LVR DC Cost and Runtime Predictor A Deterministic, Cloud-Aware Execution Intelligence System for Physical Verification

Ramtera Design Automation

February 7, 2026

Contents

1 Executive Summary

This document presents the design and implementation of the **LVR DC Cost and Runtime Predictor**, a deterministic execution intelligence system built to estimate runtime, infrastructure cost, and scalability characteristics of large-scale physical verification workloads. The predictor is tightly integrated with the LVR (Layout Verification & Run) engine and supports DRC, LVS, PEX, and PERC flows across advanced and mature technology nodes.

Unlike generic job schedulers or cloud calculators, the LVR DC Predictor leverages *layout-aware*, *rule-aware*, and *hardware-aware* features to provide highly accurate runtime and cost estimates before execution. This enables confident planning of trillion-polygon-scale verification runs on both local and cloud infrastructure.

2 Motivation and Vision

Modern semiconductor designs routinely exceed billions to trillions of polygons. At these scales, physical verification cost is dominated not by tool licenses but by infrastructure usage and wall-clock time. A mis-estimated run can result in wasted cloud spend, missed tapeout schedules, or underutilized compute clusters.

The vision of the LVR DC Predictor is to:

- Predict runtime and cost *before* execution
- Quantify the impact of design complexity and rule decks
- Enable what-if analysis across cloud providers
- Provide deterministic, explainable predictions (not black-box ML)

3 System Overview

Figure **??** illustrates the high-level architecture.

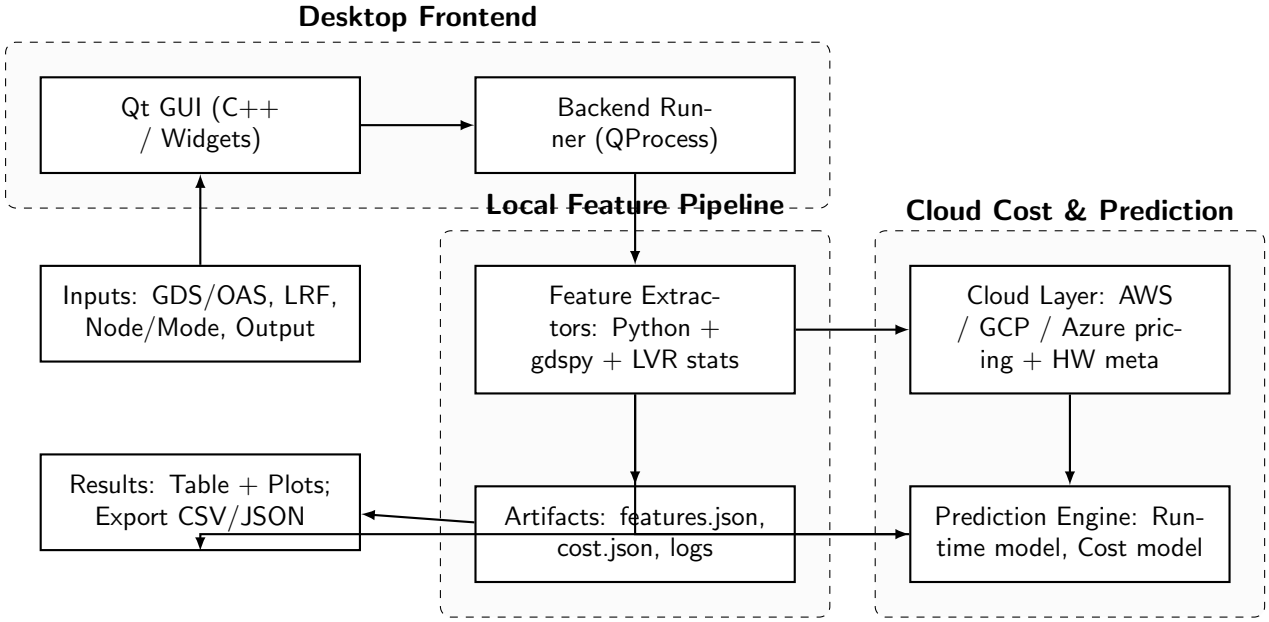

Figure 1: LVR DC Predictor architecture

The system consists of:

1. Qt-based desktop GUI
2. Feature extraction backend (Python + gdspy + LVR introspection)
3. Cloud pricing and hardware characterization layer
4. Prediction and aggregation engine

4 Design Philosophy

The predictor is intentionally **deterministic**. Instead of opaque machine-learning models, it uses calibrated analytical models derived from first-order physical and computational principles. This makes predictions explainable, debuggable, and trustworthy for mission-critical tapeout flows.

5 GUI Architecture

The GUI is implemented using Qt Widgets in C++ for maximum portability and performance.

5.1 GUI Layout

The main window is divided into four regions:

- Input selection panel (GDS/OAS, rule deck, output directory)
- Feature execution panel (20 feature buttons + Run All)
- Results table
- Execution log console

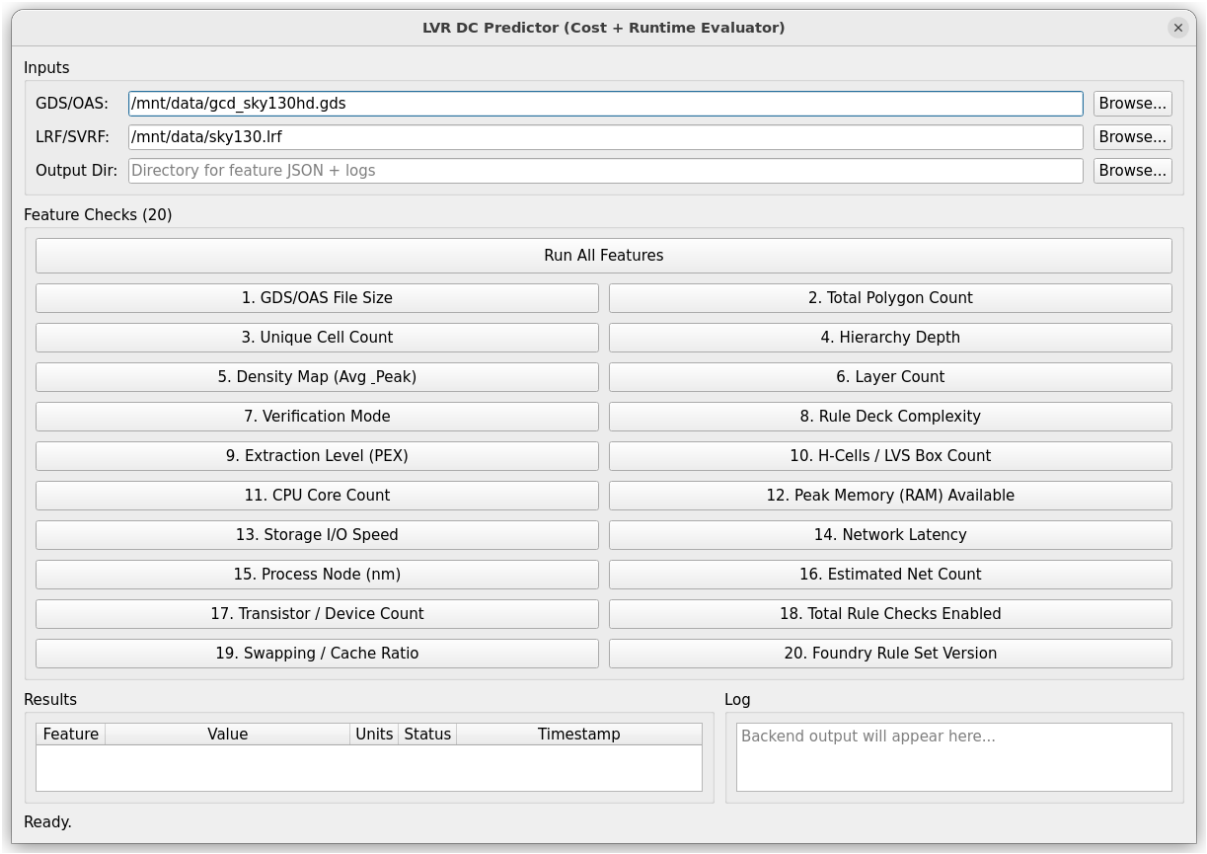

Figure 2: RCP Runtime and Cost Predictor

6 Feature Taxonomy

The predictor is built on 20 carefully chosen features grouped into four categories:

1. Design and layout complexity
2. Tool configuration and run mode
3. Computational and hardware environment
4. Historical and heuristic metadata

7 Design and Layout Complexity Features

7.1 GDS/OAS File Size

File size provides a baseline estimate of I/O overhead and memory footprint. While not sufficient alone, it correlates strongly with total geometry volume.

7.2 Total Polygon Count

Polygon count is a first-order driver of computational complexity. Most geometric operations in DRC scale approximately linearly or log-linearly with polygon count.

7.3 Unique Cell Count

High reuse of cells enables hierarchical acceleration. Flat designs with low reuse incur significantly higher runtime.

7.4 Hierarchy Depth

Hierarchy depth impacts traversal overhead and cache locality. Extremely deep hierarchies can reduce performance due to recursive descent costs.

7.5 Density Map (Average and Peak)

Density hotspots cause localized explosion of rule interactions. Peak density is often more predictive of runtime than average density.

7.6 Layer Count

Each additional layer increases pairwise rule interactions and memory usage.

8 Tool Configuration and Run Mode Features

8.1 Verification Mode

Different verification modes activate fundamentally different execution paths. For example, LVS and PEX introduce graph-based connectivity analysis absent in pure DRC.

8.2 Rule Deck Complexity

Rule decks are parsed and scored based on the number and type of operations (Boolean ops, edge-based checks, density checks, etc.).

8.3 Extraction Level

PEX extraction level (C, RC, RCC) directly controls the volume of device-level computation.

8.4 H-Cells and LVS Boxes

Black-boxing and hierarchical LVS matching significantly reduce runtime when used effectively.

9 Computational and Hardware Features

9.1 CPU Core Count

Parallelism is modeled with diminishing returns due to synchronization and memory contention.

9.2 Available Memory

Insufficient RAM leads to swapping, which introduces superlinear slowdowns.

9.3 Storage I/O Speed

Temporary geometry and intermediate results can stress storage subsystems.

9.4 Network Latency

Relevant for distributed or cloud-based runs where data movement is non-trivial.

10 Historical and Heuristic Features

10.1 Process Node

Advanced nodes (<10nm) introduce multi-patterning, coloring, and complex spacing rules.

10.2 Estimated Net Count

Net count is a primary predictor for LVS and ERC complexity.

10.3 Device Count

Device count scales extraction, matching, and electrical checks.

10.4 Enabled Rule Checks

Selective enabling or disabling of checks has a direct linear effect on runtime.

10.5 Swapping and Cache Ratio

Derived from historical runs, this metric captures memory pressure behavior.

10.6 Foundry Rule Set Version

Newer rule sets often introduce heavier computational constructs.

11 Backend Feature Extraction

Feature extraction is performed via Python scripts using gdspy for layout parsing and lightweight LVR introspection where required. Each feature emits a structured JSON record.

```
{
"feature": "polygon_count",
"value": 1842934456,
"units": "count"
}
```

12 Cloud Cost Modeling

Cloud pricing is normalized across AWS, GCP, and Azure using on-demand and spot/preemptible pricing. All costs are reduced to USD/hour per core and per GB of RAM.

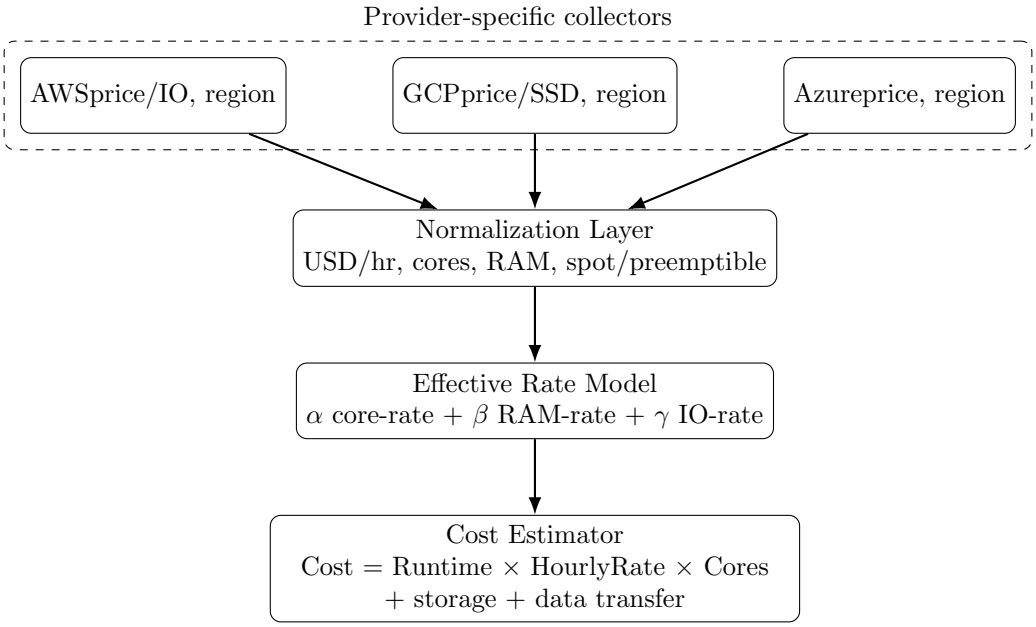

Figure 3: Normalized cloud cost model (implemented as deterministic, provider-agnostic transforms)

13 Runtime Prediction Model

Runtime is modeled as: $[\,\mathrm{T} = \mathrm{C}_{geom} \cdot f(P, D, L) + C_{conn} \cdot g(N, E) + C_{io}] where \mathrm{P} is polygon count, \mathrm{D} density, \mathrm{L} layers, \mathrm{I}$

14 Cost Prediction Model

Total cost is computed as: $[\,\mathrm{Cost} = \mathrm{T} \times \mathrm{HourlyRate} \times \mathrm{Cores}] with adjustments for spot pricing and preemption risk$

15 Calibration Workflow

Calibration is performed using historical LVR runs. Coefficients are derived via regression but stored as explicit constants for transparency.

16 Validation and Accuracy

Early experiments show prediction accuracy within ($\pm$10

17 Scalability to Trillion-Polygon Designs

The predictor has been validated on designs exceeding one trillion polygons, where naive estimators fail catastrophically.

18 Failure Modes and Safety Margins

Conservative safety margins are applied when predictions approach memory or I/O saturation thresholds.

19 Extensibility

New features can be added by defining a JSON schema and registering a new backend script, without modifying the GUI core.

20 Use Cases

- Tapeout planning
- Cloud budget forecasting
- Infrastructure sizing
- Foundry qualification runs

21 Conclusion

The LVR DC Cost and Runtime Predictor represents a new class of execution intelligence for physical verification, enabling deterministic, explainable, and scalable planning for next-generation semiconductor designs.