devoteam G Cloud

Google Cloud

January 2023

The ML on Google Cloud Workbook
by Devoteam G Cloud

Authors: Albichari Kais, Maréchal Cyril, Massimetti Giulia, Subramaniam Branavan & Van Thielen Tristan

Creative tech for Better Change

Contents

1 Introduction

Machine Learning has been a big driver of business value for companies that have been successful in its adoption. Across industries, the key challenge is the process towards a successful adoption. Despite growing demand and democratization of the tools and techniques available, many companies still face significant challenges in getting models out of their local environments and deploying them into a robust production system.

The goal of this white paper is to cover all topics to consider when developing and deploying production machine learning systems on Google Cloud Platform (GCP). To remain succinct, it will not dive into the details of data warehousing on GCP as this is a big topic on its own.

The paper will start with a brief introduction to Google Cloud as it is fundamental to understanding later sections. Afterwards, it will cover the basics of MLOps while being technology agnostic. The next section will cover the different ML services available on Google Cloud more in detail and the last section will then explain how those services can be used to implement the MLOps principles described.

2 The basics of Google Cloud

This section aims to provide a basic introduction to Google Cloud and the different concepts one needs to understand in order to work with it properly.

2.1 The GCP Resource Hierarchy

GCP resources are organized in a strict resource hierarchy.

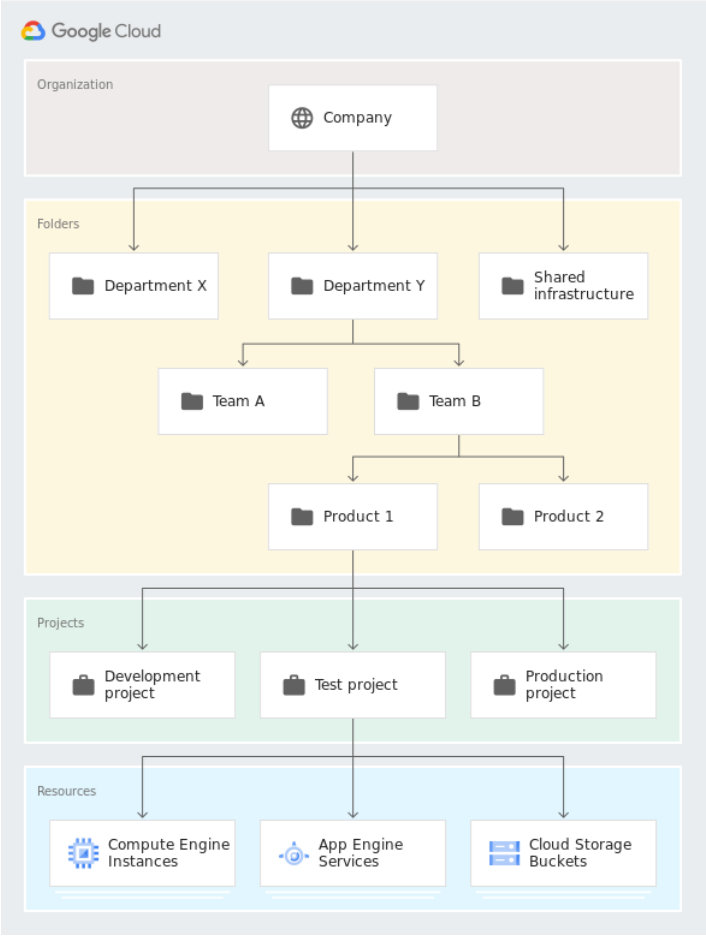

Figure 1: The GCP resource hierarchy. [2]

Projects are the core of this hierarchy, as every resource must belong to a project in order to be used. This means that you can not use GCP without creating at least one project.

Projects themselves can be organized further by using folders and organizations. An organization is most easily understood as one company, which has its own space within Google Cloud. It is the highest level on which access can be granted. Not all projects must belong to an organization, but for most companies it is highly advised.

Within an organization, projects can belong to one folder. There can be multiple projects per folder and in turn multiple folders per organization. This hierarchy can be leveraged to set up permission inheritance. Users with a specific permission on a specific folder, by default have that permission on the projects and child-folders with that folder.

2.2 Identity and Access Management

Identity and Access Management (IAM) governs who can take what action on what resource. GCP uses a principle called Role Based Access Control (RBAC) to manage access. There are some important concepts to understand here.

- An **IAM member** is someone who wants to perform an action on a resource. This can be a user (optionally belonging to a group) or a service account (a form of identity mostly used to identify specific applications).
- A **permission** represents the permission to perform a specific action on a resource. An example could be “create a virtual machine instance", which would be represented by the “compute.instances.create" permission.
- A **role** is a combination of permissions. For example the Compute Admin role will give you the permissions to create virtual machines, networks, etc. Roles have an identifier that looks like “roles/compute.admin".
- A **role binding** is an assignment of a **role** to a specific **IAM member**, which means that that IAM member gets the permissions granted by that role. Roles can be assigned to users, service accounts and groups. The recommendation is always to work with groups where possible as individual users might leave the company, change teams, etc.
- A **policy** describes a series of role bindings. Policies can be defined at resource level, project level, folder level or organization level, depending on the type of resource. By default, GCP permissions are inherited down through the resource hierarchy. An organization level Viewer role will grant the user permission to view all projects in the organization. Although there are some

ways to avoid the inheritance in specific cases, this is the default behavior of a policy.

2.3 Services

Google Cloud resources are grouped logically by services. A service is a specific API that can be used to manage resources within GCP. An example is the Compute service which is linked to *compute.googleapis.com* or Vertex AI linked to *aiplatform.googleapis.com*. In order to use resources within a service, that service must be enabled on the GCP project.

2.4 Locations, regions and zones

Services are consumed in specific locations. Which locations are available depends on the service being consumed. This can be an important consideration when it comes to local regulations (GDPR, HIPAA, ...), but also has an impact on price.

The most general location is “global", the global location indicates that there is no guarantee on where the service will be consumed. However, most services can be configured to be used in a specific “region". A region is a geographic location, for instance Europe West or US East. These are large areas comprising of multiple “zones", where each zone corresponds to a datacenter in a specific location. Some services can also be used in a “multi-region" configuration where multiple regions are combined into one. Examples of multi-region configurations are US and EU.

3 The basics of MLOps

Machine learning operations (or MLOps) is a key aspect of ML engineering which aims to simplify AI lifecycle from the beginning to its end. It is built on the concept of DevOps to add pieces that are specific to its domain. DevOps combines every tasks of a software system to provide a high quality and shortened development lifecycle. It then enables teams to deploy new features and finish projects faster. MLOps aims to apply those principles to Machine Learning, although in this case we have new ML-specific components to consider. The goal is to enable continuous development and deployment of models and to ensure proper monitoring and management. This should happen fast and secure in order to eliminate toil from the daily job of Machine Learning Engineers and Data Scientists.

3.1 Towards a reusable deployment pipeline

At the core of MLOps is the deployment pipeline. Without it, ML engineers would have to manually deploy models every time, which is error prone and time intensive. Instead, models should be deployed automatically based on the code and available data. However, many things should be considered when setting up an MLOps pipeline. This will be covered in the following sections.

3.1.1 Pipeline components

In essence, a pipeline is a series of steps or components that are executed in a specific order and which pass inputs and outputs to each other. In most cases, components are Docker containers that execute a specific script. Each component should have a clear task that it performs, for instance transforming input data, training a model, deploying a model, etc.

Output produced by components is often referred to as artifacts. These should be tracked across different pipeline executions so it can be traced back what artifacts were used as input for a model running in production.

3.1.2 Pipelines

A pipeline is a series of multiple components chained together to automate a set of tasks. In ML related terms, this most often resembles the following example:

- Fetch training data
- Split into train, validation and test set
- Perform feature engineering on training data
- Train model
- Evaluate model
- Deploy model if evaluation is good

All of this is fully automated and any output from each of these steps is tracked in a Metadata Store. Of course pipelines can differ from this example, but they will generally follow this structure.

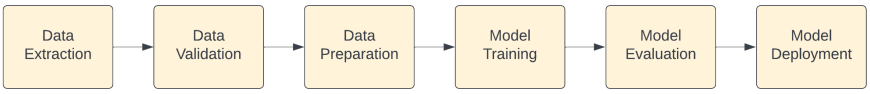

Figure 2: A simple example pipeline

One important distinction to be made is the one between a **pipeline template** and a **pipeline execution**. To enable re-usability, a pipeline should be defined in a pipeline template, which takes specific inputs and produces specific outputs. This should make it easy to run the same pipeline on different datasets and with different parameters. A pipeline execution is one run of the pipeline template with specific input parameters. A pipeline template is most often defined in a programming language like Python (which is then rendered into a static file) or a static file format like YAML or JSON.

3.1.3 Version Control System

Any team working on code together benefits from having it stored in a Version Control System (VCS). Going into the details of a VCS is out of scope here, but it plays a crucial role in proper MLOps practices.

Code for feature engineering, model training, etc. should all be versioned to enable easy rollbacks etc. One tool should get specific attention here and that is the Jupyter Notebook. Jupyter Notebooks are very popular among data scientist for doing quick experimentation and data exploration. This is a useful feature to have, however Jupyter Notebooks have an important pitfall which is that they do not encourage modularization of code. This makes them unfit for any code that makes it into production.

It is therefore recommended to move any code that needs to go into the deployment pipeline into proper Python (or other programming language) files following coding best practices. These files then go into the Version Control System where they will trigger the building of artifacts.

3.1.4 Building and storing artifacts

The build artifacts in this section are different from the pipeline artifacts described above. Build artifacts are objects generated from a specific version of the code stored in the VCS. Traditionally this is done through a CI/CD pipeline which takes the code defining the pipeline templates for example and pushes static files representing the template to an artifact registry.

The CI/CD pipeline starts whenever a new version of the code is pushed to the VCS and generally includes the following steps:

- **Test code**: it is important to perform testing on any code that will be pushed into a production system. There are multiple different types of tests:
  - **Unit tests**: these tests test a small part of the code like one function with a specific input
  - **Functional tests**: check that one application is behaving as expected (e.g. the model hosting service)
  - **Integration tests**: test that an entire integrated system is behaving as expected (e.g. the application stack interacting with the model service)
- **Build artifacts**: static artifacts are generated from the code. For MLOps systems, this mostly corresponds to:
  - **Components**: build Docker containers or files describing component behavior
  - **Pipeline templates**: build pipeline templates from code
- **Push artifacts**: after artifacts have been built, they are pushed to a registry. This is where they are stored with a specific tag and can then be fetched at runtime to perform one pipeline execution.

Figure 3: A simple CI/CD pipeline

3.1.5 An end to end example

Chaining CI/CD and the training pipeline together provides a full end to end workflow. This is where MLOps enables ML teams to effectively experiment with different approaches and deploy models easily.

For the ML engineer working, this flow looks like the following:

- ML Engineer works in local development environment (Jupyter Notebook, Python in IDE, R, ...)
- Once experimentation is done, the ML engineer ensures their code is properly structured and has good test coverage
- The ML engineer checks in their code with the remote VCS
- The new version of the code triggers the CI/CD to build and push new artifacts (pipeline templates, component Docker containers, ...)
- Either the CI/CD pipeline immediately kicks off the pipeline run or the ML engineer triggers this manually.
- The ML pipeline then prepares the training data, trains and evaluates the model and pushes it to an endpoint.

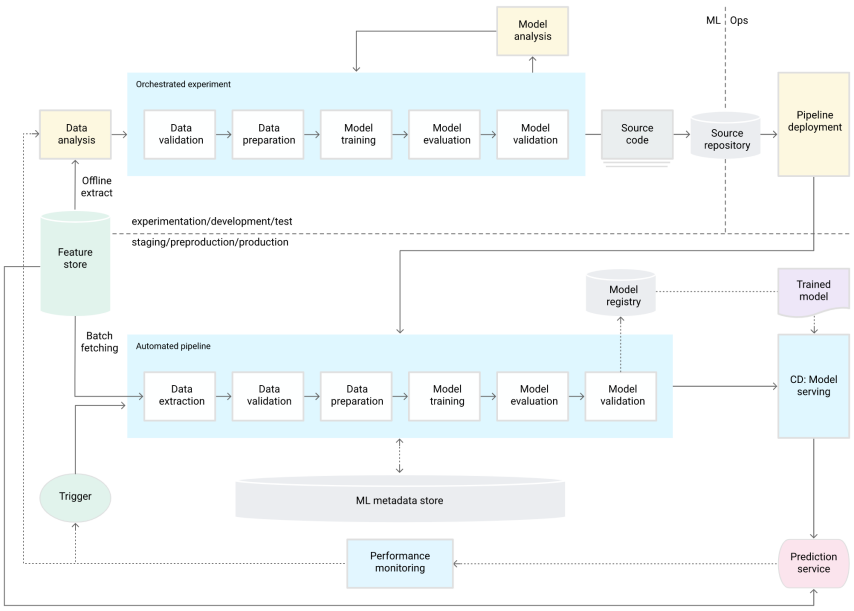

Figure 4: An end to end pipeline example. [1]

Following this flow ensures the entire process is repeatable and enables the ML engineer to reuse artifacts across environments. It also eliminates a lot of the manual work of the engineer and can take extra steps towards automation such as saving artifact metadata to the

metadata store, automating monitoring, coordinating experiments, etc.

3.2 The Feature Store

A Feature Store is a key component in MLOps. It consists in a centralized repository that stores and manages features used in machine learning models. These features can be raw data, preprocessed data, or derived features that have been created during the feature engineering phase. The main goal of the feature store is to enable data sharing within the organization.

It typically includes several key features, such as:

- Data store: It stores and manages features in a structured format.
- Data access: It provides APIs and other interfaces for data scientists and engineers to easily access and retrieve features for use in their models.
- Data lineage: It tracks the lineage of features, including where they came from and how they were processed.
- Data catalog: It allows data scientists and engineers to easily discover and search for features based on various criteria, e.g. feature names, data types, source systems, etc.
- Data versioning: It applies versioning capabilities to the features and allows to rollback to a previous version if needed.
- Data governance: It allows to implement data governance policies, e.g. data validation, data security, access controls, etc.
- Data monitoring: It can monitor for feature drift
- Automation: It can be integrated with other MLOps tools to automate the feature engineering, feature extraction, feature selection, and feature transformation.
- Feature serving: It can be customized to meet different model consumption needs, i.e. batch or real-time predictions.

Features within a feature store should be designed to be used by multiple models and pipelines. They should be a representation of the different business entities in the company.

3.3 Environment management

Deploying new models and code directly into a production environment is never a good idea. The chance of having a service interruption and as a consequence drop of revenue is something to be avoided. This is why it is recommended to have multiple environments to test changes in. Most companies will have at least three environments:

- **Development (DEV)**: This is where ML Engineers run their experiments, develop model APIs and do all their testing.
- **User Acceptance Testing (UAT)**: As the name suggests, this is where a select number of end users tests the model outputs to ensure they make sense in the context of the application they will be used by.
- **Production (PROD)**: The live environment. This is what the end users of the model will see and interact with.

When working with different environments, it is important to decide how to move something from DEV to UAT and from UAT to PROD (this is often called “promoting" to the next environment). In the most basic setup, this could be done manually by an engineer. It is easy to see how this can lead to errors during the manual process, which can lead to service interruptions.

Instead, this should be automated. A popular concept of relevance here is called GitOps, where interactions with the VCS drive the changes in the environments. Using this principle, a basic workflow can be constructed for development and promotions of models.

- ML Engineer checks in code on a feature branch. A feature branch is a branch meant for working on a specific feature, so could be a new model, enhancement of a new model, ...
- Artifacts are built by the CI/CD pipeline
- ML Engineer runs pipelines and experiments to train a new model and does the first round of testing
- ML Engineer merges his code to a shared development branch meaning his feature is ready for release
- Development is merged into the release branch which deploys the trained model into the UAT environment
- A select group of end users performs a round of testing to validate model output
- If the model validation is done, the code is merged into the production branch (often called main). This triggers deployment of the model into production.

3.4 Monitoring of deployed models

3.4.1 SLAs, SLOs and SLIs

Once a model has made it into production, it is important to be notified of any service interruptions or degradations. Traditionally this is discussed in terms of SLAs, SLOs and SLIs:

- **Service Level Agreement**: A Service Level Agreement or SLA is a contract put in place between a service provider and service consumer, which states objectives to be met for the service being provided. These are often very similar to the SLOs defined below, but less strict (SLOs should be breached before SLA).
- **Service Level Objective**: A Service Level Objective or SLO is a specific objective relating to one performance measure of the service being provided. Typical examples of this could be:
  - **Availability**: The service should be available 99.99% of the time over 12 months
  - **Accuracy**: The model should have an accuracy of 80%
  - ...
- **Service Level Indicator**: A Service Level Indicator or SLI is an attribute that is being measured to determine whether or not the service level objectives are being met. Examples are:
  - Service Uptime Percentage
  - Model accuracy
  - Latency
  - ...

3.4.2 Meeting SLOs

In order to meet the SLOs that were set, SLIs need to be measured and alerts need to be sent out promptly if there they fall below a certain defined threshold. Which measures need to be set up will be defined by the SLOs. A practical example of this could be:

- The team has set an SLO to have an average latency of 20ms
- To achieve this, the team is logging every call made to the service and every response from the service
- A metric is defined as the time between request and response
- When the team notices latency being longer than 20ms, they get an alert and investigate immediately to resolve the issue as soon as possible

When setting SLOs, it can also be useful to count in an **error budget**. An extra barrier of safety to serve as a buffer, this can be seen as the buffer between the SLO and the corresponding requirement in the SLA. Having this buffer, will also allow to account for planned downtime. Which is a good practice as it can help identify systems that rely too heavily upon each other to function. For example, the entire website should keep working, even when the recommender system is down.

3.4.3 Measuring Model Performance SLIs

One very common problem with measuring ML performance is the absence of specific metrics at inference time. The reason for this is that the "correct" value is not present at inference time. Some examples:

- A model that predicts next months' revenue, will only know the actual value the month after
- A model that recommends a product to buy, will only know if it was correct on checkout

This means that there is a need for a feedback loop, where correct values are fed back to the monitoring system to create metrics on model correctness. This can often prove challenging, but is vital to assess model performance. More on this is discussed in section 3.6.

3.5 Monitoring for Model Drift

In the ML business, it is common to see constant changes in the environment that can dramatically impact the predictive accuracy of machine learning models. This concept is well known and is called model drift. It refers to the fact that model performance is degrading compared to what it predicted during the training phase. Out of this main drift derives the concept drift, meaning there's a shift in the relationship between your models input(s) and ouput(s). A common example is the covid-19 which drastically changed the consumer behaviours, all kinds of social/economical data and therefore lots of patterns that may have been trained to be recognized by ML models. Secondly, we have data drift that can occur when the statistical properties of your input or output data change. This problem happens oftenly for input features/variables used to train the model or even for labels (also called label drift). The main reasons why we have feature drift are wide but can often be related to changes in the data sources or in the data processing layer. On the other side, label drift is mostly caused by the changing distribution of your output.

Following section 3.4.3, model performance degradation should always be monitored for. However at this point, degradation has already taken place and there was a measurable impact on the end user of the service. This impact can be lessened by monitoring for input data drift as well. It will allow the ML engineer to detect issues sooner. To achieve this, one has to examine the distribution of a specific feature over a sliding window of time and perform a statistical analysis to determine whether the distribution has changed. If data drift is detected, the techniques in section 3.6 should be applied to keep the model up to date.

3.6 Keeping models up to date

Keeping models up to date is one of the big challenges ML engineers face. As discussed above, model drift can cause performance degradation to models running into production. The good news is that retraining a model should not be a big challenge. This comes back to the pipeline template mentioned in section 3.1.

Following the approach described above, any model running in production will have been created by a pipeline run and every run will have been associated with a specific pipeline template. In theory, retraining is then as simple as creating a new run of the pipeline template, but providing a new dataset with more recent data.

When this should be triggered depends on the use case, SLO and budget. The simplest solution is to retrain on a fixed schedule, but that is hard to trace back to the actual SLO that was defined. This is why it is common to trigger a retraining either based on model performance degradation or input feature drift. This way a model can be retrained only when needed.

Another approach is to take into account the fact that data distributions might change at the start of model development. **Online learning** approaches, allow ML Engineers to design models that learn at runtime. This can be a good choice when data volatility is high or datasets are too big to be used efficiently for training.

3.7 Infrastructure as code

Now that the components of an MLOps system have been identified, it is important to think about how those components need to be set up. This is where Infrastructure as Code (IaC) is vital. The principle of IaC is that all infrastructure should be defined in code, not created manually through scripts or a UI. This has a multitude of benefits:

- **Versioning of infrastructure**: just like application code, infrastructure as code can be versioned. This means it can then be

promoted across environments just like discussed before. Infrastructure is first applied in dev, then uat and lastly prod. On top of this, rollbacks are also easy as engineers only need to apply a previous version of the infrastructure code.

- **Disaster recovery**: things out of the engineers control can happen (a datacenter going down, someone with permissions they should not have removing something manually, ...). This is when having the entire infrastructure in code is extremely valuable. It allows engineers to spin up a new instance of the environment relatively quickly.
- **Reusability**: specific parts of the infrastructure as code (modules) can be reused in order to replicate existing infrastructure for new environments or models.

Common tools used for Infrastructure as Code are Terraform, Ansible, etc. They enjoy broad community support and are running in production at many companies over the world. In the case of the system described above, the major components that need to be included in IaC would be:

- **The VCS system (can be self hosted)**: with its repositories, groups and access policies.
- **The CI/CD pipeline**: depending on which technology is used, it might be necessary to create runners for the CI/CD pipeline.
- **The artifact registry**: the artifact registry serves for storing pipeline templates and Docker containers. There are many options for this either open source or from different cloud providers.
- **The pipeline runner**: the pipelines need compute to be able to run training and data processing jobs. The most common platform for this is some flavor of Kubernetes.
- **The model registry**: when a model has been trained, it needs to be stored somewhere so it can be deployed later. This is the model registry. The most basic version of this is a blob store where the model is stored as a file.
- **The serving infrastructure**: models need some form of compute to run on. This can be a virtual machine (VM), Kubernetes or some serverless offering from a cloud provider.
- **Monitoring & alerting**: there are many options for monitoring and alerting infrastructure. In general, it includes some way to collect logs and metrics, alerting policies and notification channels.

- **Re-training infrastructure**: as mentioned in section 3.6, it is important to be prepared to retrain models. This infrastructure often includes some sort of scheduler and will probably reuse existing training infrastructure.

4 Machine Learning services on Google Cloud

The goal of this section is to introduce all relevant services for doing Machine Learning on Google Cloud. How these sections relate to the different parts of MLOps will be discussed in section 5.

4.1 Cloud Storage

Cloud Storage is a highly available blob storage service on GCP. It allows engineers to store any form of (un-)structured data. This makes it the perfect candidate to store any raw data that might be encountered.

Buckets can be configured to be multi-regional, dual-region or regional. The best configuration will be chosen based on the use case. Compliance, pricing and durability requirements can all affect this decision. In general, it is a good idea to keep data stored near to where it will be used.

Blobs are stored in a specific **storage class**. The storage class determines latency, availability (together with location) and cost. The following storage classes are available:

- **Standard**: for frequently accessed data or data that only needs to be stored for a short amount of time
- **Nearline**: for data that is accessed less than once a month, for example backups
- **Coldline**: for data accessed once every quarter or less. A common use case is disaster recovery
- **Archive**: for data accessed less than once a year

Cloud Storage can be managed either via the API or through its own CLI called **gsutil**. This allows for flexible and automated ingestion of data from a variety of sources. It is a common service for creating a "landing zone" on GCP, where data is first ingested into a data lake before being cleaned and processed.

4.2 BigQuery

BigQuery is one of Google's flagship services. It is a Dremel based SQL database optimized for running analytical workloads with support for semi-structured data. The word analytical is important here. BigQuery was designed as a columnar database with high query performance for aggregations etc. It was NOT designed to be an operational database for an application and does not have good performance for indexing and mutating specific rows.

However, BigQuery is a very powerful tool for ML Engineers, Data Engineers and Data Scientists. It can dramatically speed up data exploration on large volumes of data. It has native integrations with Cloud Storage and many other GCP services. This makes it relatively easy to transform raw data that landed on Cloud Storage into a more structured format, either as a data warehousing effort or as feature engineering.

Another benefit is the performance for dashboarding workloads. BigQuery BI Engine can serve as a cache, lowering query latency. Google's dashboarding tools (Looker and Looker Studio) natively integrate with BigQuery BI Engine, leading to a smooth end user experience.

BigQuery also has a built in ML tool called BigQuery ML, which allows engineers to easily train and use machine learning models through SQL. This is a powerful tool for rapid prototyping taking advantage of BigQuery's distributed compute engine.

All these features make BigQuery a key service on any data project.

4.3 Vertex AI

Vertex AI is the service that encompasses all different AI services in Google Cloud. Services can be divided mainly into four categories:

- **Managed APIs**: these are ML APIs that provide machine learning as a service. Examples are the Vision API, Video Intelligence API among others. They can be used, without any need for your own data, out of the box to solve a variety of use cases.
- **AutoML**: AutoML services allow users with little experience to build machine learning models by providing their own data and letting Google handle the training aspect. AutoML comes in various forms depending on the use case. Examples are AutoML Vision, AutoML Natural Language among others.
- **ML Platform Services**: these services allow users to create and manage their own Machine Learning models. It includes Vertex AI Training which allows you to run any training jobs on Google's compute services.

- **MLOps Services**: these services aim to support tasks such as model lifecycle management, monitoring, automated deployment etc. Services include Vertex AI Pipelines, Model Registry, Feature Store, etc.

The aim of this section is not to provide an exhaustive list of all services, but rather to provide a brief explanation of all the services that are relevant in most implementations of ML projects.

4.3.1 Vertex AI Training

Vertex AI training is the backbone for most ML projects on GCP. It allows engineers to submit training jobs, which are Docker containers that execute specific training code. Because this is so generic it can actually be used to run any code similar to Cloud Run for example, but it is optimized for machine learning workloads.

Training jobs can use any container in the form of **custom training**. However, Google also provides pre-built containers for training Tensorflow, Pytorch, Scikit-Learn and XGBoost models. These containers have all the necessary dependencies installed for training their specific workloads and are guaranteed to work with Vertex AI Training.

Vertex AI Training allows you to choose which compute resources you want to use for your training jobs including hardware accelerators (GPUs and TPUs). It will then communicate which resources are available by setting specific environment variables.

Virtual machines for training can also be configured to use private IP for extra security. This allows tighter control of how the VMs can be accessed.

4.3.2 Vertex AI Pipelines

Most of the time, it can be complex to deploy models, handle every phase of your machine learning system. And lots of questions around the cost and scaling of your model can occur. Provisioning, maintenance of your computing resources is an enormous task to handle. One of the best way to handle it and minimize the time to deploy is by using serverless pipelines that connect each steps of your solution.

Vertex AI Pipelines does this in the best way by playing the role of an orchestrator which schedules different pipeline steps. All pipeline steps (also the ones that do tasks such as data processing) are run as training jobs. You can then easily automate your workflow and monitor your artifacts with the integrated service Vertex ML Metadata Store. Vertex AI Pipelines serves as a runner for two open source MLOps tools, namely Kubeflow and Tensorflow Extended. As the most generic and popular of the two is Kubeflow, the terminology used in this document matches that of Kubeflow.

4.3.3 Vertex AI Metadata Store

The metadata store in Vertex AI stores information on artifacts that were generated throughout a pipeline. What information is stored depends on the type of artifacts and what the engineer decided to store. It allows users to trace back how specific artifacts were generated.

As an example, let's say a model was created by a specific pipeline run. That pipeline run pulled in some data, modified it, did some basic quality analysis and then trained a model. The metadata store will enable engineers to trace back all the way from to the trained model to the exact state of the source data at training time. This enables them to investigate why the model is behaving a specific way. There could have been outliers in the training data for example.

An important clarification is that the metadata store only stores metadata about the artifacts. It does not actually store the artifacts themselves. The type of artifact will determine what the best place is to store the artifact is. This can be a Vertex AI Dataset, a model in the Model Registry, data on Cloud Storage, etc.

4.3.4 Vertex AI Model Registry

As mentioned in section 3.7, an important component is the model registry. This is where models are stored and versioned. Vertex AI has a dedicated service for this, which allows users to store different versions of specific models on the registry.

This allows engineers to easily roll back to previous versions and manage their rollout strategy.

4.3.5 Vertex AI Feature Store

Vertex AI Feature store is a database capable of serving features in two modes, offline or online. This allows it to serve data for training and online inference use cases as the online latency is very low. The same way, data can be ingested into the feature store through two mechanisms. Batch ingestion allows users to ingest data from BigQuery or from AVRO or CSV files on Cloud Storage. Streaming ingestion allows entities to be ingested one by one through an API.

Data in feature store is stored according the following data model:

- **Entity**: an entity maps to a business concept like a user, a product, etc.
- **Feature**: a feature is one attribute of the entity such as the age group of a user
- **Feature Value**: the value of a specific feature for a specific entity at a specific point in time

Here it is important to understand that feature store has an innate understanding of time. The value of a feature of an entity can change over time. For example, users will age, so their age group can change. Because feature store links feature values to specific timestamps, it is easy to get the value of a feature at a specfic point in time.

4.3.6 Vertex AI Workbench

Vertex AI Workbench offers ML Engineers a managed JupyterLab environment where they can experiment and do data exploration. The main benefits are that it integrates very well with other GCP services and can leverage more powerful compute than would be available locally. On top of this it can sync to Github for version control.

Vertex AI Workbench is powered by Google Compute Engine, which means it has the same cost and security benefits. Notebooks can be hosted with private IPs within a specific VPC subnet and instances can be started and stopped on demand.

5 Applying MLOps on Google Cloud

Now that the basics of GCP and MLOps have been covered, this section will go over how to combine them to set up an end to end MLOps system on Google Cloud. This section will follow the workflow starting from the ML Engineer's local machine up until the model is up and running in production.

5.1 Data Exploration and Experimentation

Initially, data on GCP will be stored either on Cloud Storage or BigQuery for analysis. ML engineers will start doing data exploration to identify useful features for the model they want to build. This can be done either on a local machine (in which case one should be careful about data exfiltration risks) or on a Workbench instance, in which case the engineer is already working inside of the Google environment.

Once features have been identified, ML engineers will probably start prototyping different types of models to find out which approaches work and which do not. Vertex AI Workbench allows them to leverage powerful virtual machines for local training, or they can submit Vertex AI Training jobs instead. Using AutoML and/or pre-built APIs is also a standard practice to quickly identify the baseline performance for a given use case.

After selecting which models to further test, it is time to start working on a training pipeline, rather than local experimentation. The de-

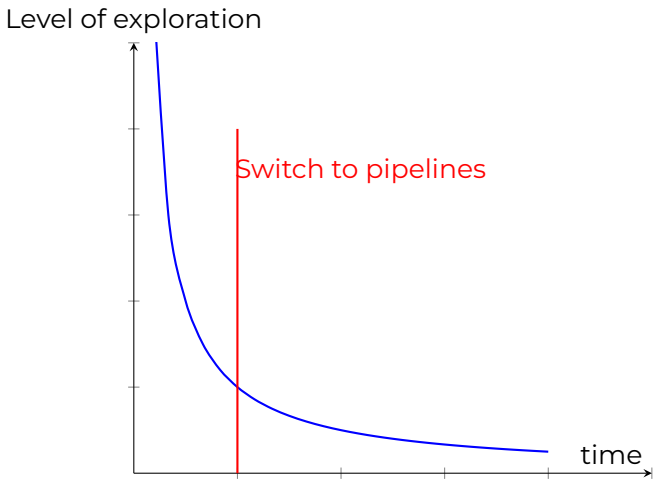

Figure 5: graph showing when to switch to Vertex AI Pipelines

velopment of an ML pipeline usually starts after most of the exploratory analysis has been carried out and the general shape of the model is defined, as shown in figure 5.1. In general there is a tradeoff between development velocity and solution robustness and quality. When to make this switch will depend on the type of problem and individual preference.

5.2 Vertex AI Feature Store

Vertex AI Feature Store is a great tool allowing you to store, serve and monitor ML features at scale. It especially fits the overall MLOps picture as it allows ML engineers to track the evolution of features over time. Feature Store can create alerts for when your ML feature drift from their original distribution. This is especially important in ML features, as data arrive continuously. At any time, the features may change from what they used to be at training time, eventually leading to model deterioration. Feature drift is thus a critical event to detect, so that models can be retrained and ensure business continuity. Also, feature drift may be a sign that the architecture of the ML models need a change. For this reason, feature drift must not only trigger a retraining of the model, but also alert relevant technical and business profiles within an organization.

5.3 Pipelines on Vertex AI

Once model selection has been narrowed down to a couple of possible architectures and data pre-processing is in a more or less stable state, it is time to properly structure the code.

The code structure depends heavily on what type of pipeline tool is used. Most often, code is split up into components that perform one specific function. In case code needs to be used across multiple components, this code can be placed into a private Python package that is then installed in the necessary components. As explained, components are executed as Docker containers. This means that each component runs in its own environment, which can be customized at build time with the packages that are needed for the specific task.

An important step when designing a component is making sure that its inputs and outputs are well defined. Not only does this ensure that the resulting artifacts are stored properly in Vertex AI Metadata, but it is also required in order to pass objects from one component to another and make them overall reusable. The dependencies between components define the execution graph of the pipeline, therefore the order in which tasks are executed. It is important to verify that the components are not circularly dependent, otherwise it will not be possible to compile them into a pipeline.

Once the pipeline is defined, it can be written as a pipeline template and stored in Artifact Registry. Just like components, pipelines can be created in a reusable fashion. For example, one may create a general pipeline template to tackle all binary classification use cases, performing all steps of the machine learning workflow, including hyperparameter tuning to ensure that the best possible model is ended up being created.

5.4 Model Training

In most cases, a pipeline will contain at least one training component. In order to benefit from the resources available in Vertex AI training, the training component should instantiate a Vertex AI training job, either by running an AutoML training job or by launching a Vertex AI custom training job in a pre-built or custom container.

5.4.1 AutoML Training

AutoML models are suitable for many use cases concerning tabular, image, video or text data. AutoML training jobs do not require any custom training script, nor is it necessary to perform hyperparameter tuning or cross validation. However, only little customization is available over the training resources. Often, you can only control the total budget for a given training job. The trained model will automatically

be uploaded to Model Registry and will be available for deployment. Optionally, AutoML models can be exported to be used on an edge device or in an on-premise environment.

The main benefit of AutoML is that it enabled less ML knowledgeable people to train their own models without too much effort. It can also solve as a useful tool to create strong baseline models.

5.4.2 Custom Training

Training a model with a custom application enables the ML engineer to have full control over the process. The training job is run in a container, either pre-built or custom. In order to run training in a pre-built container, it is required to provide a script containing the training code that will be executed during the job. When working with a custom container, the training application should be included in the container and the Vertex AI job should only specify the command to run when launching the container. Both custom training methods allow the user to fully customize the training environment as the resources (potentially, distributed) to be allocated for the task can be specified in the Vertex AI job definition. The ML engineer should ensure that cross validation is performed, if needed. This can be done as part of the application or as part of the pipeline execution. After training, it may be necessary to upload the model to Model Registry in order to deploy it to an endpoint.

5.4.3 Custom Hyperparameter Tuning Training

A particular type of custom training job is the hyperparameter tuning job. Although it is possible to implement a custom tuning application as a component, the most effective procedure to choose the best hyperparameters for a model is to use the tool provided by Vertex AI, namely, Vizier AI. It is a black-box hyperparameter tuning framework that is able to reach greater performance over diverse use cases, using notably less compute resources.

A hyperparameter tuning job can be instantiated starting from any other custom training job, with the addition of details over the parameters to be optimized, the metrics and the search algorithm. The tuning job then launches the training application multiple times in parallel, with different sets of parameters. After completion, the results of the optimization job will be available in the Vertex AI console, with extensive information over each training run allowing ML engineers to discover how their models behave in different settings.

5.5 Model Hosting

The last step in an ML pipeline is to deploy a trained model to an endpoint, so that it can be queried for online predictions. AutoML models handle predictions requests without the need for a serving application. On the other hand, custom models provide predictions through a serving container. Pre-built solutions are available but it is also possible to build a custom application to use for serving. Multiple deployment approaches are available on GCP, three solutions are compared below.

5.5.1 Vertex AI Endpoints

Vertex AI Endpoints is Vertex AI's serverless model hosting service. Any model uploaded on Model Registry can be deployed to an endpoint. Custom serving containers are supported. In this case any prediction request sent to the endpoint is handled by a custom application. A given model can be deployed on multiple endpoints at a time, which can for instance be beneficial when a model should provide predictions through several endpoints with different specifications. Moreover, multiple models can be deployed to a given endpoint. In this case it is required to specify how the traffic is split among all models. This is particularly useful for A/B tests and canary releases of new model versions.

A Vertex AI endpoint can scale horizontally, thus adding nodes if more computational power is needed to provide predictions. However, the minimum number of nodes required at any given time is 1, therefore it is not possible for the endpoint to completely scale down to 0 prediction nodes.

Vertex AI Endpoints are designed to integrate well with other Vertex AI resources, for this reason they provide more in-depth monitoring over model performances than other deployment solutions. Model Monitoring is available for models deployed on Vertex AI Endpoints, therefore providing a service to automatically detect prediction anomalies.

5.5.2 Cloud Run

Cloud Run is a serverless service that allows the user to run containerized applications. A serving application with a custom prediction script can then be hosted on Cloud Run to deploy a trained model, which does not need to be uploaded to Model Registry. This solution also allows autoscaling of nodes to accommodate variable utilization rates, including scaling down to 0 nodes unlike Vertex AI Endpoints. Cloud Run allows some customization over the resources that run the serving application. Cloud Run provides monitoring over container uti-

lization, however, model monitoring is not integrated as in Vertex AI Endpoints and should be implemented separately, for instance by the means of custom metrics. Additionally, rolling out new model versions and A/B testing must be handled on the application side as Cloud Run does not contain any logic around ML models.

5.5.3 Google Kubernetes Engine

Google Kubernetes Engine (GKE) is a managed service providing Kubernetes clusters. It is used for deployment of containerized applications, therefore it can host a serving container for model predictions as with Cloud Run. In GKE, applications are run on a set of machines forming a cluster. This environment is fully customizable and can be configured to suit all requirements in terms of computational power, scaling and availability. Both horizontal and vertical autoscaling are available, although the autoscaler will not completely scale down to 0 nodes. Monitoring in GKE is limited to infrastructure and cluster metrics, so model performance should be assessed separately.

5.5.4 When to choose which option?

Vertex AI Endpoints is the default option, as it integrates best with all other Vertex AI services and it provides the best options in terms of features. However, it is not necessarily the most cost effective solution. This due to the fact that it cannot scale down to zero.

For cases where budget is a concern, Cloud Run can be a better option due to the fact that it allows the model service to scale down to 0 instances.

Google Kubernetes engine can be interesting for teams that are already heavily using the GKE ecosystem. It can provide efficiency gains due to closer management of the underlying compute resources. The main downside is the extra infrastructure management effort.

5.6 Model Monitoring

Once a model is deployed, monitoring should be configured to ensure that the quality of the performance does not drop because of various drift. Model Monitoring is available for tabular models, both AutoML and custom, that are deployed on Vertex AI Endpoints. Skew and drift detection are managed by Model Monitoring jobs, alerts can be configured to warn the user in the event of an anomaly. The job will monitor the activity of an endpoint and produce graphs that the user can visualize in the console to further analyze the results. Input data from prediction requests is saved and analyzed in BigQuery. In particular

for skew detection it is necessary to also provide training data in BigQuery in order to compare training features with incoming data.

Moreover, for cases where your model need to predict offline (batch cases), it is possible to detect drift at the beginning of your model development with a Vertex AI pipeline component using TensorFlow Data Validation in order to calculate the statistical distribution of your features. In the same spirit, we have the possibility to extend our monitoring capability with Explainable AI by introducing feature attribution methods and making sure that your model don't drift in the attribution values.

5.7 Model transparency and fairness

Many industries face strict regulation over the usage of artificial intelligence powered features. Additionally, quickly-evolving technologies and international laws make it hard for non-lawyers to follow-up on the latest updates. For this reason, we advocate all ML engineers to adopt stricter internal practices. Mainly, this effort focuses on three equally important points:

- The use cases tackled
- The features being used
- The models that are developed

For sure, questions may be asked around the types of use cases that your company tackles. One must define proper boundaries of what they consider acceptable. Especially, be aware that some use cases can be used for other purposes than the one they were originally designed for. This of course concerns the sensitivity of the data that is being used, but also the concepts that could easily be transferred from one set of data to another.

Then, ML engineers should pay attention to which features of the data are used to train machine learning models. For instance, banks may not include the gender of a person to assess his/her credit risk score.

Finally, the type of model being used is of importance, as it may determine compliance with a given use case. One pain point in machine learning is to gain and retain user trust, and one way to alleviate this is to develop explainable models. These models are able to, thanks to various techniques, produce results and explain why it predicted it. This can vary from custom strategies, to feature attribution and even image segmentation. Importantly, model validation and evaluation play an important role in model fairness, as a peculiar analysis of the performance of your models may highlight important discrepancies in your dataset.

In general, we strongly encourage ML engineers to ensure full model transparency, at least within their own organization. This often comes with a detailed documentation explaining implementation choices, an argument of the features being used and why it complies with a company's moral boundaries, as well as a complete dissection of the performance of the models.

As for tools, Vertex AI has Explainable AI that can automatically or manually provide explanations to models' results. Additionally, Google's What-If Tool provides a great toolkit for discovering caveats in ML models. Of course, those are only tools and will not replace strong AI ethics within a company.

5.8 A practical example

The previous sections described the process for developing ML solutions, from the design stage until production. To better understand how the steps should be assembled into one pipeline, a sample project is provided on GitHub[1]. The pipeline defined in this project implements an AutoML model predicting credit card default. The model is trained on the *credit_card_default* dataset, which is publicly available on BigQuery. The target variable *default_payment_next_month* can take values 0 or 1, thus making the model a binary classifier.

The pipeline contains four components:

- Creation of a tabular dataset, using BigQuery as a data source
- Training an AutoML model using the tabular dataset as input
- Creation of an endpoint hosted on Vertex AI
- Deployment of the trained model to the endpoint

The project also includes other files, which define the infrastructure of the GCP project. This ensures the reproducibility of the sample MLOps pipeline in any environment.

6 Conclusion

Machine Learning represents a valuable asset for companies, however its full potential can only be reached by following the principles of MLOps, as to optimize the process of designing, developing and deploying models.

MLOps promotes the utilization of ML pipelines to define the workflow that brings a model to life. Pipelines are composed of modular components, each isolating a specific task, and are compiled to a

[1] Available at `https://github.com/devoteamgcloud/ml-on-gcp-pipeline`

static template. This encourages re-usability and Agile development. Moreover, the automation of tasks reduces significantly the risk of error compared to manual deployment of new models.

GCP provides an optimal environment for the development of machine learning solutions. Google Cloud Storage allows the user to organize and store data into buckets, while guaranteeing availability and granular access control. BigQuery can benefit any ML project by providing a solution for data warehousing, data exploration and feature engineering, as well as native integration with visualization tools. Vertex AI offers a set of ML products to suit all needs, from ready-to-use APIs to deployment of fully customized models. An ML engineer can then choose the best solution, depending on the scope of the project and the desired outcome.

Applying MLOps principles to ML projects is essential for an optimal delivery. Moreover, GCP provides ML engineers with services that accelerate model development and ensure that the deployed products are effective.

References

[1] Google Cloud. *End to end MLOps*. URL: `https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning`. (accessed: 12.01.2023).

[2] Google Cloud. *Resource hierarchy*. URL: `https://cloud.google.com/resource-manager/docs/cloud-platform-resource-hierarchy`. (accessed: 10.01.2023).