AgentBudget: Real-Time Cost Enforcement for AI Agents

An open-source framework for per-session budget enforcement, circuit breaking, and cost attribution in agentic systems

Sahil Jagtap
AgentBudget
sahil.jagtap45@gmail.com
https://agentbudget.dev

February 2026

Abstract

We present AgentBudget, an open-source Python SDK that provides real-time, dollar-denominated cost enforcement for AI agent sessions. AI agents introduce fundamental cost unpredictability into software systems: unlike deterministic APIs, an agent decides at runtime how many inference calls to make, which models to use, and which external tools to invoke. A single misconfigured agent can consume hundreds of dollars in minutes without detection. Existing observability tools track costs after execution but do not prevent overspend in real time.

AgentBudget addresses this gap with a per-session budget envelope that wraps LLM calls, tool invocations, and external API requests in a running cost ledger with automatic circuit breaking. The system supports drop-in integration via SDK monkey-patching (requiring zero code changes to existing agents), two-phase enforcement (pre-call estimation and post-call reconciliation), loop detection, nested budgets for multi-agent systems, and structured cost reporting. We discuss the system's architecture, its relationship to the x402 autonomous payment protocol [1], and its positioning within the emerging agent cost control ecosystem.

Keywords: AI agents, cost enforcement, budget management, circuit breaking, LLM observability, x402, autonomous payments

1 Introduction

The deployment of AI agents in production environments represents a paradigm shift in how software systems consume computational resources. Traditional software exhibits deterministic resource usage: a given API endpoint executes a predictable sequence of database queries and external service calls, and its per-request cost is knowable in advance. This property enables straightforward capacity planning and cost forecasting.

Agent-based systems break this fundamental assumption. An AI agent operating under a goal-oriented framework decides at runtime how many large language model (LLM) inference calls to make, which models to select (with costs varying by up to 15× across model tiers [2]), which external tools to invoke, and how many retry attempts to execute. The same user request may cost $0.02 or $20.00 depending on the agent's reasoning path.

Recent industry surveys indicate that average monthly AI spending reached $85,521 in 2025, representing a 36% increase from the prior year [3], with respondents reporting that a majority of engineering teams are "one prompt away from a budget disaster" [3].

This paper presents AgentBudget, a lightweight, open-source SDK that provides the missing enforcement primitive: per-session, dollar-denominated budget caps with real-time circuit breaking. We describe the system's design, technical specification, and its relationship to the x402 autonomous payment protocol [1].

2 Motivation

2.1 The Cost Unpredictability Problem

We identify four distinct failure modes arising from agent cost unpredictability:

Infinite Loop Failure. An agent enters a reasoning or retry loop, invoking the same tool or LLM call with similar parameters repeatedly without converging. At current inference prices ($0.03–$0.15 per GPT-4o call [2]), a loop generating 200 calls in 10 minutes produces $30–$50 in charges before any monitoring system detects the anomaly. Conventional rate limiting is insufficient because each individual call appears legitimate.

Cost Opacity. Agent frameworks typically report token counts, but tokens do not map linearly to dollars. A GPT-4o inference costs approximately 15× more than GPT-4o-mini for equivalent token volumes [2]. When agents combine LLM inference with paid external APIs (e.g., search at $0.01/call, data enrichment at $0.50/call), no single system provides a unified dollar-denominated view of session cost.

Multi-Provider Fragmentation. A single agent session may invoke OpenAI for reasoning, Anthropic for analysis,

Google for search grounding, and multiple SaaS APIs for data retrieval. Each provider maintains an independent billing system with different reporting latencies. Attributing the total cost of one session requires manual aggregation across multiple dashboards.

Scaling Amplification. At production scale—hundreds or thousands of concurrent sessions—even a low failure rate (e.g., 5%) produces dozens of runaway sessions per hour. Without per-session enforcement, costs become a function of worst-case agent behavior multiplied by session volume, rendering unit economics unforecastable.

2.2 Limitations of Existing Approaches

The current landscape provides observability and coarse-grained controls but lacks real-time per-session enforcement. Table 1 summarizes the capabilities and limitations of existing tools.

Table 1: Comparison of existing agent cost control approaches.

Tool	Capability	Limitation
LangSmith [4]	Per-trace cost tracking, dashboards	Post-hoc only; no enforcement
LangChain [5]	Call count limits per run	Counts calls, not dollars
Langfuse [6]	Open-source LLM observability	Monitoring only; no circuit breaking
Lava AI Spend [7]	Per-API-key dollar caps	Key-level only; no session budgets
Provider limits	Monthly org-level caps	Too coarse for session-level

The gap AgentBudget fills: *real-time, dollar-denominated, per-session budget enforcement* that spans LLM calls, tool calls, and external APIs—with automatic circuit breaking when the limit is reached.

3 System Design

3.1 Design Principles

AgentBudget is designed around four principles:

1. **Dollar-denominated enforcement.** Budgets are expressed in dollars, not tokens or call counts, because dollars are the unit operators ultimately care about.

2. **Zero-infrastructure deployment.** The system operates as an in-process library with no external dependencies—no Redis, no cloud services, no daemons.

3. **Drop-in integration.** The primary path requires zero modifications to existing code, following the monkey-patching pattern established by Sentry [8] and Datadog APM [9].

4. **Composability.** The system produces structured cost data consumed by any downstream system without coupling.

3.2 Drop-in Integration

The primary integration method requires two lines of code. The `agentbudget.init()` function patches the OpenAI and Anthropic client SDKs at the module level, intercepting all inference calls automatically:

```
import agentbudget
agentbudget.init("$5.00")

# Existing code -- no changes needed
client = openai.OpenAI()
response = client.chat.completions.create(
    model="gpt-4o",
    messages=[{"role": "user",
               "content": "..."}]
)
# Cost tracked automatically
# BudgetExhausted raised at limit

agentbudget.teardown()
```

This approach is framework-agnostic: any system using the OpenAI or Anthropic Python SDKs—including LangChain, CrewAI, AutoGen, and custom implementations—is automatically instrumented without framework-specific adapters.

3.3 Manual Integration

For explicit control, a context manager API provides direct session management:

```
from agentbudget import AgentBudget
budget = AgentBudget(max_spend="$5.00",
                     soft_limit=0.9)
with budget.session() as s:
    resp = s.wrap(
        client.chat.completions.create(...)
    )
    result = s.track(
        api_call(), cost=0.01,
        tool_name="search"
    )
```

3.4 Architecture

Figure 1 illustrates the system architecture. The monkey-patching layer intercepts SDK calls and routes them through the LLM Cost Engine (for inference) or Tool Cost Registry (for annotated tool calls). Both feed into a shared Budget Ledger that maintains the running balance and enforces limits via the circuit breaker.

4 Technical Specification

4.1 Cost Engine

The LLM Cost Engine maintains a pricing table mapping (*provider*, *model_id*) tuples to (*input_cost_per_token*, *output_cost_per_token*) values. The current implementation covers 50+ models across five providers: OpenAI, Anthropic, Google (Gemini), Mistral, and Cohere. Upon intercepting an LLM response, the engine computes:

$$C = T_{\text{in}} \times P_{\text{in}} + T_{\text{out}} \times P_{\text{out}} \tag{1}$$

where T_{in} and T_{out} are input and output token counts, and P_{in} and P_{out} are per-token prices. Dated model variants (e.g., `gpt-4o-2025-06-15`) are resolved to base model pricing via fuzzy string matching. Custom model pricing may be registered at runtime via `agentbudget.register_model()`.

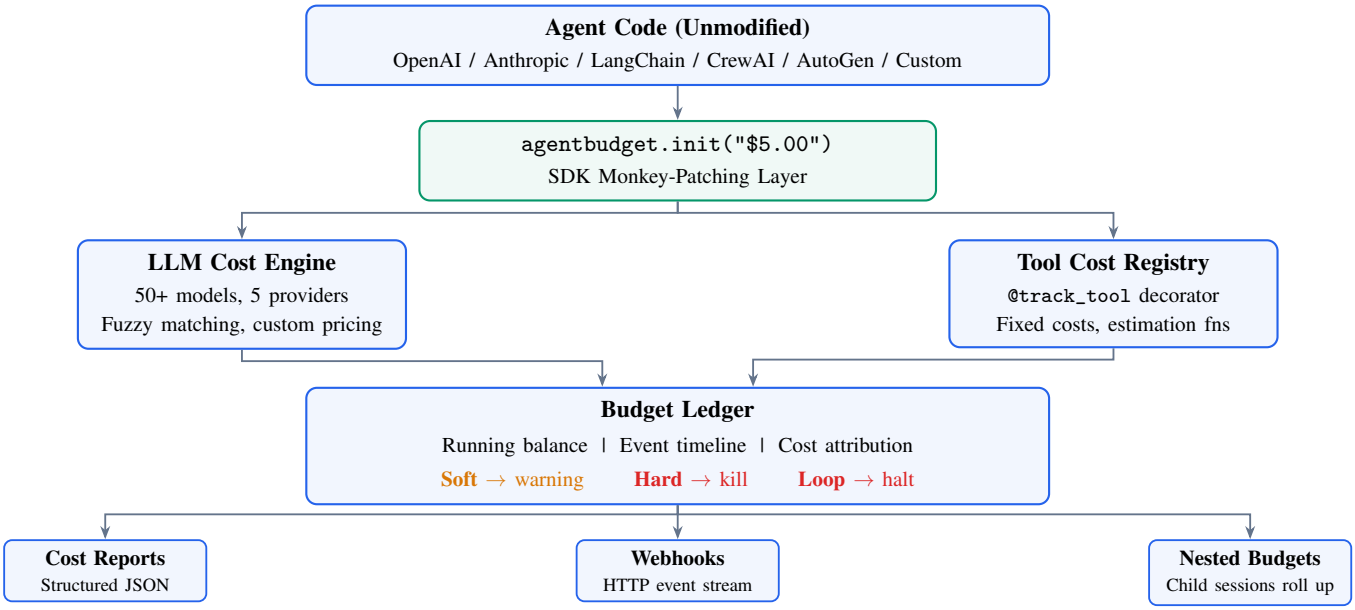

Figure 1: AgentBudget system architecture. The monkey-patching layer intercepts SDK calls transparently, routing them through the cost engine and tool registry into a shared budget ledger with three-level circuit breaking.

4.2 Two-Phase Enforcement

A key design challenge is determining when to enforce the budget relative to an LLM call. Enforcing only post-call means the cost has already been incurred; enforcing only pre-call requires estimating unknown output length. AgentBudget resolves this with two-phase enforcement (Figure 2):

Phase 1 (Pre-call): Before the inference call is dispatched, the engine estimates cost from input token count plus an average completion length for the target model. If the estimate exceeds remaining budget, the call is rejected *before* the provider API is contacted—no cost is incurred.

Phase 2 (Post-call): After the response is received, actual token usage is extracted and true cost is computed. The ledger is updated. If the session is now over budget, subsequent calls are blocked. In the worst case, overshoot is bounded to the cost of a single call.

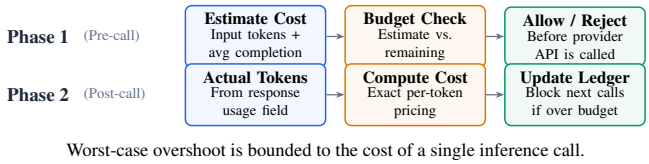

Figure 2: Two-phase budget enforcement. Pre-call estimation prevents unnecessary API charges; post-call reconciliation ensures accurate accounting.

4.3 Loop Detection

The loop detector maintains a sliding window of recent tool invocations, recorded as (*tool_name*, *argument_hash*, *timestamp*) tuples. When a new invocation matches existing entries (same tool, similar arguments) and the match count exceeds threshold N within time window W, the circuit breaker trips immediately—even if budget remains. Default parameters: $N = 10$ calls, $W = 60$ seconds. This catches pathological retry loops before they exhaust the budget.

4.4 Nested Budgets

For multi-agent systems, parent sessions allocate sub-budgets to child tasks. A parent with a $10 budget may create a child with a $2 limit. Child costs roll up to the parent automatically. This enables hierarchical budget management in frameworks like CrewAI and AutoGen where multiple agents collaborate.

4.5 Structured Cost Reports

Every session produces a JSON report containing: session identifier, total budget, total spent, remaining balance, per-model and per-tool breakdowns, session duration, termination reason (`null`, `budget_exhausted`, or `loop_detected`), and a chronological event timeline. Reports can be piped to observability platforms (Datadog, LangSmith), billing systems (Stripe, Orb), or logged for analytics.

5 Integration with x402 Payment Protocol

x402 is an open payment protocol developed by Coinbase [1] that enables AI agents to make autonomous stablecoin payments over HTTP. By leveraging the HTTP 402 "Payment Required" status code, x402 allows agents to pay for API access and services without accounts, API keys, or subscriptions. Payments settle in ∼200 ms on Base L2 with near-zero fees [1].

x402 and AgentBudget address orthogonal concerns: x402 provides the *payment rail* (how agents pay); AgentBudget provides the *spending guardrail* (how much agents can pay per session).

5.1 The Session-Level Budget Gap

The x402 protocol handles individual transactions precisely: a client requests a resource, receives HTTP 402 with payment requirements, signs a payment using EIP-712 [1], and the facilitator settles on-chain. However, the protocol intentionally does not address cumulative session-level spend.

Consider the x402 whitepaper's use case of a legal research agent paying $0.10 per court document [1]. Without session enforcement, such an agent could retrieve 10,000 documents ($1,000) if its reasoning demands exhaustive search. The facilitator verifies each payment correctly but has no visibility into cumulative cost. Budget enforcement is a client-side concern—and this separation is architecturally appropriate.

5.2 Pre-Authorization Architecture

In an integrated deployment, AgentBudget operates as a pre-authorization layer on the client side (Figure 3):

1. The agent issues an HTTP request to an x402-enabled service.
2. The service responds with HTTP 402 and payment requirements.
3. Before signing, AgentBudget checks the remaining session budget.

(4a) If within budget: payment is signed, settled on-chain, cost deducted from ledger.

(4b) If over budget: payment blocked before signing—no on-chain transaction initiated.

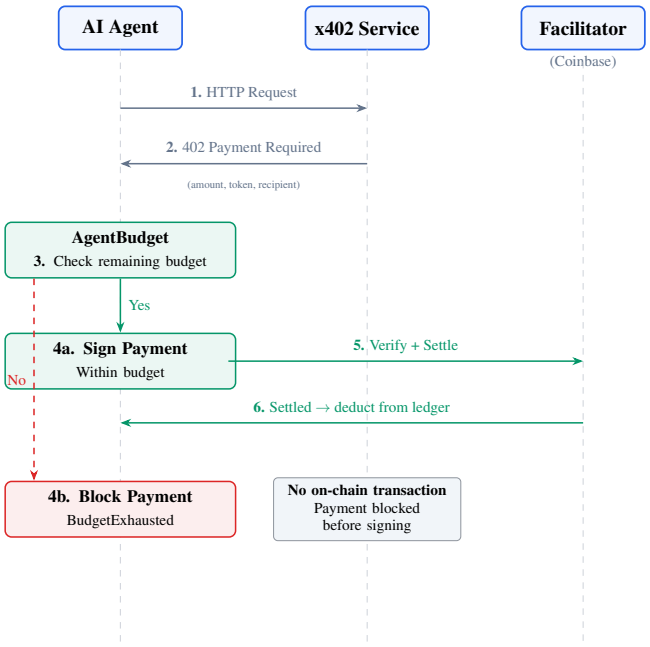

Figure 3: AgentBudget as pre-authorization layer in the x402 payment flow. Payments exceeding the session budget are blocked before signing, preventing on-chain settlement.

5.3 The *upto* Payment Scheme

The x402 specification describes two payment schemes [1]: *exact* (fixed amount per request, e.g., $0.10) and a proposed *upto* scheme (up to a maximum, where actual cost depends on resources consumed). The *exact* scheme is currently implemented; *upto* is proposed for future development.

The *upto* scheme is where AgentBudget provides the greatest value. When an agent is authorized to pay "up to $5.00" for variable-cost inference, the actual cost is unknown until response completion. AgentBudget's two-phase enforcement (Section 4.2) maps directly: estimate before signing, reconcile after settlement, block subsequent calls if over budget.

6 Evaluation

We evaluate AgentBudget along four axes: (1) feature comparison with existing solutions, (2) runtime overhead, (3) enforcement correctness, and (4) loop detection effectiveness. All benchmarks use AgentBudget v0.2.3 with live OpenAI API calls.

6.1 Feature Comparison

Table 2 compares AgentBudget against existing solutions. AgentBudget is the only solution combining dollar-denominated enforcement, per-session granularity, multi-provider coverage, real-time circuit breaking, and zero-infrastructure deployment in a single open-source package.

Table 2: Feature comparison of agent cost control solutions.

	Agent-Budget	**Lava** [7]	**Lang-Smith** [4]	**Lang-Chain** [5]
Granularity	Session	API key	Trace	Run
Cost unit	Dollars	Dollars	Display	Count
Scope	LLM+tools	LLM	LLM+tools	LLM
Enforcement	Real-time	Key-level	None	Count
Multi-provider	✓	Partial	✓	✓
Infrastructure	None	Cloud	Cloud	None
Integration	2 lines	Config	SDK+key	Callback
Open source	✓			Partial

6.2 Runtime Overhead

A critical requirement is that budget enforcement must not degrade agent performance. We measured AgentBudget's `track()` overhead over 1,000 consecutive invocations. Table 3 reports latency statistics.

Table 3: AgentBudget `track()` overhead (1,000 calls).

Metric	**Latency**
Median	3.5 μs
P95	6.3 μs
P99	13.8 μs
Max	433.0 μs

At a median of 3.5 μs per enforcement check, AgentBudget adds negligible overhead compared to typical LLM inference latency (200–1000 ms [2]). The overhead is approximately 0.001% of a typical call—effectively invisible to the agent.

6.3 Budget Enforcement Correctness

We tested hard-limit enforcement across five budget levels with $0.01 per-call costs. Table 4 shows that all budgets were enforced with zero overshoot and floating-point-precise ledger accounting.

Table 4: Budget enforcement across varying session budgets.

Budget	Calls allowed	Enforced	Overshoot
$0.01	1	✓	$0.00
$0.05	5	✓	$0.00
$0.10	10	✓	$0.00
$0.50	49	✓	$0.00
$1.00	99	✓	$0.00

6.4 Loop Detection

We evaluated the sliding-window loop detector (threshold $N=5$) across four scenarios. Table 5 summarizes results.

Table 5: Loop detection precision and recall ($N=5$).

Scenario	Expected	Result
15 unique tool calls	No trigger	0 FP
3 rotating tools, 15 calls	No trigger	0 FP
Same tool repeated	Detect at 6	Detected
10 unique then repeat	Detect repeat	Detected

The detector achieved 100% precision (zero false positives across diverse and rotating workloads) and 100% recall (caught both pure and mixed-context loops). Detection occurred at exactly $N+1$ calls, confirming deterministic threshold behavior.

6.5 Multi-Model Cost Attribution

To validate cost attribution, we ran a session mixing 4 gpt-4o-mini calls and 2 gpt-4o calls. The cost engine attributed $0.000056 to gpt-4o-mini and $0.0037 to gpt-4o—a 66× per-call cost difference, demonstrating why dollar-denominated tracking is essential over raw token counting. In a separate 7-step workflow combining 3 LLM calls ($0.000138) with 4 external API calls ($0.073), tools accounted for 99.8% of session cost—validating the unified ledger design.

6.6 Runaway Agent Case Study

We simulated a pathological agent entering an infinite retry loop. Without AgentBudget, the loop would execute indefinitely. With loop detection active, the circuit breaker halted execution after 11 calls at $0.000554. The session report recorded `terminated_by: "loop_detected"`, enabling automated alerting. At GPT-4o pricing, a 1,000-call undetected loop would cost $0.50 per session; at scale (100 concurrent sessions with 5% failure rate), this represents $2.50/hour in preventable waste.

7 Related Work

LLM Observability. LangSmith [4] and Langfuse [6] provide comprehensive tracing and cost analytics. These tools excel at post-hoc analysis but do not intervene in real time. AgentBudget is complementary: it enforces budgets and produces reports consumable by observability platforms.

API Spend Management. Lava AI Spend [7] provides per-API-key dollar caps. Key-level granularity cannot distinguish sessions sharing a key. AgentBudget operates at the session level.

Agent Frameworks. ReAct [11] established the reasoning-action loop now standard in agents. Frameworks like LangChain [5], CrewAI [12], and AutoGen [13] provide execution scaffolding but delegate cost management to operators. AgentBudget instruments these frameworks transparently via monkey-patching.

Agent Payment Protocols. x402 [1] and Google's A2A protocol [10] enable autonomous agent transactions but intentionally omit budget enforcement. AgentBudget serves as the enforcement layer for these protocols.

Resource Limiting. Unix `ulimit` [15] and Kubernetes resource quotas [16] prevent process-level resource exhaustion. AgentBudget applies the same principle to AI inference cost.

Circuit Breaking. The circuit breaker pattern [17] halts cascading failures by cutting off calls to failing services. AgentBudget applies circuit breaking to a new domain: financial cost.

8 Future Work

Several directions are planned: (1) A **real-time pricing API** maintaining costs for 500+ models. (2) Native **x402 payment interception** for pre-authorization of on-chain payments. (3) A **TypeScript/npm SDK** for JavaScript frameworks. (4) **Cost prediction** using historical data. (5) A **team budget dashboard** for aggregate visualization and allocation.

9 Conclusion

AI agents have introduced cost unpredictability into software systems for the first time. This fundamental property—flexible, goal-oriented resource decisions at runtime—is what makes agents powerful, but it necessitates a new primitive for cost control.

As agents gain financial autonomy through protocols like x402, budget enforcement becomes more acute. An agent making autonomous payments at machine speed without a spending cap is a systemic risk no production system should accept.

AgentBudget provides the missing primitive: per-session, dollar-denominated budget enforcement with real-time circuit breaking. It is lightweight (a library, not a platform), composable (structured data for downstream systems), and open (Apache 2.0, community-maintained). It complements observability tools by adding enforcement and payment protocols by adding budget control.

The agent economy requires both payment rails and spending guardrails. AgentBudget is the guardrail.

Availability. AgentBudget is open source and available at `https://github.com/sahiljagtap08/agentbudget`. Documentation: `https://agentbudget.dev`. Package:

https://pypi.org/project/agentbudget.

References

[1] E. Reppel, R. Caspers, K. Leffew, D. Organ, D. Kim, and N. Dalal. x402: An open standard for internet-native payments. Coinbase Developer Platform, May 2025. https://x402.org.

[2] OpenAI. API pricing. 2025. https://openai.com/api/pricing.

[3] Martian. The state of AI spending report. 2025.

[4] LangChain, Inc. LangSmith: LLM application observability platform. 2025. https://smith.langchain.com.

[5] LangChain, Inc. LangChain: Building applications with LLMs through composability. 2025. https://python.langchain.com.

[6] Langfuse GmbH. Langfuse: Open source LLM engineering platform. 2025. https://langfuse.com.

[7] Lava. Lava AI spend: API key spend limits for AI. 2025. https://uselava.com.

[8] Sentry. Python SDK: Automatic instrumentation. 2025. https://docs.sentry.io.

[9] Datadog, Inc. ddtrace: Datadog APM client. 2025. https://docs.datadoghq.com.

[10] Google. Agent-to-Agent (A2A) protocol. 2025. https://google.github.io/A2A.

[11] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. ReAct: Synergizing reasoning and acting in language models. In *Proc. ICLR*, 2023.

[12] J. Moura. CrewAI: Framework for orchestrating role-playing autonomous AI agents. 2024. https://crewai.com.

[13] Q. Wu, G. Banerjee, Y. Zhang, et al. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

[14] OpenAI. Function calling and other API updates. 2023. https://openai.com/index/function-calling-and-other-api-updates.

[15] M. Kerrisk. *The Linux Programming Interface*. No Starch Press, 2010.

[16] The Kubernetes Authors. Resource quotas. 2025. https://kubernetes.io/docs/concepts/policy/resource-quotas/.

[17] M. Nygard. *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2nd edition, 2018.

[18] Anthropic. Claude model pricing. 2025. https://www.anthropic.com/pricing.