novartis_datathon_2025-Arman – Copilot Agent TODO (Leaderboard Improvement Phase)

> **Scope**: You are Copilot for VS Code, operating **inside the `novartis_datathon_2025-Arman` repo**.
> Goal: **Use the existing rich infrastructure to push leaderboard scores up**, without breaking reproducibility or rules.

---

0. Safety & Reproducibility Guardrails

- [ ] **Always load configs from YAML (no hard-coding):**
  - [ ] Ensure all training / inference entry points call `load_run_config()` and **do not** override critical values inline (validation split, official metric weights, sample weights).
  - [ ] If you add new options (e.g. new ensemble, new segment), wire them into `configs/run_defaults.yaml` or separate `configs/model_*.yaml` instead of hard-coding.
- [ ] **Keep tests green:**
  - [ ] After any change in `src/data.py` , `src/features.py` , `src/validation.py` , `src/evaluate.py` , or `src/models/*` , automatically add / update tests in `tests/test_smoke.py` (or a new test file) so that `pytest` still passes with **0 failures and 0 warnings**.
  - [ ] If a change is non-trivial (new feature group, new model type, new ensemble), add at least **one minimal regression test** that would fail if the new logic broke silently.
- [ ] **Respect competition constraints:**
  - [ ] Do not introduce external data loading or extra CSVs unless they are clearly allowed by the rules and documented in `docs/planning/approach.md` .
  - [ ] Ensure **no test leakage**: all features must obey scenario cut-offs and bucket definitions as enforced by the existing validators.

---

1. Understand & Exploit the Existing Infrastructure

- [ ] **Scan configs and code layout (Arman project):**
  - [ ] Open `configs/run_defaults.yaml` , `configs/features.yaml` , and all `configs/model_*.yaml` to understand:
    - [ ] How scenarios 1 and 2 are configured (`forecast_start` , `forecast_end` , `feature_cutoff`).
    - [ ] How **sample weights** are defined over time windows and buckets.
    - [ ] Which models are currently “active” (CatBoost vs LightGBM vs XGBoost vs NN vs Hybrid vs ARIHOW).
  - [ ] Open `src/` :
    - [ ] `data.py` (panel building & pre-entry stats)
    - [ ] `features.py` (feature groups, leakage guards)

        - [ ] `validation.py` (series-level split, adversarial validation)
        - [ ] `evaluate.py` (official metric wrapper + config validation)
        - [ ] `models/*.py` (CatBoost, LGBM, XGB, NN, CNN-LSTM, KG-GCN-LSTM, ARIHOW, hybrid, ensembles)
        - [ ] `train.py` and `inference.py` (CLI, submission generation, edge-case handling).

- [ ] **Verify official metric alignment:**
    - [ ] Call `validate_config_matches_official()` from `src/evaluate.py` in at least one test / CLI command to ensure the YAML `official_metric` section exactly matches Novartis rules.
    - [ ] Confirm that **every training run and CV report uses the same metric** (no mixing of MAE/RMSE/other metrics for model selection).

---

2. Systematically Re-run & Compare Core Models

Use the existing CLI, panels, and feature matrices. Make sure the **artifacts directory and logs** stay organized per run.

- [ ] **Rebuild features (if needed, with cache awareness):**
    - [ ] Ensure all `python -m src.data ...` commands for train/test and both scenarios are still working (rebuild only if configs changed).

- [ ] **Train core models per scenario** (with current "sane" hyperparams, not yet HPO):
    - [ ] Scenario 1:
        - [ ] CatBoost
        - [ ] LightGBM
        - [ ] XGBoost
        - [ ] Linear / Ridge / ElasticNet baseline
        - [ ] Hybrid physics-ML model
        - [ ] ARIHOW model
    - [ ] Scenario 2:
        - [ ] Same family: CatBoost, LGBM, XGB, linear, hybrid, ARIHOW.

- [ ] **Parse `training_all_models.log` and artifacts:**
    - [ ] Write a small utility (e.g. in `src/utils.py` or a separate analysis script) that:
        - [ ] Reads log files in `artifacts/*` and extracts:
            - [ ] Experiment name, timestamp
            - [ ] Scenario, model type
            - [ ] CV scores (official metric, RMSE)
        - [ ] Builds a **summary table** and saves it (e.g. `artifacts/model_score_summary.csv`).
    - [ ] Use this to **rank models by official metric per scenario**.

- [ ] **Establish "hero baselines":**
    - [ ] For each scenario, mark in `docs/planning/approach.md` the **current best single model** based on official metric and stable CV (not just random seed luck).

  - [ ] These will be the anchors for ensembling.

---

3. Improve Validation, Segmentation & Weighting

Use Arman’s advanced validation tools instead of inventing new ad-hoc tricks.

- [ ] **Stress-test validation strategy:**
  - [ ] Confirm `create_validation_split()` in `src/validation.py` is using series-level stratification by bucket.
  - [ ] Run an **adversarial validation experiment** (if implemented) and inspect:
    - [ ] Whether train/validation distributions are very similar for key features.
  - [ ] Add a short script / notebook that prints key distribution comparisons (e.g. bucket, months_postgx, mean_erosion) between train and val.
- [ ] **Segmented modelling experiments (controlled):**
  - [ ] Design a small set of experiments where models are trained:
    - [ ] Only on **early months** buckets / windows (e.g. months 0–5, bucket 1) vs full horizon.
    - [ ] Possibly **separate models per bucket** (or per large therapeutic area) **if** there is enough data.
  - [ ] For each segmented experiment, log:
    - [ ] CV metric on its segment.
    - [ ] Overall official metric when plugged into the full submission flow.
- [ ] **Tune sample weights based on erosion economics (not just guesswork):**
  - [ ] Use Arman’s `sample_weights` section in `run_defaults.yaml` to:
    - [ ] Slightly increase weight on **early months** and **more critical buckets** where forecast accuracy matters more for business value.
  - [ ] Keep weight changes **moderate** and well-documented; maintain at least one “baseline” config with original weights for comparison.

---

4. Hyperparameter Optimization (HPO) with Existing Tools

The Arman project already supports HPO / sweeps. The task is to **use it surgically**, not explode compute.

- [ ] **Lock reasonable search spaces in `configs/model_*.yaml`:**
  - [ ] For CatBoost, LGBM, and XGBoost:
    - [ ] Define narrow, meaningful ranges (depth, learning rate, number of trees, regularization) instead of huge grids.
    - [ ] Include settings that are known to work well for tabular time-series (e.g. lower learning rate, more estimators + early stopping).
  - [ ] Ensure the number of combinations is **compute-friendly** given your hardware.
- [ ] **Enable HPO via CLI only when requested:**
  - [ ] Confirm `src/train.py` supports a `--hpo` or similar flag.

    - [ ] Add / update documentation in `TODO.md` or `docs/planning/approach.md` describing:
        - [ ] Exact commands to run a short HPO sweep per scenario and model type.
        - [ ] Where results (best params, plots) are stored.

- [ ] **Run targeted sweeps for hero models:**

    - [ ] Start with **CatBoost S1** and **CatBoost S2**, because they are already strong.
    - [ ] Then test HPO on **one gradient boosting alternative** (LGBM or XGB) for each scenario.
    - [ ] Update the model configs with the **best found hyperparameters**, while keeping the original configs as a commented “baseline” block for reproducibility.

---

5. Build Better Ensembles (Without Chaos)

You already have multiple model families implemented and ensemble classes in `src/models/ensemble.py`. Use them systematically.

- [ ] **Calibrate base models for ensembling:**

    - [ ] Ensure all candidate base models for an ensemble:
        - [ ] Are trained on **exactly the same train/val split**.
        - [ ] Use the same target transformation & scaling.
        - [ ] Output predictions on the same panel / feature set.

- [ ] **Implement and evaluate simple ensembles first:**

    - [ ] Use the existing averaging / weighted / stacking / blending classes to build:
        - [ ] CatBoost + LightGBM ensemble.
        - [ ] CatBoost + Hybrid physics-ML + ARIHOW ensemble.
    - [ ] For each ensemble:
        - [ ] Optimize **weights using validation official metric** (not just RMSE).
        - [ ] Log ensemble composition & weights in a structured artifact (e.g. JSON or YAML).

- [ ] **Scenario-aware ensembling:**

    - [ ] For Scenario 2, consider a **specialized ensemble** that puts more weight on models that handle early post-entry dynamics well (e.g. early-erosion features, ARIHOW / hybrid components).
    - [ ] For Scenario 1, emphasize models that generalize from pre-entry patterns (e.g. CatBoost + hybrid).

- [ ] **Guardrail checks on ensembles:**

    - [ ] Ensure ensembles never generate:
        - [ ] Negative volumes.
        - [ ] Extreme outliers (e.g. > 5× pre-entry average) without clipping or fallback.
    - [ ] Use `check_prediction_sanity()` and submission validators before saving any ensemble submission.

---

6. Explore Deep / Hybrid Architectures (Only After Strong GBM Baseline)

Once Tabular GBMs + ensembles are solid, selectively explore deep models (only if time allows).

- [ ] **CNN-LSTM / KG-GCN-LSTM experiments:**
    - [ ] Use `sequence_builder.py` to construct sequences for scenario 2 (where early post-entry dynamics exist).
    - [ ] Train at least:
        - [ ] One basic CNN-LSTM model.
        - [ ] One KG-GCN-LSTM model (if hardware allows).
    - [ ] Compare their validation metrics to CatBoost baseline **on the same panels**.
- [ ] **Use deep models mainly for ensembling:**
    - [ ] Even if deep models are slightly worse individually, check if adding them to an ensemble improves the final official metric due to complementary error patterns.
- [ ] **Control complexity:**
    - [ ] Keep architectures small enough to train reliably on local hardware / Colab.
    - [ ] Avoid huge hyperparameter searches; start from fixed configs in `configs/model_cnn_lstm.yaml` and `configs/model_kg_gcn_lstm.yaml`.

---

7. Inference, Edge Cases & Final Submission Quality

- [ ] **Harden `generate_submission()` in `src/inference.py`:**
    - [ ] Confirm fallback strategies (`exponential_decay` etc.) are triggered for edge cases:
        - [ ] Missing series.
        - [ ] Invalid or extreme predictions.
    - [ ] Ensure denormalization uses the correct `pre_entry_stats` column (e.g. `avg_vol_12m`) for **both scenarios**.
- [ ] **Scenario detection robustness:**
    - [ ] Verify `detect_test_scenarios()` correctly identifies Scenario 1 vs Scenario 2 series from the test panel.
    - [ ] Add / extend tests to cover tricky edge cases (e.g. incomplete early months).
- [ ] **Submission sanity dashboards:**
    - [ ] Implement a small report generator (could live in `src/evaluate.py` or a separate script) that:
        - [ ] Summarizes per-scenario and per-bucket prediction distributions.
        - [ ] Flags suspicious patterns (e.g. flat lines, negative growth where impossible).
    - [ ] Save this report (CSV / markdown) next to each submission in `submissions/`.

---

8. Competition Strategy & Experiment Tracking

- [ ] **Use the existing `ExperimentTracker` consistently:**

    - [ ] Ensure all new experiments log:
        - [ ] Model type, scenario, key hyperparameters.
        - [ ] CV metric and official metric.
        - [ ] Paths to artifacts and submissions.
    - [ ] If MLflow or W&B is available, track at least **the final shortlisted configs** there.

- [ ] **Integrate with FinalWeekPlaybook logic:**

    - [ ] Once you have a strong “hero config” + ensemble:
        - [ ] Register it as the **frozen best config** in the playbook.
        - [ ] Wire multi-seed training into the CLI so generating N seeds is easy.
        - [ ] Implement the **5-variant submission strategy** (best CV, underfitted, overfitted, bucket-1 focus, sanity-conservative).

- [ ] **Prepare for presentation (currently not started):**

    - [ ] Start a short `docs/presentation_outline.md` that captures:
        - [ ] Problem framing (generic erosion, scenarios 1 & 2).
        - [ ] Key modeling ideas (validation, features, models, ensembles).
        - [ ] Business interpretation (why the forecast matters, buckets, early months).

---

9. How Copilot Should Behave in VS Code

When the human edits code, you (Copilot) should:

- [ ] Prefer **editing config files + small interfaces** over duplicating complex code.
- [ ] Suggest **small, testable changes** instead of giant refactors.
- [ ] Auto-complete boilerplate for:
    - [ ] New model classes that extend `BaseModel`.
    - [ ] New tests mirroring existing patterns in `tests/test_smoke.py`.
    - [ ] CLI argument wiring in `src/train.py` and `src/inference.py`.
- [ ] Remind the user (via comments / docstrings) to:
    - [ ] Re-run `pytest` after significant changes.
    - [ ] Re-run a known baseline config to verify nothing regressed.
    - [ ] Log any new experiment in the tracking system and docs.

End state:
You have a **stable, reproducible Arman project** with:

- Strong single-model baselines in both scenarios.
- Carefully tuned GBM models.
- Smart ensembles using hybrid and possibly deep models.
- Robust validation and submission checks.
- A clear experiment log and story ready for final presentation.