2026 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SANTA CLARA, CA, USA
MARCH 2 - 5, 2026

Database Driven RTL Simulation: Fighting Billion-Gate Verification Bottlenecks

Victor Besyakov
BTA Design Service, Ottawa, Canada

BTA DESIGN SERVICES
your project, our team

Agenda

Abstract: Modern AI chips exceed multi-billion gates. At this scale, verification generates terabytes of data, and traditional simulation tools struggle to handle it efficiently. The open-source library SVDB Gateway mitigates those challenges by offloading verification data to external databases.

- Introduction & Industry Context
- Current EDA Approaches & Trade-offs
- Motivation for Database Offload
- SVDB Gateway Solution
- Use Cases & Applications
- Conclusion

Intro & Industry Context

ASIC architectural trends

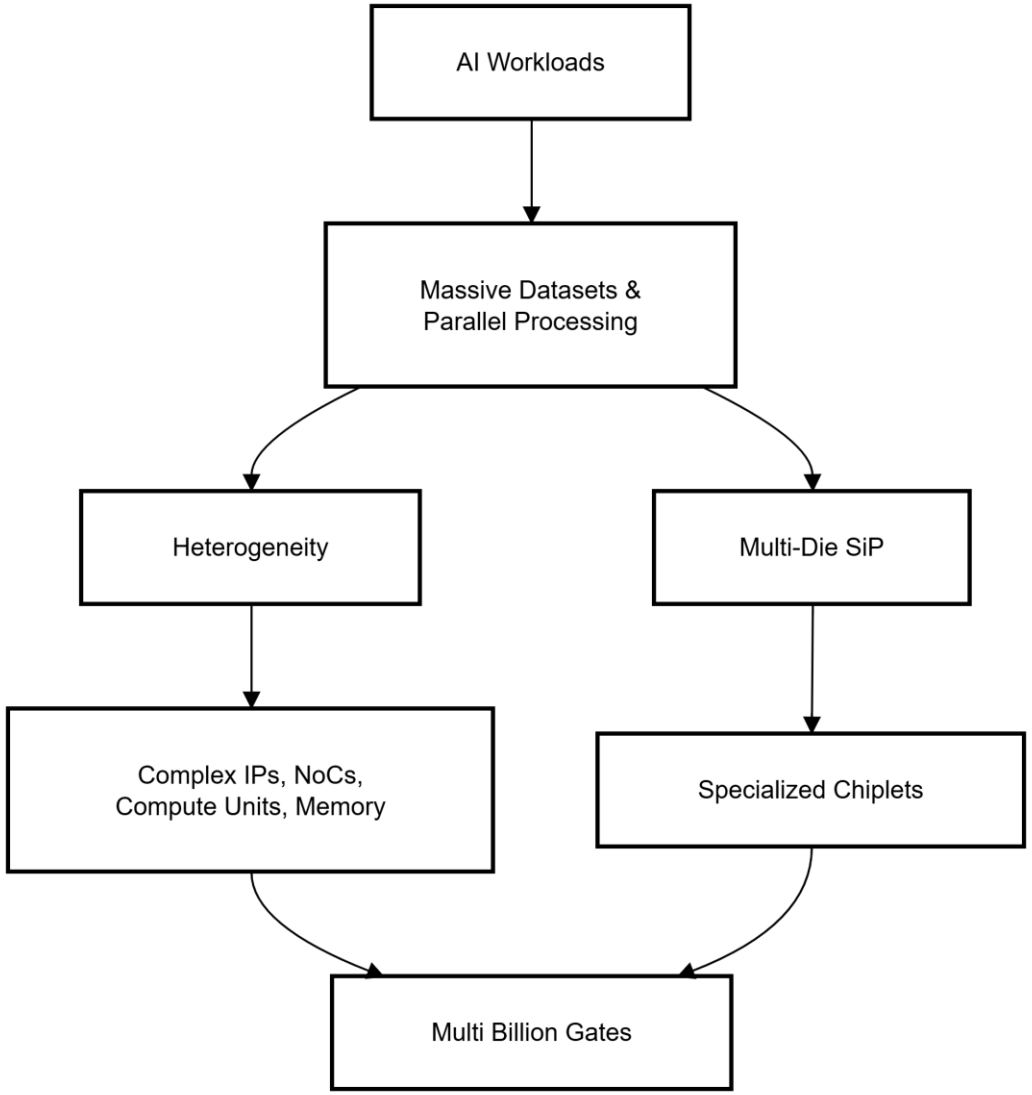
Verification Bottlenecks

Current Approaches & Trade-offs

ASIC Architectural Trends

What is driving AI chips into the multi-billion gate range?

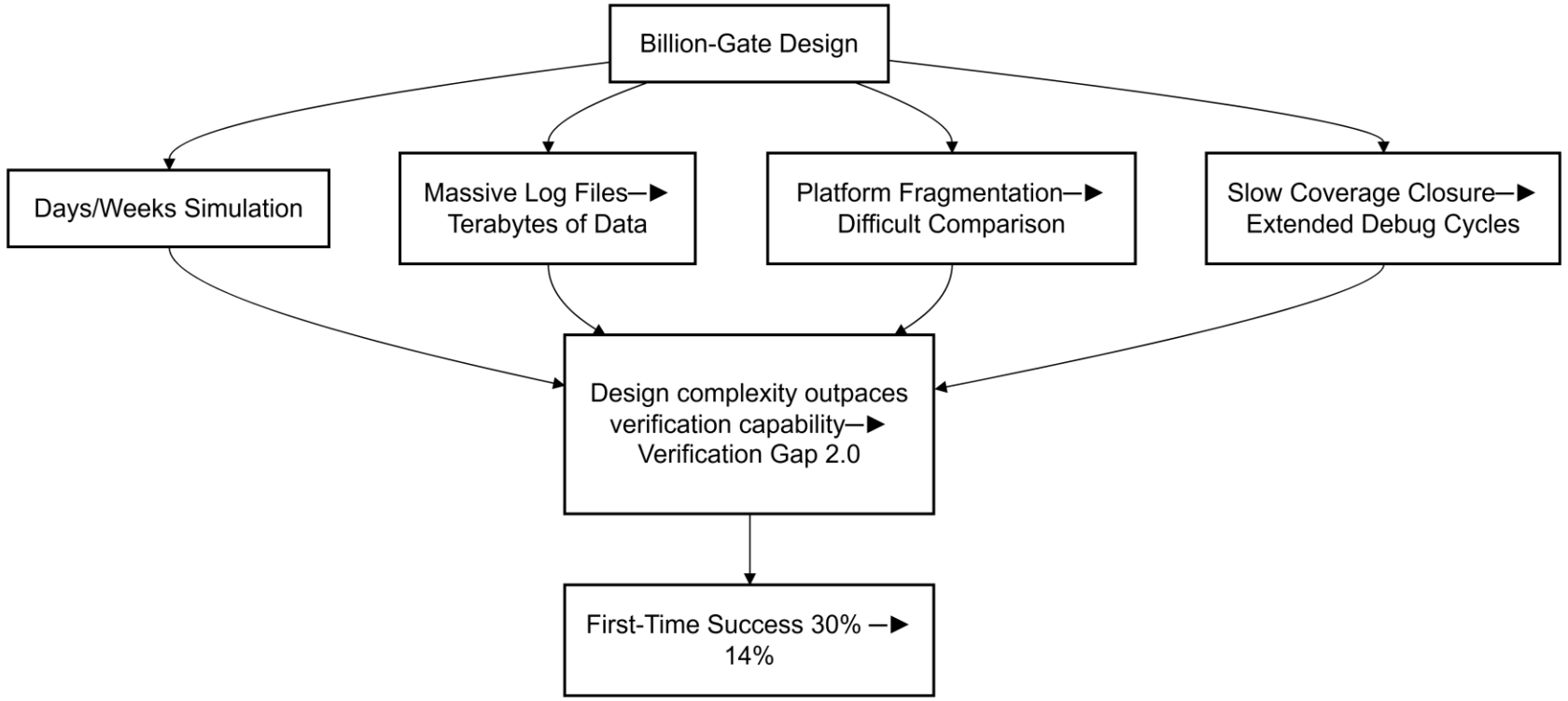
- **Training and Inference workload:**
  - AI chips must process massive datasets.
  - Higher throughput demands more parallel processing per clock cycle
- **Heterogeneity:**
  - 3rd-party high-speed communication IPs.
  - Complex NoCs.
  - Custom compute units.
  - Multi-level memory systems (on-chip & off-chip)
- **Multi-Die System-In-package (SiP):**
  - To achieve design reuse and a faster time-to-market, chiplet integration replaces large single dies with smaller, specialized tiles.



Verification Bottlenecks

For Multi-Billion Gate Designs:

- **Regression runtime:**
  - *Before:* Regression that once took hours
  - *Now:* take days or even weeks.
- **Data Volume:**
  - Regression generates terabytes of logs/waveforms (unsearchable).
- **Fragmentation:**
  - Modern verification employs multiple platforms from different vendors (RTL Sim vs Emulation vs FPGA).
  - Proprietary log and waveform formats.
- **Coverage Closure:**
  - Coverage Closure slows significantly



Highlights: Identifying failures often requires manual detective work

Current Approaches & Trade-offs

RTL simulation, cloud, hardware-assisted verification

UVM & SystemVerilog limitations

Current tools capabilities

RTL Simulators

- Multi-threading (2–4X faster)
- Incremental compilation (2-3X shorter turnaround)
- Checkpoint/restart mechanism (multi-day reruns into sub-hour restarts)
- Distributed simulation (3.2x speedup)

Cloud Simulation

- Elastic compute scaling without capital investment
- Access to high-end EDA-optimized machines
- No queue contention on a limited internal farm

Hardware-assisted Verification

- 50x-1000x speedups possible
- Early full-system validation
- HW/SW co-verification
- Closer timing behavior than pure simulation

UVM

- Standardized methodology
- Vertical & Horizontal Reuse
- Large VIP ecosystem.

SystemVerilog

- Single environment for RTL and TB
- Native support for :
  - Parallel processes, mailboxes, semaphores, and events
  - Assertion (SVA)
  - Constrained Random Verification
  - Functional Coverage

Highlights: These are proven and powerful capabilities. Why are they not scaled for the billion-gate system?

RTL simulation

- **Multi-threading:**
  - Fine-grained parallelism faces fundamental limits due to synchronization overhead and strict event-ordering requirements.
  - Large design simulation suffers from memory latency and cache contention across cores. As a result, speedup does not scale linearly.
- **Incremental compilation:**
  - While reducing build times for small changes, this approach still struggles with full-chip compilation overhead.
- **Checkpoint/restart mechanism:**
  - For designs with billions of gates, writing and reading checkpoints introduce significant I/O overhead, which slows down the save and restore processes significantly.
- **Distributed simulation :**
  - Network latency and synchronization can reduce performance when cross-partition traffic is high.
  - Complex NoCs make it difficult to define “efficient” partition boundaries and minimize cross-partition communication.

Cloud Simulation

The cloud provides flexibility and eliminates capital investment, but careful cost management is essential

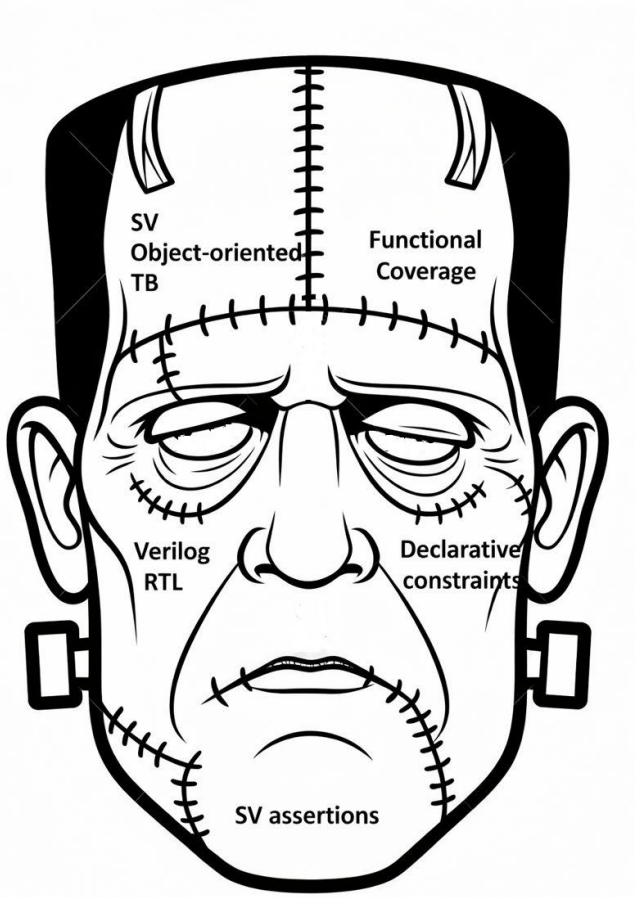
- **Operational/Performance/Expertise**
  - Requires EDA expertise + extra cloud capacity for trials, delaying schedules and raising expenses
- **Data Egress/Transfer Costs**
  - High egress fees for logs/waves: RTL simulations generate TBs of data (waveforms, logs), adding 15%-50% to the monthly bill.
- **License & Resource Scaling**
  - EDA license costs dominate: Licenses (often 80%+ of bill) scale linearly with cloud instances

Highlights: Untuned runs make the cloud more expensive than on-premises solutions for low-utilization workloads. Efficient scaling requires deep EDA expertise and extra capacity for tuning trials.

Hardware-assisted Verification

- **Cost of Ownership**
  - Upfront multi-million-dollar initial investment in hardware.
  - Hardware emulation systems have a total cost of ownership 3–5 times the purchase price over 3–5 years, driven by installation, maintenance, support, power, cooling, and downtime from failures.
  - Alternative: Use cloud-based hardware-assisted verification, but capacity and availability are limited.
- **Compile/Partitioning Overhead**
  - FPGA-based prototyping and processor-based emulators face long compile times due to partitioning complexity.
- **Debug/Observability Limitations**
  - Extensive design probing reduces max frequency and makes full-signal visibility impractical in FPGA-based hardware verification.
- **Vendor/Operational Lock-in**
  - Hardware-assisted verification requires specialized expertise for setup/maintenance, with high initial investment and operational complexity.
  - Different vendors' hardware-assisted platforms are incompatible, which significantly increases the risk of vendor lock-in.

SystemVerilog



- **Not designed as a general-purpose language**
  - Stitches together 5 different sub-languages (RTL, object – oriented verification, assertions, coverage, constraints).
  - Limited optimization and abstraction features compared with C++/Python.
  - No standard library for data structures, algorithms, or I/O.
  - Event-driven simulation imposes performance penalties. Equivalent C++ models often run orders of magnitude faster.
  - Interfacing with external software ecosystems is challenging, even with DPI-C bridges.
- **Simulation-centric architecture**
  - Data exists in the simulator memory.
  - No persistence beyond simulation runtime.

UVM

UVM trade-offs are tied directly to SystemVerilog specifics.

UVM remains an excellent and widely adopted methodology, but it faces real scaling challenges at billion-gate complexity.

Key infrastructure bottlenecks examples:

- **Configuration Database (config_db) Performance**
  - **Slow string-based lookups:** UVM config_db employs string matching with wildcards, leading to significant delays during the build phase for thousands of set/get operations.
  - **High memory overhead:** config_db stores all parameters in-memory with hierarchical scope matching.
  - **Runtime overhead for object creation:** UVM Factory enables overrides but adds significant CPU cost for every transaction/component instantiation in large testbenches, slowing simulation.
- **RAL (Register Abstraction Layer) Scalability**
  - **Massive memory for large designs:** For designs with >100K registers, standard UVM RAL creates a class object per register/memory element, causing "out of space" errors and huge memory consumption.
  - **Performance penalty for large RAL:** Modelling large memories/registers (>100K) with RAL leads to huge performance penalties in simulations due to object instantiation overhead.

Highlights: Event-driven, in-memory simulation model worked well for smaller designs but hit hard limits at the modern SoC/SiP scale.

Motivation for DB Offload

Why simulator-centric data handling breaks down

Motivation for Database Offload

Why Database Offload?

- **Problem**
  - Simulators hold huge register models, coverage data, and configurations entirely in RAM. When memory is full, the OS begins disk paging, slowing simulations.
  - Post-simulation debug means grepping through terabytes of text logs. If the data you need isn't in the logs, you have to rerun the entire simulation again. To find a specific transaction chain from component X between times A and B can take hours.
  - Each simulator/hardware-assist verification platform uses its own data format, preventing cross-tool analysis and standardization.
  - There's no simple way to answer: "What's still untested across all platforms?“
- **Solution**
  - Offload data storage to an external database engine designed for the job.
  - It’ll keep the simulator lean.
  - Replace grep with SQL queries.
  - Unified fragmented platforms under one schema.
- **Result**
  - Faster analysis, lower memory pressure, persistent data storage.

Problem		Solution
In-memory data storage	——▶	External database storage
Sequential file dumps	——▶	Indexed queries
Memory pressure / paging	——▶	Lean simulation process
Grepping large files	——▶	SQL SELECT statements
Single-run data	——▶	Persistent cross-run data
Platform fragmentation	——▶	Unified schema

Traditional approaches optimize computation and execution.
Database offload optimizes data management.

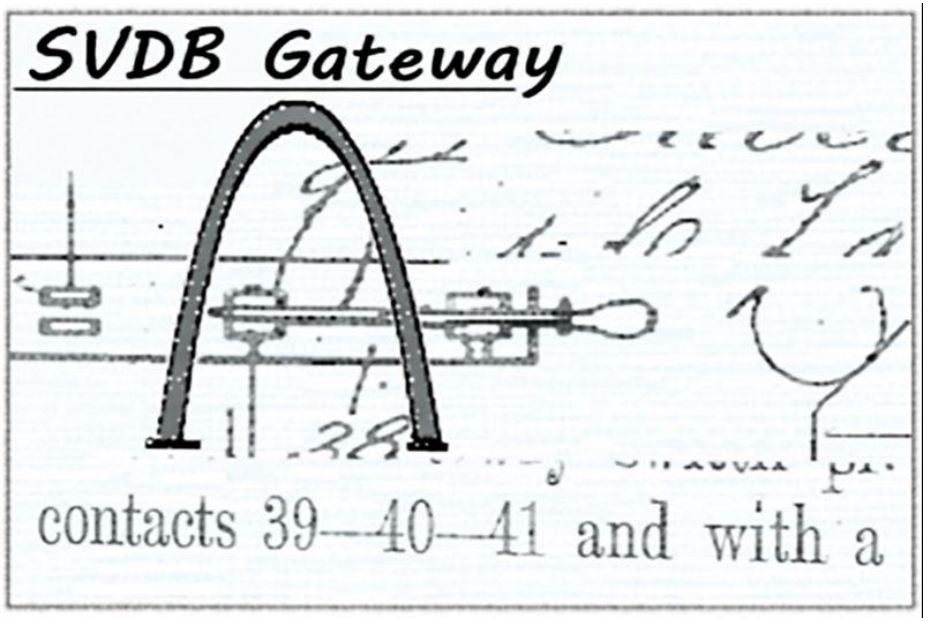
SVDB Gateway Solution

Architecture overview

DPI-C + SQLite integration

Overview

- SVDB Gateway bridges SystemVerilog verification environments and SQLite databases through DPI-C.
- Open-source DPI-C library (MIT License)
- It works with all major commercial simulators: VCS, Xcelium, Questa and with open-source tools like Verilator.
- github.com/xver/svdb_gateway



Highlights: Existing tools use databases internally but keep them proprietary.

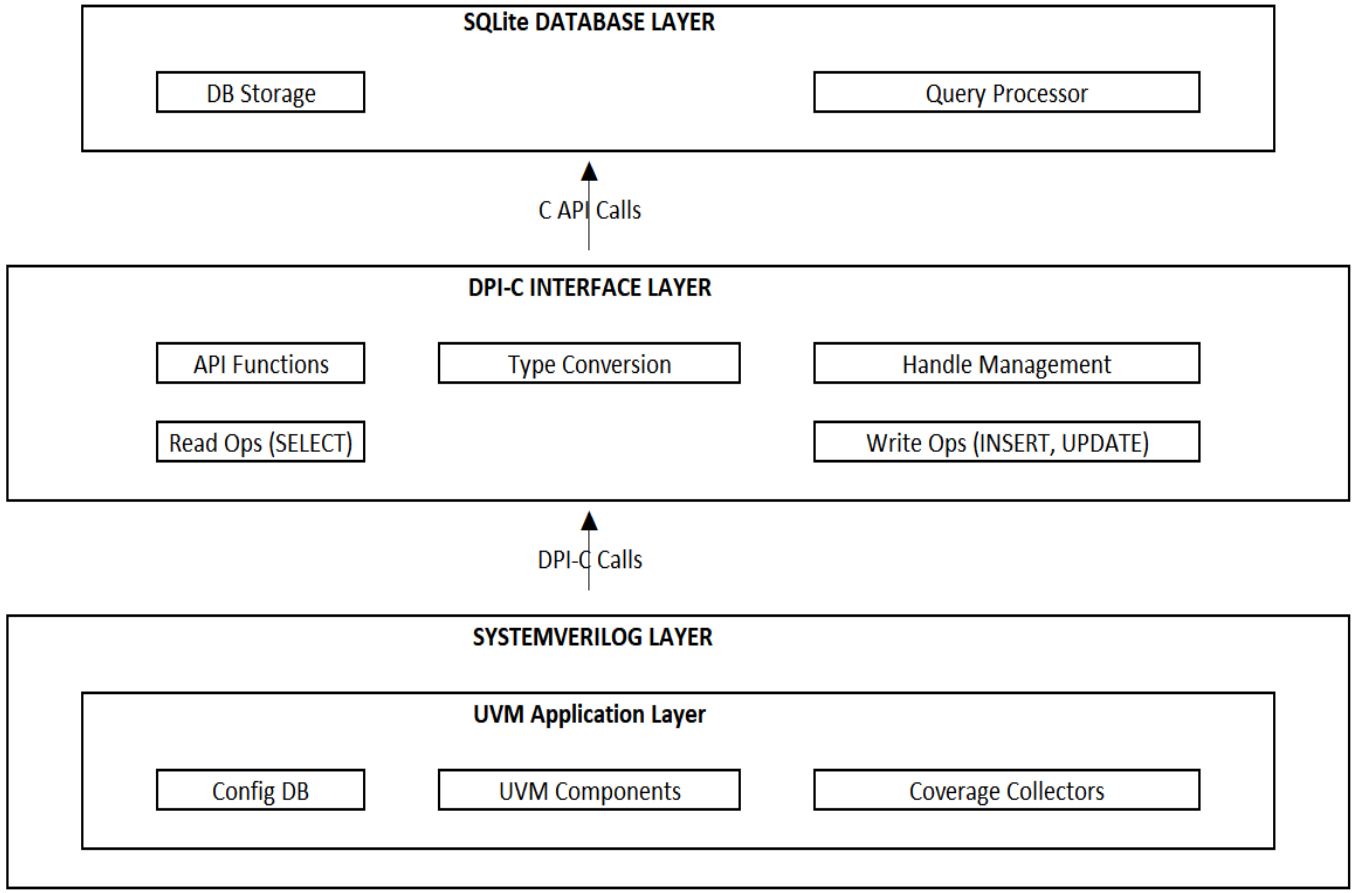
SVDB opens this up: it exposes the database directly, giving users full control.

SVDB three-Layer Architecture

- **SV/UVM:** Standard UVM code runs with minimal changes. The testbench makes simple API calls via SV/UVM wrapper classes that hide the underlying database mechanics
- **DPI-C:** provides API functions that map high-level operations (read, write, insert, update) to SQLite C API calls
- **SQLite:** production-grade, zero-configuration embedded database.

This architecture delivers:

- **Clean layer separation** (each layer has one job),
- **Simulator independence** (any simulator supporting DPI-C works)
- **Lightweight operation (**no servers or daemons to manage)
- **Portable database (**works on any OS platform)



SVDB API Functions

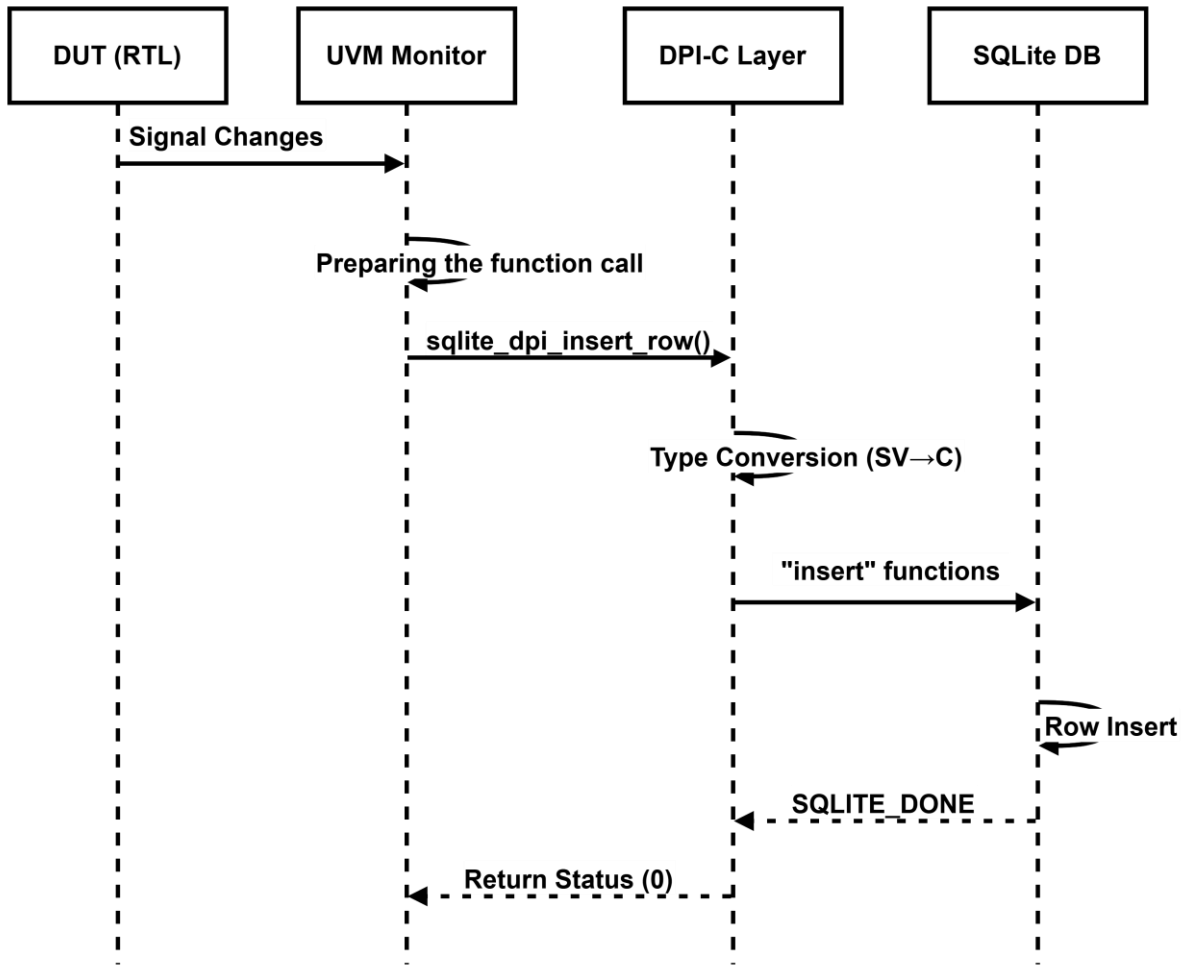
Category	Function Name	Purpose	Data Flow Direction
Connection Management	*sqlite_dpi_open_database()*	Establish database connection	SV → C → SQLite
	sqlite_dpi_close_database()	Close database connection	SV → C → SQLite
Write Operations	*sqlite_dpi_insert_row()*	Insert data into table	SV → C → SQLite
	sqlite_dpi_execute_query()	Execute SQL statement	SV → C → SQLite
Read Operations	*sqlite_dpi_get_row()*	Retrieve complete row	SQLite → C → SV
	sqlite_dpi_get_cell_value()	Retrieve single cell value	SQLite → C → SV
	sqlite_dpi_get_rowid_by_column_value()	Find row ID by value	SQLite → C → SV
Transaction Control	*sqlite_dpi_begin_transaction()*	Start transaction	SV → C → SQLite
	sqlite_dpi_commit_transaction()	Commit changes	SV → C → SQLite
	sqlite_dpi_rollback_transaction()	Revert changes	SV → C → SQLite

Highlights: Simple, high-level API with wrappers for common cases. This is open-source, you have the freedom to add any functions you need for additional functionality

SVDB Operation: Write & Read

Write Operation Sequence

- **DUT Signal Capture:** DUT signal changes —► UVM monitor samples interface signals
- **Data Formatting:** UVM monitor preparing data for DPI-C function calls.
- **DPI-C Boundary Crossing:** *sqlite_dpi_insert_row()* passing database handle, table name, columns, and values. Type conversion occurs automatically.
- **SQL Execution & Storage:** C layer constructs SQL INSERT statements by calling C SQLite API functions: *sqlite3_prepare_v2(), sqlite3_bind_text(), sqlite3_last_insert_rowed()* —► SQLite make writes to disk and returns status code back to SV
- **Batch Optimization/Transaction Control:** Multiple inserts grouped into a single transaction, achieve 50K–100K ops/second throughput (80–100× over individual commits)



Read Operation Sequence

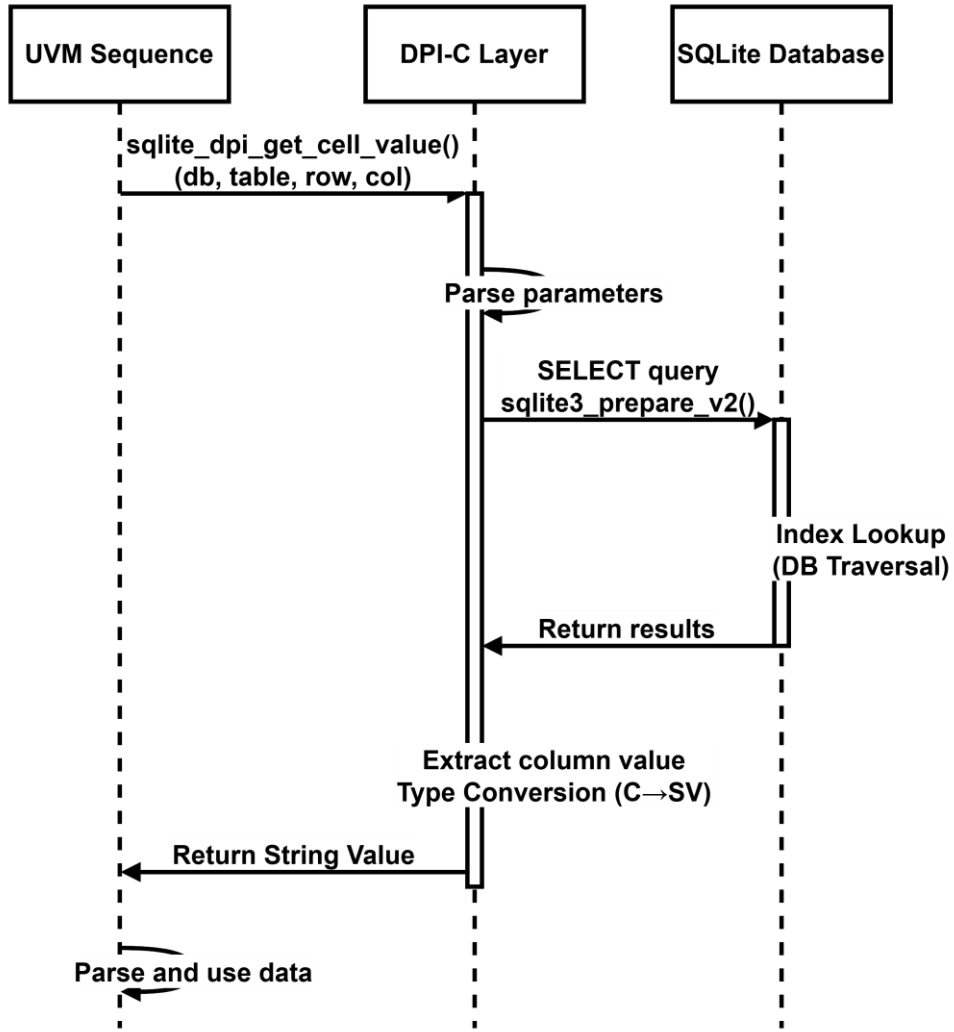
- **Query Initiation:** SV code calls DPI-C *sqlite dpi_get_cell_value()* specifying DB handler, table name, row ID, and column —► function crosses DPI-C boundary.
- ***SELECT:*** C layer constructs targeted query by calling C SQLite API functions: *sqlite3_prepare_v2(), sqlite3_bind_int(), sqlite3_step(), sqlite3_finalize()*
- ***I*ndex-Based Lookup:** SQLite begins executing the query and navigates through the database structure to find the requested information.
- **Result Extraction & Type Conversion:** SQLite retrieves the requested value —► C layer converts data from SQLite format to DPI-C-compatible SV type (e.g., string, numeric)
- **Return to SystemVerilog:** Result value crosses DPI-C boundary back to SV —► query ends when UVM object receives return value.

Query Performance: Indexed queries vs Unindexed scans

For million-row tables :

Indexed queries finish in microseconds

Unindexed full scans would take seconds



Use Cases & Applications

Use cases examples where an external database can improve simulation performance

Register Management
Store register definitions in SQLite enable lazy loading

Configuration Storage
Replace string-based config_db with indexed queries

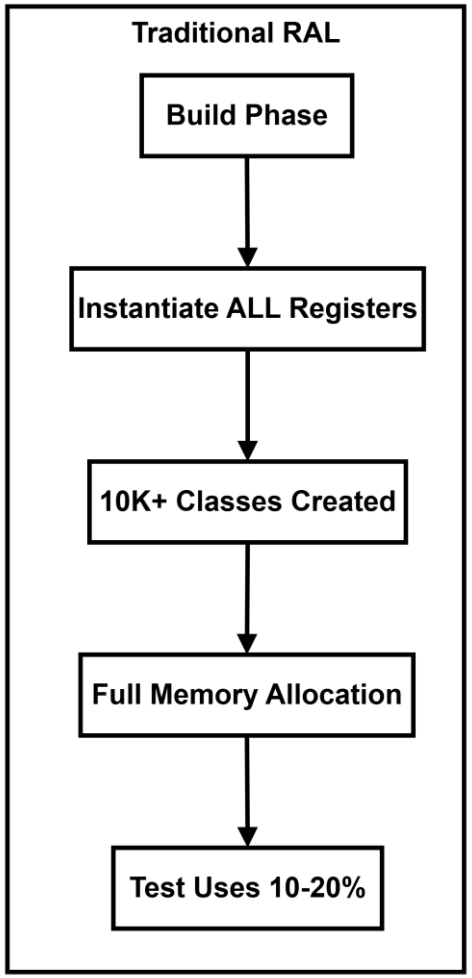
Coverage
Unified schema across platforms cross-run regression analysis trend reporting

Transaction Logging
Structured database logging SQL queries for analysis

Regression & CI/CD
Predictive analysis, test selection based on historical data

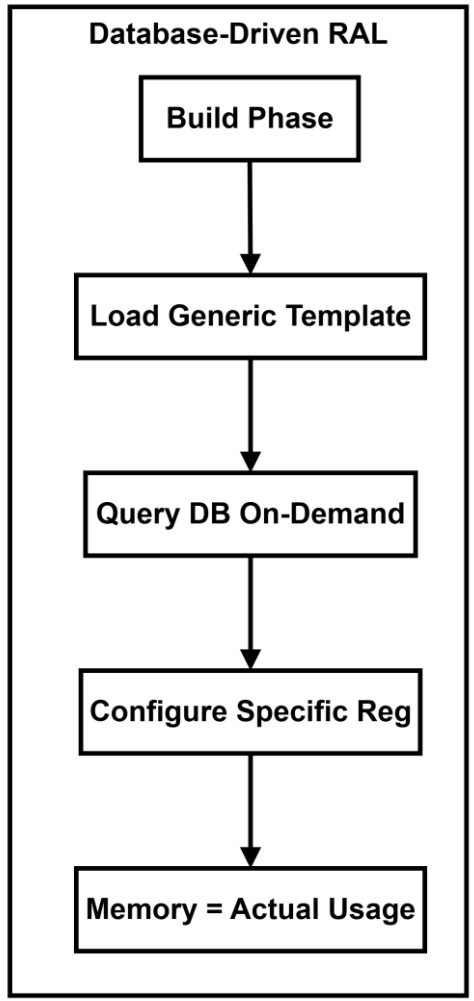
Highlights: Commercial solutions can be available for some use cases

Use Case: UVM RAL



Traditional UVM RAL
Creates class objects for each register, field, and block.
All registers are instantiated at the UVM build phase.
Load the entire hierarchy, although typical tests use just around 10% of the registers.

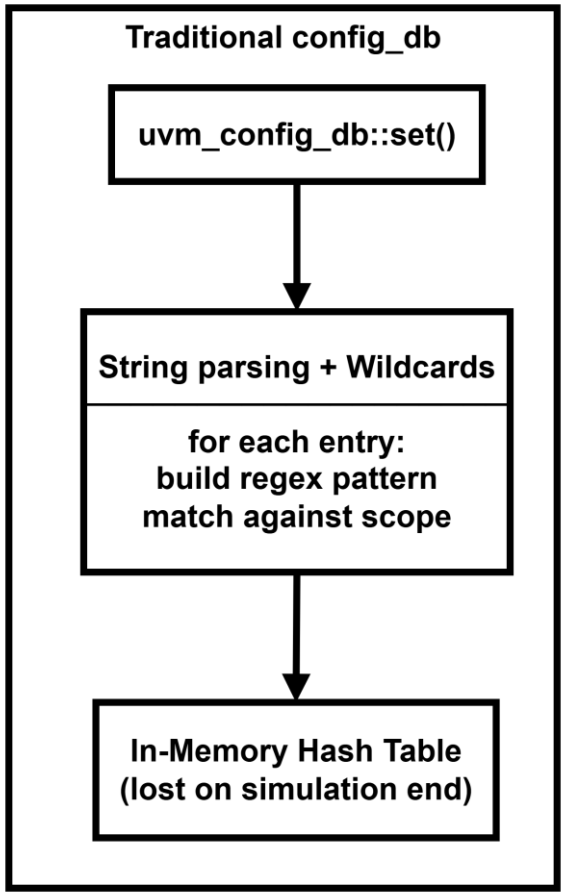
Database-Driven Dynamic RAL
Lazy loading: load only what the test requires.
A dynamic register object retrieves specific register data from the database.
Memory scales with actual usage.



Simulation results: PCIE subsystem (70K registers)

94%	39%	36%	13%
Field Memory Reduction	Compile Time Reduction	Runtime Reduction	Peak Memory Reduction

Use Case: UVM config_db

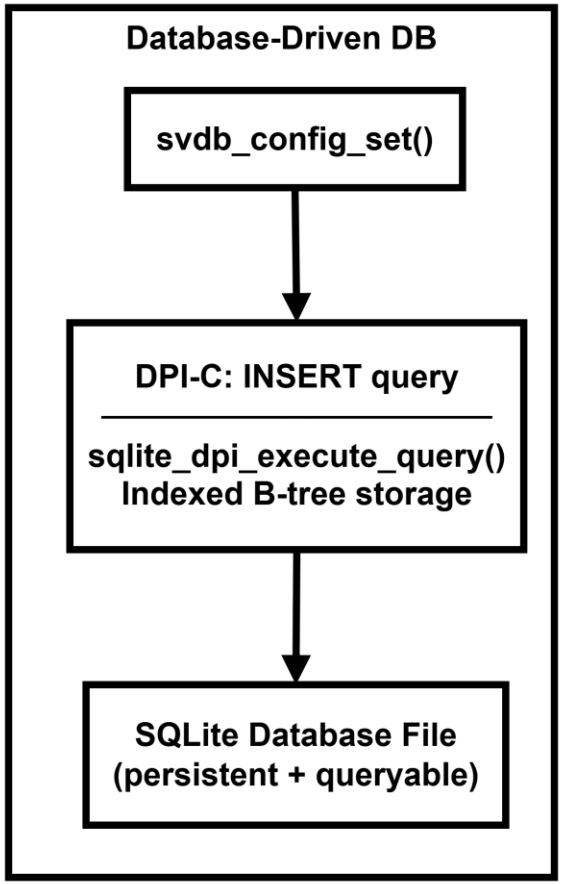


UVM config_db
Uses string-based lookups with wildcard matching. The TB with 20 to 30k calls in the build phase often experiences delays of 20 to 30 minutes.
All config_db data is lost after the simulation ends.

Database-Driven config_db
Indexed database queries.
Persistent data storage.
One-time database generation cost 10 minutes.

Estimated:

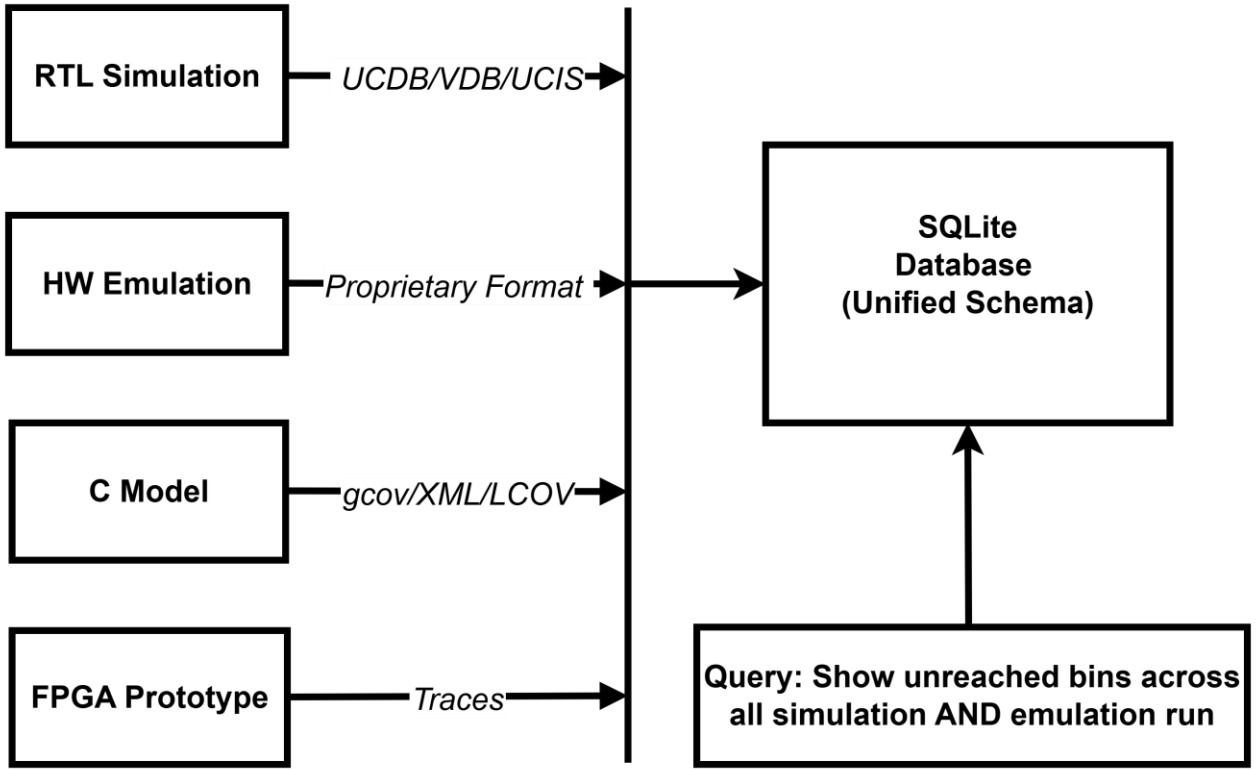
~98%	~150×	~50%	~80%
Build Phase Run Time Reduction	Operation Speed	Memory Reduction	Storage Compression



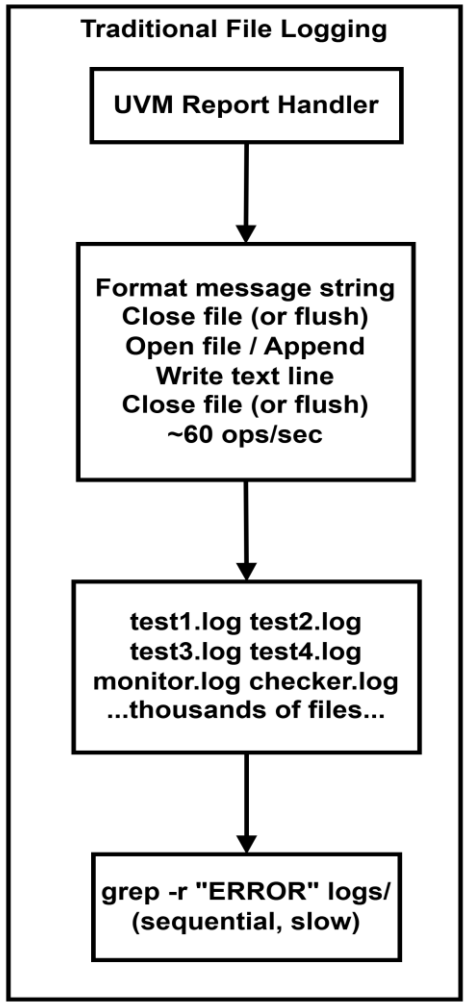
Use Case: Cross-Platform Coverage

The Problem
Modern verification runs on multiple platforms. There is no unified view across simulation, emulation, and prototyping.
Coverage data scattered across incompatible vendor formats
Cannot answer: "What bins remain uncovered across ALL platforms?“ "What coverage gaps exist across ALL platforms?"

SVDB Solution
Unified SQLite schema normalizes all coverage formats —▶ Import from RTL sim, HW emulation, C-model, FPGA traces
Running platform agnostic queries to merge coverage data
Single SQL query finds gaps across all verification platforms
With AI help, it enables informed decisions on where to focus verification efforts.



Use Case: Transaction Logging

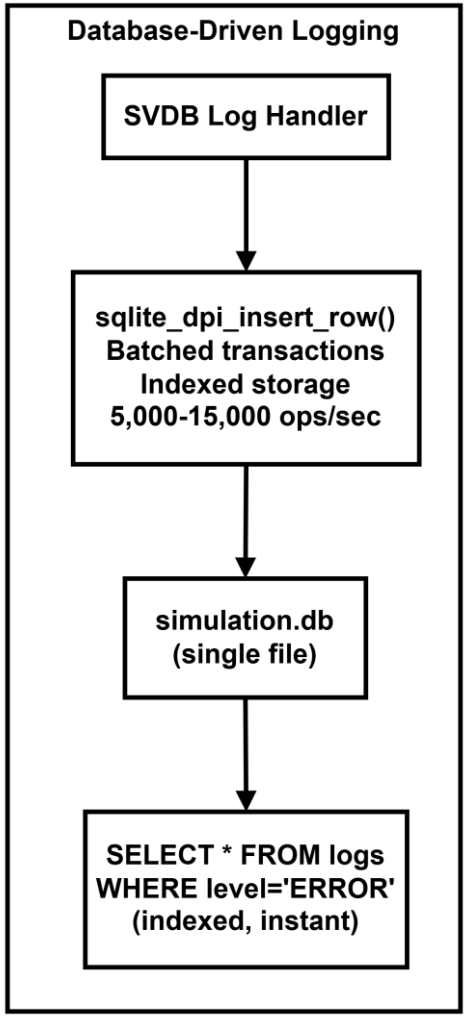


Traditional File Logging
Regression generates thousands of individual log files
Log file analysis required grep/regex sequential parsing.
High verbosity is disabled during regression runs. Re-run sims locally for debug

Database-Driven Logging
Single database
Indexed SQL queries
Always-on DB: query without re-running.
Instant trend analysis across hundreds of regressions.

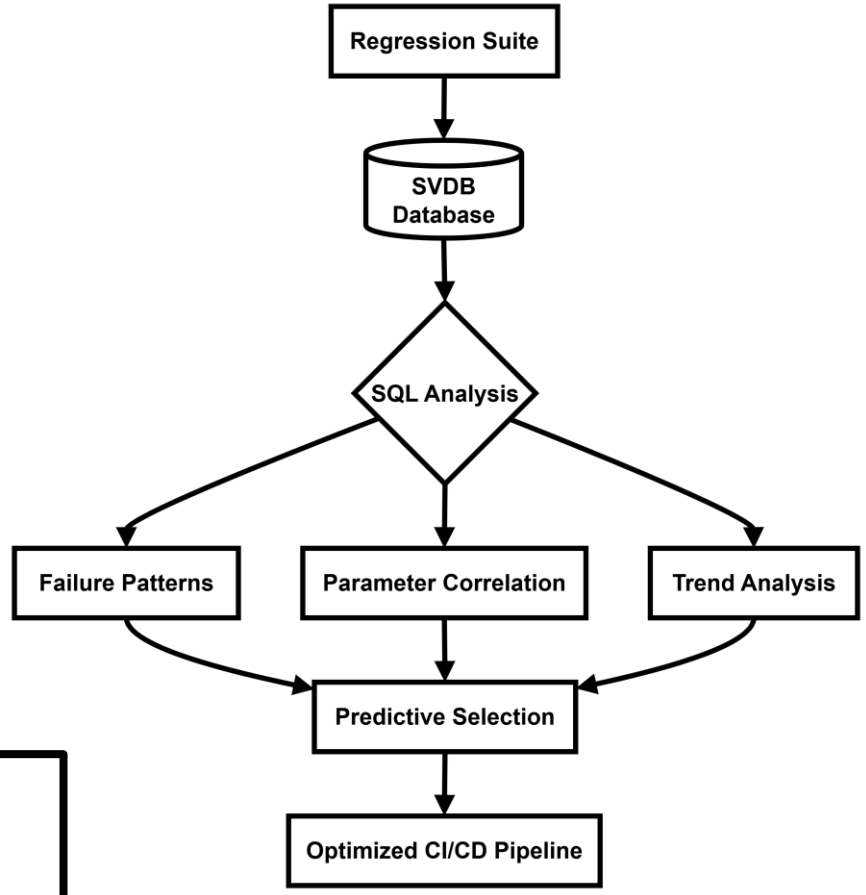
Estimated:

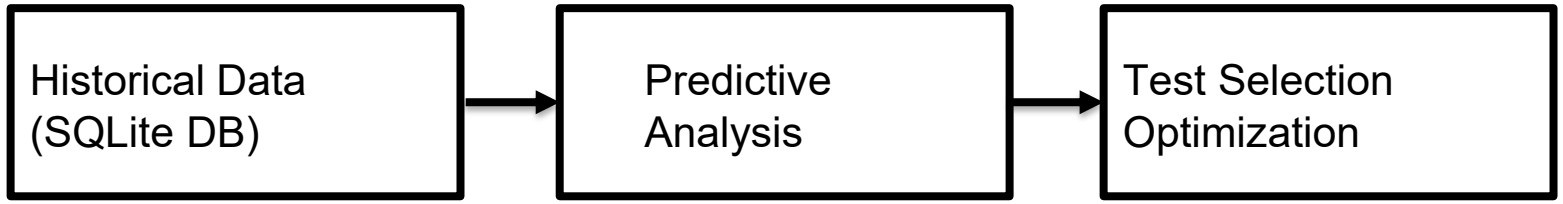
80-250×	~25%
Throughput Improvement	Storage Reduction



Use Case: Regression & CI/CD

Current Challenges	Databased-Driven
Thousands of regression tests generate scattered artifacts: pass/fail test logs, coverage data, waveforms, and more.	Queryable database for all regression data
No direct links between block-level and top-level results.	Easy to find parameter values correlating with failures
Failure analysis requires manual correlation	Track defects from block —▶ subsystem —▶ system
	Enables automation for test selection optimization



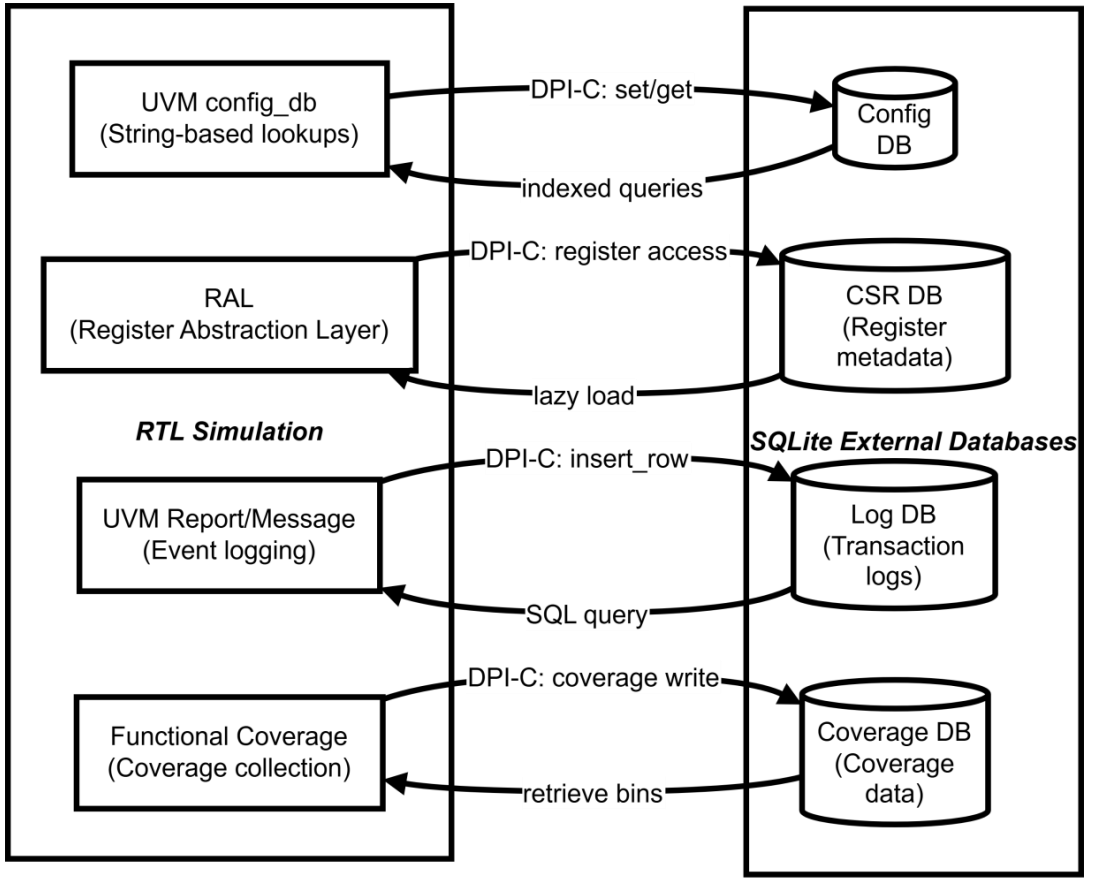


Highlights: All three major EDA companies offer proprietary regression management systems

Conclusion

TB with Database offload

- **RTL Simulation:**
  - UVM config_db: Stores dynamic configuration parameters
  - RAL: Manages register definitions and access patterns
  - UVM Report/Message: Captures events, errors, and debug information
  - Functional Coverage: Tracks coverage metrics and bins
- **External Databases:**
  - Config DB: Persistent indexed storage for configuration (replaces in-memory lookups)
  - CSR DB: Register metadata and lazy-loaded definitions
  - Log DB: Structured transaction and event logging
  - Coverage DB: Normalized coverage data for cross-run analysis

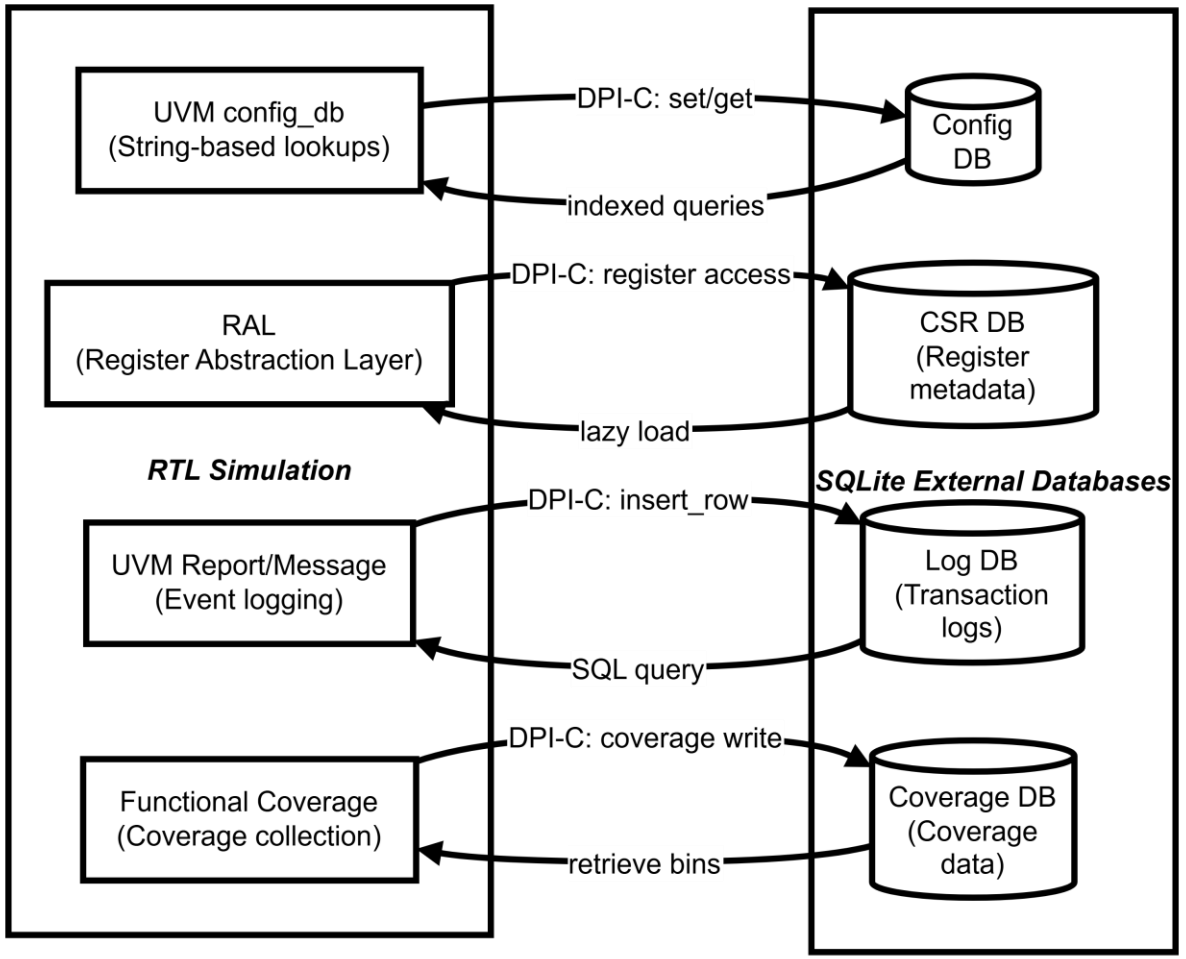


Highlights: Database offload does not replace UVM. It's enhancing **it.**

TB with Database offload "elephant in the room"

Historical Context:

- Database offloading is not a new approach.
- RTL simulators, wave viewers like Verdi, and regression management leverage database architectures for data management
- Unlike SVDB, these implementations are proprietary and closed to users. You cannot access the data, write custom queries, or integrate it with external tools.
- No interoperability between vendors
- Only initiative among big three vendors: Unified Coverage Interoperability Standard (UCIS)
- With AI adoption, open database approaches can be wake up call for EDA vendors. Integrating verification data with LLM and Retrieval-Augmented Generation(RAG) -based assistants is becoming standard practice.

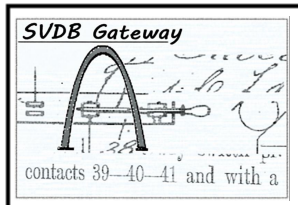


SVDB/SQLite Limitations & Mitigations, & Recommendations

Limitations	Mitigations
SQLite: unlimited readers but a single writer at a time. In multi-simulator setups, writes are serialized.	Transaction batching (50-100K ops/sec) If a higher write throughput is needed, you may consider switching to a client-server database that supports concurrent writes.
SQLite performance is optimal for database sizes below ~10 GB.	Client-server databases: PostgreSQL/MySQL for larger workloads
DPI-C overhead per operation (~0.5-1 msec of per operation)	SVDB Index optimization (100× query speedup)

Use database offload where it helps, and keep standard UVM flows elsewhere.
Start small. Prototype on one use case - maybe RAL for a large register map. Measure the impact. If it works, expand.

Resources



GitHub Repository

github.com/xver/svdb_gateway

MIT License • Examples included

Contact

vbesyakov@btadesignservices.com

BTA Design Service

Thank You!

Questions?