Vertex AI Service Map

Vertex AI is Google Cloud's unified ML platform with over a dozen interconnected services. This map helps you quickly identify which service to use for each stage of the ML lifecycle.

Service	Purpose	When to Use
Vertex AI Training	Custom model training	Need full control over training
AutoML	No-code model training	Tabular, image, text, video classification
Vertex AI Pipelines	Orchestrated ML workflows	Reproducible training pipelines
Feature Store	Centralized feature management	Shared features across models
Model Registry	Model versioning and metadata	Track, compare, and promote models
Endpoints	Model serving (online/batch)	Real-time or batch predictions
Vertex AI Studio	Prompt design and tuning	GenAI experimentation
Model Garden	Pre-trained model catalog	Use or fine-tune foundation models
Vertex AI Agent Builder	Build AI agents and RAG	Grounded generation, agent tools
Model Monitoring	Detect drift and anomalies	Production model health
Experiments	Track training runs	Compare hyperparameters, metrics

AutoML vs Custom Training

AutoML gets you a baseline model in hours with zero code; custom training gives you full control at the cost of engineering effort. Start with AutoML to establish a benchmark, then switch to custom only if you need to beat it.

Aspect	AutoML	Custom Training
Code required	None (UI/API config)	Full training code
Model control	Limited (architecture chosen)	Full (any framework)
Data prep	Minimal (automatic)	Manual feature engineering
Training time	Hours	Flexible
Best for	Prototyping, tabular data	Research, complex architectures
Cost	Higher per training hour	Lower with spot/preemptible
Explainability	Built-in	Manual (SHAP, etc.)
Export	TF SavedModel, container	Any format

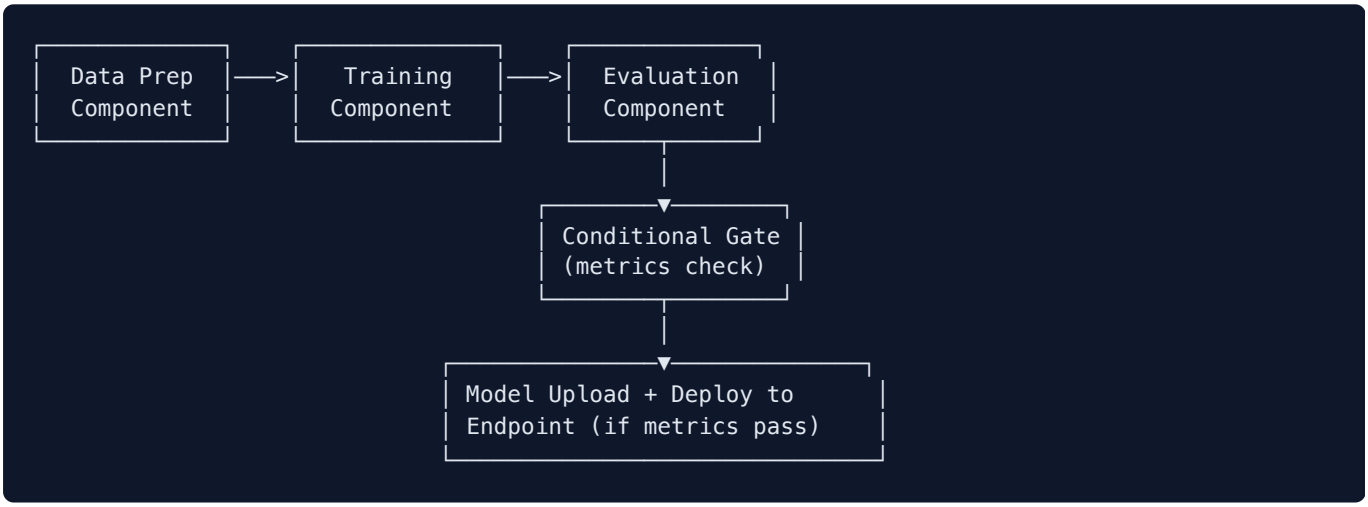
AutoML Supported Tasks

Data Type	Tasks
Tabular	Classification, regression, forecasting
Image	Classification, object detection, segmentation
Text	Classification, entity extraction, sentiment
Video	Classification, object tracking, action recognition

Vertex AI Pipelines

Pipelines make your ML workflow reproducible, auditable, and automatable. Without them, training runs are one-off scripts that nobody can reproduce six months later.

Pipeline Structure



Pipeline Definition (KFP v2)

```
from kfp import dsl
from google.cloud import aiplatform

@dsl.component(base_image="python:3.11", packages_to_install=["pandas", "scikit-learn"])
def preprocess_data(input_path: str, output_path: dsl.OutputPath("Dataset")):
    import pandas as pd
    df = pd.read_csv(input_path)
    # preprocessing logic
    df.to_csv(output_path, index=False)

@dsl.component(base_image="python:3.11", packages_to_install=["scikit-learn", "joblib"])
def train_model(
    dataset_path: dsl.InputPath("Dataset"),
    model_path: dsl.OutputPath("Model"),
    learning_rate: float = 0.01
):
    import pandas as pd
    from sklearn.ensemble import GradientBoostingClassifier
    import joblib
```

```
    df = pd.read_csv(dataset_path)
    X, y = df.drop("target", axis=1), df["target"]
    model = GradientBoostingClassifier(learning_rate=learning_rate)
    model.fit(X, y)
    joblib.dump(model, model_path)

@dsl.pipeline(name="training-pipeline")
def training_pipeline(input_path: str, lr: float = 0.01):
    preprocess_task = preprocess_data(input_path=input_path)
    train_task = train_model(
        dataset_path=preprocess_task.outputs["output_path"],
        learning_rate=lr
    )
```

Running a Pipeline

```
from google.cloud import aiplatform

aiplatform.init(project="my-project", location="us-central1")

job = aiplatform.PipelineJob(
    display_name="training-run-001",
    template_path="pipeline.json",  # compiled pipeline
    parameter_values={"input_path": "gs://bucket/data.csv", "lr": 0.01},
    pipeline_root="gs://bucket/pipeline-runs",
)
job.run(service_account="sa@project.iam.gserviceaccount.com")
```

Feature Store

Feature stores solve the problem of feature consistency between training and serving -- without one, teams recompute features inconsistently, leading to training-serving skew and duplicated engineering effort.

Key Concepts

Concept	Description
Feature Group	Logical grouping of related features (e.g., "user_features")
Feature	Individual data column with type and metadata
Entity	Key that identifies the row (e.g., user_id)
Online Store	Low-latency serving for real-time predictions
Offline Store	BigQuery-backed for batch training
Point-in-time lookup	Get feature values as of a specific timestamp

Feature Store Operations

```
from google.cloud import aiplatform
```

```
# Create feature group
fg = aiplatform.FeatureGroup.create(
    name="user_features",
    source=aiplatform.FeatureGroup.BigQuerySource(
        uri="bq://project.dataset.user_features_table"
    )
)

# Create feature (column in the source table)
feature = fg.create_feature(name="purchase_count")

# Online serving: create Feature Online Store
online_store = aiplatform.FeatureOnlineStore.create(
    name="production-store",
    bigtable=aiplatform.FeatureOnlineStore.Bigtable(
        auto_scaling=aiplatform.FeatureOnlineStore.Bigtable.AutoScaling(
            min_node_count=1, max_node_count=3
        )
    )
)

# Create feature view for online serving
view = online_store.create_feature_view(
    name="user_features_view",
    source=aiplatform.FeatureView.BigQuerySource(
        uri="bq://project.dataset.user_features_table",
        entity_id_columns=["user_id"]
    )
)
```

Model Registry

The model registry is your single source of truth for what models exist, which version is in production, and how each version performed. Without it, model management becomes spreadsheets and tribal knowledge.

Uploading a Model

```
model = aiplatform.Model.upload(
    display_name="fraud-detector-v2",
    artifact_uri="gs://bucket/models/fraud-v2/",
    serving_container_image_uri="us-docker.pkg.dev/vertex-ai/prediction/sklearn-cpu.1-3:latest",
    labels={"team": "fraud", "version": "2"},
    description="Gradient boosting fraud detection model"
)
```

Model Versioning

```
# Upload as new version of existing model
model_v2 = aiplatform.Model.upload(
    display_name="fraud-detector-v2",
    parent_model=existing_model.resource_name,  # creates new version
    artifact_uri="gs://bucket/models/fraud-v2.1/",
    serving_container_image_uri="...",
    version_aliases=["champion"],
```

```
    version_description="Improved recall by 5%"
)
```

Endpoints and Serving

Endpoints handle the operational complexity of serving models at scale -- autoscaling, traffic splitting, and health checks. Use traffic splitting for A/B tests and canary deployments instead of risky all-at-once rollouts.

Deploy Model

```
endpoint = aiplatform.Endpoint.create(display_name="fraud-endpoint")

endpoint.deploy(
    model=model,
    deployed_model_display_name="fraud-v2",
    machine_type="n1-standard-4",
    min_replica_count=1,
    max_replica_count=5,
    traffic_percentage=100,
    accelerator_type=None
)
```

Traffic Splitting (A/B Testing)

```
# Deploy new model alongside existing one
endpoint.deploy(
    model=model_v3,
    deployed_model_display_name="fraud-v3-candidate",
    machine_type="n1-standard-4",
    min_replica_count=1,
    max_replica_count=3,
    traffic_percentage=10  # 10% to new model
)
# Existing model automatically gets 90%
```

Online Prediction

```
instances = [
    {"feature1": 1.0, "feature2": "category_a", "feature3": 42}
]
prediction = endpoint.predict(instances=instances)
print(prediction.predictions)
```

Batch Prediction

```
batch_job = model.batch_predict(
    job_display_name="batch-fraud-scoring",
    gcs_source="gs://bucket/input/batch_data.jsonl",
    gcs_destination_prefix="gs://bucket/output/",
    machine_type="n1-standard-4",
```

```
    max_replica_count=10,
    accelerator_count=0
)
batch_job.wait()
```

Model Monitoring

Models degrade silently as the real world drifts away from training data. Monitoring for feature skew and prediction drift catches these problems before they impact business metrics.

Monitoring Setup

```
from google.cloud.aiplatform import model_monitoring

# Define skew/drift thresholds
skew_config = model_monitoring.SkewDetectionConfig(
    data_source="bq://project.dataset.training_data",
    skew_thresholds={"feature1": 0.3, "feature2": 0.2}
)

drift_config = model_monitoring.DriftDetectionConfig(
    drift_thresholds={"feature1": 0.3, "feature2": 0.2}
)

# Create monitoring job
monitor_job = aiplatform.ModelDeploymentMonitoringJob.create(
    display_name="fraud-monitoring",
    endpoint=endpoint,
    logging_sampling_strategy={"random_sample_config": {"sample_rate": 0.1}},
    schedule_config={"monitor_interval": {"seconds": 3600}},  # hourly
    objective_configs=[{
        "deployed_model_id": deployed_model_id,
        "objective_config": {
            "training_dataset": skew_config,
            "training_prediction_skew_detection_config": skew_config,
            "prediction_drift_detection_config": drift_config
        }
    }],
    alert_config={"email_alert_config": {"user_emails": ["team@company.com"]}}
)
```

Monitoring Metrics

Metric Type	What It Detects	Method
Feature skew	Training/serving data mismatch	Distribution comparison (Jensen-Shannon)
Feature drift	Input distribution shift over time	Distribution comparison over windows
Prediction drift	Output distribution shift	Distribution comparison
Attribution drift	Feature importance shift	Explanation-based monitoring

Compute Options

Choosing the wrong machine type wastes money on over-provisioning or causes OOM failures during training. Match your compute to your workload -- use GPUs for model training and right-size CPU machines for preprocessing and serving.

Machine Type	vCPUs	Memory	GPU	Use Case
n1-standard-4	4	15 GB	Optional	Small models, preprocessing
n1-standard-16	16	60 GB	Optional	Medium models
n1-highmem-8	8	52 GB	Optional	Memory-intensive
a2-highgpu-1g	12	85 GB	1x A100	Large model training
a2-highgpu-8g	96	680 GB	8x A100	Distributed training
g2-standard-4	4	16 GB	1x L4	Inference, light training

Common Pitfalls

Most Vertex AI failures come from skipping automation and monitoring -- the same mistakes that plague any ML platform. These shortcuts save time initially but create painful debugging sessions later.

Pitfall	Problem	Fix
No pipeline versioning	Can't reproduce results	Use pipeline templates with version tags
Manual deployments	Error-prone, slow	Automate via CI/CD pipeline
No monitoring	Silent model degradation	Set up skew/drift alerts
Over-provisioned endpoints	Wasted cost	Use autoscaling, right-size machines
Ignoring Feature Store	Feature recomputation, inconsistency	Centralize features
No A/B testing	Risky all-at-once deployments	Use traffic splitting
Training on local machine	Not reproducible	Use Vertex AI Training
Hardcoded project/location	Not portable	Use environment variables