Dependency aware Workflow Common Language in Kubernetes

Cosmin-Ionuţ Rusu under the supervision of Eric Bouillet
cosmin.rusu@epfl.ch, eric.bouillet@epfl.ch
EPFL, Switzerland

***Abstract*—We propose a dependency aware Workflow Common Language that can be used in Kubernetes for Data Science pipelines. Our architecture will run a recurrent cronjob that checks for the latest environment changes, and rerun only the updated and the corresponding dependencies.**

I. Introduction

Data Science has become an important focus of every organization nowadays. Big and small companies are hiring more and more data scientists to solve their specific business requirements or to better understand their data. The broad applicability of machine learning and the tools developed and released in the last few years made this possible. As with every new technology, the innovation wave stales at some point and the limitations become obvious. This is where the most important problems exists, and where researchers are focusing on.

A. Background

Almost all of the data science projects consist of a rather simple architecture. The high level view of the architecture consist of several parts, usually pipelined between one and another. For example, a simple linear regression model that predict prices of apartments in the San Francisco area would consist of multiple scripts chained together. In the first part, a process will gather and dump raw data in a storage system. This could be a crawler that analyses websites such as CraigsList, or Facebook Marketplace. Once that data is stored somewhere (for example, in an SQL database or a CSV file), the next step is usually a pipeline that transforms the raw data to clean, structured format. Some transformations might include parsing, filtering, joining multiple data sources or replacing missing data (this process is usually denoted as *cleaning the data*). We already can see the dependency between these two rather isolated tasks: the first one needs to finish before the second one can start. A next script might be in charge of loading the processed data in the memory, and using it to train a linear regression model. The output of such a script is usually the optimal parameters (weights) of the trained model. Nevertheless, for any model to be used in production, another *deployment step* (which might be optionally preceded by a validation step) will be finally triggered. Such a deployment task would usually update the parameters of a model that runs in production, serving requests to make predictions.

First, every step of the above described pipeline might take a lot of time to compute and might also involve network requests being made (network latency and failures). In case of any changes in the data or source code for the pipeline, all the dependent tasks needs to be rerun again. This is rather error prone, and especially hard when working in a team environment (not to mention distributed teams where people across different timezones need to collaborate together). This manual approach clearly does not scale and a software tool to make these processes easier for the data scientists is desired by the community.

The end goal is to have *reproducibility*. That is, having the ability to run the same code, using the same data, under the same runtime environment (software and hardware) and running the same workflows. That would allow researchers to easily distribute their work in a *reproducible* fashion and would allow the community to focus less on these aspects, and more on their respective fields of interest. Several solutions already exists, and the main challenge here is being able to cache and rebuild only the necessary parts without rebuilding the whole workflows every time, wasting compute resources.

B. Scheduler

To overcome this, Swiss Data Science Center's priority is to develop integrated systems that allows developers to easily create, build, test, develop and share data science projects. To this extent, this project describes the implementation of a declarative workflow common language that enable data science pipelines scheduling in a Kubernetes cluster. Kubernetes is a cloud-agnostic container orchestration platform for deploying applications. Simply put, Kubernetes makes it easy to deploy and scale containers in a simple declarative fashion. In the initial project [1], only code changes were considered, but dependencies changes, such as Docker image changes, or library changes were not captured by the project. Our aim was to extend the project to be able to capture the dependency changes across the codebase. The approaches considered, their pros and cons, are being described in the following sections.

C. Previous work

In an earlier semester project [1], code changes were detected by getting the commit hash of the repository and comparing it to the previous known commit hash. If the hashes are different, one could look at the edited files, get all the tasks that need to rerun, and rerun them, along with the dependent

tasks. Creating this DAG of dependencies is possible because every project has to clearly declare their tasks and dependency structure in a dependency file. The limitation here is the fact that some underlying library or environment dependency might change (for example, the docker image might change completely independent of code changes). We would like to have a way to detect them early on. Since all the jobs currently have a Dockerfile attached to it and due to the way Docker images are designed (by having a layered structure), we think we can use a docker diff tool that will be able to catch any subsequent change, including the dependencies, base image changes or code changes.

In the current state, all the jobs are under one repository. Each job is under the file path *src/jobName*. At the root of the repository there is a dependency file specifying each job, what inputs do they take, the output they create and the Dockerfile to run that job.

On every git push, a request is being made to a central server that uses the hash commit to check for the latest flow that was run, and in case of changes, it looks over each file that changed, and create the DAG of only the corresponding jobs. This current implementation lacks some important features. First, the Docker file base (FROM X:latest) might change, or 'apt-get install package' might install a newer version of the library. It is very important to take this into account, not only the code changes. For example, a library might change an API that is used in the code and will not work anymore. We want to be able to catch these potential problems early on.

II. State of the Art

While researching the already existing solutions, we came across some useful tools, but none of them were, on their own, suitable for our particular problem. In the end, we took multiple ideas from each of the following described topics and came up with an original approach that achieves all of defined goals.

In the next sections, the first two tools (*Docker Diff* and *Docker Push If Changed*) are related to detecting environment changes, while the last two (*Databolt Flow* and *Argo*) are implementations of schedules for task dependency workflows.

A. Docker Diff

Docker diff [2] is a command line tool that comes by default with Docker CLI, and it is used to inspect the changes to files or directories on the filesystem of an existing container. While useful, this requires to have the image already built on the cluster, and we don't need that granularity in our project, we just want to quickly find out if an image was changed or not from the last time we run that image.

B. Docker Push If Changed

Docker Push If Changed [3] is a tool developed at Yelp. They build and push base images daily to their own internal registry. Doing that enables them to get the latest security patches when they become available. While clearly this is important, some changes might not be very meaningful for them. For example, a simple change to a configuration file or a small update to a package that they don't use will not be very useful for them, and this kind of changes can be avoided, saving a lot of computation time that would otherwise go into building Docker images every day. To overcome this problem, they've come up with this heuristic-based approach based on the *docker history* - a command that returns hashes of added or changed files - and *dpkg* - a package management system in the free operating system Debian and numerous derivatives.

Yelp clearly tries to solve a similar problem to ours, but is clearly tailored for their specific needs, and this approach would not satisfy our requirements. This inspired us to think about a solution that is specially suited for our needs, rather than looking around to existing tools.

C. Databolt Flow

Although this tool is not related to the problem of detecting changed dependencies, it is still relevant for our higher purpose which is to have a Workflow Common Language running on top of Kubernetes to manage data science pipelines.

Databolt Flow [4] is a python library which enables data scientists and data engineers to build complex data science workflows easy, fast and intuitive.

Common workflows in data science involve chaining and combining together parameterized tasks passing multiple inputs and outputs between each other. Most of the times, the inputs and outputs are stored in either dataframes, files or databases, requiring the developer to keep track of where everything is. In such an environment, it quickly gets hard to rerun tasks with different parameters without rerunning the entire long-running pipelines. This approach clearly does not scale well, and Databolt released this tool as an open source project that makes it easy to chain together complex data flows and execute them. Loading input and output data for each tasks is quick, which makes the entire workflow very clear and intuitive.

Databolt Flow package allows to build a data workflow consisting of tasks with declared parameters and dependencies. It can easily check dependencies and the execution status of each tasks. It is fault tolerant since it allows to continue workflows even when some tasks fail. In case some parameters, code or data changes, it can intelligently rerun the workflow. The output data can be saved to Parquet, CSV, JSON, pickle or even in-memory. Another strong feature of Databolt Flow is the fact that it allows data scientists to easily share their workflows with each other, making their development processes scale better.

While this sounds like the perfect solution, it will still run on a single machine and hence is only vertically scalable, requiring more resources on one machine, which can quickly become very expensive. Rather, we would be interested in a horizontal scalable solution, inspired by microservices architecture, where the pipelines can run on top of Kubernetes, each task in isolated pods. A pod is a single unit of computation inside a Kubernetes cluster.

D. Argo

Argoproj [5] is a collection of tools for faster Kubernetes deployments. While Kubernetes accepts a declarative file that describes a deployment or service, as a project grows, managing service dependencies or building complex pipelines becomes tedious. Argo provides three tools to help developers effectively deploy applications to Kubernetes:

- Argo Workflow - a container-native workflow engine
- Argo CD - a declarative continuous delivery and continuous integration tools
- Argo Events - an event-based dependency manager

We will describe in detail just the Argo Workflow tools as this was the only tool used in our architecture. Argo Workflow is an open-source container-based workflow engine for running and managing parallel jobs in Kubernetes. It allows to define workflows where each task in the workflow is a container and to model multi-step workflows or even capture the dependencies between tasks using a directed acyclic graph. It is successfully used by companies such as Google or Adobe to define and run intensive jobs for machine learning or data processing pipelines in a short amount of time. Other companies uses Argo for building continuous integration / continuous delivery pipelines without complex software development products.

Using Argo is simple and intuitive. One needs to deploy Argo to the Kubernetes cluster, and then provide a declarative file defining the workflow. This file contains the list of tasks with the Docker image and the command to run inside the container, task dependencies and other metadata associated with the task (such as name, tags).

Since Argo can easily orchestrate highly parallel jobs on Kubernetes, and our needs involves that, it was a natural choice to use Argo as our underlying mechanism of writing and running Data Science Workflows in Kubernetes.

III. Design

Our main goal was to come up with an intuitive declarative workflow language that can be used by data scientist to write, debug, run and share data science pipelines. Since these tasks usually take a lot of resource and time to process, we wanted the workflows to be able to run in a cluster of machines. Given the popularity and increasing adoption of Kubernetes [6], we've decided to base our implementation around it.

The initial project already implemented a code changes aware workflow common language, and in this project, we've extended it to also be dependency aware. The main motivation is to catch errors or incompatibilities early on, without affecting productivity or revenue in a business model.

A. Proposed solutions

1) Continuously rebuild the docker images: On every push, besides the jobs that had file changes, we can rebuild the Docker images and compare the hashes of the images, before building the entire DAG of jobs that need to rerun.

The advantage here is that we are always having a consistent and reliable state of the jobs. We know, after each push that either libraries or code did not break anything and the changes are reflected.

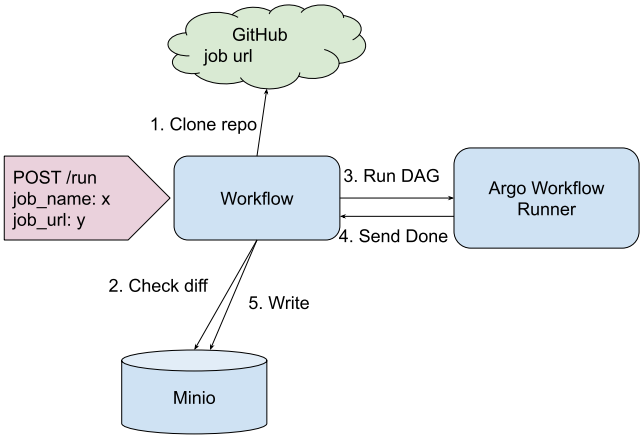

Fig. 1. Main steps to check for code changes

A huge disadvantage that his have is that the docker builds might take a long time to process. It might make the whole workflow too computationally heavy. It might also mean that the more jobs we have the more unscalable this solution is.

2) Cronjobs that run on a daily/weekly/monthly basis: Another, more scalable solution, would be to have a cronjob that rebuilds the docker images, look for the changes and rerun the DAG inferred by those changes. If run daily, this might be able to catch library changes early on, while doing it on a weekly or monthly basis might introduce a delay before the bug is caught.

A useful advantage is that the cronjob can run overnight and so it doesn't need to be very fast, and will not break users productivity. A slight disadvantage is that this approach might introduce some delay in the errors report if not run daily, the repository can be in an inconsistent state at some point.

In the end, we have decided to use the second approach, to run a cronjob on a daily basis that reruns the workflow for every changed images. Our dependency file specifies, for each task, the dependency list, a docker image, and a command to run the task inside that docker image. The *ID* of the docker build can tell us whether or not an image has been changed or not, provided that we have a way to check the last known *ID* of the docker image that was used to run the task in a previous run. For a fast and scalable way to check that, we have used Redis - an in-memory key-value data store. The concrete details will be presented in the next section.

B. Architecture

Before diving into the architecture, it is useful to get a sense of the tools the architecture is built upon, and the underlying principles behind each of them, so that the architectural decisions are clear.

1) Kubernetes: Kubernetes [6] is an open-source container orchestration platform widely used nowadays to achieve scalable software solutions. The highly adoption of the microservices architecture contributed to the growing popularity of Kubernetes.

Designed by the same principles that allows Google to scale their massive infrastructure, Kubernetes can scale without scaling the dev ops team. It allows automatic deployment, scaling, and management of containerized applications. By grouping containers making up the application into logical units, Kubernetes makes it easy to manage and discover applications running in a cloud infrastructure. In addition, Kubernetes is cloud-provider agnostic. By abstracting away the underling hardware, it can run on Amazon Web Services, Google Cloud, Microsoft Azure or even bare metal servers.

Kubernetes coordinates a cluster of machines that are connected to work as a single unit. This abstraction in Kubernetes allows the developers to focus on building the application itself and not worry exactly on which machine in will be running. This new kind of deployment requires the app to be containerized and it allows for a simple, human-readable declarative language that, given to Kubernetes, can deploy, heal and replicate an application.

Two types of resources are part of Kubernetes:

- The Master coordinates the cluster
- Nodes are the workers that run applications

When deploying Kubernetes to your cluster, these two type of processes are being created and they will gracefully handle all your deployments. Kubernetes even gives containers their own cluster-wide IP addresses and a DNS name for a set of containers, and can load-balance across them. Whenever containers fail or crash, Kubernetes will attempt to restart, replace or reschedule another one. In that sense, it is self healing and requires little human input when errors arise.

Given all these features that Kubernetes have, the decision to implement a Workflow Scheduler on top of Kubernetes was obvious. Not only it will allow access to massive scale, but it is also robust to failures which happen all the time in data science workflows. Another upside of this approach is that each data science pipeline or task can be isolated and self-contained.

2) Redis: Redis is an open source, in-memory data store, that can be used as a database, cache and message broker. It can support complex data structures such as strings, hashes, lists, sets, sorted sets, bitmaps or even streams. It achieves high performance due to the in-memory dataset it works with, but it also supports different levels of on-disk persistence. Redis also support master-slave asynchronous replication.

In the context of our implementation we have used Redis as a key-value store for strings. It is important to mention that our implementation uses only one master node, for simplicity, but multiple read-only nodes can be spawned inside the cluster in order to make this service highly available.

3) MinIO: MinIO is an open source object storage server compatible with Amazon S3 cloud storage service. It is used by engineers to store unstructured data such as photos, videos, log files or backups. The size of the stored files can go as far as $5TB$. Similar to Redis, MinIO server is light enough to be bundled with the application stack.

Our architecture consist of two main flows. The first flow is the flow that is responsible for detecting code changes. The respective architecture can be seen in Figure 1, highlighting the main steps. The first and foremost step is to clone the repository for the latest data (usually the master branch, but this is just for simplicity, any branch would work). Then, the *Workflow Controller* checks for the latest known commit that previously run in our system. By diffing these two, we can get the list of files that changed from the last run. Next, using the files we can generate the subdag of tasks that have been changed. This is easy to assemble because every repository has a list of tasks stating every dependency, the Docker image and the command to run that task. Moreover, given our strict folder structure, we can determine based on the file paths which jobs needs to rerun. After building the Argo file containing the underlying dependency structure of our subdag, we submit that to the Argo Workflow Runner which spins pods for each tasks inside the Kubernetes cluster, as described above. As soon as every task is done, the controller is notified which finally writes back to MinIO the fact that we have successfully run the latest commit. Next time, the next diff will be made against this commit instead.

The second flow can be seen in Figure 2. This flow is designed to verify for dependency changes that are not caught by the code itself. For example, some version of a used library would be changed, or the respective Docker image might have been changed and the exposed API is now incompatible with the code from the repository. This flow would run daily and in case of any problem, it would notify the data scientist using some error tracking tools such as Sentry or Rollbar. The first step is to clone the latest version of the workflow. Then, we inspect the dependency file for Docker image changes. We know for each task what Docker image the task depend on. In step 2, we use the *docker inspect tool* to get the *ID* of the docker image, and in step 3, we check if that is different from the one we used on the last run (stored in Redis). If a difference exists, we mark that task as being changed. At the end, we ensemble all the changed tasks and their respective dependent tasks in a DAG. We send that DAG to Argo, that will run the tasks in a parallel fashion, while still respecting the dependency structure. When Argo is done, we update our Redis instance with the *IDs* of the images.

IV. Implementation

Since Kubernetes strongly encourages that, we have implemented our Architecture in a microservices fashion, allowing us to isolate different parts of the application. This allows for easy testing, less dependencies, and is, overall, a scalable approach to write code. For convenience, we have created a *deploy* script that one can use to setup or teardown the cluster. It loads all the required services such as MinIO, Redis, Argo and Workflow-Controller.

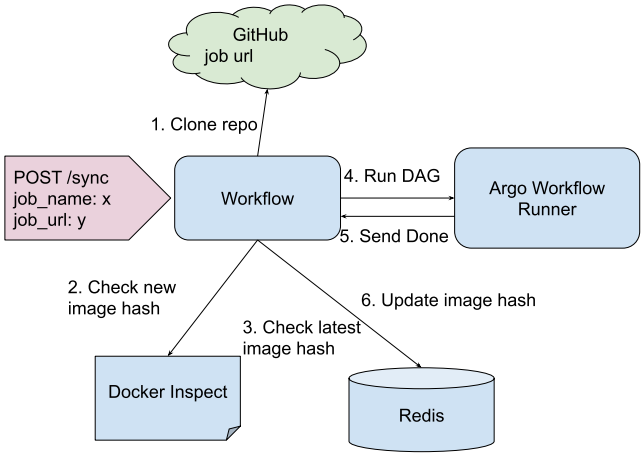

Fig. 2. Main steps to check for dependency changes

Route	Params	Responses
/sync	{job_id*, job_url*}	200 - Ok, 201 - No content
/run	{job_id*, job_url*}	200 - Ok, 404 - Not found

TABLE I
EXPOSED REST APIS

The Worflow-Controller is where all the logic and the orchestration is happening. It knows how to talk to any other service and it exposes the REST APIs to the user.

The *sync* flow, responsible for detecting dependency changes, follows the following algorithm. First, it listens for a *POST* request to sync. On every request, it gets from the *HTTP* body the url of the job and the job name. Every job contains the collection of tasks and their dependency. If the job has not previously run, it will rerun it as a whole, for the first time.

On subsequent requests, assuming the job has already been processed at least once, a couple of metadata is persisted, both in MinIO and Redis. First, in MinIO, the latest known commit is persisted. Second, in Redis, we store the *ID* for every Docker image that was used in the pipelines. Since on every push, the image *ID* will change, it will be easy to check if one images has been updated and it must be rerun. After successfully running, the latest data is again persisted and the system waits for subsequent requests.

Table I describes the APIs for the two respective flows: the code changes flow and the environment changes flow. For both, the *job_id* and *job_url* are the required *HTTP POST* parameters. The API supports *JSON*. An example of *job_id* can be *my-job* and the *job_url* would be *https://gitlab.com/rusucosmin/job*. The source code of this implementation can be seen at https://github.com/rusucosmin/workflow-controler. [7].

V. EXPERIMENTS

We have run this workflow for a simple Machine Learning workflow that contains four tasks. The first one preprocess some training data; the second one splits the processed data into two sets: the training set and the testing set; the third one trains a machine learning model based on the training set; and finally, the last one computes the score of the trained model. We can see the chain-like dependency structure here, which, using our tool, helps the data scientist focus on every individual task, and the *integration* part is automated. In some sense, we can say that our tool is an implementation of a Continuous Integration / Continuous Delivery for Data Science pipelines.

The dependency file associated with this project contained a Docker image tagged as *latest*. From the last time the workflow run, we've changed and pushed an updated image, hence modifying the underlying dependencies. As a consequence, when running our *sync* flow, we detected that discrepancy and updated accordingly all the dependent tasks. It is nevertheless easy to see that the simplicity of this approach makes the logic of detecting the DAG of changes run very fast, and the complexity of the *sync* flow is the same as the complexity of the *run* flow which is in charge of detecting code changes in the codebase itself, not underlying container.

VI. DISCUSSION AND FUTURE WORK

One idea that one can use to further develop our tool is to use webhooks from the different Docker registry that supports that. Every time we depend on a Docker image, we can subscribe to the *push* event on that image, and be notified that it has changed, and rerun the tasks that depend on that image and the inferred dependent tasks. This would allow us to incrementally update the state of our workflow as soon as the images have been built. One potential problem here is, assuming tasks take a long time to complete (training huge machine learning models such as CNNs can take even weeks), how do we make sure we don't waste resources running everything again and again. In some sense, we could use the idea from [3] to heuristically check if a diff is worth a rerun or not.

Kubernetes also supports Cronjobs inside the cluster, but for simplicity we did not use it. Such addition can be added to the deploy script to automatically support syncs on a recurring basis, when the cluster initializes.

REFERENCES

[1] B. Liamarcia, "Project workflow kubernetes," https://github.com/project-workflow-kubernetes, 2018.
[2] Docker, "Docker diff," https://docs.docker.com/engine/reference/commandline/diff/, 2019.
[3] Yelp, "docker-push-latest-if-changed," https://github.com/Yelp/docker-push-latest-if-changed, 2019.
[4] Databolt, "Databolt flow," https://github.com/d6t/d6tflow, 2019.
[5] Argo, "argo," https://github.com/argoproj/argo, 2019.
[6] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," 2016.
[7] C. Rusu, "Workflow controller," https://github.com/rusucosmin/workflow-controler, 2019.