Architecting Computer Vision Workloads on the Cloud

A Technical Report submitted to the Department of Computer Science

Presented to the Faculty of the School of Engineering and Applied Science

University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree

Bachelor of Science, School of Engineering

Arvind Anand

Spring, 2022

Technical Project Team Members

Dylan Fernandes

On my honor as a University Student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments

Haiying Shen, Department of Computer Science

Table of Contents

Problem Statement

Business Context

A significant portion of the waste created in operating rooms comes from the wasteful use of single-use sterilized tools, which cannot be used again once opened. In something as important as a surgery, tools need to be ready to use immediately, so surgeons will often open all of a set of available tools onto the scrub table. All of the opened single-use tools are thrown away at the end of the surgery, regardless of whether they have been used or not. According to a survey of the hospital systems of UVA and UCSF by the research team at Periop Green, about 25% of all single use items are thrown away without ever having been used. This practice not only accumulates environmental costs, but also accumulates financial cost as well. According to a study of 58 adult neurosurgical cases, the average cost of unused supplies was $653 per surgery (Zygourakis et al, 2016).

The research team has determined that a tool has been used if it has been moved off of the scrub table during the surgery. By determining which items have moved off the scrub table, the research team can provide a synthesized report of all tools used during the surgery, and make customized recommendations as to what tools should be kept open/closed at the start of the surgery. The team has chosen to use machine learning and computer vision to create an algorithm that can recognize objects moving on and off the scrub table. By passing the model various frames of video footage, the research team can track the locations of various items on the scrub table. Using an analytics script, the team can use the locations of each item to determine if it has been used.

One of the problems that the team has run into is compute power and storage capacity. The workload had previously been running entirely off of a laptop in the UVA Link Lab, which isn't scalable or capable of deploying production grade applications. As a small startup with very little funding, Periop Green couldn't afford to pay for the server infrastructure costs required to scale. To reduce costs to a manageable level, the research team has chosen to build the infrastructure on the AWS cloud.

The technical project is focused on the company's migration to the AWS cloud, with the primary focus being supporting the machine learning workload. By moving to the cloud, the research team can take advantage of the near infinite storage and compute power of AWS.

Requirements and Constraints

From the technical perspective, there are many methods for implementing a computer vision system for detecting one-time-use tools. In fact, it's rather straightforward to implement a deep learning image classifier using a popular framework like Tensorflow or PyTorch. However, our business use case demands the following high level requirements: scalability and correctness. Our application needs to scale to run many training jobs and testing jobs simultaneously, in order to quickly run inference on incoming data and iterate on our model. On top of that, our model implementation has to be reliable such that the behavior of its output is consistent, regardless of the result. We chose to use Darknet's Yolov4 object detection algorithm to run our computer vision stack because it's already hyper parameterized and state of the art. The output can also be configured via command line arguments as per our needs, allowing us to streamline model inference.

In addition to ML model requirements, data streaming bandwidth is a constraint PeriopGreen needs to design around. While AWS can ship terabytes of data around it's various services, a local hospital network has significantly less bandwidth to stream real time video data. We need to design a system that can intelligently stream data when bandwidth is low, without clogging the hospital network.

Designing the Architecture

With the requirements in mind, we sought out to design a backend system that can ingest and store high volumes of image data, automatically train the model via event driven triggers, and relay the results to a storage bucket or analytics engine.

The Foundational Principles

Before designing the system, we needed to agree on a set of design principles that the system must adhere to. First, it has to be cloud based. The upfront costs to maintain a cluster on premises is too much for an early stage startup; the cloud was a natural fit for us because it allowed us to provision compute resources for a low upfront cost and because we could scale said resources if needed. Second, the system had to be event driven. While reducing cost isn't our primary objective, it was still in our best interest to optimize the usage of our compute resources: i.e only run services on demand. Event driven architectures enable services to run by having an intermediary services, like AWS Lambda, listen for specific events, such as new data in the training S3 bucket, and trigger relevant services. By leveraging event driven architecture principles, we can also leverage managed services provided by AWS such as Lambda, DynamoDB, ECS, Batch, and etc. In addition to the low overhead for configuring runtime environments, managed services allow us to seamlessly scale our compute resources automatically. This takes us to our third principle: stateless scalability. Whether we're serving 1 or 100 streams of data, our application should serve each request without relying on some arbitrary application state; running any number of inference jobs should scale our infrastructure based on demand. Our final principle is to containerize as much as possible. This idea ties stateless scalability with event driven compute: by keeping the environment consistent across all instances of our running application, we can scale seamlessly on demand by spinning up and spinning down containers as we see fit. Containers can also help us enforce stateless runtime by isolating and replicating the environment Darknet runs in.

Using these foundational principles, we've established a few different components (which we've dubbed, "engines") that will comprise the final pipeline, each of which serves a different functionality and exist as separate logical components. The primary focus of this project

is on the preliminary implementation of the model engine (which controls the training, evaluation, and iteration of the machine learning model) and the design of the analytics engine (which takes the model output and turns it into a list of items that were used and not used).

The Model Engine MVP

Our design processes started with defining our minimal viable product: the model engine. At a high level, the model engine needs to be a highly available and scalable ML training/inference system that adheres to our design principles. The MVP should serve as a starting point that will ultimately help us achieve object detection at scale.

Scope

We broke down the scope of the MVP into goals, non-goals, and acceptance criteria.

Goals

- To automate training of YOLO
- To asynchronously trigger inference jobs
- To scale training and inference jobs as well as infrastructure on demand
- To expose features only to internal apps/tools

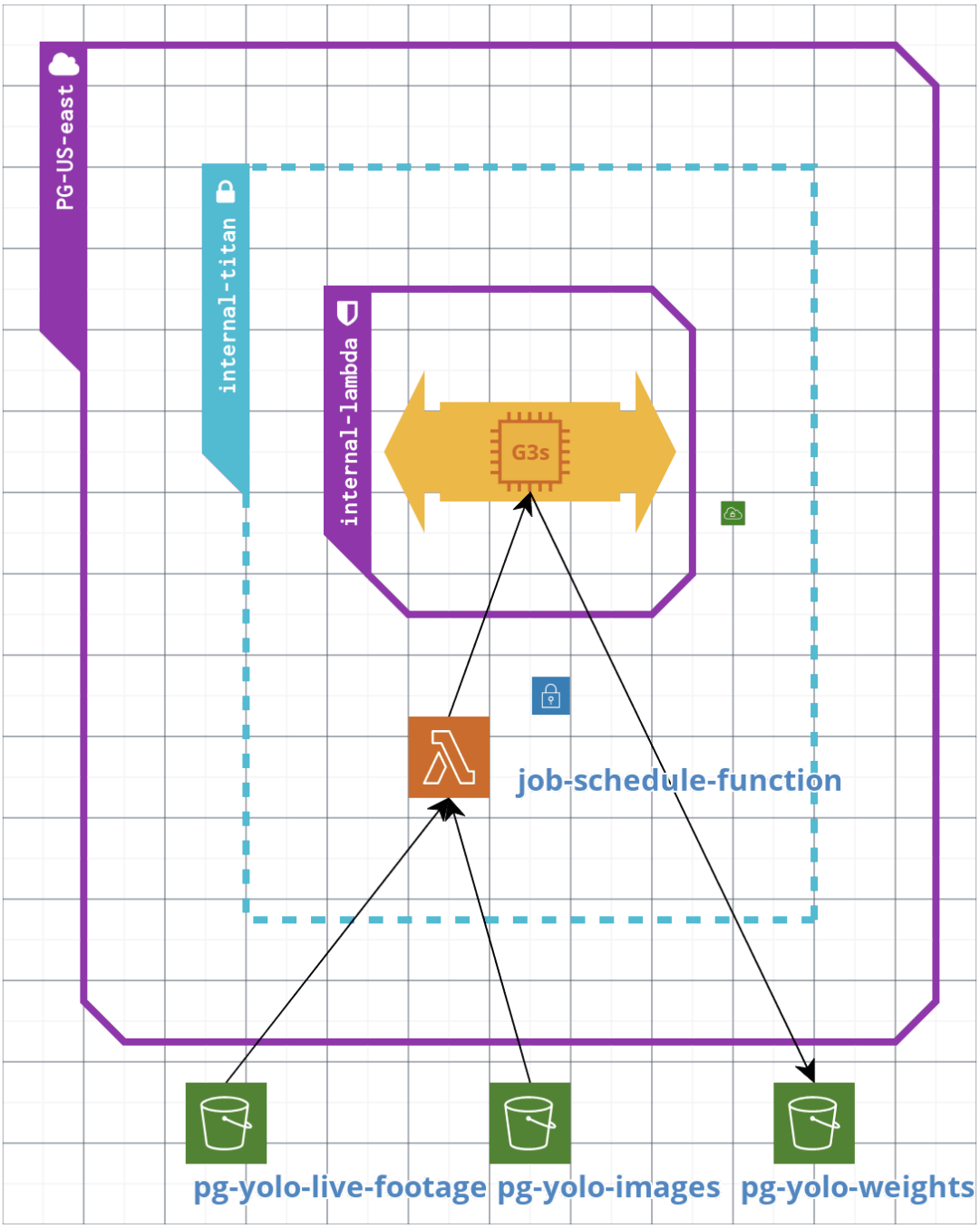
Non Goals

- To run analytics and produce reports to clients
- To clean up old/unused AWS artifacts
- To serve web pages and front end APIs to external users
- To facilitate the development of the ML model
- To extract, transform, and load data artifacts
- To manage data in S3, EFS, etc...

MVP Acceptance Criteria

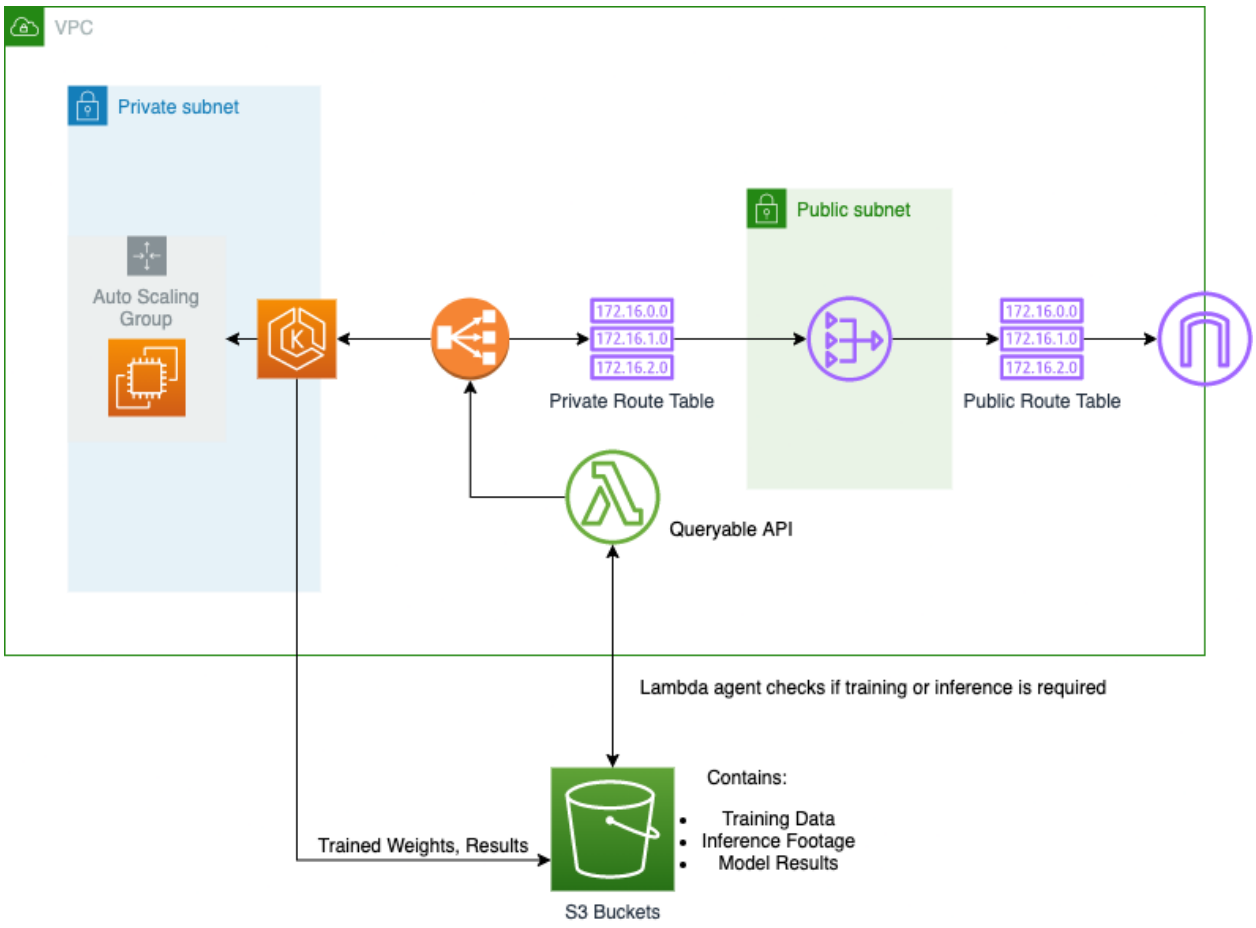
1. Recognizes when new footage/training data is uploaded into pg-yolo-images

2. Distinguishes between train and test jobs
3. Autonomously invokes darknet script for training and testing on G3 instance
4. Publishes results to results bucket/UI



The MVP is the minimum infrastructure required to schedule training, testing, and validation jobs in AWS. This architecture is kept simple in order to rapidly iterate on Yolov4 optimization.

Ultimately, we want to containerize our training and testing jobs and replicate them across multiple instances, AZs, and regions (if need be). This makes training and testing a highly available and scalable service. We also need the Model Engine to be resilient and easily recoverable if it goes down. For example, suppose a training job fails; first, the Model Engine should be resilient enough to not go down. Second, the Model Engine should recognize the failure and retry the training job. Every job run should be stateless for the system as a whole, in order to ensure proper resilience. We came up with the network architecture below to illustrate the possible final version we strive to develop the MVP into. As seen in the later sections, this design evolved overtime as a result of developmental iterations.

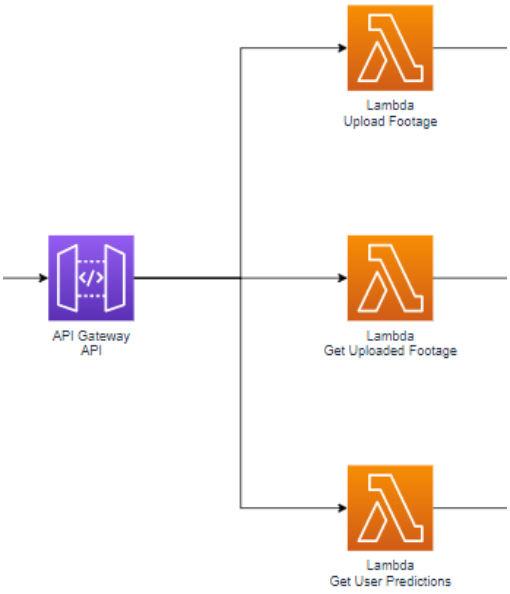


Previous Iterations of the Architecture

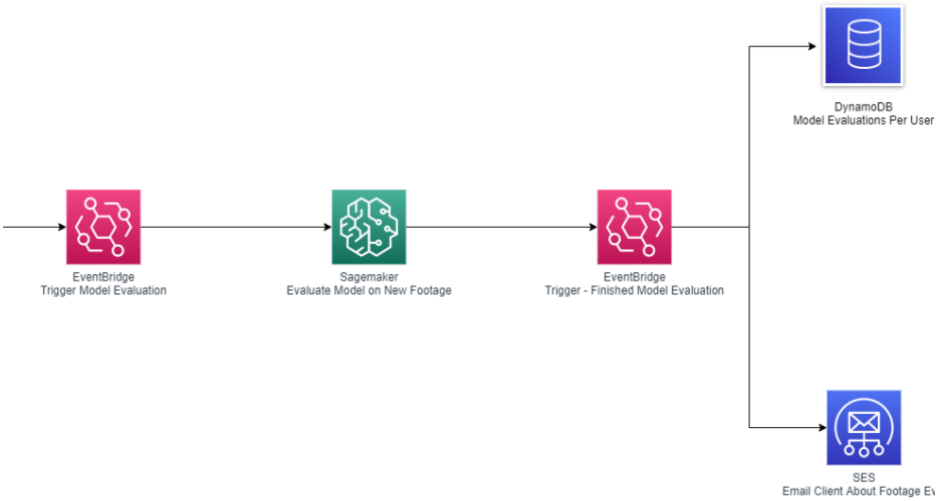
Our architecture has changed tremendously over time. In total, we've developed multiple vastly different versions of this architecture. Each of these versions represent a change in how we chose to approach this complex problem. Earlier drafts of the pipeline are less sophisticated, but are larger in scope and attempt to describe all the different ways that users would interact with our system. Later architectures are more sophisticated, but are more focused entirely on specific engines than prior versions (and are therefore not representative of the final state of our product).

Iteration 1 - A High Level, Problematic Pipeline

The first iteration of the pipeline was our first exploration into developing an architecture that discussed machine learning, user interactions via web application, and authentication. To date, it's the version of the architecture with the largest scope and the most focus on user interaction.



The entire pipeline revolves around users interacting with a REST style API with three primary actions - uploading footage, retrieving previous uploaded footage, and retrieving the reports generated for the user. These actions would all take place via a user interface, hosted on AWS Amplify. The user workflow has since changed since this diagram was created, but the heart of the user workflow remains the same - to use a CRUD based API to control all user actions.

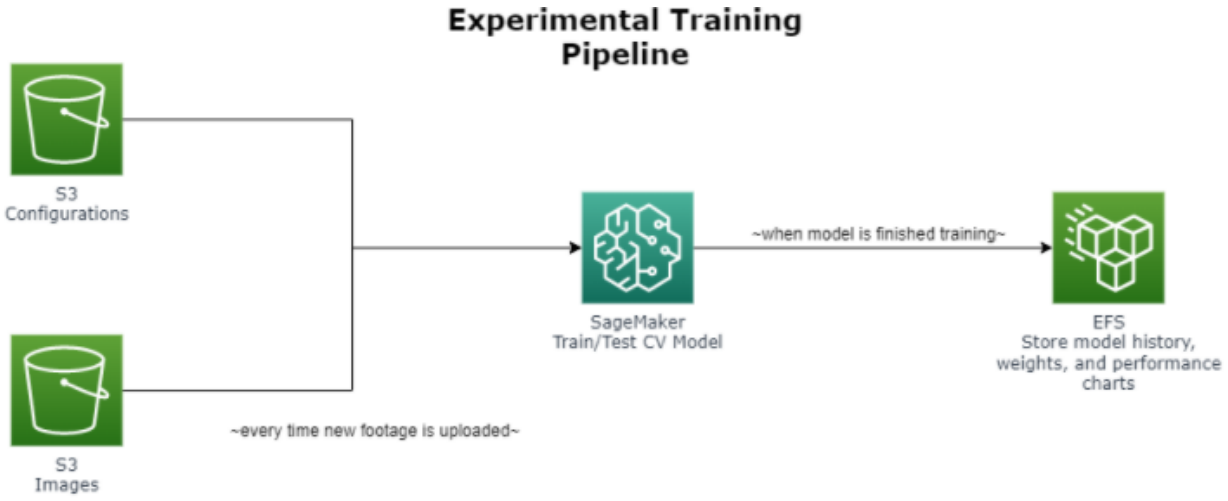


The footage upload workflow is the most interesting part of this iteration of the pipeline, as it represents our first attempt at evaluating user footage using an event driven architecture.

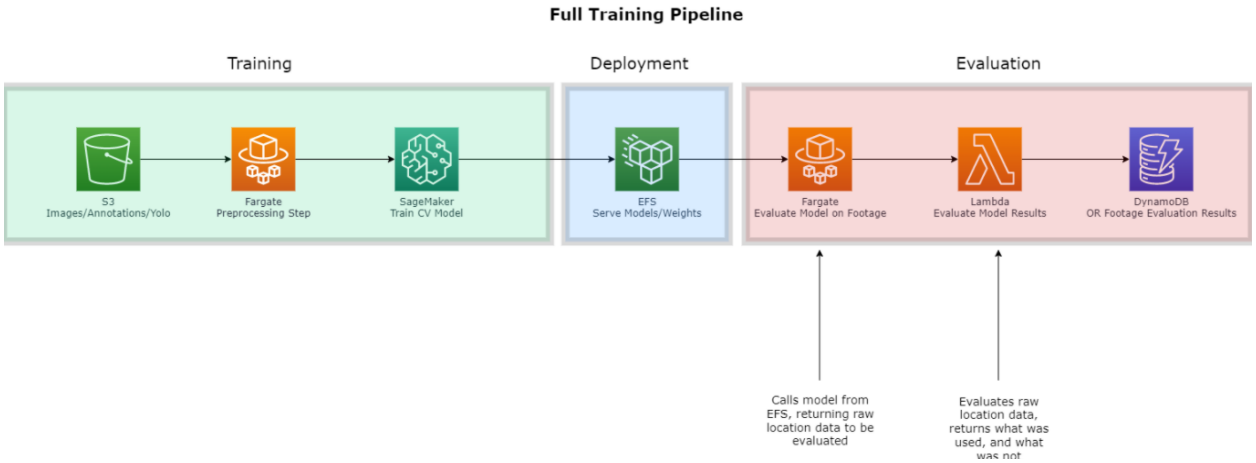
Once the API gateway receives a footage upload request, it sends the footage to an S3 bucket. The bucket triggers an event, which sends the footage to Sagemaker for evaluation, which is then sent to customers for approval.

This pipeline has a few problems, however. For instance, there's a few compute steps missing to process events. EventBridge doesn't have Sagemaker as a potential source, so the evaluation must be triggered or processed using a Lambda function. Similarly, as the evaluation is finished, EventBridge has no way to generate the report and send it to customers and store the results in DynamoDB. A Lambda function can be used to perform both of these functionalities. Finally, the pipeline doesn't include an analytics phase. We realized that the machine learning model doesn't emit a list of items used during the surgery, but instead produces a list of bounding boxes of the items. We instead need an intermediary analytics phase to produce these results.

Iteration 2 - A More Mature, Optimistic Pipeline



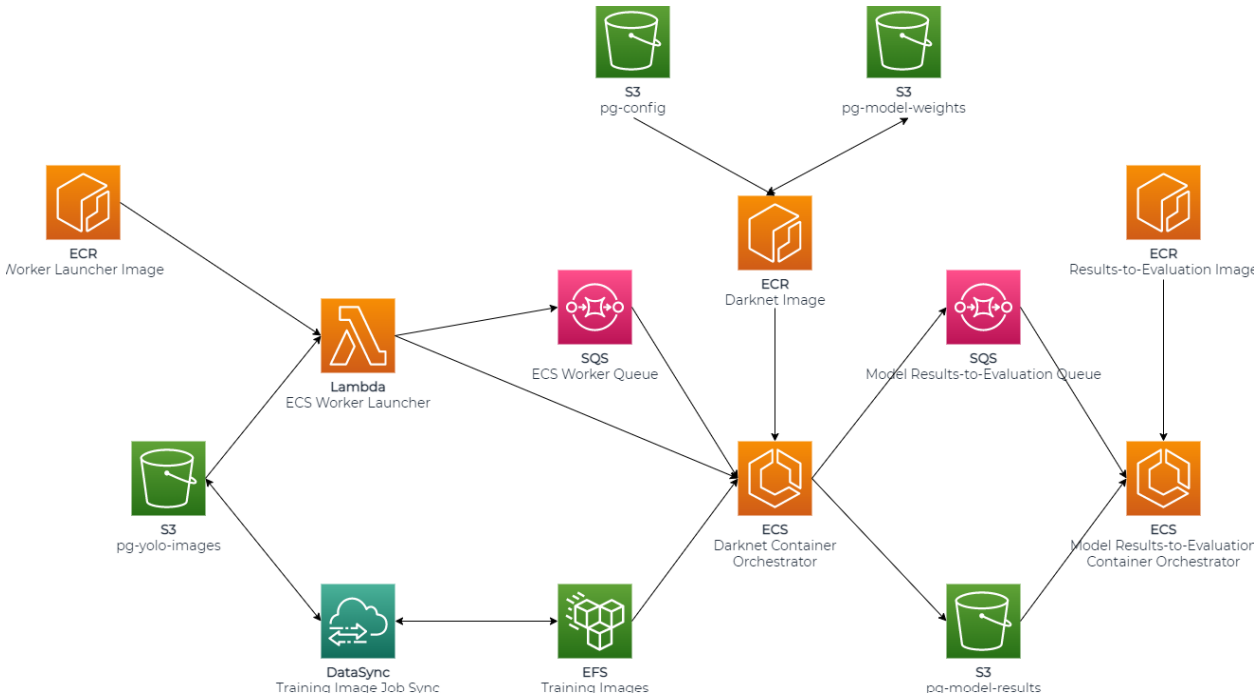
The second iteration of the architecture focused more on the model and analytics engine. The architecture focuses around Sagemaker, which encompasses the training environment that machine learning engineers can use to develop and iterate on the model. We chose to keep the training image data and the model configuration file in S3 (keep storage costs down), and model weights produced during training could be deployed to an EFS file system.



The architecture above provides a broader view, encapsulating both the model and analytics engine. It uses serverless containers and functions (Lambda and Fargate) to perform the evaluation to both save costs on GPUs and maintain simplicity in exchange for longer evaluation times. When a customer uploads footage, a serverless container would evaluate the footage, and the cloud function would interpret the results. It is slightly incomplete, however, as an SQS queue would be needed to transfer results between the container and the function.

While this pipeline was significantly better than the previous iteration, there was definitely a significant problem with this pipeline. While working on transferring the model workflow to the cloud, we realized that the way we were using Darknet's YOLOv3 was not compatible with Sagemaker. We couldn't install a lot of the dependencies required to compile the model. This forced us to pivot to a different solution.

Iteration 3 - A Container Based, Messy Workflow



The next version of the pipeline focused on creating a decoupled, container based system. Like the previous iteration of the architecture, this focused on the model and analytics engines. The engines are structured incredibly similarly - the only difference between the two engines is the container image used for the engine and the data retrieved as input.

Both engines utilize S3 to retrieve data from previous steps and store data for future steps. Data is passed to the container orchestration service (which contains the application running the compute step for that engine) using SQS. The orchestration service scales the number of containers based on the number of messages in the queue.

The data storage solution changed as well. We originally chose to utilize S3 to store training data, but we realized that there was no way to access the data for training without downloading it directly to the container. Since this dataset could be terabytes in size, we realized that this wasn't a viable solution. Instead, we chose to use S3 as an entry point for machine learning engineers to upload training data, and to use EFS (a network file system) for training. By using EFS, we can have the machine learning model train on the network instead of being constrained to an attached storage volume. In addition, multiple machine learning models can access the file system, which means that we don't have to store multiple unnecessary copies. To

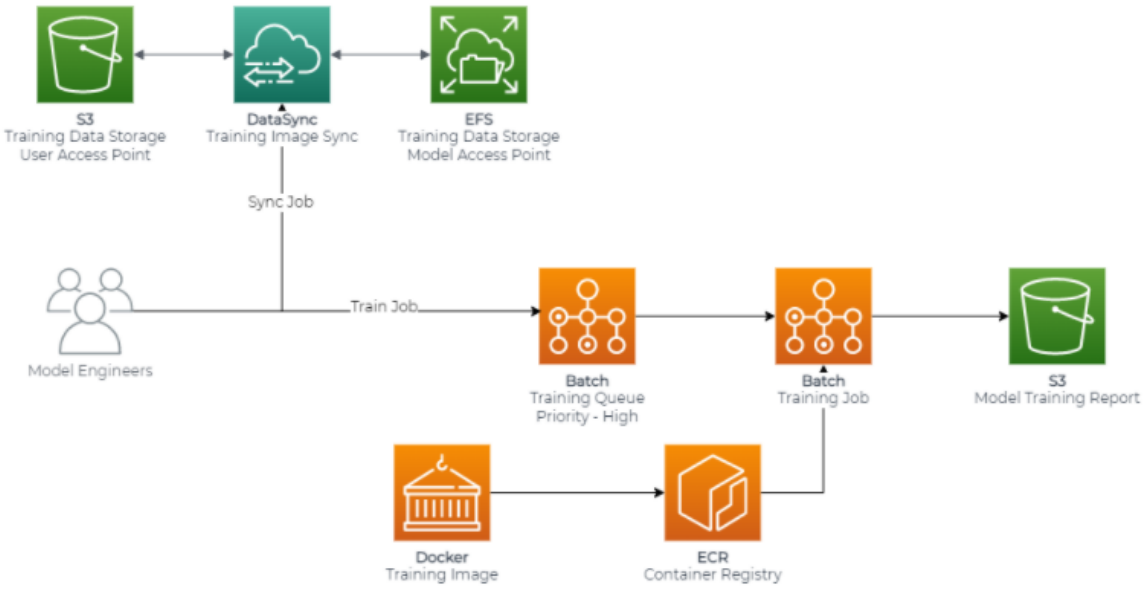
sync between the bucket and the network file system, we used a DataSync job to keep the file systems synchronized.

This pipeline was better, but still flawed. For starters, like the previous pipeline, it assumed that the training data and customer data would be stored in the same place, which isn't optimal from a convenience and security standpoint. This pipeline was also incredibly complex - there were a lot of moving parts that would have made implementing this pipeline challenging.

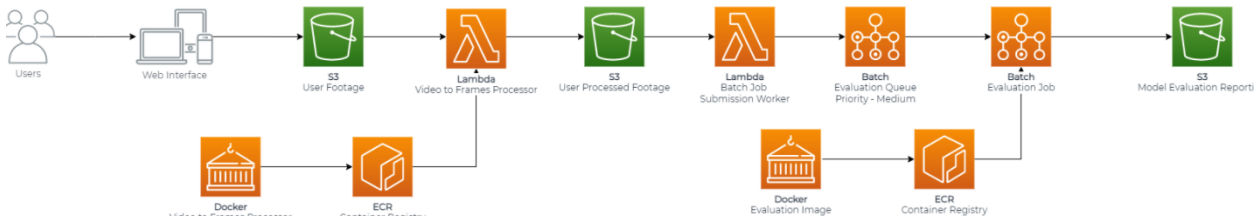
The Current Iteration - A Batch Compute Pipeline

The current version of our architecture is a vastly improved version of the previous architecture. It narrowly focuses on the model engine. Like the previous version, every compute step utilizes containers (now stored in a registry), and S3 buckets as intermediary storage steps. Similarly, the method of storing and replicating training data is replicated from the previous architecture.

However, architecture separates the training and user footage evaluation into two different engines (training and evaluation engines) for added convenience, simplicity, and security. In addition, the pipeline makes good use of AWS Batch, which is a batch computing service that utilizes multiple queues of different priorities to run different jobs (in this case, training and evaluation jobs) within the same computing environment.



The training pipeline is primarily used to iterate and improve on the model. Editing the model code involves modifying the “Training Image” container stored in the registry. Once a model is ready to train, they can run a single shell command that submits a training job into a high priority queue. After the computing environment completes the job, the machine learning models can view training weights and the model report, stored in an S3 bucket as output.



The evaluation engine utilizes a similar workflow. Once a user uploads footage into the "User Footage" bucket via the user interface, a cloud function immediately converts the video into a spliced set of images compatible with the machine learning model. From there a lambda function submits the evaluation job to the batch computing environment for training. Similarly to the training pipeline, the compute steps utilize containers, which can be modified and submitted to the container registry for automatic deployment.

This engine is currently in development. Instead of creating the infrastructure manually, we're currently creating a cloudformation script to deploy all of this infrastructure programmatically. This allows us to deploy, manipulate, and destroy the entire infrastructure across multiple regions through a single script instead of having to make manual changes to the script. The current progress of this script is available as a YAML file in the directory shared with this paper.

Works Cited

Zygourakis, Corinna & Yoon, James & Valencia, Victoria & Boscardin, Christy & Moriates, Christopher & Gonzales, Ralph & Lawton, Michael. (2016). Operating room waste: Disposable supply utilization in neurosurgical procedures. Journal of neurosurgery. 126. 1-6. 10.3171/2016.2.JNS152442.