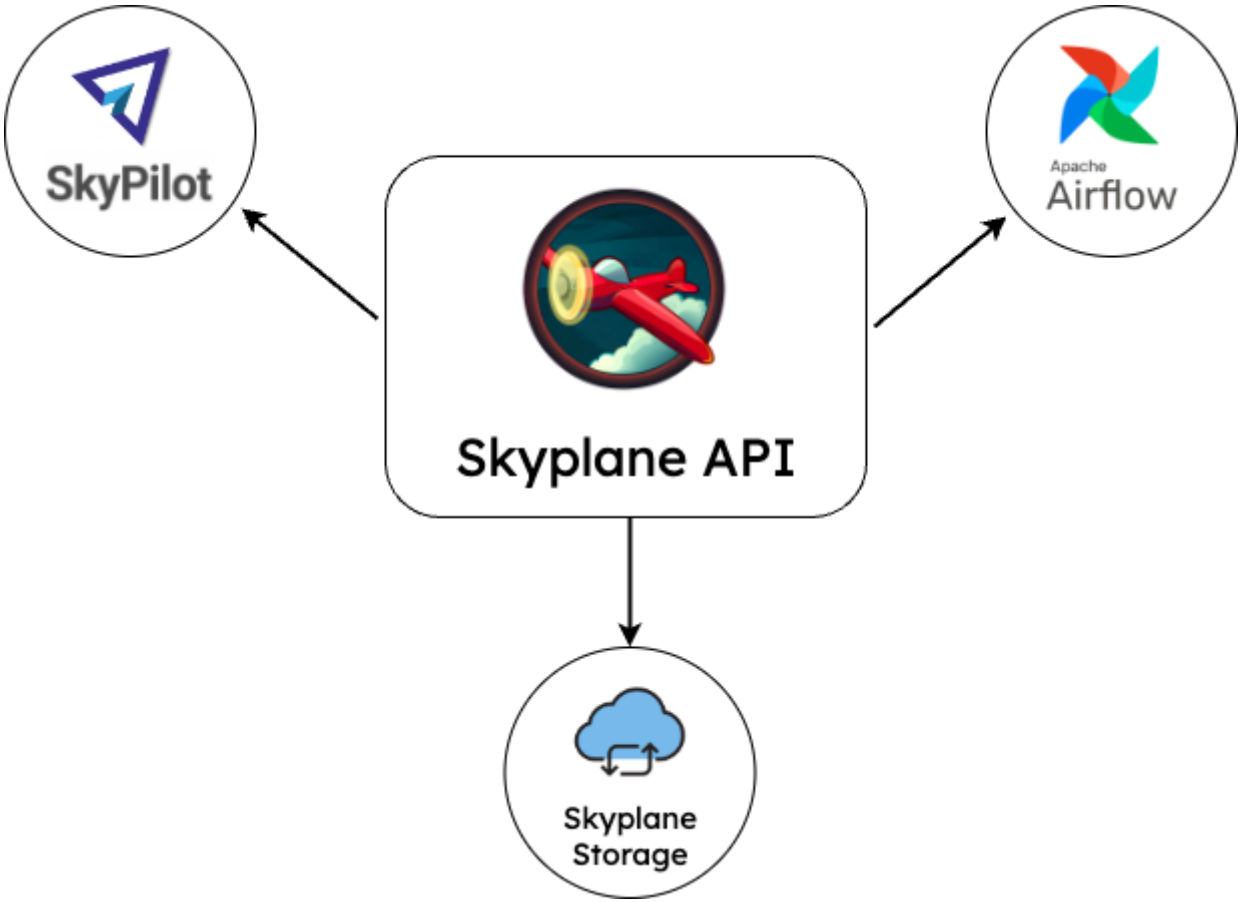
Skyplane API

Users are starting to compose Skyplane into larger applications! Data transfer across the cloud is often a critical component of most project pipelines, and Skyplane's simplicity and efficiency is an attractive tool for developers to handle this stage of the process. For easy and fast integration, Skyplane offers an API which enables users the same functionality as the CLI (e.g. copy, sync, etc.) along with some API-specific features. This has exciting implications for the growth of Skyplane and applications leveraging it going forward.

Examples of use cases include but are not limited to:

1. ML training, You will hear about Skypilot;
2. Persistent synchronization, You can have incremental syncs to enable disaster recovery for apps running in other clouds;
3. Skyplane Storage, Building storage API on top of the current API.



Overview

In [2]:
```
from skyplane.api.api_class import *
```

Simple Copy

In [ ]:
```
client = SkyplaneClient()
client.copy(src="s3://jason-us-east-1/fake_imagenet/", dst="s3://jason-us-west-2/fake_imagenet_1/",
                    recursive=True)
```

Sessions

In [ ]:
```
session = client.new_session(
    src_region="aws:us-east-1",
    dst_region="aws:us-west-2",
    num_vms=1,
    solver=DirectSolver(),
)
with session as s:
    s.auto_terminate()
    s.copy("jason-us-east-1/imagenet", "jason-us-west-2/imagenet", recursive=True)
    future1 = s.run_async()
    s.copy("skyplane-us-east-1/video", "skyplane-us-west-2/video")
    s.copy("skyplane-1/texts", "skyplane-2/texts")
    future2 = s.run_async()
    s.wait_for_completion(future2)
    s.wait_for_completion(future1)
```

ML Training Leveraging Skyplane Data Transfer

In [4]:
```
import argparse
import os
import random
import shutil
import time
import warnings
warnings.filterwarnings('ignore')

import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.distributed as dist
import torch.optim
import torch.multiprocessing as mp
import torch.utils.data
import torch.utils.data.distributed
import torchvision.transforms as transforms
from torch.utils.data import IterableDataset, DataLoader
from awsio.python.lib.io.s3.s3dataset import S3IterableDataset

import torchvision.models as models
from PIL import Image
import io
from itertools import islice
```

In [5]:
```
class ImageNetS3(IterableDataset):
    def __init__(self, url_list, shuffle_urls=False, transform=None):
        self.s3_iter_dataset = S3IterableDataset(url_list,
                                                 shuffle_urls)
        self.transform = transform


    def data_generator(self):
        try:
            while True:
                # Based on aplhabetical order of files sequence of label and image will change.
                # e.g. for files 0186304.cls 0186304.jpg, 0186304.cls will be fetched first
                label_fname, label_fobj = next(self.s3_iter_dataset_iterator)
                image_fname, image_fobj = next(self.s3_iter_dataset_iterator)
                label = int(label_fobj)
                image_np = Image.open(io.BytesIO(image_fobj)).convert('RGB')

                # Apply torch visioin transforms if provided
                if self.transform is not None:
                    image_np = self.transform(image_np)
                yield image_np, label

        except StopIteration:
            return

    def __iter__(self):
        self.s3_iter_dataset_iterator = iter(self.s3_iter_dataset)
        return self.data_generator()
```

In [6]:
```
def train(train_loader, model, criterion, optimizer, epoch):
    # switch to train mode
    model.train()

    for i, (images, target) in enumerate(train_loader):
        # compute output
        output = model(images)
        loss = criterion(output, target)
        print(f"Current loss for batch#{i} is:")
        print(loss.item())

        # compute gradient and do SGD step
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

In [7]:
```
data_url = ["s3://jason-us-east-1/imagenet-train-000000.tar"]
# We can use Skyplane!!!
client1 = SkyplaneClient()
client1.copy(src="s3://jason-us-east-1/imagenet-train-000000.tar",
             dst="s3://jason-us-west-2/imagenet-train-000000.tar", recursive=False)
data_url = ["s3://jason-us-west-2/imagenet-train-000000.tar"]
```

```
Output()
```

```
copy: s3://jason-us-east-1/imagenet-train-000000.tar to s3://jason-us-west-2/imagenet-train-000000.tar
```

In [8]:
```
batch_size = 256

# Torchvision transforms to apply on data
preproc = transforms.Compose([
    transforms.RandomResizedCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225)),
])

dataset = ImageNetS3(data_url, transform=preproc)

train_loader = DataLoader(dataset,
                          batch_size=batch_size,
                          num_workers=2)

model = models.__dict__['resnet18'](pretrained=True)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), 0.1,
                            momentum=0.9,
                            weight_decay=1e-4)
```

In [9]:
```
for epoch in range(5):
    train(train_loader, model, criterion, optimizer, epoch)
    break
```

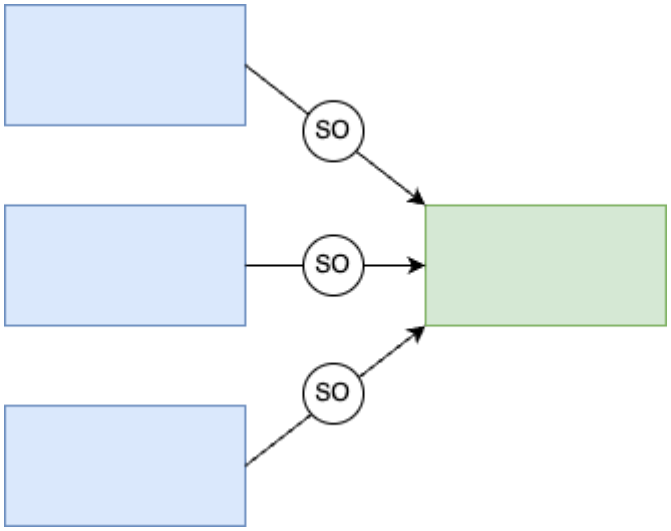
```
Current loss for batch#0 is:
1.2581912279129028
Current loss for batch#1 is:
1.3235325813293457
Current loss for batch#2 is:
1.4024291038513184
Current loss for batch#3 is:
1.588517665863037
```

In [10]:
```
# Finally, we can back up the model to the cloud, by Skyplane
torch.save(model.state_dict(), "./sample_model.pkl")
client1 = SkyplaneClient()
client1.copy(src="./sample_model.pkl",
             dst="s3://jason-us-west-2/", recursive=False)
```

```
upload: ./sample_model.pkl to s3://jason-us-west-2/sample_model.pkl
```

Integration into Existing Application Pipelines

Cross-cloud jobs on Apache Airflow are common in data engineering pipelines, and the integration of Skyplane makes this process much faster. This is an ongoing collaboration with Max Demoulin at Astronomer, and we can see a very simple example of this integration below.



In [ ]:
```
# code citation: GCS to S3 operator on Apache Airflow
from __future__ import annotations

import os
import warnings
from typing import TYPE_CHECKING, Sequence, List, Tuple

from airflow.models import BaseOperator

if TYPE_CHECKING:
    from airflow.utils.context import Context

class SkyplaneOperator(BaseOperator):
    template_fields: Sequence[str] = (
        'src_bucket',
        'src_region',
        'dst_bucket',
        'dst_region',
        'transfer_pairs',
        'config_path',
    )
    def __init__(
        self,
        *
        src_bucket: str,
        src_region: str,
        dst_bucket: str,
        dst_region: str,
        transfer_pairs: List[Tuple[str, str]],
        config_path: str,
        **kwargs,
    ) -> None:
        super().__init__(**kwargs)
        self.src_region = src_region
        self.dst_region = dst_region
        self.transfer_pairs = transfer_pairs
        self.config_path = config_path

    def execute(self, context: Context):
        # load auth credentials
        auth = SkyplaneAuth.from_config_path(self.config_path)
```

```
        # create client
        client = SkyplaneClient(auth)

        # create session
        session = client.new_session(src_region=self.src_region, dst_region=self.dst_region, num_vms=4)

        with session as s:
            s.auto_terminate()
            for src_prefix, dst_prefix in transfer_pairs:
                s.copy(self.src_bucket + src_prefix, self.dst_bucket + dst_prefix, recursive=True)
            future = s.run_async()
            s.wait_for_completion(future)
```

Exciting avenues in Skyplane Storage...

The Skyplane API enables flexible use of Skyplane and paves interesting roads for future applications built using the transfer tool. Persistent synchronization and Skyplane Storage are particularly fascinating directions to us, as we will touch on next...

In [ ]: