Distributed PRNG Analysis Pipeline (Steps 1-6)

A beginner-friendly guide to what each step does, what files it reads/writes, and how your autonomy scaffolding fits in. Prepared by Team Beta - December 2025.

What’s inside

Section	What you’ll learn
1. Big picture: what the pipeline is doing	Plain-English overview of the goal and why steps 1-6 exist.
2. Step 1: Window Optimizer (find a good “lens” for the data)	What Step 1 searches for and what it writes out.
3. Step 2-3: Feature extraction + scoring (turn seeds into numbers ML can learn)	How raw seeds become feature vectors and labels.
4. Step 4-5: Model training + anti-overfit validation	Train models that find real signal, then validate on holdout data.
5. Step 6: Prediction generator (build weighted pools of next-draw candidates)	Use the trained models to propose candidate next draws.
6. Autonomy and orchestration (what exists today)	How agents, metadata, and the WatcherAgent fit in - decision logic pending.
7. Quick end-to-end sanity check	A minimal checklist you can follow without reading the whole code.

1. Big picture: what the pipeline is doing

This project is a six-step pipeline that takes a history of lottery draws and produces a ranked list of candidate future draws (a “pool”) with confidence-like scores.

Instead of trying to perfectly reverse the internal PRNG state (often impossible), the system learns surface patterns and constraints that are consistent with the observed outputs. The whitepaper calls this functional mimicry: learn what the generator looks like from the outside.

Key idea: Each step produces artifacts (usually JSON files) that the next step consumes. That’s what makes the system modular, testable, and automation-friendly.

Inputs and outputs (high level)

Item	What it is	Typical file(s)
Lottery history	Past draws used as ground-truth context	lottery_history.json (or similar)
Survivors	Candidate seeds that survive sieve filters	bidirectional_survivors.json, survivors_with_scores
Trained model	ML model that maps features → quality score	best_model.* + best_model.meta.json
Prediction pool	Ranked candidate next draws	results/predictions/predictions_*.json

2. Step 1: Window Optimizer (find a good “view” of the data)

Step 1 searches over candidate window settings (how to slice/align the draw history against candidate PRNG hypotheses) to find configurations that produce strong constraint signals.

The main success signal is bidirectional survivors: seeds that survive both forward and reverse constraints. Fewer survivors generally means the observed outputs are more constraining (good).

Outputs you expect

Output	Meaning
optimal_window_config.json	Best window/skip settings found for the current run
bidirectional_survivors.json	Seeds that survive forward+reverse sieve under that config

```
# Typical (conceptual) Step 1 invocation
python3 window_optimizer.py \
  --lottery-history lottery_history.json \
  --output-dir results/step1
```

3. Step 2-3: Feature extraction + scoring (turn seeds into numbers ML can learn)

The ML system doesn’t train on raw seeds. It trains on feature vectors: statistics and constraint measurements computed per seed against the lottery history.

This is done by SurvivorScorer.extract_ml_features() (and batch variants). The result is a JSON list where each item is a survivor seed plus its precomputed features (and a score label).

Important: once features are precomputed, later steps should reuse them (do not re-extract them repeatedly). This prevents wasted time and prevents subtle mismatches.

Output artifact example (shape)

```
{
  "seed": 12345,
  "features": {
    "bidirectional_count": 401,
    "residue_1000_match_rate": 0.33,
    "...": "..."
  },
  "score": 0.2875
}
```

4. Step 4-5: Model training + anti-overfit selection (learn functional mimicry)

Steps 4 and 5 are where you train ML models that predict a per-seed quality score from the feature vector. The objective is not to predict the lottery directly yet; it is to learn a reliable quality function over survivor candidates.

Step 5 (meta_prediction_optimizer_anti_overfit.py) runs Optuna trials and uses k-fold cross validation plus a held-out test set to catch overfitting.

A winning model is saved along with a sidecar file best_model.meta.json that records feature ordering, hash, label range, metrics, and provenance.

```
# Example: Step 5 training run
python3 meta_prediction_optimizer_anti_overfit.py \
  --survivors survivors_with_scores.json \
  --lottery-data synthetic_lottery.json \
  --trials 20 \
  --k-folds 3 \
  --output-dir models/reinforcement
```

5. Step 6: Prediction generator (build weighted pools of next-draw candidates)

Step 6 takes forward survivors, reverse survivors, and the lottery history. It intersects forward and reverse survivors (dual-sieve mode) and builds a prediction pool.

Each candidate in the pool gets a raw model score (the model's output) plus a calibrated confidence-like value for reporting and automation. The pool is sorted and the top-k predictions are returned.

For automation: keep raw scores. Normalized values are helpful for human display, but raw scores are what enable cross-run comparisons and threshold decisions.

```
# Example: Step 6
python3 prediction_generator.py \
  --survivors-forward results/step3/forward_scored.json \
  --survivors-reverse results/step3/reverse_scored.json \
  --lottery-history lottery_history.json \
  --models-dir models/reinforcement \
  --k 10
```

6. Autonomy and orchestration (what exists today)

You already built the scaffolding for autonomy, even though the WatcherAgent decision policy is still in progress.

Component	What it does today
Agent manifests (agent_manifests/*.json)	Declare required inputs, outputs, and parameters per step for automation
Pydantic contexts	Standardize evaluation summaries: success, confidence, interpretation
integration/metadata_writer.py	Injects agent_metadata into result JSON for traceability across steps
WatcherAgent	Runs steps, monitors artifacts/timeouts; decision logic still under development

Note: the dual-LLM architecture (coder + math models) is separate infrastructure used for reasoning/automation. It is not required for the core ML training loop to function.

7. Quick end-to-end sanity check

A minimal end-to-end check is: Step 1 produces survivors, Step 3 produces scored survivors with features, Step 5 trains a model and writes best_model.meta.json, Step 6 produces predictions.

```
# 1) Step 1 (example)
python3 window_optimizer.py --lottery-history lottery_history.json --output-dir results/step1

# 2) Feature extraction / scoring (example)
python3 full_scoring_worker.py --seeds-file results/step1/bidirectional_survivors.json --output-file s

# 3) Step 5
python3 meta_prediction_optimizer_anti_overfit.py --survivors survivors_with_scores.json --lottery-dat

# 4) Step 6
python3 prediction_generator.py --survivors-forward ... --survivors-reverse ... --lottery-history lott
```

Pipeline health rule: if a step outputs “empty” (0 survivors, missing sidecar, etc.), treat it as a real signal. Don’t paper over it with defaults.