Contamination-Resistant Code Evaluation on Real Codebases

A single-model pilot with Qwen3.6-35B-A3B on the Cerberus validation framework

● Pilot – single model, single repository

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

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 here is the method and its VRAM discipline, not a leaderboard number.

§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.

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

Methodology

2.1 Structural task synthesis

Tasks are produced by a multi-stage static analysis rather than authored by hand:

```
Repo 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.

2.2 Model under test

Property	Value
Model	Qwen3.6-35B-A3B (`qwen3.6:35b`)
Architecture	**Sparse Mixture-of-Experts** – 35B total parameters, **~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, 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. 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.

2.3 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.

2.4 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.

2.5 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.

§03

Results

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

```
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.

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.

§04

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.

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.

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.

§05

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.

§06

Limitations & next steps

1. **Single repository.** All tasks derive from Cerberus; external validity requires diverse targets. The extraction engine already generalises – pointing it at FastAPI (async routing), Pydantic (validation metaclasses) or Django REST via the GitHub API is the immediate cross-domain extension. 2. **Single model.** This is a one-model pilot by design. A proper cross-model comparison (DeepSeek-R1 32B, Gemma 4-31B, Mistral NeMo 12B) is deferred until it can be run under identical measured conditions – no simulated rows. 3. **Non-discriminative task set.** Difficulty must be stratified (beginner → adversarial) so the pass rate stops saturating; this is the precondition for any capability claim. 4. **No commercial baseline.** A frontier-API gold standard (e.g. a frontier commercial model) via a batch API would anchor the local scores. 5. **Hallucination detection.** Cross-referencing generated imports and API calls against real package documentation would catch fabricated dependencies before they read as "valid" code.

Appendix – reproducibility notes

- Environment 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 is a pure function of (repo state, extraction rules).

Built and operated end-to-end under CTC AI Operations.