MICROSOFT AZURE

AI-300 Study Guide

Operationalizing Machine Learning and Generative AI Solutions

A concise revision guide for MLOps and GenAIOps engineers

Prepared by Jitendra Singh Malik

September 2026

Independent study resource

These notes were created while preparing for the AI-300 exam. They are not official Microsoft exam material. Features, previews, licensing and user-interface details may change, so validate current behaviour using Microsoft Learn and product documentation.

How to use this guide

This guide condenses AI-300's five skills-measured domains - spanning traditional MLOps and generative-AI GenAIOps - into a structured reference, grounded in Microsoft's official study guide and service documentation.

- Read chapters 1-2 for the MLOps half (Azure Machine Learning infrastructure and model lifecycle), then chapters 3-5 for the GenAIOps half (Foundry infrastructure, quality/safety, and optimization).
- Use the code-identifier callouts (in monospace) to memorize exact CLI commands, YAML fields, and function/class names.
- Review the "Common confusion" callouts before practice exams - they target the exact distinctions exam scenarios test.
- Use the final quick-reference chapter as a last-day revision sheet.

Contents

1. Design and Implement an MLOps Infrastructure (15-20%)
2. Implement Machine Learning Model Lifecycle and Operations (25-30%)
3. Design and Implement a GenAIOps Infrastructure (20-25%)
4. Implement Generative AI Quality Assurance and Observability (10-15%)
5. Optimize Generative AI Systems and Model Performance (10-15%)

Source note

This guide is based on FabricPrep's AI-300 deep-dive study guide pages, researched directly from Microsoft Learn's official AI-300 study guide, its two training learning paths, and underlying Azure Machine Learning / Microsoft Foundry documentation.

CHAPTER 1

Design and Implement an MLOps Infrastructure

Provision Azure Machine Learning workspaces, datastores, and compute, then automate that provisioning with Bicep, the Azure CLI, and GitHub Actions.

Workspace resources: datastores, compute targets, and identity

The Azure Machine Learning workspace is the top-level container every job, model, endpoint, and asset lives inside — provisioning it correctly is the first MLOps decision, since several of its settings (identity type, associated resources) can't be changed after creation.

Datastores

- A datastore is a saved reference to an existing storage account (blob container, ADLS Gen2 container, file share) — it stores the connection information, not the data itself, so registering a datastore doesn't copy anything.
- Every workspace gets a default datastore backed by its associated storage account automatically; additional datastores are registered explicitly to point at other storage, including storage owned by another team or subscription.
- Datastores authenticate either with a stored credential (account key or SAS token) or, the preferred production pattern, with the workspace's managed identity via Microsoft Entra ID — removing a secret that would otherwise need rotation.

Compute targets

- Compute instance: a single-user development VM for notebooks and interactive testing — not meant to run unattended production jobs.
- Compute cluster (`AmlCompute`): an autoscaling pool of VMs for training jobs, scaling between a configured `minNodeCount` and `maxNodeCount`; setting `minNodeCount: 0` means the cluster costs nothing while idle, at the cost of a cold-start delay on the next job.
- Kubernetes compute (attaching an existing AKS cluster) and serverless compute (Azure Machine Learning manages the compute lifecycle entirely, with no cluster to size or manage) are the other two options, most relevant for teams standardizing on Kubernetes or wanting to skip capacity planning altogether.

Identity and access for the workspace itself

- The workspace has its own system-assigned or user-assigned managed identity, used for the workspace to reach its associated Key Vault, storage account, and container registry without embedded credentials.
- RBAC roles assigned *on* the workspace (Contributor, the built-in AzureML Data Scientist, or a custom role) control who can submit jobs, register models, and manage compute — separate from the identity the workspace itself uses to reach its dependencies.

> **Common confusion**
> A datastore's own authentication (how the *workspace* reaches storage) and a user's RBAC role on the workspace (what *that person* is allowed to do) are independent layers — a user with Contributor on the workspace can still be blocked from the underlying storage account if its network rules or access policies don't separately permit it.

Workspace assets: data assets, environments, components, and registries

Assets are the versioned building blocks a job references — the point of registering something as an asset instead of using a raw path is that every job that consumed a specific version stays reproducible even after the underlying file changes.

Data assets

- A data asset wraps a path (`uri_file`, `uri_folder`, or `mltable`) with a name and version, so a training job can reference `azureml:sales-features:3` instead of a mutable path that might point at different content next week.
- `mltable` additionally captures schema and read/transform logic (column types, a delimiter, which files to include from a folder) as part of the asset definition, not just a location.

Environments

- An environment pins the Python packages, base Docker image, and any environment variables a job's code runs against — defined from a conda YAML plus a base image, or from a fully custom Dockerfile.
- Curated environments (Microsoft-maintained, prefixed `AzureML-`) cover common frameworks out of the box; a custom environment is built once and then reused by every job that needs the same dependency set, keeping training and deployment consistent.
- Environments are versioned automatically on rebuild, so a job specification like `environment: azureml:my-training-env@latest` always resolves to the most recently built image without editing every job definition.

Components

- A component packages a single reusable pipeline step — its inputs, outputs, code, environment, and command — so the same "train a model" or "validate data" step can be dropped into multiple pipelines without copy-pasting its definition.

Registries: sharing assets across workspaces

- By default, models, environments, components, and data assets live in one workspace and aren't visible from another. A registry is a separate, workspace-independent container for exactly these asset types, replicated across whichever regions you configure.
- A central platform team publishes a validated environment or a training component to a registry once; every team's workspace then references it as `azureml://registries/<registry-name>/environments/<name>/versions/<version>`, guaranteeing every workspace trains against the identical, approved definition instead of copies that can drift.

> **Common confusion**
> A workspace's default asset store and a registry solve different problems: the workspace store versions assets *for that workspace's own jobs*; a registry exists specifically so multiple workspaces can share the same asset without each team re-registering and maintaining its own copy.

Infrastructure as code: Bicep and the Azure CLI

Standing up an Azure Machine Learning workspace by hand in the portal doesn't scale past a single team, and it leaves no record of exactly how the environment was configured — Bicep and the Azure CLI make that provisioning declarative and repeatable.

Bicep for the workspace and its dependencies

- The workspace itself deploys as a `Microsoft.MachineLearningServices/workspaces` resource, but it also requires an associated storage account, Key Vault, and (for full telemetry) an Application Insights resource — a workspace Bicep template typically provisions all four together, wiring the workspace's `properties` to their resource IDs.
- Compute can be declared inline on the workspace resource's `computes` array (creating a new `AmlCompute` cluster with a `scaleSettings` block for `minNodeCount`/`maxNodeCount`) or attached separately as its own resource — both approaches are idempotent, so re-running the same deployment converges to the same state instead of duplicating resources.
- Setting the workspace identity block (`systemAssignedIdentity: true` or a specific user-assigned identity) at deployment time is what lets the workspace authenticate to its dependencies without a stored secret from the moment it's created.

Azure CLI for day-to-day and scripted operations

- The `az ml` extension (CLI v2) drives everything after the workspace exists: `az ml workspace create`, `az ml datastore create -f datastore.yml`, `az ml compute create -f cluster.yml`, and so on, each backed by a YAML file matching that resource's schema.
- CLI commands are the natural fit for a GitHub Actions step, since they're scriptable and produce predictable exit codes — a workflow step failing on a bad `az ml job create` is easy to gate a pipeline on, in a way a manual portal click isn't.

> **Common confusion**
> Bicep provisions the *infrastructure* (the workspace, its compute, its network configuration) — it doesn't submit training jobs or register models. Those day-to-day MLOps operations run through the CLI or SDK against an already-provisioned workspace, which is why real pipelines combine both: Bicep for the one-time (or infrequently changed) environment, CLI/SDK for the jobs that run against it continuously.

GitHub Actions, workload identity federation, and network restriction

Automating provisioning and training through GitHub Actions only pays off if the authentication behind it is secure and the workspace's network posture doesn't quietly undermine everything else.

Authenticating GitHub Actions to Azure without long-lived secrets

- The recommended pattern is OpenID Connect (OIDC) via the `azure/login` action: a Microsoft Entra application (or a user-assigned managed identity) is configured with a federated identity credential that trusts tokens GitHub issues for a specific repository, branch, or environment.
- With OIDC configured, the workflow authenticates using only a client ID, tenant ID, and subscription ID — no client secret is stored in GitHub at all, which removes an entire class of leaked-secret risk compared to a classic service-principal-with-secret setup.
- Role assignments (typically `Contributor` or a narrower custom role, scoped to the resource group holding the workspace) are still made against that same Entra application or managed identity — OIDC changes *how* the workflow authenticates, not *what* it's authorized to do once authenticated.

Trunk-based development and branch protection

- Short-lived feature branches merged frequently into a protected `main`/`trunk` branch, gated by required PR reviews and passing CI checks, keeps the workspace's actual state (what Bicep and job YAML define) from drifting far from what's deployed.

- A CI workflow typically lints and validates job/pipeline YAML on every pull request, and only a merge to the protected branch triggers the workflow that actually deploys infrastructure or submits a production training job.

Restricting network access

- A workspace can require private endpoint access only (`public_network_access: Disabled`), forcing all traffic — studio, SDK, CLI — through a private link inside a virtual network rather than the public internet.
- Once network-restricted, GitHub-hosted runners (which run outside your virtual network) can no longer reach the workspace directly — this is a common reason a working local `az ml` command fails identically from a GitHub Actions workflow, and is solved with a self-hosted runner placed inside the VNet, or a hosted-runner network peering/tunnel solution.

> **Common confusion**
> Workload identity federation secures *who* can trigger a deployment or job; network restriction secures *where the traffic can come from*. Configuring OIDC correctly doesn't help a workflow reach a network-isolated workspace from a public GitHub-hosted runner — the two controls have to be solved together, not interchangeably.

CHAPTER 2

Implement Machine Learning Model Lifecycle and Operations

Orchestrate training with MLflow, AutoML, and sweep jobs, then register, deploy with safe rollout, and monitor models in production.

Orchestrating training: MLflow tracking and command jobs

Azure Machine Learning uses MLflow as its native experiment-tracking API — logging with the open-source MLflow SDK inside a job automatically lands metrics, parameters, and artifacts in the workspace, with no Azure-specific logging code required.

MLflow tracking inside a job

- `mlflow.set_tracking_uri(...)` (or simply running inside an Azure Machine Learning job, where it's configured automatically) points the MLflow client at the workspace instead of a local or external tracking server.
- `mlflow.autolog()` instruments common frameworks (scikit-learn, PyTorch, LightGBM, and others) to log parameters, metrics, and the trained model automatically, without explicit `log_metric`/`log_param` calls scattered through training code.
- For anything autolog doesn't capture, `mlflow.log_metric(name, value)`, `mlflow.log_param(name, value)`, and `mlflow.log_artifact(path)` log explicitly inside a run — and a sweep job's primary metric (covered next) must be logged with exactly this API for the sweep controller to read it.

From notebook to command job

- An exploratory notebook doesn't scale to repeatable, scheduled, or CI-triggered training — the production pattern converts that logic into a standalone Python script accepting arguments (via `argparse` or similar), then runs it as a `command` job.
- A command job YAML specifies the `code` directory, the `command` to execute (for example `python train.py --learning_rate ${{inputs.learning_rate}}`), the `environment`, and the `compute` target — the same script then runs identically whether launched from the CLI, a pipeline step, or a GitHub Actions workflow.
- Job inputs/outputs use the `${{inputs.<name>}}` / `${{outputs.<name>}}` binding syntax, letting a job's parameters (a learning rate, a registered data asset) be swapped without touching the script itself.

> **Common confusion**
> `mlflow.autolog()` captures what a specific framework integration knows how to log automatically — it won't pick up a custom metric your code computes that isn't part of that framework's standard training loop; anything domain-specific still needs an explicit `mlflow.log_metric()` call.

AutoML and hyperparameter sweep jobs

Sweeping and AutoML solve related but distinct problems: AutoML searches over *which model and featurization* to use, while a sweep job searches over *hyperparameter values* for a training script you've already written.

Automated ML

- An AutoML job takes a task type (classification, regression, forecasting, or an image/NLP task), a target column, and training data, then trials multiple algorithms and featurization strategies automatically, ranking candidates by the configured primary metric.
- The Responsible AI dashboard can be generated for an AutoML (or any MLflow) model afterward, surfacing fairness, explainability, and error-analysis views for the trained model — this is a separate step from training, not something that runs automatically inside every job.

Sweep jobs: sampling algorithms

- `random`: draws values uniformly from the search space (or via the Sobol quasi-random sequence with `rule: sobol` for better space coverage and reproducibility); supports early termination and is a common starting point.
- `grid`: exhaustively tries every combination of `choice`-type hyperparameters — accurate but expensive, and unusable with continuous distributions.
- `bayesian`: picks each new trial's values based on the results of previous trials, converging on promising regions faster than random search, but is incompatible with early termination policies since it needs completed trials to inform the next one.

Early termination policies

- `BanditPolicy`: cancels a trial if its primary metric falls outside a `slack_factor` (relative) or `slack_amount` (absolute) distance from the best trial so far, checked every `evaluation_interval`.
- `MedianStoppingPolicy`: cancels a trial if its performance is worse than the median of all trials at the same point — a conservative choice, commonly cited as saving 25-35% of compute with minimal impact on the final result.
- `TruncationSelectionPolicy`: cancels the worst `truncation_percentage` of running trials at each evaluation interval — more aggressive than Bandit or Median stopping.
- The sweep job's `objective.primary_metric` must exactly match the name used in the script's `mlflow.log_metric()` call, and `objective.goal` (`maximize` or `minimize`) tells the sweep controller which direction is "better."

> **Common confusion**
> Grid and random sampling both support early termination; Bayesian sampling does not, because each new trial's parameter choices depend on the full results of prior trials rather than being independent — cutting a trial short would corrupt the very history the algorithm relies on.

Training pipelines and distributed training

A single command job is fine for one step; anything with multiple stages (prep, train, evaluate) or that needs to scale across many machines calls for a pipeline or a distributed training configuration instead.

Pipelines built from components

- A pipeline job wires multiple components together, binding one component's output to the next component's input (`${{parent.jobs.prep_step.outputs.clean_data}}`) so Azure Machine Learning can resolve the dependency graph and run independent steps in parallel automatically.
- Because each step is a versioned component, the exact same "featurize" step can be reused across a training pipeline and a batch-scoring pipeline without duplicating its definition — and a pipeline run's lineage shows exactly which component version produced which output.

- Pipelines are the unit that gets scheduled (covered in the next section) for recurring retraining, and the unit registered as an endpoint for batch inference.

Distributed training

- For a single command job that needs multiple nodes or multiple GPUs, the job's `distribution` block specifies a framework: `PyTorch` (`process_count_per_instance` plus the job's `instance_count`), `MPI`, or `TensorFlow` (with separate worker and parameter-server counts).
- Azure Machine Learning sets up the distributed environment variables (rank, world size, master address) automatically based on this configuration — the training script only needs to read them through the framework's own APIs (for example, PyTorch's `torch.distributed`), not hand-roll cluster coordination.
- Distributed training is a scaling decision independent of hyperparameter tuning — a sweep job's individual trials can themselves each be distributed multi-node jobs, for large models where even a single trial doesn't fit on one machine.

> **Common confusion**
> A pipeline's steps run as independent jobs connected by data dependencies (parallelizing *stages* or independent branches); distributed training splits *one* training job's compute across multiple nodes or GPUs working on the same task together. Needing both is common — the "train" step of a pipeline can itself be a distributed job — but they solve different scaling problems.

Model registration and versioning with MLflow

Registering a model is what turns a run's output into a durable, versioned asset other steps — deployment, another team's pipeline — can reference by name, independent of whether the original run or its compute still exists.

Registering from a run

- `mlflow.register_model(f"runs:/{run_id}/{artifact_path}", model_name)` registers a model MLflow logged inside a run, using the MLflow `runs:/` URI scheme — this preserves lineage back to the exact run and its logged parameters/metrics.
- The Azure CLI equivalent, `az ml model create --name my-model --version 1 --path runs:/<run-id>/model/ --type mlflow_model`, or the Python SDK's `Model(path="runs:/<run-id>/model/", type=AssetTypes.MLFLOW_MODEL)` followed by `ml_client.models.create_or_update(...)`, does the same thing outside the MLflow API directly.
- The `azureml://jobs/<job-name>/outputs/<output-name>/paths/<path>` URI form registers a model from any named output of a job, useful when a model wasn't logged via MLflow directly inside the run but still needs lineage back to the job that produced it.

Versioning behavior

- Registering under a model name that doesn't exist yet creates version 1; registering again under the same name creates version 2, and so on — versions are immutable once created, so "updating" a registered model always means adding a new version, never overwriting an old one.
- Consumers reference either a pinned version (`azureml:my-model:3`) for reproducibility, or `azureml:my-model@latest` to always resolve to whatever was registered most recently — the same latest-alias pattern used for environments.

Responsible AI evaluation and lifecycle

- A Responsible AI dashboard (fairness, explainability, error analysis, causal analysis components) can be generated against a registered MLflow model, typically as a step between registration and production deployment, so a model with a known fairness or explainability problem is caught before it serves traffic.
- Archiving a model version (rather than deleting it) keeps it visible in history and still resolvable by exact version for auditing or rollback, while removing it from "latest" resolution and default listings — the lifecycle equivalent of deprecating without destroying evidence of what was once in production.

> **Common confusion**
> `mlflow.register_model` and `mlflow.<flavor>.log_model(..., registered_model_name=...)` both end up registering a model, but only within the *same workspace where the run itself was tracked* — Azure Machine Learning doesn't support registering a model into a different workspace's registry directly from an MLflow call; cross-workspace sharing goes through a registry (see the MLOps infrastructure topic), not through MLflow's registration API.

Production deployment: managed endpoints and safe rollout

A managed online endpoint is a stable, versioned HTTPS URL; the actual serving code and model live in one or more *deployments* underneath it — separating the stable address from the thing currently answering requests is what makes safe rollout possible.

Endpoints and deployments

- Creating an endpoint alone doesn't serve traffic — at least one deployment (model, environment, scoring script or MLflow model, and instance count/size) must exist and receive traffic allocation before it responds to requests.
- `traffic` on the endpoint is a percentage map across deployments (`blue: 90 green: 10`) that must sum to 100 (or 0, to disable all traffic) — this is the mechanism behind blue/green deployment: create `green` with new code or a new model version, give it 0% traffic, validate it in isolation, then shift the split gradually.
- A request can bypass the traffic split entirely by setting the `azureml-model-deployment` HTTP header to a specific deployment name — useful for direct testing of a 0%-traffic deployment without touching the live split.

Traffic mirroring (shadow testing)

- `mirror_traffic` copies a percentage of *live* traffic to a second deployment without returning its results to the caller — the caller still only ever sees the primary deployment's response, but the mirrored deployment's logs and metrics can be inspected for latency or error-rate problems under real production load.
- Mirroring is capped at 50% of traffic, applies to only one deployment at a time, and isn't available for Kubernetes online endpoints (managed online endpoints only) — `az ml online-endpoint update --name $ENDPOINT_NAME --mirror-traffic "green=10"` mirrors 10% to `green` while `blue` continues serving 100% of live responses.

Rollback

- Because traffic allocation is just a percentage map, rollback is the same mechanism run in reverse: set the previous deployment back to 100% and the failing one to 0% — no redeploy is required, since both deployments are still running side by side until one is explicitly deleted.

> **Common confusion**

Mirrored traffic and a 10%-live-traffic split look similar operationally but behave very differently for the caller: a live split actually returns the new deployment's predictions to whichever fraction of callers land on it, while mirroring never changes what any caller sees — it exists purely to observe a new deployment's behavior under real traffic before it's trusted with a live split at all.

Monitoring and retraining production models

Deploying a model isn't the end of the lifecycle — its input data and its own predictions can silently drift from what it was trained and validated on, and monitoring is what turns that into something actionable instead of something discovered from a customer complaint.

How monitoring works

- Monitoring compares a *production* data window against a *reference* (baseline) data window — usually the training data, or a recent past production window — using a statistical test or distance metric per signal.
- Production inference data collection has to be explicitly enabled on the online deployment first; without it, there's no production data for a monitor to compare against at all.

Built-in monitoring signals

- Data drift: compares the distribution of input feature values in production against the reference window, using metrics like `jensen_shannon_distance`, `population_stability_index`, `normalized_wasserstein_distance` (numerical features), or `chi_squared_test` (categorical features).
- Prediction drift: the same style of comparison, but applied to the model's *output* distribution rather than its inputs — a model can have stable inputs but drifting predictions if the relationship it learned no longer holds.
- Data quality: flags problems in the incoming data itself independent of drift — `null_value_rate`, `data_type_error_rate`, `out_of_bounds_rate` catch pipeline problems upstream of the model, like a feature that started arriving as a string instead of a number.

Scheduling and alerting

- `az ml schedule create -f monitor.yaml` runs a monitoring job on a recurring schedule against a Spark compute instance; the YAML's `monitoring_target.endpoint_deployment_id` (format `azureml:<endpoint>:<deployment>`) ties the monitor to a specific live deployment.
- Each signal's `metric_thresholds` defines when a metric counts as an anomaly; `alert_enabled: true` sends an email through `alert_notification.emails` when any threshold is breached, which is the trigger a team uses to decide whether retraining is warranted.
- An out-of-box configuration needs no explicit `monitoring_signals` at all — Azure Machine Learning defaults to data drift, prediction drift, and data quality with sensible thresholds, which is enough to catch the most common production problems without hand-tuning every metric up front.

Common confusion

A monitoring alert firing doesn't mean the model is definitely wrong — it means the *statistical properties* of production data or predictions have shifted from the reference window by more than the configured threshold. Confirming whether that shift actually degraded real-world accuracy (versus a benign, expected seasonal change) is a separate investigation step before deciding to retrain.

CHAPTER 3

Design and Implement a GenAIOps Infrastructure

Configure Microsoft Foundry projects, identities, and network isolation, deploy foundation models for production, and version prompts with Git.

Foundry resources, projects, managed identities, and RBAC

A Foundry resource is the Azure-billed account that hosts one or more projects — getting its identity and role assignments right up front avoids re-architecting access control after agents and evaluations are already in production.

Resources and projects

- The Foundry resource (an `Microsoft.CognitiveServices/accounts` resource under the hood) is what carries region, networking, and billing; a project inside it scopes a specific team or workload's agents, deployments, and connections.
- A Basic agent project uses platform-managed data resources (storage, Cosmos DB, AI Search provisioned automatically); a Standard agent project instead brings your own Storage, Cosmos DB, and AI Search resources — the right choice when data residency, existing infrastructure, or tighter control over those dependencies matters.

Managed identities

- The Foundry resource's system-assigned managed identity is what it uses to reach dependent resources (Storage, Cosmos DB, AI Search) without embedded credentials, the same pattern as an Azure Machine Learning workspace.
- When network isolation is enabled, that managed identity additionally needs the built-in **Azure AI Enterprise Network Connection Approver** role (role ID `b556d68e-0be0-4f35-a333-ad7ee1ce17ea`) assigned at the resource group (or subscription) scope, so it can auto-approve the private endpoints its managed network creates.

RBAC roles

- **Foundry Owner** / **Foundry Account Owner** (recently renamed from **Azure AI Owner** / **Azure AI Account Owner** — both names still appear during the rollout): full control including role assignment, typically held by a small platform/IT admin group.
- **Foundry User** (formerly **Azure AI User**, sometimes still called **Azure AI Developer** in hub-based docs): the working role for someone building and deploying agents and model connections day to day, without subscription-wide administrative rights.
- Narrower, model-specific roles like **Cognitive Services OpenAI User** (inference calls and viewing deployments only) versus **Cognitive Services OpenAI Contributor** (can also create/edit deployments and fine-tune) let you separate "can call this model" from "can change what's deployed," independent of the broader Foundry-level roles.
- A typical enterprise layout assigns Owner to IT admins at the hub/resource scope, Foundry User (or Contributor) to team leads who create projects, and a narrower per-project role to individual developers — mirroring the same least-privilege-at-the-right-scope principle as Azure RBAC generally.

Common confusion

The Foundry-native roles (Foundry Owner/User) and the Cognitive Services roles (Cognitive Services OpenAI User/Contributor) overlap in purpose but aren't interchangeable — a role assignment made through the classic Cognitive Services resource RBAC doesn't automatically show up as an equivalent Foundry role, and some scenarios (like Entra ID inference calls) specifically require one family or the other.

Network security and private networking

Foundry supports two distinct approaches to network isolation, and the exam expects you to know which problem each one solves rather than treating them as interchangeable "make it private" checkboxes.

Managed virtual network

- A managed virtual network is provisioned and operated by Microsoft on your behalf — you choose an isolation mode (`AllowInternetOutbound`, permitting general outbound internet access, or `AllowOnlyApprovedOutbound`, restricting egress to explicitly approved destinations) and Foundry builds the network plumbing for you.
- It's created via Bicep/Terraform templates, `az rest`, or the newer `az cognitiveservices account managed-network create` command — notably, the Azure portal UI does not currently support creating it, so this is CLI/IaC territory even for a first deployment.
- Once enabled, it cannot be disabled and there's no upgrade path from a custom (BYO) VNet setup to a managed one — the resource has to be redeployed from scratch, which makes this a decision to get right at initial provisioning rather than something to reconfigure later.

Bring-your-own virtual network

- BYO VNet injection places the Foundry resource's networking inside a virtual network you already control, giving you direct control over route tables, NSGs, and peering (including a hub-and-spoke design with a centralized firewall inspecting egress).
- Inbound isolation (a private endpoint so the resource itself isn't reachable from the public internet) and outbound isolation (routing agent/evaluation traffic through your VNet) are configured somewhat independently — a Standard agent project with BYO VNet is the template to reach for when you need full control over both directions plus your own Storage/Cosmos DB/AI Search.

Agent tool connectivity under network isolation

- Tools that use Microsoft's backbone network (Code Interpreter, function calling) need no extra networking configuration even when the Foundry resource is network-isolated.
- Tools reaching public endpoints (Bing grounding, web search, SharePoint) still work without private endpoints, but only because their traffic stays on the public internet — an organization that wants to block that public egress does so with Azure Policy, not by relying on the Foundry network configuration alone.
- Azure AI Search used as a private grounding source needs its own private endpoint set up separately (Foundry doesn't auto-create it), and any indexer feeding that search index must set `executionEnvironment: "Private"` explicitly — otherwise it silently defaults to multitenant execution that can't cross the private endpoint, producing a quietly empty index rather than an obvious error.

Common confusion

Enabling network isolation on the Foundry resource doesn't automatically secure every dependency it talks to — Azure AI Search, Storage, and Cosmos DB each need their own private endpoints created separately; the managed or BYO VNet controls the Foundry resource's own boundary, not every downstream service's.

Deploying foundation models for production

Foundry's model catalog offers two fundamentally different ways to put a model into production, and picking between them (and then between deployment types within the chosen path) is a recurring exam scenario.

Serverless API vs. managed compute

- Serverless API is the default, preferred path for essentially all Foundry Models — Azure OpenAI models and select partner/community models — where Microsoft hosts the inference infrastructure entirely; you never provision or size a VM, and billing is per-token (or per-PTU, covered below).
- Managed compute is reserved for open-source, partner, and custom models (including NVIDIA NIM and other industry-specific models) that need to run on dedicated GPU capacity Foundry manages on your behalf — billing here is hourly per accelerator SKU rather than per-token, and it's the option when a model simply isn't offered as a serverless API.
- Both support private networking and keyless (Microsoft Entra ID) authentication; only serverless API currently offers built-in, customizable content filtering and the full set of regional/data-zone/global data-processing choices.

Deployment types within serverless API

- Standard (pay-per-token): the default, best for development, testing, and variable or unpredictable production traffic — no capacity is reserved, so throughput can vary with overall platform demand.
- Global / Data Zone / Regional variants of Standard trade off where data is processed (worldwide, a specific data zone like US/EU/APAC, or a single region) against latency and compliance requirements — a data-residency requirement is what pushes you toward a non-Global option even though Global is typically cheaper and has faster queueing.
- Batch: discounted, asynchronous, no latency SLA — fits bulk offline scoring workloads, not interactive agent traffic.

Provisioned throughput units (PTUs)

- A PTU reserves a fixed amount of model-processing capacity exclusively for your deployment, billed per PTU-hour whether or not you actually send traffic — the trade you're making is guaranteed, predictable low latency in exchange for paying for idle reserved capacity.
- PTU quota is model-independent (any supported model can use the same PTU pool) but region- and deployment-type-specific, and each model has its own minimum PTU count and its own PTU-to-tokens-per-minute ratio — a heavier model needs more PTUs to hit the same throughput as a lighter one.
- `az cognitiveservices account deployment create --sku-name GlobalProvisionedManaged --sku-capacity 50 ...` creates a 50-PTU provisioned deployment; the right trigger for reaching for PTUs is predictable, high-volume, latency-sensitive production traffic — not development or bursty, hard-to-forecast usage, where Standard's per-token billing is more efficient.

> **Common confusion**
> Choosing "provisioned" doesn't just change how you're billed — it changes the latency profile itself. A Standard deployment has no latency SLA at all (it shares capacity and can slow down under platform-wide demand); only provisioned and priority-processing deployment types come with a defined latency target per model, which is the actual reason mission-critical workloads use them, not just the cost-at-scale argument.

Prompt versioning and management with Git

Treating a prompt as a first-class, versioned artifact — reviewed, diffed, and rolled back exactly like application code — is what keeps a GenAI application's behavior auditable as it evolves, instead of "the prompt currently in production" being whatever someone last pasted into a portal text box.

Prompts as version-controlled assets

- Storing prompts as files in a Git repository (rather than only inside a portal UI) means every change goes through the same pull-request review, diff, and history that application code does — a regression in agent behavior can be traced to the exact commit that changed the prompt.
- A repository structure typically separates prompt templates from the code that calls them, so a prompt change doesn't require a full application redeploy, and so multiple agents or environments can reference different prompt versions from the same repo.

Variants and comparison

- A prompt *variant* is an alternative phrasing, instruction set, or few-shot example set tested against the same task — keeping variants side by side in version control (rather than only as ephemeral playground experiments) is what makes a later "why did we choose this wording" question answerable.
- Comparing variants against the same evaluation dataset (see the quality-assurance topic) closes the loop: a prompt change is proposed as a PR, evaluated against quality/cost/performance metrics, and only merged once its results are at least as good as the version it replaces.

Safe, staged promotion

- The same trunk-based, PR-gated workflow used for infrastructure code applies here: a new prompt version is developed on a branch, evaluated automatically (often via a GitHub Actions step that runs the evaluation SDK against the changed prompt), and only promoted to the environment agents actually call after that check passes.
- Because prompts are just files, rolling back a bad prompt is a Git revert — no redeployment of the underlying model deployment or agent infrastructure is required, which is precisely why keeping prompts out of hardcoded application strings matters operationally, not just for cleanliness.

> **Common confusion**
> Versioning a prompt in Git controls *what text is sent to the model*; it says nothing on its own about *whether a given version is actually better*. Git-based prompt management and automated evaluation are complementary practices — one gives you a safe way to change and roll back prompts, the other gives you evidence for whether a specific change was an improvement.

CHAPTER 4

Implement Generative AI Quality Assurance and Observability

Evaluate quality and safety with Foundry's built-in evaluators, then observe cost, performance, and failures in production with tracing.

Quality evaluation with built-in evaluators

The `azure-ai-evaluation` SDK's built-in evaluators are the standard, out-of-the-box way to score a generative AI application's outputs against recognized quality dimensions, without writing custom scoring logic for the common cases.

General-purpose and RAG evaluators

- `CoherenceEvaluator` and `FluencyEvaluator` score logical consistency and natural-language quality respectively, independent of whether there's any retrieved context involved — useful for any generated text, not just RAG.
- `GroundednessEvaluator` measures how well a response is supported by retrieved context, returning a 1-5 model-judged score; `GroundednessProEvaluator` measures the same thing but as a binary pass/fail using the Azure AI Content Safety service instead of an LLM judge, and doesn't require a model deployment to run.
- `RelevanceEvaluator` scores how well the response actually answers the query; `RetrievalEvaluator` scores the retrieval step itself (did the system fetch relevant context at all, independent of what the model then did with it) — a RAG pipeline commonly needs both, since a bad final answer could stem from bad retrieval, bad generation, or both.
- `DocumentRetrievalEvaluator` and `ResponseCompletenessEvaluator` require a `ground_truth` to compare against, unlike most of the other AI-assisted evaluators — a data-requirement distinction worth remembering when assembling an evaluation dataset.

Running evaluations

- The `evaluate()` function runs multiple evaluators together against a dataset in one call: `evaluate(data="data.jsonl", evaluators={"groundedness": groundedness_evaluator, "relevance": relevance_evaluator})`.
- The dictionary key used for each evaluator is significant, not cosmetic — it has to match the documented keyword (`"groundedness"`, `"relevance"`, `"coherence"`, and so on) for that evaluator's results to be recognized and rendered correctly in the Foundry portal's evaluation UI.
- Composite evaluators bundle several individual ones for convenience: `QAEvaluator` runs groundedness, relevance, coherence, fluency, similarity, and F1 score together for a query/response pair; this is the fast way to get a broad quality snapshot without wiring up each evaluator individually.

Evaluation data and levels

- Built-in evaluators accept either simple query/response pairs or full conversations in JSONL — and every evaluator declares which `evaluation_level` it supports (`turn`, scoring one exchange, or `conversation`, scoring the whole multi-turn interaction); a single evaluation run can't mix evaluators that support different levels.

- Test datasets should combine production traffic samples (so evaluation reflects real usage patterns) with synthetically generated edge cases (so evaluation also covers scenarios production hasn't hit yet, like adversarial phrasing or rare intents).

> **Common confusion**
> `GroundednessEvaluator` and `RelevanceEvaluator` sound similar but measure different failure modes: a response can be highly relevant to the question asked while being completely ungrounded (fabricated, not actually supported by the retrieved context), and conversely can be perfectly grounded in the context while failing to actually address what the user asked — a comprehensive RAG evaluation needs both, since neither one alone catches the other's failure mode.

Risk and safety evaluation

Safety evaluators exist alongside quality evaluators as a distinct, mandatory category — a response can score perfectly on groundedness and relevance while still containing genuinely harmful content, so responsible AI practice treats the two categories as complementary, not substitutable.

Content-harm evaluators

- `ViolenceEvaluator`, `SexualEvaluator`, `SelfHarmEvaluator`, and `HateUnfairnessEvaluator` each detect a specific category of harmful content in a response, backed by the Azure AI Content Safety service rather than a general-purpose LLM judge.
- `ContentSafetyEvaluator` is the composite that runs all four together for a single combined output — the safety-category equivalent of `QAEvaluator` bundling the quality metrics.
- Because these evaluators call the Content Safety service rather than a deployed chat model, running them requires `azure_ai_project` configuration pointing at the Foundry project (instead of the `model_config` a quality evaluator like `GroundednessEvaluator` needs) — a setup detail that trips people up when reusing evaluation code between the two categories.

Attack- and leakage-oriented evaluators

- `IndirectAttackEvaluator` (also called XPIA, cross-prompt injection attack) measures whether a response fell for a jailbreak or malicious instruction that was smuggled in through *retrieved context* rather than the user's own message — the scenario where an attacker plants an instruction inside a document the RAG pipeline later retrieves and the model unwittingly follows.
- `ProtectedMaterialEvaluator` flags unauthorized reproduction of copyrighted or protected content in a response; `CodeVulnerabilityEvaluator` scans generated code for security issues; `UngroundedAttributesEvaluator` catches fabricated details the model inferred about a user or entity that weren't actually present in the input.
- Agent-specific safety evaluators (`ProhibitedActionsEvaluator`, `SensitiveDataLeakageEvaluator`) extend this same category to autonomous agent behavior — whether an agent stayed within its allowed actions and whether it exposed information it shouldn't have.

Combining evaluators for comprehensive coverage

- A typical RAG application's evaluation suite combines Retrieval, Groundedness, and Relevance (quality) with Content Safety (Violence/Sexual/Self-Harm/Hate) as a baseline; an agent application layers on Tool Call Accuracy, Task Adherence, and Intent Resolution.

- Safety evaluation isn't a one-time gate before launch — the same evaluators run as part of ongoing automated evaluation (batch runs against production or synthetic traffic), since model updates, prompt changes, and new attack patterns can all reopen a previously-closed safety gap.

> **Common confusion**
> Prompt-injection defenses and `IndirectAttackEvaluator` protect against different entry points that are easy to conflate: a *direct* jailbreak attempt sits in the user's own message and is a content-moderation/prompt-engineering problem, while an *indirect* attack (XPIA) is smuggled in through retrieved documents, tool outputs, or other context the model treats as trustworthy background rather than untrusted user input — which is exactly why it needs its own dedicated evaluator instead of being caught by standard content-safety checks on the user's message.

Observability: monitoring, tracing, and cost

Evaluation tells you whether outputs are good *before* wide release; observability is what tells you what's actually happening to a live application, in production, as real users and real cost accumulate against it.

What to monitor in production

- Latency, throughput, and error rate are the standard health signals for any production service, generative AI included — but token usage (input and output tokens per request) is the metric unique to LLM-backed applications, since it drives cost directly in a way a typical API call's compute cost doesn't vary request-to-request nearly as much.
- Azure Monitor, paired with a Foundry project's built-in continuous monitoring, is the standard combination: Azure Monitor collects and alerts on the platform-level metrics, while Foundry's monitoring view is scoped specifically to model deployments and agent behavior.
- A production troubleshooting workflow typically starts from a cheap, fast metric-based alert (latency spike, error-rate spike) and only then pivots to detailed tracing to find the specific root cause — going straight to trace-level detail for every request would be far too much data to sift through as a first step.

Distributed tracing with OpenTelemetry

- A single agent response can fan out across a retrieval call, one or more model calls, and custom business logic — OpenTelemetry represents that as a *trace* made of *spans*, each span being one unit of work with its own duration and attributes, nested in a parent-child hierarchy that shows exactly which downstream call is responsible for overall latency.
- The Azure Monitor OpenTelemetry Distro exports this span data to an Application Insights resource via its connection string; many Azure SDKs (and Foundry's own agent SDKs) emit spans automatically for their operations, so a meaningful amount of tracing detail shows up without writing any manual instrumentation.
- Custom spans, added by hand around business logic auto-instrumentation doesn't cover, share the same operation ID as everything else in that request — which is what lets Application Insights stitch spans emitted by completely different processes back into one coherent end-to-end trace.

Cost-aware performance tuning

- Because token consumption is directly billed, an observability setup for a generative AI application typically tracks token usage per request/user/feature as its own dimension, not just as a component of overall spend — this is what turns "our bill went up" into "this specific feature's prompts are unusually long."

- Decisions this feeds into include prompt-length reduction, switching a low-stakes step to a smaller/cheaper model, moving predictable high-volume traffic to a provisioned throughput deployment (trading reserved cost for latency guarantees), or adding caching for repeated queries.

Common confusion

Enabling an OpenTelemetry SDK and exporting traces *somewhere* isn't the same as those traces reaching Application Insights — the exporter must be explicitly configured with the Azure Monitor connection string, and initializing it *before* other instrumented libraries load matters; getting the order or configuration wrong is a common reason traces come out incomplete or don't appear in the Foundry/Application Insights view at all.

CHAPTER 5

Optimize Generative AI Systems and Model Performance

Tune RAG retrieval quality with chunking and hybrid search, and apply advanced fine-tuning and synthetic data to customize model behavior.

RAG optimization: chunking, embeddings, and similarity thresholds

Retrieval quality sets a ceiling on generation quality — no amount of prompt engineering fixes a response built from the wrong retrieved chunks, which is why RAG optimization starts at indexing time, not at the model call.

Chunk size and overlap

- Chunking exists partly to respect embedding and chat model token limits (for example, `text-embedding-3-small`'s roughly 8,191-token input limit), and partly because retrieval precision degrades when a chunk mixes multiple unrelated topics together.
- Azure AI Search's Text Split skill chunks by `pages` (character-based) or `sentences`, controlled by `maximumPageLength` and `pageOverlapLength` — a commonly cited reasonable default is a 2,000-character page length with 500 characters of overlap, though the right values depend on document structure and how the same chunks will be used (a chunk size that works well for embedding might not be ideal for summarization if both share the same pipeline).
- Overlap exists specifically to avoid losing context that straddles a chunk boundary — too little overlap can silently split a sentence's meaning across chunks that then can't be independently understood if only one is retrieved.

Embedding model selection

- The embedding model determines both retrieval quality and cost/latency per indexed document and per query — a larger embedding model generally captures finer semantic distinctions but costs more to run against every chunk at index time and every query at retrieval time.
- Embedding model choice and chunk size interact: a model with a small context window forces smaller chunks regardless of what would otherwise be an ideal chunk size for the content itself.

Similarity thresholds and hybrid search

- A vector query alone can return "the closest matches available" even when none of them are actually a good semantic match for the query — a minimum similarity threshold filters out results below a relevance cutoff rather than always returning the top-K regardless of quality.
- Hybrid search runs a keyword (full-text) query and a vector query in parallel over the same request, then merges and reorders results using Reciprocal Rank Fusion (RRF) — this offsets each approach's weakness individually: keyword search misses paraphrased/semantically-similar-but-differently-worded content, while vector search alone can miss exact terminology or identifiers (product codes, names) that a keyword match would catch precisely.
- Semantic ranking adds a second-stage rescoring pass on top of hybrid search's initial results, using a more sophisticated model against just the top candidates — a two-stage design because a heavier, more accurate ranking model would be too costly to run against every document in a large index directly.

Common confusion

Increasing chunk size doesn't uniformly improve retrieval — larger chunks preserve more context per retrieved unit (good for narrative or explanation-heavy content) but reduce retrieval precision (a large chunk containing the answer buried among unrelated text scores as "relevant" even though most of it isn't), so chunk sizing is a genuine trade-off tuned per content type and use case, not a setting to maximize.

Measuring and testing retrieval relevance

Tuning any of the RAG knobs above without a way to measure whether a change actually helped is just guessing — relevance measurement and structured comparison are what turn RAG optimization into an evidence-based process.

Relevance metrics

- `RetrievalEvaluator` and `DocumentRetrievalEvaluator` (from the same evaluation SDK used for generation quality) score the retrieval step specifically — whether the system fetched relevant material at all — separately from whether the final generated answer used that material well.
- Standard information-retrieval metrics (precision/recall over labeled relevant documents, and rank-aware metrics that reward relevant results appearing near the top) apply directly to a RAG retriever's output, since retrieval is fundamentally a search-ranking problem wearing a generative-AI hat.

A/B testing retrieval and prompt changes

- Because a RAG pipeline has multiple independently tunable stages (chunking strategy, embedding model, similarity threshold, hybrid vs. vector-only, reranking on or off, and the generation prompt itself), isolating which change actually caused an improvement requires changing one variable at a time against a fixed, representative evaluation dataset — not the entire pipeline in one attempt.
- A/B testing in production (splitting live traffic between a current and candidate configuration and comparing evaluation metrics, user engagement, or task completion) is the natural extension of offline evaluation once a candidate configuration looks promising in testing but its real-world impact still needs confirmation under live conditions.

Metadata filtering alongside similarity ranking

- Combining a vector similarity `ORDER BY`-style ranking with a metadata `WHERE`-style filter (tenant ID, document category, date range) narrows the search space to only documents a given user or use case should ever see, evaluated in the same query as the similarity ranking rather than as a separate post-filter step.
- Getting the filter/ranking order wrong (filtering after retrieving only the top-K by similarity, instead of filtering the candidate set before or during ranking) can silently return fewer results than expected, or worse, leak content that should have been excluded if the filter is applied only cosmetically to the response rather than to the actual search.

Common confusion

A high similarity score between a query and a retrieved chunk doesn't guarantee the chunk is actually useful for answering that query — semantic similarity measures topical closeness, not whether the specific fact or instruction the user needs is actually present in that chunk, which is exactly the gap groundedness and relevance evaluation (applied to the final generated response) are designed to catch that a retrieval-only similarity score can't.

Advanced fine-tuning methods

Fine-tuning changes a model's weights based on examples, rather than changing what's included in each prompt at inference time — the right tool when consistent behavior needs to be baked in rather than re-specified in every call.

Training methods

- Supervised fine-tuning (SFT) is supported by essentially all non-reasoning models and trains on labeled input/output examples directly — it's the default starting point for teaching a model a specific format, tone, or domain-specific behavior.
- Direct Preference Optimization (DPO) trains from *pairs* of preferred vs. non-preferred responses to the same input, rather than a single "correct" output — useful when the goal is steering style or judgment calls where there isn't one unambiguous right answer, only a comparative preference.
- Reinforcement Fine-Tuning (RFT) is for reasoning models and grades the model's output against a scoring function (a grader) rather than an exact reference answer — training data omits system messages, the final message must come from the user, and datasets can be considerably smaller (dozens to a few hundred examples to start) than SFT typically needs, though every training file goes through an automated safety screening pass before training can begin.

Data preparation

- Training and validation data are uploaded as JSONL files (UTF-8, under the size limit for the direct upload API — larger files use a separate multipart uploads API), with each line's schema differing by method: SFT and DPO follow a chat-messages format, RFT's format adds fields a grader needs and drops system messages entirely.
- Training and validation data must not overlap — validation exists specifically to catch overfitting, which reusing training examples for validation would silently hide.

Synthetic data for fine-tuning

- Where real labeled examples are scarce (a new domain, a rare intent, an edge case that hasn't occurred in production yet), synthetically generating example input/output pairs (often using a stronger model to generate candidate examples reviewed for quality) fills the gap — the same "combine production and synthetic data" pattern used for evaluation datasets applies to training data too.
- Synthetic data quality matters more as volume increases — a larger volume of *low-quality* synthetic examples can actively degrade a fine-tuned model's behavior rather than improve it, since the model has no way to distinguish a synthetic example's subtle errors from a deliberate lesson.

> **Common confusion**
> Fine-tuning and RAG solve different problems and are often confused as interchangeable "make the model know more" techniques: fine-tuning changes how the model behaves (format, tone, task-specific judgment) based on training examples baked into its weights, while RAG supplies current, specific facts at inference time without retraining anything — a model fine-tuned on last quarter's documents still won't know about a document added yesterday, which only retrieval can supply.

Monitoring and lifecycle management of fine-tuned models

A fine-tuned model is a distinct, deployed artifact with its own cost, performance profile, and drift risk — its lifecycle doesn't end once training completes, and treating it as "set once and done" is a common way production quality regresses silently.

From training to deployment

- A completed fine-tuning job produces a fine-tuned model identifier that then has to be explicitly deployed, exactly like a base model — Global-tier training (cheaper, faster queuing, but copies data/weights outside the resource's region) versus Standard-tier training (keeps data in-region, for data-residency requirements) is chosen at job creation, not at deployment time.
- Fine-tuned model deployments bill hourly for hosting *in addition to* training-time and inference-time token costs — an easy way to accumulate unnecessary spend is forgetting to delete a fine-tuned deployment that's no longer being evaluated or used, since the hosting cost accrues whether or not it's actively receiving traffic.

Monitoring a fine-tuned model in production

- The same evaluation and observability practices covered earlier (built-in evaluators, token/latency/cost tracking, tracing) apply to a fine-tuned deployment exactly as they do to a base model deployment — fine-tuning doesn't exempt a model from ongoing quality or safety evaluation.
- Fine-tuned models carry an additional drift risk specific to them: as the underlying task or domain evolves, a model fine-tuned on last year's examples can degrade in ways a base model wouldn't, since its specialized behavior was learned from a specific, dated snapshot of examples rather than kept current by retrieval.

Full lifecycle management

- Retraining a fine-tuned model (with updated or expanded training data) follows the same develop → evaluate → compare-against-current-production → promote pattern as any other model lifecycle change — the new fine-tuned version is evaluated against the same test set the current production version was validated against, not just checked in isolation.
- Versioning fine-tuned models the same way as any other deployed model (keeping the previous fine-tuned deployment available, shifting traffic gradually, and being able to roll back to it) applies the safe-rollout practices from the model lifecycle domain to fine-tuned models specifically, rather than treating a fine-tuned model swap as a special, riskier, all-or-nothing cutover.

Common confusion

Successfully completing a fine-tuning job and seeing good training/validation metrics doesn't mean the resulting model is ready for production traffic — those metrics reflect performance against the specific training and validation data used, not against the full range of real-world inputs the evaluation suite (groundedness, relevance, safety) is designed to catch problems across.

CHAPTER 6

Quick-reference: common confusions

Every “Common confusion” callout from this guide, gathered here as a last-day revision sheet.

Design and Implement an MLOps Infrastructure

A datastore's own authentication (how the *workspace* reaches storage) and a user's RBAC role on the workspace (what *that person* is allowed to do) are independent layers — a user with Contributor on the workspace can still be blocked from the underlying storage account if its network rules or access policies don't separately permit it.

A workspace's default asset store and a registry solve different problems: the workspace store versions assets *for that workspace's own jobs*; a registry exists specifically so multiple workspaces can share the same asset without each team re-registering and maintaining its own copy.

Bicep provisions the *infrastructure* (the workspace, its compute, its network configuration) — it doesn't submit training jobs or register models. Those day-to-day MLOps operations run through the CLI or SDK against an already-provisioned workspace, which is why real pipelines combine both: Bicep for the one-time (or infrequently changed) environment, CLI/SDK for the jobs that run against it continuously.

Workload identity federation secures *who* can trigger a deployment or job; network restriction secures *where the traffic can come from*. Configuring OIDC correctly doesn't help a workflow reach a network-isolated workspace from a public GitHub-hosted runner — the two controls have to be solved together, not interchangeably.

Implement Machine Learning Model Lifecycle and Operations

`mlflow.autolog()` captures what a specific framework integration knows how to log automatically — it won't pick up a custom metric your code computes that isn't part of that framework's standard training loop; anything domain-specific still needs an explicit `mlflow.log_metric()` call.

Grid and random sampling both support early termination; Bayesian sampling does not, because each new trial's parameter choices depend on the full results of prior trials rather than being independent — cutting a trial short would corrupt the very history the algorithm relies on.

A pipeline's steps run as independent jobs connected by data dependencies (parallelizing *stages* or independent branches); distributed training splits *one* training job's compute across multiple nodes or GPUs working on the same task together. Needing both is common — the "train" step of a pipeline can itself be a distributed job — but they solve different scaling problems.

`mlflow.register_model` and `mlflow.<flavor>.log_model(..., registered_model_name=...)` both end up registering a model, but only within the *same workspace where the run itself was tracked* — Azure Machine Learning doesn't support registering a model into a different workspace's registry directly from an MLflow call; cross-workspace sharing goes through a registry (see the MLOps infrastructure topic), not through MLflow's registration API.

Mirrored traffic and a 10%-live-traffic split look similar operationally but behave very differently for the caller: a live split actually returns the new deployment's predictions to whichever fraction of callers land on it, while mirroring never changes what any caller sees — it exists purely to observe a new deployment's behavior under real traffic before it's trusted with a live split at all.

A monitoring alert firing doesn't mean the model is definitely wrong — it means the *statistical properties* of production data or predictions have shifted from the reference window by more than the configured threshold. Confirming whether that shift actually degraded real-world accuracy (versus a benign, expected seasonal change) is a separate investigation step before deciding to retrain.

Design and Implement a GenAIOps Infrastructure

The Foundry-native roles (Foundry Owner/User) and the Cognitive Services roles (Cognitive Services OpenAI User/Contributor) overlap in purpose but aren't interchangeable — a role assignment made through the classic Cognitive Services resource RBAC doesn't automatically show up as an equivalent Foundry role, and some scenarios (like Entra ID inference calls) specifically require one family or the other.

Enabling network isolation on the Foundry resource doesn't automatically secure every dependency it talks to — Azure AI Search, Storage, and Cosmos DB each need their own private endpoints created separately; the managed or BYO VNet controls the Foundry resource's own boundary, not every downstream service's.

Choosing "provisioned" doesn't just change how you're billed — it changes the latency profile itself. A Standard deployment has no latency SLA at all (it shares capacity and can slow down under platform-wide demand); only provisioned and priority-processing deployment types come with a defined latency target per model, which is the actual reason mission-critical workloads use them, not just the cost-at-scale argument.

Versioning a prompt in Git controls *what text is sent to the model*; it says nothing on its own about *whether a given version is actually better*. Git-based prompt management and automated evaluation are complementary practices — one gives you a safe way to change and roll back prompts, the other gives you evidence for whether a specific change was an improvement.

Implement Generative AI Quality Assurance and Observability

`GroundednessEvaluator` and `RelevanceEvaluator` sound similar but measure different failure modes: a response can be highly relevant to the question asked while being completely ungrounded (fabricated, not actually supported by the retrieved context), and conversely can be perfectly grounded in the context while failing to actually address what the user asked — a comprehensive RAG evaluation needs both, since neither one alone catches the other's failure mode.

Prompt-injection defenses and `IndirectAttackEvaluator` protect against different entry points that are easy to conflate: a *direct* jailbreak attempt sits in the user's own message and is a content-moderation/prompt-engineering problem, while an *indirect* attack (XPIA) is smuggled in through retrieved documents, tool outputs, or other context the model treats as trustworthy background rather than untrusted user input — which is exactly why it needs its own dedicated evaluator instead of being caught by standard content-safety checks on the user's message.

Enabling an OpenTelemetry SDK and exporting traces *somewhere* isn't the same as those traces reaching Application Insights — the exporter must be explicitly configured with the Azure Monitor connection string, and initializing it *before* other instrumented libraries load matters; getting the order or configuration wrong is a common reason traces come out incomplete or don't appear in the Foundry/Application Insights view at all.

Optimize Generative AI Systems and Model Performance

Increasing chunk size doesn't uniformly improve retrieval — larger chunks preserve more context per retrieved unit (good for narrative or explanation-heavy content) but reduce retrieval precision (a large chunk containing the answer buried among unrelated text scores as "relevant" even though most of it isn't), so chunk sizing is a genuine trade-off tuned per content type and use case, not a setting to maximize.

A high similarity score between a query and a retrieved chunk doesn't guarantee the chunk is actually useful for answering that query — semantic similarity measures topical closeness, not whether the specific fact or instruction the user needs is actually present in that chunk, which is exactly the gap groundedness and relevance evaluation (applied to the final generated response) are designed to catch that a retrieval-only similarity score can't.

Fine-tuning and RAG solve different problems and are often confused as interchangeable "make the model know more" techniques: fine-tuning changes how the model behaves (format, tone, task-specific judgment) based on training examples baked into its weights, while RAG supplies current, specific facts at inference time without retraining anything — a model fine-tuned on last quarter's documents still won't know about a document added yesterday, which only retrieval can supply.

Successfully completing a fine-tuning job and seeing good training/validation metrics doesn't mean the resulting model is ready for production traffic — those metrics reflect performance against the specific training and validation data used, not against the full range of real-world inputs the evaluation suite (groundedness, relevance, safety) is designed to catch problems across.