

오픈스택

목차

쿠버네티스 101

목차

- 소개
- 랩
- 설치 전 준비
 - HAPROXY(250)
 - DNSMASQ
 - BOOTSTRAP(10)
- 컨트롤러(10)
- 컴퓨트(40)
- 기능 확장

목차

- 랜처
 - 랜처 CLI
 - 랜처 대시보드
- 랜처 아키텍처
- 고수준 컨테이너
- 기본 명령어
- 기능확장 및 추가
- 확장 명령어
- CI/CD

소개

과정

담당자 및 강사

소개

01

다만 교육 진행 시, 다음과 같은 조건이 필요하다.

Docker 혹은 Podman과 같은 고수준 컨테이너 경험
OCI 표준도구 Buildah, Podman, Skopeo와 같은
도구 사용 경험이 있는 사용자
리눅스 커널 / 네트워크 / 스토리지 및 관리도구 사용
경험이 있는 사용자

02

기본적인 쿠버네티스 경험이 있는 사용자.

추가과정

수세 리눅스 어드민 과정

OpenSuSE부터 SuSE Enterprise Linux 기본 명령어부터 네트워크/파일 시스템 학습.

수세/레드햇 마이그레이션 과정

레드햇 계열 리눅스에서 수세 리눅스로 마이그레이션 시, 고려해야 될 사항

수세 리눅스 Pacemaker과정

수세 리눅스 기반으로 High Availability 구현 및 관리 하는 방법 학습. 레드햇 계열과 어떠한 차이점이 있는지, 그리고 마이그레이션 방법에 대해서 학습.

추가과정

Slat 자동화 과정

수세에서 인수 및 배포하는 Slat 자동화 도구. 기본적인 문법 및 활용방법 학습. 이를 기반으로 시스템 자동화와 rke2 및 rancher 설치를 간단하게 자동화 한다.

테크톤 CI/CD과정

오픈소스 표준 CD 인터페이스. RKE2에서 어떠한 방식으로 테크톤을 설치 및 파이프라인 구성 하는지 학습한다. 여기에는 기본 문법 및 Tekton HUB 사용법 까지 포함되어 있다.

강의일정

강의 일정은 아래와 같음.

DAY 1:

RKE2 및 Rancher 설치

DAY 2:

쿠버네티스 자원 활용 및 구성

DAY 3:

쿠버네티스 자원 활용 및 테크톤 기반 CD환경 구성

수업시간

- 수업 시간은 45분 수업
- 쉬는 시간은 10분

점심 시간은 최대한 사람이 많은 시간을 피하기 위해서 **11시 30분부터 12시 50분**

오픈스택 랩 정보

"<https://vlab.dustbox.kr>"으로 접근 가능. TLS키는 SelfSign이기 때문에 보안 문제는 무시 후 진행.

접근 시 사용 가능한 계정은 다음과 같음.

사용자 계정: user1~20

비밀번호: vlabhorizon

도메인: vlab-users

강의 기간 이외에 더 오랫동안 사용을 원하는 경우, 강사에게 요청 해주시면 됩니다. :)

강사

강사

강사

이름: 최국현

메일: tang@linux.com, 회신은 다른 메일 주소로 드리고 있습니다. :)

사이트: tang.dustbox.kr

언제든지 질문 및 요청 환영입니다.

오픈스택

전체적인 설명

역사

오픈스택은 본래 미국 NASA에서 프로젝트용 웹 사이트 관리를 효율적으로 하기 위해서 Rackspace에 관리를 맡겼으며, Rackspace는 여러 프로젝트 및 자원의 회수 및 재활용을 효율적으로 하기 위해서 리눅스 기반으로 관리도구를 만들기 시작하였으며, 이 도구가 점점 발전이 되면서 최종적으로 "오픈스택"이라는 이름으로 커뮤니티에 기여 및 공개가 되었다.

자세한 내용은 아래 사이트에서 확인이 가능하다.

<https://docs.openstack.org/project-team-guide/introduction.html>

주요 구성원

오픈스택은 파이썬 기반으로 작성이 되어 있으며, 이때 오픈스택은 Nova밖에 없었다. 이를 통해서 네트워크/볼륨/이미지/가상머신/보안과 같은 자원을 모두 다루다가 하나씩 서비스가 분리가 되었다.

현재 오픈스택 핵심 서비스 다음과 같다.

1. Nova
2. Keystone
3. Cinder
4. Glance
5. Heat
6. Horizon

용도

오픈스택은 퍼블릭 클라우드에서 대표적인 **AWS/GCP**와 같은 API기반의 플랫폼 구성 및 운영이 가능하다. 서비스 확장은 사용자 혹은 오픈스택 릴리즈에 맞추어서 가능하며, 이를 통해서 다양한 서비스를 사용자에게 제공이 가능하다.

오픈스택은 플랫폼 기반으로 서비스가 확장이 되는 구성을 가지고 있다. 이를 통해서 **IaaS** 및 **XaaS** 서비스 개념을 구현 및 구성하고 있다. 또한, 오픈스택은 **OpenInfra**의 핵심 구성원이다. 오픈 소스에서는 인프라 및 응용 분야에 대해서 보통 두 가지 영역으로 구별하고 있다.

1. OpenInfra
2. CNCF

OpenInfra는 말 그대로 열린 인프라 아키텍처를 지향하고 있으며, **CNCF**는 인프라 위에서 동작하는 컴퓨팅 서비스에 대해서 정의하고 있다.

사용방법

오픈스택은 현재 총 3가지 방식으로 제공하고 있다.

1. Openstack-CLI
2. Openstack Horizon Dashboard
3. Openstack Skyline Dashboard

설치 방법

현재 오픈스택은 다양한 방식으로 패키지 및 설치 방식을 제공하고 있다. 기본적으로 패키지는 아래와 같은 형식으로 제공한다.

1. deb

2. rpm

시스템 패키지가 아닌, 컨테이너 기반으로 구성을 원하는 경우, OpenStack Kolla Ansible를 통해서 설치가 가능하다. 이는 컨테이너 이미지를 통해서 런타임을 제공하며, 다음과 같은 형식의 배포판을 지원하고 있다.

1. Debian 계열

2. Redhat 계열

아쉽게도 수세/슬랙웨어 리눅스 계열에 대한 패키지 및 런타임 이미지는 아직 제공하지 않는다.

설치 방법

베어메탈 기반으로 구성 시, 자동화 도구가 필요한데 오픈스택은 다음과 형식으로 제공한다.

1. `Ironi`
2. `Bifstor (ansible + Ironi)`
3. `Kayobe`

레드햇 오픈스택 경우에는 현재는 `Ironi` 기반에 `puppet/ansible` 기반으로 복합적으로 구성되어 있음.

대용량 및 프로덕트 용도로 사용을 원하는 경우 `Kayobe` 기반으로 물리적 인프라 구성 및 컨테이너화 된 오픈스택 서비스 배포 사용을 권장한다.

오픈스택 기본 컴포넌트 소개

keystone/nova/neutron/cinder/heat/glance/ceilometer

Oslo, RabbitMQ

기본 기능

Oslo란 무엇인가?

Oslo – OpenStack 공통 유틸리티 라이브러리

- OpenStack 서비스 간 중복 기능을 표준화한 Python 라이브러리 모음
- 코드 재사용, 유지보수성, 일관성 확보 목적

주요 모듈

- oslo.config: 설정 파일 파싱
- oslo.db: DB 연결 및 마이그레이션
- oslo.log: 로깅 통합
- oslo.messaging: 메시징 추상화 (→ RabbitMQ 연동)
- oslo.concurrency, oslo.serialization 등

RabbitMQ란 무엇인가?

RabbitMQ – 메시지 브로커

- AMQP 기반 비동기 메시징 시스템
- OpenStack 서비스 간 RPC 호출, 알림 처리 담당

주요 구성 요소

- Producer/Consumer
- Exchange → Queue → 메시지 전달
- 서비스 간 메시지의 중간자 역할

사용 예시

Nova API → 메시지 전송 → RabbitMQ → Nova Compute

Oslo.messaging + RabbitMQ

Oslo와 RabbitMQ의 관계

- oslo.messaging이 메시지 송수신 로직을 추상화
- 실제 메시지 전송은 RabbitMQ가 수행

메시지 흐름

서비스 A → oslo.messaging → RabbitMQ → 서비스 B

1. Oslo는 인터페이스
2. RabbitMQ는 운송 수단

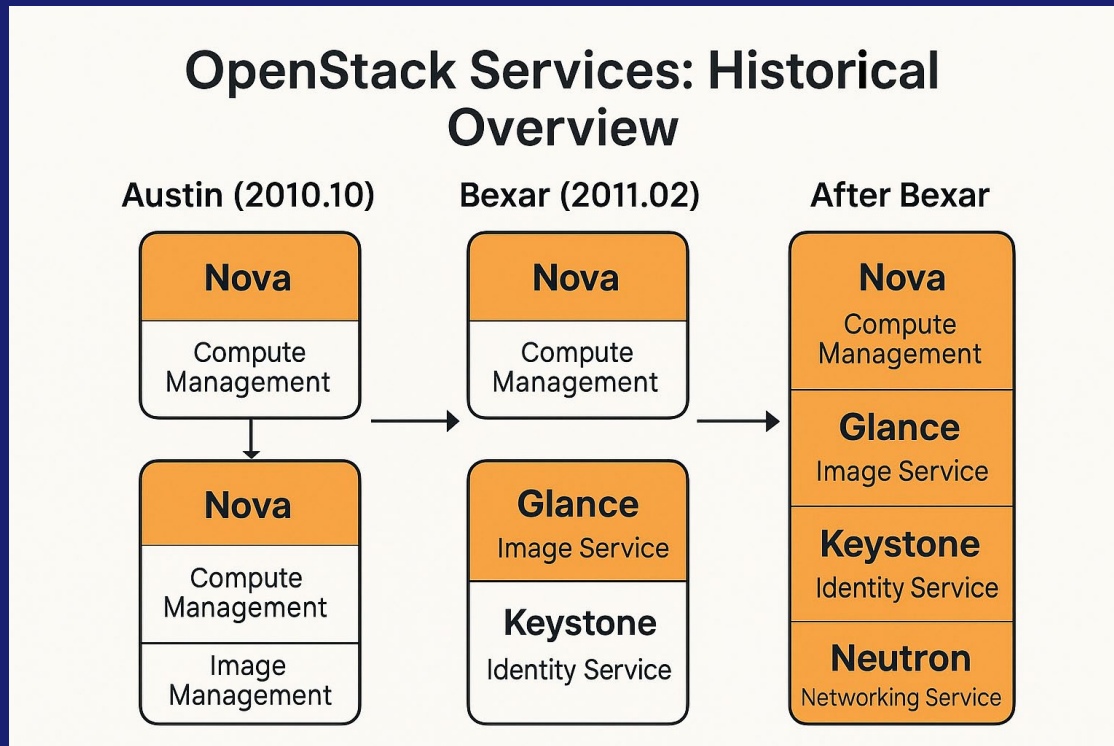
요약 및 비교

Oslo = 뼈대 / RabbitMQ = 혈관

항목	Oslo	RabbitMQ
역할	OpenStack 공통 라이브러리 (설정, 메시징 등)	메시지 브로커 (서비스 간 비동기 통신 중계)
기능	설정 관리, DB 연결, 메시징 인터페이스, 로그 등	큐잉, 라우팅, 메시지 저장 및 전달
구성	Python 패키지 모듈 (예: oslo.config, oslo.messaging 등)	AMQP 기반 구성 (Exchange, Queue, Producer 등)
주요 예시	oslo.messaging → 메시지 큐 인터페이스 제공	Nova API → 메시지 전송 → Nova Compute 처리
연관성	oslo.messaging이 메시지를 추상화하고 RabbitMQ가 처리함	실제 메시지 운송 및 수신자 전달 담당
비유	OpenStack의 뼈대 역할 (공통 구조)	OpenStack의 혈관 역할 (데이터 흐름)

코어 서비스 개요

오픈스택은 본래 가상머신 관리를 위한 도구로 나왔다. 하지만, 이 기능이 NASA의 많은 부서에서 사용하면서 현재의 IaaS 형태로 개발이 되기 시작했다.



코어 서비스 개요

OpenStack의 기원은 2010년, NASA와 Rackspace의 협업에서 시작이 되었음. 이 시점에는 오픈스택에 Oslo, RabbitMQ와 같은 기능은 포함이 되지 않았음.

1. NASA는 "Nebula"라는 내부 클라우드 프로젝트를 통해 VM 관리를 자동화 목적
2. Rackspace는 자체적인 오브젝트 스토리지 시스템(Swift)을 가지고 있었고, 이를 공개하면서 협업 시작

초기 목적은 "가상머신 인프라 자동화"였고, 이후 커뮤니티 확장을 거치며 IaaS 전체 영역(컴퓨트, 네트워크, 스토리지 포함)으로 발전.

NASA는 초기 버전의 Nova(컴퓨트 서비스)를 개발하면서, "VM 인스턴스의 프로비저닝, 스케줄링, 라이프사이클 관리"에 집중.

Keystone

기본 기능

Keystone 개요

Keystone: OpenStack의 인증 및 권한 서비스

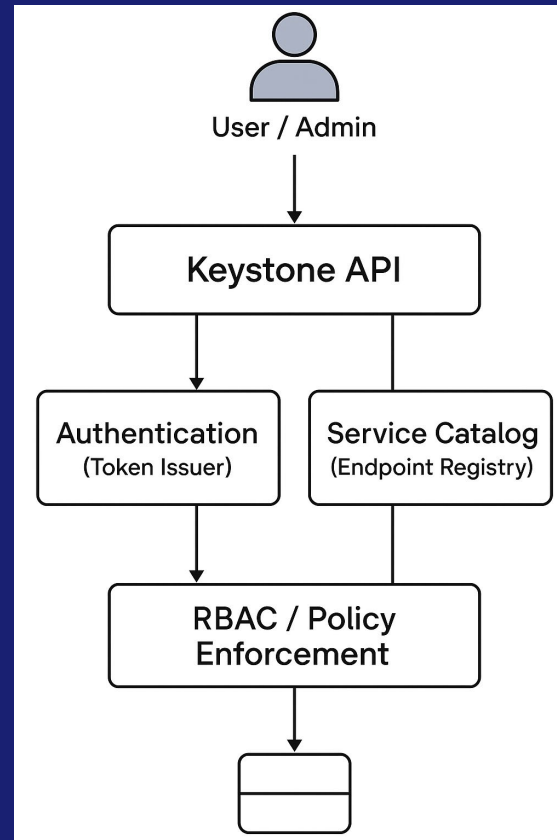
1. OpenStack의 Identity Service 역할
2. 인증(Authentication) + 권한(Authorization)
3. 사용자, 프로젝트, 역할, 서비스, 엔드포인트 관리
4. 멀티 도메인 및 역할 기반 접근 제어 지원 (RBAC)

Keystone 주요 기능 정리

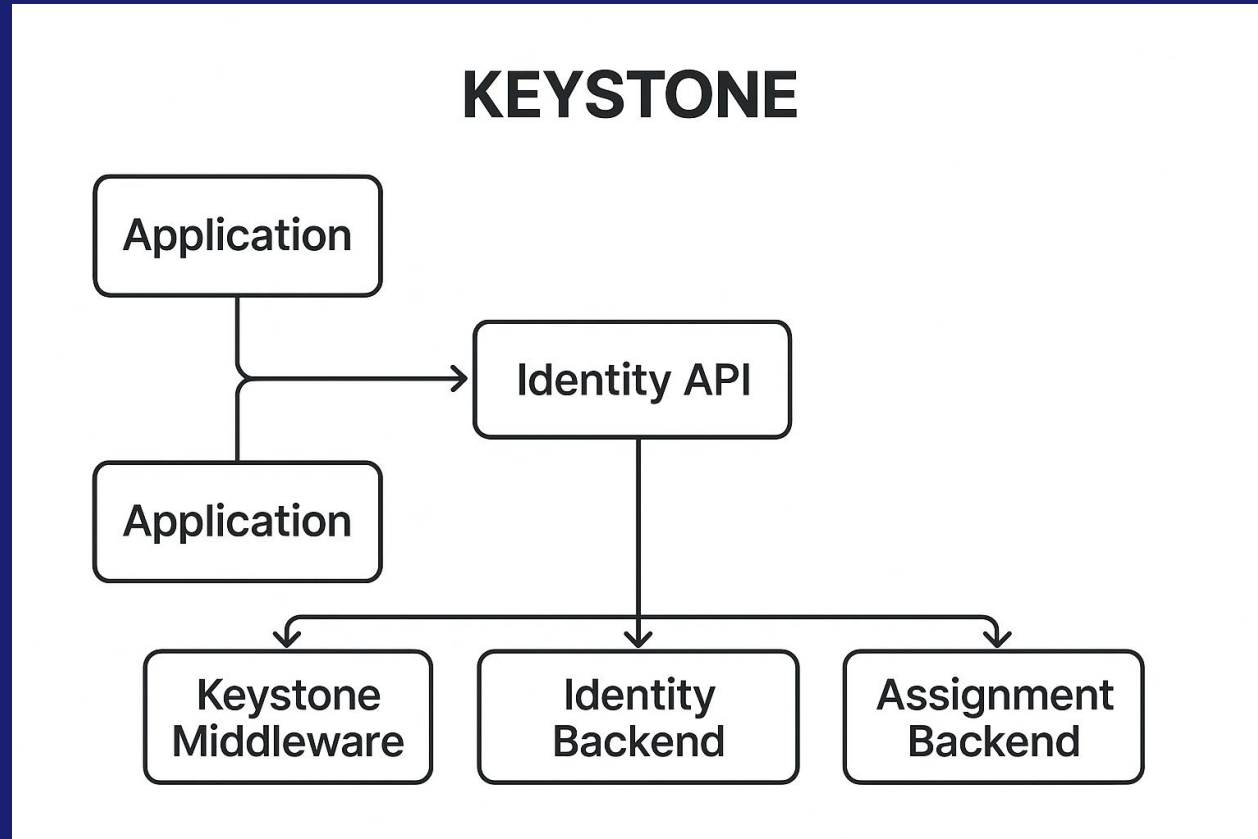
기능을 표로 정리하면 다음과 같다. 자세한 설명은 뒤에서 좀 더 다룬다.

기능 영역	설명
인증(Authentication)	사용자 인증 후 토큰 발급
권한(Authorization)	역할 기반 권한 부여 (RBAC)
서비스 카탈로그	API 서비스 목록 및 위치 정보 제공
정책 엔진	policy.yaml 기반 권한 규칙
멀티도메인 지원	도메인별 사용자/프로젝트 분리 관리
외부 연동	LDAP, SAML, OAuth, Kerberos 등과 통합 가능

Keystone 계층



Keystone 백엔드



Keystone 인증 흐름 요약

1. 사용자가 Keystone에 사용자명/비밀번호로 로그인 요청
2. Keystone이 사용자 인증 후 토큰 발급
3. 사용자는 이 토큰을 가지고 다른 OpenStack 서비스에 접근
4. 각 서비스는 Keystone에 토큰 유효성 확인 요청
5. Keystone이 유효한 토큰임을 확인하고 응답

Keystone

Keystone은 OpenStack의 인증(Identity) 서비스를 담당하며, 사용자 인증, 권한 관리, 서비스 등록 및 서비스 접근 위치(Service Endpoint) 정보 제공 등을 수행한다.

Keystone은 다음과 같은 주요 기능을 제공한다:

1. 사용자(User), 프로젝트(Project), 역할(Role)의 생성 및 권한 할당

- 이를 통해 OpenStack 자원에 대한 접근 제어를 수행할 수 있다.

2. 인증 토큰 발급 및 검증

- 사용자가 OpenStack API에 접근하기 위해서는 Keystone으로부터 토큰을 발급받아야 하며, 이 토큰은 요청 시 인증 수단으로 사용된다.

Keystone

3. 서비스 카탈로그(Service Catalog) 제공

- 등록된 OpenStack 서비스들의 목록과 해당 서비스의 접근 엔드포인트(Endpoint) 정보를 포함하며, 토큰 발급 시 함께 반환된다.
- 엔드포인트는 보통 public (외부 접근용), internal (내부 서비스간 호출용), admin (관리자용) 세 가지 URL로 구성된다.

4. 서비스 등록 및 서비스 계정(Service Account) 관리

- OpenStack의 각 서비스(Nova, Glance, Neutron 등)는 Keystone에 등록되어야 하며, 이때 해당 서비스에 접근할 수 있는 전용 사용자 계정(서비스 계정)이 생성된다.
- 서비스 계정은 일반 사용자와는 별도로 서비스 간 통신 및 인증을 위해 사용된다.

Keystone

도메인(Domain) 개념

Keystone은 사용자 및 프로젝트를 도메인 단위로 관리할 수 있으며, 멀티 도메인 환경에서 권한과 자원 관리를 유연하게 할 수 있다.

정책 관리 (Policy)

Keystone은 서비스 전반의 접근 정책을 policy.json 혹은 policy.yaml을 통해 정의하여, 역할 기반 접근 제어 (RBAC)를 적용할 수 있다.

Federation & 외부 인증

LDAP, SAML, OIDC 등 외부 인증 시스템과의 연동이 가능하며, 이를 통해 싱글사인온(SSO) 환경도 구성할 수 있다.

Glance

기본 기능

설명

Glance는 OpenStack에서 이미지 관리 서비스를 담당하며, 가상 머신 이미지의 등록, 저장, 검색 및 전송 기능을 제공한다.

기본적으로 Glance는 파일 시스템을 백엔드로 사용하며, 이는 `/var/lib/glance/images/` 디렉터리에 UUID 형식의 파일로 저장된다.

다른 저장 백엔드(예: Swift, Ceph RBD, S3 등)를 사용하는 경우, 해당 스토리지 방식에 맞게 이미지가 저장된다.

Glance는 본래부터 Nova와 별개의 서비스로 존재하며, Nova는 Glance API를 통해 이미지를 불러와 인스턴스를 부트한다.

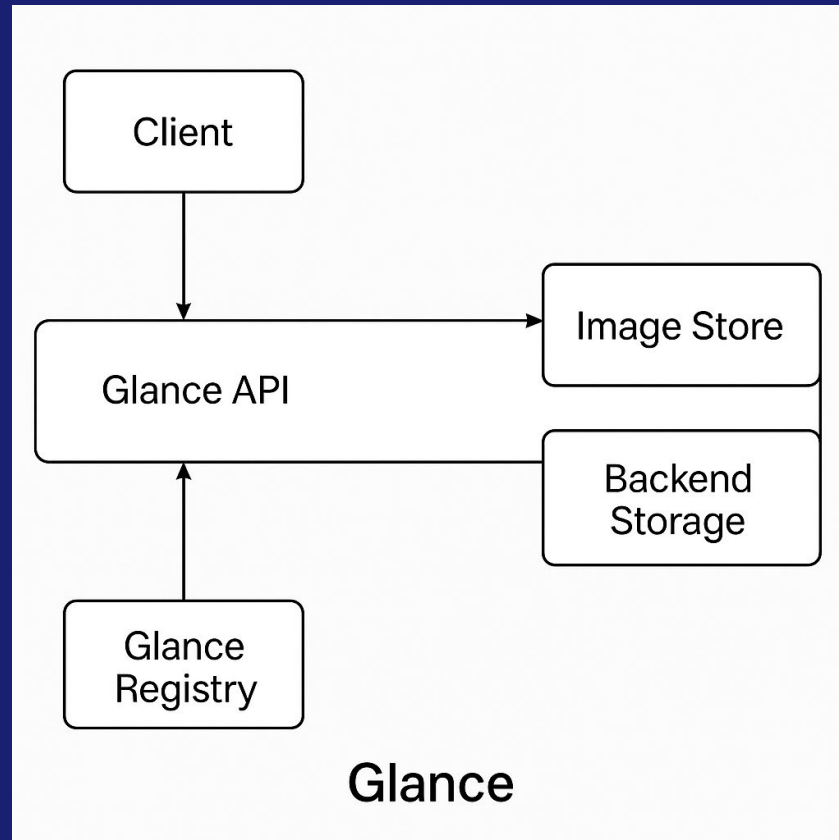
모든 이미지는 `disk_format`, `container_format` 등을 포함한 메타데이터를 가지며, 이 정보는 이미지의 저장 방식과 부팅 가능 여부를 결정한다.

설명

Glance는 구조적으로 단일 백엔드만 선택적으로 사용하는 방식을 기본으로 합니다.

1. **데이터 일관성 유지:** 여러 백엔드에 이미지를 분산 저장하면 동기화 및 정합성 관리가 복잡해짐.
2. **단순한 관리:** 운영자가 스토리지 위치나 방식에 대해 명확히 알고 있어야 하기 때문에, 단일 백엔드가 운영 및 마이그레이션에 유리함.
3. **디자인 철학:** 초기 Glance 설계는 “하나의 이미지 = 하나의 저장소”라는 단일 경로 접근을 따름.
4. **멀티 백엔드 기능은 존재하지만 기본은 아님.**
 - multi_store 기능이 존재하여 여러 백엔드를 선언할 수 있지만, 운영자는 각 이미지를 특정 스토어에 직접 지정해야 하며, 기본적으로 자동 분산은 불가능함.

Glance



Glance 기능 분리

Glance가 Nova에서 다음과 같이 릴리즈 되면서 분리가 되었다.

릴리즈	이미지 관리 방식
Austin	Nova 내부 기능 (이미지 관리 포함)
Bexar	Glance 등장! Nova와 분리된 이미지 서비스
이후	Glance가 Nova와 분리된 채로 표준 이미지 API 제공

Glance가 지원하는 주요 백엔드

Glance에서 지원하는 백엔드는 다음과 같다.

항목	내용
지원 백엔드	file, http, swift, rbd, cinder(기타: s3, gridfs, sheepdog 등 deprecated)
기본 백엔드	file (경로: /var/lib/glance/images)
설정 위치	glance-api.conf의 default_store 항목
멀티 백엔드 지원	enabled_backends 및 stores 설정으로 가능(예: default_store = fast-ceph, enabled_backends = fast-ceph:rbd, backup-swift:swift)
멀티 백엔드 제한 이유	<ul style="list-style-type: none">▪ 이미지 정합성 및 동기화 문제 발생 가능성▪ 관리 복잡성 증가▪ 설계 철학: "1 이미지 = 1 저장소"▪ 자동 분산 불가, 직접 저장소 지정 필요

Glance가 지원하는 이미지 유형

Glance에서 다음과 같은 이미지들을 지원 및 제공한다.

이미지 유형	설명	주요 사용처
QCOW2	QEMU Copy-on-Write. 스냅샷, 압축, 공간 절약 기능 지원	KVM 기반 가상머신 (주로 사용됨)
Raw	비압축 순수 이미지. 가장 단순함	성능 우선 환경, Ceph RBD 등
VHD/VHDX	Microsoft Hyper-V 이미지 형식	이종 플랫폼 간 이전 목적
VMDK	VMware의 가상 디스크 포맷	VMware에서 생성된 이미지 호환
ISO	부팅 가능한 설치 이미지	OS 설치용
AMI/AKI/ARI	EC2 호환 이미지 포맷	클라우드 호환 환경
Docker (container)	OCI 기반 컨테이너 이미지 지원 (Glance + Glare 확장 필요 시)	컨테이너 오케스트레이션 연동 목적

VMDK가 권장되지 않는 이유

VMDK사용은 가능하나, 올바르게 기능 및 성능을 기대할 수 없다.

이유	설명
변환 필요	KVM, QEMU 등에서 직접 사용 불가. QCOW2나 RAW로 변환해야 함
호환성 문제	VMDK는 VMware 전용 포맷으로, OpenStack에서의 기능 (스냅샷, resize 등)과 충돌 우려
스토리지 최적화 미지원	Ceph, Cinder 등 백엔드 스토리지에서 비효율적
보안 위험	VMDK 내부에 포함된 메타데이터 구조가 표준화되지 않아 보안 취약점 가능성 존재
운영 복잡성	여러 하이퍼바이저 포맷 지원 시 관리 부담 증가

QCOW2/3 vs RAW: 성능 비교

RAW 이미지 형식은 압축이나 메타데이터 구조 없이 순수한 디스크 블록만을 포함하는 가장 단순한 형식. 반면, QCOW2/3는 스냅샷, 압축, 암호화, 백링크 기능을 포함하는 고급 기능 지원 이미지 포맷.

1. 성능 (I/O throughput 및 latency)

- RAW 형식은 오버헤드가 전혀 없어, 디스크 I/O 성능 측면에서 가장 빠름. 특히, Ceph RBD나 직접 블록 디바이스에 연결된 스토리지에서 그 차이는 더 두드러짐.
- QCOW2/3는 내부 메타데이터 구조로 인해 디스크 접근 시마다 주소 변환, 압축 해제, 스냅샷 관리 작업이 동반되어 쓰기 성능이 저하. 특히 랜덤 I/O 환경에서는 latency 증가.

QCOW2/3 vs RAW: 성능 비교

2. 공간 효율

- QCOW2/3는 **sparse allocation**과 **압축** 기능 덕분에 실제 디스크 공간을 절약할 수 있다. 작은 테스트 VM이나 일시적인 이미지에 유리.
- RAW는 항상 **전체 용량**을 미리 할당해야 하기 때문에 디스크를 더 많이 소모.

3. 기능성

- QCOW2/3는 스냅샷, 백링크 이미지 체인 생성 등 가상화 환경에서 필요한 다양한 기능을 제공.
- RAW는 단순하지만 기능이 제한되어 있으며, snapshot은 외부 도구(KVM snapshot, Ceph snapshot 등)에 의존.

무엇이 더 나은 형태인가?

Ceph 기반이거나 혹은 EMC와 같은 SAN 장치를 사용하는 경우, QCOW2보다는 RAW사용하는게 더 효율적이고 더 빠르다. QCOW2는 외부 스토리지가 없거나 혹은 NAS형태로 구성하는 경우, QCOW2로 구성하는 더 빠른 프로 비저닝 속도를 기대할 수 있다.

항목	RAW	QCOW2/3
I/O 성능	★★★★★ (매우 우수)	★★ (메타데이터로 인한 오버헤드)
공간 효율	★ (전체 공간 할당)	★★★★ (sparse + 압축 지원)
스냅샷	외부 의존	자체 지원
VM 부팅 속도	빠름	느림 (특히 첫 부팅 시)

검증

"Ceph 기반이거나 혹은 EMC와 같은 SAN 장치를 사용하는 경우, QCOW2보다는 RAW 사용하는 게 더 효율적이고 더 빠르다."

이유:

1. Ceph RBD, EMC VMAX/VNX, NetApp 등 블록 스토리지 기반 시스템은 RAW 이미지와 직접 매핑 가능
2. QCOW2는 로컬 파일 기반의 레이어드 포맷이기 때문에, 블록 스토리지 위에 다시 파일시스템 레벨이 올라가면 중복 I/O 및 latency가 발생
3. 특히 Ceph의 경우, Glance에서 rbd 백엔드를 사용하는 경우 RAW 이미지는 RBD object로 바로 전송되고, VM에서 그대로 attach되므로 zero-copy 방식에 가까운 효율

검증

"QCOW2는 외부 스토리지가 없거나 혹은 NAS형태로 구성하는 경우, QCOW2로 구성하면 더 빠른 프로비저닝 속도를 기대할 수 있다."

1. NAS(Network Attached Storage) 또는 로컬 디스크에 이미지 저장하는 구조.
 - QCOW2는 sparse allocation 덕분에 이미지를 빠르게 복제(clone)
 - copy-on-write 방식으로 VM 프로비저닝 시간이 단축
2. Glance + Nova + libvirt의 QCOW2 snapshot chain을 활용하면 빠른 VM 생성 가능
3. I/O가 많은 VM이 다수 실행되면 QCOW2는 RAW보다 성능이 저하됨
 - 즉, 빠른 초기 프로비저닝은 QCOW2가 유리하지만, 운영 중 성능은 RAW가 우세

정리

위의 내용을 정리하면 다음과 같다.

환경	권장 이미지 포맷	이유
Ceph / SAN (EMC 등)	RAW	블록 스토리지에 직접 매핑, I/O 효율 및 zero-copy 가능
NAS / 로컬 디스크 기반	QCOW2	빠른 clone, thin provisioning, 적은 초기 공간 사용
성능 위주 장기 운영	RAW	지속적인 랜덤 I/O, 낮은 latency
프로비저닝 속도 위주	QCOW2	빠른 VM 배포 및 초기 실행

Cinder

기본 기능

Cinder 설명

Cinder는 OpenStack에서 블록 스토리지 서비스를 제공하는 컴포넌트로, 가상 머신 인스턴스에 디스크 형태의 볼륨을 생성, 첨부, 관리할 수 있도록 지원한다.

초기에는 nova-volume이라는 Nova의 내부 컴포넌트였지만, 2012년 Folsom 릴리즈부터 Cinder라는 별도 서비스로 분리되었다.

Cinder는 다양한 백엔드 드라이버를 지원하며, Ceph RBD, LVM, NFS, NetApp, iSCSI 등의 스토리지 솔루션과 연동이 가능하다.

Cinder 설명

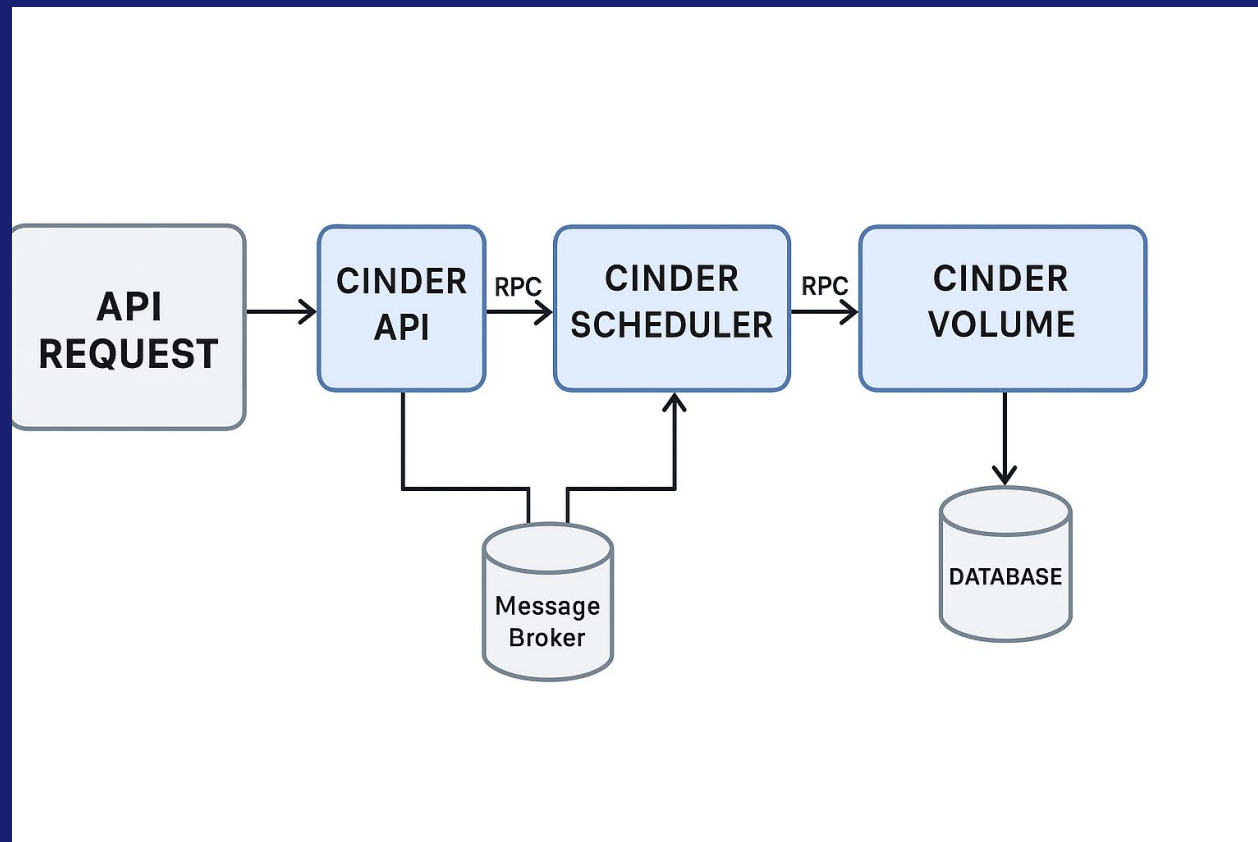
Cinder는 다음과 같은 주요 기능을 제공한다.

1. 볼륨 생성, 삭제, 첨부/분리
2. 스냅샷 및 볼륨 복제
3. 볼륨 백업/복원
4. 이기종 간 스토리지 마이그레이션
5. 공유 볼륨(multi-attach) 기능

Cinder의 주요 기능

1. 인스턴스와 분리된 블록 스토리지
2. 다중 백엔드 지원 (Multi Backend)
3. 스냅샷 기반 복제 및 백업
4. 보안 그룹 및 액세스 제어 연동
5. API를 통한 자동화된 볼륨 관리

Cinder 아키텍처



주요 백엔드 비교

주로 많이 사용하는 백엔드는 다음과 같다.

백엔드	공식 지원	유형	특징
LVM	네	블록	단일 노드, 테스트에 적합
Ceph RBD	네	블록	확장성, 성능 우수, 가장 많이 사용됨
NFS	네	파일	NAS 기반, 간편 구성
GlusterFS	네	파일	분산 파일 기반, 중소규모 적합
NetApp/EMC 등	네	블록/파일	기업용 상용 스토리지 연동

주요 백엔드 비교

선택적으로 다음과 같은 백엔드 사용이 가능하다.

환경	권장 백엔드	이유
대규모, 고성능	Ceph RBD	고가용성, 분산 구조
테스트/개발	LVM	구성 간단
NAS 기반 환경	NFS	외부 NFS 서버 연동
전용 파일 스토리지	GlusterFS	가볍고 유연한 분산 파일
엔터프라이즈 환경	NetApp 등	고성능 SAN/NAS 장비 연동

Nova

기본 기능

Nova 소개

OpenStack의 Nova는 IaaS에서 가장 핵심적인 컴퓨트 서비스로, 가상머신(VM)의 생성을 비롯하여 스케줄링, 삭제, 리사이즈 등 라이프사이클 전반을 관리하는 역할을 수행한다.

Nova는 2010년, NASA와 Rackspace가 협업하여 만든 초기 프로젝트 중 하나로, 코드명은 Nova였다. NASA는 내부적으로 Nebula라는 클라우드 프로젝트를 운영하고 있었으며, Nova는 해당 프로젝트에서 파생된 컴포넌트다. 현재까지도 OpenStack에서 가장 크고 복잡한 프로젝트 중 하나로 자리잡고 있다.

Nova는 다양한 하이퍼바이저(KVM, QEMU, Xen, VMware, Hyper-V 등)를 지원하며, 플러그인 기반 아키텍처를 통해 확장성과 유연성을 제공한다.

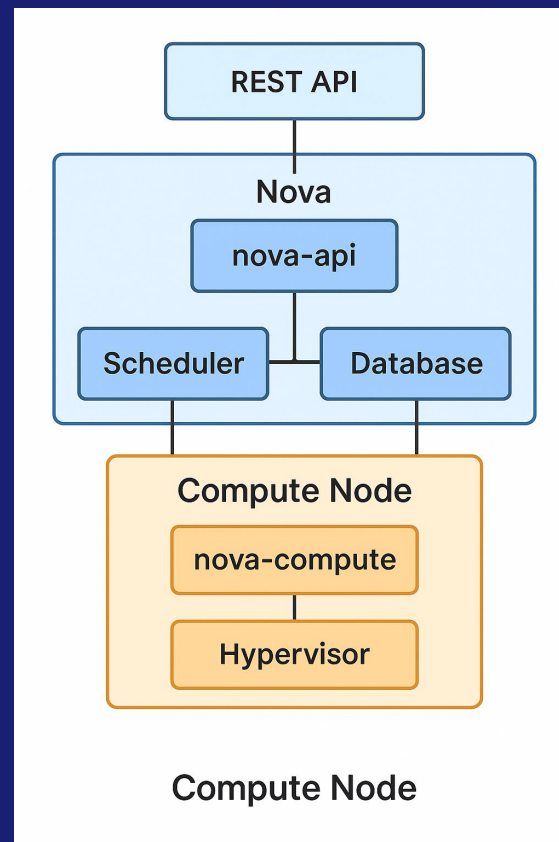
Nova 소개

현재 Nova는 다음과 같은 핵심 컴포넌트로 구성된다.

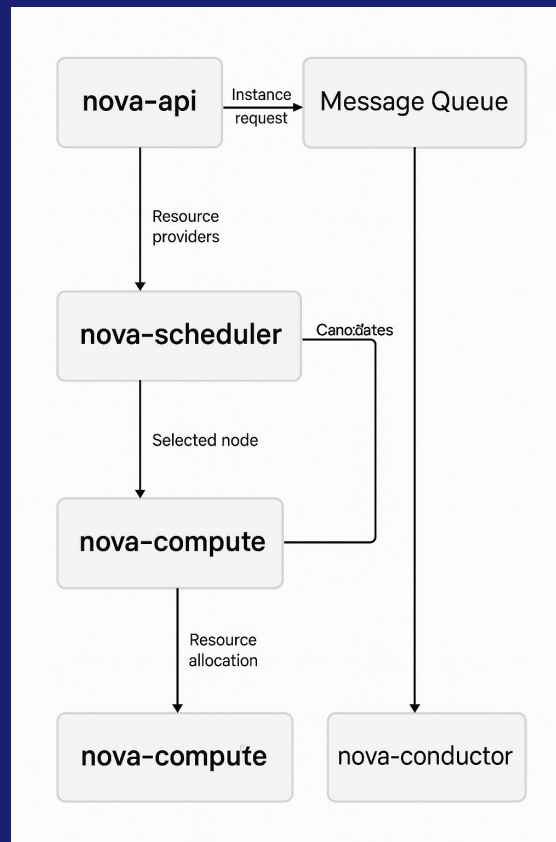
- **nova-api**: 외부에서의 REST API 요청을 수신하고, 인증 및 요청 검증 후 내부 메시지 큐로 전달하는 게이트웨이 역할.
- **nova-conductor**: nova-compute가 직접 데이터베이스에 접근하지 못하도록 RPC를 통해 중재하며, DB에 대한 보안 및 추상화 계층 제공.
- **nova-compute**: 실제 하이퍼바이저(KVM 등)에서 인스턴스를 생성/제거/변경하는 핵심 작업을 담당. 메시지 큐를 통해 명령을 수신한다.

이 외에도 **nova-scheduler**, **nova-novncproxy** 등 다양한 구성 요소가 함께 동작하여 전체 컴퓨팅 서비스를 완성한다.

Nova 아키텍처



Nova 흐름도



Nova 하이퍼바이저 목록

노바에서 제공하는 하이퍼바이저는 다음과 같다.

하이퍼바이저 종류	설명
KVM (기본)	Nova의 기본 설정이며, 대부분의 배포판에서 권장됨
QEMU	KVM 없이도 작동 가능한 하이퍼바이저로, 테스트 및 개발 환경에 사용됨
LXC	리눅스 컨테이너 기반 경량 가상화. 현재는 비활성화 혹은 제한적 지원 (비권장)
Xen	오픈소스 하이퍼바이저로, 도메인 기반 격리 제공. 일부 기업 환경에서 사용됨
VMware vSphere	VMware 환경과의 통합을 통해 ESXi 하이퍼바이저와 연동 가능
Hyper-V	Microsoft Windows 환경용 하이퍼바이저. 일부 윈도우 기반 OpenStack에서 사용됨
PowerVM	IBM Power 시스템에서 사용하는 상용 하이퍼바이저
Virtuozzo	컨테이너 기반 하이퍼바이저. 특수한 목적의 환경에서 사용
Baremetal	하이퍼바이저가 없는 베어메탈 서버 배포. Nova-Ironic 통합으로 지원됨
z/VM	IBM 메인프레임용 하이퍼바이저. 특수 환경에서 제한적으로 사용

Nova 구성원 설명

Nova 컴포넌트는 다음과 같다.

컴포넌트	설명
nova-api	외부 요청 수신 (REST API), 인증 연동 등
nova-conductor	DB 접근 추상화 계층, nova-compute와 DB 간 중재
nova-compute	VM 생성/제거/제어 등 하이퍼바이저 인터페이스
nova-scheduler	어떤 compute 노드에 VM을 배치할지 결정
nova-novncproxy	VNC 프록시 클라이언트
nova-spicehtml5proxy	SPICE 프록시 클라이언트. 현재 라이선스 이유로 레드햇 계열은 RPM에서 제외

요청 수신 및 초기 처리

Nova Scheduler는 인스턴스 생성 요청을 받고, 이를 적절한 compute node에 배치하기 위한 단계는 다음과 같다.

사용자가 인스턴스 생성 요청을 보내면, nova-api를 통해 scheduler까지 전달된다. 여기서 요청에는 인스턴스에 필요한 CPU, 메모리, 디스크 등의 자원 요구사항과 기타 정책(예: 가용 영역, affinity/anti-affinity 규칙 등)이 포함된다.

스케줄러 필터링

nova.conf에 다음과 같이 기본값 구성이 되어 있다.

순서	필터 이름	설명
①	AvailabilityZoneFilter	요청된 AZ에 포함된 호스트만 통과
②	RamFilter	인스턴스에 필요한 RAM이 충분한 호스트만 통과
③	DiskFilter	디스크 공간이 충분한 호스트만 통과
④	ComputeFilter	nova-compute 서비스가 살아있는 호스트만 통과
⑤	ComputeCapabilitiesFilter	호스트의 기능(capabilities)과 요청 일치 확인
⑥	ImagePropertiesFilter	이미지 메타데이터 기반 필터링
⑦	CoreFilter	CPU 코어 수가 충분한 호스트만 통과
⑧	NUMATopologyFilter	NUMA 구성을 지원하는 경우 NUMA 요구 사항을 확인
⑨	ServerGroupAffinityFilter, AntiAffinityFilter	서버 그룹 정책 기반 (같은 호스트 또는 다른 호스트에 배치)

후보 노드 필터링 (Filter 단계)

Scheduler는 우선 전체 compute 노드 목록 중에서, 요청 조건을 만족하는 노드들을 필터링한다. 이 과정에서 여러 가지 필터들이 사용된다.

- **자원 상태 필터:** 노드의 현재 가용 리소스가 인스턴스 요구사항을 충족하는지 확인한다.
- **정책 필터:** 특정 가용 영역이나 host aggregate, 태그를 기준으로 후보 노드를 제한한다.
- **상태 및 네트워크 가용성 필터:** 해당 노드가 장애 상태가 아니며 네트워크 연결 및 기타 부가 서비스가 정상인지 점검한다.

후보 노드 평가 및 가중치 부여 (Weigher 단계)

필터링된 후보들에 대해 개별적 가중치 점수를 산출한다. 이 과정에서는 각 후보 노드의 현재 부하, 사용 가능 자원, 과거 작업 내역 등 다양한 요소가 고려된다.

가중치 알고리즘은 단순히 자원 양만 고려하는 것이 아니라, 정책에 따른 선호도를 반영하여 최적의 노드를 선택하도록 돕는다.

Placement 서비스와의 연계

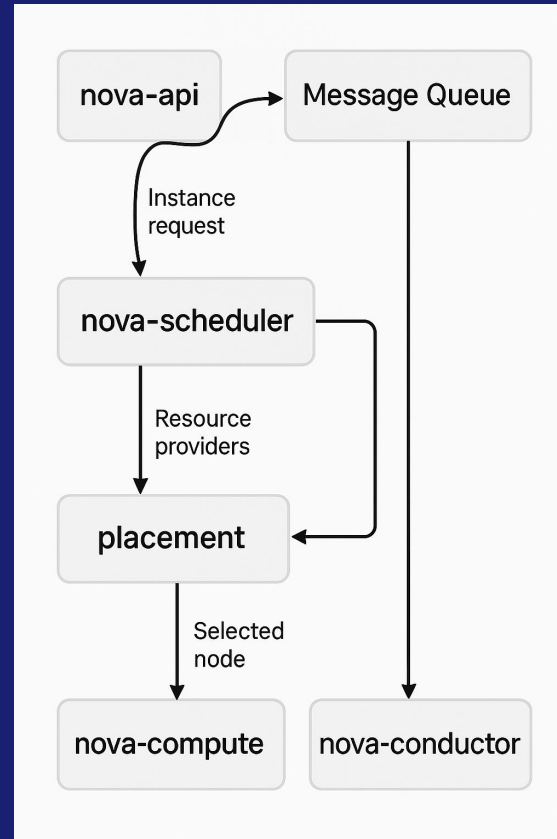
Nova Scheduler는 최신 OpenStack 아키텍처에서 Placement API를 호출해, 현재 각 노드의 자원 상태 (Inventory)를 조회하고, Reservation을 통해 자원 할당 가능성을 확인한다.

이 과정에서 Placement 서비스는 각 compute 노드(또는 Resource Provider)의 CPU, 메모리, 디스크 등 세부적인 자원 정보를 정밀하게 관리하며, 요청 조건을 만족하는 노드를 반환한다.

최종 노드 선택 및 명령 전달

필터와 가중치 평가 결과를 토대로 가장 적합한 compute 노드가 선택되고, Scheduler는 해당 노드에 인스턴스 생성 명령을 내려 배치 과정을 완료한다.

Nova-scheduler 아키텍처



Nova-conductor

nova-conductor는 Nova의 내부 컴포넌트 간 데이터 흐름 중, 데이터베이스 접근을 안전하게 중개하는 기능을 수행한다. 원래는 각 **nova-compute** 노드가 직접 데이터베이스에 접근했지만, 이는 보안 및 확장성 문제를 야기할 수 있어, 중앙에서 처리하도록 변경되었다.

예를 들어, **compute** 노드는 인스턴스를 생성하면서 상태 정보를 DB에 기록해야 하지만, 직접 DB에 접근하지 않고 **conductor**에 RPC 호출을 보내 DB 갱신을 요청한다. **conductor**는 이 요청을 수신한 후, DB에 직접 접근하여 변경사항을 반영한다.

또한 **conductor**는 스케줄링 된 인스턴스 정보 전달, 마이그레이션 준비, 볼륨 연결 정보 갱신 등 다양한 내부 처리를 맡고 있으며, Nova의 내부 통신에서 데이터 무결성과 보안을 유지하는 핵심 컴포넌트이다.

Nova-conductor

동작 순서를 서술하면 다음과 같이 동작한다.

1. 사용자 요청은 nova-api에 도달한다.
2. 요청은 메시지 큐를 통해 nova-scheduler로 전달된다.
3. nova-scheduler는 placement에 자원 목록을 요청한다.
4. Placement는 현재 가능한 compute 노드 후보를 반환한다.
5. Scheduler는 정책에 따라 적절한 노드를 선택한다.
6. 선택된 노드에 인스턴스를 배치하기 위해 nova-compute가 호출된다.
7. nova-compute는 인스턴스 상태 및 메타데이터 갱신을 conductor에 요청한다.
8. nova-conductor는 DB에 변경 사항을 안전하게 반영한다.

Nova-compute

Nova-compute는 OpenStack의 가상머신 생성 및 제어를 담당하는 컴포넌트로, 사용자의 요청을 받아 하이퍼바이저를 통해 인스턴스를 생성한다.

Nova-compute 자체가 직접 하이퍼바이저를 구성하지는 않으며, 하이퍼바이저 인터페이스(예: libvirt, VMware 등)를 통해 VM 생성을 간접 수행한다. 예를 들어, libvirt 드라이버의 경우 XML 기반 설정을 생성해 libvirtd에 전달함으로써 가상머신이 생성된다.

Nova-compute는 가상머신이 정상적으로 네트워크에 연결되도록 Neutron과 연동되며, 이를 위해 최소 하나 이상의 L2 또는 L3 네트워크 드라이버(Open vSwitch, LinuxBridge 등)가 필요하다.

Nova-compute

현재 Nova-compute가 지원하는 주요 하이퍼바이저 드라이버는 다음과 같다:

1. libvirt (KVM, QEMU, LXC 등)
2. VMware vSphere
3. XenAPI
4. Hyper-V

Placement, Cell

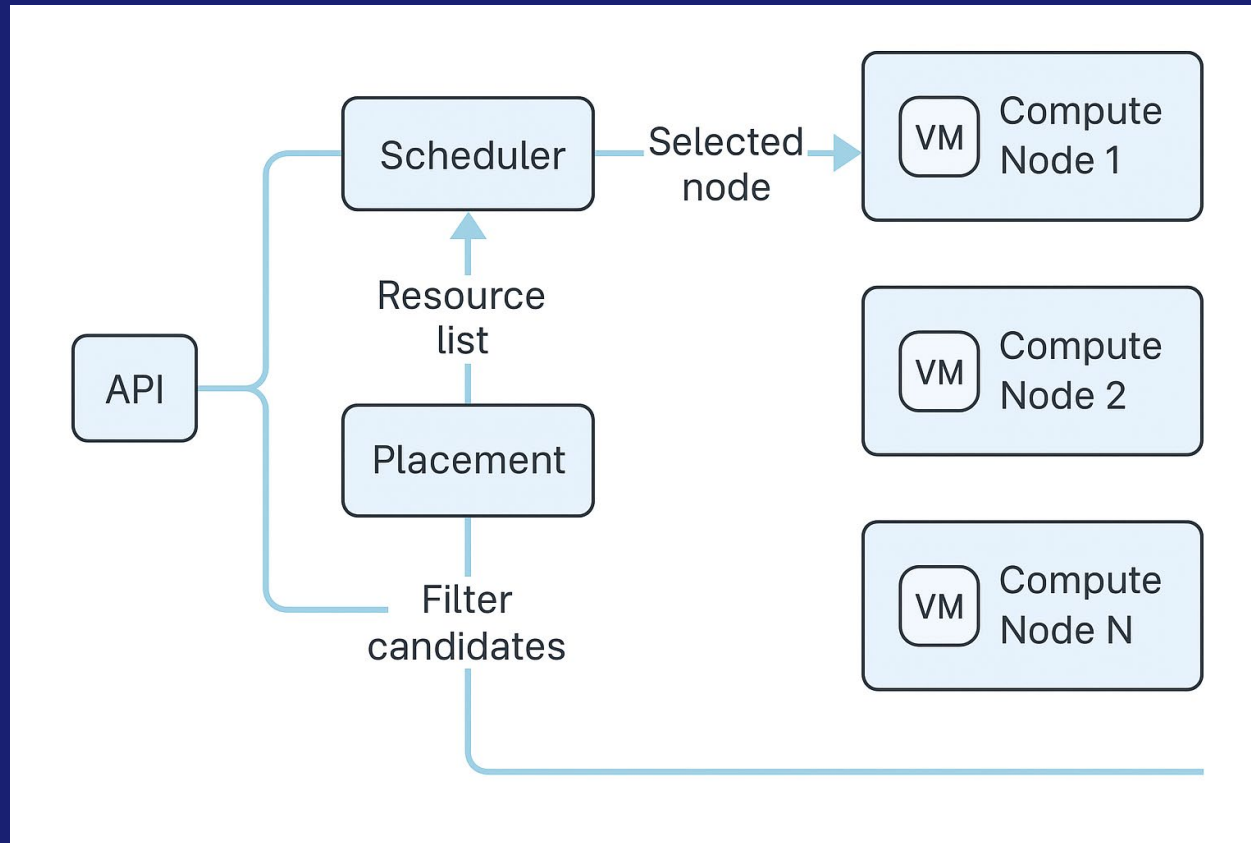
Placement

Placement은 OpenStack Nova의 리소스 추적(Resource Tracking)과 할당(Resource Allocation)을 위한 독립 서비스로, Nova 컴포넌트들과 분리되어 동작한다. OpenStack Pike 릴리즈 이후 정식 컴포넌트로 분리되었으며, Nova뿐만 아니라 Ironic, Cyborg 등의 다른 서비스에서도 활용 가능하다.

Cell

Cell은 Nova의 스케일 아웃(Scale-Out)을 위한 논리적 구조로, 수천 대의 compute 노드를 효율적으로 관리하기 위해 도입되었다. Nova는 처음부터 단일 DB 구조로 운영되었지만, 대규모 환경에서는 성능과 확장성에 한계가 있어, cell이라는 구조로 분산 배치가 가능하도록 설계되었다.

Placement, Cell 아키텍처



Placement, Cell

Placement는 다음과 같은 기능을 한다.

1. 가상머신 생성 시 필요한 CPU, RAM, DISK 등의 리소스 요구사항을 계산하고, 현재 각 compute 노드에서 사용 가능한 리소스를 비교하여, 할당 가능한 노드를 필터링한다.
2. NUMA, hugepages, PCI devices, GPU 등 고급 리소스까지 추적이 가능하며, 리소스 제공자(Resource Providers), 소비자(Consumers), 할당량(Allocations)을 추상화 기반으로 확장 가능한 자원 모델을 제공한다.

Placement, Cell

노바는 최소 한 개의 cell0번을 가지도 동작한다. 이후 cell 실제로 운영 시 사용하는 셀로 구성이 된다.

- Cell0: 스케줄링 실패 시, 실패한 인스턴스의 메타데이터를 보관하는 비 운영 셀이다.
- Cell1, Cell2...: 실제 compute node들이 존재하는 운영 셀이다. 각 셀은 자체 메시지 큐(RabbitMQ)와 데이터베이스를 가진다.
- 각 Cell은 nova-conductor, nova-compute 등의 컴포넌트를 독립적으로 갖추고, 독립적으로 스케일 가능하다.
- nova-api는 중앙에 위치하고, 셀 간 라우팅을 담당하는 nova-cells 컴포넌트를 통해 요청을 전달한다.

정리

추가된 기능에 대해서 정리하면 다음과 같다.

항목	Placement	Cell 구조
기능	자원 추적, 자원 할당	Nova 컴포넌트를 논리적으로 분산하여 확장성 확보
구조	독립 API 서비스, DB 포함	각 셀마다 독립 메시지 큐 및 DB 구성
사용 목적	정확한 리소스 매칭과 스케줄링 최적화	대규모 인프라에서 Nova 확장성과 장애 격리 지원
핵심 요소	Resource Provider, Allocation, Inventory	Cell0, Cell1~N, nova-cells 라우팅

Swift

기본 기능

Swift 소개

Swift는 OpenStack의 오브젝트 스토리지 컴포넌트로, 대규모 정적 파일 저장에 적합하며, AWS S3와 유사한 기능을 제공한다.

Swift는 Ring 구조를 기반으로 데이터를 분산 저장하며, 각 오브젝트는 해시 함수를 통해 파티션과 저장 장치에 매핑된다.

오브젝트 데이터는 파일 형태로 저장되며, 리눅스의 VFS와 로컬 파일시스템을 그대로 활용한다.

Swift의 주요 강점은 구조의 단순성과 가벼움이며, 특히 CDN이나 정적 콘텐츠 제공에 적합하다. 반면, 고성능 분산 블록 스토리지나 고가용성 환경에서는 Ceph이 더 많이 활용된다.

Swift는 기본적으로 자체 API를 사용하며, S3 API와는 직접적인 호환성을 제공하지 않지만, 일부 미들웨어를 통해 제한적 호환이 가능하다.

Swift 역사

1. Swift는 2010년 Rackspace와 NASA가 함께 시작한 OpenStack 프로젝트의 초기 구성요소 중 하나이다.
2. OpenStack의 Compute(Nova), Image(Glance)와 함께 오브젝트 스토리지를 담당하는 핵심 컴포넌트로 출발했다.
3. 초기에는 GlusterFS와 함께 오브젝트 스토리지 대안으로 고려되었으나, Swift는 보다 클라우드 네이티브한 구조로 설계되었다.
4. 2010~2014년 사이에는 주요 기업(HP, Rackspace 등)이 Swift 클러스터를 상용 운영했지만, 이후 Ceph의 부상으로 다소 영향력을 잃었다.

하지만, Swift...

현재 Swift는 점점 위기가 오고 있다. 자세한 이유는 아래 표를 참고한다.

항목	Swift	Ceph (RGW 기반)	MinIO
구조	전용 오브젝트 스토리지	유니파이드 스토리지 (블록/파일/오브젝트)	경량 고성능 오브젝트 스토리지
주요 구성요소	Proxy Server, Account/Container/Object DB	MON, OSD, RGW	단일 바이너리로 구성
데이터 중복	N-way 복제 기반	EC(Erasure Coding) 및 복제 지원	복제 및 EC 가능
확장성	수평 확장 가능하나 설계 복잡도 존재	뛰어난 수평 확장성	매우 쉬운 확장 (수동 설정 필요)

하지만, Swift...

현재 Swift는 점점 위기가 오고 있다. 자세한 이유는 아래 표를 참고한다.

항목	Swift	Ceph (RGW 기반)	MinIO
커뮤니티 지원	점점 약화되고 있음	매우 활발	활발, 특히 Kubernetes 친화적
S3 호환성	기본 미지원 (Swift API 중심)	완전 지원 (RGW)	완전 지원
설치/운영 난이도	높음 (디스크 매핑, Ring 관리 등 복잡)	높음 (구성 요소 다양)	낮음 (간단한 바이너리 실행)
사용 사례	구 버전 OpenStack 환경, 아카이빙	대규모 스토리지 통합 시스템	개발환경, 클라우드 네이티브 앱 백엔드

왜 점점 사용하지 않을까?

S3 호환성이 부족하다.

현재 대부분의 클라우드 애플리케이션이 AWS S3 API에 맞춰져 있으나, Swift는 고유 API를 사용한다.

운영이 복잡하다.

Swift는 "ring"이라는 구조로 구성되며, 장애 조치/데이터 복제/정합성 유지 등을 위해 많은 컴포넌트와 노력이 필요하다.

왜 점점 사용하지 않을까?

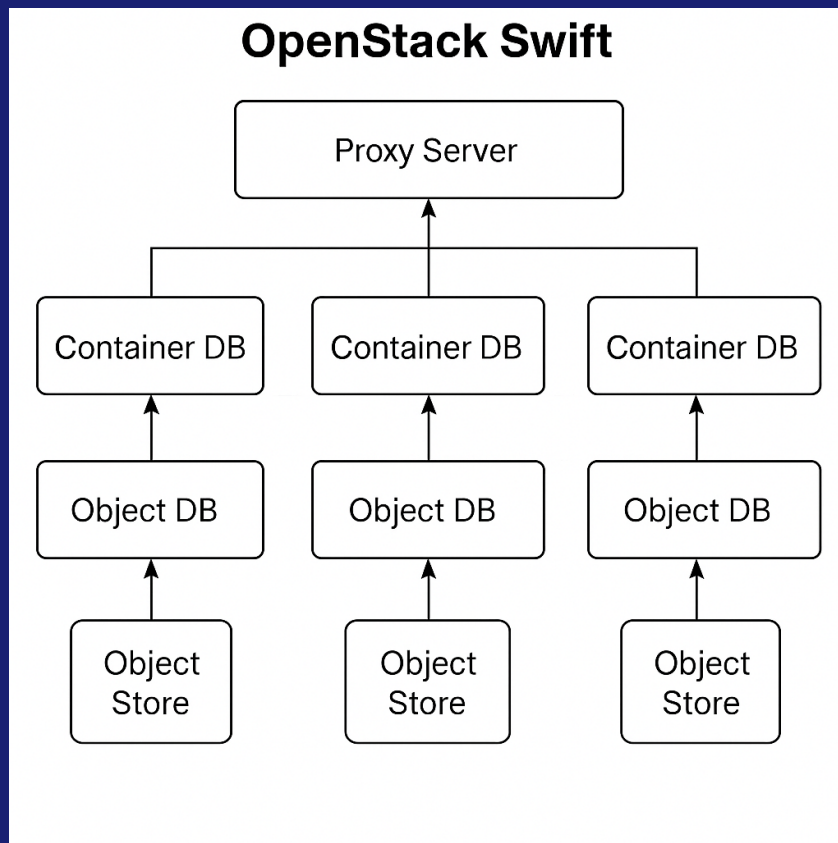
Ceph의 유니파이드 스토리지 모델이 더 매력적이다.

하나의 백엔드로 블록(Cinder), 파일(Manila), 오브젝트(RGW)를 지원할 수 있는 Ceph이 데이터센터 통합에 더 유리하다.

개발 및 커뮤니티가 활발하지 않다.

Swift의 개발 속도가 느리고, 주요 오픈소스 스토리지 트렌드에서 밀려나 있다.

Swift 아키텍처



Swift 구성요소

구성 요소	설명
Client	REST API 또는 S3 API를 통해 오브젝트 요청
Proxy Server	Swift의 중심 입구로, 요청을 처리하고 라우팅
Auth Middleware	Keystone 또는 TempAuth 기반 인증 수행
S3 API Middleware	swift3 또는 s3api 미들웨어를 통해 S3 호환 지원
Ring	해시 기반으로 오브젝트의 위치 결정
Storage Node	실제 오브젝트, 컨테이너, 계정 정보를 저장/관리
Disk	리눅스 파일시스템에 파일 형태로 저장

Swift 기능

S3 API 기능	Swift 지원 여부
PUT/GET/DELETE Bucket	지원
PUT/GET/DELETE Object	지원
Multipart Upload	부분, 일부 미들웨어에서 지원
Bucket Versioning	미지원
Access Control (ACL)	제한적 지원
Bucket Lifecycle	미지원
Presigned URL	지원 (일부 미들웨어 한정)

Neutron

기본 기능

Neutron 소개

Neutron은 OpenStack에서 네트워크 서비스를 담당하는 컴포넌트로, 초기에는 Quantum이라는 이름으로 출시되었으나 상표권 문제로 인해 현재의 이름으로 변경되었다.

초기에는 **Linux Bridge**와 **iptables**를 기반으로 네트워크를 구성하였지만, 현재는 **Open vSwitch(OVS)** 및 **OVN** 기반 구성이 권장되며, 더 높은 유연성과 확장성을 제공한다.

Neutron은 API를 통해 네트워크, 서브넷, 포트, 라우터, 보안 그룹 등의 리소스를 정의하고 제어하며, 실제 패킷 흐름은 **OVS/OVN이 L2~L4** 계층에서 처리한다.

Neutron 소개

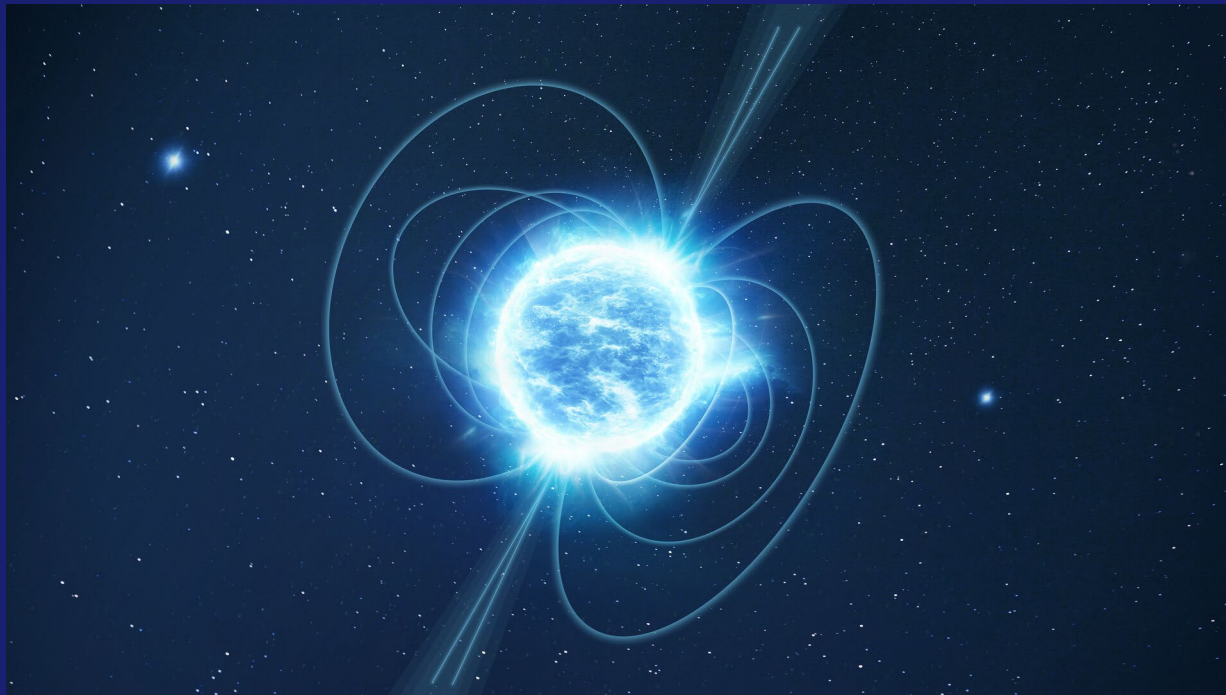
각 인스턴스의 가상 인터페이스는 네트워크 네임스페이스와 브릿지를 연결하기 위해 veth pair로 구성되며, 이는 모든 드라이버 공통 구조이다.

OVN은 고급 SDN 컨트롤 플레인으로 NAT, ACL, DHCP 등 다양한 기능을 포함하며, DPDK나 SR-IOV 등의 기술을 활용하여 성능 저하 없이도 고성능 처리가 가능하도록 설계되어 있다.

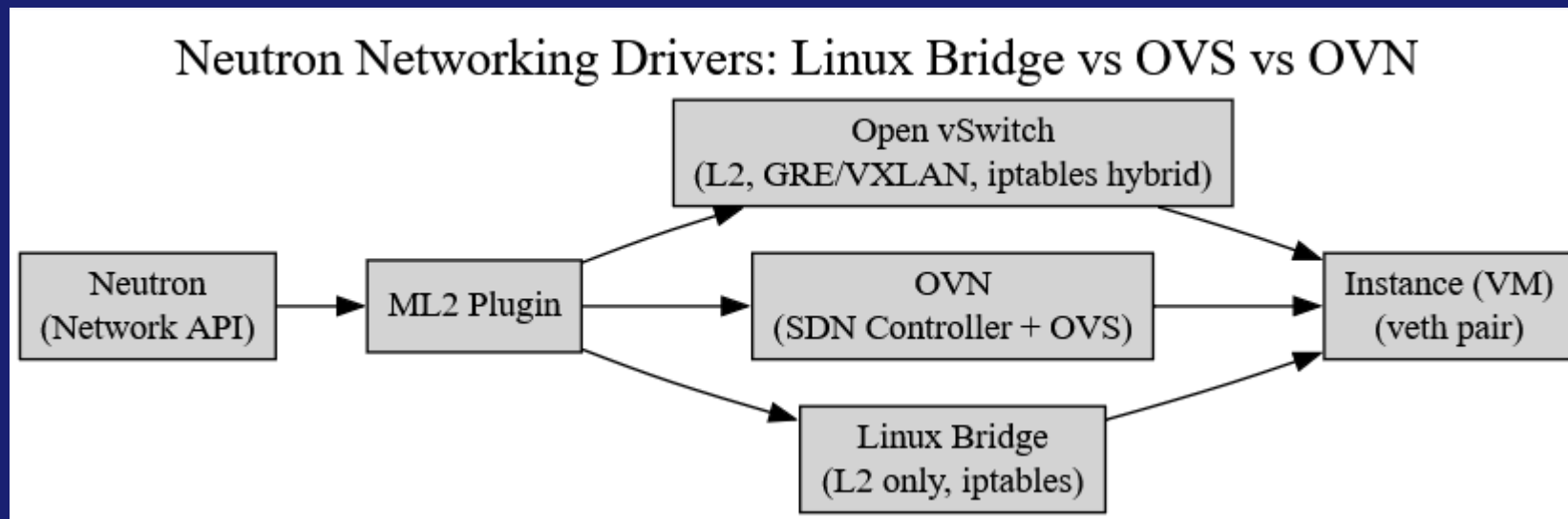
Quantum



Neutron



Neutron 소개



Neutron 소개

기존 Neutron에서 제공하던 기능과 비교 시, 다음과 같다.

항목	Linux Bridge	Open vSwitch (OVS)	OVN (OVS+SDN Layer)
L2 스위칭	기본 지원	고성능 지원	(OVS 기반)
L3 라우팅	미지원 (외부 라우터 필요)	일부 가능 (DVR 등 활용)	네이티브 지원
보안 그룹	iptables 기반	hybrid 방식	ACL 기반
VLAN 지원	예	예	예
VXLAN/GRE 터널링	아니요	예	예
SDN 지원	아니요	수동 설정 필요	예, 내장 SDN 컨트롤러 (NB/SB DB)

Neutron 소개

항목	Linux Bridge	Open vSwitch (OVS)	OVN (OVS+SDN Layer)
DHCP, NAT, L4 정책	예	아니요 (외부 서비스 필요)	예, OVN 내장 기능
성능	낮음 (커널 경로)	높음 (커널 경로, DPDK 지원)	중간~높음 (구조 복잡)
구성 복잡도	매우 쉬움	보통	복잡 (OVS + OVN DB)
대표 사용 사례	간단한 개발환경, PoC	대부분의 OpenStack 운영 환경	클라우드 네이티브 SDN, NFV

Neutron 성능 비교

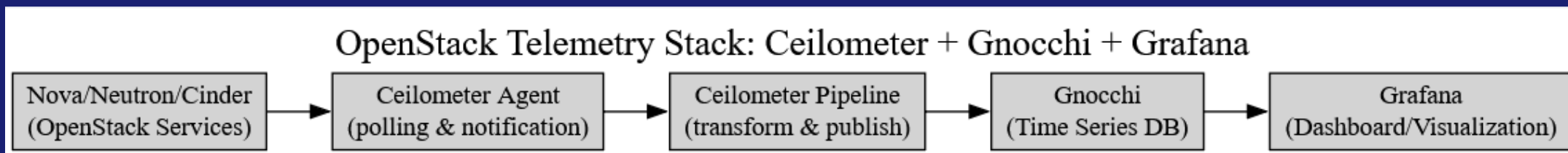
기존에 사용하던 SDN 기능과 비교 시, 다음과 같다.

항목	Linux Bridge	OVS	OVN
기본 성능 (단일 노드, L2만)	빠름 (단순 구조)	유사	느릴 수 있음
보안 그룹 + NAT 등 활성화 시	느려짐 (iptables 부하)	hybrid 방식	ACL 기반, 성능은 OVS 기반
VXLAN 등 터널링 사용 시	미지원	지원	지원
대규모 배포 시	비효율적	확장성 우수	SDN 기반 고급 기능 제공
DPDK, SR-IOV 활용 시	지원 안 함	가능	가능 (OVS 기반)

Ceilometer

기본 기능

Ceilometer 소개



Ceilometer 소개

Ceilometer는 OpenStack의 **모니터링 및 계량(metering)** 서비스를 위한 컴포넌트로, 사용자의 클라우드 자원 사용량을 측정하고 데이터를 수집하여 **빌링(billing)** 또는 모니터링 목적으로 활용할 수 있게 해준다.

이 서비스는 OpenStack의 Telemetry 프로젝트 중 하나로, 처음에는 리소스 사용량을 수집하고, 이를 통해 과금 체계를 구현하기 위한 목적에서 시작되었다.

Ceilometer는 가상머신 인스턴스, 네트워크 사용량, 블록 스토리지 등의 다양한 OpenStack 자원에 대한 정보를 수집하고 이를 중앙화 된 데이터베이스에 저장한다.

수집된 데이터는 대시보드나 외부 툴을 통해 시각화 하거나, 알람 조건으로 활용할 수 있다.

Ceilometer 소개

Ceilometer 자체는 수집만을 담당하며, 데이터 저장 및 시각화는 외부 도구에 위임한다.

대표적으로 **Gnocchi**가 시계열 데이터베이스로 사용되며, **Grafana**를 통해 시각화 할 수 있다. 필요에 따라 **collectd, fluentd**와 연동하거나, **Prometheus**등 외부 모니터링 도구로 대체되기도 한다.

이러한 구성은 사용자의 리소스 사용량 모니터링, 과금 시스템, 성능 분석 등 다양한 목적에 활용된다.

Ceilometer의 간단한 기능 분리 역사

Ceilometer는 2012년 OpenStack Folsom 릴리스에서 처음 도입되었다. 당시에는 클라우드 인프라의 리소스 사용량을 계량할 수 있는 기능이 필요하다는 수요가 많았고, Ceilometer는 이 요구를 충족시키는 핵심 프로젝트로 부상했다.

이후 OpenStack은 Telemetry 프로젝트를 확장하면서 **Aodh(알람 서비스), Panko(이벤트 저장소), Gnocchi(시간 기반 시계열 데이터 저장소)** 등의 하위 프로젝트로 Ceilometer의 기능을 분산시켰다.

Ceilometer는 주로 데이터를 수집하고, Gnocchi는 저장, Aodh는 경고 트리거, Panko는 이벤트 로깅이라는 역할 분담 구조를 갖게 된다.

Ceilometer 사용이 감소한 이유

하지만 시간이 지나면서 **Ceilometer**는 점점 덜 사용되기 시작했다. 주요 이유는 다음과 같다:

1. 복잡한 구조와 성능 문제

Ceilometer는 다양한 에이전트를 설치해야 하며, 데이터 수집 주기를 조정하거나 Gnocchi 같은 별도의 백엔드를 설정하는 등 구성이 매우 복잡하다. 또한 수집되는 데이터 양이 많아질 경우, **데이터베이스의 부하가 급격히 증가**하고, 성능 저하로 이어지는 사례가 많았다.

2. 대안의 등장

Prometheus, Grafana, Telegraf 같은 **경량화된 모니터링 툴들이 등장**하면서, 사용자들은 Ceilometer보다 더 간단하고 효율적인 도구로 넘어가기 시작했다. 특히 Kubernetes의 도입 이후, 클라우드 인프라 전체를 Prometheus 기반으로 통합 모니터링하는 사례가 늘었다.

Ceilometer 사용이 감소한 이유

3. Gnocchi의 한계 및 유지보수 문제

Ceilometer의 시계열 데이터 백 엔드로 많이 사용된 Gnocchi는 한때 OpenStack 공식 프로젝트였지만, 점차 커뮤니티 유지보수가 약화되면서 불안정해졌다. 이에 따라 Ceilometer 전체의 신뢰도도 떨어졌다.

4. 복잡한 알람 설정과 낮은 직관

Aodh와의 연동을 통해 알람 설정이 가능하지만, 설정 방식이 어렵고 사용자 친화적이지 않다는 평가가 많았다.

Ceilometer 대안?

Ceilometer는 여전히 사용하고 있으나, 아키텍처 및 구성원이 너무 점점 많아지면서 운영에 많은 어려움이 발생. 이러한 이유로, 현재는 Prometheus 기반으로 통합 및 운영을 하고 있다.

대체 도구	목적	상태	비고
Prometheus	모니터링/알람/메트릭	매우 활발	현재 가장 많이 사용됨
Gnocchi	시계열 저장소	유지보수 중단	Ceilometer 백엔드였음
Monasca	통합 텔레메트리 시스템	거의 중단	복잡하고 무겁지만 과거 SUSE 등 사용
Telegraf 등	경량 수집기	활발	Prometheus와 함께 사용
Zabbix/Nagios	시스템 모니터링	유지보수 활발	알람 및 상태 모니터링 용도 중심

Ceilometer 역할 및 관계

현재 모니터링 및 미터링 도구는 다음과 같다.

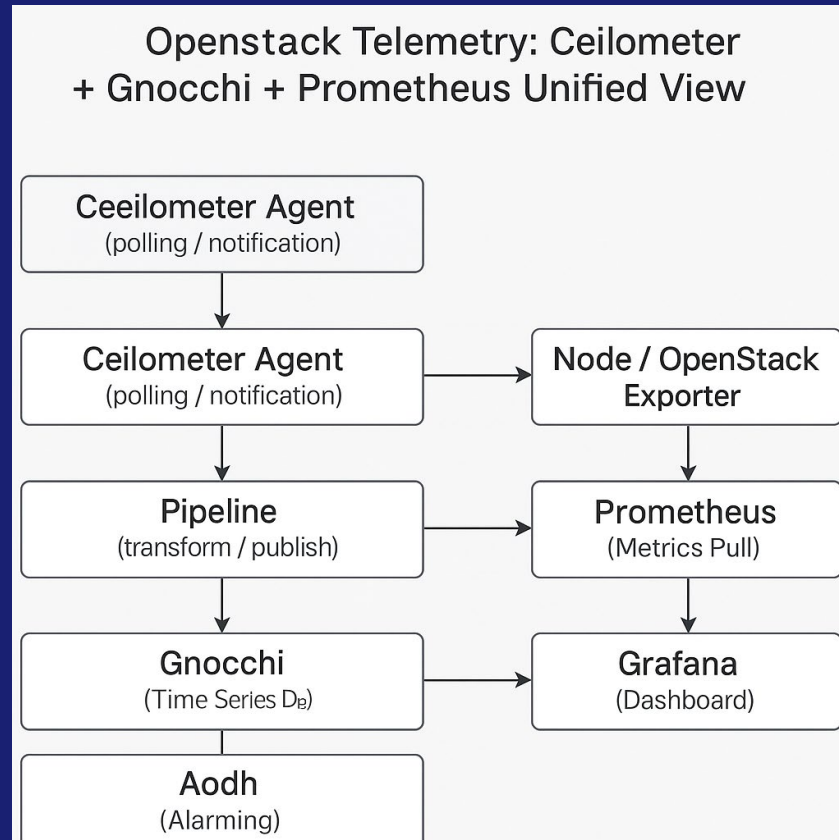
컴포넌트	역할	의존 대상	주요 기능
Ceilometer	메트릭 수집기	OpenStack 서비스, Gnocchi	리소스 사용량 수집, 알람 트리거 조건 전달
Gnocchi	시계열 데이터 저장소	Ceilometer	메트릭 저장 및 요약, API 제공
Aodh	알람 서비스	Ceilometer, Gnocchi	메트릭 기반 조건 만족 시 알람 발생 (예: CPU > 90%)

Ceilometer 역할 및 관계

위의 내용 계속 이어서...

구성 요소	역할	주요 연결 대상
Ceilometer	메트릭 수집 (agent 기반)	Gnocchi
Gnocchi	시계열 메트릭 저장소	Grafana, Aodh
Aodh	알람 조건 평가 및 트리거	Gnocchi, Nova
Exporter	메트릭 노출 (HTTP endpoint)	Prometheus
Prometheus	주기적 메트릭 수집 및 저장	Exporter, Grafana
Grafana	시각화 도구	Gnocchi, Prometheus

Ceilometer 통합 구조



Heat

기본 기능

OpenStack Heat란 무엇인가?

Heat는 OpenStack에서 제공하는 **오케스트레이션(Orchestration)** 서비스로, 클라우드 자원의 배치 및 설정을 자동화하기 위한 도구이다.

사용자는 YAML 형식의 템플릿 파일을 통해 가상머신, 네트워크, 볼륨, 보안 그룹, 키페어 등 다양한 OpenStack 자원을 한 번에 정의하고 배포할 수 있다.

Heat의 목적은 AWS의 **CloudFormation**과 유사하게, **인프라스트럭처 코드관리**(Infrastructure as Code, IaC) 하여, 복잡한 인프라 구성을 재현 가능하고 일관되게 만들 수 있도록 돕는 것이다.

이 방식은 특히 교육, 테스트, 개발 환경을 자동화하고, 반복적 작업을 줄이는 데 유용하다.

OpenStack Heat란 무엇인가?

Heat는 크게 다음과 같은 요소로 구성된다.

- Heat Engine: 템플릿을 해석하고 스택(stack)을 실제 자원으로 변환
- Heat API: 사용자 요청을 받아 엔진에 전달
- Heat Template (HOT): YAML로 작성되는 템플릿 파일
- Stack: 템플릿을 통해 생성된 실제 리소스 집합

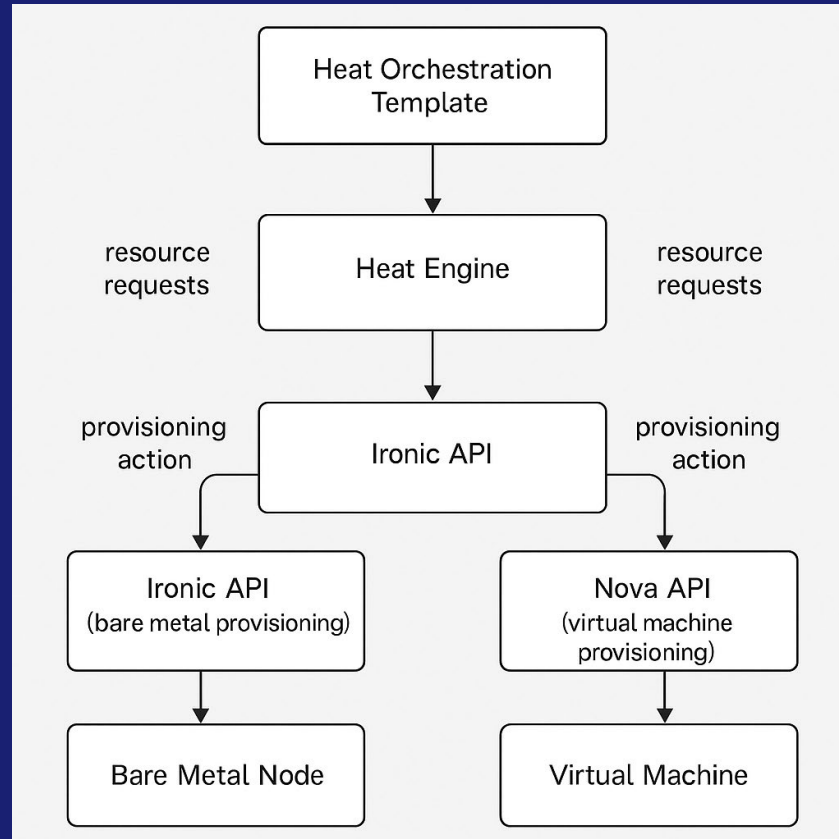
Heat의 역사

Heat는 OpenStack의 초기 릴리스인 Grizzly (2013년) 에 포함되었으며, 오케스트레이션이라는 개념을 OpenStack에 도입한 선구적 도구였다.

당시에는 인프라를 구성하기 위해 CLI 또는 Horizon 대시보드를 통해 하나씩 리소스를 만드는 수작업이 많았고, 이를 자동화하는 도구가 절실했다.

Heat는 이후 HOT(Heat Orchestration Template)라는 자체 템플릿 문법을 발전시켜 YAML을 기반으로 한 간결한 구성을 가능하게 했고, 외부 연동(Resource Plugin) 기능도 제공하며 다양한 시나리오에 활용되었다.

Heat 아키텍처



Heat 소개

Heat는 OpenStack의 오케스트레이션 도구로, YAML 기반의 HOT 템플릿을 사용하여 서버, 네트워크, 볼륨 등 자원의 자동 구성을 지원한다.

`OS::Nova::Server`, `OS::Cinder::Volume` 등과 함께 `OS::Ironi::Node` 같은 베어메탈 자원도 정의할 수 있으며,

Ironi API와 연동되어 템플릿 기반으로 베어메탈 인프라를 프로비저닝 할 수 있다.

이처럼 Heat는 OpenStack 클러스터 내부 자원 자동화에 특화되어 있으며, 다양한 리소스를 통합 관리하는 경우에는 Ansible과의 병행 사용이 추천된다.

Heat가 점점 사용되지 않는 이유

하지만 시간이 흐르면서 Heat의 사용은 점차 줄어들게 되었는데, 그 배경에는 여러 가지 현실적인 이유가 있다.

1. 기능 제한과 복잡한 템플릿 구조

- Heat는 OpenStack 자원에만 초점을 맞추고 있어, 외부 시스템 (예: DNS 설정, CI/CD, 인증 서버 등)과의 연동이 어렵다.
- 템플릿이 복잡해질수록 의존성과 순서 문제, 디버깅 어려움이 생기며, 관리가 힘들어진다.
- Jinja와 같은 템플릿은 사용하기가 어렵다.

대체 도구의 발전

현재, IaC 도구는 전체적으로 범용성 및 멀티 클라우드 시스템 위주로 되어가고 있다. 심지어, AWS, GCP에서 사용하는 IaC도구도 요즘은 점점 사용율이 낮아지는 추세이다.

- Terraform, Ansible, Pulumi, Crossplane 등 더 범용적이고 클라우드 독립적인 IaC 도구들이 Heat를 대체하게 되었다.
- 특히 Terraform은 OpenStack뿐 아니라 AWS, Azure, GCP, Kubernetes 등 멀티 클라우드 리소스를 통합 관리할 수 있어, 기업 환경에서 Heat보다 훨씬 선호된다.

Horizon UI 통합 기능 한계

Heat는 오픈스택에 통합이 되어 있지만, 어중간한 형태로 기능 제공 및 대시보드 기반에서는 거의 확인 용도만 가능하며, 관리적인 부분은 거의 지원하지 않는다.

상세한 기능은 CLI를 사용해야 되기 때문에, CLI 사용이 불가능한 사용자는 애초부터 접근이 불가능하다.

- Horizon 대시보드에서 Heat Stack을 생성하거나 관리할 수 있지만, UI에서 가능한 기능은 매우 제한적이며, 템플릿 오류 시 상세한 로그 확인이 어렵다.
- 사용자 친화적인 경험이 떨어지고, 결과적으로 CLI를 병행해야 하는 불편함이 있다.

커뮤니티 활력 저하

Heat는 여전히 OpenStack 공식 프로젝트지만, 최근 몇 년간 주요 기능의 변화나 활발한 릴리스는 줄어들었다. 대신 다른 IaC 도구들이 활발히 발전하며 커뮤니티 중심이 옮겨갔다.

다음과 같은 도구로 커뮤니티는 전환을 시작 하였다.

도구 이름	언어/표현 방식	OpenStack	MCP	특징	활동성	대체 성공사례
Terraform	HCL	공식	지원	멀티 클라우드	매우 활발	OVH, Red Hat
OpenTofu	HCL	지원	지원	오픈소스	활발	리눅스재단 후원
Ansible	YAML	지원	일부 지원	멀티 클라우드	매우 활발	CERN, OPNFV
SaltStack	YAML	지원	일부 지원	이벤트 자동화	중간 이상	SUSE OpenStack
Pulumi	Python, Go 등	지원	지원	프로그래밍 기반	활발	Snowflake 등
Crossplane	K8S CRD	제한적 지원	지원	K8s 기반	활발	Upbound 활용

Heat 대체 도구와 그 근거

1. Terraform / Opentofu

- OpenStack 공식 provider 문서: registry.terraform.io/providers/terraform-provider-openstack
- GitHub Stars (OpenTofu 기준): 10K 이상
- 많은 기업이 Heat 대신 사용, 멀티클라우드/CI-CD에 최적화

2. Ansible

- openstack.cloud 모듈 공식 문서: docs.ansible.com/collections/openstack/cloud/
- 단순 자동화에서부터 Playbook 기반 대규모 프로비저닝까지 대응

3. Pulumi

- 코드 기반 인프라를 선호하는 개발자 대상, GitOps와 잘 통합됨

4. Crossplane

- GitOps와 Kubernetes를 동시에 사용하는 환경에서 매우 빠르게 채택 중
- CNCF 소속, 활발한 PR 및 커밋 진행

Horizon

기본 기능

개요

Horizon은 OpenStack에서 제공하는 기본 웹 기반 UI 관리 도구로, 관리자와 일반 사용자 모두 GUI를 통해 손쉽게 자원을 생성하고 관리할 수 있다.

하지만 **Horizon**은 CLI나 API와 비교해 일부 고급 기능이 제공되지 않으며, 전체 기능의 약 절반 이상만 제공되는 평가가 있다.

최근에는 **Skyline**이라는 현대적인 UI 대시보드도 함께 제공되고 있지만, 아직은 핵심 서비스 위주의 기능만을 지원하며, 실제 운영 환경에서는 여전히 Horizon이 기본 선택지로 널리 사용되고 있다.

Horizon

OpenStack Horizon은 웹 기반 대시보드로, OpenStack 사용자가 프로젝트 자원(가상 머신, 네트워크, 이미지 등)을 시각적으로 관리할 수 있도록 도와주는 프론트-엔드 인터페이스이다.

Django 프레임워크 기반으로 개발되었으며, 관리자(Admin)와 일반 사용자(User) 모두가 사용할 수 있는 구조를 갖는다. CLI나 API에 익숙하지 않은 사용자들에게 직관적인 사용자 경험을 제공한다는 점에서 높은 접근성을 가진다.

Horizon은 OpenStack 초기부터 존재해 온 핵심 컴포넌트 중 하나이며, 다양한 플러그인을 통해 Nova, Neutron, Glance, Cinder, Heat 등의 서비스와 통합된다.

다만 구조상 확장성이 제한적이며, 현대적인 UX/UI 트렌드에는 다소 뒤처진다는 평가를 받는다. 특히 대규모 클라우드 환경이나 복잡한 역할 기반 접근 제어(RBAC)가 필요한 경우에는 Horizon만으로 충분하지 않은 경우도 있다.

Skyline

Skyline은 Horizon의 한계를 극복하고자 새롭게 등장한 OpenStack 대시보드 프로젝트로, React 기반의 프론트엔드와 FastAPI 기반의 백-엔드를 사용한다.

모던 웹 기술을 채택하여 성능 개선과 사용자 경험 향상을 목표로 하며, Horizon보다 빠른 렌더링과 가벼운 인터페이스를 제공한다. 그러나 Skyline은 아직 비교적 초기 단계에 있으며, 기능 커버리지가 Horizon에 비해 부족한 편이다.

특히, 세밀한 프로젝트·도메인 관리, 사용자 생성, 로깅 설정 등의 고급 관리 기능은 아직 구현되지 않았거나 제한적으로 지원된다.

그럼에도 불구하고, 향후 Horizon을 완전히 대체할 수 있는 후보로 간주되고 있으며, OpenStack 커뮤니티에서도 적극적으로 발전시키고 있다.

비교 요약

두개의 도구를 비교하면 다음과 같다.

항목	Horizon	Skyline
개발 언어	Python (Django)	Frontend: React / Backend: FastAPI
최초 릴리즈	OpenStack 초기 버전부터	OpenStack Xena (2021년 경)
주요 특징	전통적 대시보드, 다양한 서비스 플러그인	경량 UI/UX, 빠른 렌더링, RESTful 구조
확장성	제한적	마이크로서비스 기반, 높은 확장성 지향
지원 범위	대부분의 OpenStack 서비스	일부 서비스 위주 (기능 지속 확장 중)
상태	안정적, 성숙	신생, 활발한 개발 중
커뮤니티 지원	매우 활발	증가 추세

PODMAN/DOCKER

런타임

런타임 환경(Docker)

OpenStack에서 **Docker**는 전통적으로 컨테이너 관리에 사용되어 왔으며, 특히 **Kolla-Ansible**과 같은 프로젝트에서는 Docker를 전제로 많은 템플릿과 스크립트가 구성되어 있다.

Docker는 백그라운드에서 dockerd라는 **데몬이 항상 실행** 중이며, 시스템 전체 컨테이너를 중앙 집중식으로 관리한다. 이로 인해 OpenStack 서비스에서 컨테이너 상태를 모니터링하거나 제어하는 데에 편리하다.

OpenStack에서는 Dockerd가 아닌, **containerd** 기반으로 오픈스택 컨테이너를 구성 및 사용한다.

런타임 환경(Podman)

Podman은 **데몬리스** 방식으로 각 컨테이너가 사용자 프로세스로 실행되며, **rootless** 모드도 지원한다.

이는 보안상 이점을 제공하지만, OpenStack처럼 시스템 단에서 여러 계정이 동시에 컨테이너를 제어해야 하거나, 중앙에서 로그를 수집하고 상태를 모니터링해야 하는 환경에서는 일부 제약이 생길 수 있다.

또한, Podman은 일부 기능(예: volume plugin, swarm, 일부 Compose 기능 등)이 Docker와 완전히 호환되지 않는다.

가볍게 오픈스택 클러스터를 구성하는 경우 권장하며, 모든 기능을 사용하고 싶은 경우, 가급적이면 Containerd 기반으로 구성을 권장한다.

오픈스택 런타임 비교

오픈스택에서 런타임 선택 시, 다음과 같은 부분을 고려해서 선택한다.

기능/특성	Docker	Podman	비고
데몬 구조	dockerd 데몬 항상 실행	무데몬(Daemonless)	시스템 통합에 차이 발생
rootless 실행	제한적 지원	완전한 지원	Podman이 유리함
systemd 통합	별도 설정 필요	기본 통합 가능	Podman이 유리함
Kolla-Ansible 호환성	공식 지원	실험적 (추가 설정 필요)	Docker 권장
docker-compose 지원	기본 지원	별도 도구 필요	일부 기능 차이 있음
swarm 클러스터링	기본 지원	미지원	Docker 우위
volume plugin 지원	다양한 플러그인 지원	제한적 지원 (rootless에 서는 미지원)	일부 기능 제한

오픈스택 런타임 비교

위의 내용 계속 이어서...

기능/특성	Docker	Podman	비고
컨테이너 이미지 관리	docker pull/push	동일	이미지 포맷 동일
SELinux 및 AppArmor 통합	기본 지원	기본 지원 + fine-grained control	Podman이 강력함
컨테이너 디버깅	지원	대부분 동일하게 지원	유사
컨테이너 네트워크 드라이버	bridge, host, overlay 등 다양	기본 bridge, slirp4netns (rootless)	일부 제한
OCI 호환성	지원	완전 지원	동일

결론

OpenStack에서 운영 시 Docker는 여전히 표준 컨테이너 런타임으로서 가장 널리 사용되며, 대부분의 오픈스택 구성 도구(Kolla, Triple-O, Magnum 등)와 완전한 호환성을 보장한다.

반면 Podman은 보안과 유연성 면에서는 더 우수하지만, OpenStack의 기존 도구 체인과 완벽히 호환되지 않거나 일부 기능에서 추가 설정이 필요하다.

결론

Podman을 사용할 계획이라면 다음과 같은 점을 고려해야 한다.

1. rootless Podman을 사용할 경우, OpenStack이 사용하는 로그/볼륨/네트워크 통합에 제한이 생길 수 있음
2. Kolla 기반 구성에서는 docker → podman 마이그레이션이 어렵고, 호환되지 않는 CLI나 시스템 단 의존성 이슈가 존재함
3. podman generate systemd 기능 등을 활용해 자동화는 가능하지만, OpenStack의 기존 운영 모델과 통합되기 위해서는 추가적인 조정이 필요함

필요하다면 Podman 기반으로 OpenStack 환경을 테스트/학습용으로 운영할 수는 있지만, 프로덕션 환경에서는 Docker 사용이 현재로서는 더 안전하고 안정적인 선택지라고 볼 수 있다.

오픈스택 설치

배포판

RPM/DEB

kolla-ansible

배포판

현재 오픈스택 kolla-ansible에서 지원하는 배포판은 공식적으로 다음과 같다.

1. Debian
2. Ubuntu
3. Rocky/CentOS-Stream

이와 같이 지원한다. 레드햇 계열은 실제로 Rocky, CentOS-Stream 계열이 아니어도 사용이 가능하다. 수세 리눅스는 현재 지원 목록에서 제외는 되어 있지만, Rocky 이미지 기반으로 사용은 가능하다.

RPM/DEB

현재 대다수 오픈스택은 Hosted에서 RPM, DEB 기반으로 설치를 하지 않는다. 이 문서에서는 더 이상 RPM/DEB 기반으로 설치하는 다루지 않는다.

다만, kolla-build를 사용시 RPM를 사용하기 때문에, RPM 및 DEB 패키지는 모든 배포판에서 별도의 저장소로 제공하고 있다. 저장소를 검색하면 다음과 같이 화면에 출력이 된다.

```
centos-release-openstack-antelope.noarch
centos-release-openstack-bobcat.noarch
centos-release-openstack-caracal.noarch
centos-release-openstack-dalmatian.noarch
```

설치 준비

간단하게 SNO(Single Node OpenStack)를 구성하기 위해서 아래와 같이 사양이 필요하다. 이 사양은 절대 실제 서비스 제품 환경(Service Product)에서 사용하는 사양이 아니다.

- CPU: 16 core이상
- MEM: 64기가 권장. 최소 32기가(각 노드)
- DISK: 100기가 이상의 SSD 혹은 NVME디스크
- NIC: 1GiB 이상 네트워크 인터페이스
- 런타임: Podman, Docker-CE 기반으로 설치 및 구성 가능

모든 랩의 기본 사양은 오픈스택 랩 기반으로 설계가 되었으며, 일반 워크스테이션 및 노트북 기반으로 사용하는 경우, 꼭 메인보드 및 CPU에서 **nested**기능이 활성화 되어 있어야 한다.

설치 사양

오픈스택 인 오픈스택(OIO)를 구성하기 위해서는 다음과 같은 조건으로 가상머신 생성 및 구성 한다. 네트워크는 구성에 따라서 다르지만, 서비스 네트워크는 VLAN 혹은 VXLAN/Geneve 기반으로 구성한다.

아래 내용은 **오픈스택 랩 기반**에서 구성 시 네트워크 및 볼륨 사양이다.

- 볼륨 디스크: 10GiB
- 네트워크:
 - infra-net: 192.168.10.0/24, vlan10
 - net-osp-internal: 192.168.20.0/24, vlan20
 - net-osp-storage: 192.168.30.0/24, vlan30
 - infra-net: 10.0.0.0/8
- 사양: t1.openstack-controller, t1.openstack-compute

설치

kolla-ansible

오픈스택 설치

오픈스택 설치는 다음과 같은 조건으로 진행한다.

배포판

1. 레드햇 계열 배포판(RHEL/CENTOS/ROCKY/ALMA설 치 가능)
2. 데비안 계열 배포판(Debian/Ubuntu)
3. RPM, DEB를 제공하는 배포판(OpenSuSE도 설치 가능하나, RPM은 구버전만 가능)

오픈스택 설치

현재 오픈 스택은 RPM/DEB 기반으로 설치보다는 고수준 컨테이너 기반으로 설치를 권장한다. 고수준 컨테이너 런타임으로 설치시, 다음과 같은 조건이 따른다.

1. Podman

2. Docker-CE(Containerd)

모든 기능을 사용하기 위해서는 Docker기반으로 런타임 구성을 권장. 하지만, Containerd기반으로 구성하는 경우, 관리 및 런타임이 Moby기반으로 동작하기 때문에 무거움.

이러한 이유로, 몇 가지 컴포넌트를 사용하기는 어렵지만, Podman 기반으로 설치 권장. 또한, 앞으로 Kolla 기반의 컨테이너 이미지 및 애플리케이션은 Podman기반으로 마이그레이션 예정.

파이썬과 Kolla 관계

Kolla를 사용하기 위해서, 가급적이면 Python 3.11버전 이상 사용을 권장한다. 레드햇 기준으로는 기본 Python 3.9으로는 Kolla구형 그리고, kolla-ansible 명령어를 올바르게 사용하기가 어렵다.

이러한 이유로 레드햇 계열은 가급적이면 Rocky 9.x버전 기반으로 설치한다. Kolla 매뉴얼에는 Debian 리눅스에서 venv 사용을 권장하지만, 레드햇 계열 배포판도 가급적이면 venv 기반으로 kolla, kolla-ansible 설치 및 사용을 권장한다.

python, pip

다음과 같은 과정으로 파이썬 패키지를 설치 및 준비한다.

```
# dnf install python3.11 python3.11-pip.noarch
# python -m venv openstack
# source openstack/bin/activate
# pip3.11 install kolla kolla-ansible
# pip3.11 install -U pip
```

kolla-ansible

/etc/kolla/ 위치에 inventory 파일 복사한다.

```
# mkdir -p /etc/kolla
# cp -a ~/openstack/share/kolla-ansible/etc_examples/kolla/* /etc/kolla
# cp -a ~/openstack/share/kolla-ansible/ansible/inventory/multinode ~
# ls -l /etc/kolla
# ls -l ~/multinode
```

docker-registry

docker-registry는 storage서버에 구성한다.

```
# REGISTRY_HOST=192.168.20.100
# REGISTRY_DIR=/opt/registry
# CERT_DIR=/etc/docker/certs
# mkdir -p ${REGISTRY_DIR} ${CERT_DIR}
# openssl req -newkey rsa:2048 -nodes -keyout ${CERT_DIR}/domain.key \
  -x509 -days 365 -out ${CERT_DIR}/domain.crt \
  -subj "/CN=${REGISTRY_HOST}"
```

docker-registry

docker-registry는 storage서버에 구성한다.

```
# podman run -d --name registry \
  -p 5000:5000 \
  -v ${REGISTRY_DIR}:/var/lib/registry:z \
  -v ${CERT_DIR}:/certs:z \
  -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/domain.crt \
  -e REGISTRY_HTTP_TLS_KEY=/certs/domain.key \
  docker.io/library/registry:2
```

docker-registry

다른 노드도 접근 시, TLS문제가 발생하지 않도록 키를 복사한다. 모든 노드에 복사를 한다.

```
# scp /etc/docker/certs/domain.crt root@control1:/etc/pki/ca-  
trust/source/anchors/  
# scp /etc/docker/certs/domain.crt root@node4:/etc/pki/ca-  
trust/source/anchors/  
# update-ca-trust extract
```

docker-registry

컨테이너에 TLS키 오류가 발생하지 않도록 복사한다.

```
# mkdir -p /etc/containers/certs.d/${REGISTRY_HOST}:5000
# cp /etc/docker/certs/domain.crt \
/etc/containers/certs.d/${REGISTRY_HOST}:5000/ca.crt

# mkdir -p /etc/docker/certs.d/${REGISTRY_HOST}:5000
# cp /etc/docker/certs/domain.crt \
/etc/docker/certs.d/${REGISTRY_HOST}:5000/ca.crt
```

inventory

/root/inventory 설치 시, 사용할 서버를 아래와 같이 입력한다.

```
# vi /root/inventory
[control]
control1 ansible_host=192.168.10.11
control2 ansible_host=192.168.10.12
control3 ansible_host=192.168.10.13

[compute]
compute1 ansible_host=192.168.10.14
compute2 ansible_host=192.168.10.15
```

inventory

위의 내용 계속 이어서...

```
[storage]
storage ansible_host=192.168.10.16
```

```
[network]
network ansible_host=192.168.10.17
```

```
[monitoring]
control1
```

```
[deployment]
localhost ansible_connection=local
```

globals.yml

/etc/kolla/globals.yml에 다음처럼 작성한다.

```
# vi /etc/kolla/globals.yml
---
kolla_base_distro: "rocky"
kolla_install_type: "binary"
openstack_tag: "2024.02"

# VIP 설정
kolla_internal_vip_address: "192.168.10.250"
kolla_external_vip_address: "192.168.20.250"
keepalived_virtual_router_id: 51
keepalived_password: "keepalivedpass"
```

globals.yml

위의 내용 계속 이어서...

네트워크 인터페이스

network_interface: "eth0"

internal

neutron_external_interface: "eth1"

storage_interface: "eth2"

compute, storage 노드에서만 존재

globals.yml

위의 내용 계속 이어서...

```
# Docker Registry 설정 (내부 미러)
```

```
docker_registry: "192.168.20.100:5000" # storage 노드 external IP
```

```
docker_namespace: "kolla"
```

```
docker_registry_insecure: "yes"
```

```
kolla_copy_ca_into_containers: "no"
```

globals.yml

/etc/kolla/globals.yml에 다음처럼 작성한다.

MTU 설정

```
network_mtu: 9000
```

```
neutron_global_physnet_mtu: 9000
```

서비스 활성화

```
enable_cinder: "yes"
```

```
enable_cinder_backend_nfs: "yes"
```

```
enable_heat: "yes"
```

```
enable_horizon: "yes"
```

```
enable_keepalived: "yes"
```

kolla-ansible 실행 준비

다음 명령어로 설치를 준비한다.

```
# kolla-ansible bootstrap-servers -i multimode
# kolla-ansible pull --registry 192.168.20.100:5000 \
--tag 2025.02 --push
# kolla-ansible certificates -i multimode
# kolla-ansible prechecks -i multinode
# kolla-ansible deploy -I multinode
```

대표적인 확장 컴포넌트

Octavia

Designated

개요

오픈스택에서 확장 컴포넌트를 사용하기 위해서 컨테이너 런타임은 가급적이면 containerd기반으로 구성을 권장한다. Podman기반으로 구성이 가능하지만, Podman은 rootless기반으로 구성이 되어 있기 때문에 몇몇 기능은 권한 부족으로 올바르게 동작하지 않는다.

많이 사용하는 확장 서비스는 보통 다음과 같다.

1. Octavia
2. Designated

이러한 이유로, 가볍게 코어 서비스만 사용하는 경우, Podman. 그게 아니면 Containerd기반으로 구성한다.

Octavia 설명

교육 마지막에 추가할 서비스는 앞에서 말한 두 가지 서비스를 확장할 예정이다.

Designated 설명

교육 마지막에 추가할 서비스는 앞에서 말한 두 가지 서비스를 확장할 예정이다.

명령어

openstack-client

로그인

오픈스택 명령어를 사용하기 위해서 **openstack** 명령어 혹은 컴포넌트 명령어에 API 및 SECRET키를 전달해야 한다. 여기서 말하는 SECRET은 아이디 및 비밀번호까지 포함이다. 전달하기 위해서 두 가지 방법을 제공하고 있다.

1. BASH기반의 환경 변수 전달
2. YAML기반의 환경 변수 전달

초창기 및 기존 사용자는 1번 방법을 많이 사용하였으나, 여러 클러스터를 관리하면서 1번으로 여러 클러스터 관리가 어려운 부분이 있다. 이러한 이유로 여러 클러스터를 접근하기 위해서 YAML기반으로 작성 및 접근 관리 방법이 제안이 되었고, 현재 최근 릴리즈는 YAML기반으로 CLI사용을 권장하고 있다.

명령어

오픈스택 명령어는 기본적으로 파이선으로 작성되어 있다. 기능이 추가가 되면, PIP를 통해서 명령어를 확장해야 한다. 다만, **openstack** 통합 명령어는 아직 모든 컴포넌트의 명령어를 다 수행하지 않기 때문에, 종종 컴포넌트 명령어를 통해서 구성 및 관리가 필요한 경우가 있다.

오픈스택 명령어 사용 방법은 다음과 같다.

openstack <COMPONENT_NAME> <VERBE> <OPTIONS> <OBJECT_NAME>

보통 위와 같은 형식으로 명령어를 작성 및 수행한다.

명령어 확장

오픈스택 명령어 설치하는 다음과 같은 방법으로 진행한다. 다만, 배포판 및 구성에 따라서 조금씩 다르다.

```
# python3 --version
# pip3 --version
# sudo apt update
# sudo apt install python3 python3-pip -y
# python3 -m venv openstack-env
# source openstack-env/bin/activate
# pip install python-openstackclient
```

명령어 확장

명령어를 확장하기 위해서 다음과 같이 추가가 가능하다. 다만, 구성에 따라서 조금씩 달라진다.

```
# pip install python-glanceclient python-neutronclient
```

서비스	패키지 이름
Compute(Nova)	python-novaclient
Image(Glance)	python-glanceclient
Network(Neutron)	python-neutronclient
Block Storage(Cinder)	python-cinderclient
Identity(Keystone)	python-keystoneclient
Object Storage(Swift)	python-swiftclient
Heat	python-heatclient

명령어 보안(clouds.yaml)

오픈스택 명령어를 좀 더 안전하게 사용하기 위해서 컨테이너 기반으로 실행이 가능하다.

다음과 같이 포드만에서 사용이 가능하다. 아래와 같이 실행하면, 사용자가 사용하는 `clouds.yaml` 파일 기반으로 오픈스택 API에 접근이 가능하다.

```
# podman run -it --rm \  
-v $HOME/.config/openstack:/root/.config/openstack \  
docker.io/openstackclients/openstackclient:latest
```

명령어 보안(ENV)

혹은 기존 방식을 사용하는 경우, 다음과 같이 적용이 가능하다.

```
# vi $HOME/.config/openstack
export OS_AUTH_URL=https://<API_URL>:5000/v3
export OS_PROJECT_NAME=admin
export OS_USERNAME=admin
export OS_PASSWORD=<PASSWORD>
export OS_USER_DOMAIN_NAME=Default
export OS_PROJECT_DOMAIN_NAME=Default
# podman run -it --rm \
-v $HOME/.config/openstack:/root/.config/openstack \
docker.io/openstackclients/openstackclient:latest
```

명령어 보안(ENV)

혹은 기존 방식을 사용하는 경우, 다음과 같이 적용이 가능하다.

```
# podman run -it --rm \  
  -e OS_AUTH_URL \  
  -e OS_PROJECT_NAME \  
  -e OS_USERNAME \  
  -e OS_PASSWORD \  
  -e OS_USER_DOMAIN_NAME \  
  -e OS_PROJECT_DOMAIN_NAME \  
  docker.io/openstackclients/openstackclient:latest
```

명령어 보안(alias)

영구적으로 사용하기 위해서 최종적으로 아래와 같이 스크립트에 적용한다.

```
# alias osc="podman run -it --rm \  
-v $HOME/.config/openstack:/root/.config/openstack \  
docker.io/openstackclients/openstackclient:latest"
```

오픈스택 CLI 및 대시보드

Horizon/Skyline

OpenStack CLI

운영자는 무엇을 사용해야 하는가?

오픈스택 클러스터를 운영하는 관리자에게 있어 CLI(Command Line Interface)는 필수적인 도구로 간주된다. CLI는 단순한 명령 실행을 넘어서, **스크립트 기반 자동화, 배치 작업, 그리고 대규모 리소스 관리**에 최적화되어 있다.

예를 들어 수십 개의 인스턴스를 생성하거나, 다수의 네트워크와 볼륨을 조작하는 작업은 CLI를 통해 몇 줄의 스크립트로 처리할 수 있으며, 이는 운영의 일관성과 재현성을 크게 향상시킨다.

운영자는 무엇을 사용해야 하는가?

반면, **Horizon 대시보드**는 웹 기반 그래픽 사용자 인터페이스(GUI)로, 리소스의 상태를 직관적으로 확인하거나 간단한 조작을 수행할 때 유용하다. 간단한 작업이나 혹은 관리를 위해서 대시보드로 처리가 가능하다. 하지만, **대규모/반복적인 작업**에는 매우 부적절하다.

CLI는 실질적인 인프라 운영의 핵심이며, Horizon은 시각적 모니터링이나 간단한 작업 처리, 교육 목적 등에 적합한 도구이다. 이 둘을 적절히 병행하면, 운영자는 신속성과 안정성을 동시에 확보할 수 있다.

결론적으로, 오픈스택 운영자는 **CLI를 기본 도구로 사용**하되, Horizon 대시보드는 **보조 수단**으로 활용하는 것이 바람직하다.

대시보드 허용 범위

약 40~60% 수준의 CLI 기능만 대시보드(Horizon)에서 제공됩니다.

- 사용자 권한에 따라 약간의 차이는 있으나, 관리자 기준으로 전체 기능 중 절반을 넘기 어려움

Horizon은 전체 OpenStack CLI 기능의 절반 가량만 제공하며, 특히 고급 기능과 자동화, 디버깅, 세밀한 설정은 대부분 CLI 또는 REST API를 통해서만 가능하다.

따라서 운영자 입장에서는 대시보드는 시각적 보조 수단으로 활용하고, 실제 운영 핵심은 CLI를 중심으로 구성해야 한다.

항목 살펴보기

다음과 같은 항목에 대해서 비교가 가능하다.

기능 영역	CLI 지원	Horizon 지원	비고
인스턴스 생성/삭제	지원	지원	기본 기능
네트워크, 서브넷, 라우터	지원	지원	다소 제한된 옵션
보안 그룹 설정	지원	지원	고급 규칙 작성 어려움
사용자/프로젝트 관리	지원	지원	도메인 기반 권한 제어는 CLI 권장
스택(Heat) 관리	지원	네니요	템플릿 등록/로그 확인 어려움
볼륨(Cinder) 조작	지원	지원	세부 옵션 미지원
이미지 등록/관리	지원	지원	CLI는 glance 옵션 더 풍부

항목 살펴보기

다음과 같은 항목에 대해서 비교가 가능하다.

기능 영역	CLI 지원	Horizon 지원	비고
Floating IP 연결	지원	지원	제한 없음
서비스 상태 확인	지원	미지원	openstack service list, status 불가
정책, RBAC, Role 설정	지원	미지원	CLI 전용
고급 디버깅/로그 조회	지원	미지원	Horizon은 대부분 표시 불가
메타데이터, Flavor 수정	지원	미지원	CLI/REST API만 가능
대규모 반복 작업	지원	미지원	GUI는 자동화 불가

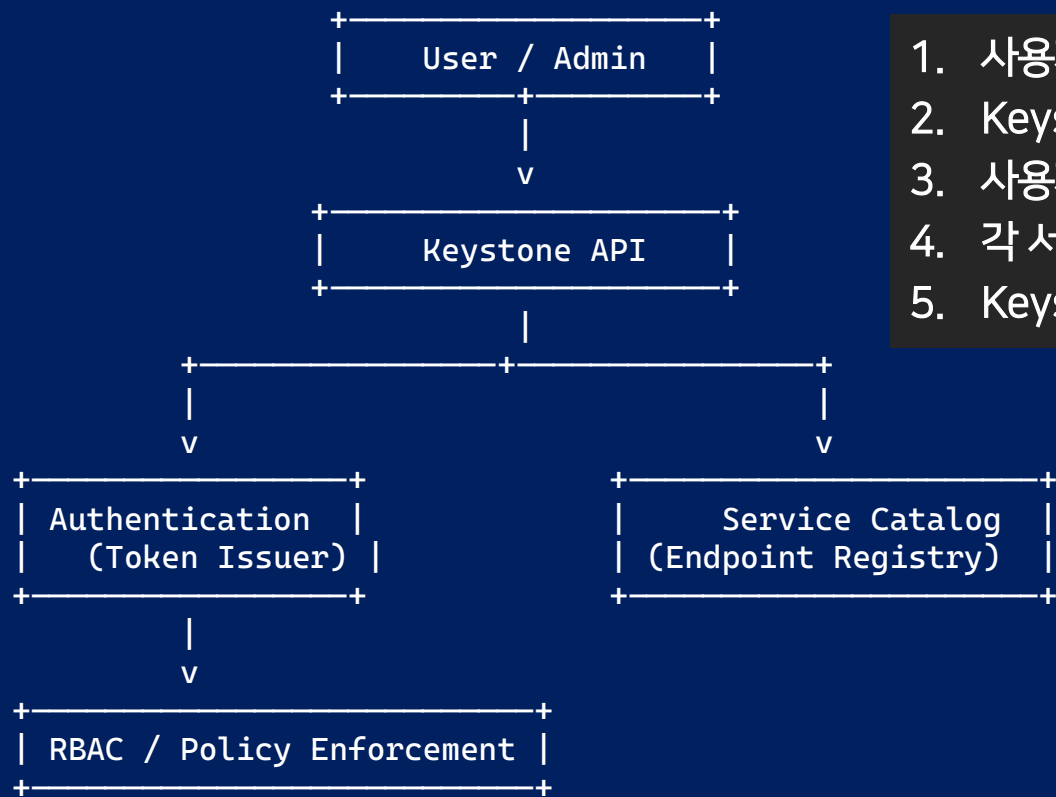
keystone

인증

2025-06-06

개요

아래와 같이 Keystone 동작한다.



1. 사용자가 Keystone에 사용자명/비밀번호로 로그인 요청
2. Keystone이 사용자 인증 후 토큰 발급
3. 사용자는 이 토큰을 가지고 다른 OpenStack 서비스에 접근
4. 각 서비스는 Keystone에 토큰 유효성 확인 요청
5. Keystone이 유효한 토큰임을 확인하고 응답

개요

1. OpenStack의 Identity Service 역할
2. 인증(Authentication) + 권한(Authorization)
3. 사용자, 프로젝트, 역할, 서비스, 엔드포인트 관리
4. 멀티 도메인 및 역할 기반 접근 제어 지원 (RBAC)

기능 영역	설명
인증(Authentication)	사용자 인증 후 토큰 발급
권한(Authorization)	역할 기반 권한 부여 (RBAC)
서비스 카탈로그	API 서비스 목록 및 위치 정보 제공
정책 엔진	policy.yaml 기반 권한 규칙
멀티도메인 지원	도메인별 사용자/프로젝트 분리 관리
외부 연동	LDAP, SAML, OAuth, Kerberos 등과 통합 가능

토큰

올바르게 로그인인지 되었는지 다음 명령어로 토큰 확인한다.

```
$ openstack token issue
```

Field	Value
expires	2025-05-14T14:02:13.321693Z
id	gAAAAABk0Lt0yWclE7 ...
project_id	4b647aa4233b4f8d87d7b6e3d3e3b7d6
user_id	f2e8312a123e49deaf8f3d7e1b2ab123

사용자 생성

사용자 생성은 다음처럼 가능하다. 사용자 이름은 user1으로 구성.

```
$ openstack user create \  
  --domain default \  
  --password-prompt \  
  --email user1@example.com \  
  user1
```

User Password:

Repeat User Password:

프로젝트 생성

사용자가 사용할 프로젝트가 최소 한 개가 있어야 한다. 아직 "--domain" 옵션은 사용하지 않는다.

```
$ openstack project create --domain default --description "Project for user1" user1
```

Field	Value
description	Project for user1
domain_id	default
enabled	True
id	11223344aabbccddeeff0011223344
is_domain	False
name	user1
parent_id	default

역할 할당

사용자, 프로젝트가 구성이 되면 역할을 할당해야 한다. 이를 통해서 RBAC 구성이 가능하다.

```
$ openstack role add \  
  --project user1 \  
  --user user1 \  
  member
```

역할 할당

할당 및 연결이 완료가 되면, 다음 명령어로 확인이 가능하다.

```
$ openstack role assignment list \  
  --user user1 \  
  --project user1 \  
  --names
```

Role	User	Group	Project	Domain	Inherited
member	user1		user1		False

사용자 비밀번호

비밀번호 변경은 다음처럼 가능하다.

```
$ openstack user set --password 새비밀번호 user1  
$ openstack user set --password 'NewP@ssw0rd!' user1
```

리전(Region) 소개

오픈스택은 Region 및 도메인 기능을 제공한다. 이를 통해서 자원을 좀 더 지역적 그리고 도메인별로 분류 및 생성 관리가 가능하다. Region은 오픈스택 Diablo(2011년 9월)부터 도입이 되었지만, 이때는 목적이 단순하였다.

1. OpenStack의 초기 릴리즈에서부터 리전(Region) 개념은 존재
 2. 엔드포인트 등록 시 Region 이름을 명시하는 수준으로, 서비스 지리적 분산을 위한 단순 태그 용도
- 하지만, Havana(2013년)부터 서비스 Catalog(카달로그)가 Region으로 포함이 되면서 지역적으로 Endpoint(접근점) 관리가 가능하게 되었다.

이를 통해서 단일 오픈스택 클러스터에서 여러 지역 및 데이터 센터를 논리적으로 관리가 가능하게 되었다.

Region 및 Domain 용도

이 두개의 용도는 간단하게 다음과 같이 설명이 가능하다.

1. Region은 보통 지역 혹은 데이터센터 분리 용도라 사용한다
2. 도메인은 고객/기관/조직을 관리 시 사용한다

도메인 소개

도메인은 리전보다 하위 개념이다. 이를 통해서 도메인을 통해서 조직 혹은 용도별로 분류가 가능하게 되었다.

OpenStack Grizzly (2013년 4월 출시)

1. Identity API v3가 도입되면서 도메인(domain) 개념이 등장
2. Grizzly에서 Keystone은 Identity v3 API를 통해 사용자, 프로젝트, 그룹을 도메인 단위로 격리 가능

도입배경

1. 기존 v2 API에서는 사용자/프로젝트가 전역으로만 존재했기에 조직 격리 구조가 어려웠음
2. v3 API 도입으로 도메인 기반 멀티 테넌시 구현이 가능해짐

정리

리전 및 도메인을 정리하면 다음과 같다.

개념	최초 릴리즈	릴리즈명	API 버전	설명
Region	2011.09	Diablo	v2	서비스 카탈로그용 태그 (엔드포인트 분리)
Domain	2013.04	Grizzly	v3	사용자/프로젝트 격리 단위

리전+도메인 기반 구성

리전+도메인 기반으로 구성하기 위해서 아래와 같은 조건으로 구성한다.

항목	예시
도메인	dev-domain, prod-domain
사용자	dev-user, prod-user
프로젝트	dev-project, prod-project
역할	dev-role, prod-role
리전	RegionOne, RegionTwo (서비스 등록 시 사용)

도메인 생성

도메인을 클러스터에 생성한다.

```
# openstack domain create dev-domain --description "Development domain"  
# openstack domain create prod-domain --description "Production domain"
```

사용자 생성

사용자를 다음과 같이 클러스터에 생성한다.

```
# openstack user create \  
  --domain dev-domain \  
  --password 'DevUserPass123!' \  
  dev-user  
  
# openstack user create \  
  --domain prod-domain \  
  --password 'ProdUserPass123!' \  
  prod-user
```

프로젝트 생성

프로젝트를 도메인 기반으로 생성한다.

```
# openstack project create \  
  --domain dev-domain \  
  --description "Development Project" \  
  dev-project  
  
# openstack project create \  
  --domain prod-domain \  
  --description "Production Project" \  
  prod-project
```

역할 생성

역할은 도메인과 무관하게 생성 및 사용이 가능하다.

```
# openstack role create dev-role
# openstack role create prod-role

# openstack role add --user dev-user \
  --project dev-project dev-role

# openstack role add --user prod-user \
  --project prod-project prod-role
```

시나리오

아래와 같은 구조로 자원들을 생성한다.

```
[dev-domain]
└─ dev-project
    └─ dev-user (role: dev-role)
```

```
[prod-domain]
└─ prod-project
    └─ prod-user (role: prod-role)
```

```
[RegionOne] — Keystone endpoint → http://controller.region1.local:5000/v3
```

```
[RegionTwo] — Keystone endpoint → http://controller.region2.local:5000/v3
```

ROLE GLOBAL DOMAIN

오픈스택은 역할 생성 시, 두 가지 옵션이 있다. 아래 사항을 잘 이해하고 사용해야 한다.

구분	--domain 없음 (글로벌)	--domain 있음 (도메인 한정)
적용 범위	모든 도메인	지정한 도메인 내부 전용
사용 예시	member, admin 등의 공통 역할	dev-role, prod-role 등 도메인 특화 역할
생성 예시	openstack role create admin	openstack role create --domain dev-domain dev-role

ROLE GLOBAL DOMAIN

특정 도메인의 역할을 찾고 싶은 경우, 아래와 같이 조회를 한다. 대다수 Keystone 자원은 도메인에 따라서 확인이 되는 경우, 안되는 경우가 있다.

```
$ openstack role list --domain dev-domain
```

Glance

이미지

개요

OpenStack의 Glance는 이미지 관리 서비스로서, 클라우드 환경에서 가상 머신 이미지를 등록, 저장, 검색, 공유 및 배포하는 역할을 수행한다. Glance는 다양한 디스크 및 컨테이너 포맷을 지원하며, 이미지의 메타데이터를 포함한 체계적인 관리를 가능하게 한다.

Glance는 이미지 저장소를 로컬 파일 시스템, NFS, Ceph(RADOS), 또는 Swift 오브젝트 스토리지 등 다양한 백엔드로 설정할 수 있으며, 확장성과 유연성을 고려한 설계를 가지고 있다. 여러 컨트롤러를 구성하여 사용하는 경우, 가급적이면 NFS기반으로 구성을 권장한다.

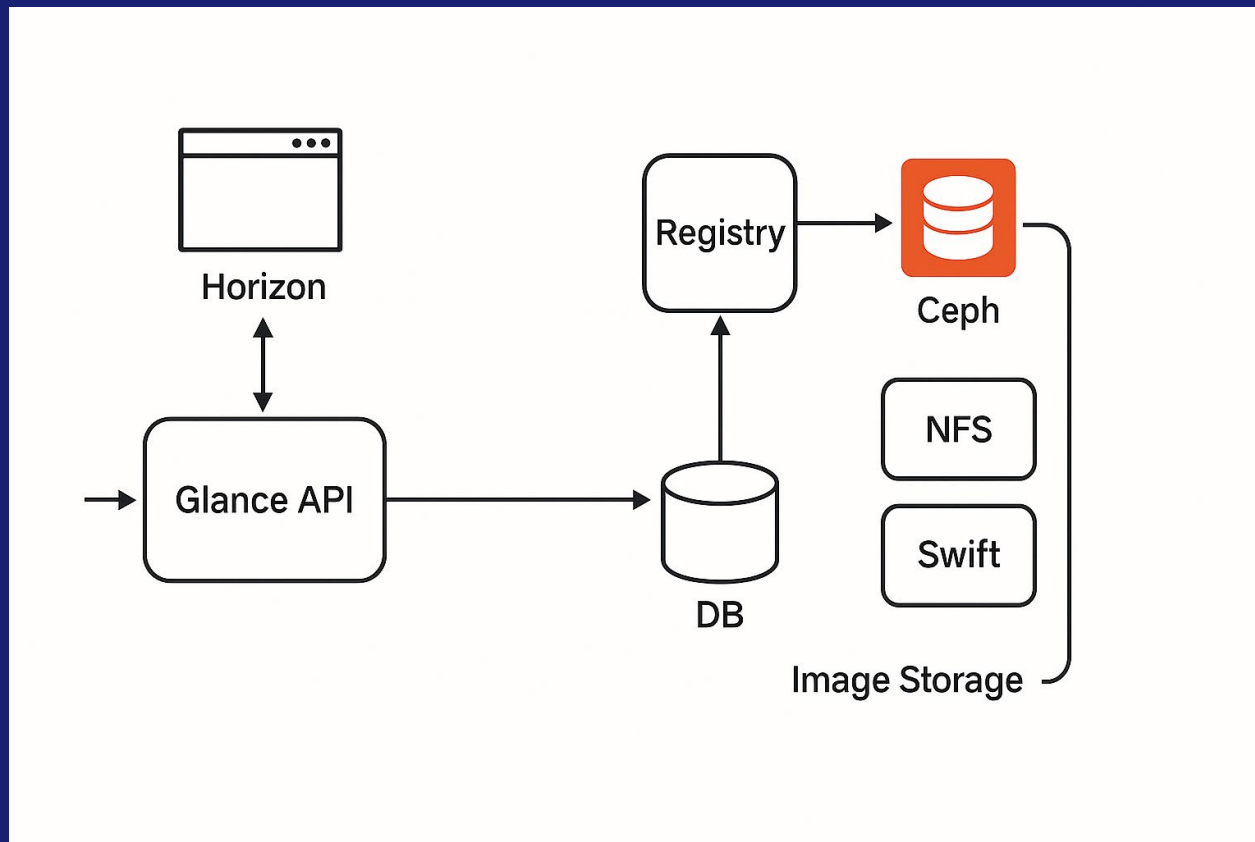
개요

Glance는 RESTful API 기반으로 동작하며, 다른 OpenStack 서비스들과 긴밀히 통합되어 있다. 예를 들어, Nova는 인스턴스 생성 시 Glance에서 이미지를 조회하고 가져오며, Cinder는 볼륨 생성을 위해 Glance의 이미지를 사용하기도 한다.

운영자는 Glance를 통해 이미지의 수명주기를 제어가 가능하다. 이미지 등록 시 운영체제, 배포판, 버전 등의 메타데이터를 함께 지정하여, 인스턴스 구성 시 도움이 되도록 한다.

최종적으로 Glance는 OpenStack 내에서 가상 자원 운영의 시작점을 제공하는 핵심 구성 요소이며, 지속 가능한 이미지 관리와 보안 정책 적용을 위한 필수적인 서비스로 간주된다.

아키텍처



지원 이미지

Glance에서 지원하는 이미지는 다음과 같다.

포맷 이름	설명
raw	가공되지 않은 디스크 이미지 (가장 기본)
qcow2	QEMU Copy-on-write v2 포맷 (스냅샷/압축 지원)
vmdk	VMware 디스크 이미지
vhd	Microsoft Virtual PC용
vhdx	Microsoft Hyper-V용 확장 디스크
vdi	VirtualBox용 이미지
iso	ISO 9660 CD-ROM 이미지 (부팅용 ISO 포함 가능)
aki	Amazon 커널 이미지
ari	Amazon 램디스크 이미지
ami	Amazon 머신 이미지

컨테이너 포맷

아래는 이미지 컨테이너 형식이다.

포맷 이름	설명
bare	컨테이너가 없는 순수 디스크 이미지 (가장 일반적)
ovf	Open Virtualization Format (OVA에서 사용됨)
aki	Amazon 커널 이미지용 컨테이너
ari	Amazon 램디스크 이미지용 컨테이너
ami	Amazon 머신 이미지용 컨테이너
docker	Docker 이미지 (컨테이너로서 사용 가능)

가상머신 이미지

이미지 업로드는 다음과 같이 실행한다.

```
# openstack image create "ubuntu-22.04" \  
--file ubuntu-22.04.qcow2 \  
--disk-format qcow2 \  
--container-format bare \  
--public
```

컨테이너 이미지

- OpenStack Wallaby 이후 버전에서 Glance는 container_format=docker를 지원함.
- 하지만 Nova는 이 이미지를 부팅용으로 사용하지 않음 → 보통 Kubernetes 이미지 저장소 대안 또는 CI/CD 백업용으로 활용됨.
- Glance backend가 Swift일 경우 이미지 분산 저장에 유리함.

컨테이너 이미지

이미지 업로드는 다음과 같이 실행한다.

```
# docker pull nginx:latest
# docker save nginx:latest -o nginx.tar
# openstack image create "nginx-container"
  --file nginx.tar \
  --disk-format raw \
  --container-format docker \
  --public
```

옵션	설명
--disk-format raw	디스크 포맷은 raw로 고정
--container-format docker	docker/OCI 컨테이너 이미지임을 명시
--file nginx.tar	tar 형식의 Docker 저장 이미지

Glance에서 Docker 이미지 확인

업로드 된 이미지는 다음과 같이 확인 가능하다.

```
# openstack image show nginx-container
```

Field	Value
name	nginx-container
disk_format	raw
container_format	docker
visibility	public
size	139586048

컨테이너 이미지 배포 권고 방법

이러한 이유로 다음과 같은 방법으로 이미지 배포를 권장 및 권고한다.



컨테이너 이미지 지원 영역

컨테이너 이미지는 업로드가 가능하지만, Glance를 통한 이미지 배포는 불가능하다.

제한 사항	설명
업로드 가능	Glance가 저장소로 사용될 수 있음
실행 불가	Nova, Cinder 등은 Docker 포맷을 직접 실행하거나 부팅할 수 없음
활용	Kubernetes에서 독립적인 이미지 저장소가 필요한 경우 (예: Glance-backed registry 역할), 또는 CI/CD 백업용

컨테이너 레지스트리

Glance

2025-06-06

컨테이너 레지스트리

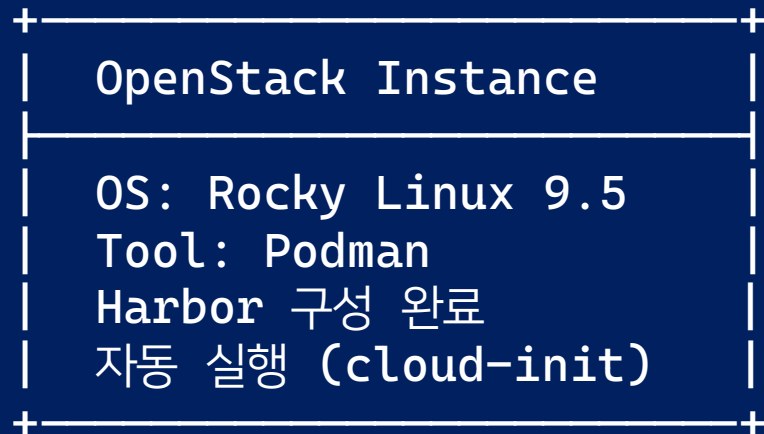
내부 서비스에 컨테이너 이미지를 배포 및 관리하기 위해서 반드시, 최소 한 개의 레지스트리 서버가 필요하다. 일반적으로 사용하는 소프트웨어는 두 가지가 있다.

1. docker-registry
2. Harbor
3. Gitlab

docker-registry는 가볍지만, 명령어 위주로 사용해야 되기 때문에, 일반적인 사용자 및 개발자는 어렵다. 이러한 이유로 많은 기업 혹은 단체는 harbor, Gitlab를 선호한다. 다만, Gitlab은 자원을 많이 사용하기 때문에 레지스트리 서버 목적으로는 Harbor를 권장한다.

컨테이너 레지스트리

구성 아키텍처는 다음과 같다.



|
(expose HTTP/HTTPS)

v

Browser/User → [http\(s\)://harbor.vlab.dustbox.kr](http(s)://harbor.vlab.dustbox.kr)

인스턴스 기반 레지스트리 서버 구성

만약, 빠르게 구성을 원하는 경우, `docker-compose`, `podman-compose`를 통해서 다음과 같이 구성이 가능하다. 파일 이름은 `cloud-init-harbor.yaml`으로 저장한다.

```
# vi cloud-init-harbor.yaml
#cloud-config
package_update: true
package_upgrade: true
packages:
  - podman
  - git
  - curl
  - wget
  - tar
  - jq
```

인스턴스 기반 레지스트리 서버 구성

위의 내용 계속 이어서...

```
write_files:
```

```
- path: /opt/install-harbor.sh
```

```
permissions: '0755'
```

```
content: |
```

```
#!/bin/bash
```

```
set -e
```

```
HARBOR_VER="v2.10.0"
```

```
cd /opt
```

```
curl -LO
```

```
https://github.com/goharbor/harbor/releases/download/\${HARBOR\_VER}/harbor-online-installer.tgz
```

인스턴스 기반 레지스트리 서버 구성

위의 내용 계속 이어서...

```
tar -zxvf harbor-online-installer.tgz
cd harbor
cp harbor.yml.tpl harbor.yml
sed -i 's/hostname: reg.mydomain.com/hostname:
harbor.vlab.dustbox.kr/' harbor.yml
sed -i 's/port: 80/port: 80/' harbor.yml
sed -i 's/https:$/# https:/g' harbor.yml # TLS는 우선 끄 (테스트 목적)
sed -i 's/certificate: /# certificate:/g' harbor.yml
sed -i 's/private_key: /# private_key:/g' harbor.yml
./install.sh

runcmd:
- [ bash, /opt/install-harbor.sh ]
```

이미지 빌드

Glance

이미지 빌드

가상머신에 사용하는 이미지는 보통, 각 회사나 커뮤니티에서 구성 후 배포한다. 하지만, 해당 이미지는 표준적인 이미지로 구성이 되어 있기 때문에, 서비스 혹은 회사에서 사용하는 용도와 맞지 않는 경우가 있다.

이러한 이유로 오픈스택 사용자는 별도로 이미지를 구성 및 배포하는 경우가 많다. 여기서, 많이 사용하는 두 가지 이미지 빌드 도구를 가지고 가상머신 이미지 구성에 대해서 다루도록 한다.

1. `diskimage-builder`

2. `oz`

3. `packer`

3번 `packer`는 오픈소스 저장소에는 등록이 안되어 있기 때문에, 3번은 제외하고 1/2번 기준으로 디스크 이미지 빌드 방식에 대해서 설명한다.

이미지 빌드(diskimage-builder)

현재 모든 가상머신 시스템은 diskimage-builder를 많이 사용한다. 구성 및 사용하기 위해서 아래와 같이 설정한다.

```
# dnf install -y python3-pip
# pip install diskimage-builder
export DIB_RELEASE=9
export DISTRO_NAME=rocky
export ELEMENTS_PATH=elements

# vi elements/rocky/root.d/50-enable-agent
#!/bin/bash
set -eux
dnf install -y qemu-guest-agent cloud-init
# disk-image-create rocky vm cloud-init -o rocky9-cloudimg
```

diskimage-builder 옵션

위에서 사용한 옵션은 다음과 같다. 배포판은 언제든지 변경이 가능하다.

여기서는 Rocky Linux 9으로 구성하였다. 오픈스택 및 대다수 IaaS를 지원하는 플랫폼들은 cloud-init를 사용하기 때문에, 부팅 시 활성화 되도록 구성한다.

옵션	설명
rocky	OS 베이스
vm	가상화용 이미지
cloud-init	cloud-init 활성화
-o	출력 파일명 지정 (rocky9-cloudimg.qcow2 생성됨)

oz로 Rocky 이미지 빌드하기

oz를 사용하는 경우, 아래와 같이 **템플릿(template)**를 생성 후, 부팅을 시작하여 이미지 빌드를 시작한다. 이 방식은 시간이 오래 걸리지만, 제일 안전하고 깔끔한 이미지를 생성 및 구성하는 장점이 있다.

```
# vi rocky.tdl
<template>
  <name>rocky9</name>
  <os>
    <name>Rocky</name>
    <version>9</version>
    <arch>x86_64</arch>
    <install type='url'>
      <url>http://mirror.rockylinux.org/rocky/9/BaseOS/x86_64/os/</url>
    </install>
```

oz로 Rocky 이미지 빌드하기

위의 내용 계속 이어서...

```
<rootpw>rockypass</rootpw>
</os>
<packages>
  <package name="cloud-init" />
  <package name="qemu-guest-agent" />
</packages>
</template>
# dnf install oz
# oz-install -d3 -t /etc/oz/templates/rocky-9.tdl
# openstack image create "rocky9-custom" \
--file rocky9-cloudimg.qcow2 --disk-format qcow2 --container-format bare \
--public
```

앤서블 예제

만약, 이미지를 자동으로 Glance에 배포하기 위해서 아래와 같이 앤서블 작성 후, Glance에 배포가 가능하다.

```
- name: Upload image to Glance
  openstack.cloud.image:
    name: rocky-9.5
    disk_format: qcow2
    container_format: bare
    filename: /root/rocky-server.qcow2
    is_public: true
    state: present
```

Cinder

블록장치

개요

OpenStack의 **Cinder**는 **블록 스토리지(Block Storage)** 서비스를 제공하는 컴포넌트로서, 가상 머신 인스턴스에 연결 가능한 영구적인 디스크 볼륨을 생성, 관리, 삭제하는 기능을 수행한다.

Cinder는 물리적인 스토리지를 추상화 하여 사용자가 논리적인 블록 장치 형태로 접근할 수 있도록 하며, 이는 전통적인 SAN, NAS 환경과 유사한 방식으로 동작한다. Nova와 긴밀하게 통합되어 인스턴스 생성 시 루트 디스크로 활용되거나, 추가 디바이스 형태로 연결할 수 있다.

사용자는 **Cinder**를 통해 **볼륨 생성, 스냅샷 생성, 볼륨 복제, 백업, 이미지**로부터 볼륨 생성 등 다양한 스토리지 기능을 수행할 수 있으며, 이러한 작업은 RESTful API 또는 CLI를 통해 요청된다.

Cinder는 다양한 스토리지 백엔드 드라이버를 지원하며, 대표적으로 **LVM, NFS, Ceph RBD, Dell EMC, NetApp** 등이 있다. 운영자는 이 백엔드 설정을 통해 단일 환경 내에서 여러 종류의 스토리지를 통합하여 운용할 수 있으며, **QoS 정책, 멀티 백엔드, 스토리지 유형(Volume Type)** 등의 고급 기능도 함께 제공된다.

개요

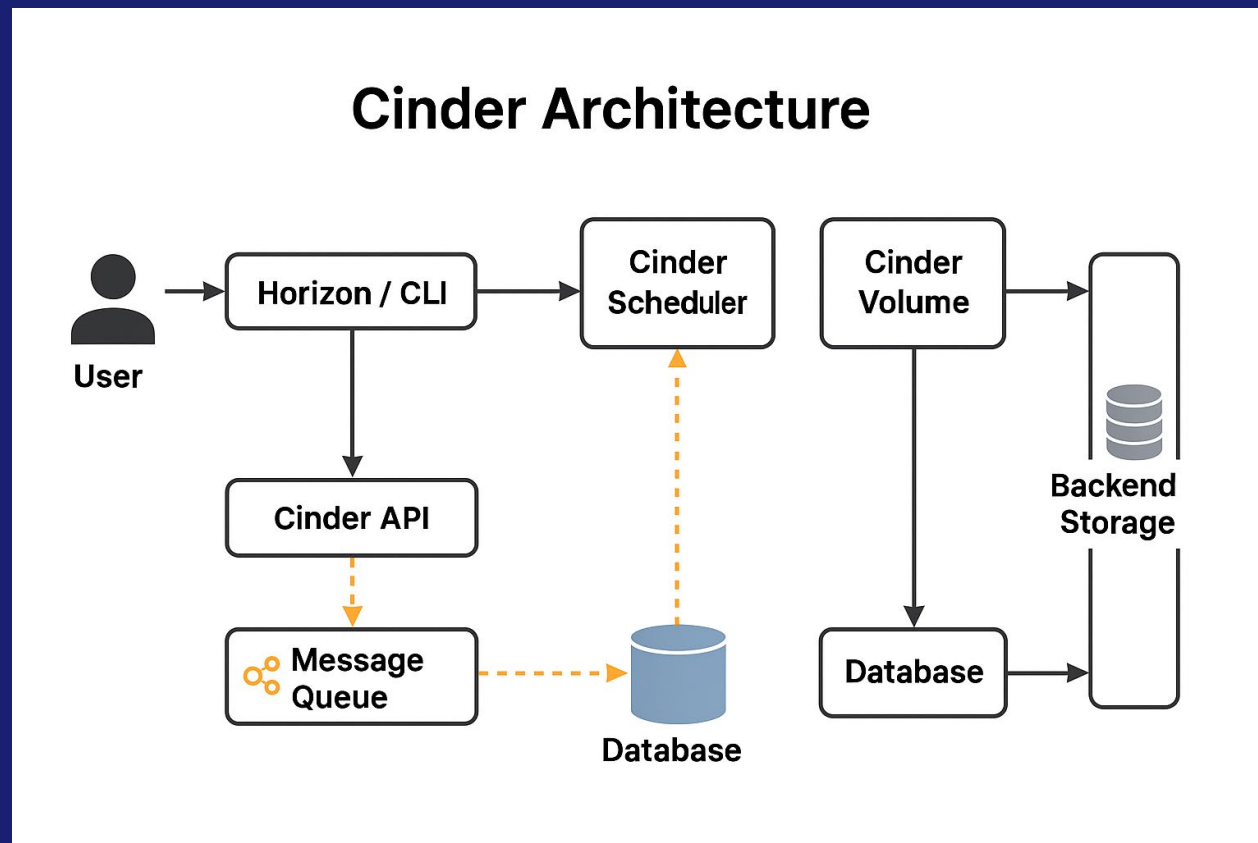
내부적으로는 **cinder-api, cinder-scheduler, cinder-volume, cinder-backup** 등의 컴포넌트로 구성되어 있으며, 이들은 OpenStack **메시지 큐(RabbitMQ)**와 **데이터베이스**를 통해 상호 통신한다.

또한 **Cinder**는 Glance와 연동되어 이미지를 기반으로 볼륨을 생성할 수 있고, 생성된 볼륨은 **Nova 인스턴스에 연결(mount)**되어 루트 디스크 또는 데이터 디스크로 활용된다.

반대로, 인스턴스에서 사용 중인 볼륨을 분리하거나 스냅샷을 생성하여 새로운 볼륨으로 복제하는 것도 가능하다

Cinder는 OpenStack 내에서 **상태를 가지는 스토리지 인프라의 핵심 구성 요소**로서,고가용성, 백업, 재해 복구, 데이터 격리를 요구하는 엔터프라이즈 환경에서도 중요한 역할을 수행한다.

아키텍처



랩 백엔드

랩에서는 Ceph/GlusterFS와 같은 백 엔드 구성 및 활용하지 않는다. 많은 자원 및 시간이 발생하기 때문에, 추후에 고급과정에서 다룬다. 랩에서는 NFS(Ganesha)기반으로 사용한다.

일반적으로 오픈소스 버전으로 오픈스택을 구성하는 경우 스토리지 백-엔드는 다음과 같이 구성한다.

종류	프로젝트명	Cinder 연동 여부	특징 및 비고
블록	LVM	공식 드라이버 있음	가장 간단한 로컬 디스크 백엔드
블록	DRBD	드라이버 있음	고가용성 구성
블록	ZFS zvol	비공식/커스텀	zvol로 블록 제공
블록	GlusterFS	공식 드라이버	Gluster 블록 또는 NFS 형태
블록	NFS (Ganesha 등)	공식 드라이버	NFS 공유 디렉터리 기반
블록	Sheepdog	비활성화	유지보수 중단 주의

랩 백엔드

아래 내용은 사용은 가능하나, 공식적인 Cinder 드라이버가 없다. 이러한 이유로 직접 인스턴스와 연결 및 구성해야 한다.

종류	프로젝트명	Cinder 연동 여부	특징 및 비고
오브젝트	Ceph (RBD)	공식 지원	가장 널리 사용됨
오브젝트	MinIO	커스텀 드라이버 가능	S3 API 기반, 중간 계층 필요
오브젝트	OpenIO, Swift, Walrus		Glance에서 Swift 사용 가능
파일	MooseFS	NFS 통해 간접 연동 가능	FUSE 또는 NFS를 통해 접근 가능
파일	Lustre, BeeGFS	-	HPC 환경에 주로 사용, 직접 연동 없음

볼륨 생성

가상머신 이미지를 부팅 가능한 볼륨 형태로 구성이 필요한 경우, 아래와 같이 실행한다. 이 방식을 사용하면 부팅 속도 및 고성능 IOPS를 보장하며, 또한 컴퓨트 노드의 워크로드를 줄여 주기도 한다.

```
# openstack volume create --size 10 --image ubuntu-22.04 --bootable ubuntu-volume
# openstack volume show my-volume
# openstack server add volume my-instance my-volume
# openstack server remove volume my-instance my-volume
# openstack volume snapshot create --volume my-volume snapshot-001
# openstack volume create --snapshot snapshot-001 --size 10 cloned-volume
# openstack volume delete my-volume
```

볼륨 암호화

여기서는 볼륨 형식을 추가한다. 다만, 추가 시, 디스크 Lukus암호화를 활성화 한다.

```
# openstack volume type create ssd-type
# openstack volume type set --property volume_backend_name=ssd ssd-type
# openstack volume type create secure-type
# openstack volume type encryption create --cipher aes-xts-plain64 \
  --control-location front-end --provider luks --key-size 256 secure-type
# openstack server show my-instance -f json | jq '.os-extended-
volumes:volumes_attached'
```

백엔드 추가

Cinder

BACKEND 추가

더 많은 백-엔드를 추가하면 좋겠지만, 리소스 문제로 간단하게 LVM2의 백-엔드를 구성한다. 백-엔드는 storage 서버에 구성한다. 여기서는 storage instance에 구성된 /dev/vdb1를 사용하여 LVM2를 구성한다.

만약, 블록 장치가 구성이 안된 사용자는 storage서버에서 아래와 같이 명령어를 실행한다.

```
# lsblk  
# pvcreate /dev/vdb1  
# vgcreate cinder-volumes /dev/vdb1
```

BACKEND CONFIG

`cinder.conf`에 다음과 같이 설정한다.

설정 파일 위치는 deployment에 `/etc/kolla/config/cinder.conf`에 아래와 같이 작성한다.

```
[DEFAULT]
enabled_backends = nfs,lvm

[lvm]
volume_driver = cinder.volume.drivers.lvm.LVMVolumeDriver
volume_backend_name = LVM
volume_group = cinder-volumes
iscsi_protocol = iscsi
iscsi_helper = lioadm
```

BACKEND CONFIG

다시, cinder container에 설정 파일 적용 및 재시작을 수행한다.

```
# kolla-ansible reconfigure -t cinder -i multimode
```

볼륨 타입 생성

아래 명령어로 오픈스택에서 사용이 가능하도록 볼륨 타입을 생성한다.

```
# openstack volume type create lvm-type  
# openstack volume type set --property volume_backend_name=LVM lvm-type
```

생성이 완료가 되면, 다음처럼 전체적으로 자원을 확인한다.

```
# vgs  
# lvs  
# openstack volume create --type lvm-type --size 1 test-volume
```

앤서블

앤서블 기반으로 볼륨 디스크를 생성 시 다음과 같이 작성 및 구성한다.

```
- name: Create and attach volume in OpenStack
  hosts: localhost
  tasks:
    - name: Create volume
      openstack.cloud.volume:
        state: present
        display_name: my-volume
        size: 10
```

앤서블

위의 내용 계속 이어서...

```
--+
```

```
- name: Attach volume to instance
  openstack.cloud.server_volume:
    server: my-instance
    volume: my-volume
    state: present
```

Neutron

네트워크

설명

Neutron은 OpenStack 환경에서 가상 네트워크를 생성하고 관리하는 네트워크 서비스 컴포넌트이다. 가상 머신 (VM) 간의 통신, 외부 네트워크와의 연결, 보안 정책 적용 등 네트워크 관련 기능을 담당한다.

이 서비스는 주로 API를 통해서 작업 요청을 받아서, 리눅스 커널 기반으로 자원들을 구성한다. 또한, Neutron서비스는 직접적으로 네트워크 구성에 관여하지 않는다.

네트워크 경우에는 리눅스 브릿지 기반으로 다음과 같은 네트워크 터널링 자원을 제공한다.

- VXLAN
- Geneve
- VLAN

대표적으로 위 3개의 네트워크 터널링을 통해서 컴퓨트 네트워크를 구성한다. 이전 오픈스택은 리눅스 브릿지 기반으로 위 자원을 통해서 터널링을 구성 하였지만, 현재는 리눅스 브릿지 사용을 권장하지 않는다.

설명

현재 오픈스택은 리눅스 브릿지 대신 다음과 같은 브릿지 및 네트워크 기능을 제공한다. 하지만, 리눅스 브릿지는 커널 영역 및 사용자 영역 연결 시 여전히 사용하고 있다.

- OVS
- OVN

이를 통해서 ML2, ML3모듈을 제공하며, ML2 모듈을 통해서 OVS, Linux Bridge, SR-IOV, OVN과 같은 도구를 제공하고 있다. OVS는 기존에 사용하던 리눅스 브릿지 기능을 제공해주며, OVN은 netfilter로 구성한 방화벽/보안그룹/DHCP/Network/Network Subnet 기능을 제공하고 있다.

OVN은 현재 선택사항이기 때문에, OVN를 사용하지 않는 경우 DNSMASQ/NFTABLES를 통해서 보안그룹 및 가상 네트워크 및 DHCP를 제공한다.

OVS (Open vSwitch)

OVS는 오픈소스 가상 스위치이다.

→ Open vSwitch는 리눅스 및 기타 플랫폼에서 사용 가능한 고성능 가상 스위치로, 가상화된 환경에서 네트워크 트래픽을 제어하기 위해 설계되었다.

OVS는 커널 모듈과 유저 스페이스 데몬으로 구성된다.

→ OVS는 고속 패킷 처리를 위해 커널 모듈을 사용하고, 복잡한 흐름 제어와 관리 작업은 ovs-vswitchd라는 유저 공간 데몬에서 수행한다.

OVS는 OpenFlow를 지원한다.

→ SDN(Software-Defined Networking)을 구현할 수 있도록 OpenFlow 프로토콜을 완전하게 지원하며, 컨트롤러 기반의 네트워크 제어가 가능하다.

OVS (Open vSwitch)

OVS는 VLAN, GRE, VXLAN 등 다양한 터널링을 제공한다.

→ OVS는 VLAN뿐 아니라 오버레이 네트워크 구성에 필요한 GRE, VXLAN과 같은 터널링을 기본적으로 지원하여, 멀티 테넌시 환경에 적합하다.

OVS는 OpenStack, Kubernetes 등 다양한 플랫폼과 통합된다.

→ OVS는 Neutron의 ML2 OVS 드라이버와 잘 연동되며, Kubernetes CNI 플러그인과도 함께 사용할 수 있다.

OVN

OVN은 OVS 위에서 동작하는 가상 네트워크 컨트롤러이다.

→ OVN은 Open vSwitch를 기반으로 한 분산 네트워크 가상화 시스템으로, 논리적 스위치/라우터를 정의하고 자동으로 데이터 플레인에 반영한다.

OVN은 Northbound/Southbound DB를 사용한다.

→ 사용자가 생성한 논리적 네트워크 설정은 Northbound DB에 저장되며, 이를 OVN 컨트롤러가 Southbound DB로 변환하여 각 노드에 전파한다.

OVN

OVN은 OVS를 데이터 플레인으로 사용한다.

→ OVN 자체는 컨트롤 플레인이며, 실제 패킷 처리는 OVS를 통해 수행된다. 이 구조는 컨트롤과 데이터의 분리를 실현하여 관리 효율성을 높인다.

OVN은 논리적인 라우팅과 ACL, DHCP, NAT 기능을 제공한다.

→ 전통적인 L2 브리징 뿐만 아니라, L3 라우팅, 보안 그룹(ACL), 가상 DHCP 서버, NAT 등을 포함한 복합 기능을 제공한다.

OVN은 오버레이 네트워크를 자동화하고 분산 처리한다.

→ VXLAN 기반의 네트워크 오버레이를 자동으로 생성하며, 각 하이퍼바이저 노드에서 분산 방식으로 설정을 적용하여 중앙 집중화의 병목을 줄인다.

OVS/OVN 비교

OVS/OVN기능을 비교하면 다음과 같다. OVS는 컨트롤 플레인, OVN은 데이터 플레인을 지원 및 관리한다.

항목	OVS (Open vSwitch)	OVN (Open Virtual Network)
역할	가상 스위치 (Layer 2 중심)	가상 네트워크 컨트롤러 (Layer 2+3 포함)
구성	커널/유저 공간에서 스위칭 동작	OVS 위에 구성된 오버레이 네트워크 제어 시스템
기능	브리지, VLAN, STP 등 제공	논리적 스위치/라우터, ACL, DHCP, NAT, BGP 지원
아키텍처	분산 구조, 직접 컨트롤러 없음	중앙 OVN 컨트롤러를 통해 상태 관리 및 분산 흐름 설정
연동	Neutron ML2/OVS 드라이버 사용	Neutron ML2/OVN 드라이버 사용
통신 방식	br-int, br-ex, br-tun 등 브리지 사용	OVSDB, NB/SB DB 기반 논리적 구성 동기화

OVS/OVN 단점

이를 사용하면, 안정적으로 시스템을 운영이 가능하다는 장점이 있지만, 다음과 같은 단점도 존재한다.

항목	한계점 설명
복잡한 디버깅	OVN은 Northbound/Southbound DB 기반이라 네트워크 오류 추적이 어렵다
확장성	수백 노드 이상에서는 SB/DB의 확장성과 Latency 문제가 나타날 수 있다
기능 제한	EVPN, BGP-based routing 등 일부 고급 네트워크 기능은 미흡하거나 실험적이다
커뮤니티 및 문서 부족	OVN은 OVS보다 늦게 등장해서 사례나 문서가 비교적 적은 편이다
인터페이스 구성 복잡성	OVN은 논리적 요소를 물리적 인터페이스와 연결할 때 구조가 복잡할 수 있다

OVS/LINUX BRIDGE 비교

앞서 이야기 하였지만, OVS 이전에는 LINUX BRIDGE를 사용하였다. 현재는 사용하지 않지만, LINUX BRIDGE를 사용하였을 때, 다음과 같은 장단점이 있다.

항목	Linux Bridge	Open vSwitch (OVS)
목적	리눅스 기본 브리지, 단순 가상 네트워크 구성	고성능, SDN을 위한 가상 스위치
커널 통합	커널 네이티브 브리지	커널 모듈 + 유저 공간 데몬(ovs-vswitchd)로 구성
기능	브리지, VLAN tagging 등 기본 기능 제공	VLAN, GRE/VXLAN, OpenFlow, QoS, ACL 등 고급 기능 제공
성능	기본적인 성능 제공	멀티큐 NIC, DPDK, flow 기반 최적화로 고성능 가능
관리 도구	brctl, ip link, nmcli, nm-connection-editor	ovs-vsctl, ovs-ofctl, ovn-* 도구 사용

OVS/LINUX BRIDGE 비교

위의 내용 계속 이어서...

항목	Linux Bridge	Open vSwitch (OVS)
확장성	작은 규모에 적합	대규모 클라우드 환경에 적합
OpenFlow 지원	지원하지 않음	OpenFlow 완전 지원
VXLAN, GRE 지원	일부 배포판에서 커널 모듈로 제한적 지원	완전 지원 (overlay 네트워크에 최적화)
OpenStack 연동	ML2 LinuxBridge 드라이버 사용	ML2 OVS or OVN 드라이버 사용
보안 그룹	iptables 기반 구현	OpenFlow 기반 필터링, 빠름

네트워크 자원생성

Neutron

네트워크 생성

오픈스택에서 제공하는 네트워크 유형은 보통 다음과 같다.

1. vlan
2. vxlan
3. Geneve
4. Flat

네트워크는 기본적으로 L2를 구성 및 생성한다. 각 네트워크는 프로젝트별로 독립적으로 구성이 되기 때문에 사용자가 만든 네트워크는 다른 사용자가 접근이 불가능하다. 또한, L2가 구성이 되어 있지 않으면, L3 구성하지 못한다.

L2와 함께 구성이 되는 자원은 보안그룹(SecurityGroup), 포트(Port)가 구성이 된다. 구성된 포트 및 보안그룹은 리눅스 브릿지 및 OVS를 통해서 생성 및 구성이 된다.

서브넷 생성

오픈스택에서 서브넷은 네트워크 유형과 상관없이, 네트워크를 사용하기 위해서 최소 한 개의 서브넷이 필요하다. 서브넷은 직접적인 아이피를 가지고 있지는 않고, L3를 구성, 이를 통해서 DHCP 혹은 Static 네트워크를 구성한다.

보통 서브넷에 구성되는 자원은 다음과 같다.

1. Subnet(IPV4/IPV6)
2. DHCP Range
3. Gateway IP
4. Cloud-init

이 구성은 OVS를 사용하는 경우, 보통 dnsmasq, network namespace, netfilter를 사용해서 구성하며, OVN를 사용하는 경우, 내부에서 router, network, subnet, security group등과 같은 자원이 생성이 된다.

네트워크/서브넷 관계

네트워크 및 서브넷 관계를 정리하면 다음과 같다.

구성 요소	설명
네트워크	L2 계층 (MAC 기반 연결) 정의, 브리지 역할
서브넷	L3 계층 (IP 주소 범위) 정의, 라우팅과 DHCP 제공
포트	네트워크 + 서브넷에 연결되는 구체적인 인터페이스 (인스턴스의 NIC)

네트워크 생성

네트워크 생성 명령어는 다음과 같다.

```
# openstack network create my-net
--share          # 공유 네트워크
--provider-network-type vxlan|vlan|flat
--external       # 외부 네트워크로 표시 (floating IP용)
```

share: 생성된 네트워크를 모든 사용자에게 공유한다. 관리자만 사용이 가능.

provider-network-type: 구성 시 사용할 네트워크 형식을 선택한다. 기본값은 flat.

external: 외부 네트워크와 연결 및 구성이 필요한 경우.

포트 생성

포트는 아이피 생성과 동일하다. 포트가 생성이 되면 OVS에서 브릿지 포트가 생성이 된다.

```
# openstack port create \  
  --network my-net \  
  --fixed-ip subnet=my-subnet,ip-address=192.168.100.10 \  
  my-port
```

fixed-ip: dhcp가 아닌, static으로 할당을 원하는 경우

subnet: 포트를 생성할 서브넷 및 아이피 주소

네트워크 생성 명령어

라우터는 아래 명령어로 생성한다. OVS를 사용하는 경우, namespace 기반으로 구성이 된다.

```
# openstack router create my-router
```

라우터 서브넷 추가

한 개 이상의 네트워크가 서로 통신이 필요한 경우, 라우터를 통해서 라우팅이 필요하다.

```
# 서브넷 연결 (인터페이스 추가)
```

```
# openstack router add subnet my-router my-subnet
```

라우터에 직접 네트워크 연결이 불가능 하다. 서브넷을 통해서 라우터 구성 및 라우팅이 가능하다.

외부 라우터

트래픽을 외부로 전달이 필요한 경우, 아래처럼 생성 및 구성해야 한다. 이 기능은 OVN 혹은 Netfilter에서 구성이 된다.

```
# 외부 네트워크 설정
```

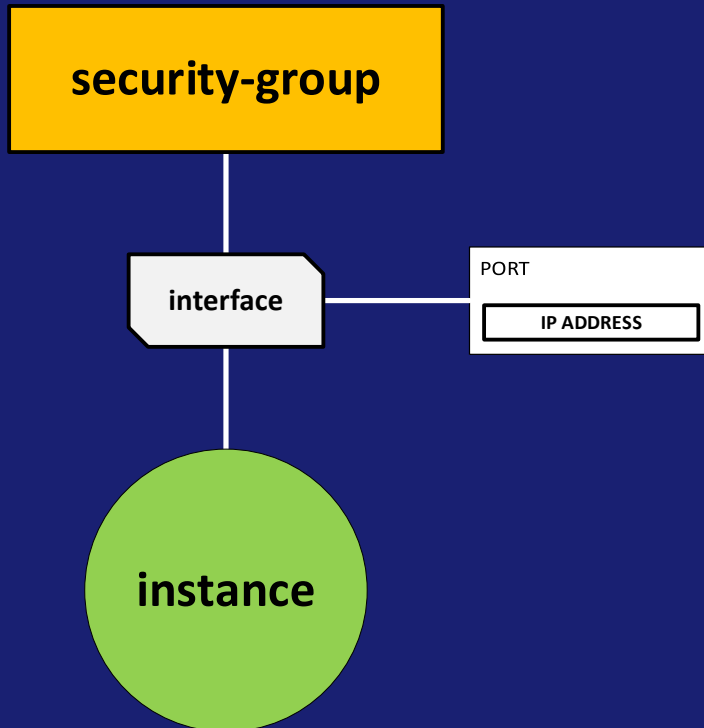
```
# openstack router set --external-gateway public-net my-router
```

external-gateway: 외부로 트래픽이 나가는 경우, 반드시 외부 게이트웨이 옵션이 활성화가 되어 있어야 한다.

보안그룹

네트워크 보안 그룹은 OVN이나 혹은 Netfilter에서 구성이 된다. 보안 그룹 생성 후, 정책을 구성한다.

```
# openstack security group create my-secgroup
```



보안그룹 정책 추가

보안그룹에 정책 추가가 가능하다. 보안 그룹은 다음과 같은 프로토콜에 적용이 가능하다.

1. TCP
2. UDP
3. ICMP

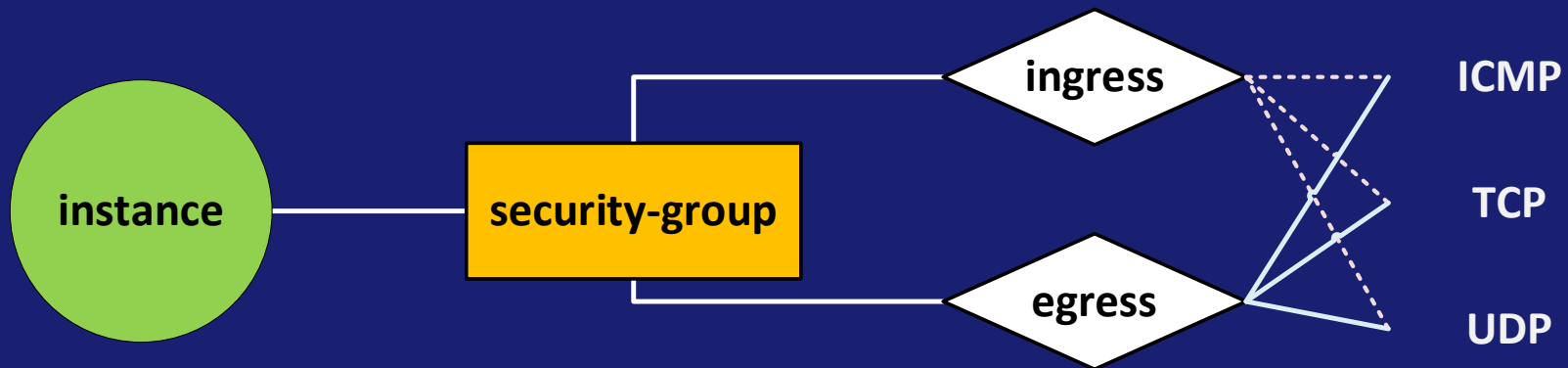
```
# SSH 허용
# openstack security group rule create \
  --protocol tcp --dst-port 22 \
  --ingress --ethertype IPv4 \
  my-secgroup
```

보안그룹 정책 추가

보안 그룹 정책은 두 가지 영역으로 분리 및 구성이 가능하다.

1. Ingress
2. Egress

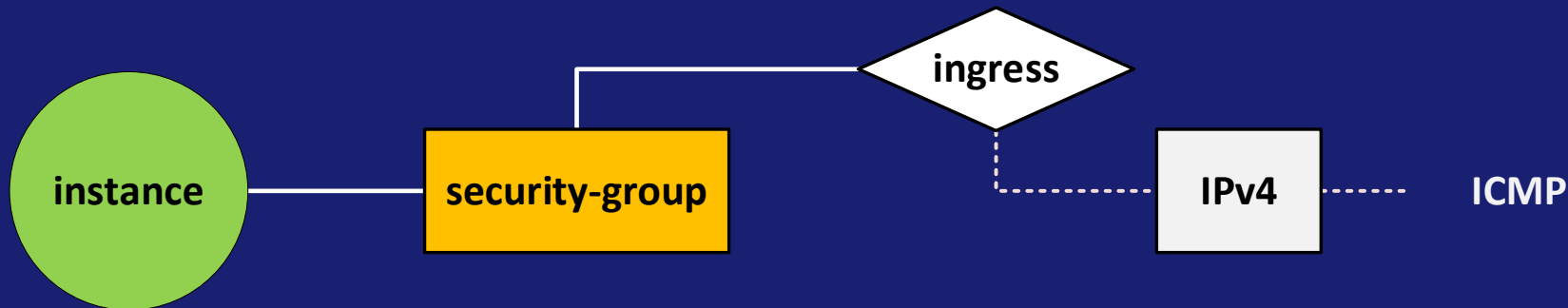
외부에서 들어온 자원은 Ingress, 내부에서 외부로 나가는 자원은 Egress라고 부른다.



보안그룹 정책 추가

만약, ICMP에서 특정 트래픽만 허용이 필요한 경우, 아래처럼 설정이 가능하다.

```
# Ping 허용
# openstack security group rule create \
  --protocol icmp \
  --ingress --ethertype IPv4 \
  my-secgroup
```

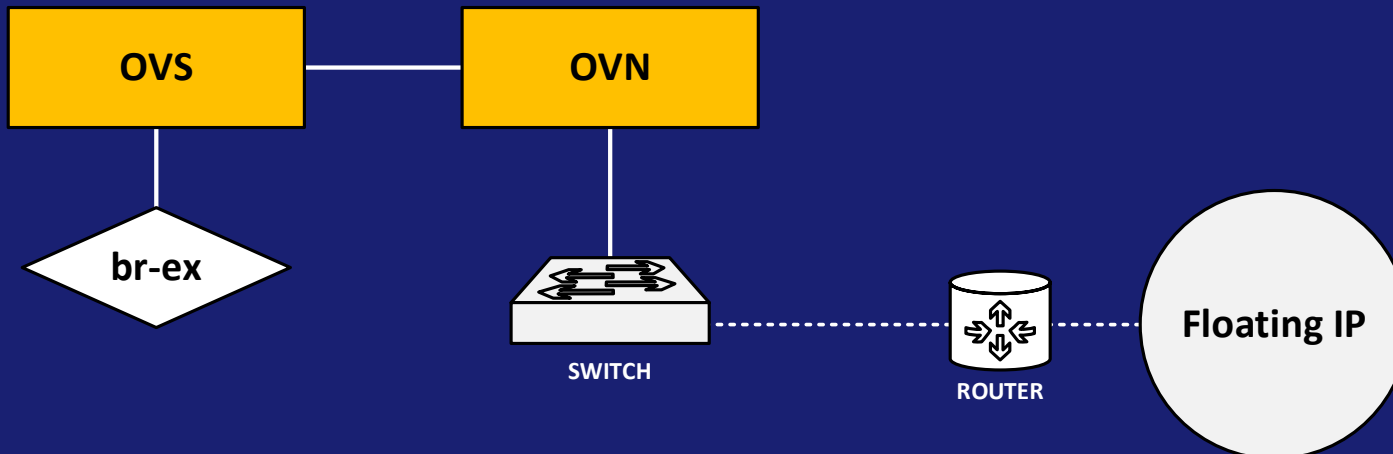


외부 아이피

외부 아이피는 외부와 연결된 네트워크 설정을 통해서 구성이 된다.

```
# openstack floating ip create public-net
```

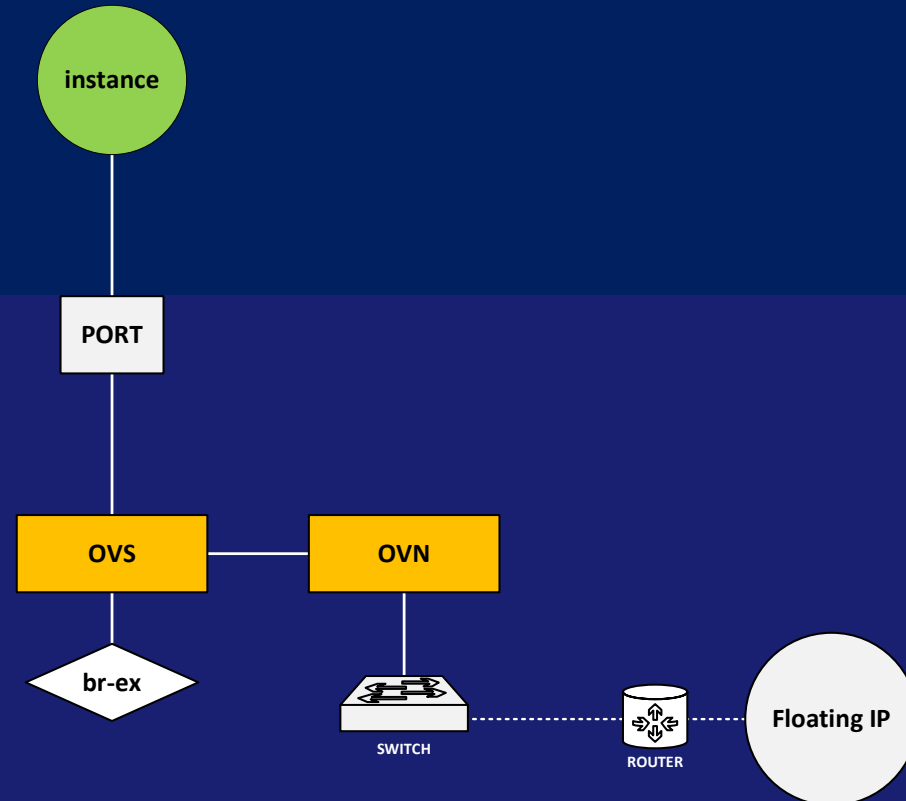
외부 네트워크는 최소 한 개의 physnetX와 연결 및 구성이 되어 있어야, 해당 물리적 인터페이스를 통해서 할당 및 구성이 가능하다.



외부 아이피

Floating IP는 보통 무작위로 할당이 되기 때문에, 특정 아이피를 사용하기 위해서, 아래처럼 Port로 구성 후 인스턴스에 할당한다.

```
# 포트 또는 인스턴스에 할당  
# openstack floating ip set \  
  --port my-port \  
  <FLOATING_IP_ID 또는 IP>
```



QOS 제한

QOS를 구성하여, 특정 인스턴스의 트래픽 제한이 가능하다.

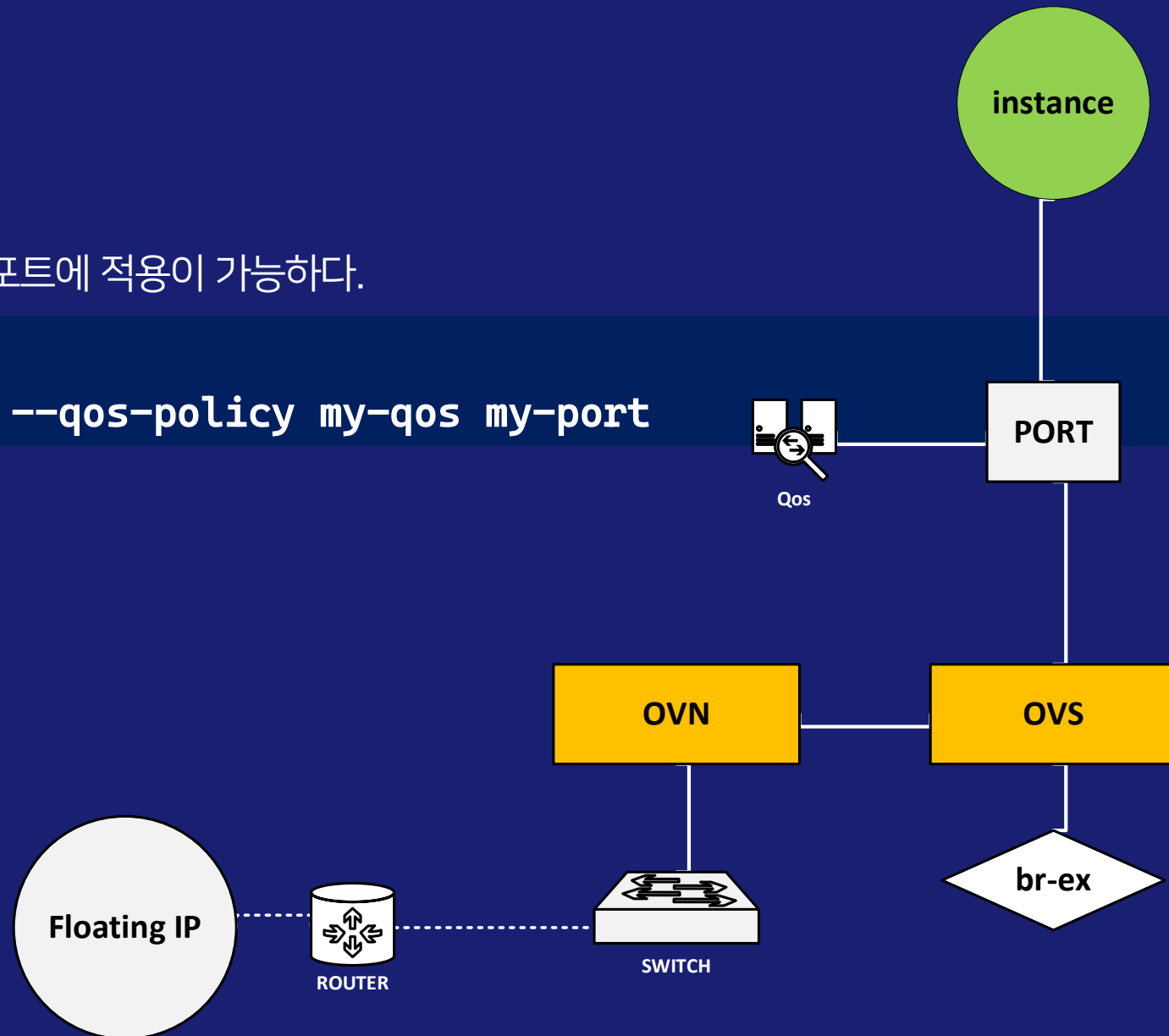
```
# openstack qos policy create my-qos
# openstack qos bandwidth-limit-rule create \
  --max-kbps 1000 --max-burst-kbps 500 \
  my-qos
```

QOS 적용

생성된 QOS를 특정 인스턴스 포트에 적용이 가능하다.

포트에 정책 적용

`openstack port set --qos-policy my-qos my-port`



트렁크 부모 포트 생성

트렁크 포트는 1개 이상의 포트를 묶어서 사용하는 경우 사용한다. 생성 시, 먼저 부모 인터페이스(포트)를 생성한다.

```
# 트렁크의 parent port 먼저 생성
```

```
# openstack port create --network my-net parent-port
```

트렁크 자식 포트 생성

부모 포트에서 사용 할 자식 포트 두 개를 생성한다. 생성 후, 부모에게 할당한다.

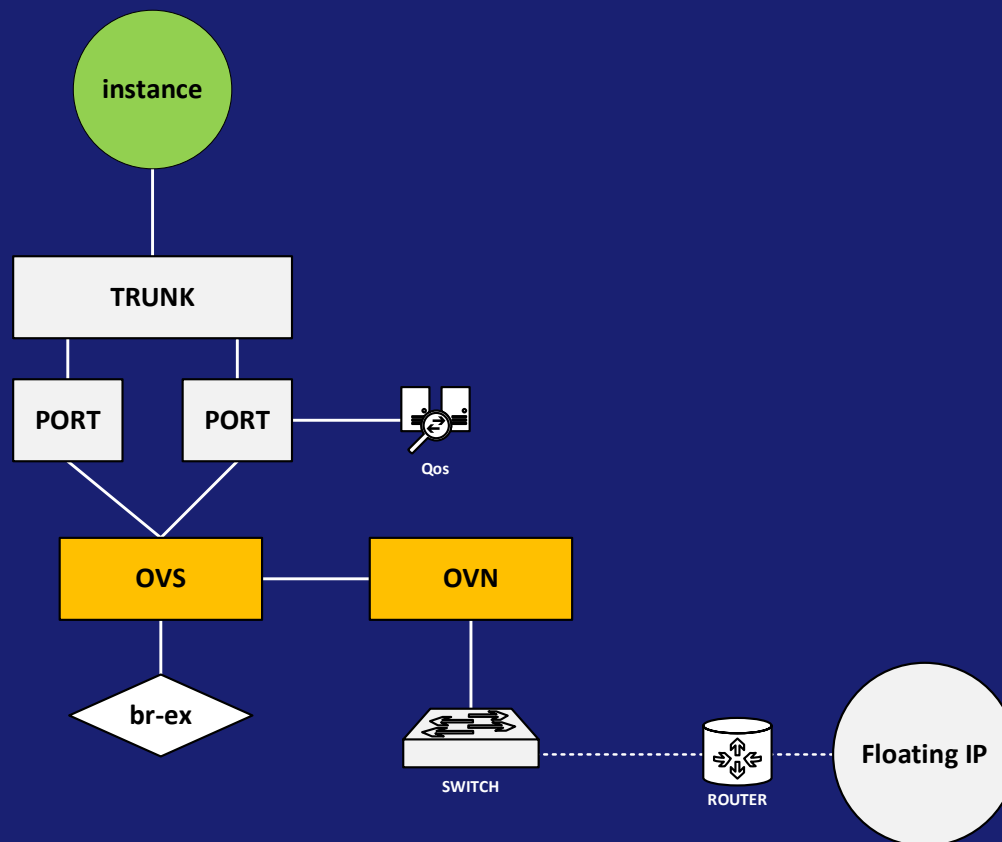
```
# 서브포트용 포트 생성  
# openstack port create --network my-net subport1  
# openstack port create --network my-net subport2
```

최종 트렁크 포트

최종적으로 부모 포트에 두 개의 인터페이스를 연결한다. 세그멘테이션은 인터페이스의 연결 유형이다.

```
# trunk 생성
# openstack network trunk create my-trunk \
  --parent-port parent-port \
  --subport port=subport1,segmentation-type=vlan,segmentation-id=101 \
  --subport port=subport2,segmentation-type=vlan,segmentation-id=102
```

최종 트렁크 포트



방화벽 VS 보안그룹

방화벽과 보안그룹의 차이점은 다음과 같다.

항목	Security Group	FWaaS (v1/v2)
적용 대상	인스턴스(VM)의 포트	라우터(L3)
적용 위치	L2/L3 계층의 인스턴스 입출구	라우터를 통한 트래픽 입출구
기반 기술	iptables 또는 OVS 기반 보안 그룹	L3 Agent에 iptables 정책 추가
트래픽 방향	VM ↔ 네트워크	외부 ↔ 내부 네트워크
기본 동작	기본적으로 'DENY ALL', 허용만 명시	기본적으로 'ALLOW ALL', 차단만 명시 가능 (v1 기준)
정책 단위	포트 또는 인스턴스	네트워크 또는 라우터 단위
세분화	세밀한 포트, CIDR, 프로토콜 설정	정책 기반으로 한 묶음 룰 처리
다중 정책 지원	1개의 security group set per port	FWaaS v2에서는 다중 정책 가능

방화벽 VS 보안그룹

위의 내용 계속 이어서...

항목	Security Group	FWaaS (v1/v2)
상태 추적	상태 추적 가능 (stateful)	상태 추적 가능 (v2 기준)
삭제 예정 여부	계속 유지	Deprecated & Removed (Wallaby~Xena)
사용 예시	VM 간 접근 제한, SSH/HTTP 제한 등	외부 공격 필터링, 내부망 접근 제어 등

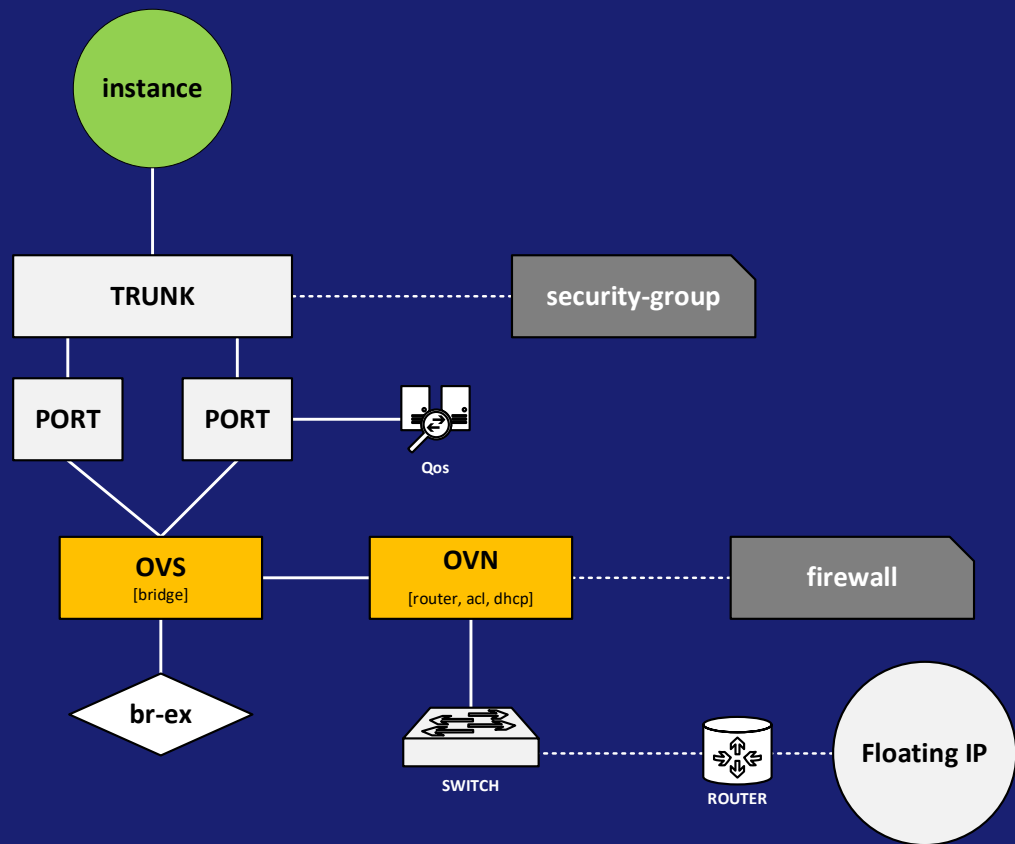
방화벽 VS 보안그룹

- Security Group: VM 단위, 내부 보안 정책. 접근 제어를 인스턴스 단위로 세밀하게 하고 싶을 때.
- FWaaS: 네트워크 경계에서의 방화벽. 트래픽 흐름 전체를 정책 단위로 제어하고 싶을 때 (과거 사용).

[외부] —▶ [Router (FWaaS 정책 적용)] —▶ [VM (Security Group 적용)]

- FWaaS: 외부에서 오는 모든 포트 차단, 10.0.0.0/24만 허용
- Security Group: VM의 포트 22, 80만 허용, 같은 네트워크의 VM만 접속 허용

방화벽/보안그룹



방화벽 생성

보안그룹은 L2에서 생성이 된다. 만약, L3수준에서 방화벽이 필요한 경우 firewalld 서비스를 통해서 구성 및 생성이 가능하다. 아직 이 서비스는 유지가 되고 있지만, 아쉽게도 이걸 유지할 인력도 부족하며 또한, 다른 솔루션을 통해서 구현을 지원하고 있다.

정책과 규칙 생성

```
# openstack firewall group create my-fw --ingress-firewall-policy my-policy
```

버전	설명	상태
FWaaS v1	L3-agent 기반, 룰을 router에 직접 적용	Deprecated (Queens) / Removed (Ussuri)
FWaaS v2	Neutron 서비스 플러그인, 다중 정책/룰 지원	Deprecated (Wallaby) / Removed (Xena)

방화벽 왜 더 이상 사용하지 않는가?

1. 보안 그룹(Security Groups)이 더 광범위하고 유연하게 사용됨
2. FWaaS는 L3에 의존적인 구조로 복잡성과 확장성에 제약이 있었음
3. Neutron의 로드맵 방향이 보안 그룹, BGP, DVR 등과 통합된 네트워크 모델로 전환되었기 때문
4. OpenStack 커뮤니티의 유지 보수 인력 부족도 큰 이유
5. 현재 OVN으로 통합, OVN기반으로 사용을 매우 권장

L3 방화벽 대안 기술

L3 방화벽은 다음과 같은 기술로 대체를 권장하고 있다.

대안 기술	설명
Security Groups	L2/L3 계층에서의 인스턴스 단위 방화벽 제어
OVN ACL (Open Virtual Network)	OVN 네트워크 사용 시 ACL 기반 고급 방화벽 정책
3rd-party 솔루션	Fortinet, Palo Alto, Juniper, A10 등과 연동
Kubernetes 기반 네트워크 정책	OpenStack 위의 K8s에선 Calico/OVN/Cilium 등 가능

방화벽 생성 및 정책 생성

FWaaS에 방화벽 적용하기 위해서 반드시 최소 한 개의 컨테이너 자원이 필요하다. 여기서 말하는 컨테이너는 애플리케이션을 위한 컨테이너가 아닌, 자원을 위한 컨테이너 개념이다.

```
# openstack firewall policy create my-policy
# openstack firewall group create my-fw --ingress-firewall-policy my-policy
# openstack firewall policy add rule my-policy allow-http
```

대안 기술	설명
firewall policy create	컨테이너 생성
firewall group create my-fw	컨테이너 밑에 그룹을 생성한다
--ingress-firewall-policy	들어오는 트래픽에 대한 정책을 구성
policy add rule	allow-http 규칙을 추가한다. 80에 대해서 허용한다

방화벽 규칙 추가

아래와 같이 방화벽에 규칙 추가가 가능하다.

```
# openstack firewall rule create --protocol tcp --destination-port 80 \
  --action allow allow-http
```

옵션	의미
openstack firewall rule create	새로운 Firewall Rule 객체 생성
--protocol tcp	TCP 프로토콜에 대한 규칙임을 지정
--destination-port 80	목적지 포트 80 (HTTP) 에 대해 적용
--action allow	해당 조건에 일치하는 트래픽을 허용(allow)
allow-http	이 룰의 이름 (관리자 식별용, 자유롭게 지정 가능)

전체 구성 흐름 요약 (FWaaS v2)

다음과 같이 트래픽이 흘러가면서 방화벽에 적용된다.

```
[ Firewall Rule ] → [ Firewall Policy ] → [ Firewall Group ] → [ Neutron Port ]  
allow-http       web-policy           web-fw-group       (ex: VM NIC)
```

특정 포트에 방화벽 적용

Ingress 서비스만 허용하는 경우, 아래와 같이 구성이 가능하다.

```
# 포트에 FW 적용
# openstack firewall group set --ingress-firewall-policy my-policy --port my-port my-fw
```

요소	설명
openstack firewall group set	기존에 만들어진 Firewall Group을 설정(수정)함
--ingress-firewall-policy	인그레스(들어오는 트래픽) 정책으로 my-policy를 설정
--port my-port	해당 방화벽 그룹에 Neutron 포트(예: VM NIC 포트 등)를 연결
my-fw	설정할 대상 Firewall Group의 이름 또는 ID

보안 그룹 적용

여러 개의 아이피를 논리적으로 묶어서 적용한다.

```
# openstack address group create my-ag --address 10.0.0.5/32 --address 10.0.0.10/32
```

요소	의미
openstack address group create	주소 그룹(Address Group)을 생성하는 명령
my-ag	주소 그룹의 이름
--address 10.0.0.5/32	주소 그룹에 포함될 첫 번째 IP 주소
--address 10.0.0.10/32	주소 그룹에 포함될 두 번째 IP 주소

랩

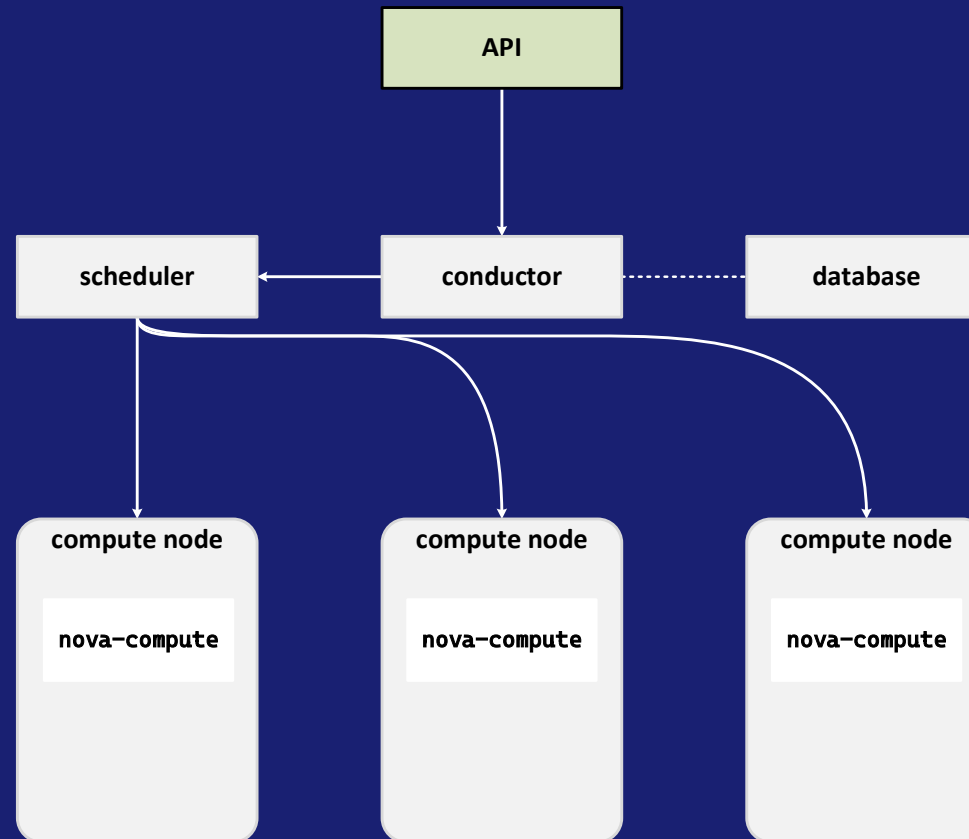
연습문제

2025-06-06

Nova

Computing

노바 아키텍처



개요

오픈스택에서 컴퓨팅 서비스는 매우 중요하다. 이 서비스를 통해서 가상머신, 즉 인스턴스를 생성 및 관리한다. 컴퓨팅 서비스 이름은 Nova라고 부르며, 초창기 오픈스택 서비스는 Nova만 존재 하였다.

Nova서비스는 기능이 확장이 되면서 분리가 되었는데, 대표적인 `nova-conductor`, `nova-compute` 그리고 `neutron`, `cinder`와 같은 서비스이다.

`nova-compute` 서비스는 직접적으로 가상머신을 생성하지 않으며, `libvirt`를 통해서 여러 하이퍼바이저 드라이버로 연결 후, `nova-compute`를 통해서 전달이 된 작업 요청 API를 통해서 작업을 수행한다.

libvirtd의 관계는 아래 슬라이드에서 좀 더 자세히 설명한다.

개요

`nova-compute`는 **OpenStack Nova**의 핵심 컴포넌트로, 가상 머신 인스턴스를 생성, 삭제, 정지 등의 작업을 수행하는 역할을 담당한다. 하지만 `nova-compute` 자체는 하이퍼바이저와 직접적으로 상호작용하지 않고, 그 중간 역할을 수행하는 추상화 계층을 통해 하이퍼바이저를 제어한다.

이때 대표적으로 사용되는 추상화 계층이 바로 `libvirt`이다.

`libvirt`는 `KVM`, `QEMU`, `Xen` 등의 다양한 하이퍼바이저를 통합적으로 제어할 수 있는 인터페이스를 제공하며, `nova-compute`는 `libvirt` 드라이버를 통해 하이퍼바이저에게 가상머신 생성 요청을 하거나 상태를 조회하는 등의 명령을 전달한다.

`nova-compute`는 `libvirt`를 통해 `KVM`과 같은 하이퍼바이저를 간접적으로 제어하며, 이 구조는 Nova의 유연성과 하이퍼바이저 독립성을 보장한다.

정리

최종적으로 Nova 서비스의 흐름은 다음과 같다.

1. 사용자가 Horizon 또는 CLI를 통해 인스턴스 생성을 요청
2. nova-api → nova-scheduler → 적절한 nova-compute로 전달
3. nova-compute는 libvirt 드라이버를 호출
4. libvirt는 KVM에 명령 전달 → VM 생성
5. VM의 상태는 다시 libvirt → nova-compute → nova-api로 보고됨

서비스 설명

기본 컴포넌트

컴퓨터 서비스 컴포넌트

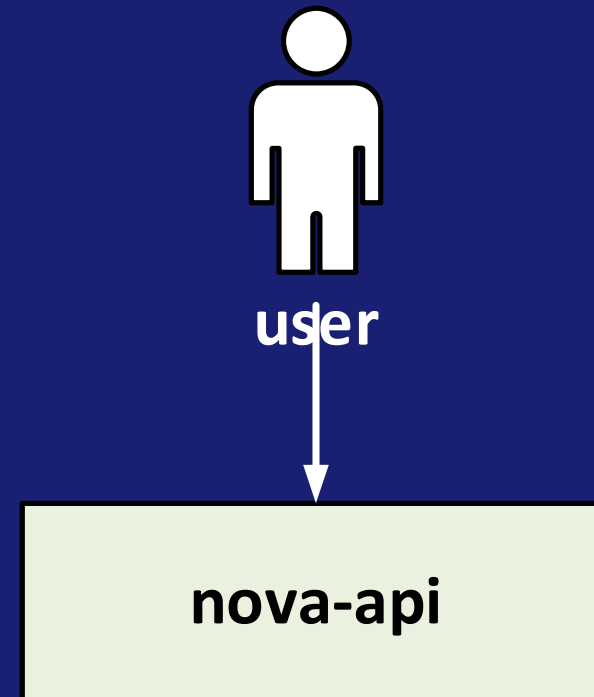
컴퓨터 서비스에서 제공하는 컴포넌트는 다음과 같다.

컴포넌트	설명	주요 역할
nova-api	RESTful API를 통해 사용자 요청을 수신	사용자 요청 처리 및 메시지 전송
nova-scheduler	요청된 인스턴스를 적절한 컴퓨터 노드에 스케줄링	VM 스케줄링 결정
nova-conductor	compute와 DB 사이의 간접 통신 수행	보안 중재자 역할
nova-compute	하이퍼바이저와 통신하여 인스턴스를 직접 생성/관리	VM 생성 및 관리
nova-placement	리소스 가용성과 정책 기반으로 배치 최적화 수행	자원 배치 결정을 위한 정보 제공
nova-consoleauth	콘솔 접근에 대한 인증 수행	VNC 콘솔 인증
nova-novncproxy	VNC 콘솔을 웹으로 제공	VNC 중계
nova-database	모든 메타데이터 및 상태 정보 저장	중앙 상태 저장소 역할

nova-api

사용자의 인스턴스 생성, 삭제, 조회 요청을 **OpenStack REST API**로 수신하고, 내부 메시지 큐 또는 데이터베이스를 통해 다른 Nova 서비스로 요청을 전달한다.

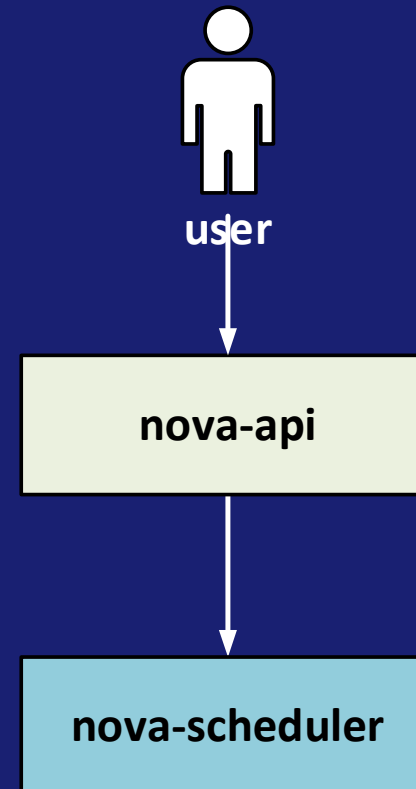
- API Gateway 역할
- 인증 토큰 처리 (keystone 연동)
- 비동기 작업을 큐에 전달 (RabbitMQ 등)



nova-scheduler

nova-api로부터 전달된 인스턴스 생성 요청을 수신하고, 가용 자원 상태를 고려하여 인스턴스를 배치할 적절한 컴퓨트 노드를 선택한다.

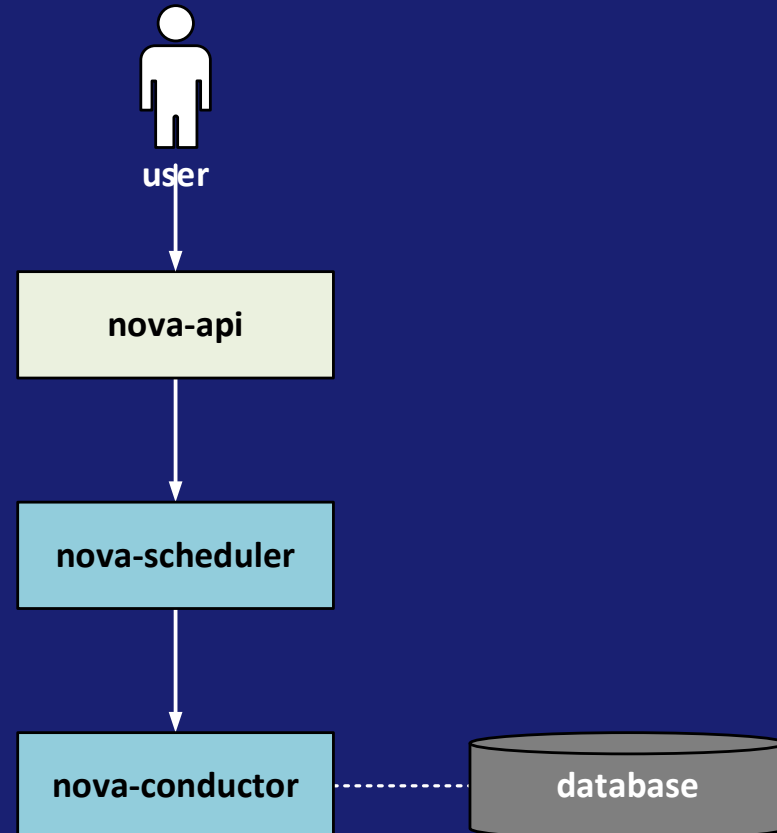
- 필터(Filter) + 가중치(Weigher) 기반
- nova-placement에서 리소스 사용량 정보 조회



nova-conductor

nova-compute가 직접 데이터베이스와 통신하지 않도록 중개자 역할을 수행하고, 보안을 강화하며 DB 연산을 위임받는다.

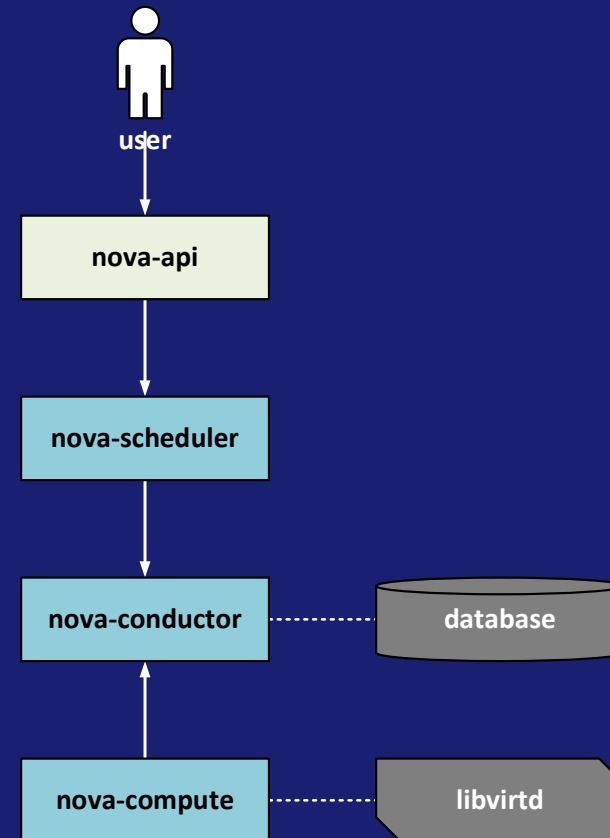
- nova-compute와 DB 사이의 간접 통신 담당
- RPC를 통해 동기/비동기 DB 작업 실행



nova-compute

nova-scheduler로부터 배정받은 인스턴스 생성 요청을 처리하며, libvirt 등 하이퍼바이저 인터페이스를 통해 실제 인스턴스를 생성하고 관리한다.

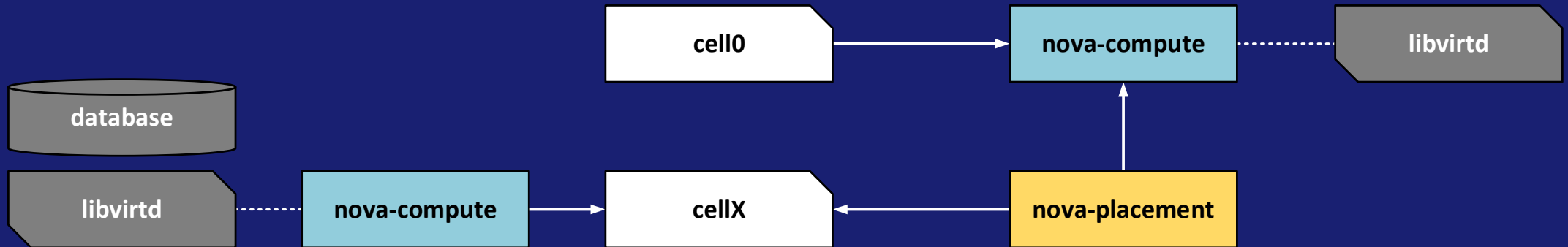
- Libvirt, Xen, VMware 등 다양한 드라이버 지원
- 각 노드에 배치됨 (agent 역할)



nova-placement

가용 자원(메모리, vCPU, NUMA 등)을 추적하고 보고하여, 스케줄러가 인스턴스를 최적으로 배치할 수 있도록 정보를 제공한다.

- 리소스 공급자(Resource Provider) 구조
- Placement API 제공 (별도 서비스)



nova-consoleauth

사용자의 콘솔 접속 시 인증 정보를 검증하고, nova-novncproxy가 세션을 중계할 수 있도록 인증 토큰을 발급한다.

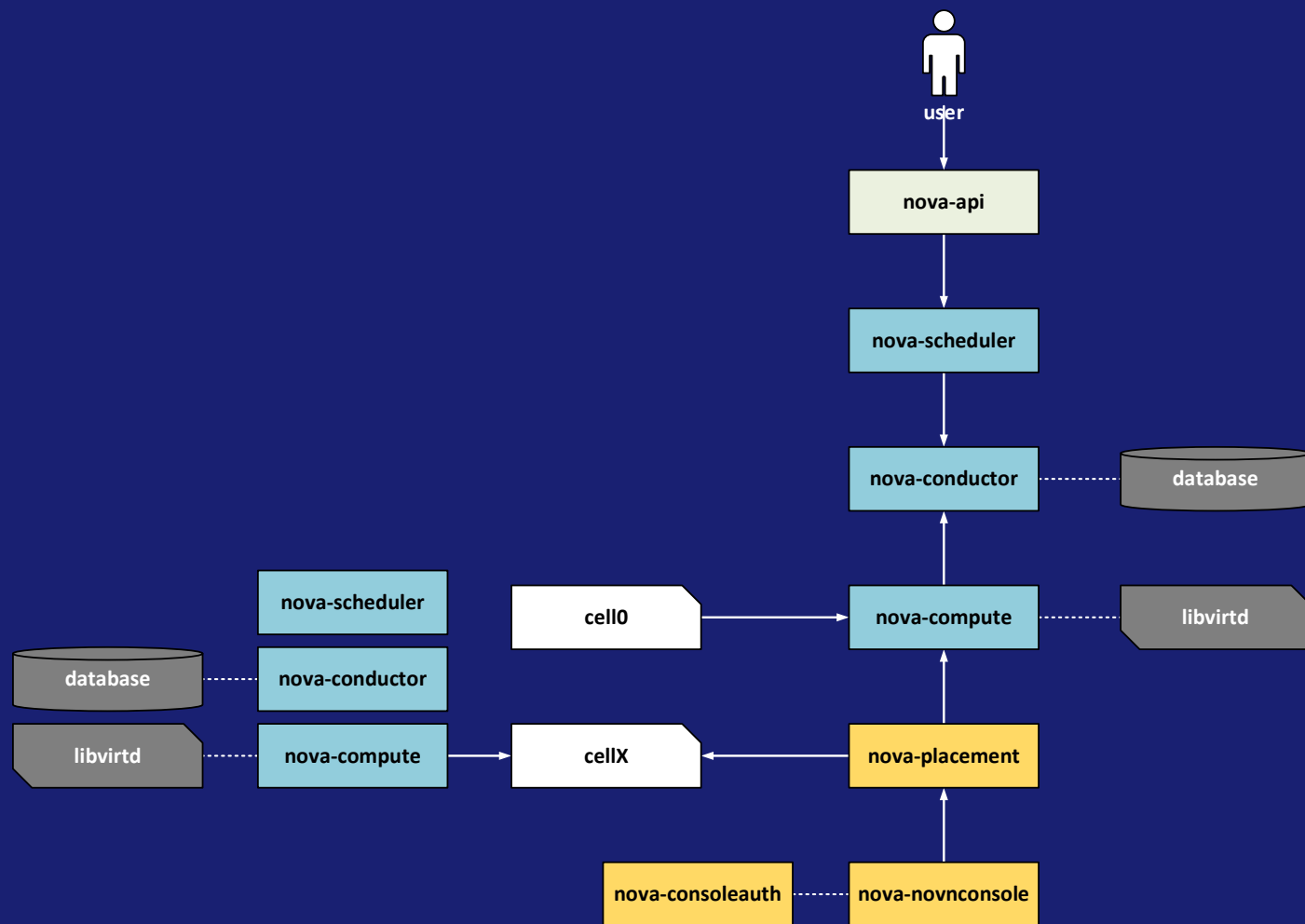
- 콘솔 접근 요청 유효성 검증
- 단시간 유효한 세션 기반 인증 처리

nova-novncproxy

웹 기반 VNC 접속을 브라우저로부터 받아 해당 인스턴스의 콘솔과 중계하며, nova-consoleauth의 인증 정보를 사용해 보안을 유지한다.

- VNC over WebSocket 지원
- 사용자 <—> novncproxy <—> 인스턴스 VNC 연결

정리



정리

위의 컴포넌트를 정리하면 다음과 같다.

컴포넌트	주요 기능	연결 대상
nova-api	사용자 요청 수신 및 전달	사용자 \leftrightarrow 내부 RPC
nova-scheduler	인스턴스 배치 결정	placement, conductor
nova-conductor	DB 연산 중계	nova-compute \leftrightarrow DB
nova-compute	인스턴스 생성 및 제어	libvirt, nova-conductor
nova-placement	자원 사용량 및 배치 정보 제공	scheduler, compute
nova-consoleauth	콘솔 인증 토큰 발급	novncproxy, API
nova-novncproxy	브라우저에서 VNC 콘솔 연결 중계	사용자 브라우저 \leftrightarrow VNC 콘솔

명령어

OpenStack Nova CLI

인스턴스 목록

인스턴스를 생성하기 위해서 먼저 이미지가 있는지 확인 해야 한다.

```
# openstack image list
```

FLAVOR(사양) 생성

역시, 가상머신을 생성하기 위해서 flavor 목록이 필요하다. 목록을 확인 후, 해당 자원이 없는 경우 생성한다. flavor는 생성 후, 수정이 안되기 때문에 신중하게 생성한다.

```
# openstack flavor list
# openstack flavor create --id 100 --ram 2048 --disk 20 --vcpus 2 \
--ephemeral 0 --swap 1024 --rxtx-factor 1.0 my-custom-flavor
```

FLAVOR(사양) 생성

Flavor 생성 시, 자주 사용하는 옵션은 다음과 같다.

옵션	의미
--id	flavor ID (자동 부여도 가능)
--ram	메모리 크기 (MB 단위)
--disk	루트 디스크 크기 (GB 단위)
--vcpus	할당할 vCPU 수
--ephemeral	임시 디스크 크기 (GB 단위)
--swap	스왑 공간 크기 (MB 단위, 선택사항)
--rxtx-factor	네트워크 전송 속도 비율
자원 이름	flavor 이름 (예: my-custom-flavor)

PRIVATE FLAVOR

특정 프로젝트만 접근 및 사용을 위해서는 아래와 같이 구성이 가능하다. 특정 프로젝트에서만 사용을 원하는 경우, `--project` 옵션을 통해서 선택이 가능하다. 어떤 프로젝트에서 접근이 가능한지 확인이 필요하다면 `access list` 명령어로 확인이 가능하다.

마지막으로, 특정 접근을 해제하기 위해서 `unset`를 통해서 제거가 가능하다.

```
# openstack flavor set --project <PROJECT_ID or PROJECT_NAME> my-private-flavor
# openstack flavor set --project project-a my-private-flavor
# openstack flavor set --project 27fc0a48edc74486aabb7cdef1234567 my-private-flavor
# openstack flavor access list --flavor my-private-flavor
# openstack flavor unset --project project-a my-private-flavor
```

네트워크 확인

오픈스택에서 인스턴스를 생성하기 위해서 neutron에서 생성한 자원을 확인해야 한다. 이전에 nova에서 생성 및 관리를 하였지만, 현재는 분리가 되었다.

```
# openstack network list
# openstack network subnet list
# openstack router list
```

KEYPAIR 생성

Keypair는 오픈스택에서 사용하는 인스턴스에 cloud-init가 동작하는 경우, 혹은, 키 인증(private, public)를 통해서 접근하는 경우 Keypair가 필요하다.

기본적으로 오픈스택 이미지는 cloud-init를 통해서 로그인을 제공한다. 만약, 사용자가 이미지에 cloud-init를 추가하지 않는 경우, 공개키는 인스턴스에 배포가 되지 않는다.

```
# openstack keypair list  
# openstack keypair create mykey > mykey.pem  
# chmod 600 mykey.pem
```

SECURITYGROUP(보안그룹)

보안그룹은 인스턴스에 구성된 PORT(IP Address)에 연결이 된다. 이를 통해서 ingress, egress 트래픽에 대해서 제어가 가능하다. 보안그룹은 계층으로 따지면 인스턴스의 L2 계층에 구성이 된다.

```
# openstack security group create my-secgroup --description "My security group"
# openstack security group rule create --protocol tcp --dst-port 22 my-secgroup
```

Instance

configuration of virtual machine

INSTANCE

위의 명령어 기반으로 Instance를 생성한다. 앞에서 다루었던 명령어로 자원 확인 혹은 생성이 되었다면, Instance 생성이 가능하다.

앞에서 이야기 했던 내용을 정리하면, Instance생성 조건은 다음과 같다.

1. 이미지
2. 사양(Flavor)
3. 네트워크(Subnet, Router)
4. 보안그룹(Security Group)
5. 접근 키(Keypair)

시작 전 INSTANCE VS VM

많은 엔지니어/강사 그리고, 아키텍처가 혼동해서 사용하는 단어 바로 Instance, VM이다. 이 두 가지의 단어는 다음과 같은 차이가 있다.

항목	Virtual Machine (VM)	Instance
정의	하이퍼바이저 위에 생성된 가상화된 컴퓨팅 환경	OpenStack에서 "프로젝트(tenant)" 관점으로 생성된 실행 단위
관점	인프라 관리자, 하이퍼바이저 수준에서 바라보는 객체	클라우드 사용자 또는 API 입장에서의 "자원 단위"
상태	멈춰 있어도 존재함 (VM은 계속 존재함)	API에서 정의된 생명주기를 따라 존재 (생성, 실행, 삭제 등)
식별	KVM/VMware 등 하이퍼바이저 수준에서 식별됨	OpenStack 내부 UUID 및 메타데이터로 식별됨
비유	하드웨어와 동일한 관리 대상	항시 유동적으로 상태가 변경이 되는 대상

인스턴스 생성

앞에서 학습한 내용을 가지고 인스턴스를 최종적으로 생성한다.

```
# openstack server create \  
  --flavor m1.small \  
  --image cirros \  
  --key-name mykey \  
  --security-group my-secgroup \  
  --network demo-net \  
my-vm
```

인스턴스 목록

올바르게 생성이 되었는지 아래 명령어로 확인한다.

```
# openstack server list
```

외부아이피 생성

현재 외부에서 접근이 안되는 상태. 외부에서 접근하기 위해서 최소 한 개의 Floating IP를 할당해야 한다. 아래 명령어로 아이피를 할당 받는다.

```
# openstack floating ip create public
```

외부아이피 연결

할당 받은 아이피를 아래 명령어로 인스턴스에 연결한다.

```
# openstack server add floating ip my-vm 203.0.113.10
```

인스턴스 콘솔 확인

인스턴스에서 발생한 콘솔 내용은 아래 명령어로 확인이 가능하다.

```
# openstack console log show my-vm
```

인스턴스 볼륨

인스턴스가 사용하는 O/S영역은 cinder volume으로 구성된 부분이 아니면, 제거 시 모든 내용은 제거가 된다. 이러한 이유로 영구적인 내용은 볼륨을 생성 및 구성하여 인스턴스에 연결한다.

```
# openstack volume create --size 5 my-volume  
# openstack server add volume my-vm my-volume  
  
# openstack server remove volume my-vm my-volume
```

인스턴스 라이프 사이클

인스턴스 라이프 사이클은 아래와 같은 명령어로 관리가 가능하다.

```
# openstack server stop my-vm  
# openstack server start my-vm  
# openstack server reboot my-vm  
# openstack server delete my-vm
```

랩

연습문제

2025-06-06

랩

metering

Ceilometer

Gnocchi

Panko

Adho

개요

OpenStack에서 미터링은 인프라 자원의 **사용량, 이벤트, 경보**를 수집하고 분석하기 위한 핵심 기능이다. 이 기능은 운영자에게 **청구, 모니터링, 자동 대응**을 가능하게 하며, 주로 다음 네 가지 서비스가 역할을 분담하여 구성된다.

구성요소	역할 서술	사용 목적
Ceilometer	OpenStack 자원에서 수집된 측정 데이터(meter) 를 통합 수집하는 기능을 담당한다. 다양한 에이전트(nova, neutron 등)를 통해 데이터를 받아드린다.	자원 사용량 측정 및 수집
Gnocchi	Ceilometer가 수집한 원시 메트릭 데이터를 저장하고, 집계 및 검색 기능을 제공한다. 시계열 데이터베이스(TSDB)로 동작하며 Ceilometer와 연동된다.	시계열 기반 데이터 저장 및 분석

개요

구성요소	역할 서술	사용 목적
Panko	OpenStack의 이벤트(event) 데이터를 저장하고 조회할 수 있는 이벤트 레코더 역할을 수행한다. 로그성 정보 저장에 가까우며, 이벤트 추적용이다.	이벤트 기반 감사 및 모니터링
Aodh	Ceilometer/Gnocchi에서 수집된 데이터를 바탕으로 경보(alarm) 을 생성하고, 조건 충족 시 알림을 제공한다. 자동화 액션(ex. Heat과 연동)도 가능하다.	경보 기반 자동화/알림

Ceilometer – 메트릭 수집기

Ceilometer는 다음처럼 자원 조회가 가능하다.

```
# openstack metric resource list
# openstack metric measures show <resource-id> --metric <metric-name>
# openstack metric list
# openstack metric resource list --type instance
```

메트릭 조회 결과

특정 메트릭 자원을 확인 시, 아래와 같이 출력이 된다.

```
$ openstack metric resource list --type instance
```

id	type	started_at	user_id	project_id
1b2c3d4e- ...	instance	2025-05 ...	user-a	project-a

메트릭 조회 결과

메트릭 자원 목록은 아래처럼 조회가 가능하다.

```
$ openstack metric list --resource-id 1b2c3d4e-...
```

+	+	+
id	name	
+	+	+
aabbccdd- ...	cpu_util	
+	+	+

메트릭 조회 결과

현재 수집된 정보를 아래처럼 수치 확인이 가능하다.

```
$ openstack metric measures show aabbccdd-...
```

timestamp	granularity	value
2025-06-02T12:00:00+00:00	300.0	32.5

Gnocchi

Gnocchi를 사용하게 된 이유는, 기존에 사용하던 MongoDB를 더 이상 사용하지 않고, Celiometer는 데이터를 전달하는 Publisher만 하고, 실제 데이터 저장 및 조회는 Gnocchi에서 한다.

Gnocchi는 이전 Ceilometer와 동일하게 MongoDB 기반으로 시계열 데이터를 저장한다.

Gnocchi – 시계열 메트릭 저장소

Gnocchi 명령어를 사용하기 위해서 다음과 같이 PIP 패키지가 설치 되어야 한다. 인스턴스 정보를 확인한다.

```
# pip3 install python3-gnocchiclient
```

```
# openstack metric resource list --type instance
```

id	type	started_at
1b2c3d4e-5f6g-7h8i-9j0k-abcde1234567	instance	2025-06-01T01:00:00

Gnocchi - 특정 리소스의 메트릭 목록 확인

아래 명령어로 특정 자원, 여기서는 인스턴스에서 사용하는 자원 상태를 확인한다.

```
# openstack metric list --resource-id 1b2c3d4e-5f6g-7h8i-9j0k-abcde1234567
```

id	name	created_by_proje
f0e1d2c3-b4a5-6789-cdef-12345678....	cpu_util	project-x
a1b2c3d4-e5f6-7890-ghij-98765432	memory.usage	project-x

Gnocchi - 특정 리소스의 메트릭 목록 확인

아래 명령어로 특정 자원, 여기서는 인스턴스에서 사용하는 자원 상태를 확인한다.

- 기본적으로 5분 간격(300초)으로 집계된 시계열 데이터.
- granularity는 메트릭을 수집한 시간 간격 (Gnocchi 설정에 따라 다름)

```
# openstack metric measures show f0e1d2c3-b4a5-6789-cdef-123456789abc
```

timestamp	granularity	value
2025-06-02T09:00:00+00:00	300.0	28.53
2025-06-02T09:05:00+00:00	300.0	31.10

Gnocchi – 자원 조회 방법

조회를 하기 위해서 보통 다음과 같은 순서로 조회 및 접근한다.

단계	목적	명령어
1단계	인스턴스 자원 조회	<code>openstack metric resource list --type instance</code>
2단계	해당 자원의 메트릭 ID	<code>openstack metric list --resource-id <resource-id></code>
3단계	메트릭 값 확인 (시계열)	<code>openstack metric measures show <metric-id></code>

메트릭 이름	설명
cpu_util	CPU 사용률 (%)
memory.usage	메모리 사용량 (bytes)
disk.read.bytes	디스크 읽기 (bytes)
network.incoming.bytes	수신 네트워크 (bytes)

Panko – 이벤트 레코더

오픈스택에서 발생한 이벤트를 조회하기 위해서 Panko를 통해서 조회가 가능하다. Panko는 openstack명령어에서 event를 통해서 조회가 가능하다.

```
# openstack event list
# openstack event show <event-id>
[
  {
    "event_type": "compute.instance.create.end",
    "generated": "2025-06-02T12:00:00",
    "traits": {
      "instance_id": "abc123",
      "user_id": "user1",
      "project_id": "proj1"
    }
  }
]
```

Aodh – 알람 및 자동화 트리거

Aodh는 이전에 Ceilometer로 구성하였던 부분을 분리하여, 별도의 알림 시스템을 구성하였다. 사용 및 구성 방법은 조금 복잡하지만, 기본적으로 다음과 같은 조건으로 동작한다.

1. 제한 수치
2. 자원 유형
3. 자원 이름

이 기능은 웹 hook 연동이 가능하기 때문에, 사용자가 원하는 경우 언제든지 웹 hook으로 알림 전달이 가능하다. 다만, 상대적으로 사용하기가 어렵기 때문에, 익숙하지 않는 사용자는 사용하기가 어려운 편이다.

Aodh - 알람 및 자동화 트리거 생성

아래처럼 알람 구성이 가능하다.

```
# openstack alarm create \  
  --name cpu_high \  
  --type gnocchi_resources_threshold \  
  --metric cpu_util \  
  --threshold 80 \  
  --resource-id <id> \  
  --resource-type instance \  
  --comparison-operator gt \  
  --aggregation-method mean \  
  --evaluation-periods 1 \  
  --granularity 300 \  
  --alarm-action 'http://alert.local/webhook'
```

Aodh – 알람 및 자동화 트리거 생성

구성이 완료가 되면, 아래 명령어로 조회가 가능하다.

```
# openstack alarm list
[
  {
    "name": "cpu_high",
    "state": "ok",
    "type": "gnocchi_resources_threshold",
    "threshold": 80,
    "aggregation_method": "mean"
  }
]
```

정리

아키텍처를 정리하면 다음과 같다.

[Ceilometer Agent]



측정 수집 API



[Gnocchi] → 시계열 DB



[Aodh] → 알람 생성 / 액션 실행



[Webhook / Heat]

[Panko] ← [Ceilometer Event API]
(이벤트 수집 저장)

랩

연습문제

2025-06-06

랩

automation

Heat

Ansible

HEAT 개요

Heat는 OpenStack에서 Infrastructure as Code(IaC)를 실현하는 핵심 컴포넌트로서, YAML 형식의 템플릿 (Heat Orchestration Template, H.O.T)을 기반으로 복잡한 인프라 구성을 자동으로 정의하고 배포할 수 있도록 한다.

1. Heat는 리소스를 선언적 방식으로 정의
2. 스택(stack)이라는 단위로 템플릿을 실행
3. 의존성 그래프를 내부적으로 구성하여 리소스 생성 순서를 자동으로 판단

예를 들어, 가상머신을 띄우는 템플릿을 작성하면 Heat는 자동으로 필요한 네트워크, 볼륨, 보안그룹 등을 순서대로 배포하고 구성한다.

다만, Heat는 오픈스택 내부에서만 사용이 가능하고, 외부에서는 사용이 불가능하다. 대표적인 활용 예는 OpenStack Director에서 오픈스택 구성 시 많이 사용한다.

HEAT 개요

Heat에서 제공하는 자원은 다음과 같다.

기능	설명
스택(Stack) 관리	리소스 집합을 스택 단위로 생성, 업데이트, 삭제
템플릿 기반 구성	HOT 또는 AWS CloudFormation 호환 JSON 사용 가능
리소스 추적	생성된 리소스 상태를 모니터링하고 롤백 기능 지원
파라미터화	템플릿에 사용자 정의 변수와 조건 사용 가능
리소스 간 의존성	depends_on 또는 참조 관계를 통해 생성 순서 자동 관리

laC 도구 비교

오픈스택에서 사용이 가능한 laC도구에 대해서 기능 비교이다.

항목	Heat	Ansible	SaltStack	Terraform
소속	OpenStack 전용	범용 자동화	범용 자동화	HashiCorp
목적	인프라 선언 및 배포	구성 관리 중심 + 일부 배포	구성 및 원격 실행	클라우드 인프라 선언적 배포
언어	YAML (HOT)	YAML (Playbook)	YAML (SLS)	HCL
작동 방식	스택 기반, OpenStack API 호출	SSH 또는 에이전트 기반 원격 실행	마스터-미니언 또는 에이전트리스	API 기반 상태관리
상태관리	자체적으로 스택 상태 추적	비상태 (Idempotent 아님)	비상태 또는 상태 저장 가능	상태 파일(state) 기반

laC 도구 비교

오픈스택에서 사용이 가능한 laC도구에 대해서 기능 비교이다.

항목	Heat	Ansible	SaltStack	Terraform
의존성 처리	리소스 간 depends_on 및 참조 자동 추론	순서 명시 필요	순서 제어 수동	리소스 간 그래프 추론
장점	OpenStack 최적화, 스택 전체 추적 가능	다양한 시스템 구성에 유리	대규모 시스템에 빠른 명령 전파	멀티 클라우드 지원, 선언적 laC
단점	타 플랫폼 미지원, 템플릿 러닝 커브	복잡한 의존성 자동화 어려움	YAML 작성이 복잡	상태 파일 충돌 가능

HEAT

laC

2025-06-06

HEAT 구조

Heat파일 구조는 YAML과 동일하다. 다만, 여러가지 버전이 있기 때문에 상단에 추가적으로 Heat버전에 대해서 선언이 필요하다.

```
heat_template_version: 2018-08-31    # 템플릿 버전 명시
description: 간단한 인스턴스 생성 예제

parameters:
  image:
    type: string
    default: cirros
  flavor:
    type: string
    default: m1.small
```

HEAT 구조

앞에 내용 계속 이어서...

```
network:
  type: string
  default: internal
resources:
  my_instance:
    type: OS::Nova::Server
    properties:
      name: test-vm
      image: { get_param: image }
      flavor: { get_param: flavor }
      networks:
        - network: { get_param: network }
```

HEAT 구조

앞에 내용 계속 이어서...

outputs:

instance_name:

value: { get_attr: [my_instance, name] }

HEAT 적용

Heat 생성 및 구성이 완료가 되면 오픈스택에 적용한다.

```
# openstack stack create -t simple-vm.yaml --parameter network=private-net  
demo-stack  
# openstack stack create -t simple-vm.yaml --stack-name demo-stack  
# openstack stack list  
# openstack stack show demo-stack  
# openstack server list
```

HEAT 적용

명령어 결과 화면 하나씩 구성하면 다음과 같이 나온다.

```
$ openstack stack create -t simple-vm.yaml --stack-name demo-stack
```

Field	Value
id	01a3f83e-e1ad-4a88-93c3-9086c4cc80a3
stack_name	demo-stack
stack_status	CREATE_IN_PROGRESS
creation_time	2025-06-02T12:00:00

HEAT 적용

명령어 결과 화면 하나씩 구성하면 다음과 같이 나온다.

```
$ openstack stack list
```

ID	Stack Name	Stack Status	Creation Time	Updated Time
01a ...	demo-stack	CREATE_COMPLETE	2025-06-02T12:00	None

HEAT 적용

명령어 결과 화면 하나씩 구성하면 다음과 같이 나온다.

```
$ openstack server list
```

ID	Name	Status	Networks	Image	Flavor
a9e123	test-vm	ACTIVE	pr=10.0	cirros	m1.small

앤서블

laC

2025-06-06

설명

Ansible은 IT 인프라의 자동화, 구성 관리, 애플리케이션 배포, 오케스트레이션을 위한 오픈소스 도구로서, YAML 기반의 '플레이북(playbook)'을 통해 작업을 자동화하는 선언적 도구이다.

1. SSH를 기반으로 동작하며, 별도의 에이전트를 설치할 필요가 없다.
2. 관리 대상 노드에 명령어를 실행하거나 설정 파일을 배포하고, 서비스 상태를 제어하는 등 시스템 운영 작업을 자동화할 수 있다.
3. 코드처럼 인프라를 정의하는 방식, 즉 Infrastructure as Code(IaC) 개념을 따르며, 사람도 읽기 쉬운 형식으로 시스템을 구성할 수 있다.

Ansible과 OpenStack의 관계

Ansible은 OpenStack의 리소스를 프로그래밍 방식으로 관리할 수 있도록 여러 모듈을 제공한다. 대표적으로 다음과 같은 관계가 있다.

1. 오픈스택 클라우드 자원 생성

- 네트워크, 서브넷, 라우터, 보안 그룹, 키페어, 이미지, 볼륨, 인스턴스 등 모든 기본 자원을 Ansible로 정의하고 배포할 수 있다.
- `openstack.cloud.os_server`, `os_network`, `os_image` 등의 모듈을 활용하여 리소스를 조작한다.

2. 클라우드 배포 자동화

- `kolla-ansible`, `openstack-ansible`, `kayobe` 등 다양한 OpenStack 배포 도구가 Ansible 기반으로 작성되어 있다.
- 따라서 Ansible은 OpenStack을 설치하는 도구이자, 운영하는 도구이기도 하다.

Ansible과 OpenStack의 관계

3. CI/CD 및 DevOps 연계

- OpenStack과 연동하여 Jenkins, GitLab CI 등의 파이프라인에서 Ansible을 호출해 클라우드 환경에 자동으로 리소스를 배포하거나 회수하는 작업을 할 수 있다.

4. Heat의 보완 도구

- Heat는 OpenStack 내부에서만 작동하지만, Ansible은 외부 시스템(OS 레벨 설정, 패키지 설치, 파일 배포 등)까지 다룰 수 있기 때문에 Heat + Ansible을 함께 사용하는 하이브리드 자동화 전략도 가능하다.

앤서블 작성

디렉터리 구조는 다음과 같다.

```
openstack_vm_setup/  
├─ create_resources.yml      # 메인 플레이북  
├─ vars.yml                  # 변수 정의  
└─ requirements.txt          # collections 정의 (openstack.cloud)  
# ansible-galaxy collection install openstack.cloud
```

변수 파일

다음과 같이 변수 파일을 작성한다.

```
# vi vars.yaml
auth:
  auth_url: http://<keystone>:5000/v3
  username: admin
  password: secret
  project_name: admin
  domain_name: default
image_name: cirros
image_url: http://download.cirros-cloud.net/0.6.2/cirros-0.6.2-x86_64-
disk.img
flavor_name: m1.tiny
network_name: demo-net
```

변수 파일

위의 내용 계속 이어서...

```
subnet_name: demo-subnet
subnet_cidr: 192.168.100.0/24
router_name: demo-router
security_group: allow-ping-ssh
volume_name: demo-volume
instance_name: demo-vm
```

메인 플레이북

다음과 같이 변수 파일을 작성한다.

```
# vi create_resources.yaml
- name: OpenStack 기본 자원 생성 및 VM 배포
  hosts: localhost
  gather_facts: no
  vars_files:
    - vars.yml
  collections:
    - openstack.cloud
```

메인 플레이북

앞에 내용 계속 이어서...

```
tasks:
  - name: 네트워크 생성
    os_network:
      cloud: undercloud
      state: present
      name: "{{ network_name }}"
      external: no
      validate_certs: no
      auth: "{{ auth }}"
```

메인 플레이북

앞에 내용 계속 이어서...

```
- name: 서브넷 생성
  os_subnet:
    cloud: undercloud
    state: present
    name: "{{ subnet_name }}"
    network_name: "{{ network_name }}"
    cidr: "{{ subnet_cidr }}"
    ip_version: 4
    dns_nameservers: [8.8.8.8]
    auth: "{{ auth }}"
```

메인 플레이북

앞에 내용 계속 이어서...

```
- name: 라우터 생성
  os_router:
    cloud: undercloud
    state: present
    name: "{{ router_name }}"
    external_gateway: public # external 네트워크 이름
    interfaces:
      - "{{ subnet_name }}"
    auth: "{{ auth }}"
```

메인 플레이북

앞에 내용 계속 이어서...

```
- name: 보안 그룹 생성
  os_security_group:
    cloud: undercloud
    state: present
    name: "{{ security_group }}"
    description: "Allow ping and ssh"
    auth: "{{ auth }}"
```

메인 플레이북

앞에 내용 계속 이어서...

```
- name: 보안 그룹 룰 추가 - SSH
  os_security_group_rule:
    cloud: undercloud
    security_group: "{{ security_group }}"
    protocol: tcp
    port_range_min: 22
    port_range_max: 22
    remote_ip_prefix: 0.0.0.0/0
    state: present
    auth: "{{ auth }}"
```

메인 플레이북

앞에 내용 계속 이어서...

```
- name: 보안 그룹 룰 추가 - Ping
  os_security_group_rule:
    cloud: undercloud
    security_group: "{{ security_group }}"
    protocol: icmp
    remote_ip_prefix: 0.0.0.0/0
    state: present
    auth: "{{ auth }}"
```

메인 플레이북

앞에 내용 계속 이어서...

```
- name: 이미지 등록
  os_image:
    cloud: undercloud
    name: "{{ image_name }}"
    container_format: bare
    disk_format: qcow2
    filename: "/tmp/{{ image_name }}.qcow2"
    state: present
    auth: "{{ auth }}"
  when: not lookup('file', '/tmp/' + image_name + '.qcow2',
errors='ignore')
```

메인 플레이북

앞에 내용 계속 이어서...

```
- name: 이미지 다운로드 (필요 시)
  get_url:
    url: "{{ image_url }}"
    dest: "/tmp/{{ image_name }}.qcow2"
    when: not lookup('file', '/tmp/' + image_name + '.qcow2',
errors='ignore')
```

메인 플레이북

앞에 내용 계속 이어서...

```
- name: 볼륨 생성
  os_volume:
    cloud: undercloud
    display_name: "{{ volume_name }}"
    size: 1
    state: present
    auth: "{{ auth }}"
```

메인 플레이북

앞에 내용 계속 이어서...

```
- name: 인스턴스 생성
  os_server:
    cloud: undercloud
    state: present
    name: "{{ instance_name }}"
    image: "{{ image_name }}"
    flavor: "{{ flavor_name }}"
    key_name: default # 사전에 키페어 생성되어 있어야 함
    network: "{{ network_name }}"
```

메인 플레이북 (create_resources.yml)

앞에 내용 계속 이어서...

```
- name: 인스턴스 생성
```

```
...
```

```
...
```

```
  security_groups:
    - "{{ security_group }}"
  boot_from_volume: yes
  volume_size: 1
  terminate_volume: yes
  auth: "{{ auth }}"
```

메인 플레이북 실행 및 적용

앞에 내용 계속 이어서...

```
# ansible-playbook -I localhost, create_resources.yaml
TASK [네트워크 생성] *****changed: [localhost]
TASK [서브넷 생성] *****changed: [localhost]
TASK [이미지 등록] *****ok: [localhost]
TASK [인스턴스 생성] *****changed: [localhost]
```

오픈스택 서비스 확장

CI/CD

Grafana/Prometheus

확장 서비스 설치 및 구성

CI/CD

ZUUL

개요

OpenStack과 Kubernetes는 각각의 목적과 구조가 다른 인프라 플랫폼이기 때문에, 그 위에서 운용되는 CI/CD 도구와 이미지 빌드 방식도 명확한 차이를 가진다.

OpenStack의 CI/CD는 인프라 자체의 구성 요소를 개발·테스트·배포하기 위한 목적이 중심이며, Kubernetes의 CI/CD는 해당 플랫폼 위에서 동작하는 애플리케이션의 개발 및 운영 자동화에 초점을 둔다.

OpenStack은 일반적으로 Zuul, Gerrit, Nodepool 등의 도구를 통해 CI/CD를 구성한다. 개발자가 Gerrit에 코드 변경을 등록하면 Zuul이 이를 감지하여 자동 테스트를 트리거하며, Nodepool을 통해 테스트 환경용 VM이 OpenStack 상에 동적으로 생성된다.

생성된 VM 위에는 DevStack 또는 Kolla-Ansible을 이용해 OpenStack의 각 컴포넌트가 설치되고, 이후 기능 테스트가 진행된다. 이 모든 과정은 코드 리뷰 기반의 파이프라인으로 설계되어 있으며, 결과는 다시 Gerrit에 피드백으로 기록된다.

개요

반면, Kubernetes에서는 보다 일반적인 CI/CD 워크플로우와 GitOps 접근 방식이 주로 사용된다.

Jenkins, GitLab CI, Tekton 같은 CI 도구는 코드 저장소(Git)에 커밋이 발생했을 때 컨테이너 이미지를 자동으로 빌드하고 Docker Registry 등에 저장한다.

이후 ArgoCD나 Flux와 같은 GitOps 도구가 Git 저장소의 상태를 감시하다가 변경 사항을 감지하면 자동으로 Kubernetes 클러스터에 해당 애플리케이션을 배포한다. 이러한 방식은 빠른 피드백과 반복적인 배포에 최적화되어 있으며, 애플리케이션 중심의 CD 환경을 제공한다.

개요

이미지 빌드 방식에서도 양자는 철학이 다르다. OpenStack은 kolla-build라는 도구를 통해 Ansible과 Jinja2 기반 Dockerfile 템플릿으로 운영자용 OpenStack 서비스 컨테이너 이미지를 빌드한다. 이 빌드는 서비스별로 다양한 설정이 내포되어 있으며, 대규모 인프라 구성에 적합하다.

반면 Kubernetes 기반 워크로드에서는 일반적인 docker build, buildah, kaniko 등을 통해 Dockerfile에서 애플리케이션 이미지를 간단히 빌드하고 경량화하는 것을 선호한다. 이는 마이크로서비스의 민첩한 배포와 업데이트를 위한 핵심 전략이다.

개요

결론적으로, OpenStack은 인프라 소프트웨어 개발과 테스트를 위한 복잡한 CI/CD 체계를 갖추고 있으며, VM 기반 환경과 코드 리뷰 기반 파이프라인을 중심으로 설계되어 있다. 반면 Kubernetes는 애플리케이션 개발을 위한 민첩한 배포 자동화 체계를 제공하며, 컨테이너 기반 CI/CD 및 GitOps 모델이 중심이다. 이로 인해 동일한 CI/CD라는 용어를 사용하더라도, 실제 구현 방식과 목적은 플랫폼의 철학에 따라 크게 달라진다.

ZUUL CI/CD

ZUUL기반으로 CI/CD를 구성하기 위해서 다음과 같은 흐름으로 작성 및 구성한다.

- [1] Git 커밋 (Gerrit or GitHub)
↓
- [2] Zuul이 감지 → 파이프라인 실행
↓
- [3] 이미지 빌드 (Packer 또는 Diskimage-builder)
↓
- [4] 빌드한 QCOW2 이미지 Glance에 등록
↓
- [5] Heat 스택으로 배포 테스트 (또는 Ansible)
↓
- [6] 결과 Gerrit에 피드백 (Success / Fail)

ZUUL COMPOSE

podman-compose를 사용해서 아래 내용을 구성한다.

```
# vi docker-compose.yaml
```

ZUUL 설치 및 구성

데비안 계열은 DEB로 제공하지만, 레드햇 계열 경우에는 RPM으로 제공하지 않는다. 사용하기 위해서 Podman이나 혹은 Docker 기반으로 사용을 권장한다.

```
# dnf install -y podman podman-compose
# git clone https://opendev.org/zuul/zuul.git
# cd zuul/tools/compose
# cp .env-sample .env
# podman-compose up -d
```

Grafana/Prometheus

모니터링

Prometheus

- Prometheus는 시계열(Time-series) 데이터베이스 시스템 및 애플리케이션의 메트릭 데이터를 주기적으로 수집(scrape) 하고, 이를 저장, 질의, 경고(Alerting)할 수 있도록 설계된 모니터링 수집기이다.
- Prometheus는 "pull 기반" 모니터링 시스템으로 설정된 endpoint(예: node-exporter, OpenStack Exporter 등)를 일정 주기로 조회하여 CPU, 메모리, 디스크, 네트워크 등의 실시간 성능 지표를 가져온다.
- 또한 자체 쿼리 언어인 PromQL을 사용하여 메트릭에 대해 복잡한 조건 검색과 경고 조건을 정의할 수 있다.

Grafana

- Grafana는 Prometheus가 수집한 시계열 데이터를 시각적으로 표현할 수 있는 대시보드 기반 시각화 도구이다.
- Grafana는 다양한 데이터 소스(Prometheus, Loki, InfluxDB 등)를 연동하여 사용자가 원하는 방식의 차트, 그래프, 알람 대시보드를 구성할 수 있도록 돕는다.
- 또한 Grafana는 경고(Alert Rule)를 시각적으로 정의하고, Slack, Email 등으로 경고 메시지를 전송할 수 있다.

Ceilometer

OpenStack의 Telemetry 서비스로, 클라우드 인프라 내 자원(예: 인스턴스, 볼륨, 네트워크 등)의 상태와 이벤트를 수집, 저장, 전달하는 역할을 수행한다. 과거에는 모니터링+미터링+알람을 모두 Ceilometer가 처리했지만, 이후 기능이 분리.

1. 메트릭 저장은 Gnocchi
2. 이벤트 저장은 Panko
3. 알람은 Aodh
4. 수집은 Ceilometer Agent

정리

Prometheus는 OpenStack과 독립적인 모니터링 스택이다. 구조적으로 다음과 같다.

1. Ceilometer 계열은 OpenStack 내부의 API, MQ(RabbitMQ), Agent를 통해 OpenStack 서비스와 밀접하게 통합됨.
2. Prometheus는 외부 exporter(node-exporter, openstack-exporter 등)를 통해 OpenStack 외부에서 수집함.

즉, OpenStack이 없어도 Prometheus는 동작 가능하고, OpenStack이 있어도 Prometheus는 OpenStack에 직접 의존하지 않음.

정리

Prometheus는 OpenStack과 독립적인 모니터링 스택이다.

1. Ceilometer 계열은 OpenStack 내부의 API, MQ(RabbitMQ), Agent를 통해 OpenStack 서비스와 밀접하게 통합됨.
2. Prometheus는 외부 exporter(node-exporter, openstack-exporter 등)를 통해 OpenStack 외부에서 수집함.

항목	Ceilometer 계열	Prometheus
연동 방식	OpenStack 서비스 내부 통합	외부 Exporter 연결
예시 수집 대상	Nova, Cinder, Neutron API	node-exporter, libvirt-exporter 등
수집 방식	AMQP(RabbitMQ) 기반 push	HTTP 기반 pull
저장소	Gnocchi	자체 TSDB
필수 구성 요소	ceilometer-agent, aodh, panko 등	Exporter만 있으면 가능

정리

Prometheus & Grafana 와 Ceilometer의 차이 및 관계.

항목	Ceilometer 계열	Prometheus/Grafana
수집 방식	push + agent 기반	pull 기반
수집 대상	OpenStack 리소스 중심	시스템 리소스 중심 (node-exporter, cadvisor 등)
저장소	Gnocchi (시계열 DB)	Prometheus 내장 TSDB
시각화	Horizon 통합 or 외부 도구	Grafana
알람	Aodh	Prometheus Alertmanager 또는 Grafana

모니터링 시각화

kolla-ansible를 사용하는 경우, 아래처럼 구성이 가능하다.

```
# vi gloabls.yaml
enable_prometheus: "yes"
enable_grafana: "yes"

# Optional: 성능 메트릭 수집을 위한 서비스들
enable_node_exporter: "yes"
enable_cadvisor: "yes"
enable_alertmanager: "yes"
grafana_admin_password: "MySecureGrafanaPass"
```

`enable_cadvisor`는 Docker 전용. Podman에서 사용 불가능.

시각화 적용

deploy를 통해서 기능추가 혹은 기존 설정 업데이트는 reconfigure를 통해서 가능하다.

```
# kolla-ansible -i /etc/kolla/inventory deploy
# kolla-ansible -i /etc/kolla/inventory reconfigure -t prometheus
# kolla-ansible -i /etc/kolla/inventory reconfigure -t grafana
# podman ps | grep prometheus
# podman ps | grep grafana
```