APPENDIX

Abstract

This paper explains the implementation of SDK Extensions working together with the Clarifai SDKs. Extensions are software modules that can register themselves against the SDK and allow the SDK to function is ways not feasible without them. Extensions must conform with with a software interface, by which the SDK can invoke functions, send/receive data, and notify of events. During the lifecycle of a call to the SDK, an extension can be invoked when certain operations will, are, or did take place.

The expansion in functionality of the SDK, via extensions, also allows for the entire system to function as a marketplace; a mediating agent between two or more transacting actors, and create a new product category in the AI space.

1. Introduction

The Clarifai SDK is an inference engine offering the service of Artificial Intelligence (AI) to general purpose apps. The inference engine can run predictions utilizing base or custom models[1], or it can train new local models, which are incremental learnings leveraging the knowledge of existing models. By embedding a Clarifai SDK in their app, customers can add AI to their respective apps, without having to have expertise in-house to build an inference engine, nor training models themselves. An SDK can contain one of more specialized models to run inferences on. E.g. a general purpose model, a face detection model, and more.

The standard modus operandi of the SDK is to take an asset (e.g. image, video) as an input parameter, run it through an inference engine (producing a prediction with a certain level of confidence), and return results containing the output asset and a list concepts[2] with the highest probabilities (steps 1 and 2 in figure 1).

[1] Base models are models trained in the Clarifai cloud and offered to all customers. Custom models are special purpose trained models, available to a subset of customers.

[2] In the Clarifai lexicon, and A.I. in general, Concepts are labels, paired with a level of confidence, assigned to an item inferred from a given asset.

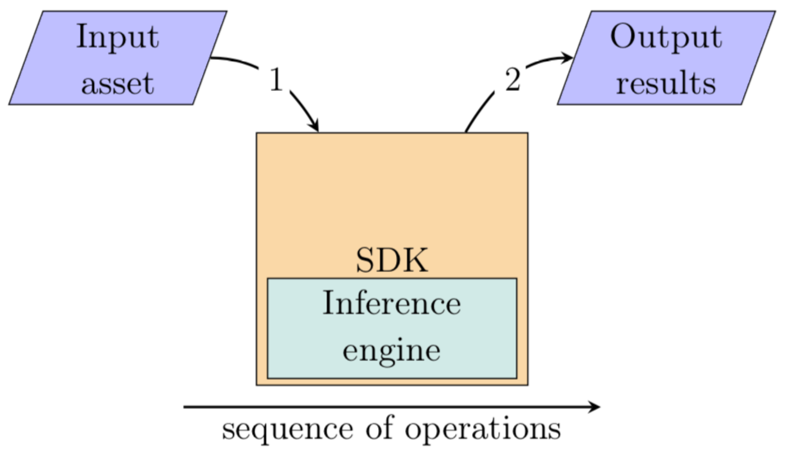

Figure 1: *SDK's standard modus operandi*

The operations are unidirectional. The input asset goes from the app to the SDK, and the results go from the SDK to the app.

As useful as this may be, the SDK, as presented so far, can only handle straight forward workflows where an asset can be sent directly to the inference engine. It does not address workflows where in-between operations are needed in order to execute successful predictions. More specifically, there are three workflows not addressed. They are:

1. Pre-processing: The input assets needs to be processed before being sent to the inference engine, but without modifying the original asset
2. During processing: The app needs to be notified that an input asset is being processed
3. Post-processing: The output asset needs to be processed, after the inference took place, but before the list of results is returned

2. SDK Extensions

SDK extensions is the solution to overcome the shortfalls in the SDK's standard modus operandi. Extensions are a self-expanding mechanism, built-in in the SDK, by which any developer can implement software modules to augment the functionality of the SDK.

Extensions are optional modules, built with special purpose, that get registered against the Clarifai SDK, via a set of rules defined in a software interface.

Clients and business partners can choose which SDK extensions to adopt in their apps, based on their business agreements with each other and/or third parties.

The extensions modus operandi allows for the SDK to support all three workflows that were previously not possible.

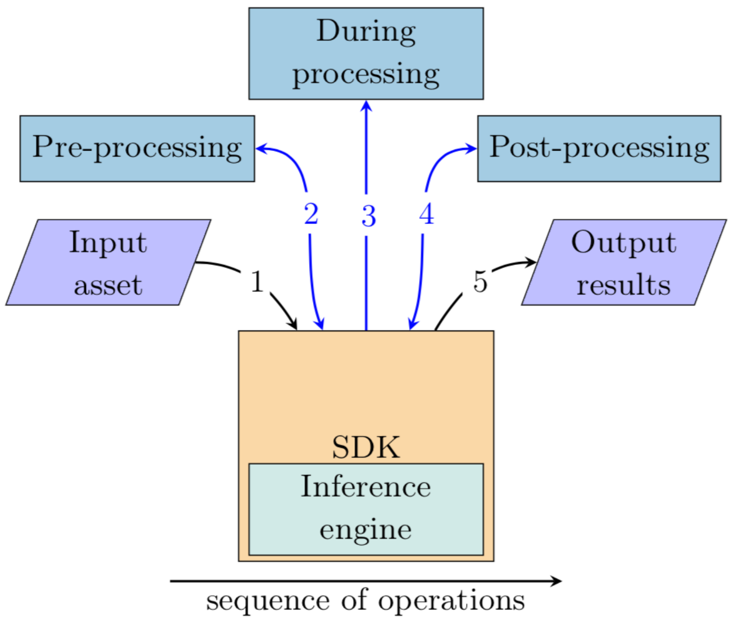

Figure 2: *SDK extensions' modus operandi*

Steps 1 and 5, shown in figure 2, work the same as previously described. We now introduce three new,in-between, steps.

In step 2, after an asset has been sent as an input parameter, but before an inference, the SDK would invokepre-processing extensions, they would operate on the asset and return the result to the SDK.

Next, the SDK would notify during processing extensions (step 3) that it is running a prediction on an asset.

After the inference is complete, and the results have been computed, but before sending the results back to the app, the SDK would invoke the post-processing extensions (step 4). They would operate on the asset and return to the SDK.

Note that steps 2 and 4 are bidirectional. Information flows from the SDK to the extensions and back.

2.1. Use cases

Let's take a look at three use cases for SDK extensions. One for pre-processing, one for during processing, and one for post-processing. In each of the following cases, we explain and see them in action; we also introduce the concept of a Marketplace, which we will explore in more details later in this paper.

In the first use case, the SDK is embedded in a app (CleverOptometry) for eye exams. Using the app, a doctor would take a picture of the patient's face and the picture would be sent to the SDK. The SDK would invoke a pre-processing extension (developed by Snapchit), passing the original image as parameter and informing it that an inference will take place–but did not happen yet. The extension would enlarge the patient's eyes and return a new image to the SDK with the transformation. The original image is preserved and the inference engine can run a prediction on the image transformed by the extension.

In the second use case, and app (Facetube) is processing a bulk of photos and needs to show a progress bar to indicate evolution of the task. The SDK would invoke a during processing extension (available for download from Clarifai) and inform that a photo is being processed, how many photos have been processed already, how many are still left to be processed, and the total number of photos in the task.

The third use case is an app for cataloging images (LenseStock). It needs to apply watermarks on their images, for copyrighting reasons, but such watermarks do interfere with the predictions. The app can, instead, send non-watermarked images to the SDK, so predictions are accurate. After predictions are completed, but before the results are returned, the SDK would invoke a post-processing extension (develop internally by LenseStock) to apply the watermark to the images and return them to the SDK.

3. Architecture

There are two fundamental points in the architecture of SDK extensions:

Static Registry : Defines a data structure for extensions to register against the SDK, implements functions to manage the registry, and keep references to all registered extensions

Interfaces : Define the software contracts between the SDK and extensions. Extensions inherit from and conform to the interface protocols by implementing all of the required functions and data structures

During the launching process of an app (mobile, desktop, etc.), static variables are initialized prior to its execution. By registering extensions statically (via public static methods), the SDK will become aware of all extensions from the moment it starts running.

The following data structures start building the foundation of SDK extension registration.

```
Enum ExtensionType {
      PreProcessingType,
      DuringProcessingType,
      PostProcessingType
}

struct Extension {
      string id
      ExtensionType type
      int priority = 100
      bool isActive = true
}
```

Algorithm 1: Foundation data structures for registering an extension against the SDK.

ExtensionType is an enumeration specifying what kind of extensions exist and are allowed to be used in the SDK. The Extension structure contains three variables:

id : Unique identifier of the extension. It can be defined by a developer, but also can be initialized with a nullvalue. In that case, the SDK will automatically generate a unique identifier to the extension

ExtensionType : The type of the extension. This is a protected variable and read-only to developers. It is set automatically depending on what extension specialization will be implemented (see algorithm 4)

priority : Establishes the order in which each extension, of the same ExtensionType, would get invoked. The lower the number, the higher the priority. By default it gets initialized with 100

isActive : By default an extension is active. In case it needs to be disabled during runtime, a developer, or the SDK, can set this to false

3.1. Automatically setting priorities

In regards to the priority of an extension, the SDK offers a very important feature. It will rank priorities automatically based on the results from an inference.

By trying the possible permutations of extensions ($p = n!$, where n is the number of registered extensions of that type), the SDK will pick the permutation that produces the highest inference results, and save that order for future use.

For example, assume there are 3 pre-processing registered extensions ($E_1, E_2, and\, E_3$). There are six ($3! = 6$) possible permutations: $[E_1, E_2, E_3]$, $[E_1, E_3, E_2]$, $[E_2, E_1, E_3]$, $[E_2, E_3, E_1]$, $[E_3, E_1, E_2]$, and $[E_3, E_2, E_1]$. The SDK will run an inference with each of them and select the one producing the highest levels of confidence in the results.

```
Function bestPermutation(extensions)
    perm = extensions
    bestPermutation = perm
    bestInference = [0, 0,..., 0]

    while perm is not null do
        inf = runInference(perm)
        if inf is better than bestInference then
            bestInference = inf
            bestPermutation = perm
        end
        perm = nextPermutation(extensions)
    end
    return bestPermutation
End
```

Algorithm 2: Compute which permutation produces the best inference results.

After computing which permutation yields the best inference, the SDK can set the priority of each extension. Starting with 100 for the extension with the highest priority (invoked first), and continuing by incrementing the priority by 1 (100 + 1, 100 + 2,..., 100 + n) for each subsequent extension, until the one with the lowest priority (invoked last).

```
extensions = *bestPermutation*([ $E_1, E_2, ..., E_n$ ])

**Function** *setPriorities*(extensions)
    numExtensions = size(extensions)
    priority = 100

    **for** idx = 1 *to* numExtensions **do**
        **if** $E_{idx} \geq 100$ **then**
            $E_{idx}$ = priority
            priority = priority +1
        **end**
    **end**
**End**
```

Algorithm 3: Set priorities of extensions based on which was the best permutation.

Developers may also choose to override the SDK’s automatic selection of priorities. By setting the priority to a number smaller than 100 (priority < 100), the SDK will respect the order of execution in which the extensions were programmed.

3.2. Implementation

Implementing an extensions starts with defining a class that inherits from one of the following three data structures, shown in algorithm 4:

```
struct AssetItem {
    Asset asset
    unsigned int batchSize
    unsigned int position
}
struct PreProcessing : Extension {
    type = PreProcessingType
    willProcessAsset(AssetItem) → Asset
}
struct DuringProcessing : Extension {
    type = DuringProcessingType
    isProcessingAsset(AssetItem)
}
struct PostProcessing : Extension {
    type = PostProcessingType
    didProcessAsset(AssetItem) → Asset
}
```

Algorithm 4: Base data structures for extensions.

Each data structure (*PreProcessing*, *DuringProcessing*, and *PostProcessing*) inherits from the *Extension* data structure, set the respective *ExtensionType*, and add a function that will be invoked by the SDK with an *AssetItem* as parameter.

PreProcessing and *PostProcessing* have functions which are expected to return an *Asset*, whereas *DuringProcessing*'s function does not return anything.

AssetItem is a data structure containing an *Asset* (e.g. image, sound), the size of the batch being processed, and the relative position of the current asset.

In algorithm 5 we show the pseudo implementation of a post-processing extension. The code in the *Watermark* class will overlay a semi-transparent copyright protection note on the original image, after it has been through the inference engine.

```
class Watermark : PostProcessing {
        ...
        didProcessAsset(AssetItem) → Asset
}

Watermark watermark {
        id = "gnu314"
}

Clarifai::registerExtension(watermark)
```

Algorithm 5: Registering a post-processing extension against the SDK.

The *Watermark* class inherits from *PostProcessing* and implements the required function (didProcessAsset). Then, the watermark instance gets initialized with *id = "gnu314"*, *type = PostProcessingType* (inherited from the *PostProcessing* data structure), and *priority = 100* (inherited from the *Extension* data structure).

The watermark instance gets registered against the SDK by passing itself as a parameter when calling the SDK's *registerExtensions()* static function.

The diagram in figure 3 shows the SDK with several registered extensions. Each extension instance implements an Interface[3] and is kept in the *Registry*.

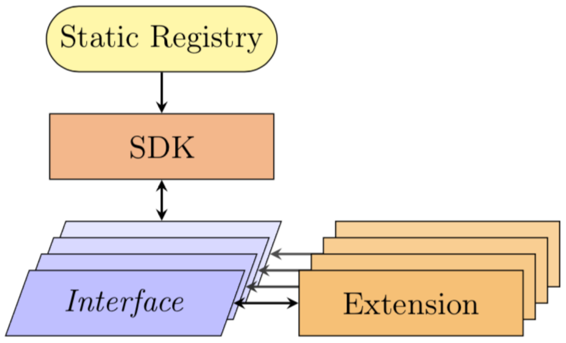

Figure 3: *SDK with extensions*

During the runtime, the SDK will function as described in the workflow shown in figure 2, where for each asset sent to the inference engine, the SDK will in turn invoke the applicable extensions and proceed to the next step.

[3] More than one extension can implement the same interface, and one extension can implement more than one interface.

3.3. Operational lifecycle

The activity diagram shown in figure 4 presents typical steps in the lifecycle of an app using the SDK with extensions. It starts with launching the app, registering extensions, and entering the app's runtime[4].

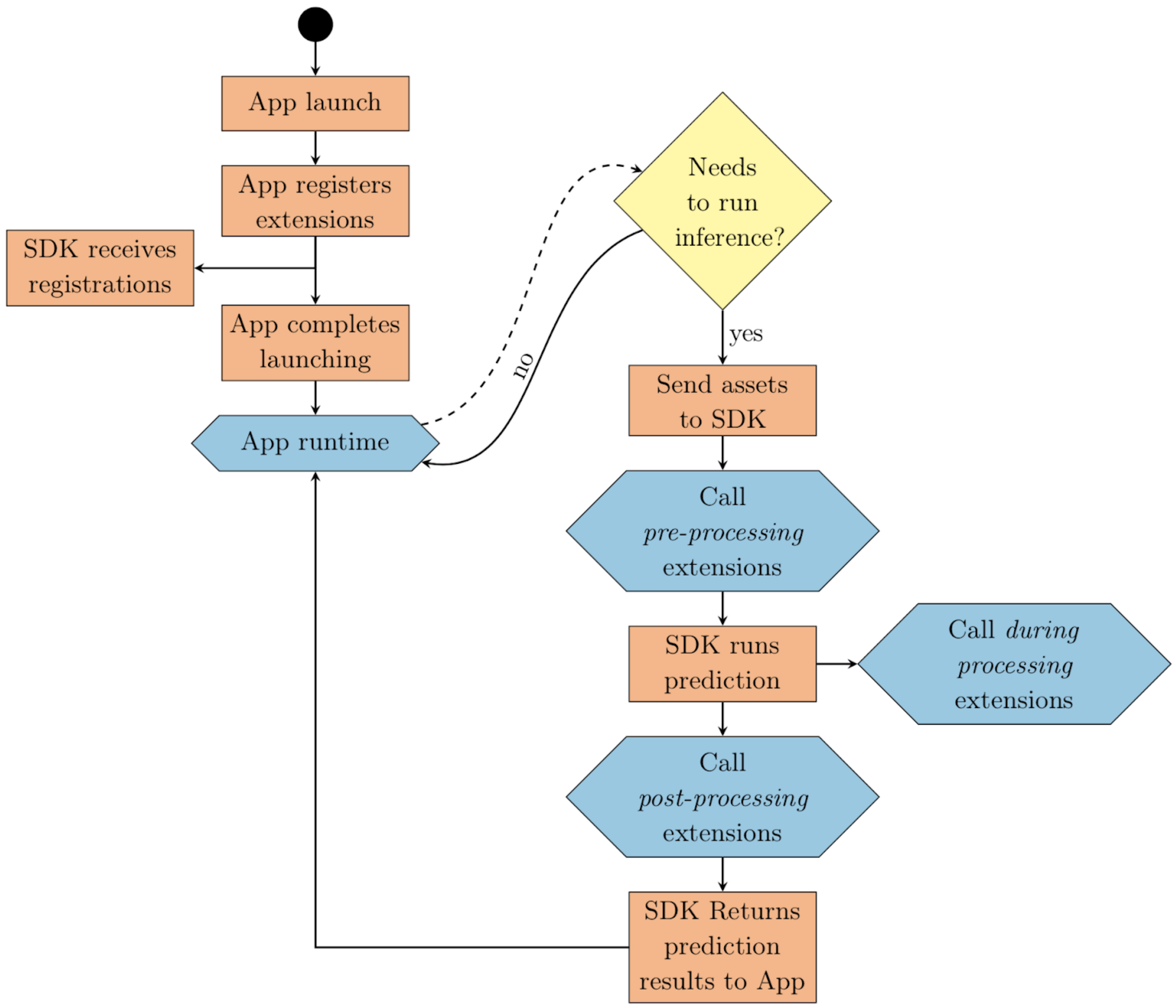
App launch
App registers extensions
SDK receives registrations
App completes launching
App runtime
Needs to run inference?
no
yes
Send assets to SDK
Call *pre-processing* extensions
SDK runs prediction
Call *during processing* extensions
Call *post-processing* extensions
SDK Returns prediction results to App

Figure 4: *SDK Extensions registration and runtime diagram.*

Every time the app needs to run an inference, it will send assets to the SDK. In turn the SDK will call all available and enabled pre-processing extensions (according to their respective priorities), run predictions on the assets and call all available and enabled during processing extensions, and before returning the results to the app, the SDK will call all available and enabled post-processing extensions.

[4] Runtime is the stage during which the system is operation.

4. Marketplace

A marketplace is an environment where transactions take place, usually exchanging goods or services. A marketplace does not create, nor commits transactions. However, it facilitates the happening of transactions by removing friction, providing exposure and recognition of value.

Extensions, as proposed in the SDK, do function as marketplaces. In addition to the augmentation of the capabilities of the SDK, extensions create and foster the environment where two or more parties (Clarifai and/or third-parties) can commit and exchange service transactions amongst themselves. Transactions that would not be possible to take place without the SDK.

4.1. Would manual search work?

By now you may be thinking that one could argue that a user could take a picture of a sofa, go to another app, search for a coffee table and a rug, copy those images and try to make a collage with all of them together. The problems with this argument are:

- It requires too many steps and a very determined user. There are too many friction points and disincentives. This would not scale. The extension allows for the computer to do the heavy lifting
- The user would need to know what they want beforehand. The extension, on the other hand, is allowing for suggestions to be made
- It requires too much time and focus from the user. They may get sidetracked by the extra searches, collage, and end up changing their minds. The extension would save time and keep the user in the app

4.2. Transaction hub

The SDK, augmented with extensions, becomes a vital part in enabling transactions amongst third parties. It increases the perception of value added.

Note: Even though the extensions in the SDK would facilitate transactions, they do not keep a ledger, nor record anything other than analytics information. It is no the intent, nor purpose of extensions, to provide accounting services or do arbitration between third parties.

5. Claims

We claim:

1. A system for optimizing results of processed assets for provision to general-purpose software applications based on determined sequences of operation, the system comprising:

a computational unit engine configured with at least one processor and non-transitory processor readable media having instructions that are executable by the at least one processor;

a plurality of models, accessible either over a communication network, or locally via a storage unit, by the engine, that are each usable by the engine to provide artificial intelligence in connection with general-purpose software applications;

a plurality of extensions received and executed by the engine to operate in accordance with a respective model for at least one of the general-purpose software applications; and

a plurality of assets provided to the engine to be processed as a function at least one of the models and a set of the plurality of extensions, wherein a given sequence of executing the set of extensions on a single one of the assets impacts results provided by the engine;

wherein the engine is configured by executing the instructions stored on the processor readable media to:

(a) execute each of the plurality of extensions in the set in a plurality of respective, different sequences to grade a respective degree of inference associated with results of each of the respective sequences;
(b) select an optimum one of the plurality of sequences as a function of determining a highest grade among the sequences;
(c) compute the most likely probability as a function of results from executing the plurality of extensions in the set, in accordance with the optimum sequence; and
(d) provide, to at least one of the general-purpose software applications, results associated with processing the assets by the engine in accordance with the average probability.

2. A system for pre-processing assets (e.g. scaling, transforming), without modifying the original ones, immediately before those assets being used in a general-purpose sequence of computational operations
3. A system for broadcasting software notifications to one or more other other listening software agents, be them delivered locally on the same computational unit engine, or on a remote computational unit engine via a network, regarding an asset that is being processed, but without interrupting the computation operation being executed
4. A system for post-processing assets (e.g. watermarking, augmenting), either modifying the original asset, or not (preference to be determined by the SDK Extension implementation). The post-processing operations do not affect the results produced by the general-purpose sequence of computational operations