eBPF Serverless

Cold Start & Hot Passion

F4 | 김선희 : 김준엽 : 오진우 : 이예은

Contents

팀원소개

이름	역할
김준엽	PM, DevOps
김선희	eBPF 커널 핸들링
오진우	Serverless Infra, 웹 프로그램
이예은	유저 공간 실험 로직

프로젝트 개요

프로젝트 목표

Cold Start 완화를 위한 eBPF 활용:

eBPF를 활용하여 서버리스 환경의 Cold Start를 분석하고,

zero-copy를 도입하여 시스템 효율성을 극대화한다.

프로젝트 목표

Cold Start 완화를 위한 eBPF 활용:

eBPF를 활용하여 서버리스 환경의 Cold Start를 분석하고,

zero-copy를 도입하여 시스템 효율성을 극대화한다.

서버리스 (Serverless)

서버가 없다?

단어 그대로 서버가 없는 것을 뜻하는게 아닌,
우리가 직접 서버를 관리하지 않아 신경 쓸 필요가 없는 경우!

클라우드 서비스들은 서버를 사용하지 않고 그냥 가동만 시켜도 시간마다 결제 ⇒ 크던 작던 손실이 발생

그래서 등장한 게 서버리스. 내가 사용한 만큼만 비용을 지불하고 싶어!!

기존 서버 방식 vs. 서버리스

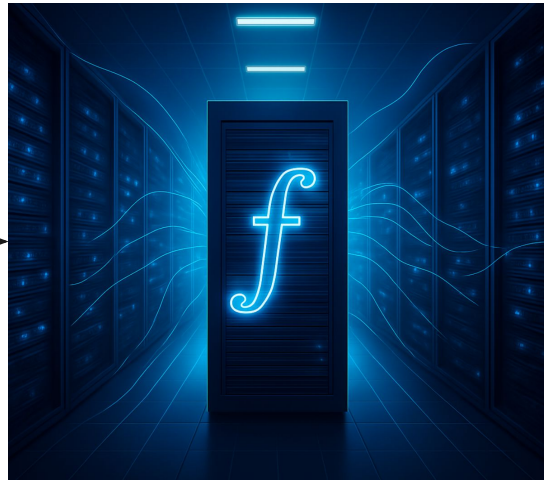
Server-Based

서버 관리 용이

확장성 증대

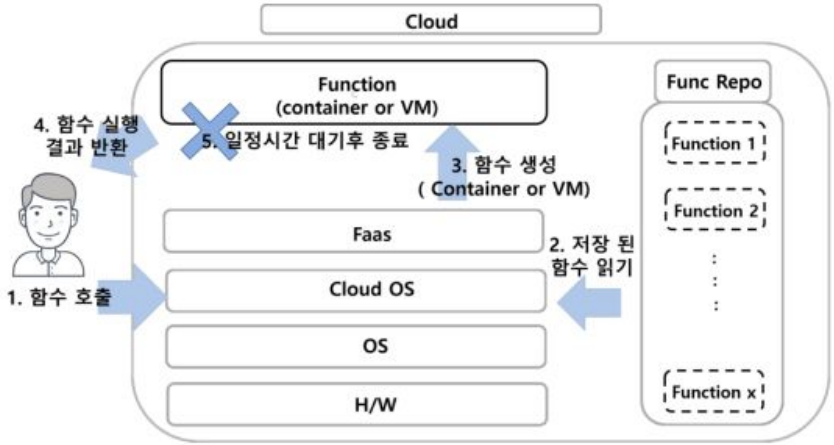
비용 절감

Serverless



콜드 스타트 (Cold Start)

예시: AWS Lambda



콜드 스타트 (Cold Start)

서버가 항시 요청에 대기하고 있는게 아니다보니,
잠들어있는 함수를 다시 깨울 때까지 시간이 걸린다

콜드 스타트가 왜 문제일까?

콜드 스타트? 그거 조금 기다리면 되는 거 아니야?

프로젝트 규모가 작다면 넘어갈 수 있겠지만
규모가 커지거나 빠른 속도를 요구하는 프로젝트라면 서버리스는 좋은 선택이 아닐 수 있다.

*아마존: "로딩 속도가 **100ms** 느려질 때마다 매출이 **1%** 감소한다."*
https://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html

*구글: "페이지 로딩이 **0.5초** 지연되면 트래픽이 **20%** 감소한다."*
Marissa Mayer at Web 2.0

콜드 스타트 문제를 개선하려면?

1. 시스템 실행 시간과 병목이 일어나는 지점을 정확히 모니터링할 수 있어야 합니다.
2. 서버 함수가 뜰 때마다 대용량 설정 파일(JSON, CSV 등)을 읽어와 작업하는 과정은 효율적이지 않습니다. 따라서 이러한 정보를 바로바로 꺼내어 사용할 수 있어야 합니다.

콜드 스타트 문제를 개선하려면?

1. 시스템 실행 시간과 병목이 일어나는 지점을 정확히 모니터링할 수 있어야 합니다.
   → **eBPF 도입**: 커널 레벨에서 컨테이너 생성을 선제적으로 감지하여 Shared Memory 연결을 준비합니다.

2. 서버 함수가 뜰 때마다 대용량 설정 파일(JSON, CSV 등)을 읽어와 작업하는 과정은 효율적이지 않습니다.
   따라서 이러한 정보를 바로바로 꺼내어 사용할 수 있어야 합니다.
   → **Shared Memory를 활용한 Zero-Copy 도입**: 호스트와 컨테이너 사이의 데이터 복사 과정을 단축시킵니다.

eBPF

리눅스 커널 소스 코드를 수정하거나 커널 모듈을 로드하지 않고도, 커널 내에서 사용자 정의 프로그램을 안전하고 효율적으로 실행할 수 있는 기술 ⇒ 이러한 eBPF의 커널 가시성을 이용해, 다음과 같은 기능을 구현할 수 있었습니다.

모니터링

프로그램이 커널 내부에 상주하며
람다 인스턴스의 시작(sys_execve())
을 캐치.

데이터 기록

람다가 뜨는 순간, eBPF가 해당 PID를
공유 메모리에 즉시 등록.

공유 메모리

데이터 복사

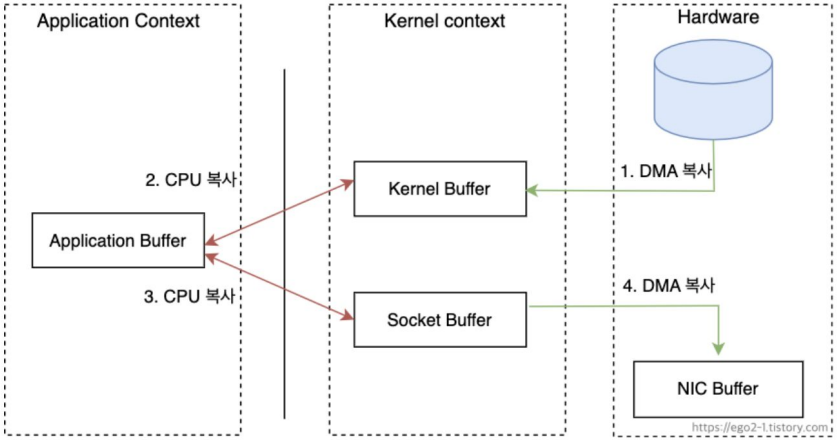
람다는 실행되자마자 공유 메모리를 확인하여,
자기가 최초 시작인지 아닌지,
어떤 데이터를 읽어야 하는지 등
정보를 바탕으로 즉시 동작할 수 있게 된다.

증명된 기술 eBPF

프로젝트	설명	GitHub Star
kindling	eBPF-based Cloud Native Monitoring Tool	1.1k
falco	Cloud Native Runtime Security	8.8k
loxilb	eBPF based cloud-native load-balancer for Kubernetes | Edge | Telco | IoT | XaaS	1.8k
calico	Cloud native networking and network security	7.1k
huatuo	A cloud-native operating system observability project based on eBPF, incubated under CCF	1.2k

표: GitHub 기준 eBPF를 이용한 클라우드 서비스와 star 수

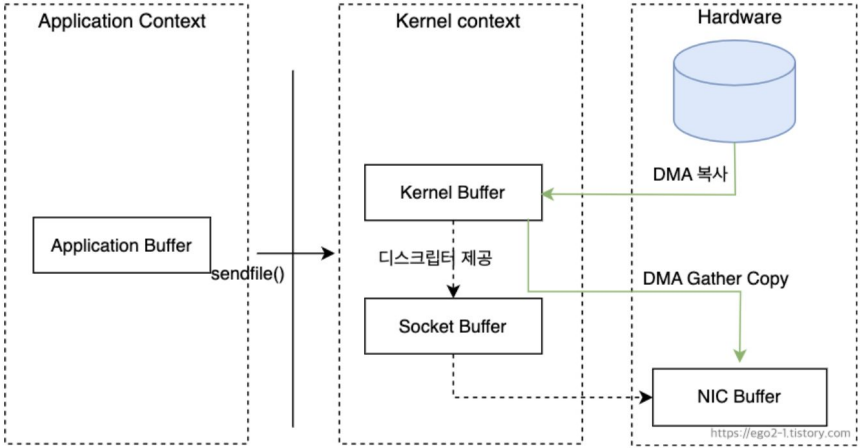
Zero-Copy



[기존 방식]

디스크 → 커널 버퍼 → 유저 어플리케이션 버퍼 → 소켓 버퍼

(여러 번의 복사 발생)



[공유 메모리를 활용한 Zero-Copy 방식]

디스크 → 커널 버퍼 → 소켓 버퍼

(유저 영역 복사를 건너뛰거나, 주소만 참조)

시장 분석 | Target

프로젝트 규모가 작다면 넘어갈 수 있겠지만
규모가 커지거나 빠른 속도를 요구하는 프로젝트라면 서버리스는 좋은 선택이 아닐 수 있다.

따라서 MVP단계 ~ 투자 직전 단계 (금전적 한계)에 있는 일반적인 커뮤니티나 쇼핑몰 도입이 목표!

시장 분석 | Compare

Cilium

Hubble

Hubble

eBPF **모니터링**에 초점

But!

Our Project

개발/테스트 과정에서 **모니터링 및 솔루션 제공**

⇒ 서버리스의 Cold Start를

eBPF로 분석하여 성능 및 효율 향상

시장 분석 | SWOT 분석

Strength

- MINISTACK 기반 → 저비용 로컬 재현 가능
- eBPF 기반 커널 수준 핸들링
- 기업 수준에서 진입하기 힘든 니치 분야
- 서버리스 Cold Start라는 명확한 문제 정의

Weakness

- 소규모 입력 데이터
  → 실제 가치 증명은 정량적 성능 개선 데이터가 필요
- Ubuntu/Lambda 타겟
  → 개발환경에 맞는 별도 설정 필요

SWOT 분석

Opportunities

- 서버리스는 쉬운 확장과 운영 단순화 때문에 꾸준히 수요 증대
- 운영자들의 Cold Start 문제 개선과 시스템 레벨 메트릭 관측에 대한 니즈

Threats

- 서버리스 플랫폼의 개선으로 Cold Start 문제가 완화될 가능성이 존재

프로젝트 상세

기술 스택

사용 언어

C

사용 환경

eBPF

Oracle Cloud

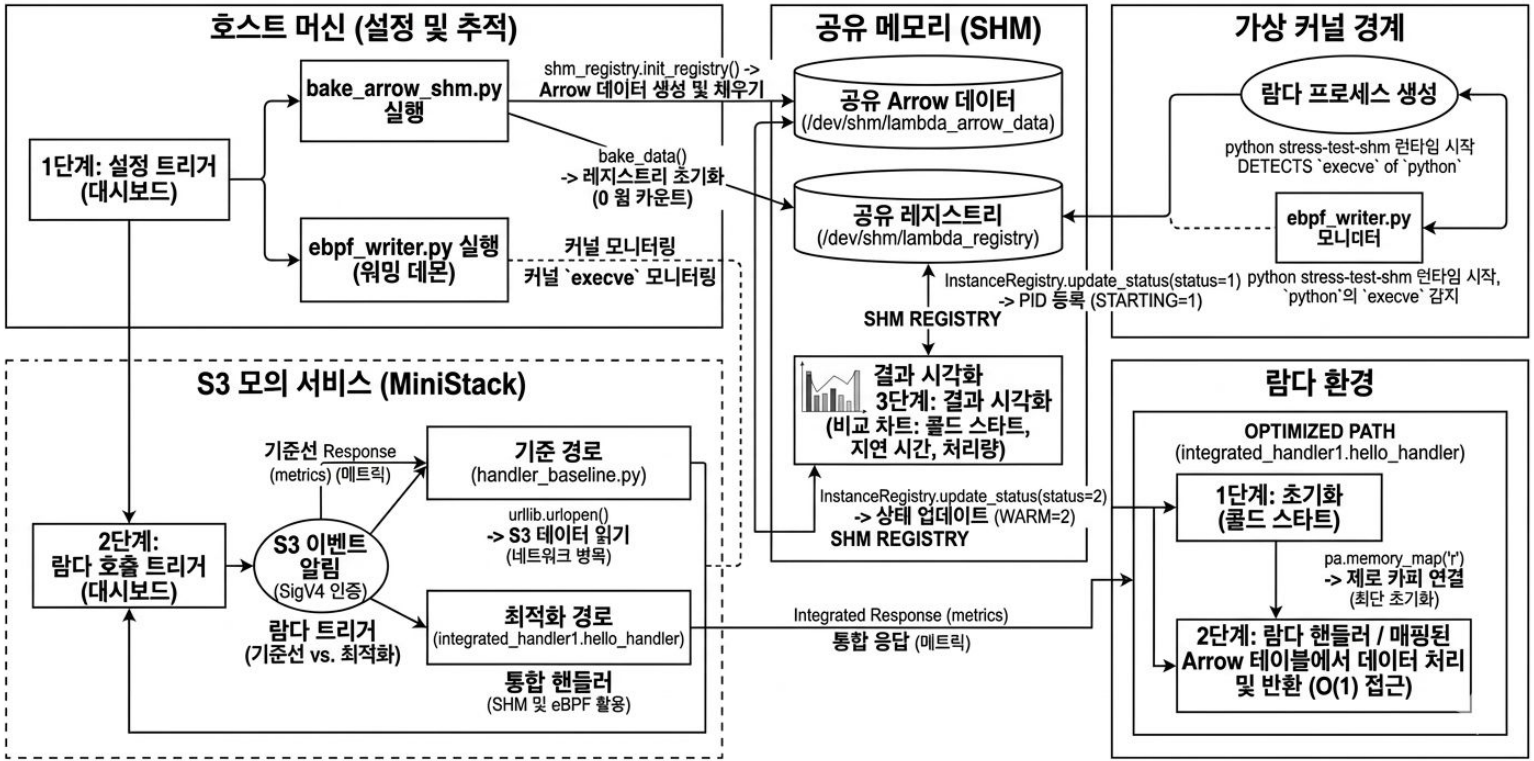
MINISTACK

Core Tech

커널의 소스 코드를 수정하지 않고도
커널 수준 프로그램을
주입 및 실행할 수 있게 해주는 기술

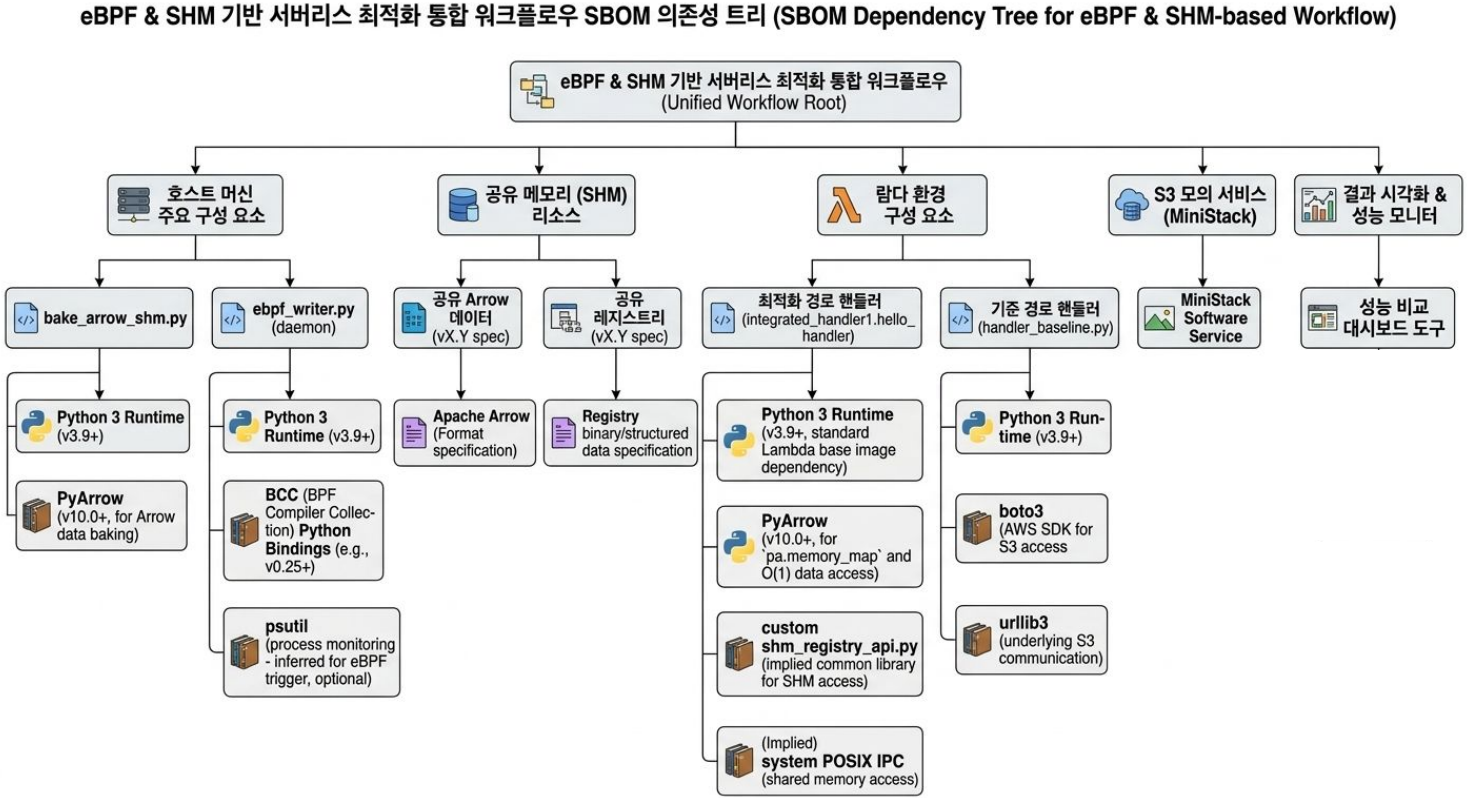
AWS와 같은 클라우스 서비스 환경을
사용자의 로컬 컴퓨터에서 Moking하여
실행할 수 있게 해주는 도구

설계 | Workflow

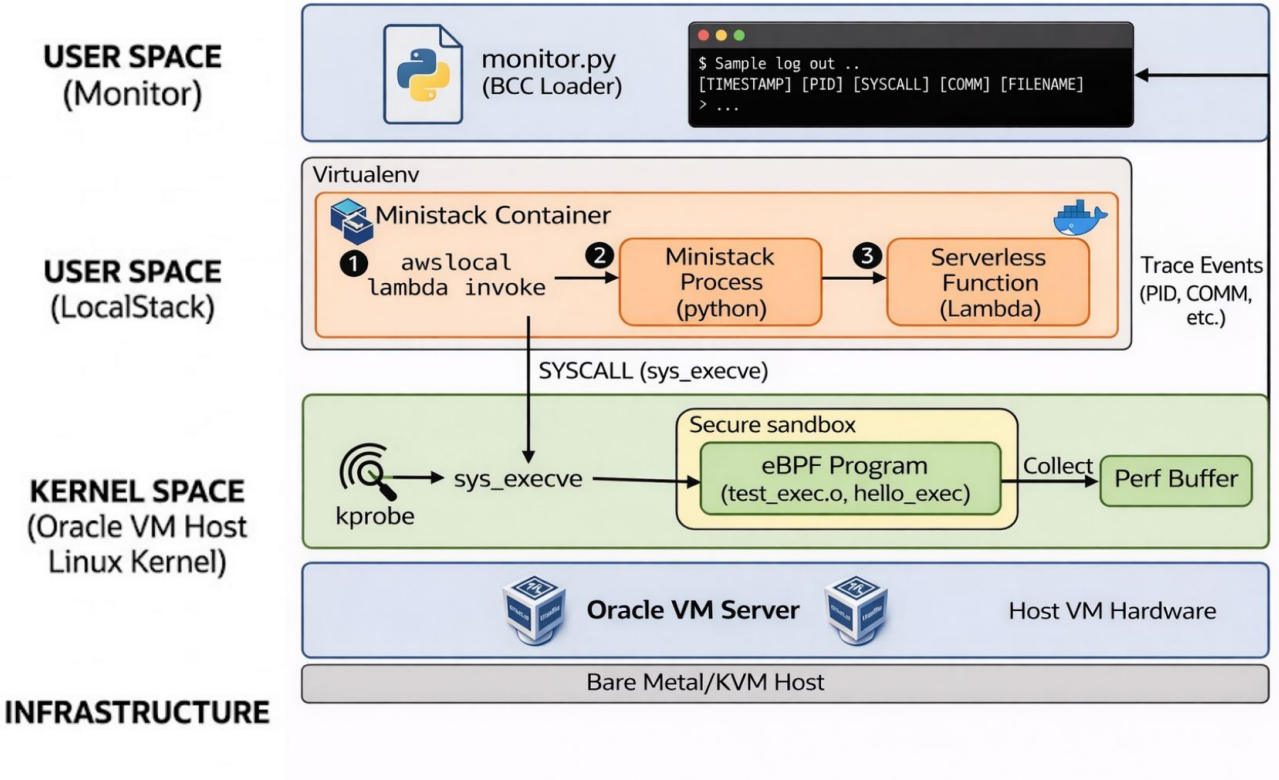

호스트 머신 (설정 및 추적)
1단계: 설정 트리거 (대시보드)
bake_arrow_shm.py 실행
ebpf_writer.py 실행 (워밍 데몬)
shm_registry.init_registry() -> Arrow 데이터 생성 및 채우기
bake_data() -> 레지스트리 초기화 (0 윔 카운트)
커널 모니터링
커널 `execve` 모니터링
공유 메모리 (SHM)
공유 Arrow 데이터 (/dev/shm/lambda_arrow_data)
공유 레지스트리 (/dev/shm/lambda_registry)
SHM REGISTRY
가상 커널 경계
람다 프로세스 생성
python stress-test-shm 런타임 시작
DETECTS `execve` of `python`
ebpf_writer.py 모니터
InstanceRegistry.update_status(status=1)
-> PID 등록 (STARTING=1)
python stress-test-shm 런타임 시작, `python`의 `execve` 감지
S3 모의 서비스 (MiniStack)
2단계: 람다 호출 트리거 (대시보드)
기준선 Response (metrics) (메트릭)
기준 경로 (handler_baseline.py)
urllib.urlopen() -> S3 데이터 읽기 (네트워크 병목)
S3 이벤트 알림 (SigV4 인증)
람다 트리거 (기준선 vs. 최적화)
최적화 경로 (integrated_handler1.hello_handler)
통합 핸들러 (SHM 및 eBPF 활용)
결과 시각화 3단계: 결과 시각화 (비교 차트: 콜드 스타트, 지연 시간, 처리량)
InstanceRegistry.update_status(status=2)
-> 상태 업데이트 (WARM=2)
SHM REGISTRY
Integrated Response (metrics)
통합 응답 (메트릭)
람다 환경
OPTIMIZED PATH
(integrated_handler1.hello_handler)
1단계: 초기화 (콜드 스타트)
pa.memory_map('r')
-> 제로 카피 연결
(최단 초기화)
2단계: 람다 핸들러 / 매핑된 Arrow 테이블에서 데이터 처리 및 반환 (O(1) 접근)

산출물 | 성능 모니터링 웹 서비스

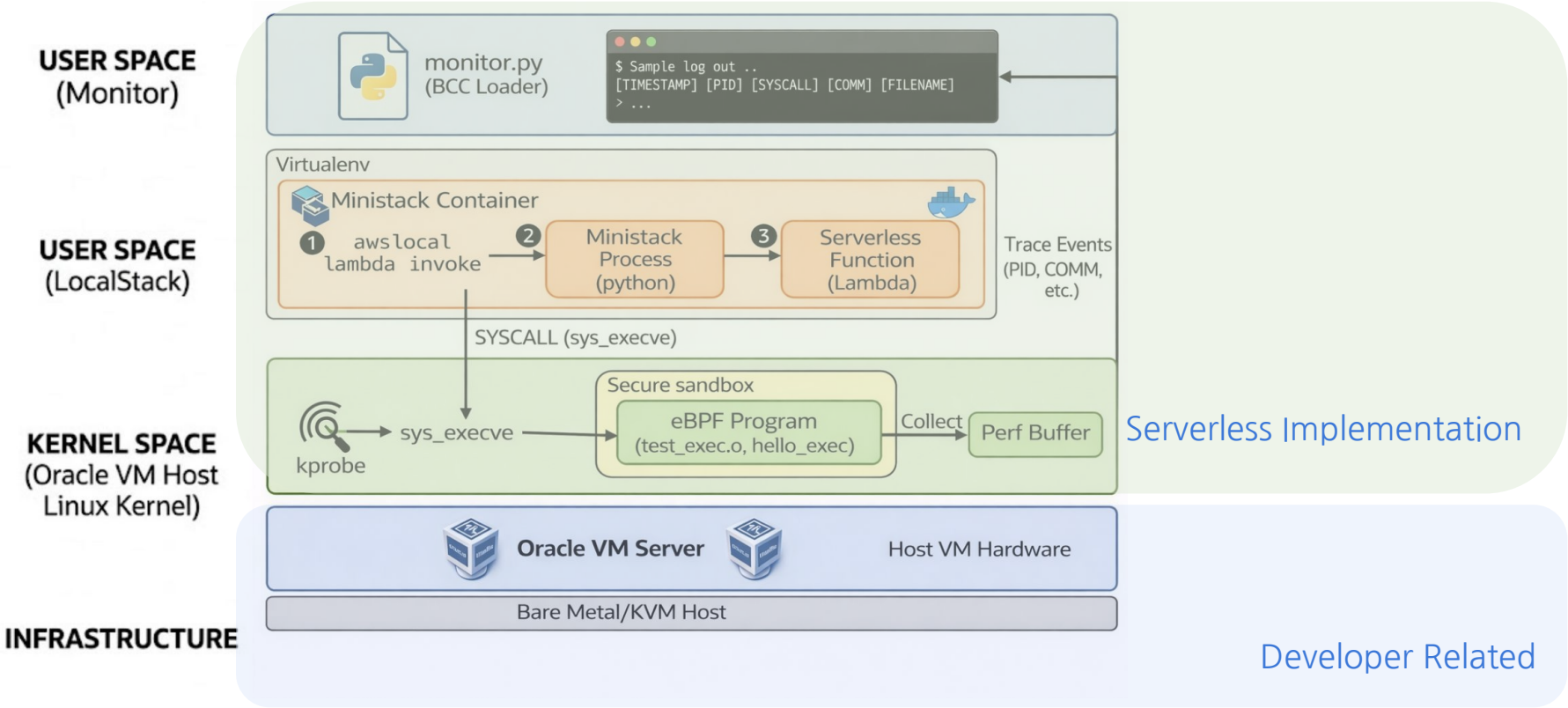
SBOM | Dependency Tree



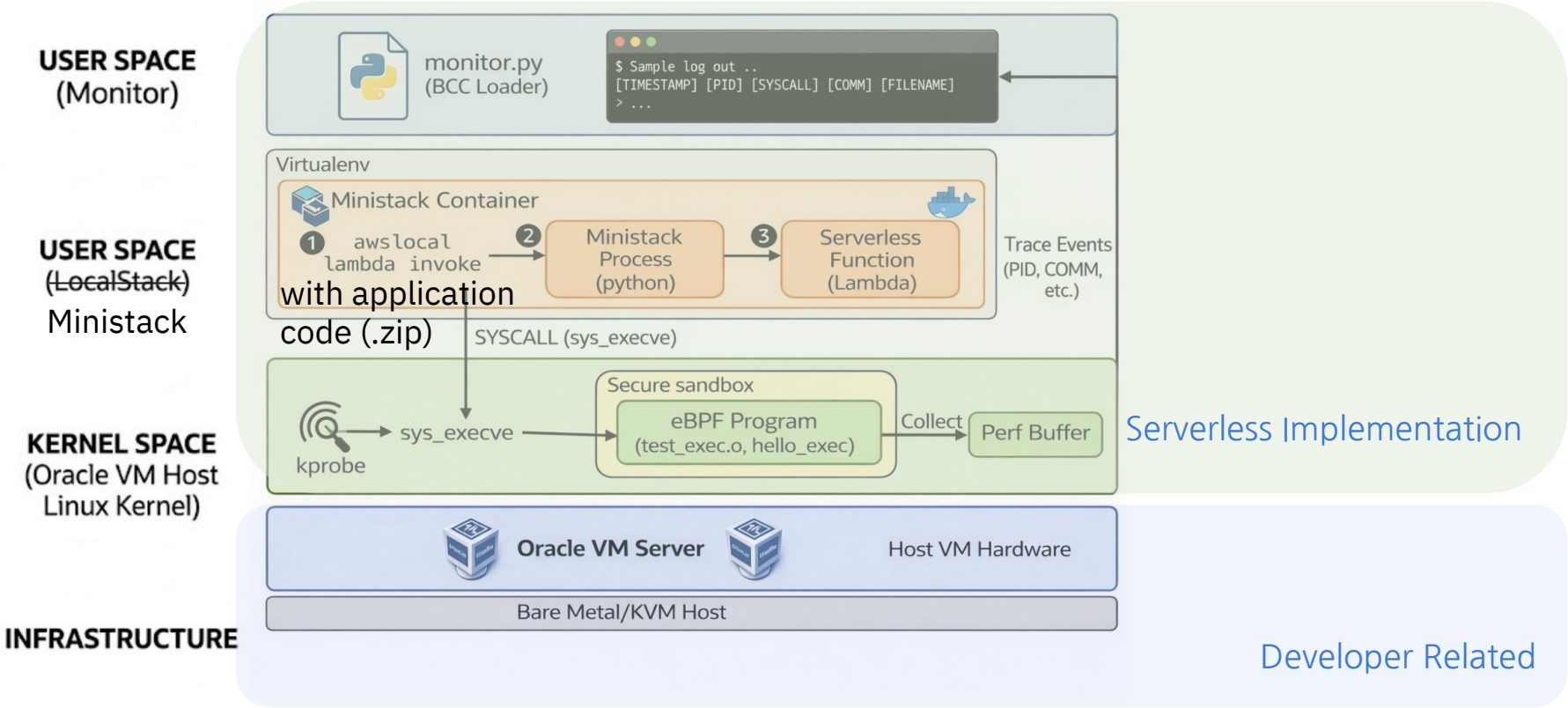
설계 | 서버리스 환경 구축


USER SPACE
(Monitor)
monitor.py
(BCC Loader)
$ Sample log out ..
[TIMESTAMP] [PID] [SYSCALL] [COMM] [FILENAME]
> ...
USER SPACE
(LocalStack)
Virtualenv
Ministack Container
1 awslocal lambda invoke
2 Ministack Process (python)
3 Serverless Function (Lambda)
Trace Events (PID, COMM, etc.)
SYSCALL (sys_execve)
KERNEL SPACE
(Oracle VM Host Linux Kernel)
kprobe
sys_execve
Secure sandbox
eBPF Program
(test_exec.o, hello_exec)
Collect
Perf Buffer
Oracle VM Server
Host VM Hardware
INFRASTRUCTURE
Bare Metal/KVM Host

서버리스 환경 구축


USER SPACE
(Monitor)
monitor.py
(BCC Loader)
$ Sample log out ..
[TIMESTAMP] [PID] [SYSCALL] [COMM] [FILENAME]
> ...
USER SPACE
(LocalStack)
Virtualenv
Ministack Container
1 awslocal lambda invoke
2 Ministack Process (python)
3 Serverless Function (Lambda)
Trace Events (PID, COMM, etc.)
SYSCALL (sys_execve)
KERNEL SPACE
(Oracle VM Host Linux Kernel)
kprobe
sys_execve
Secure sandbox
eBPF Program
(test_exec.o, hello_exec)
Collect
Perf Buffer
Serverless Implementation
Oracle VM Server
Host VM Hardware
INFRASTRUCTURE
Bare Metal/KVM Host
Developer Related

서버리스 환경 구축


USER SPACE
(Monitor)
monitor.py
(BCC Loader)
$ Sample log out ..
[TIMESTAMP] [PID] [SYSCALL] [COMM] [FILENAME]
> ...
USER SPACE
(LocalStack)
Ministack
Virtualenv
Ministack Container
1 awslocal lambda invoke
2 Ministack Process (python)
3 Serverless Function (Lambda)
Trace Events (PID, COMM, etc.)
with application code (.zip)
SYSCALL (sys_execve)
KERNEL SPACE
(Oracle VM Host Linux Kernel)
kprobe
sys_execve
Secure sandbox
eBPF Program (test_exec.o, hello_exec)
Collect
Perf Buffer
Serverless Implementation
Oracle VM Server
Host VM Hardware
INFRASTRUCTURE
Bare Metal/KVM Host
Developer Related

설계 | 파일 구조

```
.
├── 📁 baseline_src/          # [대조군] 기존 네트워크 I/O 기반 환경
│   ├── handler.py            # S3 로딩 및 데이터 파싱 (Baseline)
│   ├── Dockerfile            # 표준 런타임 이미지
│   └── docker-compose.yml    # MiniStack(S3 Mock) 인프라 설정
│
├── 📁 shm_src/               # [실험군] Zero-copy Shared Memory 기반 환경
│   ├── handler.py            # Zero-copy 핸들러
│   ├── shm_registry.py       # SHM 상태 및 lock 관리
│   ├── ringbuffer.py         # eBPF 이벤트를 위한 공유 버퍼 인터페이스
│   ├── entrypoint.sh         # 컨테이너 기동 시 SHM 매핑 스크립트
│   ├── Dockerfile            # PyArrow 및 호스트 IPC 설정 포함
│   ├── docker-compose.yml    # /dev/shm 공유 및 IPC 모드 설정
│   ├── bake_arrow_shm.py     # 호스트에서 Arrow 데이터를 SHM에 굽는 도구
│   └── shm_init.py           # SHM 레지스트리 초기화 및 메모리 할당
...
```

설계 | 파일 구조

```
.
...
├── ebpf_writer.py              # sys_execve 감시 및 SHM 기록 (커널 모니터링)
│
├── 📁 headers/                 # eBPF C 프로그램용 커널 헤더 파일 (커널 모니터링)
│
├── 📁 web_impl/                # 성능 비교 분석 대시보드
│   └── server.js               # Cold Start 개선 솔루션 성능 시각화
│
├── 📁 scripts/                 # 자동화 쉘 스크립트
│
└── 📁 dummy_data/              # 테스트용 S3 페이로드 및 Arrow dummy 데이터
```

구현 | Workflow


1
설정 트리거
대시보드
base_arrow_shm.py
공유 메모리 세팅
2
3
호출 트리거
4
결과 시각화
공유메모리 세팅
S3 (Storage) 모의 서비스
baseline_src/handler.py
I/O 발생
(기존 방식)
shm_src/handler.py
1. ebpf_writer가 프로세스 감지
2. 람다: SHM 메모리 맵핑 (Zero-copy)
3. 람다: O(1) 데이터 처리
4. SHM 레지스트리에 상태 기록 (상호 배제)

핵심 구현 | ebpf 모니터링

```
# ebpf_writer.py
# eBPF C-Program: 커널 내에서 실행

bpf_text = """
int kprobe__sys_execve(struct pt_regs *ctx) {
    char comm[16];
    bpf_get_current_comm(&comm, sizeof(comm));

    // 람다 런타임 (python 프로세스) 시작 감지
    if (comm[0] == 'p' && comm[1] == 'y') {
        u32 pid = bpf_get_current_pid_tgid() >> 32;
        // 커널에서 유저 공간(ebpf_writer.py)으로 data 전송
        events.perf_submit(ctx, &pid, sizeof(pid));
    }
    return 0;
}
"""

# Python Daemon: 커널 신호를 받아 SHM에 기록
def on_event(cpu, data, size):
    pid = struct.unpack("I", data)[0]
    # 공유 메모리 레지스트리에 STARTING(1) 상태로 기록
    registry.update_status(pid, status=1)
```

런타임 전에 구동하여, 람다 인스턴스가 시작할 때 호출되는 시스템 콜을 후킹합니다.

—

Kprobe (sys_execve):

새로운 프로세스가 실행될 때 발생하는 시스템 콜을 모니터링

비동기 상태 등록:

람다 핸들러가 로직을 수행하기 전, 커널이 미리 공유 메모리에 PID를 등록하여 **Warming** 상태로 ⇒ Mutual Exclusion

핵심 구현 | ebpf 모니터링

[모니터링 출력]

```
  >> [eBPF Detect] PID: 576156 (python) -> SHM Registry Marked as STARTING
  >> [eBPF Detect] PID: 576178 (python) -> SHM Registry Marked as STARTING
...
```

핵심 구현 | 공유 메모리 (SHM)

```
# 1. Zero-Copy 매핑
# shm_src/handler.py
def load_data_zero_copy(shm_path):
    # 파일을 여는 것이 아니라 메모리 주소를 매핑
    mmap_source = pa.memory_map(shm_path, 'r')
    # 메모리 주소에서 바로 인식 (복사 X)
    return ipc.RecordBatchFileReader(mmap_source).read_all()

# 2. 동시성 제어 및 상태 업데이트
# shm_src/shm_registry.py
def update_status(self, pid, status):
    # 파일 lock을 통해 여러 람다 간의 race condition 방지
    fcntl.flock(self.fd, fcntl.LOCK_EX)
    try:
        # SHM에 상태 기록
        offset = calculate_offset(pid)
        self.mm[offset:offset+4] = struct.pack("I", status)
    finally:
        fcntl.flock(self.fd, fcntl.LOCK_UN)
```

Deserialization 비용을 최소화하는 메모리 Direct Mapping Architecture

—

mmap (Memory Map):

파일을 메모리로 읽어오는 대신, 파일의 주소를 프로세스의 공유 메모리에 바로 연결합니다.

동시성 제어:

여러 람다가 동시에 접근할 때 발생하는 문제를 방지합니다.

시연

시연

[대조군] 네트워크 I/O 기반
vs. [실험군] Zero-Copy 기반

—

실시간 Lambda 인스턴스 실행 속도 비교를 가능하게 하는 웹 UI

시연

Step 1. Cold Start 유도: awslocal 명령어로 람다 함수를 호출하고, 첫 실행 시 발생하는 지연 상황 연출

Step 2. eBPF 실시간 감지: 커널에서 잡힌 `sys_execve()` 타임스탬프가 화면에 흐르는 모습을 웹 UI로 공개

Step 3. 결과 분석: 수집된 데이터를 통해 콜드 스타트 지점을 식별하는 과정 설명

JSON Format

```
"payload_size": 52428800,          # 처리한 데이터 크기
"cold_init_ms": 0,                 # 콜드 스타트 시간
"setup_ms": 0.164,                 # 복사 작업 전에 필요한 준비 시간
"pure_memory_copy_ms": 51.588,     # 메모리 복사에 걸린 시간
"pure_memory_mapping_ms": 0,       # 메모리 매핑에 걸린 시간
"buffer_teardown_ms": 0.002,       # 작업 후 버퍼 정리, 해제, 마무리에 걸린 시간
"process_ms": 0,                   # 복사 외 로직 처리 시간
"total_ms": 51.753                 # 총합 실행 시간
```

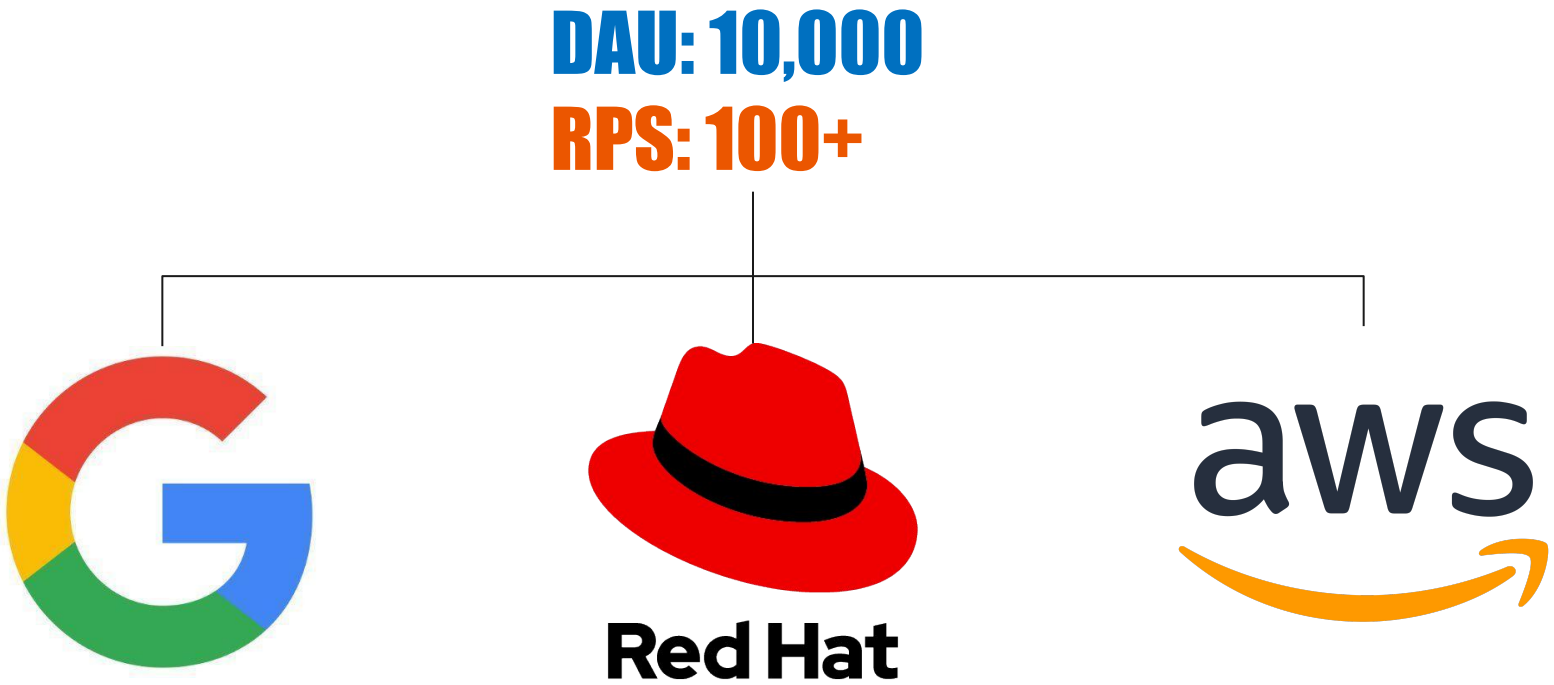
확장성 및 개선 방향

- 지금은 `sys_execve()` (인스턴스 트리거하는 역할) time stamp만을 기준으로 성능 모니터링을 진행하고 있다.
  → `open()`, `read()` 등 다양한 system call로 확장 개선 예정입니다.
- Business 수준에서 DB연결, WAS, 웹서버 등을 통해 여러 의존성이 추가된다면, 더 많은 오버헤드가 발생하는 상황
  → 이러한 상황에서 이 서비스를 도입하면 더 큰 성능 향상을 기대할 수 있을 것으로 예상합니다.

Thank You!

Qn
A

Target Reference



More about Opportunities…

서버리스 수요

*"Fact 1: 주요 클라우드 제공업체들은 **서버리스 도입이 지속적으로 크게 증가**하는 것을 목격하고 있습니다."*
지난 한 해 동안 Azure와 Google Cloud에서 운영되는 조직의 서버리스 도입률은 각각 6%와 7% 증가했으며, AWS는 3%의 성장률을 보였습니다.

Datadog, state-of-serverless report (https://www.datadoghq.com/state-of-serverless/), 2023.08

시스템 레벨 관측 니즈

클라우드 네이티브 소프트웨어를 위한 오픈 소스 관찰 가능성 프레임워크 오픈소스 **OpenTelemetry**로 증명되는 수요

- 이용 벤더: Alibaba, Delivery Hero, Dropbox DocSend, eBay, GitHub, Skyscanner 등 35개 벤더

  OpenTelemetry Homepage(https://opentelemetry.io/)
- OpenTelemetry 프로젝트 산하 GitHub 저장소들의 누적 스타 수 **: 66,974개**

  OpenTelemetry GitHub (https://github.com/open-telemetry)