has to be stored on each node, a single change has to be broadcast to every participant, and the dynamic offloading a solution is only a compromise rather than a good solution.

Failed attempts. We also tried to address this scalability problem by optimizing VxLAN and NIMITZ: (1) caching the packets that are frequently encapsulated and decapsulated, and (2) increasing the number of offloaded entries by developing expensive programmable hardware. However, these attempts still failed to resolve the C100K problem due to the fundamental overhead bottleneck of VxLAN. We also considered sharding or caching only a subset of mappings, lazy loading, and lazy cleanup. These alternatives reduce average-case state, but they do not remove the global mapping dependency: service and load-balancer traffic may route to destination nodes unpredictably, so cache misses fall on the critical path. For RoCE, missing routing/offload state during connection setup can introduce first-packet failures or connection-establishment delays, which are unacceptable for production AI workloads. Lazy cleanup also caused hardware-table pressure in practice; once the table is full, installing new entries requires identifying and evicting stale entries, which reintroduces Orchestrator synchronization pressure.

We believe now is the time to rebuild the container network from scratch.

5 Chances

Revisiting how services—especially large-scale ones—use the container network, we found a major mismatch between NIMITZ 's functionality and tenants' practical needs. NIMITZ 's VxLAN virtualization cleanly separates overlay and underlay, providing an isolated, continuous address space with maximal flexibility and advanced features (custom routing, "bring your own IP," IP migration). In practice, these features are rarely used, particularly by services spanning more than $O(10k)$ nodes. Our tenant discussions revealed that, at this scale, services often manage addressing themselves: for example, LLM training frameworks [5, 6, 29] collect participants' addresses at a master node and broadcast them, while service meshes expose a single endpoint and map connections to backend containers via load balancers or DNS [23]. As a result, tenants mainly need high-level primitives—connectivity, ACLs, and load balancing—rather than fine-grained subnet control. The overlay–underlay separation thus adds complexity and operational overhead that contributes to the C100k problem, suggesting that VxLAN—carried over from the VM era—is increasingly overkill for modern container networks.

This observation echoes NIMITZ 's CIDR optimization: binding a subnet per node reduces address flexibility but improves scalability. Our deployment experience shows this trade-off is effective, motivating us to push further in the same direction.

6 NETPILA

Since 2022, we developed NETPILA, our new-generation container network to address the scalability problem. To avoid the overhead resulting from the overlay-underlay mapping, NETPILA abandons the expensive VxLAN technique. Instead, NETPILA proposes a simple yet practical IPv6-encoding solution: encoding the node address, tenant ID, and container address into a single 128-bit IPv6 address.

Essentially, NIMITZ maps overlay to underlay by encapsulating the container address with VxLAN to obtain an underlay address, enabling forwarding on the base network. Therefore, if we abandon VxLAN, we need an encoding method that can accommodate both physical network addresses and container addresses. IPv6 offers such a solution: IPv6 provides sufficient space to encode the container addresses into the IPv6 segment, like physical nodes, allowing packet forwarding directly as if it were a physical IPv6 network.

Based on the above intuition, NETPILA appends the container address and tenant ID after the node address (shown in Figure 13(a)). In NETPILA, the "overlay" (*i.e.,* container address) becomes the base network's natural extension. The container network reuses the base network's routing capability and no longer maintains any overlay-underlay mapping. The Orchestrator no longer broadcasts the updates to the entire cluster. Each Agent only focuses on node-local packet routing and update events. The routing table of the packet processing pipeline in each node is also greatly shrunk and can be offloaded to the hardware entirely. Getting rid of the VxLAN overhead, NETPILA can provide bare-metal performance in a container network. Compared with NIMITZ, NETPILA's architecture brings better elasticity, stability, and performance at a lower maintenance cost.

6.1 NETPILA Container Network

Figure 13 shows NETPILA's IPv6 encoding scheme and traffic processing. In NETPILA, each node has a globally unique 64-bit node address that occupies the first 64 bits in the IPv6 address and is reachable in the base network. Each tenant is assigned a 24-bit tenant ID. We reserve tenant ID 0 for services deployed on the node. When a new container is created in a node and joins the cluster, NETPILA's Agent assigns a locally unique 40-bit container address. In NETPILA, the Agent is still responsible for updating the table entries offloaded in the NICs. The Agent does not coordinate with other Agents to select the container address because the globally unique 64-bit node address guarantees the final IPv6 address is also unique. The Orchestrator no longer broadcasts the IP because the base network can route the traffic correctly by only examining the first 64-bit node address, and containers can directly reach each other as long as they know their IP address. Therefore, the cluster update event is simplified from a global broadcast to a local point-to-point communication.

We redesigned NETPILA's packet processing pipeline, which is much more simplified than NIMITZ. Firstly, it only routes packets without complicated operations such as encapsulation or decapsulation. Secondly, without the overlay or underlay concept, all the IPs are globally reachable; thus, the packet processing pipeline in each node only handles intra-node routing. As shown in Figure 13(b), all traffic types can be directly routed, NETPILA no longer maintains VxLAN, Gateway, or edge switches. As shown in Figure 14, the packet processing pipeline is simplified into two stages, the entire pipeline is offloaded in the NICs. We focus on explaining the Forwarding stage. For each container deployed locally, NETPILA adds two pairs of entries, one for east-west traffic that goes in and out of the container, and one for north-south traffic. Compared with NIMITZ, the total number of entries is significantly reduced

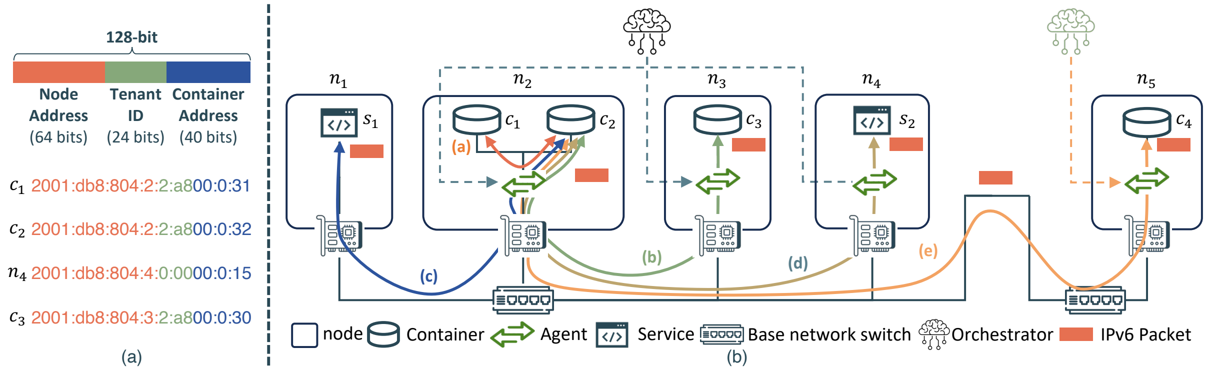
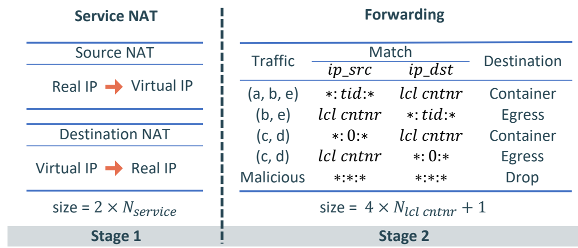


Figure 13: The NetPila IPv6 encoding example and traffic processing flow.



Service NAT
Source NAT
Real IP → Virtual IP
Destination NAT
Virtual IP → Real IP
size = $2 \times N_{service}$
Stage 1

Traffic	Match: *ip_src*	Match: *ip_dst*	Destination
(a, b, e)	*:*tid*:*	*lcl cntnr*	Container
(b, e)	*lcl cntnr*	*:*tid*:*	Egress
(c, d)	*:0:*	*lcl cntnr*	Container
(c, d)	*lcl cntnr*	*:0:*	Egress
Malicious	*:*:*	*:*:*	Drop

Forwarding — size = $4 \times N_{lcl\ cntnr} + 1$ — Stage 2

Figure 14: The packet processing pipeline of NetPila. The IP in the match field is presented in 'Node Address:Tenant ID:Container Address' format.

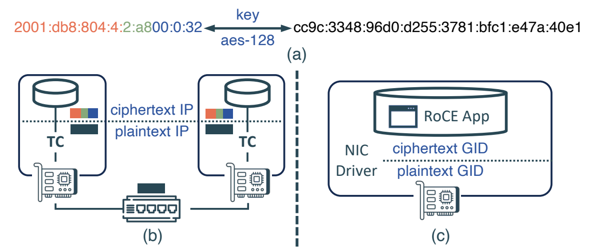


Figure 15: The encryption example for IP and traffic.

from $O(node)$ to $O(lcl\ cntr)$, and NetPila can offload all entries to the hardware. Therefore, as shown in Figure 13(b), NetPila removes OVS and directly interacts with the NIC. This changes the packet processing pipeline in the nodes of the container network from a dynamic solution to a static one, significantly improving the container network's stability. Furthermore, since the encapsulation/decapsulation overhead is removed, there is no latency or throughput overhead.

Take traffic (c) in Figure 13(b) as an example, the packet traverses $c_2 \to$ NIC on $n_2 \to$ NIC on $n_1 \to$ service s_1. Note the Agent in NetPila just plays as a channel role, and all entries have been offloaded onto the NIC. c_2 generates a packet p with the destination IP of s_1. The NIC on n_2 runs the packet processing pipeline in Figure 14. SNAT in stage 1 translates p's src IP (*i.e.,* c_2's IP) into a virtual IP (*i.e.,* service IP running on c_2). Then, since s_1 is deployed on nodes, packet p's destination IP's tenant ID is 0, which matches the fourth entry in the Forwarding table in stage 2. The packet is then forwarded to the base network. The base network forwards p by examining the first 64 bits of the destination IP. Then the packet is delivered to the service s_1 deployed in n_1.

The above solution is feasible because of two reasons. Firstly, IPv6 provides spacious address space. The IPv4 address space is crowded and we can only assign one IP per machine. While for IPv6, we can assign the entire /64 subnet to one machine. Secondly, the base network forwards IPv6 traffic based on the 64-bit prefix.

Summary. Compared with Nimitz, NetPila completely abandoned the VxLAN and proposed an IPv6 encoding-based solution. NetPila's design reduces the Orchestrator's workload, simplifies the Agent's complexity, and removes the overlay-underlay mapping overhead. It is our solution to the C100k problem. Table 1 summarizes the quantified difference between Nimitz and NetPila. We have deployed NetPila in our production for over two years and evaluated its advantages (see §7).

6.2 NetPila Isolation and Security

NetPila's IP-encoding scheme exposes node addresses and tenant IDs to containers because addresses are no longer hidden by VxLAN encapsulation as in Nimitz. This raises two risks: (1) weakened tenant isolation and (2) leakage of node addresses. We address both as follows.

Threat model. We focus on buggy or malicious tenant containers that may spoof source or destination addresses, scan address space, infer physical node information, or inject cross-tenant traffic. We assume the host OS, Agent, NIC firmware, and physical network are trusted components operated by the cloud provider; host compromise and key compromise are outside the scope of this design.

Tenant isolation. As shown in Figure 14, NetPila drops all packets by default. Each installed entry validates both source and destination IPs, including the tenant-ID field, to prevent reflection and cross-tenant traffic; with hardware-offloaded rules, only explicitly permitted traffic is forwarded. Tenant-ID validation and address encryption therefore provide defense in depth: the data plane rejects packets that do not match installed policy, while encryption hides placement information from tenant workloads.

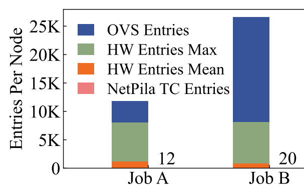


Figure 16: The scales of OVS entries and offloaded hardware entries of clusters run by NIMITZ and NETPILA.

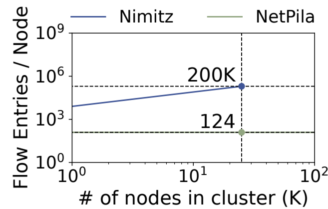


Figure 17: Comparison between NIMITZ and NETPILA on the number of flow entries per node.

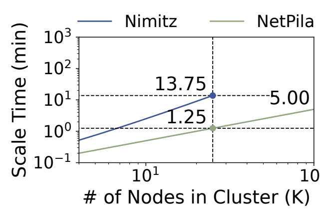


Figure 18: Comparison between NIMITZ and NETPILA on the time to scale up 10%.

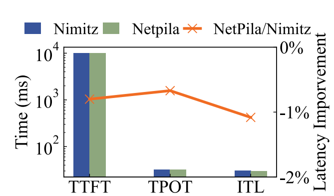


Figure 19: Comparison between NIMITZ and NETPILA in vLLM benchmark.

Table 1: Comparison between NIMITZ and NETPILA

Metric	NIMITZ	NETPILA
Packet processing pipeline space complexity	$O(cntnr) + O(node)$	$O(lcl\ cntnr)$
Number of messages to scale up 10%	$O(node^2)$	$O(node)$
Offloading channel throughput	~600 eps	N/A
Latency overhead	2.5%	0
Throughput overhead	2.6%	0

IP address encryption. To hide node information, NETPILA assigns each tenant an encryption key and gives containers an encrypted (ciphertext) IP instead of the assigned plaintext IP. The ciphertext is computed with a symmetric cipher (e.g., AES-128) over the plaintext IP and key (Figure 15(a)). On standard IP communication (e.g., $c_1 \longrightarrow c_3$ in Figure 13(b)), node n_2 decrypts source/destination IPs, forwards using plaintext addresses, and re-encrypts before delivery (Figure 15(b)). To avoid per-packet CPU overhead, NETPILA handles only the first packet in software, caches the mapping, and installs a hardware translation entry via `tc` ahead of the pipeline stages in Figure 14, keeping existing entries unchanged. In a production cluster, baseline cross-node pod RTT on the physical network is 0.082 ms. Under IP-cipher replacement, first-packet RTT is 0.812 ms on average and 1.386 ms at P95; subsequent packets hit hardware-installed entries and do not incur this software-path latency. The hardware-state cost is one translation mapping per active flow, in addition to normal forwarding rules.

The probability of unauthorized traffic entering the network, *i.e.*, the probability that a packet carrying a forged ciphertext IP also guesses the correct 24-bit tenant ID in the validated NETPILA address format and passes tenant-ID checks, is as low as 2^{-24}. Cryptographic confidentiality is provided separately by AES-128.

Encryption-based IP replacement can still add hardware-state overhead and first-packet latency for standard IP traffic. For RoCE (*i.e.,* RDMA on NICs), NETPILA instead uses GID replacement (Figure 15(c)) by modifying the host NIC driver: each VF (or SF) maintains a ciphertext GID visible to the container and a plaintext GID visible only to the host. During connection setup, the driver swaps the ciphertext GID for the plaintext one, so the NIC transmits with plaintext GIDs without rewriting other fields. This occurs on the connection-establishment path, reuses the QP context in hardware, adds no first-packet latency, incurs no extra hardware resource overhead, and requires zero tenant intervention. This distinction explains why NETPILA tracks Native IPv6 within measurement noise for RoCE workloads.

6.3 Discussions and Lessons

IP Addressing Flexibility. NETPILA's IPv6 encoding reduces tenant control over low-level routing and address allocation. For large deployments, however, this "flexibility" often becomes an operational burden: managing a highly dynamic network with 100K containers is already complex. In practice, LLM training/inference and most cloud-native services rarely use VxLAN's classic flexibility, such as IP mobility, hot migration, custom subnet control, or bring-your-own IP. These features remain valuable for VM-style virtualization, but modern large-scale container services usually rely on service discovery, load balancing, and application-level membership management. Migration resistance mainly came from legacy IPv4-heavy applications with complex dependencies. New AI workloads showed little resistance, and some external bare-metal customers explicitly requested IPv6-only deployment. Overall, customers prefer trading limited addressing freedom for better elasticity and performance.

Virtual Private Cloud (VPC) vs. Container Network. VPC targets VM connectivity in IaaS, whereas container networks target CaaS/PaaS. VPC techniques do not resolve the C100K problem: tunneling-based virtualization (common in VPCs) incurs substantial management overheads, and NIMITZ 's VxLAN inheritance from the VM setting suffers high synchronization cost (§4). Prior VPC designs show similar limits: Canal Mesh's VxLAN gateway [30] adds latency and consumes significant CPU, and Triton [22] shows that even specialized hardware struggles with large overlay–underlay mappings.

▷ *NETPILA 's IPv6 encoding is container-network specific.* Unlike CaaS, IaaS typically requires isolated address space and richer features (user-defined routing, IP migration, subnet assignment). These constraints prevent VPCs from adopting NETPILA 's IPv6 encoding, since user-selected subnets may not fit NETPILA 's address format. Thus, NETPILA 's approach is specific to container networks, which operate within a self-contained IP space—a key IaaS vs. CaaS distinction.

Does IPv6 introduce any limitation? NETPILA requires IPv6 in both the base network and container-facing addresses. Our base network is already IPv6. For tenants running IPv4 services, we provide an IPv4–IPv6 translation layer with negligible overhead;

many NetPila services run IPv4 today via this mechanism. IPv6 also does not hinder communication with IPv4-based ISPs, since we can apply NAT-like translation between IPv6 and IPv4.

Identifier Locator Addressing (ILA). ILA [10] is Meta's IPv6-based virtual networking solution [3]. It remains an overlay–underlay design: containers use a location-agnostic 64-bit ILA *identifier* under a shared SIR prefix, while a 64-bit *locator* encodes the real location and is hidden from containers. The network must look up and translate SIR:*identifier* into *locator*:*identifier* for delivery. Unlike NetPila, ILA still relies on overlay–underlay mappings, so its design principle differs fundamentally and the C100K issue (§4) remains.

NetPila extensibility and opportunities. Beyond addressing C100K, IPv6's protocol features and 128-bit space improve extensibility and open new opportunities. We defer details to Appendix B and C.

7 NetPila Evaluation

In this section, we demonstrate NetPila's elasticity and performance in our production and how NetPila solves the C100k problem. The evaluation was conducted on our production GPU clusters. Each GPU server has 8 GPUs and 4 NICs and GPUDirectRDMA (GDR) is properly configured. Unless explicitly stated, the figures report measured production data; we separate these measurements from any trend discussion beyond Nimitz's stable operating range. We compared the resource consumption, time to scale 10% cluster size, and the performance of RoCE and LLM inference jobs of two container networks, Nimitz and NetPila.

7.1 Elasticity

To quantify the benefit of simpler traffic processing, we ran two production jobs and measured per-node rule counts. Job A is an LLM training run spanning 8076 nodes; Job B is a distributed vLLM [20] inference run using 6176 nodes. Figure 16 shows that, despite its smaller scale, Job B installs more OVS entries due to higher container density and more external dependencies; both jobs peak at offloading ~8K entries to hardware. Migrating to NetPila reduces this dramatically: NetPila needs only 12–20 entries per node to realize the same network at the same scale, and all rules are installed directly in hardware without dynamic mechanisms. Since most traffic is RoCE, the encryption mechanism in §6.2 adds no extra hardware entries. This case study highlights NetPila 's practical advantage.

We further evaluated NetPila's elasticity by increasing cluster size while fixing 31 containers per node, and recording (i) entries per node and (ii) time to scale up 10% of the cluster. We use 31 containers per node because it reflects a representative GPU-cluster density in the evaluated production setting, where each server has 8 GPUs and inference workloads may run multiple containers per GPU. We use a 10% scale event because it is a common operational target for bursty expansion and contraction and the setting most frequently requested by tenants; the percentage is not special to either design. The experiment is conducted in a controlled experiment cluster. Figure 17 and Figure 18 report the results. Because Nimitz becomes unstable beyond 25K nodes, we report measured Nimitz data only up to its stable operating range and use the remaining trend discussion solely to explain asymptotic behavior. As shown in Figure 17, Nimitz's per-node entries grow linearly with cluster size, consistent with Figure 5, whereas NetPila's remain constant when container density is fixed. Figure 18 shows Nimitz's 10% scale-up time grows quadratically with cluster size (with Orchestrator throughput capped at 200 qps), while NetPila grows linearly: node updates parallelize, but the Orchestrator still bottlenecks. At 25K nodes, Nimitz needs 12.5 minutes versus 1.25 minutes for NetPila; the gap widens with scale. At 100K nodes, scaling 10% (10K nodes) costs NetPila only 5 minutes, and its linear message growth can be further reduced by scaling the Orchestrator horizontally.

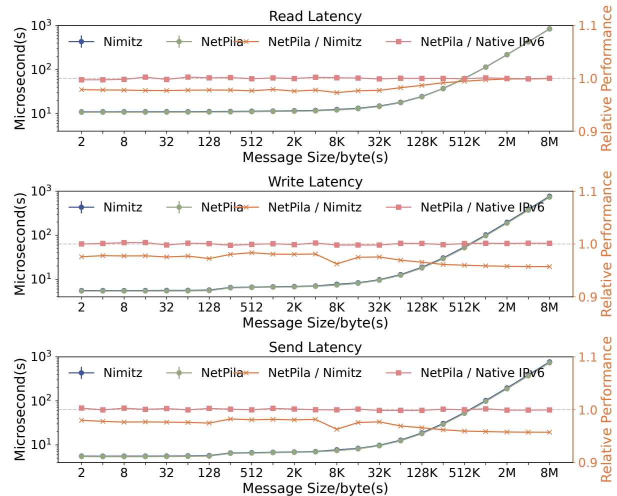


Figure 20: RDMA Read/Write/Send Latency Comparison between Nimitz and NetPila.

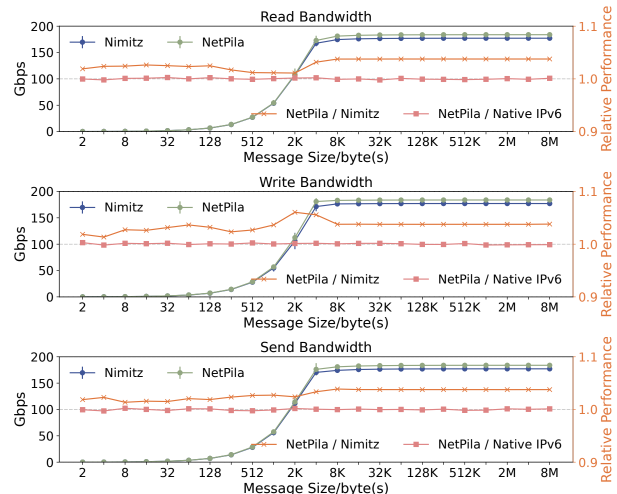


Figure 21: RDMA Read/Write/Send Bandwidth Comparison between Nimitz and NetPila.

7.2 Performance

Compared with Nimitz, not only does NetPila alleviate the overhead of large-scale containerized deployment, but also provides higher performance for containerized LLM applications. To demonstrate the AI application performance gain achieved by NetPila, we conducted both RDMA performance tests and end-to-end LLM application benchmarks. In this experiment, we used A100-SXM4-80GB GPUs, Mellanox Connect6DX NICs, and set the RoCE MTU to 1024 bytes. We also compared NetPila with the Native IPv6 baseline, where we ran the experiment on bare-metal servers with no virtualization at all.

Perftest. We chose the widely adopted perftest [11] to measure the concrete RoCEv2 performance in the container network. For RoCEv2 bandwidth test, we use 4 Queue Pairs (QPs) to avoid the Packet Per Second (PPS) bottleneck. Figure 20 and Figure 21 show the RoCEv2 latency and bandwidth performance comparison between Nimitz and NetPila respectively. NetPila achieved **4.281%** maximum reduction in latency and **6.055%** maximum increase in bandwidth. This is because NetPila bypasses the encapsulation and decapsulation operation overhead on the NIC to reduce end-to-end latency, it also avoids VxLAN header overhead to increase bandwidth utilization. We can also see the performance difference between NetPila and Native IPv6 is within the margin of error. This shows that NetPila's RoCE encryption solution only incurs negligible overhead.

LLM inference benchmark. We chose vLLM [20], a popular LLM inference framework, to measure the end-to-end LLM application performance gain achieved by NetPila. We deployed the Llama-3.2 model on two servers and recorded the Time To First Token (TTFT), Time Per Output Token (TPOT), and Inter-token Latency (ITL) for Nimitz and NetPila. The results are shown in Figure 19.

The experiment results validate that NetPila improves end-to-end LLM application performance. Overall, compared with Nimitz, NetPila brings about **0.854%** extra LLM text generation request throughput, **0.863%** extra output token throughput and **0.862%** extra total token throughput. In terms of token latency, as is shown in Fig. 19 NetPila outperforms Nimitz with **0.800%** less mean time to first token, **0.669%** less mean time per output token (excl. 1st token) and **1.081%** less mean inter-token latency.

8 Related Work

Container networks. Open-source container network frameworks, such as Flannel [9] and Calico [7], play a critical role in enabling networking within and across containerized environments. Flannel primarily focuses on providing a simple and easy-to-use overlay network, while Calico employs a more robust approach using layer-3 routing and network policies. Starlight [16] provides a fast container provisioning framework for edge settings.

Service Mesh Frameworks. Service mesh frameworks, such as Istio [12], Linkerd [14], and Consul [8], abstract network management complexities by introducing service discovery, traffic management, and advanced security policies. Together with Kubernetes [13], the frameworks commonly serve as the Orchestrator and manage the container network. Service fabric [18] proposes a distributed platform for building microservices in the cloud. SigmaOS [31] provides a uniform API for service mesh and container orchestration.

Overlay network. OVS [25] and VMware NSX [32] are common overlay network frameworks. They mainly choose tunneling protocols such as VxLAN [2], Generic Routing Encapsulation (GRE) [1], and GEneric NEtwork Virtualization Encapsulation (GENEVE) [4]. Slim [33] provides a low-overhead container overlay network implementation. Oncache [15] proposes a cache-based low-overhead container overlay network. Falcon [21] proposes a scalable packet processing pipeline in the overlay network. These systems optimize overlay networking, whereas NetPila takes an orthogonal direction by eliminating overlay-underlay mappings for large-scale container networks. Canal Mesh [30] and Triton [22] report related scaling pressure in VPC and overlay deployments: gateway-based designs incur latency and CPU overhead, while customized hardware still struggles with large mapping tables. ILA [3, 10] also uses IPv6, but it preserves an overlay-underlay abstraction by translating location-agnostic identifiers into locators; NetPila instead encodes routable node, tenant, and container information in one address and removes the global mapping dependency.

RDMA virtualization. FreeFlow [19], HyV [27], CoRD [28], and MasQ [17] provide a software-based virtual RDMA networking for containerized clouds. Harmonic [24] focuses on performance isolation for RDMA in the public cloud setting. These works improve RDMA virtualization abstractions and isolation. Our focus is complementary: we study how the container network's global overlay state, hardware offloading behavior, and address-management model affect production AI clusters at C100K scale.

9 Conclusion

This paper has presented the motivation and evolution of our two generations of systems Nimitz and NetPila. By introducing an IPv6-encoding approach, NetPila has successfully addressed the C100K problem and supported our production-scale container network for over two years.

ACKNOWLEDGEMENTS

We sincerely thank the anonymous reviewers and our shepherd, Nitinder Mohanfor their constructive feedback and valuable suggestions. Ennan Zhai is the corresponding author for this paper.

References

[1] 1994. Generic Routing Encapsulation (GRE). (1994). https://datatracker.ietf.org/doc/html/rfc1701
[2] 1994. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. (1994). https://datatracker.ietf.org/doc/html/rfc7348
[3] 2017. Internet-scale virtual networking using IPv6 ILA. (2017). https://atscaleconference.com/videos/internet-scale-virtual-networking-using-ipv6-ila/
[4] 2020. Geneve: Generic Network Virtualization Encapsulation. (2020). https://www.rfc-editor.org/rfc/rfc8926.html
[5] 2024. DeepSpeed. https://www.microsoft.com/en-us/research/project/deepspeed/. (2024).
[6] 2024. Megatron-LM. https://github.com/NVIDIA/Megatron-LM. (2024).
[7] 2025. Calico. (2025). https://docs.tigera.io/calico/latest/about/
[8] 2025. Consul. (2025). https://www.consul.io/
[9] 2025. Flannel. (2025). https://github.com/flannel-io/flannel
[10] 2025. Identifier Locator Addressing (ILA). (2025). https://docs.kernel.org/networking/ila.html

[11] 2025. Infiniband Perftest. Open Fabrics Enterprise Distribution (OFED) Performance Tests. (2025). https://github.com/linux-rdma/perftest.

[12] 2025. Istio. (2025). https://istio.io/

[13] 2025. Kubernetes. (2025). https://kubernetes.io/

[14] 2025. Linkerd. (2025). https://linkerd.io/

[15] 2025. ONCache: A Cache-Based Low-Overhead Container Overlay Network. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA. https://www.usenix.org/node/305311

[16] Jun Lin Chen, Daniyal Liaqat, Moshe Gabel, and Eyal de Lara. 2022. Starlight: Fast Container Provisioning on the Edge and over the WAN. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 35–50. https://www.usenix.org/conference/nsdi22/presentation/chen-jun-lin

[17] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. 2020. MasQ: RDMA for Virtual Private Cloud. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3387514.3405849

[18] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, Mansoor Mohsin, Ray Kong, Anmol Ahuja, Oana Platon, Alex Wun, Matthew Snider, Chacko Daniel, Dan Mastrian, Yang Li, Aprameya Rao, Vaishnav Kidambi, Randy Wang, Abhishek Ram, Sumukh Shivaprakash, Rajeet Nair, Alan Warwick, Bharat S. Narasimman, Meng Lin, Jeffrey Chen, Abhay Balkrishna Mhatre, Preetha Subbarayalu, Mert Coskun, and Indranil Gupta. 2018. Service fabric: a distributed platform for building microservices in the cloud. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 33, 15 pages. https://doi.org/10.1145/3190508.3190546

[19] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. 2019. FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 113–126. https://www.usenix.org/conference/nsdi19/presentation/kim

[20] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[21] Jiaxin Lei, Manish Munikar, Kun Suo, Hui Lu, and Jia Rao. 2021. Parallelizing packet processing in container overlay networks. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 261–276. https://doi.org/10.1145/3447786.3456241

[22] Xing Li, Xiaochong Jiang, Ye Yang, Lilong Chen, Yi Wang, Chao Wang, Chao Xu, Yilong Lv, Bowen Yang, Taotao Wu, Haifeng Gao, Zikang Chen, Yisong Qiao, Hongwei Ding, Yijian Dong, Hang Yang, Jianming Song, Jianyuan Lu, Pengyu Zhang, Chengkun Wei, Zihui Zhang, Wenzhi Chen, Qinming He, and Shunmin Zhu. 2024. Triton: A Flexible Hardware Offloading Architecture for Accelerating Apsara vSwitch in Alibaba Cloud. In *Proceedings of the ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 750–763. https://doi.org/10.1145/3651890.3672224

[23] Zhijing Li, Zihui Ge, Ajay Mahimkar, Jia Wang, Ben Y. Zhao, Haitao Zheng, Joanne Emmons, and Laura Ogden. 2018. Predictive Analysis in Network Function Virtualization. In *Proceedings of the Internet Measurement Conference 2018, IMC 2018, Boston, MA, USA, October 31 - November 02, 2018.*

[24] Jiaqi Lou, Xinhao Kong, Jinghan Huang, Wei Bai, Nam Sung Kim, and Danyang Zhuo. 2024. Harmonic: Hardware-assisted RDMA Performance Isolation for Public Clouds. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 1479–1496. https://www.usenix.org/conference/nsdi24/presentation/lou

[25] Open vSwitch Project. 2025. Open vSwitch: Production Quality, Multilayer Virtual Switch. (2025). https://www.openvswitch.org/ Accessed: 2025-01-25.

[26] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. 2015. The Design and Implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 117–130. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff

[27] Jonas Pfefferle, Patrick Stuedi, Animesh Trivedi, Bernard Metzler, Ionnis Koltsidas, and Thomas R Gross. 2015. A hybrid I/O virtualization framework for RDMA-capable network interfaces. *ACM SIGPLAN Notices* 50, 7 (2015), 17–30.

[28] Maksym Planeta, Jan Bierbaum, Michael Roitzsch, and Hermann Härtig. 2023. CoRD: Converged RDMA Dataplane for High-Performance Clouds. (2023). arXiv:cs.OS/2309.00898 https://arxiv.org/abs/2309.00898

[29] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *CoRR* abs/1909.08053 (2019). arXiv:1909.08053 http://arxiv.org/abs/1909.08053

[30] Enge Song, Yang Song, Chengyun Lu, Tian Pan, Shaokai Zhang, Jianyuan Lu, Jiangu Zhao, Xining Wang, Xiaomin Wu, Minglan Gao, et al. 2024. Canal Mesh: A Cloud-Scale Sidecar-Free Multi-Tenant Service Mesh Architecture. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 860–875.

[31] Ariel Szekely, Adam Belay, Robert Morris, and M. Frans Kaashoek. 2024. Unifying serverless and microservice workloads with SigmaOS. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 385–402. https://doi.org/10.1145/3694715.3695947

[32] VMware, Inc. 2025. VMware NSX: Network Virtualization Platform. (2025). https://www.vmware.com/products/nsx.html Accessed: 2025-01-25.

[33] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. 2019. Slim: OS Kernel Support for a Low-Overhead Container Overlay Network. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 331–344. https://www.usenix.org/conference/nsdi19/presentation/zhuo

APPENDIX

A Traffic (d) Processing

We detail traffic (d) based on Figure 4 and Figure 5 examples.

Traffic (d). For an underly service that aims to access the overlay, the traffic belongs to (d). Recall the traffic (d) from load balancer (*i.e.,* underlay) to inference cluster (*i.e.,* overlay) in the example in Figure 3. Traffic (d) is the most complex traffic case in NIMITZ. For example in Figure 4, a traffic (d) is: service s_2 → Gateway on n_4 → o_3 → VxLAN on n_4 → NIC on n_4 → NIC on n_2 → VxLAN on n_2 → o_1 → c_2. A service sends a packet p (with the destination IP of Gateway on n_4) to the Gateway on n_4. The Gateway finds the destination IP of p itself so that the Gateway translates p's destination IP into a virtual service IP tmp_IP, and then sends p to o_3. o_3 checks p and finds p's sender is a gateway; thus, o_3 fills its VNI register with 0xFF (mentioned earlier) in stage 2. In stage 3, o_3 uses DNAT to translate p's destination IP from the virtual service IP tmp_IP to a real IP (*i.e.,* c_2's IP). In stage 4, o_3 finds p's destination IP is a remote container (*i.e.,* c_2); thus, it hits the fourth entry in the forwarding table of o_3. Finally, o_3 encapsulates p via VxLAN (in stage 5) and then sends it to n_2 through the VxLAN and NIC on n_4. What happens on o_1 is similar to the one described in traffic (b), so we omit this part.

We omit traffic (e) details since it is the combination of traffic (b) and (d).

B NETPILA Extensibility

The advanced IPv6 protocol and spacious 128-bit address space not only solves the C100k problem, but also enables better extensibility. It provides an easy interface for the base network to perceive the application semantics and collect application-level information. It also exposes base network services to the application so that applications can subscribe and experience better network quality at a lower cost. In this section, we present two examples deployed in production to demonstrate such extensibility.

App-controlled statistics. One of our tenants requests billing for the public network traffic going through the data center gateway. The challenge is that the tenant only wants to bill a subset of applications deployed in their service. In NIMITZ, we can only ask the tenant to provide all containers that host the billed applications and

deploy a dedicated lookup table to collect traffic statistics. On the one hand, this solution requires a dedicated connection between the tenant's orchestrator and the gateway, which demands non-trivial engineering effort. On the other hand, the frequent synchronization and large number of containers creates a high burden for our gateway device. After migrating the container network solution from NIMITZ to NETPILA, we reserved a dedicated application ID in the last 40-bit container address field. The tenant can choose the container address with the marked application ID. The gateway device only needs to collect statics of packets that match the tenant and application ID.

Segment Routing over IPv6 dataplane (SRv6). SRv6 allows users to control the packet routes by inserting segment information in the header. As we roll out SRv6 functionality in our backbone switches, NETPILA can expose such capability to our clients. For example, providing reserved bandwidth for remote storage services, routing traffic through cheaper ISPs, *etc.* NETPILA exposes such capability via pre-defined APIs. The tenant can tag the containers with desired services and parameters, NETPILA then selects and attaches corresponding SRv6 headers to the outgoing packet, the backbone network then provides the selected services to the tenant.

C More Discussions and Lessons

Upper bound of NETPILA. The IPv6 solution of NETPILA is not only capable of scaling to 10K nodes but can even extend to 100K (even more) nodes due to its extremely low table entry complexity. We have not yet observed any scalability issues with NETPILA so far. We believe that future scalability issues with our CaaS may not arise from the network (*i.e.,* NETPILA), but could instead be due to certain bottlenecks at the distributed software system level.

Opportunities. Using IPv6 in our encoding is just our specific solution for the C100K problem. In fact, solving the C100K problem does not necessarily require using the IPv6-encoding approach (it should have alternatives). We believe that by developing a more lightweight operating system to reduce the number of table entries or by designing a more scalable addressing method, the C100K problem can also be addressed. We leave the above thoughts to the future work.