Pre/Post Processing and DLQ in RunInference

This document will outline several options for the desired user experience for 2 RunInference features: (1) using a Dead Letter Queue and (2) adding pre/postprocessing mapping functions. While these are distinct features, they have some interactions and similar properties so I'm addressing them together in this doc.

Question 1: How to define DLQ and pre/postprocessing functions

The first question we will address is how the user will define the behavior that they intend to use. There are two places we can put each piece of functionality: in the model handler or in the transform itself.

Option 1 - Everything in ModelHandler

Our first option is to stick all parameters into the ModelHandler:

```
mh = XYZModelHandler(<normal_args>, preprocess_fn=preprocess,
postprocess_fn=postprocess, use_dlq=True)

good, bad = pcoll | RunInference(mh)
```

This is the simplest extension to our existing user experience because all config can be contained in the model handler itself and reduces the need for users to look in different locations for information about changing the behavior of RunInference, but it comes with several downsides:

First, the number of parameters returned by RunInference is now disconnected from the transform itself in a non-obvious way. For example, despite being almost identical to the original example, the following would be invalid:

```
mh = XYZModelHandler(<normal_args>, preprocess_fn=preprocess,
postprocess_fn=postprocess)

good, bad = RunInference(mh)
```

This is just because we changed the model handler. So any change to the model handler's `use_dlq` parameter will require a corresponding change to the `RunInference` call itself.

In fact, the most common way of using `RunInference` with chaining:

```
return (pcoll
| beam.Map(<func>)
| RunInference(mh)
| beam.Map(<func>))
```

breaks if you introduce a dead letter queue. Any change to the DLQ parameter will require a corresponding change to how we call RunInference, so it is odd to separate them entirely. It also won't save users from needing to learn this Beam concept since they'll need to receive multiple PCollections and handle their dead letter queue (store + reprocess) anyways.

Secondly, making pre and post processing functions a parameter of the ModelHandlers adds significant complexity in how we define our types. Currently, we define RunInference to be a transform that goes from an `ExampleT` to `PredictionT`, with those types defined by the Model Handler.

```
class RunInference(beam.PTransform[beam.PCollection[ExampleT],
                                   beam.PCollection[PredictionT]]):
 def __init__(
     self,
     model_handler: ModelHandler[ExampleT, PredictionT, Any],
     ...
 )
```

If we add custom preprocessing functions to the model handler itself, typing becomes much more difficult and we likely won't be able to offer the same level of type safety that we offer today. Since we can't change the ModelHandler contract, each ModelHandler will only be able to expose 3 values - the example type, the prediction type, and the model type. At best, then, the model handler would be able to expose an example type of `Union[CurrentExampleT, InputTypeToPreprocess]`. Since the input type to the preprocess function can be any type, this would need to be a generic type, and the typing is not sophisticated enough to infer the actual underlying type of a subclass like this. So we would not be able to offer any input typing guarantees.

In contrast, if we define the type on RunInference itself, we can easily infer the type information:

```
class RunInference(beam.PTransform[beam.PCollection[Union[ExampleT, PreT]],
                                   beam.PCollection[Union[PredictionT,
                                                          PostT]]]):
 def __init__(
     self,
     model_handler: ModelHandler[ExampleT, PredictionT, Any],
     ...
```

```
    preprocess_fn: Optional[Callable[[PreT], ExampleT]] = None,
    postprocess_fn: Optional[Callable[[PredictionT], PostT]] = None):
```

So we are not able to make the same typing guarantees with this approach that we currently provide. This means that users will not receive typing errors until runtime, and they will potentially surface in unclear ways

Lastly, if we define these properties as part of the ModelHandlers instead of RunInference, every new ModelHandler will have to implement support for them. While this can be done by overriding a function defined by the base class, it introduces overhead for all authors and many custom model handler authors will likely not implement this.

SubOption a - Add with_<pre/post>processing_fn as a function on base.ModelHandler

Rather than defining our pre/postprocessing functions as parameters to the model handler, we can add them as functions on our model handler:

```
mh =
XYZModelHandler(...).with_preprocess_fn(preprocess).with_postprocess_fn(postprocess
)
```

where these functions have signatures like:

```
ModelHandler[ExampleT, PredictionT, Any].with_preprocess_fn(Callable[[ExampleX],
ExampleT] -> ModelHandler[ExampleX, PredictionT, Any]
```

This allows us to maintain strong typing, and allows us to easily compose multiple pre/postprocessing functions:

```
mh =
XYZModelHandler(...).with_preprocess_fn(pre1).with_preprocess_fn(pre2).with_preproc
ess_fn(pre3)
```

Option 2 - Everything in the RunInference transform

Our second option is to put all of our configuration in `RunInference`.

This has the disadvantage of requiring config in 2 places - the model handler and the transform. At the same time, it moves the DLQ config to the same place that DLQ records must be handled and allows us to use stronger typing. Additionally, introducing keyword parameters to

RunInference itself allows us to introduce new features to all ModelHandlers, bringing value to custom Model handlers and allowing Beam developers to more easily scale.

Within this option, there are 2 possible approaches for handling the DLQ:

SubOption a - DLQ as transform parameter

First, we can add the DLQ as a transform parameter alongside the pre/post processing functions.

```
good, bad = pcoll
            | RunInference(
                mh,
                preprocess_fn=mult_two_pre,
                postprocess_fn=mult_two_post,
                use_dlq=True
              )
```

SubOption b - DLQ using with_exception_handling

Second, we can add the DLQ using with_exception_handling. This allows us to attach our existing pattern for handling DLQs elsewhere in the Python SDK. It also provides an easy mechanism for providing additional parameters, like `use_subprocess`.

```
good, bad = pcoll
            | RunInference(
                mh,
                preprocess_fn=mult_two_pre,
                postprocess_fn=mult_two_post
              ).with_exception_handling(use_subprocess=True)
```

Option 3 - Mix and Match

Our last option is that we could put pre/postprocessing in the Model Handler and the DLQ in RunInference, or vice versa. This would look like either:

```
mh = XYZModelHandler(preprocess_fn=preprocess)

good, bad = pcoll
            | RunInference(
                mh,
                use_dlq=True
```

or:

```
mh = XYZModelHandler(use_dlq=True)

good, bad = pcoll
            | RunInference(
                mh,
                preprocess_fn=preprocess
              )
```

This comes with all the pros and cons for each method mentioned above.

Decision

We will proceed with option 3 and mix and match suboption 1a for the preprocess function and suboption 2b for the DLQ:

```
mh =
XYZModelHandler(...).with_preprocess_fn(pre1).with_preprocess_fn(pre2).with_preproc
ess_fn(pre3)
good, bad = pcoll
            | RunInference(
                mh
              ).with_exception_handling(use_subprocess=True)
```

This has the following advantages:
1) It keeps the DLQ near where its used
2) It allows us to maintain strong typing
3) It creates pattern that can be reused to easily scale support for custom model handlers and our existing model handlers
4) It attaches to the existing DLQ pattern and easily allow passing in parameters like `use_subprocess`
3) It allows for chaining of pre/postprocessing operations

It does come at the cost of requiring users to provide config in 2 places: the ModelHandler and the transform itself.

Q2: How to handle DLQ responses with pre/postprocessing

The second question we will address is how to handle RunInference instances with multiple subtransforms. Specifically, if we have pre and/or post processing operations, we can represent our DLQ as returning 2 PCollections no matter what:

```
good, bad = pcoll
            | RunInference(
                mh,
                preprocess_fn=mult_two_pre,
                postprocess_fn=mult_two_post
              ).with_exception_handling(use_subprocess=True)
```

returning 1 PCollection per operation:

```
good, bad_preprocess, bad_inference, bad_postprocess = pcoll
            | RunInference(
                mh,
                preprocess_fn=mult_two_pre,
                postprocess_fn=mult_two_post
              ).with_exception_handling(use_subprocess=True)
```

or returning 1 pcollection, and one object containing all the error pcollections as fields:

```
good, bad = pcoll
            | RunInference(
                mh,
                preprocess_fn=mult_two_pre,
                postprocess_fn=mult_two_post
              ).with_exception_handling(use_subprocess=True)
bad_preprocess = bad.failed_preprocessing
bad_inference = bad.failed_inferences
bad_postprocess = bad.failed_postprocessing
```

Returning just 2 PCollections has the advantage of being simpler, but doesn't provide as much expressiveness as the first option.

Returning one PCollection per operation allows users to understand which transform the DLQ is returning from so that they can act appropriately. For example, if an item fails during preprocessing, the correct action may be to just feed it back into the pipeline. If an item fails during postprocessing, the user likely doesn't want to feed it back into the pipeline as an example.

Returning an object with member PCollections provides the same level of expressiveness, but avoids exploding the number of parameters. It also gives us room to apply a similar pattern in the future.

Because of its additional expressiveness, we will proceed with the third option:

```
main, other = pcoll
            | RunInference(
                mh,
                preprocess_fn=mult_two_pre,
                postprocess_fn=mult_two_post
              ).with_exception_handling(use_subprocess=True)
bad_preprocess = other.failed_preprocessing
bad_inference = other.failed_inferences
bad_postprocess = other.failed_postprocessing
```

Note: If with_exception_handling is not set RunInference will return a single pcoll as per existing signature.

Future Work

In the future, we will extend `with_exception_handling` to accept an object that contains configuration on where to write results (e.g. kafka, gcs, etc…) and will automatically write the results as part of our pipeline.

We will also consider retrying individual records that fail as part of a batch in RunInference to isolate the failures to a tighter set of examples.