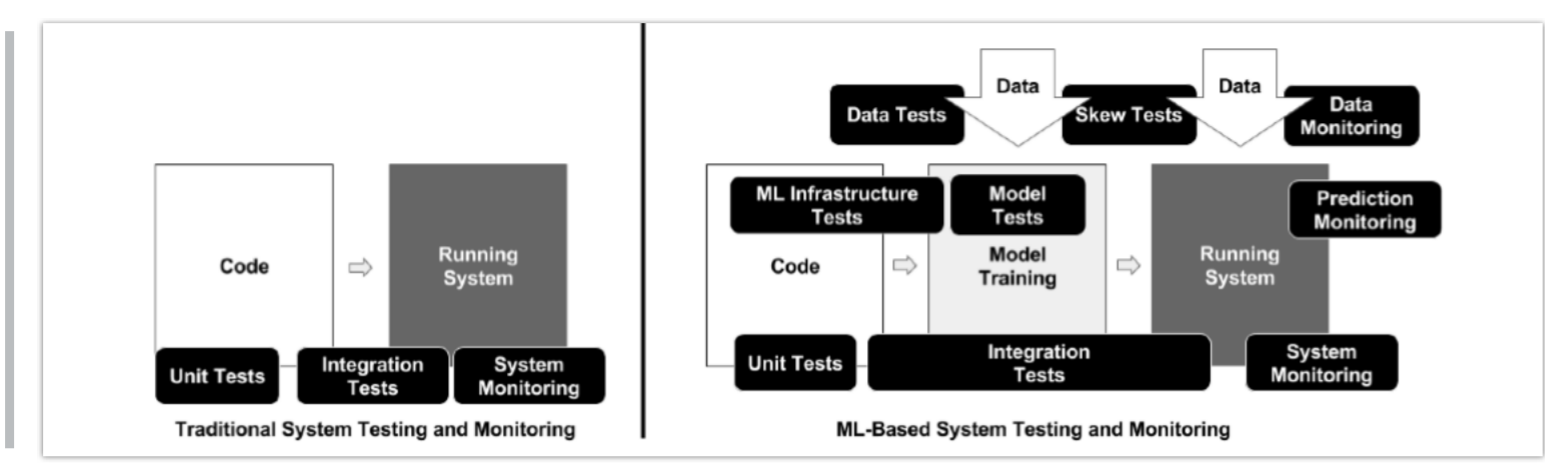


▼ Production-ready ML systems

▼ Testing and monitoring

▼ ML based system testing

(1) 2



**What to test: data, model, and infrastructure**

ML system testing is also more complex a challenge than testing manually coded systems, due to the fact that ML system behavior depends strongly on data and models that cannot be strongly specified a priori

**ML Test Score.pdf**

▼ TESTS FOR DATA

(1) 2

the behavior of ML systems is not specified directly in code but is learned from data

training data needs testing like code, and a trained ML model needs production practices like a binary does, such as debuggability, rollbacks and monitoring

**7 data tests**

attempt to add a sufficient set of tests of the data

1	Feature expectations are captured in a schema.
2	All features are beneficial.
3	No feature's cost is too much.
4	Features adhere to meta-level requirements.
5	The data pipeline has appropriate privacy controls.
6	New features can be added quickly.
7	All input feature code is tested.

**Table I**  
**BRIEF LISTING OF THE SEVEN DATA TESTS.**

▼ **Data 1: Feature expectations are captured in a schema**

(1) 2

It is useful to encode intuitions about the data in a schema so they can be automatically checked.

• an adult human is surely between one and ten feet in height. The most common word in English text is probably 'the', with other word frequencies following a power-law distribution. Such expectations can be used for tests on input data during training and serving

(1) 2

• To construct the schema, one approach is to start with calculating statistics from training data, and then adjusting them as appropriate based on domain knowledge

(1) 2

- start by writing down expectations and then compare them to the data to avoid an anchoring bias (1) 2

▼ **Data 2: All features are beneficial** (1) 2

A kitchen-sink approach to features can be tempting, but every feature added has a software engineering cost. Hence, it's important to understand the value each feature provides in additional predictive power (independent of other features).

- run this test are by computing correlation coefficients, by training models with one or two features, or by training a set of models that each have one of k features individually removed (1) 2

▼ **Data 3: No feature's cost is too much** (1) 2

It is not only a waste of computing resources, but also an ongoing maintenance burden to include ?-features that add only minimal predictive benefit [1].

- To measure the costs of a feature, consider not only added inference latency and RAM usage, but also more upstream data dependencies and other instability incurred due to dependencies (1) 2

▼ **Data 4: Features adhere to meta-level requirements** (1) 2

Your project may impose requirements on the data coming in to the system. It might prohibit features derived from user data, prohibit the use of specific features like age, or simply prohibit any feature that is deprecated. It might require all features be available from a single source. However, during model development and experimentation, it is typical to try out a wide variety of potential features to improve prediction quality

- Programmatically enforce these requirements, so that all models in production properly adhere to them (1) 2

▼ **Data 5: The data pipeline has appropriate privacy controls** (1) 2

Training data, validation data, and vocabulary files all have the potential to contain sensitive user data

- budget sufficient time during new feature development that depends on sensitive data to allow for proper handling. Test that access to pipeline data is controlled as tightly as the access to raw user data (1) 3

- test that any user-requested data deletion propagates to the data in the ML training pipeline, and to any learned models. (1) 3

• **Data 6: New features can be added quickly** (1) 3

The faster a team can go from a feature idea to the feature running in production, the faster it can both improve the system and respond to external changes

• **Data 7: All input feature code is tested** (1) 3

Feature creation code may appear simple enough to not need unit tests, but this code is crucial for correct behavior and so its continued quality is vital. Bugs in features may be almost impossible to detect once they have entered the data generation process, especially if they are represented in both training and test data.

▼ **TESTS FOR MODEL DEVELOPMENT** (1) 3

7 model tests

1	Model specs are reviewed and submitted.
2	Offline and online metrics correlate.
3	All hyperparameters have been tuned.
4	The impact of model staleness is known.
5	A simpler model is not better.
6	Model quality is sufficient on important data slices.
7	The model is tested for considerations of inclusion.

**Table II**  
**BRIEF LISTING OF THE SEVEN MODEL TESTS**

• **Model 1: Every model specification undergoes a code review and is checked in to a repository**

It can be tempting to avoid code review out of expediency, and run experiments based on one's own personal modifications.  
In addition, when responding to production incidents, it's crucial to know the exact code that was run to produce a given learned model. For example, a responder might need to re-run training with corrected input data, or compare the result of a particular modification. Proper version control of the model specification can help make training auditable and improve reproducibility.

▼ **Model 2: Offline proxy metrics correlate with actual online impact metrics**

Model 2: Offline proxy metrics correlate with actual online impact metrics: A user-facing production system's impact is judged by metrics of engagement, user happiness, revenue, and so forth. A machine learning system is trained to optimize loss metrics such as log-loss or squared error.  
A strong understanding of the relationship between these offline proxy metrics and the actual impact metrics is needed to ensure that a better scoring model will result in a better production system.

- The offline/online metric relationship can be measured in one or more small scale A/B experiments using an intentionally degraded model.

• **Model 3: All hyperparameters have been tuned**

A ML model can often have multiple hyperparameters, such as learning rates, number of layers, layer sizes and regularization coefficients. Choice of the hyperparameter values can have dramatic impact on prediction quality.

▼ **Model 4: The impact of model staleness is known**

Many production ML systems encounter rapidly changing, non-stationary data. Examples include content recommendation systems and financial ML applications. For such systems, if the pipeline fails to train and deploy sufficiently up-to-date models, we say the model is stale. Understanding how model staleness affects the quality of predictions is necessary to determine how frequently to update the model.  
If predictions are based on a model trained yesterday versus last week versus last year, what is the impact on the live metrics of interest? Most models need to be updated eventually to account for changes in the external world; a careful assessment is important to decide how often to perform the updates

- One way of testing the impact of staleness is with a small A/B experiment with older models. Testing a range of ages can provide an age-versus-quality curve to help understand what amount of staleness is tolerable.

• **Model 5: A simpler model is not better**

Regularly testing against a very simple baseline model, such as a linear model with very few features, is an effective strategy both for confirming the functionality of the larger pipeline and for helping to assess the cost to benefit tradeoffs of more sophisticated techniques.

▼ **Model 6: Model quality is sufficient on all important data slices**

Model 6: Model quality is sufficient on all important data slices: Slicing a data set along certain dimensions of interest can improve fine-grained understanding of model quality. Slices should distinguish subsets of the data that might behave qualitatively differently, for example, users by

country, users by frequency of use, or movies by genre.  
Examining sliced data avoids having fine-grained quality issues masked by a global summary metric, e.g. global accuracy improved by 1% but accuracy for one country dropped by 50%. This class of problems often arises from a fault in the collection of training data, that caused an important set of training data to be lost or late.

- Consider including these tests in your release process, e.g. release tests for models can impose absolute thresholds (e.g., error for slice x must be <5%), to catch large drops in quality, as well as incremental (e.g. the change in error for slice x must be <1% compared to the previously released model).

▼ **Model 7: The model has been tested for considerations of inclusion**

There have been a number of recent studies on the issue of ML Fairness [14], [15], which may arise inadvertently due to factors such as choice of training data. For example, Bolukbasi et al. found that a word embedding trained on news articles had learned some striking associations between gender and occupation that may have reflected the content of the news articles but which may have been inappropriate for use in a predictive modeling context [14]. This form of potentially overlooked biases in training data sets may then influence the larger system behavior.

- Tests that can be run include examining input features to determine if they correlate strongly with protected user categories, and slicing predictions to determine if prediction outputs differ materially when conditioned on different user groups.
- the approach of collecting more data to ensure data representation for potentially under-represented categories or subgroups can be effective in many cases

▼ **7 ML infrastructure tests**

TESTS FOR ML INFRASTRUCTURE

**BRIEF LISTING OF THE ML INFRASTRUCTURE TESTS**

1	Training is reproducible.
2	Model specs are unit tested.
3	The ML pipeline is Integration tested.
4	Model quality is validated before serving.
5	The model is debuggable.
6	Models are canaried before serving.
7	Serving models can be rolled back.

An ML system often relies on a complex pipeline rather than a single running binary

• **Infra 1: Training is reproducible**

Ideally, training twice on the same data should produce two identical models.

• **Infra 2: Model specification code is unit tested**

Although model specifications may seem like "configuration", such files can have bugs and need to be tested

▼ **Infra 3: The full ML pipeline is integration tested**

A complete ML pipeline typically consists of assembling training data, feature generation, model training, model verification, and deployment to a serving system.

- Faster running integration tests with a subset of training data or a simpler model can give faster feedback to developers while still backed by less frequent, long running versions with a setup that more closely mirrors production.

▼ **Infra 4: Model quality is validated before attempting to serve it**

After a model is trained but before it actually affects real traffic, an automated system needs to inspect it and verify that its quality is sufficient; that system must either bless the model or veto it, terminating its entry to the production environment

- test for both slow degradations in quality over many versions as well as sudden drops in a new version. For the former, setting loose thresholds and comparing against predictions on a validation set can be useful; for the latter, it is useful to compare predictions to the previous version of the model while setting tighter thresholds.

(1) 5

#### ▼ **Infra 5: Model allows debugability**

(1) 5

The model allows debugging by observing the step-by-step computation of training or inference on a single example

- When someone finds a case where a model is behaving bizarrely, how difficult is it to figure out why? Is there an easy, well documented process for feeding a single example to the model and investigating the computation through each stage of the model
- An internal tool that allows users to enter examples and see how the a specific model version interprets it can be very helpful. The TensorFlow debugger [17] is one example of such a tool.

(1) 5

(1) 5

#### ▼ **Infra 6: Models are tested via a canary process before deployment**

(1) 5

Models are tested via a canary process before they enter production serving environments

- One recurring problem that canarying can help catch is mismatches between model artifacts and serving infrastructure. Modeling code can change more frequently than serving code, so there is a danger that an older serving system will not be able to serve a model trained from newer code
- To mitigate the new-model risk more generally, one can turn up new models gradually, running old and new models concurrently, with new models only seeing a small fraction of traffic, gradually increased as the new model is observed to behave sanely.

(1) 5

(1) 5

- **Infra 7: Models can be rolled back**

(1) 5

Models can be quickly and safely rolled back to a previous serving version

#### ▼ **7 monitoring tests**

(1) 6

MONITORING TESTS FOR ML

1	Dependency changes result in notification.
2	Data invariants hold for inputs.
3	Training and serving are not skewed.
4	Models are not too stale.
5	Models are numerically stable.
6	Computing performance has not regressed.
7	Prediction quality has not regressed.

An ML system by definition is making predictions on previously unseen data, and typically also incorporates new data over time into training. The standard approach is to monitor the system, i.e. to have a constantly-updated “dashboard” user interface displaying relevant graphs and statistics, and to automatically alert the engineering team when particular metrics deviate significantly from expectations. For ML systems, it is important to monitor serving systems, training pipelines, and input data.

- **Monitor 1: Dependency changes result in notification**

(1) 6

ML systems typically consume data from a wide array of other systems to generate useful features. Partial outages, version upgrades, and other changes in the source system can radically change the feature’s meaning and thus confuse the model’s training or inference, without necessarily producing values that are strange enough to trigger other monitoring

▼ **Monitor 2: Data invariants hold in training and serving inputs**

(1) 6

It can be difficult to effectively monitor the internal behavior of a learned model for correctness, but the input data should be more transparent

- for detecting problems where the world is changing in ways that can confuse an ML system.

(1) 6

- Using the schema constructed in test Data 1, measure whether data matches the schema and alert when they diverge significantly. In practice, careful tuning of alerting thresholds is needed to achieve a useful balance between false positive and false negative rates to ensure these alerts remain useful and actionable

(1) 6

▼ **Monitor 3: Training and serving features compute the same values**

(1) 6

The codepaths that actually generate input features may differ at training and inference time.

the different codepaths should generate the same values, but in practice a common problem is that they do not. This is sometimes called "training/serving skew" and requires careful monitoring to detect and avoid.

- imagine adding a new feature to an existing production system. While the value of the feature in the serving system might be computed based on data from live user behavior, the feature will not be present in training data, and so must be backfilled by imputing it from other stored data, likely using an entirely independent codepath. Another example is when the computation at training time is done using code that is highly flexible (for easy experimentation) but inefficient, while at serving time the same computation is heavily optimized for low latency.

(1) 6

- To measure this, it is crucial to log a sample of actual serving traffic

(1) 6

- Another approach is to compute distribution statistics on the training features and the sampled serving features, and ensure that they match. Typical statistics include the minimum, maximum, or average, values, the fraction of missing values, etc.

(1) 6

▼ **Monitor 4: Models are not too stale**

(1) 6

- we recommend monitoring how old the system in production is, using the prior measurement as a guide for determining what age is problematic enough to raise an alert

(1) 6

▼ **Monitor 5: The model is numerically stable**

(1) 7

Invalid or implausible numeric values can potentially crop up during model training without triggering explicit errors, and knowing that they have occurred can speed diagnosis of the problem.

- Explicitly monitor the initial occurrence of any NaNs or infinities. Set plausible bounds for weights and the fraction of ReLU units in a layer returning zero values, and trigger alerts during training if these exceed appropriate thresholds.

(1) 7

▼ **Monitor 6: The model has not experienced degradation**

(1) 7

Such as a dramatic or slow-leak regressions in training speed, serving latency, throughput, or RAM usage

- it is useful to slice performance metrics not just by the versions and components of code, but also by data and model versions. Degradations in computational performance may occur with dramatic changes (for which comparison to performance of prior versions or time slices can be helpful for detection) or in slow leaks (for which a pre-set alerting threshold can be helpful for detection)

(1) 7

▼ **Monitor 7: The model has not experienced a regression in prediction quality**

(1) 7

Monitor 7: The model has not experienced a regression in prediction quality on served data

- measuring a model's quality on that validation data before pushing it to serving is only an estimate of quality metrics on actual live serving inputs

(1) 7

- Measure statistical bias in predictions, i.e. the average of predictions in a particular slice of data.

(1) 7

- In some tasks, the label actually is available immediately or soon after the prediction is made (e.g. will a user click on an ad). In this case, we can judge the quality of predictions in almost real-time and identify problems quickly.

(1) 7

- can be useful to periodically add new training data by having human raters manually annotate labels for logged serving inputs

(1) 7