Deploying Machine Learning models to production — Inference service architecture patterns

Assaf Pinhasi
Oct 17 · 13 min read

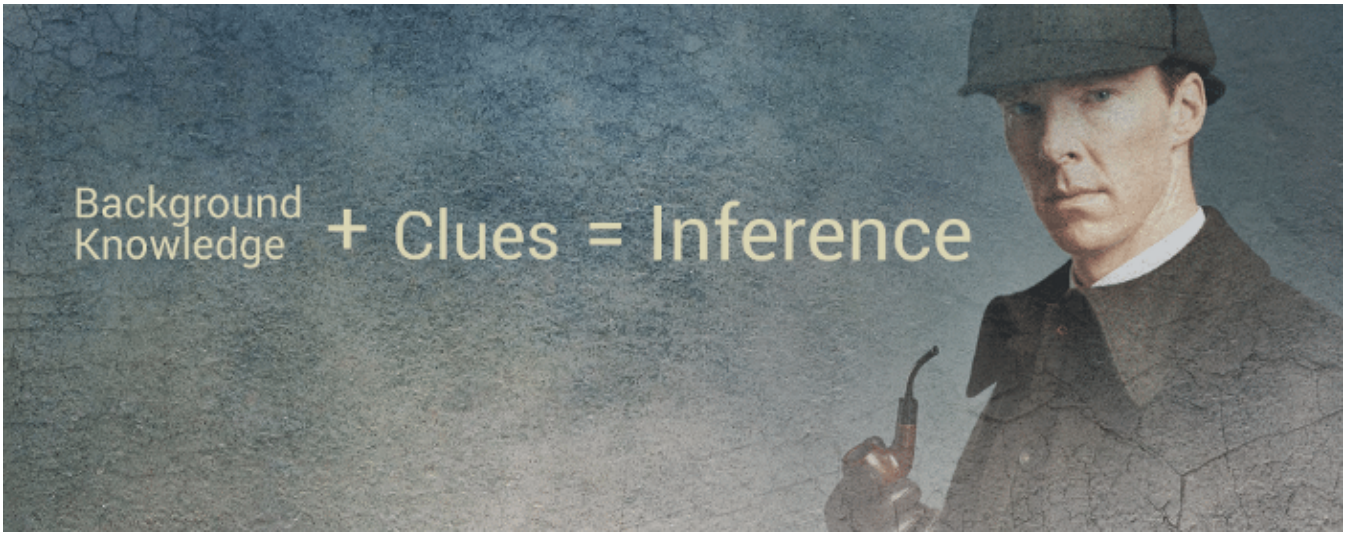


image credit: https://www.ontotext.com/knowledgehub/fundamentals/what-is-inference/

Why you should read this post

Deploying machine learning models to production in order to perform ***inference***, i.e. predict results on new data points, has proved to be a confusing and risky area of engineering.

Many projects fail to make the transition from the lab to production, partially because these difficulties are not addressed on time.

In my opinion, there are three major factors which make deployment challenging:

1. **Starting the process late** — Partially due to cultural gaps, and partially due to the nature of the research cycle, many teams leave the subject of deployment till very

late in the game, resulting in an array of nasty surprises and challenges. For example discovering that the data use for training is not available in production.

2. **Lack of mature architectural patterns** — as an emerging technology, there are relatively few companies outside Tech Giants who have perfected the art of Model deployment and serving to learn from.

3. **A confusing plethora of competing platforms and technologies** —
   Each built with a subtle change of paradigm, and optimising / emphasising different parts of the solution.
   Aa side note — *Even nomenclature is not fully established* —
   so forgive me for adopting my own for some of the concepts mentioned below.

There are many technical decisions when deploying Machine Learning to production — packaging, versioning, server stack selection, hardware profile, methodology for performance tuning, capacity planning, monitoring needs etc.

This post focuses on introducing the architectural patterns and considerations for decomposing inference tasks into services.

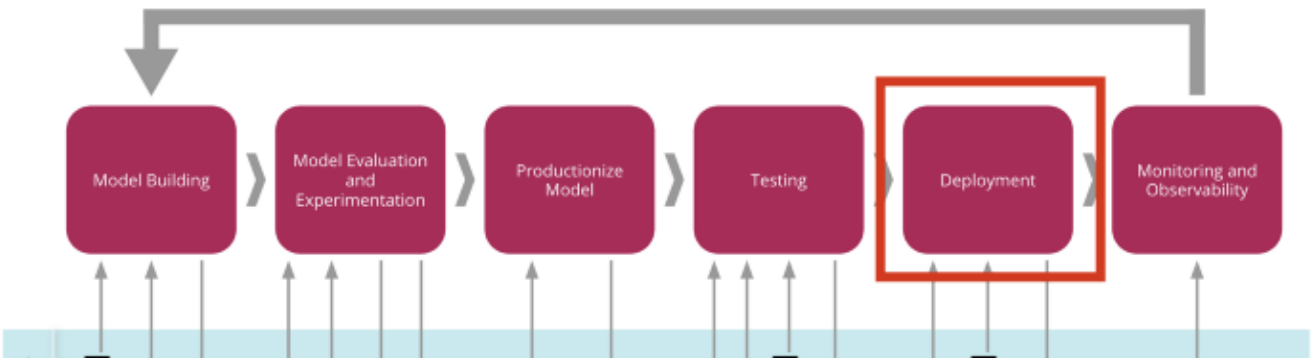
The post assumes you have some experience in deploying services to production, and focus mostly on the decisions/characteristics which are unique to Model Inference.

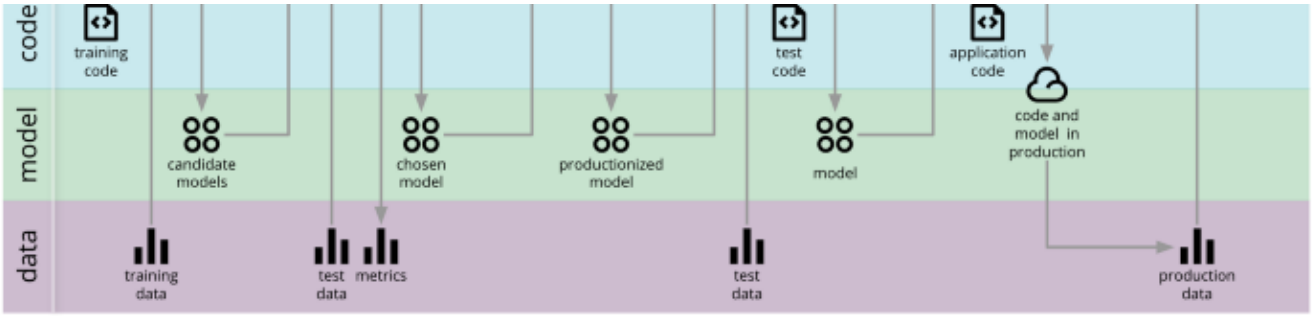
Note: deploying inference services onto a batch or streaming environment is also a very valid use-case, but is not covered here.

Inference in the context of ML Dev. Lifecycle

Deploying the model in order to perform inference means that you've trained a model, tested its performance , and decided to use it to make predictions on new data points.

If you'd like to learn more about the model dev. lifecycle, this is a good place to start.





Original image: https://martinfowler.com/articles/cd4ml.html

Inference in the context of a larger system

Model predictions are typically used to achieve some business goal, and hence model inference services need to be integrated into a larger system.

Inference pipeline boundaries

Business tasks are concerned with predicting “things” about entities in the business domain.

However, models don’t usually consume business objects to predict on.Instead, models consume some abstract *representation* of a data entity.

For example a cropped, rescaled and grayscaled image in as a 256X256X3 tensor , or a vector with numerical features as a numpy array or a dataframe.

Here is what you need to know about these representations:

1. Models are very tightly coupled to their data representation. Any small change in the format or even data distribution of one of the dimensions of the data can cause model drift
2. Much of the model’s “brain” comes from evolving this representation of the input; data scientists need to maintain the ability to iterate over this representation without requiring upstream API changes
3. Creating this representation is fundamentally different from calling a forward() API on a neural network — it involves imperative code, transformations etc. In some cases it may include IO operations to enrich or load additional data, and “data science logic” to prepare.

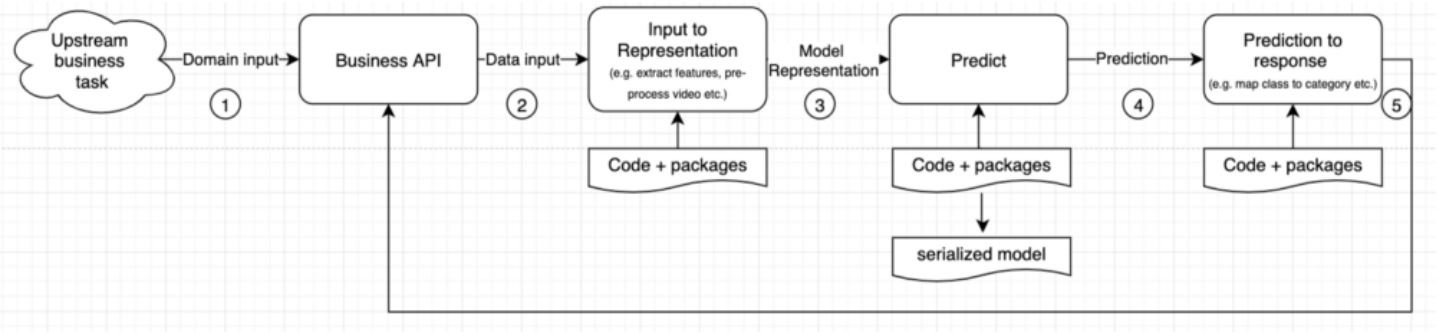
It follows that :

1. **The entry point to the Inference pipeline needs to be high level enough — either a Business domain object (e.g. a transaction) or a "data domain" object (e.g. an IP address).**

2. **The task of creating the representation is inside the scope of the Inference pipeline**

3. **The creation of the representation should be treated separately from the prediction.**

Logical inference pipeline



1. ***Business-API* is invoked to fetch a prediction.**
   The API is centred around the ***Business domain* —** entities and concepts which are relevant to the business task.
   As an example, imagine an online payments system which is processing a transaction. As part of the flow, it may need to score the transaction for suspicion of fraud. The Business API will receive a transaction object, and return a score or an enum representing the likelihood of it being fraudulent.

2. **Optional — *Business API* may need to translate the *Business-domain input* into the *Data domain*.**
   Continuing with the fraud detection example, The fraud detection model may not be interested in transactions at all, but only in the sender's IP addresses. This adaptation from the Business domain to the *"Data domain"* may be "empty" in some cases, and in others it may involve various operations like projection, enrichment, etc.

3. ***Input to Representation* —** takes a "***Data domain***" input, and transforms it into a "***Model Representation*** " of the data.

   In the case of classical ML, this transformation involves

computing features.
Here, the output **Model domain** would be a dataframe with numerical features, ready for prediction. In the case of computer vision model, it may mean **pre-processing the image**, cropping it or transposing it, and packaging it in a numpy array or a tensor.

4. ***Predict API*** is invoked on the ***Model Representation***.
At the very least, this involves taking a serialised model file, reading it (usually ahead of time), and using it to perform the computation over the input vector.
In some cases, the model's work is more complex than that, and requires imperative code.

5. ***Prediction to Response (post processing) —*** takes the ***prediction*** and translates, if needed, back into the ***Business domain***. For example a score from the fraud model needs to be transformed into one of the following "not fraud/suspicious/highly suspicious/uncertain" labels, which can be interpreted by the client.

Any of these tasks can reject their input, or fail to run properly.

For example, an image classification inference pipeline may reject images which aren't in a sufficient resolution or are corrupt in some way.

These errors and failures need to be propagated back into the 'Business domain' too, typically with a human-readable explanation.

Inference pipeline — decomposing into services

So, how do you decompose this pipeline into services? what is each service responsibility and API? what are the main considerations?

The good news is, most service architecture best practices applies to inference too. But still, here are some of the considerations which are specific to inference:

1. **Data science vs. engineering skills** — many companies have a pure Data Science team, with little or no experience in running services in production. But will want to make the team accountable to releasing their models to production, without "handing over" to anyone.

2. **The intimate relationship between Representations and Models** —
Even the slightest change in distribution of a feature may cause models to drift.
For complex enough models, creating this representation may mean numerous

data pipelines, databases, and even upstream models. Handling this relationship is not trivial to say the least

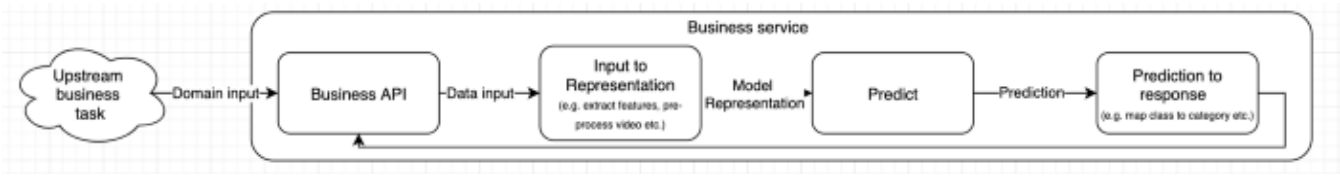
3. **Unique scale/performance characteristics** — as a general rule, the **predict()** part of the pipeline is purely compute-bound, something which is rather unique in a service environment.
   In many cases the **representation** part of the workflow is more IO bound (esp. when you need to enrich the input, by loading data / features, or retrieve the image/video you're trying to predict on).

I believe that the overwhelming factor which drives many of the emerging design patterns in inference service design, is the org/skill set one — i.e. the desire to make Data Science teams accountable to release models to production, but without requiring them to own a fully blown, standard production service.

Let's take a look at some common design patterns, when they may be useful, and what are their challenges.

Embedded inference pattern

In this pattern, you package the ML model + code inside your business service.



When to use?

1. Model is consumed by a single business service
2. The inference pipeline (including transformation and predict) is not too complex/expensive to compute
3. The Data science folks working on the model are very aligned to the business domain (perhaps even embedded in the service team).

All you need to do is package the model artefacts together with the service code, and deploy.

Challenges

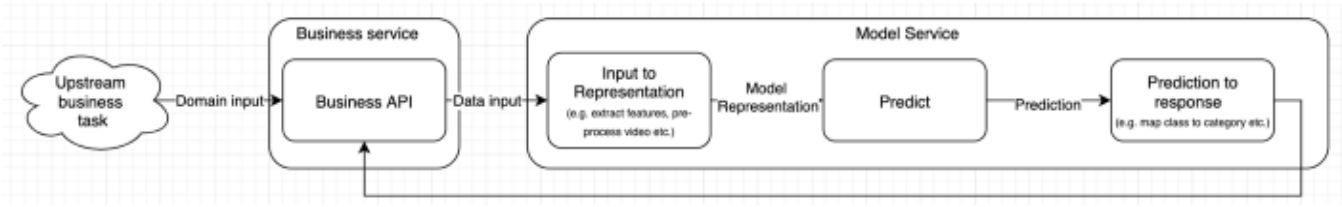
- De-facto the DS team is part of the business service team, and needs to integrate intimately.
- The business service is now exposed to the model's unique IO/Scaling requirements.
- The CI/CD pipeline now needs to include ML-driven tests, which are very different from business logic tests…

This is a relatively uncommon pattern in my experience.

Single service Inference pattern

Here, the model is deployed in a dedicated service, which (different) business services calls into with a "data input" API. The service encapsulates feature computation, prediction, and output transformations.

This is a common sense option. Default to using it.



When to use?

- Need model re-use
- Transformation logic is not too complex/costly

Challenges

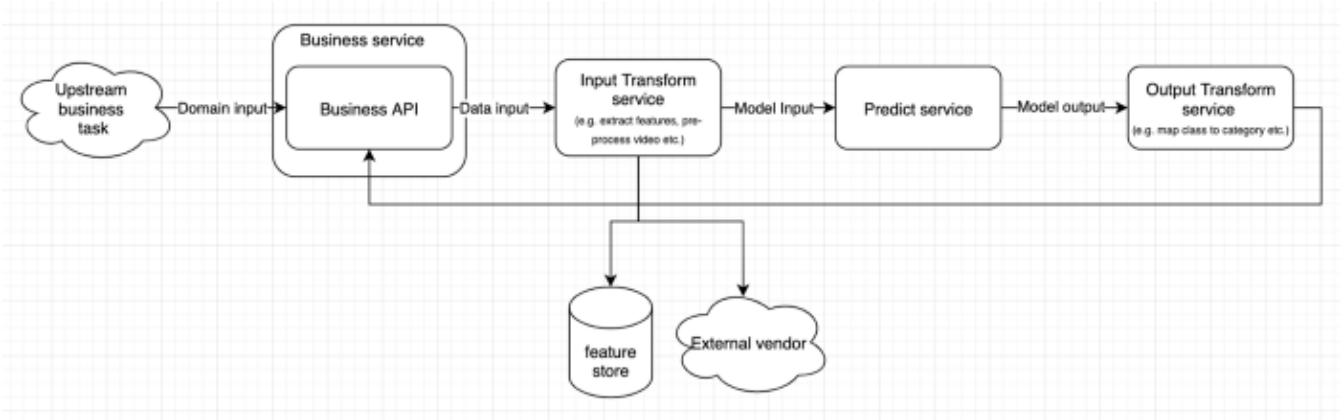
- We're still talking about a fully fledged service, someone needs to own it.
- Hard to draw the line on when the coding/scaling tasks involved have become significant enough to require an engineering team

See the section below "generic inference service pattern" which discusses some of the approaches used to overcome this challenge.

Microservices inference pattern

In this approach, the inference pipeline is further broken into sub-services, which are combined together to achieve the result.

Any microservices pattern is valid here: a central orchestrator, a pipeline, synchronous or a-synchronous.



Typically, you use this pattern when the inference pipeline is more complex.
Also, when some of the computation (typically the input transform) is expensive and can be reused between multiple models.

Examples

A good example of a systematic separation of the pipeline is the use of **Feature Stores.**

With this approach, feature computation is done by separate services or data pipelines. The results are stored in a “Feature store”.

The Model uses a thin API layer to retrieve the feature values from the Feature store (e.g. by feature name) and predict.
This pattern not only decouples feature computation from prediction, it also formalizes an API between these components, in a way which promotes reuse.

In the Deep Learning context, decomposing the input transformation to its own service is sometimes done to save compute resources.

For example, a large scale social network probably has many different Deep learning models which analyse user uploaded videos.

Preprocessing video is an expensive effort — both in engineering resources terms as well as in operational costs. The company may choose to build a video processing pipeline which will prepare data for prediction for many models. The output of the

pipeline may be a representation of video segments with a common encoding, FPS rate etc. which all models are trained on (and infer on).

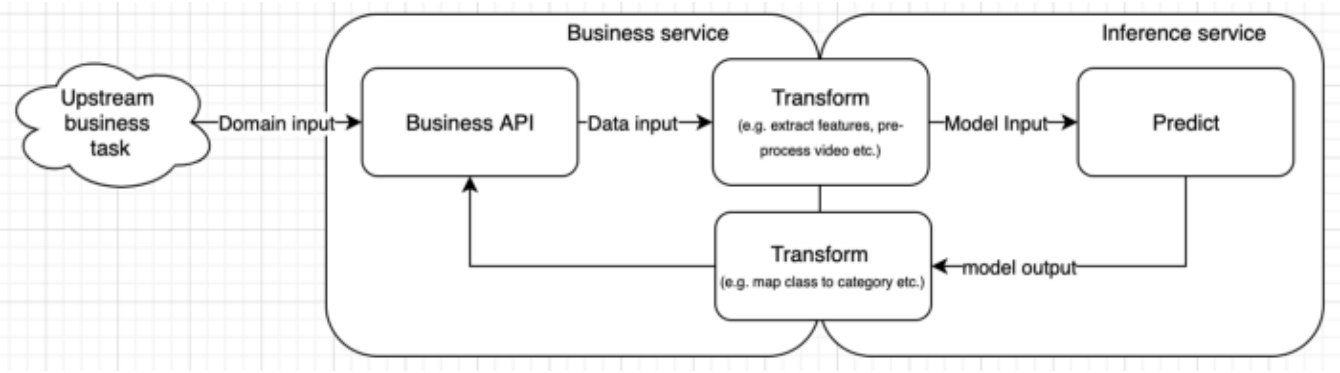
When to use

1. When you want to promote reuse of lower level elements like feature computation
2. When you need to scale by making different teams own different services (e.g. data engineers in charge of the feature calculation, DS team in charge of the model microservice).

Challenges

1. Features and Models are tightly related, **so decoupling them is hard to achieve**. For example, the only really effective and definitive testing methodology for ensuring models and features work well together requires testing them together (integration test in our case).
2. As a result, **efficiency gains from this reuse is achieved mostly when new models consume features/pre-processing code which is already deployed/mature**.
3. The converse is also true: **releasing a model which relies on new features is more difficult with multiple services.**

Getting over the engineering hump —by deploying “generic inference services”



Whichever way you look at it, some artefact of the DS team needs to end up in a service in production, and that service needs to have a CI/CD pipeline, an API, logging, monitoring etc.

The “embedded model” pattern solves this by de-facto embedding the DS dev. lifecycle inside the artefact of an engineering team; as we said, this has limitations.

So, how do you get DS to deploy to production without needing to manage their own service or have much engineering skills?
Enter the “generic inference” pattern.

Some companies develop this in-house: By creating a contract between the engineering team which owns the service in production, and the data science team which pushes code into parts of it.

Recently, this pattern appeared also in SaaS/Open source solution.

Generic inference workflow

- **Data scientists deploy a “packaged model”** (essentially a model weights file and a few minimal python files/metadata) and are picked up by the generic service. The shape of the package can be a jupyter notebook, or a .tar/.zip file with artefacts.
- **The upstream client services invoke the generic API**
- **The Generic API delegates to the “packaged model”**, collects the response, and returns it to the client service.

Here is an example of the client code needed to invoke a CV model deployed using the **TensorFlow Serving** generic framework**:**

```
Data: {"signature_name": "serving_default", "instances": ...  [0.0],
[0.0], [0.0], [0.0], [0.0], [0.0], [0.0]]]]}
headers = {"content-type": "application/json"}
json_response =
requests.post('http://localhost:8501/v1/models/fashion_model:predict
', data=data, headers=headers)
predictions = json.loads(json_response.text)['predictions']
```

Here is an example for **how to package a PyTorch model artefact for deployment in AWS SageMaker**

```
model.tar.gz/
|- model.pth
|- code/
```

```
|- code/
  |- inference.py
  |- requirements.txt  # only for versions 1.3.1 and higher
```

Benefits

1. No need for Data Scientists to code and own a production service
2. The inference API was developed by expert engineers in Google, Databricks or Amazon
3. Many of these tools offer cool features like automatic model reloading (which makes deployment of a new model a breeze), blue/green solution for how to slowly scale up a model, and more…

Challenge 1 — handling complex inference pipelines

What happens when the inference pipeline needs to include imperative code for creating the model representation, or aggregating the model results?

"Generic inference" frameworks address this, by allow you to provide **custom code and even external dependencies along with your model weights file**.

Your code implements a few hooks for the generic inference service to call:

1. **Init()** method — upon service loading, this hook will allow you to load your model weights file and do any heavy initialization
2. During service invocation, the framework will call your code in the following sequence: **pre_process(), predict(), post_process()**

The complexity barrier

If your inference pipeline is complex — with a lot of code for creating representations, which may perform IO etc. — this is going to be a challenge:

1. It's hard to cram complex pipelines into the pre_process(), predict(), post_process() API.
2. You still need an engineering capability to author, test and streamlines this pipeline.

3. The service you are hosting your code in was not tuned for your workload, and will likely be hard to tune it's performance to your specific use-case

Challenge 2 — Leaky and opaque interfaces

Another downside is that typically, a generic interface is harder for consumers to understand or test. Especially when the signatures involve numpy arrays or tensors serialized in some unknown way.

How do you populate the tensor from a .bmp file? do you need to use the image bytes? perhaps load it as a PIL image? do you expect any specific encoding? etc.

It's virtually impossible for the client to validate they've loaded the data the right way without invoking the inference service, and even then, determining if mistakes are due to wrong input or model mistakes is hard to do

When to use Generic Inference in real life

I think that the main benefit of this pattern is if

1. You absolutely insist on DS deploying their models independently.
2. You plan to have a lot more model releases than feature/representation change releases.

If one of the above applies, consider using it as follows:

1. **As "the" service in the "Single service inference pattern" —**
   but only if your processing is very light, and you're extremely short on engineers
2. **As the *Predict* microservice in "Microservices inference pattern" —**
   Make sure that this service is not exposed to the "end client" — but to another inference microservice

Generic Inference on steroids — Multi-model generic Inference

This design pattern is, I believe, somewhat unique to Inference services.

Under this regime, you deploy a **generic inference service which hosts *multiple* models — either different versions of the same model, or models which serve unrelated domains.**

The Multi-Model server "detects" new models and their packages automatically. It then exposes a unique URL for each.

This is counter intuitive at first as it goes directly against the principle of domain decomposition— by lumping together models which serve different domains/tasks in a service which is now a single point of failure to multiple domains.

It also introduces a challenge of how to do load balancing and capacity planning when the workload is potentially very heterogenous — e.g. if each model has different SLA's, volumes, and compute/IO requirements.

Multi model serving is not an anecdote— the biggest platforms and toolsets out there support it out of the box — such as TensorRT (by NVIDIA), TensorFlow Serving (by Google) and Amazon's Sagemaker.

So, what is the rationale behind this design pattern?
if I had to guess, I'd say:

1. Conway's law — continuing along the line of separating engineering from DS, you may want to set up one central engineering team who will manage 'inference' across all data science use-cases.
2. Compute utilization — Attaching GPU's to servers is expensive, and letting them sit idle is a big waste. By smartly co-locating different models onto the same server, you may be able to increase GPU utilization.

The subject of inference efficiency, especially on GPU, deserves its own post — for example how to deal with the mismatch between GPU's appetite for batching, vs. business tasks which are usually transactional…

Summary

Deploying inference services is still a relatively new discipline, with its own unique set of challenges.

A few architectural patterns have started surfacing to address these challenges, and it seems that many of them are attempting to abstract the mechanics of production from the Data Science team's practices.

With enough compromises and a lot of new dev. lifecycle practices, this can be a possibility.

However, in my opinion, if your main issue is the lack of engineering skills in your data science team, an easier and generally good approach is to mix engineers and Data scientists together in teams. The benefits far exceed the ease of deploying services to production…

Mlops Machine Learning Microservices DevOps