Topic 1 ML on Databricks — Structuring & naming experiments, runs, models & assets (scalability, traceability, governance)

The asset hierarchy (naming maps to this)

- **Experiment** → contains Runs (each training/eval execution, with params/metrics/artifacts/tags).
- **Registered model** in Unity Catalog (catalog.schema.model) → contains immutable versions → pointed to by mutable aliases (@champion, @challenger).
- Related assets — **training datasets** (Delta tables, versioned), **feature tables, artifacts** (SHAP, confusion matrix) in UC Volumes, **UC functions**.
- Golden rule: everything lives in Unity Catalog so data → features → runs → model version → serving share one governance + lineage layer.

1. **Experiments** — structure & naming
   - Use **workspace (shared) experiments**, not notebook-scoped ones, for anything production. Notebook experiments tie history to one notebook and don't scale or govern well.
   - One experiment per model problem (for ASM: per teaching-session-type), not one giant experiment.
   - One workspace experiment per session-type per environment, at a team-owned path:

     mlflow.set_experiment(f"/Workspace/Teams/PVCE/asm/{env}/asm_risk_{session}")
   - **Separate experiments per environment (dev/staging/prod)** — keeps prod history clean and auditable.
2. **Runs** — naming & tagging (where traceability is won)
   - Give runs a meaningful **run_name** — never rely on the random hash (e.g. tp1_high_recall_2026T1_gitabc123).
   - **Standard tag taxonomy every run must carry** (bake into the training template so it's automatic):
     - **env** (dev/staging/prod), git_sha, git_branch
     - **session_type** (TP1 / UNSW_Online / Canberra), **term** (FILTER_TERM_TRAIN/TEST)
     - **model_type** (high_precision / high_recall), **downsampling_rate**
     - **data_version** (Delta table version/snapshot used), **author**, **trigger** (manual/scheduled)
   - Log the **input dataset with mlflow.data** so the run records which table + version it trained on (reproducibility + lineage).

- Log **per-segment / fairness** metrics, not just global accuracy — serves the TEQSA/HESOP demographic obligation and makes the validation gate meaningful.
- Use nested runs: parent per deployment cycle, children for HP and HR variants (or a hyperparameter sweep).

3. **Models** in Unity Catalog — naming & lifecycle
   - Three-level namespace catalog.schema.model:
     - **catalog** = environment (prod / staging / dev)
     - **schema** = team/domain (pvce_asm)
     - **model** = problem + variant, e.g. prod.pvce_asm.asm_risk_tp1_high_recall
   - Don't use the old stage names (None/Staging/Production are deprecated in UC). **Use aliases for deployment status** — @champion, @challenger, optionally @shadow. Versions are immutable; promote by reassigning an alias.
   - **Tag** model versions with governance metadata: validation_status, fairness_checked, approved_by, eval_run_id.
   - Set descriptions + log the model signature + an input example so consumers know the contract.
   - ASM-specific: they run HP (week 1) and HR (week 2+) per session. Prefer separate registered models per variant (…asm_risk_tp1_high_precision and …asm_risk_tp1_high_recall), each with its own @champion — clearest given the deliberate week-1/week-2 switch. (Alternative: one model per session with HP/HR as aliases/tags.)
4. Organising **experimentation across teams & environments**
   - Environment axis: separate experiments + UC catalogs per env; the model registers to the env catalog of the code that runs it (prod code → prod catalog). This is the "deploy code, each env trains on its own tier" principle.
   - Team axis: namespace by team/domain — workspace experiment folders (/Workspace/Teams/PVCE/…) and a UC schema per team (pvce_asm). Use account groups + grants to reflect ownership: PVCE-DS owns dev/experiment write; EDO owns prod-catalog promotion; aliases reassigned only by the CI service principal / approver.
   - Standardise with a template, not documentation. Conventions scale across teams when encoded in MLOps Stacks (DABs) — every new initiative (ASM per-session, skills vectorization, recommendation engine) is scaffolded with the same experiment paths, tag taxonomy and model-naming rules. Wiki conventions drift; template conventions don't.
   - One tagging taxonomy across the team so traceability queries work uniformly (e.g. "every prod model trained on data_version X", "every run for session=UNSW_Online").

Resources (Azure):

- Manage model lifecycle in UC — https://learn.microsoft.com/en-us/azure/databricks/machine-learning/manage-model-lifecycle/
- MLflow tracking/experiments — https://learn.microsoft.com/en-us/azure/databricks/mlflow/
- Big Book of MLOps — https://www.databricks.com/resources/ebook/the-big-book-of-mlops

Topic 2 — Dev/Staging/Prod structure, prototype→prod promotion, CI/CD, governance vs velocity

Grounded in the Databricks MLOps workflow / Big Book of MLOps / MLOps Stacks docs (✅ = documented; 🔵 = ASM-specific application).

1. Structuring ML development across dev / staging / prod ✅

Three environments, **separate workspaces, each bound to its own Unity Catalog catalog** (dev / staging / prod), joined by workspace–catalog binding. Workspaces give execution + access isolation; catalogs give data + model isolation. The roles the docs prescribe:

Env	Who	Access	What happens
Dev	Data scientists	**Read-write on dev catalog; read-only on prod data**	Experiment with features/models, exploratory work
Staging	ML engineers / CI	**Own staging catalog; mirrors prod**	All pipeline code tested via CI/CD; integration tests
Prod	ML engineers / CI service principal	**DS read-only (diagnosis/monitoring); only CI/CD SP + approvers write**	Validated pipelines run; models trained + served

🔵 For ASM, this resolves the "training on dummy data is useless" objection: **dev/staging get governed read access to prod feature tables via UC** — they keep real data and still get environment separation, with no copying.

2. Promoting from prototype to production — the process ✅

The principle: **deploy code, not models.** Promote the training code dev→staging→prod through Git; **each environment retrains on its own data tier and registers to its own catalog**, so the prod model is always built by reviewed, tested prod code — not an artifact hand-carried from a laptop.

The flow:

1. Develop in dev done → open a **PR**.
2. **CI** runs unit tests + an **integration test in staging** (deploys the pipeline, runs a smoke train+infer). Must pass to merge.
3. Merge to main → **CD** deploys the pipelines to **prod** via a service principal.
4. Prod **training pipeline** trains + registers the model to the prod catalog.
5. **Validation gate** runs (format / metadata / performance + 🔵 fairness/segment checks for TEQSA) → assigns the **@challenger** alias if it passes.
6. **Deployment step** compares challenger vs the current **@champion** (offline or A/B) → on success **repoints @champion**.
7. **Inference references @champion** — picks up the new version automatically, no pipeline edit. Rollback = repoint the alias.

For GenAI (their caption LLM / future agents): same code-promotion model, but the "model" may be a pipeline / prompt / chain. Version-control and promote prompts, chain code, eval configs and endpoint configs; reference models/endpoints by UC alias per env; augment the metric gate with an **eval gate** (LLM-as-judge + human feedback). Promotion is still code + alias, not artifact.

3. CI/CD patterns that automate promotion between workspaces ✅

- **Unit of deployment = Databricks Asset Bundles (DABs):** jobs, pipelines and models declared in YAML with **per-environment targets** (each target points at a different workspace host + catalog). One bundle, three targets.
- **Template = MLOps Stacks:** Databricks' opinionated project built on DABs that ships the dev/staging/prod CI/CD workflows out of the box.
- **CI engine = Azure DevOps Pipelines** (their existing tooling) or GitHub Actions.
- **Default MLOps Stacks pattern:** on PR → CI tests + deploys the bundle to the **staging** workspace for an integration run; on merge to main → CD deploys the bundle to the **prod** workspace via a **service principal**.
- 🔵 **ASM:** they already do Azure DevOps PR + code review + a manual "run training notebook in prod" step — wrap that manual step in a **DAB deployed by the Azure DevOps pipeline**, and **parameterise per session** (FILTER_TERM_*, session, variant) so a new teaching session is a config, not a cloned notebook.

4. Balancing governance with delivery velocity ✅🔵

- **Automated gates give both at once** — speed with guardrails, instead of a manual approval committee that slows everyone.
- **Tiered controls:** light in dev (don't gate experimentation), integration tests in staging, strict in prod (validation gate + approval on alias promotion + service-principal-only writes). Friction where risk is, freedom where it isn't.
- **Governance is built-in, not bolted-on:** Unity Catalog provides lineage, access control and audit as a property of the platform — "governed" doesn't mean "slow."

- **Aliases decouple "which model is live" from deployment** → instant, safe rollback = velocity and control.
- 🔵 **Encode compliance as code:** put the TEQSA/HESOP fairness/demographic check **inside** the automated validation gate, so regulatory assurance is automatic rather than a manual sign-off bottleneck.
- **Approvals only where they add value** (the prod @champion promotion); automate everything upstream.

Resources (Azure): MLOps workflows on Databricks · CI/CD for ML · Databricks Asset Bundles · MLOps Stacks · Big Book of MLOps

Topic 3 — CI/CD and Code-First Deployment

Markers: ✅ = documented best practice (cited) · 🔵 = ASM-specific application.

Why code-first is preferred over manual model-artifact promotion ✅

Databricks recommends promoting **code, not model artifacts**, from one environment to the next (MLOps workflows on Databricks; Big Book of MLOps). The reasons:

- **Consistent review + testing:** all assets (training, inference, deployment code) go through the same code-review and CI test path; a hand-promoted artifact bypasses that.
- **Provenance:** the production model is **trained by production-approved code on production data**, so you always know exactly how it was built. Manual artifact hand-off loses the "how was this produced" lineage.
- **Reproducibility & rollback:** rollback = revert a commit (or repoint an alias to the prior immutable version); re-training is deterministic from versioned code + data.
- **Auditability:** the full chain code → run → model version → serving is captured in Git + MLflow + Unity Catalog.

Recommended CI/CD practices for reliable ML delivery ✅

Per CI/CD for ML on Databricks and the MLOps workflow:

Stage	Practice
Test	Unit tests on feature/transform logic; data-quality checks on inputs
Integration	**Deploy the pipeline to staging and run a smoke train+inference on a sample**
Validation	**Model validation gate — format, metadata and performance thresholds (🔵 + fairness/segment checks for TEQSA); pass → assign @challenger**
Deployment	**CD deploys via a service principal; compare @challenger vs @champion (offline eval or A/B), then repoint @champion**

Rollback	Repoint the @champion alias to the previous immutable version — instant, safe
Monitoring	Post-deploy Lakehouse Monitoring + retraining triggers (see Topic 6)

Reference architectures / patterns customers use ✅

- The **MLOps reference architecture** (Big Book of MLOps) — the canonical dev/staging/prod, deploy-code pattern.
- **Databricks Asset Bundles (DABs)** as infrastructure-as-code, with **MLOps Stacks** as the opinionated template (Topic 4). The default pattern: PR → tests in staging; merge to main → deploy to prod via service principal (MLOps Stacks).
- 🔵 **ASM:** they already promote code via Azure DevOps PRs — the change is to automate the manual "run training notebook in prod" step with a DAB driven by their Azure DevOps pipeline.

Resources: CI/CD for ML · MLOps workflows · Databricks Asset Bundles · Big Book of MLOps

Topic 4 — MLOps Stacks and Reusable Delivery Frameworks

What MLOps Stacks is and how to adopt it ✅

MLOps Stacks is Databricks' **opinionated, production-ready project template built on Databricks Asset Bundles**. It scaffolds the full ML project — training + batch inference pipelines — with **dev/staging/prod CI/CD workflows out of the box** (PR-triggered tests in staging, automated deploy to prod). Adopt it with `databricks bundle init mlops-stacks` (repo: github.com/databricks/mlops-stacks).

It includes, pre-wired: notebooks, MLflow, Feature Engineering, **Models in Unity Catalog** (the recommended registry), Model Serving, Lakeflow Jobs, and **GitHub Actions or Azure DevOps** CI/CD.

Reusable, CI/CD-ready templates across multiple initiatives ✅🔵

Init options let you split the work by role — ideal for their PVCE-DS vs EDO-MLOps division:

Init option	Who	Gets
CICD_and_Project	Both	Full ML code + CI/CD infra
Project_Only	Data scientists (Asjad/Daniel)	ML pipeline code structure
CICD_Only	ML Ops (Teresa) / EDO	CI/CD workflows + resource

		config

🔵 The value for them: one standardized template means **ASM per-session models, skills-vectorization, and the recommendation engine** all follow the same experiment paths, naming, tag taxonomy and CI/CD — conventions are *encoded*, not left to a wiki that drifts.

Lessons learned from enterprise-scale implementations ✅

From the MLOps Gym series (Crawl/Walk/Run):

- **Provide common infrastructure reusable across projects** (don't rebuild per team) — exactly what a shared stack delivers.
- **Implement best practices from the start** (clean code, proper configs, naming) — cheaper than retrofitting.
- **Build trust through active monitoring** of data and models.
- Adopt in **phases** — Crawl (foundations), Walk (CI/CD integration), Run (rigor + quality) — rather than boiling the ocean.

Resources: MLOps Stacks · Repo · DABs for MLOps Stacks · MLOps Gym

Topic 5 — GenAI Runtime and Search Infrastructure

Managing LLM runtime refresh cycles and model updates ✅

- **Use Provisioned Throughput for production** GenAI workloads (docs): dedicated, autoscaling inference capacity (configurable min/max tokens-per-second) — predictable cost and no per-minute (OTPM) rate-limit. Pay-per-token is for experimentation/spiky low-volume. Provisioned Throughput supports a whole model-architecture family, including fine-tuned/custom models.
- **Plan for the model deprecation lifecycle:** Databricks-hosted foundation models are retired on a published schedule (supported models) — e.g. deprecations beginning Feb 15 2026 (pay-per-token) / May 15 2026 (provisioned throughput). So treat LLM updates as a managed cycle: pin the model/version, **test the candidate model before cutover**, and use **Unity AI Gateway A/B traffic split + fallback** to swap models safely (AI Gateway).
- Reference the endpoint by config/alias so a model swap is a configuration change, not a code rewrite.

Maintaining and refreshing embeddings used by AI Search ✅

Databricks AI Search (Vector Search) keeps embeddings current automatically via a **Delta Sync Index** (create index docs):

Concern	Behaviour / best practice
Sync mode	**Continuous (seconds latency, but provisions a streaming cluster → higher cost) vs Triggered (on-demand via UI/SDK/REST → cheaper). Pick by freshness need.**
Data change	**Incremental — only changed rows are re-embedded; unchanged rows' embeddings are reused (managed embeddings), minimizing embedding-API calls and cost.**
Managed embeddings	Databricks generates + stores embeddings; saved to a UC <index>_writeback_table linked from the index page.
Embedding-MODEL change	**The expensive case: swapping the embedding model means re-embedding the entire corpus and rebuilding the index — plan a re-index or dual-index cutover (build new, validate, switch).**

Embedding lifecycle, indexing strategies, operational implications ✅🔵

- **Indexing strategy:** Delta Sync with *managed* embeddings (simplest), Delta Sync with *self-managed* embeddings (you control the model), or Direct Vector Access (you write vectors directly). Choose by how much control you need over the embedding model.
- **Operational rule of thumb:** a **data change = incremental, cheap**; an **embedding-model change = full re-embed, expensive** → schedule model swaps deliberately and validate retrieval quality before cutover.
- 🔵 For UNSW: this is the **skills-vectorization** use case (separate from ASM). Compare FMAPI embeddings vs the open-source model on cost + data residency (Azure, sovereignty), and A/B via AI Gateway before committing.

Resources: Foundation Model APIs · Provisioned Throughput · Supported models / deprecation · Create AI Search index · Unity AI Gateway

Topic 6 — Inference Architecture, Validation, and Monitoring

Modularising inference pipelines and services ✅

Separate the **feature pipeline**, the **inference pipeline**, and **serving** so each is tested and deployed independently (MLOps workflows). Choose the serving mode by need: **batch** (scheduled job / pySpark UDF — their current ASM weekly pattern), **streaming**, or **real-time**

(Mosaic AI Model Serving REST endpoint). Inference references the model by **@champion alias**, so it auto-picks a new version with no pipeline edit.

Validation frameworks — champion/challenger ✅

Per Manage model lifecycle in UC: a validation step runs format/metadata/performance checks and, if passing, assigns the **@challenger** alias. The deployment step then **compares challenger vs the current @champion** — via offline evaluation on a holdout or online A/B — and **promotes by repointing @champion** on success. Versions are immutable; the alias is the movable "what's live" pointer.

Monitoring, evaluation, and governance for production ML + GenAI ✅

- **Lakehouse Monitoring** (docs) offers three profile types — **InferenceLog**, **TimeSeries**, **Snapshot**. For model monitoring, attach an **InferenceLog** monitor to an inference table (inputs + predictions, + ground truth for accuracy). It creates two tables: a **profile metrics** table (summary stats + model accuracy) and a **drift metrics** table (distribution change). Drift can be **consecutive** (vs the previous window) or **baseline** (vs a baseline table). 🔵 This natively replaces their hand-built "Fact Precision" + the Azure ML monitoring plan, and covers their documented signal table (JS-distance, PSI, etc.).
- **GenAI evaluation/governance: Unity AI Gateway inference tables** (docs) log every request/response to a UC Delta table for quality + compliance auditing; **MLflow evaluation** (LLM-as-judge scorers + human feedback) scores GenAI output (LLMOps).

Safely introducing new models + measuring before full rollout ✅

- **Champion/challenger** (above) — promote only if the challenger wins on a holdout + 🔵 segment/fairness metrics.
- **A/B traffic split + fallback** via Model Serving / AI Gateway traffic routing — send a fraction of traffic to the new model, compare live, then ramp.
- **Shadow/canary** — run the new model alongside without serving its output, compare, then cut over.
- Always measure on **per-segment** metrics (not just global) for a regulated, demographic-reporting workload.

Resources: Lakehouse Monitoring · Manage model lifecycle / aliases · Model Serving · AI Gateway inference tables · LLMOps