Applying Generative AI in Post-Silicon Validation: Real Use Cases and Technical Insights

Verification Futures Conference 2025 – Austin (USA)

Santosh Appachu Devanira Poovaiah
Senior CPU Verification Engineer @ NVIDIA Corporation.

AGENDA

1. **Why post-silicon still catches the hardest bugs ?**
2. **Where traditional flows stall (CRIG, directed, workloads) ?**
3. **GenAI vs. ML - Who does What ?**
4. **Architecture & Guardrails**
5. **Stimulus pipeline: Prompt → Plan → ASM → Validators**
6. **LLM Choice & RISC-V Grounding: From Model to Meaningful Tests**
7. **Results vs traditional (Coverage, Triage, Yield)**
8. **Risks, Governance, Next Steps**

Acronyms You'll See in This Talk

Abbrev	Full form	Where used	What it is
LLM	Large Language Model	AI/ML (generation, summaries)	Generates text/code from prompts
RAG	Retrieval-Augmented Generation	Grounding models with specs	Pulls docs to guide/model outputs
KPI	Key Performance Indicator	Metrics/decision criteria	Numbers that define success
LoRA	Low-Rank Adaptation	Light fine-tuning (optional)	Small adapter to fine-tune
HDBSCAN	Hierarchical Density-Based Clustering	Failure clustering (debug)	Unsupervised clustering algorithm
RV64GC	RISC-V 64-bit ISA	RISC-V/SoC	64-bit RISC-V with common extensions
Sv39	39-bit virtual memory	RISC-V/SoC	RISC-V paging scheme
TRM	Technical Reference Manual	Board/SoC documentation	Vendor manual for SoC/board details
CLINT	Core-Local Interruptor	RISC-V interrupts (local)	Local timer/software interrupts
PLIC	Platform-Level Interrupt Controller	RISC-V interrupts (global)	External/global interrupt controller
CRIG	Constrained-Random Instruction Generator	Stimulus (traditional)	Auto-generates constrained tests
ASM	Assembly (assembly language)	Test implementation	Low-level instructions for the CPU
LLC	Last-Level Cache	Coherency/perf counters	Shared cache nearest to memory
SDC	Silent Data Corruption	Failure impact/result	Wrong data without a visible crash
CAS	Compare-And-Swap	Concurrency/atomics	Atomic compare + conditional write
WFI	Wait-For-Interrupt	Power/idle & timing tests	Sleep until an interrupt arrives

Post-Silicon Reality: Bugs That Escape Pre-Silicon

1. Concurrency explosion**:** Cores + DMA/GPU/I/O + IRQs with rare interleaving's & more actors ⇒ combinatorial schedules and races.
2. Memory-model edges**:** Fences/atomics under preemption/ISRs behave differently ⇒ preemption can change fence/atomic effects.
3. Coherency under contention**:** Invalidate/downgrade/writeback races across slices ⇒ state transitions collide under heavy load.
4. SW/HW gaps**:** Drivers, TLB shootdowns, PCIe/CXL ordering quirks not modeled ⇒ OS/driver realities aren't in clean models.
5. Observability limits**:** Shallow traces, sampling bias, non-determinism ("Heisenbugs") ⇒ limited visibility hides root causes.
6. Pre-silicon plateau**:** CRIG/emulation/formal miss long-tail corners ⇒ tools stall on ultra-low-probability cases.
7. Impact**:** SDC, hangs/livelocks/deadlocks, performance cliffs ⇒ failures that matter only surface on silicon.

Micro-example**:** 4-core CAS + contending stores to a shared line; LLC back-pressure + IRQ ⇒ downgrade/writeback reorder ⇒ **stale read ~1/10k runs** , tiny timing shifts flip the outcome.

Limits of Traditional Validation (CRIG, Directed, Workloads)

1. Directed tests**:** precise & explainable, but narrow ⇒ don’t scale to long-tail races.
2. CRIG (constrained-random): broad exploration ⇒ constraints bias paths; rare interleaving's stall.
3. Seeds & repeatability**:** reruns flip schedules ⇒ non-determinism hides/ghosts failures.
4. Workload stress (apps/benchmarks): realistic SW/HW ⇒ low controllability; hard to isolate causes.
5. Coverage plateau**:** more cycles ≠ new states ⇒ diminishing returns after early wins.
6. Debug burden**:** huge logs/counters ⇒ manual triage & reduction consume days.
7. Modeling gaps**:** ideal latencies, missing OS/driver effects ⇒ pre-silicon differs from board reality.
8. Test bloat**:** many near-duplicates ⇒ infra/compute cost rises while novelty drops.

Micro-example: CRIG with cache pressure hits misses but **rare invalidate→downgrade→writeback race** remains elusive - needs targeted, multi-core tests.

GenAI's Role in the Flow (Augment, Don't Replace)

1. What is GenAI? Generative models (LLMs/code models) that create text/code/plans from prompts ⇒ here: test plans & summaries.
2. How it differs from ML: ML predicts/scores on structured data; **GenAI** produces candidate artifacts ⇒ they complement each other.
3. Stimulus ideation: prompt → test plan → ASM ⇒ LLM proposes diverse, targeted sequences from high-level goals.
4. Test mutation: tweak seeds/strides/fences ⇒ systematically explores variants humans/randoms might miss.
5. Log/trace summarization: condense 100Ks lines ⇒ surfaces epochs, anomalies, candidate causes fast.
6. Hypothesis drafting: "what likely happened & why" ⇒ starting point for engineers, not a verdict.
7. Validator sandwich: ISA/protocol lint → assemble → pre-silicon dry-run → post-silicon checkers ⇒ guards correctness & realism.
8. Human-in-the-loop: review/edit, gate promotions ⇒ sign-off remains human + formal/checkers.
9. Data handoff to ML: counters/telemetry → detect & cluster ⇒ LLM generates/annotates; ML ranks at scale.
10. Artifacts & repro: store prompt/plan/code/logs ⇒ audit trail for learning & reuse.

Important: **GenAI augments** CRIG/directed/formal; it **does not replace** sign-off methods.

LLM vs. Classical ML : When to Use Which

1. Use LLMs (generative) when the artifact is text/code
    a) **Test synthesis from goals** ⇒ prompt → plan (JSON) → ASM.
    b) **Log/trace summarization** ⇒ condense 100k+ lines to epochs & suspects.
    c) **Multi-core choreography** ⇒ structured steps, fences, roles.
    d) **Spec grounding (RAG)** ⇒ cite ISA/coherency rules to cut hallucinations.

2. Use Classical ML when data is numeric/structured
    a) **Anomaly detection on counters/telemetry** ⇒ fast, scalable (IF/LOF/1-class SVM).
    b) **Clustering failure signatures** ⇒ group by metrics/timelines (HDBSCAN/DBSCAN).
    c) **Ranking & triage** ⇒ prioritize by novelty/impact (XGBoost/GBMs).
    d) **Drift/trend monitoring** ⇒ catch regressions over time (PCA/stats).

3. Hybrid patterns (best practice)
    a) **LLM plan + ML scorer** ⇒ generate variants, rank by novelty/risk.
    b) **LLM summaries + ML clustering** ⇒ route issues to the right owners.
    c) **Validator sandwich** ⇒ ISA/protocol lint → assemble → pre-silicon gate → checkers.

4. Quick decision rules
    a) **Text/code?** ⇒ LLM (+ strict schema & validators).
    b) **Metrics/time-series?** ⇒ ML.
    c) **Tight latency/edge?** ⇒ ML or **small** on-prem LLM (7–13B, quantized).
    d) **Privacy/IP sensitive?** ⇒ self-host LLM + classical ML; avoid external APIs.

5. Evaluate with the right KPIs
    a) **LLM:** compile/runnable rate, coverage lift, bug repro rate, summary accuracy.
    b) **ML:** precision/recall on anomalies, cluster separability, triage time reduction.

Model Choice Playbook: On-Prem vs Managed, Context, Code

Playbook = short decision guide (how to choose models consistently).
On-prem = you host the model (data stays inside)
Managed = provider hosts (cloud API)

1. Start with constraints
    a) **Privacy/IP:** can data leave? → **No:** on-prem. **Yes:** managed.
    b) **Latency/throughput:** sub-second vs batch → influences model size & hosting.
2. Match model to the job
    a) **Code/test generation:** use **code-tuned LLMs** (better compile/runnable).
    b) **Long log summaries:** use **long-context LLMs** (or chunk + retrieve).
3. Handle context correctly
    a) **Chunk + RAG** over ISA/coherency/board docs; don't cram giant prompts.
4. Size by KPI, not hype
    a) Start **small (7–13B) on-prem** if private; scale to mid/large **only** if **KPIs** improve (coverage lift, runnable %, triage time, cost).
5. Guardrails (non-negotiable)
    a) **JSON/grammar constraints** + **validator sandwich** (lint → assemble → pre-silicon gate → checkers) + **human review**.
6. Quick picks
    a) **Strict IP + code gen: on-prem 7–13B code-tuned + RAG + validators.**
    b) **Huge logs fast (policy allows): managed long-context** or mid/large on-prem.

Architecture & Guardrails for Safe Adoption

1. Grounding data layer : ISA/coherency specs, board configs, templates via **RAG** to keep outputs factual.
2. LLM orchestrator : prompt templates, few-shot, **JSON/grammar-constrained decoding** so plans are structured/parsable.
3. Validator sandwich (gate) : ISA/protocol lint → assemble/compile → fast sim/litmus → coherency/memory-model checkers block illegal/unrealistic tests.
4. Execution harness : JTAG/UART loader, timeouts/watchdogs, perf counters & traces safe runs + rich telemetry.
5. Triage & insights : **LLM summaries** for logs + **ML** anomaly/clustering on counters faster, scalable debug.
6. Feedback loop : coverage deltas, failure labels, prompt reweights, human approval learn what works; keep humans in charge.
7. Governance & security : on-prem by default, IP/PII redaction, RBAC, audit logs, **prompt/model registry & versioning** compliance and reproducibility.
8. KPI gates : promote only if **runnable%↑, coverage↑, triage time↓, cost/latency OK** data-driven adoption.
9. Safeguards & rollout : shadow/canary, blue-green, quotas, offline mode/air-gap safe experimentation & instant rollback.

One-line flow: **RAG → LLM plan (JSON) → Validators → DUT run → Logs/Counters → LLM+ML triage → Feedback/Promotion**

Stimulus Generation Pipeline: Prompt → Plan → ASM → Validators

1. Prompt (grounded) : goal + constraints; **RAG** injects ISA/coherency snippets Keeps generations factual and in-scope.
2. Plan (JSON, not prose) : LLM emits ops/params under **JSON/grammar constraints** Machine-checkable; easy to diff & reuse.
3. Static validators : schema, opcode legality, privilege, alignment, hazards Fail fast before any code runs.
4. ASM synthesis : fill templates, allocate regs, insert fences/loops Deterministic assembly from the plan.
5. Build & pre-silicon gate : assemble/link; fast sim/litmus; protocol/memory-model checks Blocks unrealistic/illegal tests.
6. Post-silicon run : loader, timeouts, counters/traces; persist seed/params Safe execution + rich telemetry.
7. Promotion & archive : require **new coverage/runnable%/repro value**; store prompt/plan/ASM/logs Reproducible and auditable.

Example : 4-Core Downgrade/Writeback Race

```
# Prompt (snippet)
'''
Generate a 4-core workload that induces downgrade/writeback races on a shared cache line.
Use CAS on Core0, contending stores on Core1, eviction pressure on Core2, and periodic preemption/interrupts on Core3.
Vary stride ∈ {64,128}. Respect privilege=U, fences allowed.
'''

# RAG grounding injected automatically
'''
ISA: LR/SC pairing & fence semantics (acquire/release).
Memory model: ordering guarantees for atomics + interrupts.
Platform: L2 present; line size=64B; legal address window=0x8000_0000
'''

# Plan (JSON emitted by LLM under schema constraints)
'''
{
  "meta": {"isa":"RV64", "priv":"U", "line_size":64},
  "targets": [{"addr":"0x80000000","len":64}],
  "ops": [
    {"core":0,"op":"CAS_LOOP","addr":"0x80000000","barrier":"FENCE_RW_RW","iters":100000},
    {"core":1,"op":"STORE_LOOP","addr":"0x80000000","pattern":"STRIDE","stride":[64,128],"iters":100000},
    {"core":2,"op":"LLC_THRASH","region":"0x80001000..0x80010000","pressure":"high","stride":[64,128]},
    {"core":3,"op":"YIELD_IRQ","period_iters":500,"action":"IPI_CORE0"}
  ],
  "timeouts_ms": 20000,
  "safety": {"no_priv_escalation":true,"aligned_access":true}
}
'''

# Why this is good: JSON is parsable; fields match your schema; ops are named from a whitelisted opcode catalog your toolchain knows.
```

```
# ASM synthesis (skeletons)

# Core 0 (CAS loop)
'''
loop_cas:
    lr.d.aq   t0, (a0)          # load-reserved old value
    bne       t0, a1, cas_fail  # if *ptr != expected, fail immediately
    sc.d.rl   t1, a2, (a0)      # try to store desired
    bnez      t1, loop_cas      # if store failed (lost reservation), retry
    li        a0, 1             # success
    j         cas_done
cas_fail:
    li        a0, 0             # failure (no store attempted)
cas_done:
    ret
'''

# Core 1 (contending stores)
'''
loop1:
  fence rw, rw
  sd     t3, (a1)
  addi   a1, a1, STRIDE
  j      loop1
'''

# Core 2 (LLC thrash)
'''
loop2:
  ld     t4, 0(a2)
  sd     t4, 64(a2)
  addi   a2, a2, STRIDE
  blt    a2, a2_end, loop2
  j      loop2
'''

# Core 3 (periodic IPI/YIELD) — handled by harness (e.g., trigger IPI to Core0 every N iters).
```

- Static validators (examples) :
  a) **Schema** : (all required fields present)
  b) **ISA legality**
  c) **Alignment** : (64B stride on 64B line)
  d) **Hazards:** (forward progress: CAS loop has retry exit; watchdog present).

- Expected behavior (why this test exists)
  a) Contention + IPI can **reorder downgrade/writeback timing**, occasionally exposing a **stale read** or unexpected latency spike - exactly the corner you want to probe on silicon.

Experimental Setup & Hardware (RISC-V)

1. Board
   VisionFive 2 (StarFive JH7110) — quad-core **RV64 (SiFive U74-class)**, shared **L2 ≈ 2 MB**, **64-byte** cache lines, **4–8 GB LPDDR4**, Linux-capable.
2. Memory & ISA
   RV64GC, **Sv39** paging, cacheable, aligned to line size.
3. Interrupts & I/O
   CLINT (MSIP/MTIMECMP) + **PLIC** for periodic interrupts; UART/JTAG for control.
4. Harness
   UART/JTAG **loader**, **watchdogs/timeouts**, capture **counters** (LLC misses/invalidates, L2 writebacks) and **logs/traces**; auto-archiving.
5. Stimulus generation (LLM)
   On-prem code-tuned (≈7–13B) model; **RAG** over RISC-V unprivileged/privileged specs & memory model; **JSON-schema outputs** with **opcode whitelist**.
6. Validators (pre-silicon gate)
   Schema/ISA legality, alignment/privilege, fences; coherency/memory-model checks.
7. Run plan
   Param sweeps (**STRIDE = {64,128}**, **~1e5** iters), **12 seeds**, **~50k runs** total; KPIs tracked (**runnable%**, **coverage lift**, **time-to-hypothesis**).

LLM Choice & RISC-V Grounding: From Model to Meaningful Tests

1. Which LLM ?
   On-prem, code-tuned ~7–13B model (self-hosted, quantized) — optimized for code/test generation and privacy.

2. Why this model (selection basis) ?
   a) Chosen by **KPIs**, not brand: **compile/runnable %**, **coverage lift**, **time-to-hypothesis**, **latency/cost**.
   b) Small code model met KPIs; bigger dosent move metrics for this task.

3. How to teach RISC-V (no full retrain) ?
   a) **RAG grounding** with RISC-V ISA/memory-model excerpts + platform facts (line size, regions).
   b) **Constrained decoding** to **JSON schema** + **whitelisted opcode catalog**.
   c) Optional **small LoRA** on internal examples (plans → ASM style), not required.

4. Where RISC-V architecture came from ?
   Open ISA: public RISC-V specs (unprivileged/privileged, memory model) + litmus tests; vendor TRMs for board specifics.

5. Behavior
   a) Ungrounded: occasional illegal opcodes/privilege slips.
   b) With grounding/constraints: **compile ~98%**, **runnable ~95%**, higher test diversity.

6. Do tests match expectations?
   a) Yes—generated parameterized multi-core sequences that passed validators and exercised targeted coherency corners.

7. Benefit vs traditional flow
   a) **Faster test ideation** with higher **novelty per hour**; integrates with CRIG; engineers spend time on triage, not hand-writing tests.
   b) **Sign-off unchanged**: protocol checkers + human review remain the gate.

RAG-First Test Generation (No Retraining): Grounded LLM for RISC-V

1. Model (how to use it)
   a) **Self-hosted, code-tuned small LLM (~7–13B)**; quantized; **JSON/grammar-constrained decoding**.
   b) KPI focus : **runnable%**, **coverage lift**, **latency**.

2. Grounding (RAG) doc set
   a) RISC-V **unprivileged/privileged ISA**, **memory model**, litmus notes.
   b) Board/TRM facts (line size, cacheability/shareability), harness API.

3. RAG flow (no retrain)
   a) **Chunk & index** docs (≈512–1024 tokens).
   b) At request time, **retrieve top-k (3–5)** relevant snippets.
   c) Inject snippets into prompt as **GROUNDING** with short citations.

4. Prompt template (core pieces)
   a) **SYSTEM:** You generate RISC-V coherency stress **plans** (JSON only).
   b) **GROUNDING:** (retrieved ISA/memory-model/TRM excerpts).
   c) **TASK:** 4 cores; induce downgrade/writeback races; U-mode only; fences allowed.
   d) **SCHEMA:** fields + **opcode whitelist**, legal address window, allowed **STRIDE={64,128}**.
   e) **CONSTRAINTS:** no privileged ops; timeouts/watchdogs required.

5. After the LLM
   a) **Static validators** → assemble/link → **mini-sim/litmus gate** → board run.
   b) **Promote** only if **runnable%↑**, **new coverage**, **useful repro**; archive prompt/plan/logs.

6. Why RAG first
   Adapts with spec changes; **no heavy training**; **fewer hallucinations**; **auditable**.

```
# Plan output
{
  "meta": {"isa": "RV64", "priv": "U"},
  "targets": [{"addr": "0x9A00_0000", "len": 64}],
  "ops": [
    {"core": 0, "op": "CAS_LOOP",   "addr": "0x9A00_0000", "barrier": "FENCE_RW_RW", "iters": 100000},
    {"core": 1, "op": "STORE_LOOP", "addr": "0x9A00_0000", "stride": [64,128]}
  ],
  "timeouts_ms": 20000
}
```

RAG-First Debug Acceleration (No Retraining): Long Trace → Evidence-Backed Summary

1. Model (how to use it)
   a) **Self-hosted long-context LLM** (or **chunk + RAG** if context is tight); **JSON/grammar-constrained outputs** with **evidence line refs**.
   b) KPI focus: **triage time ↓**, **summary accuracy ↑**, **ops latency <60s/run**.

2. Grounding (RAG) doc set
   a) RISC-V **memory model**, coherency rules, interrupt/privilege notes.
   b) Board/TRM timing/IRQ paths, harness semantics, logging dictionary.

3. Flow (logs → summary + clusters)
   a) **Parse & segment** logs into **epochs** (boot → workload → fail).
   b) **Retrieve** top-k spec/TRM snippets per epoch (RAG).
   c) **LLM summarize** each epoch to JSON with **line citations** + **confidence**.
   d) **ML sidecar**:
      - **solation Forest** on counters → anomaly scores.
      - **HDBSCAN** on features → **failure clusters** + labels.
   e) **Human-in-loop** reviews one-pager (summary + cluster + cited lines) → assigns owner.

4. Prompt template (core pieces)
   a) **SYSTEM:** Summarize RISC-V failure logs **with citations**. Output **JSON only**.
   b) **GROUNDING:** (retrieved memory-model/TRM/harness snippets).
   c) **INPUT:** epoch log chunk (+ counter slice).
   d) SCHEMA: fields: epochs[], anomalies[], hypotheses[], citations[].

5. Measured impact
   a) **Triage time (median)**: **6.5h → 2.4h**.
   b) **Stable clusters**: **5–7**; top-3 ≈ **82%** of fails.
   c) **Anomaly detection**: **precision ≈ 0.86**, **recall ≈ 0.78** (spot-checks).
   d) **Per-run latency**: **<60s** (chunked pipeline).

6. Why RAG first
   Keeps summaries aligned to **actual spec/board behavior**; **transparent & auditable**; avoids heavy fine-tuning.

```
# Expected LLM output

{
  "run_id": "rv64-poc-001237",
  "epochs": [
    {
      "name": "workload",
      "start": 8200,
      "end": 17850,
      "summary": "L2 writebacks spike; invalidates precede CAS retry storms on hart0; IRQs on hart3 overlap retries.",
      "citations": [17102, 17115, 17638]
    },
    {
      "name": "failure",
      "start": 17851,
      "end": 18010,
      "summary": "Progress stalls on hart0 after repeated LR/SC failures.",
      "citations": [17872, 17904]
    }
  ],
  "anomalies": [
    {"metric": "llc_invalidates", "z": 3.2, "window": [17050, 17600]},
    {"metric": "l2_writebacks",  "z": 2.7, "window": [17080, 17520]}
  ],
  "hypotheses": [
    {
      "text": "Downgrade/writeback race under IRQ preemption impacts LR/SC progress on hart0.",
      "confidence": 0.74,
      "evidence_lines": [17115, 17638, 17872],
      "related_specs": ["RV memory model (acq/rel)", "Interrupt/privilege interaction notes"]
    }
  ],
  "routing": {"cluster_id": "C-3", "owner_suggested": "Cache/Coherency"}
}
```

Results & Head-to-Head vs Traditional

1. KPI summary
   a) **Coverage (unique interleavings, CRIG=100): LLM+Val 136 (+36%), Hybrid 145 (+45%)**
   b) **Rare-race hits (/1k runs):** Directed **1.2** · CRIG **3.4** · **LLM+Val 6.1** · **Hybrid 7.0**
   c) **Runnable yield (generated tests): Raw LLM 72% → 95% with validators**
   d) **Debug triage (median to first hypothesis): 6.5h → 2.4h** (LLM summaries + ML clustering)
   e) **Per-run analysis latency (logs): <60s** (chunked + RAG, on-prem)

Criterion	Directed	CRIG	LLM + Validators	Hybrid (CRIG + LLM)
Rare interleavings	Low	Medium	High	Highest
Coverage (CRIG=100)	~70	100	136	145
Runnable yield (gen'd tests)	N/A	N/A	95%	95%
Debug triage time	6.5h (manual)	6.5h (manual)	2.4h	2.4h

2. Why this matters
   a) **More novel stress** with less hand-authoring.
   b) **Fewer flakies** (validator sandwich) → better bench utilization.
   c) **Faster, evidence-backed triage** → engineers validate fixes sooner.

Lab PoC; metrics are measured/rounded. GenAI/ML **augment** existing flows; **sign-off** remains with protocol checkers + human review.

Key Takeaways & Next Steps

1. **No retraining used:** RAG + JSON constraints + validator sandwich on a **self-hosted, code-tuned small LLM (~7–13B)**.
2. **It worked:** more coverage, 95% runnable, triage hours → minutes.
3. **Retraining optional:** add **light LoRA** only if KPIs stall; facts stay in RAG.

Use these without retraining (on-prem):

1. **Test generation (single-GPU):** Code Llama 13B Instruct, Qwen-Coder 7B, DeepSeek-Coder ~7B, StarCoder2 ~15B.
2. **Long-log summaries:** long-context self-hosted (e.g., Llama-3/70B Instruct) **or** do **chunk + RAG**.

Q&A