UNIVERSITAT DE BARCELONA

Facultat de Matemàtiques i Informàtica

UNIVERSITAT DE BARCELONA

Design and Implementation of an DL-based Video Processing Pipeline

Author:
Carla Morral I Martín

Supervisor:
Sergio Escalera Guerrero

End-of-Degree Project

13 June 2022

Abstract

This project was born as an internal need at Amazon for being able to automatically or semi-automatically perform Quality Assurance checks to videos featuring books in order to eliminate or reduce the human time spent on doing video QA. The proposed workflow uses a variety of tools and algorithms that will be detailed, including the use of deep learning. This is a good example of a real-life use-case in which deep learning is used to help improve the humans life. This video processing pipeline is composed of some checks to ensure that the videos are correctly rendered and is already integrated to the major architecture Waldo Media Factory, and ready to be used in production.

Contents

Chapter 1

Mathematics of Deep Learning

1.1 An Introduction to Machine Learning

Machine Learning is increasingly becoming a key factor in our lives. The necessity of building methods and models that can learn from real situations and produce reliable outputs is expanding through a high diversity of fields, such as Natural Language Processing, Computer Vision or Biology. Loosely speaking, it is about producing computer programs without writing them. Informally, these types of models try to learn from a huge set of data called *Training Data* to then predict or make decisions about data in the same distribution. In addition, Machine Learning has acquired a reputation of having some *Black Magic* behind, especially the neural network-based models. To my view, this is because of the huge offer of libraries offering ready-to-use implementations of the state-of-the art models and therefore, the lack of a basic understanding of the theoretical background behind these models. Moreover, it's not trivial to understand why a tiny tuning of some hyper-parameters on the model can substantially affect their performance. As a result, the purpose of this chapter is to formally expose the basics of Machine Learning, as well as an interesting point of view, PAC Learning.

1.1.1 Models an Optimization

The two main branches of Machine Learning are *Supervised Learning* and *Unsupervised Learning*. In the former, the learning algorithm receives labeled data and the model is trained so that it can predict the label of any unseen piece of data. Some applications of this algorithm include classification tasks, such as for example, determining the genre of a book based on the book cover. This might be useful when for instance, on an e-commerce website, the book publisher does not provide this type of metadata and we want to place that book on a certain section of the online store depending on the genre. On the other hand, on the latter approach the algorithm doesn't receive any type of label associated to the input data. In contrast, it is the model who has to infer patterns of the data and draw conclusions about it. A real-world use case of *Unsupervised Learning* is color quantization, a process that consists on reducing the number of colors present on an image. This process

is usually done using *clustering*, an *Unsupervised Learning* algorithm. This is an interesting problem because for instance, given a painting, this algorithm can output a color palette of the n most dominant colors on the painting. This report will focus on exploring some characteristics of *Supervised Learning*.

Supervised Learning

Definition 1.1.1. *Given a finite sequence $S = ((x_1, y_1), ..., (x_n, y_n)) \in \mathcal{X} \times \mathcal{Y}$, where y_i is called the label corresponding to x_i, a **learning algorithm** is a map*

$$\mathcal{A} : \cup_{n \in \mathbb{N}} (\mathcal{X} \times \mathcal{Y})^x \to \mathcal{Y}^n$$

where its range is the set of functions $h : \mathcal{X} \to \mathcal{Y}$. For instance, given a set $S \in \mathcal{X} \times \mathcal{Y}$ of training data, a learning algorithm outputs a function $h_s := \mathcal{A}(S)$, called **hypothesis**.

With regards to the pairs (x_i, y_i), they will always be treated as pair of random variables (X_i, Y_i), that are identically and independently distributed according to some probability P over $\mathcal{X} \times \mathcal{Y}$.

Definition 1.1.2. *A **loss function** is a map*

$$L : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$$

that given a pair $(h(x_i), y_i)$, where h is the function output by a learning algorithm, measures the distance between these numbers.

Definition 1.1.3. *The **risk** associated with an hypothesis h is defined as the expectation of the loss function:*

$$R(h) = \mathbb{E}[L(h(x), y)] := \int_{\mathcal{X} \times \mathcal{Y}} L(h(x), y) dP(x, y)$$

The goal of a learning algorithm is to minimize the *risk*. Nevertheless, we can't directly evaluate the above expression because at first, the learning algorithm doesn't know the distribution $P(x, y)$. That's why we use the below formula to approximate the *risk* value.

Definition 1.1.4. *The **empirical risk** of an hypothesis h is defined as:*

$$R_{emp}(h) = \frac{1}{n} \sum_{i=1}^{n} L(h(x_i), y_i)$$

Consequently, the goal of the learning algorithm is to choose a hypothesis $\hat{y}$ such that:

$$\hat{h} = \operatorname*{argmin}_{h \in \mathcal{H}} R_{emp}(h)$$

where $\mathcal{H}$ denotes the range of the map $\mathcal{A}$.

Thus, the learning algorithm defined by the empirical risk minimization (ERM) principle consists in solving the above optimization problem.

Depending on whether $\mathcal{Y}$ is continuous or discrete we can distinguish two types of supervised learning problems: *regression problems* and *classification problems*.

Regression

Definition 1.1.5. *We say that a learning problem is a* ***regression problem*** *when $\mathcal{Y}$ is continuous. For instance, $\mathcal{Y}$ can be a height or a salary.*

Definition 1.1.6. *Given a hypothesis h and the quadratic loss $L(y, h(x)) = |y - h(x)|^2$, the L_2* ***risk****, also known as the mean squared error, is defined as:*

$$R(h) = \mathbb{E}[L(h(x), y)] = \mathbb{E}[|y - h(x)|^2]$$

The quadratic loss is the most common loss function in case $\mathcal{Y} = \mathbb{R}$. The below theorem is one of the reasons why.

Theorem 1.1.7. *Let $h : \mathcal{X} \to \mathcal{Y} = \mathbb{R}$ be a Borel function and assume that $\mathbb{E}[Y^2]$ and $\mathbb{E}[h(X)^2]$ are both finite. Then, the conditional expectation $r(x) := \mathbb{E}[Y|X = x]$ minimizes the L_2 risk.*

Proof. We start from the following expression and we shall minimize it:

$$\mathbb{E}[|Y - h(X)|^2] = \int_{-\infty}^{\infty} Y - h(X)^2 f_{y|x}(y|x)dy$$

Decomposing the square we obtain the following:

$$\mathbb{E}[|Y - h(X)|^2] = \int_{-\infty}^{\infty} y^2 f_{y|x}(y|x)dy - 2h(x)\int_{-\infty}^{\infty} y f_{y|x}(y|x)dy + [h(x)]^2 \int_{-\infty}^{\infty} f_{y|x}(y|x)dy$$

Which can be translated to:

$$\mathbb{E}[Y - h(X)^2] = \int_{-\infty}^{\infty} y^2 f_{y|x}(y|x)dy - 2h(x)\mathbb{E}[Y|X] + [h(X)]^2$$

Which implies the following rquivalence:

$$\underset{h(x)}{\operatorname{argmin}}\, \mathbb{E}[Y - h(X)^2] = \underset{h(x)}{\operatorname{argmin}}\, (-2h(X)\mathbb{E}[Y|X] + [h(X)]^2)$$

Finally, taking the derivative w.r.t $h(X)$ on the right side of the equation and setting it equal to zero we obtain:

$$h(X) = \mathbb{E}[Y|X]$$

□

Classification

Definition 1.1.8. *We say that a learning problem is a* ***classification problem*** *when $\mathcal{Y}$ is discrete. For instance, $\mathcal{Y}$ can be a book genre or the sentiment of a text.*

Definition 1.1.9. *A function $h : \mathcal{X} \to \mathcal{Y}$ where $(\mathcal{X}, \mathcal{Y}) \subset \mathbb{R}^d \times 1, 2, ..., K$ is called a classifier. In other words, a classifier is a map that, assigns an estimate $Y = r$ to a given observation $X = x$.*

As for the classification problems, the usual loss function is

$$L(y, y') = 1 - \delta_{y,y'}$$

Consequently, according to the definition of risk we have that:

$$R(h) = \mathbb{E}[L(y, y')] = \mathbb{E}[1 - \delta_{y,y'}] = \mathbb{E}[\mathbb{1}_{h(X)\neq Y}] = P(h(X) \neq Y)$$

Definition 1.1.10. *We define the* ***Bayes classifier*** *as:*

$$h_{Bayes}(x) = \underset{r\in 1,\dots,K}{\operatorname{argmax}} P(Y = r|X = x)$$

Theorem 1.1.11. *The Bayes classifier is optimal and the Bayes error rate is minimal.*

Proof. We will show that the minimum of the function $R(h) = \mathbb{E}[L(y, y')]$ is achieved at $h = h_{Bayes}$. For the definition of conditional probability we have the following.

$$R(h) = \mathbb{E}[L(y, y')] = \mathbb{E}_X \mathbb{E}_{Y|X}[L(y, y')|X]$$

Consequently,

$$\begin{aligned}
h_{min}(x) &= \underset{c\in 0,1}{\operatorname{argmin}} \mathbb{E}_{Y|X}[L(y, c)|X] \\
&= \underset{c\in 0,1}{\operatorname{argmin}} P(Y = 0|X = x)L(0, c) + P(y = 1|X = x)L(1, c) \\
&= \underset{c\in 0,1}{\operatorname{argmin}} P(Y \neq c|X = x) \\
&= \underset{c\in 0,1}{\operatorname{argmin}} 1 - P(Y = c|X = x) \\
&= \underset{c\in 0,1}{\operatorname{argmax}} P(Y = c|X = x)
\end{aligned}$$

□

1.1.2 PAC Learning

The *Probably Approximately Correct* learning model was introduced by L.G Valiant back in 1984. This framework consists of leaning from random examples from some probability distribution that may be over the input space. Moreover, in these kind of algorithms one can allow a small probability of failure, in cases in which might be impossible to learn the target function due to the structure of the input data. More formally, we have the following definition.

Definition 1.1.12. *Let $\mathcal{C}$ be a class of boolean functions $f : 0, 1^n \to 0, 1$. We say that $\mathcal{C}$ is PAC-learnable if there exists an algorithm $\mathcal{L}$ such that for every $f \in \mathcal{C}$, for any probability distribution $\mathcal{D}$, for any ϵ where $0 \leq \epsilon < \frac{1}{2}$ and for any δ where $0 \leq \delta < 1$ algorithm $\mathcal{L}$ on input ϵ and δ and a set of random examples picked from any probability distribution $\mathcal{D}$ outputs at least with probability $1 - \delta$, an hipothesis h such that $error(h, f) \leq \epsilon$.*

Definition 1.1.13. *We say that $\mathcal{C}$ is efficiently PAC-learnable if $\mathcal{C}$ is PAC learnable and:*

- *The number of examples that $\mathcal{L}$ takes is bounded by some polynomial in n, $\frac{1}{\epsilon}$ and $\frac{1}{\delta}$.*

- $\mathcal{L}$ *runs in time asymptotically bounded by some polynomial in* n, $\frac{1}{\epsilon}$ *and* $\frac{1}{\delta}$.

Example 1.1.14. *Consider a game to learn an unknown axis-aligned rectangle i.e. a rectangle in the Euclidean plane* $\mathbb{R}^2$ *whose sides are parallel to the coordinate axes. Let* R *be the target rectangle. The examples are provided to the learner are in the form of random points* p *along with a label indicating whether* p *is contained in* R *(a positive example) or not contained in* R *(a negative example). Figure 1.1 shows the unknown rectangular region* R *along with some positive and negative examples.*

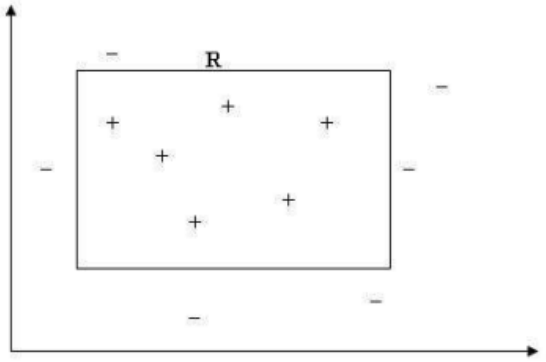


Figure 1.1: The target rectangle R along with some positive and negative points.

Theorem 1.1.15. *The concept class of axis aligned rectangles in* $\mathbb{R}^2$ *s is efficiently PAC learnable.*

Proof. Note that there is a simple and efficient way to come up with a hypothesis R' by taking a large number of examples and fitting the tightest rectangle around the positive ones so that all the given positive points lie inside it, as shown in figure 1.2.

□

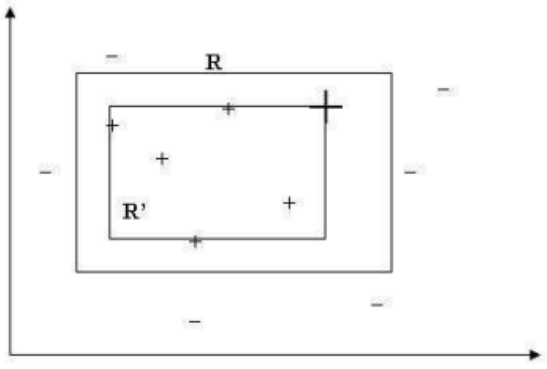


Figure 1.2: The tightest-fit rectangle R' over the given examples.

Moreover, R' will always be a subset of R. Also note that if we can guarantee that weight under $\mathcal{D}$ of each strip (i.e. the probability over $\mathcal{D}$ of falling under such strip) is not more than $\frac{\epsilon}{4}$, then we can be sure that the total error of R' is at most ϵ. This is because of the union bound:

$$P(A \cup B) \leq P(A) + P(B)$$

So, consider what can be a bad event for us in this setting. Please agree it would be a bad event if the probabilistic weight (or the probability), over $\mathcal{D}$, of the top strip (that is a part of the error region) is more than $\frac{\epsilon}{4}$ i.e. at least $\frac{\epsilon}{4}$ (see figure 1.3). In other words, it would be a bad event if the probabilistic weight of the rest of the region i.e. non-error region is at most $(1 - \frac{\epsilon}{4})$.

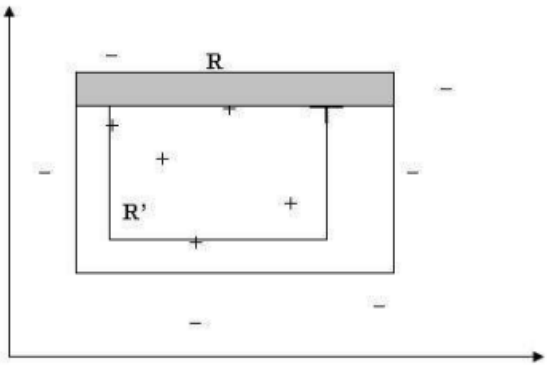


Figure 1.3: One of the four error strips.

Therefore, a bad event over m draws would be when the probability that we draw all our m examples from the non-error region is at most $(1 - \frac{\epsilon}{4})^m$. Here we made use of the fact that all the draws/examples are independent of each other. The same argument holds for the other three strips as well. So, to calculate the total probabilistic weight of the bad event, we take the union bound of all four to give $4(1 - \frac{\epsilon}{4})^m$. And this probability must be bound δ. This is because, as discussed in the formal definition of PAC learning, our confidence measure in the good event must be at least $1 - \delta$ i.e. the probability of the bad event must be at most δ. So we get:

$$4(1 - \frac{\epsilon}{4})^m \leq \delta$$

that can be solved using that $(1 - x) \leq e^{-x}$:

$$m \geq (\frac{4}{\epsilon} ln(\frac{4}{\delta})$$

So, if our algorithm takes at least m examples, then with a probability at least $1 - \delta$, the resultant hypothesis h will have an error at most ϵ with respect to f and over $\mathcal{D}$. Also, since the processing of each example would require at most four comparisons, the running time is linearly bounded by m. Consequently, it is bounded by a polynomial in n, $\frac{1}{\epsilon}$, and $\frac{1}{\delta}$ as required. Hence, PAC learnable.

1.2 Deep Neural Networks

1.2.1 Theorems of Cybenko and Hornik

In, 1989, Cybenko proved the following theorem:

Theorem 1.2.1 (Cybenko). *Let σ be a continuous monotone function with $lim_{t\to-\infty}\sigma(t) = 0$ and $lim_{t\to\infty}\sigma(t) = 1$. For intance, σ could be the sigmoid function $\sigma(t) = \frac{1}{1+e^{-t}}$. Then, the set of functions of the form:*

$$f(x) = \sum \alpha_j \sigma(w_j^T x + b_j)$$

is dense in $C_n([0,1]^n)$, where $C_n([0,1]^n) = C([0,1]^n)$ denotes the space of continuous functions from $[0,1]^n$ to $[0,1]$ with the metric:

$$d(f,g) = sup|f(x) - g(x)|$$

Later in 1991, Hornik proved a generalization of Cybenko's result, also known as *Universal Approximation Throrem*:

Theorem 1.2.2 (Hornik). *Consider the set of functions defined in Cybenko's theorem, but without conditions placed on σ. Let $L^p(\mu)$ be the space of functions f with $\int |f|^p d\mu < \infty$ with the metric $d(f,g) = (\int |f-g|^p d\mu)^{\frac{1}{p}}$ Then:*

- *If σ is bounded and non-constant, then the set is dense in $L^p(\mu)$, where μ is anby finite measure in $\mathbb{R}^{\urcorner}$.*
- *If σ is additionally continuous, then the set is dense in $C(X)$, the space of all continuous functions on X, where $X \subset \mathbb{R}^k$ is compact.*
- *If, additionally, $\sigma \in C^m(\mathbb{R}^k)$, then the set is dense in $C^m(\mathbb{R}^k)$ and also in $C^{m,p}(\mu)$ for every finite μ with compact support.*
- *If, additionally, σ has bounded derivatives up to the order m, then the set is dense in $C^{m,p}(\mu)$ for every measure μ on $\mathbb{R}^k$.*

What the above theorem says is that given any continuous function at all no matter how complicated it might be, it's always possible to find an artificial neural network which can approximate that function as well as you would like. In other words, for most activation functions, including any $\sigma : \mathbb{R} \to \mathbb{R}$ such that $lim_{t\to-\infty}\sigma(t) = 0$ and $lim_{t\to\infty}\sigma(t) = 1$, and also including the ReLu function, we can represent almost any function given enough neurons, with small error in any reasonable norm.

1.2.2 Stability of Stochastic Gradient Descent

The most widely used optimization method in machine learning practice is stochastic gradient method (SGM). Stochastic gradient methods aim to minimize the empirical risk of a model by repeatedly computing the gradient of a loss function on a single training example, or a batch of few examples, and updating the model parameters accordingly.

Let's consider the following general setting of supervised learning. Let D be an unknown distribution o and $S = (z_1, ..., z_n)$ a sample of non i.i.d. examples from D. Let $f(w, z)$

denote s loss function where w is the model and z the sample. Our goal is then to minimize the empirical risk:

$$R_S(w) = \frac{1}{n}\sum_{i=1}^{n} f(w, z_i)$$

Definition 1.2.3. *The **generalization error** of a model w is the difference*

$$R_S(w) - R(w)$$

where $R(w)$ denotes the risk defined previously.

If $w = A(S)$, where A is a random algorithm, we need to estimate the expected generalization error:

$$\epsilon_{gen} := \mathbb{E}_{S,A}[R_S(A(S)) - R(A(S)]$$

Definition 1.2.4. *The algorithm A is $\epsilon - uniform$ stable if for all data sets S, S' that differ at most one example:*

$$\sup_z \mathbb{E}_A[f(A(S), z) - f(A(S'), z)] \leq \epsilon$$

We define $\epsilon_{Stab}(A, n)$ the smallest such ϵ.

Proposition 1.2.5. *The generalization error can be controlled by $\epsilon_{Stab}(A, n)$:*

$$|\mathbb{E}_{S,A}[R_S(A(S)) - R(A(S))]| \leq \epsilon_{Stab}(A, n)$$

Proof. Let $S = (z_1, ..., z_n)$ and $S' = (z'_1, ..., z'_n)$ two independent random samples and let $S^(i) = (z_1, ..., z_{i-1}, z'_i, z_{i+1}, ..., z_n)$.Then , we have the following:

$$\begin{aligned}
\mathbb{E}_S\mathbb{E}_A[R_S(A(S))] &= \mathbb{E}_S\mathbb{E}_A[\frac{1}{n}\sum_{i=1}^{n} f(A(S)), z_i)] \\
&= \mathbb{E}_S\mathbb{S}'\mathbb{E}_A[\frac{1}{n}\sum_{i=1}^{n} f(A(S^{(i)})), z'_i)] \\
&= \mathbb{E}_S\mathbb{S}'\mathbb{E}_A[\frac{1}{n}\sum_{i=1}^{n} f(A(S)), z'_i)] + \delta \\
&= \mathbb{E}_S\mathbb{E}_A[R(A(S))] + \delta
\end{aligned}$$

where

$$|\delta| = |\mathbb{E}_S\mathbb{E}_{S'}\mathbb{A}[\frac{1}{n}\sum_{i=1}^{n}](f(A(S^{(i)}), z'_i) - f(A(S), z'_i))| \leq \epsilon_{Stab}(A, n)$$

□

Definition 1.2.6. *Given the loss function $f(w, z)$, the stochastic gradient update with learning rate $\alpha_t > 0$ is given by:*

$$w_{t+1} = w_t - \alpha_t \nabla_w f(w_t, z_{i_t})$$

where the indices i_t are picked randomly in $1, ..., n$ or according to a random permutation.

Theorem 1.2.7. *Assume that for all $z, f(,z)$ is convex and L-Lipschitz, and ∇f is $\beta -$ Lipschitz. Suppose that we run the stochastic gradient method for T steps with step sizes $\alpha_t \leq \frac{2}{\beta}$, then*

$$\epsilon_{Stab} \leq \frac{2L^2}{n} \sum_{t=1}^{T} \alpha_t$$

Proof. For the proof we will need the following result:

Proposition 1.2.8. *If f is convex and ∇f is $\beta -$ Lipschitz then:*

$$\langle \nabla f(v) - \nabla f(w), v - w \rangle \geq \frac{1}{\beta} ||\nabla f(v) - \nabla f(w)||_2^2$$

Let S and S' be two samples of size n differing in only one example. Let $\{w_t\}$ and $\{w'_t\}$ two runs of the algorithm with data sets S and S', and $\delta_t = ||w_t - w'_t||_2$.

Because the two data sets differ in a single example, at step t:

- With probability $1 - \frac{1}{n}$, the algorithms choose the same data example.
- With probability $\frac{1}{n}$, the algorithms choose different data examples.

If the first case happens at strep t, then:

$$\begin{aligned}
\delta_{t+1}^2 &= ||w_t - w'_t - \alpha_t(\nabla f(w_t, z) - \nabla f(w'_t, z))||_2^2 \\
&\leq ||w_t - w'_t||^2 + \alpha_t^2 ||\nabla f(w_t, z) - \nabla f(w'_t, z)||_2^2 - 2\alpha_t \langle w_t - w'_t, \nabla f(w_t, z) - \nabla f(w'_t, z) \rangle \\
&\leq ||w_t - w'_t||^2 = \delta_t^2
\end{aligned}$$

provided that $\alpha_t \leq \frac{2}{\beta}$, using 1.2.8.

On the other hand, if the algorithms choose different data examples z and z', we have:

$$\begin{aligned}
\delta_{t+1} &= ||w_t - w'_t - \alpha_t(\nabla f(w_t, z) - \nabla f(w'_t, z))||_2 \\
&\leq ||w_t - w'_t||_2 + \alpha_t(||\nabla f(w_t, z)||_2 + ||\nabla f(w'_t, z)||_2 \\
&\leq \delta_t 2\alpha_t L
\end{aligned}$$

Taking into account that the first case happens with probability $1 - \frac{1}{n}$ and the second one with probability $\frac{1}{n}$:

$$\begin{aligned}
\mathbb{E}[\delta_{t+1}] &\leq (1 - \frac{1}{n})\mathbb{E}[\delta_t] + \frac{1}{n}\mathbb{E}[\delta_t + 2\alpha_t L] \\
&= \mathbb{E}[\delta_t] + \frac{2\alpha_t L}{n}
\end{aligned}$$

Overall we have:

$$\mathbb{E}[\delta_T] \leq \frac{2L}{n} \sum_{i=1}^{T} \alpha_t$$

and

$$|f(w_T, z - f(w'_T, z))| \leq L||w_T - w'_T||_2 \leq \frac{2L^2}{n} \sum_{i=1}^{T} \alpha_t$$

□

Chapter 2

A Real-World Application: Design and Implementation

2.1 Problem Statement

2.1.1 Business Background

In 2021, the company launched an experiment in which they published +2K videos on Children's Books detail page on the US marketplace. The videos featured the book cover, three inside pages and the book back cover, as well as some slides with text, such as the title and the author(s). The videos were generated using an architecture called *WaldoMediaFactory*, that I will expand on afterwards, and Vivid, an internal video rendering platform. On the real world it is important that any generated content is properly validated before released to the general public. In other words, these videos needed to be validated before deployed to production. This process not only implies checking that the content on the videos is the expected, but also that the videos are accessible, for instance. Nevertheless, the company did not have any internal automated or semi-automated tool to validate that the videos were correctly generated, and therefore the videos' quality-assurance process process was fully manual. The validation was done by an internal team that spent around two weeks checking a batch of 1000 videos and making sure that they were valid and if not, discard them.

This year the plan is to launch more video experiments not only with Children's Books, but also with other book's genres, such as Fiction and Non-fiction, Manga, Comic Books or Cookbooks. Moreover, if these experiments work as expected, more videos will be generated, as well as in more marketplaces. As a result, there is a clear need of an automated or semi-automated tool to validate the videos. First, because the company cannot always rely on manual processes. Second, because we need to reduce the error rate on these kind of validation. And most important, if this process is automated, the human time spent on video QA will substantially decrease, and therefore, the whole process will be cheaper. As a result, the fact of having an automated video processing pipeline will enable us to scale.

2.1.2 Technical Requirements

More formally, the objective of this project is to build an automated video-processing pipeline with the main functionality of video validation. In other words, the workflow will process a batch of videos and perform some checks to ensure that the videos are eligible to be published on the website and contain no errors. To achieve that, three types of validation steps have been implemented:

- **The Text Content Validator**: This validator checks that the text on the video is not cut, partially occluded. It also checks if the text present on the video is the expected. It basically compares the expected text on the video against the text that is actually present on the frames.
- **The Spelling Validator**: This validator checks the spelling of the text on the video, meaning that fails if the text on the video contains spelling errors.
- **The Video Flash Detector**: And finally, the flash detector processes a video and detects the timestamps in seconds in which a video contains flashes that could likely cause seizures to the eyes. This validation step is crucial to ensure that the videos are accessible.

The above checks run in parallel for each video, and each video is pre-processed before the execution of these validators. This pre-processing step will be detailed later. A high-level diagram of the idea can be seen on figure 2.1:

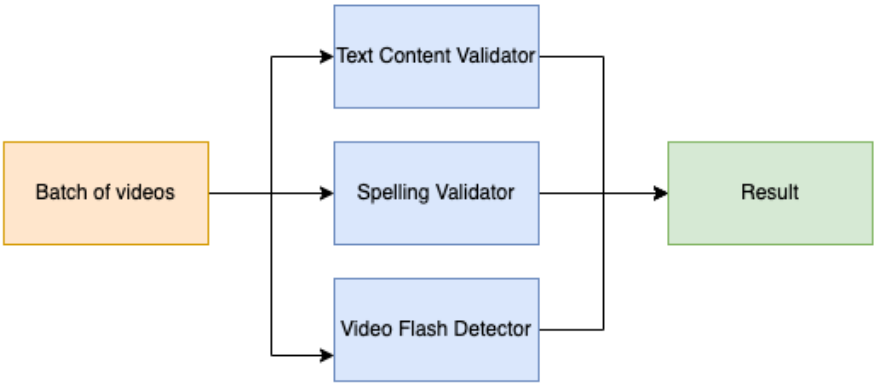


Figure 2.1: Three-step workflow - High-Level idea

Another important requirement for this architecture is that more validation steps can be easily added in the future, such as for example a color contrast check or a metadata validation step. To fulfill the requirement, the following idea shown in figure 2.2 has been implemented, based on a *plug and play* policy.

Another point to keep in mind is that this workflow must be **template-agnostic**, meaning that it can validate a diverse range of videos. As for now, the existing videos are of two

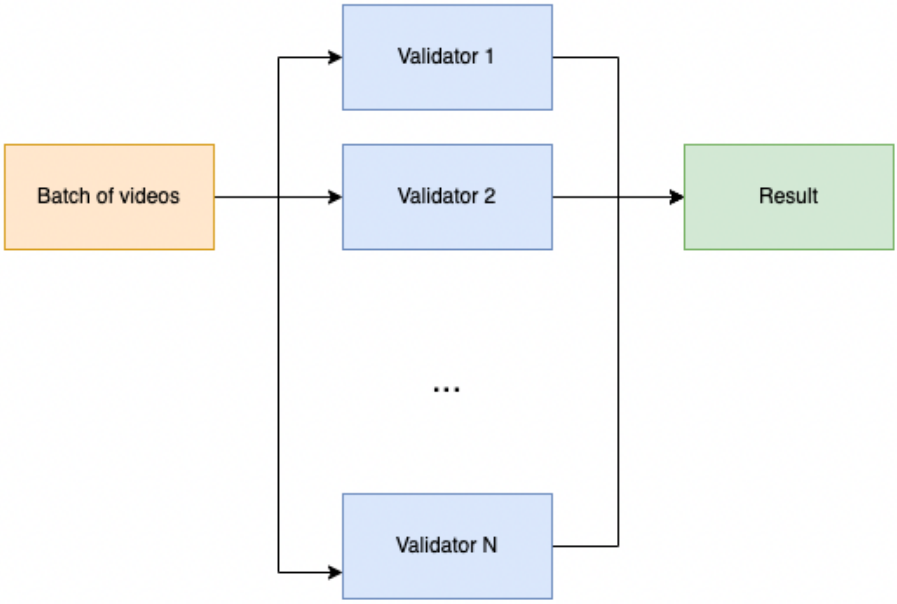


Figure 2.2: *Plug and play* workflow

types: Children's Books videos and Fiction/Non-Fiction videos. Figure 2.3 illustrates how the currently available templates look like.

(a) Children's Books Template **(b)** Fiction/Non-Fiction Template

Figure 2.3: Thumbnails of the two available templates

The architecture must of course be **scalable**, supporting big batches of videos, as well as the integration of more validation steps as said.

In short, these are the requirements.

- Implementation and integration of the Text Content Validator
- Implementation and integration of the Spelling Validator
- Implementation and integration of the Video Flash Detector
- The architecture must support multiple templates, and it must be easy to adapt it to support a new one.
- The architecture must be flexible to be able to easily add new validation steps,

following the stated *Plug and Play* policy.

- The architecture must be scalable, being able to validate huge batches of videos.
- The architecture must be efficient both in run-time and in resource usage.
- Integration of the workflow to *Waldo Media Factory*, to be able to use it in production.

2.1.3 Challenges

The main challenge to build the desired solution is the fulfillment of the *template-agnostic* requirement. In other words, the workflow must process and validate different kinds of videos, featuring different book genres, not having the same length or containing assets from different sources. Moreover, this requirement also implies that the implementation of the validation steps must not depend on the type of video, only focus on the main technical objective.

Moreover, the *Plug and Play* requirement is challenging as well, because the solution needs to be as flexible as possible on that aspect and it is not straightforward.

All in all, the designing phase is crucial and that is something that I never did before for such a big architecture, being this one more challenge to overcome.

2.2 Base Architecture: Waldo Media Factory

2.2.1 Amazon Web Services

Amazon Web Services (AWS) is a secure cloud services platform, offering compute power, database storage, content delivery and other functionality to help businesses scale and grow. In order to build a solution for the exposed problem I used some of the services offered by this platform. Moreover, the existing architecture is built on top of AWS as well. The purpose of this section is to give an overview of the services that will be mentioned from now on so that the solution can be better understood.

AWS IAM

AWS Identity Access Management is a service that enables you to control who can access which services and resources. At a high level, is a layer of security to ensure that unless we say the opposite, no service will be accessible by anyone, and this is by default. In other words, if someone is using AWS they must be using IAM. It provides the following functionalities:

- Create User identities — Add Users (unique identities that can be used to interact with AWS services) to your AWS Account. A User can be an individual, system, or application requiring access to AWS services.
- Assign and manage security credentials — Assign security credentials (such as access keys) to each User, and rotate and/or revoke these credentials as desired.

- Organize Users in groups — Create groups to more easily manage permissions for multiple Users.
- Centralize control of User access — Control which operations each User can perform, such as accessing specific AWS service APIs and resources. The operations that a User can perform are determined by the policies that the role has attached.
- Conditionally control User access — Add conditions to control how a User can use AWS, such as their originating IP address, time of day, or whether they are using SSL.
- View a single AWS bill — Receive a single bill for the activity of all Users within your AWS Account.

So for instance, let's say we set up a service A, which is an API, and we want this service A to be called by another service B, we then need to create a policy that allows to invoke the API of service A and attach it to the role of service B, so that the instance B can perform the operation of calling service A. This example is illustrated below in figure 2.4.

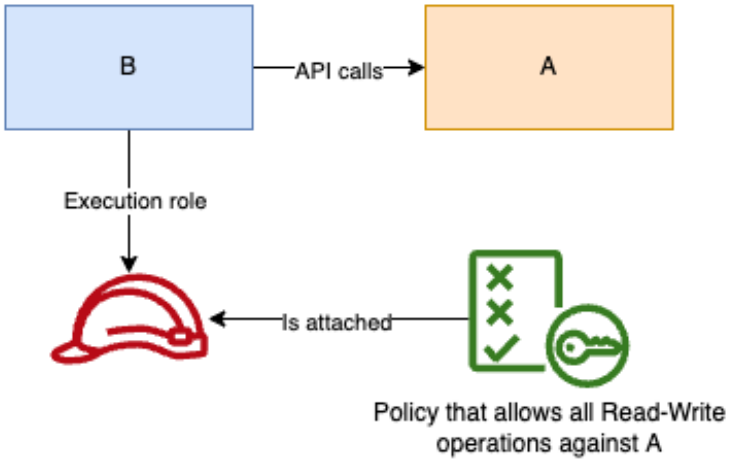


Figure 2.4: Usage example of IAM

And of course, to implement the solution I needed to create both users and roles that I will expand on afterwards.

AWS S3

S3, the Simple Storage Service, is a reliable, fast and cheap way to store data on the Internet. S3 can be used to store just about anything: XML documents, binary data, images, videos, or whatever else. The preferred interface is REST. There exist variants such as S3 Intelligent-Tiering, S3 Standard-IA, S3 One Zone-IA or S3 Glacier, and depending on the desired use case one may want to use one variant or another. For instance, S3 Glacier is suitable if the data is not at all accessed, meaning 1-2 times a year. On the other hand, S3 Standard-IA is the best option if one wants rapid access to data when needed, as it offers low latency on retrieval operations. On this use case, S3 Standard is the chosen

variant (the default one), as no specific requirements are needed with regards to storage time or retrieval operations.

Figure 2.1 depicts how S3 works at a high level. An IAM role can perform these types of operations, given of course that has the corresponding policies attached.

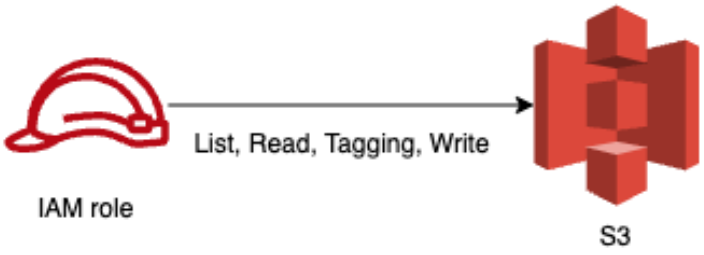


Figure 2.5: Available sets of S3 operations

AWS Lambda

AWS Lambda is a compute service that runs your code in response to events and automatically manages the compute resources for you, making it easy to build applications that respond quickly to new information. AWS Lambda starts running your code within milliseconds of an event such as an image upload, in-app activity, website click, or output from a connected device. You can also use AWS Lambda to create new back-end services where compute resources are automatically triggered based on custom requests. With AWS Lambda you pay only for the requests served and the compute time required to run your code. Billing is metered in increments of 100 milliseconds, making it cost-effective and easy to scale automatically from a few requests per day to thousands per second.

Lambda is supported in multiple run-times, such as Java and Python, but it works exactly the same. The code must have an entry-point which is a function called *handler* that receives two parameters: the *event* and the *context*. The *event* contains the input of the function and the *context* is an object containing information about the invocation, function, and execution environment.

While Lambda seems a pretty good choice to build an efficient micro-services architecture, it has its constraints. For instance, there's an execution timeout of 15 minutes and the deployment package .zip file size must be below 50 MB (and below 250 MB unzipped), which sometimes is not straightforward to achieve. As far as I have seen, this is mostly because of the dependencies. For example, *scipy*'s size unzipped is already 200 MB. As a result, if your code depends on *scipy* and its size exceeds 50 MB it can't be deployed to AWS Lambda.

As it will be seen, the Waldo Media Factory architecture is mostly based on AWS Lambda, as it's the most suitable service for the use case as it will be seen afterwards.

AWS Step Functions

Step Functions help developers build, run, and scale distributed applications using state machines and the Amazon States Language (ASL). Step Functions can be thought as a fully-managed application state tracker and coordinator in the AWS Cloud. Step Functions has been used to build completely serverless applications (via integrations with AWS Lambda and API Gateway), applications hosted in PROD or EC2 (via Coral or native programs), hybrid-cloud applications, mobile applications, and even applications built from a series of UNIX or Powershell scripts.

If you are building a distributed application that performs a series of sequential and parallel steps, want to track the state of processing, and need to recover or retry if a task fails, AWS Step Functions can help you. And this is exactly what I used to build the QA workflow.

As Step Functions is a state machine, there are different types of states. The most relevant steps are the following:

- **Pass**: This type of state just passes the input to its output without performing any operation. This state is often used for debugging operations.
- **Task**: This type of state is where the work gets done. It is usually implemented as a Lambda, like I did on this project. Step functions invokes the corresponding Lambda handler and outputs the result of the Lambda.
- **Choice**: This state enables you to add *if/else* logic on the workflow. The decision can only be made using comparison of strings. It can be useful to optimize the number of workflows by running similar workflows as branches of a general workflow instead of having to build separate state machines.
- **Wait**: This state delays the state machine for a specified time. Can be useful to *cool down* between API requests, for instance.
- **Parallel**: This interesting step can be used to create parallel branches of execution in the state machine. As I will show below, I use it to run the validation steps in parallel as they are totally independent.
- **Map**: Given an input array, this step is run one time for each element of the array. It's very useful for batch operations in which each of the elements needs to be processed separately. This state is also used in the QA workflow as I will show, because it runs for batches of videos.

On figure 2.6 there's a *Hello World* example of Step Functions. The input of the workflow is of the following form:

```
{
    "IsHelloWorldExample": bool
}
```

Then, the *Hello World example?* state is a *Choice* and evaluates the variable "IsHelloWorldExample" to determine whether to run the left part of the workflow or the right

one. On figure 2.6 the input value was *true* and therefore the right side of the workflow was executed. On this part of the workflow there is also a *Parallel* state, in which both branches of the workflow are run, with the same input, and the output of each of the branches is appended on a list, that is the output of the *Parallel* state.

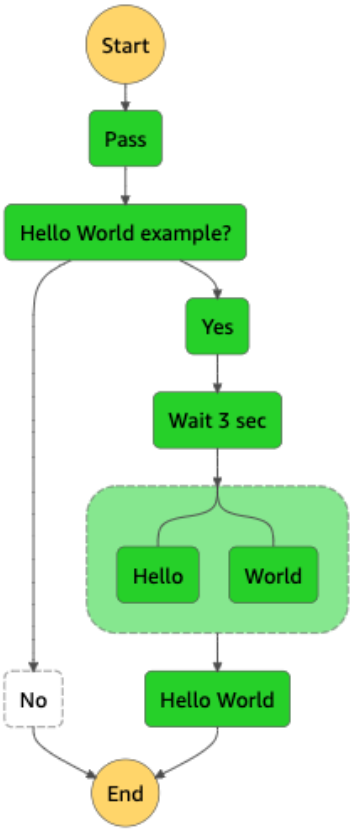


Figure 2.6: *Hello World* of AWS Step Functions

AWS API Gateway

Amazon API Gateway is a fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale. The main functionality is to create an API that acts as a "front door" for applications to access data, business logic, or functionality from any back-end services, such as workloads running on Amazon Elastic Compute Cloud (Amazon EC2), code running on AWS Lambda, or any Web application. Amazon API Gateway handles all the tasks involved in accepting and processing up to hundreds of thousands of concurrent API calls, including traffic management, authorization and access control, monitoring, and API version management.

Figure 2.7 depicts an API Gateway call flow. As said before, API Gateway handles traffic coming from multiple sources, such as Mobile Apps, Websites or Services. Amazon CloudFront is a web service for content delivery, it automatically routes incoming requests to the nearest edge location, so content is delivered with the best possible performance. Then, the requests reach API Gateway, that forwards these to the corresponding target endpoints. AWS CloudWatch is a monitoring service, one can access logs and data from

running services within an AWS account. As a result, the logs from API Gateway can be found on CloudWatch.

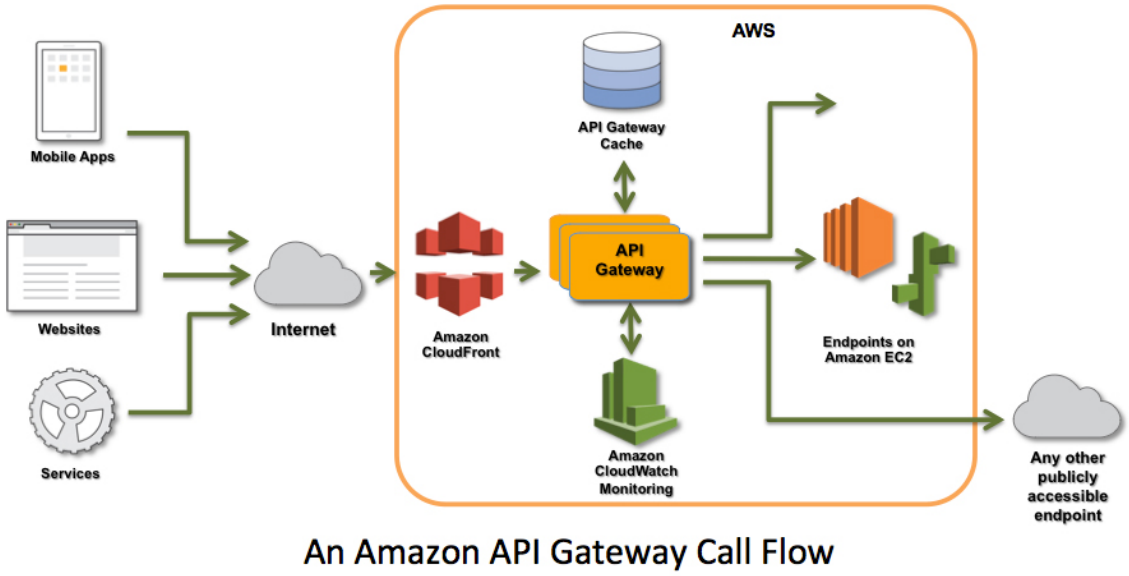


Figure 2.7: An Amazon API Gateway Call Flow

AWS Rekognition

AWS Rekognition is an AI service that provides sophisticated deep learning-based visual search and image classification algorithms. With Rekognition, one can detect objects, scenes, faces; recognize celebrities, identify inappropriate content in images and much more. Amazon Rekognition is based on deep learning technology developed by Amazon's computer vision scientists to analyze billions of images daily for Prime Photos. Amazon Rekognition uses deep neural network models to detect and label thousands of objects and scenes in images. With regards to the use case for this project, it is used to detect and recognize text on an image, as I will explain afterwards.

AWS Comprehend

AWS Comprehend is a service that provides access to numerous types of algorithms related to NLP (Natural Language Processing) and Text Analysis. Some examples include sentiment analysis, key-phrase extraction or syntax detection. In this project I use Comprehend to detect the language of a text and to perform syntax analysis to identify the proper nouns, as I will expand on in the spelling validator section.

2.2.2 Waldo Media Factory high-level architecture

Before diving deep into the design of the actual solution I will give an overview of the *context* on top of which this has to be built. *Waldo Media Factory* is a system focused on automating the creation and upload of video trailers for books with graphical content

(e.g. children's books, comics, graphical novels, etc) to VSE (Video Shopping Experience) systems. It's built on top of Amazon Web Services (AWS) and has the following features:

- Creation and rendering of videos
- Uploading and publishing videos
- Updating published videos
- Deleting/unpublishing videos

Below in figure 2.8 there is a simplified diagram depicting the Waldo Media Factory high-level architecture:

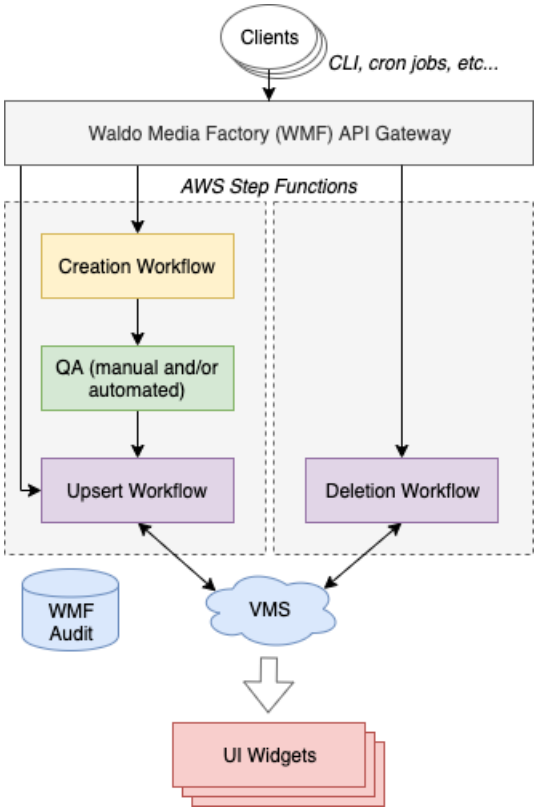


Figure 2.8: Waldo Media Factory high-level architecture

As it can be seen, there is a QA step depicted, the one that will be tackled on this project. The idea is that this QA step is triggered from the creation workflow, so that the recently created videos can be validated before their publication. As for now, the whole architecture is triggered from a CLI (Command Line Interface) and the CLI triggers the corresponding workflows via an API Gateway. The CLI itself is part of Waldo Media Factory as the solution is under development.

2.2.3 The Video Creation Workflow

The Video Creation workflow as the name implies, is responsible for the creation of the videos. Given a batch of book ASINs (Amazon Standard Identification Number) it outputs the generated videos under an S3 location. The input of the workflow is of the following

form and is illustrated in figure 2.11:

```
{
  "obfuscatedMarketplaceId": string,
  "asins": list of strings,
  "videoPrefix": string,
  "template": string,
  "locale": string
}
```

This input is directly mapped to the first component of the workflow, the *VividExternalIdPass* Lambda. This is part of a pre-processing step to ensure that the final request to the service that actually creates the video (Vivid) is correct. What I will expand on is the first map of the workflow, which from now on I will call the *FetchBookContentMap*. The last part of the workflow mainly builds the request to be sent to the renderer service.

This first component of the workflow is mainly responsible for fetching all the information from a book, such as for example the book cover images, the inside pages, the authors or the number of pages. Moreover, it also returns a color palette. This color palette is a set of n colors that are suitable to the cover of the corresponding book. This is useful because in some templates we need to customize the colors on the video and this is done with the *PaletteGenerator* lambda. All in all, the two main components of the *FetchBookContentMap* are the *FetchBookContent* Lambda and the *PaletteGenerator* Lambda. Next, I will expand on the *PaletteGenerator* Lambda because it's the part of the workflow that I implemented and is technically interesting.

The Palette Generator Lambda

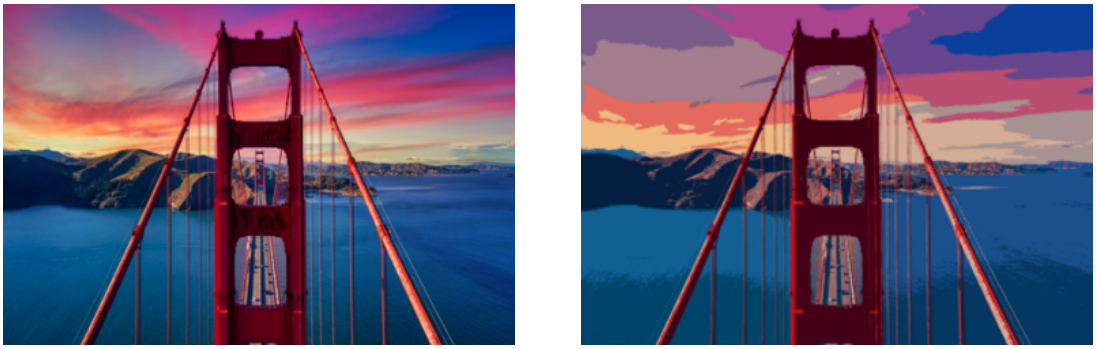
In order to automate the video creation, we also need to automate the theme generation for the trailers. For that, the first step is to be able to compute a color palette that will be applied to a given template, that will be the background of the video trailer. The process that it's going to be explained is illustrated in figure 2.9:

Figure 2.9: Color palette extraction schema

So the high level steps are:

- Given a book cover, generate a suitable n-color palette.
- Apply this color palette to a given template.

To achieve that, we need to take into account multiple factors, such as the contrast, ensure that the background fits with the book cover, how to locate the colors, and how to mitigate all the possible problems and corner cases. To be able to generate a palette that fits with the book cover theme, we must apply color quantization to an image. Color quantization is a process that reduces the number of distinct colors used in an image, usually with the intention that the new image should be as visually similar as possible to the original image. There is an example in figure 2.10.



(a) The Golden Gate **(b)** The Golden Gate quantized to 20 colors

Figure 2.10: Comparison between an original image and the quantized image

Among the many applications that this process has, one of them is to create a color palette from a given image, as shown in figure 2.9. As it can be seen, if we apply a quantization algorithm to the book cover and extract the colors we obtain an optimal representation of the main colors of the cover. And of course, by increasing n the colors are more representative. There are different algorithms to perform color quantization, but I saw that the results from the algorithm *K-Means* were slightly better and I decided to integrate it. *K-Means* is a *clustering* algorithm that aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean. So what I did is run this algorithm in which the *observations* where colors in RGB, as if they were 3D points. After that, the output will be a palette of k colors, as a result of a clustering of the colors already present in the image.

Some modifications can be done to *tweek* this algorithm and see different results. One that I tried was Spatial color quantization. This technique relies on taking into account the position of the pixels apart from the color pixels to perform the color quantization method, that's the reason of the name spatial color quantization. Therefore, the Euclidean dimension of the space is five: x, y, r, g and b. While this method takes into account the 2D position of a pixel on an image, the contribution of the spatial coordinates can be weighed with a value t between 0 and 1. In our case, we could test for different values of t to see the different results. In fact, the *K-Means* algorithm is a particular case of this method, in which $t = 0$. Nevertheless, with some experiments was seen that for our application

the best approach is to use the raw *K-Means* algorithm, without the space coordinates, as the resulting palettes showed more contrast because the algorithm only relied in the color coordinates.

The input of the *palette_generator* Lambda is of the following form:

```
{
  "palette_data": {
    "cover_image_url": <url of the book cover>,
    "template": <template id>
  }
}
```

And depending on the template the number of output colors is determined in the code. For instance, for the Children's Books template the number of colors is 7.

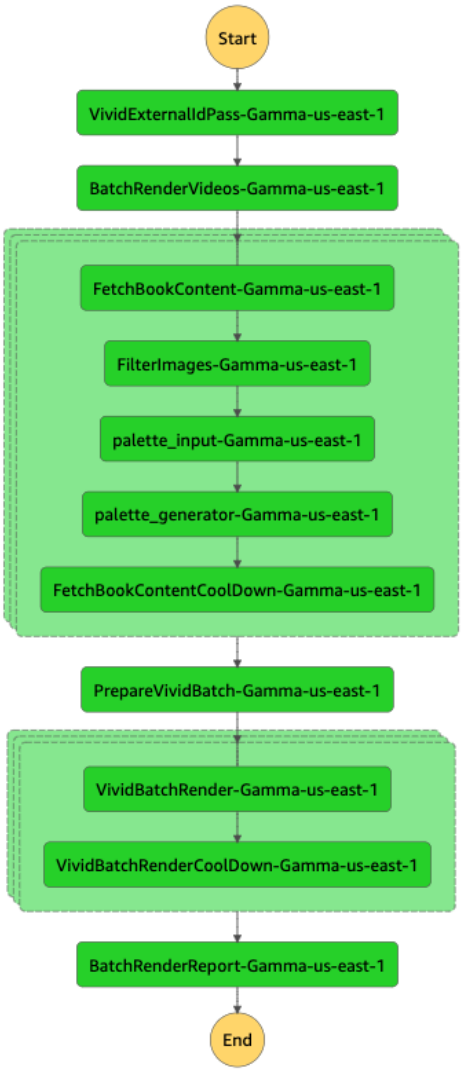


Figure 2.11: Video Creation Step Functions Workflow

2.2.4 CI/CD: The Waldo Media Factory Pipeline

In software engineering is crucial to have well defined the Continuous Integration and Continuous Delivery of the processes. For that, a pipeline is often used. The main objective is to compile the incremental code changes made by developers, and then link and package them into production. All this process is automated with the help of the Waldo Media Factory pipeline, that can be seen in figure 2.12 and I will describe at a very high level.

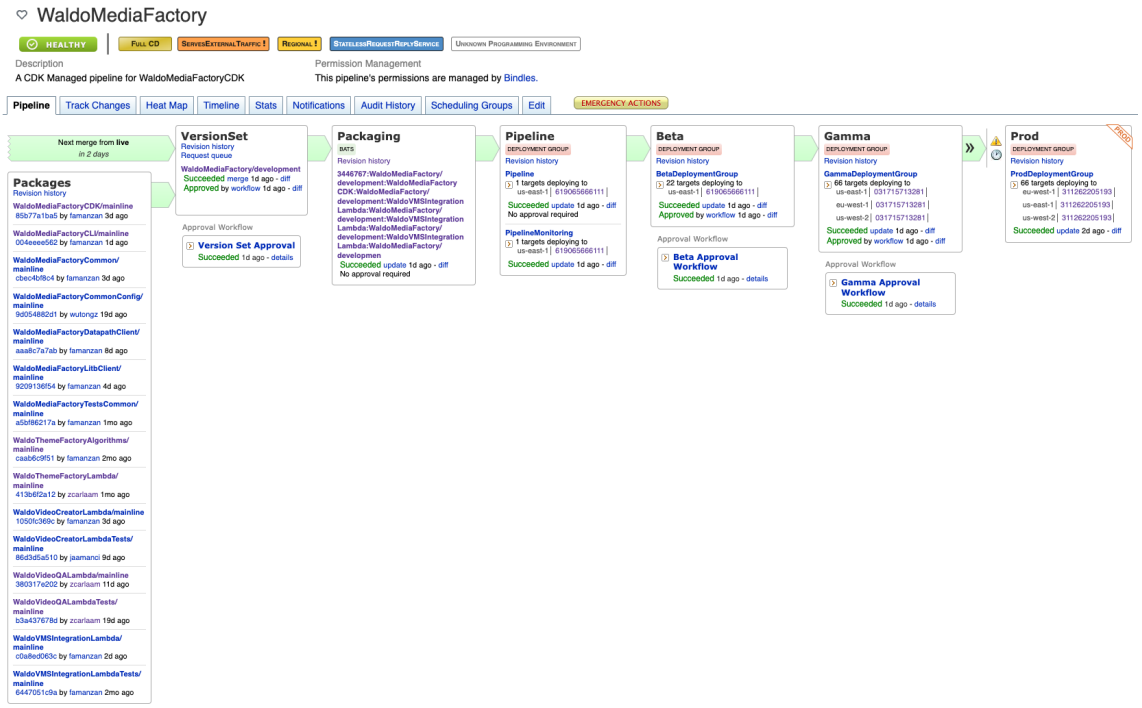


Figure 2.12: Waldo Media Factory pipeline

Waldo Media Factory is composed by a number of packages, that can be seen on the left side of the pipeline. The main idea is that whenever there is a change under a package consumed by this pipeline, this is automatically triggered and all the deployments take place. There are three environments: Beta, Gamma and Prod; and after each of the deployments the integration tests for the corresponding environment are triggered and, if at least one of the tests fail, the promotion to the next environment is automatically stopped and the pipeline needs to get unblocked in order to continue with the deployments. This means that the pipeline is continuously *monitored* and if there is a failure we're immediately notified. The CI/CD process that I followed during the implementation of this project is illustrated in figure 2.13.

2.3 The Video QA Workflow

2.3.1 High-Level System Design

A high-level architecture solution is proposed to address the stated problem. To start with, I will talk about the main workflow, that can be seen in figure 2.14.

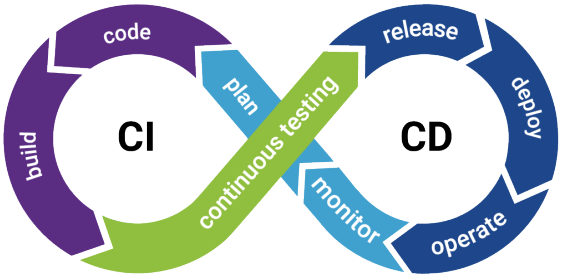


Figure 2.13: Waldo Media Factory pipeline

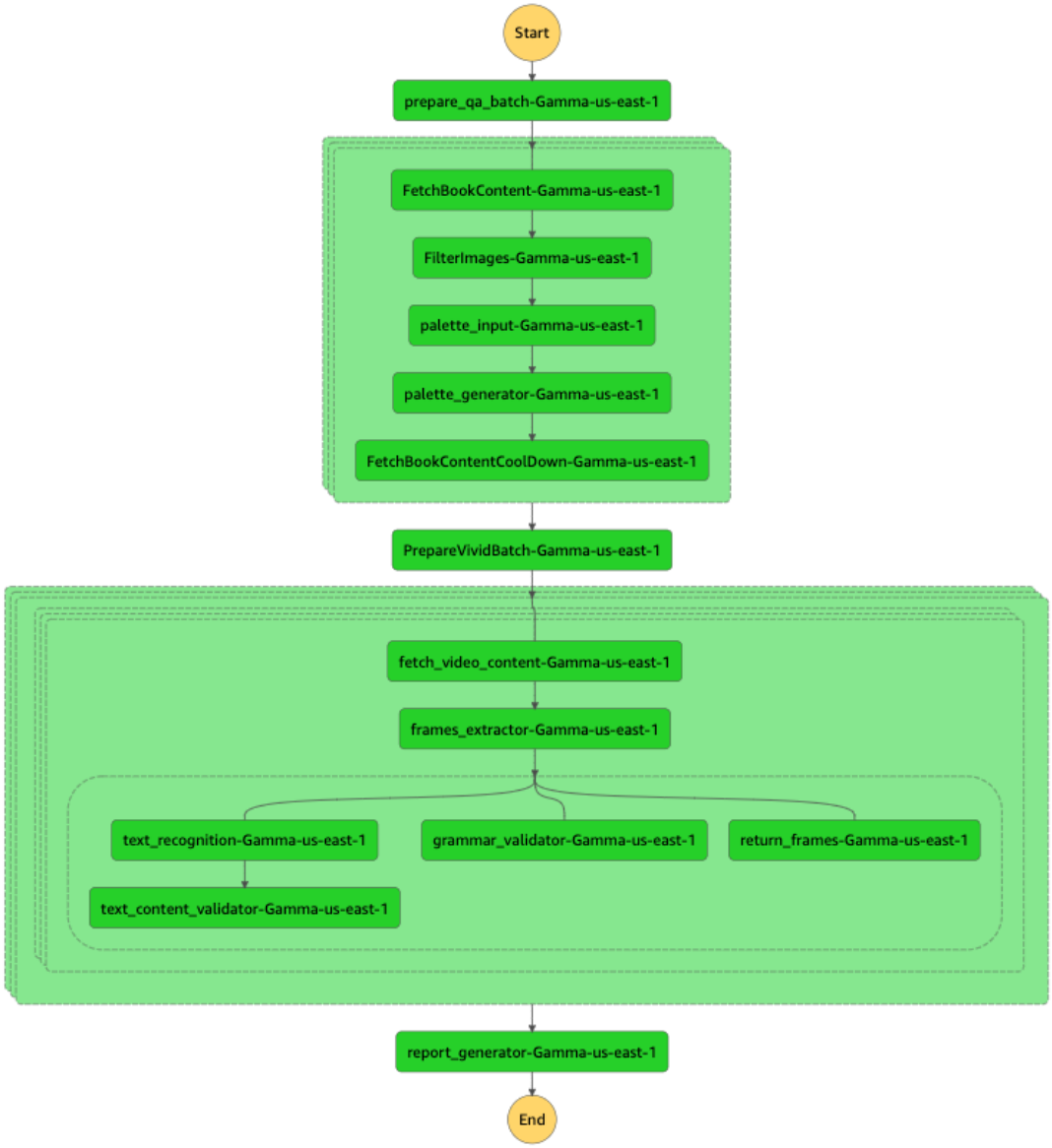


Figure 2.14: Video QA Step Functions Workflow

The video QA workflow is be implemented using AWS Step Functions and is composed by Lambda functions. The implementation of the Lambdas explained below can be found under the package *WaldoVideoQALambda*, except for the Lambdas located on the first map of the workflow and the *PrepareVividBatch* Lambda. The workflow starts with the

PrepareQABatch Lambda function, the workflow entry-point. This lambda function fetches the ASINs of the books and places them on a list. The input/output of the *PrepareQABatch* Lambda are as follows:

Input:

```
{
"bucket_name": <bucket_name> // Comes from the StepFunctions Payload
"obfuscatedMarketplaceId": <obfuscatedMarketplaceId>,
"template": <template>,
"locale": <locale>,
"videoPrefix": <videoPrefix>,
"index": <objectIndex>
}
```

Output:

```
{
  "requests": [{
      "obfuscatedMarketplaceId": <obfuscatedMarketplaceId>,
      "asin": <asin>,
      "template": <template>,
      "locale": <locale>
  }],
  "template": <template>,
  "videoPrefix": <videoPrefix>
}
```

Next, the *FetchBookContent* Map that I explained in the previous section is run, as well as the *PrepareVividBatch* Lambda, which basically *reduces* all the information of the map and places it on a list, together with more useful information.

Up to this point, no QA has been done to the videos, but everything is set to start. The next step of the process is the main workflow. As it can be appreciated, there are two steps just before the *Parallel* state of the workflow, where the main logic is. I will expand on these two components, on the *Parallel* state and finally on the triggering logic of the workflow.

The FetchVideoContent Lambda

This Lambda is mainly responsible for returning good structured information of the video at a frame level so that the validators can use these information to compute the corresponding checks. To better understand the idea, the following are the input and output of this Lambda:

Input:

```
{
```

```
    'videoPrefix': <videoPrefix>,
    'videoName': <videoName>,
    'videoSuffix': <videoSuffix>,
    'input': {video contents (template specific)},
    'template': <template>
}
```

Output:

```
{
    'key': <videoKey>,
    'video_content': {
        frame_second: {text: [expectedStrings], images: [expectedImages]},
        ...
    }
}
```

So the idea is that according to the template, this component will determine the key frames of the video and return the expected information on each of the frames of the video: the text and the images. This step is crucial and most important, is the last step of the workflow that is template-specific. From now on, no processes depend on the type of video that needs to be validated, including the validators.

The FramesExtractor Lambda

This Lambda, as the name implies extracts the frames of a video. It takes as input the output of the *FetchVideoContent* Lambda, downloads the video from S3, extracts the corresponding frames, uploads the frames on S3 and returns a JSON of the form:

```
{
    'key': <videoKey>,
    'video_content': {
      'frame_second': {
        'text': [expectedText],
        'images': [expectedImages],
        'bucket_name': <bucket>,
        'key': <frameKey>
      }
    }
}
```

This in fact will be the input of all the validation steps under the *Parallel* state. The idea behind the frames extractor is that the videos that we produce are composed of n different frames, and we do know n. As a result, I implemented this Lambda using *ffmpeg* to extract the frames on the respective timestamps.

Main QA workflow (*Parallel* state)

As said before, the main idea is to implement some validators and run them in parallel to extract a final report of the video. Moreover, a pre-processing step may run just before some validation steps to provide the required input and ease the computations. For example, the *TextRecognition* Lambda is run before the *TextContentValidator*.

One important characteristic of the validators is that they are totally independent, meaning that they can perfectly run in parallel to optimize the computation time, and this is how I did it. As it it shown in the figure, there are actually two validators integrated, the *TextContentValidator* and the *GrammarValidator* or *SpellingValidator* (I will give more details in the corresponding sections).

And another important point of the design is that more validators can be easily added to the workflow, just by adding a branch under the *Parallel* state. As I said before, the input of all the elements under the *Parallel* state will be the same, according to the definition of *Parallel*. And the input will be the output of the *FramesExtractor* Lambda. The input is designed in such a way that all the validators can fetch the useful information to perform the computations. Right now there are only *text* validators but for instance, in a future we could add an *ImageOrientationValidator* that checks that the orientation of the images is correct and therefore the *images* field would be useful. An in all, the design is thought to be as flexible an as general as possible.

The following is how the output of a validator looks like:

```
{
  <VALIDATOR_NAME>_report: {
      'frame_second': 'SUCCESS'/<VALIDATOR_ERROR_CODE>,
      ...
      'result': 'SUCCESS'/<VALIDATOR_ERROR_CODE>
  }
}
```

Then, all the outputs are joint and two reports are created using the *ReportGenerator* Lambda. This Lambda will be detailed under the "Testing and Results" section.

Triggering Logic

As I briefly explained before, the idea is that this workflow is triggered by a CLI, that calls an API Gateway that triggers the workflow. With the CLI one can:

- Trigger the workflow
- Check the execution status of the workflow

To trigger the workflow, the following command can be run:

```
npm run -- wmf qa psv s3://<S3URI> <template> -r <region> -m <marketplace> -l <locale>
```

where the *region* is an AWS region, the *marketplace* is a marketplace ID and the locale is

an identifier for the country and the language.

To check the status of the workflow one can run the following command:

```
npm run -- wmf qa psv s3://<S3URI> <template> -r <region> -m <marketplace> -l <locale>
```

The above command checks the execution status of the workflow, that can have the following values:

- RUNNING: The execution is still running.
- SUCCEEDED: The execution has succeeded.
- FAILED: The execution has failed.
- TIMED_OUT: The execution has timed out.
- ABORTED: The execution has been manually stopped.

If the execution succeeded, the CLI automatically downloads the reports and returns the local location.

As for the API Gateway, the QA workflow step functions is exposed to the endpoint *batch-qa*, so qhen the triggering command is run on the CLI, this calls that endpoint and the workflow is triggered.

2.4 Main Components of the Workflow

2.4.1 The Text-Content Validator

The text content validator is the first validator that I implemented. It performs the following checks on a video:

- Checks that the text is correctly seen, not partially occluded. For instance, the case shown in figure 2.16 would be detected:
- The content of the text is correct. For instance, in the last part of a video with the children's books template there's the number of pages. So this validator would fail if the number of pages that is displayed on the video is not correct.
- There are no encoding errors. For instance, I found that a video had displayed the title as: *HowÂ OneÂ TherapistÂ andÂ aÂ CircleÂ of StrangersÂ SavedÂ MyÂ Life*, which is not correct.

As said previously, before this validator is run there is a pre-processing step that consists on extracting the text that is displayed on the video. This is done in the *text_recognition_lambda*. This lambda iterates over the video frames S3 locations, runs AWS Rekognition to detect the text on the frames and returns a payload with the following form:

```
{
    'key': <videoKey>,
    'video_content': {
      'frame_second': {
```

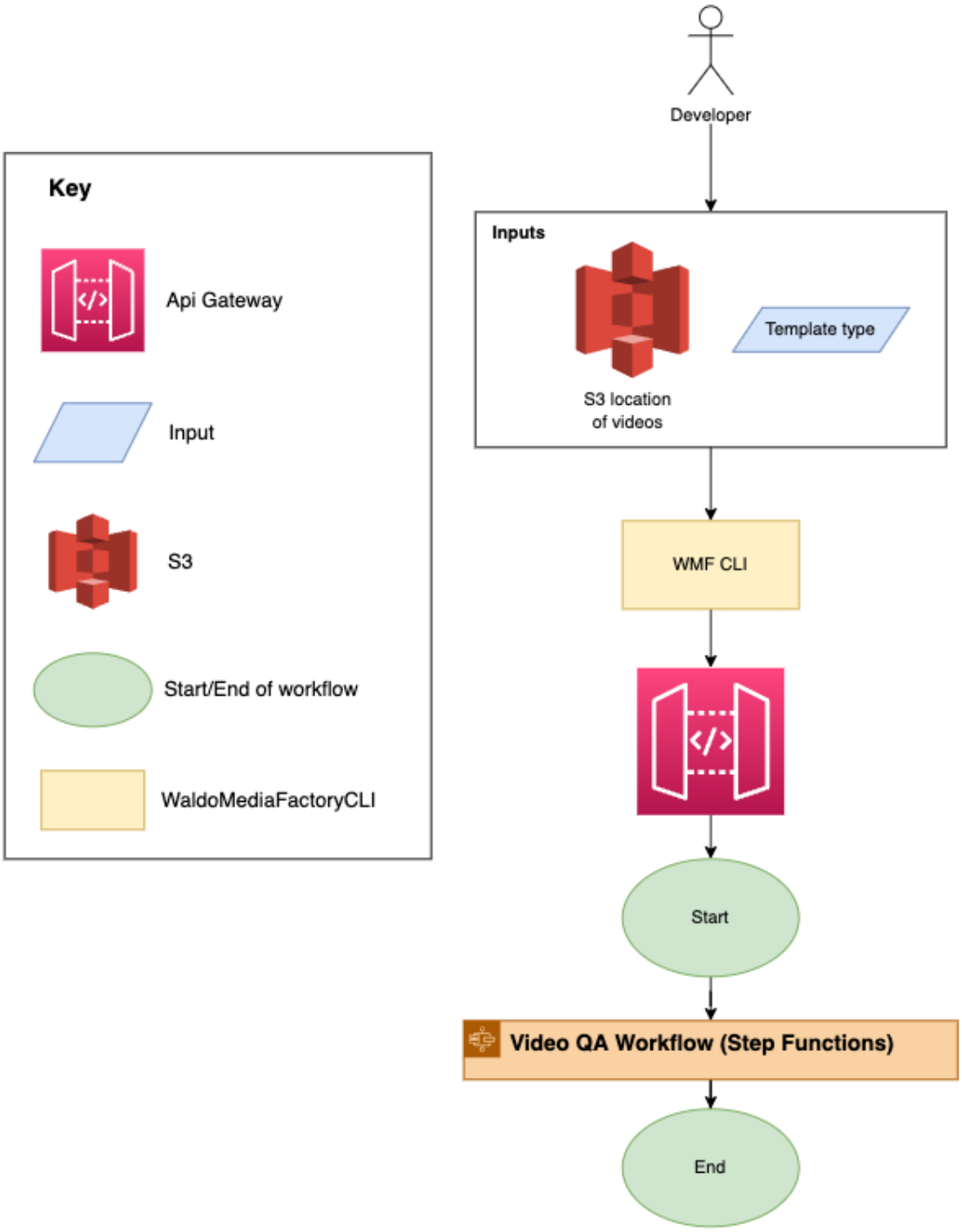


Figure 2.15: Triggering logic of the QA workflow

```
  'text': [expectedText],
  'images': [expectedImages],
  'bucket_name': <bucket>,
  'key': <frameKey>,
  'detected_text': <detectedText>
}
```

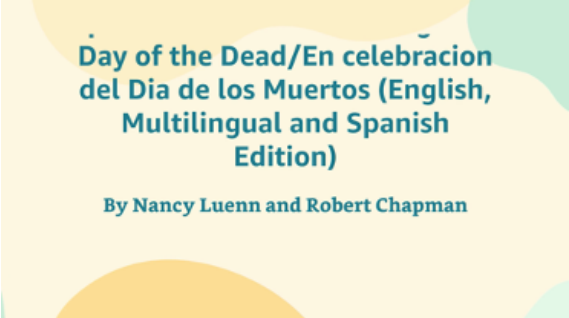


Figure 2.16: Video frame with cropped text

```
    }
}
```

This output is then directly forwarded to the *text_ content_ validator* lambda as an input as shown in figure 2.14. The following is the pseudo-code of this validator:

```
    for frame in video_frames:
        for string in expected_strings:
            remove non-braking spaces and HTML dashes from string
            expected_words = string.split()
            for word in expected_words:
                if word not in detected_text:
                    return False
    return True
```

The idea of the algorithm is to check word by word that is present on the detected text, because this check already ensures that the three checks stated above are performed. Moreover, the text recognition algorithm may not detect the text in the proper order and as a result the *detected_ text* string may not have sense. For instance, for the frame displayed in figure 2.17 we would need to check that the title and the author are correct. As a result the expected text is the following list:

```
    "expected\_text": [
        "The Night Watchman",
        "By Louise Erdrich"
      ],
```

But the *text_ recognition_ lambda* detects the following text on the video:

```
"detected_text": "Copyrighted Material \"A magisterial epic that brings her
↪  power of witness to every page.\" -Luis Alberto Urrea. New York Times
↪  Book Review THE NIGHT The Night Watchman WATCHMAN By Louise Erdrich A
↪  NOVEL LOUISE ERORICH Copyrighted Material"
```

Which is effectively present on the video frame but does not make sense as a sentence.

Lastly, on figure 2.18 it can be seen a high-level flow diagram of the text validation workflow.

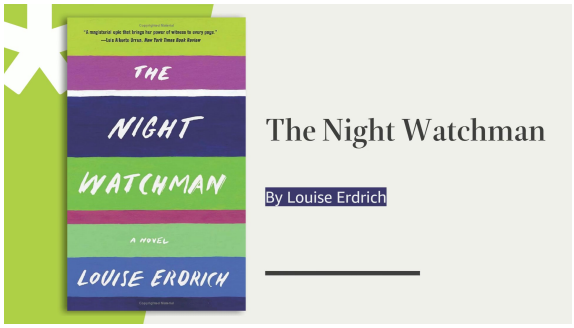


Figure 2.17: Video frame example of the fiction/non-fiction template

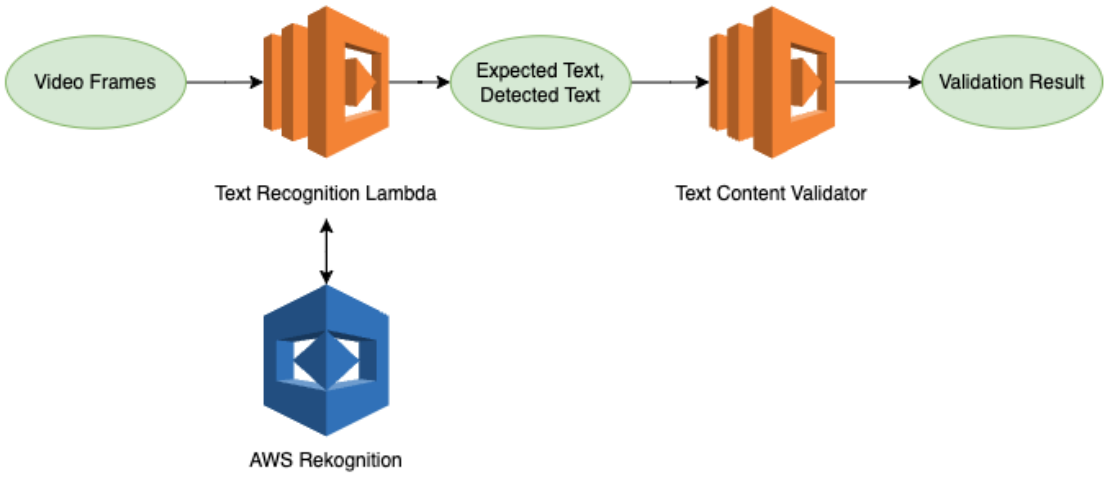


Figure 2.18: Text Content Validator workflow

2.4.2 The Spelling Valdidator

The *spelling_validator*, actually named as *grammar_validator* checks that the spelling of the text on the video is correct. In a future iteration the idea is that it also checks that the text is grammatically correct. It is important to note that this validation step runs against the *expected_text* on the video, not the *detected_text*, because the *text_content_validator* is already checking that the text is correct. Moreover, by checking the *expected_text* I ensure that I'm not carrying a possible error from the text recognition algorithm.

This validator's algorithm can be understood in the below pseudo-code. The idea of this algorithm is the following: for each frame, if the frame contains text, detect the language of the text, and this is done at a frame level because though in principle all the text in the video should be in the same language, there may be some quotes in other languages. The language is detected using AWS Comprehend and just after this step the corpus of the language is fetched, a big set of words in the detected language. Then, for each sentence of the text I detect the syntax using AWS Comprehend as well, to be able to filter proper nouns that may not appear on the corpus. This was seen in some tests that I did, in which some proper nouns weren not in the corpus, so I added this layer of validation to avoid these kind of false negatives.

Then, the text is processed word by word to check the spelling. The words are pre-processed by removing the punctuation and converted to lower-case. Then, the main step of the algorithm happens. I compute the edit distance or Levenshtein distance between the target word and the words in the corpus, and get the word with lowest distance. This distance is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other, so if if we find a word in the corpus with which the edit distance is 0, the target word is correctly spelled. This is the main idea. Nevertheless, I also had to work with some edge cases. For instance, if the word is in plural it does not appear in the corpus, so I also check the singular of the word and if it's a proper noun.

```
for frame in video_frames:
    if not text in frame:
        return True
    language = detect_language()
    corpus = get_corpus(language)
    for sentence in text:
        tags = detect_syntax()
        words = sentence.split()
        for word in words:
            # Remove punctuation from word, such as trailing commas
            word.strip(punctuation)
            if tags[word] = PROPER_NOUN or word.isdigit():
                continue
            word = word.lower()
            edit_distances = [edit_distance(word, w) for w in corpus]
            sort edit_distances
            # Get closest word
            correct_word = edit_distances[0]
            singular = singularize word and strip punctuation
            if (
                word != correct_word
                and singular != correct_word
                and singular.capitalize is not PROPER_NOUN
            ):
                return False
return True
```

Finaly, on figure 2.19 it can be seen a high-level flow diagram of the spelling validation workflow.

Figure 2.19: Spelling Validator workflow

2.4.3 The Video Flash Detector

The Video Flash detector is another tool that I built to validate the videos. The purpose of this validator is to check if the videos contain flashing sequences that could harm one's sight. This is an interesting problem that raises from the existance of the *Photosensitive Epilepsy (PSE)* disease. People with photosensitivity may have seizures, migraines or other adverse reactions to certain visual stimuli such as flashing images and alternating patterns, and around 50 million people worldwide have epilepsy, 3-5% of which had seizures triggered by luminance flashes or spatial patterns on images or videos.

Moreover, one may have this disease and may not know it until they get harmed by a flashing sequence. As a result, it is crucial to test any video that will be released to the public with a flashing detection algorithm to ensure that won't be harmful. In fact, in the U.K is compulsory to perform this kind of check before publishing any video. Another important point is that currently there's no free software available, and the prices quite high. For instance, the cost to evaluate a video with a duration under 2 minutes is around $30. As a result, the need to have this algorithm is more than justified.

To illustrate the importance of this issue, some examples of Photosensitive Epilepsy episodes are the following:

- In 1993, a broadcast advertisement *Golden Wonder Pot Noddles* triggered seizures in 3 viewers in the U.K.

- In 1997, the 25th episode of the anime *YAT Anshin!Uchu Ryokö* triggered seizures in 3 viewers.
- In 1997, the 38th episode of the 1^{st} season of *Pokemon* caused 685 seizures.
- In 2012, the London Olympic Games promotional film triggered seizures in 4 people.

To be able to introduce the algorithm, I will first give some formal definitions.

Definition 2.4.1. *Luminance is a photometric measure of the luminous flux density in a particular direction. The IS unit for luminance is Candela per square metre (cd/m^2).*

Definition 2.4.2. *Relative Luminance Relative luminance is equivalalent to luminance, but with the values normalized normalized as 0.0 to 1.0. For RGB colors, it can be computed with the following formula:*

$$Y = 0.2126R + 0.7152G + 0.0722B$$

Definition 2.4.3. *Gamma correction or gamma is a nonlinear operation used to encode and decode luminance in video or image systems. Gamma-compressed RGB values can be converted back to linear with the following formula:*

$$V(u) = \begin{cases} \frac{u}{12.92} & , u \leq 0.04045 \\ (\frac{u+0.055}{1.055})^{2.4} & otherwise \end{cases}$$

where u is a value of read, green or blue.

Definition 2.4.4. *Perceived Lightness is an approximation of the lightness response of human vision. It's defined as:*

$$L^*(Y) = \begin{cases} Y * 903.3 & , Y \leq 0.008856 \\ (Y^{1/3} * 116) - 16 & otherwise \end{cases}$$

Definition 2.4.5. *Flash sequences are sequences of three frames in which the first and last frame are significantly darker compared to the middle frame, or vice-versa.*

Definition 2.4.6. *We say that a sequence of flashes is harmful if we can find 3 or more flashes within a 1 second window.*

Figure 2.20 depicts a sequence of frames with three consecutive flashes.

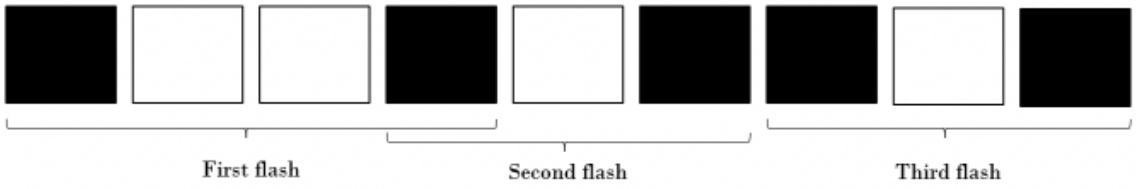


Figure 2.20: A sequence of frames, representing three consecutive flashes.

The pseudocode is the following:

```
for frame in video:
    Compute frame perceived lightness
for luma_frame in video:
```

```
    Compute the luminance variation, ΔL
    Compute average of luminance variation, ΔL‾
    Compare the signs of the previous and current variations.
    if different signs and  ΔL‾_acc >= ΔL_max then flag frame
    else ΔL‾_acc+ = ΔL‾
    Update prev variables
Flag frames that have more than three local extremes in 1s.
Harmful timestamps in seconds or Success if None
```

Lastly, as this algorithm requires to iterate over each of the pixels of the video, it is computationally expensive and therefore it can't be implemented using a Lambda, because of the time limit constraints. I tested the flash detector with one of the videos and took around 30 minutes to process. Moreover, as this validation step needs to run against all the frames of the video, it doesn't follow the pattern of the other validators. This validator is yet to be integrated to the workflow as the idea is that it can be used as a tool within the company and therefore the integration will be slightly different.

2.4.4 The Color Contrast Problem

As I explained under the Video Creation workflow section, we are using the *palette_ generator* lambda to generate a color palette that fits the cover of the book featured in the video. But we can't always ensure that this palette will be good enough. For instance, in the example in figure 2.21 the output of the algorithm contains only two distinguishable colors, but the intention was to obtain a palette containing a total of 5 colors. This is a problem because we may want to set one of the colors as the background another for text and another for some shapes, for example.

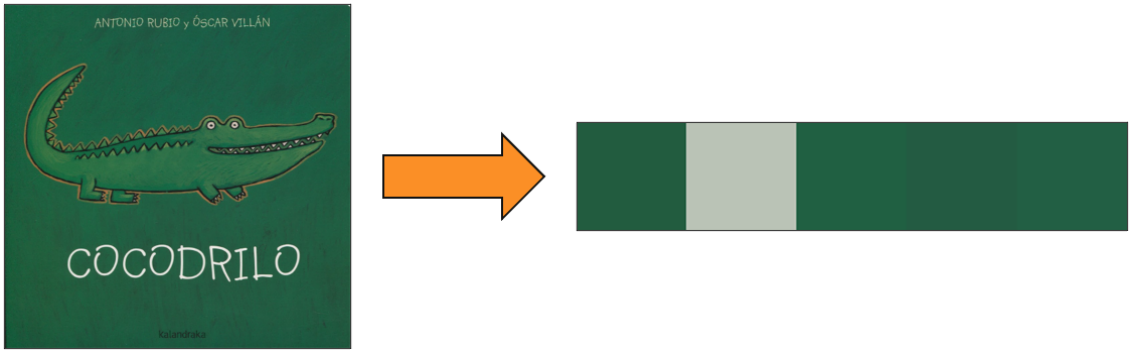


Figure 2.21: An example of a color palette without contrast between the colors.

The first step to solve the above problem is to be able to identify the *problematic* palettes. For that, I used the L*a*b* color space. The *CIELAB color space* also referred to as L*a*b* is a color space defined by the International Commission on Illumination (abbreviated CIE) in 1976. It expresses color as three values: L* for perceptual lightness, and a* and b* for the four unique colors of human vision: red, green, blue, and yellow. CIELAB was intended as a perceptually uniform space, where a given numerical change corresponds

to similar perceived change in color.

While the LAB space is not truly perceptually uniform, it nevertheless is useful in industry for detecting small differences in color. This means that this space is the most similar color space to the human eye. And that can help us identify up to which point the colors contained in a palette are different.

For that, we need to talk about the Delta E distance. The Delta E is a distance that takes values in [0, 100] between two colors defined in the L*a*b space. And with this two preliminaries we can get to the alpha score.

Definition 2.4.7. *The equation of the α score is as follows:*

$$\alpha = \frac{\sum_{i=1}^{N} \Delta E^*_{\mu c_i}}{100N}$$

where:

- *N is the number of colors.*
- *For every i, c_i is a color.*
- $\mu = \frac{\sum_{i=1}^{N} c_i}{N}$.
- $\Delta E^*_{\mu c_i}$ *is the Delta E (CIE76) distance between μ and c_i.*

This score will tell us up to what extent the colors present on a palette are distinguishable. In the following example (figure 2.22) it can be seen that the score decreases as the quality of the palette does.

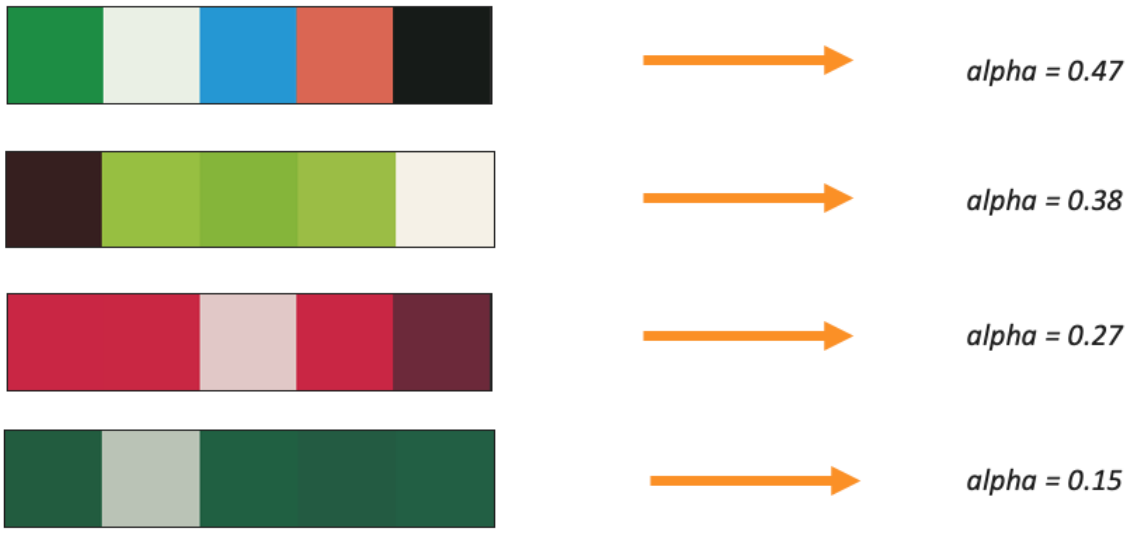


Figure 2.22: Alpha scores for four types of input palettes.

We just defined a score to compute the goodness of a color palette, but what do we do in case that the colors of the palette are undistinguishable? For that, we can implement Waldorama. The main idea is that given the most dominant color on the book cover, find n more suitable colors to that color, and generate a palette containing these colors.

For this approach, the colors are represented using the HSL coordinates (Hue , Saturation and Lightness), because then it's much easier to perform the rotations to obtain the colors.

As it will be seen later on in the visual example, this approach always ensures that the resulting palette will have different colors. As a result, although this algorithm does not compute the most appealing palettes, it solves the contrast problem and could be applied on a later stage if needed.

The pseudo-code for this approach is the following:

```
color = get_mpst_dominant_color(book_cover)
palette = [color]
hue = random hue value
# Define S and L at their mid point to avoid grayscale colors
saturation = 0.5
lightness = 0.5
step = 360 / (num_colors) - 1
for i in range(num_colors - 1):
    hue = (hue + step) % 360
    Weight S and L 80% on 0.5 and 20% on a random value
    new_color = (hue, saturation, lightness)
    colors.append(new_color)
return colors
```

For instance, if we suppose that $num\_colors = 4$, we first add the most dominant color to the palette and the three remaining colors are calculated in the following way, where 1 is a color with random hue and all the colors have the saturation and lightness of around 0.5. This schema can be seen in figure 2.23.

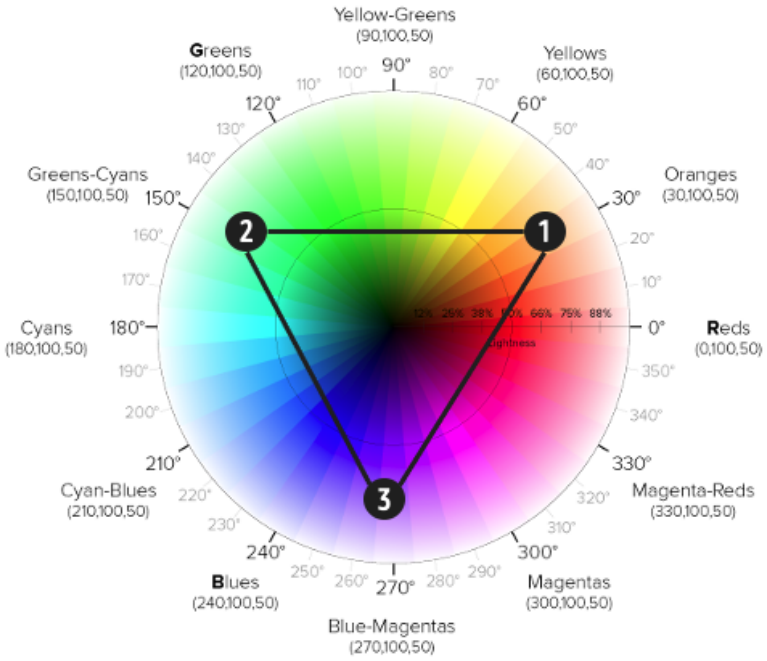


Figure 2.23: A color contrast problem solution illustrated.

And the following (figure 2.24) is an output example using the same book cover with which I obtained a palette with poor contrast.

While this is not a proper validation step, it is an algorithm that can be used to build a *color_contrast* validator, because as I said before the workflow is flexible and it is easy to add new validation steps. As a result, this validator could fetch the color palette and determine with the α score if there is enough contrast. Alternatively, the algorithm for obtaining a palette with contrast could be added to the creation workflow to reduce the number of failures.

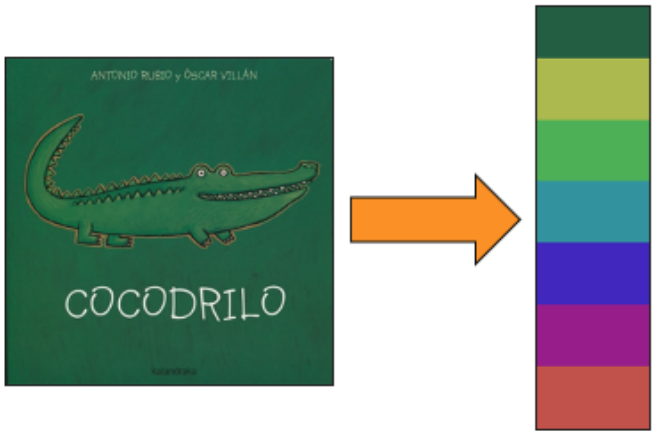


Figure 2.24: Output example of the exposed solution for the color contrast problem.

2.5 Workflow Integration: AWS CDK

The AWS Cloud Development Kit (AWS CDK) is an open-source software development framework to define cloud application resources using familiar programming languages. In my case, I used Typescript to define the necessary infrastructure ans resources for the application. It provides high-level components called constructs that pre-configure cloud resources with proven defaults, so you can build cloud applications with ease. AWS CDK provisions your resources in a safe, repeatable manner through AWS CloudFormation. It also allows you to compose and share your own custom constructs incorporating your organization's requirements, helping you expedite new projects.

Figure 2.25 shows how AWS CDK works. Inside a CDK application you can define Stacks, and inside each Stack have constructs, which are cloud components that represent architectures of any complexity, such as a single resource (S3, Lambda, SNS,...) or a multi-stack application. In my case, there is a construct for every AWS region, named *scope*.

As for the integration of the QA workflow to the existing architecture, I created a stack, *video-qa-stack.ts* and defined all the required resources, including:

- **Lambda**: I defined all the lambda functions that are needed to run the work-

flow, including: PrepareQABatch, FetchVideoContent, FramesExtractorPreprocessor, TextRecognitionPreprocessor, TextContentValidator, GrammarValidator, ReturnFrames and the ReportGenerator.

- **S3**: I defined an S3 bucket to store the output reports and the video frames.
- **IAM Policy Statements**: I needed to define a total of six policy statements to allow some Lambdas perform operations such as reading and writing from S3, running AWS Rekognition or running AWS Comprehend. Figure 2.26 shows an overview of how the architecture looks like, including the policy statements attached to the corresponding lambda functions.
- **State Machine**: This is the Step Functions workflow.
- **IAM Role**: Invocation role assumed by API Gateway do that it can trigger the workflow.

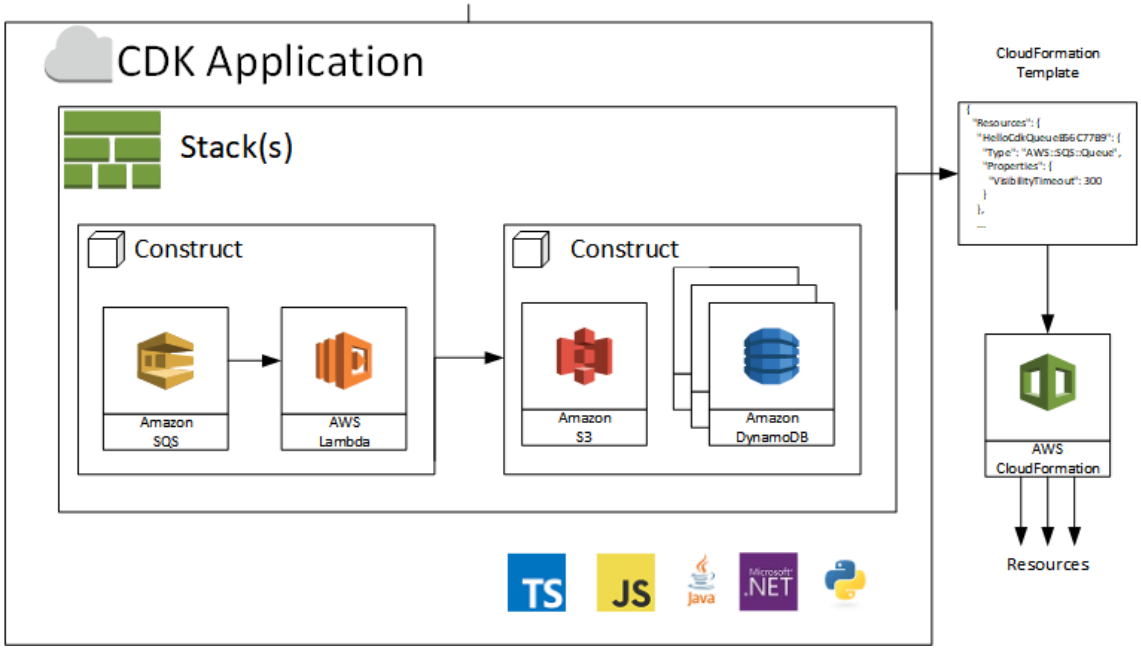


Figure 2.25: AWS CDK application diagram. Source: https://aws.amazon.com/blogs/aws/aws-cloud-development-kit-cdk-typescript-and-python-are-now-generally-available/

As seen in figure 2.26 as well, all the lambdas are placed *inside* the Video QA Step Functions workflow, because they are part of it. Another important point to mention is that the Video Bucket is an S3 bucket that contains the videos. This bucket is already created as part of the Video Creation Stack but has to be referenced in three policy statements because some of the lambdas need access to that bucket. Lastly, although the API Gateway is present on the diagram, it is not created as part of the Video QA Stack, in fact it has its proper Stack in which all the endpoints are defined. I placed it to illustrate that I am creating an invocation role that enables to call the step functions workflow and attach it to the API Gateway.

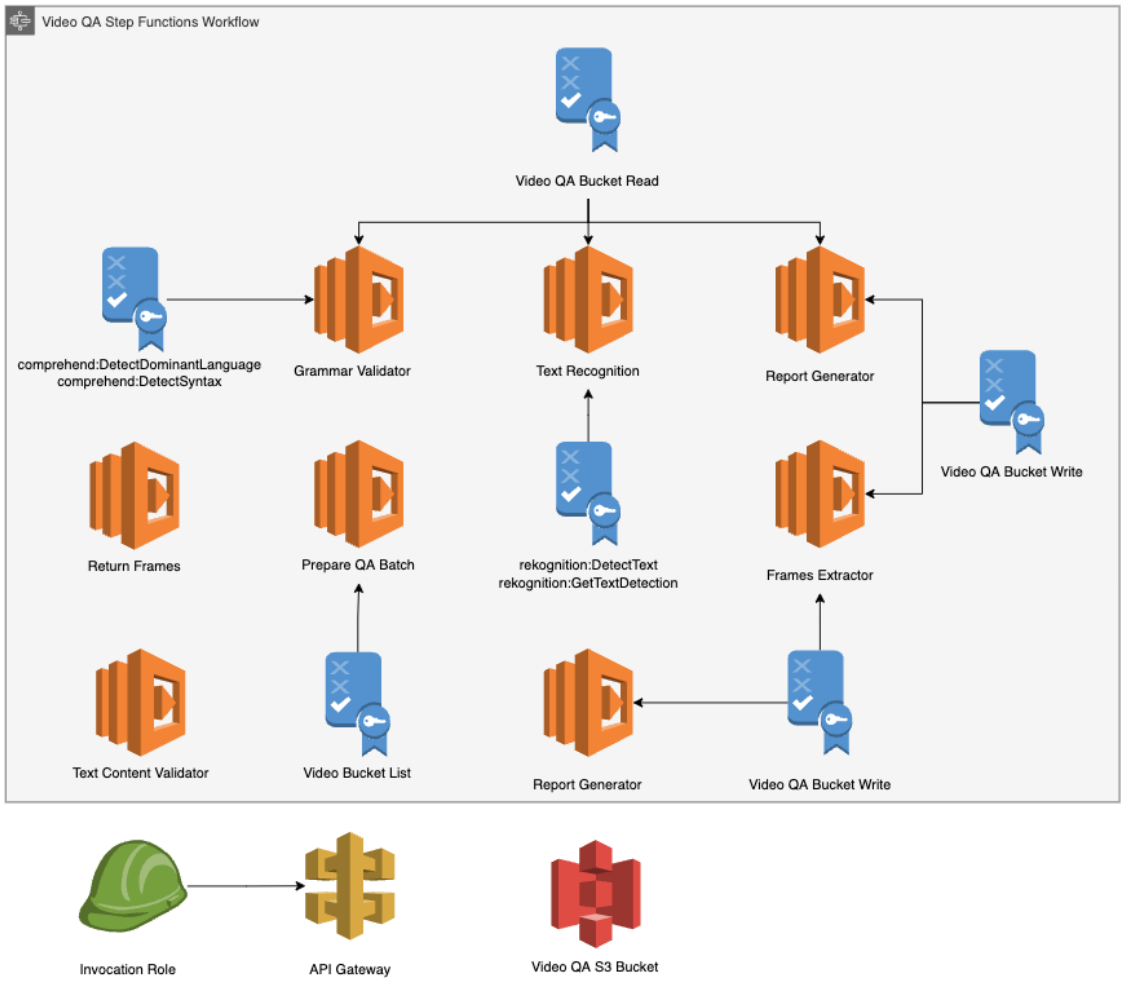


Figure 2.26: Resources created using AWS CDK for the Video QA workflow

2.6 Testing and Results

2.6.1 The Report Generator Lambda

This lambda is the last step of the workflow. It can be seen as the reduce step from the map-reduce operation that is done in the QA workflow. It mainly fetches all the validation results for all the videos and outputs two reports:

- **JPG report**: The JPG report contains the key frames of all the videos horizontally stacked. This report is useful because it enables to check the videos faster and see if for example, an image is missing or the colors of the video do not fit with the displayed images.
- **CSV report**: The CSV report contains a summary of the validation outputs at a frame level for each of the processed videos.

As it returns information at a video batch level, the input is of the form:

```
[
  [
```

```
    [
      {
        'key': <video_key>,
        'frames': [{'bucket_name': <bucket_name>, 'key': <frame_key>}]
      },
      {
        <validator_name>_'report': {
            'frame_second': 'SUCCESS'/<validator_error_code>,
            ...
            'result': 'SUCCESS'/<validator_error_code>
        }
      }
    ],
    [Next video validation outputs]
  ],
  [Next batch of videos validation outputs]
]
```

Note that the input list is organized in batches of videos. This is because on the *Prepare-VividBatch* Lambda (see figure 2.14) the videos are split into batches due to the request limit of the video renderer (Vivid).

Next I will show some results for the two types of templates that are currently supported by the workflow:

Example for the Children's Books template

Figure 2.27 depicts a reports for two Children's Books videos. This means that the input of the QA workflow were two video and as a result the output reports contain information about these videos. As said before, the JPG report consists of an image with the key frames of the videos. So these videos contain a total of six key frames.

Figure 2.27: JPG report for two Children's Books videos

As for the CSV report, it can be seen in figure 2.29. While the second video doesn't have any problem, the *TextContentValidator* fails for the last frame of the first video. This is because the text that is shown in the video is not correct. So the input data for that frame was the following:

```
"32": {
```

```
    "bucket_name":
    ↪  "videoqastack-gamma-us-ea-videoqagammauseast15ebd5-1wmbif6echwhi",
    "key": "video-frames/067088278X.TT85.PSV.MP4/sec_32.jpg",
    "text": [
      "Viking Books for Young Readers",
      "Book published by",
      "Preview for Board book (32 pages) Other formats: Audible
      ↪  Audiobook, Books, Hardcover and more"
    ],
    "images": [],
    "detected_text": "Preview for Board book (34 pages) Other formats:
    ↪  Audible Audiobook, Hardcover, Kindle Edition and more Book
    ↪  published by Puffin amazon"
}
```


And as it can be seen, the text differs from the detected text. Figure 2.28 shows this frame bigger so the text can be appreciated.

Figure 2.28: Last frame of video 067088278X.TT85.PSV.MP4

Frame (second)	result	3	7	13	17	24	32
test/067088278X.TT85.PSV.MP4	INCORRECT_TEXT	SUCCESS	SUCCESS	SUCCESS	SUCCESS	SUCCESS	INCORRECT_TEXT
test/0711245630.TT85.PSV.MP4	SUCCESS	SUCCESS	SUCCESS	SUCCESS	SUCCESS	SUCCESS	SUCCESS

Figure 2.29: CSV report for two Children's Books videos

Example for the Adults's Books template

Figure 2.30 is the JPG report for two Adult's Books videos. In this template the videos have six key frames as well, and the main difference is that this template contains far more text than the Children's Books template, because all the frames contain some text.

In figure 2.31 it can be seen that both videos fail the two validation checks. Let's first analyze the first video.

On video 1408867044.TT85.PSV.MP4, both validators fail at the frame on second 26, which is depicted on figure 2.32. The reason behind is the word "worng", which is misspelled (it should be wrong). The *TextContentValidator* fails because the text on the database contains the word correctly spelled and the *SpellingValidator* fails because it detects that this

Figure 2.30: JPG report for two Adults's Books videos

Frame (second)	result	3	13	20	26	32	40
test/1408867044.TT85.PSV.MP4	INCORRECT_TEXT,GRAMMAR_ERROR	SUCCESS	SUCCESS	SUCCESS	INCORRECT_TEXT,GRAMMAR_ERROR	SUCCESS	SUCCESS
test/1432885855.TT85.PSV.MP4	INCORRECT_TEXT,GRAMMAR_ERROR	SUCCESS	GRAMMAR_ERROR	INCORRECT_TEXT	SUCCESS	SUCCESS	SUCCESS

Figure 2.31: CSV report for two Adults's Books videos

word is not correctly spelled. One could then ask why are we using the *SpellingValidator* if the *TextContentValidator* is already detecting these issues, but that is because it could happen that the text in the database is already misspelled.

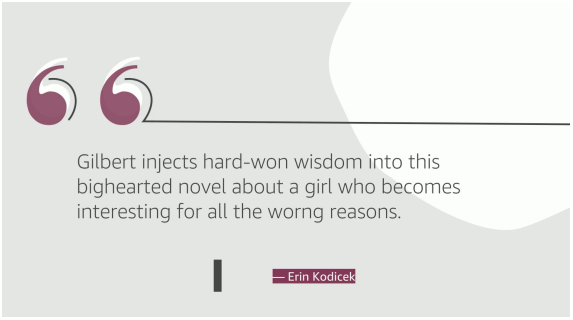


Figure 2.32: Frame on second 26 of video 1408867044.TT85.PSV.MP4

On the other hand, the second video has erros on second 13 and on second 20. Second 13 of video 1432885855.TT85.PSV.MP4 can be seen under figure 2.33. The validator that fails in this case is the *SpellingValidator*. This is because it detects this "dash" character between the words "families" and "the", and interprets it as a hyphenated word, which obviously doesn't exist.

Figure 2.33: Frame on second 13 of video 1432885855.TT85.PSV.MP4

In the case of second 20, the *TextContentValidator* is failing because the text of the second accolade is cut on the bottom, and the validator fails because the expected text contains the text that cannot be seen on the video.

Figure 2.34: Frame on second 20 of video 1432885855.TT85.PSV.MP4

Lastly, on figure 2.35 I show a report for a total of eleven videos to illustrate how a bigger reports would look like.

Figure 2.35: JPG report for eleven Adults's Books videos

Video Flash Detector Tests

Under the code folder I placed two videos in the directory:

```
VideoFlashDetector/videos
```

For **video_1.mp4**, I computed the following output:

```
This video has failed the Harding Test.
The following timestamps in seconds have been identified as harmful:
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

As it can be checked, these are all the seconds of the video. The algorithm identifies the whole video as potentially harmful because of the continuous flashing sequences that are present on the clip. On the other hand, for **video_2.mp4**, the following seconds are identified as harmful:

```
This video has failed the Harding Test.
The following timestamps in seconds have been identified as harmful:
{10, 15, 16, 17, 18, 19, 20}
```

And these timestamps of the London 2012 promotion video are the most harmful.

The script can be tested using the following command in case of **video_1.mp4**, and equivalently for **video_2.mp4**:

```
python flash_detector.py --videokey video_1.mp4
```

2.6.2 Integration Tests

As I said on the CI/CD section of the project, integration tests run on each stage of the pipeline (except Prod) and need to succeed in order to let the changes promote to the next environment. As this is a totally new workflow, I needed to write integration tests. For that, I created a package called *WaldoVideoQALambdaTests* and I wrote a couple of integration tests, one for the Children's Books template and another one for the Adult's Books template. These tests consist on triggering the workflow on the environment in which they are running to validate one video, and the workflow needs to return the execution status SUCCEEDED in order to make the test pass.

Chapter 3

Conclusions

This chapter is divided into two sections. In the first one I will explain the conclusions about the project and evaluate the outcome, discussing whether the proposed objective has been met or not. And lastly, I will give an overview of possible future work that can be done to scale the solution.

3.1 Project Conclusions

The following outcomes have been achieved:

- Having built an end-to-end architecture to perform QA on videos.
- The workflow is ready to be used in production.

3.2 Future Work

There are two main points that can be expanded:

- Adding more validation steps: As I explained, the designed architecture supports the addition of more checks, and these could include for instance, a check to ensure that the featured images in the video are not rotated, or a step to check that there is enough contrast between the colors on the video, using for example the algorithm that I proposed on the corresponding section.
- Supporting new templates: More templates can be supported as the architecture is flexible and one can add validation for a new template fairly easily.
- Integration of the Video Flash Detector on the workflow: This validation step needs to be integrated to the main QA workflow as for now is just implemented. Moreover, as I explained before, this can't be a Lambda because of the execution time constraints so a possible solution would be to set it up as a container.

Bibliography

[1] George Cybenko. Approximations by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2:183–192, 1989.

[2] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[3] Ronen Eldan and Ohad Shamir. The power of depth for feedforward neural networks. In *Conference on learning theory*, pages 907–940. PMLR, 2016.

[4] Moritz Hardt, Ben Recht, and Yoram Singer. Train faster, generalize better: Stability of stochastic gradient descent. In *International conference on machine learning*, pages 1225–1234. PMLR, 2016.

[5] Lúcia Carreira, Nelson Rodrigues, Bruno Roque, and Maria Paula Queluz. Automatic detection of flashing video content. In *2015 Seventh International Workshop on Quality of Multimedia Experience (QoMEX)*, pages 1–6. IEEE, 2015.

[6] Natalie Nylund. A photosensitive epilepsy flash pattern detection algorithm, 2020.

[7] Dalitso Hansini Banda. *Deep video-to-video transformations for accessibility applications*. PhD thesis, Massachusetts Institute of Technology, 2018.

[8] E Weinan. Machine learning and computational mathematics. *arXiv preprint arXiv:2009.14596*, 2020.

[9] Pac learning. `https://en.wikipedia.org/wiki/Probably_approximately_correct_learning`.

[10] Michael M. Wolf. Mathematical foundations of supervised learning, 2022.

[11] Universal approximation theorem. `https://en.wikipedia.org/wiki/Universal_approximation_theorem`.

[12] Bayes classifier. `https://en.wikipedia.org/wiki/Bayes_classifier`.

[13] Empirical risk minimization. `https://en.wikipedia.org/wiki/Empirical_risk_minimization`.