# Model Upload

**Learn how to perform model upload using Clarifai SDKs**

Users can now upload their custom-built models into production using Clarifai SDKs. The Clarifai SDKs offers features like a command-line interface, easy implementation, and testing in Python to make the process of deploying the model easier.

Additionally, for serving configurations, the `serving_backend` section contains custom settings, including options for NVIDIA Triton. With Triton, users can leverage high-performance GPU computation for inference tasks. NVIDIA Triton Inference Server stands out as a powerful and versatile platform. It streamlines the deployment and execution of machine learning models for inference tasks, offering a professional solution for developers and data scientists seeking to bridge the gap between model development and real-world applications. Its emphasis on framework flexibility, performance optimization, scalability, and ease of integration makes it a compelling choice for maximizing the impact of machine learning models across various industries.

## Prerequisites

- Setting up the Clarifai SDKs along with PAT. Refer to the installation and configuration with the PAT token here.

NOTE

Guide to get your PAT

- Create a project directory named `your_model_dir` and run the following commands,

```
clarifai create model --type text-to-text --working-dir your_model_dir
cd your_model_dir
```

The Clarifai SDKs will then create all the necessary files required for the deployment process inside `your_model_dir` .

```
your_model_dir
```

```
├── clarifai_config.yaml
├── inference.py
├── test.py
└── requirements.txt
```

**INFO**

- inference.py: The crucial file where users need to implement their Python code.
- clarifai_config.yaml: Contains all necessary configurations for model test, build and upload
- test.py: Predefined test cases to evaluate inference.py.
- requirements.text: Equivalent to a normal Python project's requirements.txt.

## Implementation

The next step involves the implementation of an inference class inside `inference.py` , for your custom model. There are 2 functions inside the class that you must implement:

- **init**: a method to load the model, called once.
- **predict**: a function designed to generate predictions based on the provided inputs and inference parameters. This method includes a docstring inherited from its parent, providing information on input, parameters, and output types. Refer to the docstring to confirm that the outputs of this method adhere to the correct Clarifai Output Type, as errors may occur otherwise.

Below is an example template of inference.py for a text-to-text model,

**Python**

```python
class InferenceModel(TextToText):
  """User model inference class."""

  def __init__(self) -> None:
    """
    Load inference time artifacts that are called frequently .e.g. models, tokenizers, etc.
    in this method so they are loaded only once for faster inference.
    """
    # current directory
    self.base_path: Path = os.path.dirname(__file__)

  def predict(self, input_data: list,
              inference_parameters: Dict[str, Union[str, float, int, bool]]) -> list:
```

```
    """ Custom prediction function for `text-to-text` (also called as `text generation`) model.

    Args:
      input_data (List[str]): List of text
      inference_parameters (Dict[str, Union[str, float, int, bool]]): your inference parameters

    Returns:
      list of TextOutput

    """

    raise NotImplementedError()
```

Consider a scenario where we are going to use a HuggingFace Text-To-Text model, the inference class would look like this:

Python

```
import os
from typing import Dict, Union
from clarifai.models.model_serving.model_config import *

import torch
from transformers import AutoTokenizer
import transformers

class InferenceModel(TextToText):
  """User model inference class."""

  def __init__(self) -> None:
    """
    Load inference time artifacts that are called frequently .e.g. models, tokenizers, etc.
    in this method so they are loaded only once for faster inference.
    """
    # current directory
    self.base_path = os.path.dirname(__file__)
    # where you save hf checkpoint in your working dir e.i. `your_model_dir`
    model_path = os.path.join(self.base_path, "checkpoint")
    self.tokenizer = AutoTokenizer.from_pretrained(model_path)
    self.pipeline = transformers.pipeline(
        "text-generation",
        model=model_path,
        torch_dtype=torch.float16,
        device_map="auto",
```

```python
    )

  def predict(self, input_data: list,
              inference_parameters: Dict[str, Union[str, float, int]]) -> list:
    """ Custom prediction function for `text-to-text` (also called as `text generation`) model.

    Args:
      input_data (List[str]): List of text
      inference_parameters (Dict[str, Union[str, float, int]]): your inference parameters

    Returns:
      list of TextOutput

    """
    output_sequences = self.pipeline(
        input_data,
        eos_token_id=self.tokenizer.eos_token_id,
        **inference_parameters)

    # wrap outputs in Clarifai defined output
    return [TextOutput(each[0]) for each in output_sequences]
```

## Setup Configuration File

The `clarifai_config.yaml` contains all the required files for testing, building and uploading a model.

The config file will have the following structure,

```yaml
clarifai_model:
  clarifai_model_id:
  clarifai_user_app_id:
  description:
  inference_parameters: (*)
  labels: (*)
  type: (**)
serving_backend:
  triton: (***)
    max_batch_size:
    image_shape:
```

The `clarifai_model` includes configurations necessary for the building, testing, and uploading process. It comprises several attributes: `clarifai_model_id`, which specifies the model ID on the platform; `clarifai_user_app_id`, which denotes the user ID and App ID on the platform, separated by '/'; and `description`, providing a brief model description. These attributes are crucial for the model upload process, though if not provided, they can be passed during the upload command. Additionally, there are optional attributes: inference_parameters, allowing customization of model prediction methods for testing and uploading purposes; and labels, requiring manual insertion of concept names essential for specific model types. The type attribute, generated upon initializing the working directory, must not be modified.

For serving configurations, the `serving_backend` section contains custom settings, including `triton` options such as `max_batch_size`, determining the maximum number of inputs for prediction, and `image_shape`, applicable solely for image input models, specifying the width and height of input images. The default max_batch_size is 1, but if the model supports batch inference, it can be adjusted for enhanced GPU performance. The image_shape default is [-1, -1], indicating acceptance of any image size.

## Testing

This test serves two primary purposes aimed at optimizing the testing and validation procedures:

- Implementation Validation: Before proceeding with the build or upload operations, users can utilize this feature to conduct a comprehensive assessment of their implementation. This step ensures the accuracy and preparedness of the model for deployment. The test encompasses the validation of custom configuration settings outlined in the `clarifai_config.yaml` file.

- Inference Parameter Management: Users are provided with the convenience of adding or updating inference parameters directly within the `clarifai_config.yaml` file. Additionally, the system performs automatic validation during the inference process to guarantee the accuracy and compatibility of these parameters with the model's requirements. The test ensures that only defined inference parameters with appropriate values can be utilized.

Example test case for text input,

**Python**

```python
def test_text_input(self):
  text: list = ["Tell me about Clarifai", "How deploy model to Clarifai"]
  outputs = self.model.predict(text, temperature=0.9) # In term of inference parameters for the above example, it will PASSED
  outputs = self.model.predict(text, top_k=10) # And this one will FAILED since `top_k` param is not defined when init self.model
```

Click here to know more about test files and clarifai_config.yaml file.

Each model built for inference with triton requires certain dependencies & dependency versions to be installed for successful inference execution. Therefore the next step is to add the required dependencies into `requirements.txt` file.

```
clarifai
torch=2.1.1
transformers==4.36.2
accelerate==0.26.1
```

## Deployment

In order to prepare for deployment we will have to build the files. This process will generate `*.clarifai` zip file which contains all the necessary files to get your model to work on the Clarifai platform.

```
clarifai build model
```

NOTE

You need to upload your built file to a cloud storage service in order to obtain a direct download URL for the next step.

Since we have all the files ready let's proceed to deploy the model using the following commands,

```
clarifai login
```

▸ Output

```
clarifai upload model --url <url> --user-app <your_user_id>/<your_app_id> --id
<your_model_id>
```

## Example

For the demo, we are going to upload an open-source visual segmentation model from Huggingface. Run the following commands step by step on a Google Colab instance or your local machine.

The first step is to install the required libraries,

```
pip install clarifai
```

Using the Clarifai CLI users can initialize a model from the Clarifai Examples repository into your working directory by executing the following,

```
clarifai create model --from-example --working-dir my_dir
```

NOTE

The `--working-dir` parameter will create a directory.

From the list of available models let's choose a visual segmenter as an example,

▸ Image Output

The CLI will then clone all the required files for `visual_segmenter` directly onto the working directory.

▸ Image Output

Once we are inside the `my_dir` directory, we can download the model checkpoint from HuggingFace into a `checkpoint` directory.

```
huggingface-cli download mattmdjaga/segformer_b2_clothes --local-dir
my_dir/checkpoint --local-dir-use-symlinks False --exclude *.safetensors
optimizer.pt
```

▸ Output

Next, install the dependencies from the requirements.txt file,

```
pip install -r my_dir/requirements.txt
```

Before moving on to the build process we have to make some changes in the `clarifai_config.yml` file. You will have to add `clarifai_model_id` and `clarifai_user_app_id` with the respective values. Example changes made to the `clarifai_config.yml` file are given below,

```
clarifai_model:
  clarifai_model_id: 'segmentation_model'
  clarifai_user_app_id: '8tzpjy1a841y/transfer_learn_3'
  description: ''
  inference_parameters: []
  labels:
  - background
  - hat
  - hair
  - sunglass
  - upper-clothes
  - skirt
  - pants
  - dress
  - belt
  - left-shoe
  - right-shoe
  - face
  - left-leg
  - right-leg
  - left-arm
  - right-arm
  - bag
  - scarf
  type: visual-segmenter
serving_backend:
  triton:
    max_batch_size: 4
```

After installing the dependencies and modifying the config file, we have to build the model and upload the `model.clarifai` file to cloud storage.

```
clarifai build model ./my_dir
```

▸ Output

NOTE

You can use the model from this URL for the model upload demo: `https://s3.amazonaws.com/samples.clarifai.com/model.clarifai`.

Now let's log in to the Clarifai using CLI,

```
clarifai login
```

▸ Output

The last and final step is to upload the model onto Clarifai’s platform,

```
clarifai upload model my_dir --url
https://s3.amazonaws.com/samples.clarifai.com/model.clarifai
```

▸ Output

Build your next AI app, test and tune popular LLMs models, and much more.

Get started for free

clarifai