Contamination-Resistant Code Evaluation on Real Codebases

A single-model pilot with Qwen3.6-35B-A3B on the Cerberus validation framework, run on a single 32 GB GPU

● Pilot – operational results measured; capability & contamination-resistance claims stated as falsifiable criteria

Abstract

Static coding benchmarks degrade as their tasks leak into training corpora. This document describes a small, reproducible pipeline that sidesteps that failure mode by synthesising evaluation tasks directly from a real, versioned repository via abstract-syntax-tree (AST) analysis, then scoring a model's output under strict single-stream local inference. As a first data point, the open-weight Qwen3.6-35B-A3B (sparse Mixture-of-Experts, ~3B active parameters) completed all 60 generated tasks on the Cerberus codebase in 4 min 57 s at roughly 200 tokens/s on a single RTX 5090. The headline result is deliberately not framed as a capability score: a 100% pass rate means the task set was not discriminative, which is itself the most useful finding of the pilot and sets the agenda for the next iteration. The contribution is the method and its VRAM discipline, not a leaderboard number – and, made explicit here, an honest account of the two properties the pilot does not yet demonstrate (contamination resistance and execution-verified correctness) together with an evaluation plan that turns each into a measurable criterion.

Keywords: data contamination · dynamic benchmarks · code evaluation · AST synthesis · sparse MoE · execution grounding · sovereign inference

§01

Motivation

Benchmark validity in LLM evaluation is tied to how far the test data has escaped the training set. Synthetic and public suites are increasingly compromised by data contamination – the test cases sit, un-decontaminated, inside the models' training corpora, which inflates scores without reflecting real capability [1, 2].

The response taken here is to stop using fixed public tasks and instead generate tasks on demand from real, versioned source code. A structural extraction engine reads a target repository and manufactures fresh tasks bound to that code's actual structure. Because the tasks are derived from a specific commit of a specific repository rather than a static list, contamination becomes a property that can be reasoned about and refreshed, rather than an invisible confound.

The target for this pilot is the Cerberus validation framework (pyeve/cerberus), a security-relevant Python repository specialising in deeply nested input validation with a high density of edge cases (~26 source files). Input validation is a domain where a hallucinated "fix" is maximally costly, which makes it a useful stress surface.

§02

Related work: from static to dynamic, contamination-resistant code evaluation

This pilot sits in an active line of work that has been moving code evaluation from static suites to dynamically generated, contamination-resistant benchmarks, and the design should be read against it rather than in isolation [1]. Early execution-based suites – HumanEval and MBPP – established functional-correctness testing but are single-file, synthetic, and now heavily leaked; EvalPlus hardened them with mutation-generated tests but did not solve contamination [9]. The field's response has taken three forms. Temporal / live harvesting: LiveCodeBench collects contest problems published after a cutoff date, and SWE-bench-Live (via its REPOLAUNCH pipeline) continuously harvests fresh real-world issues, each treating recency as the contamination control [2, 3]. Private or commercial sources: SWE-bench Pro incorporates held-out commercial repositories to defeat memorisation [8]. Automated generation from repositories: Code2Bench builds contamination-resistant benchmarks by extracting function candidates from recent GitHub commits, and R2E-Gym and SWE-Smith synthesise repository-level tasks and their verifying tests at scale [4, 5, 6]. This pilot's AST-driven synthesis is a member of that third family, distinguished by two deliberate constraints: it runs end-to-end on a single 32 GB GPU and air-gapped, and it targets a security-relevant validation library – the axis studied by security-code benchmarks such as SecurityEval and CodeLMSec, which the audit category here echoes [10].

Two lessons from that literature shape the honest reading of the pilot (§5) and the plan (§6). First, contamination resistance is only established by a fresh-versus-stale comparison – tasks drawn from code the model has not seen – not by the mere act of generating from a repository [1, 4]. Second, the field's standard for a code "pass" is execution-grounded: success means the repository's tests pass after the model's change; a "test oracle" of syntactic completeness is known to be unreliable [3, 7, 9]. The pilot is faithful to the generation half of this literature; §6 adopts its verification half.

§03

Methodology

3.1 Structural task synthesis

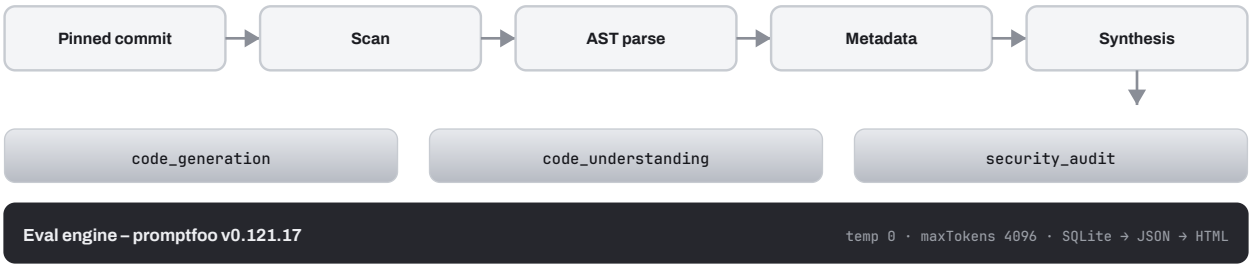
Tasks are produced by a multi-stage static analysis rather than authored by hand (Figure 1): a repository scan → file detection (glob + filter) → AST parse → metadata extraction → task synthesis. The scan deterministically enumerates source files and excludes build artefacts, `__pycache__`, and distribution directories. The AST of each remaining file is parsed to extract a bounded set of metadata: the top function/class signatures, the import dependencies from the file head, and the body of the first functionally significant method. That structured metadata feeds a synthesis step producing three task categories, up to one task per category per file:

Category	Task
code_generation	Generate a function from a requirement plus the AST-extracted import context and safety constraints.
code_understanding	Explain and locate a deliberately injected bug inside an isolated validation snippet.
security_audit	Identify vulnerabilities – injection vectors (SQL, command, path traversal), unfiltered eval/exec.

Extracted records are serialised from internal dictionaries to JSON for the evaluation engine.

Task synthesis – from a pinned commit to a scored run

Tasks are a pure function of (repo state, extraction rules) – re-derivable from the exact commit, not a public list.



3.2 Model under test

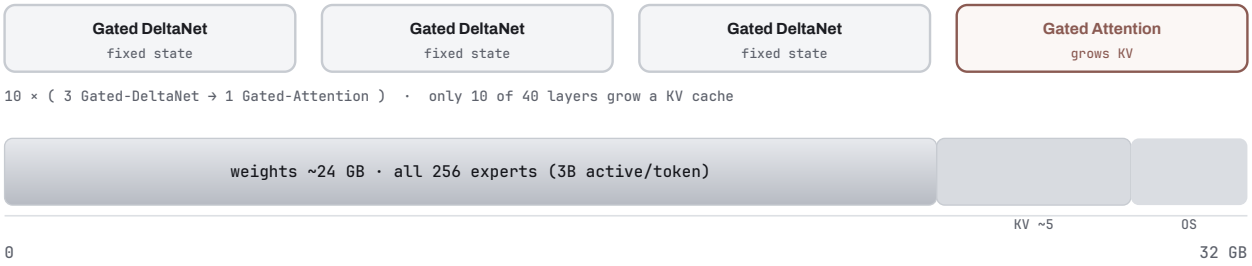
Property	Value
Model	Qwen3.6-35B-A3B (qwen3.6:35b)
Architecture	Sparse Mixture-of-Experts – 35B total, ~3B active per token (256 experts, 8 routed + 1 shared)
Quantisation	Q4_K_M
VRAM footprint	~24 GB (all expert weights resident)
Context	262 K native
Backend	Ollama (llama.cpp)
Sampling	temperature 0.0 (deterministic), maxTokens 4096

A precise note on MoE, because it is routinely misstated: the router activates only a fraction of the network per token, so compute and memory-bandwidth per token scale with ~3B, not 35B – this is what makes the throughput below achievable. But all experts must be resident, so the VRAM footprint reflects the full 35B (~24 GB at Q4_K_M). MoE buys compute and bandwidth, not memory.

Per the published architecture [11], the model runs 40 layers as ten repetitions of three Gated-DeltaNet blocks followed by one Gated-Attention block, each feeding a 256-expert MoE (8 routed + 1 shared active). Two consequences matter for a 32 GB budget (Figure 2). First, only the ten Gated-Attention layers grow a KV cache – the thirty DeltaNet layers carry a fixed-size recurrent state – so cache growth per token is roughly a quarter of what a 40-layer full-attention model of the same width would incur. Second, those attention layers use grouped-query attention with two KV heads at head-dim 256, so at q8_0 the KV cache costs on the order of ~10 KB per token, keeping even long contexts inside the ~5 GB left after the weights.

Why a 35B model fits 32 GB – sparse experts + hybrid attention

MoE buys compute and bandwidth, not memory; Gated-DeltaNet buys KV-cache headroom.



3.3 Hardware

Component	Specification
GPU	NVIDIA RTX 5090 – 32 GB GDDR7, 512-bit, ~1.79 TB/s, 680 5th-gen Tensor Cores
CPU	AMD Ryzen 7 9850X3D (Zen 5, 8C/16T, 96 MB L3)
Backend	Ollama (llama.cpp), Q4_K_M

Decoding is memory-bandwidth-bound: each generated token requires the active weights to be read once through the memory bus, so the RTX 5090's ~1.79 TB/s – against only ~3B active parameters – is what sustains the measured rate. The CPU's role is prefill and host-side orchestration; its 3D V-Cache does not materially accelerate decode, since the optimised path is on the GPU.

3.4 Single-stream determinism

Multi-model evaluation on consumer VRAM is bottlenecked not by capability but by memory. To keep latency measurements clean, the backend is pinned to a single context stream: `OLLAMA_NUM_PARALLEL=1` and `OLLAMA_MAX_LOADED_MODELS=1`. Parallel API requests would multiply the KV-cache linearly with connection count and risk a VRAM overflow into system RAM – a CPU fallback drops generation from >150 tokens/s to under a handful, turning latency data into noise. The pipeline trades theoretical concurrency for constant, reproducible throughput.

3.5 Instrumentation

Orchestration runs through promptfoo (v0.121.17) via CLI, with a separate isolated YAML configuration per model to avoid the framework's multi-provider routing race conditions on slow local boot. Output is written to SQLite, exported to JSON, and aggregated into an HTML dashboard. Tracked per task: pass/fail, latency, token usage (prompt / completion / cached), and finish reason.

§04

Results

Qwen3.6-35B-A3B on Cerberus (60 tasks: 20 generation + 20 understanding + 20 audit) (Figure 3):

```
Result:        60 / 60 completed
Duration:      4 min 57 s
Avg latency:   ~5.0 s / task
Throughput:    ~200 tokens/s
Total tokens:  74,342 (12,682 prompt / 59,392 completion)
Stability:     no latency spikes, no OOM under NUM_PARALLEL=1
```

Qualitatively, generated code was syntactically valid with consistent type hints (`Optional[str]`), preferred the standard library (`re`) over fragile third-party parsers, and handled None / empty-string / TypeError cases consistently.

Pilot result – operational signal, not a capability score

60 tasks, single stream, temperature 0; the 100% pass rate is read as a non-discriminative task set.

60 / 60	4 min 57 s	~200 tok/s	74,342
completed · no OOM	~5.0 s / task	~3B active · bw-bound	tokens · 12.7k / 59.4k

The 100% converts directly into the agenda →

Stratify difficulty	Execution-ground it	Show the gap	Frontier anchor
beginner → adversarial (C3)	verified ≠ completed (C2)	fresh - stale (C1)	cross-repo (C4/C5)

Reading the 100%. By the discrimination criterion this lab applies to every rubric – a task set on which every sample passes (or every sample fails) measures nothing – a 60/60 result is not a capability claim. It says the generated tasks were below the model's ceiling. That is the pilot's central finding: the extraction pipeline runs end-to-end, deterministically, within the VRAM budget, but the difficulty distribution needs to be raised before the numbers carry signal. The measurement that is trustworthy here is the operational one – throughput, stability, token accounting under a pinned single stream.

§05

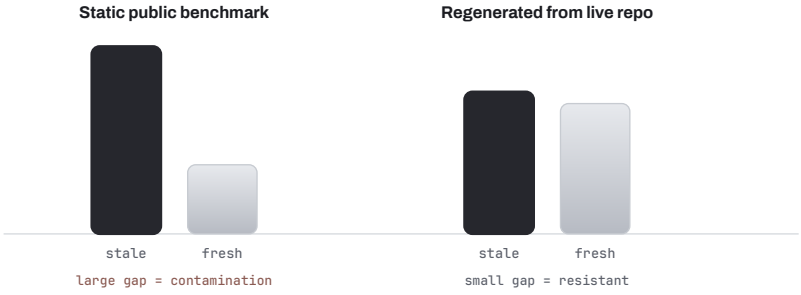
What the pilot shows – and what it does not

The pilot's honesty about the 100% is its strongest feature; the same honesty applies to two properties the run does not establish, each of which §6 turns into a test (Figure 4).

Contamination resistance is a property of the method, not yet a demonstrated result. Cerberus is a mature, widely-used repository whose code predates the model's training cutoff and is almost certainly in the training corpus. Synthesising tasks from its structure yields fresh task phrasings, but the underlying code and its idioms are plausibly memorised, so the run cannot separate genuine capability from recall. What the pilot does establish is the necessary substrate for contamination control – tasks bound to a pinned commit, re-derivable deterministically from (commit, extraction rules) – not that the model scores lower on truly unseen code. Establishing resistance requires the fresh-versus-stale comparison of Figure 4 (criterion C1), run against repositories or commits that post-date the model.

What “contamination-resistant” has to mean

Resistance is the gap between performance on stale (likely-seen) and fresh (post-cutoff) targets.



The metric (W2)

contamination gap =
score(stale) - score(fresh)

fresh = repos/commits
after the cutoff.

"Completed" is not "correct". The pilot's pass criterion is that the model produced a syntactically valid, on-task response; it does not execute the generated code against tests. The field's standard for code evaluation is execution-grounded – success means tests pass after the model's change – and by that standard the pilot reports completion, not verified correctness [3, 7, 9]. This is a boundary of what was measured, not a defect of the run, and the fix is direct: route generated outputs through execution verification (compile, generated/property tests, linter

and type-checker), i.e., the execution-grounded judging built in the Hybrid Evaluation Pipeline (W1) [13]. It matters most in the security_audit category, where a claimed vulnerability finding should be confirmed by an actual reproduction, not accepted as text.

Read together, these are not caveats bolted on after the fact; they are the precise reason the pilot refuses to report a capability number, and they define the next two experiments.

§06

Evaluation plan: making contamination resistance and correctness measurable

Following the discipline of the companion papers, the claims this method is meant to support are stated as falsifiable criteria with an instrument and a threshold, marked predicted until measured. The workstream hypothesis (W2) is: evaluation tasks regenerated from live repositories resist memorisation – a model's advantage on seen code does not transfer to structurally-matched unseen code drawn by the same pipeline. Its headline metric, per the research roadmap, is the contamination gap.

- **C1 – Contamination gap.** Metric: score(stale) – score(fresh), where fresh = tasks synthesised from repositories or commits after the model's knowledge cutoff and stale = tasks from pre-cutoff, well-known code (e.g. Cerberus). Pass: the pipeline exhibits a small gap where a static public benchmark exhibits a large one, on the same model. Falsification: the gap on the regenerated set is as large as on the static set – the method adds no resistance. (predicted)
- **C2 – Execution-grounded correctness.** Claim: a "pass" should be verified, not asserted. Instrument: the W1 execution harness and frozen-rubric judge – generated code must compile and pass generated/property tests; a located bug must be confirmed by a failing→passing test; an audit finding must be reproduced [13]. Pass: every reported pass is an execution-verified outcome. Falsification: execution verification overturns a material fraction of "completed" passes. (predicted)
- **C3 – Discrimination.** Claim: the task set must separate strong from weak output. Instrument: stratify difficulty (beginner → adversarial) until the pass rate desaturates; report the difficulty distribution and an item-level discrimination index, not a single pass rate. Pass: a difficulty band where the model's pass rate is informative (neither 0 nor 100%). (predicted)
- **C4 – Cross-model, cross-repo external validity.** Instrument: identical measured conditions across models (DeepSeek-R1 32B, Gemma 4-31B, Mistral NeMo 12B) and repositories (FastAPI async routing, Pydantic validation metaclasses, Django REST) via the GitHub API – no simulated rows. Pass: stable, repo-appropriate rankings and gaps. (predicted)
- **C5 – Frontier anchor and hallucination check.** Instrument: anchor local scores against a frontier commercial model via a batch API, and cross-reference generated imports and API calls against real package documentation to catch fabricated dependencies before they read as valid code. Pass: a quantified local-versus-frontier gap and a reported fabricated-dependency rate. (predicted)

C1 and C2 are the two that convert the pilot's honest boundaries (§5) into measured results; C3–C5 absorb and formalise the original next-steps list. All are runnable on the same single-GPU, air-gapped setup plus the evaluation pipeline already in the program, which is the point: the whole plan is verifiable under the sovereignty constraint the pilot was built to honour.

§07

Discussion

Hardware-constrained parallelism. With 32 GB of VRAM and Q4_K_M weights around 19–24 GB per model, true parallel multi-model evaluation is not possible on this hardware without a CPU fallback that corrupts the very latencies being measured. The honest architectural consequence is strict sequential execution – which this pipeline enforces rather than fights, and which the workstation companion paper treats as a first-class design (single-residency time-multiplexing).

Quantisation. A 35B model at FP16 would need ~70 GB for weights alone. Q4_K_M packs weights into super-blocks of 256 (eight 32-weight sub-blocks), storing 4-bit quants with 6-bit per-sub-block scales and mins – about 4.5 effective bits per weight – and promotes a subset of tensors (notably the value projections and part of the feed-forward path) to Q6_K. That mixed precision brings the footprint to ~24 GB while preserving enough of the weight distribution for syntactic validity to hold at this scale. It is the enabling trick for the whole exercise. The known caveat, and the reason C2/C5 matter: low-bit quantisation preserves surface validity more reliably than deep multi-step reasoning, so correctness under Q4_K_M is exactly what execution grounding must confirm rather than assume [12].

Throughput, correctly attributed. ~200 tokens/s for a "35B" model is not paradoxical: it is a ~3B-active MoE, and decode cost tracks the active path. Attributing the speed to the full parameter count (as a dense reading would) would be wrong; a dense 32B on the same card runs roughly half to a quarter of this rate.

§08

Applied scenario (STAR)

Vetting an open-weight model for an air-gapped validation library.

- **Situation.** A team maintains a security-critical Python input-validation library (the Cerberus problem class: schema coercion, nested and conditional rules). They want to adopt an open-weight coding model, but the codebase is proprietary and under a data-sovereignty constraint – nothing may leave the network. Public coding benchmarks are contaminated, so a leaderboard rank is no evidence for this codebase.
- **Task.** Produce a defensible, on-premises measurement of whether the candidate model handles this repository's edge cases and injection-vector reasoning – reproducible from the exact commit, inside a single 32 GB GPU, with no cloud dependency in the evaluation path.
- **Action.** Point the AST extractor at the pinned commit; synthesise 60 tasks bound to the repo's real signatures and imports; run Qwen3.6-35B-A3B at temperature 0 under `NUM_PARALLEL=1`; keep the run air-gapped; log latency, token counts and finish reasons per task into SQLite via one isolated promptfoo config.
- **Result.** A complete 60-task evaluation in 4 min 57 s, entirely on-premises and re-derivable from (commit, extraction rules) – defensible in an audit months later. The uniform pass rate surfaced a concrete methodological gap (the task set does not discriminate at this model's level), which converted directly into the next action: stratify difficulty and add adversarial audit cases before any score is read as a capability signal. The deliverable is a trustworthy, sovereign measurement and a precise instruction for the next iteration – not a number taken on faith.

§09

Limitations

Beyond the two boundaries made explicit in §5 (contamination resistance not yet demonstrated; completion ≠ execution-verified correctness), three residual limitations stand. First, **AST-synthesised tasks are a different family from issue-resolution tasks:** extracting a function from a signature-plus-imports is structurally cleaner than "resolve this GitHub issue so its hidden tests pass," so the pipeline complements rather than replaces SWE-bench-style evaluation, and its difficulty ceiling is one reason the pilot saturated. Second, **single operator, single hardware** bounds the external validity of the operational numbers themselves (throughput, latency), independent of the capability question. Third, the **security_audit category currently scores identification, not exploitation** – a finding that reads as correct may not be, which is precisely why C2's reproduction requirement is load-bearing for that category.

§10

Status & reproducibility

This is a pilot with measured operational results and deliberately no capability claim. Reproducibility pins: `OLLAMA_NUM_PARALLEL=1`, `OLLAMA_MAX_LOADED_MODELS=1`, `OLLAMA_FLASH_ATTENTION=1`, `OLLAMA_KV_CACHE_TYPE=q8_0`; deterministic sampling (temperature 0.0); maxTokens 4096 as a VRAM guard; one isolated promptfoo YAML per model, results aggregated to a single SQLite store; target pinned by commit, task synthesis a pure function of (repo state, extraction rules). Built and operated end-to-end under CTC AI Operations, on the same local-inference and falsifiable-claim discipline as the evaluation-pipeline, workstation, and assistant projects it sits beside; C2's correctness verification reuses the W1 execution harness directly, closing the loop between the two workstreams.

References

[1] Recent Advances in LLM Benchmarks against Data Contamination: From Static to Dynamic Evaluation. (2025). arXiv:2502.17521. [2] Jain, N., Han, K., Gu, A., et al. (2024). LiveCodeBench: Holistic and Contamination-Free Evaluation of LLMs for Code. arXiv:2403.07974. [3] Zhang, L., et al. (2025). SWE-bench Goes Live! (SWE-bench-Live; REPOLAUNCH pipeline). arXiv:2505.23419. [4] Code2Bench: Dynamic, Contamination-Resistant Benchmark Construction from Real-World Repositories. (2025). arXiv:2508.07180. [5] Jain, N., Singh, J., Shetty, M., et al. (2025). R2E-Gym: Procedural Environments and Hybrid Verifiers for Scaling Open-Weights SWE Agents. arXiv:2504.07164. [6] Yang, J., et al. (2025). SWE-Smith: Scaling repository-level software-engineering task generation. [7] Jimenez, C. E., Yang, J., Wettig, A., et al. (2024). SWE-bench: Can Language Models Resolve Real-World GitHub Issues? ICLR. arXiv:2310.06770. [8] Scale AI. (2025). SWE-bench Pro (contamination-resistant via held-out commercial repositories). [9] Liu, J., Xia, C. S., Wang, Y., Zhang, L. (2023). Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code (EvalPlus). NeurIPS. arXiv:2305.01210. [10] Siddiq, M. L., Santos, J. C. S. (2022). SecurityEval; Hajipour, H., et al. (2024). CodeLMSec – security-oriented code-generation benchmarks. [11] Qwen Team. (2026). Qwen3.6-35B-A3B – 40 layers as 10 × (3 Gated-DeltaNet → 1 Gated-Attention), 256 experts (8 routed + 1 shared); hybrid linear + gated attention; GQA (2 KV heads, head-dim 256). Hugging Face model card. [12] Li, Z., Su, Y., Yang, R., et al. (2025). Quantization Meets Reasoning: Exploring LLM Low-Bit Quantization Degradation for Mathematical Reasoning. arXiv:2501.03035. [13] Arenskrieger, M. E. (2026). Hybrid Evaluation Pipeline (CTC AI Operations, W1) – execution-grounded, frozen-rubric judging and hardened sandbox reused for the correctness verification of §6 (C2).