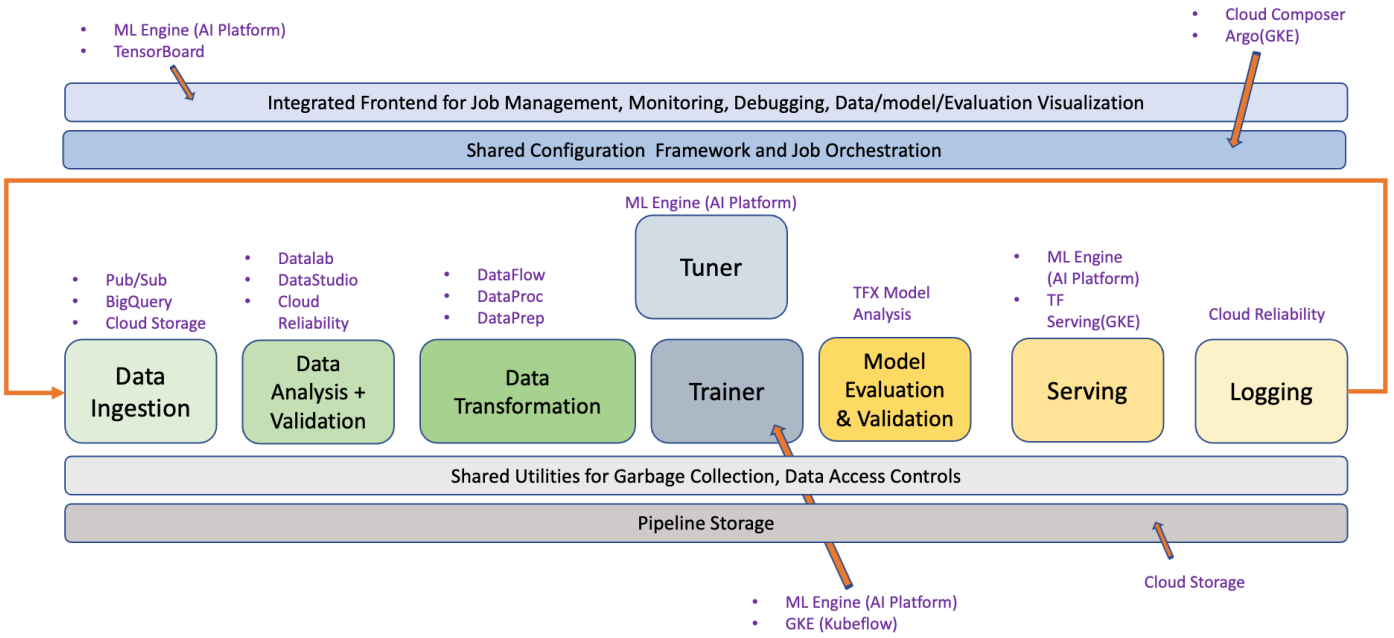
GCP ML Systems

Table of Contents

1 ML Systems

1.1 System Components



- **Data Ingestion**
  - Deals with how data is fed into the system.
  - Data can be streamed, batch processed etc.

- **Data Analysis and Validation**
  - Deals with data quality
  - Following diagnostic questions help maintain data health

1) Is the new distribution similar enough to the old one?
2) Are all expected features present?
3) Are any unexpected features present?
4) Does the feature have the expected type?
5) Does an expected proportion of the examples contain the feature?
6) Do the examples have the expected number of values for feature?

- ***Example scenarios for diagnosis***
  - Assume that your ML model accepts the prices of goods from all over the world in US dollar to make predictions about their future price. To accept prices in US dollars for all goods all over the world, the data need to be transformed from their original currency into dollars.

  - **Scenario 1**: One day, a system outside of your ML system changes the format of its data stream and your parser silently starts returning 1.0 for the conversion rate between Japanese Yen and the USD. (Ideally it would be 100 Yen = 1 USD). Because your model uses this quantity to convert Yen to dollar those items prices are now unnaturally high. Instead of 100 yen equals $1, those items show up as $100. Which question would have caught that?
    - **In this case, question 1 would catch the problem**: Is the new distribution similar to the old one.

  - **Scenario 2:** What if the parser throws an error when this happens and so the price for such items becomes null?
    - **In this case, question 2 would catch the problem**: Are all expected features present.

  - **Scenario 3**: What if the parser throws an error for such items and your converter returns the error string? So, instead of $1.05, you get the price as "currency rate not available".

        - **In this case, question 4 would catch the problem**: Does the feature have the expected type.

    - **Scenario 4**: What if the ML model uses several prices? For example, list price and discount price and all the prices exhibit the same error.
        - **In this case, question 5 would catch the problem**: Does the expected proportion of examples contain this feature.

    - **Scenario 5:** What if the error is only on Yen and all other currency conversions are fine?
        - **In this case, question 1 would catch the problem**: Is the new distribution similar to the old one.

- **Data Transformation**
    - The data transformation component allows for feature wrangling. It can do things like generate feature to integer mappings.
    - Critically, whatever mappings that are generated must be saved and reused at serving time. Failure to do this consistently results in a problem called ***Training Serving Skew***.

- **Trainer**
    - Responsible for training the model.
    - It should be able to support data parallelism and model parallelism, and scale to large numbers of workers.
    - It should also automatically monitor and log everything and support the use of experimentation.
    - The trainer should also support hyperparameter tuning.

- **Tuner**
    - There are no globally optimal values for hyperparameters, only problem-specific optima.
    - Because we expect to do hyperparameter tuning, our system needs to support it.
    - We need a way to operate multiple experiments in parallel and ideally, use early experiments to guide later ones automatically.

- **Model evaluation**
  - Purpose of this module is to ensure that the models are good before moving them into the production environment.
  - The goal is to ensure that users' experiences aren't degraded.
  - There are two main things that we care about with respect to model quality:
    - How **safe** the model is **to serve**?
      - A safe to serve model won't crash or cause errors in the serving system when being loaded or when sent unexpected inputs.
      - It also shouldn't use more than the expected amount of resources, like memory.
    - the model's **prediction quality**.
  - Model evaluation is part of the iterative process where teams try and improve their models.
    - Because it's expensive to test on live data, experiments are generally run offline first.
    - Model evaluation consists of a person or a group of people assessing the model with respect to some business-relevant metric.
    - If the model meets their criteria, then it can be pushed into production for a live experiment.

- **Model validation**
  - In contrast to the model evaluation component, which is human facing, the model validation component is not.
  - It evaluates the model against fixed thresholds and alerts engineers when things go awry.
  - One common test is to look at performance by slice of the input.
    - For example, business stakeholders may care strongly about a particular geographic market.
  - This is also another junction where ML fairness comes up.

- **Serving**
  - The serving component should be
    - Low latency
      - to respond to users quickly
    - Highly efficient
      - so that many instances can be run simultaneously,
    - scale horizontally
      - In order to be reliable and robust to failures.

        - In contrast to training when we care about scaling with our data, at serving we care about responding to variable user demand maximizing throughput and minimizing response latency.
    - Should be easy to update to new versions of the model.
        - When we get new data or engineer new features, we'll want to retrain and push a new version of the model and you want the system to seamlessly transition to the new model version.
        - More generally, the system should allow you to set up a multi-armed bandit architecture to verify which model versions are the best.

- **Logging**
    - Logging is critical for debugging and comparison and what's important is that all logs be easily accessible and integrated.
    - Also, the ability to craft alerting policies, and the ability to detect when new errors occur.

- **Shared Config and Orchestration**
    - If we make change to the trainer, potentially all components will need a change.
        - The trainer uses meta information like vocabularies collected by the data transformation component on data that had been validated and adjusted. The tuner conducts experiments, each of which involves the trainer yielding models that must then be evaluated and validated. The serving component, host models that have been evaluated, validated, and trained, and then runs them in parallel. And everything logs everything.

    - Because everything needs to talk to everything else, it's imperative that these components share resources, and a ***configuration framework***.
        - Failure to do so, can result in large amounts of glue code to tie the system together.
        - ***Glue code*** is an example of an anti-pattern.
            - Something that slows development down.
            - It accumulates because often research and engineering are very distinct organizationally.
            - Within research, ML models can be developed as self-contained black boxes.
            - Engineering on the other hand, needs to tie whatever is produced into a single production environment.

- Glue code arises from their attempts to run code that was never intended to be run in production in production.
- The best remedies for this problem are establish a common architecture for both R&D and production deployment and embed the teams together so that engineering can influence the design of code from its inception.

- ***Orchestration*** is the name for the component responsible for gluing the other components together.
  - In a system where pieces are designed thoughtfully, the orchestration component will be simple and elegant.
  - In GCP, orchestration can be done with ***Cloud composer***, which is managed ***Apache airflow***.
    - There are airflow operators for all the GCP components that we've considered so far, including Cloud Storage, BigQuery, Dataflow, and ML Engine. So, you can orchestrate all these tasks from composer.
    - Here are the steps to compose a workflow in Cloud composer.
      - (1) Define the Ops
      - (2) Arrange them into a DAG
        - The workflow engine uses a Dag to run the apps in the appropriate order, and to explore opportunities for parallelism. But it can't figure out the dependencies on its own, you need to specify them.
      - (3) Upload the Dag to the environment.
      - (4) Explore the Dag run in the web UI.
  - Below is some sample code for cloud composer:

```
# BigQuery training data query
t1 = BigQueryOperator(params)

# BigQuery training data export to GCS
t2 = BigQueryToCloudStorageOperator(params)

# ML Engine training job
t3 = MLEngineTrainingOperator(params)

# App Engine deploy new version
t4 = AppEngineVersionOperator(params)

# Establish dependencies
t1 >> t2 >> t3 >> t4
```

- Another option for orchestration is to use ***Argo on Google Kubernetes engine***.
  - Argo is a container management tool.
  - If each of your tasks, data ingest, data transformation, and model training are running containers, then Argo is a good way to orchestrate the ML pipeline consisting of such containers.
  - Because GCP products are designed to work together, the code to integrate them as simple and elegant.

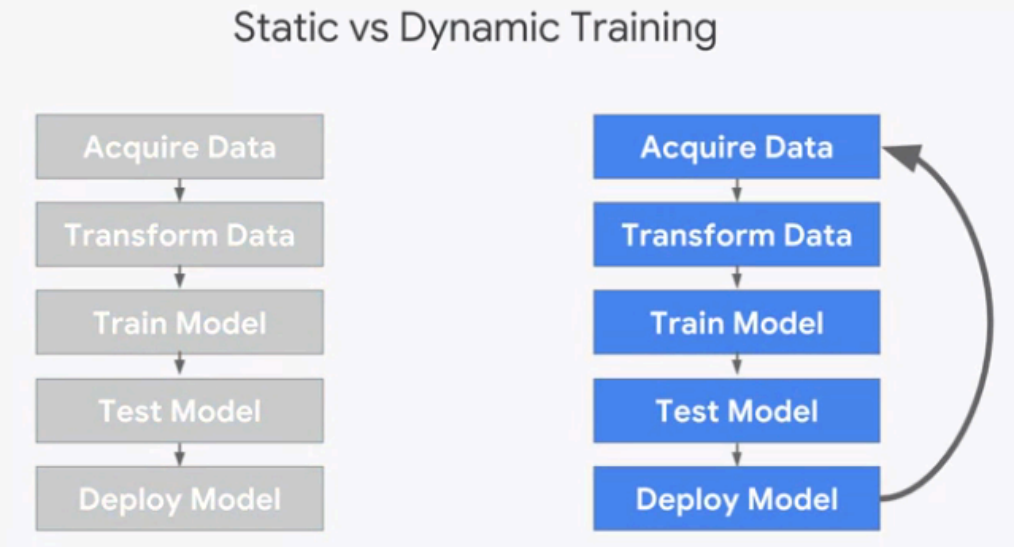
- **Integrated Frontend**
  - As everything needs to talk to everything else, the users of the system need to be able to easily accomplish their tasks in as central a location as possible.
  - In GCP, you can use TensorBoard and Cloud ML Engine.
  - TensorBoard is the visualization software that comes bundled with TensorFlow.

- **Pipeline Storage**
  - Storage is necessary for staging intermediate output of all the components.
  - In GCP that's Google Cloud Storage.

1.2 Design Decisions

1.2.1 Training Design Decision

- When training your model, there are two paradigms, static training and dynamic training.
- If the relationship you're trying to model is one that is constant, like physics, then a statically trained model may be sufficient.
- If on the other hand the relationship your trying to model is like fashion, then the dynamically trained model might be more appropriate.
- In practice, most of the time, you'll need to do dynamic, but you might start with static because it's simpler.



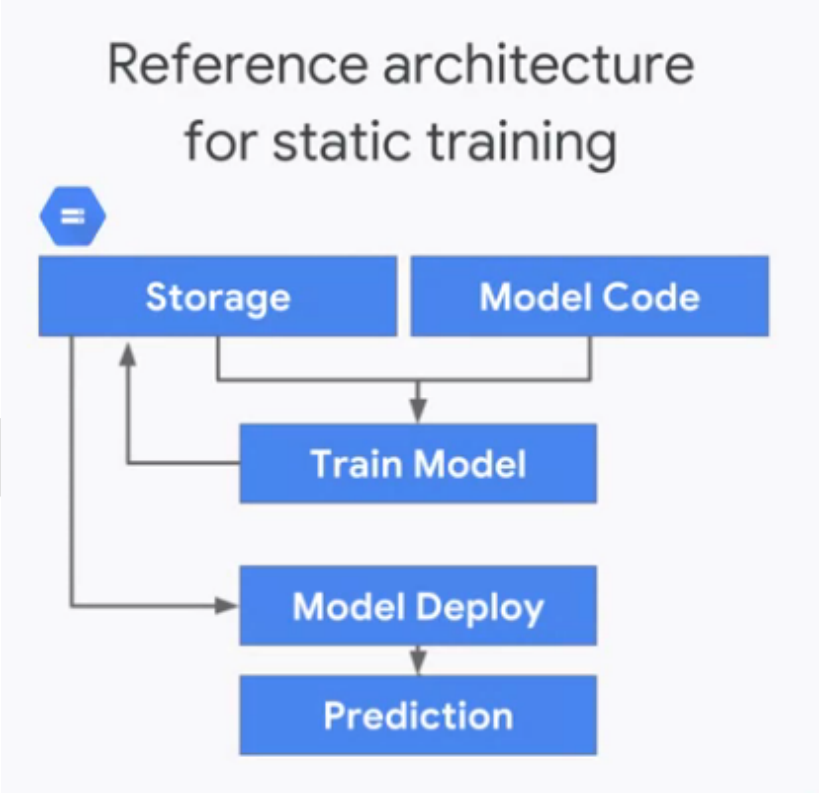
Statically Trained Models	Dynamically Trained Models
Trained once, offline	Add training data over time
Easy to build and test	Engineering is harder Have to do progressive validation
Easy to let become stale	Regularly sync out updated version Will adapt to changes

- Examples

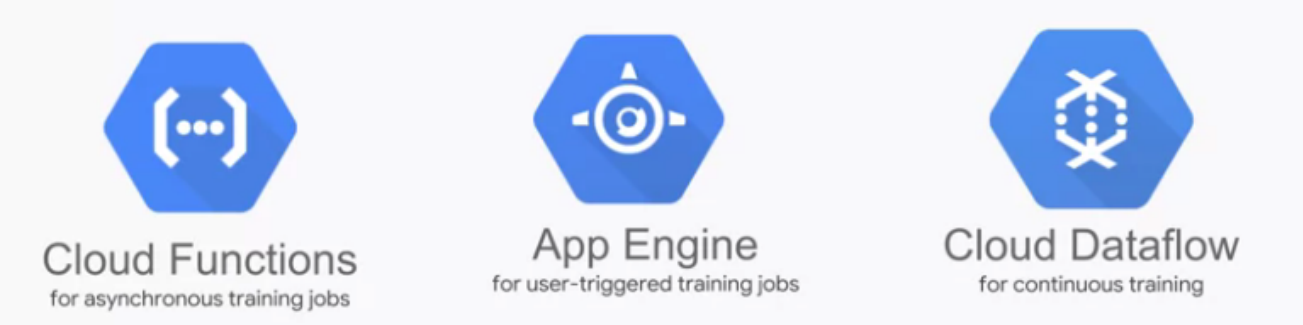
Problem	Training style (static or dynamic?)
Predict whether email is spam	**Static or Dynamic (How quickly spammers change)**
Android voice to text	**Static or Dynamic (Global vs personalized)**
Shopping ad conversion rate	**Static**

1.2.1.1 Reference Architectures

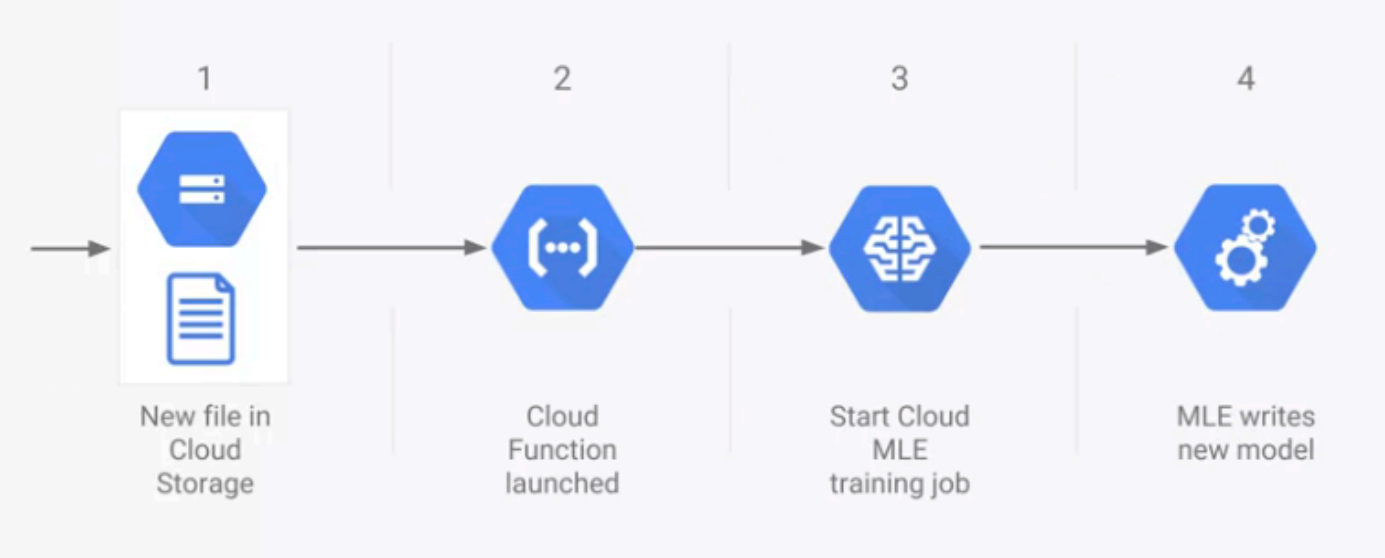
- **Static Training**: Model trained once and pushed to production.



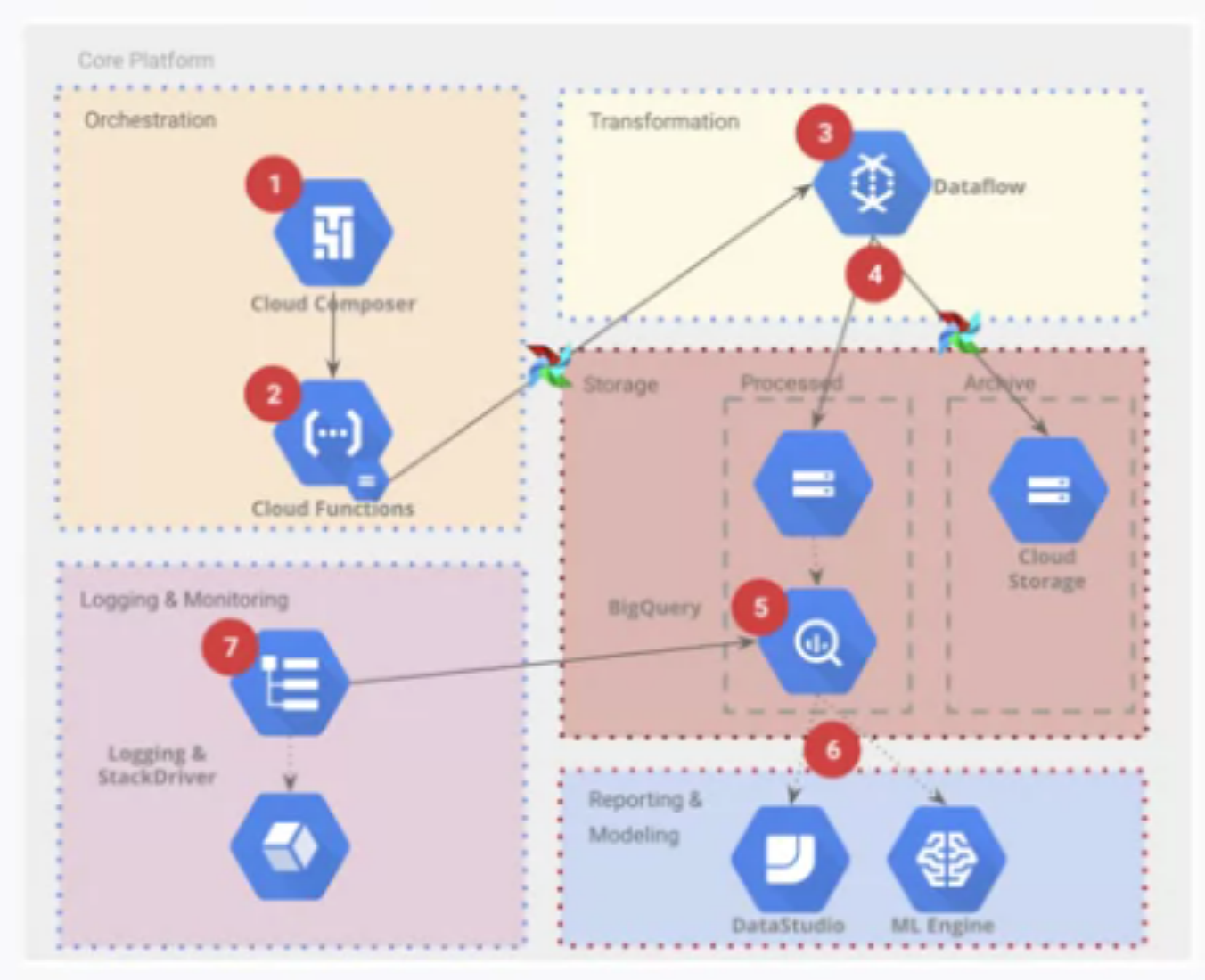
- **Dynamic Training**:
    - There are 3 potential architectures



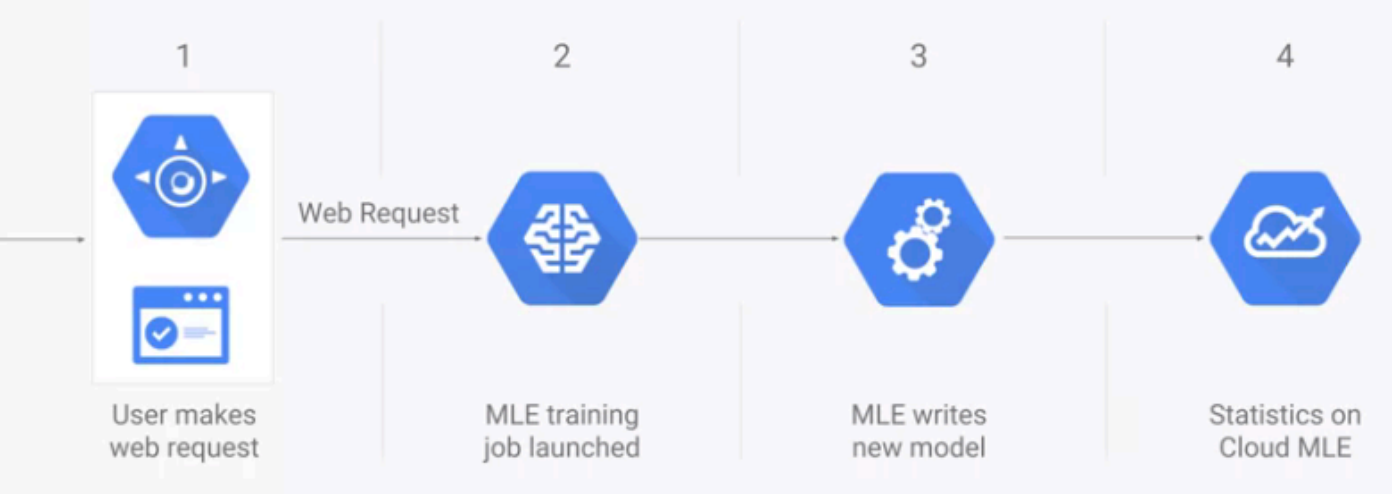
    - **Using Cloud Function (**async training**)**



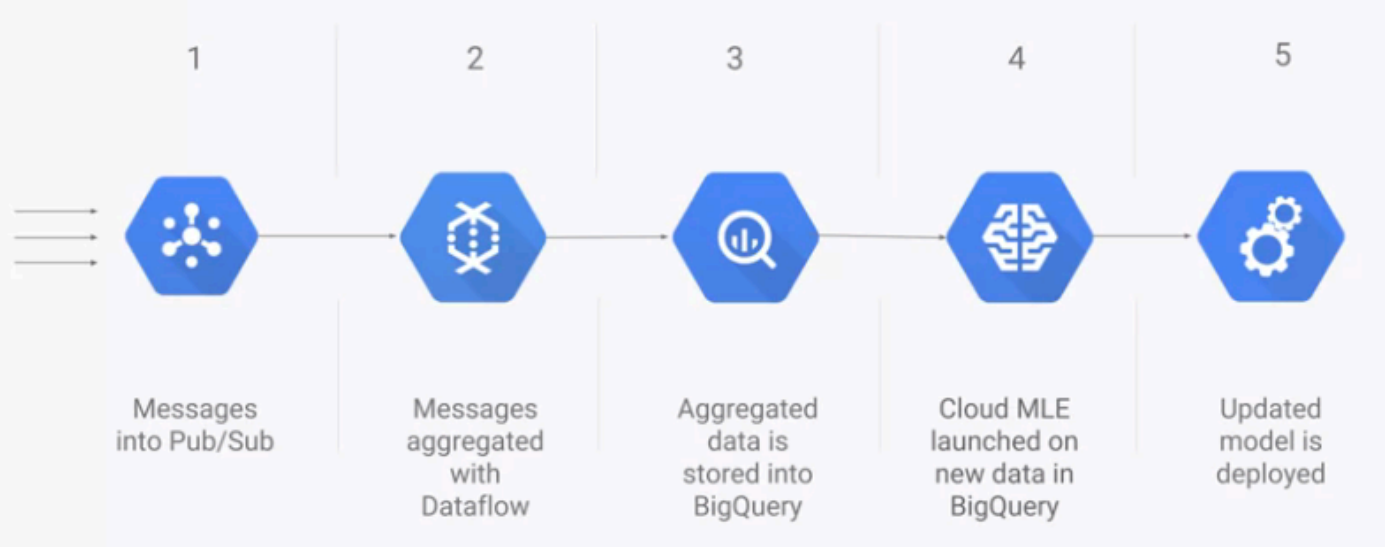
- Example using cloud composer (triggers cloud functions)



- **Using AppEngine** (user making a web request)

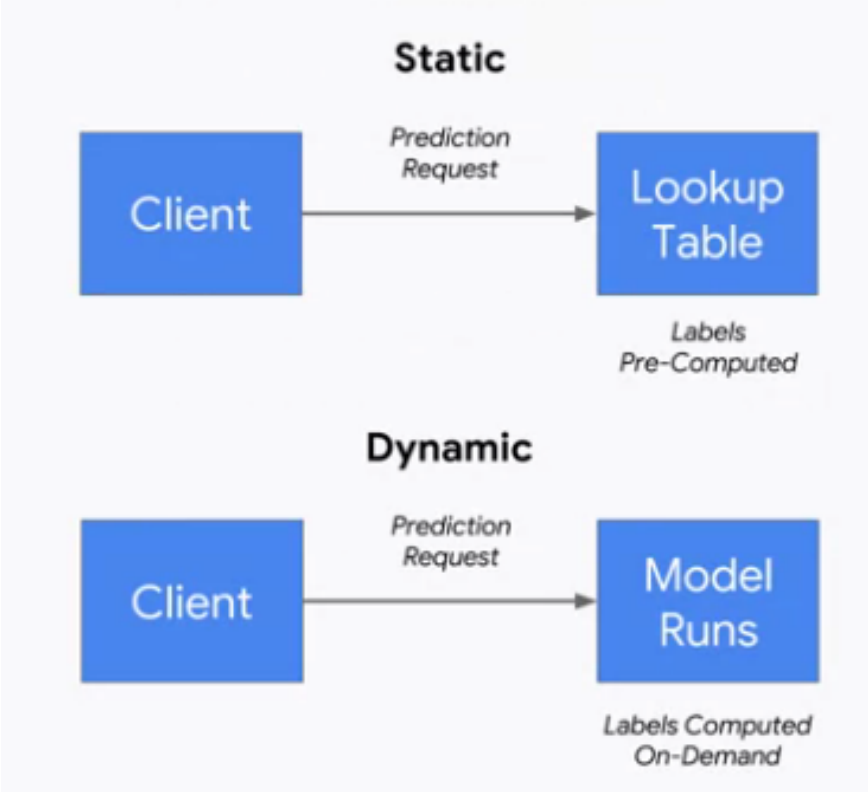


- **Using DataFlow** (continuous training)



1.2.2 Serving Design Decisions

- In designing our serving architecture, one of our goals is to minimize average latency.
- We have 2 options:



  - **Static Serving**: Precompute the predictions and persist in a lookup table (like a cache)
  - **Dynamic Serving**: Use the ML model as is to obtain predictions

- In choosing between the two, there's a space-time trade-off.

Static	Dynamic
Higher Storage Cost	Lower Storage Cost
Low, Fixed Latency	Variable Latency
Lower Maintenance	Higher Maintenance
Space intensive	Compute intensive

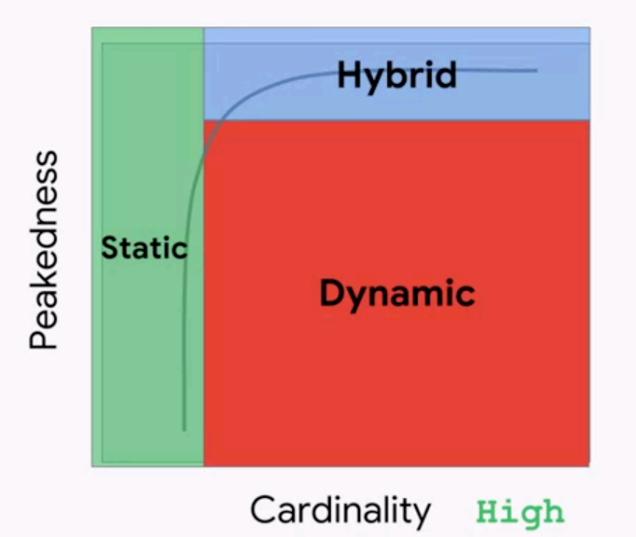
- The choice of which to use, static or dynamic serving is determined by considering how important ***latency, storage, and CPU costs*** are.
- Sometimes it can be hard to express the relative importance of these three areas.
- Helpful to consider static and dynamic serving through another lens, peakedness and cardinality.

    - **Peakedness**
        - Refers to the extent to which the distribution of the prediction workload is concentrated. (like kurtosis)
        - For example, a model that predicts the next word given the current word would be highly peaked because a small number of words account for the majority of words used.
        - In contrast, a model that predicted quarterly revenue for all sales verticals, in order to populate a report, will be run on the same verticals every time and with the same frequency for each. So, it would have very low peakedness.

    - **Cardinality**
        - Refers to the number of values in a set.
        - In this case, the set is the set of all possible things we might have to make predictions for.
        - A model predicting sales revenue given organization division number, would be fairly low cardinality.

        - A model predicting lifetime value given a user for an e-commerce platform would be very high cardinality, because the number of users and the number of characteristics of each user, are likely to be quite large.

- Taken together, these two criteria create a space.



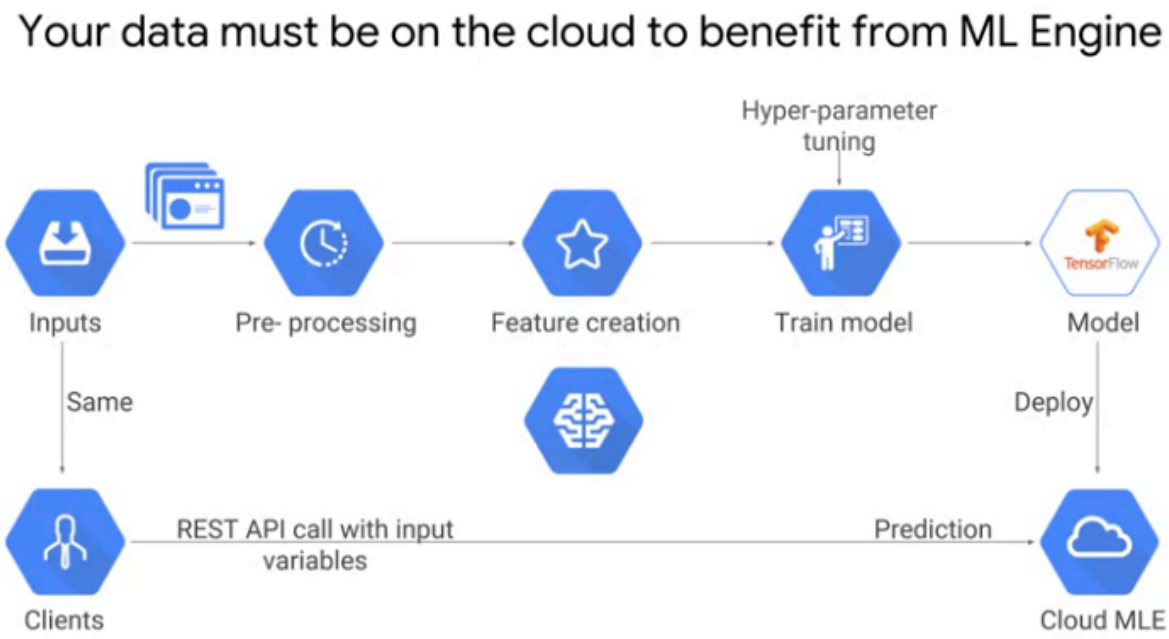
    - **Static**: When the cardinality is sufficiently low, we can store the entire expected prediction workload.
    - **Dynamic**: When the cardinality is high because the size of the input space is large, and the workload is not very peaked, we would probably want to use dynamic training.
    - **Hybrid**: In practice, we often would choose a hybrid of static and dynamic, where we statically cache some of the predictions, or responding on demand for the long tail. This works best when the distribution is sufficiently peaked.

- **Example problems**

Problem	Inference style (static or dynamic?)
Predict whether email is spam	**Dynamic**
Android voice to text	**Dynamic / Hybrid**
Shopping ad conversion rate	**Static**

1.3 Data Ingestion for Cloud based systems

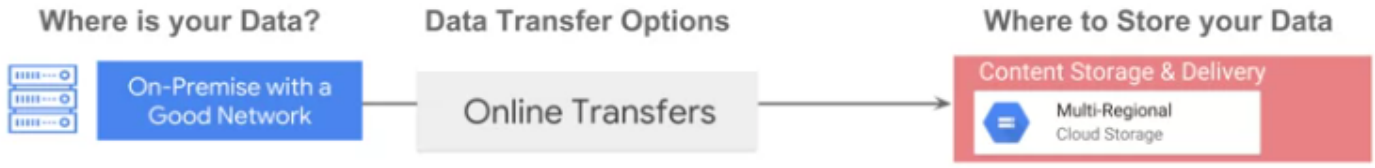
- To use Google cloud ML engine, all data needs to be on the cloud.



- Common data migration challenges
  - Too much data
  - Too little bandwidth
  - Firewalls, checksum, security etc.
  - Few resources
  - Less Time.

- Depending on the use case, there are different ways to transfer data to google cloud.

- Below are some options provided:
    - **Online Transfer**

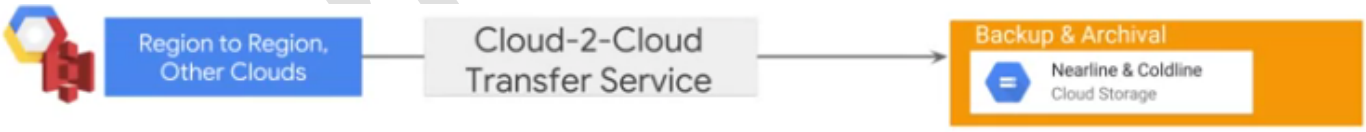


        - Gsutil is a command line interface that many of our customers use to get their data into GCS.
        - Google also offers easy drag-and-drop folders in Chrome

    - **Transfer Appliance**

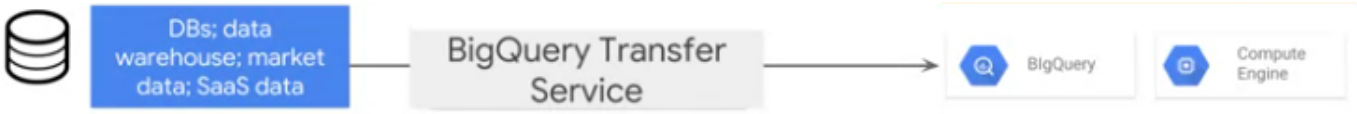
        - A rackable high-capacity storage server that is shipped by Google to fill up and ship back.

    - **Cloud to Cloud (Cloud storage transfer service)**



        - Migrating from other clouds to GCP or transferring data for backup or even transfer between regions in GCP

- **Big Query Transfer Service**



  - Automates loading data into BigQuery from YouTube, Google AdWords, and Double-click.

1.3.1 On-Premise Data

- **Option 1:**
  - Easiest approach is to drag and drop files into Cloud Storage using a web browser.

- **Option 2:**
  - Use the JSON APIs to manage buckets and store data

- **Option 3: Most used**
  - Use gsutil command
  - Gsutil is a python application that lets you access Google Cloud Storage from the command line.
  - Example to copy all txt files to a bucket

```
gsutil cp *.txt gs://my-bucket
```

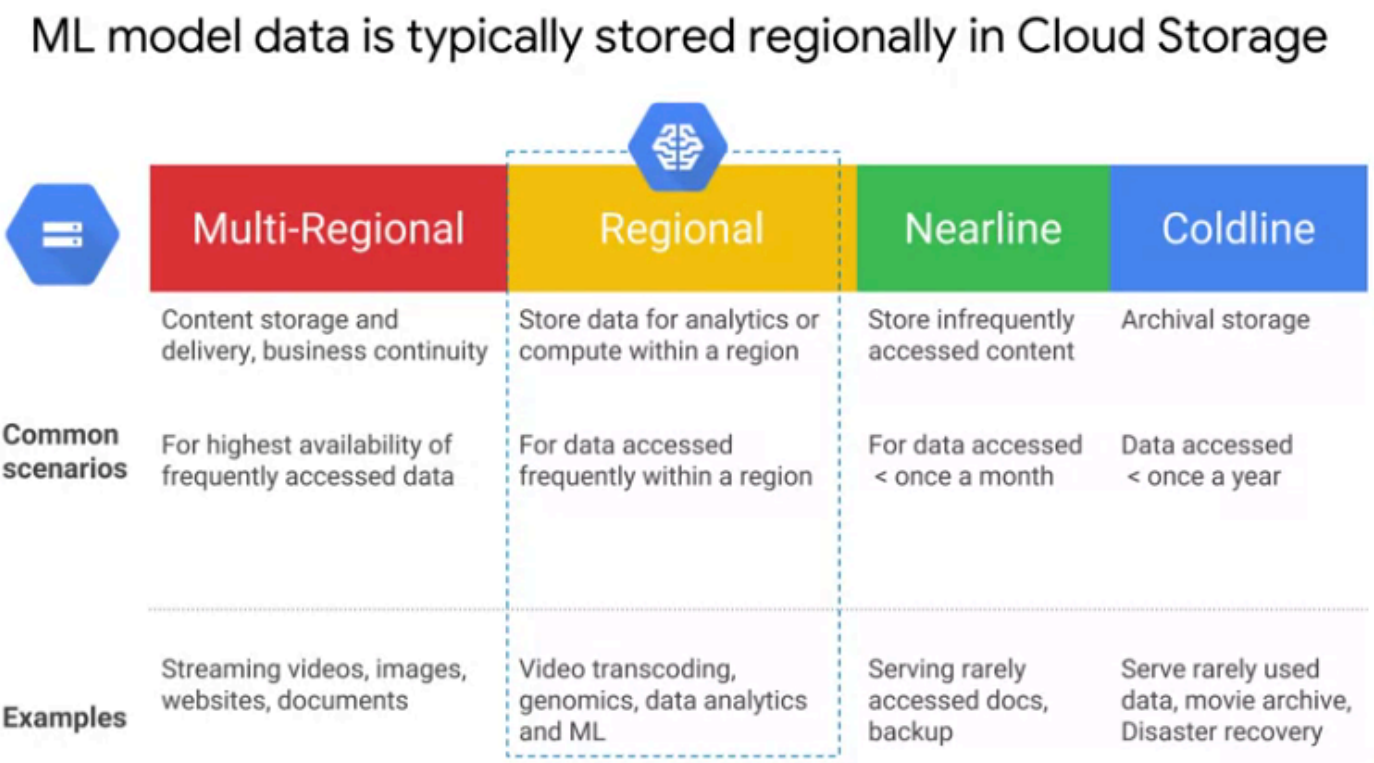
  - More generally

Use multithreaded transfers to Google Cloud Storage

```
gsutil -m cp -r
[SOURCE_DIRECTORY]
gs://[BUCKET_NAME]
```

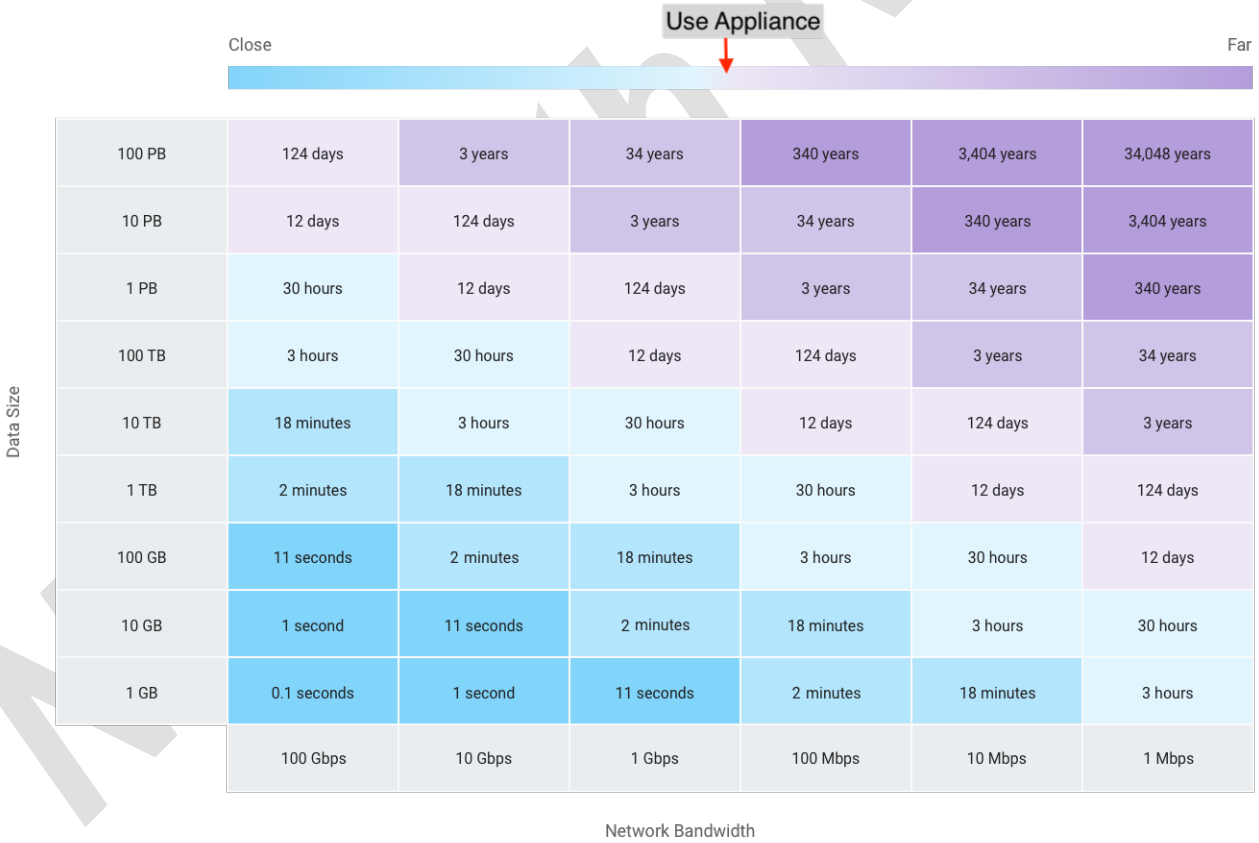
Include -m to enable multi-threading

- Determine type of Cloud storage to use based on use case



1.3.2 Large Datasets

- Best option for large datasets is to use the Google Transfer service which is an appliance shipped by Google to load the data in and send back.
- Each appliance can hold up to 1 Peta Byte.
- Large here is subjective.
    - Generally, around 60TB of data is considered large.
    - However, both data size and network bandwidth must be considered before deciding to use the transfer service.
    - Below are some considerations to use appliance
        - If you have 60TB+, or
        - 1 TB+ and a 10 MBPS network, or
        - If it takes over 1 week to transfer data

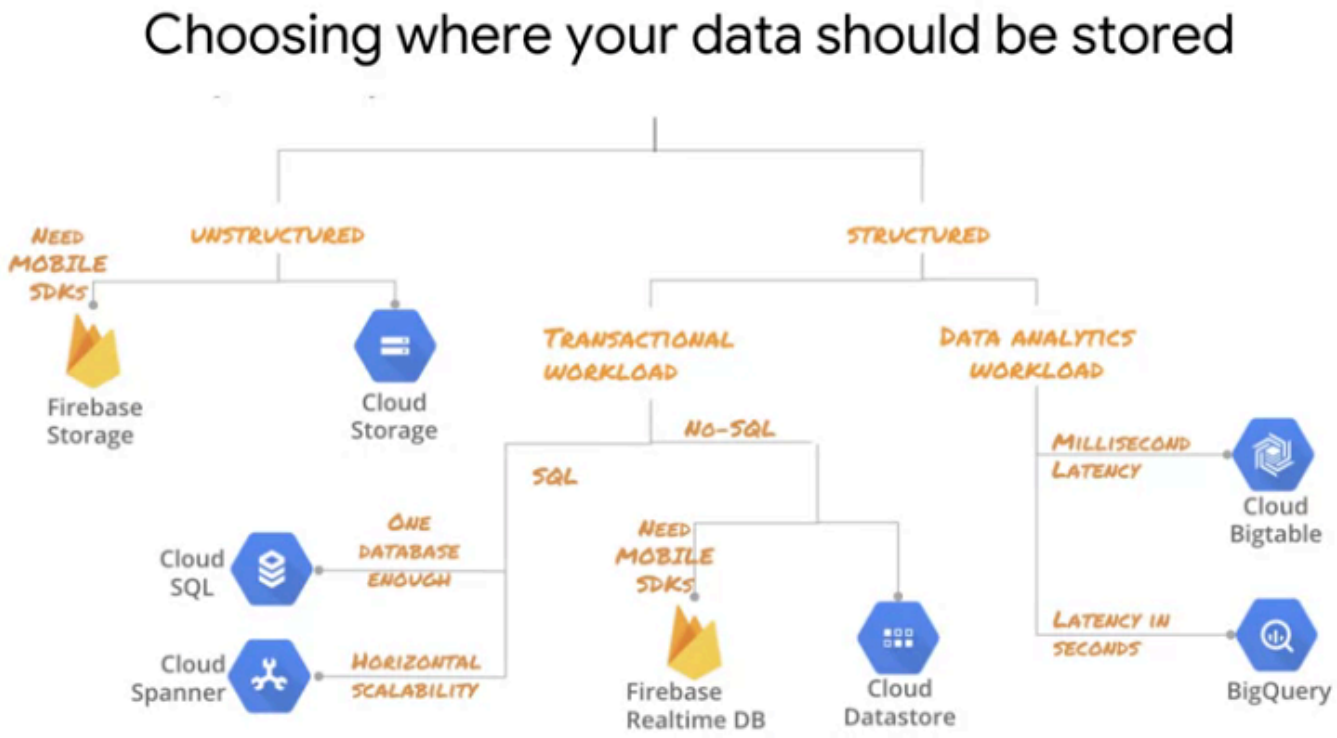


1.3.3 Cloud to Cloud transfers

- The data sink (i.e destination) is always Google Cloud Storage
- Can transfer from any Cloud source like S3 or even Google Cloud (for regional transfers)
- Can setup a one-time transfer or ongoing job for transfer.

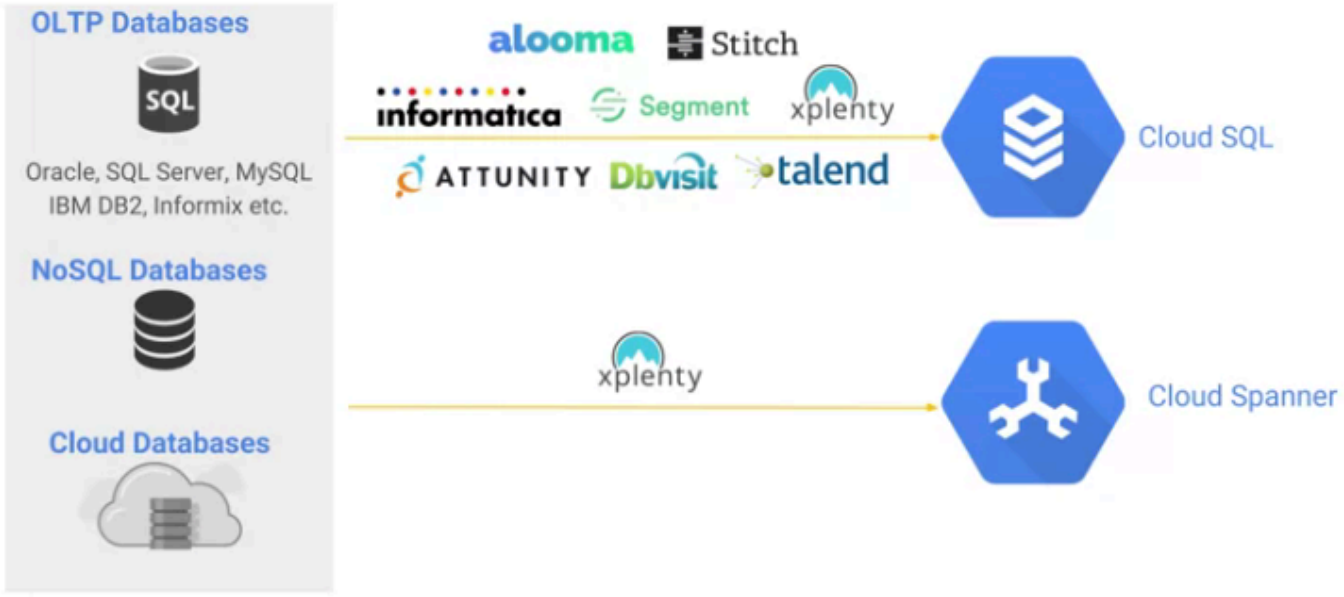
1.3.4 Existing Databases

- Many options to transfer from existing database to GCP



- For existing hadoop workloads, use dataproc. Transfer using a partner.

- Use partners to transfer data from existing databases to fully managed services like Cloud SQl\Spanner



1.3.5 Summary of migration or transfer options

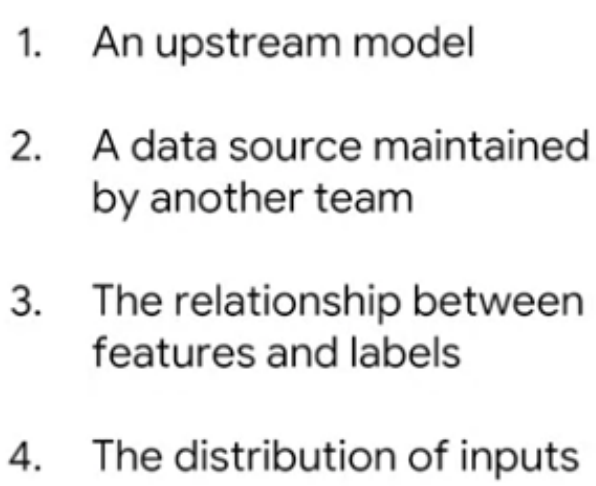
Where you're moving data from	Scenario	Suggested products
Another cloud provider (for example, Amazon Web Services or Microsoft Azure) to Google Cloud	—	Storage Transfer Service
Cloud Storage to Cloud Storage (two different buckets)	—	Storage Transfer Service
Your private data center to Google Cloud	Enough bandwidth to meet your project deadline for less than 1 TB of data	`gsutil`
Your private data center to Google Cloud	Enough bandwidth to meet your project deadline for more than 1 TB of data	Storage Transfer Service for on-premises data
Your private data center to Google Cloud	Not enough bandwidth to meet your project deadline	Transfer Appliance

1.4 Designing adaptable ML systems

- ML systems have many dependencies that need to be handled.
    - For example, model weights depend on training data. Additionally, similar data will yield similar instructions, and finally other people including other teams and our users create our data.
    - With mismanaged dependencies the model's accuracy might go down or the system might become unstable.
    - Sometimes the errors are subtle, and your team may end up spending an increasing proportion of its time debugging.

1.4.1 Managing Data Dependencies

- The following are some things that are very likely to keep changing:



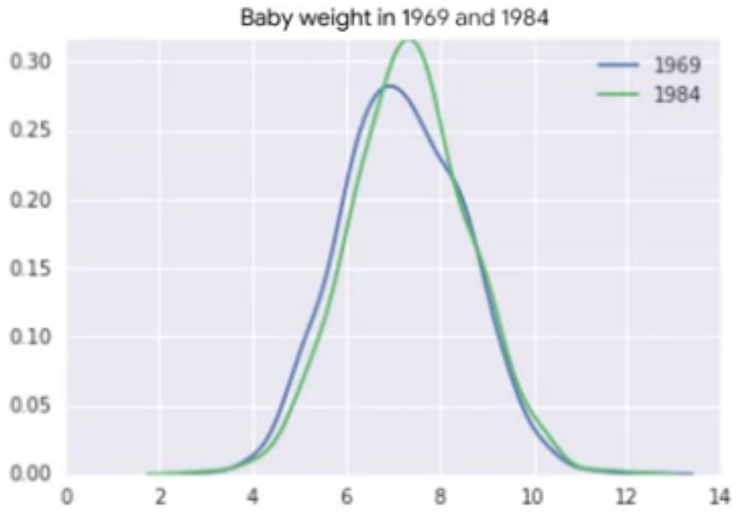
- Some ways to handle such changes:
    - Think carefully before consuming data from sources when there's a chance you won't know about changes to them.
    - Make a local version of upstream models and update it on your schedule.
    - Features should always be scrutinized before being added.
    - All features should be subjected to leave one out evaluations to assess their importance.

1.4.1.1 Change in distribution

- The statistical term for changes in the likelihood of observed values like model inputs is changes in the distribution.
- And it turns out that the distribution of the data can change for all sorts of reason.

1.4.1.1.1.1 Distribution of the label changes

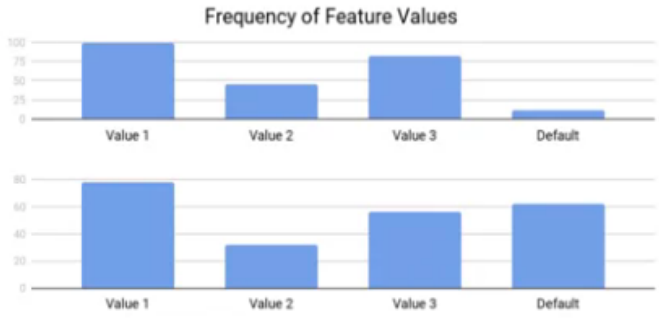
- For example, in the natality dataset, baby weight changed over time. It peaked in the 1980s and has since been declining. In the graph, we see the distributions of baby weights for 1969 and in 1984.Note that in 1969 babies weighed significantly less than they did in 1984.



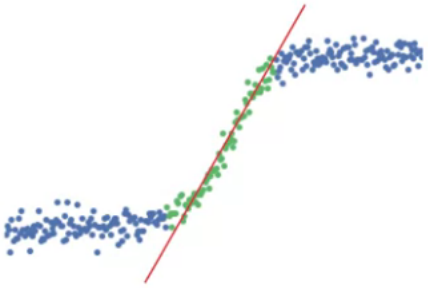
  - When the distribution of the label changes, it could mean that the relationship between features and labels is changing as well.
  - At the very least, it's likely that our model's predictions will be significantly less accurate.

1.4.1.1.1.2 Distribution of the features change

- For example, using zip codes to train a model. Surprisingly, ZIP codes aren't fixed. The government releases new ones and deprecates old ones every year. Now as a ML practitioner, you know that ZIP codes aren't really numbers. So you've chosen to represent them as categorical feature columns, but this might lead to problems because if you use a vocabulary and zip codes change, the model could get skewed towards the default.



  - In this scenario, the model is asked to make predictions on points in space that are far away from what it has seen in the training data. This is called **Extrapolation**.
    - **Extrapolation** means to generalize outside the bounds of what we've previously seen.
    - **Interpolation** is the opposite, it means to generalize within the bounds of what we've previously seen.
  - For example, let's say the model got to see the green data, but not the blue data. The red line reflects a linear regression on the green data.



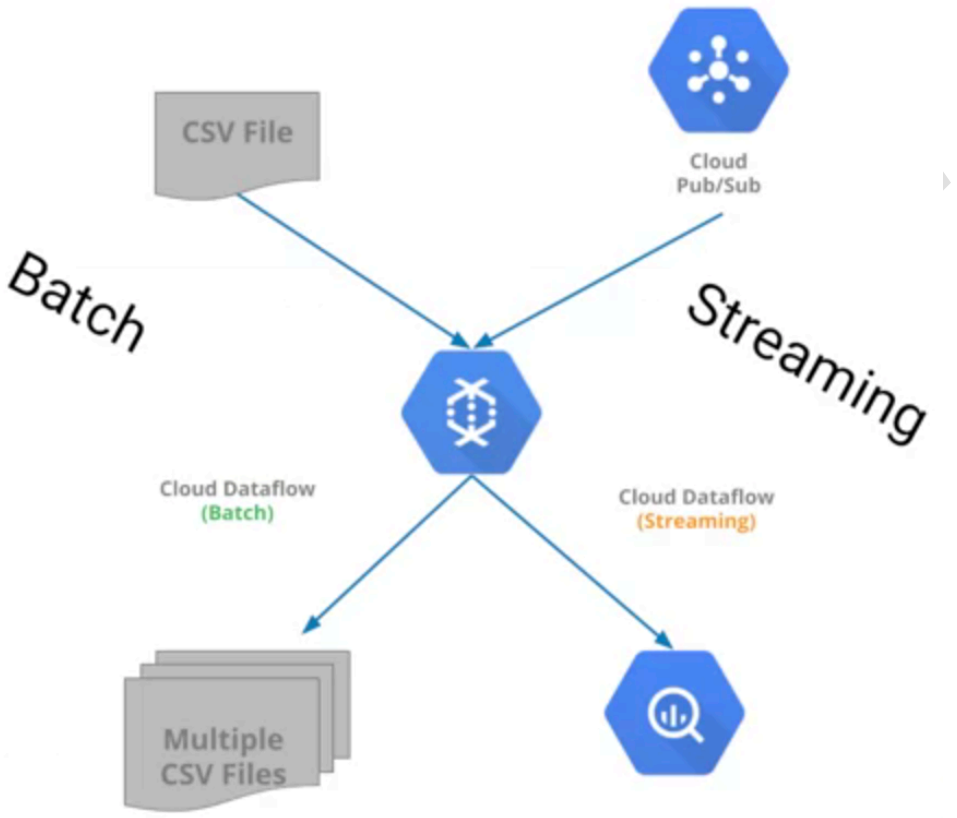
    - Predictions in the green region are interpolated and reasonably accurate.
    - In contrast, predictions in the blue ribbon are extrapolated and are increasingly inaccurate the farther we get from the green region.

1.4.1.1.1.3 Handling changes in distributions

- Monitor descriptive statistics for our inputs and outputs
- Monitor the models' residuals (that is the difference between its predictions and the labels) as a function of the input.
- Use custom weights in our loss function to emphasize data recency
- Use dynamic training architecture and regularly retrain our model
- ***Ablation analysis***: The value of individual features are computed by comparing it to a model trained without it. Helps determine what features are really required.
    - Legacy features are older features that were added because they were valuable at the time. But since then, better features have been added which have made them redundant without our knowledge.
    - Bundled features on the other hand, are features that were added as part of a bundle which collectively are valuable but individually may not be.
    - Both of the above features represent additional unnecessary data dependencies.
- It's impossible to have models unlearn things that they've learned already, but one thing you can do is to roll back your model state to a time prior to the data pollution.
    - Of course, in order to do this, you'll need infrastructure that automatically creates and saves models as well as their meta information.

1.4.2 Handling Training Serving Skew

- This refers to differences in performance that occur as a function of differences in environment.

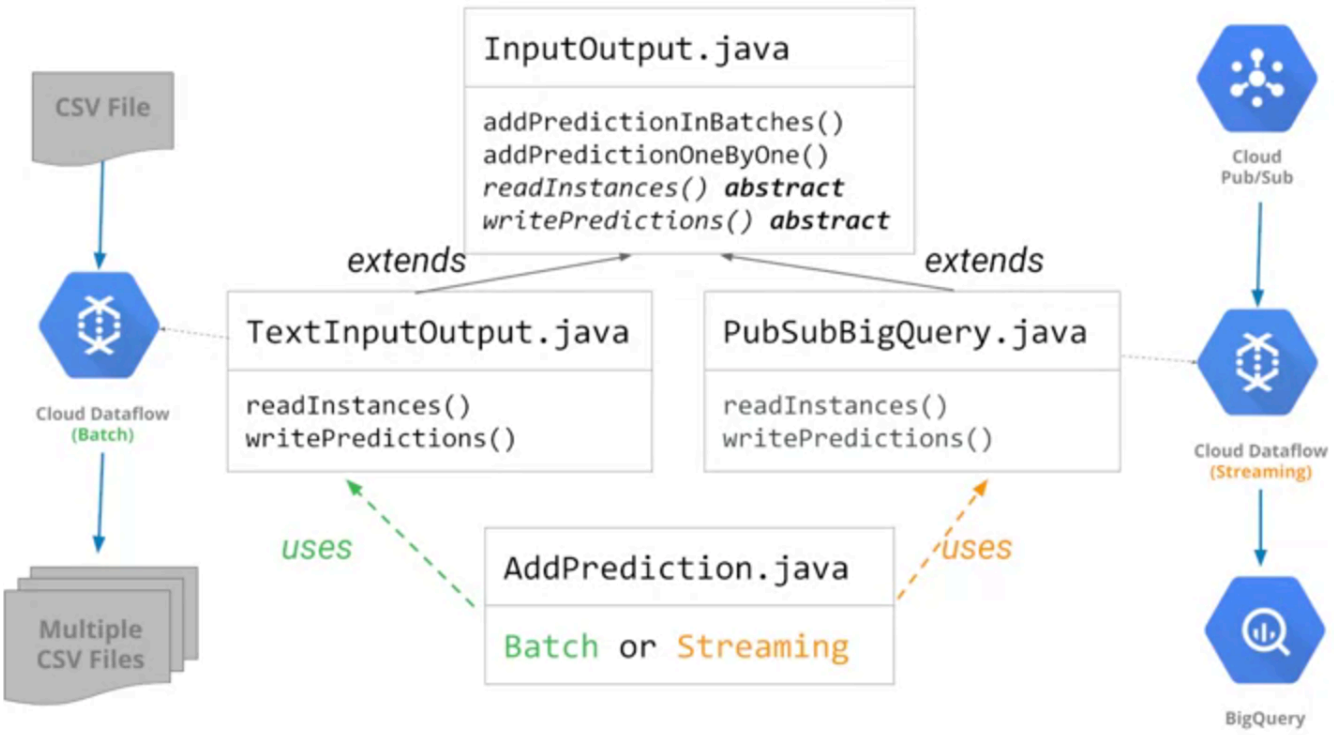


- This is caused by any of the following:
    - **Data Related**
        - A discrepancy between how you handle data in the training and serving pipelines,
        - A change in the data between when you train and when you serve,
        - A feedback loop between your model and your algorithm.
    - **Environment related**
        - For example, that in your development environment you have version two of a library, but in production you have version one.

- The libraries may be functionally equivalent, but version two is highly optimized and version one isn't. Consequently, predictions might be significantly slower or consume more memory in production than they did in development.
- Also possible that version one and version two are functionally different perhaps because of a bug.
- Also possible that different code is used in production versus development, perhaps because of recognition of one of the other issues, but though the intent was to create equivalent code, results were imperfect.

- One of the best ways of mitigating training serving skew is to write code that can be used for both development and production.
    - If development production were the same, this would be relatively trivial, but often they're different.

- We can use polymorphism in the code to abstract the environment dependent aspects of the code while also reusing the parts of the code that need to remain functionally equivalent.
    - We could have different pipelines for the input and output steps during training vs serving. But the intermediate step where we use the model to make predictions is the same.



1.5 Designing High-Performance ML Systems

- Here look solely at infrastructure performance.
- Performance related to model accuracy is covered later.

1.5.1 High performance training

Considerations include:

- **Time to train**
  - If it takes six hours to train a model on some hardware or software architecture, but only three hours to train the same model to the same accuracy on a different hardware or software architecture, the second architecture is twice as performant as the first one.

- **Budget to train**
  - You might be able to train faster on better hardware, but that hardware might cost more, so you may have to make the explicit choice to train on slightly slower infrastructure.

- Here we have 3 levers to adjust for:
  - **Time**
    - how long are you willing to spend on the model training?
    - If you're training a model every day, so as to recommend products the next day, then your training has to finish within realistically less than 18 hours. (considering tests, deploy etc.)
  - **Cost**
    - How much are you willing to spend on model training in terms of computing cost? (assuming you have only 18 hours to train)
    - You don't want to train for 18 hours everyday if the incremental benefit of this is not sufficient.
  - **Scale**
    - Models differ in terms of how computationally expensive they are.
    - Even keeping to the same model, you have a choice of how much data you're going to train on.
      - Generally, the more data, the more accurate the model.
      - But there are diminishing returns to larger and larger data sizes, so your time and cost budget may also dictate the data set size.
    - Have a choice between training on a single, more expensive machine or multiple cheaper machines.
    - Have the choice of starting from an earlier model checkpoint and training for just a few steps.
      - This compromise might allow you to reach the desired accuracy faster and cheaper.
    - In addition, there are ways to tune performance to reduce the time, reduce the cost or increase the scale.

- Model training performance will be bound by one of three things. So knowing what you're bound by, you can look at how to improve performance.
  - **IO**
    - how fast you can get data into the model for each training step.
    - Your ML training will be I/O bound if
      - the number of inputs is large, or
      - Inputs are heterogeneous requiring parsing or
      - if the model is so small the compute requirements are trivial or

        - Input data is on a storage system with very low throughput.
    - Remediation
        - Look at storing the data more efficiently on a storage system with higher throughput or parallelizing the reads.
        - Although it's not ideal, you might also consider reducing the batch size so that you're reading less data in each step.
- **CPU**
    - how fast you can compute the gradient in each training step.
    - Your ML training CPU bound if
        - the I/O is simple but the model involves lots of expensive computations.
        - you're running a model on underpowered hardware.
    - Remediation
        - See if you can run the training on a faster accelerator.
        - GPUs keep getting faster, so move to a newer generation processor.
        - If you're using Google Cloud, you also have the option of running on TPUs.
        - Even if it's not ideal, you might consider using a simpler model, a less computationally expensive activation function or simply just train for fewer steps.
- **Memory**
    - how many weights can you hold in memory so that you can do the matrix multiplications in memory or do you use the GPU or TPU.
    - Your ML training might be memory bound if
        - the number of inputs is really large or
        - if the model is complex and has lots of free parameters.
        - if your accelerator doesn't have enough memory.
    - Remediation
        - See if you can add more memory to the individual workers.
        - Consider using fewer layers in your model.
        - Reducing the batch size can also help with memory bound ML systems.

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

1.5.2 High Performance Inference

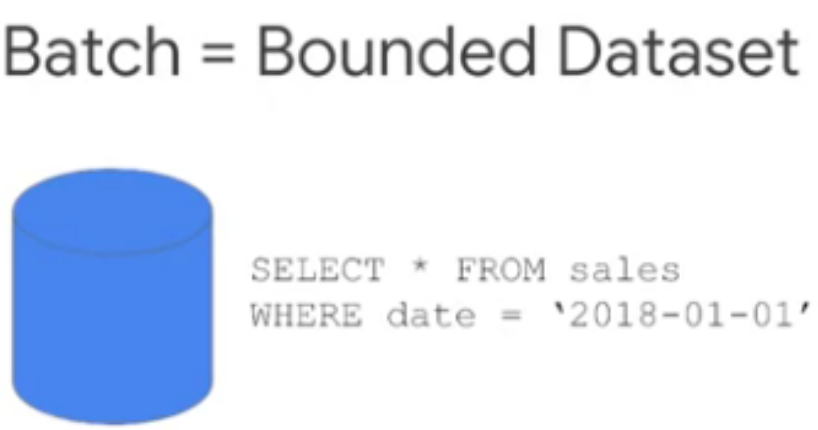
- **Batch Predictions**
  - Considerations similar to training
    - **Time**
      - How long does it take for you to do all of your predictions?
      - if you're doing product recommendations for the next day, you might want recommendations for the top 20 percent of users precomputed and available in five hours, if it takes 18 hours to do the full training.
    - **Cost**
      - What predictions are you doing and how much do you precompute is going to be driven by cost considerations?
    - **Scale**
      - Do you have to do all of this on a single machine or can you distribute it safe to multiple workers?
      - What kind of hardware is available on these workers? Do they for example have GPUs?

- **Online predictions**
  - Here, considerations are very different because we have users waiting for the results.
  - The performance consideration is not how many training steps you can carry out per minute, but how many queries you can handle per second. The unit of this **queries per second** is often called **QPS**. That's the performance target that you need to hit.

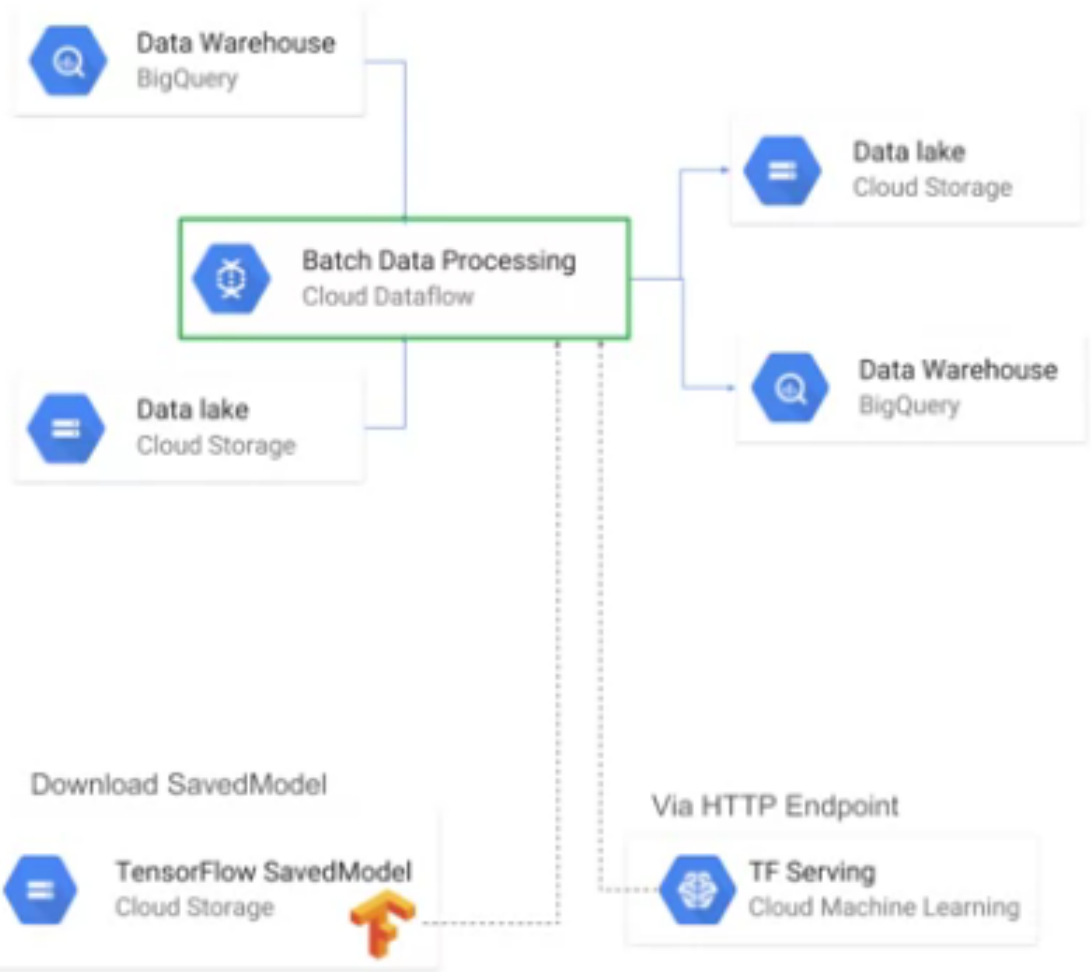
    - You typically cannot distribute the prediction graph, instead you carry out the computation for one end user on one machine.
        - However, you almost always scale out the predictions onto multiple workers.
        - Essentially, each prediction is handled by a microservice and you can replicate, and scale out the predictions using Kubernetes or App Engine
        - Cloud ML Engine predictions or higher-level abstraction, but they are equivalent to doing this.
- Unless you plan to be able to do both batch predictions and online predictions, you will be stuck with a solution that doesn't meet all of your needs.
    - For example, use precomputed batch for top 20% and online for remaining.
- When you design for higher performance, you want to consider training and prediction separately, especially if you will be doing online predictions.

- **Implementation Options**

- Below we explore the hybrid approach using DataFlow.
- In the context of inference, the word batch is used differently from what it means during training. Here we're using batch to refer to a bounded dataset.

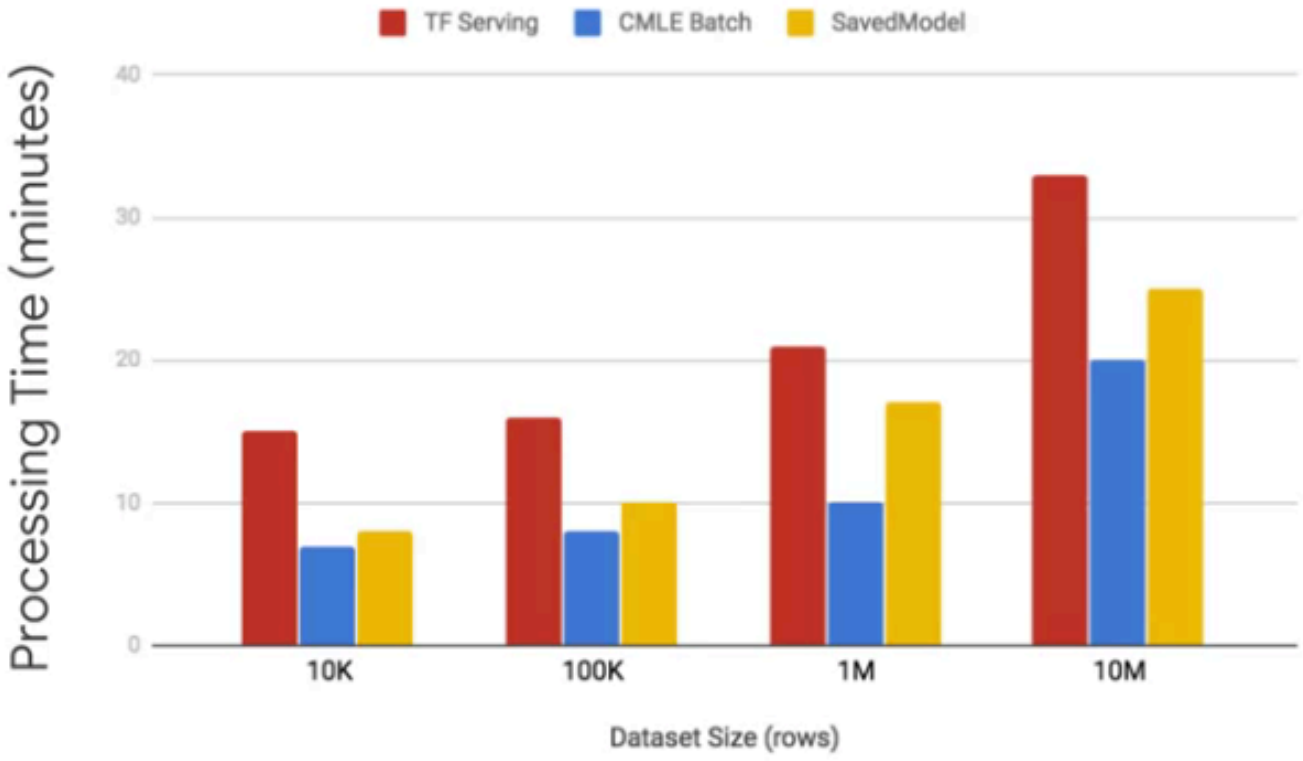


- A typical batch data pipeline reads data from some persistent storage, either a data lake like Google Cloud Storage or a data warehouse like BigQuery. It then does some processing and writes it out to the same or a different format.



- The processing carried on by Cloud Dataflow typically enriches the data with the predictions of an ML model.
- There are 2 ways to get the predictions:
  - using a ***TensorFlow SavedModel*** and loading it directly into the dataflow pipeline from cloud storage.
  - using ***TensorFlow Serving*** and accessing it via an HTTP endpoint as a microservice, either from Cloud ML Engine as shown, or using Kubeflow, running on a Kubernetes engine.

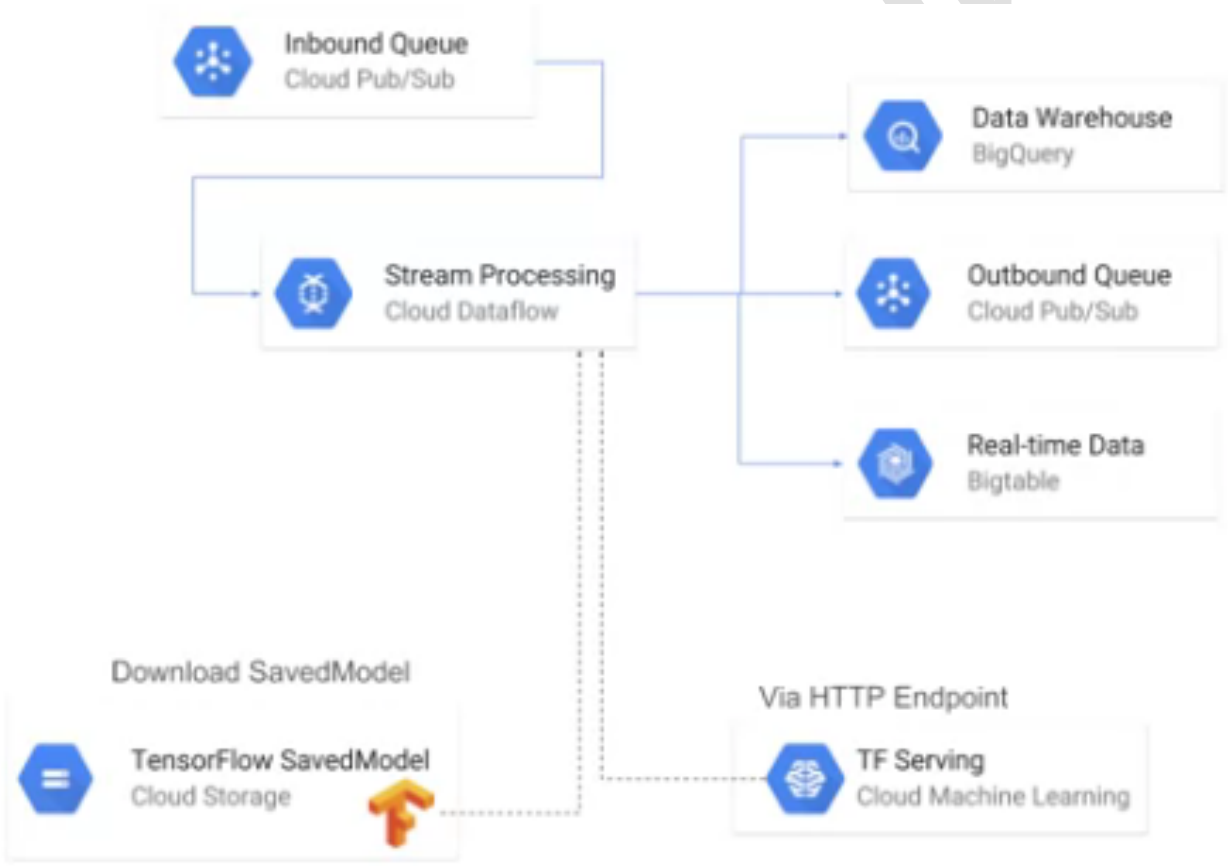
- What option gives the best performance for ***batch pipelines***?
  - **If processing speed is important**:



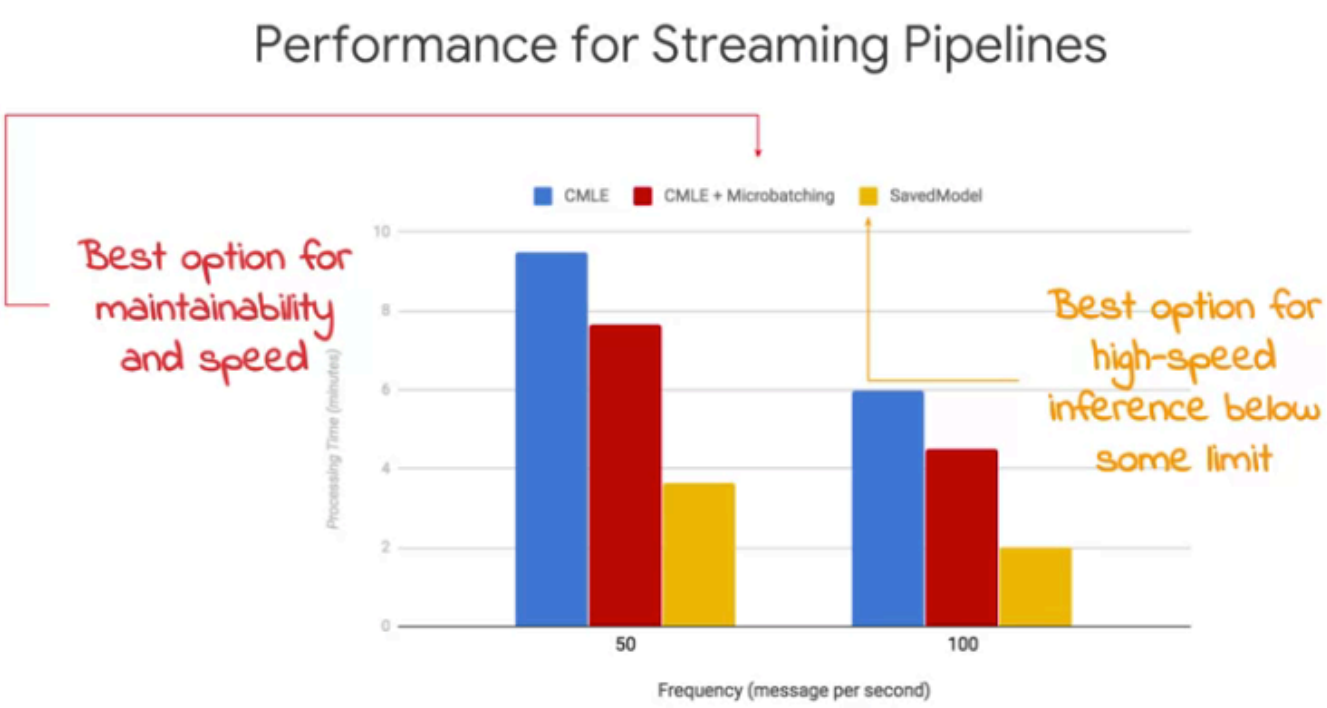
- Cloud ML Engine batch predictions is the fastest.
- next fastest is to directly load the SavedModel into your dataflow job and then invoke it.
- Third would be to use TensorFlow Serving on Cloud ML Engine.

- **If maintainability is important**:
  - Cloud ML Engine batch predictions is still the best.
  - TensorFlow Serving on Cloud ML Engine comes next.
  - Third would be to directly load the SavedModel into your dataflow job.

- A streaming pipeline is similar, except that the input dataset is not bounded.



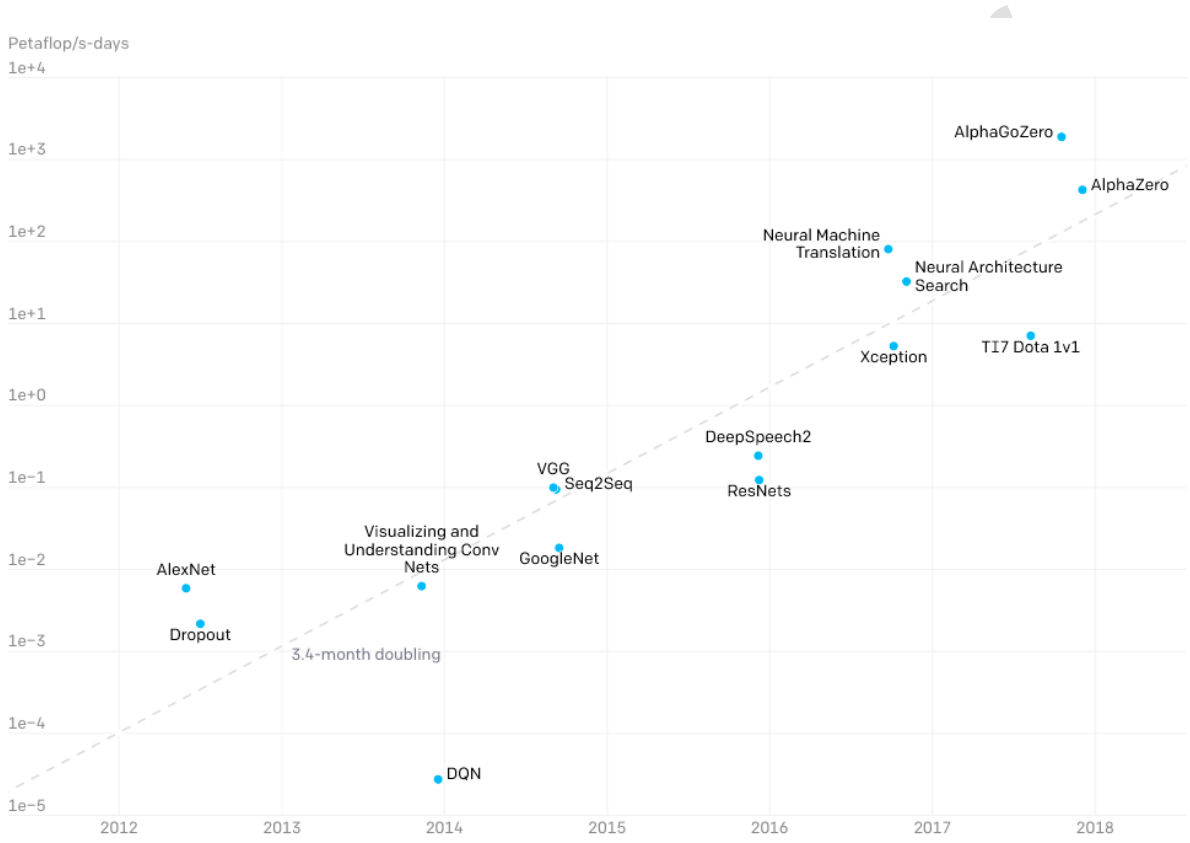
  - So we read it from an unbounded source, like a pub/sub, and we process it with dataflow.
  - You have the same options for predictions as discussed above for batch processing.



- For streaming pipelines, the SavedModel approach is the fastest.
- The Cloud ML Engine approach is much more maintainable.
- Another thing to keep in mind is that as the number of queries per second keeps increasing, at some points the saved model approach will become infeasible. But the Cloud ML Engine approach should scale indefinitely.

1.5.3 Distributed Training

- If you want high performance training and scalable inference chances are that the code that you write has to work on CPUs, GPUs and maybe even TPUs.
- The growth in algorithm complexity and data size means that distributed systems are pretty much in necessity when it comes to machine learning.



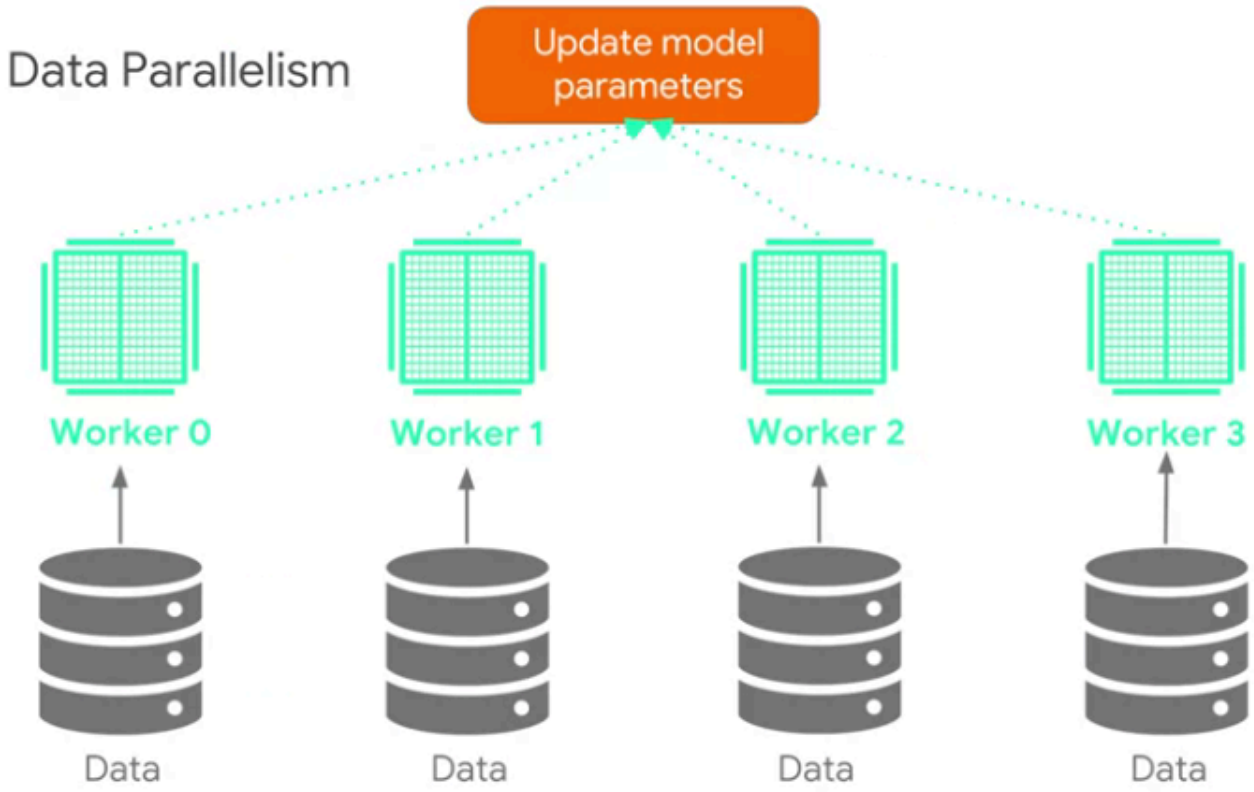
- AlexNet which started the deep learning revolution in 2013 required less than 0.01 petaflops per second day in compute per day for training.
- By the time you get to neural architecture search, the learn to learn model published by Google in 2017,you need about a 100 petaflops per second per day or a thousand times more compute than you needed for AlexNet.

- Training time is a critical factor to consider

  - If your training takes from a few minutes to a few hours it will make you productive and happy. You can try out different ideas first.
  - If it takes a few days, you can still deal with it by running a few things in parallel.
  - If it starts to take a week or more, your progress will slow down because you cannot try out new ideas quickly and
  - if it takes more than a month then that's probably not even worth thinking about and this is not an exaggeration.

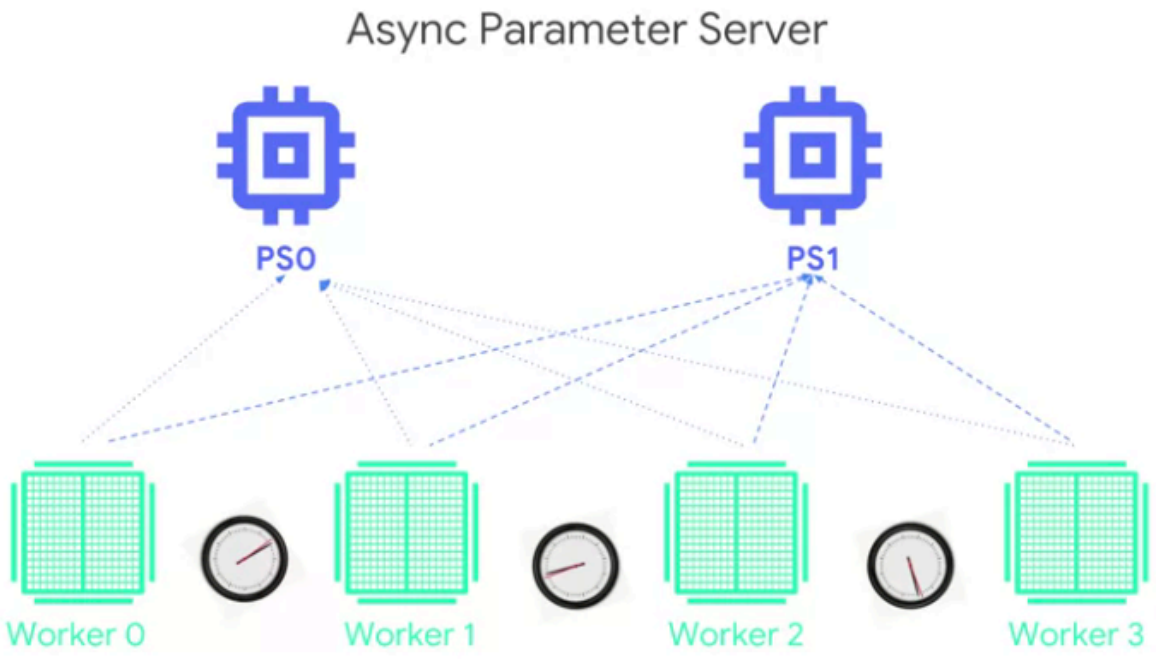
- **How does this distributed training work?**
  - There are a few different ways to scale training.
  - Which one you choose depends on the size of your model, the amount of your training data, and the devices that are available to you.
  - Two common architectures
    - Data Parallelism
    - Model Parallelism

1.5.3.1 Data Parallelism



- The most common architecture for distributed training. (think map reduce approach)
- Data parallelism you run the same model and computation on every device, but you train each of them using different training samples.
    - Each device computes loss and gradients based on the training samples it sees.
    - Then we'll update the model's parameters using all of these gradients.
    - The updated model is then used in the next round of computation.
- There are currently two approaches used to update the model using gradients from various devices.
    - Asynchronous parameter server architecture
    - Synchronous all Reduce

1.5.3.1.1 Async Parameter Server

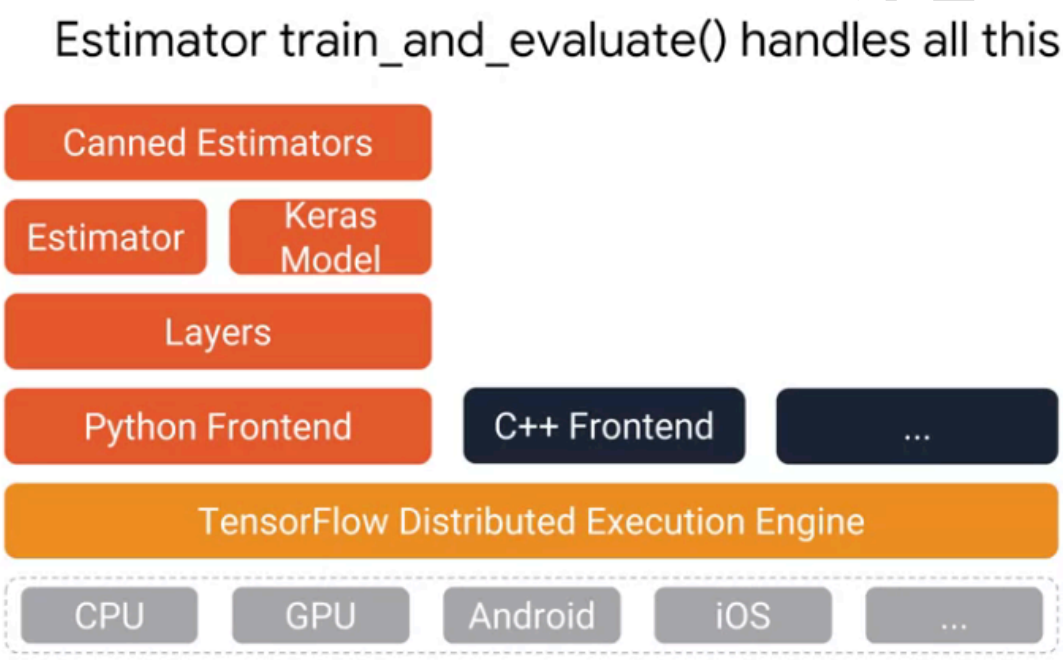


- **Works as follows**:
  - Some devices are designated parameter servers and others as workers.
  - Each worker independently fetches the latest parameters from the parameter server and computes gradients based on a subset of training samples.
  - It then sends the gradients back to the parameter server which updates its copy of the parameters with those gradients.
  - Each worker can do this independently.

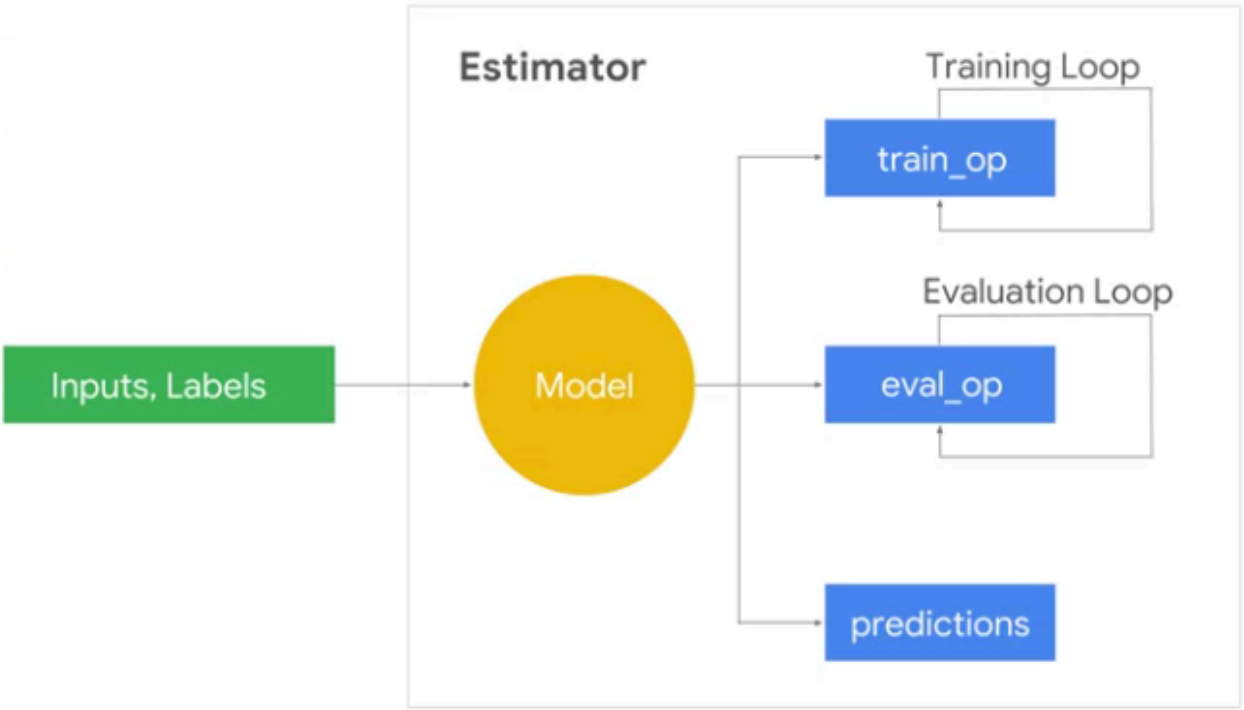
- **Benefits of approach**
  - Scales well to a large number of workers.
  - This methodology has worked very well for many models.
  - Workers might be preempted by higher priority production jobs, or machine might go down for maintenance or maybe sometimes there's an asymmetry between the workers. But these don't hurt the scaling because the workers are not waiting for each other.

- **Downside to approach**
    - Workers can get out of sync. They compute parameter updates based on stale values and then this can delay convergence.

- This architecture is used by **Tensorflow Estimator method train_and_evaluate()**.

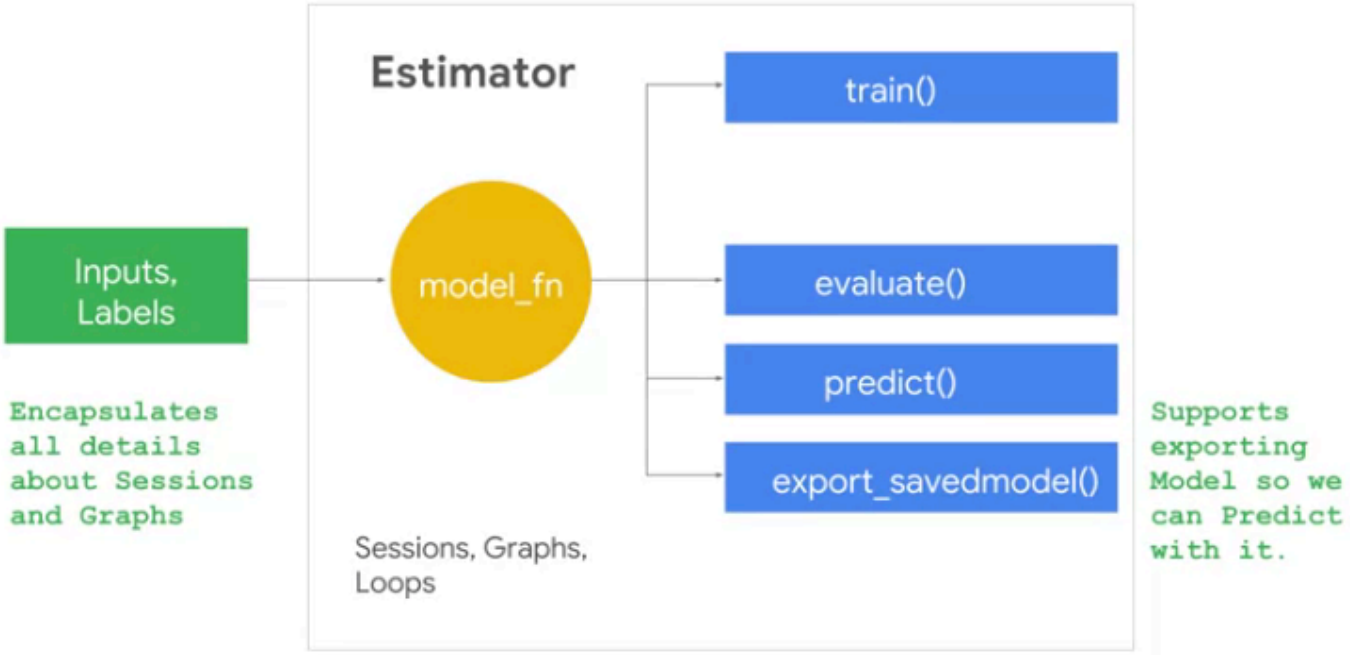


Estimator contains the implementation of three functions



- Estimator contains the implementation of three functions: training, evaluation, and serving. (as shown above)
- You provide the model as a function which returns the ops required when given inputs.
- Estimator then provides a standard interface that you can use to perform the tasks that you need with that model.
    - It encapsulates all the details about sessions and graphs and handles the details of actually training or running the model.

By encapsulating details about sessions and graphs, it also supports exporting the model for serving



- ***Example Code***

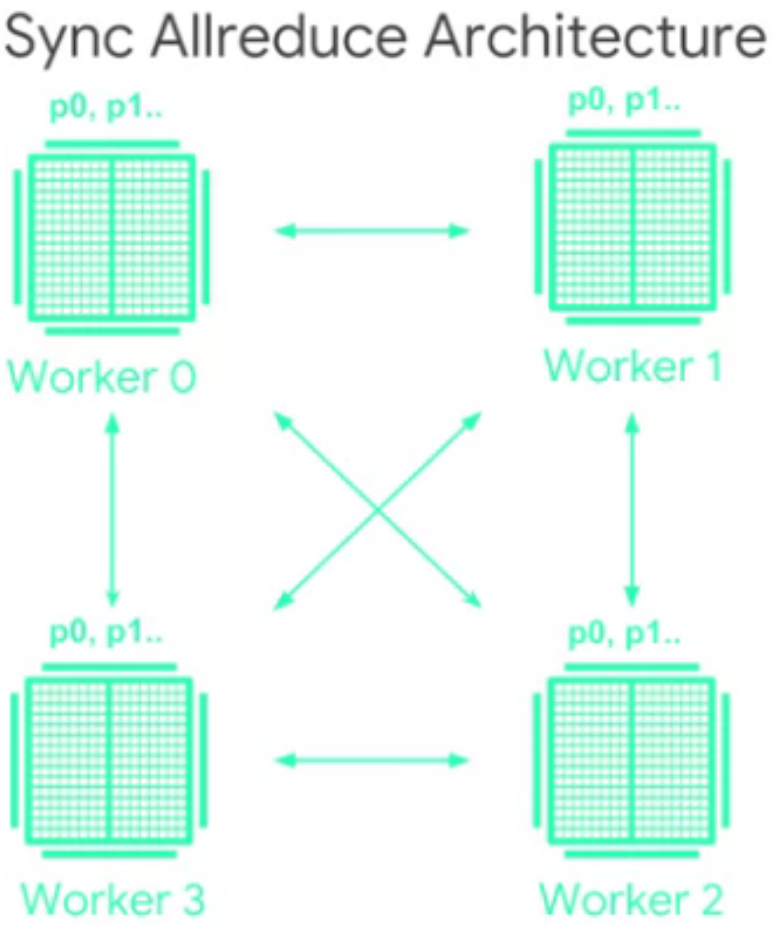
train_and_evaluate bundles together a distributed workflow


```
def train_and_evaluate(output_dir, config, params):
  features = [tf.feature_column.embedding_column(...),
              tf.feature_column.bucketized_column(...)]
  estimator = tf.estimator.Estimator(model_fn = simple_rnn,
                          model_dir = output_dir)
  train_spec = tf.estimator.TrainSpec(input_fn = get_train(),
                                      max_steps = 1000)
  exporter = tf.estimator.LatestExporter('exporter', serving_input_fn)
  eval_spec = tf.estimator.EvalSpec(input_fn = get_valid(),
                                    steps = None,
                                    exporters = exporter)
  tf.estimator.train_and_evaluate(estimator, train_spec, eval_spec)

train_and_evaluate(output_dir)
```

Runs training, evaluation, etc. on Cloud ML

- A train and evaluate function bundles together an estimator with input functions for reading a particular dataset as well as other configuration options. So, you simply specify an estimator which might be a pre-canned model. You then pass that function to train and evaluate, which will pick up the configuration from the environment and run the appropriate low-level code for you

1.5.3.1.2 Sync All Reduce

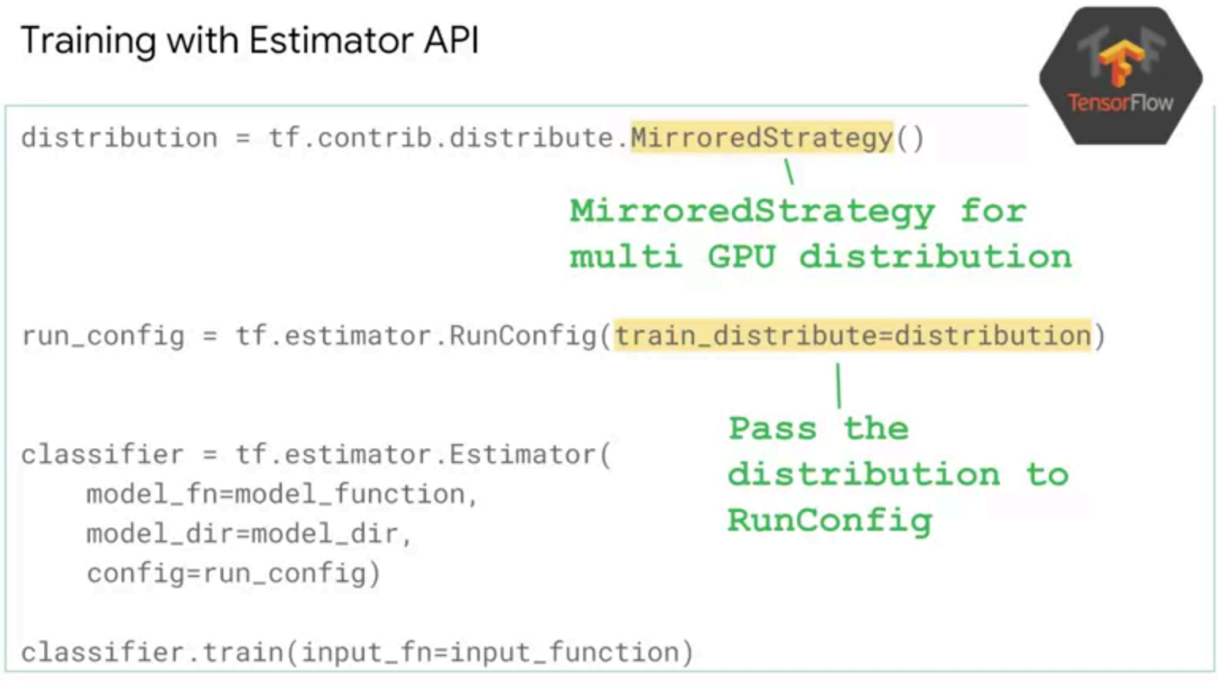


- With the rise of powerful accelerators such as TPUs and GPUs over the last few years, this approach has become more common.

- **Works as follows:**
  - Each worker holds a copy of the model's parameters.
    - There are no special service holding these parameters.
  - Each worker computes gradients based on the training samples that they see
  - They communicate between themselves to propagate the gradients and update their parameters.
  - All the workers are synchronized
    - conceptually the next forward pass does not begin until each worker has received the gradients and updated their parameters.
- With fast devices in a controlled environment, the variance between the step time on each worker is quite small.
- When combined with strong communication links between the workers the overhead of synchronization is also small.
- Overall, this approach can lead to faster convergence.

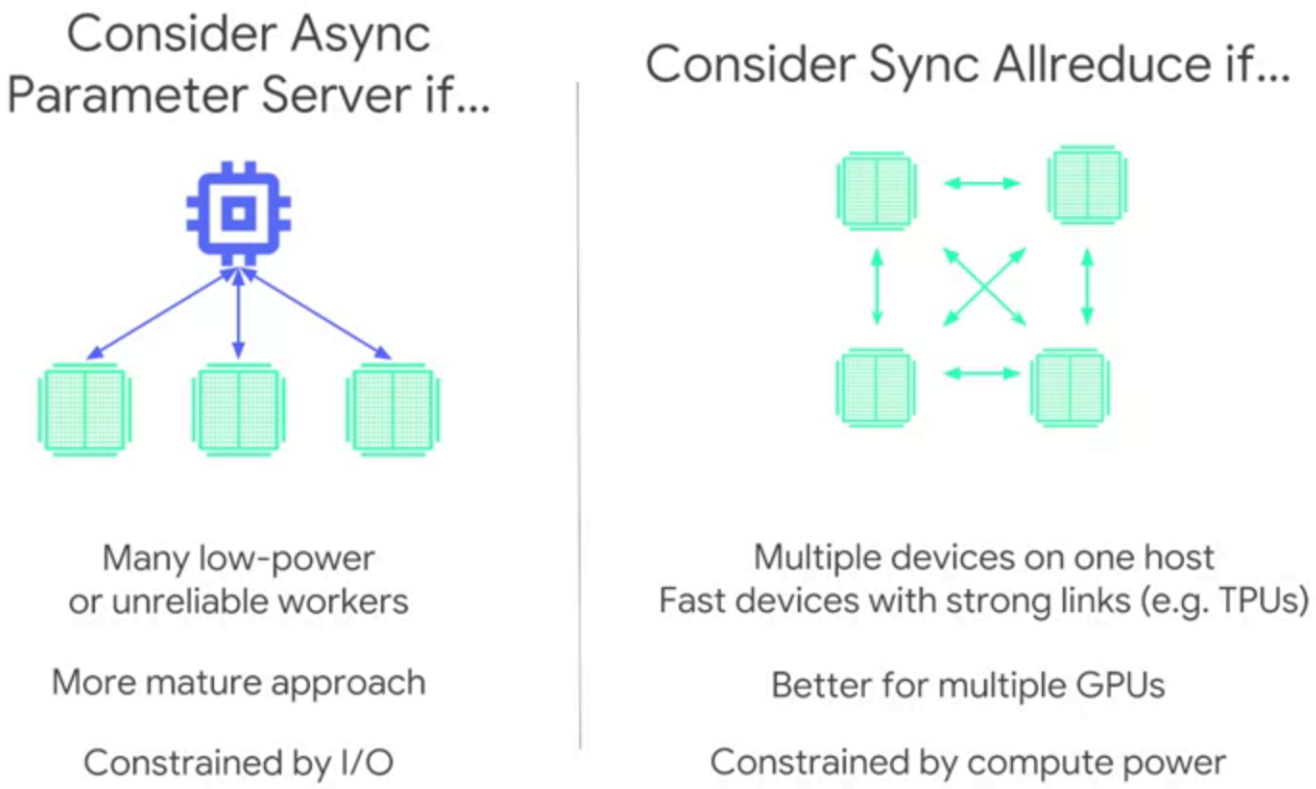
- **Tensorflow Distribution API (mirrored Strategy)**
  - The API is easy to use and fast to train.
  - With the distribution strategy API, you no longer need to place apps or parameters on specific devices.
  - You don't need to worry about structuring your model in a way that gradients or losses are aggregated correctly across devices either.
  - Distribution strategy takes care of all of that for you.

Training with Estimator API



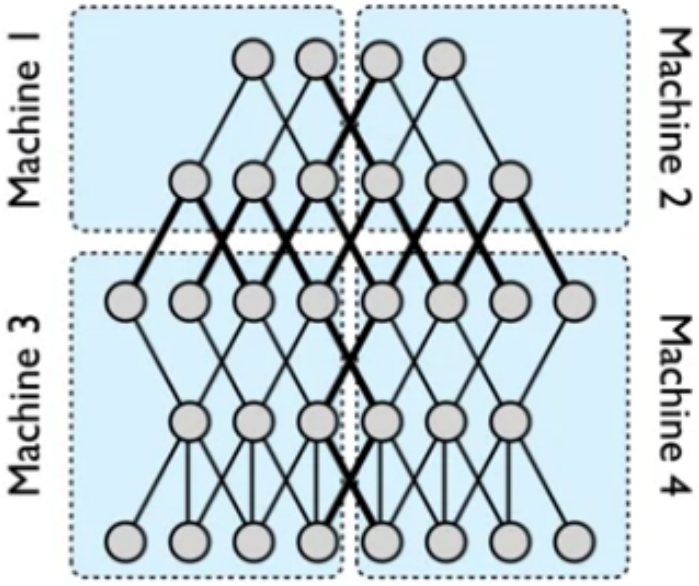
- The tf Estimator API takes 3 arguments as input:
  - **Model function**
    - Defines the model
    - It defines the parameters of the model, the loss and gradient computation, and of course how the parameters are updated.
  - **Model directory**
    - This is the directory where the model state will be persisted.
  - **Run Config**
    - Specifies things like how the model state is checkpointed, how often summaries are read in, and so on.
- Once we create the estimator, we can initiate training by calling the train method on the estimator with an input function that provides the data for training.
- In order to distribute this training on multiple GPUs you simply need to add one line.
  - You create an instance of something called a **mirrored strategy** and pass it to the run config call.
  - ***Mirrored strategy*** is part of the ***distribution strategy API***.
  - Mirrored Strategy implements synchronous all-reduce architecture right out of the box.
    - The model's parameters are mirrored across all devices.
      - ***Hence the name Mirrored Strategy***.
    - Each device computes loss and gradients based on a slice of the input data.
    - Gradients are then aggregated across all devices using an all reduce algorithm that's best for the device topology.

1.5.3.1.3 Parameter server vs All Reduce



- Use parameter server for multiple machines and all reduce for multiple devices on one machine.

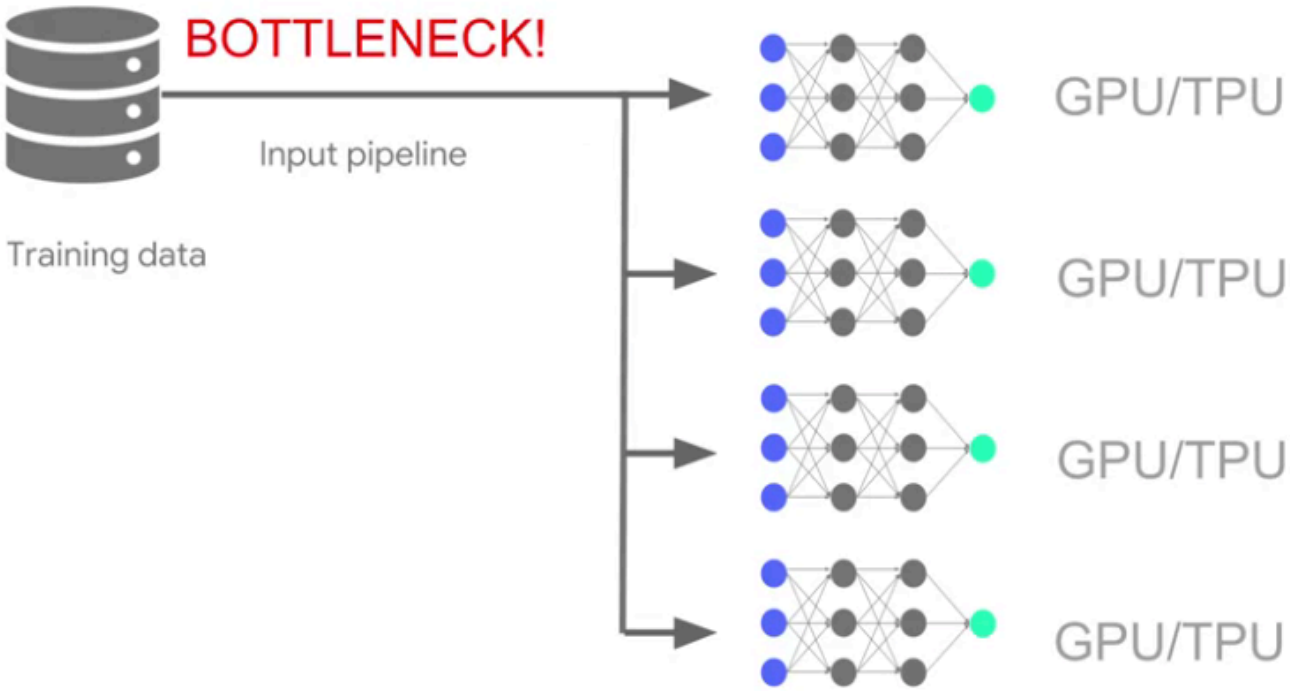
1.5.3.2 Model Parallelism



- Used when your model is so big that it doesn't fit into a single devices' memory.
    - You then have to divide it into smaller parts that can compute over the same training samples on multiple devices.
    - For example, you could put different layers on different devices.

1.5.4 Faster Input Pipelines

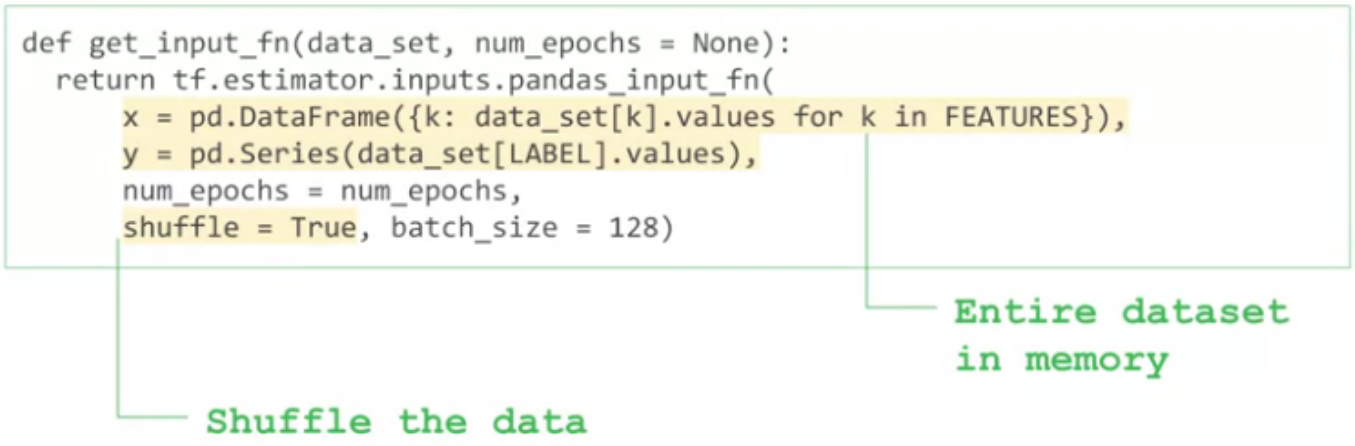
- Should always try to improve the performance of your input output pipeline.
    - This becomes essential if your model is IO bound, and this will be the case if you're using multiple GPUs or if you're using TPUs.
    - When running a model on a single GPU, the input pipeline is able to pre-process and provide input to the model as required. But GPUs and TPUs can process data much faster than CPUs and they reduce the execution time of training one step.
    - Hence, as you increase the number of GPUs, the input pipeline can no longer keep up with the training and often becomes a bottleneck.
    - Specifically, before an iteration finishes, the data for the next iteration is not yet available for processing.



- There are three approaches to reading data into TensorFlow:
    - Directly feed from Python
    - Use native TensorFlow Ops
    - Read transformed TensorFlow records

1.5.4.1 Directly feed from Python

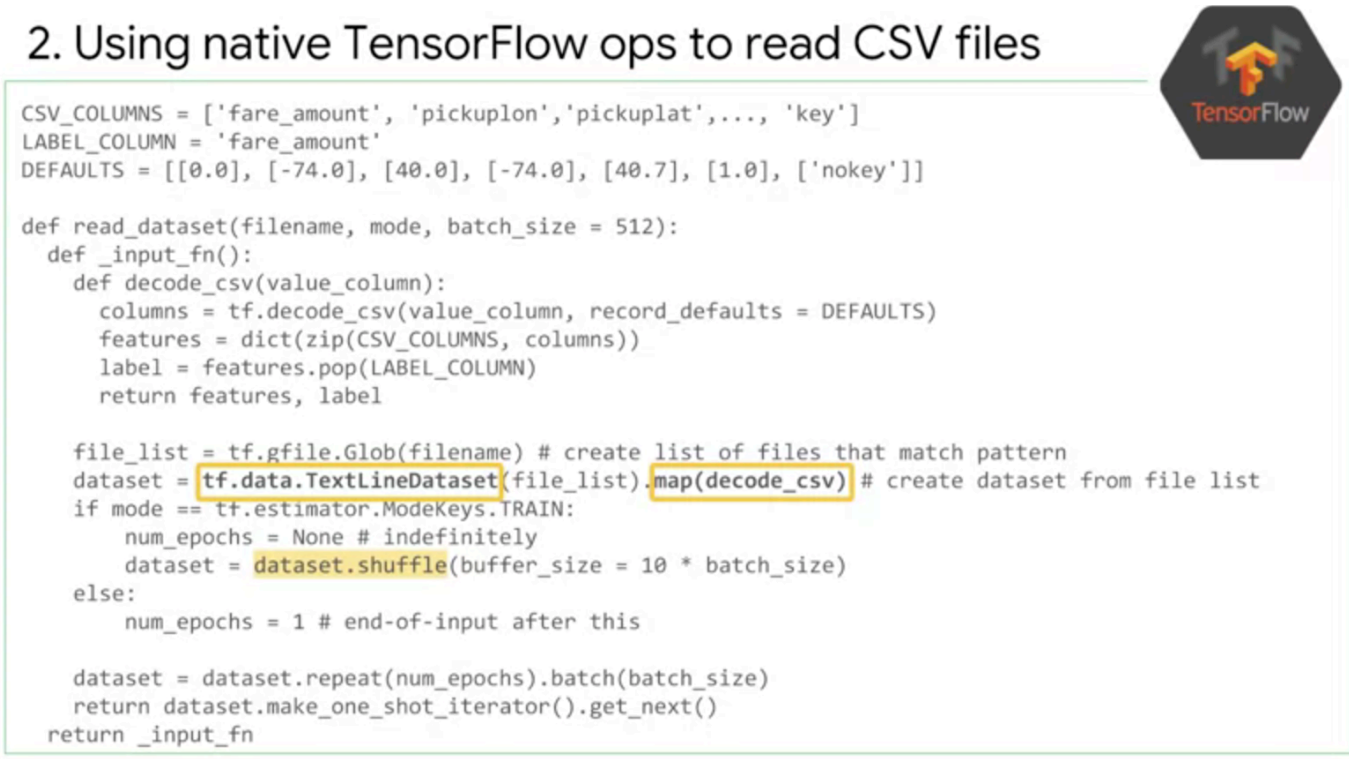
- Simplest but slowest



- pandas_input_fn() is part of TensorFlow core.
    - There is a similar function called numpy_input_fn()
- This is really fast but only because the entire dataset is held in memory.
- It's not very scalable.
- For most realistic problems, keeping all of your data and memory is a sure far a way of taking a problem that is IO bound or CPU bound, and then making it memory bound.

1.5.4.2 Use native TensorFlow Ops

- ***Example reading a CSV file***

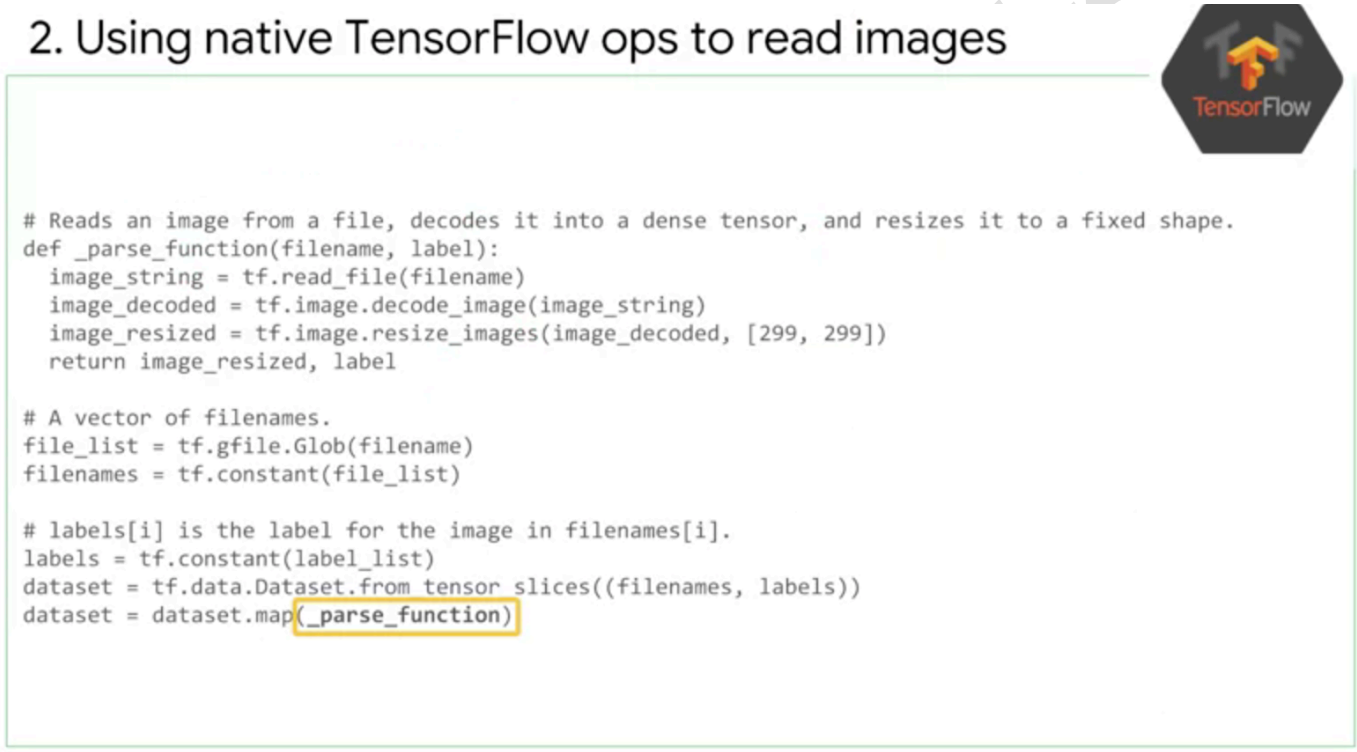


- All of these TensorFlow classes and functions tf.data, tf.datamap,tf.decode_csv are all implemented in C++,so you can get very good performance.
- The key to this is not to bring the data back into Python, that context switch can be expensive.
    - There is some Python code here in terms of the dict and zip calls. Unfortunately, that's necessary because the CSV file is not self-describing. However, this does come at the very end so it's not a back-and-forth context switch.

- ***Why is shuffle needed above?***
    - Shuffling is essential when you do distributed data using data parallelism.
    - In data parallelism, if all of the workers are computing the gradient on the same set of data, there is no need to have all of those workers. Therefore, each of the workers needs to see a different slice of data.
    - Given that we're reading from the same CSV file, how do we try to get each of the workers at different slice the data?

- We do that by reading in 10 times the batch size, shuffling it and then serving it out one batch at a time.
- The chances are that each worker is holding the data in a different order. So, the batch of examples that each worker processes will be different and that's why we shuffle.

- ***Example for images***

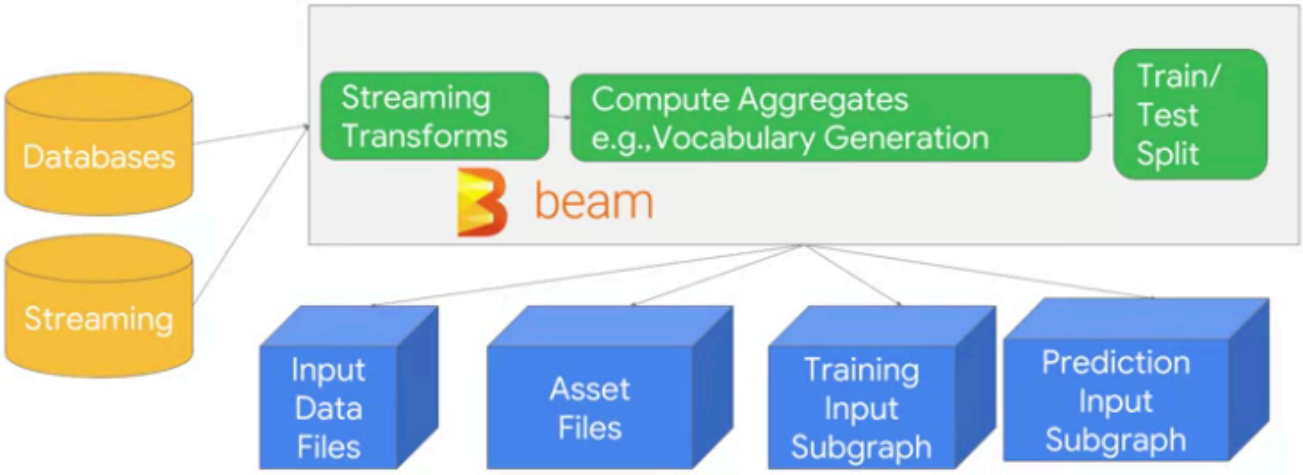


- The parse function uses tf.read_file to read in the entire file content.
- Then, tf.decode_image to parse the image formats and resize it to be 299 by 299 before it returns it.
- The point as before is to use native TensorFlow ops to keep all of the operations in C++ and therefore very efficient.

1.5.4.3 Read transformed TensorFlow records

- Fastest but complex.
- Here, you convert the data into TensorFlow records.
    - You do this in a preprocessing operation.
    - Then in the TensorFlow training program, you can read the data quite rapidly.
    - TFRecords are set for fast efficient batch reads without the overhead of having to parse the data in Python.
- You have to plan for this in your architecture and convert data from its native format to tf records.
- Often you will do this by incorporating apache beam using tf transform or apache spark in your data pipeline.

- **Example using Beam**



    - You can do the preprocessing using Apache Beam, but you could do it in any Python program.
    - If you are just converting images to TFRecords, there is no state to remember, and so plain Apache Beam will just work.
    - However, if you want to remember things from the transformation, such as the vocabulary files used for scaling constants, then you could do the conversion using tf.Transform.

- **Sample code below**:

3. Preprocess data into TFRecord

```
def convert_to_example(csvline, categories):
  filename, label = csvline.encode('ascii', 'ignore').split(',')
  if label in categories:
    coder = ImageCoder()
    image_buffer, height, width = _get_image_data(filename, coder)
    example = _convert_to_example(filename, image_buffer,
                                  categories.index(label), label, height, width)
    yield example.SerializeToString()

LABELS = ['nails', 'screws']
(p
 | beam.FlatMap(lambda line: convert_to_example(line, LABELS))
 | beam.io.tfrecordio.WriteToTFRecord(os.path.join(OUTPUT_DIR, 'train')))
)

# https://github.com/tensorflow/tpu/blob/master/tools/datasets/jpeg_to_tf_record.py
```

- **Example using Spark**


```
import org.apache.spark.sql.DataFrame


df.write.format("tfrecords").option("recordType", "Example").save(path)
```

Warning! Because preprocessing is carried out on DataFrame using Spark, you need to repeat the preprocessing during prediction using Spark Streaming (the code is different).

- For Spark, you can do a transformation on the DataFrame and just write it out.
- Repeating the preprocessing during prediction is on you, Spark and Spark Streaming don't use the same paradigm, unlike Apache Beam.

- **Sample code below**:

3. Read TFRecord produced by tf.transform or Spark

```
from tensorflow_transform.saved import input_fn_maker
def gzip_reader_fn():
  return tf.TFRecordReader(options=tf.python_io.TFRecordOptions(
      compression_type=tf.python_io.TFRecordCompressionType.GZIP))

def get_input_fn(transformed_metadata, transformed_data_paths, batch_size, mode):
  return input_fn_maker.build_training_input_fn(
      metadata=transformed_metadata,
      file_pattern=(
          transformed_data_paths[0] if len(transformed_data_paths) == 1
          else transformed_data_paths),
      training_batch_size=batch_size,
      label_keys=[TARGET_FEATURE_COLUMN],
      reader=gzip_reader_fn,
      key_feature_name=KEY_FEATURE_COLUMN,
      reader_num_threads=4,
      queue_capacity=batch_size * 2,
      randomize_input=(mode != tf.contrib.learn.ModeKeys.EVAL),
      num_epochs=(1 if mode == tf.contrib.learn.ModeKeys.EVAL else None))
```

gzipped files

tf.transform writes out the TFRecords with metadata

Each read is of Batch_SIZE recs

- Once you have the TensorFlow records , regardless of how you created it, the input function in the TensorFlow graph comes from TensorFlow transform.

1.5.4.4 Parallel Pipelines

- A typical input pipeline can be thought of as an **ETL process**
  - i.e a process involving **extract, transform, and load** phases.
  - ***Extract phase***: we read data from persistent storage, either local or remote.
  - ***Transform phase:*** we use CPU cores to parse and perform preprocessing operations on the data such as image decompression, cropping, flipping, shuffling, and batching.
  - ***Load stage***: we load the transform data onto the accelerator devices that will then execute the model.
  - This pattern effectively uses the CPU while reserving the accelerator for training your model.

- Let's look at how this applies to a simple image classification pipeline

A simple input pipeline for an image model

E

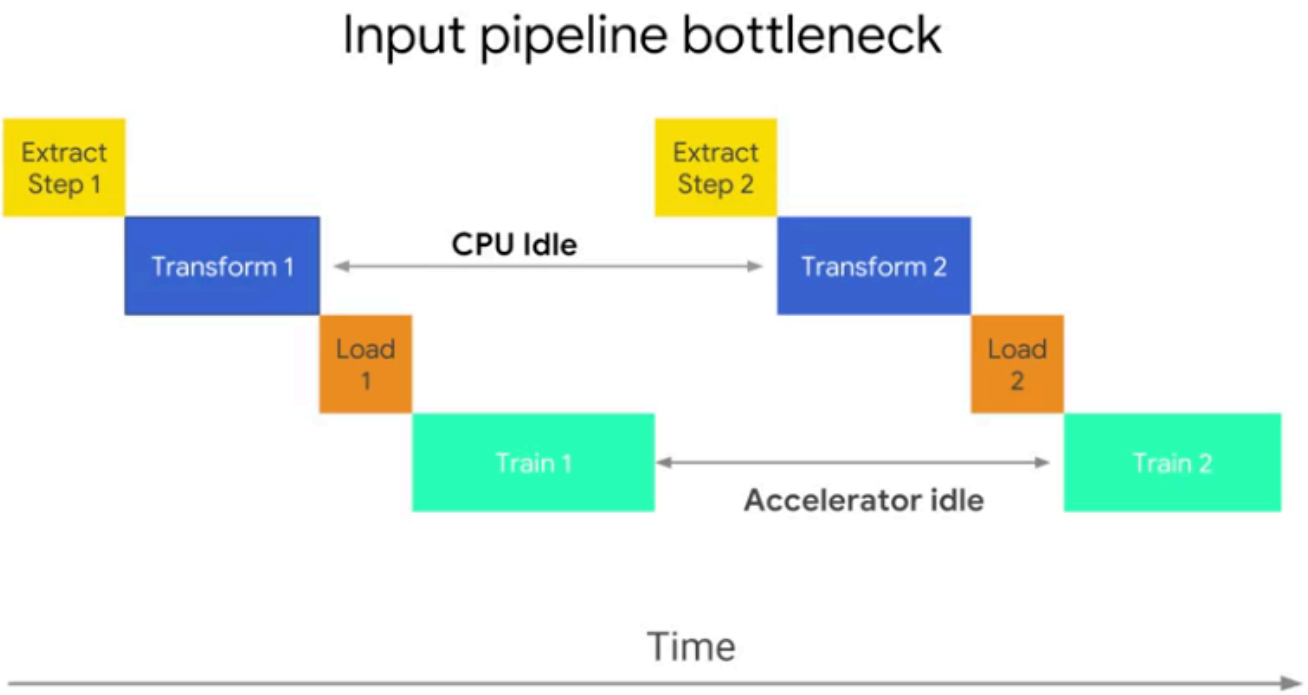
T

L

```
def input_fn(batch_size):
  files = tf.data.Dataset.list_files(file_pattern)
  dataset = tf.data.TFRecordDataset(files)

  dataset = dataset.shuffle(10000)
  dataset = dataset.repeat(NUM_EPOCHS)
  dataset = dataset.map(preproc_fn)
  dataset = dataset.batch(batch_size)

  return dataset
```

- **Extract Phase**
  - We start by getting a list of files from the local or remote data source and extract TF Records from these files.
  - TF.data API is the recommended way of building input pipelines in TensorFlow.
    - It enables you to build complex input pipelines from simple, reusable pieces.
    - It makes it easy to deal with large amounts of data, different data formats, and complicated transformations.
    - Here, we first use a method called list files to grab a bunch of input files containing the images and labels.
    - We will then parse them using a TF record dataset reader.
- **Transform Phase**
  - In the transform, we apply a number of transformations to the input records such as shuffling, mapping, and batching.
- **Load Phase**
  - The load phase then tells TensorFlow how to get the data from the dataset.

- This is what our example input pipeline would look like.



- Each stage of the input ETL extract transform load as well as the training on the accelerator will happen sequentially.
  - While the CPU is preparing the data, the accelerator is sitting idle.
  - Conversely, while the accelerator is training the model, the CPU is sitting idle.
  - The training step time is thus the sum of both the CPU preprocessing time and the accelerator training time.

- **Pipelining**
  - The different stages of the ETL process use different hardware resources.
    - The extract phase uses local storage, the transform phase uses the CPU cores to perform transformations, and the model is trained on the GPU.
  - When we parallelize those stages without resource contention, we are able to overlap preprocessing of the input data on the CPU with model training on the GPU and this is called ***pipelining***.

- Parallelizing and pipelining in various stages of the pipeline can really help reduce training time.

1. Parallelize file reading


```
def input_fn(batch_size):
  files = tf.data.Dataset.list_files(file_pattern)
  dataset = tf.data.TFRecordDataset(files, num_parallel_reads=40)
  dataset = dataset.shuffle(buffer_size=10000)
  dataset = dataset.repeat(NUM_EPOCHS)
  dataset = dataset.map(preproc_fn)
  dataset = dataset.batch(batch_size)
  return dataset
```

Parallelize file reading from Google Cloud Storage

  - You can speed up by reading multiple files in parallel using the num_parallel_reads in the TFRecordDataset constructor.

2. Parallelize map for transformations


```
def input_fn(batch_size):
  files = tf.data.Dataset.list_files(file_pattern)
  dataset = tf.data.TFRecordDataset(files, num_parallel_reads=40)
  dataset = dataset.shuffle(buffer_size=10000)
  dataset = dataset.repeat(NUM_EPOCHS)
  dataset = dataset.map(preproc_fn, num_parallel_calls=40)
  dataset = dataset.batch(batch_size)
  return dataset
```

Parallelize across many CPU cores

- You can parallelize transformations across multiple CPU cores using num_parallel_calls in the TF dataset map constructor.

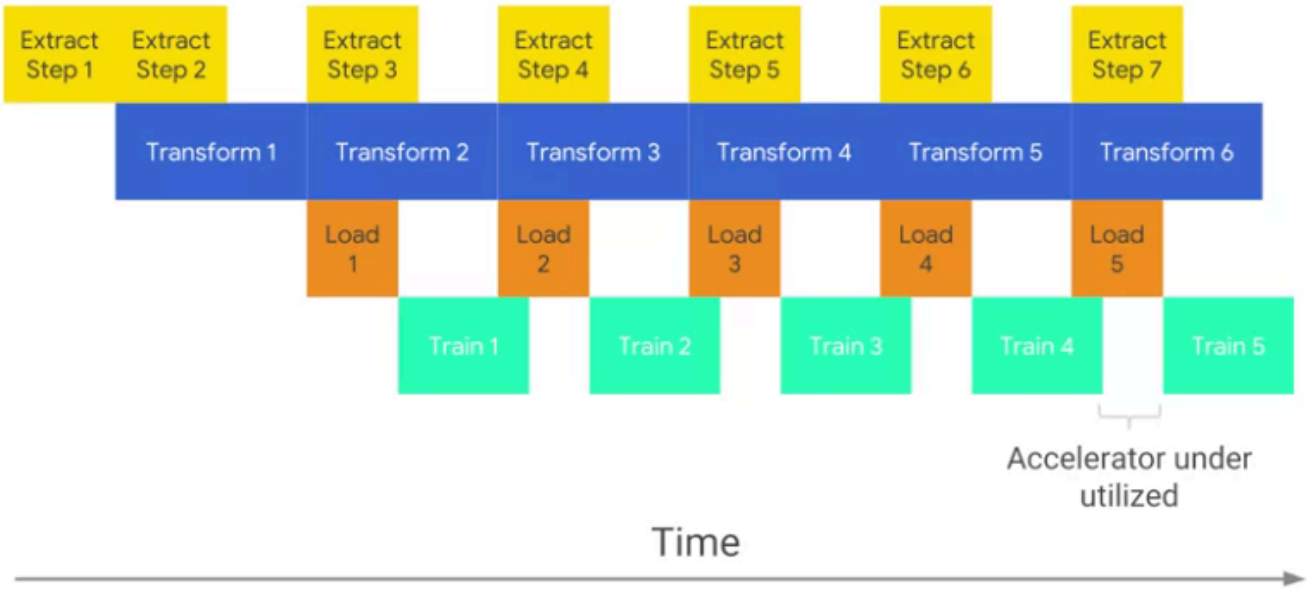
3. Pipelining with prefetching


```
def input_fn(batch_size):
  files = tf.data.Dataset.list_files(file_pattern)
  dataset = tf.data.TFRecordDataset(files, num_parallel_reads=40)
  dataset = dataset.shuffle(buffer_size=10000)
  dataset = dataset.repeat(NUM_EPOCHS)
  dataset = dataset.map(preproc_fn, num_parallel_calls=40)
  dataset = dataset.batch(batch_size)
  dataset = dataset.prefetch(buffer_size=1)
  return dataset
```

Prefetch pipelines everything above with the accelerator training

- Finally, you should use prefetch data at the end of your input transformation.
- The prefetch decouples the time data is produced from the time it is consumed.
- It prefetches the data into a buffer and parallel with the training step.
- This means that we have input data for the next training step before the current one is completed.

- This is what our input pipeline looks like now.



- The CPU and accelerator are not idle for long periods as we saw previously.
- While the accelerator is performing training step n, the CPU is preparing the data for training step n + 1.
  - Neither the CPU nor the accelerator is idle in this case.
- However, you may still find your accelerator could be slightly underused.

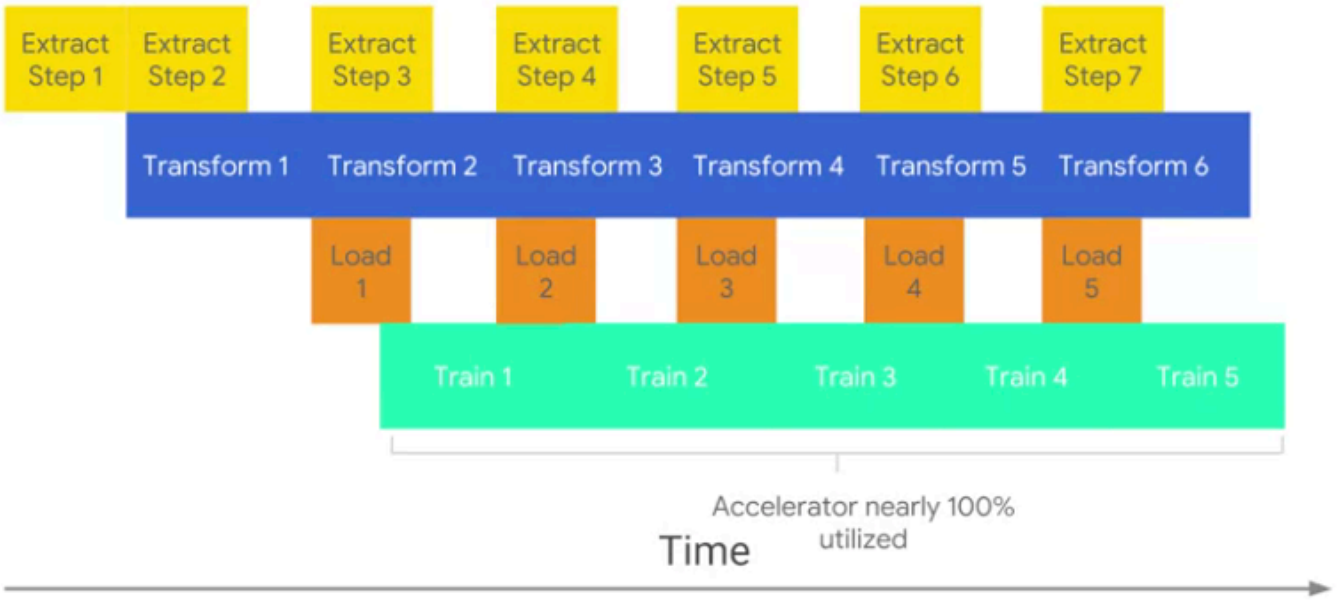
- One common advanced optimization is to switch to the fused versions of some of the transformations we've seen so far.
- ***Fusing*** is to combine operations so that specific optimizations are made possible.

4. Using fused transformation ops

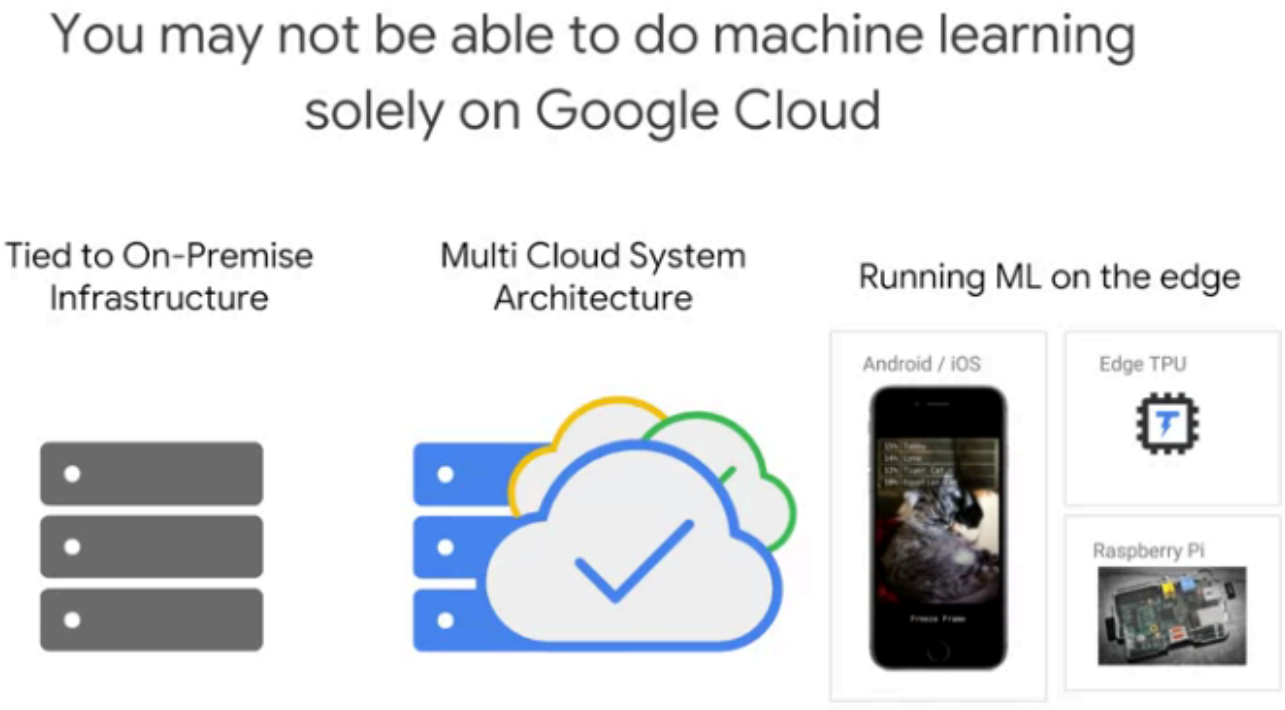

```
def input_fn(batch_size):
  files = tf.data.Dataset.list_files(file_pattern)
  dataset = tf.data.TFRecordDataset(files, num_parallel_reads=40)

  dataset = dataset.apply(
      tf.contrib.data.shuffle_and_repeat(buffer_size=10000, NUM_EPOCHS))

  dataset = dataset.apply(
      tf.contrib.data.map_and_batch(parser_fn, batch_size))

  dataset = dataset.prefetch(buffer_size=1)
  return dataset
```

  - We fuse shuffle and repeat which avoids a performance stall by overlapping the buffering for epoch n+1 while producing elements for epoch n.
  - We also fuse the map and batch which parallelizes both the execution of the map function and the data transfer of each element into the batch tensors.

- By using some of the advanced optimization techniques, we can improve pipeline input efficiency further thereby providing data to the accelerator faster. As you can see in this diagram, the accelerator is now nearly 100% utilized.

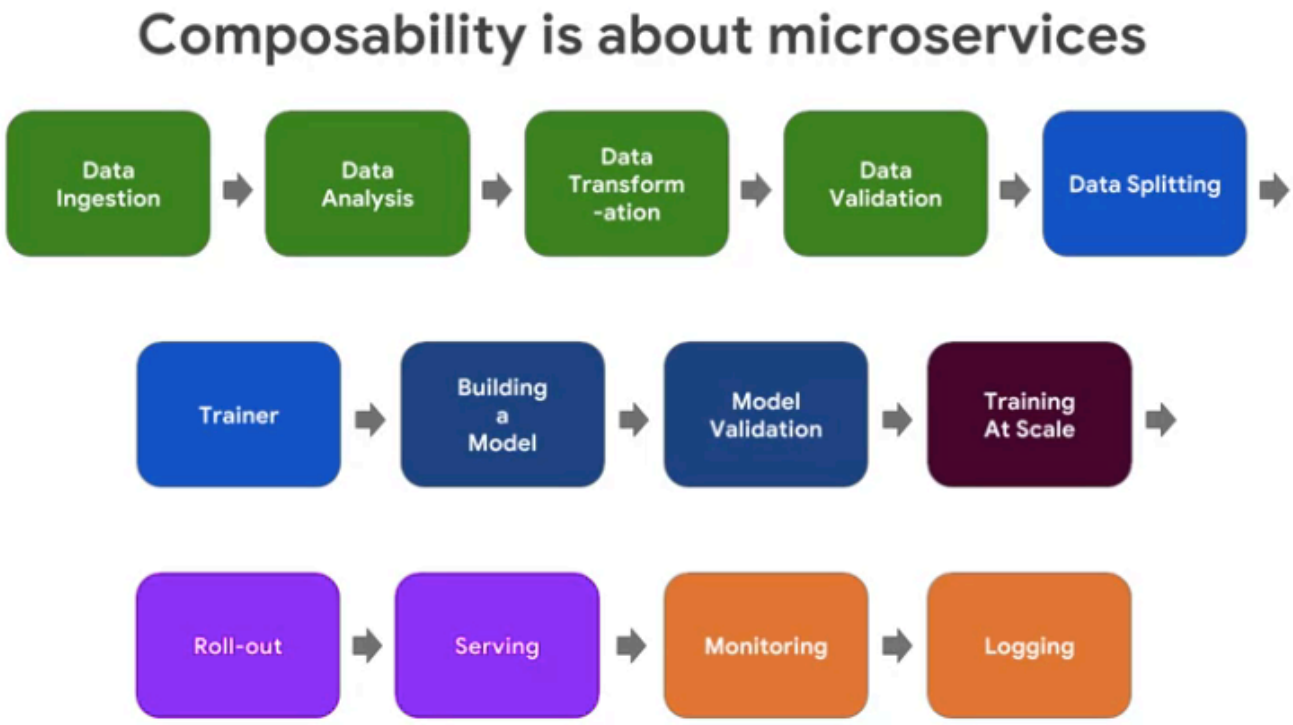


1.6 Hybrid ML Systems



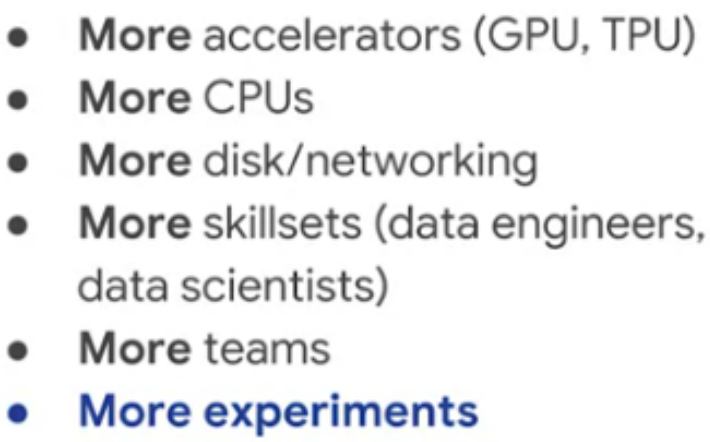
- Kubeflow is an open-source ML stack built on Kubernetes.
  - This enables hybrid ML
  - You can run Kubeflow code anywhere – cloud, on-premises or mobile phone. The code remains the same.

- In order to build hybrid machine learning systems that work well both on-premises and in the cloud, your machine learning framework has to support three things:
    - **Composability**
        - It's about the ability to compose a bunch of microservices together and the option to use what makes sense for your problem



- **Portability**
  - The framework built should be easily portable to different environments

- **Scalability**
  - Here, it includes all of the following:

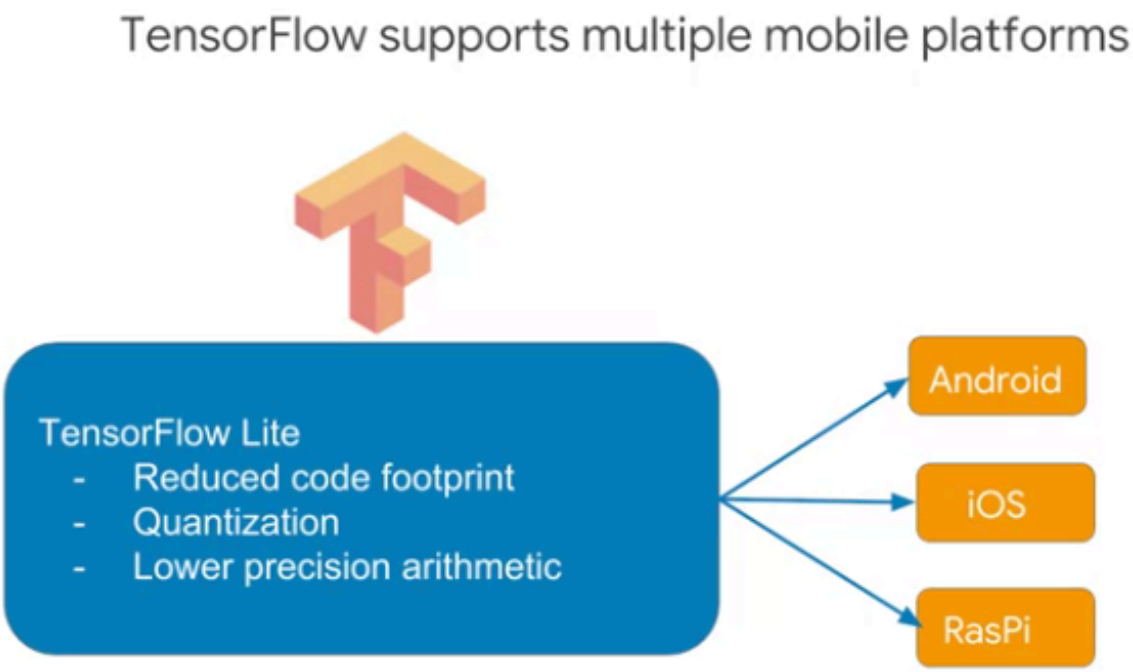


1.6.1 KubeFlow

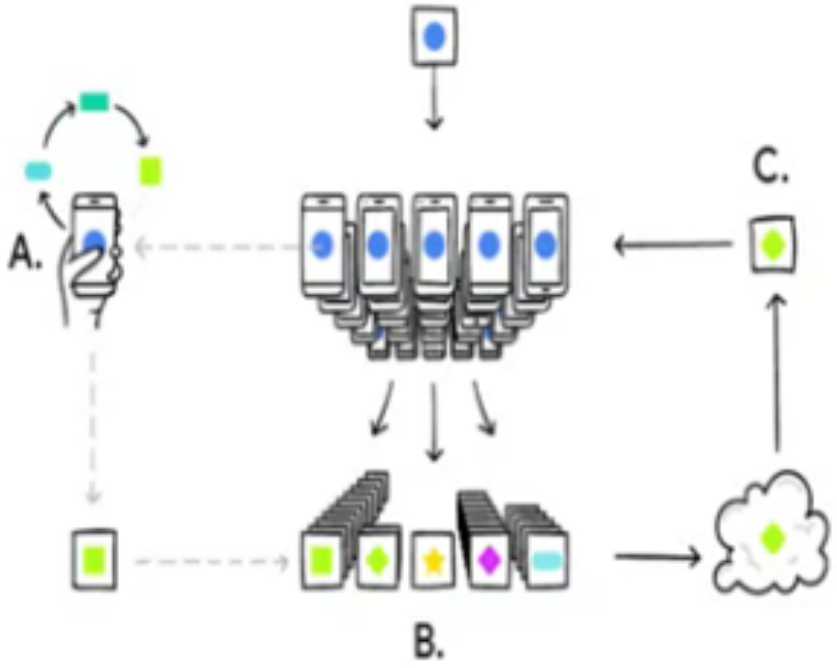
- It is designed to make it easy for everyone to Develop, Deploy and Manage Portable, Distributed ML on Kubernetes.
- KubeFlow benefits:
    - Portability: Works on any Kubernetes cluster, whether it lives on Google Cloud Platform (GCP), on-premise, or across providers.
    - Composability and reproducibility: Enhanced with service workers to work offline or on low-quality networks
    - Scalability: Can utilize fluctuating resources and is only constrained by the number of resources allocated to the Kubernetes cluster.
    - Visualization and collaboration: Pre-installed packages that can be easily deployed.

1.6.2 Embedded ML

- Build and train on the cloud and deploy on mobile devices.
- Tensorflow Lite useful for this.



1.6.3 Federated ML



- The idea is to continuously train the model on the device and then combine the model updates from a federation of user devices to update the overall model
- The goal is for each user to get their customized experience because there's model training happening on the device but still retain privacy because it's the overall model update that goes back to the cloud.

1.6.4 Optimizing for mobile (Tensorflow Lite)

- Some models can be too large to use directly on a mobile device.
- Below are some ways to reduce model size using **Tensorflow Lite**
    - Freeze the graph
    - Transform the graph
    - Quantize weights and calculations

1.6.4.1 Freeze the graph

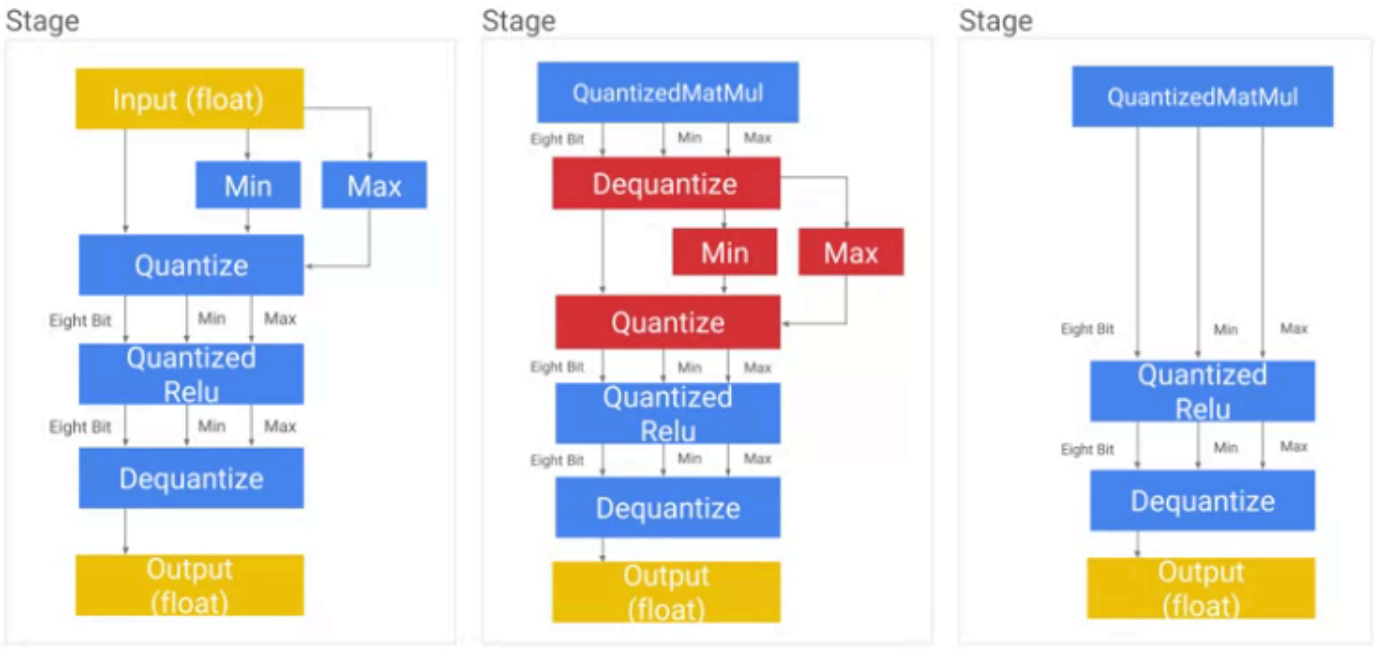
- Freezing a graph is a low time optimization.
- It converts all variable nodes into constant nodes.
- In TensorFlow, variable nodes are stored in different files, whereas constant nodes are embedded in the graph itself in the same file. So, by converting the variable nodes into constant nodes, you get a slight performance win on mobile and it's easier to handle too.
- You don't want to always freeze a graph because:
    - you cannot do continuous training,
    - you cannot do federated learning
    - because they're no longer variables to train, just constants.

1.6.4.2 Transform the graph

- The graph transform tool is part of the TensorFlow Lite distribution and the tool supports various optimization tasks. It supports some of the following transformations.
- **strip_unused_nodes()**
    - You can strip nodes that are not used during inference (but were used during the learning phase).
    - Nodes like gradient computation, batch norm, et cetera.
- **remove_nodes()**
    - Debug nodes can also be removed.
- **fold_batch_norms()**
    - Converts convolution 2D or matrix multiplication operations, that are followed by column-wise multiplications into an equivalent op where the column-wise multiplication is baked into the convolution weights.
    - So, instead of two Ops you have only one Op. This saves some computation during inference.
- **quantize_weights(), quantize_nodes()**
    - The weights themselves can be quantized to make the model more compressed.
    - If you quantize the weights you're reducing the accuracy.
    - So, you're trading off accuracy for model size.

1.6.4.3 Quantization

- When modern neural networks were first developed, accuracy and speed were the prime concerns and as a result neural networks focused on 32-bit floating point arithmetic.
- To combat inefficiencies during inference there are different techniques for storing numbers and performing calculations called ***quantization***.
- Quantization takes a floating-point value and compresses it to an eight-bit integer.
    - It reduces the size of the files
    - It reduces computational resources that you need to handle the data



- In the first stage, we see a typical ReLU, a rectified linear unit operation, with the internal conversion from float to eight-bit values.
    - The min and max values are from the input flow tensor.
    - Once the ReLU operation is performed, the values are dequantized and the output becomes floats.

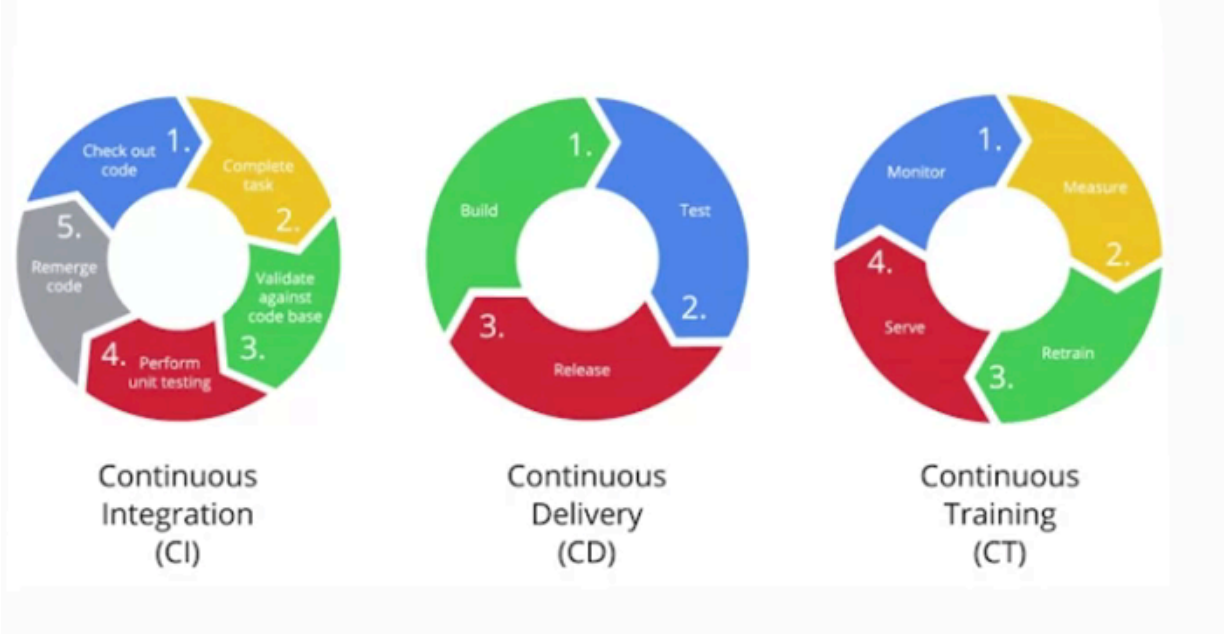
- Next, we remove the unnecessary convergence to and from the float.
    - This stage identifies any patterns in the conversions that are performed in stage number one and removes those redundancies.

- The final stage shows a graph where all the tensor calculations are done in eight bits and there are no conversions that are needed to floating point.
- With these optimizations, with all these three stages, the optimized graph of Inception v3 now becomes just 23MB which is about 75 percent smaller than the original size of 91MB.

2 Machine Learning OPS

2.1 Overview

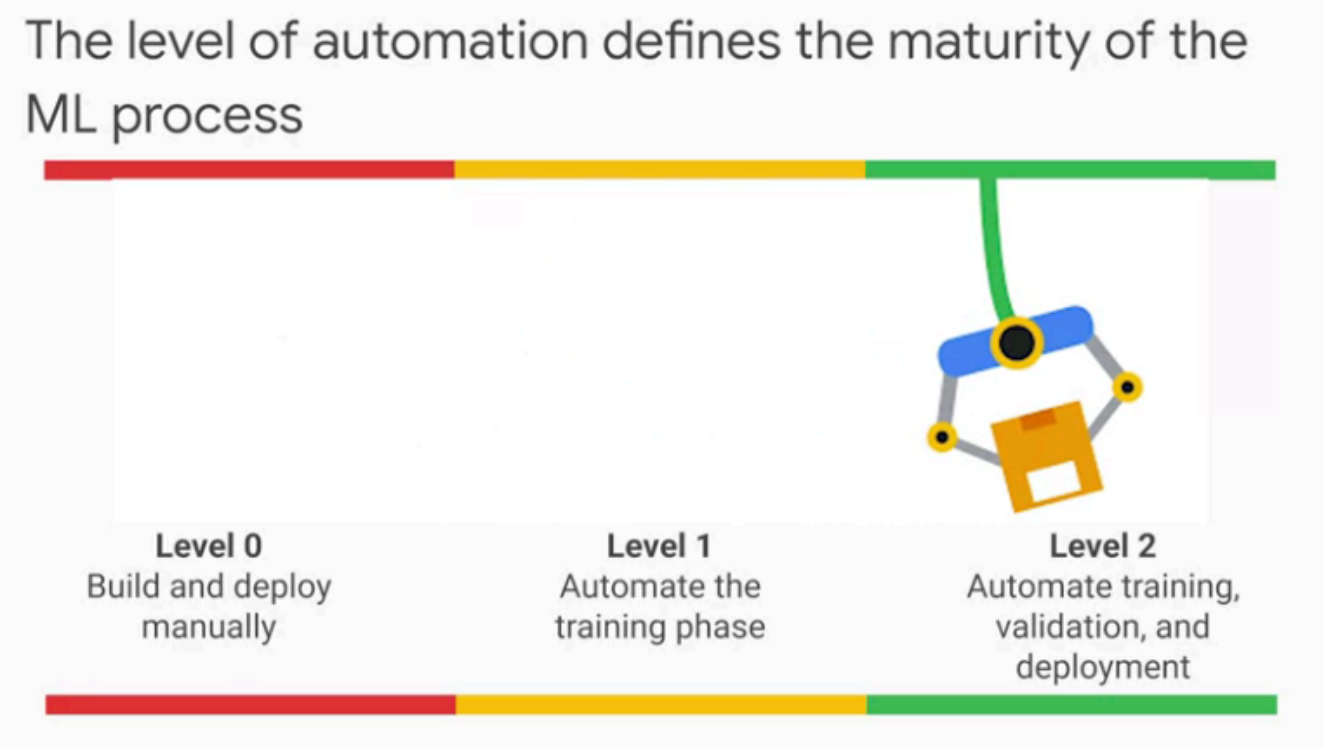
- ML Ops is a lifecycle management discipline for machine learning.



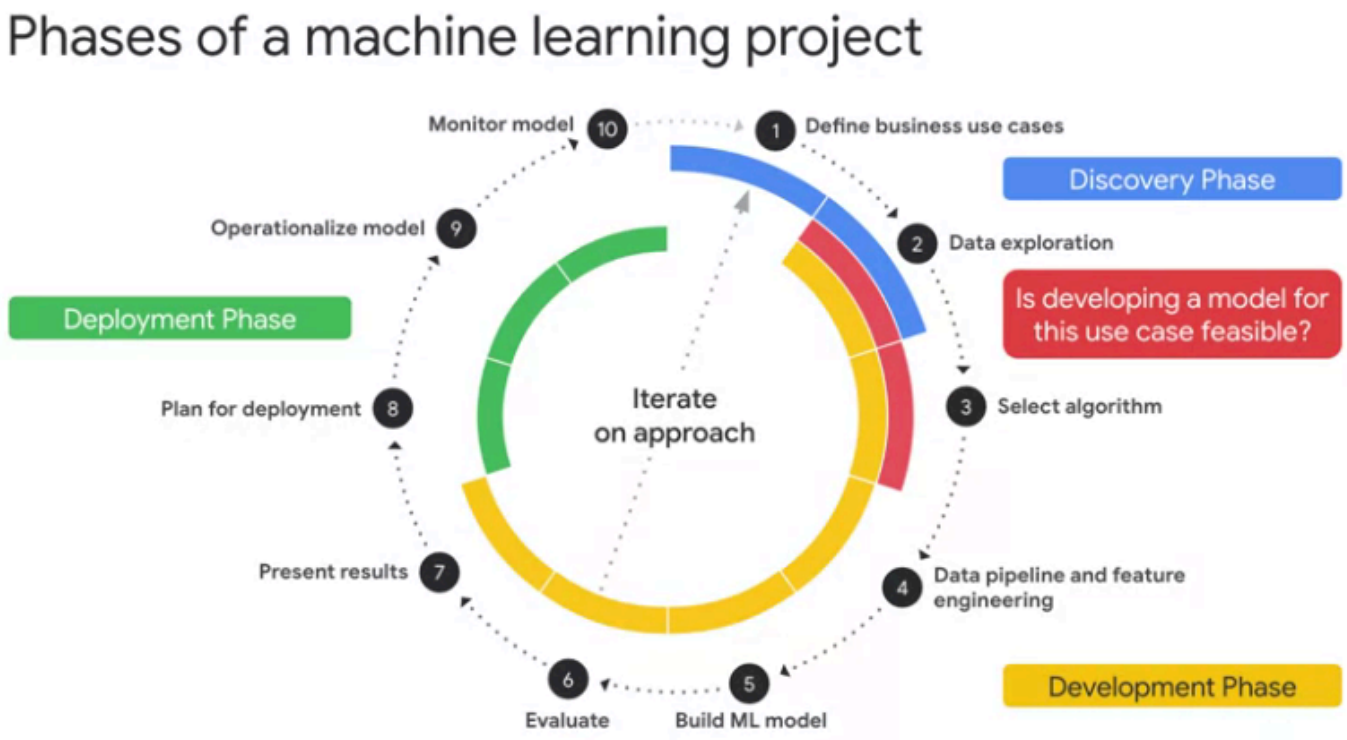
  - **Continuous Training** (CT)
    - Predictive power of models wane over time.
    - We need to continuously monitor, measure, retrain and serve the model.

	DevOps	MLOps
1	Test and validate code and components.	Also test and validate data, data schemas, and models.
2	Focus on a single software package or service.	Also consider the whole system and the ML training pipeline.
3	Deploy code and move to the next task.	Constantly monitor, retrain, and serve the model.

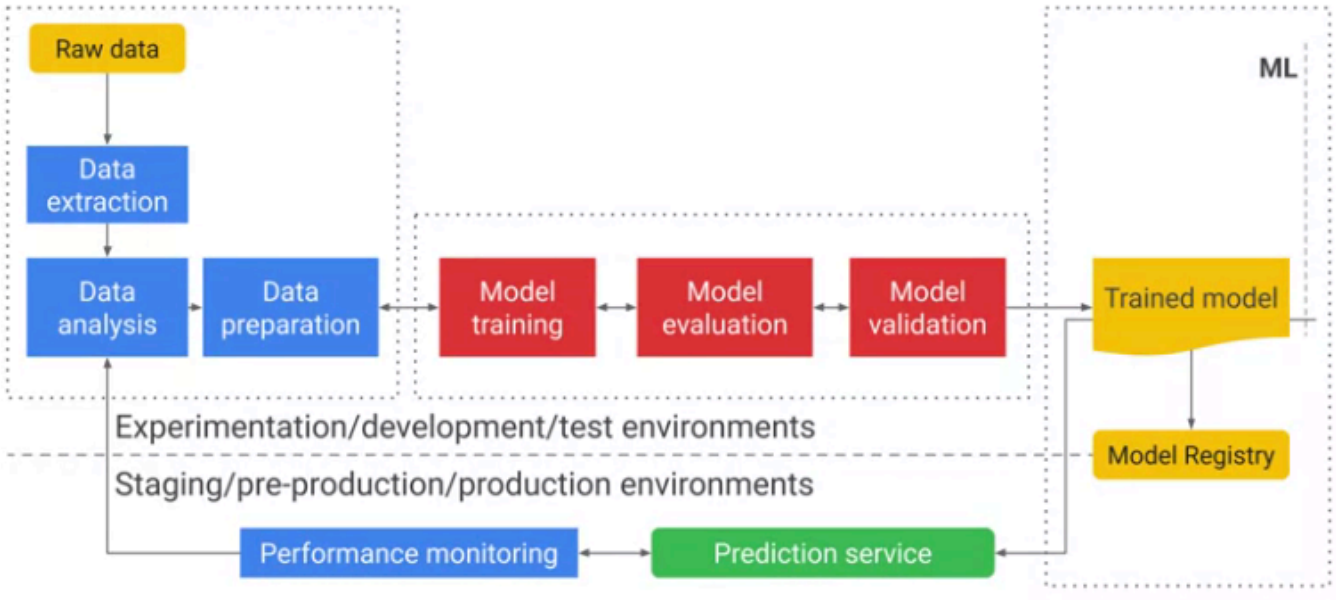
- Challenges with ML systems



2.1.1 Machine Learning Lifecycle



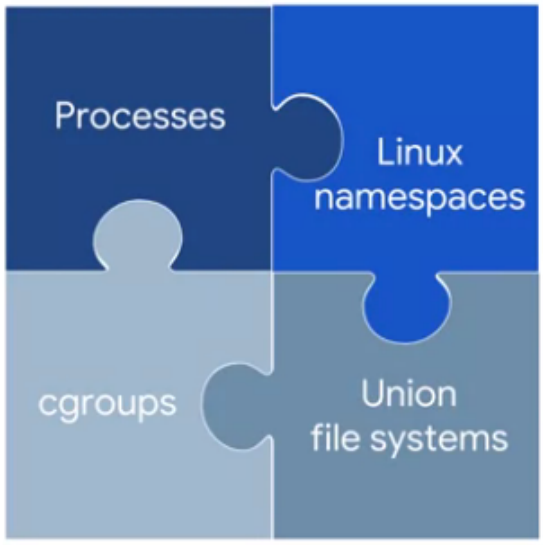
- There are well defined steps for the development and deployment phases



- For example:
    - **Data Exploration**: Includes data extraction, data analysis, and data preparation
    - **Model Building**: Includes training, evaluation, and validation
    - **Deployment**: Includes hosting the model and serving
    - **Monitoring:** Allows for continuous evaluation and training based on the performance results at a given point.

2.1.2 Containers

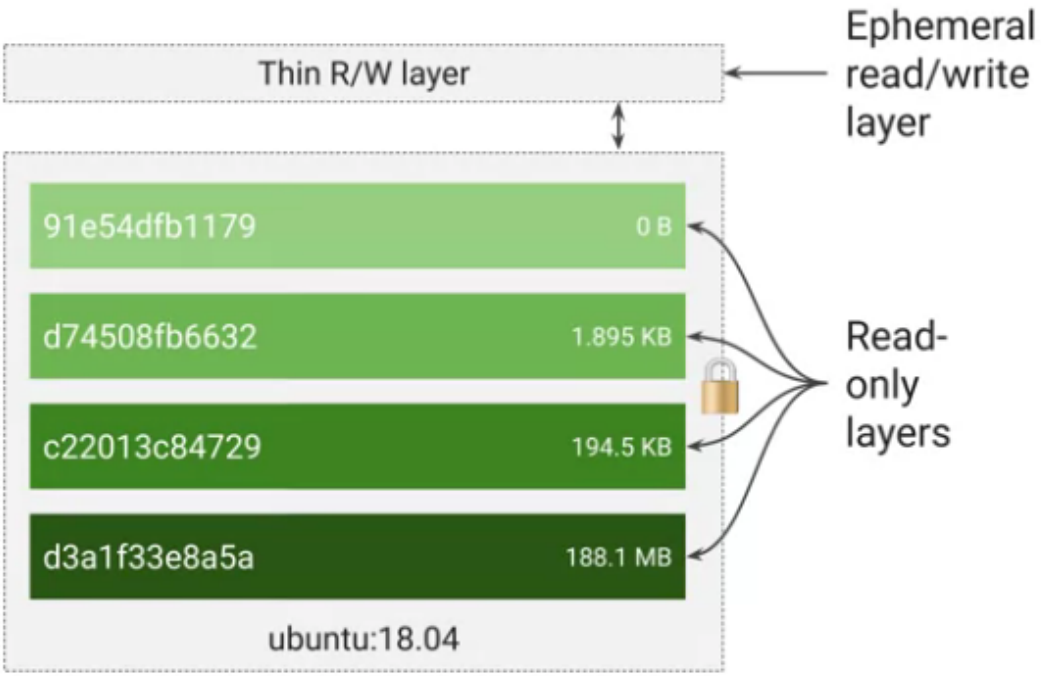
- An application and its dependencies are called an **image**.
- A **container** is simply a running instance of an image.
- **Docker** is an open-source technology that allows you to create and run applications in containers but it doesn't offer a way to orchestrate those applications at scale like Kubernetes does.
- Containers are a composition of several technologies.



    - Containers use **Linux process**. Each Linux process has its own virtual memory address phase, separate from all others. Linux processes are rapidly created and destroyed.
    - Containers use **Linux namespaces** to control what an application can see process ID numbers, directory trees, IP addresses, and more. By the way, Linux namespaces are not the same thing as Kubernetes Namespaces

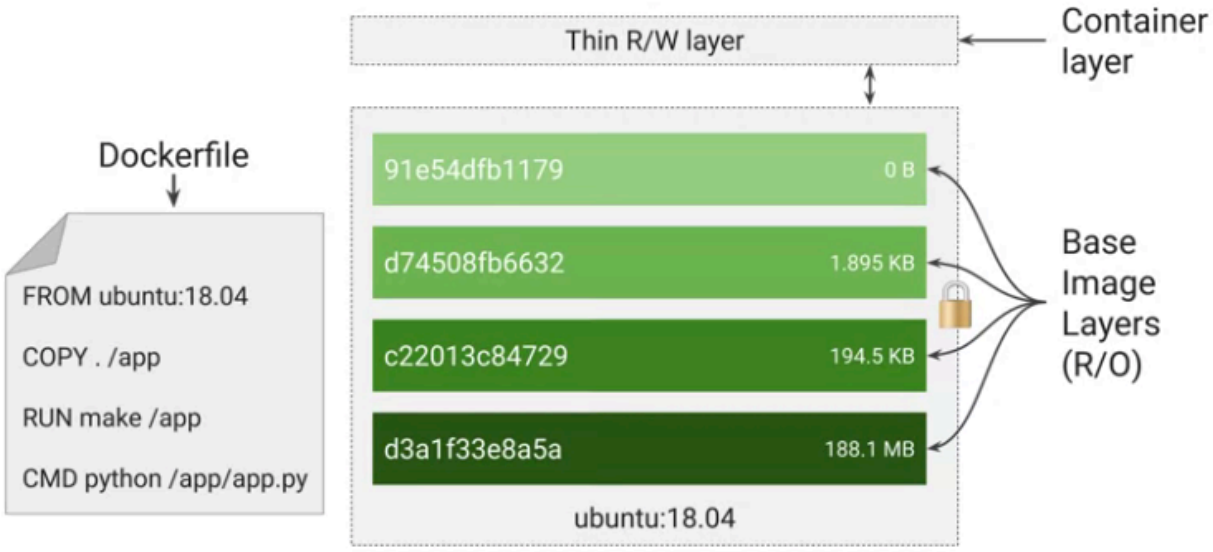
- Containers use **Linux cgroups** to control what an application can use,its maximum consumption of CPU time, memory, IO bandwidth, other resources.
- Containers use **Union File Systems** to efficiently encapsulate applications and their dependencies into a set of clean minimal layers.

- A container image is structured in layers.

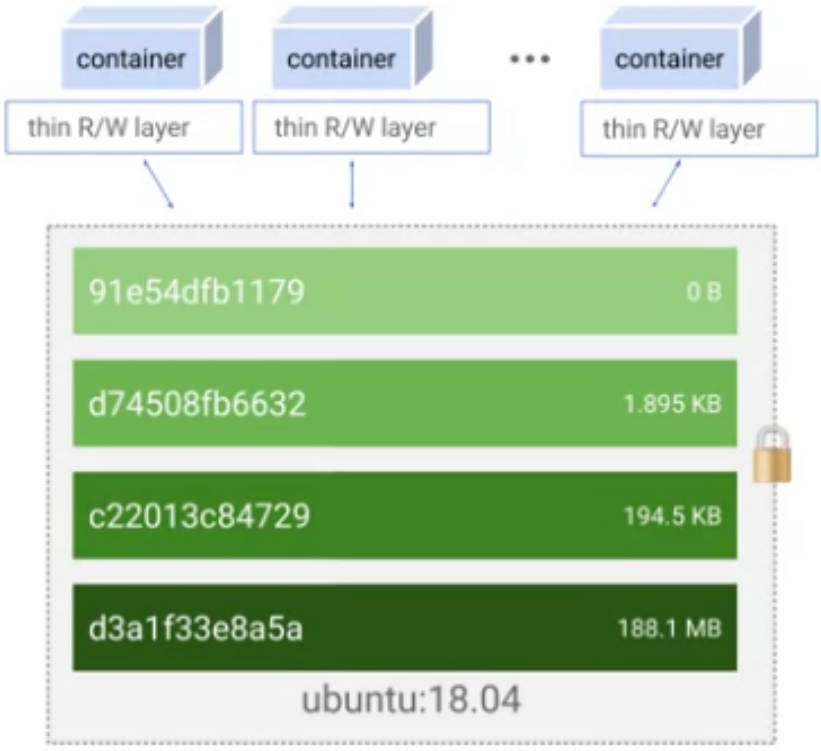


- The tool you use to build the image reads instructions from a file called ***The Container manifest***.
  - In the case of a Docker formatted Container Image the file is called a ***Docker file***.
  - Each instruction in the Docker file specifies a layer inside the container image.
  - Each layer is ***Read only***.
  - When a Container runs from this image it will also have a ***writable ephemeral top-most layer***.

- A simple Docker file will contain four commands each of which creates a layer.



  - The **From** statement starts out by creating a base layer pulled from a public repository.
  - The **Copy** command adds a new layer containing some files copied in from your build tools current directory.
  - The **Run** command builds your application using the make command and puts the result of the build into a third layer.
  - Finally, the **CMD** command specifies what command to run within the container when it's launched.
- When you write a Docker file, you should organize the layers least likely to change through to the layers that are most likely to change.
- When you launch a new container from an image, the Container Runtime adds a new writable layer on the top of the underlying layers. This layer is often called the **Container layer**.

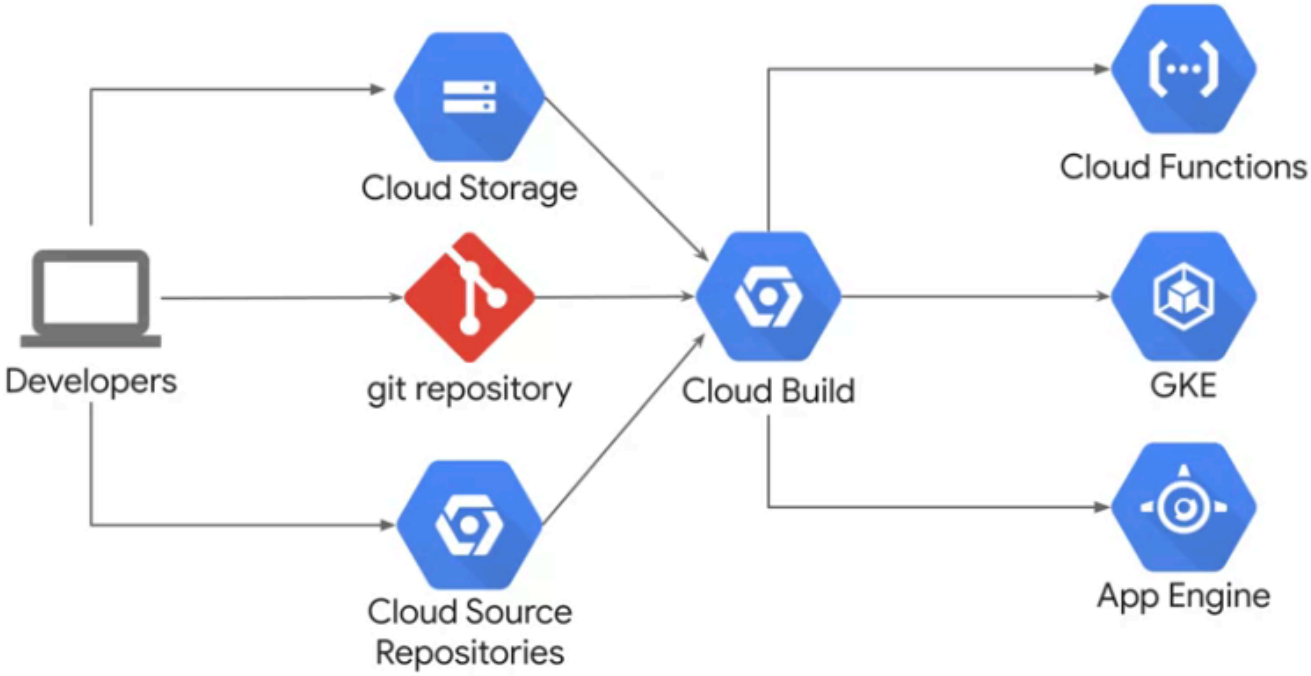


  - All changes made to the running container such as writing new files, modifying existing files and deleting files are written to this thin writable Container layer.
  - This is ephemeral i.e. when the container is deleted the contents of this writeable layer are lost forever.
  - The underlying Container Image itself remains unchanged.
  - Whenever you want to store data permanently, you must do so somewhere other than a running container image.
  - Also, each Container has its own writable Container layer an all changes are stored in this layer
  - Multiple Containers can share access to the same underlying image and yet have their own data state.

- Each layer is only a set of differences from the layer before it.
  - So, you get smaller images.
  - For example, your base application image, maybe 200 megabytes but the difference from next point release might only be 200 kilobytes.
  - When you build a container instead of copying the whole image it creates a layer with just the differences.

- It's very common to use publicly available open-source Container images as a base for your own images or for unmodified use.
- Google maintains a Container Registry ***gcr.io***.
  - This Registry contains many public, open source images and Google Cloud customers also use it to store their own private images in a way that integrates well with Cloud IAM.

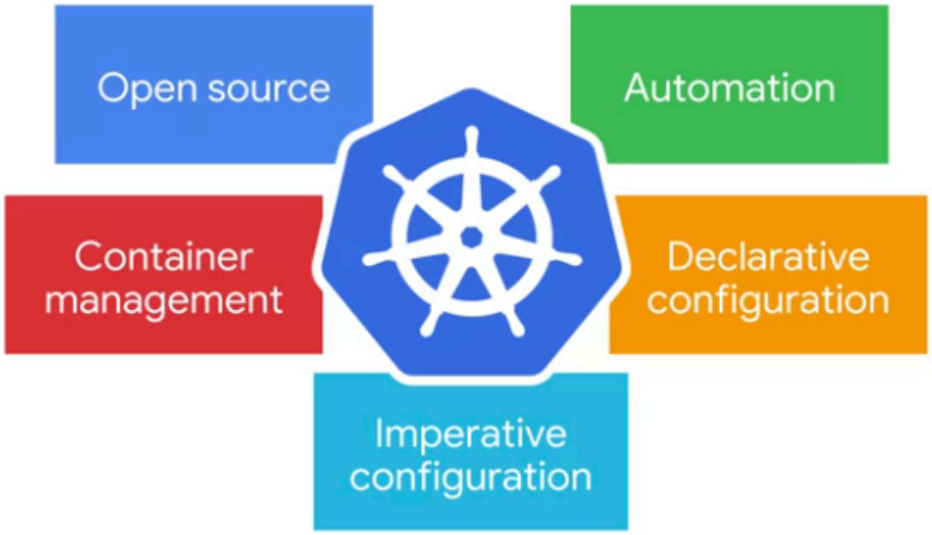
- **Google Cloud Build**



- One downside building Containers with a Docker command is that you must trust the computer that you do your builds on.
- Google provides a managed service for building Containers that's also integrated with Cloud IAM. This service is called **Cloud Build**.
  - Cloud build can retrieve the source code from a variety of different storage locations.
  - To generate a build in Cloud Build you define as series of steps. Each build step in Cloud build runs in a Docker container.
  - Then Cloud build can deliver your newly built images to various execution environments, not only GKE, but also App Engine and Cloud Functions.

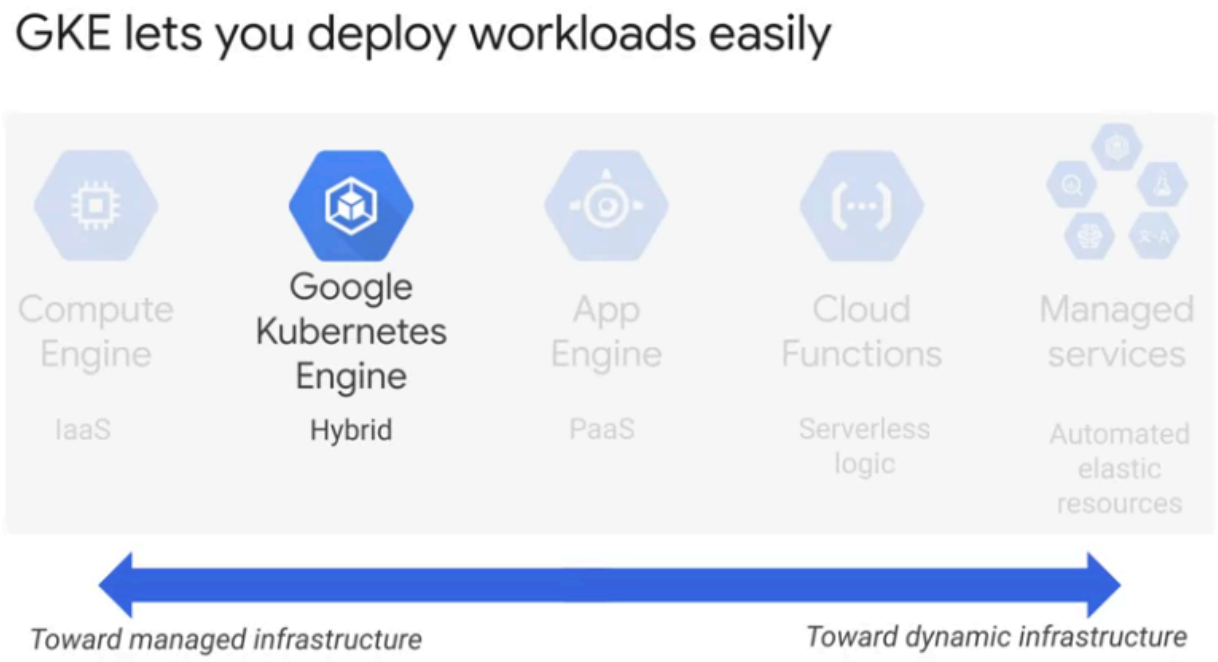
2.1.3 Kubernetes Overview

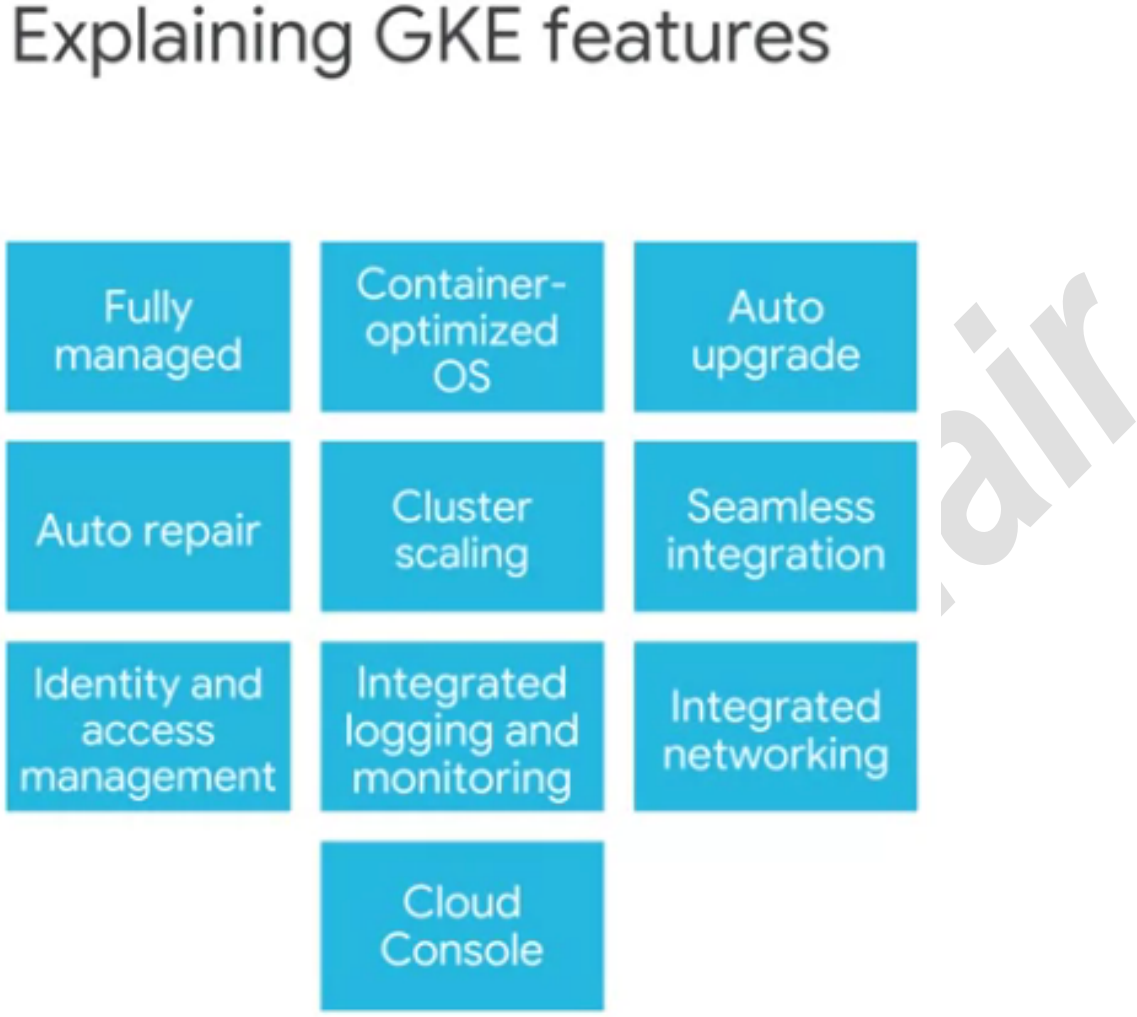
- Kubernetes is a popular container management and orchestration solution



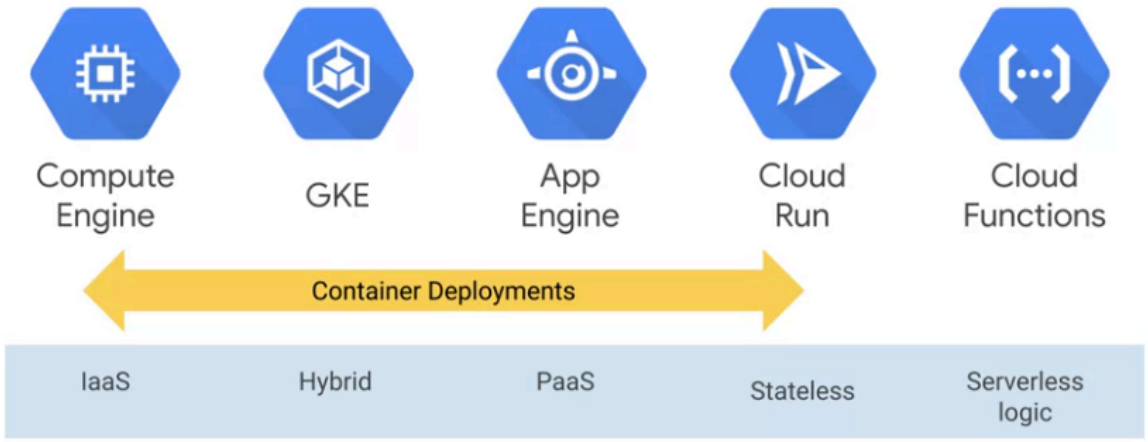
  - It is opensource
  - It automates management tasks (deployment, logging, monitoring etc.)
  - **Declarative configurations**
    - Kubernetes supports declarative configurations
    - When you administer your infrastructure declaratively, you describe the desired state you want to achieve instead of issuing a series of commands to achieve that desired state.
    - Kubernetes job is to make the deployed system conform to your desired state and then keep it there in spite of failures.
    - Because the system's desired state is always documented, it reduces the risk of error.
  - **Imperative configuration**
    - Kubernetes also allows imperative configuration.
    - Here, you issue commands to change the system state.
    - Experienced Kubernetes administrators use imperative configuration only for quick temporary fixes and as a tool in building a declarative configuration.

2.1.4 Google Kubernetes Engine (GKE)





2.1.5 Google Cloud Compute Options



Compute Engine

- Fully customizable virtual machines
- Persistent disks and optional local SSDs
- Global load balancing and autoscaling
- Per-second billing

Compute Engine use cases

1. Complete control over the OS and virtual hardware
2. Well suited for lift-and-shift migrations to the cloud
3. Most flexible compute solution, often used when a managed solution is too restrictive

App Engine

- Provides a fully managed, code-first platform.
- Streamlines application deployment and scalability.
- Provides support for popular programming languages and application runtimes.
- Supports integrated monitoring, logging, and diagnostics.
- Simplifies version control, canary testing, and rollbacks.

App Engine use cases

1. Websites
2. Mobile app and gaming backends
3. RESTful APIs

Google Kubernetes Engine

- Fully managed Kubernetes platform.
- Supports cluster scaling, persistent disks, automated upgrades, and auto node repairs.
- Built-in integration with Google Cloud services.
- Portability across multiple environments
  - Hybrid computing
  - Multi-cloud computing

GKE use cases

1. Containerized applications
2. Cloud-native distributed systems
3. Hybrid applications

Cloud Run

- Enables stateless containers.
- Abstracts away infrastructure management.
- Automatically scales up and down.
- Open API and runtime environment.

Cloud Run use cases

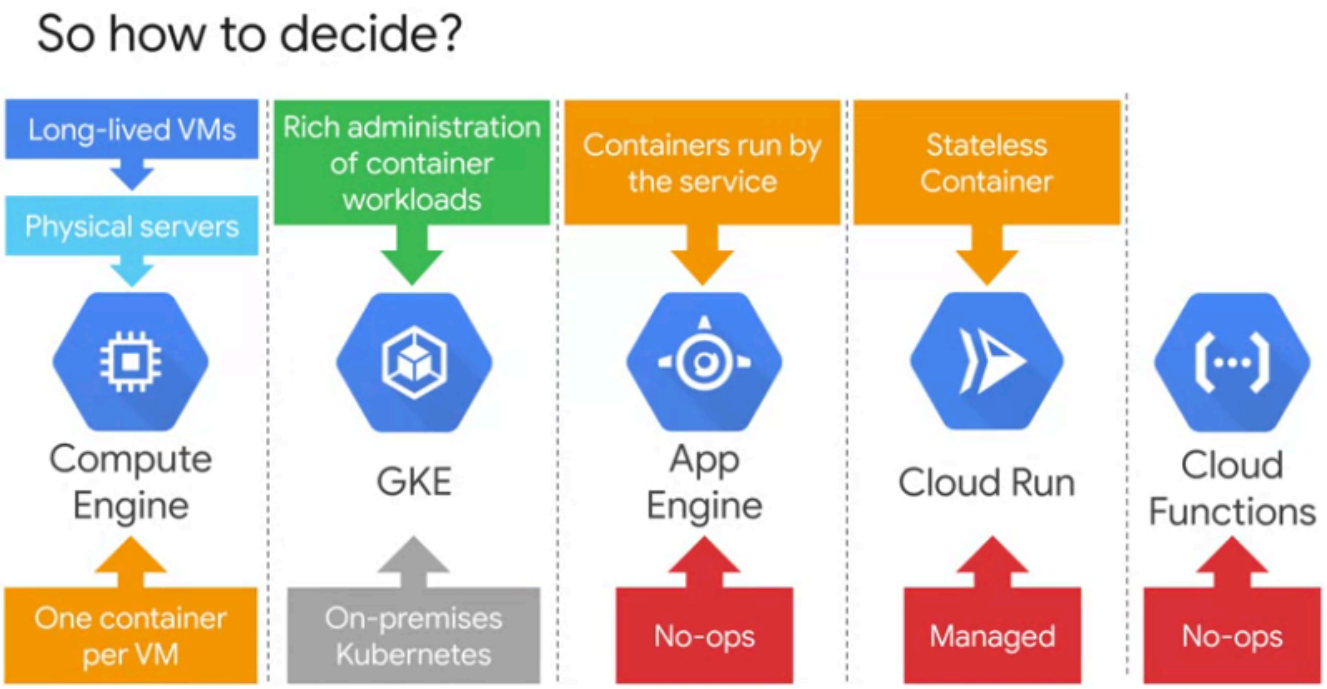
1. Deploy stateless containers that listen for requests or events.
2. Build applications in any language using any frameworks and tools.

Cloud Functions

- Event-driven, serverless compute service.
- Automatic scaling with highly available and fault-tolerant design.
- Charges apply only when your code runs.
- Triggered based on events in Google Cloud services, HTTP endpoints, and Firebase.

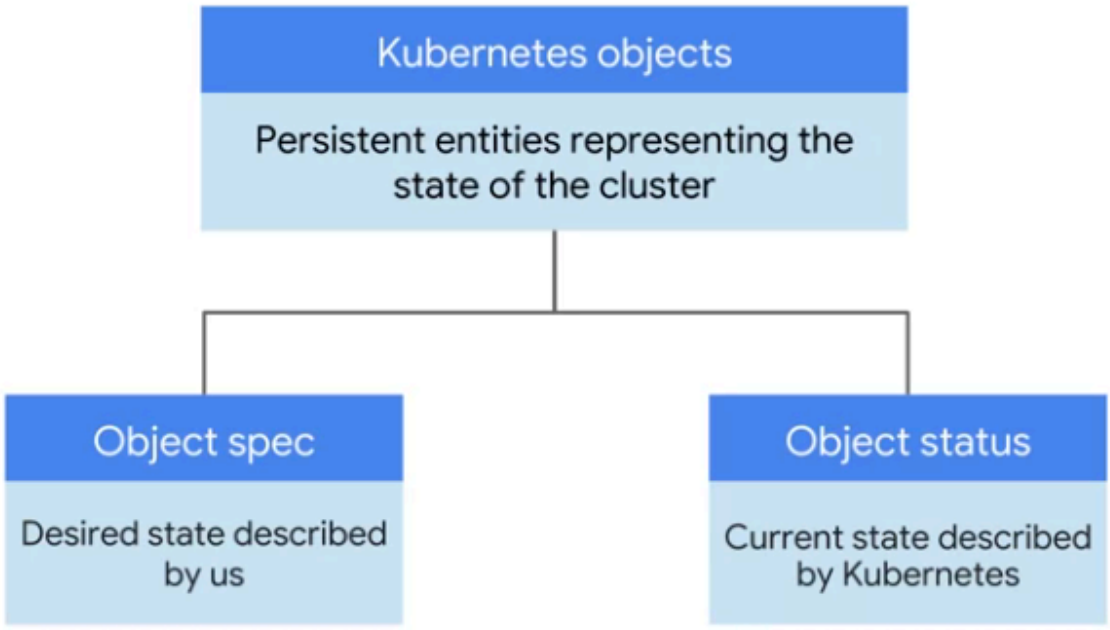
Cloud Functions use cases

1. Supporting microservice architecture
2. Serverless application backends
   - Mobile and IoT backends
   - Integrate with third-party services and APIs
3. Intelligent applications
   - Virtual assistant and chat bots
   - Video and image analysis

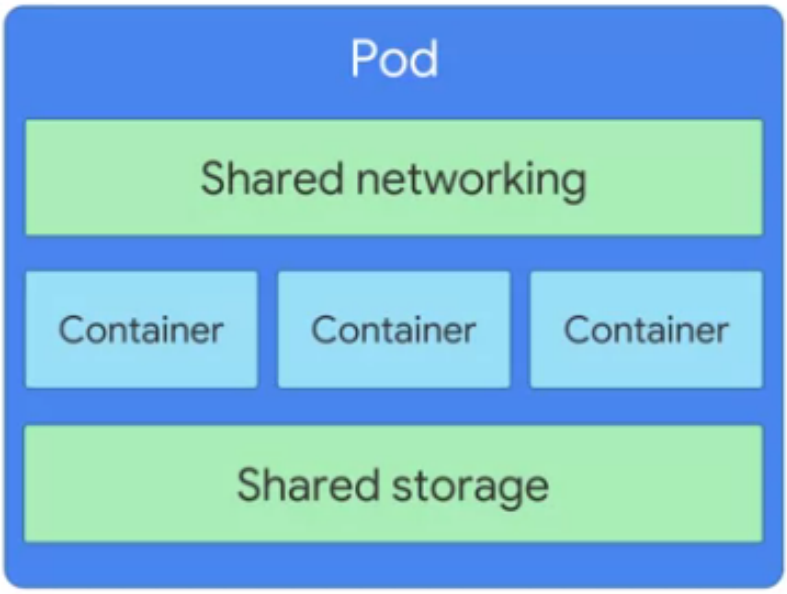


2.1.6 Kubernetes Concepts

- Two primary principles:
  - Each thing Kubernetes manages is represented by an **object**. You can view and change these objects, attributes, and state.
  - The second is the principle of **declarative management**.
    - Kubernetes expects you to tell it what you want the state of the objects under each management to be. It will work to bring that state into being and keep it there.

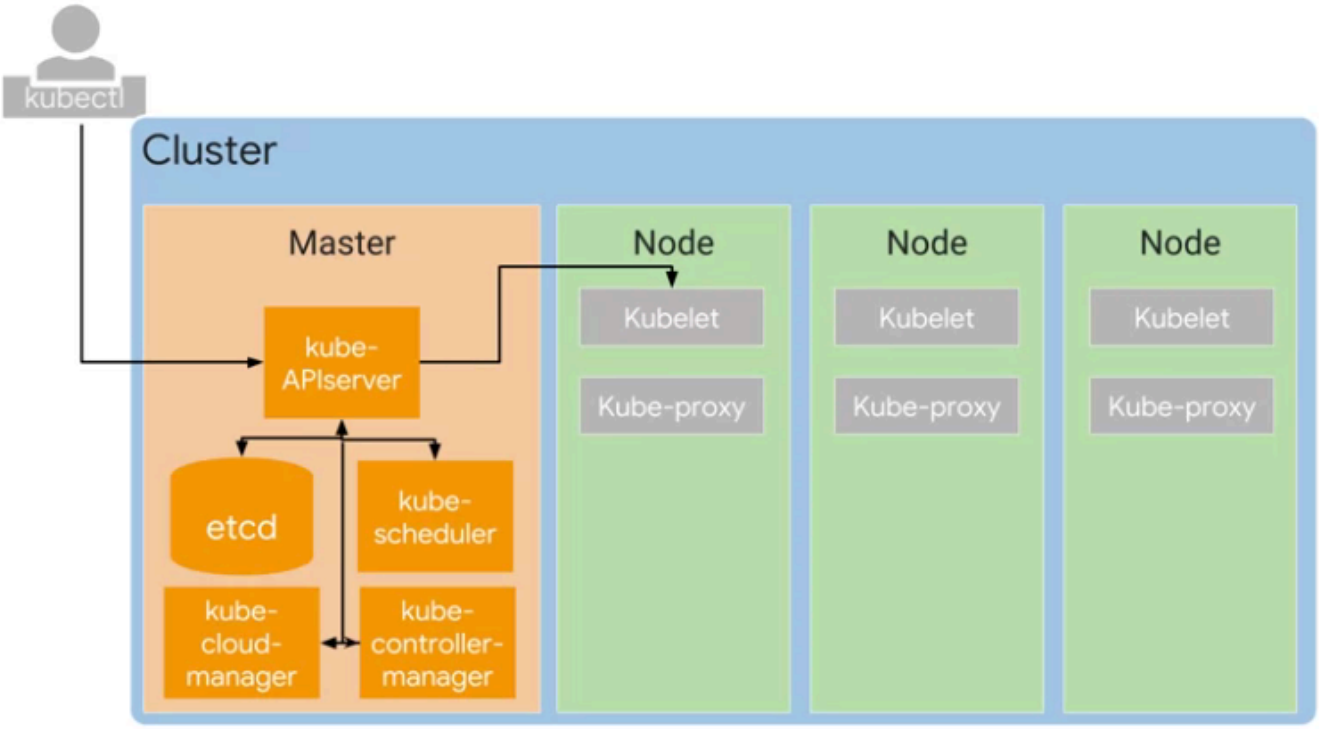


- **Pods** are basic building blocks



  - Pods can have one or more containers all sharing storage and networking.
  - Containers within the same pod can communicate via local host.

- **Control Plane**



  - Cluster has a **master** and **nodes**.
  - ***Master***
    - The master coordinates the entire cluster.
    - The single component that you interact with directly is the **kube-APIserver**.
    - This component's job is to accept commands that view or change the state of the cluster, including launching pods. We use the ***kubectl*** command for this.
    - ***kubectl*** command's job is to connect to kube-APIserver and communicate with it using the Kubernetes API.
    - **etcd** is the cluster's database.
      - Its job is to reliably store the state of the cluster.
      - This includes all the cluster configuration data and more dynamic information such as what nodes are part of the cluster, what pods should be running, and where they should be running.
      - You never interact directly with etcd.
      - Instead, kube-APIserver interacts with the database on behalf of the rest of the system.

- **Kube-scheduler** is responsible for scheduling pods onto the nodes.
    - It evaluates the requirements of each individual pod and selects which node is most suitable.
    - It doesn't do the work of actually launching pods onto nodes.
    - Whenever it discovers a pod object that doesn't yet have an assignment to a node, it chooses a node and simply writes the name of that node into the pod object. Another component of the system is responsible for then launching the pods.

- **Kube-controller manager**
    - It continuously monitors the state of a cluster through kube-APIserver.
    - Whenever the current state of the cluster doesn't match the desired state, kube-controller manager will attempt to make changes to achieve the desired state.
    - It's called the controller manager because many Kubernetes objects are maintained by loops of code called ***controllers***. These loops of code handle the process of remediation.
    - Different kinds of controller objects:
        - We can gather pod information together into a controller object called a ***deployment*** that not only keeps them running, but also lets us scale them and bring them together underneath our front end.
        - ***Node controller***: Its job is to monitor and respond when a node is offline.
        - ***Kube-cloud-manager*** manages controllers that interact with underlying cloud providers.

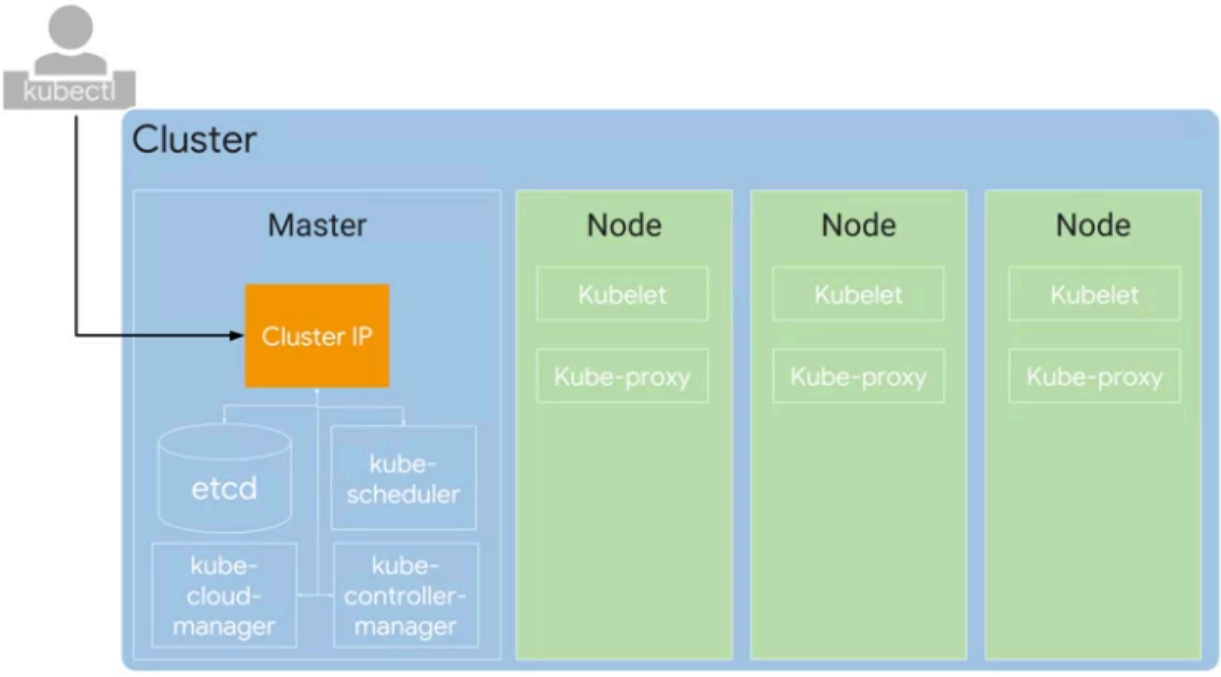
- **Nodes**:
    - Each node runs a small family of control-plane components too.
    - ***Kubelet***
        - kubelet is a Kubernetes agent on each node.
        - When the kube-APIserver wants to start a pod on a node, it connects to that node's kubelet.
        - Kubelet uses the container runtime to start the pod and monitor its lifecycle, including readiness and liveness probes, and reports back to kube-APIserver.
    - ***Container runtime*** is the software that knows how to launch a container from a container image.

- The world of Kubernetes offers several choices of container runtimes, but the Linux distribution that GKE uses for its nodes launches containers using **containerd**.

- **Kube proxy:** Its job is to maintain network connectivity among the pods in a cluster.

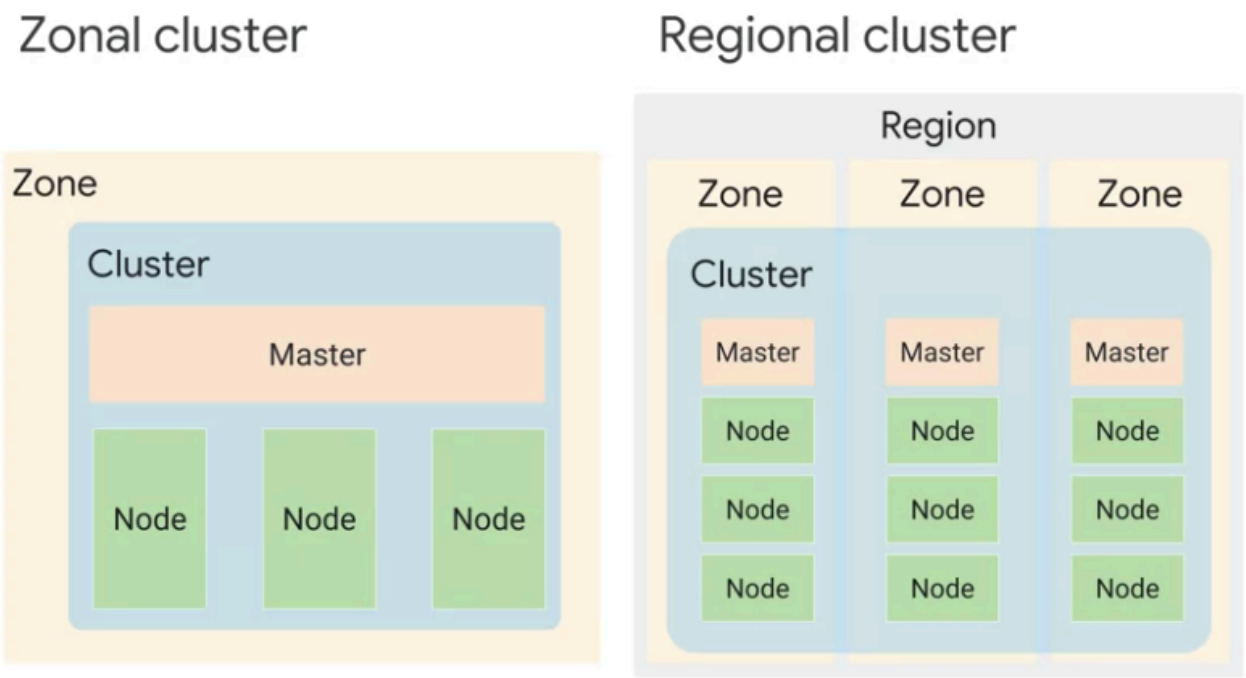
2.1.6.1 GKE Specific Concepts

- GKE provides management of nodes which is not available out of the box with Kubernetes.



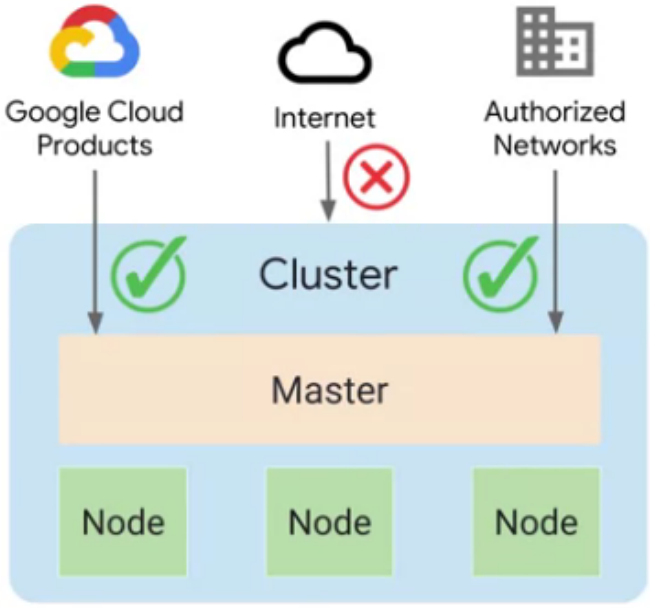
  - In any Kubernetes environment, nodes are created externally by cluster administrators, not by Kubernetes itself.
  - GKE automates this process for you. It launches Compute Engine virtual machine instances and registers them as nodes.
  - You can manage node settings directly from the GCP console.
- GKE provides ***node pooling***.

- **Zones and regions**

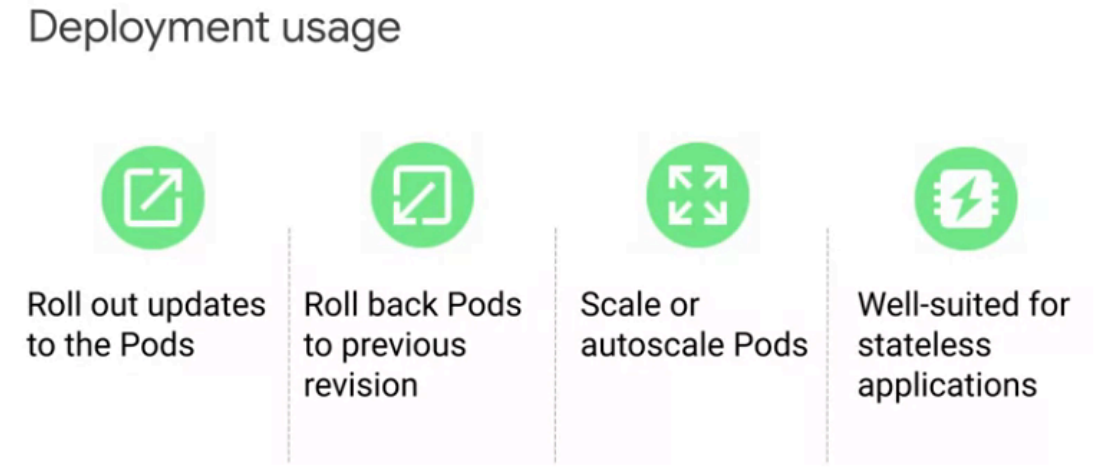


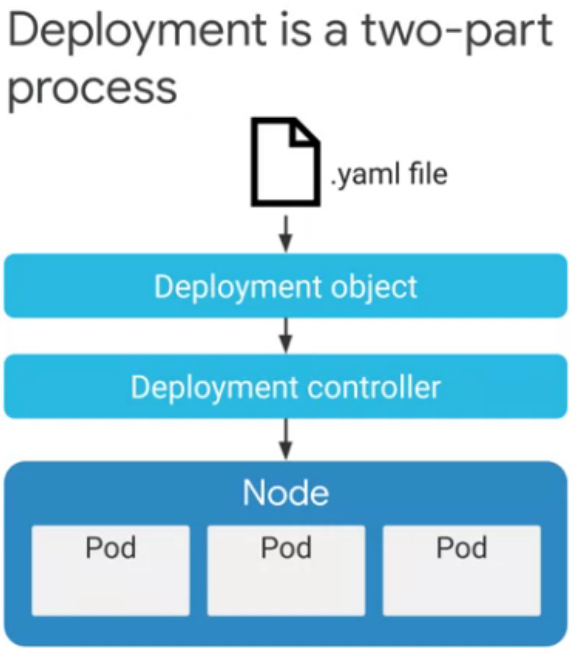
  - Default is a cluster in a single zone with 3 nodes.
  - Regional clusters have a API endpoint for the cluster, but masters and nodes are spread across zones. Default is 3 masters each with 3 nodes.

- **Private Clusters**



2.1.7 Deployments

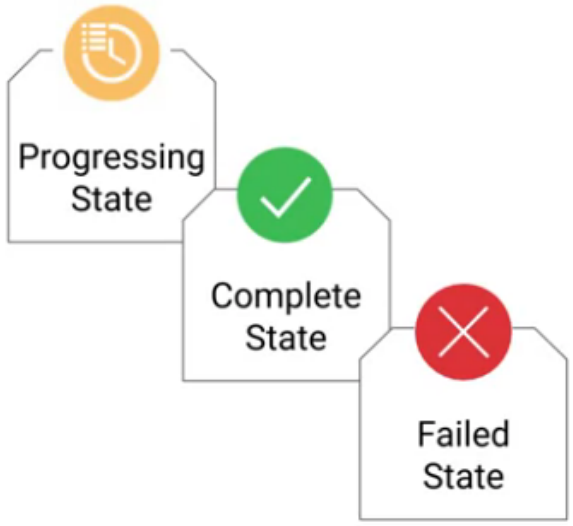




Deployment object file in YAML format

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-app
        image: gcr.io/demo/my-app:1.0
        ports:
        - containerPort: 8080
```

Deployment has three different lifecycle states



2.1.7.1 Deployment updates

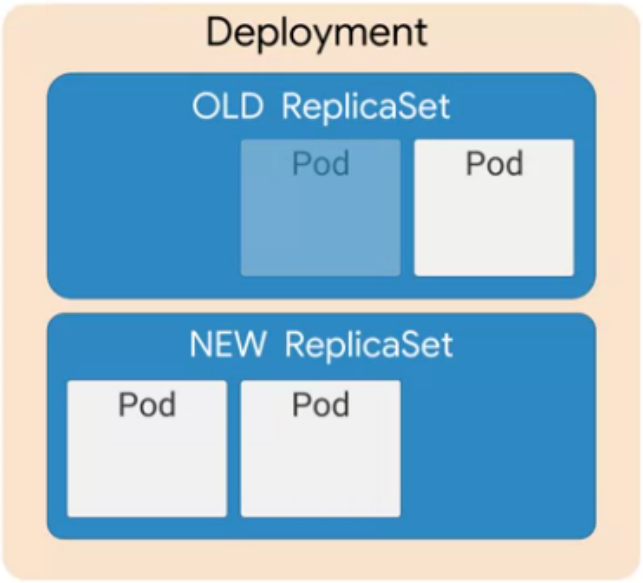
2.1.7.1.1 Recreate Strategy (Best for dev environment)

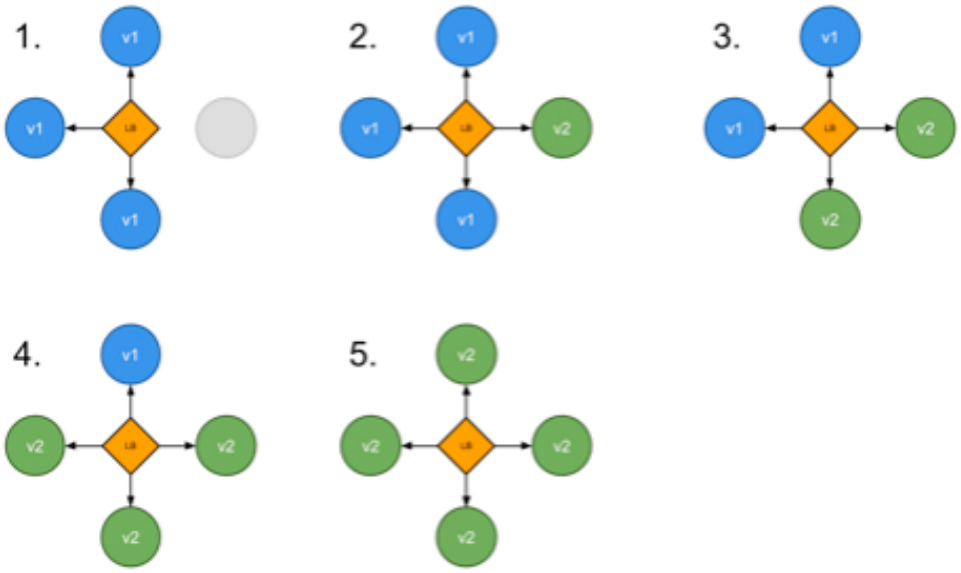
```
[...]
kind: deployment
spec:
    replicas: 10
    strategy:
        type: Recreate
[...]
```

- This update terminates all the running instances then recreate them with the newer version.
- **Pro**:
    - application state entirely renewed
- **Cons**:
    - downtime that depends on both shutdown and boot duration of the application

2.1.7.1.2 Ramped Strategy (Slow rollout)

```
[...]
kind: deployment
spec:
  replicas: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 5
      maxUnavailable: 10%
[...]
```

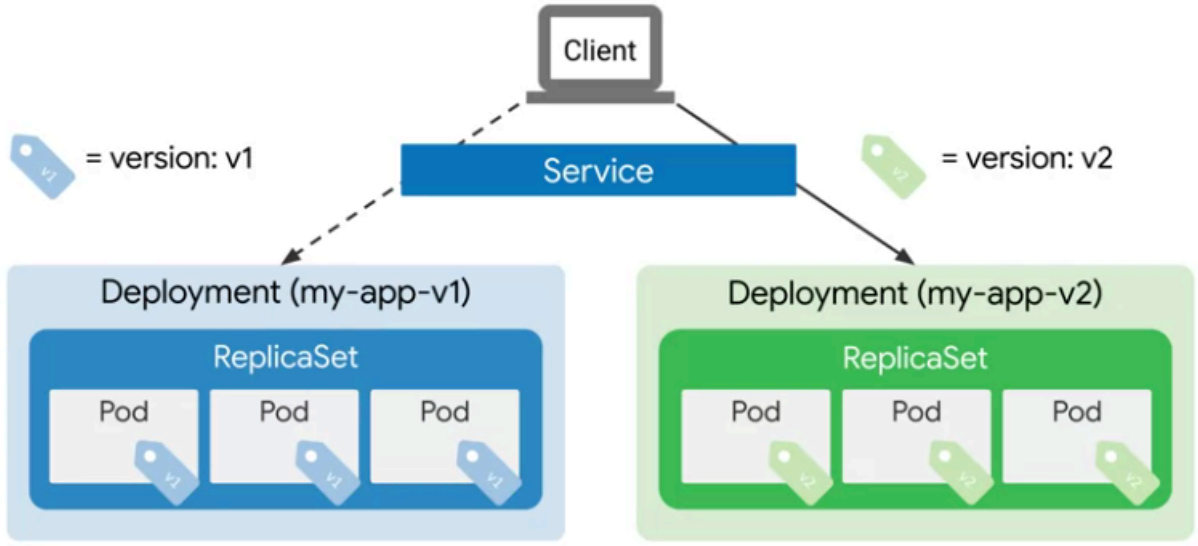




- A ramped deployment updates pods in a rolling update fashion, a secondary ReplicaSet is created with the new version of the application, then the number of replicas of the old version is decreased and the new version is increased until the correct number of replicas is reached.
    - **Max unavailable**: % of the total number of Pods across all ReplicaSets or an absolute number
    - **Max Surge**: The total number of Pods across all ReplicaSets. Value can be absolute number or percentage of total pods across ReplicaSets.

- **Pro:**
  - version is slowly released across instances
  - convenient for stateful applications that can handle rebalancing of the data
- **Cons:**
  - rollout/rollback can take time
  - supporting multiple APIs is hard
  - no control over traffic

2.1.7.1.3 Blue Green Strategy (Best to avoid API version change issues)



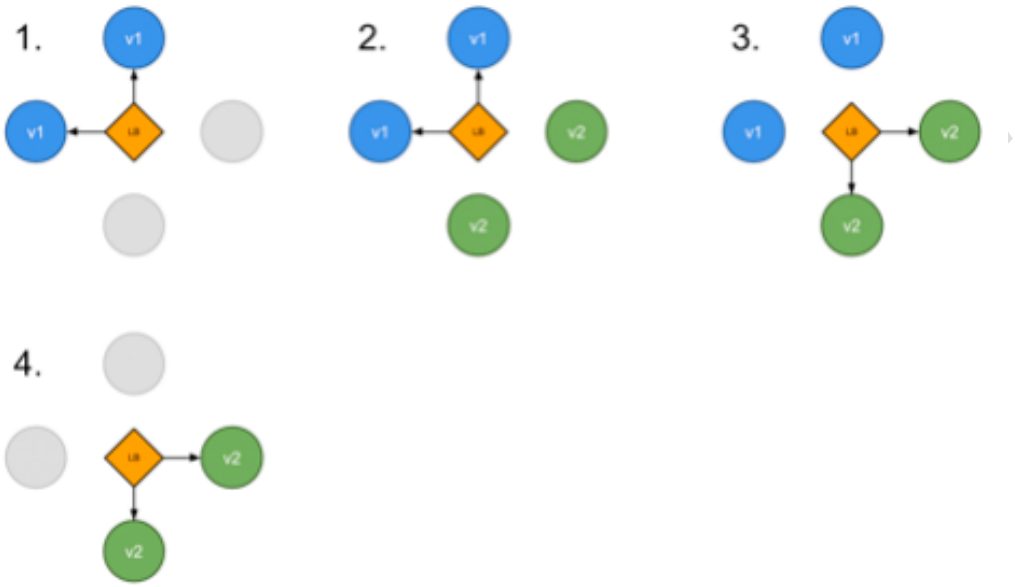

```
[...]
kind: Service
spec:
    selector:
      app: my-app
      version: v1
[...]
```

```
$ kubectl apply -f my-app-v2.yaml
```

```
[...]
kind: Service
spec:
    selector:
      app: my-app
      version: v2
[...]
```

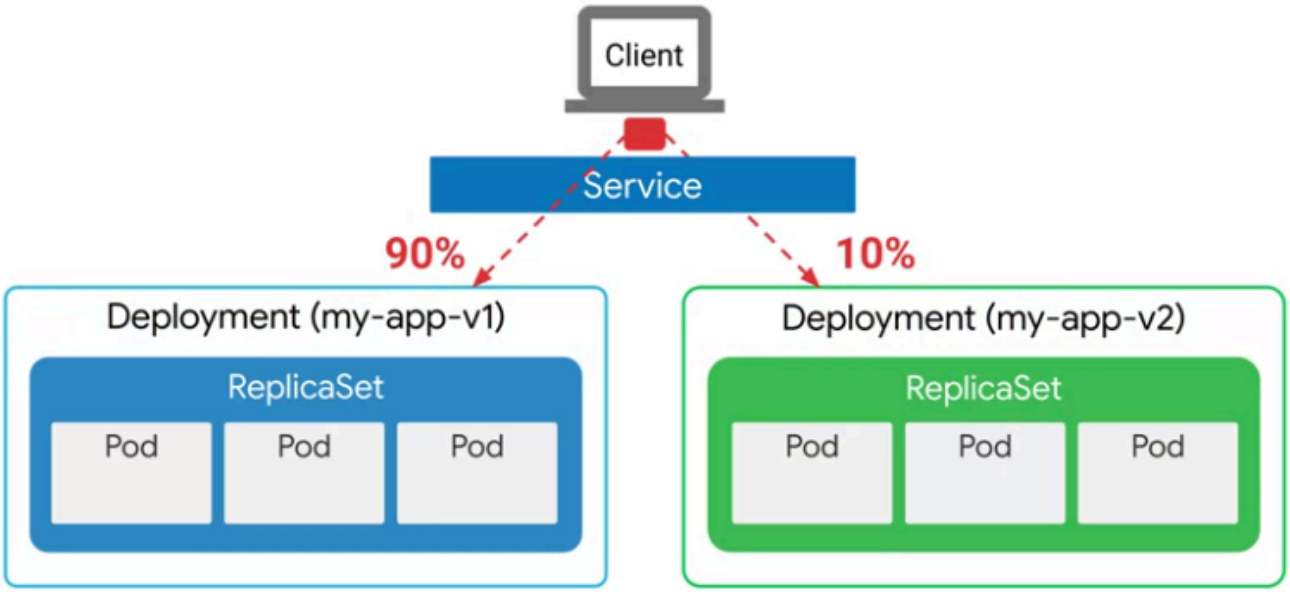
```
$ kubectl patch service my-app-service -p \
'{"spec":{"selector":{"version":"v2"}}}
```

- A blue/green deployment differs from a ramped deployment because the "green" version of the application is deployed alongside the "blue" version. After testing that the new version meets the requirements, we update the Kubernetes Service object that plays the role of load balancer to send traffic to the new version by replacing the version label in the selector field.



  - **Pro:**
    - instant rollout/rollback
    - avoid versioning issue, change the entire cluster state in one go
  - **Cons:**
    - requires double the resources
    - proper test of the entire platform should be done before releasing to production
    - handling stateful applications can be hard

2.1.7.1.4 Canary Strategy (clients of the service test updates)




```
[...]
kind: Service
spec:
     selector:
        app: my-app
[...]
```

```
$ kubectl apply -f my-app-v2.yaml
```

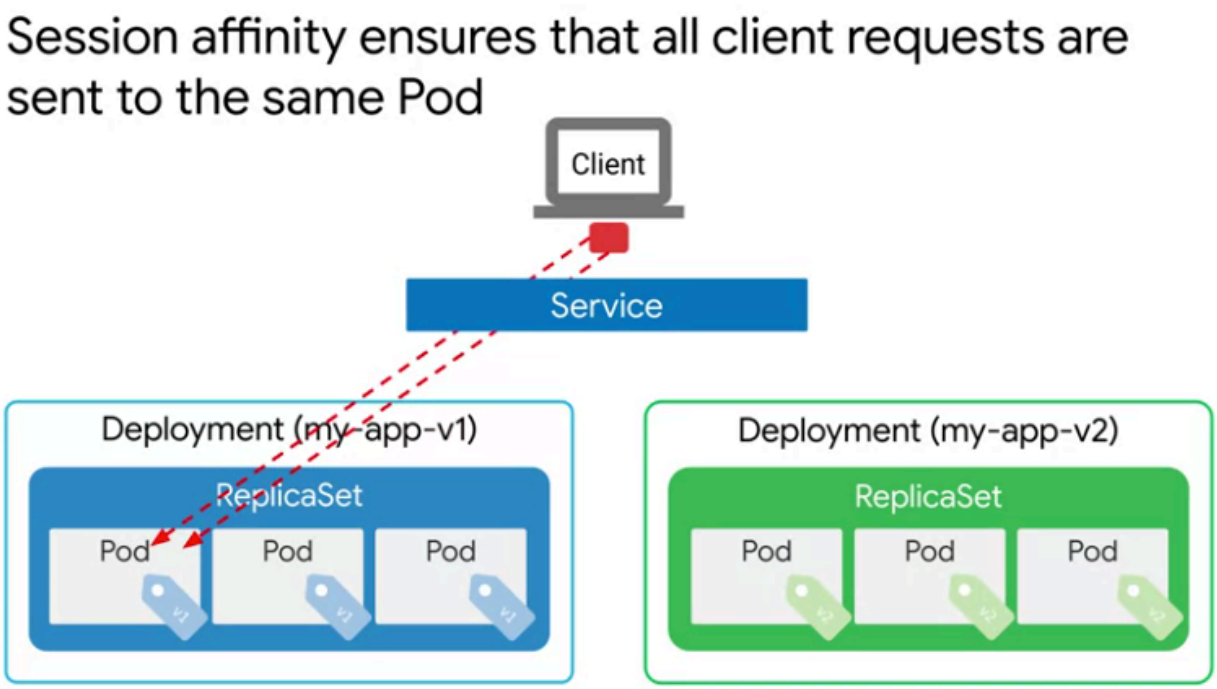
```
$ kubectl scale deploy/my-app-v2 -replicas=10
```

```
$ kubectl delete -f my-app-v1.yaml
```

- A canary deployment consists of routing a subset of users to a new functionality. In Kubernetes, a canary deployment can be done using two Deployments with common pod labels. One replica of the new version is released alongside the old

version. Then after some time and if no error is detected, scale up the number of replicas of the new version and delete the old deployment.

- In a Canary update strategy, the service selector is based on the application level and does not specify the version.
    - The Selector in this example covers all pods with the app: my- label.
    - This means that with this Canary update strategy version of the service traffic is sent to all pods, regardless of the version label, this setting allows the service to select and direct traffic to pods from both deployments.
    - Initially, the new version of the deployment will start with zero replicas running and over time as the new version is scaled up, the old version of deployment can be scaled down and eventually deleted.
- With the Canary update strategy, a subset of users will be directed to the new version, this allows you to monitor for errors in performance issues as these users use the new version and you can quickly rollback minimizing the impact to your overall user base if any issues arise.
- However, the complete rollout of a deployment using Canary strategy can be a slow process and may require tools such as STO to accurately shift the traffic.



Rolling back a Deployment

```
$ kubectl rollout undo deployment
[DEPLOYMENT_NAME]
```

```
$ kubectl rollout undo deployment
[DEPLOYMENT_NAME] --to-revision=2
```

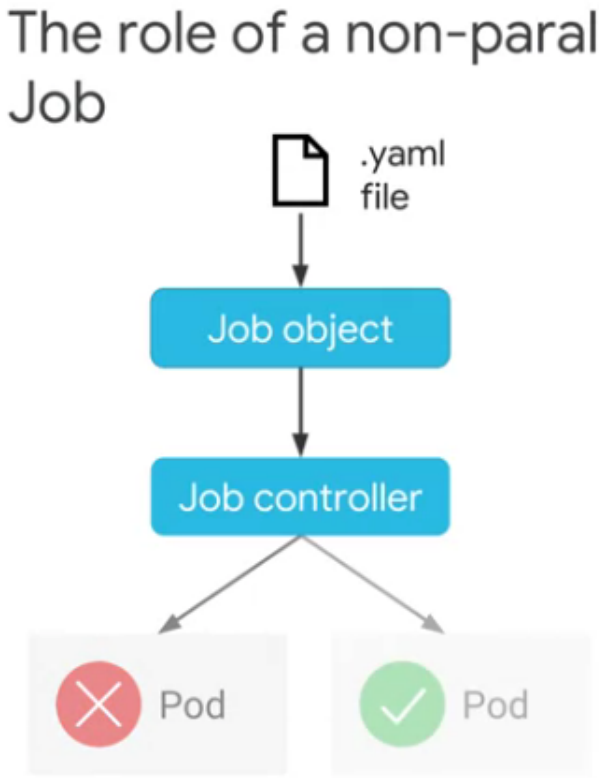
```
$ kubectl rollout history deployment
[DEPLOYMENT_NAME] --revision=2
```

Clean up Policy:

- Default: 10 Revision
- Change: .spec.revisionHistoryLimit

- **Pro:**
  - version released for a subset of users
  - convenient for error rate and performance monitoring
  - fast rollback
- **Cons:**
  - slow rollout
  - fine-tuned traffic distribution can be expensive (99% A/ 1%B = 99 pod A, 1 pod B)

2.1.7.2 Jobs



A Job computing π to 2000 places

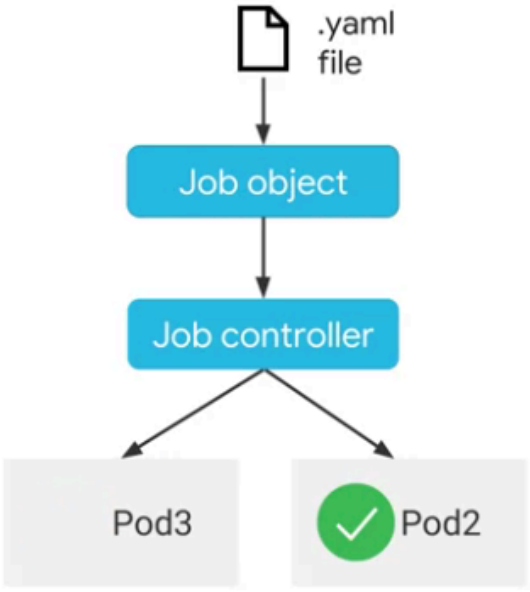
```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
  backoffLimit: 4
```

```
$ kubectl apply -f [JOB_FILE]
```

```
$ kubectl run pi --image perl --restart Never -- perl -Mbignum bpi -wle 'print bpi(2000)'
```

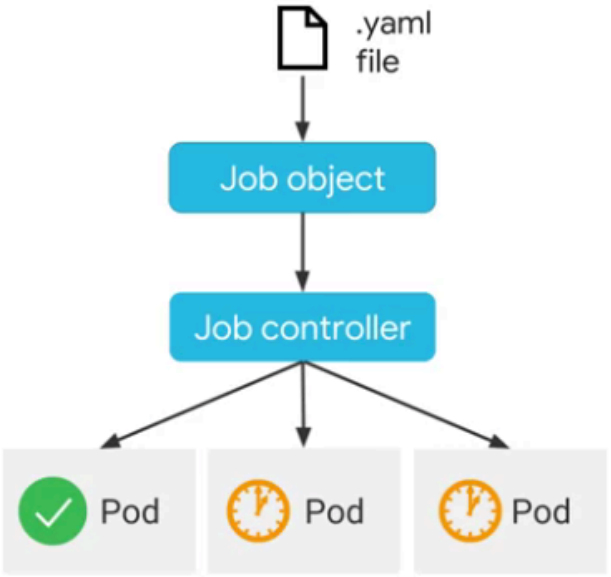
Parallel Job with fixed completion count

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: my-app-job
spec:
  completions: 3
  parallelism: 2
  template:
    spec:
[...]
```

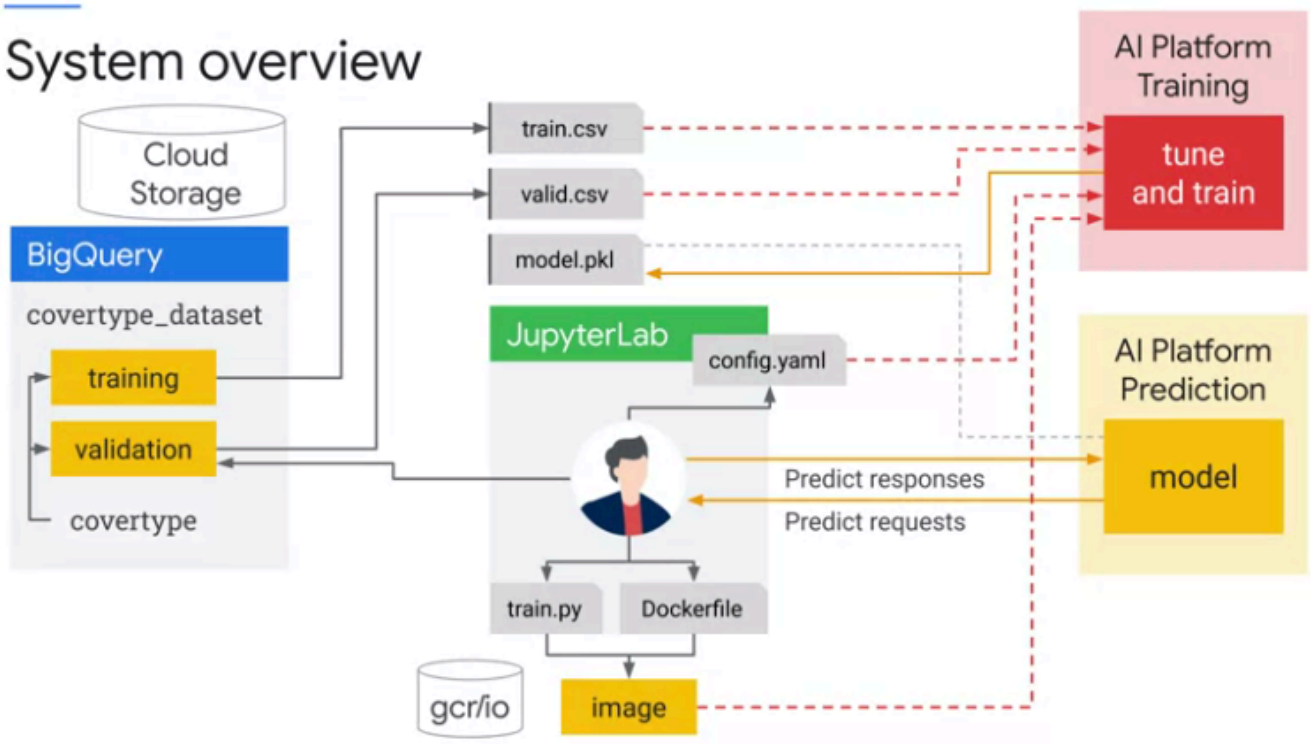


Parallel Job with a worker queue

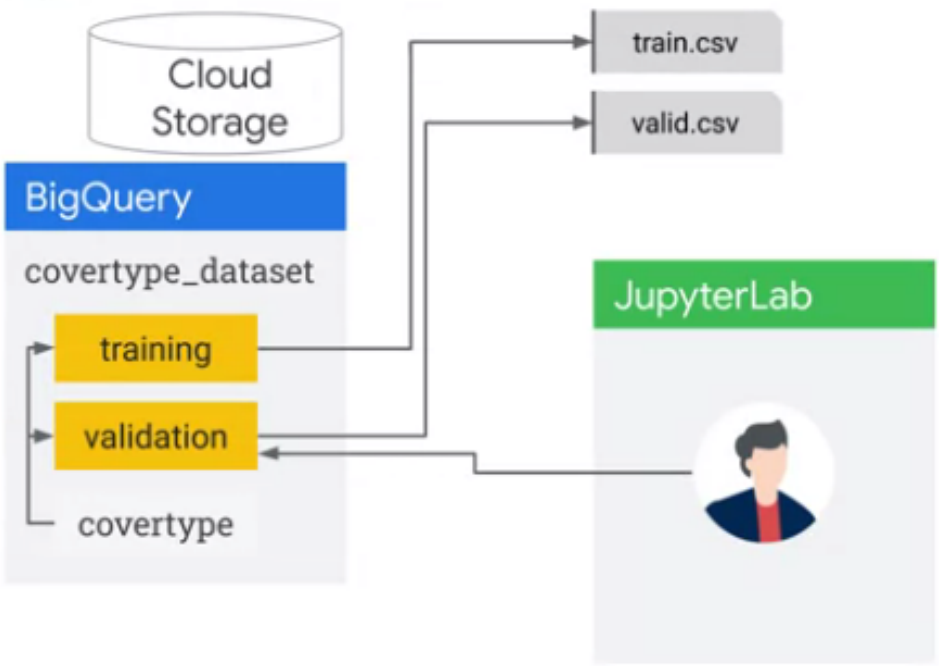
```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: my-app-job
spec:
  parallelism: 3
  template:
    spec:
[...]
```



2.2 ML Ops Systems and automation

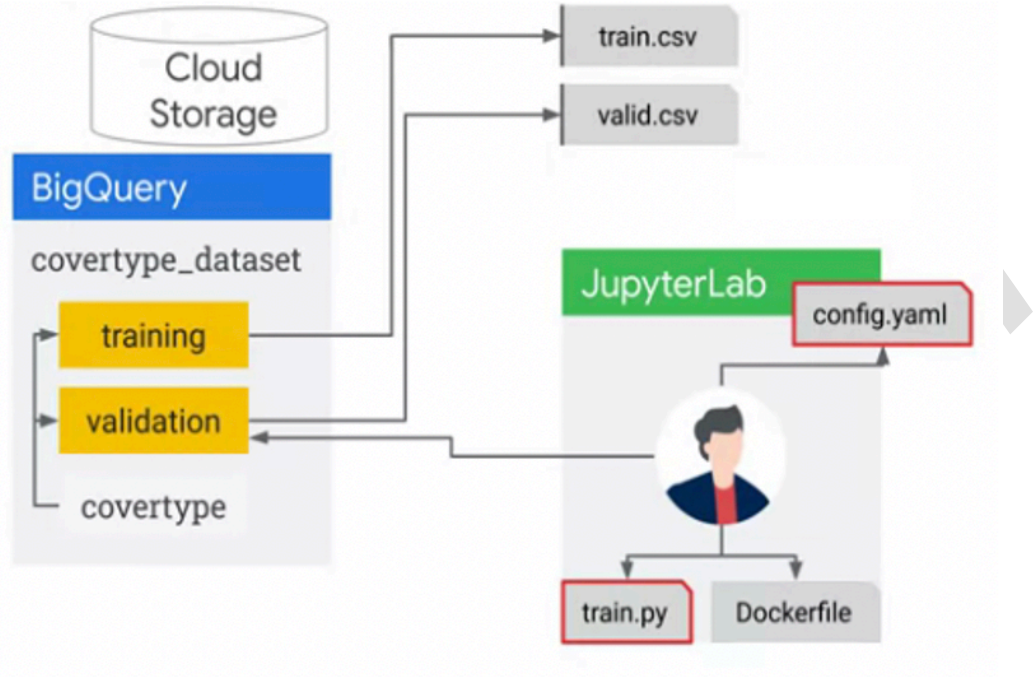


2.2.1 STEP 1: Create reproducible datasets



- Query using BigQuery. (refer hashing to see how to create reproducible datasets)
- Store the split dataset in cloud storage for training. (train.csv and valid.csv)
- Use Jupyter notebooks in Jupyter lab for experimentation.

2.2.2 STEP 2: Implement a tunable model



- Create the model code, train.py
    - Example train.py file (using sklearn pipeline)

train.py

```
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numeric_feature_indexes),
        ('cat', OneHotEncoder(), categorical_feature_indexes)
    ])

pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', SGDClassifier(loss='log', tol=1e-3))
])

pipeline.set_params(classifier__alpha=0.001, classifier__max_iter=200)
pipeline.fit(X_train, y_train)

accuracy = pipeline.score(X_validation, y_validation)
```

- To be able to leverage AI Platform to auto tune the hyper parameters, the following modifications should be done:
    1. Make the hyperparameter a command-line argument
    2. Setup cloudml-hypertune to record training metrics
    3. Export the final trained model
    4. Supply hyperparameters to the training job

2.2.2.1 Make the hyperparameter a command-line argument

- Use the ***fire*** package to make hyperparameters command line arguments

train.py

```
import fire

def train_evaluate(job_dir,
                   training_dataset_path,
                   validation_dataset_path,
                   alpha, max_iter, hptune):


    # [...]


if __name__ == "__main__":
    fire.Fire(train_evaluate)
```

```
Python train.py \
  --job_dir $JOBDIR \
  --training_dataset_path $TRAINING_PATH \
  --validation_dataset_path $VALID_PATH \
  --alpha \
  --maxiter \
  --hptune
```

- Execute locally to test it works fine.

2.2.2.2 Setup cloudml-hypertune to record training metrics

- Import the cloudml-hypertune Python package from Google to capture the tuning metrics.
- In the backend, hypertune will simply write this metric on the file system where the training Container is executed, at a location where the AI platform training knows to retrieve it.

train.py

```
import hypertune

def train_evaluate(job_dir,
                   training_dataset_path,
                   validation_dataset_path,
                   alpha, max_iter, hptune):
    # [...]

    if hptune:
        accuracy = pipeline.score(X_validation, y_validation)

        hpt = hypertune.HyperTune()

        hpt.report_hyperparameter_tuning_metric(
          hyperparameter_metric_tag='accuracy',
          metric_value=accuracy
        )

if __name__ == "__main__":
    fire.Fire(train_evaluate)
```

Capture the metrics.

2.2.2.3 Write code to export the final trained model

- The final model using the best hyper parameters will need to be persisted.
    - Model need not be saved during tuning though.

train.py

```
import pickle

def train_evaluate(job_dir,
                   training_dataset_path,
                   validation_dataset_path,
                   alpha, max_iter, hptune):
    # [...]

    if not hptune:
        model_filename = 'model.pkl'
        with open(model_filename, 'wb') as model_file:
            pickle.dump(pipeline, model_file)
        gcs_model_path = "{}/{}".format(job_dir, model_filename)
        subprocess.check_call(['gsutil', 'cp', model_filename, gcs_model_path],
        stderr=sys.stdout)

if __name__ == "__main__":
    fire.Fire(train_evaluate)
```

    - After serializing the model (using pickle), the model is copied to Google Cloud Storage.

2.2.2.4 Supply hyperparameters to the training job

- Create config.yaml which is the config as well as used to tune the hyperparameters

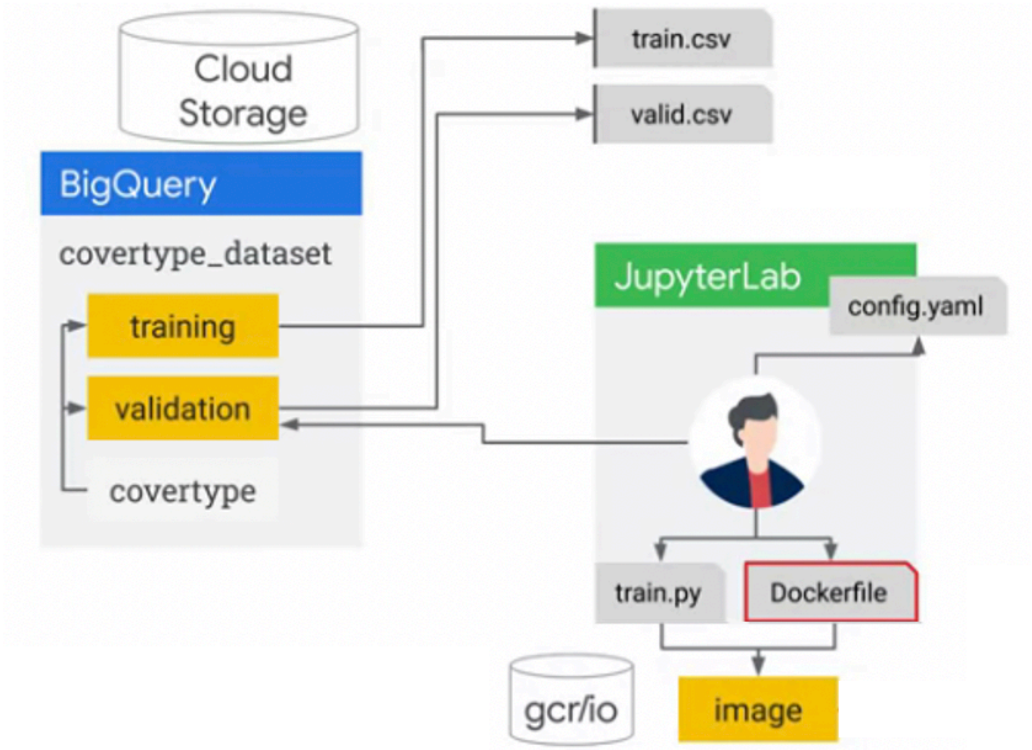
config.yaml

```
trainingInput:
  hyperparameters:
    goal: MAXIMIZE
    maxTrials: 4
    maxParallelTrials: 4
    hyperparameterMetricTag: accuracy
    enableTrialEarlyStopping: TRUE
    params:
    - parameterName: max_iter
      type: DISCRETE
      discreteValues: [
          200,
          500
          ]
    - parameterName: alpha
      type: DOUBLE
      minValue:  0.00001
      maxValue:  0.001
      scaleType: UNIT_LINEAR_SCALE
```

```
gcloud ai-platform jobs submit training $JOB_NAME \
  -- [...]
  --config config.yaml \
  -- \
  --training_dataset_path=$TRAINING_FILE_PATH \
  --validation_dataset_path=$VALIDATION_FILE_PATH \
  --hptune
```

- Launch training and tuning using the gcloud command as shown above. (done in step 4)

2.2.3 STEP 3: Build and push a container



- Create the ***training*** docker file

Dockerfile

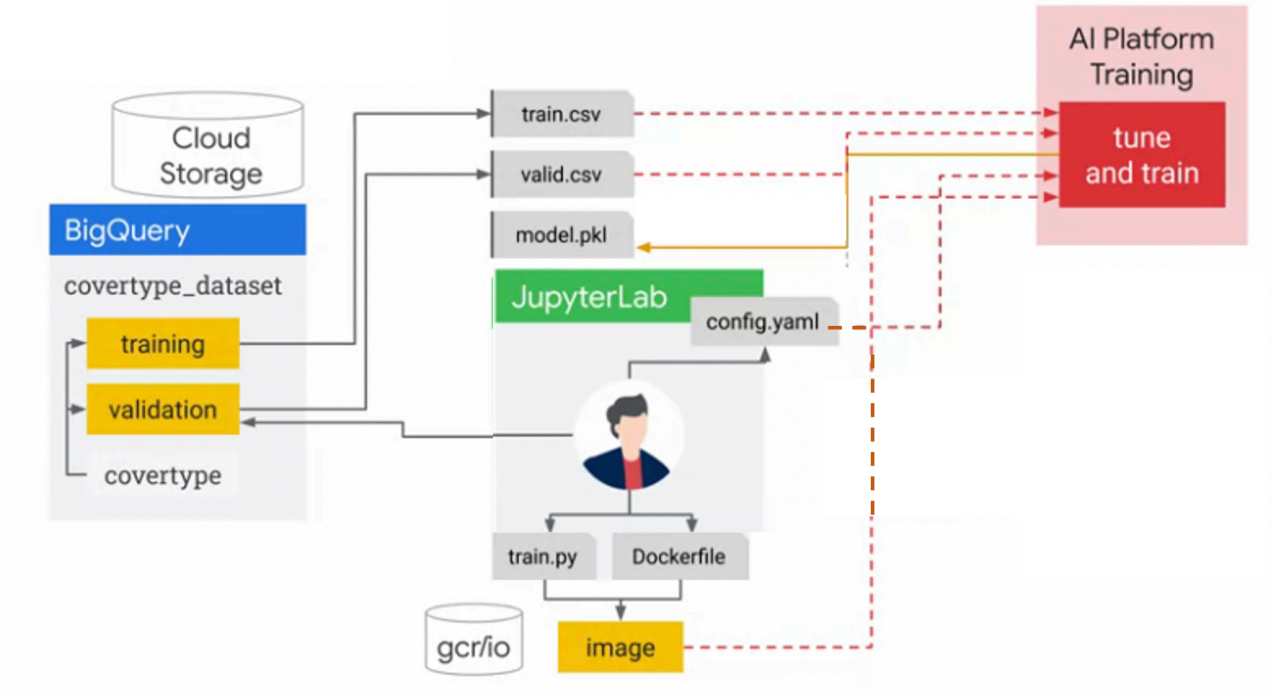
```
FROM gcr.io/deeplearning-platform-release/base-cpu

RUN pip install -U fire cloudml-hypertune scikit-learn==0.20.4 pandas==0.24.2

WORKDIR /app

COPY train.py .

ENTRYPOINT ["python", "train.py"]
```

- Build the image and submit to the Google Cloud Registry

```
gcloud builds submit --tag gcr.io/$PROJECT/$IMAGE:$TAG $TRAINING_APP_FOLDER
```

  - The $TRAINING_APP_FOLDER points to the location containing the docker file and train.py files.
  - The command also takes in the image URI that gets built as specified above.

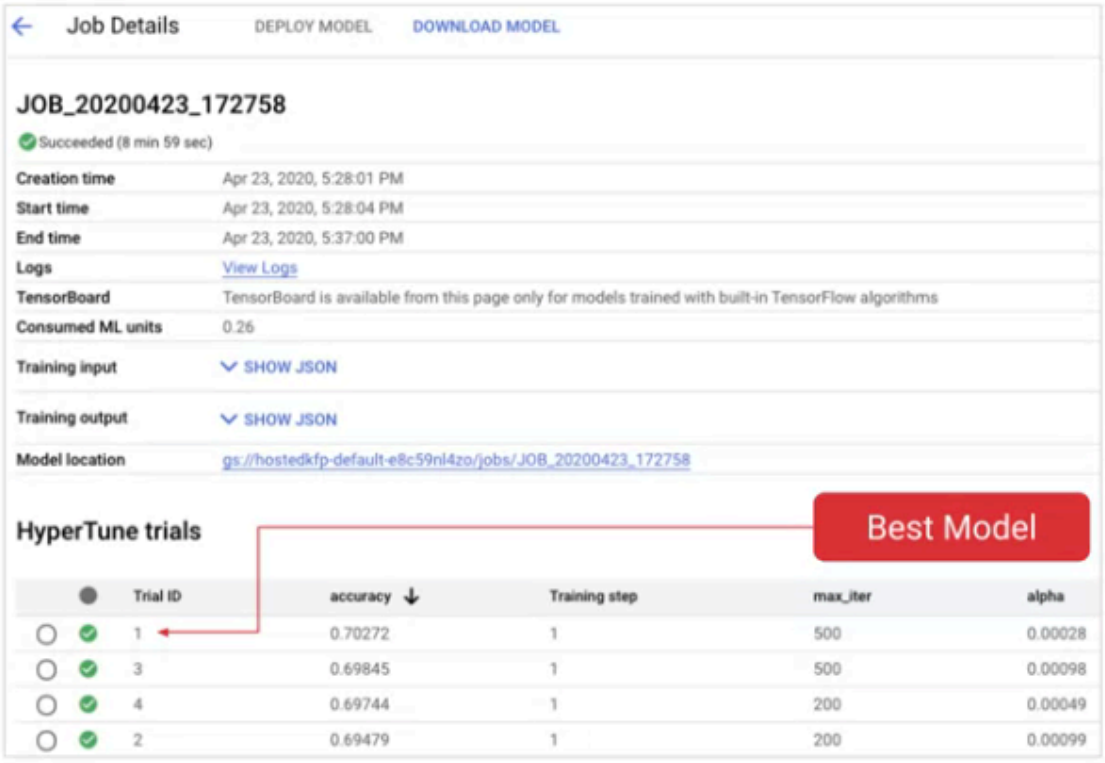
2.2.4 STEP 4: Train and Tune the model



- ***First get best hyperparameters (TUNE)***
    - Use the gcloud command to submit the training job to AI Cloud Platform

```
gcloud ai-platform jobs submit training $JOB_NAME \
  --region=$REGION \
  --job-dir=$JOB_DIR \
  --master-image-uri=$IMAGE_URI \
  --scale-tier=$SCALE_TIER \
  --config $TRAINING_APP_FOLDER/hptuning_config.yaml \
  -- \
  --training_dataset_path=$TRAINING_FILE_PATH \
  --validation_dataset_path=$VALIDATION_FILE_PATH \
  --hptune
```

    - Execute command on Jupyter Lab to start training.
    - Best model will be visible on the cloud console on the top.



- ***Next, train the model with best hyperparameters (TRAIN)***
    - Query AI Platform to retrieve hyperparameter values

```
from googleapiclient import discovery

ml = discovery.build('ml', 'v1')

job_id = 'projects/{}/jobs/{}'.format(PROJECT_ID, JOB_NAME)
request = ml.projects().jobs().get(name=job_id)

response = request.execute()

alpha = response['trainingOutput']['trials'][0]['hyperparameters']['alpha']
max_iter = response['trainingOutput']['trials'][0]['hyperparameters']['max_iter']
```

        - The first trial has the best values

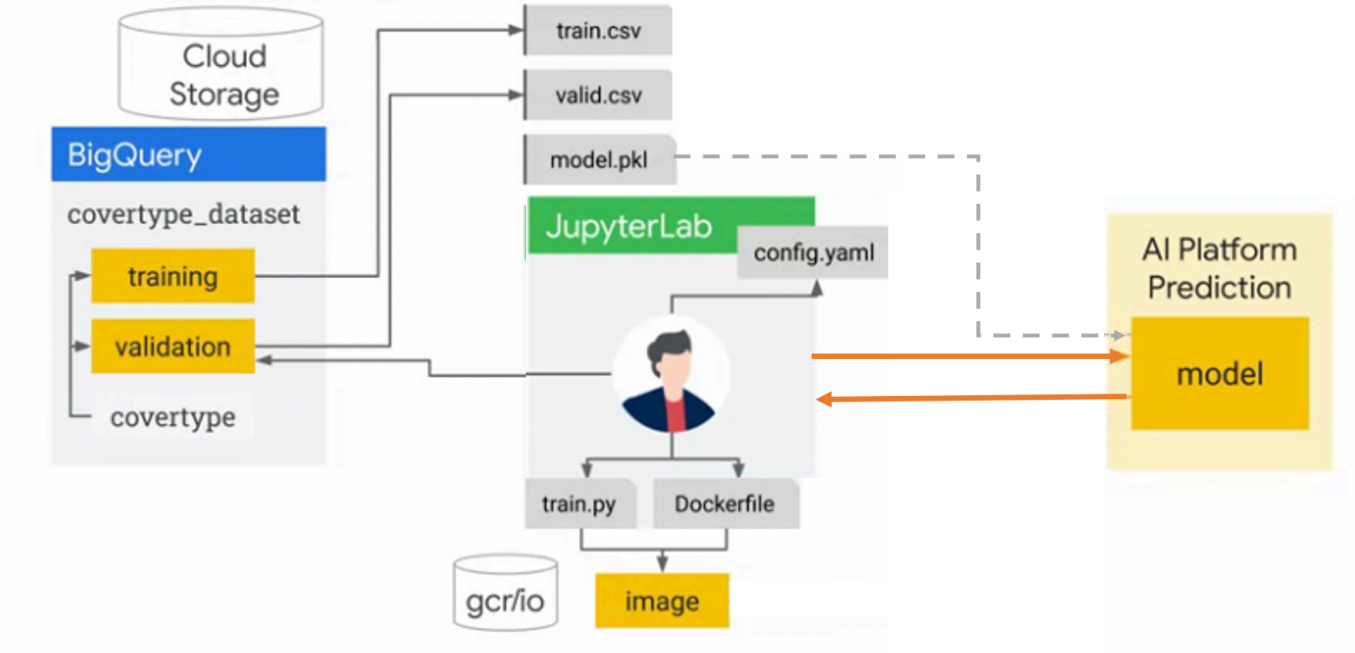
    - Next, submit a job for training the model again.

```
gcloud ai-platform jobs submit training $JOB_NAME \
      --region=$REGION \
      --job-dir=$JOB_DIR \
      --master-image-uri=$IMAGE_URI \
      --scale-tier=$SCALE_TIER \
      -- \
      --training_dataset_path=$TRAINING_FILE_PATH \
      --validation_dataset_path=$VALIDATION_FILE_PATH \
      --alpha=$alpha \
      --max_iter=$max_iter \
      --nohptune
```

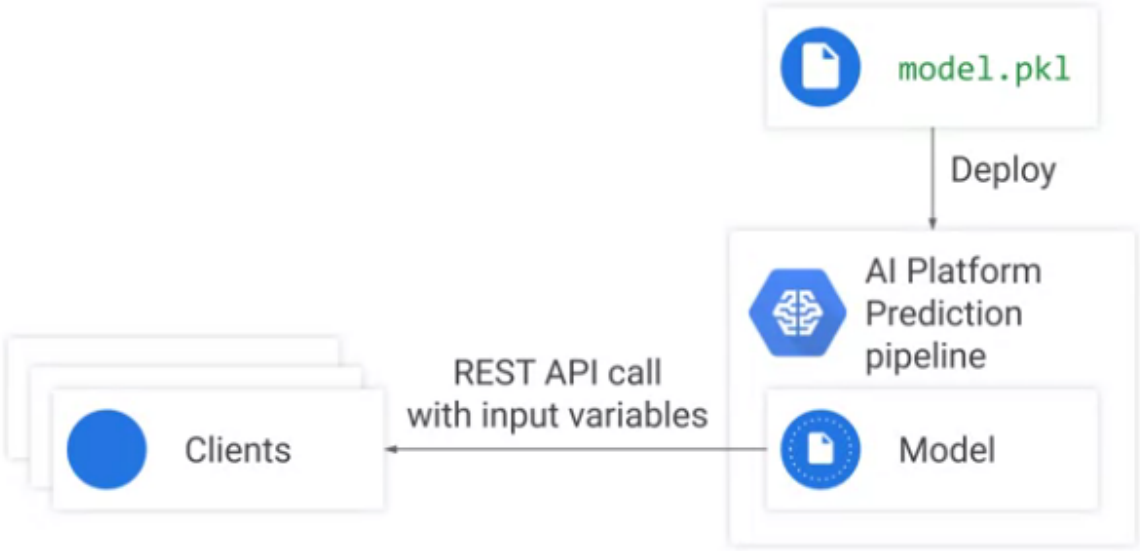
Training done with `hptune = False`

- Use the same command used for tuning, except:
  - Remove the config file
  - Specify hyperparameter values explicitly
  - Set hptune Flag to False. (This ensures model gets saved in train.py)
- By the end, the model.pkl will be exported on cloud storage.

2.2.5 STEP 5: Serve and Query the Model



- Here, we deploy model as a REST API



  - To **deploy** a model, there are 2 steps:
    - **Create a model resource(object)**

```
gcloud ai-platform models create $model_name \
  --regions=$REGION \
  --labels=$labels
```

    - **Create a model version**

```
gcloud ai-platform versions create {model_version} \
    --model={model_name} \
    --origin=$JOB_DIR \
    --runtime-version=1.15 \
    --framework=scikit-learn \
    --python-version=3.7
```

      - This where actual model is tied to a version
      - Since we use a pickle file, we specify framework as scikit-learn.

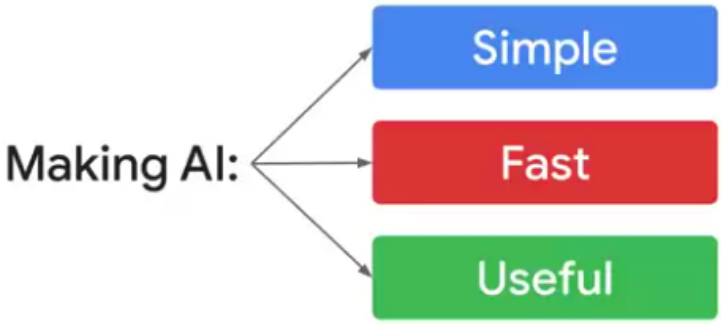
- **Query** the model (prediction request)

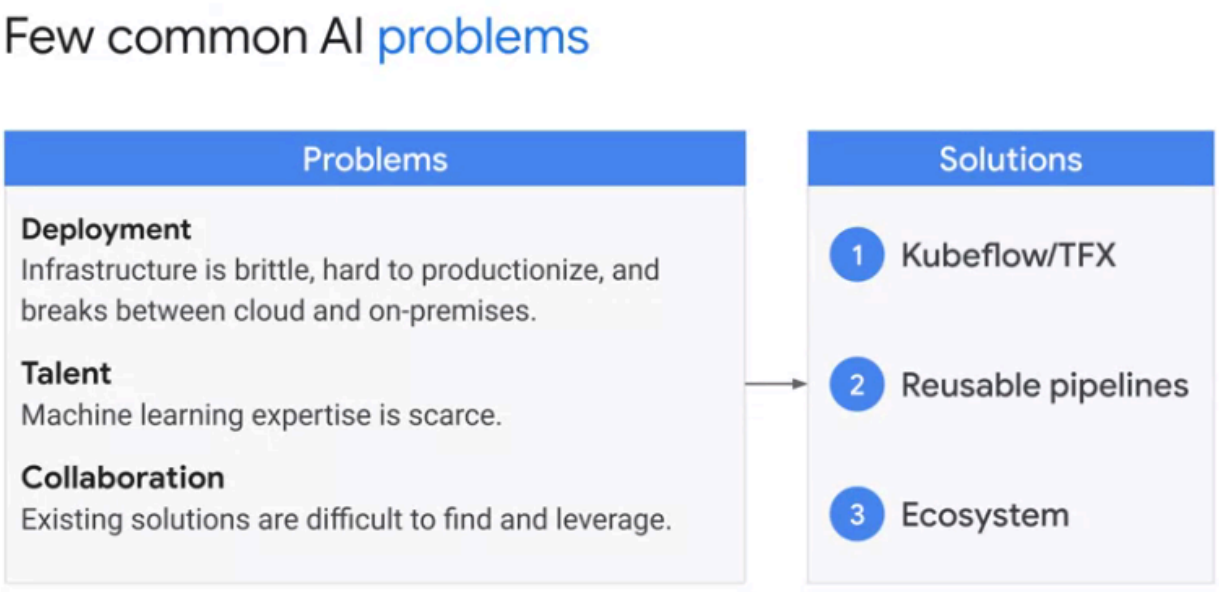
```
gcloud ai-platform predict \
  --model $model_name \
  --version $model_version \
  --json-instances $input_file
```

The data we send to the prediction API

3 ML Pipelines

3.1 AI Platform Pipelines





1 Kubeflow/TFX scalable ML services on Kubernetes

Easy to get started

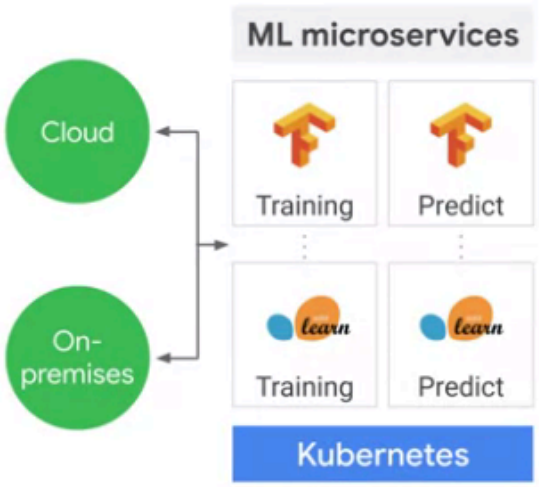
- Kubeflow: Out-of-box support for top frameworks
  - pytorch, caffe, tf, and xgboost
- TFX: Google best practices on TF
- Kubernetes manages dependencies and resources

Swappable and scalable

- Library of ML services
- GPU support
- Massive scale

Meet customers where they are

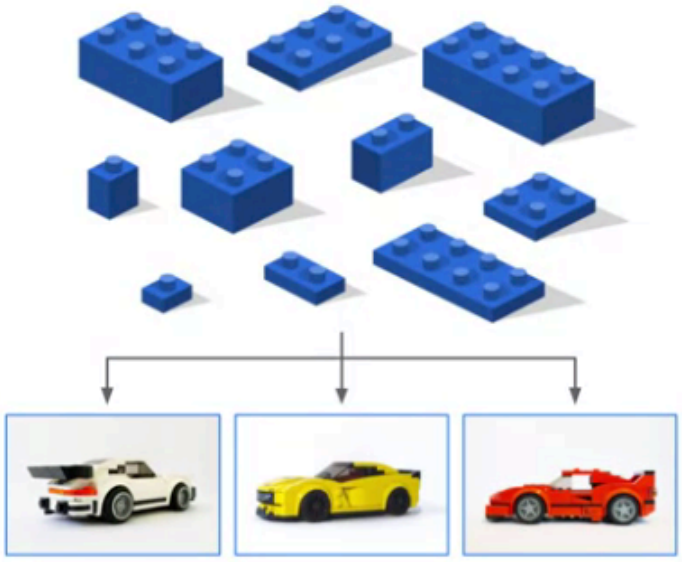
- Google Cloud
- On-premises



2 Reusable pipelines

Enable developers to build custom ML applications by easily"stitching" and connecting various components, like LEGO.

- Reuse instead of reimplement or reinvent.
- Discover, learn, and replicate successful pipelines.

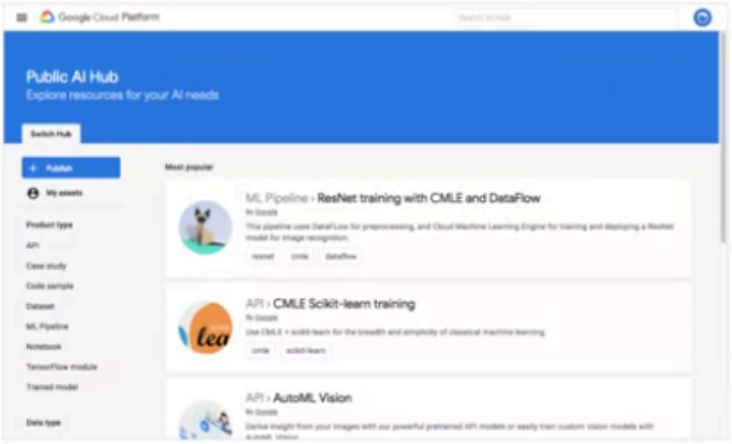


3 Ecosystem

AI Hub at a glance

1. All AI content in one place
   - Quick discovery of plug and play AI pipelines and other content build by teams across Google and by partners and customers

2. Fast and simple implementation of AI on Google Cloud
   - One-click deployment of AI pipelines via Kubeflow on Google Cloud



3. Enterprise-grade internal & external sharing
   - Foster reuse by sharing deployable AI pipelines and other content privately within organizations and publicly

What is an ML pipeline?

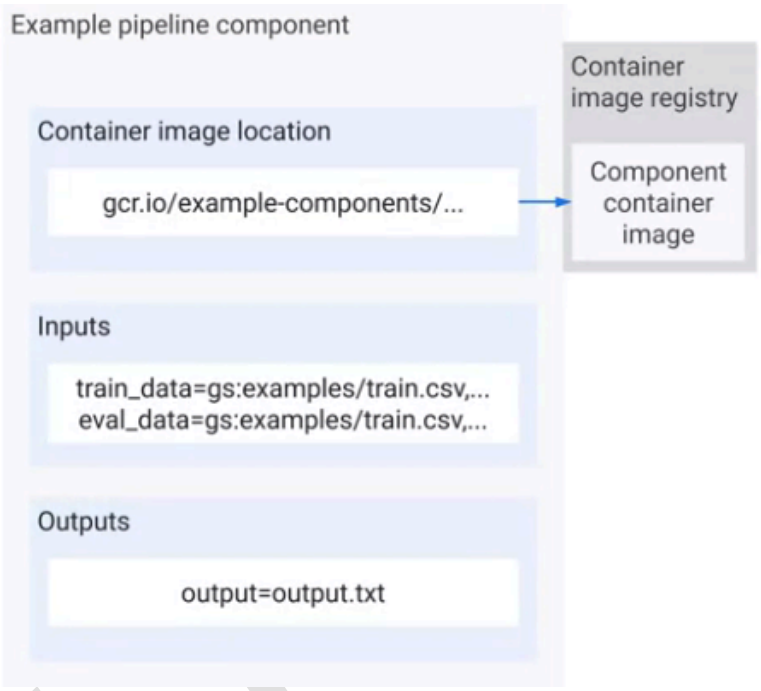
ML pipelines are portable, scalable ML workflows based on containers.

You can use ML pipelines to:

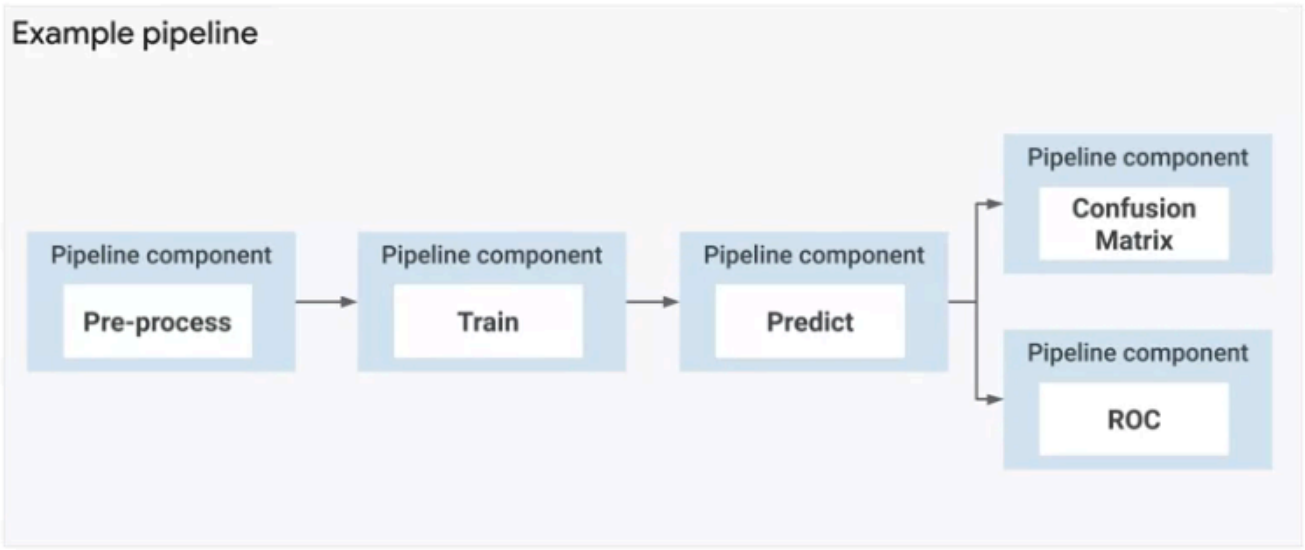
- Apply MLOps strategies to automate repeatable processes.
- Experiment by running an ML workflow with different sets of hyperparameters, number of training steps or iterations, etc.
- Reuse a pipeline's workflow to train a new model.

Understanding pipeline components

- Self-contained sets of code to perform various operations
- Composed of set of input/output parameters and location of container image
- Container image: Package that includes the component's executable code and a definition of the environment that the code runs in



Understanding a pipeline workflow



*Receiver Operating Curve (ROC)

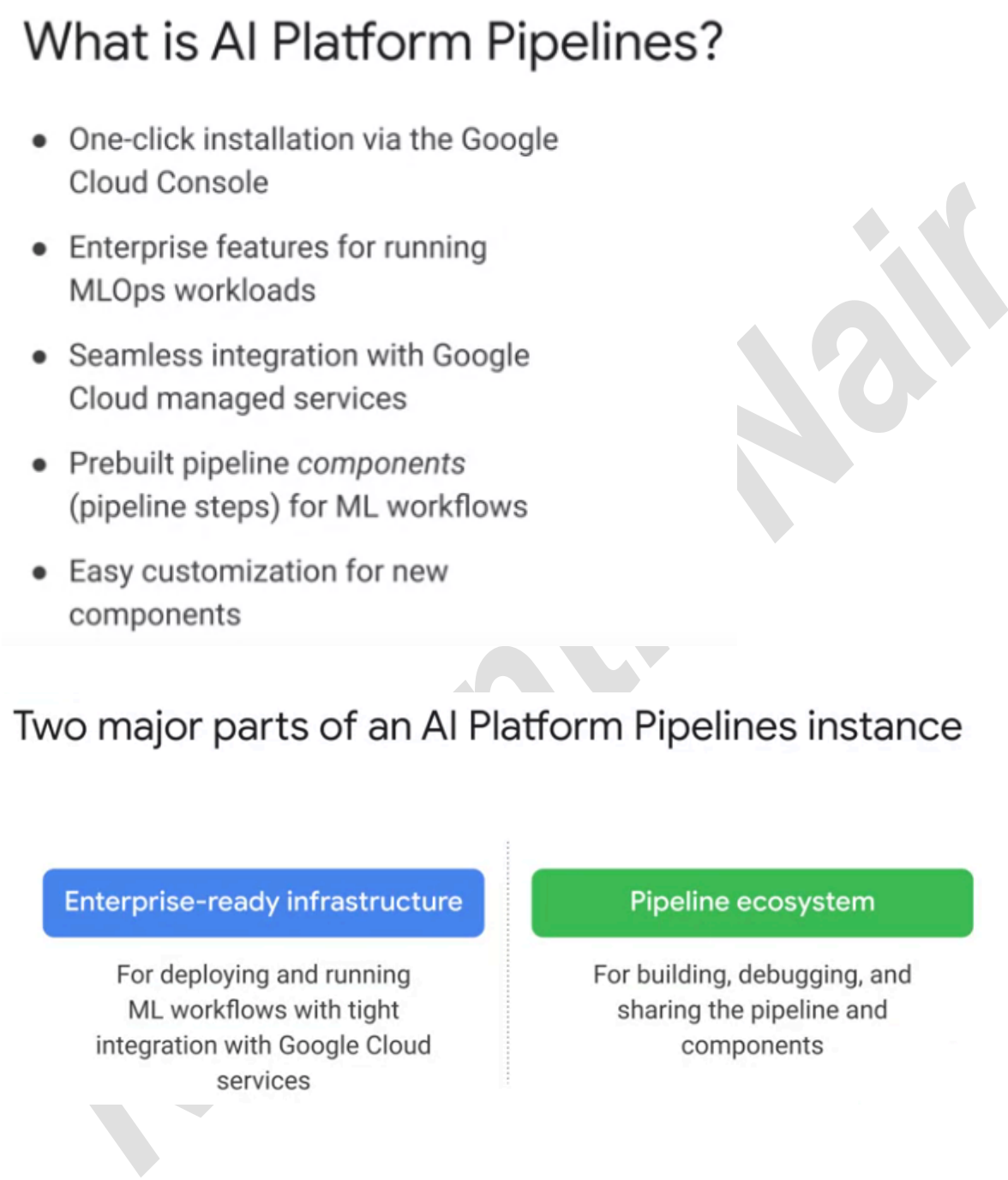
Current process for building an MLOps pipeline

- Set up a Google Kubernetes Engine (GKE) cluster.
- Create a Cloud Storage bucket for storing data.
- Install Kubeflow pipelines.
- Set up port forwarding.
- Create a process to share the pipeline with your team.

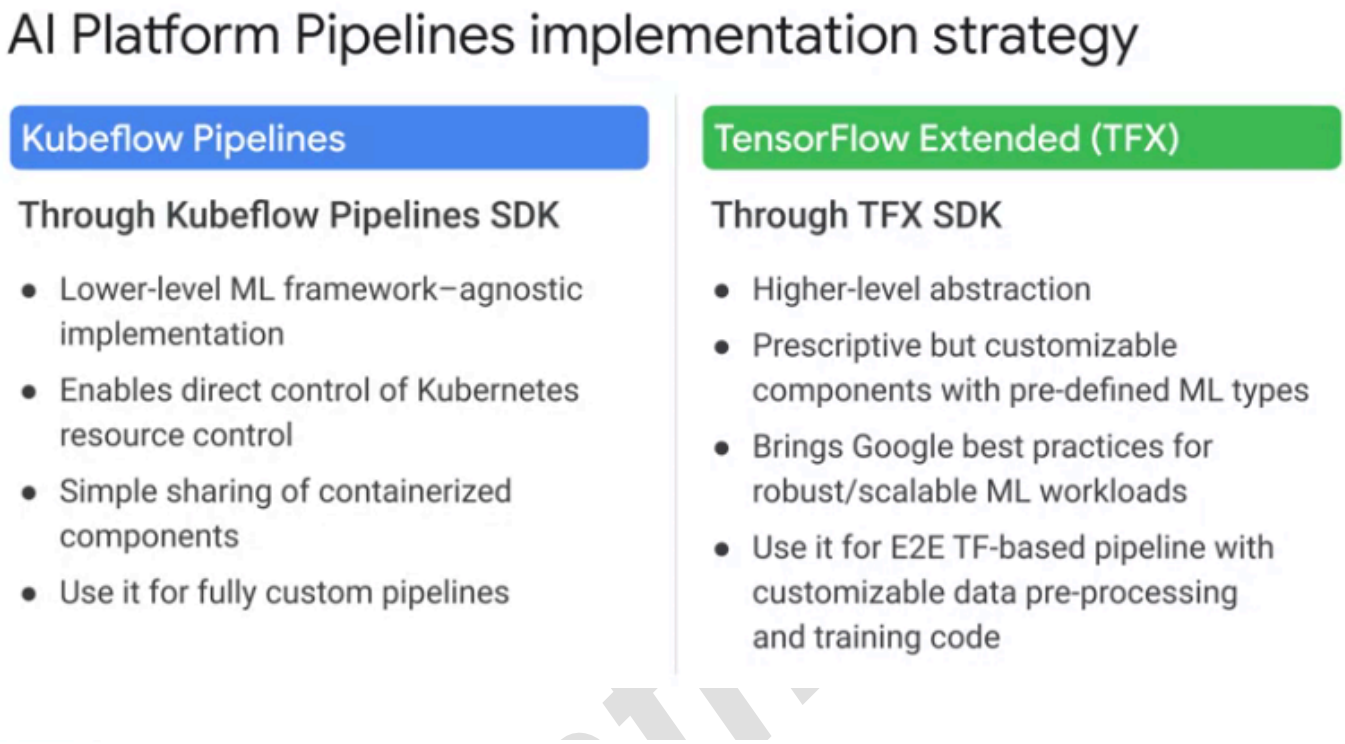
Can we automate these processes to make MLOps a seamless and easy experience?

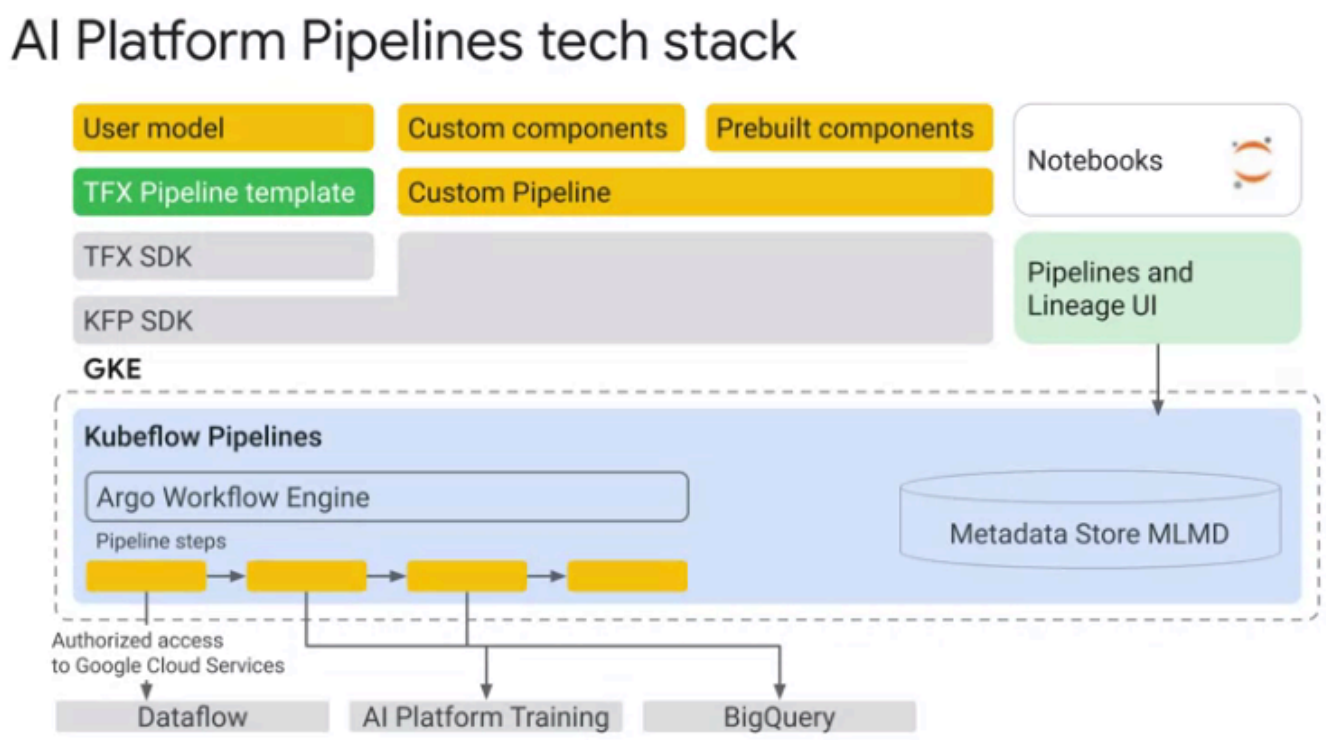
A new product was needed to deploy robust, repeatable machine learning pipelines along with monitoring, auditing, version tracking, and reproducibility and deliver an enterprise-ready, easy-to-install, secure execution environment for your ML workflows.

This resulted in AI Platform Pipelines



3.1.1 Concepts





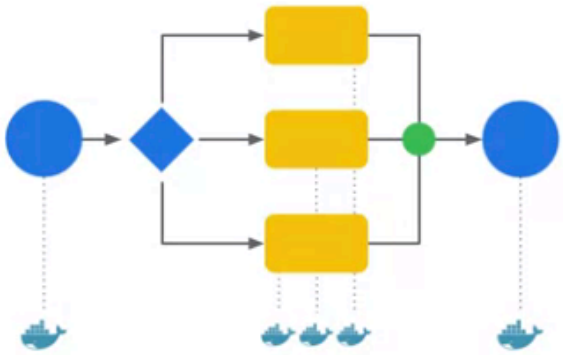
- Features of AI Pipelines
    - TFX templates
    - Versioning
    - Lineage tracking
    - Artifact Tracking

3.1.2 When to use Pipelines

- **Workflow Orchestration**

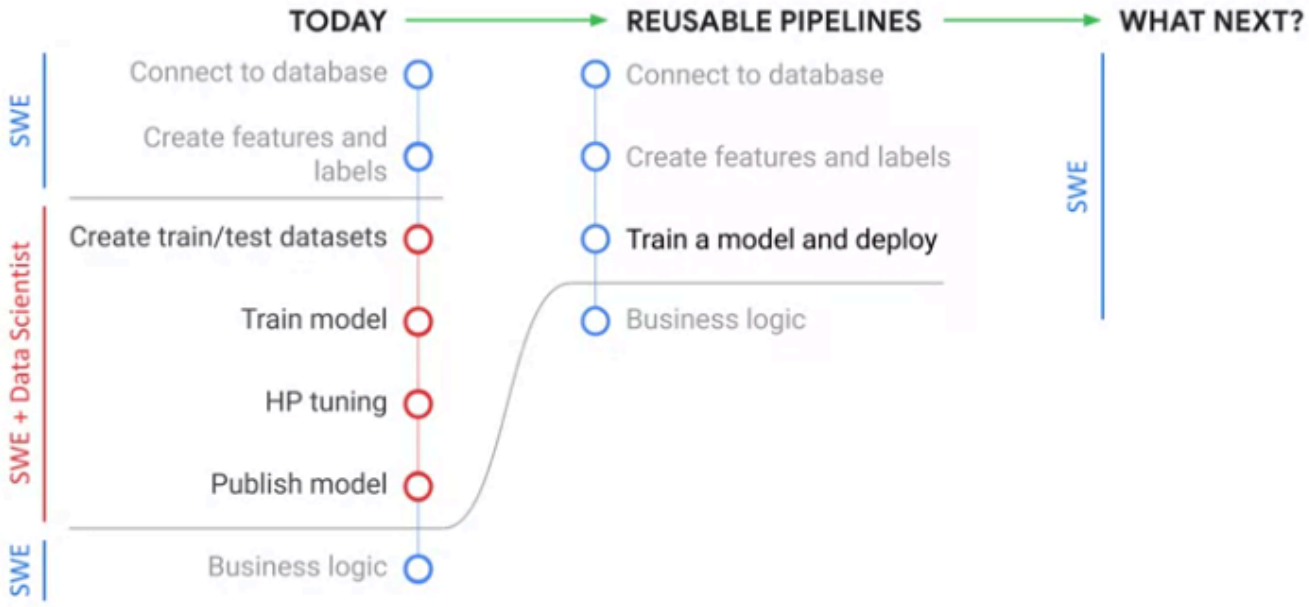
What constitutes an AI Platform Pipelines instance?

- Containerized implementations of ML tasks
    - Containers provide portability, repeatability, and encapsulation.
    - A task can be single node or *distributed*.
    - A containerized task can invoke other services, like CAIP, Dataflow, or Dataproc.
- Specification of the sequence of steps
    - Specified via Python SDK
- Input parameters
    - A "job" = A pipeline invoked with specific parameters



- Rapid and reliable Experimentation
- Share, Re-use and compose

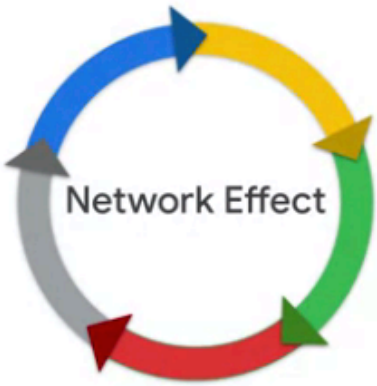
Reusable pipelines: Force multiplier for data scientists



AI Hub and Pipelines: Fast and simple adoption of AI

The flywheel of AI adoption

1. SEARCH and DISCOVER: Find best-of-breed pipelines on the hub that leverage Cloud AI solutions (AutoML, GPU, TPU, CMLE, etc.).

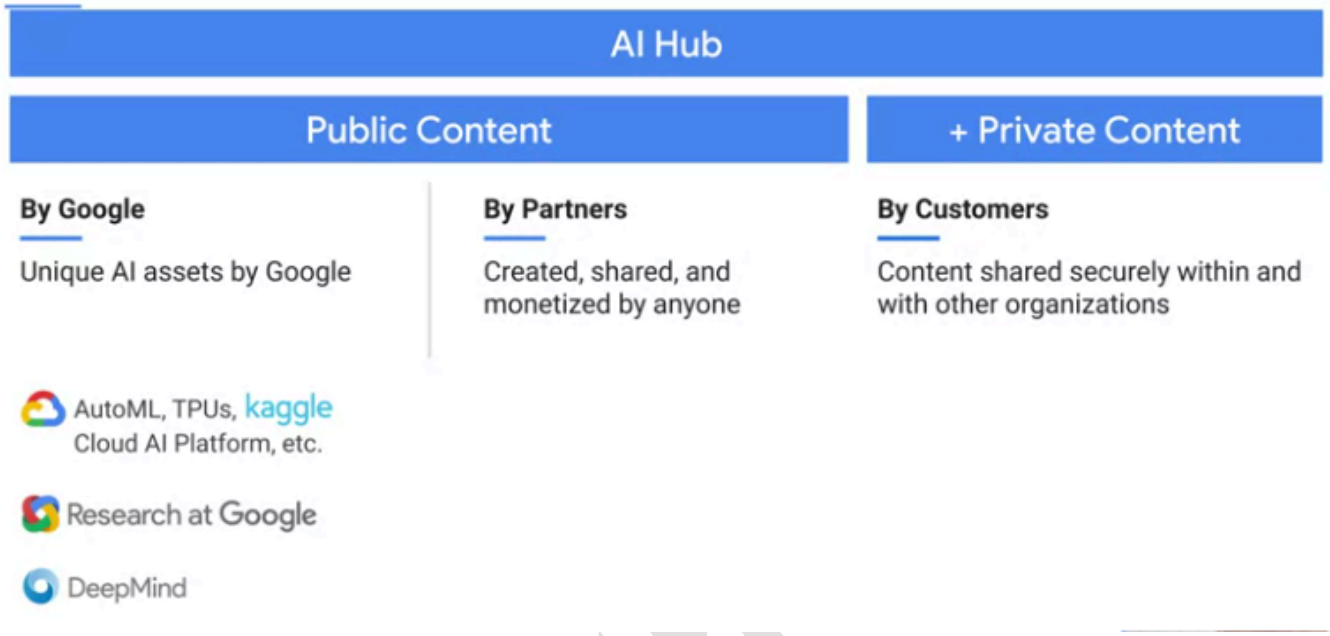


2. DEPLOY: Quick 1-click implementation of ML pipelines onto Google Cloud/GKE.

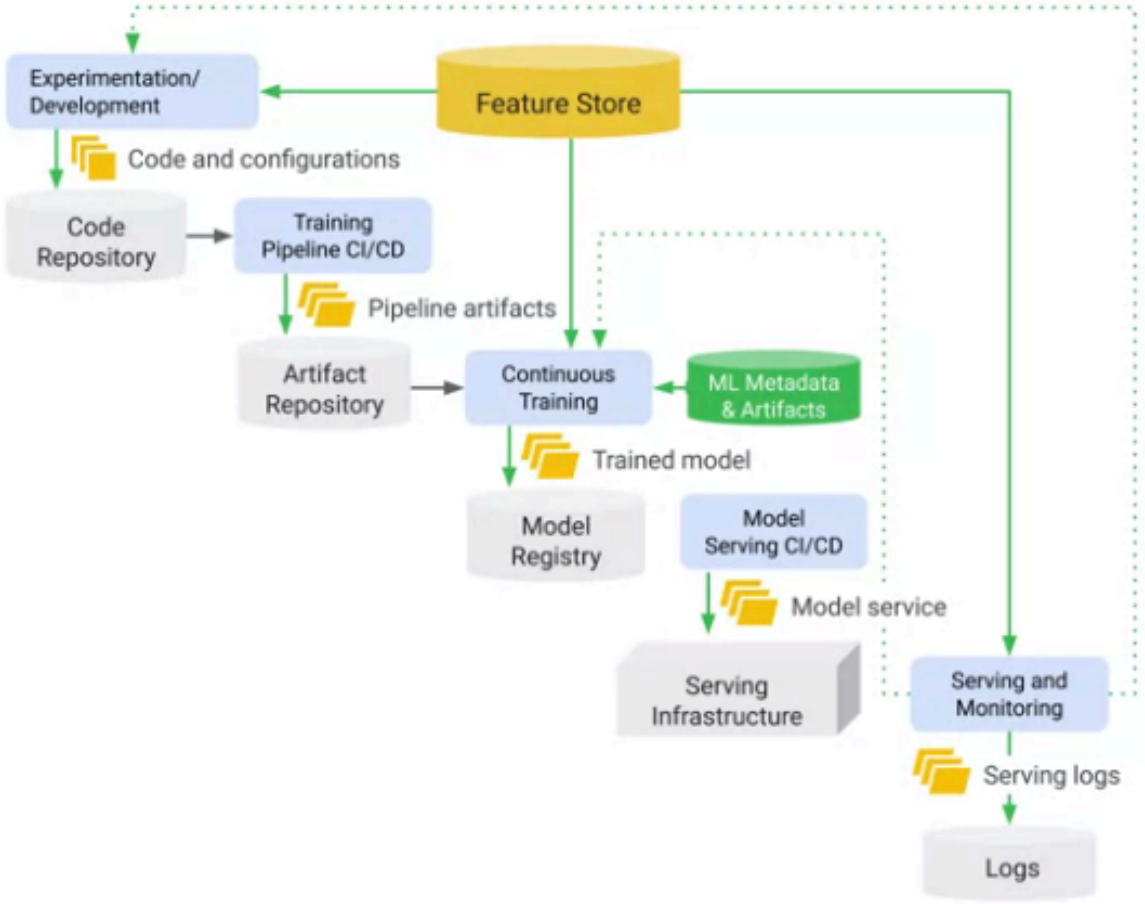
3. CUSTOMIZE: Experiment and adjust out-of-the-box pipelines to custom use cases via pipelines UI.

4. RUN IN PRODUCTION: Deploy customized pipelines in production on Google Cloud.

5. PUBLISH: Upload and share pipelines that run best on Google Cloud with your org or publicly.

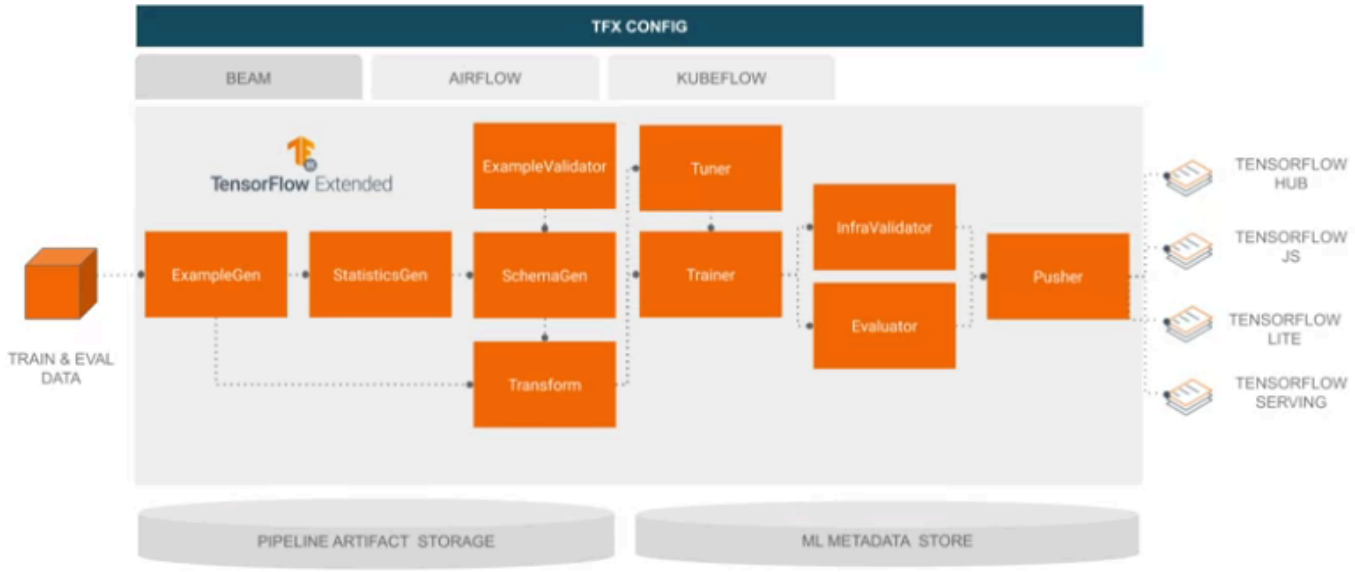

AI Hub
Public Content
+ Private Content
By Google
Unique AI assets by Google
By Partners
Created, shared, and monetized by anyone
By Customers
Content shared securely within and with other organizations
AutoML, TPUs, kaggle
Cloud AI Platform, etc.
Research at Google
DeepMind

3.1.3 Another view of the Continuous training Pipeline



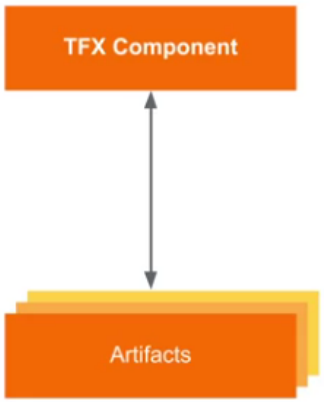
3.2 Tensorflow Extended (TFX) Pipelines

- TFX is a Google production-scale machine learning platform based on the TensorFlow ecosystem



3.2.1 TFX Concepts

- **TFX Component:** A TFX component is an implementation of the machine learning task in your pipeline.
- **TFX Artifact:** Each step of your TFX pipeline, again called a component, produces and consumes structured data representations called artifacts.

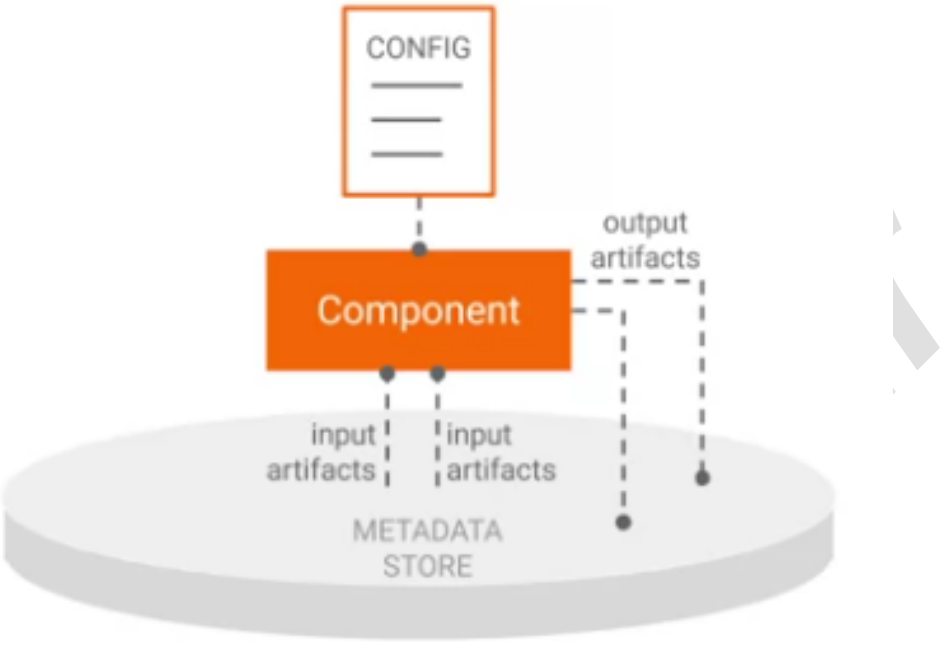


- Subsequent components in your workflow may use these artifacts as inputs.
- In this way, TFX lets you transfer data between components during the continuous execution of your pipeline.

- A component is composed of 5 elements:

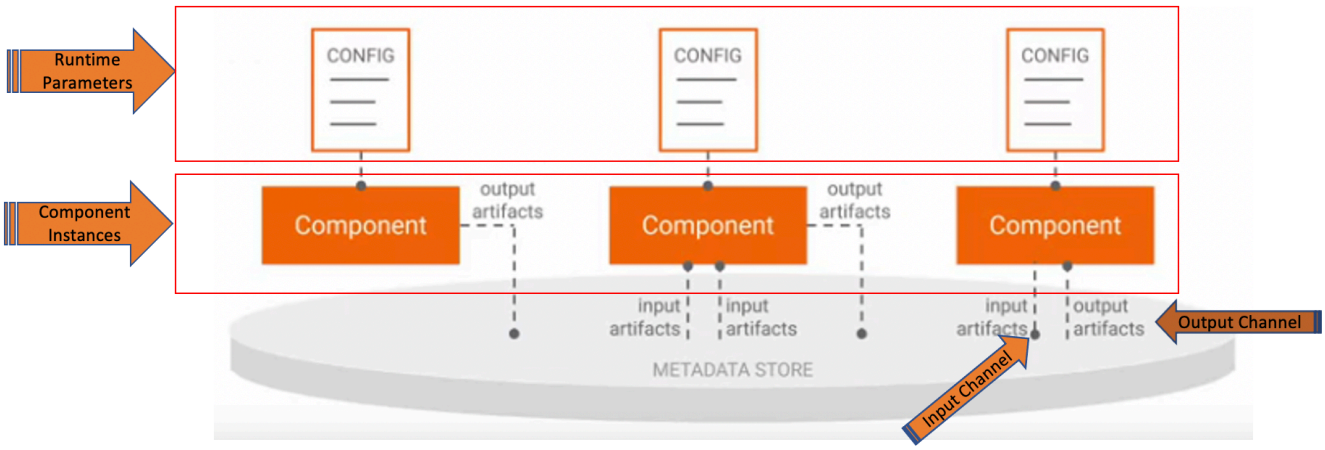
  - **Component Interface**
    - This packages the component specification, driver, executer, and publisher together as a component for use in a pipeline.
  - **Component specification**
    - The components specs define how components communicate with each other.
    - They described three important details of each component:
      - Input artifacts,
      - Output artifacts,
      - Runtime parameters
    - These are required during component execution.
    - Component specs are implemented as ***protocol buffers***, which are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data similar to XML or JSON, but smaller, faster, and simpler.
  - **Component Driver**
    - Coordinates job execution
    - Examples include reading artifact locations from the ML metadata store, retrieving artifacts from pipeline storage, and the primary executor job for transforming artifacts.
  - **Component executor**
    - Code to perform ML workflow step such as data preprocessing, model training etc.
  - **Component Publisher**
    - Updates the ML metadata store.

3.2.1.1 TFX Components Runtime



- **First**, a driver reads the component specification for runtime parameters and retrieve the required artifacts from the ML metadata store for the component.
- **Second**, an executor performs the actual computation on the retrieved input artifacts and generates the output artifacts.
- **Finally**, the publisher reads the components specification to log the pipeline component run and ML metadata and write the components output artifacts to the artifacts store.

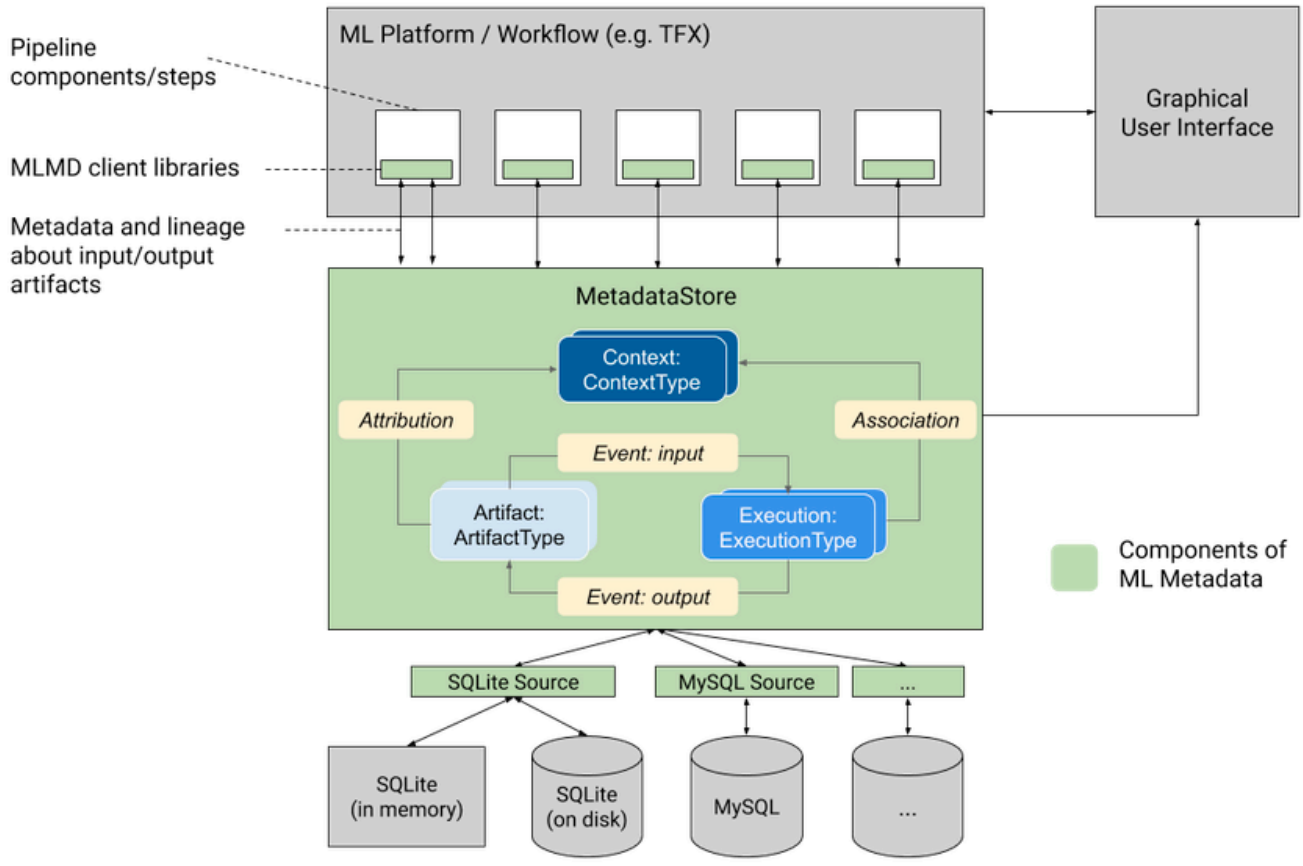
3.2.1.2 TFX Pipeline runtime



- A TFX pipeline is a sequence of ***components*** connected by ***channels*** in a directed acyclic graph (DAG) of ***artifact*** dependencies.
- They communicate through input and output channels.
- A **TFX channel** is an abstract concept that connects component data producers and data consumers.
- ***Component instances*** produce artifacts as outputs and typically depend on artifacts produced by upstream component instances as inputs.
- **Parameters** are inputs to pipelines that are known before your pipeline is executed.
    - Parameters let you change the behavior of a pipeline or part of a pipeline through configuration protocol buffers instead of changing your components and pipeline code.
    - TFX pipeline parameters are a great abstraction that allows you to increase your pipeline experimentation velocity by running your pipeline fully or partially with different sets of parameters, such as training steps, data split spans, or tuning trials without changing your pipelines code every time.

3.2.1.3 ML Metadata Store

- TFX implements a metadata store using the ***ML Metadata (MLMD)*** library, which is an open-source library to standardize the definition, storage, and querying metadata for ML pipelines.
- MLMD registers the following types of metadata in a database called the **Metadata Store**.
  - Metadata about the artifacts generated through the components/steps of your ML pipelines
  - Metadata about the executions of these components/steps
  - Metadata about pipelines and associated lineage information

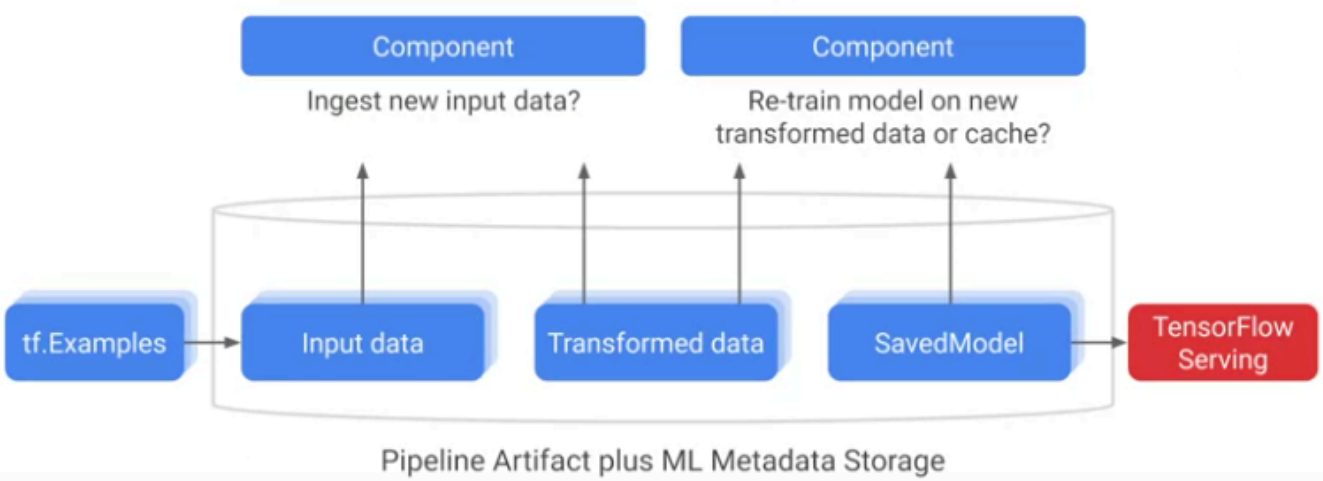


- This metadata can help you answer questions about your ML pipeline such as:
  - Which dataset did the model train on?
  - What were the hyperparameters used to train the model?
  - Which pipeline run created the model?
  - Which training run led to this model?
  - Which version of TensorFlow created this model?
  - When was the failed model pushed?
- The ml metadata libraries store the metadata in a relational backend.
  - For notebook prototypes, this can be a local SQL database
  - For production Cloud deployments this could be a managed MySQL or Postgres database.
  - **NOTE**:
    - ML metadata does not store the actual pipeline artifacts.
    - TFX automatically organizes and stores the ***artifacts*** on a local file system on a remote Cloud storage file system for consistent organization across your machine learning projects.

3.2.1.4 Orchestrator

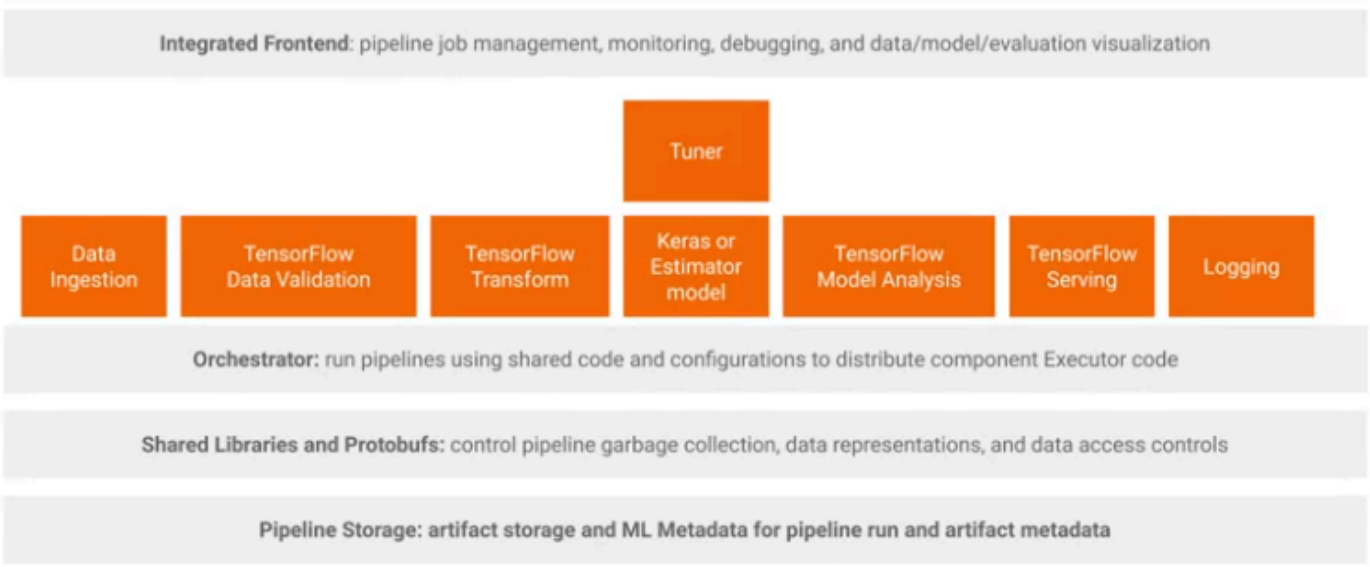
- An orchestrator is responsible for gluing the components together.
- Orchestrators coordinate pipeline runs specifically component executers sequentially from a directed graph of artifact dependencies.
- Orchestrators ensure consistency with pipeline execution order, component logging, retries and failure recovery, and intelligent parallelization of component data processing.
- TFX supports orchestrators such as: Apache Airflow, Apache Beam, and Kubeflow Pipelines.
  - TFX supports ***Apache Airflow*** to **author** workflows as directed acyclic graphs (DAGs) of tasks.
  - Several TFX components rely on ***Apache Beam*** for distributed data processing.
  - In addition, TFX can use ***Apache Beam*** to orchestrate and **execute** the pipeline DAG.
- TFX also uses the term ***DagRunner*** to refer to an implementation that supports an orchestrator.

3.2.1.5 Task and Data-Aware pipelines



- A key innovation of TFX pipelines is that they're both task and data-aware pipelines.
- A **task** can be an entire pipeline run or a partial pipeline run of an individual component and its downstream components.
- **Task aware** means that they can be authored in a script or a notebook to run manually by the user as a task.
- **Data-aware** means TFX pipelines store all the artifacts from every component over many executions.
    - So, they can schedule component runs based on whether artifacts have changed from previous runs.
- The implications of this are as follows:
    - The pipeline automatically checks whether re-computation of artifacts, such as large data ingestion and transformation tests is necessary when scheduling component runs.
    - This dramatically speeds up your model retraining and tuning velocity in a continuous training pipeline resulting in significant pipeline speed ups and compute resource efficiencies.

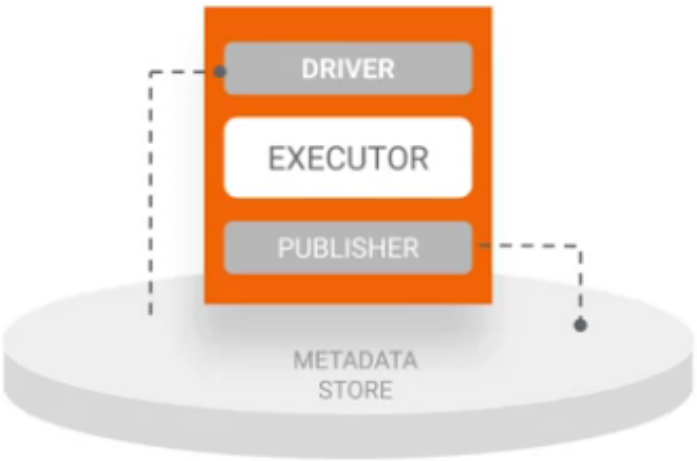
3.2.1.6 TFX Horizontal Layers



- TFX horizontal layers coordinate pipeline components.
- These are primarily shared libraries of utilities and protobufs for defining abstractions that simplify the development of
- TFX pipelines across different computing and orchestration environments.
- **Integrated Frontend Layer**
    - For most machine learning cases with TFX, you will only interact with the integrated front-end layer and don't need to engage directly with the orchestrator, shared libraries, and ML metadata unless you need additional customization.
    - The integrated frontend enables GUI-based controls over pipeline job management, monitoring, debugging, and visualization of pipeline data models and evaluations.
- **Orchestrator**
    - Orchestrators come integrated with their own frontends for visualizing pipeline-directed graphs.
    - Orchestrators run TFX pipelines with shared pipeline configuration code.
    - All orchestrators inherit from a TFX runner class.
    - TFX orchestrators take the logical pipeline object, which can contain pipeline args, components and a DAG, and are responsible for scheduling components of the TFX Pipelines sequentially based on the artifact dependencies defined by the DAG.
- **Shared Libraries and Protobufs**
    - These create additional abstractions to control pipeline garbage collection, data representations, and data access controls.

    - An example framework is the TFXIO library, which defines a common in-memory data representation shared by all TFX libraries and components in an I/O abstraction layer to produce such representations based on Apache Airflow.
- **Pipeline Storage**
    - MLMD records pipeline execution metadata and artifact path locations to share across components.
    - You also have pipeline artifacts Storage which automatically organizes artifacts on local or remote Cloud file systems

3.2.1.7 TFX Standard Components



- TFX standard components come pre-packaged with TensorFlow and are designed to help improve your pipeline development velocity.
- You can mix together standard and custom components to suit your machine learning workflow needs.
- Each standard component is designed around common machine learning tasks and encodes Google's ML best practices around tasks such as data monitoring and validation, training and serving data transformations, model evaluation, and serving.

3.2.1.7.1 TFX Standard Data Components

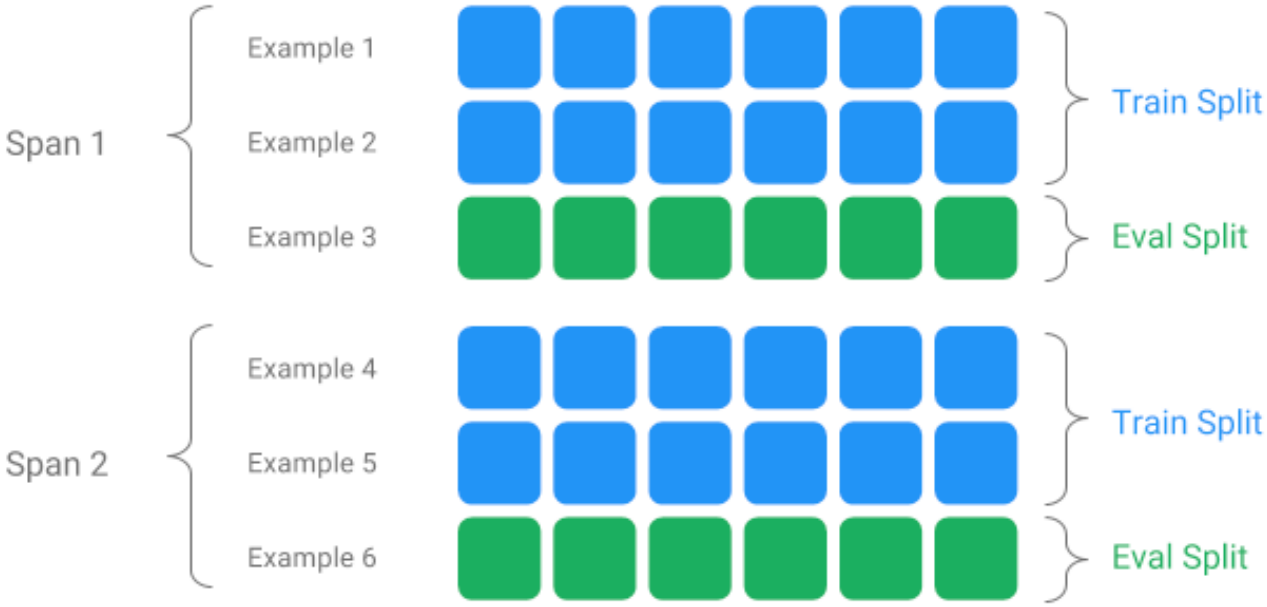
- **ExampleGen (Reference: https://www.tensorflow.org/tfx/guide/examplegen)**

  - The ExampleGen TFX pipeline component is the entry point to your pipeline, that ingests data.
  - As **inputs**, ExampleGen supports out-of-the-box ingestion of external data sources such as CSV, TF Records, Avro, and Parquet.
  - As **outputs**, ExampleGen produces TF examples, or TF sequence examples which are highly efficient in performant data set representations that can be read consistently by downstream components.
  - ExampleGen provides several benefits as part of your Machine Learning project life cycle.

ML project lifecycle benefits

- Configurable and reproducible data partitioning and shuffling
- External CSV, Avro, Parquet, BigQuery, and TFRecord ingestion
- Apache Beam supported for scalable data ingestion
- Customizable to new input data formats and ingestion methods

- ExampleGen supports advanced data management capabilities such as data partitioning, versioning, and custom splitting on features, or time.
- TFX organizes data using several abstractions with a hierarchy of concepts named
  - Spans
  - versions and
  - splits



- A **Span** is a grouping of training examples.
  - If your data is persisted on a filesystem, each Span may be stored in a separate directory.
  - The semantics of a Span are not hardcoded into TFX.
  - A Span may correspond to a day of data, an hour of data, or any other grouping that is meaningful to your task.

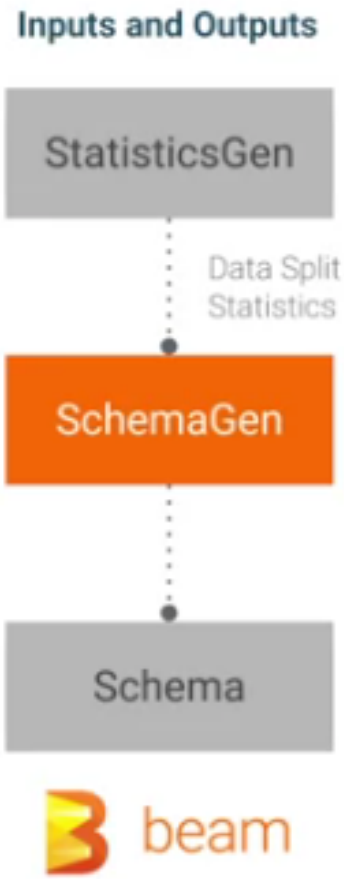
        - Each Span can hold multiple **Versions** of data.
            - To give an example, if you remove some examples from a Span to clean up poor quality data, this could result in a new Version of that Span.
            - By default, TFX components operate on the latest Version within a Span.
        - Each Version within a Span can further be subdivided into multiple **Splits**.
            - The most common use-case for splitting a Span is to split it into training and eval data.

- **StatisticsGen** (Refer: https://www.tensorflow.org/tfx/guide/statsgen)

    - As **inputs**, this StatisticsGen component is configured to take in TF examples from the ExampleGen component.
    - As **outputs**, StatisticsGen produces a data set statistics artifact that contains features statistics used for downstream components.
    - StatisticsGen performs a complete pass over your data using **Apache Beam** and the ***TensorFlow Data Validation Library*** to calculate summary statistics for each of your features over your configured Train, Dev and Test splits.

        - This includes statistics like mean, standard deviation, quantile ranges, and the prevalence of null values.
    - Benefits of using StatisticsGen

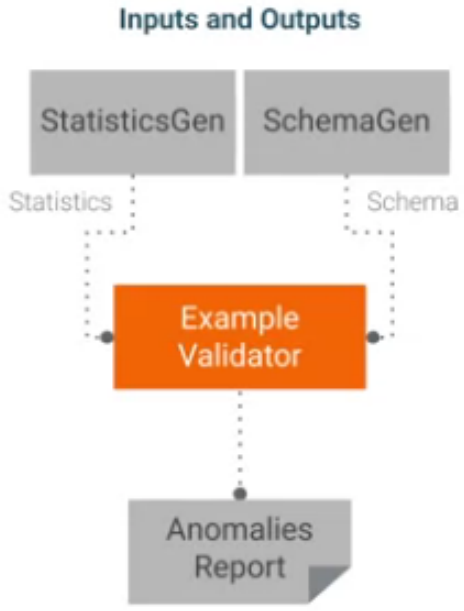
- **SchemaGen** (refer: https://www.tensorflow.org/tfx/guide/schemagen)



    - Some TFX components use a description of your input data called a ***schema*** that can be automatically generated by the SchemaGen component.

        - The schema is an instance of schema.proto.
        - Schemas are a type of protocol buffer, more generally known as a protobuf.
        - The schema can specify datatypes for feature values whether a feature has to be present in all examples, what are the allowed value ranges for each feature, and other properties.
    - As **input**, SchemaGen reads the StatisticsGen artifact to infer characteristics of your input data from the observed feature data distributions.
    - As **output**, SchemaGen produces a schema artifact, a schema.proto description of your data's characteristics.
    - The SchemaGen, is an optional component that does not need to be run on every pipeline run, and instead it is typically run during prototyping and the initial pipeline run.
    - One of the benefits of using TensorFlow Data Validation is that the SchemaGen component can be configured to automatically generate a schema by inferring types, categories in accepted ranges from their training data.
    - A key benefit of a schema file is
        - You can reflect your expectations of your data in a common data description for use by all of your pipeline components.
        - It also enables continuous monitoring of data quality during continuous training of your pipeline.

- **ExampleValidator** (Refer: https://www.tensorflow.org/tfx/guide/exampleval)



- The ExampleValidator pipeline component identifies anomalies in trained dev and test data.
    - It does this by comparing data statistics computed by the StatisticsGen pipeline component against a schema.
- As **inputs** ExampleValidator reads feature statistics from StatisticsGen and the schema artifact produced by the SchemaGen component, or imported from an external source.
- As **outputs**, ExampleValidator outputs an ***anomalies report artifact***.
- The key benefit of ExampleValidator is that it can detect different classes of anomalies in the data.
    - It can perform validity checks by comparing dataset statistics against a schema.
    - It can detect feature train serving skew by comparing training and serving data.
    - It can also detect data drift by looking at a series of feature data across different data splits.

- **Transform** (Refer: https://www.tensorflow.org/tfx/guide/transform)

- When applying machine learning to real-world data sets, a lot of effort is required to preprocess your data into a suitable format.
    - This includes converting between formats, tokenizing and stemming texts, forming vocabularies, performing a variety of numerical operations such as normalization.
    - You can do all of this with the TF Transform library, which underpins the Transform TFX component.
- The Transform TFX pipeline component performs feature engineering on:
    - TF examples data artifact emitted from the ExampleGen component using the data schema artifact from the SchemaGen or imported from external sources
    - As well as TensorFlow transformations defined in a custom preprocessing function.
        - In your transform code, you can define feature transformations with TF Transform in TensorFlow operations.
        - Some of the most common transformations include, converting sparse categorical features into dense embedding features, automatically generating vocabularies, normalizing continuous numerical features, bucketizing continuous numerical features into discrete categorical features, as well as enriching text features.
- As **outputs**, the transform component emits:
    - A ***saved model artifact*** and it encapsulates feature engineering logic.
    - A ***transform data artifact*** that is directly used by your model during training.
- Some of the benefits include:
    - The transform component brings consistent feature engineering at training and serving time to benefit your machine learning project.
    - By including feature engineering directly into your model graph you can reduce train-serving skew from differences in feature engineering, which is one of the largest sources of error in production machine learning systems.
    - Transform, like many other TFX components is also underpinned by Apache Beam, so you can scale up your feature transformations using distributed compute as your data grows

3.2.1.7.2 TFX Standard Model Components

3.2.1.8TFX Pipeline Nodes

3.2.2 TFX Libraries

3.3 Kubeflow

Kubeflow provides a standardized platform for building ML pipelines

- Leverage containers and Kubernetes so that in ML pipelines can be run on a cloud or on-premises with Anthos on GKE.
- Kubeflow is a cloud-native, multi-cloud solution for ML.
- Kubeflow provides a platform for composable, portable, and scalable ML pipelines.
- If you have a Kubernetes-conformant cluster, you can run Kubeflow.

3.3.1 Pipeline using Python

Kubeflow offers a Domain Specific Language (DSL) in Python that allows you to use Python code to describe Kubeflow tasks as they organize themselves in a Directed Acyclic Graph (DAG).

3 main types of Kubeflow components we will look at

01 Pre-built components
- Just load the component from its description and compose.

02 Lightweight Python components
- Implement the component code.

03 Custom components
- Implement the component code.
- Package it into a Docker container.
- Write the component description.