Nezha: SmartNIC-Based Virtual Switch Load Sharing

Xing Li[1,2], Enge Song[2], Bowen Yang[2], Tian Pan[2], Ye Yang[2], Qiang Fu[3], Yang Song[2], Yilong Lv[2], Zikang Chen[2], Jianyuan Lu[2], Shize Zhang[2], Xiaoqing Sun[2], Rong Wen[2], Xionglie Wei[2], Biao Lyu[1,2], Zhigang Zong[2], Qinming He[1], Shunmin Zhu[4,2]
[1]Zhejiang University [2]Alibaba Cloud [3]RMIT University [4]Hangzhou Feitian Cloud
alibaba_cloud_network@alibaba-inc.com

Abstract

Cloud providers use SmartNIC-accelerated virtual switches (vSwitches) to offer rich network functions (NFs) for tenant VMs. Constrained by limited SmartNIC resources, it is a challenge to provide sufficient network performance for high-demand VMs. Meanwhile, we observed a significant number of idle vSwitches in the data center, which led us to consider leveraging them to build a remote resource pool for high-demand virtual NICs (vNICs). In this work, we propose Nezha, a distributed vSwitch load sharing system. Nezha reuses the existing idle SmartNICs to handle the excess load from the local SmartNIC without adding new devices. Nezha offloads stateless rule/flow tables to the remote, while keeping states locally. This eliminates the need for state synchronization, facilitating load sharing and failover. The deployment cost of Nezha is only a small fraction of that required to deploy new devices. Data collected from production show that our CPS capability bottleneck has shifted from the vSwitch to the VM kernel stack, with #concurrent flows and #vNICs increased by up to 50.4x and 40x, respectively.

CCS Concepts

• **Hardware → On-chip resource management**; • **Networks → Cloud computing**; **Data center networks**.

Keywords

SmartNIC, Distributed systems, Network virtualization

ACM Reference Format:
Xing Li, Enge Song, Bowen Yang, Tian Pan, Ye Yang, Qiang Fu, Yang Song, Yilong Lv, Zikang Chen, Jianyuan Lu, Shize Zhang, Xiaoqing Sun, Rong Wen, Xionglie Wei, Biao Lyu, Zhigang Zong, Qinming He, Shunmin Zhu. 2025. Nezha: SmartNIC-Based Virtual Switch Load Sharing. In *ACM SIGCOMM 2025 Conference (SIGCOMM '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3718958.3750466

1 Introduction

SmartNIC-based vSwitches are widely deployed in data centers to provide hardware-accelerated networking for virtual machines, containers, and serverless services [25, 29, 32, 34, 35, 50]. After deploying SmartNICs in our data centers for over 8 years, we have observed a paradox in today's production environments: simultaneous vSwitch resource "shortage" and "waste". The average CPU and memory utilizations of vSwitches in a region were about 5% and 1.5%, respectively, while P9999 utilization reached 90% and 96%, respectively. This indicates that while most vSwitches are underutilized, a few are overloaded, becoming bottlenecks for users demanding high-performance network capabilities [25].

Rolling out high-capacity SmartNICs is not cost-effective as vSwitch overloads are rare. Partial rollout, however, makes the migration of heavily loaded VMs inevitable, as overloads may happen on any server in the network. Migrating these heavy VMs incurs significant overhead and downtime. Another approach is to leverage host resources to enhance vSwitch performance, but this may impact resource sales and contradict the purpose of SmartNICs.

Sirius [25] offloads the vSwitch processing logic of high-demand vNICs to a shared pool of high-performance cards, providing a pioneering solution. However, introducing dedicated devices is costly, especially when server SmartNICs are largely underutilized. In Sirius, packets need to take a detour to the shared pool or card pairs, which must maintain state, adding complexity and performance overhead. For example, Sirius replicates state in-line by ping-ponging packets between the primary and secondary cards — the NF capacity halves for this reason. Nevertheless, Sirius is a promising solution, offering high performance without the need to upgrade server SmartNICs.

In light of these issues, we follow the principle of "reuse before adding resources" and propose Nezha, which leverages idle SmartNICs as a resource pool for high-demand vNICs. Our production data show that the number of new connections per second (CPS) is the dominant capability bottlenecked at SmartNICs, followed by #concurrent flows and #vNIC. CPS is limited by CPU through rule table lookups while #concurrent flows and #vNICs are primarily limited by memory through flow/rule tables. This motivates us to move the rule/flow tables to the resource pool. Intuitively, doing so would require state synchronization for fault tolerance and state transfer for load balancing.

To this end, we propose a novel approach that decouples state from rule/flow tables. Nezha only offloads stateless tables to the resource pool while keeping state locally in one copy, unlike other offloading solutions [25, 33]. This approach effectively eliminates the need for state synchronization or transfer across remote nodes in the resource pool. To address the issue where neither local nor remote nodes can independently process packets due to the separation of state and rule/flow tables, we leverage data packets to carry the necessary information for packet processing. To ensure proper state initialization and updates, we designed workflows for

Xing Li and Enge Song contributed equally to this work. Tian Pan and Shunmin Zhu are co-corresponding authors.

SIGCOMM '25, Coimbra, Portugal

ACM ISBN 979-8-4007-1524-2/2025/09
https://doi.org/10.1145/3718958.3750466

egress and ingress packets, using in-packet transmission and notify packets. It shows that Nezha can leverage existing server SmartNIC capacity to offer high-performance network capabilities for high-demand tenants, shifting the bottleneck from the vSwitch to the VM for CPS. Our key contributions are as follows:

- We propose Nezha, an architecture that leverages idle SmartNICs as a remote resource pool to offload high-demand vNICs, without introducing additional hardware. The reuse strategy allows us to avoid extra deployment costs associated with purchasing new devices, wiring, and software development, as well as additional human efforts for future iterations. Deploying Nezha on Alibaba Cloud requires only 10% of the development effort compared to Sailfish [41], which represents solutions that need to introduce new devices into the data center.
- We present a novel design that decouples state from rule/flow tables, offloading stateless tables to remote SmartNICs while maintaining state locally in a single copy, eliminating the need for state synchronization or transfer. With this design, Nezha achieves load balancing and fault tolerance across the nodes in the resource pool using only 5-tuple hashing, without the need for complex mechanisms such as symmetric or consistent hashing.
- For production deployment, we implemented Nezha with several techniques. Nezha seamlessly offloads high-demand vNICs through a dual-stage design, falling back to local processing when the load allows. The average and P99 completion times for activating offloading are about 1s and 2s, respectively, with no service interruptions. Nezha scales out resource sharing as needed and terminates it for any vSwitch experiencing high load from its local vNIC. Moreover, Nezha can detect crashes of remote node and complete failover within 2s.
- Nezha has been deployed in Alibaba Cloud for a year, improving CPS, #concurrent flows, and #vNICs for three cloud middleboxes by 3~4.4X, 5.04~50.4X, and over 40X, respectively. Nezha can resolve over 99.9% of the vSwitch overloads on CPS and #concurrent flows and completely avoid overloads on #vNICs. We also discuss and share our experiences deploying Nezha.

2 Background and Motivation

2.1 SmartNIC-based vSwitches

In data centers, vSwitch enables network communication under tenant isolation and enforces tenant-configured rules such as routing, metering, NAT, and ACLs for their virtual networks [28, 42]. To achieve diverse network functions (NFs), vSwitch generates the final packet processing actions (*i.e.*, forward, drop, header rewrite) based on incoming packets, tenant-configured rules, and optionally, session states (*e.g.*, TCP finite-state machine (FSM), flow-level statistics), through specific processing logic (Fig. 1).

Among the three inputs, the rules for packet processing are obtained by querying the rule tables, referred to as the **slow path**. For example, if an ACL denies packets from a certain IP, a rule match will trigger an action to drop those packets. A VM requires at least one vNIC [9, 10, 25] to communicate externally. To ensure tenant isolation, each vNIC maintains its own set of rule tables. Since the rule table lookup process involves querying multiple tables (including expensive range matching), and the lookup results for packets with the same 5-tuple are identical, the matched action

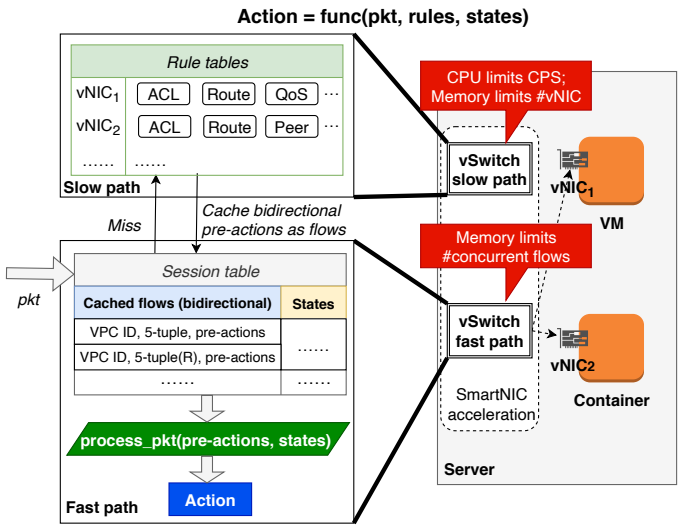


Figure 1: SmartNIC-based vSwitch architecture.

for the first packet can be cached, referred to as the **fast path** to accelerate the processing of subsequent packets of the same flow [27–29, 34, 35, 42, 50] (Fig. 1). To distinguish different tenants using the same 5-tuples, VPC ID is also recorded in the cached flows.

For stateless NFs, we can obtain packet actions by simply querying the rule tables, such as using a route table to decide how to forward packets. However, to implement stateful NFs, vSwitch needs to further combine session states with rules to derive the final packet actions. For example, even if the ACL does not permit any incoming traffic, responses to connections initiated by the local VM need to be allowed to pass through. Therefore, the "drop" action generated by the ACL table lookup is not final; the vSwitch must combine it with the direction of the first packet recorded in the state to determine the final action. In this regard, we refer to rule table lookup results as preliminary actions (pre-actions), as they are not the final packet actions for stateful NFs (Fig. 1). Since states like TCP connection status are maintained at the session level (rather than per flow), we record bidirectional flows and their session states in a single entry to avoid synchronization between separate flow entries [34, 50]. These states are initialized when the first packet arrives and may be updated by subsequent packets of the same session.

The packet processing logic of different NFs in vSwitch can be uniformly abstracted as *Action = func(pkt, rules, states)*, where, for stateless NFs, the states are null. With cached flows on the fast path, the packet processing logic can be accelerated to *process_pkt(pre-actions, states)*, with both inputs retrieved via exact matches in the session table (Fig. 1).

2.2 vSwitch Performance Bottleneck

2.2.1 Observations. **VMs overwhelming SmartNICs.** During deployment, we discovered that the demand on some network capabilities for high-load tenant VMs could not be met, as also reported by Azure [25]. Taking CPS as an example, we find that these high-CPS VMs cause severe overload on their servers' SmartNICs, with CPU utilization > 95% in all cases (Fig. 2). Meanwhile, these

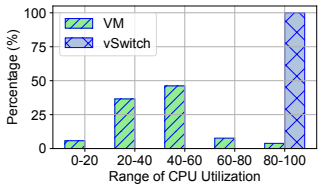


Figure 2: CPU usage of high-CPS VMs/their vSwitches.

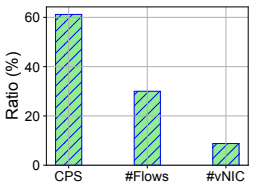
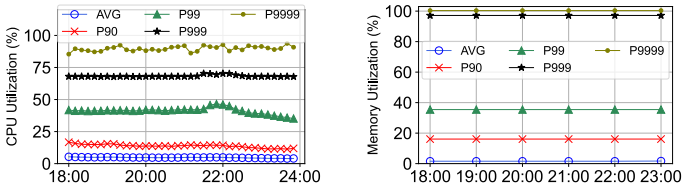


Figure 3: Hotspot distribution in a region.



(a) CPU (b) Memory

Figure 4: Resource utilization CDF on O(10K) vSwitches.

Table 1: Normalized distribution of CPS, #concurrent flows, and #vNICs usage.

	CPS	*#Concurrent flows*	*#vNICs*
P50	0.53%	0.78%	0.65%
P90	1.41%	2.36%	1%
P99	6.41%	6.39%	6%
P999	18.38%	29.17%	55%
P9999	100%	100%	100%

high-CPS VMs are lightly loaded, with 90% of them having CPU utilization below 60%. Figure 3 shows that CPS is the dominant capability bottlenecked at SmartNICs, followed by #concurrent flows and #vNIC (Appendix A.1). The key issue is the gap in available resources between the VMs and their SmartNICs. VMs with ample resources (hundreds of vCPUs, hundreds to thousands of GB of memory) [6, 7, 15] can easily overwhelm resource-limited SmartNICs (tens of CPUs, a few tens of GB of memory) [5, 17]. Essentially, VMs with high network demands deplete the SmartNICs' resources, not their own.

Extreme load imbalance across SmartNICs. Interestingly, we observe that only a few SmartNICs exhaust their CPU and memory resources, while most have very low resource utilization. We measured the CPU and memory utilization of SmartNICs across O(10K) servers in our cloud over time, as shown in Fig. 4. The CPU utilization values for average, P90, P99, P999, and P9999 are approximately 5%, 15%, 41%, 68%, 90%. For memory utilization, the values for average, P90, P99, P999, and P9999 are approximately 1.5%, 15%, 34%, 93%, 96%. The maximum CPU utilization at P9999 reaches 98%, which is about 20 times the average. For memory utilization, the P9999 is 64 times the average. This observation is further confirmed by the VM service usage shown in Table 1. For instance, P50 VMs account for only 0.53% of the CPS created by P9999 VMs, highlighting that most service usage comes from a small number of heavy users. A similar observation has been reported in [25, 41]; this paper provides an in-depth root cause analysis based on specific cloud services.

2.2.2 Insights. **CPS limited by CPU on slow path.** Each flow's first packet triggers logic execution on the vSwitch's slow path, involving queries to multiple rule tables and consuming substantial CPU resources. This limits CPS capability due to CPU constraints. In our design, establishing a new connection requires querying at least five tables (including ACL, QoS, policy, VXLAN routing, and vNIC-Server mapping) [34]. If advanced features such as policy-based routing, traffic mirroring, or flow logging are enabled, up to 12 tables need to be queried. Even with a powerful SmartNIC, its thermal design power (TDP) limits its computational capacity. Additionally, the SmartNIC needs to support various hypervisors, such as storage networks [39], container networks [36], and VMMs [55], leaving only a few CPU cores to virtual networks. We have optimized our SmartNIC's capacity to O(100K) CPS, but it still cannot meet the CPS demands of a single vNIC for some heavy users. In our cloud, some users' DNS servers and L7 load balancers handle large volumes of client requests, generating massive short-lived connections that lead to high CPS.

#Concurrent flows limited by memory on fast path. Each session table entry requires recording bidirectional flows (including the 5-tuple, VPC ID, and pre-actions) and session states (*e.g.*, the FSM, aging time, and statistics), occupying O(100B) in total. However, the memory on SmartNICs is extremely limited, typically a few tens of GB, and is shared among various functions such as storage and computation. Less than half is allocated for networking, with most of it used for packet buffering, leaving only hundreds of MB to a few GB for the session table. There are reports that VMs cannot handle enough concurrent flows during shopping festivals when serving hundreds of millions of users. Some L4 load balancers maintain persistent connections for each client, which can cause session table bloat. Even with normal connections remaining in the table for an average of 8s [25], massive concurrent flows can accumulate under high CPS.

#vNICs primarily limited by memory on slow path. Each vNIC's rule tables occupy memory on the slow path. Our production data show that most vNICs require 5.5-10MB of memory. Given the limited memory of a few GB, a single SmartNIC can support only a few hundred vNICs, far from sufficient for a middlebox instance serving thousands of tenants [46, 52]. In extreme cases, a single vNIC can consume O(100MB) of memory to store its rule tables. Some vNICs may need to store O(100K) vNIC-Server entries (mapping vNICs to their servers) to communicate with VMs in large VPCs, consuming over 200MB of memory. In this scenario, #vNICs that a single SmartNIC can support is drastically reduced to just a few. While vNICs consume CPU resources for management, maintenance and processing, #vNICs is primarily limited by memory. The rise of container and serverless services [4, 8, 13, 19] has led to high demands for vNIC provisioning [43, 50].

2.2.3 Key takeaway. **Potential to offload.** The session table and rule tables consume memory, while rule table lookups use CPU. They are the primary factors exhausting memory and CPU, respectively. Given that most SmartNICs have very low resource utilization, there is an opportunity to offload the tables and table lookups to idle SmartNICs, which drives the design of Nezha.

2.3 Potential Solutions and Their Issues

2.3.1 Rolling out high-capacity SmartNICs. As the load is extremely imbalanced, provisioning every server SmartNIC for peak load is not cost-effective [25]. One option is to upgrade select SmartNICs.

Table 2: Solutions leveraging remote resource pool.

	Stateful NF support	Prevent remote state maintenance	Without introducing additional hardware
Sailfish	×	✓	×
Sirius	✓	×	×
Tea	✓	×	×
Nezha	✓	✓	✓

However, as reported in [25], hotspot SmartNICs can occur on any server, inevitably requiring the migration of high-demand VMs to those few servers with upgraded SmartNICs. Based on our experience, live migration of high-load VMs often causes connection interruptions (see Appendix A Fig. A1). Moreover, our servers have diverse configurations to meet the varying needs of tenant VMs, such as general-purpose, compute-intensive, GPU-accelerated, or memory-optimized VM instances [15]. VMs of a certain type can only migrate between servers with the corresponding configuration to avoid performance degradation. As a result, we must provision upgraded SmartNICs for each server configuration.

2.3.2 Leveraging host resources. Another approach is to extend SmartNIC capabilities by utilizing abundant host resources for those actions implemented in software. However, this is unsuitable for bare-metal scenarios, where all host resources should be allocated to the tenant. In addition, utilizing host resources presents a few issues: a) reduced tenant-sellable resources; b) the low performance of software vSwitches; c) potential isolation risks between software vSwitches and tenant VMs.

2.3.3 Leveraging remote resource pool. To enhance the network capacity of a vNIC, solutions in [25, 33, 41] use remote resource pools for offloading, but no solution meets all of the following features (summarized in Table 2).

Stateful NF support. Sailfish [41] offloads stateless NFs (*e.g.*, VXLAN routing) to Tofino. However, with the limited on-chip memory, it cannot handle stateful NFs at cloud scale.

Prevent remote state maintenance. Stateful remote resource pools face challenges in synchronizing states across replica nodes for failure tolerance and load balancing.

For failure tolerance, Sirius and Tea adopt a primary-backup strategy in remote resource pools, requiring state synchronization between replica nodes to prevent inconsistencies [25, 33]. For example, Sirius ping-pongs packets that change states between the primary and secondary cards to achieve in-line replication of connection states. For new connections, such in-line state replication limits the achievable CPS to only half of the total capacity of the two cards.

Sirius distributes the connections of a single vNIC across multiple cards for load balancing [25]. It hashes flows into a fixed number of buckets and assigns them to different processing cards. The bucket assignment changes to move load across cards. New flows are naturally assigned to the new card, while existing flows remain with the old one until most have completed. This minimizes state transfer, which is only necessary for long-lived flows. It is an elegant solution, but still adds complexity and performance issues. For example, coordination between buckets or cards is required to implement VM-level rate limiting, which is a complex distributed rate-limiting problem [26, 45], or it has to rely on the on-host card.

Without introducing additional hardware into system. Tea leverages the DRAM on servers to address the limited memory capacity of Tofino switches, but still introduces a new component (DRAM servers) to the system. Sailfish uses Tofino to develop a cloud gateway, while Sirius constructs a high-performance Pensando DPU pool. Both offer high performance while being cost-effective. For example, with Sirius, the server SmartNICs only need to be capable of handling average load. The excess load is steered to the powerful Pensando DPUs, which can be provisioned with smaller peak-to-average ratios. However, they still require significant investment in additional hardware, development and maintenance, and are tied to vendor-specific chips for performance, resulting in upfront and ongoing CAPEX/OPEX.

3 Nezha Architecture

3.1 Decoupling State from Rule/Flow Table

We leverage the idle SmartNICs on other servers to offload the high-demand vNIC in the local SmartNIC. However, this may lead to complex state synchronization across the involved SmartNICs when sharing the load or handling failures. To eliminate the synchronization issue, we propose a novel approach to decouple state from rule tables/cached flows and keep state locally in *one* copy (Fig. 5).

Stateful NF processing requires three inputs: *packets*, *rules*, and *states*. Traditionally, cloud providers store and manage rules and states together (Fig. 5). For example, VFP keeps them in the local vSwitch [28], while Sirius stores rules and states of high-demand vNICs in the remote DPU pool [25]. Although all three inputs need to be present at the moment of packet processing, this does not imply that states and rules cannot be stored in a disaggregated manner.

We validate the feasibility of decoupling with a strawman architecture: storing states on a local SmartNIC while keeping rule tables and cached flows on remote SmartNICs. Using stateful ACL as an example, we demonstrate the equivalence of processing results under this separation architecture. The rule table defines packet processing pre-action for a certain prefix (*i.e.*, drop or accept) and the state records the direction of the first packet (*i.e.*, TX or RX). The final packet processing action enforces RX packet acceptance when the first packet direction is TX, even if the pre-action is "drop" for any incoming traffic. However, when the rule table and cached flows are moved to the remote while states are kept locally, the following steps are required.

When ingress (RX) packets first reach the remote vSwitch to obtain the ACL table lookup results for this flow (*e.g.*, TX: accept; RX: drop), they are unaware of the first packet direction. To address this, the remote vSwitch forwards the packet to the local vSwitch, piggybacked with the preliminary packet drop/accept decision (pre-action) according to its 5-tuple. The local vSwitch can then make the final decision based on the first packet direction recorded in the state (see the blue data flow in Fig. 5). The egress (TX) packets, however, first reach the local vSwitch to obtain the first packet direction but are unaware of the ACL rules. Similarly, the local vSwitch forwards the packet, piggybacked with the first packet direction (state), to the remote vSwitch, where the decision of dropping or accepting is finalized by querying the pre-actions (see the red data flow in

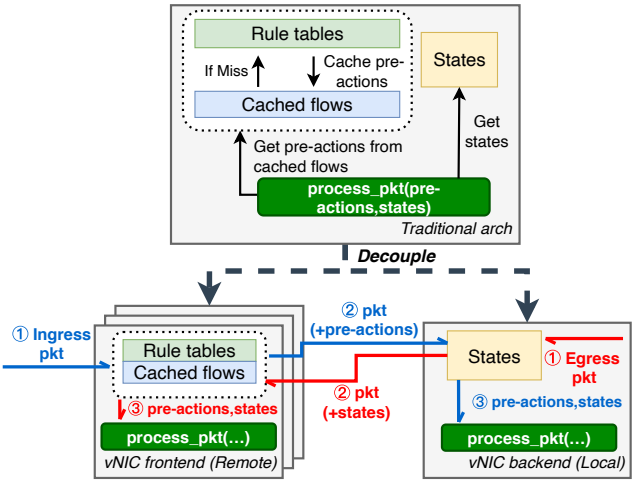


Figure 5: Nezha architecture.

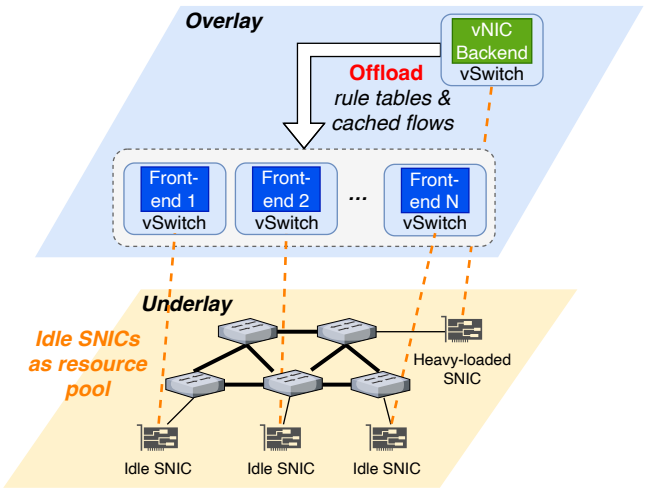


Figure 6: Reuse of existing SmartNICs as resource pool.

Fig. 5). This example demonstrates the feasibility of the separation architecture decoupling state from rule/flow tables, which allows Nezha to reuse existing SmartNICs as a resource pool (see Fig. 6).

3.2 Architecture Overview

3.2.1 vNIC Backend and Frontend . Nezha offloads the stateless rule tables and cached flows of the high-demand vNICs to a remote pool of idle SmartNICs on other servers while maintaining the states locally (see Fig. 5 and Fig. 6). We configure the rule tables on the remote via the controller. The cached flows are regenerated on the fly by querying the rule tables. The node that manages the vNIC states locally is called vNIC backend (BE), while the remote node is referred to as vNIC frontend (FE).

Since the two inputs for packet processing (*i.e.*, rules/flows and states) are stored separately in FE and BE, Nezha uses *packets* to carry the information from one end to the other, bringing the inputs together for processing. As shown in steps ① and ② of the red data flow in Fig. 5, for egress (TX) packets sent from the VMs, the BE encapsulates its local states into the packet's outer header (*e.g.*, NSH [44]) and transmits them to the FE. Subsequently, the FE processes the packets based on the pre-actions recorded in the local cached flows and the states carried in the packet, as shown in step ③. If there is a flow cache miss, a rule table lookup will be performed. The ingress (RX) packets, as depicted in the blue data flow in Fig. 5, are first directed to the FE via route configuration. The FE then encapsulates the queried pre-actions into the outer header and forwards the packets to the BE, where the packets loaded with the pre-actions are processed using the locally maintained states.

In general, we aim to offload packet processing to the remote as much as possible, thereby freeing up local resources. However, we do not offload the processing of stateful NFs (*i.e.*, process_pkt()) entirely to the remote FE. For RX packets, Nezha chooses to have the local BE handle the processing. This is because offloading the processing of RX packets to the FE would introduce multiple routing steps between the FE and BE: the packets first obtain the pre-actions from the FE, then fetch the states from the BE to be processed at the FE, and finally are delivered to the VMs via the BE. By keeping process_pkt() in both FE and BE, Nezha ensures that the packet processing for both RX and TX adds only one extra hop in the end-to-end transmission. The extra hop does not significantly impact bandwidth or latency. The bandwidth overhead is negligible in comparison to the underutilized bandwidth in data centers. The latency increase of a few tens of μs is also imperceptible, as most applications have end-to-end latencies in the tens to hundreds of ms [46]. Without Nezha, the vSwitch with excess load would significantly degrade end-to-end latency.

3.2.2 Initialization and updating of local states. The rule table is initialized/updated by the controller according to the tenant's intent, while the cached flow is generated through a rule table lookup when a flow cache miss occurs. In our design, when the rule table changes, the associated cached flows are invalidated and deleted, which will be regenerated after subsequent rule table lookups.

Nezha's separation architecture eliminates state synchronization. However, states need to be handled carefully to ensure the correct packet processing. Stateful NFs initialize/update states in different ways. Some rely on the results of rule table lookups (rule table involved), while others utilize the information embedded in packets (rule table not involved). State handling can therefore be divided into two categories based on whether the rule table is involved.

Rule table involved. Flow-level statistics is an example that relies on rule tables for state initialization/update, as the state dictates what statistics to record, which can only be obtained by querying the statistics policy table.

TX workflow. The challenge arises because TX packets first go through the BE and then the FE, making it impossible to determine how to initialize/update the rule table-involved state at the BE. To address this, we leverage designated packets to notify the BE on state initialization or updates. While these notify packets incur additional bandwidth and processing overhead, they are generated infrequently for two reasons. 1) Rule table lookup is executed only upon cached flow misses, which generate the rule table-involved state. Cached flow misses occur only on the first packet or rule table updates. 2) The notify packets are generated only when the state from the rule table lookup differs from that carried by the packet, thereby further reducing the notify packet rate. For example, we

notify the BE to update the state when the flow-level statistics policy for a particular flow changes (*e.g.*, counting bytes in and out is no longer required).

RX workflow. The challenge for the RX packets is that the packet has not yet passed through the BE. The FE cannot determine if the state from the rule table lookup differs from the one in the BE. In other words, it is unclear if the BE's state needs to be updated. This can be resolved by verifying the BE's state. However, we decided not to perform the verification, but to simply notify the BE of the state from the rule table lookup to avoid the additional delays and processing. We encapsulate the state into the outer header of the packet instead of using a separate notify packet.

Rule table not involved. Some stateful NFs initialize and update their state based on the information provided in the packets, without relying on the rule table. For instance, a stateful ACL uses the direction of the first packet (RX/TX) to apply different rules, storing this direction as its state (see the case study in § 5.1).

For some NFs, the challenge is that RX packets may lose information necessary for state initialization/updates after being processed by the FE. For example, FE may replace the outer source IP of the RX packet with its own, causing the loss of information needed for stateful decap [25]. To this end, FE encapsulates in the packet header the information needed for BE to initialize/update the state and forwards the packet to BE. TX packets are not affected, as they are sent directly to BE (see the case study in § 5.2).

3.2.3 Load balancing and active-active fault tolerance. To prevent a single FE failure from causing end-to-end unavailability and to achieve load balancing, FE and BE are in a many-to-one mapping relation, as depicted in Fig. 5.

Hash-based load balancing. ①As FEs only maintain stateless rule tables and cached flows, packets can be processed correctly by any FE without synchronization. Thus, Nezha avoids the need for a complex *consistent hashing* to address hashing inconsistencies when #FEs changes. ②Since the per-session shared state is always stored on the BE that bidirectional flows of the same session must pass through, Nezha can distribute the bidirectional flows across different FEs for flexible load-balancing. Hence, the complex *symmetric hashing* is not needed. ③Although packet-level load balancing improves load sharing, it reduces cache friendliness. Distributing packets of the same flow across multiple FEs not only introduces duplicated rule table lookups but also wastes FE's memory due to multiple copies of the same cached flows. As such, Nezha only does flow-level load balancing by leveraging *Hash(5-tuple)* to distribute flows. Although dynamic FE addition/removal without consistent hashing can lead to cache misses for ongoing flows, the effect is negligible in our design. Since each FE is stateless, handling such cache misses simply requires a re-execution of the rule table lookup on the new FE. This process takes only slightly more than 10 microseconds and therefore has a negligible effect on overall end-to-end latency. On top of that, we do aim to reduce the frequency of elastic scaling (as discussed in Appendix B.2).

Active-active fault tolerance. Active-active and active-passive failover are common failure tolerance strategies. In an active-active system, all nodes can perform tasks, allowing for better utilization of total resources. When a single node fails, only $1/n$ of the traffic is affected in an active-active system, while in an active-passive system, all the traffic is affected. However, active-active failover has drawbacks, including higher deployment costs and increased overhead for maintaining data consistency as the number of nodes grows. Nezha does not have these issues. First, all the FEs are already deployed on running vSwitches, incurring no extra costs for deployment and management. Additionally, since each node is stateless, there is no need for state synchronization. Nezha only needs to configure FEs through the controller when there are rule table changes, leading to low overhead for maintaining data consistency across active nodes. As a result, Nezha keeps all FEs in an active state to achieve high availability.

To clarify, multiple FEs in Nezha are used solely for load balancing and fault tolerance purposes, and packets are not transmitted across multiple FEs during processing. Each FE maintains a complete copy of the rule tables. As a result, all tables are processed within a single FE to generate the pre-actions, without requiring cross-FE table lookups or intermediate packet forwarding between FEs.

4 Nezha implementation

4.1 Deployment Issues

There are a few deployment issues to be addressed.

Seamless switch to/back from Nezha. Hotspots are unpredictable and can happen on any server, requiring timely dynamic offloading vNICs to the remote. When a vNIC no longer needs remote resources, the system should fallback from Nezha to avoid the latency from the extra hop. During the switch, delays in configuration changes taking effect may cause some packets not to follow the expected data flow. For example, Nezha requires all RX packets to query the rule tables at the remote first, but in-flight packets may still be forwarded to the local vSwitch. Since the local vSwitch has offloaded the rule tables and cached flows to the remote, these packets cannot be processed promptly. Rerouting and packet retransmission can eventually get the packet processed, but may negatively impact user experience.

Scale-out/-in remote resource pool. To meet the dynamic demand on network capacity, the remote resource pool needs to support on-demand scaling out/in. In addition, when a vSwitch experiences high load from its local vNIC, Nezha needs to remove the FEs on that vSwitch from the resource pool, prioritizing local services.

Remote pool high availability. Nezha splits the packet processing logic originally executed in the local vSwitch into a collaborative approach involving both remote and local vSwitches. It is critical to ensure the high availability of the remote pool in the event of node failures. The challenge is to quickly and accurately check the health of the remote vSwitch, since there are multiple other hypervisors supported by the same SmartNIC.

4.2 Seamless vNIC Offload & Fallback

4.2.1 Offload to remote. **Strategy for selecting offload vNIC.** To promptly identify the risk of vSwitch overload, each vSwitch periodically reports its resource utilization to the controller. If the controller detects a vSwitch's resource utilization exceeding a threshold, Nezha is triggered. Then, Nezha offloads vNICs in descending order of CPU/memory consumption (depending on which resource triggered Nezha) until vSwitch utilization falls below a safe level.

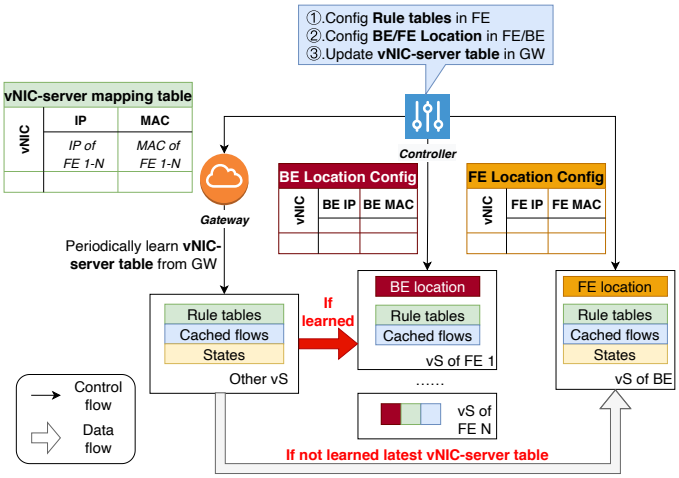


Figure 7: Workflow of remote offloading.

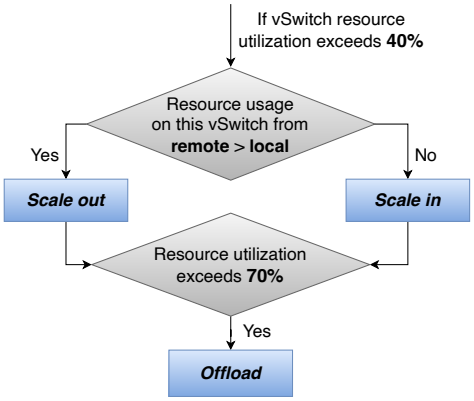


Figure 8: Three methods (*scale-out*, *scale-in*, and *offload*) to mitigate vSwitch resource utilization in Nezha.

Strategy for selecting idle vSwitches. The selection of idle vSwitches as FEs can considerably affect end-to-end latency and maintenance costs. For example, the distance between FEs and BE can result in significant differences in end-to-end latency. Moreover, more FEs require more memory to record their locations in the config. We look for n idle vSwitches under the same ToR switch with similar performance-affecting attributes, such as link bandwidth usage, to ensure a consistent experience between flows within the vNIC (Appendix B.1). If no suitable n vSwitches are found, we continue searching in higher layer switches (*e.g.*, aggregation/core). The goal is to keep n at a minimum without creating bottlenecks. In production, we initialize n to 4 (as discussed in Appendix B.2).

Workflow of user-transparent offloading. After the above selections, the following two stages are triggered.

Dual-running stage: We maintain necessary packet processing information in both the FE and BE to ensure proper packet processing during this stage. As shown in Fig. 7, the controller performs three tasks in this stage. ①The controller configures the vNIC rule tables in all selected FEs. ②To enable packet forwarding between FE and BE, configure the BE/FE location in the vSwitch of the FE/BE. ③To enable other vSwitches to send packets directly to the FE instead of the BE, the controller updates the vNIC-server table with the IP and MAC of the server where the FE is located.

In practice, due to the large size of the vNIC-server table (global routing table) and the fact that most vSwitches only require a small subset of its entries, we configure the table at the gateway and enable vSwitches to learn from it on demand [27, 37, 50]. However, this means that the vSwitch cannot guarantee that its local vNIC-server table is the latest, and thus it still sends packets directly to the BE, as shown by the gray data flow in Fig. 7. To ensure that packets sent directly to the BE are properly processed in a timely manner, we retain the rule tables and cached flows on the BE for a period. When the latest vNIC-server table is learned, packets destined for the offload vNIC will be forwarded to its FE, as shown by the red data flow in Fig. 7.

Final stage: The dual-running stage does not last long. With a learning interval of 200ms for vSwitches in our cloud, it takes no more than 200ms for all vSwitches to obtain the latest vNIC-server table. Considering the transmission delay of in-flight packets, we can delete the rule tables and cached flows on the BE after $200ms + RTT$, entering the final stage.

4.2.2 Fallback to local. For vNICs that no longer need remote resources from FEs, Nezha supports fallback to local to avoid the additional latency caused by offloading. To achieve that, the controller periodically monitors the total resource consumption of each offload vNIC on the FEs. A fallback is triggered only when the controller estimates that the local vSwitch's total resource utilization remains below a safe level, taking into account the additional consumption caused by the fallback. Like offload, fallback consists of dual-running stage and final stage. The only difference in the fallback dual-running stage is that the address in the vNIC-server table corresponds to the BE address instead of the FE address, allowing packets destined for the vNIC to be sent directly to the BE.

4.3 Remote Resource Scale-out/-in

To prevent sudden traffic surges from overloading vSwitches, Nezha continuously monitors the resource utilization of vSwitches hosting FEs. When the resource utilization of these vSwitches triggers an alert (exceeding a certain threshold), scaling out/in is activated. When the alert is triggered by excessive resource usage from offloading, Nezha scales out more FEs for load sharing. If the alert is triggered by excessive resource usage from local traffic, Nezha scales in FEs by removing all FEs on this vSwitch to prioritize resources for local traffic. The scale-in may trigger subsequent scale-out on other vSwitches if more FEs are needed to accommodate the offloading. Notably, unlike the common cloud elastic scaling, Nezha does not reduce the number of FEs even if the utilization of the remote resource is low. The reasons are: 1) Low resource utilization does not negatively impact packet processing performance; 2) Reducing FEs may require flows to re-execute rule table lookups, causing delay; 3) The vSwitch still needs to process packets for its local vNIC, so deleting FEs will not reduce costs (Appendix B.2).

Compared to offloading, scaling out/in does not cost as much. We therefore set the trigger threshold for scaling out/in lower than that for offloading. An example to show the relationship between scale-out, scale-in and offload is presented in Fig. 8.

Workflow of scaling out. After triggering scale-out, the controller follows the strategy in §4.2.1 to select idle vSwitches. To enable the new vSwitches to share load, the controller performs the following tasks: ①Configure rule tables and BE locations in the new FEs. ②Insert the location hosting these new FEs into the BE's *FE location config* and the gateway's *vNIC-server table*. Nezha does not require consistent hashing, and thus cannot ensure that the same flow is always directed to the same FE when the number of FEs changes. However, we only need a rule table lookup if there is a flow cache miss.

Workflow of scaling in. After triggering scale-in, the controller conducts the following tasks to let the vSwitch focus on its local vNIC traffic: ①Confirm which BE's FE is hosted on this vSwitch. ②For each BE and FE pair, delete this vSwitch's location from the BE's *FE location config* and from the gateway's *vNIC-server table*. Due to the delay in the new configurations taking effect, to ensure that user services are not interrupted, the rule tables and the BE location config of these FEs will be temporarily retained for a short period (no longer than the learning interval + RTT).

4.4 Detection and Failover of FE Failures

Crash detection. Either the hardware or software of the SmartNIC can fail, leading to a crash of the hosted FEs. To quickly detect FE crashes, we use a centralized monitoring module to conduct health checks on all vSwitches hosting FEs (*e.g.*, ping polling). Since there are only a few VMs requiring offloading, the monitoring targets are limited, keeping detection overhead low. To ensure that the monitoring reflects the vSwitch's health rather than that of the other hypervisors on the SmartNIC, the monitoring module sets a specific destination port for probes, with *flow direct* rules configured on all SmartNICs to forward these packets directly to the vSwitch's virtual functions (VF in Single Root I/O Virtualization) [21]. The monitoring module needs a mechanism to detect the link connectivity between BE and FE, and in practice we encountered false alarms of faulty FEs (as discussed in Appendix C).

Failover. When the vSwitch is unreachable via multiple pings, it should be deleted immediately from all BE's FEs, following the scale-in logic in §4.3. However, scale-in may increase resource usage of remaining FEs, posing an overload risk for the vSwitches hosting them. To this end, we will maintain a minimum of 4 FEs. If one of the 4 FEs crashes, we will delete the faulty FE and add a new one. If there are more than 4 FEs (say, 6) and one of them crashes, we will only delete the faulty FE, and whether to add a new FE should be independently assessed based on the scale-out logic in §4.3.

5 Case Study

5.1 Stateful ACL in Nezha

Stateful ACL offers advanced connection-based access control capabilities. For example, even if an ACL blocks incoming traffic, it must allow responses to connections initiated by the local VM. To implement the stateful ACL, we use state to record the direction of the first packet of the session (*i.e.*, TX or RX). The pre-action of bidirectional cached flows records the ACL table lookup result for the corresponding direction (*i.e.*, drop or accept). The final packet processing decision needs to be generated by combining both the pre-action and the state. For example, if the pre-action for a RX packet is "drop" and for a TX packet is "accept", and the state is TX, then the final action for both RX and TX packets will be "accept". If the state is RX, the final action for the RX packet will be "drop", indicating an unsolicited flow.

To simplify the explanation of stateful ACL processing logic in Nezha, the following discussion excludes the impact of other NFs on packet processing.

TX workflow. For a TX packet, the BE queries the recorded state, which may be TX, RX, or empty. If it is empty, the state is initialized to TX. After that, the packet carries the state to the FE. Upon receiving it, the FE queries the cached flow for the pre-action. Finally, the FE executes the same code as before deploying Nezha to generate the final action based on the state and pre-action. However, if the final action at the FE is "drop", the BE is unaware of this and still retains its state, leading to memory waste. To this end, the BE applies a shorter aging time to the states for establishing sessions, removing incomplete sessions and thus reducing potential memory waste (see the SYN flood issue in §7.3).

RX workflow. The RX packet cannot determine the direction of the first packet. The FE queries the cached flows and encapsulates the pre-actions in the packet. For example, these pre-actions can be "RX: accept; TX: accept". When the packet arrives at BE, it queries the state, and the result may be TX, RX, or empty. If it is empty, the state is initialized to RX to record the first packet direction. Finally, BE executes the same code as before to generate the final action.

5.2 Stateful decapsulation in Nezha

Stateful decapsulation is commonly used in load balancers (LB). LB distributes user packets to multiple real servers (RS). To enable the RS to send response packets back to the LB, the RS' vSwitch needs to record the overlay source IP (*i.e.*, LB address) when decapsulating the packet overlay header, which is commonly referred to as stateful decap [25]. This allows the RS to send the response packet to the LB. If stateful decap is not conducted and the LB is required to maintain the client's address unchanged, the RS will send the response packet directly back to the client since the inner source IP in the packet received by the RS is the client's address. However, since the client has only established a TCP connection with the LB, the response from the RS will be dropped.

For stateful decap, the final action of the TX packet is determined by the state, *i.e.*, encapsulating the overlay header of the TX packet based on the recorded IP in the state. Therefore, the TX packet needs to carry the IP recorded in the state to the FE, which will perform the overlay encapsulation. The RX packet only needs to carry the overlay source IP to the BE for state initialization (as described in §3.2.2).

6 Evaluation

6.1 Experimental Settings

Small-scale testbed. We first set up a small-scale testbed to evaluate Nezha without the presence of other tenants to skew the measurements. The testbed consists of hundreds of servers, each equipped with an in-house developed SmartNIC featuring 2x100Gbps NICs, powered by a combination of CPU and FPGA. The vSwitch is allocated 8 CPU cores and 10GB memory. The Client and Server VMs are hosted on different servers to avoid resource contention,

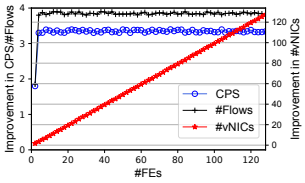


Figure 9: Performance gain under different #FEs.

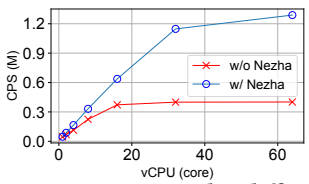


Figure 10: CPS under different #vCPU cores in VM.

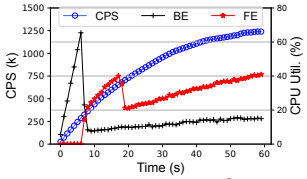


Figure 11: CPU utilization during offloading/scaling.

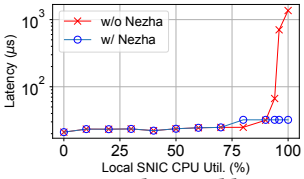


Figure 12: End-to-end latency with/without Nezha.

with each VM equipped with a 64-core Intel Xeon Platinum 8369B CPU and 128GB memory. Other servers serve as a remote resource pool. The results in §6.2 are from the testbed.

Production deployment. Nezha has been deployed on Alibaba Cloud since 2024 and is available to cloud middleboxes and major customers who have extreme demands for network capabilities. Nezha has been released to all vSwitches in several major regions, and these vSwitches can be used as remote resource pools. However, only the allowed vNICs can leverage Nezha to achieve remote offloading. The results in §6.3 and §6.4 are collected from production regions.

6.2 Performance in Testbed

6.2.1 Performance gain with #FEs. Figure 9 shows the performance gain of Nezha with different #FEs for remote offloading, with FE auto-scaling disabled.

CPS. Netperf TCP_CRR [22] is used to simulate a traffic pattern that primarily consists of short connections requiring high CPS. As shown in Fig. 9, when the #FEs is less than or equal to 4, the CPS improvement increases with #FEs. However, when the #FEs exceeds 4, the CPS improvement stays around 3.3X, indicating a performance bottleneck, which is the user virtual machine (see §6.2.2).

#vNICs. Each vNIC consumes memory to store its rule table. With limited local memory, a vSwitch cannot provide a large number of vNICs for VMs or containers. By utilizing remote memory to store the rule table, Nezha can significantly increase #vNICs, as shown in Fig.9. The improvement is proportional to #FEs. However, there is a potential bottleneck to the improvement. The local vSwitch requires 2KB memory to store BE data such as FE location information and some essential metadata to support functionalities that cannot be offloaded remotely. When #vNICs becomes too large, it can exhaust the memory released by storing the vNIC rule table remotely, leading to a bottleneck. In production, the rule table consumes at least 2MB memory; therefore, in theory, Nezha can improve #vNIC by 1000X (=2MB/2KB).

#Concurrent flows. Because Nezha offloads cached flows to the remote, the vSwitch can utilize the released memory to maintain additional states, thus increasing #concurrent flows. However, the state still requires local memory for maintenance. Thus, the local memory becomes the bottleneck in increasing #concurrent flows. As shown in Fig. 9, when the #FEs exceeds 4, the improvement stays around 3.8X.

6.2.2 CPS improvement with #vCPU cores in VM. Without Nezha, the vSwitch CPU is the bottleneck for CPS (Fig. 2). With Nezha, the VM CPU now becomes the bottleneck (Fig. 9). Figure 10 shows the impact of #vCPU cores in the VM on CPS. Theoretically, with Nezha, there are sufficient remote resources, enabling the CPS performance to grow with #vCPU cores. However, as shown in Fig. 10, the growth of CPS does not always correlate with #vCPU cores. This is due to the processing bottlenecks in the VM kernel (such as kernel locks and the limits on manageable connections), which hinder the increase in CPS.

6.2.3 CPU utilization during offloading/scaling. In Fig. 11, we increase the CPS of a single vNIC through scripts to trigger remote offloading and FE scaling out. The figure shows the vSwitch CPU utilization of the BE and the average vSwitch CPU utilization of the FEs during this process. The BE vSwitch CPU utilization increases rapidly with the CPS. Once the utilization exceeds the threshold, remote offloading to 4 FEs is triggered. After that, the BE vSwitch CPU utilization dropped quickly from 70% to about 10%, indicating that Nezha can effectively reduce the CPU utilization of the local vSwitch. The CPU utilization stays low, as the BE still needs to perform some tasks such as stateful NFs for RX traffic and encapsulating state into TX packets. When the average vSwitch CPU utilization of the FEs exceeds 40%, FE scaling out is triggered (increasing the #FEs to 8), which significantly decreases the vSwitch CPU utilization of FEs.

6.2.4 End-to-end latency after remote offloading. To assess the impact of introducing an additional hop with Nezha on end-to-end latency, we measure the latency both with and without Nezha. We adjust the packet rate of a single flow, with the CPU utilization of the vSwitch (without Nezha) on the x-axis and the latency (with/without Nezha) at this traffic scale on the y-axis, as illustrated in Fig. 12.

It shows that with CPU utilization below 70%, the latency of both cases is identical because remote offloading has not been triggered. At around 80% CPU load, the latency is slightly increased due to the additional hop introduced by remote offloading. However, for most cloud services, end-to-end latency typically remains within tens of milliseconds [46]. The additional latency of less than $10\mu s$ introduced by Nezha is negligible. As the CPU load continues to increase, the latency without Nezha deteriorates rapidly due to the local vSwitch becoming overloaded and unable to process packets promptly. In contrast, with Nezha, the end-to-end latency remains largely unaffected by increasing traffic, highlighting the effectiveness of offloading.

6.3 Performance on Cloud

6.3.1 Performance improvement with middleboxes. We evaluate Nezha using three popular middleboxes: Load Balancer (LB) [18], Network Address Translation (NAT) gateway [16] and Transit Router (TR) [23] — a core cloud router that enables network communication between VPCs, between VPCs and on-premises networks, and across regions.

Table 3: Performance gain with three middleboxes.

	CPS	#vNICs	#Concurrent flows
Load-balancer	4X	> 40X	5.04X
NAT gateway	4.4X	> 40X	50.4X
Transit router	3X	> 40X	15.3X

Table 4: Completion time for activating offloading.

Avg (ms)	P90 (ms)	P99 (ms)	P999 (ms)
1077	1503	2087	2858

CPS. As shown in Table 3, the CPS improvements for the three middleboxes range from 3X to 4.4X. Since the CPS of the three middleboxes varied before Nezha took effect but all reached around 1.3M afterward, the performance gains differ. Specifically, the more complex the rule table lookup, the lower the CPS without Nezha, leading to a higher performance gain with Nezha. TR has the simplest rule table lookup as it bypasses the ACL rules, resulting in the least performance gain in CPS. In contrast, LB and NAT need to perform ACL lookups, leading to relatively higher performance gains of 4X and 4.4X, respectively.

#vNICs. With Nezha, the improvement in #vNIC is proportional to the rule table size. Cloud middleboxes often have a wide range of network features and access control rules. Their rule table sizes are typically much larger than 2MB. For example, the rule table sizes of LB, NAT and TR are generally O(100MB). The memory released by storing rule tables remotely can support O(10K) BEs. However, in production, a single VM does not require such a large number of vNICs to avoid issues such as excessive blast radius or resource exhaustion from servicing too many users. Our production data indicate that a single VM generally needs to increase #vNICs to O(1K), or by several tens of times. As shown in Table 3, Nezha can increase #vNICs for the three middleboxes by more than 40X in production.

#Concurrent flows. The maintenance of BE data only requires a small portion of the memory released by storing the rule table remotely. We utilize the remaining memory to store the states to further increase #concurrent flows. LB requires a massive number of concurrent flows to maintain long-lived connections with multiple real servers, resulting in a session table larger than those of other middleboxes. Therefore, the improvement increases only from 3.8X (see §6.2.1) to 5.04X (Table 3), but it reaches roughly 30M flows with some more capable server SmartNICs. In contrast to LB, NAT and TR have fewer concurrent flows due to the absence of long-lived connections, improving performance by 50.4X and 15.3X, respectively.

6.3.2 Completion time for activating offloading. Table 4 presents the time taken from the triggering of remote offloading until all traffic is forwarded through the FEs within a cluster over a month. It shows that the average and P99 completion times are approximately 1s and 2s, respectively. This indicates that as long as the vSwitch can manage the current load within 2s, Nezha can help prevent overloading the vSwitch in most cases. The completion time for triggering scaling out/in and fallback is of the same order as they follow similar procedures to take full effect.

6.3.3 Daily overload occurrence. Figure 13 shows the daily overload occurrences before and after Nezha's deployment. Nezha can mitigate over 99.9% of vSwitch overloads caused by high demands on CPS and #concurrent flows and completely prevent overloads due to excessive #vNICs. A small number of overloads for CPS and #concurrent flows still occur because Nezha's remote offloading does not take full effect immediately (with a P999 completion time of about 2.8s). Once remote offloading is in full effect, these overloads are eliminated. For #vNICs, there are no such issues as vNIC rule tables can be directly created on the FEs.

Table 5: Deployment costs of Sailfish/Nezha.

	Sailfish	Nezha
Hardware development	100 person-month	0
Software development	48 person-month	15 person-month
Extra human effort for iteration	20 person-month	0
Time required to scale out	1 ~ 3 months	1 ~ 7 days

6.3.4 Response time for failover. With automated anomaly monitoring and failover mechanisms, traffic can be quickly redirected to other healthy FEs. Figure 14 illustrates the average packet loss rate at the region level. When an FE crashes, the loss rate experiences a surge lasting approximately 2s. Based on the deployment experience from Alibaba and Google, many customers are not perceptibly impacted by brief outages lasting seconds [51]. vSwitch failure will cause VM failure. The probability of vSwitch failure is therefore lower than that of VM failure. Major cloud providers typically offer VM instance-level SLAs (Service Level Agreement) exceeding 99.5% (*e.g.*, AWS EC2: 99.5% [2], GCP Compute Engine: 99.95% [14], Alibaba Cloud ECS: 99.975% [3]). We expect the availability of vSwitch to be significantly higher than that and consider this probability of transient packet losses acceptable. Note that without Nezha, packets exceeding the processing capacity of the local vSwitch would otherwise be completely discarded.

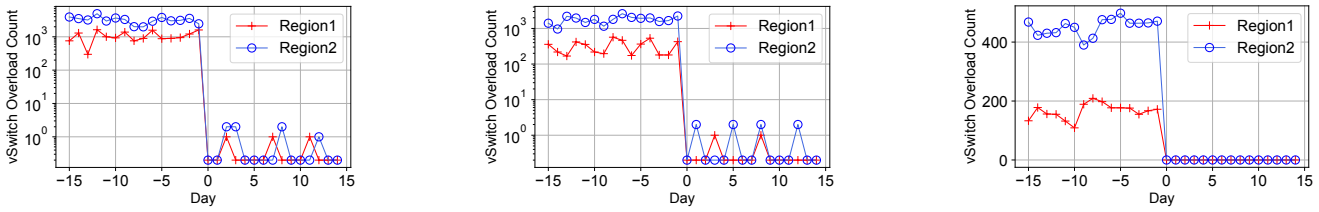
6.4 Deployment Cost

To highlight Nezha's advantage of not introducing new devices, we compare its deployment costs to Sailfish [41], which represents solutions requiring new devices [25, 33].

Introducing new hardware requires substantial human effort for tasks such as chip selection, design, prototype testing, security assessment, and performance optimization. As shown in Table 5, Sailfish needs about 100 person-month (P-M) in hardware development. As Sailfish requires complete functionality development for a new device, its software development cost is about 48 P-M. Nezha utilizes existing SmartNICs and modifies less than 5% of the existing vSwitch code, resulting in only 15 P-M for software development.

After being deployed in production, Sailfish needs ongoing iterations to accommodate new features, improve performance, fix bugs, *etc.* Approximately 20 P-M are allocated for these iterations. The slight modification to the vSwitch code allows Nezha to fully reuse the existing vSwitch team. Sailfish requires additional electricity to power a new network device, whereas Nezha introduces only a slight increase in power consumption for reusing an idle SmartNIC as an FE, since the idle SmartNIC is already powered even when unused. Nezha does, however, introduce additional BE-FE traffic for the purpose of load sharing. This can be accommodated by modern datacenter networks (*e.g.*, 100Gbps+ links), which are provisioned with significant headroom to avoid congestion.

Deploying Sailfish to a new region or scaling out requires selecting a data center, finding suitable racks and setting up equipment.



(a) CPS (b) #Concurrent flows (c) #vNIC

Figure 13: vSwitch daily overload occurrence before/after using Nezha in two regions.

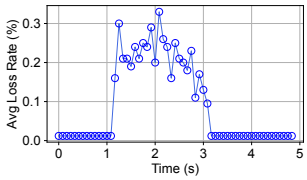


Figure 14: Impact of FE crash on packet loss rate.

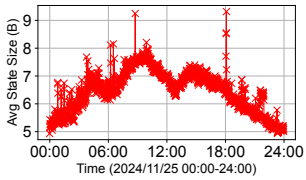


Figure 15: Avg state size in a region.

These tasks take at least one month, with timelines extending to three months if device procurement is involved. Nezha only needs to launch the new version to the vSwitches, with a cluster-level gray release typically taking 1 to 7 days, depending on the cluster size.

7 Experience

7.1 Potential to Increase #Concurrent Flows

A fixed state size prevents memory fragmentation, but can lead to significant memory waste. In extreme cases, a flow that does not require a stateful NF may have an empty state but still occupies 64B of memory. The average state size ranges from 5B to 8B, as illustrated in Fig. 15. Since most state sizes are smaller than the allocated memory, using variable-length states could further improve #concurrent flows. With an average state size of 8B, the improvement could be up to 8X (64B/8B).

7.2 New Capabilities Introduced by Nezha

With its ability to redirect traffic, Nezha enables several new capabilities. For example, it can redirect traffic to upgraded or bug-free vSwitches where FEs reside. For VM live migration, it can seamlessly redirect traffic to the new BE by quickly updating BE location config on FEs.

Flexible new feature release. In the traditional architecture, upgrading all vSwitches is needed to enable a new feature for all users. Fully upgrading tens of thousands of vSwitches in a region is labor-intensive and time-consuming. With Nezha, releasing new features has become much easier and more flexible. We only need to upgrade a portion of the vSwitches in each region and offload the vNICs that require the new feature to these upgraded vSwitches. This approach simplifies both testing and deployment.

Cost-effective fault recovery. In the past, bugs have caused issues with inserting cached flows into session table during the release of a new vSwitch version, resulting in packet loss for all vNICs on the affected vSwitches. In the previous architecture, all VMs had to be migrated off the servers for fault recovery. With Nezha, vNICs can be offloaded to bug-free (older version) vSwitches to avoid local vSwitch failures caused by cached flows. Remote offloading of vNICs is more cost-effective than VM migration, as it eliminates the need for complex tasks that maintain consistency before and after migration, such as state snapshots, pausing and replication. Additionally, our production data show that the completion time and downtime of VM migration increase with the amount of purchased resources (see Fig. A1 in Appendix A). For a VM with 1024 GB of memory, the migration process can take tens of minutes to complete. In contrast, remote offloading takes 2s (P99) to take full effect, independent of VM resource usage, without interrupting the services as long as the VM can handle the current load during this period.

Efficient VM live migration. VM live migration requires creating the VM, copying its state, configuring a new vNIC on the target vSwitch, and updating the global routing tables (vNIC-Server mapping table in our cloud) at the gateway. For rule tables consuming hundreds of MB of memory, configuring the vNIC is time-consuming and can take several seconds. Additionally, since the vNIC-Server mapping table cannot be updated instantly, packets will inevitably be sent to the old address for around tens of milliseconds in the cloud [27, 50]. To avoid packet losses, hairpin flows must be enabled on the migration source host [27, 49, 50]. With Nezha, since the vNIC has been offloaded to the remote, we only need to update BE location config on FEs to direct traffic to the new BE, which takes effect in less than 1ms, resulting in efficient VM live migration.

7.3 Optimizations When Deploying Nezha

Response to SYN flood. Local VMs may send a large number of SYN packets, which consume the BE's memory. If these SYN packets are dropped by the FE (due to the rule table not allowing them to be sent), it is a significant waste of BE memory. We set a relatively short aging time for session entries in the SYN state to remove these incomplete sessions.

Packet processing acceleration at BE. Without cached flows, BE cannot perform fast packet processing with exact matches in the fast path. To accelerate packet processing, we insert per-flow processing logic into hardware, which defines header rewrite policies, such as writing the FE's IP to the outer DIP and encapsulating state.

7.4 Support for Massive vNICs on VMs

With Nezha, vSwitch memory is no longer a bottleneck for supporting massive vNICs on a single VM. However, the *bus/device/function* (BDF) number used to identify a device function has become the new bottleneck in deployment. Each vNIC requires a BDF number.

Without SR-IOV [20] or SIOV [53], the BDF number is restricted to the 8 bits in the *bus* field, as the *device* and *function* fields remain fixed for the same physical function. Therefore, a VM is limited to 256 BDF numbers, most of which are allocated to essential functions such as storage, compute and encryption, leaving only a few dozen for vNICs. We address this issue through the following two methods.

I/O device virtualization. With SR-IOV or SIOV enabled on virtual I/O adapters, the *device* field (5 bits) and *function* field (3 bits) can be utilized to add 256 more BDF numbers. The additional BDF numbers can be allocated to increase the number of vNICs that a single VM can have. However, this method relies on support for virtual device standards; for instance, SR-IOV requires virtio version 1.1 or higher.

Child vNIC. In the absence of SR-IOV or SIOV, the vSwitch can bind multiple child vNICs to a single I/O adapter vNIC, using tags (*e.g.*, VLAN) in packets to differentiate their traffic. They use the I/O adapter of the parent vNIC for packet transmission to the VM, where it is up to the application to distinguish traffic from different vNICs using tags. This method also effectively increases the number of vNICs that a single VM can support, although sharing the I/O adapter may lead to bandwidth contention. However, our experience shows that tenants needing a large number of vNICs typically have low bandwidth demands, making the bandwidth contention not a particular issue.

7.5 Handling Load Imbalance in Nezha

While flow-based load balancing, such as five-tuple hashing, is widely adopted for its simplicity and efficiency, it can lead to uneven workloads due to hash collisions or the presence of elephant flows. Nezha incorporates several strategies to effectively mitigate such imbalances.

When workload skew occurs due to hash distribution issues, we can dynamically scale out additional FEs or reconfigure the hash function at the source side to redistribute the traffic more evenly across the FEs.

Although mitigating incast caused by an elephant flow is not the primary design goal of Nezha, the system can intelligently assign the elephant flow to a dedicated FE, allowing the flow to nearly monopolize the resources of a single SmartNIC. This improves the performance of the elephant flow while isolating it from other tenant traffic. If even a dedicated SmartNIC is insufficient, Nezha can further apply sender-side rate throttling via backpressure mechanisms [47] to prevent overload and maintain system stability.

8 Related Work

Sirius [25] is the most relevant work and, to our knowledge, the first effort to offload server SmartNICs to a shared resource pool. Sirius leverages Pensando DPUs [5] to build the shared pool. This eliminates the need to upgrade server SmartNICs, which only need to be capable of handling average load. Excess load is steered to the powerful Pensando pool, which can be provisioned with lower peak-to-average load ratios. As a result, Sirius offers high performance while being cost-effective. However, it does not address the issue that most server SmartNICs are largely underutilized, and introducing high-spec devices into data centers incurs significant upfront and ongoing costs. To replicate state for fault tolerance, Sirius adopts a novel in-line replication method by ping-ponging packets between the replicas. For load balancing, Sirius proposes an elegant solution that hashes flows into a fixed number of buckets and assigns these buckets to different processing locations. When moving load, the bucket assignment changes. New flows are immediately assigned to the new processing location, while existing flows remain with the old one until most have completed, to minimize state transfer. State transfer is therefore only needed for long-lived flows. State transfer or synchronization is challenging [27]. Nezha, however, can largely avoid these issues by decoupling state from stateless rule/flow tables and keeping state locally in one copy.

The idea of offloading or sharing existing resources is not new. There are some interesting works in this area. In CDN, FastRoute [30] and T-SAC [31] redirect traffic from overloaded servers to those with available capacity. FairNIC [32] and S-NIC [56] offer strong isolation guarantees for SmartNICs, mitigating resource contention. Lynx [48] enables NF remote offloading for servers without SmartNICs (such as GPU servers) by utilizing a resource pool of existing SmartNICs. Some solutions utilize high-performance centralized remote resource pools to enhance NF performance. LuoShen [40] integrates CPU, FPGA, and Tofino into a 2U server switch to achieve the desired performance while adhering to the stringent constraints of hardware budget and deployment footprint. Some systems [24, 38, 54] offload specific stateful NFs, such as load balancers, to programmable switches or FPGA; however, limited on-chip memory restricts the traffic volume they can handle. Sailfish [41] offloads stateless NFs to programmable switches, constructing a high-performance cloud gateway. Meanwhile, Tea [33] uses DRAM servers to address Tofino's limited memory capacity for storing extensive per-session state.

9 Conclusion

In today's data centers, individual server SmartNICs struggle to handle peak load, while much of their capacity remains unused. This motivated us to design Nezha, a distributed vSwitch load sharing system that utilizes idle SmartNICs as a resource pool, without introducing new devices. Nezha features a novel design that decouples state from rule/flow tables, shifting stateless tables to remote SmartNICs while keeping state locally in a single copy. This eliminates the need for state synchronization, allowing Nezha to offload most packet processing to the remote while simplifying load balancing across remote nodes and achieving active-active fault tolerance with low cost and high availability. To deploy in production, we implemented Nezha with seamless vNIC offloading and fallback, on-demand remote pool scaling, and timely crash detection and failover. The results from deployment in Alibaba Cloud show that Nezha effectively resolves vSwitch overloads and removes it as a bottleneck.

Ethics. *This work does not raise any ethical issues.*

Acknowledgments

The authors would like to thank the shepherd Hongqiang Liu and the anonymous reviewers for their constructive comments. This work was partially supported by the Key R&D Program of Zhejiang Province, China (2023R5202).

References

[1] 2015. OpenFlow Switch Specification - Version 1.5.1. https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf.

[2] 2022. Amazon Compute Service Level Agreement. https://aws.amazon.com/compute/sla/?nc1=h_ls.

[3] 2023. Alibaba Clous's Elastic Compute Service (ECS) Service Level Agreement. https://www.alibabacloud.com/help/en/legal/latest/elastic-compute-service-service-level-agreement.

[4] 2025. Alibaba Function Compute. https://www.alibabacloud.com/en/product/function-compute.

[5] 2025. AMD Pensando Networking. https://www.amd.com/en/products/accelerators/pensando.html.

[6] 2025. AWS - c8g.48xlarge. https://instances.vantage.sh/aws/ec2/c8g.48xlarge.

[7] 2025. Azure - Dpsv6 and Dplsv6-series. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/general-purpose/d-family.

[8] 2025. Azure Functions. https://azure.microsoft.com/en-us/products/functions.

[9] 2025. Create and manage a Windows virtual machine that has multiple NICs. https://learn.microsoft.com/en-us/azure/virtual-machines/windows/multiple-nics.

[10] 2025. Elastic network interfaces. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-eni.html.

[11] 2025. Enabling ECMP Load Balancing Consistency. https://support.huawei.com/enterprise/en/doc/EDOC1100198830/5cb5bed6/enabling-ecmp-load-balancing-consistency.

[12] 2025. Enabling ECMP Load Balancing Consistency Enhancement. https://support.huawei.com/enterprise/en/doc/EDOC1100352660/87e11860/enabling-ecmp-load-balancing-consistency-enhancement.

[13] 2025. Google Cloud Run functions. https://cloud.google.com/functions.

[14] 2025. Google Clous's Compute Engine Service Level Agreement (SLA). https://cloud.google.com/compute/sla.

[15] 2025. Instance families available for purchase. https://www.alibabacloud.com/help/en/ecs/user-guide/overview-of-instance-families.

[16] 2025. NAT Gateway. https://www.alibabacloud.com/en/product/nat?_p_lc=1&spm=a3c0i.7911826.2564562790.4.6a323870SZ7vLu.

[17] 2025. NVIDIA BlueField Networking Platform. https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[18] 2025. Server Load Balancer. https://www.alibabacloud.com/en/product/server-load-balancer?_p_lc=1&spm=a3c0i.7911826.2564562790.4.6a323870SZ7vLu.

[19] 2025. Serverless Computing - AWS Lambda. https://aws.amazon.com/pm/lambda.

[20] 2025. Single root input/output virtualization (SR-IOV). https://en.wikipedia.org/wiki/Single-root_input/output_virtualization.

[21] 2025. SR-IOV Virtual Functions (VFs). https://learn.microsoft.com/en-us/windows-hardware/drivers/network/sr-iov-virtual-functions--vfs-.

[22] 2025. TCP_CRR. https://hewlettpackard.github.io/netperf/doc/netperf.html#TCP_005fCRR.

[23] 2025. Transit Router. https://www.alibabacloud.com/en/product/transit_router?_p_lc=1&spm=a3c0i.7911826.2564562790.1.6a323870SZ7vLu.

[24] Manikandan Arumugam, Deepak Bansal, Navdeep Bhatia, James Boerner, Simon Capper, Changhoon Kim, Sarah McClure, Neeraj Motwani, Ranga Narasimhan, Urvish Panchal, Tommaso Pimpo, Ariff Premji, Pranjal Shrivastava, and Rishabh Tewari. 2022. Bluebird: High-performance SDN for Bare-metal Cloud Services. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 355–370. https://www.usenix.org/conference/nsdi22/presentation/arumugam

[25] Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, Balakrishnan Raman, Avijit Gupta, Sachin Jain, Deven Jagasia, Evan Langlais, Pranjal Srivastava, Rishiraj Hazarika, Neeraj Motwani, Soumya Tiwari, Stewart Grant, Ranveer Chandra, and Srikanth Kandula. 2023. Disaggregating Stateful Network Functions. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1469–1487. https://www.usenix.org/conference/nsdi23/presentation/bansal

[26] Lilong Chen, Xiaochong Jiang, Xiang Hu, Tianyu Xu, Ye Yang, Xing Li, Bingqian Lu, Chengkun Wei, and Wenzhi Chen. 2024. CMDRL: A Markovian Distributed Rate Limiting Algorithm in Cloud Networks. In *Proceedings of the Asia-Pacific Workshop on Networking* (Sydney, Australia) *(APNet '24)*. Association for Computing Machinery, New York, NY, USA, 59–66. https://doi.org/10.1145/3663408.3663417

[27] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. 2018. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 373–387. https://www.usenix.org/conference/nsdi18/presentation/dalton

[28] Daniel Firestone. 2017. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 315–328. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/firestone

[29] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 51–66. https://www.usenix.org/conference/nsdi18/presentation/firestone

[30] Ashley Flavel, Pradeepkumar Mani, David A. Maltz, Nick Holt, Jie Liu, Yingying Chen, and Oleg Surmachev. 2015. FastRoute: A Scalable Load-Aware Anycast Routing Architecture for Modern CDNs. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 381–394. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/flavel

[31] Qiang Fu, Bradley Rutter, Hao Li, Peng Zhang, Chengchen Hu, Tian Pan, Zhangqin Huang, and Yibin Hou. 2018. Taming the Wild: A Scalable Anycast-Based CDN Architecture (T-SAC). *IEEE Journal on Selected Areas in Communications* 36, 12 (2018), 2757–2774. https://ieeexplore.ieee.org/document/8468187

[32] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C. Snoeren. 2020. SmartNIC Performance Isolation with FairNIC: Programmable Networking for the Cloud. In *Proceedings of the ACM Special Interest Group on Data Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 681–693. https://doi.org/10.1145/3387514.3405895

[33] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. 2020. TEA: Enabling State-Intensive Network Functions on Programmable Switches. In *Proceedings of the ACM Special Interest Group on Data Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 90–106. https://doi.org/10.1145/3387514.3405855

[34] 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 Special Interest Group on Data Communication* (Sydney, NSW, Australia) *(SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 750–763. https://doi.org/10.1145/3651890.3672224

[35] Anthony Liguori. 2018. The Nitro Project–Next Generation AWS Infrastructure. In *Hot Chips: A Symposium on High Performance Chips*. https://www.old.hotchips.org/hc31/HC31_T1_AWS_Nitro_Hot_Chips_20190818-2.pdf

[36] Yunzhuo Liu, Junchen Guo, Bo Jiang, Pengyu Zhang, Xiaoqing Sun, Yang Song, Wei Ren, Zhiyuan Hou, Biao Lyu, Rong Wen, Shunmin Zhu, and Xinbing Wang. 2024. Understanding Network Startup for Secure Containers in Multi-Tenant Clouds: Performance, Bottleneck and Optimization. In *Proceedings of the 2024 ACM on Internet Measurement Conference* (Madrid, Spain) *(IMC '24)*. Association for Computing Machinery, New York, NY, USA, 635–650. https://doi.org/10.1145/3646547.3688436

[37] Biao Lyu, Enge Song, Tian Pan, Jianyuan Lu, Shize Zhang, Xiaoqing Sun, Lei Gao, Chenxiao Wang, Han Xiao, Yong Pan, Xiuheng Chen, Yandong Duan, Weisheng Wang, Jinpeng Long, Yanfeng Wang, Kunpeng Zhou, Zhigang Zong, Xing Li, Guangwang Li, Pengyu Zhang, Peng Cheng, Jiming Chen, and Shunmin Zhu. 2024. POSEIDON: A Consolidated Virtual Network Controller that Manages Millions of Tenants via Config Tree. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 1083–1099. https://www.usenix.org/conference/nsdi24/presentation/lyu

[38] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 15–28. https://doi.org/10.1145/3098822.3098824

[39] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. 2022. From Luna to Solar: The Evolutions of the Compute-to-Storage Networks in Alibaba Cloud. In *Proceedings of the ACM Special Interest Group on Data Communication* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 753–766. https://doi.org/10.1145/3544216.3544238

[40] Tian Pan, Kun Liu, Xionglie Wei, Yisong Qiao, Jun Hu, Zhiguo Li, Jun Liang, Tiesheng Cheng, Wenqiang Su, Jie Lu, Yuke Hong, Zhengzhong Wang, Zhi Xu, Chongjing Dai, Peiqiao Wang, Xuetao Jia, Jianyuan Lu, Enge Song, Jun Zeng, Biao Lyu, Ennan Zhai, Jiao Zhang, Tao Huang, Dennis Cai, and Shunmin Zhu. 2024. LuoShen: A Hyper-Converged Programmable Gateway for Multi-Tenant Multi-Service Edge Clouds. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 877–892. https://www.usenix.org/conference/nsdi24/presentation/pan

[41] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, Enge Song, Jiao Zhang, Tao Huang, and Shunmin Zhu. 2021. Sailfish: Accelerating Cloud-Scale Multi-Tenant Multi-Service Gateways with Programmable Switches. In *Proceedings of the ACM Special Interest Group on Data Communication* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 194–206. https://doi.org/10.1145/3452296.3472889

[42] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martín 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

[43] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and K. K. Ramakrishnan. 2022. SPRIGHT: Extracting the Server from Serverless Computing! High-performance eBPF-based Event-driven, Shared-memory Processing. In *Proceedings of the ACM Special Interest Group on Data Communication* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 780–794. https://doi.org/10.1145/3544216.3544259

[44] Paul Quinn, Uri Elzur, and Carlos Pignataro. 2018. RFC 8300: Network Service Header (NSH). https://datatracker.ietf.org/doc/rfc8300/

[45] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. 2007. Cloud Control with Distributed Rate Limiting. *SIGCOMM Computer Communication Review* 37, 4 (2007), 337–348. https://doi.org/10.1145/1282427.1282419

[46] Enge Song, Yang Song, Chengyun Lu, Tian Pan, Shaokai Zhang, Jianyuan Lu, Jiangu Zhao, Xining Wang, Xiaomin Wu, Minglan Gao, Zongquan Li, Ziyang Fang, Biao Lyu, Pengyu Zhang, Rong Wen, Li Yi, Zhigang Zong, and Shunmin Zhu. 2024. Canal Mesh: A Cloud-Scale Sidecar-Free Multi-Tenant Service Mesh Architecture. In *Proceedings of the ACM Special Interest Group on Data Communication* (Sydney, NSW, Australia) *(SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 860–875. https://doi.org/10.1145/3651890.3672221

[47] Enge Song, Nianbing Yu, Tian Pan, Qiang Fu, Liang Xu, Xionglie Wei, Yisong Qiao, Jianyuan Lu, Yijian Dong, Mingxu Xie, Jun He, Jinkui Mao, Zhengjie Luo, Chenhao Jia, Jiao Zhang, Tao Huang, Biao Lyu, and Shunmin Zhu. 2022. MIMIC: SmartNIC-aided Flow Backpressure for CPU Overloading Protection in Multi-Tenant Clouds. In *IEEE 30th International Conference on Network Protocols (ICNP 22)*. 1–11. https://doi.org/10.1109/ICNP55882.2022.9940340

[48] Maroun Tork, Lina Maudlej, and Mark Silberstein. 2020. Lynx: A SmartNIC-driven Accelerator-centric Architecture for Network Servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 117–131. https://doi.org/10.1145/3373376.3378528

[49] Yi Wang, Eric Keller, Brian Biskeborn, Jacobus van der Merwe, and Jennifer Rexford. 2008. Virtual Routers on the Move: Live Router Migration as a Network-Management Primitive. *SIGCOMM Computer Communication Review* 38, 4 (2008), 231–242. https://doi.org/10.1145/1402946.1402985

[50] Chengkun Wei, Xing Li, Ye Yang, Xiaochong Jiang, Tianyu Xu, Bowen Yang, Taotao Wu, Chao Xu, Yilong Lv, Haifeng Gao, Zhentao Zhang, Zikang Chen, Zeke Wang, Zihui Zhang, Shunmin Zhu, and Wenzhi Chen. 2023. Achelous: Enabling Programmability, Elasticity, and Reliability in Hyperscale Cloud Networks. In *Proceedings of the ACM Special Interest Group on Data Communication* (New York, NY, USA) *(SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 769–782. https://doi.org/10.1145/3603269.3604859

[51] David Wetherall, Abdul Kabbani, Van Jacobson, Jim Winget, Yuchung Cheng, Charles B. Morrey III, Uma Moravapalle, Phillipa Gill, Steven Knight, and Amin Vahdat. 2023. Improving Network Availability with Protective ReRoute. In *Proceedings of the ACM Special Interest Group on Data Communication* (New York, NY, USA) *(SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 684–695. https://doi.org/10.1145/3603269.3604867

[52] Tingting Xu, Bengbeng Xue, Yang Song, Xiaomin Wu, Xiaoxin Peng, Yilong Lyu, Xiaoliang Wang, Chen Tian, Baoliu Ye, Camtu Nguyen, Biao Lyu, Rong Wen, Zhigang Zong, and Shunmin Zhu. 2024. CyberStar: Simple, Elastic and Cost-Effective Network Functions Management in Cloud Network at Scale. In *2024 USENIX Annual Technical Conference (ATC 24)*. USENIX Association, Santa Clara, CA, 227–246. https://www.usenix.org/conference/atc24/presentation/xu-tingting

[53] Yifan Yuan, Ren Wang, Narayan Ranganathan, Nikhil Rao, Sanjay Kumar, Philip Lantz, Vivekananthan Sanjeepan, Jorge Cabrera, Atul Kwatra, Rajesh Sankaran, Ipoom Jeong, and Nam Sung Kim. 2024. Intel Accelerators Ecosystem: An SoC-Oriented Perspective : Industry Product. In *ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA 24)*. 848–862. https://ieeexplore.ieee.org/document/10609705

[54] Chaoliang Zeng, Layong Luo, Teng Zhang, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, Xiongfei Geng, Tao Feng, Feng Ning, Kai Chen, and Chuanxiong Guo. 2022. Tiara: A Scalable and Efficient Hardware Acceleration Architecture for Stateful Layer-4 Load Balancing. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 1345–1358. https://www.usenix.org/conference/nsdi22/presentation/zeng

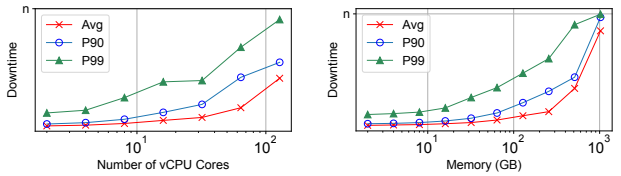
[55] Xiantao Zhang, Xiao Zheng, Zhi Wang, Qi Li, Junkang Fu, Yang Zhang, and Yibin Shen. 2019. Fast and Scalable VMM Live Upgrade in Large Cloud Infrastructure. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 93–105. https://doi.org/10.1145/3297858.3304034

[56] Yang Zhou, Mark Wilkening, James Mickens, and Minlan Yu. 2024. SmartNIC Security Isolation in the Cloud with S-NIC. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 851–869. https://doi.org/10.1145/3627703.3650071

Appendices

Appendices are supporting material that has not been peer-reviewed.

A Additional figures and table



(a) CPU (b) Memory

Figure A1: VM migration downtime with different VM vCPU cores and memory in a region.

A.1 Hotspot distribution in a region

Figure 3 shows the distribution of vSwitch overload due to high demand for three network capabilities (*i.e.*, CPS, #concurrent flows, #vNICs). It shows that vSwitch overloads caused by CPS account for the highest proportion, approximately 61%. In contrast, vSwitch overloads resulting from #concurrent flows and #vNIC are relatively low, accounting for about 30% and 9%, respectively. The reason is that only a small number of tenants require their VMs to serve a vast number of end-users, resulting in a relatively low frequency of needing an extremely large #vNICs. Regardless of the number of end-users, both CPS and #concurrent flows may spike with traffic surges, resulting in a relatively high frequency of vSwitch overload caused by CPS and #concurrent flows.

Tasks offloaded to SmartNICs such as NFs are typically CPU-intensive, and can become memory-intensive if a large number of active flows need to be tracked. This makes the vSwitch's CPU more susceptible to overload compared to its memory, and may explain why CPU overload from CPS occurs more frequently than memory overload from #concurrent flows.

Table A1: Nezha's rule table lookup throughput (Mpps) under different packet sizes and #ACL rules.

Pkt Size (B)	Number of ACL rules					
	0	1	8	64	100	1000
64	6.612M	6.609M	6.333M	5.973M	5.966M	5.422M
128	6.543M	6.455M	6.303M	5.826M	5.702M	5.365M
256	6.415M	6.341M	6.030M	5.430M	5.685M	5.228M
512	5.985M	5.925M	5.455M	5.258M	5.035M	4.762M

A.2 Rule table lookup throughput

The first packet of each new flow requires a rule table lookup, and the CPU cycles consumed for this process are influenced by various factors, including the packet size, the number and size of rule tables, *etc.* In this experiment, we measure the rule table lookup throughput with Nezha under different packet sizes and #ACL rules in the ACL table. As shown in Table A1, with 64B packet size and no ACL rules, vSwitch can handle SYN packets at 6.61Mpps. As the number of ACL rules increases, the throughput gradually decreases, as the complexity and time required for a single ACL table lookup increase with the #rules. Additionally, as the packet size increases, the throughput also gradually declines. This is because, while rule table lookup throughput is theoretically independent of packet size — since it does not rely on packet payload — larger packets increase the time needed to move packets from the NIC to the vSwitch.

B Strategy for selecting idle SmartNICs

B.1 SmartNIC selection

To serve as FEs, SmartNICs can be selected based on underlay hop distance to minimize latency or CPU/memory utilization to ensure sufficient resources. Moreover, since multiple FEs are provided for one BE, resulting in multiple data paths for its flows, we aim to ensure a consistent experience between the flows across these paths. Thus, we should not only choose the SmartNICs with metrics (such as hardware specifications and resource usage) for the best performance, but also ensure that these metrics are similar across the selected SmartNICs. For example, selecting SmartNICs with similar specs, distance to the BE, resource utilization and link loads can effectively prevent significant latency discrepancies between flows of the same BE that may traverse different FEs.

B.2 Decision on #FEs

To ensure load balancing and prevent a single FE failure from causing end-to-end unavailability, there is a many-to-one mapping between FE and BE. However, determining the number of FEs is not straightforward. Although the number of FEs can be dynamically adjusted based on user demand through elastic scaling, Nezha hashes the 5-tuple to assign flows to specific FEs. As a result, flows may be reassigned to different FEs after scaling, and these FEs may not have the cached flows. To minimize the additional overhead from rule table lookups due to cached flow misses, we aim to reduce the frequency of elastic scaling.

Avoid scale-in execution. Since each FE node is deployed on the running SmartNIC, reducing the #FEs does not result in cost savings. Therefore, we prohibit scaling in due to low FE resource utilization, and opt for Nezha fallback when FEs are no longer needed (see §4.2.2).

Mitigate scale-out execution. Scaling out is generally triggered due to an insufficient number of FEs to handle the load. Having too many FEs diminishes the gain and can lead to high maintenance costs. For example, when the #FEs exceeds 4, CPS and #concurrent flows improvements stay around 3.3X and 3.8X, respectively (see Fig.9). To balance out, the initial number of FEs should be set to "the minimum required to satisfy the performance needs of most Nezha users." Additionally, the initial number of FEs should ideally be a power of 2, as many vendors' devices achieve better load balancing with this configuration [11, 12]. This also reduces memory fragmentation, improving storage and access efficiency. Therefore, we set the initial number of FEs to 4.

Production test. To validate our configuration, we conducted a 30-day test in a production cluster of tens of thousands of servers. In the 30 days, there were 2,499 offload events, involving 10,062 FEs. Since the initial value is 4, theoretically, for the 2,499 offloaded vNICs, 9,996 FEs (= $2499 * 4$) would be provisioned initially. Based on the accumulated total of 10,062 FEs, Nezha scaled out 66 FEs (= $10062 - 9996$), suggesting that there were a maximum of 66

scaling out events. Thus, at most only 2.6% (= 66/2499) of the resource pool underwent scaling out. If multiple FEs were scaled to a single resource pool, the ratio of scaled resource pools would further decrease. This indicates that having 4 FEs strikes a balance between performance and the cost of scaling out.

C Centralized FE crash monitoring

C.1 FE-BE link connectivity

Our centralized FE crash monitoring (§4.4) can only monitor the health of the vSwitch, but cannot determine the link connectivity between the FE and the BE. To this end, we implemented periodic mutual pinging between the FE and BE to remove the FE when connectivity issues are detected. However, the mutual ping frequency is much lower than that of the centralized monitoring module, as modern data center networks generally have inherent link fault tolerance (OpenFlow fast failover groups [1]), making complete disconnection between servers rare.

C.2 False positives

In the past, we encountered situations where the centralized monitoring module reported that the majority (or even all) of the FEs were non-functional. Based on our experience, such widespread failures often indicate false positives, potentially caused by bugs in the monitoring module. Therefore, we decided to suspend the process of "automatically removing unresponsive FEs", with manual intervention to verify if the widespread failure was happening.