nsdi '24

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, Kunpeng Zhou, Zhigang Zong, Xing Li,

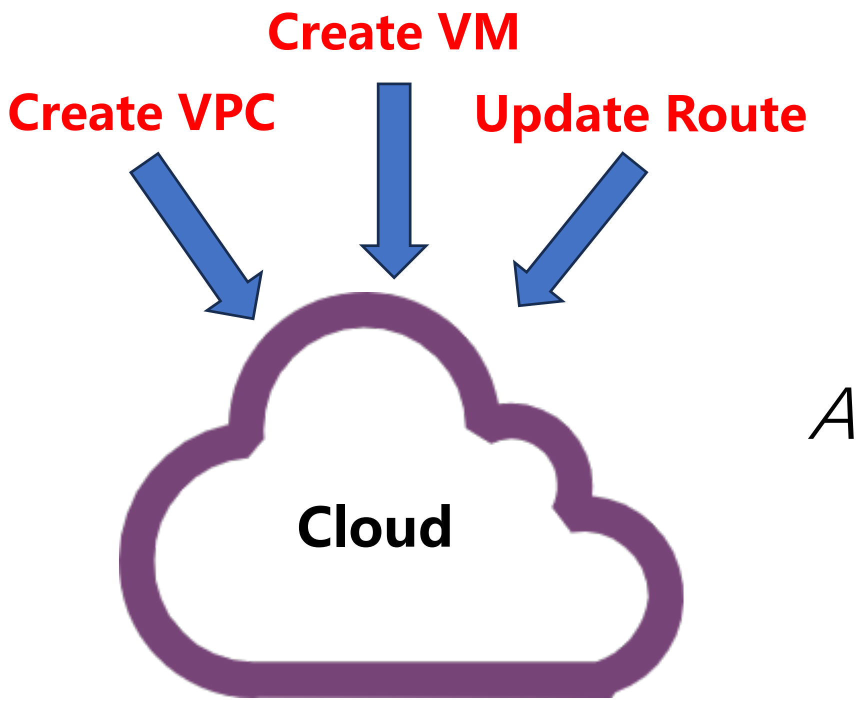
Guangwang Li, Pengyu Zhang, Peng Cheng, Jiming Chen, and Shunmin Zhu.

浙江大學 ZHEJIANG UNIVERSITY

Alibaba Cloud

清華大學 Tsinghua University

Background: Configure Cloud via Controller



Create VM

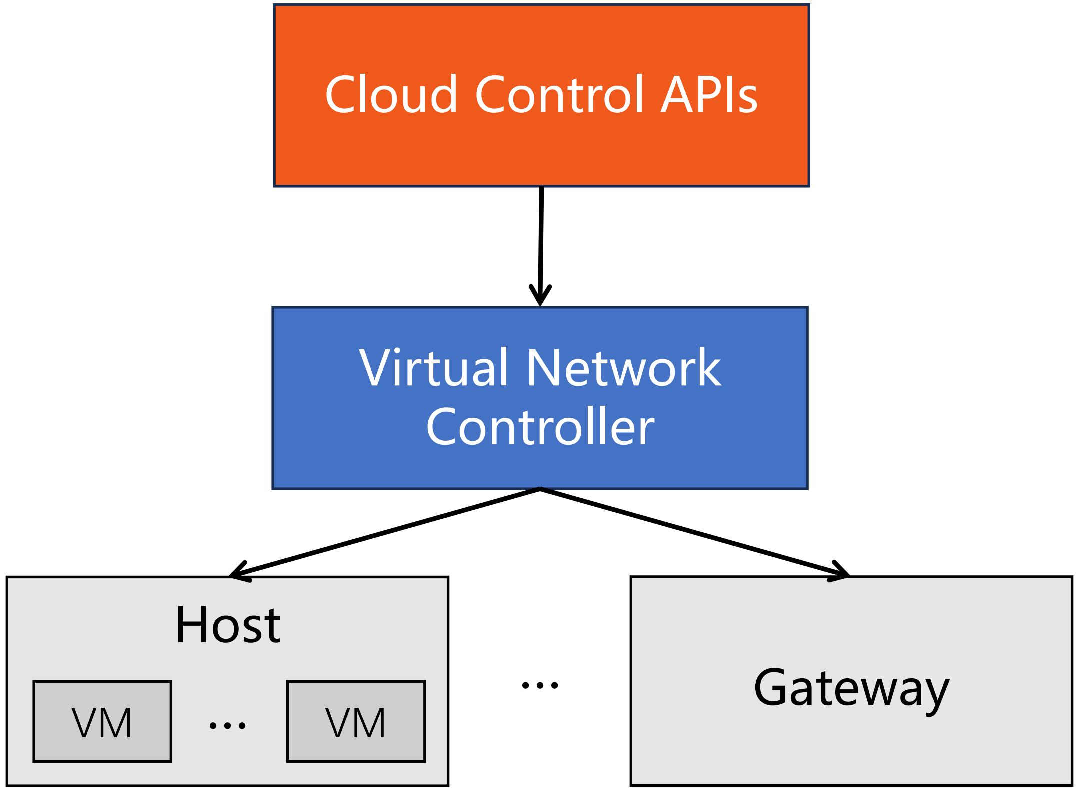
Create VPC

Update Route

Cloud

User leverages **cloud control APIs** to manage their network

After that …



Cloud Control APIs

Virtual Network Controller

Host

VM

…

VM

…

Gateway

Upon receiving APIs, **virtual network controller** updates the configurations of **physical devices**

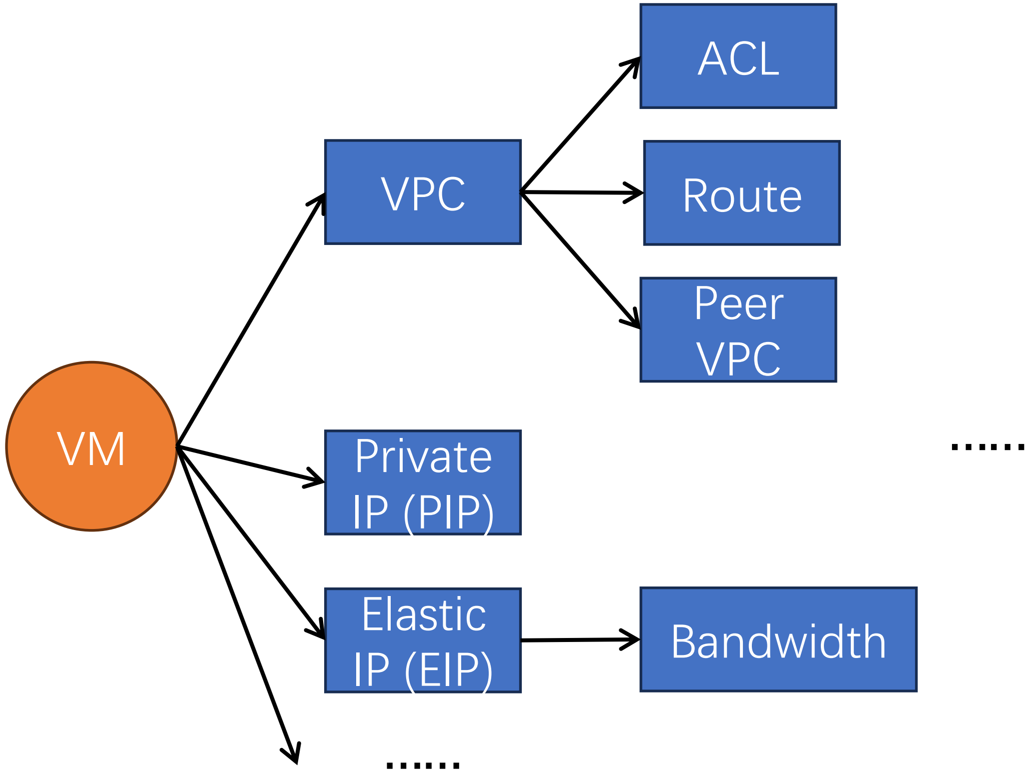
Background: Workflow of Controller

①Receive the cloud control APIs

②Calculate device config changes

a) Identify all the **dependent configs**

③Configure physical device



There are complex dependencies between virtual network configurations.

Background: Workflow of Controller

①**Receive the cloud control APIs**

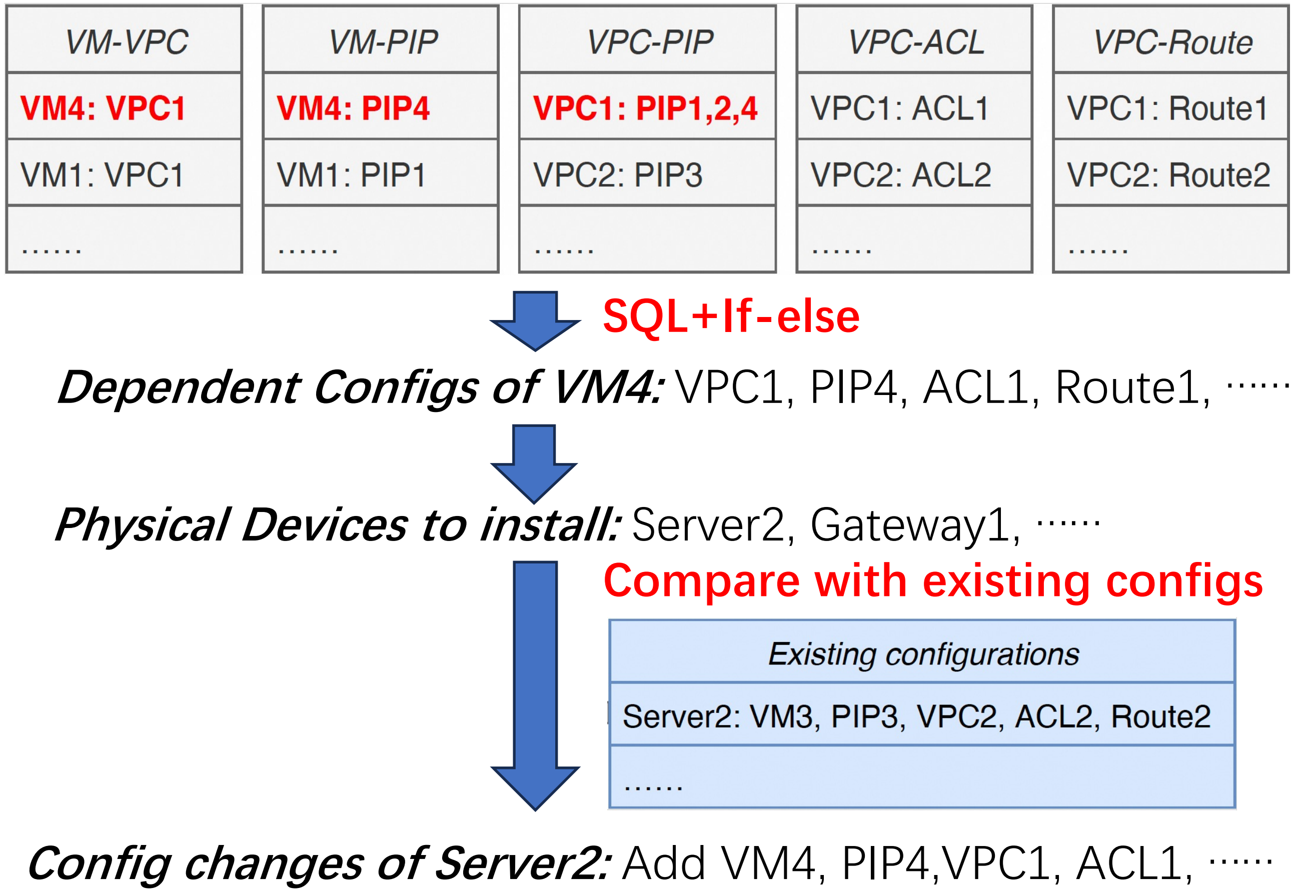
②**Calculate device config changes**

a) Identify all the **dependent configs**

b) Identify the **physical devices** onto which the configs must be installed

c) Config **changes** calculation

③**Configure physical device**



VM-VPC	*VM-PIP*	*VPC-PIP*	*VPC-ACL*	*VPC-Route*
VM4: VPC1	**VM4: PIP4**	**VPC1: PIP1,2,4**	VPC1: ACL1	VPC1: Route1
VM1: VPC1	VM1: PIP1	VPC2: PIP3	VPC2: ACL2	VPC2: Route2
……	……	……	……	……

SQL+If-else

Dependent Configs of VM4: VPC1, PIP4, ACL1, Route1, ……

Physical Devices to install: Server2, Gateway1, ……

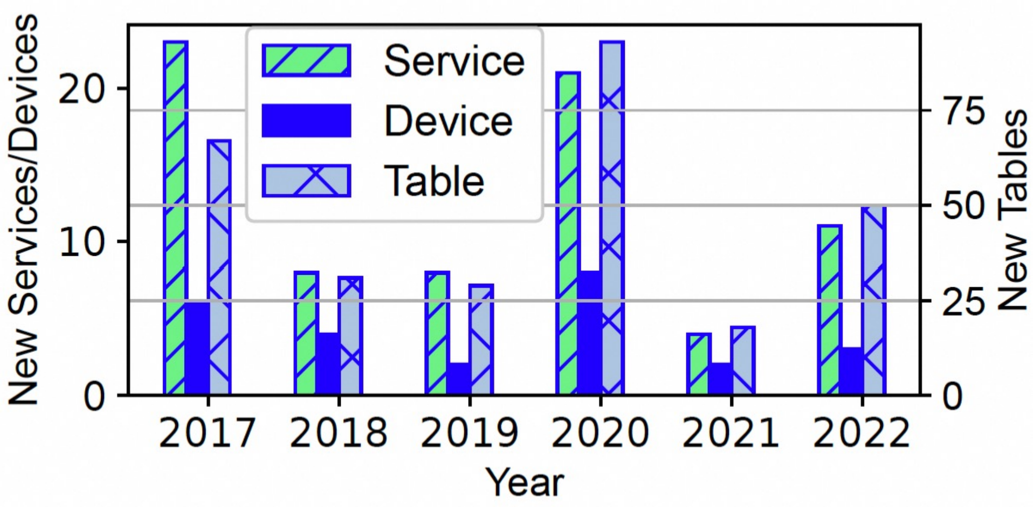
Compare with existing configs

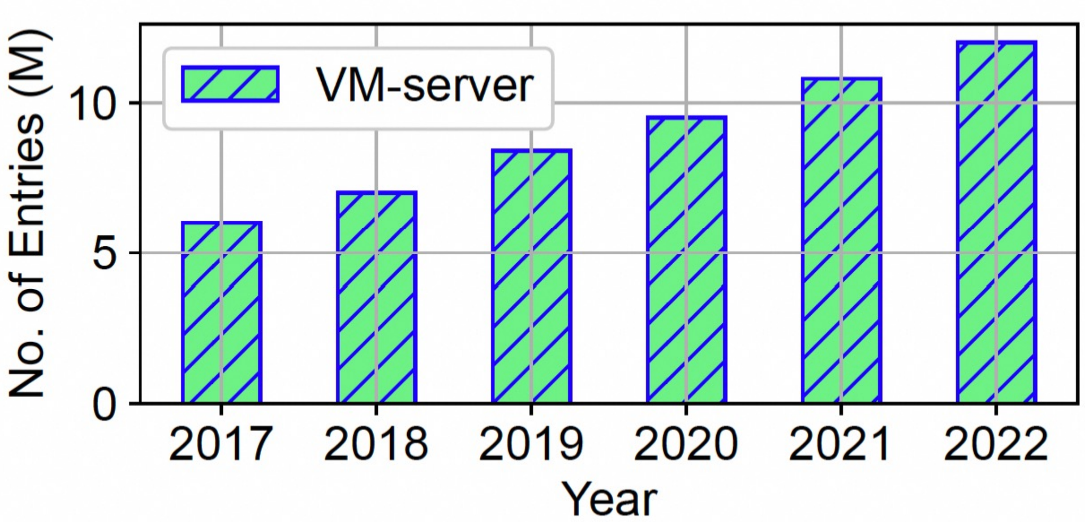
Existing configurations
Server2: VM3, PIP3, VPC2, ACL2, Route2
……

Config changes of Server2: Add VM4, PIP4,VPC1, ACL1, ……

Challenge#1: Performance Degradation

——Performance degradation due to longer search chain and larger table size





Cloud venders provide more services

Lots of new devices/tables are added

Longer database search chain

Number and scale of virtual networks increase

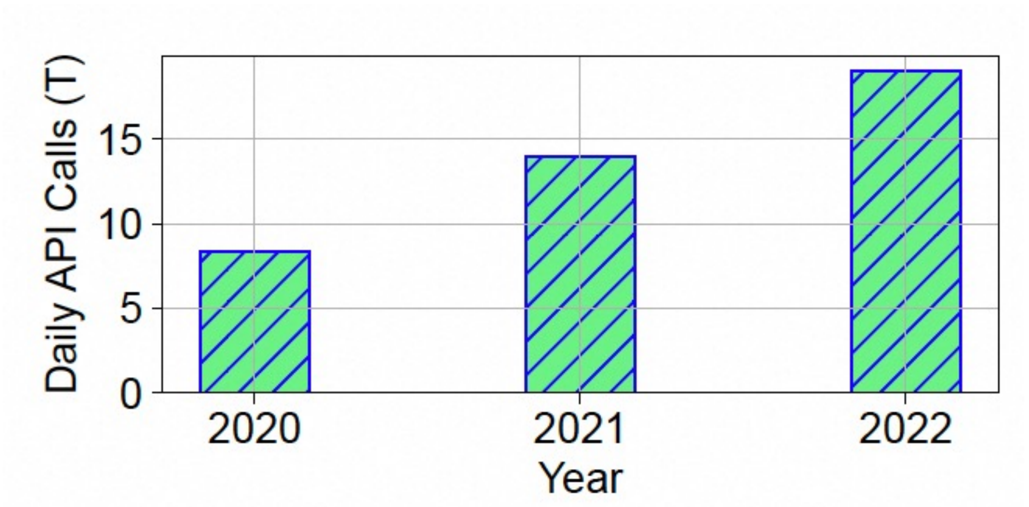
Number of entries in major tables grows

Longer database search/update latency

P99 latency of calculating config changes in our cloud has nearly doubled.

Challenge#2: Rapid Growth of Workload

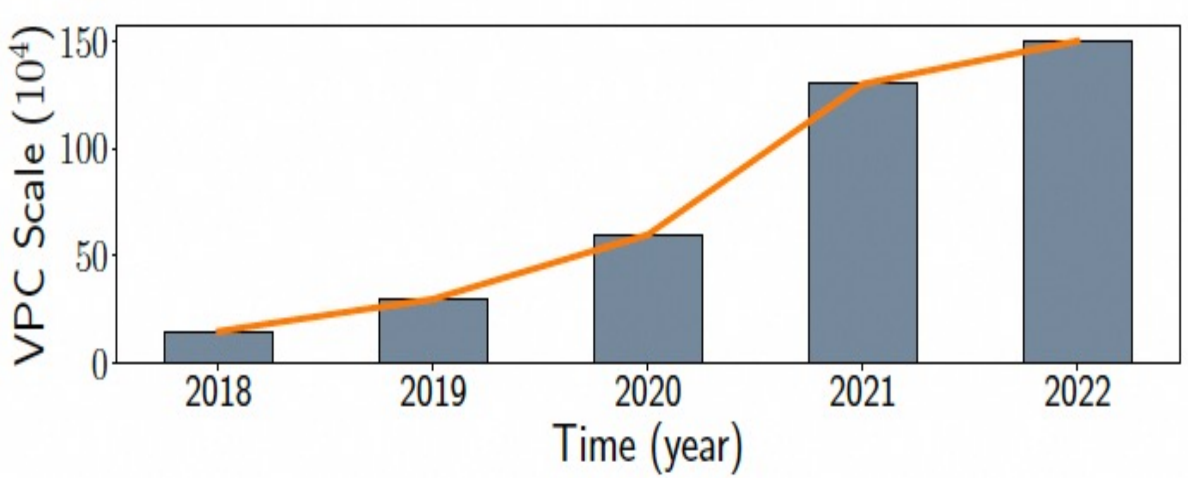
——Rapid growth of northbound API calls and southbound devices



Usage of cloud resources by tenants increases

More resources need to be managed/configured

More Northbound API calls



The scale of an individual VPC increases

An individual VPC covers more southbound devices

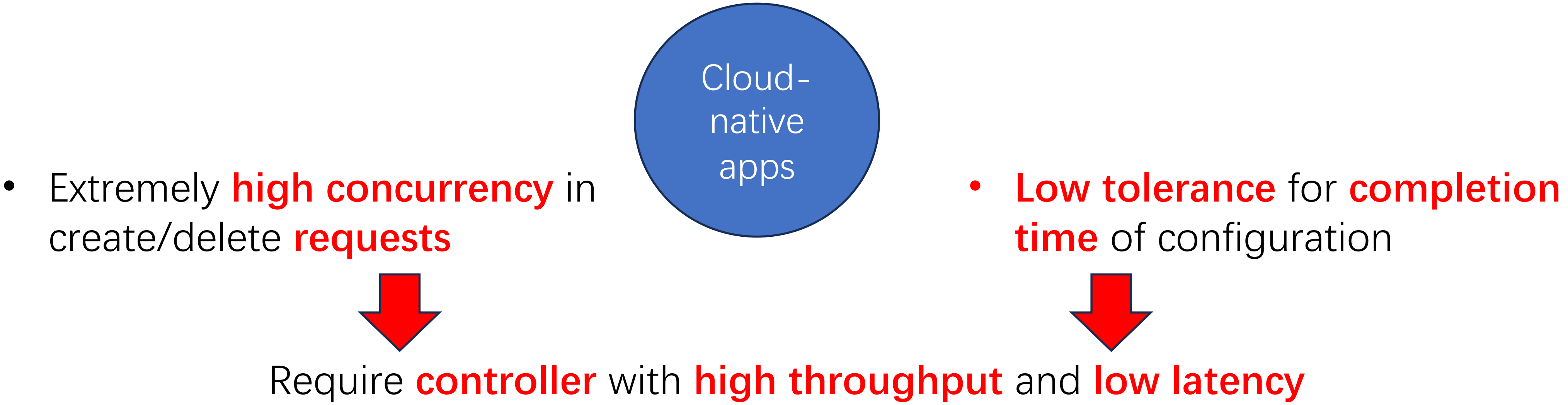
More southbound devices to configure

The daily API calls have doubled in two years, reaching ~10,000,000,000 calls.

A single ACL rule may reach more than 100,000 servers.

Challenge#3: Cloud-Native Burst Requests

——Cloud-native apps intensify the performance requirements



Cloud-native apps

- Extremely **high concurrency** in create/delete **requests**

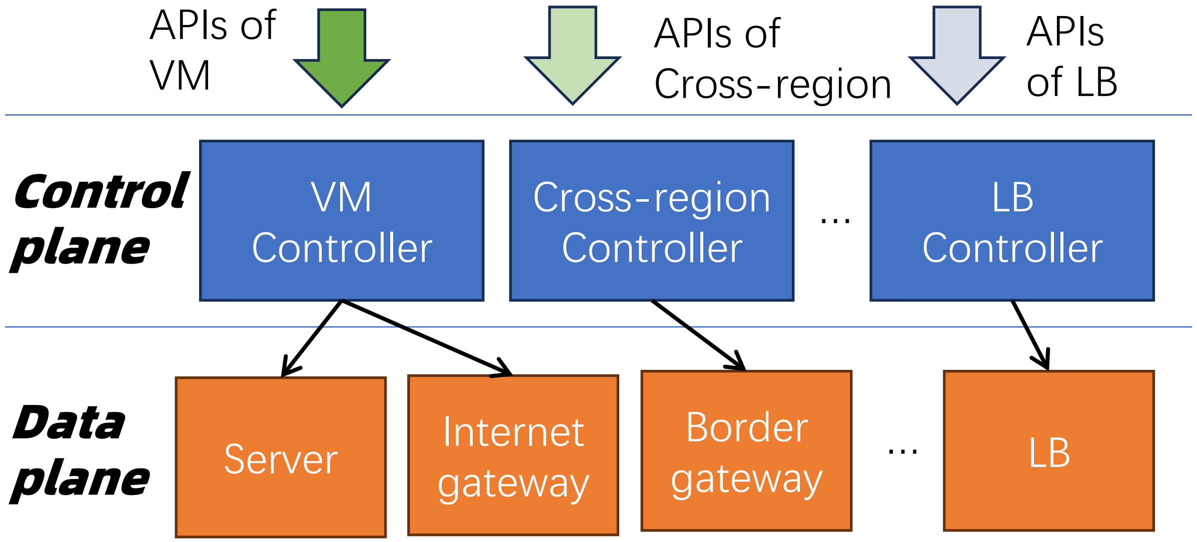
- **Low tolerance** for **completion time** of configuration

Require **controller** with **high throughput** and **low latency**

Case1: In **ecommerce business**, resources will be massively scaled up (e.g., **tens of thousands of containers**) just **before the peak arrives**.

Case2: **Social media applications** need to handle surges during hot events. **Thousands of backends** need to be elastically scaled in **a short interval (e.g., 500ms).**

Challenge#4: High Code Redundancy and OpEx



Controller	Lines of Codes
LB1	167K
LB2	76.9K
VPC	873K
NAT	107K
VPN	97K
Private Link	31.8K
Accelerator	135K

Flexibility

Individual Controllers for each service → More than 50 controllers developed over years. → **High OpEx** and **Code Redundancy** across controllers.

Design Goals and Overview of Poseidon

Design goals

1. Improve API throughput and latency
2. Reduce the OpEx

Design overview

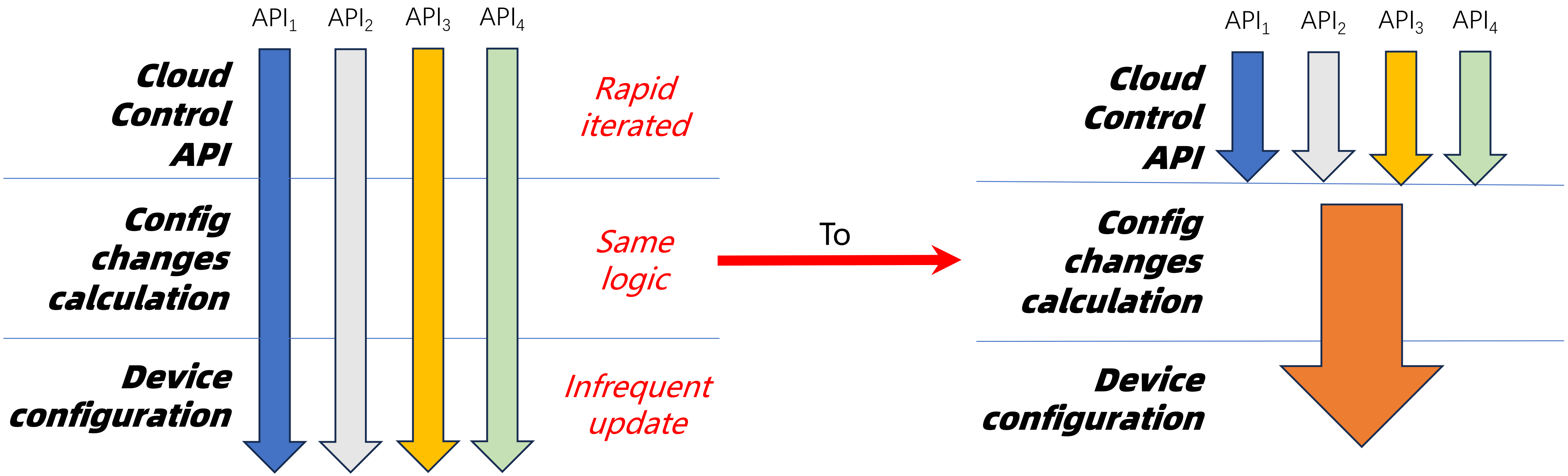
- ***Architecture:*** Partially consolidate the common modules of separate controllers into a unified Poseidon controller

- ***Abstraction:*** We propose service- and device-independent abstraction to unify the management of heterogeneous devices and diverse services’ APIs;

- ***Acceleration:*** To accelerate config calculation, we design a Tree-based config changes calculation logic

Design#1: Partial consolidation architecture

——Observation and our choice



We choose

1. **Leave** the development and maintenance of **cloud control APIs** to **each service**
2. **Consolidate** the implementations of **config changes calculation** and **device configuration**

Design#1: Partial consolidation architecture

——Observation and our choice

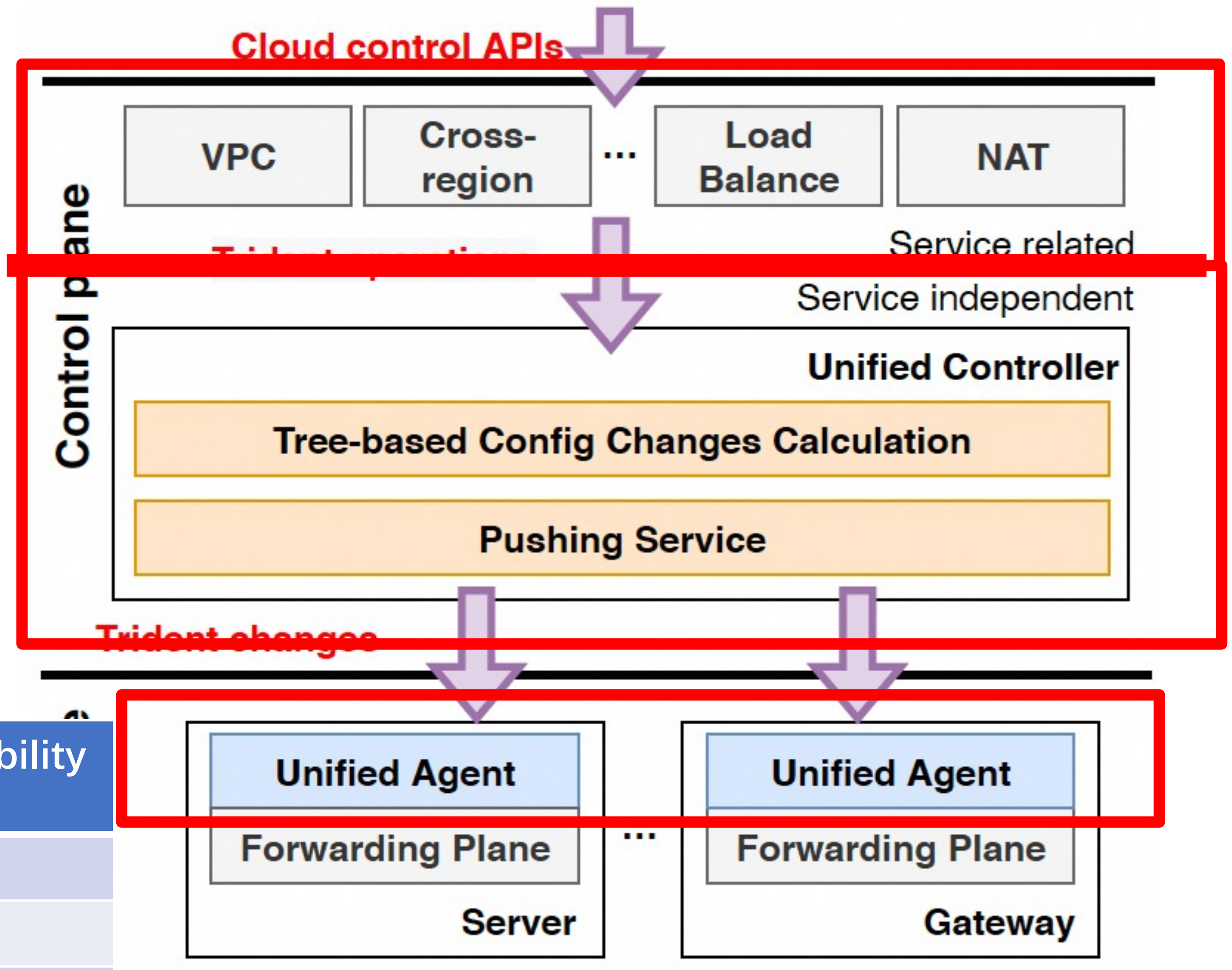
Partition the control plane into 2 layers

- □ **Service-related layer**: For processing APIs
- □ **Service-independent layer**: For calculating/pushing config changes

Unified agent on heterogeneous devices

- □ For translating unified config changes into underlying primitives

	OpEx and Code redundancy	Flexibility
Individual	High	High
Full consolidation	Low	Low
Partial consolidation	Low	High

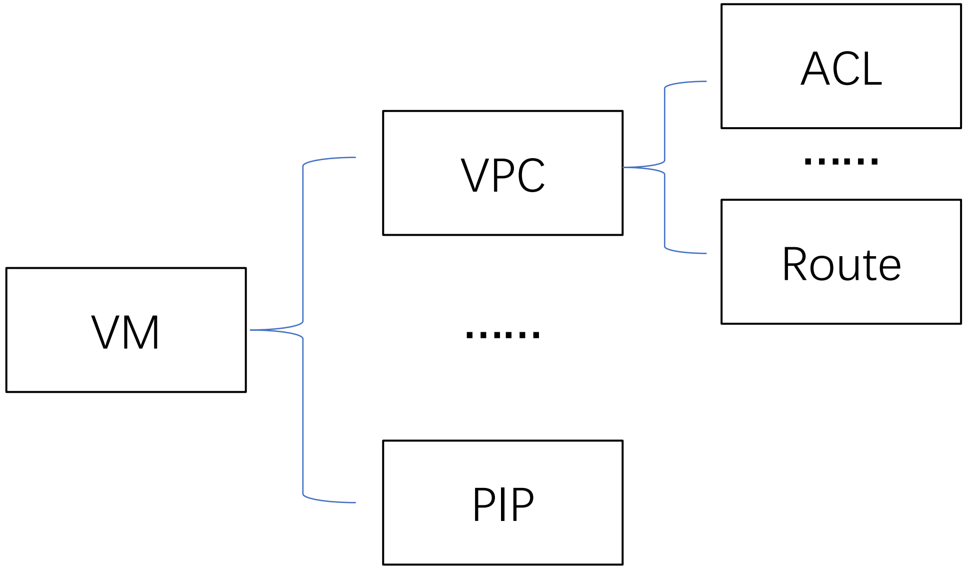


Design#2: Trident abstraction

——Design Consideration

Objects in Controller:

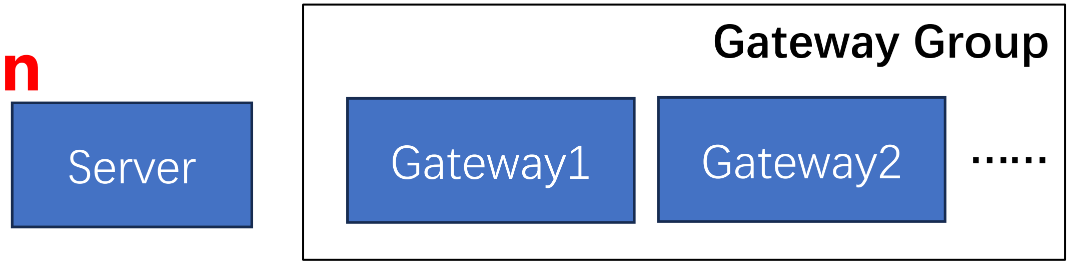
- **Service: VPC、LB、NAT、etc.**
- **Config: ACL、Route、etc.**
- **Device：LVS、Router、Switch 、etc.**



Core Actions:

□ **CRUD config/device and their relation**

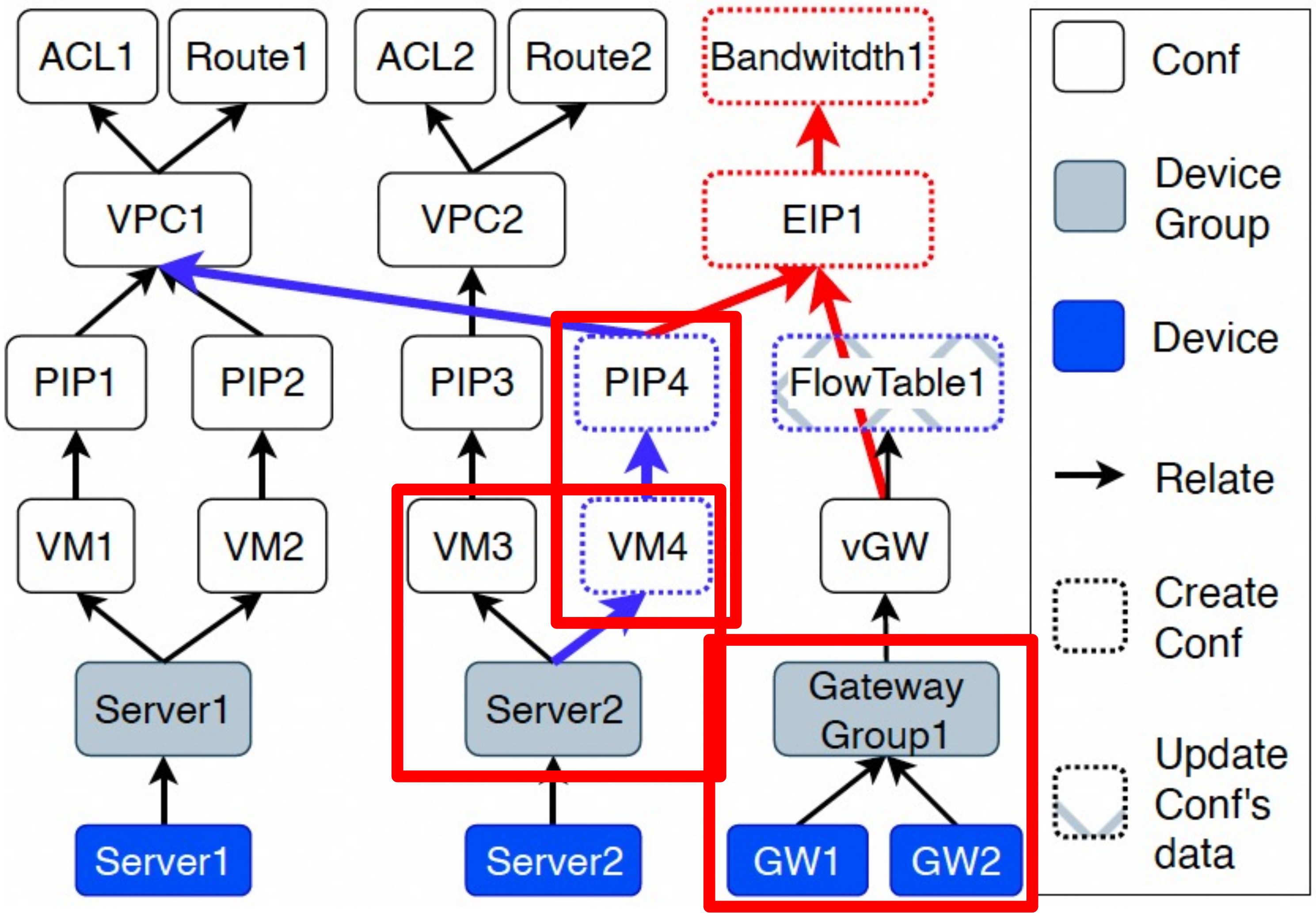
□ **Install config to device**



We choose to abstract config, device and their relation in a unified way

Design#2: Trident abstraction

——Design Details



Three basic objects:

- □ **Conf:** config on device
- □ **Device:** physical device
- □ **Device Group:** devices with same configs

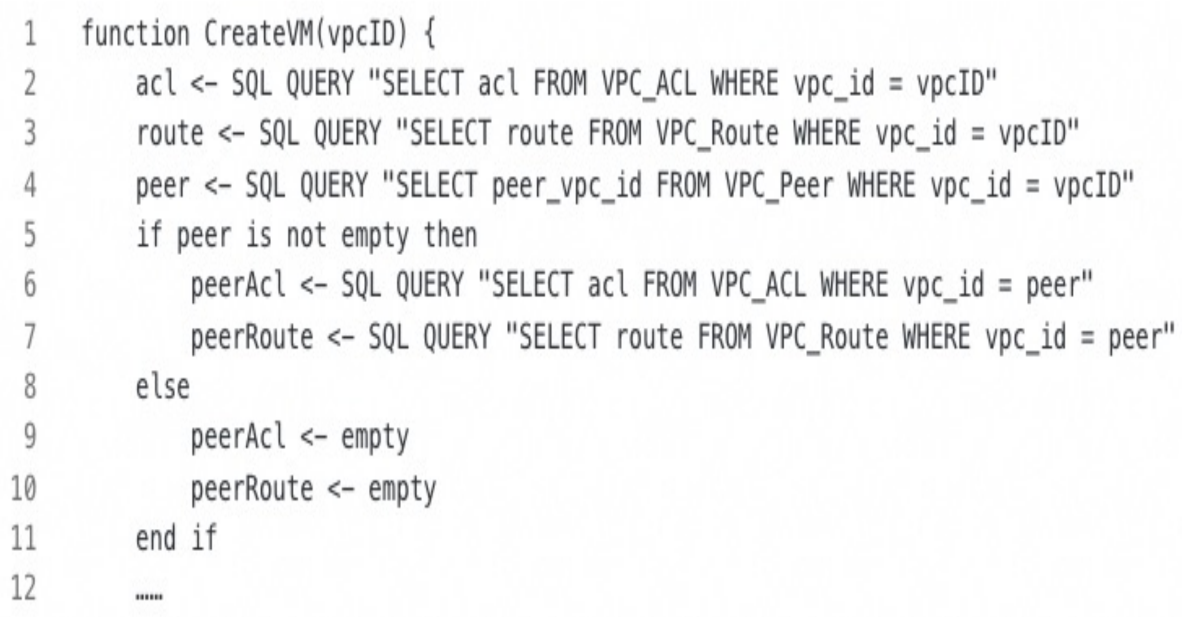
Five atomic operations:

- □ **Create/Delete/Update:** traditional atomic operations
- □ **Relate/Unrelate:** modify the relation between objects

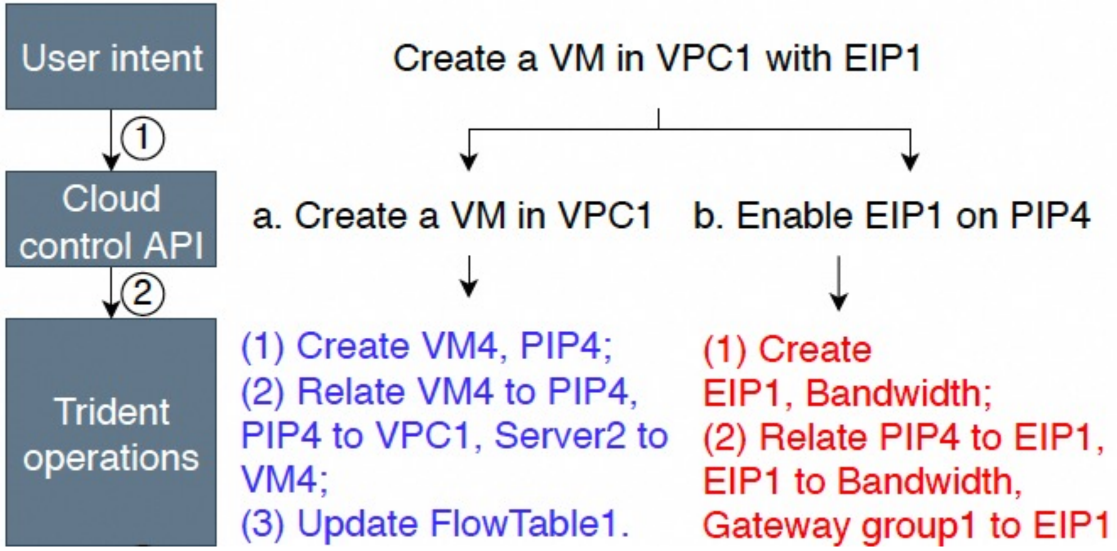
Relation between:

- ➢ **two Confs**: one depend on another
- ➢ **Device** and **Group**: holding same configs with other devices in group
- ➢ **Group** and **Conf**: devices in the group need to install this Conf

Design#2: Trident abstraction



```
function CreateVM(vpcID) {
    acl <- SQL QUERY "SELECT acl FROM VPC_ACL WHERE vpc_id = vpcID"
    route <- SQL QUERY "SELECT route FROM VPC_Route WHERE vpc_id = vpcID"
    peer <- SQL QUERY "SELECT peer_vpc_id FROM VPC_Peer WHERE vpc_id = vpcID"
    if peer is not empty then
        peerAcl <- SQL QUERY "SELECT acl FROM VPC_ACL WHERE vpc_id = peer"
        peerRoute <- SQL QUERY "SELECT route FROM VPC_Route WHERE vpc_id = peer"
    else
        peerAcl <- empty
        peerRoute <- empty
    end if
    ......
```



User intent

①

Cloud control API

②

Trident operations

Create a VM in VPC1 with EIP1

a. Create a VM in VPC1

b. Enable EIP1 on PIP4

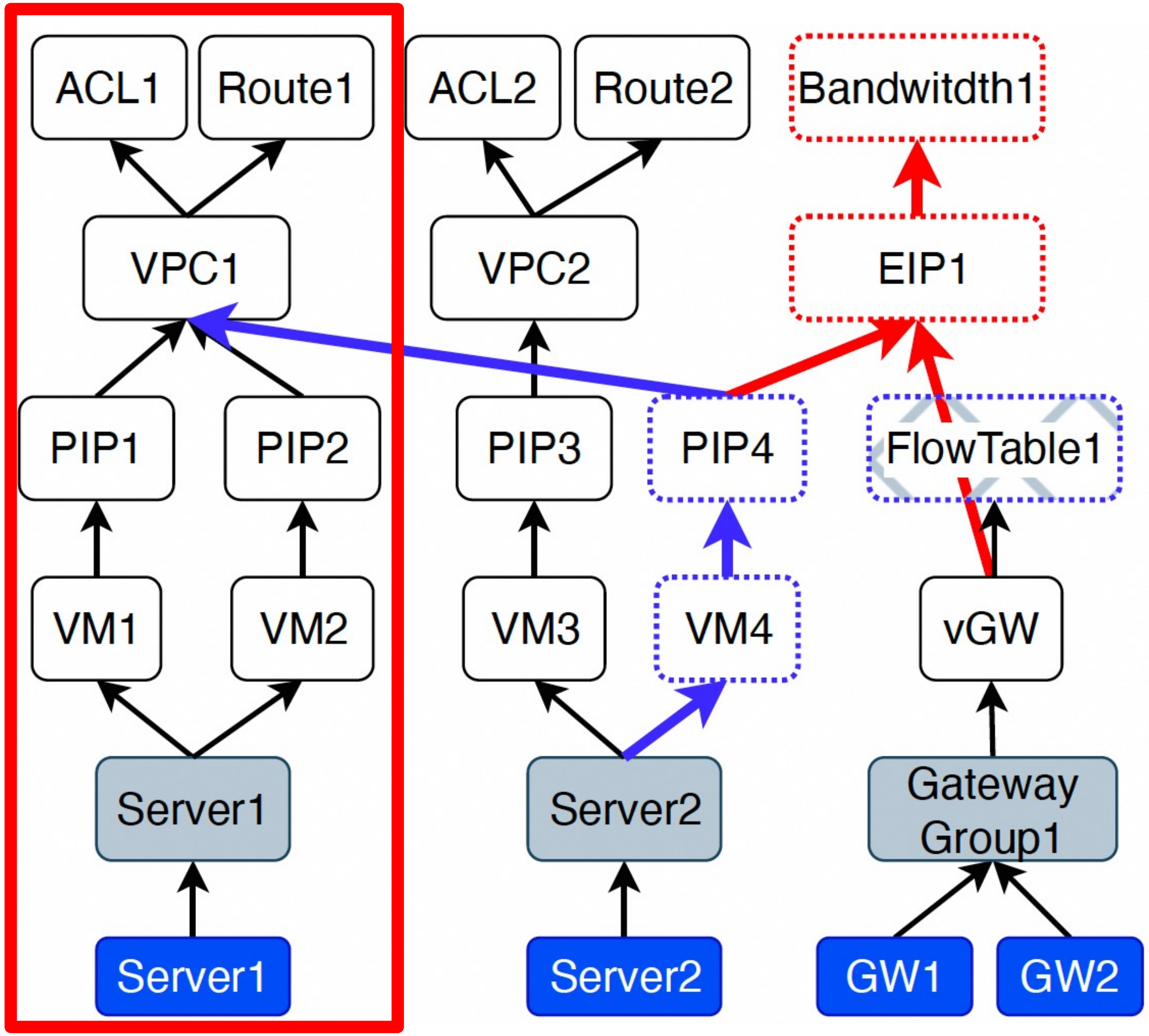
(1) Create VM4, PIP4;
(2) Relate VM4 to PIP4, PIP4 to VPC1, Server2 to VM4;
(3) Update FlowTable1.

(1) Create EIP1, Bandwidth;
(2) Relate PIP4 to EIP1, EIP1 to Bandwidth, Gateway group1 to EIP1

In the past, each API was implemented using individual codes (SQL+if-else).

With Trident, cloud control APIs is represented by a combination of 5 atomic operations over 3 basic objects

Design#3: Tree-based config changes calculation



Trident Tree

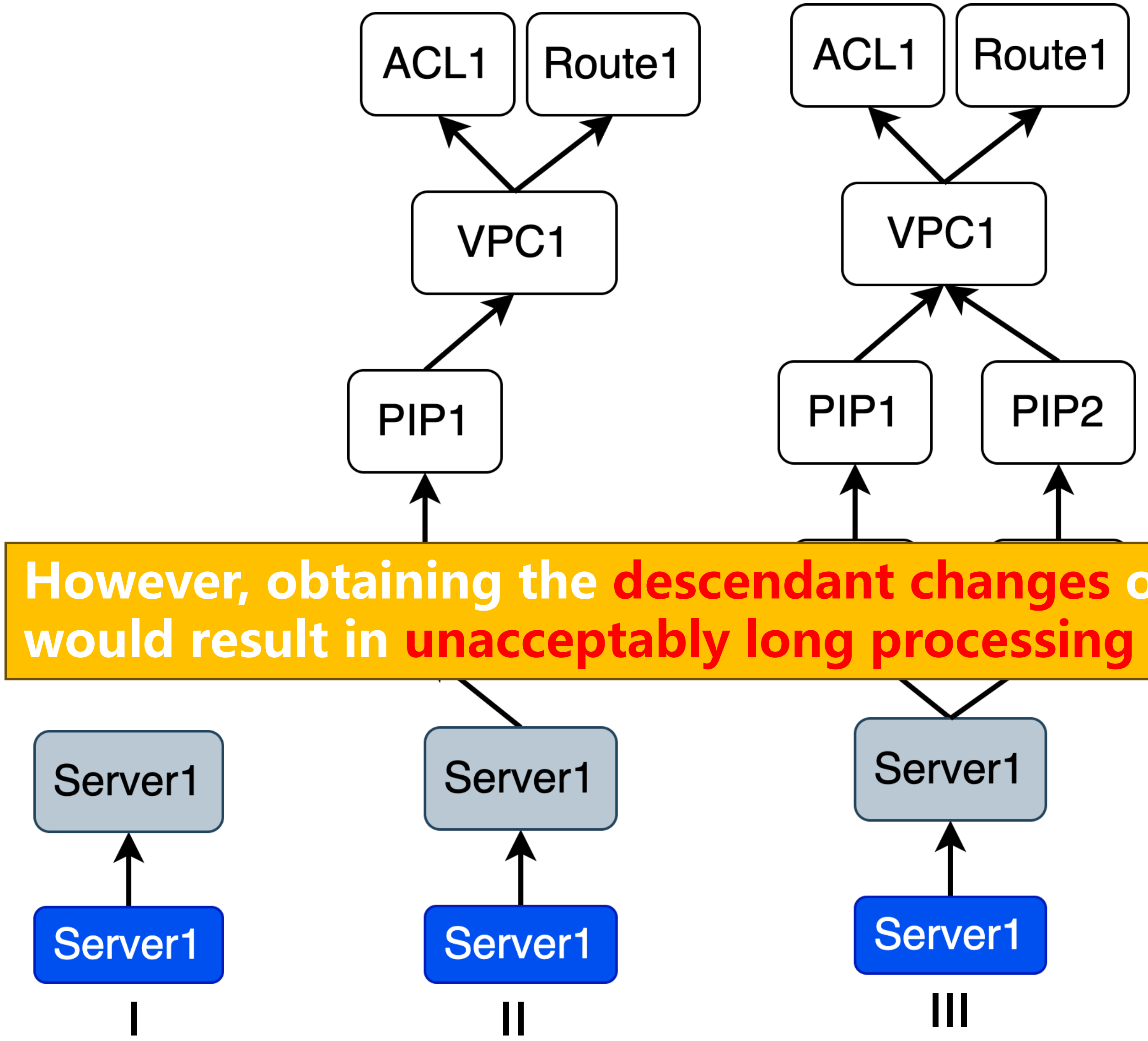
In **Trident Tree**, the **Device' s descendants** imply all its required **Confs**

Configuration updates of Device
=
Difference in Device's descendants **before** and **after** the Trident Tree changes

With Trident operations, multiple "**Device→Group→Conf** " chains are formed.

By caching these chains, we obtain a "Tree",

Design#3: Tree-based config changes calculation



In **Trident Tree**, the **Device' s descendants** imply all its required **Confs**

Configuration updates of Device
=
Difference in Device's descendants **before** and **after** the Trident Tree changes

	Descendants of	Config changes of
II	VM1, PIP1, VPC1, ACL1, Route1	*Add* VM1, PIP1, VPC1, ACL1, Route1
III	VM1, PIP1, VPC1, ACL1, Route1, VM2, PIP2	*Add* VM2, PIP2

However, obtaining the descendant changes of root nodes through naive tree traversal would result in unacceptably long processing time for a cloud-scale Trident Tree.

Design#3: Tree-based config changes calculation

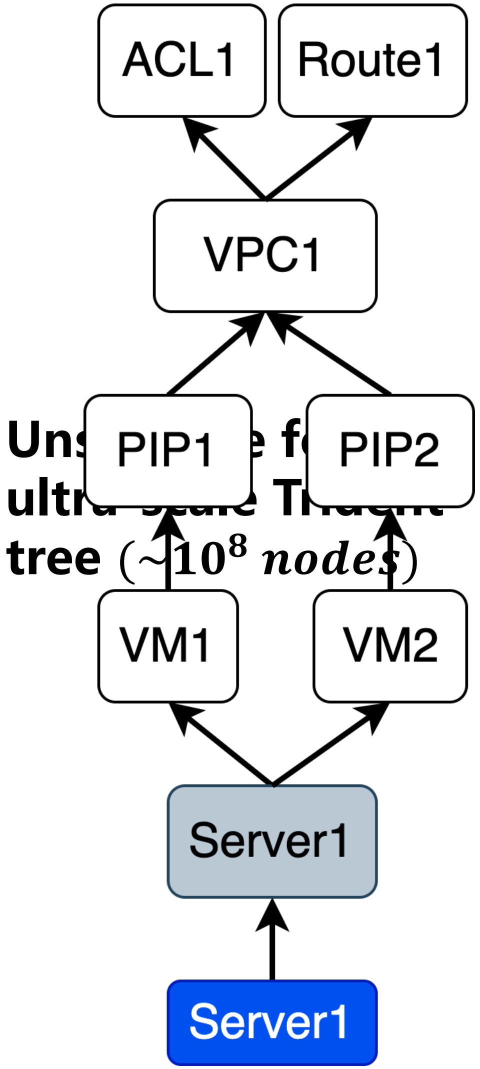
Core Task

Whether A Conf is descendant of a Device?

Naive tree tra l from the root

Descendant relation is transitive from parent to child

Inheriting from all parent nodes



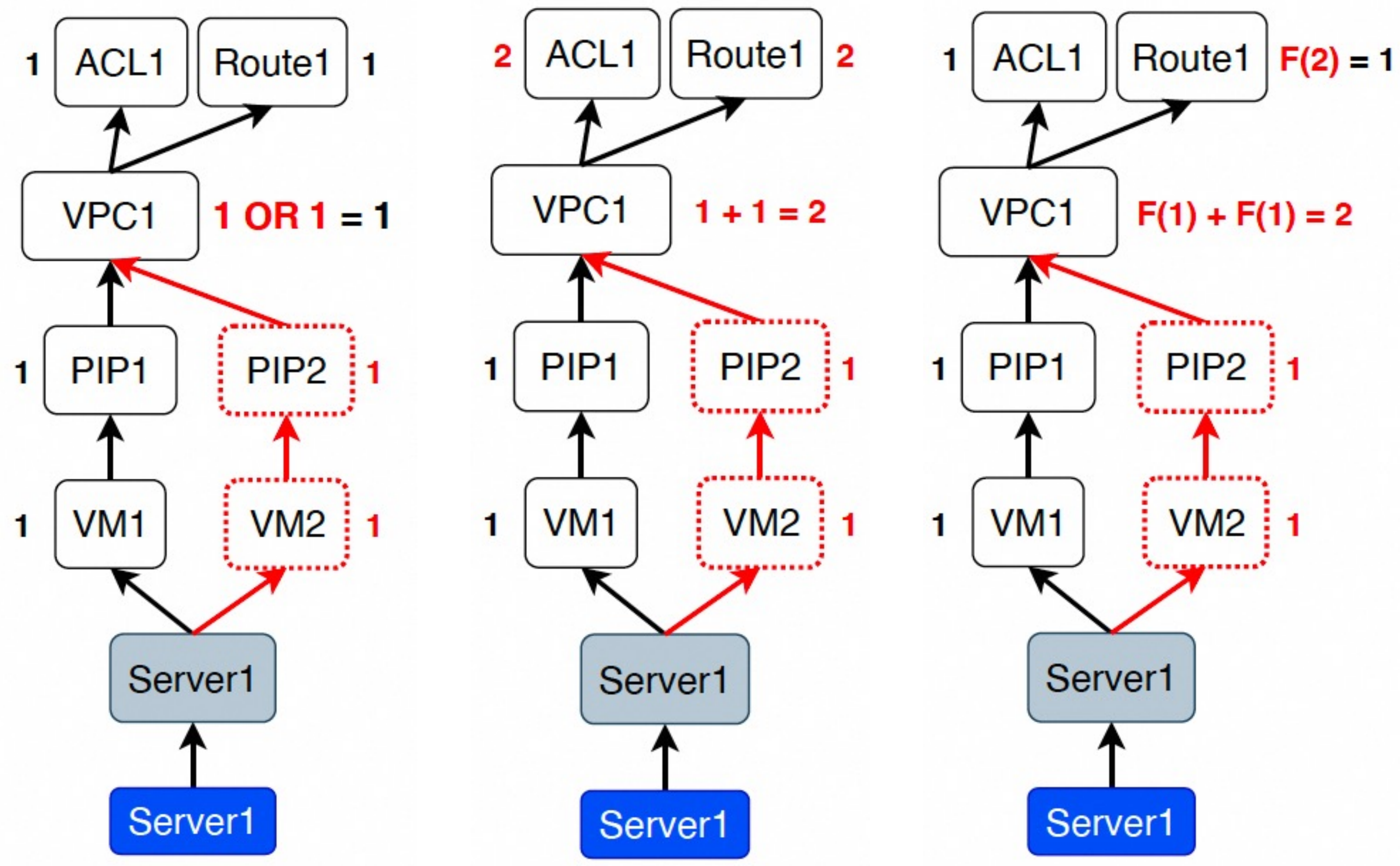
Uns e f ultr e Trident tree ($\sim 10^8$ *nodes*)

If parent Conf is descendant of Device, its children Conf are also the Device’s descendant

The complexity is low because:

- only **a small number** of nodes' relations need to be refreshed
- **no** need to **traverse from roots**

Design#3: Tree-based config changes calculation



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

	Complexity	Wide Tree	Deep Tree
OR-	$O(Parent_i)$	✗	✓
SUM-	$O(1)$	✓	✗
Function-	$O(1)$	✓	✓

OR-based: $\vee_i \, ref(Group, Parent_i)$

- Non-reversible -> $O(Parent_i)$ -> Non-scalable for **wide** trees

SUM-based: $\sum_i ref(Group, Parent_i)$

- Reversible -> $O(1)$
- Transit to all descendants -> Non-scalable for **deep** trees

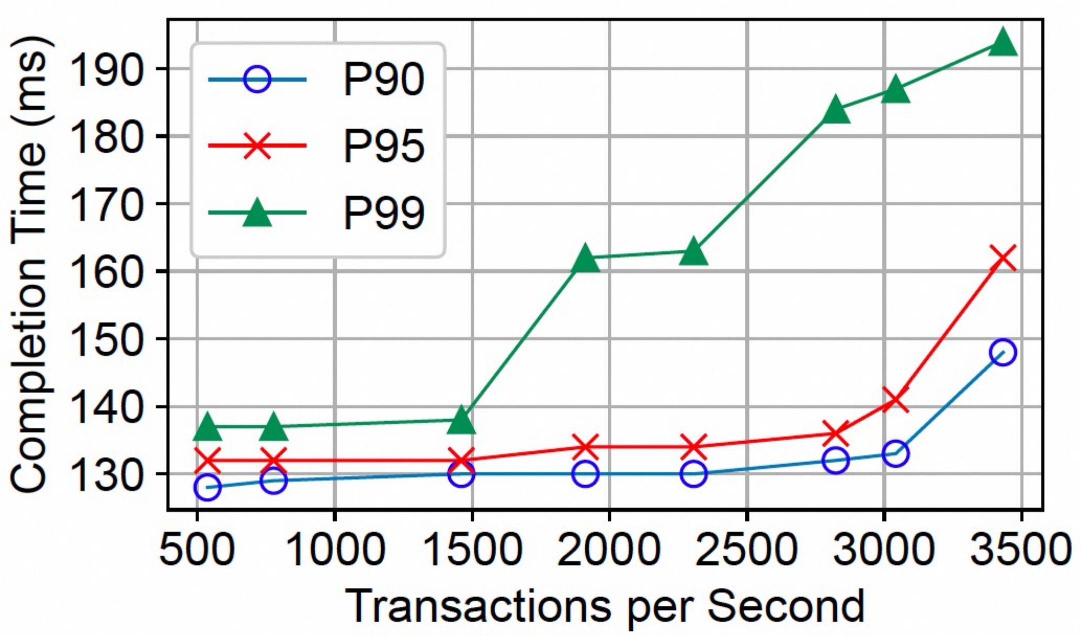
Function-based:

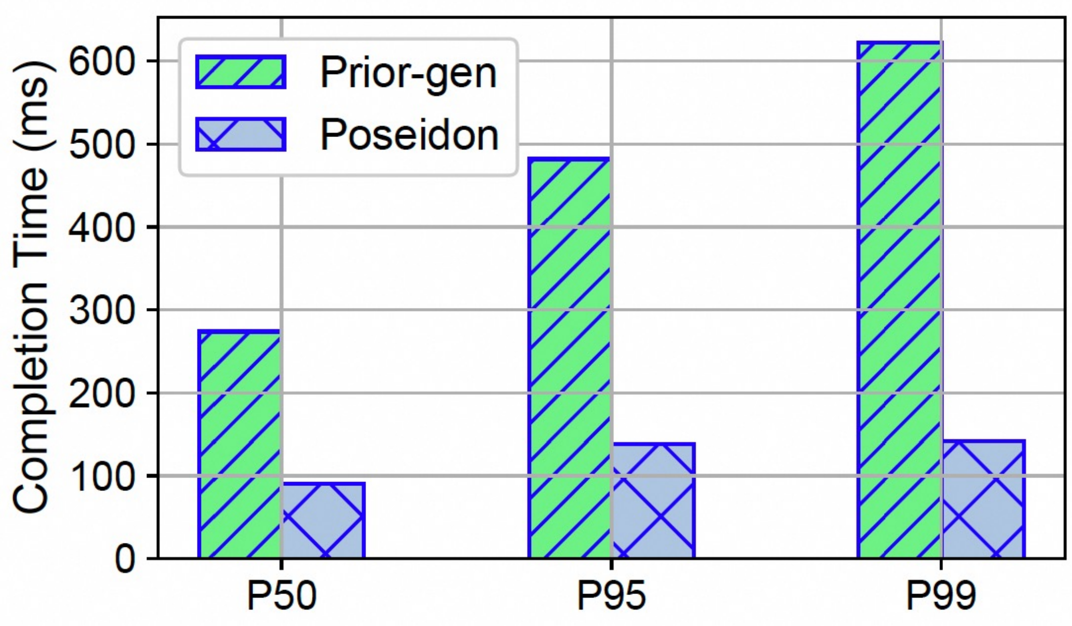
$$\sum_i F(ref(Group, Parent_i))$$

$$F(x) = \begin{cases} 1, & x > 0 \\ 0, & x \le 0 \end{cases}$$

How to design the transitive method of $ref(Server1, Conf_x)$ for cloud?

Evaluation#1: Throughput and P99 completion time improved by 21x and 4.4x

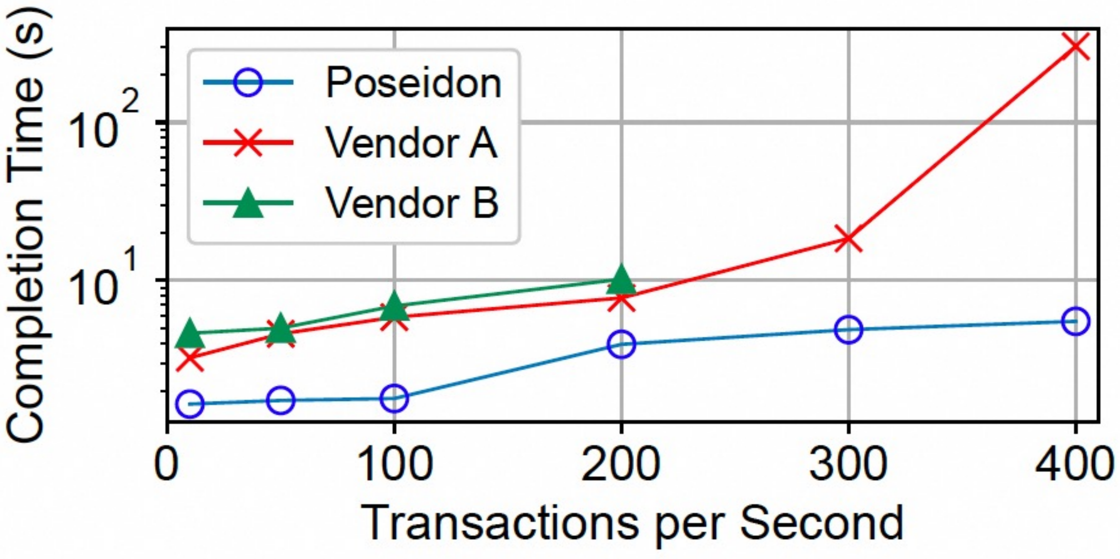


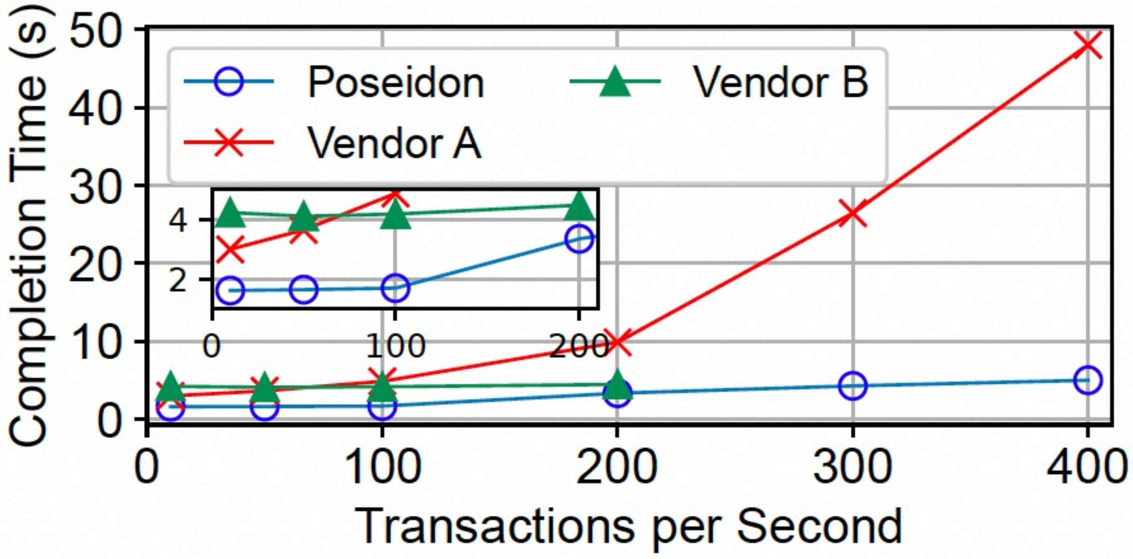


Throughput has increased from 160 TPS to more than 3400 TPS (21x).

P50, P95 and P99 completion time improvement: 3x, 3.5x and 4.4x.

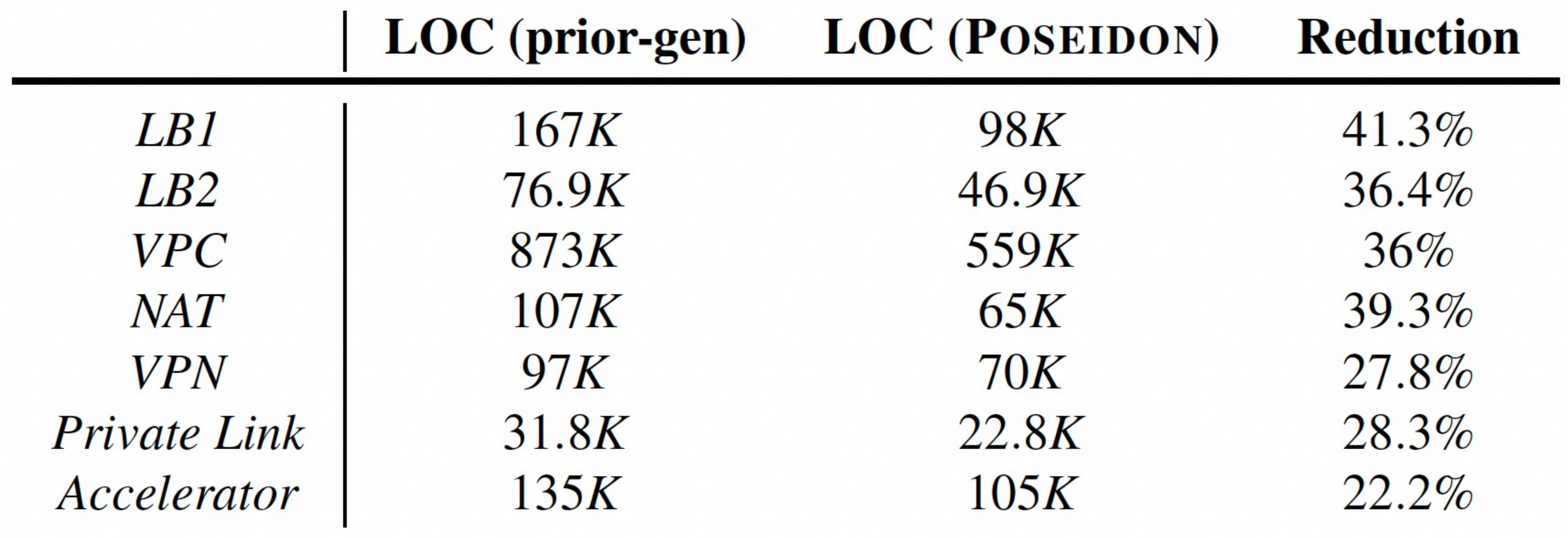
Evaluation#2: Outperforms other Top5 vendors in concurrent APIs processing (1.8x~55x faster)





The completion time of Vendor A and Vendor B is 1.8x~55x and 2.6x~4.8x higher than that of Poseidon.

Evaluation#3: Reduces 22%~41% (LOC)



	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%

The reduction in OpEx and development cost has not been added to Poseidon, as LOC of the Poseidon is only around 150K, which is much lower than the total LOC reduction.

Experiences

□ How to migrate to Poseidon? = Changing the engine while the plane is flying

□ Poseidon’s performance in extreme situations (e.g., extensive route fluctuations)

□ Where to record the descendant relation between Conf and Group?

□ How to detect Redis failover and recover the data for Poseidon?

□ How to choose pushing and pulling when configuring devices?

□ How to deploy Poseidon to a small-scale virtual network?

Summary

- We demonstrated the challenges and issues faced by virtual network management of large-scale cloud provider, especially in the era of cloud-native computing.
- To save the OpEx of managing numerous controllers without sacrificing flexibility of services iterations, we propose partial consolidation architecture, service- and device-independent abstraction, and tree-based config changes calculation algorithm.
- To improve IO performance (the bottleneck of old controller), we proposed hierarchical storage structure that utilizes Redis, memory, and database simultaneously.
- 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. With Poseidon, our virtual network management performance greatly surpasses that of other major cloud vendors.

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

Q & A