DevOps

1. 정의
   A. Development + Operation (개발 + 운영)
   B. 그렇다면, 개발과 운영을 같이 하는 것을 말하는가? → DevOpps!
   C. 방법론 뿐 아니라 어떠한 문화를 이야기 하는 것으로
      i. 소프트웨어 제품을 효율적으로 개발, 배포, 운영하는 방법론 및 문화
      ii. 애플리케이션 개발과 배포 주기를 빠르고 안정적으로 만드는 것을 목표
2. 관련 개념
   A. 협업
      - 개발자가 업무를 위해서 협업한다는 것의 의미란 무엇일까?
   B. 소프트웨어 개발 수명 주기

   C. Agile 개발 방법론
      i. 개발 방법론이란? (미국 CMS 홈페이지에서 발췌)
         - 소프트웨어 개발 프로세스는 소프트웨어 개발을 계획하고 관리하는 프로세스
         - 개발 업무를 조금 더 작고, 병렬적 혹은 순차적인 단계 혹은 하부 프로세스로 나누어 소프트웨어 설계와 제품 관리 수준을 향상시키기 위한 방법론
         - 구체적인 제품의 사전 정의 단계를 포함하며, 제품을 개발하거나 유지보수하기 위해서 프로젝트 팀이 생산하는 '모든 것'도 포함됨
      ii. 대표적인 소프트웨어 개발 방법론
         - 구조적 방법론
            ✓ 대체적으로 "요구사항 분석 → 구조 분석 → 구조 설계 → 구조를 구현"하는 순서로 이행
            ✓ DFD, DD, STD, Spec.을 만들고, 이를 구현

- ✓ 장점 : 명확한 요구사항을 추출하고 설계에 반영
- ✓ 단점 : 데이터 모델링이 부족, 기업 단위의 통합 부족, 명확한 방법론의 지침이 없이 설계와 코딩에 의존하는 바가 큼

- 정보공학 방법론
  - ✓ 비즈니스 관점으로 정보 전략을 수립하고, 이를 바탕으로 진행하는 개발 방법론
  - ✓ 전략에 따라 요구사항을 분석하고, 분석 결과는 데이터 모델링, 프로세스 모델링, 그리고 데이터와 프로세스 간의 상관 관계를 정의하는 식으로 설계에 반영
  - ✓ 정보화 전략 수립 → 업무 영역 분석 → 업무 시스템 설계 → 시스템 구축
  - ✓ ERD / BSD / Presentation Layer의 개념이 등장
  - ✓ 장점 : 전략에 따른 기회 식별 및 구현 방안 제공, 중/장기적인 시스템의 설계 및 구축이 가능, 일관성 있고 통일된 정보 시스템의 구축, 데이터를 중심으로 한 유연성
  - ✓ 단점 : 시간과 인력이 많이 들어가며, 디테일한 단위 정보로 가면 길을 잃게 됨. 독자적인 비즈니스 추진 시에 어려움
- 객체지향 방법론
  - ✓ 현실 세계의 존재하는 대상을 속성과 메소드가 결합된 객체로 표현하고, 이 표현이 실체화된 대상을 객체(Object)라는 형태로 실체화
  - ✓ 분석, 설계, 구현을 객체지향화하여 수행
  - ✓ 캡슐화(Encapsulation), 추상화(Abstraction), 다형성(Polymorphism), 상속성(Inheritance) 등 객체를 기반으로 현실 세계를 모델링하기 위한 개념이 도입
  - ✓ 장점 : 모형의 적합성, 재사용성 증가, 빠른 개발, 설계의 적정성, 모듈화와 유연성, 코드 가독성 증가, 유지보수가 쉬워짐
  - ✓ 단점 : 과거의 개발 방법론과의 이질성(?), 구현된 시스템 상에서의 성능 문제(?), 과한 설계
- 컴포넌트 기반 방법론
  - ✓ 소프트웨어를 재사용 가능한 독립적인 구성요소(컴포넌트)로 구성
  - ✓ 이러한 컴포넌트를 조합하여 애플리케이션을 구축
  - ✓ 컴포넌트 개발 단계와 컴포넌트 기반 개발 단계가 분리가 되며, 컴포넌트 기반 개발 단계에서는 기존의 컴포넌트를 선택해 사용하지만, 필요에 따라 컴포넌트를 개발하는 하위 프로세스를 포함하기도 함
  - ✓ 컴포넌트 간 상호작용을 위한 인터페이스를 개발하며, 이 인터페이스가 컴포넌트 재사용, 모듈화의 중요한 요소
  - ✓ 컴포넌트의 표준화가 핵심이며, 이를 위한 다양한 인증도 존재
  - ✓ 장점 : 재사용성, 모듈화, 유연성이 객체지향 개발 방법론보다 뛰어나며, 이 덕분이 개발 시간, 비용이 절감되고, 유지 보수 용이성이나 분산 또는 병렬 개발이 가능해짐

- ✓ 단점 : 컴포넌트의 스파게티, 일관성 및 문서의 중요성, 컴포넌트의 적정성을 담보하지 못하는 경우 개발 일정이 쉽게 엉망이 됨

iii. 애자일 방법론

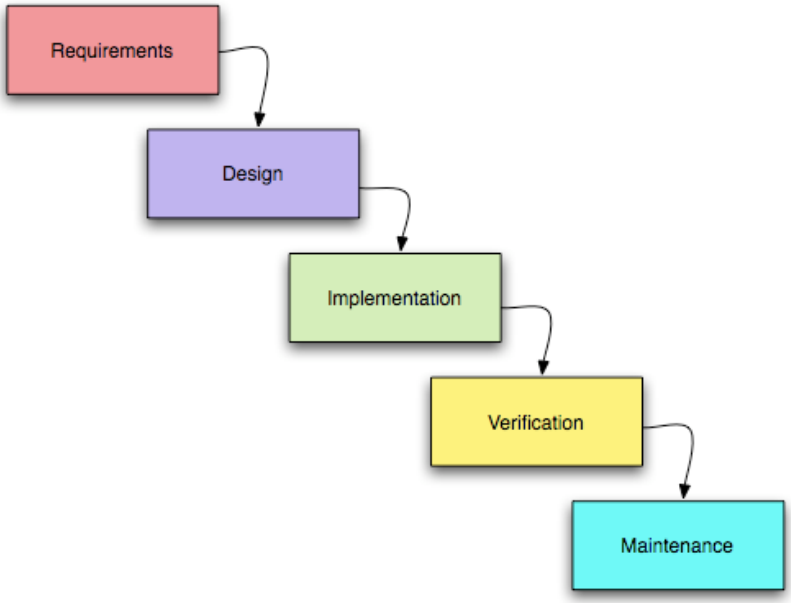
- 정의
  - ✓ 소프트웨어 개발 프로세스를 유연하고 적응적으로 진행하는 접근 방식
  - ✓ 변화에 대응하고, 고객의 요구 사항을 최우선 적으로 반영
  - ✓ 작은 주기의 반복적인 개발을 통해 빠른 결과물을 제공하는 것을 목표로 함
- 특징
  - ✓ 변화에 대한 대응 : 요구 사항 또는 환경의 변화에 빠르게 대응하기 위해 작은 주기로 개발을 진행하고, 필요에 따라 변화를 설계 및 구현에 반영
  - ✓ 고객 중심 개발 : 고객의 요구 사항을 중요 판단 기준으로 삼고, 고객과 지속적으로 소통함. 작동 가능한 소프트웨어를 가급적 자주 고객에게 전달하며, 고객으로 부터 피드백도 가능한한 자주 받아서 더 나은 제품을 제공
  - ✓ 자기조직화된 팀 : 자기조직화되어 개발 작업을 계획하고 실행함. 자기조직화된 팀은 역할과 책임이 명확하고, 팀원 간의 협력과 의사 소통이 강조됨
- 실천 방식
  - ✓ 스크럼 : 일정한 기간 (가능한한 짧게. 보통 2주) 주기로 개발을 진행하며, 구현되지 않은 사항은 백로그를 작성해서 관리하고, 백로그의 우선 순위에 따라서 스프린트라고 불리는 작업 단위를 설정해, 스프린트 동안 정해진 범위를 개발하며, 그 결과를 매 스프린트마다 회고를 해서 개선
  - ✓ 익스트림 프로그래밍 (XP) : 간단한 설계로 수행할 수 있는 짧은 주기로 반복 개발을 진행하며, 이를 위해서 개발 이전에 테스트를 먼저 작성하고 이에 맞게 코드를 개발하는 테스트 주도 개발(TDD)를 실천. 지속적인 통합은 코드 통합을 가능한한 이른 시점에 진행하여 충돌과 문제를 조기에 발견하고 해결
- 장점
  - ✓ 빠른 결과물 제공 : 작은 주기로 개발을 진행, 고객의 피드백을 바로 반영할 수 있음
  - ✓ 고객 만족도 향상 : 지속적 소통을 전제로 고객 중심 개발을 진행하므로 요구사항을 정확하게 파악하거나 명확하게 가이드하고, 이를 충족시킬 수 있음
  - ✓ 유연성과 적응성 : 변화에 바르게 대응할 수 있고, 필요에 따라 요구 사항을 수정하거나 추가할 수 있음
- 단점

✓ 상세한 계획의 부족 – 기술 부채의 폭주 가능성

✓ 팀원의 역량이 고도화되어야 함

iv. Waterfall vs. Scrum

- Waterfall 방식



	Waterfall 방법론	Scrum 방법론
개요	선형적 개발 프로세스	반복적, 증분적 개발 프로세스
개발 순서	요구 사항 분석 → 설계 → 개발 → 테스트 → 운영/유지보수	요구 사항 정의 → 스프린트 계획 → 개발 → 테스트 → 스프린트 검토 → 스프린트 회고
변경 관리	변경이 어려움	변화에 유연하게 대응 가능
팀 구성	역할과 책임이 명확하게 정의됨	자기조직화된 팀 구성
고객 관여	초기 요구 사항 정의 후 개발 진행	지속적인 고객 참여와 피드백
작업 간격	단계적이고 순차적인 진행	반복적이고 반복 주기를 갖는 진행
결과물 제공	전체 시스템이 완료될 때까지 결과물 제공	각 스프린트마다 작은 결과물 제공
변경 및 오류	변경이 어려워 오류 발생할 경우 수정이 어려움	변경에 대한 유연한 대응과 조기 오류 발견 및 수정
프로젝트 관리	계획 중심으로 프로젝트 관리	진척 상황, 기술부채와 장애물을 중심으로 한 지속적인 관리
유지보수성	초기 계획이 변경되지 않아서 유지보수에 좋음	유연한 변경 대응으로 유지보수에 좋음

예측 가능성	상세한 계획에 따른 예측성이 좋음	자주 바뀔 수 있으므로 초기에는 예측이 불가

D. 마이크로서비스 아키텍쳐

i. 정의

- 애플리케이션을 작고 독립적인 서비스들로 분리하는 소프트웨어 아키텍쳐 패턴
- 전통적 단일 (모노리스) 구조와는 달리 작은 규모의 서비스로 구성되며
- 각 서비스는 자체적으로 배포되고 운영

ii. 특징

- 서비스 분리
- 독립적 배포 – A를 수정, 테스트, 배포하더라도 B, C, D… 등은 영향을 받지 않음
- 서비스별로 최적의 특정 기술 스택을 사용
- 서비스 간 커뮤니케이션을 통한 통합 – RESTful API, 메시지 큐, 이벤트 저장소 통신 등 활용
- 자율적 팀 구성

iii. 장점

- 확장성과 유연성
- 독립적인 개발과 배포
- 장애 격리와 복원 용이성

iv. 단점

- 운영 및 관리의 복잡성
- 시스템 복잡성 증가
- 테스트와 디버깅이 복잡해짐
  - ✓ 특히 상호작용하는 서비스 간은 쌍으로 테스트가 필요

E. 지속적 통합 (Continuous Integration / CI)

i. 정의

- 개발자들이 작업한 코드 변경 사항을 빈번하게 통합하는 개발 방식

ii. 필요성

- 개발 완료 후 품질 관리를 하면 생기는 일들에 대한 회고
- 개발 완료 후 하는 통합은 지옥과 같은 무엇인가.

iii. 중요성

- 충돌 예방 : 지속적 통합은 개발자들이 작업한 코드를 빠르게 통합함으로써 충돌을 사전에 감지하고 예방
- 품질 유지 : 지속적인 통합과 자동화된 테스트를 통해 코드 품질을 유지
- 빠른 피드백 : 버그를 조기에 발견, 수정할 수 있도록 도와줌
- 협업 강화

iv. 특징

- 자동화 : 통합 및 빌드 과정을 자동화하여 개발자들이 수동으로 작업할

필요성을 해소

- 주기적인 통합 : 정기적으로 코드 변경 사항을 통합하며, 일반적으로 개발자들의 작업 주기에 맞춰 통합 작업을 수행
- 테스트와 결합된 프로세스 : 자동화된 테스트를 통해 코드 변경 사항의 품질을 확인하고 문제점을 조기에 발견

v. 실천 방법

- 원칙의 수립과 합의
  - ✓ 협업에 참여하는 사람은 자신의 작업을 제출하기 전에 반드시 완벽한 빌드와 테스트를 수행
  - ✓ 모든 프로그래머는 모든 작업을 표준 코드와 일치시키며 시작
  - ✓ 모든 프로그래머의 모든 작업을 표준 코드에 결과물을 제출하는데서 마무리
- 공유 저장소
- 빌드 자동화
- 테스트 자동화

vi. 장점

- 충돌 예방과 버그 조기 발견
- 코드 품질 유지
- 개발자 간 협업 강화
- 빠른 피드백 제공
- 소프트웨어 전달 속도 향상

vii. 단점

- 구현 및 유지 관리 비용 증가
- 초기 설정과 관리에 대한 추가 작업이 필요
- 테스트 커버리지에 비용이 들어가며, 테스트 커버리지가 부족하면 오히려 하느니만 못한 작업

F. 지속적 전달 (Continuous Delivery / CD)

i. 정의 : 자동화된 프로세스를 통해 빌드, 테스트, 배포하는 개발 방법론으로, 품질 보증, 피드백 반영, 자동화 등의 철학을 따르며, 고객 요구나 환경 변화에 맞춰 언제든지 소프트웨어를 출시할 수 있는 가능성을 제공

ii. 중요성

- (자동화된 프로세스를 통한) 빠른 출시
- (지속적인 자동화된 테스트를 통한) 품질 유지
- (빠른 배포와 피드백으로 고객의 요구에 대한 빠른 대응으로) 고객 만족도 향상

iii. 특징

- 빌드, 테스트, 배포 등의 자동화를 통한 효율성의 극대화
- 지속적 배포 : 신기능 또는 패치 등이 준비되면 자동으로 배포되어 고객에게 제공

iv. 실천 방법

- 지속적 통합과 결합 (CI/CD)
- 배포 자동화 : 배포 프로세스를 자동화하여 신속하고 안정적으로 소프트웨어를 배포
- 환경 일치성 : 개발, 테스트, 운영 환경을 일치시켜 문제를 사전에 발견하고 예방

v. 장점
- 빠른 출시와 고객 만족도 향상
- 자동화된 테스트를 통한 품질 유지
- 안정적인 배포와 작업 효율성 향상

vi. 단점
- 초기 설정과 관리에 대한 추가 작업 필요
- 비용 및 노력
- 정형화된 정답이 없는 과정

G. 새로운 개발 환경을 위한 담보 (Agile, MSA, CI, CD)

i. 개발과 운영의 협력 강화
- 개발자와 운영자의 사일로는 개발한 소프트웨어를 운영 환경으로 원활하게 전달하는데 어려움이 있음

ii. 빠른 소프트웨어 전달
- 고객의 눈높이와 변화를 따라잡기 위한 최소한의 기본 요건

iii. 자동화와 표준화
- 반복적이고 지루한 작업을 배제
- 인력과 시간을 절약하고 오류를 줄일 수 있음
- 과정의 일관성을 유지하여 최대한 예측 가능한 결과를 도출

iv. 정량적 가치 측정
- 고객에게 전달할 가치 또는

v. 모니터링 (업무와 제품 모두)
- 고객에게 전달할 가치를 측정해야 하며
- 제품의 품질, 개발 과정의 품질을 측정해야 하며
- 이를 통합적으로 관리해 배포까지 연계 가능한 일원화된 모니터링 환경이 필요

vi. 안정성과 신뢰성
- 결과물은 '동작 가능한 소프트웨어'로부터 시작해 고객 가치를 증가시킬 수 있는 수준으로 발전해야 하며, 이를 위해 필요한 덕목
- 문제 발생 시에 빠르게 대응할 수 있어야 함

vii. 문화적인 변화

3. DevOps (Revised)

A. 필요성
- 빠른 소프트웨어 전달
- 안정적인 운영 환경
- 문제 신속 대응

- 개선된 품질

→ '적어도' 앞에서 소개한 최신 개발 방법론들이 유용하고 도입을 해야 한다고 결정하였다면, DevOps는 필수적인 선택

B. 특징

i. 문화적인 변화
- 개발자와 운영팀 간의 협업, 공유, 상호 이해를 강조하는 문화를 형성
- 개발자의 도구를 운영자가, 운영자의 도구를 개발자가 필요에 따라 활용하거나 구축할 수 있어야 함

ii. 자동화
- 반복적이고 일정한 작업이 많아지게 되며
- 이를 자동화하여 효율성을 높이고 오류를 줄이며
- 궁극적으로 내가 DevOps를 위해 별도의 수고를 하지 않고 있다고 느낄 수 있어야 함

iii. 지속적 통합 및 지속적 배포
- 개발된 소프트웨어를 지속적으로 통합하고 배포하여
- 릴리즈 주기를 단축하고 문제를 조기에 발견

iv. 모니터링 및 로깅
- 운영 환경의 상태를 모니터링하고 로그를 수집하여 시스템 동작과 문제를 추적

C. Tools

i. 버전 관리 도구
- 코드 변경 사항을 추적하고 관리하여 협업과 코드의 버전을 관리하는데 도움을 줌
- Git, SVN

ii. 지속적 통합 도구
- 개발된 코드를 자동으로 빌드하고 테스트하여 지속적으로 통합하는 프로세스를 자동화
- Jenkins, CircleCI, Travis CI

iii. 지속적 배포 / 배포 자동화 도구
- 소프트웨어를 자동으로 테스트, 빌드하고 배포하는 프로세스를 자동화
- Jenkins, CircleCI, GitLab CI/CD

iv. 환경 구성 자동화 도구
- 환경 구성을 코드로 관리
- 환경 프로비저닝, 구성, 관리를 자동화
- Terraform, AWS CloudFormation, Ansible, Chef, Puppet

v. 컨테이너 관리 도구
- 환경의 확장성과 유연성을 제공하기 위해 컨테이너화된 애플리케이션을 관리하고 배포하는데 사용
- Docker, Kubernetes, Docker Compose

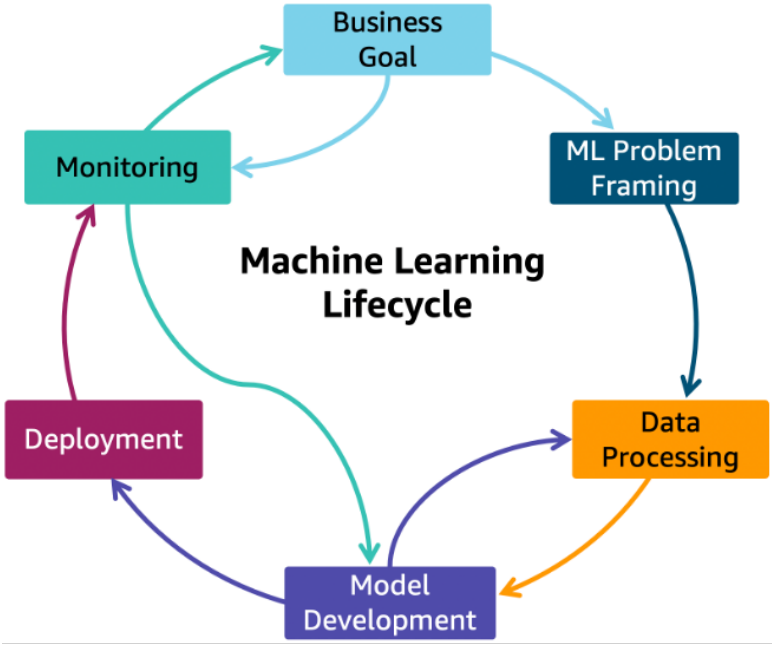
vi. 모니터링 및 로깅 도구
- 시스템, 앱, 인프라 등의 모니터링과 로그 분석을 통해 운영 환경을 모니터링하고 문제를 식별

- Grafana, Prometheus, Elasticsearch + Kibana

vii. 협업 도구
- 협업과 의사소통을 강화하기 위해 사용
- 이슈 관리, 문서 공유, 실시간 소통 등에 사용
- Jira, Confluence, Slack

viii. 테스트 자동화 도구
- 테스트를 자동화하여 앱의 품질을 검증하고 오류를 발견
- Selenium, JUnit, NUnit, PyTest

4. AI 환경에서의 DevOps
   A. ML + DevOps = MLOps
   B. 필요성
      - 머신러닝 역시 머신러닝 제품의 수명 주기를 가지고 있으며, 이는 전체적인 개발 수명 주기의 하위 싸이클로 관리가 되어야 함



      - 제품으로서 갖춰야 할 각각의 단계에 더불어 '최적화를 위한 과다한 반복', '시간의 흐름에 따른 데이터의 변호' 등으로 인해 ML 자체만의 단계의 자동화와 공유도 매우 중요한 이슈

   C. MLOps의 Requirement
      - 머신러닝 전 수명 주기를 관리할 수 있어야 함
      - 머신러닝에 필요한 데이터의 정제
      - 머신러닝 모형의 구축
      - 모형이 배포될 환경 전체를 관리

   D. Kubeflow (온프레미스)

      i. 쿠브플로우 노트북
         - 주피터 노트북을 제공하여 개발 및 결과 공유 등을 지원

      ii. 학습 오퍼레이터
         - 쿠버네티스 클러스터 상의 리소스를 적절히 배분, 관리하여 어플리케이션

의 성능을 관리

- 분산 학습을 지원하여 학습 성능을 향상

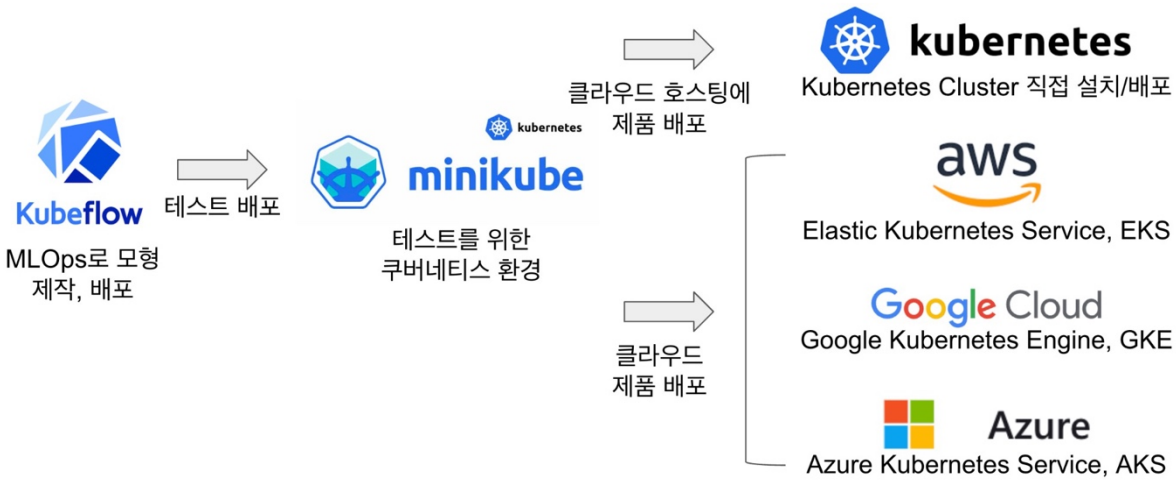
iii. Katib

- 모형 하이퍼파라미터 최적화를 위한 반복 작업을 자동화하여 최적의 학습률, 레이어 구조, 노드 수 등에 대한 실험을 자동화

iv. 쿠브플로우 파이프라인

- 학습에서 모형 배포까지의 전체 과정을 관리해주는 플랫폼
- 모형을 만들고 배포하는 과정의 각 구성요소를 UI를 사용해 오케스트레이션

E. Cloud 환경에서의 MLOps

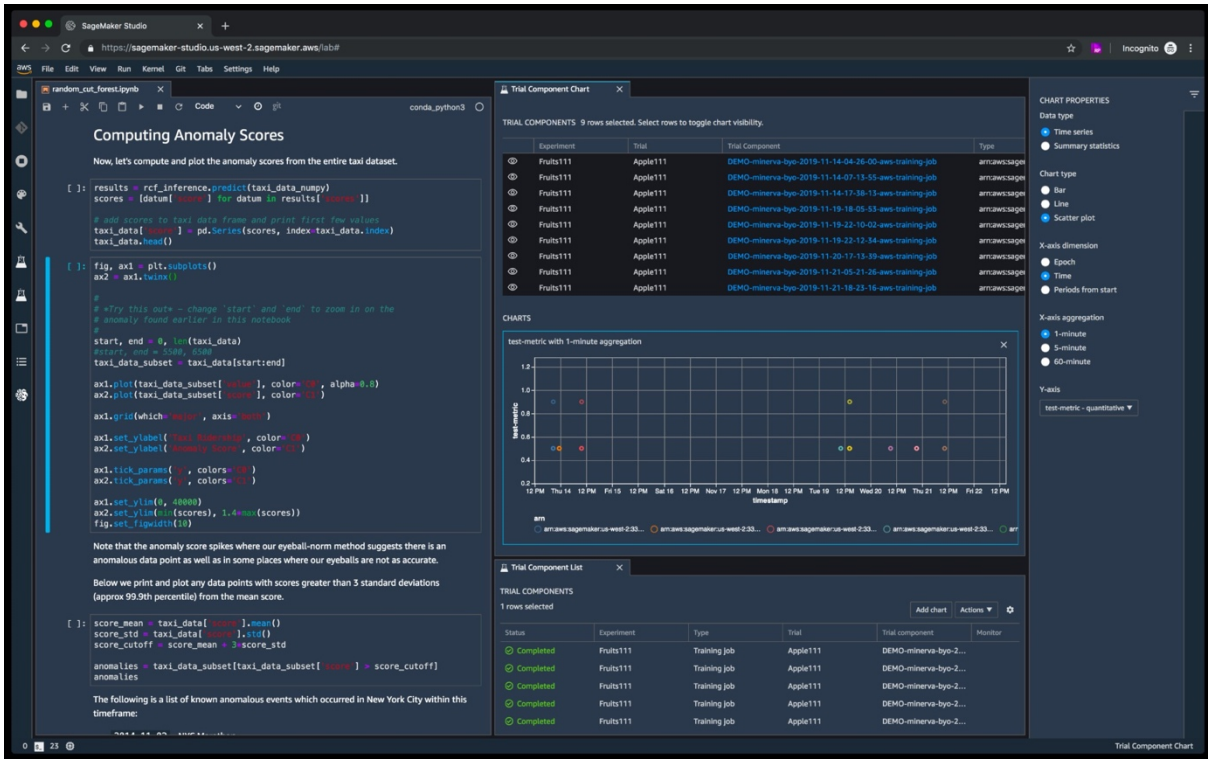


i. 온프레미스와 차별화된 요구사항

- 스팟 인스턴스 또는 서버리스 학습 기능
- 학습 자동화
- 워크플로우 파이프라인과 엔드포인트 배포
- 사전 학습된 모형의 쉬운 활용 지원
- No Code / Low Code 학습

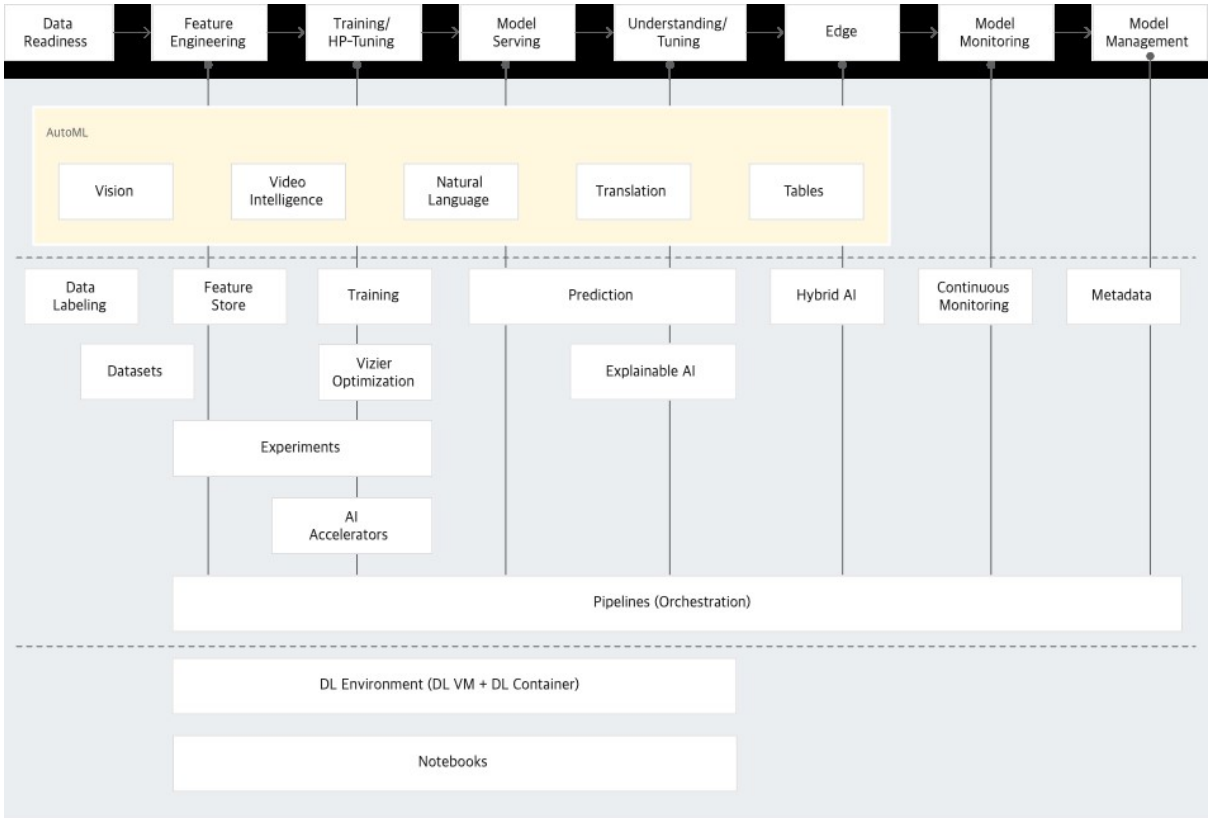
ii. Amazon SageMaker

- 완전 관리형 머신러닝 서비스로 머신러닝 모형을 빠르게 구축, 훈련, 최적화 배포할 수 있도록 지원
- 제공되는 콘솔 또는 스튜디오 도구에서 간략화된 방식으로 모형을 제작하거나, 저작 노트북을 인스턴스 형태로 제공하여 개발 서버 관리 없이 머신러닝 코드 작성 가능
- 만들어진 모형을 아마존 클라우드의 호스트 서버에 배포하는 기능을 제공하며, 이를 위한 모형 레지스트리, 모형 구축 파이프라인 및 프로젝트 제품군을 포함



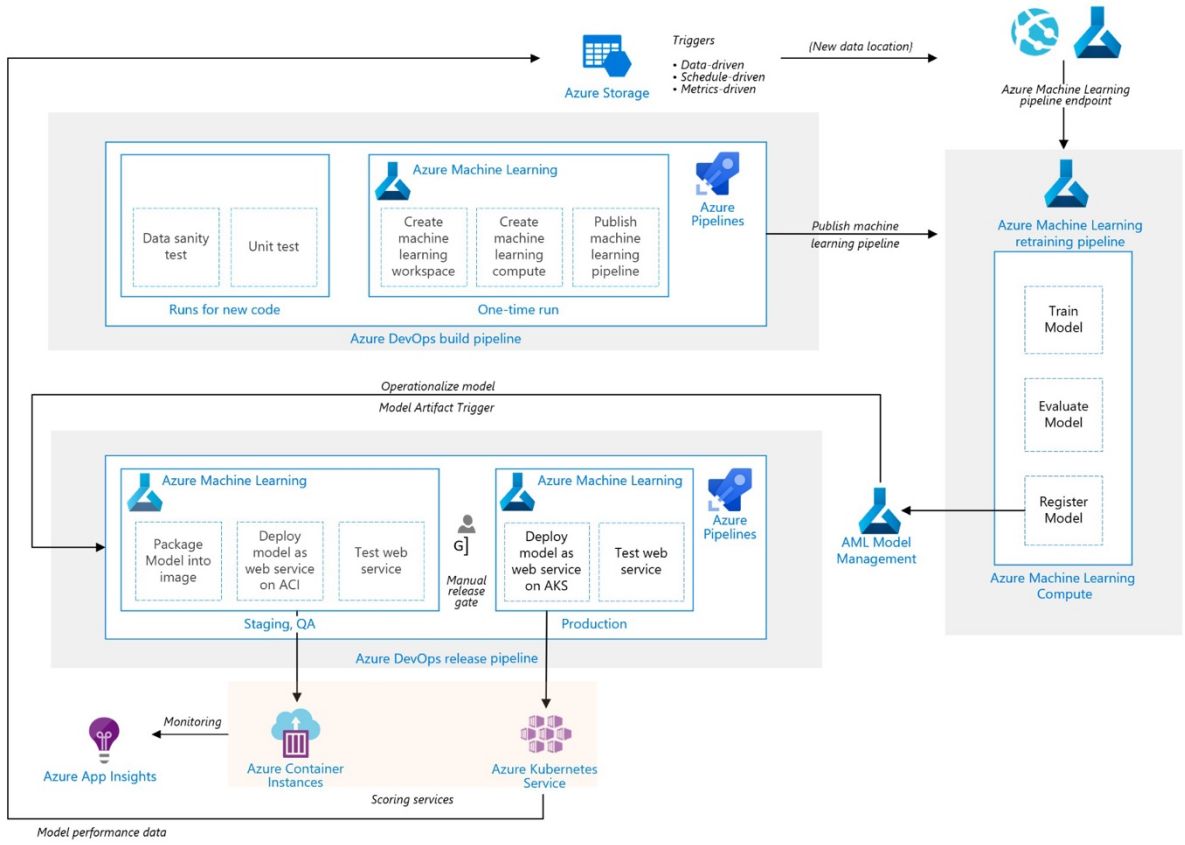
iii. 구글 Vertex AI

- 서버리스 방식으로 동작
- 구글이 제공하는 다양한 사전 학습된 모형을 사용하거나, 모형 생성 가능
- 최신 모형 학습에 사용되는 거의 모든 머신러닝 도구 사용 가능
- 강력한 AutoML 기능
- 제공되는 MLOps 도구를 활용, 하나의 UI 또는 API를 통해서 학습에서 배포까지의 전체 과정을 자동화할 수 있음
- 구글의 BigQuery ML 뿐 아니라 표준 SQL, Spark 등 데이터 처리를 사용되는 도구를 폭넓게 활용할 수 있는 데이터 통합 기능

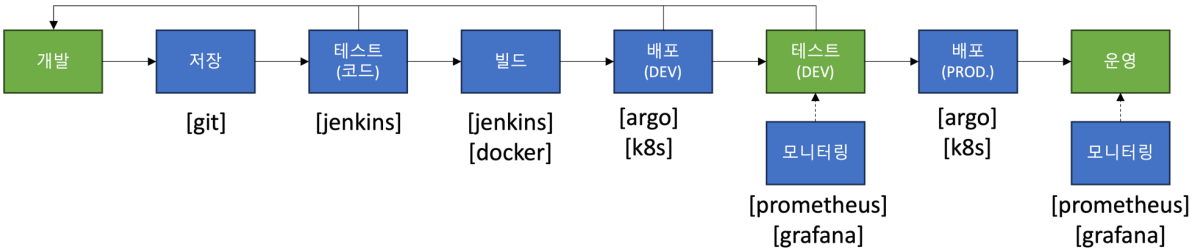


iv. Microsoft Azuer Pipeline

- 윈도우즈, 리눅스 및 맥OS 모두에서 사용할 수 있는 클라우드 기반 파이프라인을 제공
- 애플리케이션이나 데이터 파이프라인을 구축하고 배포하는 과정을 자동화
- 클라우드/온프레미드 양쪽 모두의 배포를 지원하며, Azure 이외의 다른 클라우드를 대상으로 배포도 가능
- Azure Machine Learning과 통합하여 Python과 머신러닝 라이브러리를 사용한 모형을 생성하고 제품 환경으로 배포하는 것이 가능



5. 일반적인 적용 예시



6. 이번 주차의 학습 내용
   A. 목표
      - 간단한 DevOps 환경 구축
   B. 최종적으로 실천해야 할 일
      - 간단한 머신러닝 앱을 구축하고 배포
      - 학습 데이터가 등록되면 자동으로 모형을 학습
      - 소스 코드를 수정하거나, 모형을 새로 학습하면 테스트를 자동으로 수행
      - 테스트가 성공하면 도커 이미지를 자동으로 빌드
      - 빌드가 성공하면 이를 자동으로 배포
      - 배포되어 동작하는 앱을 모니터링
   C. 학습 Coverage

- Git
- Docker/K8S
- Jenkins
- Prometheus + Grafana