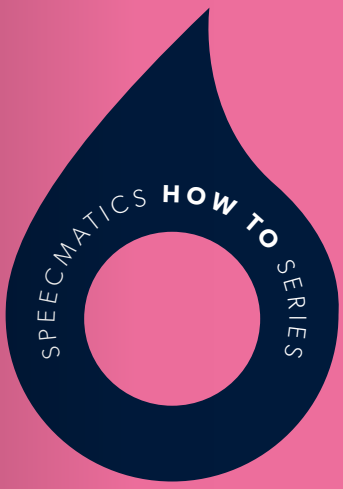
SPEECHMATICS



HOW TO

deploy machine learning in production

WWW.SPEECHMATICS.COM

Part 1

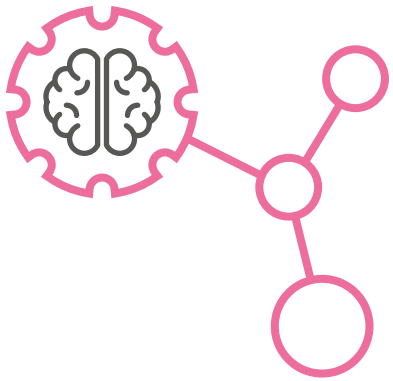
How can you help your business deploy ML into production?

Hints and tips on how you can help your business deploy machine learning into production so that you and your customers can consume it. This eBook is based on my own experiences of using machine learning and applying it to speech recognition at Speechmatics.

With the increased commercialised application of machine learning (ML), the ability to deliver and deploy models effectively is becoming ever more important in a business environment. ML is certainly trending at the moment, everyone seems to be doing it, but only a subset of ML systems are useful to deliver value to the end-user, and only a subset of *those* systems get deployed cleanly and effectively. This eBook will give some high-level pointers on what "clean" and "effective" might mean and show the impact making the right choices can have from a business perspective.

By way of introduction, it is important to realise that ML is a discipline with a heavy focus on making predictions on unseen data.

Due to the growing volume and variety of available data – be it internal business data or large externally sourced datasets – it is becoming increasingly important to build and deploy models that can usefully leverage this data. ML can exploit these giant datasets by building models that learn to predict a quantity of interest. For example, in automatic speech recognition, models are essential for two concrete tasks: predicting phonemes from audio frames and predicting the probability of the next word in a sentence. Other examples abound. If there is a large dataset, someone, somewhere in the world will be building a model on it.



But now that you have that model, how do you and your customers consume it? How can we best iterate quickly and deploy that model into production? What makes a deployment effective for the end-user?

Before beginning, in true ML fashion we can formulate the answer by defining an objective function that we must optimise: jointly maximise commercial value and developer velocity. These two concepts describe what you might refer to as an "effective" and "clean" deployment. "Effective" because users are getting hold of something truly useful to them. "Clean" because developers can iterate rapidly without worrying about infrastructure issues.

A WORD OF WARNING

This article is an opinionated high-level piece focussing on this final stage of the ML pipeline. It's built on hard-won experience of what works and what doesn't in automatic speech recognition pipelines but we all have different goals so your mileage may vary.

Effective ML deployments

Imagine as the CTO of DogFace your customers are desperate for you to deploy a model that can correctly classify the breed of dog in their images. Your engineering team tells you this task is an absolute synch and two days later you have a model in your hands. What are the next steps?

1

Ensure there is a correct alignment between what users find useful and what the modelling team has been optimising.

Do the users want uncertainty estimates in their output? Do they just care about the most probable class or do they want the top four breeds predicted? There must be a crisp understanding of what the user **actually wants** and that must be quickly and continuously fed back to engineering.

EXAMPLE

In speech recognition, we assume the end-user cares about word error rate (WER) (a number which correlates the accuracy of a transcript) but we must ask: is this entirely justified? Perhaps we should care more about avoiding a one-word major gaffe, than five-words of incorrect transcription. It's easy to optimise a number but stepping back here with a pause for thought will pay dividends.

2

Determine the correct operating point for your model.

As with most problems in Computer Science we can always trade-off memory for compute. In ML we often have the related trade-off of latency vs accuracy. Find the perfect trade-off between these variables for your business use case, making the most cost-effective use of available hardware. Measure it, measure it again and iteratively refine it until you **hit the sweet spot.**

EXAMPLE

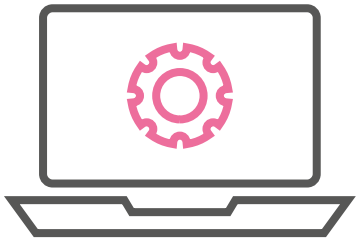
Consider deploying a machine translation (MT) system. First ask: does the system need to be streaming or can we batch at the sentence-level? If we can batch, then we can immediately gain some computational efficiency through better parallelisation. (Here, the problem definition itself has given us access to an important trade-off.) Next, we train a series of candidate models – some smaller, faster and less accurate; some larger, slower but more accurate – and plot them against each other forming a speed/accuracy curve. The task is to then decide which point gives the best trade-offs on that graph. Large models often give diminishing accuracy returns and smaller models will give diminishing latency gains. The MT system that you end up deploying should be done with the speed/accuracy trade-off evaluated, discussed and settled.

3

Wrap it up with a user-appropriate interface.

You can have the best and most accurate model in the world but if it can't be easily integrated into workflows, or consumed by an API, libraries or client-SDKs, then it's likely that some commercial value is going to be left on the table. So, when designing the interface:

- Don't expose or force the end-user to understand the internal operation in any way. This will allow the flexibility to swap out entire sections of the internal ML pipeline without affecting the contract with the user.
- Define the schema or interface with extensibility in mind. The core of the contract will be most likely very stable with "features" added over time – design with forwards compatibility in mind.
- Be upfront that the quality of results will always improve with respect to some metric but that the specific results may vary over time. In other words, the API itself must be stable but the content (i.e. specific predictions or results) must be free to change over time so that models can be improved and changed frequently.
- Consider using a framework that allows batching. Frameworks such as TensorFlow Serving will internally batch requests so that throughput can be maximised.

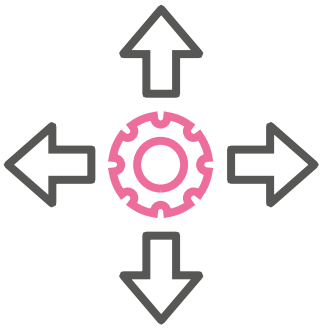


Clean ML deployments

Your customers now have something in their hands that does what it says on the tin, exactly matches their use case, and has been easily integrated into their workflow. In reality, that is only the beginning of the lifecycle.

Just like any other software artifact, improvements will be made to the model and its runtime profile may well change over time. Our task now is to make this process as seamless as possible with engineers deploying new models into production on a fast cadence. Or in some use cases, just ensuring the model continues to produce good solid results.

So, what are the best practices?



1. Decouple concerns

Production environments can be complex and require a lot of engineering time and effort. Problems that arise may be interrelated such as scalability, reliability and performance. Target them separately. A concrete example: model code should know very little, ideally nothing, about how the service will be scaled out. All the model code should know about is its SLAs and performance profile. Scaling can be left to something such as a container orchestrator's autoscaler.

2. Prioritise metrics and validate against them before and after deploying

Use an automated system to validate a candidate model before scheduling it for deployment. Some metrics might only need to have loose thresholding where we tolerate some degradation, such as job turnaround time. Others might be stricter such as performance against some validation set or internal SLAs.

Measuring the performance of your model in production against a metric that you are confident matters to your customer (e.g. word error rate, accuracy, precision, BLEU score etc.) is **the way** to be able to move forward with confidence. Even better is to have a completely unseen test set drawn from production data itself. One can improve against an academic benchmark but that may well be a largely irrelevant metric unless your customers are pushing academic datasets through your system!

Finally, all the metrics should ideally be boiled down to one single number that everyone agrees on – this helps cohesion between teams who might have orthogonal concerns. Releases can be gated on this number.

3. Keep runtime dependencies isolated

Minimise the difference between development and production environments. In particular, containers can help to minimise the differences between environments. This is as true for ML setups as any other piece of software. Subtle differences can easily creep in such as numerical precision differences between CPU and GPU execution, CPUs with different Streaming SIMD Extension support, different library versions and so on. Compiling all the dependencies down into one single binary (if possible) and running the whole pipeline or model in a container atop a minimal and stable OS can minimise these issues.

4. Couple the model from the inference code

This might be the most important item learned at the coalface. Never decouple the model itself from the code that performs inference on it. Even at the expense of violating the DRY principle, it is imperative that the model can be queried without having to know which inference code to use. As an external user, all I care about is running queries and getting results: I should never need to know details about the internal pipeline or what versions of inference code work with which model. This might come at the cost of larger artifacts – take that hit gladly in exchange for guaranteed correctness.

5. Continuous deployment with rollbacks and canaries

To deploy continuously into production, we need a very high level of confidence that we have all our ducks in a row. ML deployments are really no different to other deployments in this regard. All conventional wisdom applies:

- Ensure the build is fully reproducible and deterministic.
- Add functional tests around the model that capture its desired behaviour as extensively as possible. This is especially relevant for structured-output tasks where we can add many sanity checks (e.g. for a translation task, check that the output length in the target translation is non-zero for difficult inputs such as very rare foreign names etc.).
- Add performance tests to check that the operating point in terms of memory consumed and compute consumed is still in an acceptable range.
- Rollout new models gradually with production traffic being switched over from old to new.
- Soak test models for memory leaks and performance regressions; especially relevant for always-up streaming models.
- Add button-push rollbacks. They not only mitigate the effects of bad deployments but adds another tool to the arsenal for de-risking continuous deployment.

Some use cases mean this level of automation is unsuitable, but it is likely that for most ML use cases this is indeed possible and can give your business an edge in being able to be nimble.

6. Consider microservices

Microservices are an excellent candidate for ML-based deployments. Querying a model with an input is often a stateless action. This means most deployments can straightforwardly use container technologies such as Docker to hermetically seal the model and wrap it in a very lightweight way. Horizontal autoscaling of the service should then just "work out of the box".

7. Run the model "hot"

Precious time can be lost loading a model into memory and initialising data-structures. If you are running inference over an audio file of many hours maybe this isn't the top priority, but if you are classifying images then running "hot" is critical for latency.

8. Bonus item: Cheat and cache results!

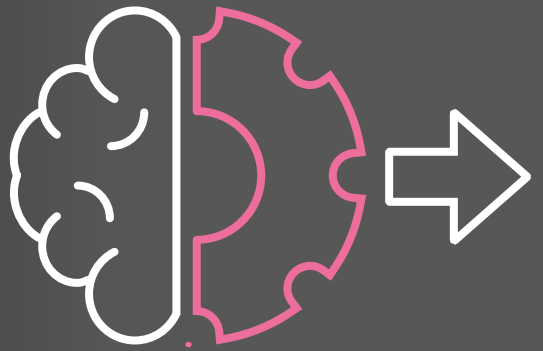
A very simple trade-off of compute for memory: hash the input and then lookup the result in a hash table. It's almost too obvious to mention but it's a strategy which could save serious amounts of compute in some domains. If we can obtain a unique hash of the input with low overhead, results are cheap to store, and we already have a result on the currently deployed model, then we can happily cheat and return immediately with the result.

REMEMBER

There is no generic deployment scheme that fits every problem and every company, so you must find one that works for you and your specific needs. Now, with those abstract principles in place, we can look at some more specific possibilities for an ML pipeline that attempts to pull them together and put them into practice.

Part 2

Putting it all together – deploying in practice



We've previously seen some high-level principles that we might want to bring to bear when deploying an ML model or pipeline in production. This part takes those principles one step further and suggests how they can be instantiated in the real world.

Many tools now exist such as TFX and even SaaS companies like Algorithmia which aim to automate as much of this as possible. It's great to be able to leverage the efforts of other projects but anything but the simplest of ML deployments will invariably require many custom pieces; in other words, we can't always palm everything off to a 3rd party tool.

There's always...

1. Build the API

(Principle: Decouple concerns, Cheat and cache results!)

The first interaction users will have with our ML deployment is much more prosaic than the ML pipeline that we have created. The API definition and schemas are the first impressions the users get; if we make this obscure, expose too many settings or make it difficult to parse or use, then we fall at the first hurdle.

The first step is to recognise that a well thought through and stable API is a core part of the product and it requires as much care as the modelling itself. To that end, something like OpenAPI specs will help us out. We can declaratively define and version the output, autogenerate documentation and client-SDKs, and validate our pipeline output against the current schema (see opposite).

By choosing a canonical and widely supported schema, we can easily generate client-SDKs for a wide range of languages, enabling simple integration. Before we deploy the model, we may require API changes to support new functionality – ideally this can occur in lock-step with the release of the new model. This API piece has high coherence with the model itself, so I recommend versioning them together if possible.

An important consideration at this point is whether to make our API synchronous or asynchronous, batch or streaming. If our latency requirement is very low for one-off jobs such as image classification, then we should be targeting a synchronous API. For longer inputs, we might prefer asynchronous, or perhaps we support both to provide maximal flexibility for our end-users. If our use case is a natural fit for streaming input or output then we might consider using WebSockets or some kind of streaming model with gRPC to define a streaming API that can return partial results or results at regular intervals.

For example: speech recognition APIs could conceivably fit any of the possibilities above. The key is to understand the needs and use cases of customers ahead of time so the API can be as useful as possible.

We may additionally want to integrate a component here which hashes the input and performs a lookup in some key-value store (such as Redis, the database flavour of the month, or even in-memory caches) to see if the request has been made before. The key can be a hash of the whole input and the value of a reference to some object in a file store or the result itself. This could be decoupled from the API itself, depending on how elaborate the caching mechanism becomes.

Here's an example snippet from Speechmatics' schema:

```
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "Speechmatics Transcriber Results",
  "description": "The Speechmatics ASR transcription detailed
result format.",

  "type": "object",
  "properties": {
    "format": {
      "type": "string",
      "description": "Speechmatics JSON transcript format
version number."
    },
    "job": {
      "$ref": "#/definitions/JobInfo"
    },
    "metadata": {
      "$ref": "#/definitions/RecognitionMetadata"
    },
    "results": {
      "type": "array",
      "items": {
        "$ref": "#/definitions/RecognitionResult"
      }
    }
  },
  "required": [
    "format",
    "metadata",
    "results"
  ],
…
```

2. Measure and validate the candidate model

(Principles: Prioritise metrics and validate against them before and after deploying, Continuous deployment with rollbacks and canaries, Couple the model with the inference code)

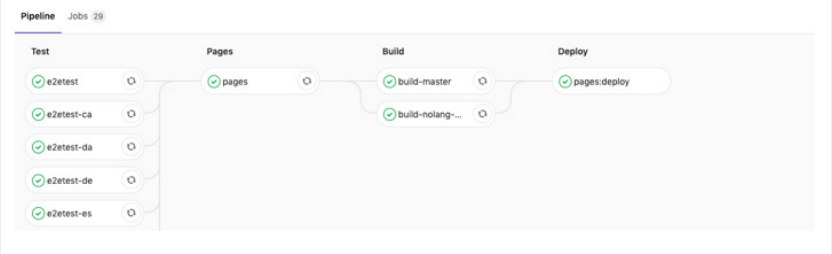
We now have a model in our hands which is a candidate for release and a well-defined API. We now make a commit in version control which captures the state of the inference code that will be used along with this model. That revision can then be used to trigger a pipeline of tests:

- Unit tests and smoke tests that capture the most basic functionality. E.g. does the model return a non-negative result, do we get the same result if we run the same small input through multiple times.
- Functional tests that capture desired behaviour of the model – we can focus on edge cases and awkward inputs, very large inputs, check all metrics are within tolerance levels. E.g. does the BLEU score or F1 score degrade by more than X% relative compared to the previous release?
- End-to-end tests that ensure API contracts are met; a battery of common inputs can go all the way through the pipeline to produce sensible results. E.g. this can amount to accuracy on very easy data, acceptable performance on files that have previously caused problems.

If the test coverage is sufficient to produce a high-level amount of confidence in this candidate model, then we can proceed to build (see below), and then automatically roll out some kind of canary release where only a fraction of traffic arrives at our new model. Container orchestrators such as Kubernetes have good support for defining workflows here which match many use cases and risk tolerances at deploy-time.

References:
https://docs.gitlab.com/ee/topics/autodevops/
https://www.kubeflow.org/

Example pipeline in GitlabCI



3. Build a production container

(Principles: Couple the model with the inference code, Run the model "hot", Consider microservices, Keep runtime dependencies isolated)

We need deployed artifacts to be immutable and as isolated from system dependencies as possible. Containers are a popular unit of delivery to use both inside and outside the ML world. I won't delve too deep into the pros and cons of containers, but I will say this: they feel like the appropriate level of abstraction for ML pipelines. The user has a set of inputs and all we need is a stateless black box which returns some form of prediction on the input.

Whether we want to run the model as a service behind an API (such as Flask) or we are happy to have ephemeral one-off batch jobs, containers can get the job done for us.

One of the most commonly used containers is Docker which is an open platform for people to build, ship and run distributed applications across a variety of platforms. Making ML applications self-contained means that they can operate in a highly distributed environment and we can place those containers close to the data that the applications are analysing.

So how should we organise and build the container? The two key principles are to shrink wrap as tightly as possible and ensure that the test-time environment is **as close as possible** to the train-time environment of the model.

Shrink wrap as tightly as possible

From an ML perspective, there are a number of things we can do. Model compression techniques such as quantisation, distillation and pruning can take a large model and produce a smaller model which maintains most of the performance of the larger model. With neural networks, it is often the case that upwards of 80% of the parameters are largely redundant. If we want to save memory, we can prune and store sparse matrices. In addition, distillation can save us both on runtime cost and memory overhead by training a smaller model to mimic the behaviour (or rather, the precise output distribution) of the larger model.

For applications where the parameter count is dominated by some lookup table (e.g. word embeddings), we can apply vector quantisation or other quantisation methods to the embeddings, drastically reducing the disk footprint of the model.

Some frameworks make this a synch. For example, an export script in TensorFlow only needs a one-liner to enable quantisation:

```
import tensorflow as tf

...
# quantization
transforms = ["quantize_weights"]
output_graph_def = tf.tools.graph_transforms.TransformGraph(
    output_graph_def, input_names, output_names, transforms
)

with tf.gfile.GFile(frozen_graph, 'wb') as fh:
    fh.write(output_graph_def.SerializeToString())
```

If there are any stochastic operations inside the model, we might now want to factor that stochastic operation out and set it as an explicit input to the model – making the model itself deterministic. Deterministic operation is critical for reproducible and solid testing. Any other steps to ensure deterministic operation should be made at this point.

Half-precision inference can also be beneficial. If the runtime architecture supports half precision operations natively then we obtain benefits for both storage and execution speed. Even without native support for half precision, it can still be used as a convenient storage format and be expanded to floats at runtime.

Finally, I recommend packing all runtime configuration and settings directly within the model. Any hyperparameters or input files which don't change at runtime (such as a vocab list) can be packaged or pre-set directly in the model. DevOps engineers should not need to understand or know about what settings are used in the model – these should be hidden and predetermined ahead of time.

At this point, we should have a binary blob of model parameters with any non-changing hyperparameters or inputs hardcoded inside.

Ensure the test-time environment is the same as train-time

With the model in good shape, we can now look to add the remaining items to the Docker image. Our inference code should now be producing the same (or very similar) results to the full-fat model. Ideally, we can now compile that code down along with the API to a single binary which is run as the container's 'entrypoint'. A key consideration here is to ensure the inference code is as close as possible to how evaluation was performed at train-time. Differences should be minimised.

```
##########  ARGS for image names ##########

ARG BASE_IMAGE=dockerhub.artifacts.dogface.io/ubuntu:xenial-20180417
ARG MODEL_IMAGE=docker-data.artifacts.dogface.io/dogbreed-
classifier:b4123a

########## EXTERNAL IMAGES ##########

FROM $MODEL_IMAGE AS modelimage

########## BASE ##########

FROM $BASE_IMAGE AS base

ENV MODEL_DIR=/model \
    LANG=C.UTF-8

RUN apt-get update \
  && apt-get install -y --no-install-recommends
  <your runtime depdencies>

COPY dogbreed dogbreed

ENTRYPOINT ["<your server or tool to process input>"]
```

Continued on page 20

```
########## PRODUCTION ##########
# note: we put the language pack as the *last* layer so layers in the
base are shared on same host

FROM base as production

ARG BASE_IMAGE
ARG CI_COMMIT_REF_NAME
ARG CI_COMMIT_SHA
ARG CI_PIPELINE_ID
ARG CI_PROJECT_NAME
ARG MODEL_IMAGE

LABEL maintainer="support@dogface.com" \
      com.dogface.git_sha="${CI_COMMIT_SHA}" \
      com.dogface.image.base="${BASE_IMAGE}" \
      com.dogface.image.model_version="${MODEL_IMAGE}" \
      com.dogface.pipeline="${CI_PIPELINE_ID}" \
      com.dogface.service="${CI_PROJECT_NAME}" \
      com.dogface.version="${CI_COMMIT_REF_NAME}"

RUN /var/lib/apt/lists/* /etc/apt/sources.list.d/* \
&& apt-get purge -y <build-time deps>

COPY --from=modelimage /model /model

########## DEV ##########
# note: we put the model *above* code/depdencies when running tests
# as it will not change as frequently

FROM base as dev

COPY --from=modelimage /model /model

RUN <install dev depdencies>
ARG CACHEBUST
RUN echo $CACHEBUST

COPY dogbreed dogbreed
COPY unittests unittests
COPY functests functests

FROM dev AS lint
RUN pyflakes dogbreed unittests functests
    pycodestyle dogbreed unittests functests
    pylint -j2 dogbreed unittests functests

FROM dev AS test
RUN pytest -v unittests functests
```


Example Dockerfile

I argue that inference code and models should live together. If so, then they should also be **versioned** together. This limits flexibility (perhaps we want to separate and drop in different models dynamically) but gives greater confidence in correctness. By packing both the inference code and the model inside a Docker container together, we can create a robust artifact which can be used, re-used, packaged up further and deployed in multiple forms and for numerous use cases (see below).

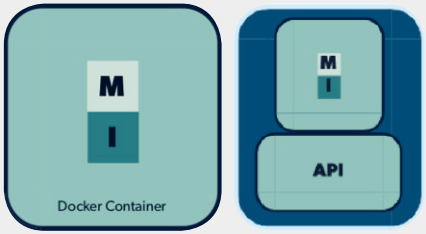
In commercial deployments, we might want to violate this principle somewhat to produce custom builds for customers. What if your customer wants a speech recognition system that can detect particular words such as (heaven forbid) "Brexit" but never filler words such as "mmm" or "umm"? Then we might want to have one final customisation step that incorporates customer-specific changes.

But as with everything here, there are many trade-offs in play. Do we create custom models for different customers? Or do we create the best generic model you can? The decision ultimately depends on the resources available, the business model and the company's long-term strategy.

Once out in the wild, it is important that containers are actively monitored. We must have ongoing confidence that there have been no performance regressions in the field and that the metrics that we have defined and care about are still within acceptable tolerances. We must be able to track and continuously monitor what works, and what doesn't, allowing us to make necessary improvements in a principled way. Ideally, this will recapitulate the testing and metric testing that happened previously in the CI pipeline. Tools such as Prometheus can be configured to achieve this.

Example inference code and model packaged inside a Docker container

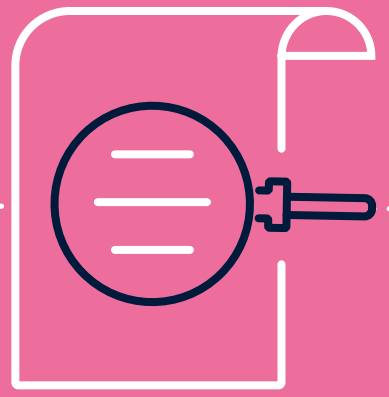
The following diagram shows the final packaged core artifact whereby an inference (I) and model (M) are packaged up and placed inside a Docker container. That container is then further wrapped with an API to service requests.



Summary

Deploying ML in a clean and effective way is non-trivial and requires careful thought. Frameworks will come and go but there are some high-level principles which should stand the test of time. Additionally, there will always be many trade-offs to consider, including speed vs accuracy, cost vs convenience, and API simplicity vs extensibility. It is the job of the ML practitioner to strike the right balance across all these trade-offs to deliver a product which is solid and continues to delight the end-user.

Will Williams, Machine Learning Engineer, Speechmatics




SPEECHMATICS