Poseidon: A Consolidated Virtual Network Controller that Manages Millions of Tenants via Config Tree

Biao Lyu, *Zhejiang University and Alibaba Cloud;* 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, and Zhigang Zong, *Alibaba Cloud;* Xing Li, *Zhejiang University and Alibaba Cloud;* Guangwang Li and Pengyu Zhang, *Alibaba Cloud;* Peng Cheng and Jiming Chen, *Zhejiang University;* Shunmin Zhu, *Tsinghua University and Alibaba Cloud*

https://www.usenix.org/conference/nsdi24/presentation/lyu

This paper is included in the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

POSEIDON: A Consolidated Virtual Network Controller that Manages Millions of Tenants via Config Tree

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*[*], *Shunmin Zhu*[¶†□]

[*]*Zhejiang University* [†]*Alibaba Cloud* [¶]*Tsinghua University*

Abstract

As the cloud rapidly expands in scale, the virtual network controller must manage an increasing number of devices with higher update frequencies. Furthermore, the emergence of cloud-native services has substantially intensified program-triggered updates, leading to more frequent API invocations. To enhance performance and extensibility, we propose POSEIDON, a novel virtual network control framework. Specifically, to reduce operational expenses (OpEx), we have consolidated the common functions of multiple service controllers into a single controller. To manage heterogeneous devices and eliminate the multi-table lookup complexity due to config dependencies, we introduce *Trident*, a tree-based service- and device-independent abstraction, so that config dependency calculation can be replaced by more efficient tree traversal. After deploying POSEIDON on Alibaba Cloud, we observed a 21x increase in the throughput of virtual network configuration tasks, along with a 4.4x decrease in the P99 API processing latency. POSEIDON completes the task of enabling hundreds of Elastic IP addresses (EIPs) 1.8 to 55 times faster than Vendors A and B, both of which are among the top 5 providers [6], for identical network configuration jobs.

1 Introduction

Today's cloud virtual networks are managed by multiple separate controllers [14], which are responsible for configuring, monitoring, and recovering individual service [14, 30, 31] in the cloud virtual networks, such as virtual private cloud (VPC), VMs, *etc*. Managing and configuring large-scale cloud virtual networks are extremely challenging due to the following reasons:

- **Rapid growth of northbound API calls and southbound devices:** The controller needs to process massive number of network configuration API calls which are generated by cloud users, network operators, *etc*. In 2022, our cloud controller processed $\sim$17 trillion API calls per day. To make it worse, the number of devices to be managed also increases dramatically. In our cloud, a single ACL rule change might lead to config updates across $\sim$100,000 servers.
- **Long search chains and large table size:** A virtual network service, such as VPC, usually depends on other services, such as ACL and routing. When configuring virtual networks, the controller needs to query databases associated with the targeted service and its dependent services. Since the number of services increase dramatically over years, the dependencies become so complex such that we need to spend lots of engineering effort to process each API call. In addition, the size of tables in each database increases dramatically as well, which makes the table lookups for API parsing more time-consuming. We observed that due to these two factors, the 90% (P90) tail completion time for handling an API call has nearly doubled over the past year.
- **Cloud-native applications intensify the controller performance requirements:** Cloud-native applications [4] require extremely high throughput for concurrent resource creation/deletion, which demands much higher controller performance compared to traditional console-based network configuration. For instance, to handle surges in user access during peak events, social media applications require the creation of tens of thousands of backends to be completed within an exceptionally short time.
- **High cost of managing separate controllers:** To ensure the agility and iterative development of various services, we build and maintain separate controllers, one for each service (same as Andromeda [14]). In the past few years, with the growth of services, the number of controllers has boomed to over 50, leading to a substantial increase in the costs of managing numerous controllers by separate teams.

Some existing works focus on improving control efficiency and minimizing direct human interaction with network devices [13], with device-independent unified config abstraction/model [10, 16, 20–22], or even automating control decision-making with user intent [17–19, 27]. Andromeda [14] introduces how to improve the performance and scalability of pushing the configurations to Google's virtual network devices. However, they do not touch how to compute

□ Co-corresponding author

the updates of configurations (abbreviated as configs) needed on a physical server based on the tenant's intent, which is the actual performance bottleneck of a virtual network controller.

To address the above challenges, we propose POSEIDON, a virtual network controller consisting of a unified virtual network control abstraction over heterogeneous devices and services. POSEIDON introduces innovations in three aspects.

1) At the architecture level, we find that among several modules in a controller, only the module that parses user intents changes frequently due to the addition of new APIs, devices and dependencies. Other parts do not change much. Therefore, we consolidate and unify the more stable modules of different controllers into one, which greatly reduces the development and operational costs of the controller when introducing new services and new devices.

2) At the abstraction level, in order to unify the management of heterogeneous devices and diverse services, we abstract them into generic objects. Then, to eliminate the table lookup complexity due to dependencies between configurations, we propose *Trident*, a novel abstraction for describing configuration dependencies. We find that the dependency between configs is transitive, so we can associate multiple services to generate a config tree. Based on this tree, we can easily find the configs that a particular config depends on, as well as the devices they configure, by traversing the nodes, thus eliminating traditional complex database table lookups.

3) At the implementation level, to address the issue of large config tree traversal time in production, we deploy a traversal cache to bypass the time-consuming reverse tree traversal for top tenants and design a hierarchical storage structure for achieving both high I/O performance and strong data stability.

We deploy POSEIDON on Alibaba Cloud and keep it running for more than 3 years. We obtain the following results.

- With POSEIDON, virtual network controller's latency is significantly reduced and throughput is increased. When enabling hundreds of EIPs (Elastic IP address) [8] in a same VPC, POSEIDON's latency is 1.8x~55x and 2.6x~4.8x lower than cloud vendor A (Top 5 [6]) and cloud vendor B (Top 5). Compared with the previous controller, its throughput has increased from 160 TPS to more than 3400 TPS (21x).
- The costs of developing and maintaining lots of controllers has been significantly mitigated with the consolidated controller in POSEIDON. The controllers' lines of codes has been significantly reduced by 22%~41%. Beneath the consolidated controller, the human efforts of developing a controller for a new service is reduced by 50%.
- The consolidated controller has taken over lots of the workload from each service controller, reducing the CPU and memory consumption by 50%.

2 Background and Challenges

In this section, we present the virtual network configuration workflow and discuss the challenges encountered during years of production deployment of our virtual network controller.

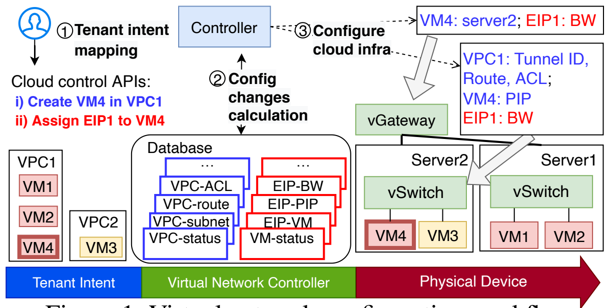


Figure 1: Virtual network configuration workflow.

2.1 Background

As Fig. 1 shows, the workflow consists of three steps:

① **Tenant intents mapping to cloud control APIs.** When tenants attempt to configure their virtual networks (*e.g.*, VM create/deletion), they use the cloud control APIs [1–3, 7] provided by cloud vendors to describe their intents. The cloud control APIs are standard interfaces for tenants to create, read, update and delete their virtual resources (*e.g.*, VMs). Different vendors offer different suites of cloud control APIs [1–3, 7]. Taking VPC configuration as an example, in Fig. 1, a tenant intent of "create VM4 in VPC1 with a public address EIP1" can be implemented with two cloud control APIs as "create VM4 in VPC1" and "assign EIP1 to VM4", which are fed to the virtual network controller for further processing.

②**Device config changes calculation.** After receiving tenants' API calls, the controller needs to compute the necessary updates to be configured on the physical devices. This is done with the following steps.

- Step 1: Identify all the dependent configs. Let us use the VM4 creation in VPC1 as an example (see Fig. 1). Because VM4 belongs to a specific VPC, to create VM4, we need to identify the VPC VM4 belongs to, which is VPC1. In addition, we also need to identify the dependencies of VPC1, such as ACL rules and routing tables. This step is done by doing SQL-based database queries. In this process, many tables, such as VPC-ACL, VPC-VM, *etc*, are queried.
- Step 2: Identifying the physical devices onto which the configs must be installed, as any network config is implemented on the physical device. As shown in Fig. 1, Server2 and vGateway need to be configured with the dependent configs.
- Step 3: Config changes calculation. After executing step 1, we know the configs needed for a API call. With step 2, we know the targeted devices and their existing configs. If the configs in step 1 already exist on the device, no config change is applied on the physical server. In contrast, the difference between the two is pushed to the physical server.

In our initial controller implementation, for each cloud control API, we write SQL code to manipulate databases and use if-else control block to arrange the SQL query order based on our service logic and the value acquired by table lookups. As device number, table size and if-else logic grow, the config changes calculation time is increasing rapidly.

③**Physical device configuration.** In the last step, the calculated config changes are pushed to the physical device. For

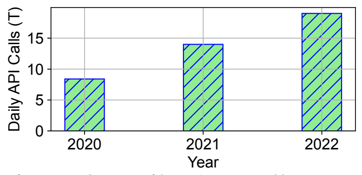


Figure 2: Daily API calls grow year by year in our cloud.

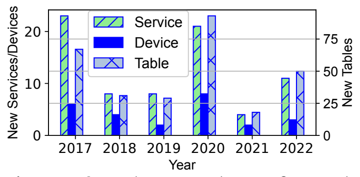


Figure 3: The number of yearly added services, devices and tables.

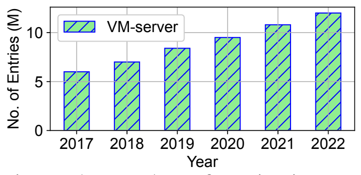


Figure 4: Number of entries in VM-server table of a region.

example, in Fig. 1, the intent of "create VM4 in VPC1" is eventually translated into the following device configs: the private IP (PIP) and VPC ID are configured on vSwitch for VM communication and VXLAN tunnel encapsulation, the routes and ACLs of the VPC are also added to vSwitch to regulate VM traffic, the VM-server mapping is installed to vGateway to route traffic from remote VMs to the VMs on the local host. Apart from that, in production, we need to deal with devices with different hardware forms, *e.g.*, x86- or Tofino-based vGateway [25]. Therefore, the controller needs to ensure that the translated configs in installed on these heterogeneous devices both timely and correctly.

2.2 Challenges

In this section, we describe two challenges of building a virtual network controller that manages millions of tenants.

2.2.1 Insufficient Performance for Growing Workloads

The scale of networks and the number of services grow dramatically over the past decades. We find that previous controller has degrading performance when handling the growing workloads due to the following three reasons.

Rapid growth of northbound API calls and southbound devices. Over the years, the usage of cloud resources by tenants has significantly increased. For example, in the past two years, the average number of PIPs within a VPC has increased from hundreds of thousands to millions. This causes a rapid growth in the number of VMs in VPCs. When a cloud user creates, deletes or updates of a VM, the user has to call the cloud controller API to take the corresponding actions. As the number of VMs increases, the number of cloud control API calls increases dramatically as well. Meanwhile, the growing number of tenants further exacerbates the frequencies of API calls that the controller needs to handle. As shown in Fig. 2, the API calling frequencies have doubled in two years, reaching tens of trillions of calls within a single day.

Furthermore, as the scale of an individual VPC increases, the physical devices the VPC covers also expand. This leads to a sharp rise in the number of southbound devices that a single API call needs to configure, imposing additional performance overhead on the controller. For instance, in extreme cases, a change of a single ACL rule in a large VPC may require a batch configuration to more than 100,000 servers.

Long search chains and large table size. With expansion of our cloud scale, the number of services and their dependencies increase, leading to a growing number of tables that need to be queried sequentially to finish the process of a single API call (sometimes dozens of tables are queried). Fig. 3 shows the recent growth of tables in our cloud. When a new table is added to record the relationship between services, the number of tables involved in the SQL query logic of some APIs may increase. Since the cloud controller needs to handle trillions of API calls each day, the additional table queries in a single API causes a significant burden on our database. Simultaneously, with the increase in the number and scale of virtual networks, the number of entries in major tables also grows, resulting in a longer database search/update latency. Using the VM-server table as an example, Fig. 4 shows its recent growth.

Due to the significant growth in table size and the number of tables being queried, over the span of one year, the P99 latency of calculating config changes for one API in our cloud (configure the vGateway) has increased from 860ms to 1615ms. The growth in latency has far exceeded the performance improvements of our controller that employs nested table searches, resulting in a lack of scalability in performance.

We follow a distinct cyclical pattern to iterate our systems in Alibaba, typically spanning several years. In the beginning of a span, a significant number of new services/devices/tables will be added to our cloud (*e.g.*, 2017 and 2020 in Fig. 3). After that, a considerable amount of time is required for debugging and large-scale deployments. In this period, new devices/services/tables (*e.g.*, 2018, 2019, 2021 and 2022 in the figure) are added only for reliability or security reasons, hence the number will be very limited. After that, we will kick-off a new cycle with a batch of new services/devices/tables.

Cloud-native applications intensify the controller performance requirements. The emerging cloud-native applications [4] require that the controller allocates and releases resources in a short time to adapt to the elastic changes in workloads. Such program-triggered API calling approach results in a significantly higher frequency than tenants manually configuring the network, posing challenges to the controller's performance on throughput and latency. For example, in e-commerce business, the time of peak traffic is known, and to save costs, resources will be massively scaled up (*e.g.*, tens of thousands of containers) just before the peak arrives, which needs a high-throughput controller. Similarly, cloud-native based social media applications need to handle surges in user access during hot events. To cope with the sudden increase in traffic, thousands of backends need to be elastically scaled in a short interval (*e.g.*, 500ms). This poses challenges to both the controller's throughput and latency. Insufficient controller performance will lead to backlog of tasks and delayed configurations. In the past, this will only impact the experience of network operators as they need to wait a bit longer before configurations take effect. However, for cloud-native applications, when the controller cannot support rapid network configurations for a large number of new instances, the unexpected surge in user traffic cannot be handled, resulting in service interruptions and user experience degradation.

2.2.2 High Cost of Managing Separate Controllers

Service-dedicated controllers for rapid iteration. In the beginning, we had only a small number of services, and different service teams developed and maintained service-dedicated controllers independently to ensure rapid iteration and isolated deployment. If we develop a monolithic controller that covers all services, huge continuous code integration efforts will be needed for such a complicated system, hindering iteration pace. Google's Andromeda [14] also holds a similar viewpoint. They believe that different services and their associated devices need to be managed by separate controllers [14].

However, as our services expand, the number of controllers has increased to dozens (*e.g.*, LB controllers, VPC controllers, cross-region controllers), and the development and maintenance costs of these controllers gradually become unsustainable. Fig. 3 shows the annual additions of services, devices, and tables between services in our cloud. When a new service is added, we need to not only develop new APIs, but also deal with possible dependencies with existing services. When a new device is added, it requires extensive development of device-specific tasks in the controller, such as life cycle management, consistency checks, device-dependent rule translations. Furthermore, during rapid iteration, each controller requires dedicated personnel from the service team for maintenance. The maintenance costs also increase linearly with the number of controllers.

Code redundancy and logic dependencies between controllers. In the service-dedicated controller architecture, we discover that the code redundancy for the southbound interfaces of different controllers is relatively high, as these interfaces are service-agnostic and primarily responsible for device management and configuration. However, each service team independently optimizes the performance of their interfaces, leading to significant waste of human resources.

In addition, as the business grows, the dependencies between services have become increasingly complex. A cloud control API processing may require multiple controllers to collaborate. For example, the Internet gateway is managed by both the VPC controller and the cross-region controller, so any modifications to the VPC controller must also consider the potential impact on the cross-region controller. The code redundancy and logic dependencies diminish the benefits of managing separate controllers.

3 Design

3.1 Design Overview

To address these challenges, we propose POSEIDON, a virtual network controller that can manage a large-scale cloud with millions of tenants. It contains the following 5 key designs:

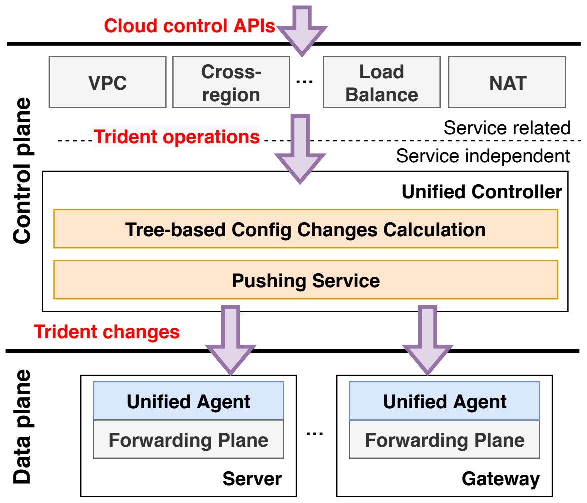


Figure 5: Layered controller architecture of POSEIDON.

Partial consolidation architecture. To reduce the costs of developing and maintaining multiple service-dedicated controllers, POSEIDON introduces a layered design that incorporates the consolidation of common logic among various services into a unified controller (§3.2), as shown in Fig. 5.

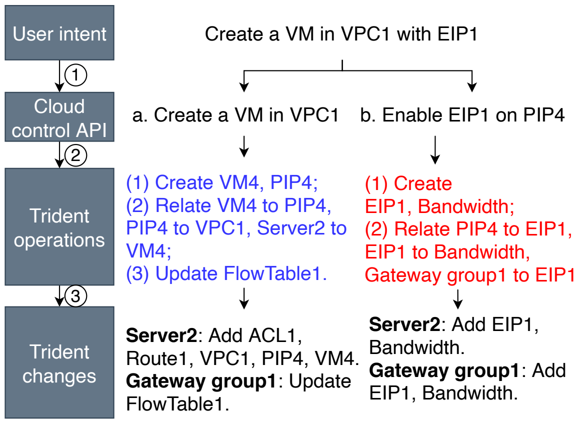
Trident: Service-independent abstraction. To unify the management of heterogeneous devices and diverse services, we design *Trident*, a service- and device-independent cloud control abstraction for the unified controller. Trident exposes 5 operations over 3 objects, allowing for flexible programming of existing cloud control APIs. Specifically, in order to achieve equivalence with SQL-based config changes calculation, Trident offers Relate/Unrelate, a pair of unique operations to represent the relations between objects (§3.3).

Config representaion through Trident tree. By continuously recording Trident operations, we can build a tree-like data structure named as Trident tree. Trident tree represents configurations, devices and relations between them (§3.4).

Tree-based config changes calculation. With Trident tree, the service-related table lookup logic (SQL+if-else) is replaced by equivalent tree traversal which decoupled config changes calculation from specific service logic. To accelerate the config calculation process, we take advantage of the observation that the descendant relation is transitive in the Trident tree and propose function-based descendant relation calculation with $O(1)$ computational complexity (§3.5).

Cloud-scale performance optimizations. For production deployment, we design a hierarchical storage structure for POSEIDON and deploy a traversal cache to bypass the time-consuming reverse Trident tree traversal for top tenants who have substantial number of virtual network elements (§3.6).

Fig. 6 shows the workflow of device configuration of POSEIDON. In POSEIDON, after tenant intents are mapped to cloud control APIs, instead of writing (SQL + if-else) query logic, the service-related controllers maintained by individual service teams translate the APIs into Trident operations.


User intent
Create a VM in VPC1 with EIP1
Cloud control API
a. Create a VM in VPC1
b. Enable EIP1 on PIP4
Trident operations
(1) Create VM4, PIP4; (2) Relate VM4 to PIP4, PIP4 to VPC1, Server2 to VM4; (3) Update FlowTable1.
(1) Create EIP1, Bandwidth; (2) Relate PIP4 to EIP1, EIP1 to Bandwidth, Gateway group1 to EIP1
Trident changes
Server2: Add ACL1, Route1, VPC1, PIP4, VM4. Gateway group1: Update FlowTable1.
Server2: Add EIP1, Bandwidth. Gateway group1: Add EIP1, Bandwidth.
1
2
3
Figure 6: Device configuration workflow of POSEIDON.

Then, the unified controller changes the config tree topology according to the Trident operations and conducts tree-based config changes calculation to generate the device-independent Trident changes, which will finally be configured into devices.

3.2 Consolidation of Separate Controllers

In this section, we discuss the problems of consolidating all the functions of multiple controllers into a big one. Then, we introduce the partial consolidation adopted by POSEIDON.

3.2.1 Problems with full consolidation of controllers

To reduce the overhead of multiple controllers, it is natural to consider consolidating their functions into one big controller. However, full consolidation incurs significant costs. In our clouds, each service-dedicated controller is rapidly iterated to satisfy changing service requirements. Before each iteration goes online, it is necessary to conduct comprehensive testing of the controller on all relevant APIs and their parameters. Given there are x controllers (in our cloud x is 50~60), and each is iterated $Iter_i$ times a month and has $Test_i$ test cases to cover before put online, the total test cost is $O(\Sigma_{i=1}^{x} Iter_i * Test_i)$. Comparatively, after full controller consolidation, the iteration frequency will be the sum of the iteration frequencies of each controller. Moreover, on each iteration, all the test cases need to be examined and the total test cost grows to $O(\Sigma_{i=1}^{x} Iter_i * \Sigma_{i=1}^{x} Test_i)$, which is likely to impede the rapid iteration of cloud services.

3.2.2 POSEIDON's choice: partial consolidation

Despite the significant cost of full consolidation, we have observed that some parts of the controller are service-independent, which can be consolidated without high cost. For example, at the northbound of the controller, the API parsing logic is service-related and constantly undergoes rapid iterations. In the middle, although config changes calculation for different services may involve orchestration of different table query sequences, the underlying table query mechanism is quite similar. At the southbound, the controller needs to interact with devices. The code for interacting with different devices often shares a significant amount of common logic such as life cycle management which transitions between device states. Additionally, the device update frequencies are usually much lower compared to the service iteration frequencies (Fig. 3), making southbound logic more stable.

Based on the above observations, to maintain the flexible iterative capability of cloud service at the northbound and reduce the OpEx of maintaining multiple controllers, we carry out partial consolidation of common parts of each controller. Specifically, we build a service-independent abstraction layer in the middle, offering a set of atomic operations, enabling flexible orchestration of diverse northbound service requirements using these atomic operations. Past SQL + if-else approach poses a high demand for developers' understanding of mechanisms of cloud networks. With the new abstraction, developers only need to parse APIs into these atomic operations that are independent of both services and devices. Beneath this abstraction layer, we consolidate the implementations of different controllers for config changes calculation and device configuration to reduce redundant development and maintenance costs, as shown in Fig. 5. As mentioned in §2.1, config changes calculation is the most time-consuming step. Consolidation helps us concentrate on its optimization collectively, rather than conducted by individual teams.

Furthermore, for installing configs on heterogeneous device, we deployed a unified agent on each device for receiving Trident changes, as shown in Fig. 5. There is a device-specific program for translating unified Trident changes into device-specific primitives on the agent. The program is developed by the device management team, who is more familiar with the command-line interface of the device.

3.3 Service-independent Abstraction

We first introduce three goals of designing abstractions for managing virtual network, then elaborate on how POSEIDON abstracts the API parsing logic and various services/devices.

3.3.1 Goals of designing abstraction

- **Service-independent:** The processing logic is agnostic to service, namely, all APIs share a unified codebase.
- **Device-independent:** The objects should hide device heterogeneity to ease the programming with the abstraction.
- **Equivalence:** The results obtained with the abstraction should be the same as past SQL + if-else based approach.

3.3.2 POSEIDON's abstraction

Based on the design rationale, we propose *Trident*, the cloud control abstraction for the unified controller in POSEIDON. Trident exposes 3 objects (Conf, Device, Group) and 5 operations (Create, Update, Delete, Relate, Unrelate) upward, facilitating the interpretation of existing cloud control APIs.
Conf: During virtual network configuration, the manipulation of tenant resources through cloud control APIs will finally translate into manipulation of configs (*e.g.*, routes and ACLs) on the physical device. We use *Conf* to denote the manipulable configs (*e.g.*, VM, ACL in Fig. 7) on the device.

Device: In order to hide the differences of underlying devices in the abstraction, we use *Device* to abstract different kinds of devices (*e.g.*, server, gateway in Fig. 7) that carry configs. **Group:** To manage increasing workloads, horizontally scaling devices into clusters is common in the cloud [15, 25, 26]. Within the same cluster, all device share same configs. To avoid the repetitive config changes calculation for devices within the same cluster, we introduce *Group* as the representative of the devices to carry Confs for the cluster. For consistency, we also create a Group with the same name for an isolated device that does not form a cluster (*e.g.*, Server1). **Create/Update/Delete:** The CRUD (Create, Read, Update, Delete) of tenant resources through cloud control APIs internally turns into CRUD of related Confs/Devices/Groups by the unified controller. Since this paper focuses on virtual network configuration due to Trident tree changes, we do not include the *read* operation in this paper, which will only involve retrieving values from the tree. The Create/Delete Devices takes place when devices are allocated/deallocated for the scale-out or scale-in of cloud workloads.
Relate/Unrelate: There may be relations between Confs, Devices and Groups. Specifically, for relations between two Confs, we use Conf1→Conf2 to denote that the successful configuration of Conf1 depends on Conf2. For example, VPC→ACL meaning that each time configuring a VPC, we need to configure its ACL simultaneously. For relations between Devices and Groups, we use Device1→Group1 to denote that Device1 belongs to Group1 as Device1 has the same configurations with other devices (if any) in Group1. For relations between Groups and Confs, we use Group1→Conf1 to denote that Conf1 is configured to all the devices attached to Group1. As Group1 represents a cluster of devices, we no longer associate Conf1 with each device attached to Group1 to reduce redundant configurations. In other words, the relation between Conf and Device will not exist. In Trident, we introduce *Relate* to add → between two objects if they have a relation and *Unrelate* to remove → if the relation disappears.

3.4 POSEIDON's Trident tree

With Trident, the processing of cloud control API is abstracted into creating/deleting objects and relating/unrelating objects to other objects. After that, multiple "Device→Group→Conf" chains are formed. To enhance visualization and comprehension, we represent these interconnected chains as a tree structure, referred as "Trident tree", *e.g.*, the tree in Fig. 7.

3.4.1 Building Trident tree

Fig. 6 shows the equivalent Trident operations for the two example APIs. These operations create, delete various objects, and modify the relations between them. For example, the Trident operations of API *a* in Fig. 6 will create two Confs, VM4 and PIP4, as shown in Fig. 7. Then, it will build relations between VM4, PIP4 and the dependent Confs. Additionally, VM4 needs to be related with the corresponding Group, indicating its creation on the Devices represented by that Group (*i.e.*, Server2). These two API calls form a path (marked with

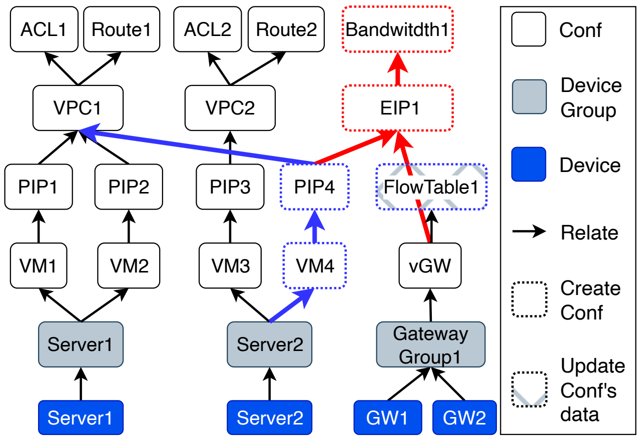


Figure 7: Trident abstraction and an example of Trident tree.

the blue and red path in Fig. 7). With more API calls, finally, a Trident tree that records the dependencies between objects is created as shown in Fig. 7. Since the configs within the Group are identical, we consider the Group as the root of the Trident tree in the following config changes calculation procedure.

3.4.2 Handling API calls with Trident tree

Any API call can be represented as changes to the Trident tree, and based on the Confs changes of the Group, we can determine what updates should be made to the Devices within it. We use the API "create VM4 in VPC1" to illustrate this statement. To execute this API call, we have two steps:

1) We identify the Confs that VM4 depends on. This is achieved by traversing from VM4 to its descendants, which are PIP4, VPC1, ACL1, and Route1. Then, by traversing from these dependent Confs to their roots with a reverse tree traversal, we can locate the target Groups that carry these Confs (*i.e.*, Server1 and Server2).

2) We deduce the actual changes that need to be pushed to the devices based on the relations between dependent Confs and Groups. We can obtain the set of Confs carried by target Groups (*e.g.*, Server2) by traversing their descendants before and after changing the tree topology with Trident operations. By performing a diff operation on the two Conf sets, we can obtain config changes for target Groups. In Fig. 7, the added descendants of Server2 include VM4, PIP4, VPC1, ACL1, and Route1. Hence, the config changes of Server2 are "add VM4, PIP4, VPC1, ACL1, Route1". These changes are represented with Trident objects and device-independent.

3.5 Tree-based Config Changes Calculation

3.5.1 Basic procedure

Algo. 1 describes the solution adopted by POSEIDON. API calls change the tree topology by either adding/deleting tree nodes or modifying the relations between them (line 3). After the tree is updated, we can find all the dependent Confs of these Trident operations by traversing the tree (line 4). Then, by reversely traversing from these dependent Confs to their roots, we can obtain the target Devices. As Devices are represented by their Groups, finding Groups is enough for subsequent calculation (line 5). For each Group, the Confs that need to be added/deleted/updated are obtained by detecting the changes of its descendants (line 6-13). For example, when a Group's descendants expand due to a new Conf being added, we need to add a corresponding configuration rule to the Devices represented by this Group (line 8-9).

Algorithm 1: Tree-based config changes calculation.

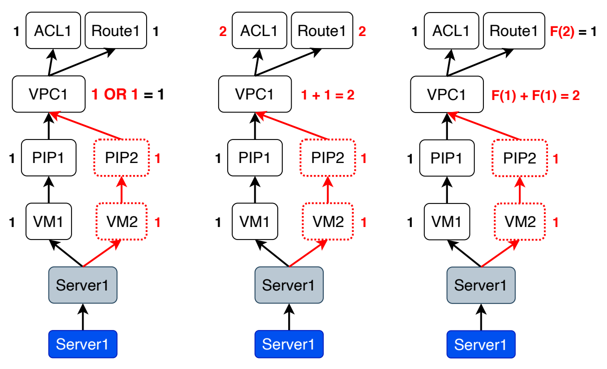
```
1  Function ChangesCalculate(Cloud control API):
2      Mapping API to Trident operations.
3      Change the Trident objects and their relations according to
         Trident operations.
4      $conf\_set$ = all the dependent Confs of Trident operations.
5      $group\_set$ = FindRoots($conf\_set$).
6      for group in $group\_set$ do
7          for c in $conf\_set$ do
8              if c will be added into descendants of group then
9                  Add c to group.
10             else if c will be removed from descendants of group
                 then
11                 Delete c from group.
12             else if c changes & c is in descendants of group then
13                 Update c in group.
```

3.5.2 Fast finding descendant changes for a Group

A critical step in this procedure is to verify whether a Conf resides in a Group's descendants (line 8, 10, 12).

Naive tree traversal is unscalable. The naive approach employs a widely used traversal algorithm, *e.g.*, depth-first search, which has a time complexity of $O(nodes)$, starting from the Group and checking each visited tree node. However, this solution is very time-consuming for large-scale cloud networks, as most devices accommodate a significant number of configs (*i.e.*, Confs in Trident tree), and the number of configs keeps growing as the device performance improves. For example, our gateway accommodates millions of VXLAN routing table entries [25]. It is inefficient if each API call requires a complete tree traversal, which will consumes hundreds of seconds for traversing the Trident tree with millions of nodes.

Descendant relation is transitive. We adopt descendant relation between Conf and Group to indicate that a Conf is within the descendants of a Group, that is, the Conf is configured to the Devices represented by the Group. We observe that in Trident tree, the descendant relation is transitive and can be propagated from parent nodes to their children. Specifically, if a Conf is configured on a Group, then all of its children must also be configured on the same Group. Because parent nodes need to be configured together with the dependent Confs (*i.e.*, their children) on the Group. Therefore, for a Conf, determining whether it has a descendant relation with a Group only requires examining its parents. If any parent of the Conf has a descendant relation with a Group, the Conf inherits the descendant relation with that Group. We employ *reference count* to record the descendant relation between Conf and Group, denoted as $ref(Group, Conf)$. For example, we can use $ref = 1$ to indicate there is a descendant relation while



(a) OR-based (b) SUM-based (c) Function-based

Figure 8: $ref(Server1, Conf_X)$ calculation. The changes and unchanges are marked by red and black, respectively.

$ref = 0$ to indicate no relation.

There are multiple potential solutions to execute an API call on the Trident tree with transitive descendant relation. Here, we illustrate the pros and cons of two potential solutions.

Solution1: OR-based ref calculation. To figure out whether a Conf is a descendant of a Group, we can check if either of its parents is a descendant of the Group. This indicates that we can apply the *OR* operation on the reference count of all parents to obtain the reference count of the Conf itself. Thus, we have $ref(Group, Conf_A) = \vee_i ref(Group, P_i)$, where P_i is the i^{th} parent of $Conf_A$. As shown in Fig. 8, $ref(Server1, VPC1) = ref(Server1, PIP1)$ OR $ref(Server1, PIP2) = 1$. The computational complexity is $O(P_i)$ (for a detailed analysis, please refer to §B.3).

Based on our experience, a Conf's parents can reach several million in our cloud. For example, during shopping festivals, a VPC can have millions of VMs. Therefore, the OR calculation due to a single API call (like VM create or delete) could take more than tens of seconds. In order to ensure accuracy of ref calculation, when a Conf's parent undergoes a change, the calculation triggered by changes of other parents will be blocked until the refresh of the ref for this round (*e.g.*, tens of seconds) is completed. This approach significantly limits the system's throughput and increases its latency, making it unsuitable for production deployment.

Solution2: SUM-based ref calculation. Another solution is asking a Conf to inherit all parents' ref by using a summation method, as defined by $ref(Group, Conf_A) = \Sigma_i ref(Group, P_i)$. Whenever the $ref(Group, Conf)$ becomes zero, we need to delete the Conf from the Group. Because the summation method adheres to the *reversibility property*, the computational complexity of this SUM-based calculation is $O(1)$ (for a detailed analysis, please refer to §B.4).

However, this solution also has a limitation, as any change in a node's ref propagates through all its descendants layer by layer. For example, creating VM2 of VPC1 on Server1 will cause $ref(Server1, VPC1)$ to increase from 1 to 2, as shown in Fig. 8. As ACL1 and Route1 are children of VPC1, this will result in synchronized changes for $ref(Server1, ACL1)$

Table 1: Changes of $ref(Group, Conf)$ when Trident tree changes. The blue and red text denote the changes due to the blue and red operations in Fig. 7, respectively. "$0 \rightarrow 1$" represents the value changes from 0 to 1.

Group \ Conf	VM1, PIP1 VM2, PIP2	VM3, PIP3, VPC2 ACL2, Route2	VM4 PIP4	VPC1	ACL1 Route1	EIP1 Bandwidth1	FlowTable1
$Server1$	1	0	0	2	1	0	0
$Server2$	0	1	$0 \rightarrow 1$	$0 \rightarrow 1$	$0 \rightarrow 1$	$0 \rightarrow 1$	0
$Gateway\ Group1$	0	0	0	0	0	$0 \rightarrow 1$	1

and $ref(Server1, Route1)$ as well. When the ref of a Conf changes only within the positive range, it implies that a descendant relation always exists and there is no need to trigger a configuration to the corresponding Group. For cloud networks, service dependencies are intricate and the tree depth can reach dozens of layers. While this propagation may trigger ref changes in many layers, it won't incur configuration updates, leading to a waste of computation resources.

Our solution: Function-based ref calculation. As discussed, the OR-based solution has high computational complexity, while the SUM-based solution has ineffective propagation. With lessons of both, we propose function-based ref calculation. Specifically, to reduce computational complexity, we retain the reversibility property advantage of the SUM-based solution, refreshing ref with $O(1)$ complexity. In addition, to address the ineffective propagation, we filter out irrelevant changes from a Conf's parents by counting the number of its parents with a non-zero ref. The ref calculation is defined by $ref(Group, Conf_A) = \Sigma_i F(ref(Group, P_i))$, where $F(x)$ is a piecewise function as $F(x) = 1, x > 0; F(x) = 0, x = 0$. With this function, any changes of x within the positive range will not alter its function value (remain 1), thereby preventing ineffective ref calculation propagation (hide children from irrelevant ref changes of parents).

Assuming we continuously create VMs of VPC1 on Server1, $ref(Server1, VPC1)$ will keep increasing. With our solution, $ref(Server1, ACL1)$ and $ref(Server1, Route1)$ will always remain 1, as shown in Fig. 8. This demonstrates that children won't inherit irrelevant ref changes from their parent. However, if there are no more VMs of VPC1 on Server1, $ref(Server1, VPC1)$ will be 0. With our solution, $ref(Server1, ACL1)$ and $ref(Server1, Route1)$ will become 0 in response because $F(0) = 0$. To sum up, our solution addresses the performance and overhead issues of previous two solutions, making it suitable for large-scale cloud networks.

3.5.3 Fast finding updates for a Conf.

When we need to update a Conf, we perform a reverse traversal on the tree from the Conf to locate the Groups that accommodate it. Then, we push the modification of the Conf to the Devices under these Groups. In production environment, there are always concurrent updates to the same Conf. To ensure the correctness of the order in which updates are pushed to devices, we introduce a *version* field to Conf. On each update, the version is incremented by 1. When multiple updates of a Conf are pushed to a device, the on-device unified agent maintains the update install order based on the version numbers and performs reordering for out-of-order updates.

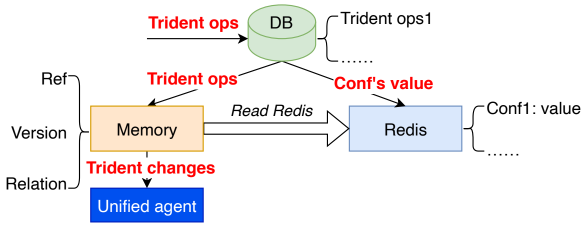


Figure 9: Hierarchical storage structure of POSEIDON.

3.5.4 A tree-based config changes calculation example

With the ability of ref and $version$ to fast detect the changes of Group's descendants and updates of Conf, we can modify lines 8, 10, and 12 in Algo. 1. For example, we can replace the logic of line 8 by checking if ref transitions from 0 to a positive value. By observing changes in $version$ and ensuring ref is not 0, we can determine whether an update needs to be pushed to a certain Group, which can replace the logic of line 12. With ref and $version$, unnecessary configurations can be efficiently filtered out. Finally, we read the value of the Confs that require deployment to generate the final Trident changes and push them to the unified agent. Table 1 illustrates the changes in ref resulting from Trident operations parsed from API a and API b in Fig. 6. Using these changes, we can efficiently compute config changes. For example, when $ref(Server2, ACL1)$ transitions from 0 to 1, it indicates that ACL1 needs to be configured on Server2. Similarly, an update to FlowTable1 will increase its $version$ by 1, requiring this update to be pushed to the Gateway Group1, which has a descendant relation (with a non-zero ref) with FlowTable1.

3.6 Cloud-Scale Performance Optimizations

Hierarchical storage structure of POSEIDON. In order to achieve higher I/O performance while ensuring data reliability, we propose a hierarchical storage structure for POSEIDON, consisting of memory, Redis and database, as shown in Fig. 9. Among these, memory provides the highest performance but is vulnerable to data loss in case of power failure. Redis offers intermediate performance and a certain level of reliability, as it periodically batches data onto hard disk. The database is stored onto disk, offering the lowest performance but ensuring data persistence even during power loss.

POSEIDON needs to store various data objects, including Trident operations, $refs$, $versions$, Conf relation and Conf's value. Based on the data read frequency and reliability requirements, they are stored in three different storage mediums. Among them, Trident operations are derived from the interpretation of APIs. All other data structures are calculated based on them. Hence, Trident operations serve as the source of

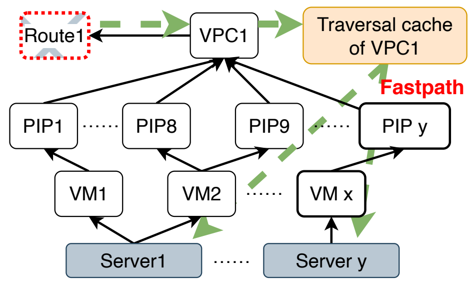


Figure 10: Traversal cache.

data for config changes calculation but will not be frequently accessed during the calculation. Therefore, they are suitable for storage in database as ground truth data. In the event of memory power loss, they can be used to restore other data. The ref, *version*, and relation involve high-frequency read and write requirements, thus they reside in memory. The entire computation process takes place in memory for ultimate performance. In addition, we store the Conf's value in Redis, as the Conf's value has a lower read frequency compared to ref/*version*/relation but higher than Trident operations. This is because the controller only needs to read the Conf's value in the final step of config changes calculation. To minimize the storage pressure on memory, we store it in Redis. In addition, in order to prevent data lost of Redis due to failover, we design a failover detection method in §B.6.

Traversal cache. Based on POSEIDON's deployment experiences, we have observed performance degradation when performing config changes calculation for some top tenants (*e.g.*, those with millions of VMs). In a real case, the reverse tree traversal time reaches a surprising 150s when a visited Conf has 10^5 parent nodes. To tackle this issue, we introduce *traversal cache*. The basic idea is to maintain an additional cache for nodes with an excessive number of parent nodes, which directly records the Groups obtained from the reverse traversal. As shown in Fig. 10, as the parents of VPC1 continue to increase, when reaching a certain threshold, we will attach a traversal cache to VPC1, which records the Groups holding descendant relation with VPC1. When Route1 changes, VPC1 will be visited during the reverse traversal to find the Groups. When VPC1 is visited, the Groups recorded in its traversal cache will be directly retrieved, eliminating the need for time-consuming traversal of the entire tree. This is shown in Fig. 10, where the green line with traversal cache bypasses the entire tree traversal. When Trident tree is updated by Trident operations, the traversal cache will be refreshed according to the function-based ref calculation.

4 Evaluation

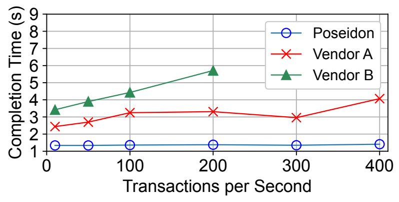
POSEIDON has been deployed in our production cloud for over three years and is responsible for handling the majority of API processing. The following data were collected from a production region containing hundreds of thousands of vSwitches, thousands of GWs, and tens of millions of VMs.

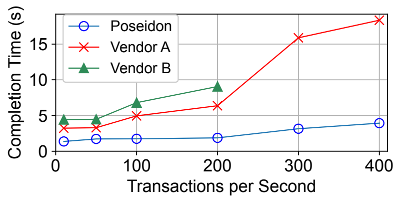
4.1 Performance improvement

End-to-end completion time under different TPS. *Experiment setting.* For a controller, the key performance metric is its concurrent processing capacity and the corresponding completion time. Since the concurrent capacity of unrelated API calls can be enhanced through horizontal scaling, we focus on the capacity for processing related API calls. Specifically, we conducted experiments to measure the concurrent capacity of Vendor A (Top 5 [6]), Vendor B (Top 5), and ours (Top 5) on two of the most widely used APIs in our public cloud services, that is, *Enable and Disable Elastic IP (EIP) [5] for a VM* in a same VPC. We have selected these two APIs for two specific reasons. Firstly, they enable us to precisely gauge the time it takes for the controller to process an API call, excluding any time spent on unrelated modules like the VM's startup time, which occurs when utilizing the VM creation API. Secondly, these two APIs represent a 10% of the 1200 APIs available to our customers. It should be noted that as POSEIDON utilizes the same logic and codes to handle all APIs, there is no specific optimization to these two APIs. We leverage the Cloud control API interfaces provided by the major cloud vendors to initiate API calls with different transactions per second (TPS). By recording the time taken for successful (for testing enabling EIP) or unsuccessful (for testing disabling EIP) pings to these EIPs, we can obtain the P50, P90, and P99 completion time. It is important to note that the timing starts from when the controllers of Vendor A/B and ours receive the API calls returning timestamp when the controller's acknowledgment of receiving the API calls), which effectively eliminates the impact of transmission latency on the measurements. All the experiments were conducted in the region of Jakarta[††] from August 20th to 29th, 2023.

Performance analysis. Fig. 11 depicts the P50, P90, and P99 completion time for enabling EIP. As shown in the figure, the completion time of Vendor A and Vendor B is 1.8x~55x and 2.6x~4.8x higher than that of POSEIDON. Moreover, our system demonstrates superior stability in high concurrent scenarios. Even with 400TPS, the P50 of POSEIDON remains consistently stable at 1.3s. The reason for testing Vendor B's performance only up to 200TPS is that when we attempted 300TPS, an unsupported error occurred. The specific error details can be found in the appendix (Fig. A1). Furthermore, it is worth noting that during the testing at 400 TPS, the majority of completion time for Vendor A are below 30s. However, there are 8 specific cases where the process of enabling EIP took over 300s. Thus, the P99 of Vendor A is much higher than that of ours (about 55x). This observation suggests that the Vendor A control plane encounters instability under high TPS scenarios. In the case of disabling EIP, the completion time for Vendor A and Vendor B are 1.6x~12x and 1.3x~2.5x compared to ours, respectively, which will be detailed in the appendix (Fig. A2) due to the space limits.

[††]We repeated the experiments across multiple regions and observed consistent results; for brevity, we illustrate only one set of results in this paper.


(a) P50


(b) P90

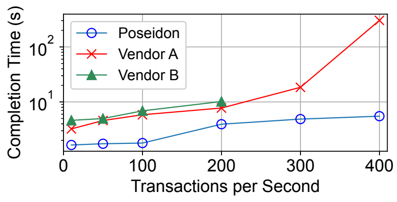

(c) P99

Figure 11: End-to-end completion time of API calling for enabling EIP.

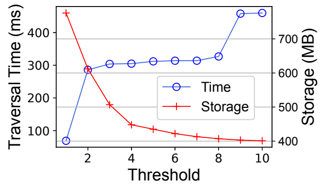


Figure 12: Threshold selection of traversal cache.

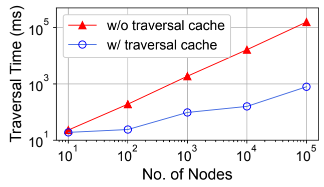


Figure 13: Traversal time optimization with traversal cache.

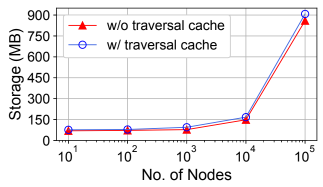


Figure 14: The storage usage (w/ vs. w/o traversal cache).

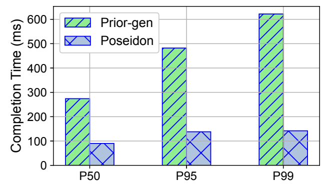


Figure 15: Completion time of config changes calculation.

Traversal cache. *Threshold selection.* The most important metric for the traversal cache is the threshold, which determines the number of parent nodes that trigger the attachment of a traversal cache for a Conf. We conducted experiments in a Trident tree containing hundreds of thousands of nodes. Fig. 12 shows the effects of the threshold on the P99 completion time of reverse traversal from all nodes to find their roots, as well as the storage consumed by the traversal cache. As the threshold increases from 2 to 8, the P99 completion time remains relatively stable, while the storage consumption decreases significantly. Once the threshold exceeds 8, there is a notable increase in the P99. Therefore, the threshold of 8 is a sweet trade-off point and is adopted in our production.

Traversal time optimization. Fig. 13 shows the P99 completion time for traversing (traversal time) different scales of Trident tree. As the number of nodes increases, the traversal time increases in both scenarios. When the Trident tree contains 100,000 nodes, the traversal time without traversal cache reaches 157s, which has a significant impact on the user experience due to a long configuring delay. When we attach a traversal cache for the node whose parents number exceeds the threshold, the traversal time is notably reduced. Even if the number of nodes reaches 100,000, the traversal time with traversal cache is less than 1s, that is, only 792ms.

Storage usage. Fig. 14 illustrates the storage usage with/without traversal cache. As shown in the figure, traversal cache only causes a small piece of extra storage costs.

Config changes calculation. *Bottleneck of prior-gen.* After conducting a thorough analysis of the performance bottleneck of the prior generation controllers, we have figured that the main bottleneck resides in the table lookup I/O of database for calculating config changes. Based on our historical records, for the prior-gen, the P99 completion time for config changes calculation reached 9s at 160TPS. In order to overcome the I/O bottleneck in config changes calculation, POSEIDON caches the Trident tree in the memory and Redis, which significantly improves the the completion time and

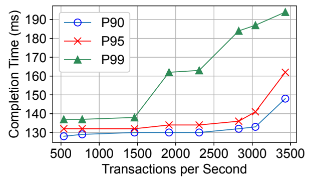


Figure 16: Completion time of config changes calculation.

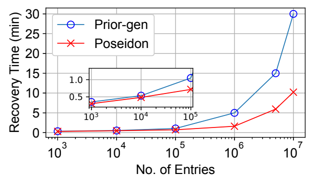


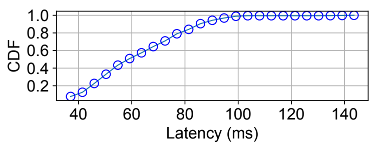
Figure 17: Recovery time after restarts.

concurrent capacity of calculating config changes.

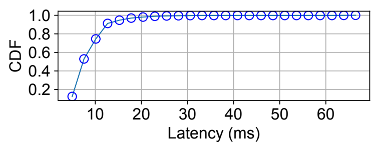
Completion time improvement. Fig. 15 depicts the config changes calculation completion time of the same API in the prior-gen and POSEIDON controller. The experiment is conducted by calling the API 100 times repeatedly. As illustrated in the figure, the completion time is notably improved compared to the prior-gen. The P50, P95 and P99 completion time of prior-gen are 3x, 3.5x and 4.4x larger than that of POSEIDON. In addition, compared to the prior-gen, a smaller difference between P99 and P50 of POSEIDON indicates a smaller variance and thus a more stable performance.

Concurrent capacity improvement. Fig. 16 shows the config changes calculation completion time of POSEIDON under different TPS. As illustrated in the figure, P99 experiences a drastic increases after 1500TPS. There is a sharp increase in P90 and P95 when the frequency of concurrent calls exceed 3000TPS. The P99 of POSEIDON is less than 200ms when the frequency of concurrent calls is 3432TPS. Compared to the prior-gen, where the performance sharply deteriorated to 9s after reaching 160TPS, the concurrent processing capability of the config changes calculation module in POSEIDON has been greatly enhanced.

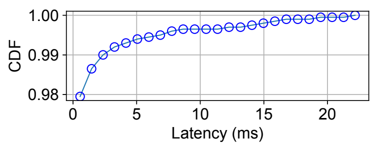
Completion time of key steps in POSEIDON. Fig. 18 depicts the completion time distribution of three key steps in the POSEIDON workflow. The concurrent calling frequency is about 1,000TPS. As shown in the figure, config changes calculation is the most time-consuming work due to its high complexity and a multitude of I/O between memory and Redis. The P99 of config changes calculation, pushing and enabling is about



(a) Config changes calculation



(b) Config changes pushing



(c) Config changes enabling

Figure 18: Completion time of three key steps in the POSEIDON workflow.

100ms, 20ms and 3ms, respectively. It also can be seen from the figure that the CDF curve of config changes calculation (Fig. 18(a)) is the shallowest one since the completion time is greatly affected by the number of Confs and their dependencies in the Trident tree (as shown in Fig. 13), which varies dramatically. Besides, the tail in Fig. 18(b) and Fig. 18(c) is much longer than that of Fig. 18(a). The reason for the long tail in Fig. 18(b) is the extreme case for a small piece of customers. When pushing the updates of the routing table for a large VPC, the changes need to be pushed to hundreds of thousands of devices, resulting in a few outliers with extremely long completion time. The long tail in Fig. 18(c) is caused by resource constraints on the device when other applications exhaust the available resources.

4.2 OpEx and development cost optimization

OpEx. Since the common logic of controllers has been taken over by the POSEIDON unified controller, the service-related controllers focus on translating the API into Trident operations, leading to a notable reduction in the number of Lines of Codes (LOC) of controllers. As shown in Table 2, the reduction rate of LOC are 22%~41%, which demonstrates a significant mitigation in OpEx.

Development cost. Additionally, as the controller's development no longer needs to care about lots of works (*e.g.*, config changes calculation and pushing, consistency checking, *etc*), the development cost has also gained a substantial reduction. By comparing the development cost of two LB controllers with similar business logic, we found that the cost has been reduced by half. When developing a controller without POSEIDON, the human effort is 6 person-months and 66K lines of code are written. With POSEIDON, the human effort amounts to 3 person-months and we write 30K lines of code.

It is important to note that the reduction in OpEx and development cost has not been added to the POSEIDON platform, as the LOC of the POSEIDON is only around 150K, which is much lower than the total LOC reduction.

Table 2: Reduction in Lines of Code of controllers.

	LOC (prior-gen)	LOC (POSEIDON)	Reduction
LB1	167*K*	98*K*	41.3%
LB2	76.9*K*	46.9*K*	36.4%
VPC	873*K*	559*K*	36%
NAT	107*K*	65*K*	39.3%
VPN	97*K*	70*K*	27.8%
Private Link	31.8*K*	22.8*K*	28.3%
Accelerator	135*K*	105*K*	22.2%

Table 3: Resource consumption of unified agent.

	Physical device		Virtual element		
	SNA	Server	VM	LB	GW
CPU-Avg	0.12%	0.21%	0.81%	0.47%	0.16%
CPU-Max	0.94%	1.04%	1.88%	1.41%	0.99%
Memory	0.16%	1.30%	0.47%	0.31%	0.73%

4.3 Benefit and cost of unified agent

Recovery time after restarts. Fig. 17 illustrates the recovery time for devices carrying different numbers of entries. The more entries, the more reduction. The persistent storage of config changes by the unified on-device agent enables fast data recovery. The substantial optimization of recovery time mitigates the downtime of devices, which improves the overall utilization of devices and facilitates device iterations.

Resource consumption. The POSEIDON agent is deployed on the device, consuming computing and storage resources. We evaluate resources consumption for different physical devices (*e.g.*, Smart Network Appliance (SNA) [25]) and virtual elements, as shown in Table 3. For the CPU, the average usage remains below 1%, and the maximum usage is below 2%. As for memory usage, it does not exceed 1% for all cases except the server. This indicates that the agent only needs to consume a small piece of on-device resources.

5 Experiences of deploying POSEIDON

How to migrate to POSEIDON? For a large-scale public cloud provider, how to migrate from its old controller to POSEIDON without affecting the quality of ongoing services is a very challenging task. Initially, we leveraged lots of test cases to verify the consistency between the POSEIDON and the old controller. Only upon passing all test cases would we allow the replacement of the old controller with POSEIDON in the production environment. However, infrequently used services not covered by the test cases led to continuous online configuration errors. The practice of debugging after errors has seriously compromised the availability of our services, resulting in a substantial number of complaints. We gradually recognized that test cases could not guarantee comprehensive coverage of all usage of numerous services. Consequently, we opted for a dual-running test in production, leveraging massive real user API calls to eliminate uncovered corner cases. Specifically, we apply a dual-running test where both the old controller and POSEIDON are running to handle the same API calls, but deliver configs to production devices and test devices, respectively. For each physical device in production, POSEIDON will fully take over the configuration management only after it has accurately handled the API calls of over 90% of the tenants using the device, over 95% of the services implemented by the device, and over 90% of the APIs related to the device for a specified period—this period being one week for smaller regions and one month for larger regions. Employing this strategy, the faults attributed to POSEIDON have been significantly mitigated throughout the year.

POSEIDON's performance in extreme situations: *Extensive route fluctuations.* For tenants with self-built Internet data center (IDC), their IDC's BGP routes must be learned by their VPCs in the cloud. Some tenants operate large-sized IDCs with thousands of BGP routes. When there are fluctuations in the links between IDCs and the cloud, it triggers massive BGP route convergence. In extreme cases, BGP route oscillation may also occur. All changes in BGP routes must be processed and configured by the controller. For prior-gen controller, it takes over five minutes to converge two thousand routes. With POSEIDON, the converge time is reduced to tens of seconds, significantly mitigating the impact on tenant services.

Large-scale device restart. Normally, network devices undergo scattered restarts due to upgrades or SW/HW issues. Extremely, there may be a widespread batch restart of devices, *e.g.*, during a power outage. Over the past years, we experienced three data center-level power outage incidents. The first two were handled by the prior-gen controllers. During restarts, a sudden surge in config retrieval requests from a vast number of devices directly saturated the database connections. Consequently, the recovery took hours. For the third incident with POSEIDON deployed, due to its trident tree algorithm and hierarchical storage, the recovery was finished in minutes.

Where to record the descendant relation between Conf and Group? How to record these relations in a Trident tree will affect the performance of configuration delivery, including the incremental configuration delivery (*i.e.*a tenant adding a new Conf) which ensures the timeliness of user configuration delivery, and the full configuration delivery (*i.e.*device restart) which is crucial to device upgrades and maintenance. Specifically, if each Conf records its targeted Device, it will be easy to calculate the incremental config changes, but the full configuration delivery will be very slow because all Confs must be traversed. In contrast, if each Group records the Conf it needs, the full configuration delivery will be fast because all required configs are already recorded in the Group, while the incremental configuration delivery will be slightly slower because the Trident tree needs to be reversely traversed to find the roots. Note that we cannot record the relation in both Conf and Group as this implementation contains too many locks, leading to terrible performance. Considering POSEIDON servers a large-scale public cloud, where device updates are extremely frequent, and stability maintenance is a critical task, we choose Group to record the descendant relation to achieve the best performance of full configuration delivery.

6 Related work

The previous abstraction works [9, 12, 16, 22–24, 27] provide a unified abstraction to shield the heterogeneity of devices, making the network configuration easier for network operators. For example, based on NETCONF [16], YANG [9] has been proposed to facilitate device-independent configuration. However, these abstractions only provide device-independent capabilities and do not support service-independent config changes calculation. Our proposed abstraction, Trident, eliminates the need to maintain service-specific config changes calculation logic for each controller, greatly reducing OpEx.

The architecture of network control systems has also been widely researched. [17, 21] manage various services and heterogeneous devices of physical network in unified manner, which could notably reduces the OpEx of managing networks consists of diverse devices. [17–19] introduce intent-driven network management, which takes network management a step further. Andromeda [14] and Achelous [30] design the mechanism of configuring the devices in their VPC with the config changes. However, all the above works do not touch the config changes calculation procedure in the virtual network, whose optimization is first discussed in this paper.

Additionally, there are several notable efforts [11, 28, 29] dedicated to achieving rapid, highly scalable, and automated database-based incremental view maintenance. We acknowledge that the adoption of these methods could indeed improve the controller performance with SQL-based optimizations. While further SQL-based optimization of the controller may meet current performance requirements, it sets us on a path where future performance improvements become heavily reliant on SQL-database enhancements—a challenge we have struggled with for over a decade. To have a better assurance of controller performance with predictable improvement, we choose to optimize the whole architecture, especially in virtual network abstraction and API processing logic. In this way, the performance enhancement is guaranteed irrespective of the SQL-database we employ. Nevertheless, combining SQL-based optimizations [11, 28, 29] and architecture optimization proposed in this paper might be an interesting exploration and the community could be inspired by POSEIDON.

7 Conclusion

This paper presents POSEIDON, a pioneering effort to build a unified virtual network controller capable of managing a virtual network at the scale of millions of tenants. With POSEIDON, we can save the OpEx of managing numerous controllers without sacrificing flexibility of services iterations. The innovative config dependency abstraction enables the calculation of config changes independent of both service logic and physical devices. We have been operating Alibaba Cloud with POSEIDON over 3 years, and it has substantially mitigated the workload of adding services/devices, lowered the configuration latency, and reduced network incidents.

Acknowledgements: The work was partially supported by National Science Foundation of China (NSFC) under Grants (62293511, 62372053). We thank our shepherd Soudeh Ghorbani and the anonymous reviewers for their helpful comments that greatly improved the paper.

References

[1] Alibaba Cloud API. https://api.alibabacloud.com/home, 2023.

[2] AWS Cloud Control API, manage cloud infrastructure with a consistent set of APIs. https://aws.amazon.com/cloudcontrolapi/, 2023.

[3] Azure REST API reference . https://learn.microsoft.com/en-us/rest/api/azure/, 2023.

[4] Build cloud-native applications in Azure. https://azure.microsoft.com/en-us/solutions/cloud-native-apps/, 2023.

[5] Elastic IP Address in Alibaba Cloud. https://www.alibabacloud.com/en/product/eip, 2023.

[6] Gartner Says Worldwide IaaS Public Cloud Services Revenue Grew 30% in 2022, Exceeding 100 Billion for the First Time. https://www.gartner.com/en/newsroom/press-releases/2023-07-18-gartner-says-worldwide-iaas-public-cloud-services-revenue-grew-30-percent-in-2022-exceeding-100-billion-for-the-first-time, 2023.

[7] Google Cloud Platform APIs and references . https://cloud.google.com/compute/docs/apis, 2023.

[8] What is an EIP? https://www.alibabacloud.com/help/en/eip/product-overview/what-is-eip, 2024.

[9] M Bjorklund. RFC 6020: YANG-A Data Modeling Language for the Network Configuration Protocol. *Tail-f Systems*, 2010.

[10] M Bjorklund. RFC 7950: The YANG 1.1 Data Modeling Language, 2016.

[11] Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. DBSP: Automatic Incremental View Maintenance for Rich Query Languages. *Proceedings of the VLDB Endowment*, 16(7):1601–1614, 2023.

[12] Eric Hayden Campbell, William T Hallahan, Priya Srikumar, Carmelo Cascone, Jed Liu, Vignesh Ramamurthy, Hossein Hojjat, Ruzica Piskac, Robert Soulé, and Nate Foster. Avenir: Managing Data Plane Diversity with Control Plane Synthesis. In *USENIX NSDI 21*, pages 133–153, 2021.

[13] Huangxun Chen, Yukai Miao, Li Chen, Haifeng Sun, Hong Xu, Libin Liu, Gong Zhang, and Wei Wang. Software-Defined Network Assimilation: Bridging the Last Mile Towards Centralized Network Configuration Management with NAssim. In *Proceedings of the ACM SIGCOMM 2022*, pages 281–297, 2022.

[14] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, et al. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *USENIX NSDI 18*, pages 373–387, 2018.

[15] Danielle E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *USENIX NSDI 16*, pages 523–535, 2016.

[16] R Enns, M Bjorklund, J Schoenwaelder, and A Bierman. RFC 6241: Network configuration protocol (NETCONF), 2011.

[17] Andrew D Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, et al. Orion: Google's Software-Defined Networking Control Plane. In *USENIX NSDI 21*, pages 83–98, 2021.

[18] Victor Heorhiadi, Sanjay Chandrasekaran, Michael K Reiter, and Vyas Sekar. Intent-Driven Composition of Resource-Management SDN Applications. In *Proceedings of the ACM CoNEXT 2018*, pages 86–97, 2018.

[19] Arthur S Jacobs, Ricardo J Pfitscher, Rafael H Ribeiro, Ronaldo A Ferreira, Lisandro Z Granville, and Sanjay G Rao. Deploying Natural Language Intents with Lumi. In *Proceedings of the ACM SIGCOMM 2019 Posters and Demos*, pages 82–84, 2019.

[20] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. Onix: A Distributed Control Platform for Large-scale Production Networks. In *USENIX OSDI 10*, 2010.

[21] Hongqiang Harry Liu, Xin Wu, Wei Zhou, Weiguo Chen, Tao Wang, Hui Xu, Lei Zhou, Qing Ma, and Ming Zhang. Automatic Life Cycle Management of Network Configurations. In *Proceedings of the ACM SIGCOMM 2018 Workshop on SelfDN*, pages 29–35, 2018.

[22] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[23] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. OpenDaylight: Towards a Model-Driven SDN Controller Architecture. In *Proceeding of the IEEE WoWMoM 2014*, pages 1–6. IEEE, 2014.

[24] Jeffrey C Mogul, Drago Goricanec, Martin Pool, Anees Shaikh, Douglas Turk, Bikash Koley, and Xiaoxue Zhao. Experiences with Modeling Network Topologies at Multiple Levels of Abstraction. In *USENIX NSDI 20*, pages 403–418, 2020.

[25] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, et al. Sailfish: Accelerating Cloud-Scale Multi-Tenant Multi-Service Gateways with Programmable Switches. In *Proceedings of the ACM SIGCOMM 2021*, pages 194–206, 2021.

[26] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. Ananta: Cloud Scale Load Balancing. *ACM SIGCOMM Computer Communication Review*, 43(4):207–218, 2013.

[27] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky HY Wong, and Hongyi Zeng. Robotron: Top-down Network Management at Facebook Scale. In *Proceedings of the ACM SIGCOMM 2016*, pages 426–439, 2016.

[28] Debnil Sur, Ben Pfaff, Leonid Ryzhyk, and Mihai Budiu. Full-stack SDN. In *Proceedings of the ACM HotNets 2022*, pages 130–137, 2022.

[29] Lalith Suresh, João Loff, Faria Kalim, Sangeetha Abdu Jyothi, Nina Narodytska, Leonid Ryzhyk, Sahan Gamage, Brian Oki, Pranshu Jain, and Michael Gasch. Building Scalable and Flexible Cluster Managers Using Declarative Programming. In *USENIX OSDI 20*, pages 827–844, 2020.

[30] 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. Achelous: Enabling Programmability, Elasticity, and Reliability in Hyperscale Cloud Networks. In *Proceedings of the ACM SIGCOMM 2023*, pages 769–782, 2023.

[31] Shunmin Zhu, Jianyuan Lu, Biao Lyu, Tian Pan, Chenhao Jia, Xin Cheng, Daxiang Kang, Yilong Lv, Fukun Yang, Xiaobo Xue, et al. Zoonet: A Proactive Telemetry System for Large-Scale Cloud Networks. In *Proceedings of the ACM CoNEXT 2022*, pages 321–336, 2022.

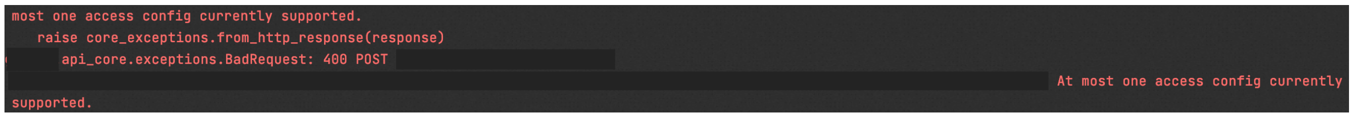
Appendices

A Additional Figures

End-to-end completion time for disabling EIP. Fig. A2 illustrates the completion time of disabling EIP under different TPS. Compared to Vendor A, our cloud is less affected by TPS. The P50, P90 and P99 of Vendor A increase dramatically (about 8x, 15x and 16x compared to their lowest value, respectively). Meanwhile, the P50, P90 and P99 of ours only experience a slight increase, that is, the increase rate is 1.02x, 1.9x and 3.1x. respectively. The P50, P90 and P99 of Vendor B remain stable throughout all the tests, but it still cannot measure data above 200TPS. Moreover, Vendor A performs better than Vendor B at lower TPS.

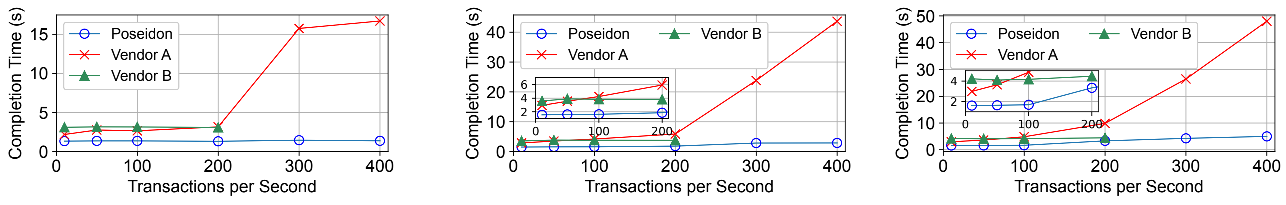
For Vendor A, the completion time of enabling EIP (as shown in Fig. 11) is higher than that of disabling EIP, whereas for us and Vendor B, it is the other way around. When conducting enabling and disabling EIP in our cloud, both server and gateway require configuration. For the case of enabling EIP, both devices must be configured successfully for a successful ping. While for the case of disabling EIP, a unsuccessful ping is achieved when either device has been configured. Therefore, the completion time of disabling EIP is less than that of enabling EIP in our cloud.

Resource usage analysis. While theoretically we can use a single POSEIDON controller to serve all service-related controllers, it may lead to an explosion radius affecting all services. To minimize the explosion radius, we deploy independent POSEIDON controller for each service-related controller. However, the development and maintenance of these POSEIDON controllers are managed by a single team to reduce the OpEx. This low-sharing deployment may not effectively improve the overall resource consumption and, in some cases, may even increase it. We conducted a comprehensive analysis of CPU and memory consumption of the VPC controller with and without POSEIDON, as well as the resource usage of POSEIDON for VPC controller. Fig. A3 shows the total resource usage for processing all the API of VPC. The POSEIDON in the figure represents the total resource usage of VPC controller and POSEIDON controller. As shown in the figure, the CPU and memory usage of POSEIDON is 2.5x and 1.75x than that of prior-gen VPC controller. Based on our observations, the resource usage of the VPC controller halved because lots of works have been taken over by the POSEIDON controller. However, the works taken over by POSEIDON involve are the CPU- and memory-intensive tasks, *e.g.*, config changes calculation. In order to provide significantly improved config changes calculation performance compared to the prior-gen, the resource consumption of the POSEIDON's config changes calculation component is slightly higher than that of prior-gen VPC controller. In summary, the total resource usage of POSEIDON and VPC controller is higher than that of prior-gen



```
most one access config currently supported.
    raise core_exceptions.from_http_response(response)
[redacted] api_core.exceptions.BadRequest: 400 POST [redacted]
[redacted] At most one access config currently
supported.
```

Figure A1: Error of testing Vendor B for 400TPS.



(a) P50 (b) P90 (c) P99

Figure A2: End-to-end completion time of API calling for disabling EIP.

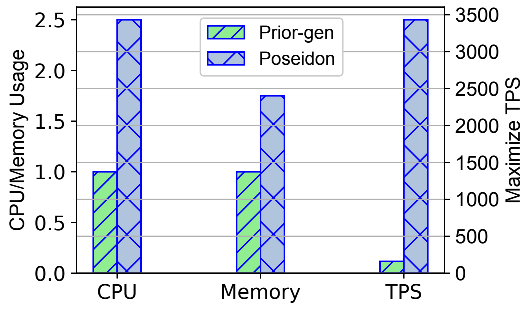


Figure A3: Resource usage increase and concurrent processing capacity improvement.

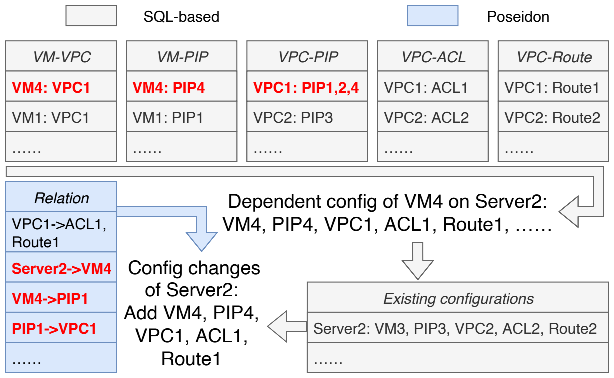


Figure A4: Config changes calculation (SQL-based vs. POSEIDON).

VPC controller. However, we believe that this trade-off is well worth it because the concurrent processing capacity of config changes calculation has increase by about 21x, as shown in Fig. A3.

B Supplementary Materials

B.1 Design goal achievement analysis of abstraction

Why Trident-based config changes calculation is service-independent. According to the Algo. 1, it can be observed that our code for config changes calculation is the same for any API, meaning it is independent of the service.

Why Trident-based config changes calculation is device-independent. Since we abstract all heterogeneous devices into a standardized object type, namely "Device", our implementation of config changes calculation and pushing is agnostic to the specific devices. The device-specific translation is done by the unified agent.

Why SQL-based and Trident-based config changes calculation is equivalent. To guarantee the equivalence, we analyze the essence of the cloud control APIs and their execution. The majority of cloud control APIs can be regarded as the CRUD (create/read/update/delete) of tenant resources (see §B.2). The execution of cloud control APIs is to translate tenant intents for resource manipulation into configurations on the physical device. As mentioned in §2.1, the controller will make config changes calculation for each cloud control API, which can further be divided into two steps: (1) find all the dependent configs of the target tenant resource; (2) search a table recording all the existing configs on each device to calculate the necessary changes to install. The original API execution approach is service-dependent since the key/value of each table and the table query sequence need prior knowledge of the related service. For example, in Fig. A4, for the API "create VM4 in VPC1", by sequentially querying five tables, its dependent configs of "VM4, PIP4, VPC1, ACL1, Route1" can be obtained. By querying the existing configs of the device, the necessary changes can be calculated as "Add VM4, PIP4, VPC1, ACL1, Route1 on Server2". Parsing the API with SQL needs a deep understanding of the service logic. To guarantee the equivalence, the information of dependent configs and existing configs can also be obtained through the implementation of Trident. Specifically, we can get the needed information by searching for the relation table, as shown in Fig. A4. This procedure is service-independent since the query table is same for all APIs (*i.e.*, Relation table).

B.2 Trident operations of cloud control API

Based on our analysis of the cloud control APIs provided by major cloud vendors (including ourselves), most of them

Table A1: API of configuring Public (Elastic) IP.

	Create	Delete	Update	Read
AWS	1	3	2	5
GCP	1	1	1	2
Azure	4	1	1	3
Ali Cloud	5	4	3	8

Algorithm 2: Abstraction of API: Create Resource1

```
1 Create $Conf_{R1}$ of Resource1
2 for $C$ in $Conf_{R1}$ 's dependant Conf do
3     Relate $Conf_{R1}$ to $C$
4 for $G$ in $Conf_{R1}$'s target Group that needs to be configured do
5     Relate $G$ to $Conf_{R1}$
```

are Create/Delete/Update/Read, as shown in Table A1. In addition, there are a few other APIs such as Associate and Disassociate. Algo. 2,3,4,5 detail how to abstract the main cloud control API into Trident operations.

B.3 Computational complexity analysis of OR-based calculation

The OR-based ref calculation has excessively high computational complexity. When $ref(Group, P_i)$ changes, recalculation is needed to refresh $ref(Group, Conf_A)$. The change of $ref(Group, P_i)$ can have two cases: *i.e.*, $ref(Group, P_i)$ turns from 0 to 1 or 1 to 0. For the 0 to 1 case, according to the OR calculation rule, as long as at least one $ref(Group, P_i)$ is 1, $ref(Group, Conf_A)$ is 1. Hence, we can directly infer that $ref(Group, Conf_A)$ is 1 without complex computation. By contrast, for the 1 to 0 case, we cannot assert that $ref(Group, Conf_A)$ becomes 0 unless we perform the OR calculation to each $ref(Group, P_i)$. In another case, if one parent node is unrelated due to tree changes, because the OR operation does not satisfy reversibility, once a bitwise OR operation is performed, it cannot simply be reversed to obtain the value before this parent node was included. In other words, it's not possible to determine the updated OR value based on the change in a parent node's ref and the current OR value. Because it is necessary to perform a bitwise OR on the ref values of all parent nodes to obtain the OR result, the complexity is proportional to the number of parents as $O(P_i)$.

B.4 Computational complexity analysis of SUM-based calculation

To address the performance issue of OR-based ref calculation, we can adopt SUM-based solution to inherit Conf's ref from its parents. The advantage of SUM is that during calculation, we can refresh Conf's current ref based on the change of its parent's ref and Conf's past ref. In SUM-based solution, if $ref = 0$, it indicates no descendant relation, while in other

Algorithm 3: Abstraction of API: Delete Resource1

```
1 for $C1$ in $Conf_{R1}$ 's children do
2     Unrelate $Conf_{R1}$ to $C2$
3 for $C2$ in $Conf_{R1}$ 's parents Conf do
4     Unrelate $C2$ to $Conf_{R1}$
5 Delete $Conf_{R1}$
```

Algorithm 4: Abstraction of API: Update Resource1

```
1 Update $Conf_{R1}$ of Resource1
```

cases, it indicates the presence of a descendant relation. At the start, we initialize $ref(Group, Group.children)$ to 1 and continuously propagate this value to the remaining Confs according to the SUM rule, as defined by $ref(Group, Conf_A) = \Sigma_i ref(Group, P_i)$.

Since SUM adheres to the reversibility property, for scenarios where a parent node is removed, the new result can be obtained simply by subtracting that parent node's ref. Similarly, for scenarios where a new parent node is added, the result can be updated by simply adding the ref of that new parent node. Moreover, for scenarios where the ref of a parent node changes, the $ref(Group, Conf_A)$ can also be expressed as $ref(Group, P_x) + \Sigma_{i \neq x} ref(Group, P_i)$. When only $ref(Group, P_x)$ undergoes a change, we can deduce that $\Delta ref(Group, Conf_A) = \Delta ref(Group, P_x)$. It implies that when a $ref(Group, P_x)$ undergoes a change, $ref(Group, Conf_A)$ will experience a similar change. For instance, when a $ref(Group, P_x)$ transitions from 0 to 3, $ref(Group, Conf_A)$ will also increase by 3 accordingly. Since the refreshing mechanism focuses solely on the change of a specific parent, the computational complexity is only $O(1)$.

B.5 Case study of POSEIDON

To clearly illustrate the workflow of POSEIDON, we present a step-by-step introduction using the API call of "create VM4 in VPC1" as an example.

Step 1: After the user issues this API call, service-related controllers (*e.g.*, VPC controller in Fig. 5 in this example) will map the API into Trident operations. This step is service-related and generates different Trident operations based on the processing logic specific to each service API. The Trident operations corresponding to this API are shown as the blue color text in Fig. 6.

Step 2: Subsequently, upon receiving the Trident operations, the service-independent unified controller will add/delete/update Confs or modify their relations according to the operations. The changes incurred by the above operations are depicted in blue in Fig. 7.

Step 3: Then, based on the nodes/topology changes in the Trident tree, we calculate the appropriate Trident changes for Groups. §3.4.2 details the procedure for this example.

Algorithm 5: Abstraction of API: Associate Resource1 to Resource2

```
1 Relate $Conf_{R1}$ of Resource1 to $Conf_{R2}$ of Resource2
```

Step 4: Finally, the unified controller will push the Trident changes to the unified agent residing on the devices within Groups. Specifically, this example involves pushing "Add ACL1, Route1, VPC1, PIP4, VM4" to Server2 and "Update FlowTable1" to GW1 and GW2, according to Fig. 6 and Fig. 7. Upon receiving the Trident changes, a program on the agent translates them into device-specific primitives to enable the user's intent.

B.6 Redis failover detection and data recovery

Unlike the real-time data storage in the database, Redis caches the latest data in memory and performs batch storage. When failover occurs, the cached data in memory will be lost. Therefore, we need to detect the failover and conduct data recovery. We inject lots of probes and accumulate them at the frequency of batch storage. When probes' data meet expectations, this batch of data are stored in Redis. Otherwise, this batch of data must be lost due to failover. We can locate the batch of data affected by failover according to the timestamp of the probe. Then, the data obtained from the database will be re-pushed.

C Additional Experiences

C.1 How to choose pushing and pulling when configuring devices?

Most traditional controllers deliver configurations to the physical devices in a *pushing* mode, namely, the controller actively pushes config changes to devices so that user's intent can take effect in a timely manner. However, in large-scale public clouds, the pushing method will cause large pressure on systematical performance (*e.g.*, TPS, southbound bandwidth), as there are enormous underlying devices. What's worse, users may frequently modify their network, and each modification may consist of thousands of underlying config changes (*e.g.*, a newly added route entry may involve thousands of VM host servers). Another way to deliver configurations is the *pulling* method, where devices pull configs from the controller on-demand or periodically [14, 30]. This method eliminates the above performance bottlenecks at the cost of adding configuration delay. POSEIDON employs the pushing method for most configuration delivery, and utilizes the on-demand pulling method for FlowTable. This is because communications between VMs usually occur within a small subset, so most information in the FlowTable will not be queried by the device. Besides, in our production enviroment, the additional latency caused by FlowTable configuration delay is too short to be perceived by users.

C.2 How to deploy POSEIDON to a small-scale virtual network?

POSEIDON is an SDN architecture designed for high-performance control of large-scale virtual networks. Its Trident abstraction is implemented through a Trient Tree structure maintained in memory, cache, and database. However, for a small-scale virtual network with medium performance requirements, POSEIDON may be over-engineered and cost-prohibitive. For example, a standard deployment of POSEIDON requires at least 8 high-performance physical servers to support virtual network control in a data center of 10,000 servers. However, the number of servers may be less than 100 in some edge regions or small private cloud environments. (Even the number of servers in some cluster is less than 20 in some edge POP site.) In such a small-scale deployment, POSEIDON's architecture needs to be optimized to reduce resource usage, especially memory usage. One possible idea is to design a variant of Trident tree with high compression and efficient data exchange. The Tridient tree currently implemented by tree structure requires that all configuration relationships should be maintained in memory, which is a performance-first but cost-ignoring approach. Therefore, limited memory resources in small-scale scenarios can be solved by compressing Trident data, or even maintaining only part of the relationship in memory through data exchange.