Gartner

Incorporate, Test, Deploy and Maintain Machine Learning Models in Production Applications

Published 22 November 2022 - ID G00764853 - 48 min read

By Analyst(s): Steve Deng, Daniel Cota

Initiatives: Application Architecture and Integration for Technical Professionals; Adopt Modern Architectures and Technologies; Analytics and Artificial Intelligence for Technical Professionals

Incorporating machine learning models into applications enables delivery of capabilities such as real-time personalization and sentiment analysis. Application technical professionals must embrace new patterns and processes to address the unique challenges of building ML into applications.

Overview

Key Findings

- Using machine learning (ML) models to implement artificial intelligence (AI) features within applications adds an extra layer of complexity due to the life cycle and probabilistic nature of the models.
- ML models need extensive validation before moving to production due to the probabilistic nature of the training process.
- Monitoring and tracking a specific ML model in production requires a combination of traditional application performance monitoring (APM) approaches with real-time observations for model-specific metrics.
- Successful production deployments of ML models that deliver AI features for critical business applications use cloud-native packaging and deployment techniques such as containers, serverless, CI/CD, APIs and monitoring.

Recommendations

Application technical professionals seeking to incorporate ML models into their applications must:

- Ensure that all ML models deployed into production are actively monitored to detect, and updated to avoid, erosion of model accuracy as live data diverges from the training data used.
- Create a universal identification system for model versions for use by all teams to ensure traceability of deployed models and the dataset and related parameters. Link this identifier to all deliverables from model training and label all log entries associated with a specific model version.
- Automate the process of packaging ML models with their associated dependencies into target model inference runtimes and link them to a CI/CD process.
- Establish and iterate on probabilistic model testing criteria with the data science team and the business to ensure these ongoing tests account for uncertainty and variance with predictions.

Problem Statement

The use of machine learning (ML) has exploded over the last decade, and the demand and opportunities to use ML-augmented applications continues to grow. On the journey to build AI-enabled systems using ML, organizations' data science teams have developed well-defined processes to build, train and test models. These ML models are useful to application architects only when they can be integrated into an application. The challenge is that ML models cannot simply be integrated into an application and supported in the same manner as other software components.

Machine learning includes a training process that generates a model by taking a model configuration and iterating over a set of existing training data. The resulting model can predict an answer based on input data that is similar to — but often unique from — the training data. The trained model is a static data structure that can be copied, deployed and served to provide predictive capabilities in applications and business processes.

ML models are subject to a variety of factors that are unfamiliar to application architects, including:

- Model updates and optimizations are required to achieve the desired accuracy and performance in the inference environment.
- Model outputs provide a probability of correctness rather than a definitive answer.

- Model relevance decays and drifts as data in the runtime environment becomes more distinct from the training data over time.

These factors require different approaches compared to more traditional software components that must be addressed in two distinct ways — integration and life cycle. This raises the question that this research aims to answer:

How do I integrate machine learning models into my application and manage the life cycle of these models in production?

This research guides application architects through the process of integrating ML into their applications to provide AI behaviors through the use of new practices, tools and processes. It also discusses the upfront work of moving ML from experimental proof of concept into production systems as well as ongoing maintenance.

This research does not cover data science responsibilities, including model creation and development, model training, data pipelines or model governance.

We recommend reviewing the following research for details on these topics:

- Understanding MLOps to Operationalize Machine Learning Project
- Machine Learning Playbook for Data and Analytics Professionals
- Democratize Data Science Initiatives With Augmented DSML
- Demystifying XOps: DataOps, MLOps, ModelOps, AIOps and Platform Ops for AI
- Incorporate Explainability and Fairness Within the AI Platform
- Applying AI in Business Domains
- Applying AI — Techniques and Infrastructure

The Gartner Approach

ML and AI are important trends in modern application architectures. Many organizations are experimenting with ML models. However, bringing these models from experimentation to production requires an extension to the software development life cycle.

The framework described in this document is designed for application architects who have experience in architecting modular applications and are considering the use of machine learning and/or artificial intelligence models to enhance their applications. The data driven, iterative, probabilistic nature of ML model development are very different to traditional software development. Application architects should treat machine learning as a new service implementation architecture. Application professionals, data scientists, and ML engineering teams should learn from each other. and collaborate closely to implement continuous model testing and validation throughout your model development and productization life cycle

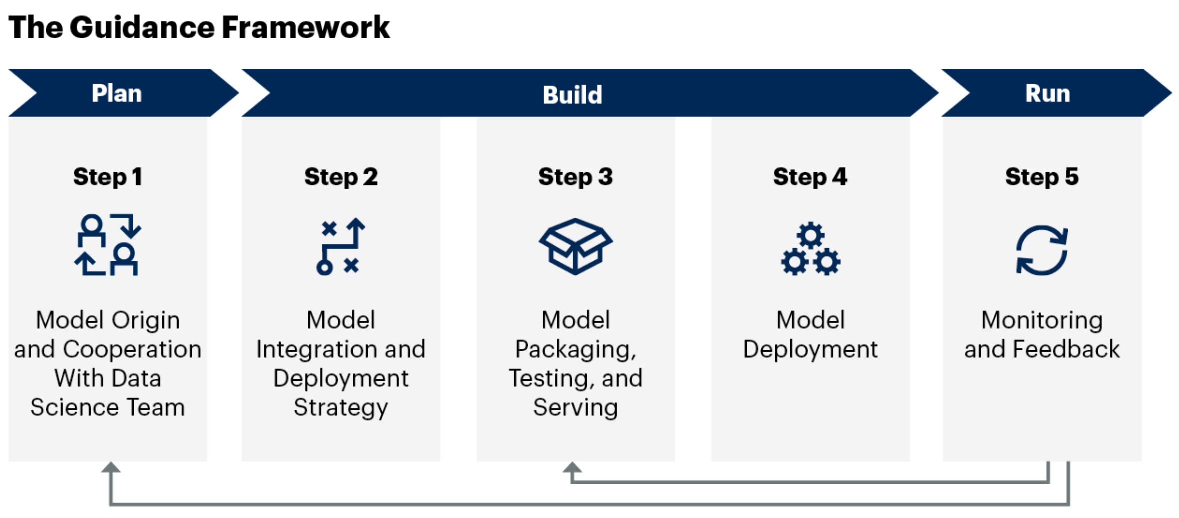
This guidance framework provides a systematic approach for including machine learning models in production applications.

The Guidance Framework

The Gartner framework for successfully integrating ML models into your applications and delivery life cycle consists of five steps, as depicted in Figure 1 and explained in detail in the following sections.

- Step 1: Model Origin and Collaboration With Data Science Team
- Step 2: Model Integration and Deployment Strategy
- Step 3: Model Packaging, Testing, and Serving
- Step 4: Model Deployment
- Step 5: Monitoring Model Performance and Creating a Feedback Loop

Figure 1: The Guidance Framework


The Guidance Framework
Plan
Build
Run
Step 1
Model Origin and Cooperation With Data Science Team
Step 2
Model Integration and Deployment Strategy
Step 3
Model Packaging, Testing, and Serving
Step 4
Model Deployment
Step 5
Monitoring and Feedback

Source: Gartner
764853_C

Step 1: Model Origin and Collaboration With Data Science Team

ML skills are in high demand, and many technical professionals are eager to develop them. As the first step, confirm that there is indeed a valid use case for using ML and that there will be a suitable ROI to justify the investment requirements. Additionally, identify suitable key performance indicators with your model authors/engineers, as well as security, legal and business partners and set up appropriate measurements. Create a cadence for review of these KPIs at least quarterly to ensure that they remain relevant.

Refer to Gartner's User-Case Prism line of research such as:

- Infographic: Artificial Intelligence Use-Case Prism for Finance
- Infographic: Artificial Intelligence Use-Case Prism for Supply Chain
- Infographic: Artificial Intelligence Use-Case Prism for Life Science Manufacturers

These infographics will help you to identify the best AI use cases to deliver the right business values.

Identify the Model Origin

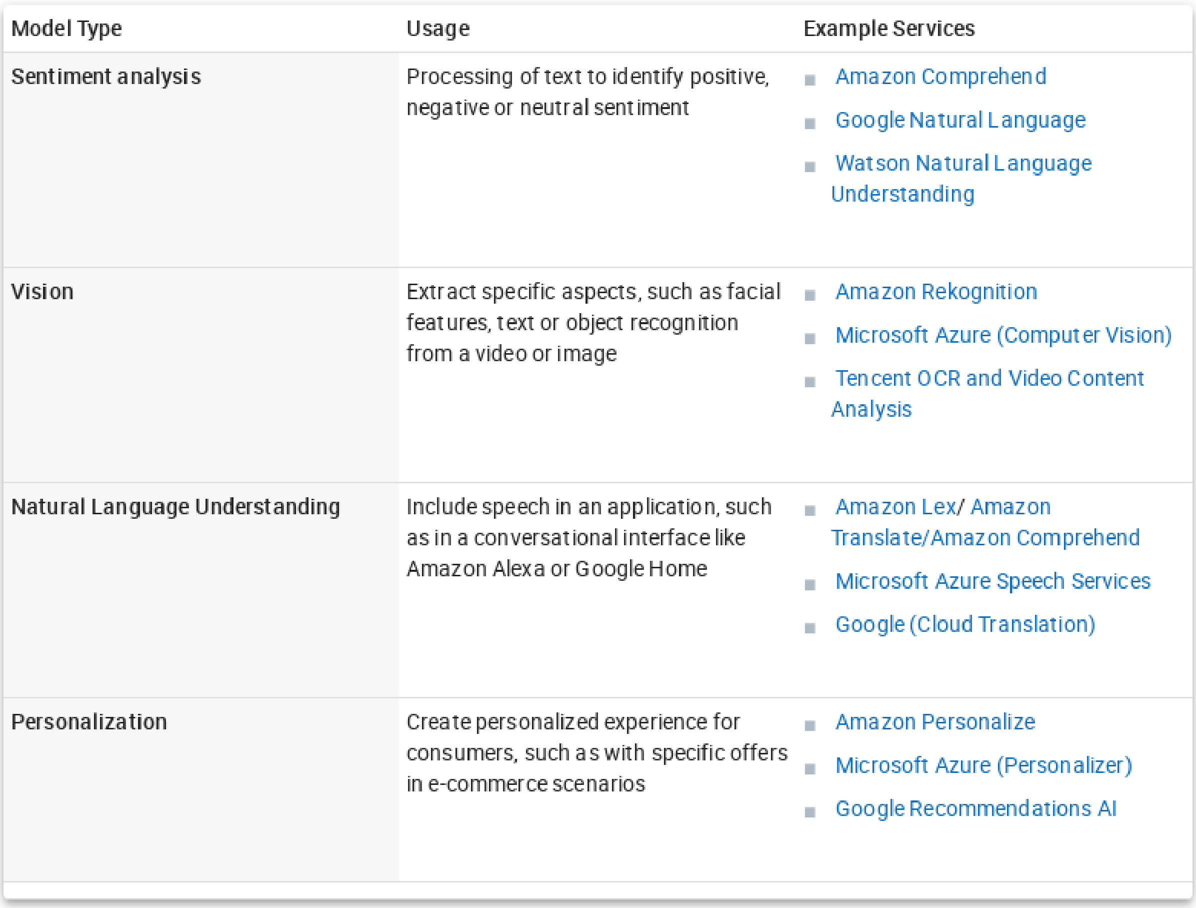
ML models originate from a number of locations. Application and ML teams may utilize models that are pretrained, auto-generated, bespoke, or any combination of these options. The models are orchestrated in workflow to solve business problems. The origin of these models and who is responsible for the model over the lifetime of the application must be clearly identified before it is integrated into an application.

The application architect must confirm that the model is providing a unique and differentiated feature. ML models can bring unique features and insights to an application, but not every model needs to be, or should be, custom-developed and trained. Common scenarios that can be solved with off-the-shelf models and ML-based services include sentiment analysis, image recognition, speech to text, text to speech and personalization engines.

Cloud AI developer services, such as Alibaba Cloud, Amazon Web Services, Microsoft, Google, and IBM Watson, provide APIs for accessing such models, which will meet the requirements for many organizations. Table 1 gives an indicative list of such services. See Critical Capabilities for Cloud AI Developer Services for more detail.

Table 1: API-Based ML Services

(Enlarged table in Appendix)



Model Type	Usage	Example Services
Sentiment analysis	Processing of text to identify positive, negative or neutral sentiment	■ Amazon Comprehend ■ Google Natural Language ■ Watson Natural Language Understanding
Vision	Extract specific aspects, such as facial features, text or object recognition from a video or image	■ Amazon Rekognition ■ Microsoft Azure (Computer Vision) ■ Tencent OCR and Video Content Analysis
Natural Language Understanding	Include speech in an application, such as in a conversational interface like Amazon Alexa or Google Home	■ Amazon Lex/ Amazon Translate/Amazon Comprehend ■ Microsoft Azure Speech Services ■ Google (Cloud Translation)
Personalization	Create personalized experience for consumers, such as with specific offers in e-commerce scenarios	■ Amazon Personalize ■ Microsoft Azure (Personalizer) ■ Google Recommendations AI

Source: Gartner (November 2022)

Define Nontechnical Contracts

In most organizations, the creation and training of ML models will be done as part of a larger data science and ML operations (MLOps) pipeline. To consistently integrate these models into applications, a baseline SLA must be agreed on between the ML engineer/architect and the application architect, and adhered to. Even if a model is developed by the application team, the same set of nontechnical contracts must be in place.

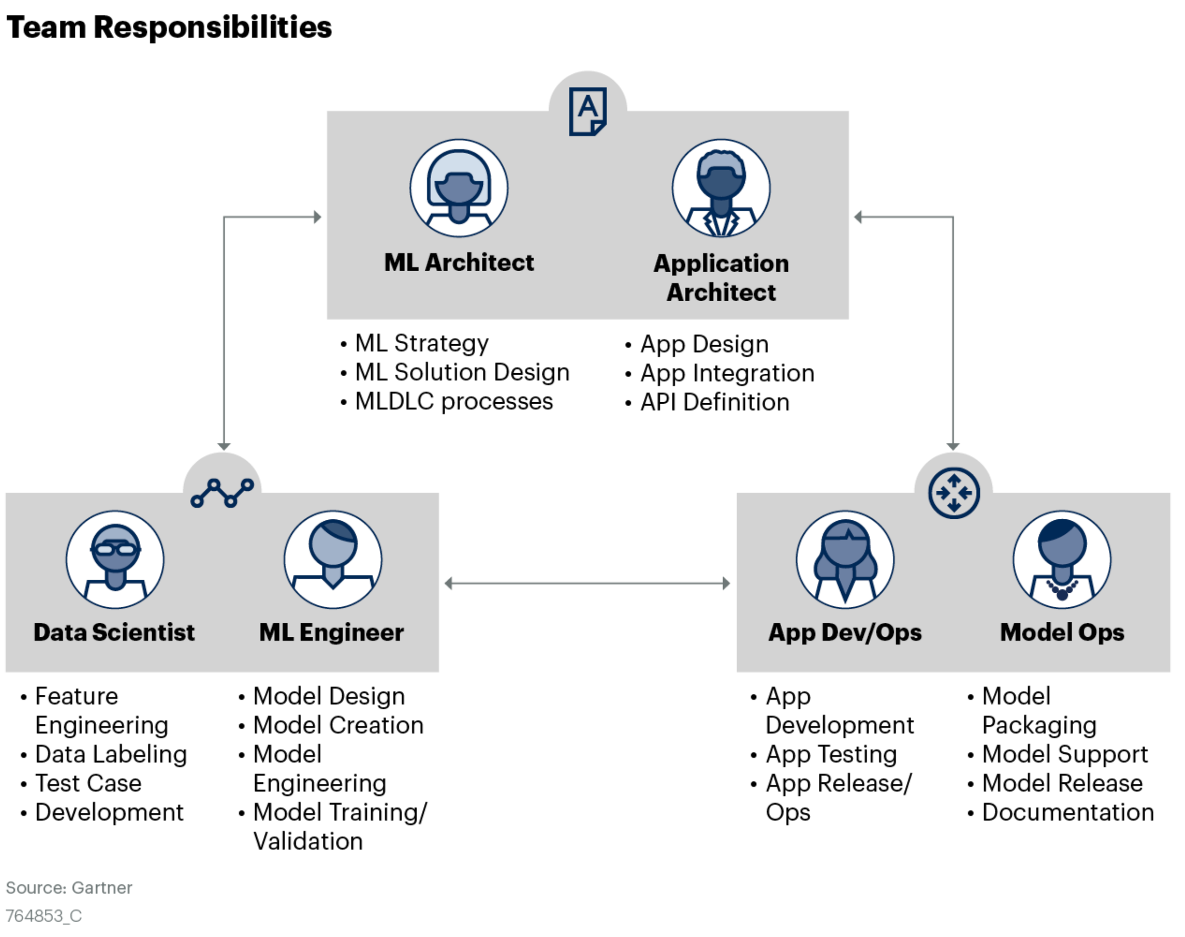
The data science team will have a number of roles, from traditional data scientist to machine learning engineer or architect. It is important to note that the ML engineer/architect may not be a discrete role and could be fulfilled by a team member with other areas of responsibility.

For more details on MLOps and the skills required by machine learning engineers and architects, see this Gartner research:

- Understanding MLOps to Operationalize Machine Learning Projects
- Essential Skills for Machine Learning Architects
- Essential Skills for AI Engineers

The ML architect and application architect are the primary points of contact for the application and data science team. The data scientists and ML engineers jointly curate the data and develop the ML model. Model Ops operationalize the model and the DevOps team develop, deploy and operate the ML-driven application. See Figure 2 for more details on the responsibilities of the different roles.

Figure 2: Team Responsibilities



The application architect and ML architect and engineer must agree on the following components of nontechnical contract:

Unique Model ID, Version, Deliverables and Metadata

A model saved in source control that includes:

- Model data and representation details
- Model output specification
- Descriptions on any specific hypertuning parameters used
- Code and configuration of any feature engineering
- Link to a snapshot of the training data

You can also tag each model with a human-readable identifier, and versions.

Sometimes related models may also be associated with each other in a model group.

Version Information for Libraries and Dependencies

Version information should be collected and stored in a machine-readable format to enable automation.

Key information to capture includes:

- Specific libraries and dependencies used by this iteration of the model.
- Training framework and runtime versions used, for example, is a specific Python module required?

Testing and Validation

What testing has been completed on the model, and what metrics should be applied to detect model decay and other anomalies?

Examples include:

- **Prediction distribution:** Compare the prediction distributions in the inference environment with similar findings from the training environment. Deviations beyond an agreed-upon tolerance should result in system alerts.

Additional metrics to measure model performance may be:

- **Confusion matrix:** The two-by-two matrix that lists the true positives, false positives, true negatives and false negatives.
- **"Gain and lift" charts:** Visual aids measuring the effectiveness of a predictive model calculated as the ratio between the results obtained with and without the predictive model.
- **Root mean square error (RMSE):** Provides the standard deviation between predictions and observations. This should be similar between the training dataset, test dataset and ongoing predictions.

These testing details will also be used in monitoring the model once in production.

Performance Characteristics

What are the baseline performance characteristics of the model?

Examples include:

- **Model size:** What is the actual size of the model?
- **Inference performance:** How quickly is a result returned for inference?
- **Memory consumption:** How much memory is the model using once in memory?

These characteristics are important for deciding if a model is suitable for use in production. For example, if the predictive performance of a model increases by 1%, but the interference performance decreases by a factor of 10, should it be a candidate for promotion into a production environment?

Ground Truth Dataset

The data science team should provide a fully labeled dataset that can be used as "ground truth" for testing and validation. This may be an existing dataset, a known base model, or a dataset that is specifically labeled by the data science team.

This dataset can be randomly sampled and run against models in production to detect anomalies that may need to be investigated.

Model Governance

The governance of ML models is outside the scope of this research. However, it is important that the application understands how a model is governed and who holds responsibility for this governance. Organizations that are planning to consume ML models via an easily consumed service (such as those defined in Table 1) should confirm that the governance of these models and available information meets their legal and regulatory requirements.

In Hype Cycle for Artificial Intelligence, 2022 Gartner defines AI TRiSm as follows: “AI trust, risk and security management ensures AI model governance, trustworthiness, fairness, reliability, robustness, efficacy and data protection. This includes solutions and techniques for model interpretability and explainability, AI data protection, model operations and adversarial attack resistance.”

For more details on AI trust and risk management see:

- A Comprehensive Guide to Responsible AI
- Top 5 Priorities for Managing AI Risk Within Gartner’s MOST Framework
- Market Guide for AI Trust, Risk and Security Management

Step 2: Model Integration and Deployment Strategy

How and where to integrate ML models in your application is a major consideration. You must ensure that you are choosing the correct approaches around integration, deployment and hardware requirements for your application. This will have a direct impact on how you deploy, manage and update models in your application.

The key focus areas are:

- Identify the Correct Integration Approach
- Identify the Correct Deployment Mode
- Define the Frequency of Model Updates
- Identify Specific Hardware Requirements

Identify the Correct Integration Approach

ML models can be integrated into applications in four distinct ways. The interface to the model will also dictate its output format.

These four integration approaches are:

1. **Consumption as a service:** In this approach, the model is invoked via an interface such as a REST or gRPC API. For example, in a microservices environment, you could make the model available as a service to which other services send a request and consume the response.

2. **Embedded:** The ML model is "embedded" in a device. For example, you are doing some edge processing on data to make a decision. The Embedded model may be called directly within the code, such as invoking a PyTorch model from C++. The model is deployed as an integral part of the application and used exclusively by the application.

3. **Streaming data:** The model is invoked as part of a stream processing step. For example, if you are using Kafka Streams, the model may be invoked as part of a Kafka Streams application, so every piece of data that passes through the stream can have the model applied as part of its processing in real time.

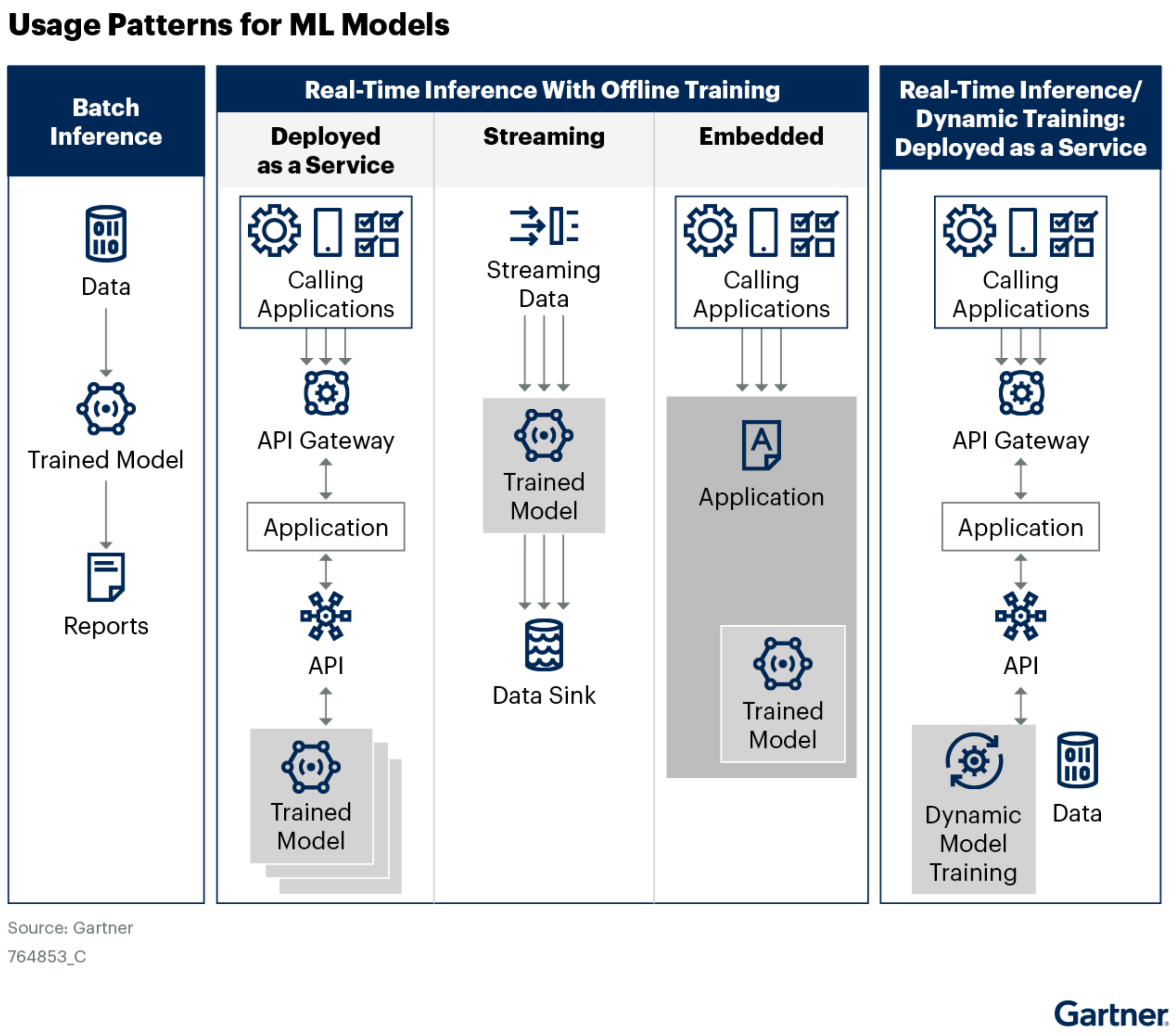
4. **Batch or file reference:** The mode is invoked as part of a batch processing job or by means of a reference to a data file, for example, as part of a reporting system. Executing model inference in batch can deliver higher transaction throughput due to better CPU/GPU utilization when compared to invoking the model as a service for every transaction

Identify the Correct Deployment Mode

ML model deployments can be categorized into one of three modes (see Figure 3):

- Batch inference.
- Real-time inference with offline training, including:
    - Consumption as a service
    - Streaming
    - embedded
- Real-time inference with incremental training.

Figure 3: Usage Patterns for Machine Learning Models



Gartner 2019

It is important to note that most "real-time" training of models is in fact the second of these modes, where there is a rapid iterative retraining of a model with new data, and the model is introduced into production using a deployment strategy. We discuss three common deployment strategies in Step 4.

Batch Inference

In a batch inference mode, a set of input data is run through a model and a set of output predictions is generated. The output for a given input will not change until the model is retrained or updated and the batch job is rerun. For most organizations, this will feel very similar to a standard analytics pipeline.

Use batch inference for workloads that involve large datasets or require significant processing that can be executed offline.

Real-Time Inference With Offline Training

Real-time inference coupled with offline training is the most common scenario and usage pattern for ML in production applications. In this approach, a model is integrated into your application using one of three integration patterns discussed earlier: consumption as a service, embedded mode or streaming data.

Alongside this integration point, the data science team should be retraining and optimizing the model as new data arrives. This process must be automatic and completely reproducible. New models are then released to the application team, validated and promoted into the application using one of the deployment techniques which we discuss further in Step 4. The cadence of this update process is defined on a per-model basis. It will be driven by the cost of training, the availability of data to improve the model, and the likelihood of model drift as data and expected behavior in the runtime environment change (through seasonal or consumer trend changes, for example).

A high cadence for this training and redeployment is often used to give the impression of dynamic training.

Real-Time Inference With Incremental Training

The real-time inference/incremental training scenario is rare and comes with significant technical challenges. Generally, dynamic training is only possible when the inferencing scenario has an inherent feedback loop that can detect gaps in model performance and supply new training data and labels, for example, from live human agents or users.

In most cases that have the appearance of dynamic training, what is actually occurring is rapid — and automated — iterative retraining of the data with controlled deployments. This is real-time inference/offline training, but at a speed that appears to be dynamic.

Define the Frequency of Model Updates

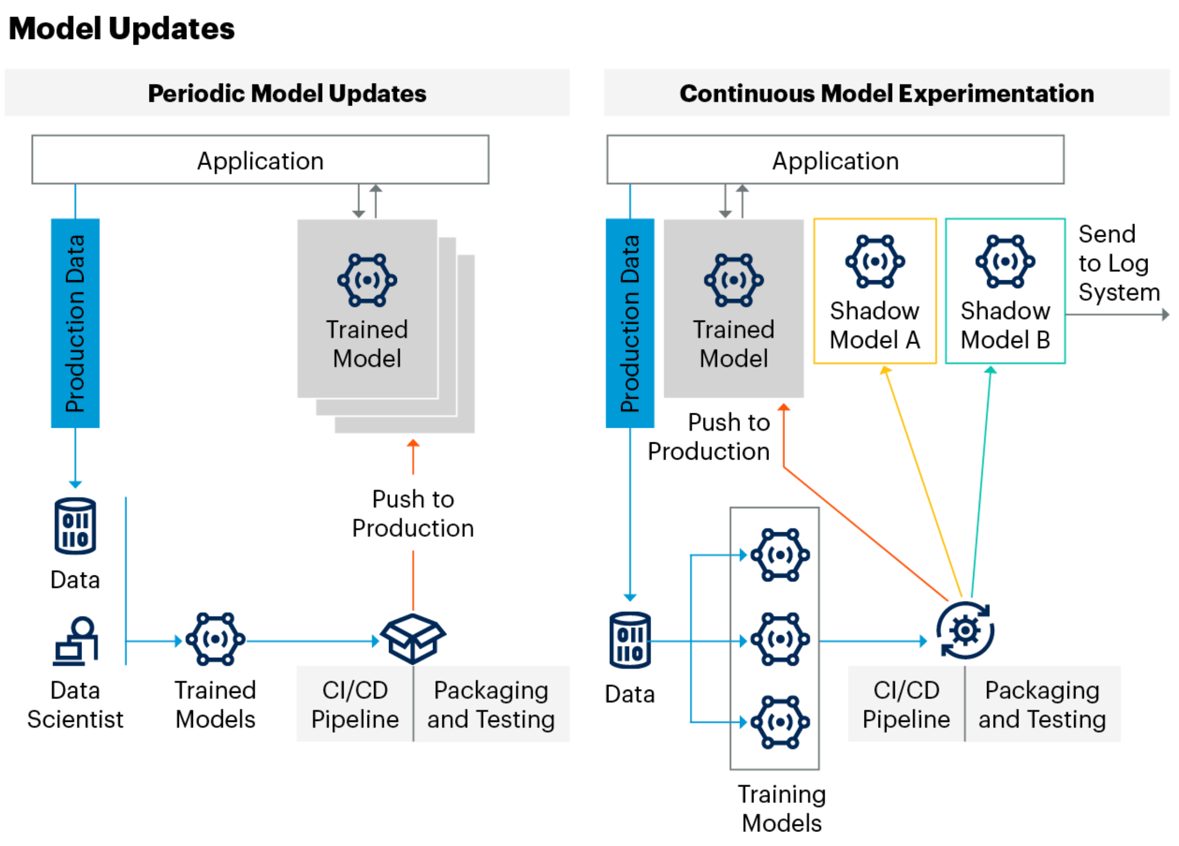
For real-world use, models will be relatively frequently updated. How you update and release the model is closely integrated with the work of your data science team, but these updates may not require the direct involvement of the data science team on a day-to-day basis.

Two scenarios exist (see Figure 4):

- **Periodic model updates:** Models are retrained and released in a clear systematic process with a handover from the data science team to the application team. The model is then brought into production.

- **Continuous model experimentation:** Several variants of the model are run in production, "shadowing" the production model. These experiments may be automatically generated, for example, by modifying hyperparameters or frequently retraining with new data, and released into a testing/shadowing pipeline. This approach is very similar to the kind of A/B testing often seen in a standard software development life cycle.

Figure 4: Model Updates



Source: Gartner
764853_C

In practical terms, if you have the correct release and deployment automation in place, you will be able support both scenarios for model updates, but you may need additional investment in three specific capabilities:

- **Traffic and request routing:** Can you route traffic requests to different endpoints within your application transparently, including support for shadow requests where no response is returned?

- **Comparative performance analytics:** Can you easily compare relevant performance metrics to systematically compare various models? It must be possible to automatically use this performance information to trigger follow-up actions, such as promoting a model to production.
- **Logging system:** Data from the models — such as when and how it is invoked, how long each inference request takes and any errors encountered — should be sent to a log system for ongoing analysis.

Identify Specific Hardware Requirements

Your model may require the use of specialized hardware such as GPUs to optimize inference performance. While you may not need this during your initial development effort, you need to evaluate your inference performance and scale requirements for production use.

The three key areas to consider are:

1. **Training environment:** There are times where GPUs are used for training and CPUs are used for inferencing. This is especially true in scalable cloud environments. Can you take advantage of hyperparameter tuning techniques such as autoML to optimize your model for deployment environment and inference performance?

2. **Deployment environment:** Will your model be deployed on existing hardware on-premises or in a cloud environment? Are there any specific hardware constraints, such as deploying on a mobile device?

3. **Software support:** Will the serving framework you choose be compatible with the type of hardware acceleration you are planning to use? For example, a framework that works well for GPUs may not support field-programmable gate arrays (FPGAs).

Step 3: Packaging, Testing and Serving

The packaging and testing of your ML models will have a direct impact on how you deploy, manage and update models. You must automate all aspects of the packaging and testing life cycle to be successful with bringing ML into production at scale. Once the ML engineer has prepared a model, it must then be packaged into a usable artifact for deployment.

The key focus areas are:

- Model Optimization
- Ongoing Dependency Management
- Model Testing and Validation
- Model Serving Frameworks
- Logging and Telemetry Data

Model Optimization

Post-training ML models are, at their core, complex graphs with an associated configuration file. They often require too much memory, network bandwidth, or computing power to execute well in your target inference environment.

You must optimize these models across these key criteria to improve their inference performance:

1. Reduce latency and cost of inference by minimizing compute time.
2. Reduce costs by minimizing the memory, compute and power required to run the model.
3. Reduce storage and network bandwidth by decreasing the payload size of the model.

Common optimization techniques include: [1]

- **Model pruning:** Reduces parameter count and structural pruning
- **Post-training quantization:** Reduces precision on some or all of the model operations on tensors to reduce model size and take advantage of CPU and hardware acceleration.
- **Weight clustering:** Reduces the number of unique weight values in the model by sharing similar weights to improve model compression and reduce memory footprint.
- **Graph optimization:** Simplifies mode topology for more efficient and faster execution.

ONNX and TensorFlow both provide libraries, toolkits or APIs to complete the most common optimization steps. These steps may be performed in online or offline mode. In online mode, all optimizations are applied at initialization time before inferencing, which often increases start up time. In offline mode, optimization must be run on all models prior to bringing them into a production environment. TensorFlow's Model Optimization Toolkit (MOT) offers experimental collaborative optimization APIs that allow you to combine different optimization techniques for your specific use case and requirement. While model optimization should be completed by the MLOps team, it can also be incorporated into your CI pipeline.

Threading and Parallelism

Running an evaluation on the same instance of an ML model may not be thread-safe. For example, this would be an issue when a model is embedded in a multithreaded application and each thread attempts to use the same instance of the model.

If your application is multithreaded, you must ensure that a separate instance of the model loads for each inference. Many ML runtime platforms (such as TensorFlow Python threading), offer configuration/tuning parameters to allocate resource pools to support parallel execution of a particular model. There will be a performance impact with this approach, and you should carefully evaluate the trade-offs versus processing each request sequentially.

Ongoing Dependency Management

To bring a model into production, you require more than just the model itself. You will also need to include supporting software components, such as a model serving framework. Depending on your architecture, it can be beneficial to update these components independently of the model and redeploy into a continuous integration pipeline as you would do for any other software component.

The key dependencies to take into account are:

- **Operating system:** What operating system will be used to host the model framework and model? This is most relevant on edge devices. Common operating systems include Linux, Windows, macOS, iOS, Android and BlackBerry QNX.
- **Supporting low-level device frameworks:** Will specialized hardware such as GPUs be required for inferencing? What specific driver and software support is needed? Sample hardware device frameworks include CUDA and NVIDIA Triton.

- **ML frameworks:** Which ML framework are you using, and what components of it are required for inferencing? Are there external dependencies that need to be taken into account? Are these general-purpose (training and inference) or optimized for inference? Are any model translation tools required? In the case of a containerized deployment, you will only want to include one framework. Examples include TensorFlow, PyTorch, Apache MXNet, scikit-learn and Keras.

- **ML model serving platform:** When it's time to deploy your model for inference/predictions, you need a hosting platform to optimize your models for execution. Do you need to increase throughput and reduce latency? Do you need to optimize your model for specific hardware or GPU? Do you want a serving platform to also monitor and track your model's deployed instances and versions?

- **Programming language dependencies:** What programming language runtimes and development tools will you need in place? Python is the most common language used with modern ML frameworks. In the case of a containerized deployment, you will only want to include one language runtime. Other languages include R, Java, Scala and .NET.

- **Training dataset and feature extraction code:** What is the version of the training dataset and associated feature extraction code? What are the relevant statistical and other properties of the training dataset that can be used to detect data drift.

Each of these components will have security as well as functional releases, and your packaging process must proactively update these components in line with your organization's security posture.

Use containers for packaging and managing your model and its dependencies if you already have container knowledge in-house, and if serving the model in a separate remotely invoked process is practical for your application.

It is possible to build and maintain these ML container images in-house, but given the requirements for ongoing maintenance, you may prefer to use external sources. Every major cloud provider offers a variety of prebuilt container and VM images that can be used as building blocks for bringing models into production (see AWS Deep Learning Containers, Microsoft Azure Data Science Virtual Machines and Google Cloud Deep Learning Containers, for example). Additionally, hardware vendors such as NVIDIA provide curated container images for a variety of frameworks (see Containers, NVIDIA).

No single provider supports all available frameworks, and there are significant variations in the release and update schedules. You must fully research your chosen provider to ensure it meets your requirements.

Once you have chosen your preferred packaging mechanism, and identified the requirements, you must put in place a standard build-and-release process for these components. Details about the contents of your specific images must be kept in version control, and the entire packaging process must be automated.

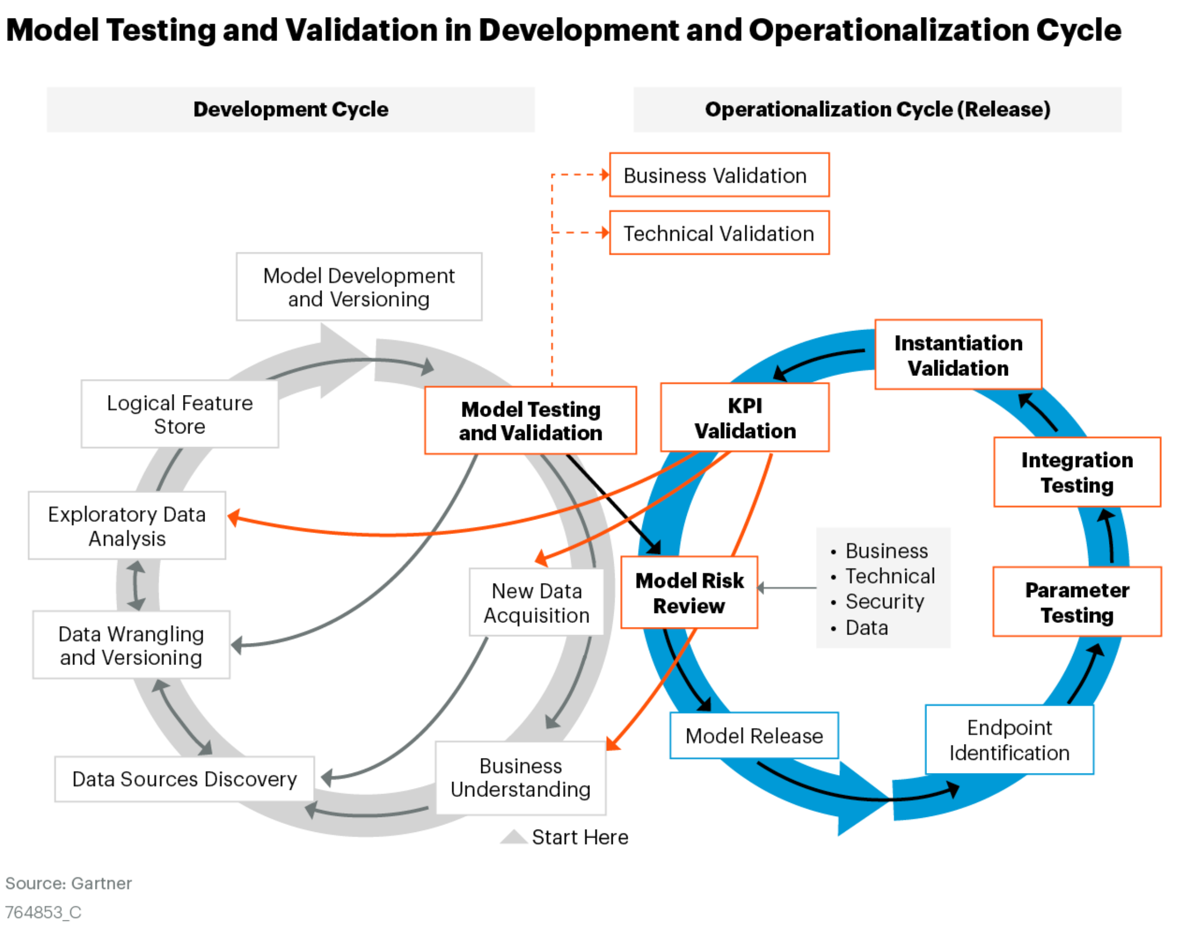
This part of the process is no different to any other continuous integration pipeline. To learn more about containerization, continuous integration and deployment refer to the following Gartner research:

- Assessing Containerization for Advanced Analytics Initiatives
- Solution Path for Continuous Delivery With DevOps
- Designing and Operating DevOps Workflows to Deploy Containerized Applications With Kubernetes

Model Testing and Validation

Model testing and validation should be an integral part of your model development and operational life cycle that requires collaboration among members of your ML engineering practice as shown in Figure 5.

Figure 5: Model Testing and Validation



In the model development cycle, model testing and validation involves:

- **Business validation** to ensure the developed model can deliver the desired business value.
- **Technical validation** to ensure the model is delivering the technical accuracy and performance needed when tested against the training dataset while meeting all the interpretability requirements.

In the model operationalization cycle, application and model testing, validation, and auditing process is more extensive, including

- **Model risk review:** Thoroughly test and review the model to ensure that it does not expose the organization to business risks and that it has sufficient protection against — and is being continuously monitored for — data poisoning attacks, adversarial attacks, query attacks and model manipulation attacks.
- **Parameter testing:** Verify that the input data used by the application using the model is aligned and consistent with input data used during model development.
- **Integration testing:** Test and validate that the model is functioning as expected when deployed and accessed along with other of the application components and systems.
- **Instantiation validation:** Validate that individual model instances or an ensemble of model instances are configured and deployed correctly.
- **KPI validation:** Measure and validate model performance against business and technical KPIs such as business drift, mathematical drift and data drift.

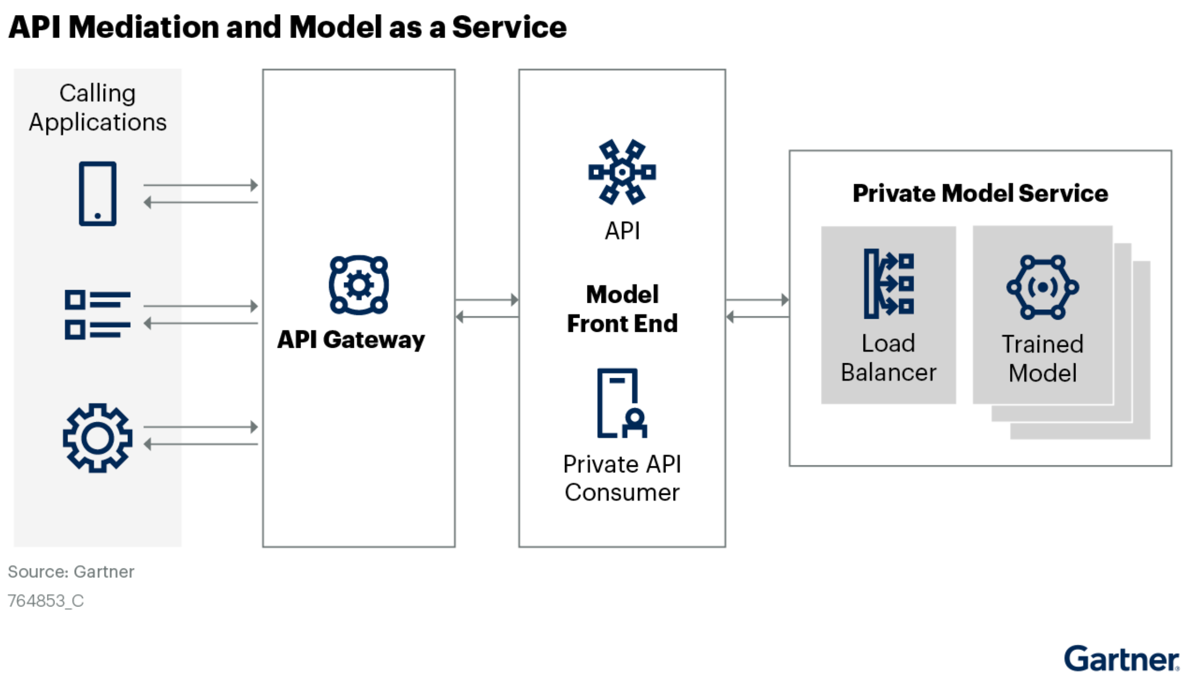
For details on application and model testing and validation, see Use Gartner's MLOps Framework to Operationalize Machine Learning Projects and Guide to Understanding and Developing Initial Approaches to Auditing AI-Related Risks

Model Serving Framework

After a ML model is trained and validated, a model serving framework would deploy it on a runtime environment to serve up the model's predictive capabilities to applications via API on cloud or on-premises. The serving platform is also responsible for runtime life cycle management of that model, including monitoring it for drift, and redeploying it when it's updated.

Many model serving frameworks are designed to be invoked directly via generic HTTP or gRPC, but these are not consumer-friendly API endpoints. If the inference API endpoints do not support functionality such as load balancing, throttling, SSL and authentication, they shouldn't be accessed by consumers directly. The best practice is to package them in a container, and use a mediated API pattern, as indicated in Figure 6. You will need to define a consumer-friendly API and then implement the resource and payload mediation logic in the model front end component to interact with the private model services.

Figure 6: API Mediation and Model as a Service



Key features of a model serving platform are:

- **Support multiple ML development frameworks:** A model serving platform should support multiple common ML frameworks (such as TensorFlow, PyTorch, XGBoost, ONNX, and scikit-learn) as well as custom-developed models. Data science teams want to retain maximum flexibility and the autonomy to select the best tools to satisfy their model development needs. KServe, BentoML, TorchSever and Triton are a few model serving platforms that support multiple ML frameworks

- **Management APIs/CLIs:** A model serving platform should provide management application programming interfaces (APIs) or command-line interfaces (CLIs) for dynamic configuration and turning of its behaviors. MLOps engineering teams use these tools to manage and tune model deployment and performance. They can also use the management APIs/CLIs to automate model deployment, monitoring, and validation as part of the CI/CD and MLOPs pipeline. Triton Model Analyzer and Model Navigator, TyTorch Management APIs are examples of a Serving platform management API/CLI.

- **Preprocessing and postprocessing:** Models are often trained with some preprocessing and postprocessing logic to prepare the input and to transform the prediction. These logics often need to be replicated in the inference platform, along with the model. The model serving platform must have the ability to incorporate preprocessing steps to, for example, augment or enrich the input by reaching out to a feature store, and postprocessing steps to transform to raw prediction output from the model for ease of consumption or any additional validation. These processing flows are sometimes called “Transformers,” as in KServe.

- **Inference APIs:** Inference APIs are interfaces provided by the serving platform to invoke a prediction, or sometimes to get an explanation of a prediction. Inference APIs are designed to be invoked directly via generic HTTP or gRPC endpoints, but they are not consumer-friendly API endpoints. Neither of these model serving frameworks should be directly accessible to a consumer, because they do not support functionality such as load balancing, throttling, SSL and authentication. The best practice is to package them in a container and use a mediated API pattern, as indicated in Figure 5. You will need to define a consumer-friendly API and then implement the resource and payload mediation logic in the model front-end component to interact with the private model services.

- **Model scaling:** More advanced model serving platforms provide inbuilt capabilities to manage scaling of model instances according to demand. The platform can scale up or down the number of model replicas based on metrics such as resource consumption, resource utilization and loads, with the ability to scale to and from zero. The serving platform effectively offers a serverless serving option to optimize cost and resource utilization.

- **Model monitoring and validation:** Continuous model monitoring, validation and explainability are integral parts of model inferencing. Model serving platforms provide these capabilities as inbuilt features or add-on components that integrate with third-party offerings.

- **Model composition and orchestration:** Some model serving platforms can deploy and manage multiple models in concert to implement more sophisticated predictions or analytical logic required by applications. Common model composition topologies supported include model chaining, model parallel execution, model ensemble and model aggregation. [2] Ray on Anyscale, KServer with ModelMesh and TorchServe Workflow are just some examples of serving platforms that have started to add model composition and orchestration capabilities to their offerings.

It is also possible to develop your own serving mechanism, using tools such as Flask, a simple application server framework for Python, to wrap a model and the relevant framework in a REST API. Fully evaluate the ongoing engineering efforts that will be involved in such an approach if you decide to go down this route.

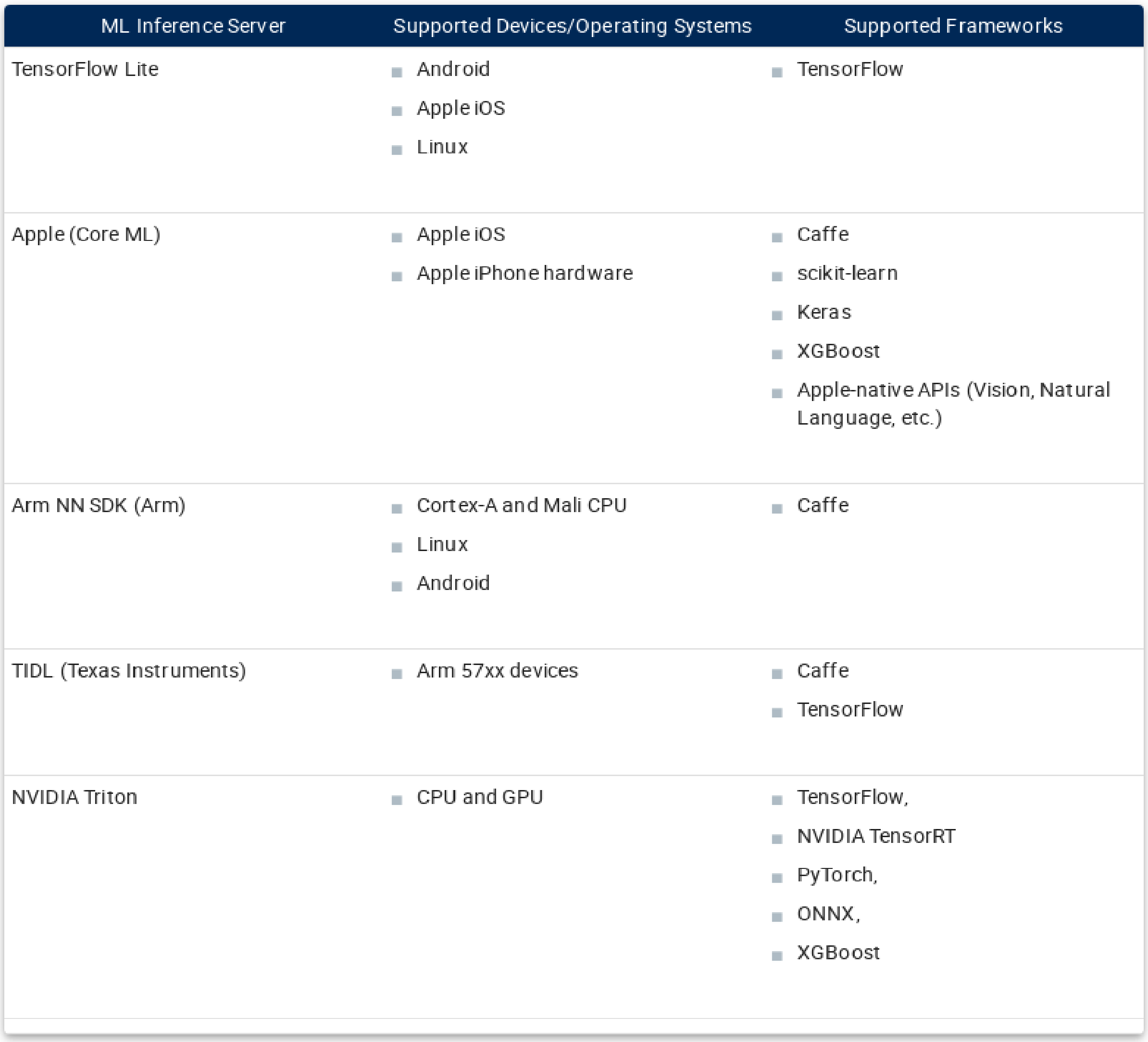
Refer to Mediated APIs: An Essential Application Architecture for Digital Business and Decision Point for Mediating API and Microservices Communication to learn more about API mediation patterns.

Mobile and IoT Device Model Serving

As with choosing a model serving framework, choosing the ML or AI model format you will use for mobile or edge devices will impact your decision on what inference server to use. However, your choices will be constrained by the model formats supported on the devices you are considering, as illustrated in Table 2.

Table 2: Mobile and IoT Device ML Inference Servers

(Enlarged table in Appendix)



ML Inference Server	Supported Devices/Operating Systems	Supported Frameworks
TensorFlow Lite	▪ Android ▪ Apple iOS ▪ Linux	▪ TensorFlow
Apple (Core ML)	▪ Apple iOS ▪ Apple iPhone hardware	▪ Caffe ▪ scikit-learn ▪ Keras ▪ XGBoost ▪ Apple-native APIs (Vision, Natural Language, etc.)
Arm NN SDK (Arm)	▪ Cortex-A and Mali CPU ▪ Linux ▪ Android	▪ Caffe
TIDL (Texas Instruments)	▪ Arm 57xx devices	▪ Caffe ▪ TensorFlow
NVIDIA Triton	▪ CPU and GPU	▪ TensorFlow, ▪ NVIDIA TensorRT ▪ PyTorch, ▪ ONNX, ▪ XGBoost

Source: Gartner (November 2022)

For a more detailed discussion on how to use ML for IoT devices, see Architecting Machine Learning With IoT and Create a Data Strategy to Ensure Success in Machine Learning Initiatives

Logging and Telemetry Data

Logging and telemetry data are an essential, but often overlooked, part of any modern application environment. ML models require additional information.

As we noted in Step 1, each model iteration must be assigned a unique identifier, often derived from the SHA-1 hash generated for the commit or revision in the Git repository used to manage model versioning. You may also wish to tag the model with a human-readable identifier, along with a model instance version. The identifier and instance version must be used in all logs and metrics related to the model, such that you can distinguish the shadow instances from the production instance.

These metrics break down into two areas:

- Standard operational metrics
- Model-specific metrics

Standard Operational Metrics and Monitoring

The infrastructure and software supporting the ML model must be monitored in the same way as you would any other application:

- **Health check endpoint:** A simple endpoint to confirm an instance of a model is still in production. A REST endpoint of metrics is common. If there is no response, an immediate alert must be raised.
- **Resources usage information:** Classic system-level metrics of CPU and/or GPU usage, memory footprint and I/O measurements. For example, a combination of high I/O and model latency (see the Model-Specific Metrics section below) may indicate a model that is being repeatedly loaded into memory.
- **Host data:** Metadata on the host, such as software versions and dependencies.

Model-Specific Metrics

The inference behavior of ML models is inherently probabilistic, but the training process for a model is often probabilistic, because training processes inject an element of randomness (using random seeds) to avoid overfitting in the training process. The probabilistic nature of training means your application design, testing and monitoring need to account for changes in behavior caused by seemingly subtle changes in the model. This includes simply retraining the model with exactly the same configuration and training set. The randomization will result in different outcomes — inference tests that were borderline on the certainty threshold may cross that threshold.

These changes in behavior are dependent on many factors, including:

- **Client behavior:** What is the usage pattern of the client that is consuming the model? For example, a spiky workload may result in additional latency for predictions, which may not be obvious when using traditional monitoring techniques.
- **Input data:** What data is submitted to the model for use in generating a prediction? For example, in an image recognition scenario, a client may change the type of image it is submitting to the model, which can significantly change the results. The answers are still correct, but a traditional monitoring approach may flag this change as an error.
- **Model training:** How a model is trained and optimized can have significant impacts on the predictions being produced. For example, retraining a model with new data may lead to a change in the distribution of predictions.

You need to gather two subsets of information: model execution and model accuracy. As mentioned in Step 1, each model must have a unique identifier, which must be reported in all logs and metrics.

Model Execution

Monitoring how efficiently your model is executing is part of your architectural design. Important factors to consider include:

- **Requests:** How many requests is this instance of the model serving?
- **Latency:** How long does it take for a request to complete? As well as reporting an average, it is also important to record the latency for each individual request to a logging system. You must analyze these logs for outliers and investigate the cause.
- **Scaling:** Do you queue execution for a single model when the model may not be thread safe? Do you deploy our model instances using fixed or dynamic resource pools? Can you add more model instances at a time to scale with load? Can you scale to zero when there is no traffic to release unused compute resources?

Model Accuracy

Monitoring for model accuracy depends on your agreement with your data science team when defining the nontechnical contract in Step 1. Important factors to consider include:

- **Prediction values:** The outputs for the model should be evaluated against the prediction distribution agreed on in Step 1. When results from the inference fall outside this distribution, an alert should be raised. Common model and data-specific metrics include characteristic stability index (CSI), population stability index (PSI) and covariate shift.
- **"Ground truth":** Regularly evaluate the model against random "ground truth" values. When a model produces an unexpected result, an alert must be raised for further investigation.
- **Model- and data-specific metrics:** Such as characteristic stability index (CSI), population stability index (PSI), covariate shift.
- **Model-only metrics:** Include RMSE, Gini coefficient and Kolmogorov-Smirnov.

Step 4: Deployment

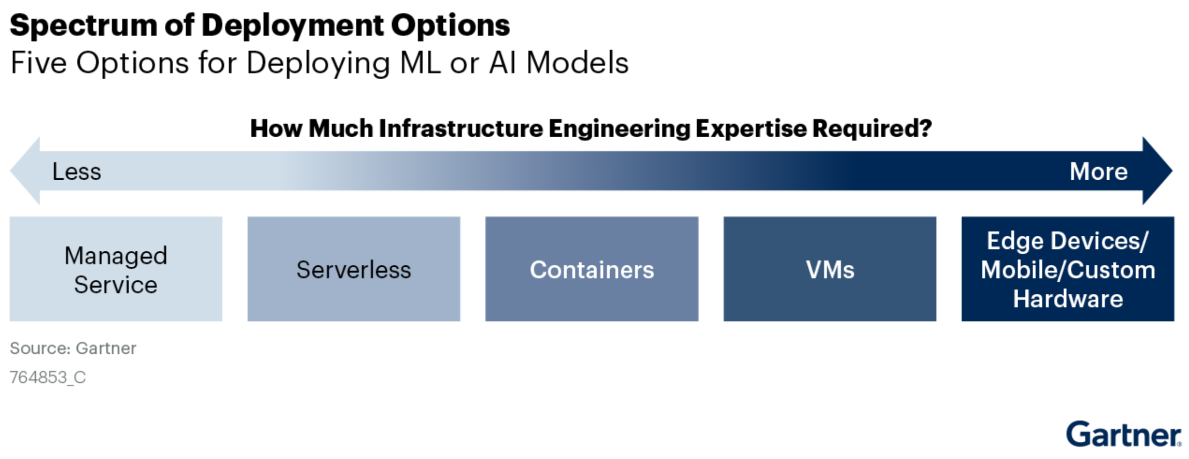
By the time you reach deployment, you will have established the format of your model, defined your packaging and ongoing testing approaches, established the key telemetry metrics and logging approach and chosen a model serving approach.

Containers are a natural fit to bring ML models into production. However, if your organization does not have experience using containers, it may be too much of a shift, and a VM-based infrastructure may be more appropriate.

Spectrum of Deployment Approaches

As with packaging, there is a spectrum of possible deployment options for your ML models, as illustrated in Figure 7.

Figure 7: Spectrum of Deployment Options


Spectrum of Deployment Options
Five Options for Deploying ML or AI Models
How Much Infrastructure Engineering Expertise Required?
Less
More
Managed Service
Serverless
Containers
VMs
Edge Devices/ Mobile/Custom Hardware
Source: Gartner
764853_C
Gartner

At one end of the spectrum, when the models are deployed on edge or mobile devices or custom hardware, application and platform engineering teams are responsible for some or all of the engineering and operation of the model inferencing infrastructure. At the other end, model deployment is carried out using managed services provided by the service or cloud provider, requiring little to no infrastructure expertise.

Notable characteristics of each deployment model are described below:

- **Managed service:** Models are deployed as managed services provided by the service provider.
  - **Pros:** There is no infrastructure to manage with tiered pricing models and access to accelerated CPU/GPU options.
  - **Cons:** Tightly coupled to a service provider's offering.

- **Serverless:** Models are deployed as serverless functions or applications (such as AWS SAM) without the burden of provisioning or managing the runtime infrastructure. This option is maybe cloud only, tightly coupled to a cloud provider's serverless offering. Alternatively, it can be built on a serverless framework, such as KNative, implemented on any Kuberentes environment.
  - **Pros:** There is no infrastructure to manage with a pay-as-you-go (PAYG) pricing model. You get the ability to scale to and from zero.
  - **Cons:** Cold starts can introduce latency due to initial model loading. Compute and memory limitations can constrain your model size and serving framework choices. ML models can be larger than a serverless environment can directly use; workarounds such as reading models from object storage may introduce additional latency.
- **Containers:** Models are deployed in containers managed by a container orchestration platform such as Kubernetes, running on-premises, in the cloud or in a hybrid environment.
  - **Pros:** Containers can be fully managed as code in a self-contained and modular approach for each iteration of a model, with individually customized scaling options.
  - **Cons:** The onramp for containers can be daunting. It demands significant maturity in your SDLC, as well as additional infrastructure (container registry, orchestration tools, etc.).
- **VMs:** Models are deployed in VMs, providing flexible deployment options on-premises, in the cloud or in a hybrid environment.
  - **Pros:** It is easy to access for most organizations, with a familiar approach and proven tools.
  - **Cons:** VMs can be more compute-resource-intensive, requiring greater operational support.

- **Edge devices/mobile/custom hardware:** Models are custom-packaged and optimized for execution in custom- or purpose-built hardware. Can deploy in the field, on-premises or in the cloud.
  - **Pros:** It can deliver the highest performance, or be highly customized for specialized use cases.
  - **Cons:** Additional engineering, development and testing effort required. It also requires the management of multiple tiers of infrastructure.

Kubernetes and ML/AI Workloads

> **If you do not have Kubernetes in your environment, do not introduce it specifically to serve ML models.**

If you already have a Kubernetes-based infrastructure available to you in your environment, it may be the best place to run your ML workloads, provided appropriate support structures are in place. Gartner recommends using a "platform ops" strategy when providing platforms such as Kubernetes. Please refer to Using Platform Ops to Scale and Accelerate DevOps Adoption for a more detailed discussion on this topic.

Platforms for managing the entire data science life cycle, such as Kubeflow and Polyaxon, have been created for Kubernetes. While these are possible solutions, you should be aware that these platforms are, in the main, more focused on the data science and training aspects of delivering ML models than on serving models in production. Refer to the serving examples discussed in Step 3 for alternative solutions.

Deployment Approaches

New models must be released into production in a controlled manner. As noted earlier, you must plan for models being updated (based on model KPIs or changes in business demands) and put appropriate automation in place.

Gartner observes that organizations commonly follow four patterns when deploying ML models into production:

- Canary release

- Blue/green deployments
- Challenger/champion deployment
- Feature toggles

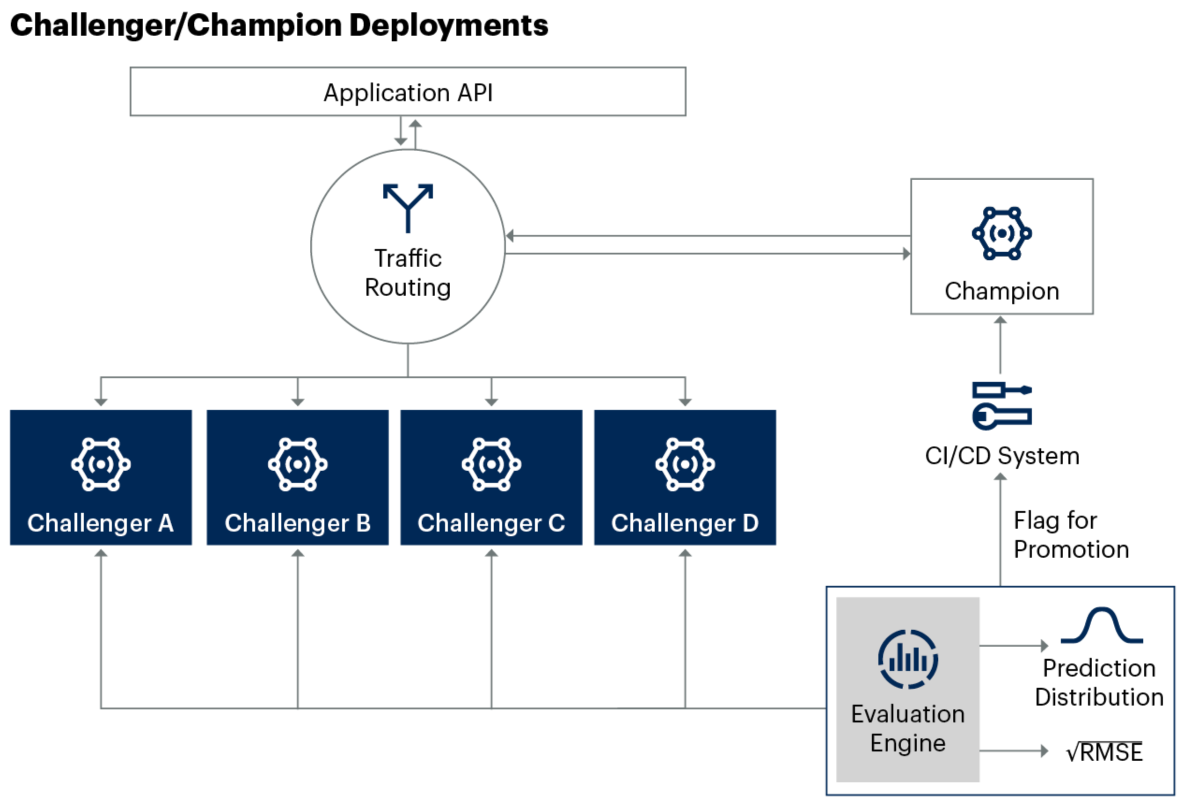
The first two patterns, canary release and blue/green deployment, are also common patterns for application deployment. A canary deployment involves incrementally introducing a feature into production, where requests are gradually routed to the feature (e.g., 10%, 20%, etc.) and the results monitored. Once there is confidence that the feature is stable, all traffic is routed to the new version. In a blue/green deployment, the relevant parts of the environment are replicated, updated and the traffic is switched over. If a problem is detected, you switch back to the original environment.

For more discussion on canary deployments and blue/green deployments, refer to Getting Started With Machine Learning Monitoring in Production.

Challenger/Champion Deployment

A challenger/champion deployment model is well-suited to the task of testing new ML models in a production environment. In this approach, multiple versions of a model (the "challengers") "shadow" the primary production model (the "champion"). The application routes requests to, and returns responses from, the champion model. A traffic router also routes the requests to one or more challenger models, and the results of each request are logged to a separate system to be evaluated. When a model has sustained higher accuracy, as measured using the probability distribution and RMSE values discussed in Step 1, the model is then flagged for promotion to production, as shown in Figure 8.

Figure 8: Challenger/Champion Deployments



Source: Gartner
764853_C

Gartner

This technique is commonly used in fraud detection applications. The behaviors of fraudsters and customers are constantly changing, and the best model for detecting this behavior also needs to continuously evolve. [3]

Several ML deployment platforms (such as DataRobot and AWs Sagemaker) support the implementation of challenger/champion deployment as a form of A/B testing on new ML models. [4]

Feature Flags

Feature flags (or "feature toggles") is a common DevOps technique that enables a development team to manage the incremental release of new functionality. This is achieved by coding a feature behind a toggle switch that is turned off by default. A feature remains dormant until the DevOps team is ready to activate it by flipping the feature toggle on, sometimes only in the preproduction environment to test the feature, or for a limited release to a specific user group. When the feature has been tested and determined to be stable, the feature toggle is typically removed for subsequent releases.

In the context of an ML application, there are a number of scenarios in which you can take advantage of feature flags to manage the safe release of new model functionality and roll it back if issues arise. For example, you may:

- Use feature flags to control the deployment of a shadow model alongside your live model, and gather performance metrics for the shadow model together with the live model.
- Encapsulate the ML model with corresponding pre- and postprocessing logic under a feature toggle within an inference runtime, such that certain complex ML feature engineering steps are executed only if the corresponding model is loaded.
- Use dynamic feature flags passed as parameters into an inference API to select the right ML model from a model ensemble.
- Use feature flags to facilitate multivariate A/B testing, canary releases and blue/green deployments.

Feature flags are power tools to enable new software features to be deployed and tested quickly and safely in a production environment. However, use feature flags with caution. Their successful use requires a mature feature life cycle management process that continues to evaluate the usefulness of every feature's flags and removes them once the feature is released and stable. [5]

Step 5: Monitoring and Feedback

Once your models are in production, they must be monitored as you would monitor any other component, with normal operational checks for items such as:

- **System availability:** Polling a service to ensure it is available, generally via a simple endpoint.
- **System-level statistics:** Resource utilization, such as memory and CPU usage.
- **Logs:** Recording error and output logs.

In addition to these metrics, you will also need to put in place monitoring and feedback loops, as discussed in Step 3, for:

- Model-specific operational performance

- Continuous testing and validation

All of these metrics must be available to the team responsible for maintaining the application and the data science team.

System Availability, Statistics and Logs

Use tried-and-tested approaches, such as synthetic transactions and healthcheck endpoints, to ensure your service is available. This approach works well when your model is made available as a service and consumed through an HTTP endpoint.

In streaming applications, for example, where a model is invoked as part of a Kafka Streams application, you may not have an endpoint available. In this case, use runtime monitoring interfaces, such as JMX for Java implementations, to expose basic stats.

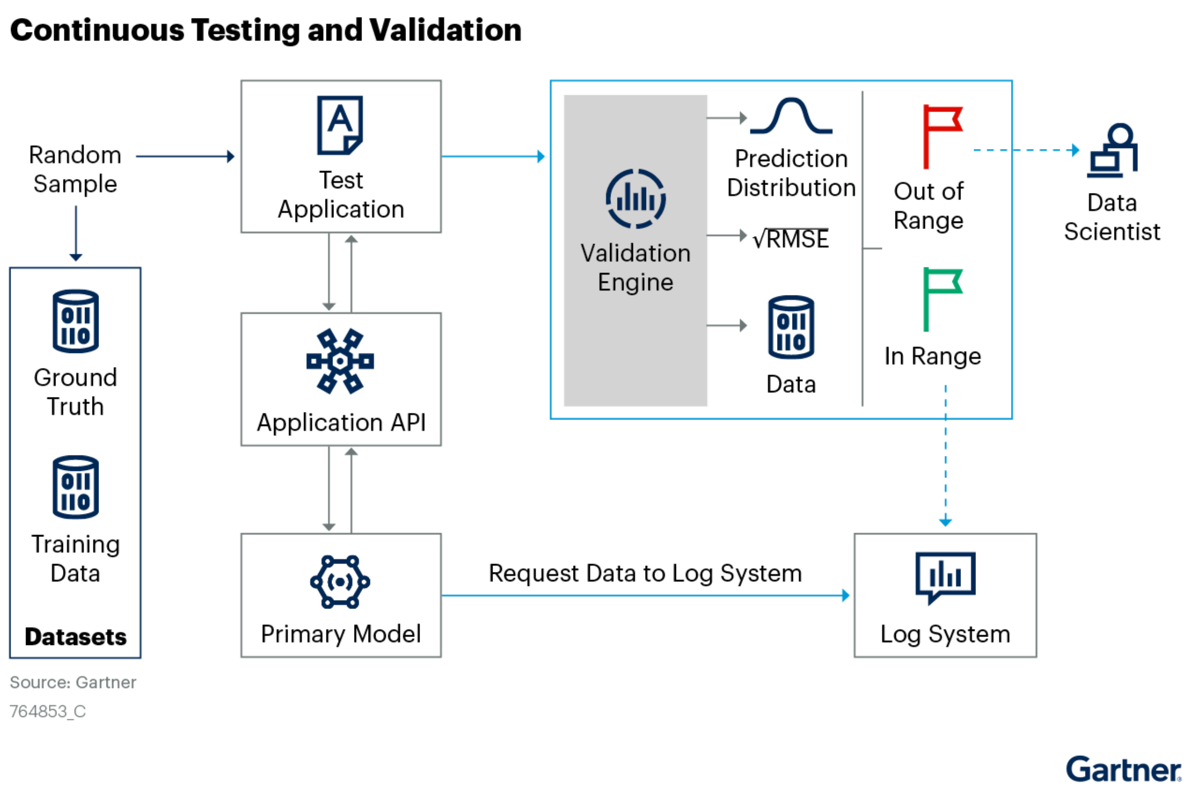
Gather system-level statistics for CPU, GPU, memory and other system-level metrics at regular intervals. Logs from your application, including any requests made to the application and errors that may have occurred, should be sent to your logging system. Your logging analysis should enable you to correlate request information with system-level statistics in support of basic analysis.

Continuous Testing and Validation

A model in production, and any models that are challengers, must be tested on an ongoing basis against the key criteria identified in Step 1. Given the probabilistic nature of ML inference, your results will vary over the lifetime of any individual model or set of models.

For example, you will want to randomly sample data from your ground truth and training datasets and run inference checks against your model. You should then verify these results against your prediction distribution and RMSE values identified with the data science team, as indicated in Figure 9.

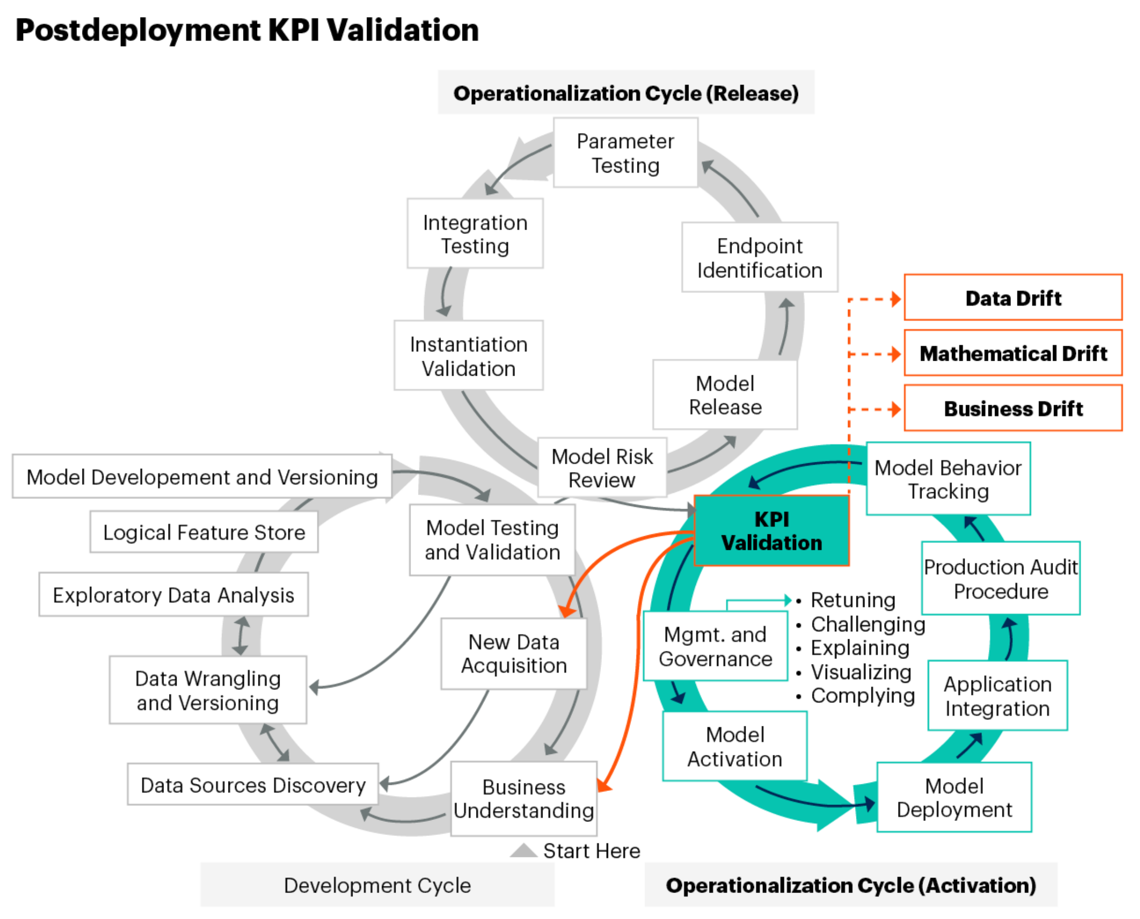
Figure 9: Continuous Testing and Validation



Postdeployment Model KPI Validation

Model performance should be measured and validated against business and technical KPIs throughout the model's operationalization cycle, as shown in Figure 10

Figure 10: Postdeployment KPI Validation



Source: Gartner
764853_C

- **Business drift**: This may include changes in business operation conditions and processes, or shifts in business understanding and concepts that were set forth when the project was originally conceived.
  - If the model fails or degrades due to business drift, then go back to the beginning of the development cycle through the Business Understanding step to reevaluate the original business assumptions.

- **Mathematical drift (or model drift):** Your model is trained to approximate the mathematical function that captures the relationships or patterns inherent in the training data. However, when real-world data deviates from the training set in such a way that affects the underlying relationship and patterns the model learns, we call it "mathematical drift" or "model drift." This can result in your trained model degrading in performance or precision slowly over time (model decay) or becoming entirely ineffective (model stale) in the new problem space, and needing to be retrained.
  - If the model fails or decays due to mathematical drift, then go back to the Exploratory Data Analysis step in the development cycle.
- **Data drift:** This accounts for any shift from the original training dataset to the actual data seen in production that may render the model ineffective. Examples of data drift include changes in data format or structure, data range and distribution. When the drift occurs with the input features, it is called "feature drift." When the drift occurs on the output side, it is called "label drift."
  - If the model fails or decays due to data drift. then return to the New Data Acquisition step in the development process.

For a full description of KPI validation, see Use Gartner's MLOps Framework to Operationalize Machine Learning Projects.

Model-Specific Operational Metrics

As discussed in Step 3, you will have chosen the metrics to track related to the model. These metrics should be measured in two ways:

- **Request details:** The frequency and, in the case of a model delivered as a service, the origin of requests.
- **Latency:** How long does it take for a model to return a result from the moment a request is made to its being returned? How you approach this will vary depending on the integration point. The most common pattern is to measure the start and end of a request in your API mediation layer.

Risks and Pitfalls

Building ML models into applications is new to many development teams. Technical professionals responsible for incorporating ML models into their application architecture should be aware of the following risks and pitfalls with this approach.

Falling Into the “Science Project” Trap

Organizations are keen to bring ML to their applications. Many organizations create special project teams or hire external specialists specifically for this task, but they lack a coherent strategy and tactical planning. This approach frequently leads to one-off, nonrepeatable models that have no place in a production application.

To avoid this pitfall, carefully examine the project for the four key warning signs of an ML science project:

- No reproducible pipeline
- No data pipeline
- Widely available functionality
- Complex framework used for a simple problem.

Lack of Explainability and Fairness

The inherent black-box nature of ML models limits our ability to analyze and understand their predictive behaviors and the extent of their applicability and accuracy to a business problem. When an ML-powered service makes mistakes, it can expose an organization to a variety of risks, such as data risks, ethical, legal and regulatory implications, erosion of trust and damage to brand reputation.

To mitigate these risks, organizations should establish a governance framework for all ML-based solutions. This should include implementing fairness and explainability functions by engaging risk, quality assurance, functional and technical teams throughout the model requirements, development, testing and implementation life cycle. See Incorporate Explainability and Fairness Within the AI Platform for detailed guidance.

Insufficient Logging, Monitoring and Feedback

Application architects are asked to quickly bring ML models into their applications once a data science team has developed a model for the business. In their haste, architects may introduce a model into an application without any of the necessary additional logging, monitoring and feedback loops.

To mitigate this risk, ensure that each version of the model has a unique identifier as discussed in Step 1, and that this unique identifier is propagated through all logging, monitoring and feedback systems. You must also ensure a base level of testing is in place to detect model drift, and the testing data must be made available to the data science team.

Underestimating the Costs of ML Application Delivery

ML-based systems and applications require significantly more time, effort and resources to develop and operate than traditional software development. ML model development relies on large amounts of high-quality data and an effective data processing and training pipeline, because data is the source code of ML models. ML applications are also more challenging to monitor, test and validate, operate and debug than traditional applications. There are many areas and situations (intended or unintended), such as staffing, skill uplift, engineering, operation and computation costs, that can lead to excessive cost overrun if not managed properly.

To avoid this pitfall, organizations should gain an understanding and appreciation of the technical and business challenges of bringing ML applications to production, ensuring that ML is the appropriate approach to the problem at hand. Joint ML and application development teams should walk through the ML development and operation life cycle. At every step, they should account for, manage and optimize the cost of ML application delivery. Some areas of focus include:

- Staffing
- Infrastructure and engineering costs associated with data acquisition
- Data quality and labeling
- Model training and inferencing
- Model evaluation and monitoring for explainability, bias and drifts.

Failing to Track Model Inference Response Time

Just providing a result from a model when a request is made is not enough for real-world applications. The result must be made available in a timely manner. Customers are sensitive to latency in applications, and your model cannot become a bottleneck for the overall application.

To mitigate this risk, track the inference time for all requests and immediately flag any requests that are outside the acceptable parameters. It is essential that this is monitored on an ongoing basis, because changes to a model may radically impact inference time. Also consider simpler alternative solutions that do not incur expensive or resource-intensive inference execution.

Related Guidance

This guidance framework applies to bringing ML models into an application, which is only part of a larger set of activities related to successfully developing an ML or AI model or selecting an appropriate service. Technical professionals responsible for bringing ML models into production applications will find the following research relevant:

- Comparing Platforms and Capabilities for Data Science and AI
- Achieve DSML Value by Aligning Diverse Roles in an MLOps Framework
- Magic Quadrant for Data Science and Machine Learning Platforms
- Critical Capabilities for Data Science and Machine Learning Platforms
- Toolkit: RFP for Data Science and Machine Learning Platforms
- Market Guide for Multipersona Data Science and Machine Learning Platforms
- Essential Skills for Machine Learning Architects
- Essential Skills for Citizen Data Scientists
- Essential Skills for AI Engineers
- Essential Skills for Machine Learning Architects
- Machine Learning Playbook for Data and Analytics Professionals
- Demystifying XOps: DataOps, MLOps, ModelOps, AIOps and Platform Ops for AI
- MASA: Create Agile Application Architecture With Apps, APIs and Services
- Solution Path for Applying Microservices Architecture Principles
- Assessing Patterns for Deploying Distributed Kubernetes Clusters
- Emerging Technologies: Critical Insights on Edge Container Management

- Keys to DevOps Success
- How to Evaluate API Management Solutions

Evidence

[1] TensorFlow Model Optimization,TensorFlow.

[2] Ensemble Models: What Are They and When Should You Use Them?, Built In.

[3] Champion-Challenger Analysis for Credit Card Fraud Detection: Hybrid Ensemble and Deep Learning, ScienceDirect.

[4] Dynamic A/B Testing for Machine Learning Models With Amazon SageMaker MLOps Projects, Amazon.

[5] Feature Toggles (aka Feature Flags), martinfowler.com.

Document Revision History

How to Build Machine Learning and Artificial Intelligence Into Production Applications - 1 October 2019

Recommended by the Authors

Some documents may not be available as part of your current Gartner subscription.

A Guidance Framework for Operationalizing Machine Learning

Preparing and Architecting for Machine Learning: 2018 Update

Integrating Machine Learning Into Your Application Architecture

A Guidance Framework for Continuous Integration: The Continuous Delivery 'Heartbeat'

How to Architect Continuous Delivery Pipelines for Cloud-Native Applications

How to Succeed With Microservices Architecture Using DevOps Practices

Solution Path for Applying Microservices Architecture Principles

Artificial Intelligence Maturity Model

Innovation Insight for Federated Machine Learning

Use Gartner's MLOps Framework to Operationalize Machine Learning Projects

2023 Planning Guide for Analytics and Artificial Intelligence

Demystifying XOps: DataOps, MLOps, ModelOps, AIOps and Platform Ops for AI

Table 1: API-Based ML Services

Model Type	Usage	Example Services
Sentiment analysis	Processing of text to identify positive, negative or neutral sentiment	▪ Amazon Comprehend ▪ Google Natural Language ▪ Watson Natural Language Understanding
Vision	Extract specific aspects, such as facial features, text or object recognition from a video or image	▪ Amazon Rekognition ▪ Microsoft Azure (Computer Vision) ▪ Tencent OCR and Video Content Analysis
Natural Language Understanding	Include speech in an application, such as in a conversational interface like Amazon Alexa or Google Home	▪ Amazon Lex/ Amazon Translate/Amazon Comprehend ▪ Microsoft Azure Speech Services ▪ Google (Cloud Translation)
Personalization	Create personalized experience for consumers, such as with specific offers in e-commerce scenarios	▪ Amazon Personalize ▪ Microsoft Azure (Personalizer) ▪ Google Recommendations AI

Source: Gartner (November 2022)

Table 2: Mobile and IoT Device ML Inference Servers

ML Inference Server	Supported Devices/Operating Systems	Supported Frameworks
TensorFlow Lite	■ Android ■ Apple iOS ■ Linux	■ TensorFlow
Apple (Core ML)	■ Apple iOS ■ Apple iPhone hardware	■ Caffe ■ scikit-learn ■ Keras ■ XGBoost ■ Apple-native APIs (Vision, Natural Language, etc.)
Arm NN SDK (Arm)	■ Cortex-A and Mali CPU ■ Linux ■ Android	■ Caffe
TIDL (Texas Instruments)	■ Arm 57xx devices	■ Caffe

		■ TensorFlow
NVIDIA Triton	■ CPU and GPU	■ TensorFlow, ■ NVIDIA TensorRT ■ PyTorch, ■ ONNX, ■ XGBoost

Source: Gartner (November 2022)