Walle: An End-to-End, General-Purpose, and Large-Scale Production System for Device-Cloud Collaborative Machine Learning

Chengfei Lv, Zhejiang University and Alibaba Group; Chaoyue Niu, Shanghai Jiao Tong University and Alibaba Group; Renjie Gu, Xiaotang Jiang, Zhaode Wang, Bin Liu, Ziqi Wu, Qiulin Yao, Congyu Huang, Panos Huang, Tao Huang, Hui Shu, Jinde Song, Bin Zou, Peng Lan, and Guohuan Xu, Alibaba Group; Fei Wu, Zhejiang University; Shaojie Tang, University of Texas at Dallas; Fan Wu and Guihai Chen, Shanghai Jiao Tong University

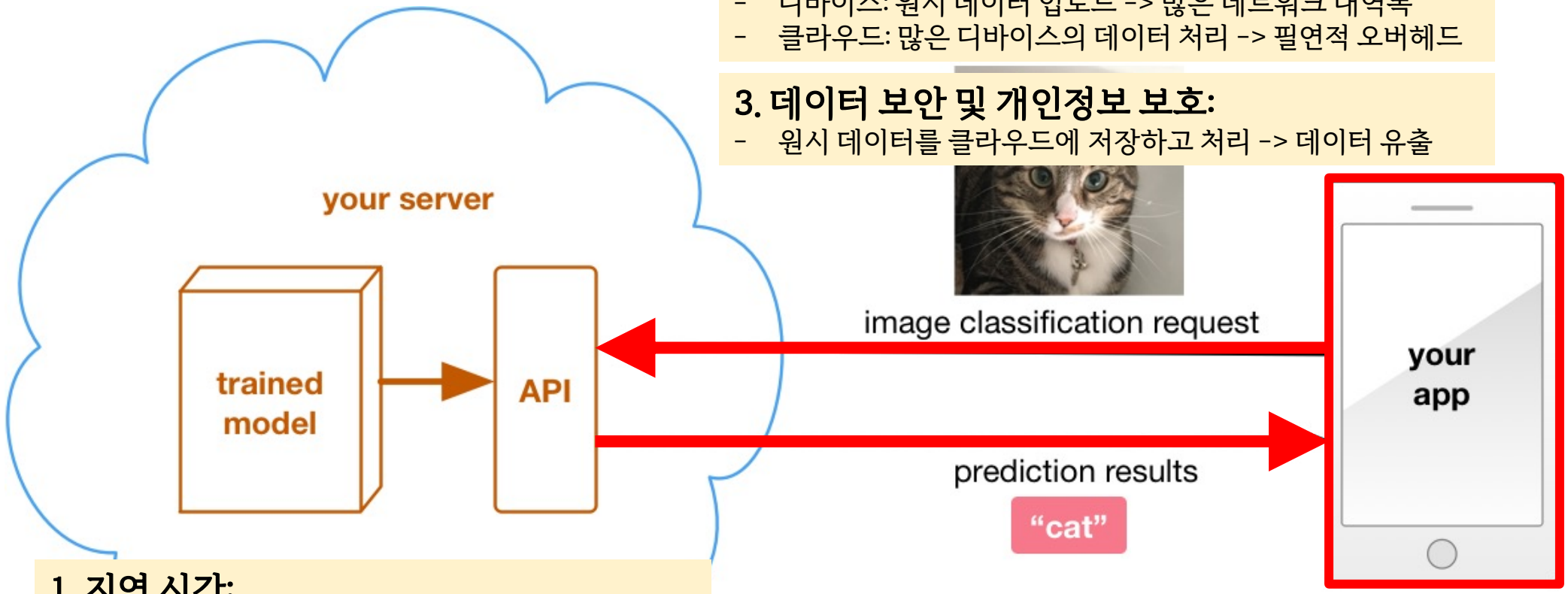
USENIX OSDI'22

EWHA WOMANS UNIVERSITY
Distributed Computing & Operating Systems Lab

Background – Bottlenecks of ML Framework in Cloud

2. 높은 비용 및 과부하:
- 디바이스: 원시 데이터 업로드 -> 많은 네트워크 대역폭
- 클라우드: 많은 디바이스의 데이터 처리 -> 필연적 오버헤드

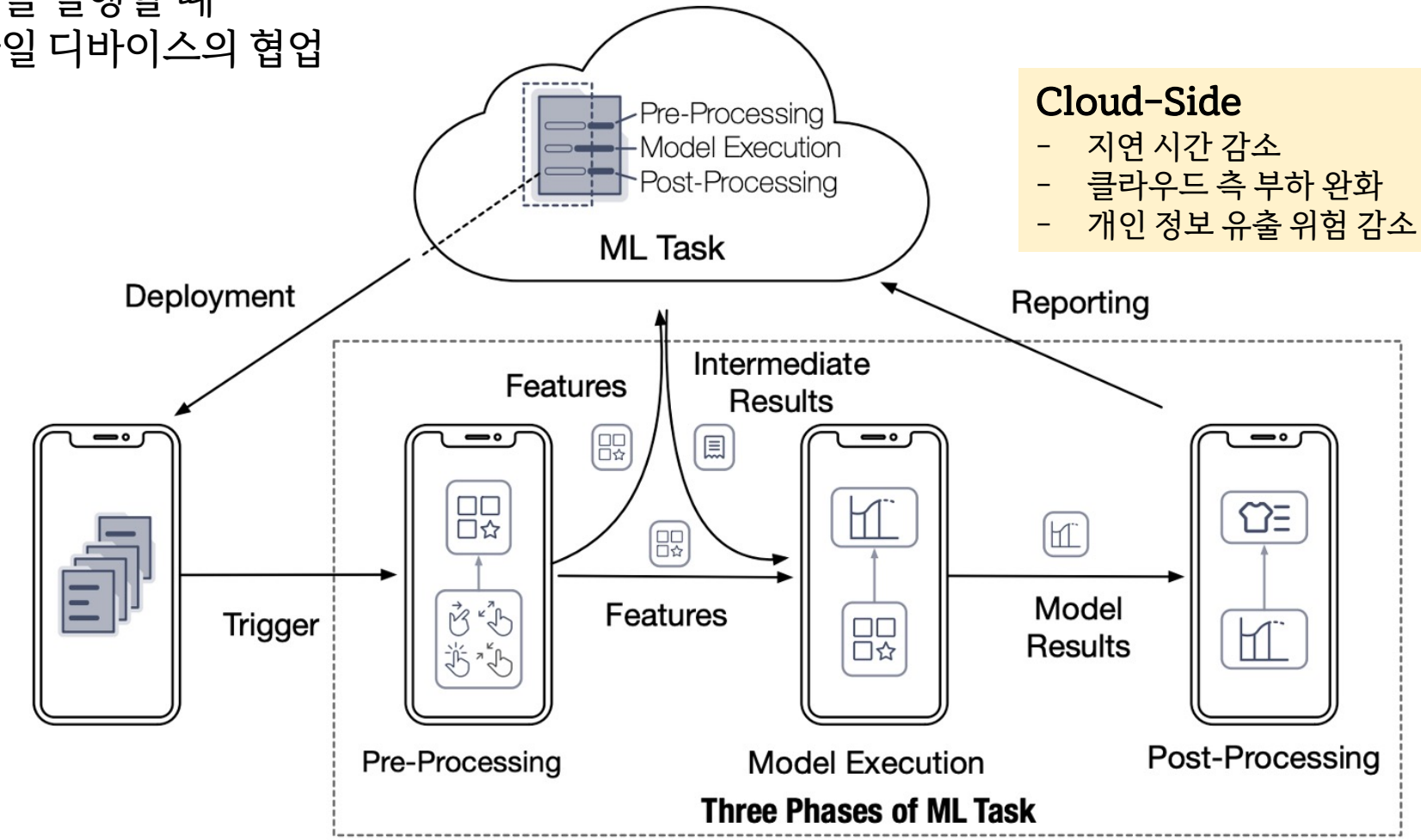
3. 데이터 보안 및 개인정보 보호:
- 원시 데이터를 클라우드에 저장하고 처리 -> 데이터 유출



1. 지연 시간:
- 모바일 디바이스 – 클라우드 간 네트워크 지연 시간
- 클라우드에서 요청 처리 지연 시간

Background – Device-Cloud Collaborative ML Framework

: 복잡한 ML 작업을 실행할 때
클라우드와 모바일 디바이스의 협업



Cloud-Side
- 지연 시간 감소
- 클라우드 측 부하 완화
- 개인 정보 유출 위험 감소

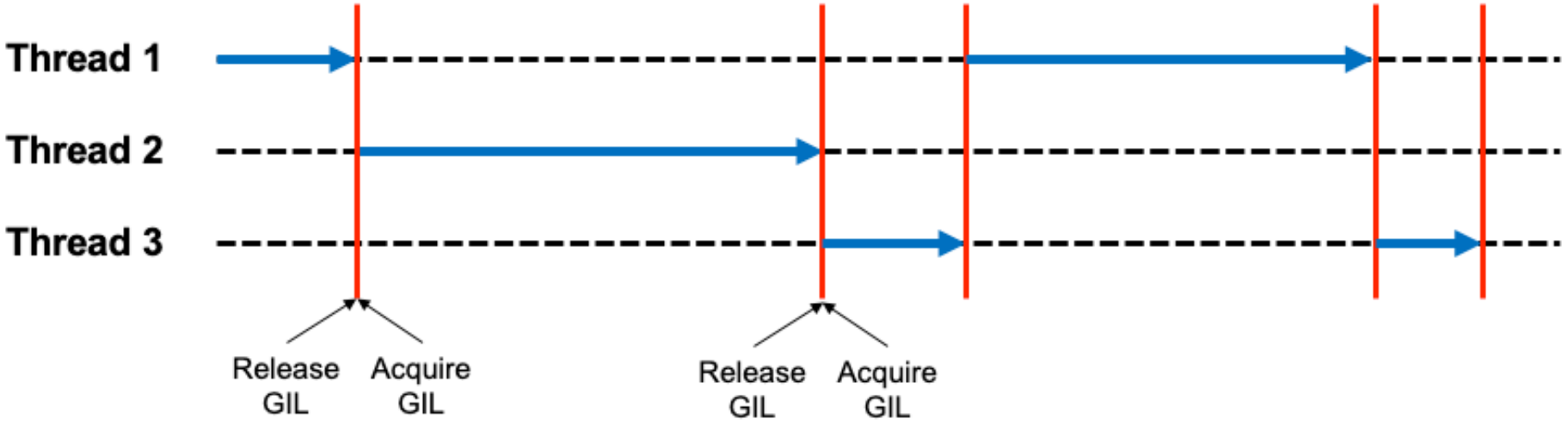
Device-Side
- 통신 비용 절감
- 지연 시간 감소

Background – Multithreading in Python

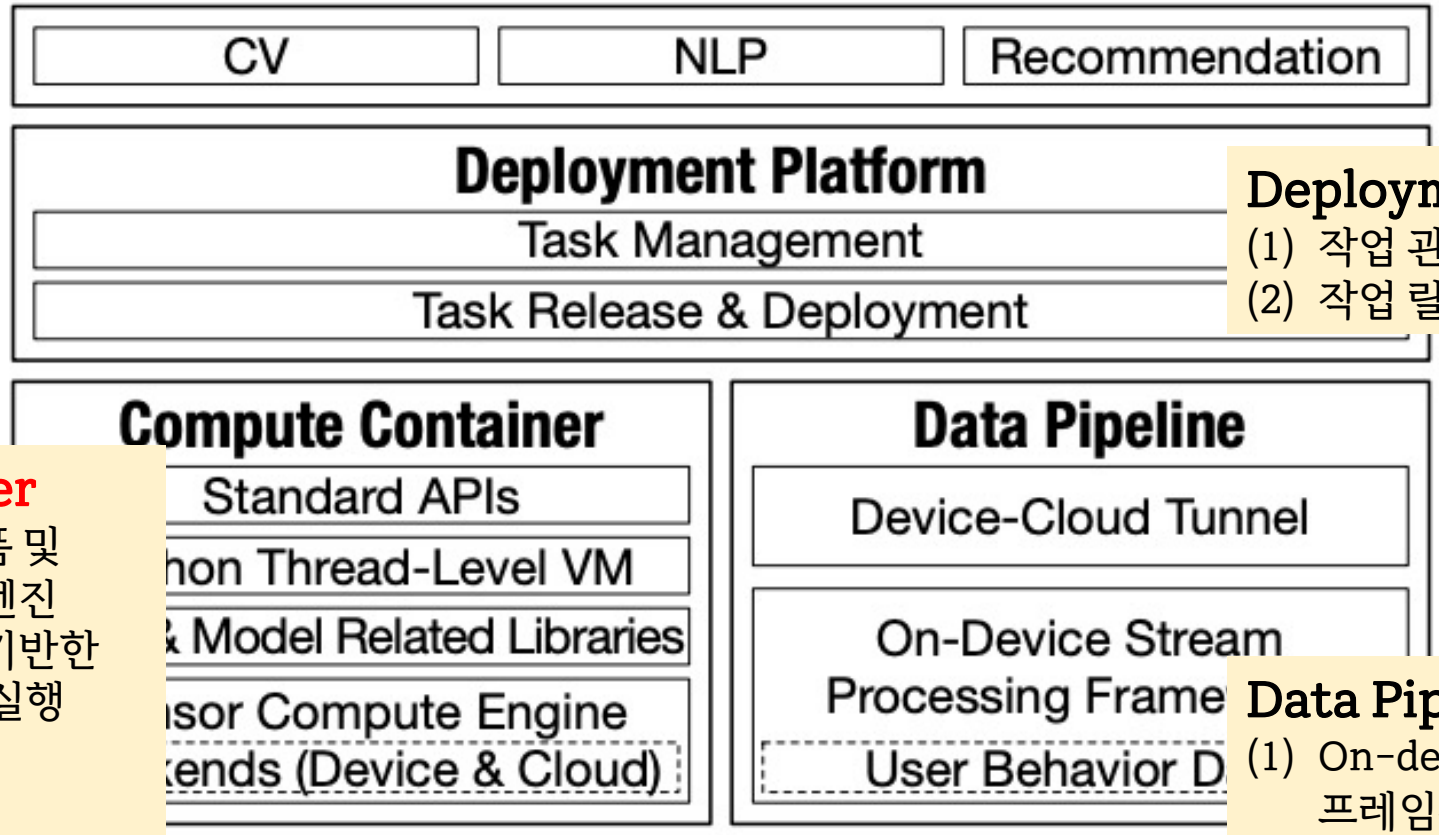
Python GIL(Global Interpreter Lock):

- 모바일 앱에는 한 번에 하나의 프로세스가 가동
- ***모바일 앱의 성능을 향상시키고 싶다면 멀티 스레드 사용하는 것이 일반적***
- GIL은 프로세스 내에서 한 번에 하나의 스레드만 처리할 수 있도록 한다.

⇒ 처리 능력이 제한적인 모바일 기기에서 효율성을 향상시키려는 경우 CPython에서 멀티 스레딩을 지원하지 않는 것이 문제가 됨



Architecture of Walle



Deployment Platform
(1) 작업 관리 모듈
(2) 작업 릴리스 및 배포 모듈

Compute Container
(1) 하단의 크로스 플랫폼 및 고성능 텐서 컴퓨팅 엔진
(2) 텐서 컴퓨팅 엔진에 기반한 데이터 처리 및 모델 실행 라이브러리
(3) Python 스레드 수준 가상머신
(4) 상단의 표준 API

Data Pipeline
(1) On-device 스트림 처리 프레임워크
(2) 실시간 device-cloud 터널 도입

1. Compute Container

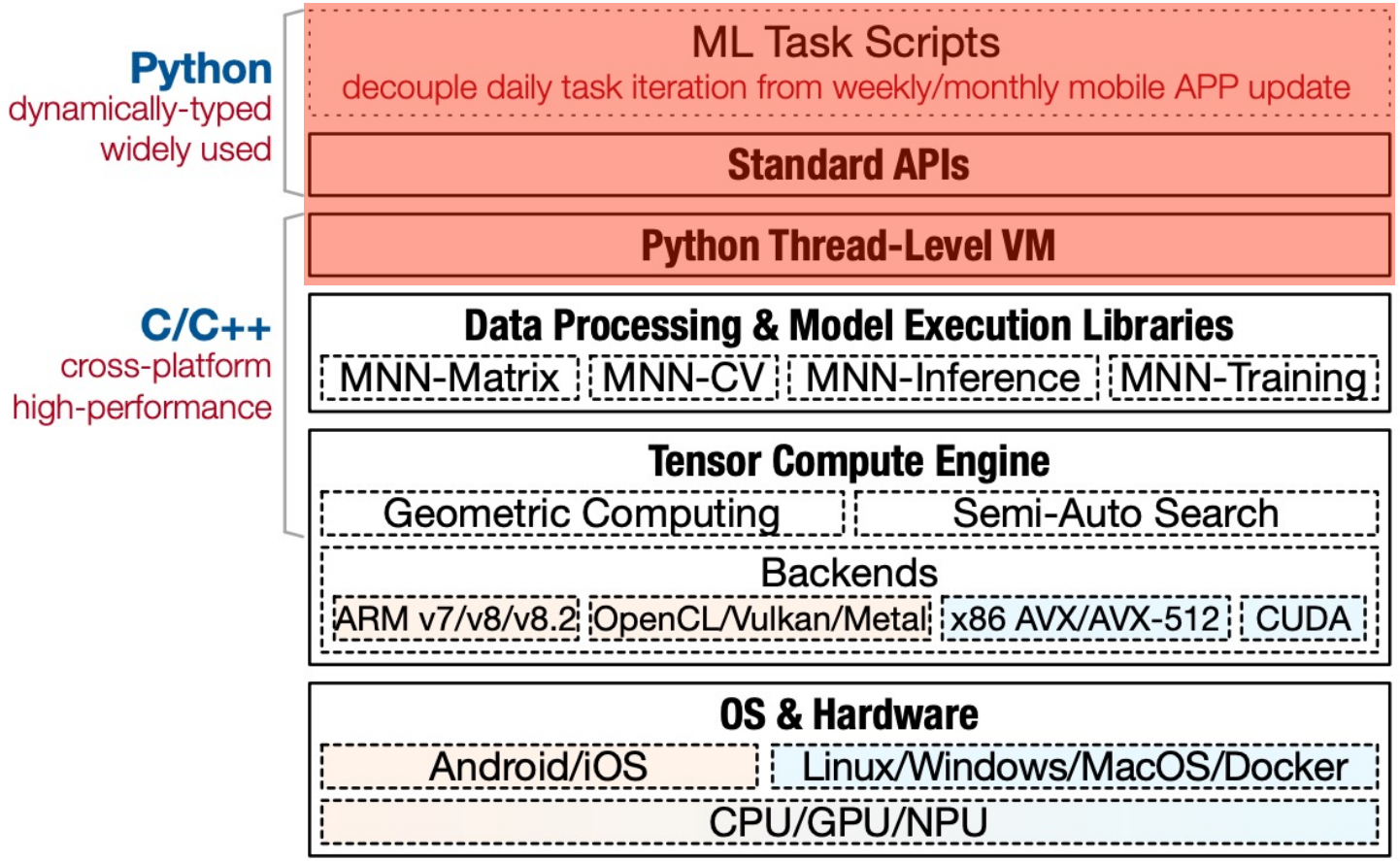


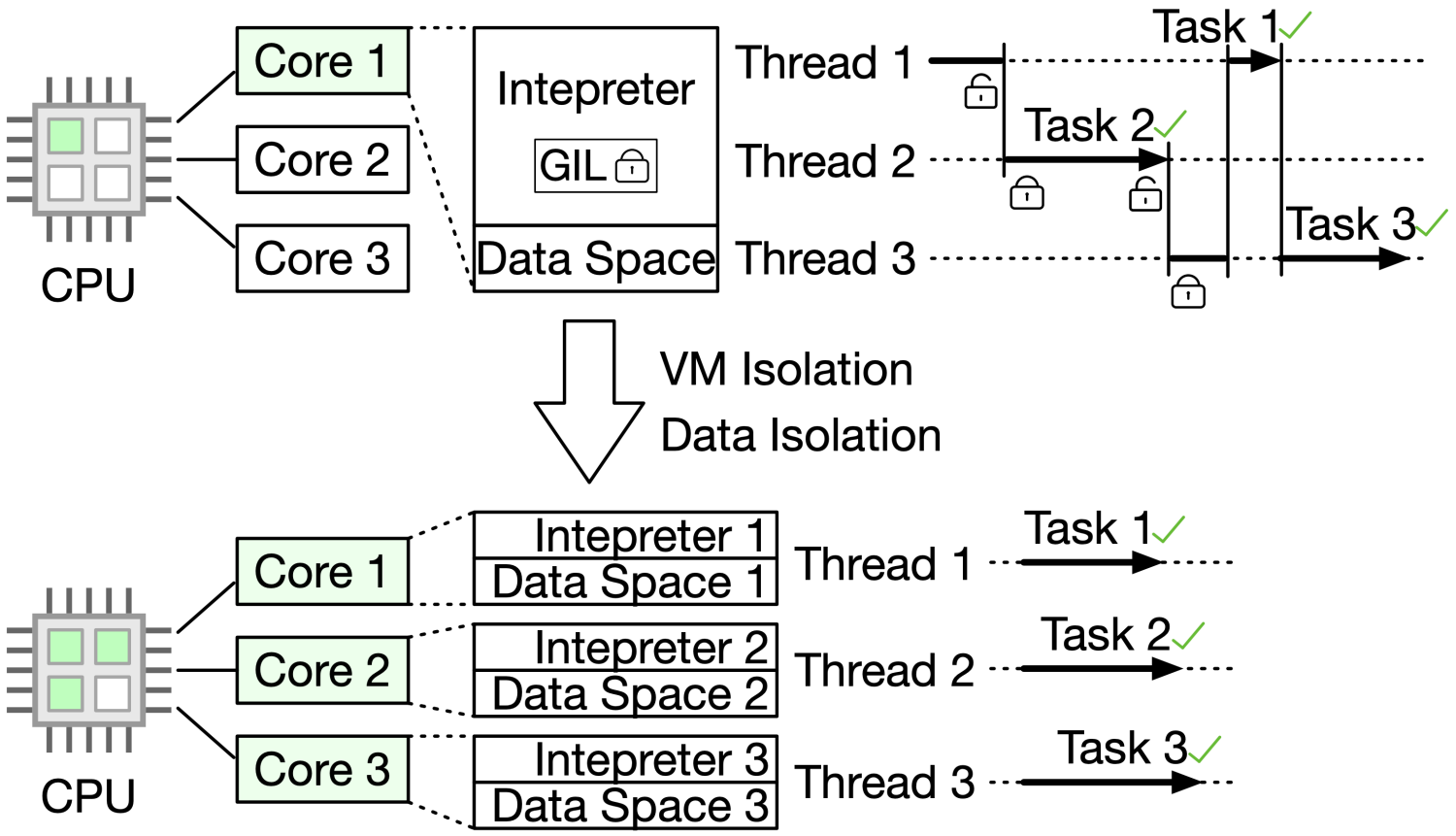
Figure 3: Architecture of compute container.

리소스 제약이 있는 모바일 디바이스의 경우
CPython의 포팅 과정에서 두 가지 주요 문제:

1. 패키지의 크기가 크다
- 컴파일 단계는 클라우드에 남겨두고 실행을 위해 바이트코드만 모바일 기기로 전송하여 컴파일 모듈을 모두 삭제
- 모바일 기기용 경량 Python 인터프리터 구현
- 패키지 크기 10MB 이상 -> iOS 1.3MB로 감소

2. 멀티 스레딩을 지원하지 못한다
- 멀티 스레딩과 관련해서는 GIL을 포기하고 업계 최초로 파이썬 스레드 레벨 인터프리터를 구현
- 스레드 보호를 위해 스레드 수준의 VM 격리 및 데이터 격리를 수행

1. Compute Container: thread-level VM



각 작업은 특정 스레드에 스케줄링 되어 독립적인 VM 생성

(1) **VM 격리**: 하나의 프로세스가 여러 개의 스레드 수준 VM을 보유할 수 있고 각 VM이 독립적인 수명 주기를 갖도록 VM 인스턴스 생성

✓ CPython에서 VM은 PyInterpreterState 구조체로 정의

✓ 각 스레드에 대해 PyInterpreterState 인스턴스를 생성하고 초기화하도록 수정

(2) **데이터 격리**: VM 런타임의 컨텍스트도 스레드 수준에서 격리

✓ 스레드마다 고유한 데이터 공간 확보

✓ 타입 시스템, 버퍼 풀, 오브젝트 할당, 가비지 컬렉션에 적용

1. Compute Container: Tensor Compute

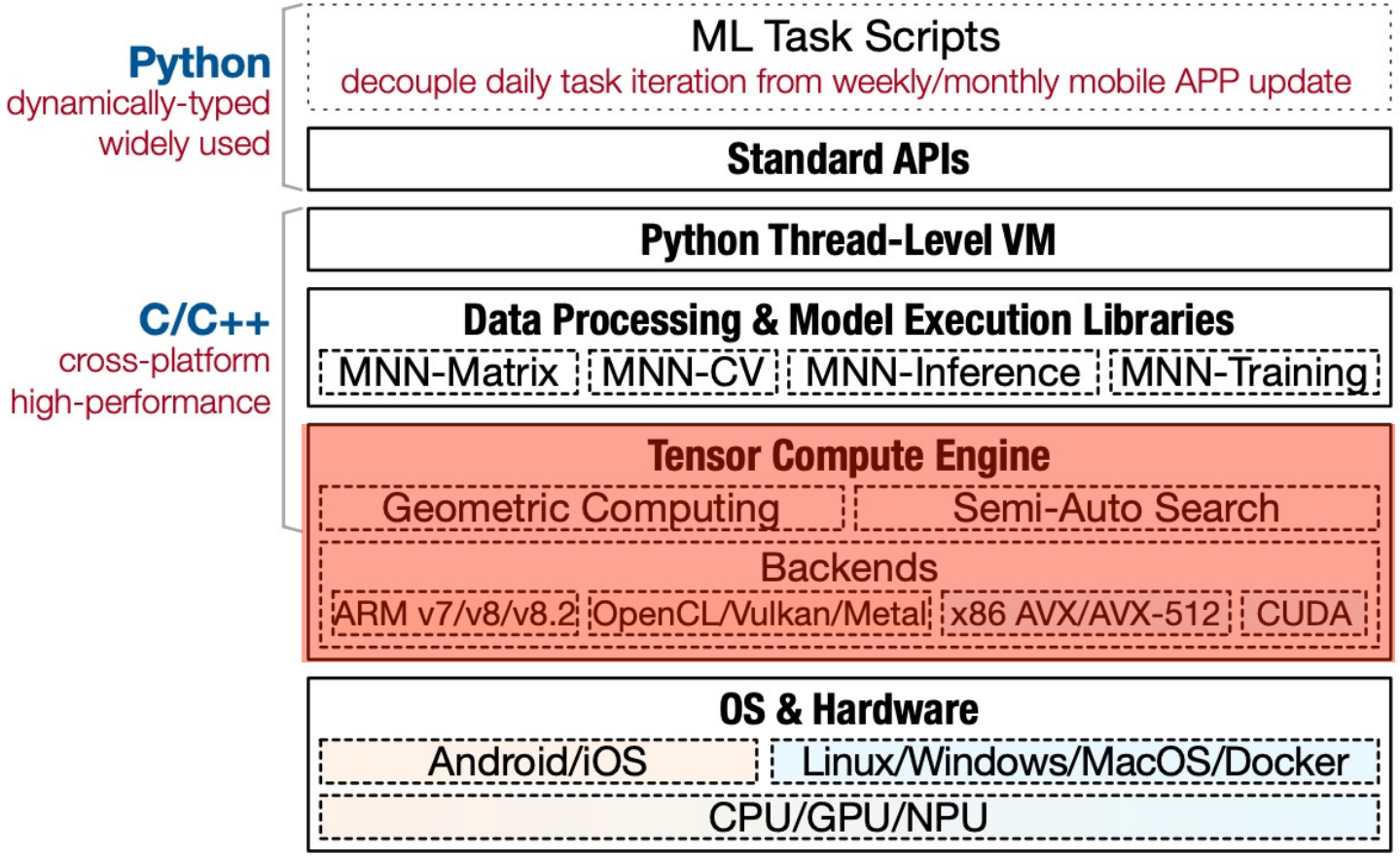


Figure 3: Architecture of compute container.

: 데이터 처리와 머신러닝의 기초

(1) 원자 연산자: 단항 연산자(제곱), 이진 연산자 (사칙연산)
(2) **변환 연산자**: 전치, 조각, 연결, 순열 등 요소의 모양을 변경하거나 순서를 바꾸는 연산자
(3) **복합 연산자**: 3D 컨볼루션 및 풀링, 정규화, 지수 선형 단위, 장단기 메모리 셀 등 원자 연산자와 변환 연산자로 분해 가능
(4) 제어 흐름 연산자: if 및 while

MNN의 모든 백엔드에 대한 연산자를 구현/최적화 작업량 =

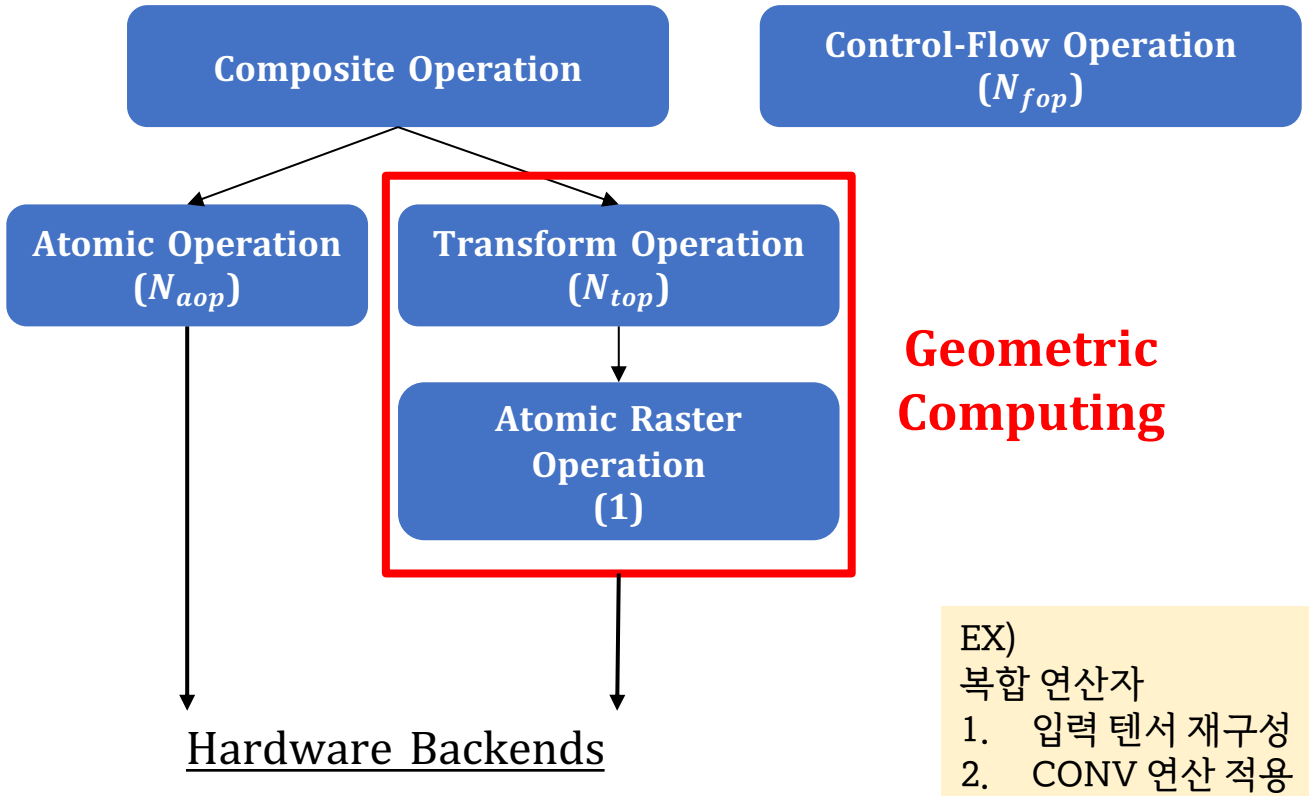
$$O\left(\underbrace{(N_{aop}+N_{top}+N_{cop})}_{\text{사용 가능한 모든 유형의 연산자 구현의 총 수}}\times N_{ba}+N_{fop}\right)$$

모든 백엔드에서 이러한 연산을 구현하는데 필요한 작업의 양

- N_{aop}: 원자 연산자의 수
- N_{top} : 변환 연산자의 수
- N_{cop} : 복합 연산자의 수
- N_{fop} : 제어 흐름 연산자의 수
- N_{ba} : 백엔드 개수

⇒ **연산에 필요한 작업의 양을 줄여야 한다**

1. Compute Container: Tensor Compute Engine



Geometric Computing

1. Geometric Computing:
좌표 이동을 통해 입력 텐서 – 출력 텐서 간 요소를 이동하는 데 사용되는 매커니즘

✓ Raster 연산자 정의: 변환 연산자와 복합 연산자를 모두 래스터 연산자와 원자 연산자로 분해

✓ 계산 작업량을 46% 감소

✓ $O\left(\left(N_{aop}+N_{top}+N_{cop}\right)\times N_{ba}+N_{fop}\right)$

$\Rightarrow O\left(\left(N_{aop}+1\right)\times N_{ba}+N_{top}+N_{cop}\right)$

매번 새로운 코드를 처음부터 작성하는 대신
개별 원자 연산자에 대해 최적화된
기존 구현을 여러 백엔드에서 재사용함
⇒ 변환 및 복합 연산자 구현에 수반되는 워크로드를
O((Ntop + Ncop) × Nba) 에서
O((Naop + 1) × Nba)로 줄일 수 있다

EX)
복합 연산자
1. 입력 텐서 재구성
2. CONV 연산 적용
3. 다른 텐서를 사용하여 요소 별 덧셈 수행

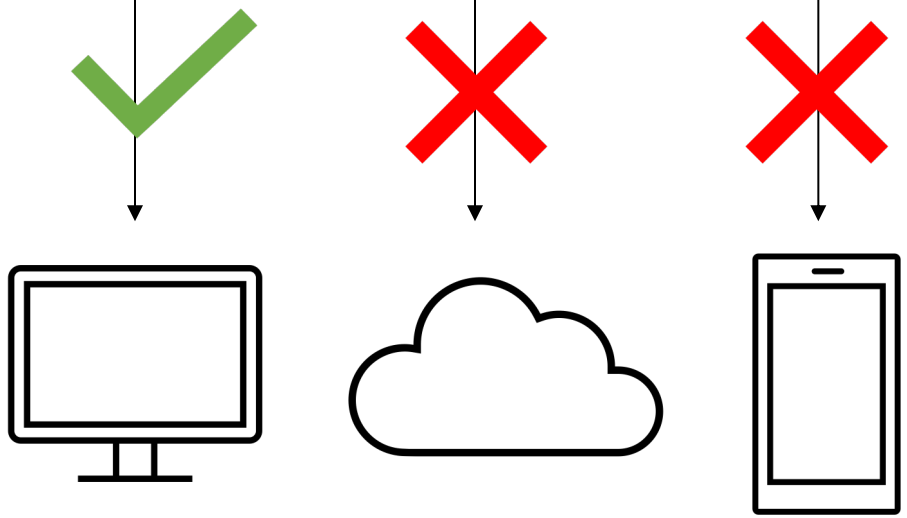
사용 가능한 모든 백엔드에 대해
복합 연산자를 구현하기 위해서는
세 가지 연산 각각에 대해
별도의 코드를 작성하고,
개별적으로 최적화 필요

복합 연산자를 래스터 연산자 사용하여 원자연산자로 분해
1. 래스터를 사용하여 메모리 주소에 따라 텐서 간 요소 이동 (=reshaping)
2. CONV 대신 원자 연산자 사용 (행렬 곱셈 및 요소 별 활성화 함수)
3. 요소별 덧셈 대신 원자 덧셈 연산과 함께 래스터 다시 사용

1. Compute Container: Tensor Compute Engine

A series of operators

$$op1 \rightarrow op2 \rightarrow \cdots \rightarrow opn$$



Available Backends

2. Semi-Auto Search:

모바일 디바이스/클라우드 서버에서 사용 가능한 백엔드를 빠르게 파악

✓ 백엔드마다 연산자에 대한 구현과 최적화가 다름

✓ 자동화된 검색 & 수동 튜닝을 통해 주어진 모델 아키텍처에 맞는 최적의 하이퍼파라미터 찾기 = 제한된 리소스에서 최적의 비용(제약 최적화 문제)

백엔드 ba의 비용 $C_{ba} = \sum_{i=1}^{n} C_{op_i,ba},$

백엔드 ba의 최소 비용 $\arg\min_{ba \in BA} C_{ba}.$

최적 파라미터와 입력의 크기가 주어졌을 때, 얻을 수 있는 알고리즘 alg의 기본 계산 횟수

$$C_{op_i,ba} = \min_{alg \in algs(op_i,ba)} \frac{Q_{alg}}{P_{ba}} + S_{alg,ba},$$

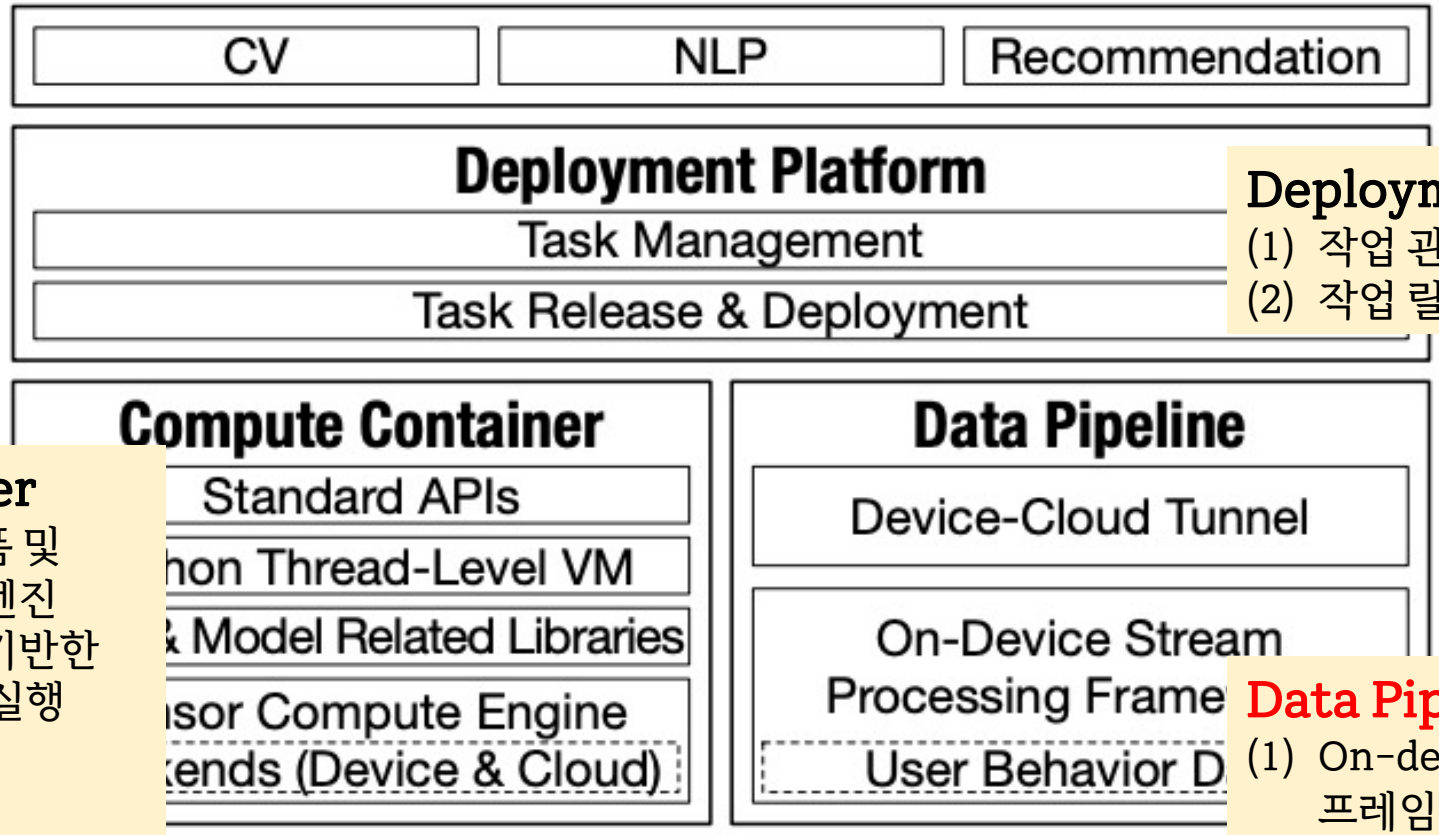
P_{ba}: 백엔드 ba의 성능

$S_{alg,ba}$: 백엔드 ba에서 알고리즘 alg의 스케줄링 비용

타일 크기를 최적화하여 메모리 읽기/쓰기 연산 최소화

$$\min_{t_e,t_b} \frac{e}{t_e} \times \frac{b}{t_b} \times (a \times t_e + a \times t_b + t_e \times t_b),$$
$$\text{s.t.} \quad t_e \times t_b + t_e + t_b \leq N_r, \tag{4}$$

Architecture of Walle



Deployment Platform
(1) 작업 관리 모듈
(2) 작업 릴리스 및 배포 모듈

Compute Container
(1) 하단의 크로스 플랫폼 및 고성능 텐서 컴퓨팅 엔진
(2) 텐서 컴퓨팅 엔진에 기반한 데이터 처리 및 모델 실행 라이브러리
(3) Python 스레드 수준 가상머신
(4) 상단의 표준 API

Data Pipeline
(1) On-device 스트림 처리 프레임워크
(2) 실시간 device-cloud 터널 도입

2. Data pipeline: Stream Processing

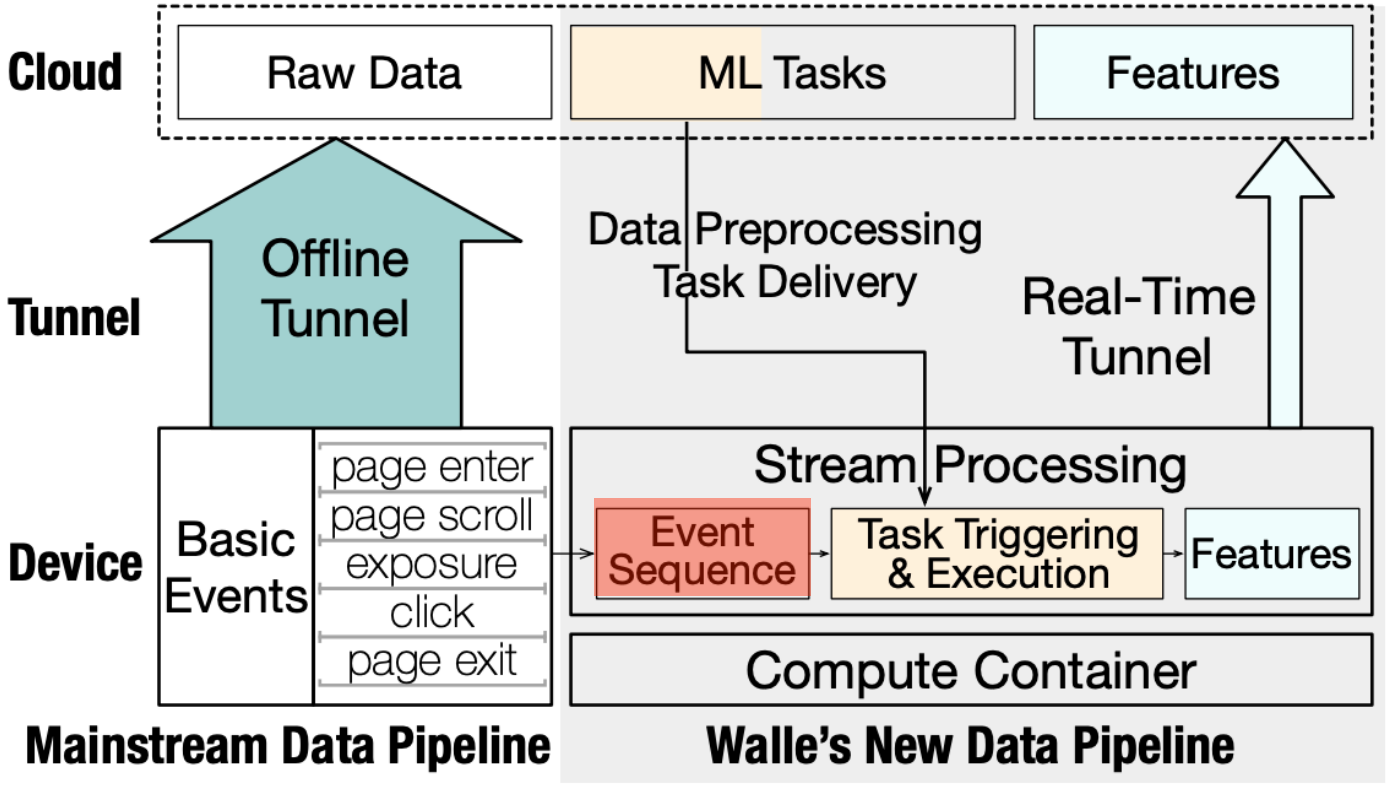


Figure 4: Architecture of data pipeline.

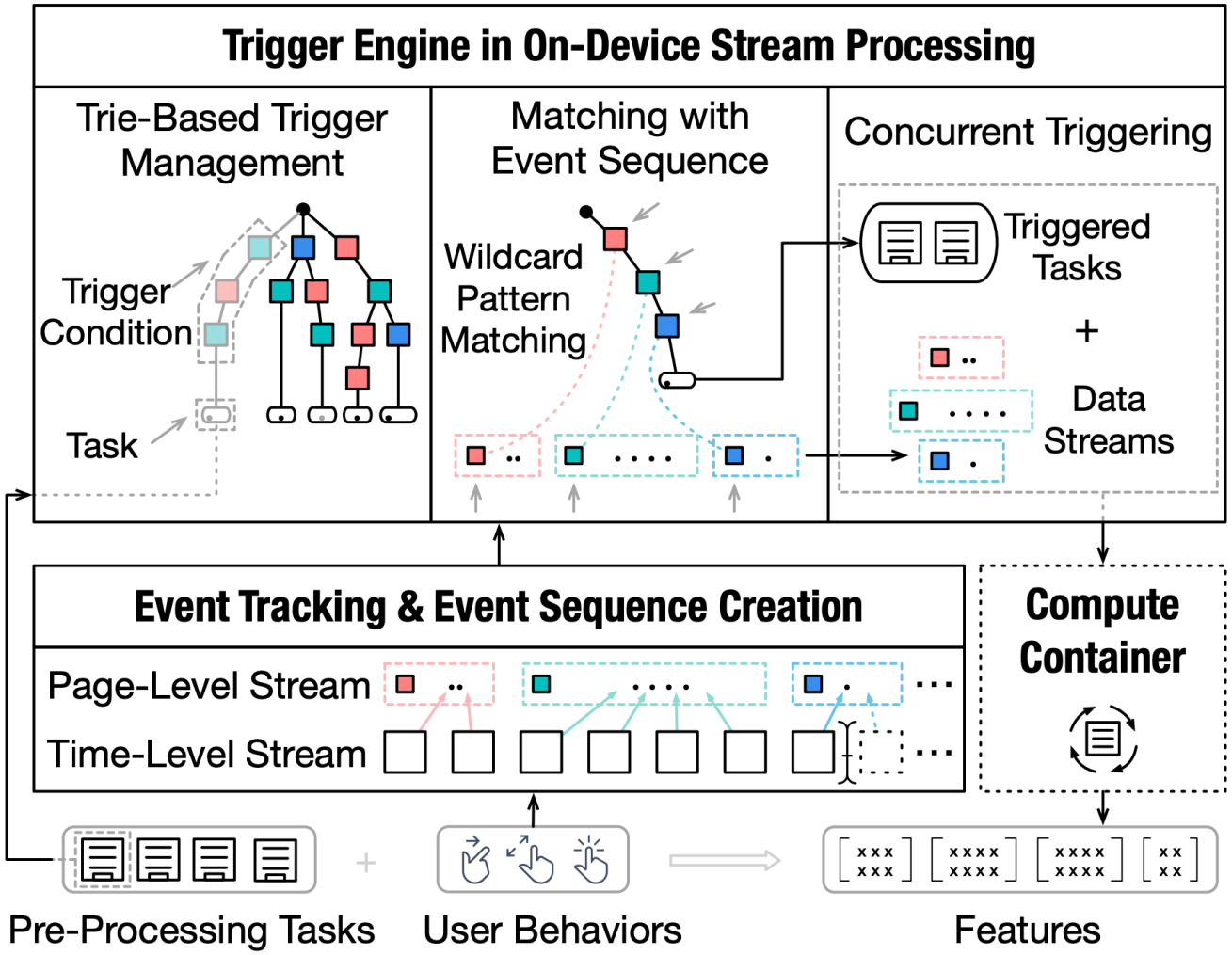
단일 장치에서 무제한 데이터 스트림에 대한 상태 저장 연산을 지원하도록 설계:
⇒ 디바이스의 리소스는 제한되어 있으므로 스트림 처리 작업의 트리거 조건을 잘 관리해야

1. 이벤트 추적 및 시퀀스 생성

- 사용자가 모바일 앱과 상호 작용할 때, 시간 단위의 이벤트 시퀀스로 기록
- 시간 단위의 이벤트 시퀀스를 기반으로 같은 페이지의 이벤트를 취합하여 **페이지 단위의 이벤트 시퀀스 스트림** 생성

✓ 기본 이벤트: 페이지 진입, 페이지 스크롤, 노출, 클릭, 페이지 종료

2. Data pipeline: Stream Processing



2. 트리거 관리/작업 트리거링

- 새로운 이벤트가 발생하면 트리거된 작업 집합이 반환되고, 컴퓨팅 컨테이너에서 스크립트가 실행되어 관련 이벤트를 처리
- 효율적인 트리거 매칭을 위해 **Trie** 사용
- 동시 트리거 지원

✓ 정적 보류 리스트: 이벤트가 발생하면 다른 노드를 확인하기 전에 이 목록에 대해 먼저 확인

✓ 동적 보류 리스트: 각 이벤트에 대해 진행 중인 매칭의 원하는 다음 노드를 버퍼에 저장

2. Data pipeline:

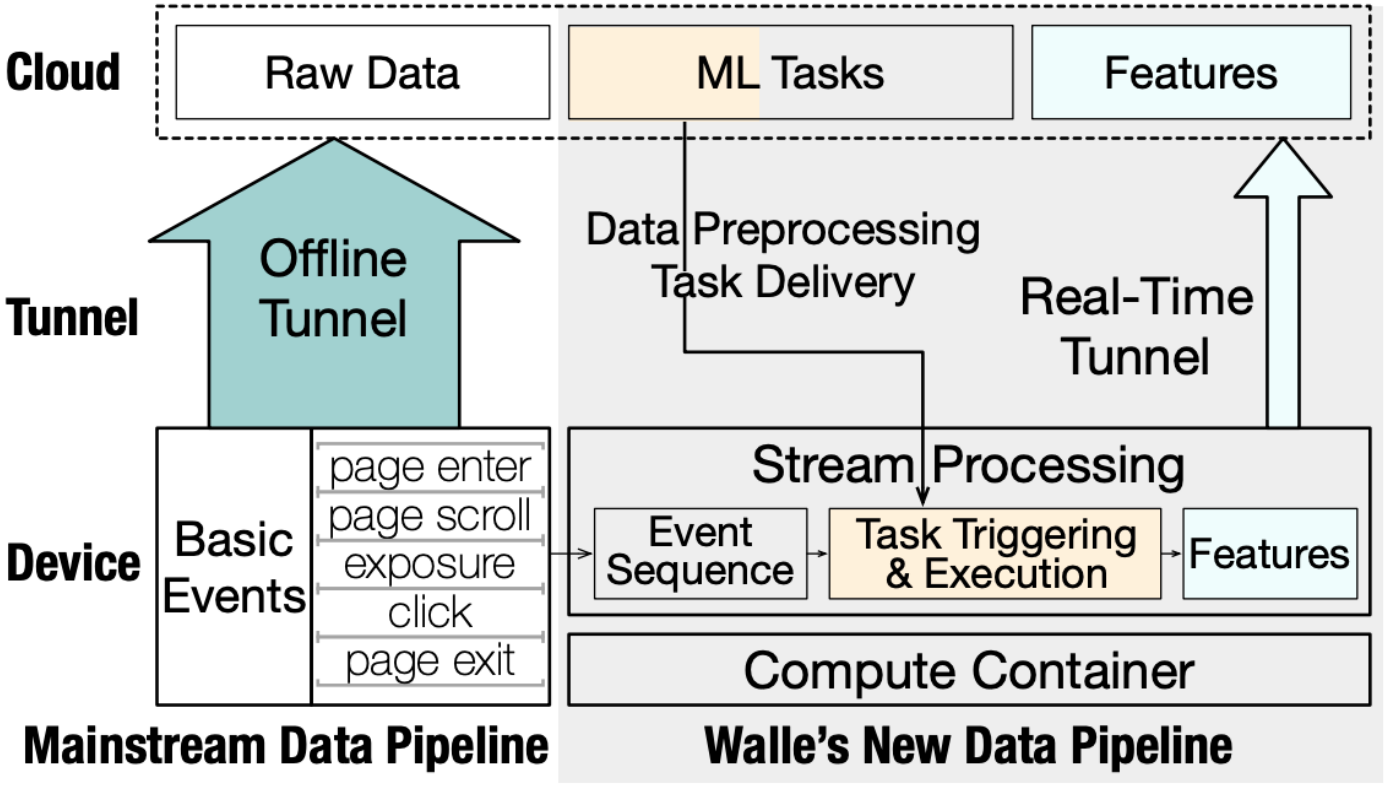


Figure 4: Architecture of data pipeline.

3. 작업 실행 및 집단 저장소

- 태스크 트리거링을 통해 적절한 스크립트가 식별되면 컴퓨팅 컨테이너에서 실행
- 각 스트림 처리 작업의 기능은 SQLite를 사용하여 테이블로 저장
- 배치 데이터 저장 = 쓰기 횟수를 줄여 성능 향상

✓ 스트림 처리 태스크는 여러 번 트리거될 수 있는 반면, 1회 출력의 크기는 작다

✓ 메모리 버퍼링 테이블

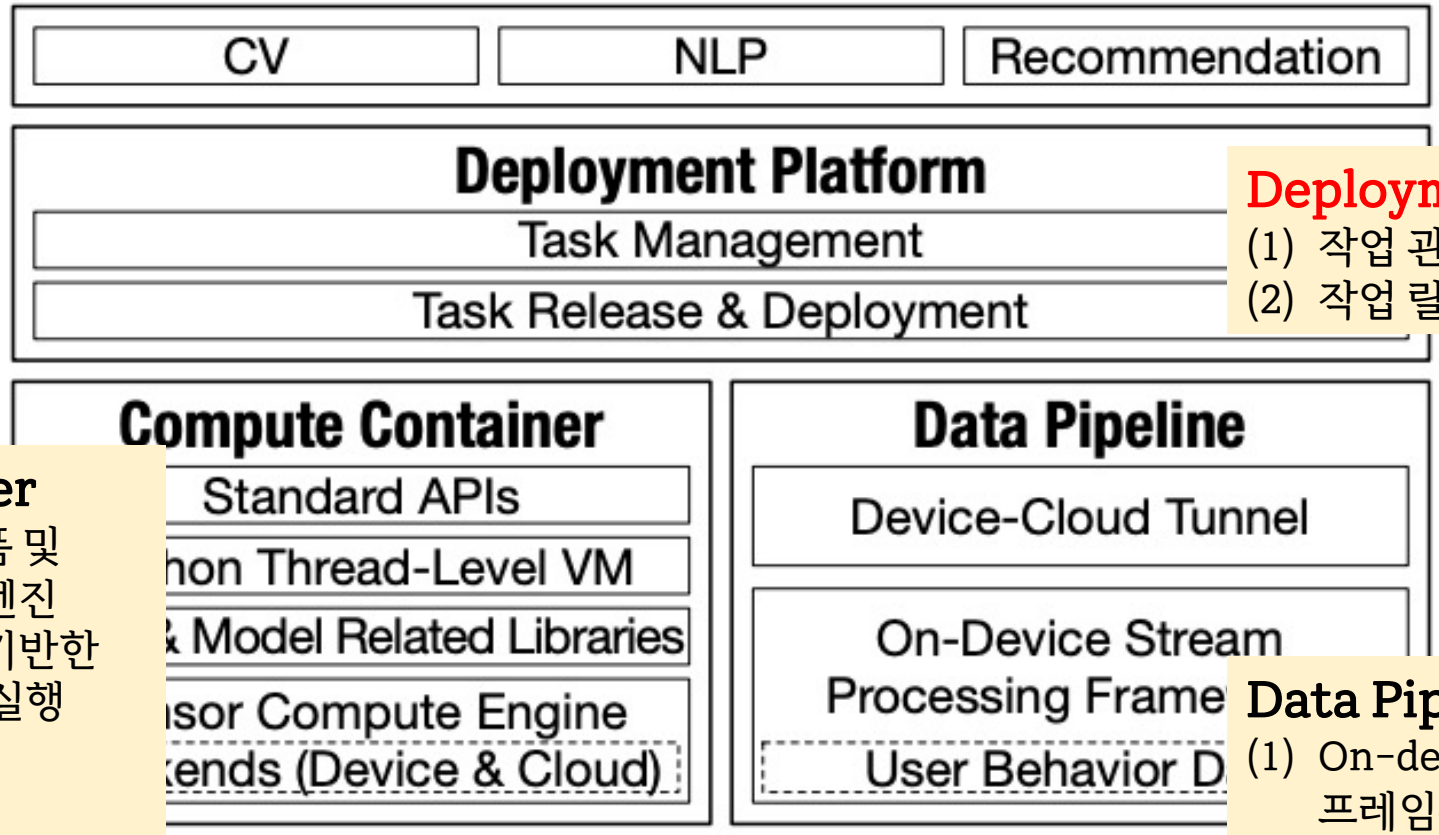
4. 실시간 터널

- SSL 프로토콜을 사용하여 암호화 및 암호 해독을 최적화하는 영구 연결을 기반으로 하는 장치-클라우드 터널

✓ SSL 프로토콜 최적화 하여 연결 설정/암호화/복호화 시간을 단축

✓ 최대 30KB의 데이터를 500ms 이내 전송

Architecture of Walle

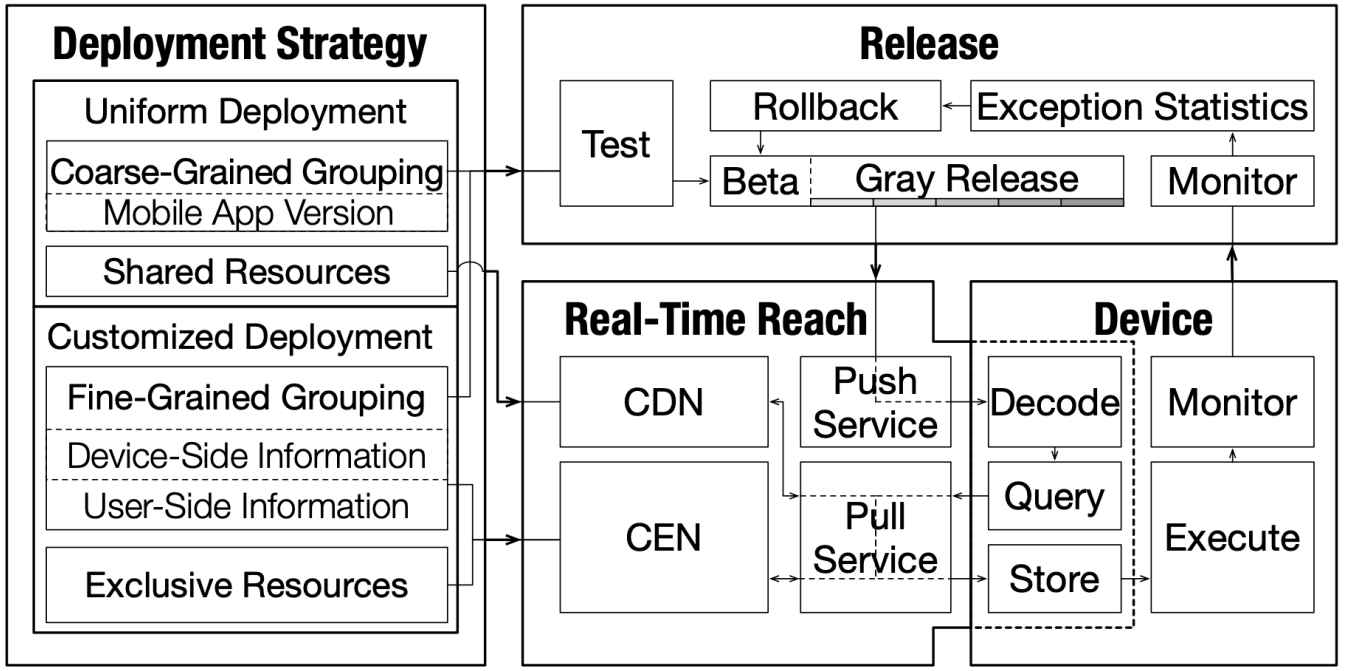


Deployment Platform
(1) 작업 관리 모듈
(2) 작업 릴리스 및 배포 모듈

Compute Container
(1) 하단의 크로스 플랫폼 및 고성능 텐서 컴퓨팅 엔진
(2) 텐서 컴퓨팅 엔진에 기반한 데이터 처리 및 모델 실행 라이브러리
(3) Python 스레드 수준 가상머신
(4) 상단의 표준 API

Data Pipeline
(1) On-device 스트림 처리 프레임워크
(2) 실시간 device-cloud 터널 도입

3. Deployment Platform



- **Push-then-Pull 정책**: 모바일 디바이스는 해당 CDN 또는 CEN 주소를 사용하여 가장 가까운 노드에서 해당 파일을 가져온다. = 먼저 클라우드에서 클라이언트로 정보를 푸시한 다음 수신된 정보를 기반으로 클라이언트별로 데이터를 가져옴
  - ✓ Push: 클라우드가 비즈니스 요청에 대한 응답으로 모바일 디바이스의 로컬 작업 프로필에 정보(예: CDN 주소)를 push
  - ✓ Pull: 모바일 디바이스가 수신된 정보를 기반으로 해당 CDN 또는 CEN 주소에서 파일을 가져오는 것

- Git으로 작업 엔티티를 관리하고, 파일을 공통으로 사용할 수 있는 디바이스 수에 따라 작업 관련 파일을 공유 파일과 전용 파일로 분류

  - ✓ 전체 태스크 관리 = 1 Git group
  - ✓ 각 비즈니스 시나리오 = Git Repository
  - ✓ 비즈니스 시나리오의 각 태스크 = Branch
  - ✓ 태스크의 각 버전 = Tag
  - ⇒ 파일 분류는 태스크 배포의 균일하고 맞춤화된 정책을 더욱 용이하게 한다.

- 공유 파일:
  - → 다수의 모바일 디바이스에서 사용 = 균일정책
- 전용 파일:
  - → 소수의 디바이스 또는 특정 디바이스만 사용 = push-then-pull 정책

Evaluation: Computing Container in Live Streaming

❖ E-Commerce Scenarios: Walle 컴퓨팅 컨테이너의 높은 성능과 디바이스-클라우드 협업의 실질적인 효과 확인

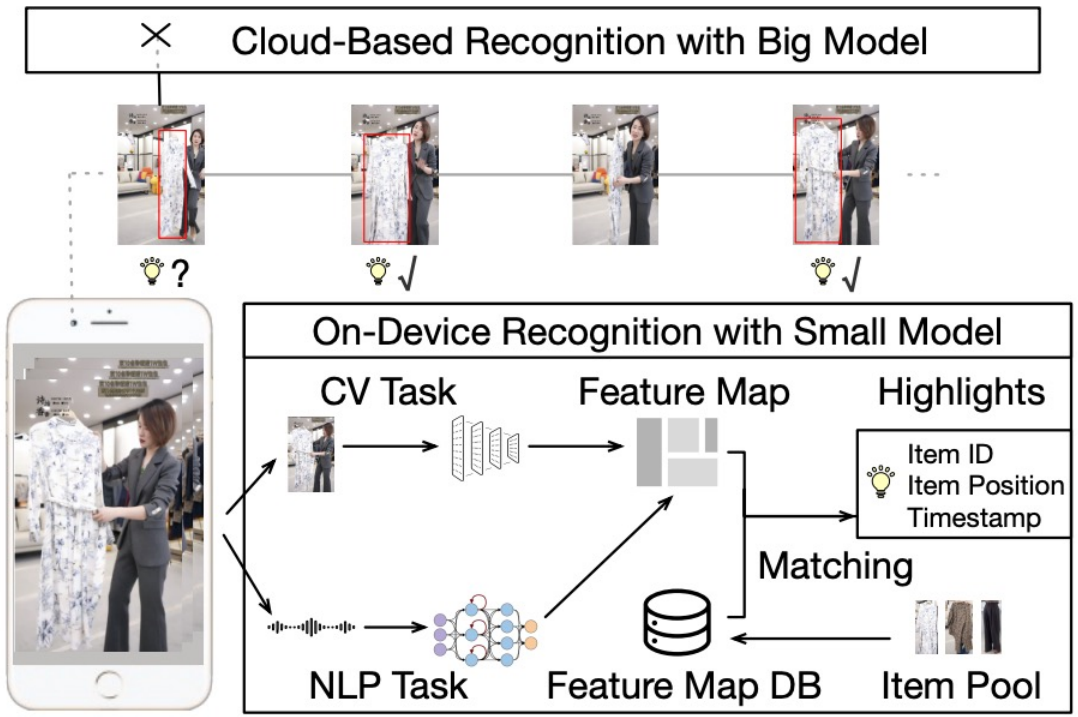


Figure 9: Workflow of device-cloud collaborative highlight recognition in e-commerce livestreaming.

- ML 작업 = 스트리머가 상품에 대한 매력적인 정보를 소개하는 시점을 찾아내는 하이라이트 인식

- 기존) 클라우드의 부하가 너무 커서 비디오 스트림의 일부와 몇 개의 샘플링된 프레임만 분석할 수 있다.

- Walle) 경량 모델을 스트리머의 모바일 디바이스로 오프로드하여 하이라이트 인식 작업을 스트리머의 모바일 장치에서 수행

✓ 디바이스 측 모델에서 높은 신뢰도로 인식된 하이라이트는 후처리 후 사용자에게 전달, 낮은 신뢰도로 인식된 하이라이트는 클라우드 측의 대형 모델에서 처리

✓ 하이라이트 인식이 가능한 스트리머 수를 늘리고(123%증가)

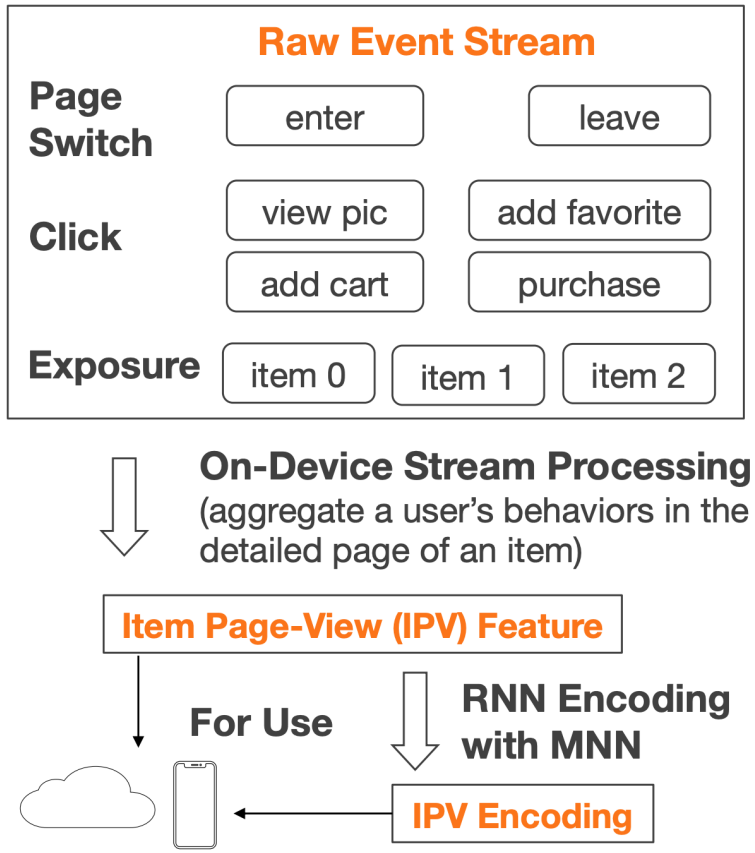
✓ 하이라이트 인식당 클라우드의 컴퓨팅 부하를 감소(87%감소)

✓ 클라우드 비용 단위당 일일 인식되는 하이라이트의 크기를 늘릴 수 있다(74%증가)

✓ 총 지연 시간은 각각 130.97ms(Huawei P50 Pro)와 90.42ms(iPhone 11)

Evaluation: Data Pipeline in Recommendation

❖ Recommendation Scenarios: Walle의 새로운 데이터 파이프라인이 실제로 디바이스-클라우드 통신 비용과 클라우드 측 부하를 줄이면서 기능의 적시성과 유효성을 개선할 수 있음을 확인



- ML 작업 = 사용자 행동을 기록하고 IPV 기능을 생성하는 추천 모델
- 알리바바의 클라우드 기반 및 디바이스 기반 추천 모델에서는 항목의 상세 페이지에서 사용자의 행동(예: 즐겨찾기 추가, 장바구니 추가, 구매)을 기록하는 항목 페이지 뷰(IPV) 기능이 매우 중요하다.

- 기존) 스트림 처리를 위해 모든 사용자 원시 이벤트 데이터를 클라우드에 업로드해야 하므로 시간과 리소스 집약적

- Walle) 온 디바이스 스트림 처리 프레임워크를 사용하면 각 모바일 디바이스에서 해당 사용자의 로컬 이벤트 중 일부만 처리할 수 있어 효율성이 높아진다.

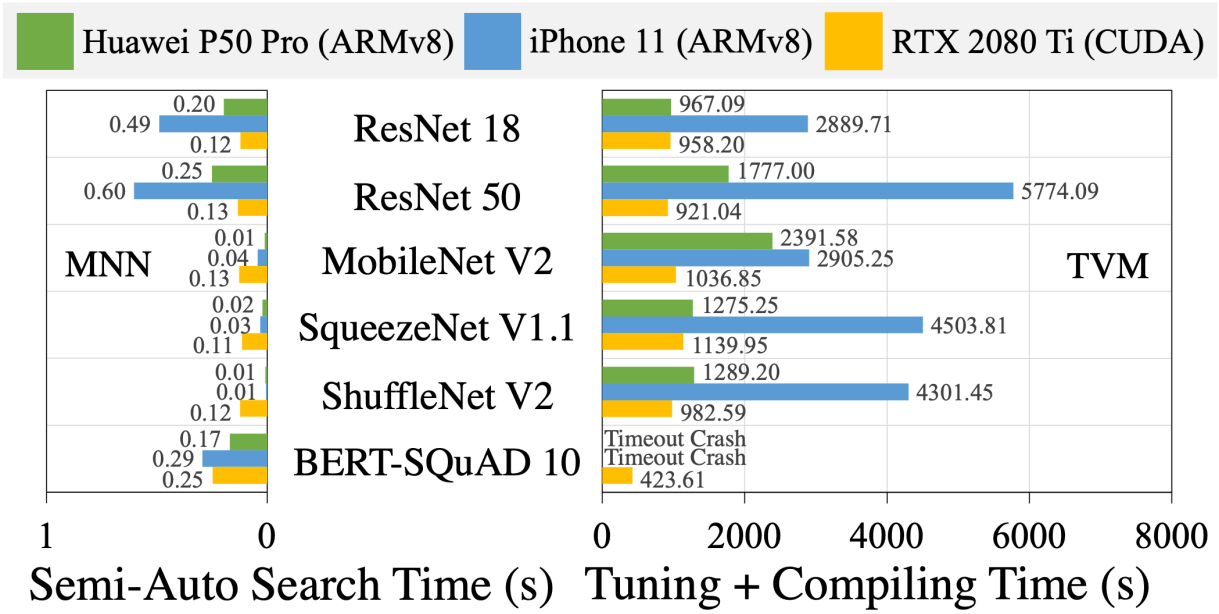
✓ 평균적으로 하나의 IPV 기능은 약 1.3KB 크기이며, 하나의 IPV 인코딩은 128바이트에 불과하다.
= 스트림 처리를 위해 원시 이벤트 데이터를 클라우드로 전송하는 기존 패러다임에 비해 새로운 IPV 데이터 파이프라인을 사용하면 통신 비용을 90% 이상 절감

✓ 평균 온 디바이스 지연 시간은 44.16ms에 불과 (클라우드 기반 처리는 33.73s)

Evaluation: Benchmark

❖ Android 및 iOS 기기와 Linux 서버에서 MNN을 TensorFlow(Lite) 및 PyTorch(모바일)와 비교
- 디바이스 측 테스트: 단일 스레드 및 OpenCL과 Metal을 사용하는 Huawei P50 Pro와 iPhone 11
- 서버 측 테스트에는 AMD Ryzen 9 3900X(x86), Alibaba Cloud의 ecs.g6e.4xlarge(인텔 제온(캐스케이드 레이크) 플래티넘 8269CY, 16 vCPU, 64GiB 메모리), NVIDIA Geforce RTX 2080 Ti를 사용하여 각각 4 스레드의 AVX256/AVX512/CUDA를 지원

❖ MNN과 TVM을 비교: 빈번하고 빠른 작업 반복이 필요한 이기종 디바이스일 때



- 수동 작업자 수준 및 백엔드 수준 최적화를 통해 각 백엔드 서버에서 각 모델에 대한 추론 시간 측면에서 MNN이 TVM보다 성능이 뛰어나다.

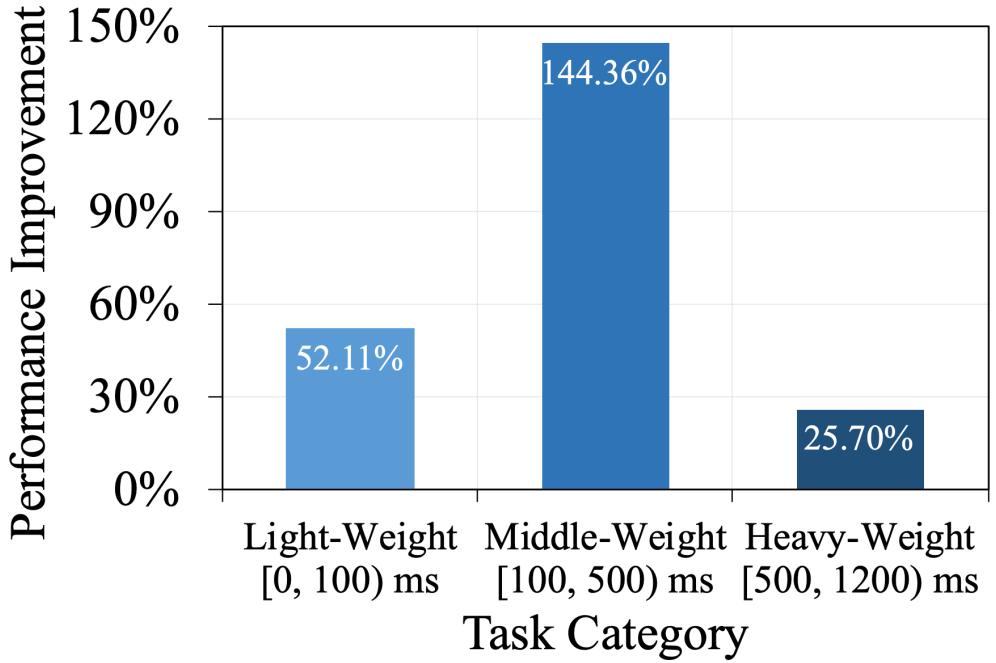
✓ TVM 자동 튜닝과 컴파일에 대략 수천 s가 소요 vs 런타임 최적화를 위한 MNN의 반자동 검색은 약 수백 ms의 비용이 소요

✓ MNN은 수많은 이기종 디바이스의 빈번하고 빠른 작업 반복이 필요한 산업 시나리오에 더 적합하지만 TVM은 그렇지 않음

Evaluation: Benchmark

❖ Python VM

: 성능을 태스크 실행 시간의 역수로 정의

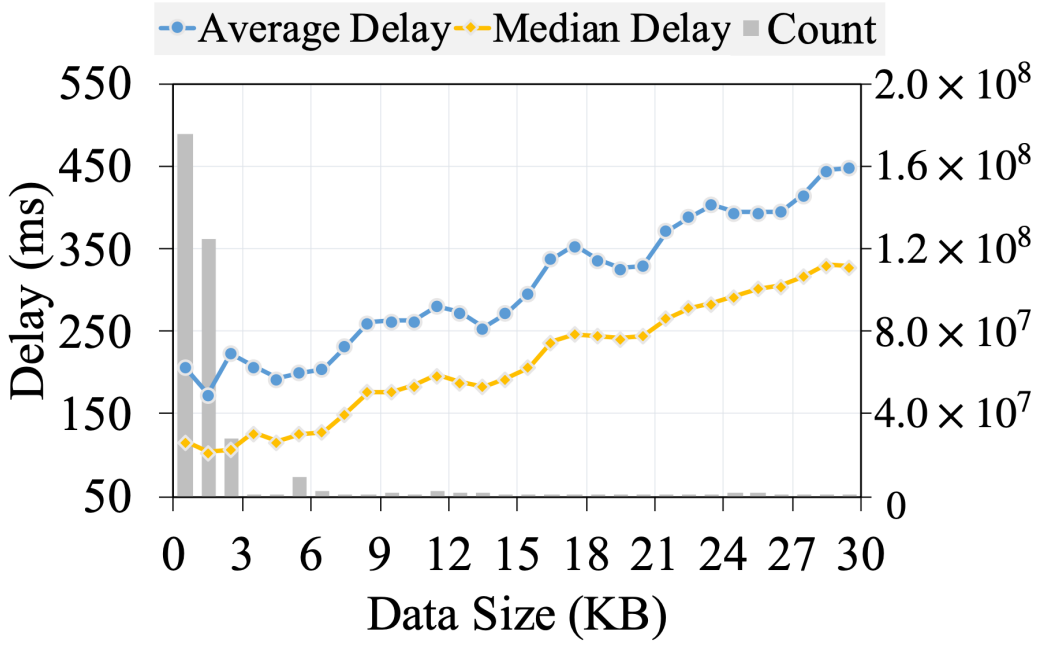


- 경량, 중간 중량, 중량 태스크의 경우 Python 스레드 수준 VM은 각각 52.11%, 144.36%, 25.70%의 성능 개선

⇒ GIL이 없는 작업 수준 멀티스레딩이 성능 향상의 핵심

❖ RealTime Tunnel

: 다양한 데이터 크기에 대한 지연 시간과 업로드 횟수 평가



- 90% 이상의 업로드가 3KB 미만이면서 평균 250ms 미만
- 0.1% 업로드의 크기가 30KB로 증가하더라도 평균 지연 시간은 약 450ms까지만 증가

⇒ 실시간 터널이 Walle의 배포 플랫폼에서 높은 확장성 & 적시성

Summary

❖ Walle

- 디바이스-클라우드 협업 머신 러닝을 위한 시스템: 긴 대기 시간, 높은 오버헤드 비용, 높은 서버 로드, 높은 개인 정보 보호 및 보안 위험과 같은 주류 클라우드 서버 기반 머신 러닝 프레임워크의 병목 현상을 해결
- 모바일 Inference 작업을 최적화