Outline of a CI/CD Pipeline for Video Processing

Mark Bools

2022-01-31

Last Modified: 2022-01-31

Abstract

In this article we are going to discuss the outline for a Continuous Integration/Continuous Deployment (CI/CD)[1] pipeline to process videos. Specifically, we are going to discuss a pipeline to take a main video, an edit list, some inserts (ads, promos, etc.), some title cards (opening, closing, transitions, etc.), and some metadata (like the description, title, author, tags, categories, etc.) and use these to assemble a number of final versions to be uploaded to various target services (YouTube, Vimeo, lbry.tv, etc.).

The basic flowchart for our pipeline looks something like this.

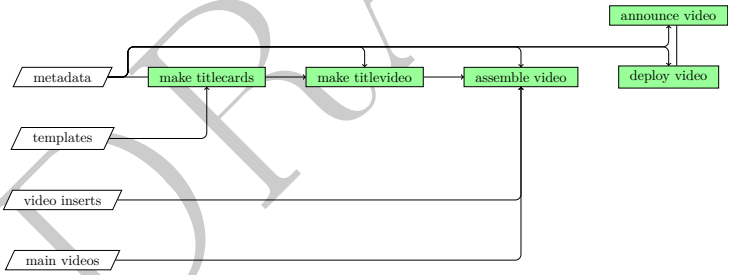

Figure 1: Video Process Pipeline Basic Flowchart

We need to take care not to create a pipeline that requires massive storage. A typical CI/CD pipeline starts with a change in something like a git repository, perhaps the source code for our project. This implies that the resource holds *all* the source for the pipeline. In our video workflow this is probably not practicable as the resources are copied into the local fly worker container. Copying a growing set of videos would swiftly result in gigabytes of files being copied through our pipeline; not cool.

To avoid the potential growth of data we will key everything from our metadata, using this as control to trigger a pipeline run and the content of the

[1] Automated process for building, testing, and deploying assets.

metadata will drive tasks in our pipeline to selectively upload other materials from remote 'pools'.

We will still be importing some resources in their entirety:

metadata This may end up being several files, or even several resources. Things like Edit Decision List (EDL)[2] used to determine cut points for inserts, metadata for deployment and announcements, data describing which videos are to be edited and which videos are to be inserted in promo or ad slots, that sort of thing.

templates This resource provides things like the LaTeX template used to generate title cards.

On the output side. Do we want to keep all the generated video locally? Not really. So long as all the source material exists we do not really need to keep the final product (it is after all being uploaded to remote services), but we do want to have some may of verifying remote copies (i.e. those that have been deployed so that we can determine if they need to be redeployed).

The Development Environment

We'll start with a Python development environment I anticipate several tools we need will be better as Python scripts than Bash scripts.

Obtain the base Python VM[3].

```bash
git clone https://gitlab.com/python-utils2/pythonvm.git videoprocessor
```

You may need to change the `Vagrantfile` to map the `concourse` port to something other than `9080`. If you already have something mapped to that port `vagrant` will complain when starting the VM.

Move into the new VM (this first run will take several minutes as the system is updated and various Python tools are installed).

```bash
cd videoprocessor
vagrant up
```

Once your VM is up and running, `ssh` into it.

[2] A list of 'edits'; cut, insert, delete, etc. Usually presented in a standard text format such as CMX 3600

[3] If you want to be sure you're using the setup in this tutorial then `git checkout -b v0.1.0` once you're in the workspace.

```bash
vagrant ssh
```

From this point on we will work almost exclusively in this VM, so I will highlight only operations *not* run in the VM/

Taking a look around

You will find that this VM has a few things already installed.

Concourse This is the 'doer of things' (their description) that we will use to manage and run our build pipeline.

Docker Used to run the Concourse CI system on this development VM. We will also use this to create containers to use in the Concourse pipeline.

Python We will use this to create the more complicated scripts in our pipeline.

Poetry A tool for Python package management.

A few commands help confirm everything is installed.

```bash
docker --version
docker ps
fly --version
python --version
poetry --version
```

And you should see output something like the following (I've allowed the version to 'float', so you're versions will likely be different to those shown below)..

```docker
Docker version 20.10.3, build 48d30b5
CONTAINER ID   IMAGE                  COMMAND                  CREATED       STATUS       PORTS                    NAMES
fcf478d39d95   concourse/concourse    "dumb-init /usr/loca..."   2 hours ago   Up 2 hours   0.0.0.0:8080->8080/tcp   concourse_ci_concourse_1
79df83906661   postgres               "docker-entrypoint.s..."   2 hours ago   Up 2 hours   5432/tcp                 concourse_ci_concourse-db_1
```

```
fly -version
```

```
6.7.5
```

```
python -version
```

```
Python 3.8.2
```

```
poetry -version
```

```
Poetry version 1.1.4
```

The First Build Task: Making title cards

This is fairly straightforward; take a LaTeX template, feed it a title and logo and have it produce a PNG titlecard. Our inputs are therefore:

- LaTeX template
- Logo
- Title string

Our outputs are:

- A title card PNG

The tools required to transform our inputs to our outputs are:

- LaTeX processor
- ImageMagic, to convert the LaTeX output (a PDF) to a PNG

We could create a specialised Docker Image for this, but I will reuse my general purpose document generator image `saltyvagrant/latex-docker` (this is itself a custom made image, but is not specialised in the sense that it contains almost the entire LaTeX Live installation making it a very large image, but I reuse it for many Concourse tasks).

I have a logo already available (I prefer to preserve all images in SVG format until final conversion to a target format). The LaTeX template it fairly simple.

titlecard.tex

```
\documentclass[oneside]{standalone}

\usepackage{saltyvagrant}

\usepackage{graphicx}
\usepackage{xcolor}
\usepackage{tgadventor}
\renewcommand*\familydefault{\sfdefault}
\usepackage[T1]{fontenc}

\SetWatermarkScale{ 0.5 }

\begin{document}
  \begin{minipage}{572bp}
    \includegraphics[height=720bp]{logo_and_mascot/salty-logo-avatar-clipped.png}
  \end{minipage}
  \begin{minipage}[c][720bp]{705bp}
    \fontsize{100}{130}\selectfont
    \color{svblue}
    \begin{center}
      \getenv{TITLE}
    \end{center}
  \end{minipage}
\end{document}
```

The title itself is provided from a metadata file and inserted into the LaTeX process using the `TITLE` environment variable. We will discuss the LaTeX process in more detail in a later article in this series.

Where to start a build

One decision you need to make when designing build systems is 'where does the build start?' By this I mean what do you consider the raw material for your build. With straightforward software builds the most obvious raw material is your source code. Less obvious raw materials are things like internationalisation strings, embedded images, and the build itself (e.g. the `Makefile`). In the real world it is seldom this obvious.

Even in a relatively straightforward build you have complications to consider. If you're building for multiple target platforms then it is likely that your `Makefile` will be generated using configuration specific to the target platform. This either moves your raw materials back one level (an additional configuration and `Makefile` generation step) or you manually create several `Makefile`s and thereby shift some of the burden from the build process to whomever makes

the builds (not a good option, but an option).

In the case of video processor some of the raw materials are also apparently obvious; the main video and audio, the metadata (title, description, etc.). Other, less obvious, sources include the build system itself. The various scripts and tools used to perform the build. These scripts may well also be making decisions about what and how to perform the build.

Beyond these 'obvious' points we may decide to create the title videos manually, so these now become sources rather than the metadata and templates etc. proposed above. Where you start your build will often be a matter of judgement, experience, project requirements, and technical limitations. A simple example from our current build; why not start from raw video and audio footage and have the build cut together the main video using an EDL also provided as source? This is entirely possible but also impractical for our purpose. Editing the main video together is simpler using an Non-Linear Editor (NLE)[4], this video processing pipeline is performing the tedious tasks that can be easily automated whereas pulling raw video and audio together into a coherent video is (at least for me) much simpler using manual tools. (That said, if I were to be truly bloody-minded I could have the NLE generate the EDL and then have the build effectively repeat the main video cut, but that seems redundant to my workflow.)

Musings on metadata

We now encounter our first issue. Generally we want changes to the 'make title' task sources to trigger a build (we can trigger tasks in several ways, but source changes are the most common). Our source are; the metadata, the logo, and the template. Here's the issue, if the template changes do we *really* want to trigger the task? If so, should it rebuild *all* title cards using the new template? (Bearing in mind that creating new title cards implies that any videos that used those title cards should also be regenerated.) What about if the logo changes? Same issue as the template. What about the metadata? The only piece of metadata we want to trigger a new title card is the title, any other changes can be ignored for now.

I think that generally I do not want template or logo changes to trigger complete rebuilds (maybe later I'll look at manually triggering these wholesale changes, following a rebranding for example, but day-to-day it's safe to assume changes to template or logo should effect only new title cards).

That leaves us with metadata changes. Title cards come in two types; those specific to a video (for example the lead title card) and more general 'inserts' (like '... and now a word from our sponsor') that may be reused in several productions. Should we therefore maintain metadata as one large 'database', or 'one data file per production', or 'one data file per title card'?

[4]Tool/technique for editing video that allows the user to choose any point in the project to work on.

We could make a case for any of these solutions. I'm going with one per production but allowing a 'production' to be one or more stand alone title cards. This seems like a reasonable compromise to me as it keeps everything more or less in one place for most videos but allows some flexibility to generate stand alone items too. The corollary to this being that each production can refer to other productions (so video A can use a title card produced by title B). I may live to regret this decision, but this is an iterative process and we need somewhere to start.

Why not go for one big lump of metadata? Mainly I think it would quickly become unwieldy and difficult to edit by hand (no doubt leading to yet another utility to maintain just the metadata and that roads leads to using databases, all of which seems like a bit of overkill at the moment). Also, the design of this whole pipeline means I may well not want to keep all metadata feeding in to the start of the pipeline (more on this in a while).

The new task

Our new task has one more input, the script to perform the build. It is possible, when your task's image is specialised enough, to continue with the simple 'one command' but this task needs something more complex. Specifically, we need the task to scan metadata and make decisions about which title cards to produce.

Title Videos

We have a choice here. We can either make short stand alone videos from our title cards or we can use some `ffmpeg` magic later on to insert the title cards as short static section.

If we make short videos now it may reduce time rebuilding later (for example reusing the titles in another project) but this also implies storing the title videos somewhere so they can be reused. This seems unnecessary to shave a few seconds of downstream activities.

If we insert the titles 'on-the-fly' we make the assembly of the final video more complex (and consequently potentially more error prone) and more demanding on our downstream activity (the video must be reconstructed each time).

I think that, for now, we will create them on-the-fly and only create them separately if this proves troublesome.

Assembling the Final Video(s)

We may be assembling several version of the final video. Different ads, sponsors, or promotions being placed into, or omitted from, the main video according to the target platform. All this to be driven by our metadata.

Deploying the Final Video(s)

Our metadata can provide common information for each remote source (such as authentication details). It can also present specific information for each target, for example descriptions and 'flags' (those pop-over links available on platforms like YouTube).

Announcing New Videos

Once the videos are deployed we can announce their availability. As with deployment our metadata can provide authentication details for the various announcement channels along with details of what each announcement should contain. One of the main things in this section is to ensure we do not announce multiple times if something goes wrong.

Final Thoughts

In this article I've started thinking about the CI/CD process for making and publishing videos. Many of the details remain to be established but the broad outline seems like a good start.