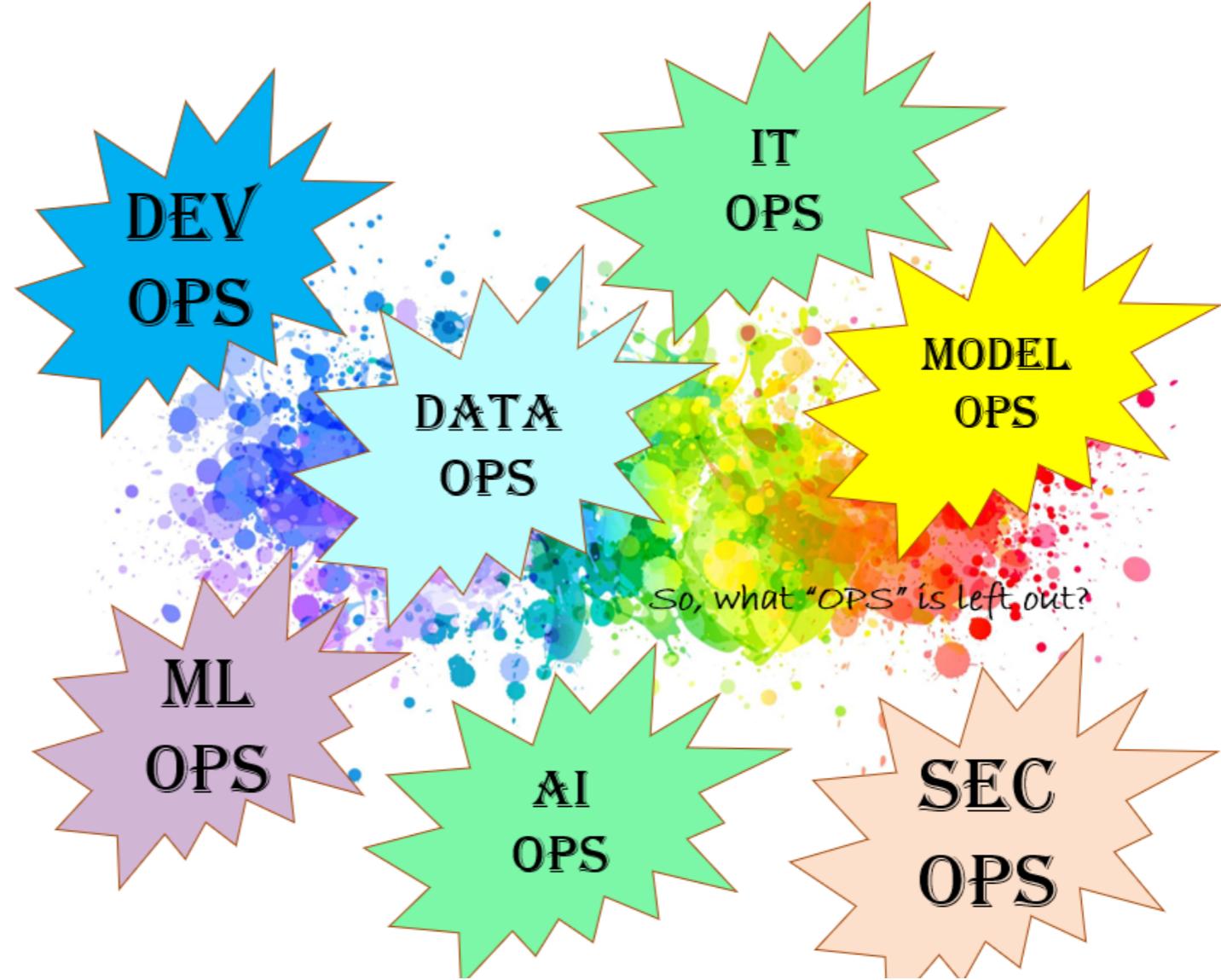


Because of  
you,

**AGILE**,

I Need to  
Learn --->

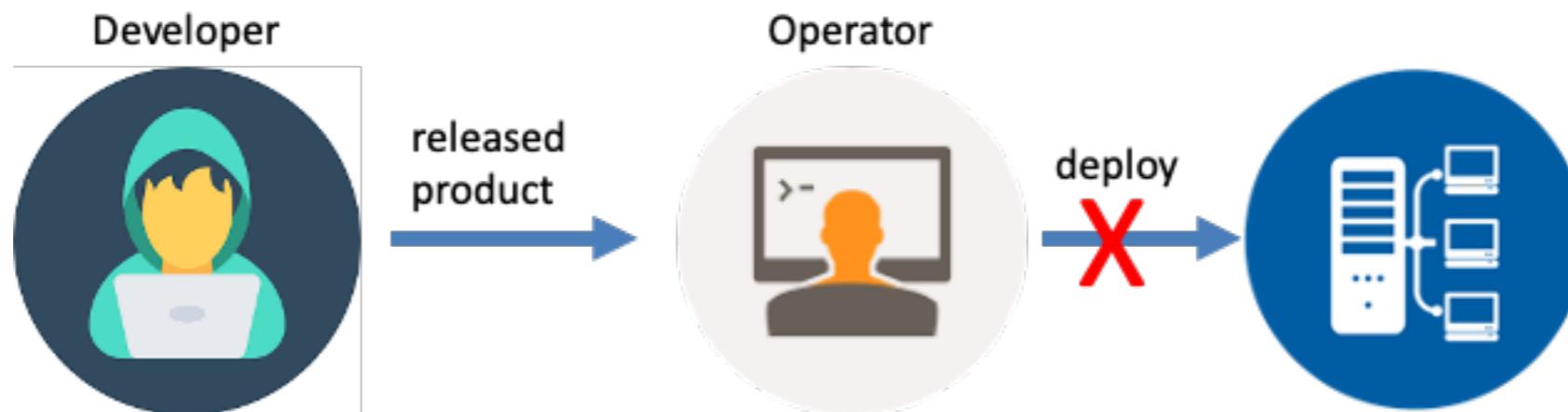
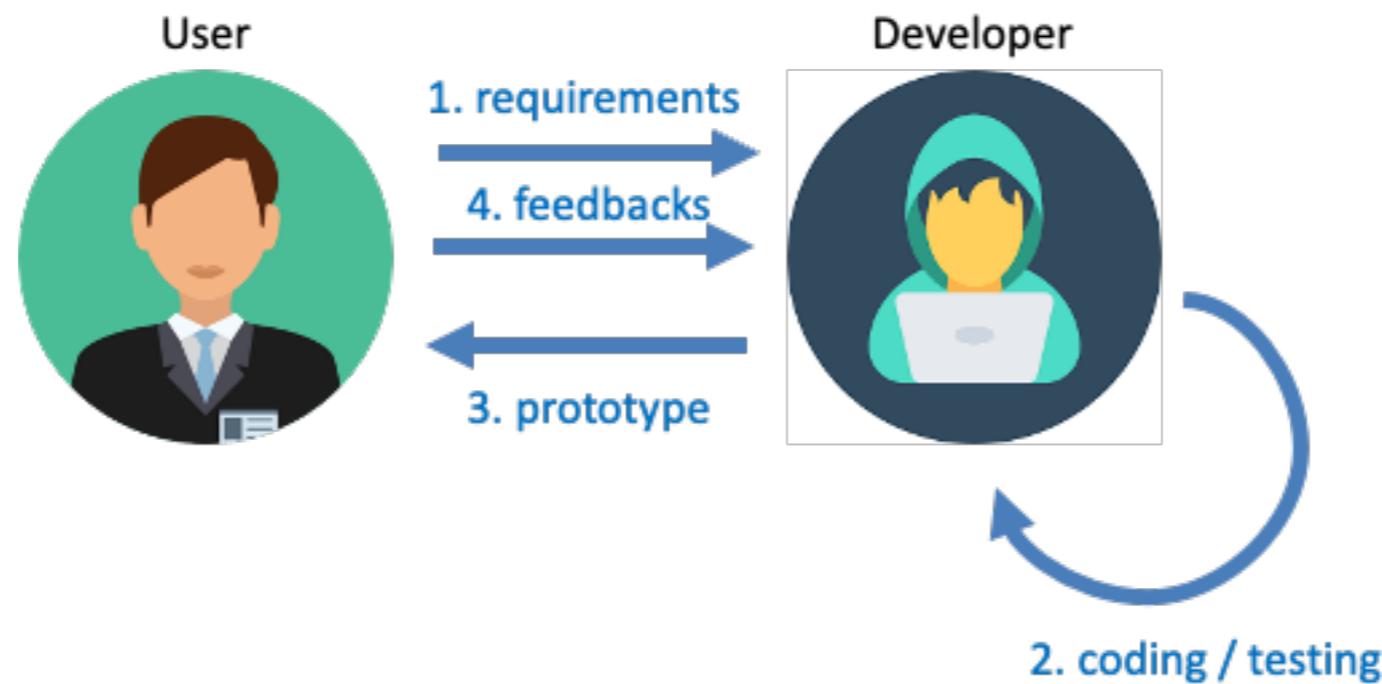


2110446 - Data Science and Data Engineering

## Ops Stars

Asst.Prof. Natawut Nupairoj, Ph.D.  
Department of Computer Engineering  
Chulalongkorn University  
[natawut.n@chula.ac.th](mailto:natawut.n@chula.ac.th)

# Agile Development and the Real World Problems



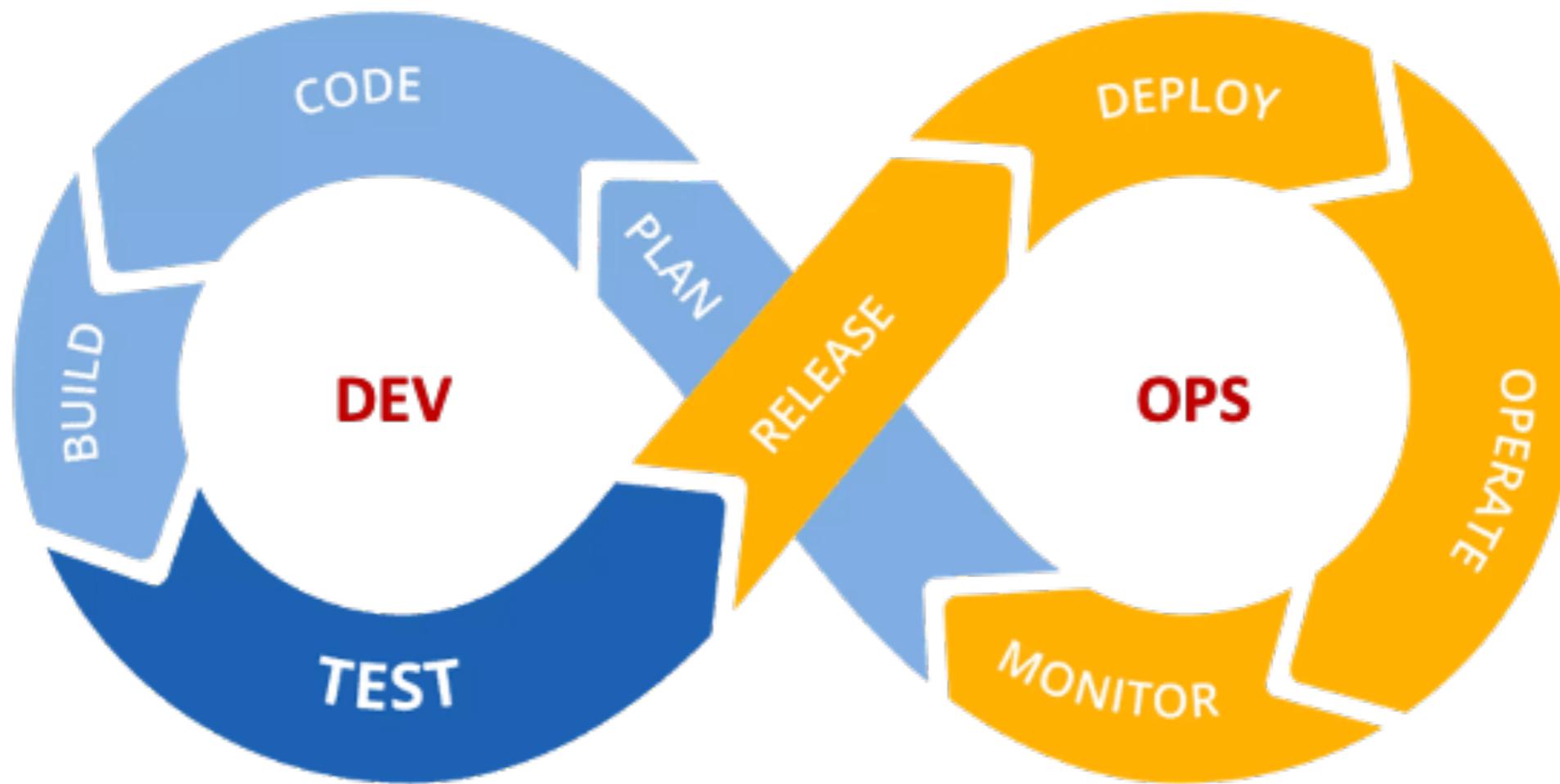
# Solution: DevOps

---

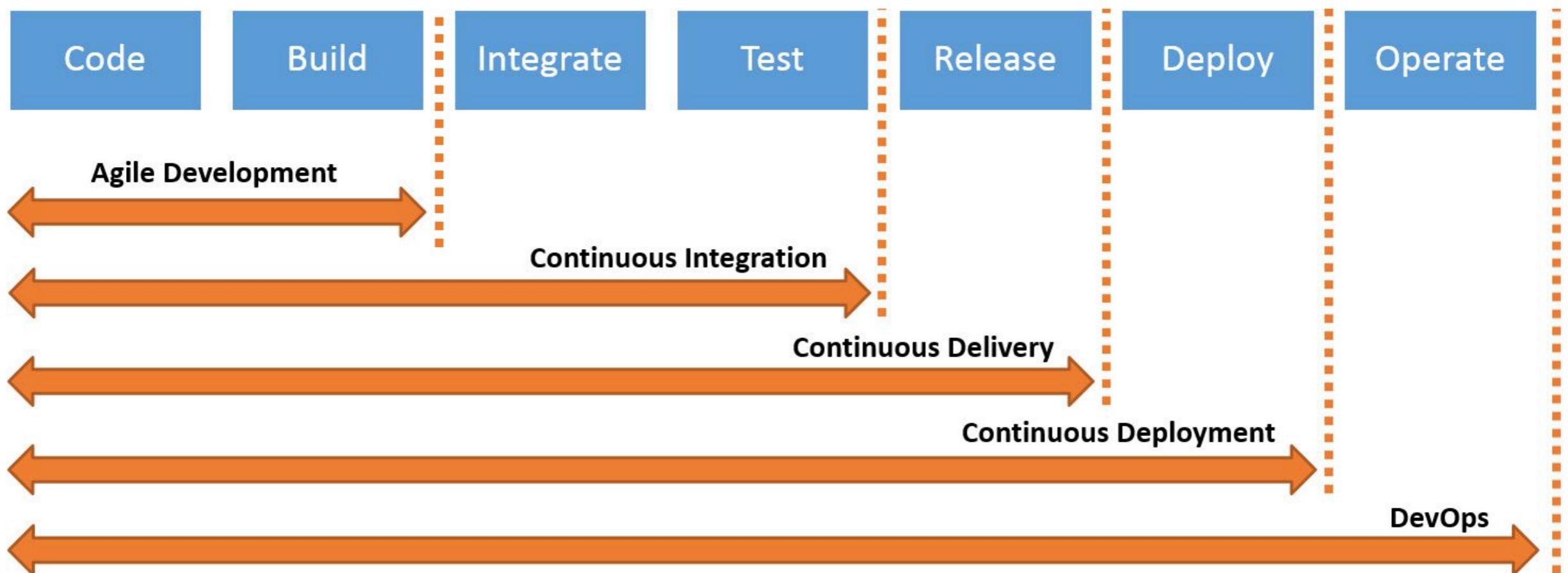
- A set of practices that combines software development (Dev) and IT operations (Ops)
- Aim to shorten SDLC with continuous delivery and high software quality
  - Tackle deployment architecture at early stages
  - Involve everyone in SDLC in a continuous fashion
  - Automate all necessary steps as much as and as early as possible
- Software-defined system (Infrastructure as Code) plays a very important role to ensure smooth integration and delivery

# The Infinite Lifecycle

---



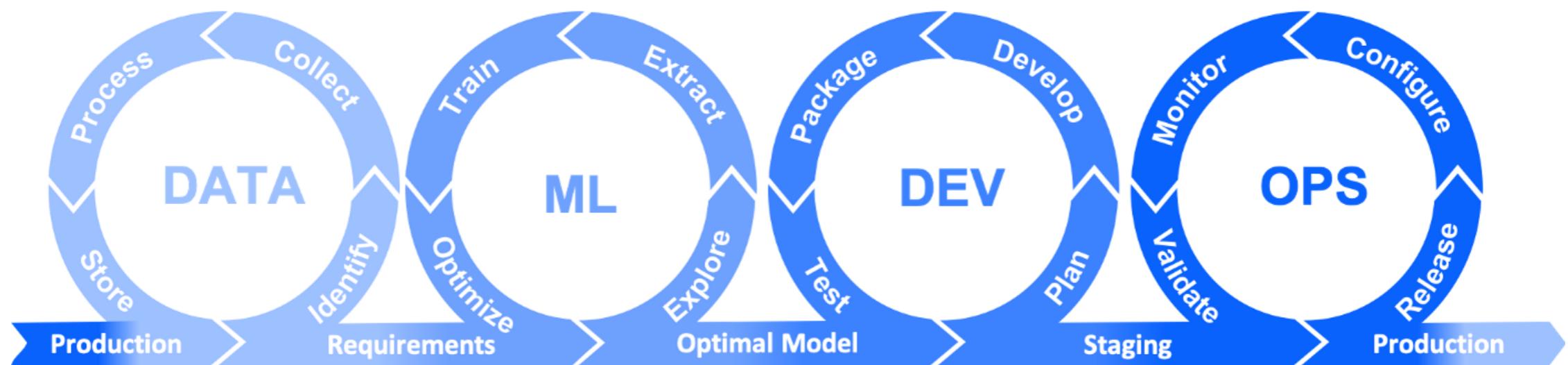
# DevOps Automation



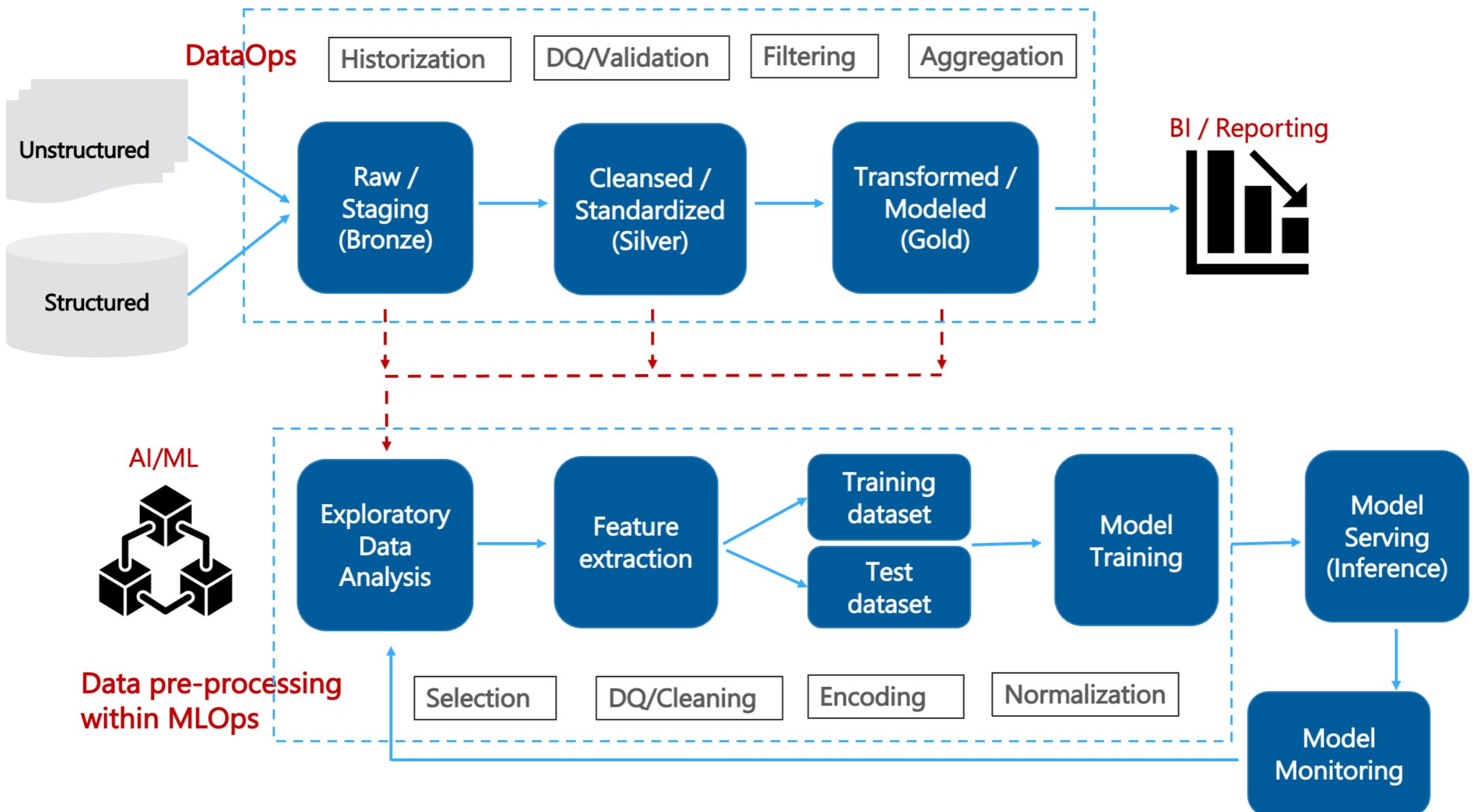
Source: <https://www.bmc.com/blogs/continuous-delivery-continuous-deployment-continuous-integration-whats-difference/>

# With ML and Data Science, these cycles become more complicated

---

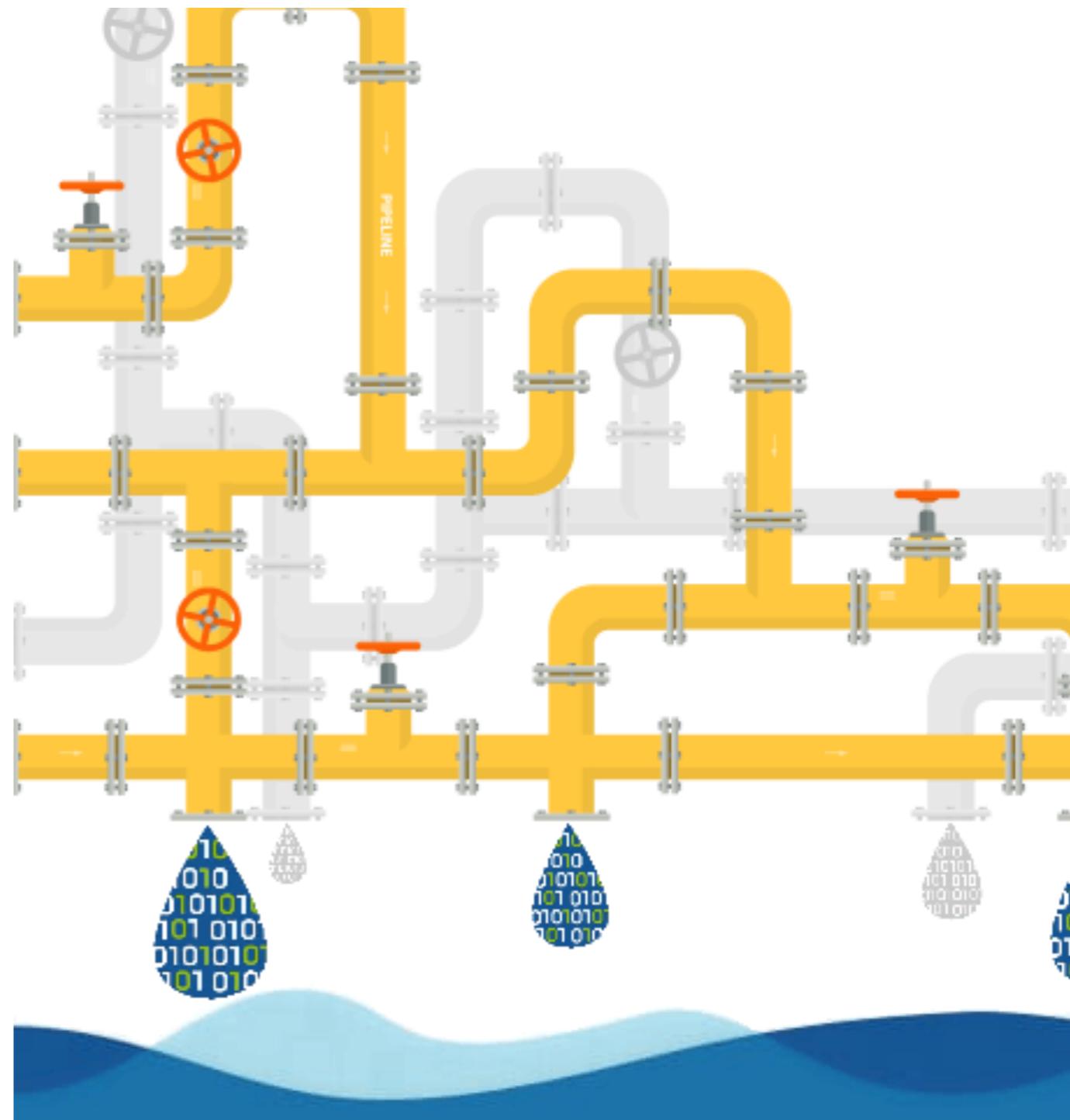


# DataOps and MLOps

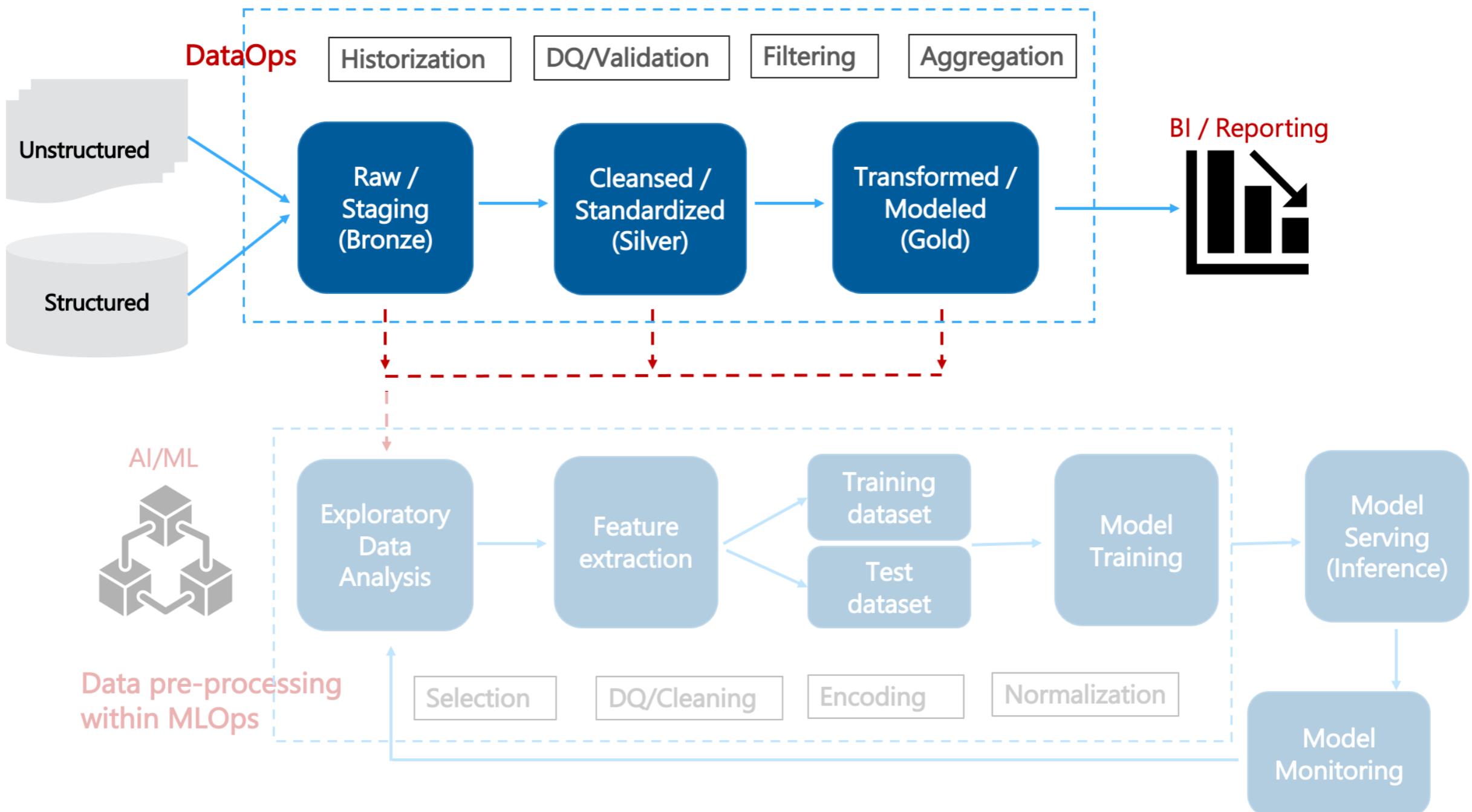


# Orchestrating Data Pipeline for DataOps

Ops Stars



# DataOps and MLOps

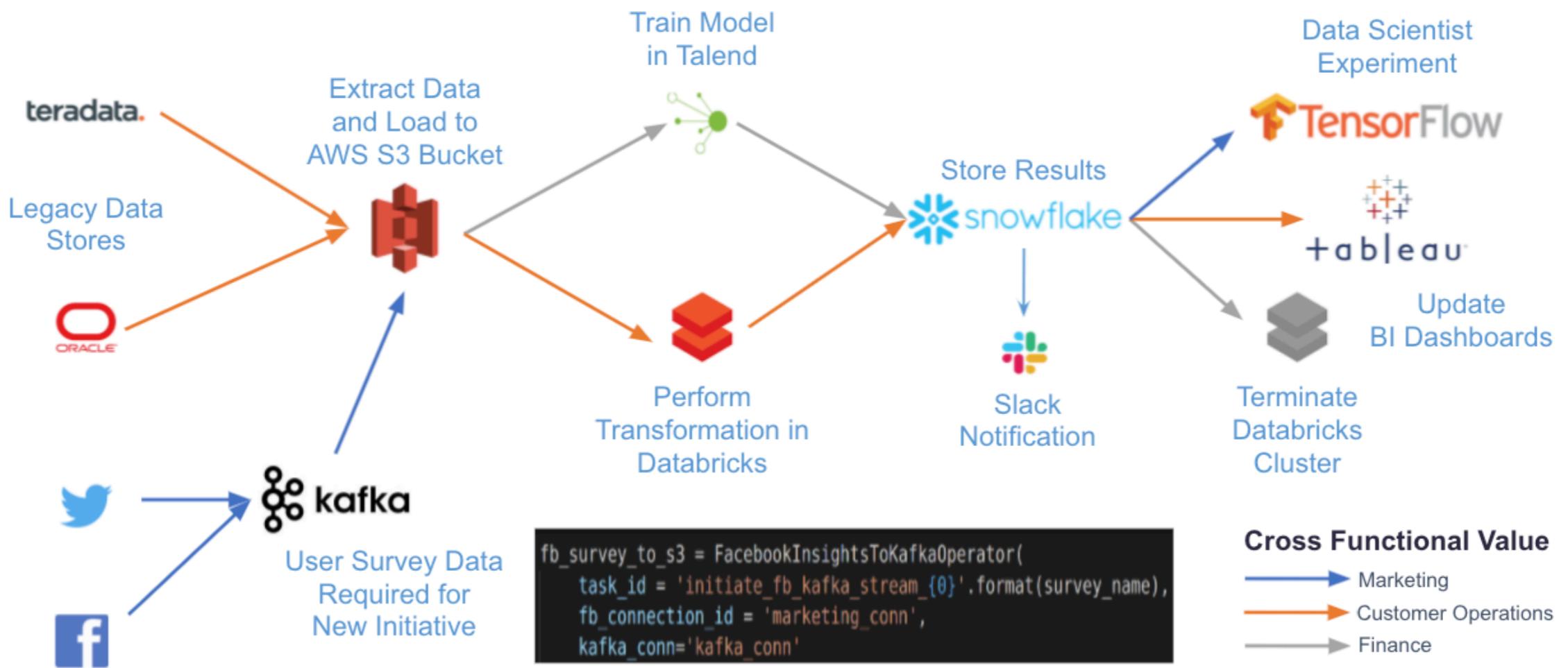


# Apache Airflow

---

- Open-source workflow management platform for data engineering pipelines, started by Airbnb in 2014
- Designed under the principle of "configuration as code"
  - Uses directed acyclic graphs (DAGs) to manage workflow orchestration
  - Tasks and dependencies are defined in Python (possible for just one file)
  - DAG can be started by defined calendar schedule or triggered by external event
  - Airflow manages the scheduling and execution
- Provide managing and monitoring facility with web-based UI

# Flexibility of Data Pipelines as Code



Complex pipeline can be implemented using operators, provider packages, and external services — with minimal code writing

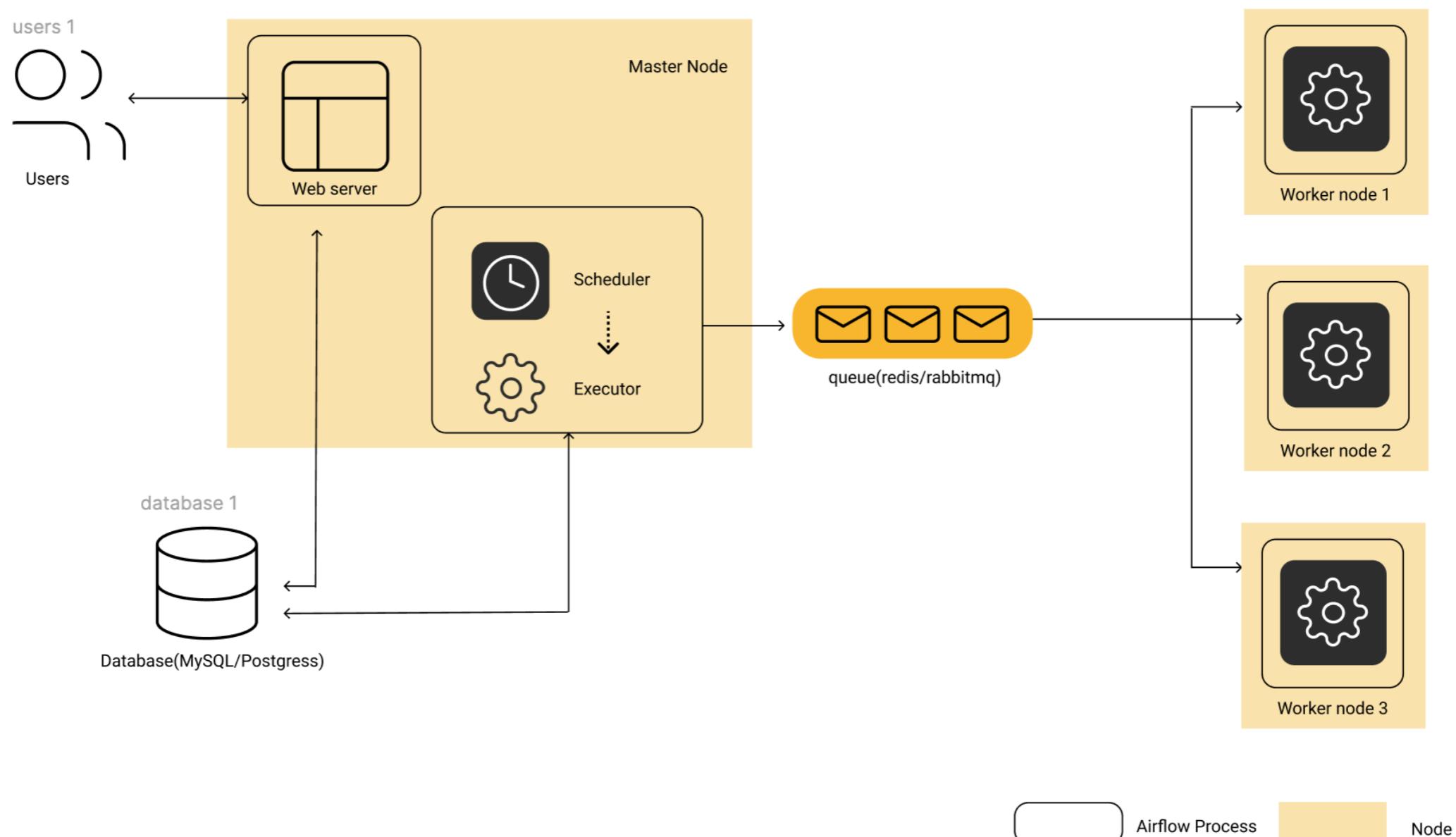
# Provider Packages

---

Providers packages include integrations with third party projects. They are updated independently of the Apache Airflow core. [Read the documentation >>](#)

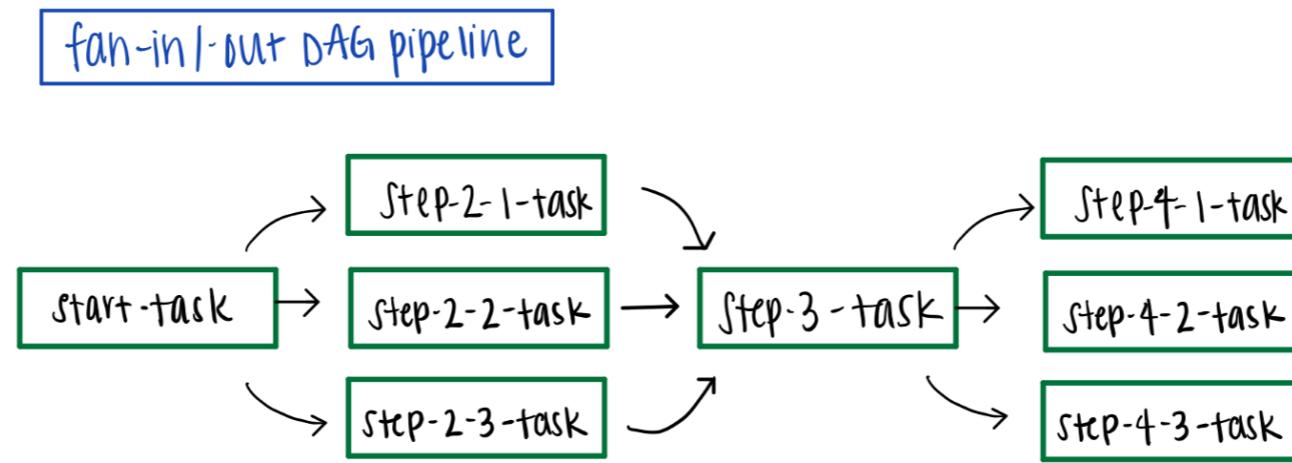
- [Airbyte](#)
- [Alibaba](#)
- [Amazon](#)
- [Apache Beam](#)
- [Apache Cassandra](#)
- [Apache Drill](#)
- [Apache Druid](#)
- [Apache HDFS](#)
- [Apache Hive](#)
- [Apache Kylin](#)
- [Apache Livy](#)
- [Apache Pig](#)
- [Apache Pinot](#)
- [Apache Spark](#)
- [Apache Sqoop](#)
- [Asana](#)
- [Celery](#)
- [IBM Cloudant](#)
- [Kubernetes](#)
- [Databricks](#)
- [Datadog](#)
- [DBT cloud](#)
- [Dingding](#)
- [Discord](#)
- [Docker](#)
- [Elasticsearch](#)
- [Exasol](#)
- [Facebook](#)
- [File Transfer Protocol \(FTP\)](#)
- [Github](#)
- [Google](#)
- [gRPC](#)
- [Hashicorp](#)
- [Hypertext Transfer Protocol \(HTTP\)](#)
- [Influx DB](#)
- [Internet Message Access Protocol \(IMAP\)](#)
- [Java Database Connectivity \(JDBC\)](#)
- [Jenkins](#)
- [Jira](#)
- [Microsoft Azure](#)
- [Microsoft PowerShell Remoting Protocol \(PSRP\)](#)
- [Microsoft SQL Server \(MSSQL\)](#)
- [Windows Remote Management \(WinRM\)](#)
- [MongoDB](#)
- [MySQL](#)
- [Neo4J](#)
- [ODBC](#)
- [OpenFaaS](#)
- [Opsgenie](#)
- [Oracle](#)
- [Pagerduty](#)
- [Papermill](#)
- [Plexus](#)
- [PostgreSQL](#)
- [Presto](#)
- [Qubole](#)
- [Redis](#)
- [Salesforce](#)
- [Samba](#)
- [Segment](#)
- [Sendgrid](#)
- [SFTP](#)
- [Singularity](#)
- [Slack](#)
- [Snowflake](#)
- [SQLite](#)
- [SSH](#)
- [Tableau](#)
- [Telegram](#)
- [Trino](#)
- [Vertica](#)
- [Yandex](#)
- [Zendesk](#)

# Airflow Architecture



# Airflow Components

---



- **DAG** (directed acyclic graph) is a collection of tasks in pipeline
- **Task** is the basic unit of execution, an instantiated operator object with unique task id, and can be arranged into DAG to define dependencies among tasks
- **Operator** is a predefined task template being used in many situations e.g. BashOperator, PythonOperator, EmailOperator, Sensors, TaskFlow, and others (database operators, cloud storage operators, etc.)
- Others: hooks, coms, trigger, pool, variables, etc.

# Example DAG

---

```
from datetime import datetime

from airflow import DAG
from airflow.decorators import task
from airflow.operators.bash import BashOperator

# A DAG represents a workflow, a collection of tasks
with DAG(dag_id="demo", start_date=datetime(2022, 1, 1), schedule="0 0 * * *") as dag:
    # Tasks are represented as operators
    hello = BashOperator(task_id="hello", bash_command="echo hello")

    @task()
    def airflow():
        print("airflow")

    # Set dependencies between tasks
    hello >> airflow()
```

# How to run airflow

---

- Two recommended approaches, standalone and docker

```
pip install apache-airflow
```

```
airflow db init
```

```
airflow users create --username admin --firstname Natawut --  
lastname Nupairoj --role Admin --email natawut.n@chula.ac.th
```

```
airflow standalone
```

# Airflow UI

The screenshot shows the Airflow UI interface on a Mac OS X system. The title bar includes the Airflow logo, navigation links (DAGs, Security, Browse, Admin, Docs), a timestamp (09:32 UTC), and a user icon (NN). Two yellow warning banners are present: one about using SQLite and another about SequentialExecutor.

The main section is titled "DAGs". It features a table with columns: DAG, Owner, Runs, Schedule, Last Run, Next Run, and Recent Tasks. The table lists several DAGs:

DAG	Owner	Runs	Schedule	Last Run	Next Run	Recent Tasks
concurrent_dag	airflow	1 (green circle), 2 (red circle)	1 day, 0:00:00	2022-03-28, 08:33:25	2022-03-28, 00:00:00	3 (grey circles)
cp_webcheck	airflow	3 (green circle)	1 day, 0:00:00	2022-03-28, 08:59:01	2022-03-28, 00:00:00	3 (grey circles)
cp_webcheck_failed	airflow	2 (red circle)	1 day, 0:00:00	2022-03-28, 09:00:46	2022-03-28, 00:00:00	3 (grey circles)
example_bash_operator example example2	airflow	2 (green circle)	0 0 ***	2015-01-02, 00:00:00	2022-03-27, 00:00:00	3 (grey circles)
example_branch_datetime_operator_2 example	airflow	4 (grey circles)	@daily		2022-03-27, 00:00:00	3 (grey circles)
example_branch_dop_operator_v3 example	airflow	4 (grey circles)	*/1 ****		2022-03-28, 09:30:00	3 (grey circles)
example_branch_labels	airflow	4 (grey circles)	@daily		2022-03-27, 00:00:00	3 (grey circles)

# Example: Simple BashOperator

dsde\_simplebash.py

---

```
from airflow.utils.dates import days_ago
from airflow import DAG
from airflow.operators.bash import BashOperator

# declare a DAG
with DAG(dag_id='dsde_simplebash', start_date=days_ago(1)):

    # declare a TASK
    echo = BashOperator(task_id='echo_template',
bash_command='echo "run_id = {{ run_id }} and ds = {{ ds }}"'')
```

# BashOperator

---

- BashOperator allows Airflow to run a task using bash script
- The exit code of the program/script will determine the result of the task execution
  - 0 - Success
  - 99 - Skipped
  - Something else - Failed
- Parameterizing bash\_command is possible using Jinja templates (refer to <https://airflow.apache.org/docs/apache-airflow/stable/templates-ref.html> for more details)
- Airflow captures the last line of the output from the script in ‘output’

# Example: Task Dependency

dsde\_simplebash2.py

```
from airflow.utils.dates import days_ago
from airflow import DAG
from airflow.operators.bash import BashOperator

# declare a DAG
with DAG(dag_id='dsde_simplebash_2', start_date=days_ago(1)):

    # declare 2 TASKS
    echo = BashOperator(task_id='echo_template', bash_command='echo "run_id
= {{ run_id }} and ds = {{ ds }}"')
    echo2 = BashOperator(task_id='echo_template_2', bash_command='echo "[2]
run_id = {{ run_id }} and ds = {{ ds }}"')

    # declare dependency
    echo >> echo2
```

# Example: Concurrent Tasks

dsde\_concurrent.py

---

```
from airflow.utils.dates import days_ago
from airflow import DAG

from airflow.operators.bash import BashOperator
from airflow.operators.dummy import DummyOperator

# 'with' enables DAG to become context managers; automatically assign new operators to
that DAG
with DAG('concurrent_dag', start_date=days_ago(1)) as dag:
    start = DummyOperator(task_id='start_task')
    ping = BashOperator(task_id='cp_check', bash_command='curl https://
www.cp.eng.chula.ac.th')
    ping2 = BashOperator(task_id='eng_check', bash_command='curl https://
www.eng.chula.ac.th')
    inform = BashOperator(task_id='inform_status', bash_command='echo "CP website still
works!"')

# creating DAG dependencies can be a long flow or multiple short flows
# start >> [ping, ping2] >> inform
start >> [ping, ping2]
ping >> inform
ping2 >> inform
```

# Airflow Common Operators

---

- BashOperator - executes a bash command
- PythonOperator - calls an arbitrary Python function
- EmailOperator - sends an email
- SimpleHttpOperator - sends an HTTP request
- MySqlOperator, SqliteOperator, PostgresOperator, MsSqlOperator, OracleOperator, JdbcOperator, etc. - executes a SQL command
- Sensor - an Operator that waits (polls) for a certain time, file, database row, S3 key, etc...
- Other specific operators: DockerOperator, HiveOperator, S3FileTransformOperator, PrestoToMySqlTransfer, SlackAPIOperator, etc.

# PythonOperator / PythonVirtualenvOperator

---

- PythonOperator provides more flexibility of running python code e.g. perform data conversion, train a model
- Input parameters can be passed via op\_kwargs/op\_args and XCom
- Return values are captured in ‘output’ variable

# Example: PythonOperator

dsde\_python.py

```
from airflow.utils.dates import days_ago
from airflow import DAG

from airflow.operators.bash import BashOperator
from airflow.operators.dummy import DummyOperator
from airflow.operators.python import PythonOperator

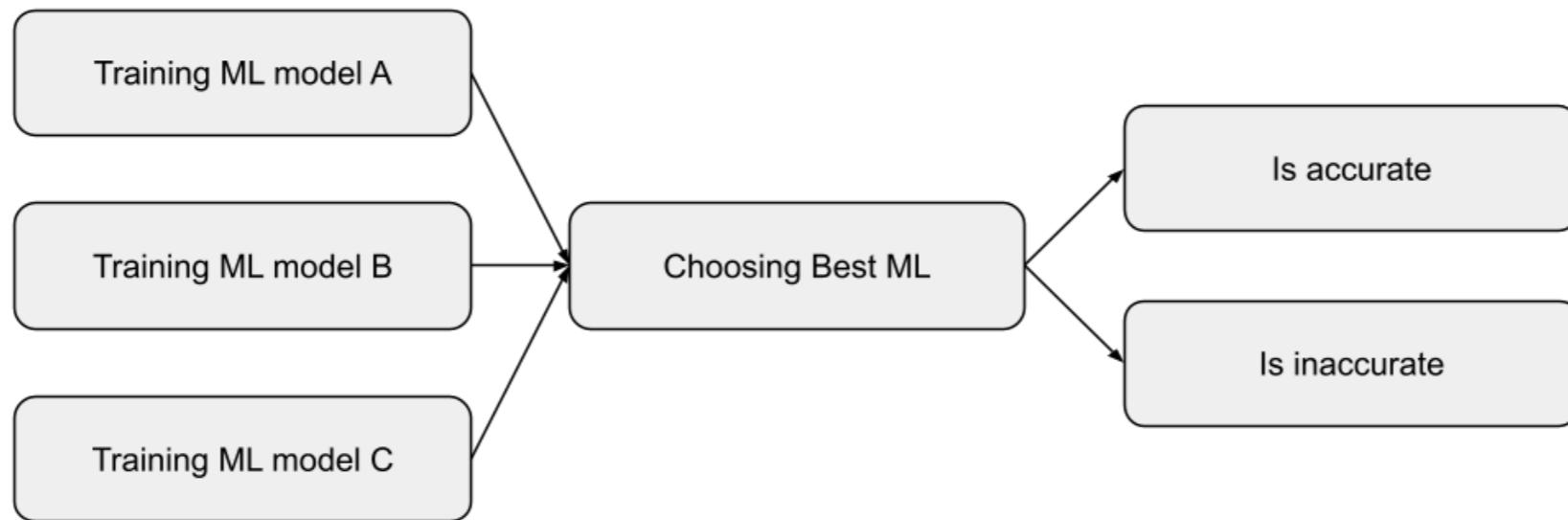
def show_status(web1, web2):
    print('All websites are running!')
    print('web1 = ', web1)
    print('web2 = ', web2)
    print('--- DONE ---')

# 'with' enables DAG to become context managers; automatically assign new operators to that DAG
with DAG('python_operator', start_date=days_ago(1)) as dag:
    start = DummyOperator(task_id='start_task')
    ping = BashOperator(task_id='cp_check', bash_command='curl https://www.cp.eng.chula.ac.th')
    ping2 = BashOperator(task_id='eng_check', bash_command='curl https://www.eng.chula.ac.th')
    inform = PythonOperator(task_id='inform_status', python_callable=show_status,
op_args=[ping.output, ping2.output])

    # creating DAG dependencies can be a long flow or multiple short flows
    # start >> [ping, ping2] >> inform
    start >> [ping, ping2]
    ping >> inform
    ping2 >> inform
```

# BranchPythonOperator

---



- Allows to follow a specific path according to a condition, evaluated in a python callable function
- Executes a Python function returning a list of task ids of the next tasks to execute

# Example: PythonBranchOperator

dsde\_branch.py

```
from airflow import DAG
from airflow.operators.python import BranchPythonOperator
from airflow.operators.dummy import DummyOperator
from datetime import datetime

def _choose_best_model(accuracy):
    if accuracy > 0.8:
        return 'accurate'
    return 'inaccurate'

with DAG('branch_oper', start_date=datetime(2021, 1, 1), catchup=False) as dag:
    choose_best_model = BranchPythonOperator(task_id='choose_best_model',
                                              python_callable=_choose_best_model,
                                              op_args=[0.75])
    accurate = DummyOperator(task_id='accurate')
    inaccurate = DummyOperator(task_id='inaccurate')
    choose_best_model >> [accurate, inaccurate]
```

See more DAG examples in airflow

# More about DAG

---

- DAG can be declared with the following parameters
  - start\_date - when to start, can be exact date or relative e.g. days\_ago(1)
  - schedule\_interval - how often should this DAG run
  - catchup - should scheduler kick off a DAG Run for any data interval that has not been run since the last data interval
  - default\_args - args to get passed on to each operator in this DAG, can override them on a per-task basis
- Airflow 2.0 introduces TaskFlow API with DAG and Task decorators

# Sensors

---

- a special type of Operator that are designed to wait until something happens, and then succeed so their downstream tasks can run
- Can be time-based, or waiting for a file, or an external event
- Sensors have three different modes of running
  - poke (default): The Sensor takes up a worker slot for its entire runtime
  - reschedule: The Sensor takes up a worker slot only when it is checking, and sleeps for a set duration between checks
  - smart sensor: There is a single centralized version of this Sensor that batches all executions of it

# Other Components

---

- Connectors - repository of configurations to connect to external systems
- Hooks - high-level interface (driver) to an external platform
- Variables - generic way to store and retrieve configured values (key-value pair), similar to environment variables
- XCom - operator cross-communication - let tasks exchange information
- Executors - mechanism how task instances get to run (default = LocalExecutors)
- Trigger - run DAG from CLI, REST API, Web UI, TriggerDagRunOperator

# Use Cases

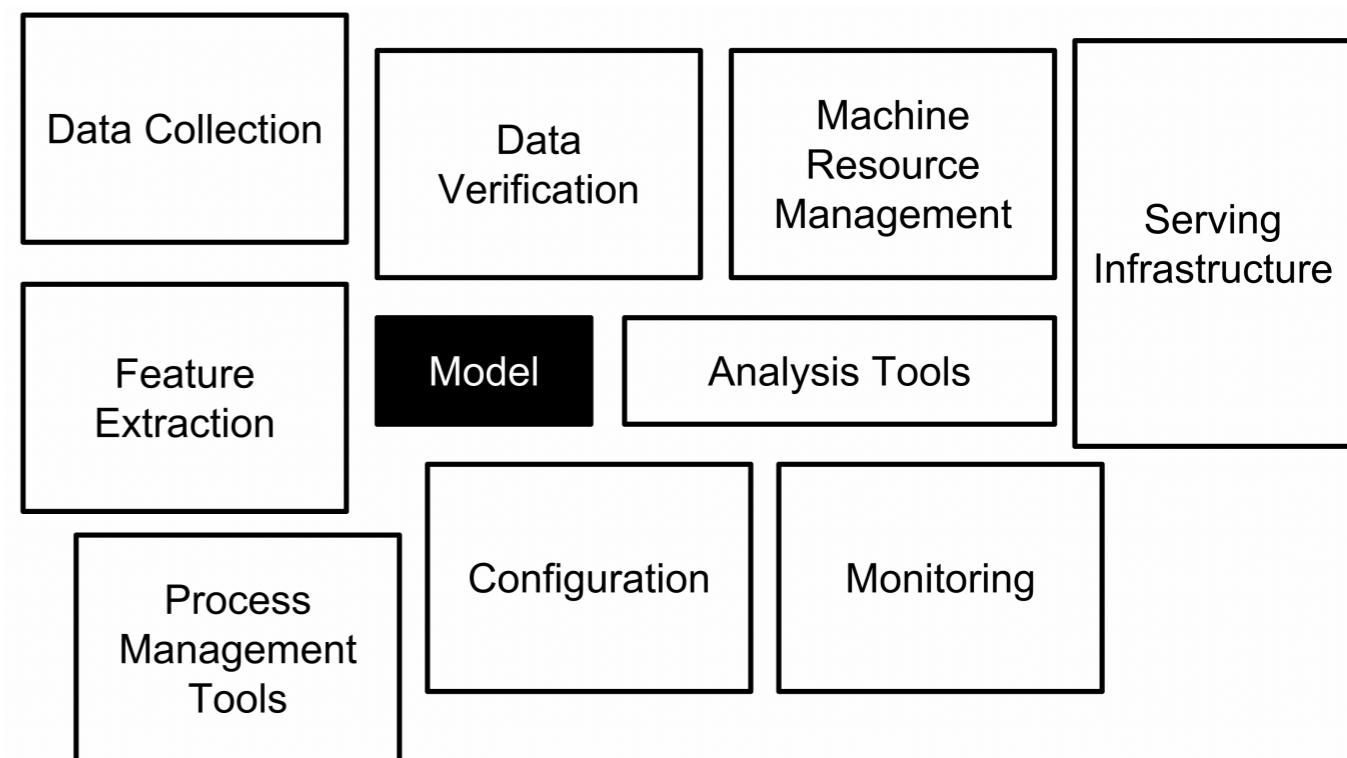
---

- ETL Pipelines that extract data from multiple sources and run Spark jobs or any other data transformations
- Collecting Sensor data and move to Data Lake/Data warehouse
- Training machine learning models
- Orchestrating automated testing
- Report generation

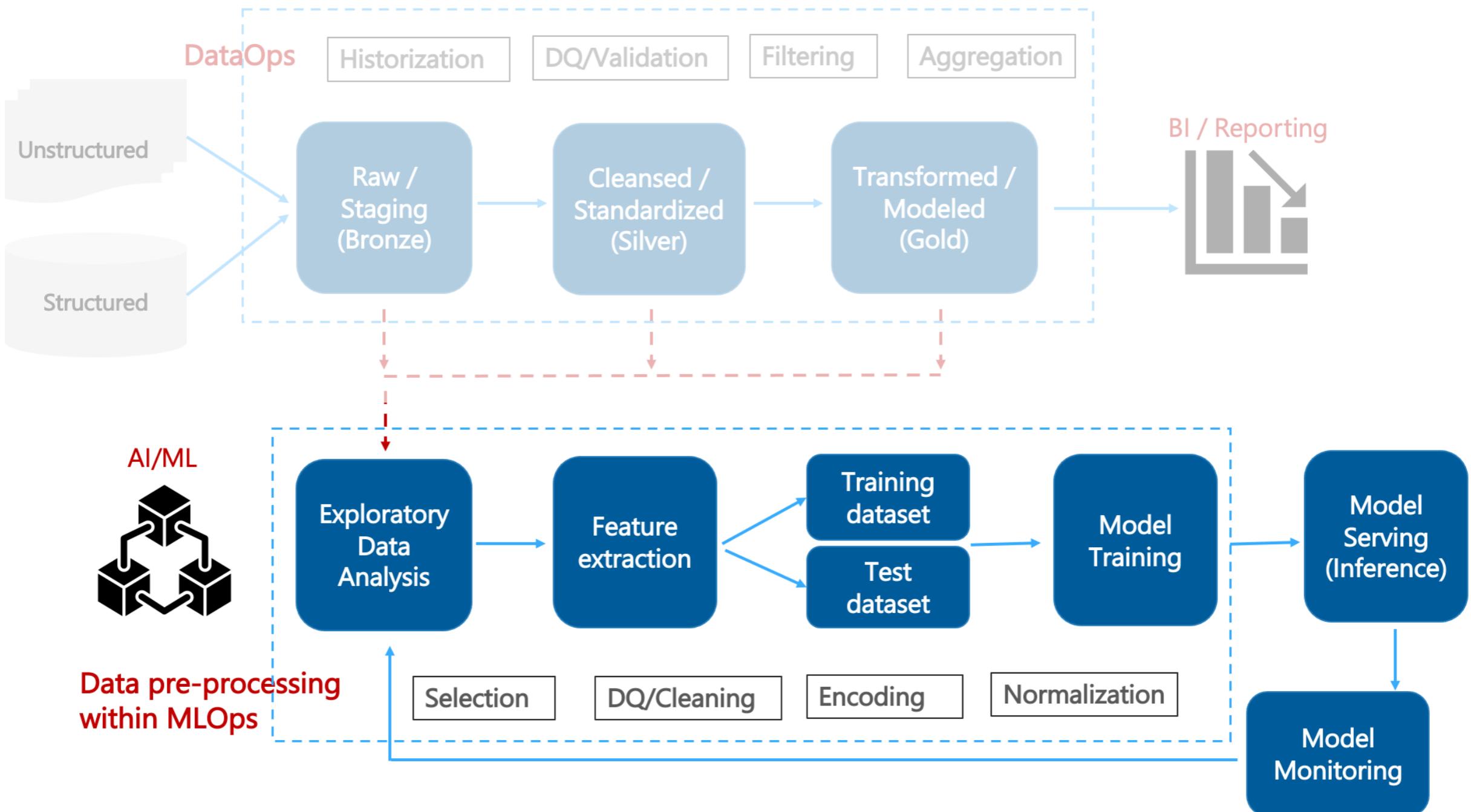
# Orchestrating ML Pipeline

---

Ops Stars



# DataOps and MLOps



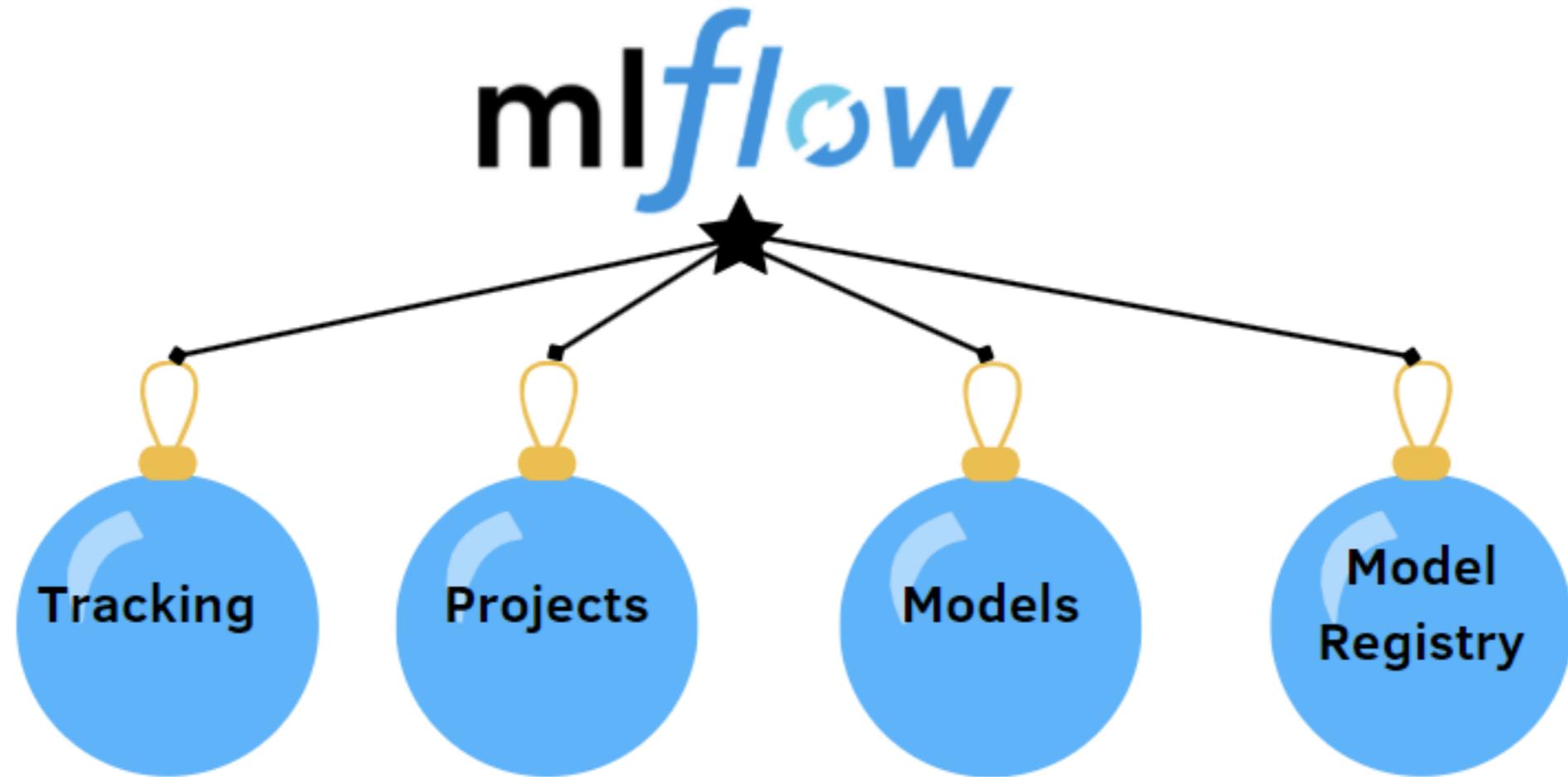
# MLOps Orchestration Requirements

---

- In addition to workflow control, orchestrating MLOps requires some unique requirements
  - Code + Data = combination of development and data pipeline
  - Model version control
  - Model performance tracking during training (and serving)

# MLflow Components

---



Track and record experiments

Package code in a reusable and reproducible way

Persist ML model for deployment in several environments

Manage models and their lifecycle centrally

# Key Concepts in MLflow

---

- A run is a collection of parameters, metrics, labels, and artifacts related to the training process of a machine learning model
- An experiment is the basic unit of MLflow organization ; all MLflow runs belong to an experiment
- For each experiment, you can analyze and compare the results of different runs, and easily retrieve metadata artifacts for analysis using downstream tools

# How to run MLflow

---

- Install the MLflow

```
pip install mlflow
```

- Install the tutorial code

```
git clone https://github.com/mlflow/mlflow
```

Harutaka Kawamura, 2 years ago | 4 authors (Matei Zaharia and others)

```
1 # The data set used in this example is from http://archive.ics.uci.edu/ml/datasets/Wine+Quality Matei Zaharia, 4 years ago
2 # P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis.
3 # Modeling wine preferences by data mining from physicochemical properties. In Decision Support Systems, Elsevier, 47(4):547-560.
4
5 import os
6 import warnings
7 import sys
8
9 import pandas as pd
10 import numpy as np
11 from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
12 from sklearn.model_selection import train_test_split
13 from sklearn.linear_model import ElasticNet
14 from urllib.parse import urlparse
15 import mlflow
16 import mlflow.sklearn
17
18 import logging
19
20 logging.basicConfig(level=logging.WARN)
21 logger = logging.getLogger(__name__)
22
23
24 def eval_metrics(actual, pred):
25     rmse = np.sqrt(mean_squared_error(actual, pred))
26     mae = mean_absolute_error(actual, pred)
27     r2 = r2_score(actual, pred)
28     return rmse, mae, r2
29
30
31 if __name__ == "__main__":
```

```
58     with mlflow.start_run():
59         lr = ElasticNet(alpha=alpha, l1_ratio=l1_ratio, random_state=42)
60         lr.fit(train_x, train_y)
61
62         predicted_qualities = lr.predict(test_x)
63
64         (rmse, mae, r2) = eval_metrics(test_y, predicted_qualities)
65
66         print("Elasticnet model (alpha=%f, l1_ratio=%f):" % (alpha, l1_ratio))
67         print("  RMSE: %s" % rmse)
68         print("  MAE: %s" % mae)
69         print("  R2: %s" % r2)
70
71         mlflow.log_param("alpha", alpha)
72         mlflow.log_param("l1_ratio", l1_ratio)
73         mlflow.log_metric("rmse", rmse)
74         mlflow.log_metric("r2", r2)
75         mlflow.log_metric("mae", mae)
76
77     tracking_url_type_store = urlparse(mlflow.get_tracking_uri()).scheme
78
79     # Model registry does not work with file store
80     if tracking_url_type_store != "file":
81
82         # Register the model
83         # There are other ways to use the Model Registry, which depends on the use case,
84         # please refer to the doc for more information:
85         # https://mlflow.org/docs/latest/model-registry.html#api-workflow
86         mlflow.sklearn.log_model(lr, "model", registered_model_name="ElasticnetWineModel")
87     else:
88         mlflow.sklearn.log_model(lr, "model")
```

# Running Model and View Results

---

- At mlflow tutorial code repo:

```
cd mlflow/examples  
python sklearn_elasticnet_wine/train.py  
python sklearn_elasticnet_wine/train.py [alpha] [l1_ratio]
```

- View results (run at mlflow/examples) and use your browser to <http://127.0.0.1:5000>

```
cd mlflow/examples  
mlflow ui
```

mlflow 1.30.0 Experiments Models GitHub Docs

Experiments Default 

Search Experiments  X

Default  

Track machine learning training runs in experiments. [Learn more](#)

Experiment ID: 0

> Description [Edit](#)

 Refresh  Compare  Delete  Download CSV  Created  All time

Columns  Only show differences   metrics.rmse < 1 and params.model = "tree"   Filter  Clear

Showing 5 matching runs

	↓ Created	Duration	Run Name	User	Source	Version	Models	Metrics	Parameters			
								mae	r2	rmse	alpha	I1_ratio
<input type="checkbox"/>	4 minutes ago	2.5s	selective-o...	natawut	 train.py	fe60af	 sklearn	0.627	0.109	0.793	0.5	0.5
<input type="checkbox"/>	16 minutes ago	2.4s	bedecked-...	natawut	 train.py	fe60af	 sklearn	0.673	0.017	0.833	1.0	1.0
<input type="checkbox"/>	16 minutes ago	2.5s	rare-calf-8...	natawut	 train.py	fe60af	 sklearn	0.627	0.109	0.793	0.5	0.5
<input type="checkbox"/>	31 minutes ago	2.6s	redolent-cr...	natawut	 train.py	fe60af	 sklearn	0.673	0.017	0.833	1.0	1.0
<input type="checkbox"/>	31 minutes ago	2.5s	bedecked-...	natawut	 train.py	fe60af	 sklearn	0.627	0.109	0.793	0.5	0.5

[Load more](#)

# Packaging and Serving Model

---

- Model can be packaged for running with its own python environment (for other to test or run on cloud)

```
mlflow run sklearn_elasticnet_wine -P alpha=0.42
```

- Model can be deployed as a standalone service by selecting the run that we want to use

```
mlflow models serve -m ./mlruns/0/12334a9519a94de99a3363341b69c97c/artifacts/model -p 1234
```

```
curl -X POST -H "Content-Type:application/json; format=pandas-split" --data '{"columns": ["alcohol", "chlorides", "citric acid", "density", "fixed acidity", "free sulfur dioxide", "pH", "residual sugar", "sulphates", "total sulfur dioxide", "volatile acidity"], "data": [[12.8, 0.029, 0.48, 0.98, 6.2, 29, 3.33, 1.2, 0.39, 75, 0.66]]}' http://127.0.0.1:1234/invocations
```

mlflow 1.30.0 Experiments Models GitHub Docs

Default > selective-owl-216

## selective-owl-216

Run ID: 12334a9519a94de99a3363341b69c97c Date: 2022-10-31 20:48:19 Source: train.py

Git Commit: fe60af98c8a783ed3ced24150f2784f980ce790 User: natawut Duration: 2.5s

Status: FINISHED Lifecycle Stage: active

> Description Edit

> Parameters (2)

> Metrics (3)

> Tags

> Artifacts

Full Path: ./mlruns/0/12334a9519a94de99a3363341b69c97c/artifacts/model

**MLflow Model**

The code snippets below demonstrate how to make predictions using the logged model. You can also register it to the [model registry](#) to version control

**Model schema**

Input and output schema for your model. [Learn more](#)

Name	Type
No schema. See <a href="#">MLflow docs</a> for how to include input and output schema with your model.	

**Make Predictions**

Predict on a Spark DataFrame:

```
import mlflow
from pyspark.sql.functions import struct, col
logged_model = 'runs:/12334a9519a94de99a3363341b69c97c/model'
```

# Load model as a Spark UDF. Override result\_type if the model does not return double values.

```
loaded_model = mlflow.pyfunc.spark_udf(spark, model_uri=logged_model, result_type='double')
```

# Predict on a Spark DataFrame.

```
df.withColumn('predictions', loaded_model(struct(*map(col, df.columns))))
```

Predict on a Pandas DataFrame:

# Provisioning ML Model

---

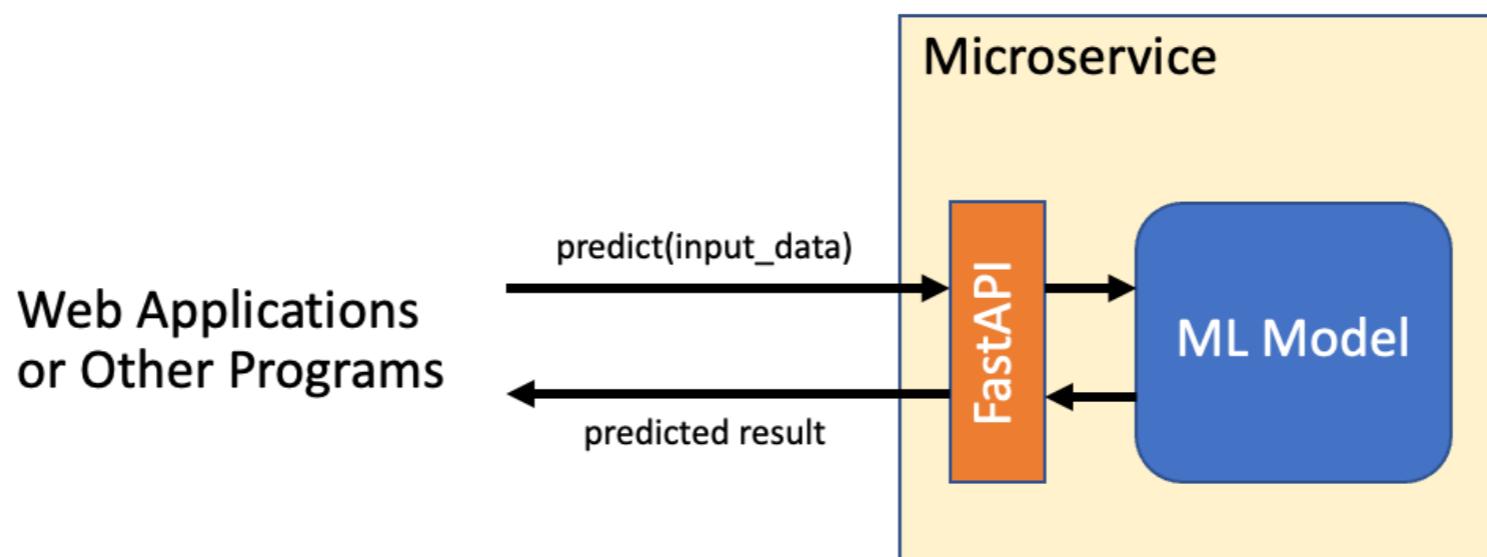
Ops Stars



# Provisioning Machine Learning Model

---

- Encapsulating your model as a microservice is a popular approach to deploy machine learning model
- The model becomes a black box and can be invoked using REST protocol
- Using a microframework such as FastAPI can simplify microservice development



# Microframework

---

- Minimalistic web application frameworks designed for Microservices and API development
- Facilitate receiving an HTTP request, routing the HTTP request to the appropriate controller, dispatching the controller, and returning an HTTP response
- Designed for building the APIs for another service or application; therefore, lacking most of the functionality expected in full-stack framework

# REST - Heart and Soul of Microservices

---

- REpresentational State Transfer
- Lightweight and simple
  - Only simple XML or JSON on HTTP
  - Utilize HTTP methods instead of using “envelope” style
  - URI as method identification
  - Emulate CRUD operations

# REST over HTTP



# FastAPI

---

- Web framework for developing RESTful APIs in Python based on Pydantic and starlette
- Provide several facilities for REST development
  - Data validation / serialization / deserialization
  - Auto-generate OpenAPI documents
  - Asynchronous programming
  - Dependency Injection
  - Support GraphQL for complex data model
  - Run with Uvicorn and Unicorn

# Install Fastapi

---

- Very simple installation process with pip
- Make sure that you install both fastapi and unicorn

```
pip install fastapi uvicorn
```

# Example: Simple Service

---

```
# simple fastapi example

import uvicorn
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def root():
    return {"message": "My first FastAPI service"}

if __name__ == "__main__":
    uvicorn.run('simple:app', host="0.0.0.0", port=8000, reload=True)
```

# Testing Our First Service

---

- Test the service with your browser at <http://localhost:8000>

```
{"message": "My first FastAPI service"}
```

- More information at <http://localhost:8000/docs>

The screenshot shows the FastAPI documentation for version 0.1.0. At the top, it says "FastAPI 0.1.0 OAS3 /openapi.json". Below this, under the "default" section, there is a "GET / Root" operation. The "Parameters" section shows "No parameters". The "Responses" section lists a single 200 status code with a "Successful Response" description. The "Media type" dropdown is set to "application/json", which is highlighted with a green border. Below the dropdown, it says "Controls Accept header.", "Example Value", and "Schema". The schema example is shown as "string".

- You can also try <http://localhost:8000/redoc>

The screenshot shows the Redoc interface for the FastAPI documentation. At the top, it says "FastAPI (0.1.0)" and "Download OpenAPI specification: Download". Below this, it says "Root". Under the "Responses" section, it shows a green bar indicating "> 200 Successful Response". At the bottom, there is a dark-themed panel for the "Response samples" of the root endpoint. It shows a "GET /" button, a dropdown menu, a "200" status code button, and a "Content type application/json" section.

- Beware of browser cache
- You can also use other tools: curl, postman, etc.

# Example: Path Parameter with Type Validation

---

```
# path parameter example

import uvicorn
from fastapi import FastAPI

app = FastAPI()

@app.get('/items/{item_id}')
async def read_item(item_id: int):
    return { 'item_id': item_id }

if __name__ == "__main__":
    uvicorn.run('path_param:app', host="0.0.0.0", port=8000, reload=True)
```

Test this service at: <http://localhost:8000/items/1> and api document at: <http://localhost:8000/docs>

# REST CRUD Operations

---

HTTP	CRUD	Decorator
GET	Read	<code>@app.get( )</code>
POST	Create, Update, Delete	<code>@app.post( )</code>
PUT	Create, Overwrite / Replace	<code>@app.put( )</code>
DELETE	Delete	<code>@app.delete( )</code>

## Note

- FastAPI doesn't enforce any specific meaning
- Other operations (options, head, patch, trace) are also supported

# POST Operation

---

- POST operation allows passing complex input data in JSON via request body
- Declaring data type with pydantic, FastAPI will
  - Read the body of the request as JSON
  - Convert the corresponding types if needed
  - Validate the data and return a clear error if it is invalid
  - Put the received data in the input parameter of the function

# Example: POST for Complex Input Parameter

---

```
# post example

import uvicorn
from fastapi import FastAPI

from typing import Optional
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    description: Optional[str] = None
    price: float
    tax: Optional[float] = None

app = FastAPI()

@app.post("/items/")
async def create_item(item: Item):
    return item

@app.get('/items/{item_id}')
async def read_item(item_id: int):
    return { 'item_id': item_id }

if __name__ == "__main__":
    uvicorn.run('post:app', host="0.0.0.0", port=8000, reload=True)
```

# Example: Using Path Parameter and Request Body

---

```
# put example

import uvicorn
from fastapi import FastAPI

from typing import Optional
from pydantic import BaseModel

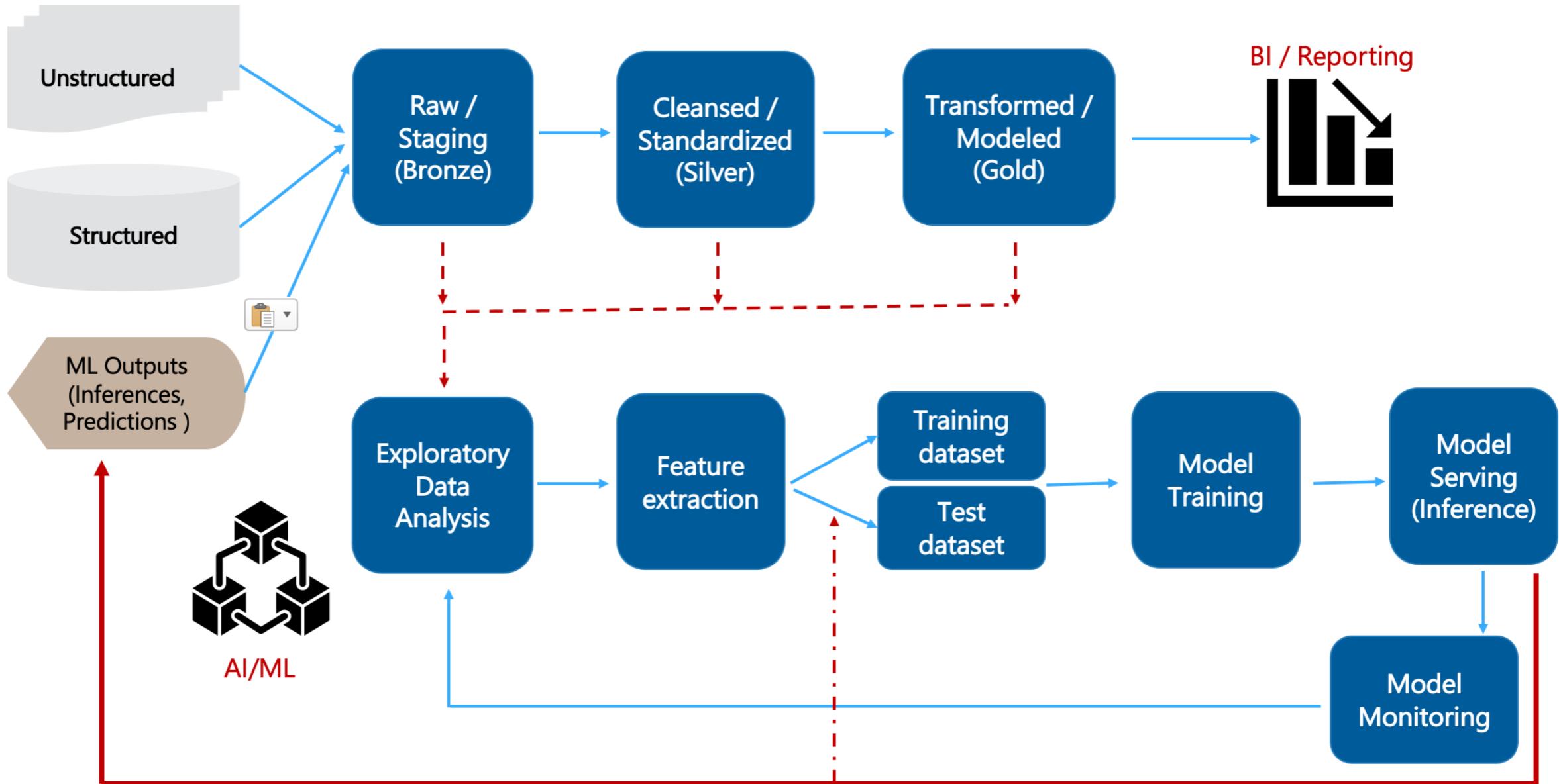
class Item(BaseModel):
    name: str
    description: Optional[str] = None
    price: float
    tax: Optional[float] = None

app = FastAPI()

@app.put("/items/{item_id}")
async def update_item(item_id: int, item: Item):
    return {"item_id": item_id, **item.dict()}

if __name__ == "__main__":
    uvicorn.run('put:app', host="0.0.0.0", port=8000, reload=True)
```

# DataOps and MLOps Full Loop



# Conclusion

---

- Data pipeline can be very complex as it requires task orchestration across multiple platforms
- Airflow provides sophisticated data pipeline orchestration system with configuration as code and other comprehensive facilities
- Airflow is good for orchestrating data pipeline, ETL, and batch execution, but not a Spark/Flink replacement

# Conclusion

---

- MLflow provides comprehensive framework for MLOps including performance tracking, model packaging, model serving
-

# Conclusion

---

- Microframework is one of simplest approach to create a microservice
- FastAPI becomes very popular to ML service deployment as it can integrate with other python packages with ease
- FastAPI facilitates input data validation / deserialization, return data serialization, and automatic API document generation
- Combine with docker, ML microservice will be scalable and simple to deploy

# References

---

- P.S. Tan, “Differences: DevOps, ITOps, MLOps, DataOps, ModelOps, AIOps, SecOps, DevSecOps”, <https://medium.com/vitrox-publication/differences-devops-itops-mlops-dataops-modelops-aiops-secops-devsecops-part-1-3-8b238cf72942>
- D. Biswas, “Bridging DataOps and MLOps: ML model inferences as a new Data Source”, <https://towardsdatascience.com/bridging-dataops-mlops-301f010caf30>
- run.ai, “MLflow: the Basics and a Quick Tutorial”, <https://www.run.ai/guides/machine-learning-operations/mlflow>
- T. Danka, “You Should Start Using FastAPI Now”, <https://towardsdatascience.com/you-should-start-using-fastapi-now-7efb280fec02>
- T. Danka, “How to properly ship and deploy your machine learning model”, <https://towardsdatascience.com/how-to-properly-ship-and-deploy-your-machine-learning-model-8a8664b763c4>