DC Launcher

Distributed Compute Launcher for LVR

Ramtera Design Automation

February 6, 2026

Contents

1 Introduction

DC Launcher (DCL) is a production-grade distributed execution controller designed to safely launch, monitor, and analyze large-scale LVR workloads across public cloud environments. It is not a wrapper script or a thin GUI front-end; it is a deterministic orchestration system with explicit state, validation, and failure control.

Physical verification workloads such as DRC, LVS, and PERC are both compute-intensive and cost-sensitive. A single incorrect launch can result in hours of wasted runtime and significant cloud expenditure. DC Launcher addresses this problem by enforcing a strict execution sequence that prevents illegal or unsafe operations.

The launcher is implemented using Qt and C++ for the GUI and orchestration layer, combined with a shell-based provider abstraction layer. This hybrid approach ensures high performance, portability, and debuggability.

2 System Architecture

DC Launcher follows a layered architecture that separates user interaction, execution control, and cloud-specific behavior. This separation is intentional and is critical for long-term maintainability.

2.1 Architectural Layers

- GUI Layer (Qt Widgets)
- Execution Controller (C++)
- Provider Abstraction Layer (Shell Scripts)
- Cloud Infrastructure

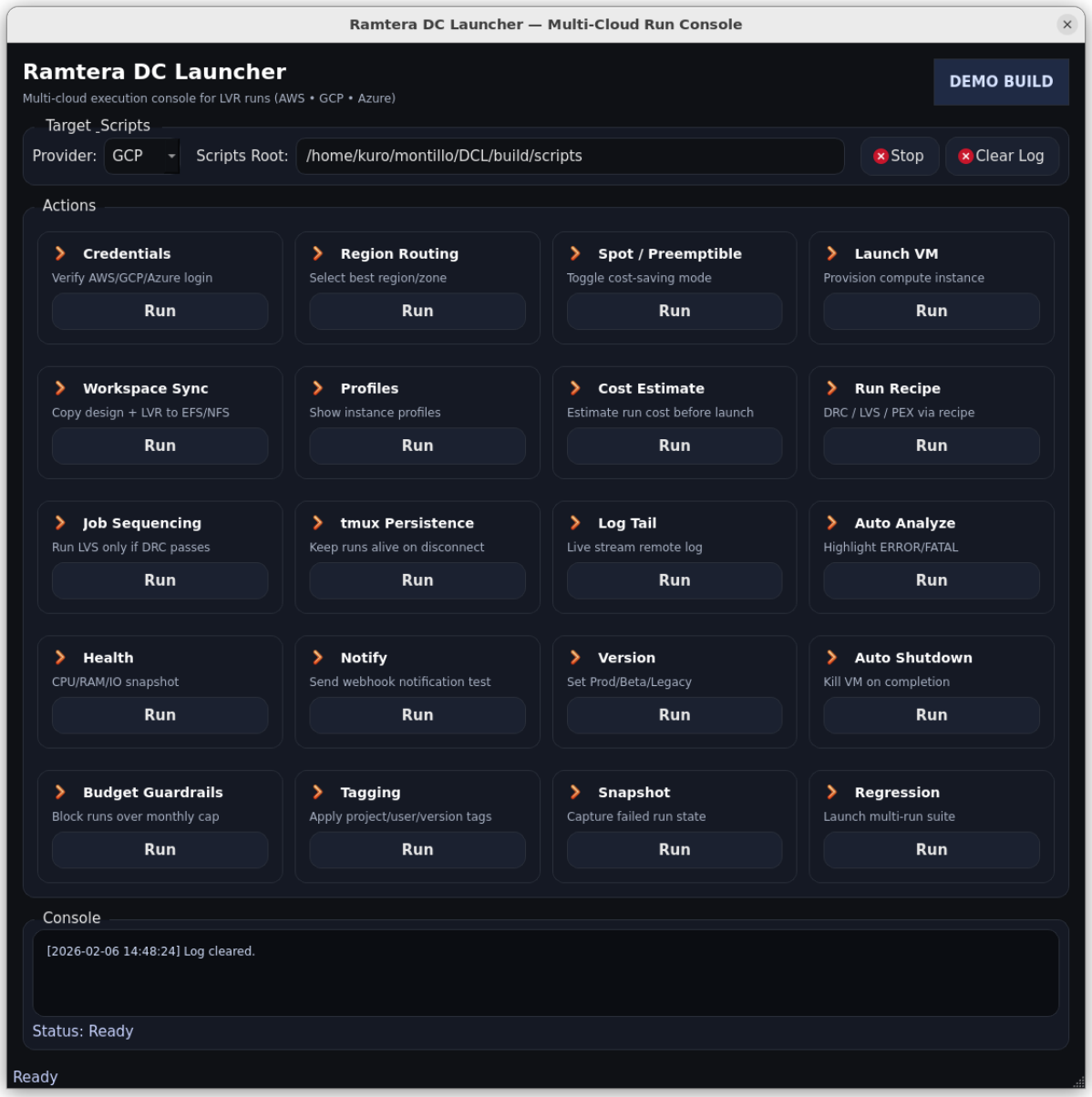
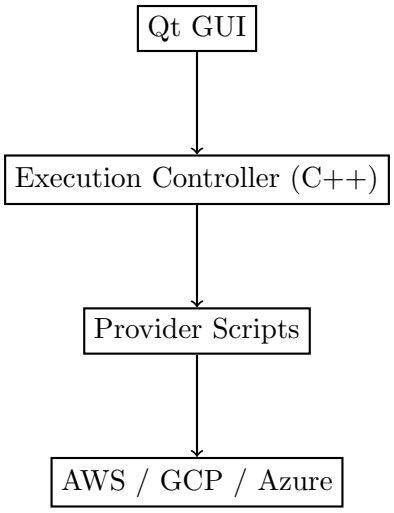


Figure 1: DC Launcher

2.2 Architecture Diagram



This architecture ensures that cloud provider changes never require GUI modifications, and execution logic remains stable across environments.

3 Cloud-Agnostic Execution Model

DC Launcher is designed to be completely independent of any specific cloud provider. The GUI and execution controller contain zero cloud SDKs, APIs, or provider-specific logic.

All cloud-dependent behavior is implemented through provider scripts that adhere to a strict execution contract. As a result, switching from AWS to GCP or Azure does not alter the execution semantics, validation logic, or failure handling behavior.

This approach enables enterprises to compare cost, performance, and availability across cloud providers without retraining engineers or rewriting automation workflows.

3.1 Execution Contract

Phase	Description
Validation	Verify credentials, quotas, environment variables
Cost Estimation	Compute projected cost before provisioning
Launch	Provision compute and start LVR job
Monitoring	Stream logs and block until completion
Analysis	Parse results and generate reports

4 Cloud-Agnostic Execution Model

DC Launcher is designed to be completely independent of any specific cloud provider. The GUI and execution controller contain zero cloud SDKs, APIs, or provider-specific logic.

All cloud-dependent behavior is implemented through provider scripts that adhere to a strict execution contract. As a result, switching from AWS to GCP or Azure does not alter the execution semantics, validation logic, or failure handling behavior.

This approach enables enterprises to compare cost, performance, and availability across cloud providers without retraining engineers or rewriting automation workflows.

4.1 Execution Contract

Phase	Description
Validation	Verify credentials, quotas, environment variables
Cost Estimation	Compute projected cost before provisioning
Launch	Provision compute and start LVR job
Monitoring	Stream logs and block until completion
Analysis	Parse results and generate reports

5 Environment-Driven Configuration

All runtime configuration in DC Launcher is expressed as environment variables. This design ensures stateless execution and makes every run reproducible and debuggable.

5.1 Exported Variables

```
RECIPE=lvr_drc
RUN_PROFILE=large
LVR_VERSION=2026.1
PROJECT_TAG=asap7_demo
USER_TAG=sameer
RUNTIME_HOURS=6
```

Because these variables are exported explicitly, any execution can be reproduced outside the GUI by sourcing the same environment.

6 Deterministic State Machine

DC Launcher enforces correctness using a persistent state machine. Each execution phase transitions the launcher into a new state, which is recorded on disk.

6.1 State Persistence

`~/.ramtera/dclauncher/state/<provider>.env`

6.2 State Transitions

State	Trigger	Next State
INIT	GUI Load	CONFIGURED
CONFIGURED	Validation OK	VALIDATED
VALIDATED	Cost Approved	READY
READY	Launch	RUNNING
RUNNING	Completion	ANALYZE
ANALYZE	Parsed	DONE

7 Pre-flight Validation Engine

Before any cloud resource is created, DC Launcher performs a comprehensive pre-flight validation. This includes environment completeness, credential verification, runtime bounds, and provider availability.

Validation failures are surfaced synchronously in the GUI and block further execution until resolved. This eliminates accidental launches with incomplete or invalid configuration.

8 Cost Estimation and User Acknowledgement

DC Launcher enforces a mandatory cost estimation phase prior to job launch. Estimated cost is calculated using runtime duration, instance profile, and provider pricing data.

Execution cannot proceed until the user explicitly acknowledges the estimated cost. This design prevents accidental cloud spend and provides clear financial visibility.

9 Blocking Execution Semantics

Execution steps in DC Launcher are strictly ordered. Monitoring scripts block until the remote job has completed, ensuring that analysis is never performed on partial data.

This guarantees consistent execution semantics across all runs.

10 Real-Time Log Streaming

Logs are streamed from the remote execution environment and displayed locally in real time. This allows users to detect failures early and abort runs if necessary, minimizing wasted compute time.

11 Auto-Analyze Engine

After job completion, DC Launcher automatically parses execution logs and metadata to generate a structured summary.

Metric	Description
Wall Time	Total runtime
CPU Hours	Aggregate CPU usage
Peak Memory	Maximum memory observed
Exit Status	Success or classified failure

12 Resume and Replay Capability

DC Launcher is designed with the assumption that long-running distributed jobs may be interrupted due to user actions, network issues, GUI restarts, or system crashes. To address this reality, the launcher supports resume and replay semantics driven entirely by persisted state.

When the GUI starts, it loads the most recent provider-specific state file and reconstructs the execution context. Based on the recorded state, the launcher determines which actions are safe to resume and which must remain disabled. This prevents duplicated launches and accidental resource creation.

Replay functionality allows users to re-run a previous execution with the same parameters by reusing the stored environment and configuration. This is particularly valuable for regression

testing, cost comparison across providers, and performance benchmarking of different LVR versions.

13 Failure Isolation and Containment

DC Launcher treats failures as first-class events rather than exceptional edge cases. Any failure occurring during validation, cost estimation, launch, monitoring, or analysis is immediately captured and classified.

Failures are fully isolated within the current execution context. No subsequent steps are allowed to proceed until the failure is acknowledged by the user. This containment strategy ensures that partial execution paths do not propagate inconsistent state or leave cloud resources running.

By isolating failures at well-defined phase boundaries, DC Launcher enables clear root-cause analysis and deterministic recovery.

14 Provider Isolation Model

Each cloud provider supported by DC Launcher is fully isolated through its own script set and execution context. Provider-specific logic is never shared directly across providers, which eliminates the risk of accidental behavioral coupling.

This isolation allows provider implementations to evolve independently. For example, authentication changes or pricing model updates in one cloud do not affect other providers or the GUI.

From a maintenance perspective, this design dramatically reduces regression risk and simplifies validation when adding new providers.

15 Script Contract and Enforcement

DC Launcher enforces a strict script contract that all providers must follow. Scripts must be non-interactive, environment-driven, and deterministic.

Each script is expected to:

- Consume configuration exclusively via environment variables
- Exit with a non-zero status on failure
- Produce machine-parseable output where applicable

This contract allows the execution controller to reason about script behavior without requiring provider-specific knowledge, enabling consistent control flow and error handling.

16 Stateless Execution Design

All provider scripts used by DC Launcher are stateless by design. They do not maintain internal persistent data beyond the execution state explicitly managed by the launcher.

Statelessness simplifies debugging, as scripts can be executed manually outside the GUI with the same behavior. It also enables integration with external automation frameworks such as CI/CD pipelines or batch schedulers.

This design philosophy aligns with best practices for large-scale distributed systems.

17 Safe Resource Teardown

DC Launcher guarantees that cloud resources are never left running due to partial or failed executions. Teardown is performed in a controlled and ordered manner after job completion or failure.

By centralizing teardown logic and enforcing execution ordering, the launcher prevents orphaned instances and unexpected cloud spend. This is a critical requirement for enterprise environments with strict cost governance.

18 Multi-User Safety Model

DC Launcher is designed to operate safely in environments where multiple users may share cloud accounts or infrastructure quotas. User-specific identifiers are embedded into execution tags and state files.

This prevents accidental interference between concurrent executions and enables clear attribution of cloud usage. It also supports auditing and cost allocation across teams.

19 Integration with LVR Recipes

DC Launcher is tightly integrated with LVR execution recipes and runtime profiles. Recipes define the type of verification workload, while profiles control resource sizing and execution behavior.

The launcher validates recipe and profile compatibility before execution, ensuring that invalid combinations cannot be launched. This reduces user error and improves overall reliability.

20 CI/CD and Automation Friendliness

Although DC Launcher provides a full graphical interface, its execution model is fully automation-friendly. All functionality exposed through the GUI can be invoked via scripts using the same environment variables.

This enables integration with continuous integration pipelines, nightly regression runs, and automated benchmarking workflows without modification to the core system.

21 Enterprise Readiness

DC Launcher is engineered for production use in enterprise environments. Its explicit state model, deterministic execution flow, and strong failure containment make it suitable for mission-critical verification workloads.

The launcher complements LVR by providing operational control, cost safety, and reproducibility at scale. Together, they form a complete solution for distributed physical verification.