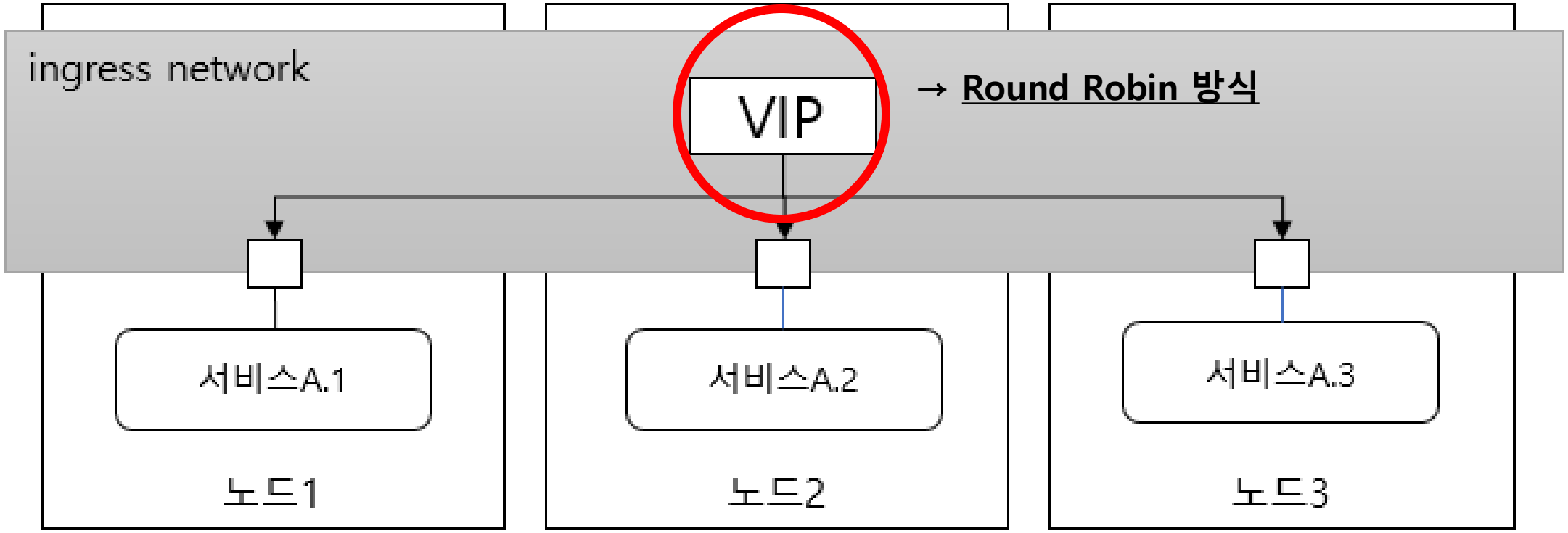
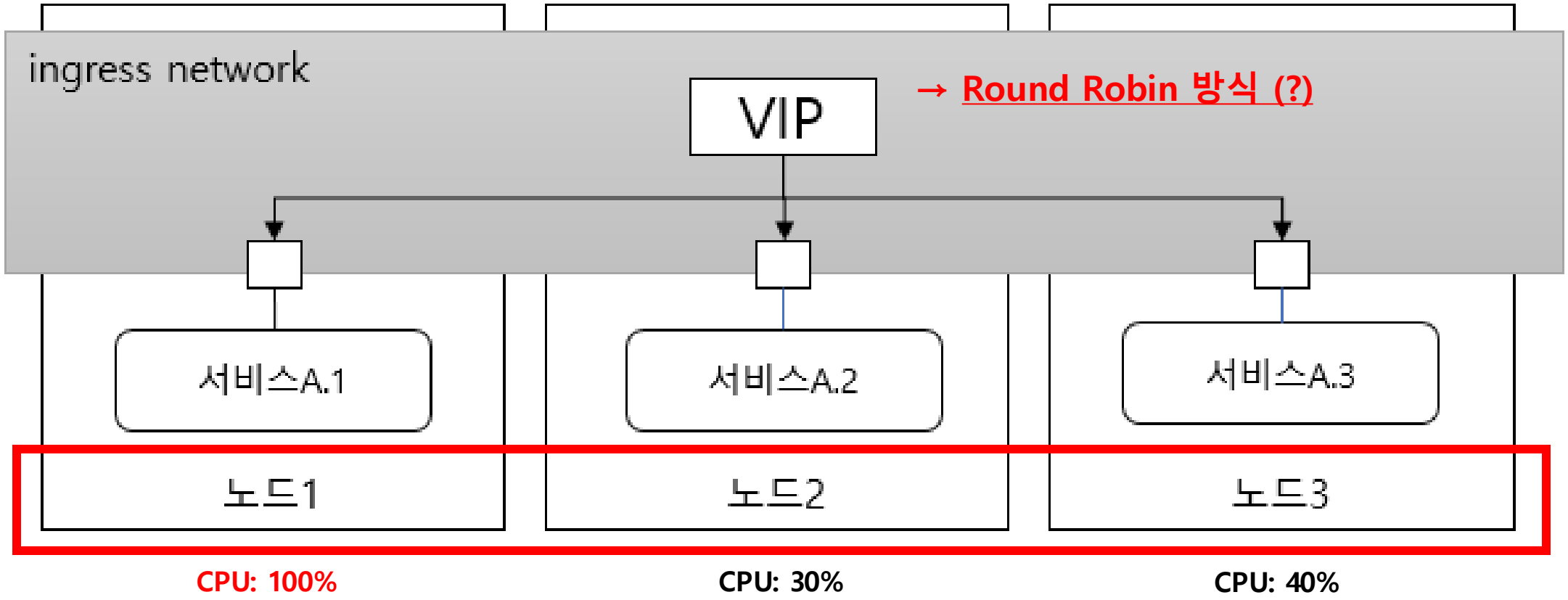
물리 자원 상태를 고려한 Dynamic Load Balancing

[5조 개복치]
김용범 김찬훈 이종서

Docker Swarm의 한계



Docker Swarm의 한계



기존 대응 방안

□ **Kubernetes의 경우 kube-scheduler를 활용한 Auto-Scaling으로 대응하는 것이 일반적**

Auto-Scaling Tool	특징	비고
Cluster Autoscaler	Pod를 추가할 노드(인스턴스)가 부족할 경우 자동 확장 (GKE, EKS 등)	kube-scheduler를 활용하여 물리자원 상태에 기반한 Auto-Scaling 가능
Karpenter	추가할 Pod의 요구사항에 따른 지능형 노드 확장 (EKS 등)	

- **다만, Docker Swarm의 경우 이러한 대응책이 없음**
- **수동 scaling 전, 효율적인 자원 활용을 위해**
  실시간으로 물리자원 상태를 감지하여 트래픽 분산하는 기능이 필요하다고 판단

기존 로드밸런서 특징

로드밸런서	특징	물리자원 기준 동적 분산
HAProxy	다양한 알고리즘 제공	직접 지원 X (Docker Swarm 자동 인식 불가, 수동 설정 필요)
Traefik	Round Robin, Sticky Session, Random 등 알고리즘 제공	직접 지원 X **(Docker Swarm 자동 인식 가능, 수동 설정 불필요)**
Envoy (+ Istio)	쿠버네티스 기반 L7 라우팅	직접 지원 X

- **일반적으로 트래픽 라우터는 L4/L7만 보기 때문에**
  CPU 사용률 등 "노드 상태"는 고려하지 않음
- **Docker Swarm에서 활용 가능한 Traefik에 python script를 접목시켜**
  물리자원 상태를 고려한 Dynamic Load Balancer를 개발하고자 함

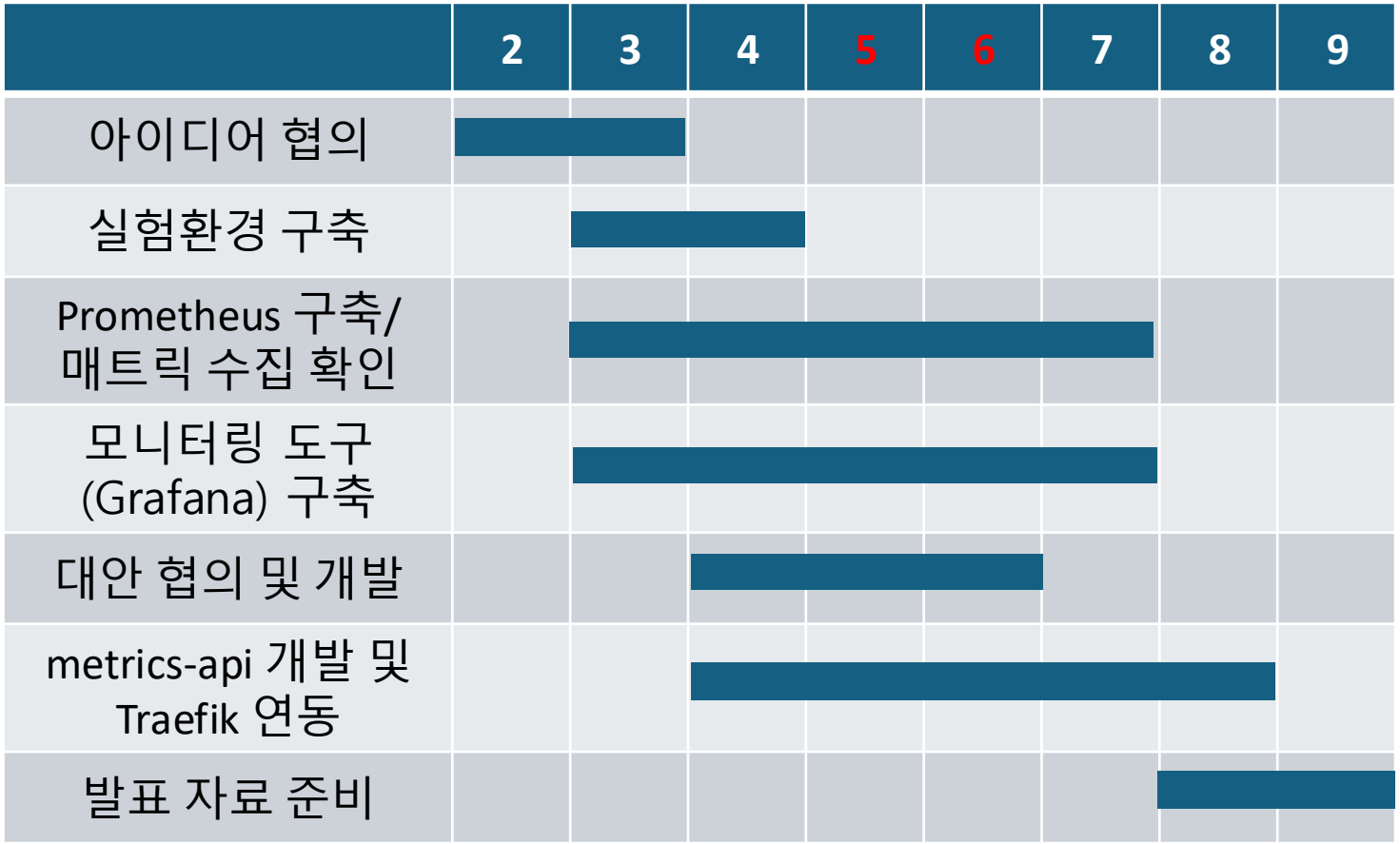
프로젝트 진행 내용

GitHub

OneDrive

[역할 분담]

- 김용범 : 자동화 도구 (metrics-api) 개발
- 김찬훈 : 모니터링 도구 (Grafana 등) 구축
- 이종서 : 기획 및 테스트 환경 구축



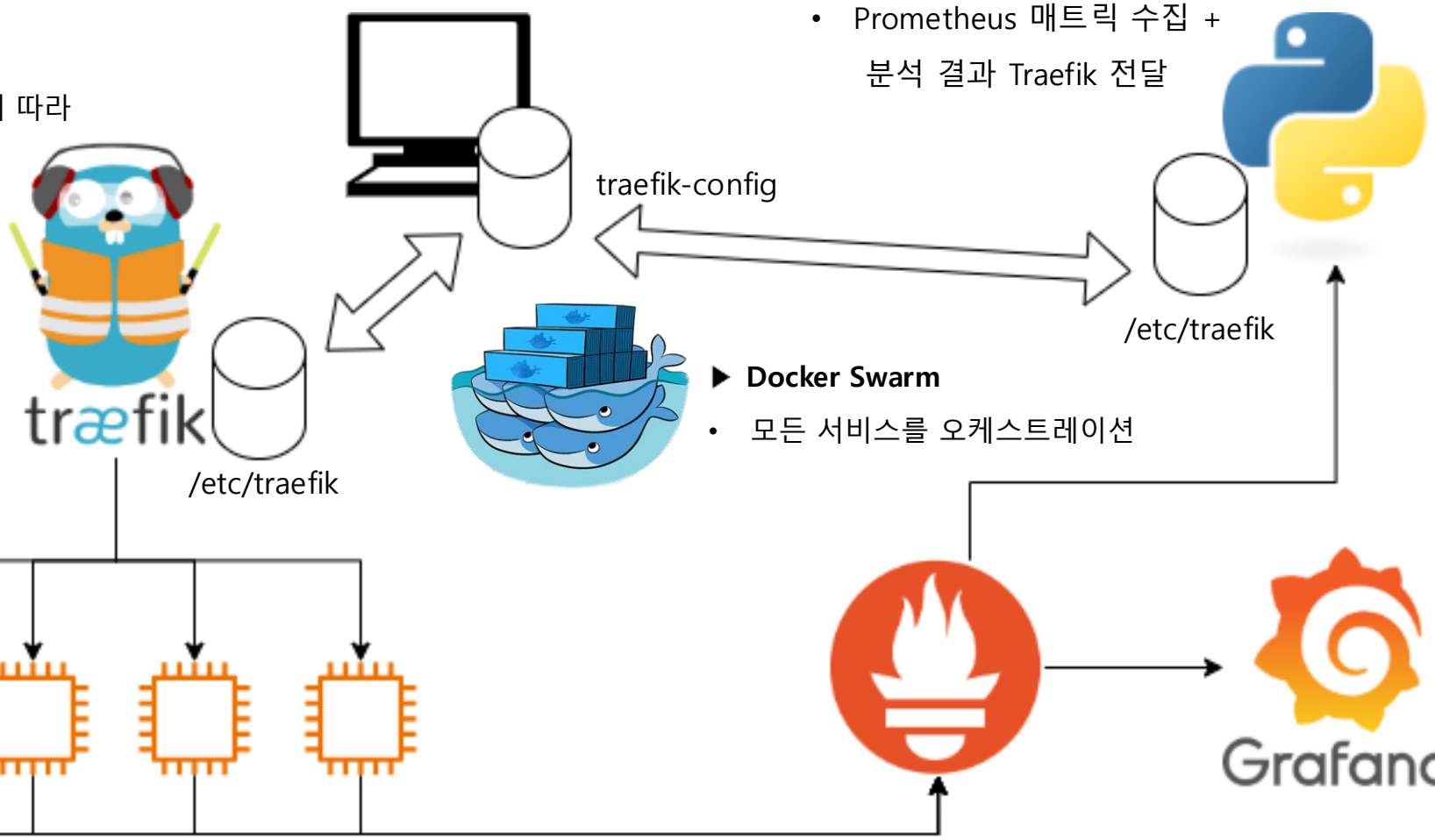
프로젝트 구성

▶ **Traefik**

- metrics-api 분석에 따라 트래픽 분산

▶ **metrics-api**

- Prometheus 매트릭 수집 + 분석 결과 Traefik 전달



▶ **Docker Swarm**

- 모든 서비스를 오케스트레이션

▶ **Grafana (Port# 3000)**

- Prometheus와 연결하여 시각화 및 모니터링
- 사용자 정의 대시보드로 실시간 부하 관찰

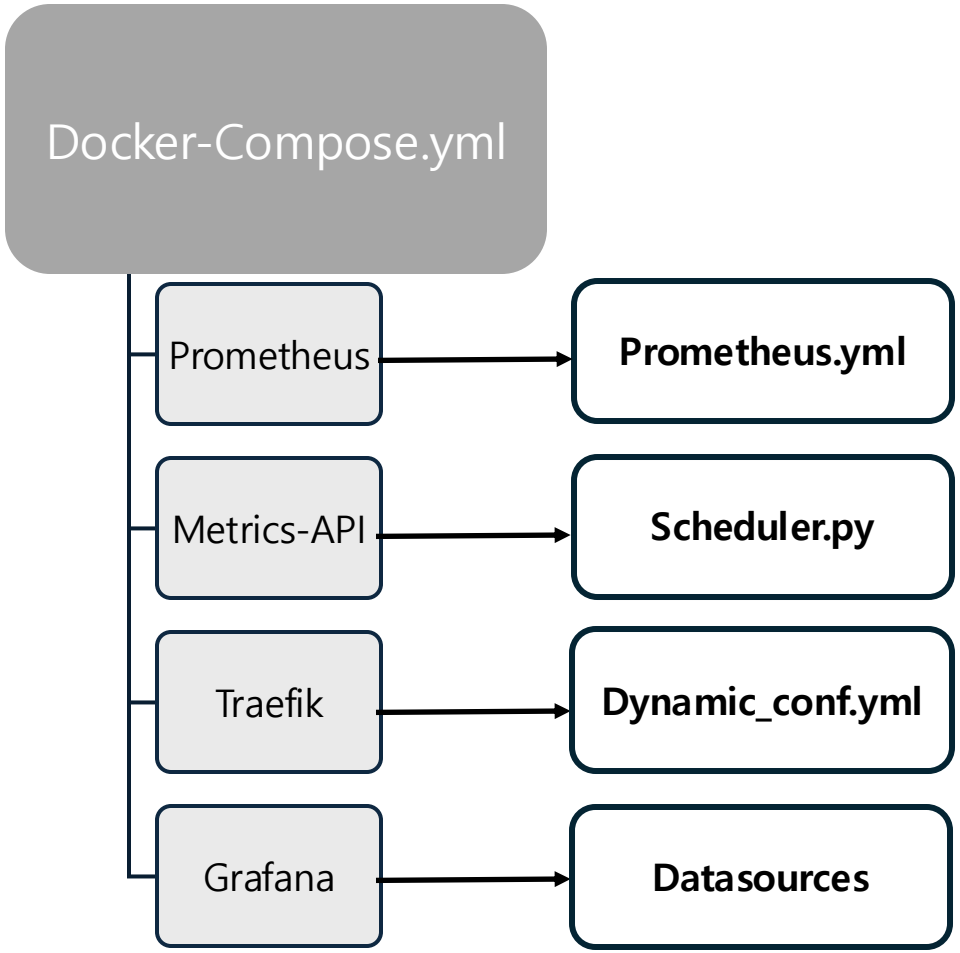
▶ **Node Exporter (Port# 9100)**

- 각 노드의 CPU/RAM 등 리소스 사용량 수집

▶ **Prometheus (Port# 9090)**

- 메트릭 수집 및 쿼리 기능 제공

시스템 구성



- prometheus.yml: 데이터 수집 대상 노드 및 수집 주기 설정
- scheduler.py: 자원 사용량 분석 및 최적 노드 선택 로직 구현
- dynamic_conf.yml: Traefik이 참조할 동적 라우팅 설정 파일
- datasources: Grafana에서 Prometheus와 연결하고 대시보드 구성

Prometheus 설정

```yaml
version: '3.7'

▷Run All Services
services:

  ▷Run Service
  prometheus:
    image: prom/prometheus:latest
    user: root
    ports:
      - "9090:9090"
    configs:
      - source: prometheus_config
        target: /etc/prometheus/prometheus.yml
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.retention.time=1m'
      - '--storage.tsdb.retention.size=100MB'
    deploy:
      mode: replicated
      replicas: 1
      placement:
        constraints: [node.role == manager]        # manager 노드에게만 prometheus 설치
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    networks:
      - monitor_network
    environment:
      - TZ=Asia/Seoul                              # 시간을 UTC -> KST로 변경
```

- Prometheus 서비스를 정의한 docker-compose 설정
- 이미지, 포트, 설정 파일, 실행 노드 조건 등 지정
- 시스템 정보를 수집하기 위해서는 root 권한이 필요
- 컨테이너 간 통신을 위한 별도의 overlay 네트워크를 지정

Prometheus 설정

```yaml
global:
  scrape_interval: 5s   # 데이터 수집 최소 간격

scrape_configs:
  - job_name: 'node-exporter'

    dockerswarm_sd_configs:
      - host: unix:///var/run/docker.sock
        role: tasks

    # DNS로 가져온 컨테이너의 IP를 무시하고, 대신 호스트 IP를 스크랩

    relabel_configs:
      - source_labels: [__meta_dockerswarm_node_address]
        target_label: __address__
        replacement: $1:9100
      - source_labels: [__meta_dockerswarm_node_role]
        regex: manager
        action: drop
```

자동으로 검색

- Prometheus가 node-exporter로부터 CPU/RAM 사용량을 수집하도록 설정
- Swarm 내부에 배포된 컨테이너를 자동으로 검색
- Docker Swarm에서 각 노드의 실제 IP를 추출해 모니터링

Metrics-API 설정

```python
from flask import Flask
from scheduler import generate_dynamic_config
import threading, time, os
from datetime import datetime

app = Flask(__name__)

CONFIG_PATH = "/etc/traefik/dynamic_conf.yml"

# metric-api 업데이트 로그 생성
def update_config():
    config = generate_dynamic_config()
    with open(CONFIG_PATH, "w") as f:
        f.write(config)
    print(f"[{datetime.now()}] [INFO] Config updated.")
    return config

# 주기적으로 (5초) scheduler.py 실행
def schedule_loop():
    while True:
        update_config()
        time.sleep(5)

# 실행 시작
if __name__ == '__main__':
    threading.Thread(target=schedule_loop, daemon=True).start()
    app.run(host="0.0.0.0", port=8000)
```

- Flask 서버에서 metrics 설정을 5초마다 갱신
- 자동으로 생성된 config 파일을 Traefik이 실시간 감지하여 반영

Metrics-API 설정

```python
# 물리 자원에 대한 쿼리 정의
CPU_QUERY = '100 - avg by(instance)(rate(node_cpu_seconds_total{mode="idle"}[15s])) * 100'
RAM_QUERY = '(1 - (node_memory_MemAvailable_bytes / node_memory_MemTotal_bytes)) * 100'
```

```python
def get_best_node():
    cpu_usages = get_cpu_usages()
    ram_usages = get_ram_usages()

    if not cpu_usages or not ram_usages:
        print("[ERROR] Failed to fetch resource data from Prometheus.")
        return None

    # CPU와 RAM을 고려한 node들의 가용 점수 계산
    scores = {}

    # CPU와 RAM에 대한 가중치 설정
    cpu_weight = 0.8
    ram_weight = 0.2

    for instance in cpu_usages:
        cpu = cpu_usages.get(instance, 100)
        ram = ram_usages.get(instance, 100)
        score = (100 - cpu) * cpu_weight + (100 - ram) * ram_weight
        scores[instance] = score
        print(f"[INFO] {instance} → CPU: {cpu:.1f}%, RAM: {ram:.1f}%, SCORE: {score:.1f}")

    best_node = max(scores.items(), key=lambda x: x[1])
    print(f"[INFO] Idle node: {best_node[0]} (SCORE: {best_node[1]:.2f})")
    return best_node[0]
```

SCORE 계산

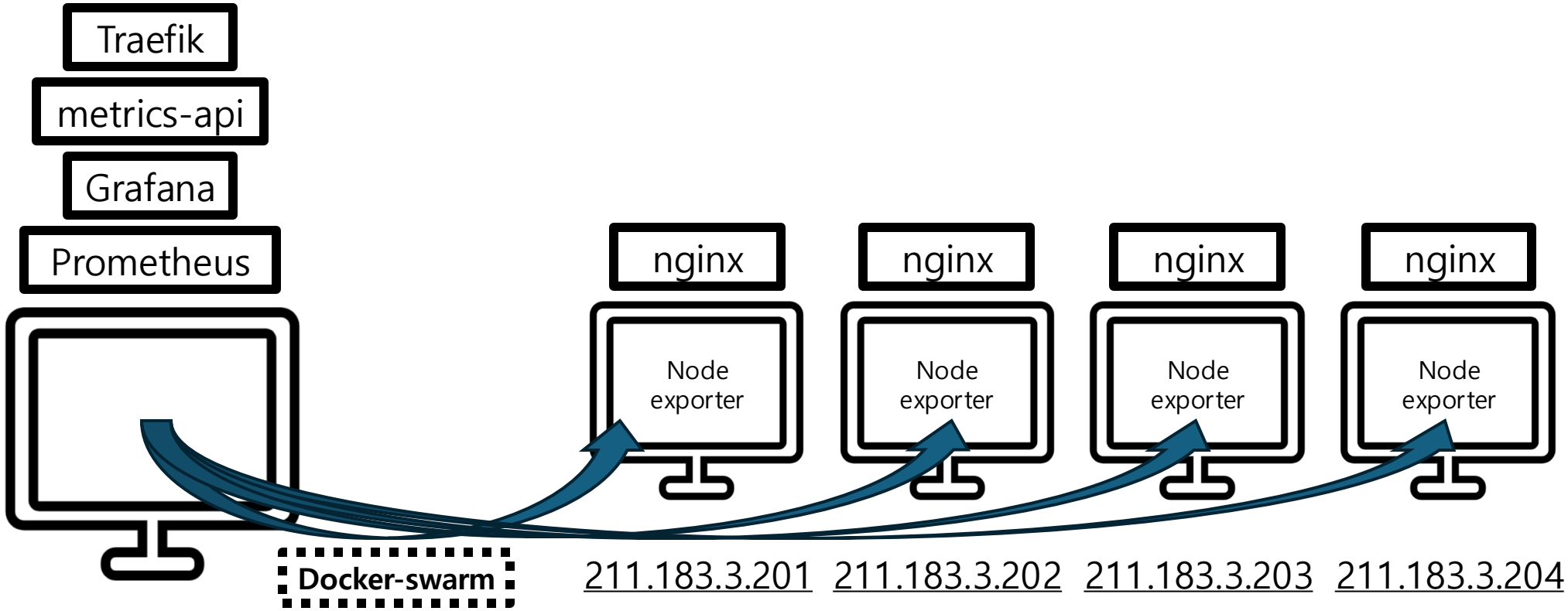
- Prometheus에서 수집한 CPU/RAM 사용량을 바탕으로 점수 계산
- 사용자의 니즈에 따라 가중치를 따로 설정 가능
- 가장 여유 있는 노드를 best_node로 선정

Metrics-API 설정

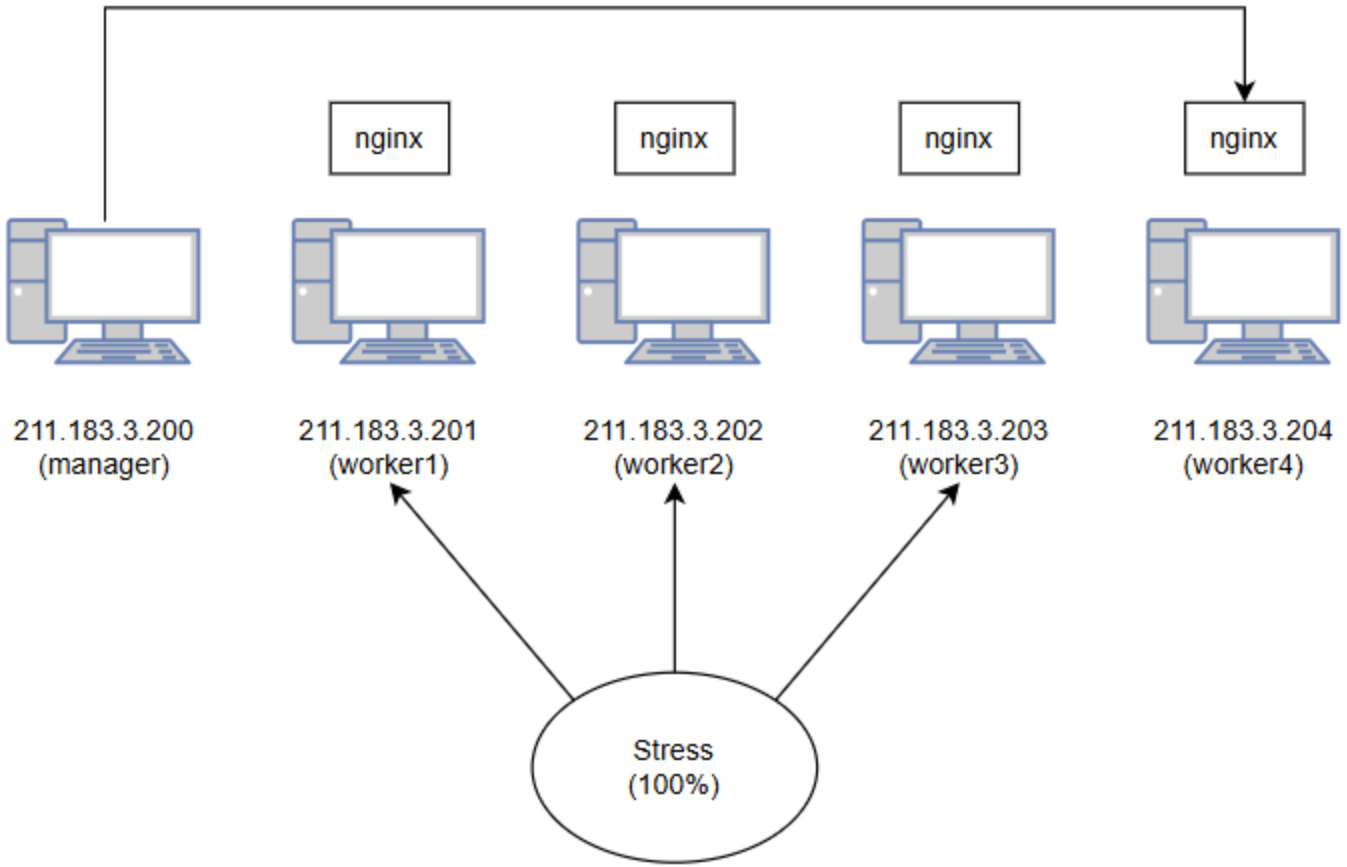
```
# dynamic_conf.yml 생성
def generate_dynamic_config():
    best_node = get_best_node()

    if best_node and ":" in best_node:
        best_ip = best_node.split(":")[0]
    else:
        best_ip = best_node or "localhost"

    print(f"[DEBUG] Best IP: {best_ip}")

    return f"""\
http:
  routers:
    dynamic-router:
      rule: "PathPrefix(`/`)"
      service: dynamic-service
      entryPoints:
        - web

    dashboard:
      rule: "PathPrefix(`/dashboard`) || PathPrefix(`/api`)"
      service: api@internal
      entryPoints:
        - api

  services:
    dynamic-service:
      loadBalancer:
        servers:
          - url: "http://{best_ip}:8020"
"""
```

- 선택된 best_node의 IP를 기반으로 Traefik의 라우팅 설정 파일 생성
- 사용자의 요청이 가장 여유 있는 노드에게로 전달
- PathPrefix("/")는 기본 경로에 대한 라우팅 정의
- api@internal은 traefik 대시보드에 대한 경로를 정의

테스트 환경

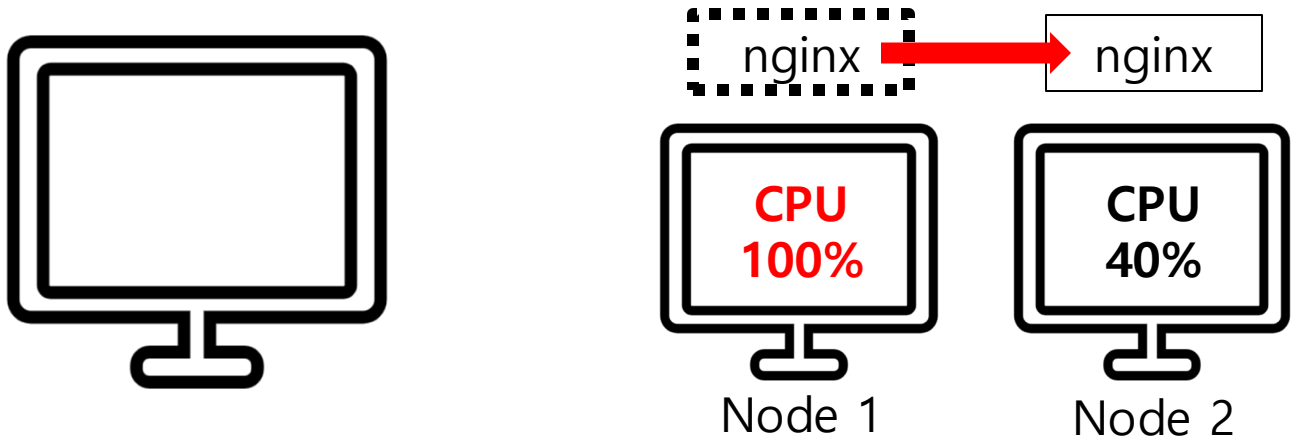

Traefik
metrics-api
Grafana
Prometheus
nginx
nginx
nginx
nginx
Node exporter
Node exporter
Node exporter
Node exporter
Docker-swarm
211.183.3.201
211.183.3.202
211.183.3.203
211.183.3.204

테스트 방식


nginx
nginx
nginx
nginx
211.183.3.200
(manager)
211.183.3.201
(worker1)
211.183.3.202
(worker2)
211.183.3.203
(worker3)
211.183.3.204
(worker4)
Stress
(100%)

추후 개선 계획

① (Docker Swarm 내 배포된 컨테이너가 하나일 경우)
CPU 사용량에 따라 **컨테이너를 자동 migration** 할 수 있는 기능 추가



추후 개선 계획

② 다양한 어플리케이션에 적용시킬 수 있도록
맞춤형 PromQL 구성

CPU-Bound	- 스트리밍 처리 / 실시간 변환 - 멀티스레드로 작은 작업 반복 - AI inference에서 모델이 작을 때 - 디스크 I/O + 압축/암호화 - Busy-wait or tight loop
RAM-Bound	- 데이터 캐싱 서비스 - 대용량 모델 로딩 (AI, NLP) - 파일 시스템 캐시 / 메모리 매핑 - Python 대형 리스트 / 딕셔너리 등 데이터 - 웹 서버에서 커넥션 / 세션 관리 - DB 서버 (예: PostgreSQL, MySQL) - 이미지, 영상, 텐서 등 대용량 버퍼

프로젝트 기대효과

- Docker Swarm 환경에서 불필요한 CPU 과부화를 막고 **물리자원 안정화, 비용 절감**으로 이어질 수 있음

- 구조 자체가 가벼운 것이 특징으로, 쿠버네티스 베이스가 아닌 **소규모 프로젝트**에 적합할 것으로 판단
  * PoC 등 로컬 마이크로서비스 테스트
  * 자원이 한정적인 Edge Computing

감사합니다

부록

CPU 위주 Application

Application	설명
계산량 많은 연산 (CPU-bound)	RAM에 많은 데이터를 올리진 않지만, 반복적인 계산이 계속 돌고 있을 때 (예: 숫자 계산, 압축, 해싱, 인코딩)
스트리밍 처리 / 실시간 변환	영상이나 오디오 실시간 인코딩/디코딩은 CPU를 많이 쓰지만, 한 번에 처리하는 데이터 양은 작아서 RAM은 적게 사용
멀티스레드로 작은 작업 반복	많은 CPU 코어를 활용하지만, 각 스레드가 사용하는 메모리는 적을 때
AI inference에서 모델이 작을 때	모델이 작아서 RAM에는 크게 부담 없지만, 추론 연산 자체가 CPU에 부담
디스크 I/O + 압축/암호화	예: 파일 업로드 시 압축 or 암호화. CPU는 바쁜데 RAM은 그다지 안 씀
Busy-wait or tight loop	코드 상에서 sleep 없이 루프가 돌아가면서 CPU만 계속 쓰는 경우 (버그성 코드 포함)

RAM 위주 Application

Application	설명
데이터 캐싱 서비스	Redis, Memcached, Varnish처럼 데이터를 메모리에 통째로 올려놓는 서비스들. CPU는 요청 처리만 잠깐 쓰고, 대부분 RAM만 점유
대용량 모델 로딩 (AI, NLP)	모델은 메모리에 미리 올라가 있지만, 추론 요청이 적으면 CPU는 놀고 RAM만 계속 사용 중
파일 시스템 캐시 / 메모리 매핑	OS가 자주 쓰는 파일을 미리 메모리에 올려두는 경우. CPU는 안 쓰는데 RAM 사용률은 높음 (mmap, page cache 등)
Python 대형 리스트/딕셔너리 등 데이터 구조	프로세스는 거의 계산 안 하는데, 메모리상에 큰 데이터 구조를 들고 있을 때
웹 서버에서 커넥션/세션 관리	Node.js, Django, Flask 등에서 많은 사용자 세션을 메모리에 올려놓는 경우. CPU는 유휴 상태일 수 있음
DB 서버 (예: PostgreSQL, MySQL)	DB는 많은 데이터를 메모리에 캐시함. 쿼리가 없을 땐 CPU는 놀지만 메모리는 계속 사용 중
이미지, 영상, 텐서 등 대용량 버퍼	RAM에 미디어 데이터를 올려놓고 대기하는 구조. 실시간 처리가 없으면 CPU 사용 거의 없음