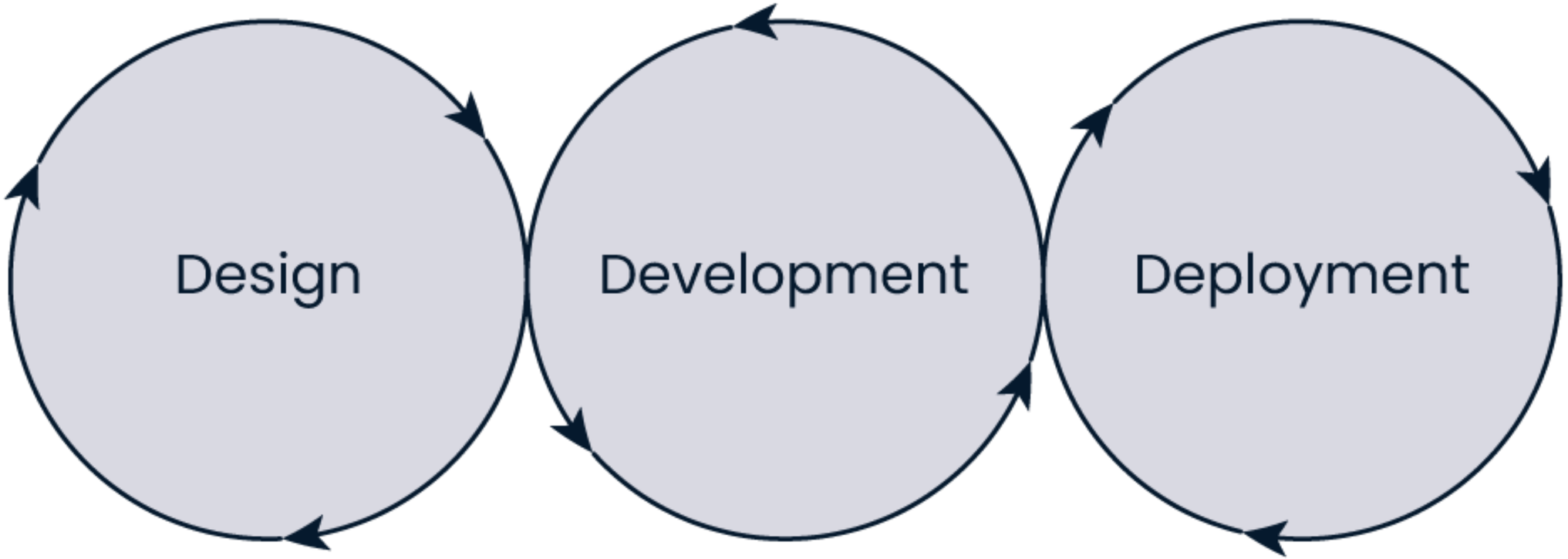
Preparing model for deployment

MLOPS CONCEPTS

Runtime environment



- Added value
- Business requirements
- Key metrics
- Data processing

- Feature engineering
- Experiment tracking
- Model training & evaluation

- Runtime environments
- Microservices architecture
- CI/CD pipeline
- Monitoring & retraining

MLOps에서 Runtime Environment(런타임 환경)는 **머신러닝 모델의 실행과 관련된 모든 소프트웨어, 라이브러리, 설정, 그리고 하드웨어 환경**을 의미

- 이는 머신러닝 모델이 배포 및 운영되는 환경에서 안정적이고 일관되게 작동하도록 보장하기 위해 필수적인 요소

Runtime Environment의 구성 요소

1.소프트웨어 스택 (Software Stack):
모델 실행에 필요한 모든 소프트웨어 구성 요소를 포함.
예: 운영 체제, Python 인터프리터, 라이브러리, 드라이버 등.

2.종속 라이브러리 (Dependencies):
머신러닝 모델 실행에 필요한 라이브러리 및 프레임워크.
예: TensorFlow, PyTorch, NumPy, Pandas, Scikit-learn 등.

3.하드웨어 환경 (Hardware Environment):
모델 실행에 필요한 물리적 또는 가상 하드웨어.
예: CPU, GPU, TPU, 메모리, 디스크 등.

4.환경 설정 (Environment Configuration):
모델 실행에 필요한 설정 정보.
예: 환경 변수, 파일 경로, 네트워크 설정.

5.컨테이너화 도구 (Containerization Tools):
런타임 환경을 패키징하고 배포하기 위해 사용.
예: Docker, Kubernetes.

Runtime Environment의 중요성

1.재현성 (Reproducibility):
동일한 코드와 데이터를 사용하더라도, 실행 환경이 다르면 결과가 달라질 수 있음.
런타임 환경을 일관되게 설정하면 모델의 결과를 재현할 수 있음.

2.이식성 (Portability):
특정 환경에서 잘 작동하는 모델을 다른 환경으로 쉽게 옮길 수 있음.
예: 개발 환경에서 실행한 모델을 프로덕션 환경으로 배포.

3.디버깅과 문제 해결:
환경 문제(예: 라이브러리 버전 충돌)가 모델의 성능 저하나 예기치 않은 결과를 초래할 수 있음.
명확한 런타임 환경 정의는 문제 해결을 용이하게 함.

4.스케일링 (Scalability):
동일한 환경 설정으로 여러 인스턴스에서 모델을 실행할 수 있어 확장 가능성을 높임.

Runtime Environment의 설계 원칙

1.일관성 유지:
개발 환경과 배포 환경 간의 차이를 최소화.
예: 동일한 Docker 이미지 사용.

2.경량화:
런타임 환경을 구성할 때 필요하지 않은 요소는 제외하여 크기를 최소화.
예: Slim Docker 이미지 사용.

3.확장성 고려:
모델이 대규모 트래픽을 처리할 수 있도록 설계.
GPU/TPU 활용 가능성 포함.

4.보안 강화:
환경 설정 파일과 민감 정보(예: API 키, 비밀번호)는 안전하게 관리.
예: Vault, 환경 변수 암호화.

Runtime Environment 관리 도구

1.환경 정의:

- **Docker**:
  - 컨테이너를 사용하여 실행 환경을 패키징.
  - 코드, 라이브러리, 설정 파일을 포함한 "모든 것"을 하나의 이미지로 관리.
- **Conda/Virtualenv**:
  - Python 패키지와 환경을 격리.

2.오케스트레이션 도구:

- **Kubernetes**:
  - 컨테이너화된 애플리케이션을 대규모로 배포하고 관리.

3.종속성 관리:

- **Pip requirements**:
  - requirements.txt 파일로 Python 라이브러리의 종속성을 관리.
- **Poetry**:
  - 프로젝트 종속성 및 가상 환경 관리.
- **Pyenv**:
  - 다양한 Python 버전을 프로젝트별로 관리.

4.프로덕션 환경 모니터링:

- **Prometheus, Grafana**:
  - 런타임 상태와 성능 모니터링.
- **Sentry**:
  - 오류 및 충돌 감지.

실제 MLOps에서 Runtime Environment 사용 예시

1. 머신러닝 모델 배포
모델 학습 및 저장:
- 개발 환경에서 학습된 모델을 저장.

런타임 환경 패키징:
- Docker 이미지를 생성.
- 예: Dockerfile 작성.

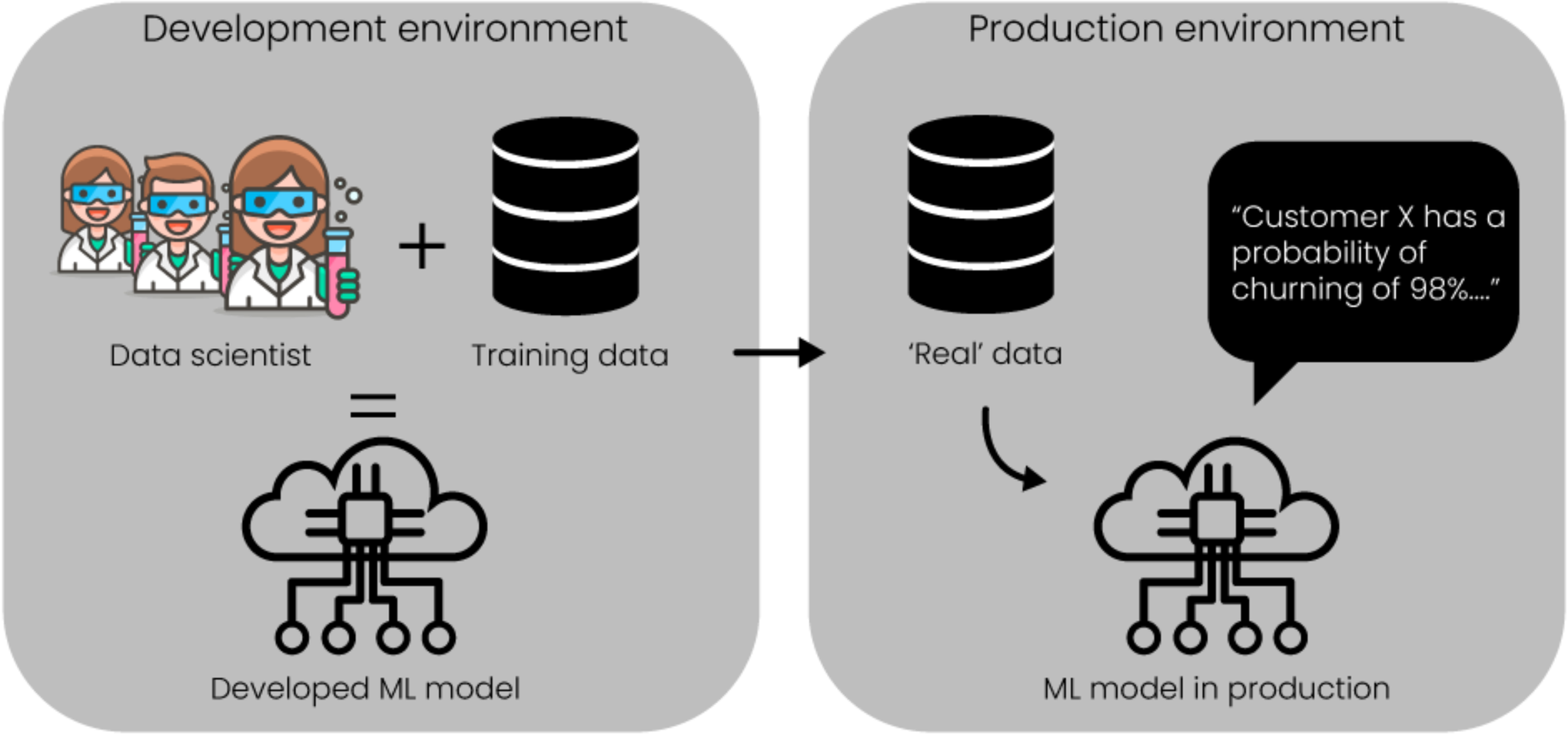
•배포:
- Kubernetes 클러스터에 컨테이너 배포.

2. API를 통한 예측 제공
런타임 환경에서 Flask/FastAPI를 통해 모델 예측 결과 제공.
Docker 컨테이너를 활용하여 동일한 API 환경 유지.

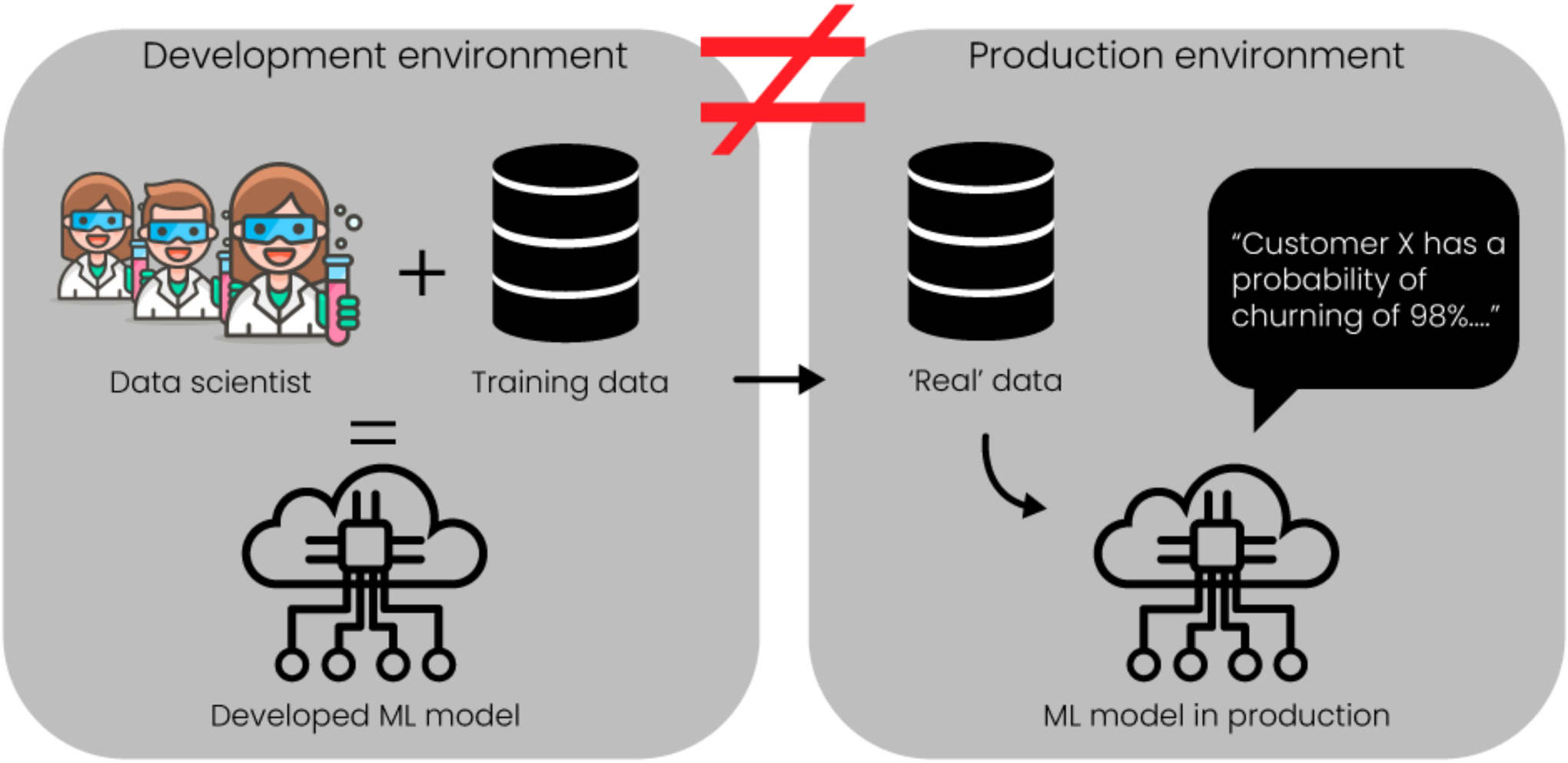
3. 데이터 파이프라인 통합
데이터 수집, 전처리, 모델 예측을 위한 통합 런타임 환경 구성.
예: Apache Airflow와 함께 동작하는 Docker 기반 데이터 처리 파이프라인.

MLOps에서 **Runtime Environment**는 머신러닝 모델의 안정적인 실행과 배포를 보장하는 데 핵심적인 역할을 함
- 이를 효과적으로 관리하면 재현성과 확장성이 확보되고, 배포와 운영 단계에서의 오류 가능성을 줄일 수 있음
- Docker와 같은 컨테이너 도구, Kubernetes 오케스트레이션, 종속성 관리 시스템을 활용하면 이러한 환경을 체계적으로 설계하고 유지할 수 있음

Development to deployment

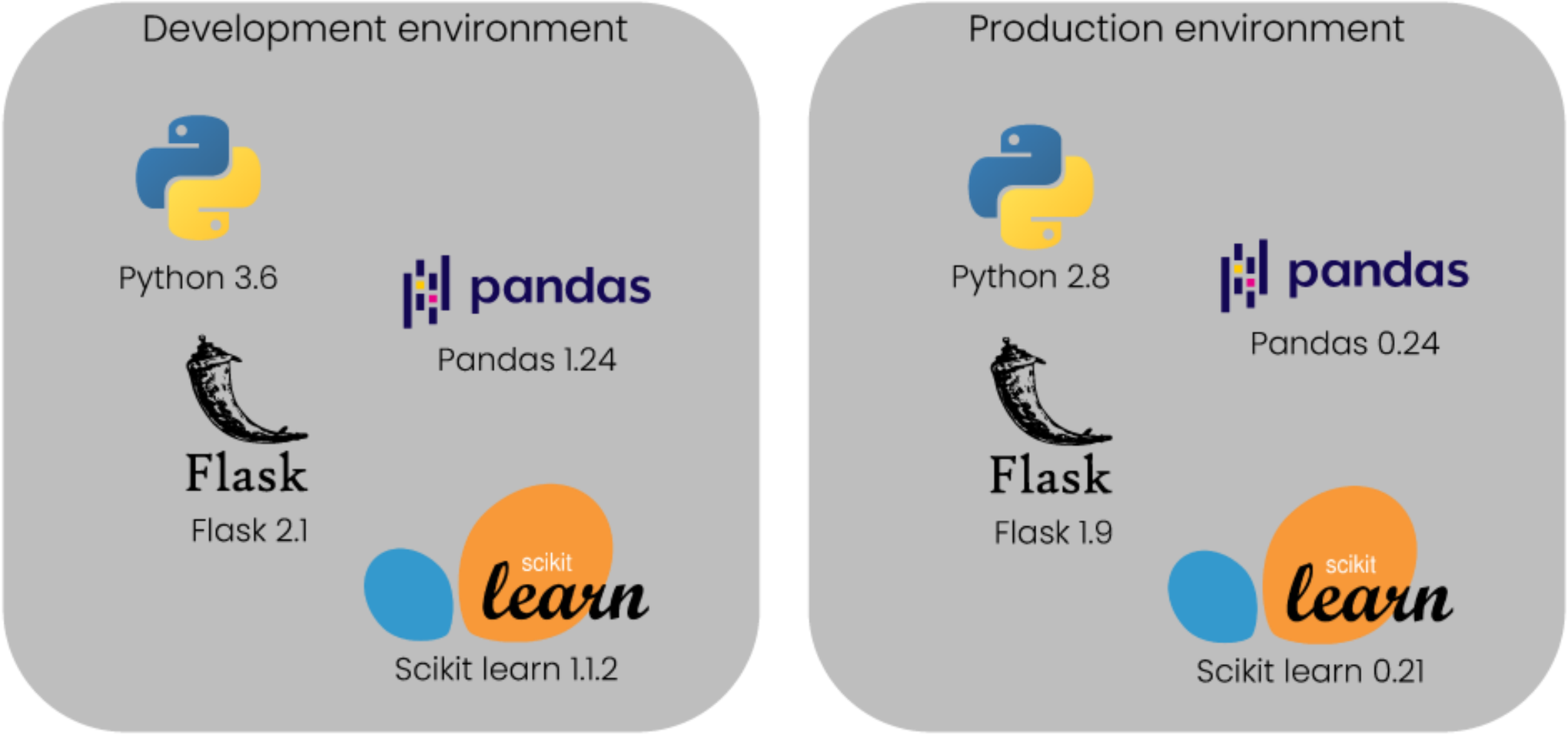


Development to deployment

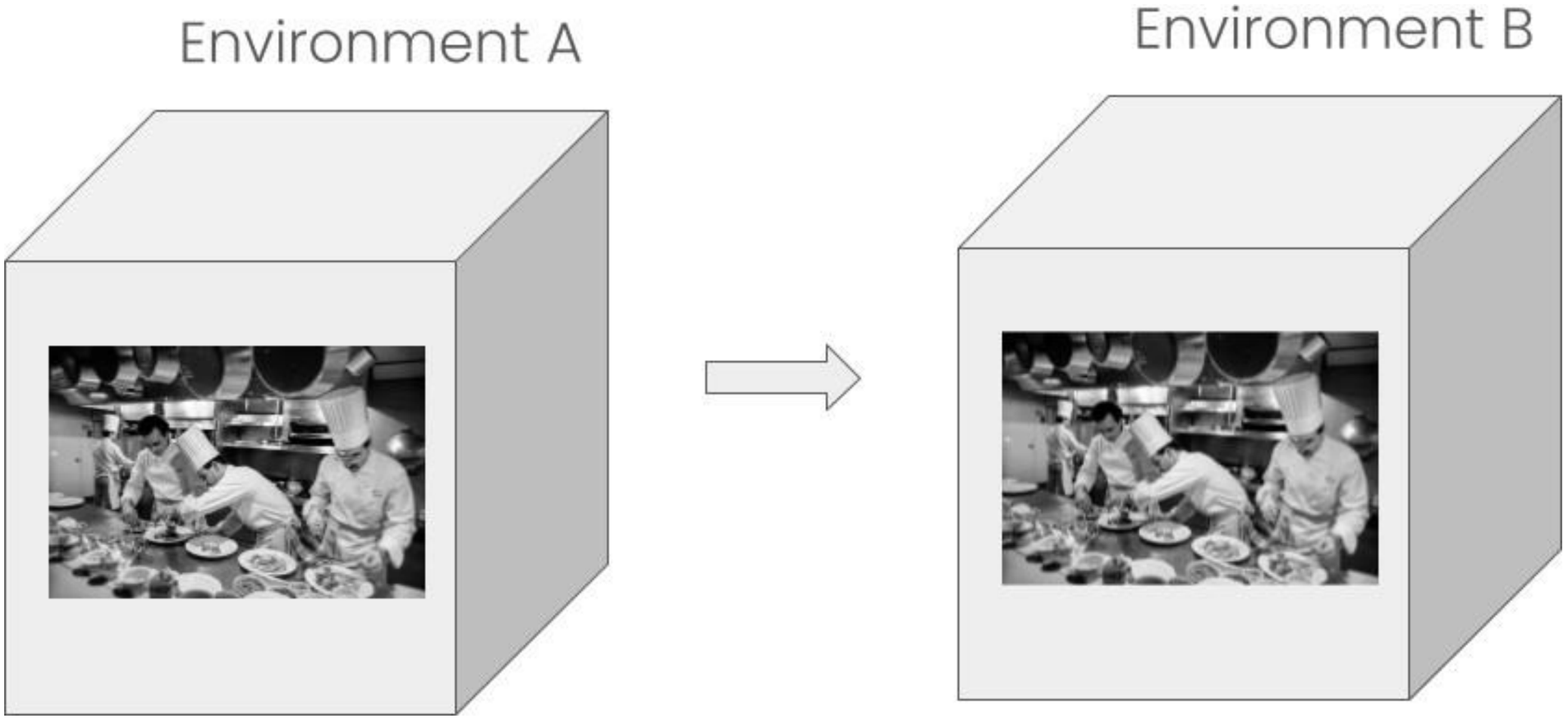


Runtime environments

Runtime environments



Container



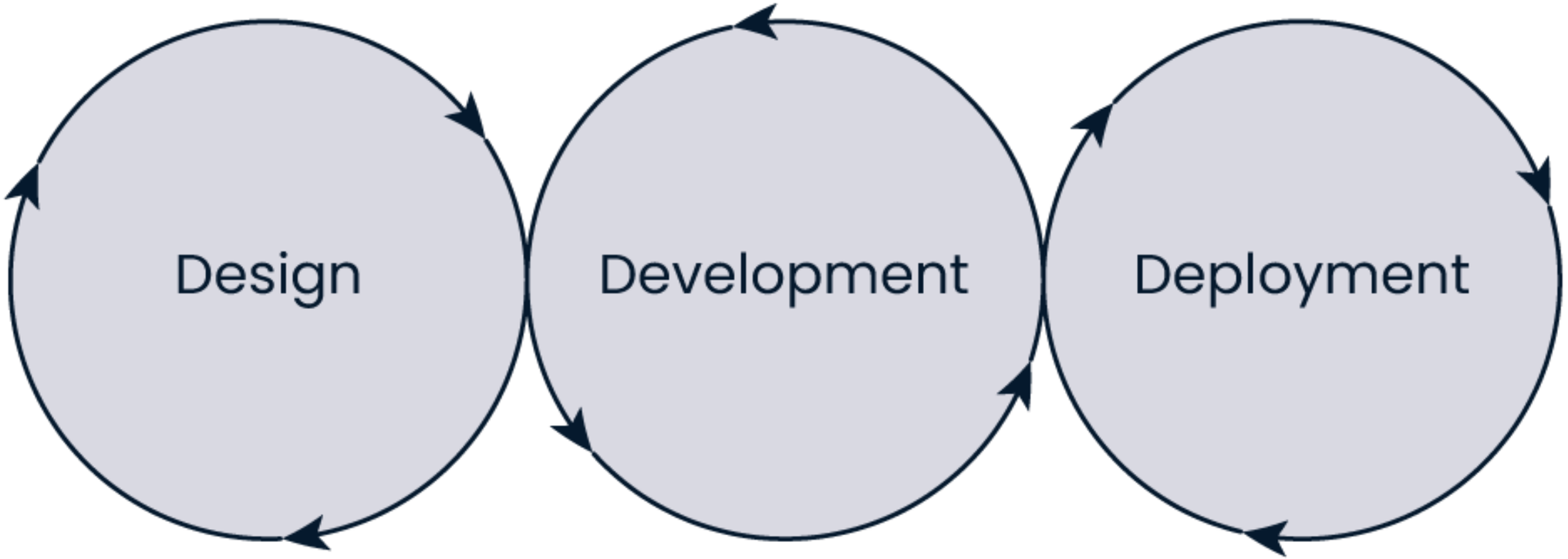
Benefits containers

- Easier to maintain
- Portable
- Fast to start up

Machine learning deployment architecture

MLOPS CONCEPTS

Microservices architecture



- Added value
- Business requirements
- Key metrics
- Data processing

- Feature engineering
- Experiment tracking
- Model training & evaluation

- Runtime environments
- Microservices architecture
- CI/CD pipeline
- Monitoring & retraining

Microservice Architecture(마이크로서비스 아키텍처)는 애플리케이션을 독립적으로 배포하고 관리할 수 있는 작은 서비스들로 나누는 설계 방식

- MLOps에서 마이크로서비스 아키텍처는 머신러닝 모델 및 관련 기능을 **작은 독립형 서비스**로 구축하고, 이를 조합하여 전체 시스템을 운영하는 방법론을 의미

Microservice Architecture의 주요 특징

1.독립적인 서비스:
각각의 서비스가 독립적으로 개발, 배포, 확장 가능.
예: 데이터 수집 서비스, 모델 학습 서비스, 모델 배포 서비스 등이 독립적으로 실행.

2.서비스 간 통신:
각 서비스는 HTTP REST API, gRPC, 메시지 큐(RabbitMQ, Kafka) 등을 통해 통신.

3.유연성:
특정 서비스만 업데이트하거나 확장 가능.
예: 새로운 모델을 배포해도 데이터 수집 시스템에는 영향을 미치지 않음.

MLOps에서 Microservice Architecture의 역할

1.모델 배포 및 관리:
각 머신러닝 모델을 개별 마이크로서비스로 배포.
새로운 모델 배포 시 기존 시스템에 최소한의 영향을 미침.

2.스케일링:
트래픽 증가 시 특정 서비스(예: 모델 추론 API)를 독립적으로 확장.
자원을 효율적으로 사용.

3.유지보수와 디버깅:
문제 발생 시 해당 서비스만 수정 가능.
전체 시스템을 중단하지 않고 부분적으로 업데이트 가능.

4.다중 모델 지원:
여러 모델을 각각의 마이크로서비스로 운영.
예: 추천 모델, 예측 모델, 분류 모델을 별도로 관리.

Microservice Architecture의 장점

1.확장성 (Scalability):
트래픽이 높은 서비스만 독립적으로 확장 가능.
예: 추론 API를 GPU 인스턴스에서 스케일링.

2.유연성 (Flexibility):
특정 서비스만 수정하거나 교체 가능.
예: 모델 학습 로직을 업데이트하면서도 배포 서비스는 그대로 유지.

3.고가용성 (High Availability):
하나의 서비스 장애가 전체 시스템에 영향을 미치지 않음.
예: 데이터 전처리 서비스가 실패해도 추론 서비스는 계속 작동.

4.다중 모델 및 워크로드 지원:
여러 모델을 동시에 운영하며 서로 다른 워크로드를 처리 가능.

5.효율적인 개발 및 배포:
서비스별로 팀을 나누어 병렬로 개발 가능.
각 서비스는 독립적인 배포 주기를 가짐.

Microservice Architecture의 단점 및 도전 과제

1.복잡성 증가:
서비스 수가 많아질수록 아키텍처 관리가 어려워짐.
예: 서비스 간 통신, 인증, 로깅 관리의 복잡성.

2.통합 테스트:
여러 서비스 간의 상호작용을 테스트하는 과정이 복잡.

3.성능 오버헤드:
네트워크를 통한 서비스 간 통신으로 인한 지연 발생.

4.모니터링 및 디버깅 어려움:
각 서비스의 로그와 메트릭을 통합적으로 관리해야 함.

MLOps에서의 활용 예제
마이크로서비스 아키텍처 구성 예

데이터 처리 서비스:
- Kafka로 실시간 데이터 수집 및 전처리.

모델 학습 서비스:
- 새로운 데이터가 들어오면 모델 재학습.

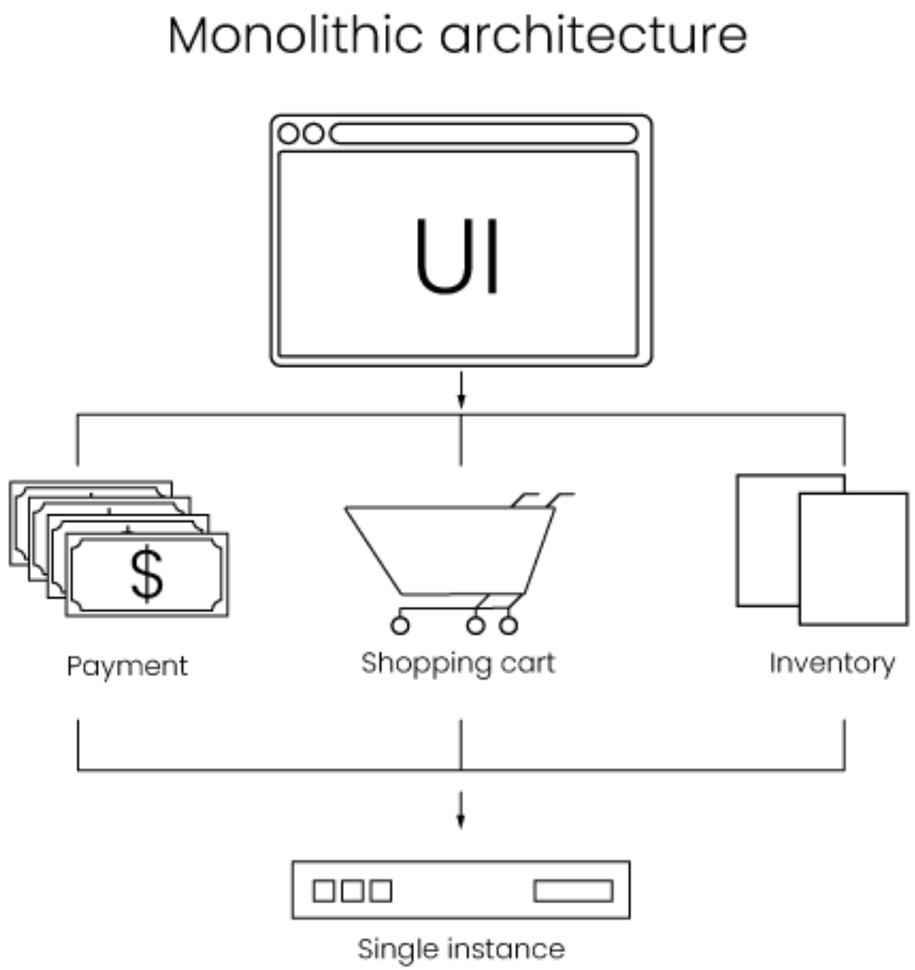
모델 배포 서비스:
- 학습된 모델을 TensorFlow Serving으로 API 제공.

모니터링 서비스:
- Evidently AI로 데이터 드리프트와 모델 성능 모니터링.

MLOps에서 **Microservice Architecture**는 머신러닝 시스템의 **유연성**, **확장성**, **관리 효율성**을 극대화하는 설계 방식

- 각 기능을 독립적인 서비스로 구성함으로써 시스템이 더 안정적이고 유지보수가 용이해짐
- 그러나 복잡성이 증가하므로 서비스 간 통신, 모니터링, CI/CD 파이프라인의 구축이 필수
- Microservice Architecture는 특히 대규모 데이터와 다중 모델 환경에서 효과적

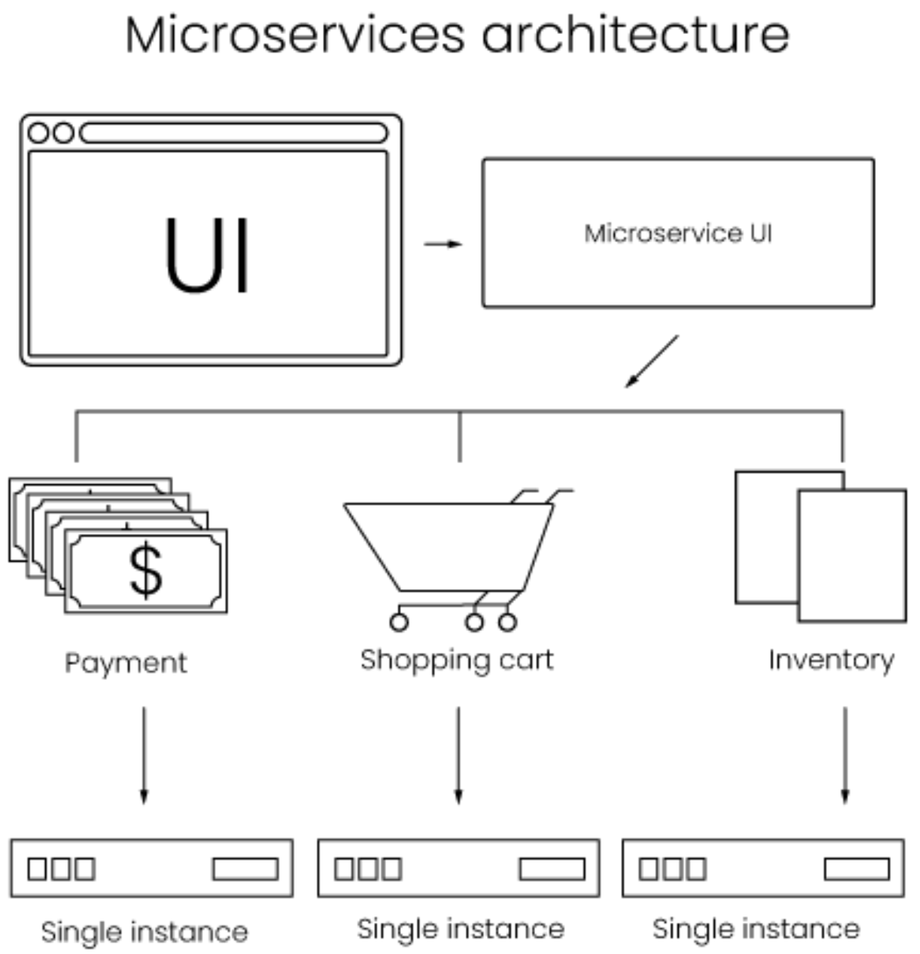
Monolith vs. microservice architecture



- One uniform application containing all services

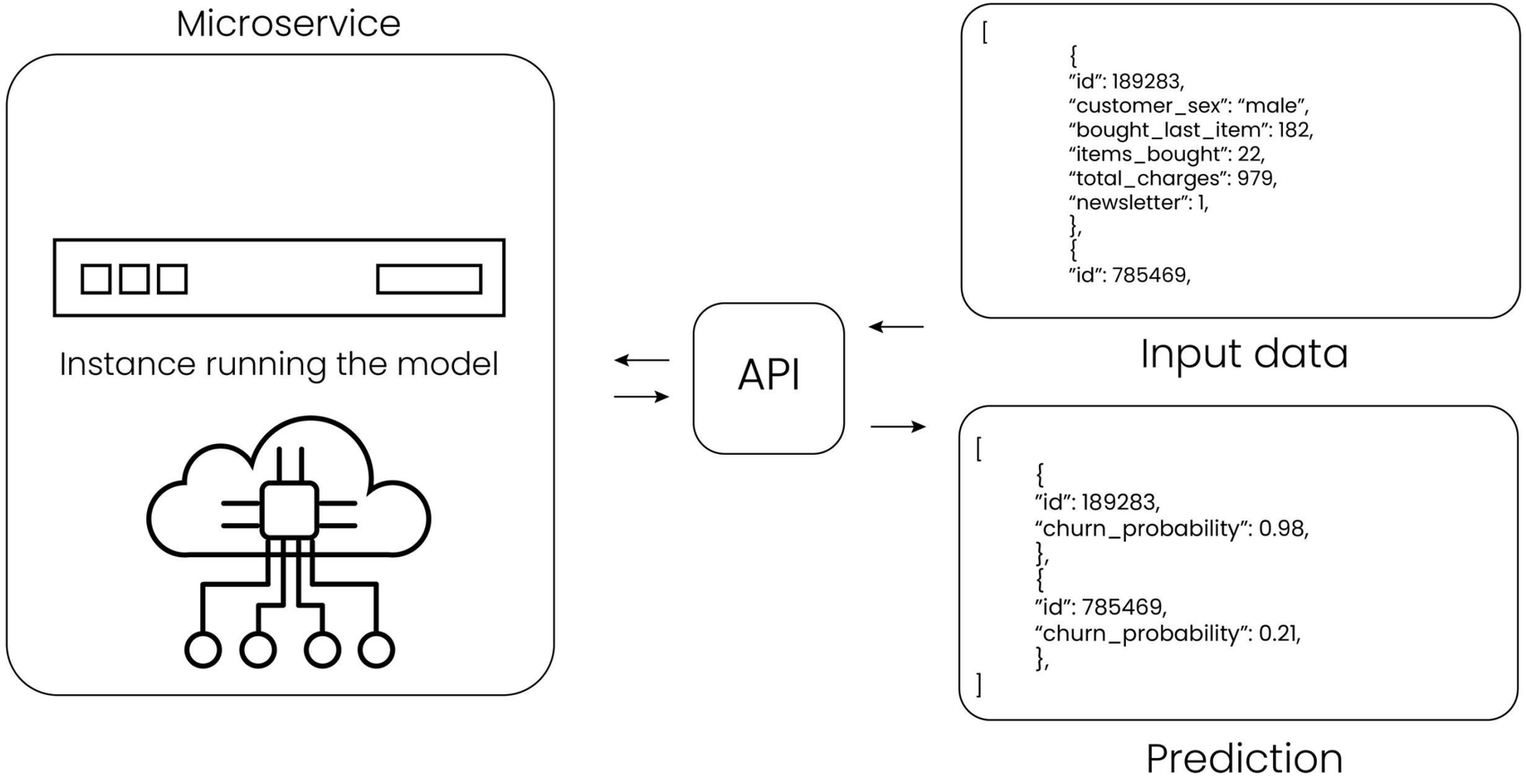
Monolith vs. microservice architecture

- Collection of smaller, independent services

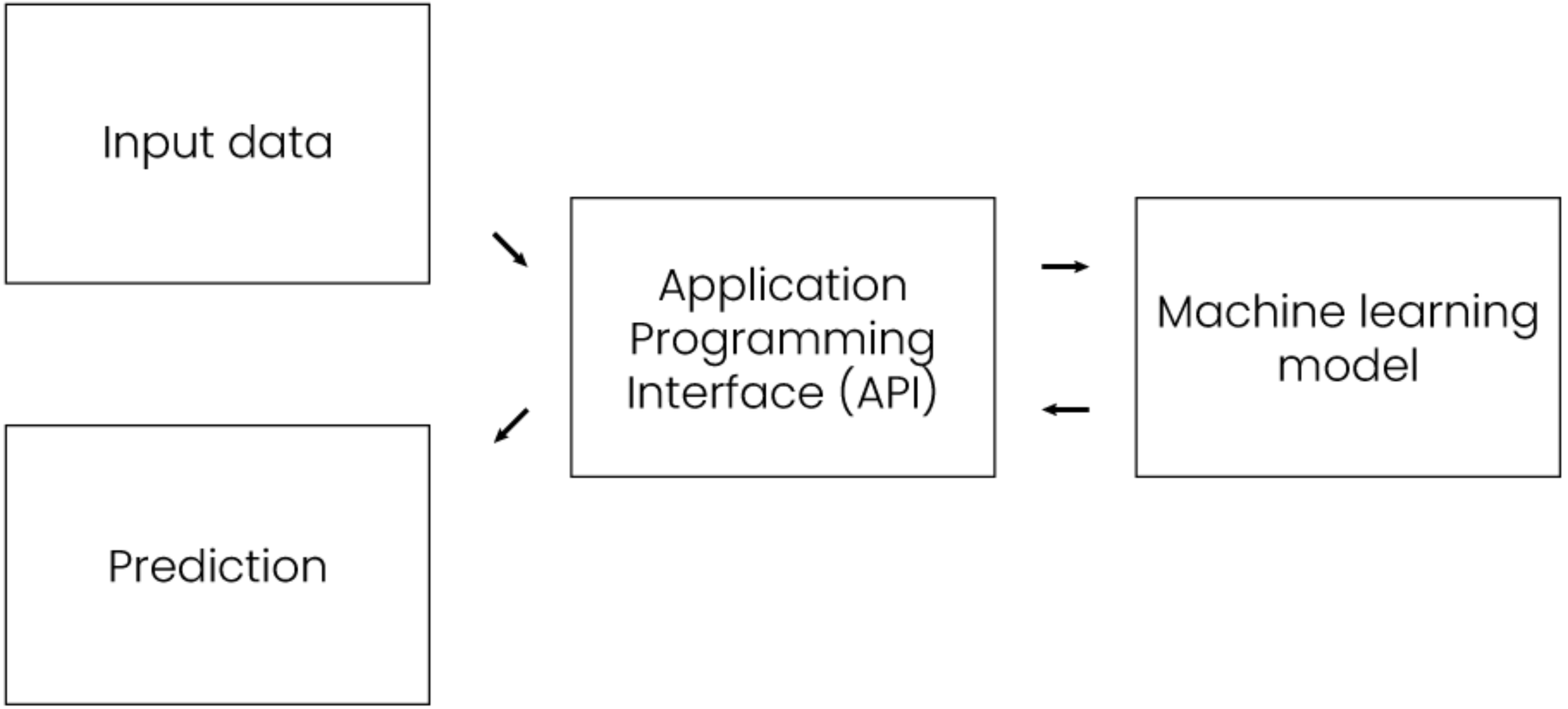


Inferencing

Inferencing is the process in which we send new input to the machine learning model and receive output from the model.



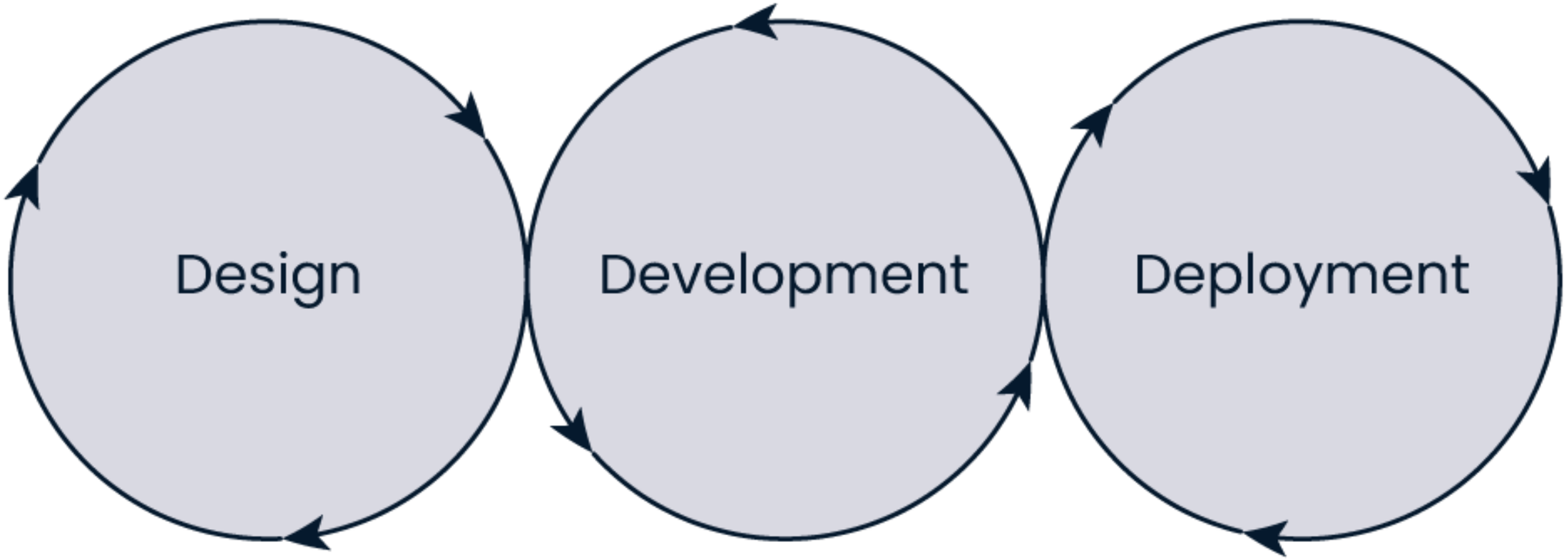
Application Programming Interface (API)



CI/CD and deployment strategy

MLOPS CONCEPTS

CI/CD pipeline

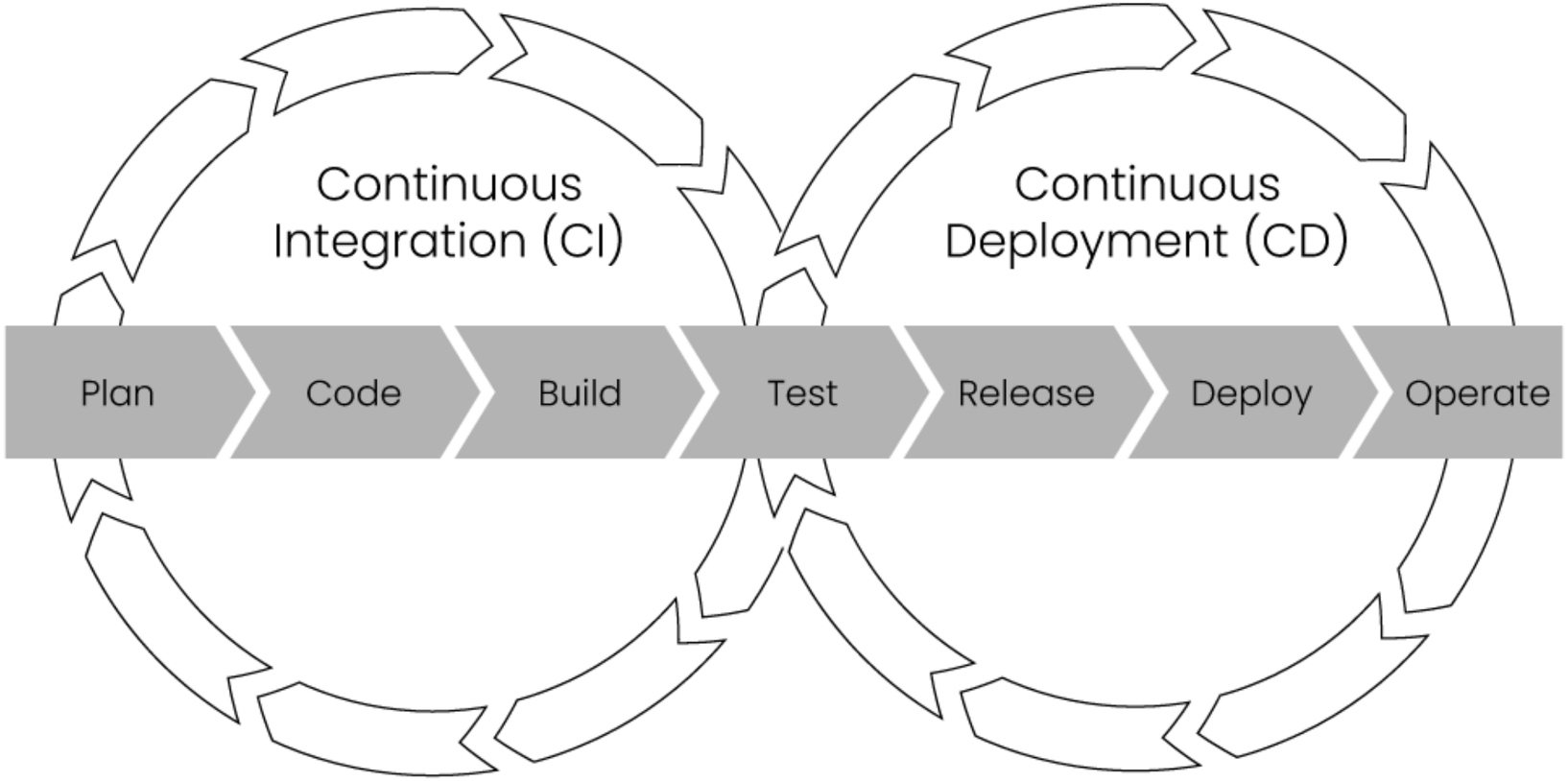


- Added value
- Business requirements
- Key metrics
- Data processing

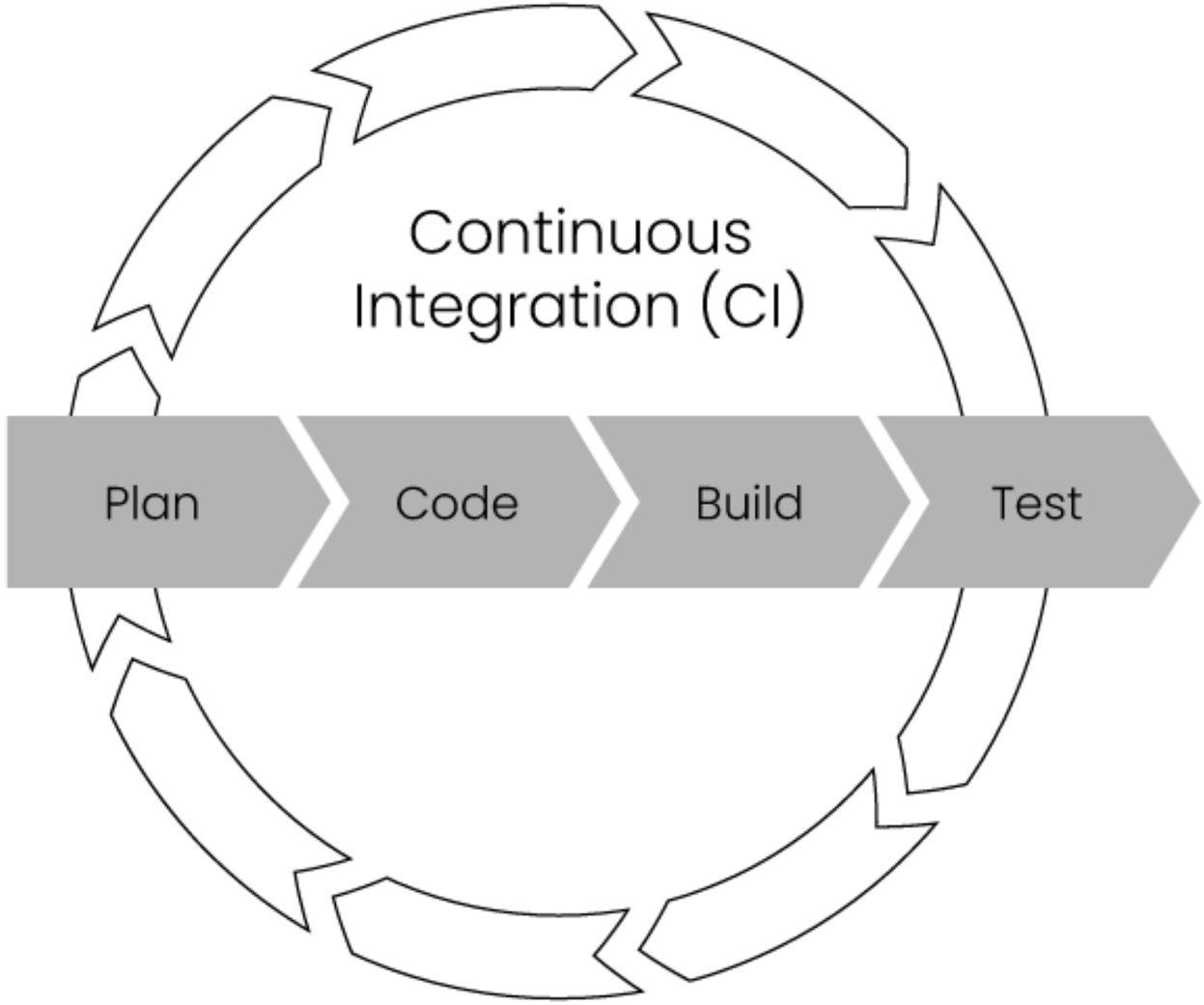
- Feature engineering
- Experiment tracking
- Model training & evaluation

- Runtime environments
- Microservices architecture
- CI/CD pipeline
- Monitoring & retraining

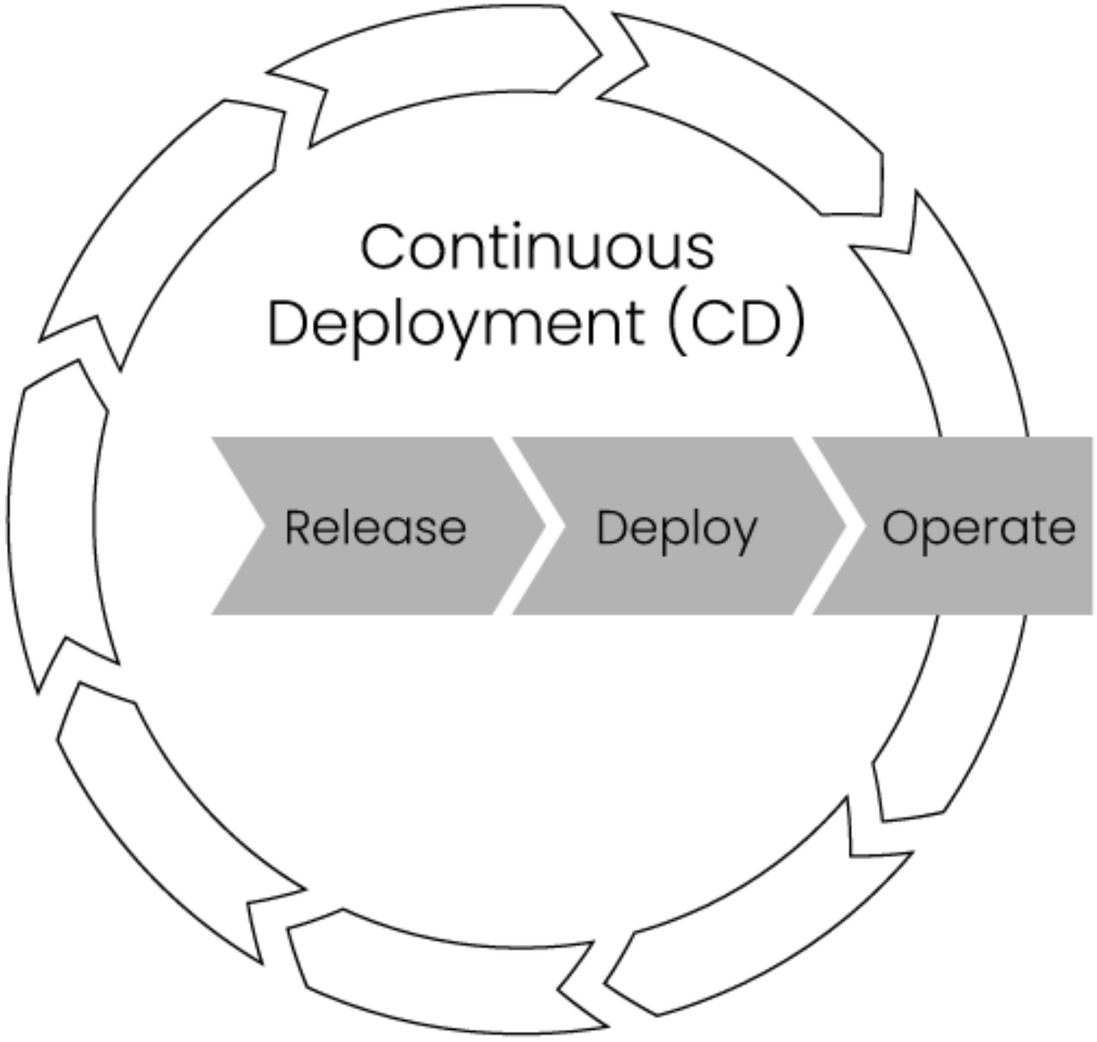
CI/CD pipeline



Continuous Integration



Continuous Deployment



Deployment strategies

- Basic deployment
- Shadow deployment
- Canary deployment

Basic Deployment

Basic Deployment는 기존 모델을 **새로운 모델로 즉시 교체**하는 가장 단순한 배포 방식
새로운 모델이 배포되면 이전 모델은 즉시 사용되지 않으며, 모든 트래픽이 새로운 모델로 전환됨

장점
1. 단순함:
    설정과 구현이 간단하여 빠르게 적용 가능.
2. 리소스 효율성:
    추가 리소스나 트래픽 분할이 필요하지 않아 리소스를 절약.

단점
1. 위험성:
    새 모델에 문제가 있을 경우 전체 서비스가 영향을 받을 수 있음.
2. 롤백 어려움:
    문제가 발생하면 빠른 롤백이 어렵거나 복잡할 수 있음.

사용 사례
테스트가 충분히 완료된 상태에서 비교적 작은 위험이 예상되는 변경 사항을 배포할 때 사용.

Shadow Deployment

Shadow Deployment는 새로운 모델을 프로덕션 환경에 배포하지만, **실제 사용자 요청에 영향을 미치지 않고** 요청의 복사본을 새 모델로 전달하여 모델 성능을 테스트하는 방식
결과는 로그나 분석에 사용되며, 사용자 응답에는 영향을 주지 않습니다.

작동 방식
1. 사용자 요청은 기존 모델로 전달되어 응답을 생성
2. 요청의 복사본은 새로운 모델로 전달되지만, 이 모델의 응답은 사용자에게 반환되지 않음
3. 두 모델의 결과를 비교하여 새로운 모델의 성능을 평가

장점
1.안전성:
    새로운 모델의 문제가 사용자 경험에 영향을 미치지 않음.
2.실제 트래픽 테스트:
    실시간 트래픽으로 모델 성능을 평가 가능.
3.성능 분석:
    기존 모델과 새로운 모델의 결과를 비교하여 품질 보증.

단점
1.리소스 요구:
    기존 모델과 새로운 모델 모두 요청을 처리해야 하므로 추가적인 리소스가 필요.
2.복잡성 증가:
    트래픽 복제 및 분석 시스템 설정이 복잡할 수 있음.

사용 사례
대규모 서비스에서 모델 변경이 사용자 경험에 중요한 영향을 미칠 가능성이 높을 때 사용.
기존 모델과 새로운 모델의 성능을 실시간으로 비교하고 성능 데이터를 수집할 때.

Canary Deployment는 새로운 모델을 일부 사용자 트래픽에만 적용하여 성능과 안정성을 검증하는 방식
초기에는 소수의 트래픽에 적용하고, 안정성이 확인되면 점진적으로 더 많은 트래픽을 새로운 모델로 전환

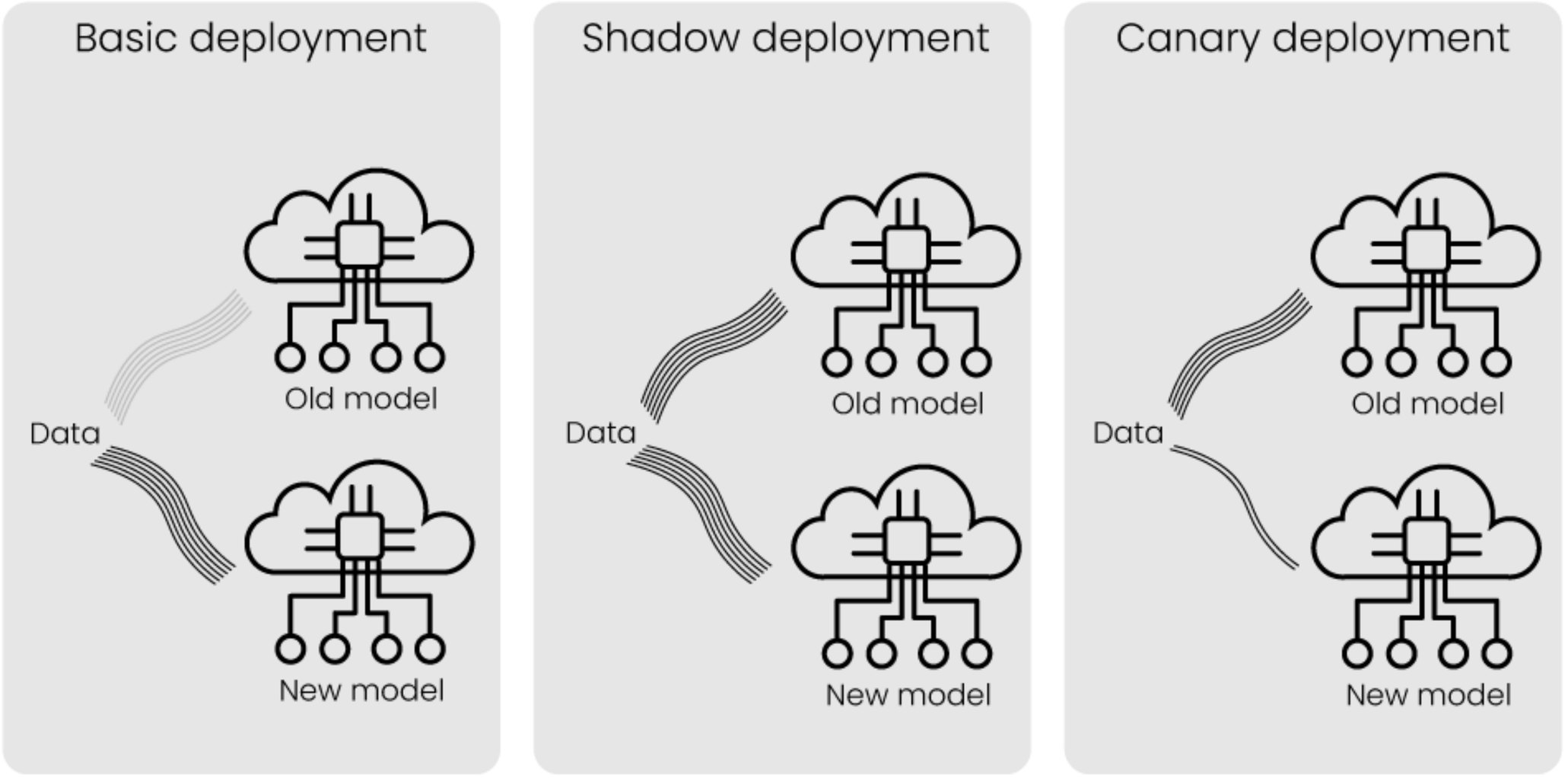
작동 방식
1. 초기 단계:
   새로운 모델이 전체 트래픽의 1%~5%를 처리.
2. 안정성 검증:
   오류율, 성능, 사용자 피드백 등을 평가.
3. 단계적 확장:
   안정성이 확인되면 점진적으로 트래픽 비율을 증가.
4. 완전 배포:
   새로운 모델이 모든 트래픽을 처리하도록 전환.

장점
1. 리스크 완화:
   단계적으로 배포하므로 새로운 모델의 문제를 조기에 발견 가능.
2. 사용자 기반 피드백:
   초기 트래픽에서 사용자 경험과 성능 데이터를 수집.
3. 롤백 용이성:
   문제가 발생하면 초기 트래픽 비율로 쉽게 롤백 가능.

단점
1. 복잡한 관리:
   트래픽 분할과 단계적 확장이 필요하므로 관리가 복잡할 수 있음.
2. 추가 리소스:
   두 모델을 동시에 운영해야 하는 경우 리소스 부담 증가.

사용 사례
모델 성능이 중요한 프로덕션 환경에서 새로운 모델의 안정성을 점검하며 배포할 때.
대규모 사용자 기반에서 신중한 변경이 필요한 상황.

Deployment strategies



Deployment strategies

Strategy	Pros	Cons
Basic deployment	Straightforward, easy to implement, low resources	High risk if the model does not work as expected.
Shadow deployment	Easy to implement, no risk if model does not work as expected	Double resources.
Canary deployment	Small risk if model does not work as expected.	Slightly harder to implement, medium amount of resources.

Automation and scaling

MLOPS CONCEPTS

Automation의 개념
자동화는 머신러닝 워크플로우의 반복적이고 수동적인 작업을 자동으로 수행하도록 설정하는 것을 의미
모델 개발부터 배포, 모니터링, 재학습까지의 모든 과정을 자동화하여 개발 및 운영 속도를 향상시키고 오류를 줄임

자동화가 필요한 영역
1. 데이터 준비 및 처리:
    데이터 수집, 정제, 변환, 저장 과정을 자동화.
    예: 스케줄링 도구로 데이터를 정기적으로 파이프라인에 공급.
2. 특성 엔지니어링:
    특성 생성, 선택, 변환 작업 자동화.
    예: Feature Store를 활용하여 재사용 가능한 특성 관리.
3. 모델 학습 및 실험:
    모델 학습과 하이퍼파라미터 튜닝을 자동화.
    예: AutoML 도구를 통해 다양한 알고리즘과 매개변수를 실험.
4. 모델 배포:
    모델을 프로덕션 환경으로 자동으로 배포.
    예: CI/CD 파이프라인을 설정하여 새로운 모델을 자동 배포.
5. 모니터링 및 알림:
    배포된 모델의 성능 및 데이터 변화를 실시간 모니터링하고 경고 알림을 자동으로 생성.
6. 재학습 및 업데이트:
    성능 저하가 감지되면 새 데이터를 기반으로 모델을 재학습하고 업데이트.

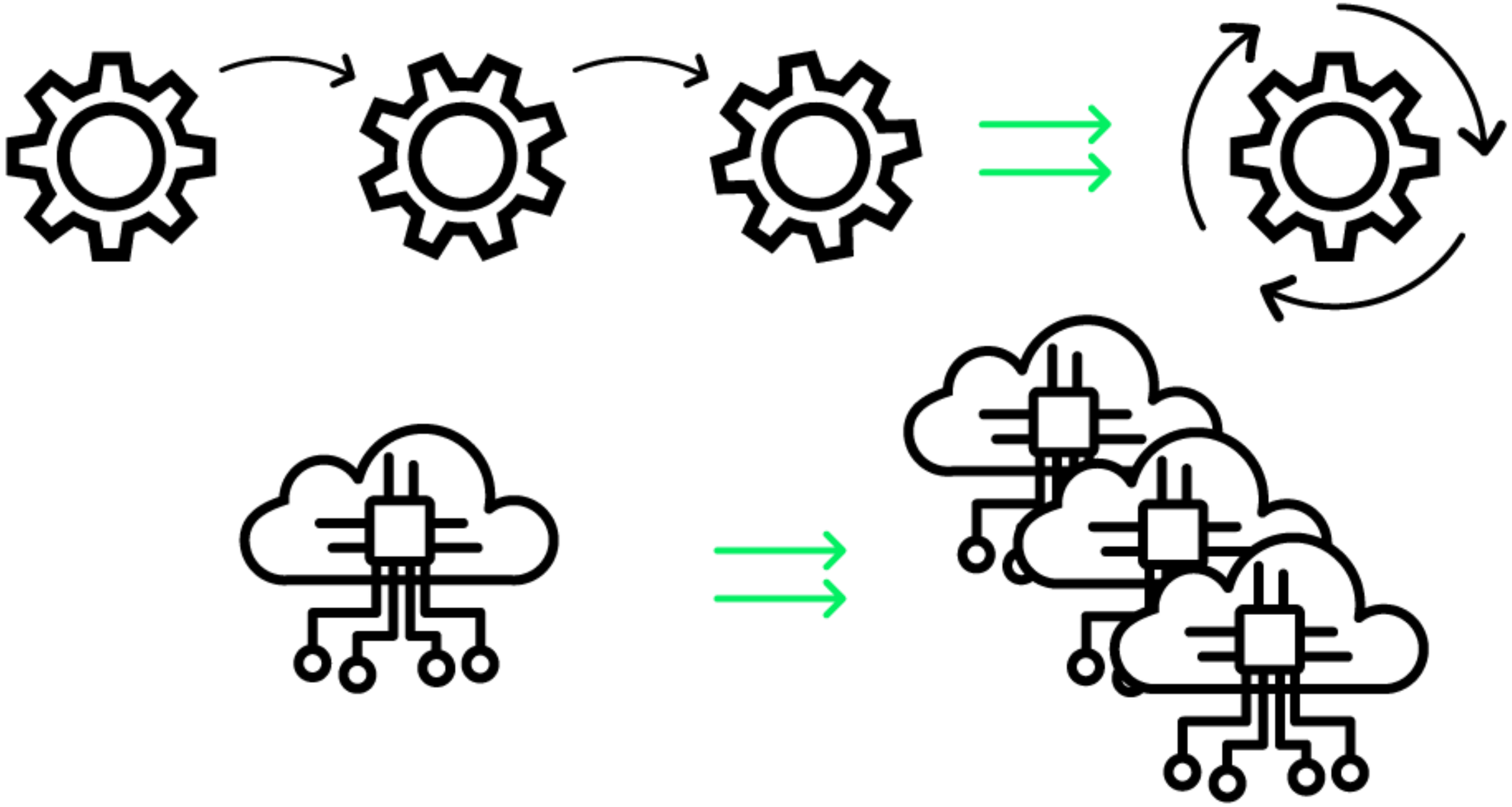
Automation의 장점
효율성: 수동 작업을 줄여 시간과 리소스를 절약.
일관성: 파이프라인의 각 단계에서 동일한 결과를 보장.
빠른 배포: 개발에서 배포까지의 시간을 단축.
에러 감소: 수동 개입을 줄임으로써 인간 실수를 최소화.

Scaling의 개념
확장성은 데이터, 사용자 수, 처리 요구사항이 증가하더라도 시스템이 안정적이고 효율적으로 작동할 수 있는 능력을 의미.
MLOps에서 확장성은 대규모 데이터 처리, 다중 모델 배포, 실시간 서비스 제공과 같은 요구사항을 충족하는 데 필수

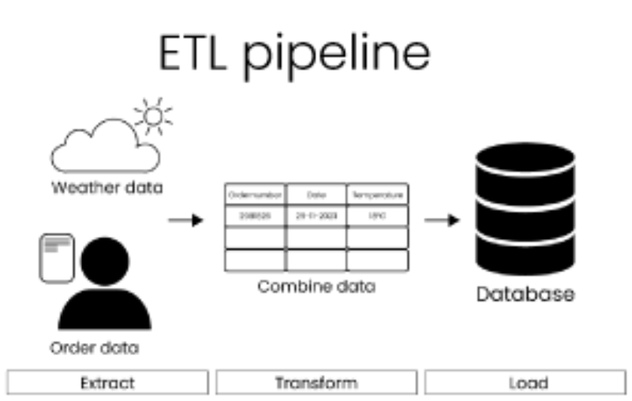
확장이 필요한 영역
1. 데이터 스케일링:
데이터의 크기와 복잡도가 증가해도 데이터 준비와 학습 파이프라인이 안정적으로 작동해야 함.
예: 분산 데이터 저장소와 처리 시스템 사용(Apache Spark, Hadoop).
2. 모델 학습 스케일링:
모델 학습 프로세스를 병렬 처리하거나 분산 학습을 통해 처리 속도를 높임.
예: Horovod, PyTorch Distributed Training.
3. 모델 배포 스케일링:
여러 사용자 요청을 처리하기 위해 모델 배포를 확장.
예: Kubernetes를 사용한 컨테이너 오케스트레이션.
4. 리소스 스케일링:
클라우드 기반 인프라에서 필요에 따라 컴퓨팅 리소스를 동적으로 추가하거나 축소.
예: AWS Auto Scaling, GCP Autoscaler.
5. 모델 관리 스케일링:
여러 모델과 버전을 동시에 관리.
예: SageMaker Model Registry, MLflow.

Scaling의 장점
대규모 데이터 처리: 데이터 크기가 증가해도 처리 시간이 크게 증가하지 않음.
사용자 증가 대응: 서비스 사용량이 증가해도 안정적으로 요청을 처리.
유연성: 필요에 따라 리소스를 동적으로 할당.
비용 효율성: 클라우드 기반의 확장을 통해 필요한 만큼만 리소스를 사용.

Automation and scaling



Design phase



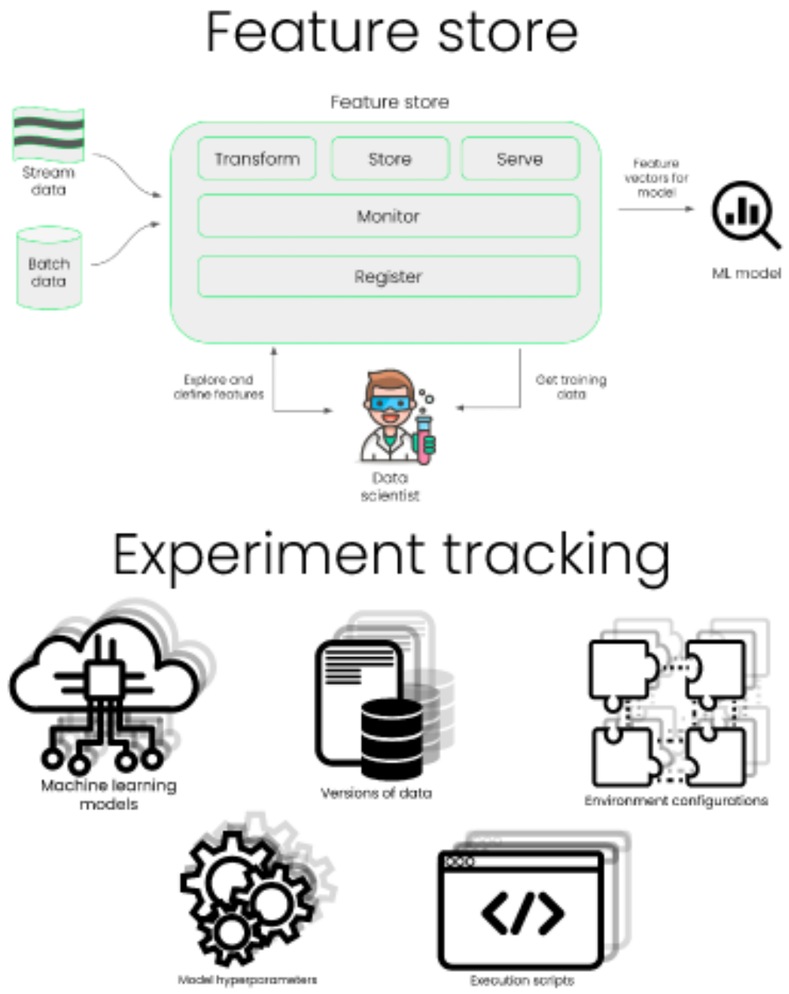
Project design

- Project design remains a manual process
- Use templates to automate and scale

Data acquisition

- Can be automated
- Enables high data quality

Development phase



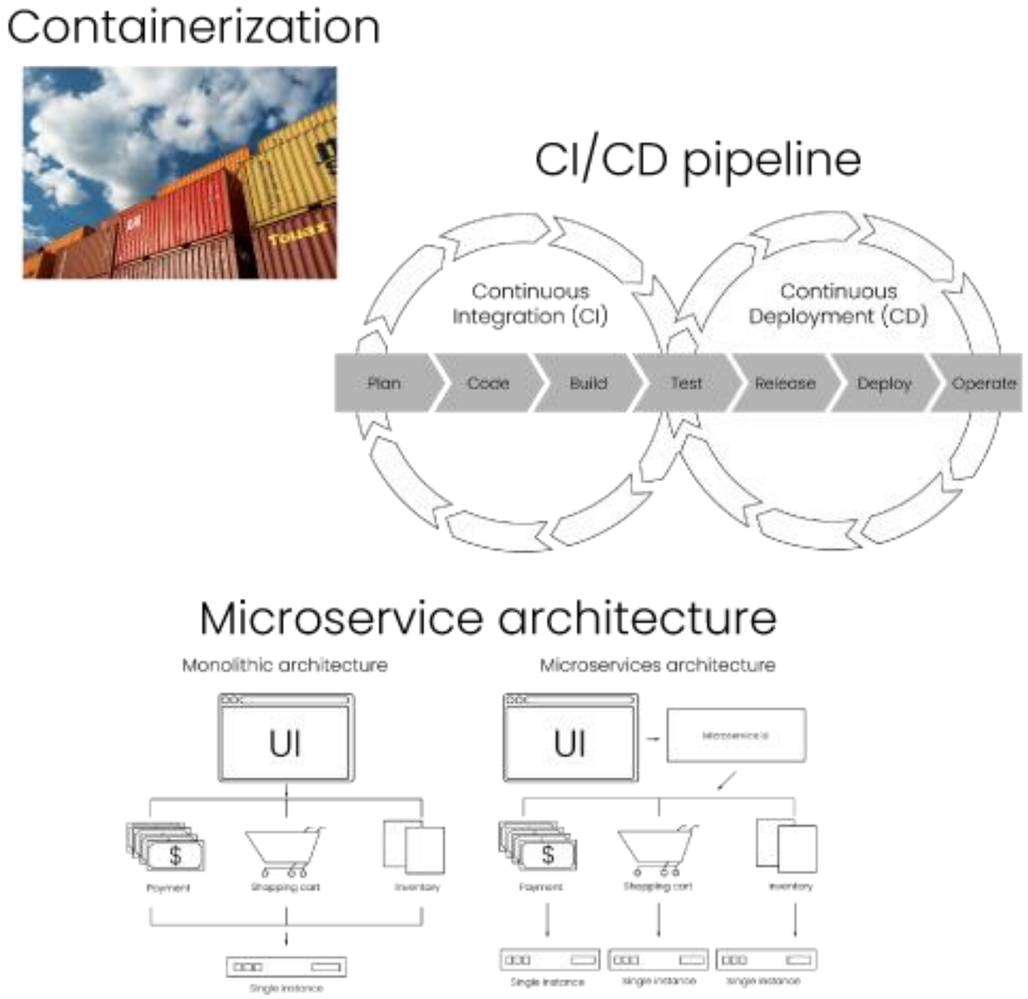
Feature store

- Saves time building the same features
- Helps to scale

Experiment tracking

- Automates tracking
- Ensures reproducibility

Deployment phase



Containerization

- Easy to start up copies of the same application
- Improves scalability

CI/CD pipeline

- Automates development and deployment
- Increases velocity of processes

Microservices architecture

- Improves scalability
- Independent development and deployment