Distributed PRNG Analysis Pipeline - Technical Addendum

Deeper details on feature contracts, sidecars, leakage guardrails, multi-model GPU behavior, and Step 6 pool scoring. Prepared by Team Beta - December 2025.

What's inside

Section	What you'll learn
A1. Pipeline contract and artifacts	Exact handoffs, file shapes, and why sidecars exist.
A2. Feature system (48 per-seed + 14 global)	What "same features" means, and how ModelFactory uses the schema.
A3. Label leakage checks and why "R² = 1" can happen	The three failure modes we already saw and how we guardrail them.
A4. Multi-model training and GPU reality	Why subprocess isolation exists and how it fits your modularity rules.
A5. Step 6 prediction pools and Jaccard overlap	How pairwise intersection works and how to interpret agreement.
A6. Confidence scores: raw vs calibrated vs normalized	Why you never see "all 1.0" and how to use the three score types.
A7. Autonomy integration points to keep Step 6 composable	Master seeds, CLI patterns, compatibility, and lineage tracking.

A1. Pipeline contract and artifacts

The system is designed around artifacts: outputs that can be validated, hashed, and consumed by later steps and automation. This prevents “magic state” and makes cluster execution reproducible.

Key artifact contracts

Step	Primary script(s)	Produces	Consumed by
1	window_optimizer.py	optimal_window_config.json, bidirectional_survivors.json	Step 2/3 scoring + WatcherAgent evaluatio
2/3	full_scoring_worker.py (+ job scripts)	survivors_with_scores.json (seed + features + score)	Step 5 training
4	meta_prediction_optimizer.py / adaptive_meta_optimizer.py	optimal_meta.json (historically)	Step 5 (may ignore model-specific config w
5	meta_prediction_optimizer_anti_overfit.py	best_model.* + best_model.meta.json	Step 6 prediction_generator.py
6	prediction_generator.py	predictions_*.json (pool + scores)	WatcherAgent + downstream reporting

Rule (Team Beta): no file-extension guessing. Step 6 must load model_type and checkpoint_path only from best_model.meta.json.

A2. Feature system (48 per-seed + 14 global)

Your current training setup uses 48 per-seed features plus 14 global features (GlobalStateTracker). The per-seed features come from SurvivorScorer.extract_ml_features().

The trained model expects features in a fixed order. That order is declared in the sidecar (feature_names) and validated by a feature_schema_hash. This prevents silent column shuffles.

Are these the same 50 features from earlier?

Yes in spirit, with an important change: earlier files carried 50 keys inside the features dict, but two keys (score and confidence) are label-ish fields that can cause leakage. The current contract excludes them, leaving 48 true input features. Global features (14) are added at training and prediction time, giving 62 total inputs.

```
# Sidecar snippet (conceptual)
"feature_schema": {
  "per_seed_feature_count": 48,
  "global_feature_count": 14,
  "total_features": 62,
  "excluded_features": ["score", "confidence"],
  "ordering": "lexicographic_by_key",
  "feature_schema_hash": "..."
}
```

Guardrail: Step 5 must not re-run feature extraction when survivors_with_scores.json already contains precomputed features. Use the dicts as-is.

A3. Label leakage checks and why “$R^2 \approx 1$” can be a trap

A near-perfect R^2 can be legitimate, but in this pipeline it is also a classic symptom of leakage: the label (or a proxy for it) sneaks into the features.

Guardrails you already applied: exclude score/confidence from features, log feature schema hash, and log label ranges (observed_min/observed_max).

Always validate that the trainer is learning from 48 per-seed feature inputs (plus 14 global), not from a hidden score field or from mutated arrays.

A4. Multi-model training and GPU reality (OpenCL vs CUDA)

LightGBM often uses OpenCL on GPU, while PyTorch/XGBoost/CatBoost use CUDA. CUDA initialization can block OpenCL and cause LightGBM error -9999.

The robust fix is subprocess isolation: each trial runs in a fresh subprocess so GPU state is clean. The coordinator stays GPU-neutral and workers import GPU libraries only after parsing args.

```
# Coordinator pattern (conceptual)
# - main process: no torch/lightgbm imports
# - each trial: subprocess trains exactly one model
subprocess.run([
  'python3','train_single_trial.py',
  '--model-type','lightgbm',
  '--data-path','/tmp/trial_data.npz',
  '--params','{...}',
  '--save-model',
  '--model-output-dir','/tmp/trial_models'
])
```

A5. Step 6 prediction pools and Jaccard explained

Step 6 builds a pool of candidate next draws by scoring survivor candidates and aggregating their implied next outputs. Dual-sieve mode intersects forward and reverse survivors to focus on consistent candidates.

Jaccard index

Given forward survivor set A and reverse survivor set B:

```
Jaccard(A, B) = |A ∩ B| / |A ∪ B|
```

Values near 1.0 mean strong agreement (intersection close to union). Values near 0 mean disagreement (small overlap). This is a great health metric for automation gating.

A6. Confidence scores: raw vs calibrated vs normalized

Step 6 now emits three score tracks:

Field	What it is	Use it for
raw_scores	Direct model outputs per candidate	Automation thresholds; cross-run comparability
confidence_scores	Calibrated sigmoid(z-score) values	Readable certainty; avoids saturation
confidence_scores_normalized	Normalized-by-max display scores	Human UI only

Team Beta position: keep raw_scores always; add normalization only as an extra field.

A7. Autonomy integration points to keep Step 6 compatible

To keep the pipeline automation-ready, Step 6 must remain callable from CLI with stable argument names and must emit agent_metadata consistent with the schema.

Item	Expectation
CLI args stable	No renames; add new flags with safe defaults (example: --parent-run-id optional)
Manifest updated	Step 6 manifest should declare new outputs (raw_scores, normalized)
Sidecar loading	best_model.meta.json is single source of truth for model_type + checkpoint_path + schem
Lineage tracking	Step 6 should accept --parent-run-id and also auto-read from sidecar as fallback

WatcherAgent decision policy is intentionally not specified here. This addendum focuses on strong contracts so a future policy can be added safely.