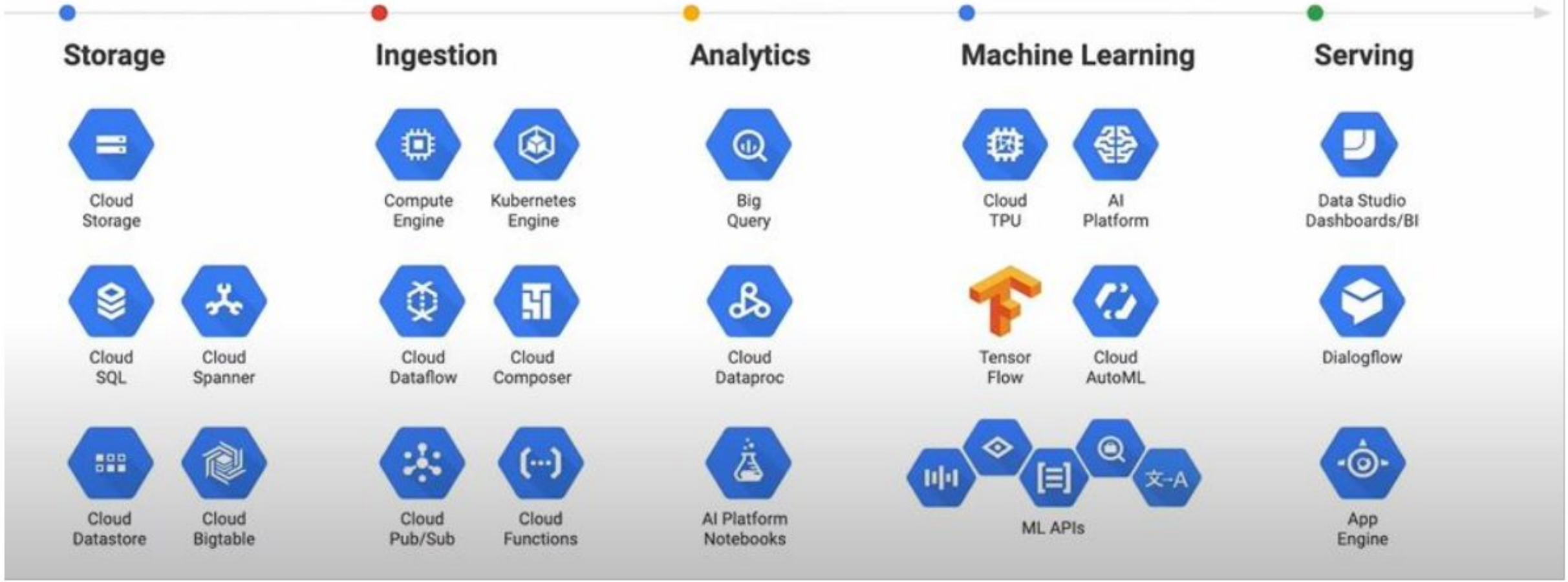
A Professional Machine Learning Engineer designs, builds, and productionizes ML models to solve business challenges using Google Cloud technologies and knowledge of proven ML models and techniques. The ML Engineer collaborates closely with other job roles to ensure long-term success of models. The ML Engineer should be proficient in all aspects of model architecture, data pipeline interaction, and metrics interpretation. The ML Engineer needs familiarity with application development, infrastructure management, data engineering, and security. Through an understanding of training, retraining, deploying, scheduling, monitoring, and improving models, they design and create scalable solutions for optimal performance.


COMPUTE ENGINE
APP ENGINE
KUBERNETES ENGINE
CLOUD RUN
CLOUD STORAGE
NETWORKING EGRESS
CLOUD LOAD BALANCING
INTERCONNEC & CLOUD VPN
BIGQUERY
BIGQUERY ML
DATASTORE
FIRESTORE
DATAPROC
DATAFLOW
CLOUD SQL
CLOUD BIGTABLE
PUB/SUB
OPERATIONS SUITE
CLOUD DNS
MICROSOFT AD
CLOUD TRANSLATION
CLOUD VISION
CLOUD CDN
SPEECH-TO-TEXT
NL API
AI PLATFORM
CLOUD KMS
CLOUD SPANNER
CLOUD FUNCTIONS
CLOUD ENDPOINTS
CLOUD DLP
DATAPREP
IOT CORE
VIDEO INTELLIGENCE
MEMORY STORE
MEMORY STORE
DIALOGFLOW
CLOUD COMPOSER
HEALTHCARE API
IDENTITY PLATFORM
SCC
CLOUD SCHEDULER
FILESTORE
ARTIFACT REGISTRY
SECRET MANAGER
RECAPTCHA
ORBITERA
TALENT SOLUTION


ROLE-BASED SUPPORT
PREMIUM SUPPORT
PSO
Free!
FREE

The suite of big data products on Google Cloud



Ingest	Store	Process & Analyze	Explore & Visualize
App Engine	Cloud Storage	Cloud Dataflow	Cloud Datalab
Compute Engine	Cloud SQL	Cloud Dataproc	Google Data Studio
Kubernetes Engine	Cloud Datastore	BigQuery	Google Sheets
Cloud Pub/Sub	Cloud Bigtable	Cloud ML	
Stackdriver Logging	BigQuery	Cloud Vision API	
Cloud Transfer Service	Cloud Storage for Firebase	Cloud Speech API	
Transfer Appliance	Cloud Firestore	Translate API	
	Cloud Spanner	Cloud Natural Language API	
		Cloud Dataprep	
		Cloud Video Intelligence API	

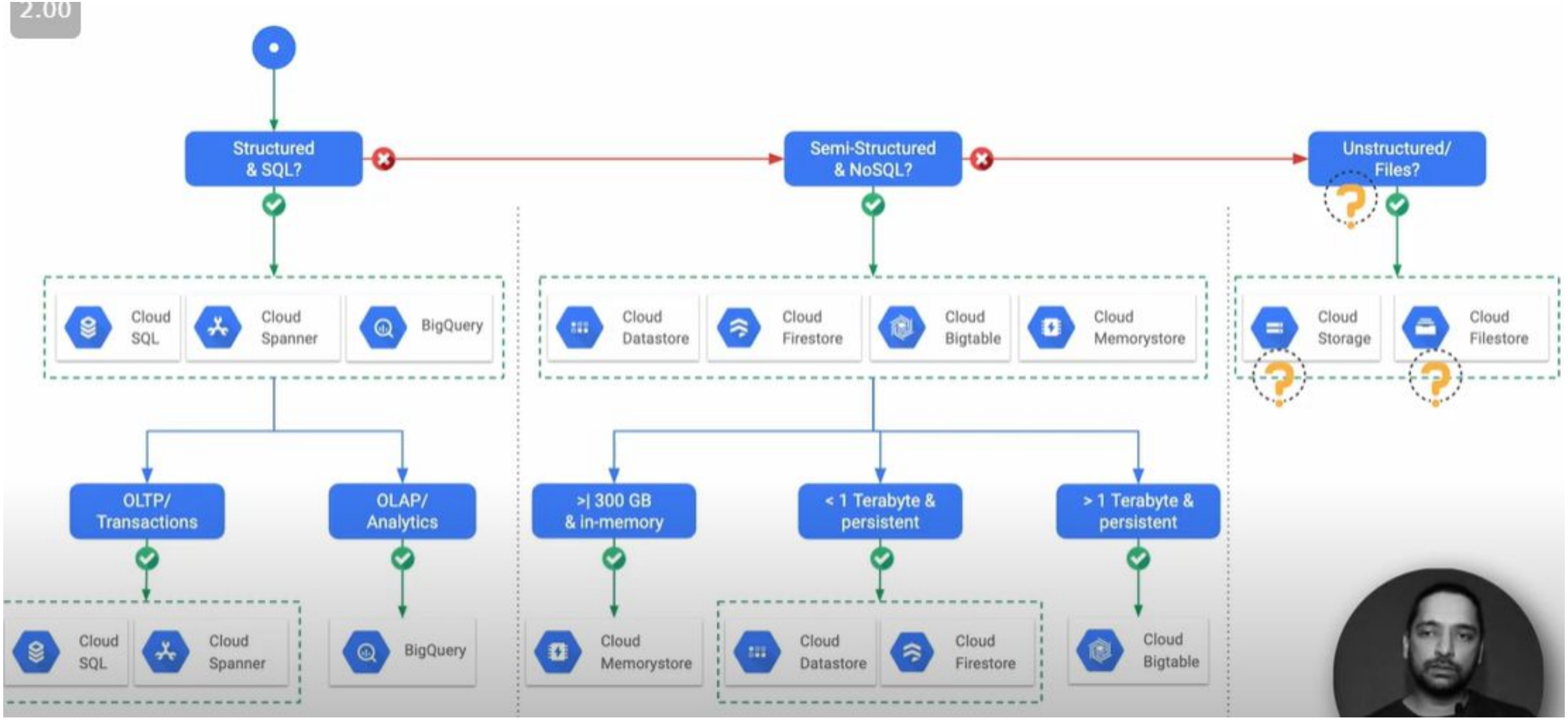
Transactions and ACID Properties

- A sequence of database operations that satisfies the ACID properties is a transaction in Database terminology.
- Atomicity: each transaction is a single "unit", succeeds or fails completely
  - the transaction cannot be observed to be in progress by another database client
- Consistency: a transaction takes a database from one valid state to another
  - any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof
- Isolation: concurrent execution of transactions results in same state as sequential execution
- Durability: guarantees that once a transaction has been committed, it will remain committed even in the case of a system failure
  - completed transactions are recorded in persistent storage

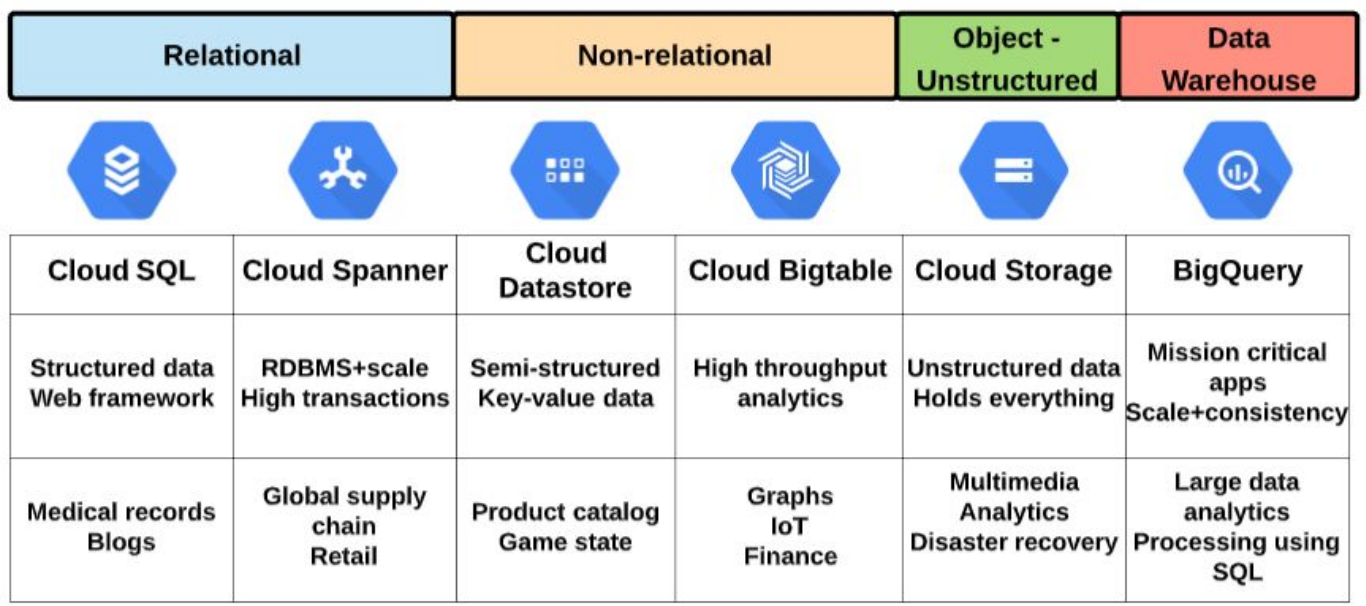
SQL vs NoSQL vs NewSQL

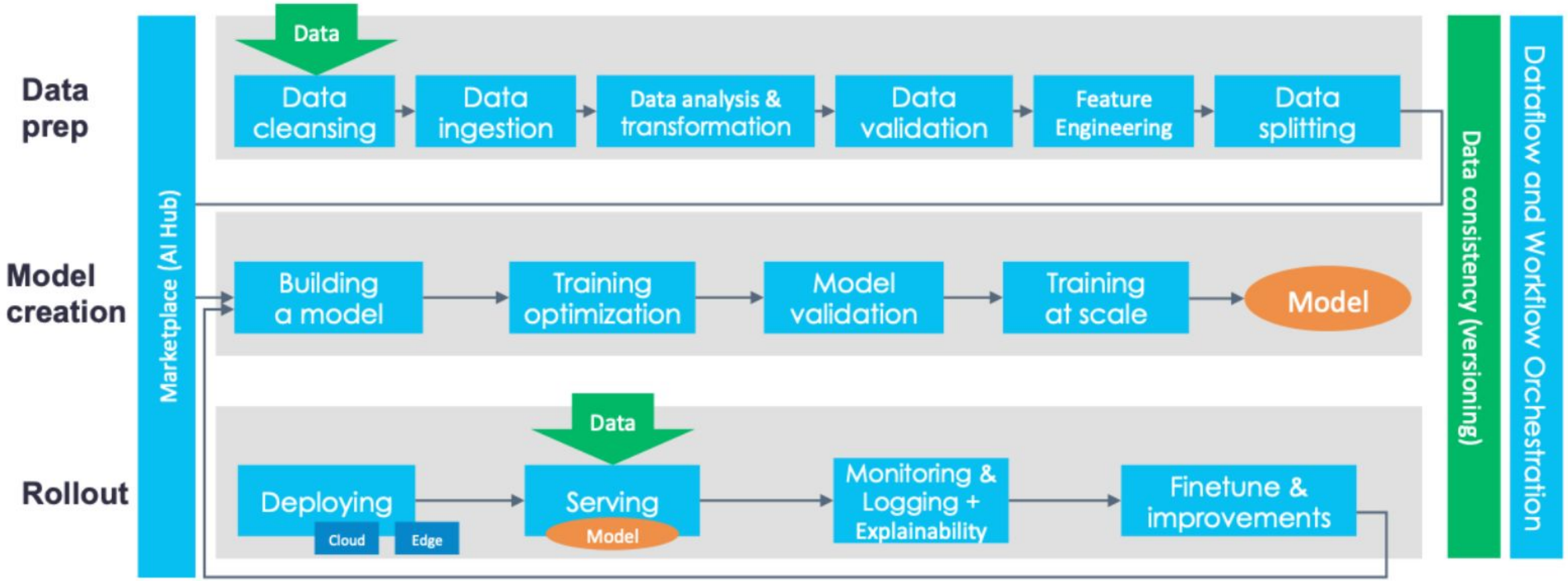
- SQL databases are typically transactional databases.
- Transactional databases follow ACID principles, adhering to which reduces horizontal scalability.
- NoSQL databases forego ACID properties often to achieve horizontal scale.
  - E.g. Bigtable
- NoSQL - databases that do not use SQL to access the data.
- Broad category of DBs with different storage formats - JSON, Key-Value, etc.
  - Data is non-relational. Eg. Datastore, Firestore, Memorystore, Bigtable
- NoSQL data is typically semi-structured
- NewSQL DBs combine both horizontal scaling and transactions processi
  - E.g. Spanner


Is your data structured?
NO
YES
Do you need Mobile SDKs?
YES
NO
Cloud Storage for Firebase
Cloud Storage
Is your workload analytics?
YES
NO
Is your data relational?
YES
NO
Do you need horizontal scalability?
YES
NO
Cloud Spanner
Cloud SQL
Do you need Mobile SDKs?
YES
NO
Cloud Firestore for Firebase
Cloud Datastore
Do you need updates or low-latency?
YES
NO
Cloud Bigtable
BigQuery


2:00
Structured & SQL?
Semi-Structured & NoSQL?
Unstructured/ Files?
Cloud SQL
Cloud Spanner
BigQuery
Cloud Datastore
Cloud Firestore
Cloud Bigtable
Cloud Memorystore
Cloud Storage
Cloud Filestore
OLTP/ Transactions
OLAP/ Analytics
>| 300 GB & in-memory
< 1 Terabyte & persistent
> 1 Terabyte & persistent
Cloud SQL
Cloud Spanner
BigQuery
Cloud Memorystore
Cloud Datastore
Cloud Firestore
Cloud Bigtable

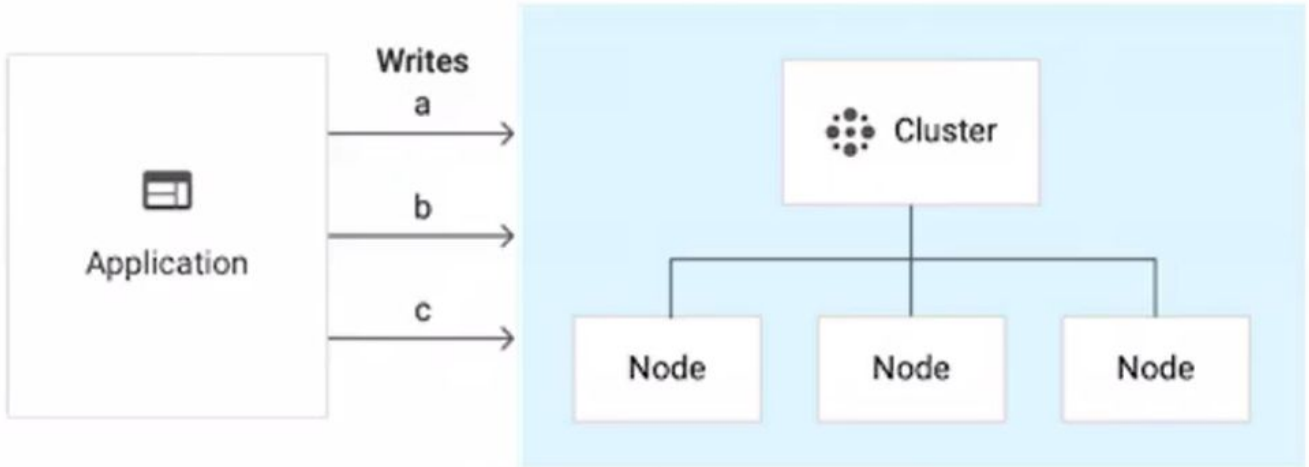
Google Platform Product	Service Function
Storage	Unified object storage
CloudSQL	Fully-managed MySQL database
BigTable	NoSQL massive data big data service
BigQuery	Petabyte scale data warehouse
Pub/Sub	Asynchronous messaging service
Cloud Dataflow	Data Processing (Pipelines)
Cloud DataProc	Managed Hadoop and Spark
TensorFlow	Machine learning language
Cloud Datastore	NoSQL database (think adhoc storage)



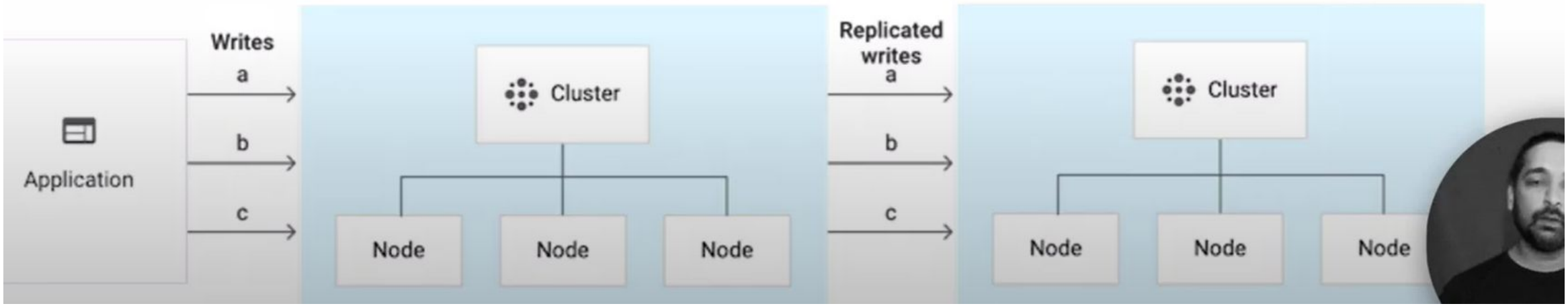


https://databaseline.tech/a-tour-of-end-to-end-ml-platforms/

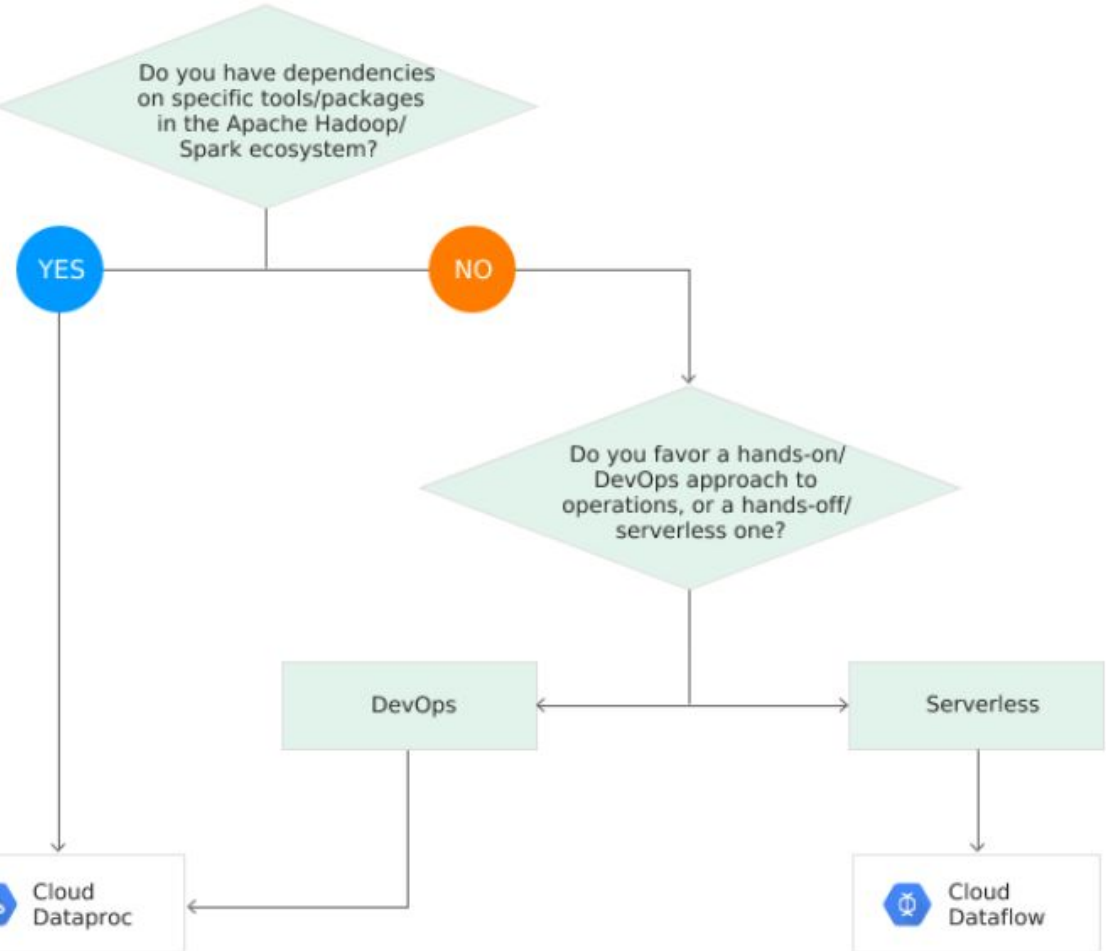
BigTable Performance



If replicated across multiple clusters, reads could become faster from a closer region, but write performance might reduce due to replicated writes.



Dataflow vs. Dataproc decision tree



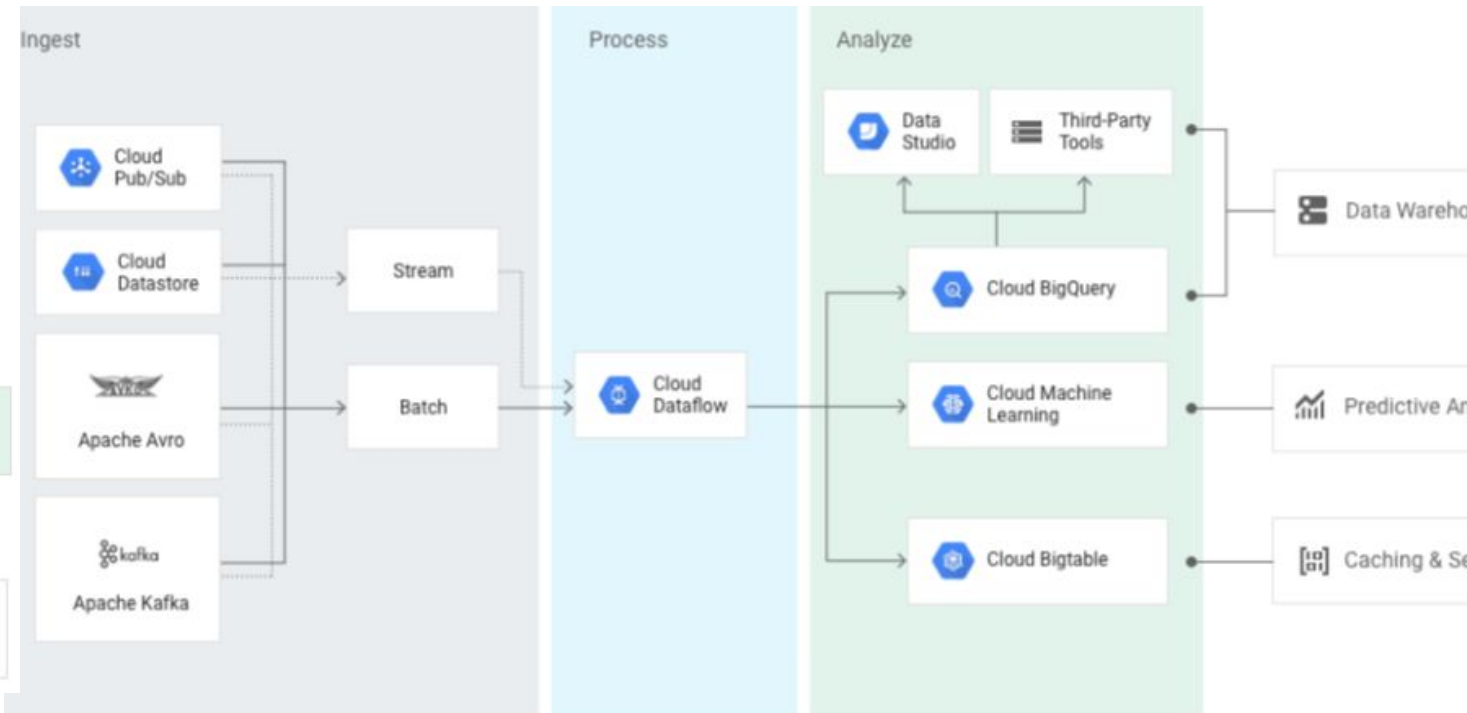
Dataproc:

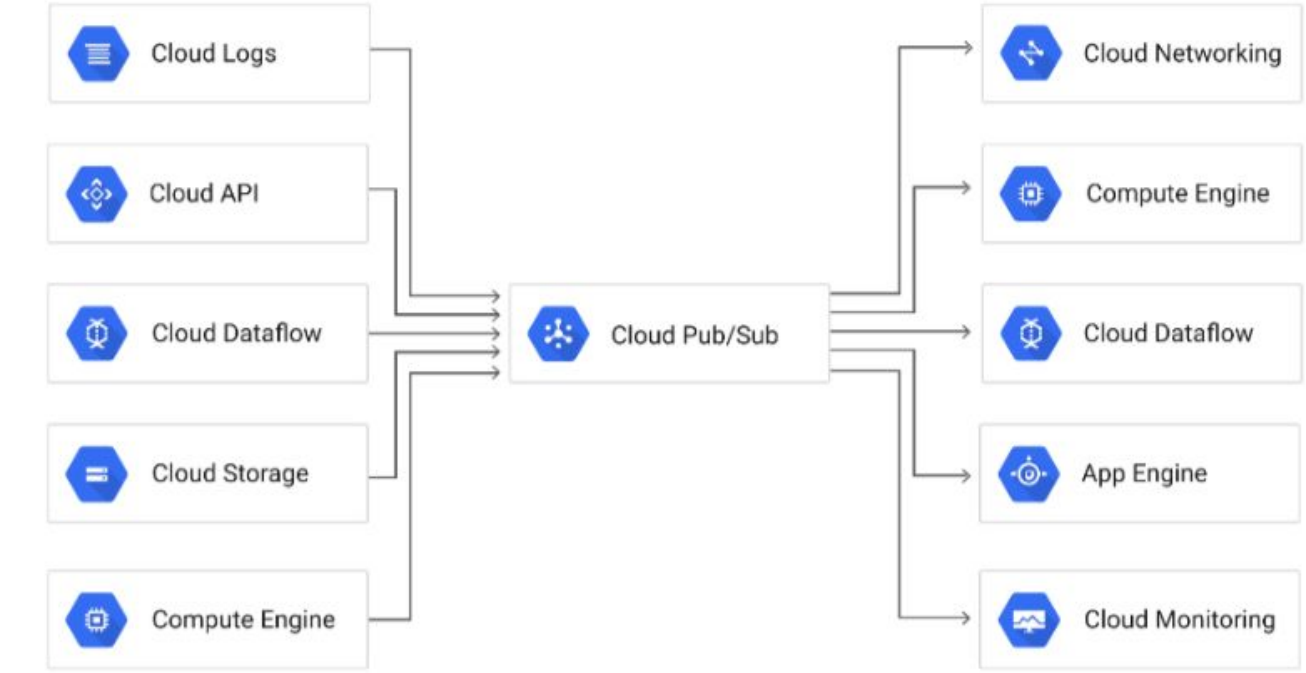
- Familiar tools/packages
- Employee skill sets
- Existing pipelines

Dataflow:

- Less Overhead
- Unified batch and stream processing
- Pipeline portability across Dataflow, Spark, and Flink as runtimes

WORKLOADS	CLOUD DATAPROC	CLOUD DATAFLOW
Stream processing (ETL)		X
Batch processing (ETL)	X	X
Iterative processing and notebooks	X	
Machine learning with Spark ML	X	
Preprocessing for machine learning		X (with Cloud ML Engine)





High-performance object storage

Backup and archival storage

HIGH FREQUENCY ACCESS

Multi-Regional

Most projects start with Multi-Regional Storage, which is optimized for **geo redundancy** and **end-user latency.**

Regional

Use Regional Storage when your project requires **higher performance local access** to computing resources — for example, when you need to support **high-frequency analytics workloads.**

LOW FREQUENCY ACCESS

Nearline

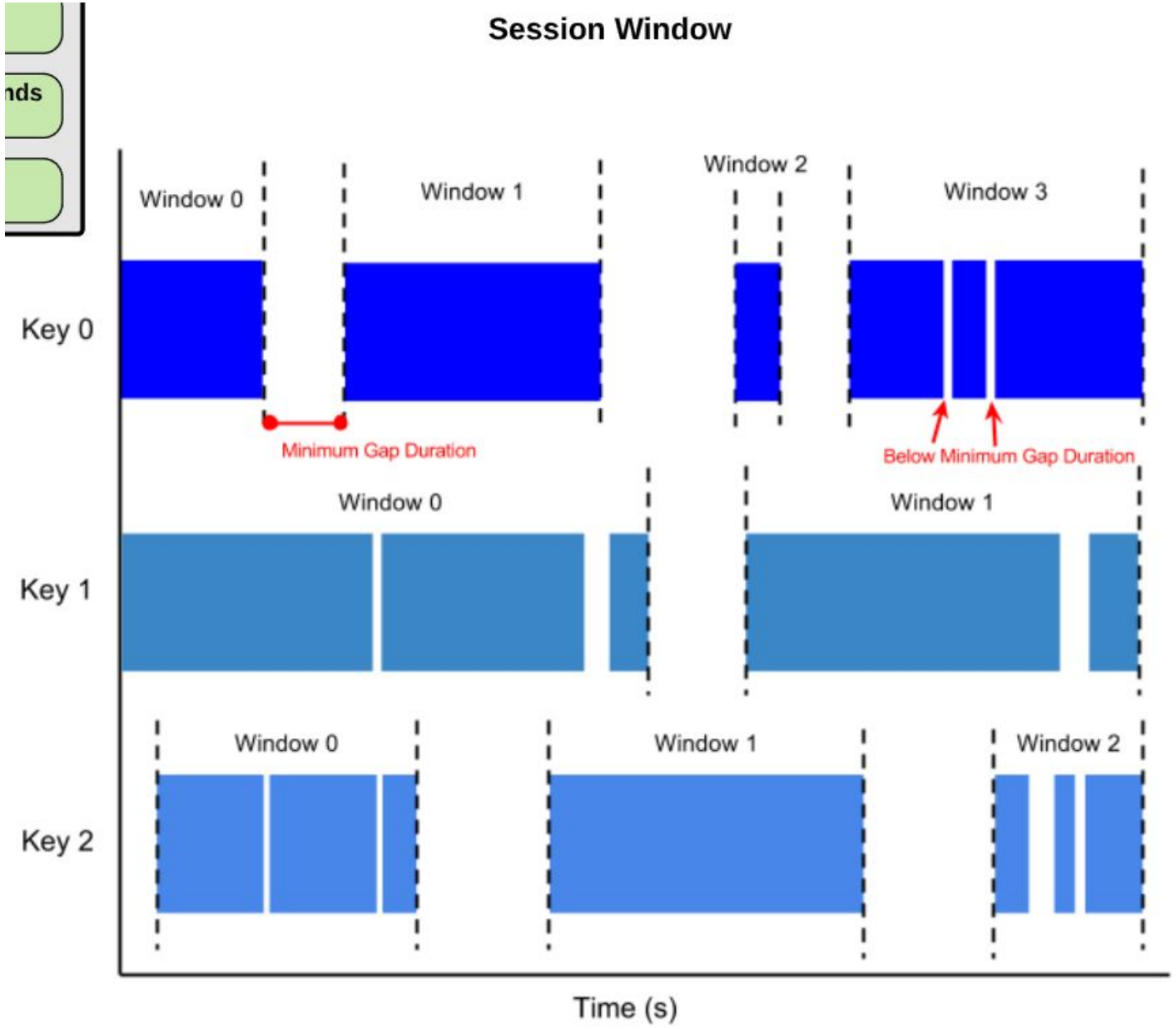
Nearline Storage is fast, highly durable storage for data accessed less than **once a month** .

LOWEST FREQUENCY ACCESS

Coldline

Coldline Storage is fast, highly durable storage for data accessed less than **once a year** .

DataFlow


Session Window
Window 0
Window 1
Window 2
Window 3
Key 0
Minimum Gap Duration
Below Minimum Gap Duration
Window 0
Window 1
Key 1
Window 0
Window 1
Window 2
Key 2
Time (s)

Dataproc Overview

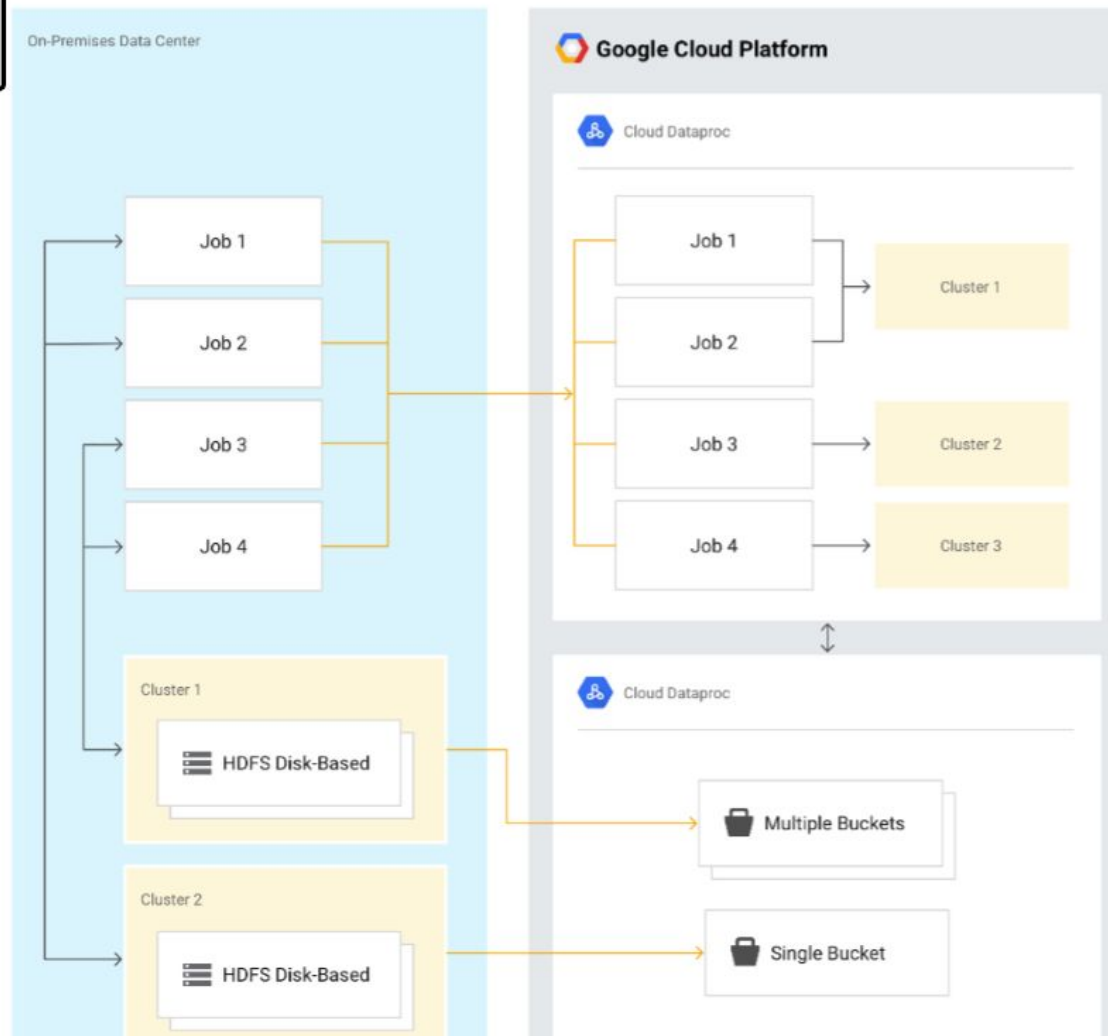
Configure Dataproc Cluster and Submit Job

Migrating and Optimizing for Google Cloud

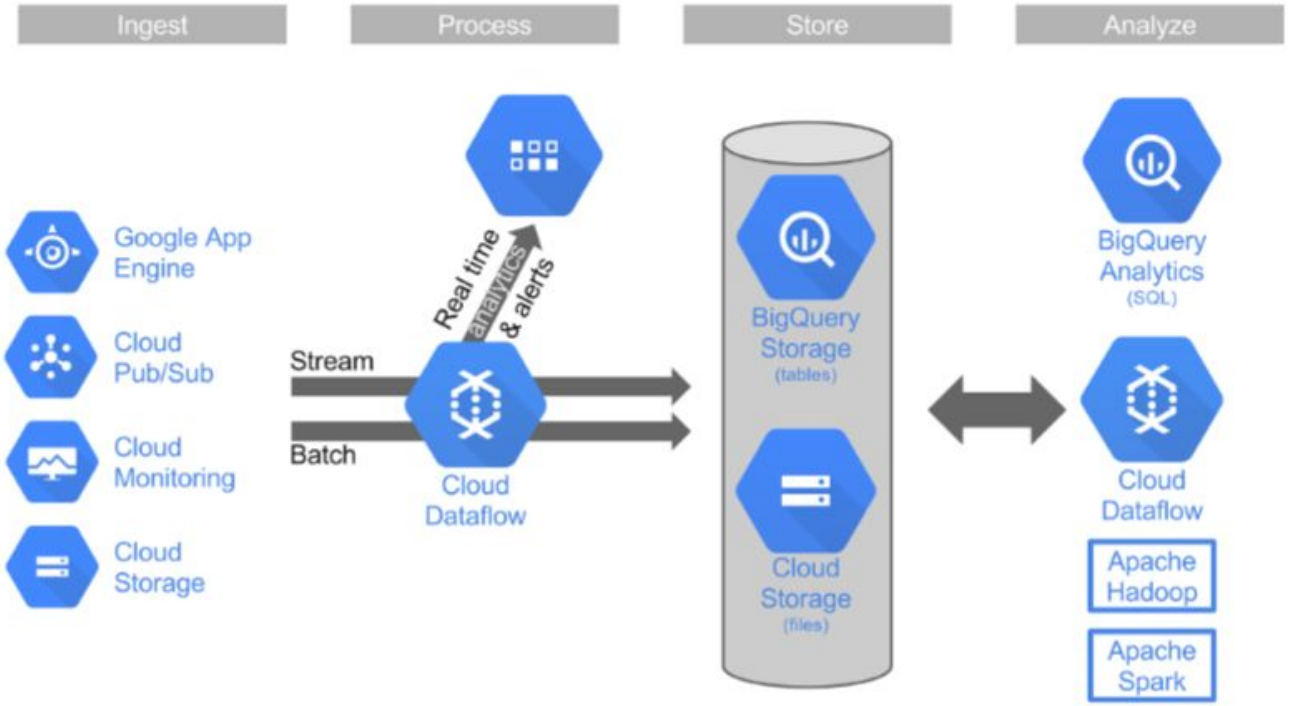
Best Practices for Cluster Performance

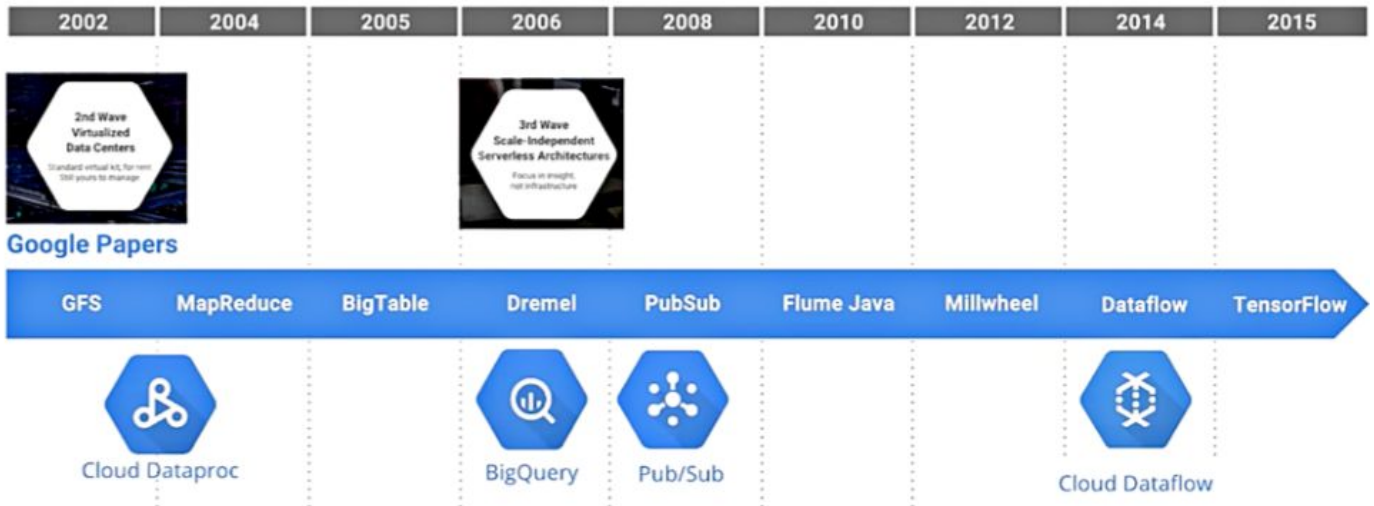
What are we moving/optimizing?

- **Data (from HDFS)**
- **Jobs (pointing to Google Cloud locations)**
- **Treating clusters as ephemeral (temporary) rather than permanent entities**



Install Cloud Storage connector to connect to GCS (Google Cloud Storage).





Pricing

- **Storage, Queries, Streaming insert**
- **Storage = $0.02/GB/mo (first 10GB/mo free)**
  - **Long term storage (not edited for 90 days) = $0.01/GB/mo**
- **Queries = $5/TB (first TB/mo free)**
- **Streaming = $0.01/200 MB**
- **Pay as you go, with high end flat-rate query pricing**
- **Flat rate - starts at $40K per month with 2000 slots**

BigQuery Best Practices

Data Format for Import

Next

- Best performance = Avro format
- Scenario: Import multi-TB databases with millions of rows

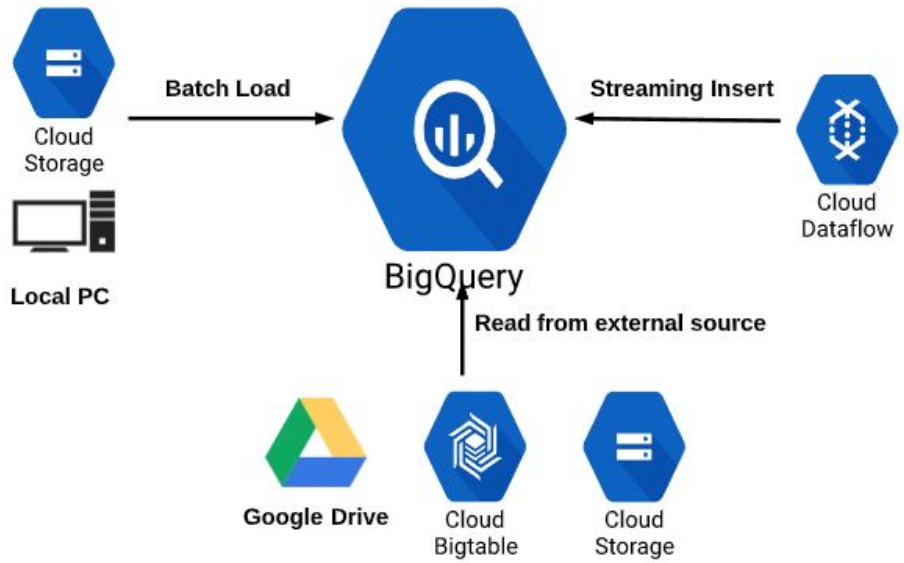
Faster
Avro - Compressed
Avro - Uncompressed
Parquet
CSV
JSON
CSV - Compressed
JSON - Compressed
Slower

Choose a Lesson

- BigQuery Overview
- Interacting with BigQuery
- Load and Export Data
- Optimize for Performance and Costs
- Streaming Insert Example
- BigQuery Logging and Monitoring
- BigQuery Best Practices

Loading and reading sources

Next



Data formats:

Load

- CSV
- JSON (Newline delimited)
- Avro - best for compressed files
- Parquet
- Datastore backups

Read

- CSV
- JSON (Newline delimited)
- Avro
- Parquet

Why use external sources?

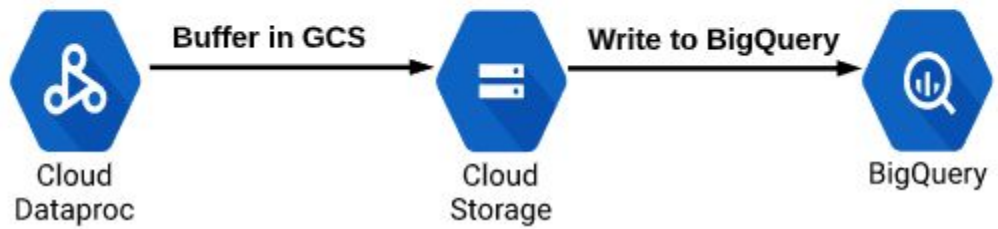
- Load and clean data in one pass from external, then write to BigQuery
- Small amount of frequently changing data to join to other tables

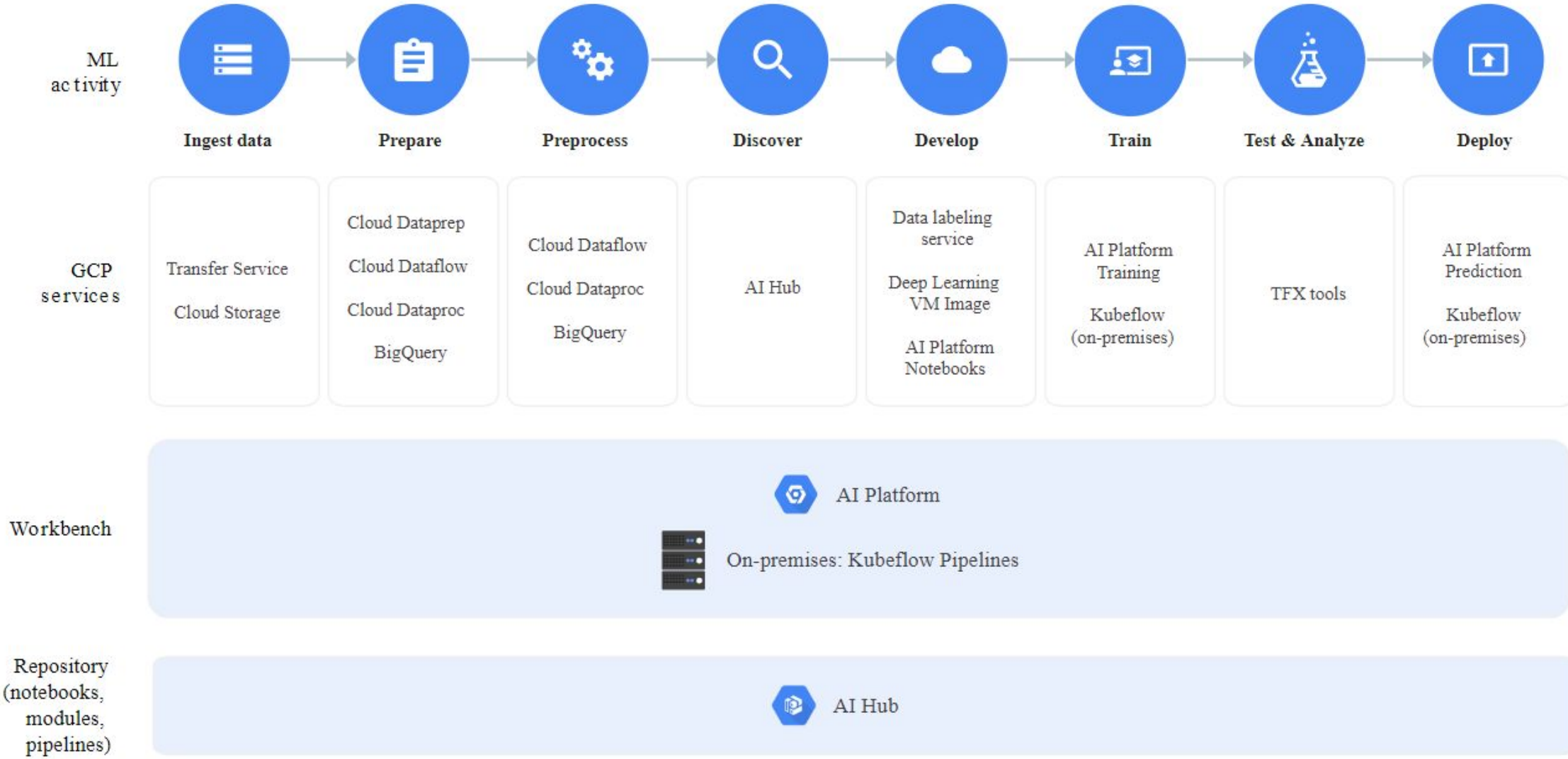
Loading data with command line

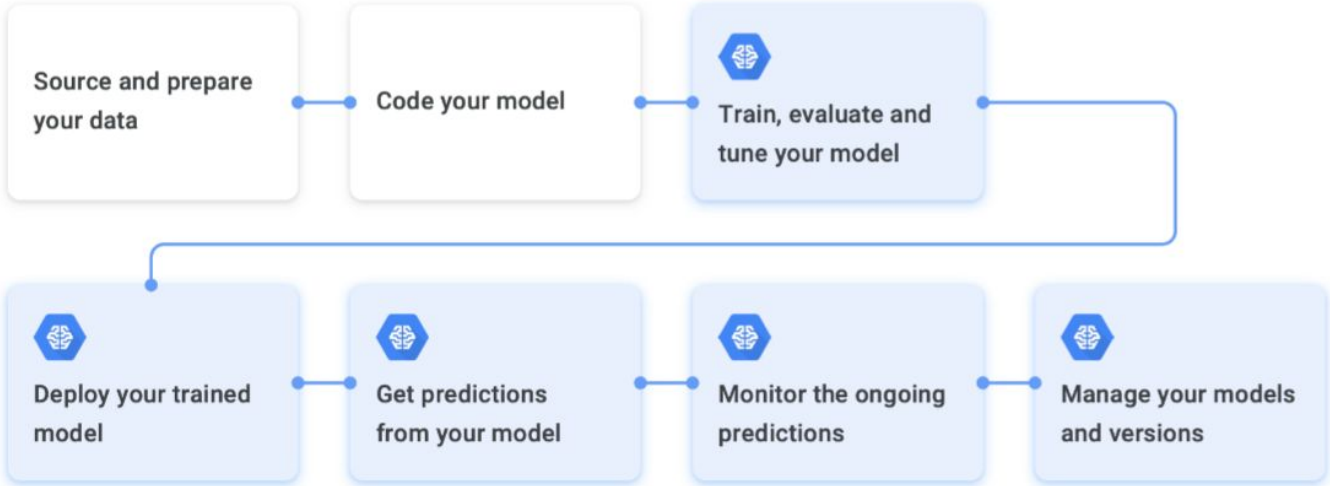
- bq load --source_format=[format] [dataset].[table] [source_path] [schema]
- Can load multiple files with command line (not WebUI)

Connecting to/from other Google Cloud services

- Dataproc - Use BigQuery connector (installed by default), job uses Cloud Storage for staging

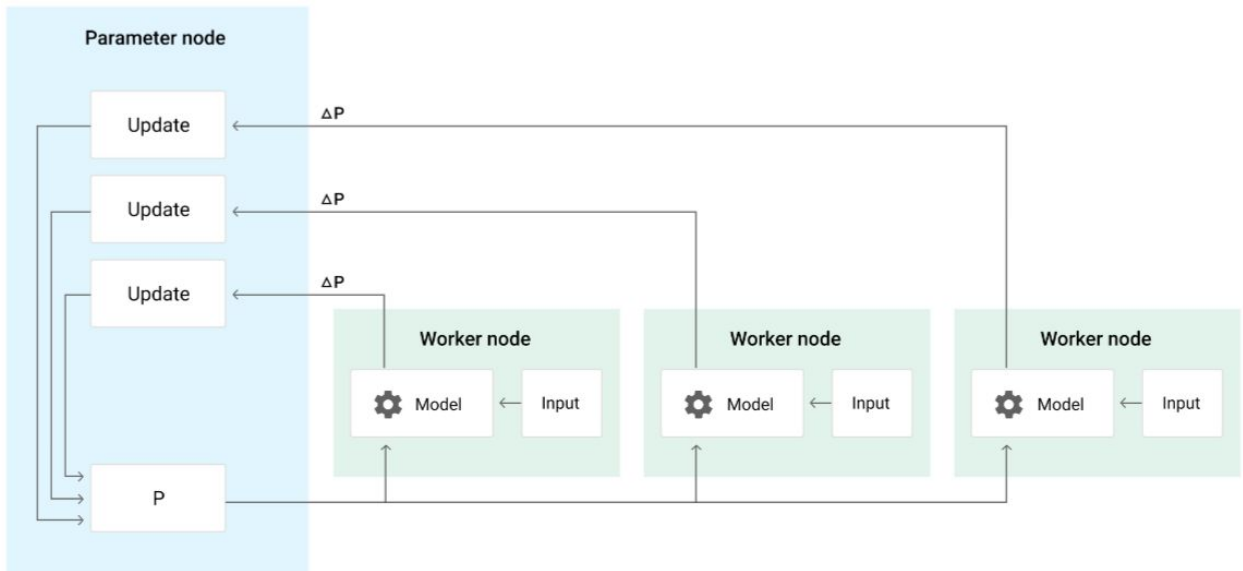



ML activity
Ingest data
Prepare
Preprocess
Discover
Develop
Train
Test & Analyze
Deploy
GCP services
Transfer Service
Cloud Storage
Cloud Dataprep
Cloud Dataflow
Cloud Dataproc
BigQuery
Cloud Dataflow
Cloud Dataproc
BigQuery
AI Hub
Data labeling service
Deep Learning VM Image
AI Platform Notebooks
AI Platform Training
Kubeflow (on-premises)
TFX tools
AI Platform Prediction
Kubeflow (on-premises)
Workbench
AI Platform
On-premises: Kubeflow Pipelines
Repository (notebooks, modules, pipelines)
AI Hub



Get Predictions - two types:

- **Online:**
  - High rate of requests with minimal latency
  - Give job data in JSON request string, predictions returned in its response message
- **Batch:**
  - Get inference (predictions) on large collections of data with minimal job duration
  - Input and output in Cloud Storage



Key Terminology

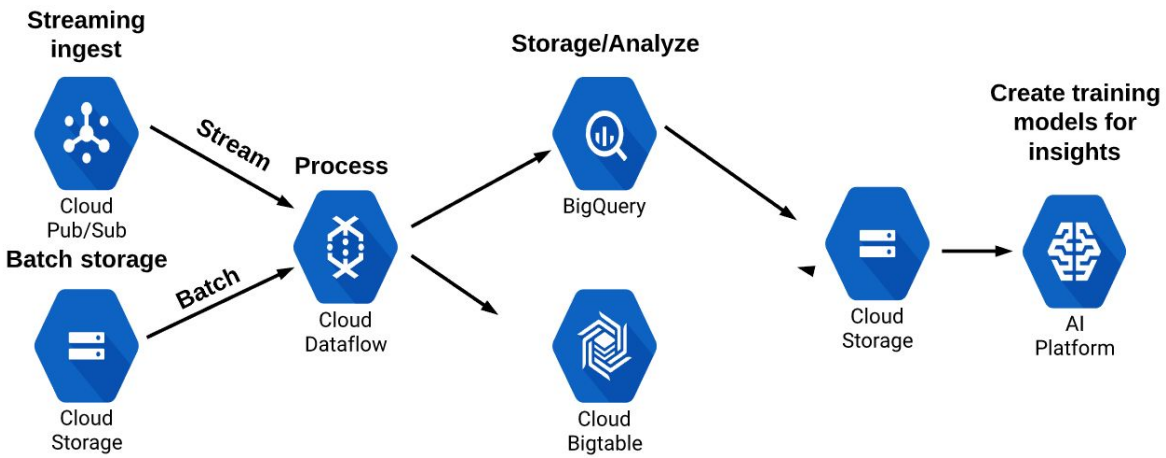
- **Model** - Logical container of individual solutions to a problem:
  - Can deploy multiple versions
  - e.g. Sale price of houses given data on previous sales
- **Version** - Instance of model:
  - e.g. version 1/2/3 of how to predict above sale prices
- **Job** - interactions with AI Platform:
  - Train models:
    - Command = 'submit job train model' on AI Platform
  - Deploy trained models:
    - Command = 'submit job deploy trained model' on AI Platform
  - 'Failed' jobs can be monitored for troubleshooting

IAM roles:

- **Project and Models:**
  - **Admin** - Full control
  - **Developer** - Create training/prediction jobs, models/versions, and send prediction requests
  - **Viewer** - Read-only access to above
- **Models only:**
  - **Model Owner:**
    - Full access to model and versions
  - **Model User:**
    - Read models and use for prediction
    - Easy to share specific models

Using BigQuery for data source:

- Can read directly from BigQuery via training application
- Recommended to pre-process into Cloud Storage
- Using `gcloud` commands, only works with Cloud Storage



What is Dataprep?

How It Works

Backed by Cloud Dataflow:

- After preparing, Dataflow processes via Apache Beam pipeline
- "User-friendly Dataflow pipeline"

Dataprep process:

- Import data
- Transform sampled data with recipes
- Run Dataflow job on transformed dataset
- Export results (GCS, BigQuery)

Intelligent suggestions:

- Selecting data will often automatically give the best suggestion
- Can manually create recipes, however simple tasks (remove outliers, de-duplicate) should use auto-suggestions

IAM:

- Dataprep User - Run Dataprep in a project
- Dataprep Service Agent - Gives Trifecta necessary access to project resources:
  - Access GCS buckets, Dataflow Developer, BigQuery user/data editor
  - Necessary for cross-project access + GCE service account

Pricing:

- 1.16 * cost of Dataflow job

Hands On - Cloud Composer

How It Works

Behind the scenes:

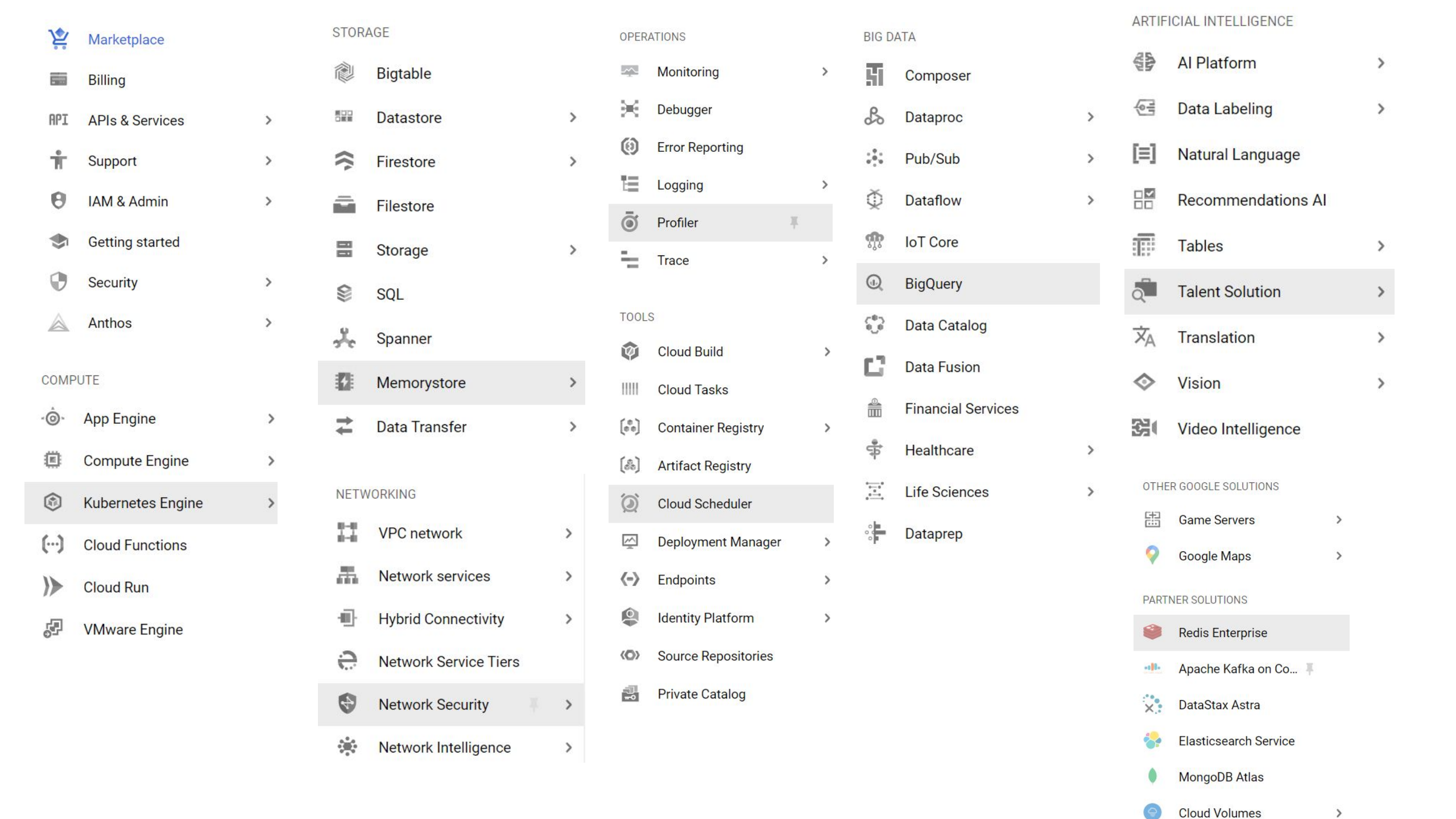
- GKE cluster with Airflow implemented
- Cloud Storage bucket for workflow files (and other application files)

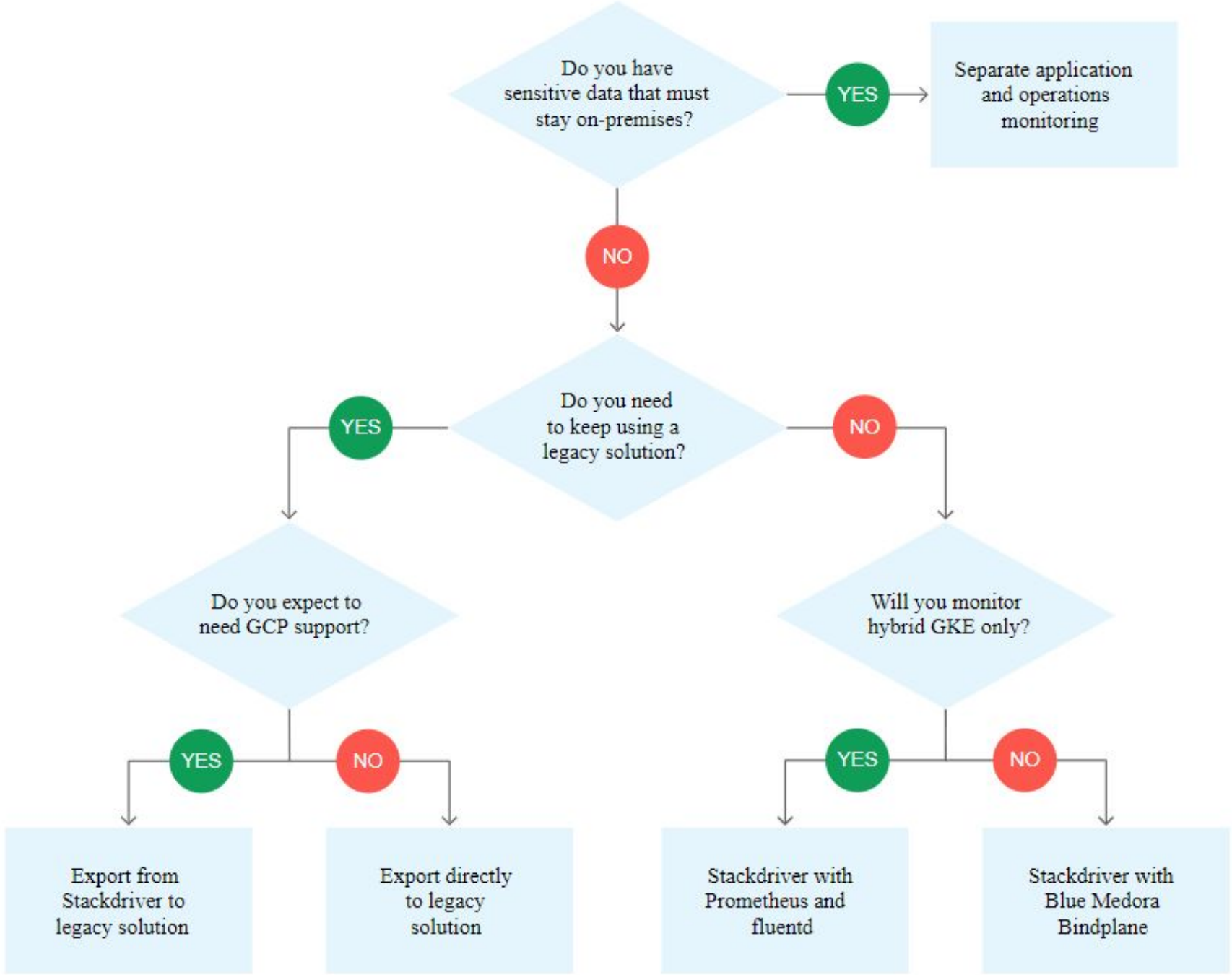
Workflows?

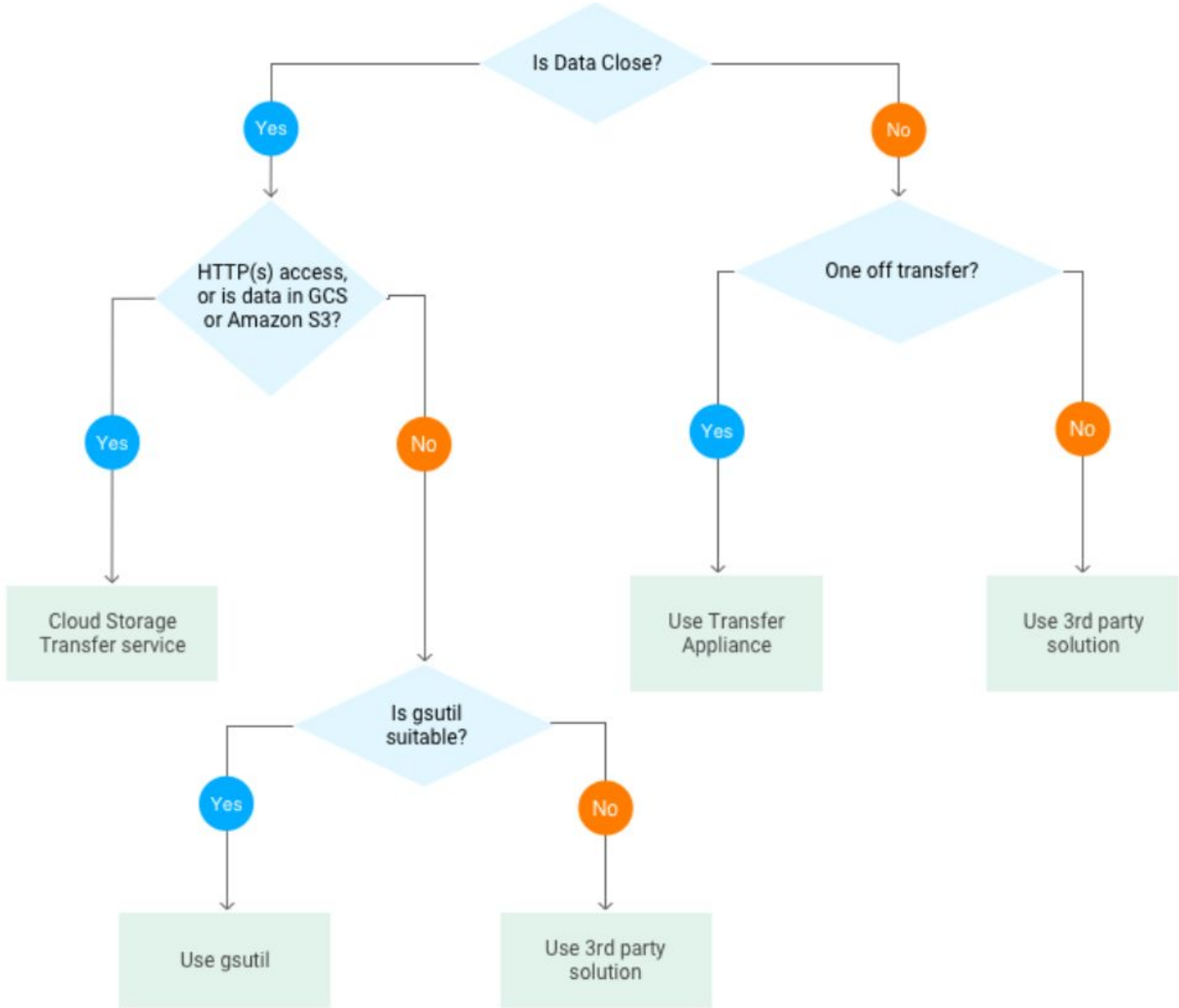
- Orchestrate data pipelines:
  - Like a walkthrough of tasks to run
- Format = Direct Acyclic Graph (DAG):
  - Written in Python
  - Collection of organized tasks that you want to schedule and run
- **Cloud Composer** creates **workflows** using **DAG** files

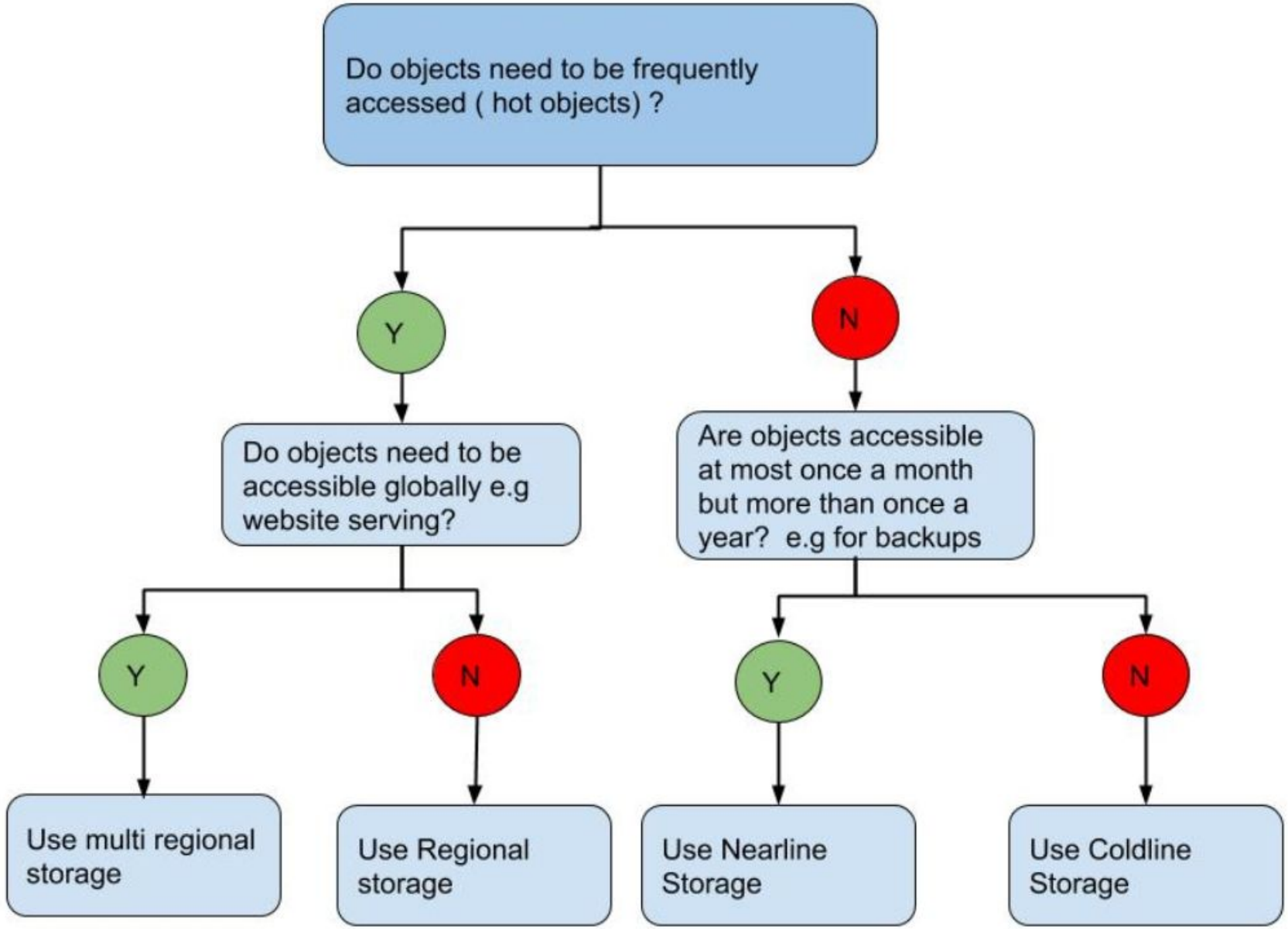
The Process

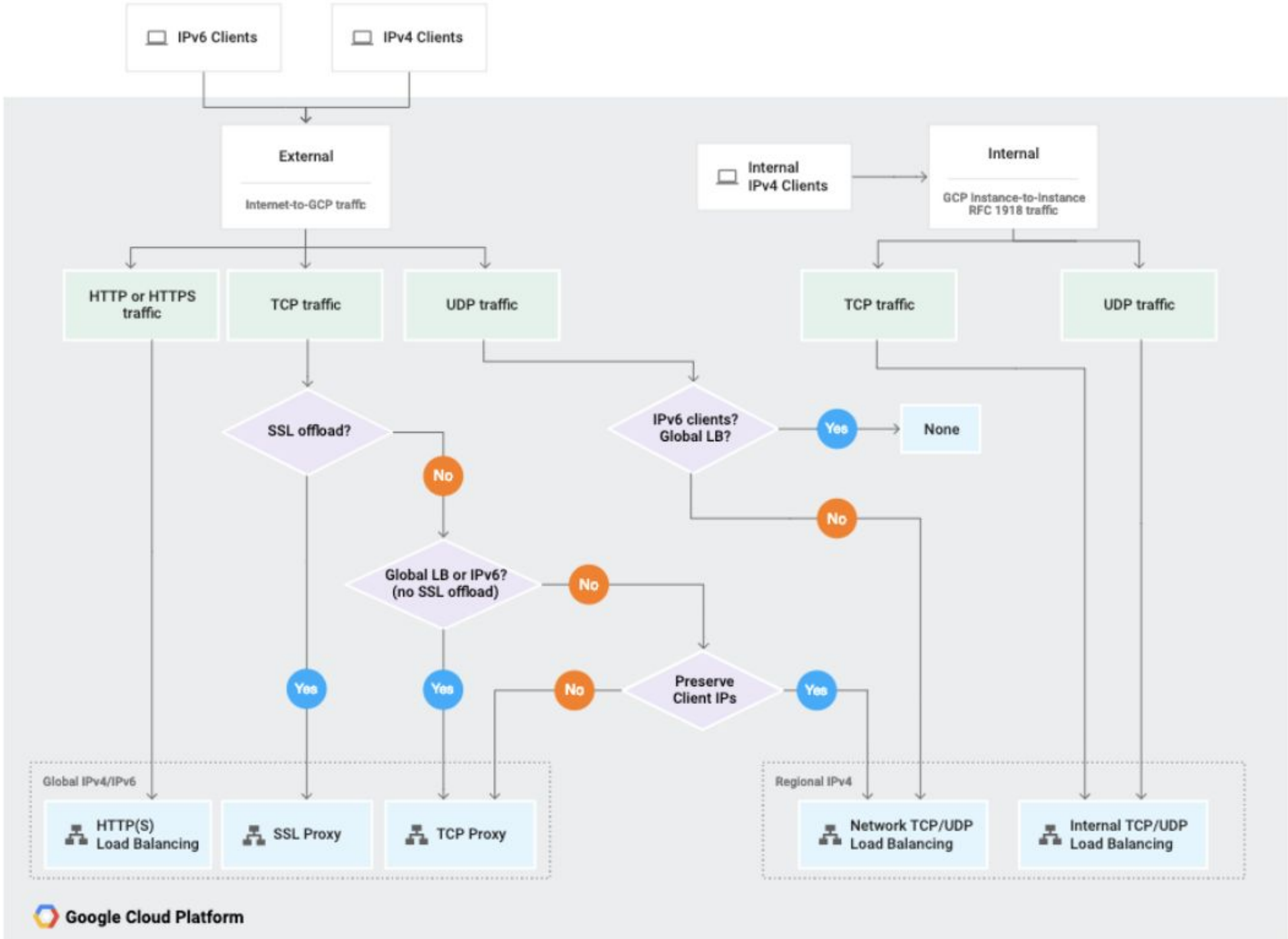
- Create Composer Environment
- Set Composer variables (i.e. project ID, GCS bucket, region)
- Add Workflows (DAG files), which Composer will execute


Marketplace
Billing
APIs & Services
Support
IAM & Admin
Getting started
Security
Anthos
COMPUTE
App Engine
Compute Engine
Kubernetes Engine
Cloud Functions
Cloud Run
VMware Engine
STORAGE
Bigtable
Datastore
Firestore
Filestore
Storage
SQL
Spanner
Memorystore
Data Transfer
NETWORKING
VPC network
Network services
Hybrid Connectivity
Network Service Tiers
Network Security
Network Intelligence
OPERATIONS
Monitoring
Debugger
Error Reporting
Logging
Profiler
Trace
TOOLS
Cloud Build
Cloud Tasks
Container Registry
Artifact Registry
Cloud Scheduler
Deployment Manager
Endpoints
Identity Platform
Source Repositories
Private Catalog
BIG DATA
Composer
Dataproc
Pub/Sub
Dataflow
IoT Core
BigQuery
Data Catalog
Data Fusion
Financial Services
Healthcare
Life Sciences
Dataprep
ARTIFICIAL INTELLIGENCE
AI Platform
Data Labeling
Natural Language
Recommendations AI
Tables
Talent Solution
Translation
Vision
Video Intelligence
OTHER GOOGLE SOLUTIONS
Game Servers
Google Maps
PARTNER SOLUTIONS
Redis Enterprise
Apache Kafka on Co...
DataStax Astra
Elasticsearch Service
MongoDB Atlas
Cloud Volumes


Do you have sensitive data that must stay on-premises?
YES
Separate application and operations monitoring
NO
Do you need to keep using a legacy solution?
YES
NO
Do you expect to need GCP support?
Will you monitor hybrid GKE only?
YES
NO
YES
NO
Export from Stackdriver to legacy solution
Export directly to legacy solution
Stackdriver with Prometheus and fluentd
Stackdriver with Blue Medora Bindplane


Is Data Close?
Yes
No
HTTP(s) access, or is data in GCS or Amazon S3?
One off transfer?
Yes
No
Yes
No
Cloud Storage Transfer service
Use Transfer Appliance
Use 3rd party solution
Is gsutil suitable?
Yes
No
Use gsutil
Use 3rd party solution


Do objects need to be frequently accessed (hot objects) ?
Y
N
Do objects need to be accessible globally e.g website serving?
Are objects accessible at most once a month but more than once a year? e.g for backups
Y
N
Y
N
Use multi regional storage
Use Regional storage
Use Nearline Storage
Use Coldline Storage


IPv6 Clients
IPv4 Clients
External
Internet-to-GCP traffic
Internal
IPv4 Clients
Internal
GCP instance-to-instance
RFC 1918 traffic
HTTP or HTTPS traffic
TCP traffic
UDP traffic
TCP traffic
UDP traffic
SSL offload?
IPv6 clients?
Global LB?
Yes
None
No
No
Global LB or IPv6?
(no SSL offload)
No
Yes
Yes
No
Preserve
Client IPs
Yes
Global IPv4/IPv6
Regional IPv4
HTTP(S)
Load Balancing
SSL Proxy
TCP Proxy
Network TCP/UDP
Load Balancing
Internal TCP/UDP
Load Balancing
Google Cloud Platform

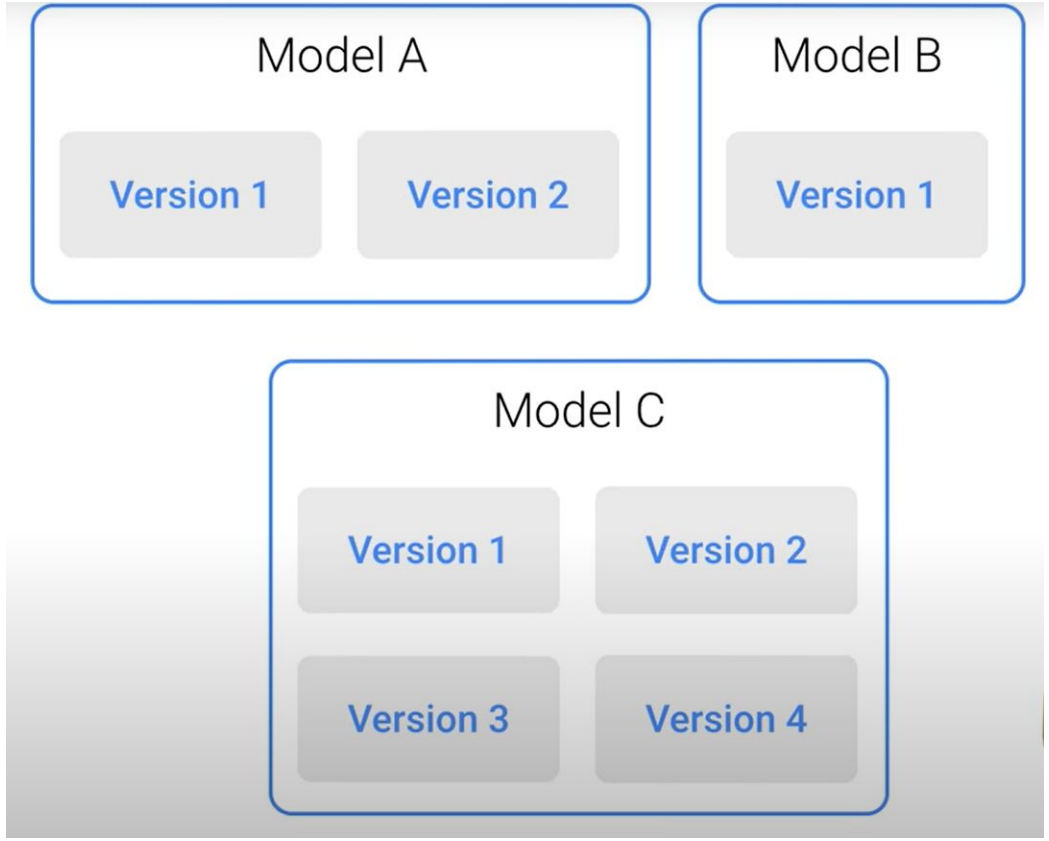
ook the exam today and there were 50 questions. The notes in the blog helped me to focus my study a lot; thanks! Here are my revisions as the exam isn't a beta anymore:

oud Storage and Cloud Datastore – ~2 questions
oud SQL – ~2 questions
gtable – ~8 questions. How to optimize perf or troubleshoot slowdowns. What use cases would fit, etc.
gQuery – ~8 questions. Data partitioning techniques, optimizing performance. Sharing data with other orgs. How to give list priv needed to users. How to avoid costly queries. Loading and differences of availability of data for streaming vs. batch.
b/Sub – ~3 questions. Basic knowledge was enough
pache Hadoop – ~3 questions that were not GCP knowledge but Hadoop ecosystem knowledge; specifically pig and hive and what scenarios would push you to one or the other
oud Dataflow – ~8 questions. Understand batch and streaming designs. How to integrate with BigQuery and constraints you might have.
oud Dataproc – ~3 questions.
nsorFlow, Machine Learning, – ~10 questions that were mostly ML domain (and not TensorFlow specific) basically about training. Nothing about the Cloud ML service!
oud DataLab – ~2 questions about visualization, permissions/restricting access, creating dynamic dashboards
ackdriver – ~1 question about auditing and viewing who did what in BigQuery

On both times I tried, it was heavy on BigQuery (20-25 questions) and TF/ML (10-15 questions), and spread out among the dataproc, dataflow and pubsub, with a couple of questions about SQL and datastore, and another couple of questions about general sysadmin/dba basic stuff. All the questions are scenario simulations where you have to choose which option would be the best way to deal with the situation, according to google standards and documentation. Be it choosing the right tech or the right way to configure it. They focus a lot on BigQuery's interaction with cloud storage and other solutions. What I didnt do much when studying the last two times was exercises and practicing with the solutions.I focused mainly on the theoretical stuff, and really thought I was doing well on my second try, specially since I was comfortably answering the BigQuery questions. But still failed... :(I'm focusing on hands-on practice now, and will try one last time on august.

I have summarized and would like to post what I have felt after I failed the test. Hope that this will help some ones. - The content of the exam covers all the knowledge about Google Cloud Platform (GCP) for Data Engineering, including: Storage (20% of questions), Big Data Processing (35%), Machine Learning (18%), case studies (15%) and others (Hadoop and security about 12%). - GCP Storage (20%): Covering knowledge about Cloud Storage, Cloud SQL, Data Store, Big Table and Big Query. To answer these questions, we need to deeply understand in which situation which storage technology is used to give us the best optimal solution. There are some questions related to design schema for Data Store and Big Table. - Big Data Processing (35%): Covering knowledge about Big Query, Cloud Dataflow, Cloud Dataproc, Cloud Datalab and Cloud Pub/Sub. There are many questions related to Dataproc and Dataflow in my test. In each question, each choice usually is a combination of some GCP technologies in order to create a solution, then we need to choose which solution is the best suitable (in technically, other solutions may be possible but not the best choice). - Machine Learning (18%): Covering knowledge on GCP API (Vision API, Speech API, Natural Language API and Translate API) and Tensorflow. I was a little bit surprised when there were fewer Machine Learning (ML) questions in my test than I expected. Problems and targets of some ML questions are not very clear and I felt vague on selecting the correct answer. - Case Studies (15%): There are 2 case studies which are as same as in the GCP website: a logistic Flowlogistic company and a communications hardware MJTelco company. Each case study includes about 4 questions which ask how to transform current technologies of that company to use GCP technologies. We can learn details about these case studies in LinuxAcademy. - Others (12%): Covering knowledge on Hadoop, and Security Issues. There are some questions that are out of the scope of the GCP Data Engineering, in my opinion, such as questions on Google Cloud Architect and Encryption technology of Security. In my opinion, these are difficult questions for my because I don't have background and GCP architect. To answer these questions, we need to prepare some knowledge in Google Cloud Architect. One notice is that there are some questions where we need to select multiple choices. For example, we need to select 3 answers from 6 choices. In such of case, the first and second choices are usually easier to select than the last choice.

Introduction Priocept consultants have recently been participating in the Google Cloud Platform beta certification exams. We have been working with Google Cloud Platform for many years – since the original launch of Google App Engine – but the new certification scheme allows us to formalize our consultants' expertise on the platform. The "beta" nature of the exams means that our consultants have acted as Google guinea pigs to some degree. Very little study material or practice questions are available at the moment for the certification exams, and you have to rely on prior practical experience and reading the core documentation. So this blog article is intended to give an overview of the content for the Certified Data Engineer exam, as taken by our consultants in January 2017. The Data Engineer certification covers a wide range of subjects including Google Cloud Platform data storage, analytical, machine learning, and data processing products. Below we have given an overview, product-by-product, of what we were subjected to in the exam. Cloud Storage and Cloud Datastore Surprisingly, these products are not covered much in the exam, perhaps because they are covered more extensively in the Cloud Architect exam. Just know the basic concepts of each product and when it is appropriate (or not appropriate) to use each product, and you should be fine. Cloud SQL There were surprisingly few questions on this product in the exam. If you have practical experience using the product, you should be fine to answer any questions that do come up. As with questions related to other data storage products, be sure to know in what scenarios it is appropriate to use Cloud SQL and when it would be more appropriate to use Datastore, Bigquery, Bigtable, etc. Bigtable This product is covered quite extensively in the exam. You should at least know the basic concepts of the product, such as how to design an appropriate schema, how to define a suitable row key, whether Bigtable supports transactions and ACID operations, and you should also know (at least approximately) what the size limits for Bigtable are (cell and row size, maximum number of tables, etc). BigQuery Lots of questions on BigQuery in the Data Engineer exam, as expected. You should know about the basic capabilities of BigQuery and what kind of problem domains it is suitable for. You should also know about BigQuery security and the level at which security can be applied (project and datastore level, but not table or view level). Partitioned tables, table wildcard queries ("backtick" syntax), streaming inserts, query planning and data skew are also covered. You should also have an understanding of the methods available to connect external systems or tools to BigQuery for analytics purposes, how the BigQuery billing model works, and who gets billed when queries cross project and billing account boundaries. Pub/Sub The exam contains lots of questions on this product, but all reasonably high level so it's just important to know the basic concepts (topics, subscriptions, push and pull delivery flows, etc). Most importantly you should know when it is appropriate to introduce Pub/Sub as a messaging layer in an architecture, for a given set of requirements. Apache Hadoop Technically not part of Google Cloud Platform, but there are a few questions around this technology in the exam, since it is the underlying technology for Dataproc. Expect some questions on what HDFS, Hive, Pig, Oozie or Sqoop are, but basic knowledge on what each technology is and when to use it should be sufficient. Cloud Dataflow Lots of questions on this product, which is not surprising as it is a key area of focus for Google with regard to data processing on Google Cloud Platform. In addition to knowing the basic capabilities of the product, you will also need to understand concepts like windowing types, triggers, PCollections, etc. Cloud Dataproc Not many questions on this besides the Hadoop questions mentioned above. Just be sure to understand the differences between Dataproc and Dataflow and when to use one or the other. Dataflow is typically preferred for a new development, whereas Dataproc would be required if migrating existing on-premise Hadoop or Spark infrastructure to Google Cloud Platform without redevelopment effort. TensorFlow, Machine Learning, Cloud DataLab The exam contains a significant amount of questions on this – more than we were expecting. Fortunately we have been busy working with TensorFlow and Cloud Datalab at Priocept for a while now. You should understand all the basic concepts of designing and developing a machine learning solution on TensorFlow, including concepts such data correlation analysis in Datalab, and overfitting and how to correct it. Detailed TensorFlow or Cloud ML programming knowledge is not required but a good understanding of machine learning design and implementation is important. Stackdriver A surprising numbers of questions on this, given that Stackdriver is more of an "ops" product than a "data engineering" product. Be sure to know the sub-products of Stackdriver (Debugger, Error Reporting, Alerting, Trace, Logging), what they do and when they should be used. Conclusion The Data Engineer certification exam is a fair assessment of the skills required if you want to be able to demonstrate the ability to work effectively with Google Cloud Platform on analytics, big data, data processing, or machine learning projects. If you have used the majority of these products already on real-world products, the exam should not present you with too many problems. If you haven't yet used some of the products above, then get studying! If you take the exam and get caught out in any areas that we haven't covered above, please let us know. Priocept provides both consultancy and bespoke training services for Google Cloud Platform, so please get in touch if we can help your organisation on your journey towards adopting the platform.



Small to Medium Datasets

```
$ gsutil -m cp <source> <destination>
```

Run copy command in parallel

Large Datasets

Connection type	Time to upload 1 Petabyte
100 mbps	~3 years
1000 mbps	~4 months
Transfer appliance	~25 *hours*

What is TensorFlow Serving?

- C++ Libraries
  - TensorFlow model save / export formats
  - Generic core platform
- Binaries
  - Best practices out of the box
  - Docker containers, K8s tutorial
- Hosted Service

kubernetes

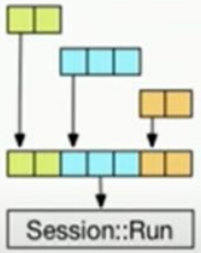
docker

Google Cloud Platform

```
// Serve a model on a model server binary

// Build the model server
$> bazel build //tensorflow_serving/model_servers:tensorflow_model_server

// Run the server with a single model
$> bazel-bin/tensorflow_serving/model_servers/tensorflow_model_server
--port=9000 --model_name=video-rank --model_base_path=/tmp/video-rank/

// Run the server with a config file containing one or more models
$> bazel-bin/tensorflow_serving/model_servers/tensorflow_model_server
--port=9000 --model_config_file=/tmp/my_config
```

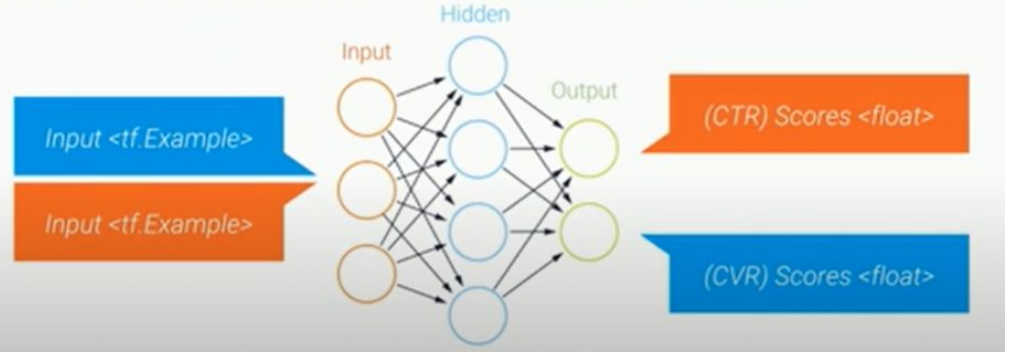
Batching

- Achieve the performance of mini-batching
- Enables efficient use of GPUs & TPUs; e.g. 25x speedup moving to batched GPU evaluation [nyu.edu/projects/bowman/spinn.pdf]
- Multiplexing across multiple models & versions: ../batching/shared_batch_scheduler.h
- Available in both libraries & binaries



Multi-headed Inference

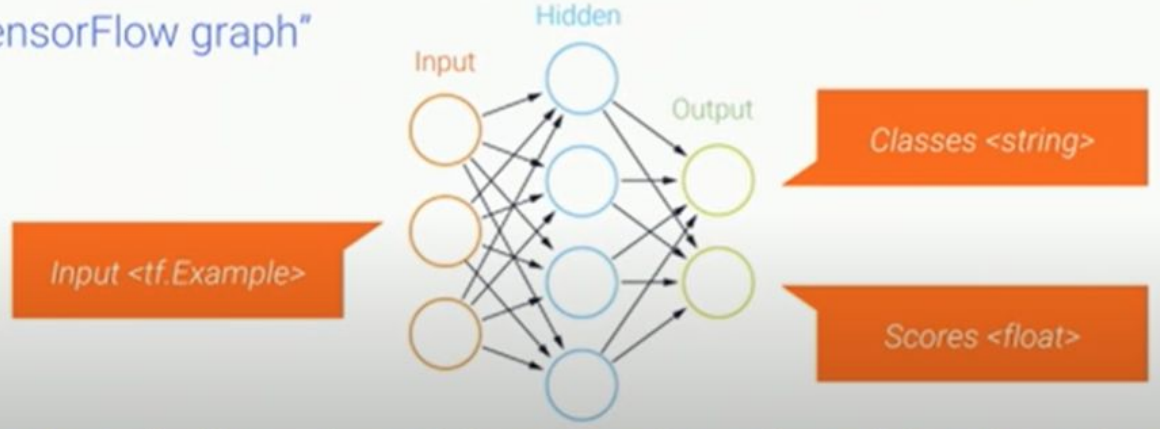
Multiple SignatureDefs for multiple "heads" of a model.



Inputs are sent only once, saving bandwidth, cpu, and latency.

SavedModel: SignatureDef

SignatureDef "defines the signature of a computation supported by a TensorFlow graph"



core/protobuf/saved_model.proto, cc/saved_model, py/saved_model

Inference APIs

. Predict

. Regress

. Classify

. MultiInference

../apis/prediction_service.proto

```
// Multi Headed Inference Request.
MultiInferenceRequest {
  tasks {
    model_spec {
      name: "video-rank"
      signature_name: "pctr"
    }
    model_spec {
      name: "video-rank"
      signature_name: "pcvr"
    }
  }
  input ...
```

Cloud Machine Learning Engine

- Step 1: Packaging your Python Code
- Step 2: Configuration file
- Step 3: Submit training job to the cloud

Step 3: Submit training job to the cloud

```
$ gcloud ml-engine jobs submit training
  job-id <unique ID>
  package-path=trainer
  module-path=trainer.task
  region=us-central-1
  job_dir=gs://cloudml-demo/iris-demo/
```

Prediction via REST

```
{
   "requests": [
      {
         "image": {
            "source": {
               "gcsImageUri":
                  "gs://YOUR_BUCKET/YOUR_REMOTE_IMAGE_FILE"
            }
         }
         "features": [
            {"type": "CUSTOM_LABEL_DETECTION", "maxResults": 10 },
            {"type": "LABEL_DETECTION", "maxResults": 10}
         ],
         "customLabelDetectionModels":
            "projects/cloudml-demo/models/chairs_table_bike/
versions/chairs_table_bike_201804
         }
      ]
}
```

Calling Cloud ML Engine Online Prediction

```
gcloud beta ml-engine predict \
 --model zoo_sklearn_model \
 --version version1 \
 --json-instances input_4.json
```

Comparing built-in algorithms

The following table provides a quick comparison of the built-in algorithms:

Algorithm name	ML model used	Type of problem	Example use case(s)	Supported accelerators for training
Linear learner	TensorFlow Estimator LinearClassifier and LinearRegressor.	Classification, regression	Sales forecasting	GPU
Wide and deep	TensorFlow Estimator DNNLinearCombinedClassifier, DNNLinearCombinedEstimator, and DNNLinearCombinedRegressor.	Classification, regression, ranking	Recommendation systems, search	GPU
XGBoost	XGBoost	Classification, regression	Advertising click-through rate (CTR) prediction	None (CPU only)
Image classification	TensorFlow image classification models	Classification	Classifying images	CPU, GPU, TPU
Object detection	TensorFlow Object Detection API	Object detection	Detecting objects within complex image scenes	CPU, GPU, TPU

Limitations

Please note the following limitations for training with built-in algorithms:

- Distributed training is not supported. To run a distributed training job on AI Platform Training, you must create a training application.
- Training jobs submitted through the Google Cloud Console use only legacy machine types. You can use Compute Engine machine types with training jobs submitted through `gcloud` or the Google APIs Client Library for Python. Learn more about machine types for training.
- GPUs are supported for some algorithms. Refer to the detailed comparison of all the built-in algorithms for more information.
- Multi-GPU machines do not yield greater speed with built-in algorithm training. If you're using GPUs, select machines with a single GPU.
- TPUs are not supported for tabular built-in algorithm training. You must create a training application. Learn how to run a training job with TPUs.

Any further limitations for specific built-in algorithms are noted in the corresponding guides for each algorithm.

REST API

The AI Platform REST API provides RESTful services for managing jobs, models, and versions, and for making predictions with hosted models on Google Cloud.

You can use the Google APIs Client Library for Python to access the APIs. When using the client library, you use Python representations of the resources and objects used by the API. This is easier and requires less code than working directly with HTTP requests.

We recommend the REST API for serving online predictions in particular.

Interpretability

Google is working intensely to advance the areas of AI interpretability and accountability, through open-sourcing tools and publishing research. For instance:

- **Tensor Flow Lattice:** enabling anyone to train flexible models that capture a priori knowledge about whether an input should only have a positive effect on an output[2]
- **Tensor Flow Debugger:** enabling developers to look inside models during training[3]
- **Building blocks of interpretability:** illustrating how different techniques can be combined to provide powerful interfaces for explaining neural network outputs[4]

Privacy

Google has long supported efforts in the research and development of privacy and anonymity techniques for AI systems, including publishing new open-source code as privacy-protection best practices. For example:

- **Our open-source RAPPOR technology:** deployed worldwide as the first large-scale data-collection mechanism with strong protection guarantees of local differential privacy[8]
- **Secure aggregation protocol for federated learning model updates:** provides strong cryptographic privacy for individual user's updates, averaging only updates from large groups of participants[9]

Security

One of the biggest threats to AI systems currently comes from "adversarial attacks", in which bad actors fool the system by making very small, not human detectable, changes to model inputs. Fortunately such attacks are very difficult and hence not (yet) widespread, but Google researchers are at the forefront of tackling them. Publicly released research includes:

- **Adversarial Logit Pairing (ALP):** state of the art in defenses against adversarial examples[5]
- **Ensemble Adversarial Training:** the previous state of the art (before ALP) in defenses against black-box adversarial examples, developed in collaboration with Stanford[6]
- **CleverHans:** a machine learning security research library maintained by Google team[7]

Fairness

AI systems are shaped by what their training data leaves out and what it over-represents. Human biases within the data, model design, and methods for training and testing can lead to outcomes that affect different groups of people differently. Addressing these disparate outcomes is a primary goal in the emerging research area of fairness in machine learning. Google is an active contributor to this field, including in the provision of developer tools. For example:

- **Facets:** interactive visualization tool that lets developers see a holistic picture of their training data at different granularities[10]
- **Mmd-critic:** exploratory data analysis tool that looks for statistical minorities in the data[11]

Google AI principles

- **AI should be socially beneficial:** with the likely benefit to people and society substantially exceeding the foreseeable risks and downsides
- **AI should not create or reinforce unfair bias:** avoiding unjust impacts on people, particularly those related to sensitive characteristics such as race, ethnicity, gender, nationality, income, sexual orientation, ability and political or religious belief
- **AI should be built and tested for safety:** designed to be appropriately cautious and in accordance with best practices in AI safety research, including testing in constrained environments and monitoring as appropriate
- **AI should be accountable to people:** providing appropriate opportunities for feedback, relevant explanations and appeal, and subject to appropriate human direction and control
- **AI should incorporate privacy design principles:** encouraging architectures with privacy safeguards, and providing appropriate transparency and control over the use of data
- **AI development should uphold high standards of scientific excellence:** Technological innovation is rooted in the scientific method and a commitment to open inquiry, intellectual rigor, integrity, and collaboration
- **AI should be made available for uses that accord with these principles:** We will work to limit potentially harmful or abusive applications

https://github.com/tensorflow/docs/blob/master/site/en/r1/guide/feature_columns.md

Passing feature columns to Estimators

As the following list indicates, not all Estimators permit all types of `feature_columns` argument(s):

- `tf.estimator.LinearClassifier` and `tf.estimator.LinearRegressor`: Accept all types of feature column.
- `tf.estimator.DNNClassifier` and `tf.estimator.DNNRegressor`: Only accept dense columns. Other column types must be wrapped in either an `indicator_column` or `embedding_column`.
- `tf.estimator.DNNLinearCombinedClassifier` and `tf.estimator.DNNLinearCombinedRegressor`:
  - The `linear_feature_columns` argument accepts any feature column type.
  - The `dnn_feature_columns` argument only accepts dense columns.

Demonstrate several types of feature columns

- Numeric columns
- Bucketized columns
- Categorical columns
- Embedding columns
- Hashed feature columns
- Crossed feature columns

BigQuery ML increases the speed of model development and innovation by removing the need to export data from the data warehouse. Instead, BigQuery ML brings ML to the data. The need to export and reformat data has the following disadvantages:

- Increases complexity because multiple tools are required.
- Reduces speed because moving and formatting large amounts data for Python-based ML frameworks takes longer than model training in BigQuery.
- Requires multiple steps to export data from the warehouse, restricting the ability to experiment on your data.
- Can be prevented by legal restrictions such as HIPAA guidelines.

Comparing feature attribution methods

AI Explanations offers three methods to use for feature attributions: *sampled Shapley*, *integrated gradients*, and *XRAI*.

Method	Basic explanation	Recommended model types	Example use cases
Integrated gradients	A gradients-based method to efficiently compute feature attributions with the same axiomatic properties as the Shapley value.	Differentiable models, such as neural networks. Recommended especially for models with large feature spaces. Recommended for low-contrast images, such as X-rays.	• Classification and regression on tabular data • Classification on image data
XRAI (eXplanation with Ranked Area Integrals)	Based on the integrated gradients method, XRAI assesses overlapping regions of the image to create a saliency map, which highlights relevant regions of the image rather than pixels.	Models that accept image inputs. Recommended especially for *natural images*, which are any real-world scenes that contain multiple objects.	• Classification on image data
Sampled Shapley	Assigns credit for the outcome to each feature, and considers different permutations of the features. This method provides a sampling approximation of exact Shapley values.	Non-differentiable models, such as ensembles of trees and neural networks[1]	• Classification and regression on tabular data

Advantages and use cases

If you inspect specific instances, and also aggregate feature attributions across your training dataset, you can get deeper insight into how your model works. Consider the following advantages and use cases:

- **Debugging models**: Feature attributions can help detect issues in the data that standard model evaluation techniques would usually miss. For example, an image pathology model achieved suspiciously good results on a test dataset of chest X-Ray images. Feature attributions revealed that the model's high accuracy depended on the radiologist's pen marks in the image.
- **Optimizing models**: You can identify and remove features that are less important, which can result in more efficient models.

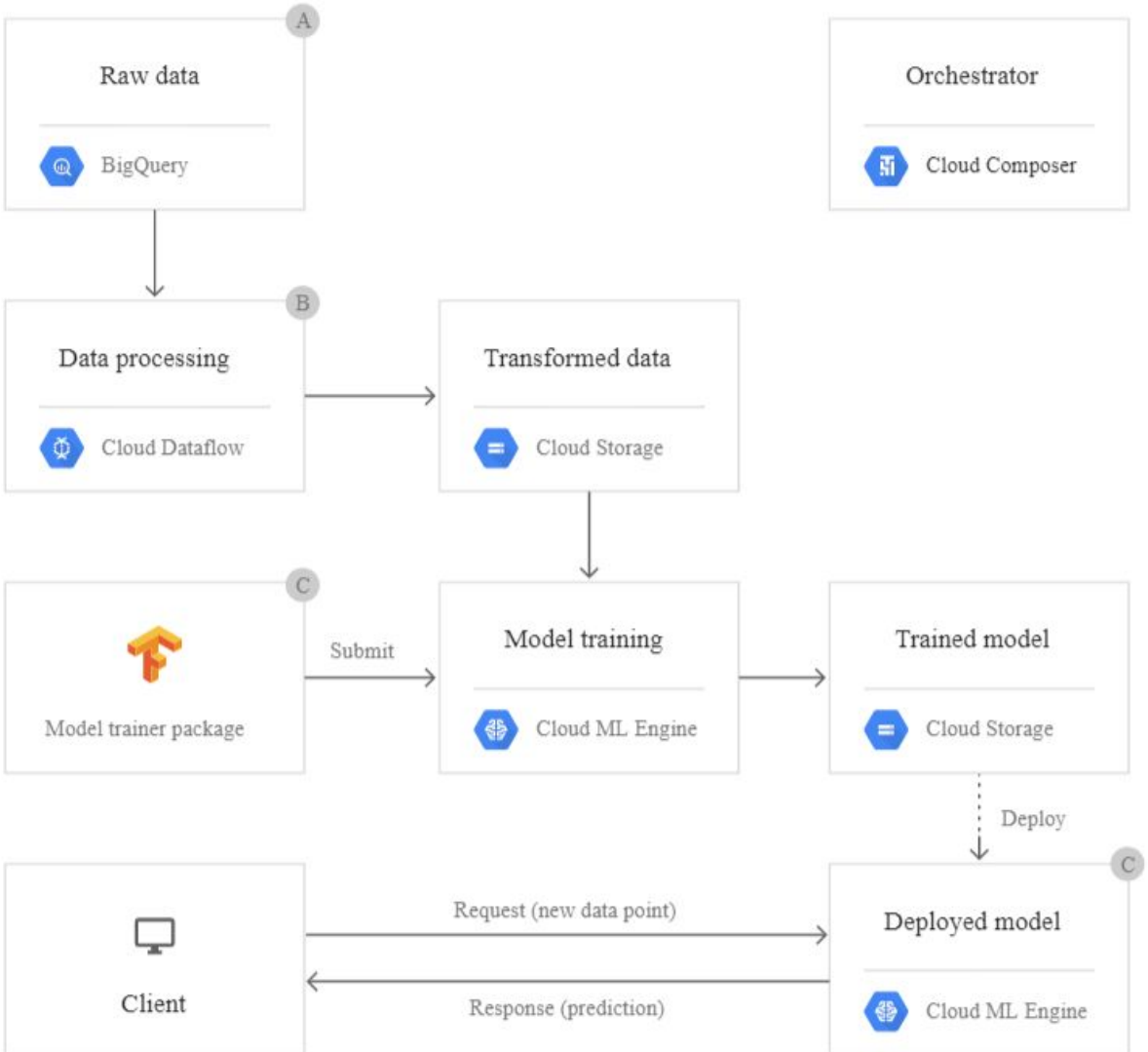
Service limitations

Consider the following limitations of the AI Explanations feature attributions in AI Platform Predictior

- The explanations service currently supports only models trained in TensorFlow 1.x.
- If you're using Keras, do **not** use `tf.saved_model.save` to save your model. Instead, use `tf.keras.estimator.model_to_estimator`.

Considerations for using local feature importance:

- Local feature importance results are available only for models trained on or after 15 November, 2019.
- Enabling local feature importance on a batch prediction request with more than 1,000,000 rows or 300 columns is not supported.
- Each local feature importance value shows only how much the feature affected the prediction for that row. To understand the overall behavior of the model, use model feature importance.
- Local feature importance values are always relative to the baseline value. Make sure you reference the baseline value when you are evaluating your local feature importance results. The baseline value is available only from the Cloud Console.
- The local feature importance values depend entirely on the model and data used to train the model. They can tell only the patterns the model found in the data, and can't detect any fundamental relationships in the data. So, the presence of a high feature importance for a certain feature does not demonstrate a relationship between that feature and the target; it merely shows that the model is using the feature in its predictions.
- If a prediction includes data that falls completely outside of the range of the training data, local feature importance might not provide meaningful results.
- Generating feature importance increases the time and compute resources required for your prediction. In addition, your request uses a different quota than for prediction requests without feature importance. Learn more.
- Feature importance values alone do not tell you if your model is fair, unbiased, or of sound quality. You should carefully evaluate your training dataset, procedure, and evaluation metrics, in addition to feature importance.



In other typical architectures, the client app directly calls the deployed model API for online predictions (as shown earlier in Figure 2). In that case, if preprocessing operations are implemented in Dataflow to prepare the training data, these operations are not applied to the prediction data going directly to the model. Thus, transformations like these should be an integral part of the model during serving for online predictions.

Option A: BigQuery

Typically, the logic for the following operations is implemented in BigQuery:

- Sampling: randomly selecting a subset from the data.
- Filtering: removing irrelevant or invalid instances.
- Partitioning: splitting the data to produce training, evaluation, and test sets.

BigQuery SQL scripts can be used as a source query for the Dataflow preprocessing pipeline. This is the data processing step in Figure 2. For example, imagine that a system will be used in Canada, but the data warehouse has transactions from around the world. Filtering to get Canadian-only training data is best done in BigQuery.

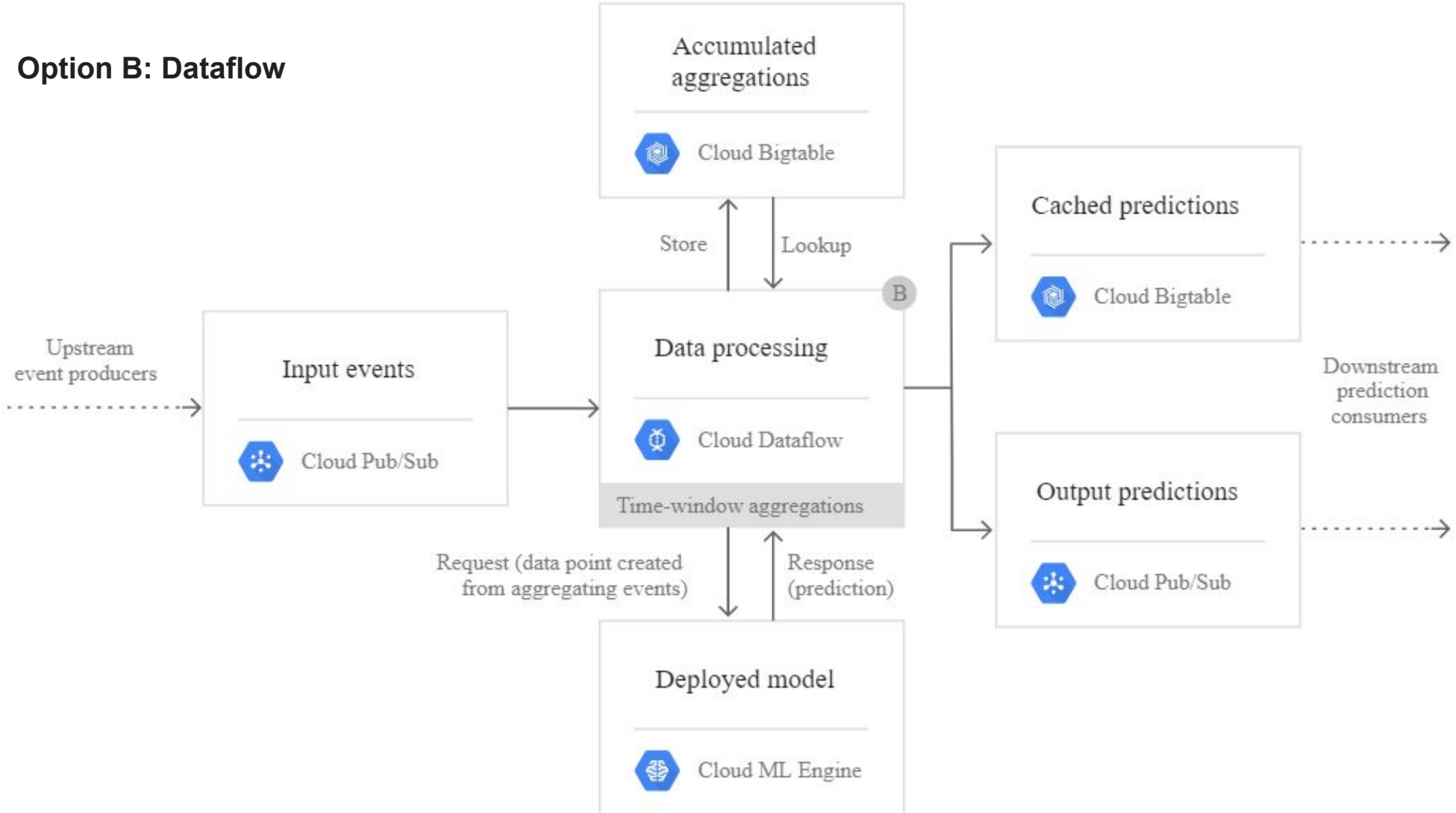
It is possible to implement instance-level transformations, stateful full-pass transformations, and window aggregations feature transformations in a BigQuery SQL script to prepare the training data. However, this is not recommended.

If you are deploying the model for online predictions, you have to replicate the SQL preprocessing operations on the raw data points that you generated from other systems and that will be deployed model's API. In other words, you need to implement the logic twice. The first time is in SQL to preprocess training data in BigQuery. The other time is in the logic of the app that consumes the model to preprocess online data points for prediction. For example, if your client app is written in Java, you need to reimplement the logic in Java. This can introduce errors due to implementation discrepancies. (See the discussion of training/serving skew in the preprocessing challenges section later in this document.)

In addition, maintaining two different implementations is extra overhead. Whenever you change the logic in SQL to preprocess the training data, you need to change the Java implementation accordingly to preprocess data at serving time.

If you are using your model only for batch prediction (scoring) using AI Platform batch prediction, and if your data for scoring is sourced from BigQuery, it is feasible to implement these preprocessing operations as part of the BigQuery SQL script. In that case, you can use the same preprocessing SQL script to prepare both training and scoring data. This implies that you are using auxiliary tables to store quantities needed by stateful transformations, such as means and variances to scale numerical features. It also means increased complexity in the SQL scripts, and intricate dependency between training and the scoring SQL scripts.

Option B: Dataflow



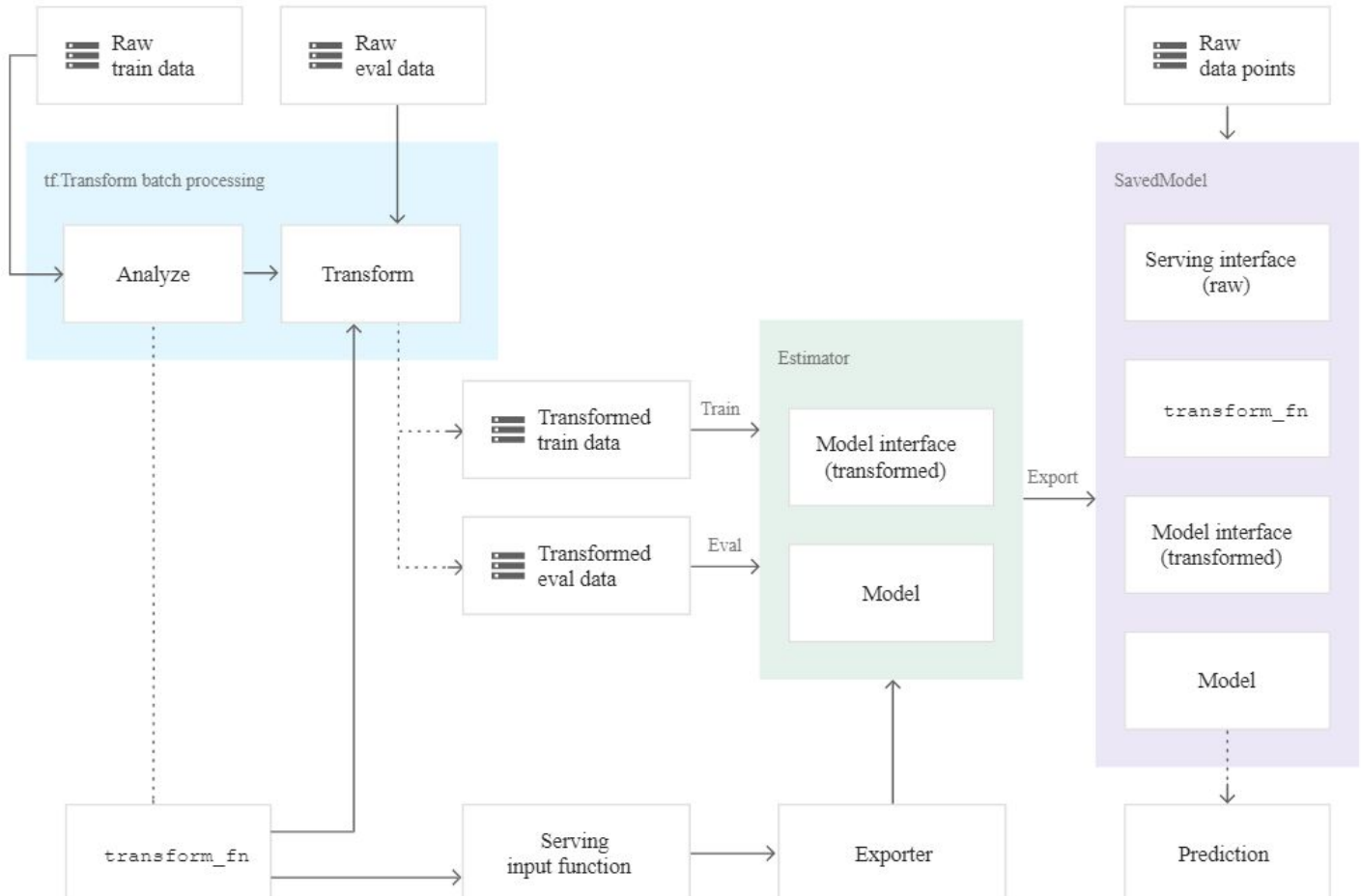
The same Apache Beam implementation can be used to batch-process training data that comes from an offline datastore like BigQuery and stream-process real-time data for serving online predictions.

- **Preparing the data up front for better training efficiency**. Implementing instance-level transformations as part of the model can affect the efficiency of the training process. Imagine that you have raw training data with 1000 features, and you apply a mix of instance-level transformations to generate 10,000 features. If you implement these transformation as part of your model, and if you then feed the model the raw training data, these 10,000 operations are applied *N* times on each instance, where *N* is the number of epochs. On top of that, if you are using accelerators (GPUs or TPUs), they sit idle while the CPU is performing those transformations, which is not an efficient use of your costly accelerators.

  Ideally, the training data is transformed before training using the technique described under Option B: Dataflow, where the 10,000 transformation operations are applied only once on each training instance. The transformed training data is then presented to the model. No further transformations are applied, and the accelerators are busy all of the time. In addition, using Dataflow helps preprocessing large amount of data at scale, using a fully managed service.

You can see how preparing the training data up front can improve training efficiency. However, implementing the transformation logic outside of the model (the approaches described under Option A: BigQuery or Option B: Dataflow) does not resolve the issue of training-serving skew. This transformation logic would need to be implemented somewhere to be applied on new data points coming for prediction, because the model interface is now expecting transformed data. The TensorFlow Transform (`tf.Transform`) library can address this issue.

The following diagram shows `tf.Transform` preprocessing and transforming data for training and prediction.



The `tf.Transform` library is useful for transformations that require a full pass. The output of `tf.Transform` is exported as a TensorFlow graph that represents the instance-level transformation logic, as well as the statistic computed from full-pass transformations, to use for training and serving. Using the same graph for both training and serving can prevent skew, because the same transformations are applied in both stages. In addition, `tf.Transform` can run at scale in a batch processing pipeline on Dataflow to prepare the training data up front and improve training efficiency.

	Instance-level (stateless transformations)		Full-pass during training instance-level during serving (stateful transformations)		Real-time (window) aggregations during training and serving (streaming transformations)	
	Batch scoring	**Online prediction**	**Batch scoring**	**Online prediction**	**Batch scoring**	**Online prediction**
BigQuery (SQL)	**OK** The same transformation implementation is applied on data during training and batch scoring.	**Possible** to process training data but **not recommended** Results in training/serving skew, because you process serving data using different tools.	**Possible** to use statistics computed using BigQuery for instance-level batch/online transformations. Not easy; you must maintain a stats store to be populated during training and used during prediction.		N/A Aggregates like these computed based on real-time events.	**Possible** to process training data but **not recommended**. Results in training/serving skew, because you process serving data using different tools.
Dataflow (Apache Beam)		**OK** if data at serving time comes from Pub/Sub to be consumed by Dataflow. Otherwise, results in training/serving skew.	**Possible** to use statistics computed using Dataflow for instance-level batch/online transformations. Not easy; you must maintain a stats store to be populated during training and used during prediction.			**OK** The same Apache Beam transformation is applied on data during training (batch) and serving (stream).
Dataflow (Apache Beam + TFT)	**OK** The same transformation implementation is applied to data during training and batch scoring.	**Recommended** Avoids training/serving skew and prepares training data up front.	**Recommended** Transformation logic + computed statistics during training are stored as a `tf.Graph` that's attached to the exported model for serving.			
TensorFlow* (`input_fn` & `serving_input_fn`)	**Possible** but **not recommended** For training & prediction efficiency, it's better to prepare the training data up front.	**Possible** but not **recommended** For training efficiency, it's better to prepare the training data up front.	**Not Possible**			**Not Possible**

* Transformations such as crossing, embedding, and one-hot encoding should be performed declaratively as feature_columns.

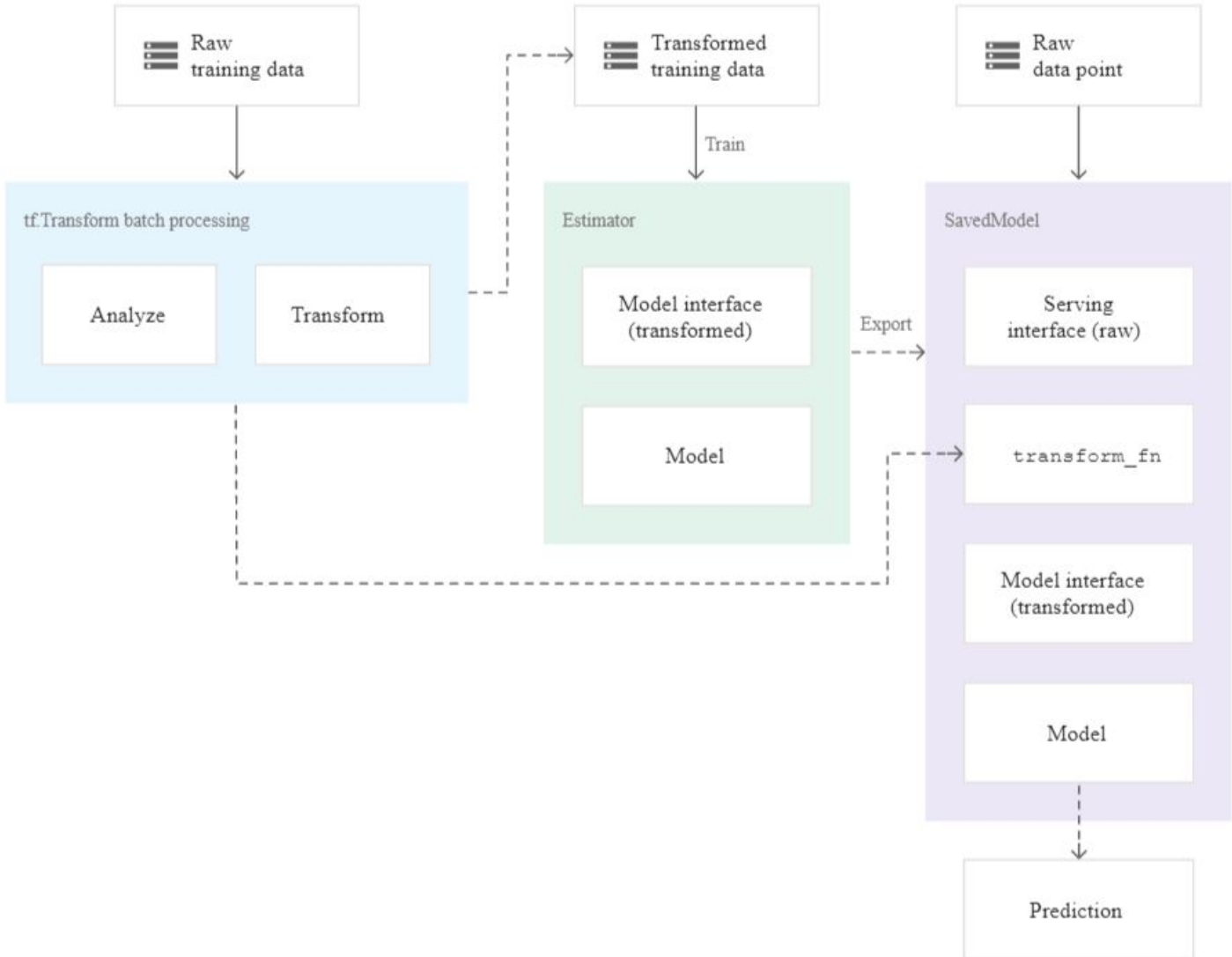


Figure 1. The behavior of `tf.Transform`

1. Read training data from BigQuery.
2. Analyze and transform training data using `tf.Transform`.
3. Write transformed training data to Cloud Storage as `tfRecords`.
4. Read evaluation data from BigQuery.
5. Transform evaluation data using the `transform_fn` produced by step 2.
6. Write transformed training data to Cloud Storage as `tfrecords`.
7. Write transformation artifacts to Cloud Storage for creating and exporting the model.

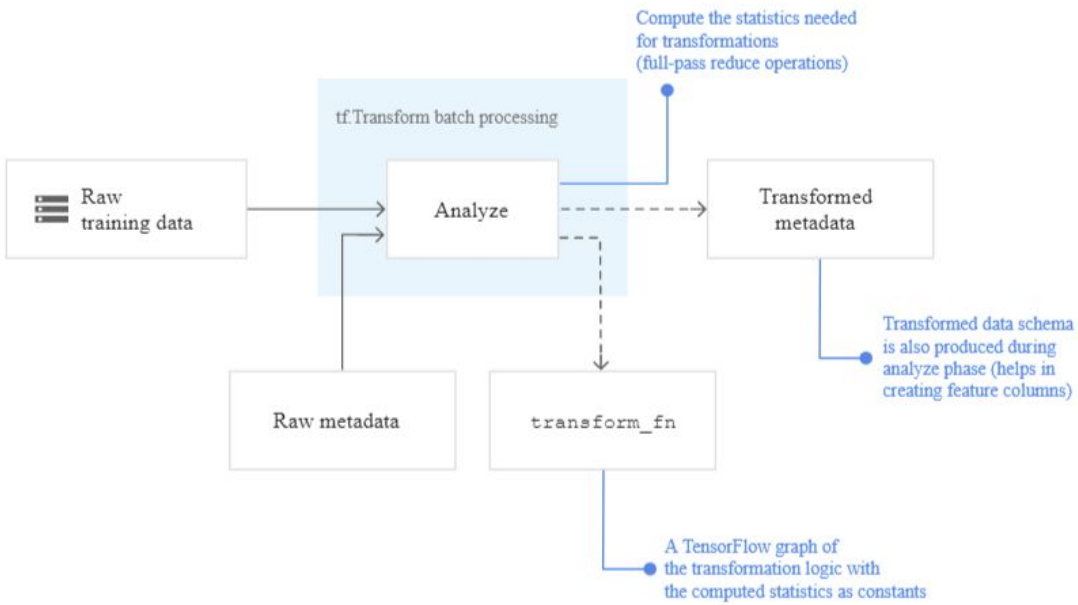


Figure 2. The `tf.Transform` analyze phase

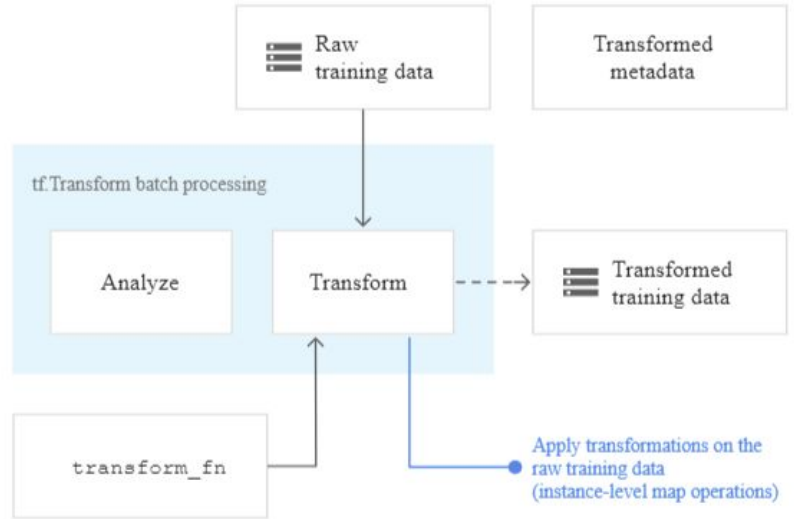


Figure 3. The `tf.Transform` transform phase

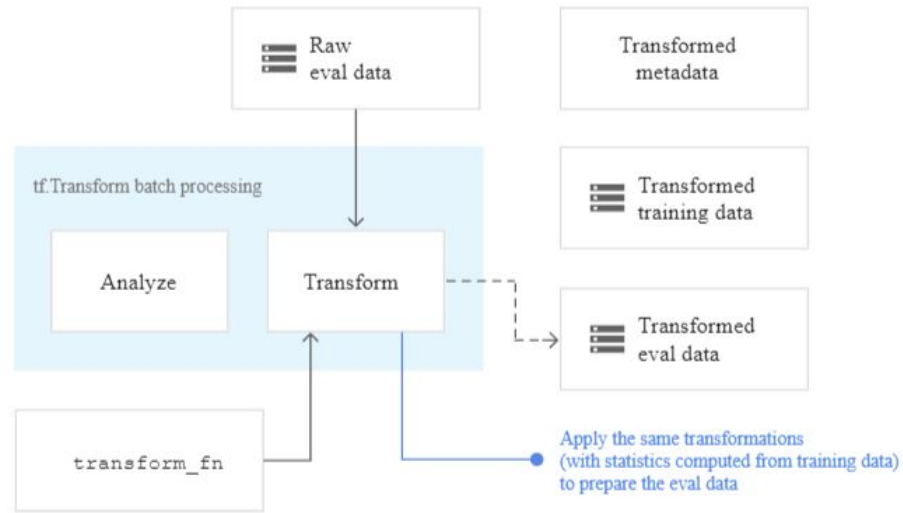
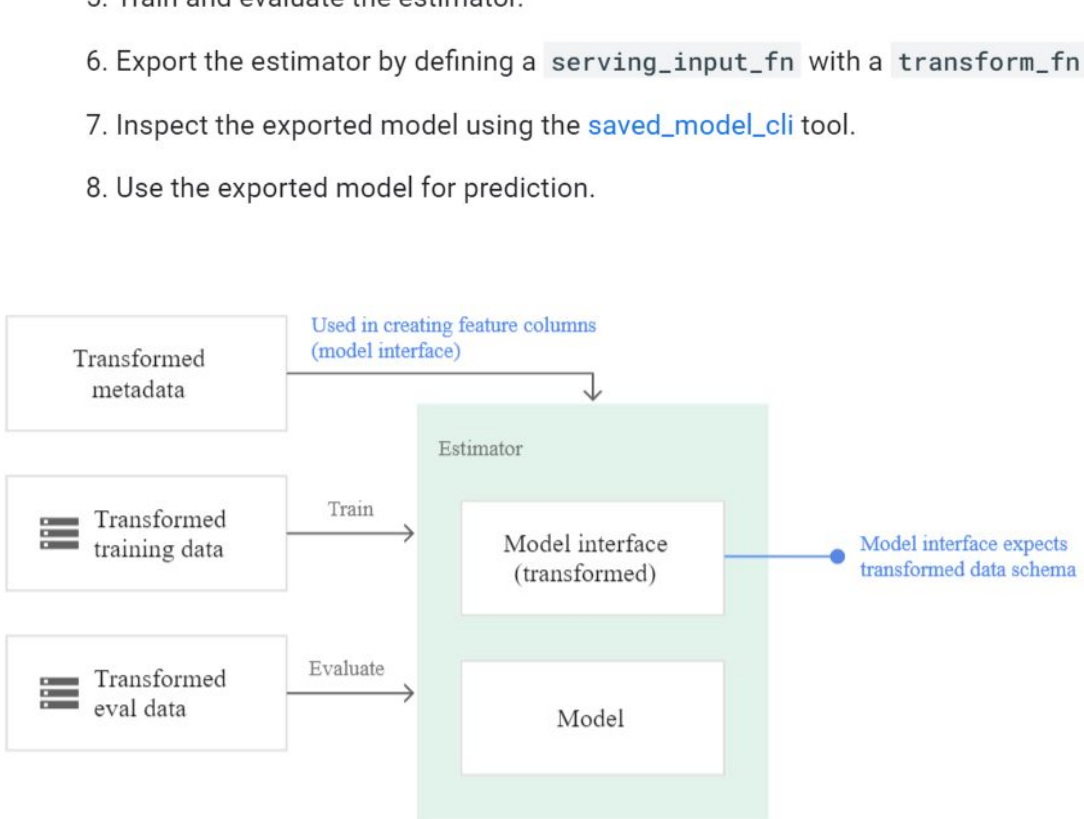
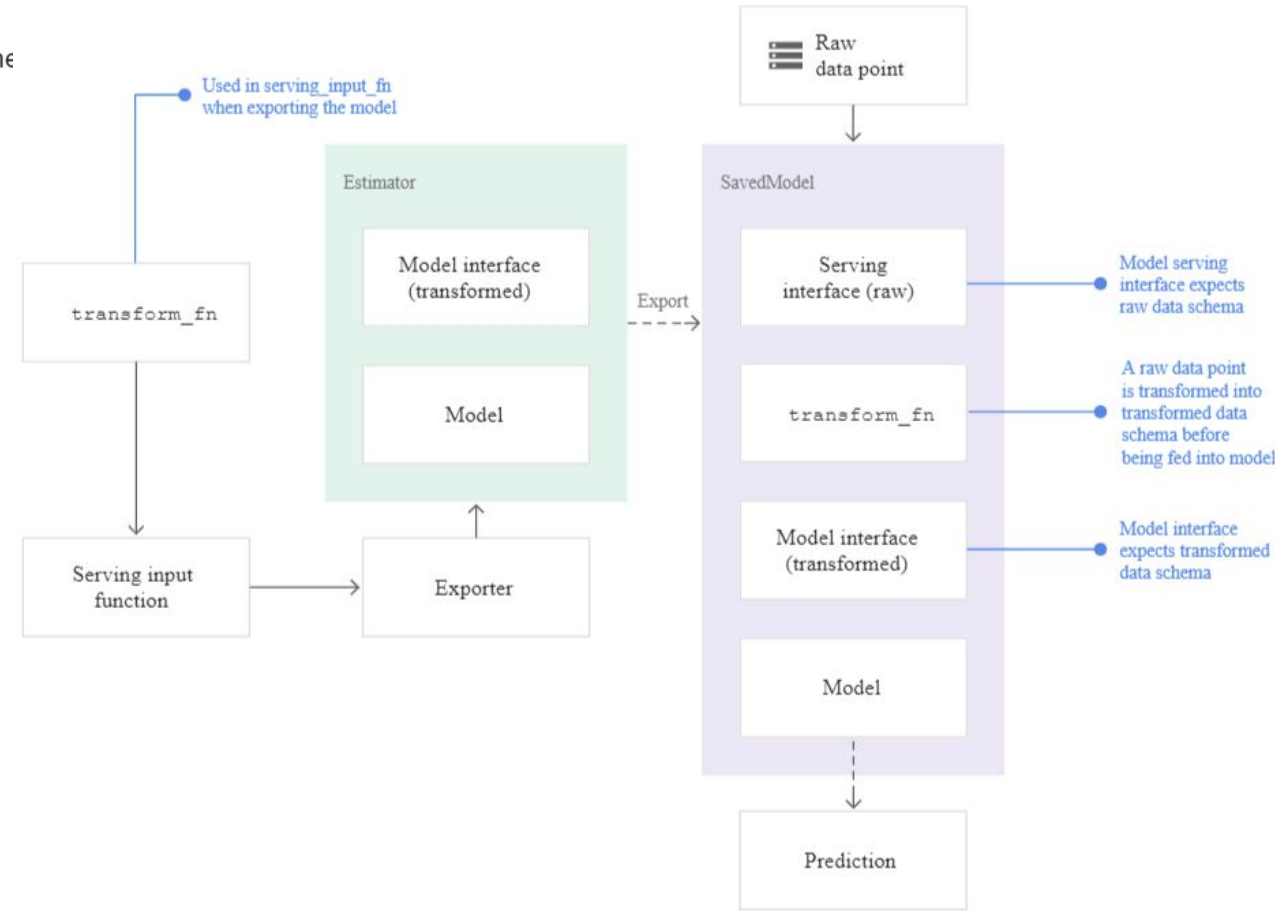


Figure 4. Transforming evaluation data using `transform_fn`

The steps for creating the model are as follows:

1. Load the `transform_metadata` object.
2. Create an `input_fn` to read and parse the training and evaluation data using the `transform_metadata` object.
3. Create feature columns using the `transform_metadata` object.
4. Create the `DNNLinearCombinedRegressor` estimator with the `feature_columns`.
5. Train and evaluate the estimator.
6. Export the estimator by defining a `serving_input_fn` with a `transform_fn` attache
7. Inspect the exported model using the saved_model_cli tool.
8. Use the exported model for prediction.





https://cloud.google.com/blog/products/ai-machine-learning/how-to-serve-deep-learning-models-using-tensorflow-2-0-with-cloud-functions

Feature	Compute Engine	AI Platform Predictions	Cloud Functions
ML frameworks Installation	Pre-loaded TensorFlow and other frameworks when using Deep Learning Images or Deep Learning Containers	Pre-loaded TensorFlow and other frameworks	Installation of libraries is required Libraries are installed via *requirements.txt*
Configuration	Minimal	Simple via UI, CLI or YAML file	Requires developing server code
Infrastructure management	Required	Not required	Not required
Scalability	Scalable infrastructure	Scalable infrastructure via configuration	Scalable infrastructure
Framework support	Latest ML frameworks supported	Pre-defined ML versions supported for inference	Any ML framework for inference
Environment	Production	Production	Experimentation

Cost	Compute Engine Custom machine type	Cloud Function
Cost per 1 GB RAM	0.004446 per hour	0.009 per hour
Cost per 1 vCPU	0.033174 per hour	0.072 per hour

As we can see, Compute Engine pricing per hour looks extremely attractive (with utilizing preemptible instances or commitment use, the price will be even lower), but the catch here is that cost of instance is per one second, with a minimum of one minute. Cloud Functions, on the other hand, have billing per 100ms without a minimum time period. This means Cloud Functions are great for short, inconsistent jobs, but if you need to handle a long, consistent stream of jobs, Compute Engine might be a better choice.

First of all, keep in mind differences between ***cold invocation***, when the function needs time to download and initialize the model, and warm invocation, when function uses a cached model. Increasing the ratio of warm invocations to cold invocations not only allows you to increase processing speed, it also decreases the cost of your inference. There are multiple ways you can increase the ratio. One way could be warming up functions so they will be warm when a high load comes in. Another way is to use Pub/Sub to normalize a load so that it will be processed by warm containers.

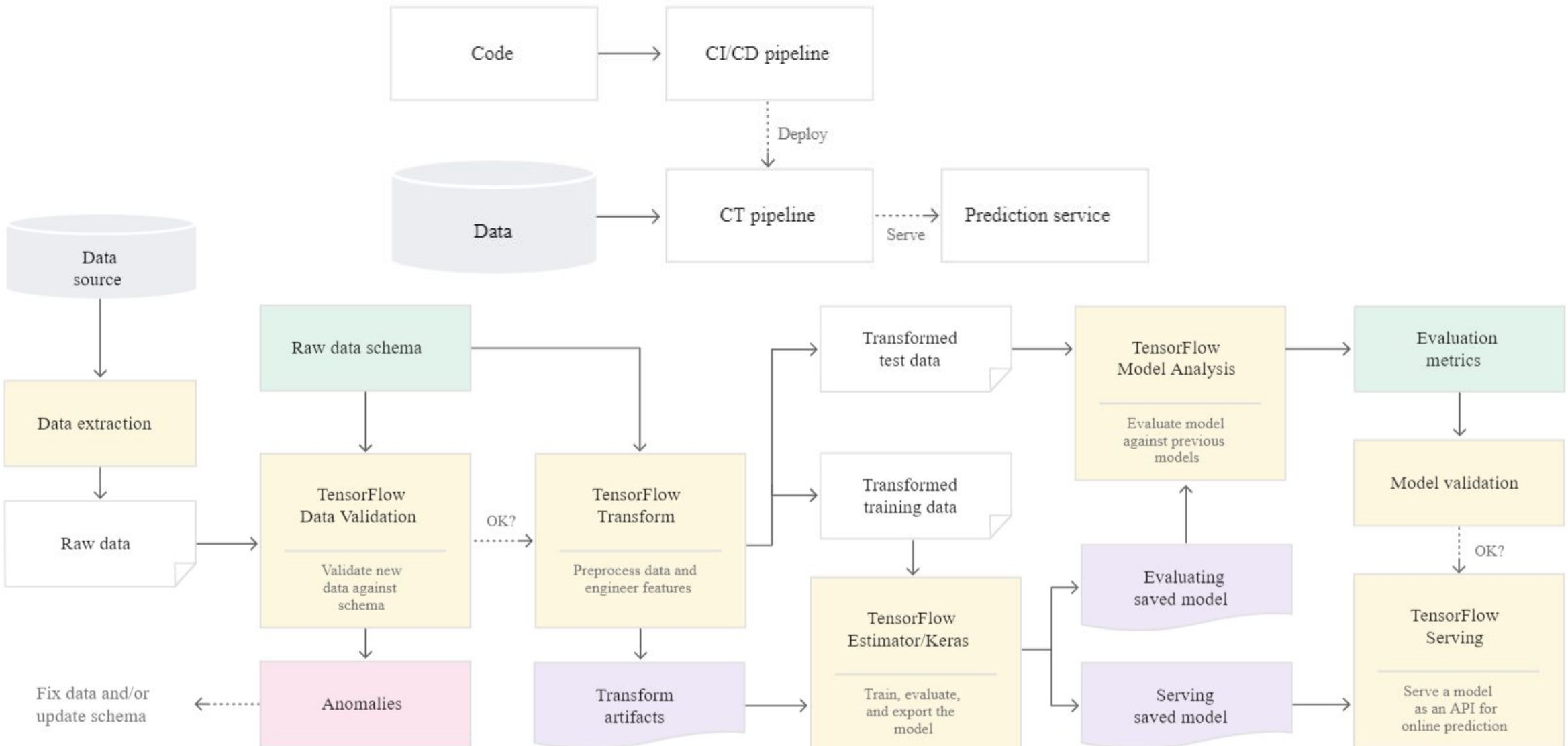
Secondly, you can use batching to optimize the cost and speed of processing. Because a model can run on the whole batch instead of running separately on each image, batching allows you to decrease the difference between cold and warm invocation and improve overall speed.

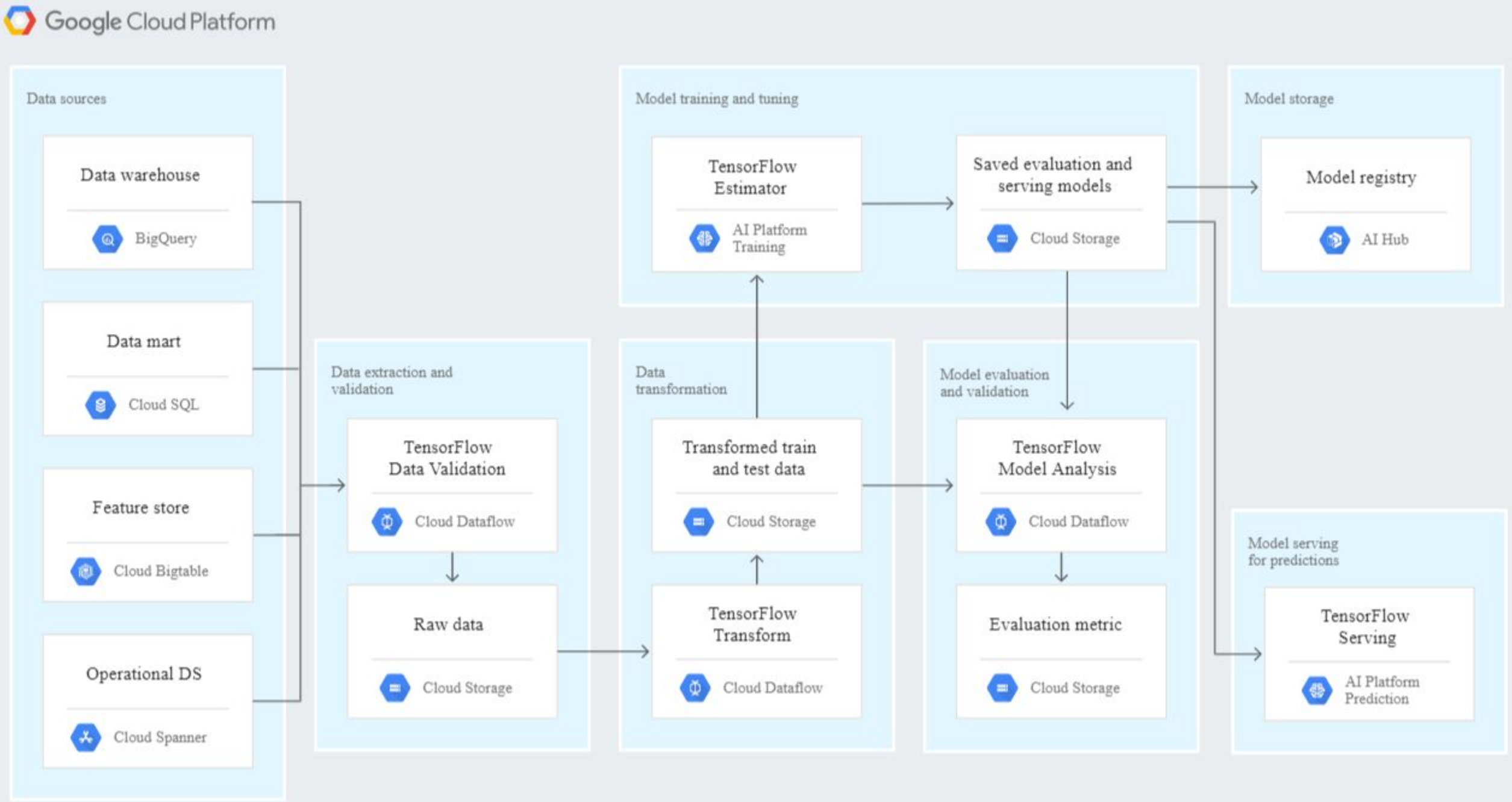
Finally, you can have some part of your model saved as part of your libraries. This allows you to save time downloading the model during cold invocation. You could also try to divide the model into layers and chain them together on separate functions. In this case, each function will send an intermediate activation layer down the chain and neither of the functions would need to download the model.

https://cloud.google.com/docs/authentication

Requirement	Recommendation	Comment
Accessing public data anonymously	API key	An API key only identifies the application and doesn't require user authentication. It is sufficient for accessing public data.
Accessing private data on behalf of an end user	OAuth 2.0 client	An OAuth 2.0 client identifies the application and lets end users authenticate your application with Google. It allows your application to access Google Cloud APIs on behalf of the end user.
Accessing private data on behalf of a service account inside Google Cloud environments	Environment-provided service account	If your application runs inside a Google Cloud environment, such as Compute Engine, App Engine, GKE, Cloud Run, or Cloud Functions, your application should use the service account provided by the environment. Google Cloud Client Libraries will automatically find and use the service account credentials.
Accessing private data on behalf of a service account outside Google Cloud environments	Service account key	You need to create a service account, and download its private key as a JSON file. You need to pass the file to Google Cloud Client Libraries, so they can generate the service account credentials at runtime. Google Cloud Client Libraries will automatically find and use the service account credentials by using the `GOOGLE_APPLICATION_CREDENTIALS` environment variable.

- If given new implementation, a successful CI/CD pipeline deploys a new ML CT pipeline.
- If given new data, a successful CT pipeline serves a new model prediction service.




Google Cloud Platform
Data sources
Data warehouse
BigQuery
Data mart
Cloud SQL
Feature store
Cloud Bigtable
Operational DS
Cloud Spanner
Data extraction and validation
TensorFlow Data Validation
Cloud Dataflow
Raw data
Cloud Storage
Data transformation
Transformed train and test data
Cloud Storage
TensorFlow Transform
Cloud Dataflow
Model training and tuning
TensorFlow Estimator
AI Platform Training
Model evaluation and validation
TensorFlow Model Analysis
Cloud Dataflow
Evaluation metric
Cloud Storage
Saved evaluation and serving models
Cloud Storage
Model storage
Model registry
AI Hub
Model serving for predictions
TensorFlow Serving
AI Platform Prediction

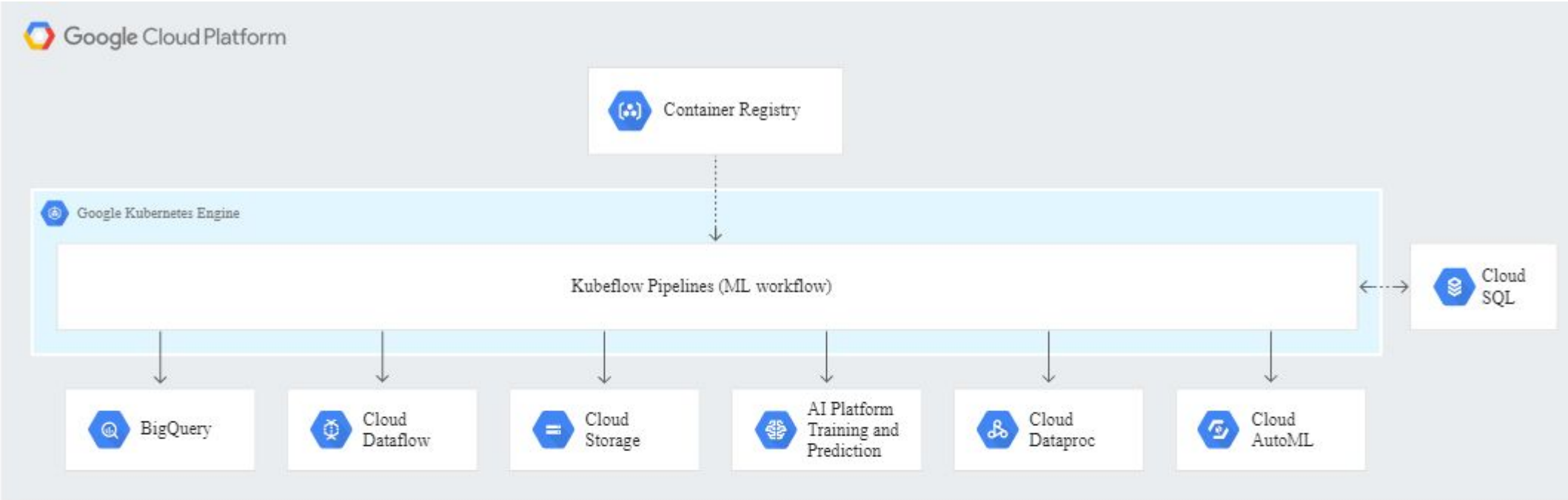


Figure 5. ML pipeline with Kubeflow Pipelines and Google Cloud managed services.

Kubeflow Pipelines lets you orchestrate and automate a production ML pipeline by executing the required Google Cloud services. In figure 5, Cloud SQL serves as the ML metadata store for Kubeflow Pipelines.

Kubeflow Pipelines components aren't limited to executing TFX-related services on Google Cloud. These components can execute any data-related and compute-related services, including Dataproc for SparkML jobs, AutoML, and other compute workloads.

You can automate the ML production pipelines to retrain your models with new data. You can trigger your pipeline on demand, on a schedule, on the availability of new data, on model performance degradation, on significant changes in the statistical properties of the data, or based on other conditions.

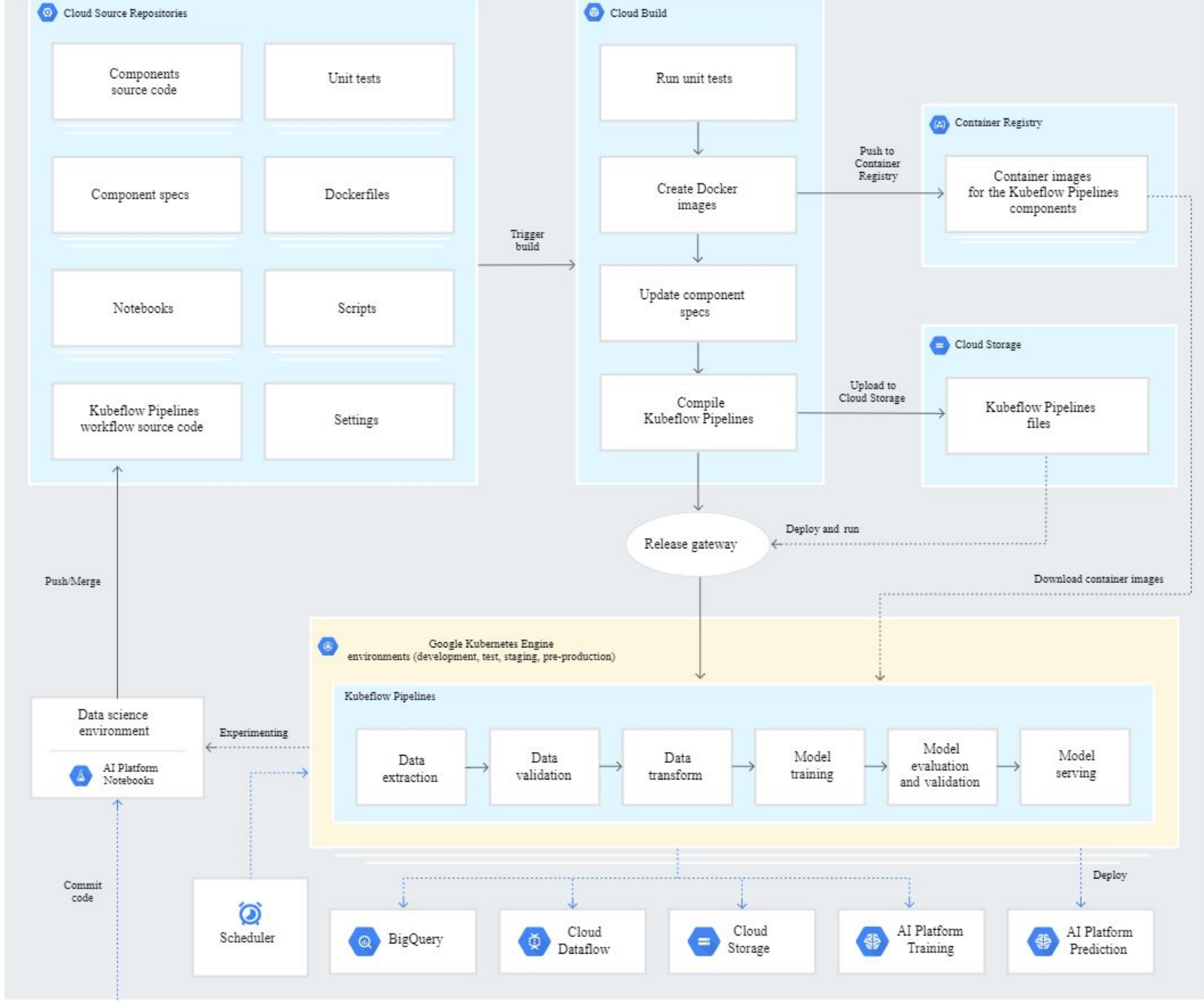
Triggering and scheduling Kubeflow Pipelines

When you deploy a Kubeflow pipeline to production, you need to automate its executions, depending on the scenarios discussed in the ML pipeline automation section.

Kubeflow Pipelines provides a Python SDK to operate the pipeline programmatically. The `kfp.Client` class includes APIs to create experiments, and to deploy and run pipelines. By using the Kubeflow Pipelines SDK, you can invoke Kubeflow Pipelines using the following services:

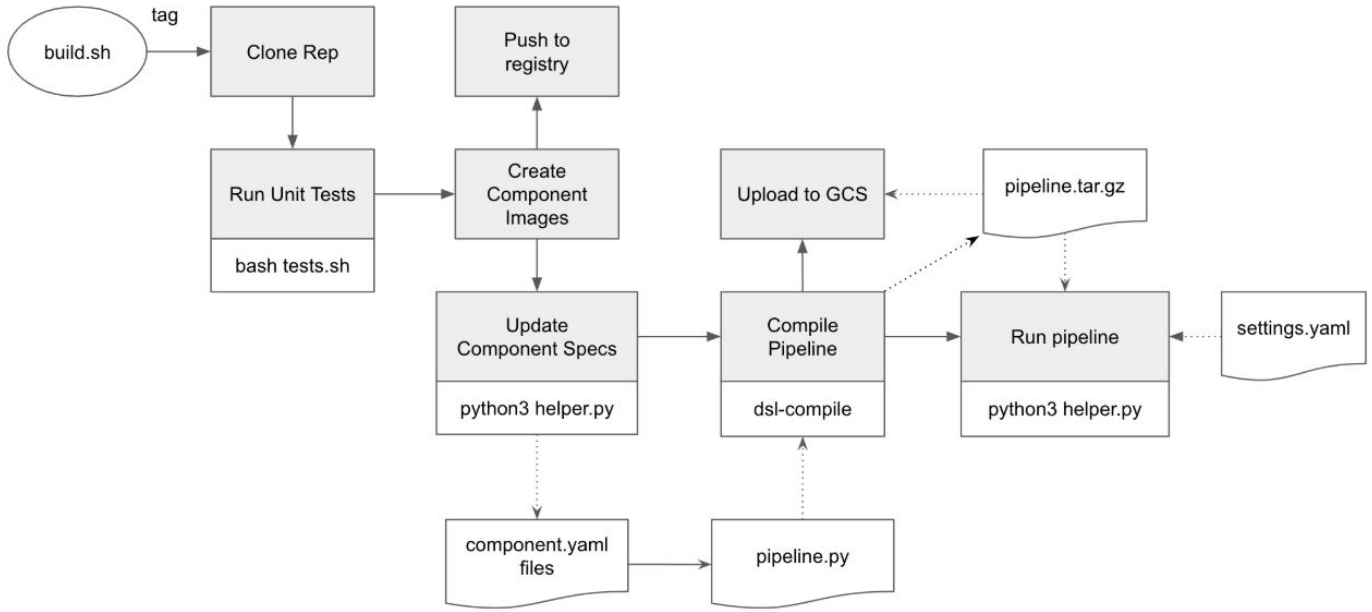
- On a schedule, using Cloud Scheduler.
- Responding to an event, using Pub/Sub and Cloud Functions. For example, the event can be the availability of new data files in a Cloud Storage bucket.
- As part of a bigger data and process workflow, using Cloud Composer or Cloud Data Fusion.

Kubeflow Pipelines also provides a built-in scheduler for recurring pipelines in Kubeflow Pipelines.


Cloud Source Repositories
Components source code
Unit tests
Component specs
Dockerfiles
Notebooks
Scripts
Kubeflow Pipelines workflow source code
Settings
Trigger build
Cloud Build
Run unit tests
Create Docker images
Update component specs
Compile Kubeflow Pipelines
Push to Container Registry
Container Registry
Container images for the Kubeflow Pipelines components
Upload to Cloud Storage
Cloud Storage
Kubeflow Pipelines files
Release gateway
Deploy and run
Push/Merge
Download container images
Google Kubernetes Engine
environments (development, test, staging, pre-production)
Kubeflow Pipelines
Data extraction
Data validation
Data transform
Model training
Model evaluation and validation
Model serving
Data science environment
AI Platform Notebooks
Experimenting
Commit code
Scheduler
Deploy
BigQuery
Cloud Dataflow
Cloud Storage
AI Platform Training
AI Platform Prediction

Cloud Build Steps

The following diagram shows the build steps:



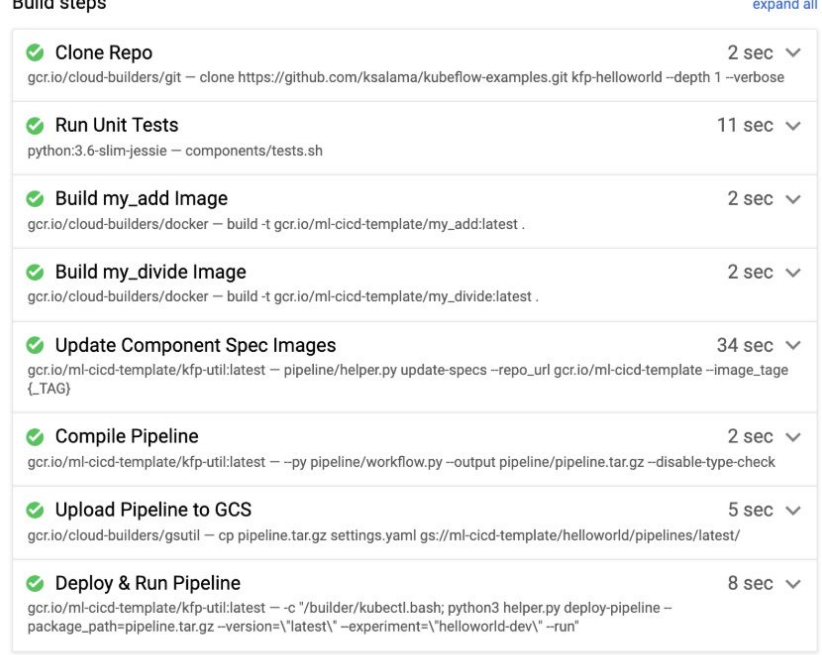
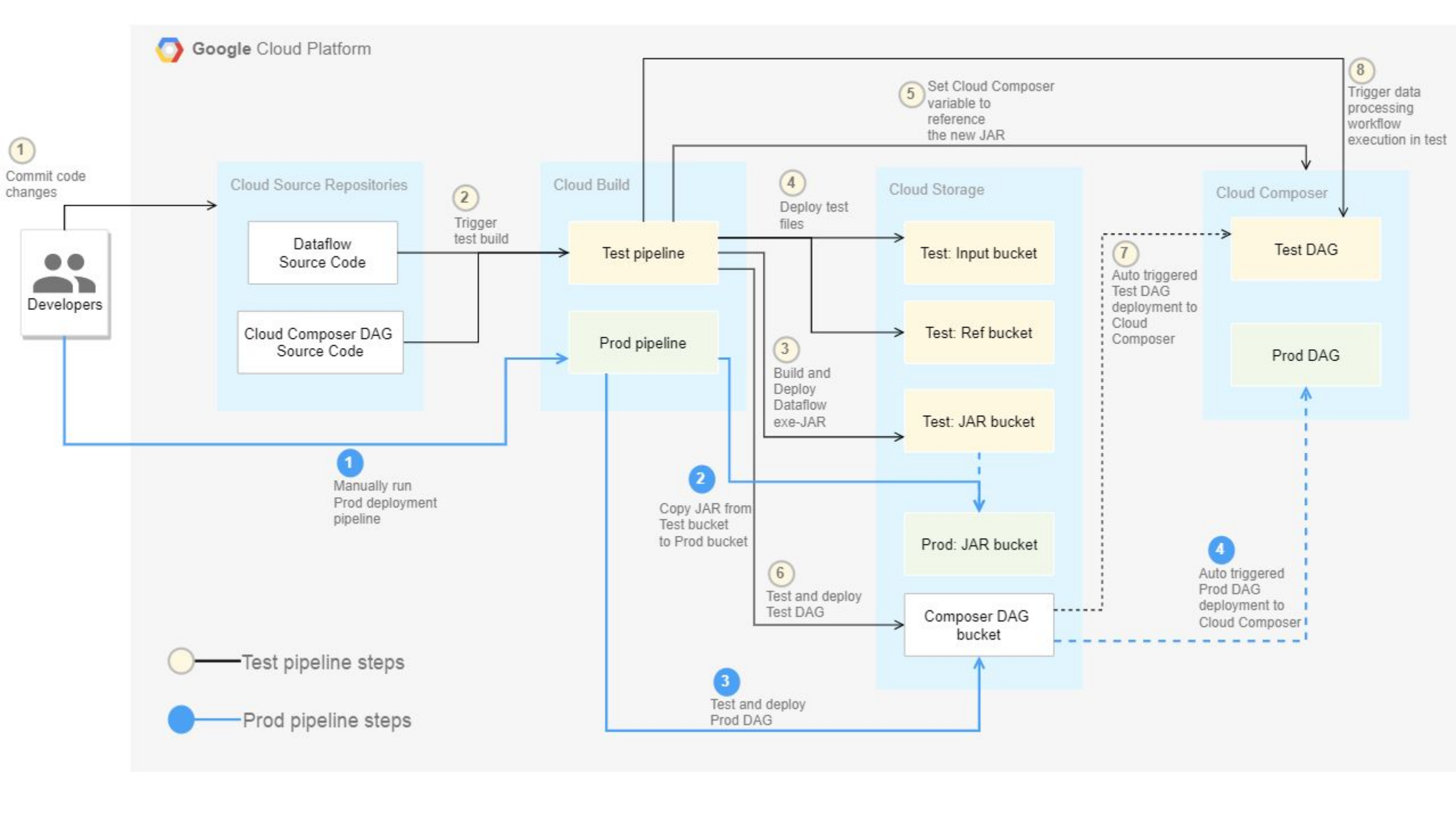


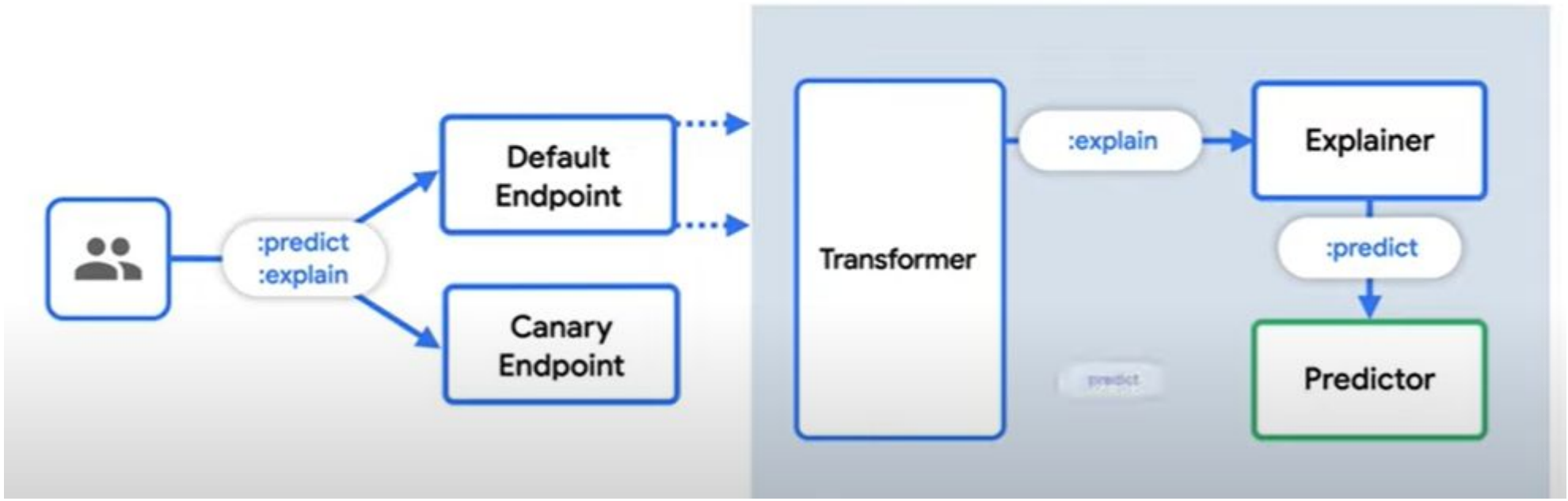
Figure 7. Example build steps performed by Cloud Build.

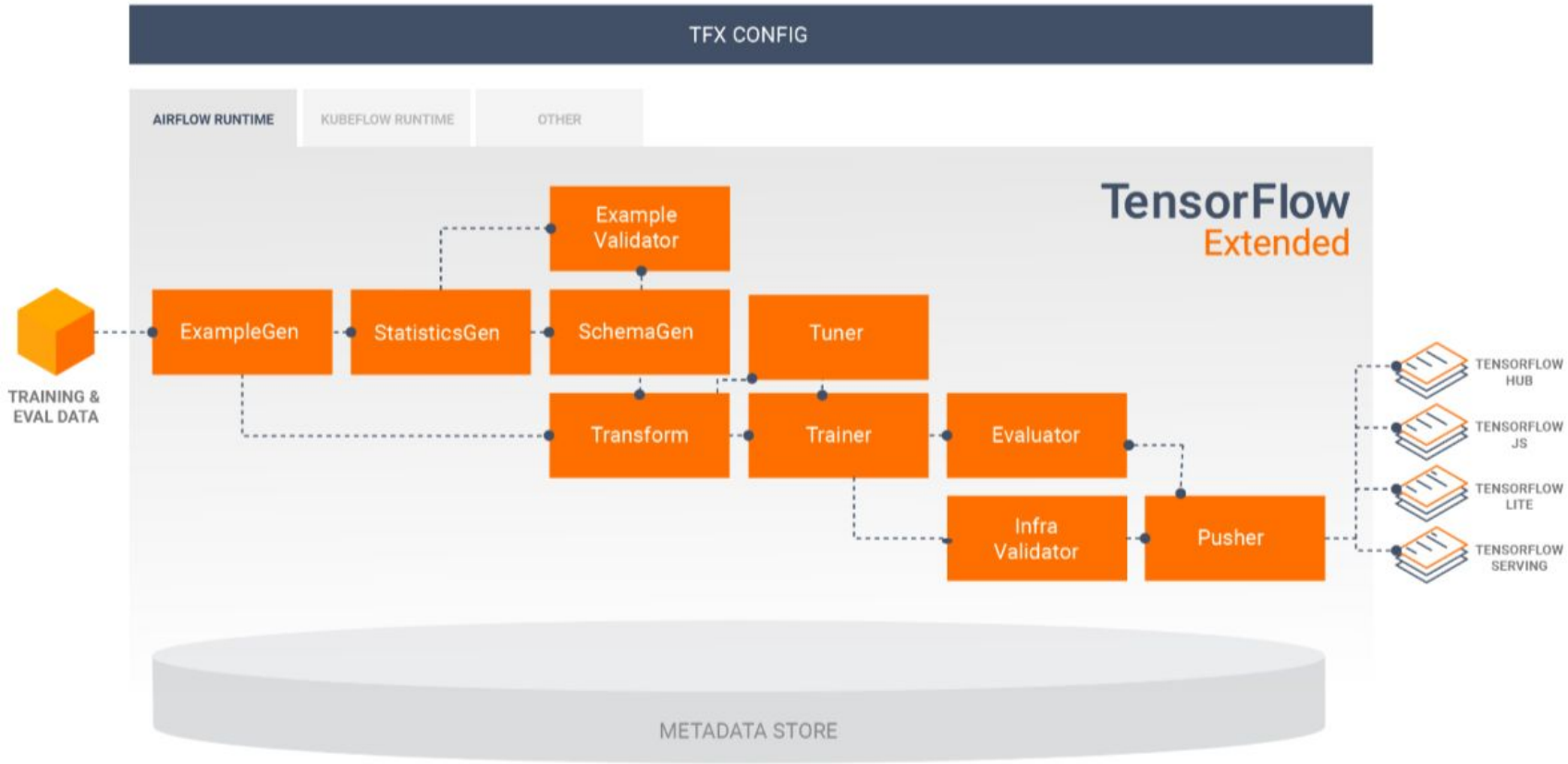
As shown in figure 7, Cloud Build performs the following build steps:

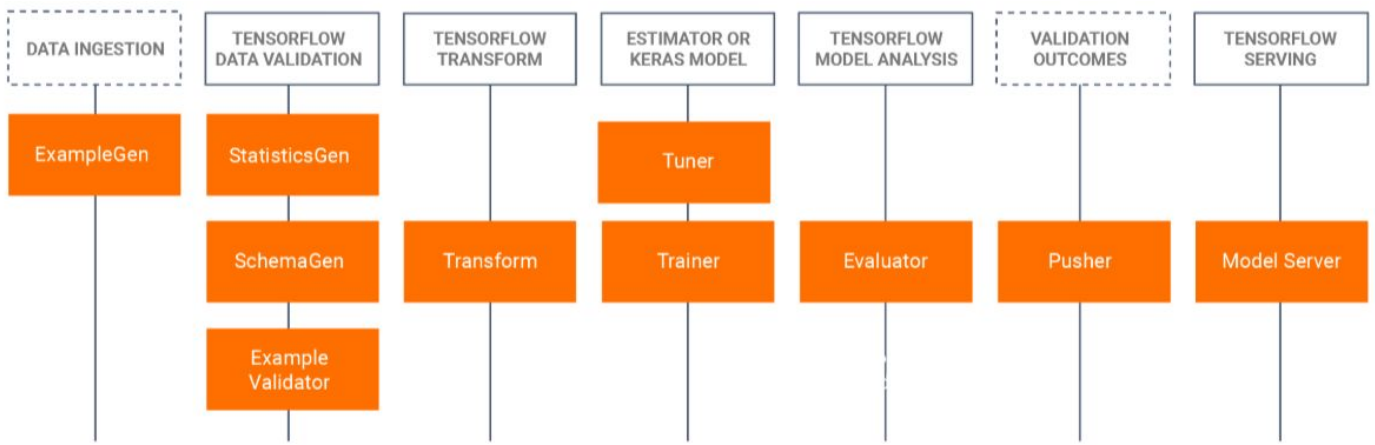
1. The source code repository is copied to the Cloud Build runtime environment, under the `/workspace` directory.
2. Unit tests are run.
3. (Optional) Static code analysis is run, such as Pylint.
4. If the tests pass, the Docker container images are built, one for each Kubeflow Pipelines component. The images are tagged with the `$COMMIT_SHA` parameter.
5. The Docker container images are uploaded to the Container Registry.
6. The image URL is updated in each of the `component.yaml` files with the created and tagged Docker container images.
7. The Kubeflow Pipelines workflow is compiled to produce the `workflow.tar.gz` file.
8. The `workflow.tar.gz` file is uploaded to Cloud Storage.
9. The compiled pipeline is deployed to Kubeflow Pipelines, which involves the following steps:
    a. Read the pipeline parameters from the `settings.yaml` file.
    b. Create an experiment (or use an existing one).
    c. Deploy the pipeline to Kubeflow Pipelines (and tag its name with a version).
10. (Optional) Run the pipeline with the parameter values as part of an integration test or production execution. The executed pipeline eventually deploys a model as an API on AI Platform.


Google Cloud Platform
1
Commit code changes
Developers
Cloud Source Repositories
Dataflow Source Code
Cloud Composer DAG Source Code
2
Trigger test build
Cloud Build
Test pipeline
Prod pipeline
4
Deploy test files
3
Build and Deploy Dataflow exe-JAR
5
Set Cloud Composer variable to reference the new JAR
8
Trigger data processing workflow execution in test
Cloud Storage
Test: Input bucket
Test: Ref bucket
Test: JAR bucket
Prod: JAR bucket
Composer DAG bucket
Cloud Composer
Test DAG
Prod DAG
7
Auto triggered Test DAG deployment to Cloud Composer
1
Manually run Prod deployment pipeline
2
Copy JAR from Test bucket to Prod bucket
6
Test and deploy Test DAG
3
Test and deploy Prod DAG
4
Auto triggered Prod DAG deployment to Cloud Composer
Test pipeline steps
Prod pipeline steps

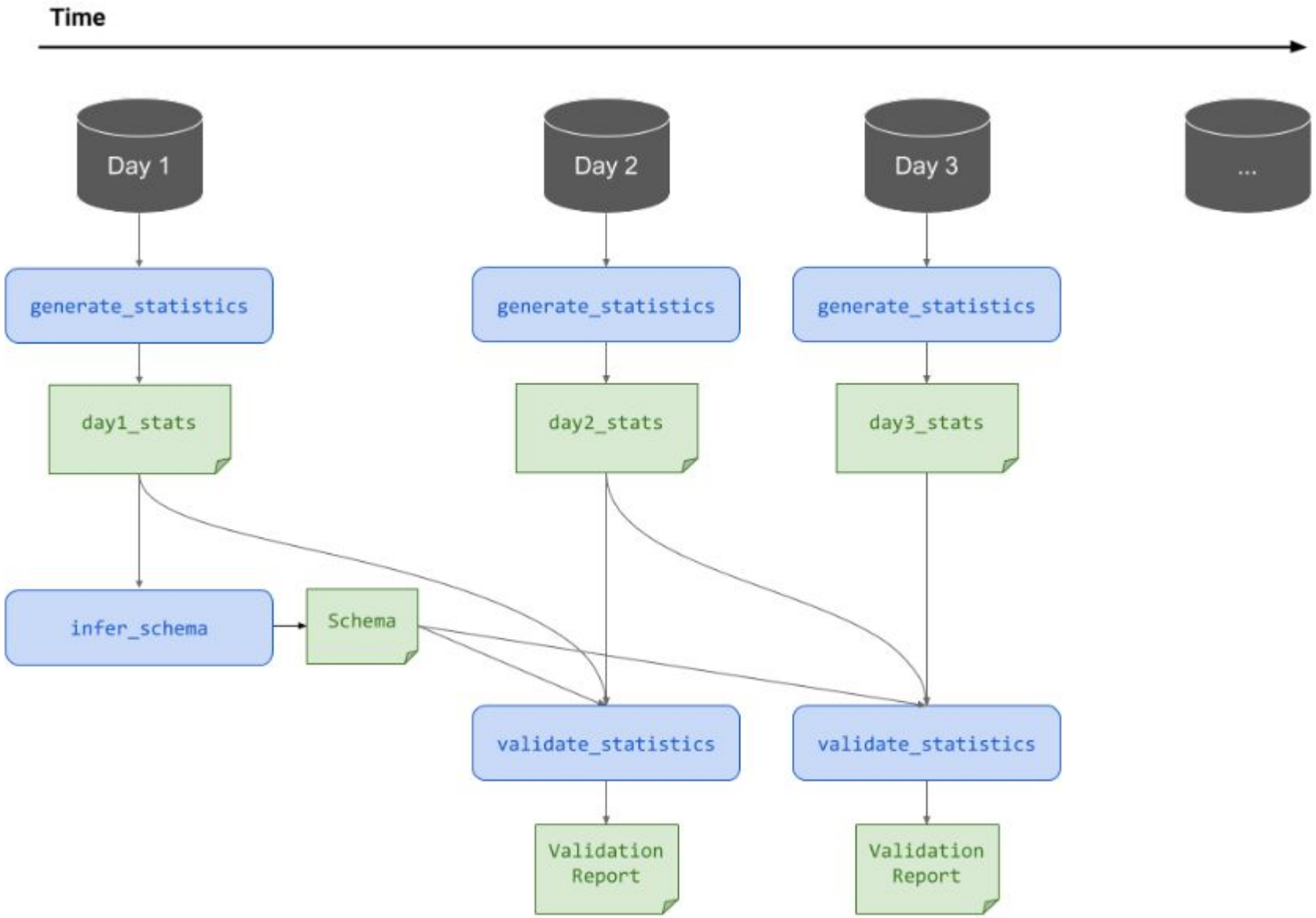
Opinionated ML microservices




TFX CONFIG
AIRFLOW RUNTIME
KUBEFLOW RUNTIME
OTHER
TensorFlow
Extended
TRAINING &
EVAL DATA
ExampleGen
StatisticsGen
Example
Validator
SchemaGen
Tuner
Transform
Trainer
Evaluator
Infra
Validator
Pusher
TENSORFLOW
HUB
TENSORFLOW
JS
TENSORFLOW
LITE
TENSORFLOW
SERVING
METADATA STORE


DATA INGESTION
TENSORFLOW
DATA VALIDATION
TENSORFLOW
TRANSFORM
ESTIMATOR OR
KERAS MODEL
TENSORFLOW
MODEL ANALYSIS
VALIDATION
OUTCOMES
TENSORFLOW
SERVING
ExampleGen
StatisticsGen
SchemaGen
Example
Validator
Transform
Tuner
Trainer
Evaluator
Pusher
Model Server

https://blog.tensorflow.org/2018/09/introducing-tensorflow-data-validation.html



https://towardsdatascience.com/scaling-apache-airflow-for-machine-learning-workflows-f2446257e495

Input	Execution	Output
Data files	Author	Logs
Code repository	Duration	Model
Code commit	Cost	Results
Docker image	CPU/GPU and memory usage	Metrics
Parameters		Comments

Important information for machine learning version control.

The steps involved in using containers

The following steps show the basic process for training with custom containers:

1. Set up a Google Cloud project and your local environment.
2. Create a custom container:
    a. Write a Dockerfile that sets up your container to work with AI Platform Training, and includes dependencies needed for your training application.
    b. Build and test your Docker container locally.
3. Push the container to Container Registry.
4. Submit a training job that runs on your custom container.

Using hyperparameter tuning or GPUs requires some adjustments, but the basic process is the same.

For use with AI Platform Training, your Dockerfile needs to include commands that cover the following tasks:

- Choose a base image
- Install additional dependencies
- Copy your training code to the image
- Configure the entry point for AI Platform Training to invoke your training code

```
# Specifies base image and tag
FROM image:tag
WORKDIR /root

# Installs additional packages
RUN pip install pkg1 pkg2 pkg3

# Downloads training data
RUN curl https://example-url/path-to-data/data-filename --output /root/data-filename

# Copies the trainer code to the docker image.
COPY your-path-to/model.py /root/model.py
COPY your-path-to/task.py /root/task.py

# Sets up the entry point to invoke the trainer.
ENTRYPOINT ["python", "task.py"]
```

```
export PROJECT_ID=$(gcloud config list project --format "value(core.project)")
export BUCKET_NAME=custom_containers
export MASTER_IMAGE_REPO_NAME=master_image_name
export MASTER_IMAGE_TAG=master_tag
export MASTER_IMAGE_URI=gcr.io/$PROJECT_ID/$MASTER_IMAGE_REPO_NAME:$MASTER_IMAGE_TAG
export WORKER_IMAGE_REPO_NAME=worker_image_name
export WORKER_IMAGE_TAG=worker_tag
export WORKER_IMAGE_URI=gcr.io/$PROJECT_ID/$WORKER_IMAGE_REPO_NAME:$WORKER_IMAGE_TAG
export PS_IMAGE_REPO_NAME=ps_image_name
export PS_IMAGE_TAG=ps_tag
export PS_IMAGE_URI=gcr.io/$PROJECT_ID/$PS_IMAGE_REPO_NAME:$PS_IMAGE_TAG
export MODEL_DIR=distributed_example_$(date +%Y%m%d_%H%M%S)
export REGION=us-central1
export JOB_NAME=distributed_container_job_$(date +%Y%m%d_%H%M%S)

docker build -f Dockerfile-master -t $MASTER_IMAGE_URI ./
docker build -f Dockerfile-worker -t $WORKER_IMAGE_URI ./
docker build -f Dockerfile-ps -t $PS_IMAGE_URI ./

docker run $MASTER_IMAGE_URI --epochs 1
docker run $WORKER_IMAGE_URI --epochs 1
docker run $PS_IMAGE_URI --epochs 1

docker push $MASTER_IMAGE_URI
docker push $WORKER_IMAGE_URI
docker push $PS_IMAGE_URI

gcloud ai-platform jobs submit training $JOB_NAME \
  --region $REGION \
  --master-machine-type complex_model_m \
  --master-image-uri $MASTER_IMAGE_URI \
  --worker-machine-type complex_model_m \
  --worker-image-uri $WORKER_IMAGE_URI \
  --worker-count 9 \
  --parameter-server-machine-type large_model \
  --parameter-server-image-uri $PS_IMAGE_URI \
  --parameter-server-count 3 \
  -- \
  --model-dir=gs://$BUCKET_NAME/$MODEL_DIR \
  --epochs=10
```

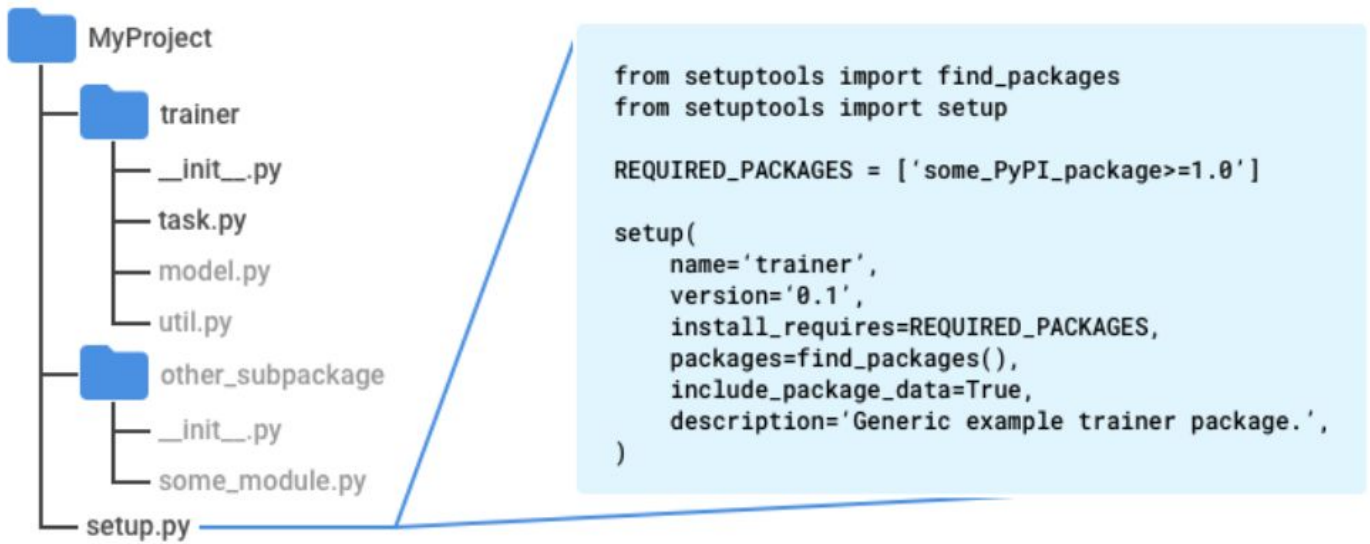
Limitations

Please note the following limitations for training with built-in algorithms:

- Distributed training is not supported. To run a distributed training job on AI Platform Training, you must create a training application.
- Training jobs submitted through the Google Cloud Console use only legacy machine types. You can use Compute Engine machine types with training jobs submitted through `gcloud` or the Google APIs Client Library for Python. Learn more about machine types for training.
- GPUs are supported for some algorithms. Refer to the detailed comparison of all the built-in algorithms for more information.
- Multi-GPU machines do not yield greater speed with built-in algorithm training. If you're using GPUs, select machines with a single GPU.
- TPUs are not supported for tabular built-in algorithm training. You must create a training application. Learn how to run a training job with TPUs.

If you run training jobs using the AI Platform Training and Prediction API directly, you must stage your dependency packages in a Cloud Storage location yourself and then use the paths to the packages in that location.

> ★ **Note:** It's not essential to build your package manually even if you need to perform some setup operations when your application package is installed. You can instead include a custom **`__init__.py`** file in your package's root directory and use the **`gcloud`** command-line tool to create and upload your package. If you include **`--package-path`** when you run the **`gcloud ai-platform jobs submit`** command, the tool automatically looks for **`__init__.py`** files to use.



When you run `gcloud ai-platform jobs submit training`, set the `--package-path` to `trainer`. This causes the `gcloud` tool to look for a `setup.py` file in the *parent* of `trainer`, your main project directory.

```
gcloud ai-platform jobs submit training $JOB_NAME \
    --staging-bucket $PACKAGE_STAGING_PATH \
    --package-path /Users/mluser/models/faces/trainer \
    --module-name $MAIN_TRAINER_MODULE \
    --packages dep1.tar.gz,dep2.whl \
    --region us-central1 \
    -- \
    --user_first_arg=first_arg_value \
    --user_second_arg=second_arg_value
```

Similarly, the example below specifies packaged dependencies named `dep1.tar.gz` and `dep2.whl` (one each of the supported package types), but with a built training application:

```
gcloud ai-platform jobs submit training $JOB_NAME \
    --staging-bucket $PACKAGE_STAGING_PATH \
    --module-name $MAIN_TRAINER_MODULE \
    --packages trainer-0.0.1.tar.gz,dep1.tar.gz,dep2.whl
    --region us-central1 \
    -- \
    --user_first_arg=first_arg_value \
    --user_second_arg=second_arg_value
```

TensorFlow Lite: ML for mobile and IoT devices (TF Dev Summit '20)

https://github.com/google/ml-metadata/blob/master/g3doc/get_started.md

Functionality Enabled by MLMD

Tracking the inputs and outputs of all components/steps in an ML workflow and their lineage allows ML platforms to enable several important features. The following list provides a non-exhaustive overview of some of the major benefits.

- **List all Artifacts of a specific type.** Example: all Models that have been trained.
- **Load two Artifacts of the same type for comparison.** Example: compare results from two experiments.
- **Show a DAG of all related executions and their input and output artifacts of a context.** Example: visualize the workflow of an experiment for debugging and discovery.
- **Recurse back through all events to see how an artifact was created.** Examples: see what data went into a model; enforce data retention plans.
- **Identify all artifacts that were created using a given artifact.** Examples: see all Models trained from a specific dataset; mark models based upon bad data.
- **Determine if an execution has been run on the same inputs before.** Example: determine whether a component/step has already completed the same work and the previous output can just be reused.
- **Record and query context of workflow runs.** Examples: track the owner and changelist used for a workflow run; group the lineage by experiments; manage artifacts by projects.

1) Register ArtifactTypes

2) Register ExecutionTypes

3) Create DataSet Artifact

4) Create Execution for Trainer

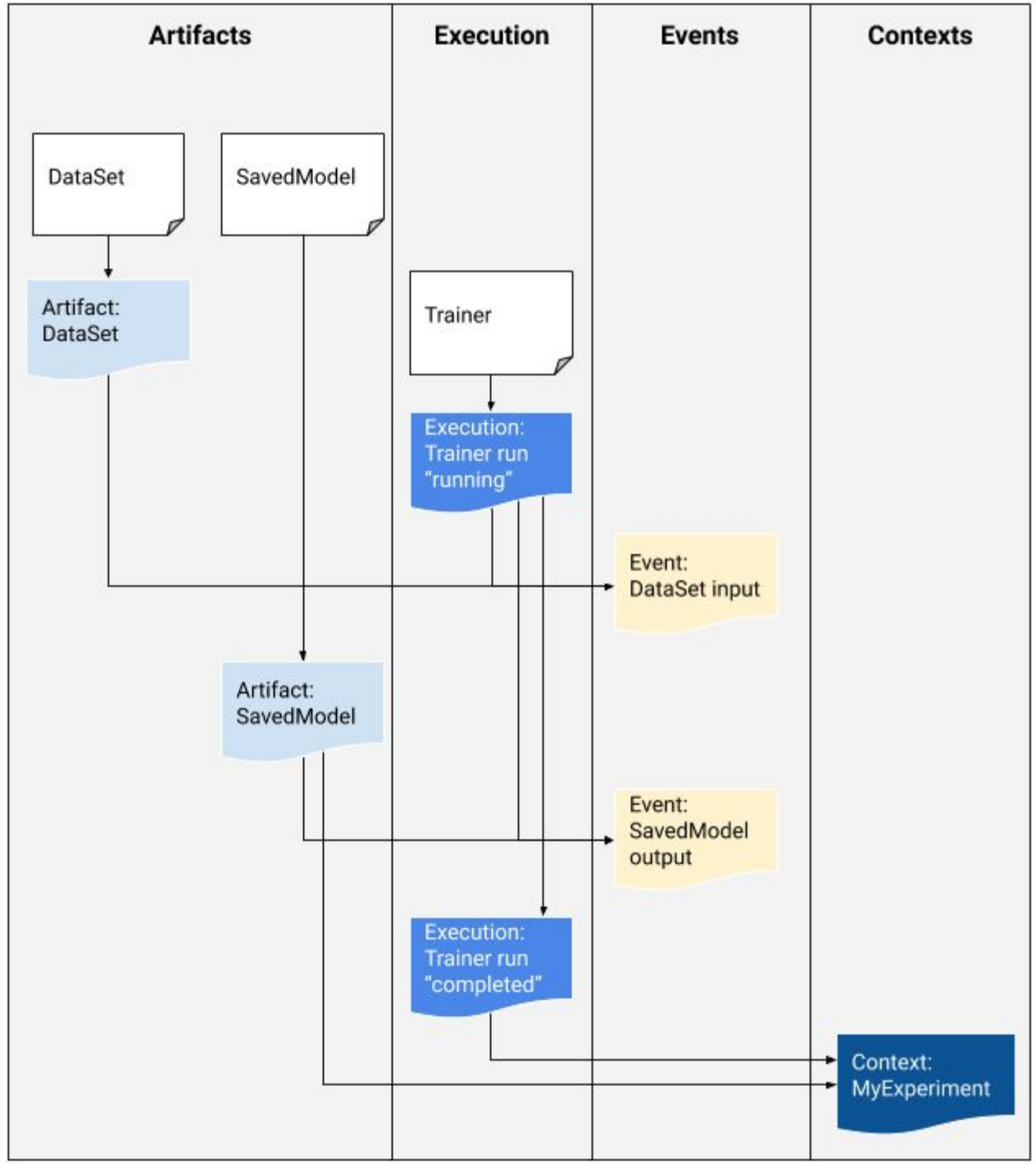
5) Read DataSet and record input event

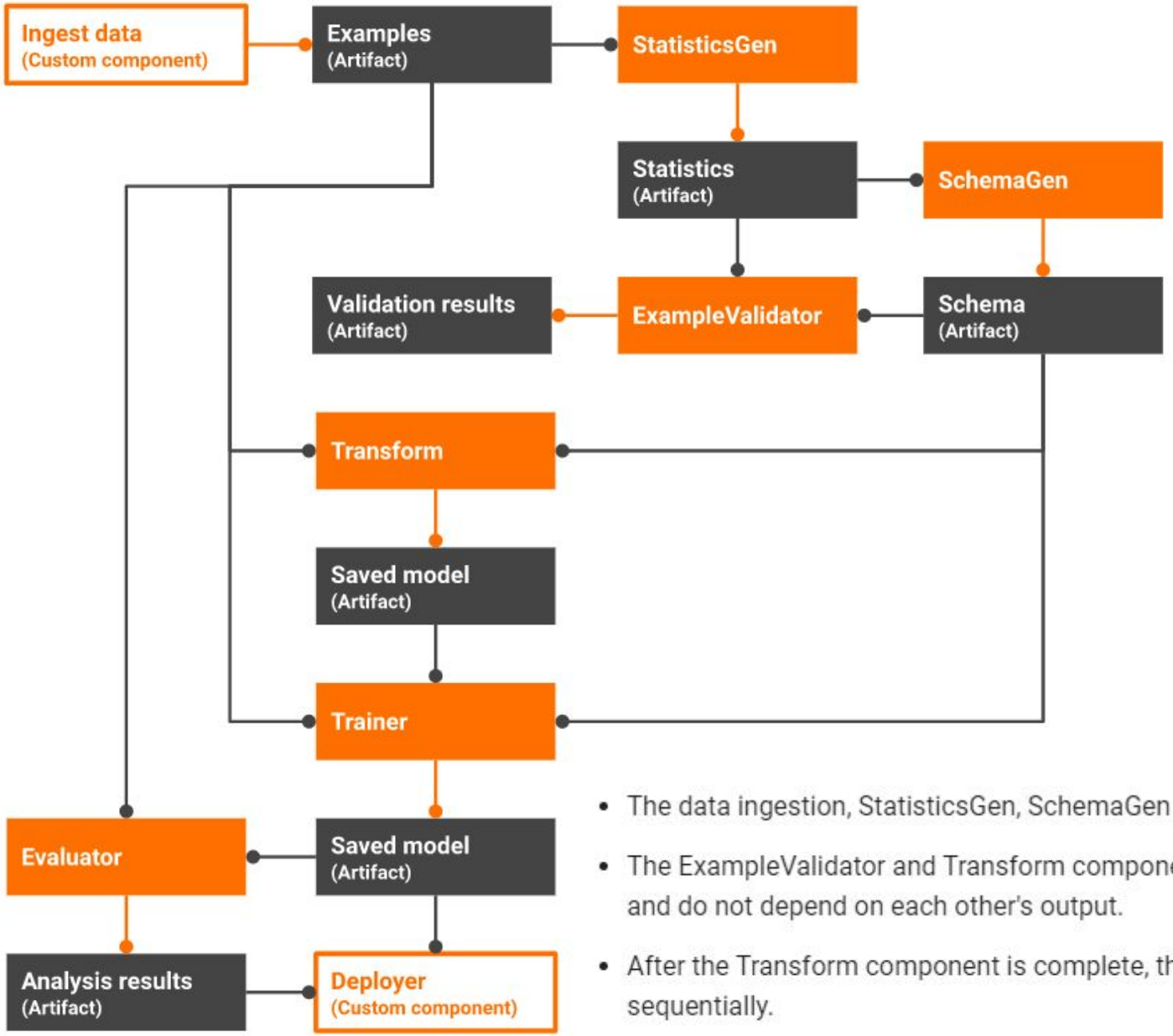
6) Train Model and Create SavedModel Artifact

7) Write SavedModel and record output event

8) Mark Execution completed

9) Annotate the experiment with a Context





- The data ingestion, StatisticsGen, SchemaGen component instances sequentially.
- The ExampleValidator and Transform components can run in parallel since they share input artifact dependencies and do not depend on each other's output.
- After the Transform component is complete, the Trainer, Evaluator, and custom deployer component instances run sequentially.

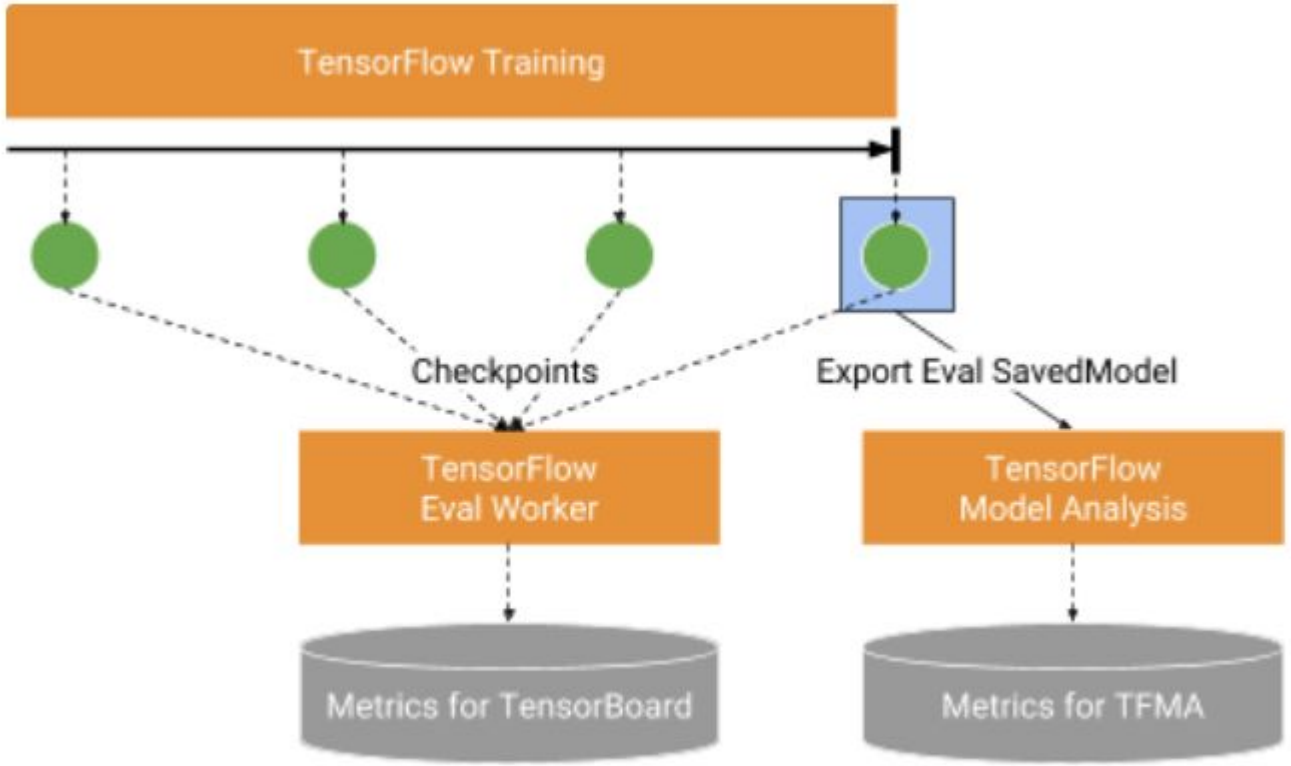


Figure 1: TensorBoard visualizes streaming metrics that are computed from checkpoints. TFMA computes and visualizes metrics using an exported SavedModel.

Note that TFMA computes the same TensorFlow metrics that are computed by the TensorFlow eval worker, just more accurately by doing a full pass over the specified dataset. TFMA can also be configured to compute additional metrics that were not defined in the model.

Furthermore, if evaluation datasets are sliced to compute metrics for specific segments, each of those segments may only contain a small number of examples. To compute accurate metrics, a deterministic full pass over those examples is important.

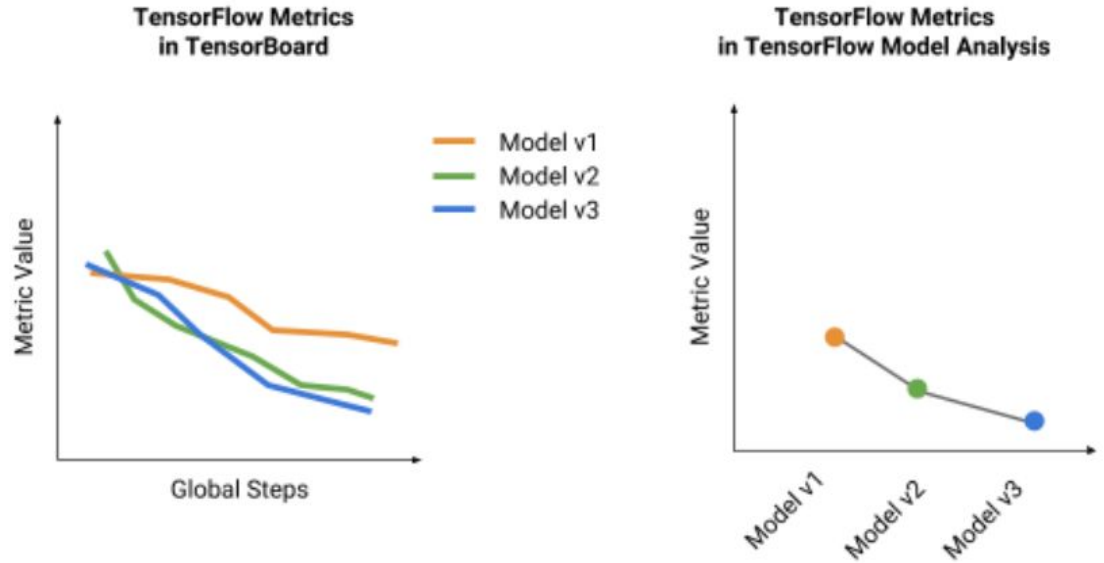


Figure 2: TensorBoard can overlay streaming metrics of multiple models over global training steps. TFMA only shows metrics computed based on the exported SavedModel, but can visualize those as a time series across multiple model versions.

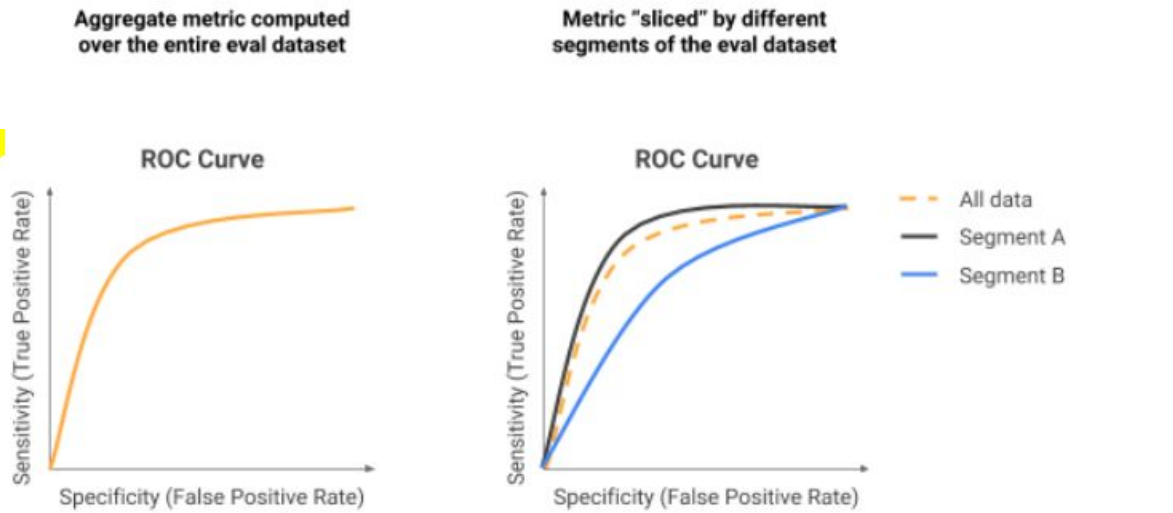
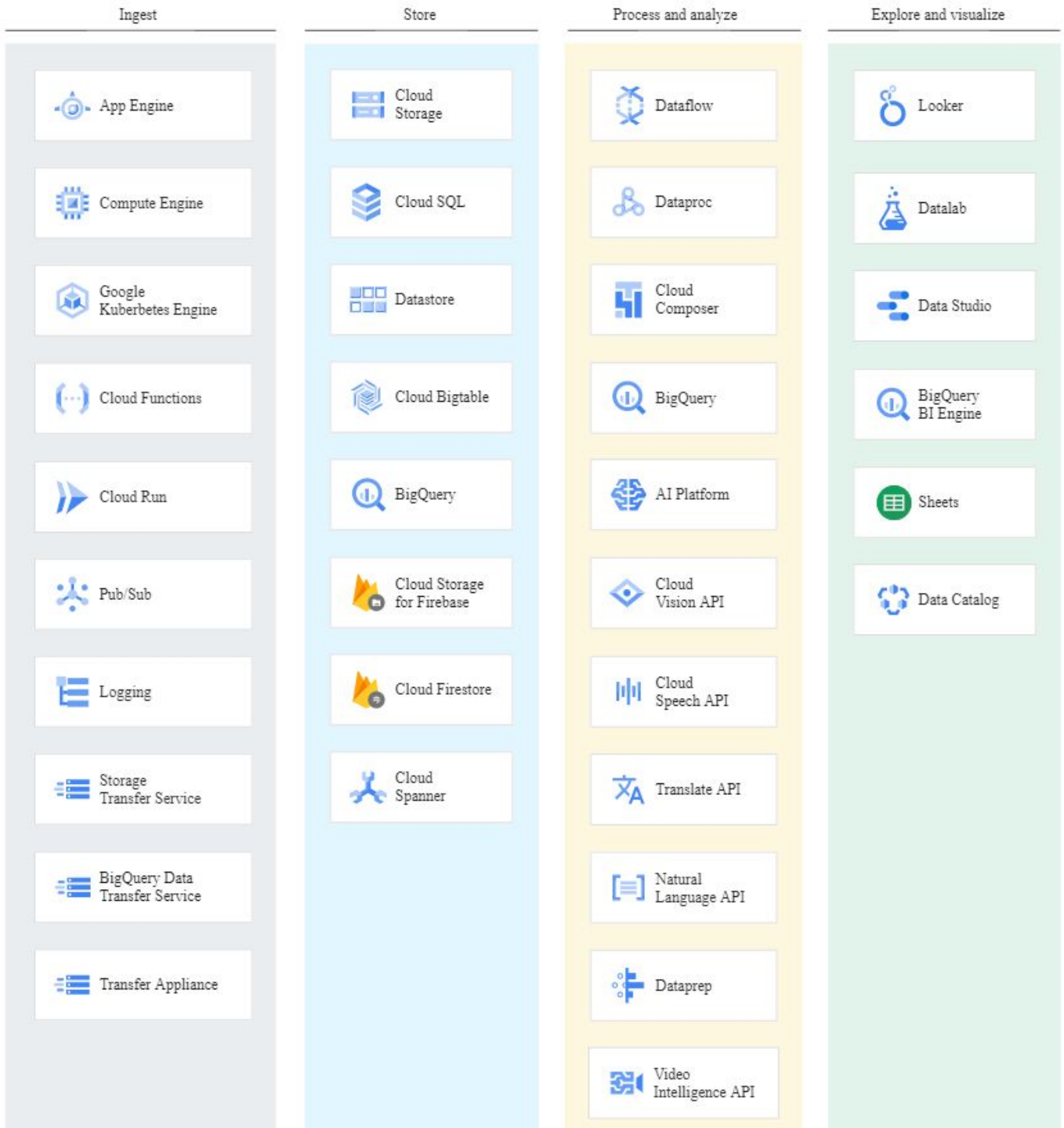
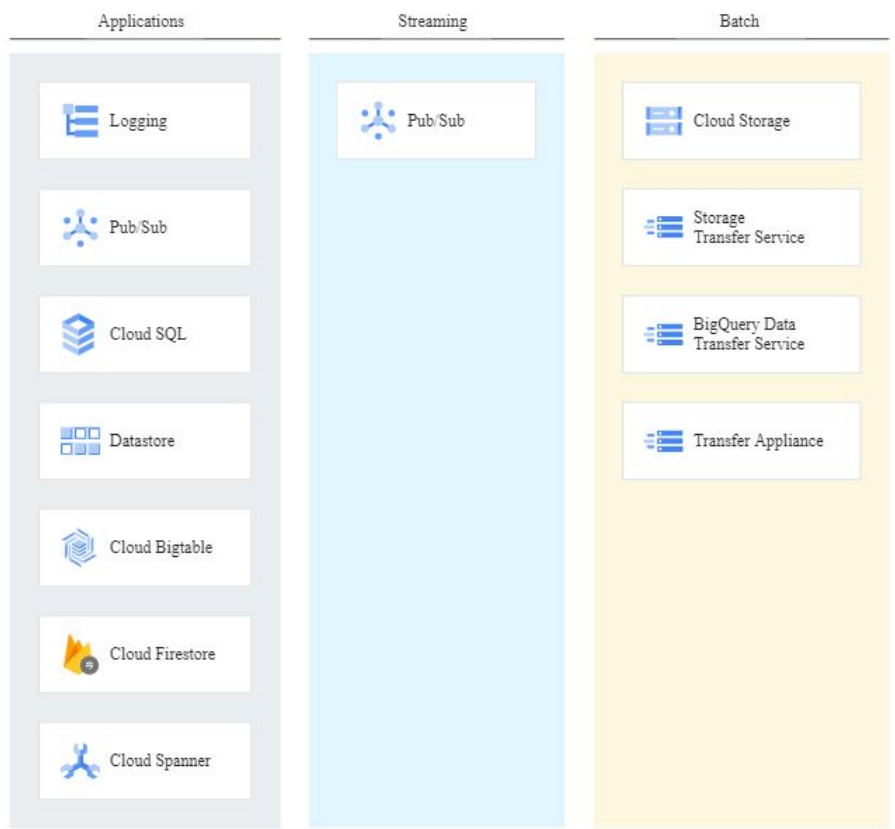


Figure 3: TFMA allows us to slice a metric by different segments of our eval dataset, enabling more fine grained analysis of a model and how it performs on different slices.



Ingest types

The following chart shows how Google Cloud services map to app, streaming, and batch workloads.



- Demand forecasting: estimating the demand for products by store on a daily basis for stock and intake optimization.
- Segmentation analysis: identifying your customer segments, emerging segments, and customers who are migrating across segments each month.
- Sentiment analysis and topic mining: identifying the overall sentiment regarding your products or services every week by analyzing customer feedback, and extracting trending topics in your market from social med

- **Offline prediction**. This is when your ML model is used in a *batch scoring* job for a large number of data points, where predictions are not required in real-time serving. In offline recommendations, for example, you only use *historical* information about customer-item interactions to make the prediction, without any need for online information. Offline recommendations are usually performed in retention campaigns for (inactive) customers with high propensity to churn, in promotion campaigns, and so on.
- **Online prediction**. This is when your ML system is used to serve *real-time* predictions, based on online requests from the operational systems and apps. In contrast to offline prediction, in online recommendations you need the *current context* of the customer who's using your application, along with historical information, to make the prediction. This context includes information such as datetime, page views, funnels, items viewed, items in the basket, items removed from the basket, customer device, and device location.

- Predictive maintenance: synchronously predicting whether a particular machine part will fail in the next N minutes, given the sensor's real-time data.
- Real-time bidding (RTB): synchronously recommending an ad and optimizing a bid when receiving a bid request. This information is then used to return an ad reference in real time.
- Predictive maintenance: asynchronously predicting whether a particular machine part will fail in the next N minutes, given the averages of the sensor's data in the past 30 minutes.
- Estimating asynchronously how long a food delivery will take based on the average delivery time in an area in the past 30 minutes, the ingredients, and real-time traffic information.

Real-time predictions can be delivered to the consumers (users, apps, systems, dashboards, and so on) in several ways:

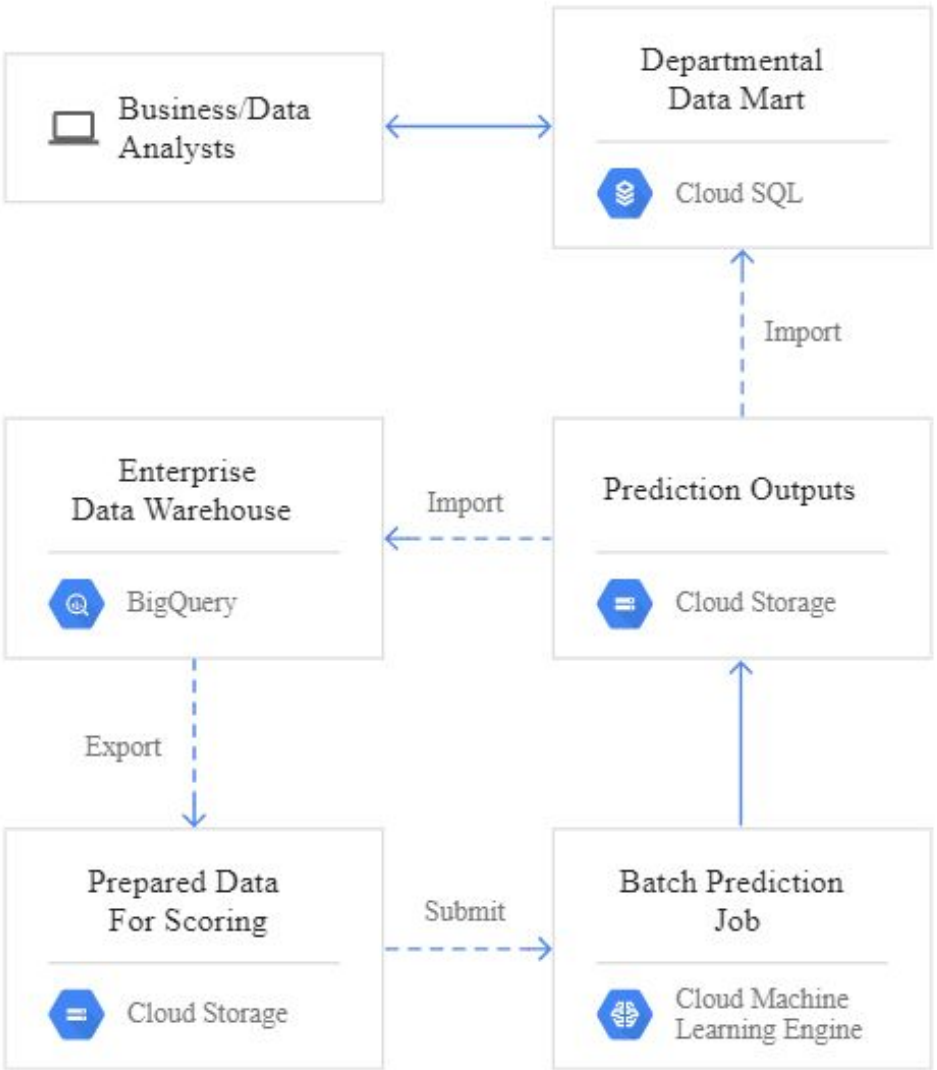
- **Synchronously**. The request for prediction and the response (the prediction) are performed in sequence between the caller and the ML model service. That is, the caller waits until it receives the prediction from the ML service before performing the subsequent steps.
- **Asynchronously**. Predictions or their subsequent actions, based on events streaming data, are delivered to th consumer independently of the request for prediction. This includes:
  - **Push**. The model generates predictions and pushes them to the caller or consumer as a notification. An example is fraud detection, when you want to notify other systems to take action when a potentially fraudulent transaction is identified.
  - **Poll**. The model generates predictions and stores them in a low read-latency database. The caller or consumer periodically polls the database for available predictions. An example is targeted marketing, where the system checks the propensity scores predicted in real time for active customers in order to decide whether to send an email with a promotion or a retention offer.

Making and serving predictions

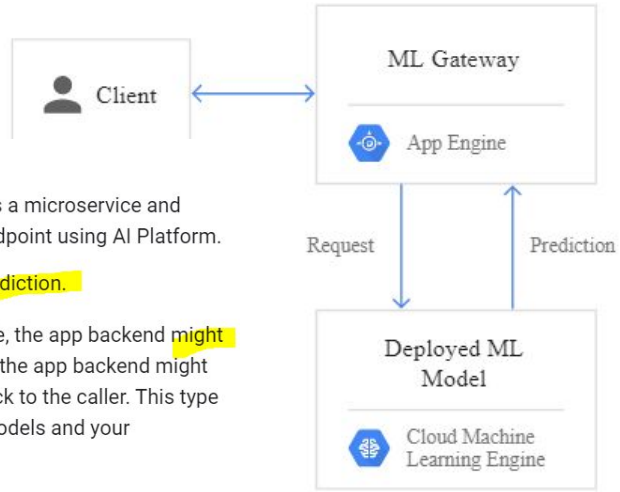
For real-time use cases, minimizing latency to serve prediction is important, because the expected action should happen immediately. You can usually improve serving latency at two levels:

- The model level, where you minimize the time your model takes to make a prediction when it's invoked with data point. This includes building smaller models, as well as using accelerators for serving your models.
- The serving level, where you minimize the time your system takes to serve the prediction when it receives request. This includes storing your input features in a low read-latency lookup data store, precomputing predictions in an offline batch-scoring job, and caching the predictions.

Batch



Online (direct REST call)

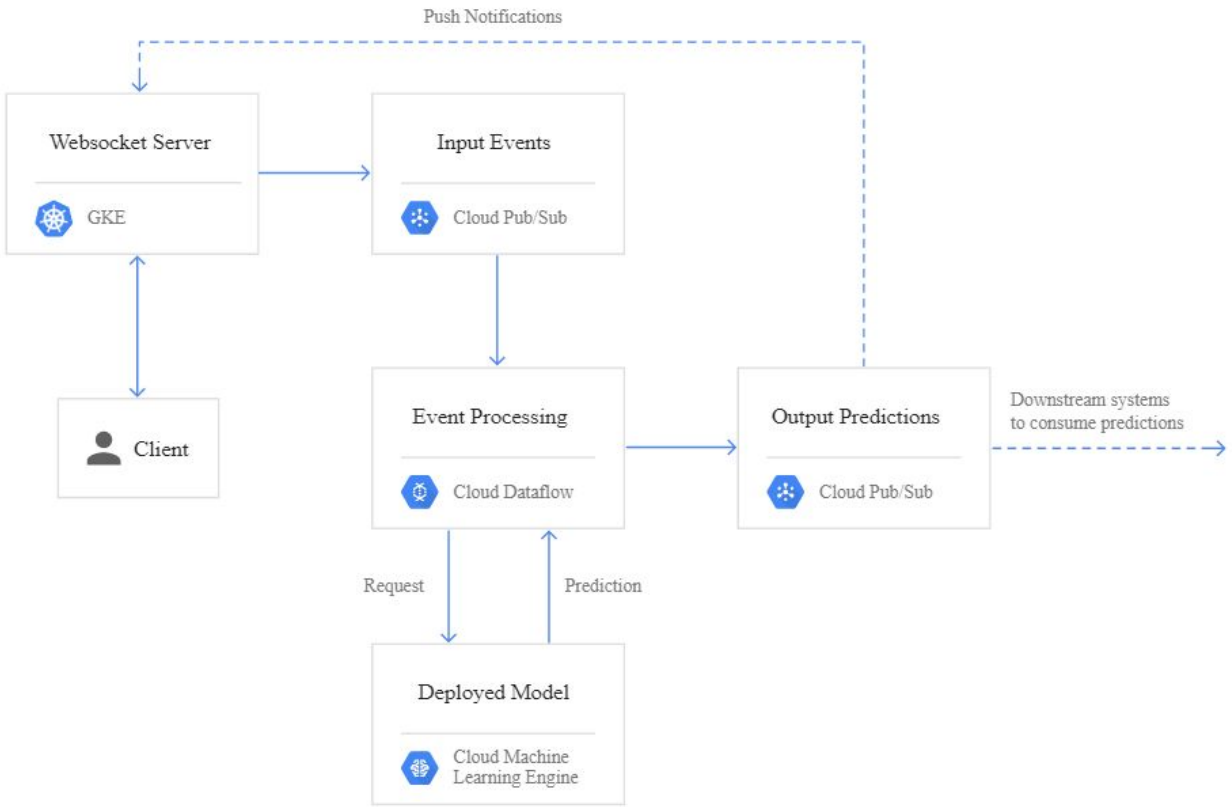


In real-time prediction:

1. Your online application sends HTTP requests to your ML model, which is deployed as a microservice and which exposes a REST API for prediction. You can deploy your model as an HTTP endpoint using AI Platform.
2. The caller application receives the response as soon as your model produces the prediction.

You might need to implement logic in your app backend as part of the system. For example, the app backend might need to perform preprocessing of the input data point before you invoke the ML model. Or the app backend might need to perform post-processing on the output prediction before sending the response back to the caller. This type of backend processing is an *ML gateway*, which acts as a wrapper for your ML model or models and your preprocessing and post-processing logic.

Online (Dataflow invoke model)



So there's a tradeoff between the model's predictive effectiveness and its prediction latency. And depending on the use case, you need to decide on two metrics:

1. The model's optimizing metric, which reflects the model's predictive effectiveness, like accuracy, precision, mean square error, and so on. The better the value of this metric, the better the model.

2. The model's satisficing metric, which reflects an operational constraint that the model needs to satisfy, such as prediction latency. You set a latency threshold to a particular value, such as 200 milliseconds, and any model that doesn't meet the threshold is not accepted. Another example of a satisficing metric is the size of the model, if you plan on deploying your model to low-spec hardware (like mobile and embedded devices).

Beside optimizing your model by balancing optimizing and satisficing metrics, you can use accelerators in the serving infrastructure to help improve the response time performance of the model:

- GPUs are optimized for parallel throughput. Several GPUs types are available across Google Cloud, including in Compute Engine, GKE, and AI Platform. For an in-depth look, see Running TensorFlow inference workloads at scale with TensorRT 5 and NVIDIA T4 GPUs.

- Cloud TPUs, a Google-built technology, are optimized for machine learning workloads. TPUs come as pods that can be used by Google Cloud products such as Compute Engine and AI Platform. Typically, you use TPUs only when you have large deep learning models and large batch sizes.

For an ML model to provide a prediction when given a data point, the data point must include all of the input features that the model expects. The expected features are the ones that are used to train the model. For example, if you train a model to estimate the price of a house given its size, location, age, number of rooms, and orientation, the trained model requires values for those features as inputs in order to provide an estimated price. However, in many use cases, the caller of your model does not provide these input feature values; instead, they're read in real time from a data store.

There are two types of input features that are fetched in real time to invoke the model for prediction:

- **Static reference features**. These feature values are static or slowly changing attributes of the entity for which a prediction is needed.
- **Dynamic real-time features**. These feature values are dynamically captured and computed based on real-time events.

In practice, online prediction use cases include a mix of user-supplied features, static reference features, and real-time computed features.

Diagram mismatches text

Text says Pub/sub to client and Datastore to downstream

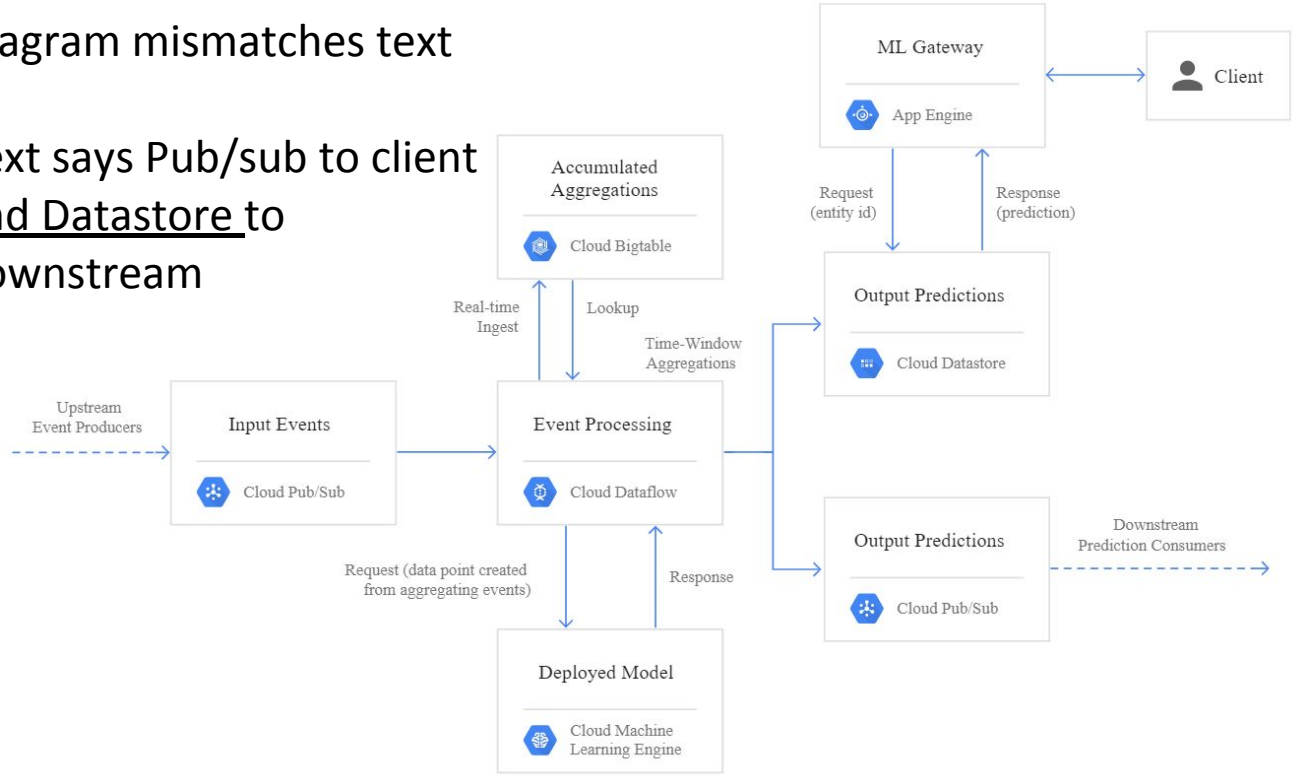


Figure 5. A high-level architecture for maintaining real-time data to be used for prediction

Handling dynamic real-time features

Real-time features are computed on the fly, typically in an event-stream processing pipeline. The difference between real-time features and the batch approach illustrated in Figure 3 is that for real-time features, you need a list of aggregated values for a particular window (fixed, sliding, or session) in a certain period of time, and not an overall aggregation of values within that period of time. Dynamic real-time data is relevant in use cases like the following:

- Predicting whether an engine will fail in the next hour, given real-time sensor data such as maximum, minimum, and average temperature; pressure; and vibration for each minute in the last half-hour.
- Recommending the next news article to read based on the list of last N viewed articles by the user during the current session.
- Estimating how long food delivery will take based on the list of incoming orders, as well as related data such as how many outstanding orders per minute have been made in the past hour.

Handling static reference features

Static features include descriptive attributes, such as customer demographic information. They also include summary attributes, such as customer purchase behavior indicators, like recency, frequency, and purchase total. Static reference data like this is relevant for the following use cases:

- Predicting the propensity of the customer to respond to a given service promotion, based on the customer demographic and historical purchase information, in order to display an ad.
- Estimating the price of a house based on the location of the house, including schools, shopping, and transportation, plus mean prices in the area.
- Recommending similar products given the attributes of the products that a customer is currently viewing.

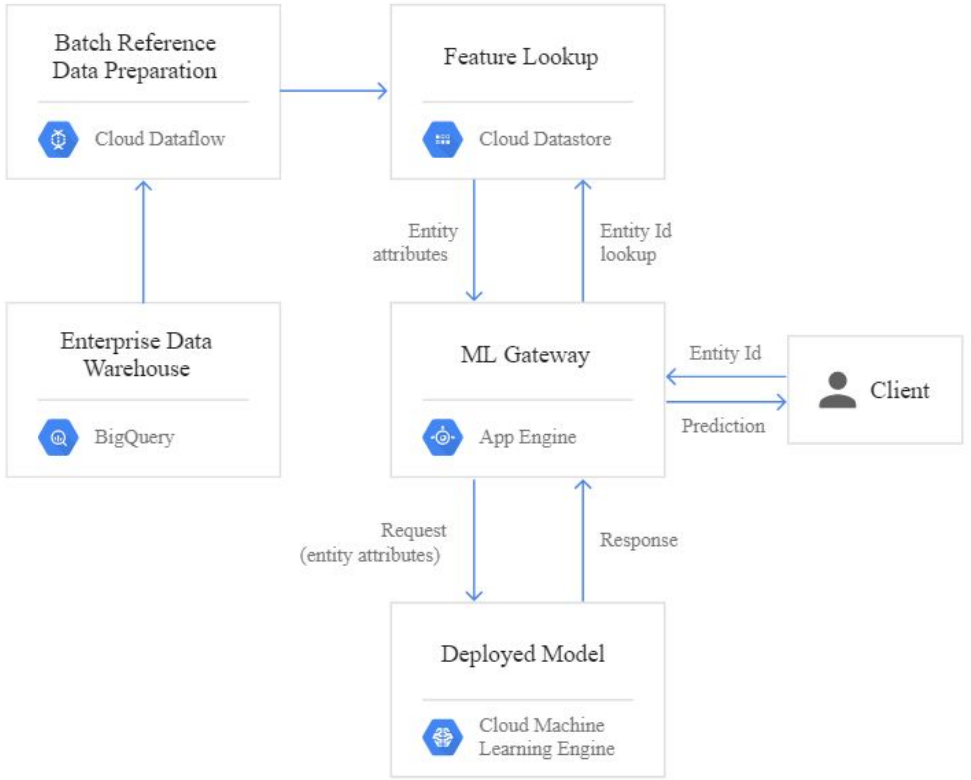


Figure 4. A high-level architecture for storing and serving reference data

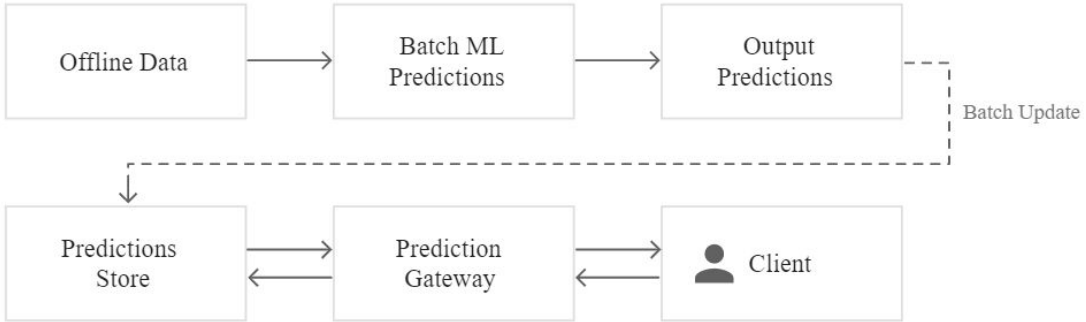
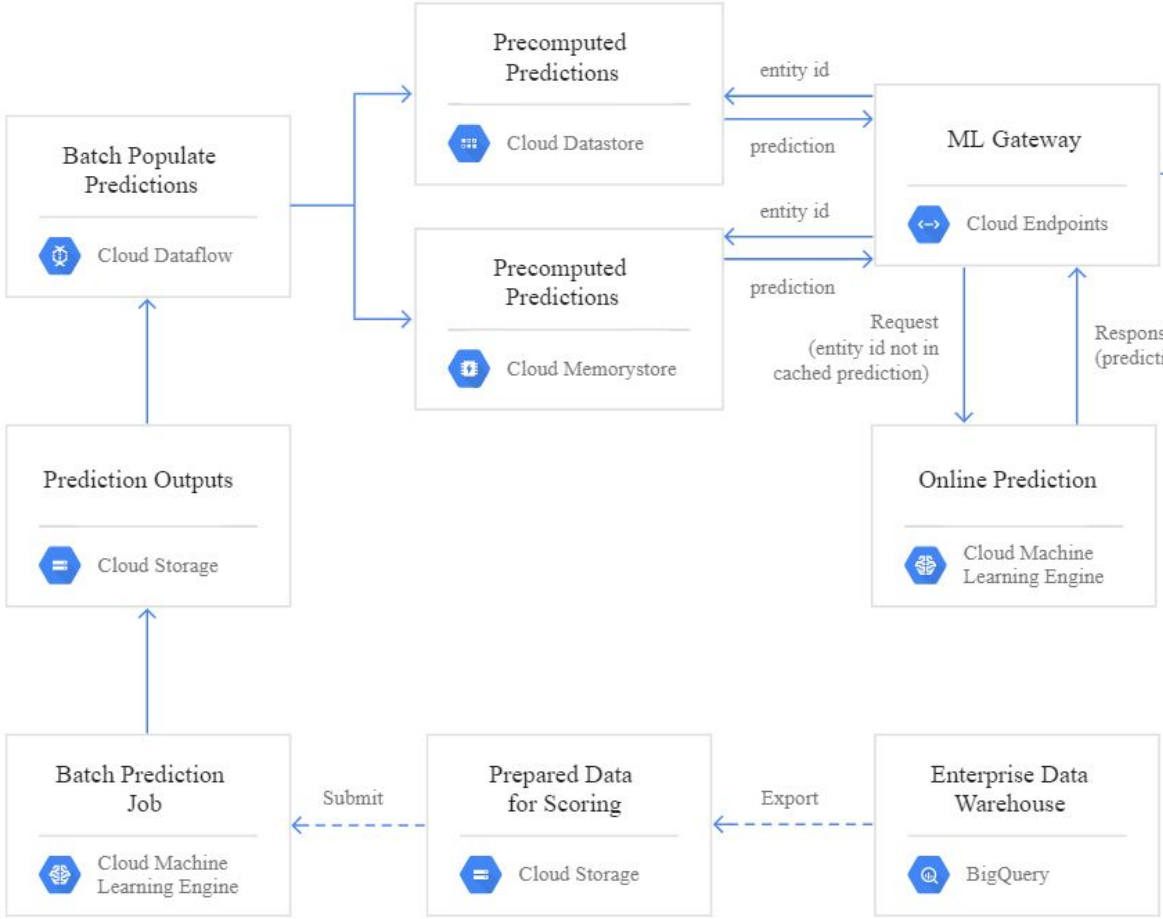


Figure 6. Functional architecture for precomputing and caching predictions for real-time serving

Prediction requests can be used for two categories of lookup keys:

- **Specific entity**. The predictions relate to a single entity based on an ID, like a customer, a product, a location, or a device. Use cases for specific-entity predictions include:
  - Preparing promotions, recommendations, or offers for a unique user ID at the beginning of the user's session.
  - Finding items (movies, articles, songs, and so on) that are similar to popular ones in order to produce related-item recommendations.
- **Specific combination of input features**. The predictions are for entities that are unique in time, and that cannot be based on a single entity ID. Instead, a combination of feature values defines the context for an entity. The key that represents that context is the unique combination of those feature values. Predictions are usually computed for frequent feature-value combinations. Use cases for specific feature combinations include:
  - Maximizing a bid price in real-time bidding for an incoming bid request. The bid request has a unique ID in time, but the request context (which might include a combination of website category, user taxonomy, and ad performance) can reoccur.
  - Predicting whether an anonymous or a new customer might buy something on your website based on a combination of location, number of products in the cart, and UTM data.



Handling predictions by entities

When you precompute predictions for specific entities, you might face the following situations:

- You have a manageable number of entities (low cardinality), which makes precomputing predictions for all entities possible. In that case, you probably want to preprocess the entities and save them all in a key-value store. An example is forecasting daily sales by store when you have hundreds or just a few thousand stores.
- You have too many entities (high cardinality), which makes it challenging to precompute prediction in a limited amount of time. An example is forecasting daily sales by item when you have hundreds of thousands or millions items. In that case, you can use a hybrid approach, where you precompute predictions for the top N entities, such as for the most active customers or the most viewed products. You can then use the model directly for online prediction for the rest of the long-tail entities.

Figure 7. A high-level architecture for precomputing and caching predictions for online serving using Datastore and Memorystore

Handling predictions for combinations of feature values

If you decide to precompute predictions not for a specific entity ID but for a combination of input feature values, you need the following:

- **Key**. A hashed combination of all possible input features. Combinations are not permutations, which means that you need to consistently build the key by using the features in the same order.
- **Value**. The prediction.

A simple example is to predict whether an anonymous or a new customer will buy something on your website. At the high level, you can start with features like those listed in the following table.

Feature name	Description	Example cardinality
`origin_continent`	Can vary between 5 and 9	6
`mobile`	yes or no	2
`category_most_visited`	Custom to your store inventory	10

You then do the following:

1. Decide the order of features when building your key string. For example, `europe_yes_female_sport` is the same prediction as `europe_female_sport_yes`, but the two keys result in a different hash value.
2. Compute predictions offline for all 120 possible combinations (6 x 2 x 10 in the example).
3. Store the predictions in a low read-latency database using the following:
    - The key, which could be something like `hash(europe_yes_female_sport)`.
    - The value of the prediction, such as `0.82345`.

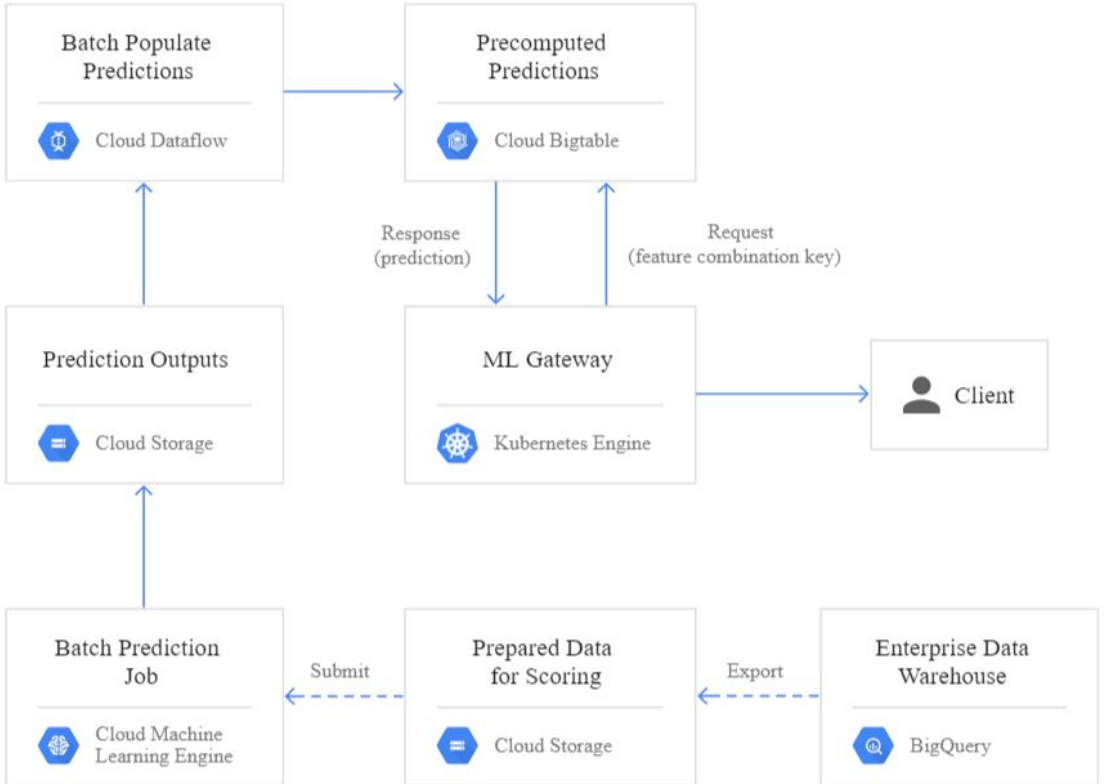


Figure 8. A high-level architecture for precomputing and caching predictions for online serving using Bigtable

- Creating all possible combinations of features can lead to millions or billions of records, depending on your data. Assuming that all features make sense as is and you need all combinations, storing them requires a scalable data store that can still serve single records with minimum latency.
- Continuous values such as `average_price` cannot be used as part of the possible combinations, because the possibilities could be infinite. Thus, you need to bucketize them appropriately, keeping in mind that it would impact your model effectiveness. The bucket size can be a hyperparameter.
- Some categorical features such as `postal_codes` can have a high cardinality, and if they're combined can create billions of possible combinations. You must decide whether each combination is relevant, or whether you could decrease the cardinality by using lower postcode resolution or using hash bucketing, without losing too much of your model effectiveness.

- **Memorystore**. Memorystore is a managed in-memory database. When you use its Redis offering, you can store intermediate data for submillisecond read access. Keys are binary-safe strings, and values can be of different data structures. Typical use cases for Memorystore are:
  - User-feature lookup in real-time bidding that requires submillisecond retrieval time.
  - Media and gaming applications that use precomputed predictions.
  - Storing intermediate data for a real-time data pipeline for creating input features.
- **Datastore**. Datastore is a fully-managed, scalable NoSQL document database built for automatic scaling, high performance, and ease of application development. Data objects in Datastore are known as *entities*. An entity has one or more named properties, in which you store the feature values required by your model or models. A typical use case for Datastore is a product recommendation system in an e-commerce site that's based on information about logged-in users.
- **Bigtable**. Bigtable is a massively scalable NoSQL database service engineered for high throughput and for low-latency workloads. It can handle petabytes of data, with millions of reads and writes per second at a latency that's on the order of milliseconds. The data is structured as a sorted key-value map. Bigtable scales linearly with the number of nodes. For more details, see Understanding Bigtable performance. Typical use cases for Bigtable are:
  - Fraud detection that leverages dynamically aggregated values. Applications in Fintech and Adtech are usually subject to heavy reads and writes.
  - Ad prediction that leverages dynamically aggregated values over all ad requests and historical data.
  - Booking recommendation based on the overall customer base's recent bookings.

In most of those cases, data in those stores has:

- A unique record key that refers to a user ID, machine ID, or other unique string that represents the item that the system needs to provide a prediction for.
- A feature value that can represent several data points, such as maximum temperature and average pressure in the last minute in an IoT context, or the count of bookings and page views in the last 10 minutes for a travel booking context. How the value is stored depends on the data store you choose.

To summarize, if you need:

- Submillisecond retrieval latency on a limited amount of quickly changing data, retrieved by a few thousand clients, use Memorystore.
- Millisecond retrieval latency on slowly changing data where storage scales automatically, use Datastore.
- Millisecond retrieval latency on dynamically changing data, using a store that can scale linearly with heavy reads and writes, use Bigtable.

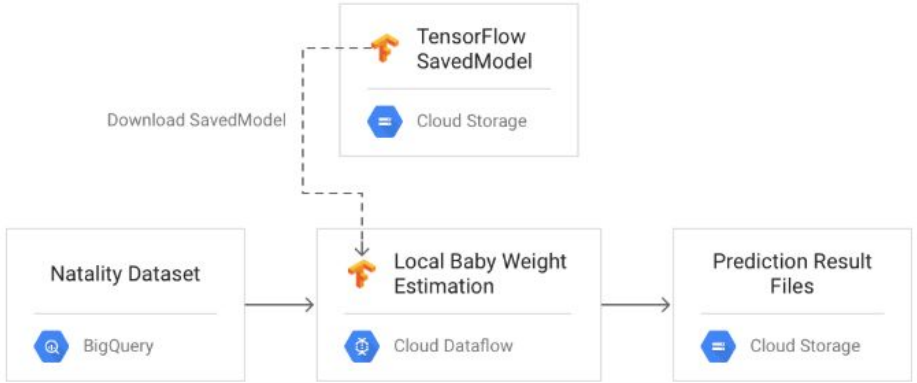
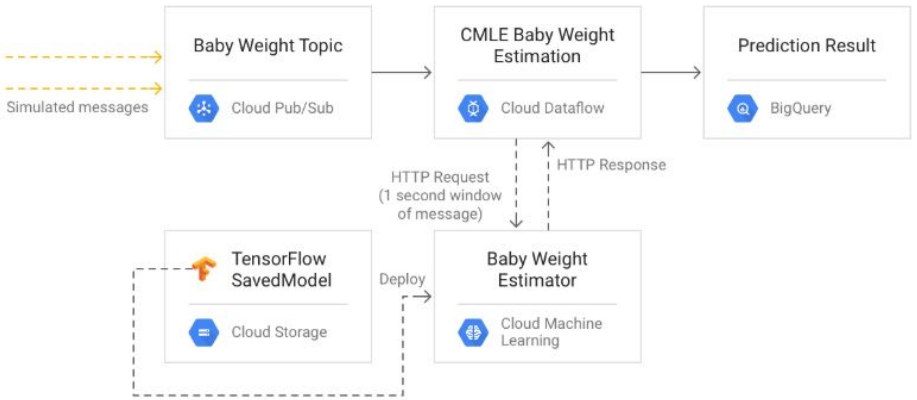


Figure 5. Batch Approach 1: Dataflow with direct model prediction



re 8. Stream Approach 1: Dataflow with AI Platform online prediction. The HTTP request might include a single data point or a group of data points in a micro-batch.

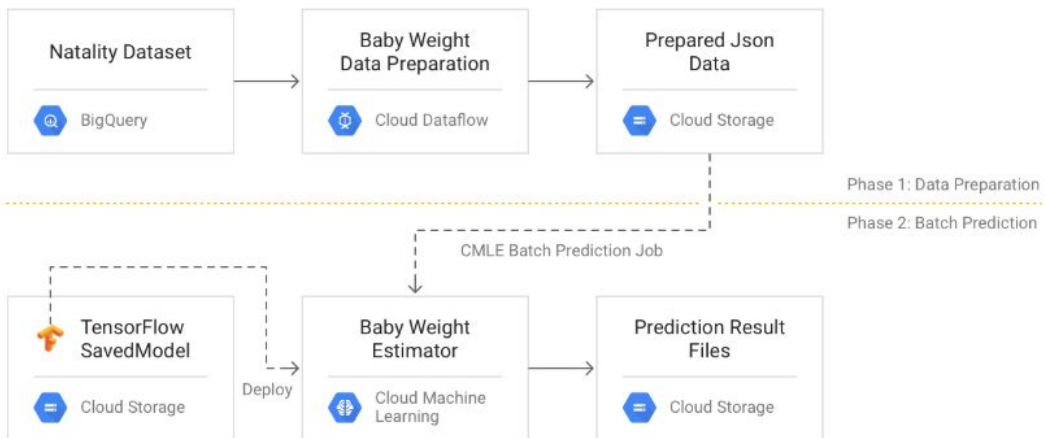
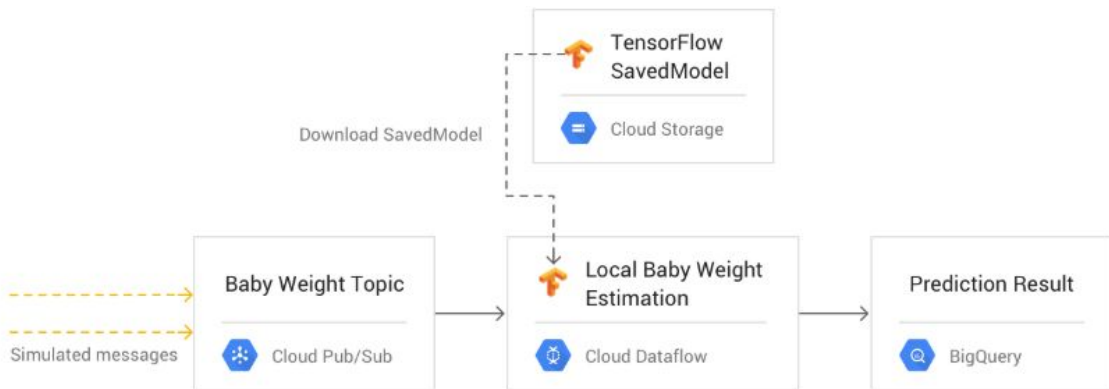
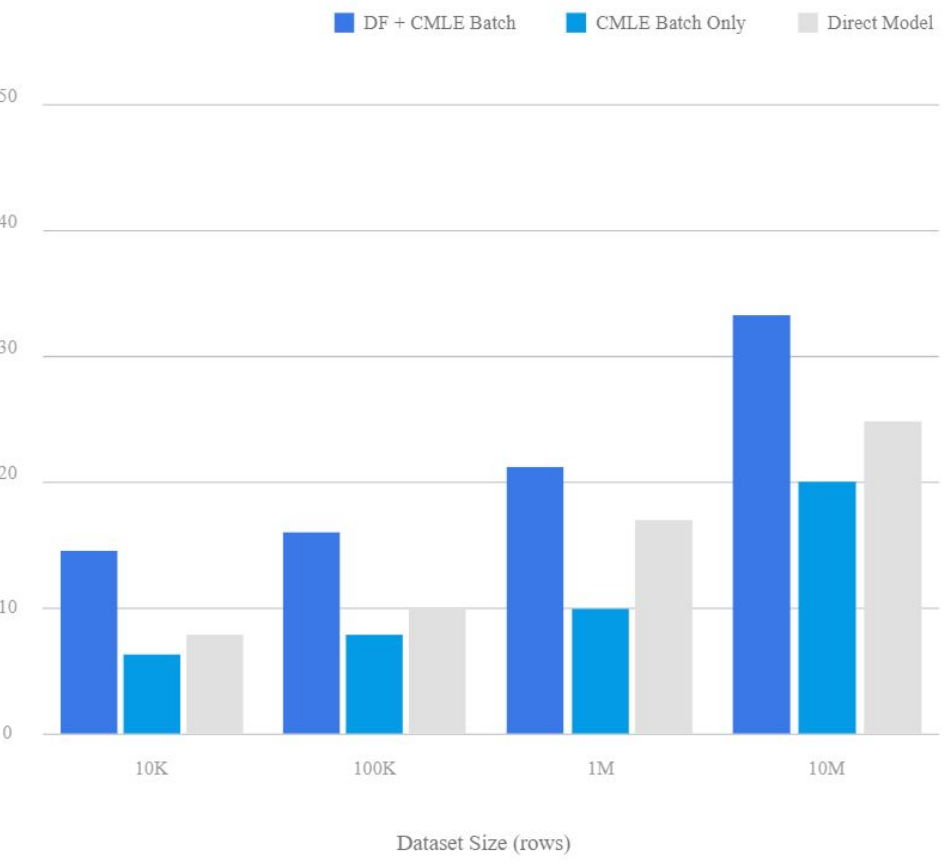


Figure 6. Batch Approach 2: Dataflow with AI Platform batch prediction

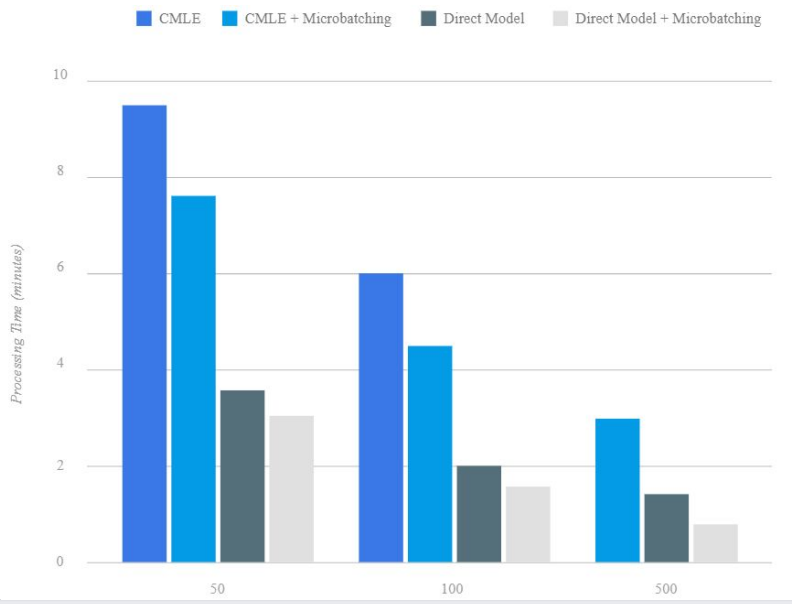


e 9. Stream approach 2: Dataflow with direct-model prediction



Batch

- Direct is faster for total time (can microbatch to improve)
- AI Platform phase in 2 phase is fast but have to prepare data to json first in phase 1 so total time longer

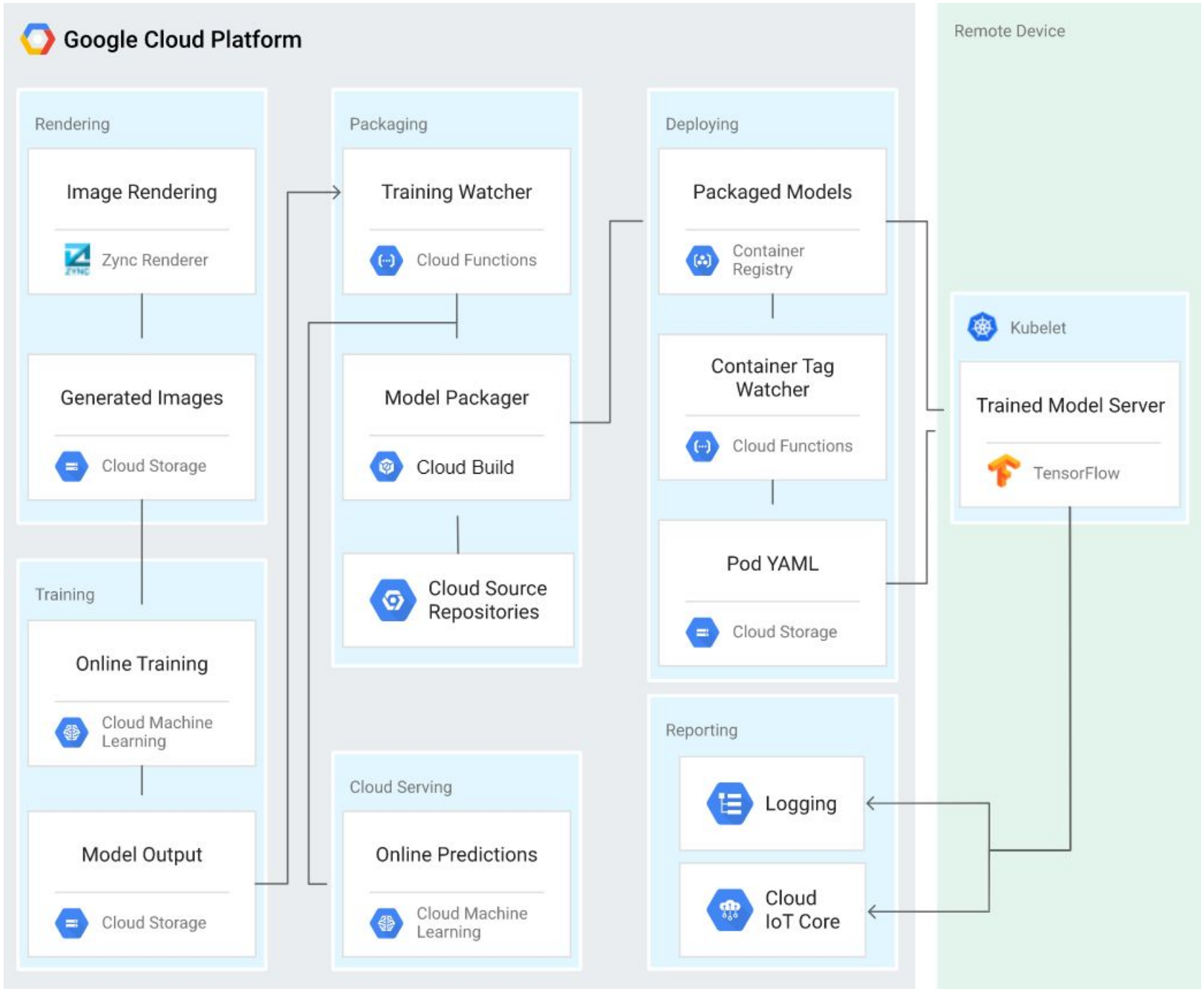


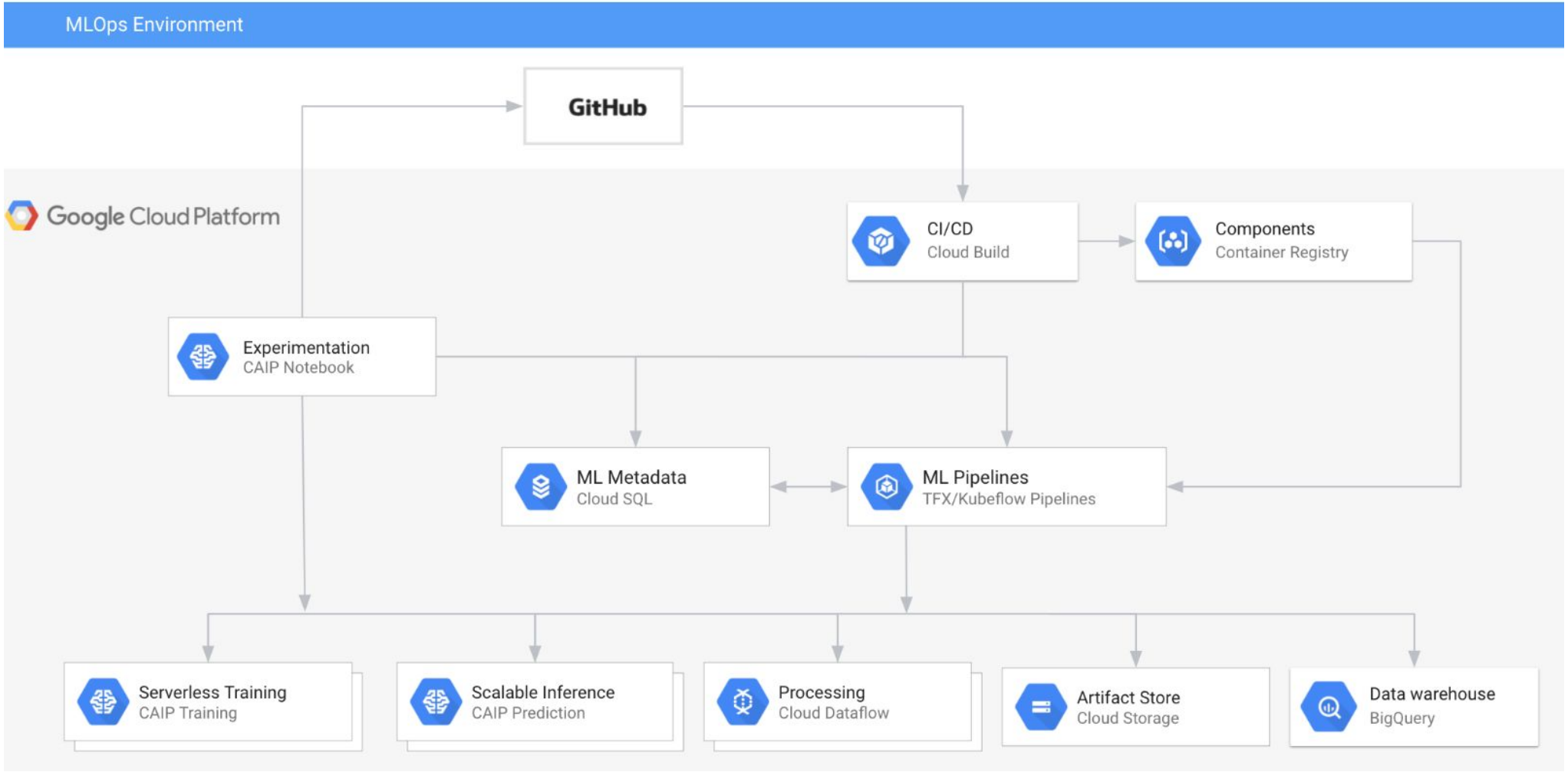
Stream

- Direct is faster
- Microbatch improves for any setup

Stream messages frequency	Metric	Dataflow with AI Platform online prediction		Dataflow with direct-model prediction	
		Single message	Micro-batching	Single message	Micro-batching
50 msg per sec (Simulation time: 3 min 20 sec)	Total time	9 min 34 sec	7 min 44 sec	3 min 43 sec	3 min 22 sec
100 msg per sec (Simulation time: 1 min 40 sec)	Total time	6 min 03 sec	4 min 34 sec	1 min 51 sec	1 min 41 sec
500 msg per sec (Simulation time: 20 sec)	Total time	NA - Default AI Platform Online Prediction Quota	2 min 47 sec	1 min 23 sec	48 sec

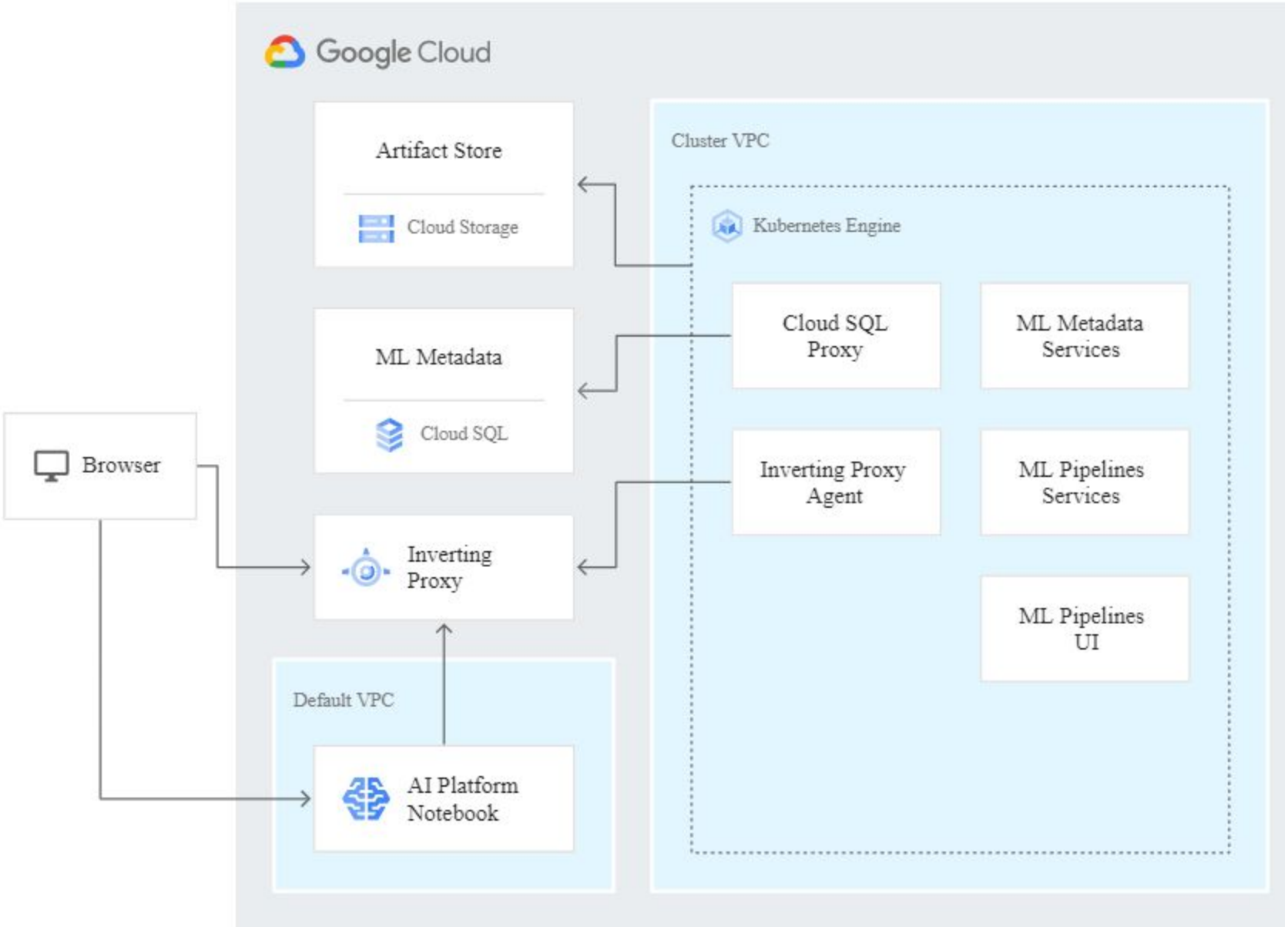
https://cloud.google.com/solutions/automating-iot-machine-learning



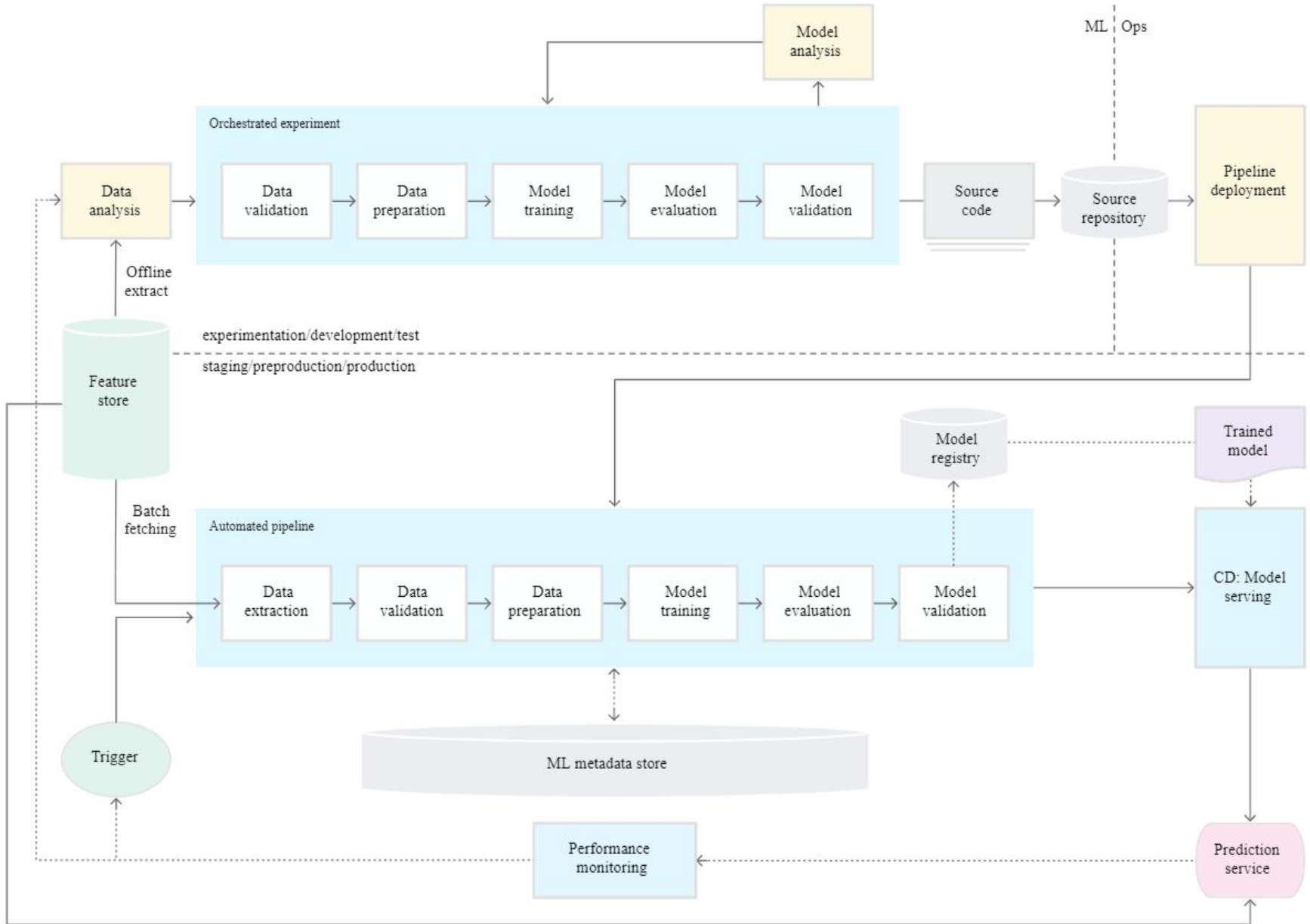


Note: AI Platform Pipelines makes it easier to get started with MLOps by saving you the difficulty of setting up Kubeflow Pipelines with TFX. However, AI Platform Pipelines deployment does not allow you to configure Cloud SQL as a backend for ML Metadata to host the metadata for the ML pipeline execution. Instead, MySQL runs on a GKE cluster. On the other hand, in the MLOps environment, ML Metadata is deployed as a managed Cloud SQL instance. This gives you better manageability, portability, and centralization of the metadata for ML.

https://cloud.google.com/solutions/machine-learning/setting-up-an-mlops-environment



MLOps level 1: Deploy whole training pipeline rather than trained model (level 0)



Data and model validation

When you deploy your ML pipeline to production, one or more of the triggers discussed in the Triggering the ML pipeline section automatically executes the pipeline. The pipeline expects new, live data to produce a new model version that is trained on the new data (as shown in Figure 3). Therefore, automated *data validation* and *model validation* steps are required in the production pipeline to ensure the following expected behavior:

- Data validation: This step is required before model training to decide whether you should retrain the model or stop the execution of the pipeline. This decision is automatically made if the following was identified by the pipeline.
  - Data schema skews: These skews are considered anomalies in the input data, which means that the downstream pipeline steps, including data processing and model training, receives data that doesn't comply with the expected schema. In this case, you should stop the pipeline so the data science team can investigate. The team might release a fix or an update to the pipeline to handle these changes in the schema. Schema skews include receiving unexpected features, not receiving all the expected features, or receiving features with unexpected values.
  - Data values skews: These skews are significant changes in the statistical properties of data, which means that data patterns are changing, and you need to trigger a retraining of the model to capture these changes.
- Model validation: This step occurs after you successfully train the model given the new data. You evaluate and validate the model before it's promoted to production. This *offline model validation* step consists of the following.
  - Producing evaluation metric values using the trained model on a test dataset to assess the model's predictive quality.
  - Comparing the evaluation metric values produced by your newly trained model to the current model, for example, production model, baseline model, or other business-requirement models. You make sure that the new model produces better performance than the current model before promoting it to production.
  - Making sure that the performance of the model is consistent on various segments of the data. For example, your newly trained customer churn model might produce an overall better predictive accuracy compared to the previous model, but the accuracy values per customer region might have large variance.
  - Making sure that you test your model for deployment, including infrastructure compatibility and consistency with the prediction service API.

In addition to offline model validation, a newly deployed model undergoes *online model validation*—in a canary deployment or an A/B testing setup—before it serves prediction for the online traffic.

Feature store

An optional additional component for level 1 ML pipeline automation is a feature store. A feature store is a centralized repository where you standardize the definition, storage, and access of features for training and serving. A feature store needs to provide an API for both high-throughput batch serving and low-latency real-time serving for the feature values, and to support both training and serving workloads.

The feature store helps data scientists do the following:

- Discover and reuse available feature sets for their entities, instead of re-creating the same or similar ones.
- Avoid having similar features that have different definitions by maintaining features and their related metadata.
- Serve up-to-date feature values from the feature store.
- Avoid training-serving skew by using the feature store as the data source for experimentation, continuous training, and online serving. This approach makes sure that the features used for training are the same ones used during serving:
  - For experimentation, data scientists can get an offline extract from the feature store to run their experiments.
  - For continuous training, the automated ML training pipeline can fetch a batch of the up-to-date feature values of the dataset that are used for the training task.
  - For online prediction, the prediction service can fetch in a batch of the feature values related to the requested entity, such as customer demographic features, product features, and current session aggregation features.

Metadata management

Information about each execution of the ML pipeline is recorded in order to help with data and artifacts lineage, reproducibility, and comparisons. It also helps you debug errors and anomalies. Each time you execute the pipeline, the ML metadata store records the following metadata:

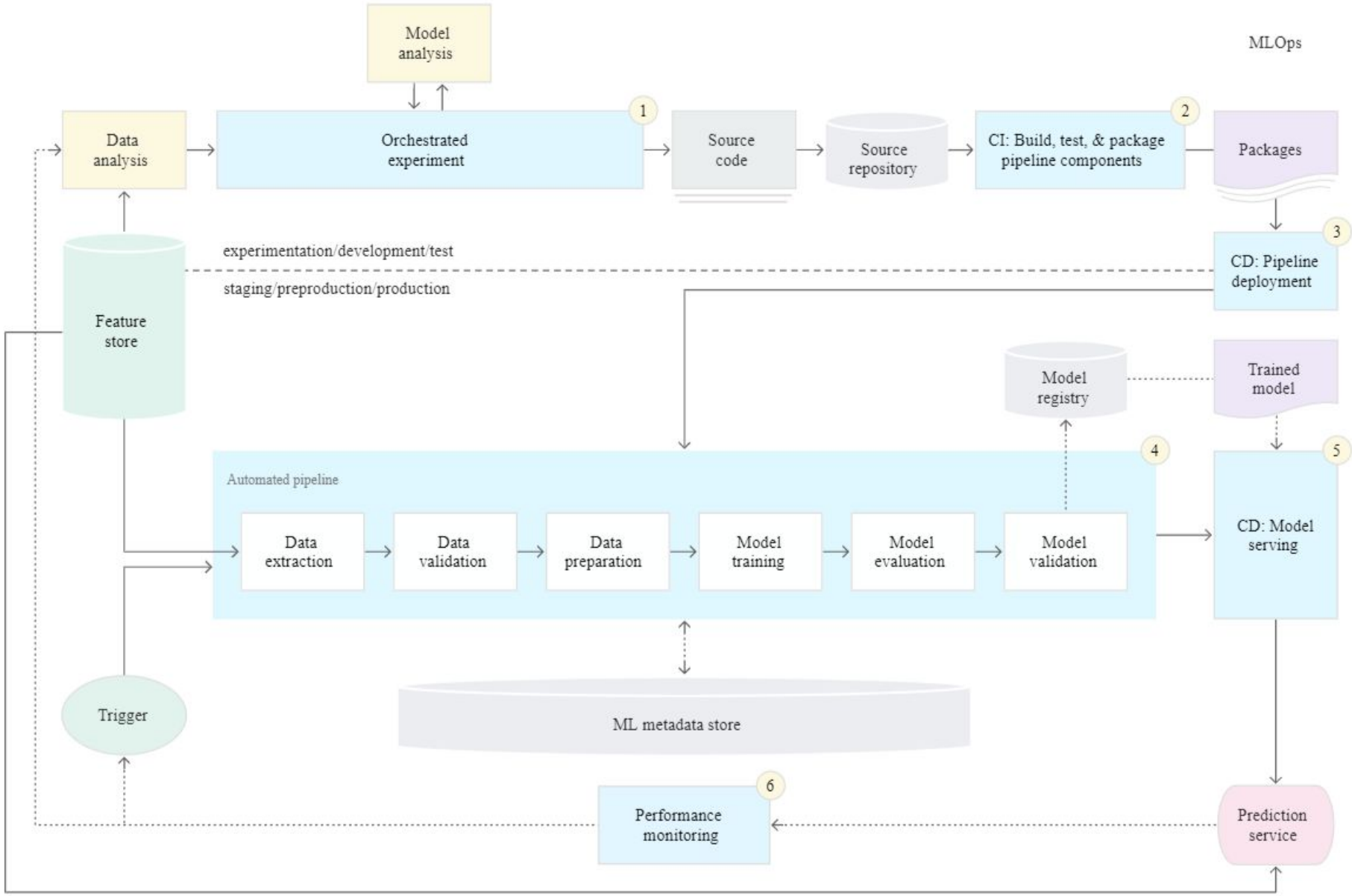
- The pipeline and component versions that were executed.
- The start and end date, time, and how long the pipeline took to complete each of the steps.
- The executor of the pipeline.
- The parameter arguments that were passed to the pipeline.
- The pointers to the artifacts produced by each step of the pipeline, such as the location of prepared data, validation anomalies, computed statistics, and extracted vocabulary from the categorical features. Tracking these intermediate outputs helps you resume the pipeline from the most recent step if the pipeline stopped due to a failed step, without having to re-execute the steps that have already completed.
- A pointer to the previous trained model if you need to roll back to a previous model version or if you need to produce evaluation metrics for a previous model version when the pipeline is given new test data during the model validation step.
- The model evaluation metrics produced during the model evaluation step for both the training and the testing sets. These metrics help you compare the performance of a newly trained model to the recorded performance of the previous model during the model validation step.

ML pipeline triggers

You can automate the ML production pipelines to retrain the models with new data, depending on your use case:

- On demand: Ad-hoc manual execution of the pipeline.
- On a schedule: New, labelled data is systematically available for the ML system on a daily, weekly, or monthly basis. The retraining frequency also depends on how frequently the data patterns change, and how expensive it is to retrain your models.
- On availability of new training data: New data isn't systematically available for the ML system and instead is available on an ad-hoc basis when new data is collected and made available in the source databases.
- On model performance degradation: The model is retrained when there is noticeable performance degradation.
- On significant changes in the data distributions (*concept drift*). It's hard to assess the complete performance of the online model, but you notice significant changes on the data distributions of the features that are used to perform the prediction. These changes suggest that your model has gone stale, and that needs to be retrained on fresh data.

MLOps level 2: Pipeline is automatically build, test, package, deployed rather than manual (level 1)



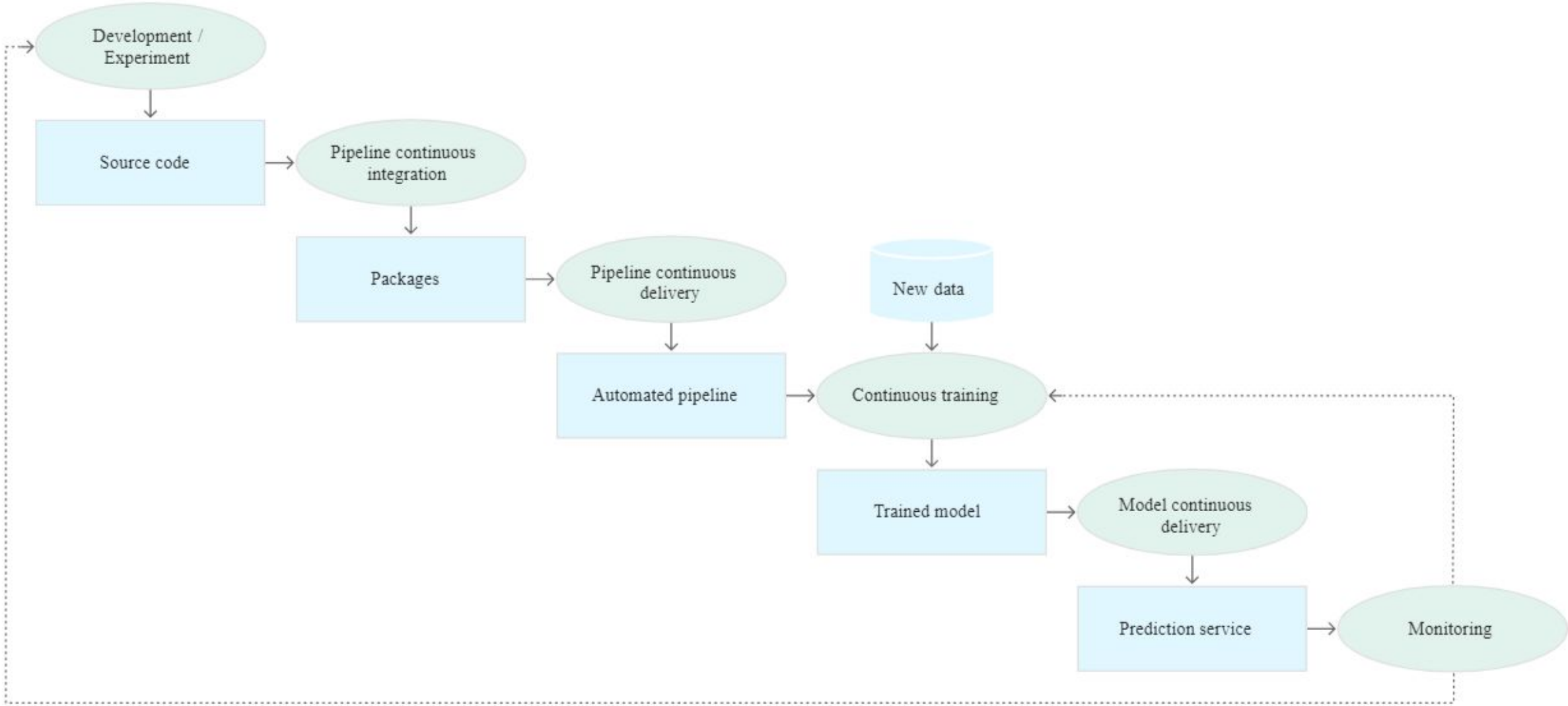


Figure 5. Stages of the CI/CD automated ML pipeline.

Continuous integration

In this setup, the pipeline and its components are built, tested, and packaged when new code is committed or pushed to the source code repository. Besides building packages, container images, and executables, the CI process can include the following tests:

- Unit testing your feature engineering logic.
- Unit testing the different methods implemented in your model. For example, you have a function that accepts a categorical data column and you encode the function as a one-hot feature.
- Testing that your model training converges (that is, the loss of your model goes down by iterations and overfits a few sample records).
- Testing that your model training doesn't produce NaN values due to dividing by zero or manipulating small or large values.
- Testing that each component in the pipeline produces the expected artifacts.
- Testing integration between pipeline components.

Continuous delivery

In this level, your system continuously delivers new pipeline implementations to the target environment that in turn delivers prediction services of the newly trained model. For rapid and reliable continuous delivery of pipelines and models, you should consider the following:

- Verifying the compatibility of the model with the target infrastructure before you deploy your model. For example, you need to verify that the packages that are required by the model are installed in the serving environment, and that the memory, compute, and accelerator resources that are available.
- Testing the prediction service by calling the service API with the expected inputs, and making sure that you get the response that you expect. This test usually captures problems that might occur when you update the model version and it expects a different input.
- Testing prediction service performance, which involves load testing the service to capture metrics such as queries per seconds (QPS) and model latency.
- Validating the data either for retraining or batch prediction.
- Verifying that models meet the predictive performance targets before they are deployed.
- Automated deployment to a test environment, for example, a deployment that is triggered by pushing code to the development branch.
- Semi-automated deployment to a pre-production environment, for example, a deployment that is triggered by merging code to the master branch after reviewers approve the changes.
- Manual deployment to a production environment after several successful runs of the pipeline on the pre-production environment.

L_1 vs. L_2 regularization.

L_2 and L_1 penalize weights differently:

- L_2 penalizes $weight^2$.
- L_1 penalizes $|weight|$.

Consequently, L_2 and L_1 have different derivatives:

- The derivative of L_2 is 2 * *weight*.
- The derivative of L_1 is *k* (a constant, whose value is independent of weight).

You can think of the derivative of L_2 as a force that removes x% of the weight every time. As Zeno knew, even if you remove x percent of a number *billions of times*, the diminished number will still never quite reach zero. (Zeno was less familiar with floating-point precision limitations, which could possibly produce exactly zero.) At any rate, L_2 does not normally drive weights to zero.

You can think of the derivative of L_1 as a force that subtracts some constant from the weight every time. However, thanks to absolute values, L_1 has a discontinuity at 0, which causes subtraction results that cross 0 to become zeroed out. For example, if subtraction would have forced a weight from +0.1 to -0.2, L_1 will set the weight to exactly 0. Eureka, L_1 zeroed out the weight.

Vanishing Gradients

The gradients for the lower layers (closer to the input) can become very small. In deep networks, computing these gradients can involve taking the product of many small terms.

When the gradients vanish toward 0 for the lower layers, these layers train very slowly, or not at all.

The ReLU activation function can help prevent vanishing gradients.

Exploding Gradients

If the weights in a network are very large, then the gradients for the lower layers involve products of many large terms. In this case you can have exploding gradients: gradients that get too large to converge.

Batch normalization can help prevent exploding gradients, as can lowering the learning rate.

Dead ReLU Units

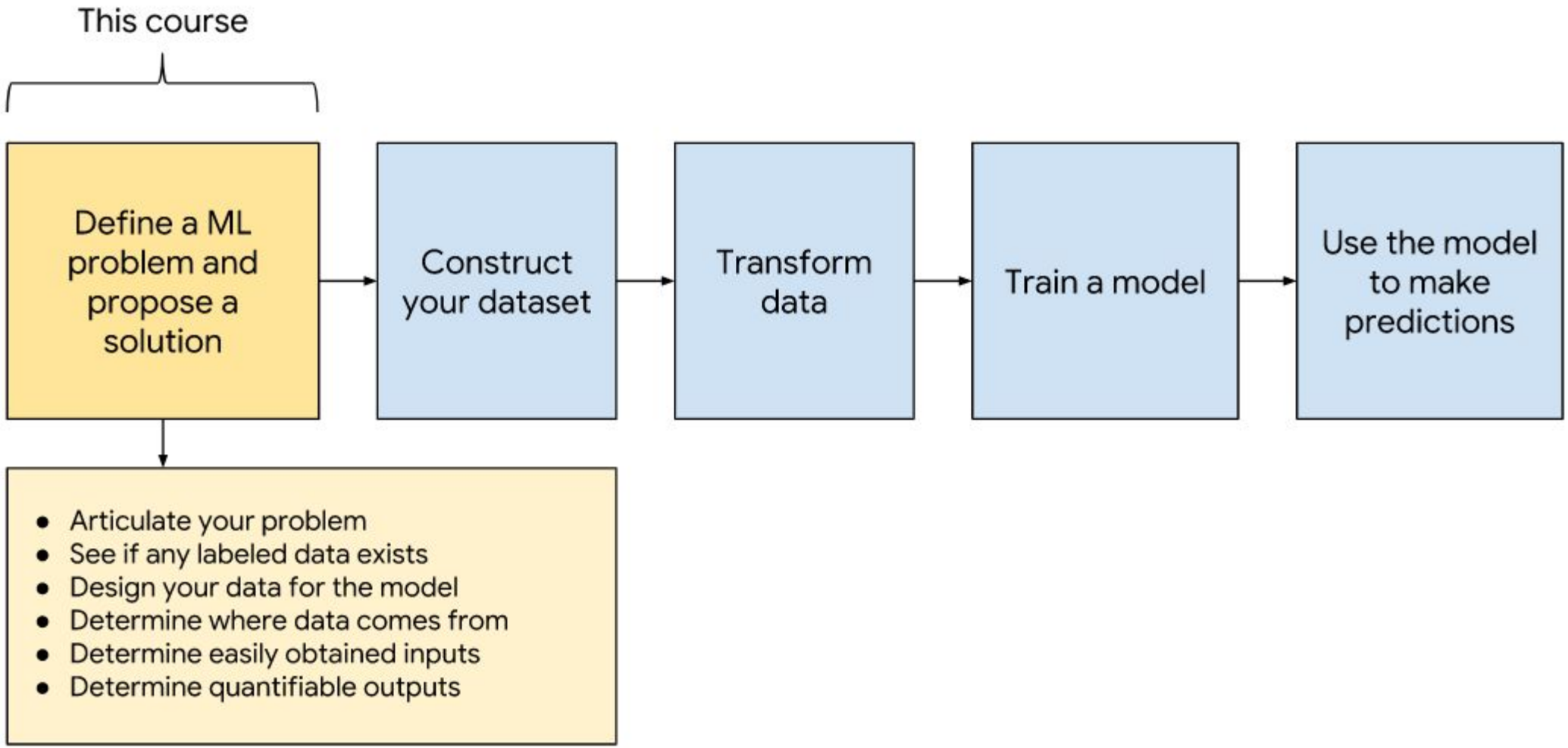
Once the weighted sum for a ReLU unit falls below 0, the ReLU unit can get stuck. It outputs 0 activation, contributing nothing to the network's output, and gradients can no longer flow through it during backpropagation. With a source of gradients cut off, the input to the ReLU may not ever change enough to bring the weighted sum back above 0.

Lowering the learning rate can help keep ReLU units from dying.

https://www.tensorflow.org/extras/candidate_sampling.pdf

candidate sampling

A training-time optimization in which a probability is calculated for all the positive labels, using, for example, **softmax**, but only for a random sample of negative labels. For example, if we have an example labeled *beagle* and *dog* candidate sampling computes the predicted probabilities and corresponding loss terms for the *beagle* and *dog* class outputs in addition to a random subset of the remaining classes (*cat*, *lollipop*, *fence*). The idea is that the **negative classes** can learn from less frequent negative reinforcement as long as **positive classes** always get proper positive reinforcement, and this is indeed observed empirically. The motivation for candidate sampling is a computational efficiency win from not computing predictions for all negatives.


This course
Define a ML problem and propose a solution
Construct your dataset
Transform data
Train a model
Use the model to make predictions
Articulate your problem
See if any labeled data exists
Design your data for the model
Determine where data comes from
Determine easily obtained inputs
Determine quantifiable outputs

What is Your Ideal Outcome?

Adding your ML model to your system should produce a desirable outcome. This outcome may be quite different from how you assess the model and its quality.

Examples	
Transcoding	Our ideal outcome is to use fewer resources on transcoding less popular videos. Transcoding is the process of converting the user's uploaded content into a more efficient format that YouTube shows to users. More than 1/3 of videos on YouTube are watched fewer than 10 times! If we can identify these videos and only prepare lower resolution versions of them, we could save a lot of resources.
Video Recommendations	Our ideal outcome is to suggest videos that people find useful, entertaining, and worth their time.

Prediction	Decision
What video the learner wants to watch next.	Show those videos in the recommendation bar.
Probability someone will click on a search result.	If P(click) > 0.12, prefetch the web page.
What fraction of a video ad the user will watch.	If a small fraction, don't show the user the ad.

Output	Ideal Outcome
Predict whether the user will share the article.	Show the user articles they will like.
Predict whether the video will be popular.	Suggest videos that people find useful and worth their time.
Predict whether the user will install an app from an app store.	Find apps the user will use often and enjoy.
Make sure there is a strong connection between the two!	

Think back to our model that predicts the next video someone will watch on YouTube. Expand each section below by clicking the plus icon to learn why each successive proposed objective is not good for this problem.

Maximize Click Rate

Users may click on something but then not stay on it very long. This optimizes clickbait, so maybe we should try something else.

Maximize Watch Time

Users may watch a long time, but then exit the session.

Example
A Minecraft video gets a 0.1% audience that watches video for 3 hours, 8% of whom watch another video for 5 minutes, while the rest quit watching altogether. The system is maximizing watch time, so the users' "watch next" list will consist solely of long Minecraft videos.

It is important to note that multiple short watches can be just as good as one long watch and can even increase the overall session watch time. On to the next objective!

Maximize Session Watch Time

This model still favors longer videos, which is still a problem. This model does mine particular interests really well. For example, if a user watches a video of LeBron James dunking, the system will show them every LeBron dunking video. Ever. The video recommendation system is *really* good at that, but user experience suffers. Each person can see YouTube as the place to go to watch a specific type of video. Diversity suffers.

Increase Diversity & Maximize Session Watch Time

What problems do you think might arise from this objective? Keep in mind Goodhart's law , "When a measure becomes a target, it ceases to be a good measure."

This section is a guide to the suggested approach for framing an ML problem:

1. Articulate your problem.
2. Start simple.
3. Identify Your Data Sources.
4. Design your data for the model.
5. Determine where data comes from.
6. Determine easily obtained inputs.
7. Ability to Learn.
8. Think About Potential Bias.

Our problem is best framed as 3-class, single-label classification, which predicts whether a video will be in one of three classes—`{very popular, somewhat popular, not popular}`—28 days after being uploaded.

> We will predict whether an uploaded video is likely to become popular or not (binary classification).
>
> We will predict an uploaded video's popularity in terms of the number of views it will receive within a 28 day window (regression).

Our data set consists of 100,000 examples about past uploaded videos with popularity data and video descriptions.

Title	Channel	Upload Time	Uploader's Recent Videos	Output (label)
My silly cat	Alice	2018-03-21 08:00	Another cat video, yet another cat	Very popular
A snake video	Bob	2018-04-03 12:00	None	Not popular

We applied the labels `{very popular, somewhat popular,not popular}` to each video that fell within a determined range of views and "thumbs ups" and determined keyword descriptions for each video. Hand-generating descriptions is not sustainable, so we are considering adding a keyword description to the upload form.

> Consider the engineering cost to develop a data pipeline to prepare the inputs, and the expected benefit of having each input in the model. Focus on inputs that can be obtained from a single system with a simple pipeline. Start with the minimum possible infrastructure.

Will the ML model be able to learn? List aspects of your problem that might cause difficulty learning. For example:

- The data set doesn't contain enough positive labels.
- The training data doesn't contain enough examples.
- The labels are too noisy.
- The system memorizes the training data, but has difficulty generalizing to new cases.

Example

The measure "popular" is subjective based on the audience and inconsistent across video genres. Tastes change over time, so today's "popular" video might be tomorrow's "not popular" video.

Since the measure "popular" is subjective, it is possible that the model will serve popular videos that reinforce unfair or biased societal views.

Reasons for Data Transformation

We transform features primarily for the following reasons:

1. **Mandatory transformations** for data compatibility. Examples include:
    - Converting non-numeric features into numeric. You can't do matrix multiplication on a string, so we must convert the string to some numeric representation.
    - Resizing inputs to a fixed size. Linear models and feed-forward neural networks have a fixed number of input nodes, so your input data must always have the same size. For example, image models need to reshape the images in their dataset to a fixed size.
2. **Optional quality transformations** that may help the model perform better. Examples include:
    - Tokenization or lower-casing of text features.
    - Normalized numeric features (most models perform better afterwards).
    - Allowing linear models to introduce non-linearities into the feature space.

Strictly speaking, quality transformations are not necessary--your model could still run without them. But using these techniques may enable the model to give better results.

Transforming prior to training

In this approach, we perform the transformation before training. This code lives separate from your machine learning model.

👍 Pros

- Computation is performed only once.
- Computation can look at entire dataset to determine the transformation.

👎 Cons

- Transformations need to be reproduced at prediction time. Beware of skew!
- Any transformation changes require rerunning data generation, leading to slower iterations.

Skew is more dangerous for cases involving online serving. In offline serving, you might be able to reuse the code that generates your training data. In online serving, the code that creates your dataset and the code used to handle live traffic are almost necessarily different, which makes it easy to introduce skew.

Transforming within the model

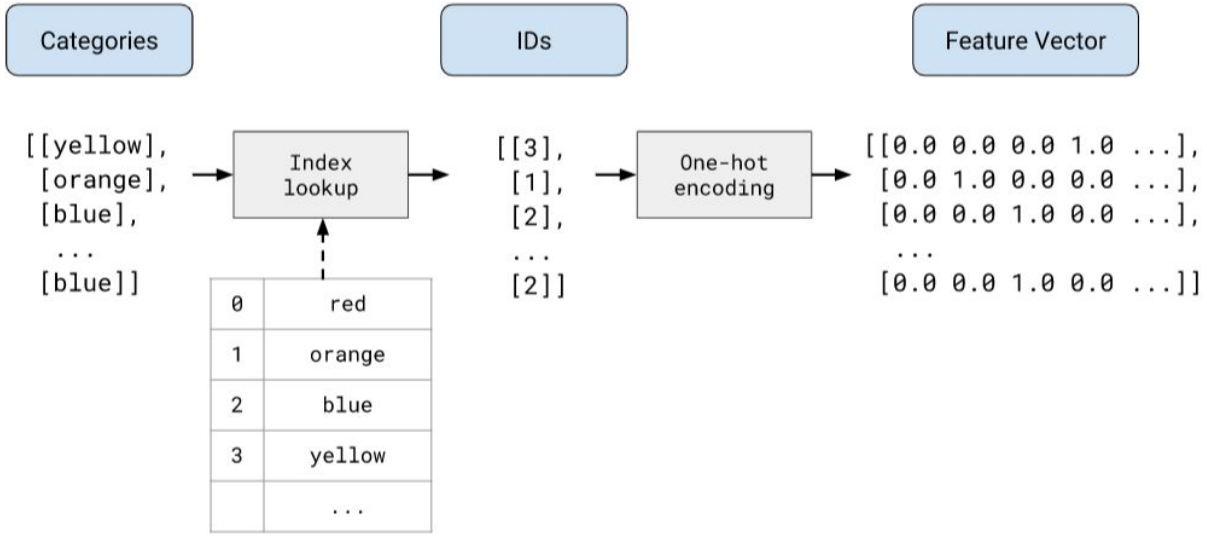
For this approach, the transformation is part of the model code. The model takes in untransformed data as input and will transform it within the model.

👍 Pros

- Easy iterations. If you change the transformations, you can still use the same data files.
- You're guaranteed the same transformations at training and prediction time.

👎 Cons

- Expensive transforms can increase model latency.
- Transformations are per batch.



Note about sparse representation

If your categories are the days of the week, you might, for example, end up representing Friday with the feature vector [0, 0, 0, 0, 1, 0, 0]. However, most implementations of ML systems will represent this vector in memory with a sparse representation. A common representation is a list of non-empty values and their corresponding indices—for example, 1.0 for the value and [4] for the index. This allows you to spend less memory storing a huge amount of 0s and allows more efficient matrix multiplication. In terms of the underlying math, the [4] is equivalent to [0, 0, 0, 0, 1, 0, 0].

Out of Vocab (OOV)

Just as numerical data contains outliers, categorical data does, as well. For example, consider a data set containing descriptions of cars. One of the features of this data set could be the car's color. Suppose the common car colors (black, white, gray, and so on) are well represented in this data set and you make each one of them into a category so you can learn how these different colors affect value. However, suppose this data set contains a small number of cars with eccentric colors (mauve, puce, avocado). Rather than giving each of these colors a separate category, you could lump them into a catch-all category called **Out of Vocab** (**OOV**). By using OOV, the system won't waste time training on each of those rare colors.

Hashing

Another option is to hash every string (category) into your available index space. Hashing often causes collisions, but you rely on the model learning some shared representation of the categories in the same index that works well for the given problem.

For important terms, hashing can be worse than selecting a vocabulary, because of collisions. On the other hand, hashing doesn't require you to assemble a vocabulary, which is advantageous if the feature distribution changes heavily over time.

Hybrid of Hashing and Vocabulary

You can take a hybrid approach and combine hashing with a vocabulary. Use a vocabulary for the most important categories in your data, but replace the OOV bucket with multiple OOV buckets, and use hashing to assign categories to buckets.

The categories in the hash buckets must share an index, and the model likely won't make good predictions, but we have allocated some amount of memory to attempt to learn the categories outside of our vocabulary.

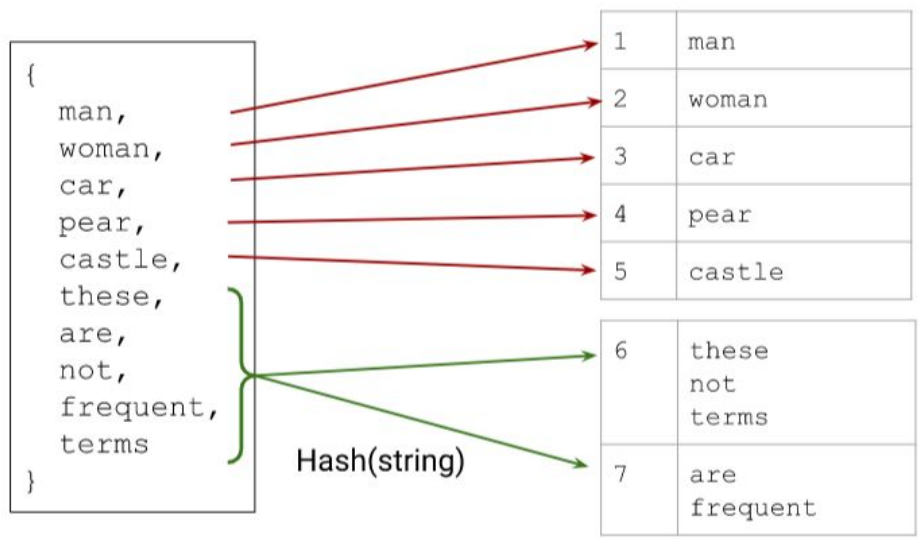


Figure 5: Hybrid approach combining vocabulary and hashing.

https://developers.google.com/machine-learning/crash-course/embeddings/translating-to-a-lower-dimensional-space

Embeddings as lookup tables

An embedding is a matrix in which each column is the vector that corresponds to an item in your vocabulary. To get the dense vector for a single vocabulary item, you retrieve the column corresponding to that item.

But how would you translate a sparse bag of words vector? To get the dense vector for a sparse vector representing multiple vocabulary items (all the words in a sentence or paragraph, for example), you could retrieve the embedding for each individual item and then add them together.

If the sparse vector contains counts of the vocabulary items, you could multiply each embedding by the count of its corresponding item before adding it to the sum.

Embedding lookup as matrix multiplication

The lookup, multiplication, and addition procedure we've just described is equivalent to matrix multiplication. Given a 1 X N sparse representation S and an N X M embedding table E, the matrix multiplication S X E gives you the 1 X M dense vector.

https://developers.google.com/machine-learning/glossary#embeddings

embeddings

A categorical feature represented as a continuous-valued feature. Typically, an embedding is a translation of a high-dimensional vector into a low-dimensional space. For example, you can represent the words in an English sentence in either of the following two ways:

- As a million-element (high-dimensional) **sparse vector** in which all elements are integers. Each cell in the vector represents a separate English word; the value in a cell represents the number of times that word appears in a sentence. Since a single English sentence is unlikely to contain more than 50 words, nearly every cell in the vector will contain a 0. The few cells that aren't 0 will contain a low integer (usually 1) representing the number of times that word appeared in the sentence.
- As a several-hundred-element (low-dimensional) **dense vector** in which each element holds a floating-point value between 0 and 1. This is an embedding.

Recall from the Machine Learning Crash Course that an **embedding** is a categorical feature represented as a continuous-valued feature. Deep models frequently convert the indices from an index to an embedding.

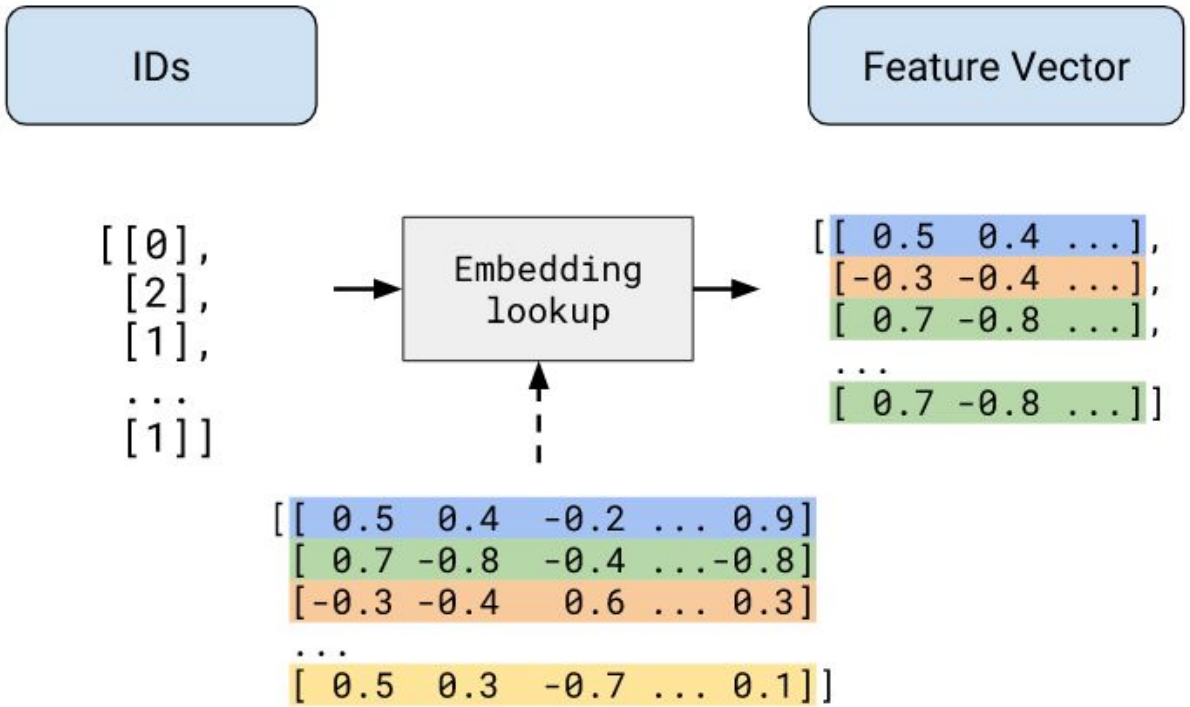


Figure 6: Sparse feature vectors via embedding

The other transformations we've discussed could be stored on disk, but embeddings are different. Since embeddings are trained, they're not a typical data transformation—they are part of the model. They're trained with other model weights, and functionally are equivalent to a layer of weights.

What about pretrained embeddings? Pretrained embeddings are still typically modifiable during training, so they're still conceptually part of the model.

Validate Input Data Using a Data Schema

To monitor your data, you should continuously check your data against expected statistical values by writing rules that the data must satisfy. This collection of rules is called a **data schema**. Define a data schema by following these steps:

1. For your feature data, understand the range and distribution. For categorical features, understand the set of possible values.
2. Encode your understanding into rules defined in the schema. Examples of rules are:
    - Ensure that user-submitted ratings are always between 1 and 5.
    - Check that "the" occurs most frequently (for an English text feature).
    - Check that categorical features have values from a fixed set.
3. Test your data against the data schema. Your schema should catch data errors such as:
    - anomalies
    - unexpected values of categorical variables
    - unexpected data distributions

Ensure Splits are Good Quality

Your test and training splits must be equally representative of your input data. If the test and training splits are statistically different, then training data will not help predict the test data. To learn how to sample and split data, see the Sampling and Splitting Data section in the Data Preparation and Feature Engineering in ML course.

Monitor the statistical properties of your splits. If the properties diverge, raise a flag. Further, test that the ratio of examples in each split stays constant. For example, if your data is split 80:20, that ratio should not change.

Test Engineered Data

While your raw data might be valid, your model only sees engineered feature data. Because engineered data looks very different from raw input data, you need to check engineered data separately. Based on your understanding of your engineered data, write unit tests. For example, you can write unit tests to check for the following conditions:

- All numeric features are scaled, for example, between 0 and 1.
- One-hot encoded vectors only contain a single 1 and N-1 zeroes.
- Missing data is replaced by mean or default values.
- Data distributions after transformation conform to expectations. For example, if you've normalized using z-scores, the mean of the z-scores is 0.
- Outliers are handled, such as by scaling or clipping.

Establish a Baseline

Comparing your model against a baseline is a quick test of the model's quality. When developing a new model, define a **baseline** by using a simple heuristic to predict the label. If your trained model performs worse than its baseline, you need to improve your model.

Examples of baselines are:

- Using a linear model trained solely on the most predictive feature.
- In classification, always predicting the most common label.
- In regression, always predicting the mean value.

Once you validate a version of your model in production, you can use that model version as a baseline for newer model versions. Therefore, you can have multiple baselines of different complexities. Testing against baselines helps justify adding complexity to your model. A more complex model should always perform better than a less complex model or baseline.

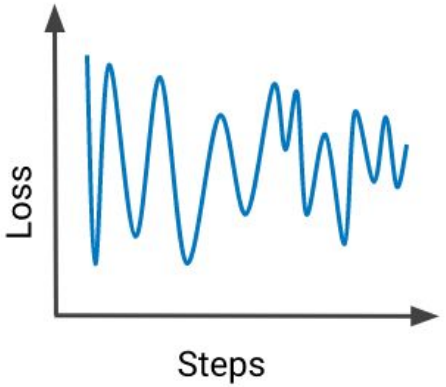
Implement Tests for ML Code

The testing process to catch bugs in ML code is similar to the testing process in traditional debugging. You'll write unit tests to detect bugs. Examples of code bugs in ML are:

- Hidden layers that are configured incorrectly.
- Data normalization code that returns NaNs.

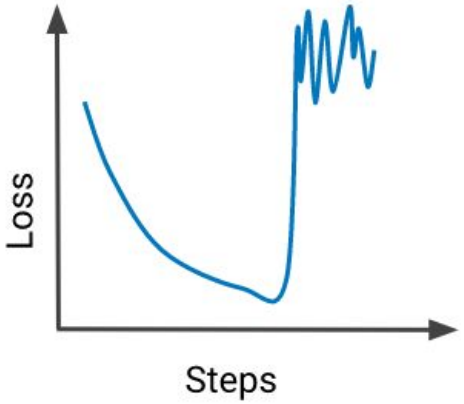
A sanity check for the presence of code bugs is to include your label in your features and train your model. If your model does not work, then it definitely has a bug.

Hyperparameter	Description
Learning Rate	Typically, ML libraries will automatically set the learning rate. For example, in TensorFlow, most TF Estimators use the AdagradOptimizer, which sets the learning rate at 0.05 and then adaptively modifies the learning rate during training. The other popular optimizer, AdamOptimizer, uses an initial learning rate of 0.001. However, if your model does not converge with the default values, then manually choose a value between 0.0001 and 1.0, and increase or decrease the value on a logarithmic scale until your model converges. Remember that the more difficult your problem, the more epochs your model must train for before loss starts to decrease.
Regularization	First, ensure your model can predict without regularization on the training data. Then add regularization only if your model is overfitting on training data. Regularization methods differ for linear and nonlinear models. For linear models, choose L_1 regularization if you need to reduce your model's size. Choose L_2 regularization if you prefer increased model stability. Increasing your model's stability makes your model training more reproducible. Find the correct value of the regularization rate, λ, by starting at 1e-5 and tuning that value through trial and error. To regularize a deep neural network model, use Dropout regularization. Dropout removes a random selection of a fixed percentage of the neurons in a network layer for a single gradient step. Typically, dropout will improve generalization at a dropout rate of between 10% and 50% of neurons.
Training epochs	You should train for at least one epoch, and continue to train so long as you are not overfitting.
Batch size	Typically, the batch size of a mini-batch is between 10 and 1000. For SGD, the batch size is 1. The upper bound on your batch size is limited by the amount of data that can fit in your machine's memory. The lower bound on batch size depends on your data and algorithm. However, using a smaller batch size lets your gradient update more often per epoch, which can result in a larger decrease in loss per epoch. Furthermore, models trained using smaller batches generalize better. For details, see On large-batch training for deep learning: Generalization gap and sharp minima N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. ICLR, 2017. Prefer using the smallest batch sizes that result in stable training.
Depth and width of layers	In a neural network, depth refers to the number of layers, and width refers to the number of neurons per layer. Increase depth and width as the complexity of the corresponding problem increases. Adjust your depth and width by following these steps: 1. Start with 1 fully-connected hidden layer with the same width as your input layer. 2. For regression, set the output layer's width to 1. For classification, set the output layer's width to the number of classes. 3. If your model does not work, and you think your model needs to be deeper to learn your problem, then increase depth linearly by adding a fully-connected hidden layer at a time. The hidden layer's width depends on your problem. A commonly-used approach is to use the same width as the previous hidden layer, and then discover the appropriate width through trial-and-error. The change in width of successive layers also depends on your problem. A practice drawn from common observation is to set a layer's width equal to or less than the width of the previous layer. Remember, the depth and width don't have to be exactly right. You'll tune their values later when you optimize your model.



Your model is not converging. Try these debugging steps:

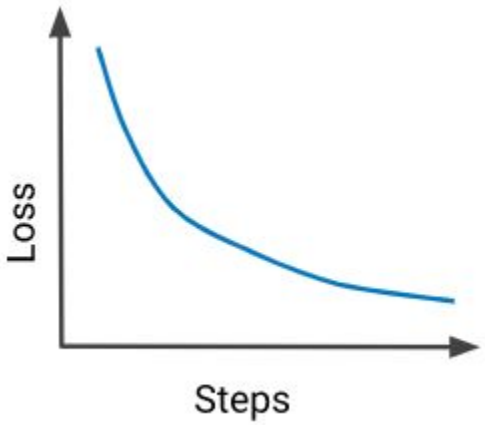
- Check if your features can predict the labels by following the steps in Model Debugging.
- Check your data against a data schema to detect bad examples.
- If training looks unstable, as in this plot, then reduce your learning rate to prevent the model from bouncing around in parameter space.
- Simplify your dataset to 10 examples that you know your model can predict on. Obtain a very low loss on the reduced dataset. Then continue debugging your model on the full dataset.
- Simplify your model and ensure the model outperforms your baseline. Then incrementally add complexity to the model.

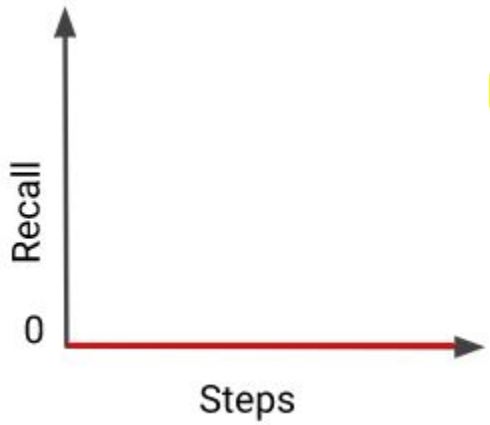


A large increase in loss is typically caused by anomalous values in input data. Possible causes are:

- NaNs in input data.
- Exploding gradient due to anomalous data.
- Division by zero.
- Logarithm of zero or negative numbers.

To fix an exploding loss, check for anomalous data in your batches, and in your engineered data. If the anomaly appears problematic, then investigate the cause. Otherwise, if the anomaly looks like outlying data, then ensure the outliers are evenly distributed between batches by shuffling your data.

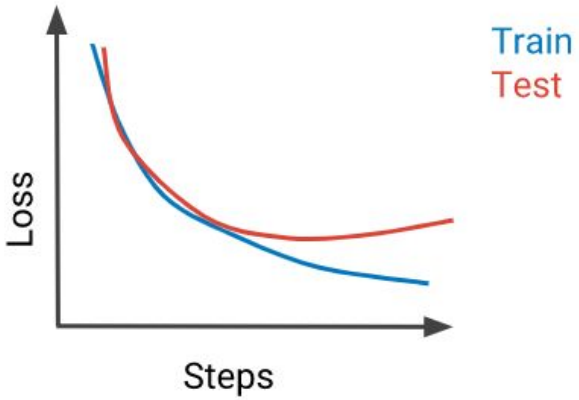




Recall is stuck at 0 because your examples' classification probability is never higher than the threshold for positive classification. This situation often occurs in problems with a large class imbalance. Remember that ML libraries, such as TF Keras, typically use a default threshold of 0.5 to calculate classification metrics.

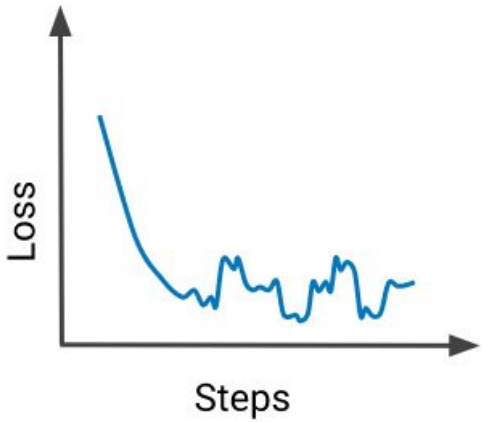
Try these steps:

- Lower your classification threshold.
- Check threshold-invariant metrics, such as AUC.



Your model is overfitting to the training data. Try these steps:

- Reduce model capacity.
- Add regularization.
- Check that the training and test splits are statistically equivalent.

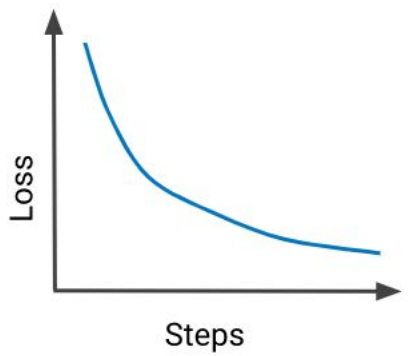


Your loss is showing repetitive, step-like behavior. It's probable that the input data seen by your model is itself exhibiting repetitive behavior. Ensure that shuffling is removing repetitive behavior from input data.

It's Working!

"It's working perfectly now!" Mel exclaims. She leans back into her chair triumphantly and heaves a big sigh. The curve looks great and you beam with accomplishment. Mel and you take a moment to discuss the following additional checks for validating your model.

- real-world metrics
- baselines
- absolute loss for regression problems
- other metrics for classification problems



Problem	Evaluating Quality
Regression	Besides reducing your absolute Mean Square Error (MSE), reduce your MSE relative to your label values. For example, assume you're predicting prices of two items that have mean prices of 5 and 100. In both cases, assume your MSE is 5. In the first case, the MSE is 100% of your mean price, which is clearly a large error. In the second case, the MSE is 5% of your mean price, which is a reasonable error.
Multiclass classification	If you're predicting a small number of classes, look at per-class metrics individually. When predicting on many classes, you can average the per-class metrics to track overall classification metrics. Alternatively, you can prioritize specific quality goals depending on your needs. For example, if you're classifying objects in images, then you might prioritize the classification quality for people over other objects.

Check Metrics for Important Data Slices

After you have a high-quality model, your model might still perform poorly on subsets of your data. For example, your unicorn predictor must predict well both in the Sahara desert and in New York City, and at all times of the day. However, you have less training data for the Sahara desert. Therefore, you want to track model quality specifically for the Sahara desert. Such subsets of data, like the subset corresponding to the Sahara desert, are called **data slices**. You should separately monitor data slices where performance is especially important or where your model might perform poorly.

Use your understanding of the data to identify data slices of interest. Then compare model metrics for data slices against the metrics for your entire data set. Checking that your model performs across all data slices helps remove bias. For more, see Fairness: Evaluating for Bias.

Use Real-World Metrics

Model metrics do not necessarily measure the real-world impact of your model. For example, you might change a hyperparameter and increase your AUC, but how did the change affect user experience? To measure real-world impact, you need to define separate metrics. For example, you could survey users who see a unicorn appearance prediction to check whether or not they saw a unicorn. Measuring real-world impact helps compare the quality of different iterations of your model.

Add Useful Features

You can improve model performance by adding features that encode information not yet encoded by your existing features. You can find linear correlations between individual features and labels by using correlation matrices. To detect nonlinear correlations between features and labels, you must train the model with and without the feature, or combination of features, and check for an increase in model quality. You must justify the feature's inclusion by an increase in model quality.

Tune Model Depth and Width

While debugging your model, you only increased model depth and width. In contrast, during model optimization, you either increase or decrease depth and width depending on your goals. If your model quality is adequate, then try reducing overfitting and training time by decreasing depth and width. Specifically, try halving the width at each successive layer. Since your model quality will also decrease, you need to balance quality with overfitting and training time.

Conversely, if you need higher model quality, then try increasing depth and width. For an example, see this Neural Network Playground exercise. Remember that increases in depth and width are practically limited by accompanying increases in training time and overfitting. To understand overfitting, see Generalization: Peril of Overfitting.

Since the depth and width are hyperparameters, you can use hyperparameter tuning to optimize depth and width.

Testing for Deploying

Role of Testing in ML Pipelines

In software development, the ideal workflow follows test-driven development (TDD). However, writing ML tests is not straightforward. Your tests depend on your data, model, and problem. For example, before training your model, you cannot write a test to validate the loss. Instead, you discover the achievable loss during model development and *then* test new model versions against the achievable loss.

You need tests for:

- Validating input data.
- Validating feature engineering.
- Validating quality of new model versions.
- Validating serving infrastructure.
- Testing integration between pipeline components.

Test Model Updates with Reproducible Training

No doubt you want to continue improving your unicorn appearance predictor. Say you refactor the feature engineering code for the "time of day" feature. How do you test that the code is correct? You decide to train your model again and see if you get the same result. Oh no, you find that your model training is not reproducible. Determined to continue predicting unicorn appearances, you investigate further. You find that you can achieve reproducibility by following these steps:

- Deterministically seed the random number generator (RNG). For details, see randomization in data generation from the Data Preparation and Feature Engineering in ML course.
- Initialize model components in a fixed order to ensure the components get the same random number from the RNG on every run. ML libraries typically handle this requirement automatically.
- Average several runs of the model.
- Use version control, even for preliminary iterations, so that you can pinpoint code and parameters when investigating your model or pipeline.

Even after taking these steps, you could have other sources of non-determinism.

Validate Model Quality before Serving

Before pushing a new model version to production, test for these two types of degradations in quality:

- Sudden degradation: A bug in the new version could cause significantly lower quality. Validate new versions by checking their quality against the previous version.
- Slow degradation: Your test for sudden degradation might not detect a slow degradation in model quality over multiple versions. Instead, ensure your model's predictions on a validation dataset meet a fixed threshold. If your validation dataset deviates from live data, then update your validation dataset and ensure your model still meets the same quality threshold.

Testing API Calls

How do you test updates to API calls? Sure, you could retrain your model, but that's time intensive. Instead, write a unit test to generate random input data and run a single step of gradient descent. You want the step to complete without runtime errors.

Testing for Algorithmic Correctness

A model must not only predict correctly, but do so because it is algorithmically correct, not lucky. For example, if 99% of emails aren't spam, then classifying all email as not spam gets 99% accuracy through chance. Therefore, you need to check your model for algorithmic correctness. Follow these steps:

- Train your model for a few iterations and verify that the loss decreases.
- Train your algorithm without regularization. If your model is complex enough, it will memorize the training data and your training loss will be close to 0.
- Test specific subcomputations of your algorithm. For example, you can test that a part of your RNN runs once per element of the input data.

★ **Note:** Because training a model takes so long, you might try training a model only partially before comparing its quality against a previously trained model. However, avoid testing partially-trained models because the test is hard to maintain and interpret.

Write Integration tests for Pipeline Components

In an ML pipeline, changes in one component can cause errors in other components. Check that components work together by writing a test that runs the entire pipeline end-to-end. Such a test is called an **integration test**.

Besides running integration tests continuously, you should run integration tests when pushing new models and new software versions. The slowness of running the entire pipeline makes continuous integration testing harder. To run integration tests faster, train on a subset of the data or with a simpler model. The details depend on your model and data. To get continuous coverage, you'd adjust your faster tests so that they run with every new version of model or software. Meanwhile, your slow tests would run continuously in the background.

Validate Model-Infra Compatibility before Serving

If your model is updated faster than your server, then your model will have different software dependencies from your server, potentially causing incompatibilities. Ensure that the operations used by the model are present in the server by staging the model in a sandboxed version of the server.

Testing in Production

Check for Training-Serving Skew

Training-serving skew means your input data differs between training and serving. The following table describes the two important types of skew:

Type	Definition	Example	Solution
Schema skew	Training and serving input data do not conform to the same schema.	The format or distribution of the serving data changes while your model continues to train on old data.	Use the same schema to validate training and serving data. Ensure you separately check for statistics not checked by your schema, such as the fraction of missing values
Feature skew	Engineered data differs between training and serving.	Feature engineering code differs between training and serving, producing different engineered data.	Similar to schema skew, apply the same statistical rules across training and serving engineered data. Track the number of detected skewed features, and the ratio of skewed examples per feature.

Test Quality of Live Model on Served Data

You've validated your model. But what if real-world scenarios, such as unicorn behavior, change after recording your validation data? Then the quality of your served model will degrade. However, testing quality in serving is hard because real-world data is not always labelled. If your serving data is not labelled, consider these tests:

- Generate labels using human raters.
- Investigate models that show significant statistical bias in predictions. See Classification: Prediction Bias.
- Track real-world metrics for your model. For example, if you're classifying spam, compare your predictions to user-reported spam.
- Mitigate potential divergence between training and serving data by serving a new model version on a fraction of your queries. As your validate your new serving model, gradually switch all queries to the new version.

Using these tests, remember to monitor both sudden and slow degradation in prediction quality.

Monitor Model Age Throughout Pipeline

If the serving data evolves with time but your model isn't retrained regularly, then you will see a decline in model quality. Track the time since the model was retrained on new data and set a threshold age for alerts. Besides monitoring the model's age at serving, you should monitor the model's age throughout the pipeline to catch pipeline stalls.

Test that Model Weights and Outputs are Numerically Stable

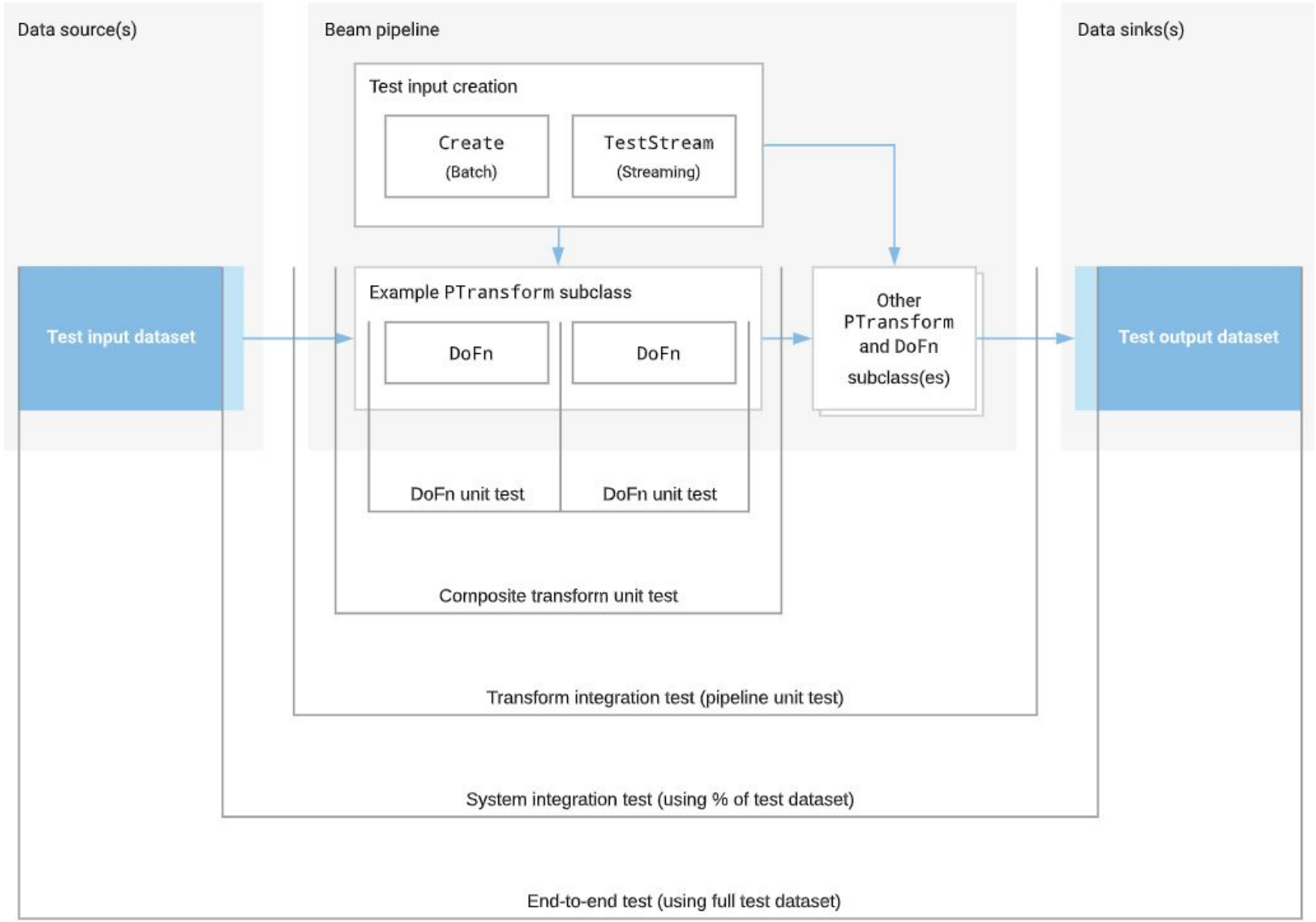
During model training, your weights and layer outputs should not be NaN or Inf. Write tests to check for NaN and Inf values of your weights and layer outputs. Additionally, test that more than half of the outputs of a layer are not zero.

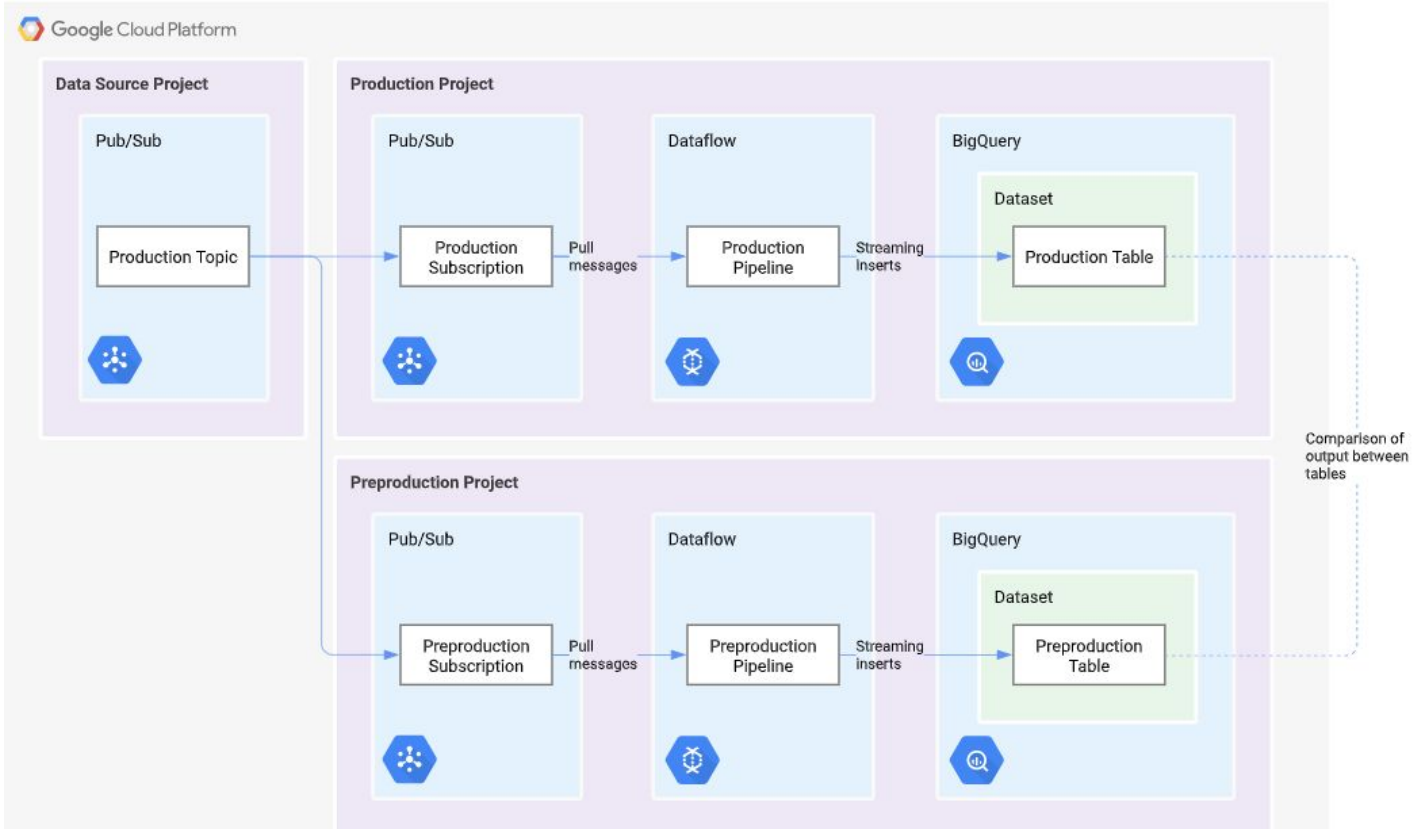
Monitor Model Performance

Your unicorn appearance predictor has been more popular than expected! You're getting lots of prediction requests and even more training data. You think that's great until you realize that your model is taking more and more memory and time to train. You decide to monitor your model's performance by following these steps:

- Track model performance by versions of code, model, and data. Such tracking lets you pinpoint the exact cause for any performance degradation.
- Test the training steps per second for a new model version against the previous version and against a fixed threshold.
- Catch memory leaks by setting a threshold for memory use.
- Monitor API response times and track their percentiles. While API response times might be outside your control, slow responses could potentially cause poor real-world metrics.
- Monitor the number of queries answered per second.

- **Planning**: What non-functional requirements should you capture before you begin development? Why should you think about SLOs during the planning stage, and how should you set them? How should SLOs inform the design of your data pipelines? What potential impact do you need to consider for upstream and downstream systems?
- **Development and testing**: What best practices should you observe when you write Apache Beam code? What tests should you build and automate? How should pipeline developers work with different deployment environments to write, test, and run code?
- **Deployment**: How should you update pipelines that are already running when a new release is available? How can you apply CI/CD practices to data pipelines?
- **Monitoring**: What monitoring should you have for production pipelines? How do you detect and resolve performance problems, data-handling issues, and code errors in production?

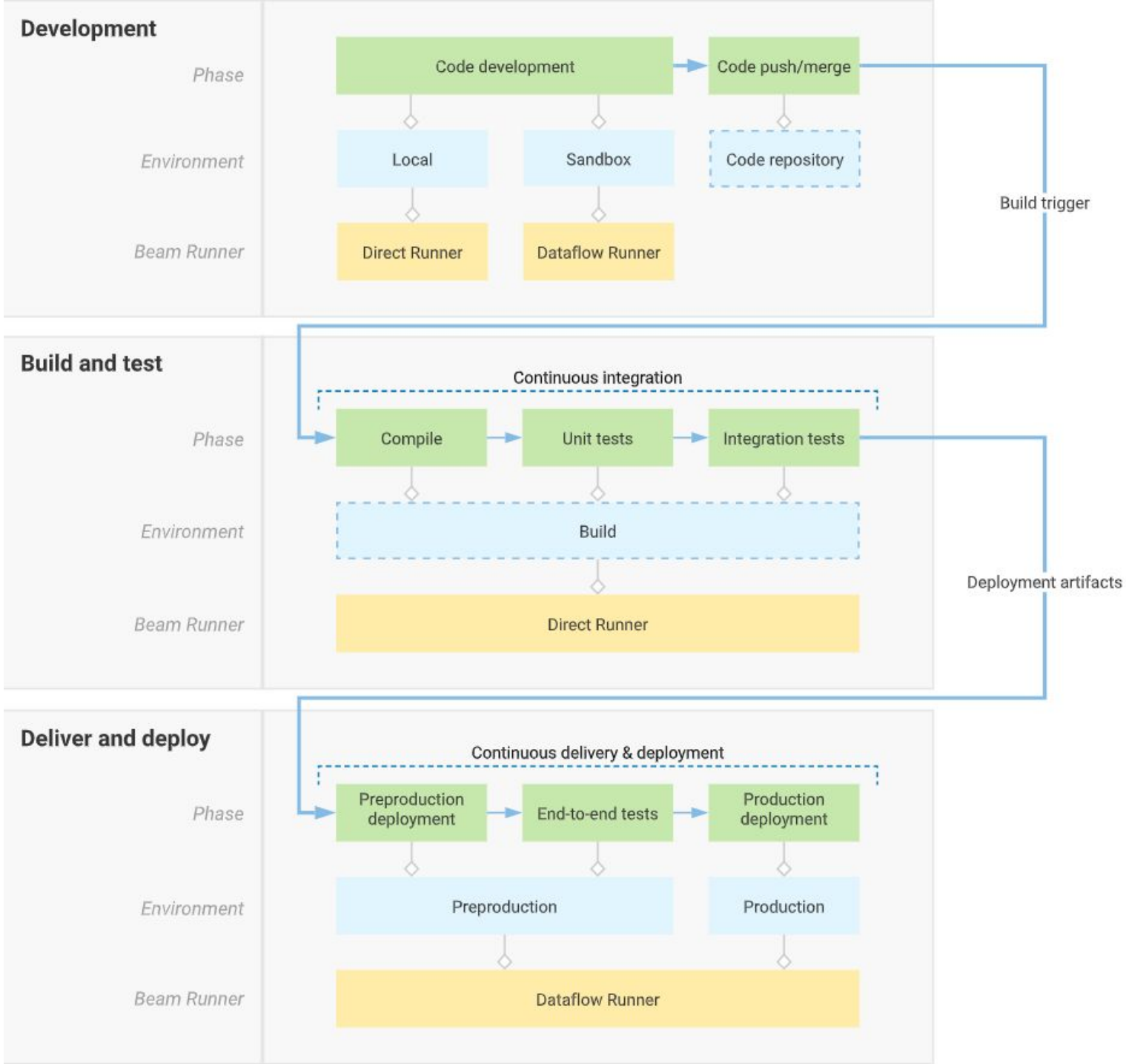

Data source(s)
Beam pipeline
Data sinks(s)
Test input creation
Create
(Batch)
TestStream
(Streaming)
Test input dataset
Example PTransform subclass
DoFn
DoFn
Other
PTransform
and DoFn
subclass(es)
Test output dataset
DoFn unit test
DoFn unit test
Composite transform unit test
Transform integration test (pipeline unit test)
System integration test (using % of test dataset)
End-to-end test (using full test dataset)

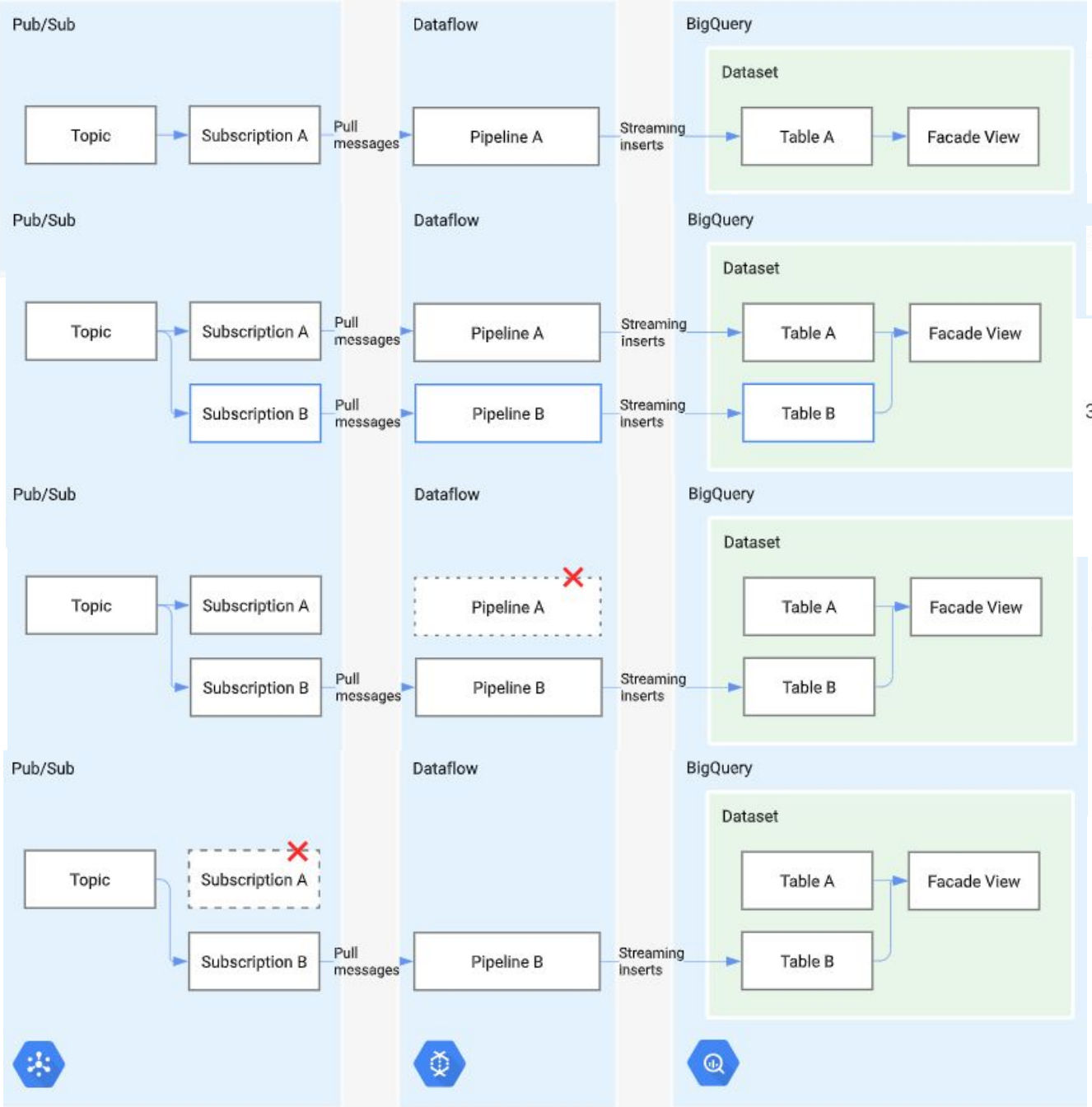


In the diagram, both pipelines read from the same Pub/Sub topic, but they use separate subscriptions. This allows the two pipelines to process the same data independently and allows you to compare the results. The test pipeline uses a separate service account from the production project, and therefore avoids using the Pub/Sub subscriber quota for the production project.

Unlike batch pipelines, streaming pipelines continue to run until they're explicitly cancelled. In end-to-end tests, you should decide whether to leave the pipeline running (perhaps until the next end-to-end test is run), or cancel the pipeline at a point that represents test completion so that you can inspect the results.

The type of test data you use influences this decision. For example, if you use a bounded set of test data that's fed to the streaming pipeline, such as a table that's converted into a stream, you will likely cancel the pipeline when all elements have completed processing. Alternatively, If you use a real data source (such as an existing Pub/Sub topic that's used in production) or if you otherwise generate test data continually, you might want to keep test pipelines running over a longer period. This lets you compare the behavior against the production environment, or even against other test pipelines.


Development
Phase
Environment
Beam Runner
Code development
Code push/merge
Local
Sandbox
Code repository
Direct Runner
Dataflow Runner
Build trigger
Build and test
Continuous integration
Compile
Unit tests
Integration tests
Build
Direct Runner
Deployment artifacts
Deliver and deploy
Continuous delivery & deployment
Preproduction deployment
End-to-end tests
Production deployment
Preproduction
Production
Dataflow Runner



1. In the initial state, the existing streaming pipeline (**Pipeline A**) is running and reading messages from a Pub/Sub topic (**Topic**) using a subscription (**Subscription A**). The results are written to a BigQuery table (**Table A**). Results are consumed through a BigQuery view, which acts as a façade to mask underlying table changes. This is an application of a design method called the façade pattern. The following diagram shows the initial state.

2. You create a new subscription (**Subscription B**) for the updated pipeline. You deploy the updated pipeline (**Pipeline B**), which reads from the Pub/Sub topic (**Topic**) using **Subscription B** and writes to a separate BigQuery table (**Table B**). The following diagram illustrates this flow.

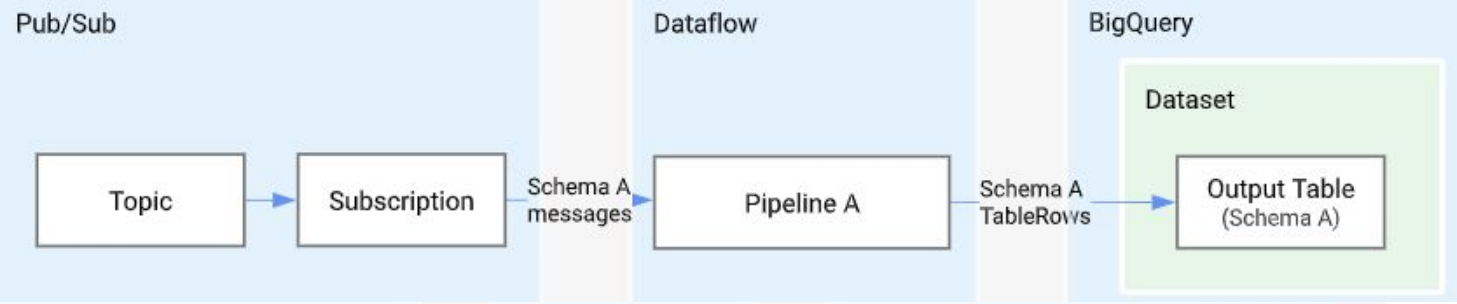
   At this point, **Pipeline A** and **Pipeline B** are running in parallel and writing results to separate tables. You record ti *t* as the timestamp of the earliest complete window that's processed by **Pipeline B**.

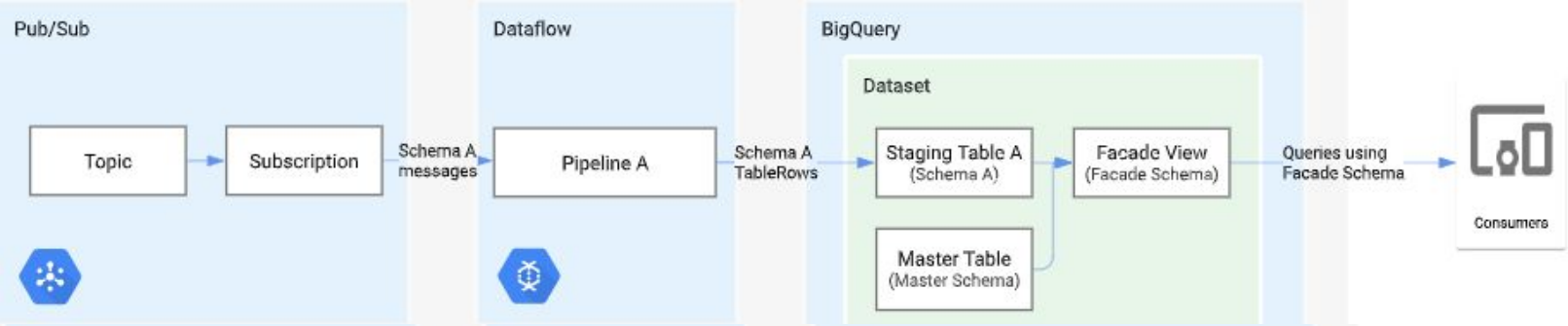
3. You allow **Pipeline A** to drain when its watermark has exceeded time *t*. This closes any open windows and completes processing for any in-flight data. If complete windows are important (assuming no late data), you allo both pipelines to run until you have complete overlapping windows before draining the first pipeline. You stop the streaming job for **Pipeline A** after all in-flight data is processed and written to **Table A**. The following diagram shows this stage.

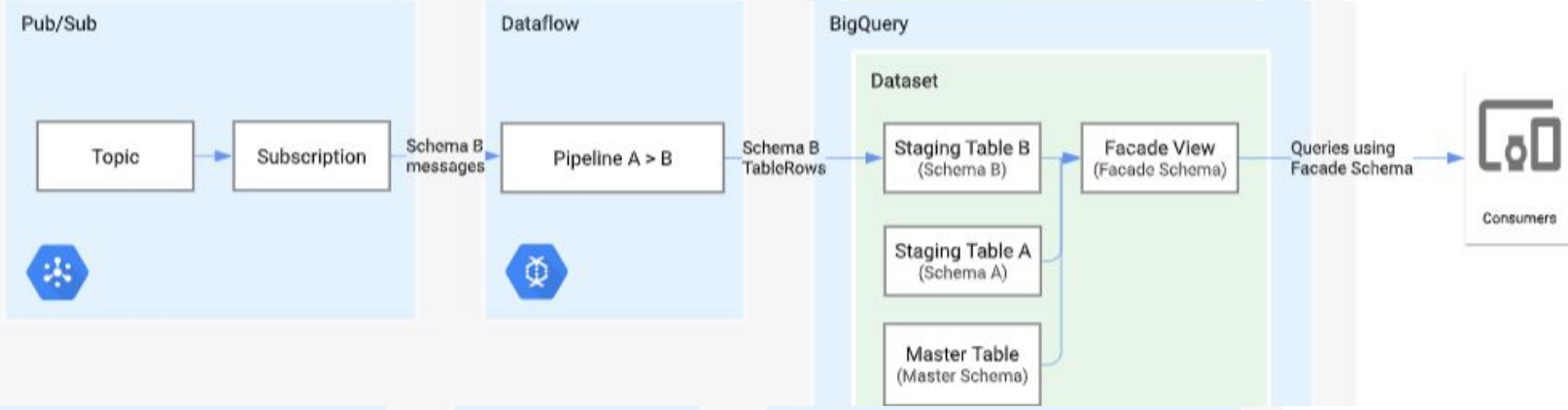
4. At this point, only **Pipeline B** is running. You can query from a BigQuery view (**Facade View**), which acts as a façade for **Table A** and **Table B**. For rows that have the same timestamp in both tables, you configure the view to return results from **Table B** or fall back to **Table A** if the rows don't exist in **Table B**. The following diagram shows the view (**Facade View**) reading from both **Table A** and **Table B**.

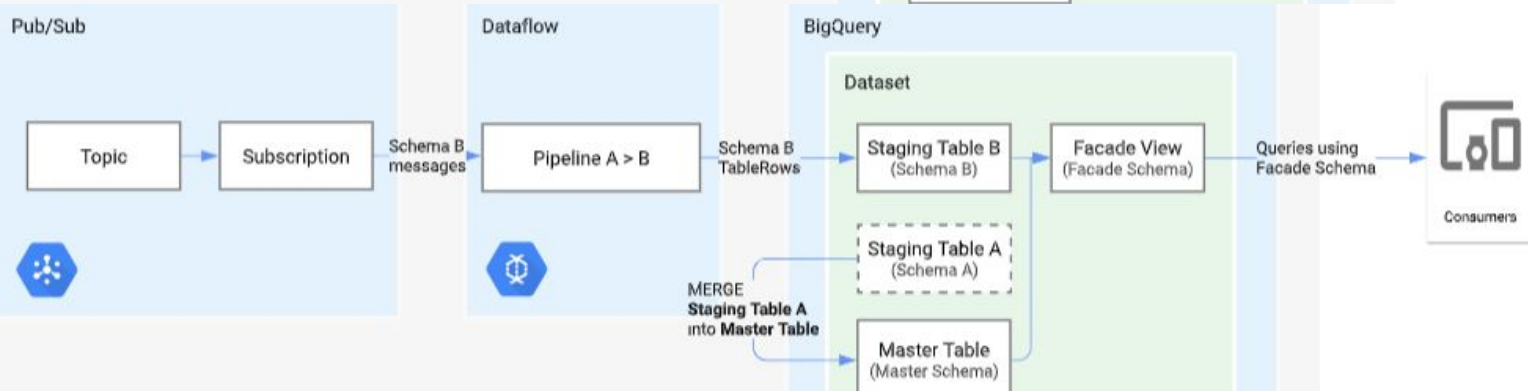
At this point, you can delete **Subscription A** if you want. You can merge the two BigQuery tables or keep them separate.

Deploying overlapped (parallel) pipelines can simplify rollback if any issues are detected with a new pipeline deployment. In the previous example, you might want to keep **Pipeline A** running while you monitor **Pipeline B** for correct operation. If there are any issues with **Pipeline B**, you can roll back to using **Pipeline A**.


Pub/Sub
Dataflow
BigQuery
Dataset
Topic
Subscription
Schema A messages
Pipeline A
Schema A TableRows
Output Table (Schema A)


Pub/Sub
Dataflow
BigQuery
Dataset
Topic
Subscription
Schema A messages
Pipeline A
Schema A TableRows
Staging Table A (Schema A)
Facade View (Facade Schema)
Queries using Facade Schema
Consumers
Master Table (Master Schema)

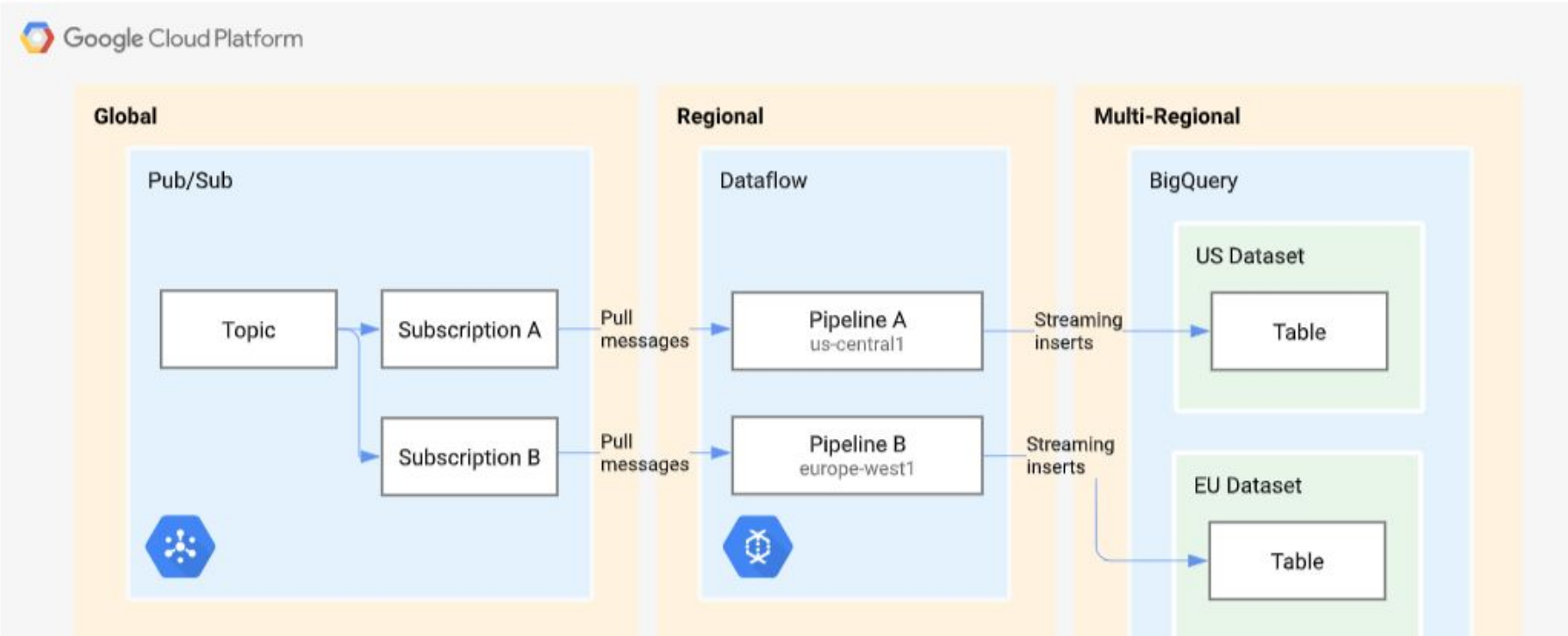

Pub/Sub
Dataflow
BigQuery
Dataset
Topic
Subscription
Schema B messages
Pipeline A > B
Schema B TableRows
Staging Table B (Schema B)
Facade View (Facade Schema)
Queries using Facade Schema
Consumers
Staging Table A (Schema A)
Master Table (Master Schema)


Pub/Sub
Dataflow
BigQuery
Dataset
Topic
Subscription
Schema B messages
Pipeline A > B
Schema B TableRows
Staging Table B (Schema B)
Facade View (Facade Schema)
Queries using Facade Schema
Consumers
Staging Table A (Schema A)
MERGE Staging Table A into Master Table
Master Table (Master Schema)

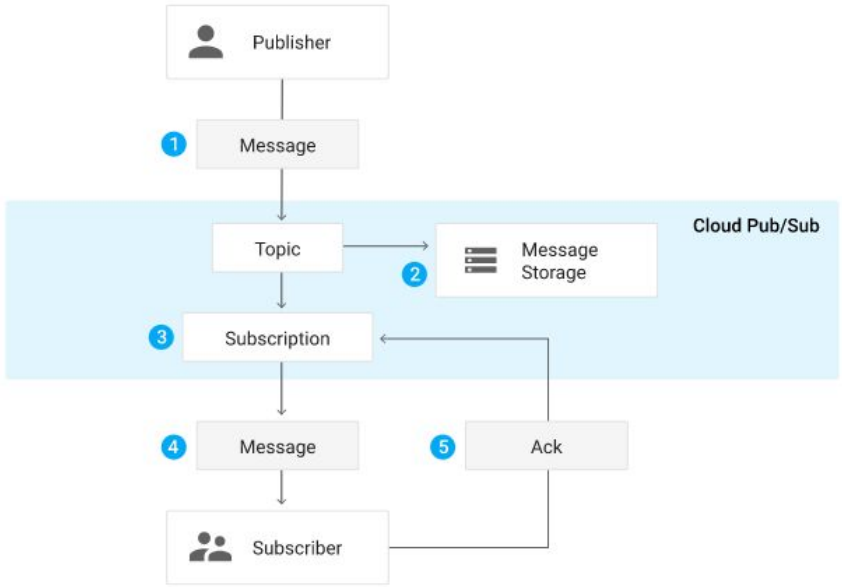
High availability and geographic redundancy

You can run multiple streaming pipelines in parallel for high-availability data processing. For example, you can run two parallel streaming jobs in different regions, which provides geographical redundancy and fault tolerance for data processing.

By considering the geographic availability of data sources and sinks, you can operate end-to-end pipelines in a highly available, multi-region configuration. Tthe following diagram shows an example deployment.



https://cloud.google.com/pubsub/docs/overview



1. A *publisher* application creates a *topic* in the Pub/Sub service and sends *messages* to the topic. A message contains a payload and optional *attributes* that describe the payload content.
2. The service ensures that published messages are retained on behalf of subscriptions. A published message is retained for a subscription until it is acknowledged by any subscriber consuming messages from that subscription.
3. Pub/Sub forwards messages from a topic to all of its subscriptions, individually.
4. A subscriber receives messages either by Pub/Sub *pushing* them to the subscriber's chosen endpoint, or by the subscriber *pulling* them from the service.
5. The subscriber sends an acknowledgement to the Pub/Sub service for each received message.
6. The service removes acknowledged messages from the subscription's message queue.

Kubeflow Pipelines with AI Platform

Objectives

In this lab, you will perform the following tasks:

- Create a Kubernetes cluster and install Kubeflow Pipelines
- Launch an AI Platform Notebook
- Download example notebooks
- Create and run and end-to-end ML Pipeline using Cloud AI Platform and pre-built components
- Examine and verify the output of each step
- Test the online prediction of your finished model
- Convert a series of basic python function to pipeline components
- Assemble and execute a new pipeline with these python functions

Tuning Performance to reduce training time, reduce cost, and increase scale

Constraint	Input / Output	CPU	Memory
Commonly Occurs	Large inputs Input requires parsing Small models	Expensive computations Underpowered Hardware	Large number of inputs Complex model
Take Action	Store efficiently Parallelize reads Consider batch size	Train on faster accel. Upgrade processor Run on TPUs Simplify model	Add more memory Use fewer layers Reduce batch size