



BINARY  
SEARCH

# 이차원 이진 탐색 기반 초기 컨테이너 리소스 할당 최적화 시스템

김두현, 조재영



# Overview

- 프로젝트 개요
- 시스템 아키텍쳐
- 각 모듈 소개
- 향후 계획 및 결론
- Q&A

# 프로젝트 개요

## Why?

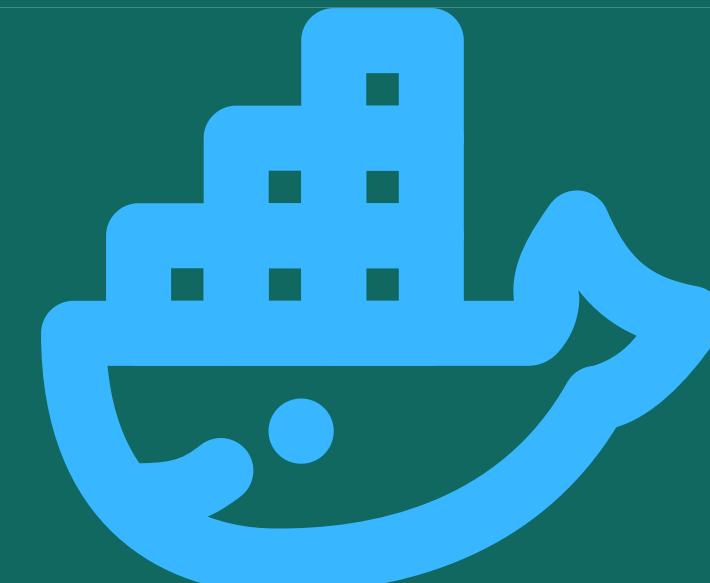
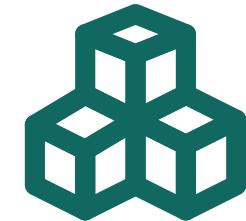
현대 클라우드 환경에서 컨테이너는 애플리케이션 배포의 핵심이 되었지만,  
리소스 할당이 최적화되지 않으면 초기 컨테이너 할당시 불필요한 비용이 발생



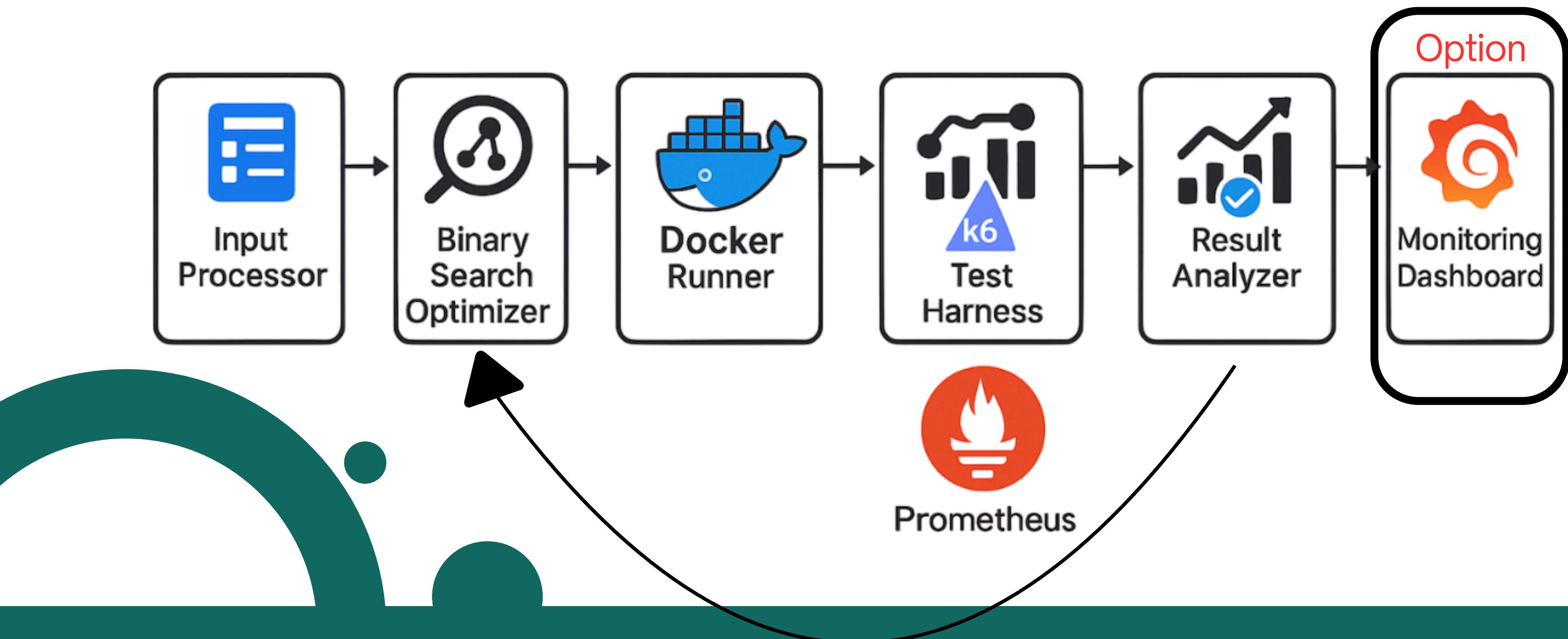
클라우드 환경에서 컨테이너를 운영할 때  
자원(cpu, memory)을 **최적화**하여  
비용 절감을 이루는 자동화된 알고리즘 개발

### 최종 목표

서비스 요청( $n$ 회)을 정해진 시간( $n$ 초) 이내로  
모두 처리하는 최소 리소스를 가진  
컨테이너 추천 알고리즘 개발



# System Architect



# Input Processor

input.json



```
{  
    "requests": 1000,  
    "deadline_ms": 3000,  
    "cpu_range": [0.5, 1, 2, 4],  
    "mem_range": [256, 512, 1024, 2048, 4096],  
    "unit_cpu_cost": 10,  
    "unit_mem_cost": 8  
}
```

## 입력 파싱 및 검증 단계

- 테스트 요청 수
- 테스트 통과 기준
- cpu 리스트
- memory 리스트
- 단위 cpu 비용
- 단위 메모리 비용

# Binary Search Optimizer

최소한의 리소스로 주어진 요청을 처리할 수 있는  
최적의 CPU/Memory 조합을 빠르게 찾아주는 핵심 알고리즘 모듈

## 기본 아이디어

- CPU 수, 메모리 크기로 구성된 2차원 배열 정의
- 각 리소스를 기준으로 오름차순으로 col/row 정렬
- CPU 루프 내에서 메모리 크기를 이진 탐색(Binary Search)으로 탐색하며, 각 CPU 값에 대해 메모리 값의 최소값과 최대값 사이에서 최적 지점을 빠르게 탐색

⇒ 추정 시간 복잡도:  $O(CPU \times \log(Memory))$

CPU ↓ \\ Memory →	128MB	256MB	512MB	1024MB	2048MB	4096MB
1.0	X	X	X	X	X	O
2.0	X	X	X	O	O	O
3.0	X	X	O	O	O	O
4.0	X	X	O	O	O	O
5.0	X	X	O	O	O	O
6.0	X	O	O	O	O	O

# Binary Search Optimizer

## 💡 상세 알고리즘

- CPU\_n, Memory\_k) 조합으로 테스트했을 때 deadline안에 요청 수를 만족하면 성공으로 간주
- 각 탐색 단계에서 k6 테스트를 실행하고, 해당 조합의 성능 결과를 받아 조건을 만족하는지를 판단
- 메모리 이진 탐색 중 성공한 조합이 발견되면 좌측(mid-1) 범위로 이동하여 더 낮은 자원으로 가능한지 추가 확인
- 실패한 경우는 우측(mid+1) 범위로 이동하며 기준을 만족할 수 있는 최소 메모리를 찾음
- 다음 cpu 사이클로 넘어가기 전에 해당 단계에서 성공한 조합 중 memory index가 가장 낮은 케이스를 Cost Function에 계산 후 Min Heap에 삽입



## 탐색 범위 최적화

### 가정

(n, k) 리소스를 가진 컨테이너가 응답 성공했으면,  
(n+1, k) or (n, k+1) 리소스를 가진 컨테이너 또한 통과할 거라고 기대

“이전 CPU 값에서 찾은 최소 메모리 인덱스를 다음 CPU 값의 탐색 범위로 제한”

# Binary Search Optimizer

## Cost Function

$$\text{Cost} = (\text{단위 CPU cost}) * \text{cpu\_n} + (\text{단위 mem cost}) * \text{mem\_size}$$

### 비용 함수란?

- 최종 cpu, memory 조합을 pick하기 위한 도구
- 테스트에 통과한 (cpu, mem) 조합을 cost로 계산

### 단위 cpu cost, 단위 mem cost

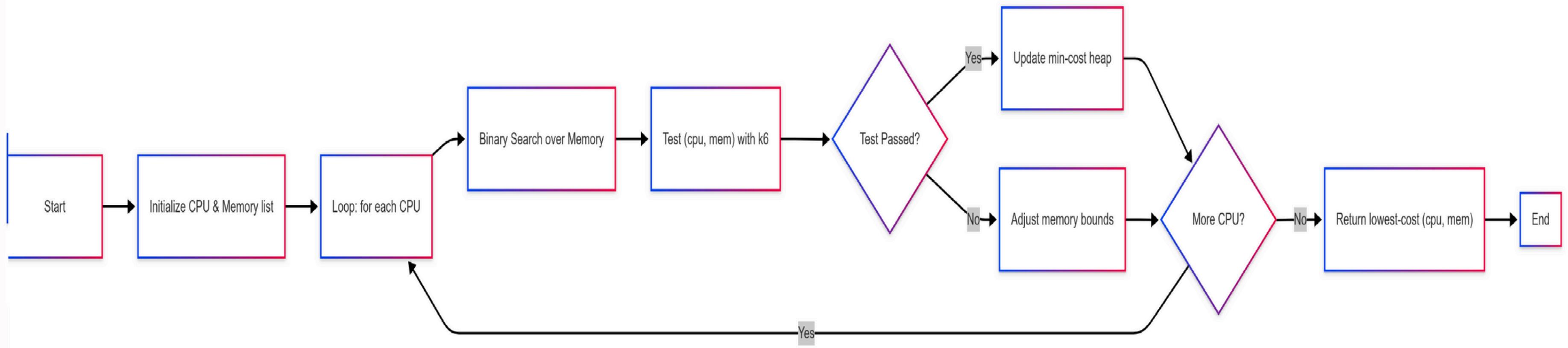
- 실제 유명 클라우드 회사(AWS, GCP, Azure, etc ...)의 instance 요금 정책을 기반으로 각 리소스(cpu, memory) 별 단위 cost 비용을 추정
- 예시 단위 비용 (실제 요금 추정 제외, 상대적 비율로 단순화)>
  - 단위 CPU cost = 10 (1 vCPU당)
  - 단위 mem cost = 8 (1024MB당)

775개의 사용 가능한 인스턴스 보기					
인스턴스 이름	온디맨드 시간당 요금	vCPU	메모리	스토리지	네트워크 성능
t4g.nano	USD 0.0042	2	0.5GiB	EBS 전용	최대 5기가비트
t4g.micro	USD 0.0084	2	1GiB	EBS 전용	최대 5기가비트
t4g.small	USD 0.0168	2	2GiB	EBS 전용	최대 5기가비트
t4g.medium	USD 0.0336	2	4GiB	EBS 전용	최대 5기가비트
t4g.large	USD 0.0672	2	8GiB	EBS 전용	최대 5기가비트
t4g.xlarge	USD 0.1344	4	16GiB	EBS 전용	최대 5기가비트
t4g.2xlarge	USD 0.2688	8	32GiB	EBS 전용	최대 5기가비트
t3.nano	USD 0.0052	2	0.5GiB	EBS 전용	최대 5기가비트
t3.micro	USD 0.0104	2	1GiB	EBS 전용	최대 5기가비트

<AWS EC2 요금 정책>

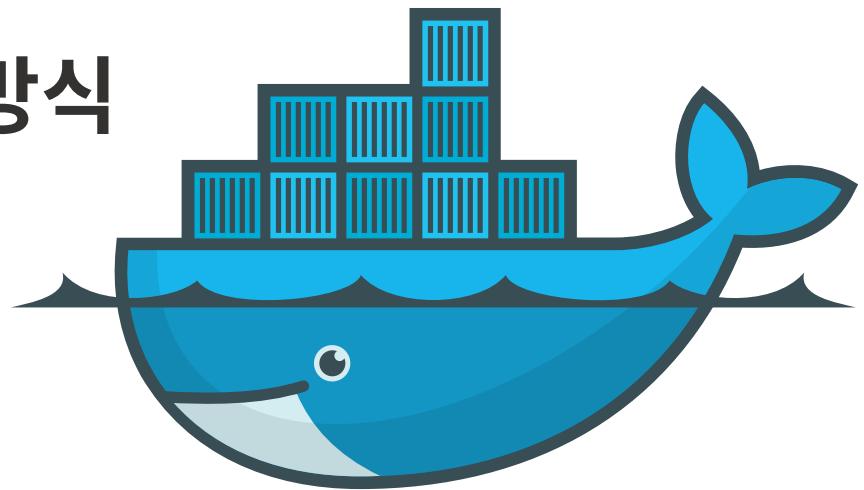
# Binary Search Optimizer

## (Flow Chart)



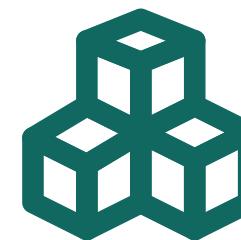
# Docker Runner

## 🐳 컨테이너 실행 방식



이진 탐색 알고리즘에 의해 결정된 CPU/Memory 리소스 조합을 기반으로 실제 테스트용 컨테이너를 실행하는 모듈

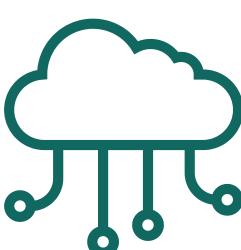
- k6 부하 테스트를 위한 API server를 컨테이너화



각 실행 시점에 할당할 CPU 및 메모리 자원을 환경변수로 전달하여 리소스 제한이 적용된 컨테이너가 구동

```
CPU_ALLOC=1 MEM_ALLOC=1024 docker-compose up -d api
```

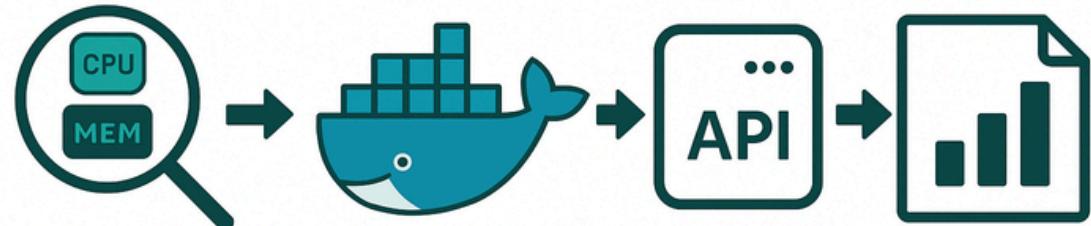
## docker runner 실행 흐름



1. 이진 탐색 모듈이 탐색 중인 조합 (cpu, mem)을 설정
2. run\_docker\_container(cpu, mem) 함수 실행  
→ child process로 docker 명령 수행
3. 컨테이너 내부의 api.js 서버가 k6 테스트를 대기
4. k6가 설정된 부하를 발생시켜 해당 API에 요청 수행
5. 테스트 완료 후, 컨테이너 제거 및 결과 수집

# Docker Runner

## 🐳 dockerRunner.js 예시 코드



## Docker Runner

```
// dockerRunner.js
const { exec } = require("child_process");

function run_docker_container(cpu, mem) {
  return new Promise((resolve, reject) => {
    const CPU_ALLOC = cpu;
    const MEM_ALLOC = mem;
    const command = `CPU_ALLOC=${CPU_ALLOC} MEM_ALLOC=${MEM_ALLOC} docker-compose up -d api`;

    console.log(`[RUN] ${command}`);
    exec(command, (error, stdout, stderr) => {
      if (error) {
        console.error(`🚨 Docker run error: ${stderr}`);
        return reject(error);
      }
      console.log(`✓ Docker container started: ${stdout}`);
      resolve(stdout);
    });
  });
}
```

# Test Harness

부하 테스트 자동화 모듈

탐색된 리소스 조합에 대해 k6를 활용해 API 부하 테스트를 자동으로 실행하고, 결과를 수집하는 모듈



- JavaScript 기반의 부하 테스트 도구
- API 성능 테스트에 특화
- 결과를 자동으로 수집/분석하는 CLI 도구



```
→ developerexperience k6 run loadtesting.js
  _/\_ /--/ /--/
  | | | / \ / /
  | | | \ \ \ \ \ \ \ \ \ .io

execution: local
script: loadtesting.js
output: -

scenarios: (100.00%) 1 scenario, 10 max VUs, 50s max duration (incl. graceful stop):
* default: 10 looping VUs for 20s (gracefulStop: 30s)

running (20.3s), 00/10 VUs, 800 complete and 0 interrupted iterations
default ✓ [=====] 10 VUs 20s

  data_received.....: 191 kB 9.4 kB/s
  data_sent.....: 79 kB 3.9 kB/s
  http_req_blocked....: avg=18.62µs min=1µs med=4µs max=1.18ms p(90)=8µs p(95)=10µs
  http_req_connecting....: avg=5.17µs min=0s med=0s max=477µs p(90)=0s p(95)=0s
  http_req_duration....: avg=2.71ms min=760µs med=2.43ms max=19.46ms p(90)=3.69ms p(95)=4.61ms
  http_req_receiving....: avg=53.45µs min=22µs med=44µs max=429µs p(90)=89µs p(95)=111.04µs
  http_req_sending....: avg=24.22µs min=6µs med=18µs max=839µs p(90)=36µs p(95)=48.04µs
  http_req_tls_handshaking...: avg=0s min=0s med=0s max=0s p(90)=0s p(95)=0s
  http_req_waiting....: avg=2.63ms min=667µs med=2.36ms max=19.34ms p(90)=3.62ms p(95)=4.5ms
  http_reqs.....: 800 39.449383/s
  iteration_duration....: avg=253.06ms min=250.88ms med=252.72ms max=272.47ms p(90)=254.01ms p(95)=254.96ms
  iterations.....: 800 39.449383/s
  vus.....: 10 min=10 max=10
  vus_max.....: 10 min=10 max=10

→ developerexperience █
```

테스트 시간, 요청 수 등을 설정하기 용이해  
손쉽게 docker runner로 생성된  
API 서버 컨테이너의 성능을 테스트 가능

# Result Analyzer

테스트 결과 평가 및 조합 선정

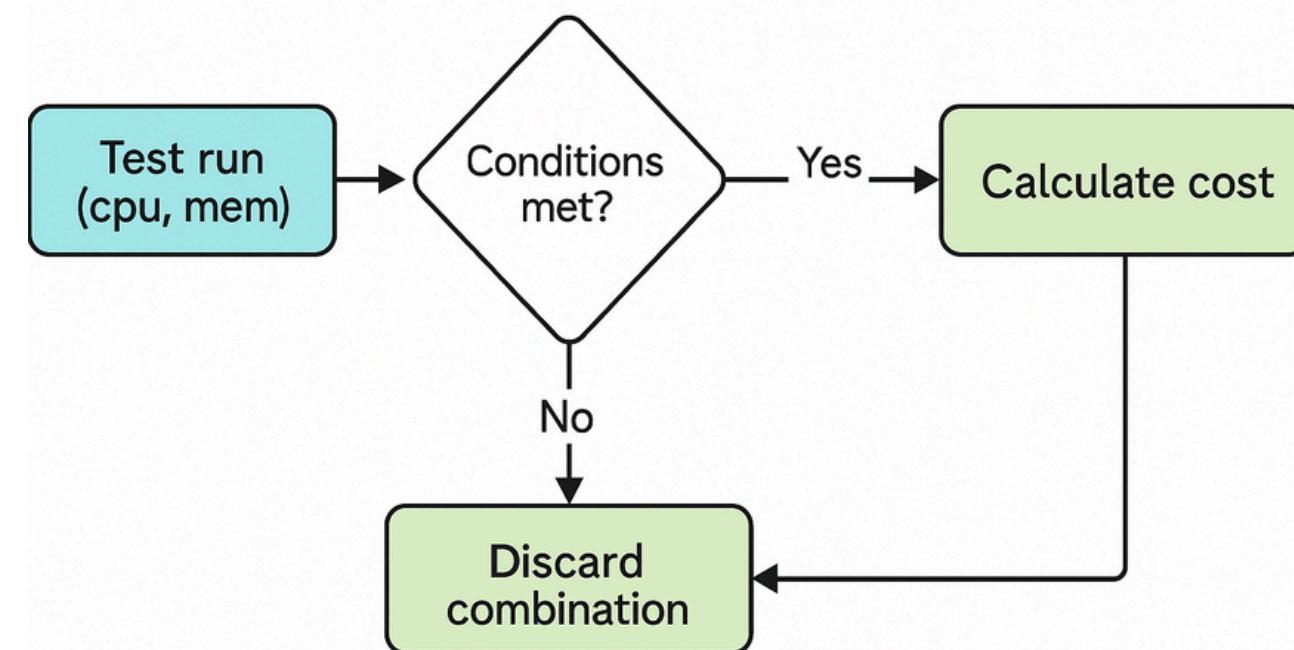
k6 테스트 결과를 기반으로, 해당 자원 조합이 성능 기준을 만족하는지  
평가하고 비용 계산 후 최적 조합을 선택하는 분석 모듈

- ◆ 평가 기준
- ✓ 평균 응답 시간  $\leq$  deadline\_ms
- ✓ 누적 요청 수  $\geq$  요청 목표량(requests)
- ✓ 오류율 또는 실패율이 낮을 것(OOM Killed)



Test Harness를 통해 측정된 결과 메트릭스를 수집하고 컨테이너의 리소스 사용량을 수집

수집된 결과를 Cost Function을 통해 비용을 계산하고 Min Heap에 삽입 → (cost, (cpu\_n, mem\_k))



# 결론

## 향후 계획

### ⚙️ 병렬 테스트 실행

→ 테스트 시간 단축, 대규모 탐색 최적화

### 🧠 워크로드 다양화

→ API 외에 DB, 배치 작업, 메시지 큐 등 실험 가능

## 결론

### 관점

### 의미



최소 리소스로도 SLA 만족 → 클라우드 비용 감소



탐색 + 테스트 + 분석까지 자동화된 검증 프로세스



K8s 없이도 구성 가능, 로컬 & 클라우드 양쪽 대응

부하 기반 자원 최적화 문제를 비용과 성능 사이, 최적의 균형을 찾는 자동화 테스트 프레임워크

# Q&A