Passive Token Accounting For Cost-Aware LLM Assistance In Continuous SoC Verification Workflows

Ajith Jose
Dhenara Inc.
ajith.jose@dhenara.com

Abstract - Large-language models (LLMs) are increasingly embedded in pre-silicon verification flows: log summarization, failure clustering, specification Q&A, and automated debug planning. Teams report strong productivity gains but face a new operational risk: token spend. Continuous regressions can generate tens of thousands of logs; each LLM call, however small, accumulates into non-trivial cloud bills or competes with on-prem GPU capacity. Traditional EDA infrastructure has no native notion of "LLM cost," leaving verification leads unable to answer: What did the triage bot cost last week? Which tests burn the most tokens? Did a heuristics change double our API bill?

We present Passive Token Accounting (PTA), an implementation in the dVI™ (Dhenara Verification Intelligence) product line that provides provider-agnostic, low-overhead cost observability without modifying existing test or orchestration scripts. PTA extracts usage from model responses, normalizes costs across providers, and continuously aggregates per-regression snapshots consumed by a lightweight API and UI. The approach enables per-phase budgets, model downgrades under soft caps, and fine-grained attribution to tests and agents—all while preserving provider portability.

I. Introduction

The last two years have seen LLMs adopted across verification flows. Yet most teams treat the LLM runtime as a black box. Costs can spike due to long logs, cascading failures, or subtle retry loops. This is more chaotic in agentic workflows where a large number of agents may be making concurrent calls with multiple turns. Because regressions run continuously, even moderate waste compounds into material spend. An effective solution needs to be:

- **Transparent:** Works with any LLM provider and existing code
- **Low-overhead:** No perceptible slowdown to regressions
- **Actionable:** Exposes per-regression and per-agent breakdowns
- **Enforceable:** Supports soft caps and budget-aware policies

PTA meets these needs in dVI™ by integrating cost telemetry into the LLM layer (***dhenara-ai***[1]), propagating normalized metrics to the agent runtime, and aggregating them at the regression boundary as a single source of truth.

Contributions: (1) a provider-agnostic usage and cost schema captured at the LLM client layer (dhenara-ai[1]) rather than a single provider SDK; (2) execution-level aggregation suitable for multi-step / tool-using agent workflows; and (3) a regression-scoped, append-friendly snapshot that enables attribution and budget-aware policies without modifying existing orchestration.

II. Background and Related Work

Many teams rely on provider dashboards, bespoke wrappers around a single SDK, or general observability pipelines to understand LLM usage. These approaches can be valuable but often do not travel well across providers, and they can fragment metrics across tools and environments. PTA instead relies on a provider-agnostic type system in dhenara-ai [1] that captures usage consistently:

dhenara-ai is MIT-licensed open source [1]. It is available publicly, and its provider adapters and typed response schema can be used as-is as a standalone Python package.

- **ChatResponseUsage**: prompt, completion, total, and optional reasoning tokens
- **UsageCharge**: normalized USD cost and optional charged amount (cost × margin)

This allows dVI™ to support OpenAI[2], Anthropic[3], Google AI[4], Azure, Bedrock and local model servers with uniform accounting. The same schema also backs the agent framework (DAD) to surface per-node usage and aggregate per-agent cost, avoiding bespoke adapters per provider.

III. DVI Architecture Overview (where PTA lives)

- **dhenara-ai:** Typed, provider-agnostic LLM client that parses provider responses and emits usage + charge in a common schema. (Truly open source PyPI package, published[1] in Github)
- **dhenara-agent-frameworks** (Internal): agent runtime that collects node-level usage and aggregates totals per execution
- **dVI™** (Dhenara Verification Intelligence) : product workflow that runs curators and deep-debug agents on regression artifacts
- **Backend Gateway**: Stateless backend that serves costs for the UI from on-disk snapshots

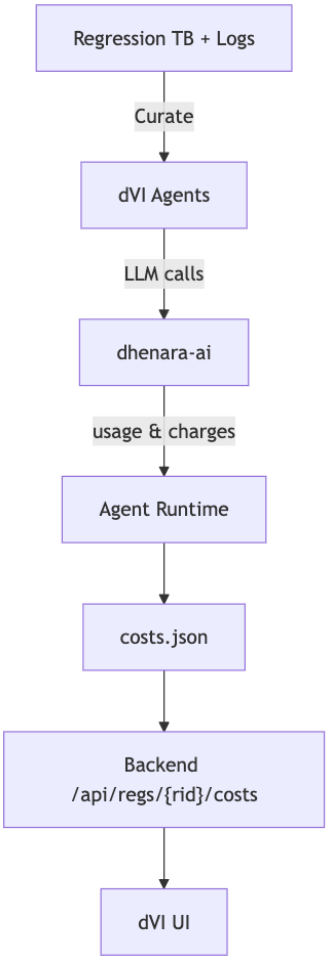

Fig.1 – dVI™ Architecture

IV. Passive Token Accounting: Design

PTA has three layers that map directly to the codebase.

1 Provider-Agnostic Usage Extraction (dhenara-ai)

This layer captures usage in a provider-agnostic form directly from the model response.

- For OpenAI Responses API, Anthropic Messages, Google GenAI APIs, and local engines, the LLM client parses provider responses to fill ChatResponseUsage: total_tokens, prompt_tokens, completion_tokens, and reasoning_tokens (if reported by the provider).

- A model cost table maps usage → cost via ChatModelCostData:
    - *cost = prompt_tokens × input_rate + completion_tokens × output_rate*.
- A cost_multiplier_percentage optionally derives customer-visible "charge," enabling transparent pass-through or markup when needed.
- The same schema is present in streaming: token deltas accrue and are consolidated in a final usage report.

2 Execution-Level Aggregation (agent runtime)
This layer ensures accounting remains accurate across multi-step agent workflows.

- Agents run as DAGs. The AI-model nodes attach usage and charge to each node execution result.
- Component executors aggregate shallow child usage to produce a per-agent total
    - a*gg_usage_cost, agg_usage_charge, agg_usage_prompt/completion/total_tokens*.
- This keeps accounting accurate even when complex tool-using plans iterate or branch.

3 Regression-Level Snapshot (dVI™)
This layer persists a regression-scoped accounting "source of truth" artifact.

- Each agent run records into a regression-scoped *costs.json* snapshot using record_cost_entry(*reg_id=..., agent_name=..., exec_id=..., ...)*.
- The snapshot schema is intentionally simple and append-friendly:
    - version, last_updated
    - totals: total_cost, total_charge, total_prompt_tokens, total_completion_tokens, total_tokens
    - per_agent: map with cost, charge, token totals, run count, last_run_at
    - runs: list of run entries with exec, agent, kind (agent-loops or dad-agent), status, timestamps, aggregated usage, and sanitized metadata (e.g. test name, used tools)
- Deep-debug agents additionally read ai-orchestrator usage.json(l) when present to incorporate per-phase token/cost data into the snapshot.

The backend gateway exposes this snapshot at */api/regs/{rid}/costs* for the UI. No database is used during evaluations; the gateway is stateless and reads JSON directly from the artifact root.

V. Implementation Notes with Code Anchors

- Usage and charge types: *dhenara_ai/.../ai_model/_metrics.py* (ChatResponseUsage, UsageCharge)
- Cost calculation: *dhenara_ai/.../ai_model/_ai_model.py* (ChatModelCostData.calculate_usage_charge)
- Provider parsing (example): *dhenara_ai/.../providers/openai/responses.py*
    - Extracts total/prompt/completion and reasoning tokens; attaches usage and charge to ChatResponse
- Agent aggregation (open-source variant): *dhenara_agent/.../dsl/base/component/executor.py* (agg_usage_cost, agg_usage_charge, tokens)
- Internal code refs
    - Multiple Agents, Regression snapshot, Backend API exposure

These anchors are provided to support reproducibility; the design works similarly across providers and execution modes (sync/async, streaming).

Public evidence (dhenara-ai types)

The provider-agnostic types below are from the open-source dhenara-ai package (see References). They illustrate the normalized schema used throughout PTA:

```python
class UsageCharge(BaseModel):
  cost: float
  charge: float | None

class ChatResponseUsage(BaseModel):
  total_tokens: int
  prompt_tokens: int
  completion_tokens: int
  reasoning_tokens: int | None
```

Cost calculation converts per-million token rates to per-token costs:

```
cost = round(
  usage.prompt_tokens * (input_per_million/1_000_000)
  + usage.completion_tokens * (output_per_million/1_000_000),
  6,
)
```

VI. Policy Controls: Budgets, Soft Caps, and Model Selection

dVI™ exposes budget-aware knobs close to where they matter:

- **Per-phase limits in deep-debug:** max_subloop_cost, max_subloop_tokens, max_subloop_runs/depth/time via controls in the configuration (see DeepDebug SubloopLimits). These enable walking back exploration before it becomes expensive.
- **Model selection per phase:** individual phase execution models can be distinct; the system allows lightweight models or higher-quality models per agents/ node.
- **Soft caps at the regression level:** enforced operationally by reading the costs snapshot and adjusting model choices (e.g., downgrading for the remainder of a nightly run). Because usage and cost are normalized at the LLM layer, such policies remain provider-agnostic.

VII. Integration with Continuous Verification (dVI)

PTA is intentionally passive: it requires no changes to generator/testbench code or to regression orchestration. It works in both local and farm settings. dVI™ writes artifacts under dedicated artefact dir and maintains per-regression state in a .dad directory, including costs.json. The UI then surfaces:

- Current total cost and tokens for the regression
- Per-agent breakdowns
- A run list with timestamps and links to summaries

This enables predictable, budget-aware operation of LLM-assisted flows alongside traditional EDA metrics.

VIII. Evaluation

We report three practical outcomes based on the open test bundle and internal experience integrating PTA in test suits.

- Negligible overhead: extracting usage from provider responses and writing JSON snapshots adds no measurable latency to agent runs (I/O is local and amortized; the LLM call dominates runtime).
- Actionability: cost visibility revealed that a small subset of failing tests produced the majority of token spend due to deep stack traces in logs. Enabling selective truncation in the curator reduced cost with no impact on clustering and triage.
- Reproducibility: the repository includes a sandboxed test bundle (verification/dvi_test_bundle) with make targets to generate logs and verify agent behavior end-to-end. While the bundle is sized for developer machines, it exercises the same PTA plumbing as larger farms.

1 Local runs across providers

We ran Deep Debug on the same failing test across three LLM providers during an RTL simulation run. The DUT and testbench were derived from the open-source CORE-V repositories[5-7]. PTA captured token usage consistently in usage.json, including an aggregate summary row:

- Anthropic (Claude Sonnet 4.5):


```
{
  "prompt_tokens": 78249,
  "completion_tokens": 8458,
  "total_tokens": 86707,
  "cost": 0.361617
}
```


- OpenAI (GPT-5 mini):


```
{
  "prompt_tokens": 57513,
  "completion_tokens": 17184,
  "total_tokens": 74697,
  "cost": 0.048747000000000006
}
```


- Google AI (Gemini 2.5 Pro):


```
{
  "prompt_tokens": 79620,
  "completion_tokens": 10847,
  "total_tokens": 90467,
  "cost": 0.20799499999999999
}
```


Example line from usage.jsonl (provider, model, phase):


```
{
    "phase": "execute",
    "loop": 0,
    "model": "claude-sonnet-4-5-20250929",
    "provider": "anthropic",
    "prompt_tokens": 9000,
    "completion_tokens": 1582,
    "total_tokens": 10582,
    "cost": 0.05073
}
```


Listing 1: example PTA envelope JSON (schema excerpt)

Summary table (same test, different providers/models):

Provider	Model	Total tokens	Cost (USD)
Anthropic	claude-sonnet-4-5-20250929	86,707	0.361617
OpenAI	gpt-5-mini-2025-08-07	74,697	0.048747
Google AI	gemini-2.5-pro	90,467	0.207995

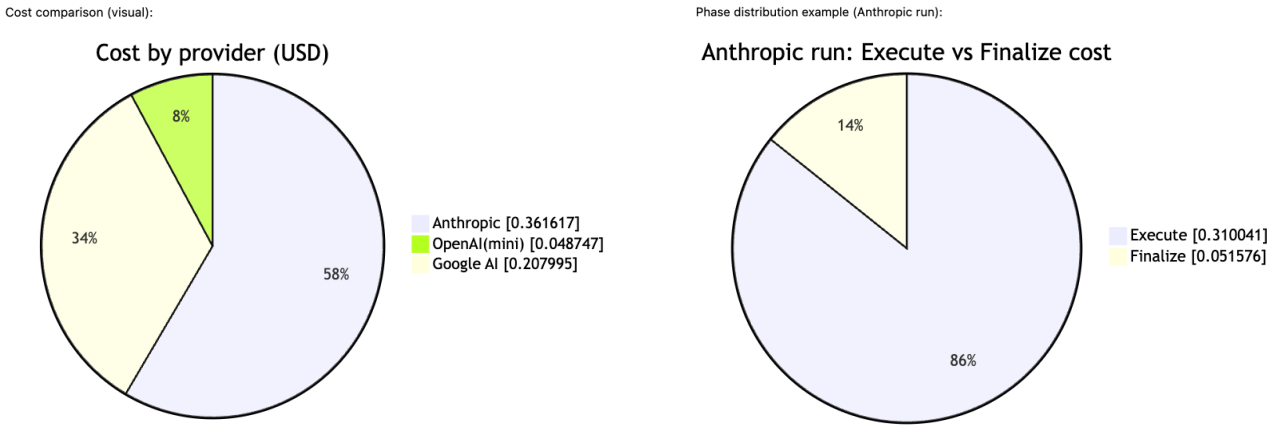

Fig.2 – Token/cost summary for representative runs

Notes:

- The absolute numbers reflect per-provider pricing tables applied to identical token counts; PTA's normalization allows direct comparison without provider-specific code.
- *usage.jsonl* also exposes per-phase slices enabling budget-aware policies (e.g., lower-cost model for multi phase execution loops, higher-quality model for reasoning).

IX. Discussion

- Provider portability: capturing usage at the client layer keeps accounting consistent even when switching providers or model families. The same agent logic and UI continue to work.
- Attribution: per-run entries include sanitized metadata (test name, tool usage, run IDs), which supports drill-downs and forensics without leaking sensitive contents.
- What we do not do: PTA in dVI™ deliberately avoids invasive HTTP interception or LD_PRELOAD shims in production builds. Instead, it relies on structured responses surfaced by modern SDKs. For local models that do not emit usage, the client can fall back to byte counts or configured rates.

X. Threats to Validity and Limitations

- Provider reporting gaps: some providers omit usage on failures or midway through streaming. We address this with conservative aggregation (final usage beats deltas) and optional artifact-level usage.json emitted by multi agents.
- Local models without native token metrics: cost must be approximated (byte count or configured per-second costs) and can under/over-estimate actual GPU spend. Alternatively a light weight SDK can be developed based on *dhenara-ai* [1] package.
- Budget enforcement scope: cost-aware stopping is currently per agent/sub-agent; cross-regression global budgets are enforced operationally via the gateway/UI.

XI. Future Work

- Unified live dashboard: expose Prometheus metrics from the gateway to allow alerting and external dashboards without a database.
- Dataset: publish anonymized PTA logs from larger nightly regressions to seed benchmarking of cost-aware heuristics in hardware verification.
- Policy language: lightweight rules ("if daily_cost > X use model Y") defined in config, with dry-run and audit modes.

XII. Conclusion

Passive Token Accounting in dVI™ delivers the missing piece for sustainable LLM usage in continuous verification: provider-agnostic, low-overhead cost observability tied directly to regression artifacts. By normalizing usage and costs at the LLM layer, aggregating at the agent boundary, and exposing a simple per-regression snapshot, teams can set budgets, pick models wisely, and keep assistance switched on—without rewriting flows or locking into a single provider.

XIII. References

[1] dhenara-ai (Open-Source). Provider-agnostic LLM client and types. GitHub: https://github.com/dhenara/dhenara-ai
[2] OpenAI API. "Usage and Rate Limits; Responses API Usage fields," https://platform.openai.com/docs
[3] Anthropic API. "Token usage and reasoning tokens," https://docs.claude.com/en/api/overview
[4] Google Gemini API. "Token accounting and pricing," https://ai.google.dev/api
[5] OpenHW Group. "CORE-V CV32E40P," GitHub: https://github.com/openhwgroup/cv32e40p
[6] OpenHW Group. "core-v-verif," GitHub: https://github.com/openhwgroup/core-v-verif
[7] Google/Chipsalliance "riscv-dv: Random Instruction Generator for RISC-V," GitHub: https://github.com/google/riscv-dv

(All sites accessed Oct 2025).

Appendix A: Selected Schemas (for reproducibility)

- ChatResponseUsage: {total_tokens, prompt_tokens, completion_tokens, reasoning_tokens?}
- UsageCharge: {cost, charge?}
- costs.json (regression snapshot):
    - totals: total_cost, total_charge, total_prompt_tokens, total_completion_tokens, total_tokens
    - per_agent: map → {cost, charge, prompt_tokens, completion_tokens, total_tokens, runs, last_run_at}
    - runs: list → {exec, agent, kind, status, start/end, agg_usage_* fields, summary_file?, run_dir?, metadata}

Notes: Field names are stable at the time of writing and can be inspected in the referenced files in the repository.

Appendix B: Test Bundle Sources (submodules used for log generation)

For reproducibility, the cv32e40p test bundle uses the following external open-source repositories as submodules (read-only; used solely to generate logs):

- rtl/externals/cv32e40p[5] @ e1891cd
- tb/externals/core-v-verif[6] @ d4b94d4
- tb/externals/riscv-dv[7] @ 7e54b67

We acknowledge these projects in References; they are not modified and are included to create realistic, sandboxed RTL/TB logs.