Flyte

Open Source Cloud Native Machine Learning and Data Processing Platform

Ketan Umare
Software Engineer
Lyft
@KetanUmare
@KetanUmare

Haytham Abuelfutuh
Software Engineer
Lyft
@HaythamAfuluh
@HaythamAfuluh
Motivation
What motivated us to build Flyte?

Goals
Desirable properties of an ideal production ML system

Introducing Flyte
Principle offering & architecture

Demo
Everyone loves demos!

Conclusion
Learn more, get involved, & get started
Motivation

Developing large-scale, complex ML & Data pipelines is hard.

The overhead of infrastructure and difficulty collaborating adds significant friction.
Motivation

Data and machine learning are converging.

There is increasing need for a single tool for both.
Motivation

ML is more than just the model
Motivation

Data & infrastructure are big hurdles
Motivation

ML & Data services are increasingly complex and interdependent

- AV Simulation
- Autonomous Maps
- Street Imagery
- Manual Map correction
- Mapping
- Routing
- Estimated time of arrivals
- Rider
- Forecasting
- Shared rides
- Insurance
- Pricing
- Incentive Model
- Feature store
- Fraud
- Driver

3500+ Unique Workflows
300k+ Workflow executions per month
1 million+ task executions per month
10 million+ containers executed per month
Goals

Flyte wants to make it easy to

Orchestrate ML & Data Workflows

Collaborate, Reuse, and perform ML Ops Across Teams
Introducing Flyte

Hosted, scalable and serverless
Orchestration Platform

Fabric that connects disparate compute technologies

Extensible and Observable

Integrates best of the breed open source solutions

Auditable and Secure
Introducing Flyte

Tasks

Atomic unit of work & entrypoint to user code

- Explicitly versioned
- Strongly typed Interface
- Arbitrarily complex: can be single process, multi-process, distributed or remote executions
- Extensible
- Declarative Specified in Protocol Buffers

```python
@inputs(rides=Types.Schema[...], k=Types.Integer)
@outputs(dest=[Types.String])
@spark_task(spark_conf={...})
def find_topk_destinations(ctx, spark_ctx, rides, k, dest):
    # Find the top k destinations for the given set of rides ordered by frequency
```

```python
def run_shell_sort = ContainerTask(metadata=..,
    interface={inputs:{file:..}, outputs:{..}},
    container=Container(
    image=..., command=['/bin/sort', '-n'],
    args=['{{.inputs.file}}'],
    resources=Resources(req,limit),
    env={}, config={}))
```
Introducing Flyte

Workflows

Specify the data dependency between tasks (as DAGs)

- Strongly typed Interface
- Composable & Dynamic Workflows can be extended by composition of other workflows statically or dynamically
- Versioned @Lyft by git commits
- Declarative Specified in Protocol Buffers

Decoupled Scheduling, scheduler triggers executions at a scheduled time passing the time as input

```python
@workflow_class
class TrainModel(object):
    # Accept inputs
    data = Input(Types.Schema[...])
    hyperparam = Input(Types.Float)
    # Split the dataset
    split = split8020(data=data)
    # Fit the model
    model = fit_xgboost(
        data=split.train,
        hyperparam=hyperparam)
    # Evaluate the model
    pred = eval_xgboost(data=split.val,
        m=model.outputs.v)
    # Compute the metrics
    metrics = compute_metrics(
        data=split.val,
        pred=pred.y_pred)
    # Create outputs
    model = Output(model.outputs.v)
    accuracy = Output(metrics.outputs.acc)
```
Serverless for users

User should only worry about business logic

- They only specify resource requirements like CPU, GPU, memory, number of spark executors etc.
- They can work on multiple versions of code.
- Their code is containerized.
- Multi-tenancy: They do not worry about other users.
- Resource pooling and Quota: Downstream resource are protected from Brown-outs.
- All of Flyte’s power is available using a simple gRPC/REST interface.
- They can use multiple languages, with first class support for Python - Flytekit.
Introducing Flyte

Architecture Overview

Default: Single Kubernetes cluster with scale-out options to cloud services like AWS Batch.
Introducing Flyte

Architecture Overview @Lyft MultiCluster

@Lyft: we use multiple Kubernetes clusters to isolate multiple failure domains and scale-out.

FlyteAdmin supports this mode out of the box.
Introducing Flyte

Grouping & Sharing

Projects, Domains & Versions
- Projects offer **logical grouping** of Workflows & Tasks and can be split across one or more repositories, one or more containers
- Domains and Versions provide **CI/CD like semantics** to Workflows & Tasks
  - Users can **push new** versions to production, **rollback** to previous version etc.
  - Users can have workflows in **integration/staging** env
- Domains are **configured globally** for the system (by administrators)

Sharing & Accounting
- Workflows can **refer to tasks and workflows** from other projects
- Executions **accounted/billed** under the **requesters project & domain (Infraspend)**
Introducing Flyte

Shareability: Flytekit Example

```python
@workflow_class
class PipelineA(object):
in1 = Input(Types.Integer)
in2 = Input(Types.Integer)
...
out1 = Output(print2.outputs.out)

@inputs(x=Types.Integer, y=Types.Integer)
@outputs(z=Types.Integer)
@task
def my_model(x, y):
    ...

@workflow_class
class CompositePipeline(object):
    composed_wf = lps.fetch(
        "ProjectA",
        "Production",
        "PipelineA",
        "1.0.2"
    )(in1, in2)

t1 = local_task(composed_wf.outputs.out)
t2 = tasks.fetch(
    "ProjectA",
    "Production",
    "my_model",
    "2.0.0"
)(x=t1.outputs.x, y=10)

Project: ProjectA

Project: ProjectB
```
Every task execution in Flyte is recorded by default in Catalog Service. This enables Flyte executions to have,

**Artifact Lineage**
- **Causal** dependencies between data and processes is tracked

**Memoization**
- Each task execution has a **unique signature**, which includes the input values & version of code
- **Repeated** executions with matching signatures are cached
Introducing Flyte

Observability for the User

Extensive user visibility (per workflow, per project etc) - e.g graffana macro @ Lyft
Introducing Flyte

Designed for ease of operations

Alerting and notifications
Customizable notifications, with existing integrations - pagerduty, slack and email
Coming soon Subscribable notifications for Workflows & node state transitions

Security
Per execution access controls using ServiceAccounts, IAM Roles
Oauth2 auth flow

Ofcourse we have Deep platform level visibility for Admins
Introducing Flyte

Extensible: Container-Only Flytekit Plugins

**What:** Flytekit offers easy extensibility, takes care of the boilerplate and provides tooling for development, testing, and deployment.

**How:** These plugins are executed in containers. Find `@flytekit/contrib`

**Why:** Useful in rapidly extending capabilities of Flyte

```python
@sensor_task
def my_test_task(ctx):
    ...
    E.g. sensor that waits for a hive partition to land. This is added as a contrib.
    ...
    return MyHivePartitionSensor()

@sensor_task
def my_test_task(ctx):
    ...
    E.g. sensor that waits for a hive partition to land. This is added as a contrib.
    ...
    return MyHivePartitionSensor()

@sensor_task
def my_test_task(ctx):
    ...
    E.g. sensor that waits for a hive partition to land. This is added as a contrib.
    ...
    return MyHivePartitionSensor()
```

```python
@xgboost_hpo_task(
    static_hyperparameters={
        "eval_metric": "auc",
        "objective": "binary:logistic",
    },
    train=train_data,
    validation=validation_data,
)
```
Introducing Flyte

Extensible: Notebooks and Papermill

**What:** Flytekit makes it possible to author any task type (Spark, Hive, Python, etc.) from a Python notebook with a full set of input/outputs. Papermill notebooks can be run for any kernel with primitive inputs/outputs.

**How:** Flytekit provides wrappings to enter notebook environments and marshall I/O

**Why:** It provides an easy path from development to production with excellent debuggability.

```python
import types

task = notebook_task(
    "notebooks/train_model.ipynb",
    "inputs": {
        "train": types.Schema(
            [(
                "label", types.Integer,
            ), ...]
        ),
        "validation": types.Schema(
            [(
                "label", types.Integer,
            ), ...]
        ),
    },
    "outputs": {
        "model": types.Blob
    }
)
```
Introducing Flyte

Extensible: Backend Plugins

**What:** Flyte backend is extensible. This provides deep integration into Flyte.

**How:** A Simple Golang interface available under FlytePlugins (pluginmachinery)

**Why:** This is great for adding tasks that need

- Special visualization
- Custom logging and other information
- Guaranteed cleanup of resources
- Perfect for managing CRD’s
Demo

DAG Creation
Use Flytekit to create tasks & workflows

Registration
Register tasks, workflows & launch plans

Flyte UI
Visualize, launch, & monitor Flyte workflows

Sharing Tasks & Workflows
How Flyte enables collaboration

Data Catalog & Memoization
How to increase efficiency & decrease costs with Flyte DataCatalog

Docs
Where to go to learn, get started, & do more with Flyte
Flyte.org
Demo

Clean

Dirty
Recap

- **Protobuf-based** language specification.
- Task and workflow interfaces are **strongly typed**.
- Tasks and workflows are **shareable & discoverable**.
- Workflows are **composable**.
- Task outputs can be **cached** to speed up re-execution.
- Executions are **repeatable**.
Introducing Flyte

Registration Process

1. Build one or more docker containers
2. User write workflow
3. Compile Task
4. Compile Workflow
5. Workflow Specification
6. Task Specification
7. Upload the containers to a registry
8. Container Registry
9. FlyteAdmin Service
10. Upload to Service
11. Verify
12. Store with version
13. Store compiled Workflow version
Introducing Flyte

Executing a Registered Workflow
Introducing Flyte

Ecosystem
Flyte is constantly evolving and new features are coming soon like,

- **Reactive workflows** (respond to data publication events)
- Enhancements to **type system** and **Flytekit**
- **More extensions**
- Richer **data catalog**
many more...

To find more details visit our docs and the Roadmap section. Also join our fledgeling community and help us shape the future of Flyte. We appreciate contributions and suggestions.
Thanks!
Learn more, get started & keep in touch at Flyte.org

@KetanUmare
@KetanUmare
@HaythamAfutuh
@HaythamAfutuh